Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs

Simon Garcia De Gonzalo
CS and Coordinated Science Lab
UIUC
grcdgnz2@illinois.edu

Sitao Huang
ECE and Coordinated Science Lab
UIUC
shuang91@illinois.edu

Juan Gómez-Luna
Computer Science
ETH Zurich
juang@ethz.ch

Simon Hammond
Scalable Computer Architecture
Sandia National Laboratories
sdhammo@sandia.gov

Onur Mutlu
Computer Science
ETH Zurich
omutlu@ethz.ch

Wen-mei Hwu
ECE and Coordinated Science Lab
UIUC
w-hwu@illinois.edu

***Abstract*—Since the advent of GPU computing, GPU hardware has evolved at a fast pace. Since application performance heavily depends on the latest hardware improvements, performance portability is extremely challenging for GPU application library developers. Portability becomes even more difficult when new low-level instructions are added to the ISA (e.g., warp shuffle instructions) or the microarchitectural support for existing instructions is improved (e.g., atomic instructions). Library developers, besides re-tuning the code for new hardware features, deal with the performance portability issue by hand-writing multiple algorithm versions that leverage different instruction sets and microarchitectures. High-level programming frameworks and Domain Specific Languages (DSLs) do not typically support low-level instructions (e.g., warp shuffle and atomic instructions), so it is painful or even impossible for these programming systems to take advantage of the latest architectural improvements.**

In this work, we design a new set of high-level APIs and qualifiers, as well as specialized Abstract Syntax Tree (AST) transformations for high-level programming languages and DSLs. Our transformations enable warp shuffle instructions and atomic instructions (on global and shared memories) to be easily generated. We show a practical implementation of these transformations by building on Tangram, a high-level kernel synthesis framework. Using our new language and compiler extensions, we implement *parallel reduction*, a fundamental building block used in a wide range of algorithms. Parallel reduction is representative of the performance portability challenge, as its performance heavily depends on the latest hardware improvements. We compare our synthesized parallel reduction to another high-level programming framework and a hand-written high-performance library across three generations of GPU architectures, and show up to $7.8\times$ speedup ($2\times$ on average) over hand-written code.

I. Introduction

The current landscape of High Performance Computing (HPC) is heavily dominated by Graphics Processing Units (GPU) that are used as accelerators. As of December 2018, half of the top 10 most powerful supercomputers and 7 of the top 10 most energy-efficient supercomputers deploy GPUs [1], [2]. All leading cloud service providers offer GPU-based solutions [3]. In order to take full advantage of these GPU-based systems, many application developers need to become adept at GPU computing Application Programming Interfaces (APIs), such as NVIDIA's CUDA [4] or OpenCL [5]. Those who are not experts largely rely on carefully-crafted libraries such as Thrust [6] and CUB [7], higher level programming frameworks such as Tangram [8] or Kokkos [9], or Domain Specific Languages (DSLs) such as Halide [10], which abstract away the complexity of GPU software development.

Regardless of who generates the GPU code (e.g., an application programmer, a library developer, a programming framework, or a high-level compiler), there are performance portability challenges that should be tackled. Different GPU architectures deploy different implementations of certain instructions, such as atomic instructions, or incorporate new low-level primitives to an evolving Instruction Set Architecture (ISA). These differences offer new means to optimize algorithms on each architecture [11], [8]. Thus, rather than developing a single implementation of an algorithm, generating code specifically optimized for each hardware generation provides higher performance.

Parallel reduction, which is a fundamental building block in widely-used algorithms such as Histogram [12], [13] and Scan [14], is a computational pattern that is representative of the performance portability challenge in GPUs. Its performance heavily depends on hand-written code that takes advantage of the latest hardware improvements [7]. Library developers have to deal with this portability challenge by constantly adapting and upgrading their code to leverage new architectural features, while maintaining backward compatibility by keeping previous implementations available to be used on older generations of GPUs. Similarly, high-level programming frameworks, such as Kokkos [9], deal with performance portability of common computational patterns (e.g., reduction) by using in-house or third-party libraries with multiple hand-

978-1-7281-1436-1/19/$31.00 © 2019 IEEE

CGO 2019, Washington, DC, USA
Research Papers

written code versions for different GPU architectures. DSLs such as Halide [10] prevent programmers from having to re-write their code by providing API abstractions of low-level GPU instructions, such as warp shuffle instructions. However, Halide does not expose atomic instructions on global or shared memory, which are useful for optimization of common computational patterns. A state-of-the-art high-level programming framework, Tangram [8], provides composable code blocks that can be synthesized to provide different GPU architectures with algorithmic choices, but lacks support for atomic instructions and warp shuffle instructions.

In order to provide library developers and users of high-level programming frameworks and DSLs with an efficient way to optimize their code for different architectures, we develop a new set of high-level APIs and programming constructs that expose low-level primitives, such as warp shuffle and atomic instructions, to programmers. By building our techniques on top of the Tangram programming framework, we make the following contributions:

- We augment the code optimization choices with different types of atomic instructions on global memory by introducing a new set of APIs and corresponding Abstract Syntax Tree (AST) transformations. The new APIs extend Tangram's *Map* primitive.
- We propose an AST pass to automatically identify warp-shuffle instructions, without requiring manual source code modification. We implement our AST pass in Tangram to enable a larger number of code variants to be synthesized.
- We make atomic instructions on shared (scratchpad) memory available by introducing a new set of data array qualifiers that work in conjunction with existing memory directives (e.g., Tangram's *__shared* qualifier). By doing so, we enable code variants that take advantage of the improved hardware support for atomic instructions on shared memory in newer GPU architectures.
- We develop high-level code for parallel reduction and compare the performance of our synthesized code, which leverages warp shuffle and atomic instructions, against a high-level performance portability framework, and hand-written library code. Our results demonstrate significant performance across three generations of GPU architectures, with $2\times$ average speedup (up to $7.8\times$) over hand-written code.

II. Background and Motivation

This section gives an overview of low-level GPU instructions (namely, warp shuffle and atomic instructions) and high-level programming frameworks with a special attention to Tangram [15], [8].

A. GPU ISA and Microarchitectural Support

The NVIDIA CUDA ISA [16] has continuously evolved with every GPU generation, since the launch of the first CUDA-capable GPU architecture. With the goal of improving programmability and performance, new instructions have been added [17], [18] and the microarchitectural support improved. Warp shuffle instructions [18] and atomic instructions [17] are good examples of CUDA ISA's evolution. Supporting these instructions in high-level programming languages makes performance portability (i.e., the ability to achieve high performance on existing and future architectures without software re-development) more feasible and keeps the programming effort low.

1) Warp Shuffle Instructions: NVIDIA's Kepler architecture [19] introduced low-level primitives called warp shuffle instructions. These instructions allow threads in the same warp to exchange private register values via execution unit data paths during SIMD execution without going through the shared memory. Shuffle instructions 1) have shorter latency than shared memory load instructions, and 2) reduce shared memory footprint. They have different modes, corresponding to different types of shuffling: shift up or down exchange (*__shfl_up_sync()* or *__shfl_down_sync()*), butterfly exchange (*__shfl_xor_sync()*), and indexed (any to any, *__shfl_sync()*). They can also operate on subwarps [20], where the shuffling is applied independently on each subwarp.

2) Atomic Instructions: CUDA has offered atomic instructions on global and shared memories since early in its conception. These include arithmetic operations, such as *atomicAdd()*, *atomicSub()*, *atomicMin()*, *atomicMax()* and logicals. CUDA atomic operations have evolved over different generations of NVIDIA GPU architectures. In the first generations, they were so inefficient that GPU algorithms had to resort to complex data manipulation to avoid or completely remove the need for atomic operations [21], [22]. From the Fermi [23] to the Kepler architecture [19], the addition of buffers in the L2 atomic units sped up global memory atomics, opening the door for algorithms that utilized atomic operations for programmability, performance or both.

Before the Maxwell architecture [24], atomic instructions on shared memory were implemented in software using a *lock-update-unlock* mechanism that became expensive under high load for highly-contested shared memory locations [13]. Kepler library developers used warp shuffle instructions to avoid the use of atomic instructions on shared memory whenever possible [25]. The Maxwell architecture improved the atomic instructions on shared memory by giving them native microarchitectural support. After that, Pascal [26] added *scope* to the atomic instructions. *System* scope allows for atomic visibility between CPUs and peer GPUs (e.g., *atomicAdd_system()*). *Device* scope, which is the default scope, enforces atomic visibility within a single GPU device. *Block* scope implies atomic visibility only within a CUDA block (e.g., *atomicAdd_block()*). Knowing when and where to use different scopes can be an error-prone process, but the scopes can improve both the algorithmic implementations and execution efficiency.

In a span of four years, three GPU generations, the hardware for atomic instructions on global/shared memory improved significantly and atomic scopes were introduced. Both facts made atomic instructions much more appealing for the imple-

mentation of GPU algorithms.

However, taking advantage of the improved atomic instructions requires application and library developers to re-write code. Otherwise, they might lose opportunities for improved performance. At the same time, many developers are required to write applications that are backward-compatible with previous GPU architectures. As a consequence, both leveraging the latest low-level instructions (e.g., warp shuffle and atomic instructions) and guaranteeing backward-compatibility, to achieve performance portability across architecture generations, need huge programming effort. High-level programming frameworks, such as Tangram [8], can significantly alleviate the programming effort by providing backward-compatibility and allowing programmers to use the latest low-level instructions. In this work, we enable the use of low-level instructions (namely, warp shuffle instructions and atomic instructions on global/shared memory) through new user-level APIs and new AST code analysis and transformation passes for Tangram, as an example of high-level programming framework.

B. High-level Programming Frameworks

There has been increasing interest in high-level abstractions that can be used to minimize the amount of code re-writing and re-tuning that is required for high-performance execution [8], [15], [9], [27], [10], [28], [29], [30], [31]. In general, all high-level frameworks or languages can be described in terms of primitives to abstract 1) computation, and 2) data manipulation and distribution. These primitives enable application and library developers to quickly try different algorithms or code optimizations. The work described in this paper is implemented on top of Tangram [8], but other frameworks or high-level languages could also benefit from our contributions.

1) Tangram: The Tangram programming model [8], [15] is based on the idea of expressing architecture-neutral computations through interchangeable and composable building blocks called *spectrums* and *codelets*. A *spectrum* represents a unique computation with a defined set of inputs, outputs, and side effects. A *codelet* represents a specific algorithmic implementation of a *spectrum*. A *spectrum* can have many *codelets* that implement it. For example, Figure 1 shows three *codelets* with different optimization techniques for the `sum` reduction *spectrum*.

To build *codelets*, Tangram relies on built-in primitives that explicitly express data parallelism (e.g., *Map*), data partitioning (e.g., *Partition*), access patterns (e.g., *Sequence*), data containers (e.g., *Array*), and multiple thread cooperation (e.g., *Vector*). Tangram adds a set of array qualifiers that helps dictate data placement (e.g., *__shared*) and parameters that can be tuned at compile time or run time (e.g., *__tunable*).

Codelets can be classified as *atomic autonomous*, *compound* or *atomic cooperative*. Figure 1(a) shows an *atomic autonomous codelet* that computes the `sum` reduction of elements of an array sequentially. This *codelet* is 1) *atomic* because it cannot be divided into other *codelets*, and 2) *autonomous* because it represents the computation of one single thread [8].

Figure 1(b) depicts a *compound codelet* that is expressed with *Sequence*, *Map*, and *Partition* primitives. It is *compound* because it can be decomposed into *atomic autonomous codelets* (e.g., the *atomic* `sum` reduction in Figure 1(b)). With the definition of the *Sequence* primitives, a developer can describe a *tiled* or *strided* access pattern, as shown at the bottom of Figure 1(b). A *Partition(c, n, start, inc, end)* primitive returns n sub-containers c_i of c where c_i goes from *start[i]* to *end[i]*, with increment *inc[i]*. The number of partitions, `p` in Figure 1(b), is declared as *__tunable* (line 3). A *Map(f, c)* primitive applies a function f to each element of data container c. Thus, *Map(f, partition(c, n, start, inc, end))*, applies a function f to each sub-container of *partition*. In Figure 1(b), f is the *atomic codelet* `sum`.

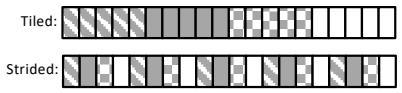
Figure 1(c) shows an *atomic cooperative codelet*, which

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   unsigned len = in.Size();
4   int accum = 0;
5   for(unsigned i=0; i < len; in.Stride()) {
6     accum += in[i]; // Sequential sum
7   }
8   return accum;
9  }
```

(a) Atomic Autonomous Codelet.

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   __tunable unsigned p; // Number of partitions
4   unsigned len = in.Size();
5   unsigned tile = (len+p-1)/p;
6   Sequence start(…); // Sequence for access pattern
7   Sequence end(…);   // E.g., Tiled or Strided
8   Sequence inc(…)
    // Apply sum to p sub-containers of in
9   Map map(sum, partition(in, p, start, inc, end));
10  map.atomicAdd(); // Atomic API (Section III.A)
11  return sum(map);
12 }
```

Access Patterns specified by Sequence API:



(b) Compound Codelet.

```
1  __codelet
2  int sum(const Array<1,int> in) {
3   Vector vthread(); // Vector declaration
4   __shared int partial[vthread.MaxSize()]; // Shared arrays
5   __shared int tmp[in.Size()];
6     int val = 0;
    // Input read
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8     tmp[vthread.ThreadId()] = val;
    // Tree-based summation by each vector
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ?
11                          tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13  }
    // Check if partial sums can be summed by one single vector
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
      // Write partial sum in shared array
15    if(vthread.LaneId() == 0)
16      partial[vthread.VectorId()] = val;
      // Final sum by Vector ID = 0
17    if(vthread.VectorId() == 0){
18      val = (vthread.ThreadId() <= (in.Size() / vthread.MaxSize())) ?
19                          partial[vthread.LaneId()] : 0;
        // Tree-based summation of partial sums
20      for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
21        val += (vthread.LaneId() + offset < vthread.Size()) ?
22                          partial[vthread.ThreadId()+offset] : 0;
23        partial[vthread.ThreadId()] = val;
24      }
25    }
26  }
27  return val;
28 }
```

(c) Atomic Cooperative Codelet.

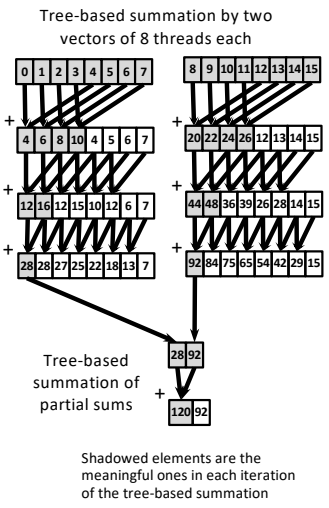
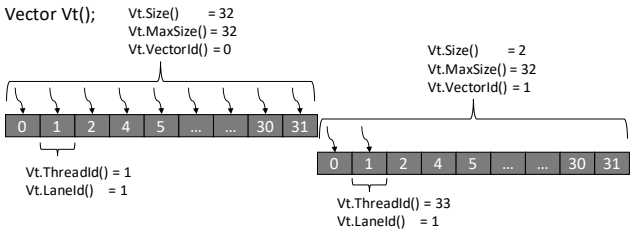


Fig. 1. Codelet Examples for the `sum` Reduction Spectrum.



Function	Description	CUDA equivalent
Size()	Number of threads in Vector	warpSize
MaxSize()	Maximum number of threads in a Vector	32
ThreadId()	Global position of the thread with respect to all threads	threadIdx.x
LaneId()	Local position of thread within one Vector	threadIdx.x % warpSize
VectorId()	Global position of the vector with respect to all vectors	threadIdx.x / warpSize

Fig. 2. Vector Primitive API. One CUDA Block with 34 Threads is Broken Down into Two Warps with 32 and 2 Active Threads Respectively.

allows for multiple parallel threads of execution to coordinate work (i.e., to *cooperate*). The algorithm is depicted pictorially to the right of the same *codelet*. This particular *codelet* performs tree-based summation on arrays marked as *__shared*. The *codelet* uses the *Vector* primitive. The *Vector* primitive represents a collection of threads performing SIMD/SIMT parallel execution. The *Vector* primitive contains member functions (*MaxSize()*, *Size()*, *ThreadId()*, and *LaneId()*) to obtain the architecture-specific properties of the group of threads and the ID for a lane and/or thread. Figure 2 depicts the *Vector* primitive API and its member functions with their CUDA equivalents.

In this work we need to augment only the *Map* and *Vector* primitives to support warp shuffle and atomic instructions, and add an additional qualifier to the Tangram language for atomic instructions on shared memory.

2) Tangram's Code Generation: We use the *codelet* in Figure 1(b) to illustrate how Tangram generates GPU code. The code in Listing 1 is one of the multiple synthesis outcomes of Tangram. Tangram can synthesize a *Map(f, partition(c, n, start, inc, end))* primitive at the GPU's grid level. In that case, *Map(...)* translates to a GPU kernel launch where each sub-container is assigned to a different CUDA block, as shown in lines 37 to 40 of Listing 1. The dimension of the GPU grid (i.e., the number of blocks) is the number of partitions. This number is `p` in Figure 1(b) (line 3), which translates to a template parameter as shown in lines 27 and 28 in Listing 1. Another possibility is that Tangram synthesizes *Map(f, partition(c, n, start, inc, end))* at the GPU block level. In that case, it translates to a device function where each sub-container is processed by a different thread (see line 19 of Listing 1 for an example). The number of partitions is the number of threads per block. Thread-coarsening optimizations [32] are possible with this approach. *Map(...)*, therefore, has flexible semantics depending on the level of the GPU software hierarchy.

Because *Map* applies a function to sub-containers of the original array, the partial results of each partition have to be stored in memory. *Map* automatically allocates global memory through the CUDA *cudaMalloc* API, if it is at the grid level, or lets one thread per block allocate global memory using the C++ *new* allocator and share that address with the rest of the threads, if it is at the block level. Lines 33-35 and 15-17 of Listing 1 show examples of both types of memory allocations respectively. Each partial result is written to memory based on the *threadIdx.x* or *blockIdx.x* (lines 6 and 24 of Listing 1). The partial results will be input to another `sum` reduction spectrum call to aggregate them. Hence, Tangram generates a second kernel launch, at the grid level, or another device function call at the block level, not shown for brevity.

C. Goal

Our goal in this work is to alleviate the performance portability challenge across GPU architectures by enabling the use of low-level instructions (in particular, warp shuffle and atomic instructions) in high-level GPU programming frameworks. With this goal in mind, we design a new set of high-level APIs, array qualifiers, and AST transformations for high-level programming languages and DSLs, and demonstrate a practical implementation on Tangram. We show how Tangram leverages our techniques to augment the code optimization choices and synthesize performance-portable code for *parallel reduction* on GPUs.

III. Extending High-level Programming Frameworks for Low-level Instructions

This section explains our techniques to extend high-level programming frameworks with language and compiler extensions that enable the use of low-level instructions. In particular, we extend Tangram [8] to be able to synthesize code variants using GPU warp shuffle instructions and atomic instructions on global and shared memory. The new code variants increase the search space of Tangram, which can find the best performing code by using heuristics [8] or dynamic kernel selection at runtime [33].

A. Enabling Atomic Instructions on Global Memory

This section describes how to extend Tangram with atomic instructions on global memory to allow application and library developers to explicitly state that some partial results should be atomically accumulated on global memory. Using atomic instructions on global memory can lead to significant savings in terms of resource usage and executed instructions, which can potentially increase performance. For example, in parallel reduction, using atomic instructions on global memory dramatically reduces 1) the size of the arrays that are allocated in global memory for storing partial results, which increases the likelihood that these arrays fit in cache, and 2) the number of executed instructions to accumulate partial results, which likely reduces the number of execution cycles.

In Tangram, we add new API functions to the *Map* primitive that expose atomic instructions on global memory to

programmers. The new API functions include *atomicAdd()*, *atomicSub()*, *atomicMax()*, and *atomicMin()*. Figure 1(b) (line 10) shows the syntax of *atomicAdd()*. Parallel reduction can take advantage of different atomic instructions because different applications require different types of reductions (e.g., addition, subtraction, maximum, minimum).

Tangram can generate different atomic versions of the code for the new APIs and non-atomic versions. We illustrate the code generation with the *compound codelet* for the parallel reduction in Figure 1(b), where partial results are accumulated either with a non-atomic *spectrum* call (line 11) or with an atomic API (line 10). The non-atomic *spectrum* call and the atomic API are *mutually exclusive*: thus, Tangram will only use the first one for the non-atomic version and the second one for the atomic version. In order to disable one of them to generate the corresponding code version, Tangram implements a pre-processing step where an AST pass looks for *Map* primitives that use an atomic API (line 10 in Figure 1(b)). If such a *Map* primitive is an input to a *spectrum* call (line 11 in Figure 1(b)), the AST pass checks whether the *spectrum* call applies to the input the same computation as the atomic API. If so, the AST pass disables the *spectrum* call for the generation of the atomic version. If it is not the same computation, the AST pass does not disable the *spectrum* call. We can use similar pre-processing steps with AST passes to enable other advanced optimizations, such as loop unrolling [34]. We leave them for future work.

Listing 1 shows the non-atomic version of the code that Tangram generates for the *compound codelet* for parallel reduction in Figure 1(b). Listing 2 shows the version that uses atomic instructions on global memory. We highlight the main differences between both versions. First, lines 17 and 34-35 of both codes show memory allocations. The non-atomic version needs arrays of size `p` (i.e., the number of partitions in the *compound codelet*) for partial results. However, the atomic version allocates only a single variable, since an atomic operation sums all partial results into a single accumulator. Second, lines 6 and 24 show how partial results are handled. The non-atomic version stores partial results in arrays. They will be input to another *spectrum* call to accumulate them. For the atomic version, Tangram generates *atomicAdd_block()* for reduction at the block level, and *atomicAdd()* for reduction at the grid level.

B. Enabling Atomic Instructions on Shared Memory

In this section, we describe how we extend Tangram to expose atomic instructions on shared memory to application and library developers. With atomic instructions on shared memory, it is possible to perform reduction operations without requiring to allocate an array for partial results in shared memory. Instead, a single shared variable is the accumulator. The shared memory footprint becomes significantly smaller, which can lead to higher GPU occupancy (i.e., higher number of active threads) and, potentially, higher performance [20]. Atomic instructions on shared memory also allow developers to implement algorithms that require atomic updates on shared arrays (e.g., Histogram [12], [13]).

```
1  __inline__ __device__
2  void Reduce_Thread(int *Return,
3                     int *input_x,
4                     ...){
5    ...
6    Return[threadIdx.x] = ...;
7  }
8
9  __global__
10 void Reduce_Block(int *Return,
11                   int *input_x,
12                   ...){
13   int p = blockDim.x;
14   ...
15   __shared__ int *map_return;
16   if (threadIdx.x == 0)
17     map_return = new int[p];
18   __syncthreads();
19   Reduce_Thread(map_return,
20                 input_x + ,
21                 ...);
22   ...
23   if (threadIdx.x == 0)
24     Return[blockIdx.x] = ...;
25 }
26
27 template
28 <unsigned int TGM_TEMPLATE_0>
29 int Reduce_Grid(int *input_x,
30                 ...){
31   int p = TGM_TEMPLATE_0;
32   ...
33   int *map_return_block;
34   cudaMalloc(map_return_block,
35              (p)*sizeof(int));
36   ...
37   Reduce_Block<<<p,...>>
38                (map_return_block,
39                 input_x,
40                 ...);
41 }
```

Listing 1. Baseline Reduction. Highlighted Lines are the Differences with Listing 2.

```
__inline__ __device__
void Reduce_Thread(int *Return,
                   int *input_x,
                   ...){
  ...
  atomicAdd_block(Return, ...);
}

__global__
void Reduce_Block(int *Return,
                  int *input_x,
                  ...){
  int p = blockDim.x;
  ...
  __shared__ int *map_return;
  if (threadIdx.x == 0)
    map_return = new int[1];
  __syncthreads();
  Reduce_Thread(map_return,
                input_x + ,
                ...);
  ...
  if (threadIdx.x == 0)
    atomicAdd(Return, ...);
}

template
<unsigned int TGM_TEMPLATE_0>
int Reduce_Grid(int *input_x,
                ...){
  int p = TGM_TEMPLATE_0;
  ...
  int *map_return_block;
  cudaMalloc(map_return_block,
             sizeof(int));
  ...
  Reduce_Block<<<p,...>>
               (map_return_block,
                input_x,
                ...);
}
```

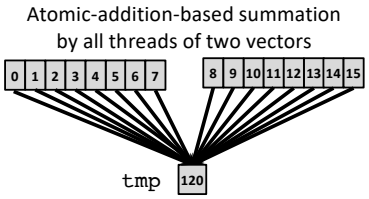
Listing 2. Reduction with Global Atomic Instructions.

In Tangram, we expose atomic instructions on shared memory by adding several qualifiers (namely, $\_\_atomicAdd$, $\_\_atomicSub$, $\_\_atomicMax$, $\_\_atomicMin$) that we use in conjunction with the *__shared* qualifier. We can use them to generate two new *cooperative codelets* for parallel reduction, which represent alternatives to the *cooperative codelet* in Figure 1(c). The *codelet* in Figure 3(a) uses a single *__shared* accumulator that is atomically updated ($\_\_atomicAdd$) by all threads of all vectors. This *codelet* reduces significantly the shared memory footprint and the number of executed instructions with respect to the *codelet* in Figure 1(c), but might suffer from high contention due to atomically updating the accumulator. The *codelet* in Figure 3(b) performs the reduction in two steps. First, each vector carries out tree-based summation. Second, the first thread of each vector updates the shared accumulator atomically. With this *codelet*, contention on the accumulator is low, while still reducing the shared memory footprint and the number of executed instructions with respect to the *codelet* in Figure 1(c).

We illustrate the code generation with the *cooperative codelet* in Figure 3(b). Listing 3 shows Tangram-synthesized code for this *cooperative codelet*. Tangram uses an AST pass that identifies *__shared* variables with atomic qualifiers. When this AST pass finds a write operation on an atomic shared variable, Tangram generates an atomic operation on shared memory.

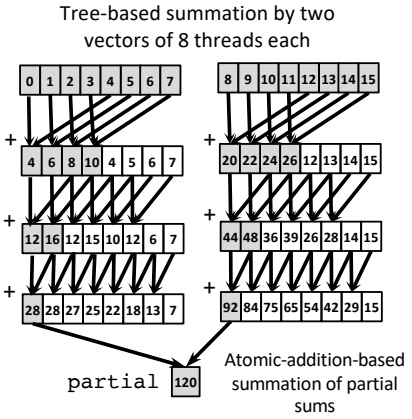
The *cooperative codelet* in Figure 3(b) declares a *__shared* atomic variable `partial` in line 4. First, the AST pass identi-

```
1  __codelet __coop __tag(shared_V1)
2  int sum(const Array<1,int> in) {
3   Vector vthread(); // Vector declaration
4   __shared _atomicAdd int tmp; // Shared variable to be atomically written
5   int val = 0;
    // Input read
6   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
7   tmp = val; // Execution of the atomic addition: all threads update tmp atomically
8   return tmp;
9  }
```



(a) Cooperative Codelet with a Single Accumulator (`tmp`) -- updated atomically by all threads of all vectors

```
1  __codelet __coop __tag(shared_V2)
2  int sum(const Array<1,int> in) {
3   Vector vthread(); // Vector declaration
4   __shared _atomicAdd int partial; // Shared variable to be atomically written
5   __shared int tmp[in.Size()]; // Shared array
6   int val = 0;
    // Input read
7   val = (vthread.ThreadId() < in.Size()) ? in[vthread.ThreadId()] : 0;
8    tmp[vthread.ThreadId()] = val;
    // Tree-based summation by each vector
9   for(int offset = vthread.MaxSize()/2; offset > 0; offset /= 2){
10    val += (vthread.LaneId() + offset < vthread.Size()) ? tmp[vthread.ThreadId()+offset] : 0;
12    tmp[vthread.ThreadId()] = val;
13   }
14  if(in.Size() != vthread.MaxSize() && in.Size()/vthread.MaxSize() > 0){
15    if(vthread.LaneId() == 0) // The first lane of each vector executes the next instruction
16      partial = val; // Execution of the atomic addition: all active lanes update tmp atomically
17    if(vthread.VectorId() == 0)
18      val = partial
19  }
20  return val;
21 }
```



(b) Cooperative Codelet with a Single Accumulator (`partial`) -- only updated by the first thread of each vector

Fig. 3. New Cooperative Codelets with Atomic Instructions on Shared Memory.

fies it, and Tangram generates the declaration and initialization of `partial` in lines 5 to 7 of Listing 3. One single thread per block initializes `partial`. Second, the AST pass finds the write operation in line 16 of Figure 3(b). As a result, Tangram generates the atomic instruction in line 27 of Listing 3.

```
1   __global__
2   void Reduce_block(int *Return, int *input_x, int SourceSize,
3                     int ObjectSize) {
4     unsigned int blockID = blockIdx.x;
5     __shared__ int partial;
6     if(threadIdx.x == 0)
7       partial = 0;
8     __syncthreads();
9     extern __shared__ int tmp[];
10    tmp[threadIdx.x] = 0;
11    __syncthreads();
12    int val = 0;
13    val = (((threadIdx.x < ObjectSize)) &&
14          ((blockIdx.x * blockDim.x + threadIdx.x) < SourceSize))
15          ? input_x[blockIdx.x * blockDim.x + threadIdx.x]
16          : 0;
17    tmp[threadIdx.x] = val;
18    for (int offset = (32 / 2); (offset > 0); offset /= 2) {
19      val += (((threadIdx.x % warpSize + offset) < warpSize))
20            ? tmp[(threadIdx.x + offset)]
21            : 0;
22      tmp[threadIdx.x] = val;
23      __syncthreads();
24    }
25    if (((ObjectSize != 32) && ((ObjectSize / 32) > 0))) {
26      if ((threadIdx.x % warpSize == 0)) {
27        atomicAdd(partial, val);
28        __syncthreads();
29      }
30      if ((threadIdx.x / warpSize == 0))
31        val = partial;
32    }
33    if (threadIdx.x == 0)
34      Return[blockID] = val;
35  }
```

Listing 3. Reduction Code with Atomic Instructions on Shared Memory for Figure 3(b). Highlighted Lines Declare, Initialize, and Atomically Update Shared Memory Arrays.

The *cooperative codelet* also declares the *__shared* array `tmp`, which is not atomic, in line 5 of Figure 3(b). `tmp` is used in the first step of the *cooperative codelet*, tree-based summation. Tangram declares `tmp` as *extern* (line 9 of Listing 3). This way, `tmp` can be dynamically allocated at kernel launch [20], since its size depends on the input size (`in.Size()` in Figure 3(b)). All threads cooperate in the initialization of `tmp` (line 10 of Listing 3).

C. Enabling Warp Shuffle Instructions

This section describes how we extend Tangram with an AST pass that identifies opportunities for code variants using warp shuffle instructions. Similar AST passes could identify other warp instructions. We leave them for future work.

Warp shuffle instructions perform fast data exchange across threads of the same vector, without using shared memory. The shared memory footprint becomes smaller, enabling higher GPU occupancy and higher performance. Specific read-write patterns benefit from warp shuffle instructions. An example is Kogge-Stone tree-based summation [21], which the *cooperative codelet* in Figure 1(c) implements.

In the pre-processing stage of Tangram, we include a new AST pass that detects opportunities for warp shuffle instructions. Figure 4 shows the algorithm of this AST pass. It looks for *forloop* nodes of the Abstract Syntax Tree with specific read-write patterns. First, upon reaching a *forloop* node, the algorithm checks in step (1) if the bounds of the *forloop* are determined by member functions of a *Vector* primitive (e.g., *MaxSize()* in Figure 4). Step (2) checks if the *forloop* iterator decreases by a constant every iteration. Next, the algorithm traverses the body of the *forloop*. Step (3) looks for a *__shared* array (e.g., `tmp` in Figure 4) whose contents are read and

accumulated in a local variable (e.g., `val`). Step (4) checks that the index of the shared array is a function of *ThreadId()* of the *Vector* primitive and the *forloop* iterator. Finally, steps (5), (6), and (7) check that the local accumulator is stored in the shared array at a position indexed by a function of *ThreadId()*. If the algorithm finishes, using a warp shuffle instruction is possible. The type of shuffling (e.g., shift up or down exchange) depends on how the *forloop* iterates. Tangram generates *__shfl_down()* if the *forloop* iterates in the negative direction of *Vector*, and *__shfl_up()* if the *forloop* iterates in the positive direction.

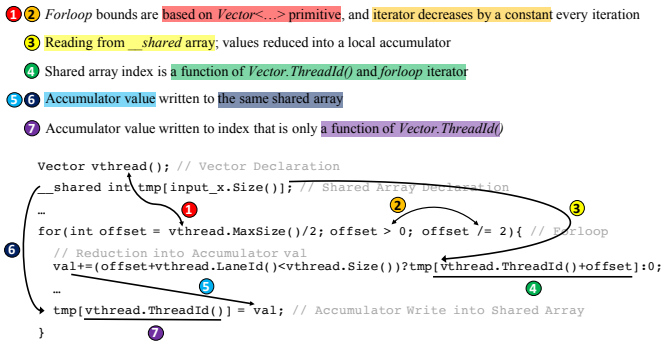


Fig. 4. Algorithm to Detect Opportunities for Warp Shuffle Instructions.

We show how Tangram translates the *cooperative codelet* in Figure 1(c) into the code variant in Listing 4, which uses warp shuffle instructions. In Figure 1(c), two *forloops* (lines 9-13 and lines 20-24) fulfill all conditions checked by the algorithm in Figure 4. First, from the body of the *forloop* in lines 9 to 13 of Figure 1(c), Tangram generates the warp shuffle instruction in line 15 of Listing 4. The AST pass disables array `tmp`, because its contents come directly from the input array. Thus, no shared memory is allocated for it. Second, Tangram replaces the body of the *forloop* in lines 20 to 24 of Figure 1(c) with another warp shuffle instruction (line 27 in Listing 4). In this case, the AST pass does not disable array `partial`, because there is a producer-consumer relation between the two *forloops*. The contents of `partial` come from local accumulator `val` – line 16 of Figure 1(c) translates to line 19 of Listing 4.

D. Generating Code Variants

As described in the previous sections, we add support for Tangram to generate multiple new code variants which can use atomic instructions on global or shared memory and warp shuffle instructions. New AST passes enable the new code variants. Figure 5 shows a visual representation of the pre-processing steps before the actual code generation. First, Tangram planner [8] generates the Abstract Syntax Tree (AST). Second, Tangram traverses the AST to apply general transformations and gather metadata for later transformations. Third, Tangram applies CUDA specific transformations. We include here the new AST passes for atomic instructions and warp shuffle instructions. When the AST passes encounter new code variants, Tangram records them for the actual CUDA code generation step. When there are no new variants, CUDA code generation takes place one last time, generating a plain version with no variants.

```
1  __global__
2  void Reduce_block(int *Return, int *input_x, int SourceSize,
3                    int ObjectSize) {
4    unsigned int blockID = blockIdx.x;
5    __shared__ int partial[32];
6    if(threadIdx.x < 32)
7      partial[threadIdx.x] = 0;
8    __syncthreads();
9    int val = 0;
10   val = (((threadIdx.x < ObjectSize)) &&
11            ((blockIdx.x * blockDim.x + threadIdx.x) < SourceSize))
12              ? input_x[blockIdx.x * blockDim.x + threadIdx.x]
13              : 0;
14   for (int offset = (32 / 2); (offset > 0); offset /= 2){
15     val += __shfl_down(val, offset, 32);
16   }
17   if (((ObjectSize != 32) && ((ObjectSize / 32) > 0))){
18     if ((threadIdx.x % warpSize == 0)) {
19       partial[threadIdx.x / warpSize] = val;
20     }
21     __syncthreads();
22     if ((threadIdx.x / warpSize == 0)) {
23       val = ((threadIdx.x <= ((ObjectSize / 32))))
24                   ? partial[threadIdx.x % warpSize]
25                   : 0;
26       for (int offset = (32 / 2); (offset > 0); offset /= 2){
27         val += __shfl_down(val, offset, 32);
28       }
29     }
30   }
31   if (threadIdx.x == 0)
32     Return[blockID] = val;
33
34 }
```

Listing 4. Reduction Code with Warp Shuffle Instructions for Figure 1(c). Hightlighted Lines are Warp Shuffle Instructions.

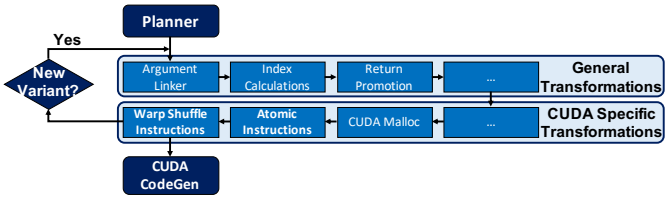


Fig. 5. Tangram's Pre-processing for Generation of Code Variants.

The code variants with atomic and warp shuffle instructions can be further extended for future work. For example, aggregate atomics [25] could be supported through the atomic APIs and qualifiers described in Sections III-A and III-B with new AST passes and transformations.

IV. Evaluation

A. Experimental Setup

Tangram generates CUDA code by using multiple Clang [35] AST traversals. The output CUDA code is then compiled using the NVIDIA *nvcc* compiler [20].

We run experiments on three GPUs from different NVIDIA architectures: Kepler K40c [19], Maxwell GTX980 [24], and Pascal P100 [26]. For these architectures, we use *nvcc* versions 8.0.44, 9.1.85, and 9.2.88 respectively. These GPU architectures represent different stages in the evolution of atomic instructions and warp shuffle instructions. Thus, they are good testbeds for testing Tangram's performance portability.

We compare Tangram-synthesized code versions to two state-of-the-art implementations of GPU-based parallel reduction: 1) NVIDIA's CUB 1.8.0 hand-written low-level library of cooperative primitives [7], and 2) Kokkos performance

portability programming model for HPC applications [9] using the GPU backend. We use input arrays of 32-bit single-precision elements. The size of the arrays is between 64 and 260M elements. Since we test on small and medium-size arrays, we also compare against CPU-based parallel reduction using the OpenMP 4.0 reduce pragma [36]. The OpenMP codes run on an IBM Minsky HPC system with two dual-socket 8-core 3.5GHz POWER8+ CPUs, using OpenMP 4.0 compiled with gcc 5.4.0.

B. Tangram Search Space

Tangram can generate multiple code versions by synthesizing different codelets at different levels of the GPU software hierarchy (i.e., grid, block, thread). The original Tangram framework [8], [15] is able to generate 10 unique versions of GPU parallel reduction by using the three *codelets* in Figure 1.

After enabling atomic and warp shuffle instructions, the total number of code versions of Tangram-synthesized GPU parallel reduction becomes 89. 10 of the new code versions use only atomic instructions on global memory, 38 more versions are possible by enabling atomic instructions on shared memory, and 31 more versions by employing warp shuffle instructions.

We prune the search space by removing code versions that consistently provide low performance in preliminary experiments on all GPU architectures. They are all code versions that require the launch of a second CUDA kernel for the reduction of partial per-block sums. Among them, we find the original 10 versions, 28 of the new versions with atomic instructions on shared memory, and 21 of the new versions with warp shuffle instructions. Thus, pruning the search space brings the total number of synthesized codes down to 30 versions, all of which use atomic instructions on global memory to reduce partial per-block sums.

In order to illustrate the composition of Tangram-synthesized versions, Figure 6 shows 16 of the final 30 versions. All of these 16 versions use *Global Atomic Tile Distribution* (i.e., a *compound codelet* with *tiled* access pattern and atomic instructions on global memory for partial results) at the grid level. At the block level, they use *compound codelets* (versions *a* to *k* in Figure 6), or *cooperative codelets* (versions *l* to *p*). Versions that use *compound codelets* at the block level (*a* to *k*) perform summation of partial per-thread results with *cooperative codelets* (e.g., Figure 1(c), Figure 3(a), and Figure 3(b)). As the next section shows, the best-performing 8 versions are included in Figure 6.

C. Results

This section shows the evaluation results for Tangram-synthesized code for GPU parallel reduction and in comparison to hand-written CUB library, Kokkos framework, and OpenMP CPU code. All Tangram code versions are tuned using *__tunable* parameters to determine optimal block and grid dimensions [8]. This is done with a simple script that runs all versions with different tuning parameters for the biggest problem size. It takes about 20 minutes.

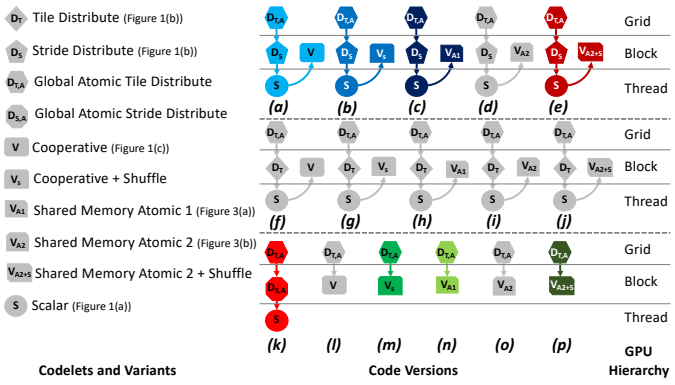


Fig. 6. 16 out of the Total 30 Tangram Code Versions Tested. Arrows Pointing up Show the Need to Compute Partial Results. Colored Code Versions are the 8 Best-Performing Ones.

1) Comparison to CUB and OpenMP: Figure 7 shows the speedup of Tangram-synthesized code over the hand-written CUB library for parallel reduction on GPU and over the OpenMP version on the CPU. We only show the results for the best-performing Tangram-synthesized version on the three GPU architectures. The x-axis is the size of the input array and the y-axis the speedup over CUB baseline.

We observe that Tangram-synthesized code performs significantly better than the hand-written CUB code for small and medium-size arrays, i.e., below 1M elements. The speedup is between 2× and 6× on average depending on the GPU architecture and the array size. For large arrays, i.e., over 1M elements, Tangram-synthesized code is between 17% and 38% slower than the CUB code. The reason is that CUB applies bandwidth optimizations for large arrays, such as vector loads [37]. We profiled both Tangram and CUB codes, and observed that the total number of memory reads for the CUB code is significantly smaller than for the Tangram code. This observation correlates well with the optimizations for higher bandwidth utilization, which are currently not available in Tangram.

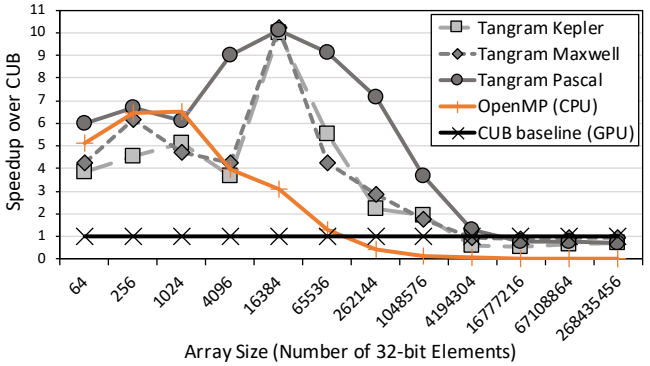


Fig. 7. Speedup of Best-performing Tangram-synthesized Code on Kepler, Maxwell, and Pascal GPUs over CUB Baseline Code. Higher is Better. Speedups of the OpenMP Version are with respect to CUB Baseline Code on Pascal GPU.

The comparison to the OpenMP version on the CPU is especially interesting for arrays below 1M elements, since such small arrays might be a better fit for CPUs. For this comparison, we do not include data transfers in our timings, because they might entail an overhead for both CPU and GPU codes. For instance, if an application is running on the GPU and, eventually, needs to compute the reduction of a small or medium-size array, executing it on the CPU with the OpenMP version could be a good choice. In that case, we would need to include the GPU-to-CPU data transfer time and back. We make several observations. First, the OpenMP version is clearly faster (by about 4x) than the CUB code below 65K elements for all GPU architectures. This indicates that CUB does not apply special optimizations for small arrays. Second, on the Kepler and the Maxwell GPUs, the OpenMP version outperforms the Tangram-synthesized code for small arrays (below 4K elements). This is because there is not enough data parallelism to overcome the higher latency on the GPU due to its lower clock frequency than that of the CPU. Third, on the Pascal GPU, the Tangram-synthesized code is competitive for small arrays due to Pascal's higher clock frequency over prior GPU architectures. For medium-size arrays (between 4K and 65K), the Tangram-synthesized code on the Pascal GPU is between $3\times$ and $6\times$ faster than the OpenMP version.

2) Detailed Comparison to CUB, Kokkos, and OpenMP on the Kepler GPU: Figure 8 compares the Tangram-synthesized code to CUB, Kokkos, and OpenMP codes on the Kepler GPU. For each array size, we show the Tangram code version (Figure 6) that provides the highest performance.

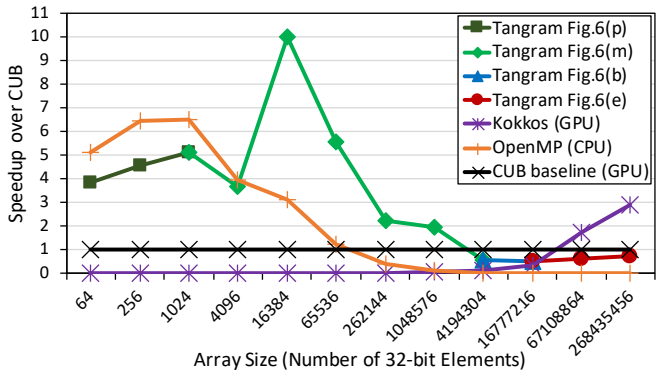


Fig. 8. Speedup of Tangram-synthesized Code on Kepler GPU over CUB Baseline Code and Kokkos Code. Different Tangram Code Versions correspond to Figure 6.

For small arrays (64 to 1K elements), the best Tangram-synthesized code is version *(p)* in Figure 6. Tangram generates this version from a *cooperative codelet* that uses atomic instructions on shared memory, as shown in Figure 3(b). The array is split among CUDA blocks, which execute tree-based summation using warp shuffle instructions. Partial results are accumulated with atomic instructions on shared memory. The resulting per-block partial results are atomically added on global memory. It is surprising that this version is the fastest on the Kepler GPU, since the Kepler architecture does *not* have support for fast atomic instructions on shared memory (see Section II-A). However, we observe that the number of array elements assigned per CUDA block in this version is small enough to have one single active warp per block. Thus, there is no contention for the atomic instructions on shared memory when accumulating the single partial result. For small arrays, the OpenMP version on the CPU is the fastest.

For medium-size arrays (1K to 4M elements), the best Tangram-synthesized code is version *(m)* in Figure 6. Tangram generates this version from a *cooperative codelet* like the one in Figure 1(c). In every block, warps execute tree-based summation using warp shuffle instructions. The partial per-warp results are added by a second tree-based summation, and not by atomic instructions on shared memory (version *(p)* in Figure 6) because the number of active warps per block is larger than that for small arrays. Profiling shows that, for version *(p)* in Figure 6, branch divergence [38], [39] is very high, which is mainly due to the lock-update-unlock mechanism in the Kepler architecture that uses branches [13]. For medium-size arrays, the Tangram code is on average $4.6\times$ faster than the CUB code, and $3.6\times$ faster than OpenMP code.

For large arrays (more than 4M elements), the best Tangram-synthesized versions distribute the input array over GPU software hierarchy levels twice (versions *(b)* and *(e)* in Figure 6). First, the array is partitioned across blocks with a *tiled* access pattern, and then across threads with a *strided* access pattern, by applying the *compound codelet* in Figure1(b) twice. The *strided* access pattern allows the thread coarsening optimization, which is available in the original Tangram [8]. Each thread applies serial sum (*codelet* in Figure 1(a)), and per-thread partial results are reduced by *cooperative codelets* (like Figure 1(c) and Figure 3(b)) with warp shuffle instructions. For large arrays, the Tangram code is about 38% slower than the CUB code. Beyond 10M elements, the Kokkos code outperforms CUB and Tangram codes by an average of $2.5\times$. In order to understand the significant speedup of the Kokkos code, we profile Tangram, CUB, and Kokkos codes. We discover that the Kokkos code uses multiple GPU kernels, and the most time-consuming kernel is compute-bound, *not* memory-bound as in Tangram and CUB codes. Memory-bound kernels cause significant slowness for large inputs, but compute-bound ones do not. The Kokkos code works by staging memory accesses for the main kernel through other sister kernels. These optimizations are not present in either Tangram or CUB, and are orthogonal to the optimizations studied in this work.

3) Detailed Comparison to CUB, Kokkos, and OpenMP on the Maxwell GPU: Figure 9 compares Tangram-synthesized code to CUB, Kokkos, and OpenMP codes on the Maxwell GPU. For each array size, we show the Tangram code version (Figure 6) that obtains the highest performance.

For small arrays (64 to 65K elements), the best Tangram-synthesized code is version *(n)* in Figure 6. Tangram generates this version from a *cooperative codelet* that uses atomic instructions on shared memory, as shown in Figure 3(a). The array is split among CUDA blocks. All threads of each block

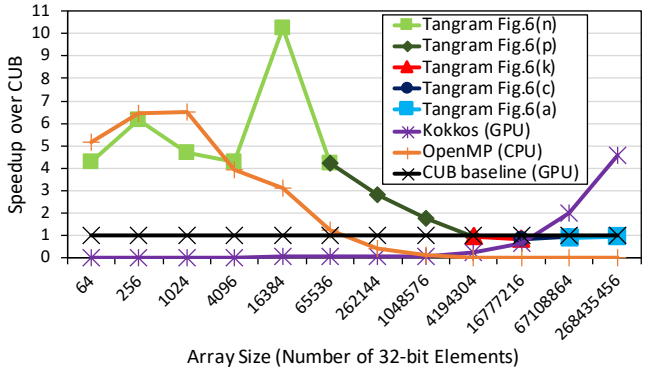


Fig. 9. Speedup of Tangram-synthesized Code on Maxwell GPU over CUB Baseline Code and Kokkos Code. Different Tangram Code Versions correspond to Figure 6.

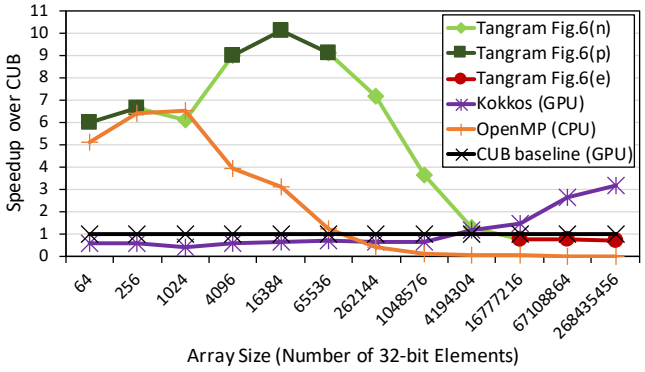


Fig. 10. Speedup of Tangram-synthesized Code on Pascal GPU over CUB Baseline Code and Kokkos Code. Different Tangram Code Versions correspond to Figure 6.

update a single accumulator with atomic instructions on shared memory. This is a clear example of how microarchitectural support for fast atomic instructions dictates the algorithm and optimization strategies that result in the the highest-performing code. Thus, the Maxwell GPU prefers version *(n)* in Figure 6 over version *(m)*, which is the version that the Kepler GPU prefers for arrays between 1K and 65K elements.

For medium-size arrays (65K to 4M elements), the best Tangram-synthesized code is version *(p)* in Figure 6. Tangram generates this version from a *cooperative codelet* like the one in Figure 3(b). In every block, warps execute tree-based summation using warp shuffle instructions. The partial per-warp results are added by atomic instructions on shared memory. For medium-size arrays, the Tangram code is on average $4.6\times$ faster than the CUB code, and $3.4\times$ faster than the OpenMP code.

For large arrays (more than 4M elements), the best Tangram-synthesized versions distribute the input array twice (versions *(a)*, *(c)*, and *(k)* in Figure 6). First, the array is partitioned across blocks with a *tiled* access pattern, and then across threads with a *strided* access pattern. Per-thread partial results are reduced by *cooperative codelets* without atomic instructions (Figure 1(c)) or with atomic instructions on shared memory (Figure 3(b)). For large arrays, the Tangram code is about 7% slower than the CUB code, and about $2.7\times$ slower than the Kokkos code for the same reasons as on the Kepler GPU.

4) Detailed Comparison to CUB, Kokkos, and OpenMP on the Pascal GPU: Figure 10 compares Tangram-synthesized code to CUB, Kokkos, and OpenMP codes on the Pascal GPU. For each array size, we show the Tangram code version (Figure 6) that obtains the highest performance.

As described in Section II-A, the Pascal architecture further improves the atomic instructions over the Maxwell architecture by introducing scopes. The additional support has direct effect on the best-performing Tangram-synthesized code versions. The Pascal GPU prefers algorithms that use one of the *cooperative codelets* in Figure 3. Thus, the best-performing Tangram codes are versions *(n)* and *(p)* in Figure 6.

For small arrays with up to 1K elements, the Tangram code performs on par with the OpenMP code. For arrays of size between 4K and 65K elements, the Tangram code outperforms the CUB code by about $8.5\times$ and the OpenMP code by about $4.8\times$, on average.

For medium-size arrays (65K to 4M elements), the Tangram code provides an average speedup of $4\times$ over the CUB code.

For large arrays (more than 4M elements), the Tangram code is, on average, 27% slower than the CUB code and $2.2\times$ slower than the Kokkos code, for the same reasons described for large arrays in the Kepler GPU and the Maxwell GPU (i.e., bandwidth optimizations in CUB and compute-bound kernel in Kokkos).

V. Related Work

To our knowledge, this paper is the first to enable the use of low-level instructions (in particular, warp shuffle instructions and atomic instructions on global and shared memory) in high-level GPU programming frameworks, via high-level APIs, array qualifiers, and AST transformations. By leveraging warp shuffle and atomic instructions, we develop performance-portable code for parallel reduction in the Tangram GPU programming framework [8]. We already extensively compared our approach and Tangram-synthesized code to two closely related works: the approach of and code generated by another performance-portable GPU framework (Kokkos [9]) and a hand-written library (CUB [7]). In this section, we describe other related work. First, we describe optimized hand-written implementations of parallel reduction on GPUs. Second, we discuss the existing support (if any) for low-level instructions and parallel reduction in state-of-the-art high-level programming frameworks.

Hand-written Reductions. There exist a substantial body of work on optimization for parallel reduction on GPUs. Harris [40] presents a tree-based algorithm and shows how to optimize it for shared memory, communication between CUDA blocks and warp divergence. Luitjens [41] applies warp shuffle instructions, available since the NVIDIA Kepler architecture, and explores optimizations with atomic instructions. Catanzaro

[42] discusses two-stage partition schemes to have a partial reduction per streaming multiprocessor, and a final reduction of partial results. All of the above strategies and optimizations are now available in Tangram via our contributions in this paper.

High-level Programming Frameworks. In terms of reduction-specific abstractions in high-level programming frameworks, several hand-written libraries provide optimized reduction through simple APIs [6], [7]. Higher-level performance portability frameworks, such as Kokkos [9] and Raja [27], provide *reduce* as an API that calls an in-house or a third-party library. Raja lets the user choose between different optimized versions of reduction through a *reduce_policy* template parameter (e.g., *cuda_reduce_async*, *cuda_reduce_atomic*, etc.). Both Kokkos and Raja rely on pre-written optimized reduction codes. High-level functional data parallel language Lift [28], similarly to Tangram, provides language primitives that help abstract data-parallel computation. Primitives such as *mapWarp* and *mapLane* are used for low-level optimizations, and the *toLocal*, *toGlobal* primitives are used for memory placement. As the name suggests, Lift is meant for data parallel computation, but coordinating lanes of parallel execution via warp shuffle instructions is not supported. The use of re-write rules [43], in order to transform high-level code to low-level optimized OpenCL code, has been previously proposed. However, OpenCL does not expose low-level intrinsics such as warp shuffle instructions. Thus, OpenCL cannot support the optimizations presented in this paper.

Halide [10] is a Domain Specific Language (DSL) for image processing pipelines with support for reduction operations [44]. Halide provides high-level APIs (e.g., *gpu_lane*) to expose warp shuffle instructions, and scheduling directives (e.g., *rfactor*) that split and compute partial results over slices of the reduction domain. However, as of the writing of this paper, *rfactor* is not implemented for GPU code generation and support for different GPU atomic instructions does not exist. PENCIL [45], an intermediate DSL for accelerators, has been optimized to support a number of low-level transformations for GPU reduction [46]. However, warp shuffle instructions and atomic instructions on global or shared memory are not exposed. Transformations discussed in our paper could be adopted by PENCIL and Halide.

VI. Conclusion

We introduce a new set of high-level APIs and memory qualifiers, as well as AST transformations, for high-level performance-portable programming frameworks and DSLs to enable automatic generation of warp shuffle instructions and atomic instructions on GPUs. We implement our techniques on the Tangram high-level programming synthesis framework, and augment Tangram's code generation capability with pre-processing for code variants. We implement *parallel reduction*, a building block for many complex and widely-used algorithms, and show how, depending on the ISA and the microarchitectural support for atomic instructions, different parallel reduction algorithms are more suitable for different arrays sizes and GPU architectures. We compare the performance of our Tangram-synthesized code against another performance-portable GPU framework (Kokkos [9]) and a hand-written library (CUB [7]), and show that our Tangram-synthesized code outperforms hand-written code by up to $7.8\times$ ($2\times$ on average) on three generations of GPU architectures.

Acknowledgments

This research is based in part upon work supported by: The Center for Exascale Simulation of Plasma-Coupled Combustion, one of six PSAAPII centers, funded by the U.S. Department of Energy, under Award Number DE-NA0002374. Sandia National Laboratories managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. The Center for Applications Driving Architectures (ADA), a JUMP Center co-sponsored by SRC and DARPA. The Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation award OCI-0725070 and the state of Illinois.

References

[1] E. Strohmaier, J. Dongarra, S. Horst, and M. Meuer, "Top500 List June 2018," *https://www.top500.org/lists/2018/06/*.

[2] F. Wu and T. Scogland, "Green500 List June 2018," *https://www.top500.org/green500/lists/2018/06/*.

[3] RightScale, "Rightscale 2018 state of the cloud report," *https://assets.rightscale.com/uploads/pdfs/RightScale-2018-State-of-the-Cloud-Report.pdf*.

[4] NVIDIA, "CUDA zone," *https://developer.nvidia.com/cuda-zone*.

[5] Khronos Group, "OpenCL 2.0 API specification," *https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf*, 2014.

[6] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*, 2011.

[7] D. Merrill and NVIDIA-Labs, "CUDA unbound (CUB) library," *NVIDIA-Labs*, 2015.

[8] L.-W. Chang, I. E. Hajj, C. Rodrigues, J. Gómez-Luna, and W.-m. Hwu, "Efficient kernel synthesis for performance portable programming," in *MICRO*, 2016.

[9] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, 2014.

[10] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*, 2013.

[11] J. Gómez-Luna, L.-W. Chang, I.-J. Sung, W.-M. Hwu, and N. Guil, "In-place data sliding algorithms for many-core architectures," in *ICPP*, 2015.

[12] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, 2013.

[13] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Performance modeling of atomic additions on GPU scratchpad memory," *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[14] S. Yan, G. Long, and Y. Zhang, "Streamscan: Fast scan algorithms for GPUs without global barrier synchronization," in *PPoPP*, 2013.

[15] L.-W. Chang, I. El Hajj, H.-S. Kim, J. Gómez-Luna, A. Dakkak, and W.-m. Hwu, "A programming system for future proofing performance critical libraries," in *PPoPP*, 2016.

[16] NVIDIA, "PTX: Parallel thread execution ISA version 6.2," *https://docs.nvidia.com/cuda/parallel-thread-execution*, 2018.

[17] L. Nyland and S. Jones, "Understanding and using atomic memory operations," *NVIDIA GTC*, 2013.
[18] J. Demouth, "Shuffle: Tips and tricks," *NVIDIA GTC*, 2013.
[19] NVIDIA, "NVIDIA next generation CUDA compute architecture: Kepler GK110 whitepaper," 2013.
[20] NVIDIA, "CUDA C programming guide," *https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*, 2018.
[21] D. B. Kirk and W.-M. W. Hwu, *Programming massively parallel processors: a hands-on approach*. 2016.
[22] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *GPGPU*, 2013.
[23] NVIDIA, "Fermi whitepaper," 2012.
[24] NVIDIA, "NVIDIA GeForce GTX 980: Featuring Maxwell, the most advanced GPU ever made," 2014.
[25] A. Adinets, "CUDA pro tip: Optimized Filtering with Warp-Aggregated Atomics," 2014.
[26] NVIDIA, "NVIDIA Tesla P100 GPU," 2016.
[27] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: overview and status," tech. rep., Lawrence Livermore National Lab., 2014.
[28] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: a functional data-parallel ir for high-performance gpu code generation," in *CGO*, 2017.
[29] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *MICRO*, 2016.
[30] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The Locality Descriptor: A holistic cross-layer abstraction to express data locality in GPUs," in *ISCA*, 2018.
[31] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *ISCA*, 2018.
[32] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *PACT*, 2014.
[33] L.-W. Chang, H.-S. Kim, and W.-m. W. Hwu, "Dysel: Lightweight dynamic selection for kernel-based data-parallel programming model," in *ASPLOS*, 2016.
[34] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *IPDPS*, 2010.
[35] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, 2008.
[36] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, 1998.
[37] J. Luitjens, "CUDA pro tip: Increase performance with vectorized memory access," 2013.
[38] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *MICRO*, 2007.
[39] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *MICRO*, 2011.
[40] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA CUDA SDK*, 2008.
[41] J. Luitjens, "Faster parallel reductions on Kepler," *NVIDIA*, 2014.
[42] B. Catanzaro, "OpenCL optimization case study: Simple reductions," 2010.
[43] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code," in *ICFP*, 2015.
[44] P. Suriana, A. Adams, and S. Kamil, "Parallel associative reductions in halide," in *CGO*, 2017.
[45] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, *et al.*, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *PACT*, 2015.
[46] C. Reddy, M. Kruse, and A. Cohen, "Reduction drawing: Language constructs and polyhedral compilation for reductions on GPU," in *PACT*, 2016.