CARP

D2.2.A: Requirements Analysis

Grant Agreement:	287767
Project Acronym:	CARP
Project Name:	Correct and Efficient Accelerator Programming
Instrument:	Small or medium scale focused research project (STREP)
Thematic Priority:	Alternative Paths to Components and Systems
Start Date:	1 December 2011
Duration:	36 months
Document Type[1]:	D (Deliverable)
Document Distribution[2]:	CO (Confidential)
Document Code[3]:	CARP-ARM-RP-001
Version:	v1.0
Editor (Partner):	A. Lokhmotov (ARM)
Contributors:	ARM, REAL, RIGHT
Workpackage(s):	WP2
Reviewer(s):	RWTHA
Due Date:	30 September 2012
Submission Date:	10 November 2012
Number of Pages:	258

[1]MD = management document; TR = technical report; D = deliverable; P = published paper; CD = communication/dissemination.
[2]PU = Public; PP = Restricted to other programme participants (including the Commission Services); RE = Restricted to a group specified by the consortium (including the Commission Services); CO = Confidential, only for members of the consortium (including the Commission Services).
[3]This code is constructed as described in the Project Handbook.

D2.2.A: Requirements Analysis

A. Kravets[1], **G. Kouveli**[1], **A. Lokhmotov**[1], **E. Hajiyev**[2], **L. Marák**[2], **T. Virolainen** [3]

[1]ARM, [2]REAL, [3]RIGHT

REVISION HISTORY

Date	Version	Author	Modification
2012-09-06	0.1	A. Kravets (ARM), G. Kouveli (ARM), A. Lokhmotov (ARM)	First version
2012-09-21	0.2	E. Hajiyev (REAL)	Changes from REAL
2012-10-03	0.3	T. Virolainen (RIGHT)	Changes from RIGHT
2012-10-08	0.4	E. Hajiyev (REAL)	Updates following comments from ARM
2012-10-31	0.5	A. Kravets (ARM), G. Kouveli (ARM)	Updates following comments from ICL
2012-11-08	0.6	T. Virolainen (RIGHT)	Updates following comments from ARM
2012-11-09	1.0	A. Kravets (ARM), G. Kouveli (ARM), A. Lokhmotov (ARM)	First version sent to the European Commission

APPROVALS

Role	Name	Partner	Date
Workpackage Leader	A. Lokhmotov	ARM	
Project Manager	A.F. Donaldson	ICL	

Contents

1 Executive Summary **5**

2 Benchmark Analysis **6**
2.1 Definitions . . . 6
2.1.1 Statements . . . 6
2.1.2 Work-item identifiers . . . 6
2.1.3 Data structures . . . 7
2.1.4 Iteration mapping . . . 8
2.1.5 Memory access mapping . . . 8
2.1.6 Dependences . . . 9
2.1.7 Code abstractions . . . 10
2.2 SHOC Level 1 . . . 12
2.2.1 Triad . . . 12
2.2.2 Stencil2D . . . 17
2.2.3 Reduction . . . 27
2.2.4 Scan . . . 32
2.2.5 Sort . . . 43
2.2.6 MD . . . 55
2.2.7 SGEMM . . . 60
2.2.8 Sparse matrix-vector multiplication . . . 70
2.3 Rodinia . . . 81
2.3.1 Back Propagation . . . 81
2.3.2 Breadth-First Search . . . 93
2.3.3 CFD Solver . . . 101
2.3.4 Gaussian Elimination . . . 112
2.3.5 Heart Wall . . . 120
2.3.6 Hot Spot . . . 123
2.3.7 K-means . . . 130
2.3.8 LavaMD . . . 140
2.3.9 Leukocyte Tracking . . . 148
2.3.10 LU Decomposition . . . 158
2.3.11 k-Nearest Neighbors . . . 170
2.3.12 Needleman-Wunsch . . . 176
2.3.13 Particle Filter . . . 188
2.3.14 PathFinder . . . 203
2.3.15 Speckle Reducing Anisotropic Diffusion . . . 209
2.3.16 Stream Cluster . . . 225
2.4 Parboil . . . 230
2.4.1 Breadth-First Search . . . 230
2.4.2 SGEMM . . . 230
2.4.3 Sparse Matrix-Vector Multiplication . . . 230
2.4.4 Stencil . . . 230
2.4.5 Other benchmarks . . . 231

2.5 RealEyes 232
2.5.1 Arithmetic Operations 232
2.5.2 GEMM – General Matrix Multiply 235
2.5.3 Image Statistics 239
2.5.4 Image Normalization 243
2.5.5 Image (Data) Repacking 244
2.6 BasemarkCL 246
2.6.1 Histogram equalization 246
2.6.2 SPH fluid 250

1 Executive Summary

Software for accelerated systems is predominantly written using low-level APIs, such as OpenCL and CUDA. Often developers compromise on performance portability (by tuning only to a specific architecture *e.g.* NVIDIA's Fermi) and correctness portability (by relying on certain architectural features *e.g.* warp-based execution). We aim to address the performance portability problem by designing PENCIL, a platform-neutral compute intermediate language. PENCIL will be used as both a target language for compilers from domain-specific languages (DSLs) and an efficiency-layer language for expert programmers, in particular, library implementers. Advanced code generation techniques based on the polyhedral model will translate PENCIL code into efficient platform-specific code.

To gather design requirements for PENCIL, we have analyzed two prominent GPGPU benchmark suites, SHOC [8](§2.2) and Rodinia [5, 6](§2.3), as well as validation cases from Realeyes (§2.5). We focused our analysis on two levels: a high-level description of a benchmark (a mathematical formulation of the problem, abstract data structures, iteration domains, dependences) and low-level implementation details (concrete data structures, partitioned iteration domains, memory access patterns, target-specific optimisations).

The high-level description may only specify the desired result, not the algorithm, so in general is unsuitable for representing accelerator workloads. On the other hand, the low-level description may be too implementation-specific. Therefore, we aim to design PENCIL to be somewhere in between these two levels.

Our analysis is the largest effort to date evaluating access/execute specifications [12]. The execute specifications describe the iteration spaces, dependences and partitioning, while the access specifications describe the memory locations accessed on each iteration. Given the emphasis on iteration space partitioning and data movement in accelerator programming, we believe incorporating such specifications is crucial to success of an accelerator programming model.

The detailed analysis also led to discovering several bugs in both Rodinia and SHOC.

2 Benchmark Analysis

We analyze several benchmark suites.

For each benchmark we describe the algorithm and data structures at both high level (mathematical formulation) and low level (OpenCL implementation). See §2.2.1 for an introductory example.

2.1 Definitions

2.1.1 Statements

Definition 1 *A statement is a sequence of operations under linear control flow that can be defined in terms of its side-effects.*

According to this definition, a maximal linear sequence of operations (*basic block*) can be considered a single statement and associated with a single iteration of an iteration space.

2.1.2 Work-item identifiers

For each dimension $i = 0, 1, 2$:

- Λ_i denotes the local work size, *i.e.* the number of unique local-ids in dimension i:

$$\Lambda_i = \texttt{get\_local\_size}(i) \tag{2.1}$$

- λ_i denotes the local-id in dimension i, $0 \leq \lambda_i < \Lambda_i$:

$$\lambda_i = \texttt{get\_local\_id}(i) \tag{2.2}$$

- N_i denotes the number of work-groups in dimension i:

$$N_i = \texttt{get\_num\_groups}(i) \tag{2.3}$$

- ν_i denotes the work-group-id in dimension i, $0 \leq \nu_i < N_i$:

$$\nu_i = \texttt{get\_group\_id}(i) \tag{2.4}$$

- Γ_i denotes the global work size, *i.e.* the number of unique global-ids in dimension i:

$$\Gamma_i = \texttt{get\_global\_size}(i) \tag{2.5}$$
$$\Gamma_i = N_i \Lambda_i \tag{2.6}$$

- Δ_i denotes the global-id offset in dimension i:

$$\Delta_i = \texttt{get\_global\_offset}(i) \tag{2.7}$$

- γ_i denotes the global-id in dimension i, $\Delta_i \leq \gamma_i < \Gamma_i + \Delta_i$:

$$\gamma_i = \texttt{get\_global\_id}(i) \tag{2.8}$$
$$\gamma_i = \Lambda_i \nu_i + \lambda_i + \Delta_i \tag{2.9}$$

2.1.3 Data structures

Abstract data structures

Range A monotonic sequence $\{\mathtt{R}_i\}$ of integer numbers denoted as `start : end : stride`, where

$$\mathtt{R}_i = \mathtt{start} + \mathtt{i} \times \mathtt{stride}, 0 \leq i < |\mathtt{R}| = 1 + (\mathtt{end} - \mathtt{start})/\mathtt{stride}$$

If `start` $\leq$ `end`, `start : end` is a shorthand for `start : end : 1`; otherwise, `start : end` is a shorthand for `start : end :` -1.

Vector An ordered collection of numbers. The i^{th} element of vector $\vec{A}$ is denoted as A_i, $0 \leq i < |\vec{A}|$, where $|\vec{A}|$ is the number of elements in vector $\vec{A}$ (the *size* of vector $\vec{A}$).

Slice For range $\mathtt{R} = \mathtt{start} : \mathtt{end} : \mathtt{stride}$ and vector $\vec{A}$, *slice* $\vec{A}_R$ denotes vector $\vec{A}'$ of $|\mathtt{R}|$ elements:

$$A'_i = A_{R_i}, 0 \leq i < |\mathtt{R}|$$

$A[\,]$ is a shorthand for $A[0 : N-1]$, where N is the size of $\vec{A}$.

Matrix A two-dimensional table of numbers:

$$M : \begin{pmatrix} M_{0,0} & M_{0,1} & \cdots & M_{0,n-1} \\ M_{1,0} & M_{1,1} & \cdots & M_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ M_{m-1,0} & M_{m-1,1} & \cdots & M_{m-1,n-1} \end{pmatrix}$$

where m is the number of rows and n is the number of columns in the table. The element in row i and column j is denoted as $M_{i,j}$.

Graph For a graph $G = (V, E)$ the following notation is used:

- V denotes the set of vertices.
- E denotes the set of edges.
- E^v denotes the set of edges connected to vertex $v \in V$.
- e_h denotes the source vertex (*head*) of edge $e \in E$.
- e_t denotes the sink vertex (*tail*) of edge $e \in E$.
- $|v| = |E^v|$ denotes the degree of vertex $v \in V$.

Concrete data structures

***N*-dimensional array** A systematic arrangement of data objects, each identified by N-element tuple of integer numbers (index). `A[i0][i1]..[iN]` denotes the element of array `A` with index `(i0,i1,..,iN)`.

Buffer A one-dimensional array. The only device data structure apart from **Image**.

Two-dimensional image (texture)

2.1.4 Iteration mapping

- I denotes the iteration space in high-level descriptions.
- P denotes the iteration space in low-level descriptions.
- $f : I \to P$ denotes the mapping from I to P.
- $f^{-1} : P \to I$ denotes the inverse mapping from P to I: $f^{-1} \circ f = \mathrm{id}_\mathrm{I}$.

2.1.5 Memory access mapping

In high-level descriptions:

- $M_r(i)$ denotes the set of uniform memory locations read on iteration $i \in I$.
- $M_w(i)$ denotes the set of uniform memory locations written on iteration $i \in I$.
- $M_{mr}(i)$ denotes the set of uniform memory locations possibly read on iteration $i \in I$.
- $M_{mw}(i)$ denotes the set of uniform memory locations possibly written on iteration $i \in I$.
- $F_r(i) = M_r(i) \cup M_{mr}(i)$
- $F_w(i) = M_w(i) \cup M_{mw}(i)$
- $F_{rw}(i) = F_r(i) \cup F_w(i)$

In low-level descriptions:

- $H_r(p)$ denotes the set of host memory locations read on iteration $p \in P$.
- $H_w(p)$ denotes the set of host memory locations written on iteration $p \in P$.
- $H_{mr}(p)$ denotes the set of host memory locations possibly read on iteration $p \in P$.
- $H_{mw}(p)$ denotes the set of host memory locations possibly written on iteration $p \in P$.
- $G_r(p)$ denotes the set of global memory locations read on iteration $p \in P$.
- $G_w(p)$ denotes the set of global memory locations written on iteration $p \in P$.
- $G_{mr}(p)$ denotes the set of global memory locations possibly read on iteration $p \in P$.
- $G_{mw}(p)$ denotes the set of global memory locations possibly written on iteration $p \in P$.
- $L_r(p)$ denotes the set of local memory locations read on iteration $p \in P$.
- $L_w(p)$ denotes the set of local memory locations written on iteration $p \in P$.
- $L_{mr}(p)$ denotes the set of local memory locations possibly read on iteration $p \in P$.
- $L_{mw}(p)$ denotes the set of local memory locations possibly written on iteration $p \in P$.

- $C_r(p)$ denotes the set of constant memory locations read on iteration $p \in P$.
- $C_{mr}(p)$ denotes the set of constant memory locations possibly read on iteration $p \in P$.
- $P_r(p)$ denotes the set of private memory locations read on iteration $p \in P$.
- $P_w(p)$ denotes the set of private memory locations written on iteration $p \in P$.
- $P_{mr}(p)$ denotes the set of private memory locations possibly read on iteration $p \in P$.
- $P_{mw}(p)$ denotes the set of private memory locations possibly written on iteration $p \in P$.

2.1.6 Dependences

The notation above can be used for a single statement S. For example, $M_r(\mathtt{S})$ denotes the set of uniform memory locations read by statement S.

Definition 2 (Memory-based data dependence)
We say that there is a memory-based data dependence from statement $\mathtt{S}_1$ to *statement* $\mathtt{S}_2$ *(equivalently,* $\mathtt{S}_2$ *depends on* $\mathtt{S}_1$*) and write* $\mathtt{S}_1 \dot{\rightarrow} \mathtt{S}_2$*, if and only if:*

1. *statement* $\mathtt{S}_1$ *is executed before statement* $\mathtt{S}_2$*,*
2. $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *access the same memory location and at least one of them writes into it:* $((F_r(\mathtt{S}_1) \cap F_w(\mathtt{S}_2)) \cup (F_w(\mathtt{S}_1) \cap F_r(\mathtt{S}_2)) \cup (F_w(\mathtt{S}_1) \cap F_w(\mathtt{S}_2))) \neq \emptyset$.

Dependences are traditionally classified according to the relative order of memory accesses.

Definition 3 (Classes of memory-based dependences)

True (or flow, or read-after-write) dependence: *Statement* $\mathtt{S}_1$ *writes into a memory location from which statement* $\mathtt{S}_2$ *later reads (denoted as* $\mathtt{S}_1 \xrightarrow{t} \mathtt{S}_2$*):*

```
S1:  A  = ...
S2: ... =  A
```

Anti (or write-after-read) dependence: *Statement* $\mathtt{S}_1$ *reads from a memory location into which statement* $\mathtt{S}_2$ *later writes (denoted as* $\mathtt{S}_1 \xrightarrow{a} \mathtt{S}_2$*):*

```
S1: ... =  A
S2:  A  = ...
```

Output (or write-after-write) dependence: *Both statements* $\mathtt{S}_1$ *and* $\mathtt{S}_2$ *write into the same memory location (denoted as* $\mathtt{S}_1 \xrightarrow{o} \mathtt{S}_2$*):*

```
S1:  A  = ...
S2:  A  = ...
```

If there exist a dependence between $\mathtt{S}_1$ on iteration $(i_0^1, \ldots, i_{k_1}^1) \in I_1$ and $\mathtt{S}_2$ on iteration $(i_0^2, \ldots, i_{k_2}^2) \in I_2$, we write:

$$I_1 : (i_0^1, \ldots, i_{k_1}^1) \rightarrow I_2 : (i_0^2, \ldots, i_{k_2}^2)$$

The set of all the dependences to statements in iteration space I is denoted as δ_I.

2.1.7 Code abstractions

Host code abstractions

- ALLOCATEHOSTMEMORY (`size`): Allocate host array of `size` elements.
- ALLOCATEBUFFER (`size`, `prop`[, `hptr`]): Allocate device buffer of `size` elements with access property `prop`, which can be one of the following:
  - `READ_ONLY`: the buffer can only be read on device.
  - `WRITE_ONLY`: the buffer can only be written on device.
  - `READ_WRITE`: the buffer can be read and written on device.

  Additional properties can also be specified, if `hptr` is given:
  - `USE_HOST_PTR`: The slice `hptr`$[0 : (\texttt{size} - 1)]$ should be used as buffer storage (default).
  - `COPY_HOST_PTR`: The slice `hptr`$[0 : (\texttt{size} - 1)]$ should be copied to created device buffer.

  This function corresponds to the `clCreateBuffer` OpenCL API function.
- ALLOCATEIMAGE2D (`size`, `prop`[, `hptr`]): Allocate two dimensional device image of `size` $= (\texttt{size1}, \texttt{size2})$ elements with access property `prop`. The set of properties is the same as for the ALLOCATEBUFFER, but READ_WRITE is not allowed. This function corresponds to the `clCreateImage2D` OpenCL API function.
- ALLOCATEIMAGE3D (`size`, `prop`[, `hptr`]): Allocate three dimensional device image of `size` $= (\texttt{size1}, \texttt{size2}, \texttt{size3})$ elements with access property `prop`. The set of properties is the same as for the ALLOCATEBUFFER, but READ_WRITE is not allowed. This function corresponds to the `clCreateImage3D` OpenCL API function.
- BUILDPROGRAM (`source`): Build program from `source`. This function corresponds to a call to the `clCreateProgramWithSource` followed by call to the `clBuildProgram` OpenCL API functions.
- BUILDKERNEL (`program`, `name`): Build kernel called `name` from `program` obtained from a BUILDPROGRAM call. This function corresponds to the `clCreateKernel` OpenCL API function.
- GETKERNELINFO (`kernel`, `prop`): For `kernel`, query the value of property `prop`, which can be:
  - `MAX_WORK_SIZE`: The maximum work-group size the given kernel can be launched with (*cf.* `CL_DEVICE_MAX_WORK_GROUP_SIZE`).
- COPYTODEVICE (`dst`, `src`, `blocking`, {`events`}): Copy host buffer `src` to device buffer `dst`. If Boolean `blocking` is `true`, the operation will be synchronous, *i.e.* the call will return after the copy is complete; otherwise, the call will return immediately without

waiting for the operation to complete. Wait for all events in the {events} set to complete before starting the copy. This function corresponds to the clEnqueueWriteBuffer OpenCL API function.[1]

- COPYTOHOST (src, dst, blocking, {events}): Copy device buffer src to host buffer dst. If Boolean blocking is true, the operation will be synchronous, *i.e.* the call will return after the copy is complete; otherwise, the call will return immediately without waiting for the operation to complete. Wait for all events in the {events} set to complete before starting the copy. This function corresponds to the clEnqueueReadBuffer OpenCL API function.

- COPYBUFFER (src, dst, {events}): Copy device buffer src to device buffer dst. Wait for all events in the {events} set to complete before starting the copy. This function corresponds to the clEnqueueCopyBuffer OpenCL API function.

- SETKERNELARGUMENTS (kernel, arg0, arg1, .., argN): Use arg0, arg1, ..argN as arguments for kernel. This function corresponds to a sequence of clSetKernelArgument OpenCL API calls (one for each argument).

- ENQUEUEKERNEL (kernel, gws, lws, {events}): Execute kernel on device, using gws as the global work size and lws as the local work size. Wait for all events in the {events} set to complete before launching kernel. If lws is **none**, the local work size is determined by the runtime. This function corresponds to the clEnqueueNDRangeKernel OpenCL API function.

Device code abstractions

- barrier({flags}): All work-items in a work-group must execute this function before any are allowed to continue execution. The {flags} argument can be a logical or of the two values:

  - LOCAL_MEM: Ensure correct ordering of operations to local memory.
  - GLOBAL_MEM: Ensure correct ordering of operations to global memory.

  This function corresponds to the OpenCL built-in synchronization function barrier.

[1] There is a difference in the semantics of a blocking call to this function and a blocking all to clEnqueueWriteBuffer. However, we have not encountered a case where the difference is significant for the host code abstractions provided here.

2.2 SHOC Level 1

For this document, we only analyze the Level 1 benchmarks. The Level 0 benchmarks are not representative of real workloads (benchmarking the OpenCL driver and platform). Due to time constraints, analysis of the Level 2 benchmark (S3D) and of the BFS and FFT Level 1 benchmarks is omitted. S3D is very complex (consisting of 27 kernels), but has regular control and memory accesses. The BFS implementation is based on a parallelization technique that is not valid for OpenCL, requiring synchronization between work-groups.[2] Another implementation of the BFS benchmark is described as part of the Rodinia suite. We believe an FFT implementation would be hard to derive automatically from a high-level description.

Caveat The SHOC level 1 benchmarks use in-order command queues, which ensures commands are completed in the order they are submitted. Therefore, the event-based synchronization in host code is redundant.

2.2.1 Triad

The Triad benchmark streams data from host main memory to device global memory and back: by design, there is no temporal data reuse.

High-level description

Abstract data structures $\vec{A}, \vec{B}$ and $\vec{C}$ are N-element vectors of floating point numbers.

Computation $\vec{C} = \vec{A} + s\vec{B}$

Iteration space The iteration space is one dimensional:

$$I = \{i : 0 \leq i < N\}$$

Dependences No dependences exist between iterations.

Memory access mapping On each iteration $i \in I$, one element from each of vectors $\vec{A}$ and $\vec{B}$ is read, and one element of vector $\vec{C}$ is written:

$$\begin{aligned} M_r(i) &= \{A_i, B_i\} \\ M_w(i) &= \{C_i\} \end{aligned}$$

Low-level implementation details

Device code The kernel computes one output element per work-item. See Algorithm 1.

[2] The SHOC implementers are working on a replacement version.

Algorithm 1 The Triad device code abstraction.

Input: `global float A[]`
Input: `global float B[]`
Input: `float s`
Output: `global float C[]`
1: **kernel** TRIAD (A, B, C, s)
2: $\quad C[\gamma] \leftarrow A[\gamma] + s \times B[\gamma]$
3: **end kernel**

Host code The one-dimensional iteration space I is partitioned into *blocks* of the same size `T`. The implementation is *double-buffered*: for each array two buffers are used. The device computation for one block is overlapped[3] with transferring the inputs for the next block from host and the outputs of the previous block to host.

The operation is repeated with the following block sizes: `blockSizes` $= \{2^{14}, 2^{15}, \ldots, 2^{22}\}$ (resulting in 64 KiB, 128 KiB, ..., 16 MiB for single-precision floating-point data).

Host data structures

- **`float`** `h[N]`: $A_x \rightarrow$ `h[x]`, $B_x \rightarrow$ `h[x]`, $C_x \rightarrow$ `h[x]`
- **`float`** `dA[2][N]`: $A_x \rightarrow$ `dA[(x/T)%2][x%T]`
- **`float`** `dB[2][N]`: $B_x \rightarrow$ `dB[(x/T)%2][x%T]`
- **`float`** `dC[2][N]`: $C_x \rightarrow$ `dC[(x/T)%2][x%T]`

Device data structures

- **`global float`** `A[N]`: $A_x \rightarrow$ `A[x%T]`
- **`global float`** `B[N]`: $B_x \rightarrow$ `B[x%T]`
- **`global float`** `C[N]`: $C_x \rightarrow$ `C[x%T]`

Input datasets The input data is random floating point numbers in the range $[0.0, 10.0)$.

Partitioned iteration space A given block size T results in the following iteration space:

$$P = \{(b, \gamma) : 0 \leq b < N/T, 0 \leq \gamma < \Gamma = T\}$$

The following function maps each iteration $i \in I$ to iteration $(b, \gamma) \in P$:

$$f : i \rightarrow (i/T, i\%T)$$

The following function gives the inverse mapping:

$$f^{-1} : (b, \gamma) \rightarrow T \times b + \gamma$$

[3] At least conceptually: see caveat in the beginning of §2.2.

Algorithm 2 The Triad host code abstraction.

```
1:  procedure ALLOCATEMEMORY()                          ▷ Allocate host memory and device buffers.
2:      h ← ALLOCATEHOSTMEMORY(N)
3:      dA[0] ← ALLOCATEBUFFER(N, READ_ONLY)
4:      dA[1] ← ALLOCATEBUFFER(N, READ_ONLY)
5:      dB[0] ← ALLOCATEBUFFER(N, READ_ONLY)
6:      dB[1] ← ALLOCATEBUFFER(N, READ_ONLY)
7:      dC[0] ← ALLOCATEBUFFER(N, WRITE_ONLY)
8:      dC[1] ← ALLOCATEBUFFER(N, WRITE_ONLY)
9:  end procedure
10: procedure INIT(h[0 : N − 1])                        ▷ Initialize host memory.
11:     for i in 0 : (N/2 − 1) do
12:         h[i] ← RAND(0.0,10.0)
13:         h[i + N/2] ← h[i]
14:     end for
15: end procedure
16: procedure VERIFY(h[0 : N − 1])                      ▷ Verify results.
17:     for i in 0 : (N/2 − 1) do
18:         if h[i] ≠ h[i + N/2] then
19:             ERROR( )
20:         end if
21:     end for
22: end procedure
23: procedure MAIN
24:     ALLOCATEMEMORY( )
25:     program ← BUILDPROGRAM("triad.cl")              ▷ See Algorithm 1.
26:     kernel[0] ← BUILDKERNEL(program,"triad")        ▷ Build two copies of
27:     kernel[1] ← BUILDKERNEL(program,"triad")        ▷ the triad kernel.
28:     maxLocalWorkSize ← GETKERNELINFO(MAX_WORK_SIZE)
29:     Λ ← MAX(128, maxLocalWorkSize)                  ▷ Local work size ≤ 128.
30:     for T in blockSizes do
31:         INIT(h[ ])
32:         Γ ← T                                       ▷ Global work size.
33:         toDeviceA[0] ← COPYTODEVICE(dA[0][0 : T − 1], h[0 : T − 1], false, ∅)
34:         toDeviceB[0] ← COPYTODEVICE(dB[0][0 : T − 1], h[0 : T − 1], false, ∅)
                                                        ▷ Start first block transfer to device.
35:         SETKERNELARGUMENTS(kernel[0], dA[0], dB[0], dC[0], 1.75f)
36:         nqKernel[0] ← ENQUEUEKERNEL(kernel[0], Γ, Λ, {toDeviceA[0], toDeviceB[0]})
                        ▷ Start first kernel execution, wait for writes to dA[0][ ] and dB[0][ ].
37:         if T < N then
38:             toDeviceA[1] ← COPYTODEVICE(dA[1][0 : T − 1], h[T : 2 × T − 1], false, ∅)
39:             toDeviceB[1] ← COPYTODEVICE(dB[1][0 : T − 1], h[T : 2 × T − 1], false, ∅)
                                                        ▷ Start second block transfer to device.
40:         end if
```

Algorithm 2 The Triad host code abstraction (continued).

```
41:        b ← 1                                                    ▷ Block index.
42:        i ← 0                                                    ▷ Host array index.
43:        while i < N do
44:            this ← b%2                                           ▷ 0 if b is even, 1 otherwise.
45:            that ← 1 − this                                      ▷ 1 if b is even, 0 otherwise.
46:            toHostC[that] ← CopyToHost(dC[that][0 : T − 1], h[i : i + T − 1], false, {nqKernel[that]})
                                                                    ▷ Start transfer of previous block from device.
47:            i ← i + T                                            ▷ Increase index.
48:            if i < N then
49:                SetKernelArguments(kernel[this], dA[this], dB[this], dC[this], 1.75f)
50:                eventsToWaitOn ← {toDeviceA[this], toDeviceB[this]}
51:                if b > 1 then
52:                    eventsToWaitOn ← eventsToWaitOn ∪ {toHostC[this]}
53:                end if
54:                nqKernel[this] ← EnqueueKernel(kernel[this], Γ, Λ, eventsToWaitOn)
55:            end if
56:            if i + T < N then
57:                toDeviceA[that] ← CopyToDevice(dA[that][0 : T − 1], h[i + T : i + 2 × T − 1], false, {nqKernel[that]})
58:                toDeviceB[that] ← CopyToDevice(dB[that][0 : T − 1], h[i + T : i + 2 × T − 1], false, {nqKernel[that]})
                                                                    ▷ Start transfer of next block to device.
59:            end if
60:            b ← b + 1                                            ▷ Increase block index.
61:        end while
62:        WaitForEvents({toHostC[that]})                           ▷ Wait for last transfer from device.
63:        Verify(h[ ])                                             ▷ Verify results.
64:    end for
65: end procedure
```

Global memory access mapping Each work-item performs two reads and one write:

$$\begin{aligned} G_r(\gamma) &= \{\texttt{A}[\gamma], \texttt{B}[\gamma]\} \\ G_w(\gamma) &= \{\texttt{C}[\gamma]\} \end{aligned}$$

Performance metrics

1. Speed (*GFLOP/s*): the total number of floating point operations ($2N$) divided by the total time for all data transfers plus calculations.
2. Bandwidth (*GB/s*): the total number of memory operations ($2N$ reads and N writes) divided by the total time for all data transfers plus calculations.

Data transfers between host and device are expected to dominate the execution time, and better performance is expected for larger block sizes.

Verification mechanism The two halves of the input buffers are initialized with the same data, so the two halves of the output buffer are expected to contain the same results. This is checked after the operation.

Target-specific optimisations applied Allocating memory for the host buffer can be done either using the C++ `new` operator (or equivalent methods *e.g.* `malloc`), or using `clCreateBuffer` with the flag `ALLOC_HOST_PTR`. This benchmark uses the latter method, which is good practice for NVIDIA cards because the allocated pages get pinned to main memory, enabling copies to device global memory to use DMAs instead of keeping the host busy.

Target-specific optimization opportunities Kernel code can be easily vectorized, which would improve the performance of this benchmark on vector-based architectures.

On unified memory systems, the operations for reading and writing the buffers would result in unnecessary memory copies, which would give misleading memory bandwidth figures. On such systems, it would be beneficial to replace calls to `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` with calls to `clEnqueueMapBuffer`.

2.2.2 Stencil2D

High-level description

The benchmark performs a 9-point two-dimensional (2D) stencil on floating point data.

Typically, the computation is repeated until convergence is achieved, which implies a variable number of iterations depending on the initial conditions. However, in this benchmark, the computation is repeated only for a fixed number of time steps T.

Abstract data structures The benchmark operates on a three-dimensional array A of floating point values, where the first dimension t corresponds to the time step and the other two dimensions (i, j) correspond to the position in a two dimensional grid of size $(R+2) \times (C+2)$.

Computation The following computation takes place at every time step t for each inner element of array A:

$$\begin{aligned} A_{t,i,j} &= w_0 \times A_{t-1,i,j} + \\ &\quad w_c \times (A_{t-1,i-1,j} + A_{t-1,i,j-1} + A_{t-1,i+1,j} + A_{t-1,i,j+1}) + \\ &\quad w_d \times (A_{t-1,i-1,j-1} + A_{t-1,i-1,j+1} + A_{t-1,i+1,j-1} + A_{t-1,i+1,j+1}) \end{aligned}$$

Iteration space

$$I = \{(t,i,j) : 0 \le t < T, 1 \le i \le R+1, 1 \le j \le C+1\}$$

Dependences

$$\begin{aligned} \delta_I = {} & I:(t-1,i-1,j-1) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i-1,j) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i-1,j+1) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i,j-1) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i,j) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i,j+1) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i+1,j-1) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i+1,j) \xrightarrow{t} I:(t,i,j) \wedge \\ & I:(t-1,i+1,j+1) \xrightarrow{t} I:(t,i,j) \end{aligned}$$

Memory access mapping On each iteration, there are nine reads and one write:

$$\begin{aligned} M_r(t,i,j) &= \{A_{t-1,i,j}, A_{t-1,i-1,j}, A_{t-1,i,j-1}, A_{t-1,i+1,j}, A_{t-1,i,j+1}, \\ &\quad A_{t-1,i-1,j-1}, A_{t-1,i-1,j+1}, A_{t-1,i+1,j-1}, A_{t-1,i+1,j+1}\} \\ M_w(t,i,j) &= \{A_{t,i,j}\} \end{aligned}$$

Low-level implementation details

The benchmark operates on floating point data of type `TYPE`, where `TYPE` is `float` for single precision and `double` for double precision.

Device code There are two kernels in this benchmark:

- `CopyRect()` This kernel is similar in functionality to the `clEnqueueCopyBufferRect` API call of the OpenCL 1.1 specification, implementations of which were not widely available when the benchmark was written. It copies the left and right halo columns from one device buffer to another. See Algorithm 3.
- `Stencil()` Each kernel launch performs one time step. See Algorithm 4.

Algorithm 3 The Stencil2D device code abstraction (`CopyRect`).

Output: `global TYPE dst[]`
Input: `int doffset` ▷ Offset for `dst`.
Input: `int dpitch` ▷ Pitch for `dst`.
Input: `global TYPE src[]`
Input: `int soffset` ▷ Offset for `src`.
Input: `int spitch` ▷ Pitch for `src`.
Input: `int width` ▷ Number of columns to copy.
Input: `int height` ▷ Number of rows to copy.

1: **kernel** CopyRect (`dst, doffset, dpitch, src, soffset, spitch, width, height`)
2: `r` $\leftarrow v_0 \times \Lambda_0 + \lambda_0$ ▷ `r` $= \gamma_0$
3: **if** `r` $<$ `height` **then**
4: **for** `c` **in** $0 : (\texttt{width} - 1)$ **do**
5: `dst`$[\texttt{r} \times \texttt{dpitch} + \texttt{doffset} + \texttt{c}] \leftarrow$ `src`$[\texttt{r} \times \texttt{spitch} + \texttt{soffset} + \texttt{c}]$
6: **end for**
7: **end if**
8: **end kernel**

Host code See Algorithm 5.

Host data structures Conceptually, the benchmark operates on a three-dimensional array, the first dimension being time. However, keeping all the intermediate results is unnecessary. Instead, the computation can be done by alternating between two arrays, one for the current time step and one for the previous time step.

- **TYPE** `data[Rp * Cp]`: $A_{0,x,y} \rightarrow$ `data[x * Cp + y]`, $A_{T-1,x,y} \rightarrow$ `data[x * Cp + y]`
- **TYPE** `buffer0[Rp * Cp]`: $A_{t,x,y} \rightarrow$ `buffer0[x * Cp + y]`, if t is even
- **TYPE** `buffer1[Rp * Cp]`: $A_{t,x,y} \rightarrow$ `buffer1[x * Cp + y]`, if t is odd

Device data structures For the `Stencil()` kernel:

- **global TYPE** `din[Rp * Cp]`: $A_{t,x,y} \rightarrow$ `din[x * Cp + y]`
- **global TYPE** `dout[Rp * Cp]`: $A_{t,x,y} \rightarrow$ `dout[x * Cp + y]`

Algorithm 4 The Stencil2D device code abstraction (`Stencil`).

Input: `int row` ▷ Row index, including halo.
Input: `int col` ▷ Column index, including halo.
Input: `int rowPitch` ▷ Row pitch, including halo.
1: **function** GETOFFSET(`row`, `col`, `rowPitch`)
2: **return** `row` $\times$ `rowPitch` $+$ `col`
3: **end function**

Input: `global TYPE din[]`
Input: `int alignment` ▷ Alignment required for rows (determines padding).
Input: `TYPE w0, wc, wd` ▷ Weights for stencil operation.
Input: `int LR` ▷ Number of rows per work-item.
Output: `global TYPE dout[]`
Local: `local TYPE ld[LRp` $\times$ `LCp]` ▷ $\mathtt{LRp} = \mathtt{LR}+2, \mathtt{LCp} = \mathtt{LC}+2 = \Lambda_1+2$
4: **kernel** STENCIL (`din`, `dout`, `alignment`, `w0`, `wc`, `wd`)
5: $\mathtt{r} \leftarrow \nu_0 \times \mathtt{LR}$ ▷ Row in the original grid, $\Lambda_0 = 1$.
6: $\mathtt{c} \leftarrow \nu_1 \times \Lambda_1 + \lambda_1$ ▷ Column in the original grid.
7: $\mathtt{C} \leftarrow N_1 \times \Lambda_1 + 2$ ▷ Total columns including halo.
8: $\mathtt{C_p} \leftarrow \mathtt{C}$
9: **if** $\neg\, C_p \vdots \mathtt{alignment}$ **then** ▷ Misaligned pitch.
10: $\mathtt{C_p} \leftarrow \lceil C_p/\mathtt{alignment} \rceil \times \mathtt{alignment}$ ▷ Next aligned pitch.
11: **end if**
12: **for** `lr` **in** $0 : (\mathtt{LR}+1)$ **do** ▷ Copy data in global memory to local memory (excluding left and right halo).
13: $\mathtt{lidx} \leftarrow$ GETOFFSET$(\lambda_0 + \mathtt{lr}, \lambda_1 + 1, \Lambda_1 + 2)$
14: $\mathtt{gidx} \leftarrow$ GETOFFSET$(\mathtt{r} + \mathtt{lr}, \mathtt{c} + 1, C_p)$
15: $\mathtt{ld}[\mathtt{lidx}] \leftarrow \mathtt{din}[\mathtt{gidx}]$
16: **end for**
17: **if** $\lambda_1 = 0$ **then** ▷ Copy left halo to local memory.
18: **for** `lr` **in** $0 : (\mathtt{LR}+1)$ **do**
19: $\mathtt{lidx} \leftarrow$ GETOFFSET$(\lambda_0 + \mathtt{lr}, \lambda_1, \Lambda_1 + 2)$
20: $\mathtt{gidx} \leftarrow$ GETOFFSET$(\mathtt{r} + \mathtt{lr}, \mathtt{c}, C_p)$
21: $\mathtt{ld}[\mathtt{lidx}] \leftarrow \mathtt{din}[\mathtt{gidx}]$
22: **end for**
23: **end if**
24: **if** $\lambda_1 = \Lambda_1 - 1$ **then** ▷ Copy right halo to local memory.
25: **for** `lr` **in** $0 : (\mathtt{LR}+1)$ **do**
26: $\mathtt{lidx} \leftarrow$ GETOFFSET$(\lambda_0 + \mathtt{lr}, \lambda_1 + 2, \Lambda_1 + 2)$
27: $\mathtt{gidx} \leftarrow$ GETOFFSET$(\mathtt{r} + \mathtt{lr}, \mathtt{c} + 2, C_p)$
28: $\mathtt{ld}[\mathtt{lidx}] \leftarrow \mathtt{din}[\mathtt{gidx}]$
29: **end for**
30: **end if**
31: BARRIER(`CLK_LOCAL_MEM_FENCE`) ▷ Wait until transfers are finished.

Algorithm 4 The Stencil2D device code abstraction (`Stencil`) (continued).

32: **for** `lr` **in** $0 : (\texttt{LR} - 1)$ **do** ▷ Stencil operation.
33: `cidx` $\leftarrow$ GETOFFSET($\lambda_0 + 1 + \texttt{lr}, \lambda_1 + 1, \Lambda_1 + 2$)
34: `nidx` $\leftarrow$ GETOFFSET($\lambda_0 + \texttt{lr}, \lambda_1 + 1, \Lambda_1 + 2$)
35: `sidx` $\leftarrow$ GETOFFSET($\lambda_0 + 2 + \texttt{lr}, \lambda_1 + 1, \Lambda_1 + 2$)
36: `eidx` $\leftarrow$ GETOFFSET($\lambda_0 + 1 + \texttt{lr}, \lambda_1 + 2, \Lambda_1 + 2$)
37: `widx` $\leftarrow$ GETOFFSET($\lambda_0 + +1\texttt{lr}, \lambda_1, \Lambda_1 + 2$)
38: `neidx` $\leftarrow$ GETOFFSET($\lambda_0 + \texttt{lr}, \lambda_1 + 2, \Lambda_1 + 2$)
39: `seidx` $\leftarrow$ GETOFFSET($\lambda_0 + 2 + \texttt{lr}, \lambda_1 + 2, \Lambda_1 + 2$)
40: `nwidx` $\leftarrow$ GETOFFSET($\lambda_0 + \texttt{lr}, \lambda_1, \Lambda_1 + 2$)
41: `swidx` $\leftarrow$ GETOFFSET($\lambda_0 + 2 + \texttt{lr}, \lambda_1, \Lambda_1 + 2$)
42: `centerValue` $\leftarrow$ `ld`[`cidx`]
43: `cardinalValues` $\leftarrow$ `ld`[`nidx`] + `ld`[`sidx`] + `ld`[`eidx`] + `ld`[`widx`]
44: `diagonalValues` $\leftarrow$ `ld`[`neidx`] + `ld`[`seidx`] + `ld`[`nwidx`] + `ld`[`swidx`]
45: `dout`[GETOFFSET(`r` + 1 + `lr`, `c` + 1, C_p)] $\leftarrow$ `w0` $\times$ `centerValue` + `wc` $\times$ `cardinalValues` + `wd` $\times$ `diagonalValues`
46: **end for**
47: **end kernel**

Algorithm 5 The Stencil2D host code abstraction.

1: **procedure** INITIALIZEGRID(`M`$[R+2][C+2]$, `haloVal`) ▷ Initialize stencil grid.[a]
2: **for** `i` **in** $1 :$ `R` **do** ▷ Initialize grid, excluding halo.
3: **for** `j` **in** $1 :$ `C` **do**
4: `M`[`i`][`j`] $\leftarrow$RAND(0.0,1.0)
5: **end for**
6: **end for**
7: **for** `j` **in** $0 : (\texttt{C} + 1)$ **do** ▷ Initialize top and bottom halo rows.
8: `M`[0][`j`] $\leftarrow$ `haloVal`
9: `M`[`R` + 1][`j`] $\leftarrow$ `haloVal`
10: **end for**
11: **for** `i` **in** $1 :$ `R` **do** ▷ Initialize left and right halo columns.
12: `M`[`i`][0] $\leftarrow$ `haloVal`
13: `M`[`i`][`C` + 1] $\leftarrow$ `haloVal`
14: **end for**
15: **end procedure**

[a] When M is passed to this function, it is actually a flattened, one-dimensional array. Two-dimensional indexing is used for convenience.

Algorithm 5 The Stencil2D host code abstraction (continued).

```
16: procedure PERFORMSTENCIL(input[R+2][C+2], T, w0, wc, wd)
                                          ▷ Perform stencil computation on host.[a]
17:     tmp ← ALLOCATEMEMORY((R+2) × (C+2))
18:     for t in 0 : (T−1) do
19:         tmp[ ] ← input[ ]                                          ▷ Copy grid.
20:         for i in 1 : R do
21:             for j in 1 : C do
22:                 oldCenterValue ← tmp[i][j]
23:                 oldNSEWValues ← tmp[i − 1][j] + tmp[i + 1][j] + tmp[i][j − 1] +
    tmp[i][j+1]
24:                 oldDiagonalValues ← tmp[i−1][j−1]+tmp[i+1][j−1]+tmp[i−
    1][j+1]+tmp[i+1][j+1]
25:                 input[i][j] ← w0 × oldCenterValue + wc × oldNSEWValues + wd ×
    oldDiagonalValues
26:             end for
27:         end for
28:     end for
29: end procedure

30: procedure VALIDATERESULT(expected[0 : N × M − 1], actual[0 : N × M −
    1], relErrorThreshold)                                          ▷ Validate results.
31:     for i in 0 : (N−1) do
32:         for j in 0 : (M−1) do
33:             expVal ← expected[i × M + j]
34:             actualVal ← actual[i × M + j]
35:             delta ← |actualVal − expVal|
36:             if expVal ≠ 0 then
37:                 relError ← delta/expVal
38:             else
39:                 relError ← 0                                       ▷ Potential bug.
40:             end if
41:             if relError > relErrorThreshold then ERROR( )
42:             end if
43:         end for
44:     end for
45: end procedure
```

[a] When *input* is passed to this function, it is actually a flattened, one-dimensional array. Two-dimensional indexing is used for convenience.

Algorithm 5 Stencil2D : Host code abstraction (continued)

Input: `int R` ▷ Number of rows in the stencil grid.
Input: `int C` ▷ Number of columns in the stencil grid.
Input: `int T` ▷ Time steps.
Input: `float haloVal` ▷ Value for halo points of the grid.
Input: `TYPE w0` ▷ Weights for the stencil operation.
Input: `TYPE wc`
Input: `TYPE wd`
Input: `int LR` ▷ Number of rows per work-item.
Input: `int LC` ▷ Λ_1
46: `alignment` $\leftarrow 16$ ▷ Architecture-specific?
47: $\mathtt{R_p} \leftarrow \mathtt{R} + 2$
48: $\mathtt{C_p} \leftarrow \mathtt{C} + 2$
49: **if** $\neg\, C_p \vdots \mathtt{alignment}$ **then** ▷ Misaligned pitch.
50: $\quad \mathtt{C_p} \leftarrow \lceil C_p / \mathtt{alignment} \rceil \times \mathtt{alignment}$ ▷ Next aligned pitch.
51: **end if**
52: `expected` $\leftarrow$ AllocateMemory($\mathtt{R_p} \times \mathtt{C_p}$)
53: InitializeGrid(`expected`, `haloVal`)
54: PerformStencil(`expected`, `T`)
55: `actual` $\leftarrow$ AllocateMemory($\mathtt{R_p} \times \mathtt{C_p}$)
56: InitializeGrid(`actual`, `haloVal`)
57: `stencilProgram` $\leftarrow$ BuildProgram("`stencil2d.cl`")
58: `copyRectKernel` $\leftarrow$ BuildKernel(`stencilProgram`, "`CopyRect`")
59: `stencilKernel` $\leftarrow$ BuildKernel(`stencilProgram`, "`Stencil`")
60: `buffer0` $\leftarrow$ AllocateBuffer($\mathtt{R_p} \times \mathtt{C_p}$, `READ_WRITE`) ▷ Allocate device buffers.
61: `buffer1` $\leftarrow$ AllocateBuffer($\mathtt{R_p} \times \mathtt{C_p}$, `READ_WRITE`)
62: `in` $\leftarrow$ `buffer0` ▷ Pointers to the buffers.
63: `out` $\leftarrow$ `buffer1`

Algorithm 5 Stencil2D : Host code abstraction (continued)

writeEvent ← COPYTODEVICE(in, actual, **False**, ∅)
COPYBUFFER(in[0 : $C_p - 1$], out[0 : $C_p - 1$], {writeEvent}) ▷ Copy top and bottom halo rows.
COPYBUFFER(in[$(R_p - 1) \times C_p : R_p \times C_p - 1$], out[$(R + 1) \times C_p : R_p \times C_p - 1$], ∅) ▷ Synchronization here is problematic.
SETKERNELARGUMENTS(copyRectKernel, out, 0, C_p, in, 0, C_p, 1, R_p) ▷ Copy left and right halo columns.
cwEvent ← ENQUEUEKERNEL(copyRectKernel, R_p, **none**, ∅)
SETKERNELARGUMENTS(copyRectKernel, out, $C + 1$, C_p, in, $C + 1$, C_p, 1, R)
ceEvent ← ENQUEUEKERNEL(copyRectKernel, R_p, **none**, ∅)
WAITFOREVENTS({cwEvent, ceEvent})
prevKernelEvent ← ∅
for t **in** 1 : (T) **do**
    SETKERNELARGUMENTS(stencilKernel, in, out, alignment, w0, wc, wd)
    kernelEvent ← ENQUEUEKERNEL(stencilKernel, (R/LR, C), (1, LC), prevKernelEvent)
    prevKernelEvent ← {kernelEvent}
    if in = buffer0 **then** ▷ Swap buffers.
        in ← buffer1
        out ← buffer0
    else
        in ← buffer0
        out ← buffer1
    end if
end for
COPYTOHOST(in, actual, **True**, prevKernelEvent)
▷ Buffers have been swapped, so in is the buffer we should copy to host.
VALIDATERESULT(expected, actual)

- **`local TYPE ld[LRp * LCp]`:**
  $A_{t,x,y} \rightarrow$ `ld[(x - 1) % LR + 1][(y - 1) % LC + 1]`, if $1 \leq x \leq R \wedge 1 \leq y \leq C$
  The mapping of elements of $A[\,]$ to the halo elements of `ld[ ]` is omitted, as it is fairly complicated.[4]

Input datasets There are four classes for the sizes of the datasets on which the benchmark operates: classes 1, 2, 3 and 4 work on arrays of dimensions $512 \times 512, 1024 \times 1024, 2048 \times 2048$ and 4096×4096 respectively (the dimensions do not include the halo, for which additional memory is allocated). The computation is repeated for random single and double precision floating point inputs in the range $[0.0, 1.0)$.

Command line parameters The following command line parameters are available:

- `seed`: The seed for the random number generator.
- `haloVal`: The value used for the halo elements; the default is 0.
- `weight-center`, `weight-cardinal`, `weight-diagonal`: The weights used for stencil computation; the defaults are $(w_0, w_c, w_d) = (0.25, 0.15, 0.05)$.
- `lsize`: The parameters for partitioning the iteration space; the defaults are $(\texttt{LR}, \texttt{LC}) = (8, 256)$.
- `val-threshold`: The relative error threshold used for validation; the default is 0.01.

Partitioned iteration space The following partitioned iteration space is used:

$$\begin{aligned} P \quad = \{(t, v_0, v_1, \lambda_0, \lambda_1, l) : \quad & 0 \leq t < T, \\ & 0 \leq v_0 < N_0 = R/LR, \\ & 0 \leq v_1 < N_1 = C/LC, \\ & \lambda_0 = 0, 0 \leq \lambda_1 < \Lambda_1 = LC, \\ & 0 \leq l < LR\} \end{aligned}$$

which means the following hold:

$$\begin{aligned} (\Gamma_0, \Gamma_1) &= (R/LR, C) \\ (\gamma_0, \gamma_1) &= (v_0, v_1 \times LC + \lambda_1) \end{aligned}$$

Every iteration (t, i, j) of the original iteration space maps to this new iteration according to the following function:

$$f : (t, i, j) \rightarrow (t, i/LR, j/LC, 0, j\%LC, i\%LR)$$

The inverse mapping is given by the function:

$$f^{-1} : (t, v_0, v_1, \lambda_0, \lambda_1, l) \rightarrow (t, v_0 \times LR + l, v_1 \times LC + \lambda_1)$$

[4] There are eight different cases: four for the corner halo elements, one for each of the top and bottom halo rows and one for each of the left and right halo columns.

No special care (e.g. row padding, special handling of remaining elements) is taken for the case when C is not divided exactly by LC. Therefore we assume in this subsection that C is a multiple of LC.

For copying data between global and local memory, a slightly different iteration space is used:

$$\begin{aligned} P \quad = \{(t,v_0,v_1,\lambda_0,\lambda_1,l): \quad & 0 \le t < T, \\ & 0 \le v_0 < R/LR, \\ & 0 \le v_1 < C/LC, \\ & 0 \le \lambda_0 < 1, 0 \le \lambda_1 < LC, \\ & 0 \le l < LR+2\} \end{aligned}$$

Note that the only difference is two additional iterations for the innermost index.

Device memory access mapping Each work-item executing the `Stencil()` kernel performs the following reads and writes:

- For copying data between global and local memory:

$$\begin{aligned} G_r(t,v_0,v_1,\lambda_0,0,l) &= \{\texttt{din}[(N_0 \times LR + \lambda_0 + l) \times \texttt{C}_\texttt{p} + \gamma_1 + 1], \\ &\quad \texttt{din}[(N_0 \times LR + \lambda_0 + l) \times \texttt{C}_\texttt{p} + \gamma_1]\} \\ L_w(t,v_0,v_1,\lambda_0,0,l) &= \{\texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1 + 1], \\ &\quad \texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1]\} \\ G_r(t,v_0,v_1,\lambda_0,\Lambda_1 - 1,l) &= \{\texttt{din}[(N_0 \times LR + \lambda_0 + l) \times \texttt{C}_\texttt{p} + \gamma_1 + 1], \\ &\quad \texttt{din}[(N_0 \times LR + \lambda_0 + l) \times \texttt{C}_\texttt{p} + \gamma_1 + 2]\} \\ L_w(t,v_0,v_1,\lambda_0,\Lambda_1 - 1,l) &= \{\texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1 + 1], \\ &\quad \texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1 + 2]\} \\ G_r(t,v_0,v_1,\lambda_0,\lambda_1,l) &= \{\texttt{din}[(N_0 \times LR + \lambda_0 + l) \times \texttt{C}_\texttt{p} + \gamma_1 + 1]\} \\ L_w(t,v_0,v_1,\lambda_0,\lambda_1,l) &= \{\texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1 + 1]\} \end{aligned}$$

- For the actual stencil computation:

$$\begin{aligned} L_r(t,v_0,v_1,\lambda_0,\lambda_1,l) &= \{\texttt{ld}[(\lambda_0 + l + 1) \times (\Lambda_1 + 2) + \lambda_1 + 1], \\ &\quad \texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1 + 1], \\ &\quad \texttt{ld}[(\lambda_0 + l + 2) \times (\Lambda_1 + 2) + \lambda_1 + 1], \\ &\quad \texttt{ld}[(\lambda_0 + l + 1) \times (\Lambda_1 + 2) + \lambda_1 + 2], \\ &\quad \texttt{ld}[(\lambda_0 + l + 1) \times (\Lambda_1 + 2) + \lambda_1], \\ &\quad \texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1 + 2], \\ &\quad \texttt{ld}[(\lambda_0 + l + 2) \times (\Lambda_1 + 2) + \lambda_1 + 2], \\ &\quad \texttt{ld}[(\lambda_0 + l) \times (\Lambda_1 + 2) + \lambda_1], \\ &\quad \texttt{ld}[(\lambda_0 + l + 2) \times (\Lambda_1 + 2) + \lambda_1]\} \\ G_w(t,v_0,v_1,\lambda_0,\lambda_1,l) &= \{\texttt{dout}[(N_0 \times LR + \lambda_0 + l + 1) \times \texttt{C}_\texttt{p} + \gamma_1 + 1]\} \end{aligned}$$

Performance metrics Performance is evaluated based on a time measurement that includes the transfers of data between host and device, one execution of `CopyRect` and T executions of `Stencil`.

Validation mechanism The results from a run on device are compared again a reference run on host. Since the comparison is between floating point numbers, the relative error is compared against a threshold (`val-threshold`).

The validation mechanism contains a potential bug: when the expected value is zero, the relative error is also set to zero regardless of the actual value, missing potential errors.

Target-specific optimisations The kernel uses local memory. This increases performance when local memory is implemented on-chip (as on desktop GPUs). This, however, might decrease performance when local memory is in the same physical space as global memory, since it implies redundant memory copies and barriers.

Target-specific optimization opportunities Vectorization can potentially improve performance if the penalty for unaligned memory access is tolerable.

2.2.3 Reduction

This benchmark performs a reduction operation on a vector of floating point numbers.

High-level description

The reduction operation is defined as follows:

$$O = \sum_i I_i, 0 \leq i < N$$

Abstract data structures

- $\vec{I}$ is an input N-element vector.
- $\vec{R}$ is an intermediate B-element vector.
- O is an output value.

Computation Range $[0 : (N-1)]$ is partitioned into B disjoint ranges by a monotonically increasing sequence $\{i_k\} : 0 = i_0 < i_1 < \ldots < i_B = N$:

$$[0 : (N-1)] = \bigcup_{b=0}^{B-1} [i_b : (i_{b+1} - 1)]$$

Reduction (2.10) is performed in two steps:

- Reduction on blocks:

$$R_b = \sum_j I_j,\ 0 \leq b < B \wedge i_b \leq j < i_{b+1}$$

- Reduction of intermediate results:

$$O = \sum_b R_b, 0 \leq b < B$$

Reduction is a well-known parallel primitive, which is assumed to be associative and commutative (*i.e.* additions can be performed in any order). Since the iteration space, dependences and memory accesses depend on how this primitive is actually implemented, we only analyze the low-level implementation.

Low-level implementation details

The benchmark operates on floating point data of type `TYPE`, where `TYPE` is `float` for single precision and `double` for double precision.

The first step of the algorithm is performed on the device, whereas the second step is performed on the host.

Device code See Algorithm 6.

Algorithm 6 The Reduction device code abstraction.

Input: `global const TYPE I[N]`
Output: `global TYPE R[B]`
Local: `local TYPE lR[`Λ_0`]` ▷ Partial reduction results.

1: **kernel** REDUCE (I, R)

2: $\texttt{lR}[\lambda_0] \leftarrow 0$
3: **for** i **in** $[(2 \times \Lambda_0 \times \nu_0 + \lambda_0) : (N-1) : (2 \times \Lambda_0 \times N_0)]$ **do**
▷ Reduce multiple elements per work-item.
4: $\texttt{lR}[\lambda_0] \leftarrow \texttt{lR}[\lambda_0] + \texttt{I}[\texttt{i}] + \texttt{I}[\texttt{i} + \Lambda_0]$
5: **end for**
6: BARRIER(CLK_LOCAL_MEM_FENCE)

7: **for** $\texttt{s} \leftarrow \Lambda_0/2$ **until** $\texttt{s} \leq 0$ **step** $\texttt{s} \leftarrow \lfloor \texttt{s}/2 \rfloor$ **do** ▷ Tree-like reduction of partial results.
8: **if** $\lambda_0 < \texttt{s}$ **then**
9: $\texttt{lR}[\lambda_0] \leftarrow \texttt{lR}[\lambda_0] + \texttt{lR}[\lambda_0 + \texttt{s}]$
10: **end if**
11: BARRIER(CLK_LOCAL_MEM_FENCE)
12: **end for**

13: **if** $\lambda_0 = 0$ **then** ▷ Write result to global memory.
14: $\texttt{R}[\nu_0] \leftarrow \texttt{lR}[0]$
15: **end if**
16: **end kernel**

Algorithm 7 The Reduction host code abstraction.

```
1:  procedure ValidateResult(hI[N], hR[B])
2:      refSum ← 0
3:      for i in 0 : (N − 1) do
4:          refSum ← hI[i]
5:      end for
6:      devSum ← 0
7:      for i in 0 : (B − 1) do
8:          devSum ← hR[i]
9:      end for
10:     error ← |refSum − devSum|
11:     if error ≥ threshold then
12:         Error( )
13:     end if
14: end procedure

Input: N                                                    ▷ Input size.
15: hI ← AllocateMemory(N)
16: hR ← AllocateMemory(B)
17: for i in 0 : (N − 1) do
18:     hI[i] ← i%3
19: end for
20: numWorkGroups ← 64                                      ▷ numWorkGroups = B
21: localWorkSize ← 256
22: globalWorkSize ← numWorkGroups × localWorkSize
23: dI ← AllocateBuffer(N, READ_WRITE)                      ▷ Should be READ_ONLY.
24: dR ← AllocateBuffer(B, READ_WRITE)                      ▷ Should be WRITE_ONLY.
25: CopyToDevice(dI, hI, True, ∅)
26: reductionProgram ← BuildProgram("reduction.cl")
27: reductionKernel ← BuildKernel(reductionProgram, "reduce")
28: SetKernelArguments(reductionKernel, dI, dR)
29: EnqueueKernel(reductionKernel, globalWorkSize, localWorkSize, ∅)
30: CopyToHost(dR, hR, True, ∅)
31: ValidateResult(hI, hR)
```

Host code See Algorithm 7.

Host data structures

- **`TYPE`** `hI[N]`: $I_x \rightarrow$ `hI[x]`
- **`TYPE`** `hR[B]`: $R_x \rightarrow$ `hR[x]`

Device data structures

- **`global const TYPE`** `I[N]`: $I_x \rightarrow$ `I[x]`
- **`global TYPE`** `R[B]`: $R_x \rightarrow$ `R[x]`
- **`local TYPE`** `lR[`Λ_0`]`: intermediate results

Input datasets The input sizes considered are $2^{20}, 2^{23}, 2^{25}$ and 2^{26} bytes of data, which result in $N \in \{2^{18}, 2^{21}, 2^{23}, 2^{24}\}$ for single-precision and $N \in \{2^{17}, 2^{20}, 2^{22}, 2^{23}\}$ for double-precision.

The i^{th} element of the input array is initialized with the value $i\%3$, resulting in a pattern of $\{0,1,2,0,1,2,\ldots\}$.

Partitioned iteration space The algorithm (reduction) uses the following iteration spaces:

- Algorithm 6, line 2:

$$P_{R_1} = \{(\nu_0, \lambda_0) : 0 \leq \nu_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

- Algorithm 6, lines 3–5:

$$\begin{aligned} P_{R_2} = \{&(\nu_0, \lambda_0, i) : 0 \leq \nu_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256, \\ &i = 2 \times \Lambda_0 \times \nu_0 + k \times 2 \times \Lambda_0 \times N_0, 0 \leq k < (N - 2 \times \Lambda_0 \times \nu_0 - \lambda_0)/(2 \times \Lambda_0 \times N_0)\} \end{aligned}$$

- Algorithm 6, lines 7–12:

$$\begin{aligned} P_{R_3} = \{&(\nu_0, \lambda_0, s) : 0 \leq \nu_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256, \\ &s = \lfloor \Lambda_0 / 2^k \rfloor, k \in \{1, 2, \ldots, \log_2 \Lambda_0\}\} \end{aligned}$$

- Algorithm 6, lines 13–15:

$$P_{R_4} = \{(\nu_0, \lambda_0) : 0 \leq \nu_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

Device memory access mapping

- For P_{R_1}:

$$\begin{aligned} G_r(\nu_0, \lambda_0) &= \emptyset \\ G_w(\nu_0, \lambda_0) &= \emptyset \\ L_r(\nu_0, \lambda_0) &= \emptyset \\ L_w(\nu_0, \lambda_0) &= \{\texttt{lR}[\lambda_0]\} \end{aligned}$$

- For P_{R_2}:

$$\begin{aligned} G_r(v_0,\lambda_0,i) &= \{\texttt{I}[i],\texttt{I}[i+\Lambda_0]\} \\ G_w(v_0,\lambda_0,i) &= \emptyset \\ L_r(v_0,\lambda_0,i) &= \{\texttt{lR}[\lambda_0]\} \\ L_w(v_0,\lambda_0,i) &= \{\texttt{lR}[\lambda_0]\} \end{aligned}$$

- For P_{R_3}:

$$\begin{aligned} G_r(v_0,\lambda_0,s) &= \emptyset \\ G_w(v_0,\lambda_0,s) &= \emptyset \\ L_r(v_0,\lambda_0,s) &= \{\texttt{lR}[\lambda_0],\texttt{lR}[\lambda_0+s]\},\lambda_0<s \\ L_w(v_0,\lambda_0,s) &= \{\texttt{lR}[\lambda_0]\},\lambda_0<s \\ L_r(v_0,\lambda_0,s) &= \emptyset,\lambda_0\geq s \\ L_w(v_0,\lambda_0,s) &= \emptyset,\lambda_0\geq s \end{aligned}$$

- For P_{R_4}:

$$\begin{aligned} G_r(v_0,0) &= \emptyset \\ G_w(v_0,0) &= \{\texttt{R}[v_0]\} \\ L_r(v_0,0) &= \{\texttt{lR}[0]\} \\ L_w(v_0,0) &= \emptyset \\ G_r(v_0,\lambda_0) &= \emptyset,\lambda_0\neq 0 \\ G_w(v_0,\lambda_0) &= \emptyset,\lambda_0\neq 0 \\ L_r(v_0,\lambda_0) &= \emptyset,\lambda_0\neq 0 \\ L_w(v_0,\lambda_0) &= \emptyset,\lambda_0\neq 0 \end{aligned}$$

Performance metrics

- Bandwidth (*GB/s*) based on the execution time: the total size of all the memory operations divided by the total kernel execution time.
- Bandwidth (*GB/s*) based on total time: the total size of all the memory operations divided by the total time for all data transfers plus calculations.
- Parity: the transfer (upload and download) time divided by the kernel execution time.

Validation mechanism The results of the computation are verified against the results of the same operation performed on the host.

Target-specific optimisations The global and local work sizes are independent of the problem size and hardcoded in the benchmark code, which suggests their choice is target-specific.

Work items within a work-group with consecutive local ids access consecutive memory locations, which improves the memory system performance (caching, coalescing).

Target-specific optimization opportunities For a vector target architecture, vectorization of the additions in the device code would significantly improve performance.

2.2.4 Scan

This benchmark performs a scan (prefix sum) operation on a vector of floating point numbers.

High-level description

The scan operation is defined as follows:

$$O_i = \sum_{j=0}^{i} I_j, 0 \leq i < N \tag{2.10}$$

Abstract data structures

- $\vec{I}$ is an input N-element vector.
- $\vec{O}$ is an output N-element vector.
- $\vec{R}$ is an intermediate B-element vector.
- $\vec{T}$ is an intermediate B-element vector.

Computation Range $[0 : (N-1)]$ is partitioned into B disjoint ranges by a monotonically increasing sequence $\{i_k\} : 0 = i_0 < i_1 < \ldots < i_B = N$:

$$[0 : (N-1)] = \bigcup_{b=0}^{B-1} [i_b : (i_{b+1} - 1)]$$

Scan (2.10) is performed in three steps:

- Segmented Reduction:

$$R_b = \sum_j I_j,\ 0 \leq b < B \wedge i_b \leq j < i_{b+1}$$

- Top Scan:

$$T_0 = 0,\ T_b = \sum_k R_k,\ 0 \leq k < b < B$$

- Bottom Scan:

$$O_i = T_b + \sum_j I_j,\ 0 \leq b < B \ \wedge \ i_b \leq j \leq i < i_{b+1}$$

Reductions and scans are well-known parallel primitives, which are assumed to be associative and commutative (*i.e.* additions can be performed in any order). Since the iteration space, dependences and memory accesses depend on how these primitives are actually implemented, we only analyze the low-level implementation.

Low-level implementation details

The benchmark operates on floating point data of type **`TYPE`** and **`VECTYPE`**, where **`TYPE`** is **`float`** and **`VECTYPE`** is **`float4`** for single precision, **`TYPE`** is **`double`** and **`VECTYPE`** is **`double4`** for double precision.

Device code Each step of the high-level algorithm maps to one device kernel:

- Segmented Reduction: Algorithm 8.
- Top Scan: Algorithm 10 (calls Algorithm 9).
- Bottom Scan: Algorithm 11 (calls Algorithm 9).

Host code See Algorithm 12.

Host data structures

- **`TYPE`** `hI[N]`: $I_x \rightarrow$ `hI[x]`
- **`TYPE`** `hO[N]`: $O_x \rightarrow$ `hO[x]`
- **`TYPE`** `dI[N]`: $I_x \rightarrow$ `dI[x]`
- **`TYPE`** `dO[N]`: $O_x \rightarrow$ `dO[x]`
- **`TYPE`** `dR[B]`: $R_x \rightarrow$ `dR[x]`, $T_x \rightarrow$ `dR[x]`

Device data structures

- Segmented Reduction:
  - **`global const TYPE`** `I[N]`: $I_x \rightarrow$ `I[x]`
  - **`global TYPE`** `R[B]`: $R_x \rightarrow$ `R[x]`
  - **`local TYPE`** `lR[`Λ_0`]`: intermediate results
- Top Scan:
  - **`global TYPE`** `R[B]`: $R_x \rightarrow$ `R[x]`, $T_x \rightarrow$ `R[x]`
  - **`local TYPE`** `lS[`$2 \times \Lambda_0$`]`: intermediate results.
- Bottom Scan:
  - **`global const TYPE`** `I[N]`: $I_x \rightarrow$ `I[x]`
  - **`global const TYPE`** `T[B]`: $T_x \rightarrow$ `T[x]`
  - **`global TYPE`** `O[N]`: $O_x \rightarrow$ `O[x]`
  - **`local TYPE`** `lS[`$2 \times \Lambda_0$`]`: intermediate results
  - **`local TYPE`** `localSeed`: intermediate result

Algorithm 8 The Scan (Segmented Reduction) device code abstraction.

Input: `global const TYPE I[N]`
Output: `global TYPE R[B]`
Local: `local TYPE lR[`Λ_0`]` ▷ Partial reduction results.

1: **kernel** REDUCE (I,R) ▷ $N_0 = B,\ v_0 = b$
2: `blockSize` $\leftarrow \lfloor \lfloor N/4 \rfloor / N_0 \rfloor \times 4$ ▷ Alignment of 4 elements.
▷ All work-groups but the last work on `blockSize` elements.
3: `blockStart` $\leftarrow v_0 \times$ `blockSize` ▷ `blockStart` $= i_{v_0}$
4: **if** $v_0 = N_0 - 1$ **then** ▷ Last work-group gets remaining elements.
5: `blockEnd` $\leftarrow N$ ▷ `blockEnd` $= i_N$
6: **else**
7: `blockEnd` $\leftarrow$ `blockStart` $+$ `blockSize` ▷ `blockEnd` $= i_{v_0+1}$
8: **end if**
9: `sum` $\leftarrow 0$

10: **for** `i` **in** $[(\texttt{blockStart} + \lambda_0) : (\texttt{blockEnd} - 1) : \Lambda_0]$ **do** ▷ Reduce multiple elements per work-item.
11: `sum` $\leftarrow$ `sum` $+$ `I`$[$`i`$]$
12: **end for**

13: `lR`$[\lambda_0] \leftarrow$ `sum` ▷ Store partial result into local memory.
14: BARRIER(CLK_LOCAL_MEM_FENCE)

15: **for** `s` $\leftarrow \Lambda_0/2$ *ensuremath***to** `s` > 0 **step** `s` $\leftarrow \lfloor$ `s`$/2 \rfloor$ **do** ▷ Tree-like reduction of partial results.
16: **if** $\lambda_0 <$ `s` **then**
17: `lR`$[\lambda_0] \leftarrow$ `lR`$[\lambda_0] +$ `lR`$[\lambda_0 +$ `s`$]$
18: **end if**
19: BARRIER(CLK_LOCAL_MEM_FENCE)
20: **end for**

21: **if** $\lambda_0 = 0$ **then** ▷ Write result to global memory.
22: `R`$[v_0] \leftarrow$ `lR`$[0]$
23: **end if**
24: **end kernel**

Algorithm 9 The Scan (`scanLocalMem`) device code abstraction.

▷ Each work-item λ_0 in a work-group calls this function with its value `v`. The function performs a parallel Kogge-Stone style scan[a] on these values. The i^{th} element of the exclusive scan is returned to work-item with $\lambda_0 = i$.

Input: `TYPE v`
Local: `local TYPE lS[`$2\times\Lambda_0$`]`

1: **function** SCANLOCALMEM(v)
2: `lS[`λ_0`]` $\leftarrow 0$ ▷ Initialize first half of the array with zeros.
3: `lS[`$\Lambda_0+\lambda_0$`]` $\leftarrow$ `v` ▷ Initialize second half of the array with input values.
4: BARRIER(CLK_LOCAL_MEM_FENCE)

▷ Use $2\times\Lambda_0$ local memory elements to eliminate control statements.
5: **for** `i` $\leftarrow 1$ *ensuremath***to** `i` $<\Lambda_0$ **step** `i` $\leftarrow$ `i` $\times 2$ **do**
6: `t` $\leftarrow$ `lS[`$\Lambda_0+\lambda_0-$`i]`
7: BARRIER(CLK_LOCAL_MEM_FENCE)
8: `lS[`$\Lambda_0+\lambda_0$`]` $\leftarrow$ `lS[`$\Lambda_0+\lambda_0$`]` $+$ `t`
9: BARRIER(CLK_LOCAL_MEM_FENCE)
10: **end for**

11: **return** `lS[`$\Lambda_0+\lambda_0-1$`]` ▷ Return exclusive scan elements.
12: **end function**

[a] `http://en.wikipedia.org/wiki/Kogge-Stone_adder`

Algorithm 10 The Scan (top scan) device code abstraction.

Input: `global TYPE R[B]`

1: **kernel** TOPSCAN (R)
2: `v` $\leftarrow 0$
3: **if** $\lambda_0 < B$ **then**
4: `v` $\leftarrow$ `R[`λ_0`]`
5: **end if**
6: `r` $\leftarrow$ SCANLOCALMEM(v)
7: **if** $\lambda_0 < B$ **then**
8: `R[`λ_0`]` $\leftarrow$ `r`
9: **end if**
10: **end kernel**

Algorithm 11 The Scan (bottom scan) device code abstraction.

Input: `global const TYPE I[N]`
Input: `global const TYPE T[B]`
Output: `global TYPE O[N]`
Local: `local TYPE localSeed`

1: **kernel** BOTTOMSCAN (I, T, O)
2: VECTYPE∗ in4 ← (VECTYPE∗)I
3: VECTYPE∗ out4 ← (VECTYPE∗)O
4: blockSize ← $\lfloor \lfloor N/4 \rfloor / N_0 \rfloor$ ▷ Alignment of 4 elements.
▷ All work-groups but the last work on blockSize elements.
5: blockStart ← $v_0 \times$ blockSize ▷ blockStart $= i_{v_0}$
6: **if** $v_0 = N_0 - 1$ **then** ▷ Last work-group gets remaining elements.
7: blockEnd ← N ▷ blockEnd $= i_N$
8: **else**
9: blockEnd ← blockStart + blockSize ▷ blockEnd $= i_{v_0+1}$
10: **end if**
11: seed ← T$[v_0]$

12: **for** w **in** [blockStart : (blockEnd − 1) : Λ_0] **do**
13: **if** w + λ_0 < blockEnd **then**
14: v ← in4[w + λ_0]
15: **else**
16: v ← $\vec{0}$
17: **end if**
18: v.y ← v.y + v.x
19: v.z ← v.z + v.y
20: v.w ← v.w + v.z
21: r ← SCANLOCALMEM(v.w)
22: v.x ← v.x + r + seed
23: v.y ← v.y + r + seed
24: v.z ← v.z + r + seed
25: v.w ← v.w + r + seed
26: **if** w + λ_0 < blockEnd **then**
27: out4[w + λ_0] ← v
28: **end if**
29: **if** $\lambda_0 = \Lambda_0 - 1$ **then**
30: localSeed ← v.w
31: **end if**
32: BARRIER(CLK_LOCAL_MEM_FENCE)
33: seed ← localSeed
34: **end for**
35: **end kernel**

Algorithm 12 The Scan host code abstraction.

Input: `const int N` ▷ Number of input and output elements.

```
hI ← AllocateMemory(N)
hO ← AllocateMemory(N)
for i in 0 : (N − 1) do
    hI[i] ← i%3                                   ▷ To aid validation.
    hO[i] ← −1
end for
globalWorkSize ← 16384
localWorkSize ← 256
numWorkGroups ← globalWorkSize/localWorkSize      ▷ numWorkGroups = B = 64
dI ← AllocateBuffer(N, READ_WRITE)
dO ← AllocateBuffer(N, READ_WRITE)
dR ← AllocateBuffer(numWorkGroups, READ_WRITE)
CopyToDevice(dI, hI, True, ∅)
scanProgram ← BuildProgram("scan.cl")
reduceKernel ← BuildKernel(scanProgram, "reduce")
topScanKernel ← BuildKernel(scanProgram, "topScan")
bottomScanKernel ← BuildKernel(scanProgram, "bottomScan")
SetKernelArguments(reduceKernel, dI, dR)
SetKernelArguments(topScanKernel, dR)
SetKernelArguments(bottomScanKernel, dI, dR, dO)
EnqueueKernel(reduceKernel, globalWorkSize, localWorkSize, ∅)
EnqueueKernel(topScanKernel, localWorkSize, localWorkSize, ∅)
EnqueueKernel(bottomScanKernel, globalWorkSize, localWorkSize, ∅)
CopyToHost(dO, hO, True, ∅)
```

Input datasets The input data is of sizes $2^{20}, 2^{23}, 2^{25}$ and 2^{26} bytes, which results in $N \in \{2^{18}, 2^{21}, 2^{23}, 2^{24}\}$ for single-precision and $N \in \{2^{17}, 2^{20}, 2^{22}, 2^{23}\}$ for double-precision.

The i^{th} element of $I[\,]$ is initialized with the value $i\%3$, resulting in a pattern of $0, 1, 2, 0, 1, 2, \ldots$.

Partitioned iteration spaces

Segmented Reduction, Algorithm 8:

- Lines 2–9:

$$P_{R_1} = \{(v_0, \lambda_0) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

- Lines 10–12:

$$P_{R_2} = \{(v_0, \lambda_0, i) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256, \\ i = i_{v_0} + \lambda_0 + k \times \Lambda_0, 0 \leq k < (i_{v_0+1} - i_{v_0} - \lambda_0)/\Lambda_0\}$$

- Lines 13–14:

$$P_{R_3} = \{(v_0, \lambda_0) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

- Lines 15–20:

$$P_{R_4} = \{(v_0, \lambda_0, s) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256, \\ s = \lfloor \Lambda_0 / 2^k \rfloor, k \in \{1, 2, \ldots, \log_2 \Lambda_0\}\}$$

- Lines 21–23:

$$P_{R_5} = \{(v_0, \lambda_0) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

`scanLocalMem`, Algorithm 9:[5]

- Lines 2–4:

$$P_{SL_1} = \{(v_0, \lambda_0) : 0 \leq v_0 < N_0, 0 \leq \lambda_0 < \Lambda_0\}$$

- Lines 5–10:

$$P_{SL_2} = \{(v_0, \lambda_0, i) : 0 \leq v_0 < N_0, 0 \leq \lambda_0 < \Lambda_0, \\ i = 2^k, k \in \{0, 1, 2, \ldots, \log_2 \Lambda_0 - 1\}\}$$

- Line 11:

$$P_{SL_3} = \{(v_0, \lambda_0) : 0 \leq v_0 < N_0, 0 \leq \lambda_0 < \Lambda_0\}$$

Top Scan, Algorithm 10:

[5] `scanLocalMem` can be viewed as a blackbox.

- Lines 2–9:

$$P_{TS_1} = \{(v_0, \lambda_0) : v_0 = 0, N_0 = 1, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

Bottom Scan, Algorithm 11:

- Lines 2–12:

$$P_{BS_1} = \{(v_0, \lambda_0) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256\}$$

- Lines 13–36:

$$\begin{aligned} P_{BS_2} = \{&(v_0, \lambda_0, w) : 0 \leq v_0 < N_0 = 64, 0 \leq \lambda_0 < \Lambda_0 = 256, \\ &w = i_{v_0} + k \times \Lambda_0, 0 \leq k < (i_{v_0+1} - i_{v_0})/\Lambda_0\} \end{aligned}$$

Device memory access mapping
Segmented Reduction:

- For P_{R_1}:

$$\begin{aligned} G_r(v_0, \lambda_0) &= \emptyset \\ G_w(v_0, \lambda_0) &= \emptyset \\ L_r(v_0, \lambda_0) &= \emptyset \\ L_w(v_0, \lambda_0) &= \emptyset \end{aligned}$$

- For P_{R_2}:

$$\begin{aligned} G_r(v_0, \lambda_0, i) &= \{\mathtt{I}[i]\} \\ G_w(v_0, \lambda_0, i) &= \emptyset \\ L_r(v_0, \lambda_0, i) &= \emptyset \\ L_w(v_0, \lambda_0, i) &= \emptyset \end{aligned}$$

- For P_{R_3}:

$$\begin{aligned} G_r(v_0, \lambda_0) &= \emptyset \\ G_w(v_0, \lambda_0) &= \emptyset \\ L_r(v_0, \lambda_0) &= \emptyset \\ L_w(v_0, \lambda_0) &= \{\mathtt{lR}[\lambda_0]\} \end{aligned}$$

- For P_{R_4}:

$$\begin{aligned} G_r(v_0, \lambda_0, s) &= \emptyset \\ G_w(v_0, \lambda_0, s) &= \emptyset \\ L_r(v_0, \lambda_0, s) &= \{\mathtt{lR}[\lambda_0], \mathtt{lR}[\lambda_0 + s]\}, \lambda_0 < s \\ L_w(v_0, \lambda_0, s) &= \{\mathtt{lR}[\lambda_0]\}, \lambda_0 < s \\ L_r(v_0, \lambda_0, s) &= \emptyset, \lambda_0 \geq s \\ L_w(v_0, \lambda_0, s) &= \emptyset, \lambda_0 \geq s \end{aligned}$$

- For P_{R_5}:

$$\begin{aligned} G_r(v_0,0) &= \emptyset \\ G_w(v_0,0) &= \{\texttt{R}[v_0]\} \\ L_r(v_0,0) &= \{\texttt{lR}[0]\} \\ L_w(v_0,0) &= \emptyset \\ G_r(v_0,\lambda_0) &= \emptyset, \lambda_0 \neq 0 \\ G_w(v_0,\lambda_0) &= \emptyset, \lambda_0 \neq 0 \\ L_r(v_0,\lambda_0) &= \emptyset, \lambda_0 \neq 0 \\ L_w(v_0,\lambda_0) &= \emptyset, \lambda_0 \neq 0 \end{aligned}$$

`scanLocalMem`:

- For P_{SL_1}:

$$\begin{aligned} L_r(v_0,\lambda_0) &= \emptyset \\ L_w(v_0,\lambda_0) &= \{\texttt{lS}[\lambda_0], \texttt{lS}[\lambda_0+\Lambda_0]\} \end{aligned}$$

- For P_{SL_2}:

$$\begin{aligned} L_r(v_0,\lambda_0,i) &= \{\texttt{lS}[\lambda_0+\Lambda_0]\}, \texttt{lS}[\lambda_0+\Lambda_0-i] \\ L_w(v_0,\lambda_0,i) &= \{\texttt{lS}[\lambda_0+\Lambda_0]\} \end{aligned}$$

- For P_{SL_3}:

$$\begin{aligned} L_r(v_0,\lambda_0) &= \{\texttt{lS}[\lambda_0+\Lambda_0-1]\} \\ L_w(v_0,\lambda_0) &= \emptyset \end{aligned}$$

Top Scan:

- For P_{TS_1}:

$$\begin{aligned} G_r(v_0,\lambda_0) &= \{\texttt{R}[\lambda_0]\}, \lambda_0 < 64 \\ G_r(v_0,\lambda_0) &= \emptyset, \lambda_0 \geq 64 \\ G_w(v_0,\lambda_0) &= \{\texttt{R}[\lambda_0]\}, \lambda_0 < 64 \\ G_w(v_0,\lambda_0) &= \emptyset, \lambda_0 \geq 64 \\ L_r(v_0,\lambda_0) &= \{\texttt{lS}[\lambda_0+\Lambda_0]\}, \texttt{lS}[\lambda_0+\Lambda_0-1], \texttt{ls}[\lambda_0+\Lambda_0-i]\}, i = 2^k, k \in \{0,1,2,\ldots,\log_2\Lambda_0-1\} \\ L_w(v_0,\lambda_0) &= \{\texttt{lS}[\lambda_0], \texttt{lS}[\lambda_0+\Lambda_0]\} \end{aligned}$$

Bottom Scan:

- For P_{BS_1}:

$$\begin{aligned} G_r(v_0,\lambda_0) &= \{\texttt{T}[v_0]\} \\ G_w(v_0,\lambda_0) &= \emptyset \\ L_r(v_0,\lambda_0) &= \emptyset \\ L_w(v_0,\lambda_0) &= \emptyset \end{aligned}$$

- For P_{BS_2}:

$$\begin{aligned}
G_r(v_0,\lambda_0,w) &= \{\texttt{in}[w+\lambda_0+i]\}, i \in \{0,1,2,3\}, w+\lambda_0 < i_{v_0+1} \\
G_r(v_0,\lambda_0,w) &= \emptyset, w+\lambda_0 \geq i_{v_0+1} \\
\\
G_w(v_0,\lambda_0,w) &= \{\texttt{O}[w+\lambda_0+i]\}, i \in \{0,1,2,3\}, w+\lambda_0 < i_{v_0+1} \\
G_w(v_0,\lambda_0,w) &= \emptyset, w+\lambda_0 \geq i_{v_0+1} \\
\\
L_r(v_0,\lambda_0,w) &= \{\texttt{localSeed}, \texttt{lS}[\lambda_0+\Lambda_0]\}, \texttt{lS}[\lambda_0+\Lambda_0-1], \texttt{ls}[\lambda_0+\Lambda_0-i]\}, \\
&\quad i = 2^k, k \in \{0,1,2,\ldots,\log_2\Lambda_0-1\} \\
L_w(v_0,\lambda_0,w) &= \{\texttt{lS}[\lambda_0], \texttt{lS}[\lambda_0+\Lambda_0]\}, \lambda_0 \neq \Lambda_0-1 \\
L_w(v_0,\Lambda_0-1,w) &= \{\texttt{localSeed}, \texttt{lS}[\lambda_0], \texttt{lS}[\lambda_0+\Lambda_0]\}
\end{aligned}$$

Performance metrics

- Bandwidth (*GB/s*) based on execution time: the size of the input array divided by the total kernel execution time.
- Bandwidth (*GB/s*) based on total time: the size of the input array divided by the total time for data transfers plus kernel execution.
- Parity: the data transfer time divided by the kernel execution time.

Validation mechanism The results of the computation are verified against the results of the same operation performed on the host. What is interesting, however, is that the validation is done through an immediate comparison and not taking into account the relative error. This can result into false positives when detecting errors, as floating point addition is not associative and therefore changing the order of the operations can make the results vary between the host and device calculations.

Target-specific optimisations

- An implicit assumption becomes obvious from the above description: the local work size used for all three kernels has to be greater than or equal to the number of work-groups used for the first and third kernel:

$$\Lambda_0 \geq N_0 = \Gamma_0/\Lambda_0$$

  The local work size also needs to be a power of two, in order for the tree-like reductions to work properly. Other than that, the kernels do not seem to rely on particular local work sizes, even though it is mentioned in comments in the code that these kernels would only work with the local work size of 256. In practice, with the global work size of 16384 that is hardcoded in the benchmark, we can reduce the local work size to 128 and still get correct results.

- Another point is that when the input array is partitioned between work-groups, care is taken that the blocks have a size that is a multiple of 4. For an input array with N elements, and a number of work-groups given by $N_0 = \Gamma_0/\Lambda_0$, each block will have a size of $\lfloor \lfloor N/4 \rfloor / N_0 \rfloor \times 4$, with any extra elements assigned to the last work-group. This could lead to significantly imbalanced loads if the input array is small and the number of work-groups is large. It also gives some insight to the selection of $\Gamma_0 = 16384$ for the benchmark, with respect to the input array sizes used. To keep the number of elements per work-group and per-thread meaningful, the value of Γ_0 will need to scale down when the problem size is reduced.

- The choice of the number four as an alignment factor is not clearly documented, and is probably another architecture specific choice.

Target-specific optimization opportunities

- Vectorization could improve the performance for vector architectures.

- The choice of different work sizes can also be a significant architecture-dependent choice – especially since the sizes are hardcoded in the benchmark code and do not depend on the input size.

- More importantly, launching exactly as many work-items as necessary (B) for the top scan, rather than re-using the same local work-size as for the other two kernels, the complicated assumptions regarding the global and local work sizes would be eliminated, and the control statements in `topScan` could be eliminated.

2.2.5 Sort

This benchmark performs a radix sort on unsigned integers.

High-level description

Given an input N-element integer vector $\vec{I}$, the result of sorting $\vec{I}$ gives an N-element vector $\vec{O}$ such that $\vec{O}$ is a permutation of $\vec{I}$ and the following holds:

$$O_i \leq O_{i+1},\ 0 \leq i < N-1 \tag{2.11}$$

If the radix is r and the integer bitwidth is w, each element can be represented using $N_d = \lceil w/r \rceil$ digits.

Least significant digit (LSD) radix sort performs N_d steps: at each step d, the elements are sorted according to the value of their d^{th} least significant digit.

Abstract data structures

- I is an $N_d \times N$-element matrix, where I_0 is the original input vector.
- O is an $N_d \times N$-element matrix, where O_{N_d-1} is the final output vector.
- C is an intermediate $B \times 2^r$-element matrix.
- S is an intermediate $B \times 2^r$-element matrix.
- X is an intermediate N-element vector.

Computation Range $[0:(N-1)]$ is partitioned into B disjoint ranges by a monotonically increasing sequence $\{i_k\} : 0 = i_0 < i_1 < \ldots < i_B = N$:

$$[0:(N-1)] = \bigcup_{b=0}^{B-1} [i_b : (i_{b+1}-1)]$$

We define for convenience: $\texttt{digit}(a,d,r) \stackrel{\text{def}}{=} \lfloor a/2^{d\times r} \rfloor \% 2^r$, where a is an integer element, d is a digit position and r is the radix.

For every digit $d \in \{0,1,\ldots,\lceil w/r \rceil\}$ the following three steps are performed:

- Segmented Reduction:

$$C_{b,v} = |\{I_{d,j} : \texttt{digit}(I_{d,j},d,r) = v,\ i_b \leq j < i_{b+1}\}|,\ 0 \leq b < B \wedge 0 \leq v < 2^r$$

- Top Scan:

$$\begin{aligned} S_{b,0} &= \sum_k C_{k,0},\ 0 \leq k < b < B \\ S_{b,v} &= \sum_k C_{k,v} + S_{B-1,v-1},\ 0 \leq k < b < B \wedge 0 \leq v < 2^r \end{aligned}$$

- Bottom Scan:

$$\begin{aligned} X_i &= S_{b,\texttt{digit}(I_{d,i},d,r)} + \left|\{I_{d,j} : \texttt{digit}(I_{d,j},d,r) = \texttt{digit}(I_{d,i},d,r),\ i_b \leq j \leq i\}\right|, \\ & \quad 0 \leq b < B \wedge i_b \leq i < i_{b+1} \\ O_{d,X_i} &= I_{d,i},\ 0 \leq i < N \end{aligned}$$

Finally, the buffers are swapped:

$$I_{d+1} = O_d,\ 0 \leq d < N_d - 1$$

Reductions and scans are well-known parallel primitives, which are assumed to be associative and commutative (*i.e.* additions can be performed in any order). Since the iteration space, dependences and memory accesses depend on how these primitives are actually implemented, we only analyze the low-level implementation.

Low-level implementation details

Device code Each step of the high-level algorithm maps to one device kernel:

- Segmented Reduction: Algorithm 13.
- Top Scan: Algorithm 14 (calls Algorithm 9, which is reused from the Scan benchmark).
- Bottom Scan: Algorithm 15 (calls Algorithm 9).

Host code See Algorithm 16.

Host data structures

- `uint hI[N]`: $I_{0,x} \rightarrow$ `hI[x]`
- `uint hO[N]`: $O_{N_d-1,x} \rightarrow$ `hO[x]`
- `uint dI[N]`: $I_{2k,x} \rightarrow$ `dI[x]`, $O_{2k+1,x} \rightarrow$ `dI[x]`
- `uint dO[N]`: $O_{2k,x} \rightarrow$ `dO[x]`, $I_{2k+1,x} \rightarrow$ `dO[x]`,
- `uint dC[B]`: $C_{b,v} \rightarrow$ `dC[v * N0+ b]`, $S_{b,v} \rightarrow$ `dC[v* N0+ b]`

Device data structures

- Segmented Reduction:
  - **`global const`** `uint I[N]`: $I_{d,x} \rightarrow$ `I[x]`
  - **`global`** `uint C[B * 16]`: $C_{b,v} \rightarrow$ `C[v * N0+ b]`
  - **`local`** `uint lC[`Λ_0`]`: intermediate results
  - **`private int`** `pC[16]`: intermediate results
- Top Scan:

Algorithm 13 The Sort (Segmented Reduction) device code abstraction.

Input: `global const uint I[N]`
Input: `int d` ▷ Perform this step for the d^{th} least significant digit.
Output: `global uint C[B * 16]` ▷ $r = 4$, so each digit can have $2^4 = 16$ different values.
Local: `local uint lC[`Λ_0`]` ▷ Partial reduction results.

1: **kernel** REDUCE (I, d, C)
2: blockSize $\leftarrow \lfloor \lfloor N/4 \rfloor / N_0 \rfloor \times 4$ ▷ $N_0 = B,\ v_0 = b$ ▷ Alignment of 4 elements. ▷ All work-groups but the last work on blockSize elements.
3: blockStart $\leftarrow v_0 \times$ blockSize ▷ blockStart $= i_{v_0}$
4: **if** $v_0 = N_0 - 1$ **then** ▷ Last work-group gets remaining elements.
5: blockEnd $\leftarrow N$ ▷ blockEnd $= i_N$
6: **else**
7: blockEnd $\leftarrow$ blockStart + blockSize ▷ blockEnd $= i_{v_0+1}$
8: **end if**
9: pC$[0:15] \leftarrow \vec{0}$ ▷ Private histogram vector of size 16.

10: **for** i **in** $[(\text{blockStart} + \lambda_0) : (\text{blockEnd} - 1) : \Lambda_0]$ **do** ▷ Reduce multiple elements per work-item.
11: digit $\leftarrow \lfloor \text{I[i]} / 2^{d \times 4} \rfloor \% 16$ ▷ $r = 4$.
12: pC[digit] $\leftarrow$ pC[digit] $+ 1$
13: **end for**

14: **for** v **in** $[0:15]$ **do** ▷ Iterate over all possible digit values.
15: lC$[\lambda_0] \leftarrow$ pC[v] ▷ Store partial result into local memory.
16: BARRIER(CLK_LOCAL_MEM_FENCE)

17: **for** s $\leftarrow \Lambda_0/2$ *ensuremath***to** s ≤ 0 **step** s $\leftarrow \lfloor$ s$/2 \rfloor$ **do** ▷ Tree-like reduction of partial results.
18: **if** $\lambda_0 <$ s **then**
19: lC$[\lambda_0] \leftarrow$ lC$[\lambda_0] +$ lC$[\lambda_0 +$ s$]$
20: **end if**
21: BARRIER(CLK_LOCAL_MEM_FENCE)
22: **end for**

23: **if** $\lambda_0 = 0$ **then** ▷ Write result to global memory.
24: C$[\text{v} \times N_0 + v_0] \leftarrow$ lC$[0]$
25: **end if**
26: **end for**
27: **end kernel**

Algorithm 14 The Sort (Top Scan) device code abstraction.

Input: `global uint C[B * 16]`
Local: `int localSeed`

```
1:  kernel TOPSCAN (C)
2:      localSeed ← 0
3:      BARRIER(CLK_LOCAL_MEM_FENCE)
4:      for v in [0 : 15] do                          ▷ Iterate over all possible digit values.
5:          c ← 0
6:          if λ0 < B then
7:              c ← C[B × v + λ0]
8:          end if
9:          r ← SCANLOCALMEM(c)
10:         if λ0 < B then
11:             C[B × v + λ0] ← r + localSeed
12:         end if
13:         if λ0 = B − 1 then                        ▷ Last thread to do useful work.
14:             localSeed ← localSeed + r + c
15:         end if
16:         BARRIER(CLK_LOCAL_MEM_FENCE)
17:     end for
18: end kernel
```

- **global** `uint C[B * 16]`: $C_{b,v} \rightarrow$ `C[v * N0+ b]`, $S_{b,v} \rightarrow$ `C[v * N0+ b]`
- **local** `uint lS[2×Λ0]`: intermediate results (`scanLocalMem`)
- **local int** `localSeed`: intermediate result

- Bottom Scan:
  - **global const** `uint I[N]`: $I_x \rightarrow$ `I[x]`
  - **global const** `uint S[B * 16]`: $S_{b,v} \rightarrow$ `S[v * N0+ b]`
  - **global** `uint O[N]`: $O_{d,x} \rightarrow$ `O[x]`
  - **local** `uint lS[2×Λ0]`: intermediate results (`scanLocalMem`)
  - **local** `uint lSeed[16]`: intermediate results
  - **local** `uint lC[16]`: intermediate results
  - **private int** `pC[16]`: intermediate results

Input datasets The input data is of sizes $2^{20}, 2^{23}, 2^{25}$ and 2^{26} bytes, which results in $N \in \{2^{18}, 2^{21}, 2^{23}, 2^{24}\}$ for single-precision and $N \in \{2^{17}, 2^{20}, 2^{22}, 2^{23}\}$ for double-precision.

The i^{th} element of $I[\,]$ is initialized with the value $i\%16$, so all input values are in the range $[0, 16)$. This can affect the validation of the results of this benchmark (see below).

Partitioned iteration spaces
Segmented Reduction, Algorithm 13:

Algorithm 15 The Sort (Bottom Scan) device code abstraction.

Input: `global const uint I[N]` ▷ The input array.
Input: `global const uint S[B * 16]` ▷ The results of top scan.
Input: `int d` ▷ Perform this step for the d^{th} least significant digit.
Output: `global uint O[N]` ▷ The output array.
Local: `local uint lSeed[16]` ▷ Cached elements of S used by work-group.
Local: `local uint lC[16]` ▷ Histogram of digit values seen so far, local for work-group.

1: **kernel** BOTTOMSCAN (I, S, d, O)
2: `uint4 * in4` $\leftarrow$ `(uint4*)I`
3: `blockSize` $\leftarrow \lfloor \lfloor N/4 \rfloor / N_0 \rfloor$ ▷ Alignment of 4 elements.
▷ All work-groups but the last work on `blockSize` elements.
4: `blockStart` $\leftarrow v_0 \times$ `blockSize` ▷ `blockStart` $= i_{v_0}$
5: **if** $v_0 = N_0 - 1$ **then** ▷ Last work-group gets remaining elements.
6: `blockEnd` $\leftarrow N$ ▷ `blockEnd` $= i_N$
7: **else**
8: `blockEnd` $\leftarrow$ `blockStart` $+$ `blockSize` ▷ `blockEnd` $= i_{v_0+1}$
9: **end if**
10: **if** $\lambda_0 < 16$ **then**
11: `lSeed`$[\lambda_0] \leftarrow$ `S`$[\lambda_0 \times N_0 + v_0]$ ▷ S_{v_0,λ_0}
12: `lC`$[\lambda_0] \leftarrow 0$ ▷ Initialize work-group local histogram to zeros.
13: **end if**
14: BARRIER(CLK_LOCAL_MEM_FENCE)
15: `pC`$[0:15] \leftarrow \vec{0}$ ▷ Private histogram vector of size 16.

16: **for** `w` **in** [`blockStart` : (`blockEnd` -1) : Λ_0] **do**
17: **if** `w` $+ \lambda_0 <$ `blockEnd` **then**
18: `val` $\leftarrow$ `in4`$[$`w` $+ \lambda_0]$
19: `digit.x` $\leftarrow \lfloor$ `val.x`$/2^{d\times 4} \rfloor \% 16$ ▷ $digit($`val.x`$, d, 4)$.
20: `digit.y` $\leftarrow \lfloor$ `val.y`$/2^{d\times 4} \rfloor \% 16$ ▷ $digit($`val.y`$, d, 4)$.
21: `digit.z` $\leftarrow \lfloor$ `val.z`$/2^{d\times 4} \rfloor \% 16$ ▷ $digit($`val.z`$, d, 4)$.
22: `digit.w` $\leftarrow \lfloor$ `val.w`$/2^{d\times 4} \rfloor \% 16$ ▷ $digit($`val.w`$, d, 4)$.
23: `pC`$[$`digit.x`$]$ $\leftarrow$ `pC`$[$`digit.x`$] + 1$
24: `pC`$[$`digit.y`$]$ $\leftarrow$ `pC`$[$`digit.y`$] + 1$
25: `pC`$[$`digit.z`$]$ $\leftarrow$ `pC`$[$`digit.z`$] + 1$
26: `pC`$[$`digit.w`$]$ $\leftarrow$ `pC`$[$`digit.w`$] + 1$
27: **end if**
28: **for** `v` **in** $[0:15]$ **do** ▷ Iterate over all possible digit values.
29: `pC`$[$`v`$]$ $\leftarrow$ SCANLOCALMEM(`pC`$[$`v`$]$)
30: BARRIER(CLK_LOCAL_MEM_FENCE)
31: **end for**

Algorithm 15 The Sort (Bottom Scan) device code abstraction (continued).

```
32:         if w + λ0 < blockEnd then
33:             idx ← pC[digit.x] + lSeed[digit.x] + lC[digit.x]
34:             out[idx] ← v.x
35:             pC[digit.x] ← pC[digit.x] + 1
36:             idx ← pC[digit.y] + lSeed[digit.y] + lC[digit.y]
37:             out[idx] ← v.y
38:             pC[digit.y] ← pC[digit.y] + 1
39:             idx ← pC[digit.z] + lSeed[digit.z] + lC[digit.z]
40:             out[idx] ← v.z
41:             pC[digit.z] ← pC[digit.z] + 1
42:             idx ← pC[digit.w] + lSeed[digit.w] + lC[digit.w]
43:             out[idx] ← v.w
44:             pC[digit.w] ← pC[digit.w] + 1
45:         end if
46:         BARRIER(CLK_LOCAL_MEM_FENCE)          ▷ Make sure lC is no longer needed.
47:         if λ0 = Λ0 − 1 then
48:             for v in [0 : 15] do              ▷ Update work-group local histogram.
49:                 lC[v] ← lC[v] + pC[v]
50:             end for
51:         end if
52:         BARRIER(CLK_LOCAL_MEM_FENCE)
53:     end for
54: end kernel
```

Algorithm 16 The Sort host code abstraction.

```
1: procedure ValidateResult(A[N])
2:     for i in 0 : (N − 2) do
3:         if A[i] > A[i + 1] then
4:             Error( )
5:         end if
6:     end for
7: end procedure

Input: N                                   ▷ Number of input and output elements.
8: hI ← AllocateMemory(N)
9: hO ← AllocateMemory(N)
10: for i in 0 : (N − 1) do
11:     hI[i] ← i%16                       ▷ Inadequate input range may leave errors undetected.
12:     hO[i] ← −1
13: end for
14: globalWorkSize ← 16384
15: localWorkSize ← 256
16: numWorkGroups ← globalWorkSize/localWorkSize          ▷ numWorkGroups = B
17: dI ← AllocateBuffer(N, READ_WRITE)
18: dO ← AllocateBuffer(N, READ_WRITE)
19: dC ← AllocateBuffer(numWorkGroups × 16, READ_WRITE)
20: CopyToDevice(dI, hI, True, ∅)
21: sortProgram ← BuildProgram("sort.cl")
22: reduceKernel ← BuildKernel(sortProgram, "reduce")
23: topScanKernel ← BuildKernel(sortProgram, "topScan")
24: bottomScanKernel ← BuildKernel(sortProgram, "bottomScan")
25: SetKernelArguments(topScanKernel, dC)
26: for d in [0 : 7] do                    ▷ Sorting 32-bit integers with r = 4 in 8 steps.
27:     if d%2 = 0 then                                                  ▷ d is even.
28:         SetKernelArguments(reduceKernel, dI, d, dC)
29:         SetKernelArguments(bottomScanKernel, dI, dC, d, dO)
30:     else                                                             ▷ d is odd.
31:         SetKernelArguments(reduceKernel, dO, d, dC)
32:         SetKernelArguments(bottomScanKernel, dO, dC, d, dI)
33:     end if
34:     EnqueueKernel(reduceKernel, globalWorkSize, localWorkSize, ∅)
35:     EnqueueKernel(topScanKernel, localWorkSize, localWorkSize, ∅)
36:     EnqueueKernel(bottomScanKernel, globalWorkSize, localWorkSize, ∅)
37: end for
38: CopyToHost(dI, hO, True, ∅)
39: ValidateResult(hO)
```

- Lines 2–9:

$$P_{R_1} = \{(v_0,\lambda_0) : 0 \le v_0 < N_0 = 64, 0 \le \lambda_0 < \Lambda_0 = 256\}$$

- Lines 10–13:

$$P_{R_2} = \{(v_0,\lambda_0,i) : 0 \le v_0 < N_0 = 64, 0 \le \lambda_0 < \Lambda_0 = 256, \\ i = i_{v_0} + \lambda_0 + k \times \Lambda_0, 0 \le k < (i_{v_0+1} - i_{v_0} - \lambda_0)/\Lambda_0\}$$

- Lines 14–26:[6]

$$P_{R_3} = \{(v_0,\lambda_0,v,s) : 0 \le v_0 < N_0 = 64, 0 \le \lambda_0 < \Lambda_0 = 256, \\ 0 \le v < 16, s = \lfloor \Lambda_0/2^k \rfloor, k \in \{0,1,\ldots,\log_2 \Lambda_0 + 1\}\}$$

`scanLocalMem`, Algorithm 9:

- Lines 2–4:

$$P_{SL_1} = \{(v_0,\lambda_0) : 0 \le v_0 < N_0, 0 \le \lambda_0 < \Lambda_0\}$$

- Lines 5–10:

$$P_{SL_2} = \{(v_0,\lambda_0,i) : 0 \le v_0 < N_0, 0 \le \lambda_0 < \Lambda_0, \\ i = 2^k, k \in \{0,1,\ldots,\log_2 \Lambda_0 - 1\}\}$$

- Line 11:

$$P_{SL_3} = \{(v_0,\lambda_0) : 0 \le v_0 < N_0, 0 \le \lambda_0 < \Lambda_0\}$$

Top Scan, Algorithm 14:

- Lines 2–3:

$$P_{TS_1} = \{(v_0,\lambda_0) : v_0 = 0, N_0 = 1, 0 \le \lambda_0 < \Lambda_0 = 256\}$$

- Lines 4–17:

$$P_{TS_2} = \{(v_0,\lambda_0,v) : v_0 = 0, N_0 = 1, 0 \le \lambda_0 < \Lambda_0 = 256, 0 \le v < 16\}$$

Bottom Scan, Algorithm 15:

- Lines 2–15:

$$P_{BS_1} = \{(v_0,\lambda_0) : 0 \le v_0 < N_0 = 64, 0 \le \lambda_0 < \Lambda_0 = 256\}$$

- Lines 16–53:[7]

$$P_{BS_2} = \{(v_0,\lambda_0,w) : 0 \le v_0 < N_0 = 64, 0 \le \lambda_0 < \Lambda_0 = 256, \\ w = i_{v_0} + k \times \Lambda_0, 0 \le k < (i_{v_0+1} - i_{v_0})/\Lambda_0\}$$

[6] These lines contain an imperfect loop nest. For convenience, we analyze the inner loop by adding two extra iterations, one before and one after the actual iterations. The non-perfectly nested statements in lines 15–16 then belong to the first innermost loop iteration and the statements in lines 23–25 belong to the last inner loop iteration.

[7] These lines contain an imperfect loop nest. For simplicity, we summarize the memory accesses for the inner loops of lines 28–31 and 48–50 for every iteration of the outer loop.

Device memory access mapping
Segmented Reduction:

- For P_{R_1}:

$$\begin{aligned} G_r(v_0,\lambda_0) &= \emptyset \\ G_w(v_0,\lambda_0) &= \emptyset \\ L_r(v_0,\lambda_0) &= \emptyset \\ L_w(v_0,\lambda_0) &= \emptyset \\ P_r(v_0,\lambda_0) &= \emptyset \\ P_w(v_0,\lambda_0) &= \{\texttt{pC}[0:15]\} \end{aligned}$$

- For P_{R_2}:

$$\begin{aligned} G_r(v_0,\lambda_0,i) &= \{\texttt{I}[i]\} \\ G_w(v_0,\lambda_0,i) &= \emptyset \\ L_r(v_0,\lambda_0,i) &= \emptyset \\ L_w(v_0,\lambda_0,i) &= \emptyset \\ P_m r(v_0,\lambda_0,i) &= \{\texttt{pC}[0:15]\} \\ P_m w(v_0,\lambda_0,i) &= \{\texttt{pC}[0:15]\} \end{aligned}$$

- For P_{R_3}:

$$\begin{aligned} G_r(v_0,\lambda_0,v,\Lambda_0) &= \emptyset \\ G_w(v_0,\lambda_0,v,\Lambda_0) &= \emptyset \\ L_r(v_0,\lambda_0,v,\Lambda_0) &= \emptyset \\ L_w(v_0,\lambda_0,v,\Lambda_0) &= \{\texttt{lC}[\lambda_0]\} \\ P_r(v_0,\lambda_0,v,\Lambda_0) &= \{\texttt{pC}[v]\} \\ P_w(v_0,\lambda_0,v,\Lambda_0) &= \emptyset \end{aligned}$$

$$\begin{aligned} G_r(v_0,\lambda_0,v,s) &= \emptyset, 0 < s < \Lambda_0 \\ G_w(v_0,\lambda_0,v,s) &= \emptyset, 0 < s < \Lambda_0 \\ L_r(v_0,\lambda_0,v,s) &= \{\texttt{lC}[\lambda_0], \texttt{lC}[\lambda_0+s]\}, 0 \le \lambda_0 < s < \Lambda_0 \\ L_w(v_0,\lambda_0,v,s) &= \{\texttt{lC}[\lambda_0]\}, 0 \le \lambda_0 < s < \Lambda_0 \\ L_r(v_0,\lambda_0,v,s) &= \emptyset, \lambda_0 \ge s, 0 < s < \Lambda_0 \\ L_w(v_0,\lambda_0,v,s) &= \emptyset, \lambda_0 \ge s, 0 < s < \Lambda_0 \\ P_r(v_0,\lambda_0,v,s) &= \emptyset, 0 < s < \Lambda_0 \\ P_w(v_0,\lambda_0,v,s) &= \emptyset, 0 < s < \Lambda_0 \end{aligned}$$

$$
\begin{aligned}
G_r(v_0,\lambda_0,v,0) &= \emptyset, \lambda_0 \neq 0 \\
G_w(v_0,0,v,0) &= \{\texttt{C}[v \times N_0 + v_0]\} \\
G_w(v_0,\lambda_0,v,0) &= \emptyset, \lambda_0 \neq 0 \\
L_r(v_0,0,v,0) &= \{\texttt{lC}[0]\} \\
L_r(v_0,\lambda_0,v,0) &= \emptyset, \lambda_0 \neq 0 \\
L_w(v_0,\lambda_0,v,0) &= \emptyset \\
P_r(v_0,\lambda_0,v,0) &= \emptyset \\
P_w(v_0,\lambda_0,v,0) &= \emptyset
\end{aligned}
$$

`scanLocalMem`:[8]

- For P_{SL_1}:

$$
\begin{aligned}
L_r(v_0,\lambda_0) &= \emptyset \\
L_w(v_0,\lambda_0) &= \{\texttt{lS}[\lambda_0], \texttt{lS}[\lambda_0 + \Lambda_0]\}
\end{aligned}
$$

- For P_{SL_2}:

$$
\begin{aligned}
L_r(v_0,\lambda_0,i) &= \{\texttt{lS}[\lambda_0 + \Lambda_0]\}, \texttt{lS}[\lambda_0 + \Lambda_0 - i] \\
L_w(v_0,\lambda_0,i) &= \{\texttt{lS}[\lambda_0 + \Lambda_0]\}
\end{aligned}
$$

- For P_{SL_3}:

$$
\begin{aligned}
L_r(v_0,\lambda_0) &= \{\texttt{lS}[\lambda_0 + \Lambda_0 - 1]\} \\
L_w(v_0,\lambda_0) &= \emptyset
\end{aligned}
$$

Top Scan:

- For P_{TS_1}:

$$
\begin{aligned}
G_r(v_0,\lambda_0) &= \emptyset \\
G_w(v_0,\lambda_0) &= \emptyset \\
L_r(v_0,\lambda_0) &= \emptyset \\
L_w(v_0,\lambda_0) &= \{\texttt{localSeed}\}
\end{aligned}
$$

- For P_{TS_2}:

$$
\begin{aligned}
G_r(v_0,\lambda_0,v) &= \begin{cases} \{\texttt{C}[B \times v + \lambda_0]\}, & \lambda_0 < B \\ \emptyset, & \lambda_0 \geq B \end{cases} \\
G_w(v_0,\lambda_0,v) &= \begin{cases} \{\texttt{C}[B \times v + \lambda_0]\}, & \lambda_0 < B \\ \emptyset, & \lambda_0 \geq B \end{cases}
\end{aligned}
$$

[8] `scanLocalMem` can be viewed as a blackbox function.

$$L_r(v_0,\lambda_0,v) = \begin{cases} \{\texttt{lS}[\lambda_0+\Lambda_0],\ \texttt{lS}[\lambda_0+\Lambda_0-1],\ \texttt{ls}[\lambda_0+\Lambda_0-i],\ \texttt{localSeed}\}, \\ \qquad i=2^k, k\in\{0,1,\ldots,\log_2\Lambda_0-1\}, \lambda_0<B \\ \{\texttt{lS}[\lambda_0+\Lambda_0],\ \texttt{lS}[\lambda_0+\Lambda_0-1],\ \texttt{ls}[\lambda_0+\Lambda_0-i]\}, \\ \qquad i=2^k, k\in\{0,1,\ldots,\log_2\Lambda_0-1\}, \lambda_0\geq B \end{cases}$$

$$L_w(v_0,\lambda_0,v) = \begin{cases} \{\texttt{localSeed},\texttt{lS}[\lambda_0],\texttt{lS}[\lambda_0+\Lambda_0]\}, & \lambda_0 = B-1 \\ \{\texttt{lS}[\lambda_0],\texttt{lS}[\lambda_0+\Lambda_0]\}, & \lambda_0 \neq B-1 \end{cases}$$

Bottom Scan:

- For P_{BS_1}:

$$G_r(v_0,\lambda_0) = \begin{cases} \{\texttt{S}[\lambda_0\times N_0+v_0]\}, & \lambda_0<16 \\ \emptyset, & \lambda_0\geq 16 \end{cases}$$

$$\begin{aligned} G_w(v_0,\lambda_0) &= \emptyset \\ L_r(v_0,\lambda_0) &= \emptyset \end{aligned}$$

$$L_w(v_0,\lambda_0) = \begin{cases} \{\texttt{lSeed}[\lambda_0],\texttt{lC}[\lambda_0]\}, & \lambda_0<16 \\ \emptyset, & \lambda_0\geq 16 \end{cases}$$

$$\begin{aligned} P_r(v_0,\lambda_0) &= \emptyset \\ P_w(v_0,\lambda_0) &= \{\texttt{pC}[0:15]\} \end{aligned}$$

- For P_{BS_2}:

$$G_r(v_0,\lambda_0,w) = \begin{cases} \{\texttt{in}[w+\lambda_0+i]\}, & i\in\{0,1,2,3\},^{9}\ w+\lambda_0<i_{v_0+1} \\ \emptyset, & w+\lambda_0\geq i_{v_0+1} \end{cases}$$

$$G_{mw}(v_0,\lambda_0,w) = \begin{cases} \{\texttt{O}[0:(N-1)]\}, & w+\lambda_0<i_{v_0+1} \\ \emptyset, & w+\lambda_0\geq i_{v_0+1} \end{cases}$$

$$G_w(v_0,\lambda_0,w) = \emptyset, w+\lambda_0\geq i_{v_0+1}$$

$$L_r(v_0,\lambda_0,w) = \begin{cases} \{\texttt{lS}[\lambda_0+\Lambda_0]\}, & \texttt{lS}[\lambda_0+\Lambda_0-1],\ \texttt{ls}[\lambda_0+\Lambda_0-i]\}, \\ & \quad i=2^k, k\in\{0,1,\ldots,\log_2\Lambda_0-1\}, \lambda_0\neq\Lambda_0-1 \\ \{\texttt{lS}[\lambda_0+\Lambda_0]\}, & \texttt{lS}[\lambda_0+\Lambda_0-1],\ \texttt{ls}[\lambda_0+\Lambda_0-i],\ \texttt{lC}[0:15]\}, \\ & \quad i=2^k, k\in\{0,1,\ldots,\log_2\Lambda_0-1\} \end{cases}$$

$$L_{mr}(v_0,\lambda_0,w) = \begin{cases} \{\texttt{lSeed}[0:15]\}, & w+\lambda_0<i_{v_0+1} \\ \emptyset, & w+\lambda_0\geq i_{v_0+1} \end{cases}$$

$$L_w(v_0,\lambda_0,w) = \begin{cases} \{\texttt{lS}[\lambda_0],\texttt{lS}[\lambda_0+\Lambda_0]\}, & \lambda_0\neq\Lambda_0-1 \\ \{\texttt{lC}[0:15],\texttt{lS}[\lambda_0],\texttt{lS}[\lambda_0+\Lambda_0]\}, & \lambda_0=\Lambda_0-1 \end{cases}$$

$$\begin{aligned} P_r(v_0,\lambda_0,w) &= \{\texttt{pC}[0:15]\} \\ P_w(v_0,\lambda_0,w) &= \{\texttt{pC}[0:15]\} \end{aligned}$$

Performance metrics

- Bandwidth (*GB/s*) based on execution time: the size of the input array divided by the total kernel execution time.
- Bandwidth (*GB/s*) based on total time: the size of the input array divided by the total time for data transfers plus kernel execution.
- Parity: the data transfer time divided by the kernel execution time.

Validation mechanism The results of the computation are checked to ensure that every element of the output array is less than or equal to the next element. The validation is not adequate, however, as the input array contains only values in the range $[0, 16)$, which can be represented using a single 4-bit digit, which leaves errors such as the one shown in Algorithm 16, line 38, undetected.

Target-specific optimisations and optimization opportunities For target-specific optimisations and optimization opportunities refer to §2.2.4, as this benchmark is closely related to Scan.

2.2.6 MD

High-level description

This benchmark calculates the Lennard-Jones potential for particles in a cube, using a neighbor-list algorithm.

Abstract data structures

- p is an input P element array of coordinate vectors $\vec{p} = (p_x, p_y, p_z)$.
- f is an output P element array of forces $\vec{f} = (f_x, f_y, f_z)$.
- n is an input $P \times N$ array of integer indices: $n_{i,j}$ is the j^{th} neighbor particle of the i^{th} particle.

Computation For every particle, the force is calculated based on its coordinates and the coordinates of its N neighbor particles.

---

for i **in** $0 : (P-1)$ **do**
  $\vec{f_i} \leftarrow 0$
  for j **in** $0 : (N-1)$ **do**
    $\vec{d} \leftarrow \vec{p_i} - \vec{p_{n_{i,j}}}$
    $r \leftarrow ||\vec{d}||$
    if $r < R$ **then** ▷ R is the cutoff distance.
      $c \leftarrow \mathrm{lj}_1/r^{14} - \mathrm{lj}_2/r^8$ ▷ lj_1 and lj_2 are Lennard-Jones constants.
      $\vec{f_i} \leftarrow \vec{f_i} + c \times \vec{d}$
    end if
  end for
end for

Iteration space The iteration space is two-dimensional:

$$I = \{(i, j) : 0 \leq i < P,\ 0 \leq j < N\}$$

Dependences The outer loop carries no dependences. The dependence carried by the inner loop is reduction.

Memory access mapping

$$\begin{aligned} M_r(i,j) &= \{\vec{p_i}, n_{i,j}, \vec{p_{n_{i,j}}}\} \\ M_w(i,j) &= \{\vec{f_i}\} \end{aligned}$$

Low-level implementation details

The benchmark operates on floating point data of type `TYPE` and `VECTYPE`, where `TYPE` is `float` and `VECTYPE` is `float4` for single precision, and `TYPE` is `double` and `VECTYPE` is `double4` for double precision.

Device code See Algorithm 17.

Algorithm 17 The MD device code abstraction.

Input: `global VECTYPE positions[P]`
Input: `global int neighbors[P * N]`
Output: `global VECTYPE forces[P]`

kernel COMPUTE_LJ_FORCE (positions, neighbors, forces)
  ipos $\leftarrow$ positions[γ]
  f $\leftarrow \vec{0}$
  for j **in** $0 : (N-1)$ **do**
    jidx $\leftarrow$ neighbors[j $\times P +$ γ] ▷ Coalesced read.
    jpos $\leftarrow$ positions[jidx] ▷ Uncoalesced read.
    dx $\leftarrow$ ipos.x $-$ jpos.x
    dy $\leftarrow$ ipos.y $-$ jpos.y
    dz $\leftarrow$ ipos.z $-$ jpos.z
    r2 $\leftarrow$ dx^2 + dy^2 + dz^2
    $R \leftarrow 4$ ▷ R is the cutoff distance.
    if r2 $< R^2$ **then**
      r2inv $\leftarrow 1.0/$r2
      r6inv $\leftarrow$ r2inv $\times$ r2inv $\times$ r2inv
      c $\leftarrow$ r2inv $\times$ r6inv $\times$ (lj1 $\times$ r6inv $-$ lj2) ▷ lj1 and lj2 are constants.
      f.x $\leftarrow$ f.x + c $\times$ dx
      f.y $\leftarrow$ f.y + c $\times$ dy
      f.z $\leftarrow$ f.z + c $\times$ dz
    end if
  end for
  forces[γ] $\leftarrow$ f
end kernel

Host code See Algorithm 18.

Host data structures

- **VECTYPE** h_positions[P]: $\vec{p}_o \rightarrow$ h_positions[o]
- **VECTYPE** h_forces[P]: $\vec{f}_o \rightarrow$ h_forces[o]
- **int** h_neighbors[P * N]: $n_{p,o} \rightarrow$ h_neighbors[P * o + p]

Device data structures

- **global VECTYPE** positions[P]: $\vec{p}_o \rightarrow$ positions[o]
- **global VECTYPE** forces[P]: $\vec{f}_o \rightarrow$ forces[o]
- **global int** neighbors[P * N]: $n_{p,o} \rightarrow$ neighbors[P * o + p]

Algorithm 18 The MD host code abstraction.

Input: `const int P` ▷ Number of particles.
Input: `const int N` ▷ Maximum number of nearest neighbors.
Input: `const int cutoff` ▷ Cutoff distance (for neighbor calculation).
Input: `const int lj1, lj2` ▷ Constants for LJ computation.
Input: `const int epsilon` ▷ Relative error for validation.

1: **function** DIST2(pos1, pos2)
2: **return** $(\texttt{pos1.x} - \texttt{pos2.x})^2 + (\texttt{pos1.y} - \texttt{pos2.y})^2 + (\texttt{pos1.z} - \texttt{pos2.z})^2$
3: **end function**

4: **procedure** INSERTINORDER(currDist, currList, j, distIJ, N) ▷ Insert a particle in the neighbor list if the particle is close enough.
5: **if** distIJ $<$ MAX(currDist) **then** ▷ Compare with the furthest neighbor already in the list.
6: **for** i **in** $0 : (N-1)$ **do**
7: **if** distIJ $<$ currDist[i] **then**
8: INSERT(currDist, i, distIJ) ▷ Insert in neighbor lists.
9: INSERT(currList, i, j)
10: RESIZE(currDist, N) ▷ Trim lists.
11: RESIZE(currList, N)
12: **end if**
13: **end for**
14: **end if**
15: **end procedure**

16: **procedure** BUILDNEIGHBORLIST(positions[P], neighbors[N][P]) ▷ Initialize neighbor list.[a]
17: **for** i **in** $0 : P-1$ **do**
18: currList $\leftarrow \{-1, \ldots, -1\}$ ▷ N elements.
19: currDist $\leftarrow \{\infty, \ldots, \infty\}$ ▷ N elements.
20: **for** j **in** $0 : P-1$ **do**
21: **if** i $\neq$ j **then**
22: distIJ $\leftarrow$ DIST2(positions[i], positions[j])
23: INSERTINORDER(currDist, currList, j, distIJ, N)
24: **end if**
25: idx $\leftarrow 0$
26: **for** n **in** currList **do**
27: neighbors[idx][i] $\leftarrow$ n
28: idx $\leftarrow$ idx $+ 1$
29: **end for**
30: **end for**
31: **end for**
32: **end procedure**

[a] When passed to this function, neighbors is actually a flattened, one-dimensional array. Two-dimensional indexing is used for convenience.

Algorithm 18 The MD host code abstraction (continued).

```
33: procedure VALIDATEMD(positions[P], neighbors[P × N], forces[P])
34:     expected_forces[ ] ← HOSTMD(positions[ ], neighbors[ ])
                          ▷ Host code is nearly identical to device code, and hence omitted.
35:     for i in 0 : P − 1 do
36:         diff.x ← (forces[i].x − expected_forces[i].x)/forces[i].x
37:         diff.y ← (forces[i].y − expected_forces[i].y)/forces[i].y
38:         diff.z ← (forces[i].z − expected_forces[i].z)/forces[i].z
39:         err ← √(diff.x^2) + √(diff.y^2) + √(diff.z^2)
40:         if err > 3 × epsilon then ERROR( )
41:         end if
42:     end for
43: end procedure

44: h_positions ← ALLOCATEMEMORY(P)                          ▷ Allocate host memory.
45: h_forces ← ALLOCATEMEMORY(P)
46: h_neighbors ← ALLOCATEMEMORY(P × N)
47: d_positions ← ALLOCATEBUFFER(P, READ_WRITE)              ▷ Should be READ_ONLY.
48: d_neighbors ← ALLOCATEBUFFER(P × N, READ_WRITE)          ▷ Should be READ_ONLY.
49: d_forces ← ALLOCATEBUFFER(P, READ_WRITE)                 ▷ Should be WRITE_ONLY.
50: for i in 0 : P − 1 do          ▷ Initialize particle positions with random floating-point data.
51:     h_positions[i].x ← RAND(0,20)
52:     h_positions[i].y ← RAND(0,20)
53:     h_positions[i].z ← RAND(0,20)
54: end for
55: BUILDNEIGHBORLIST(h_positions, h_neighbors)              ▷ Initialize neighbor list.
56: COPYTODEVICE(d_positions, h_positions, True, ∅)
57: COPYTODEVICE(d_neighbors, h_neighbors, True, ∅)
58: MDProgram ← BUILDPROGRAM("MD.cl")                                ▷ Algorithm 17.
59: MDKernel ← BUILDKERNEL(MDProgram, "compute_lj_forces")
60: SETKERNELARGUMENTS(MDKernel, d_positions, d_neighbors, d_forces)
61: Γ ← P
62: Λ ← 128                                                  ▷ Hardcoded local work size.
63: kernelEvent ← ENQUEUEKERNEL(MDKernel, Γ, Λ, ∅)
64: WAITFOREVENTS({kernelEvent})
65: COPYTOHOST(d_forces, h_forces, True, ∅)
66: VALIDATEMD(h_positions, h_neighbors, h_forces)
```

Input datasets The benchmark operates on randomly generated data,[10] with $N = 128$ and $P = \{12288, 24576, 36864, 73728\}$. The positions are uniformly distributed in a $20.0 \times 20.0 \times 20.0$ cube. The neighbor list is calculated based on the positions and the cutoff distance.

Partitioned iteration space

$$P = \{(\gamma, j) : 0 \leq \gamma < P, 0 \leq j < N\}$$

$$\begin{aligned} f : (i, j) &\rightarrow (i, j) \\ f^{-1} : (\gamma, j) &\rightarrow (\gamma, j) \end{aligned}$$

Device memory access mapping

$$\begin{aligned} G_r(\gamma) &= \{\texttt{positions}[\gamma]\} \\ G_r(\gamma, j) &= \{\texttt{neighbors}[j \times P + \gamma], \texttt{positions}[\texttt{neighbors}[j \times P + \gamma]]\} \\ G_w(\gamma) &= \{\texttt{forces}[\gamma]\} \end{aligned}$$

Performance metrics

- Speed (*GFLOP/s*): the total number of floating point operations ($8 \times P \times N + 13 \times N_{pairs}$, where N_{pairs} is the number of pairs of particles within the cutoff distance) divided by the total time for all data transfers plus calculations.
- Speed (*GFLOP/s*) based on the kernel execution time only: the total number of floating point operations divided by the kernel execution time.
- Bandwidth (*GB/s*) based on the transfer time: the total size of all the memory operations divided by the total time for all data transfers.
- Bandwidth (*GB/s*) based on total time: the total size of all the memory operations divided by the total time for all data transfers plus calculations.
- Parity: the transfer (upload and download) time divided by the kernel execution time.

Validation mechanism The results of running the computation on the device are compared against running the same computation on the host (see Algorithm 18).

Target-specific optimisations The elements of `neighbors[ ]` are stored in column-major order to enable coalesced memory reads.

Target-specific optimization opportunities The device code is explicitly scalarized.

[10] Using random data could potentially lead to non-reproducibility of results. In this case, however, the pseudo-random number generator is seeded with a fixed value.

2.2.7 SGEMM

High-level description

The benchmark performs a matrix-matrix multiplication on floating-point data.

Abstract data structures A, B and C are matrices of size $N \times N$.

Computation

$$C_{i,j} = \alpha \times \sum_{k=0}^{N-1} (A_{i,k} \times B_{k,j}) + \beta \times C_{i,j}, 0 \leq i < N, 0 \leq j < N$$

Iteration space and dependences The iteration space is three-dimensional: $I = \{(i,j,k) : 0 \leq i < N, 0 \leq j < N, 0 \leq k < N\}$.

One of the many ways in which the above operation can be performed is the following, which we will assume for dependence analysis: Considering, for simplicity, the statement for

---

for i **in** $0 : N-1$ **do**
  for j **in** $0 : N-1$ **do**
    $C_{i,j} \leftarrow \beta \times C_{i.j}$
    for k **in** $0 : N-1$ **do**
      $C_{i,j} \leftarrow C_{i.j} + \alpha \times A_{i,k} \times B_{k,j}$
    end for
  end for
end for

---

initialization of $C_{i.j}$ to happen at $(i,j,-1)$, the dependences that arise are:

$$\begin{aligned} \delta_I = \; & I : (i,j,k-1) \xrightarrow{t} I : (i,j,k), \\ & I : (i,j,k-1) \xrightarrow{a} I : (i,j,k), \\ & I : (i,j,k-1) \xrightarrow{o} I : (i,j,k) \end{aligned}$$

These dependences are related to the order in which the additions are performed. The calculation of the matrix-matrix multiply requires essentially one dot product calculation per element of the output matrix, which is a reduction and can be implemented in many ways for parallelism. Obviously, some of the dependences can be removed by introducing new intermediate variables.

When describing the mapping of the above code to the device, we will be concerned with the multiplication operations. The mapping between the additions is not always meaningful, as there can be additions between different intermediate results.

Memory access mapping

$$\begin{aligned} M_r(i,j,-1) &= \{C_{i,j}\} \\ M_w(i,j,-1) &= \{C_{i,j}\} \\ M_r(i,j,k) &= \{C_{i,j}, A_{i,k}, B_{k,j}\} \\ M_w(i,j,k) &= \{C_{i,j}\} \end{aligned}$$

Once more, the above mapping depends on the implementation of the matrix-matrix multiply, which will be significantly different for parallel implementations.

Low-level implementation details

The benchmark operates on floating point data of type **`TYPE`**, where **`TYPE`** is **`float`** for single precision and **`double`** for double precision.

Device code The matrix-matrix multiplication is performed using two different, independent kernels:

- `sgemmNT()`: N and T mean that the matrix A is non-transposed and the matrix B is transposed. See Algorithm 20.
- `sgemmNN()`: N and T mean that the matrices A and B are both non-transposed. See Algorithm 21.

The meaning of transposed and non-transposed is more complicated by the fact that the matrices are assumed to be stored in column-major order. So, non-transposed means a matrix is stored in column-major order and transposed means it is stored in row-major order.

Algorithm 19 SGEMM benchmark : Device code (SAXPY operation)

```
1:  procedure SAXPY(A,B[0:15],C[0:15])
2:      C[0] ← C[0] + A × B[0]
3:      C[1] ← C[0] + A × B[1]
4:      C[2] ← C[0] + A × B[2]
5:      C[3] ← C[0] + A × B[3]
6:      C[4] ← C[0] + A × B[4]
7:      C[5] ← C[0] + A × B[5]
8:      C[6] ← C[0] + A × B[6]
9:      C[7] ← C[0] + A × B[7]
10:     C[8] ← C[0] + A × B[8]
11:     C[9] ← C[0] + A × B[9]
12:     C[10] ← C[0] + A × B[10]
13:     C[11] ← C[0] + A × B[11]
14:     C[12] ← C[0] + A × B[12]
15:     C[13] ← C[0] + A × B[13]
16:     C[14] ← C[0] + A × B[14]
17:     C[15] ← C[0] + A × B[15]
18: end procedure
```

Host code Since the invocation of both kernels is straightforward and exactly the same, only the invocation of `sgemmNT` is shown in Algorithm 22.

Host data structures All matrices are assumed to be stored in column-major order.

When invoking the `sgemmNT()` kernel:

Algorithm 20 SGEMM benchmark : Device code (`sgemmNT` kernel)

Input: `global const TYPE A[]`
Input: `int lda`
Input: `global const TYPE B[]`
Input: `int ldb`
Output: `global TYPE C[]`
Input: `int ldc`
Input: `int k`
Input: `TYPE alpha`
Input: `TYPE beta`
Local: `local TYPE bs[4][16]`

1: **kernel** SGEMMNT (A, lda, B, ldb, C.ldc, k, alpha, beta)
2: id $\leftarrow \lambda_0 + \lambda_1 \times 16$
3: Ap $\leftarrow$ A $+ \nu_0 \times 64 +$ id ▷ Pointer arithmetic.
4: Bp $\leftarrow \nu_1 \times 16 + \lambda_0 + \lambda_1 \times$ ldb
5: Cp $\leftarrow \nu_0 \times 64 +$ id $+ \nu_1 \times 16 \times$ ldc
6: counter $\leftarrow 0$
7: **for** i **in** $[0:3]$ **do**
8: a[i] $\leftarrow$ Ap[i $\times$ lda] ▷ **private TYPE** a[4]
9: **end for**
10: b $\leftarrow$ Bp[0] ▷ **private TYPE** b
11: Ap $\leftarrow$ Ap $+ 4 \times$ lda
12: Bp $\leftarrow$ Bp $+ 4 \times$ ldb
13: counter $\leftarrow$ counter $+ 4 \times$ ldb
14: **for** i **in** $[0:15]$ **do**
15: c[i] $\leftarrow 0.0$ ▷ **private TYPE** c[16]
16: **end for**

Algorithm 20 SGEMM benchmark : Device code (`sgemmNT` kernel) (continued)

```
17:     repeat
18:         for i in [0 : 3] do
19:             as[i] ← a[i]                                  ▷ private TYPE as[4]
20:         end for
21:         bs[λ0][λ1] ← b
22:         BARRIER(CLK_LOCAL_MEM_FENCE)
23:         for i in [0 : 3] do
24:             a[i] ← Ap[i × lda]
25:         end for
26:         b ← Bp[0]
27:         SAXPY(as[0], bs[0], c)
28:         SAXPY(as[1], bs[1], c)
29:         SAXPY(as[2], bs[2], c)
30:         SAXPY(as[3], bs[3], c)
31:         Ap ← Ap + 4 × lda
32:         Bp ← Bp + 4 × ldb
33:         counter ← counter + 4 × ldb
34:         BARRIER(CLK_LOCAL_MEM_FENCE)        ▷ Synchronize so that bs[] won't get
    overwritten.
35:     until counter ≥ k × ldb
36:     bs[λ0][λ1] ← b
37:     BARRIER(CLK_LOCAL_MEM_FENCE)
38:     SAXPY(as[0], bs[0], c)
39:     SAXPY(as[1], bs[1], c)
40:     SAXPY(as[2], bs[2], c)
41:     SAXPY(as[3], bs[3], c)
42:     for i in [0 : 15] do
43:         Cp[0] ← alpha × c[i] + beta × Cp[0]
44:         Cp ← Cp + ldc
45:     end for
46: end kernel
```

Algorithm 21 SGEMM benchmark : Device code (`sgemmNN` kernel)

Input: `global const TYPE A[]`
Input: `int lda`
Input: `global const TYPE B[]`
Input: `int ldb`
Output: `global TYPE C[]`
Input: `int ldc`
Input: `int k`
Input: `TYPE alpha`
Input: `TYPE beta`
Local: `local TYPE bs[16][17]`

1: **kernel** SGEMMNN (A, lda, B, ldb, C.ldc, k, alpha, beta)
2: id $\leftarrow \lambda_0 + \lambda_1 \times 16$
3: Ap $\leftarrow$ A $+ \nu_0 \times 64 +$ id ▷ Pointer arithmetic.
4: Bp $\leftarrow \lambda_0 + (\lambda_1 + \nu_1 \times 16) \times$ ldb
5: Cp $\leftarrow \nu_0 \times 64 +$ id $+ \nu_1 \times 16 \times$ ldc
6: **for** i **in** $[0:15]$ **do**
7: c[i] $\leftarrow 0.0$ ▷ **private TYPE** c[16]
8: **end for**
9: counter $\leftarrow 0$
10: **repeat**
11: **for** i **in** $[0:3]$ **do**
12: a[i] $\leftarrow$ Ap[i $\times$ lda] ▷ **private TYPE** a[4]
13: **end for**
14: bs$[\lambda_0][\lambda_1] \leftarrow$ Bp$[0 \times$ ldb$]$
15: bs$[\lambda_0][\lambda_1 + 4] \leftarrow$ Bp$[4 \times$ ldb$]$
16: bs$[\lambda_0][\lambda_1 + 8] \leftarrow$ Bp$[8 \times$ ldb$]$
17: bs$[\lambda_0][\lambda_1 + 12] \leftarrow$ Bp$[12 \times$ ldb$]$
18: BARRIER(CLK_LOCAL_MEM_FENCE)
19: Ap $\leftarrow$ Ap $+ 4 \times$ lda
20: SAXPY(as[0], bs[0], c)
21: a[0] $\leftarrow$ Ap$[0 \times$ lda$]$
22: SAXPY(as[1], bs[1], c)
23: a[1] $\leftarrow$ Ap$[1 \times$ lda$]$
24: SAXPY(as[2], bs[2], c)
25: a[2] $\leftarrow$ Ap$[2 \times$ lda$]$
26: SAXPY(as[3], bs[3], c)
27: a[3] $\leftarrow$ Ap$[3 \times$ lda$]$
28: Ap $\leftarrow$ Ap $+ 4 \times$ lda
29: SAXPY(as[0], bs[4], c)
30: a[0] $\leftarrow$ Ap$[0 \times$ lda$]$
31: SAXPY(as[1], bs[5], c)
32: a[1] $\leftarrow$ Ap$[1 \times$ lda$]$
33: SAXPY(as[2], bs[6], c)
34: a[2] $\leftarrow$ Ap$[2 \times$ lda$]$
35: SAXPY(as[3], bs[7], c)
36: a[3] $\leftarrow$ Ap$[3 \times$ lda$]$

Algorithm 21 SGEMM benchmark : Device code (`sgemmNN` kernel) (continued)

```
37:         Ap ← Ap + 4 × lda
38:         SAXPY(as[0], bs[8], c)
39:         a[0] ← Ap[0 × lda]
40:         SAXPY(as[1], bs[9], c)
41:         a[1] ← Ap[1 × lda]
42:         SAXPY(as[2], bs[10], c)
43:         a[2] ← Ap[2 × lda]
44:         SAXPY(as[3], bs[11], c)
45:         a[3] ← Ap[3 × lda]
46:         Ap ← Ap + 4 × lda
47:         SAXPY(as[0], bs[12], c)
48:         SAXPY(as[1], bs[13], c)
49:         SAXPY(as[2], bs[14], c)
50:         SAXPY(as[3], bs[15], c)
51:         Bp ← Bp + 16
52:         counter ← counter + 16
53:         BARRIER(CLK_LOCAL_MEM_FENCE)        ▷ Synchronize so that bs[] won't get
    overwritten.
54:     until counter ≥ k
55:     for i in [0 : 15] do
56:         Cp[0] ← alpha × c[i] + beta × Cp[0]
57:         Cp ← Cp + ldc
58:     end for
59: end kernel
```

Algorithm 22 SGEMM : Host code abstraction

Input: N ▷ Size of matrices

1: $\texttt{A} \leftarrow$ ALLOCATEMEMORY($N \times N$) ▷ Allocate host memory.
2: $\texttt{B} \leftarrow$ ALLOCATEMEMORY($N \times N$)
3: $\texttt{C} \leftarrow$ ALLOCATEMEMORY($N \times N$)
4: **for** i **in** $[0 : N-1]$ **do**
5: **for** j **in** $[0 : N-1]$ **do**
6: $\texttt{A}[\texttt{i} \times N + \texttt{j}] \leftarrow 0.5 + 1.5 \times$ RAND(0.0,1.0)
7: $\texttt{B}[\texttt{i} \times N + \texttt{j}] \leftarrow 0.5 + 1.5 \times$ RAND(0.0,1.0)
8: $\texttt{C}[\texttt{i} \times N + \texttt{j}] \leftarrow 0.0$
9: **end for**
10: **end for**
11: $\texttt{Agpu} \leftarrow$ ALLOCATEBUFFER($N \times N$, READ_WRITE) ▷ Allocate device buffers.
12: $\texttt{Bgpu} \leftarrow$ ALLOCATEBUFFER($N \times N$, READ_WRITE)
13: $\texttt{Cgpu} \leftarrow$ ALLOCATEBUFFER($N \times N$, READ_WRITE)
14: $\texttt{writeEvent} \leftarrow$ COPYTODEVICE(Agpu[], A[], **True**, $\emptyset$)
15: $\texttt{writeEvent} \leftarrow$ COPYTODEVICE(Bgpu[], B[], **True**, $\emptyset$)
16: $\texttt{writeEvent} \leftarrow$ COPYTODEVICE(Cgpu[], C[], **True**, $\emptyset$)
17: $\texttt{sgemmProgram} \leftarrow$ BUILDPROGRAM("sgemm.cl")
18: $\texttt{sgemmKernel} \leftarrow$ BUILDKERNEL(sgemmProgram, "sgemmNT")
19: SETKERNELARGUMENTS(sgemmKernel, Agpu, N, Bgpu, N, Cgpu, N, N, α, β)
20: $\texttt{kernelEvent} \leftarrow$ ENQUEUEKERNEL(sgemmKernel, $(N/4, N/4)$, $(16, 4)$, $\emptyset$)
21: WAITFOREVENTS({kernelEvent})
22: COPYTOHOST(Cgpu[], C[], **True**, $\emptyset$)

- **TYPE** `A[N * N]`: $A_{x,y} \rightarrow$ `A[y * N + x]`,
- **TYPE** `B[N * N]`: $B_{x,y} \rightarrow$ `B[x * N + y]`,
- **TYPE** `C[N * N]`: $C_{x,y} \rightarrow$ `C[y * N + x]`,
- **TYPE** `Agpu[N * N]`: $A_{x,y} \rightarrow$ `Agpu[y * N + x]`,
- **TYPE** `Bgpu[N * N]`: $B_{x,y} \rightarrow$ `Bgpu[x * N + y]`,
- **TYPE** `Cgpu[N * N]`: $C_{x,y} \rightarrow$ `Cgpu[y * N + x]`

When invoking the `sgemmNN()` kernel:

- **TYPE** `A[N * N]`: $A_{x,y} \rightarrow$ `A[y * N + x]`,
- **TYPE** `B[N * N]`: $B_{x,y} \rightarrow$ `B[y * N + x]`,
- **TYPE** `C[N * N]`: $C_{x,y} \rightarrow$ `C[y * N + x]`,
- **TYPE** `Agpu[N * N]`: $A_{x,y} \rightarrow$ `Agpu[y * N + x]`,
- **TYPE** `Bgpu[N * N]`: $B_{x,y} \rightarrow$ `Bgpu[y * N + x]`,
- **TYPE** `Cgpu[N * N]`: $C_{x,y} \rightarrow$ `Cgpu[y * N + x]`

Device data structures For the `sgemmNT()` kernel:

- **global const TYPE** `A[N * N]`: $A_{x,y} \rightarrow$ `A[y * N + x]`,
- **global const TYPE** `B[N * N]`: $B_{x,y} \rightarrow$ `B[x * N + y]`,
- **global TYPE** `C[N * N]`: $C_{x,y} \rightarrow$ `C[y * N + x]`,
- **local TYPE** `bs[4][16]`: $B_{x,y} \rightarrow$ `bs[x % 4][y % 16]`

For the `sgemmNN()` kernel:

- **global const TYPE** `A[N * N]`: $A_{x,y} \rightarrow$ `A[y * N + x]`,
- **global const TYPE** `B[N * N]`: $B_{x,y} \rightarrow$ `B[y * N + x]`,
- **global TYPE** `C[N * N]`: $C_{x,y} \rightarrow$ `C[y * N + x]`,
- **local TYPE** `bs[16][17]`: $B_{x,y} \rightarrow$ `bs[x % 16][y % 16]`

Input datasets There are four classes for the sizes of the datasets on which the benchmark operates: classes 1, 2, 3 and 4 work on arrays of dimensions $256 \times 256, 1024 \times 1024, 2048 \times 2048$ and 4096×4096 respectively for single-precision data, and on arrays of dimensions $128 \times 128, 512 \times 512, 1024 \times 1024$ and 2048×2048 respectively for double-precision data.

The computation is repeated for random single and double precision floating point inputs, initialized as shown in Algorithm 22.

Partitioned iteration space For the mapping between iteration spaces, we consider the mapping of the multiplication operations, as the additions are a reduction and can be performed in different ways.

$$P = \{(\nu_0, \nu_1, \lambda_0, \lambda_1) \quad : \quad 0 \leq \nu_0 < N_0 = N/64, 0 \leq \nu_1 < N_1 = N/16, \\ 0 \leq \lambda_0 < \Lambda_0 = 16, 0 \leq \lambda_1 < \Lambda_1 = 4\}$$

The following function maps every iteration $(i, j, k) \in I$ to iteration $(\nu_0, \nu_1, \lambda_0, \lambda_1) \in P$:

$$f : (i, j, k) \rightarrow (i/64, j/16, (i\%64)\%16, (i\%64)/16)$$

The following (one-to-many) relation gives the inverse mapping:

$$f^{-1} : (\nu_0, \nu_1, \lambda_0, \lambda_1) \rightarrow (64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1, 16 \times \nu_1, p), p \in \{0, 1, 2, \ldots, N-1\}$$

Device memory access mapping

- For the `sgemmNT()` kernel:

$$\begin{aligned}
G_r(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{C}[64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1 + 16 \times \nu_1 + \chi \times N], \\
&\quad \texttt{A}[64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1 + \psi \times N], \\
&\quad \texttt{B}[16 \times \nu_1 + \lambda_0 + \lambda_1 \times N + \zeta \times N]\}, \\
&\quad \chi \in \{0, 1, 2, \ldots, 15\}, \psi \in \{0, 1, 2, \ldots, N-1\}, \zeta \in \{0, 4, 8, \ldots, N-4\} \\
G_w(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{C}[64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1 + 16 \times \nu_1 + \chi \times N]\}, \chi \in \{0, 1, 2, \ldots, 15\} \\
L_r(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{bs}\} \\
L_w(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{bs}[\lambda_1][\lambda_0]\}
\end{aligned}$$

- For the `sgemmNN()` kernel:

$$\begin{aligned}
G_r(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{C}[64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1 + 16 \times \nu_1 + \chi \times N], \\
&\quad \texttt{A}[64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1 + \psi \times N], \\
&\quad \texttt{B}[\lambda_0 + (16 \times \nu_1 + \lambda_1) \times N + \omega \times N + \zeta]\}, \\
&\quad \chi \in \{0, 1, 2, \ldots, 15\}, \psi \in \{0, 1, 2, \ldots, N-1\}, \omega \in \{0, 4, 8, 12\}, \\
&\quad \zeta \in \{0, 16, 32, \ldots, N-16\} \\
G_w(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{C}[64 \times \nu_0 + \lambda_0 + 16 \times \lambda_1 + 16 \times \nu_1 + \chi \times N]\}, \chi \in \{0, 1, 2, \ldots, 15\} \\
L_r(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{bs}\} \\
L_w(\nu_0, \nu_1, \lambda_0, \lambda_1) &= \{\texttt{bs}[\lambda_0][\lambda_1], \texttt{bs}[\lambda_0][\lambda_1 + 4], \texttt{bs}[\lambda_0][\lambda_1 + 8], \texttt{bs}[\lambda_0][\lambda_1 + 12]\}
\end{aligned}$$

Performance metrics The performance of each algorithm is evaluated using the following metrics:

- Speed (*GFLOP/s*) based on total execution time: the total number of floating point operations ($2 \times N^3$) divided by the total time for all data transfers plus calculations.
- Speed (*GFLOP/s*) based on kernel execution time only: the total number of floating point operations ($2 \times N^3$) divided by the kernel execution time.
- Parity: the transfer (upload and download) time divided by the kernel execution time.

Validation mechanism No validation mechanism is used for this benchmark.

Target-specific optimisations

- The kernels use local memory, which will increase performance for systems with non-unified memory architectures.
- The local array in the `sgemmNN()` kernel is padded by one column, which is a common technique for NVIDIA cards to reduce bank conflicts in the shared (local) memory.

Target-specific optimization opportunities There are obvious vectorization opportunities missed (e.g. SAXPY operation).

2.2.8 Sparse matrix-vector multiplication

This benchmark multiplies a sparse matrix by a dense vector.

High-level description

$$\vec{y} = A \times \vec{x}$$

Abstract data structures

- A is an input $N \times N$ sparse matrix, with Nz non-zero elements.
- $\vec{x}$ is an input N-element vector.
- $\vec{y}$ is an output N-element vector.

A sparse matrix is a matrix populated primarily with zeros. Since storing a sparse matrix as a two-dimensional array is not space-efficient, many specialized representations have been developed. This benchmark uses two formats, Compressed Sparse Row (CSR) and Ellpack-R, which will be described in separate subsections.

Compressed Sparse Row (CSR)

Data structures

- vals is an Nz-element array that stores the non-zero elements of A.
- cols an Nz-element array that stores the column indices of the non-zero elements.
- rows is an $(N+1)$-element array: rows_i is the index in vals where the first element of the i^{th} row of A is stored, and $\text{rows}_N = Nz$. If row i only contains zeros, $\text{rows}_i = \text{rows}_{i+1}$.

Computation

$$y_i = \sum_{j=\text{rows}_i}^{\text{rows}_{i+1}-1} \text{vals}_j \times x_{\text{cols}_j}$$

Iteration space

$$I = \{(i,j) : 0 \leq i < N, \text{rows}_i \leq j < \text{rows}_{i+1}\}$$

Dependences The outer loop carries no dependences. The dependence carried by the inner loop is reduction.

Memory access mapping

$$\begin{aligned} M_r(i,j) &= \{\text{rows}_i, \text{rows}_{i+1}, \text{vals}_j, \text{cols}_j, x_{\text{cols}_j}, y_i\} \\ M_w(i,j) &= \{y_i\} \end{aligned}$$

Ellpack-R

Data structures Let Nz_R be the maximum number of non-zero elements per row, calculated over all the rows of A.

- rl in an N-element array: rl_i is the number of non-zero elements in the i^{th} row.
- vals is an $N \times Nz_R$-element array that stores the values of non-zero elements of A. The array is stored in column-major order: $\text{vals}[i + j \times N]$ is the j^{th} non-zero element of the i^{th} row, if $j < \text{rl}_i$, or zero if $j \geq \text{rl}_i$.
- cols is an $N \times Nz_R$-element array that stores the column indices corresponding to the non-zero elements stored in vals.

Computation

$$y_i = \sum_{j=0}^{\text{rl}_i - 1} \text{vals}_{i+j\times N} \times x_{\text{cols}_{i+j\times N}}$$

Iteration space

$$I = \{(i, j) : 0 \leq i < N, 0 \leq j < \text{rl}_i\}$$

Dependences The outer loop carries no dependences. The dependence carried by the inner loop is reduction.

Memory access mapping

$$\begin{aligned} M_r(i, j) &= \{\text{rl}_i, \text{vals}_{i+j\times N}, \text{cols}_{i+j\times N}, x_{\text{cols}_{i+j\times N}}, y_i\} \\ M_w(i, j) &= \{y_i\} \end{aligned}$$

Low-level implementation details

The benchmark operates on floating point data of type **`TYPE`**, where **`TYPE`** is **`float`** for single precision and **`double`** for double precision.

Compressed Sparse Row (CSR)

Device code See Algorithm 23 and Algorithm 24 (based on [2, 16]).

Host code See Algorithm 25, Algorithm 26, Algorithm 27.

Host data structures

- **`TYPE`** `hrows[N+1]`: $\text{rows}_i \rightarrow$ `hrows[i]`
- **`TYPE`** `hvals[Nz]`: $\text{vals}_i \rightarrow$ `hvals[i]`
- **`TYPE`** `hcols[Nz]`: $\text{cols}_i \rightarrow$ `hcols[i]`
- **`TYPE`** `hin[N]`: $x_i \rightarrow$ `hin[i]`
- **`TYPE`** `hout[N]`: $y_i \rightarrow$ `hout[i]`

Algorithm 23 The SpMV : `spmvCsrScalar` device code abstraction.

Input: **`global const TYPE`** `vals[`Nz`]`
Input: **`global const TYPE`** `cols[`Nz`]`
Input: **`global const TYPE`** `rows[`$N+1$`]`
Input: **`global const TYPE`** `in[`N`]`
Output: **`global TYPE`** `out[`N`]`

```
1: kernel SPMVCSRSCALAR (vals, cols, rows, in, out)
2:     if γ0 < N then
3:         t ← 0
4:         for j in rows[γ0] : (rows[γ0 + 1] − 1) do
5:             t ← t + vals[j] × in[cols[j]]
6:         end for
7:         out[γ0] ← t
8:     end if
9: end kernel
```

Device data structures For `spmvCsrScalar`:

- **`global const TYPE`** `vals[Nz]`: $\text{vals}_i \rightarrow$ `vals[i]`
- **`global const TYPE`** `cols[Nz]`: $\text{cols}_i \rightarrow$ `cols[i]`
- **`global const TYPE`** `rows[N+1]`: $\text{rows}_i \rightarrow$ `rows[i]`
- **`global const TYPE`** `in[N]`: $x_i \rightarrow$ `in[i]`
- **`global TYPE`** `out[N]`: $y_i \rightarrow$ `out[i]`

For `spmvCsrVector`:

- **`global const TYPE`** `vals[Nz]`: $\text{vals}_i \rightarrow$ `vals[i]`
- **`global const TYPE`** `cols[Nz]`: $\text{cols}_i \rightarrow$ `cols[i]`
- **`global const TYPE`** `rows[N+1]`: $\text{rows}_i \rightarrow$ `rows[i]`
- **`global const TYPE`** `in[N]`: $x_i \rightarrow$ `in[i]`
- **`local TYPE`** `ls[128]`: intermediate results
- **`global TYPE`** `out[N]`: $y_i \rightarrow$ `out[i]`

Input datasets A file in the Matrix Market format[11] can be provided for initializing the sparse matrix A. The details of this initialization are omitted in the host code abstraction.

If no such file is provided, the matrix is initialized randomly as follows. The matrix sizes considered are $N \times N$, where $N \in \{1024, 8192, 12288, 16384\}$. The number of non-zero elements is $Nz = N^2/100$. vals and $\vec{x}$ contain pseudo-random values between zero and a user-defined maximum value. rows and cols are also randomly initialized,[12] as shown in Algorithm 26.

[11] `http://math.nist.gov/MatrixMarket/`

[12] Using random data could potentially lead to non-reproducibility of results. While the pseudo-random number generator is seeded with the same fixed value on all platforms, the results may still vary depending on the platform-specific generator implementation.

Algorithm 24 The SpMV : `spmvCsrVector` device code abstraction.

Input: `global const TYPE vals[`N_z`]`
Input: `global const TYPE cols[`N_z`]`
Input: `global const TYPE rows[`$N+1$`]`
Input: `global const TYPE in[`N`]`
Output: `global TYPE out[`N`]`
Local: `local TYPE ls[128]` ▷ $\Lambda_0 \leq 128$ holds.

1: **kernel** SPMVCSRVECTOR (`vals`, `cols`, `rows`, `in`, `out`)
▷ Despite the name of the kernel, the kernel does not contain vector operations. Each row is split in multiple "vectors", that are multiplied in parallel by `V` threads.
2: `id` $\leftarrow \lambda_0 \% \mathtt{V}$ ▷ `V` is the vector size. `id` is the position of the element that the work-item is calculating for each vector of the row.
3: `i` $\leftarrow \nu_0 \times \lfloor \Lambda_0 / \mathtt{V} \rfloor + \lambda_0 / \mathtt{V}$ ▷ Row to operate on.
4: `ls`$[\lambda_0] \leftarrow 0$
5: **if** `i` $< N$ **then**
6: `t` $\leftarrow 0$
7: **for** `k` **in** $0 : (\lceil (\mathtt{rows[i+1]} - \mathtt{rows[i]} - \mathtt{id}) / \mathtt{V} \rceil - 1)$ **do** ▷ `V` work-items calculate the result for row `i` in parallel.
8: `j` $\leftarrow \mathtt{k} \times \mathtt{V} + \mathtt{id}$
9: `t` $\leftarrow \mathtt{t} + \mathtt{vals[j]} \times \mathtt{in[cols[j]]}$
10: **end for**
11: `ls`$[\lambda_0] \leftarrow$ `t` ▷ Reduction of partial sums – assumes `V` $= 32$.
12: BARRIER(CLK_LOCAL_MEM_FENCE)
13: **if** `id` < 16 **then**
14: `ls`$[\lambda_0] \leftarrow$ `ls`$[\lambda_0] +$ `ls`$[\lambda_0 + 16]$
15: **end if**
16: BARRIER(CLK_LOCAL_MEM_FENCE)
17: **if** `id` < 8 **then**
18: `ls`$[\lambda_0] \leftarrow$ `ls`$[\lambda_0] +$ `ls`$[\lambda_0 + 8]$
19: **end if**
20: BARRIER(CLK_LOCAL_MEM_FENCE)
21: **if** `id` < 4 **then**
22: `ls`$[\lambda_0] \leftarrow$ `ls`$[\lambda_0] +$ `ls`$[\lambda_0 + 4]$
23: **end if**
24: BARRIER(CLK_LOCAL_MEM_FENCE)
25: **if** `id` < 2 **then**
26: `ls`$[\lambda_0] \leftarrow$ `ls`$[\lambda_0] +$ `ls`$[\lambda_0 + 2]$
27: **end if**
28: BARRIER(CLK_LOCAL_MEM_FENCE)
29: **if** `id` < 1 **then**
30: `ls`$[\lambda_0] \leftarrow$ `ls`$[\lambda_0] +$ `ls`$[\lambda_0 + 1]$
31: **end if**
32: BARRIER(CLK_LOCAL_MEM_FENCE)
33: **if** `id` $= 0$ **then**
34: `out`$[$`i`$] \leftarrow$ `ls`$[\lambda_0]$
35: **end if**
36: **end if**
37: **end kernel**

Algorithm 25 The SpMV : CSR host code abstraction.

```
1:  procedure VALIDATERESULT(cpuC[N], gpuC[N])
2:      for i in 0 : (N − 1) do
3:          error ← |cpuC[i] − gpuC[i]|/cpuC[i]
4:          if error ≥ threshold then
5:              ERROR( )
6:          end if
7:      end for
8:  end procedure

9:  procedure SPMVCPU(vals[Nz], cols[Nz], rows[N + 1], in[N], out[N])
10:     for i in 0 : (N − 1) do
11:         t ← 0
12:         for j in rows[i] : (rows[i + 1] − 1) do
13:             t ← t + vals[j] × in[cols[j]]
14:         end for
15:         out[i] ← t
16:     end for
17: end procedure
```

Algorithm 26 The SpMV : CSR host code abstraction.

```
1:  procedure INITRANDOMMATRIX(cols[Nz], rows[N + 1])
2:      count ← 0
3:      P ← Nz/N^2          ▷ Probability that the current element gets assigned a non-zero value.
4:      fillRemaining ← False
5:      for i in 0 : (N − 1) do
6:          rows[i] ← count
7:          for j in 0 : (N − 1) do
8:              Nrem ← N^2 − (i × N + j)                    ▷ Remaining elements.
9:              if Nrem ≤ Nz − count then
10:                 fillRemaining ← True    ▷ All remaining elements need to be non-zero.
11:             end if
12:             if (count < Nz ∧ RAND(0.0, 1.0) ≤ P) ∨ fillRemaining then
13:                 cols[count] ← j
14:                 count ← count + 1
15:             end if
16:         end for
17:     end for
18:     rows[N] ← Nz
19: end procedure

20: procedure FILLRANDOM(m[Nz])
21:     for i in 0 : (Nz − 1) do
22:         m[i] ← MAX_VAL × RAND(0.0, 1.0)
23:     end for
24: end procedure
```

Algorithm 27 The SpMV : CSR host code abstraction.

Input: `int N` ▷ Problem size.

1: $Nz \leftarrow N^2/100$ ▷ 1% of elements is non-zero.
2: `hin` $\leftarrow$ AllocateMemory(N)
3: `hout` $\leftarrow$ AllocateMemory(N)
4: `refout` $\leftarrow$ AllocateMemory(N)
5: `hrows` $\leftarrow$ AllocateMemory($N+1$)
6: `hcols` $\leftarrow$ AllocateMemory(Nz)
7: `hvals` $\leftarrow$ AllocateMemory(Nz)
8: FillRandom(`hin`)
9: InitRandomMatrix(`hcols`, `hrows`)
10: FillRandom(`hvals`)
11: spmvCpu(`hvals`, `hcols`, `hrows`, `hin`, `refout`) ▷ Obtaining reference output.
12: `din` $\leftarrow$ AllocateBuffer(N, `READ_WRITE`) ▷ Should be `READ_ONLY`.
13: `dout` $\leftarrow$ AllocateBuffer(N, `READ_WRITE`)
14: `drows` $\leftarrow$ AllocateBuffer($N+1$, `READ_WRITE`) ▷ Should be `READ_ONLY`.
15: `dvals` $\leftarrow$ AllocateBuffer(Nz, `READ_WRITE`) ▷ Should be `READ_ONLY`.
16: `dcols` $\leftarrow$ AllocateBuffer(Nz, `READ_WRITE`) ▷ Should be `READ_ONLY`.
17: CopyToDevice(`din`, `hin`, **True**, $\emptyset$)
18: CopyToDevice(`drows`, `hrows`, **True**, $\emptyset$)
19: CopyToDevice(`dvals`, `hvals`, **True**, $\emptyset$)
20: CopyToDevice(`dcols`, `hcols`, **True**, $\emptyset$)
21: `spmvProgram` $\leftarrow$ BuildProgram("spmv.cl")
22: `csrScalarKernel` $\leftarrow$ BuildKernel(`spmvProgram`, "spmvCsrScalar")
23: SetKernelArguments(`csrScalarKernel`, `vals`, `cols`, `rows`, `in`, `out`)
24: `csrVectorKernel` $\leftarrow$ BuildKernel(`spmvProgram`, "spmvCsrVector")
25: SetKernelArguments(`csrVectorKernel`, `vals`, `cols`, `rows`, `in`, `out`)

26: EnqueueKernel(`csrScalarKernel`, $(\Gamma_0 = N), (\Lambda_0 = 128), \emptyset$)
27: CopyToHost(`dout`, `hout`, **True**, $\emptyset$)
28: ValidateResult(`refout`, `hout`)
29: CopyToDevice(`dout`, `hin`, **True**, $\emptyset$) ▷ Clobber previous results.

30: $\Gamma_0 \leftarrow N \times$ `V` ▷ `V` is the vector size ($V = 32$).
31: $\Lambda_{\texttt{max}} \leftarrow$ GetKernelInfo(`csrVectorKernel`, `MAX_WORK_SIZE`)
32: $\Lambda_0 \leftarrow \max(\texttt{V}, \texttt{V} \times \lfloor \min(128, \Lambda_{\texttt{max}})/\texttt{V} \rfloor)$ ▷ If $\Lambda_{\texttt{max}} < \texttt{V} = 32$, `enqueueKernel` will fail.
33: EnqueueKernel(`csrVectorKernel`, $\Gamma_0, \Lambda_0, \emptyset$)
34: CopyToHost(`dout`, `hout`, **True**, $\emptyset$)
35: ValidateResult(`refout`, `hout`)

Partitioned iteration space For `spmvCsrScalar`:

$$P = \{(\gamma_0, j) : 0 \leq \gamma_0 < N, \text{rows}[\gamma_0] \leq j < \text{rows}[\gamma_0 + 1]\}$$

$$\begin{aligned} f : (i, j) &\rightarrow (i, j) \\ f^{-1} : (\gamma_0, j) &\rightarrow (\gamma_0, j) \end{aligned}$$

For `spmvCsrVector`:

$$\begin{aligned} P = \{&(\nu_0, \lambda_0, k) : 0 \leq \nu_0 < N_0 = N/w, 0 \leq k < \Lambda_0 = w \times V, w \in \mathbb{N}, \\ &0 \leq k < \lceil (\text{rows}[r+1] - \text{rows}[r] - \lambda_0 \% \mathtt{V})/\mathtt{V} \rceil, \\ &r = \nu_0 \times (\Lambda_0/\mathtt{V}) + \lfloor \lambda_0/\mathtt{V} \rfloor\} \end{aligned}$$

$$\begin{aligned} f : (i, j) &\rightarrow \lfloor (i \times \mathtt{V})/\Lambda_0 \rfloor, \mathtt{V} \times (i \% (\Lambda_0/\mathtt{V})) + j \% \mathtt{V}, \lfloor j/\mathtt{V} \rfloor) \\ f^{-1} : (\nu_0, \lambda_0, k) &\rightarrow (\nu_0 \times (\Lambda_0/\mathtt{V}) + \lfloor \lambda_0/\mathtt{V} \rfloor, k \times \mathtt{V} + \lambda_0 \% \mathtt{V}) \end{aligned}$$

We do not describe the imperfectly nested statements as separate iteration spaces.

Device memory access mapping For `spmvCsrScalar`:

- Before the main loop body:

$$G_r(\gamma_0) = \{\mathtt{rows}[\gamma_0], \mathtt{rows}[\gamma_0 + 1]\}$$

- Main loop body:

$$G_r(\gamma_0, j) = \{\mathtt{vals}[j], \mathtt{cols}[j], \mathtt{in}[\mathtt{cols}[j]]\}$$

- After the main loop body:

$$G_w(\gamma_0) = \{\mathtt{out}[\gamma_0]\}$$

For `spmvCsrVector`:

- Before the main loop body:

$$\begin{aligned} G_r(\nu_0, \lambda_0) &= \{\mathtt{rows}[r], \mathtt{rows}[r+1]\}, \\ &\quad r = \nu_0 \times (\Lambda_0/\mathtt{V}) + \lfloor \lambda_0/\mathtt{V} \rfloor, r < N \\ L_w(\nu_0, \lambda_0) &= \{\mathtt{ls}[\lambda_0]\}, \end{aligned}$$

- Main loop body:

$$\begin{aligned} G_r(\nu_0, \lambda_0, k) &= \{\mathtt{vals}[j], \mathtt{cols}[j], \mathtt{in}[\mathtt{cols}[j]]\}, j = k \times \mathtt{V} + \lambda_0 \% \mathtt{V} \\ &\quad r = \nu_0 \times (\Lambda_0/\mathtt{V}) + \lfloor \lambda_0/\mathtt{V} \rfloor, r < N \end{aligned}$$

- After the main loop body:

$$L_w(\nu_0,\lambda_0) \quad = \quad \{\texttt{ls}[\lambda_0]\},$$

$$L_r(\nu_0,\lambda_0) \quad = \begin{cases} \{\texttt{ls}[\lambda_0],\texttt{ls}[\lambda_0+16],\texttt{ls}[\lambda_0+8],\texttt{ls}[\lambda_0+4],\texttt{ls}[\lambda_0+2],\texttt{ls}[\lambda_0+1]\}, & \lambda_0 = 0 \\ \{\texttt{ls}[\lambda_0],\texttt{ls}[\lambda_0+16],\texttt{ls}[\lambda_0+8],\texttt{ls}[\lambda_0+4],\texttt{ls}[\lambda_0+2]\}, & \lambda_0 = 1 \\ \{\texttt{ls}[\lambda_0],\texttt{ls}[\lambda_0+16],\texttt{ls}[\lambda_0+8],\texttt{ls}[\lambda_0+4]\}, & 2 \leq \lambda_0 < 4 \\ \{\texttt{ls}[\lambda_0],\texttt{ls}[\lambda_0+16],\texttt{ls}[\lambda_0+8]\}, & 4 \leq \lambda_0 < 8 \\ \{\texttt{ls}[\lambda_0],\texttt{ls}[\lambda_0+16]\}, & 8 \leq \lambda_0 < 16 \end{cases}$$

$$G_w(\nu_0,\lambda_0) \quad = \quad \{\texttt{out}[r]\}, r = \nu_0 \times (\Lambda_0/\texttt{V}) + \lfloor \lambda_0/\texttt{V} \rfloor, r < N$$

Performance metrics

- Speed (*GFLOP/s*) based on calculations: the total number of floating point operations ($2 \times Nz$) divided by the time for calculations.
- Speed (*GFLOP/s*) based on total time: the total number of floating point operations ($2 \times Nz$) divided by the total time for all data transfers plus calculations.

Validation mechanism The results of the computation are verified against the results of the same sparse-matrix vector multiplication performed on the host, using the CSR format.

Target-specific optimisations

- When the device supports images, $\vec{x}$ is stored using an image type. This enables the use of texture memory on NVIDIA graphics cards, which will have different effects on performance for different architectures, even for different generations of NVIDIA graphics cards.
- The two kernels are run two times: for the first run, the arrays rows, vals and cols are initialized as shown in the host code abstraction; for the second run, vals and cols are padded so that each row of values is a multiple of a padding factor. rows is adjusted accordingly. (Details of this padding are omitted in the host code representation).
- In `spmvCsrVector`, `V` work-items within a work-group with consecutive local ids access consecutive elements of the arrays vals and cols, which improves the memory system performance (caching, coalescing).

Target-specific optimization opportunities Vectorization could be applied to the device code, however the indirect accesses to $\vec{x}$ cannot be vectorized.

Sparse-matrix vector multiplication is a well-studied problem with various different formats and corresponding algorithms. Choosing the right data representation to implement is a target-specific choice.

Ellpack-R

Device code See Algorithm 28 (based on [9]).

Algorithm 28 The SpMV : `spmvEllpackR` device code abstraction.

Input: **global const TYPE** `vals[`$N \times NzR$`]`
Input: **global const TYPE** `cols[`$N \times NzR$`]`
Input: **global const TYPE** `rl[`N`]`
Input: **global const TYPE** `in[`N`]`
Output: **global TYPE** `out[`N`]`

1: **kernel** SPMVELLPACKR (`vals`, `cols`, `rl`, `in`, `out`)
2: **if** $\gamma_0 < N$ **then**
3: `t` $\leftarrow 0$
4: **for** `j` **in** $0 : (\texttt{rl}[\gamma_0] - 1)$ **do**
5: `idx` $\leftarrow \gamma_0 + \texttt{j} \times N$ ▷ Column-major storage.
6: `t` $\leftarrow \texttt{t} + \texttt{vals}[\texttt{idx}] \times \texttt{in}[\texttt{cols}[\texttt{idx}]]$
7: **end for**
8: `out`$[\gamma_0] \leftarrow$ `t`
9: **end if**
10: **end kernel**

Host code See Algorithm 25, Algorithm 26, Algorithm 29.

Host data structures

- **TYPE** `hrl[N]`: $\text{rl}_i \rightarrow$ `hrl[i]`
- **TYPE** `hvals_cm[N * NzR]`: $\text{vals}_i \rightarrow$ `hvals_cm[i]`
- **TYPE** `hcols_cm[N * NzR]`: $\text{cols}_i \rightarrow$ `hcols_cm[i]`
- **TYPE** `hin[N]`: $x_i \rightarrow$ `hin[i]`
- **TYPE** `hout[N]`: $y_i \rightarrow$ `hout[i]`

Device data structures

- **global const TYPE** `vals[N * NzR]`: $\text{vals}_i \rightarrow$ `vals[i]`
- **global const TYPE** `cols[N * NzR]`: $\text{cols}_i \rightarrow$ `cols[i]`
- **global const TYPE** `rl[N]`: $\text{rl}_i \rightarrow$ `rl[i]`
- **global const TYPE** `in[N]`: $x_i \rightarrow$ `in[i]`
- **global TYPE** `out[N]`: $y_i \rightarrow$ `out[i]`

Input datasets The input datasets used are the same as for the CSR implementation.

Algorithm 29 The SpMV : Ellpack-R host code abstraction.

```
1: Nz ← N^2/100                                   ▷ 1% of elements is non-zero.
2: hin ← AllocateMemory(N)
3: hout ← AllocateMemory(N)
4: refout ← AllocateMemory(N)
5: hrows ← AllocateMemory(N+1)
6: hcols ← AllocateMemory(Nz)
7: hvals ← AllocateMemory(Nz)
8: FillRandom(hin)
9: InitRandomMatrix(hcols, hrows)
10: FillRandom(hvals)
11: spmvCpu(hvals, hcols, hrows, hin, refout)     ▷ Obtaining reference output.
12: hrl ← AllocateMemory(N)
13: NzR ← 0
14: for i in 0 : (N − 1) do
15:     hrl[i] ← hrows[i + 1] − hrows[i]
16:     if hrl[i] > NzR then
17:         NzR ← hrl[i]
18:     end if
19: end for
20: hvals_cm ← AllocateMemory(N × NzR)
21: hcols_cm ← AllocateMemory(N × NzR)
22: ConvertToColumnMajor(hvals, hcols, hrows, hvals_cm, hcols_cm, hrl)   ▷
    Details omitted.
23: din ← AllocateBuffer(N, READ_WRITE)           ▷ Should be READ_ONLY.
24: dout ← AllocateBuffer(N, READ_WRITE)
25: drl ← AllocateBuffer(N, READ_WRITE)           ▷ Should be READ_ONLY.
26: dvals_cm ← AllocateBuffer(N × NzR, READ_WRITE)  ▷ Should be READ_ONLY.
27: dcols_cm ← AllocateBuffer(N × NzR, READ_WRITE)  ▷ Should be READ_ONLY.
28: CopyToDevice(din, hin, True, ∅)
29: CopyToDevice(drl, hrl, True, ∅)
30: CopyToDevice(dvals_cm, hvals_cm, True, ∅)
31: CopyToDevice(dcols_cm, hcols_cm, True, ∅)
32: spmvProgram ← BuildProgram("spmv.cl")
33: ellpackrKernel ← BuildKernel(spmvProgram, "spmvEllpackR")
34: SetKernelArguments(ellpackrKernel, vals, cols, rows, in, out)

35: EnqueueKernel(ellpackrKernel, (Γ0 = N), (Λ0 = 128), ∅)
36: CopyToHost(dout, hout, True, ∅)
37: ValidateResult(refout, hout)
```

Partitioned iteration space

$$P = \{(\gamma_0, j) : 0 \leq \gamma_0 < N, 0 \leq j < \text{rl}[\gamma_0]\}$$

$$\begin{aligned} f : (i, j) &\rightarrow (i, j) \\ f^{-1} : (\gamma_0, j) &\rightarrow (\gamma_0, j) \end{aligned}$$

We do not describe the imperfectly nested statements as separate iteration spaces.

Device memory access mapping

- Before the main loop body:

$$G_r(\gamma_0) = \{\texttt{rl}[\gamma_0]\}$$

- Main loop body:

$$G_r(\gamma_0, j) = \{\texttt{vals}[\gamma_0 + j \times N], \texttt{cols}[\gamma_0 + j \times N], \texttt{in}[\texttt{cols}[\gamma_0 + j \times N]]\}$$

- After the main loop body:

$$G_w(\gamma_0) = \{\texttt{out}[\gamma_0]\}$$

Performance metrics The performance metrics used are the same as for the CSR implementation.

Validation mechanism The validation mechanism used is the same as for the CSR implementation.

Target-specific optimisations In case the device supports images, $\vec{x}$ is stored using an image type. This enables the use of texture memory on NVIDIA graphics cards, which will have different effects on performance for different architectures, even for different generations of NVIDIA graphics cards.

Storing the vals and cols arrays in column-major format causes work items with consecutive global ids to access consecutive memory locations, which improves the memory system performance (caching, coalescing).

Target-specific optimization opportunities Refer to the target-specific optimization opportunities section for the CSR implementation.

2.3 Rodinia

The Rodinia 2.1 benchmark suite[5, 6] from Virginia university.

2.3.1 Back Propagation

The benchmark uses the Back Propagation machine-learning algorithm to train the weights of edges in an artificial layered neural network [15].

High-level description

Abstract data structures The benchmark operates over the artificial layered neural network $P = (I, H, O, W^I, W^H)$:

- I is an $(N_I + 1)$ element layer of the input nodes.
- H is an $(N_H + 1)$ element layer of the hidden nodes.
- O is an $(N_O + 1)$ element layer of the output nodes.
- W^I is an $(N_I + 1) \times (N_H + 1)$ input matrix of weights from the input layer to the hidden layer.
- W^H is an $(N_H + 1) \times (N_O + 1)$ input matrix of weights from the hidden layer to the output layer.
- $W^{I'}$ is an $(N_I + 1) \times (N_H + 1)$ intermediate matrix of weights from the input layer to the hidden layer.
- $W^{H'}$ is an $(N_H + 1) \times (N_O + 1)$ intermediate matrix of weights from the hidden layer to the output layer.
- $\vec{S}$ is an $(N_I + 1)$ element input vector of values for the input layer.
- $\vec{T}$ is an $(N_O + 1)$ element input vector of target values for the output layer.
- $\vec{D}^O$ is an $(N_O + 1)$ element intermediate vector of delta values for the output layer.
- $\vec{D}^H$ is an $(N_H + 1)$ element intermediate vector of delta values for the hidden layer.

Computation The following *squash* function is used for Forward Propagation:

$$F(x) = (1 + e^{-x})^{-1}$$

Initially input nodes layer I contains the values from input vector $\vec{S}$

- Initialize:

$$\begin{aligned} I &= S \\ I_0 &= 1 \\ W^{H'} &= \|0\| \\ W^{I'} &= \|0\| \end{aligned}$$

- Compute the hidden nodes from the input nodes (Forward Propagation):

$$H_0 = 1$$

$$H_i = F(\sum_{0 \le j \le N_I} I_j \times W^I_{j,i}), 0 < i \le N_H$$

- Compute the output nodes from the hidden nodes (Forward Propagation):

$$O_i = F(\sum_{0 \le j \le N_H} H_j \times W^H_{j,i}), 0 < i \le N_O$$

- Compute the deltas between the target and actual output vectors (Backward Propagation):

$$D^O_i = O_i \times (1 - O_i) \times (O_i - T_i), 0 < i \le N_O$$

- Compute the "deltas" for the hidden nodes (Backward Propagation):

$$D^H_i = H_i \times (1 - H_i) \times \left(\sum_{0 < j \le N_O} D^O_j \times W^H_{i,j} \right), 0 < i \le N_H$$

- Update the hidden-to-output weights (Weights Adjustment):

$$W^{H'}_{i,j} = 0.3 \times D^O_j \times H_i + 0.3 \times W^{H'}_{i,j}, 0 \le i \le N_H, 0 < j \le N_O$$

$$W^H_{i,j} = W^H_{i,j} + W^{H'}_{i,j}, 0 \le i \le N_H, 0 < j \le N_O$$

- Update the input-to-hidden weights (Weights Adjustment):

$$W^{I'}_{i,j} = 0.3 \times D^H_j \times I_i + 0.3 \times W^{I'}_{i,j}, 0 \le i \le N_I, 0 < j \le N_H$$

$$W^I_{i,j} = W^I_{i,j} + W^{I'}_{i,j}, 0 \le i \le N_I, 0 < j \le N_H$$

Iteration spaces

- Compute the hidden nodes from the input nodes:

$$I_H = \{(i, j) : 0 < i \le N_H, 0 \le j \le N_I\}$$

$$I_H : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \le \begin{pmatrix} N_H \\ N_I \\ -1 \\ 0 \end{pmatrix}$$

- Compute the output nodes from the hidden nodes:

$$I_O = \{(i,j) : 0 < i \leq N_O, 0 \leq j \leq N_H\}$$

$$I_O : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} N_O \\ N_H \\ -1 \\ 0 \end{pmatrix}$$

- Compute the delta between the target and actual output:

$$I_{D_O} = \{(i) : 0 < i \leq N_O\}$$

$$I_{D_O} : \begin{pmatrix} 1 \\ -1 \end{pmatrix} \times (i) \leq \begin{pmatrix} N_O \\ -1 \end{pmatrix}$$

- Compute the "delta" for the hidden nodes:

$$I_{D_H} = \{(i,j) : 0 < i \leq N_H, 0 < j \leq N_O\}$$

$$I_{D_H} : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} N_H \\ N_O \\ -1 \\ -1 \end{pmatrix}$$

- Update the hidden-to-output weights:

$$I_{W_H} = \{(i,j) : 0 \leq i \leq N_H, 0 < j \leq N_O\}$$

$$I_{W_H} : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} N_H \\ N_O \\ 0 \\ -1 \end{pmatrix}$$

- Update the input-to-hidden weights:

$$I_{W_I} = \{(i,j) : 0 \leq i \leq N_I, 0 < j \leq N_H\}$$

$$I_{W_I} : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} N_I \\ N_H \\ 0 \\ -1 \end{pmatrix}$$

$$I = I_H \cup I_O \cup I_{D_O} \cup I_{D_H} \cup I_{W_H} \cup I_{W_I}$$

Dependences

$$\begin{aligned}
\delta_{I_O} &= I_H : (j, (1 : N_I)) \xrightarrow{t} I_O : (i, j) \\
\delta_{I_{D_O}} &= I_O : (i, (1 : N_H)) \xrightarrow{t} I_{D_O} : (i) \\
\delta_{I_{D_H}} &= I_H : (i, (1 : N_I)) \xrightarrow{t} I_{D_H} : (i, j) \wedge \\
&\quad I_{D_O} : (j) \xrightarrow{t} I_{D_H} : (i, j) \\
\delta_{I_{W_H}} &= I_H : (i, (1 : N_I)) \xrightarrow{t} I_{W_H} : (i, j) \wedge \\
&\quad I_{D_O} : (j) \xrightarrow{t} I_{W_H} : (i, j) \\
\delta_{I_{W_I}} &= I_{D_H} : (j, (1 : N_O)) \xrightarrow{t} I_{W_I} : (i, j)
\end{aligned}$$

Memory access mapping

- Compute the hidden nodes from the input nodes: for $(i, j) \in I_H$:

$$\begin{aligned}
M_r(i, j) &= \{I_j, W^I_{j,i}\} \\
M_w(i, j) &= \{H_i\}
\end{aligned}$$

- Compute the output nodes from the hidden nodes: for $(i, j) \in I_O$:

$$\begin{aligned}
M_r(i, j) &= \{H_j, W^H_{j,i}\} \\
M_w(i, j) &= \{O_i\}
\end{aligned}$$

- Compute the delta between the target and actual output: for $(i) \in I_{D_O}$:

$$\begin{aligned}
M_r(i) &= \{O_i, T_i\} \\
M_w(i) &= \{D^O_i\}
\end{aligned}$$

- Compute the delta for the hidden nodes: for $(i, j) \in I_{D_H}$:

$$\begin{aligned}
M_r(i, j) &= \{H_i, W^H_{i,j}, D^O_j\} \\
M_w(i, j) &= \{D^H_i\}
\end{aligned}$$

- Update hidden to output weights: for $(i, j) \in I_{W_H}$:

$$\begin{aligned}
M_r(i, j) &= \{D^O_j, W^{H'}_{i,j}, H_i, W^H_{i,j}\} \\
M_w(i, j) &= \{W^{H'}_{i,j}, W^H_{i,j}\}
\end{aligned}$$

- Update input to hidden weights: for $(i, j) \in I_{W_I}$:

$$\begin{aligned}
M_r(i, j) &= \{D^H_j, W^{I'}_{i,j}, I_i, W^I_{i,j}\} \\
M_w(i, j) &= \{W^{I'}_{i,j}, W^I_{i,j}\}
\end{aligned}$$

Algorithm 30 The Forward Propagation device code abstraction.

Input: `global float dI[]`
Input: `global float dWI[]`
Input: `int Nh`
Output: `global float dH[]`
Local: `local float dlI[]`
Local: `local float dlW[]`

1: **kernel** BPKERNEL1 (dI, dWI, Nh, dH)
2: $\texttt{xidx} \leftarrow \lambda_0 + 1$
3: $\texttt{yidx} \leftarrow \gamma_1 + 1$
4: $\texttt{idx} \leftarrow (\texttt{Nh} + 1) \times \texttt{yidx} + \texttt{xidx}$
5: **if** $\lambda_0 = 0$ **then**
6: $\texttt{dlI}[\lambda_1] \leftarrow \texttt{dI}[\texttt{yidx}]$
7: **end if**
8: BARRIER(CLK_LOCAL_MEM_FENCE)
9: $\texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0] \leftarrow \texttt{dWI}[\texttt{idx}]$ ▷ Unnecessary memory access.
10: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.
11: $\texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0] \leftarrow \texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0] \times \texttt{dlI}[\lambda_1]$
12: BARRIER(CLK_LOCAL_MEM_FENCE)
13: **for** i **in** $1 : \log_2(\Lambda_1)$ **do**
14: **if** $\lambda_1 \bmod (2^i) = 0$ **then**
15: $\texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0] \leftarrow \texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0] + \texttt{dlW}[(\lambda_1 + 2^{i-1}) \times \Lambda_0 + \lambda_0]$
16: **end if**
17: BARRIER(CLK_LOCAL_MEM_FENCE)
18: **end for**
19: $\texttt{dWI}[\texttt{idx}] \leftarrow \texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0]$ ▷ Unnecessary memory access (see device memory access mapping for more details).
20: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.
21: **if** $\lambda_0 = 0$ **then**
22: $\texttt{dH}[\gamma_1] \leftarrow \texttt{dlW}[\lambda_1 \times \Lambda_0 + \lambda_0]$
23: **end if**
24: **end kernel**

Algorithm 31 The Weights Adjustment device code abstraction.

Input: `global float dDH[]`
Input: `global float dI[]`
Input/Output: `global float dWI[]`
Input/Output: `global float dWIo[]`
Input: `int Nh`
Output: `global int dCout[]`

1: **kernel** BPKERNEL2 (dDH, dI, dWI, dWIo, Nh)
2: xidx $\leftarrow \lambda_0 + 1$
3: yidx $\leftarrow \gamma_1 + 1$
4: idx $\leftarrow (\mathtt{Nh} + 1) \times \mathtt{yidx} + \mathtt{xidx}$
5: dWIo[idx] $\leftarrow 0.3 \times$ dDH[xidx] $\times$ dI[yidx] $+ 0.3 \times$ dWIo[idx]
6: dWI[idx] $\leftarrow$ dWI[idx] $+$ dWIo[idx]
7: BARRIER(CLK_LOCAL_MEM_FENCE)
8: **if** $\lambda_1 = 0 \ \wedge \ v_1 = 0$ **then**
9: dWIo[xidx] $\leftarrow 0.3 \times$ dDH[xidx] $+ 0.3 \times$ dWIo[xidx]
10: dWI[xidx] $\leftarrow$ dWI[xidx] $+$ dWIo[xidx]
11: **end if**
12: **end kernel**

Low-level implementation details

Only Forward Propagation and Weights Adjustment are implemented as OpenCL kernels (both only for the input-to-hidden computation).

Device code See Algorithms 30 and 31.

Host code See Algorithm 32.

Host data structures

- **float** `hI[Ni+1]`: $I_i \rightarrow$ `hI[i]`
- **float** `hH[Nh+1]`: $H_i \rightarrow$ `hH[i]`
- **float** `hO[No+1]`: $O_i \rightarrow$ `hO[i]`
- **float** `hT[No+1]`: $T_i \rightarrow$ `hT[i]`
- **float** `hDH[Nh+1]`: $D_i^H \rightarrow$ `hDH[i]`
- **float** `hDO[No+1]`: $D_i^O \rightarrow$ `hDO[i]`
- **float** `hWI[Ni+1][Nh+1]`: $W_{i,j}^I \rightarrow$ `hWI[i][j]`
- **float** `hWH[Nh+1][No+1]`: $W_{i,j}^H \rightarrow$ `hWH[i][j]`
- **float** `hWIo[Ni+1][Nh+1]`: $W_{i,j}^{I'} \rightarrow$ `hWIo[i][j]`
- **float** `hWHo[Nh+1][No+1]`: $W_{i,j}^{H'} \rightarrow$ `hWHo[i][j]`

Algorithm 32 The Back Propagation host code abstraction.

procedure ALLOCATEMEMORY(Ni, Nh, No) ▷ Allocate host memory and device buffers.
    hI ← ALLOCATEHOSTMEMORY(Ni + 1)
    hH ← ALLOCATEHOSTMEMORY(Nh + 1)
    hO ← ALLOCATEHOSTMEMORY(No + 1)
    hT ← ALLOCATEHOSTMEMORY(No + 1)
    hDH ← ALLOCATEHOSTMEMORY(Nh + 1)
    hDO ← ALLOCATEHOSTMEMORY(No + 1)
    hWI ← ALLOCATEHOSTMEMORY((Ni + 1) × (Nh + 1))
    hWH ← ALLOCATEHOSTMEMORY((Nh + 1) × (No + 1))
    hWIo ← ALLOCATEHOSTMEMORY((Ni + 1) × (Nh + 1))
    hWHo ← ALLOCATEHOSTMEMORY((Nh + 1) × (No + 1))
    hPS ← ALLOCATEHOSTMEMORY(Nh × N_1)
    dI ← ALLOCATEBUFFER(Ni + 1, READ_WRITE)
    dH ← ALLOCATEBUFFER(Nh × N_1, READ_WRITE)
    dDH ← ALLOCATEBUFFER(Nh + 1, READ_WRITE)
    dWI ← ALLOCATEBUFFER((Ni + 1) × (Hh + 1), READ_WRITE)
    dWIo ← ALLOCATEBUFFER((Ni + 1) × (Hh + 1), READ_WRITE)
end procedure

procedure INITIALIZEMEMORY(Ni, Nh, No) ▷ Initialize host memory.
    hI ← RANDOMELEMENTS(Ni + 1)
    hT ← RANDOMELEMENTS(No + 1)
    hWI ← RANDOMELEMENTS((Ni + 1) × (Nh + 1))
    hWH ← RANDOMELEMENTS((Nh + 1) × (No + 1))
    hWIo ← ZEROELEMENTS((Ni + 1) × (Nh + 1))
    hWHo ← ZEROELEMENTS((Nh + 1) × (No + 1))
end procedure

procedure FORWARDPROPOGATION(hH, hO, hWH, Nh, No)
    hO[0] ← 1
    for j **in** 1 : No **do**
        sum ← 0
        for k **in** 0 : Nh **do**
            sum ← sum + hWH[k][j] × hH[k]
        end for
        Ho[j] ← $(1 + e^{\mathrm{sum}})^{-1}$
    end for
end procedure

procedure OUTPUTERROR(hDO, hT, hO, No)
    for j **in** 1 : No **do**
        gen ← hO[j]
        trg ← hT[j]
        hDO[j] ← sum + gen × (1 − gen) × (trg − gen)
    end for
end procedure

Algorithm 32 The Back Propagation host code abstraction (continues).

```
44: procedure HIDDENERROR(hDH, hDO, hWH, hH, Nh, No)
45:     for j in 1 : Nh do
46:         h ← hH[j]
47:         sum ← 0
48:         for j in 1 : No do
49:             sum ← sum + hWH[j][k] × hDO[k]
50:         end for
51:         hDH[j] ← h × (1 − h) × sum
52:     end for
53: end procedure
54: procedure ADJUSTWEIGHTS(hDO, hH, hWH, hWHo, Nh, No)
55:     hH[0] ← 1
56:     for j in 1 : No do
57:         for k in 0 : Nh do
58:             hWHo[k][j] ← 0.3 × hDO[j] × hH[k] + 0.3 × hWHo[k][j]
59:             hWH[k][j] ← hWH[k][j + hWHo[k][j]
60:         end for
61:     end for
62: end procedure
Input: int Ni                                   ▷ Number of input nodes.
Input: int Nh                                   ▷ Number of hidden nodes.
Input: int No                                   ▷ Number of output nodes.
63: ALLOCATEMEMORY(Ni, Nh, No)
64: INITIALIZEMEMORY(N, M, file)
65: bpProgram ← BUILDPROGRAM("bpProgram.cl")
66: bpKernel1 ← BUILDKERNEL(bpProgram, "bpKernel1")
67: bpKernel2 ← BUILDKERNEL(bpProgram, "bpKernel2")
68: COPYTODEVICE(dI, hI, True, ∅)
69: COPYTODEVICE(dWI, hWI, True, ∅)
70: SETKERNELARGUMENTS(bpKernel1, dI, dWI, Nh, dH)
71: ENQUEUEKERNEL(bpKernel1, (16, Ni), (16, 16), ∅)
72: COPYTOHOST(dH, hPS, True, ∅)
73: for j in 1 : Nh do
74:     sum ← 0
75:     for k in 0 : (N1 − 1) do
76:         sum ← sum + hPS[k × Nh + j − 1]
77:     end for
78:     sum ← sum + hWI[0][j]
79:     hH[j] ← (1 + e^sum)^−1
80: end for
```

Algorithm 32 The Back Propagation host code abstraction (continues).

81: FORWARDPROPOGATION(hH, hO, hWH, Nh, No)
82: OUTPUTERROR(hDO, hT, hO, No)
83: HIDDENERROR(hDH, hDO, hWH, hH, Nh, No)
84: ADJUSTWEIGHTS(hDO, hH, hWH, hWHo, Nh, No)
85: COPYTODEVICE(dDH, hDH, **True**, ∅)
86: COPYTODEVICE(dWI, hWI, **True**, ∅)
87: COPYTODEVICE(dWIo, hWIo, **True**, ∅)
88: SETKERNELARGUMENTS(bpKernel2, dDH, dI, dWI, dWIo, Nh)
89: ENQUEUEKERNEL(bpKernel1, (16, Ni), (16, 16), ∅)
90: COPYTOHOST(dI, hI, **True**, ∅) ▷ Unnecessary memory coping.
91: COPYTOHOST(dWI, hWI, **True**, ∅)

Device data structures

- **global float** `dI[Ni+1]`: $I_i \rightarrow$ `dI[i]`
- **global float** `dH[Nh * (`N_1`)]`: $H_i \rightarrow$ `dH[i + Nh * g]`, $0 \leq g < N_1$
- **global float** `dDH[Nh+1]`: $D_i^H \rightarrow$ `dDH[i]`
- **global float** `dWI[(Ni+1) * (Nh+1)]`: $W_{i,j}^I \rightarrow$ `dWI[i * (Nh+1) + j]`
- **global float** `dWIo[(Ni+1) * (Nh+1)]`: $W_{i,j}^{I'} \rightarrow$ `dWI[i * (Nh+1) + j]`
- **local float** `dlI[`Λ_1`]`: $I_i \rightarrow$ `dlI[(i - 1) % (`Λ_1`)]`
- **local float** `dlW[(`Λ_1`) * (`Λ_0`)]`: $W_{i,j}^I \rightarrow$ `dlW[((i - 1) % (`Λ_0`)) * (`Λ_1`) + (j - 1)]`

Input datasets The benchmark uses a randomly generated network with the following parameters:

- The number of input nodes: 65536 (can be changed via command line).
- The number of hidden nodes: 16 (Cannot be changed).
- The number of output nodes: 1 (Cannot be changed).

Partitioned iteration space

- Compute the hidden nodes from the input nodes (device):

$$I_H = \{(\gamma_1, \lambda_0, \lambda_1, i) : 0 < \gamma_1 \leq (\Gamma_1), 0 \leq \lambda_0 < (\Lambda_0), 0 \leq \lambda_1 < (\Lambda_1), 1 \leq i \leq \log_2(\Lambda_1)\}$$

$$\begin{aligned} \Lambda_0 &= N_H = 16 \\ \Lambda_1 &= 16 \\ \Gamma_1 &= N_I/\Lambda_1 \end{aligned}$$

$$f(i,j) \rightarrow ((i-1)/\Lambda_1, j-1, (i-1)\%\Lambda_1, 1:\log_2(\Lambda_1))$$

$$f^{-1}(\gamma_1, \lambda_0, \lambda_1, i) \rightarrow (\gamma_1 \times \Lambda_1 + \lambda_1 + 1, \lambda_0 + 1)$$

- Compute the output nodes from the hidden nodes (host):

$$P_O = \{(i,j) : 0 < i \leq N_O, 0 \leq j \leq N_H\}$$

$$f(i,j) \rightarrow (i,j)$$

$$f^{-1}(i,j) \rightarrow (i,j)$$

- Compute the delta between the target and actual output (host):

$$P_{D_O} = \{(i) : 0 < i \leq N_O\}$$

$$f(i) \rightarrow (i)$$

$$f^{-1}(i) \rightarrow (i)$$

- Compute the delta for the hidden nodes (host):

$$P_{D_H} = \{(i,j) : 0 < i \leq N_H, 0 < j \leq N_O\}$$

$$f(i,j) \rightarrow (i,j)$$

$$f^{-1}(i,j) \rightarrow (i,j)$$

- Update hidden to output weights (host):

$$P_{W_H} = \{(i,j) : 0 \leq i \leq N_H, 0 < j \leq N_O\}$$

$$f(i,j) \rightarrow (i,j)$$

$$f^{-1}(i,j) \rightarrow (i,j)$$

- Update input to hidden weights (device):

$$P_{W_I} = \{(\gamma_1, \lambda_0) : 0 \leq \gamma_1 < N_I, 0 \leq \lambda_0 < N_H\}$$

$$f(i,j) \rightarrow (i-1, j-1)$$

$$f^{-1}(\gamma_1, \lambda_0) \rightarrow (\gamma_1 + 1, \lambda_0 + 1)$$

Device memory access mapping Each work-item executing bpKernel1 performs the following reads and writes:

$$\begin{aligned} G_r(\gamma_1,\lambda_0,\lambda_1,\mathtt{i}) &= \{\mathtt{dI}[\gamma_1+1],\mathtt{dWI}[(\mathtt{Nh}+1)\times(\gamma_1+1)+\lambda_0+1], \\ &\quad \mathtt{dlI}[\lambda_1],\mathtt{dlW}[\lambda_1\times\Lambda_0+\lambda_0],\mathtt{dlW}[(\lambda_1+2^{i-1})\times\Lambda_0+\lambda_0]\} \\ G_w(\gamma_1,\lambda_0,\lambda_1,\mathtt{i}) &= \{\mathtt{dlI}[\lambda_1],\mathtt{dlW}[\lambda_1\times\Lambda_0+\lambda_0],\mathtt{dH}[\gamma_1],\mathtt{dWI}[(\mathtt{Nh}+1)\times(\gamma_1+1)+\lambda_0+1]\} \end{aligned}$$

$\mathtt{dWI}[(\mathtt{Nh}+1)\times(\lambda_1+1)+\lambda_0+1]$ memory access is unnecessary since the dWI buffer would be overwritten by host after the kernel execution is finished.

Each work-item executing the bpKernel2 performs the following reads and writes:

$$\begin{aligned} G_r(\gamma_1,\lambda_0,\lambda_1,\mathtt{i}) &= \{\mathtt{dWIo}[(\mathtt{Nh}+1)\times(\gamma_1+1)+\lambda_0+1],\mathtt{dWI}[(\mathtt{Nh}+1)\times(\gamma_1+1)+\lambda_0+1], \\ &\quad \mathtt{dWIo}[\lambda_0+1],\mathtt{dWI}[\lambda_0+1],\mathtt{dDH}[\lambda_0+1],\mathtt{dI}[\gamma_1+1]\} \\ G_w(\gamma_1,\lambda_0,\lambda_1,\mathtt{i}) &= \{\mathtt{dWIo}[(\mathtt{Nh}+1)\times(\gamma_1+1)+\lambda_0+1],\mathtt{dWI}[(\mathtt{Nh}+1)\times(\gamma_1+1)+\lambda_0+1], \\ &\quad \mathtt{dWIo}[\lambda_0+1],\mathtt{dWI}[\lambda_0+1]\} \end{aligned}$$

Host memory access mapping

- Compute the output nodes from the hidden nodes: for $(i,j)\in I_O$:

$$\begin{aligned} H_r(i,j) &= \{\mathtt{hH}[\mathtt{j}],\mathtt{hWH}[\mathtt{j}][\mathtt{i}]\} \\ H_w(i,j) &= \{\mathtt{hO}[\mathtt{i}]\} \end{aligned}$$

- Compute the delta between the target and actual output: for $(i)\in I_{D_O}$:

$$\begin{aligned} H_r(i) &= \{\mathtt{hO}[\mathtt{i}],\mathtt{hT}[\mathtt{i}]\} \\ H_w(i) &= \{\mathtt{hDO}[\mathtt{i}]\} \end{aligned}$$

- Compute the delta for the hidden nodes: for $(i,j)\in I_{D_H}$:

$$\begin{aligned} H_r(i,j) &= \{\mathtt{hH}[\mathtt{i}],\mathtt{hWH}[\mathtt{i}][\mathtt{j}],\mathtt{hDO}[\mathtt{j}]\} \\ H_w(i,j) &= \{\mathtt{hDH}[\mathtt{i}]\} \end{aligned}$$

- Update hidden to output weights: for $(i,j)\in I_{W_H}$:

$$\begin{aligned} H_r(i,j) &= \{\mathtt{hDO}[\mathtt{j}],\mathtt{hH}[\mathtt{i}],\mathtt{hWHo}[\mathtt{i}][\mathtt{j}],\mathtt{hWH}[\mathtt{i}][\mathtt{j}]\} \\ H_w(i,j) &= \{\mathtt{hWHo}[\mathtt{i}][\mathtt{j}],\mathtt{hWH}[\mathtt{i}][\mathtt{j}]\} \end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- The enqueue time for the bpKernel1 launch.
- The total time for each kernel launch (bpKernel1 and bpKernel2).
- The total time for all kernel launches (the whole computation time).

Validation mechanism The benchmark supports two implementations of the algorithm – parallel (OpenCL based) and sequential. To validate the OpenCL version the outputs of the both implementations can be compared.

Target-specific optimizations

- **Blocking.** The input data is processed in blocks having the number of columns equal to the number of work items per work group. Consecutive work items of the same group access consecutive memory addresses which improves the memory subsystem performance (caches, coalescing).
- **Taking overhead into account.** Although the hidden to output forward propagation and delta calculation can be implemented as OpenCL kernels, the input data size is not sufficient (the overhead for the kernels launches and memory transfer is greater then the benefits from the parallel computations).
- **Tree reduction.** The OpenCL code implements parallel reduction (sum) using a tree-based approach, which requires additional memory barriers.

2.3.2 Breadth-First Search

The benchmark performs breadth-first traversal on an ordered unweighted graph computing the shortest path to each node of the graph from the initial node.

High-level description

Abstract data structures The benchmark operates on the following data objects:

- $G = (V, E)$ is an input graph with vectors of vertices V and edges E.
- C is an output vector of costs ($|V| = |C|$).
- v_0 is the initial vertex.

Computation The algorithm computes the cost of the shortest path for each node according to the following rules:

$$S_0 = \{v_0\}$$

$$S_{\tau+1} = \{v : v \notin S_j, 0 \leq j \leq \tau \wedge \exists u \in S_\tau : (u, v) \in E\}$$

$$C_j = \tau, \forall v_j \in S_\tau$$

The algorithm terminates for τ such that $S_{\tau+1} = \emptyset$.

Iteration spaces and dependences This algorithm requires two iteration spaces:

- Selecting next front ($S_{\tau+1}$):

$$I_S = \{(\tau, v, e) : \tau \geq 0, v \in S_\tau, e \in E^v\}$$

- Marking the selected front as visited:

$$I_M = \{(\tau, v) : \tau \geq 0, v \in S_{\tau+1}\}$$

The computation generates the following dependences:

$$\begin{aligned}
\delta_{I_S} &= I_S : (\{\tau' \in 0 : (\tau - 1)\}, \{v' \in S_{\tau'}\}, \{e' \in E^{v'}\}) \xrightarrow{t} I_S : (\tau, v, e) \wedge \\
&\quad I_S : (\{\tau' \in 0 : (\tau - 1)\}, \{v' \in S_{\tau'}\}, \{e' \in E^{v'}\}) \xrightarrow{o} I_S : (\tau, v, e) \wedge \\
&\quad I_M : (\tau - 1, \{v' \in S_{\tau-1}\}) \xrightarrow{t} I_S : (\tau, v, e) \\
\delta_{I_M} &= I_S : (\{\tau' \in 0 : \tau\}, \{v' \in S_{\tau'}\}, \{e' \in E^{v'}\}) \xrightarrow{t} I_M : (\tau, v)
\end{aligned}$$

Memory access mapping

- Selecting next front: for $(\tau, v, e) \in I_S$:

$$\begin{aligned} M_r(\tau, v, e) &= \{e_h, v = e_t\} \\ M_w(\tau, v, e) &= \{v\} \\ M_{mw}(\tau, v, e) &= \{e_t\} \end{aligned}$$

- Marking the selected front as visited: for $(\tau, v) \in I_M$:

$$\begin{aligned} M_r(\tau, v, e) &= \{v\} \\ M_w(\tau, v, e) &= \{v\} \end{aligned}$$

Low-level implementation details

Each kernel launch computes next front $S_{\tau+1}$ [11].

Device code See Algorithms 33 and 34. In Algorithm 33 different work items can write to the same memory location. But since all vertices in the same front have the same cost these memory writes do not cause data races.

Algorithm 33 The Breadth-First Search (New Front Selection) device code abstraction

```
Input: global Vertex dV[]                                   ▷ Array of all nodes
Input: global int dE[]                                      ▷ Array of all edges
Input/Output: global bool dM[]                              ▷ True for current front vertices.
Input/Output: global bool dU[]                              ▷ True for next front vertices.
Input/Output: global bool dS[]                              ▷ True for visited vertices.
Input/Output: global int dC[]
Input: int N
 1: kernel BFSKERNEL1 (dV, dE, dM, dU, dS, dC, N)
 2:     if (γ < N ∧ dM[γ]) then
 3:         dM[γ] ← false                  ▷ Remove vertex being processed from current front.
 4:         for e in dV[γ].offset : (dV[γ].offset + dV[γ].length − 1) do
 5:             v ← dE[e]
 6:             if dS[v] = false then
 7:                 dC[v] ← dC[γ] + 1
 8:                 dU[v] ← true
 9:             end if
10:         end for
11:     end if
12: end kernel
```

Host code The host code executes bfsKernel1 and bfsKernel2 until no new nodes are marked as visited. See Algorithm 35.

Algorithm 34 The Breadth-First Search (Front Status Update) device code abstraction

Input/Output: `global bool dM[]`
Input/Output: `global bool dU[]`
Input/Output: `global bool dS[]`
Input/Output: `global int dF[]`
Input: `int N`

1: **kernel** BFSKERNEL2 (dM, dU, dS, dF, N)
2: **if** ($\gamma < \mathtt{N} \wedge \mathtt{dU}[\gamma]$) **then**
3: $\mathtt{dM}[\gamma] \leftarrow$ **true**
4: $\mathtt{dS}[\gamma] \leftarrow$ **true**
5: $\mathtt{dU}[\gamma] \leftarrow$ **false**
6: $\mathtt{dF}[0] \leftarrow$ **true**
7: **end if**
8: **end kernel**

Host data structures

- **struct** Vertex:

```
struct Vertex
{
  int offset;
  int length;
};
```

- **struct** Vertex hV[Nv]: $V_i \rightarrow$ hV[i]
- **int** hE[Ne]: $E^{V_i} \rightarrow$ hE[hV[i].offset:(hV[i].offset + hV[i].length - 1)]
- **int** hC[Nv]: $C_i \rightarrow$ hC[i]
- **bool** hM[Nv]: $V_i \rightarrow$ hM[i]
- **bool** hU[Nv]: $V_i \rightarrow$ hU[i]
- **bool** hS[Nv]: $V_i \rightarrow$ hS[i]

The graph is represented by array of vertices `hV[]` and array of edges `hE[]`. Each edge is represented by an integer, which is the index of its tail vertex in array `hV[]`.

Each vertex is represented by a structure, which describes the slice in array `hE[]`, which corresponds to edges for which this vertex is the head.

Device data structures

- **struct** Vertex:

```
struct Vertex
{
  int offset;
  int length;
};
```

- **struct** Vertex dV[Nv]: $V_i \rightarrow$ dV[i]

Algorithm 35 The Breadth-First Search host code abstraction.

```
1:  procedure AllocateMemory(Nv, Ne)          ▷ Allocate host memory and device buffers.
2:      hV ← AllocateHostMemory(Nv)
3:      hE ← AllocateHostMemory(Ne)
4:      hM ← AllocateHostMemory(Nv)
5:      hU ← AllocateHostMemory(Nv)
6:      hS ← AllocateHostMemory(Nv)
7:      hC ← AllocateHostMemory(Nv)
8:      hCref ← AllocateMemory(Nv)
9:      dV ← AllocateBuffer(Nv, READ_ONLY, hV)
10:     dE ← AllocateBuffer(Ne, READ_ONLY, hE)
11:     dM ← AllocateBuffer(Nv, READ_WRITE, hM)
12:     dU ← AllocateBuffer(Nv, READ_WRITE, hU)
13:     dS ← AllocateBuffer(Nv, READ_WRITE, hS)
14:     dC ← AllocateBuffer(Nv, READ_WRITE, hC)
15:     dF ← AllocateBuffer(1, READ_WRITE, hC)
16: end procedure

17: procedure ResetMemory(Nv, v0)             ▷ Reset temporary memory for reuse.
18:     for i in 0 : (Nv − 1) do
19:         hM[i] ← false
20:         hU[i] ← false
21:         hS[i] ← false
22:     end for
23:     hM[v0] ← true
24:     hS[v0] ← true
25: end procedure
26: procedure InitializeMemory(file, Nv, Ne, v0)   ▷ Initialize host memory.
27:     ResetMemory(Nv, v0)
28:     for i in 0 : (Nv − 1) do
29:         hC[i] ← −1
30:         hCref[i] ← −1
31:     end for
32:     hV ← ReadFromFile(file, Nv)
33:     hE ← ReadFromFile(file, Ne)
34:     hC[v0] ← 0
35:     hCref[v0] ← 0
36: end procedure

37: procedure CompareResults(hC, hCref, Nv)
38:     error ← false
39:     for i in 0 : (NV − 1) do
40:         if hC[i] ≠ hCref[i] then
41:             error ← true
42:         end if
43:     end for
44: end procedure
```

Algorithm 35 The Breadth-First Search host code abstraction (continues).

```
1:  procedure BFSHOST(hV, hE, hM, hU, hS, hC, Nv)
2:      repeat
3:          hF ← false
4:          for k in 0 : (Nv − 1) do
5:              if hM[k]) then
6:                  hM[k] ← false
7:                  for i in hV[k].offset : (hV[k].offset + hV[k].length − 1) do
8:                      id ← hE[i]
9:                      if hS[id] = false then
10:                         hC[id] ← hC[k] + 1
11:                         hU[id] ← true
12:                     end if
13:                 end for
14:             end if
15:         end for
16:         for v in 0 : (Nv − 1) do
17:             if hM[v]) then
18:                 hM[v] ← true,hS[v] ← true,hF ← true
19:                 hU[v] ← false
20:             end if
21:         end for
22:     until hF = false
23: end procedure
Input: int Nv                                   ▷ Number of vertices in the input graph.
Input: int Ne                                   ▷ Number of edges in the input graph.
Input: int v0 = 0                               ▷ Initial vertex.
Input: const char * file                        ▷ File with pre-generated graph.
24: ALLOCATEMEMORY(Nv, Ne)
25: INITIALIZEMEMORY(file, Nv, Ne, v0)
26: bfsProgram ← BUILDPROGRAM("bfsProgram.cl")
27: bfsKernel1 ← BUILDKERNEL(bfsProgram, "bfsKernel1")
28: bfsKernel2 ← BUILDKERNEL(bfsProgram, "bfsKernel2")
29: COPYTODEVICE(dV, hV, True, ∅)
30: COPYTODEVICE(dE, hE, True, ∅)
31: COPYTODEVICE(dM, hM, True, ∅)
32: COPYTODEVICE(dU, hU, True, ∅)
33: COPYTODEVICE(dS, hS, True, ∅)
34: COPYTODEVICE(dC, hC, True, ∅) ▷ Probably a bug, since CL_MEM_USE_HOST_PTR is used
    when creating buffers.
```

Algorithm 35 The Breadth-First Search host code abstraction (continues).

35: **repeat**
36: `hF` $\leftarrow$ **false**
37: COPYTODEVICE(`dF`, `hF`, **True**, $\emptyset$)
38: SETKERNELARGUMENTS(`bfsKernel1`, `dV`, `dE`, `dM`, `dU`, `dS`, `dC`, `Nv`)
39: ENQUEUEKERNEL(`bfsKernel1`, (`Nv`, `Nv`), **none**, $\emptyset$)
40: SETKERNELARGUMENTS(`bfsKernel2`, `dM`, `dU`, `dS`, `dF`, `Nv`)
41: ENQUEUEKERNEL(`bfsKernel2`, (`Nv`, `Nv`), **none**, $\emptyset$)
42: COPYTOHOST(`dF`, `hF`, **True**, $\emptyset$)
43: **until** `hF` = **false**
44: COPYTOHOST(`dC`, `hC`, **True**, $\emptyset$)
45: RESETMEMORY(`Nv`, `v0`)
46: BFSHOST(`hV`, `hE`, `hM`, `hU`, `hS`, `hCref`, `Nv`)
47: COMPARERESULTS(`hC`, `hCref`, `Nv`)

- **int** `dE[Ne]`: $E^{V_i} \rightarrow$ `dE[dV[i].offset:(dV[i].offset + dV[i].length - 1)]`
- **int** `dC[Nv]`: $C_i \rightarrow$ `dC[i]`
- **bool** `dM[Nv]`: $V_i \rightarrow$ `dM[i]`
- **bool** `dU[Nv]`: $V_i \rightarrow$ `dU[i]`
- **bool** `dS[Nv]`: $V_i \rightarrow$ `dS[i]`

Input datasets The benchmark uses a set of pre-generated graphs with the following sizes:

- 4096 vertices; 24576 edges
- 65536 vertices; 393216 edges
- 1000000 vertices; 5999970 edges

Partitioned iteration space

- Selecting next front (device):

$$P_S = \{(\tau,\gamma,i) : 0 \leq \tau, 0 \leq \gamma < \Gamma = \texttt{Nv}, \texttt{dV}[\gamma].\texttt{offset} \leq i < \texttt{dV}[\gamma].\texttt{offset} + \texttt{dV}[\gamma].\texttt{length}\}$$

$$f(\tau, v = V_i, e = E^v_j = E_k) \rightarrow (\tau, i, k)$$

$$f^{-1}(\tau,\gamma,i) \rightarrow (\tau, V_\gamma, E_i)$$

- Marking the selected front as visited (device):

$$P_M = \{(\tau,\gamma) : 0 \leq \tau, 0 \leq \gamma < \Gamma = \texttt{Nv}\}$$

$$f(\tau, v = V_i) \rightarrow (\tau, i)$$

$$f^{-1}(\tau, \gamma) \rightarrow (\tau, V_\gamma)$$

- Selecting next front (host):

$$P_S = \{(\tau, l, i) : 0 \leq \tau, 0 \leq l < \texttt{Nv}, \texttt{hV}[l].\texttt{offset} \leq i < \texttt{hV}[l].\texttt{offset} + \texttt{hV}[l].\texttt{length}\}$$

$$f(\tau, v = V_i, e = E_j^v = E_k) \rightarrow (\tau, i, k)$$

$$f^{-1}(\tau, l, i) \rightarrow (\tau, V_l, E_i)$$

- Marking the selected front as visited (host):

$$P_M = \{(\tau, l) : 0 \leq \tau, 0 \leq l < \texttt{Nv}\}$$

$$f(\tau, v = V_i) \rightarrow (\tau, i)$$

$$f^{-1}(\tau, \gamma) \rightarrow (\tau, V_\gamma)$$

Device memory access mapping Each work-item executing the bfsKernel1 performs the following reads and writes:

$$\begin{aligned} G_r(\tau, \gamma, \texttt{i}) &= \{\texttt{dV}[\gamma], \texttt{dE}[\texttt{i}], \texttt{dC}[\gamma], \texttt{dM}[\gamma]\} \\ G_{mr}(\tau, \gamma, \texttt{i}) &= \{\texttt{dS}[\,]\} \\ G_w(\tau, \gamma, \texttt{i}) &= \{\texttt{dM}[\gamma]\} \\ G_{mw}(\tau, \gamma, \texttt{i}) &= \{\texttt{dC}[\,], \texttt{dU}[\,]\} \end{aligned}$$

Each work-item executing the bfsKernel2 performs the following reads and writes:

$$\begin{aligned} G_r(\tau, \gamma) &= \{\texttt{dM}[\gamma]\} \\ G_w(\tau, \gamma) &= \{\texttt{dM}[\gamma], \texttt{dS}[\gamma], \texttt{dU}[\gamma]\} \end{aligned}$$

Host memory access mapping On each iteration on the host corresponding to the next front selection the following memory accesses are performed:

$$\begin{aligned} H_r(\tau, \mathtt{k}, \mathtt{i}) &= \{\mathtt{hV[k]}, \mathtt{hE[i]}, \mathtt{hC[k]}, \mathtt{hM[k]}\} \\ H_{mr}(\tau, \mathtt{k}, \mathtt{i}) &= \{\mathtt{hS[\,]}\} \\ H_w(\tau, \mathtt{k}, \mathtt{i}) &= \{\mathtt{hM[k]}\} \\ H_{mw}(\tau, \mathtt{k}, \mathtt{i}) &= \{\mathtt{hC[\,]}, \mathtt{hU[\,]}\} \end{aligned}$$

On each iteration on the host corresponding to the marking the selected front as visited the following memory accesses are performed:

$$\begin{aligned} H_r(\tau, \mathtt{k}) &= \{\mathtt{dM[k]}\} \\ H_w(\tau, \mathtt{k}) &= \{\mathtt{dM[k]}, \mathtt{dS[k]}, \mathtt{dU[k]}\} \end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- The total time for each kernel launch (computation time for each front).
- The total time for all kernel launches (the whole computation time).

Validation mechanism The benchmark runs two implementations of the algorithm – parallel (using OpenCL) and sequential. In order to validate the OpenCL version the outputs of the both implementations are compared.

Target-specific optimizations

- **Synchronization-free kernels.** Although there are output dependences between iterations performed by different working items in parallel, no synchronization primitives are used. In general such implementation causes data races and invalid results. In given implementation different work items can write to the same memory location, as long as they write the same values, which ensures the correctness of the implementation.

Target-specific optimizations opportunities

- **Reduce number of global memory accesses.** In the current algorithm implementation the same vertex can be marked as processed multiple times (by multiple working items). Although this doesn't cause data races (see explanation above), additional memory accesses can affect the performance of the code. They can be avoided by caching the results in work group local memory and performing the global memory update by a single work item.

2.3.3 CFD Solver

High-level description

The benchmark solves three-dimensional Euler equations for compressible flow [7]:

$$\frac{d}{dt}\int_{\Omega} u d\Omega + \int_{\Gamma} F \cdot n d\Gamma = 0, \tag{2.12}$$

where

$$u = \begin{Bmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \\ \rho e \end{Bmatrix}, F = \begin{Bmatrix} \rho v_x & \rho v_y & \rho v_z \\ \rho v_x^2 + p & \rho v_x v_y & \rho v_x v_z \\ \rho v_y v_x + p & \rho v_y^2 + p & \rho v_y v_z \\ \rho v_z v_x + p & \rho v_z v_y & \rho v_z^2 + p \\ v_x(\rho e + p) & v_y(\rho e + p) & v_z(\rho e + p) \end{Bmatrix}, \tag{2.13}$$

and

$$p = (\gamma - 1)\rho \left[e - \frac{1}{2}\|v\|^2 \right] \tag{2.14}$$

- ρ denotes the density.
- v_x, v_y, v_x denote the x, y, z velocities.
- e denotes the total energy.
- p denotes the pressure.
- γ denotes the ratio of the heats.

The continuous equations (2.12), (2.13) and (2.14) are discretized using a cell-centered, finite-volume scheme of the form:

$$\texttt{vol}_i \frac{du_i}{dt} = R_i = -\sum_{\texttt{faces}} \|s\| \left[\frac{1}{2}(f_i + f_j) - \beta \cdot \lambda_{\max} \cdot (u_i - u_j) \right] \tag{2.15}$$

where

$$f_i = \frac{s}{\|s\|} \cdot F_i, \quad \lambda_{\max} = \|v\| + c \tag{2.16}$$

- $\texttt{vol}_i$ denotes the volume of the i^{th} element.
- s denotes the face normal.
- j is the index of the neighboring element.
- β is the parameter, controlling the amount of artificial viscosity.
- c is the speed of sound.

Abstract data structures

- A is an input N element areas array.
- C is an input $N \times N_{nb}$ elements neighbor array.
- $C_{i,j}$ the j^{th} neighbor of the i^{th} cell.
- U is an intermediate N elements cell array.
- U^f is an input initial value of the U array elements.
- N is an intermediate $N \times N_{nb}$ elements normal array.
- $N_{i,j}$ the j^{th} normal of the i^{th} cell.
- F is an intermediate N element fluxes array.
- S is an intermediate N element step factors array.

Computation The following function are used for computation:

- `ComputeStepFactor`(a,u). Pure function, which computes the step factor.
- `ComputeFlux`(c,n,u,U). Function, which computes the flux. It might read any element of the U array.
- `ComputeTimeStep`(u,s,f,j). Pure function, which computes new value of the U_i variable.

On every iteration k **in** $0:(K_{\max}-1)$ the following computations are performed:

- Compute step factor:

$$S_i = \texttt{ComputeStepFactor}(A_i, U_i), 0 \leq i < N$$

- For each dimension j **in** $0:(\texttt{RK}-1)$:
  - Compute flux:

$$F_i = \texttt{ComputeFlux}(C_i, N_i, U_i, U), 0 \leq i < N$$

  - Compute time step:

$$U_i = \texttt{ComputeTimeStep}(U_i, S_i, F_i, j), 0 \leq i < N$$

Iteration spaces

- Compute step factor:

$$I_S = \{(k,i) : 0 \leq k < k_{\max}, 0 \leq i < N\}$$

- Compute flux:

$$I_F = \{(k,j,i,n) : 0 \leq k < k_{\max}, 0 \leq j < \texttt{RK}, 0 \leq i < N, 0 \leq n < N_{nb}\}$$

- Compute time step:

$$I_T = \{(k,j,i) : 0 \leq k < k_{\max}, 0 \leq j < \texttt{RK}, 0 \leq i < N\}$$

Dependences

$$\begin{aligned}
\delta_{I_S} &= I_T(k-1,\mathtt{RK}-1,i) \xrightarrow{t} I_V : (k,i) \\
\delta_{I_F} &= I_T(k-1,\mathtt{RK}-1,0:(N-1)) \xrightarrow{t} I_F : (k,0,i) \wedge \\
&\quad\ I_T(k,j-1,0:(N-1)) \xrightarrow{t} I_F : (k,j,i) \\
\delta_{I_T} &= I_F(k,j,i) \xrightarrow{t} I_T : (k,j,i)
\end{aligned}$$

Memory access mapping

- Compute step factor:

$$\begin{aligned}
M_r(k,i) &= \{A_i, U_i\} \\
M_w(k,i) &= \{S_i\}
\end{aligned}$$

- Compute flux:

$$\begin{aligned}
M_r(k,j,i,n) &= \{C_{i,n}, N_{i,n}, U_i\} \\
M_{mr}(k,j,i,n) &= \{U\} \\
M_w(k,j,i,n) &= \{F_i\}
\end{aligned}$$

- Compute time step:

$$\begin{aligned}
M_r(k,j,i) &= \{U_i, S_i, F_i\} \\
M_w(k,j,i) &= \{U_i\}
\end{aligned}$$

Low-level implementation details

In low level implementation each cell is represented as five element slice of the $[hd]U$ array. See Host/Device data structures for more details.

Device code See Algorithm 36,Algorithm 37,Algorithm 38,Algorithm 39, Algorithm 40.

Algorithm 36 The CFD device code abstraction (Memset).

Input: `char Val`
Output: `global char dM[]`

```
1: kernel CFDKERNEL1 (Val,dM)
2:     dM[γ] ← Val
3: end kernel
```

Host code See Algorithm 41.

Algorithm 37 The CFD device code abstraction (Initialize).

Input: `global float dUf[]`
Input: `global int N`
Output: `global float dU[]`

1: **kernel** CFDKERNEL2 (dUf, N, dUf)
2: **for** j **in** $0 : (\texttt{Nvar} - 1)$ **do**
3: $\texttt{dU}[\gamma + \texttt{j} \times N] \leftarrow \texttt{dUf}[\texttt{j}]$
4: **end for**
5: **end kernel**

Algorithm 38 The CFD device code abstraction (Compute step factor).

Input: `global float dU[]`
Input: `global float dA[]`
Input: `global int N`
Output: `global float dS[]`

1: **kernel** CFDKERNEL3 (dU, dA, N, dS)
2: $\rho \leftarrow \texttt{dU}[\gamma + \texttt{N} \times 0]$
3: $\texttt{Ux} \leftarrow \texttt{dU}[\gamma + \texttt{N} \times 1]$
4: $\texttt{Uy} \leftarrow \texttt{dU}[\gamma + \texttt{N} \times 2]$
5: $\texttt{Uz} \leftarrow \texttt{dU}[\gamma + \texttt{N} \times 3]$
6: $\texttt{Ue} \leftarrow \texttt{dU}[\gamma + \texttt{N} \times 4]$
7: $\texttt{dS}[\gamma] \leftarrow$COMPUTESTEPFACTOR($\texttt{dA}[\gamma], \rho, \texttt{Ux}, \texttt{Uy}, \texttt{Uz}, \texttt{Ue}$)
8: **end kernel**

Algorithm 39 The CFD device code abstraction (Compute flux).

Input: `global float dC[]`
Input: `global float dN[]`
Input: `global float dU[]`
Input: `global int N`
Output: `global float dF[]`

1: **kernel** CFDKERNEL4 (dC, dN, dU, N, dF)
2: $\mathtt{U}_\rho \leftarrow \mathtt{dU}[\gamma + \mathtt{N} \times 0]$
3: $\mathtt{U}_x \leftarrow \mathtt{dU}[\gamma + \mathtt{N} \times 1]$
4: $\mathtt{U}_y \leftarrow \mathtt{dU}[\gamma + \mathtt{N} \times 2]$
5: $\mathtt{U}_z \leftarrow \mathtt{dU}[\gamma + \mathtt{N} \times 3]$
6: $\mathtt{U}_e \leftarrow \mathtt{dU}[\gamma + \mathtt{N} \times 4]$ ▷ Extract cell data.
7: $\mathtt{f}_\rho \leftarrow$GETINITIALFLUXDENSITY()
8: $\mathtt{f}_x \leftarrow$GETINITIALFLUXMOMENTUMX()
9: $\mathtt{f}_y \leftarrow$GETINITIALFLUXMOMENTUMY()
10: $\mathtt{f}_z \leftarrow$GETINITIALFLUXMOMENTUMZ()
11: $\mathtt{f}_e \leftarrow$GETINITIALFLUXENERGY() ▷ Get initial flux values.
12: **for** j **in** $0 : (\mathtt{Nnb} - 1)$ **do**
13: $\mathtt{idx} \leftarrow \mathtt{dC}[\gamma + \mathtt{j} \times \mathtt{N}]$ ▷ Get neighbor index.
14: $\mathtt{N}_x \leftarrow \mathtt{dN}[\gamma + \mathtt{N} \times (\mathtt{j} + 0 \times \mathtt{Nnb})]$
15: $\mathtt{N}_y \leftarrow \mathtt{dN}[\gamma + \mathtt{N} \times (\mathtt{j} + 1 \times \mathtt{Nnb})]$
16: $\mathtt{N}_z \leftarrow \mathtt{dN}[\gamma + \mathtt{N} \times (\mathtt{j} + 2 \times \mathtt{Nnb})]$ ▷ Extract normal data.
17: $\mathtt{C}_\rho \leftarrow \mathtt{dU}[\mathtt{idx} + \mathtt{N} \times 0]$
18: $\mathtt{C}_x \leftarrow \mathtt{dU}[\mathtt{idx} + \mathtt{N} \times 1]$
19: $\mathtt{C}_y \leftarrow \mathtt{dU}[\mathtt{idx} + \mathtt{N} \times 2]$
20: $\mathtt{C}_z \leftarrow \mathtt{dU}[\mathtt{idx} + \mathtt{N} \times 3]$
21: $\mathtt{C}_e \leftarrow \mathtt{dU}[\mathtt{idx} + \mathtt{N} \times 4]$ ▷ Extract neighboring cell data.
22: $\mathtt{f}_\rho \leftarrow$UPDATEFLUXDENSITY($\mathtt{U}_\rho, \mathtt{U}_x, \mathtt{U}_y, \mathtt{U}_z, \mathtt{U}_e, \mathtt{C}_\rho, \mathtt{C}_x, \mathtt{C}_y, \mathtt{C}_z, \mathtt{C}_e, \mathtt{N}_x, \mathtt{N}_y, \mathtt{N}_z$)
23: $\mathtt{f}_x \leftarrow$UPDATEFLUXMOMENTUMX($\mathtt{U}_\rho, \mathtt{U}_x, \mathtt{U}_y, \mathtt{U}_z, \mathtt{U}_e, \mathtt{C}_\rho, \mathtt{C}_x, \mathtt{C}_y, \mathtt{C}_z, \mathtt{C}_e, \mathtt{N}_x, \mathtt{N}_y, \mathtt{N}_z$)
24: $\mathtt{f}_y \leftarrow$UPDATEFLUXMOMENTUMY($\mathtt{U}_\rho, \mathtt{U}_x, \mathtt{U}_y, \mathtt{U}_z, \mathtt{U}_e, \mathtt{C}_\rho, \mathtt{C}_x, \mathtt{C}_y, \mathtt{C}_z, \mathtt{C}_e, \mathtt{N}_x, \mathtt{N}_y, \mathtt{N}_z$)
25: $\mathtt{f}_z \leftarrow$UPDATEFLUXMOMENTUMZ($\mathtt{U}_\rho, \mathtt{U}_x, \mathtt{U}_y, \mathtt{U}_z, \mathtt{U}_e, \mathtt{C}_\rho, \mathtt{C}_x, \mathtt{C}_y, \mathtt{C}_z, \mathtt{C}_e, \mathtt{N}_x, \mathtt{N}_y, \mathtt{N}_z$)
26: $\mathtt{f}_e \leftarrow$UPDATEFLUXENERGY($\mathtt{U}_\rho, \mathtt{U}_x, \mathtt{U}_y, \mathtt{U}_z, \mathtt{U}_e, \mathtt{C}_\rho, \mathtt{C}_x, \mathtt{C}_y, \mathtt{C}_z, \mathtt{C}_e, \mathtt{N}_x, \mathtt{N}_y, \mathtt{N}_z$) ▷ Update flux values.
27: **end for**
28: $\mathtt{dF}[\gamma + 0 \times N] \leftarrow \mathtt{f}_\rho$
29: $\mathtt{dF}[\gamma + 1 \times N] \leftarrow \mathtt{f}_x$
30: $\mathtt{dF}[\gamma + 2 \times N] \leftarrow \mathtt{f}_y$
31: $\mathtt{dF}[\gamma + 3 \times N] \leftarrow \mathtt{f}_z$
32: $\mathtt{dF}[\gamma + 4 \times N] \leftarrow \mathtt{f}_e$
33: **end kernel**

Algorithm 40 The CFD device code abstraction (Compute time step).

Input: `global float dUold[]`
Input: `global float dS[]`
Input: `global float dF[]`
Input: `global int N`
Input: `global int j`
Output: `global float dU[]`

1: **kernel** CFDKERNEL5 (dUold, dS, dF, Nj, dU)
2: factor $\leftarrow$ dF[γ]/(RK + 1 − j)
3: dU[γ + N × 0] $\leftarrow$ dUold[γ + N × 0] + factor × dF[γ + N × 0]
4: dU[γ + N × 1] $\leftarrow$ dUold[γ + N × 1] + factor × dF[γ + N × 1]
5: dU[γ + N × 2] $\leftarrow$ dUold[γ + N × 2] + factor × dF[γ + N × 2]
6: dU[γ + N × 3] $\leftarrow$ dUold[γ + N × 3] + factor × dF[γ + N × 3]
7: dU[γ + N × 4] $\leftarrow$ dUold[γ + N × 4] + factor × dF[γ + N × 4]
8: **end kernel**

Host data structures

- **float** `hA[N]`: $A_i \rightarrow$ `dA[i]`
- **float** `hN[N * Nnb * 3]`: $N_{i,j} \rightarrow$ `hN[(i + (j + 0 * Nnb) * N):(i + (j + 2 * Nnb) * N):(N * Nnb)]`
  - $N_{i,j}.x \rightarrow$ `hN[i + (j + 0 * Nnb) * N]`
  - $N_{i,j}.y \rightarrow$ `hN[i + (j + 1 * Nnb) * N]`
  - $N_{i,j}.z \rightarrow$ `hN[i + (j + 2 * Nnb) * N]`
- **int** `hC[N * Nnb]`: $C_{i,j} \rightarrow$ `hC[i + j ( N]`
- **float** `hUf[5]`: $\mathrm{U^f} \rightarrow$ `hUf`
  - $U^f.\rho \rightarrow$ `hUf[0]`
  - $U^f.x \rightarrow$ `hUf[1]`
  - $U^f.y \rightarrow$ `hUf[2]`
  - $U^f.z \rightarrow$ `hUf[3]`
  - $U^f.e \rightarrow$ `hUf[4]`

Device data structures

- **global float** `dA[N]`: $A_i \rightarrow$ `dA[i]`
- **global float** `dN[N * Nnb * 3]`: $N_{i,j} \rightarrow$ `dN[(i + (j + 0 * Nnb) * N):(i + (j + 2 * Nnb) * N):(N * Nnb)]`
  - $N_{i,j}.x \rightarrow$ `dN[i + (j + 0 * Nnb) * N]`
  - $N_{i,j}.y \rightarrow$ `dN[i + (j + 1 * Nnb) * N]`
  - $N_{i,j}.z \rightarrow$ `dN[i + (j + 2 * Nnb) * N]`

Algorithm 41 The CFD host code abstraction.

```
1: procedure AllocateMemory(N, Nnb)        ▷ Allocate host memory and device buffers.
2:     hA ← AllocateHostMemory(N)
3:     hN ← AllocateHostMemory(N × Nnb × 3)
4:     hC ← AllocateHostMemory(N × Nnb)
5:     hUf ← AllocateHostMemory(5)
6:     dA ← AllocateBuffer(N, Nnb, READ_WRITE)
7:     dN ← AllocateBuffer(N × Nnb × 3, READ_WRITE)
8:     dF ← AllocateBuffer(N × 5, READ_WRITE)
9:     dC ← AllocateBuffer(N × Nnb, READ_WRITE)
10:    dS ← AllocateBuffer(N, READ_WRITE)
11:    dU ← AllocateBuffer(N × 5, READ_WRITE)
12:    dUold ← AllocateBuffer(N × 5, READ_WRITE)
13:    dUf ← AllocateBuffer(5, READ_WRITE)
14: end procedure
15: procedure InitializeMemory(file)        ▷ Initialize host memory.
16:    hA ←ReadFromFile(file)
17:    hN ←ReadFromFile(file)
18:    hC ←ReadFromFile(file)
19:    hUf ←GenerateInitial( )
20: end procedure
Input: int N                                ▷ Number of cells.
Input: int Ni                               ▷ Number of iterations.
Input: const char * file                    ▷ File with pre generated data.
21: Nnb ← 4
22: AllocateMemory(N, Nnb)
23: InitializeMemory(file)
24: CopyToDevice(dA, hA, True, ∅)
25: CopyToDevice(dC, hC, True, ∅)
26: CopyToDevice(dN, hN, True, ∅)
27: CopyToDevice(dUf, hUf, True, ∅)
28: cfdProgram ← BuildProgram("cfdProgram.cl")
29: cfdKernel1 ← BuildKernel(cfdProgram, "cfdKernel1")
30: cfdKernel2 ← BuildKernel(cfdProgram, "cfdKernel2")
31: cfdKernel3 ← BuildKernel(cfdProgram, "cfdKernel3")
32: cfdKernel4 ← BuildKernel(cfdProgram, "cfdKernel4")
33: cfdKernel5 ← BuildKernel(cfdProgram, "cfdKernel5")
34: SetKernelArguments(cfdKernel2, N, dU, dUf)
35: EnqueueKernel(sradKernel2, (N), (192), ∅)
36: SetKernelArguments(cfdKernel2, N, dUold, dUf)
37: EnqueueKernel(sradKernel2, (N), (192), ∅)
38: SetKernelArguments(cfdKernel2, N, dF, dUf)
39: EnqueueKernel(sradKernel2, (N), (192), ∅)
40: SetKernelArguments(cfdKernel1, 0, dS)
41: EnqueueKernel(sradKernel2, (N × sizeof(float)), (192), ∅)
```

Algorithm 41 The CFD host code abstraction (continues).

42: **for** `i` **in** $0 : (\texttt{Ni} - 1)$ **do**
43: `RK` $\leftarrow 3$
44: CopyBuffer(`dU`, `dUold`, $\emptyset$)
45: SetKernelArguments(`cfdKernel3`, `dU`, `dA`, `N`, `dS`)
46: EnqueueKernel(`sradKernel3`, (`N`), (192), $\emptyset$)
47: **for** `j` **in** $0 : (\texttt{RK} - 1)$ **do**
48: SetKernelArguments(`cfdKernel4`, `dC`, `dN`, `dU`, `N`, `dF`)
49: EnqueueKernel(`sradKernel4`, (`N`), (192), $\emptyset$)
50: SetKernelArguments(`cfdKernel5`, `dUold`, `dS`, `dF`, `Nj`, `dU`)
51: EnqueueKernel(`sradKernel5`, (`N`), (192), $\emptyset$)
52: **end for**
53: **end for**

- **global float** `dF[N * 5]`: $F_i \rightarrow$ `dF[(i + 0 * N):(i + 4 * N):N]`
  - $F_i.\rho \rightarrow$ `dF[i + 0 * N]`
  - $F_i.x \rightarrow$ `dF[i + 1 * N]`
  - $F_i.y \rightarrow$ `dF[i + 2 * N]`
  - $F_i.z \rightarrow$ `dF[i + 3 * N]`
  - $F_i.e \rightarrow$ `dF[i + 4 * N]`
- **global int** `dC[N * Nnb]`: $C_{i,j} \rightarrow$ `dC[i + j ( N]`
- **global float** `dS[N]`: $S_i \rightarrow$ `dS[i]`
- **global float** `dUf[5]`: $U^f \rightarrow$ `dUf`
  - $U^f.\rho \rightarrow$ `dUf[0]`
  - $U^f.x \rightarrow$ `dUf[1]`
  - $U^f.y \rightarrow$ `dUf[2]`
  - $U^f.z \rightarrow$ `dUf[3]`
  - $U^f.e \rightarrow$ `dUf[4]`
- **global float** `dUold[N * 5]`: $U_i \rightarrow$ `dUold[(i + 0 * N):(i + 4 * N):N]`
  - $U_i.\rho \rightarrow$ `dUold[i + 0 * N]`
  - $U_i.x \rightarrow$ `dUold[i + 1 * N]`
  - $U_i.y \rightarrow$ `dUold[i + 2 * N]`
  - $U_i.z \rightarrow$ `dUold[i + 3 * N]`
  - $U_i.e \rightarrow$ `dUold[i + 4 * N]`
- **global float** `dU[N * 5]`: $U_i \rightarrow$ `dU[(i + 0 * N):(i + 4 * N):N]`

- $U_i.\rho \rightarrow$ `dU[i + 0 * N]`
- $U_i.x \rightarrow$ `dU[i + 1 * N]`
- $U_i.y \rightarrow$ `dU[i + 2 * N]`
- $U_i.z \rightarrow$ `dU[i + 3 * N]`
- $U_i.e \rightarrow$ `dU[i + 4 * N]`

Input datasets The benchmark uses pre-generated input datasets with following number of elements (N): 97046, 193474, 232536.

For all input data sets the number of neighboring elements (N_{nb}) is 4.

Partitioned iteration space

- Compute step factor (cfdKernel3):

$$P_S = \{(k,\gamma) : 0 \leq k < N_i, 0 \leq \gamma < \Gamma = N\}$$

$$f(k,i) \rightarrow (k,i)$$

$$f^{-1}(k,\gamma) \rightarrow (k,\gamma)$$

- Compute flux (cfdKernel4):

$$P_F = \{(k,j,\gamma,n) : 0 \leq k < N_i, 0 \leq j < \texttt{RK}, 0 \leq \gamma < \Gamma = N, 0 \leq n < N_{nb}\}$$

$$f(k,j,i,n) \rightarrow (k,j,i,n)$$

$$f^{-1}(k,j,\gamma,n) \rightarrow (k,j,\gamma,n)$$

- Compute time step (cfdKernel5):

$$P_T = \{(k,j,\gamma) : 0 \leq k < N_i, 0 \leq j < \texttt{RK}, 0 \leq \gamma < \Gamma = N\}$$

$$f(k,j,i) \rightarrow (k,j,i)$$

$$f^{-1}(k,j,\gamma) \rightarrow (k,j,\gamma)$$

Device memory access mapping

- Compute step factor (cfdKernel3):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dA}[\gamma], \texttt{dU}[\gamma+\texttt{N}\times 0], \texttt{dU}[\gamma+\texttt{N}\times 1], \texttt{dU}[\gamma+\texttt{N}\times 2], \texttt{dU}[\gamma+\texttt{N}\times 3], \texttt{dU}[\gamma+\texttt{N}\times 4]\} \\ G_w(k,\gamma) &= \{\texttt{dS}[\gamma]\} \end{aligned}$$

- Compute flux (cfdKernel4):

$$\begin{aligned} G_r(k,j,\gamma,n) &= \{\texttt{dN}[\gamma+\texttt{N}\times(\texttt{j}+0\times\texttt{Nnb})], \texttt{dN}[\gamma+\texttt{N}\times(\texttt{j}+1\times\texttt{Nnb})], \texttt{dN}[\gamma+\texttt{N}\times(\texttt{j}+2\times\texttt{Nnb})], \\ &\quad \texttt{dC}[\gamma+\texttt{j}\times\texttt{N}], \texttt{dU}[\gamma+\texttt{N}\times 0], \texttt{dU}[\gamma+\texttt{N}\times 1], \texttt{dU}[\gamma+\texttt{N}\times 2], \texttt{dU}[\gamma+\texttt{N}\times 3], \texttt{dU}[\gamma+\texttt{N}\times 4]\} \\ G_{mr}(k,j,\gamma,n) &= \{\texttt{dU}\} \\ G_w(k,j,\gamma,n) &= \{\texttt{dF}[\gamma+0\times N], \texttt{dF}[\gamma+1\times N], \texttt{dF}[\gamma+2\times N], \texttt{dF}[\gamma+3\times N], \texttt{dF}[\gamma+4\times N]\} \end{aligned}$$

- Compute time step (cfdKernel5):

$$\begin{aligned} G_r(k,j,\gamma) &= \{\texttt{dF}[\gamma+0\times N], \texttt{dF}[\gamma+1\times N], \texttt{dF}[\gamma+2\times N], \texttt{dF}[\gamma+3\times N], \texttt{dF}[\gamma+4\times N], \\ &\quad \texttt{dUold}[\gamma+\texttt{N}\times 0], \texttt{dUold}[\gamma+\texttt{N}\times 1], \texttt{dUold}[\gamma+\texttt{N}\times 2], \\ &\quad \texttt{dUold}[\gamma+\texttt{N}\times 3], \texttt{dUold}[\gamma+\texttt{N}\times 4], \texttt{dS}[\gamma]\} \\ G_w(k,j,\gamma) &= \{\texttt{dU}[\gamma+\texttt{N}\times 0], \texttt{dU}[\gamma+\texttt{N}\times 1], \texttt{dU}[\gamma+\texttt{N}\times 2], \texttt{dU}[\gamma+\texttt{N}\times 3], \texttt{dU}[\gamma+\texttt{N}\times 4]\} \end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- The total execution time.
- The OpenCL context creation time.
- The OpenCL context release time.
- Device memory allocation time.
- Device memory free time.
- The time to transfer data from host to device.
- The time to transfer data from device to host.
- The time to transfer data from device to device.
- The OpenCL kernels total execution time.
- The OpenCL kernels total compilation time.

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can write the solution to the file, which can be validated by external tools (not supplied).

Target-specific optimizations The following target-specific optimization are used in the algorithm implementation:

- **Coalescing-friendly memory access (probably CUDA only).** In order to achieve coalesced memory access the input data buffer is logically transposed. For CUDA devices this transformation makes the data access in kernel coalesced. In other hand this greatly reduces the data locality. Although for CUDA devices the advantages of the coalesced memory access overrides the disadvantages of the non-localized memory access, this might not be the same for OpenCL. Since OpenCL standard doesn't require the consecutive memory accesses within threads of the same working group to be coalesced, this transformation might cause a performance degradation.

Target-specific optimizations opportunities

- **Buffer swapping elimination.** Device to device memory coping (dU to $dUold$) can be eliminated by either using only one buffer (as described in height level description) or using the logical I/O buffers alternation (see PathFinder target specific optimizations section for more details).

2.3.4 Gaussian Elimination

The benchmark uses Gaussian Elimination to solve a linear system of equations $A\vec{x} = \vec{b}$:

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,n-1} \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix} \tag{2.17}$$

High-level description

Abstract data structures The benchmark operates on the following floating point data objects:

- A is an input $n \times n$ matrix of coefficients.
- M is an intermediate $n \times n$ matrix of multipliers.
- $\vec{b}$ is an input n-element vector of constant terms.
- $\vec{x}$ is an output n-element vector of solutions.

Computation The algorithm consists of two steps:

1. Forward Elimination: Matrix A is reduced to triangular form by elementary rows transformations using intermediate matrix M; the same transformations are applied to vector $\vec{b}$ to preserve the original solution: $A\vec{x} = \vec{b} \Rightarrow A'\vec{x} = \vec{b}'$.

$$\begin{pmatrix} A'_{0,0} & A'_{0,1} & \cdots & A'_{0,n-1} \\ 0 & A'_{1,1} & \cdots & A'_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & A'_{n-1,n-1} \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b'_0 \\ b'_1 \\ \vdots \\ b'_{n-1} \end{pmatrix} \tag{2.18}$$

---

for k **in** $0 : (n-2)$ **do**
  $M_{i,k} = A_{i,k} / A_{k,k}, \quad k < i < n$ ▷ M: Compute multiplier matrix
  $A_{i,j} = A_{i,j} - M_{i,k} \times A_{k,j}, \quad k < i < n, k \leq j < n$ ▷ A: Modify coefficient matrix
  $b_i = b_i - M_{i,k} \times b_k, \quad k < i < n$ ▷ b: Modify constant term vector
end for

This step requires $O(n^3)$ operations.

2. Back Substitution: Triangular system (2.18) is solved from x_{n-1} to x_0:

   This step requires $O(n^2)$ operations.

for i **in** $(n-1):0$ **do**
$x_i = \left(b'_i - \sum_{i<j<n} A'_{i,j} x_j\right) / A'_{i,i}$
end for

Iteration spaces and dependences This algorithm requires several iteration spaces:

- Compute multiplier matrix:

$$I_M = \{(k,i) : 0 \leq k < n-1, k < i < n\}$$

$$I_M : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 1 & -1 \end{pmatrix} \times \begin{pmatrix} k \\ i \end{pmatrix} \leq \begin{pmatrix} n-2 \\ n-1 \\ 0 \\ -1 \end{pmatrix}$$

- Modify coefficient matrix:

$$I_A = \{(k,i,j) : 0 \leq k < n-1, k < i < n, k \leq j < n\}$$

$$I_A : \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} k \\ i \\ j \end{pmatrix} \leq \begin{pmatrix} n-2 \\ n-1 \\ n-1 \\ -1 \\ -1 \\ 0 \end{pmatrix}$$

- Modify constant term vector:

$$I_b = \{(k,i) : 0 \leq k < n-1, k < i < n\}$$

$$I_b : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 1 & -1 \end{pmatrix} \times \begin{pmatrix} k \\ i \end{pmatrix} \leq \begin{pmatrix} n-2 \\ n-1 \\ 0 \\ -1 \end{pmatrix}$$

- Compute solution (Back Substitution):

$$I_x = \{(i,j) : 0 \leq i < n, i < j < n\}$$

$$I_x : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 1 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} n-1 \\ n-1 \\ 0 \\ -1 \end{pmatrix}$$

$$I = I_A \cup I_M \cup I_b \cup I_x$$

The computation generates the following dependences:

$$\begin{aligned}
\delta_{I_A} &= I_M : (j,i) \xrightarrow{a} I_A : (k,i,j) \wedge \\
&\quad I_M : (i,(i+1:n)) \xrightarrow{a} I_A : (k,i,i) \wedge \\
&\quad I_M : (k,i) \xrightarrow{t} I_A : (k,i,j) \wedge \\
&\quad I_A : (k-1,i,j) \xrightarrow{t} I_A : (k,i,j) \wedge \\
&\quad I_A : (k-1,k,j) \xrightarrow{t} I_A : (k,i,j) \wedge \\
&\quad I_A : (k-1,i,j) \xrightarrow{a} I_A : (k,i,j) \\
\delta_{I_b} &= I_M : (k,i) \xrightarrow{t} I_b : (k,i) \wedge \\
&\quad I_b : (k-1,i) \xrightarrow{t} I_b : (k,i) \wedge \\
&\quad I_b : (k-1,k) \xrightarrow{t} I_b : (k,i) \wedge \\
&\quad I_b : (k-1,i) \xrightarrow{a} I_b : (k,i) \wedge \\
\delta_{I_x} &= I_x : (j,(j+1:n)) \xrightarrow{t} I_x : (i,j) \wedge \\
&\quad I_A : (i,i,j) \xrightarrow{t} I_x : (i,j) \wedge \\
&\quad I_b : (i,i+1:n) \xrightarrow{t} I_x : (i,j) \wedge
\end{aligned}$$

Memory access mapping

- Compute multiplier matrix: for $(k,i) \in I_M$:

$$\begin{aligned}
M_r(k,i) &= \{A_{i,k}, A_{k,k}\} \\
M_w(k,i) &= \{M_{i,k}\}
\end{aligned}$$

- Modify coefficient matrix: for $(k,i,j) \in I_A$:

$$\begin{aligned}
M_r(k,i,j) &= \{A_{k,j}, A_{i,j}, M_{i,k}\} \\
M_w(k,i,j) &= \{A_{i,j}\}
\end{aligned}$$

- Modify constant vector: for $(k,i) \in I_b$:

$$\begin{aligned}
M_r(k,i) &= \{b_k, b_i, M_{i,k}\} \\
M_w(k,i) &= \{b_i\}
\end{aligned}$$

- Compute solution (Back Substitution): for $(i,j) \in I_x$:

$$\begin{aligned}
M_r(i,j) &= \{x_j, A_{i,j}\} \\
M_r(i) &= \{b_i, A_{i,i}\} \\
M_w(i,j) &= \{x_i\}
\end{aligned}$$

Low-level implementation details

Only Forward Elimination (step 1) is implemented using OpenCL. Back Substitution (step 2) is implemented using a sequential host loop.

Device code Each kernel launch performs one iteration of forward elimination. See Algorithms 42 and 43.

Algorithm 42 The Gaussian Elimination (Coefficient computations) device code abstraction.

Input: `global float dA[]`
Input: `int n`
Input: `int k`
Output: `global float dM[]`

```
1: kernel GEKERNEL1 (dM, dA, n, k)
2:     i ← γ + k + 1
3:     if i < n then
4:         dM[n × i + k] ← dA[n × i + k]/dA[n × k + k]
5:     end if
6: end kernel
```

Algorithm 43 The Gaussian Elimination (Forward Elimination) device code abstraction.

Input/Output: `global float dA[]`
Input/Output: `global float db[]`
Input: `global float dM[]`
Input: `int n`
Input: `int k`

```
1: kernel GEKERNEL2 (dM, dA, db, n, k)
2:     i ← γ0 + k + 1
3:     j ← γ1 + k
4:     if (i < n) and (j < n) then
5:         dA[n × i + j] ← dA[n × i + j] − dA[n × k + j] × dM[n × i + k]
6:         if γ1 = 0 then
7:             db[i] ← db[i] − dM[n × i + k] × db[k]
8:         end if
9:     end if
10: end kernel
```

Host code The outer loop is executed on host, while the inner loops are executed as kernels on device. See Algorithm 44.

Host data structures

- **float** `hA[n * n]`: $A_{y,x} \rightarrow$ `hA[y * n + x]`
- **float** `hM[n * n]`: $M_{y,x} \rightarrow$ `hM[y * n + x]`

Algorithm 44 The Gaussian Elimination (Forward Elimination) host code abstraction.

```
1:  procedure AllocateMemory(n)                          ▷ Allocate host memory and device buffers.
2:      hA ← AllocateHostMemory(n^2)
3:      hM ← AllocateHostMemory(n^2)
4:      hb ← AllocateHostMemory(n)
5:      hx ← AllocateHostMemory(n)
6:      dA ← AllocateBuffer(n^2, READ_WRITE)
7:      dM ← AllocateBuffer(n^2, READ_WRITE)
8:      db ← AllocateBuffer(n, READ_WRITE)
9:  end procedure

10: procedure InitializeMemory(file)                     ▷ Initialize host memory.
11:     hA ← ReadFromFile(file, n^2)
12:     hb ← ReadFromFile(file, n)
13: end procedure

14: procedure BackSubstitution(hA, hb, hx, n)
15:     for i in (n − 1) : 0 do
16:         hx[i] ← hb[i]
17:         for j in (n − 1) : (i + 1) do
18:             hx[i] ← hx[i] − hA[i × n + j] ∗ hx[j]
19:         end for
20:         hx[i] ← hx[i]/hA[i × n + i]
21:     end for
22: end procedure

Input: int n                                             ▷ Coefficient matrix number of rows (columns).
Input: const char * file                                 ▷ File with pre-generated coefficient matrix and constant term.
23: AllocateMemory(n)
24: InitializeMemory(file)
25: geProgram ← BuildProgram("geProgram.cl")
26: geKernel1 ← BuildKernel(geProgram, "geKernel1")
27: geKernel2 ← BuildKernel(geProgram, "geKernel2")
28: CopyToDevice(dA, hA, True, ∅)
29: CopyToDevice(db, hb, True, ∅)
30: CopyToDevice(dM, hM, True, ∅)                        ▷ In-order execution model is used, so no additional
    synchronization required.
31: for k in 0 : (n − 2) do
32:     SetKernelArguments(geKernel1, dM, dA, n, k)
33:     EnqueueKernel(geKernel1, (n), none, ∅)
34:     SetKernelArguments(geKernel2, dM, dA, db, n, k)
35:     EnqueueKernel(geKernel2, (n, n), none, ∅)
36: end for
37: CopyToHost(dA, hA, True, ∅)
38: CopyToHost(db, hb, True, ∅)
39: CopyToHost(dM, hM, True, ∅)
40: BackSubstitution(hA, hb, hx, n)
```

- **float** `hb[n]`: $b_y \rightarrow$ `hb[y]`
- **float** `hx[n]`: $x_y \rightarrow$ `hx[y]`

Device data structures

- **global float** `dA[n * n]`: $A_{y,x} \rightarrow$ `dA[y * n + x]`
- **global float** `dM[n * n]`: $M_{y,x} \rightarrow$ `dM[y * n + x]`
- **global float** `db[n]`: $b_y \rightarrow$ `db[y]`
- **global float** `dx[n]`: $x_y \rightarrow$ `dx[y]`

Input datasets The benchmark uses pre-generated matrices of single precision floating point numbers in range $[-0.9, 0.9]$ with the following values of n: $\{3, 4, 16, 208, 1024\}$. It also contains a script for generating matrices and vectors for any n.

Partitioned iteration space

- Intermediate multiplier matrix calculation (geKernel1):

$$P_M = \{(k,\gamma) : 0 \leq k < n, 0 \leq \gamma < (\Gamma = n) - k - 1\}$$

$$f(k,i) \rightarrow (k, i-k-1)$$

$$f^{-1}(k,\gamma) \rightarrow (k, k+\gamma+1)$$

- Coefficient matrix and constant vector modification (geKernel2):

$$P_A = \{(k,\gamma_0,\gamma_1) : 0 \leq k < n, 0 \leq \gamma_1 < (\Gamma_0 = n) - k - 1, 0 \leq \gamma_1 < (\Gamma_1 = n) - k\}$$

$$f(k,i,j) \rightarrow (k, i-k-1, j-k)$$

$$f^{-1}(k,\gamma_0,\gamma_1) \rightarrow (k, k+\gamma_0+1, k+\gamma_1)$$

- Back Substitution (host code):

$$P_x = \{(i,j) : 0 \leq i < n, i < j < n\}$$

$$f(i,j) \rightarrow (i,j)$$

$$f^{-1}(i,j) \rightarrow (i,j)$$

Device memory access mapping Each work-item executing the geKernel1 performs the following reads and writes:

$$\begin{aligned} G_r(\mathtt{k},\gamma) &= \{\mathtt{dA}[\mathtt{n}\times(\gamma+\mathtt{k}+1)+\mathtt{k}], \mathtt{dA}[\mathtt{n}\times\mathtt{k}+\mathtt{k}]\} \\ G_w(\mathtt{k},\gamma) &= \{\mathtt{dM}[\mathtt{n}\times(\gamma+\mathtt{k}+k)+\mathtt{k}]\} \end{aligned}$$

Each work-item executing the geKernel2 performs the following reads and writes:

$$\begin{aligned} G_r(\mathtt{k},\gamma_0,\gamma_1) &= \{\mathtt{dM}[\mathtt{n}\times(\gamma_0+\mathtt{k}+1)+\mathtt{k}], \mathtt{dA}[\mathtt{n}\times\mathtt{k}+\mathtt{k}+\gamma_1], \mathtt{dA}[\mathtt{n}\times(\gamma_0+\mathtt{k}+1)+\mathtt{k}+\gamma_1]\} \\ G_r(\mathtt{k},\gamma_0,0) &= \{\mathtt{db}[\gamma_0+\mathtt{k}+1], \mathtt{db}[\mathtt{k}]\} \\ G_w(\mathtt{k},\gamma_0,\gamma_1) &= \{\mathtt{dA}[\mathtt{n}\times(\gamma_0+\mathtt{k}+1)+\mathtt{k}+\gamma_1]\} \\ G_w(\mathtt{k},\gamma_0,0) &= \{\mathtt{db}[\gamma_0+\mathtt{k}+1]\} \end{aligned}$$

Host memory access mapping On each iteration on the host the following memory accesses are performed:

$$\begin{aligned} H_r(\mathtt{i},\mathtt{n}-1) &= \{\mathtt{hb}[\mathtt{i}]\} \\ H_r(\mathtt{i},\mathtt{j}) &= \{\mathtt{hx}[\mathtt{i}], \mathtt{hx}[\mathtt{j}], \mathtt{hA}[\mathtt{i}\times\mathtt{n}+\mathtt{j}]\} \\ H_r(\mathtt{i},\mathtt{i}+1) &= \{\mathtt{hA}[\mathtt{i}\times\mathtt{n}+\mathtt{i}]\} \\ H_w(\mathtt{i},\mathtt{j}) &= \{\mathtt{hx}[\mathtt{i}]\} \end{aligned}$$

Performance metrics The performance is evaluated using the following metrics:

- The total time for all kernel launches (computation time).
- The total memory transfer time from host to device.
- The total memory transfer time from device to host.

The performance is evaluated only for the part of the algorithm implemented in OpenCL (Forward Elimination).

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can print the transformed matrices and the solution for the original system, which can be validated by external tools (not supplied).

Target-specific optimizations The following target-specific optimization are used in the algorithm implementation:

- **Consecutive memory access.** Work items with consecutive global ids in dimension 1 access consecutive memory locations which improves the memory system performance (caching, coalescing).

Target-specific optimizations opportunities

- The intermediate multiplier array $M[\,][\,]$ can be reduced to one dimensional array $m[\,]$, so that this algorithm:

  for k **in** $0:(n-2)$ **do**
  $M_{i,k} = A_{i,k}/A_{k,k} \quad k < i < n$
  $A_{i,j} = A_{i,j} - A_{k,j} \times M_{i,k} \quad k < i < n, k \leq j < n$
  $b_i = b_i - M_{i,k} \times b_k \quad k < i < n$
  end for

  becomes:

  for k **in** $0:(n-2)$ **do**
  $m_i = A_{i,k}/A_{k,k} \quad k < i < n$
  $A_{i,j} = A_{i,j} - m_i \times A_{k,j} \quad k < i < n, k \leq j < n$
  $b_i = b_i - m_i \times b_k \quad k < i < n$
  end for

  This transformation decreases total memory consumption by almost 50% and increases locality.

2.3.5 Heart Wall

High-level description

The benchmark tracks the movements of the mouse heart walls on the series of ultrasound images. See [21] for more details.

Abstract data structures The benchmark operates on the following data objects:

- S is an input N_f video sequence (array of frames).
- S_i is an input $N_c \times N_r$ frame.
- D is an input $N_{\texttt{endo}}$ element array of endo-cardial surfaces coordinates.
  - D_i^r denotes row coordinate of the i^{th} surface.
  - D_i^c denotes column coordinate of the i^{th} surface.
- P is an input $N_{\texttt{epi}}$ element array of epi-cardial surfaces coordinates (same as D).
- D' is an intermediate $N_{\texttt{endo}} \times N_f$ array of endo-cardial surfaces coordinates.
- P' is an intermediate $N_{\texttt{epi}} \times N_f$ array of endo-cardial surfaces coordinates.
- DT is intermediate $(N_{\texttt{DT}}) \times N_{\texttt{endo}}$ buffer of floating point values.
- PT is intermediate $(N_{\texttt{PT}}) \times N_{\texttt{epi}}$ buffer of floating point values.
- IN2 is intermediate $(N_{\texttt{IN2}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- C is intermediate $(N_{\texttt{C}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- IN2P is intermediate $(N_{\texttt{IN2P}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- IN2S is intermediate $(N_{\texttt{IN2S}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- IN2SC is intermediate $(N_{\texttt{IN2SC}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- IN2SCS is intermediate $(N_{\texttt{IN2SCS}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- IN2SQR is intermediate $(N_{\texttt{IN2SQR}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- IN2SQR2 is intermediate $(N_{\texttt{IN2SQR2}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- INSQR is intermediate $(N_{\texttt{INSQR}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- M is intermediate $(N_{\texttt{M}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.
- MC is intermediate $(N_{\texttt{MC}}) \times (N_{\texttt{epi}} + N_{\texttt{endo}})$ buffer of floating point values.

Computation The algorithm tracks the movements of the surfaces:

GENERATETEMPLATE(S_0)
for F **in** S **do** ▷ For each frame.
  for d **in** D **do** ▷ For each inner point (Endo-cardial surface).
    TRACKINNERPOINTMOVEMENT(F, d)
  end for
  for p **in** P **do** ▷ For each outer point (Epi-cardial surface).
    TRACKOUTERPOINTMOVEMENT(F, p)
  end for
end for

All intermediate buffers are used within initialization (`GenerateTemplate`) and tracking (`TrackInnerPointMovement`, `TrackOuterPointMovement`) procedures.

Low-level implementation details

The benchmark device code consists of a massive kernel (about 1.2k loc). The iteration space of the kernel is quite complex (20 loop nests with depth from 1 to 4). The further analysis briefly covers the host/device data structures.

Host data structures

- **int** `hS[Nx * Ny]`: $S_{k,i,j} \rightarrow$ `hS[i * Nx + j]`
- **int** `hDC[Nendo]`: $D^c_i \rightarrow$ `hDC[i]`
- **int** `hDR[Nendo]`: $D^r_i \rightarrow$ `hDR[i]`
- **int** `hPC[Nepi]`: $D^c_i \rightarrow$ `hPC[i]`
- **int** `hPR[Nepi]`: $D^r_i \rightarrow$ `hPR[i]`
- **int** `hDCT[Nendo * Nf]`: $D'^c_{k,i} \rightarrow$ `hDCT[i * Nf + k]`
- **int** `hDRT[Nendo * Nf]`: $D'^r_{k,i} \rightarrow$ `hDRT[i * Nf + k]`
- **int** `hPCT[Nepi * Nf]`: $P'^c_{k,i} \rightarrow$ `hPCT[i * Nf + k]`
- **int** `hPRT[Nepi * Nf]`: $P'^r_{k,i} \rightarrow$ `hPRT[i * Nf + k]`

Device data structures

- **int** `dS[Nx * Ny]`: $S_{k,i,j} \rightarrow$ `dS[i * Nx + j]`
- **int** `dDC[Nendo]`: $D^c_i \rightarrow$ `dDC[i]`
- **int** `dDR[Nendo]`: $D^r_i \rightarrow$ `dDR[i]`
- **int** `dPC[Nepi]`: $D^c_i \rightarrow$ `dPC[i]`
- **int** `dPR[Nepi]`: $D^r_i \rightarrow$ `dPR[i]`
- **int** `dDCT[Nendo * Nf]`: $D'^c_{k,i} \rightarrow$ `dDCT[i * Nf + k]`

- **int** dDRT[Nendo * Nf]: $D'^{r}_{k,i} \rightarrow$ dDRT[i * Nf + k]
- **int** dPCT[Nendo * Nf]: $P'^{c}_{k,i} \rightarrow$ dPCT[i * Nf + k]
- **int** dPRT[Nendo * Nf]: $P'^{r}_{k,i} \rightarrow$ dPRT[i * Nf + k]
- **int** dC[Nc * (Nendo + Nepi)]: $\mathtt{C}_i \rightarrow$ dC[i]
- **int** dIN2P[Nin2p * (Nendo + Nepi)]: $\mathtt{IN2P}_i \rightarrow$ dIN2P[i]
- **int** dIN2S[Nin2s * (Nendo + Nepi)]: $\mathtt{IN2S}_i \rightarrow$ dIN2S[i]
- **int** dIN2SC[Nin2sc * (Nendo + Nepi)]: $\mathtt{IN2SC}_i \rightarrow$ dIN2SC[i]
- **int** dIN2SCS[Nin2scs * (Nendo + Nepi)]: $\mathtt{IN2SCS}_i \rightarrow$ dIN2SCS[i]
- **int** dIN2SQR[Nin2sqr * (Nendo + Nepi)]: $\mathtt{IN2SQR}_i \rightarrow$ dIN2SQR[i]
- **int** dIN2SQR2[Nin2sqr2 * (Nendo + Nepi)]: $\mathtt{IN2SQR2}_i \rightarrow$ dIN2SQR2[i]
- **int** dINSQR[Ninsqr * (Nendo + Nepi)]: $\mathtt{INSQR}_i \rightarrow$ dINSQR[i]
- **int** dM[Nm * (Nendo + Nepi)]: $\mathtt{M}_i \rightarrow$ dM[i]
- **int** dMC[Nmc * (Nendo + Nepi)]: $\mathtt{MC}_i \rightarrow$ dMC[i]

Input datasets The benchmark uses pre generated input data set with following parameters:

- Number of frames to process (N_f). Can be adjusted via command line. The default value is 5.
- Size of the frame ($N_c \times N_r$) = (656×744).
- Number of endo-cardial surfaces (N_{endo}) = 20.
- Number of epi-cardial surfaces (N_{epi}) = 31.

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- Total time to read input data from file.
- Total time for OpenCL initialization.
- Total time for Device memory allocation and Host-Device memory transfer.
- Total time for all Heart Wall kernel launches.
- Total time for Device memory deallocation.

Validation mechanism The benchmark has no validation mechanisms.

2.3.6 Hot Spot

The benchmark estimates the temperature of a processor based on its two-dimensional floorplan and simulated power measurements using a system of differential equations [13].

High-level description

Abstract data structures The benchmark operates on the following data objects:

- P is an input $N \times M$ matrix of dissipated power values.
- T^0 is an input $N \times M$ matrix of initial temperature values.
- T^k is an intermediate $N \times M$ matrix of average temperature values on k^{th} time step $(0 \leq k < k_{\max})$.
- $T^{k_{\max}}$ is an output $N \times M$ matrix of average temperature values.

Computation The algorithm iterates over the temperature grid:

$$\begin{aligned} T_{i,j}^{k+1} &= T_{i,j}^{k} + S_{\texttt{div}} \times (P_{i,j} + (T_{i+1,j}^{k} + T_{i-1,j}^{k} - 2 \times T_{i,j}^{k}) \times R_y^{-1} \\ &\quad + (T_{i,j+1}^{k} + T_{i,j-1}^{k} - 2 \times T_{i,j}^{k}) \times R_x^{-1} + (T_{\texttt{base}} - T_{i,j}^{k}) \times R_z^{-1}) \end{aligned}$$

The following constants are used:

$$\begin{aligned} T_{\texttt{base}} &= 80.0 \quad \text{Ambient temperature, assuming no package at all} \\ C_t &= 0.0005 \quad \text{Chip parameter} \\ C_f &= 0.5 \quad \text{Capacitance fitting factor} \\ C_h &= 0.016 \quad \text{Chip height} \\ C_w &= 0.016 \quad \text{Chip width} \\ G_h &= C_h / N \quad \text{Grid step height} \\ G_w &= C_w / M \quad \text{Grid step width} \\ K_{si} &= 100 \\ S_h &= 1.75 \times 10^6 \\ \texttt{PD}_{\max} &= 3 \times 10^6 \quad \text{Maximum power density} \\ S_{\max} &= \texttt{PD}_{\max} / (C_f \times C_t \times S_h) \\ \texttt{Cap} &= C_f \times S_h \times C_t \times G_w \times G_h \\ S_{\texttt{div}} &= 10^{-3} / (S_{\max} \times \texttt{Cap}) \\ R_x &= G_w / (2 \times K_{si} \times C_t \times G_h) \\ R_y &= G_h / (2 \times K_{si} \times C_t \times G_w) \\ R_z &= C_t / (K_{si} \times G_h \times G_w) \end{aligned}$$

Iteration spaces and dependences On each iteration the algorithm updates the temperature values for each cell in the grid:

$$I = \{(k,i,j) : 0 \leq k < k_{\max}, 0 \leq i < N, 0 \leq j < M\}$$

$$I : \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} k \\ i \\ j \end{pmatrix} \leq \begin{pmatrix} K_{\max} - 1 \\ N - 1 \\ M - 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The computation generates the following dependences:

$$\begin{aligned} \delta_I \quad = \quad & I : (k-1,i,j) \xrightarrow{t} I : (k,i,j) \wedge \\ & I : (k-1,i-1,j) \xrightarrow{t} I : (k,i,j) \wedge \\ & I : (k-1,i+1,j) \xrightarrow{t} I : (k,i,j) \wedge \\ & I : (k-1,i,j-1) \xrightarrow{t} I : (k,i,j) \wedge \\ & I : (k-1,i,j+1) \xrightarrow{t} I : (k,i,j) \wedge \end{aligned}$$

Memory access mapping For $(k,i,j) \in I$:

$$\begin{aligned} M_r(i,j,k) &= \{P_{i,j}, T^k_{i,j}, T^k_{i-1,j}, T^k_{i+1,j}, T^k_{i,j-1}, T^k_{i,j+1}\} \\ M_w(i,j,k) &= \{T^{k+1}_{i,j}\} \end{aligned}$$

Low-level implementation details

Device code Each kernel launch performs `S` time steps on the entire grid. See Algorithm 45.

Host code See Algorithm 46.

Host data structures

- **`float`** `hP[N * M]`: $P_{i,j} \rightarrow$ `hP[i * M + j]`
- **`float`** `hT[N * M]`: $T^k_{i,j} \rightarrow$ `hT[i * M + j]`

Device data structures

- **`global float`** `dP[N * M]`: $P_{i,j} \rightarrow$ `dP[i * M + j]`
- **`global float`** `dTin[N * M]`: $T^{k+1}_{i,j} \rightarrow$ `dTin[i * M + j]`
- **`global float`** `dTout[N * M]`: $T^k_{i,j} \rightarrow$ `dTout[i * M + j]`

Algorithm 45 The Hot Spot device code abstraction.

Input: `int Iter` ▷ Number of iterations.
Input: `global float dP[]` ▷ Power value for each cell in a grid.
Input: `global float dTin[]` ▷ Initial temperature values.
Output: `global float dTout[]` ▷ Resulting temperature values.
Input: `int N` ▷ Number of rows in the grid.
Input: `int M` ▷ Number of columns in the grid.
Input: `int Br` ▷ Number of border rows.
Input: `int Bc` ▷ Number of border columns.
Input: `float Cap` ▷ Device parameter constant.
Input: `float Rx` ▷ Device parameter constant.
Input: `float Ry` ▷ Device parameter constant.
Input: `float Rz` ▷ Device parameter constant.
Input: `float Step` ▷ Device parameter constant.
Local: `local float dlP[`Λ_1`][`Λ_0`]` ▷ Temporary power values buffer.
Local: `local float dlTin[`Λ_1`][`Λ_0`]` ▷ Temporary input temperature values.
Local: `local float dlTout[`Λ_1`][`Λ_0`]` ▷ Temporary output temperature values.

1: **kernel** HSKERNEL (`Iter`, `dP`, `dTin`, `dTout`, `N`, `M`, `Br`, `Bc`, `Cap`, `Rx`, `Ry`, `Rz`, `Step`)
2: `Sdiv` $\leftarrow$ `Step`/`Cap`
3: `Tbase` $\leftarrow 80$
4: `smallBlockCols` $\leftarrow \Lambda_0 - 2 \times$ `Iter`
5: `smallBlockRows` $\leftarrow \Lambda_1 - 2 \times$ `Iter`
6: `blkXmin` $\leftarrow \nu_0 \times$ `smallBlockCols` $-$ `Bc`
7: `blkYmin` $\leftarrow \nu_1 \times$ `smallBlockRows` $-$ `Br`
8: `blkXmax` $\leftarrow$ `blkX` $+ \Lambda_0 - 1$
9: `blkYmax` $\leftarrow$ `blkY` $+ \Lambda_1 - 1$
10: `xidx` $\leftarrow$ `blkXmin` $+ \lambda_0$
11: `yidx` $\leftarrow$ `blkYmin` $+ \lambda_1$
12: `index` $\leftarrow$ `N` $\times$ `yidx` $+$ `xidx` ▷ Probably a bug, the global row number `yidx` should be multiplied by the number of columns (`M`) rather then number of rows (`N`).
13: **if** (`yidx` **in** $(0 : (\texttt{N} - 1))) \wedge$ (`xidx` **in** $(0 : (\texttt{M} - 1))$) **then**
14: `dlP`$[\lambda_1][\lambda_0] \leftarrow$ `dP`[`index`]
15: `dlTin`$[\lambda_1][\lambda_0] \leftarrow$ `dTin`[`index`]
16: **end if**
17: BARRIER(`CLK_LOCAL_MEM_FENCE`)
18: `Xmin` $\leftarrow \max(0, -$`blkXmin`$)$
19: `Ymin` $\leftarrow \max(0, -$`blkYmin`$)$
20: `Xmax` $\leftarrow \min(\Lambda_0 - 1, \Lambda_0 - 1 -$ (`blkXmax` $-$ `N` $+ 1$))
21: `Ymax` $\leftarrow \min(\Lambda_1 - 1, \Lambda_1 - 1 -$ (`blkYmax` $-$ `N` $+ 1$))
22: `N` $\leftarrow \max($`Ymin`$, \lambda_1 - 1)$
23: `S` $\leftarrow \min(\Lambda_1 + 1,$ `Ymax`$)$
24: `W` $\leftarrow \max($`Xmin`$, \lambda_0 - 1)$
25: `E` $\leftarrow \min(\Lambda_0 + 1,$ `Xmax`$)$

Algorithm 45 The Hot Spot device code abstraction (continues).

```
26:     for i in 0 : (Iter − 1) do
27:         isComputed ← false
28:         if (λ0 in (i + 1) : Λ0 − i − 2) ∧ (λ0 in Xmin : Xmax) ∧ (λ1 in (i + 1) : Λ1 − i − 2) ∧
            (λ1 in Ymin : Ymax) then
29:             isComputed ← true
30:             dlTout[λ1][λ0] ← dlTin[λ1][λ0] + Sdiv × (dlP[λ1][λ0] + (dlTin[S][λ0] +
            dlTin[N][λ0] − 2 × dlTin[λ1][λ0]) × Ry^−1 + (dlTin[λ1][E] + dlTin[λ1][W] − 2 ×
            dlTin[λ1][λ0]) × Rx^−1 + (Tbase − dlTin[λ1][λ0]) × Rz^−1)        ▷ See (2.19).
31:         end if
32:         BARRIER(CLK_LOCAL_MEM_FENCE)
33:         if i = Iter − 1 then
34:             break
35:         end if
36:         if isComputed then
37:             dlTin[λ1][λ0] ← dlTout[λ1][λ0]
38:         end if
39:         BARRIER(CLK_LOCAL_MEM_FENCE)
40:     end for
41: end kernel
```

- **local float** `dlP[`Λ_0`][`Λ_1`]`:
  $T^k_{i,j} \rightarrow$ `dlP[(i % (`Λ_0`- 2 * S) + S)][j % (`Λ_1`- 2 * S) + S]`
- **local float** `dlTin[`Λ_0`][`Λ_1`]`:
  $T^k_{i,j} \rightarrow$ `dlTin[(i % (`Λ_0`- 2 * S) + S)][j % (`Λ_1`- 2 * S) + S]`
- **local float** `dlTout[`Λ_0`][`Λ_1`]`:
  $T^{k+1}_{i,j} \rightarrow$ `dlTout[(i % (`Λ_0`- 2 * S) + S)][j % (`Λ_1`- 2 * S) + S]`

where `S` is the number of time steps performed by a single kernel launch.

Input datasets The benchmark uses a pre-generated 512×512 initial temperature and power values grids. The total number of iterations $K_{\max}$ and the number of iterations processed by a single kernel launch S can be specified via the command line. By default $K_{\max} = 60$ and $S = 1$.

Partitioned iteration space

$$P = \{(r, \nu_1, \nu_0, \lambda_1, \lambda_0, i) : 0 \leq r < K_{\max}, 0 \leq \nu_1 < N_1, 0 \leq \nu_0 < N_0, 0 \leq \lambda_1 < \Lambda_1, 0 \leq \lambda_0 < \Lambda_0\}$$

$$\begin{aligned} N_1 &= N \\ N_0 &= M \end{aligned}$$

Algorithm 46 The Hot Spot host code abstraction

```
1: procedure AllocateMemory(N, M)                    ▷ Allocate host memory and device buffers.
2:     hP ← AllocateHostMemory(N × M)
3:     hT ← AllocateHostMemory(N × M)
4:     dP ← AllocateBuffer(N × M, READ_ONLY, hP)
5:     dTin ← AllocateBuffer(N × M, READ_ONLY, hT)
6:     dTout ← AllocateBuffer(N × M, READ_ONLY)
7:     hTt[0] ← dTin
8:     hTt[1] ← dTout
9: end procedure

10: procedure InitializeMemory(N, M, file)           ▷ Initialize host memory.
11:     hP ← ReadFromFile(file, N × M)
12:     hT ← ReadFromFile(file, N × M)
13: end procedure

Input: int N                                         ▷ Number of rows in input array.
Input: int M                                         ▷ Number of columns in input array.
Input: int Imax                                      ▷ Number of iterations.
Input: int Iter   ▷ Number of iterations for each kernel launch (see Algorithm 45 for more
    details).
Input: const char * file                             ▷ File with pre-generated data.
14: size ← n × m
15: AllocateMemory(N, M)
16: InitializeMemory(N, M, file)
17: hsProgram ← BuildProgram("hsProgram.cl")
18: hsKernel ← BuildKernel(hsProgram, "hsKernel")
19: src, ret ← 0, 1
20: for r in 0 : (Imax − 1) : Iter do
21:     it ← min(Iter, Imax − r − 1)
22:     Cap ← Cap                                    ▷ See the Computation section.
23:     Rx ← R_x                                     ▷ See the Computation section.
24:     Ry ← R_y                                     ▷ See the Computation section.
25:     Rz ← R_z                                     ▷ See the Computation section.
26:     Step ← 10^−3 / S_max                         ▷ See the Computation section.
27:     SetKernelArguments(pfKernel, it, dP, hTt[src], hTt[ret], N, M, Iter, Iter,
    Cap, Rx, Ry, Rz, Step)
28:     EnqueueKernel(hsKernel, (M/(16 − 2 × Iter) × 16, N/(16 − 2 × Iter) ×
    16), (16, 16), ∅)
29:     src, ret ← ret, src
30: end for
31: CopyToHost(hTt[ret], hT, True, ∅)
```

$$\begin{aligned} f(k,i,j) \quad \rightarrow \quad & ((i/S)\times S, i/(\Lambda_1 - S\times 2), j/(\Lambda_0 - S\times 2), \\ & i \bmod (\Lambda_1 - S\times 2) + S, j \bmod (\Lambda_0 - S\times 2) + S, k \bmod S\ ) \end{aligned}$$

$$f^{-1}(r,\nu_1,\nu_0,\lambda_1,\lambda_0,i) = (r+i, \nu_1 * (\Lambda_1 - S\times 2) - S + \lambda_1, \nu_0 * (\Lambda_0 - S\times 2) - S + \lambda_0)$$

Device memory access mapping Each work-item performs the following reads and writes:

$$\begin{aligned} G_r(r,\nu_0,\nu_1,\lambda_0\lambda_1,i) &= \{\texttt{dP}[\texttt{N}\times((\Lambda_1 - 2\times S)\times\nu_1 - S + \lambda_1) + (\Lambda_0 - 2\times S)\times\nu_0 - S + \lambda_0)], \\ &\quad \texttt{dTin}[\texttt{N}\times((\Lambda_1 - 2\times S)\times\nu_1 - S + \lambda_1) + (\Lambda_0 - 2\times S)\times\nu_0 - S + \lambda_0)]\} \\ G_w(r,\nu_0,\nu_1,\lambda_0\lambda_1,i) &= \{\texttt{dTout}[\texttt{N}\times((\Lambda_1 - 2\times S)\times\nu_1 - S + \lambda_1) + (\Lambda_0 - 2\times S)\times\nu_0 - S + \lambda_0)]\} \\ L_r(r,\nu_0,\nu_1,\lambda_0\lambda_1,i) &= \{\texttt{dlTin}[\lambda_1][\lambda_0], \texttt{dlTin}[\lambda_1+1][\lambda_0], \texttt{dlTin}[\lambda_1-1][\lambda_0] \\ &\quad \texttt{dlTin}[\lambda_1][\lambda_0+1], \texttt{dlTin}[\lambda_1][\lambda_0-1], \texttt{dlP}[\lambda_1][\lambda_0]\} \\ L_w(r,\nu_0,\nu_1,\lambda_0\lambda_1,i) &= \{\texttt{dlTin}[\lambda_1][\lambda_0], \texttt{dlTout}[\lambda_1][\lambda_0], \texttt{dlP}[\lambda_1][\lambda_0]\} \end{aligned}$$

The global memory access offset is probably wrong for non-square matrices (`M` should be used instead of `N` as a multiplier).

Performance metrics The performance is evaluated using the following metric:

- The total time for all kernel launches (computation time).

Validation mechanism The benchmark has no validation mechanisms.

Target-specific optimizations

- **Blocking.** The input data is processed in blocks having the number of columns equal to the number of work items per work group. Consecutive work items of the same group access consecutive memory addresses which improves the memory subsystem performance (caches, coalescing).
- **Ghost zone[14].** The computation of the $(i,j)^{th}$ element on the k^{th} iteration requires the $(i,j-1)^{th}, (i,j+1)^{th}, (i-1,j)^{th}, (i+1,j)^{th}$ elements from the previous iteration. Since these data can be produced by another work group, the ghost zone technique is used to avoid data races: Each kernel launch performs an S iterations over input data. Given a block with Λ rows and Λ columns the $(\Lambda - S\times 2)^2$ elements of the block can be computed on the last iteration by a single work group without communicating with other groups. Therefor, the whole grid is divided into non-overlapping blocks with $(\Lambda - 2\times S)$ rows and $(\Lambda - 2\times S)$ columns. Each group allocates a local buffer for Λ^2 elements, which is used to compute the resulting $(\Lambda - 2\times S)^2$ elements without affecting the other groups' data. This technique allows each work group to compute the result independently, although the elements located in the ghost areas are computed twice.

- **I/O buffers alternation.** The output of the kernel execution on the current iteration is used as kernel input for the next one. To avoid additional memory transfers, the input and output buffers are logically swapped between the kernel launches:

  ...
  `kernel`(`buffptr`[0], `buffptr`[1])
  `kernel`(`buffptr`[1], `buffptr`[0])
  `kernel`(`buffptr`[0], `buffptr`[1])
  ...

  The 2-element array `buffptr` contains the pointers to the device memory buffers.

2.3.7 K-means

High-level description

The benchmark uses k-means clustering to partition a cluster of objects into several sub-clusters based on objects' features.

Abstract data structures

- P is an input N_e element vector of objects. Each object is an N_f element vector of features (floating point values).
- S is an output set of sub-clusters. Each object from P belongs to a single sub-cluster from S.
- C is an intermediate vector of the centroids.
- Q is an intermediate vector of clusterization costs.

Computation The algorithm partitions a cluster of objects into several sub-clusters ($S = \{S_i\} : \mathtt{K}_{\min} \leq |S| \leq \mathtt{K}_{\max}$) based on objects' features. Each object is associated with a sub-cluster, having the nearest centroid, according to the Euclidean distance.

- C_A denotes the centroid of the sub-cluster A:

$$C_A = \frac{\sum_{p \in A} p}{|A|}$$

- Q_S denotes the quality (cost) of the partitioning:

$$Q_S = \sum_{S_i \in S} \sum_{p \in S_i} \|p - C_{S_i}\|$$

The algorithm tries to determine the minimal-cost partitioning:

$$S = \min_{Q_S}(S^k : \mathtt{K}_{\min} \leq k = |S^k| \leq \mathtt{K}_{\max})$$

For each partitioning size (number of clusters) k **in** $K_{\min} : K_{\max}$, the algorithm starts with a random partitioning of this size and iteratively recalculates partitioning until a fixed point is reached.

- $S^k_{i,\tau}$ denotes the i^{th} sub-cluster on the τ^{th} iteration.
- S^k_i denotes the i^{th} sub-cluster when the fixed point is reached. set.
- $S^k = \{S^k_i\}$

1. Initialize centroids:

$$C_{S^k_{i,0}} = \mathtt{RandomCentroid}(), 0 \leq i < k$$

2. Recalculate clusters:

$$S_{i,\tau}^k = \{p : \|p - C_{S_{i,\tau}^k}\| < \|p - C_{S_{j,\tau-1}^k}\|, \forall j \neq i\}, 0 \leq i < k, 1 \leq \tau$$

3. Recalculate centroids:

$$C_{S_{i,\tau}^k} = \frac{\sum_{p \in S_{i,\tau}^k} p}{|S_{i,\tau}^k|}, 0 \leq i < k, 1 \leq \tau$$

4. Repeat steps 2 and 3 (increasing τ) until a fixed point is reached:

$$S^k = \{S_{i,\tau}^k : 0 \leq i < k, S_{i,\tau}^k = S_{i,\tau+1}^k\}$$

5. Calculate the cost:

$$Q_{S_k} = \sum_{S_i^k \in S^k} \sum_{p \in S_i^k} \|p - C_{S_i^k}\|$$

The final partitioning is the one having the minimal cost:

$$S = \min_{Q_{S_k}} S^k$$

Iteration spaces

- Initialize centroids:

$$I_I = \{(k, c, f) : K_{\min} \leq k \leq K_{\max}, c \in C_0^k, f \in c\}$$

- Recalculate clusters:

$$I_S = \{(k, \tau, p, s, f) : K_{\min} \leq k \leq K_{\max}, 1 \leq \tau, p \in F, s \in \{S_{i,\tau}^k : 0 \leq i < k\}, f \in p\}$$

- Recalculate centroids:

$$I_P = \{(k, \tau, s, p, f) : K_{\min} \leq k \leq K_{\max}, 1 \leq \tau, s \in \{S_{i,\tau}^k : 0 \leq i < k\}, p \in s, f \in p\}$$

- Calculate cost:

$$I_C = \{(k, s, p, f) : K_{\min} \leq k \leq K_{\max}, s \in S^k, p \in s, f \in p\}$$

The τ iteration variable represents the repeat-until loop, which terminates when a fixed point is reached.

Dependences

$$\begin{aligned}
\delta_{I_I} &= \emptyset \\
\delta_{I_S} &= I_I(k, C_s, f) \xrightarrow{t} I_S(k, 1, p, s, f) \wedge \\
&\quad I_P(k, \tau - 1, s, s, f) \xrightarrow{t} I_S(k, \tau, p, s, f) \\
\delta_{I_P} &= I_S(k, \tau, p, \{S_{i,\tau}^k : 0 \leq i < k\}, f) \xrightarrow{t} I_P(k, \tau, s, p, f) \\
\delta_{I_C} &= I_S(k, \{\tau : \tau > 0\}, s, p, f) \xrightarrow{t} I_C(k, s, p, f)
\end{aligned}$$

Memory access mapping

- Initialize centroids:

$$M_w(k,c,f) = \{c\}$$

- Recalculate clusters:

$$\begin{aligned} M_r(k,\tau,p,s,f) &= \{p,C_s\} \\ M_w(k,\tau,p,s,f) &= \{s\} \end{aligned}$$

- Recalculate centroids:

$$\begin{aligned} M_r(k,\tau,s,p,f) &= \{p\} \\ M_w(k,\tau,s,p,f) &= \{C_s\} \end{aligned}$$

- Calculate cost:

$$\begin{aligned} M_r(k,s,p,f) &= \{p,C_s\} \\ M_w(k,s,p,f) &= \{Q_{S_k}\} \end{aligned}$$

Low-level implementation details

Device code See Algorithms 47 and 48.

Host code See Algorithm 49. The algorithm is repeated `Niter` times for each k.[13]

Host data structures

- **`global float`** `hP[Ne][Nf]`: $P_{p,f} \rightarrow$ `hP[p][f]`
- **`global int`** `hM[Ne]`: For every element `p` `hM[p]` denotes the cluster the element belongs to before kernel execution.
- **`global int`** `hM2[Ne]`: For every element `p` `hM[p]` denotes the cluster the element belongs to after kernel execution.
- **`global float`** `hL[k]`: $\|S^k_{i,\tau}\| \rightarrow$ `hL[i]`
- **`global float`** `hC[K][Nf]`: $C_{c,f} \rightarrow$ `hC[c][f]` before kernel execution.
- **`global float`** `hC2[K][Nf]`: $C_{c,f} \rightarrow$ `hC2[c][f]` after kernel execution.

Device data structures

- **`global float`** `dP[Ne * Nf]`: $P_{p,f} \rightarrow$ `dP[p * Nf + f]`
- **`global float`** `dP2[Ne * Nf]`: $P_{p,f} \rightarrow$ `dP[f * Np + p]`
- **`global float`** `dC[K * Nf]`: $C_{c,f} \rightarrow$ `dC[c * Nf + f]`
- **`global int`** `dM[Ne]`: For every element `p` `dM[p]` denotes the cluster the element belongs to.

[13] Probably for benchmarking purposes since it's not required by the algorithm.

Algorithm 47 The K-Means device code abstraction (Clusters Recalculation).

Input: `int Ne`
Input: `int Nf`
Input: `int K` ▷ Number of clusters.
Input: `global float dP[]`
Input: `global float dC[]`
Output: `int dM[]`

1: **kernel** KMKERNEL1 (dP, dC, dM, Ne, Nf, K)
2: **if** $\gamma <$ Ne **then**
3: idx $\leftarrow 0$
4: min $\leftarrow$ MAX_FLOAT
5: **for** i **in** $0 : (\text{K} - 1)$ **do**
6: ans $\leftarrow 0$
7: **for** f **in** $0 : (\text{Nf} - 1)$ **do**
8: ans $\leftarrow$ ans $+ (\text{dP}[\text{f} \times \text{Ne} + \gamma] - \text{dC}[\text{i} \times \text{Nf} + \text{f}])^2$
9: **end for**
10: **if** ans $<$ min **then**
11: min $\leftarrow$ ans
12: idx $\leftarrow$ i
13: **end if**
14: **end for**
15: dM$[\gamma] \leftarrow$ idx
16: **end if**
17: **end kernel**

Algorithm 48 The K-Means device code abstraction (Matrix Transposition).

Input: `int Ne`
Input: `int Nf`
Input: `global float dP[]`
Output: `global float dP2[]`

1: **kernel** KMKERNEL2 (dP, dP2, Ne, Nf)
2: **for** f **in** $0 : (\text{Nf} - 1)$ **do**
3: dP2$[\text{f} \times \text{Ne} + \gamma] \leftarrow$ dP$[\gamma \times \text{Nf} + \text{f}]$
4: **end for**
5: **end kernel**

Algorithm 49 The K-Means host code abstraction.

```
1:  procedure AllocateMemoryInit(Ne, Nf)
2:      hP ← AllocateHostMemory(Ne × Nf)
3:      hM ← AllocateHostMemory(Ne)
4:  end procedure
5:  procedure AllocateMemoryIter(Ne, Nf, K)
6:      dP ← AllocateBuffer(Ne × Nf, READ_WRITE)
7:      dP2 ← AllocateBuffer(Ne × Nf, READ_WRITE)
8:      dC ← AllocateBuffer(K × Nf, READ_WRITE)
9:      dM ← AllocateBuffer(Ne, READ_WRITE)
10:     kmProgram ← BuildProgram("kmProgram.cl")
11:     kmKernel1 ← BuildKernel(kmProgram, "kmKernel1")
12:     kmKernel2 ← BuildKernel(kmProgram, "kmKernel2")
13:     CopyToDevice(dP, hP, True, ∅)
14:     SetKernelArguments(kmKernel2, dP, dP2, Nf, K)
15:     EnqueueKernel(kmKernel1, (Ne), none, ∅)
16: end procedure
17: procedure InitializeMemory(Ne, Nf, file)
18:     hP ← ReadFromFile(file, Ne × Nf)
19: end procedure
20: function KMEANS(Nf, Ne, K)
21:     CopyToDevice(dC, hC, True, ∅)
22:     SetKernelArguments(kmKernel1, dP, dC, dM, Ne, Nf, K)
23:     EnqueueKernel(kmKernel1, (Ne), none, ∅)
24:     CopyToHost(dM, hM2, True, ∅)
25:     delta ← 0
26:     for i in 0 : (Ne − 1) do
27:         Cid ← hM2[i]
28:         hL[Cid] ← hL[Cid] + 1
29:         if hM2[i] ≠ hM[i] then
30:             delta ← delta + 1
31:             hM[i] ← hM2[i]
32:         end if
33:         for j in 0 : (Nf − 1) do
34:             hC2[Cid][j] ← hC2[Cid][j] + hP[i][j]
35:         end for
36:     end for
37:     return delta
38: end function
```

Algorithm 49 The K-Means host code abstraction (continues).

```
39: function DISTANCE(P1, P2, S)
40:     sum ← 0
41:     for i in 0 : (S − 1) do
42:         sum ← sum + (P2[i] − P1[i])^2
43:     end for
44:     return sum
45: end function
46: function FINDNEARESTPOINT(P, Nf, Objs, S)
47:     min ← FLT_MAX
48:     for i in 0 : (S − 1) do
49:         dist ←DISTANCE(P, Objs[i], Nf)
50:         min ← min(min, dist)
51:         idx ← argmin(min, dist)
52:     end for
53:     return idx
54: end function
55: function RMSERROR(Nf, Ne, K)
56:     sum ← 0
57:     for i in 0 : (Ne − 1) do
58:         idx ←FINDNEARESTPOINT(hP[i], Nf, hC, K)
59:         sum ← sum+DISTANCE(hP[i], hC[idx], Nf)
60:     end for.State return √(sum/Ne)
61: end function
```

Algorithm 49 The K-Means host code abstraction (continues).

Input: `int Niter` ▷ Number of iterations (for benchmarking).
Input: `int Ne` ▷ Number of elements in the input set.
Input: `int Nf` ▷ Number of features in each element.
Input: `int Kmax` ▷ Maximal number of clusters.
Input: `int Kmin` ▷ Minimal number of clusters.
Input: `const char * file` ▷ File with pre-generated elements.

62: ALLOCATEMEMORYINIT(Ne, Nf)
63: INITIALIZEMEMORY(Ne, Nf, file)
64: RmseMin ← FLT_MAX
65: **for** K **in** Kmin : Kmax **do**
66: ALLOCATEMEMORYITER(Ne, Nf, K)
67: **for** iter **in** 0 : (Niter − 1) **do**
68: hC ← ALLOCATEHOSTMEMORY(Nf × K)
69: hC2 ← ALLOCATEHOSTMEMORY(Nf × K)
70: hL ← ALLOCATEHOSTMEMORY(K)
71: hM2 ← ALLOCATEHOSTMEMORY(Ne)
72: **for** c **in** 0 : (K − 1) **do**
73: **for** f **in** 0 : (Nf − 1) **do**
74: hC[c][f] ← RANDOMFEATURE( )
75: **end for**
76: **end for**
77: delta ← 0
78: threshold ← 0.001 ▷ Probably a bug, since delta is less than threshold only when delta is 0.
79: **repeat**
80: delta ←KMEANS(Nf, Ne, K)
81: **for** i **in** 0 : (K − 1) **do**
82: **for** j **in** 0 : (Nf − 1) **do**
83: **if** hL[i] > 0 **then**
84: hC[i][j] ← hC2[i][j]/hL[i]
85: **end if**
86: hC2[i][j] ← 0
87: **end for**
88: hL[i] ← 0
89: **end for**
90: **until** delta < threshold
91: rmse ←RMSERROR(Nf, Ne, K)
92: **if** rmse < RmseMin **then**
93: RmseMin ← rmse
94: BestK ← K
95: BestC ← hC
96: **end if**
97: **end for**
98: **end for**

Input datasets The benchmark uses the following pre generated input data sets:

- 100 objects with 34 features.
- 204800 objects with 34 features.
- 494020 objects with 34 features.
- 819200 objects with 34 features.

Each feature is a floating point number.

Partitioned iteration space

- Initialize centroids (Host):

$$P_I = \{(k,i,c,f) : K_{\min} \leq k \leq K_{\max}, 0 \leq i < \texttt{Niter}, 0 \leq c < K, 0 \leq f < \texttt{Nf}\}$$

$$f(k, c = C_i, f = c_j) \rightarrow (k, 0 : (\texttt{Niter} - 1), i, j)$$

$$f^{-1}(k,i,c,f) \rightarrow (k, C_c, c_f)$$

- Recalculate clusters (kmKernel1):

$$P_{Sd} = \{(k,i,\tau,\gamma,c,f) : K_{\min} \leq k \leq K_{\max}, 0 \leq i < \texttt{Niter}, 0 \leq \tau, 0 \leq \gamma < \Gamma = N_e, 0 \leq c < K, 0 \leq f < \texttt{Nf}\}$$

$$f(k, \tau, p = P_i, s = C_j, f = p_l) \rightarrow (k, 0 : (\texttt{Niter} - 1), \tau, i, j, l)$$

$$f^{-1}(k,i,\tau,\gamma,c,f) \rightarrow (k, \tau, p = P_\gamma, C_c, p_f)$$

- Recalculate clusters (Host):

$$P_{Sh} = \{(k,i,\tau,p,f) : K_{\min} \leq k \leq K_{\max}, 0 \leq i < \texttt{Niter}, 0 \leq \tau, 0 \leq p < N_e, 0 \leq f < \texttt{Nf}\}$$

$$f(k, \tau, p = P_i, s, f = p_l) \rightarrow (k, 0 : (\texttt{Niter} - 1), \tau, i, l)$$

$$f^{-1}(k,i,\tau,e,f) \rightarrow (k, \tau, p = P_e, C_{hM[e]}, p_f)$$

- Recalculate centroids (Host):

$$P_P = \{(k,i,\tau,c,f) : K_{\min} \leq k \leq K_{\max}, 0 \leq i < \texttt{Niter}, 0 \leq \tau, 0 \leq p < K, 0 \leq f < \texttt{Nf}\}$$

$$f(k, \tau, s = C_i, p, f = p_l) \rightarrow (k, 0 : (\texttt{Niter} - 1), \tau, i, l)$$

$$f^{-1}(k,i,\tau,c,f) \rightarrow (k, \tau, c, P, c_f)$$

- Calculate cost (Host):

$$P_C = \{(k,i,p,s,f) : K_{\min} \leq k \leq K_{\max}, 0 \leq i < \texttt{Niter}, 0 \leq p < N_e, 0 \leq s < K, 0 \leq f < \texttt{Nf}\}$$

$$f(k,\tau,s=C_i,p=P_j,f=p_l) \rightarrow (k,0:(\texttt{Niter}-1),\tau,j,i,l)$$

$$f^{-1}(k,i,\tau,e,c,f) \rightarrow (k,\tau,C_{hM[e]},C_c,p=P_e,p_f)$$

Device memory access mapping

- kmKernel1:

$$\begin{aligned} G_r(\texttt{k},\gamma,\texttt{i},\texttt{f}) &= \{\texttt{dP2}[\texttt{f}\times\texttt{Ne}+\gamma],\texttt{dC}[\texttt{i}\times\texttt{Nf}+\texttt{f}]\} \\ G_w(\texttt{k},\gamma) &= \{\texttt{dM}[\gamma]\} \end{aligned}$$

- kmKernel2:

$$\begin{aligned} G_r(\texttt{k},\gamma,\texttt{f}) &= \{\texttt{dP}[\gamma\times\texttt{Nf}+\texttt{f}]\} \\ G_w(\texttt{k},\gamma,\texttt{f}) &= \{\texttt{dP2}[\texttt{f}\times\texttt{Ne}+\gamma]\} \end{aligned}$$

Host memory access mapping

- Initialize centroids:

$$G_w(k,i,c,f) = \{\texttt{hC}[\texttt{c}][\texttt{f}]\}$$

- Recalculate clusters:

$$\begin{aligned} G_r(k,i,\tau,p,f) &= \{\texttt{hM2}[\texttt{p}],\texttt{hM}[\texttt{p}],\texttt{hL}[\texttt{hM2}[\texttt{p}]],\texttt{hC2}[\texttt{hM2}[\texttt{p}]][\texttt{f}],\texttt{hP}[\texttt{p}][\texttt{f}]\} \\ G_w(k,i,\tau,p,f) &= \{\texttt{hL}[\texttt{hM2}[\texttt{p}]],\texttt{hM}[\texttt{p}],\texttt{hC2}[\texttt{hM2}[\texttt{p}]][\texttt{f}]\} \end{aligned}$$

- Recalculate centroids:

$$\begin{aligned} G_r(k,i,\tau,c,f) &= \{\texttt{hL}[\texttt{c}],\texttt{hC2}[\texttt{c}][\texttt{f}]\} \\ G_w(k,i,\tau,p,f) &= \{\texttt{hL}[\texttt{c}],\texttt{hC}[\texttt{c}][\texttt{f}],\texttt{hC2}[\texttt{c}][\texttt{f}]\} \end{aligned}$$

- Calculate cost:

$$G_r(k,i,p,c,f) = \{\texttt{hP}[\texttt{p}][\texttt{f}],\texttt{hC}[\texttt{c}][\texttt{f}]\}$$

Performance metrics The performance of the algorithm implementation is evaluated by measuring the total time required to perform clusterization for the given set of objects (both host and device parts).

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can print the traceback values which can be validated by external tools (not supplied).

Target-specific optimizations

- **Coalescing-friendly memory access (probably CUDA only).** In order to achieve coalesced memory access the input data buffer is logically transposed. For CUDA devices this transformation makes the data access in kernel coalesced. In other hand this greatly reduces the data locality. Although for CUDA devices the advantages of the coalesced memory access overrides the disadvantages of the non-localized memory access, this might not be the same for OpenCL. Since OpenCL standard does not require the consecutive memory accesses within threads of the same working group to be coalesced, this transformation might cause a performance degradation.

Target-specific optimizations opportunities

- **Vector operations.** The device code can benefit from using vector operations.

2.3.8 LavaMD

High-level description

The benchmark calculates particle potential and relocation within a large 3D space based on the mutual forces between particles.

Abstract data structures The benchmark operates on particles in a 3D space. The space is divided into boxes. Each box contains N_{pb} particles. Each non-boundary box is surrounded by 26 neighbor boxes. The number of neighbors for the boundary boxes is smaller (17 for sides and 11 for edges and 7 for corners). Each particle is defined as a quintuple (x, y, z, v, q):

- x denotes the x coordinate of the particle.
- y denotes the y coordinate of the particle.
- z denotes the z coordinate of the particle.
- v denotes the velocity of the particle.
- q denotes the charge of the particle.

The benchmark operates the on the following data structures:

- B is an $N_{bx} \times N_{by} \times N_{bz}$ input three dimensional array of boxes. Each box contains N_{pb} particles.
- $B_{i,j,k}$ is the N_{pb} element input array of initial configurations of the particles, located in the box with indices (i, j, k) in the array B.
- $P = (x, y, z, v, q)$ is the particle configuration.
- N_b is a set of neighbor boxes for the box b including the b.
- $F_{i,j,k}$ is the N_{pb} element output array of updated configurations of the particles, located in the box with indices (i, j, k) in the array B.

Computation The new configurations $F_{i,j,k}$ for the particles in the box $B_{i,j,k}$ are calculated as follows for each particle $P_s = (x, y, z, v, q)$ in $B_{i,j,k}$

$$
\begin{aligned}
F_{i,j,k} &= \{P'_s, 0 \leq s < N_{pb}\} \\
P'_s &= (x', y', z', v', q') \\
x' &= \sum_{B' \in N_{B_{i,j,k}}} \sum_{C \in B'} 2 \times e^{-\frac{v + C_v - (x,y,z)\cdot(C_x,C_y,C_z)}{2}} \times (x - C_x) \times C_q \\
y' &= \sum_{B' \in N_{B_{i,j,k}}} \sum_{C \in B'} 2 \times e^{-\frac{v + C_v - (x,y,z)\cdot(C_x,C_y,C_z)}{2}} \times (y - C_y) \times C_q \\
z' &= \sum_{B' \in N_{B_{i,j,k}}} \sum_{C \in B'} 2 \times e^{-\frac{v + C_v - (x,y,z)\cdot(C_x,C_y,C_z)}{2}} \times (z - C_z) \times C_q \\
v' &= \sum_{B' \in N_{B_{i,j,k}}} \sum_{C \in B'} e^{-\frac{v + C_v - (x,y,z)\cdot(C_x,C_y,C_z)}{2}} \times C_q \\
q' &= q
\end{aligned}
$$

Iteration space and dependences For each particle in the each box the algorithm iterates over all particles in the neighbor boxes:

$$I = \{(b, p, n, c) : b \in B, p \in b, n \in N_b, c \in n\}$$

The only dependences present in this computation are the dependences introduced by the "reduce" operation (for summation).

Memory access mapping

$$
\begin{aligned}
M_r(b, p, n, c) &= \{b, p, n, c\} \\
M_w(b = B_{i,j,k}, p = P_s, n, c) &= \{p' = P'_s \in F_{i,j,k}\}
\end{aligned}
$$

Low-level implementation details

The benchmark launches $N_{bx} \times N_{by} \times N_{bz}$ work groups *i.e.* one group for each box [20, 19].

Device code See Algorithm 50.

Host code See Algorithm 51.

Host data structures

- **struct** Neighbor

```
struct Neighbor
{
  int x, y, z; //Coordinates.
  int number; //Global neighbor box id.
  long offset; //Box offset in Particles array.
}
```

Algorithm 50 LavaMD device code abstraction.

Input: `global struct Box dB[]`
Input: `global struct Particle dP[]`
Input: `global float dQ[]`
Input: `int S`
Input: `int Npb` ▷ $S = N_{bx} \times N_{by} \times N_{bz}$
Output: `global struct Particle dF[]`
Local: `local Particle dlA[`Λ`]`
Local: `local Particle dlB[`Λ`]`
Local: `local Float dlQ[`Λ`]`

1: **kernel** LMDKERNEL (dB, dP, dQ, S, dF, Npb)
2: **if** $\gamma <$ S **then**
3: alpha $\leftarrow 0.5$
4: $\text{first}_\text{i} \leftarrow$ dB[γ].offset
5: **for** wtx **in** $\lambda : (\text{Npb} - 1) : \Lambda$ **do**
6: dlA[wtx] $\leftarrow$ dP[first_i + wtx]
7: **end for**
8: BARRIER(CLK_LOCAL_MEM_FENCE)
9: **for** k **in** $0 : (\text{dB}[\gamma].\text{nn})$ **do**
10: **if** k $= 0$ **then**
11: ptr $\leftarrow \gamma$
12: **else**
13: ptr $\leftarrow$ dB[γ].nei[k $- 1$].number
14: **end if**
15: $\text{first}_\text{j} \leftarrow$ dB[ptr].offset
16: **for** wtx **in** $\lambda : (\text{Npb} - 1) : \Lambda$ **do**
17: dlB[wtx] $\leftarrow$ dP[first_j + wtx]
18: dlQ[wtx] $\leftarrow$ dQ[first_j + wtx]
19: **end for**
20: BARRIER(CLK_LOCAL_MEM_FENCE)
21: **for** wtx **in** $\lambda : (\text{Npb} - 1) : \Lambda$ **do**
22: **for** j **in** $0 : (\text{Npb} - 1)$ **do**
23: r2 $\leftarrow$ dlA[wtx].v + dlB[wtx].v + dlA[wtx].(x,y,z) $\cdot$ dlB[wtx].(x,y,z)
24: u2 $\leftarrow 2 \times \text{alpha}^2 \times$ r22
25: $\text{v}_\text{i,j} \leftarrow e^{-\text{u2}}$
26: fs $\leftarrow 2 \times \text{v}_\text{i,j}$
27: dx $\leftarrow$ dlA[wtx].x $-$ dlB[j].x
28: $\text{fx}_\text{i,j} \leftarrow$ fs $\times$ dx
29: dy $\leftarrow$ dlA[wtx].y $-$ dlB[j].y
30: $\text{fy}_\text{i,j}$fs $\times$ dy
31: dz $\leftarrow$ dlA[wtx].z $-$ dlB[j].z
32: $\text{fz}_\text{i,j}$fs $\times$ dz

Algorithm 50 LavaMD device code abstraction (continues).

```
33:                     dF[first_i + wtx].v ← dlQ[j] × v_{i,j}
34:                     dF[first_i + wtx].x ← dlQ[j] × fx_{i,j}
35:                     dF[first_i + wtx].y ← dlQ[j] × fy_{i,j}
36:                     dF[first_i + wtx].z ← dlQ[j] × fz_{i,j}
37:                 end for
38:                 BARRIER(CLK_LOCAL_MEM_FENCE)
39:               end for
40:               BARRIER(CLK_LOCAL_MEM_FENCE)
41:           end for
42:       end if
43: end kernel
```

- **struct** Box

```
typedef struct Box
{
  int x, y, z; //Coordinates.
  int number; //Global box id.
  long offset; //Box offset.

  int nn; //Number of neighbors.
  struct Neighbor nei[26]; //Neighbors.

};
```

- **struct** Particle

```
struct Particle
{
  int x;
  int y;
  int z;
  int v;
}
```

- **struct** Box hB[Nbx * Nby * Nbz]: $B_{i,j,k} \rightarrow$ hB[i * Nby * Nbx + j * Nbx + k]
- **struct** Particle hP[Nbx * Nby * Nbz * Npb]:
  $B_{i,j,k}[s] \rightarrow$ hP[(i * Nby * Nbx + j * Nbx + k) * Npb + s]
- **struct** Particle hF[Nbx * Nby * Nbz * Npb]:
  $F_{i,j,k}[s] \rightarrow$ hP[(i * Nby * Nbx + j * Nbx + k) * Npb + s]
- **float** hQ[Nbx * Nby * Nbz * Npb]:
  $B_{i,j,k}[s].q \rightarrow$ hQ[(i * Nby * Nbx + j * Nbx + k) * Npb + s]

Device data structures

- **struct** Neighbor

```
struct Neighbor
{
  int x, y, z; //Coordinates.
```

Algorithm 51 LavaMD host code abstraction

```
1: procedure AllocateMemory(Nbx, Nby, Nbz, Npb) ▷ Allocate host memory and device buffers.
2:     hB ← AllocateHostMemory(Nbx × Nby × Nbz)
3:     hP ← AllocateHostMemory(Nbx × Nby × Nbz × Npb)
4:     hF ← AllocateHostMemory(Nbx × Nby × Nbz × Npb)
5:     hQ ← AllocateHostMemory(Nbx × Nby × Nbz × Npb)
6:     dB ← AllocateBuffer(Nbx × Nby × Nbz, READ_WRITE)
7:     dP ← AllocateBuffer(Nbx × Nby × Nbz × Npb, READ_WRITE)
8:     dF ← AllocateBuffer(Nbx × Nby × Nbz × Npb, READ_WRITE)
9:     dQ ← AllocateBuffer(Nbx × Nby × Nbz × Npb, READ_WRITE)
10: end procedure

11: procedure InitializeMemory(Nbx, Nby, Nbz, Npb)          ▷ Initialize host memory.
12:     hP ← RandomElements(Nbx × Nby × Nbz × Npb)
13:     hQ ← RandomElements(Nbx × Nby × Nbz × Npb)
14:     hF ← ZeroElements(Nbx × Nby × Nbz × Npb)
15:     hB ← InitElements(Nbx × Nby × Nbz)
16: end procedure

Input: int Nbx
Input: int Nby
Input: int Nbz                                              ▷ Cube size.
Input: int Npb                                              ▷ Number of particles in each box.
17: AllocateMemory(Nbx, Nby, Nbz, Npb)
18: InitializeMemory(Nbx, Nby, Nbz, Npb)
19: lmdProgram ← BuildProgram("lmdProgram.cl")
20: lmdKernel ← BuildKernel(lmdProgram, "lmdKernel")
21: CopyToDevice(dB, hB, True, ∅)
22: CopyToDevice(dP, hP, True, ∅)
23: CopyToDevice(dF, hF, True, ∅)
24: CopyToDevice(dQ, hQ, True, ∅)
25: S ← Nbx × Nby × Nbz
26: Lsize ← 128
27: SetKernelArguments(lmdKernel, dB, dP, dQ, S, dF, Npb)
28: EnqueueKernel(lmdKernel, (Lsize × S), (Lsize), ∅)
29: CopyToHost(dF, hF, True, ∅)
```

```
  int number; //Global neighbor box id.
  long offset; //Box offset in Particles array.
}
```

- **struct** Box

```
typedef struct Box
{
  int x, y, z; //Coordinates.
  int number; //Global box id.
  long offset; //Box offset.

  int nn; //Number of neighbors.
  struct Neighbor nei[26]; //Neighbors.

};
```

- **struct** Particle

```
struct Particle
{
  int x;
  int y;
  int z;
  int v;
}
```

- **global struct** Box dB[Nbx * Nby * Nbz]: $B_{i,j,k} \rightarrow$ dB[i * Nby * Nbx + j * Nbx + k]
- **global struct** Particle dP[Nbx * Nby * Nbz * Npb]: $B_{i,j,k}[s] \rightarrow$ dP[(i * Nby * Nbx + j * Nbx + k) * Npb + s]
- **global struct** Particle dF[Nbx * Nby * Nbz * Npb]: $F_{i,j,k}[s] \rightarrow$ dP[(i * Nby * Nbx + j * Nbx + k) * Npb + s]
- **global float** dQ[Nbx * Nby * Nbz * Npb]: $B_{i,j,k}[s].q \rightarrow$ dQ[(i * Nby * Nbx + j * Nbx + k) * Npb + s]
- **local float** dlA[Npb]: $B_{i,j,k}[s] \rightarrow$ dlQ[s]
- **local float** dlB[Npb]: $B_{i,j,k}[s] \rightarrow$ dlQ[s]
- **local float** dlQ[Npb]: $B_{i,j,k}[s].q \rightarrow$ dlQ[s]

Input datasets The benchmark uses a randomly generated $10 \times 10 \times 10$ cube of boxes. Each box contains 100 particles with randomly generated attributes.

Partitioned iteration space

$$\begin{aligned} S &= N_{bx} \times N_{by} \times N_{bz} \\ i &= \gamma/(\texttt{Nbx} \times \texttt{Nby}) \\ j &= (\gamma/\texttt{Nbx}) \bmod \texttt{Nby} \\ k &= \gamma \bmod \texttt{Nbx} \end{aligned}$$

$$P = \{(\nu, \lambda, \mathtt{k}, \mathtt{wtx}, \mathtt{j}) : 0 \leq \gamma < \Gamma = S, 0 \leq \lambda < \Lambda = 128, 0 \leq k < |N_{B_{i,j,k}}|, \mathtt{wtx}\ \textbf{in}\ \lambda : (N_{pb} - 1) : \Lambda, 0 \leq j < N_{pb}\}$$

$$f(B_{i,j,k}, B_{i,j,k}[s], (B_{i',j',k'} = N_{B_{i,j,k}}[l]), B_{i',j',k'}[s']) \rightarrow (i \times N_{bx} \times N_{by} + j \times N_{bx} + k, s \bmod (\Lambda), l, (s/\Lambda) \times \Lambda, s')$$

$$f^{-1}(\nu, \lambda, \mathtt{k}, \mathtt{wtx}, \mathtt{j}) \rightarrow (B_{i,j,k}, B_{i,j,k}[\mathtt{wtx} + \lambda], N_{B_{i,j,k}}[\mathtt{k}], N_{B_{i,j,k}}[\mathtt{k}][\mathtt{j}])$$

Device memory access mapping The function, to obtain the neighbor index in the global array based on the its index in local array.

$$ptr(k, \nu) = \mathtt{dB}[\nu].\mathtt{nei}[\mathtt{k} - 1].number$$

Each work-item performs the following reads and writes:

$$\begin{aligned}
G_r(\nu, \lambda, \mathtt{k}, \mathtt{wtx}, \mathtt{j}) &= \{\mathtt{dB}[\nu], \mathtt{dB}[\mathtt{ptr}(\mathtt{k}, \nu)], \mathtt{dQ}[\mathtt{ptr}(\mathtt{k}, \nu) \times N_{pb} + \mathtt{wtx}], \mathtt{dP}[\mathtt{ptr}(\mathtt{k}, \nu) \times N_{pb} + \mathtt{wtx}], \} \\
G_w(\nu, \lambda, \mathtt{k}, \mathtt{wtx}, \mathtt{j}) &= \{\mathtt{dF}[\nu \times N_{pb} + \mathtt{wtx}]\} \\
L_r(\nu, \lambda, \mathtt{k}, \mathtt{wtx}, \mathtt{j}) &= \{\mathtt{dlA}[\mathtt{wtx}], \mathtt{dlB}[\mathtt{j}], \mathtt{dlQ}[\mathtt{j}]\} \\
L_w(\nu, \lambda, \mathtt{k}, \mathtt{wtx}, \mathtt{j}) &= \{\mathtt{dlA}[\mathtt{wtx}], \mathtt{dlB}[\mathtt{wtx}], \mathtt{dlQ}[\mathtt{wtx}]\}
\end{aligned}$$

Performance metrics The performance is evaluated using the following metric:

- Total OpenCL initialization time:
  - OpenCL context creation.
  - OpenCL command queue creation.
  - OpenCL program compilation.
  - OpenCL lmdKernel compilation.
- Device memory allocation time.
- Host to device memory transfer time.
- Kernel (lmdKernel) execution time.
- Device to host memory transfer time.
- Device memory freeing time.

Validation mechanism The benchmark has no validation mechanisms.

Target-specific optimisations

- **Consecutive memory access.** Work items with consecutive global ids in dimension 0 access consecutive memory locations which improves the memory system performance (caching, coalescing).
- **Branching elimination.** In order to avoid heavily branching kernel code, the neighbors information is computed on the host side as a benchmark input (code, which computes this information uses a lot of conditional jumps, which degrade the performance on the device).

Target-specific optimization opportunities

- **Dynamic local work size.** One iteration variable `wtx` and one loop can be eliminated from the kernel code if the local work size is equal to the number of particles per box. In current implementation $\Lambda = 128$ and $N_{pb} = 100$ which results in additional boundary checking (currently performed via while loop).

2.3.9 Leukocyte Tracking

High-level description

The benchmark detects and tracks rolling white blood cells (leukocytes) in the input video stream. Cells are detected in the first video frame and tracked in following frames. See [4] for more details.

The description below is applied to the following parts of the algorithm:

- Cells detection:
  - Image dilation calculation.
  - Gradient Inverse Coefficient of Variation (GICOV) score computation.
- Cells tracking:
  - Motion Gradient Vector Flow (MGVF) computation see [17] for more details.

Only these parts of the algorithm are implemented as OpenCL kernels *i.e.* are the most interesting from GPGPU benchmark analysis point of view.

Abstract data structures The benchmark operates on the following data objects:

- Image dilation:
  - I is an input $I_n \times I_m$ frame (image).
  - M is an input $M_n \times M_m$ dilate mask.
  - D is an intermediate $I_n \times I_m$ dilated image.
- GICOV score computation:
  - N_c denotes the number of ellipses to try.
  - N_p denotes the number of sample points.
  - X is an input $I_n \times I_m$ matrix of image X-gradients.
  - Y is an input $I_n \times I_m$ matrix of image Y-gradients.
  - G is an intermediate $I_n \times I_m$ image GICOV score array.
  - T is an intermediate N_p angles array.
  - C is an intermediate N_p cos array.
  - S is an intermediate N_p sin array.
  - Tx is an intermediate $N_c \times N_p$ offset array.
  - Ty is an intermediate $N_c \times N_p$ offset array.
- MGVF computation:
  - I is an input $I_n \times I_m$ frame (image).
  - W is an intermediate $I_n \times I_m$ MGVF array.

Computation

- Image dilation calculation:

$$D_{i,j} = \max_{0 \le x < M_n, 0 \le y < M_m, M_{x,y} \neq 0} D_{i+x-M_n/2, j+y-M_m/2}, 0 \le i < I_n, 0 \le j < I_m$$

- GICOV score computation (with radius R):

$$\begin{aligned} T_i &= 2 \times \pi \times i / N_p, 0 \le i < N_p \\ C_i &= \cos(T_i), 0 \le i < N_p \\ S_i &= \sin(T_i), 0 \le i < N_p \end{aligned}$$

For every point $(i, j) : R+2 \le i < I_n - R - 2, R+2 \le j < I_m - R - 2$:

$$\begin{aligned} \mathtt{Gt}_{k,n} &= X_{j+\mathtt{Ty}_{k,n}, i+Tx_{k,n}} \times C_n + Y_{j+\mathtt{Ty}_{k,n}, i+Tx_{k,n}} \times S_n, 0 \le k < N_c, 0 \le n < N_p \\ M_k &= \left(\sum_n \mathtt{Gt}_{k,n} \right) / N_p \\ V_k &= \left(\sum_n (\mathtt{Gt}_{k,n} - \mathtt{M}_k)^2 \right) / (N_p - 1) \\ G_{j,i} &= \max_k M_k^2 / V_k \end{aligned}$$

- MGVF computation (for every cell c **in** $0 : (N_c - 1)$):
  - Initialize MGVF array:

$$W^0 = I$$

  - Iteratively compute the MGVF array (W):

$$\begin{aligned} W_{i,j}^{\tau+1} &= W_{i,j}^{\tau} + \frac{\mu}{\lambda} \sum_{l=-1}^{1} \sum_{m=-1}^{1} H_e\left((lv^y + mv^x) \times (W_{i+l,j+m}^{\tau} - W_{i,j}^{\tau}) \right) \times (W_{i+l,j+m}^{\tau} - W_{i,j}^{\tau}) \\ &\quad - \frac{1}{\lambda} I_{i,j} (W_{i,j}^{\tau} - I_{i,j}) \end{aligned}$$

  - Compute new W^{τ} till fixed point is reached:

$$W = W^{\tau}, \| W^{\tau} - W^{\tau-1} \| < \varepsilon$$

  - The $H_e()$ function:

$$H_e(x) \rightarrow (\arctan(x) \times \frac{1}{\pi}) + \frac{1}{2}$$

Iteration spaces

- Image dilation calculation:

$$I_D = \{(i, j, x, y) : 0 \le i < I_n, 0 \le j < I_m, 0 \le x < M_n, 0 \le y < M_m\}$$

- GICOV score computation (with radius R):

$$I_G = \{(i, j, k, n) : R+1 \le i < I_n - R - 2, R+1 \le j < I_m - R - 2, 0 \le k < N_c, 0 \le n < N_p\}$$

- MGVF computation:

$$I_M = \{(c, \tau, i, j) : 0 \le c < N_c, 0 \le \tau, 0 \le i < I_n, 0 \le j < I_m\}$$

Dependences The dependencies are introduced by the other parts of the algorithm (not included in this description).

Memory access mapping

- Image dilation calculation:

$$\begin{aligned} M_r(i,j,x,y) &= \{D_{i+x-S_n/2,j+y-S_m/2}, M_{x,y}\} \\ M_w(i,j,x,y) &= \{D_{i,j}\} \end{aligned}$$

- GICOV score computation (with radius R):

$$\begin{aligned} M_r(i,j,k,n) &= \{\mathtt{Tx}_{k,n}, \mathtt{Ty}_{k,n}, C_n, S_n, \mathtt{Gt}_{k,n}\} \\ M_{mr}(i,j,k,n) &= \{X, Y\} \\ M_w(i,j,k,n) &= \{\mathtt{Gt}_{k,n}, G_{j,i}\} \end{aligned}$$

- MGVF computation:

$$\begin{aligned} M_r(c,\tau,i,j) &= \{I_{i,j}, W^{\tau}_{(i-1):(i+1),(j-1):(j+1)}\} \\ M_w(c,\tau,i,j) &= \{W^{\tau}_{i,j}\} \end{aligned}$$

Low-level implementation details

Device code See Algorithm 52, Algorithm 53, Algorithm 54.

Device data structures

- **int** `dI[In * Im]`: $I_{i,j} \rightarrow$ `dI[i * Im + j]`
- **int** `dW[In * Im]`: $W_{i,j} \rightarrow$ `dW[i * Im + j]`
- **int** `dD[In * Im]`: $D_{i,j} \rightarrow$ `dD[i * Im + j]`
- **int** `dM[Mn * Mm]`: $M_{i,j} \rightarrow$ `dM[i * Mm + j]`
- **int** `dX[In * Im]`: $X_{i,j} \rightarrow$ `dX[i * Im + j]`
- **int** `dY[In * Im]`: $Y_{i,j} \rightarrow$ `dY[i * Im + j]`
- **int** `dG[In * Im]`: $G_{i,j} \rightarrow$ `dG[i * Im + j]`
- **int** `dT[Np]`: $T_i \rightarrow$ `dT[i]`
- **int** `dC[Np]`: $C_i \rightarrow$ `dC[i]`
- **int** `dS[Np]`: $S_i \rightarrow$ `dS[i]`

Algorithm 52 The Leukocyte device code abstraction (GICOV calculation).

Input: `int Im`
Input: `global float dX[]`
Input: `global float dY[]`
Input: `constant float dS[]`
Input: `constant float dC[]`
Input: `constant int dTx[]`
Input: `constant int dTy[]`
Output: `global float dG[]`

1: **kernel** LKKERNEL1 (Im, dX, dX, dS, dC, dTx, dTy, dG)
2: i $\leftarrow \nu +$ R $+ 2$
3: j $\leftarrow \lambda +$ R $+ 2$
4: $G_{max} \leftarrow 0$
5: **for** k **in** $0 : (N_c - 1)$ **do**
6: sum $\leftarrow 0$
7: M2 $\leftarrow 0$
8: mean $\leftarrow 0$
9: **for** n **in** $0 : (N_p - 1)$ **do**
10: y $\leftarrow$ j $+$ dTy[k $\times N_p +$ n]
11: x $\leftarrow$ i $+$ dTx[k $\times N_p +$ n]
12: addr $\leftarrow$ x $\times$ Im $+$ y
13: p $\leftarrow$ dX[addr] $\times$ dC[n] $+$ dY[addr] $\times$ dS[n]
14: sum $\leftarrow$ sum $+$ p
15: delta $\leftarrow$ p $-$ mean
16: mean $\leftarrow$ mean $+$ delta$/$(n $+ 1$)
17: M2 $\leftarrow$ M2 $+$ delta$/$(n $+ 1$)
18: **end for**
19: mean $\leftarrow$ sum$/N_p$
20: var $\leftarrow$ M2$/(N_p - 1)$
21: $G_{max} \leftarrow \max(\text{mean}^2/\text{var}, G_{max})$
22: **end for**
23: dG[i $\times$ Im $+$ j] $\leftarrow G_{max}$
24: **end kernel**

Algorithm 53 The Leukocyte device code abstraction (Dilation calculation).

Input: `int Im`
Input: `int In`
Input: `int Mm`
Input: `int Mn`
Input: `global constant float dM[]`
Input: `global float dI[]`
Output: `global float dD[]`

1: **kernel** LKKERNEL2 (Im, In, Mm, Mn, dM, dI, dD)
2: $C_i \leftarrow Mm/2$
3: $C_j \leftarrow Mn/2$
4: $i \leftarrow \gamma \bmod Im$
5: $j \leftarrow \gamma/Im$
6: $max \leftarrow 0$
7: **for** el_i **in** $0 : (Mm - 1)$ **do**
8: $y \leftarrow i - C_i + el_i$
9: **if** $y \geq 0 \wedge y < Im$ **then**
10: **for** el_j **in** $0 : (Mn - 1)$ **do**
11: $x \leftarrow j - C_j + el_j$
12: **if** $x \geq 0 \wedge x < In \wedge dM[el_i \times Mn + el_j] \neq 0$ **then**
13: $addr \leftarrow x \times Im + y$
14: $tmp \leftarrow dI[addr]$
15: $max \leftarrow \max(max, tmp)$
16: **end if**
17: **end for**
18: **end if**
19: **end for**
20: $dD[i \times In + j] \leftarrow max$
21: **end kernel**

Algorithm 54 The Leukocyte device code abstraction (MGVF calculation).

Input: `global int dI[]`
Input: `constant int dIO[]`
Input: `constant int dMO[]`
Input: `constant int dNO[]`
Input: `float vx`
Input: `float vy`
Input: `float e`
Input: `float cutoff`
Input: `int Imax`
Input/Output: `global float dW[]`
Local: `float dlW[]`
Local: `float dlB`
Local: `int Conv`

1: **kernel** LKKERNEL3 (dI, dIO, dMO, dNO, vx, vy, e, cutoff, Imax, dW)
2: I $\leftarrow$ dIO[v]
3: dpW $\leftarrow$ &(dIW[I])
4: dpI $\leftarrow$ &(dI[I])
5: m $\leftarrow$ &(dMO[v])
6: n $\leftarrow$ &(dNO[v])
7: max $\leftarrow$ (m + n + Λ − 1)/Λ
8: **for** tb **in** 0 : (max − 1) **do**
9: offset $\leftarrow$ tb × Λ
10: i $\leftarrow$ (λ + offset)/n
11: j $\leftarrow$ (λ + offset) mod n
12: **if** i < m **then**
13: dlW[i × n + j] $\leftarrow$ dpW[i × n + j]
14: **end if**
15: **end for**
16: BARRIER(CLK_LOCAL_MEM_FENCE)
17: **if** $\lambda = 0$ **then**
18: flag $\leftarrow$ **true**
19: **end if**
20: BARRIER(CLK_LOCAL_MEM_FENCE)
21: tmod $\leftarrow$ λ mod n
22: tbmod $\leftarrow$ Λ mod n
23: iter $\leftarrow$ 0
24: **while** flag $\wedge$ iter < Imax **do**
25: diff $\leftarrow$ 0
26: j $\leftarrow$ tmod − tbmod ▷ Possible bug here: i is not initialized (value from the previous loop is used).

Algorithm 54 The Leukocyte device code abstraction (IMGVF calculation continues).

```
27:     for tb in 0 : (max − 1) do
28:         oldi ← i
29:         oldj ← j
30:         offset ← tb × Λ
31:         i ← (λ + offset)/n
32:         j ← j + tbmod
33:         if j ≥ n then
34:             j ← j − n
35:         end if
36:         nval ← 0
37:         oval ← 0
38:         if i < m then
39:             oval ←FETCH2D(dlW, i, j)
40:             iU ← max(0, i − 1)
41:             iD ← min(m − 1, i + 1)
42:             jL ← max(0, j − 1)
43:             jR ← min(n − 1, j + 1)
44:             U ←FETCH2D(dlW, iU, j)
45:             D ←FETCH2D(dlW, iD, j)
46:             L ←FETCH2D(dlW, i, jL)
47:             R ←FETCH2D(dlW, i, jR)
48:             UR ←FETCH2D(dlW, iU, jR)
49:             DR ←FETCH2D(dlW, iD, jR)
50:             UL ←FETCH2D(dlW, iU, jL)
51:             DL ←FETCH2D(dlW, iD, jL)
52:             nval ←COMPUTENEWVALUE(U, D, L, R, UR, DR, UL, DL, oval)
53:             vI ← dpI[i × n + j]
54:             nval ← nval − 1/4 × vI × (nval − vI)
55:         end if
56:         if tb > 0 then
57:             offset ← (tb − 1) × Λ
58:             if oldi < m then
59:                 dlW[oldi × n + oldj] ← dlB[λ]
60:             end if
61:         end if
62:         if tb < max − 1 then
63:             dlB[λ] ← nval
64:         else
65:             if i < m then
66:                 dlW[i × n + j] ← nval
67:             end if
68:         end if
69:         diff ← diff + |nval − oval|
70:         BARRIER(CLK_LOCAL_MEM_FENCE)
71:     end for
```

Algorithm 54 The Leukocyte device code abstraction (IMGVF calculation continues).

```
72:         dlB[λ] ← diff
73:         if λ ≥ 2^8 then
74:             dlB[λ − 2^8] ← dlB[λ − 2^8] + dlB[λ]
75:         end if
76:         th ← 2^7
77:         while th > 0 do
78:             if λ < th then
79:                 dlB[λ] ← dlB[λ] + dlB[λ + th]
80:             end if
81:             BARRIER(CLK_LOCAL_MEM_FENCE)
82:             th ← th/2
83:         end while
84:         if λ = 0 then
85:             mean ← dlB[0]/(m × n)
86:             if mean < cutoff then
87:                 flag ← false
88:             end if
89:         end if
90:         BARRIER(CLK_LOCAL_MEM_FENCE)
91:         iter ← iter + 1
92:     end while
93:     for tb in 0 : (max − 1) do
94:         offset ← tb × Λ
95:         i ← (λ + offset)/n
96:         j ← (λ + offset) mod n
97:         if i < m then
98:             dpW[i × n + j] ← dlW[i × n + j]
99:         end if
100:    end for
101: end kernel
```

- **int** `dTx[Nc * Np]`: $\texttt{Tx}_{i,j} \rightarrow$ `dTx[i * Np + j]`
- **int** `dTy[Nc * Np]`: $\texttt{Ty}_{i,j} \rightarrow$ `dTy[i * Np + j]`
- **int** `dIO[Nc]`: Intermediate offsets buffer for image buffer `dI`
- **int** `dNO[Nc]`: Intermediate boundaries buffer for image buffer `dI`
- **int** `dMO[Nc]`: Intermediate boundaries buffer for image buffer `dI`
- **int** `dlW[41 * 81]`: Local buffer for `dW`.
- **int** `dlB[`Λ`]`: Local buffer to compute reduction.

Input datasets The benchmark uses pre generated input video file with following parameters:

- Number of frames to process. Can be adjusted via command line. The default value is 10.
- Size of the video frame *i.e.* single image ($I_n \times I_m$) = (640×480).

Partitioned iteration space

- Image dilation calculation (lkKernel2):

$$P_D = \{(\gamma, i, j) : 0 \leq \gamma < \Gamma = I_m \times I_n, \}$$

$$f(i, j, x, y) \rightarrow (i \times I_m + j, x, y)$$

$$f^{-1}(\gamma, i, j) \rightarrow (\gamma / I_m, \gamma \bmod I_m, i, j)$$

- GICOV score computation (lkKernel1):

$$P_G = \{(\nu, \lambda, k, n) : 0 \leq \nu < N = (I_n - 2 \times R), 0 \leq \lambda < \Lambda = (I_n - 2 \times R), 0 \leq k < N_c, 0 \leq n < N_p\}$$

$$f(i, j, k, n) \rightarrow (i - R - 2, j - R - 2, k, n)$$

$$f^{-1}(\nu, \lambda, k, n) \rightarrow (\nu + R + 2, \lambda + R + 2, k, n)$$

- MGVF computation (lkKernel3):

$$P_G = \{(\nu, \lambda, \tau, t) : 0 \leq \nu < N = Nc, 0 \leq \lambda < \Lambda = 256, 0 \leq \tau, 0 \leq t < I_m \times I_n / \Lambda\}$$

$$f(c, \tau, i, j) \rightarrow (c, (i \times n + j) \bmod \Lambda, \tau, (i \times n + j) / \Lambda)$$

$$f^{-1}(\nu, \lambda, \tau, t) \rightarrow (\nu, \tau, (\lambda + t \times \Lambda) / I_n, (\lambda + t \times \Lambda) \bmod I_n)$$

Device memory access mapping

- Image dilation calculation (lkKernel2):

$$\begin{aligned} G_r(\gamma,i,j) &= \{\texttt{dM}[i \times M_n + j], \texttt{dI}[(\gamma/I_m - M_n/ + j) \times I_m + \gamma \bmod I_m - M_m/2 + i],\} \\ G_w(\gamma i,j) &= \{\texttt{dI}[(\gamma \bmod I_m) \times I_n + (\gamma/I_m)]\} \end{aligned}$$

- GICOV score computation (lkKernel1):

$$\begin{aligned} G_r(\nu,\lambda,k,n) &= \{\texttt{dTy}[k \times N_p + n], \texttt{dTx}[k \times N_p + n], \texttt{dS}[n], \texttt{dC}[n],\} \\ G_{mr}(\nu,\lambda,k,n) &= \{\texttt{dX}[], \texttt{dY}[]\} \\ G_w(\nu,\lambda,k,n) &= \{\texttt{dG}[(\nu + R + 2) \times I_m + (\lambda + R + 2)]\} \end{aligned}$$

- MGVF computation (lkKernel3):

$$\begin{aligned} G_r(\nu,\lambda,\tau,t) &= \{\texttt{dIO}[\nu], \texttt{dMO}[\nu], \texttt{dNO}[\nu]\} \\ G_{mr}(\nu,\lambda,\tau,t) &= \{\texttt{dW}[], \texttt{dI}[]\} \\ G_{mw}(\nu,\lambda,\tau,t) &= \{\texttt{dW}[]\} \\ L_r(\nu,\lambda,\tau,t) &= \{\texttt{dlB}[\lambda]\texttt{dlB}[\lambda - 2^8]|_{\lambda \geq 2^8}, \texttt{dlB}[\lambda + 2^i]|_{\lambda < 2^i, 0 < i < 8}\} \\ L_{mr}(\nu,\lambda,\tau,t) &= \{\texttt{dlW}[]\} \\ L_w(\nu,\lambda,\tau,t) &= \{\texttt{dlW}[\lambda + t \times \Lambda], \texttt{dlB}[\lambda]\} \end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- GICOV score computation time.
- Dilation computation time.
- MGVF computation time.
- Total tracking time.
- Total application execution time.

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can print number of detected cells, which can be validated by external tools (not supplied).

Target-specific optimizations

- **Consecutive memory access.** Work items with consecutive global ids access consecutive memory locations which improves the memory system performance (caching, coalescing).
- **Local memory usage.** In order to decrease the number of global memory accesses each work group allocates local buffers, which are used to perform all computations. This technique also increases data locality since the data layout can be changed while copying from global to local memory and vice versa.

2.3.10 LU Decomposition

High-level description

The benchmark computes the LU Decomposition of a square matrix M:

$$M = L \times U$$

where L is a lower triangular matrix and U is an upper triangular matrix.

Abstract data structures The benchmark operates on the following floating point data objects:

- M is an input $n \times n$ square matrix.
- L is an output $n \times n$ square lower triangular matrix.
- U is an output $n \times n$ square upper triangular matrix.

Computation The matrices are computed as follows:

$$\begin{aligned} U_{i,j} &= M_{i,j} - \sum_{k=0}^{i-1} L_{i,k} \times U_{k,j} \quad 0 \le i < n, i \le j < n \\ L_{j,i} &= \frac{1}{U_{i,i}} \times (M_{j,i} - \sum_{k=0}^{i-1} L_{j,k} \times U_{k,i}) \quad 0 \le i < n, i < j < n \end{aligned}$$

Iteration spaces

- Compute U:

$$I_U = \{(i,j,k) : 0 \le i < n, i \le j < n, 0 \le k < i\}$$

$$I_U : \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ k \end{pmatrix} \le \begin{pmatrix} n-1 \\ n-1 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- Compute L:

$$I_L = \{(i,j,k) : 0 \le i < n, i < j < n, 0 \le k < i\}$$

$$I_L : \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ k \end{pmatrix} \le \begin{pmatrix} n-1 \\ n-1 \\ -1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

Dependences

$$\begin{aligned}
\delta_{I_U} &= I_U : (k, j, 0 : (k-1)) \xrightarrow{t} I_U : (i, j, k) \wedge \\
&\quad\ I_L : (k, i, 0 : (k-1)) \xrightarrow{t} I_U : (i, j, k) \\
\delta_{I_L} &= I_U : (i, i, 0 : (i-1)) \xrightarrow{t} I_L : (i, j, k) \wedge \\
&\quad\ I_U : (k, i, 0 : (k-1)) \xrightarrow{t} I_L : (i, j, k) \wedge \\
&\quad\ I_L : (k, j, 0 : (k-1)) \xrightarrow{t} I_L : (i, j, k)
\end{aligned}$$

Memory access mapping

- Compute U:

$$\begin{aligned}
M_r(i, j, k) &= \{M_{i,j}, L_{i,k}, U_{k,j}\} \\
M_w(i, j, k) &= \{U_{i,j}\}
\end{aligned}$$

- Compute L:

$$\begin{aligned}
M_r(i, j, k) &= \{U_{i,i}, U_{k,i}, M_{j,i}, L_{j,k}\} \\
M_w(i, j, k) &= \{L_{j,i}\}
\end{aligned}$$

Low-level implementation details

The computation is performed by three OpenCL kernels, which update the input matrix in-place. Figure 2.1 shows the elements updated by each kernel (`offset` is incremented by block size $B = 16$ on each iteration).

Device code See Algorithm 55, Algorithm 56 and Algorithm 57.

Host code See Algorithm 58.

Host data structures

- **`global float`** `hM[n * n]`: $M_{i,j}, L_{i,j}, U_{i,j} \rightarrow$ `hM[i * n + j]`

Device data structures See Figure 2.2.

- **`global float`** `dM[n * n]`: $M_{i,j}, L_{i,j}, U_{i,j} \rightarrow$ `dM[i * n + j]`
- **`local float`** `dlD[B * B]`: $M_{i,j}, L_{i,j}, U_{i,j} \rightarrow$ `dlD[(i % B) * B + j % B]`, $0 \leq i < n, ((i/B) \times B) \leq j < ((i/B)+1) \times B)$ (only blocks on the main diagonal).
- **`local float`** `dlR[B * B]`: $M_{i,j}, L_{i,j}, U_{i,j} \rightarrow$ `dlR[(i % B) * B + j % B]`
- **`local float`** `dlC[B * B]`: $M_{i,j}, L_{i,j}, U_{i,j} \rightarrow$ `dlC[(i % B) * B + j % B]`

Algorithm 55 The LU Decomposition (Diagonal elements) device code abstraction.

Input/Output: `global float dM[]`
Local: `local float dlD[]`
Input: `int n`
Input: `int offset`

1: **kernel** LUDKERNEL1 (dM, n, offset)
2: dMoffset $\leftarrow$ offset $\times$ n + offset
3: **for** i **in** $0 : (B-1)$ **do** ▷ Prefetch global data to local buffers (dlD).
4: dlD[i $\times$ B + λ] $\leftarrow$ dM[dMoffset + i $\times$ n + λ]
5: **end for**
6: BARRIER(CLK_LOCAL_MEM_FENCE)
7: **for** i **in** $0 : (B-1)$ **do** ▷ Compute diagonal block (dlD).
8: **if** $\lambda >$ i **then**
9: **for** j **in** $0 :$ (i -1) **do**
10: dlD[$\lambda \times$ B + i] $\leftarrow$ dlD[$\lambda \times$ B + i] $-$ dlD[$\lambda \times$ B + j] $\times$ dlD[j $\times$ B + i]
11: dlD[$\lambda \times$ B + i] $\leftarrow$ dlD[$\lambda \times$ B + i]/dlD[i $\times$ B + i]
12: **end for**
13: **end if**
14: BARRIER(CLK_LOCAL_MEM_FENCE)
15: **if** $\lambda >$ i **then**
16: **for** j **in** $0 :$ i **do**
17: dlD[(i + 1) $\times$ B + λ] $\leftarrow$ dlD[(i + 1) $\times$ B + λ] $-$ dlD[(i + 1) $\times$ B + j] $\times$ dlD[j $\times$ B + λ]
18: **end for**
19: **end if**
20: BARRIER(CLK_LOCAL_MEM_FENCE)
21: **end for**
22: dMoffset $\leftarrow$ (offset + 1) $\times$ n + offset
23: **for** i **in** $0 : (B-1)$ **do** ▷ Write local buffers to global memory (dlD).
24: dM[dMoffset + λ] $\leftarrow$ dlD[i $\times$ B + i $\times$ n + λ]
25: **end for**
26: **end kernel**

Algorithm 56 The LU Decomposition (Perimeter Elements) device code abstraction.

Input/Output: `global float dM[]`
Local: `local float dlR[]`
Local: `local float dlC[]`
Input: `local float dlD[]`
Input: `int n`
Input: `int offset`

1: **kernel** LUDKERNEL2 (dM, n, offset)
2: **if** $\lambda <$ B **then** ▷ Prefetch global data to local buffers.
3: idx $\leftarrow \lambda$
4: dMoffset $\leftarrow$ offset $\times$ n + offset
5: **for** i **in** $0 : (B/2 - 1)$ **do** ▷ Prefetch dlD (rows 0:(B/2 - 1).
6: dlD[i $\times$ B + idx] $\leftarrow$ dM[dMoffset + i $\times$ n + idx]
7: **end for**
8: dMoffset $\leftarrow$ offset $\times$ n + offset
9: **for** i **in** $0 : (B - 1)$ **do** ▷ Prefetch dlR.
10: dlR[i $\times$ B + idx] $\leftarrow$ dM[dMoffset + $(\nu + 1) \times$ B + i $\times$ n + idx]
11: **end for**
12: **else**
13: idx $\leftarrow \lambda -$ B
14: dMoffset $\leftarrow$ (offset) $\times$ n + offset
15: **for** i **in** $(B/2) : (B - 1)$ **do** ▷ Prefetch dlD (rows (B/2):(B-1).
16: dlD[i $\times$ B + idx] $\leftarrow$ dM[dMoffset + i $\times$ n + idx]
17: **end for**
18: dMoffset $\leftarrow$ (offset + $(\nu + 1) \times$ B) $\times$ n + offset
19: **for** i **in** $0 : (B - 1)$ **do** ▷ Prefetch dlC.
20: dlC[i $\times$ B + idx] $\leftarrow$ dM[dMoffset + i $\times$ n + idx]
21: **end for**
22: **end if**
23: BARRIER(CLK_LOCAL_MEM_FENCE)

Algorithm 56 The LU Decomposition (Perimeter Elements) device code abstraction (continues).

```
24:     if λ < B then                                   ▷ Compute data in local memory.
25:         idx ← λ
26:         for i in 1 : (B − 1) do
27:             for j in 0 : (i − 1) do                 ▷ Compute dlR.
28:                 dlR[i × B + idx] = dlR[i × B + idx] − dlD[i × B + j] × dlR[j × B + idx]
29:             end for
30:         end for
31:     else
32:         idx ← λ − B
33:         for i in 0 : (B − 1) do
34:             for j in 0 : (i − 1) do                 ▷ Compute dlC.
35:                 dlC[idx × B + i] = dlC[idx × B + i] − dlC[idx × B + j] × dlD[j × B + i]
36:             end for
37:             dlC[idx × B + i] = dlC[idx × B + i]/dlD[i × B + i]
38:         end for
39:     end if
40:     BARRIER(CLK_LOCAL_MEM_FENCE)
41:     if λ < B then                                   ▷ Write local buffers to global memory.
42:         idx ← λ
43:         dMoffset ← (offset) × n + offset
44:         for i in 1 : (B − 1) do                     ▷ Write dlR.
45:             dM[dMoffset + (v + 1) × B + i × n + idx] ← dlR[i × B + idx]
46:         end for
47:     else
48:         idx ← λ − B
49:         dMoffset ← (offset + (v + 1) × B) × n + offset
50:         for i in 0 : (B − 1) do                     ▷ Write dlC.
51:             dM[dMoffset + i × n + idx] ← dlC[i × B + idx]
52:         end for
53:     end if
54: end kernel
```

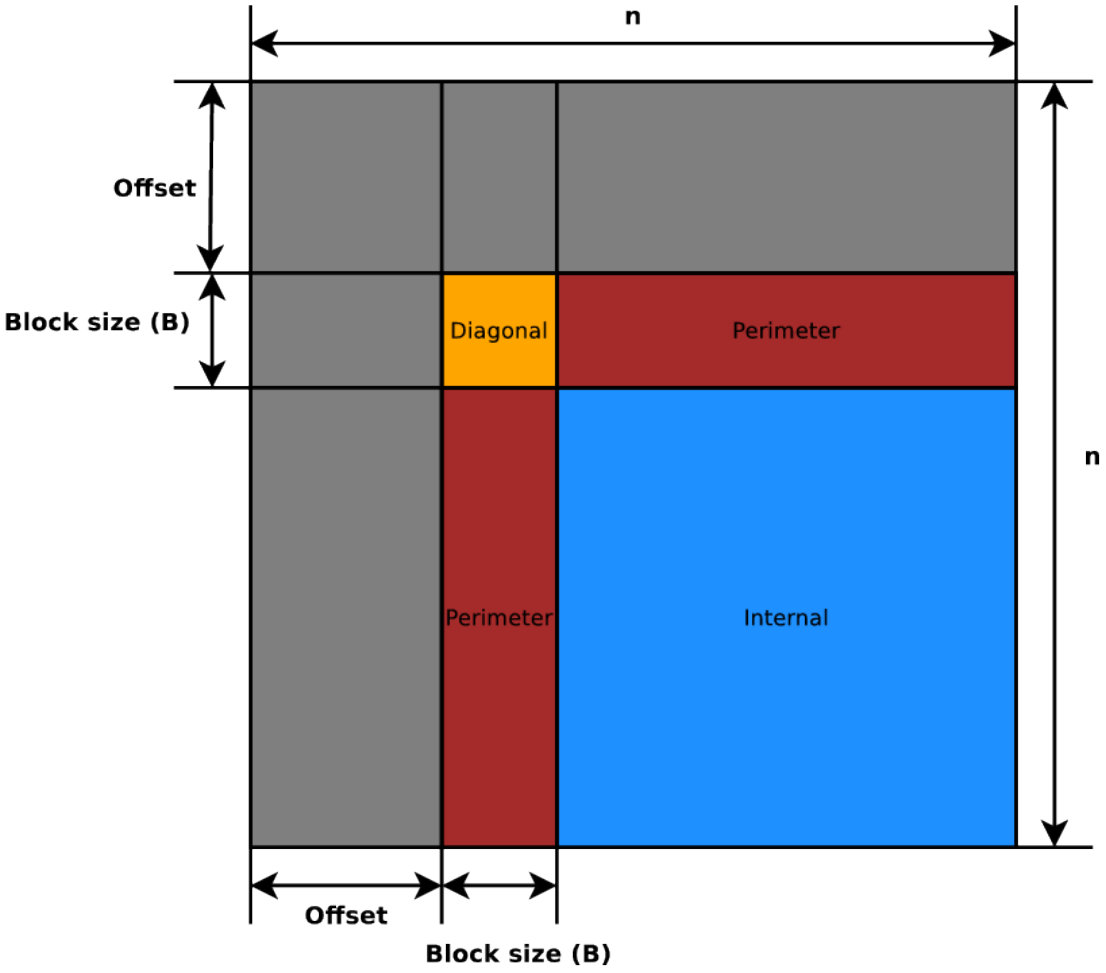

Figure 2.1: The LU Decomposition partitioned iteration space.

Input datasets The benchmark uses pre-generated matrices of single-precision floating point values $\in (0,1)$ of the following sizes: 64×64, 256×256, 512×512, 2048×2048.

Partitioned iteration space

- Compute diagonal elements (ludKernel1):

$$\begin{aligned} P_1 = \{(\texttt{offset}, \lambda, \texttt{i}, \texttt{j}) : \quad & \texttt{offset} \textbf{ in } 0 : (\texttt{n} - \texttt{B} - 1) : \texttt{B}, \\ & 0 \leq \texttt{i} < \texttt{B}, \\ & \texttt{i} < \lambda < \Lambda = \texttt{B}, \\ & 0 \leq \texttt{j} \leq \texttt{i}\} \end{aligned}$$

Algorithm 57 The LU Decomposition (Internal Elements) device code abstraction.

Input/Output: `global float dM[]`
Local: `local float dlR[]`
Local: `local float dlC[]`
Input: `int n`
Input: `int offset`

1: **kernel** LUDKERNEL3 (dM, n, offset)
2: globalRow $\leftarrow$ offset $+ (v_1 + 1) \times$ B
3: globalCol $\leftarrow$ offset $+ (v_0 + 1) \times$ B
4: dlR$[\lambda_1 \times$ B $+ \lambda_0] \leftarrow$ dM$[($offset $+ \lambda_1) \times$ n $+$ globalCol $+ \lambda_0]$
5: dlC$[\lambda_1 \times$ B $+ \lambda_0] \leftarrow$ dM$[($globalRow $+ \lambda_1) \times$ n $+$ offset $+ \lambda_0]$
6: BARRIER(CLK_LOCAL_MEM_FENCE)
7: sum $\leftarrow 0$
8: **for** i **in** $0 : ($B $- 1)$ **do**
9: sum $\leftarrow$ sum $+$ dlC$[\lambda_1 \times$ B $+$ i$] \times$ dlR$[$i $\times$ B $+ \lambda_0]$
10: **end for**
11: dM$[($globalRow $+ \lambda_1) \times$ n $+$ globalCol $+ \lambda_0] \leftarrow$ dM$[($globalRow $+ \lambda_1) \times$ n $+$ globalCol $+ \lambda_0] -$ sum
12: **end kernel**

$$
\begin{aligned}
I_U : f(i,j,k) &\rightarrow ((\texttt{offset} = ((i \bmod \texttt{B}) \times \texttt{B})), i - \texttt{offset}, j - \texttt{offset}, k - \texttt{offset}) \\
I_L : f(i,j,k) &\rightarrow ((\texttt{offset} = ((j \bmod \texttt{B}) \times \texttt{B})), j - \texttt{offset}, i - \texttt{offset}, k - \texttt{offset})
\end{aligned}
$$

$$
\begin{aligned}
f^{-1}(\texttt{offset}, \lambda, \texttt{i}, \texttt{j}) \rightarrow\ & P_U : (\texttt{offset} + \lambda, \texttt{offset} + \texttt{i}, \texttt{offset} + \texttt{j}) \wedge \\
& P_L : (\texttt{offset} + \texttt{i}, \texttt{offset} + \lambda, \texttt{offset} + \texttt{j})
\end{aligned}
$$

- Compute perimeter elements (ludKernel2):

$$
\begin{aligned}
P_2 = \{(\texttt{offset}, v, \lambda, \texttt{i}, \texttt{j}) :\ & \texttt{offset}\ \textbf{in}\ 0 : (\texttt{n} - \texttt{B} - 1) : \texttt{B}, \\
& 0 \leq v < N = (\texttt{n} - \texttt{offset})/\texttt{B} - 1, \\
& 0 \leq \lambda < \Lambda = 2 \times \texttt{B}, \\
& 0 \leq \texttt{i} < \texttt{B}, \\
& 0 \leq \texttt{j} < \texttt{i}\}
\end{aligned}
$$

$$
\begin{aligned}
I_U : f(i,j,k) &\rightarrow ((\texttt{offset} = ((k \bmod \texttt{B}) \times \texttt{B})), (j - \texttt{offset})/B - 1, i - \texttt{offset} - 1, k - \texttt{offset}) \\
I_L : f(i,j,k) &\rightarrow ((\texttt{offset} = ((k \bmod \texttt{B}) \times \texttt{B})), (i - \texttt{offset})/B - 1, j - \texttt{offset} - 1, k - \texttt{offset})
\end{aligned}
$$

Algorithm 58 The LU Decomposition host code abstraction.

```
1: procedure AllocateMemory(n)                    ▷ Allocate host memory and device buffers.
2:     hM ← AllocateHostMemory(n × n)
3:     dM ← AllocateBuffer(n × n, READ_WRITE)
4: end procedure

5: procedure InitializeMemory(n)                  ▷ Initialise host memory.
6:     hM ← ReadFromFile(n × n)
7: end procedure

Input: int n                                      ▷ Input matrix size.
8: AllocateMemory(n)
9: InitializeMemory(n)
10: ludProgram ← BuildProgram("ludProgram.cl")
11: ludKernel1 ← BuildKernel(ludProgram, "ludKernel1")
12: ludKernel2 ← BuildKernel(ludProgram, "ludKernel2")
13: ludKernel3 ← BuildKernel(ludProgram, "ludKernel3")
14: B ← 16                                        ▷ Block size.
15: CopyToDevice(dM, hM, True, ∅)
16: for offset in 0 : (n − 1 − B) : B do
17:     SetKernelArguments(ludKernel1, dM, n, offset)
18:     EnqueueKernel(ludKernel1, (B), (B), ∅)
19:     SetKernelArguments(ludKernel2, dM, n, offset)
20:     EnqueueKernel(ludKernel2, (B × 2 × ((n − offset)/B − 1)), (B × 2), ∅)
21:     SetKernelArguments(ludKernel3, dM, n, offset)
22:     EnqueueKernel(ludKernel3, (B × ((n − offset)/B − 1), B × ((n − offset)/B − 1)), (B, B), ∅)
23: end for
24: CopyToHost(dM, hM, True, ∅)
```

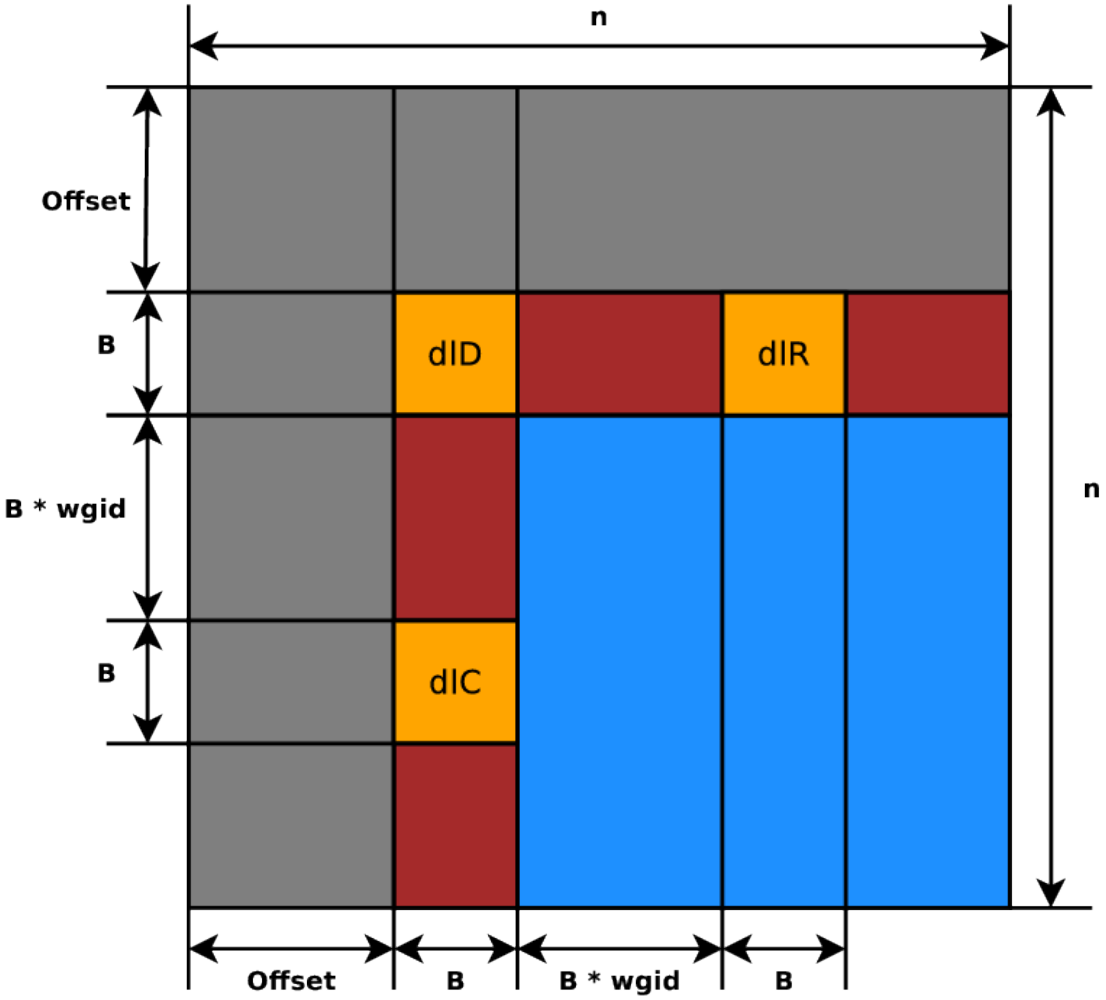

Figure 2.2: The LU Decomposition partitioned iteration space.

$$f^{-1}(\texttt{offset}, v, \lambda, \texttt{i}, \texttt{j}) \rightarrow P_U : (\texttt{offset} + 1 + i, \texttt{offset} + (v+1) \times B + \lambda, \texttt{offset} + j) \wedge P_L : (\texttt{offset} + (v+1) \times B + \lambda, \texttt{offset} + 1 + i, \texttt{offset} + j)$$

- Compute internal elements (ludKernel3):

$$\begin{aligned} P_3 = \{(\texttt{offset}, \nu_0, \nu_1, \lambda_0, \lambda_1, \texttt{i}) : \quad & \texttt{offset}\ \textbf{in}\ 0 : (\texttt{n} - \texttt{B} - 1) : \texttt{B}, \\ & 0 \leq \nu_0 < N_0 = (\texttt{n} - \texttt{offset})/\texttt{B} - 1, \\ & 0 \leq \nu_1 < N_1 = (\texttt{n} - \texttt{offset})/\texttt{B} - 1, \\ & 0 \leq \lambda_0 < \Lambda_0 = \texttt{B}, \\ & 0 \leq \lambda_1 < \Lambda_1 = \texttt{B}, \\ & 0 \leq \texttt{i} < \texttt{B}\} \end{aligned}$$

$$\begin{aligned} I_U : f(i,j,k) \quad \rightarrow \quad & ((\texttt{offset} = ((k \bmod \texttt{B}) \times \texttt{B})), (j - \texttt{offset})/B - 1, (i - \texttt{offset})/B - 1, \\ & (j - \texttt{offset}) \bmod B, (i - \texttt{offset}) \bmod B, k - \texttt{offset}) \\ I_L : f(i_L, j_L, k_L) \quad \rightarrow \quad & ((\texttt{offset} = ((k \bmod \texttt{B}) \times \texttt{B})), (i - \texttt{offset})/B - 1, (j - \texttt{offset})/B - 1, \\ & (i - \texttt{offset}) \bmod B, (j - \texttt{offset}) \bmod B, k - \texttt{offset}) \end{aligned}$$

$$\begin{aligned} f^{-1}(\texttt{offset}, \nu_0, \nu_1, \lambda_0, \lambda_1, \texttt{i}) \quad \rightarrow \quad & P_U : (\texttt{offset} + (\nu_1 + 1) \times B + \lambda_1, \\ & \texttt{offset} + (\nu_0 + 1) \times B + \lambda_0, \texttt{offset} + \texttt{i}) \wedge \\ & P_L : (\texttt{offset} + (\gamma_0 + 1) \times B + \lambda_0, \\ & \texttt{offset} + (\nu_1 + 1) \times B + \lambda_1, \texttt{offset} + \texttt{i}) \end{aligned}$$

Device memory access mapping

- Compute diagonal elements (ludKernel1):

$$\begin{aligned} G_r(\texttt{offset}, \lambda, \texttt{i}, \texttt{j}) &= \{\texttt{dM}[\texttt{offset} \times (\texttt{n} + 1) + \texttt{i} \times \texttt{n} + \lambda]\} \\ G_w(\texttt{offset}, \lambda, \texttt{i}, \texttt{j}) &= \{\texttt{dM}[\texttt{offset} \times (\texttt{n} + 1) + \texttt{i} \times \texttt{n} + \texttt{n} + \lambda]\} \\ L_r(\texttt{offset}, \lambda, \texttt{i}, \texttt{j}) &= \{\texttt{dlD}[\lambda \times \texttt{B} + \texttt{i}], \texttt{dlD}[\lambda \times \texttt{B} + \texttt{j}], \texttt{dlD}[\texttt{i} \times \texttt{B} + \texttt{i}], \texttt{dlD}[\texttt{j} \times \texttt{B} + \texttt{i}], \\ & \quad \texttt{dlD}[\texttt{j} \times \texttt{B} + \lambda], \texttt{dlD}[(\texttt{i} + 1) \times \texttt{B} + \texttt{j}], \texttt{dlD}[(\texttt{i} + 1) \times \texttt{B} + \lambda]\} \\ L_w(\texttt{offset}, \lambda, \texttt{i}, \texttt{j}) &= \{\texttt{dlD}[\lambda \times \texttt{B} + \texttt{i}], \texttt{dlD}[\texttt{i} \times \texttt{B} + \lambda], \texttt{dlD}[(\texttt{i} + 1) \times \texttt{B} + \texttt{i}], \} \end{aligned}$$

- Compute perimeter elements (ludKernel2):

$$
\begin{aligned}
G_r(\texttt{offset},\nu,\lambda,\texttt{i},\texttt{j}) &= \{\texttt{dM}[\texttt{offset}\times(\texttt{n}+1)+\texttt{n}\times\texttt{i}+\lambda]_{\lambda<\texttt{B}},\\
&\quad \texttt{dM}[\texttt{offset}\times(\texttt{n}+1)+\texttt{n}\times\texttt{i}+\lambda+(\nu+1)\times\texttt{B}]_{\lambda<\texttt{B}},\\
&\quad \texttt{dM}[(\texttt{offset}+\texttt{B}/2)\times\texttt{n}+\texttt{offset}+\texttt{n}\times\texttt{i}+\lambda-\texttt{B}]_{\lambda\geq\texttt{B}},\\
&\quad \texttt{dM}[(\texttt{offset}+(\nu+1)\times\texttt{B})\times\texttt{n}+\texttt{offset}+\texttt{n}\times\texttt{i}+\lambda-\texttt{B}]_{\lambda\geq\texttt{B}}\}\\
G_w(\texttt{offset},\nu,\lambda,\texttt{i},\texttt{j}) &= \{\texttt{dM}[\texttt{offset}\times(\texttt{n}+1)+\texttt{n}\times\texttt{i}+\lambda+(\nu+1)\times\texttt{B}]_{\lambda<\texttt{B}},\\
&\quad \texttt{dM}[(\texttt{offset}+(\nu+1)\times\texttt{B})\times\texttt{n}+\texttt{offset}+\texttt{n}\times\texttt{i}+\lambda-\texttt{B}]_{\lambda\geq\texttt{B}}\}\\
L_r(\texttt{offset},\nu_0,\nu_1,\lambda_0,\lambda_1,\texttt{i},\texttt{j}) &= \{\texttt{dlR}[\texttt{i}\times\texttt{B}+\lambda]_{\lambda<\texttt{B}},\texttt{dlR}[\texttt{i}\times\texttt{B}+\texttt{j}]_{\lambda<\texttt{B}},\\
&\quad \texttt{dlR}[\texttt{j}\times\texttt{B}+\lambda]_{\lambda<\texttt{B}},\texttt{dlC}[\texttt{i}\times\texttt{B}+\lambda-\texttt{B}]_{\lambda\geq\texttt{B}},\\
&\quad \texttt{dlC}[(\lambda-\texttt{B})\times\texttt{B}+\texttt{i}]_{\lambda\geq\texttt{B}},\texttt{dlC}[(\lambda-\texttt{B})\times\texttt{B}+\texttt{j}]_{\lambda\geq\texttt{B}},\\
&\quad \texttt{dlC}[\texttt{j}\times\texttt{B}+\texttt{i}]_{\lambda\geq\texttt{B}},\texttt{dlD}[\texttt{i}\times\texttt{B}+\texttt{i}]_{\lambda\geq\texttt{B}}\}\\
L_w(\texttt{offset},\nu_0,\nu_1,\lambda_0,\lambda_1,\texttt{i},\texttt{j}) &= \{\texttt{dlD}[\texttt{i}\times\texttt{B}+\lambda]_{\lambda<\texttt{B}},\texttt{dlR}[\texttt{i}\times\texttt{B}+\lambda]_{\lambda<\texttt{B}},\\
&\quad \texttt{dlD}[\texttt{i}\times\texttt{B}+\lambda-\texttt{B}]_{\lambda\geq\texttt{B}},\texttt{dlC}[\texttt{i}\times\texttt{B}+\lambda-\texttt{B}]_{\lambda\geq\texttt{B}},\\
&\quad \texttt{dlC}[(\lambda-\texttt{B})\times\texttt{B}+\texttt{i}]_{\lambda\geq\texttt{B}}\}
\end{aligned}
$$

- Compute internal elements (ludKernel3):

$$
\begin{aligned}
G_r(\texttt{offset},\nu_0,\nu_1,\lambda_0,\lambda_1,\texttt{i}) &= \{\texttt{dM}[(\texttt{offset}+\lambda_1)\times\texttt{n}+\texttt{offset}+(\nu_0+1)\times\texttt{B}+\lambda_0],\\
&\quad \texttt{dM}[(\texttt{offset}+(\nu_1+1)\times\texttt{B}+\lambda_1)\times\texttt{n}+\texttt{offset}+\lambda_0]\}\\
G_w(\texttt{offset},\nu_0,\nu_1,\lambda_0,\lambda_1,\texttt{i}) &= \{\texttt{dM}[(\texttt{offset}+(\nu_1+1)\times\texttt{B}+\lambda_1)\times\texttt{n}+\texttt{offset}+\lambda_0]\}\\
L_r(\texttt{offset},\nu_0,\nu_1,\lambda_0,\lambda_1,\texttt{i}) &= \{\texttt{dlR}[\texttt{i}\times\texttt{B}+\lambda_0],\texttt{dlC}[\lambda_1\times\texttt{B}+\texttt{i}]\}\\
L_w(\texttt{offset},\nu_0,\nu_1,\lambda_0,\lambda_1,\texttt{i}) &= \{\texttt{dlR}[\lambda_1\times\texttt{B}+\lambda_0],\texttt{dlC}[\lambda_1\times\texttt{B}+\lambda_0]\}
\end{aligned}
$$

Performance metrics The performance of the algorithm implementation is evaluated as the total time required by the OpenCL implementation to perform:

1. Write data to device.
2. Perform computations on device.
3. Read data from device.

Validation mechanism The benchmark multiplies matrix L by matrix U and compares the result to the original matrix with error tolerance 10^{-4}.

Target-specific optimizations The following target-specific optimization are used in the algorithm implementation:

- **Consecutive memory access.** Work items with consecutive global ids access consecutive memory locations which improves the memory system performance (caching, coalescing).

- **Local memory usage.** In order to decrease the number of global memory accesses each work group allocates local buffers, which are used to perform all computations. This technique also increases data locality since the data layout can be changed while copying from global to local memory and vice versa.

2.3.11 k-Nearest Neighbors

Give a point B and a set of points $\{P_i\}$ in a two-dimensional Euclidean space, find k nearest points from $\{P_i\}$ to B.

High-level description

Abstract data structures The benchmark operates on the following objects:

- Each point is a pair $(\mathtt{x},\mathtt{y})$, where x is the latitude and y is the longitude of the point.
- P is an input array of n points.
- D is an intermediate array of distances.
- B is an input point.
- R is an output array of k points nearest to B.

Computation The algorithm consists of two steps:

1. Distance Calculation: For each point P_i, $0 \leq i < n$, the distance from P_i to B is calculated:

$$D_i = \sqrt{(P_i^x - B^x)^2 + (P_i^y - B^y)^2}$$

2. Neighbors Selection: k points with the minimal distance from B are selected:

$$R = \{p : |R| = k \wedge \forall p \in R, p' \notin R \rightarrow D_p \leq D_{p'}\}$$

Iteration domain and dependences This algorithm requires several iteration spaces:

- Distance Calculation:

$$I_D = \{(i) : 0 \leq i < n\}$$

$$I_D : \begin{pmatrix} 1 \\ -1 \end{pmatrix} \times (i) \leq \begin{pmatrix} n-1 \\ 0 \end{pmatrix}$$

- Neighbors Selection:

$$I_R = \{(i,j) : 0 \leq i < k, i \leq j < n\}$$

$$I_R : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 1 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} k-1 \\ n-1 \\ 0 \\ -1 \end{pmatrix}$$

$$I = I_D \cup I_R$$

The computation generates the following dependences:

$$
\begin{aligned}
\delta_{I_R} \quad = \quad & I_R : (i, i : (j-1)) \xrightarrow{t} I_R : (i,j) \wedge \\
& I_R : (0 : (i-1), n-1) \xrightarrow{t} I_R : (i,j) \wedge \\
& I_R : (0 : (i-1), 0 : (n-1)) \xrightarrow{t} I_R : (i,j) \wedge \\
& I_D : (i : j) \xrightarrow{t} I_R : (i,j) \wedge \\
& I_D : (0 : (n-1)) \xrightarrow{t} I_R : (i, n-1) \wedge \\
& I_D : (0 : (n-1)) \xrightarrow{a} I_R : (i, n-1)
\end{aligned}
$$

Memory access mapping The following data accesses are performed in each of the iteration spaces:

- Distance calculation: for $(i) \in I_D$:

$$
\begin{aligned}
M_r(i) &= \{P_i\} \\
M_w(i) &= \{D_i\}
\end{aligned}
$$

- Neighbors Selection: for $(i,j) \in I_R$:

$$
\begin{aligned}
M_r(i,j) &= \{D_j\} \\
M_{mr}(i,j) &= \{D[i:j]\} \\
M_w(i, n-1) &= \{D_i, P_i\} \\
M_{mw}(i, n-1) &= \{P[], D[]\}
\end{aligned}
$$

Low-level implementation details

Only Distance Calculation (step 1) is implemented using OpenCL. Neighbors Selection (step 2) is implemented using a sequential host loop.

Device code Each work item calculates the distance from one point to the base point. See Algorithm 59.

Host code See Algorithm 60.

Host data structures

- **struct** point:

```
struct point
{
  long y;
  long x;
};
```

- **struct** point hP[n]: $P_u \rightarrow$ hP[u]

Algorithm 59 The Nearest Neighbors (distance calculation) device code abstraction

Input: `global struct point dP[]`
Output: `global int dD[]`
Input: `int n`
Input: `int x`
Input: `int y`
1: **kernel** NNKERNEL (dP, dD, n, x, y)
2: **if** $\gamma < n$ **then**
3: $\mathtt{dD}[\gamma] \leftarrow \sqrt{(\mathtt{x} - \mathtt{dP}[\gamma].\mathtt{x})^2 + (\mathtt{y} - \mathtt{dP}[\gamma].\mathtt{y})^2}$
4: **end if**
5: **end kernel**

- **struct** point hR[n]: $R_u \rightarrow$ hR[u]
- **long** hD[n]: $D_u \rightarrow$ hD[u]
- **struct** point hB: $D \rightarrow$ hB

Device data structures

- **struct** point:

```
struct point
{
  long y;
  long x;
};
```

- **struct** point dP[n]: $P_u \rightarrow$ dP[u]
- **struct** point dR[n]: $R_u \rightarrow$ dR[u]
- **long** dD[n]: $D_u \rightarrow$ dD[u]

Input datasets The benchmark uses a set of pre-generated arrays of points. Each set contains 10691 points.

Partitioned iteration space

- Distance Calculation (nnKernel):

$$P_D = \{(\gamma) : 0 \leq \gamma < \Gamma = n\}$$

$$f(i) \rightarrow (i)$$

$$f^{-1}(\gamma) \rightarrow (\gamma)$$

Algorithm 60 The Nearest Neighbors (distance calculation) host code abstraction

```
 1: procedure AllocateMemory(n, k)                    ▷ Allocate host memory and device buffers.
 2:     hP ← AllocateHostMemory(n)
 3:     hD ← AllocateHostMemory(n)
 4:     hR ← AllocateHostMemory(k)
 5:     dP ← AllocateBuffer(n, READ_OLNY)
 6:     dD ← AllocateBuffer(n, WRITE_ONLY)
 7: end procedure

 8: procedure InitializeMemory(file)                  ▷ Initialize host memory.
 9:     hP ← ReadFromFile(file, n)
10: end procedure

11: procedure SortNeighbors(hP, hD, k, n)
12:     for i in 0 : (k − 1) do
13:         min ← i
14:         for j in i : (n − 1) do
15:             if hD[j] < hD[min] then
16:                 min ← j
17:             end if
18:         end for
19:         Swap(hP[min], hP[i])
20:         Swap(hD[min], hD[i])
21:     end for
22: end procedure

Input: int n                                         ▷ Number of points in the input set.
Input: int k                                         ▷ Number of points in the output set.
Input: struct point hB                               ▷ Base point.
Input: const char * file                             ▷ File with pre-generated locations data.
23: AllocateMemory(n, k)
24: InitializeMemory(file)
25: nnProgram ← BuildProgram("nnProgram.cl")
26: nnKernel ← BuildKernel(nnProgram, "nnKernel")
27: CopyToDevice(dP, hP, True, ∅)
28: SetKernelArguments(nnKernel, dP, dD, n, B.x, B.y)
29: EnqueueKernel(nnKernel, (n), none, ∅)
30: CopyToHost(dD, hD, True, ∅)
31: SortNeighbors(hP, hD, k, n)
32: hR ← hP[0 : k − 1]
```

- Neighbors Section (Host code):

$$P_x = \{(i,j) : 0 \leq i < k, i < j < n\}$$

$$f(i,j) \rightarrow (i,j)$$

$$f^{-1}(i,j) \rightarrow (i,j)$$

Device memory access mapping Each work-item performs the following reads and writes:

$$\begin{aligned} G_r(\gamma) &= \{\texttt{dP}[\gamma]\} \\ G_w(\gamma) &= \{\texttt{dD}[\gamma]\} \end{aligned}$$

Host memory access mapping

$$\begin{aligned} H_r(i,j) &= \{\texttt{dD}[\texttt{j}]\} \\ H_{mr}(i,j) &= \{\texttt{dD}[\texttt{i} : \texttt{j}]\} \\ H_w(i,n-1) &= \{\texttt{dD}[\texttt{i}], \texttt{dP}[\texttt{i}]\} \\ H_{mw}(i,n-1) &= \{\texttt{dD}[\,], \texttt{dP}[\,]\} \end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- The total time for the kernel execution (computation time).
- The total memory transfer time from host to device.
- The total memory transfer time from device to host.

The performance is evaluated only for the part of the algorithm implemented in OpenCL (distance calculation).

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can print the transformed matrices and the solution for the original system, which can be validated by external tools (not supplied).

Target-specific optimizations The following target-specific optimization are used in the algorithm implementation:

- **Consecutive memory access.** Work items with consecutive global ids in dimension 0 access consecutive memory locations which improves the memory system performance (caching, coalescing).

Target-specific optimizations opportunities

- **Parallelizing Neighbors Selection.** Neighbors Selection (step 2) can be implemented using OpenCL for selecting a minimal element of D. The algorithm can be modified to avoid swap operations:

```
1:  procedure SORTNEIGHBORS(hP, hD, hR, k, n)
2:      for i in 0 : (k − 1) do
3:          min ← 0
4:          for j in i : (n − 1) do
5:              if hD[j] < hD[min] then
6:                  min ← j
7:              end if
8:          end for
9:          hD[min] ← MAX_DIST
10:         hR[i] ← hP[min]
11:     end for
12: end procedure
```

  The `MAX_DIST` value can be defined as $\sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2}$ + 1, where

  - $x_{max} = 180$ (maximal longitude)
  - $x_{min} = -180$ (minimal longitude)
  - $y_{max} = 90$ (maximal latitude)
  - $y_{min} = -90$ (minimal latitude)

  i.e. the value that is guaranteed to be greater then any value, produced by distance calculation.

2.3.12 Needleman-Wunsch

High-level description

The benchmark uses Needleman-Wunsch algorithm to align two DNA sequences. Having two DNA sequences $A = aabc$ and $B = abbc$ the alignment can be represented as follows:

$$\begin{array}{cccccc} a & a & b & * & c \\ * & a & b & b & c \end{array}$$

The score of the alignment is determined using the similarity matrix S. For the alignment below the score would be:

$$S(a,*) + S(a,a) + S(b,b) + S(*,b) + S(c,c)$$

Abstract data structures The benchmark operates on the following data objects:

- A is N elements input DNA sequences (vector).
- B is N elements input DNA sequences (vector).
- S is an input similarity matrix.
- C is an $(N+1) \times (N+1)$ intermediate score matrix.
- T is an $(N+1) \times (N+1)$ intermediate traceback matrix:

$$T_{i,j} = \{\textbf{up}, \textbf{left}, \textbf{diag}, \textbf{done}\}$$

- P is an input penalty value.
- R is an output alignment ($*$ denotes gap in the sequence):

$$R_i = (a,b) : a \in A \cup \{*\}, b \in B \cup \{*\}$$

Computation The Needleman-Wunsch algorithm uses the dynamic programming technique to determine the alignment with maximal score:

$$\begin{aligned} T_{0,0} &= \textbf{done} \\ T_{0,j} &= \textbf{left}, 0 \leq j \leq N \\ T_{i,0} &= \textbf{up}, 0 \leq i \leq N \\ C_{0,j} &= -j \times P, 1 \leq j \leq N \\ C_{i,0} &= -i \times P, 1 \leq i \leq N \end{aligned}$$

For every $(i,j) : 1 \leq i \leq N, 1 \leq j \leq N$:

$$\begin{aligned} q^d_{i,j} &= C_{i-1,j-1} + S(A_{i-1}, B_{j-1}) \\ q^u_{i,j} &= C_{i-1,j} + P \\ q^l_{i,j} &= C_{i,j-1} + P \\ C_{i,j} &= \max(q^d_{i,j}, q^u_{i,j}, q^l_{i,j}) \end{aligned}$$

$$T_{i,j} = \begin{cases} \textbf{diag} & q^d_{i,j} = \max(q^d_{i,j}, q^u_{i,j}, q^l_{i,j}) \\ \textbf{up} & q^u_{i,j} = \max(q^d_{i,j}, q^u_{i,j}, q^l_{i,j}) \\ \textbf{left} & q^l_{i,j} = \max(q^d_{i,j}, q^u_{i,j}, q^l_{i,j}) \end{cases}$$

The alignment is determined using the traceback matrix:

$i \leftarrow n$
$j \leftarrow n$
while $i \neq 0 \vee j \neq 0$ **do**
  if $T_{i,j} =$ **diag then**
    $i \leftarrow i-1$
    $j \leftarrow j-1$
    $R \leftarrow R + (A_i, B_j)$
  else if $T_{i,j} =$ **left then**
    $j \leftarrow j-1$
    $R \leftarrow R + (*, B_j)$
  else if $T_{i,j} =$ **up then**
    $i \leftarrow i-1$
    $R \leftarrow R + (A_i, *)$
  end if
end while

Iteration spaces and dependences

- Forward computation:

$$I_F = \{(i,j) : 1 \leq i \leq n, 1 \leq j \leq n\}$$

$$I_F : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} n \\ n \\ -1 \\ -1 \end{pmatrix}$$

- Traceback:

$$I_T = \{(\tau) : 0 \leq \tau \leq 2 \times n\}$$

The computation generates the following dependences:

$$\begin{aligned} \delta_{I_F} &= I_F : (i-1, j-1) \xrightarrow{t} I_F : (i,j) \wedge \\ &\quad I_F : (i-1, j) \xrightarrow{t} I_F : (i,j) \wedge \\ &\quad I_F : (i, j-1) \xrightarrow{t} I_F : (i,j) \\ \delta_{I_T} &= I_F : (1:n, 1:n) \xrightarrow{t} I_T : (\tau) \wedge \\ &\quad I_T : (\tau - 1) \xrightarrow{t} I_T : (\tau) \end{aligned}$$

Memory access mapping For $(i,j) \in I_F$:

$$\begin{aligned} M_r(i,j) &= \{C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1}, A_{i-1}, B_{i-1}, S\} \\ M_w(i,j) &= \{C_{i,j}, T_{i,j}\} \end{aligned}$$

For $(\tau) \in I_T$:

$$\begin{aligned} M_r(\tau) &= \{T_{i(\tau),j(\tau)}\} \\ M_w(\tau) &= \{R\} \end{aligned}$$

Low-level implementation details

Only the first part of the algorithm (intermediate score matrix computation) is implemented using OpenCL. The abstract iteration spaces is divided into two concrete iteration spaces, which process the upper-left and the lower-right parts of the score matrix.

Device code See Algorithms 61 and 62.

Host code See Algorithm 63.

Host data structures

- `int hC[(n + 1) * (n+1)]`: $C_{i,j} \rightarrow$ `hC[i * (n+1) + j]`
- `int hT[(n + 1) * (n+1)]`: $T_{i,j} \rightarrow$ `hT[i * (n+1) + j]`
- `int hS[(n + 1) * (n+1)]`: $S_{A_i,B_j} \rightarrow$ `hS[i * (n+1) + j]`

Device data structures `B` is the block size ($B = 16$).

- `global int dC[(n + 1) * (n+1)]`: $C_{i,j} \rightarrow$ `dC[i * (n+1) + j]`
- `global int dS[(n + 1) * (n+1)]`: $S_{A_i,B_j} \rightarrow$ `dS[i * (n+1) + j]`
- `local int dlC[(B + 1) * (B+1)]`: $C_{i,j} \rightarrow$ `dlC[(i % B) * B + j % B]`
  $C_{i,j}|_{i \bmod B=0, j \bmod B=0} \rightarrow$ `dlC[B * B + B]` (Boundary).
- `local int dlS[B * B]`: $S_{A_i,B_j} \rightarrow$ `dlS[(i % B) * B + j % B]`

Input datasets The benchmark uses randomly generated pair of 2048 elements DNA sequence. The size of the sequences can be changed via command line.

Algorithm 61 The Needleman Wunsch (Upper-left) device code abstraction.

Input: `global float dS[]`
Input/Output: `global float dC[]`
Input: `local float dlC[]`
Input: `local float dlS[]`
Input: `int P` ▷ Penalty.
Input: `int Cols` ▷ $n+1$.

1: **kernel** NWKERNEL1 (dS, dC, P, Cols)
2: B $\leftarrow 16$
3: $\texttt{idx}_x \leftarrow \nu$
4: $\texttt{idx}_y \leftarrow N - \nu - 1$
5: $\texttt{idx} \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x + \lambda + \texttt{Cols} + 1$
6: $\texttt{idx}_n \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x + \lambda + 1$
7: $\texttt{idx}_w \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x + \texttt{Cols}$
8: $\texttt{idx}_{nw} \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x$
9: **if** $\lambda = 0$ **then**
10: $\texttt{dlC}[0] \leftarrow \texttt{dC}[\texttt{idx}_{nw}]$
11: **end if**
12: BARRIER(CLK_LOCAL_MEM_FENCE)
13: **for** i **in** $0 : (\texttt{B} - 1)$ **do**
14: $\texttt{dlS}[\texttt{i} \times \texttt{B} + \lambda] \leftarrow \texttt{dS}[\texttt{idx} + \texttt{Cols} \times \texttt{i}]$
15: **end for**
16: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.
17: $\texttt{dlC}[(\lambda + 1) \times (\texttt{B} + 1)] \leftarrow \texttt{dC}[\texttt{idx}_w + \texttt{Cols} \times \lambda]$
18: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.
19: $\texttt{dlC}[\lambda + 1] \leftarrow \texttt{dC}[\texttt{idx}_n]$
20: BARRIER(CLK_LOCAL_MEM_FENCE)
21: **for** m **in** $0 : (\texttt{B} - 1)$ **do**
22: **if** $\lambda \leq \texttt{m}$ **then**
23: $\texttt{tidx}_x \leftarrow \lambda + 1$
24: $\texttt{tidx}_y \leftarrow \texttt{m} - \lambda + 1$
25: $\texttt{q}_{nw} \leftarrow \texttt{dlC}[(\texttt{tidx}_y - 1) \times (\texttt{B} + 1) + \texttt{tidx}_x - 1] + \texttt{dlS}[(\texttt{tidx}_y - 1) \times \texttt{B} + \texttt{tidx}_x - 1]$
26: $\texttt{q}_w \leftarrow \texttt{dlC}[(\texttt{tidx}_y) \times (\texttt{B} + 1) + \texttt{tidx}_x - 1] - P$
27: $\texttt{q}_l \leftarrow \texttt{dlC}[(\texttt{tidx}_y - 1) \times (\texttt{B} + 1) + \texttt{tidx}_x] - P$
28: $\texttt{dlC}[\texttt{tidx}_y \times (\texttt{B} + 1) + \texttt{tidx}_x] \leftarrow \max(q_{nw}, q_w, q_l)$
29: **end if**
30: BARRIER(CLK_LOCAL_MEM_FENCE)
31: **end for**
32: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.

Algorithm 61 The Needleman Wunsch (Upper-left) device code abstraction (continues).

```
33:     for m in (B − 2) : 0 : (−1) do
34:         if λ ≤ m then
35:             tidx_x ← λ + B − m
36:             tidx_y ← B − λ
37:             q_nw ← dlC[(tidx_y − 1) × (B + 1) + tidx_x − 1] + dlS[(tidx_y − 1) × B + tidx_x − 1]
38:             q_w ← dlC[(tidx_y) × (B + 1) + tidx_x − 1] − P
39:             q_l ← dlC[(tidx_y − 1) × (B + 1) + tidx_x] − P
40:             dlC[tidx_y × (B + 1) + tidx_x] ← max(q_nw, q_w, q_l)
41:         end if
42:         BARRIER(CLK_LOCAL_MEM_FENCE)
43:     end for
44:     for i in 0 : (B − 1) do
45:         dC[idx + Cols × i] ← dlC[(i + 1) × (B + 1) + λ + 1]
46:     end for
47: end kernel
```

Partitioned iteration space

- Upper-left part (nwKernel1), upper-left part of the block:

$$P_U = \{(b,\nu,\lambda,m) : 0 \le b \le n/B, 0 \le \nu < \Gamma = b, 0 \le \lambda < \Lambda = B, \lambda \le m < B\}$$

$$f(i,j) \to ((i-1)/B + (j-1)/B + 1, (j-1)/B, (j-1) \bmod (B), (i+j-2) \bmod (B))$$

$$f^{-1}(b,\nu,\lambda,m) \to ((b-\nu-1) \times B + m - \lambda + 1, B \times \nu + \lambda + 1)$$

- Upper-left part (nwKernel1), lower-right part of the block:

$$P_U = \{(b,\nu,\lambda,m) 0 \le b \le n/B, 0 \le \nu < \Gamma = b, 0 \le \lambda < \Lambda = B, \lambda \le m < B-1\}$$

$$f(i,j) \to ((i-1)/B + (j-1)/B + 1, (j-1)/B, i \bmod (B) \times -1, (i+j) \bmod (B) \times -1)$$

$$f^{-1}(b,\nu,\lambda,m) \to ((b-\nu-1) \times B + B - \lambda, B \times \nu + \lambda + B - m)$$

- Lower-right part (nwKernel2), upper-left part of the block:

$$P_U = \{(b,\nu,\lambda,m) 1 \le b < n/B, 0 \le \nu < \Gamma = b, 0 \le \lambda < \Lambda = B, \lambda \le m < B\}$$

$$\begin{aligned} f(i,j) \quad \to \quad & (2 \times n/B - ((i-1)/B + (j-1)/B), n/B - 1 - (i-1)/B, \\ & (j-1) \bmod (B), (i+j-2) \bmod (B)) \end{aligned}$$

Algorithm 62 The Needleman Wunsch (Lower-right) device code abstraction.

Input: `global float dS[]`
Input/Output: `global float dC[]`
Input: `local float dlC[]`
Input: `local float dlS[]`
Input: `int P` ▷ Penalty.
Input: `int Cols` ▷ $n+1$.
Input: `int Bw` ▷ n/B

1: **kernel** NWKERNEL2 (dS, dC, P, Cols, Bw)
2: B $\leftarrow 16$
3: $\texttt{idx}_x \leftarrow \nu + \texttt{Bw} - N$
4: $\texttt{idx}_y \leftarrow \texttt{Bw} - \nu - 1$
5: $\texttt{idx} \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x + \lambda + \texttt{Cols} + 1$
6: $\texttt{idx}_n \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x + \lambda + 1$
7: $\texttt{idx}_w \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x + \texttt{Cols}$
8: $\texttt{idx}_{nw} \leftarrow \texttt{Cols} \times \texttt{B} \times \texttt{idx}_y + \texttt{B} \times \texttt{idx}_x$
9: **if** $\lambda = 0$ **then**
10: $\texttt{dlC}[0] \leftarrow \texttt{dC}[\texttt{idx}_{nw}]$
11: **end if**
12: BARRIER(CLK_LOCAL_MEM_FENCE)
13: **for** i **in** $0 : (\texttt{B} - 1)$ **do**
14: $\texttt{dlS}[\texttt{i} \times \texttt{B} + \lambda] \leftarrow \texttt{dS}[\texttt{idx} + \texttt{Cols} \times \texttt{i}]$
15: **end for**
16: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.
17: $\texttt{dlC}[(\lambda + 1) \times (\texttt{B} + 1)] \leftarrow \texttt{dC}[\texttt{idx}_w + \texttt{Cols} \times \lambda]$
18: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.
19: $\texttt{dlC}[\lambda + 1] \leftarrow \texttt{dC}[\texttt{idx}_n]$
20: BARRIER(CLK_LOCAL_MEM_FENCE)
21: **for** m **in** $0 : (\texttt{B} - 1)$ **do**
22: **if** $\lambda \leq \texttt{m}$ **then**
23: $\texttt{tidx}_x \leftarrow \lambda + 1$
24: $\texttt{tidx}_y \leftarrow \texttt{m} - \lambda + 1$
25: $\texttt{q}_{nw} \leftarrow \texttt{dlC}[(\texttt{tidx}_y - 1) \times (\texttt{B} + 1) + \texttt{tidx}_x - 1] + \texttt{dlS}[(\texttt{tidx}_y - 1) \times \texttt{B} + \texttt{tidx}_x - 1]$
26: $\texttt{q}_w \leftarrow \texttt{dlC}[(\texttt{tidx}_y) \times (\texttt{B} + 1) + \texttt{tidx}_x - 1] - P$
27: $\texttt{q}_l \leftarrow \texttt{dlC}[(\texttt{tidx}_y - 1) \times (\texttt{B} + 1) + \texttt{tidx}_x] - P$
28: $\texttt{dlC}[\texttt{tidx}_y \times (\texttt{B} + 1) + \texttt{tidx}_x] \leftarrow \max(q_{nw}, q_w, q_l)$
29: **end if**
30: BARRIER(CLK_LOCAL_MEM_FENCE)
31: **end for**
32: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ Unnecessary memory barrier.

Algorithm 62 The Needleman Wunsch (Lower-right) device code abstraction (continues).

```
33:     for m in (B−2) : 0 : (−1) do
34:         if λ ≤ m then
35:             tidx_x ← λ + B − m
36:             tidx_y ← B − λ
37:             q_nw ← dlC[(tidx_y − 1) × (B+1) + tidx_x − 1] + dlS[(tidx_y − 1) × B + tidx_x − 1]
38:             q_w ← dlC[(tidx_y) × (B+1) + tidx_x − 1] − P
39:             q_l ← dlC[(tidx_y − 1) × (B+1) + tidx_x] − P
40:             dlC[tidx_y × (B+1) + tidx_x] ← max(q_nw, q_w, q_l)
41:         end if
42:         BARRIER(CLK_LOCAL_MEM_FENCE)
43:     end for
44:     for i in 0 : (B − 1) do
45:         dC[idx + Cols × i] ← dlC[(i+1) × (B+1) + λ + 1]
46:     end for
47: end kernel
```

$$f^{-1}(b,\nu,\lambda,m) \rightarrow ((n/B-\nu-1)\times B+m-\lambda+1, B\times(\nu+n/B-b)+\lambda+1)$$

- Lower-right part (nwKernel2), lower-right part of the block:

$$P_U = \{(b,\nu,\lambda,m) 1 \leq b < n/B, 0 \leq \nu < \Gamma = b, 0 \leq \lambda < \Lambda = B, \lambda \leq m < B-1\}$$

$$f(i,j) \rightarrow (2\times n/B-((i-1)/B+(j-1)/B), n/B-1-(i-1)/B, , i \bmod (B)\times -1, (i+j) \bmod (B)\times -1)$$

$$f^{-1}(b,\nu,\lambda,m) \rightarrow ((n/B-\nu-1)\times B+B-\lambda, B\times(\nu+n/B-b)+\lambda+B-m)$$

Device memory access mapping

- Upper-left part (nwKernel1):
  - Global to local memory transfer:

$$\begin{aligned}
G_r(b,\nu,\lambda,i) &= \{\texttt{dS}[((b-\nu-1)\times B\times(n+1)+\nu\times B+\lambda+n+2+i\times(n+1)]\} \\
G_r(b,\nu,0,0) &= \{\texttt{dC}[(b-\nu-1)\times B\times(n+1)+\nu\times B]\} \\
G_r(b,\nu,\lambda,0) &= \{\texttt{dC}[(b-\nu-1)\times B\times(n+1)+\nu\times B+(n+1)\times(\lambda+1)]\} \\
G_r(b,\nu,\lambda,0) &= \{\texttt{dC}[(b-\nu-1)\times B\times(n+1)+\nu\times B+\lambda+1]\} \\
L_w(b,\nu,\lambda,i) &= \{\texttt{dlS}[\texttt{i}\times B+\lambda]\} \\
L_w(b,\nu,0,0) &= \{\texttt{dlC}[0]\} \\
L_w(b,\nu,\lambda,0) &= \{\texttt{dlS}[\lambda+1], \texttt{dlS}[(\lambda+1)\times B]\}
\end{aligned}$$

Algorithm 63 The Needleman Wunsch host code abstraction.

```
1:  procedure TRACEBACK(n)
2:      i ← n
3:      j ← n
4:      while i ≠ 0, j ≠ 0 do                ▷ Bug. Logical 'OR' should be used instead.
5:          if i = 0 then
6:              nw = limit
7:              n = limit
8:              w = hC[j − 1]
9:          else if j = 0 then
10:             nw = limit
11:             n = hC[(i − 1) × (n + 1)]
12:             w = limit
13:         else
14:             nw = hC[(i − 1) × (n + 1) + j − 1]
15:             n = hC[(i − 1) × (n + 1) + j]
16:             w = hC[i × (n + 1) + j − 1]
17:         end if
18:         nw_new ← nw + hS[i × (n + 1) + j]
19:         w_new ← w − P
20:         n_new ← n − P
21:         trace = max(nw_new, n_new, w_new)
22:         if trace = nw_new then
23:             i ← i − 1
24:             j ← j − 1
25:         else if trace = w_new then
26:             j ← j − 1
27:         else
28:             i ← i − 1
29:         end if
30:     end while
31: end procedure
```

Algorithm 63 The Needleman Wunsch host code abstraction (continues).

```
32: procedure AllocateMemory(n)                    ▷ Allocate host memory and device buffers.
33:     hC ← AllocateHostMemory((n+1) × (n+1))
34:     hS ← AllocateHostMemory(n × n)
35:     dC ← AllocateBuffer((n+1) × (n+1), READ_WRITE)
36:     dS ← AllocateBuffer(n × n, READ_WRITE)
37: end procedure

38: procedure InitializeMemory(n, P)               ▷ Initialize host memory.
39:     for i in 1 : n do
40:         hC[i] ← i × P
41:         hC[i × (n+1)] ← i × P
42:     end for
43:     A ← RandomElements(n)                      ▷ Temporary vector.
44:     B ← RandomElements(n)                      ▷ Temporary vector.
45:     for i in 0 : (n−1) do
46:         for j in 0 : (n−1) do
47:             hS[i × n + j] ← S[A[i]][B[j]]
48:         end for
49:     end for
50: end procedure

Input: int n                                       ▷ Input sequence size.
Input: int P                                       ▷ Penalty.
Input: int S[24][24]                               ▷ Similarity matrix.
51: AllocateMemory(n)
52: InitializeMemory(n, P)
53: nwProgram ← BuildProgram("ludProgram.cl")
54: nwKernel1 ← BuildKernel(nwProgram, "nwKernel1")
55: nwKernel2 ← BuildKernel(nwProgram, "nwKernel2")
56: B ← 16                                         ▷ Block size.
57: CopyToDevice(dS, hS, True, ∅)
58: CopyToDevice(dC, hC, True, ∅)
59: for blk in 1 : (n/B) do
60:     SetKernelArguments(nwKernel1, dS, dC, P, n+1)
61:     EnqueueKernel(nwKernel1, (B × blk), (B), ∅)
62: end for
63: for blk in (n/B − 1) : 1 : (−1) do
64:     SetKernelArguments(nwKernel2, dS, dC, P, n+1, B/B)
65:     EnqueueKernel(nwKernel2, (B × blk), (B), ∅)
66: end for
67: CopyToHost(dC, hC, True, ∅)
68: Traceback(n)
```

- Upper-left part of the block:

$$\begin{aligned} L_r(b,v,\lambda,m) &= \{\texttt{dlC}[(m-\lambda)\times(B+1)+\lambda], \texttt{dlC}[(m-\lambda+1)\times(B+1)+\lambda], \\ &\quad \texttt{dlC}[(m-\lambda)\times(B+1)+\lambda+1], \texttt{dlS}[(m-\lambda)\times B+\lambda]\} \\ L_w(b,v,\lambda,m) &= \{\texttt{dlC}[(m-\lambda+1)\times(B+1)+\lambda+1],\} \end{aligned}$$

- Lower-right part of the block:

$$\begin{aligned} L_r(b,v,\lambda,m) &= \{\texttt{dlC}[(B-\lambda-1)\times(B+1)+\lambda+B-m-1], \\ &\quad \texttt{dlC}[(B-\lambda)\times(B+1)+\lambda+B-m-1], \\ &\quad \texttt{dlC}[(B-\lambda-1)\times(B+1)+\lambda+B-m], \\ &\quad \texttt{dlS}[(B-\lambda-1)\times B+\lambda+B-m-1],\} \\ L_w(b,v,\lambda,m) &= \{\texttt{dlC}[(B-\lambda)\times(B+1)+\lambda+B-m],\} \end{aligned}$$

- Local to global memory transfer:

$$\begin{aligned} G_w(b,v,\lambda) &= \{\texttt{dC}[((b-v-1)\times B\times(n+1)+v\times B+\lambda+n+2+i\times(n+1)]\} \\ L_r(b,v,\lambda) &= \{\texttt{dlC}[(\texttt{i}+1)\times(B+1)+\lambda+1]\} \end{aligned}$$

- Lower-right part (nwKernel2):

- Global to local memory transfer:

$$\begin{aligned} G_r(b,v,\lambda,i) &= \{\texttt{dS}[((n/B-v-1)\times B\times(n+1)+(v+n/B-b)\times B+\lambda+n+2+i\times(n+1)]\} \\ G_r(b,v,0,0) &= \{\texttt{dC}[(n/B-v-1)\times B\times(n+1)+(v+n/B-b)\times B]\} \\ G_r(b,v,\lambda,0) &= \{\texttt{dC}[(n/B-v-1)\times B\times(n+1)+(v+n/B-b)\times B+(n+1)\times(\lambda+1)]\} \\ G_r(b,v,\lambda,0) &= \{\texttt{dC}[(n/B-v-1)\times B\times(n+1)+(v+n/B-b)\times B+\lambda+1]\} \\ L_w(b,v,\lambda,i) &= \{\texttt{dlS}[\texttt{i}\times B+\lambda]\} \\ L_w(b,v,0,0) &= \{\texttt{dlC}[0]\} \\ L_w(b,v,\lambda,0) &= \{\texttt{dlS}[\lambda+1], \texttt{dlS}[(\lambda+1)\times B]\} \end{aligned}$$

- Upper-left part of the block:

$$\begin{aligned} L_r(b,v,\lambda,m) &= \{\texttt{dlC}[(m-\lambda)\times(B+1)+\lambda], \texttt{dlC}[(m-\lambda+1)\times(B+1)+\lambda], \\ &\quad \texttt{dlC}[(m-\lambda)\times(B+1)+\lambda+1], \texttt{dlS}[(m-\lambda)\times B+\lambda]\} \\ L_w(b,v,\lambda,m) &= \{\texttt{dlC}[(m-\lambda+1)\times(B+1)+\lambda+1],\} \end{aligned}$$

– Lower-right part of the block:

$$\begin{aligned} L_r(b,v,\lambda,m) &= \{\texttt{dlC}[(B-\lambda-1)\times(B+1)+\lambda+B-m-1], \\ &\quad \texttt{dlC}[(B-\lambda)\times(B+1)+\lambda+B-m-1], \\ &\quad \texttt{dlC}[(B-\lambda-1)\times(B+1)+\lambda+B-m], \\ &\quad \texttt{dlS}[(B-\lambda-1)\times B+\lambda+B-m-1],\} \\ L_w(b,v,\lambda,m) &= \{\texttt{dlC}[(B-\lambda)\times(B+1)+\lambda+B-m]\} \end{aligned}$$

– Local to global memory transfer:

$$\begin{aligned} G_w(b,v,\lambda) &= \{\texttt{dC}[((n/B-v-1)\times B\times(n+1)+(v+n/B-b)\times B+\lambda+n+2+i\times(n+1)]\} \\ L_r(b,v,\lambda) &= \{\texttt{dlC}[(\texttt{i}+1)\times(B+1)+\lambda+1]\} \end{aligned}$$

Host memory access mapping On each iteration on the host the following memory accesses are performed:

$$\begin{aligned} H_r(\tau) = H_r(\texttt{i}(\tau),\texttt{j}(\tau)) &= \{\texttt{hC}[(\texttt{i}-1)\times(\texttt{n}+1)+\texttt{j}-1],\texttt{hC}[(\texttt{i}-1)\times(\texttt{n}+1)+\texttt{j}], \\ &\quad \texttt{hC}[\texttt{i}\times(\texttt{n}+1)+\texttt{j}-1],\texttt{hS}[\texttt{i}\times(\texttt{n}+1)+\texttt{j}]\} \end{aligned}$$

Performance metrics The performance is evaluated using the following metrics:

- The total time for all nwKernel1 launches.
- The total time for all nwKernel2 launches.
- The total time for all kernel launches (computation time).
- The total queuing time.

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can print the traceback values which can be validated by external tools (not supplied).

Target-specific optimizations The following target-specific optimization are used in the algorithm implementation:

- **Consecutive memory access.** Work items with consecutive global ids in dimension 1 access consecutive memory locations which improves the memory system performance (caching, coalescing).
- **Local memory usage.** In order to decrease the number of global memory accesses each work group allocates a local buffer, which is used to perform all computations. This technique also increases data locality since the data layout can be changed while copying from global to local memory and vise versa.

Target-specific optimizations opportunities

- **Global memory usage.** In some OpenCL implementations local and global memories resident in the same memory spaces, so copying data from global to local memory just introduces additional overhead.
- **Avoiding unnecessary synchronizations.** Not all memory barriers used in the algorithm implementation are necessary (see device code abstraction). Removing such barriers can decrease kernel execution time.

2.3.13 Particle Filter

High-level description

The benchmark uses particle filter algorithm to track object movement in the video sequence between consecutive frames.

Abstract data structures The benchmark operates on the following data objects:

- V is an N_f-frame input video sequence.
- $F = V_i$ is a $N_x \times N_y$ video frame.
- N_p is an input number of particles to be used for approximation.
- R is an input radius of the particle.
- $P[N_p]$ is an intermediate array of particles coordinates within the video frame.
- P_i^x is the x coordinate of the i^{th} particle.
- $L[N_p]$ is an intermediate array of particles likelihoods.
- $S[N_p]$ is an intermediate array of point sets.
- $W[N_p]$ is an intermediate array of particles weights.
- $C[N_p], U[N_p]$ are intermediate arrays.
- $E = 100$ is an empty point value.
- $O = 228$ is an object point value.

Computation The initial data is initialized as follows:

$$P_i \quad = \quad (N_x/2, (N_y/2)), 0 \leq i < N_p$$

For each frame $F = V_k, 0 \leq k < N_f$ the following computations are performed:

1. Particles locations recalculation:

$$P_i = (P_i^x + 1 + 5 \times \textbf{rand}(), P_i^y - 2 + 2 \times \textbf{rand}()), 0 \leq i < N_p$$

2. Points sets generation:

$$S_i = \{(x,y) : (x - P_i^x)^2 + (y - P_i^y)^2 \leq R^2\}$$

3. Likelihood calculation:

$$L_i = \left(\sum_{(x,y):\in S_i} ((I_{x,y} - E)^2 - (I_{x,y} - O)^2)/50 \right) / |S_i|$$

4. Weights calculation:

$$W_i = \exp(L_i)/N_p, 0 \leq i < N_p$$

5. Weights normalization:

$$\begin{aligned} S_w &= \sum_{0 \leq i < N_p} W_i \\ W_i &= W_i/S_w, 0 \leq i < N_p \end{aligned}$$

6. Intermediate arrays (C, U) calculation:

$$\begin{aligned} U_{\texttt{tmp}} &= \textbf{rand}() \\ U_i &= i/N_p + U_{\texttt{tmp}}, 0 \leq i < N_p \\ C_i &= \sum_{0 \leq j \leq i} W_j, 0 \leq i < N_p \end{aligned}$$

7. Particles locations update:

$$\begin{aligned} P' &= P \\ P^x_i &= P'^x_j : C_j > U_i \wedge C_{j-1} \leq U_i, 0 \leq i < N_p \\ P^y_i &= P'^y_j : C_j > U_i \wedge C_{j-1} \leq U_i, 0 \leq i < N_p \end{aligned}$$

The **rand**() function call generates the normally distributed random numbers.

Iteration spaces and dependences This algorithm requires several iteration spaces:

1. Particles locations recalculation:

$$I_P = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

2. Points sets generation:

$$I_S = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

3. Likelihood calculation:

$$I_L = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

4. Weights calculation:

$$I_W = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

5. Weights sum calculation:

$$I_{S_w} = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

6. Weights normalization:

$$I_N = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

7. Intermediate array U calculation:

$$I_U = \{(k,i) : 0 \leq k < N_f, 0 \leq i < N_p\}$$

8. Intermediate array C calculation:

$$I_C = \{(k,i,j) : 0 \leq k < N_f, 0 \leq i < N_p, 0 \leq j \leq i\}$$

9. Particles locations update:

$$I_F = \{(k,i,j) : 0 \leq k < N_f, 0 \leq i < N_p, 0 \leq j < N_p\}$$

The computation generates the following dependences:

$$\begin{aligned}
\delta_{I_P} &= I_F : (k-1,i,0 : (N_p-1)) \xrightarrow{t} I_P : (k,i) \wedge \\
&\quad I_S(k-1,i) \xrightarrow{a} I_P : (k,i) \\
\delta_{I_S} &= I_P : (k,i) \xrightarrow{t} I_S : (k,i) \wedge \\
&\quad I_L(k-1,i) \xrightarrow{a} I_S : (k,i) \\
\delta_{I_L} &= I_S : (k,i) \xrightarrow{t} I_L : (k,i) \wedge \\
&\quad I_W(k-1,i) \xrightarrow{a} I_L : (k,i) \\
\delta_{I_W} &= I_L : (k,i) \xrightarrow{t} I_W : (k,i) \wedge \\
&\quad I_N(k-1,i) \xrightarrow{a} I_W : (k,i) \\
\delta_{I_{S_w}} &= I_W : (k,i) \xrightarrow{t} I_{S_w} : (k,i) \wedge \\
&\quad I_N(k-1,0 : (N_p-1)) \xrightarrow{a} I_{S_w} : (k,i) \\
\delta_{I_N} &= I_W : (k,i) \xrightarrow{t} I_N : (k,i) \wedge \\
&\quad I_{S_w} : (k,0 : (N_p-1)) \xrightarrow{t} I_N : (k,i) \wedge \\
&\quad I_C(k-1,i : (N_p-1),i) \xrightarrow{a} I_N : (k,i) \\
\delta_{I_U} &= I_F : (k-1,i,0 : (N_p-1)) \xrightarrow{a} I_U : (k,i) \wedge \\
\delta_{I_C} &= I_N : (k,j) \xrightarrow{t} I_C : (k,i,j) \wedge \\
&\quad I_F(k-1,0 : (N_p-1),i) \xrightarrow{a} I_C : (k,i,j) \wedge \\
\delta_{I_F} &= I_U : (k,i) \xrightarrow{t} I_F : (k,i,j) \wedge \\
&\quad I_C : (k,j,0 : j) \xrightarrow{t} I_F(k,i,j) \wedge \\
&\quad I_P : (k,i) \xrightarrow{a} I_F(k,i,0 : (N_p-1))
\end{aligned}$$

Memory access mapping

- Particles locations recalculation:

$$\begin{aligned}
M_r(k,i) &= \{P_i\} \\
M_w(k,i) &= \{P_i\}
\end{aligned}$$

- Points sets generation:

$$\begin{aligned} M_r(k,i) &= \{P_i\} \\ M_w(k,i) &= \{S_i\} \end{aligned}$$

- Likelihood calculation:

$$\begin{aligned} M_r(k,i) &= \{S_i\} \\ M_w(k,i) &= \{L_i\} \end{aligned}$$

- Weights calculation:

$$\begin{aligned} M_r(k,i) &= \{L_i\} \\ M_w(k,i) &= \{W_i\} \end{aligned}$$

- Weights sum calculation:

$$M_r(k,i) = \{W_i\}$$

- Weights normalization:

$$\begin{aligned} M_r(k,i) &= \{W_{0:(0-N_p-1)}\} \\ M_w(k,i) &= \{W_i\} \end{aligned}$$

- Intermediate array U calculation:

$$M_w(k,i) = \{U_i\}$$

- Intermediate array C calculation:

$$\begin{aligned} M_r(k,i,j) &= \{W_j\} \\ M_w(k,i,j) &= \{C_i\} \end{aligned}$$

- Particles locations update:

$$\begin{aligned} M_r(k,i,j) &= \{C_j, U_i\} \\ M_w(k,i,j) &= \{P_i\} \end{aligned}$$

Low-level implementation details

The detailed description of the algorithm implementation can be found in [10]. The optimized implementation is described above, *i.e.* all computations are performed on device.

Device code See Algorithm 64,Algorithm 65, Algorithm 66 and Algorithm 67.

Host code See Algorithm 68.

Algorithm 64 The Particle Filter (Likelihood recalculation) device code abstraction.

1: **function** CALCLIKELIHOODSUM(dV, dS, SSize, γ)
2: sum $\leftarrow 0$
3: **for** x **in** $0 : (\texttt{SSize} - 1)$ **do**
4: sum $\leftarrow$ sum $+ ((\texttt{dV}[\texttt{dS}[\texttt{SSize} \times \gamma + \texttt{x}]] - 100)^2 - (\texttt{dV}[\texttt{dS}[\texttt{SSize} \times \gamma + \texttt{x}]] - 228)^2)/50$
5: **end for**
6: **return** sum
7: **end function**

Input: `int Np` ▷ Number of particles.
Input: `int Ny`
Input: `int Nf`
Input: `int SSize` ▷ Size of the S_i.
Input: `int VSize` ▷ $N_f \times N_x \times N_y$
Input: `int k`
Input: `global int dS[]`
Input: `global int dV[]`
Input: `global int dSbase[]`
Output: `global float dX[]`
Output: `global float dY[]`
Input: `global float dX2[]`
Input: `global float dY2[]`
Input/Output: `global float dW[]`
Output: `global int dL[]`
Output: `global int dSums[]`
Local: `local int dlB[]`
8: **kernel** PFKERNEL1 (Np, Ny, Nf, SSize, VSize, k, dS, dV, dSbase, dX, dY, dW, dL, dSums, dX2, dY2)
9: **if** $\gamma <$ Np **then**
10: dX$[\gamma] \leftarrow$ dX2$[\gamma]$
11: dY$[\gamma] \leftarrow$ dY2$[\gamma]$
12: dX$[\gamma] \leftarrow$ dX$[\gamma] + 1 + 5 \times$ RAND( )
13: dY$[\gamma] \leftarrow$ dY$[\gamma] - 2 + 2 \times$ RAND( )
14: **end if**
15: BARRIER(CLK_GLOBAL_MEM_FENCE)
16: **if** $\gamma <$ Np **then**
17: **for** y **in** $0 : (\texttt{SSize} - 1)$ **do**
18: idxX $\leftarrow$ dX$[\gamma]$ + dSbase$[2 \times \texttt{y} + 1]$
19: idxY $\leftarrow$ dY$[\gamma]$ + dSbase$[2 \times \texttt{y}]$
20: dS$[\gamma \times$ SSize + y$] \leftarrow$ idxX $\times$ Ny $\times$ Nf + IdxY $\times$ Nf + k
21: **if** dS$[\gamma \times$ SSize + y$] \geq$ VSize **then**
22: dS$[\gamma \times$ SSize + y$] \leftarrow 0$
23: **end if**
24: **end for**
25: dL$[\gamma] \leftarrow$ CALCLIKELIHOODSUM(dV, dS, SSize, γ)
26: dL$[\gamma] \leftarrow$ dL$[\gamma]$/SSize
27: dW$[\gamma] \leftarrow$ dW$[\gamma] \times \exp($dL$[\gamma])$
28: **end if**

Algorithm 64 The Particle Filter (Likelihood recalculation) device code abstraction (continues).

```
29:     BARRIER(CLK_GLOBAL_MEM_FENCE|CLK_LOCAL_MEM_FENCE)
30:     if γ < Np then
31:         dlB[λ] ← dW[γ]
32:     end if
33:     BARRIER(CLK_LOCAL_MEM_FENCE)
34:     s ← Λ/2
35:     while s > 0 do
36:         if λ < s then
37:             dlB[λ] ← dlB[λ] + dlB[λ + s]
38:         end if
39:         BARRIER(CLK_GLOBAL_MEM_FENCE)
40:         if λ = 0 then
41:             dSums[v] ← dlB[0]
42:         end if
43:         s ← s/2
44:     end while
45: end kernel
```

Algorithm 65 The Particle Filter (Sum) device code abstraction.

Input: `int Np` ▷ Number of particles.
Input/Output: `global float dSums[]`

```
1: kernel PFKERNEL2 (Np, dSums)
2:     if γ = 0 then
3:         sum ← 0
4:         for x in 0 : (Np/512 − 1) do
5:             sum ← sum + dSums[x]
6:         end for
7:         dSums[0] ← sum
8:     end if
9: end kernel
```

Algorithm 66 The Particle Filter (Normalize Weights) device code abstraction.

```
1: procedure CDFCALC(dC, dW, Np)
2:     dC[0] ← dW[0]
3:     for x in 0 : (Np − 1) do
4:         dC[x] ← dW[x] + dC[x − 1]
5:     end for
6: end procedure
Input: int Np                                              ▷ Number of particles.
Input: global float dSums[]
Output: global float dC[]
Output: global float dU[]
Input/Output: global float dW[]
7: kernel PFKERNEL3 (Np, dW, dSums, dC, dU)
8:     if λ = 0 then
9:         sum ← dSums[0]
10:    end if
11:    BARRIER(CLK_LOCAL_MEM_FENCE)
12:    if γ < Np then
13:        dW[i] ← dW[i]/sum
14:    end if
15:    BARRIER(CLK_GLOBAL_MEM_FENCE)
16:    if γ = 0 then
17:        CDFCALC(dC, dW, Np)
18:        dU[0] ← RAND( ) / Np
19:    end if
20:    BARRIER(CLK_GLOBAL_MEM_FENCE)
21:    if λ = 0 then
22:        u1 ← dU[0]
23:    end if
24:    BARRIER(CLK_LOCAL_MEM_FENCE)
25:    if γ < Np then
26:        dU[i] ← u1 + i/Np
27:    end if
28: end kernel
```

Algorithm 67 The Particle Filter (Find Index) device code abstraction.

Input: `global float dU[]`
Input: `global float dX[]`
Input: `global float dY[]`
Input: `global float dC[]`
Output: `global float dX2[]`
Output: `global float dY2[]`
Output: `global float dW[]`
Input: `int Np` ▷ Number of particles.

1: **kernel** PFKERNEL4 (Np, dU, dC, dX, dY, dX2, dY2, dW)
2: **if** $\gamma < \texttt{Np}$ **then**
3: idx $\leftarrow -1$
4: **for** x **in** $0 : (\texttt{Np} - 1)$ **do**
5: **if** $\texttt{dC}[\texttt{x}] \leq \texttt{u}[\gamma]$ **then**
6: idx $\leftarrow$ x
7: **break**
8: **end if**
9: **end for**
10: **if** $\texttt{idx} = -1$ **then**
11: idx $\leftarrow \texttt{Np} - 1$
12: **end if**
13: $\texttt{dX2}[\gamma] \leftarrow \texttt{dX}[\texttt{idx}]$
14: $\texttt{dY2}[\gamma] \leftarrow \texttt{dY}[\texttt{idx}]$
15: $\texttt{dW}[\gamma] \leftarrow 1/\texttt{Np}$
16: **end if**
17: BARRIER(CLK_GLOBAL_MEM_FENCE)
18: **end kernel**

Algorithm 68 The Particle Filter host code abstraction.

```
1: procedure AllocateMemory(Nx, Ny, Nf, Np, SSize)        ▷ Allocate host memory and device buffers.
2:     hV ← AllocateHostMemory(Nx × Ny × Nf)
3:     hX ← AllocateHostMemory(Np)
4:     hY ← AllocateHostMemory(Np)
5:     hW ← AllocateHostMemory(Np)
6:     hSbase ← AllocateHostMemory(SSize × 2)
7:     dX ← AllocateBuffer(Np, READ_WRITE)
8:     dY ← AllocateBuffer(Np, READ_WRITE)
9:     dX2 ← AllocateBuffer(Np, READ_WRITE)
10:    dY2 ← AllocateBuffer(Np, READ_WRITE)
11:    dC ← AllocateBuffer(Np, READ_WRITE)
12:    dU ← AllocateBuffer(Np, READ_WRITE)
13:    dL ← AllocateBuffer(Np, READ_WRITE)
14:    dW ← AllocateBuffer(Np, READ_WRITE)
15:    dV ← AllocateBuffer(Nx × Ny × Nf, READ_WRITE)
16:    dSbase ← AllocateBuffer(SSize × 2, READ_WRITE)
17:    dS ← AllocateBuffer(SSize × Np, READ_WRITE)
18:    dSums ← AllocateBuffer(Np, READ_WRITE)
19: end procedure

20: procedure InitializeMemory(Nx, Ny, Nf, Np, SSize, R)        ▷ Initialize host memory.
21:     for i in 0 : (Np − 1) do
22:         hX[i] ← Nx/2
23:         hY[i] ← Ny/2
24:         hW[i] ← 1/Np
25:     end for
26:     hV ← RandomElements(Nx × Ny × Nf)
27:     idx ← 0
28:     for x in 0 : (2 × R − 2) do
29:         for y in 0 : (2 × R − 2) do
30:             if (x − R + 1)^2 + (y − (R + 1)^2 < R^2 then
31:                 hSbase[idx] ← y
32:                 hSbase[idx + 1] ← x
33:                 idx ← idx + 2
34:             end if
35:         end for
36:     end for
37: end procedure
```

Algorithm 68 The Particle Filter host code abstraction (continues).

Input: `int Nx` ▷ Input video sequence width.
Input: `int Ny` ▷ Input video sequence height.
Input: `int Nf` ▷ Input video sequence number of frames.
Input: `int Np` ▷ Number of particles to use for approximation.
Input: `int R` ▷ Target object radius.

38: `SSize` ← GETCIRCLEAREA(`R`)
39: `VSize` ← `Nx` × `Ny` × `Nf`
40: ALLOCATEMEMORY(`Nx`, `Ny`, `Nf`, `Np`, `SSize`)
41: INITIALIZEMEMORY(`Nx`, `Ny`, `Nf`, `Np`, `SSize`, `R`)
42: `pfProgram` ← BUILDPROGRAM("`pfProgram.cl`")
43: `pfKernel1` ← BUILDKERNEL(`pfProgram`, "`pfKernel1`")
44: `pfKernel2` ← BUILDKERNEL(`pfProgram`, "`pfKernel2`")
45: `pfKernel3` ← BUILDKERNEL(`pfProgram`, "`pfKernel3`")
46: `pfKernel4` ← BUILDKERNEL(`pfProgram`, "`pfKernel4`")
47: `WGsize` ← 512 ▷ Work group size.
48: `NWG` ← (`Np` + `WGsize` − 1)/`WGsize` ▷ Work groups number.
49: COPYTODEVICE(`dV`, `hV`, **True**, ∅)
50: COPYTODEVICE(`dW`, `hW`, **True**, ∅)
51: COPYTODEVICE(`dSbase`, `hSbase`, **True**, ∅)
52: COPYTODEVICE(`dX2`, `hX`, **True**, ∅)
53: COPYTODEVICE(`dY2`, `hY`, **True**, ∅)
54: `width` ← `IMG_MAX_WIDTH`
55: `height` ← `Np`/(4 × `IMG_MAX_WIDTH`)
56: **for** `k` **in** 1 : (`Nf` − 1) **do**
57: SETKERNELARGUMENTS(`pfKernel1`, `Np`, `Ny`, `Nf`, `SSize`, `VSize`, `k`, `dS`, `dV`, `dSbase`, `dX`, `dY`, `dX2`, `dY2`, `dW`, `dL`, `dSums`)
58: ENQUEUEKERNEL(`pfKernel1`, (`WGsize` × `NWG`), (`WGsize`), ∅)
59: SETKERNELARGUMENTS(`pfKernel2`, `Np`, `dSums`)
60: ENQUEUEKERNEL(`pfKernel2`, (`WGsize` × `NWG`), (`WGsize`), ∅)
61: SETKERNELARGUMENTS(`pfKernel3`, `Np`, `dW`, `dSums`, `dC`, `dU`)
62: ENQUEUEKERNEL(`pfKernel3`, (`WGsize` × `NWG`), (`WGsize`), ∅)
63: SETKERNELARGUMENTS(`pfKernel4`, `Np`, `dU`, `dC`, `dX`, `dY`, `dX2`, `dY2`, `dW`)
64: ENQUEUEKERNEL(`pfKernel4`, (`WGsize` × `NWG`), (`WGsize`), ∅)
65: **end for**
66: COPYTOHOST(`dW`, `hW`, **True**, ∅)
67: COPYTOHOST(`dX`, `hX`, **True**, ∅)
68: COPYTOHOST(`dY`, `hY`, **True**, ∅)

Host data structures

- **int** `hV[Nx * Ny * Nf]`: $V_{k,i,j} \rightarrow$ `hV[i * Ny * Nf + j * Nf + k]`
- **float** `hW[Np]`: $W_i \rightarrow$ `hW[i]`
- **float** `hX[Np]`: $P_i^x \rightarrow$ `hX[i]`
- **float** `hY[Np]`: $P_i^y \rightarrow$ `hY[i]`
- **int** `hSbase[|S| * 2]`: Base values for S_i generation.

Device data structures

- **global int** `dV[Nx * Ny * Nf]`: $V_{k,i,j} \rightarrow$ `dV[i * Ny * Nf + j * Nf + k]`
- **global float** `dW[Np]`: $W_i \rightarrow$ `dW[i]`
- **global float** `dX[Np]`: $P_i^x \rightarrow$ `dX[i]`
- **global float** `dY[Np]`: $P_i^y \rightarrow$ `dY[i]`
- **global float** `dX2[Np]`: $P'^x_i \rightarrow$ `dX[i]`
- **global float** `dY2[Np]`: $P'^y_i \rightarrow$ `dY[i]`
- **global float** `dL[Np]`: $L_i \rightarrow$ `dL[i]`
- **global float** `dC[Np]`: $C_i \rightarrow$ `dC[i]`
- **global float** `dU[Np]`: $U_i \rightarrow$ `dU[i]`
- **global int** `dSbase[|S| * 2]`: Base values for S_i generation.
- **global int** `dS[|S| * Np]`: $S_i \rightarrow$ `dI[|S| * i:(|S| * (i + 1) - 1)]`
- **global float** `dSums[Np]`: Intermediate array used for parallel sum calculation.
- **local float** `dlB[Λ]`: $W_i \rightarrow$ `dlB[i % (Λ)]`

`IMG_MAX_WIDTH` denotes OpenCL image maximal width.

Input datasets The benchmark uses randomly generated video sequence, which consists of 10 frames with size 128×128. It uses 10000 particles to approximate the target object. The size of the video sequences and the number of particles used for approximation can be changed via command line.

Partitioned iteration space

- Particles locations recalculation (pfKernel1):

$$P_P = \{(k,\gamma) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p\}$$

$$f(k,i) \rightarrow (k,i)$$

$$f^{-1}(k,\gamma) \rightarrow (k,\gamma)$$

- Points sets generation (pfKernel1):

$$P_S = \{(k,\gamma,y) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p, 0 \leq y < \|S_i\|\}$$

$$f(k,i) \rightarrow (k,i,0 : (\|S_i\| - 1))$$

$$f^{-1}(k,\gamma,y) \rightarrow (k,\gamma)$$

- Likelihood calculation (pfKernel1):

$$P_L = \{(k,\gamma,y) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p, 0 \leq y < \|S_i\|\}$$

$$f(k,i) \rightarrow (k,i,0 : (\|S_i\| - 1))$$

$$f^{-1}(k,\gamma,y) \rightarrow (k,\gamma)$$

- Weights calculation (pfKernel1):

$$P_W = \{(k,\gamma) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p\}$$

$$f^{-1}(k,i) \rightarrow (k,i)$$

$$f^{-1}(k,\gamma) \rightarrow (k,\gamma)$$

- Weights partial sum calculation (pfKernel1):

$$P_{S_w} = \{(k,\nu,\lambda,s) : 0 \leq k < N_f, 0 \leq \nu < N = N_p/\Lambda, 0 \leq \lambda < \Lambda, s \in \{s_i = 2^i, 0 \leq i < \log_2(\Lambda)\}\}$$

  There is no mapping between abstract and concrete iteration spaces in that case, since the reduction operation (summation) is parallelized assuming that the sum operation is commutative and distributed *i.e.* the order of additions is completely different.

- Weights sum calculation (pfKernel2):

$$P_W = \{(k,x) : 0 \leq k < N_f, 0 \leq x < N_p/\Lambda\}$$

$$f^{-1}(k,i) \rightarrow (k, i/\Lambda)$$

$$f^{-1}(k,x) \rightarrow (k, (x \times \Lambda) : (x \times \Lambda + \Lambda - 1))$$

- Weights normalization (pfKernel3):

$$P_N = \{(k,\gamma) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p\}$$

$$f(k,i) \rightarrow (k,i)$$

$$f^{-1}(k,\gamma) \rightarrow (k,\gamma)$$

- Intermediate array U calculation (pfKernel3):

$$P_U = \{(k,\gamma) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p\}$$

$$f(k,i) \rightarrow (k,i)$$

$$f^{-1}(k,\gamma) \rightarrow (k,\gamma)$$

- Intermediate array C calculation (pfKernel3):

$$P_C = \{(k,j) : 0 \leq k < N_f, 0 \leq j < N_p\}$$

$$f(k,i,j) \rightarrow (k,j)$$

$$f^{-1}(k,j) \rightarrow (k, j : (N_p - 1), j)$$

- Particles locations update (pfKernel4):

$$P_F = \{(k,\gamma,x) : 0 \leq k < N_f, 0 \leq \gamma < \Gamma = N_p, 0 \leq x < N_p\}$$

$$f(k,i,j) \rightarrow (k,i,j)$$

$$f^{-1}(k,\gamma,j) \rightarrow (k,\gamma,j)$$

Device memory access mapping

- Particles locations recalculation (pfKernel1):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dX}[\gamma],\texttt{dY}[\gamma],\} \\ G_w(k,\gamma) &= \{\texttt{dX}[\gamma],\texttt{dY}[\gamma],\} \end{aligned}$$

- Points sets generation (pfKernel1):

$$\begin{aligned} G_r(k,\gamma,y) &= \{\texttt{dX}[\gamma],\texttt{dY}[\gamma],\texttt{dSbase}[2\times\texttt{y}],\texttt{dSbase}[2\times\texttt{y}+1]\} \\ G_w(k,\gamma,y) &= \{\texttt{dS}[\gamma\times\texttt{SSize}+\texttt{y}]\} \end{aligned}$$

- Likelihood calculation (pfKernel1):

$$\begin{aligned} G_r(k,\gamma,y) &= \{\texttt{dV}[\texttt{dS}[\gamma\times\texttt{SSize}+\texttt{y}]]\} \\ G_w(k,\gamma,y) &= \{\texttt{dL}[\gamma]\} \end{aligned}$$

- Weights calculation (pfKernel1):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dW}[\gamma],\texttt{dL}[\gamma]\} \\ G_w(k,\gamma) &= \{\texttt{dW}[\gamma],\texttt{dY}[\gamma]\} \end{aligned}$$

- Weights partial sum calculation (pfKernel1):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dW}[\gamma]\} \\ G_w(k,\nu,\lambda=0,s) &= \{\texttt{dSums}[\nu]\} \\ L_r(k,\nu,\lambda) &= \{\texttt{dlB}[\lambda],\texttt{dlB}[\lambda+\texttt{s}]\} \\ L_w(k,\nu,\lambda,s) &= \{\texttt{dlB}[\lambda]\} \end{aligned}$$

- Weights sum calculation (pfKernel2):

$$\begin{aligned} G_r(k,x) &= \{\texttt{dSums}[\texttt{x}]\} \\ G_w(k,x) &= \{\texttt{dSums}[0]\} \end{aligned}$$

- Weights normalization (pfKernel3):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dW}[\gamma]\} \\ G_w(k,\gamma) &= \{\texttt{dW}[\gamma]\} \end{aligned}$$

- Intermediate array U calculation (pfKernel3):

$$G_w(k,\gamma) = \{\texttt{dU}[\gamma]\}$$

- Intermediate array C calculation (pfKernel3):

$$\begin{aligned} G_r(k,j) &= \{\texttt{dW}[\texttt{j}],\texttt{dC}[\texttt{j}-1]\} \\ G_w(k,j) &= \{\texttt{dC}[\texttt{j}]\} \end{aligned}$$

- Particles locations update (pfKernel4):

$$\begin{aligned} G_r(k,\gamma,x) &= \{\texttt{d}[\texttt{x}],\texttt{dU}[\gamma]\} \\ G_{mr}(k,\gamma,x) &= \{\texttt{dX}[],\texttt{dY}[]\} \\ G_w(k,\gamma,x) &= \{\texttt{dX2}[\gamma],\texttt{dY2}[\gamma],\texttt{dW}[\gamma]\} \end{aligned}$$

Performance metrics The performance is evaluated using the following metrics:

- The total time for initial host to device memory transfer.
- The total time for all kernel launches (computation time).
- The time for `dW` device to host transfer.
- The time for `dX` device to host transfer.
- The time for `dY` device to host transfer.
- The total time, required by the OpenCL implementation to perform.

Validation mechanism The benchmark has no validation mechanisms.

Target-specific optimizations

- **Avoiding memory transfers between host and device.** In order to minimize number of memory transfers between host and device the parts of algorithm, which must be executed sequentially have been implemented as OpenCL kernels. Although such implementation is slower, that the host-implementation it doesn't allows keeping all data in device memory only. See [10] for more details.
- **Image memory usage.** Image memory usage for some data objects (C and partial sums) could increase the performance of the device code.

Target-specific optimizations opportunities

- **Avoiding unnecessary synchronizations.** Not all memory barriers used in the algorithm implementation are necessary (see device code abstraction). Removing such barriers can decrease kernel execution time. Also in some cases global memory fences can be replaced by local ones.

2.3.14 PathFinder

The benchmark uses dynamic programming to find a path in the 2D grid from the bottom row to the top row with the minimum accumulated weight. On each step the path moves straight ahead or diagonally ahead.

High-level description

Abstract data structures The benchmark operates on the following floating point data objects:

- W is an input $n \times m$ grid of weights where n is the number of rows and m is the number of columns.
- C is an intermediate $n \times m$ matrix of accumulated weights (costs).

Computation The algorithm iterates over rows, picking for each node a neighboring node in the previous row with the smallest accumulated weight.

$$C_{0,j} = W_{0,j} \quad 0 \leq j \leq m-1$$

$$C_{i,j} = \min(C_{i-1,j-1}, C_{i-1,j}, C_{i-1,j+1}) + W_{i,j}, 0 < i < n, 0 < j \leq m-2$$

$$C_{i,0} = \min(C_{i-1,0}, C_{i-1,1}) + W_{i,0}, 0 < i < n$$

$$C_{i,m-1} = \min(C_{i-1,m-2}, C_{i-1,m-1}) + W_{i,m-1}, 0 < i < n$$

$$R = \min_{0 \leq j < m} (C_{n-1,j})$$

Iteration spaces and dependences The algorithm iterates over the 2D grid starting from the second row:

$$I = \{(i,j) : 1 \leq i < n, 0 \leq j < m\}$$

$$I : \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} n-1 \\ m-1 \\ -1 \\ 0 \end{pmatrix}$$

The computation generates the following dependences:

$$\begin{aligned} \delta_I \quad = \quad & I : (i-1, j-1) \xrightarrow{t} I : (i,j) \wedge \\ & I : (i-1, j) \xrightarrow{t} I : (i,j) \wedge \\ & I : (i-1, j+1) \xrightarrow{t} I : (i,j) \end{aligned}$$

Memory access mapping For $(i,j) \in I$:

$$\begin{aligned} M_r(i,j)|_{j=0} &= \{W_{i,j}, C_{i-1,j}, C_{i-1,j+1}\} \\ M_r(i,j)|_{0<j<m-1} &= \{W_{i,j}, C_{i-1,j}, C_{i-1,j-1}, C_{i-1,j+1}\} \\ M_r(i,j)|_{j=m-1} &= \{W_{i,j}, C_{i-1,j}, C_{i-1,j-1}\} \\ M_w(i,j) &= \{C_{i,j}\} \end{aligned}$$

Low-level implementation details

Device code See Algorithm 69.

Host code See Algorithm 70.

Host data structures

- **int** `hW[n * m]`: $W_{i,j} \rightarrow$ `hW[i * m + j]`
- **int** `hR[m]`: $C_{n-1,j} \rightarrow$ `hR[j]`

Device data structures

- **global int** `dW[n * m]`: $W_{i,j} \rightarrow$ `dW[i * m + j]`
- **global int** `dCin[m]`: $C_{i,j} \rightarrow$ `dCin[j]`
- **global int** `dCout[m]`: $C_{i,j} \rightarrow$ `dCout[j]`
- **local int** `dlP[Λ]`: $C_{i,j} \rightarrow$ `dlP[j % (Λ - S * 2) + S]`
- **local int** `dlR[Λ]`: $C_{i,j} \rightarrow$ `dlR[j % (Λ - S * 2) + S]`

where `S` is the number of rows in a slice processed by a single kernel launch.

Input datasets The benchmark uses a randomly generated 100×100000 table of integer weights.

Partitioned iteration space

$$P = \{(r,\nu,\lambda,i) : 0 \leq r < n, 0 \leq \nu < N = (n \times m)/\Lambda, 0 \leq \lambda < \Lambda, 0 < i < S\}$$

$$f(i,j) = ((i/S) \times S, j/(\Lambda - S \times 2), j \bmod (\Lambda - S \times 2) + S, i \bmod S)$$

$$f^{-1}(r,\nu,\lambda,i) = (r+i, \nu \times (\Lambda - S \times 2) - S + \lambda)$$

Algorithm 69 The Path Finder device code abstraction

Require: $S \times 2 < \Lambda$
Input: `global int dW[]`
Input: `global int dCin[]`
Output: `global int dCout[]`
Local: `local int dlP[`Λ`]`
Local: `local int dlR[`Λ`]`
Input: `int S` ▷ Number of rows in a slice.
Input: `int m` ▷ Number of columns.
Input: `int r` ▷ Base row.
Input: `int b` ▷ Number of border columns.

1: **kernel** PFKERNEL (S, dW, dCin, dCout, m, r, b)
2: smallBlockCols $\leftarrow \Lambda - (S \times 2)$ ▷ Calculate the actual size of the output block.
3: blkXmin $\leftarrow$ (smallBlockCols $\times \gamma$) $-$ b ▷ Starting position of the block (including the ghost zone).
4: blkXmax $\leftarrow$ blkXmin $+ \Lambda - 1$ ▷ End position of the block (including the ghost zone).
5: xidx $\leftarrow$ blkXmin $+ \lambda$ ▷ Column coordinate.
6: validXmin $\leftarrow \max(-$blkXmin$, 0)$ ▷ Index of the first valid element within the block
7: validXmax $\leftarrow \min(\Lambda - 1 -$ (blkXmax $-$ m $+ 1), \Lambda - 1)$ ▷ Index of the last valid element within the block
8: W $\leftarrow \max($validXmin$, \lambda - 1)$
9: E $\leftarrow \min($validXmax$, \lambda + 1)$ ▷ Diagonal elements adjusted according to the boundaries.
10: isValid $\leftarrow \lambda \in$ (validXmin : validXmax $- 1$) ▷ Check whether work item should perform any work (only data within the valid range should be processed).
11: **if** xidx $\in 0 :$ m $- 1$ **then**
12: dlP$[\lambda] \leftarrow$ dCin[xidx]
13: **end if**
14: BARRIER(CLK_LOCAL_MEM_FENCE)
15: **for** i **in** $0 :$ (S $- 1$) **do**
16: isComputed $\leftarrow$ **false**
17: **if** $\lambda \in$ (i $+ 1 :$ $\Lambda -$ i $- 2) \wedge$ isValid **then**
18: isComputed $\leftarrow$ **true**
19: left, up, right $\leftarrow$ dlP[W], dlP$[\lambda]$, dlP[E]
20: shortest $\leftarrow \min($left, up, right$)$
21: index $\leftarrow$ m $\times$ (r $+$ i) $+$ xidx
22: dlR$[\lambda] \leftarrow$ shortest $+$ dW[index]
23: **end if**
24: BARRIER(CLK_LOCAL_MEM_FENCE)
25: **if** isComputed **then**
26: dlP$[\lambda] \leftarrow$ dlR$[\lambda]$
27: **end if** ▷ Probably a bug. Local memory barrier required at this point to ensure that dlP has been fully updated before next iteration.
28: **end for**
29: **if** isComputed **then**
30: dCout[xidx] $\leftarrow$ dlR$[\lambda]$
31: **end if**
32: **end kernel**

Algorithm 70 The Path Finder host code abstraction

```
1: procedure AllocateMemory(n, m)                    ▷ Allocate host memory and device buffers.
2:     hW ← AllocateHostMemory(n × m)
3:     hR ← AllocateHostMemory(m)
4:     dW ← AllocateBuffer((n − 1) × m, READ_OLNY, hW + m)
5:     dCin ← AllocateBuffer(m, READ_WRITE, hW)
6:     dCout ← AllocateBuffer(m, READ_WRITE)
7:     hC[0] ← dCin
8:     hC[1] ← dCout
9: end procedure

10: procedure InitializeMemory(n, m)                 ▷ Initialize host memory.
11:     hW ← RandomElements(n × m)
12: end procedure

Input: int n                                         ▷ Number of rows in input array.
Input: int m                                         ▷ Number of columns in input array.
Input: int b   ▷ Height of the computational block (see Algorithm 69 for more details).
13: size ← n × m
14: b ← S
15: AllocateMemory(n, m)
16: InitializeMemory(n, m)
17: pfProgram ← BuildProgram("pfProgram.cl")
18: pfKernel ← BuildKernel(pfProgram, "pfKernel")
19: src, ret ← 1, 0
20: for r in 0 : (n − 1) : b do
21:     src, ret ← ret, src
22:     S ← min(b, n − r − 1)
23:     SetKernelArguments(pfKernel, S, dW, hC[src], hC[ret], m, r, b)
24:     EnqueueKernel(pfKernel, (size), none, ∅)   ▷ Probably a bug, since only
    (m/(Λ − 2 × S) × Λ) threads perform actual computations.
25: end for
26: CopyToHost(hC[ret], hR, True, ∅)
```

Device memory access mapping Each work-item performs the following reads and writes:

$$
\begin{aligned}
G_r(r,v,\lambda,i) &= \{\texttt{dW}[(\texttt{r}+\texttt{i})\times\texttt{m}+(\Lambda-\texttt{S}\times 2)\times\gamma-\texttt{S}+\lambda], \texttt{dCin}[(\Lambda-\texttt{S}\times 2)\times\gamma-\texttt{S}+\lambda]\} \\
G_w(s,v,\lambda,i) &= \{\texttt{dCout}[(\Lambda-\texttt{S}\times 2)\times\gamma-\texttt{S}+\lambda]\} \\
L_r(s,v,\lambda,i) &= \{\texttt{dlP}[\lambda], \texttt{dlP}[\lambda+1], \texttt{dlP}[\lambda-1], \texttt{dlR}[\lambda]\} \\
L_w(s,v,\lambda,i) &= \{\texttt{dlP}[\lambda], \texttt{dlR}[\lambda]\}
\end{aligned}
$$

Performance metrics The performance is evaluated using the following metric:

- The total time for all kernel launches (computation time).

Validation mechanism The benchmark has no validation mechanisms.

Target-specific optimizations

- **Blocking.** The input data is processed in blocks having the number of columns equal to the number of work items per work group. Consecutive work items of the same group access consecutive memory addresses which improves the memory subsystem performance (caches, coalescing).

- **Ghost zone [14].** The computation of the j^{th} element on the i^{th} iteration requires the $(j-1)^{th}$ and $(j+1)^{th}$ elements from the previous iteration. Since these data can be produced by another work group, the ghost zone technique is used to avoid data races: Each kernel launch processes an N rows slice of input data. Given a block with N rows and M columns the $M-N\times 2$ elements of the last row of the block can be computed by a single work group without communicating with other groups. Therefor, the whole slice is divided into non-overlapping blocks with N rows and $(\Lambda-2\times N)$ columns. Each group allocates a local buffer for Λ elements, which is used to compute the resulting $(\Lambda-2\times N)$ elements without affecting the other groups' data. This technique allows each work group to compute the result independently, although the elements located in the ghost areas are computed twice.

- **I/O buffers alternation.** The output of the kernel execution on the current iteration is used as kernel input for the next one. In order to avoid additional memory transfers, the input and output buffers are logically swapped between the kernel launches:

  ```
  ...
  kernel(buffptr[0], buffptr[1])
  kernel(buffptr[1], buffptr[0])
  kernel(buffptr[0], buffptr[1])
  ...
  ```

  The 2-element array `buffptr` contains the pointers to the device memory buffers.

Target-specific optimizations opportunities

- **Work items number adjusting.** In the current implementation each kernel launch spawns $n \times m$ work items, while only $((m \times \Lambda)/(\Lambda - 2 \times S))$ perform actual computation. Using the precisely required number of work items for each kernel launch should decrease the overhead for kernel launch.

2.3.15 Speckle Reducing Anisotropic Diffusion

High-level description

The benchmark uses Speckle Reducing Anisotropic Diffusion (SRAD) algorithm to remove from an image locally correlated noise, known as speckles, while preserving important image features. See [21] and [22] for more details.

Abstract data structures The benchmark operates on the following data objects:

- I is a compressed $N_r \times N_c$ image.
- C is an intermediate $N_r \times N_c$ array of floats.
- N is an intermediate $N_r \times N_c$ array of floats.
- S is an intermediate $N_r \times N_c$ array of floats.
- W is an intermediate $N_r \times N_c$ array of floats.
- E is an intermediate $N_r \times N_c$ array of floats.

Computation

- Decompression:

$$I_{i,j} = \exp(I_{i,j}/255)$$

- Perform continuous image processing (N_i iterations):
  - Calculate Statistics (Mean and Standard Deviation):

$$\begin{aligned} M &= \left(\sum_{0 \le i < N_r, 0 \le j < N_c} I_{i,j} \right) / (N_r \times N_c) \\ Q &= \left(\left(\sum_{0 \le i < N_r, 0 \le j < N_c} I_{i,j}^2 \right) / (N_r \times N_c) - M^2 \right) / M^2 \end{aligned}$$

- SRAD1. For each $(i,j) : 0 \leq i < N_r, 0 \leq j < N_r$:

$$\begin{aligned}
N_{i,j} &= I_{\max(i-1,0),j} - I_{i,j} \\
S_{i,j} &= I_{\min(i+1,N_r-1),j} - I_{i,j} \\
W_{i,j} &= I_{i,\max(j-1,0)} - I_{i,j} \\
E_{i,j} &= I_{i,\min(j+1,N_c-1)} - I_{i,j} \\
G &= (N_{i,j}^2 + S_{i,j}^2 + W_{i,j}^2 + E_{i,j}^2)/I_{i,j}^2 \\
L &= (N_{i,j} + S_{i,j} + W_{i,j} + E_{i,j})/I_{i,j} \\
N &= 0.5 \times G - 4^{-2} \times L^2 \\
D_1 &= 1 + L/4 \\
Q_2 &= N/D_1^2 \\
D_2 &= (Q_2 - Q)/(Q \times (Q+1)) \\
C_{i,j} &= \min(1/(1+D_2), 1) \\
C_{i,j} &= \max(C_{i,j}, 0)
\end{aligned}$$

- SRAD2. For each $(i,j) : 0 \leq i < N_r, 0 \leq j < N_r$:

$$\begin{aligned}
C_N &= C_{i,j} \\
C_S &= C_{\min(i+1,N_r-1),j} \\
C_W &= C_{i,j} \\
C_E &= C_{i,\min(j+1,N_c-1)} \\
D &= (C_N \times N_{i,j} + C_S \times S_{i,j} + C_W \times W_{i,j} + C_E \times E_{i,j})/4 \\
I_{i,j} &= I_{i,j} + D \times \Lambda
\end{aligned}$$

- Compress image:

$$I_{i,j} = \log(I_{i,j}) * 255$$

Λ is an input parameter equal to 0.5.

Iteration spaces

- Decompression:

$$I_E = \{(i,j) : 0 \leq i < N_r, 0 \leq j < N_c\}$$

- Statistics:

$$I_S = \{(k,i,j) : 0 \leq k < N_i, 0 \leq i < N_r, 0 \leq j < N_c\}$$

- SRAD1:

$$I_1 = \{(k,i,j) : 0 \leq k < N_i, 0 \leq i < N_r, 0 \leq j < N_c\}$$

- SRAD2:

$$I_2 = \{(k,i,j) : 0 \leq k < N_i, 0 \leq i < N_r, 0 \leq j < N_c\}$$

- Compression:

$$I_C = \{(i,j) : 0 \leq i < N_r, 0 \leq j < N_c\}$$

Dependences

$$
\begin{aligned}
\delta_{I_S} &= I_E : (i,j) \xrightarrow{t} I_S : (0,i,j) \wedge \\
&\quad I_2 : (k-1,i,j) \xrightarrow{t} I_S : (k,i,j) \\
\delta_{I_1} &= I_E : (i,j) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_E : (\max(i-1,0),j) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_E : (\min(i+1,N_r-1),j) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_E : (i,\max(j-1,0)) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_E : (i,\min(N_c-1,j+1)) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_2 : (k-1,i,j) \xrightarrow{t} I_1 : (k,i,j) \wedge \\
&\quad I_2 : (k-1,\max(i-1,0),j) \xrightarrow{t} I_1 : (k,i,j) \wedge \\
&\quad I_2 : (k-1,\min(i+1,N_r),j) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_2 : (k-1,i,\max(j-1,0)) \xrightarrow{t} I_1 : (k,i,j) \wedge \\
&\quad I_2 : (k-1,i,\min(N_c,j+1)) \xrightarrow{t} I_1 : (k,i,j) \wedge \\
&\quad I_S : (k-1,0:(N_r-1),0:(N_c-1)) \xrightarrow{t} I_1 : (k,i,j) \\
\delta_{I_2} &= I_E : (i,j) \xrightarrow{t} I_1 : (0,i,j) \wedge \\
&\quad I_2 : (k-1,i,j) \xrightarrow{t} I_2 : (k,i,j) \wedge \\
&\quad I_1 : (k,i,j) \xrightarrow{t} I_2 : (k,i,j) \wedge \\
&\quad I_1 : (k,\min(i+1,N_r-1),j) \xrightarrow{t} I_2 : (k,i,j) \wedge \\
&\quad I_1 : (k,i,\min(j+1,N_c-1)) \xrightarrow{t} I_2 : (k,i,j) \\
\delta_{I_C} &= I_2 : (N_i-1,i,j) \xrightarrow{t} I_C : (i,j)
\end{aligned}
$$

Memory access mapping

- Image Extraction:

$$
\begin{aligned}
M_r(i,j) &= \{I_{i,j}\} \\
M_w(i,j) &= \{I_{i,j}\}
\end{aligned}
$$

- Statistics Calculation:

$$
M_r(k,i,j) = \{I_{i,j}\}
$$

- SRAD1:

$$
\begin{aligned}
M_r(k,i,j) &= \{I_{i,j}, I_{\max(i-1,0),j}, I_{i,\max(j-1,0)}, I_{\min(i+1,N_r-1),j}, I_{i,\min(j+1,N_c-1)}\} \\
M_w(k,i,j) &= \{C_{i,j}, N_{i,j}, S_{i,j}, E_{i,j}, W_{i,j}\}
\end{aligned}
$$

- SRAD2:

$$\begin{aligned} M_r(k,i,j) &= \{C_{i,j}, C_{\min(i+1,N_r-1),j}, C_{i,\min(j+1,N_c-1)}, N_{i,j}, S_{i,j}, E_{i,j}, W_{i,j}\} \\ M_w(k,i,j) &= \{I_{i,j}\} \end{aligned}$$

- Image Compression:

$$\begin{aligned} M_r(i,j) &= \{I_{i,j}\} \\ M_w(i,j) &= \{I_{i,j}\} \end{aligned}$$

Low-level implementation details

Device code

- Decompression - see Algorithm 71.
- Statistics (calculated using reduction):
  - Prepare data for reduction - see Algorithm 72.
  - Statistics - see Algorithm 73.
- SRAD1 - see Algorithm 74.
- SRAD2 - see Algorithm 75.
- Compression - see Algorithm 76.

Algorithm 71 The SRAD device code abstraction (Decompression).

Input: `int Ne` ▷ $N_e = N_r \times N_c$.
Input/Output: `global float dI[]`

```
1: kernel SRADKERNEL1 (Ne, dI)
2:     if γ < Ne then
3:         dI[γ] ← exp(dI[γ]/255)
4:     end if
5: end kernel
```

Host code See Algorithm 77.

Host data structures

- `float hI[Nr * Nc]`: $I_{i,j} \rightarrow$ `hI[j * Nr + i]`
- `global int hIN[Nr]`:`hIN[i]` $= \max(i-1,0)$
- `global int hIS[Nr]`:`hIS[i]` $= \min(i+1,N_r-1)$
- `global int hJW[Nc]`:`hJW[j]` $= \max(j-1,0)$
- `global int hJE[Nc]`:`hJE[j]` $= \min(j+1,N_c-1)$

Algorithm 72 The SRAD device code abstraction (Prepare).

Input: `int Ne` ▷ $N_e = N_r \times N_c$.
Input: `global float dI[]`
Output: `global float dSums[]`
Output: `global float dSums2[]`

1: **kernel** SRADKERNEL2 (Ne, dI, dSums, dSums2)
2: **if** $\gamma <$ Ne **then**
3: dSums$[\gamma] \leftarrow$ dI$[\gamma]$
4: dSums2$[\gamma] \leftarrow$ dI$[\gamma]^2$
5: **end if**
6: **end kernel**

Algorithm 73 The SRAD device code abstraction (Statistics).

Input: `int Ne` ▷ $N_e = N_r \times N_c$.
Input: `int No` ▷ Number of elements in array.
Input: `int Step`
Input: `int Dim`
Input/Output: `global float dSums[]`
Input/Output: `global float dSums2[]`
Local: `local float dlSums[]`
Local: `local float dlSums2[]`

1: **kernel** SRADKERNEL3 (Ne, No, Step, Dim, dSums, dSums2)
2: Nf $\leftarrow \Lambda - ($Dim $\times \Lambda -$ No$)$ ▷ Number of elements in the last block.
3: **if** $\gamma <$ No **then**
4: dlSums$[\lambda] \leftarrow$ dSums$[\gamma \times$ Step$]$
5: dlSums2$[\lambda] \leftarrow$ dSums2$[\gamma \times$ Step$]$
6: **end if**
7: **if** Nf $= \Lambda$ **then**
8: i $\leftarrow 2$
9: **while** i $\leq \Lambda$ **do**
10: **if** i $\mid (\lambda + 1)$ **then**
11: dlSums$[\lambda] \leftarrow$ dlSums$[\lambda] +$ dlSums$[\lambda -$ i$/2]$
12: dlSums2$[\lambda] \leftarrow$ dlSums2$[\lambda] +$ dlSums2$[\lambda -$ i$/2]$
13: **end if**
14: BARRIER(CLK_LOCAL_MEM_FENCE)
15: i $\leftarrow 2 \times$ i
16: **end while**
17: **if** $\lambda = \Lambda - 1$ **then**
18: dSums$[\nu \times$ Step $\times \Lambda] \leftarrow$ dlSums$[\lambda]$
19: dSums2$[\nu \times$ Step $\times \Lambda] \leftarrow$ dlSums2$[\lambda]$
20: **end if**

Algorithm 73 The SRAD device code abstraction (Reduction continues).

```
21:     else
22:         if ν ≠ (Dim − 1) then
23:             i ← 2
24:             while i ≤ Λ do
25:                 if i | (λ + 1) then
26:                     dlSums[λ] ← dlSums[λ] + dlSums[λ − i/2]
27:                     dlSums2[λ] ← dlSums2[λ] + dlSums2[λ − i/2]
28:                 end if
29:                 BARRIER(CLK_LOCAL_MEM_FENCE)
30:                 i ← 2 × i
31:             end while
32:             if λ = Λ − 1 then
33:                 dSums[ν × Step × Λ] ← dlSums[λ]
34:                 dSums2[ν × Step × Λ] ← dlSums2[λ]
35:             end if
36:         else
37:             i ← 2
38:             while i ≤ Λ do
39:                 if Nf ≥ i then
40:                     Df ← i
41:                 end if                          ▷ Df is the nearest power of 2 for Nf.
42:                 i ← 2 × i
43:             end while
44:             i ← 2
45:             while i ≤ Df do
46:                 if i | (λ + 1) ∧ λ < Df then
47:                     dlSums[λ] ← dlSums[λ] + dlSums[λ − i/2]
48:                     dlSums2[λ] ← dlSums2[λ] + dlSums2[λ − i/2]
49:                 end if
50:                 i ← 2 × i
51:             end while
52:             if λ = Λ − 1 then
53:                 for i in (ν × Λ + Df) : (ν × Λ + Nf − 1) do
54:                     dlSums[λ] ← dlSums[λ] + dSums[i]
55:                     dlSums2[λ] ← dlSums2[λ] + dSums2[i]
56:                 end for
57:                 dSums[ν × Step × Λ] ← dlSums[λ]
58:                 dSums2[ν × Step × Λ] ← dlSums2[λ]
59:             end if
60:         end if
61:     end if
62: end kernel
```

Algorithm 74 The SRAD device code abstraction (SRAD1).

Input: `int Ne` ▷ $N_e = N_r \times N_c$.
Input: `float L`
Input: `int Nr`
Input: `float Q`
Input: `global int dIN[]`
Input: `global int dIS[]`
Input: `global int dJW[]`
Input: `global int dJE[]`
Input: `global float dI[]`
Output: `global float dN[]`
Output: `global float dS[]`
Output: `global float dE[]`
Output: `global float dW[]`
Output: `global float dC[]`

1: **kernel** SRADKERNEL4 (Ne, L, Nr, Q, dIN, dIS, dJW, dJE, dI, dN, dS, dW, dE, dC)
2: row $\leftarrow \gamma \bmod$ Nr
3: col $\leftarrow \gamma /$ Nr
4: **if** $\gamma <$ Ne **then**
5: $dJc \leftarrow$ dI$[\gamma]$
6: dlN $\leftarrow$ dI[dIN[row] + Nr $\times$ col] $-$ dJc
7: dlS $\leftarrow$ dI[dIS[row] + Nr $\times$ col] $-$ dJc
8: dlE $\leftarrow$ dI[row + Nr $\times$ dIW[col]] $-$ dJc
9: dlW $\leftarrow$ dI[row + Nr $\times$ dIE[col]] $-$ dJc
10: dG2 $\leftarrow$ (dlN^2 + dlS^2 + dlE^2 + dlW^2)/dJc^2
11: dL $\leftarrow$ (dlN + dlS + dlE + dlW)/dJc
12: dNum $\leftarrow 0.5 \times$ dG2 $-$ dL$^2/16$
13: dDen $\leftarrow 1 +$ dL$/4$
14: dQ $\leftarrow$ dNum/dDen^2
15: dDen $\leftarrow$ (dQ $-$ Q)/(Q $\times$ (Q + 1))
16: dlC $\leftarrow 1/(1 +$ dDen)
17: dlC $\leftarrow \max($dlC$, 0)$
18: dlC $\leftarrow \min($dlC$, 1)$
19: dN$[\gamma] \leftarrow$ dlN
20: dS$[\gamma] \leftarrow$ dlS
21: dW$[\gamma] \leftarrow$ dlW
22: dE$[\gamma] \leftarrow$ dlE
23: dC$[\gamma] \leftarrow$ dlC
24: **end if**
25: **end kernel**

Algorithm 75 The SRAD device code abstraction (SRAD2).

Input: `int Ne` ▷ $N_e = N_r \times N_c$.
Input: `float L`
Input: `int Nr`
Input: `float Q`
Input: `global int dIS[]`
Input: `global int dJE[]`
Input: `global float dN[]`
Input: `global float dS[]`
Input: `global float dE[]`
Input: `global float dW[]`
Input: `global float dC[]`
Input/Output: `global float dI[]`

1: **kernel** SRADKERNEL5 (Ne, L, Nr, Q, dIS, dJE, dI, dN, dS, dW, dE, dC)
2: row $\leftarrow \gamma \bmod$ Nr
3: col $\leftarrow \gamma /$ Nr
4: **if** $\gamma <$ Ne **then**
5: dCN $\leftarrow$ dC[γ]
6: dCS $\leftarrow$ dC[dIS[row] + Nr $\times$ col]
7: dCW $\leftarrow$ dC[γ]
8: dCE $\leftarrow$ dC[row + Nr $\times$ dIE[col]]
9: dD $\leftarrow$ dCN $\times$ dN[γ] + dCS $\times$ dS[γ] + dCW $\times$ dW[γ] + dCE $\times$ dE[γ]
10: dI[γ] $\leftarrow$ dI[γ] + L $\times$ dD/4
11: **end if**
12: **end kernel**

Algorithm 76 The SRAD device code abstraction (Compress).

Input: `int Ne` ▷ $N_e = N_r \times N_c$.
Input/Output: `global float dI[]`

1: **kernel** SRADKERNEL6 (Ne, dI)
2: **if** $\gamma <$ Ne **then**
3: dI[γ] $\leftarrow \log($dI[γ]$) \times 255$
4: **end if**
5: **end kernel**

Algorithm 77 The SRAD host code abstraction.

```
1:  procedure AllocateMemory(Nr, Nc)        ▷ Allocate host memory and device buffers.
2:      hI ← AllocateHostMemory(Nr × Nc)
3:      hIN ← AllocateHostMemory(Nr)
4:      hIS ← AllocateHostMemory(Nr)
5:      hJW ← AllocateHostMemory(Nc)
6:      hJE ← AllocateHostMemory(Nc)
7:      dI ← AllocateBuffer(Nr × Nc, READ_WRITE)
8:      dN ← AllocateBuffer(Nr × Nc, READ_WRITE)
9:      dS ← AllocateBuffer(Nr × Nc, READ_WRITE)
10:     dW ← AllocateBuffer(Nr × Nc, READ_WRITE)
11:     dE ← AllocateBuffer(Nr × Nc, READ_WRITE)
12:     dC ← AllocateBuffer(Nr × Nc, READ_WRITE)
13:     dSums ← AllocateBuffer(Nr × Nc, READ_WRITE)
14:     dSums2 ← AllocateBuffer(Nr × Nc, READ_WRITE)
15:     dIN ← AllocateBuffer(Nr, READ_WRITE)
16:     dIS ← AllocateBuffer(Nr, READ_WRITE)
17:     dJW ← AllocateBuffer(Nc, READ_WRITE)
18:     dJE ← AllocateBuffer(Nc, READ_WRITE)
19: end procedure
20: procedure InitializeMemory(Ni, Nh)          ▷ Initialize host memory.
21:     hI ← ReadImage(Ni, Nh)
22:     for i in 0 : (Nr − 1) do
23:         hIN[i] ← i − 1
24:         hIS[i] ← i + 1
25:     end for
26:     hIN[0] ← 0
27:     hIS[Nr − 1] ← Nr − 1
28:     for i in 0 : (Nc − 1) do
29:         hIW[i] ← i − 1
30:         hIE[i] ← i + 1
31:     end for
32:     hIW[0] ← 0
33:     hIE[Nr − 1] ← Nc − 1
34: end procedure
```

Algorithm 77 The SRAD host code abstraction (continues).

Input: `int Nr` ▷ Number of rows to process.
Input: `int Nc` ▷ Number of cols to process.
Input: `int Ni` ▷ Number of iterations.
Input: `float L` ▷ Lambda multiplier.

35: AllocateMemory(Nr, Nc)
36: InitializeMemory(Nr, Nc)
37: CopyToDevice(dI, hI, **True**, ∅)
38: CopyToDevice(dIN, hIN, **True**, ∅)
39: CopyToDevice(dIS, hIS, **True**, ∅)
40: CopyToDevice(dJW, hJW, **True**, ∅)
41: CopyToDevice(dJE, hJE, **True**, ∅)
42: Ne ← Nr × Nc
43: LWS ← 512
44: NWG ← Ne/LWS
45: **if** NWG × LWS < Ne **then**
46: NWG ← NWG + 1
47: **end if**
48: GWS ← LWS × NWG
49: sradProgram ← BuildProgram("sradProgram.cl")
50: sradKernel1 ← BuildKernel(sradProgram, "sradKernel1")
51: sradKernel2 ← BuildKernel(sradProgram, "sradKernel2")
52: sradKernel3 ← BuildKernel(sradProgram, "sradKernel3")
53: sradKernel4 ← BuildKernel(sradProgram, "sradKernel4")
54: sradKernel5 ← BuildKernel(sradProgram, "sradKernel5")
55: sradKernel6 ← BuildKernel(sradProgram, "sradKernel6")
56: SetKernelArguments(sradKernel1, Ne, dI)
57: EnqueueKernel(sradKernel1, (GWS), (LWS), ∅)
58: **for** k **in** 0 : (Ni − 1) **do**
59: SetKernelArguments(sradKernel2, Ne, dI, dSums, dSums2)
60: EnqueueKernel(sradKernel2, (GWS), (LWS), ∅)
61: No ← Ne
62: Step ← 1
63: Dim ← NWG
64: GWS2 ← GWS

Algorithm 77 The SRAD host code abstraction (continues).

```
65:     while Dim ≠ 0 do
66:         SetKernelArguments(sradKernel3, No, Step, Dim, dSums, dSums2)
67:         EnqueueKernel(sradKernel3, (GWS2), (LWS), ∅)
68:         No ← Dim
69:         if Dim = 1 then
70:             Dim ← 0
71:         else
72:             Step ← Step × LWS
73:             NWG ← Dim/LWS
74:             if NWG × LWS < Dim then
75:                 NWG ← NWG + 1
76:             end if
77:             Dim ← NWG
78:             GWS2 ← NWG × LWS
79:         end if
80:     end while
81:     CopyToHost(dSums, T, True, ∅)
82:     CopyToHost(dSums2, T2, True, ∅)
83:     M ← T/Ne
84:     V ← T2/Ne − M^2
85:     Q ← V/M^2
86:     SetKernelArguments(sradKernel4, Ne, L, Nr, Q, dIN, dIS, dJW, dJE, dI, dN, dS, dW, dE, dC)
87:     EnqueueKernel(sradKernel4, (GWS), (LWS), ∅)
88:     SetKernelArguments(sradKernel5, Ne, L, Nr, Q, dIS, dJE, dI, dN, dS, dW, dE, dC)
89:     EnqueueKernel(sradKernel5, (GWS), (LWS), ∅)
90: end for
91: SetKernelArguments(sradKernel6, Ne, dI)
92: EnqueueKernel(sradKernel6, (GWS), (LWS), ∅)
93: CopyToHost(dI, hI, True, ∅)
```

Device data structures

- **global float** `dI[Nr * Nc]`: $I_{i,j} \rightarrow$ `dI[j * Nr + i]`
- **global float** `dN[Nr * Nc]`: $N_{i,j} \rightarrow$ `dN[j * Nr + i]`
- **global float** `dS[Nr * Nc]`: $S_{i,j} \rightarrow$ `dS[j * Nr + i]`
- **global float** `dW[Nr * Nc]`: $W_{i,j} \rightarrow$ `dW[j * Nr + i]`
- **global float** `dE[Nr * Nc]`: $E_{i,j} \rightarrow$ `dE[j * Nr + i]`
- **global float** `dC[Nr * Nc]`: $C_{i,j} \rightarrow$ `dC[j * Nr + i]`
- **global int** `dIN[Nr]`:`dIN[i]` $= \max(i-1,0)$
- **global int** `dIS[Nr]`:`dIS[i]` $= \min(i+1,N_r-1)$
- **global int** `dJW[Nc]`:`dJW[j]` $= \max(j-1,0)$
- **global int** `dJE[Nc]`:`dJE[j]` $= \min(j+1,N_c-1)$
- **global float** `dSums[Nr * Nc]`: Intermediate buffer, used for reduction operation.
- **global float** `dSums2[Nr * Nc]`: Intermediate buffer, used for reduction operation.
- **local float** `dlSums[`Λ`]`: Intermediate buffer, used for reduction operation.
- **local float** `dlSums2[`Λ`]`: Intermediate buffer, used for reduction operation.

Input datasets The benchmark uses pre-generated 502×498 input image.
Command line parameters:

- The number of iterations (N_i). The default value is 100.
- Lambda multiplier (Λ). The default value is 0.5.
- The Number of rows (N_r). The default value is 502.
- The Number of columns (N_c). The default value is 498. The benchmark processes an $N_r \times N_c$ area of the input image staring from the top left corner.

Partitioned iteration space

- Image Extraction (sradKernel1):

$$P_E = \{(\gamma) : 0 \leq \gamma < \Gamma = N_e\}$$

$$f(i,j) \rightarrow (j \times N_r + i)$$

$$f^{-1}(\gamma) \rightarrow (\gamma \bmod (N_r), \gamma / N_r)$$

- Statistics Calculation (sradKernel2):

$$P_S = \{(k,\gamma) : 0 \leq k < N_i, 0 \leq \gamma < \Gamma = N_e\}$$

$$f(k,i,j) \rightarrow (k, j \times N_r + i)$$

$$f^{-1}(k,\gamma) \rightarrow (k, \gamma \bmod (N_r), \gamma/N_r)$$

- Statistics Calculation (sradKernel3):

$$P_R = \{(k,s,\gamma,i) : 0 \leq k < N_i, s \in \{s_l = \Lambda^l, 0 \leq l < \log_\Lambda N_e, 0 \leq \gamma < \Gamma = N_e/s, i = \{i_l = 2^l, 1 \leq l \leq \log_2 \Lambda\}$$

  There is no mapping between abstract and concrete iteration spaces in this case, since the reduction operation (summation) is parallelized assuming that the sum operation is commutative and distributive *i.e.* the order of additions is changed.

- SRAD1 (sradKernel4):

$$P_1 = \{(k,\gamma) : 0 \leq k < N_i, 0 \leq \gamma < \Gamma = N_e\}$$

$$f(k,i,j) \rightarrow (k, j \times N_r + i)$$

$$f^{-1}(k,\gamma) \rightarrow (k, \gamma \bmod (N_r), \gamma/N_r)$$

- SRAD2 (sradKernel5):

$$P_2 = \{(k,\gamma) : 0 \leq k < N_i, 0 \leq \gamma < \Gamma = N_e\}$$

$$f(k,i,j) \rightarrow (k, j \times N_r + i)$$

$$f^{-1}(k,\gamma) \rightarrow (k, \gamma \bmod (N_r), \gamma/N_r)$$

- Image Compression (sradKernel6):

$$P_C = \{(\gamma) : 0 \leq \gamma < \Gamma = N_e\}$$

$$f(i,j) \rightarrow (j \times N_r + i)$$

$$f^{-1}(\gamma) \rightarrow (\gamma \bmod (N_r), \gamma/N_r)$$

Device memory access mapping

- Image Extraction (sradKernel1):

$$\begin{aligned} G_r(\gamma) &= \{\texttt{dI}[\gamma]\} \\ G_w(\gamma) &= \{\texttt{dI}[\gamma]\} \end{aligned}$$

- Statistics Calculation (sradKernel2):

$$\begin{aligned} G_r(\gamma) &= \{\texttt{dI}[\gamma]\} \\ G_w(\gamma) &= \{\texttt{dSums}[\gamma]\texttt{dSums2}[\gamma]\} \end{aligned}$$

- Statistics Calculation (sradKernel3):

$$\begin{aligned} G_r(k,s,\gamma = \nu \times \Lambda + \lambda, i) &= \{\texttt{dSums}[\gamma \times \texttt{s}], \texttt{dSums2}[\gamma \times \texttt{s}]\} \\ G_w(k,s,\gamma = \nu \times \Lambda + \lambda, i) &= \{\texttt{dSums}[\Lambda \times \texttt{s} \times \nu], \texttt{dSums2}[\Lambda \times \texttt{s} \times \nu]\} \\ L_r(k,s,\gamma = \nu \times \Lambda + \lambda, i) &= \{\texttt{dlSums}[\lambda], \texttt{dlSums2}[\lambda], \texttt{dlSums}[\lambda + \texttt{i}/2], \texttt{dlSums2}[\lambda + \texttt{i}/2]\} \\ L_r(k,s,\gamma = \nu \times \Lambda + \lambda, i) &= \{\texttt{dlSums}[\lambda], \texttt{dlSums2}[\lambda]\} \end{aligned}$$

- SRAD1 (sradKernel4):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dIN}[\gamma \bmod \texttt{Nr}], \texttt{dIS}[\gamma \bmod \texttt{Nr}], \texttt{dJW}[\gamma/\texttt{Nr}], \texttt{dJE}[\gamma/\texttt{Nr}], \texttt{dI}[\gamma], \\ &\quad \texttt{dI}[\texttt{Nr} \times (\gamma/\texttt{Nr}) + \max(\gamma \bmod \texttt{Nr} - 1, 0)], \\ &\quad \texttt{dI}[\texttt{Nr} \times (\gamma/\texttt{Nr}) + \min(\gamma \bmod \texttt{Nr} + 1, \texttt{Nr} - 1)], \\ &\quad \texttt{dI}[\texttt{Nr} \times (\max(\gamma/\texttt{Nr}) - 1, 0) + \gamma \bmod \texttt{Nr}], \\ &\quad \texttt{dI}[\texttt{Nr} \times (\min(\gamma/\texttt{Nr}) - 1, \texttt{Nc} - 1) + \gamma \bmod \texttt{Nr}], \} \\ G_w(k,\gamma) &= \{\texttt{dN}[\gamma], \texttt{dS}[\gamma], \texttt{dW}[\gamma], \texttt{dE}[\gamma], \texttt{dC}[\gamma]\} \end{aligned}$$

- SRAD2 (sradKernel5):

$$\begin{aligned} G_r(k,\gamma) &= \{\texttt{dIS}[\gamma \bmod \texttt{Nr}], \texttt{dJE}[\gamma/\texttt{Nr}], \texttt{dI}[\gamma], \texttt{dC}[\gamma], \texttt{dN}[\gamma], \texttt{dS}[\gamma], \texttt{dW}[\gamma], \texttt{dE}[\gamma], \\ &\quad \texttt{dC}[\texttt{Nr} \times (\gamma/\texttt{Nr}) + \min(\gamma \bmod \texttt{Nr} + 1, \texttt{Nr} - 1)], \\ &\quad \texttt{dC}[\texttt{Nr} \times (\min(\gamma/\texttt{Nr}) - 1, \texttt{Nc} - 1) + \gamma \bmod \texttt{Nr}], \} \\ G_w(k,\gamma) &= \{\texttt{dI}[\gamma]\} \end{aligned}$$

- Image Compression (sradKernel6):

$$\begin{aligned} G_r(\gamma) &= \{\texttt{dI}[\gamma]\} \\ G_w(\gamma) &= \{\texttt{dI}[\gamma]\} \end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- The time to read an image.
- Host arrays (N, S, W, E) initialization time.
- Total time required by OpenCL implementation to perform compilation:
  - Context and Command Queue creation.
  - Device memory allocation.
  - Host to device memory transfer.
  - Kernel launches.
  - Device to host memory transfer.

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can write output image to the file, which can be validated by external tools (not supplied).

Target-specific optimizations

- **Tree reduction.** The OpenCL code implements parallel reduction (sum) using a tree-based approach, which requires additional memory barriers.

Target-specific optimizations opportunities

- **Halo usage.** Current algorithm implementation uses 4 additional arrays (N, S, W, E) to avoid having boundary checks. While this approach eliminates such checks, it also increases the memory consumption and makes the global memory access non regular (from the OpenCL compiler point of view). Another approach to avoiding boundary checks is to use one element width halo for each buffer. See Figure 2.3 for more details. OpenCL image data type can also be used.

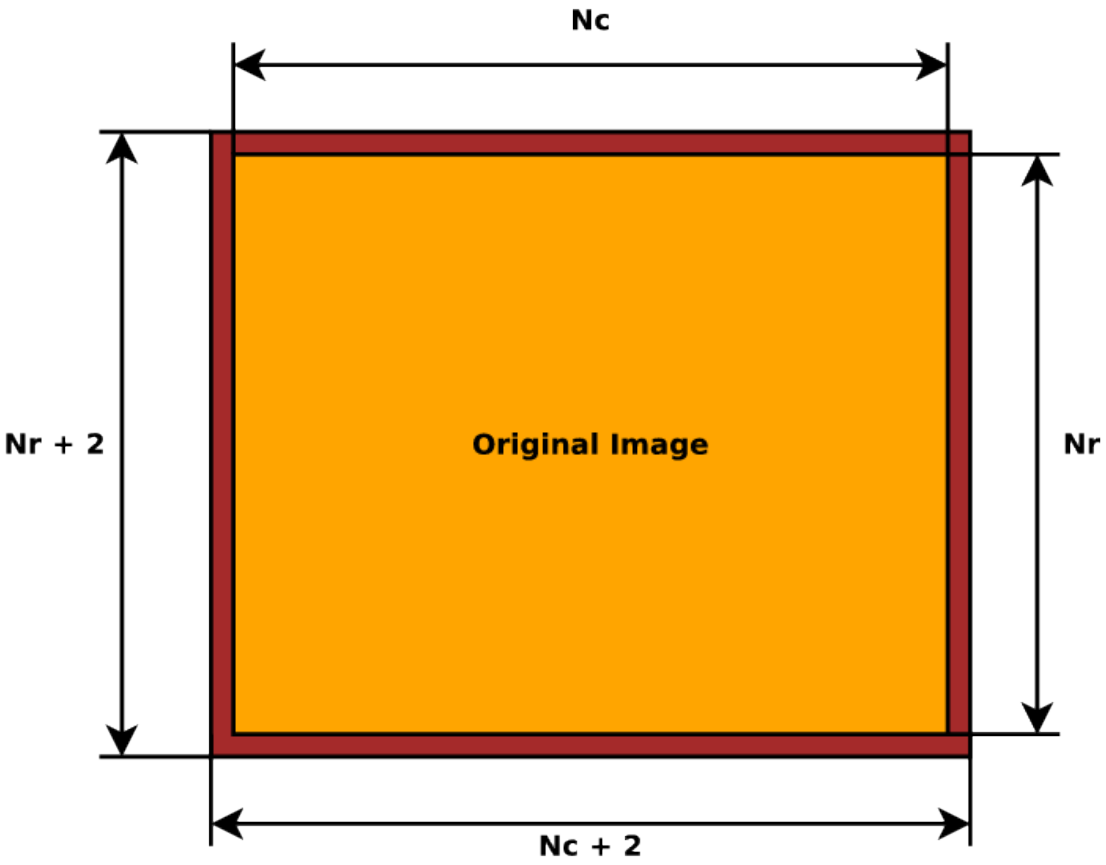

Figure 2.3: Halo for SRAD.

2.3.16 Stream Cluster

High-level description

The benchmark finds a predetermined number of medians in the input stream of points, so that each point is assigned to its nearest median. The quality of the clustering is measured as a sum of square distances (SSQ metric).

Abstract data structures The benchmark operates on the following objects:

- S is an input N_e elements stream (vector) of Points divided into N_c elements chunks. Number of chunks is ($N_s = N_e/N_c$).
- P is a point, characterized by the following features:
  - P^w denotes the weight of the point P.
  - P^c denotes the N_f vector of the coordinates of the point P.
- C^i is an intermediate set of clusters centers ($K_{\min} \leq \|C^i\| \leq K_{\max}$) for the i^{th} chunk.
- C is an output set of clusters centers ($K_{\min} \leq \|C\| \leq K_{\max}$) for the whole stream P.

Computation The algorithm reads point by chunks from the input stream and updates the set of clusters to minimize the SSQ: On the first step the intermediate C_i is calculated for the each chunk C_i. On the second step C is generated as a set of clusters centers for the $\{C_i\}$ set.

- Generate C_i for each input chunk S_i:
  - Randomly select initial centers.
  - Try to improve selected centers by opening new centers (opening a new center can cause some opened centers to be closed). See below for more details.
- Generate C from C_i: The same algorithm is applied to the set of centers obtained on the step one.

In order to determine which which centers should be opened the cost of opening a new center should be calculated. Only this part of the algorithm is implemented as OpenCL kernel (the rest of the document provides the description of this part only):

$$Q_i = \left(\sum_{0 \leq j < N_c} \|P_i - P_j\| \right) \times P_i^w, 0 \leq i < N_c$$

If the new cost Q_i is less than original cost of the point P_i, than it should be marked as a new center.

Iteration space

$$I = \{(i, j) : 0 \leq i < N_c, 0 \leq j < N_c\}$$

Dependences The dependencies are introduced by the other parts of the algorithm (not included in this description).

Memory access mapping

$$\begin{aligned} M_r(i,j) &= \{P_i, P_j\} \\ M_w(i,j) &= \{Q_i\} \end{aligned}$$

Low-level implementation details

The implementation is based on the PARSEC Benchmark Suite implementation [3].

Device code See Algorithm 78.

Device data structures

- **struct** `Point`:

```
struct Point
{
  float cost;
  float weight;
  int assign;
};
```

- **global struct** `Point dP[Nc]`: $S_i \rightarrow$ `dP[i % Nc]`
- **global float** `dPC[Nc * Nf]`: $S_i^c[j] \rightarrow$ `dPC[j * Nc + i % Nc]`
- **global float** `dW[Kmax * Nc]`: Intermediate buffer using for reduction.
- **global int** `dC[Nc]`: $C_i \rightarrow$ `dC[i]`
- **global bool** `dM[Nc]`: $S_i \rightarrow$ `dC[i % Nc]` If `dM[i]` is **true** then S_i is marked as center.
- **local float** `dlPC[Nf]`:$S_i^c[j] \rightarrow$ `dPC[j]`

Input datasets The benchmark can either use randomly generated stream of points, or read the stream from file. The following parameters can be specified via command line:

- Minimal number of centers allowed ($K_{\min}$). The default value is 10.
- Maximal number of centers allowed ($K_{\max}$). The default value is 20.
- Dimension of each data point (N_f). The default value is 256.
- Number of data points in the input stream (N_e). The default value is 65536.
- Number of data points in each chunk (N_c). The default value is 65536.
- Maximal number of intermediate centers. The default value is 1000.

Be default the randomly generated stream of points is used.

Algorithm 78 Stream Cluster device code abstraction.

Input: `global struct Point dP[]`
Input: `global float dPC[]`
Input/Output: `global float dW[]`
Input: `global int dC[]`
Output: `global bool dM[]`
Local: `local float dlPC[]`
Input: `int Nc`
Input: `int Nf`
Input: `int X` ▷ Current center.
Input: `int K` ▷ Number of centers.

1: **kernel** SCKERNEL (dP, dPC, dW, dC, dM, Nc, Nf, X, K)
2: **if** $\gamma < \texttt{Nc}$ **then**
3: **if** $\lambda = 0$ **then**
4: **for** i **in** $0 : (\texttt{Nf} - 1)$ **do**
5: $\texttt{dlPC}[\texttt{i}] \leftarrow \texttt{dPC}[\texttt{i} \times \texttt{Nc} + \texttt{X}]$
6: **end for**
7: **end if**
8: BARRIER(CLK_LOCAL_MEM_FENCE)
9: $\texttt{cost} \leftarrow 0$
10: **for** i **in** $0 : (\texttt{Nf} - 1)$ **do**
11: $\texttt{cost} \leftarrow \texttt{cost} + (\texttt{dPC}[\texttt{i} \times \texttt{Nc} + \gamma] - \texttt{dlPC}[\texttt{i}])^2$
12: **end for**
13: $\texttt{cost} \leftarrow \texttt{cost} \times \texttt{dP}[\gamma].\texttt{weight}$
14: $\texttt{cost0} \leftarrow \texttt{dP}[\gamma].\texttt{cost}$
15: $\texttt{base} \leftarrow \gamma \times (\texttt{K} + 1)$
16: **if** $\texttt{cost} < \texttt{cost0}$ **then**
17: $\texttt{dM}[\gamma] \leftarrow$ **true**
18: $\texttt{dW}[\texttt{base} + \texttt{K}] \leftarrow \texttt{cost} - \texttt{cost0}$
19: **else**
20: $\texttt{addr} \leftarrow \texttt{base} + \texttt{dC}[\texttt{dP}[\gamma].\texttt{assign}]$
21: $\texttt{dW}[\texttt{addr}] \leftarrow \texttt{dW}[\texttt{addr}] + \texttt{cost0} - \texttt{cost}$
22: **end if**
23: **end if**
24: **end kernel**

Partitioned iteration space

$$P = \{(x,\gamma) : 0 \leq x < N_c, 0 \leq \gamma < N_c\}$$

$$f(i,j) \rightarrow (i,j)$$

$$f^{-1}(x,\gamma) \rightarrow (x,\gamma)$$

Device memory access mapping

$$\begin{aligned}
G_r(x,\gamma) &= \{\texttt{dPC}[x : (x + \texttt{Nc} \times (\texttt{Nf} - 1)) : \texttt{Nc}], \texttt{dPC}[\gamma : (\gamma + \texttt{Nc} \times (\texttt{Nf} - 1)) : \texttt{Nc}], \texttt{dP}[\gamma]\} \\
G_{mr}(x,\gamma) &= \{\texttt{dC}[\,], \texttt{dW}[\,]\} \\
G_w(x,\gamma) &= \{\texttt{dM}[\gamma]\} \\
G_{mw}(x,\gamma) &= \{\texttt{dW}[\,]\} \\
L_r &= \{\texttt{dlPC}[\,]\} \\
L_w &= \{\texttt{dlPC}[\,]\}
\end{aligned}$$

Performance metrics The performance of the algorithm implementation is evaluated using the following metrics:

- Total time for OpenCL initialization.
- Total time for Host to Device memory transfers.
- Total time for Device to Host memory transfers.
- Total time for Device memory allocation.
- Total time for Device memory deallocation.
- Total time for all scKernel launches.

Validation mechanism The benchmark has no built-in validation mechanisms. The benchmark can print the computed centers, which can be validated by external tools (not supplied).

Target-specific optimizations The following target-specific optimization are used in the algorithm implementation:

- **Coalescing-friendly memory access (probably CUDA only).** In order to achieve coalesced memory access the input data buffer is logically transposed. For CUDA devices this transformation makes the data access in kernel coalesced. In other hand this greatly reduces the data locality. Although for CUDA devices the advantages of the coalesced memory access overrides the disadvantages of the non-localized memory access, this might not be the same for OpenCL. Since OpenCL standard doesn't require the consecutive memory accesses within threads of the same working group to be coalesced, this transformation might cause a performance degradation.

Target-specific optimizations opportunities

- **Vector operations usage.** Distance calculation in scKernel can be vectorized, which would improve the performance on vector-based architectures.

2.4 Parboil

The Parboil benchmark suite,[14] developed at the University of Illinois at Urbana-Champaign, was the first CUDA benchmarking suite. It was updated [18] to include OpenCL versions of the benchmarks. Unlike Rodinia and SHOC, Parboil includes both base (naïve) implementations and implementations optimized for NVIDIA hardware.

2.4.1 Breadth-First Search

The BFS implementation in Parboil is similar to the one in Rodinia (see §2.3.2). The difference is that the Parboil implementation assigns costs to each edge, which requires locks to ensure the new frontier is correctly computed. The locks are implemented using the following OpenCL built-in functions: `atom_min`, `atom_xchg` and `atom_add`.

2.4.2 SGEMM

The base implementation of SGEMM in Parboil uses Algorithm 79. This unoptimized kernel is equivalent to the optimized `sgemmNT` kernel in SHOC (§2.2.7).

Algorithm 79 Parboil SGEMM device code abstraction.

Input: `global const float dA[]`
Input: `global const float dB[]`
Input: `int Na`
Input: `int Nb`
Input: `int Nc`
Input: `int k`
Input: `float alpha`
Input: `float beta`
Input/Output: `global float dC[]`

1: **kernel** SGEMMNT (dA, dB, Na, Nb, Nc, k, alpha, beta, dC)
2: sum $\leftarrow 0$
3: **for** i **in** $0 : (\mathtt{k} - 1)$ **do**
4: sum $\leftarrow$ sum + dA[$\gamma_0 + \mathtt{i} \times \mathtt{Na}$] $\times$ dB[$\gamma_1 + \mathtt{i} \times \mathtt{Nb}$]
5: **end for**
6: dC[$\gamma_0 + \gamma_1 \times \mathtt{Nc}$] $\leftarrow$ dC[$\gamma_0 + \gamma_1 \times \mathtt{Nc}$] $\times$ beta + alpha $\times$ sum
7: **end kernel**

2.4.3 Sparse Matrix-Vector Multiplication

The implementation of SpMV in Parboil uses the Ellpack format, while the implementations of SpMV in SHOC use the Compressed Sparse Row (CSR) and Ellpack-R formats (§2.2.8).

2.4.4 Stencil

The base implementation of a 7-point 3D stencil operation in Parboil uses Algorithm 80. The Stencil implementation in SHOC is that of a 9-point 2D stencil operation (§2.2.2).

[14] `http://impact.crhc.illinois.edu/parboil.aspx`

Algorithm 80 The Parboil base Stencil device code abstraction.

```
1: procedure INDEX3D(Nx, Ny, i, j, k)
2:     return i + Nx × (j + Ny × k)
3: end procedure
Input: global const dAin[]
Input: int Nx, Ny, Nz
Input: float c0, c1
Output: global float dAout[]
4: kernel STENCILKERNEL1 (dAin, Nx, Ny, Nz, c0, c1, dAout)
5:     i ← γ0 + 1
6:     j ← γ1 + 1
7:     k ← γ2 + 1
8:     if i < Nx − 1 then
9:         dAout[INDEX3D(Nx, Ny, i, j, k)] ←
10:            c1 × Σ_{i'=i±1} Σ_{j'=j±1} Σ_{k'=k±1} dA[INDEX3D(Nx, Ny, i', j', k')] +
11:            c0 × dA[INDEX3D(Nx, Ny, i, j, k)]
12:    end if
13: end kernel
```

2.4.5 Other benchmarks

The other Parboil benchmarks [18] are:

- Cutoff-limited Coulombic Potential
- MRI Cartesian Gridding
- Histogram
- Lattice-Boltzman Method simulation
- MRI Non-Cartesian Q Matrix Calculation
- Sum of Absolute Differences
- Two-Point Angular Correlation Function

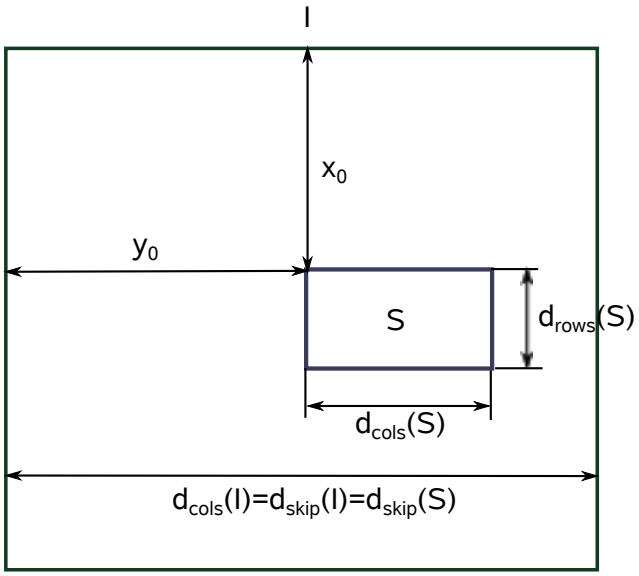

Figure 2.4: The conceptual schema of the image representation of the image in the memory. Image S is a sub-image of image I.

2.5 RealEyes

2.5.1 Arithmetic Operations

In image processing, per pixel arithmetic operations are an important common use case. The operation might even seem simplistic, but it is important from the compiler point-of-view for two reasons. First, arithmetic operations take lots of computing time, so it is desirable to complete them in parallel. Second, because they can be accelerated considerably using local memory. In this section we introduce some standard concepts of image storage as well as the importance of the local accelerator memory even when performing seemingly trivial tasks.

Initially, when an image I is allocated, it is stored in memory as a continuous array. In many cases, however, we want to work only with sub-pictures, which are usually rectangular or cubical subsets of the original image. In this case, to avoid the need for reallocating and copying the image into a new array, we create the sub-picture by defining a set of properties. On Fig. 2.4 we can see an image representation in the memory. The properties describing the image S are: the number of rows ($\texttt{rows}(S)$), the number of columns ($\texttt{cols}(S)$), the distance between two consecutive lines ($\texttt{step}(S)$) and the offset ($\texttt{offset}(S)$). These values are derived from the original image and the desired properties of the sub-image.

Let's take an example. Consider an image I of size $\{\texttt{rows}(I) \times \texttt{cols}(I)\}$. If we want to find a pixel (x,y) in I we can do so as:

$$I_{x,y} = I[\texttt{rows}(I) \times x + y] \tag{2.19}$$

Now we want to consider an image $S \subset I$ of size $\{\texttt{rows}(S) \times \texttt{cols}(S)]$, where $S_{0,0} = I_{x_o,y_o}$. If

we want to find pixel (q,w) in S, we can do that as:

$$S_{q,w} = I[\texttt{offset}(S) + \texttt{step}(S) \times q + w] \tag{2.20}$$

where $\texttt{offset}(S) = \texttt{rows}(I) \times x_0 + y_0$. To simplify the formalism, we consider $S_{0,0} = I[\texttt{offset}(S)]$ and

$$S_{q,w} = S[\texttt{step}(S) \times q + w] \tag{2.21}$$

We apply the image processing operators on images like S. In this section we consider arithmetic and trigonometric operators. These are used to transform the images into different fields. Simple operators are, for example, sin, cos, exp, or any closed combination of thereof. For the sake of simplicity, we only consider the exp operator; the other combinations follow similarly.

Notation

Through this section we are using the following notations:

- λ_1 the current row in the 2D iteration as returned by the OpenCL framework
- λ_0 the current column in the 2D iteration as returned by the OpenCL framework
- τ_{λ_1} the current row of the thread within the group of threads
- τ_{λ_0} the current column of the thread within the group of the threads
- ν_1 the current row of the group within the groups of the kernel
- ν_0 the current column of the group within the groups of the kernel
- $\texttt{rows}(.)$ the number of rows of the image
- $\texttt{cols}(.)$ the number of columns of the image
- N_1 the number group rows (the number of groups is $N_1 \times N_0$)
- N_0 the number of group columns

Benchmark

The general equation can be stated as:

$$\underline{\underline{\mathbf{R}}} = \exp \underline{\underline{\mathbf{A}}} \tag{2.22}$$

High-level Description

The algorithm, operates on the following data:

- $\underline{\underline{\mathbf{A}}}$ is the input matrix of size $m \times n$.
- $\underline{\underline{\mathbf{R}}}$ is the result matrix of size $m \times n$.

Computation We implement two versions of the algorithm. In the naïve implementation we implement the arithmetic operation directly, whereas in the optimized calculation we use the local memory.

Algorithm 81 Arithmetic Operator

1: **kernel** EXP (image, result)
Input: $\lambda_1, \lambda_0, \tau_{\lambda_1}, \tau_{\lambda_0}, \nu_1, \nu_0, N_1, N_0$
2: $q_{\texttt{image}} \leftarrow \texttt{step}(\texttt{image}) \cdot \lambda_1 + \lambda_0$ ▷ We compute the image specific index for the current thread.
3: $q_{\texttt{result}} \leftarrow \texttt{step}(\texttt{result}) \cdot \lambda_1 + \lambda_0$ ▷ We compute the result specific index for the current thread.
4: $\texttt{result}[q_{\texttt{result}}] \leftarrow \exp(\texttt{image}[q_{\texttt{image}}])$
5: **end kernel**

Algorithm 82 Optimized Arithmetic Operator

1: **kernel** EXP (image, result)
Input: $\lambda_1, \lambda_0, \tau_{\lambda_1}, \tau_{\lambda_0}, \nu_1, \nu_0, \texttt{rows}, \texttt{cols}, N_1, N_0$
Local: $\texttt{buff}[N_0 \times N_1]$
2: $q_{\texttt{image}} \leftarrow \texttt{step}(\texttt{image}) \cdot \lambda_1 + \lambda_0$ ▷ We compute the image specific index for the current thread.
3: $q_{\texttt{buff}} \leftarrow N_0 \cdot \tau_{\lambda_1} + \tau_{\lambda_0}$ ▷ We compute the buffer specific index for the current thread.
4: $\texttt{buff}[q_{\texttt{buff}}] \leftarrow \texttt{image}[q_{\texttt{image}}]$
5: BARRIER(CLK_LOCAL_MEM_FENCE) ▷ At this point we have filled up the local memory with the input image data.
6: $q_{\texttt{result}} \leftarrow \texttt{step}(\texttt{result}) \cdot \lambda_1 + \lambda_0$ ▷ We calculate the result specific index for the current thread.
7: $\texttt{result}[q_{\texttt{result}}] \leftarrow \exp(\texttt{buff}[q_{\texttt{buff}}])$
8: **end kernel**

Naïve Implementation The naive implementation can be observed in algorithm 81.
In the optimized implementation we make sure, that all the reads and writes are coalesced. The optimized implementation can be observed in algorithm 82.

Dependences The computation is embarrassingly parallel: there are no dependences beyond the availability of the input.

Complexity The arithmetic operations have $O(\texttt{rows}(I) \times \texttt{cols}(I))$ complexity.

Memory Access Mapping and Complexity This algorithm is a 2D iteration. The iteration variables are λ_1 and λ_0. In the light of the above, the memory access mapping can be written as:

$$G_r(\lambda_1, \lambda_0) = \texttt{image}(\lambda_1, \lambda_0) \tag{2.23}$$
$$G_w(\lambda_1, \lambda_0) = \texttt{result}(\lambda_1, \lambda_0) \tag{2.24}$$

Computational Complexity The computational complexity is linear with the number of pixels $O(\Lambda_1 \times \Lambda_0)$

Optimized Implementation In the optimized implementation we are using the local cache to read and write the memory in a coalesced way. This approach also works, and is beneficial also if the size of the types differ, that is to say if we do type-casting during the operation.

2.5.2 GEMM – General Matrix Multiply

The benchmark uses a simple general matrix multiplication algorithm to test the effects of the local memory on the computation performance. The general equation can be stated as:

$$\underline{\underline{\mathbf{R}}} = \alpha \underline{\underline{\mathbf{A}}} \cdot \underline{\underline{\mathbf{B}}} + \beta \underline{\underline{\mathbf{C}}} \tag{2.25}$$

or

$$\underline{\underline{\mathbf{C}}} = \alpha \underline{\underline{\mathbf{A}}} \cdot \underline{\underline{\mathbf{B}}} + \beta \underline{\underline{\mathbf{C}}} \tag{2.26}$$

High-level Description

The benchmark operates on he following floating point data objects:

- $\underline{\underline{\mathbf{A}}}$ is the first multiplier matrix of size $m \times n$
- $\underline{\underline{\mathbf{B}}}$ is the second multiplier matrix of size $n \times o$
- $\underline{\underline{\mathbf{C}}}$ is the addend matrix of size $m \times o$
- $\underline{\underline{\mathbf{R}}}$ is the result matrix of size $m \times o$

Depending on the application the multiplication can either be implemented in-place of the $\underline{\underline{\mathbf{C}}}$ matrix or using the functional programming paradigm creating a result $\underline{\underline{\mathbf{R}}}$ matrix.

Computation The algorithm uses the definition of the matrix multiplication[15]:

$$\underline{\underline{\mathbf{R}}}_{i,j} = \beta \underline{\underline{\mathbf{C}}}_{i,j} + \sum_{q=0}^{n-1} \alpha \underline{\underline{\mathbf{A}}}_{i,q} \underline{\underline{\mathbf{B}}}_{q,o} \tag{2.27}$$

We compare two implementations. The first is the naïve implementation, whereas the second is optimized for the local memory usage and coalesced reading.

Naïve Implementation

The naïve implementation can be found in algorithm 83.

Algorithm 83 GEMM Naïve Implementation

Input: `global float A[]`
Input: `global float B[]`
Input: `global float C[]`
Output: `global float R[]`
CLVar: λ_1, λ_0 ▷ current row: λ_1, current column: λ_0

1: **kernel** GEMM_NAIVE (A,B,C,R)
2: `sub` $\leftarrow 0$
3: **for** q **in** $0 \ldots \texttt{cols(A)} - 1$ **do**
4: `sub` $\leftarrow$ `sub` $+ \texttt{A}[\lambda_1 \times \texttt{cols(A)} + q] \cdot \texttt{B}[q \times \texttt{cols}(B) + \lambda_0]$ ▷ we calculate the multiplicand
5: **end for**
6: $R[\lambda_1 \times \texttt{cols}(R) + \lambda_0] \leftarrow \texttt{sub} + \texttt{C}[\lambda_1 \times \texttt{cols(C)} + \lambda_0]$ ▷ we calculate the linear combination
7: **end kernel**

Computational Dependencies The computation does not generate any dependencies beyond the dependencies on the availability of the input.

Memory Access Mapping It is important to note that most of the memory accesses of the naïve kernel are non-coalesced, which means that the computation time cannot be estimated with the global memory bandwidth. (There is only one iteration.)

$$G_r(\lambda_1, \lambda_0) = A(\lambda_1, \cdot) \cup B(\cdot, \lambda_0) \cup C(\lambda_1, \lambda_0) G_w(\lambda_1, \lambda_0) \qquad = R(\lambda_1, \lambda_0) \tag{2.28}$$

Complexity The row of matrix A and the column of matrix B is accessed in the calculation of each final point. This results in a memory access complexity of:

$$\text{uncoalesced reads} : m \times n \times o \tag{2.29}$$
$$\text{coalesced reads} : m \times n \times o \tag{2.30}$$
$$\text{coalesced writes} : m \times o \tag{2.31}$$
$$\text{local writes} : 1 \tag{2.32}$$
$$\text{local reads} : 1 \tag{2.33}$$

[15] we use the informatics notation in the algorithms, where the indices start from 0

$O(m \times n \times o)$. This complexity is to be understood for uncoalesced global memory access count.

Computation Complexity The basic algorithm has an $O(m \times n \times o)$ complexity.

Optimized Implementation

The optimized implementation is using a local "cache"-like buffer which is much faster than the global memory. On top of this the global memory accesses are solely coalesced. The memory is accessed linearly from the input images.
The algorithm creates a window of the size of the work group of threads. The windows are read into the local memory where the window related multiplications are performed. After the work on the window is finished, the window is moved along the multiplication axis.
The optimized implementation can be observed in algorithm 84.

Computational Dependencies In the optimized computation we have introduced two new local memories: `buffA` and `buffB`. The buffers hold the patch of the input images on which the current calculation is performed.

$$\delta_{I_R} = R \xrightarrow{t} (\texttt{sub} \cup C) \bigwedge \tag{2.34}$$

$$\texttt{sub} \xrightarrow{t} \texttt{buffA} \cup \texttt{buffB} \tag{2.35}$$

Memory Access Mapping and Complexity The complete data is accessed during the iteration.

$$G_r = \{A \cup B \cup C\} \tag{2.36}$$

$$G_w = \{R\} \tag{2.37}$$

Matrices A and B are accessed strictly once in coalesced order. The local matrices are accessed by each result pixel.

$$\text{coalesced reads} : m \times n + n \times o \tag{2.38}$$

$$\text{coalesced writes} : m \times o \tag{2.39}$$

$$\text{local writes} : m \times n + n \times o \tag{2.40}$$

$$\text{local reads} : m \times n \times o \tag{2.41}$$

Computational Complexity The basic algorithm has an $O(m \times n \times o)$ complexity.

Optimization Opportunities

Target-specific Optimizations The size of the work-group can be tuned for the target device.

Optimal Alignment If the data is aligned to a special value (row and cols are divisible by 4 or 64), we can set up kernel without boundary checking. This can accelerate the kernels. Also if the image cannot be embedded into an aligned memory space we can still cut up the image into 4 sub-images. One image with sizes divisible by 64 and three non-symmetric matrices. In this case two differently optimized kernels can be called on the sub images accelerating the computation time.

Algorithm 84 GEMM Optimized Implementation

Input: `global float A[]`
Input: `global float B[]`
Input: `global float C[]`
Output: `global float R[]`
CLVar: $\lambda_1, \lambda_0, \tau_{\lambda_1}, \tau_{\lambda_0}, \nu_1, \nu_0$ ▷ current row: λ_1, current column: λ_0, thread row: τ_{λ_1}, thread column: τ_{λ_0}, the row of the group: ν_1, the column of the group: ν_0
Local: `local buffA[16(16+1)]` ▷ We allocate +1 to avoid memory banks conflicts
Local: `local buffB[16(16+1)]` ▷ We allocate +1 to avoid memory banks conflicts

1: **kernel** GEMM_OPTIMIZED (A, B, C, R)
2: $d \leftarrow 16$ ▷ block size; We are working with 16×16 threads per group.
3: $\mathtt{A}_{\mathtt{begin}} \leftarrow d \cdot \nu_1 \cdot \mathtt{cols}(\mathtt{A})$ ▷ the first element of the A matrix to process
4: $\mathtt{B}_{\mathtt{begin}} \leftarrow d \cdot \nu_0$ ▷ the first element of the B matrix to process
5: $\mathtt{step}(\mathtt{A}) \leftarrow d$
6: $\mathtt{step}(\mathtt{B}) \leftarrow d \cdot \mathtt{cols}(\mathtt{B})$
7: $N_{\mathtt{blocks}} \leftarrow (\mathtt{cols}(\mathtt{A}) + d - 1) \mod d$ ▷ the number of total blocks to process
8: $\mathtt{sub} \leftarrow 0$ ▷ $\mathtt{sub} \leftarrow 0$
9: $\mathtt{A}_{\mathtt{offset}} \leftarrow \tau_{\lambda_1} + \tau_{\lambda_0} \cdot \mathtt{cols}(\mathtt{A})$
10: $\mathtt{B}_{\mathtt{offset}} \leftarrow \tau_{\lambda_1} + \tau_{\lambda_0} \cdot \mathtt{cols}(\mathtt{B})$
11: **for** block **in** $0 \ldots N_{\mathtt{blocks}} - 1$ **do**
12: **if** $\left(\mathtt{block} \cdot d + \tau_{\lambda_1} < \mathtt{cols}(\mathtt{A}) \bigwedge \nu_1 \cdot d + \tau_{\lambda_0} < \mathrm{rows}(\mathtt{A})\right)$ **then**
13: $\mathtt{buffA}[\tau_{\lambda_0} \cdot (d+1) + \tau_{\lambda_1}] \leftarrow \mathtt{A}[\mathtt{A}_{\mathtt{begin}} + \mathtt{A}_{\mathtt{offset}}]$
14: **else**
15: $\mathtt{buffA}[\tau_{\lambda_0} \cdot (d+1) + \tau_{\lambda_1}] \leftarrow 0$
16: **end if**
17: **if** $\left(\mathtt{block} \cdot d + \tau_{\lambda_0} < \mathrm{rows}(\mathtt{B}) \bigwedge \nu_0 \cdot d + \tau_{\lambda_1} < \mathtt{cols}(\mathtt{B})\right)$ **then**
18: $\mathtt{buffB}[\tau_{\lambda_1} \cdot (d+1) + \tau_{\lambda_0}] \leftarrow \mathtt{B}[\mathtt{B}_{\mathtt{begin}} + \mathtt{B}_{\mathtt{offset}}]$
19: **else**
20: $\mathtt{buffB}[\tau_{\lambda_1} \cdot (d+1) + \tau_{\lambda_0}] \leftarrow 0$
21: **end if**
22: BARRIER(CLK_LOCAL_MEM_FENCE)
23: **for** i **in** $0 \ldots 20 - 1$ **do**
24: $\mathtt{sub} \leftarrow \mathtt{sub} + \mathtt{buffA}[\tau_{\lambda_1} \cdot d + i] \cdot \mathtt{buffB}[\tau_{\lambda_0} \cdot d]$
25: **end for**
26: BARRIER(CLK_LOCAL_MEM_FENCE)
27: $\mathtt{A}_{\mathtt{begin}} \leftarrow \mathtt{A}_{\mathtt{begin}} + \mathtt{step}(\mathtt{A})$
28: $\mathtt{B}_{\mathtt{begin}} \leftarrow \mathtt{B}_{\mathtt{begin}} + \mathtt{step}(\mathtt{B})$
29: **end for**
30: **if** $\lambda_1 < \mathtt{rows}(\mathtt{A}) \bigwedge \lambda_0 < \mathtt{cols}(\mathtt{B})$ **then**
31: $R[\lambda_1 \times \mathtt{cols}(\mathtt{B}) + \lambda_0] \leftarrow \mathtt{sub} + \mathtt{C}[\lambda_1 \times \mathtt{cols}(\mathtt{C}) + \lambda_0]$
32: **end if**
33: **end kernel**

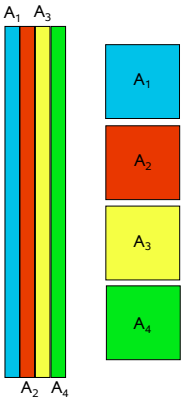

Figure 2.5: The conceptual schema of the image repacking. Images $A_1 \ldots A_4$ are repacked as columns of image I.

SIMD Enforcement On architectures which do support SIMD instructions, the data can be collected into a local aligned transposed buffer. The multiplication can then be performed using the SIMD instructions. Some architectures might support SIMD operations.

2.5.3 Image Statistics

This benchmark extracts statistical properties of an image: maximum value, minimum value and the sum of the image's pixels. To simplify calculation, we work on several images at once and the images are re-packed into a big image as the big image's columns. The concept can be seen on fig. 2.5.

Input Data Structures

- $I_{0\ldots n-1}$ is the set of input images (all images have the same size)
- `sample` is an image containing the repacked I images
- `M` is an intermediate array containing intermediate results

We repack the images I_q into a single image `sample` the following way:

$$\texttt{sample}(q,w) = I_w\,(q \div \texttt{cols}(I_w), q \mod \texttt{cols}(I_w)) \tag{2.42}$$

The `M` structure is an array of records, each containing a maximum value, a minimum value and a sum value. These are noted by $\max(\texttt{sample}[i])$, $\min(\texttt{sample}[i])$ and $\text{sum}(\texttt{sample}[i])$ respectively.
In the end of the calculation `M` will contain the following values:

$$\forall i \in \{0 \ldots \texttt{cols}(\texttt{sample}) - 1\} \tag{2.43}$$

$$\max(\texttt{M[i]}) = \max\{\texttt{sample}(*,i)\} = \max\{I_i\} \tag{2.44}$$

$$\min(\texttt{M[i]}) = \min\{\texttt{sample}(*,i)\} = \min\{I_i\} \tag{2.45}$$

$$\text{sum}(\texttt{M[i]}) = \sum_{p \in \{\texttt{sample}(*,i)\}} p = \sum_{p \in I_i} p \tag{2.46}$$

In the remainder of this discussion, for brevity, we consider the maximum value only, the rest of the values can be treated similar way.

It is difficult to extract the maximum on an parallel accelerator efficiently. In the naïve algorithm we consider each pixel p and compare it with a candidate value c. If $p > c$ then we replace the candidate value with p. In a parallel accelerator all the comparisons would be carried out once, so in the end we would choose an undefined value. On the other hand, if we would make the comparison atomically, then we would end up with a sequential algorithm. The solution is to work by tiles, and collect the statistical properties of the patches in a temporary array of records.

Computation The computation is done in two phases. In the first phase we divide the image into tiles and we extract the maximum value of each group into a contracted array (`M`). In the second phase we iterate on the array `M`, contracting it further in each iteration until in the end we are left with a single line containing the maximum value of each image. For the simplicity of the implementation we choose the patches of size $(k \times \texttt{cols}(\texttt{sample}))$. This way we only contract the rows and not the columns. The algorithm can be observed in algorithm 85.

Algorithm 85 Image Statistics Conceptual Schema

Input: `sample`
Input: Λ_1 ▷ block size; number of rows in the group
Input: Λ_0 ▷ block size; number of columns in the group
1: CREATEARRAY(`M`, $\texttt{rows}(\texttt{sample}) \div \Lambda_1$, $\texttt{cols}(\texttt{sample})$)
2: **for** q **in** $0 \ldots \texttt{rows}(sample) \div \Lambda_1 - 1$ **do** ▷ We initialize the `M` array.
3: **for** w **in** $0 \ldots \Lambda_0 - 1$ **do**
4: $\max(\texttt{M}(q,w)) \leftarrow \max\{\texttt{sample}(q\Lambda_1 \ldots q(\Lambda_1+1), w)\}$ ▷ We contract the image by extracting the maximum value from each tile
5: **end for**
6: **end for**
7: $r \leftarrow \texttt{rows}(\texttt{sample}) \div \Lambda_1$
8: $s \leftarrow 1$
9: **while** $r > 1$ **do** ▷ We keep exponentially contracting the image until we get a single line containing the maxes for all the images in the `sample`
10: **for** q **in** $0 \ldots \texttt{rows}(M) \div (\Lambda_1 s) - 1$ **do**
11: **for** w **in** $0 \ldots \Lambda_0 - 1$ **do**
12: $$\max(\texttt{M}(q * s, w)) \leftarrow \max\left\{ \bigcup_{i \in 0 \ldots \Lambda_1 - 1} \texttt{M}((q+i)s, w) \right\}$$
13: **end for**
14: **end for**
15: $r \leftarrow r \div \Lambda_1$
16: $s \leftarrow s\Lambda_1$
17: **end while**

Now we are going to discuss the dependencies in the memory accesses. In the above schema the maximum values are collected in the first line of the patch. This way the patches can be further contracted independently. In each iteration i we access the contracted image or array of

records, so the memory access evolves as follows:

$$M_r(0) = \texttt{sample} \tag{2.47}$$

$$M_r(i>0) = \bigcup_{q \in \frac{\texttt{rows(M)}}{\Lambda_1^i}} \texttt{M}(q\Lambda_1^i, *) \tag{2.48}$$

$$M_w(i) = \bigcup_{q \in \frac{\texttt{rows(M)}}{\Lambda_1^{i+1}}} \texttt{M}(q\Lambda_1^{i+1}, *) \tag{2.49}$$

If we consider the memory access complexity, we have to note that all the global reads and writes are coalesced. The memory complexity is as follows:

$$\text{coalesced reads} : \frac{\texttt{cols(sample)} \cdot \texttt{rows(sample)}}{\Lambda_1^i} \tag{2.50}$$

$$\text{coalesced writes} : \frac{\texttt{cols(sample)} \cdot \texttt{rows(sample)}}{\Lambda_1^{i+1}} \tag{2.51}$$

$$\text{local reads} : \frac{\texttt{cols(sample)} \cdot \texttt{rows(sample)}}{\Lambda_1^i} \tag{2.52}$$

$$\text{local writes} : \frac{\texttt{cols(sample)} \cdot \texttt{rows(sample)}}{\Lambda_1^{i+1}} \tag{2.53}$$

Low-level Implementation Details

The first and the second phase of the algorithm are implemented as two separate kernels. The kernels are then called in a wrapper function. The implementation can be seen in algorithm 86.

As presented above, after making the copy to the local memory of the group of threads, only one thread per column continues the computation. As we work simultaneously n images, this results working with one thread per $(d_\tau \Lambda_0) \div n$. The optimal number of images[16] is $(d_\tau \Lambda_0) \div 2$.

Computational Dependencies In the kernel every information us cached locally before it is processed. This way global memory access is always coalesced.

$$\delta_{I_R} = M \xrightarrow{t} \texttt{buff} \bigwedge \tag{2.54}$$

$$\texttt{buff} \xrightarrow{t} \texttt{sample} \tag{2.55}$$

Low-level Memory Mapping From the above implementation, we can see that the calculation does not depend on the host's memory once the image is on the device. The device memory access evolves as follows:

$$G_r(0) = \texttt{sample} \tag{2.56}$$

$$G_r(i>0) = \bigcup_{q \in \frac{\texttt{rows(M)}}{\Lambda_1^i}} \texttt{M}(q\Lambda_1^i, *) \tag{2.57}$$

$$G_w(i) = \bigcup_{q \in \frac{\texttt{rows(M)}}{\Lambda_1^{i+1}}} \texttt{M}(q\Lambda_1^{i+1}, *) \tag{2.58}$$

[16] from the point of view of thread occupancy

Algorithm 86 Image Statistics Implementation Details

1: **procedure** STATISTICS($\texttt{sample}, \Lambda_1, \Lambda_0, \texttt{M}$)
2: MAXESPATCH($\texttt{sample}, \texttt{M}, \texttt{rows}(\texttt{sample}) \div \Lambda_1, \texttt{cols}(\texttt{sample})$)
3: $r \leftarrow \texttt{rows}(\texttt{sample}) \div \Lambda_1$
4: $s \leftarrow 1$
5: **while** $r > 1$ **do**
6: MAXESITERATEPATCH($\texttt{M}, r, \texttt{cols}(\texttt{sample}), s$)
7: $r \leftarrow r \div \Lambda_1$
8: $s \leftarrow s\Lambda_1$
9: **end while**
10: **end procedure**
11: **kernel** MAXESPATCH ($\texttt{sample}, \texttt{M}, \texttt{M}_{\texttt{row}}, \texttt{M}_{\texttt{col}}$)
Input: $\lambda_1, \lambda_0, \tau_{\lambda_1}, \tau_{\lambda_0}, v_1, v_0, \nu_1, \Lambda_0$
12: $\texttt{buff}[1024]$
13: $T \leftarrow \tau_{\lambda_1} \times \Lambda_0 + \tau_{\lambda_0}$
14: $\texttt{buff}[T] \leftarrow \texttt{sample}[\lambda_1 \times \texttt{cols}(\texttt{sample}) + \lambda_0]$
15: BARRIER(CLK_LOCAL_MEM_FENCE)
16: **if** $\tau_{\lambda_1} == 0$ **then**
17: $v \leftarrow \texttt{buff}[\tau_{\lambda_0}]$
18: **for** $\mu_{\texttt{row}}$ **in** $1 \ldots \Lambda_1$ **do**
19: $I \leftarrow \mu_{\texttt{row}} \times \Lambda_0 + \tau_{\lambda_0}$
20: **if** $v > \texttt{buff}[I]$ **then**
21: $v \leftarrow \texttt{buff}[I]$
22: **end if**
23: **end for**
24: $\texttt{M}[v_1 \times \texttt{cols}(\texttt{sample}) + \lambda_0] \leftarrow v$
25: **end if**
26: **end kernel**
27: **kernel** MAXESITERATEPATCH ($\texttt{M}, \texttt{M}_{\texttt{row}}, \texttt{M}_{\texttt{col}}, s$)
Input: $\lambda_1, \lambda_0, \tau_{\lambda_1}, \tau_{\lambda_0}, v_1, v_0, \nu_1, \Lambda_0$
28: $\texttt{buff}[1024]$
29: $T \leftarrow \tau_{\lambda_1} \times \Lambda_0 + \tau_{\lambda_0}$
30: $\texttt{buff}[T] \leftarrow \texttt{M}[s \cdot \lambda_1 \times \texttt{cols}(\texttt{sample}) + \lambda_0]$
31: BARRIER(CLK_LOCAL_MEM_FENCE)
32: **if** $\tau_{\lambda_1} == 0$ **then**
33: $v \leftarrow \texttt{buff}[\tau_{\lambda_0}]$
34: **for** $\mu_{\texttt{row}}$ **in** $1 \ldots \Lambda_1$ **do**
35: $I \leftarrow \mu_{\texttt{row}} \times \Lambda_0 + \tau_{\lambda_0}$
36: **if** $v > \texttt{buff}[I]$ **then**
37: $v \leftarrow \texttt{buff}[I]$
38: **end if**
39: **end for**
40: $\texttt{M}[sv_1 \times \texttt{cols}(\texttt{sample}) + \lambda_0] \leftarrow v$
41: **end if**
42: **end kernel**

Target-specific Optimization

The statistical extraction relies on the exact size and grouping if the threads during the computation. If it's possible during the repacking, the sample size can be adjusted to be optimal compared to the specific hardware.

2.5.4 Image Normalization

This benchmark performs image normalization by scaling and shifting. To optimize the GPU occupancy, the normalization is performed parallel on several images of the same size.

Input Data Structures

- $I_{0\ldots n-1}$ is the set of input images (all have the same size)
- `sample` is an image containing the repacked I images
- M is an array of records containing the scaling and the shifting constant

We repack the images I_q into a single image `sample` the following way:

$$\texttt{sample}(q,w) = I_w\left(q \div \text{cols}(I_w), q \mod \text{cols}(I_w)\right) \tag{2.59}$$

High-level Description The normalization can be described with the following formula:

$$\texttt{sample}(q,w) = \text{scale}(M[w]) \cdot \texttt{sample}(q,w) + \text{shift}(M[w]) \tag{2.60}$$

High-level Memory Access Pattern The memory access can be expressed the following way:

$$M_r = \texttt{sample} \tag{2.61}$$
$$M_w = \texttt{sample} \tag{2.62}$$

Low-level Implementation Details The algorithm is implemented in the following kernel 87.

Algorithm 87 Image Normalization Implementation Details

1: **kernel** MAXESPATCH (`sample`, `M`)
Input: $\rho, \kappa, \tau_\rho, \tau_\kappa, g_\rho, g_\kappa, d_\rho, d_\kappa$
2: `shift`[1024]
3: `scale`[1024]
4: **if** $\tau_\rho == 0$ **then**
5: `shift`$[\tau_\kappa] \leftarrow \text{shift}(\texttt{M}[\kappa])$
6: `scale`$[\tau_\kappa] \leftarrow \text{scale}(\texttt{M}[\kappa])$
7: **end if**
8: BARRIER(`CLK_LOCAL_MEM_FENCE`)
9: $I \leftarrow \rho \times \text{cols}(M) + \kappa$
10: `sample`$[I] \leftarrow$ `scale`$[\tau_\kappa] \cdot$ `sample`$[I] +$ `shift`$[\tau_\kappa]$
11: **end kernel**

Computational Dependencies The input data is read into a local buffer. As the buffers are reused this optimizes the global memory reads. The full dependency looks as follows:

$$\delta_{I_R} = \texttt{sample}(\cdot, \lambda_0) \xrightarrow{t} \texttt{shift}[\lambda_0] \bigwedge \tag{2.63}$$

$$\texttt{sample}(\cdot, \lambda_0) \xrightarrow{t} \texttt{scale}[\lambda_0] \bigwedge \tag{2.64}$$

$$\texttt{scale}[\lambda_0] \xrightarrow{t} M[\lambda_0] \bigwedge \tag{2.65}$$

$$\texttt{shift}[\lambda_0] \xrightarrow{t} M[\lambda_0] \tag{2.66}$$

Computational Complexity The basic algorithm has linear $O(\text{rows}(sample) \cdot \text{cols}(sample))$ complexity.

2.5.5 Image (Data) Repacking

In machine learning algorithms we often need to extract higher level information from our data. For the algorithms to work on multiple types of data, the input data has to be standardized. The *de facto* standard is to organize the input data into matrices. In the special case of face tracking, one sample consists of the collection of patches around a candidate point. We, therefore, need to repack the patches from around a candidate point into a single matrix line-by-line.

This benchmark operator performs a matrix reorganization based on predefined permutation rules.

High-level Description

The benchmark operates on an input image I. On this image we define patches P as

$$P_{x,y,s} = I\left(\left[x - \frac{s}{2}, x + \frac{s}{2}\right], \left[y - \frac{s}{2}, y + \frac{s}{2}\right]\right) \tag{2.67}$$

where x, y is the center of the patch and s is the size of the patch. We create a collection of tiles for every given patch, that fits into the image. These patches are then packed line-by-line into the result matrix. That is, for every patch P_i, we have

$$R_{i,q} = P_i[q] = (P_i)_{(q \div s, q \mod s)} \tag{2.68}$$

where R is the repacked image.

Computation The algorithm we implement in this benchmark uses the reduction for the repacking of the data. In the first part of the algorithm, we read the content of the patch into a local buffer and in the second part we feed the content of the buffer into the result matrix. As the patches and the matrix lines are non-continuous, we over-fill the buffer so we could fetch all the pixel which are necessary to finish the line of the matrix. The schema of the computation can be observed in fig. 2.5.5.

Implementation Pseudo-code is shown in algorithm 88.

Computational Dependences The computation does not generate any dependencies beyond the availability of the input.

Algorithm 88 Filling the Sample Matrix with Patches

1: **kernel** FILLPATCH (image, patchSize, result)
Input: $\lambda_1, \lambda_0, \tau_{\lambda_1}, \tau_{\lambda_0}, \nu_1, \nu_0, \Lambda_1, \Lambda_0$
Local: buff[1024]
2: $\text{row}_{\text{start}}(\text{result}) \leftarrow \nu_1 \cdot \Lambda_1$
3: $\text{col}_{\text{start}}(\text{result}) \leftarrow \nu_0 \cdot \Lambda_0$
4: $\text{row}_{\text{start}}(\text{image}) \leftarrow \text{row}_{\text{start}}(\text{result}) \mod \text{patchSize}$
5: $\text{col}_{\text{start}}(\text{image}) \leftarrow \text{col}_{\text{start}}(\text{result}) + \text{row}_{\text{start}}(\text{result}) \div \text{patchSize}$
6: $\text{row}_{\text{end}}(\text{image}) \leftarrow (\text{row}_{\text{start}}(\text{result}) + \Lambda_1 - 1) \div \text{patchSize}$
7: $\text{col}_{\text{end}}(\text{image}) \leftarrow (\text{col}_{\text{start}}(\text{result}) + \Lambda_0 - 1) + (\text{row}_{\text{start}}(\text{result}) + \Lambda_1 - 1) \mod \text{patchSize}$
8: $\tau \leftarrow \tau_{\lambda_1} \cdot \Lambda_0 + \tau_{\lambda_0}$
9: $\text{row}_{\text{current}}(\text{image}) \leftarrow \text{row}_{\text{start}}(\text{image}) + (\text{col}_{\text{start}}(\text{image}) + \tau) \div \text{cols}(\text{image})$
10: $\text{col}_{\text{current}}(\text{image}) \leftarrow (\text{col}_{\text{start}}(\text{image}) + \tau) \div \text{cols}(\text{image})$
11: $\tau_{\text{current}}(\text{image}) \leftarrow \text{row}_{\text{current}}(\text{image}) \cdot \text{cols}(\text{image}) + \text{col}_{\text{current}}(\text{image})$
12: step $\leftarrow$ **false**
13: **if** $\text{row}_{\text{current}}(\text{image}) = \text{row}_{\text{end}}(\text{image}) \bigwedge \text{col}_{\text{current}}(\text{image}) > \text{col}_{\text{end}}(\text{image})$ **then**
14: step $\leftarrow$ **true**
15: **end if**
16: **if** $\text{row}_{\text{current}}(\text{image}) > \text{row}_{\text{end}}(\text{image})$ **then**
17: step $\leftarrow$ **true**
18: **end if**
19: **if** !step **then**
20: $\text{buff}[\tau] \leftarrow \text{image}[\tau_{\text{current}}(\text{image})]$
21: **end if**
22: BARRIER(CLK_LOCAL_MEM_FENCE)
23: $\text{row}_{\text{current}}(\text{result}) \leftarrow \text{row}_{\text{start}}(\text{result}) + \tau_{\lambda_1}$
24: $\text{col}_{\text{current}}(\text{result}) \leftarrow \text{col}_{\text{start}}(\text{result}) + \tau_{\lambda_0}$
25: $\tau_{\text{current}}(\text{result}) \leftarrow \text{row}_{\text{current}}(\text{result}) \cdot \text{cols}(\text{result}) + \text{col}_{\text{current}}(\text{result})$
26: $\lambda_1(\text{image}) \leftarrow \text{row}_{\text{current}}(\text{result}) \div \text{patchSize}$
27: $\lambda_0(\text{image}) \leftarrow \text{col}_{\text{current}}(\text{result}) + \text{row}_{\text{current}}(\text{result}) \mod \text{patchSize}$
28: $\tau_{\text{buff}} = (\text{row}_{\text{image}}) - \text{row}_{\text{start}}(\text{image}) \cdot \text{cols} + (\lambda_0(\text{image}) - \text{col}_{\text{start}}(\text{image}))$
29: **if** $\text{row}_{\text{current}}(\text{image}) > \text{rows}(\text{result})$ **then**
30: **return**
31: **end if**
32: $\text{result}[\tau_{\text{current}}(\text{result})] \leftarrow \text{buff}[\tau_{\text{buff}}]$
33: **end kernel**

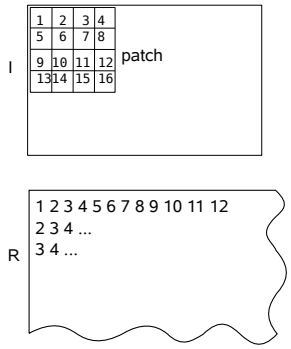

Figure 2.6: The schema of the reorganization of the patch image for the MLP training. All the input patches will be repacked into a single R matrix.

Memory Access Mapping and Complexity All of the global memory accesses are coalesced and we assure reading the input and the output only once. The access map is as follows:

$$G_r = \{I\} \tag{2.69}$$
$$G_w = \{R\} \tag{2.70}$$

Computational Complexity The basic algorithm has $O(\texttt{rows}(I) \times \texttt{cols}(I) \times \texttt{patchSize})$ complexity.

Optimization Opportunities

The key of the optimization is the use of local memory for non-coalesced reads and writes. Therefore it is crucial to exploit this local buffering programming paradigm in the implementation.

2.6 BasemarkCL

2.6.1 Histogram equalization

Description

This benchmark performs a histogram equalization to an image. All color channels are equalized independently.

Threads in a local work-group all read a single color value of the input image and increment the corresponding value in the local histogram array using atomic operations. After this the local histogram array is summed into the global histogram array by again avoiding memory race condition by using atomic operations. Cumulative sum is applied to the global histogram. On the final kernel threads read a single pixel value from input image, use the cumulative histogram value on that pixel value to calculate an output color and writes it into output image.

Data structures

- W is the width of the image.
- H is the height of the image.
- HR histogram array for red channel
- HG histogram array for green channel
- HB histogram array for blue channel
- HA histogram array for alpha channel
- `img` is the 2 dimensional image.

Iteration space and dependencies

$$I = \{(i, j) \quad : \quad 0 \leq i < W, 0 \leq j < H\}$$

Memory access mapping

A pixel is read from the image and the corresponding indices in the histogram arrays are incremented. Histogram calculation: for $(i, j) \in I_D$:

$$\begin{aligned} M_r(i,j) &= \{\texttt{img}_{i,j}\} \\ M_{rw}(i,j) &= \{\texttt{HR}_{img_{i,j}}\} \\ M_{rw}(i,j) &= \{\texttt{HG}_{img_{i,j}}\} \\ M_{rw}(i,j) &= \{\texttt{HB}_{img_{i,j}}\} \\ M_{rw}(i,j) &= \{\texttt{HA}_{img_{i,j}}\} \end{aligned}$$

Device specific data structures

- histogram is an array of 1024 unsigned integers
- local_histogram is an array of 1024 unsigned integers in local memory
- img is the 2d RGBA image with 8bits per color channel

Device code abstraction

Input: `image2d img`
Input/Output: `global uint histogram[]`
Local: `local uint local_histogram[]`
Local: `uint4 color`

kernel GENERATEHISTOGRAM (`img`,`histogram`)
  for `x` **in** $(\lambda_0 + \lambda_1 \times \Lambda_0) : 1024 : \Lambda_0 \times \Lambda_1$ **do**
    `local_histogram`[`x`] $\leftarrow 0$
  end for
  BARRIER(CLK_LOCAL_MEM_FENCE)

```
        color ← READ_IMAGE(img, γ0, γ1)
        ATOM_INC(local_histogram[color.x])
        ATOM_INC(local_histogram[color.y] + 256)
        ATOM_INC(local_histogram[color.z] + 256 × 2)
        ATOM_INC(local_histogram[color.w] + 256 × 3)
        BARRIER(CLK_LOCAL_MEM_FENCE)
        for x in (λ0 + λ1 × Λ0) : 1024 : (Λ0 × Λ1) do
            ATOM_ADD(global_histogram[x], local_histogram[x])
        end for
    end kernel

Input: image2d src
Input: uint cumulative_histogram[]
Output: image2d dst
Local: uint4 color
    kernel APPLYHISTOGRAM (src, dst, cumulative_histogram)
        color ← READ_IMAGE(γ0, γ1)
        scaling_factor ← 255 / (W×H)
        color.x ← cumulative_histogram[color.x] × scaling_factor
        color.y ← cumulative_histogram[color.y + 256] × scaling_factor
        color.z ← cumulative_histogram[color.y + 512] × scaling_factor
        color.w ← cumulative_histogram[color.y + 768] × scaling_factor
        WRITE_IMAGE(γ0, γ1, color)
    end kernel

Input/Output: global uint histogram[]
Local: uint csum
    kernel GENERATECUMULATIVEDISTRIBUTION (histogram)
        csum ← 0
        for x in 0 : 255 do
            csum ← csum + histogram[x + γ0 × 255]
            histogram[x + γ0 × 256] ← csum
        end for
    end kernel

Input/Output: global uint histogram[]
    kernel ZERODISTRIBUTION (histogram)
        for x in 0 : M do
            histogram[x + γ0 × 256] ← 0
        end for
    end kernel
```

Host code abstraction

```
  procedure HISTOGRAM(inputImage, outputImage, histogram)
      SETKERNELARGUMENTS(zeroDistribution, histogram)
      ENQUEUEKERNEL(zeroDistribution, 4)
```

SETKERNELARGUMENTS(generateHistogram, inputImage, histogram)
ENQUEUEKERNEL(generateHistogram, W, H)
SETKERNELARGUMENTS(generateCumulativeDistribution, histogram)
ENQUEUEKERNEL(generateCumulativeDistribution, 4)
SETKERNELARGUMENTS(applyHistogram, inputImage, outputImage, histogram)
ENQUEUEKERNEL(applyHistogram, W, H)
end procedure

Input datasets

This benchmark operates on a set of 5 2048x2048 input images with 4 8-bit color channels (RGBA).

Partitioned iteration space

$$P = \{(\gamma_0, \gamma_1, \lambda_0, \lambda_1, i) \quad : \quad 0 \leq \gamma_0 < \Gamma_0 = 2048, 0 \leq \gamma_1 < \Gamma_1 = 2048, 0 \leq i < B = 256\}$$

The device is allowed to choose it's local workgroup size therefore the local id:s can run between any limit that divides the global size exactly.

Device memory access mapping

- Histogram calculation

$$\begin{aligned} G_r(\gamma_0, \gamma_1) &= \{\texttt{img}_{\gamma_0,\gamma_1}\} \\ L_{rw}(\gamma_0, \gamma_1) &= \{\texttt{l\_hist}[\texttt{img}_{\gamma_0,\gamma_1}]\} \\ G_{rw}(i) &= = \{\texttt{histogram}[i]\} \end{aligned}$$

- Apply histogram:

$$\begin{aligned} G_{rw}(\gamma_0, \gamma_1) &= \{\texttt{img}_{\gamma_0,\gamma_1}\} \\ G_r(\gamma_0, \gamma_1) &= \{\texttt{histogram}[\texttt{img}_{\gamma_0,\gamma_1}]\} \end{aligned}$$

- Zero distribution: $0 \leq \gamma 0 < 4$

$$G_w(\gamma_0, i) = \{\texttt{histogram}[i + \gamma_0 \times B]\}$$

- Generate cumulative histogram: $0 \leq \gamma 0 < 4$

$$G_{rw}(\gamma_0, i) = \{\texttt{histogram}[i + \gamma_0 \times B]\}$$

Performance metrics

The performance is evaluated by taking the geometric mean of inverse frame times thus giving the geometric average of frames per second.

Validation mechanism

The benchmark has no built-in validation mechanism. The validation is visual only.

2.6.2 SPH fluid

High-level description

The benchmark performs a smoothed particle hydrodynamics fluid simulation [1].

Data structures

The benchmark operates on particles in 3D space. Density of the fluid on a location of a particle depends on the amount particles within a distance `h` from the particle under consideration. Viscous and pressure dependent forces acting upon a particle are calculated by examining the neighboring particles within distance `h`. The space is divided into a regular grid of voxels where particles reside in order to speed up the neighboring particle search.

The benchmark operates on following data structures and has following constants:

- `M` is the amount of particles.
- `V` is the amount of voxels in the three dimensional space.
- `keys` is an array containing M tuples of values containing the index of a voxel and the index of a particle residing in said voxel.
- `voxelParticles` is an array of V scalar values containing the smallest particle index of all particles in the voxel.
- `sortedVelocity` is an array of M vectors containing the velocity of a particle in sorted order.
- `sortedDensity` is an array of M scalar values containing the density at the particles position in sorted order.
- `position` is an array of M vectors containing the position of each particle.
- `velocity` is an array of M vectors containing the velocity of each particle.
- `acceleration` is an array of M vectors containing the acceleration applied for each particle.

Computation

Voxel index for each particle is calculated for each particle into the array `keys` using `position` such that $\texttt{keys}[\gamma_0] = (\texttt{VoxelIndexAt}(\texttt{position}[\gamma_0]), \gamma_0)$. The `keys` array is then sorted so that the smallest voxel index resides at the start of the array. After sorting the arrays `sortedPosition` and `sortedVelocity` are updated so that $\texttt{sortedPosition}[\gamma_0] = \texttt{position}[\texttt{keys}[\gamma_0].\text{y}]$. Using the sorted `keys` array the `voxelParticles` is updated so that $\texttt{voxelParticles}_\text{i}$ contains the index of the first particle residing in said voxel so that the `sortedVelocity` and `sortedPosition` can be indexed using that information.

Density and acceleration of each particle is then calculated by spawning a thread for each particle, looking up the voxel index of said particle using `sortedPosition` array and then looping trough neighboring voxels and the voxel the particle currently resides in. Neighboring particles in each voxel are found by looking the index of the first particle residing in said voxel by using the `voxelParticles` array and then increasing the index until $\texttt{keys}_{\gamma}$ is different from the previous iteration, after which a second voxel is examined. Arrays `sortedPosition`, `sortedDensity` and `sortedVelocity` are accessed using the index. At the end acceleration for a particle is written into `acceleration` array by looking up the required index from the `keys` array.

As a last step a simple Eulerian integration is used to update the data in `position` and `velocity` arrays by using the information in `acceleration` array. If particle crosses the boundary of the world it bounces inelastically and the position gets clamped to the edge.

Iteration space

Memory access mapping

- updateKeys: for $(i) \in I_D$:

$$\begin{aligned} M_r(i) &= \{\texttt{position}_i\} \\ M_w(i) &= \{\texttt{keys}_i\} \end{aligned}$$

- sortPostProcess: for $(i) \in I_D$:

$$\begin{aligned} M_r(i) &= \{\texttt{position}_i\} \\ M_w(i) &= \{\texttt{sortedPosition}_i, \texttt{sortedVelocity}_i\} \end{aligned}$$

Device code abstraction

Input: `int4 position`
Input: `uint4 voxelCount`
function INDEXFROMPOSITION(`position`, `voxelCount`) ▷ Returns the voxel on which a particle resides based on the current physical location
    return `voxelCount.x` × `voxelCount.y` × `position.z` × `voxelCount.x` × `position.y` × `position.x`
end function

Input: `float4 position`
Input: `uint4 voxelCount`
Input: `float h`
function GETPOSCOORD (`position`, `voxelCount`)
    `minx` ← FLOOR($\frac{\texttt{position.x}}{\texttt{h}}$)
    `miny` ← FLOOR($\frac{\texttt{position.y}}{\texttt{h}}$)
    `minz` ← FLOOR($\frac{\texttt{position.z}}{\texttt{h}}$)
    `posCoord.x` ← CLAMP(`minx`, 0, `voxelCount.x` − 1)
    `posCoord.y` ← CLAMP(`miny`, 0, `voxelCount.y` − 1)
    `posCoord.z` ← CLAMP(`minz`, 0, `voxelCount.z` − 1)
    `posCoord.w` ← 0

```
12:     return posCoord
13: end function

Input: global uint2 keys[]
Input: global float4 sortedPosition[]
Input: global int voxelParticles[]
Input: uint4 voxelCount
Input: input float h
Input: float m
Output: float sortedDensity[]
Local: totalDensity
Local: ownVoxel
Local: ownPosition
14: kernel CALCULATEDENSITY (keys, sortedPosition, voxelParticles, voxelCount, h, m)
15:     ownVoxel ← GETPOSCOORD(sortedPosition[γ0], voxelCount, h)
16:     ownVoxel ← CLAMP(ownVoxel, (1,1,1,1), (voxelCount.x − 2, voxelCount.y − 2, voxelCount.z −
    2, 0))
17:     ownPosition ← sortedPosition[γ0]
18:     totalDensity ← 0
19:     for z in −1 : 1 do
20:         for y in −1 : 1 do
21:             for x in −1 : 1 do
22:                 pid ← voxelParticles[INDEXFROMPOSITION(ownVoxel + (x, y, z, 0), voxelCount)]
23:                 while pid < Γ0 ∧ keys[pid].x = index do
24:                     distance ← ownPosition − sortedPosition[pid]
25:                     if distance ≤ h then
26:                         totalDensity ← totalDensity+ W(distance, h)
27:                     end if
28:                     pid ← pid + 1
29:                 end while
30:             end for
31:         end for
32:     end for
33:     sortedDensity[γ0] ← totalDensity × m
34: end kernel

Input: global uint2 keys[]
Input: global float4 sortedPosition[]
Input: global int voxelParticles[]
Input: global float sortedDensity[]
Input: global float4 sortedVelocity[]
Input: uint4 voxelCount
Input: float h
Input: float m
Input: float u
Output: global float acceleration[]
Local: float4 pressureAcceleration
```

Local: `float4 viscousAcceleration`
Local: `int4 ownVoxel`
Local: `float4 ownPosition`
kernel CALCULATEACCELERATION (keys, sortedPosition, voxelParticles, sortedDenisty, sortedVeloc
    ownVoxel ← GETPOSCOORD(sortedPosition[γ_0], voxelCount, h)
    ownVoxel ← CLAMP(ownVoxel, (1,1,1,1), (voxelCount.x − 2, voxelCount.y − 2, voxelCount.z − 2, 0))
    ownPosition ← sortedPosition[γ_0]
    pressureAcceleration ← 0
    viscousAcceleration ← 0
    for z **in** −1 : 1 **do**
        for y **in** −1 : 1 **do**
            for x **in** −1 : 1 **do**
                pid ← voxelParticles[INDEXFROMPOSITION(ownVoxel + (x, y, z, 0), voxelCount)]
                while pid < $\Gamma_0 \wedge$ keys[pid].x = index **do**
                    distance ← ownPosition − sortedPosition[pid]
                    if distance ≤ h **then** ▷ W2 and W3 refer simply to purely mathematical manipulations on the values supplied there.
                        pressureAcceleration ← pressureAcceleration + W2(sortedDensity[pid], distar
                        viscousAcceleration ← viscousAcceleration + W3(sortedVelocity[pid], distanc
                    end if
                    pid ← pid + 1
                end while
            end for
        end for
    end for
    acceleration[keys[γ_0].y] ← pressureAcceleration + viscousAcceleration + gravity
end kernel

Input/Output: `global float4 position[]`
Input/Output: `global float4 velocity[]`
Input: `global float4 acceleration[]`
Input: `uint4 voxelCount`
Input: `float h`
Input: `float dt`
kernel INTEGRATE (position, velocity, acceleration, voxelCount, h, dt)
    newVelocity ← velocity[γ_0] + acceleration[γ_0] × dt
    position[γ_0] ← position[γ_0] + newVelocity × dt
    if position[γ_0].x < 0 ∨ position[γ_0].x > (voxelCount.x − 1) × h **then**
        position[γ_0].x ←CLAMP(position[γ_0].x, 0, (voxelCount.x − 1) × h)
        newVelocity.x ← −newVelocity.x
    end if
    if position[γ_0].y < 0 ∨ position[γ_0].y > (voxelCount.y − 1) × h **then**
        position[γ_0].y ←CLAMP(position[γ_0].y, 0, (voxelCount.y − 1) × h)
        newVelocity.y ← −newVelocity.y
    end if

```
69:     if position[γ0].z < 0 ∨ position[γ0].z > (voxelCount.z − 1) × h then
70:         position[γ0].z ←CLAMP(position[γ0].z, 0, (voxelCount.z − 1) × h)
71:         newVelocity.z ← −newVelocity.z
72:     end if
73:     velocity[γ0] ← newVelocity
74: end kernel

Input: global float4 position[]
Input: uint4 voxelCount
Input: float h
Output: global uint2 keys[]
75: kernel UPDATEKEYS (position, voxelCount, h, keys)
76:     posCoord ←GETPOSCOORD(position[γ0], voxelCount, h)
77:     keys[γ0] ← (voxelCount.x × voxelCount.y × posCoord.z + voxelCount.x × posCoord.y +
    posCoord.x, γ0)
78: end kernel

Input/Output: global uint2 keys[]
Input: uint stage
Input: uint pass
79: kernel BITONICSORT (keys, stage, pass)
80:     pairDistance ← 2^(stage−pass)
81:     leftID ← (γ0 & (pairDistance − 1)) | ((γ0 & ∼(pairDistance − 1)) × 2)
82:     direction ← ((γ0 ≫ stage) & 1) == 1?0 : 1
83:     rightID ← leftID + pairDistance
84:     left ← keys[leftID]
85:     right ← keys[rightID]
86:     larger ← left.x > right.x?left : right
87:     smaller ← left.x > right.x?right : left
88:     keys[leftID] ← direction?smaller : larger
89:     keys[rightID] ← direction?larger : smaller
90: end kernel

Input: global uint2 keys[]
Input: global float4 position[]
Input: global float4 velocity[]
Output: global float4 sortedPosition[]
Output: global float4 sortedVelocity[]
91: kernel SORTPOSTPROCESS (keys, position, velocity, sortedPosition, sortedVelocity)
92:     sortedPosition[γ0] ← position[keys[γ0].y]
93:     sortedVelocity[γ0] ← velocity[keys[γ0].y]
94: end kernel
Output: global int voxelParticles[]
95: kernel CLEARVOXELS (voxelParticles)
96:     voxelParticles[γ0] ← −1
97: end kernel
Input: global uint2 keys[]
```

Output: `global int voxelParticles[]`

```
98:  kernel SETVOXELS (keys, voxelParticles)
99:      if γ0 = 0 then
100:         voxelParticles[keys[0].x] ← 0
          return
101:     end if
102:     if keys[γ0 − 1].x ≠ keys[γ0].x then
103:         voxelParticles[keys[γ0].x] ← γ0
104:     end if
105: end kernel
```

Host code abstraction

```
procedure INITIALIZE(keys, voxelParticles, sortedVelocity, sortedPosition,
position, velocity, acceleration )        ▷ Run only once at start of the program
    SETKERNELARGUMENTS(updateKeys, keys, position, voxelCount, h)
    SETKERNELARGUMENTS(sortPostProcess, keys, position, velocity, sortedPosition,
sortedVelocity)
    SETKERNELARGUMENTS(clearVoxels, voxelParticles)
    SETKERNELARGUMENTS(setVoxels, voxelParticles, keys)
    SETKERNELARGUMENTS(calculateDensity, keys, sortedPosition, voxelParticles,
sortedDensity, voxelCount, h, m)
    SETKERNELARGUMENTS(calculateAcceleration, keys, sortedPosiition,
voxelParticles, sortedDensity, sortedVelocity, acceleration, voxelCount,
h, m, u)
    SETKERNELARGUMENTS(integrate, position, velocity, acceleration,
voxelCount, h, dt)
    SETKERNELARGUMENTS(clearVelocityAcceleration, velocity, acceleration)
    ENQUEUEKERNEL(clearVelocityAcceleration, M)
    ENQUEUEKERNEL(updateKeys, M)
    SORT(keys)
    ENQUEUEKERNEL(sortPostProcess, M)
    ENQUEUEKERNEL(clearVoxels, V)
    ENQUEUEKERNEL(setVoxels, M)
end procedure
procedure SORT(keys)
    for stage  in 0:log2(M)-1 do
        for pass  in 0:stage do
            SETKERNELARGUMENTS(bitonicsort, keys, stage, pass)
            ENQUEUEKERNEL(bitonicSort, M/2)
        end for
    end for
end procedure

procedure UPDATESCENE                                  ▷ Called once per frame
    ENQUEUEKERNEL(updateKeys, M)
    SORT(keys)
```

```
    ENQUEUEKERNEL(sortPostProcess, M)
    ENQUEUEKERNEL(clearVoxels, V)
    ENQUEUEKERNEL(setVoxels, M)
    ENQUEUEKERNEL(calculateDensity, M)
    ENQUEUEKERNEL(calculateAcceleration, M)
    ENQUEUEKERNEL(integrate, M)
end procedure
```

Partitioned iteration space

Device memory access mapping

Performance metrics

The performance is evaluated by taking the geometric mean of inverse frame times thus giving the geometric average of frames per second.

Validation mechanism

The benchmark does not contain any internal validation mechanisms. The validation is visual only.

Bibliography

[1] Alan Heirich. Smoothed particle hydrodynamics, 2010.

[2] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[4] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[6] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[7] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, number AIAA 2009-4001 in AIAA Conference Proceedings, June 2009.

[8] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter". The Scalable Heterogeneous Computing (SHOC) benchmark suite. In D. R. Kaeli and M. Leeser, editors, *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, volume 425 of *ACM International Conference Proceeding Series*, pages 63–74. ACM, 2010.

[9] F. Vázquez and E. M. Garzón and J. A. Martínez and J. J. Fernández. The sparse matrix vector product on GPUs. Technical report, University of Almeria, 2009.

[10] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron. Parallelization of particle filter algorithms. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 139–149, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] P. Harish and P. Narayanan. Designing Perspectively-Correct Multiplanar Displays. *IEEE Transactions on Visualization and Computer Graphics*, 99:1–1, 2012.

[12] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2009.

[13] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotspot: acompact thermal modeling methodology for early-stage VLSI design. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(5):501–513, May 2006.

[14] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[15] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[16] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. Technical report, IBM, 2008.

[17] N. Ray and S. T. Acton. Motion gradient vector flow: an external force for tracking rolling leukocytes with shape and size constrained active contours. *IEEE Trans. Med. Imaging*, 23(12):1466–1478, 2004.

[18] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[19] F. H. Streitz, J. N. Glosli, M. V. Patel, B. Chan, R. K. Yates, B. R. de Supinski, J. Sexton, and A. Gunnels. 100+ TFlop Solidification Simulations on BlueGene/L. In *Proceedings of the 2005 Supercomputing Conference (SC 05)*, Seattle, WA, 2005.

[20] L. G. Szafaryn, T. Gamblin, B. deSupinski, and K. Skadron. Experiences with Achieving Portability across Heterogeneous Architectures. In *WOLFHPC workshop at 25th International Conference on Supercomputing*, Tucson, AZ, 2010.

[21] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences accelerating matlab systems biology applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits (BiC)*, June 2009.

[22] Y. Yu and S. Acton. Speckle reducing anisotropic diffusion. In *IEEE Transactions on Image Processing*, pages 1260–1270, 2002.