SEVENTH FRAMEWORK PROGRAMME

CARP

D2.1: State-of-the-Art Survey

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]:	PU (Public)
Document Code[3]:	CARP-ICL-RP-004
Version:	v1.15
Editor (Partner):	J. Ketema (ICL)
Contributors:	ARM, ENS, ICL, MONO, REAL, RIGHT, RWTHA, UT
Workpackage(s):	WP2
Reviewer(s):	A. Cohen (ENS), J. Dubreil (MONO), M. Huisman (UT)
Due Date:	31 May 2012
Submission Date:	31 May 2012
Number of Pages:	97

[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.

Copyright © 2012 by the CARP Consortium.

D2.1: State-of-the-Art Survey

J. Ketema[3] (Editor)
D. Mansell[1], T. Glauert[1], A. Lokhmotov[1] (Chapter 3)
A. Cohen[2] (Chapter 4)
A. Betts[3] C. Dehnert[7] D. Distefano[4] F. Gretz[7] J.-P. Katoen[7]
A. Lokhmotov[1] (Chapter 5)
A.F. Donaldson[3] D. Distefano[4] M. Huisman[8] C. Jansen[7]
J. Ketema[3] M. Mihelčić[8] (Chapter 6)
E. Hajiyev[5] D. Takacs[5] T. Virolainen[6] (Chapter 7)

[1]ARM [2]ENS [3]ICL [4]MONO [5]REAL [6]RIGHT [7]RWTHA [8]UT

REVISION HISTORY

Date	Version	Author	Modification
2 May 2012	1.0	J. Ketema (ICL)	Initial setup of LaTeX document
10 May 2012	1.1	J. Ketema (ICL)	Added Chapter 4, received from Albert Cohen (ENS), and Chapter 6
14 May 2012	1.2	J. Ketema (ICL)	Added Chapter 3, received from Anton Lokhmotov (ARM)
18 May 2012	1.3	J. Ketema (ICL)	Added Chapter 5, received from Joost-Pieter Katoen (RTWTHA)
22 May 2012	1.4	A.F. Donaldson (ICL)	Edits to abstract, introduction and summary; revisions to Chapter 6
23 May 2012	1.5	E. Hajiyev (REAL)	Added performance metrics in image processing to Chapter 7
24 May 2012	1.6	A. Betts (ICL)	Added text on cost analysis for GPU programs to Chapter 5
27 May 2012	1.7	C. Dehnert (RWTHA)	Revisions to Chapter 5
27 May 2012	1.8	A. Cohen (ENS)	Revisions to Chapter 4
28 May 2012	1.9	D. Takacs (REAL)	Added text on GPU acceleration of image processing to Chapter 7
28 May 2012	1.10	M. Huisman (UT)	Revisions to Chapter 6
30 May 2012	1.11	A. Lokhmotov (ARM)	Revisions to Chapters 3 and 5
30 May 2012	1.12	J. Ketema (ICL)	Added text on OpenCL benchmarking provided by T. Virolainen (RIGHT) to Chapter 7
30 May 2012	1.13	J.P. Katoen (RWTHA)	Revisions to Chapter 5
30 May 2012	1.14	A. Lokhmotov (ARM)	Revisions to Chapter 5
31 May 2012	1.15	A.F Donaldson (ICL)	Minor revisions to all chapters.

APPROVALS

Role	**Name**	**Partner**	**Date**
Workpackage Leader	A. Lokhmotov	ARM	31 May 2012
Coordinator	A.F. Donaldson	ICL	31 May 2012

Contents

1 Executive Summary

Many-core accelerator processors, in particular general-purpose graphics processing units (GPGPUs) are the current architecture of choice for achieving maximum performance within problem domains exhibit a high degree of parallelism. Although great speed-ups can be achieved, programming accelerator devices is currently a dark art, as the programming models in use expose the programmer to many low-level, architecture-specific details. As a result, the portability of programs written for accelerator devices is currently limited.

In the coming years, the CARP project aims to address the aforementioned accelerator programming issues. To be able to do so, it is important to be familiar with the current state-of-the-art in the field of accelerator programming. In particular, it is important to know the current state-of-the-art in: (1) programming models in use for accelerator programming, (2) compiler technologies for accelerator programming, (3) methods to gage the execution costs of accelerator programs (e.g., costs in terms of energy consumption, as accelerators are often embedded in mobile devices), and (4) methods to verify the correct working of accelerator programs. In addition, as the methods and tools developed within the CARP project will need to be validated, it is also important to know (5) the state-of-the-art in a number of application areas for accelerator programming.

This deliverable describes the state-of-the-art in all of the above areas, where the state-of-the-art with regards to applications is restricted to eye tracking and benchmarking (as appropriate given the industrial partners of the project). Because the field of accelerator programming is currently in its infancy, the state-of-the-art is limited. For this reason, the state-of-the-art in the wider field of multi-threaded programming is reported in areas which are relevant to CARP, and which we believe will soon influence accelerator programming.

Briefly, the state-of-the-art in accelerator programming is as follows:

1. Programming models for accelerator devices are close to the actual hardware, which complicates the programming of these devices.
2. Initial efforts have been made to apply advanced compilation techniques, including polyhedral compilation, to automate the generation of efficient accelerator code for loops. Currently, only simple loops exhibiting regular access patterns can be dealt with.
3. Although general quantitative cost analysis methods are available, these have thus far not been applied specifically to accelerated systems.
4. Preliminary static verification methods for GPU kernels have been designed. The main limitations of these methods are: limited support for loops and procedures; rudimentary handling of pointers; limited scalability to large numbers of threads.
5. Image processing has been successfully accelerated using GPUs, but only using low-level programming models such as CUDA and OpenCL.
6. Several collections of OpenCL benchmarks are publicly available for evaluation.

2 Introduction

We describe the state-of-the-art in accelerator programming, with a particular emphasis on general purpose GPU (GPGPU) programming. Our survey is focused according to the aims of the CARP project.

The survey is structured via a chapter targeting each technical work package of the project, thus to a large extent the chapters can be read relatively independently of one another. However, due to strong links between the project's work packages, there are frequent cross-references between chapters.

As already remarked in the *Executive Summary*, accelerator programming is a field that is currently in its infancy, hence the state-of-the-art is limited in some areas. We discuss the state-of-the-art in the wider field of multi-threaded programming in cases where we believe this state-of-the-art is relevant to the CARP project, and likely to find application to accelerator programming in the near future.

The structure of this deliverable is as follows:

- **Parallel Programming Models** (Chapter 3). We describe the current state-of-the-art in models for multi-threaded programming. CARP involves the design of a novel intermediate language for accelerator programming, *PENCIL*. PENCIL will lie in the landscape of existing programming models for parallel systems, thus up-to-date knowledge of this landscape is pivotal in the design of PENCIL.
- **Compiler Optimization and Code Generation** (Chapter 4). The state-of-the-art in compiler optimization and code generation for accelerator programming is discussed. In particular, we discuss the current state in applying polyhedral compilation techniques: these are among the most advanced compiler optimization techniques to date, and will be applied and extended during the project.
- **Quantitative Cost Analysis** (Chapter 5). We discuss the state-of-the-art in quantitative methods that can be used to gage the cost of system execution, in particular, cost in terms of execution time, memory usage, and energy consumption.
- **Static Verification** (Chapter 6). We discuss the state-of-the-art in static verification of accelerator programs using theorem proving and model checking. We also provide an overview of verification techniques for pointer-manipulating programs, and techniques for the analysis of traditional concurrent software, which we plan to apply to accelerator software during the project.
- **Applications and Validation Metrics** (Chapter 7). The techniques developed during CARP will be evaluated with respect to two primary application domains: eye-tracking software, and accelerator benchmarks. We discuss the state-of-the-art in these areas with respect to GPGPUs.

3 Parallel Programming Models

3.1 Multiprocessor Architecture

Since 1945, programming models for single processor systems have evolved with the assumption that the underlying hardware implements the Von Neumann architecture, a single thread executing instructions in program order. Improved semiconductor manufacturing and advanced microarchitectural techniques have increased performance by many orders of magnitude while presenting an essentially unchanged model to the programmer. More recently, the rate of increase has slowed, as both of these avenues of improvement have neared their limit. Therefore, other ways of improving computing performance are sought.

An obvious way to improve performance is to use multiple discrete processors rather than just a single one. Major performance benefits can be realized this way, but as the number of parallel threads of execution (and thus performance) increases, the architectural model becomes more and more dissimilar to the Von Neumann model. Therefore, new programming models are needed to allow programmers to exploit the large number of threads becoming available.

This chapter explores the various programming models which have sprung up to fill this gap. In order to understand the constraints these programming models are working with, a brief discussion of multiprocessor architectures is also included.

3.1.1 Homogeneous and Heterogeneous Systems

Any system including more than one processing element can be broadly categorized as either homogeneous or heterogeneous.

Homogeneous Systems In a homogeneous system, all processing elements are identical and behave almost entirely as peers. (There may be a ”boot” element that has a special role at system startup, but once the system is running it is no longer special).

This implies that the processing elements must be self-supervising; the OS runs on all processors and there is no intrinsic master/slave relationship between processors. It is possible for an OS to use the different processors in an asymmetric way (e.g. making all scheduling decisions or handling all peripheral interrupts on a single processor), but the OS must have a presence on each processor as there is no provision for external supervision.

As all the processors are identical there is no need (or indeed opportunity) for software to adapt to different processor types; software just needs to be able to cope with running on several processors concurrently.

Heterogeneous Systems A heterogeneous system contains more than one type of processor. The different processor types might have different execution models, memory models or instruction sets. A heterogeneous system allows each processor type to be specialized to run a particular task or class of tasks. Usually there is a master/slave relationship where one processor or group of processors is responsible for supervising the activities of the others. This relieves the other processors of the need to run supervisory code such as an OS.

Producing software for heterogeneous systems is a greater challenge; software must adapt at some level to the different processor types and arrange for different processors to run tasks that are suited to their specializations.

3.1.2 Threading Models

In describing processors that support multiple threads of execution, various terms are used to specify how the individual threads can be controlled by the programmer.

Both MPMD and SPMD (described below) are cases of MIMD (Multiple Instruction Multiple Data) execution, where each thread notionally executes its own instruction stream.

Single Instruction Multiple Data (SIMD) SIMD is not a multi-threading model at all, but a way to extract more performance from a single thread. SIMD is an architectural feature that allows a single processor to process a given data set using fewer total operations. A processor supporting SIMD includes registers that can contain several discrete data values, and instructions that can simultaneously perform the same operation on all data values packed into registers. The number of values that can be stored and operated on in this way is termed the ”SIMD width” (in modern processors, typically four or eight 32-bit values).

The maximum possible performance increases by a factor approaching the SIMD width (typically 4 or 8), but this improvement can only be seen when the workload is able to use the SIMD features effectively. For maximum performance every operation in the algorithm must be converted to use the SIMD operations, otherwise the non-SIMD operations become the bottleneck, which requires regularity in the algorithm. As the SIMD width increases, the amount of regularity required increases too; this limits the usefulness of increasing the width.

In practice many algorithms do see a significant performance improvement when SIMD features are used, but there is relatively little scope for further improvement as increasing SIMD width has diminishing returns.

Multiple Program Multiple Data (MPMD) In an MPMD model, each thread is independent of the others. Each thread executes its own program with its own data, so that a given thread can be doing a different task and processing different data from any other thread.

MPMD systems support a fixed number of hardware threads. A larger number of software threads can be supported by switching each hardware thread amongst several software threads in the OS. This makes thread creation relatively expensive, as each thread must be individually created by software. This places constraints on how these software created threads can be used by programmers, as the useful work done by a thread has to justify the cost of creating it. It also requires at least some component of the OS to run on all processors in order to handle thread switching. (It must be an OS component as the virtual memory settings may need to be reconfigured, which requires OS level privilege). This places limits on the maximum sensible number of threads since the OS has to manage them all.

Traditional threading models, such as POSIX threads [27] (see Section 3.4.2) are geared towards the MPMD paradigm.

Single Program Multiple Data (SPMD) In an SPMD model, a group of threads work together on the same task (they run the same program). Each thread is identical apart from a set of values (identifiers) that is unique to that thread. Each thread uses the identifiers to determine its input and output data. This model works well, for example, for graphics problems, where for each pixel in the output image a thread is created which performs the same processing on different input pixes. The SPMD paradigm is at the heart of programming models for GPU-accelerated systems, including OpenCL [17], CUDA [22] and C++ AMP [1]. These programming models are discussed further in Section 3.4.1.

An SPMD program is usually called a *kernel*; the range of values on which it is executed is known as the *domain*. Most models support 2D and 3D domains in addition to 1D domains, and most parallel processors are designed to work efficiently with such domains.

While SPMD allows a large number of threads to be created, the threads are not guaranteed to be active at the same time, which can cause problems with shared memory and synchronization. Therefore, the domain is often divided into smaller *tiles*, with the guarantee that all threads within a tile are executing simultaneously. Where tiles are supported, the domain is usually constrained to be a multiple of the tile size. In some cases, tiles are further sub-divided into groups of threads executing in SIMD fashion (in NVIDIA's CUDA programming model [22] these SIMD groups are called *warps*).

Nothing about SPMD programs prevents an MPMD-capable machine from running them. Moreover, an MPMD program can be in principle converted to on SPMD one (with a series of if/then statements to allow each thread to pick out its own program); in practice, however, SPMD processors have significant optimizations that rely on threads taking similar paths through the program.

3.1.3 Symmetric Multiprocessing (SMP)

An SMP processor contains several identical *cores*. Each core is similar to the processor found in a uniprocessor system. The SMP processor includes additional logic to allow the cores to operate simultaneously. Due to the identical cores, an SMP processor on its own is an example of a homogeneous multiprocessing system implementing the MPMD threading model. An SMP processor can also be combined with other processors or accelerators to form a heterogeneous multiprocessing system.

The cores of an SMP system have a single view of memory: changes to memory made on one core are visible to the others. Each architecture supporting SMP defines a memory ordering model – a set of rules defining precisely when changes to memory made by one processor are made visible to the other processors.

The memory system of an SMP processor is similar to that found in uniprocessor systems – a hierarchy of one or more levels of cache and main memory. Each level of cache may be duplicated by all processors, shared by all processors, or fall somewhere between these extremes. Where multiple instances of a cache exist at a given level, a cache coherency mechanism is usually used to ensure that the memory ordering model is honoured. This is a tradeoff: the cache coherency system imposes area, power and performance costs to provide a memory ordering model that is easier for the programmer to use. More programmer-friendly memory ordering models typically impose a higher cost in the cache coherency mechanism.

An architecture supporting SMP includes facilities to allow synchronization between processors. These usually take the form of composite atomic instructions (e.g. swap, compare-and-swap, add, subtract, etc.) or atomic primitives (e.g. load-locked, store-conditional) which can be combined with data processing operations to produce the desired effect.

Typically, user programs running on an SMP machine have a virtualized view of memory, controlled by OS software and implemented by the memory management unit (MMU) in each processor. If desired, threads running on different processors can use the same MMU configuration, and thus share an identical view of memory.

As each core supports a single thread, that thread has performance comparable to a similar uniprocessor. This means that in general performance is relatively high, but memory accesses within a given thread need to demonstrate good temporal and spatial locality to make the most of the cache system and hence maximize performance. However, as each processor usually has at least one level of cache not shared with other processors, these locality requirements do not apply between threads.

Multithreading (MT) In a multithreading system, a single core supports multiple hardware threads. In other words, more than one hardware thread shares a set of execution resources. Several such cores can be combined to form an SMP processor which also implements MT, or one core on its own can comprise a uniprocessor system. Two popular multithreading models are **Simultaneous Multithreading (SMT)** and **Fine Grained Multithreading (FGMT)**.

In a *simultaneous multithreading* processor, each hardware thread supports the full architectural model and thus has its own copy of the register bank. Each thread runs for as long as the processor is active. The same OS mechanisms used on SMP systems are used to provide the illusion of multiple threads, and to switch between them; the OS sees each hardware thread as a separate processor (but will typically have some understanding of the relationship between threads and physical processors). SMT machines support the same MPMD threading model as SMP machines. Existing SMT implementations tend to have just 2 threads, as this provides the bulk of the benefit with the minimum overhead.

SMT allows a given set of execution resources to generate higher overall performance by allowing the extra threads to utilize the resources that a single thread cannot. This can occur in a superscalar system where one thread does not have enough instruction level parallelism (ILP) to fully exploit the functional units, but more interestingly it can occur when one thread is blocked due to a cache miss. This means that SMT provides a limited degree of latency tolerance. However, each thread shares cache resources with other threads running on the same processor, which raises the possibility of different threads impairing each others' performance by causing cache thrashing. Minimizing this requires an even greater degree of memory access locality within each thread, or alternatively a degree of memory access locality between threads.

Most SMT designs aim to avoid impairing the performance of a single thread, if only one thread is active.

A *fine grained multithreading* processor typically supports more hardware threads than an SMT machine. Hardware threads proceed down the pipeline in turn, with each thread having to wait a certain number of cycles before another instruction can enter the pipeline from that thread. This can extend all the way to the entire pipeline length, so that each thread can only be active in one pipeline stage at a time.

Some FGMT processors support a full architectural model for each thread and thus appear to the OS similar to an SMT processor. Other FGMT processors provide a limited amount of state per thread and do not support traditional OSs, but can instead run as a slave processor in a heterogeneous system (certain GPUs use this model, discussed in more detail below). In all cases, as each thread is independent of the others, FGMT processors can in principle support the MPMD threading model.

As each thread must wait several clock cycles between its instructions, fewer issue slots are lost to a cache miss, which makes FGMT processors latency tolerant. Depending on the processor, additional threads may be available for scheduling while a thread is blocked, further increasing latency tolerance.

Unlike SMT designs, an FGMT processor does not offer the full performance of the processor to a single thread, a certain minimum number of threads must be active before the processor becomes fully efficient. Therefore, programs not exploiting multiple threads are unsuitable for FGMT processors.

Many core processors *Many core* processors are distinguished from traditional SMP systems in that they generally have a much larger number of processors (systems with over 1000 cores have been designed), but each one is simpler than in a traditional SMP system. In particular, many techniques usually employed to maximize single thread throughput (e.g. deep pipelining, branch prediction, speculative execution, superscalar, out of order processing) are not used. Each processor therefore offers lower performance, but in return takes up less area and consumes less power. Many core processors aim to offer improved overall throughput and power efficiency compared to an SMP system of comparable size.

Many core processors do not tend to have the same level of memory coherency as SMP systems, since cache coherency protocols do not scale well to such sizes. Many core processors often include "scratch RAM" private to each processor and managed explicitly by the programmer to avoid this complexity. A shared view of memory is usually supported at some level, but performance may be low. Some many core processors group processors into clusters, providing higher performance communication between cluster members.

Each core is independent of all the others, so there is no need for different cores to run the same program. However, in a clustered system overall performance can be improved if the cores of the cluster work together on shared data.

Many core processors do not necessarily include full support for running the OS on each processor, as OSs do not tend to scale well to such high core counts and the support for privileged operation adds overhead to each processor.

As each core is independent, many core processors support the MPMD threading model.

General Purpose Graphics Processing Units (GPGPUs) Since their inception as fixed function graphics accelerators, GPUs have evolved into powerful parallel processors capable of running general-purpose programs On appropriate workloads GPUs offer far higher performance and power efficiency than similarly sized SMP systems.

A key difference between GPUs and CPUs is that while CPUs can operate on their own by running supervisory (OS) software as well as performing useful work, the threading model of GPUs makes them unsuitable for running OS software. Any system with a GPU must therefore include a CPU as well, which runs driver software to control the GPU and issue tasks to it. Therefore there is no need for the GPU to implement privilege levels or system instructions.

GPUs implement a Single Program Multiple Data (SPMD) model, so for example when running a pixel shader, the GPU creates a thread for each pixel to be processed, which runs the program for that specific pixel and then terminates.

Unlike CPUs, which provide a fixed number of hardware threads that run continuously and are switched between programmer threads in software, GPUs include dedicated hardware to create and terminate threads. When the host CPU requests execution of a program over a domain, the GPU does not necessarily start all of the requested threads immediately; it can start a smaller number and continue to spawn additional threads as existing ones complete. This hardware thread management significantly reduces the penalty for short-lived threads: graphics tasks often include programs executing a handful of instructions per thread.

Similarly to many core systems, the performance of each individual thread is not important: the overall throughput (performance of all threads in aggregate) determines the performance of the GPU system. GPUs take advantage of this fact by running a very large number of threads (usually thousands) in parallel, while only providing modest performance for each individual thread.

Execution resources are shared amongst threads. Most GPUs implement a fine-grained multithreading (FGMT) model, where threads are switched at short intervals (typically on every cycle). Generally sufficient threads are present that any given thread only needs to have a single instruction in flight through the pipeline at a time. This eliminates the need for branch prediction (as a thread can wait until the branch is resolved before beginning to execute the next instruction) and forwarding paths (results can always be committed to the register bank before the next instruction runs), reducing the area needed per execution unit and the energy used to execute each instruction. If a thread has to wait a long time for a memory access, other threads can be run in the meantime; the waiting thread causes little or no drop in throughput. However, if many threads issue memory accesses in parallel then the overall memory bandwidth available can limit the rate at which requests can be satisfied. This means that compared to CPUs, GPUs are tolerant of latency and sensitive to memory bandwidth.

FGMT-based GPUs can achieve greater performance by using SIMD techniques to allow each thread to do more work. In graphics workloads, manipulation of pixel data is often amenable to being accelerated in this way because each pixel has several color components which must be manipulated in the same way. The 4 channels of RGBA pixel data map neatly onto 4-lane SIMD features. GPUs designed this way can expose these SIMD features to programmers of GPGPU tasks.

Many GPUs achieve further area and energy savings by using a Single Instruction Multiple Thread (SIMT) model. In this model, several threads are grouped together. Such groups are often referred to as *warps* (though strictly this term refers to NVIDIA GPUs). The same instruction is run simultaneously by all the threads in a warp, which means the threads can share all the fetch and decode logic. Warps are then scheduled in a manner similar to the threads on an FGMT GPU, with several warps being active at a time. When the threads of a warp *diverge*, following different execution paths, one candidate instruction is selected and only those threads that need to run that instruction proceed while the others wait. SIMT machines are therefore sensitive to thread divergence.

Both FGMT and SIMT GPUs are able to exploit further parallelism by putting down multiple sets of execution units (each set is called a 'core'). This allows families of GPUs with varying performance characteristics to be designed relatively easily by varying the number of cores included.

Discrete GPUs have dedicated *global* memory, which offers significantly higher bandwidth than main memory. Some GPUs employ caches to reduce the need for off-chip memory accesses, but others include software-managed local memory instead of or in addition to a cache.

Since several threads running on a GPU share execution resources and access to the memory system, there is considerable benefit to memory access locality between threads. If a GPU uses warps, it may be able to combine (coalesce) memory accesses from several threads within the warp if they are accessing a contiguous block of memory. In GPUs which include small local memories close to the execution resources, threads within a warp (or larger group of threads) may need to co-operate on their use of this memory.

Modern GPUs support virtual memory, but since they do not run the OS they cannot handle demand paging directly. This can be worked around by making a request to the CPU to fix up page faults, but for the sake of driver simplicity and performance many GPU drivers opt to use a different memory map for the GPU and 'lock' all memory objects used by the GPU into memory to prevent page faults. This approach is also necessary for GPUs with private dedicated memory.

Because GPUs evolved to run graphics tasks, they often have specialized instruction sets tailored to include these graphics functions. Historically, the interface to GPUs has been graphics libraries such as DirectX [21] or OpenGL [16], with programs (called *shaders*) specified in relatively high-level languages. The actual instruction set of the GPUs is therefore not exposed to programmers directly.

3.2 Programming Model Design Decisions

In Section 3.4 we provide a comprehensive overview of parallel programming models. To prepare for this, we first provide an overview of the various components and features that must be considered when designing a parallel programming model.

3.2.1 The origins of different programming models

A number of different models have been developed in order to address the problem of programming parallel processors. The existence of so many different models is driven by three factors.

1. To support the wide range of different parallel architectures a model has to be complex enough to deal with all the differences or abstract enough to hide them from the programmer. Neither of these options can deliver full performance across the range of architectures. As a result, a number of different models have been developed that target particular parallel architectures but which may be inefficient (or even impossible to implement) on other architectures.
2. Another contributing factor is the difference in the nature of the problems that are being addressed. A model that works well for intensive computation on simple data sets may not be appropriate for problems that require complex data structures or algorithms. The importance of parallel processors in graphics and high-performance computing has led to architectures and software that are biased towards intensive computation on simple data sets.
3. The final factor is the political and commercial factors surrounding each model. In some cases similar models have been developed by different commercial entities in order to avoid using technology developed by a competitor. In other cases models have been developed in isolation to avoid the complex political issues involved in a standardization process. And sometimes new models have been developed simply because existing models were ”Not Invented Here”.

These factors continue to affect the popularity of existing models and the development of new models, and so there is no prospect of a single model gaining universal acceptance. These models are still evolving, and we anticipate that in the long-term a few models will dominate and the remaining ones will disappear or remain restricted to particular programming niches.

3.2.2 Describing parallelism with threads

In order to take advantage of a parallel processor the programming model must allow the programmer to create a number of parallel threads and describe what each of the threads is going to do. Two basic models for this are *Multiple Program Multiple Data* (MPMD) and *Single Program Multiple Data* (SPMD), which we described in Section 3.1.2.

The MPMD model is effective for targeting SMP and MT processors but does not perform well on the massively-parallel architectures. SPMD is more effective for targeting massively-parallel processors, but is a subset of MPMD so can also be used to target SMP systems.

With either model it is possible to take advantage of SIMD instructions by using them within the individual threads.

3.2.3 Choice of programming language

A program for a parallel processor can generally be divided into two parts: serial code and parallel code. The language used to program these two parts varies significantly between models.

- The *serial code* manages the user interface, prepares the data for parallel computation and deals with the results of the computation once it is complete. Most models use an existing programming language for this part of the program, though in some instances a new language has been developed that encompasses both the serial and parallel parts of the program.
- The *parallel code* is used for the parts of program that can take advantage of the multi-threading features of the parallel processor. There is much more variety in the languages used for this code. Some models use the same language as the serial code and represent parallel processes using libraries or by marking up the code. Most models use a parallel language that is closely related to the serial language but with restrictions on existing features and with the addition of new language features to address issues specific to parallel programming. A few models use a completely different language for the parallel code while others allow parallel code to be called from serial code written in a variety of standard programming languages.

The way in which parallel code is integrated into the serial code varies between different programming models. Where the parallel code uses the same language as the controlling program the integration can be seamless, with serial and parallel code being mixed within the same function or even the same block.

Where a different language is used for the parallel code it is written as a separate function. In some models this function must be in a different file that is compiled separately from the main program while others allow mixing of serial functions and parallel functions in the same source file.

In some cases the model is supported using only API functions, in other cases it is supported via extensions to the serial language or a combination of API functions and language extensions.

We discuss language choices for serial and parallel code for a variety of programming models in Section 3.4.

3.2.4 Machine-code generation

The time at which the parallel code is compiled into the machine code varies between programming models.

Traditionally (and e.g., in POSIX threads [27] and OpenMP [24]), the parallel code is compiled at the same time as the serial code and delivered as part of the program. This is often the case where the same language is used for the whole program, or where the same instruction set is being targeted by both parts.

Compilation may be delayed so that the parallel code is delivered in source form and compiled when the program is run; this is the approach used e.g. in OpenCL [17]. The run-time compiler can generate and optimize code for the particular processor implementation available in the system. In this case it is possible for the run-time compiler to optimize for processor implementations that were not available when the program was written.

A hybrid approach (taken e.g. by NVIDIA's CUDA programming model [22]) involves two-stage compilation in which the parallel program is converted to an intermediate form which is then delivered with the rest of the program. When the program is run this intermediate form is compiled to the final machine code for the parallel processor. This model simplifies the process of run-time compilation while still allowing optimization for the specific available processors. This model is also useful when a hardware vendor does not wish to expose the true instruction set of their architecture, but is willing to make available a suitably low-level byte code which can be targeted by optimizing compilers.

3.2.5 Memory model

Memory architectures have a significant effect on the programming model.

Uniform memory Some programming models provide a uniform view of memory so that parallel code has exactly the same view of memory as the serial code. Memory allocated by the serial code can be used directly by the parallel code, and in some cases memory can be allocated directly by the parallel code.

While this is the most convenient model for the programmer it excludes the use of parallel processors with segmented memory architectures, which includes most of the massively parallel processors.

Segmented memory Processors with segmented memory architectures require data to be explicitly copied between the different areas of memory. Therefore programming models need a mechanism to describe how input data is written to the parallel processor and how results are read back. There is often a limited amount of memory available so there also needs to be a mechanism for allocating memory on the parallel processor that is separate from memory allocation on the serial processor.

Programming models will typically allow the program to define blocks of *shared memory* that have matching allocations on both the serial processor memory and parallel processor memory. Some programming models require the programmer to explicitly copy data between the different memory segments at the right time. Other models will automatically copy the data to where it is needed, in which case there must be a mechanism for describing whether memory is an input to the parallel code, output from the parallel code or both.

Private memory Each thread has its own stack but some models also provide access to *private memory* which is additional local memory for each individual thread. This memory is uninitialized when the thread starts and is discarded when the thread ends and can therefore be used only as temporary storage. In many architectures this memory is significantly faster than other memory and can be used to store intermediate results that are too large to be held in registers.

Tile memory When tiles are supported a form of local memory is usually available that can be accessed by all threads in a tile. This can be use to share results between threads efficiently, but some form of tile synchronization is required to ensure that reading and writing this memory happens in the right order.

Some programming models allow efficient copying of data between the different types of memory in a segmented memory system and, in particular, efficient copying of data into local memory. This can take advantage of any acceleration hardware in the parallel processor.

Constant memory Some models provide explicit support for constant memory which is initialized by the serial code before the parallel code and is available to all threads. With segmented memory models this data is automatically copied to the parallel processor as required.

Shared memory In some cases the parallel processor uses the same physical memory as the serial processor but not the same address space. In this case some models provide the ability to create areas of shared memory that can be accessed by both the serial and parallel code without copying. The model still needs to provide a mechanism for mapping between addresses on the serial processor and addresses on the parallel processor.

Graphics memory Graphics has been a major driver for the development of parallel programming and there is often an overlap between the compute and rendering parts of an application. Therefore some models allow the allocation of memory that is specifically designed to be shared between the parallel code and other graphics libraries. This allows parallel code to be used to create images, textures and geometric data that are subsequently used to render graphical frames. There will usually be constraints on the size and placement of the memory for these buffers to allow efficient sharing.

3.2.6 Parallel data types

Parallel code may support new types that are not supported in serial code. These types include fixed precisions numbers, 16-bit floating point (half float) and non-IEEE floating point. There may be support for vector types, either as explicit 2/4/8 lane SIMD vectors or via coordinate types with explicit x, y, z and w elements.

Conversely, parallel code may not support types that are available in serial code. Specifically, high-precision floating point (double and higher) may not be available on the parallel processor and thus may not be available in the programming model. Languages targeted at pure computation may also lack other types such as Booleans, enumerations and bit fields.

Pointers

Pointers types may not be supported or may be restricted to simple data types rather than pointing to objects or functions. This is particularly the case where a segmented memory architecture is supported because pointers do not translate between the two memory areas. The type of data that can be transferred between serial and parallel code are usually limited to scalar values, simple vectors and simple structures.

Programming models that support multiple memory spaces tend to provide a set of pointer *qualifiers*, allowing the programmer to specify which memory space a pointer refers to [22, 17, 6, 8]. For example, in the OpenCL programming model [17] a pointer can be marked as `__private`, `__local`, `__global` or `__constant`. This allows the compiler to perform type checking, providing some guarantee that memory spaces are used correctly.

Object-orientation Parallel languages will often lack the features used by object-oriented programming, such as inheritance, data hiding and polymorphism. The lack of functions pointers also makes it impossible to define abstract types.

Image types Special image data types may be provided to support the acceleration available in graphics-based parallel processors. These allow fast access to 2D or 3D data but may have restrictions on the dimensions of the image and the data type of the individual elements.

Textures Texture data types may be provided to take advantage of the support for textures in graphical processors. These types allow efficient access to a 2D or 3D image and also provide very fast interpolation to allow sampling of the texture at higher resolution. Textures may also support efficient access to a filtered lower-resolution version of the image which can be used to optimize a range of non-graphical problems in vision, image processing and elsewhere.

3.2.7 Scheduling

Each programming model defines the way in which work is scheduled for the parallel processor by the serial code. This is typically in the form of a task that is created by the serial code and handed to the parallel processor for execution. Each task creates one or more threads on the parallel processor. In a simple synchronous scheduling model the serial code must wait until the task is complete before continuing.

Asynchronous scheduling allows the serial code to continue execution while the task is being executed by the parallel processor. This requires a mechanism to allow the serial code to wait for the task to complete before the results of the task can be read. This also allows the possibility of multiple tasks being active at once which may allow for more efficient use of the parallel processor.

In segmented memory architectures the cost of copying data between serial and parallel code can be a significant part of the overall cost. Most models allow this copying to happen asynchronously with kernel execution so that the results from the previous kernel and the input for the next kernel can be copied while the current kernel is executing.

If a number of tasks need to be scheduled it can be complex for the serial code to manage, especially if there are dependencies between tasks. Some models therefore allow groups of tasks to be started together, with some description of the dependencies between the tasks (typically a Directed Acyclic Graph). Likewise it may be possible to create queues of tasks that are dispatched in order and executed on as many threads as are available.

3.2.8 Synchronization

Both the MPMD and SPMD models create multiple threads that can execute in an unpredictable order and in parallel. The programmer cannot make assumptions about the order in which they execute. In particular, threads that share memory locations must work correctly whatever order the memory is written and read by different threads. In some cases this restriction is enforced by the model but this is usually left up to the programmer to manage.

Atomics The model may provide mechanisms to avoid synchronization problems by allowing multiple threads to safely access the same data. This may include the provision of atomic operations on memory locations (+, -, `min`, `max` etc.) and critical subsections to protect longer subsections of code.

Fences Memory fence primitives ensure that changes made by a thread are visible to all other threads. This can be used to implement some simple data sharing mechanisms without the need for more complex (and expensive) primitives.

Barriers

Models that provide support for tile memory will typically provide barrier instructions that ensure that all threads have reached a given point before any thread goes past it. This ensures that the results of all write operations before the barrier are visible to all threads after the barrier. Barriers are a core feature of GPU programming models, including CUDA [22], OpenCL [17] and C++ AMP [1].

Other synchronization primitives

Some models provide more complex synchronization primitives such as semaphores and channels, but these bring with them a greater risk of introducing deadlocks into the parallel code.

3.2.9 Library support

In most cases the serial code has full access to all the system libraries including file I/O, UI code and access to devices via drivers. The parallel code is typically very restricted in which system services it can access and usually only has limited access to programming libraries for maths and simple data structures.

Some parallel architectures provide acceleration for certain operations such as maths functions, logical operations and bit manipulation primitives. These can be presented as new library functions that are only available to the parallel code or as optimized versions of existing libraries.

3.3 Modifying Programs for Parallel Processing

It is usually necessary to modify programs to suit a parallel programming model in order to take best of advantage of a parallel processor, or indeed to run on a parallel processor at all. These modifications can vary from simple changes to a complete re-working of the whole program. The modifications affect a number of different aspects of the program, including the underlying algorithms and the data structures.

3.3.1 Algorithm Modifications

The extent to which the existing scalar algorithm has to be modified to run well on the parallel processor can range from minimal changes in the calling code to a bottom-up re-write of the algorithm. For any given algorithm the position in this range will depend on the programming model being used and the nature of the algorithm itself. At the extreme end of this range it might be necessary to find a completely different algorithm for solving the same problem which has more inherent parallelism. For example, a bin-based sorting algorithm is going to be more efficient on a parallel processor than one which requires access to all data on every iteration, even if it is less efficient on a serial processor.

Data and task parallelism An algorithm can be made parallel in two distinct ways:

1. With *data parallelism* the threads are arranged so that each thread works on a different part of the input data, with each thread performing essentially the same computation.
2. With *task parallelism* the threads are arranged so that each thread works on a different part of the problem, typically in a pipeline, with different threads performing different computations.

For more complex problems these models are combined in different ways. Tasks can be nested (e.g. a pipeline of tasks which use data parallelism internally) and they can be applied sequentially (e.g. a data parallel step followed by a task parallel step).

Algorithms for MPMD Where an MPMD model is being used there is typically good data sharing between threads but a relatively small number of threads are available. In this case a task-parallel algorithm is often a good fit: it typically requires fewer threads, and data can flow efficiently between the various stages of the pipeline. If data parallelism is used it is typically at a coarse grain, with each thread working on a significant portion of the input data.

Algorithms for SPMD An SPMD model is most efficient when each thread is performing the same computation on contiguous data, so in this case a data parallel model is more applicable.

There is significant advantage in simplifying an algorithm to fit into a regular pattern, even if this involves more computation than a more complicated algorithm. The overheads of managing the complexity outweigh the reduced amount of computation. For example, an SPMD implementation of an n-body solver might compute the full N^2 set of forces even though, by symmetry, only $N^2/2$ computations are actually required, because taking advantage of the symmetry severely reduces the efficiency of the parallel processor.

In massively-parallel architectures the cost of computation is relatively lower than the cost of data movement compared to a scalar processor. This moves the balance towards algorithms that re-computed values rather than computing them once and caching them in memory. Thus the use of techniques such as pre-computed look-up tables on a massively-parallel processor may not provide the efficiency gains that would be expected on a scalar processor.

Use of SIMD Some problems can take advantage of SIMD with little modification, especially algorithms that involve significant computation on 3D/4D coordinates or 3/4-channel images. Models that provide support for parallel vectors (e.g. Intel's ABB) can also take advantage of SIMD automatically.

In other cases it is usually necessary to split loops or turn them inside-out to take advantage of SIMD. For example, a loop that computes two set of values and writes them to two arrays might be re-written as two loops, one for each array. With SIMD this could be more efficient even if it meant re-calculating some values that were shared in the scalar version.

While these kinds of operations can occasionally be optimized automatically by a vectorizing compiler, the programmer will usually have to explicitly code the SIMD operators or at very least re-write the loop so that the vectorizing compiler can detect the pattern. The state-of-the-art in automatic vectorization is surveyed in Section 4.2.3.

SIMD code is also inefficient if the number of elements being computed is not a multiple of the lane width of the SIMD processor as some lanes will be unused. This kind of optimization also introduces a dependency on specific hardware because the number of lanes and the particular set of operations that are available is different between different hardware architectures or implementations.

3.3.2 Data modifications

The data structures used by a program depend largely on the algorithm used, but within a given algorithm there may be changes that have to be made to fit a particular programming model or in order to achieve acceptable performance.

Scalar types The underlying data type may need to be altered to improve efficiency on a parallel processor. This is especially true if double-precision floating-point values are used because a number of massively-parallel architectures do not support them or support them with limited performance. The algorithm may have to be modified to cope with the reduced accuracy of single-precision floating-point.

Conversely, some algorithms may be able to accept the reduced accuracy provided by half-precision floating-point or fixed point numbers which may provide better performance on a parallel processor. It may also be necessary to pay more attention to the use of unsigned data types as these may perform differently from signed data types.

Structures of Arrays Many algorithms work on structured data, where a number of values are grouped into a structure which is then replicated in a large array. This arrangement is known as an Array of Structures (AOS) and is efficient on a scalar processor where the single thread is operating on each array element in turn.

AOS is not so efficient where multiple threads are operating in parallel on different elements of the array. In this case the data is more efficiently stored by having separate arrays for each element of the structure. This arrangement is known as a Structure of Arrays (SOA) and is particularly effective on SIMD architectures where multiple values can be read and written from sequential memory.

With very large data sets the separation between related data values in AOS can cause problems with caching and TLBs. In this case it can be effective to limit the size of an SOA to fit within the page size and then replicate this multiple times, leading to an Array of Structures of Arrays (AOSOA).

In each case the most efficient partitioning between structures and arrays will depend on the particular architecture being targeted.

Image formats Algorithms that process pixel data can choose between the usual interleaved formats or planar formats where the data for each channel is in a separate area of memory. The trade-offs are the same as for the AOS vs. SOA choice for non-pixel data.

The 2D nature of image data makes other data arrangements even more efficient for parallel processing. In particular the data can be arranged in image tiles where each tile covers a 2D rectangle of the image but is stored linearly in memory. This provides better memory locality for algorithms that operate on regions of the screen, but it increases the overhead of address calculation of a pixel. Parallel processors avoid this overhead by providing special hardware to compute these addresses if the image data is stored in a supported format.

3.3.3 Partitioning

A key issue when targeting a parallel processor is determining which parts of the program can be accelerated and which parts are best left running on the serial processor. This depends greatly on the size of the problem and the overheads associated with using the parallel processor. Section 3.5 on performance covers this issue in more detail.

3.3.4 Automatic translation

In some cases it is possible to transform serial code into parallel code automatically. The most mature form of this is vectorization of scalar code to allow the use of SIMD instructions, which is implemented in most modern compilers. In some cases this can be extended to creating kernels for SPMD machines. Translation of serial code to parallel code using MPMD threads is more complex and is not applicable to most problems without sophisticated compiler analyses, significant modification to the original program, or both.

The most promising techniques for automatic transformation of serial to parallel code are based on the *polyhedral model*; we present the state-of-the-art for these techniques in Chapter 4.

3.4 Current Programming Models

3.4.1 Major GPU Programming Models

We provide a detailed overview of the three main programming models for GPUs: OpenCL, CUDA and C++ AMP. We note that OpenCL and C++ AMP are general-purpose models and go beyond the scope of GPU programming.

OpenCL

OpenCL (Open Compute Language) was originally created by Apple and is now maintained by the Khronos group. It is a pure C API which allows execution of SPMD kernels on a variety of devices. OpenCL is particularly suited for executing tasks on GPUs.

Motivation. OpenCL was created as Apple wanted to exploit the potential of GPGPUs, but did not want to be tied to NVIDIA's CUDA. By working with the Khronos group they were able to make it a vendor-neutral API which would be compatible with any GPU on any OS.

Parallel Language. Kernels are written in OpenCL C. This is a derivative of C99 which adds support for features commonly found on GPUs such as native SIMD data types, and removes support for features such as function pointers. There is no library support, but an extensive set of built in functions is provided. These include essential SPMD model functions (i.e. get coordinates), math functions (sine, cosine, etc.), image functions (to access texture functionality) and synchronization functions.

Task Dispatch. All OpenCL tasks are dispatched to queues. Each queue is associated with a single device but it is possible to associate several queues with the same device. The programmer can specify whether tasks submitted to a queue must execute in order or can be reordered by the runtime.

Kernels are dispatched from serial code running on the host processor via afunction called `clEnqueueNDRangeKernel()`. The programmer specifies the kernel, domain and optionally tile (called a 'workgroup' in OpenCL) size. It is also possible to dispatch other operations such as memory copies. Each submitted task optionally has an event object associated with it — these can be used to express dependencies between operations, enquire about the status of on operation, or direct the host processor to wait until a given task has completed.

Synchronization. A variety of synchronization features are provided. Within kernels, atomic operations are supported via built in functions. These include a variety of integer operations and compare-and-swap. Moreover, within kernels, barriers are supported which cause each thread in a tile to wait until all threads have reached the barrier before all threads continue running.

Between kernels, or between kernels and the host CPU, command queue barriers are available which cause all tasks before the barrier to complete before any subsequent task starts. There is also the facility to cause the host to wait for the completion of a specific task before continuing.

Memory Model. OpenCL uses a segregated memory model. All memory operated on by kernels has to be in the form of 'memory objects'. At the runtime level these can be either 'buffers' (flat arrays of memory) or 'images'. In kernel code, a buffer appears as a C-style pointer and can be used in a similar fashion. Images appear as opaque objects and can only be accessed by built-in image access functions. These functions use a GPU's texturing capabilities and so images can be more efficient than buffers, depending on circumstances and the GPU in use.

On the host processor, there are explicit functions to copy data between host memory and memory objects. There are also routines to 'map' a memory object directly into host memory. When creating a memory object, it's possible to provide a pointer and ask the runtime to use this to store the data via the `CL_MEM_USE_HOST_PTR` flag, but it is up to the runtime how this is actually done. In cases where the host CPU is being used as the compute device this is obviously most efficient as the memory can be used directly, but when a GPU is being used as the compute device it's likely the runtime will perform a copy anyway. Even when `CL_MEM_USE_HOST_PTR` is used, it is still necessary to call the mapping functions before the memory is guaranteed to reflect any updates made by the GPU device.

Compilation Model. In order to execute a kernel, a kernel object must be created at runtime. There is an option to use a precompiled binary for this, but compatibility is only guaranteed when using the same driver revision and the same device. The recommended route is online compilation from source at runtime. Source is provided via one or more C strings, which allows code to be generated at runtime or just read in from a file.

Minimum Steps Needed to Run Parallel Code. OpenCL is a complex API with a lot of explicit control available to the programmer; therefore there are a lot of steps involved:

1. Get list of supported platforms (often only 1).
2. Get list of supported devices on chosen platform (often only 1 GPU).
3. Create a context object against the relevant platform and device.
4. Create a command queue against the device and context.
5. Create a program object from the OpenCL C source code.
6. Compile the program object.
7. Create a kernel object from the program object.
8. Create any memory objects needed and copy input data.
9. Set kernel arguments.
10. Enqueue kernel execution on the queue.
11. Flush the queue to make execution actually occur.
12. Copy output data back to the host.

Discussion. Although it is pitched as a 'universal' compute language, OpenCL is pretty clearly tailored to the strengths and weaknesses of current generation GPUs. The memory model, instruction set and task scheduling aspects are all neatly captured by memory objects, online compilation (program objects) and NDRanges.

As a pure C API, compatibility is maximized (integration is via an include file and a shared library, as for any library code), but the lack of a language extension does lead to a lot of convoluted boilerplate code. However, this is ripe for encapsulation in friendlier higher level routines, in particular C++ objects. Standard C++ bindings exist which are easier to use.

OpenCL can be used to exploit the multiple cores of an SMP system as well as a SPMD GPU system, although this is not an attractive option in isolation as other programming models (e.g. OpenMP) are much easier to use for this purpose. However, the ability to use a common interface and runtime to target either a GPU or a CPU can be a distinct advantage as it allows portable programs to be created which can exploit the GPU if present, or fall back on the CPU otherwise, without having to maintain two copies of the code.

Another aspect of OpenCL is its flexibility; the hierarchy of platforms and devices means that it can potentially work on a system with several GPUs or accelerators from a mixture of vendors, but at the cost of complexity which the programmer has to deal with even in the most common case of a single GPU from a single vendor.

CUDA

CUDA (Compute Unified Device Architecture) [22] was the first mainstream language for performing compute on Graphics Processors. It was developed by NVIDIA and is currently supported only on their devices, though there are plans to make it available more broadly.

CUDA represents the GPU as a coprocessor that can run data-parallel kernels and provides extensions to the C language to (1) allocate data on the GPU, (2) transfer data between the GPU and the CPU and (3) launch kernels.

Motivation. As OpenGL and GPUs became more programmable and powerful, programmers started experimenting with expressing problems as OpenGL shaders, using various tricks to get data into the required format for OpenGL processing, and extract data from the output. This worked but was limited by the constraints of the OpenGL API. NVIDIA recognized that with minimal extra hardware, GPUs could be made more accessible for general purpose compute, and this could be a competitive advantage in the GPU market. CUDA was created to facilitate this.

Parallel Language. The parallel language is based on C99 with a number of extensions. Kernels are written as separate functions and can be in the same source code file as the serial code or in a separate file. Parallel functions that can be accessed by serial code are specified using the `__global__` attribute and local functions for the parallel code use the `__device__` attribute. The kernel has access to built-in variables which provide the location of the kernel within the domain, the tile size and the location within the tile.

A CUDA kernel executes a piece of sequential code on a large number of threads in parallel. These threads are organized into a grid of CUDA blocks. Threads within a block can cooperate with each other by (1) efficiently sharing data through a shared low-latency local memory and (2) synchronizing their execution via barriers. The scheduling unit is not the individual thread but a group of threads called a *warp*. Every cycle, the scheduler in each multiprocessor chooses the next warp to execute. If threads in a warp execute different code paths, only threads on the same path can be executed simultaneously and a penalty is incurred.

A number of vector data types are provided containing 1–4 elements of the standard integer and floating-point types. Moreover, CUDA provides a comprehensive set of maths libraries for parallel code. It also includes some special operations that acknowledge the fact that the implementation uses warps internally, including the ability to shuffle data between different threads in the same warp.

Task Dispatch. CUDA kernels are dispatched from the serial code using a language extension to specify the tile size and the number of tiles in each dimension. The scheduling mechanism allows asynchronous operation of kernels by creating streams of commands. Events can be placed within the stream to allow the serial code to monitor the progress of the parallel operations.

Synchronization. Tile synchronization is available via barriers using the built-in function `__syncthreads()` from within the kernel, with variants that allow simple voting between threads. Various fence commands are provided to ensure consistency within various levels of memory. Moreover, a broad range of atomic operations is available, including add/sub, min/max and bitwise operations.

Memory Model. CUDA supports global, local and constant memory. Variables can be marked with the appropriate attribute to indicate where values should be placed and pointers can be marked to indicate which kind of memory they are addressing. In most cases data is explicitly copied between host and GPU using the `cudaMemcpy()` API function. CUDA also supports shared memory which is treated as global memory by the kernel but required special handling by the serial code. It is also possible to allocate write-combined memory which has performance advantages for data that is passed from the serial to the parallel code. Images and textures are supported using a templated texture class.

Compilation Model. CUDA uses a special compiler (nvcc) that separates the serial and parallel code in the source file. Serial code is fully compiled for the target CPU. Parallel code can be fully compiled but is typically compiled to a low-level intermediate language called PTX. This language is close to the instruction set of the underlying processors but can be optimized by the compiler before execution on a particular platform. Using PTX is the only way to support portability of code to future NVIDIA platforms. It is also possible to write PTX directly for highly optimized code.

Many details must be exposed to the compiler in order to generate efficient CUDA code. The fine grained multi-threading or thread block to multiprocessor mapping, are hidden, however. The GPU can be seen as an arbitrary large number of (virtual) SIMD-like processors could be present, each with its own register file and private scratchpad memory, all sharing a common DRAM-based main memory. In addition, some constraints are defined when accessing these different memory levels. Unlike the execution model, where the execution unit is the warp, the memory access unit is the half-warp. When threads within the same half-warp access to global memory, the threads must access to consecutive memory positions, or more than one request is launched; this is commonly called *uncoalesced access*. On the other hand, the shared memory is a banked memory; so threads within the same half-warp that access different elements in the same bank will incur *bank conflicts*. The violation of these restrictions usually result in slowdowns.

Minimum Steps needed to Run Parallel Code.

1. Allocate device memory.
2. Copy input data in to device memory.
3. Invoke kernel.
4. Copy output data from device memory.

Discussion. CUDA is the original GPU compute language and is the incumbent that others have to beat. However it is a closed language and therefore compares poorly against cross-platform standards. CUDA is designed specifically to operate on NVIDIA processors and encourages a model that is efficient on such machines but might not perform so well on other architectures.

Given the paramount importance of the memory system, the compiler must be aware of the coalescing requirements, so it must explicitly take into account the difference between a warp and a block of threads even if they are executed on the same virtual SIMD processor. Nevertheless, as for other language extensions, CUDA makes it easier for programmers to get code running by requiring less boilerplate code which must be written and debugged. Mixing parallel and serial source code also makes it easier for existing code to be adapted, although programmers still need to separate parallel code into separate functions and take account of the memory model requirements.

C++ AMP

C++ AMP (Accelerated Massive Parallelism) [1] is an SPMD model from Microsoft that uses versions of the C++ language for both the serial and parallel code.

Motivation. C++ AMP is an attempt to simplify the processor of writing programs for massively-parallel processors in a Microsoft environment by using C++ language features. It is essentially a way of targeting the DirectCompute part of DirectX without having to create all the graphics infrastructure which that requires.

Parallel Language. The parallel language is a subset of C++ with a small language extension to identify parallel code. The restrict keyword can be applied to any function and it defines whether that function can be used in serial code, parallel code or both. Parallel functions are labeled `restrict(amp)` and serial functions are labeled `restrict(cpu)` or left unlabelled. Parallel code can only call functions that are restricted to amp (or both cpu and amp) which prevents the use of any system functions and most library functions. This also provides a mechanism for providing libraries that support specific acceleration features on the parallel processor that are not available to serial code. C++ polymorphism allows the creation of separate serial and parallel versions of the same function with the appropriate version being selected by the compiler based on the restrictions of the calling function.

There are significant restrictions on the C++ features that are allowed in parallel code. Pointer support is limited to simple data pointers held in variables (hence no virtual functions), and there is no access to static or global variables. The code must be able to be inlined at compile time which prevents the use of recursion or any form of dynamic linking.

C++ AMP is an evolving standard and some restrictions will be removed in future versions or are specific to the Microsoft implementation.

Task Dispatch. Tasks are dispatched using a `parallel_for_each()` function that takes a domain and a kernel. The kernel takes a single parameter, the index, which defines a position in the domain. `parallel_for_each()` calls the kernel in parallel for every position in the domain and the function can extract the *x* and *y* offsets from the index as required.

The scheduling mechanism allows multiple queues to be created, each of which allows multiple `parallel_for_each` calls to be submitted without waiting for the previous one to complete. Moreover, various atomic operations are available.

Memory Model. C++ AMP provides a segmented memory model and uses C++ templates to create areas of memory that can be used by the parallel code. The runtime manages the copying of data between the different memory segments as appropriate, and uses ‘const’ and ‘write-only’ hints to avoid copying data unnecessarily.

C++ AMP allows the domain to be sub-divided into tiles and supports local memory by adding the `tile_static` keyword to a variable declaration within the parallel code. A single instance of the `tile_static` variable is created for each tile and this is visible to all threads in that tile. The index contains additional information about the position of the thread within the tile and also provides a barrier object to allow synchronization between all threads in a tile.

Textures are supported using a texture class, and some short vector types are also available. There is a `fast_math` library that provides fast but less precise versions of various maths functions and a `precise_math` library the supports double-precision values.

Compilation Model. Serial code is fully compiled for the target CPU but parallel code is compiled to an intermediate language, DirectX HLSL. Functions that support both forms of code are compiled twice, once for the CPU and once for the parallel processor.

Minimum Steps Needed to Run Parallel Code

1. Wrap host memory in array classes.
2. Call API to invoke kernel.

Discussion. C++ AMP provides a reasonably clean mechanism for adding parallel code to existing applications and allows programmers to re-use some knowledge of C++ when writing kernels. It is a relatively new model and lacks the more sophisticated features of other, more mature models.

3.4.2 Other Major Programming Models

We now consider four further parallel programming models that are in widespread use and of significant relevance to the CARP project, though not specifically geared towards GPU programming. We present a brief description of each in the main text, and in Appendix A we provide a comprehensive overview of each model.

Table 3.1 compares the programming models discussed in this section and Section 3.4.1 according to several key features.

POSIX threads The POSIX threading API (‘pthreads’) [27] is an IEEE standard and serves as the native thread API on most UNIX-like systems. It is a pure C API implementing the MPMD model. Some compilers (including GCC) include a language extension for thread local storage (`__thread` keyword).

OpenMP OpenMP [24] is maintained by the OpenMP Architectural Review Board, an independent non-profit corporation composed of members from various interested parties. OpenMP is a language extension for C and FORTRAN which allows programmers to use pragmas to indicate parallel subsections of code. OpenMP is generally implemented on top of a native threading library — pthreads in the case of UNIX-like systems.

Renderscript Renderscript was created and is owned by Google as part of their Android platform [2]. Renderscript was introduced as a way to allow applications to include high performance code without making them machine dependent. In addition to its compute features Renderscript also includes graphics rendering functions. We consider it strictly as a compute acceleration language.

	POSIX	OpenMP	CUDA	OpenCL	Renderscript	OpenACC	C++ AMP
Model	MPMD	<both>	SPMD	SPMD	<both>	SPMD	SPMD
Language	Single	Single	Single	Dual	Dual	Single	Single
Segmented?	No	No	Yes	Yes	?	Yes	Yes
Automatic Copy?	-	-	NO	NO	?	Yes	Yes
Dynamic libraries	Yes	Yes	NO	NO	?	NO	NO
Recursion	Yes	Yes	NO	NO	?	NO	NO
Task Chaining	No	Yes	Yes	Yes	?	?	?
Task Nesting	Yes	Yes	NO	NO	?	?	NO
Standard Libraries	Yes	Yes	NO	NO	NO	NO	NO
API / Extension	API	Extension	Extension	API	API	Extension	Extension
Intermediate Language	No	No	Yes	Optional	Yes	Optional	Yes
Minimum steps	1	1	4	12	5	1	2

Table 3.1: Comparison of major programming models discussed in Sections 3.4.1 and 3.4.2.

OpenACC OpenACC [23] is a maintained by a non-profit corporation very similar to OpenMP. OpenACC provides an API very similar to OpenMP in approach which allows offload of processing to accelerators such as GPUs (as opposed to OpenMP which explicitly assumes that all processing will be undertaken on the host CPU or SMP system).

3.4.3 Other Programming Models

We now survey a series of other parallel programming models which have recently been proposed.

Thrust NVIDIA provide a C++ template library called Thrust [12] which provides some STL-like abstractions for straightforward kernels. This hides the complexity of kernel dispatch and simplifies copying between host and device memory. However, this is currently still tied to CUDA.

Accelerator The PGI Accelerator language and framework [25] focuses on abstracting the performance details of the target and automating much of the tuning, aiming for the high-performance computing domain. It is much higher level than CUDA and OpenCL, and has been the primary source of experience for the design of OpenACC, together with HMPP.

CUDA-lite CUDA-lite [29] was the first research project aiming to achieve better performance portability on CUDA programs. The user provides basic annotations about how to exploit parallelism and to deal with the offloading of live-in and live-out data. Based on these, the compiler performs loop tiling and unrolling to exploit temporal locality, local (shared) memory management, and attempts to help memory coalescing. Accelerator provides a similar abstraction level and compilation techniques.

HMPP The CAPS HMPP language and workbench [11] uses markup in a simliar way to Accelerator and provides back ends for both CUDA and OpenCL to allow cross-platform portability. It allows functions to be marked as codelets that can be executed on a parallel processor. Unlike Accelerator, it enables much finer control on the mapping and target-specific optimizations, which may also be seen a drawback in terms of abstraction level. Like Accelerator, it was a major source of inspiration for OpenACC.

Brook Brook is a very early model used to program GPUs for computation [5]. It is an extension to C that provides stream-based classes. Kernels can be written to read and write streams which can be converted to parallel code for efficient implementation.

Array Building Blocks (ABB) Intel provide a C++ template/API combination for optimizing parallel computation [13]. As the name suggests, the focus is on problems that can be expressed as operations on rectangular arrays and this is less applicable to more complex or irregular problems.

Thread Building Blocks (TBB). Thread Building Blocks is a more comprehensive library that originated at Intel but is now Open Source [14]. It provides library functions for a MPMD programming model with uniform address space.

Cilk Cilk was the pioneer MPMD task model and scheduling framework, provided by a simple C++ language extension for spawning functions as new threads [4]. The academic version featured a runtime system and a GCC-based compiler. It has more recently been acquired by Intel, which provides it as a simpler, language-level alternative to TBB.

GCD (Grand Central Dispatch) According to Apple, GCD comprises "language features, runtime libraries, and system enhancements that provide [. . .] support for concurrent code execution" for iOS and Mac OS X [3]. This is an MPMD model in which tasks are added to queues (FIFOs) and executed by a set of worker threads.

DirectCompute DirectCompute [20] is an SPMD model for graphics processors that is similar to CUDA and OpenCL but is more tightly bound with the DirectX graphics system and less suitable for general purpose programs.

Offloading to short-SIMD accelerators Short vector SIMD architectures have also been the subject of active programming model research and development. Offload [6, 7] is a set of high-level C++ extensions, a compiler and a runtime system to accelerate kernels on the Cell BE. Intel designed the similar, and contemporary, LEO programming model (Language Extensions for Offload), targeting its x86-based GPU architecture [28]. The Sieve programming model, geared towards automatic parallelisation of computational kernels via a novel semantics of delayed side-effects [18, 9] has also been used for offloading to accelerator cores of the Cell BE architecture [8].

Dynamic and managed languages GPGPU compilers, programming models, and active libraries have also been proposed for managed or dynamic languages, such as Matlab [26, 15] and Python (NumPy) [10]. These works facilitate the automatic (largely dynamic) extraction of offloaded kernels, and the automatic generation of communication code. Loop and array-oriented transformations are still needed to reduce the abstraction penalty and eliminate spurious synchronizations and communications. On the other hand, the objective is to take advantage from GPU accelerators but not to compete with the efficiency of lower level programming models.

3.5 Performance

The performance of a parallel program can be affected by many factors, ranging from the overall structure of the program down to the detailed choice of instructions and data structures.

3.5.1 Partitioning overheads

The first factor that affects performance is the way in which a program is partitioned between the serial and parallel processors. There is a significant overhead in running on a massively-parallel processor which may outweigh the benefits of parallel processing even if the algorithm itself runs faster. It is therefore important that the parallel code is only used for problems that can benefit from the acceleration that a parallel processor can provide, and that the overhead of using the parallel processor does not exceed any benefit that is gained.

One-off overheads Some one-off costs are incurred by any use of a parallel processor within a program. The system will have to locate the available devices, load the drivers for those devices and determine the basic capabilities. In some cases the parallel processor may be idle or even turned off, so there is a significant cost in time and energy to enable the device even before any computing is done. These overheads are lower for SIMD and MT systems because these features are usually built-in to the serial processor and cannot be disabled. Likewise, SMP systems often have all the processors enabled even if they are not being used so there is less one-off overhead. Models such as OpenMP and CUDA can hide most of the complexity of this process from the programmer but there is still an overhead involved.

Per-session overheads Additional costs are incurred every time a program switches between the serial code and one or more parallel tasks.

For systems with a segmented memory architecture, it is necessary to copy the input data from the serial processor's memory into the parallel processor's memory which can take a significant amount of time. The results must then be copied back at the end of the computation.

Where multiple tasks are being executed, it is sometimes possible to overlap the copying of data with computation. For example, the data for the first task is copied, and then while the task is executed the data for the second task is copied, and so on. If the program is compute-bound, this can reduce the copying overhead to a single copy in and out of the processor rather than a copy for each task; copying the data, however, still costs energy even if it does not add to the processing time.

Systems with shared memory do not need any copying but there will still be other costs. In some cases, it will be necessary to flush the caches to ensure that all processors see a consistent view of memory. In most cases, the different processors will have separate caches so the data will have to be copied, costing time and energy.

These overheads can be reduced by packing the input and output data as much as possible. For example, using single rather than double-precision numbers can significantly reduce the copying cost and may be acceptable as long as the computation itself is done with double precision. It is also important to run as many tasks as possible on the data once it is on the parallel processor rather than switching frequently between serial and parallel code.

Per-task overheads Additional overheads are incurred every time a task is executed, and for each thread withing that task. The per-thread cost can vary widely between different architectures and different implementations of those architectures. The worst case is for a full OS thread in an SMP system where there will be a significant amount of overhead in the operating system to create and destroy the thread. In this case it is almost always more efficient to create a pool of OS threads and use them to execute multiple tasks. With POSIX threads this has to be done by the programmer but models such as OpenMP, Cilk and Grand Central Dispatch will handle this process automatically.

Massively parallel architectures typically have hardware support for creating and deleting these threads. While this significantly reduces the overhead of running a task, it is still generally more efficient to combine computation into a few tasks where possible. However, there is a tension between reducing the number of tasks and having enough tasks to allow data copying to be efficiently overlapped with computation. The best choice of tasks often depends on the particular target hardware.

3.5.2 Processing performance

Once the problem has been appropriately partitioned between serial and parallel code it is important to make sure that the parallel code is as efficient as possible. This involves selection of algorithms that match the hardware and consideration of how the threads and tasks are synchronized.

Parallel code for SMP and MT devices tends to have the same performance constraints as serial code and so this section focuses on the performance of kernels running on massively-parallel devices with an SPMD model.

Optimizing scalar code The code for an SPMD kernel is similar to code for a standard CPU and many of the same optimizations apply but with different trade-offs. So, for example, it is generally helps to unroll loops but care must be taken to fit inside the instruction cache, which is usually smaller than a CPU instruction cache. Likewise it is important to eliminate common sub-expressions and cache values in registers, but using too many registers can restrict the number of threads that can be run simultaneously.

Compilers for SPMD programming models will perform many of these optimizations automatically but they are typically less mature than compilers for established CPUs, so programmers may need to perform more subtle or complex optimizations manually.

Built-in functions Massively-parallel devices usually have a number of built-in functions that are implemented directly in hardware at high speed. These are typically oriented towards graphics and geometry, such as sin/cos/tan and inverse square root. Using these functions can significantly improve the performance of a kernel but in some cases they are less accurate than the CPU equivalents so care must be taken to ensure the accuracy of the overall calculation. Built-in functions are exposed to the programmer and/or compiler as *intrinsics*.

Divergence Many massively-parallel architectures implement some form of warp-based SIMT which operates most efficiently when all the threads in a warp execute the same instructions at the same time. These architectures suffer a significant performance penalty when a branch is taken by some threads in a warp but not by others. The threads are said to *diverge*, and the processor will typically have to execute the two code paths in separate clock cycles, effectively halving the performance of the warp.

Divergence is particularly expensive if there are loops involved and some threads take more iterations than others. In this case a significant proportion of the execution time can be spent executing partially full warps, greatly reducing the efficiency.

If it is possible to duplicate calculations instead of taking a branch then this can improve performance on warp-based systems. To avoid branches these devices often implement some form of conditional execution by setting flags that control whether or not a series of operations will be performed, a technique known as *predication*. However this is not exposed explicitly in the parallel language so it is hard to take advantage of this for purposes of optimization.

The impact of divergence will depend on the particular architecture and even the particular device being used, so achieving the very best performance requires tailoring the parallel program to the specific target device.

Synchronization A key factor that affects performance is synchronization between tasks and threads.

Thread synchronization is required when one thread reads data written by another thread, but this will reduce performance if many threads are idle waiting for other threads to complete.

Atomic operations should be used where possible as these are implemented directly in hardware and typically do not block other threads while they execute. Critical sections are a good way of protecting data structures that cannot be implemented using atomic operations, but too much time spent in critical sections will reduce the time during which other threads can run.

Tile barriers can be used to synchronize all threads in a tile but this reduces the number of executable threads until all threads for the tile have reached the barrier. Massively-parallel processors rely on executing a large number of threads to be efficient and so barriers can significantly reduce performance. This is particularly true if there is significant divergence between the threads because all other threads will wait idly at the barrier until the longest-running thread has reached the barrier.

Some models support warp-level barriers which apply to a smaller number of threads. This is more efficient that a tile-based barrier but as the warp width is processor-specific this can make code much less portable.

MPMD systems do not have the same issues with warps but primitives such as critical sections and semaphores can be less efficient because the individual processors are less tightly integrated than those in massively-parallel architectures.

In order to get the best performance out of a massively parallel processor there need to be many threads running at the same time. If the program requires *task synchronization*, where one task must complete before the second starts there will be a period when only a few threads are active as the first task winds down and when the second task starts up. Where possible the program should schedule multiple overlapping tasks so that there are always a large number of threads to be executed. Models such as OpenCL provide mechanisms for scheduling sequences of tasks and describing the synchronization between them. This is also an area where models are advancing to allow more sophisticated ordering and to allow threads within one task to synchronize with threads in other tasks.

Overlapping serial and parallel code Most models allow tasks to be scheduled asynchronously to allow the serial code to continue executing while the parallel task is running. This can improve performance if there is a significant amount of information that needs to be pre-computed for future parallel tasks. However this may be less energy efficient than simply turning off the serial processor while the parallel processor is running, so a trade-off needs to be made.

The serial processor may be capable of running the parallel code, especially in SMP and MT systems, so in this case there is no trade-off to be made.

3.5.3 Memory performance

The compute performance of massively-parallel processors is so high that the memory performance becomes a major factor in overall program performance. The two key factors are the locality of the data and the degree of sequential access to that data.

Sequential access Accessing memory in sequential order is always good for performance but it is especially important on warp-based processors. When the threads in a warp access sequential memory locations the memory system can read memory in large blocks in a few cycles, especially when the block is aligned with a cache boundary. If the data for each thread is held in a structure then moving from AOS to SOA (see Section 3.3.2) can greatly increase performance because the data for each warp is laid out sequentially in memory. Arranging data so that memory is accessed this way can improve performance by factors of 2 or more.

Locality The large number of active threads means that a limited amount of cache is available for each thread. Therefore it is important to arrange the computation so that threads that operate on the same data are scheduled at the same time.

One way to improve locality is to use the image and texture classes that are provide by models such as CUDA and OpenCL. These are optimized for fast access for algorithms that operate on local regions of an image.

Tile memory In many cases the use of thread-local memory and tile memory can significantly improve performance. This memory is typically much faster than global memory and it is worthwhile copying data into that memory if it is going to be accessed multiple times. A typical coding pattern is to have each thread in a tile load part of the local memory and then execute a barrier, after which all the threads can access all the data in local memory. This can improve performance by a factor of 2 or more on architectures with fast local memory.

Impact of data organization on serial code In many cases reorganization of data to support parallelism will increase the performance of the parallel code at the expense of the serial code, so it is important to consider this when partitioning the program. For example the use of SOA rather than AOS may be more efficient for parallel code but it is likely to be less efficient for a serial processor.

3.5.4 Automatic optimization

There is some research activity around creating systems that automatically optimize parallel code for different platforms. This takes the form of a translator that converts a program in one model to a second program in the same model that is optimized for the target platform.

Models that use an intermediate language, such as CUDA and C++ AMP, can do a certain amount of optimization when they compile the intermediate form for the target device. They have limited ability to re-order operations to optimize memory access and re-code sections to match the specific instruction timings. However much of the abstract program information has been lost by the time the intermediate language is generated so there are limits to the level of optimization that can be achieved.

Models that allow run-time compilation of kernels, such as OpenCL, can perform broader optimizations with potentially greater performance improvements. This comes at the cost of a significant one-off overhead for compilation.

3.5.5 Coding time and efficiency

Another hidden cost is the time and expertise required to implement code using the parallel model and to optimize it for best performance.

Converting serial code to parallel code will typically take longer than writing the original serial code and the tools for debugging parallel code are significantly less mature than those available for serial code. It is also a more complex problem to produce working code because of the subtle timing issues that occur when many threads execute at the same time, further increasing the development time.

Likewise the process of optimizing parallel code is much more difficult that optimizing serial code. This is particularly difficult for a massively-parallel processor because of the range of factors that must be taken in to account. Whereas optimizing serial code typically produces code that is 2-5x faster, optimizing code for massively parallel code can change the performance by 10-50x.

If a program is going to be run a large number of times on a large data set then it is worthwhile investing a lot of time and energy into getting the most optimal program. But for smaller problems with short development timescales the time take to develop a fully efficient parallel application may add unacceptable delay to getting the desired results. The software team may also lack the level of expertise to write and optimized this code, further delaying the development and increasing costs.

Energy efficiency The amount of energy used by a computation is often as important as the time taken. None of the programming models has explicit support for 'energy efficient' computing, but the highest performing code is usually the most energy efficient because the parallel processor is active for less time. This is especially true of memory-bound problems where the energy use to read and write the data dominates the total energy cost. For other problems it is possible that data is being read and written more often than necessary but this is being hidden by the time taken to perform the computation. Massively-parallel processors are particularly good at reducing the impact of memory latency which has the unfortunate side-effect of hiding this inefficiency. In this case an algorithm that was more careful about memory access would be able to reduce the total energy even if it did not reduce the time.

3.6 Portability

Code portability is an important issue in programming in general; producing more portable code allows more value to be extracted from the development and debugging effort invested in it.

In the context of parallel programming models, there are various forms of portability that may be considered:

1. Portability between different classes of parallel processors.
2. Portability between different parallel processors of the same class.
3. Portability between operating systems.

In all cases there are also degrees of portability: *code portability* means that a program will run correctly on a different target, while *performance portability* means that a program will achieve acceptable performance on a different target. Since these programming models are mostly used where performance is of a high degree of importance, there is limited value to code portability if there is a large performance loss.

This section discusses portability in terms of the broad categories outlined above as it applies across the various programming models discussed in Section 3.4.

3.6.1 Portability between classes of parallel processors

The broad classes of programming models are MPMD and SPMD. These models map directly onto the underlying hardware architecture, so in general an SPMD programming model must be used when programming an SPMD processor, and an MPMD programming model when programming an MPMD processor.

A machine optimized for SPMD programs cannot generally run MPMD programs in a meaningful way, but the reverse is not true: a machine designed for MPMD programs can be instructed to run the same program on several processing elements at once, making it compatible with SPMD programming models. However, SPMD models do not allow the full flexibility of MPMD processors to be exploited.

Parallel models on serial processors OpenMP and OpenACC can both be used to create programs portable onto serial processors: their use of pragmas in C or special comments in FORTRAN allows non-compliant compilers to ignore the directives and compile a serial program.

However, both models also include explicit API calls which would cause incompatibility with a non-compliant compiler. To avoid problems in this area, preprocessor macros are defined by compliant compilers. OpenACC or OpenMP dependent code can then be guarded by checks for these macros. Therefore maintaining full compatibility with non-compliant compilers is possible but requires some care on the part of the programmer.

In other cases, a serial machine might be treated as a special case of an MPMD machine with only one concurrent thread. pthreads and OpenMP can both be used on serial systems although no performance benefit will be realized (and in fact performance may suffer due to thread switching overheads). This can sometimes be useful where each thread has a distinct self-contained task; the multi-threaded program can be easier for a programmer to reason about and work with than a single-threaded program where the various tasks are intermingled. This benefit needs to be balanced against the need to manage inter-thread communication and synchronization in the multi-threaded program.

The SPMD-oriented models may provide compatibility with serial processors as a side effect of supporting MPMD processors, which is described below.

MPMD models on SPMD processors As described above, the MPMD-optimized models (e.g. pthreads and OpenMP) do not offer any portability to SPMD processors. However, as an OpenMP program can be derived from a purely serial program very easily with a small number of pragmas, it may be possible to port such a program to OpenACC with little effort (in the simplest case by just changing `pragma omp` to `pragma acc`), but a program making more complex use of OpenMP directives may not be portable at all.

SPMD models on MPMD processors The SPMD-optimized models provide varying degrees of portability on to MPMD processors.

Renderscript offers the greatest level of code portability: at no point in Renderscript code does the programmer nominate a target device; rather, the runtime is responsible for deciding which device a given script will be run on. This makes it easier to generate a program which will run correctly on many devices, but can cause problems if the runtime makes a scheduling decision that results in poor performance. For example, a parallel script may end up running on the main processor because the programmer has inadvertently used a feature the SPMD processor does not support. This can be hard for the programmer to diagnose and fix.

OpenCL also offers portability onto MPMD processors, but under explicit control of the programmer. Kernels in OpenCL are submitted to queues, each of which is associated with a particular device. Depending on the OpenCL implementation, all processors in the system might be exposed as OpenCL devices, so porting from one device to another is as simple as changing the device associated with the queue. However, an OpenCL implementation might include various extensions which may not be supported to an equal degree on the various devices included in the system. For example, the CPU device might not support images, which may be essential to good performance on the GPU device (images are not an extension but device support for them is similarly optional).

CUDA, by default, requires a CUDA-compliant GPU in order to work. However, there are various third-party additions and tools which allow CPU-only operation, but the third-party experimental nature of these add-ons make them a poor choice for production programs. The situation is very similar for C++ AMP.

OpenACC code can always be run on a serial processor (by ignoring the OpenACC directives). There is nothing to prevent an implementation using the directives to split a kernel across the various processing elements of an MPMD processor.

Performance Portability

Performance portability depends on the algorithm used and how it has been optimized for SPMD processors. Many optimization techniques useful when programming SPMD processors make assumptions that are inapplicable on MPMD processors. In some cases the effect is benign, but in others it can result in significantly lower performance than an MPMD-native implementation. Many common optimizations have this problem, for example:

- Choice of a 'parallel algorithm' where this is more expensive on an MPMD processor (e.g. bucket sort).
- Repeated computation to avoid storing intermediate values.
- Reorganizing code to generate extra parallelism at the expense of extra steps.
- Coercing the domain to suit SPMD restrictions (i.e. 1D, 2D or 3D space).
- Reorganizing code to avoid thread divergence.
- Use of local memory resources that the MPMD processor does not necessarily have, incurring unnecessary copy overhead.

Most of these optimizations make fundamental changes to how the program is structured, and thus are independent of the programming model used. None of the programming models have the ability to fundamentally restructure programs to avoid these problems, so even though Renderscript (for example) can transparently run scripts on the 'best' device without code changes, it cannot make GPU-optimized code run as well on a CPU as CPU-optimized code would.

In some cases, relatively small changes can be made to make a program run reasonably well on both SPMD and MPMD processors. For instance, an OpenCL program could include two versions of a kernel and select the most appropriate one depending on which device is being used at runtime. This is much easier for the programmer than maintaining two different versions of the entire program.

3.6.2 Portability between different parallel processors of the same class

Many programs will run on a variety of different hardware configurations. Making a programming model choice that will limit portability will have to take account of this. The importance varies according to the application: for example, a high cost application for which users would be expected to purchase a dedicated hardware platform can make more restrictive choices than a low cost mass market program where the maximum possible potential audience is desired.

MPMD models The MPMD models considered above are open and hardware platform agnostic. Programs written for them might include non-portable optimizations (such as SIMD intrinsics) but there is nothing about the models themselves that is tied to a particular platform.

SPMD models OpenACC, OpenCL, Renderscript and C++ AMP are agnostic about the SPMD processor targeted. Relatively mature OpenCL can be expected to work with any suitable compute capable device. C++ AMP works on any device supporting DirectCompute. Renderscript and OpenACC in principle are portable to any device, but in practice they are less mature and implementations are less widespread.

CUDA stands out by being tied to a particular hardware vendor (NVIDIA), which limits the portability of CUDA programs. This has spurred development of tools such as CU2CL [19] which address this limitation by translating CUDA programs to widely supported OpenCL.

Performance portability As for portability between SPMD and MPMD, optimizations made for running on one device may be detrimental to performance on another. As this may involve fundamental algorithm changes, this is not something the programming model's compiler or runtime can easily address.

In some cases, particular devices might suffer more performance portability problems than others. For example, a vector GPU which uses SIMD execution units might require a kernel to use SIMD types and operations for best performance. If this kernel is run on a GPU that has scalar execution units, the SIMD operations can be emulated by running them in series one lane at a time, which would realize most of the available performance on the scalar GPU. On the other hand, a kernel written for the scalar GPU would not be able to exploit the performance potential of the vector GPU, unless the compiler is able to autovectorize the kernel.

3.6.3 Portability between Operating Systems

Most models are reasonably portable across operating systems.

As pthreads is a POSIX specification, it works on POSIX-based operating systems, which means most mainstream operating systems except Windows, although there is at least one third-party implementation available. Windows includes a native threading library with similar features, so pthreads code can be reasonably straightforwardly ported. C++11 includes native threading support which will work with any compliant compiler.

OpenCL is platform agnostic; use of OpenCL on a particular device is conditional on an OpenCL driver existing for that device and platform, but support is widespread.

OpenMP is platform agnostic; compilers exist for all common platforms.

Like OpenMP, OpenACC is platform agnostic. The compiler from PGI, a member of the OpenACC consortium, only works on Linux with NVIDIA hardware.

CUDA is available for Windows, Mac OS and Linux.

Currently. C++ AMP is only available on Windows, while Renderscript is only available on Android. In principle, both could be implemented on other platforms.

Bibliography

[1] C++ accelerated massive parallelism http://msdn.microsoft.com/en-us/library/hh265137.

[2] Android Developers. Renderscript, 2012. http://developer.android.com/guide/topics/renderscript/index.html.

[3] Apple Inc. Grand central dispatch (gcd) reference, 2012. https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[5] I. Buck, T. Foley, D. R. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[6] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload - automating code migration to heterogeneous multicore systems. In *HiPEAC*, pages 337–352, 2010.

[7] A. F. Donaldson, U. Dolinsky, A. Richards, and G. Russell. Automatic offloading of C++ for the Cell BE processor: A case study using Offload. In L. Barolli, F. Xhafa, S. Vitabile, and H.-H. Hsu, editors, *CISIS*, pages 901–906. IEEE Computer Society, 2010.

[8] A. F. Donaldson, P. Keir, and A. Lokhmotov. Compile-time and run-time issues in an auto-parallelisation system for the Cell BE processor. In E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, editors, *Euro-Par Workshops*, volume 5415 of *Lecture Notes in Computer Science*, pages 163–173. Springer, 2008.

[9] A. F. Donaldson, C. Riley, A. Lokhmotov, and A. Cook. Auto-parallelisation of Sieve C++ programs. In L. Bougé, M. Forsell, J. L. Träff, A. Streit, W. Ziegler, M. Alexander, and S. Childs, editors, *Euro-Par Workshops*, volume 4854 of *Lecture Notes in Computer Science*, pages 18–27. Springer, 2007.

[10] R. Garg and J. N. Amaral. Compiling python to a hybrid execution environment. In *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*, 2010.

[11] HMPP workbench: a directive-based multi-language and multi-target hybrid programming model `http://www.caps-entreprise.com/hmpp.html`.

[12] J. Hoberock and N. Bell. Thrust - parallel algorithms library, 2012. http://thrust.github.com/.

[13] Intel. Array building blocks, 2012. `http://software.intel.com/en-us/articles/intel-array-building-blocks/`.

[14] Intel. Intel threading building blocks for open source, 2012. `http://threadingbuildingblocks.org/`.

[15] Jacket home page. `http://www.accelereyes.com`.

[16] Khronos Group. Opengl, 2012. `http://www.opengl.org/`.

[17] Khronos OpenCL Working Group. The OpenCL specification, version 1.1, 2011. Document Revision: 44.

[18] A. Lokhmotov, A. Mycroft, and A. Richards. Delayed side-effects ease multi-core programming. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 641–650. Springer, 2007.

[19] G. Martinez, M. Gardner, and W.-c. Feng. CU2CL: A CUDA-to-OpenCL translator for multi- and many-core architectures. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 300–307, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Microsoft Corporation. DirectX 11 DirectCompute: a teraflop for everyone, 2010. `http://www.microsoft.com/en-us/download/details.aspx?id=16995`.

[21] Microsoft Corporation. How to download and install directx, 2012. `http://support.microsoft.com/kb/179113`.

[22] NVIDIA Corporation. *NVIDIA CUDA Programming guide 4.0*, 2011.

[23] OpenACC: Directives for accelerators. `http://www.openacc-standard.org`.

[24] OpenMP Architecture Review Board. Openmp application programming interface version 3.1, 2011. `http://openmp.org/wp/openmp-specifications/`.

[25] The Portland Group. *PGI Accelerator Programming Model for Fortran & C*, v1.3 edition, Nov. 2010.

[26] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of matlab programs for synergistic execution on heterogeneous processors. In *Proc. Conf. on Programming Language Design and Implementation (PLDI'11)*, June 2011.

[27] D. Robbins. POSIX threads explained, 2000. `http://www.ibm.com/developerworks/linux/library/l-posix1/index.html`.

[28] L. Seiler and D. C. et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, pages 18:1–18:15, 2008.

[29] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. Hwu. CUDA-lite: Reducing GPU programming complexity. In *Proc. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, Oct. 2008.

4 Compiler Optimization and Code Generation

This chapter describes the state-of-the-art research and tools related to compiler optimization and code generation as relevant to WP4 of the CARP project. Section 4.1 discusses the state-of-the-art parallelizing compilers, converting loop nests in plain C code into optimized GPU code. Section 4.2 surveys the most important work related to the compilation flow of the CARP project.

4.1 Automatic Parallelization Tools for GPUs

The three tools we survey in this section target NVIDIA GPUs using the CUDA programming model (see Section 3.4.1). Automatic parallelization tools for other GPU architectures are just starting to emerge; it is too early to evaluate them comparatively.

The CARP project will use CUDA when targeting NVIDIA devices, but favors OpenCL as a more open, portable programming model. Future WP4 deliverables will refer to the specific changes that have to be addressed when migrating from CUDA to OpenCL. The differences between CUDA and OpenCL are presented in Section 3.4.1. From the point of view of compilation, these differences are small. In contrast, differences among target architectures are of course very important (the ARM MALI architecture is widely different from NVIDIA desktop GPUs, for example). Harnessing these differences is one of the challenges to be tackled in the CARP project.

4.1.1 Polyhedral Model Overview

The polyhedral model is a high level mathematical representation of a program. It uses polyhedra and related mathematical abstractions to represent all dynamic instances of the statements surrounded by loops, and all dependences between any pair of dynamic instances of any pair of statements. It is applicable to the so called Static Control Parts (SCoP) of the program [7], which are formed by loops with affine manifest boundaries and bodies with statements accessing arrays with subscript expressions affine in the surrounding loop iterators and the problem parameters.

The polyhedral model assigns to each statement an *iteration vector*, with one dimension per loop surrounding the statement. The first dimension represents the outermost loop, the next dimension the following loop, and so forth. Each valid *iteration vector* for a statement represents one of the dynamic executions of the statement, corresponding to a valid combination of values for the loop iterators in the loops surrounding the statement. Each of these dynamic executions of a statement is called a statement *instance*. The set of all valid *iteration vectors* for a statement is called the statement *domain*.

In addition, the polyhedral model provides a compact representation of all the data dependences between all pairs of instances. For a given pair of statements S_1 and S_2, all the dependences from any instance of S_1 to any instance of S_2 form a relation between the domains of S_1 and S_2. This relation is called the *dependence relation* between S_1 and S_2.

A SCoP is then represented in the polyhedral model by a directed graph. Each node represents a statement, and is tagged with its domain. An edge going from S_1 to S_2 represents all the dependences from any instance of S_1 to any instance of S_2, and is tagged with the dependence relation from S_1 to S_2.

This representation has been very useful in finding correct transformations of the original SCoP, to expose parallelism [20, 21, 12] or to improve locality [12, 26, 3]. The usual way to handle code transformation in the polyhedral model is by the construction of a *schedule* for each statement. A *schedule* is an affine function that assigns to each point in the original domain of the statement a point in a target space. The target space is common for all statements, but several points of the original domains can be mapped to the same point in the target space. Usually this target space is seen as a *date* space, in which the first dimension is more important than the second dimension and so forth (like hours, minutes, . . .). In this way the schedule fixes a relative and partial ordering for all the instances. The transformed code is obtained by generating code that scans all the statement *domains* respecting the partial ordering specified by the schedule [6, 44]. However, it is also possible to build schedules in which some dimensions have different interpretations, for instance mapping to computing units (processors) instead of time units (*dates*). In addition, a non-injective schedule can be extended to an injective schedule by adding new dimensions that are in essence parallel.

The process to build the schedule depends on the goal, whether it is parallelism, locality, etc. Feautrier proposed an algorithm to find the set of all valid affine schedules for a given SCoP [20, 21]. The algorithm exploits the affine form of the Farkas Lemma, expressing the schedule for a statement as an affine combination of the inequalities in the dependence relation. It basically consists of building a system of equations on the coefficients of the affine combination, expressing the causality condition: if there is a data dependence from one instance to another instance, the latter must be executed at a later date than the former. The valid solutions can be obtained by solving this system of equations with any ILP solver. The set of all valid solutions form a convex $\mathscr{Z}$-polyhedron in the parameter space. To select among all valid schedules, usually the ILP problem is re-formulated as a minimization problem. Selecting a good minimization criterion to achieve a given goal is not trivial [20].

4.1.2 Loop Transformations and Code Generation for GPUs

We can split the methodologies followed by the tools selected for the evaluation, C-to-CUDA [5], Par4All [34] and PPCG [45], in four main stages:[1]

- *Exposing parallelism.* Both C-to-CUDA and PPCG use the polyhedral model for this stage. They select affine schedules for each statement to maximize the number of parallel dimensions in the target space. On the other hand, Par4All only a combination of loop fusion for locality enhancement and loop distribution to expose parallelism.
- *Mapping to the CUDA model.* The parallel dimensions are mapped to CUDA's *thread block* and *thread* identifiers.
- *Data mapping*. Basically, the tools decide where, in the memory hierarchy, to place all data that is being accessed. If some array is to be placed in shared memory,[2] specific code is added to take care of the transfers from global memory to shared memory.
- *Code generation.* For both the host and the kernels.

[1] We used the implementation of C-to-CUDA included in pluto 0.6.2, Par4All 1.3 and version 3bf74858 from PPCG's git repository.

[2] By *shared* memory here we refer to what was called *tile* memory in Section 3.2.5. In the CUDA programming model, tile memory is known as shared memory, and in this section we focus on polyhedral compilation techniques that target CUDA.

The first stage is similar for the three tools. However, some differences exist that affect performance. Par4All treats each original loop nest independently, generating a specific kernel for each one. On the other hand, C-to-CUDA and PPCG work on maximal SCoPs and apply some locality optimization criteria to generate the *schedule*. As a result, the different loop nests can be fused, at least partially. The generation of the *schedule* in both C-to-CUDA and PPCG is based on the algorithm proposed in PLUTO [14, 11], although they currently evolve independently. Par4All uses the schedule provided in the input source.

In the second stage, the three tools look for a *parallel band* in their respective *schedules*. In this context, a *parallel band* is a set of one or more consecutive parallel dimensions (loops). If more than one *parallel band* exists all three tools select the outermost band. The selected parallel band is then mapped onto CUDA's *thread block* and *thread* identifiers. Some important differences exist between the three tools in how this mapping is performed.

In Par4All at most three parallel dimensions are selected from the *parallel band*, which are then tiled. Tiling a dimension translates to strip-mining + interchange. Strip-mining divides the dimension into bands, or stripes of iterations [2]. It replaces the original dimension by two new dimensions, the first identifying the stripe and the second one identifying the point inside the stripe. Combined with interchange, these stripes form multidimensional tiles. The outer dimensions on stripes are called *tile dimensions* and the inner ones on points inside tiles is called *point dimensions*. The rest of the *schedule*'s dimensions are not modified. The *tile dimensions* are then mapped onto *thread block* identifiers. The *point dimensions* are mapped onto *thread* identifiers. Furthermore, unique to Par4All is the support for dynamic tile size selection..

On the other hand, both C-to-CUDA and PPCG also tile the dimensions following the first two parallel ones. In some cases, the additional sequential *tile dimensions* (that translate to loops in the kernel code) before the parallel *point loops*, may increase the chances of exploiting data reuse in the shared memory at the expense of introducing some thread synchronizations. However, we should highlight that no analysis is performed to evaluate the effect on performance of these extra tilings and reorderings, for the code being mapped.

In addition, both C-to-CUDA and PPCG take the tile sizes, *thread block* sizes and *grid* sizes as user parameters. This translates to the necessity of applying extra tilings on the parallel *tile* and *point* dimensions, to wrap those dimensions on the actual *thread block* and *grid* sizes selected by the user. This has its advantages and drawbacks. On one hand, the user can fine tune these parameters for a given algorithm. However, a non-expert user, without knowledge on how the tool works, would probably need an exhaustive search to find those optimal parameters. On the other hand, an extra degree of complexity is introduced for the final code generation stage, as the *schedule* dimensions increase with these extra levels of tiling, making it sometimes inefficient. In our opinion, the tool should select the values for these parameters, after an analysis on the code being compiled: for the designer that decision is indeed too tedious and complex to make. Using default values independently of the code being compiled, like Par4All does, is also suboptimal.

The tools also differ in the *data mapping* stage. Par4All does not consider the exploitation of reuse in the shared memory or the register file. All accesses are performed directly on the global memory. For the most advanced Fermi architectures, it configures the memory hierarchy to use as much hardware cache as possible, relying on it to exploit data reuse.

C-to-CUDA takes a different systematic approach, every array is mapped to the shared memory. Thanks to the extra tilings on the dimensions after the parallel band, this is many times possible. However, as no analysis was performed on the code to perform these tilings, it can lead to invalid codes that use too much shared memory. In addition, mapping an array to shared memory when it does not exhibit inter-thread data reuse reduces performance, as we have the extra accesses to the shared memory without any compensation. We will see that this translates to a poor performance in many cases.

PPCG is, in this aspect, the most advanced tool. It performs the following analysis to decide if a given array is to be accessed directly from the global memory or if it is to be mapped on the shared memory or the registers file:

- If the data can be put in registers and there is reuse, then put it in registers.
- Otherwise, if the data can be put in shared memory and either there is reuse or the access is non-coalesced, then put it in shared memory.
- Otherwise, put it in global memory.

Finally, the tools also differ in the *code generation* stage. Only Par4All and PPCG generate both host and kernel codes, including all code to manage CPU-GPU transfers. In addition, while all three tools are capable of generating parametric code, Par4All is much better at detecting that a seemingly parametric input program is not actually parametric due to its powerful interprocedural analysis. This has the advantage of generating simpler kernel codes, significantly reducing the number of dynamic instructions executed by the kernel. On the other hand, as mentioned before, the extra tilings introduced to leave the user the control over the tiling parameters lead to very complex kernel codes, frequently inefficient. This could be solved by providing the tools with a more sophisticated algorithm in the *Mapping to the CUDA model* stage.

4.2 Related Work

As seen in Chapter 3, GPGPU programming models such as CUDA and OpenCL expose application developers to a relatively abstract model of an SPMD many-core architecture. However, platform-specific details remain largely exposed: it is critical to utilize the hardware resources efficiently in order to achieve high performance.

So far, the practice of GPU programming has been dominated by heroic porting and tuning efforts, resulting in a large number of application-specific studies. We survey different aspects of the compilation challenge for GPUs: performance modeling and tuning considerations; programming models; polyhedral approaches to GPU compilation; exploration of the low-level issues related to vectorization for short-SIMD architectures such as the ARM Mali GPU; and finally linking these issues with the complete code generation flow of the CARP project through the concept of split compilation.

4.2.1 Performance Modeling and Tuning

Porting applications is complicated by the heterogeneity of GPU-accelerated platforms with distributed memory, and the restrictions of the GPGPU programming models. Ryoo et al. have shown that the optimization space for CUDA programs can be highly non-linear, and highly challenging for model-based heuristics, performance modeling, or feedback-directed methods [39, 38]. G-ADAPT is a compiler framework to search and predict the best configuration for different input sizes for GPGPU programs [28]; it starts from already optimized code and aims to adapt the code to different input sizes.

The previous methods are not fully automatic or integrated into a GPGPU compiler. Progress towards fully automatic performance tuning have been achieved by the Crest [43] and CHiLL [17] projects.

Rather than targeting a low-level programming model like OpenCL or CUDA, Apricot [35] focuses on the automatic insertion of offload constructs, targeting LEO. It rounds some of the performance tuning difficulties by performing selective offloading at runtime. As the hardware and tool flow mature, we believe the performance tuning aspects will become more and more important and challenging.

4.2.2 Polyhedral Compilation

The polyhedral model has been the basis for major advances in automatic optimization and parallelization of programs [15, 20, 21, 27, 13]. Many efforts have been invested to develop source-to-source compilers such as PoCC [33], PLUTO [14] and CHiLL [17], which use a polyhedral framework to perform loop transformations. Nowadays traditional compilers such as GCC, LLVM [23] and IBM XL make use of polyhedral frameworks to compile for multicore architectures. Progress is also underway to extend the applicability of polyhedral techniques to dynamic, data dependent control flow [8].

With the emergence of GPUs, the polyhedral model has been applied to develop efficient source-to-source compilers for them. Baskaran's C-to-CUDA [5] is the first end-to-end, automatic source-to-source polyhedral compiler for GPU targets. It is based on PLUTO's algorithms and explicitly manages the software controllable part of the memory system. However, it remains a prototype and handles only a small set of benchmarks. Building on Baskaran's experience, Reservoir Labs developed its own compiler based on R-Stream [26], which introduces a more advanced algorithm to exploit the memory hierarchy. However, the paper glosses over many of the details, and no source code is available to inspect. Gpuloc is a variation of the algorithm proposed by Baghdadi et al. [4], which is also based on PLUTO. It uses a ranking based technique [24] to transfer data to and from shared memory, which results in inefficient accesses to memory that limit the overall performance. Moreover, at least one of the proposed implementations wastes memory and limits the GPU computation power. In addition, it is not in development anymore. Par4All [34, 3] is an open source initiative developed by the HPC Project to unify efforts concerning compilers for parallel architectures. It supports the automatic integrated compilation of applications for hybrid architectures including GPUs. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, which also involves polyhedra; this allows Par4All to perform powerful interprocedural analysis on the input code. CHiLL developers also extended their compiler to generate GPU code — they introduced CUDA-CHiLL [37] during 2011. The framework does not perform an automatic parallelization and mapping to CUDA but instead offers high-level constructs that allow a user or search engine to perform such a transformation. PPCG (Polyhedral Parallel Code Generator) is a relatively new tool [45]. PPCG tries to overcome previous limitations and efficiently manages the memory system and exhibits a very robust behavior.

Unfortunately we were unable to obtain a copy of Reservoir Labs' R-Stream [26] or CUDA-CHILL [37] to provide a deeper assessment of these tools' capabilities.

4.2.3 Automatic Vectorization

Fine-grain data level parallelism is one of the most effective ways to achieve scalable performance of numerical computations. It is pervasive in graphical accelerators. Automatic vectorization for modern short-SIMD instructions has been a popular topic, with target-specific as well as retargetable compilers for the ARM Neon, Intel AVX an SSE, IBM Altivec and Cell SPU; all of these works had a successful impact on production compilers [47, 9, 31, 32]. Exploiting subword parallelism in modern SIMD architectures, however, suffers from several limitations and overheads (involving alignment, redundant loads and stores, support for reductions and more) which complicate the optimization dramatically. Automatic vectorization was also extended to handle more sophisticated control-flow restructuring including if-conversion [41] and outer-loop vectorization [32]. Classical techniques of loop distribution and loop interchange [46, 2, 42] can dramatically impact the profitability of vectorization.

Leading optimizing compilers recognize the importance of devising a cost model for vectorization, but have so far provided only partial solutions. Wu et al. conclude [47] regarding the XL compiler that:

> *"Many further issues need to be investigated before we can enjoy the performance benefit of simdization for a wide range of applications. The more important features among them are more advanced handling of non stride-one accesses and the ability to decide when simdization is profitable. Equally important is a better understanding of the interaction between simdization and other optimizations in a compiler framework."*[3]

Likewise, Bik stresses the importance of user hints in Intel's ICC vectorizer profitability estimation [9], to avoid vectorization slowdowns due to:

> *"the performance penalties of data rearrangement instructions, misaligned memory references, failure of store-to-load forwarding, or additional overhead of run-time optimizations to enable vectorization."*

On the other hand opportunities may be missed due to overly conservative heuristics. These state-of-the-art vectorizing compilers incorporate a cost model to decide whether vectorization is expected to be profitable. These models however typically apply to a single loop or basic-block, and do not consider alternatives combined with other transformations at the loop-nest level.

Loop-nest auto-vectorization in conjunction with loop-interchange has been addressed in prior work [1, 46, 2]. However, this was typically in the context of traditional vector machines (such as Cray), and interchange was employed as a preprocessing enabling transformation. Overheads related to short-SIMD architectures (such as alignment and fine-grained reuse) or GPUs were not considered until recently. Realignment and data-reuse were considered together with loop-unrolling [40] in the context of straight-line code vectorization. A cost model for vectorization of accesses with non-unit strides was proposed in [31], but it does not consider other overheads or loop transformations. The most advanced cost model for loop transformations-enabled vectorization was proposed by Trifunovic et al. [42]. It is based on polyhedral compilation, capturing the main factors contributing to the profitability of vectorization in the polyhedral representation itself.

[3]Simdization refers to a program transformation resulting in the use of short-vector SIMD instructions. The access stride refers to the address increment in between two consecutive accesses to a given array.

4.2.4 Split Compilation

Deferred compilation refers to a compilation process that is decomposed into several steps along the "lifetime" of the program. Traditional tool chains using byte-code representations distribute the roles among offline and online compilers. Verification and code compaction are typically assigned to offline compilation, while target-specific optimizations are performed by online compilation.

Split compilation reconsiders this notion: it allows a single optimization algorithm to be split into multiple compilation steps, transferring the semantic information between different moments of the lifetime of a program through carefully designed (byte-code) language annotations. A split compilation process may run expensive analyses offline to prune the optimization space, deferring a more educated optimization decision to the online step, when the precise execution context is known. Many JIT compilation efforts tried to leverage the accurate information obtained through dynamic analysis to outperform native compilers. Split compilation is a concrete path to gather the best of both worlds.

This definition of split compilation easily extends to all steps of a program lifetime: offline/ahead-of-time compilation, linking, installation, loading, online/run time, and even the idle time between different runs. Each step brings additional knowledge about the run-time environment (shared libraries, operating system, processor, input values), opening the door to new classes of optimizations.

During the transformation from high-level language, optimizers gain increasing knowledge about the final run-time environment, but they also gradually lose semantic information: arrays become pointers, typed integers become 32-bit amorphous values, etc. There have been many attempts to help compilers with information it cannot derive by itself, including pragmas in OpenMP, or explicit multistage programming like DyC [22] or [a]C# [16] to drive partial evaluation [19]. On the other hand, LLVM [25] is a widely used framework which demonstrated that lifelong program analysis and transformation can be made available to arbitrary software, and in a manner that is transparent to programmers. For deferred compilation to be effective, high-level information must be propagated while lowering the program representation. This is one of the purposes of PENCIL, designed in WP3.

We are not aware of any split compilation approach for automatic parallelization in general, or for GPUs in particular. On the other hand, the approach has seen some successes in the more specific area of automatic vectorization. Vapor SIMD is a combined static-dynamic infrastructure for vectorization, capable of supporting diverse SIMD targets, across a wide range of vectorizable kernels, with performance comparable to monolithic compiler vectorization [30, 36]. It leverages the split compilation principles to facilitate the support of heterogeneous architectures with SIMD accelerators, and the acceleration of portable code over a diverse ecosystem of such heterogeneous architectures. The CARP project will build on this approach in the design of the PENCIL intermediate language.

Clark et al. follow a different path, also related to split compilation, in their Liquid SIMD method [18]. There, a static compiler auto-vectorizes the code, but then scalarizes it to emit standard scalar instructions. The scalar patterns are later detected by the hardware, if equipped with suitable SIMD capabilities, resurrecting vector instructions and executing them. We also propose to use a static auto-vectorizing compiler, but instead of confining it to a scalar ISA (thereby limiting the vectorizable patterns), we will introduce PENCIL, a richer intermediate language to convey additional, explicit information from the static compiler to the JIT.

Bocchino and Adve proposed Vector LLVA [10], a common vector format that can be automatically translated to different platforms, focusing on manual programming using their format, and demonstrating the translation of a few kernels to AltiVec, SSE, and RSVP. As in LLVA, Intel's Array Building Blocks [29] also introduces general vector idioms to support manual vectorization. While these works focus on target-agnostic manual vectorization, our work focuses on a target-agnostic auto-vectorization engine. In that sense, the two approaches are orthogonal and complementary. Our approach is especially advantageous over the manual approach in situations involving several alternatives for vectorizing the code, each preferable for a different SIMD target. A programmer will pre-determine one vectorization alternative when vectorizing the code manually, whereas using the offline auto-vectorizer, as in our approach, the generated byte-code will convey enough information to allow the JIT to choose the right vectorization scheme for a given SIMD platform.

Bibliography

[1] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Tr. on Programming Languages and Systems*, 9(4):491–542, 1987.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[3] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static compilation analysis for host-accelerator communication optimization. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'11)*, LNCS. Springer-Verlag, Oct. 2011.

[4] S. Baghdadi, A. Grösslinger, and A. Cohen. Putting automatic polyhedral compilation for GPGPU to work. *Proc. Compilers for Parallel Computer (CPC)*, 2010.

[5] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, Volume 6011 of Lecture Notes in Computer Science, pp. 244-263. Springer.*, 2010.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, France, September 2004.

[7] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30, College Station, Texas, Oct. 2003. Springer-Verlag.

[8] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (CC'10)*, number 6011 in LNCS, Paphos, Cyprus, Mar. 2010. Springer-Verlag.

[9] A. J. C. Bik. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[10] R. L. Bocchino, Jr. and V. S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *VEE*, pages 46–56, 2006.

[11] U. Bondhugula. PLuTo: An automatic parallelizer and locality optimizer for multicores. `http://pluto-compiler.sourceforge.net/`.

[12] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *LNCS*, pages 132–146, 2008.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43:101–113, June 2008.

[14] U. Bondhugula, J. Ramanujam, and et al. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI) 08*, 2008.

[15] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.*, 24:421–444, May 1998.

[16] W. Cazzola, A. Cisternino, and D. Colombo. [a]C#: C# with a Customizable Code Annotation Mechanism. In *Proc. of the 10th Symp. on Applied Computing*, pages 1274–1278, Santa Fe, NM, 2005.

[17] C. Chen, J. Chame, and M. Hall. A framework for composing high-level loop transformations. Technical report, USC Computer Science, 2008.

[18] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *HPCA'07*, pages 216–227, Washington, DC, USA, 2007.

[19] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Computing Surveys*, 1998.

[20] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, 21:313–347, 1992.

[21] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *International Journal of Parallel Programming*, 21:389–420, 1992.

[22] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report UW-CSE-97-03-03, Univ. of Washington, 1999.

[23] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly: Polyhedral optimization in llvm. In *First Intl. Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[24] A. Grösslinger. Preciese management of scratchpad memories for localising array accesses in scientific codes. *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, 2009.

[25] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, USA, Mar. 2004.

[26] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[27] A. Lim. Improving parallelism and data locality with affine partitioning. Master's thesis, Stanford University, 2001.

[28] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proc. IEEE International Parallel & Distributed Processing Symp.*, 2009.

[29] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Array building blocks, a retargetable, dy-

namic compilation framework. In *Proc. Intl. Symp. on Code Generation and Optimization (CGO'10)*, Chamonix, France, Apr. 2010.

[30] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD – auto-vectorize once, run everywhere. In *Proc. Intl. Symp. on Code Generation and Optimization (CGO'11)*, Chamonix, France, Apr. 2011.

[31] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proc. Conf. on Programming Language Design and Implementation (PLDI'06)*, 2006.

[32] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short SIMD architectures. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'08)*, October 2008.

[33] PoCC: the polyhedral compiler collection. `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`.

[34] H. Project. Par4All automatic parallelization. `http://www.par4all.org`.

[35] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. Apricot: An optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In *Intl. Conf. on Supercomputing (ICS'12)*, June 2012.

[36] E. Rohou, S. Dyshel, D. Nuzman, I. Rosen, K. Williams, A. Cohen, and A. Zaks. Speculatively vectorized bytecode. In *HiPEAC*, Heraklion, Greece, Jan. 2011.

[37] G. Rudy, M. M. Khan, M. Hall, C. Chen, and C. Jacqueline. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. Hwu. Optimization principles and application perform- ance evaluation of a multithreaded GPU using CUDA. In *Proc. Symp. on Principles and Practice of Parallel Programming (PPoPP'08)*, Jan. 2008.

[39] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, J. A. S. S. Ueng, and W.-M. Hwu. Optimization space pruning for a multithreaded GPU. In *Proc. Intl. Symp. on Code Generation and Optimization (CGO'08)*, Oct. 2008.

[40] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'02)*, Sept. 2002.

[41] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proc. Intl. Symp. on Code Generation and Optimization (CGO'05)*, March 2005.

[42] K. Trifunović, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT'09)*, Raleigh, North Carolina, Sept. 2009.

[43] S. Unkule, C. Shaltz, and A. Qasem. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction (CC'12)*, number 7210 in LNCS. Springer-Verlag, 2012.

[44] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS 3923, pages 185–201, Vienna, Austria, Mar. 2006. Springer-Verlag. Classement CORE : A, nombre de papiers acceptés : 20, soumis : 71.

[45] S. Verdoolaege. PPCG: Polyhedral parallel code generator. `http://freecode.com/projects/ppcg`.

[46] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[47] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated Simdization framework using virtual vectors. In *Intl. Conf. on Supercomputing (ICS'05)*, 2005.

5 Quantitative Cost Analysis

In this chapter we survey the current state-of-the-art in quantitative cost analysis as relevant to WP5 of the CARP project. Section 5.1 discusses several models that estimate the performance of parallel programs when executed on GPGPUs. These models are based on a static analysis of the program at hand, or their control flow graph. An alternative approach is to develop an abstract machine model of the GPU at hand, and derive probabilistic statements about execution time, resource usage and so on. To that end probabilistic reward models can be used, and analysed using model checking. This is surveyed in Section 5.2. A complementary view is probabilistic program analysis where average cost analysis is possible using a semi-automated deductive approach. This is discussed in Section 5.3, and is a more general appraoch not specifically tailored to GPGPUs. Finally, Section 5.4 is concerned with obtaining estimations of the memory consumption of programs. These are based on the so-called memory footprint.

5.1 Code-Based Average Cost Analysis

A cardinal goal of the CARP project is to improve efficiency of GPU programs so that they run faster and consume less energy. Opportunities to optimize the execution time of programs are exposed by constructing a *performance* model of the application. As GPU computing is still not quite mainstream, there has been little research in this regard; nevertheless, this section reviews the current state of the art.

Performance Models. NVIDIA ship a so-called *occupancy calculator*, which calculates the ratio between the number of active warps to the maximum number of warps on a multiprocessor of the GPU. The calculation is based on the parameters of a GPU kernel, in particular the thread block size, the size of shared memory, and the number of registers. As the ratio tends towards 1.0, it is claimed better performance can be expected because warps can be scheduled while long-latency global memory requests of other warps are pending.

However, as Hong and Kim [20] elucidated, the correlation between high occupancy and fast execution time is not that straightforward. They instead propose a more refined model of GPU kernel execution which effectively considers how warps are scheduled on a particular multiprocessor, analogously to how a pipeline model of a CPU is constructed for instruction throughput. The effect of barrier synchronization and coalesced/uncoalesced memory accesses on warp progress are also considered. These abstract models of the GPU produce a system of equations from which an execution time is yielded for a specific GPU configuration. For a number of micro-benchmarks, their results show that the model predicts the actual execution time with relative accuracy.

The original framework nevertheless had a number of limitations, which have been remedied to some extent in recent work [30]. First, since older-generation GPUs did not incorporate caches, there were no provisions for cache effects on timing. As this trend is over (e.g. NVIDIA's Fermi has a hardware-managed cache), a very simple extension is proposed. But a major stumbling block is that details of the cache configuration (e.g. cache line size) and its operation (e.g. cache line replacement policy) are normally proprietary, without which it is impossible to model with any degree of accuracy. Second, some parts of the analysis now operate at a lower level, namely with the binary rather than an intermediate representation (PTX, the virtual instruction set architecture of NVIDIA). The rationale is that, because there is a final optimization stage while assembling PTX into binary, the true measure of performance comes from analyzing that which runs on the bare metal.

Baghsorkhi et al. [5] have also recently proposed a performance model for GPU code that is based on an extension of a program's control flow graph called the Work Flow Graph (WFG); nodes represent instructions while edges represent memory accesses with high latencies. This is a more detailed model in that it is able to handle diverging control flow between threads and memory bank conflicts (this is where there are multiple requests for data from the same memory bank, leading to a serialization of memory accesses). With respect to several relatively complex GPU kernels, such as FFT and prefix sum, their predictions are in close proximity to the actual execution times. The biggest weakness is that the model is constructed in the early phases of compilation, before the full array of optimizations has been applied.

Nugteren and Corporaal [28] present the 'boat hull' model based on the 'roofline' model [31] (so called because of the shape of the region where a program's performance is expected to lie in). The roofline model estimates performance by assuming that a program is either memory-bound or compute-bound, depending on its operational intensity (measured in operations per byte) and available hardware. The performance range given by the roofline model can be very wide. For example, if a program is memory bound, the roof of model corresponds to the most favourable memory access pattern (typically sequential), while the ceiling to the most unfavourable one (*e.g.* random), the difference in performance between which may be an order of magnitude on some older GPUs. The boat hull model predicts performance within a tighter range by taking into account the memory access pattern of a particular algorithm class. For image processing and computer vision applications, the predicted performance is within 8% of the measured performance on different CPUs and GPUs.

Power Models. Hong and Kim [21] proposed a refined model that predicts both power and performance for a given GPU program (kernel). The total power consumption is comprised of the *idle power* (which is consumed even when no kernel is running), and the *runtime power* (which is consumed by running the given kernel). The runtime power is comprised of the power consumed by off-chip memory, and by architectural components of a GPU core such as various functional units and on-chip memories. An architectural component is characterised by the maximum power which is obtained empirically by running synthetic microbenchmarks stressing the component. A kernel is characterised by how often it accesses an architectural component per unit of time (*access rate*). The model further considers the impact of the runtime power on the temperature, and of the temperature on the static power. To achieve reasonable accuracy (with the geometric mean of the prediction error of 2.5% for the microbenchmarks used for training and 9% for simple kernels), the model is fairly specific to NVIDIA's G200 architecture (*e.g.* considers sharing of resources between streaming multiprocessors, warp-based execution, segmented memory organisation, *etc.*). Kernels are characterised only by the rates of accessing architectural components, without taking into consideration effects of *e.g.* control-flow divergence and locality.

5.2 Quantitative Model Checking

Conventional model checking as introduced in the 1980's by Clarke, Emerson, Queille and Sifakis [14, 29] is a fully automated technique to verify properties of interest of the system under consideration. However, most often real-world systems tend to involve stochastic and timed behavior. For example, hardware components inside a system may fail with a given probability or it might be of crucial interest that the system satisfies certain timing constraints. As the models for qualitative model checking abstract from numerical phenomena, there is virtually no way of analyzing quantitative properties.

Quantitative model checking extends conventional model checking in a way that enables formal analysis of models involving quantitative aspects. Besides the aforementioned correctness analysis this also includes performance and dependability – often combined as *performability* – measures such as the time needed to reach a given goal [18]. Quantitative model checking is a push-button technique which is typically restricted to finite state models. This does, however, not mean that it is not possible to model real-valued phenomena. On the contrary: there exist formalisms that support real-valued quantities such as time despite having a finite state representation.

We will now focus on real-time models, because performance analysis of GPGPU programs is one of the primary goals of the CARP project.

Quantitative Model Classes. There exist various models to formally capture timed or stochastic behavior.

Alur and Dill proposed Timed Automata (TA) [3] as a means to model the behavior of real-time systems over time. Formally, TA are a finite-state representation of an underlying infinite transition system. The state space of the timed automaton is given by a set of locations in which time can pass unless the invariant of the state is violated. The time that passed increases the values of the finitely many clocks associated with the automaton. Transitions between locations are guarded by conditions over these clocks and may thus only be taken when all conditions are met. Upon taking a transition, it may reset any number of clocks before entering the target state.

While timed automata elegantly reflect real-time behavior, choices during the execution of the original system can only be modeled by non-determinism. In the absence of information about these choices that may be appropriate, but otherwise this represents a rather harsh abstraction. Drawing inspiration from Markov Decision Processes (MDPs), Kwiatkowska et al. [26] proposed Probabilistic Timed Automata (PTA), which allow for specifying discrete probability distributions over transitions. MDPs are an extension of Discrete-Time Markov Chains (DTMCs) in that they are not restricted to one probability distribution over successor states in each state, but to a non-deterministic choice over probability distributions. That means that MDPs themselves can model non-determinism as well as probabilistic behavior, but lack the support for continuous phenomena.

Dropping the feature of non-determinism, DTMCs can be extended to Continuous-Time Markov Chains (CTMCs). Instead of probabilities each transition between states in a CTMC is equipped with a so-called rate. The information these numbers give is twofold. First, they indicate the likelihood of each transition being taken in relation to the other “competing” transitions and secondly, they give rise to a probability distribution over the time the current execution stays in the source state. More precisely, the sojourn time in a certain state is given by an exponential distribution with the mean value equal to the sum of outgoing rates. Put differently, CTMCs are able to capture time as a continuous quantity as well as the probabilistic behavior of the modeled system. Additionally, as the rates are the mean value of a distribution, uncertainties in the timing behavior of the system can be elegantly reflected.

However, CTMCs are not suitable to model non-deterministic behavior, as their behavior is of purely stochastic nature. That is, using rates to "emulate" non-determinism corresponds to making assumptions about the scheduling of actions, which is not suitable in many cases. To cope with this, Hermanns proposed Interactive Markov Chains (IMCs) [19]. This model class extends CTMCs by *interactive* states. In these states, an evolution of the system non-deterministically chooses one of the possible successor states and moves there.

Often, not only the stochastic and timed-behavior of a process is of interest, but also an additional different, yet related quantitative measure. For example, the energy-consumption until a certain condition holds or the expected number of rounds a protocol has to be run in order to be successful. This can be reflected in the discussed Markov models by attaching *rewards* to states. The reward gained in a state during an execution is equal to the reward value of the state multiplied by the time that is spent in that state. Quantitative model checking could now for instance assert that the average reward gained during executions leading to a particular target state group is lower than some given bound.

Measures-of-Interest. Probably the most prominent measure of interest is *time-bounded reachability*. Under this category fall, e.g., properties of the form "what is the probability that the system reaches a certain configuration within t time units". For instance, for a model of a network configuration protocol (e.g. the IPv4 zeroconf protocol), one might ask how likely it is that the protocol finds a suitable configuration in, say, 2 seconds.

Another property of interest is the long-run average. Consider a workstation cluster, which serves queries from a queue. In order to guarantee that enough users can be served, it could be of crucial importance that — in the long run — the system is available for requests at least 80% of the time.

Using the aforementioned reward structures, even more complicated measures can be computed. As an example, reconsider the network protocol example. Assuming the model attaches rewards to its states corresponding to the energy consumption at a given time, not only can we compute the likelihood that the protocol finds a suitable configuration without using more energy than a certain bound, but we can even compute the expected amount of energy needed spent on finding such a configuration.

If a system incorporates non-determinism, the measures of interest can take several values depending on a concrete resolution of non-determinism. In this case, quantitative model checking approaches the solution conservatively. That is, it computes extremal values for the best and the worst resolution of non-determinism, respectively. Consequently, the measure of interest will lie in the derived interval for every possible resolution of non-determinism. If the resulting interval is too wide, this is rather a problem of the model as for the specified model there exist resolutions of non-determinism that achieve the extremal values. If that behavior, however, cannot be observed in the real system, the model has to be refined.

In practice, all measures described here are formulated succinctly using temporal logics like TCTL [2], PCTL [8] and CSL [4] and according reward extensions. Furthermore, this has the advantage of unambiguity, as it leaves no room for interpretation of the semantics of a property.

Brief Description of Main Model Checking Algorithms. For the TA-based models, further analysis is typically preceded by the construction of a region graph. This graph consists of a finite number of clock regions, each of which represent a set of clock valuations that behave identically with respect to the property to verify. As this graph is potententially intractably large, the verification of certain properties can be done by using the – possibly smaller – zone graph.

[1] This preprocessing step is then followed by either a standard qualitative model checking process (for TA) or a quantitative analysis for PTA.

At the heart of quantitative model checking lie numerical methods to compute the measure of interest. For purely probabilistic models the solution of a set of linear equations generally suffices. [8] Usually, the transition probabilities are stored in a representation of a matrix, which – after some modifications – reflects the linear equations to solve. Although the exact solution of this system can be obtained in polynomial time in the number of states of the model, it is infeasible already for moderately sized models due to enormous state space sizes. In practice, the use of numerical iterative solution techniques proved to be feasible, especially considering the advanced data structures used for the storage of the transition probability matrix such as variants of Binary Decision Diagrams (BDDs) [24].

However, for continuous-time models such as CTMCs an analysis may become more intricate. While non-timed properties may be directly verified over the *embedded* DTMC, the solution for timed properties is given by a set of integral equations. This can – at the cost of being computationally expensive – be numerically approximated. Alternatively, Baier et al. proposed a modification of the CTMC such that the verification of the given property boils down to transient analysis of the CTMC [7].

Including non-determinism in the model generally makes it harder, as it is not clear beforehand which resolutions of non-determinism will yield the extremal values. Instead of finding the solution to this problem by an iterative technique as before, the problem may be formulated as an Linear Programming problem and solved via e.g., a Simplex-based method or value iteration.

If the CTMC model involves rewards, the solution procedure can become even more difficult and involve further steps such as discretisation [18, 6].

Available Tool Support. There exist several stochastic model checking tools, out of which PRISM [25], MRMC [23] and LIQUOR [13] are the most prominent ones. PRISM[1] is arguably by far the most widely used. It supports a variety of modeling formalisms as well as additional optimization techniques. Among the supported model classes are DTMCs, MDPs, CTMCs as well as PTAs and the appropriate (branching) temporal logics. It offers three different model checking engines building on symbolic and sparse data structures and the corresponding techniques to solve the problem at hand. Besides, it features a high-level modeling language, making it easy to model structured systems. While being particularly feature-rich, PRISM, however, is not capable of analyzing IMCs. Based on ideas to reduce the analysis of an IMC to a variation of a problem over a simpler model [15], a prototype of an Interactive Markov Chain Analyzer (IMCA[2]) is currently under development for which first experiments show encouraging results.

5.3 Probabilistic Program Analysis

Probabilistic programs can be used as a model of real software where we abstract from concrete implementation details and instead quantify the likelihood that some computation is performed or not. This allows for average case estimations regarding runtime or power consumption.

Formally, probabilistic programs are imperative, sequential programs enriched with a random choice statement. This means, there are assignments, loops, conditional and *non-deterministic*

[1] `http://www.prismmodelchecker.org/`
[2] `http://www-i2.informatik.rwth-aachen.de/imca/`

choices. In addition, there is a probabilistic choice operator that allows to choose between two subprograms probabilistically with some probability p and $1-p$ respectively [27]. Such programs can be categorized with respect to the data and expressions that they are allowed to work with. For example, the variables of a program might be only Boolean (or more generally finite domain variables) or they might be real-valued (or infinite domain in general). The guards of loops and branching statements as well as the right-hand side of assignments is composed of arithmetic expressions. These expressions can be restricted to be only linear, for example, or they can be generalized to be polynomial. Finally, the probability parameter of probabilistic choice can be a fixed, real number in $[0,1]$ or some fixed, symbolic parameter, e.g. p, or an expression over the program variables like $\frac{1}{x}$ which is assumed to evaluate to a number in $[0,1]$ at runtime.

If we restrict the programs to finite domain variables, then probabilistic model checkers such as PRISM [25], MRMC [23] and LIQUOR [13] can effectively analyse various properties of such programs. However they require that probabilities are given explicitly. PARAM [16] can work also with programs where probabilities are fixed but unknown constants. If integer variables are allowed in models, the state space can become infinite. Then, again for explicit probabilities, the tool PASS [17] can compute reachability probabilities.

There is a need to consider more general probabilistic programs than those treated by the automated tools above. Hence different approaches are being developed. One is to reason about probabilistic programs not in an operational way but in terms of *expectation transformers* [27]. This allows to represent the programs and their properties in a finite way regardless of the underlying state-space size or the involved expressions.

An expectation is function that maps a program state, i.e. the evaluation of the variables, to a real value. Expectations are used in the same way as a *predicates* are used as program annotations in Dijkstra's *wp*-calculus: Given a program we relate an annotation at the end of the program, called a post-expectation, to an annotation at the beginning, called pre-expectation in the following way: the post-expectation is a random variable that is evaluated at the end of the program, the pre-expectation is the expectation[3] of this random variable with respect to to the distribution that is generated by the execution of the probabilistic program in between.

Using expectations we can express what is the probability of a certain event, say "at the end of the program, the value of x is between 4 and 4.5" or we can express what is the expected number of, say iterations that a loop does before it terminates.

From a theoretical point of view this is a very powerful vehicle that allows to analyze complex sequential probabilistic programs. However this expressivity comes at a price. Only for (trivial) i.e. loop-free programs, there is an easy way to compute a pre-expectation for every post-expectation. For loops the pre-expectation is defined as the solution of a fix-point equation over expectations. So there is no hope to find a fully automated tool that would perform the analysis of such programs in general.

However, as for non-probabilistic programs, invariants are used to find a lower bound on the aforementioned fix-points. The problem of discovering such invariants spawned a research branch on its own that deals with *automated invariant generation*. For non-probabilistic programs this research is already going on for many years while invariant generation for probabilistic programs is a new topic. One of the first approaches is described in [22]. A first prototype tool called PRINSYS (PRobabilistic INvariant SYnthesiS) is currently being

[3]Actually, due to non-determinism the program generates a set of distributions and we take the *least* favorable one, i.e. the one that will minimize the expectation.

developed[4]. Currently, a *linear* invariant for *linear* probabilistic programs with *one* loop can be generated semi-automatically. This works as follows: a user enters a template, that is a linear expectation with unknown parameters and PRINSYS automatically tells the user which evaluations of the template parameters yield an invariant expectation. The next immediate challenge is to treat templates that are non-linear.

5.4 Determining Memory Footprints

The memory footprint of a piece of code is the part of computer memory accessed by that code while performing its operations. The notion of footprint is central in local reasoning. Separation logic provides mechanisms whereby a specification can concentrate on only the cells accessed by a program, while allowing the specification to be used in wider contexts via a special rule called the frame rule (see also Chapter 6). In the context of automatic verification and program analysis this suggests, when considering a code fragment in isolation, to try to discover assertions that describe the footprint, rather than the entire global state of the system. This is the key idea that makes analysis based on separation logic viable: the entire global state can be enormous, or even unknown, where one can use much smaller assertions to talk about the footprint.

There are two main articles in the literature which automatically determine memory footprints.

Intra-Procedural Case. A method for determining memory footprint in the intra-procedural case has been developed in [10]. This method, called *footprint analysis*, defines a shape analysis that is able to discover preconditions (as well as postconditions) and builds on the work on shape analysis with separation logic.

Footprint analysis symbolically runs the program forwards, collecting heap information for each node of the control flow graph in the usual way of shape analysis. However, when a dereference to a potentially-dangling pointer is encountered, that pointer is added into a special "footprint assertion", which describes the cells needed for the program to run safely. When starting the analysis with the empty heap in the initial footprint assertion, then the analysis will find the collection of safe states. These are the states that do not lead to a dereference of a dangling pointer or other memory fault.

Bi-abductive Inference. The intra-procedural footprint analysis has been extended to the general case in [11]. This new analysis is based on a new notion called *bi-abduction* which combines the idea of *abductive inference* and *frame inference*.

When reasoning about the heap, abductive inference discovers the memory needed to execute a command safely. It can be defined as follows: given (separation logic) formula H' representing a portion of memory needed to execute a command safely, and H representing the memory available before executing the command, compute a formula A such that $H * A \vdash H'$ holds. In other words, A is the missing part of memory which when combined with the available memory H is enough to represents the needed memory H'.

Frame inference is a dual notion. It is defined as: given (separation logic) formulas H and H' compute a formula F such that $H \vdash H' * F$ holds. In practice, frame inference compute the part of memory that is not involved in the execution of a command. An algorithm for inferring frames was introduced in [9].

[4]`http://www-i2.informatik.rwth-aachen.de/prinsys/`

Bi-abductive inference is then defined as: given (separation logic) formulas H and H' compute a frame F and an anti-frame A such that $H * A \vdash H' * F$ holds. Bi-abduction provides a way to automatically compute (approximations of) footprints of commands and preconditions of procedures. In particular, bi-abduction is the main ingredient allowing for an analysis method where pre/post specs of procedures are inferred independently of their context. This has opened up a way to design compositional shape analyses for sequential [11], and recently concurrent programs [12]. Compositional analysis has a great ability to scale since procedures are analyzed in isolation.

Bibliography

[1] R. Alur. Timed automata. In *CAV*, pages 8–22, 1999.

[2] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous time Markov chains. In *CAV*, pages 269–276, 1996.

[5] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM.

[6] C. Baier, L. Cloth, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Performability assessment by model checking of Markov reward models. *Formal Methods in System Design*, 36(1):1–36, 2010.

[7] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time markov chains by transient analysis. In *CAV*, pages 358–372, 2000.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[9] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[10] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In H. R. Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 402–418. Springer, 2007.

[11] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009.

[12] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In Z. Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2009.

[13] F. Ciesinski and C. Baier. Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Third International Conference on the Quantitative Evaluation of*

Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA, pages 131–132. IEEE Computer Society, 2006.

[14] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP*, pages 169–181, 1980.

[15] D. Guck, T. Han, J.-P. Katoen, and M. R. Neuhäußer. Quantitative timed analysis of interactive Markov chains. In *NASA Formal Methods*, pages 8–23, 2012.

[16] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PARAM: A model checker for parametric Markov models. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 660–664. Springer, 2010.

[17] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PASS: Abstraction refinement for infinite probabilistic models. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 353–357. Springer, 2010.

[18] B. R. Haverkort, L. Cloth, H. Hermanns, J.-P. Katoen, and C. Baier. Model checking performability properties. In *DSN*, pages 103–112, 2002.

[19] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002.

[20] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[21] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[22] J.-P. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-invariant generation for probabilistic programs: - automated support for proof-based methods. In R. Cousot and M. Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 390–406. Springer, 2010.

[23] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.*, 68(2):90–104, 2011.

[24] M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[25] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[26] M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theor. Comput. Sci.*, 282(1):101–150, 2002.

[27] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2004.

[28] C. Nugteren and H. Corporaal. The boat hull model: adapting the roofline model to enable performance prediction for parallel computing. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 291–292, New York, NY, USA, 2012. ACM.

[29] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[30] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[31] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

6 Static Verification

In this chapter we give an overview the current state-of-the-art in static verification as relevant to WP6 of the CARP project. WP6 focuses on static verification of accelerator applications, including verification of code written using the GPU programming models discussed in Chapter 3, and techniques for verifying memory safety of accelerator programs. The work package also has a focus on the use of dynamic instrumentation to expose bugs that have been detected using verification; we are aware of no prior work on this topic so do not discuss it further here.

We begin with an overview of the current state-of-the-art in the static verification of accelerator programs (Section 6.1). As will be apparent from this overview, research in this particular sub-domain of static verification is currently in its infancy. In particular, no attempts have been made to adapt techniques for the analysis of pointer manipulation (e.g., separation logic, shape analysis, and hyperedge replacement grammars), or more advanced techniques for the analysis of concurrent programs based on Hoare logic [69], separation logic [109] and model checking [32], so that they can be applied to the verification of accelerated software.

Following the overview of the state-of-the-art in static verification of accelerator programming, we describe in turn the state-of-the-art in techniques for analysis of pointer-manipulating programs (Section 6.2) and multi-threaded concurrent programs (Section 6.3).

6.1 Static Verification of Accelerator Programs

The state-of-the-art in verification of accelerator programming is necessarily limited in scope due to its relatively recent introduction. All techniques proposed so far are geared towards the analysis of either CUDA or OpenCL kernels (see Section 3.4.1 for a detailed overview of each of these programming models), or programs that execute on the Cell Broadband Engine processor. The focus is on adapting well-understood techniques for *sequential* program analysis to the setting of accelerator programs.

6.1.1 Formal semantics for GPU programming models

Before we discuss the verification efforts for GPGPUs, it is important to mention that static verification depends on a formal semantic, i.e., a mathematical description of the execution of a program.

Formal semantics for programming models in the vein of CUDA and OpenCL have been studied. Semantics in which the threads can be arbitrarily interleaved are presented in [90] and [87], the differences between the two are minor, although the former also specializes this to a semantics in which all threads run in lock-step. Pure lock-step semantics are presented in [63] and [62]. The former defines a semantics for a high-level language in which accelerator programs may be written; the latter defines a semantics for the byte-code that may be run on an NVIDIA GPU.

We note that it is important for any new programming language to have strong semantic foundations, and that the CARP work will involve specifying formal semantics for the new PENCIL intermediate language which we will design.

6.1.2 Verification of GPGPU Programs

We now categorize the formal verification efforts attempted to date according to the verification technique adopted. The relevant techniques are: pairwise verification of threads and symbolic execution. For completeness, and because Task 6.4 of CARP is on the integration of static verification with dynamic instrumentation, we also discuss recent dynamic verification approaches.

Pairwise Verification of Threads. Pairwise verification of threads has as its starting point the observation that any data race in a GPGPU program will be due to *two* threads accessing the same memory location without proper synchronization, where at least one of the threads performs a write. The observation was first made by Li and Gopalakrishnan and their co-workers [88]. As they report, considering threads in a pairwise fashion allows them to scale to programs that were beyond the capabilities of their previous bounded model checking approach [89] (see below).

Pairwise verification has been adopted by Li and Gopalakrishnan [88, 94] and by Tripakis et al. [119], and is the basis of current work on GPU kernel verification by the CARP team at Imperial College London. Besides considering threads in a pairwise fashion, another crucial observation made by these approaches is that in the case of data races it suffices to consider program executions between barriers, which are the main means of synchronization in GPGPU programming. This considerably reduces the complexity the logical formulas that need to be checked, allowing for the verification of more complex programs.

The work of Tripakis et al. [119] only considers data races and is limited to verifying absence of races only in programs where all loops have fixed bounds. They, however, claim to be working on an extension of their method which mitigates the loop bound limitation.

The work of Li and Gopalakrishnan [88], implemented in their tool PUG, is more advanced and allows for verification of programs with unknown loop bounds, using techniques for automatic loop invariant inference. Their technique also allows loop invariants to be specified manually for cases where inference fails. The technique exploits the fact that CUDA kernels cannot exhibit recursion, and inlines all procedure calls. This means that discovery of procedure pre- and post-conditions is not necessary. Although the technique is necessarily incomplete, Li and Gopalakrishnan claim that the technique can be highly successful in the case of GPGPU programs, as most loops in these programs follow certain fixed patterns.

As well as checking for data races, Li and Gopalakrishnan allow for checking of arbitrary functional properties by means of adding additional assertions to programs. In their later work [94], they also consider performance problems in GPGPU programs. In particular, so-called *bank conflicts* and *memory coalescing errors* are considered. In the case of bank conflicts, multiple threads at the same time try to access a memory bank shared between these threads, which requires the serialization of these accesses, reducing performance. In the case of memory coalescing errors, multiple reads or writes to memory by different threads to the global memory of the GPGPU cannot be combined into one memory operation, also reducing performance.

Our current work on GPU kernel verification furthers the work of Li and Gopalakrishnan in several ways. We have devised a precise technique for checking the correctness of barrier synchronization (see Section 3.2.8) based on *predicated execution* of GPU threads. We are designing a technique for loop invariant inference using the Houdini algorithm [55], by identifying common GPU programming idioms and capturing the correctness criteria for these idioms in invariant generation rules. Our techniques are implemented in a prototype tool, GPUVerify, which builds on the Boogie verification system [5].

Bounded Model Checking. In bounded model checking, the loops in a program are unrolled to a certain bound. Next, a logical formula is constructed for the program and a property the program needs to satisfy, where the formula only considers the unrolled part of the loop. Finally, automated theorem proving is applied, as in the case of contract-based verification. Remark that this method is necessarily incomplete, as a counterexample to a property may only be found by unrolling the loop further than the chosen bound.

A bounded model checking approach to GPGPU program verification is described by Li and Gopalakrishnan and their co-workers [89], i.e., the authors of PUG. As in the case of PUG, the focus is on data races and it is remarked that the method can be extended to arbitrary program properties by adding assertions to the programs being verified. However, unlike PUG and due to this work actually pre-dating PUG, no use is made of the observation that it suffices to consider only two threads when trying to detect data races. Instead, so-called *partial order reduction* is used to reduce the number of thread schedules that need to be considered, which, as reported in [88] is less effective than restricting attention to two threads.

Symbolic Execution. In symbolic execution, a program is actually executed. However, part of the program is kept in symbolic form. For each path through the program that can be taken constraints on the symbolic data are generated. These constraints can be used to generate concrete test cases (ensuring high code coverage). Moreover, the constraints on two programs that are claimed to be equivalent can be generated and compared to actually prove equivalence.

The route of proving equivalence of two programs is taken by Collingbourne et al. with their tool KLEE-CL [36]. KLEE-CL uses symbolic execution to establish the equivalence between a plain C or C++ version of a program and a OpenCL version of the same program. As reported in [36], the technique is very effective for finding differences between plain programs and GPGPU implementations of the same programs.

Li and Gopalakrishnan and their co-workers [90] use their tool GKLEE to generate test cases with high code-coverage. In addition, during symbolic execution GKLEE attempts to detect memory races and bank conflicts (see above). Moreover, GKLEE also detects what is sometimes called *barrier divergence*, which means that some barrier synchronization points in GPGPU program are not hit by all threads; something which is not allowed on current GPGPU architectures. To limit the number of execution paths that are taken, GKLEE depends on the lock-step semantics of threads, instead of using a more general interleaving semantics. The lock-steps semantics suffices, as the current GPGPU hardware is also based on this execution model.

Dynamic Verification. In dynamic verification, instrumentation code is added to programs to be able to observe certain unwanted program behavior, e.g., data races can be detected by adding code that records the memory locations accessed by different threads. The instrumented programs are run on concrete inputs to see if the unwanted program behavior occurs in practice. Note that adding instrumentation code is likely to reduce the performance of a programs and, hence, should be kept to a minimum to ensure reasonable execution times.

Boyer et al. [23] use code instrumentation to detect data races and bank conflicts in GPGPU code. However, they do no attempt to keep the instrumentation to minimum, which results in significant slowdowns.

Improving on the work by Boyer et al., Zheng et al. [127] use static analysis techniques to reduce the instrumentation to a minimum in the case of data races. Moreover, they introduce special data structures that reduce the overhead of the instrumentation. In this way, a significant performance gain is obtained compared to the work of Boyer et al.

Finally, although strictly speaking a static verification technique, Leung et al. [87] use dynamic verification of a single program execution to establish that this execution is race free. Moreover, using static analysis of the GPGPU program they try to establish that the memory accesses in the program do not depend on the concrete data that was used as input. Combining both analyses they have a theorem that states that if the execution was race free and memory accesses were independent of the concrete inputs, then the program is race free on every input.

The technique of Leung et al. is highly effective to show the race freedom of a large number of GPGPU programs. However, there seems to be no obvious way to extend the method to programs where memory accesses do depend on the data that is used as input.

6.1.3 Verification of Cell Programs

The Cell Broadband Engine processor [72] is a heterogeneous multicore architecture (see Section 3.1.1) consisting of a host processor, the *power processor element* (PPE) and eight accelerator processors, the *synergistic processor elements* (SPEs). The system has *segmented* memory (see Section 3.2.5): each SPE has its own portion of scratch-pad memory which can be accessed without contention. A thread running on an SPE must explicitly copy data to/from main memory from/to its scratch-pad memory using direct memory access (DMA) operations. In order to hide latency, an SPE thread can launch many DMA operations simultaneously, and must ensure that concurrent DMA operations do not *race*: if two DMA operations are simultaneously pending, either issued by the same or different threads, and if these operations access overlapping regions of memory, then neither operation can modify this overlapping region of memory, otherwise undefined results will be computed.

A static technique for detecting races between DMA operations issued by a *single* SPE thread has been proposed, based on bounded model checking [48, 49]. In this approach, DMA operations are tracked by instrumentation variables, and assertions are added to the program under analysis such that a failed assertion indicates a DMA race. This work also shows that the *k*-induction method [118, 46] can be used to prove *absence* of DMA races in programs that use multi-buffering as a latency hiding technique. This DMA race analysis technique is implemented in the SCRATCH tool [50], which builds on the bounded model checker CBMC [33].

Another technique for static verification of programs that use DMA operations, based on separation logic [20, 19], is discussed in Section 6.3.1.

6.2 Static Verification of Pointer Manipulating Programs

Automatic software verification has seen an upsurge of interest in recent years. This is exemplified by tools such as SLAM [4] and ASTRÉE [17], which have been used to verify properties of special classes of real-world software, e.g., device drivers and avionics code. Crucial in this reinvigoration of automatic software verification has been the employment of methods from static program analysis which have the advantage to lessen annotation burden (e.g., by automatically inferring loop invariants and procedure summaries).

Hoare logic [69] has been the traditional basis for program verification, where one typically wishes to show that for any possible input in some unbounded domain, the program behaves in a certain way. Several tools based on Hoare logic exist and have been successfully applied to sequential programs written in realistic programming languages (e.g., ESC/Java2 [35], Spec# [7], and Krakatoa [97]).

Pointer Manipulation. Advances in automatic verification have happened also for pointer manipulating programs. This has traditionally been a persistent trouble area that has hindered verification-oriented program analysis to be applied to a wider range of real software. Traditional shallow pointer analyses, which infer dereferencing information of bounded length, often do not give enough information for verification purposes. For example, for automatically proving that a device driver manipulating a collection of nested cyclic linked lists, does not dereference `null` or a dangling pointer, the analysis technique needs to be able to look unboundedly deep into the heap. Various software verification approaches have been tailored or extended to pointer programs such as extensions of Hoare-style reasoning using e.g., separation logic [109], abstract reachability analysis using regular model checking [22], shape analysis [114, 115], and dedicated extensions of temporal logic approaches [41]. However, until very recently, automatic verification of pointer programs could only be applied to tiny programs; for evidence of this in the case of shape analysis see, e.g., [25, 111, 112, 96, 40, 21].

6.2.1 Shape Analysis and Separation Logic

Shape analyses [114, 115] aim to be accurate in the presence of deep-heap updates — they go beyond aliasing or points-to relationships to infer properties such as whether a variable points to a cyclic or acyclic linked list. Separation logic [109] is an extension of Hoare Logic for reasoning about programs that access and mutate data held in computer memory. It is based on a special conjunction operation $*$ called the *separating conjunction* and on program-proof rules that exploit separation to provide modular reasoning about programs. Thanks to the separation conjunction, separation logic supports *inductive definitions* providing natural descriptions of mutable data structures and *frame axioms*, which state what does not change. Specifying what does not change is essential to prove many programs involving pointer manipulations. These features of separation logic together enable specifications and proofs for pointer programs that are dramatically simpler than was possible previously [13, 105, 109, 43].

Separation logic has been combined with shape analysis to obtain scalable automatic verification techniques. Prominently among these is Space Invader/Abductor [42, 12, 125, 27], an automatic shape analysis tool based on separation logic aiming at the verification of real software. The driving forces behind Space Invader/Abductor are: On the one hand, the idea of local reasoning in separation logic [75] that allows specifications to concentrate on the *footprint*: the cells touched by a command. On the other hand, there is the notion of bi-abductive inference [27] (see also Chapter 5) which allow the verification to be compositional and therefore to scale to very large programs (millions of lines of code).

A number of other verification systems for separation logic have been developed (some of which take some inspiration from Space Invader), varying from the level of automation to the kind of programming features they handle [79, 52, 103, 15, 57, 14, 95, 30, 122, 43].

6.2.2 Hyperedge Replacement Grammars

The verification method for pointer-manipulating programs based on hyperedge replacement grammars (HRGs) introduced in [110, 65, 81] aims at abstracting the (possibly) infinite state space arising during the analysis of heap-manipulating programs into a finite one. It employs an intuitive *graph-based model of heap structures* where vertices can be interpreted as heap objects while labeled edges (i.e., pairs of vertices) visualize selector references between the objects. Additionally, a mapping of program variables to vertices is required to link the program to the heap. Thus pointer-manipulating operations correspond to graph transformations.

For the purpose of obtaining a finite state space, *hyperedge replacement grammars (HRGs)* [51] are utilized to shrink and expand parts of the heap. The production rules of the HRGs reflect the data structures that occur in the program, such as lists, trees, lists of trees, etc. The key idea is to employ hyperedges to represent abstract parts of a heap, e.g., a singly-linked list of arbitrary length. Thus, heap representations are partially concrete and partially abstract. To guarantee soundness of the abstraction approach, additional requirements are to be fulfilled by the HRGs. We are able to list these requirements and show that there exist transformation procedures that output suitable HRGs only [81].

The basic approach has been adapted towards Java programs and implemented in the abstraction framework *Juggrnaut* [64] featuring e.g. object types, null pointers, method stacks and local as well as static variables. The tool has been successfully applied to various case studies including the Lindstrom variant [91] of the Deutsch-Schorr-Waite algorithm [117], which is considered a benchmark for pointer program analysis [93, 22].

6.2.3 A Brief Comparison

We briefly compare techniques for analyzing pointer programs based, on the one hand, on separation Logic and shape Analysis [115] and, on the other hand, on HRGs.

As mentioned earlier, separation logic is an extension of Hoare logic with inductive definitions to describe the shape of heap parts. It is classically employed in the form of Hoare-style verification using annotations of programs where decidability of entailment is essential. Whereas this problem is undecidable in general, decidable sub-logics for lists and trees have been developed in [13, 14]. There are several separation-logic tools such as Space Invader [125, 27] (supporting linear data structures), Smallfoot [14] (aiming at (doubly) linked lists and trees), jStar [43], and Verifast [76] (allowing general user defined predicates and data structures). The main advantage of these tools is their scalability [125].

The HRG-method is based on exhaustive (abstract) state space exploration and can support a rich set of user-defined data structures. As pointed out by Dodds and Plump in [44], there exists a strong correspondence between inductive definitions in separation logic and the nonterminals that are used in the abstract heap representations based on graphs. This also means it would be interesting to investigate if this relationship can be exploited e.g., to allow for usage of user-defined data structures given by HRGs in separation logic contexts while preserving decidability of entailment.

Shape analysis is based upon three-valued logic, where the value “don’t care” is used to deal with loss of information caused by abstraction. Heaps are described by shape graphs, where nodes, indistinguishable by properties expressed as predicated in the logic, are summarized. Typical predicates are shape properties like reachability, cycle membership, etc. Most of these are derivable from the heap graph representation [38] and are implicitly provided in the HRG state space exploration. Whereas in shape analysis all nodes satisfying the same predicates are summarized, in the HRG approach nodes that constitute a well-defined (fragment of a) data structure are collapsed.

6.3 Static Verification of Concurrent Programs

We give an overview of techniques used to statically verify concurrent programs. We discuss both techniques based on Hoare logic and model checking techniques.

6.3.1 Methods Based on Hoare Logic

This section first of all considers techniques based directly on Hoare logic [69]. This is followed by an introduction to concurrent separation logic, including a discussion of permissions. Finally, we consider interval temporal logic and some approaches to the verification of concurrent data structures. We expect that these techniques will also form the basis for the concurrency-related aspects of the verification of accelerator programs.

Extensions Based on Hoare Logic. Owicky and Gries present a language for parallel programming with a primitive construct for synchronization and mutual exclusion [106]. They develop a method for verifying concurrent programs by proving non-interference between parallel executing processes. Assertions about processes have to be proven invariant under parallel execution of other processes; this is the notion of *interference freedom*. One major drawback of the Owicki-Gries approach is its non-compositionality: Changes in one component can effect the proofs of all other parallel components. In addition, one has to prove for all individual atomic program steps that there is no interference, quickly resulting in an enormous blow-up in the number of proof obligations.

To tackle the compositionality problem, Jones [82] developed the Rely-Guarantee method. This method adds rely- and-guarantee-conditions to programs that run in an interfering environment. The Rely-Guarantee method is a very good choice for modeling concurrent programs with interference, but the specifications are usually very complex because they describe the entire global state.

Ábrahám et al. [1, 2] present an assertion proof system for multi-threaded Java, extending the method of Owicki and Gries. They reason about Java-specific concurrency issues like synchronous message passing, dynamic thread creation, shared-variable concurrency via instance variables, and coordination via re-entrant synchronization monitors.

Jacobs et al. [77, 80, 78] propose a programming model that prevents data races and deadlocks and supports local reasoning in the presence of object aliasing and concurrency. They implement a verifier for programs developed according to their model in a custom version of the Spec# programming system.

Concurrent Separation Logic. Both the Owicki-Gries and Rely-Guarantee method can result in very complex specifications as a result of describing global states. Resource separation has been recognized as a solution to control the complexity of process interactions and reducing the possibility of time-dependent errors. This will also be essential for the verification of accelerator programs.

O'Hearn [104] describes the idea of separating resources using separation logic. The main ideas behind O'Hearn's approach are ownership and separation: Ownership properties describe that a code fragment can access only those portions of a state that it owns. Separation properties describe that at any time during execution the state can be partitioned into parts that are owned by individual processes and a common part that is protected by mutual exclusion techniques.

Brookes [26] presents a trace semantics for a language of parallel programs that share access to mutable data. He introduces a resource-sensitive logic for partial correctness. His logic allows proofs of parallel programs in which ownership of critical data, such as the right to access, update, or deallocate a pointer, is transferred dynamically between concurrent processes. Brookes extends O'Hearn's work by allowing nested resource declarations and nested parallel compositions. He also introduces a formal definition of resource contexts.

Vafeiadis [120] presents a new soundness proof for concurrent separation logic in terms of a standard operational semantics. The proof gives a direct meaning to concurrent separation logic judgments, which can easily be adapted to accommodate extensions such as permissions and storable locks, as well as more advanced program logics.

Vafeiadis and Parkinson [121] propose a system that combines separation logic with the Rely-Guarantee method. They give a natural description of interference using a relation as in Rely-Guarantee, while reasoning locally as in separation logic.

Gotsman et al. [58] present a resource-oriented program logic that is able to reason about concurrent heap-manipulating programs with an unbounded number of dynamically allocated locks and threads. Their logic is inspired by concurrent separation logic. They demonstrate that the proposed logic allows for local reasoning about programs for which there exists a notion of dynamic ownership of the heap part using locks and threads. The logic resolves some of the limitations of separation logic by assuming a bounded number of non-aliased and pre-allocated locks and threads.

Jacobs et al. [76] present the VeriFast program verifier, which is based on separation logic approach for the specification and verification of safety properties of pointer-manipulating imperative programs. The safety properties to be verified are specified as annotations in the source code, in the form of pre-conditions and post-conditions expressed in separation logic.

Hobor et al. [70] define a modular operational semantics for Concurrent C minor. They present a concurrent separation logic with first-class locks and threads, and prove its soundness with respect to the operational semantics. Their Concurrent C minor operational semantics is designed to connect to Leroy's optimizing (sequential) C minor compiler [86]. They use modal logic to build a model that advances time as locks are acquired/released and an oracle machine to be able reasoning about sequential programs with concurrent communication elements.

Permissions. Separation logic describes the exclusive ownership of parts of the heap. Each heap cell may be owned by only one thread. This property reduces complexity of assertions significantly but is not enough in practice. In practice, parts of the heap may safely be shared between concurrent threads under some conditions. This is the main reason for extending separation logic with permissions. Similar techniques will also be necessary to reason about accelerator programs sharing access to a heap location.

Boyland [24] introduces the concept of fractional permissions for checking interference. This concept permits reads of a shared location with fractional permissions whereas writes require complete permissions. Intuitively, multiple reads do not conflict while accessing some shared location. Boyland gives an operational semantics and defines a permission type system. He proves that checkable parallel constructs do not interfere.

Leino et al. [83] present a verification methodology for concurrent, object-based programs which enforces the absence of data races and deadlocks. The methodology uses fractional permissions, which allow them to support fine-grained locking and multi-object monitor invariants, sharing and unsharing of objects, and concurrent reading. The methodology is implemented as a translator from the language Chalice to the verification language Boogie. It has been used to verify several examples. Chalice is explained from the user prospective in [84]. Boogie is described in more detail by Barnett et al. [6].

Haack et al. [74, 61] present a program logic for reasoning about multi-threaded Java-like programs with concurrency primitives such as thread creation, thread joining and re-entrant object monitors. The logic is based on concurrent separation logic. Concurrent reads are supported through fractional permissions. In order to distinguish between initial monitor entrances and monitor reentrances, an auxiliary variable keeps track of the multiset of currently held monitors.

Heule et al. [68] describe a methodology for supporting fractional permissions while allowing the user to work at the abstract level of read and write permissions. The methodology has also been implemented in the verification tool Chalice. Two main kinds of permissions are used: A full permission and a read permission. Read permissions are abstract, i.e., they do not specify a concrete mathematical fraction.

Bornat et al. [18] add the notion of counting permission, modifying the definition of a heap: A heap is now a partial map from addresses to values with permissions. While the fractional permissions model allows permissions to always be split off and combined, the counting permission method uses a central permissions authority (the main thread) that holds a source permission, annotated with the number of read permissions that have been split off. The split-off read permissions cannot be split any further and only a source with no split-off children can give total read/write/dispose ownership.

Botincan et al. [20, 19] present a method and a tool for proving memory-safety and race-freedom of multicore programs that use asynchronous memory operations. Their approach uses separation logic with permissions, and their tool `asyncStar` automates this method. The tool targets a C-like core language. They describe possible solutions for problems like syntactic reasoning about permissions and arrays, integration of numerical abstract domains, and utilization of an SMT solver. They demonstrate their methods on a double-buffering algorithm example.

Interval Temporal Logic-Based Verification. Rely-Guarantee and separation logic are not the only ways to reason about interleaved programs sharing common resources. Interval temporal logic (ITL) [100] has been recognized as being able to model interleaved behavior and to reason about steps in the system, by introducing the notion of time. ITL is equipped with the necessary compositional operators to reason about interleaving.

Schellhorn et al. [116] present a logic that extends ITL with explicit, interleaved programs. They integrate the logic with higher-order logic, adding recursive procedures and rules to reason about fairness. They show how rules for Rely-Guarantee reasoning can be derived and outline the application of some features to verify concurrent programs in practice. One of the crucial design criteria of their logic is to make interleaving compositional.

The temporal logic and the programming language described in [116] has been directly implemented in the theorem prover KIV, as explained in [108]. KIV supports the entire design process from formal specification to verified code. Moreover, it supports functional as well as state-based modeling. It is to be investigated whether ITL-based verification will also be appropriate for accelerator programs.

Verification of Concurrent Data Structures. Finally, many of the verification techniques mentioned above have been used and adapted to reason about common concurrent data structures.

Hobor and Gherghina [71] develop a concurrent separation logic for pthread-style barriers. They give a formal characterization of sound barrier definitions and design a Hoare rule in separation logic for verifying barrier calls. They have also extended a program verification tool chain to automatically apply their Hoare rules to concurrent programs using barrier synchronization. Barrier stages are modeled as finite automata and are build into barrier definition.

Bell et al. [11] formulate an operational semantics and a concurrent separation logic to reason about multi-threaded programs that communicate asynchronously through channels and shared memory. They demonstrate how to transform a sequential proof into a parallelized proof that targets the output of the parallelizing optimization. They use histories in their logic to overcome limitations in local reasoning and to prove the correctness of parallelized programs in the presence of asynchronous communication.

There is some ongoing work on formal behavior specification of concurrent data structures at the University of Twente in the context of the Vercors project[1]. Zaharieva et al. propose history-based specifications for verifying concurrent data structures. A combination of JML and permission-based separation logic, including abstract predicates, is used as a specification language. This gives an abstract, modular way of specifying the behavior of concurrent queues.

6.3.2 Model Checking Techniques

Model checking [32] has come to denote a set of techniques for analyzing hardware and software systems through the construction of a finite *model* of the system and checking of logical properties of the model. If the model accurately captures system behaviors then properties established for the model can be inferred for the original system, and bugs in the original system may be revealed by showing that the model violates corresponding logical properties. Model exploration is either *explicit state*, where states are enumerated individually, or *symbolic*, where sets of states are represented as logical formulas, typically using binary decision diagrams. Canonical examples of explicit state and symbolic model checkers are SPIN [73] and SMV [98] respectively.

We now provide an overview of software model checking techniques, and applications of model checking to concurrent software. We focus on techniques for analysis of software written in system-level languages like C, as these techniques are most relevant to the CARP project. Software model checking has also been applied successfully to higher-level languages; see for example the Java PathFinder tool [85, 123].

Software Model Checking. Two styles of model checking have been successfully used for analysis of software source code: *bounded model checking* and model checking via *counterexample guided abstraction refinement*.

Bounded model checking [16] involves constructing, from a system description, a logical formula describing all executions of the system up to a given depth, such that satisfying assignments to the formula correspond to erroneous behaviors. The resulting logical formula can be efficiently checked by a state-of-the-art SAT or SMT solver. A popular bounded model checker for C programs is the CBMC tool [33].

Counterexample-guided abstraction refinement [31] involves constructing a finite-state abstract model of a software program using an appropriate over-approximating abstraction, which is usually *predicate abstraction* [59]. The abstract model is checked using a symbolic

[1]`http://fmt.cs.utwente.nl/research/projects/VerCors/`

model checker, e.g. SMV. If model checking succeeds, the original program is deemed correct, due to the over-approximating abstraction. Otherwise an abstract counterexample is generated, and simulated with respect to the original program. This either reveals a bug, or indicates that the abstraction was too coarse. In the latter case, the abstraction is automatically refined (e.g., by considering additional predicates, or using existing predicates more precisely), and the process repeats. The practical success of CEGAR-based model checking was demonstrated by the SLAM tool from Microsoft Research [3], which has since been integrated into the Windows device driver development kit. Other notable CEGAR-based model checkers are BLAST [67], which pioneered the use of *lazy* abstraction, and SatAbs [34] which uses a SAT solver for abstraction and refinement.

CEGAR-Based Model Checking for Concurrent Programs. An extension of BLAST for checking data races in concurrent programs is presented [66]; the MAGIC model checker [29] uses the CEGAR approach for analyzing concurrent programs where communication between threads is through message passing.

The SatAbs tool has been extended to support verification of concurrent C programs [47, 45], achieving scalability in the case of replicated C programs by exploiting symmetry between threads [99, 124]. This involves model checking finite-state concurrent abstract programs. For this purpose, the BOOM tool is employed [9], which combines symmetry reduction with symbolic model checking through the use of *counter abstraction* [10].

Bounded Model Checking for Concurrent Programs. ESBMC (Efficient SMT-based Context-Bounded Model Checker) [37] is a bounded model checker for concurrent C programs, based on the CPROVER framework[2] (on which CBMC and SatAbs are also built). ESBMC aims to find bugs in concurrent programs by bounding both search depth and number of thread context switches. The latter idea, context-bounded verification [101] exploits the empirical observation that real-world concurrency bugs usually require only a small number of unforced thread preemptions to manifest, and thus are still detected when search is restricted to paths that consider no more than a small number of such context switches.

Concurrency Testing Using Model Checking-Based Techniques. Context-bounding, together with other model checking techniques, notably *dynamic partial order reduction* [54] are the basis of CHESS, a concurrency testing tool which finds bugs by enumerating thread schedules [102].

Bibliography

[1] E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theor. Comput. Sci.*, 331(2-3):251–290, 2005.

[2] E. Ábrahám-Mumm, F. S. de Boer, W. P. de Roever, and M. Steffen. A tool-supported proof system for multithreaded Java. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2002.

[2] `http://www.cprover.org`

[3] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213. ACM, 2001.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In de Boer et al. [39], pages 364–387.

[7] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [8], pages 49–69.

[8] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2005.

[9] G. Basler, M. Hague, D. Kroening, C.-H. L. Ong, T. Wahl, and H. Zhao. Boom: Taking boolean program model checking one step further. In Esparza and Majumdar [53], pages 145–149.

[10] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2009.

[11] C. J. Bell, A. W. Appel, and D. Walker. Concurrent separation logic for pipelined parallelization. In R. Cousot and M. Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2010.

[12] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2007.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In Lodaya and Mahajan [92], pages 97–109.

[14] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In de Boer et al. [39], pages 115–137.

[15] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In Gopalakrishnan and Qadeer [56], pages 178–183.

[16] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[17] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 2003.

[18] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In Palsberg and Abadi [107], pages 259–270.

[19] M. Botincan, M. Dodds, A. F. Donaldson, and M. J. Parkinson. Automatic safety proofs for asynchronous memory operations. In Cascaval and Yew [28], pages 313–314.

[20] M. Botincan, M. Dodds, A. F. Donaldson, and M. J. Parkinson. Safe asynchronous multicore memory operations. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 153–162. IEEE, 2011.

[21] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2005.

[22] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In Yi [126], pages 52–70.

[23] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In R. Rabbah, editor, *Proceedings of the Third Workshop on Software Tools for MultiCore Systems, STMCS 2008, Boston, MA, USA, April 6, 2008*, 2008. Online proceedings at `http://people.csail.mit.edu/rabbah/conferences/08/cgo/stmcs/`.

[24] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[25] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2003, San Diego, California, USA, June 7, 2003*, pages 55–65. ACM, 2003.

[26] S. D. Brookes. A semantics for concurrent separation logic. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.

[27] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 289–300. ACM, 2009.

[28] C. Cascaval and P.-C. Yew, editors. *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*. ACM, 2011.

[29] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Asp. Comput.*, 17(4):461–483, 2005.

[30] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In H. R. Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401. Springer, 2007.

[31] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[32] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[33] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[34] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[35] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Barthe et al. [8], pages 108–128.

[36] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic testing of OpenCL code. In K. Eder and J. Lourenço, editors, *Proceedings of the 7th Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011*, 2011.

[37] L. Cordeiro. SMT-based bounded model checking for multi-threaded software in embedded systems. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 373–376. ACM, 2010.

[38] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [113], pages 313–400.

[39] F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*. Springer, 2006.

[40] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? In Lodaya and Mahajan [92], pages 250–262.

[41] D. Distefano, J.-P. Katoen, and A. Rensink. Safety and liveness in concurrent pointer programs. In de Boer et al. [39], pages 280–312.

[42] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[43] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In G. E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 213–226. ACM, 2008.

[44] M. Dodds and D. Plump. From hyperedge replacement to separation logic and back. *ECEASST*, 16, 2008.

[45] A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 2012. DOI: 10.1007/s10703-012-0155-3.

[46] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using

k-induction. In E. Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2011.

[47] A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In Gopalakrishnan and Qadeer [56], pages 356–371.

[48] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In Esparza and Majumdar [53], pages 280–295.

[49] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of dma races using model checking and *k*-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[50] A. F. Donaldson, D. Kroening, and P. Rümmer. SCRATCH: a tool for automatic analysis of DMA races. In Cascaval and Yew [28], pages 311–312.

[51] F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement, graph grammars. In Rozenberg [113], pages 95–162.

[52] K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In Gopalakrishnan and Qadeer [56], pages 372–378.

[53] J. Esparza and R. Majumdar, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*. Springer, 2010.

[54] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In Palsberg and Abadi [107], pages 110–121.

[55] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[56] G. Gopalakrishnan and S. Qadeer, editors. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.

[57] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In Yi [126], pages 240–260.

[58] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In Z. Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2007.

[59] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[60] A. Gupta and S. Malik, editors. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[61] C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for Java, 201x. Submitted.

[62] A. Habermaier. The model of computation of CUDA and its formal semantics. Technical Report 2011-14, University of Augsburg, 2011.

[63] A. Habermaier and A. Knapp. On the correctness of the SIMT execution model of GPUs. In H. Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 316–335. Springer, 2012.

[64] J. Heinen, H. Barthels, and C. Jansen. Juggrnaut - an abstract JVM. In B. Beckert and F. Damiani, editors, *Formal Verification of Object-Oriented Software, FoVeOOS 2011, October 5-7, 2011, Turin, Italy, Papers presented at the 2nd International Conference*, volume 2011-26 of *Karlsruhe Reports in Informatics*, pages 226–243. Karlsruhe Institute of Technology, 2011. Online proceedings at `http://foveoos2011.cost-ic0701.org/proceedings`.

[65] J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. *Electr. Notes Theor. Comput. Sci.*, 266:93–107, 2010. Proc. of 3rd Int. Workshop on Harnessing Theories for Tool Support in Software (TTSS).

[66] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In W. Pugh and C. Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 1–13. ACM, 2004.

[67] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[68] S. Heule, K. R. M. Leino, P. Müller, and A. Summers. Fractional permissions without the fractions. In S. Freund, editor, *Proceedings of the 13th Workshop on Formal Techniques for Java-like Programs, FTfJP 2011, Lancaster, UK, July 26, 2011*, 2011. Online proceedings at `http://www.cs.williams.edu/FTfJP2011/index.html`.

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[70] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In S. Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[71] A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In G. Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 276–296. Springer, 2011.

[72] H. P. Hofstee. Cell broadband engine processor. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 234–241. Springer, 2011.

[73] G. Holzmann. *The Spin model checker: primer and reference manual.* Addison-Wesley Professional, first edition, 2003.

[74] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic.* PhD thesis, Université Nice - Sophia Antipolis, Sept. 2009.

[75] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, January 17-19, 2001, London, UK, pages 14–26. ACM, 2001.

[76] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[77] B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM '05, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

[78] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 31(1):1:1–1:48, Dec. 2008.

[79] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[80] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, volume 4260 of *Lecture Notes in Computer Science*, pages 420–439. Springer, 2006.

[81] C. Jansen, J. Heinen, J.-P. Katoen, and T. Noll. A local Greibach normal form for hyperedge replacement grammars. In A. H. Dediu, S. Inenaga, and C. Martín-Vide, editors, *Language and Automata Theory and Applications - 5th International Conference, LATA 2011, Tarragona, Spain, May 26-31, 2011. Proceedings*, volume 6638 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 2011.

[82] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[83] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2009.

[84] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[85] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In M. B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer, 2001.

[86] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54. ACM, 2006.

[87] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In J. Vitek and H. Lin, editors, *Proceedings of the 33nd*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, June 11-16, 2012. ACM, 2012. To appear.

[88] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In G.-C. Roman and K. J. Sullivan, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 187–196. ACM, 2010.

[89] G. Li, G. Gopalakrishnan, R. M. Kirby, and D. Quinlan. A symbolic verifier for CUDA programs. In R. Govindarajan, D. A. Padua, and M. W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, Bangalore, India, January 9-14, 2010*, pages 357–358. ACM, 2010.

[90] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 215–224. ACM, 2012.

[91] G. Lindstrom. Scanning list structures without stacks or tag bits. *Inf. Process. Lett.*, 2(2):47–51, 1973.

[92] K. Lodaya and M. Mahajan, editors. *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*. Springer, 2004.

[93] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In Yi [126], pages 261–279.

[94] J. Lv, G. Li, A. Humphrey, and G. Gopalakrishnan. Performance degradation analysis of GPU kernels. In *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly, EC^2 2011, Snwobird, UT, USA, July 14-15, 2011*, 2011.

[95] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In Gupta and Malik [60], pages 428–432.

[96] R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2005.

[97] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.

[98] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[99] A. Miller, A. F. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006.

[100] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, Feb. 1985.

[101] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 446–455. ACM, 2007.

[102] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In R. Draves and R. van Renesse,

editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 267–280. USENIX Association, 2008.

[103] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2007.

[104] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[105] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[106] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[107] J. Palsberg and M. Abadi, editors. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 2005.

[108] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction - A Basis for Applications*, Applied logic series v. 8-10. Kluwer Academic Publishers, 1998.

[109] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[110] S. Rieger and T. Noll. Abstracting complex data structures by hyperedge replacement. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2008.

[111] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In Palsberg and Abadi [107], pages 296–309.

[112] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2005.

[113] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[114] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[115] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[116] G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In C. Combi, M. Leucker, and F. Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck , Germany, September 12-14, 2011*, pages 99–106. IEEE, 2011.

[117] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[118] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. H. Jr. and S. D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[119] S. Tripakis, C. Stergiou, and R. Lublinerman. Checking equivalence of SPMD programs using non-interference. In G. Lowney and D. Patterson, editors, *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism, HotPar'10, Berkeley, CA, USA, June 14-15, 2010*, 2010. Online proceedings at `http://static.usenix.org/events/hotpar10/`.

[120] V. Vafeiadis. Concurrent separation logic and operational semantics. *Electr. Notes Theor. Comput. Sci.*, 276:335–351, 2011.

[121] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[122] J. Villard, É. Lozes, and C. Calcagno. Tracking heaps that hop with heap-hop. In Esparza and Majumdar [53], pages 275–279.

[123] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[124] T. Wahl and A. F. Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2(2):799–847, 2010.

[125] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In Gupta and Malik [60], pages 385–398.

[126] K. Yi, editor. *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*. Springer, 2006.

[127] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GRace: a low-overhead mechanism for detecting data races in GPU programs. In Cascaval and Yew [28], pages 135–146.

7 Applications and Validation Metrics

We conclude the survey with a discussion of the state-of-the-art in key application areas relevant to the CARP project, as well as performance metrics related to these application areas.

In Section 7.1 we discuss current techniques for accelerating image processing using GPUs. We follow this with an overview of performance metrics for image processing in Section 7.2. Finally, in Section 7.3 we discuss state-of-the-art benchmark suites for OpenCL. These areas provide requisite background for WP7 of CARP, during which the CARP technology will be validated.

7.1 GPU Acceleration of Image Processing Applications

Computer vision tasks (and image processing tasks in general) are usually computationally intensive and repetitive, and have soft real-time requirements at the same time. However, partly because of this repetitiveness, many computer vision operations map efficiently onto the modern GPU. This efficiency appears not just in speed (although this is more important for the research community), but in power consumption too, which opens the possibility of the usage of these algorithms in mobile computing environments.

Importance of Speed. The gain from faster processing speed is threefold:

- *Qualitative benefit:* It makes it possible to get better accuracy from algorithms that are already real-time on the CPU by running it on higher resolution or with more iteration.
- *Quantitative benefit:* It makes it possible to run these algorithms more times in the same timeframe: for example to track more objects.
- *Ability of using new algorithms:* It also makes it possible to use algorithms which previously didn't fit in the timeframe allowed by the real-time requirements.

The authors of [22] showed that their CPU implementation was capable to track one face with 12 fps while their GPU implementation was capable to track 6 faces with 28 fps.

Cabido et al. in [5] measured 10x speedup compared to their CPU implementation and with this computation capacity they exceed the CPU implementation in terms of number of tracked objects and tracking quality.

In [6] the authors showed 400× speedup compared to their CPU based implementation.

In [32] the estimated speedup varied between 20 and 40.

Source of Parallelism. Most algorithms in Computer Vision are embarrassingly parallel, which means they usually need little or no effort to separate them into a number of parallel tasks and tend to require little or no communication of results between these tasks. These two features makes them ideal to run on the GPU. These features come from the fact that the majority of the algorithms are part of one or more of the following categories:

- *Pixel-level algorithms:* These algorithms work on the pixel level and use only local information, usually only the neighbouring pixels. All the different convolution filters, edge and corner detectors, template based trackers, color space conversion algorithms, SVM and neural network based filters belong to this category.

The author of OpenVIDIA showed the simplicity of these implementations in [10]. In [34] the authors show how well the various image feature calculations map to the modern GPU hardware.

- *Neural-inspired algorithms:* These algorithms are inspired by the inner workings of the brain, especially the visual cortex. Just like brain cells, these algorithms are highly parallel with only locally shared information.
- *Metaheuristics:* Many problems in Computer Vision are formalized as an optimization problem, and sometimes no direct solving algorithms or convergent iterative methods are known. In this cases one can try to solve them with the help of metaheuristics. These methods optimize a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality.
  Some of the classes of the metaheuristics like evolutionary algorithms, swarm intelligence or stochastic optimizations are evaluating different candidates independently in each iteration, making these calculations easily parallelizable [22, 5, 32]

Current Technologies. Because of the close relationship the pixel level computer vision algorithms were the first algorithms which were ran on the GPUs.

- *1st generation:* These early implementations used OpenGL Cg or GLSL and the calculation itself happened in the standard OpenGL pipeline with the help of FBO-s [6, 23, 27] The creators of GPUCV [12] and OpenVidia [24] used this technology to implement parts of OpenCV [39] as the de facto standard computer vision library.
- *2nd generation:* After its release, the adoption of CUDA was immediate [3, 5, 13, 20, 22, 25, 29, 32, 39, 24]. The adoption of CUDA has not been largely affected by the newer OpenCL standard, except that library writers usually favor OpenCL because of its openness [39, 24, 11, 18, 15].

A detailed overview of the CUDA and OpenCL programming models is provided in Section 3.4.1.

Future Technologies. Along with current ones, we anticipate that two new technologies are likely to gain traction: *Renderscript* and *C++ AMP*. These technologies are discussed in detail in Sections 3.4.1 and 3.4.2 respectively.

7.2 Performance Metrics in Image Processing

The comparison of the CPU and GPU architectures can be done through various metrics. There exist simple algorithms which exploit a special characteristic of each architecture as well as the performance increase provided by the compiler.

Benchmarking Through Hardware Virtualization. The authors of [1] propose a virtual environment to make a detailed analysis of the binary code on the GPU architecture. They have developed GPGPU-sim, which is a clock accurate simulator which supports the CUDA binaries and can simulate their execution on a GPU.

On the memory side, they consider all four memory types accessible to CUDA warps (a warp constitutes of 32 threads), the global, local, constant and texture; and implement penalties for accessing them. The global memory access is simulated through separate send and receive interconnection networks.

They implement a round-robin virtual scheduler. The threads are selected in groups and each group is executed through four clock cycles. If a thread arrives to a branch in the execution, then the rest of the threads in the warp are paused until the branched execution finishes.

From the hardware point-of-view threads they implement grouping in CTA (Cooperative Thread Arrays). This approach is hardware based and has benefits in some cases. The threads from the CTA-s can access a common shared memory and the CTA-s can synchronize between each other. This means that more CTA-s with a carefully chosen size can saturate the resources better than a single huge CTA.

Using the GPU simulator the authors/users can obtain a detailed measurement on the Instruction (SFU, Control Flow, Fused Multiply-Add, ALU operators), the Memory Instruction Classification (Shared, Texture, Constant, Global, Local), the IPC (Instruction per Clock) and the Warp Occupancy.

Benchmarking the Energy Efficiency. To this date little research has been conducted to evaluate the energy efficiency of the GPU for general purpose computation. Even less effort has been made for compilers/development which would optimize energy the consumption of the application. These aspects are of paramount importance on embedded devices, especially for applications that run permanently possibly in the background.

The authors of [14] propose a metric to analyse the energy efficiency of modern GPU hardware. They launch the applications they would like to profile and sample the power consumption using an external meter. They attribute two properties to a computation: 1. The consumed energy W (in Joule) and 2. the computation time t(in s). From these two metrics, one could define an abstract value, the energy delay (E_D) by

$$E_D := W \cdot t \tag{7.1}$$

The energy delay is a measure of the efficiency. E_D is linear in energy and time, which means we can improve the efficiency twice by either reducing the computation time (though not the energy consumption) to half of reduce the required energy to half (calculating for the same time, but, for example, reducing the occupancy on the GPU SPE cores)

Such benchmarks can be performed on both sequential and parallel implementations, as well as CPU, GPU, SoC and embedded systems.

Benchmark Using Specialized Algorithms. The authors of [28] propose a set of algorithms for benchmarking the image processing algorithms on GPU. According to them the main aspects of GPU execution efficiency are:

- *Parallel fraction.* Is the fraction of the computation time which is parallelizable. ref. Gustafsons law
- *The ratio of floating-point computation to global memory access.* If we consider a simple linear saturation algorithm $q \cdot I$. In this case the floating point performance of the processor is negligible compared to the necessary memory bandwidth, whereas in the convolution or matrix multiplication the pixels are reused several times, thus stressing the floating point performance rather than the pure memory bandwidth. Higher this ratio is the better is the occupancy of the GPU.

- *Per-pixel floating point instructions.* The GPU can perform 20× the floating point instruction per second than CPUs can. Therefore the more instructions are necessary per pixel higher will be the performance gain of a GPU implementation.
- *Per-pixel memory access.* The above fact is also true for the memory bandwidth. As the memory controllers are physically distributed around the cores, the GPU hase 10× higher pure memory bandwidth than the CPU.
- *Branching diversity.* Branching can influence significantly the performance of the algorithms. It is crucial, therefore, to eliminate the branches from the binary code wherever possible. Some branches can be eliminated using the embedded conditional operator, so it is important for the compiler to automatically recognize and apply such optimizations. An example operator can contain lots of hand-optimizable branching.
- *Task dependency.* The image processing algorithm can be divided into several subtasks: operators. The dependency between the operators is not necessarily sequential. The compiler should therefore try to recognize the dependency tree of the operators to reduce the waiting during the execution.

Afterwards, we can measure the efficacy of each aspect by selecting an image processing application for each metric.

Benchmarks on Handheld GPUs. The authors of [33] considered handheld devices for image processing. As these devices use two orders of magnitude less power than their desktop counterparts, it is more difficult to reach to their computation power. Therefore the authors used OpenGL ES to show, that some image processing applications can be programmed using the GPU shaders.

As OpenCL is becoming more and more available, we can use the same code to benchmark embedded and desktop devices. Moreover if the system consists of a GPU and a multi core CPU, the computational performance can be cumulated as these usually share the same memory.

Parallel Programming Models. The authors [8, 19, 9] evaluate different parallel programming environments. Any given algorithm can be implemented in several programming frameworks (programming language / compiler / environment). There are several domain specific languages (CUDA, OpenCL, UPC, OpenGL ES) as well as programming environments (PyCUDA, PyOpenCL, OpenMP).

The development time can be considerably different using different languages, so this aspect may be considered in the benchmark of the programs. Also domain specific languages (CUDA, OpenCL) or language extension (OpenMP) can guide the compiler in the parallelization of the algorithms. Implementation in several different programming environments can be compared and the development time can also be benchmarked against the simple compiler-optimized binary code.

Hardware Profiling. A crucial part of software optimization is identifying bottlenecks. Bottlenecks are parts of the code that saturate a certain aspects of the hardware. In a streaming application, for example, the network speed can be a bottleneck. This means that even if we accelerate the algorithm an order of magnitude, the runtime will not decrease, as most of the time it will be waiting for the network to deliver the data to process. In the aspect of GPU the bottleneck can be the branching, the memory speed or, more often, the speed of the PCI bus[1].

[4] provides an overview of the standard profiling techniques. The main method is the CUDA profiler provided by NVidia. This is a statistical profiler. During statistical profiling we launch the executable, that we would like to analyze. During the execution we stop the system, and log the place where the system was stopped. It is referred to as collecting the sample. After some statistical considerations the higher is the proportion of the samples collected in a particular part of the code (this can be a line in the source code or a particular assembly instruction), the higher is the proportion of time that the system spends in executing that instruction.

A second way of profiling can be done by modifying the source code of the application. Typically some modules can be switched off and their output replaced by artificial results. This way the we can determine how much time is spent in the particular module. This way we can select the modules of the source code, which influence the most the total running time. Usually these modules are the best candidates for optimization.

A detailed hardware profiling of an application can consist of statistics for each function A:

- The time that the program have spent in A
- The time that the program have spent in A including the times that the program have spent in each functions B called from A

$$T(A) = A + \sum_{A \to B} T(B) \tag{7.2}$$

- The memory bandwidth used in A
- The IPC of A

From this we can determine, whether the given binary occupies the GPU/CPU optimally.

Proposed Benchmarks to Evaluate the CARP Compiler. We can define a base of algorithms each exploiting a particular characteristic of the target architectures. Examples include:

- *High parallel fraction operators*. morphological opening, convolution are operators which are embarrassingly parallel so they could be a good candidates to check the parallelization capabilities of the compiler
- *High memory throughput operators*. threshold, mean, matrix transposition, histogram are operators which require few operations, therefore they are likely to saturate the memory bandwidth
- *High IPM* (instructions per global memory access). operators as matrix multiplication require the elements of the image several times, therefore consist of many floating point operations
- *Higher level optimizations*. Small Matrix Multiplication. In some cases small matrices can be regrouped into bigger matrices and we can perform several matrix multiplication using single matrices. In this case the size of the matrix can be determined using the hardware characteristics (RAM size/ Cache size).

Note that the energy efficiency profiling is more complex that the above mentioned techniques as it requires a separate physical device and profiler to be attached to the system that we benchmark. We can, however implement such benchmark if we get assistance from the hardware vendor.

[1]The PCI bus connects the GPU with the rest of the system. Before we can process an image, we have to transfer it from the system RAM to the GPU RAM through the system bus

7.3 OpenCL Benchmarks

The current benchmarks for OpenCL can be divided into two roughly different segments. One of which is the scientific computing arena and another is consumer benchmarks.

Dedicated consumer benchmarks tend to report their results as an arbitrary score, benchmarks developed from actual applications tend to measure either seconds or operations/second, whichever is more relevant for the application at hand. The aim is to produce a single number to allow quick comparison between devices, focused more on qualitative rather than quantitative comparisons. The tests in consumer benchmarks mainly tend to consist of image processing, raytracing, simple physics simulations and synthetic tests such as Julia fractals and raw memory bandwidth tests.

Benchmarks originating from the scientific computing arena tend to have a wider array of tests than consumer benchmarks. Due to popularity of CUDA in scientific computing the benchmarks usually have support for CUDA devices. Unlike in consumer oriented benchmarks there is no tendency to make the result marketable by producing a single combined result of all the tests.

7.3.1 Major OpenCL Benchmark Suites

SHOC: The Scalable Heterogeneous Computing Benchmark Suite. SHOC [37] is an Open Source benchmark mainly focused for scientific computing. It includes features suitable for testing clusters of OpenCL devices. The tests are divided into three categories: (1) a set of simple synthetic performance tests, (2) a set of parallel algorithms such as FFT, sort, graph traversal and parallel reduction, and (3) two fully fledged applications.

ParBoil Benchmarks. The ParBoil Benchmarks [16, 38] are a suite of Open Source benchmarks focused on throughput computing applications. The tests range from matrix multiplication to fully fledged fluid dynamics simulations. As well as OpenCL, all ParBoil benchmarks are available in CUDA and OpenMP forms.

Rodinia Benchmark Suite. Rodinia [36, 7] is an Open Source benchmark for scientific computing. It includes tests ranging from simple graph traversal to application specific tasks such as analysing ultrasound images of a heart. In addition to OpenCL, Rodinia has support for CUDA and OpenMP.

Rightware Basemark CL. Basemark CL [31] is a commercial benchmark developed by RIGHT. The tests in Basemark CL are SPH fluid, 2d fluid, FFT waves, Image filters and fractals. Basemark CL features workloads that stress the OpenCL implementation in a realistic way, therefore yielding performance measurement data that is objective and relevant. With the help of OpenCL, it is possible to improve games, applications and user interfaces for instance by including physics-based animations and game-like elements.

LuxMark. LuxRender [21] is Open Source rendering engine. LuxMark was developed from their OpenCL accelerated rendering code.

Kishonti CLBenchmark. CLBenchmark by Kishonti informatics [17] is commercial benchmark for desktop devices. The tests include SPH fluid, Raytracing, Optical flow, Image filters and set of simple algorithms such as bitonic sort and tree traversal.

Bitcoin Hashing. Bitcoin hashing [2] is a popular benchmark for OpenCL; practically all profitable Bitcoin mining is performed with GPUs using OpenCL. Performance in Bitcoin hashing is heavily dependent on the integer processing speed of the hardware.

7.3.2 Other Suites Including OpenCL Benchmarks

In addition to the above OpenCL focused benchmarks there are several other tools which include OpenCL as part of their benchmark set, such as GPU Caps Viewer [26], SiSoftware Sandra [35] and Phoronix Test Suite [30]. These tests tend to be either simple synthetic benchmarks or fractals.

Bibliography

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[2] Bitcoin. OpenCL miner. `https://en.bitcoin.it/wiki/OpenCL_miner`.

[3] M. D. Breitenstein, D. Kuettel, T. Weise, L. van Gool, and H. Pfister. Real-time face pose estimation from single range images. *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[4] A. R. Brodtkorb, T. R. Hagen, and M. L. Sæ tra. GPU programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, May 2012.

[5] R. Cabido, A. S. Montemayor, and J. J. Pantrigo. High performance memetic algorithm particle filter for multiple object tracking on modern GPUs. *Soft Computing*, 16(2):217–230, May 2011.

[6] A. C. Campilho and M. S. Kamel, editors. *Image Analysis and Recognition, Third International Conference, ICIAR 2006, Póvoa de Varzim, Portugal, September 18-20, 2006, Proceedings, Part II*, volume 4142 of *Lecture Notes in Computer Science*. Springer, 2006.

[7] 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 *IISWC*, pages 44–54. IEEE, 2009.

[8] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, Oct. 2011.

[9] T. a. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed. Benchmarking parallel compilers: A UPC case study. *Future Generation Computer Systems*, 22(7):764–775, Aug. 2006.

[10] J. Fung. Computer Vision on the GPU. *GPU Gems*, pages 651–668, 2005.

[11] GEGL. `http://www.gegl.org/`.

[12] GPUCV. `https://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/`.

[13] A. Herout and R. Josth. GP-GPU Implementation of the Local Rank Differences Image Feature. *Computer Vision and Graphics*, pages 380–390, 2009.

[14] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[15] ImageMagick. `http://www.imagemagick.org`.

[16] IMPACT group, University of Illinois. Parboil Benchmarks. `http://impact.crhc.illinois.edu/parboil.aspx`.

[17] Kishonti informatics. CLBenchmark. `http://www.kishontiinformatics.com/product_clbenchmark.jsp`.

[18] Kitware. ITK. `http://www.itk.org/`.

[19] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, Mar. 2012.

[20] P. Li. An Efficient Particle Filterbased Tracking Method Using Graphics Processing Unit (GPU). *Journal of Signal Processing Systems*, Sept. 2011.

[21] LuxRender. LuxMark. `http://www.luxrender.net/wiki/LuxMark`.

[22] O. Mateo Lozano and K. Otsuka. Real-time Visual Tracker by Stream Processing. *Journal of Signal Processing Systems*, 57(2):285–295, July 2008.

[23] E. Murphy-Chutorian and M. Trivedi. Head pose estimation and augmented reality tracking: an integrated system and evaluation for monitoring driver awareness. *, IEEE Transactions on*, 11(2):300–311, 2010.

[24] OpenVidia. `http://openvidia.sourceforge.net`.

[25] K. Otsuka and J. Yamato. Fast and Robust Face Tracking for Analyzing Multiparty Face-to-Face Meetings. *Machine Learning for Multimodal Interaction*, pages 14–25, 2008.

[26] oZone3D.Net. Gpu caps viewer. `http://www.ozone3d.net/gpu_caps_viewer/`.

[27] G. Panin and A. Knoll. A GPU-accelerated particle filter with pixel-level likelihood. *2008: proceedings, October 8-10, 2008*, 2008.

[28] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim. Design and performance evaluation of image processing algorithms on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):91–104, Jan. 2011.

[29] S.-Y. Park, S.-I. Choi, J. Kim, and J. S. Chae. Real-time 3D registration using GPU. *Machine Vision and Applications*, 22(5):837–850, July 2010.

[30] Phoronix. Suldal improvements; new OpenCL, VDPAU benchmarks. `http://www.phoronix.com/scan.php?page=news_item&px=MTA5NDA`.

[31] Rightware. Basemark CL. `http://www.rightware.com/en/Benchmarking+Software/Basemark%99+CL/`.

[32] B. Rymut and B. Kwolek. GPU-supported object tracking using adaptive appearance models and particle swarm optimization. *Computer Vision and Graphics*, pages 227–234, 2010.

[33] N. Singhal, I. K. Park, and S. Cho. Implementation and optimization of image processing algorithms on handheld gpu. In *ICIP*, pages 4481–4484. IEEE, 2010.

[34] S. Sinha, J. Frahm, and M. Pollefeys. GPU-based video feature tracking and matching. *EDGE, Workshop on Edge*, 012(May):1–15, 2006.

[35] SiSoftware. SiSoftware zone. `http://www.sisoftware.co.uk/`.

[36] K. Skadron et al. Rodinia Benchmark Suite. `http://lava.cs.virginia.edu/Rodinia/`.

[37] K. Spafford. The Scalable HeterOgeneous Computing (SHOC) benchmark suite. `https://github.com/spaffy/shoc/wiki`.

[38] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. ParBoil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[39] WillowGarage. OpenCV. `http://opencv.willowgarage.com/`.

A Overview of Major Programming Models

In Section 3.4.2 we presented a brief overview of several major programming models. In this appendix we provide a comprehensive discussion of each.

A.1 POSIX threads

The POSIX threading API (‘pthreads’) is an IEEE standard and serves as the native thread API on most UNIX-like systems. It is a pure C API implementing the MPMD model. Some compilers (including GCC) include a language extension for thread local storage (`__thread` keyword).

Motivation. The pthreads API was created as a way to standardize behavior of multi-threaded programs on POSIX systems. Actual implementation is at the discretion of the individual POSIX implementation; it is possible to implement pthreads within a single ‘kernel’ thread without much support from the OS. Threads are desirable on POSIX systems because the alternative method of providing multiple threads of execution (multiple processes) is very heavyweight. The POSIX method of creating a new process (the `fork()` call) causes every aspect of the existing process (including the entire memory space) to be copied in to the new one. Shared memory regions can be set up and attached to in multiple processes to achieve similar semantics to threads, but it is a slow and resource-intensive process. Modern POSIX implementations use various techniques to optimize `fork()`, but using threads is still easier for the programmer and allows better integration of other POSIX semantics such as signals.

Parallel Language. There are no language restrictions for parallel code, which is not demarcated in any way from the serial code. Any C function or feature can be used from any thread. The API can also be used in C++ programs, in which case the same applies. It is possible that certain libraries may not be thread safe - this is dependent on the library and system in question. In the case of non-thread safe libraries the programmer is responsible for ensuring that only one thread calls into the library at a time.

Task Dispatch. New threads are created one at a time via the `pthread_create()` function call. The arguments include an arbitrary function pointer indicating the code to be run in the new thread and a user pointer (`void *`) which can be used to provide context information to the new thread. A new thread is created and the function pointer provided is called, with the user pointer as an argument. This initial function is free to call any other functions in the program. If the initial function returns, the thread is terminated.

Synchronization. Low-level mutual exclusion primitives are provided by the library in the form of mutexes (a simple mutual exclusion lock) and condition variables (which allow a thread to block waiting for a specific signal from another thread). A thread can wait for another thread to exit with the `pthread_join()` function.

Memory Model. Threads have equal access to memory to the main thread. All threads share a memory space, so changes made on one thread (e.g. memory allocation or mapping a file into memory) are visible to other threads.

Compilation Model. Parallel code is no different from serial code with regards to compilation. In general this means it is compiled in advance. However, mechanisms like `dlopen()` allow code to be loaded from dynamic libraries at runtime, so a system requiring runtime compilation could be implemented on top of C/pthreads by compiling into a dynamic library and loading it at runtime.

Minimum Steps Needed to Run Parallel Code. Although pthreads is a low level API, the implicit shared instruction set and memory make it easy to get started:

1. Call `pthread_create()` to launch a new thread executing a named function.

Discussion. pthreads is a low-level API which is well suited to creating multiple threads with potentially disparate purposes on an SMP system. Other programming models can be supported by building libraries (and possibly compiler support) on top of pthreads. As the primitive for creating threads (`pthread_create()`) creates a single thread, pthreads is not a suitable model for use on a SPMD-based machine.

pthreads does essentially nothing to help the programmer deal with the complexities of concurrent programming. The programmer needs to be aware of the requirements for synchronization and deadlock avoidance and must use the primitives supplied by pthreads to build a program which works correctly. This can lead to complex and frustrating debugging where a subtle race condition is missed by the programmer and causes sporadic errors.

A.2 OpenMP

OpenMP is maintained by the OpenMP ARB, an independent non-profit corporation composed of members from various interested parties. OpenMP is a language extension for C and FORTRAN which allows programmers to use pragmas to indicate parallel subsections of code. OpenMP is generally implemented on top of a native threading library — pthreads in the case of UNIX-like systems.

Motivation. OpenMP was originally developed for FORTRAN. Simple directives to indicate parallel loops are much easier for programmers than having to deal with an explicit threading library, and in many cases can provide just as much performance improvement, particularly when the programmers are experts in the problem being solved but not necessarily dealing with a threading library. OpenMP was an effort to avoid a proliferation of incompatible compiler- and machine-specific schemes to achieve this.

Parallel Language. Parallel code is written in the same language as serial code. There are no restrictions on which language features are used by the parallel code. The programmer is responsible for any synchronization needed or bugs arising from dependencies which prevent their program working properly.

Pragmas are a C language feature which are defined to be ignored by compilers which do not understand them (special comments are used in FORTRAN to similar effect). OpenMP code can therefore be built with non OpenMP compilers (or with OpenMP features turned off) to create a serial version of the program.

Using OpenMP can be as simple as putting a single pragma in front of a loop which is to be parallelized (the iterations of the loop are divided over a number of threads). Additional control is available such as specifying which variables are shared or private to the thread, how many threads to use, how to divide up the loop iterations and so on. Used in this way, OpenMP provides an SPMD model. Alternatively, a block of code can be divided into subsections, one thread executing each subsection. This is a much more MPMD like approach.

Task Dispatch. When an OpenMP program is compiled, relevant pragmas are turned into function calls to create additional threads to run the parallelized workload. By default, all threads synchronize together at the end of a parallel subsection (although this behavior can be turned off).

Synchronization. OpenMP supports a variety of synchronization options. Within a parallel block of code, sub-blocks can be marked as ’critical’ (only one thread will execute in the critical subsection at a time), ’single’ (only a single thread will run the code in the single subsection), ’master’ (only the master thread will execute). Within a parallel block of code, all threads can be synchronized via a barrier pragma (all threads will wait at the barrier until all threads have reached it). Parallel blocks can specify reduction variables, for example a loop summing a series of values can be parallelized, with the runtime adding the totals produced by each thread together.

Memory Model. OpenMP’s memory model is identical to that of pthreads. OpenMP threads have equal access to memory to the main thread. Threads share a memory space, so allocations made by one thread are visible to the others.

Compilation Model. Parallel code is compiled in advance. Usually, OpenMP pragmas are handled by ‘lowering’ into standard C or FORTRAN at compile time. This consists of putting parallel blocks into new functions (building a context structure as appropriate to capture any local variables used from the parent function) and inserting a call to a library function at the original site. The library function then handles creating the threads to do the actual work. The lowering can be done by the normal compiler used for serial code if it supports OpenMP, or it can be done as a preprocessing step.

As for pthreads, it is possible to build additional code at runtime and load it via `dlopen()` or equivalent.

Minimum Steps Needed to Run Parallel Code. OpenMP is very easy to get started with:

1. Add `#pragma omp parallel` for before the for loop which is to be parallelized.

Discussion. OpenMP is a very low overhead way to make code parallel — a single pragma and compile flag change is enough to parallelize a loop over several CPUs. This is considerably simpler than manually writing boilerplate code to spawn a thread, work out which iterations the thread is to handle, etc. As always, there does need to be inherent parallelism present before OpenMP can help. For example, if a loop carries dependencies from one iteration to the next it cannot be parallelized by OpenMP — pragmas can be added, and the OpenMP implementation will dutifully make it parallel, but the resulting program will not work properly.

Although OpenMP's capabilities are essentially similar to pthreads', the parallel loop pragmas allow a SPMD programming model to be used with less boilerplate code than pthreads requires. This is very helpful as this is the easiest sort of parallelism for programmers to reason about, and the lack of boilerplate code removes a potential source of bugs.

A.3 Renderscript

Renderscript was created and is owned by Google as part of their Android platform.

Motivation. Renderscript was designed to plug a gap in the Android application model as well as specifically addressing parallelism and accelerators. Standard Android applications are programmed in Java and run on the Dalvik virtual machine. As a virtual machine architecture it is not ideal for high performance code. Android's original solution to this problem was to allow applications to include native code written in C/C++, but this either limits compatibility, increases complexity or potentially compromises performance as there are a variety of platforms with varying capabilities the code may have to run on. Renderscript was introduced as a way to allow applications to include high performance code without making them machine dependent. It is also designed to integrate well with Dalvik applications.

In addition to its compute features Renderscript also includes graphics rendering functions. We consider it strictly as a compute acceleration language.

Parallel Language. Parallel code is written in C99 in separate files to the Java serial code. Use of certain 'problematic' features (such as recursion) is not disallowed, but may limit the devices the parallel code can subsequently be run on. Some extra types are provided (e.g. for SIMD vectors), and one or two special header pragmas are used to identify the Renderscript version and identify its classpath for integration with the Java application.

The Android build system automatically generates a set of classes based on the Renderscript code to allow the Renderscript routines to be controlled and invoked from the serial application code.

Task Dispatch. Tasks are dispatched by calling a generated function from the Java serial code. This handles the passing of any parameter values down to the parallel program and causes it to be run. Functions execute asynchronously so cannot directly return values to the runtime.

The runtime is entirely responsible for selecting the target device. The initial Renderscript implementation only targeted CPU devices, but it is explicitly designed to target GPUs. Target device selection can be based on availability or characteristics of the code being invoked (which can be determined at compile time and matched to any GPU support present at runtime).

Either scalar routines or kernels can be invoked (scalar routines in Renderscript can in turn invoke kernels). When a kernel is invoked, the domain is based on the input memory allocation passed to it.

Synchronization. Renderscript parallel code executes asynchronously, but a memory object which parallel code has written to will synchronize automatically (wait for parallel code to complete) when accessed from serial code.

Memory Model. Renderscript uses a split memory model. Any memory to be used in parallel code must be explicitly allocated from serial code and initialized if necessary. If a memory object is used by parallel Renderscript code, subsequent attempts to use it from serial code trigger a synchronization event.

Compilation Model. Renderscript code is compiled to an intermediate representation at compile time. At run time it translated into machine code for the specific machine it is running on.

Minimum Steps Needed to Run Parallel Code. Renderscript has some explicit context and memory management requirements:

1. Create a context.
2. Create memory objects for passing input/output data.
3. Copy input data to memory object.
4. Invoke the kernel.
5. Copy output data from memory object.

Discussion. Renderscript is different from most of the other models considered here because it is closely integrated with one particular platform, with no hint of portability. Designing from the ground up, rather than adding bindings for an existing model (OpenCL would be the obvious candidate) to Android allows for maximum integration in the Android environment but at the expense of adding yet another dialect of 'parallel C' for programmers to deal with. It also makes life harder for GPU vendors as they need to create explicit Renderscript support rather than relying on standardized bindings onto an existing API.

It seems likely that the decision to go this way for Renderscript was at least partly motivated by the political situation between Google (creators of Android) and Apple (creators of OpenCL).

A.4 OpenACC

OpenACC is a maintained by a non-profit corporation very similar to OpenMP. OpenACC provides an API very similar to OpenMP in approach which allows offload of processing to accelerators such as GPUs (as opposed to OpenMP which explicitly assumes that all processing will be undertaken on the host CPU or SMP system).

Motivation. The desire to bring the simplicity of OpenMP to accelerator programming is an obvious motivation. Either by slightly modifying and extending OpenMP, or using automatic communication generation and offloading techniques. The best representative of the latter is the work based on the Cetus compiler framework, aiming to translate OpenMP to CUDA directly.

In parallel, the OpenMP group started a working group with the goal of integrating OpenMP with accelerators. This working group is making good progress in this direction, but not good enough to release a specification with the OpenMP name on it. OpenACC was therefore created as a way to get something released without having to worry about incompleteness and future incompatibility with OpenMP, and as a way to get real-world feedback for when the features are integrated into the next version of OpenMP.

Parallel Language. OpenACC does not use a separate parallel language. The whole program is written in C, C++ or FORTRAN with pragmas indicating parallel regions. The current OpenACC specification lists a few restrictions on code that can appear in parallel regions, it is possible that more will exist in the final version. The OpenACC compiler and runtime handle the separate compilation and dispatch of parallel code onto the accelerator. Like OpenMP, OpenACC aims to be "optional" in that a valid OpenACC program should build and run correctly on a non-OpenACC system.

Task Dispatch. The OpenACC runtime handles dispatch of kernels to the accelerator. The programmer can specify conditions for dispatch (e.g. total size of computation to be done) which are evaluated at runtime, and if the conditions fail the loop is performed on the CPU instead.

Synchronization. At present OpenACC does not include explicit operations for synchronization within parallel blocks, but it is likely they will be added in future. OpenACC does support synchronizing between parallel blocks and the host processor; either the host processor implicitly waits for a parallel block to complete when it is encountered, or (via an optional async clause) can opt to explicitly wait at a later point.

Memory Model. One of OpenACC's main divergences from OpenMP is the memory model. OpenACC assumes that memory is not necessarily coherent between CPU and GPU. The programmer can indicate for each variable, array or sub-array whether it needs copying from host to device before executing the parallel region, copying back after execution, both, or neither. Alternatively, it is possible to use a 'data construct' to describe areas which potentially containing several parallel regions and specify that data is to be copied at the start or end of the area, and remain on the device between parallel regions. This allows the programmer to avoid needless extra copies between the device and host.

If the programmer chooses not to include any of the above directives, the runtime will assume that copies in both directions are needed for all data used in the parallel region. Hence, although the programmer does not need to include specific commands to copy data, for optimum performance the programmer still needs to correctly determine and then specify which copies are to be performed when. However, the default of copying everything always provides a useful starting point.

Compilation Model. The compilation model is not specified in the OpenACC specification. Like OpenMP, code is in general present at compile time, but it is down to the details of the compiler and runtime implementation how the two cooperate to arrange for compilation of parallel regions to be performed. It is possible for all code to be compiled in advance, alternatively the source code for parallel regions could be captured at compile time and then actually compiled at run time. A third alternative is a compilation to an intermediate representation at compile time, with final compilation onto the specific accelerator present occurring at runtime.

Minimum Steps Needed to Run Parallel Code. Although the underlying memory and execution model are similar to that of OpenCL, the runtime takes care of practically all of the housekeeping automatically. Therefore, it is as easy to get started with as OpenMP:

1. Add `#pragma acc parallel for` before the for loop which is to be parallelized.

Discussion. OpenACC is an interesting attempt to make accelerator programming more accessible. The relationship between OpenACC and OpenCL is very similar to the relationship between OpenMP and pthreads; it is not intrinsically any more or less powerful, but allows a much easier starting point. As with OpenMP, in principle a first attempt at parallelization can be made with a single pragma, although more pragmas and clauses may be required for best performance.