

# ALPINIST: an Annotation-Aware GPU Program Optimizer<sup>\*</sup>

Ömer Şakar<sup>1</sup>, Mohsen Safari<sup>1</sup>, Marieke Huisman<sup>1</sup>, and Anton Wijs<sup>2</sup>

<sup>1</sup> Formal Methods and Tools, University of Twente, Enschede, The Netherlands  
{o.f.o.sakar,m.safari,m.huisman}@utwente.nl

<sup>2</sup> Software Engineering & Technology, Eindhoven University of Technology,  
Eindhoven, The Netherlands  
a.j.wijs@tue.nl

**Abstract.** GPU programs are widely used in industry. To obtain the best performance, a typical development process involves the manual or semi-automatic application of optimizations prior to compiling the code. To avoid the introduction of errors, we can augment GPU programs with (pre- and postcondition-style) annotations to capture functional properties. However, keeping these annotations correct when optimizing GPU programs is labor-intensive and error-prone.

This paper introduces ALPINIST, an annotation-aware GPU program optimizer. It applies frequently-used GPU optimizations, but besides transforming code, it also transforms the annotations. We evaluate ALPINIST, in combination with the VerCors program verifier, to automatically optimize a collection of verified programs and reverify them.

**Keywords:** GPU · Optimization · Deductive verification · Annotation-aware · Program transformation

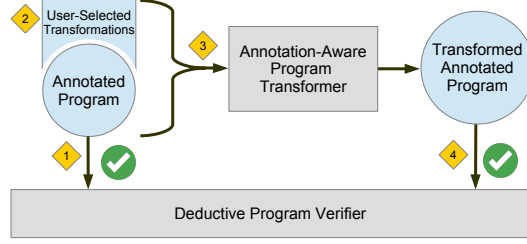
## 1 Introduction

Over the course of roughly a decade, graphics processing units (GPUs) have been pushing the computational limits in fields as diverse as computational biology [62], statistics [34], physics [6], astronomy [23], deep learning [28], and formal methods [16, 42, 63]. Dedicated programming languages such as CUDA [33] and OpenCL [41] can be used to write GPU source code. To achieve the most performance out of GPUs, developer should apply incremental optimizations, tailored to the GPU architecture. Unfortunately, this is to a large extent a *manual* activity. The fact that for different GPU devices, the same code tends to require a different sequence of transformations [20] makes this procedure even more time consuming and error-prone. Recently, automating this has received some attention, for instance by applying machine learning [2].

Reasoning about the correctness of GPU software is hard, but necessary. Multiple verification techniques and tools have been developed to aid in this task

---

<sup>\*</sup> This work is supported by NWO grant 639.023.710 for the Mercedes project and by NWO TTW grant 17249 for the ChEOPS project



**Fig. 1:** Annotation-Aware Program Transformation.

aimed at detecting data races, see e.g., [7, 9, 13, 31, 32], and for a recent overview, see [21]. Some of these techniques apply deductive program verification, which requires a program to be *manually* augmented with pre- and postcondition-style annotations. However, annotating a program is often time consuming. The more complex a program is, the more challenging it becomes to annotate it. In particular, as a program is being optimized repeatedly, its annotations tend to change frequently.

This paper presents ALPINIST, a tool that can apply *annotation-aware transformations* [25] on annotated GPU programs. It can be used with the deductive program verifier VerCors [8]. VerCors can verify the *functional correctness* of GPU programs [9]. It allows the verification of many typical GPU computations, see e.g., [46, 48, 49]. The purpose of ALPINIST is twofold (see Fig. 1): First, it automates the optimization of GPU code, to the extent that the developer needs to indicate which optimization needs to be applied where, and the tool performs the transformation. Interestingly, the presence of annotations is exploited by ALPINIST to determine whether an optimisation is actually applicable, and in doing so, can sometimes apply an optimization where a compiler cannot. Second, as it applies a code transformation, it also transforms the related annotations, which means that once the developer has annotated the unoptimized, simpler code, any further optimized version of that code is automatically annotated with updated pre- and postconditions, making it reverifiable. This avoids having to re-annotate the program every time it is optimized for a specific GPU device.

ALPINIST supports GPU code optimizations that are used frequently in practice, namely loop unrolling, tiling, kernel fusion, iteration merging, matrix linearization and data prefetching. In the current paper, we discuss how ALPINIST has been implemented, how it can be applied on annotated GPU code, and how some of the more complex optimizations work. In addition, we evaluate the effect of applying several of these optimizations, both in terms of annotation size and time needed to verify a program, to a collection of examples including the verified case studies in [46, 47, 49].

*Outline.* Section 2 demonstrates how ALPINIST optimizes a verified GPU program while preserving its provability. Section 3 discusses the architecture of ALPINIST. Section 4 discusses the most complex optimizations supported by ALPINIST in detail, namely loop unrolling, tiling and kernel fusion, and briefly

```

1  /*@ context_everywhere N > 0 && N < a.length;
2  req (\forallall* int i; 0 <= i < a.length; Perm(a[i], 1));
3  ens (\forallall* int i; 0 <= i < a.length; i != a.length-1 ==> Perm(a[i+1], 1));
4  ens (\forallall* int i; 0 <= i < a.length; i == a.length-1 ==> Perm(a[0], 1));
5  ens (\forallall int i; 0 <= i < a.length-1; a[i+1] == N*i);
6  ens a[0] == N*(a.length-1); @*/
7  void Host(int[] a, int size, int N) {
8    par Kernel1 (int tid = 0 .. a.length)
9      /*@ context Perm(a[tid], 1);
10     ens a[tid] == 0; @*/
11     { a[tid] = 0; }
12    par Kernel2 (int tid = 0 .. a.length)
13      /*@ context tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
14      req tid != a.length-1 ? a[tid+1] == 0 : a[0] == 0;
15      ens tid != a.length-1 ? a[tid+1] == N*tid : a[0] == N*tid; @*/
16      { /*@ inv k >= 0 && k <= N;
17        inv tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
18        inv tid != a.length-1 ? a[tid+1] == k*tid : a[0] == k*tid; @*/
19        for(int k = 0; k < N; k++) {
20          if (tid != a.length-1) { a[tid+1] = a[tid+1] + tid; }
21          else { a[0] = a[0] + tid; }
22        } } }

```

Fig. 2: A verified GPU-style program

discusses the remaining three. Section 5 presents the results of experiments in which the tool has been applied on a collection of programs. Section 6 discusses related work and Section 7 concludes the paper, and discusses future work.

## 2 Annotation-Aware Optimization using ALPINIST

This section illustrates how ALPINIST can optimize a verified GPU program while preserving its provability. Fig. 2 shows a GPU program with annotations [9] that is verified by VerCors. The example is written in a simplified version of VerCors’ own language PVL. The program initializes an array `a`, and subsequently updates the values in `a`, `N` times. The workflow of a GPU program in general is that the host (i.e., CPU) invokes a *kernel*, i.e., a GPU function, executed by a specified number of GPU threads. These threads are organized in one or more *thread blocks*. In this program, there are two kernels, both executed by one thread block of `a.length` threads (lines 8 and 12 (l.8, l.12))<sup>3</sup>. Each thread has a unique identifier, in the example called `tid`. In the first kernel (l.8-l.11), each thread initializes `a[tid]` to 0. In the second kernel (l.12-l.22), each thread updates `a[tid+1]` (modulo `a.length`) `N` times, by adding `tid` to it. In the main `Host` function, `Kernel1` is called, followed by `Kernel2`.

The kernels, the for-loop and the host function are annotated for verification (in blue), using permission-based separation logic [5, 10, 11]. Permissions capture which memory locations may be accessed by which threads; they are fractional values in the interval  $(0, 1]$  (cf. Boyland [11]): any fraction in the interval  $(0, 1)$  indicates a read permission, while 1 indicates a write permission. A write

<sup>3</sup> In practice, the size of a block cannot exceed a specific upper-bound, but for this example, we assume that `a.length` is sufficiently small.

```

1  /*@ context_everywhere N > 0 && N < a.length;
2  req (\forallall* int i; 0 <= i < a.length; Perm(a[i], 1));
3  ens (\forallall* int i; 0 <= i < a.length; i != a.length-1 ==> Perm(a[i+1], 1));
4  ens (\forallall* int i; 0 <= i < a.length; i == a.length-1 ==> Perm(a[0], 1));
5  ens (\forallall int i; 0 <= i < a.length-1; a[i+1] == N*i);
6  ens a[0] == N*(a.length-1); @*/
7  void Host(int[] a,int size,int N){
8      par Fused_Kernel(int tid = 0 .. a.length)
9          /*@ req Perm(a[tid], 1);
10         ens tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
11         ens tid != a.length-1 ? a[tid+1] == N*tid : a[0] == N*tid; @*/
12         {
13             a[tid] = 0;
14             /*@ req Perm(a[tid], 1);
15             req a[tid] == 0;
16             ens tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
17             ens tid != a.length-1 ? a[tid+1] == 0 : a[0] == 0; @*/
18             barrier(Fused_Kernel)
19
20             int a_reg_0, a_reg_1;
21             if (tid != a.length-1) { a_reg_1 = a[tid+1] } else { a_reg_0 = a[0] }
22             int k = 0;
23             if (tid != a.length-1) { a_reg_1 = a_reg_1 + tid; }
24             else { a_reg_0 = a_reg_0 + tid; }
25             k ++;
26             /*@ inv k >= 0 + 1 && k <= N;
27             inv tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
28             inv tid != a.length-1 ? a_reg_1 == k*tid : a_reg_0 == k*tid; @*/
29             for(k; k < N; k++) {
30                 if (tid != a.length-1) { a_reg_1 = a_reg_1 + tid; }
31                 else { a_reg_0 = a_reg_0 + tid; }
32             }
33             if (tid != a.length-1) { a[tid+1] = a_reg_1 } else { a[0] = a_reg_0 };
34         } }

```

Fig. 3: An optimized GPU-style program, annotated for verification

permission can be split into multiple read permissions and read permissions can be added up, and transformed into a write permission if they add up to 1. The soundness of the logic ensures that for each memory location, the total number of permissions among all threads does not exceed 1.

To specify permissions, predicates are used of the form  $\text{Perm}(L, \pi)$  where  $L$  is a heap location and  $\pi$  a fractional value in the interval  $(0, 1]$  (e.g.,  $1/3$ ). Pre- and postconditions, denoted by keywords **req** and **ens**, should hold at the beginning and the end of an annotated function, respectively. The keyword **context** abbreviates both **req** and **ens** (l.9, l.13). The keyword **context\_everywhere** is used to specify a property that must hold throughout the function (l.1). Note that  $\backslash\text{forall}^*$  is used to express a universal separating conjunction over permission predicates (l.2-l.4) and  $\backslash\text{forall}$  is used as standard universal conjunction over logical predicates (l.5). For logical conjunction,  $\&\&$  is used and  $**$  is used as separating conjunction in separation logic.

In the example, write permissions are required for all locations in **a** (l.2). The pre- and postconditions of the first kernel specify that each thread needs write permission for **a[tid]** (l.9). The postcondition states that **a[tid]** is set to 0 (l.10). In the second kernel, all threads have write permission for **a[tid+1]**, except thread **a.length-1** which has write permission for **a[0]** (l.13). Moreover,

it is required that  $a[tid+1]$  (modulo  $a.length$ ) is 0 (l.14). For the for-loop (l.19-l.22), loop invariants are specified:  $k$  is in the range  $[0, N]$  (l.16), each thread has write permission for  $a[tid+1]$  (modulo  $a.length$ ) (l.17) and this location always has the value  $k * tid$  (l.18). The postconditions of the second kernel and the host function are similar to this latter invariant.

Fig. 3 shows an optimized version of the program, with updated annotations to make it verifiable. ALPINIST has applied three optimizations:

1. *Fusing the two kernels*: in GPU programs, the only *global* synchronisation points (used, for instance, to avoid data races) exist implicitly between kernel launches. However, if such a global synchronisation point is not really needed between two specific kernels, then fusing them gives several benefits, in particular the ability to store intermediate results in (fast) thread-local register memory as opposed to (slow) GPU global memory, and it has a positive effect on power consumption [60]. In the example, the kernels are combined into `Fused_Kernel`, and a *thread block-local* barrier is introduced (l.18) to avoid data races within the single thread block executing the code.
2. *Using register memory*; register variables can be used to reduce the number of global memory accesses. Here, the use of `a_reg_0` and `a_reg_1` has been enabled by kernel fusion.
3. *Unrolling the for-loop*; the for-loop has been unrolled once here (l.20-l.25). Since GPU threads are very light-weight, compared to CPU threads, any checking of conditions that can be avoided benefits performance. When unrolling a loop, this means that fewer checks of the loop-condition are needed. Note that here, ALPINIST benefits from the knowledge that  $N > 0$  (l.1), so it knows that the for-loop can be unrolled at least once.

To preserve provability of the optimized program, ALPINIST changed the annotations, in particular the pre- and postcondition of the fused kernel and the loop invariants (highlighted in Fig. 3). Moreover, ALPINIST introduced an annotated barrier (l.14-l.18). Since threads synchronize at a barrier, it is possible to redistribute the permissions. In the rest of the paper, we discuss how ALPINIST performs these annotation-aware transformations.

### 3 The Design of ALPINIST

This section gives a high-level overview of the design of ALPINIST. The optimizations supported by ALPINIST are discussed in Section 4. To understand the design of ALPINIST, we first explain the architecture of the VerCors verifier.

#### 3.1 VerCors' Architecture

VerCors is a deductive program verifier, which is designed to work for different input languages (e.g., Java and OpenCL). It takes as input an annotated program, which is then transformed in several steps into an annotated Silver program. Silver is an intermediate verification language, used as input for Viper [36, 58].

Viper then generates proof obligations, which can be discharged by an automated theorem prover, such as Z3 [35].

The internal transformations in VerCors are defined over our internal AST representation (written in the Common Object Language or COL [50]), which captures the features of all input languages. Some of the transformations are generic (e.g., splitting composite variable declarations) and others are specific to verification (e.g., transforming contracts). The transformations implemented as part of Alpinist are also applied on the COL AST, but they are developed with a different goal in mind, and in particular several of the transformation are specific to the supported optimizations.

Using VerCors and its architecture to implement ALPINIST gives us some benefits. First, existing helper functions can be reused, which simplifies tasks such as gathering information regarding specific AST nodes. Second, some generic transformations of VerCors can be reused, such as splitting composite variable declarations or simplifying expressions. This helps to simplify the implementation of the optimizations. Third, using the architecture of VerCors allows us to prove assertions that we generate relatively easily by invoking VerCors internally.

### 3.2 ALPINIST’s Architecture

ALPINIST takes a verified file as its input, annotated with special optimization annotations that indicate where specific optimizations should be applied. ALPINIST is written in Java and Scala and runs on Windows, Linux and macOS. Fig. 4 gives a high-level overview of the internal design of ALPINIST. The input program goes through four phases: the *parsing* phase, the *applicability checking* phase, the *transformation* phase and the *output* phase.

The *parsing phase* transforms the input file into a COL AST, after which the *applicability checking phase* checks if the optimization can be applied. Some optimizations, such as tiling (see Section 4.2), are always applicable, hence their applicability check always passes. For other optimizations, prerequisites have to be established. Sometimes, a syntactical analysis of the AST suffices, for instance when considering kernel fusion (see Section 4.3). For this optimization, it must be determined whether there is any data dependency between two selected kernels. When analysis of the AST is not enough, VerCors can be used to perform more complex reasoning. An example of this is loop unrolling (see Section 4.1). Its prerequisite is that for the loop to be unrollable  $k$  times, it is guaranteed that the loop executes at least  $k$  times. This prerequisite is encoded as an assertion to be proven by VerCors.

The applicability checking phase is one of the strengths of ALPINIST. It exploits the fact that the input program is annotated to determine whether an optimization is applicable, and relies on the fact that VerCors can perform complex reasoning. Moreover, this approach allows to distinguish failure due to unsatisfied prerequisites and due to mistakes in the transformation procedure.

If the applicability check passes (i.e., the optimization is applicable), the transformation phase is next, otherwise a message is generated that the prerequisites could not be proven.

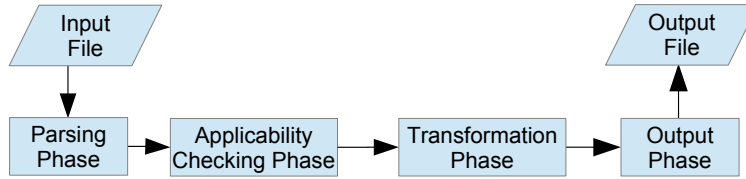


Fig. 4: The internal design of ALPINIST.

The *transformation* phase applies the optimizations to the input AST. The *output phase* either prints the optimized program in the same language as the input program, or a message is printed, signifying either a failure in optimizing or a verification failure in the applicability checking phase.

## 4 GPU Optimizations

ALPINIST currently supports six frequently-used GPU optimizations, namely loop unrolling, tiling, kernel fusion, iteration merging, matrix linearization and data prefetching. This section discusses loop unrolling, tiling, and kernel fusion in detail. The rest of the optimizations follow the same approach in spirit and are discussed only briefly, but they can be found in the implementation of ALPINIST [15]. Each optimization is first introduced in the context of GPU programs. Then, we discuss how to apply them. Interesting insights on their implementation are discussed where relevant.

### 4.1 Loop Unrolling

Loop unrolling is a frequently-used optimization technique that is applicable to both GPU and CPU programs. It unrolls some iterations of a loop, which increases the code size, but can have a positive impact on program performance; e.g., see [20, 37, 44, 57, 61] for its impact, specifically on GPU programs. Fig. 5 shows an example of unrolling an (annotated) loop twice: the body of the loop is duplicated twice before the loop. This has the following effect on the annotations: the loop invariant bounding the loop variable (l.5) changes in the optimized program (l.14). Note that the other loop invariants (i.e., `Inv(i)`) remain the same. Moreover, after each unrolling part, we add all invariants as assertions (l.8-l.10) except after the last unroll. This captures that the code produced by unrolling the loop should still satisfy the original loop invariants.

Our approach to loop unrolling is more general than optimization techniques during compilation. For instance, the `unroll` pragma in CUDA [53] and the `unroll` function in Halide [54] unroll loops by calculating the number of iterations to see if unrolling is possible, i.e., it should be computable at compile time. This difference is illustrated in Fig. 5 where `N` (i.e., the number of iterations) is unknown at compile time. Their approach *cannot automatically* handle this case, while our approach *can automatically* unroll the loop, since annotations

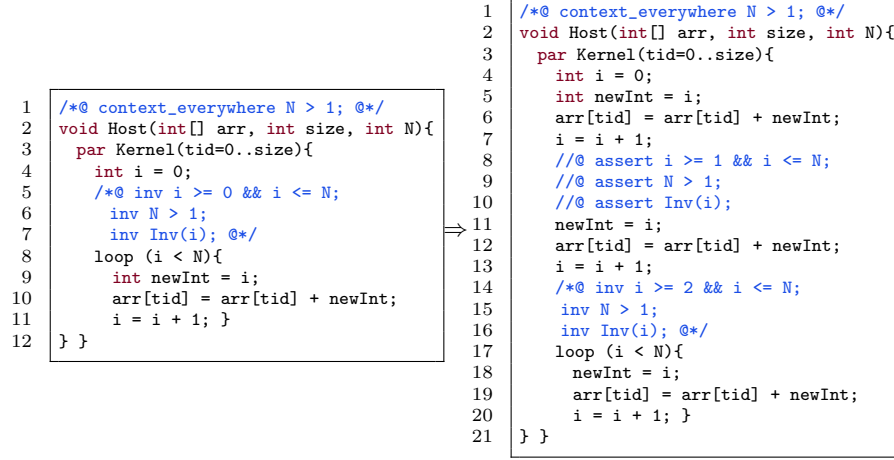


Fig. 5: An example of unrolling a loop 2 times.

```

1  void Host(int[] array, int size){
2      par Kernel(tid=0..size){
3          int i = init; // The loop variable
4          :
5          /*@ assert (i == a) || (i == b); // Depending on initialization of i only one
6              // of the conditions is specified
7          /*@ inv i >= a && i <= b; // The lowerbound of i (a), The upperbound of i (b)
8              inv Inv(i); @*/ // Additional loop invariants
9          loop (cond(i)) { // The loop condition
10             body(i); // The loop body, a sequence of statements in the ith iteration.
11             i = upd(i); } // The update function of i, restricted to (i + c), (i - c),
12 } } // (i × c) or (i/c) where c is a positive integer constant4.

```

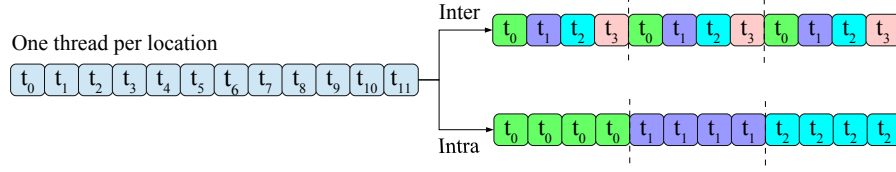
Fig. 6: A general template of a loop inside a kernel.

(l.1, l.6) specify the lower-bound of  $N$  (provided by the programmer, who knows that this is a valid lower-bound). VerCors verifies that the unrolling is valid.

Fig. 6 shows a loop template in a verified GPU program. We would like to automatically unroll the loop  $k$  times and preserve the provability of the program. To accomplish this, we follow a procedure consisting of three parts: the main, checking and updating part. In the *main part*, an annotated (verified) GPU program and positive  $k$  are given as input. Next we go to the *checking part*, to see if it is possible to unroll the loop  $k$  times. This part corresponds with the applicability checking phase. Thus, we statically calculate the number of loop iterations, by counting how many times the condition ( $\text{cond}(i)$ ) holds starting from either  $a$  (as the lowerbound of  $i$ ) or  $b$  (as the upperbound of  $i$ ), depending on the operation of  $\text{upd}(i)$ . If  $k$  is greater than the total number of loop iterations at the end of the checking part, then we report an error. Otherwise we go to the *updating part*, in which we update either  $a$  or  $b$  according to the

<sup>4</sup> If  $c$  was negative, for the multiplication and division,  $i$  would oscillate between positive and negative values and hence would not always be useful as array index. Hence we consider  $c$  to be positive.





**Fig. 7:** Inter- and intra-tiling of an array as  $T = 12$ ,  $N = 4$  and  $\lceil T/N \rceil = 3$ .

```

1 void Host(int[] a, int T){
2   par Kernel(tid = 0..T)
3     /*@ // Preconditions related to permissions and functional correctness
4       req prePerm(a[tid]) ** preFunc(a[tid]);
5       // Postconditions related to permissions and functional correctness
6       ens postPerm(a[tid]) ** postFunc(a[tid]); @*/
7   { body(a[tid]); }
8 }

```

**Fig. 8:** A general unoptimized GPU program to apply for tiling.

operation in `upd(i)`. If the operation is addition or multiplication, then the loop variable `i` (in the unoptimized program) goes from `a` to `b`. That means, after unrolling, `a` should be updated according to the constant `c` from the update expression and `k`. If the operation is subtraction or division, `i` goes from `b` to `a`. Thus, after unrolling, `b` should be updated. After the updating part, we return to the main part to unroll the loop `k` times.

## 4.2 Tiling

Tiling is another well-known optimization technique for GPU programs. It increases the workload of the threads to fully utilize GPU resources by assigning more data to each thread. Concretely, we assume there are  $T$  threads and a one-dimensional array of size  $T$  in the unoptimized GPU program where each thread is responsible for one location in that array (Fig. 8). To apply the optimization, we first divide the array into  $\lceil T/N \rceil$  chunks, each of size  $N$  ( $1 \leq N \leq T$ )<sup>5</sup>. There are two different ways to create and assign threads to array cells (as in Fig. 7):

- *Inter-Tiling* We define  $N$  threads and assign them to one specific location in each chunk. That means each thread serially iterates over all chunks to be responsible for a specific location in each chunk.
- *Intra-Tiling* We define  $\lceil T/N \rceil$  threads and assign one thread to one chunk (i.e., 1-to-1 mapping) to serially iterate over all cells in that chunk.

Both forms of tiling can have a positive impact on GPU program performance; e.g., see [24, 27, 45, 65] for the impact of this optimization.

Fig. 9 shows the optimized version of Fig. 8 by applying inter-tiling. Regarding program optimization, two major changes happen: 1) the total number of threads has reduced (l.2), and 2) the body is encapsulated inside a loop (l.16-l.18). As mentioned, in inter-tiling, we define  $N$  threads instead of  $T$ . The number

<sup>5</sup> Since  $N$  is in the range  $1 \leq N \leq T$ , the last chunk might have fewer cells.

```

1 void Host(int[] a, int T){
2   par Kernel(tid = 0..N)
3     /*@ req (\forallall* int i; 0 <= i && i < ceiling(T, N) && tid+i×N < T;
4         pre(a[tid+i×N]));
5         ens (\forallall* int i; 0 <= i && i < ceiling(T, N) && tid+i×N < T;
6             post(a[tid+i×N])); @*/
7   {
8     int j = 0;
9     /*@ inv j >= 0 && j <= ceiling(T, N);
10    inv (\forallall* int i; 0 <= i && i < ceiling(T, N) && tid+i×N < T;
11        prePerm(a[tid+i×N]));
12    inv (\forallall int i; j <= i && i < ceiling(T, N) && tid+i×N < T;
13        preFunc(a[tid+i×N]));
14    inv (\forallall* int i; 0 <= i && i < j && tid+i×N < T;
15        postFunc(a[tid+i×N])); @*/
16    loop (tid+j×N < T){
17      body(a[tid+j×N]);
18      j = j + 1; }
19  } }

```

**Fig. 9:** Optimized version of the GPU program of Fig. 8 after applying inter-tiling.

of chunks is indicated by the function `ceiling(T, N)`. Each thread in the newly added loop iterates over all chunks (in the range 0 to `ceiling(T, N)-1`) to be responsible for a specific location. This happens by the loop variable `j` and the loop condition `tid+j×N < T`. This means, each thread `tid` can access its own location at index `tid` in each chunk. To preserve verifiability, we add invariants to the loop (l.9-l.17). Therefore, we specify:

- the boundaries of the loop variable `j`, which iterates over all chunks.
- a permission-related invariant for each thread in each chunk (l.10). This comes from the precondition of the kernel and is quantified over all chunks.
- an invariant to indicate functional properties of the locations that have not yet been updated by each thread in the body of the loop (l.12). This comes from the functional property as the precondition of the kernel and is quantified over all chunks.
- an invariant to specify how each thread updates the array in each chunk (l.14). This comes from the functional property as the postcondition of the kernel and is quantified over all chunks.

Moreover, we modify the specification of the kernel (l.3-l.6). Note that we have the condition `tid+j×N < T` in all universally quantified invariants, because the last chunk might have fewer cells than `N`. We quantified the pre- and postcondition of the kernel over the chunks in the same way as the invariants.

Intra-tiling is in essence similar to inter-tiling with two major differences: 1) the total number of threads is `ceiling(T, N)`, and 2) each thread in the loop iterates over cells within its own chunk. Therefore, we have different conditions in the loop and the quantified invariants. ALPINIST also supports this.

Above, each thread is assigned to one cell. This can easily be generalized to have each thread assigned to one or more consecutive cells (i.e., a task). A similar procedure can be applied as long as the tasks do not overlap, i.e., each cell is assigned to at most one thread.

### 4.3 Kernel Fusion

Kernel fusion is a GPU optimization where we merge two or more consecutive kernels into one. It increases the potential to use thread-local registers to store intermediate results (see Section 2) and can lead to less power consumption. See [1, 18, 59, 60, 64] for the impact of kernel fusion on GPU programs. We provide a generalized procedure to fuse an *arbitrary number* of consecutive kernels while considering *data dependency* between them. The idea is to fuse them by repeatedly fusing the first two kernels (i.e., kernel reduction). In each iteration, if there is no data dependency between the two kernels, we safely fuse them. Else if there is only one thread block then we fuse the two kernels by inserting a barrier between the bodies, else fusion fails.

A benefit of this approach is that it only considers two kernels at a time. In this way, it can be determined whether a barrier is necessary between two specific kernels, and we do not miss any possible fusion optimization. Another benefit of this approach is that when a data dependency between two kernels  $P$  and  $P + 1$  ( $1 < P < \#kernels - 1$ ) is detected, the output of the approach is the fusion of the first  $P$  kernels, and the remaining unfused kernels after  $P$ . This allows the user to not only find out that there is a data dependency between  $P$  and  $P + 1$ , but also to obtain fused kernels where possible.

There are multiple challenges in this transformation: (1) how to detect data dependency between two kernels? (2) how to collect the pre- and postconditions for the fused kernel? and (3) how to deal with permissions so that in the fused kernel the permission for a location does not exceed 1? The main difficulty in addressing these challenges is that we have to consider many different possible scenarios. Fortunately, we can use the information from the contract of the two kernels. The permission patterns in the contract indicate for each thread which locations it reads from and writes to. We provide procedures to separately collect pre- and postconditions related to permissions and to functional correctness. Due to space limitations, we only discuss the essential steps to collect the precondition related to permissions for array accesses of the fused kernel in Alg. 1. Collecting the rest of the contract uses a similar procedure.

Alg. 1 requires kernels **k1** and **k2** to not lose any permissions, only possibly redistribute them (using a barrier). Furthermore, for ease of presentation, we assume that in both **k1** and **k2**, each thread accesses at most one cell of array **a**, and that the expressions used to compute array indices only combine constants and thread ID variables, using standard arithmetic operators.

We compare the postcondition of **k1** and the precondition of **k2** (l.2) to understand how to add permissions of the preconditions of **k1** and **k2** to the precondition of the fused kernel. Note that **prePerm** and **postPerm** correspond to a permission-related pre- and postcondition, respectively. We use the postcondition of **k1** for this comparison since the permission at the end of **k1** needs to be sufficient to satisfy the precondition of **k2**. If the index expressions **e1** and **e2** to access an array **a** are syntactically the same, then they refer to the same array cell. In that case, we first add to the precondition of the fused kernel the *original* permission from the precondition of **k1** that corresponds to the permis-

**Algorithm 1** Kernel fusion procedure for collecting precondition permissions.

---

```

1: Add all precondition permissions related to non-shared arrays (i.e., accessed by only one of the
  two kernels) into the contract of the fused kernel kf.
2: for each shared array a with a permission postPerm(a[e1], p1) in the postcondition of the first
  kernel k1 and a permission prePerm(a[e2], p2) in the precondition of the second kernel k2 do
3:   if patterns e1 and e2 are syntactically the same then
4:     Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
5:     if p1 < p2 then
6:       Add prePerm(a[e2], p2-p1) as pre. to kf
7:   else if patterns e1 and e2 are not syntactically the same then
8:     if p1 + p2 ≤ 1 then
9:       Add pre. of k1 corresp. to postPerm(a[e1], p1) and prePerm(a[e2], p2) as pre. in kf
10:    else if p1 + p2 > 1 && p1 < 1 && p2 < 1 then
11:      Add pre. of k1 corresp. to postPerm(a[e1], p1) with permission p3 and prePerm(a[e2],
12:      p4) as pre. s.t. p3 + p4 == 1
13:      else if p1 == 1 (i.e., write) then ▷ Data dependency, add barrier
14:        Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
15:      else p2 == 1 ▷ Data dependency, add barrier
16:        Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
17:        Add prePerm(a[e2], 1-p1) as pre. to kf

```

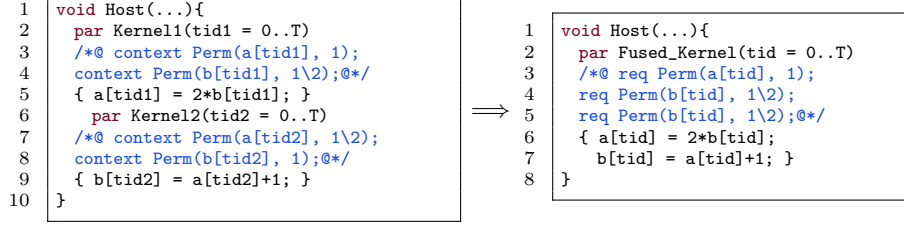
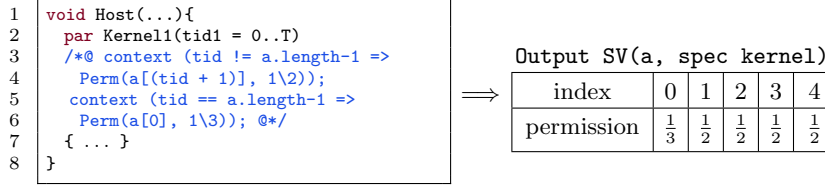
---

sion for **a[e1]** in the postcondition of **k1** (remember that the latter permission may have been obtained in **k1** after permission redistribution). Second, if **p1** is not sufficient for the precondition of **k2** (l.5), we add additional permission to the precondition of the fused kernel to satisfy the precondition of **k2** (l.6).

The remaining different cases in the algorithm correspond to the different edge cases that we should consider when **e1** and **e2** are not syntactically the same. In particular, data dependency happens when the accumulated permission (in both kernels) for one location is greater than 1, and there is at least one write permission. Therefore, we have to distinguish multiple cases: 1) **p1 + p2** does not exceed 1 (l.8), 2) **p1 + p2** exceeds 1, but no write permission is involved (l.10), or 3) and 4) at least one write is involved (l.13 and l.15). In the latter two cases, a barrier must be introduced to take care of distributing permissions from the access in **k1** to the access in **k2**, and possibly additional permission for the latter must be added to the precondition of the fused kernel (l.17). After constructing the contract of the fused kernel, we check for data dependency.

Fig. 10 shows an example of fusing two kernels. We only present the permission precondition expressions which are collected with Alg. 1. There are two shared arrays **a** and **b**. To collect permission preconditions in the fused kernel, we follow steps {l.2→l.3→l.4} for array **a** and steps {l.2→l.3→l.4→l.5→l.6} for array **b**. As there is no data dependency, we can safely fuse the two kernels.

*Implementing Data Dependency Detection.* One of the implementation challenges of kernel fusion is to check for data dependency in the applicability checking phase. To do this for a specific shared array, the function **SV** is used. Fig. 11 shows an example of the output of **SV**. Here, the kernel has  $1 \setminus 2$  permission for **a[tid+1]** and  $1 \setminus 3$  permission for **a[0]** if **tid+1** is out of bounds. **SV** takes the name of an array and the pre- and postconditions of a kernel (of the form **cond(tid) => Perm(a[patt(tid)], p)**) on l.3-l.6, and returns a mapping from indices **patt(tid)** to the permissions **p** (on the right in Fig. 11).

**Fig. 10:** An example of collecting preconditions in fusing two kernels.**Fig. 11:** Example output of the SV function for array **a**

If the function  $SV$  is executed for two kernels to fuse with the same shared array **a**, the results  $SV_1(\mathbf{a})$  and  $SV_2(\mathbf{a})$  can be compared to determine whether there is data dependency between the two kernels. This comparison is described generally at l.8-l.16 in Algorithm 1. For each corresponding location in  $SV_1(\mathbf{a})$  and  $SV_2(\mathbf{a})$ , we can determine, for example, whether both permissions combined do not exceed 1 (l.8) or whether the location in **k1** has write permission (l.12).

#### 4.4 Other Optimizations

We briefly discuss the three remaining optimizations supported by ALPINIST. Iteration merging is an optimization technique that is applicable to both GPU and CPU programs<sup>6</sup>. It merges several iterations of a loop into a single iteration. As a result, there are fewer iterations, which leads to fewer comparison operations in the loop. Iteration merging can have a positive performance impact; see [37, 44, 51] for the effectiveness of this optimization on GPU programs.

Matrix linearization is an optimization where we transform two-dimensional arrays into one dimension ones. This optimization can result in better memory access patterns, thereby improving caching. See [4, 12, 52] for the impact of matrix linearization on GPU programs.

The last optimization implemented in ALPINIST is data prefetching. Suppose there is a verified GPU program where each thread accesses an array location in global memory multiple times. In this optimization, we prefetch the values of those locations that are in global memory into registers which are local to each thread. A similar optimization, in which intermediate results are stored in register memory, is applied in Section 2. Therefore, instead of multiple accesses to the high latency global memory, we benefit from low-latency registers. Data prefetching can have a positive performance impact; see [3, 56, 66].

<sup>6</sup> Iteration merging is also referred to as loop unrolling/vectorization in the literature.

**Table 1:** A summary of the optimization and verification times for all optimizations.

Optimization	Optim. time (s)				Verif. time (orig.) (s)				Verif. time (opt.) (s)			
	min.	max.	avg.	med.	min.	max.	avg.	med.	min.	max.	avg.	med.
Loop unrolling	0.067	0.238	0.116	0.098	7.6	50.7	18.2	14.3	7.6	57.5	20.8	17.3
Tiling	0.044	0.052	0.048	0.047	16.7	21.5	18.7	18.1	19.3	31.4	24.7	20.8
Kernel fusion	0.099	0.338	0.173	0.137	16.7	54.5	24.6	20.0	14.9	22.3	19.0	19.5
Iteration merging	0.042	0.592	0.152	0.097	6.9	51	17.0	12.7	7.3	64	20.0	13.8
Matrix linearization	0.011	0.044	0.022	0.017	11.6	16	14.3	14.1	11.5	16.8	14.4	15.1
Data prefetching	0.010	0.068	0.051	0.053	9.7	23	14.0	13.4	10.4	23	13.5	12.7

## 5 Evaluation

This section describes the evaluation of ALPINIST. The goal is to

- Q1** test whether ALPINIST works on GPU programs.
- Q2** investigate how long it takes for ALPINIST to transform GPU programs and how this affects the verification time.
- Q3** investigate the usability of ALPINIST on real-world complex examples.

### 5.1 Experiment Setup

ALPINIST is evaluated on examples from three different sources. The first source consists of hand-made examples that cover different scenarios for each optimization. The second source is a collection of verified programs from VerCors’ example repository<sup>7</sup>. The third source consists of complex case studies that are already verified in VerCors: two parallel prefix sum algorithms [49], parallel stream compaction and summed-area table algorithms [46], a variety of sorting algorithms [47], a solution [26] to the VerifyThis 2019 challenge 1 [17] and a Tic-Tac-Toe example [55] based on [22]. In total, we applied the optimizations 30 times in the first category, 23 times in the second category and 17 times in the third category (in total 70 experiments). All the examples are annotated with special optimization annotations such that ALPINIST can apply those optimizations automatically. All these examples are publicly available at [14]. All the experiments were conducted on a MacBook Pro 2020 (macOS 11.3.1) with a 2.0GHz Intel Core i5 CPU. Each experiment was performed ten times, after which the average times, i.e., optimization and verification times, of those executions were recorded for the experiment.

### 5.2 Results & Discussion

**Q1** To test whether ALPINIST works on GPU programs, we applied the six optimizations in all 70 experiments and used VerCors to reverify all the resulting programs. All these tests were successful.

**Q2** To investigate how long it takes for ALPINIST to transform GPU programs, we recorded the transformation time for each optimization applied to all the

<sup>7</sup> The example repository of VerCors is available at <https://github.com/utwente-fmt/vercors/tree/dev/examples>.

**Table 2:** An overview of optimizing case studies, where  $\#$  is the unroll factor (for loop unrolling) or the merge factor (for iteration merging), **OT** the time it takes to optimize, **VB** the original verification time (Verification Before) and **VA** the optimized verification time (Verification After). All times are in seconds.

Case	Loop unrolling				Iter. merging				Matrix lin.			Data pref.		
	#	OT	VB	VA	#	OT	VB	VA	OT	VB	VA	OT	VB	VA
BubbleSort [47]	1	0.101	25.4	27.3	4	0.170	29.8	34.1	N/A	N/A	N/A	N/A	N/A	N/A
InsertionSort [47]	1	0.134	25.6	25.8	3	0.225	24.1	28.0	N/A	N/A	N/A	N/A	N/A	N/A
SelectionSort [47]	1	0.107	23.5	25.7	2	0.592	22.8	27.7	N/A	N/A	N/A	N/A	N/A	N/A
TimSort [47]	2	0.216	29.3	38.5	3	0.182	29.1	37.9	N/A	N/A	N/A	N/A	N/A	N/A
Blelloch [49]	1	0.129	50.7	57.5	3	0.355	51.0	64.0	N/A	N/A	N/A	N/A	N/A	N/A
Kogge-Stone [49]	1	0.238	23.0	25.6	2	0.082	21.8	25.6	N/A	N/A	N/A	0.103	23.0	23.0
TicTacToe [55]	3	0.106	19.8	21.0	2	0.076	17.3	19.6	N/A	N/A	N/A	N/A	N/A	N/A
VerifyThis [26]	1	0.144	26.2	28.7	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Transpose [46]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.022	16.0	16.0	N/A	N/A	N/A

examples. Table 1 summarizes the best and worst optimization times for the six optimizations (as reported by ALPINIST). To investigate the impact on the verification time, the table also shows the (best and worst) verification times of the original and optimized programs (as reported by VerCors). The table shows the minimum, maximum, average and median times of all examples. It can be observed that ALPINIST takes insignificant time to apply each optimization to all the examples. Moreover, the verification time after optimizing generally increases. For loop unrolling, tiling and iteration merging, the verification time increases. This can be attributed to the additional code that is generated. For kernel fusion, the verification time decreases. This is due to verifying fewer kernels. For matrix linearization and data prefetching, the verification time slightly increases. This can be attributed to the linear expressions in matrix linearization and the extra statements to read from/write to the registers in data prefetching.

**Q3** To investigate the usability of ALPINIST on real-world examples, we successfully applied it on the third category with the complex case studies. Table 2 shows the optimization and verification times of applying loop unrolling, iteration merging, matrix linearization and data prefetching to these case studies. Note that in the case studies only these four optimizations could be applied. In the table, N/A indicates that the optimization is not applicable to the example.

## 6 Related Work

To the best of our knowledge, this is the first paper to showcase a tool that implements annotation-aware transformations. We categorize the related work into three parts, covering both tools and optimizations.

*Automatic Optimizations without Correctness.* There is a large body of related work, see e.g., [1, 3, 18, 24, 27, 45, 59, 60, 64–66], that shows the impact of automated optimizations on GPU programs, but does not consider *correctness*, or the preservation of it. Our tool can potentially complement these approaches by preserving the provability of the optimized programs, and it can exploit annotations to further automate optimisation.

*Correctness Proofs for Transformations.* Another body of related work focuses on different approaches to preserve provability not specific to GPU programs. COMPCert [29, 30] is a formally verified C compiler which preserves semantic equivalence of the source and compiled program, by proving correctness of each transformation in the compilation process. De Putter and Wijs [43] prove the preservation of functional properties over transformations on models of concurrent systems. They prove preservation of model-independent properties. This approach differs from ours as they work on models instead of concrete programs.

*Compiler Optimization Correctness.* Finally, there is related work that focusses on the compilation of sequential programs, performing transformations from high-level source code to lower-level machine code while preserving the semantics. These approaches neither consider parallelization, nor target different architectures. In GPU programming, the optimizations often need to be applied manually rather than during the compilation process.

Namjoshi and Xu [40] use a proof checker to show equivalence between an original WebAssembly program and optimized program. An equivalence proof is generated based on the transformations. Namjoshi and Singhanian [39] created a semi-automatic loop optimizer with user-directives. The loops are verified during compilation. For each transformation, semantics are defined to guarantee semantical equivalence to the original program. Namjoshi and Pavlinovic [38] focus on recovering from precision loss due to semantics-preserving program transformations and propose systematic approaches to simplify analysis of the transformed program. Finally Gjomemo et al. [19] help compiler optimizations by supplying high-level information gathered by external static analysis (e.g., Frama-C). This information is used by the compiler for better reasoning.

## 7 Conclusion

In this paper, we presented ALPINIST, the annotation-aware GPU program optimizer. Given an unoptimized, annotated GPU program, we showed how ALPINIST transforms both the code and the annotations, with the goal to preserve the provability of the optimized GPU program. ALPINIST supports loop unrolling, tiling, kernel fusion, iteration merging, matrix linearization and data prefetching, of which the first three are discussed in detail. We discussed the design and implementation of ALPINIST, and we validated it by verifying a set of examples and reverifying their optimized counterparts.

For future work, there are other optimizations that could be supported, such as data prefetching for all memory patterns as mentioned by Ayers et al. [3]. Another open question is if and how this approach can be used in program compilation. We also plan to extend this approach to preserve the provability of transpiled code, e.g., CUDA to OpenCL conversions. Moreover, we plan to investigate how ALPINIST can be combined with techniques such as *autotuning* that automatically detect the potential for applying specific optimizations and identify optimal parameter configurations [2, 61].



## References

1. Ashari, A., Tatikonda, S., Boehm, M., Reinwald, B., Campbell, K., Keenleyside, J., Sadayappan, P.: On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices* **50**(8), 173–182 (2015)
2. Ashouri, A., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys* **51**(5), 96:1–96:42 (2018)
3. Ayers, G., Litz, H., Kozyrakis, C., Ranganathan, P.: Classifying memory access patterns for prefetching. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. pp. 513–526 (2020)
4. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. Tech. rep., Citeseer (2008)
5. Berdine, J., Calcagno, C., O’Hearn, P.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W. (eds.) *FMCO*. LNCS, vol. 4111, pp. 115–137. Springer (2005)
6. Bertolli, C., Betts, A., Mudalige, G., Giles, M., Kelly, P.: Design and Performance of the OP2 Library for Unstructured Mesh Applications. In: *Proceedings of the 1st Workshop on Grids, Clouds and P2P Programming (CGWS)*. *Lecture Notes in Computer Science*, vol. 7155, pp. 191–200. Springer (2011). [https://doi.org/10.1007/978-3-642-29737-3\\_22](https://doi.org/10.1007/978-3-642-29737-3_22)
7. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: *OOPSLA*. pp. 113–132. ACM (2012)
8. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors Tool Set: Verification of Parallel and Concurrent Software. In: *iFM*. LNCS, vol. 10510, pp. 102 – 110. Springer (2017)
9. Blom, S., Huisman, M., Mihelčić, M.: Specification and Verification of GPGPU programs. *Science of Computer Programming* **95**, 376–388 (2014)
10. Bornat, R., Calcagno, C., O’Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. pp. 259–270 (2005)
11. Boyland, J.: Checking Interference with Fractional Permissions. In: *SAS*. LNCS, vol. 2694, pp. 55–72. Springer (2003)
12. Catanzaro, B., Keller, A., Garland, M.: A decomposition for in-place matrix transposition. *ACM SIGPLAN Notices* **49**(8), 193–206 (2014)
13. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of OpenCL code. In: *Haifa Verification Conference*. pp. 203–218. Springer (2011)
14. Şakar, O., Safari, M., Huisman, M., Wijs, A.: The repository for the examples used in ALPINIST, <https://github.com/OmerSakar/Alpinist-Examples.git>
15. Şakar, O., Safari, M., Huisman, M., Wijs, A.: The repository for the implementations of ALPINIST, <https://github.com/utwente-fmt/vercors/tree/gpgpu-optimizations/src/main/java/vct/col/rewrite/gpgpuoptimizations>
16. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.: Swarm Model Checking on the GPU. *International Journal on Software Tools for Technology Transfer* **22**, 583–599 (2020). <https://doi.org/10.1007/s10009-020-00576-x>
17. Dross, C., Furia, C.A., Huisman, M., Monahan, R., Müller, P.: Verifythis 2019: a program verification competition. *International Journal on Software Tools for Technology Transfer* pp. 1–11 (2021)

18. Filipovič, J., Madzin, M., Fousek, J., Matyska, L.: Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* **71**(10), 3934–3957 (2015)
19. Gjomemo, R., Namjoshi, K.S., Phung, P.H., Venkatakrishnan, V., Zuck, L.D.: From verification to optimizations. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. pp. 300–317. Springer (2015)
20. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a High-Level Language Targeted to GPU Codes. In: *Proc. 2012 Innovative Parallel Computing (InPar)*. pp. 1–10. IEEE (2012). <https://doi.org/10.1109/InPar.2012.6339595>
21. van den Haak, L., Wijs, A., M.G.J. van den Brand, Huisman, M.: Formal Methods for GPGPU Programming: Is The Demand Met? In: *Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020)*. *Lecture Notes in Computer Science*, vol. 12546, pp. 160–177. Springer (2020). [https://doi.org/10.1007/978-3-030-63461-2\\_9](https://doi.org/10.1007/978-3-030-63461-2_9)
22. Hamers, R., Jongmans, S.S.: Safe sessions of channel actions in Clojure: a tour of the discourje project. In: *International Symposium on Leveraging Applications of Formal Methods*. pp. 489–508. Springer (2020)
23. Herrmann, F., Silberholz, J., Tiglio, M.: Black Hole Simulations with CUDA. In: *GPU Computing Gems Emerald Edition*, chap. 8, pp. 103–111. Morgan Kaufmann (2011)
24. Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., Sadayappan, P.: Adaptive sparse tiling for sparse matrix multiplication. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. pp. 300–314 (2019)
25. Huisman, M., Blom, S., Darabi, S., Safari, M.: Program correctness by transformation. In: *8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. LNCS, vol. 11244. Springer (2018)
26. Huisman, M., Joosten, S.: A solution to VerifyThis 2019 challenge 1, <https://github.com/utwente-fmt/vercors/blob/97c49d6dc1097ded47a5ed53143695ace6904865/examples/verifythis/2019/challenge1.pv1>
27. Konstantinidis, A., Kelly, P.H., Ramanujam, J., Sadayappan, P.: Parametric GPU code generation for affine loop programs. In: *International Workshop on Languages and Compilers for Parallel Computing*. pp. 136–151. Springer (2013)
28. Le, Q., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Ng, A.: On Optimization Methods for Deep Learning. In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*. pp. 265–272. Omnipress (2011)
29. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 42–54 (2006)
30. Leroy, X.: A formally verified compiler back-end. *Journal of Automated Reasoning* **43**(4), 363–446 (2009)
31. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: *SIGSOFT FSE 2010, Santa Fe, NM, USA*. pp. 187–196. ACM (2010)
32. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: *ACM SIGPLAN Notices*. vol. 47, pp. 215–224. ACM (2012)
33. Lindholm, L., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* **28**(2), 39–55 (2008). <https://doi.org/10.1109/MM.2008.31>

34. Liu, X., Tan, S., Wang, H.: Parallel Statistical Analysis of Analog Circuits by GPU-Accelerated Graph-Based Approach. In: Proceedings of the 2012 Conference and Exhibition on Design, Automation & Test in Europe (DATE). pp. 852–857. IEEE Computer Society (2012). <https://doi.org/10.1109/DATE.2012.6176615>
35. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
36. Müller, P., Schwerhoff, M., Summers, A.: Viper - a verification infrastructure for permission-based reasoning. In: VMCAI (2016)
37. Murthy, G.S., Ravishankar, M., Baskaran, M.M., Sadayappan, P.: Optimal loop unrolling for GPGPU programs. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). pp. 1–11. IEEE (2010)
38. Namjoshi, K.S., Pavlinovic, Z.: The impact of program transformations on static program analysis. In: International Static Analysis Symposium. pp. 306–325. Springer (2018)
39. Namjoshi, K.S., Singhanian, N.: Loopy: Programmable and formally verified loop transformations. In: International Static Analysis Symposium. pp. 383–402. Springer (2016)
40. Namjoshi, K.S., Xue, A.: A Self-certifying Compilation Framework for WebAssembly. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 127–148. Springer (2021)
41. The OpenCL 1.2 specification (2011)
42. Osama, M., Wijs, A., Biere, A.: SAT Solving with GPU Accelerated Inprocessing. In: Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Computer Science, vol. 12651, pp. 133–151. Springer (2021). [https://doi.org/10.1007/978-3-030-72016-2\\_8](https://doi.org/10.1007/978-3-030-72016-2_8)
43. de Putter, S., Wijs, A.: Verifying a verifier: on the formal correctness of an LTS transformation verification technique. In: International Conference on Fundamental Approaches to Software Engineering. pp. 383–400. Springer (2016)
44. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* **48**(6), 519–530 (2013)
45. Rocha, R.C., Pereira, A.D., Ramos, L., Góes, L.F.: Toast: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience* **29**(8), e4053 (2017)
46. Safari, M., Huisman, M.: Formal verification of parallel stream compaction and summed-area table algorithms. In: International Colloquium on Theoretical Aspects of Computing. pp. 181–199. Springer (2020)
47. Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: International Conference on Integrated Formal Methods. pp. 257–275. Springer (2020)
48. Safari, M., Oortwijn, W., Huisman, M.: Automated verification of the parallel Bellman–Ford algorithm. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) *Static Analysis*. pp. 346–358. Springer International Publishing, Cham (2021)
49. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: *NASA Formal Methods Symposium*. pp. 170–186. Springer (2020)
50. Şakar, O.: Extending support for axiomatic data types in vercors (April 2020), <http://essay.utwente.nl/80892/>
51. Shimobaba, T., Ito, T., Masuda, N., Ichihashi, Y., Takada, N.: Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL. *Optics express* **18**(10), 9955–9960 (2010)

52. Sundfeld, D., Havgaard, J.H., Gorodkin, J., De Melo, A.C.: CUDA-Sankoff: using GPU to accelerate the pairwise structural RNA alignment. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 295–302. IEEE (2017)
53. The CUDA team: Documentation of the CUDA unroll pragma (Accessed Oct 6, 2021), <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#pragma-unroll>
54. The Halide team: Documentation of the Halide unroll function (Accessed Oct 6, 2021), [https://halide-lang.org/docs/class\\_halide\\_1\\_1\\_func.html#a05935caceb6efb8badd85f306dd33034](https://halide-lang.org/docs/class_halide_1_1_func.html#a05935caceb6efb8badd85f306dd33034)
55. The verification of tictactoe program, <https://github.com/utwente-fmt/vercors/blob/0a2fdc24419466c2d3b7a853a2908c37e7a8daa7/examples/session-generate/MatrixGrid.pvl>
56. Unkule, S., Shaltz, C., Qasem, A.: Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In: International Conference on Compiler Construction. pp. 21–40. Springer (2012)
57. Van Werkhoven, B., Maassen, J., Bal, H.E., Seinstra, F.J.: Optimizing convolution operations on GPUs using adaptive tiling. *Future Generation Computer Systems* **30**, 14–26 (2014)
58. Viper project website: (2016), <http://www.pm.inf.ethz.ch/research/viper>, <http://www.pm.inf.ethz.ch/research/viper>
59. Wahib, M., Maruyama, N.: Scalable kernel fusion for memory-bound GPU applications. In: SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 191–202. IEEE (2014)
60. Wang, G., Lin, Y., Yi, W.: Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In: 2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing. pp. 344–350. IEEE (2010)
61. Werkhoven, B.v.: Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* **90**, 347–358 (2019)
62. Wienke, S., Springer, P., Terboven, C., Mey, D.: OpenACC - First Experiences with Real-World Applications. In: Proceedings of the 18th European Conference on Parallel and Distributed Computing (EuroPar). Lecture Notes in Computer Science, vol. 7484, pp. 859–870. Springer (2012). [https://doi.org/10.1007/978-3-642-32820-6\\_85](https://doi.org/10.1007/978-3-642-32820-6_85)
63. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. In: Proceedings of the 21st International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 9995, pp. 694–701. Springer (2016). [https://doi.org/10.1007/978-3-319-48989-6\\_42](https://doi.org/10.1007/978-3-319-48989-6_42)
64. Wu, H., Damos, G., Wang, J., Cadambi, S., Yalamanchili, S., Chakradhar, S.: Optimizing data warehousing applications for GPUs using kernel fusion/fission. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. pp. 2433–2442. IEEE (2012)
65. Xu, C., Kirk, S.R., Jenkins, S.: Tiling for performance tuning on different models of GPUs. In: 2009 Second International Symposium on Information Science and Engineering. pp. 500–504. IEEE (2009)
66. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. *ACM Sigplan Notices* **45**(6), 86–97 (2010)