Program Correctness by Transformation

Marieke Huisman[1](✉), Stefan Blom[2], Saeed Darabi[3], and Mohsen Safari[1]

[1] University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl
[2] BetterBe, Enschede, The Netherlands
[3] ASML, Veldhoven, The Netherlands

Abstract. Deductive program verification can be used effectively to verify high-level programs, but can be challenging for low-level, high-performance code. In this paper, we argue that compilation and program transformations should be made *annotation-aware*, i.e. during compilation and program transformation, not only the code should be changed, but also the corresponding annotations. As a result, if the original high-level program could be verified, also the resulting low-level program can be verified. We illustrate this approach on a concrete case, where loop annotations that capture possible loop parallelisations are translated into specifications of an OpenCL kernel that corresponds to the parallel loop. We also sketch how several commonly used OpenCL kernel transformations can be adapted to also transform the corresponding program annotations. Finally, we conclude the paper with a list of research challenges that need to be addressed to further develop this approach.

1 Introduction

Over the last decade, substantial progress has been made in the development of techniques for deductive program verification. By now, it is possible to verify non-trivial programs effectively, as illustrated for example by the attempt to verify the TimSort algorithm [12], as well as by the advent of logics to reason about concurrent software, see e.g. [9,14,15,18]. However, there still is a gap to bridge between the programs that can be verified effectively, and the actual code that is running on systems – even when the deductive verification tool at hand actually supports reasoning about a real programming language.

For deductive program verification to be successful, one needs to understand the algorithm, and the invariants that are preserved by the algorithm. As a consequence, verification is typically most suited when the verified program is written at a relatively high-level, using high-level data structures to abstract the program's state space. However, when code is implemented, for performance reasons often low-level implementation choices are made, which might obfuscate the high-level algorithm, and can make verification of the actual implemented code much more challenging than the verification of the high-level algorithm.

Therefore, in this paper we argue that verification should be done step-wise, and that we should extend compilers and program transformations in such a way

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11244, pp. 365–380, 2018.
https://doi.org/10.1007/978-3-030-03418-4_22

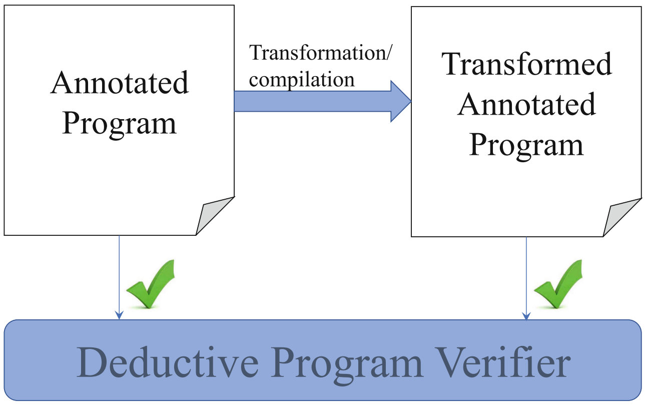
Annotated Program
Transformation/ compilation
Transformed Annotated Program
Deductive Program Verifier

Fig. 1. Annotation-aware program transformation and compilation

that they can preserve *provability of correctness*. Thus, we do not aim at proving that the compilation process or a program transformation preserves correctness, but instead we argue that the compilation or transformation process should be extended to include the extra information that is needed to make sure that the resulting program can be proven correct.

In the particular case of this paper, we look at deductive program verification. As deductive program verification techniques require code to be annotated with intermediate properties, we argue that compilation and program transformation should become *annotation-aware*, i.e. they should not only transform the program, but also the corresponding program annotations, as illustrated by Fig. 1. If this annotation transformation is done properly, then the low-level, high-performance code can be verified (provided the original high-level program was verified). Ideally, this process is done fully automatically, i.e, the high-level program is annotated and verified, and subsequently the developer applies a collection of different program transformations, searching for optimal performance, while the correctness is preserved fully automatically.

We believe that such an approach to correctness preservation by compilation and program optimisation is a necessary step to make verification of high-performance low-level code feasible and scalable. We discuss first steps in this direction, as well as the research challenges that need to be addressed to realise this scenario. In particular, we show how the verification of compiler directives for the parallelisation of loops can be used to compile the loop into an OpenCL kernel (suitable for GPU architectures), such that the loop specification is compiled into a kernel specification (based on ideas we presented in [4]). We also discuss various program transformations that can be used to improve the performance of OpenCL kernels, and sketch how the annotations of a verified kernel should be adapted to ensure that the result of the program transformation can be verified again. In this paper, we focus on transformations that are suitable to OpenCL kernels, but similar techniques also should be applicable to transformations that improve performance of a program running on a CPU.

The ideas presented in this paper are currently partly supported by our VerCors tool set, which supports the verification of concurrent software for multiple input languages, such as Java, C with OpenMP annotations, OpenCL kernels, and our own prototype language PVL [5]. Support for OpenCL at the moment is still lacking some features, in particular to reason about barriers. However, this is only an implementation issue: for the PVL language we can reason about parallel blocks, using barriers for synchronisation. Therefore, in this paper, where suitable we give annotated OpenCL examples as should be verifiable in the near future (and where we currently can verify the corresponding PVL version).

The remainder of this paper is organised as follows. Section 2 provides the necessary background on permission-based separation logic. We discuss in particular how we use this to verify loop parallelisations, and OpenCL kernels. Next, Sect. 3 discusses how loops and their specifications are compiled into OpenCL kernels with corresponding specifications. Section 4 then sketches how similar ideas can be used when transforming the OpenCL kernels to improve their performance, while Sect. 5 concludes with the open research challenges that we believe need to be addressed to fully realise this scenario.

2 Background

Before discussing how program specifications are preserved by compilation and transformation, we first give a brief introduction to our programming specification language, and discuss the basics behind our verification approach. In particular, we discuss loop iteration contracts, which is a technique that we use to reason about high-level programs, and we discuss the main ingredients of our verification technique for OpenCL programs.

2.1 Program Specification Language

Our program specification language is based on permission-based separation logic [1,7], combined with the look-and-feel of the Java Modeling Language (JML) [16]. In this way we exploit the expressiveness and readability of JML, while using the power of separation logic to support thread-modular reasoning. We briefly explain the syntax of formulas and how it extends the standard JML program annotation syntax, where JML annotations are expressions in first-order logic. For the precise semantics of our formulas, we refer to [8,10].

Every thread holds permissions to access memory locations. These permissions are encoded as fractional values (cf. Boyland [8]): any fraction in the interval $(0, 1)$ denotes a *read permission*, while 1 denotes a *write permission*. Permissions can be split (by subtraction) and combined (by addition), and soundness of the program logic ensures that for every memory location the total sum of permissions over all threads to access this location does not exceed 1. This guaranty is sufficient to ensure data race freedom of any verified program: if a thread holds a write permission to a location, no other thread will have access to this location; if a thread holds a read permission to a location, any other thread also can have at most a read permission. The set of permissions that a thread holds is typically called its *resources*.

Formulas F in our program specification language extend first-order logic formulas with the following expressions: permission predicates $\mathsf{Perm}(e_1, e_2)$, conditional expressions $(\circ?\circ : \circ)$, separating conjunction $\star$, and universal separating conjunction $\bigstar$ over a finite set I. The syntax of formulas is formally defined as follows:

$$\begin{aligned}
&F ::= b \mid \mathsf{Perm}(e_1, e_2) \mid b?F : F \mid F \star F \mid \bigstar_{i \in I} F(i)\\
&b ::= \textbf{true} \mid \textbf{false} \mid e_1 == e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2 \mid \ldots\\
&e ::= [e] \mid v \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots
\end{aligned}$$

where b is a side-effect free boolean expression, e is a side-effect free arithmetic expression, $[\circ]$ is a unary dereferencing operator – thus $[e]$ returns the value stored at the address e in shared memory – v ranges over variables and n ranges over numerals. Wellformedness requires that the first argument of the $\mathsf{Perm}(e_1, e_2)$ predicate is always an address, while the second argument is a fraction. We use the array notation $a[e]$ as syntactic sugar for $[a + e]$ where a is a variable containing the base address of the array a and e is the subscript expression; together they point to the address $a + e$ in shared memory.

For the semantics of the formulas we refer to [10], but as an example we define the semantic of universal separating conjunction as $\bigstar_{i \in I} F(i) \equiv F(p) + F(p+1) + \cdots + F(q)$ (if $I = \{p, \cdots, q\}$).

2.2 Iteration Contract

To reason about loop parallelisations, and to show that the parallelisation of a loop does not change its behaviour, we introduced the notion of *iteration contracts* [4,11]: an iteration contract specifies a contract that should hold for every iteration of the loop. This iteration contract should specify at least which variables are read and written by one iteration of the loop, but it can be extended with functional properties. Information about the variables read and written can be used directly to verify whether a loop can be parallelised (because all iterations are independent), or whether a loop can be vectorised (i.e., it can be safely parallelised in lock step). In the latter case, the verification requires us to add some additional annotations that indicate the minimal synchronisation that is necessary between the different iterations. Note that this is different from a loop invariant, where a condition must hold immediately before (and after) each iteration. Loop invariants do not give us any insights in how to parallelise a loop.

Example 1. Suppose we have the following loop:

```
//@ requires a.length == b.length;
for (int i = 0; i < a.length; i++) {
        a[i] = 2 * b[i];
}
```

We can prove that this loop respects the following iteration contract:

```
/*@
requires Perm(a[i], 1) ** Perm(b[i], 1/2);
ensures Perm(a[i], 1) ** Perm(b[i], 1/2);
ensures a[i] == 2 * b[i];
@*/
```

where the *requires* and *ensures* keywords indicate the pre- and postcondition, respectively, and $**$ is the ascii-notation for $\star$. From the fact that we can verify this iteration contract, we can conclude that this loop can be safely parallelised without changing its behaviour [4]. Moreover, from this iteration contract we can also conclude that after successful termination of this (parallelised) loop, we have `(\forall int i; 0 <= i && i < a.length; a[i] == 2 * b[i])`.

Example 2. Now suppose that we have the following loop (with forward dependencies), where all arrays `a`, `b` and `c` have length `N`:

```
//@ requires a.length == N;
//@ requires b.length == N;
//@ requires c.length == N;
for (int i = 0; i < N; i++) {
      b[i] = c[i] * 2;
      if (i > 0) {
         a[i] = b[i-1];
      }
}
```

Also for this loop we can give an iteration contract:

```
/*@
Perm(a[i], 1) ** Perm(b[i], 1) ** Perm(c[i], 1/2);
ensures Perm(a[i], 1) ** Perm(b[i], 1/2) ** Perm(c[i], 1/2);
ensures i > 0 ==> Perm(b[i -1], 1/2);
ensures i == N - 1 ==> Perm(b[i], 1/2);
@*/
```

However, in order to verify this iteration contract, we need to add an extra annotation, indicated by the keyword send, in the loop that indicates that a read permission on `b[i]` is transferred from one iteration to another. The annotation that we need here is:

```
for (int i = 0; i < N; i++) {
     b[i] = c[i] * 2;
     //@ send i < N - 1 ==> Perm(b[i], 1/2), 1
     if (i > 0) {
        a[i] = b[i-1];
     }
}
```

This send-annotation captures that after the first assignment, the permission to access `b[i]` is transferred to iteration `i + 1` as indicated by the "`,1`'. Thus, if we execute the loop instructions in parallel, at this point synchronisation will be needed. We can either vectorise the loop (i.e., execute the instructions in a lock-based manner) or parallelise it by using appropriate synchronisations. For clarity we can add the matching receive-annotation, but this is not strictly necessary:

```
//@ receive i > 0 ==> Perm(b[i - 1], 1/2);
```

Note that also in this case, the iteration contract can be extended with functional properties expressing how the contents of the different arrays are updated. For this loop, one could prove that `(\forall int i; 0 <= i && i < N; i > 0 ==> a[i] = c[i - 1] * 2)` holds afterwards. However, in order to prove this, also knowledge about the contents of `c[i-1]` would be necessary, thus also a (read) permission on `c[i]` would have to be transferred from iteration `i` to `i+1`.

2.3 Verification of OpenCL Kernels

One possibility to improve the performance of a program is to compile into a GPU-compliant kernel program (using e.g. the PENCIL compiler, developed within the CARP project [3]). Also OpenCL programs can be verified using permission-based separation logic. We briefly describe the main ideas of the verification approach, for full details we refer to [6].

An OpenCL kernel program divides the work over a group of work groups. Each work group consists of a fixed number of threads. Each thread executes the same, sequential program. To avoid data races, each thread should only access its own part of the shared data. Threads within a work group synchronise by means of a barrier.

To verify an OpenCL kernel, the behaviour of each thread should be specified with a pre- and postcondition. The specification of a work group is the universal separating conjunction of all its thread specifications. The specification of a kernel is the universal separating conjunction of all its work group specifications. Note that wellformedness of the universal separating conjunction expressions guaranties that the total sum of all accesses to a location never exceeds 1, thus guarantying data race freedom. Each thread is verified using standard sequential program verification techniques. Barriers need a specification that indicates how they transfer permissions. When a thread invokes a barrier, it has to fulfil the barrier precondition, and then can assume the barrier postcondition. Additionally, it has to be shown that the barrier only re-distributes the resources that are handed in by the threads upon entering the barrier.

Example 3. As an example, we show the annotated version of a simple kernel that rotates the elements of an array to the right[1]. The invariant property specifies a property that holds throughout the execution. Further we specify the behaviour of a single kernel thread (the specification of the work group and

[1] As mentioned above, currently the VerCors tool set does not support all OpenCL features. Therefore we have actually verified a PVL variant of this kernel.

kernel can be derived from this). The precondition just indicates the necessary permissions, the postcondition also specifies the rotation. Read permissions are split over all threads; for readability therefore we simply write `read` instead of giving the precise fraction. Each thread, which is responsible for one array location, first reads the location on the left of this one, then it synchronises at the barrier (where it gives up the read permission on its left location, and obtains a write permission on its own location) and subsequently it writes the value read before to the location it is responsible for.

```
/*@
    invariant array != NULL;
    requires get_global_id(0) != 0 ==>
                Perm(array[get_global_id(0) - 1], read);
    requires get_global_id(0) == 0 ==> Perm(array[size - 1], read);
    ensures Perm(array[get_global_id(0)], 1);
    ensures get_global_id(0) != 0 ==>
                array[get_global_id(0)] == \old(array[get_global_id(0) -
                      1]);
    ensures get_global_id(0) == 0 ==>
                array[get_global_id(0)] == \old(array[size - 1]);
@*/
__kernel void rightRotation(int array[], int size) {
    int tid = get_global_id(0);  // get the index
    int temp;
    if (tid != 0) { temp = array[tid - 1]; }
    else { temp = array[size - 1]; }
    /*@
         requires tid != 0 ==> Perm(array[tid - 1], read);
         requires tid == 0 ==> Perm(array[size - 1], read);
         ensures Perm(array[tid], 1);
    @*/
    barrier(CLK_GLOBAL_MEM_FENCE);
    array[tid] = temp;
}
```

3 From Loop to Kernel Program

This section shows the connection between loops specified with iteration contracts and annotated OpenCL kernels (as also discussed in [4]). We assume that the code is compiled using a basic parallelising compiler, without further optimisation. In the next section, we will look into how further optimisations can be applied, while still preserving correctness.

Independent Loops. Given an independent loop, the basic compilation to kernel code is simple: create a kernel with as many threads as there are loop iterations and each kernel thread executes one iteration. Moreover, the iteration contract can be used as the thread contract for each parallel thread in the kernel directly. The size of the work group can be chosen at will, because no barriers are used.

Example 4. Consider the simple loop in Example 1. This can be compiled into the following annotated OpenCL kernel[2].

```
/*@
   invariant a != NULL;
   invariant b != NULL;
   requires Perm(a[get_global_id(0)], 1);
   requires Perm(b[get_global_id(0)], 1/2);
   ensures Perm(a[get_global_id(0)], 1);
   ensures Perm(b[get_global_id(0)], 1/2);
   ensures a[get_global_id(0)] == 2 * b[get_global_id(0)];
@*/
__kernel void example1(int a[], int b[]) {
     int tid = get_global_id(0);
     a[tid] = 2 * b[tid];
}
```

Forward Loop-Carried Dependencies. Given a loop, we consider loop-carried dependency as: if there exists two statements S_{src} and S_{dst} in the body of the loop and there exists two iterations i and j such that first $i < j$, second, in iteration i an instance in S_{src} and in iteration j an instance in S_{dst} access to the same location and third, at least one of them is a write. In this case we have loop-carried dependency and if S_{src} syntactically appears before S_{dst} we call it forward loop-carried dependency.

If the loop has forward dependencies, the kernel must ensure that its body respects these dependencies. Thus in particular, any send-annotation that is necessary to verify the iteration contract results in a barrier, where the barrier specification is derived from the send (and corresponding receive) annotation.

```
for(int j=0; j < N; j++)
/*@
  requires $\phi(j)$;
  ensures $\psi(j)$;
@*/
{
   $L_1$: if($g_1(j)$) { $I_1(j)$; }
      ⋮
  $L_k$: if($g_k(j)$) { $I_k(j)$; }
}
```

$\Rightarrow$

```
/*@
requires $\phi$(tid);
ensures $\psi$(tid);
@*/
__kernel loop(...)
{
  $C_1$(tid);
  ⋮
  $C_k$(tid);
}
```

Fig. 2. From a loop with a forward loop-carried dependency to an OpenCL kernel

[2] Note that the invariant properties are necessary to prove the iteration contract correct, but they are given as a global specification.

Consider the loop pattern for a loop with a forward loop-carried dependency on the left side of Fig. 2. Each statement S_k which is labeled by L_k consists of an atomic instruction I_k and its guard g_k. For simplicity, we assume that both the number of threads and the size of the work group is N. Naive compilation generates the annotated kernel on the right side of the figure, where:

- if $I_k(j)$ is a `send` annotation then it is ignored: $C_k(j) \equiv \{\ \}$
- if $I_k(j)$ is a `receive` statement with a matching `send` statement at L_i, then it is replaced by a barrier $C_k(j) \equiv$

  $$\text{barrier}(\dots)\text{ requires } g_i(j) \Rightarrow \phi_{send}(j);\text{ ensures } g_k(j) \Rightarrow \phi_{receive}(j);$$

  where the barrier contract specifies how the permissions are exchanged at the barrier.
- if $I_k(j)$ is any other statement then it is copied:

  $$C_k(j) \equiv \texttt{if}\ (g_k(j)\ \{I_k(j);\}$$

Example 5. Applying this approach to the loop in Example 2 results in the following kernel.

```
/*@
   invariant a != NULL;
   invariant b != NULL;
   invariant c != NULL;
   requires Perm(a[get_global_id(0)], 1));
   requires Perm(c[get_global_id(0)], 1/2);
   requires Perm(b[get_global_id(0)], 1));
   ensures Perm(a[get_global_id(0)], 1));
   ensures Perm(c[get_global_id(0)], 1/2);
   ensures Perm(b[get_global_id(0)], 1/2));
   ensures get_global_id(0) > 0 ==> Perm(b[get_global_id(0) - 1], 1/2);
   ensures get_global_id(0) == N -1 ==> Perm(b[get_global_id(0)], 1/2);
@*/
__kernel void example2(int a[], int b[], int c[], int N) {
     int tid = get_global_id(0);
     b[tid] = c[tid] * 2;
     /*@
        requires Perm(a[tid], 1) ** Perm(b[tid], 1);
        requires Perm(c[tid], 1/2);
        ensures Perm(a[tid], 1) ** Perm(b[tid], 1/2);
        ensures Perm(c[tid], 1/2);
        ensures tid > 0 ==> Perm(b[tid - 1], 1/2);
        ensures tid == N - 1 ==> Perm(b[tid], 1);
     @*/
     barrier(CLK_GLOBAL_MEM_FENCE);
     if (tid > 0) {
        a[tid] = b[tid-1];
   }
}
```

```
// kernel code
/*@
requires Pre(tid, iter);
ensures Post(tid, iter);
@*/
__kernel K(int iter) {
   body (tid, iter);
}


// hostcode
//@ loop_invariant (\forall tid;
    Post(tid, i));
for (int i = 0; i < N; i++) {
  invoke kernel K(i);
}
```

$\Rightarrow$

```
// iterating kernel
/*@
requires Pre(tid, 0);
ensures Post(tid, N - 1);
@*/
__kernel K(int N) {
   body (tid, 0);
   //@ loop_invariant Post(tid, i -
         1);
   for (int i = 1; i < N; i ++) {
      // barrier_specification
      /*@
      requires Post(tid, i - 1)
      ensures Pre(tid, i);
      @*/
      barrier();
      body (tid, i);
   }
}
```

Fig. 3. Sketch of barrier introduction transformation

4 Correctness and Compiler Optimisations

The previous section discussed compilation from verified source programs to verified target programs. This has the advantage that the main verification effort can be invested into the high-level program, which in this concrete case is still a sequential program, and therefore its verification is still relatively easy to follow. By compiling the program and its specification, the result is a verifiable parallel OpenCL program.

However, as mentioned above, the compiler performs a very naive compilation, and the code will probably not be performing very well (of course, the concrete examples in the previous section are all very small, but imagine a similar approach is applied to larger and more realistic examples). Therefore, what typically happens in the design of an OpenCL kernel is that the kernel developer then starts to change and optimize the implementation, in order to improve the performance of the application. In the literature, a large collection of program transformations to improve performance of GPU applications is available, see e.g. [2,13,17,20]. We argue that such transformations should be specified formally, and whenever the transformation is applied to the code, the corresponding specifications also should be transformed, in such a way that the resulting program can be verified again (provided the original program could be verified). This enables a developer to apply various optimisations, trying to identify the best program performance, while making sure that the functionality of the application is not affected.

This section describes several commonly used program transformations that are used to make OpenCL programs more efficient. For each of these program transformations, we sketch the corresponding specification transformations. This description is still handwaving; it is future work to define the specification transformation precisely, and to implement it as part of the VerCors tool set. However, we believe that the sketches already show the feasibility of the approach. The other existing approach is to have a verified compiler to generate a verified low-level code. This method is very expensive and is not general enough in practice. We conjecture that an approach like the one advocated here is the only feasible way to make verification of high-performance, low-level concurrent software possible. Notice that, in our method, it is not necessary to have a meta-theorem that proves correctness of all individual transformations, it is sufficient to reverify the resulting annotated program.

Barrier Introductions. One common optimisation is to take an OpenCL kernel that implements a single iteration, and that is invoked repeatedly, and to transform this into an OpenCL kernel that repeatedly executes the same code. In order to do this safely, barriers need to be inserted to ensure all threads are working on the same iteration of the code.

Figure 3 sketches this transformation, both on the code and on the specifications. In the original program, we have host code that invokes a kernel `K` multiple times. The behaviour of each thread inside the kernel is specified by a thread specification, which depends on how many times the kernel has already been invoked (variable iter). In the transformed program, the new kernel `K` is invoked only once. It will first execute the original kernel body once, followed by `N - 1` iterations, where each iteration first synchronises on a barrier and then executes the body another more. The new thread's precondition is the precondition of the very first iteration; the new thread's postcondition is the postcondition of the very last iteration. In order to make the code verifiable, a loop invariant and barrier specifications are added. The loop invariant states that the postcondition of the previous iteration holds. The barrier specification states that when the barrier is entered, the postcondition of the previous iteration holds, and that the precondition of the next iteration can be assumed. Since after the barrier, the body will be executed again, the loop invariant will be re-established. Thus, if the original program could be verified, the annotated program resulting from this transformation should also be verifiable.

Change of Data Locations. Another commonly used optimisation to improve the performance of an OpenCL kernel is to move the data from global to local memory (as the work groups have very fast access to local memory). For simplicity, Fig. 4 describes this transformation for the case of a single work group.

```
/*@
requires Pre();
ensures Post();
@*/
__kernel K(args) {
   body (tid);
}
```

⇒

```
/*@
requires Pre();
ensures Post();
@*/
__kernel K(args) {
   copy data from global to local memory
   // barrier_specification
   /*@
   requires Pre()
   ensures Pre_local() // Pre[local/global]
   @*/
   barrier();
   body[local/global] (tid);
   // barrier_specification
   /*@
   requires Post_local() // Post[local/global]
   ensures Post()
   @*/
   barrier();
   copy data from local to global memory
}
```

Fig. 4. Sketch of change of data location transformation

Suppose that the behaviour of the original kernel was specified with precondition `Pre()` and postcondition `Post()`, capturing a property over the global memory. The new kernel first copies all global data to local memory, and then all threads synchronise with a barrier. The annotations show that at this point, the precondition still holds, except that all global memory locations are substituted by local memory locations. Next, the body of the kernel is executed, but again with all global memory locations substituted by the corresponding local memory locations. This should establish the kernel postcondition, modulo the local memory locations. This information is again made explicit as a barrier specification. After the second barrier, the local memory is copied back to global memory, and the kernel ends. Again, if the original program could be verified, the annotated program resulting from this transformation should also be verifiable.

Notice that if the work is distributed over multiple work groups, a similar approach can be used, but the transformation needs to be written down more precisely, to take the distribution of the data over the different work groups into account.

Data Redistribution. In a naive compilation from a sequential loop to an OpenCL kernel, one often ends up with each thread accessing a single entry of the data. This means that there are many threads, and the individual threads do not have that much work to do. An obvious optimisation is to increase the amount of work that each thread has to do by making it iterate over multiple entries of the data. This transformation is sketched in Fig. 5.

```
/*@
requires Pre(tid);
ensures Post(tid);
@*/
 __kernel K(args) {
   body (tid);
}
```

⇒

```
/*@
requires (\forall int i; tid * N <= i && i < (tid + 1) * N;
                     Pre(i));
ensures (\forall int i; tid * N <= i && i < (tid + 1) * N;
                     Post(i));
@*/
 __kernel K(args, N) {
   /*@
   loop_invariant tid * N <= j && j < (tid + 1) * N;
   loop_invariant (\forall int i; tid * N <= i && j < i;
                              Post(i));
   @*/
   for (int j = tid * N, j < (tid + 1) * N, j++) {
      body(j);
   }
}
```

Fig. 5. Sketch of data redistribution transformation

Where in the original kernel, each thread satisfied a pre-postcondition specification for a single item, now the properties become universally quantified. In the transformation as described here, thread `tid` now works on the data entries from `tid * N` to `(tid + 1)* N - 1`, where `N` is the factor that determines how the transformation redistributed the data. Therefore, the precondition now requires that the original precondition for all these entries is satisfied, and the kernel will establish the postcondition for all these entries. In order to prove this, we need some loop invariants that indicate that the postcondition is established for all data entries handled so far by the current thread. Again, if the original kernel program could be verified, also the transformed program can be verified.

Matrix Representations. In the case of 2-dimensional data, memory access patterns can have a great impact on the performance of the kernel: in certain cases column-major access patterns will perform better, in other cases row-major access patterns will be better (this can depend for example on the specific hardware on which the kernel is executed; notably there is a difference between CPU and GPU here [19]). Again, this can be described as a program transformation, where all `M * N` matrices are transformed into `N * M` matrices, and all matrix access operations `m[i,j]` are rewritten into `m[j,i]`. In that case, the same rewriting has to be applied on all specifications (taking care that all expression bounds are still correct) in order to make sure that the transformed program remains verifiable.

Loop Unrollings. Finally, the last program transformation that we consider is loop unrolling. This increases the code size, but can have a positive impact on program performance, for example because it simplifies control flow and can reduce branch penalty.

```
__kernel K(args) {
  /*@
  loop_invariant 0 <= i && i < N;
  loop_invariant Inv(i);
  @*/
  for (int i = 0; i < N; i++) {
    body (i);
  }
}
```

⇒

```
__kernel K(args) {
  body(0);
  /*@
  loop_invariant 1 <= i && i < N;
  loop_invariant Inv(i);
  @*/
  for (int i = 1; i < N; i++) {
    body (i);
  }
}
```

Fig. 6. Sketch of loop unrolling transformation

Figure 6 sketches a very basic loop unrolling example, where the first iteration of the loop is taken out of the loop body. Note that this transformation is only possible if we know that `N > 0`. The resulting kernel and its specification look almost the same, however the loop invariant that provides the bound on the loop variable has been adjusted. Again, if the original kernel could be verified, also the kernel resulting from the transformation can be verified.

5 Future Research Challenges

In this paper, we have argued that to develop provably correct low-level, high-performance code, verification should be part of the development chain. After proving an initial, unoptimised version of the program correct, both program and annotations should be gradually transformed in such a way that the program can be reverified after each transformation (where compilation from one language to another can also be considered as a form of program transformation). This paper considered some concrete cases, and showed how this could be achieved. However, the description is still very informal and hand-waiving.

In order to turn this approach into a full-fledged verification methodology, we still need to extend it much further. This section concludes the paper by outlining the open research challenges that need to be addressed.

– The compilation that we discussed in this paper was very limited. In practice, one would want to apply this on more complex examples, for example containing multiple parallelisable loops. In order to support this approach, the compilation step for program and annotations should be defined compositionally: for basic building blocks we define how they are compiled, and on top of that we define the compilation approach for more complex programs (and their annotations).
– The program transformations as they are defined here are all given in a handwaving manner, and skip over many details. For a fully general approach, we would have to define this much more precisely, maybe with special instances in which the compilation can be simplified.
– The program transformations as they are discussed in this paper are given in ad hoc manner. For a more systematic approach, one would need to define a catalogue of known program transformations, with many variations (and possibly combine this with performance considerations, i.e., in which case would one expect a transformation to improve performance).
– The program transformations should be thoroughly tested, in particular to ensure that all auxiliary annotations that are necessary to reverify the program are indeed generated.

Acknowledgements. We are indebted to Jeroen Vonk, who started studying some of the annotation-aware program transformations as part of his Bachelor project. This work is supported by the ERC 258405 VerCors project (Huisman, Blom), the EU FP7 STREP 287767 CARP project (Huisman, Blom, Darabi), and the NWO VICI 639.023.710 Mercedes project (Huisman, Safari).

References

1. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1) (2015)
2. Amini, M.: Source-to-source automatic program transformations for GPU-like hardware accelerators. Master's thesis, Ecole Nationale Supérieure des Mines de Paris (2012)
3. Baghdadi, R., et al.: PENCIL: towards a platform-neutral compute intermediate language for DSLs. CoRR, abs/1302.5586 (2013)
4. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 202–217. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_14
5. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
6. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014)

7. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
8. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
9. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: a logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 207–231. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_9
10. Darabi, S.: Verification of program parallelization. Ph.D. thesis, University of Twente (2018)
11. Darabi, S., Blom, S.C.C., Huisman, M.: A verification technique for deterministic parallel programs. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 247–264. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_17
12. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
13. Huang, D., et al.: Automated transformation of GPU-specific OpenCL kernels targeting performance portability on multi-core/many-core CPUs. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 210–221. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_18
14. Jung, R., et al.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL, pp. 637–650. ACM (2015)
15. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.-H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 696–723. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_26
16. Leavens, G.T., et al.: JML Reference Manual. Department of Computer Science, Iowa State University, February 2007. http://www.jmlspecs.org
17. Nandakumar, D.: Automatic translation of CUDA to OpenCL and comparison of performance optimizations on GPUs. Master's thesis, University of Illinois at Urbana-Champaign (2011)
18. Sergey, I., Nanevski, A., Banerjee, A.: Specifying and verifying concurrent algorithms with histories and subjectivity. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 333–358. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_14
19. Shen, J.: Efficient high performance computing on heterogeneous platforms. Ph.D. thesis, Technical University of Delft (2015)
20. Wu, B., Chen, G., Li, D., Shen, X., Vetter, J.: Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In: ICS 2015 Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 119–130. ACM (2015)