CARP

D6.4: Update on verification of PENCIL, PENCIL→OpenCL transformations, and verification of OpenCL

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:	39 months
Document Type[1]:	D (Deliverable)
Document Distribution[2]:	PU (Public)
Document Code[3]:	CARP-ICL-RP-009
Version:	v1.4
Editor (Partner):	Alastair F. Donaldson (ICL)
Contributors:	ICL, UT, ENS
Workpackage(s):	WP6
Reviewer(s):	RWTHA, ENS
Due Date:	30 June 2014
Submission Date:	7 August 2014
Number of Pages:	57

[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.

D6.4: Update on verification of PENCIL, PENCIL→OpenCL transformations, and verification of OpenCL

Alastair F. Donaldson[1], Ethel Bardsley[1], Adam Betts[1], Jeroen Ketema[1], John Wickerson[1], Marieke Huisman[2], Stefan Blom[2], Saeed Darabi[2], Sven Verdoolaege[3], Albert Cohen[3]

[1]ICL, [2]UT, [3]ENS

REVISION HISTORY

Date	Version	Author	Modification
15 July 2014	0.1	A.F. Donaldson (ICL)	Added document skeleton
15 July 2014	0.2	J. Ketema (ICL)	Corrected document number
28 July 2014	0.3	S. Blom (UT)	Imported initial versions of program logic verification chapters
29 July 2014	0.4	S. Blom (UT)	Added introductory text for program logic chapters
30 July 2014	0.5	S. Blom (UT)	Updated program logic chapters
30 July 2014	0.5	S. Blom (UT)	Updated program logic chapters
30 July 2014	0.6	A.F. Donaldson (UT)	Added executive summary
30 July 2014	0.7	S. Blom (UT)	Implemented fixes suggested by ICL
31 July 2014	0.8	A.F. Donaldson (ICL)	Imported kernel interceptor and termination chapters to document
31 July 2014	0.9	J. Ketema (ICL)	Fixed imported chapters
31 July 2014	0.10	S. Blom (UT)	Improvements to program logic chapters
31 July 2014	0.11	A.F. Donaldson (ICL)	Translation validation chapter
1 August 2014	1.0	A.F. Donaldson (ICL)	Deliverable sent to RWTHA for internal review
6 August 2014	1.1	A.F. Donaldson (ICL)	Revised based on comments from C. Jansen at RWTHA
6 August 2014	1.2	S. Blom (UT)	Revised based on comments from C. Jansen at RWTHA
7 August 2014	1.3	S. Blom (UT)	Revised based on comments from S. Verdoolaege at ENS
7 August 2014	1.4	A. Donaldson (ICL)	Revised based on comments from S. Verdoolaege at ENS, and signed off deliverable

APPROVALS

Role	Name	Partner	Date
Workpackage Leader	A.F. Donaldson	ICL	7 August 2014
Project Manager	A.F. Donaldson	ICL	7 August 2014

Contents

1 Executive Summary

This deliverable reports the status of verification efforts in the CARP project related to the PENCIL intermediate language of WP3, to the OpenCL programming model for heterogeneous many-core platforms, and to the process of transforming PENCIL programs into OpenCL code using the compilation tools of WP4. Specifically we report full details of the PENCIL verification method that was outlined in Deliverable D6.1, and present an update on tooling issues related to the verification method for OpenCL using permission-based separation logic introduced in D6.2. We describe the use of GPUVerify for proving race-freedom of OpenCL code generated by the PENCIL→OpenCL compiler of WP4. We then discuss two advancements in analysis methods for OpenCL kernels: a technique for further automating verification via *kernel interception*, and a novel method for automatically proving that OpenCL kernels terminate.

2 Introduction

This deliverable summarises the progress of the CARP research work into verification methods for accelerator programming. Specifically we update on techniques outlined in Deliverable D6.1 for analysing programs written in the PENCIL intermediate language of WP3, and methods detailed in Deliverable D6.2 for verifying basic and functional properties of OpenCL kernels. We also study the interface between these two languages: the compilation tools of WP4 that translate PENCIL programs into optimised OpenCL code. Our contribution regarding this interface is a method for checking that compiled code is free from data races by combining the strengths of the GPUVerify verification method with domain knowledge available from the PENCIL→OpenCL compiler.

Verification methods based on program logic. Chapters 3 and 4 of this deliverable deal with the verification of PENCIL and OpenCL kernels using program logic.

Chapter 3 is on the specification and verification of parallel loops, and extends the work in Deliverable D6.1 by giving a formal definition of the correctness claims of the method and by proving that they hold. Moreover, the chapter reports on how a new verification backend was integrated into the tool that verifies the specified programs.

Chapter 4 on OpenCL kernel verification focuses on how a parallel loop can be translated to a kernel. This translation preserves not only the computational results, but also the specifications attached to the code. In order to do this, we introduce a variant of the specification method for kernels introduced in Deliverable D6.2. We discuss the difference between the old and the new method.

In both chapters 3 and 4 we include many examples of specified programs that have been verified with the tool.

Translation validation for PENCIL→OpenCL compilation Chapter 5 is devoted to a discussion of how we have conducted translation validation for the PENCIL→OpenCL compiler of WP4. Our initial strategy here was to simply use GPUVerify to prove data race-freedom for the kernels emitted by the compiler. We found that in practice these kernels can be very complex and go beyond what GPUVerify can cope with fully automatically. To overcome this, we have improved the invariant inference capabilities of GPUVerify so that it can cope with many basic properties of these kernels, and we have worked with the compiler team at ENS to extend the PENCIL→OpenCL compiler to emit invariants characterising the access patterns of kernels. These invariants would be very difficult to speculate post-hoc, but are readily available from domain-specific knowledge inside the compiler. We demonstrate that this combination of strengths allows highly automated race-freedom proofs for a set of stencil kernels.

Advancements in automatic OpenCL kernel analysis Chapters 6 and 7 describe two advances to the automated methods for OpenCL kernel analysis proposed in Deliverable D6.2.

Chapter 6 describes *KernelInterceptor*, an add-on to the GPUVerify tool that reduces the need for users to annotate their kernels with preconditions by intercepting running OpenCL applications and taking a snapshot at each kernel entry point. From a run of an OpenCL application, KernelInterceptor produces a set of automatically annotated kernels which can be verified offline by GPUVerify. The method was presented at the 2014 International Workshop on OpenCL [5].

Chapter 7 describes a technique for automatically proving that GPU kernels *terminate*. Termination is an essential property for a GPU kernel, since under current programming models there is no standard manner by which a kernel can communicate partial results to the host application during execution. We present a theorem inspired by the two-thread reduction of GPUVerify (see Deliverable D6.2) showing that termination analysis can be performed soundly with respect to a "one-thread reduction". This leads to a scalable method for proving termination that is effective for data-independent kernels; we present an evaluation using a set of 598 OpenCL and CUDA examples. This work was presented at the 2014 International Workshop on Termination [36].

3 Update on PENCIL Verification

In Deliverable D6.1 a program logic was introduced for parallel loops with dependences. In this chapter, we present a formal semantics for this logic and we explain how a checker for this logic has been implemented in the VerCors tool [9].

3.1 Background

This section briefly recalls some background on the theory of loop dependences and separation logic.

3.1.1 Loop Dependences

For a single loop with multiple statements, several types of loop dependences can be identified. There exists a *dependence* from statement S_{src} to statement S_{sink} in the body of a loop if there exist two iterations i and j of that loop, such that:

- Iteration i is before iteration j, *i.e.*, $i \leq j$.
- If the iterations are the same ($i = j$) then S_{src} must syntactically occur before S_{sink}.
- Statement S_{src} on iteration i and statement S_{sink} on iteration j access the same memory location.
- At least one of these accesses is a write.

The *distance of a dependence* is defined as the difference between j and i.

Loop dependences with distance 0, *i.e.*, when $i = j$, are called *loop independent dependences*. These dependences only have to be considered when the loop body has to be transformed, which is out of the scope of this deliverable.

Loop dependences with a positive distance are called *loop-carried dependences* and are classified into forward and backward dependences. When S_{src} syntactically appears before S_{sink} (or if they are the same statement) there is a *forward loop-carried dependence* and when S_{sink} syntactically appears before S_{src} there is a *backward loop-carried dependence*. The following examples illustrate forward and backward loop-carried dependences.

Example 3.1.1 (Forward Loop Dependence)**.**

```
for(int i=0;i<N;i++){
  S1: a[i] = b[i] + 1;
  S2: if(i>0) c[i] = a[i−1] + 2;
}
```

iteration = 1

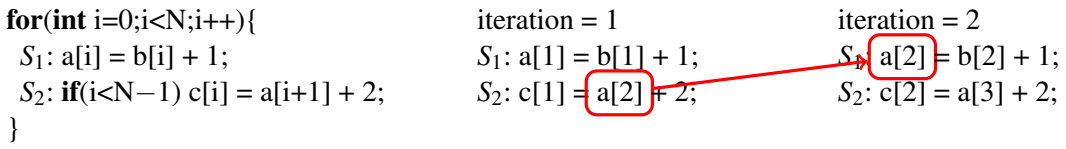
```
S1: a[1] = b[1]+ 1;
S2: c[1] = a[0] + 2;
```

iteration = 2

```
S1: a[2] = b[2] + 1;
S2: c[2] = a[1] + 2;
```

Here, S_1 is the source of the dependence and S_2 is the sink. The i^{th} element of the array a is shared between iteration i and $i-1$, as visualised by the first and second iteration (on the right).

Example 3.1.2 (Backward Loop Dependence)**.**



Here, the sink of the dependence (S_2) appears before the source (S_1) in the body of the loop. Therefore this is a backward loop-carried dependence.

The distinction between forward and backward dependences is important. Independent parallel execution of a loop with dependences is in general unsafe, because it may change the result. For loops with forward dependences only, vectorisation may be applied (see section 3.3.1 for details).

3.1.2 Separation Logic.

Separation logic is described in detail in Chapter 5 of Deliverable D6.2 as a way to reason about OpenCL kernels. However, for the sake of completeness, we give a brief introduction here.

Separation logic [48] was originally developed as an extension of Hoare logic [31] to reason about programs with pointers, as it allows to reason explicitly about the heap. In classical Hoare logic, assertions are properties over the state and no distinction between variables on the heap and variables on the stack is made, while in separation logic, the state is explicitly divided in the heap and a store related to the stack frame of the current method call. Separation logic is also suited to reason modularly about concurrent programs [45]: two threads that operate on disjoint parts of the heap do not interfere, and thus can be verified in isolation.

However, classical separation logic requires use of mutual exclusion mechanisms for all shared locations, and it forbids simultaneous reads to shared locations. To overcome this, Bornat et al. [11] extended separation logic with fractional permissions. Permissions, originally introduced by Boyland [12], denote access rights to a shared location. A full permission 1 denotes a write permission, whereas any fraction in the interval $(0, 1)$ denotes a read permission. Permissions can be split and combined, thus a write permission can be split into multiple read permissions, and all of the read permissions can be joined into a write permission. In this way, data race freedom of programs using different synchronisation mechanisms can be proven. The set of permissions that a thread holds are known as its *resources*.

We write access permissions as $\mathbf{Perm}(e, \pi)$, where e is an expression denoting a memory location and π is a fraction.

In separation logic there are two conjunction operators: *Boolean conjunction* (&&) and *separating conjunction* ($**$). The latter is resource sensitive, the former is not. For example

$\mathbf{Perm}(x, \pi)$ && $\mathbf{Perm}(x, \pi) \equiv \mathbf{Perm}(x, \pi)$
$\mathbf{Perm}(x, \pi) ** \mathbf{Perm}(x, \pi) \equiv \mathbf{Perm}(x, 2 \cdot \pi)$

To specify properties of the value stored at a location we just reference the location in our formulas. To ensure well-definedness of our formulas, we are forced to check that every expression is *self-framed*, i.e., we need to check that only locations are accessed for which we have access permissions. This is different from traditional separation logic, which uses the **PointsTo** primitive that has an additional argument to denote the value stored at the location

```
1   for(int i=0;i<N;i++) /*@
2     requires Perm(a[i],1) ** Perm(c[i],1) ** Perm(b[i],1/2);
3     ensures Perm(a[i],1) ** Perm(c[i],1) ** Perm(b[i],1/2);
4   @*/ {
5     a[i] = b[i] + 1;
6     c[i] = a[i] + 2;
7   }
```

Listing 1: Specification of an Independent Loop

and cannot refer to the location otherwise. However, it has been proven that both logics are equivalent [47].

3.2 A Specification Method for Parallel Loops

In this section, we present a refined version of the specification method for parallel loops with dependences.

3.2.1 Approach

The classical way to specify the effect of a loop is by means of an invariant that has to hold before and after the execution of each iteration in the loop. Unfortunately, this offers no insight into possible parallel executions of the loop. Instead we will consider every iteration of the loop in isolation. To be able to handle dependences, we specify restrictions on how the execution of the statements for each iteration is scheduled. In particular, each iteration is specified by its own contract, *i.e.*, its *iteration contract*. In the iteration contract, the precondition specifies resources that a particular iteration requires and the postcondition specifies the resources which are released after the execution of the iteration. In other words, we treat each iteration as a specified block [29].

Listing 1 gives an example of an *independent loop*, specified by its iteration contract. The contract requires that at the start of iteration i, permission to write both c[i] and a[i] is available, as well as permission to read b[i]. The contract also ensures that these permissions are returned at the end of iteration i. The iteration contract implicitly requires that the separating conjunction of all iteration preconditions holds before the first iteration of the loop, and that the separating conjunction of all iteration postconditions holds after the last iteration of the loop. In Listing 1, the loop iterates from 0 to $N-1$, so the contract implies that before the loop, permission to write the first N elements of both a and c must be available, as well as permission to read the first N elements of b. The same permissions are ensured to be available after termination of the the loop.

To specify *dependent loops*, we need the additional ability to specify what happens when the computations have to synchronise due to a dependence. During such a synchronisation, permissions should be transferred from the iteration containing the source of a dependence to the iteration containing the sink of that dependence. To specify a *permission transfer* we introduce two keywords: **send** and **recv**:

//@ L_S: **send** ϕ **to** L_R, d;

```
1   for(int i=0;i < N;i++) /*@
2     requires Perm(a[i],1) ** Perm(b[i],1/2) ** Perm(c[i],1);
3     ensures Perm(a[i],1/2) ** Perm(b[i],1/2) ** Perm(c[i],1);
4     ensures (i>0 ==> Perm(a[i−1],1/2)) ** (i==N−1 ==> Perm(a[i],1/2));
5   @*/ {
6     a[i]=b[i]+1;
7     /*@
8       S1:if (i< N−1) {
9         send Perm(a[i],1/2) to S2,1;
10      }
11    @*/
12    S2:if (i>0) {
13      //@ recv Perm(a[i−1],1/2) from S1,1;
14      c[i]=a[i−1]+2;
15    }
16  }
```

Listing 2: Specification of a Forward Loop-Carried Dependence

//@ L_R: **recv** ψ **from** L_S, d;

The **send** specifies that (at label L_S) the permissions and properties expressed by the separation logic formula ϕ are transferred to the statement labelled L_R in the iteration $i+d$, where i is the current iteration and d is the distance of dependence. The **recv** specifies that they are received in the form of the formula ψ.

The **send** and **recv** keywords can be used to specify loops with both forward and backward dependences. For example, in Listing 2 we show how our example of a loop with a forward dependence can be annotated with an iteration contract. Listing 3 displays an iteration contract for the backwards dependence example.

We discuss the annotations of the first program in some detail. Each iteration i starts with write permission on a[i] and c[i] as well as read ($\frac{1}{2}$) permission on b[i] The first statement is a write to a[i], which needs write permission. The value written is computed from b[i] for which read permission is needed. The second real statement reads a[i−1], which is not allowed unless read permission is available. This statement is not executed in the first iteration, where this read permission cannot be made available because it refers to a non-existing item of the array. For all subsequent iterations, permission must be transferred. Hence a **send** annotation is specified after the first assignment that transfers a read permission on a[i] to the next iteration (and in addition, keeps a read permission itself). The postcondition of the iteration contract reflects this: it ensures that the original permission on c[i] is released, as well as the read permission on a[i], which was not sent, and also the read permission on a[i−1], which was received. Finally, since the last iteration cannot transfer a read permission on a[i], the iteration contract's postcondition also specifies that the last iteration returns this non-transferred read permission on a[i].

The specifications in both listings are valid. Hence every execution order of the loop bodies that respects the order implied by the **send** annotations yields the same result as sequential execution. In the case of the forward dependence example, this can be achieved by adding appropriate synchronisation in the parallelised code. All parallel iterations should synchronise

```
1    /*@
2      requires Perm(a[i],1/2) ** Perm(b[i],1/2) ** Perm(c[i],1);
3      requires (i==0 ==> Perm(a[i],1/2)) ** (i < N-1 ==> Perm(a[i+1],1/2));
4      ensures Perm(a[i],1/2) ** Perm(a[i],1/2) ** Perm(b[i],1/2) ** Perm(c[i],1);
5    @*/
6      {
7      /*@
8        S1:if (i>0) {
9          recv Perm(a[i],1/2) from S2,1;
10       }
11     @*/
12     a[i]=b[i]+1;
13     S2:if (i < N-1) {
14       c[i]=a[i+1]+2;
15       //@ send Perm(a[i+1],1/2) to S1,1;
16     }
17   }
18 }
```

Listing 3: Specification of a Backward Loop-Carried Dependence

each **send** annotation with the location of the specified label to ensure proper permission transfer. For the backward dependence example, only sequential execution respects the ordering.

3.2.2 Proof Rules and Obligations.

In order to be able to properly define the rules for writing valid specifications with **send** and **recv** as well as to define and prove properties about their semantics, we restrict the kind of loop that we consider to be a non-nested for-loop with K statements that is executed for N iterations. Each of the K statements S_k consists of a block of statements B_k, which is executed if a guard g_k is true. These guards must be expressions that are constant with respect to the execution of the loop iterations. That is, they may not contain any variable that is assigned to in any iteration.

The generic form of loop that we consider is:

for(**int** i=0;i < N;i++){
  S_1: **if** (g_1) { B_1 }
    ⋮
  S_K: **if** (g_K) { B_K }
}

Where the iteration variable i cannot be assigned anywhere in the loop body.

Formally, each block B_k consist of a single atomically executable statement. However, any block B_k with either at most one **recv** at the beginning or at most one **send** at the end and an arbitrary number of atomic program statements is accepted as syntactic sugar. Given a block with multiple statements, we can replace the single guarded statement by several guarded statements by copying the guards. The single **recv** or **send** would inherit the original statement label, while any others would get a new label. Of course, labels which are not used can be

omitted. Similarly, the **if**(**true**) prefixes for a statement that is executed in each iteration is superfluous.

The classical rule for a while loop with an invariant requires that the invariant holds before the loop and ensures that both the invariant and the negation of the loop condition holds afterwards. When a loop is specified with an iteration contract then the requirements are derived form the iteration contract as follows: Before entering the loop, the separating conjunction of the preconditions of all iteration contracts should hold. Afterwards, the conjunction of all post-conditions is ensured:

$$\frac{\{P(i)\}\ body(i)\ \{Q(i)\} \quad \text{for } i = 0 \cdots N-1}{\{\bigstar_{i=0}^{N-1} P(i)\}\ \textbf{for}(\textbf{int}\ \text{i=0;i<N;i++})\ \text{req}\ P(i)\ \text{ens}\ Q(i)\ \{\ body(i)\ \}\ \{\bigstar_{i=0}^{N-1} Q(i)\}}$$

Note that this rule for a loop with an iteration contract is a special case of the rule for parallel execution, which allows arbitrary blocks of code to execute in parallel. (See for example [46].)

The rules for the **send** and **recv** are similar to those for unlock and lock, respectively. (See for example [28].) This is because the **send** is used to give up (unlock) resources that the **recv** acquires (locks). This behaviour is captured in the following two rules:

$$\frac{}{\{P\}\ \textbf{send}\ P\ \textbf{to}\ L,d\ \{\mathsf{true}\}} \qquad \frac{}{\{\mathsf{true}\}\ \textbf{recv}\ P\ \textbf{from}\ L,d\ \{P\}}$$

To prevent the **recv** from creating resources that don't exist, the send and receive statements must occur in matched pairs. That is if block (B_s) ends with the statement

send $\phi(i)$ **to** S_s, d;

then block B_r should start with a receive statement

recv $\psi(i)$ **from** S_r, d;

and the following two requirements should hold. First, if the receive is enabled then d iterations earlier, the send should be enabled:

$$\forall i \in [0,\cdots,N).g_r(i) \implies i \geq d \wedge g_s(i-d) \ . \tag{3.1}$$

Second, the information and resources received should be implied by those sent:

$$\forall i \in [d,\cdots,N).\phi(i-d) \implies \psi(i) \ . \tag{3.2}$$

In the next section, we will consider what properties are guaranteed by a valid specification.

3.3 Proof of Correctness

We will consider three execution paradigms for loops: parallelisable, vectorisable, and sequential. To help the compiler to know to which of these paradigms a particular loop belongs, the programmer may add some loop annotations. In PENCIL, there are **independent** and **ivdep** annotations for parallelisable and vectorisable loops respectively and the absence of annotation means that the loops should be executed sequentially.

To verify the correctness of loop annotations written by a programmer, we use the iteration contracts discussed above. Each class of loop-carried dependencies is matched to a specific execution paradigm. In particular, independent loops, forward and backward loop-carried dependence loops are executable in parallel, vectorised and sequential fashions respectively.

In addition to verification of annotations, our specification language is able to specify the functional behaviour of each iteration separately. Based on these specifications, we are also able to reason about the functional correctness of the specified loop.

In this section, first we define the semantics of three real loop execution paradigms: sequential, vectorised, and parallel. Second, we explain the virtual semantics of the loop which is specified by our specification language. Then we show that the specified loop is data-race free and its functional behaviour is equivalent to the sequential execution. Finally, we prove that a parallelised or a vectorised execution of a particular loop is data-race free if its semantics is derivable from the semantics of the specified loop. This is sufficient to show that vectorised and parallelised loops are data-race free and they preserve the functional behaviour of sequential loops.

3.3.1 Parallel Execution of Loops

The simplest way of parallelising a loop is to simply execute all of its iterations in parallel. To allow the compiler to generate this kind of code for a program, the PENCIL language can annotate a loop with an **independent** pragma. We will denote a *parallel loop* by writing **parfor** instead of just **for**. We will prove later that parallel execution is correct if the body of the specified loop contains no **send** or **recv** statements.

Another way of speeding up computations is to exploit the vector capabilities of hardware. Many processors and co-processors, especially GPUs, can compute with vectors of values instead of with single values. Thus, they can efficiently execute a single statement from several adjacent iterations in parallel. To allow the compiler to generate this kind of code for a program, the PENCIL language can annotate a loop with an **ivdep** pragma. We abstract away from the exact details of how this compilation is done, by considering vectorisation as a program transformation.

Given a loop with N iterations and a divisor V of N, we define the *V-vectorisation* of our loop as follows:

for(**int** i=0;i < N;i+=V){
  parfor(**int** v=0;v<V;v++) $S_1(i+v)$;
  $\vdots$
  parfor(**int** v=0;v<V;v++) $S_K(i+v)$;
}

We will prove that if in the specified program every send statement occurs before the matching receive then for any V, vectorisation of the loop is a correct implementation of that loop. Before discussing the proofs, we will first introduce some notation.

3.3.2 Semantics of Parallel Execution of Loops

Throughout, we assume that every statement is executed atomically. To keep our theory modular, we split the semantics of a program into two layers. The upper layer determines which sequences of atomic statements, called *computations*, a program can make and the second layer defines the effect of each atomic statement. We will not go into the details of the lower layer here because we do not deviate from the standard definitions.

Given our standard loop

for(**int** i=*0*;i < *N*;i++){
  S_1: **if** (g_1) { B_1 }
    ⋮
  S_k: **if** (g_k) { B_k }
}

S_i is defined as an indivisible guarded block of code. Each statement instance is defined as an instantiation of a statement in the loop body in a particular loop iteration. For instance, S_i^j is an instantiation of S_i in the j^{th} iteration of loop. The semantics of a statement instance $[\![S_i^j]\!]$ in the above loop is defined as the atomic execution of the statements in the block of code labelled by S_i^j respecting the evaluation of its guard condition.

The sequential semantics of a program is going to be a single sequence. The semantics of the various parallel execution methods will be sets of computations. To define those sets, we will need to introduce some auxiliary operators: *Concatenation* and *Interleaving*.

The concatenation of two sets of computation is denoted by $++$:

$$C_1 ++ C_2 = \{c_1 \cdot c_2 \mid c_1 \in C_1, c_2 \in C_2\}$$

The concatenation of multiple sets can be written using the following notation:

$$\mathsf{Concat}_{i=1}^{N} C_i = C_1 ++ \cdots ++ C_N$$

To weave several sequences into a set of interleaved sequences, we will use an interleaving operator. We parameterise this definition with a happens before relation $<$, in order to limit the result to sequences in which steps that are supposed to happen before other steps cannot occur in the wrong order. To define the interleaving operator ($\mathsf{Interleave}_<$), we use an auxiliary operator[1] that denotes interleaving with a fixed first step ($\mathsf{Interleave}_<^i$):

$$\begin{array}{lll}
\mathsf{Interleave}_<(c_1,\cdots,c_n) & = \bigcup_{i=1}^{n} \mathsf{Interleave}_<^i(c_1,\cdots,c_n) & \\
\mathsf{Interleave}_<^i(\varepsilon,\cdots,\varepsilon) & = \{\varepsilon\} & \\
\mathsf{Interleave}_<^i(c_1,\cdots \varepsilon \cdots,c_n) & = \emptyset & \\
\mathsf{Interleave}_<^i(c_1,\cdots s_i\, c_i \cdots,c_n) & = \emptyset & , \text{if } \exists s \in (\bigcup_{i=1}^{n} c_i).s < s_i \\
\mathsf{Interleave}_<^i(c_1,\cdots s_i\, c_i \cdots,c_n) & = \{s_i \cdot s \mid s \in \mathsf{Interleave}_<(c_1,\cdots c_i \cdots,c_n)\} & , \text{otherwise}
\end{array}$$

where ε is the empty computation. Again, we have sum-like notation for interleaving N computations:

$$\mathsf{Interleave}_<^{i=1..N} c_i = \mathsf{Interleave}_<(c_1,\cdots,c_n)$$

The most important property that we want to prove is that all of our parallelisations are data-race free. To prove that, we must first define what it means for a computation to be data-race free. To do so, we need to know for every atomic step (t), which locations in memory it writes ($\mathsf{write}(t)$), which locations in memory it reads ($\mathsf{read}(t)$) and by which thread it is executed ($\mathsf{thread}(t)$). We define the set of accessed locations as $\mathsf{access}(t) = \mathsf{write}(t) \cup \mathsf{read}(t)$.

To make the definition easier, we assume that the natural program order is included in the happens before relation, which is a quasi order. We can then define a race condition in a trace as a pair or statements that both access a location and are not ordered by the happens before relation:

[1] The idea came from the axiomatisation of the merge operator in process algebra using the left merge operator.

Definition 3.3.1. *A computation contains a data-race, if it contains two steps s and t, such that*

$$\mathsf{write}(s) \cap \mathsf{access}(t) \neq \emptyset \wedge \neg(s \leq t \vee t \geq s)$$

Besides reasoning about data-race freeness, we will also reason about the end result of a computation. We do so based on the fact that swapping adjacent independent statements does not change the end result.

Proposition 3.3.1. *Swapping two adjacent statements in a data-race free computation, which are unordered in the happens before relation does not change the outcome of the computation.*

Proof. Because the statements are unordered and the computation is data race free, the set of locations written by each of the actions cannot affect the set of locations accessed by the other. Hence neither step can see the effect of the other. □

Similarly, we can ignore steps that do not change the state. A good approximation of the set of steps that change the state is the set of steps that write to at least one location. Thus, we define two computations to be functionally equivalent if they perform the same writing steps in the same order.

Definition 3.3.2. *Given two computations c_1 and c_2. The computations c_1 and c_2 are functionally equivalent if* $\mathsf{mods}(c_1) = \mathsf{mods}(c_2)$*, where*

$$\mathsf{mods}(c) = \begin{cases} \varepsilon & , \text{if } c = \varepsilon \\ \mathsf{mods}(c') & , \text{if } c = t \cdot c' \wedge \mathsf{write}(t) = \emptyset \\ t \cdot \mathsf{mods}(c') & , \text{if } c = t \cdot c' \wedge \mathsf{write}(t) \neq \emptyset \end{cases}$$

To prevent data races, computations must be synchronised. We employ the following four synchronisation steps between the host thread and the worker threads that extend the happens before relation:

fork Host thread computations happen before any computation in any worker thread.

join Worker thread computations happen before host thread computations.

barrier Every worker thread must reach this statement before the first one continues.

send/recv For every matching pair of **send** and **recv**, the **send** statement happens before the **recv** statement.

We consider two happens before orders: *program order* (PO), which maintains the order of statements plus the order of fork, join and barrier statement between threads, and specification order (SO), which extends programs order by also enforcing the ordering of the **send** and **recv** pairs.

Based on the auxiliary operators of Concat and Interleave, we are able to define the semantics of the four different loop executions that we consider in this deliverable. The sequential order simply executes all steps sequentially, the parallel order allows any interleaving that preserves program order within the loop bodies and vectorisation executes multiple iterations in lockstep:

Definition 3.3.3. *Given a loop in standard form: Loop. Let S_i^j be the instance of S_i in the j^{th} iteration.*

- *The semantics of* sequential execution *is*

$$[\![Loop]\!]^{Seq} = \mathsf{Concat}_{j=0}^{N-1}\mathsf{Concat}_{i=0}^{k}[\![S_i^j]\!]$$

- *The semantics of* vectorised execution *for vector length V is defined as*

$$[\![Loop]\!]^{Vec(V)} = \mathsf{Concat}_{j=0}^{(N/V)-1}\mathsf{Concat}_{i=0}^{k}\left(\mathsf{fork}_i^j\left(\mathsf{Interleave}_{\emptyset}^{0..v-1}[\![S_i^j]\!]\right)\mathsf{join}_i^j\right)$$

- *The semantics of* parallel execution *is*

$$[\![Loop]\!]^{Par} = \mathsf{fork}\left(\mathsf{Interleave}_{\mathsf{PO}}^{0..N-1}\mathsf{Concat}_{i=0}^{k}[\![S_i^j]\!]\right)\mathsf{join}$$

- *The semantics of* specified execution *is*

$$[\![Loop]\!]^{Spec} = \mathsf{fork}\left(\mathsf{Interleave}_{\mathsf{SO}}^{0..N-1}\mathsf{Concat}_{i=0}^{k}[\![S_i^j]\!]\right)\mathsf{join}$$

3.3.3 Correctness of Parallel Loops

The key invariants of programs with separation logic specifications are that non-zero permission is held if a read occurs, permission 1 is held if a write occurs and at no time do all of the threads together hold more than permission 1 on a location. Because every **send** has to happen before the matching **recv** this invariant is maintained in the specification order. Thus, every specification order computation is data-race free and has the same result as the sequential computation.

Theorem 3.3.2. *Given a loop with a valid specification.*

1. *All computation in* $[\![Loop]\!]^{Spec}$ *are data-race free.*
2. *All computations in* $[\![Loop]\!]^{Spec}$ *and* $[\![Loop]\!]^{Seq}$ *are functionally equivalent.*

Proof. 1. Because there is a valid specification, we know that for every location the sum over all threads of the permissions held for that location cannot exceed 1. The only way to lose permissions is a **send** and the only way to gain them is a **recv**. Those statements have to happen in that order due to the happens before relation.

Suppose that there are two unordered statements s and t where one writes a location and the other accesses the same location. Then because the specification is valid the thread that executes s has permission 1 and the other thread has permission $p > 0$. So the sum over all threads is at least $1 + p$. Contradiction.

2. We must prove that every computation in $[\![Loop]\!]^{Spec}$ is functionally equivalent to the single computation $[\![Loop]\!]^{Seq}$.
First, the sequential order is one of the orders allowed in the specification order and is functionally equivalent with itself.
Thus, we need to show that all computations in $[\![Loop]\!]^{Spec}$ are mutually equivalent. We do so by showing that given any computation, we can reorder it by swapping independent statements to yields the sequential order, which preserves the end result due to proposition 3.3.1.
Assume that the first n steps of the given computation are in the same order as the sequential computation. Then step t_{n+1} in the sequential has to be at a certain position in

the given sequence. Because each sequence contains the same steps and the sequential computation is in happens before order, all of the steps that have to happen before t_{n+1} are already included in the prefix. Hence, step t_{n+1} is independent of all of the steps after the prefix and before itself in the given sequence and can be swapped with them one-by-one until it is the next step. This grows the number of steps in the sequential order prefix by 1, so we can repeat this.
Thus every computation can be reordered until it matches the sequential order.

□

An immediate corollary is that loops that can be specified without synchronisation are correct too.

Corollary 3.3.3. *Given a loop with a valid specification, that does not make use of* **send** *or* **recv**.

1. *All computation in* $[\![Loop]\!]^{Par}$ *are data-race free.*

2. *All computations in* $[\![Loop]\!]^{Par}$ *and* $[\![Loop]\!]^{Seq}$ *are functionally equivalent.*

Proof. If the specification does not make use of **send** or **recv** then program order coincides with specification order and the result follows from Theorem 3.3.2. □

This proof was easy because the set of parallel order executions was identical to the set of specification order computations. If the specifications use **send** and **recv** then some parallel execution order may contain data races. But if the **send** occurs before the matching **recv** in the loop then vectorisation is possible.

Theorem 3.3.4. *Given a loop with a valid specification, such that every* **send** *occurs before the matching* **recv** *in the body, and* V *that divides* N.

1. *All computations in* $[\![Loop]\!]^{Vec(V)}$ *are data-race free.*

2. *All computations in* $[\![Loop]\!]^{Vec(V)}$ *and* $[\![Loop]\!]^{Seq}$ *are functionally equivalent.*

Proof. Because every **send** occurs before the matching **recv**, every computation that may occur in $[\![Loop]\!]^{Vec(V)}$ can also occur in $[\![Loop]\!]^{Spec}$. That is, we can construct a specification order sequence in which the computational steps occur in the same order and in which the happens-before relation on the vectorised sequence are more restrictive than those in the specification order sequence. Hence all vectorised sequences are data-race free because all specification order sequences are data-race free (Theorem 3.3.2). Moreover, every vectorised computation is functionally equivalent to a specification order sequence and thus functionally equivalent to $[\![Loop]\!]^{Seq}$ (Theorem 3.3.2). □

This completes the proofs of correctness. We continue with a discussion of the implementation.

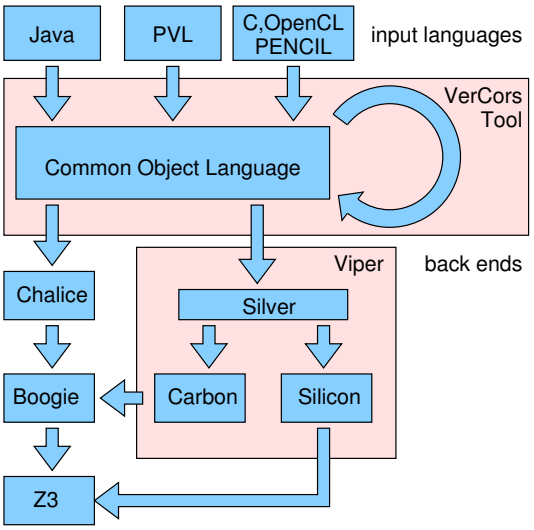


Figure 3.1: Overall architecture VerCors tool set

3.4 Tool Support

This section discusses how our logic for the functional verification of parallel loops, outlined in section 3.2, is implemented in the VerCors tool set. It can be tried online at at `http://www.utwente.nl/vercors`. The VerCors tool was originally developed as a tool to reason about multi-threaded Java programs, but it has been extended in order to be able to reason about OpenCL kernels [10] and parallel loops [8]. It encodes programs in several program transformation steps into simpler programs in either Chalice [38] or in Silver [34]. Figure 3.1 sketches the overall architecture of the tool set.

Chalice is a verifier for an idealised multi-threaded programming language, using permission-based separation logic as a specification language. Chalice in turn gives rise to an encoding in Boogie [6], which gives rise to SMT-compliant proof obligations.

Silver is the intermediate language used for the Viper project [34, 13]. The project maintains two verifiers: Carbon [30], which like Chalice employs verification condition generation and Silicon [35], which like VeriFast [33] employs symbolic execution.

The verifier that is used for the parallel loop specifications is the Silicon verifier which can support quantified permissions on arrays. This support is limited to array permissions of the form

(\forall∗ **int** i; low <= i && i < high ; Perm(array[i],p))

where i may not occur in low and high. Moreover, low, high and p may not be modified in the loop body.

To get the formulas that have to be verified into this form, a number of translations must be applied. To give an idea of how this transformation works, we will consider a few examples based on specifications that we have shown before.

The backend cannot work if the index into the array is different from the quantified variable. So if we use an array index i+c for some constant c then we need to use the fact that

(\forall∗ **int** i; low <= i && i < high ; Perm(array[i+c],p(i)))

is equivalent with

(\forall∗ **int** i; low+c <= i && i < high+c ; Perm(array[i],p(i−c)))

in order to get the formula into the correct from. If there are additional constraint on the array permission then they can in general be moved into the permission expression. That is,

(\forall∗ **int** i; low <= i && i < high ; cond ==> Perm(array[i],p))

is equivalent with

(\forall∗ **int** i; low <= i && i < high ; Perm(array[i],cond?p:none))

in general. However, for specific kinds of conditions it is better to move them into the range check. For example:

(\forall∗ **int** i; low <= i && i < high ; i < high2 ==> Perm(array[i],p))

is equivalent with

(\forall∗ **int** i; low <= i && i < max(high,high2) ; i < high2 ==> Perm(array[i],p))

While both translations fall into the supported category, the special case works better in practice. The reason for that is that while permission expressions are allowed to depend on the quantified variables, problems that use that feature are harder for the backend verifier to deal with than problems that do not.

3.4.1 Encoding into Silicon

The Silicon backend does not know about parallel loops and/or the **send/recv** keywords. Therefore these constructs are encoded by means of methods with contracts. The idea is that every parallel loop

for(**int** i=0;i < *N*;i++)
  requires *pre*(*i*);
  ensures *post*(*i*);
{
  body
}

that occurs somewhere in the code is replaced by a call to the method loop_main, whose contract encodes the application of the Hoare Logic rule for parallel loops:

/∗@ **requires** (**\forall∗ int** i;0<=i && i<N; pre(i));
    ensures (**\forall∗ int** i;0<=i && i<N; post(i)); @∗/
loop_main(**int** N,free(*S*)));

This rule requires that every iteration satisfies the iteration contract. To verify this, we create a method that executes the body of the loop and checks it against the iteration contract for all possible values of the iteration variable:

/∗@ **requires** (0<=i && i<N) ∗∗ pre(i);
    ensures post(i); @∗/
loop_body(**int** i,**int** N,free(*S*))){ *body*; }

Within the body there may be pairs of **send** and **recv** statements.

//@ S_s: **if** $(g_s(i))$ { **send** $\phi(i)$ **to** S_r, d;}
//@ S_r: **if** $(g_r(i))$ { **recv** $\psi(i)$ **from** S_s, d;}

The guards are left in place, but the statements are replaced with method calls

//@ S_s: **if** $(g_s(i))$ { send_s_to_r(i,N,free$(\phi(i))$;}
//@ S_r: **if** $(g_r(i))$ { recv_s_to_r(i,N,free$(\psi(i))$;}

where

requires $\phi(i)$;
send_s_to_r(**int** i,**int** N,free(S)));

ensures $\psi(i)$;
recv_s_to_r(**int** i,**int** N,free(S)));

The proof obligations that protect the valid use of **send** and **recv** are also encoded with methods. The guard requirement 3.1, is encoded as the method

requires 0 <= i && i < N;
requires $g_r(i)$;
ensures d <= i;
ensures $g_s(i-d)$;
void check_guard_send_from_s_to_r(**int** i,...){ }

And the resource requirement 3.2 is checked with the method:

requires d <= i && i < N;
requires $g_r(i)$;
requires $g_s(i-d)$;
requires $\phi(i-d)$;
ensures $g_r(i)$;
ensures $\psi(i)$;
void check_resource_send_from_s_to_r(**int** i,...){ }

3.4.2 Examples

In this section, we discuss our three running examples complete with functional specifications. In Listing 4, we provide a fully specified version of vector addition. Both the contract of the procedure and that of the iteration have been formatted in three parts: the required permissions, the ensured permissions and the ensured functional properties. The permissions on array a have to be write, because the elements of this array are assigned to. The permissions for the other two arrays are $\frac{1}{2}$, which indicates that they are read but not written.

In Listing 5, we show a specified version of the loop with a forward dependence. In this case the specifications consist of four blocks each: required permissions, required functional properties, ensured permissions, and ensured functional properties. To show how functional requirements can be specified and verified, we require that every element of the array b initially contains the index of the element in the array and prove that every well-ordered computation will not change this value, and will put index plus 1 in a and index plus 2 in c, with the exception of the first element of c, which is not assigned and therefore does not contain a known value. Note how the send and receive statements transfer both a read permission and the knowledge about the newly assigned value for the array element a[i−1].

```
1  /*@
2     requires \length(a)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(a[i],1));
3     requires \length(b)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(b[i],1/2));
4     requires \length(c)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(c[i],1/2));
5
6     ensures \length(a)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(a[i],1));
7     ensures \length(b)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(b[i],1/2));
8     ensures \length(c)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(c[i],1/2));
9
10    ensures (\forall int i ; 0 <= i && i < len ; a[i]==b[i]+c[i]);
11    ensures (\forall int i ; 0 <= i && i < len ; b[i]==\old(b[i]));
12    ensures (\forall int i ; 0 <= i && i < len ; c[i]==\old(c[i]));
13 @*/
14 void vector_add(int a[],int b[],int c[],int len){
15    for(int i=0;i < len;i++)
16    /*@
17       requires \length(a)==len ** Perm(a[i],1);
18       requires \length(b)==len ** Perm(b[i],1/2);
19       requires \length(c)==len ** Perm(c[i],1/2);
20
21       ensures \length(a)==len ** Perm(a[i],1);
22       ensures \length(b)==len ** Perm(b[i],1/2);
23       ensures \length(c)==len ** Perm(c[i],1/2);
24
25       ensures b[i]==\old(b[i]);
26       ensures c[i]==\old(c[i]);
27       ensures a[i]==b[i]+c[i];
28    @*/
29    {
30       a[i]=b[i]+c[i];
31    }
32 }
```

Listing 4: Specification of an Independent Loop

```
/*@
  requires \length(a)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(a[i],1));
  requires \length(b)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(b[i],1/2));
  requires \length(c)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(c[i],1));

  requires (\forall int tid; 0 <= tid && tid < len ; b [ tid ] == tid);

  ensures \length(a)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(a[i],1));
  ensures \length(b)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(b[i],1/2));
  ensures \length(c)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(c[i],1));

  ensures (\forall int i; 0 <= i && i < len ; a[i] == i+1);
  ensures (\forall int i; 0 <= i && i < len ; b[i] == i );
  ensures (\forall int i; 0 < i && i < len ; c[i] == i+2);
@*/
void example(int a[],int b[],int c[],int len){
  for(int i=0;i < len;i++) /*@
    requires \length(a)==len ** Perm(a[i],1);
    requires \length(b)==len ** Perm(b[i],1/2);
    requires \length(c)==len ** Perm(c[i],1);

    requires b[i]==i;

    ensures \length(a)==len ** Perm(a[i],1/2);
    ensures \length(b)==len ** Perm(b[i],1/2);
    ensures \length(c)==len ** Perm(c[i],1);
    ensures i>0 ==> Perm(a[i-1],1/2);
    ensures i==\length(a)-1 ==> Perm(a[i],1/2);

    ensures a[i]==i+1;
    ensures b[i]==i;
    ensures i>0 ==> c[i]==i+2;
  @*/ {
    a[i]=b[i]+1;
    /*@
      S1:if (i< len-1) {
        send Perm(a[i],1/2) ** a[i]==i+1 to S2,1;
      }
    @*/
    S2:if (i>0) {
      //@ recv Perm(a[i-1],1/2) ** a[i-1]==i from S1,1;
      c[i]=a[i-1]+2;
    }
  }
```

Listing 5: Specification of a Forward Loop-Carried Dependence

```
class Ref {
  /*@
    requires (\forall* int i ; 0 <= i && i < a.length ; Perm(a[i],1));
    ensures (\forall* int i ; 0 <= i && i < a.length ; Perm(a[i],1));
  @*/
  public void main(int a[]){
    for(int i=0;i<a.length;i++)
    /*@
      requires Perm(a[i],1/2);
      requires i==0 ==> Perm(a[i],1/2);
      requires i < a.length-1 ==> Perm(a[i+1],1/2);
      ensures Perm(a[i],1);
    @*/
    {
      //@ S1:if(i>0){ recv Perm(a[i],1/2) from S2,1; }
      S2:if (i < a.length-1) {
        a[i]=a[i+1];
        //@ send Perm(a[i+1],1/2) to S1,1;
      }
    }
  }
}
```

Listing 6: Specified Array Shift

To show what the tool generated encodings look like, we consider a loop that shifts all elements in an array one to the left, given in Listing 6. The result after encoding the parallel loop and the **send** and **recv** statements can be found in Listing 7. The translation into silver is given in Listing 8.

For completeness, we show a specified version of the loop with a backward dependence in Listing 9. This example has the additional requirement that all elements of the array a have to be 0. The ensures values for a and b are the same, but the ensures values of c is now 2 for every element, except the last element.

```
class Ref{
  /*@
    requires (\forall* int i;i in ([ 0 , a.length ));Perm( a [ i ] , 1 / 2 ));
    requires Perm( a [ 0 ] , 1 / 2 );
    requires (\forall* int i;0 <= i && i < a.length − 1;Perm( a [ (i + 1) ] , 1 / 2 ));
    ensures (\forall* int i;i in ([ 0 , a.length ));Perm( a [ i ] , 1));
  @*/
  void loop_main_11(int[] a);

  /*@
    ensures Perm( a [ i ] , 1 / 2 );
  @*/
  void recv_body_30(int[] a,int i);

  /*@
    requires Perm( a [ (i + 1) ] , 1 / 2 );
  @*/
  void send_body_36(int[] a,int i);

  /*@
    requires i in ([ 0 , a.length ));
    requires Perm( a [ i ] , 1 / 2 );
    requires (i == 0 ==> Perm( a [ i ] , 1 / 2 ));
    requires (i < a.length − 1 ==> Perm( a [ (i + 1) ] , 1 / 2 ));
    ensures i in ([ 0 , a.length ));
    ensures Perm( a [ i ] , 1);
  @*/
  void loop_body_11(int[] a,int i){
    /*@
      S1:if (i > 0) {
        recv_body_30(a,i);
      @*/
    }
  @*/
  S2:if (i < a.length − 1) {
    a [ i ]=a [ (i + 1) ];
    /*@
      send_body_36(a,i);
    @*/
  }
}

/*@
  requires i in ([ 0 , a.length ));
  requires i > 0;
  ensures 1 <= i;
  ensures i − 1 < a.length − 1;
@*/
void guard_check_S2_S1(int[] a,int i){
}

/*@
  requires i in ([ 0 , a.length ));
  requires i − 1 < a.length − 1;
  requires i > 0;
  requires Perm( a [ (i − 1 + 1) ] , 1 / 2 );
  ensures i − 1 < a.length − 1;
  ensures Perm( a [ i ] , 1 / 2 );
@*/
void resource_check_S2_S1(int[] a,int i){
}

/*@
  requires (\forall* int i;0 <= i && i < a.length;Perm( a [ i ] , 1));
  ensures (\forall* int i;0 <= i && i < a.length;Perm( a [ i ] , 1));
@*/
void main(int[] a){
  loop_main_11(a);
}

Ref(){
}
```

Listing 7: Encoded Specification of Array Shift

```
field Integer_value: Int

method loop_main_11(this: Ref, a: Seq[Ref])
  requires (forall i: Int :: (i in [0..|a|)) ==> acc(a[i].Integer_value, 1 / 2))
  requires acc(a[0].Integer_value, 1 / 2)
  requires (forall i: Int :: (i in [0 + 1..|a| − 1 + 1)) ==> acc(a[i].Integer_value, 1 / 2))
  ensures (forall i: Int :: (i in [0..|a|)) ==> acc(a[i].Integer_value, 1))
{
  inhale false
}

method recv_body_30(this: Ref, a: Seq[Ref], i: Int)
  ensures acc(a[i].Integer_value, 1 / 2)
{
  inhale false
}

method send_body_36(this: Ref, a: Seq[Ref], i: Int)
  requires acc(a[i + 1].Integer_value, 1 / 2)
{
  inhale false
}

method loop_body_11(this: Ref, a: Seq[Ref], i: Int)
  requires (i in [0..|a|))
  requires acc(a[i].Integer_value, 1 / 2)
  requires (i == 0) ==> acc(a[i].Integer_value, 1 / 2)
  requires (i < |a| − 1) ==> acc(a[i + 1].Integer_value, 1 / 2)
  ensures (i in [0..|a|))
  ensures acc(a[i].Integer_value, 1)
{
  if (i > 0) {
    recv_body_30(this, a, i)
  }
  if (i < |a| − 1) {
    a[i].Integer_value := a[i + 1].Integer_value
    send_body_36(this, a, i)
  }
}

method guard_check_S2_S1(this: Ref, a: Seq[Ref], i: Int)
  requires (i in [0..|a|))
  requires i > 0
  ensures 1 <= i
  ensures i − 1 < |a| − 1
{
}

method resource_check_S2_S1(this: Ref, a: Seq[Ref], i: Int)
  requires (i in [0..|a|))
  requires i − 1 < |a| − 1
  requires i > 0
  requires acc(a[i − 1 + 1].Integer_value, 1 / 2)
  ensures i − 1 < |a| − 1
  ensures acc(a[i].Integer_value, 1 / 2)
{
}

method main(this: Ref, a: Seq[Ref])
  requires (forall i: Int :: (i in [0..|a|)) ==> acc(a[i].Integer_value, 1))
  ensures (forall i: Int :: (i in [0..|a|)) ==> acc(a[i].Integer_value, 1))
{
  loop_main_11(this, a)
}
```

Listing 8: Silver Specification of an Array Shift

```
/*@
   requires \length(a)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(a[i],1));
   requires \length(b)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(b[i],1/2));
   requires \length(c)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(c[i],1));

   requires (\forall int tid; 0 <= tid && tid < len ; a [ tid ] == 0);
   requires (\forall int tid; 0 <= tid && tid < len ; b [ tid ] == tid);

   ensures \length(a)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(a[i],1));
   ensures \length(b)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(b[i],1/2));
   ensures \length(c)==len ** (\forall* int i ; 0 <= i && i < len ; Perm(c[i],1));

   ensures (\forall int i; 0 <= i && i < len ; a[i] == i+1);
   ensures (\forall int i; 0 <= i && i < len ; b[i] == i );
   ensures (\forall int i; 0 <= i && i < len ; c[i] == 2 );
@*/
void example(int a[],int b[],int c[],int len){
   for(int i=0;i < len;i++)
    /*@
      requires \length(a)==len ** Perm(a[i],1/2);
      requires i==0 ==> Perm(a[i],1/2);
      requires i < len-1 ==> Perm(a[i+1],1/2);
      requires \length(b)==len ** Perm(b[i],1/2);
      requires \length(c)==len ** Perm(c[i],1);
      requires i < len-1 ==> a[i+1]==0;
      requires b[i]==i;

      ensures \length(a)==len ** Perm(a[i],1);
      ensures \length(b)==len ** Perm(b[i],1/2);
      ensures \length(c)==len ** Perm(c[i],1);
      ensures a[i]==i+1;
      ensures b[i]==i;
      ensures i < len-1 ==> c[i]==2;
    @*/
      {
      /*@
         S1:if (i>0) {
            recv i == (i-1)+1 ** \length(a)==len ** Perm(a[i],1/2) from S2,1;
         }
       @*/
      a[i]=b[i]+1;
      S2:if (i < len-1) {
         c[i]=a[i+1]+2;
         //@ send \length(a)==len ** Perm(a[i+1],1/2) to S1,1;
      }
   }
}
```

Listing 9: Specification of a Backward Loop-Carried Dependence

4 Update on Verification of OpenCL Using Permission-Based Separation Logic

In this chapter, we briefly explain how the specification method for kernels in Deliverable D6.2 works. We then explain how that specification method is modified slightly in order to both provide shorter specifications for functional specifications and allow a straightforward translation of vectorisable parallel loops to kernels that includes translating the specifications.

4.1 Kernel Specification and Verification

To experiment with specification and verification of kernels, we use a prototyping language whose syntax is based on Java. The specification language built into the language is based on Separation Logic and JML. The programming model behind OpenCL kernels is based on parallel execution with **tcount** threads, which are subdivided into **gcount** work groups that have **gsize** threads each. The code written is the code execution by one thread, which knows to which workgroup it belongs (**gid**) and also knows its own thread identity both within the group of all threads (**tid**) and within the workgroup it belongs to (**lid**). Two kinds of memory are available to threads in kernels: global memory which is shared between all threads in the kernel and local memory which is shared between threads in a workgroup and not accessible to threads from other workgroups. Threads within the same working group are guaranteed to run at the same time and they can synchronise using barriers. These barriers can be marked to indicate if they should act as memory barriers for the local and/or the global memory.

The specification of a kernel is written in the same way as a specification of a method is written. That is, a kernel has a contract which states the pre- and post-condition of one thread and loop invariants have to be provided. In addition barriers have to be annotated with a contract. In Deliverable D6.2, we proposed a *permission redistribution* interpretation for barrier contracts. In this interpretation, the contract of a barrier states which properties must be true before a barrier, which properties must be true after the barrier and which permissions will be held after the barrier. For a barrier contract to be valid, for each workgroup the conjunction of the pre-conditions over the thread in the workgroup must imply the conjunction of the post-conditions and the separating conjunction of the permission provided to the threads in the workgroup has to be greater than the separating conjunction of the permissions to be held after the barrier.

Under these rules, specifications can be short because it is not necessary to specify which permissions are given up during a barrier. Unfortunately, because all permissions are automatically revoked by a barrier that also means that all knowledge about the values contained in locations is lost. Thus, writing a functional specification under these rules makes the specifications longer because all information has to be repeated at every barrier. Hence, we also consider the *permission exchange* interpretation for barrier contracts. In this interpretation, the contract of a barrier states which permissions are to be given to other threads and which properties holds for the location to which permissions are provided and then ensures the same permissions and properties for the threads that must receive them. For a barrier contract to be valid, for each workgroup the conjunction of the pre-conditions over the thread in the workgroup must again

```
kernel Ref {
  global int[tcount] a;
  global int[tcount] b;
  global int[tcount] c;

  requires Perm(a[tid],1);
  requires Perm(b[tid],1);
  requires Perm(c[tid],1/4);

  void main(){
    b[tid]=c[tid];
    barrier(global){
      ensures Perm(a[tid],1);
      ensures Perm(b[tid],1/4);
      ensures tid>0 ==> Perm(b[tid−1],1/4);
    }
    if(tid>0) {
      a[tid]=b[tid−1]+b[tid];
    } else {
      a[tid]=b[tid];
    }
  }
}
```

Listing 10: Data race freedom specification.

imply the conjunction of the post-conditions and the separating conjunction of the permission provided to the barrier has to be greater than the separating conjunction of the permissions to be held after the barrier.

We will show the effect of this change with an example. In Listing 10 we have written a redistributing barrier specified version of kernel with a barrier, which first copies the array c[**tid**] to b[**tid**] and then assigns the sum of b[**tid**−1] and b[**tid**] to a[**tid**]. The specification is valid and proves that the kernel is data race free. To do so the barrier has to specify permissions needed for the statement after the barrier only: permission to write a[**tid**] and permission to read b[**tid**−1] and b[**tid**].

In addition, we want to prove that if all elements of c are positive at the start then all three array are positive at the end:

```
requires Perm(c[tid],1/4) ∗∗ c[tid] > 0;
ensures Perm(a[tid],1/4) ∗∗ a[tid] > 0;
ensures Perm(b[tid],1/4) ∗∗ b[tid] > 0;
ensures Perm(c[tid],1/4) ∗∗ c[tid] > 0;
```

To achieve this, the barrier specification has to be extended to require and ensure the fact that the elements of b and c are positive:

```
    barrier(global){
      requires b[tid]>0;
```

```
    requires c[tid]>0;
    ensures Perm(a[tid],1);
    ensures Perm(b[tid],1/4);
    ensures Perm(c[tid],1/4);
    ensures tid>0 ==> Perm(b[tid−1],1/4);
    ensures b[tid]>0;
    ensures tid>0 ==> b[tid−1]>0;
    ensures c[tid]>0;
  }
```

this requires an additional 6 clauses in the specification. The exchanging barrier specification for the same program is

```
  barrier(global){
    requires Perm(b[tid],1/4);
    requires b[tid]>0;
    ensures tid>0 ==> Perm(b[tid−1],1/4);
    ensures tid>0 ==> b[tid−1]>0;
  }
```

which is considerably shorter.

4.2 Compilation of Parallel Loops to Kernels

One of the possibilities for compiling loops is to rewrite them as kernels. Given an independent loop, the process is very simple: you create a kernel with as many threads as there are loop iterations and each kernel thread executes one iteration. Moreover, if the loop was specified with an iteration contract then the iteration contract can be used as the kernel contract. The size of the workgroup can be chosen at will because no barriers are used. For example, in Listing 11 we list the specified code of a kernel that implements the parallel loop in Listing 4.

If the loop has forward dependences then the kernel must mimic vectorised execution of the loop. Consider the specified parallel loop

```
for(int i=0;i < N;i++)
  requires φ(i);
  ensures ψ(i);
{
  S1: if (g1(i)) { B1(i); }
    ⋮
  SK: if (gK(i)) { BK(i); }
}
```

and assume that it has forward dependences only. For now, let us assume that both the number of threads and the size of the working group are N. Then we translate the loop to the kernel

```
  requires φ(tid);
  ensures ψ(tid);
loop()
{
  C1(tid);
```

```
1  kernel Ref {
2    global int[tcount] a;
3    global int[tcount] b;
4    global int[tcount] c;
5
6    requires Perm(a[tid],1);
7    requires Perm(b[tid],1/2);
8    requires Perm(c[tid],1/2);
9
10   ensures Perm(a[tid],1);
11   ensures Perm(b[tid],1/2);
12   ensures Perm(c[tid],1/2);
13
14   ensures b[tid]==\old(b[tid]);
15   ensures c[tid]==\old(c[tid]);
16   ensures a[tid]==b[tid]+c[tid];
17
18   void main(){
19     a[tid]=b[tid]+c[tid];
20   }
21 }
```

Listing 11: Kernel Implementing an Independent Loop

$\vdots$

C_K(**tid**);
}

where

- if B_k is a **send** statement then it is ignored

  $C_k(i) \equiv \{\}$

- if $B_k(i)$ is a **recv** statement with a matching **send**

  S_j: **if** $(g_j(i))$ { **send** $\phi_S(i)$ **to** S_k, d; }
  S_k: **if** $(g_k(i))$ { **recv** $\phi_R(i)$ **from** S_j, d; }

  it is replaced by a barrier

  $C_k(i) \equiv$
  barrier(){
  requires $g_j(i)$ ==> $\phi_S(i)$;
  ensures $g_k(i)$ ==> $\phi_R(i)$;
  }

  where the contract has to be considered in the exchange style.

- if B_k is any other statement then it is copied

  $C_k(i) \equiv$ **if** $(g_1(i))$ { $B_1(i)$; }

```
1  kernel Ref {
2    global int[tcount] a;
3    global int[tcount] b;
4    global int[tcount] c;
5
6    requires Perm(a[tid],1);
7    requires Perm(b[tid],1/2);
8    requires Perm(c[tid],1);
9
10   requires b[tid]==tid;
11
12   ensures Perm(a[tid],1/2);
13   ensures Perm(b[tid],1/2);
14   ensures Perm(c[tid],1);
15   ensures tid>0 ==> Perm(a[tid−1],1/2);
16   ensures tid==tcount−1 ==> Perm(a[tid],1/2);
17
18   ensures a[tid]==tid+1;
19   ensures b[tid]==tid;
20   ensures tid>0 ==> c[tid]==tid+2;
21
22   void main(){
23     a[tid]=b[tid]+1;
24     barrier(global){
25       requires tid<tcount−1 ==> Perm(a[tid],1/2);
26       requires tid<tcount−1 ==> a[tid]==tid+1;
27       ensures tid>0 ==> Perm(a[tid−1],1/2);
28       ensures tid>0 ==> a[tid−1]==tid;
29     }
30     if (tid>0) {
31       c[tid]=a[tid−1]+2;
32     }
33   }
34 }
```

Listing 12: Kernel implementing the Forward Dependent Loop, alternate barrier semantics

In Listing 12 we show the kernel that is derived in this way from the forward dependence example in Listing 5. For comparison, we have also written a kernel specification that uses a redistribution style contract. This kernel can be found in Listing 13. Note that the barrier specification in the first case is not only much shorter because instead of having to specify all information, only the modified information has to be specified. Moreover, while the first specification was derived form the **send** and **recv** specifications, the barrier contract in the second example had to be extended with clauses that state that a and b did not change. As long as the formulas involved do not use recursively defined predicates then these clauses should be derivable in a fully automatic way. As soon as recursive predicates are used, the problem is likely to be undecidable. Moreover, it makes the contracts more complicated than necessary.

4.3 Conclusion

We have introduced a new interpretation of barrier contracts, which is used when iteration contracts are translated to kernels contracts. We have compared the old redistribution interpretation with the new exchange interpretation using examples that were manually derived from the forward loop dependence example in the previous chapter. We have described how to construct a specified kernel given a specified parallel loop, for the special case of the number of threads being equal to the number of loop iterations. It is future work to adapt this translation for the number of threads being a divisor of the number of iterations and to implement the transformation.

```
1  kernel Ref {
2    global int[tcount] a;
3    global int[tcount] b;
4    global int[tcount] c;
5
6    requires Perm(a[tid],1);
7    requires Perm(b[tid],1/2);
8    requires Perm(c[tid],1);
9
10   requires b[tid]==tid;
11
12   ensures Perm(a[tid],1/2);
13   ensures Perm(b[tid],1/2);
14   ensures Perm(c[tid],1);
15
16   ensures a[tid]==tid+1;
17   ensures b[tid]==tid;
18   ensures tid>0 ==> c[tid]==tid+2;
19
20   void main(){
21     a[tid]=b[tid]+1;
22     barrier(global){
23       requires a[tid]==tid+1;
24       ensures tid>0 ==> Perm(a[tid−1],1/2);
25       ensures tid>0 ==> a[tid−1]==tid;
26
27       ensures Perm(a[tid],1/2);
28       ensures Perm(b[tid],1/2);
29       ensures Perm(c[tid],1);
30       ensures a[tid]==\old(a[tid]);
31       ensures b[tid]==\old(b[tid]);
32     }
33     if (tid>0) {
34       c[tid]=a[tid−1]+2;
35     }
36   }
37 }
```

Listing 13: Kernel implementing the Forward Dependent Loop

5 Translation Validation for the PENCIL→OpenCL Compiler

The PENCIL→OpenCL compiler of WP4 takes a high-level description of a computation, expressed in the PENCIL language [2] (see Deliverable D3.5) and uses polyhedral compilation techniques [25, 24, 51] to generate optimised OpenCL [37] code to run on a variety of GPU and CPU platforms.

For correctness of the generated code, and to ensure semantic equivalence across platforms, it is essential that the generated OpenCL is free from data races.

If a PENCIL program is expressed without the use of *summary functions* or *assume statements* (see Deliverable D3.5) then the PENCIL→OpenCL compiler in principle guarantees that generated code is race-free. Nevertheless, the compiler is a sophisticated and complex tool, and compiler bugs are an accepted reality [52]. There is thus value in performing translation validation to check that the generated OpenCL code is indeed free from data races.

In cases where the polyhedral analysis performed by the compiler is not sufficient to enable desired optimisations, the programmer can annotate a PENCIL application using summary functions and assume statements. This allows known facts from a given application domain to be fed to the compiler. The compiler takes these *metadata* facts on trust and may exploit them to generate efficient code. A compelling potential use of metadata is in the use of PENCIL as a target language for DSL compilers. A DSL→PENCIL compiler (for instance, the VOBLA→PENCIL compiler described in Deliverable D3.3) may emit metadata in the generated code capturing facts that were readily available at the DSL level but which would be hard to recover through general-purpose program analysis of PENCIL code. The risk of exploiting metadata is that should the metadata be *wrong*, the PENCIL→OpenCL compiler may produce meaningless OpenCL code. One manner in which this meaninglessness may manifest itself is through data races, thus the translation validation methods proposed here could be useful in flagging cases of incorrect metadata.

Our initially idea for providing translation validation for race-freedom was to extend the GPUVerify method of Deliverable D6.2 to enable fully automatic analysis of compiler-generated code. We discuss in Section 5.1, using an example, that this approach proved possible in principle, but extremely challenging in practice. We could see an ad-hoc path for improving automation of the analysis on a case-by-case basis, but were troubled by the lack of a general principle for handling arbitrary kernels generated by the compiler.

The ICL team communicated this initial experience to the compiler team at ENS, explaining the sorts of invariants GPUVerify requires in order to prove race-freedom, and how these invariants must be expressed. For PENCIL programs that are not equipped with metadata (i.e., programs that can be handled directly using polyhedral methods) the ENS team established that the invariants required by GPUVerify can be extracted directly from the dependence information on which polyhedral code generation is based. For this class of programs the ENS team were able to extend the PENCIL→OpenCL compiler to automatically generate an invariant sufficient for proving race-freedom. In some sense, this invariant is a *certificate* from the compiler that it has produced race-free code. Crucially, GPUVerify does not take this certificate on trust. Rather, GPUVerify simultaneously (a) proves that the certificate is valid (that the specified invariant is indeed an invariant), and (b) uses the certificate in a proof of race-freedom for the kernel. We describe this process of integrating GPUVerify with the PENCIL→OpenCL tools in Section 5.2.

```
__kernel void kernel0(__global double *A, __global double *B, int n, int tsteps, int h0)
{
    int b0 = get_group_id(0);
    int t0 = get_local_id(0);
    __local double shared_A[32][32];

    define min(x,y) ((x) < (y) ?  (x) :  (y))define max(x,y) ((x) > (y) ? (x) : (y))
    for (int g1 = 32 * b0; g1 < n; g1 += 1048576) {
      for (int g5 = 0; g5 < n; g5 += 32) {
        if (n >= t0 + g5 + 1) {
          for (int c0 = 0; c0 <= min(31, n - g1 - 1); c0 += 1) {
            shared_A[c0][t0] = A[(g1 + c0) * n + (t0 + g5)];
          }
        }
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
        if (n >= t0 + g1 + 1) {
          for (int c2 = max(-g5 + 1, 0); c2 <= min(31, n - g5 - 1); c2 += 1) {
              B[(t0 + g1) * n + (g5 + c2)] =
                (B[(t0 + g1) * n + (g5 + c2)] - ((shared_A[t0][c2] * shared_A[t0][c2]) /
                 B[(t0 + g1) * n + (g5 + c2 - 1)]));
          }
        }
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
      }
    }
}
```

Figure 5.1: An example kernel generated by the PENCIL→OpenCL compiler.

We have not yet applied the integrated setup to PENCIL programs that are annotated with metadata. In such cases we believe that analysis using GPUVerify may be capable of identifying *incorrect* metadata. On the other hand, for the same reasons that the given metadata could not be derived automatically by the PENCIL→OpenCL compiler, it may be the case that GPUVerify is similarly unable to prove race-freedom of a kernel that has been optimised based on metadata annotations.

In Section 5.3 we describe an experience report using this setup to prove race-freedom for a set of stencil kernels generated by the PENCIL→OpenCL compiler. We demonstrate that analysis is highly automatic, and argue that with some additional engineering the process can be made completely automatic.

5.1 Manual verification of a compiler-generated kernel

Figure 5.1 shows an example OpenCL kernel generated by the PENCIL→OpenCL compiler of WP4. The kernel performs one stage of a stencil computation, processing data from input array `A` into output array `B`.

Details of the kernel's behaviour are not important for the discussion here; what is important to note about this kernel is that it contains a significant number of loops (four), and that concurrent threads write to the global array `B` and the local array `shared_A`.

In order to prove that the kernel is free from data races, GPUVerify must determine that whenever a thread may access `B` or `shared_A` at a given index, no other thread can write to the array at this index.

To conduct this reasoning in the presence of loops, GPUVerify requires *loop invariants* to abstractly describe the effect of an arbitrary number of loop iterations. As discussed in

Deliverable D6.2, and studied in detail in [7], GPUVerify uses the Houdini method [26] for loop invariant inference. Our approach to loop invariant generation in GPUVerify using Houdini has been to manually determine invariants for a number of examples and then equip GPUVerify with rules to automatically guess the types of invariants that crop up frequently.

We spent some time attempting to verify kernels such as the example of Figure 5.1 through manual invariant annotation. Our efforts were successful in the sense that, with enough perseverance, we could eventually determine suitable invariants. However, the process was time-consuming and did not yield obvious insights into how to generate the required invariants automatically.

As an example, Figure 5.2 shows the kernel of Figure 5.1 after we annotated it with invariants necessary for verification of race-freedom to succeed. The purpose of Figure 5.1 is to illustrate the scale of the invariant generation challenge these kernels present; specific details of the invariants (the lines starting with `__global_invariant`) are not important for this discussion.

We were able to extend GPUVerify to automatically infer many of the simpler invariants in Figure 5.2, but we did not manage to devise a general-purpose mechanism for inferring invariants characterising the access pattern for the array `B`; these are the invariants containing `__read_implies(B, ...)` and `__write_implies(B, ...)`.

5.2 Extending the PENCIL→OpenCL compiler to issue a certificate of race-freedom

Having understood the sorts of invariants required to verify a collection of examples, such as the annotated kernel of Figure 5.2, the ICL team approached the team at ENS developing the PENCIL→OpenCL compiler for advice on how to understand the access patterns of generated kernels in general, in order to come up with an automated method for invariant generation.

After some discussion the teams established that the PENCIL→OpenCL compiler has, during compilation, a precise representation of the access pattern for the kernel that is being generated, thus we explored the idea of having the compiler emit an invariant characterising this access pattern, rather than having GPUVerify attempt to discover such an invariant post-hoc via program analysis.

To this end, we equipped GPUVerify with a new annotation construct, `__function_wide_invariant`, for "function-wide" invariants. A function-wide invariant annotation is a formula that should be attached as an invariant to every loop inside the function where the annotation appears. The PENCIL→OpenCL compiler then generates two function-wide invariants per global array used by the kernel, one characterising the locations to which a thread may write to the array, the other the locations from which a thread may read from the array.

Figure 5.3 shows an example function-wide invariant generated by the compiler, characterising writes to a global array `B`. As with the examples of Figures 5.1 and 5.2, we are not concerned here with the precise details of Figure 5.3; rather, the purpose of the figure is to illustrate that the compiler is able to automatically emit a precise invariant that would be challenging to recover from the source code of the emitted kernel.

It is important to note that the function-wide invariant emitted by the compiler is *not* taken by GPUVerify on trust. Instead, GPUVerify takes responsibility for both (a) checking that the provided invariant is indeed an invariant, and (b) using the provided invariant to prove race-freedom for the kernel.

Currently the function-wide invariant issued by the PENCIL→OpenCL compiler does not

```
__kernel void kernel0(__global double *A, __global double *B, int n, int tsteps, int h0)
{
    __requires(tsteps == 32);
    __requires(n == 1024);
    int b0 = get_group_id(0);
    int t0 = get_local_id(0);
    __local double shared_A[32][32];

    define min(x,y) ((x) < (y) ?  (x) :  (y))define max(x,y) ((x) > (y) ? (x) : (y))
    for (int g1 = 32 * b0;
         __global_invariant(__write_implies(B,
           (((__write_offset_bytes(B)/sizeof(double))/n) % 1048576) == 32 * b0 + t0)),
         __global_invariant(__read_implies(B,
           (((__read_offset_bytes(B)/sizeof(double))/n) % 1048576) == 32 * b0 + t0)),
         __global_invariant(__implies(g1 < n, __enabled())),
         __global_invariant(__implies(__same_group, __no_read(shared_A))),
         __global_invariant(__implies(__same_group, __no_write(shared_A))),
         g1 < n; g1 += 1048576) {
      for (int g5 = 0;
           __global_invariant(__write_implies(B,
             (((__write_offset_bytes(B)/sizeof(double))/n) % 1048576) == 32 * b0 + t0)),
           __global_invariant(__read_implies(B,
             (((__read_offset_bytes(B)/sizeof(double))/n) % 1048576) == 32 * b0 + t0)),
           __global_invariant(__implies((g1 < n) & (g5 < n), __enabled())),
           __global_invariant(__implies(__same_group, __no_read(shared_A))),
           __global_invariant(__implies(__same_group, __no_write(shared_A))),
           g5 < n; g5 += 32) {
        if (n >= t0 + g5 + 1) {
          for (int c0 = 0;
               __global_invariant(__implies(__same_group & __write(shared_A),
                 (g1 < n) & (g5 < n) & (n >= t0 + g5 + 1))),
               c0 <= min(31, n - g1 - 1); c0 += 1) {
            shared_A[c0][t0] = A[(g1 + c0) * n + (t0 + g5)];
          }
        }
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
        if (n >= t0 + g1 + 1) {
          for (int c2 = max(-g5 + 1, 0);
               __invariant(c2 >= max(-g5 + 1, 0)),
               __global_invariant(__write_implies(B,
                 (((__write_offset_bytes(B)/sizeof(double))/n) % 1048576) == 32 * b0 + t0)),
               __global_invariant(__read_implies(B,
                 (((__read_offset_bytes(B)/sizeof(double))/n) % 1048576) == 32 * b0 + t0)),
               __global_invariant(__implies(__same_group, __no_write(shared_A))),
               __global_invariant(__implies(__same_group & __read(shared_A),
                 (g1 < n) & (g5 < n) & (n >= t0 + g1 + 1))),
               c2 <= min(31, n - g5 - 1); c2 += 1) {
              B[(t0 + g1) * n + (g5 + c2)] =
                (B[(t0 + g1) * n + (g5 + c2)] - ((shared_A[t0][c2] * shared_A[t0][c2]) /
                 B[(t0 + g1) * n + (g5 + c2 - 1)]));
          }
        }
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
      }
    }
}
```

Figure 5.2: The kernel of Figure 5.1 annotated with invariants that allows race-freedom to be proven.

```
__function_wide_invariant(
  __write_implies(B,
      32 * b0 + t0 >= 0
    & 32 * b0 + t0 <= 1048575
    & n >= (__write_offset_bytes(B) / sizeof(double) / n % n) + 1
    & __write_offset_bytes(B) / sizeof(double) / n % n >= 0
    & __write_offset_bytes(B) / sizeof(double) % n >= 1
    & n >= (__write_offset_bytes(B) / sizeof(double) % n) + 1
    & (32 * b0 + t0 - (__write_offset_bytes(B) / sizeof(double) / n % n)) % 1048576 == 0
  )
);
```

Figure 5.3: An example function-wide invariant emitted by the PENCIL→OpenCL compiler.

generate an invariant for race-freedom on OpenCL `local` memory regions; this is left for future work.

5.3 Analysing a set of compiler-generated stencil kernels

We applied the PENCIL→OpenCL compiler to the `adi` stencil example from the PolyBench benchmark suite [27].

This leads to the generation of five OpenCL kernels expressing various stages of this stencil computation. Each kernel is equipped with a function-wide invariant generated by the compiler.

Using GPUVerify we were able to verify race-freedom for four of the five kernels fully automatically. The fifth kernel required one auxiliary invariant related to the values that may be taken by a loop counter to be manually supplied. For fully automatic generation we must either extend GPUVerify to infer this invariant automatically, or extend the invariants emitted by the PENCIL→OpenCL compiler so that facts about loop counters are also captured.

As part of the validation phase of the CARP project we will assess translation validation for larger set of examples.

6 KernelInterceptor: Dynamic Interception of OpenCL Kernels for Offline Verification

The GPUVerify tool [7], presented in Deliverable D6.2 and discussed with application to translation validation in Chapter 5, can be used to verify that GPU kernels, written in either CUDA[1] or OpenCL,[2] are free from data races and barrier divergence. The analysis is performed *statically*; that is, GPUVerify does not actually run the kernel, but merely examines its source code. GPUVerify is useful for discovering defects in kernels, but can also go further than any testing tool can: it is able to certify that a given kernel is free from data races and barrier divergence under *any* execution schedule. GPUVerify has already proved itself to be of practical use when applied to non-trivial OpenCL and CUDA kernels [7]. For instance, it is able to verify, without user intervention, 49 of the 70 kernels in version 2.6 of the AMD Accelerated Parallel Processing SDK.[3]

GPUVerify is intended as a completely-automatic tool, requiring minimal expertise and minimal effort from its users. However, assembling all of the necessary inputs to GPUVerify is a significant manual effort. The user must examine the source code of their application, and supply to GPUVerify:

- the source code of each kernel,
- the precise number of work items and work groups that will execute each kernel,
- constraints on the values of selected kernel arguments (where necessary for kernel correctness), and
- barrier invariants [15] and loop invariants (where necessary for successful verification).

In this chapter we describe an extension to GPUVerify, called KernelInterceptor, that automates the extraction of the first three items above from a given OpenCL application. The fourth item, invariant discovery, remains a challenging research topic, as discussed in Section 6.3. Nevertheless, KernelInterceptor marks a significant step towards fully automated verification of GPU kernels.

KernelInterceptor is used as follows.

1. **The user prepares an application for interception.** Small modifications must be made to the source code and build process of the OpenCL application to be analysed.
2. **The user executes the application.** As the application executes, KernelInterceptor intercepts each kernel launch and records the kernel's source code and the parameters passed.
3. **The user executes GPUVerify.** GPUVerify presents a list of intercepted kernels. The user can then ask GPUVerify to try to verify all or some of these kernels.

[1] `http://www.nvidia.com/object/cuda_home_new.html`
[2] `http://www.khronos.org/opencl/`
[3] `http://developer.amd.com/sdks/amdappsdk`

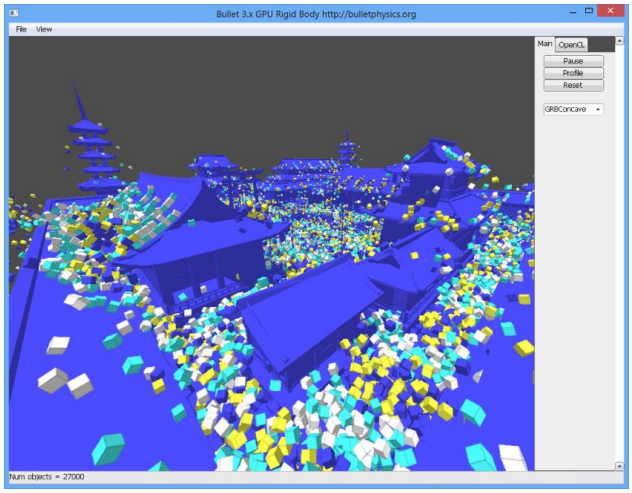


Figure 6.1: The Bullet rigid body simulator in action, simulating hundreds of thousands of bodies and their collisions, all in real-time. Picture credit: Erwin Coumans [19].

In the remainder of this chapter, we describe in detail how KernelInterceptor is used (Section 6.1) and how it is implemented (Section 6.2). Section 6.3 evaluates KernelInterceptor's limitations and the extent to which it improves the usability of GPUVerify, and also discusses related and future work.

6.1 Usage

This section explains how KernelInterceptor works from the user's perspective. As a running example, we use an OpenCL application that simulates collisions of rigid bodies [19]. This application is part of the open source Bullet Physics library (version 3)[4] and the code is available online.[5] The capabilities of the simulator are demonstrated in Fig. 6.1.

6.1.1 Instrumenting the source code

To use KernelInterceptor, the user must first download GPUVerify, with which the KernelInterceptor header file (`cl_interceptor.h`) and library (`cl_interceptor.cpp`) are shipped.

The line

```
#include "/path/to/cl_interceptor.h"
```

must be added to each `.cpp` file that includes the OpenCL headers (`cl.h` or `opencl.h`). In the case of the Bullet simulator, the only relevant file is `b3OpenCLInclude.h`.

The user must modify their build process so that it compiles `cl_interceptor.cpp` and links it against their application. In the case of the Bullet simulator, it suffices to add `cl_interceptor.o` as a build target in the relevant makefiles.

The application can now be built and run as normal. The interception process records entire kernel texts and writes them to disk on every kernel invocation, which may incur nontrivial

[4] `http://bulletphysics.org`
[5] `https://github.com/erwincoumans/bullet3`

```
[0] Name: AddOffsetKernel
    File: .gpuverify/AddOffsetKernel001.cl
    local_size=128 global_size=12544
    args=0x7f4b000000800000006300006271
    Built at b3OpenCLUtils.cpp:880
    Run at b3LauncherCL.h:117

[1] Name: AddOffsetKernel
    File: .gpuverify/AddOffsetKernel002.cl
    local_size=128 global_size=12544
    args=0x7f4b000000800000006300006271
    Built at b3OpenCLUtils.cpp:880
    Run at b3LauncherCL.h:117

[2] Name: AddOffsetKernel
    File: .gpuverify/AddOffsetKernel003.cl
    local_size=128 global_size=896
    args=0x7f4b000000800000000800000780
    Built at b3OpenCLUtils.cpp:880
    Run at b3LauncherCL.h:117
...
```

Figure 6.2: Abridged output obtained from the command `gpuverify --show-intercepted`

runtime overhead. We therefore recommend enabling KernelInterceptor only as part of a debug build.

6.1.2 Inspecting the intercepted kernels

The user can view information about the intercepted kernels using the command

`gpuverify --show-intercepted`.

After running KernelInterceptor on the Bullet simulator, this command produces the output shown in Fig. 6.2.

Each kernel instance is identified by a number, which is given in brackets. For each instance, the command reports:

- the name of the kernel;
- the file that contains the kernel's source code;
- the work group size (`local_size`) and the total number of work items (`global_size`);
- the hexadecimal values of the kernel's scalar arguments (see remark below);
- the position in the application's source code where this kernel was compiled; and
- the position in the application's source code where this kernel was invoked.

We remark that KernelInterceptor does not record non-scalar arguments (i.e., array or pointer arguments), since they tend not to affect the correctness of the kernel. Indeed, GPUVerify ignores

```
GPUVerify kernel analyser checked 37 kernels.
Successfully verified 35 kernels.
Failed to verify 2 kernels.

Successes:
[0]  Verification of AddOffsetKernel
     (.gpuverify/AddOffsetKernel001.cl) succeeded with:
     local_size=128 global_size=12544 args=3
...
Failures:
[13] Verification of scatterKernel
     (.gpuverify/scatterKernel003.cl) failed with:
     local_size=12 global_size=256 args=14,8
[27] Verification of SubtractKernel
     (.gpuverify/SubtractKernel020.cl) failed with:
     local_size=12 global_size=24 args=7
Run ‘gpuverify --check-intercepted=<number>’ for
more details.
```

Figure 6.3: Abridged output obtained from the command `gpuverify --check-all-intercepted`

the values of such arguments as part of its abstraction. Scalar values are stored in hexadecimal format because GPUVerify deals only with untyped bitvectors.

Reporting where each kernel instance was compiled and where it was invoked is valuable to users because tracing the origin of a kernel obtained by KernelInterceptor can be tricky: the kernel's source code may not be simply read from a file, but pieced together from multiple files and string constants at runtime, and possibly configured based on user input.

6.1.3 Verifying the intercepted kernels

Having inspected the intercepted kernels, the user can now ask GPUVerify to check their correctness.

The command

```
gpuverify --check-all-intercepted
```

instructs GPUVerify to attempt to verify all of the kernel instances. In an effort to maintain readability when there are many kernel instances, the output from GPUVerify is abbreviated, so as to identify only those kernels that failed to verify. These kernels can then be examined and re-verified individually. An illustrative output is shown in Fig. 6.3.

The command

```
gpuverify --check-intercepted=2
```

instructs GPUVerify to attempt to verify the kernel instance identified as number 2. In this case, GPUVerify outputs a message that it has verified the kernel, which implies that there are no data races and no instances of barrier divergence. Had GPUVerify detected the potential for any of these defects, it would have directed the user to the relevant line(s) in the `AddOffsetKernel003.cl` file. The third possible result from running GPUVerify is a timeout, which occurs when GPUVerify is unable to prove or to disprove the kernel's correctness.

```
1   // --local_size=128 --global_size=896
        --kernel-args=AddOffsetKernel,0x00007f4b000000080000000800000780
2   // Built at ../../src/Bullet3OpenCL/Initialize/b3OpenCLUtils.cpp:880
3   // Run at ../../src/Bullet3OpenCL/ParallelPrimitives/b3LauncherCL.h:117
    ...
94  __kernel
95  void AddOffsetKernel(__global u32 *dst, __global u32 *blockSum, uint4 cb)
96  {
       ...
106 }
```

Figure 6.4: Data logged in `AddOffsetKernel003.cl` for the third instance of the `AddOffsetKernel` kernel

6.2 Implementation

We now discuss some of the technical details of the implementation of KernelInterceptor. We continue to use the Bullet simulator as a running example.

6.2.1 Intercepting kernel launches

Relevant OpenCL host functions, such as `clCreateBuffer`, `clCreateProgramWithSource` and `clSetKernelArg`, are intercepted at the source level, such that, for example, a call to `clSetKernelArg` in the host code actually calls our wrapper function, `clSetKernelArg_hook`. The wrapper functions log the relevant information and then pass the parameters to the original functions, as normal.

6.2.2 Logging kernel parameters

Each time a kernel is invoked, KernelInterceptor creates a file, whose name is formed from the name of the kernel, followed by a unique identifier to avoid name clashes. These files are stored in a `.gpuverify` directory, which KernelInterceptor creates in either the application's main directory, or in a directory specified by the environment variable `GPUV_KI_DIR`. In the case of the Bullet simulator, when executed for a few seconds on several of the standard demonstrations, over a thousand such files were created, corresponding to the invocations of 44 different kernels.

Let us now consider one of these files, `AddOffsetKernel003.cl`, which is created when KernelInterceptor intercepts the third launch of the kernel called `AddOffsetKernel`. Its contents is shown in Fig. 6.4. The file contains the kernel's source code, preceded by three commented lines. The first of these records the work group size and total number of work items, plus the hexadecimal value of `AddOffsetKernel`'s sole scalar argument (which is named `cb`). The second and third lines record the positions in the source code where the kernel was compiled and invoked, respectively.

6.2.3 Passing kernel arguments to GPUVerify

We have extended GPUVerify to accept a `--kernel-args` flag through which values for the arguments of a given kernel can be provided.

If K is the name of a kernel, and K's scalar arguments are $x_1, \ldots, x_n$, then

`--kernel-args=`K`,`v_1`,...,`v_n

instructs GPUVerify to assume the precondition

`__requires(`x_i`==`v_i`)`

for each $0 < i \leq n$, when verifying the kernel K. The order of the values provided to `--kernel-args` matches the order in which K's scalar arguments are declared.

An argument can be left unconstrained by inserting an asterisk. For instance, if K accepts three scalar arguments, `a`, `b` and `c`, then the flag

```
--kernel-args=binning_kernel,*,0x42,*
```

will insert the single precondition

```
__requires(b==0x42).
```

It is allowable to pass several `--kernel-args` flags to GPUVerify, each providing arguments for a different kernel in the same `.cl` file. By default, GPUVerify seeks to verify all the kernels in a given file, but we arrange that when one or more `--kernel-args` flags are provided, GPUVerify only checks those kernels that are named in those flags. A `.cl` file may contain a large number of kernels, only some of which are used by an application; our arrangement ensures that GPUVerify seeks to verify only those kernels that are actually invoked.

6.2.4 Caching verification results

When multiple kernel instances share the same source code, launch parameters and kernel arguments, the results of attempting to verify them will be the same. To avoid redundant calls to GPUVerify, we arrange that the results of successful verification attempts are written to a cache file, whose path is specified using the command-line flag

`--cache=<path>`.

The cache file is consulted before each verification attempt, and if there is a match, the cached result is displayed. Failed verification attempts are not cached, since such attempts might become successful when a more capable version of GPUVerify becomes available.

6.3 Discussion

In this section, we comment on the usability of our tool, discuss related work, consider some limitations of our tool, and suggest some future lines of enquiry.

6.3.1 Usability of KernelInterceptor

The GPUVerify team used KernelInterceptor to assist with the verification of the Parboil benchmark suite [50]. This suite consists of 12 programs and 25 unique kernels, some programmatically generated.

KernelInterceptor accelerated the process of extracting kernel source, compiler options, and valid local and global sizes. We observe that some kernels, such as those in the `stencil`

benchmark, are only race free when given certain arguments; this would have been difficult to infer without the data provided by KernelInterceptor.

Using KernelInterceptor required adding just a handful of lines to the benchmark source and makefiles. It removed a significant amount of labour in the preparation of a recent conference paper [4].

6.3.2 Limitations

Discovery of invariants Although this work increases the degree of automation in GPU kernel verification, we should point out that *completely automatic* verification requires significant further research, due to the problem of discovering invariants for verifying barrier statements [15] and loop statements. Many kernels cannot be verified without these invariants, and although much progress has been made in using heuristics to infer these automatically, the task of supplying them often falls back to the user.

Dependence on particular kernel parameters Note that because the parameters are extracted from a *particular execution* of the OpenCL application, we cannot claim every kernel to be 'fully verified': the kernel may not be correct when launched with different parameters. What we *can* claim is that with these parameters, the kernel is correct under any execution schedule.

6.3.3 Future directions

Generalising parameters As noted above, a successfully verified kernel is only guaranteed to be defect-free when launched with specific parameters. In future work, we plan to investigate how to generalise these parameters, in order to strengthen the verification result.

Consider `SubtractKernel`, one of the kernels from the Bullet simulator. Starting from a successful verification with parameters

```
--local_size=64 --global_size=256
--kernel-args=SubtractKernel,0x000065f4,0x00000100
```

one could greedily unconstrain values, by setting them to "`*`", until a minimal set of constraints is obtained. We find that the correctness of this particular kernel does not depend on the kernel arguments, so the constraints

```
--local_size=64 --global_size=256 --kernel-args=SubtractKernel,*,*
```

are sufficient.

When there are many kernel instances to check, this parameter generalisation technique may lead to fewer calls to GPUVerify being required. For instance, all instances of `SubtractKernel` where `local_size` is `64` and `global_size` is `256` can now be considered verified, regardless of the other parameters, since the stronger result has already been proven.

We also plan to investigate other ways to unconstrain kernel parameters. Constraints such as 'this parameter must be a power of 2' or 'that parameter must not exceed 1024' could reasonably be conjectured by a tool such as Daikon [21], and then checked.

Run-time instrumentation We are considering implementing an alternative mechanism that operates solely at run-time. This would be even less intrusive to the user than the current mechanism, because no recompilation would be necessary. However, it would require additional work on our part to ensure compatibility with all platforms and drivers.

In the case of a Linux environment, we would make use of the `LD_PRELOAD` environment variable. This identifies a directory of libraries that should, at run-time, be linked before any other. By pointing this variable to our library of wrappers for the relevant OpenCL host functions, we can attain run-time interception.

Support for other kernel programming languages We plan to extend our kernel interception technique to support kernels that have been pre-compiled to the SPIR[6] intermediate representation. GPUVerify has direct support for the LLVM[7] intermediate representation [18], of which SPIR is a dialect, so this should prove quite straightforward. We plan also to support kernels written in CUDA, but we note that the run-time linking trick described above would not work in a CUDA setting, where host programs are typically linked statically.

Static analysis We plan to investigate the use of static analysis on the host program as an alternative way to discover kernel parameters. This would mean that the OpenCL application would not need to be executed at all; our tool would simply examine the application's source code. An advantage of an approach based on static analysis is that the correctness of the kernel can be guaranteed for *all possible* executions of the application, rather than just a particular execution. A disadvantage, however, is that the kernel verification is more likely to fail. It may, for instance, be understood that the application is only to be provided with positive inputs, but unless this requirement is codified as an explicit precondition in the source code, the static analysis will be ignorant of this and report that the kernel is incorrect in general.

6.3.4 Related work

There has been significant interest recently in methods for analysing and verifying GPU kernels.

Li and Gopalakrishnan's PUG analyser shares the problem of requiring the user to supply kernel arguments and the number of work items manually [40]. Our technique for addressing this problem only applies to OpenCL kernels, and hence is not directly applicable to PUG, which analyses only CUDA kernels.

The GKLEE [41] and KLEE-CL [17] tools, which are based on dynamic symbolic execution, do not have this problem because they execute symbolically both host and device code. However, although these tools seek to *discover* data races, they do not attempt to verify their *absence* as GPUVerify does.

The technique of Leung et al. [39] for verifying race-freedom of CUDA kernels is based on dynamic analysis and thus already exploits information about thread configurations and kernel arguments.

In Deliverable D6.1, and discussed further in Chapter 4, we presented CARP research on a technique to allow functional verification of GPU kernels without the need to fix the number of work items [32]. We observe that many kernels require some constraints on the number of work items (such as 'must be a power of 2' or 'must not exceed 1024') in order to be correct. The KernelInterceptor concept could therefore prove useful in this setting.

[6] `http://www.khronos.org/spir`
[7] `http://llvm.org/`

7 Automated Termination Analysis for GPU Kernels

As GPUs are separate devices to which kernels are offloaded, it is generally difficult to perform live debugging. Hence, different means are needed to identify bugs. In previous work—Deliverable D6.2 and [7, 18]—we have looked at uncovering data races. Here we consider termination.

Unlike CPU applications, which may be reactive, GPU kernels are *required* to terminate: any data computed by a kernel is inaccessible from the CPU as long as the kernel has not terminated. Besides this practical consideration, termination is also important from a theoretical perspective: the data race detection method described in [7], which underpins our data race detection tool GPUVerify, is only sound for terminating kernels.

We describe and evaluate a method for proving termination of kernels. Termination analysis is complicated by concurrency, but there is no need to reason about recursive calls or dynamically changing data structures since recursion and dynamic memory allocation are generally not supported by kernel programming languages.

The contributions of this chapter are two-fold:

1. We leverage termination analysis for sequential programs to obtain an analysis technique for GPU kernels; the technique abstracts from all but one arbitrary thread.

2. We adapt an existing termination analysis tool—`KITTeL` [22, 23]—and show that it can be successfully applied to a large set of real-world kernels.

7.1 Reduction to Sequential Termination Analysis

We present the kernel programming language from [16], which has the following grammar:

$$\begin{array}{lll} \text{expr} & e & ::= c \mid v \mid A[e] \mid e_1 \mathbin{op} e_2 \\ \text{stmt} & s & ::= v := e \mid A[e_1] := e_2 \mid \text{if}\,(e)\,\{ss_1\}\,\text{else}\,\{ss_2\} \mid \text{while}\,(e)\,\{ss\} \mid \text{barrier} \\ \text{stmts} & ss & ::= \varepsilon \mid s; ss \end{array}$$

Here, c and v represent constants and *thread-private* variables; A and op represent *shared* arrays and arbitrary binary operators. As explained in the introduction, the barrier statement allows for synchronisation between threads; ε represents the empty sequence of statements. A *kernel program* P is a sequence of statements ss; all threads execute the same sequence ss. For technical reasons we assume that an expression e has at most one sub-expression of the form $A[e']$, so that evaluating an expression involves reading at most once from the shared state; a kernel can be trivially preprocessed to satisfy this restriction.

A *thread state* is a pair (σ_v, ss), where σ_v represents the private memory of a thread—mapping private variables to values—and where ss is the sequence of the statements the thread needs to execute. A *kernel state* is a pair (σ_A, K), where σ_A represents the shared memory of the kernel—mapping shared arrays to sequences of values—and where K is a map from a *finite* set of thread identifiers t to thread states. The *initial kernel state* of a kernel program P is any state such that the second component of each $K(t)$ is P.

Figure 7.1 gives the operational semantics of the language. For brevity, we omit the rules for the assignments and if-statement, which are straightforward, and refer the reader to [16]. The

$$\frac{[\![e]\!]^{\sigma_v}_{\sigma_A}}{(\sigma_v,\sigma_A,\text{while}\,(e)\,\{ss\};ss') \to_s (\sigma_v,\sigma_A,ss\cdot\text{while}\,(e)\,\{ss\};ss')}\ (\textsc{Loop-T}) \qquad \frac{\neg[\![e]\!]^{\sigma_v}_{\sigma_A}}{(\sigma_v,\sigma_A,\text{while}\,(e)\,\{ss\};ss') \to_s (\sigma_v,\sigma_A,ss')}\ (\textsc{Loop-F})$$

(a) The thread-level rules (operating over thread states (σ_v, ss) and shared memory σ_A)

$$\frac{K(t)=(\sigma_v,ss) \qquad (\sigma_v,\sigma_A,ss)\to_s(\sigma'_v,\sigma'_A,ss') \qquad K'=K[t\mapsto(\sigma'_v,ss')]}{(\sigma_A,K)\to_k(\sigma'_A,K')}\ (\textsc{Step})$$

$$\frac{\left(\forall t:\exists\sigma_v:\bigvee\begin{array}{c}(\exists ss:K(t)=(\sigma_v,\text{barrier};ss)\wedge K'(t)=(\sigma_v,ss))\\ (K(t)=(\sigma_v,\varepsilon)\qquad\wedge K'(t)=(\sigma_v,\varepsilon))\end{array}\right)\qquad \exists t,\sigma_v,ss:K(t)=(\sigma_v,\text{barrier};ss)}{(\sigma_A,K)\to_k(\sigma_A,K')}\ (\textsc{Barrier})$$

(b) The Kernel-level rules (operating over kernel states (σ_A, K))

Figure 7.1: Operational semantics of our kernel programming language

rules for the while-statement evaluate the guard e under σ_v and σ_A, denoted $[\![e]\!]^{\sigma_v}_{\sigma_A}$, and proceed accordingly. As can be seen from rule STEP, the language has an interleaving semantics. Rule BARRIER is used for synchronisation between threads: no thread can proceed beyond a barrier unless all threads have either reached a barrier or have terminated. The rule requires that at least one thread is actually at a barrier; this ensures that the rule no longer fires once all threads have terminated (i.e., once all have reached a state (σ_v, ε)).

We next reduce the termination problem for kernel programs to a sequential termination problem. The reduction makes termination analysis for kernel programs thread-modular by checking termination of a single, arbitrary thread under an environmental abstraction that over-approximates the effects of the other threads.

To obtain the abstraction, existentially quantify the premise of each thread-level rule over all array stores σ_A and replace rule BARRIER by the thread-level rule $(\sigma_v,\sigma_A,\text{barrier};ss)\to(\sigma_v,\sigma_A,ss)$. Denote by $\to_{s,\star}$ the over-approximating thread-level reduction relation such obtained. The relation ensures that the contents of σ_A is irrelevant and that a thread no longer has to wait for any other thread once it reaches a barrier. We have the following.

Theorem 7.1.1. *Let P be a kernel program. If for each σ_v and σ_A it holds that all reductions $(\sigma_v,\sigma_A,P)\to_{s,\star}\cdots\to_{s,\star}\cdots$ are finite, then P terminates under the semantics of Figure 7.1.*

Proof. Suppose the contrary, then there exists an infinite reduction ρ of P. As the number of threads is finite, there is a thread t that is selected an infinite number of times by rule STEP. We construct an infinite reduction for t under $\to_{s,\star}$: (i) for each application of STEP selecting t apply the over-approximating version of the thread-level rule employed and (ii) for each application of rule BARRIER employ the over-approximating barrier rule. The over-approximating rules fire, as (i) the existential quantification over all array stores σ_A ensures that e can be evaluated precisely as in ρ and as (ii) the thread-level barrier rule essentially skips a barrier. Hence, we have an infinite reduction for t under $\to_{s,\star}$, contradiction. □

A theorem related to the one above underpins the soundness of GPUVerify, where shared state abstraction allows race-freedom to be verified by considering just *two* arbitrary threads [7]. Observe that the theorem only modifies the operational semantics; kernel programs are left unchanged. Furthermore, the reverse of the theorem does not hold: termination might depend on shared memory sub-expressions evaluating to specific values.

7.2 Experimental Evaluation

To evaluate the effectiveness of Theorem 7.1.1, we adapted the `KITTeL` termination analysis tool [22, 23] and applied it to a suite of 598 kernels, 381 of which have loops. To demonstrate that our approach works out-of-the-box, we included the loop-free kernels in our evaluation. The kernels have on average 86 lines of code and originate from nine sources:

- *AMD Accelerated Parallel Processing SDK* v2.6 [1] (78 kernels, 54 of which have loops).
- *NVIDIA GPU Computing SDK* v5.0 [43] (183 kernels, 109 of which have loops); we also include 8 kernels from v2.0 of the SDK, 7 of which have loops.
- *C++ AMP Sample Projects* [42] (20 kernels, 16 of which have loops)
- The *gpgpu-sim* benchmarks [3] (33 kernels, 24 of which have loops)
- The *Parboil* benchmarks v2.5 [50] (25 kernels, 19 of which have loops)
- The *Rodinia* benchmark suite v2.4 [14] (36 kernels, 24 of which have loops)
- The *SHOC* benchmark suite [20] (87 kernels, 53 of which have loops)
- The *PolyBench/GPU* benchmark suite [27] (64 kernels, 49 of which have loops)
- Rightware *Basemark CL* v1.1 [49] (64 kernels, 26 of which have loops)

Each suite is publicly available except for *Basemark CL* which was provided to us under an academic license. The collection covers all the publicly available GPU benchmark suites we are aware of. The kernel counts above do not include 5 kernels that we manually removed because they use CUDA surfaces or thread fences [44], which we currently do not support.

KITTeL The `KITTeL` termination analysis tool [22, 23] consists of a front-end, `llvm2KITTeL`, which takes llvm bitcode[1] and translates this into an integer-based rewrite system. The back-end automatically tries to prove termination of the generated rewrite system.

We adapted `llvm2KITTeL` to handle kernels (compiled to bitcode by Clang[2]); we did not make any changes to the termination analysis back-end. As `llvm2KITTeL` models only a single thread and already abstracts from most memory operations (yielding nondeterministic values for loads from memory), the changes we needed to make were minimal. To summarise: (i) we ensured that `llvm2KITTeL` abstracts loads even in cases where it usually does not (e.g., when a pointer points to a unique global variable representing a single integer), (ii) we disabled the hoisting of loop-invariant loads from loops (due to concurrency the loaded value might differ between loop iterations), and (iii) we made `llvm2KITTeL` aware of the fact that the number of threads executing a kernel is constant for the duration of an execution (the number of threads is often referred to in loop guards; hence, awareness that this number is constant—or at least bounded—is often critical for showing termination).

[1] `http://llvm.org/`
[2] `http://clang.llvm.org/`

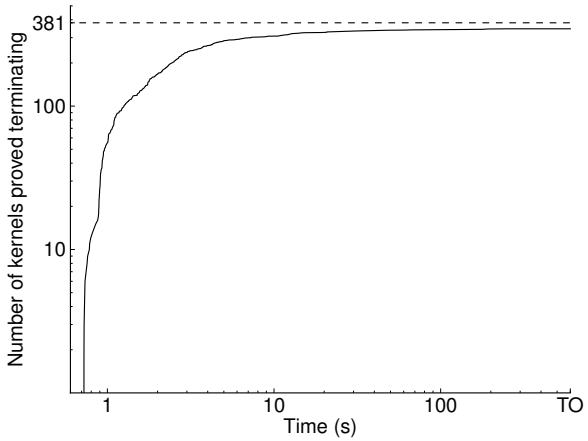


Figure 7.2: Cumulative histogram showing the time taken to prove termination of the kernels with loops

Loop Invariants Currently, `KITTeL` does not infer loop invariants that may be needed for proving termination. We provided these invariants by hand and proved them correct with GPUVerify prior to running our experiments. In principle we could extend GPUVerify's invariant inference engine [7] to generate the needed invariants; this would require infrastructure to feed the generated invariants into `KITTeL`.

We required loop invariants stating: (i) the loop counter must be positive (31 kernels), (ii) the step value for the loop counter is positive (18 kernels), (iii) the loop counter is always smaller than or equal to a value which is subtracted from it in the loop guard (2 kernels).

Experimental Setup All experiments were conducted on a Mid 2009 MacBook Pro with a 2.53GHz Intel Core 2 Duo and 4GB RAM running OS X 10.9.2 and Clang/llvm 3.4. The reported times are averages over three runs and include the time needed to compile a kernel into bitcode. The timeout used was 10 minutes. We adapted `llvm2KITTeL` as described above and always invoked the tool with its `-increase-strength` option—turning left and right shifts into multiplications and divisions, respectively. The latter facilitates termination analysis of kernels where the loop counter is being shifted. The SMT solver used with `KITTeL` was Z3 v4.3.1. Both `llvm2KITTeL` and `KITTeL` were downloaded on 21-04-2014.[3]

Results Unsurprisingly, `KITTeL` managed to prove termination of all 217 loop-free kernels. On average termination of these kernels was shown in 0.63s and the maximum time required was 6.15s. Six of the kernels needed over 1s; this was either due to a long compilation time or the kernel consisting of a large number of subroutines.

Of the 381 kernels with loops, 346 could be shown terminating. On average termination was shown in 7.23s and the maximum time needed was 254.17s (see also Figure 7.2). Of the 35 kernels for which termination could not be shown, 31 reached the timeout of 10 minutes. In 4 cases `KITTeL` indicated that the constructed rewrite system was nonterminating (this does not imply that the original kernels are nonterminating, as the constructed rewrite system in general over-approximates the behaviour of a thread).

[3] `https://github.com/s-falke/{llvm2kittel,kittel-koat}`

We manually inspected the 35 kernels to see why termination could not be shown. All 4 cases where KITTeL indicated nontermination would require reasoning over floating point numbers. In 4 other cases built-in atomic increment operations would need to be modelled as returning monotonically increasing values—instead of arbitrary ones, as is currently the case. In 19 cases termination would require reasoning about shared memory and, hence, the over-approximation from Theorem 7.1.1 is too coarse.

The above leaves 8 kernels, all of which timed out. Of these, 2 could be shown terminating using a very coarse over-approximation of division—yielding unconstrained nondeterministic values. One kernel could be shown terminating with llvm2KITTeL's `-only-loop-conditions` option, which abstracts all basic blocks except those from which loops can be exited.

In the case of 2 kernels the function bodies were very large which resulted in a timeout in llvm2KITTeL (these were the only timeouts in llvm2KITTeL). In the 3 remaining cases a timeout occurred in KITTeL, although the generated rewrite system was terminating.

7.3 Conclusion

We have described an approach for proving termination of massively parallel GPU kernels by reducing the termination problem for these kernels to a sequential termination problem. With the help of an adapted version of KITTeL the reduction allowed us to prove termination of 94% of the kernels in our benchmark set and of 91% of the kernels with loops.

As part of future work we would like to automatically infer loop invariants that are required for proving termination. Moreover, we would like to investigate whether performance can be improved by outlining—as opposed to inlining—loops into separate procedures.

Bibliography

[1] AMD. AMD Accelerated Parallel Processing (APP) SDK. `http://developer.amd.com/sdks/amdappsdk`.

[2] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a platform-neutral compute intermediate language for DSLs. *CoRR*, abs/1302.5586, 2013. Presented at the WOLFHPC 2012 workshop.

[3] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174, 2009.

[4] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, and S. Qadeer. Engineering a static verification tool for GPU kernels. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV '14)*, 2014. To appear.

[5] E. Bardsley, A. F. Donaldson, and J. Wickerson. KernelInterceptor: automating GPU kernel verification by intercepting kernels and their parameters. In *IWOCL*. ACM, 2014.

[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 F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[7] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*, pages 113–132, New York, NY, USA, 2012. ACM.

[8] S. Blom, S. Darabi, and M. Huisman. Verifying parallel loops with separation logic. In A. F. Donaldson and V. T. Vasconcelos, editors, *PLACES*, volume 155 of *EPTCS*, pages 47–53, 2014.

[9] S. Blom and M. Huisman. The vercors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM*, volume 8442 of *Lecture Notes in Computer Science*, pages 127–131. Springer, 2014.

[10] S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of {GPGPU} programs. *Science of Computer Programming*, 2014. available online.

[11] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[12] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[13] Chair of Programming Methodology, ETH Zürich. Viper project website. `http://www.pm.inf.ethz.ch/research/viper`.

[14] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization*, pages 44–54, 2009.

[15] N. Chong, A. F. Donaldson, P. H. Kelly, J. Ketema, and S. Qadeer. Barrier invariants: A shared state abstraction for the analysis of data-dependent GPU kernels. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*, pages 605–622, New York, NY, USA, 2013. ACM.

[16] N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.

[17] P. Collingbourne, C. Cadar, and P. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering*, 2014. To appear.

[18] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, volume 7792 of *Lecture Notes in Computer Science*, pages 270–289, Berlin, 2013. Springer.

[19] E. Coumans. GPU rigid body simulation using OpenCL. In *Multithreading and VFX*, a tutorial at *SIGGRAPH 2013*, 2013. `http://www.multithreadingandvfx.org/course_notes/GPU_rigidbody_using_OpenCL.pdf`.

[20] A. Danalis et al. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, pages 63–74, 2010.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[22] S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA*, pages 41–50, 2011.

[23] S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.

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

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

[26] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[27] S. Grauer-Gray et al. Auto-tuning a high-level language targeted to GPU codes. In *InPar*, 2012.

[28] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *APLAS*, volume 5356 of *LNCS*, pages 171–187, 2008.

[29] E. Hehner. Specified blocks. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.

[30] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In G. Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 451–476. Springer, 2013.

[31] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[32] M. Huisman and M. Mihelčić. Specification and verification of GPGPU programs using permission-based separation logic. In *The 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '13)*, 2013.

[33] B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VSTTE workshop on Tools & Experiments*, August 2010.

[34] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[35] I. T. Kassios, P. Müller, and M. Schwerhoff. Comparing verification condition generation with symbolic execution: An experience report. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2012.

[36] J. Ketema and A. F. Donaldson. Automatic termination analysis for GPU kernels. In *WST*, 2014.

[37] Khronos OpenCL Working Group. The OpenCL specification, version 1.2, 2012.

[38] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[39] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 383–394, New York, 2012. ACM.

[40] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pages 187–196, New York, 2010. ACM.

[41] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '12)*, pages 215–224, New York, 2012. ACM.

[42] Microsoft Corporation. C++ AMP sample projects for download (MSDN blog). http://goo.gl/eb8ob.

[43] NVIDIA. GPU Computing SDK. https://developer.nvidia.com/gpu-computing-sdk.

[44] NVIDIA. CUDA C programming guide, version 5.0, 2012.

[45] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

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

[47] M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

[48] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[49] Rightware Oy. Basemark CL. http://www.rightware.com/benchmarking-software/basemark-cl/.

[50] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[51] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM TACO*, 9(4):54, 2013.

[52] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 2011.