Fireiron: A Data-Movement-Aware Scheduling Language for GPUs

Bastian Hagedorn*
University of Münster
b.hagedorn@wwu.de

Archibald Samuel Elliott*
lowRISC
sam@lenary.co.uk

Henrik Barthels*
AICES, RWTH Aachen University
barthels@aices.rwth-aachen.de

Rastislav Bodik*
University of Washington
bodik@cs.washington.edu

Vinod Grover
NVIDIA
vgrover@nvidia.com

ABSTRACT

High GPU performance can only be achieved if a kernel efficiently uses the multi-layered compute and memory hierarchies. For example, accelerators such as NVIDIA's Tensor Cores require specific mappings of threads to data that must be considered in data movements to and from registers. Current compilers struggle to match the performance of vendor libraries like cuBLAS, which are developed by experts in assembly. This manual low-level coding is time-consuming and complicates to unlock the full GPU potential, preventing experimentation to achieve even higher performance.

In this paper we introduce Fireiron, a scheduling language aimed at performance experts. Fireiron provides high-level abstractions for expressing GPU optimizations that are unavailable to compilers today and which so far must be written in assembly. Our innovation is that both computations and data movements are first class concepts that can be separately mapped to threads, as required for the efficient use of specialized hardware like Tensor Cores.

We evaluate Fireiron on three GPU architectures against expert-written advanced matrix multiplications. First, we show that Fireiron schedules are able to express the strategies of these implementations requiring about 6× less lines of code. Second, we show that the code generated by Fireiron schedules outperforms the fastest implementations (provided by cuBLAS) by more than 2×.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Parallel programming languages**; **Software performance**.

KEYWORDS

Data Movement; GPU; Optimization; Compilers; Fireiron

*The work described in this paper was done while the authors where at NVIDIA

PACT '20, October 3–7, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7092-9/20/08…$15.00
https://doi.org/10.1145/3410463.3414632

ACM Reference Format:
Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414632

1 INTRODUCTION

Developing high-performance kernels for GPUs is challenging because of their complex multi-layered compute and memory hierarchies. Only if a kernel makes optimal use of both hierarchies, implementations achieve performance close to the theoretical peak.

On modern GPUs, achieving optimal performance essentially boils down to careful data movements and the precise use of specialized hardware units such as NVIDIA's Tensor Cores. Today, there remains a significant gap between what optimizing compilers can achieve versus what human experts achieve by hand-tuning implementations using low-level assembly. Figure 1 (a) shows the performance of the best matrix multiplication implementations we found for Halide [23] and TVM [7], two state-of-the-art compilers, compared to the performance achieved by NVIDIA's experts providing manually tuned implementations in the high-performance cuBLAS library. Manually developing high-performance implementations is time-intensive and error-prone even for experts, and more crucially, it complicates experimentation and thus hinders potentially unlocking even higher performance.

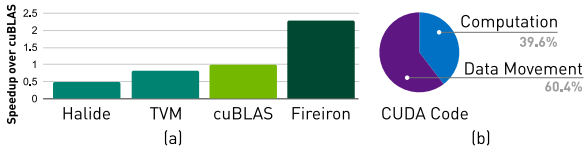


Figure 1: (a) Comparing Halide's [11] and TVM's [28] matrix multiplication performance against cuBLAS (higher is better) reveals a significant remaining gap. Fireiron allows GPU experts to specify implementations that even outperform hand-tuned cuBLAS library code. (b) The Fireiron-generated CUDA code achieving this performance contains mostly data movement optimizations which motivates a scheduling language where data movements are first-class constructs.

Schedule-based compilation [7, 23], which gained popularity with the introduction of Halide, is a huge step towards providing experts with a powerful tool for developing high-performance programs. However, the current approaches prevent experts from closing the remaining performance gap because they treat data movements as second-class citizens: In order to unlock the highest performance, it is crucial to define precise mappings of both, computations to parallel compute units but also how data movements are coordinated through the memory hierarchy.

In this paper, we propose Fireiron, a scheduling language, IR, and compiler for performance experts. With Fireiron, programmers can define where computation *and* data movement take place. This is required to unlock the potential of specialized hardware such as Tensor Cores. In this paper, we make the following contributions:

(1) We introduce a compiler IR in which both computations and *data movements* are first-class citizens, meaning that they can be scheduled with the same primitives.
(2) Fireiron's scheduling language provides high-level abstractions for *progressively decomposing* computations and data movements until they match assembly instructions, accelerator primitives, or predefined microkernels. This is achieved by representing each using precise *specifications*.
(3) We show that Fireiron schedules are able to express optimization strategies used in handwritten kernels while requiring 6× less code. With Fireiron, experts are able to develop high-performance GPU kernels computing matrix multiplications —as of today the best understood and most heavily optimized GPU computation—that even outperform cuBLAS hand-tuned library implementations by more than 2× as shown in Figure 1.

2 ACHIEVING HIGH GPU PERFORMANCE

Efficiently using the GPU's compute and memory hierarchy requires the coordinated application of multiple optimizations. When optimizing data movements for example, depending on which instructions are used, the ownership (that is the specific mapping of threads to data) is fixed which complicates achieving efficient memory access patterns. To alleviate this, data movements can be decomposed into two steps, allowing threads to exchange data in between, for achieving more efficient reads and writes. With the introduction of Tensor Cores, providing 12× more throughput than regular FMA instructions, data movements become even more challenging because their `mma.sync` instructions [18] impose complex ownerships involving groups of eight threads called Quad-Pairs.

Consider the epilog of a matrix multiplication kernel as an (often neglected) example which however is crucial to optimize because it is completely memory bandwidth limited. After performing the computation, in the epilog each thread-block must move the computed results of its tile back to global memory. Typically, these results are distributed across the registers of its threads. Figure 2 (left) shows a simple epilog in which every thread directly copies its computed results to global memory, and CUDA code implementing this data movement. This implementation is not very efficient due to uncoalesced writes to global memory, i.e., `C` needs to be accessed by writing rows instead of 8×8 tiles.

Figure 2 (middle) shows an optimized epilog and here, we additionally assume that the results were computed using Tensor Cores.

```
# Naive Algorithm
k = te.reduce_axis((0, K), 'k')
A = te.placeholder((M, K), name='A')
B = te.placeholder((K, N), name='B')
C = te.compute((M, N),
 lambda x,y: te.sum(A[x,k]*B[k,y], axis=k),name='C')

# Optimized Algorithm
packedB = te.compute(
 (N/bn,K,bn),lambda x,y,z:B[y,x*bn+z],name='packedB')
C_opt = te.compute((M, N),
 lambda x, y: te.sum(A[x, k] *
  packedB[y // bn, k, tvm.tir.indexmod(y, bn)],
  axis=k), name = 'C_opt')
```

Listing 1: Performing storage layout transformations in TVM requires to modify the algorithm instead of the schedule.

When Tensor Cores are used, at runtime a warp is partitioned into four so-called Quad-Pairs, groups of eight specific threads, which cooperatively execute an `mma.sync` instruction to compute an 8×8 tile of the output. Tensor Cores are programmable using a family of `mma.sync` variants for different operand storage layouts (row- or column-major) and accumulation precision (`FP16` or `FP32`). Each variant prescribes a different quad-pair-level ownership. In this version, the quad-pairs, and thus their threads, computed logically distributed tiles that are physically stored in contiguous registers. In order to achieve coalesced writes to global memory, threads have to exchange data in shared memory first to be able to store results which they themselves have not computed. Every block allocates a temporary buffer in shared memory for the coordinated data exchange (indicated by the different write and read patterns). The second step of this optimized epilog (`SHtoGL`) is rather standard: All threads of a warp move a complete row from shared- to global memory achieving coalesced and vectorized writes. The first step (`RFtoSH`), is more complicated due to the dictated Tensor Core ownership. Instead of simply materializing the ownership in shared memory in the first step, the data movement to and from shared memory can be further optimized: Padding the shared memory buffer with additional columns achieves a skewed read and write access pattern reducing the number of read and write bank conflicts.

Implementing the optimized epilog requires about 7× more lines of code and is significantly more complex than the simple version. In the next section, we explain how to express both versions in a concise and precise way using Fireiron as shown in Figure 2 (right).

Limitations of Today's Schedule-Based Approaches. In schedule-based compilers like TVM and Halide, data movements are treated as second-class citizens and expressing optimizations for them is a stretch and blurs the line between algorithms and schedules. For example, expressing optimizations like storage layout transformations require drastic changes to the algorithm instead of being expressible as a schedule. Listing 1 shows a naive matrix multiplication algorithm in TVM [27]. Because existing scheduling languages can only decompose computations, a modified and optimized algorithm in which a new function `packedB` must be introduced to permute elements in memory. Additionally, the existing languages allow to allocate temporary buffers in specific locations (e.g., using Halide's

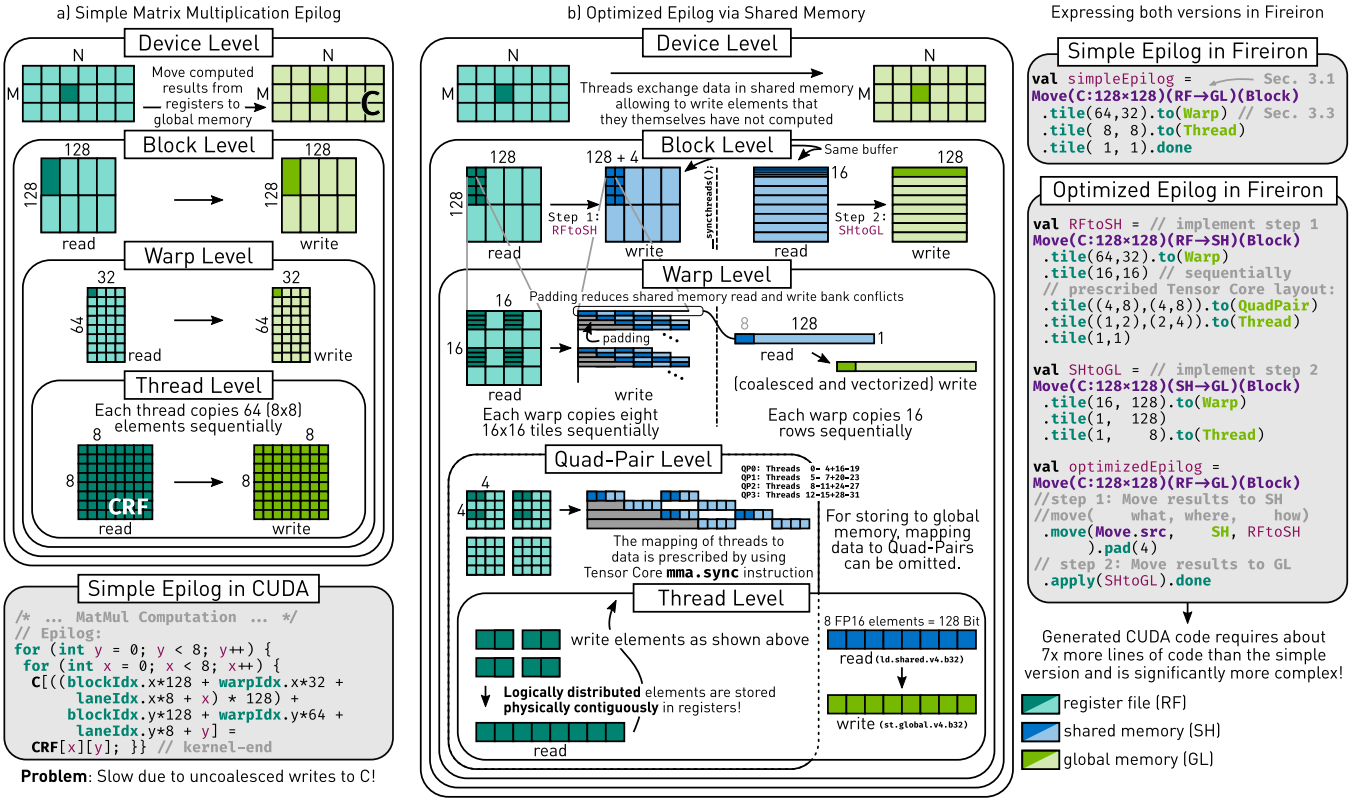


Figure 2: Two options for implementing the epilog of a matrix multiplication. Option 1 (top): Directly copy the results from register file (CRF) to global memory (C). Option 2 (bottom): Synchronize via shared memory. This allows vectorized and coalesced stores to global memory while avoiding bank conflicts using padding in shared memory.

store_in primitive) and by introducing identity computations as redundant compute stages, data movements are implicitly scheduled by scheduling the computation of the producer and consumer stage. The compiler then needs to infer the implications for the associated reads and writes of this data movement during code generation. However, inferring the coordination required for expressing advanced data movements such as the epilog shown in Figure 2 is beyond the reach of automatic compiler analysis.

Additionally, using Tensor Cores efficiently is challenging: A warp executes four mma.sync instructions simultaneously, each collectively executed with eight specific threads (a quad-pair) operating on the same 8×8 tile. NVIDIA provides a CUDA interface (WMMA [17]) which exposes a conceptually single, warp-wide macro-mma using a fixed data-to-quad-pair mapping that is optimal in some cases but not all. Some situations require more sophisticated mappings such as the one shown Figure 2b where a quad-pair, and thus its threads, operate on interleaved distributed tiles. In Fireiron one can a) flexibly decompose warp-level computations to quad-pairs and b) implement the required data movements by treating moves as schedulable operations. Existing scheduling languages lack these mechanism and currently only target WMMA which potentially explains the remaining gap in performance shown in Figure 1.

3 RETHINKING SCHEDULING LANGUAGES

In this section, we introduce Fireiron's core concepts:

(1) *Specifications* for describing the task performed within a certain region of code in a GPU kernel, and
(2) *Decompositions* which form our scheduling language and describe how to implement a given specification.

The key idea behind Fireiron is that implementation strategies are described by a schedule, i.e., a sequence of decompositions that gradually decompose the given spec into a partial implementation and one or more nested sub-specifications. Applying a schedule to an initial specification describing the kernel-computation creates our IR of nested specifications which naturally reflects the hierarchical structure present in high-performance GPU implementations.

In Figure 3, we show this structure and the role of data movements using a simple matrix multiplication kernel as an example. Conceptually, the computation is hierarchically decomposed into (i) sub-computations of the same kind and (ii) data movements. Here, for example, the outermost computation is decomposed into matrix multiplications operating on smaller shapes (blue boxes), until eventually every thread computes a single FMA instruction in registers (innermost blue box) that can be viewed as a matrix multiplication of matrices containing only a single element. In between, data is moved to lower levels of the memory hierarchy (purple boxes), and

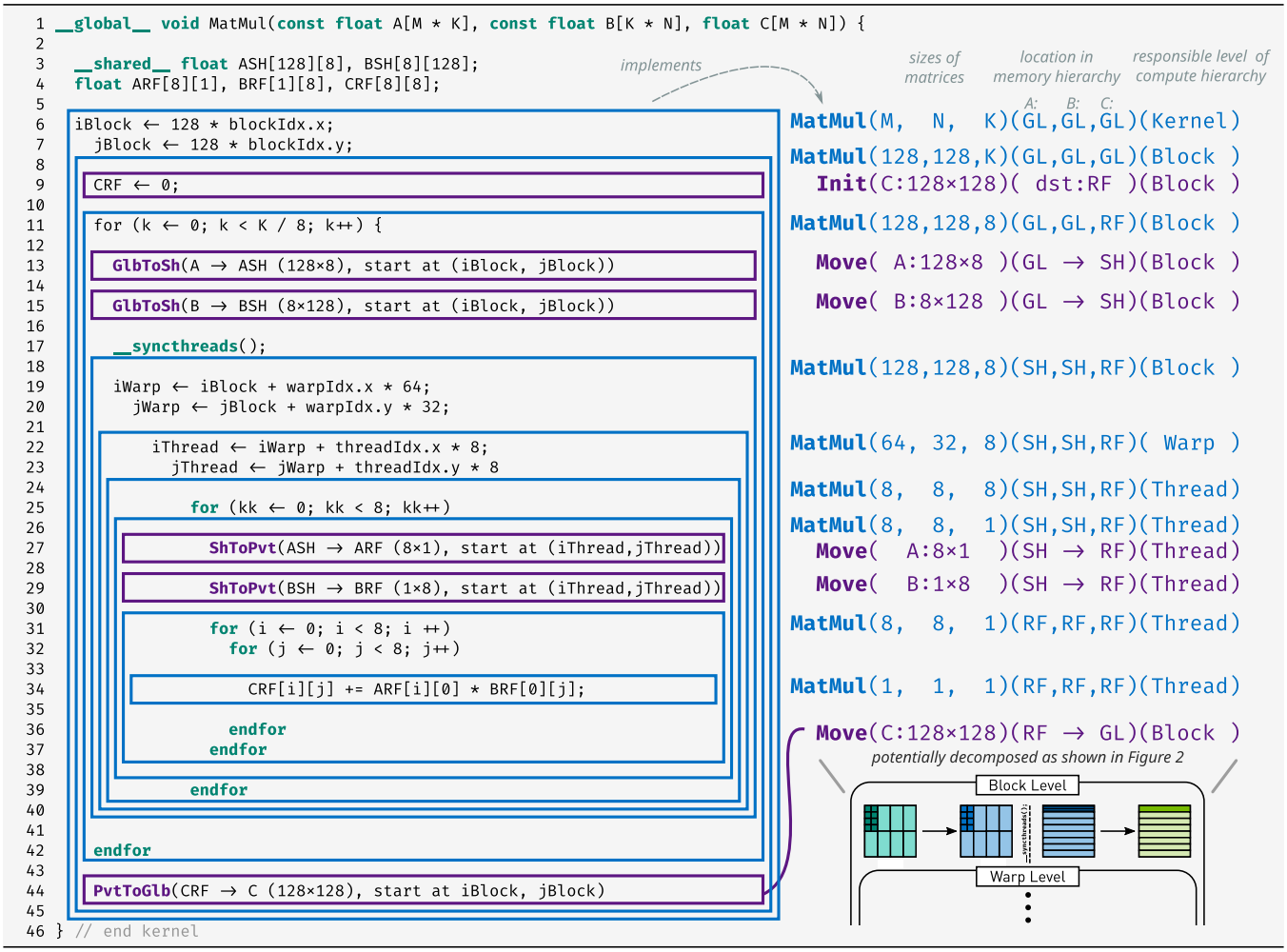


Figure 3: Showing the hierarchical structure of GPU kernels using a matrix multiplication as example. Within a typical GPU kernel, we gradually descend the compute and memory hierarchy while computing smaller instances of the original problem.

for brevity, no data movement implementations are shown, i.e., no purple box contains nested boxes. However, every data movement is similarly decomposed as indicated on the bottom where the epilog of this kernel (the last purple box) might be implemented analogously to the data movements explained in Figure 2.

Figure 4 shows how the code in Figure 3 is expressed using Fireiron's scheduling language. The corresponding IR is created by progressively decomposing the nested specifications according to the schedule. For brevity, we omit the partial implementations (as shown later in Figure 6), and only show the nesting of specifications. Here, we show the decomposition of the epilog by applying the `simpleEpilog` visualized in Figure 2. For brevity, we again omit the details for the remaining data movements and instead indicate their decomposition (using `.apply(`*schedule*`)`).

Currently, Fireiron is implemented as a domain-specific language embedded in Scala and generates CUDA kernels with inline PTX assembly. The rest of this section describes Fireiron's *Specifications* and *Decompositions* in more detail.

3.1 Specifications

Note that each box in Figure 3 can be labeled with a precise description of the task performed inside it. We call this the *Specification* (spec). Its implementation is observed by looking inside the box where the task is further decomposed. In Fireiron, a specification is a data-structure describing the current task to implement. A spec contains enough information such that a programmer would be able to manually provide an implementation. This especially entails that it contains the shapes, locations and storage layouts of its input and output operands, including the responsible level of the compute hierarchy performing this operation. Currently, Fireiron supports two main classes of specs: matrix multiplication (**`MatMul`**), and data movement (**`Move`**). **`Move`** specs explicitly and exclusively introduce data movements and allow Fireiron to express precise implementation strategies for them. Figure 5 (top) shows a kernel-level **`MatMul`** spec and a **`Move`** spec describing the movement of a matrix `src` from global to shared memory during which the storage layout is transformed from column- to row-major.

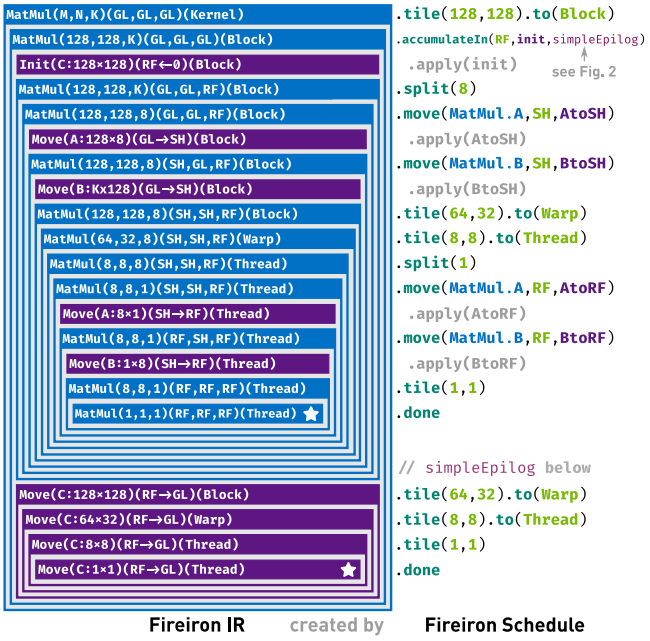


Figure 4: Describing and representing the implementation shown in Figure 3 using Fireiron's decompositions constructing the IR of nested specifications. (*Executable* specifications are annotated with a star)

Example `MatMul` Spec:

```
MatMul(ComputeHierarchy: Kernel,
  A:Matrix((M x K),FP32,GL,ColMajor),
  B:Matrix((K x N),FP32,GL,ColMajor),
  C:Matrix((M x N),FP32,GL,ColMajor))
```

Example `Move` Spec:

```
Move(ComputeHierarchy: Block,
 src:Matrix((128×8),FP32,GL,ColMajor),
 dst:Matrix((128×8),FP32,SH,RowMajor))
```

Example for an *executable* (★) `MatMul` Spec: (operating on `FP16`)

```
MatMul(1,1,1)(RF,RF,RF)(Thread)                    ★
C[ ... ] = __hfma(A[ ... ],B[ ... ],C[ ... ]);
```

Example for an *executable* `Move` Spec: (operating on `FP16`)

```
Move(A:1×8)(GL→RF)(Thread)                         ★
int4 *       rf_ptr = (int4 *)       &ARF[0];
const char * gl_ptr = (const char *) &A[ ... ];
asm volatile( \                  copying 8 __half (FP16) at once
 "{ld.global.nc.v4.b32 {%0,%1,%2,%3},[%4];}" \
     : "=r"(rf_ptr[0].x), "=r"(rf_ptr[0].y)
     : "=r"(rf_ptr[0].z), "=r"(rf_ptr[0].w)
     :  "l"(gl_ptr));
```

Figure 5: Examples of Fireiron specs: A kernel-level matrix multiplication computation and a specification for a data movement from global to shared memory. An *executable* spec directly corresponds to an instruction like `__hfma` in CUDA or `ld.global.nc.v4.b32` in PTX.

For matrices, Fireiron supports both constant and symbolic shapes shapes written as arithmetic expressions, e.g., `M=((x+y)%z)` where x, y and z are only known at compile-time. For brevity in figures, we sometimes omit information and, for example, write `MatMul(M,N,K)(GL,GL,GL)(Kernel)` and `Move(src:128x8)(GL->SH)(Block)` for the two upper specs shown in Figure 5.

Inspired by Chapel [5], Fireiron also provides the illusion of block-wide matrices residing in the thread's registers. This *distributed array* abstraction allows Fireiron users to think of this matrix as a contiguous block-wide matrix whereas actually every thread contains only a small tile in its registers. During code generation, we naturally lower this abstraction to low-level CUDA or inline PTX code which does not provide this abstraction, as described in the next section.

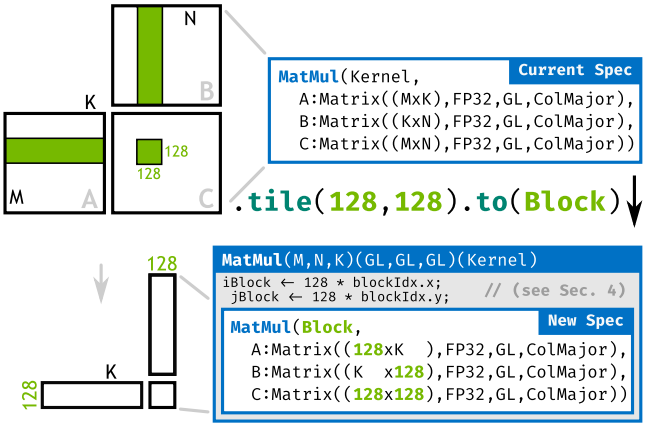
Executable Specifications. A specification is called *executable* when it describes the semantics of a built-in instruction or library implementation. Fireiron, provides a predefined set of executable specs matching different CUDA and PTX instructions. Figure 5 (bottom) shows examples for executable `MatMul` and `Move` specs and their associated code snippets. The idea is to gradually decompose specifications until only executable specs remain for which we know how to generate code.

3.2 Decompositions

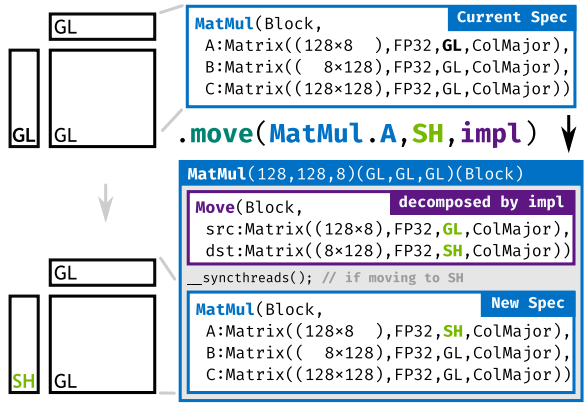
A *Decomposition* describes how to implement a spec. For example, we might decide to implement a matrix multiplication in a tiled fashion which effectively decomposes the initial computation into tiles of smaller matrix multiplications and a loop-nest describing how to iterate over these tiles. The smaller matrix multiplication tiles are then further decomposed by the subsequent decompositions in a given Fireiron schedule. Fireiron provides four decompositions for describing the implementation strategies of both computations (`MatMul`) and data movements (`Move`).

The `.tile` Decomposition. Figure 6a shows the application of the `tile` decomposition to a `MatMul` spec. Generally, `spec.tile(r,c)` partitions the output into $r \times c$ shaped tiles. The input matrices are tiled accordingly, i.e., tiling a `MatMul` spec partitions the A matrix into row-tiles and B into column-tiles. Tiles can also be non-contiguous as shown in Figure 2 in which case `.tile` expects a width and offset for both dimensions. The `tile` decomposition allows to assign tasks to a level of the compute hierarchy: We can *refine* the tiling using `.to(level)` which changes the responsible compute hierarchy level for the resulting tiled spec. `tile` is also applicable to a `Move` spec in which case the input (`src`) and output (`dst`) matrices are tiled in the same way. In our current implementation, the input sizes must be evenly divisible by the tile sizes. This limitation could be resolved with using predicated PTX instructions for dealing the partial tiles at the boundaries.

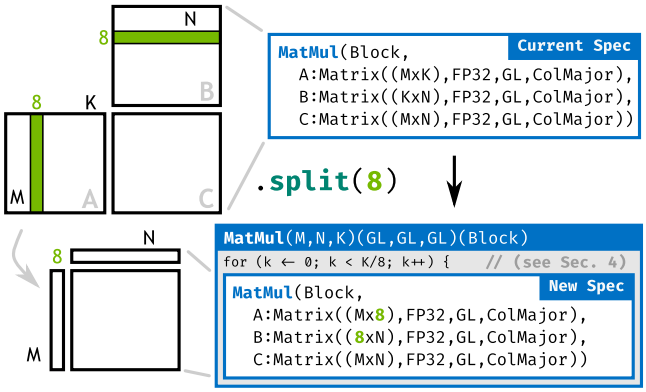
The `.move` Decomposition. Applying `.move` explicitly introduces data movements for moving the operand of a given spec to a new location in the memory hierarchy. Figure 6b shows the application of the `move` decomposition to a `MatMul` spec. Here, we move the `A` operand from global to shared memory. The `move` decomposition expects three arguments: the matrix to move, a destination in the



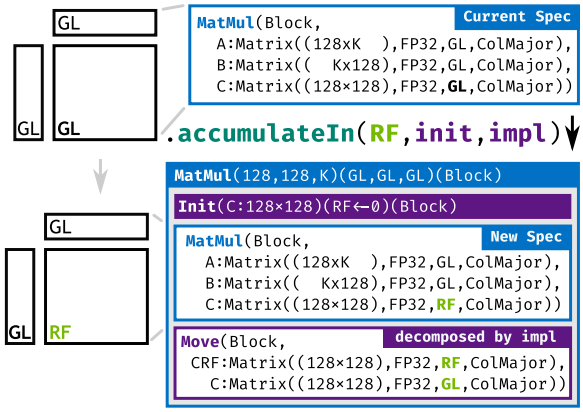
(a) Tiling a `MatMul` spec results in a decomposed subspec with adjusted dimensions and optionally adjusted compute hierarchy to indicate parallel execution.



(b) Applying `move` to a `MatMul` spec results in a new spec in which the memory location of the specified operand has changed. A `Move` spec is created representing the data movement which is implemented as specified in the strategy *impl*.



(c) The `split` decomposition allows to create tiles in the K-dimension of the input operands of a `MatMul` spec.



(d) The `accumulateIn` decomposition allows to accumulate the results of a matrix multiplication in lower levels of the memory hierarchy and to specify the data movement of the results back to global memory.

Figure 6: Visualization of Fireiron's Decompositions.

memory hierarchy and a schedule *impl* describing how to implement the movement. Applying `move` always creates *two* new nested specs: First, a `Move` representing the data movement to the new location whose implementation is described in the *impl* schedule. Second, an updated version of the input spec where the location of the moved operand has changed. The `move` decomposition can also be applied to a `Move` spec which allows us to specify data movements via an indirection as described and shown in Figure 2.

The `.split` Decomposition. The `tile` decomposition creates tiles in the M and N dimension of the `MatMul` operands, but we also need to be able to create tiles in the K-dimension. Figure 6c shows the application of the `split` decomposition which enables this.

The `.accumulateIn` Decomposition. Finally, we need to be able to specify that results shall be accumulated in lower levels of the memory hierarchy, typically in registers, and their movement back to global memory in the epilog of a matrix multiplication kernel. Figure 6d shows the application of the `accumulateIn` decomposition which expects three arguments: The location of the accumulation buffer, a schedule *init* describing its initialization, and a schedule *impl* specifying how to move the computed results back to global memory. Similar to `move`, `accumulateIn` creates multiple sub-specs. First, an `Init` spec (a variant of `Move` without source operand) representing the initialization of the buffer. Its implementation is described in the *init* schedule. Second, the new `MatMul` spec with an updated location of the `C` matrix. Third, the `Move` representing the movement of the results to global memory.

At any time in a schedule, one can also provide a micro-kernel implementing the current spec. This allows to use a custom implementation for a specific region of the kernel whose behavior is described by a spec but for which our decompositions do not provide suitable abstractions.

3.3 Advanced Optimization using Refinements

Fireiron provides a set of *refinements*, i.e., optional modifications for decompositions, exposing more fine-grained control required to achieve high performance.

By default, `tile` creates tiles that will be computed sequentially using nested `for`-loops. The `to` refinement allows to compute tiles in parallel instead. Internally, Fireiron uses one-dimensional compute hierarchy indices that are mapped in a row-major order to the two dimensional tile arrangement. The `layout` refinement allows to change this order to column-major and `swizzle` enables even more complex mappings by first permuting the one-dimensional indices arbitrarily before assigning them to the tiles. Using `unroll` emits `#pragma unroll` above the loops.

The `move` decomposition can be refined as well. Using `pad(n)` modifies the memory allocation (discussed in the next section) for the destination buffer associated with the created `Move` spec and allocates n additional columns to avoid memory bank conflicts. `prefetch` generates double-buffered, prefetched versions of the data movement. We emit `__syncthreads()` if the destination location is shared memory. This can be suppressed using `noSync` in situations in which no explicit synchronization is necessary. Finally, `storageLayout` allows to specify a row- or column-major storage layout for the destination buffer.

Similar to `tile`, the `split` decomposition can also be refined using `unroll`, to unroll the generated loop. Using the `sync` refinement emits `__syncthreads();` as the last statement in the body of the created for-loop. This may be required depending on how shared memory is used in a strategy.

We are aware that some refinements, especially `noSync` and `sync`, allow the specification of incorrect implementation strategies that might lead to race conditions. However, a decomposition without refinements always generates correct code. So far, this has caused no problems as Fireiron is meant to be used by performance experts. We intend to improve the analyses of strategies to ensure these refinements cannot cause correctness issues in the future.

4 CODE GENERATION AND OPTIMIZATION

Fireiron's IR of nested specs naturally reflects how GPU kernels are structured. Therefore, code generation almost boils down to pretty printing the IR, traversing it from top to bottom.

As mentioned in the previous section, applying a decomposition to a spec describes how to implement it by creating one or more specs, and by emitting code to implement the decomposition. For example, using `tile` generally emits two `for`-loops that sequentially iterate over the created tiles (e.g, Figure 3, lines 31-32). If tiles are assigned to the compute hierarchy using `.to`, instead of emitting sequential loops, the tiles are computed in parallel using the unique compute hierarchy indices for accessing the matrices. Figure 2 (right-top) shows a Fireiron example using both sequential and parallel tiles and the corresponding CUDA code we generate is shown at the bottom left.

Using the `split` decomposition emits one loop iterating over the tiles in the K-dimension. The `accumulateIn` and `move` decomposition emit no code themselves (except for synchronization if the destination is shared memory). Instead, the created sub-specs will be further compiled to CUDA code. The `done` operator is called on an executable spec to trigger code generation where we inject the associated code snippet. Optionally, `done` accepts a `String`: a micro-kernel we inline during code generation that implements the current spec.

Memory Allocation. In order to allocate enough memory, we traverse the IR once and register all `Move` specs which specify how much to allocate where - by definition. Generally, every level of the compute hierarchy is associated with a level of the memory hierarchy (`Kernel`→`GL`, `Block`/`Warp`→`SH`, `Thread`→`RF`). There are two potential cases to consider: Either, the responsible compute hierarchy of the `Move` (or `Init`) spec matches the associated destination, or they do not match. The first case is straightforward. For example, when visiting the spec `Move(A:128x8)(GL->SH)(Block)`, we emit

```
float __shared__ ASH[128*8];
```

at the beginning of the kernel.

The second case (i.e., memory allocation for distributed arrays) is more complicated. Since the responsible compute hierarchy of the `Move` or `Init` spec does not match the associated destination, we need to continue traversing the decomposition as shown in the example. For example, visiting `Init(C:128x128)(GL->RF)(Block)` specifies the need to allocate memory in the register file. However, allocating 128×128 elements per thread are far too many because a single thread only owns a small piece of the whole matrix. In order to find out how many elements we need to allocate per thread, we need to traverse the decomposition until we find the tile size assigned to threads:

```
Init(C:128x128)(GL->RF)(Block)
.tile(64,32).to(Warp)
.tile( 8, 8).to(Thread)//<-allocate 8x8 floats per thread
.tile( 1, 1).done
```

In this case, we emit `float CRF[8*8];`.

Index Computation. Index expressions for every operand are computed automatically while decomposing specs. Every operand has an associated row- and column index that is gradually updated. Every application of a decomposition returns a new spec in which we either sliced the operands or moved them to a new memory location. For example, applying `MatMul.tile(rs,cs)` generates two nested sequential for-loops with indices `rowTile`, and `colTile`. The index expressions for the matrices in the resulting tiled `MatMul` spec are updated as follows:

```
C.rowIndex += rowTile * rs;
C.colIndex += colTile * cs;
A.rowIndex += rowTile * rs;
B.colIndex += colTile * cs;
```

If tiles are computed in parallel, by for example using `.to(Block)`, we use `blockIdx.x` and `blockIdx.y` instead of `rowTile` and `colTile`. Similarly to memory allocation for distributed arrays, we have to compare the current compute hierarchy with the memory location of the array to update: In the following example

```
MatMul(128,128,8)(GL,GL,RF)(Block).tile(64,32).to(Warp)
```

we do update the `A` and `B` index expressions but not the index expressions for `C`. This is because `C` resides in registers and accessing

```
MatMul(16,16,16)(FR,FR,FR)(Warp)
  wmma::mma_sync(CFR, AFR, BFR, CFR);
Move(A:MxK)(GL→FR)(Warp)
  wmma::load_matrix_sync(AFR, A[...], K);
Move(C:MxN)(FR→GL)(Warp)
  wmma::store_matrix_sync(
    &(C[...]), CFR, N, wmma::mem_col_major);
```

Figure 7: Supporting the CUDA WMMA API in Fireiron by adding new warp-level executable specifications.

```
1  val simpleWMMA = MatMul(M,N,K)(GL,GL,GL)(Kernel)
2    .tile(64,64).to(Block)
3    .accumulateIn(FR, // accumulate in warp-level fragments
4      Move.tile(16,16).to(Warp).done, //  init strategy
5      Move.tile(16,16).to(Warp).done) // store strategy
6    .split(16)
7    .tile(16,16).to(Warp)
8    .move(MatMul.A, FR, Move.done) // 16x16 A tiles to FR
9    .move(MatMul.B, FR, Move.done) // 16x16 B tiles to FR
10   .done // => residual: MatMul(16,16,16)(FR,FR,FR)(Warp)
```

Listing 2: Simple Fireiron WMMA decomposition describing the implementation of the first cudaTensorCoreGemm kernel shown in the CUDA samples [15].

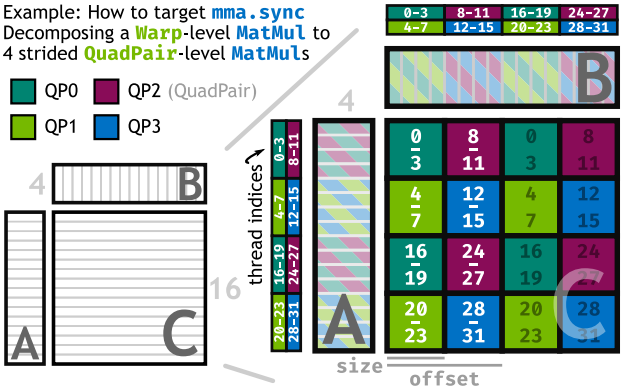
it using the warp indices would be incorrect. Instead, we only start updating the index expression for `C` as soon as we pass the `Thread`-level. The CUDA code in Figure 2 shows exactly this effect where `C` (residing in global memory) is accessed using all compute hierarchy indices whereas `CRF` (residing in registers) is only accessed using the indices used below the `Thread`-level. The `split` decomposition updates the indices as expected and every time we allocate a matrix in a new memory location (using `move` or `accumulateIn`), we start with fresh index expressions in the nested specs.

In order to support Tensor Cores, we simply extended the set of executable specs and target them using decompositions.

4.1 Supporting WMMA in CUDA

The WMMA-API in CUDA introduces warp-wide matrix multiply primitives operating on register collections called *fragments*. For generating kernels using WMMA primitives, we extend Fireiron in two ways: First, we extend the memory hierarchy and add a new level `Fragment<M,N,K>` (labeled `FR` if $M = N = K = 16$) in between shared memory and registers. Fragments are parameterized because in the CUDA API, sizes are part of the fragment type.

Second, we define new executable specs corresponding to the CUDA API calls. Figure 7 shows examples of new executable WMMA specs. Listing 2 shows how these additions allow to write a strategy targeting the new executable WMMA specs. It computes the matrix multiplication as follows: 1) assign 64×64 elements to a block (line 2); 2) initialize 16 (4×4) accumulator fragments (line 4); 3) fill operand fragments (lines 8–9); 4) compute the result (line 10); and 5) store results from fragments to global memory (line 5). Note that we only need to decompose specs to level of warps because of the new warp-level executable specs.




```
MatMul(16,16,4)(RF,RF,RF)(Warp)
  .tile((size=4,offset=8),(4,8))
  .to(QuadPair).layout(ColMajor)
```

decomposes to

```
MatMul(ComputeHierarchy: QuadPair,
A:Matrix((8×4),FP16,RF,RowMajor),
B:Matrix((4×8),FP16,RF,ColMajor),
C:Matrix((8×8),FP16,RF,RowMajor))
```

```
int * a = (int *) &ARF[0]; // contains 4 __half
int * b = (int *) &BRF[0]; // contains 4 __half
int * c = (int *) &CRF[0]; // contains 8 __half
  asm volatile( \
    "mma.sync.aligned.m8n8k4.row.col.f16.f16.f16.f16 \n" \
    "     {%0, %1, %2, %3}, \n" \
    "     {%4, %5}, \n" \
    "     {%6, %7}, \n" \
    "     {%0, %1, %2, %3}; \n" \
      : "+r"(c[0]), "+r"(c[1]), "+r"(c[2]), "+r"(c[3])
      :  "r"(a[0]),  "r"(a[1])
      ,  "r"(b[0]),  "r"(b[1]));
```

Figure 8: Supporting `mma.sync` in Fireiron by decomposing `MatMul` to the new `QuadPair`-level executable spec.

4.2 Supporting `mma.sync` in PTX

Using the `mma.sync` PTX instruction [18] allows even more fine-grained control over how Tensor Cores are used. First, we define the different `mma.sync` variants as executable `QuadPair`-level specs as shown at the bottom of Figure 8. We can then flexibly target the new executable specs in multiple ways. Figure 8 shows one possible decomposition of a contiguous `Warp`-level `MatMul`-spec to four strided executable `QuadPair`-level specs (indicated by different colors). Here, every thread of a `QuadPair` stores four elements from each input operand and, after collectively executing the `mma.sync` instruction, contains eight elements of the C matrix in its registers. We use the `layout` *refinement* for `.tile`, which we explain shortly, to assign tiles to quad-pairs in a column-major order.

As the two Tensor Core examples show, supporting new instructions in Fireiron requires only small changes and allows to target complex low-level PTX instructions using simple high-level abstractions. With the introduction of new instructions, e.g., the Turing and Ampere architectures contain even wider `mma` instructions, new executable specs can simply be added.

Reference	Description
`maxwell`	Manually-tuned CUDA kernel written by NVIDIA's performance experts targeting the Maxwell architecture (without Tensor Cores)
`wmma`	Publicly available CUDA sample [15] targeting the WMMA Tensor Core API
`cuBLAS`	NVIDIA's high-performance math library (using `cublasGemmEx` with `CUBLAS_GEMM_DEFAULT_TENSOR_OP` to enable Tensor Cores on Volta and Turing)

Table 1: Reference implementations used in the evaluation

	Reference Code	Fireiron Schedule	Fireiron Generated Code
`maxwell`	72 (68.1%)	44 (81.8%)	94 (67.0%)
`wmma`	122 (41.0%)	26 (76.9%)	113 (65.4%)
`cuBLAS`	closed source	49 (83.7%)	260 (60.4%) (small)
`cuBLAS`		46 (84.8%)	309 (72.2%) (large)

Table 2: Lines of Code and data-movement related lines (in %) for references, Fireiron strategies and generated code. For comparing with cuBLAS, we use two different strategies.

5 EVALUATION

In this section, we seek answers to the following questions: If data movements are as important as we think, how much data movement code is present in high-performance GPU kernels? Are we able to express the optimizations experts apply as strategies using Fireiron's decompositions? Does the code we generate perform as well as the manually written implementations? And finally, can Fireiron be used to improve the performance of state-of-the-art implementations?

References and GPU architectures. Table 1 shows the reference implementations used in this evaluation. We choose these because they apply different optimizations targeting specific GPU architectures. We used three GPUs: GeForce GTX 750 Ti (Maxwell), Quadro GV100 (Volta) and GeForce RTX 2080 Ti (Turing) because they cover different architectures that need to be optimized differently.

Methodology. We used CUDA-10.0, Driver Version 425.00 and compiled kernels using `-O3 –use_fast_math -arch= sm_XX` where `XX` = 52, 70, and 75 for Maxwell, Volta, and Turing respectively. We locked the clocks to fixed frequencies, report the minimum kernel runtime of 1000 runs using *nvprof* and omit the time required for data transfers because we are only interested in the quality of our generated kernel code.

The performance reported in Figure 1 was measured using public Halide [11] and TVM [28] code, their best matrix multiplication versions we are aware of. At the time of measuring, the hardware used for the other experiments was not available to the authors anymore. Instead we used a Titan XP (Pascal, latest architecture without Tensor Cores) for Halide and a GeForce RTX 2080 (Turing) for TVM because they target Tensor Cores. The TVM code was tuned according to the instructions and we report the best found performance.

Hypothesis A: *Code related to data movements makes up a significant fraction in high-performance kernels.*

If this is true, we argue that scheduling languages should treat data movements as first-class concepts. We find that about 2/3 of a kernel is devoted to optimize data movements.

We count and label the lines of our reference implementations as either related to data movements or to computations. Since there is not always a clear purpose associated with a single line of code, we made a conservative distinction and only count lines for data movements that are: a) declarations of temporary buffers, b) `__syncthreads()`, c) swizzling index computations solely used for avoiding bank conflicts, and finally, loops that *only* copy data in their bodies. Everything else counts as 'computation' lines.

Table 2 shows our results. Because `cuBLAS` is closed source, we additionally analyzed the TVM generated code (Figure 1), which contains 49 LoC with a data-movement fraction of 77.6%. We also analyzed the corresponding Fireiron strategies and generated code to show how our generated code relates to code written by experts. For comparing against `cuBLAS`, we developed two different schedules, one more suitable for smaller and one for larger input matrices, as explained in more detail in Hypothesis C.

We are aware of our inconclusive small sample size. However, these numbers already show that data movements optimizations cannot be neglected as they currently are in existing scheduling languages. This is further underlined by the performance our data-movement-heavy kernels achieve (evaluated in Hypothesis C and D) compared to state-of-the-art implementations.

Hypothesis B: *Fireiron can express optimizations that are applied by experts in manually-tuned code.*

We find that this is *mostly* true and that limitations of our scheduling language can be circumvented by inlining micro-kernels for sub-specifications.

Figure 2 showed how Fireiron allows to describe complex optimizations as high-level strategies. The optimized data movement is used in one of our `cuBLAS`-strategies. Listing 3 shows two Fireiron strategies expressing the `maxwell` reference (top) and the `wmma` reference (bottom). In the `maxwell`-schedule for example, we use different strategies for moving A (lines 16–20) and B (lines 22–26) to shared memory because considering the storage layouts separately enables coalesced global memory loads for both operands. We use swizzling (line 2) [20], and specify which loops to unroll and where to add or avoid synchronization with refinements. We also use vectorized loads (lines 34 and 35) and strided tiles (line 30).

However, the `maxwell` kernel uses a clever trick in its epilog: It streams data through shared memory in a way that allows to allocate less memory than we currently do. We cannot yet express this in Fireiron but the overall epilog is still precisely described by a `Move` specification, allowing us to inline a micro-kernel during code generation instead (line 13). Having specifications describing every decomposed sub-problem enables a fine-grained reuse of efficient implementations as inlined micro-kernels during code generation.

```
val swizz: Swizzle = id => // permutation of thread-ids
  ((id >> 1) & 0x07) | (id & 0x30) | ((id & 0x01) << 3)
val storeCUDA: String = //* CUDA Epilog Micro Kernel  */
// MATMUL-KERNEL ////////////////////////////////////////
val maxwellOptimized = MatMul(M,N,K)(GL,GL,GL)(Kernel)
///// BLOCK-LEVEL ////////////////////////////////////////
  .tile(128,128).to(Block).layout(ColMajor)
//--- accumulate in RF and use microkernel for epilog--//
  .accumulateIn(RF, Init
     .tile(64,32).to(Warp)
     .tile(8,  8).to(Thread) // alloc 64 reg per thread
     .tile(1,  1).unroll.done,
    Move.done(storeCUDA) /* use microkernel (18 LoC) */ )
  .split(8).sync
//--- move A to SH -----------------------------------//
  .move(MatMul.A, SH, Move(A:128x8)(GL->SH)(Block)
    .tile(128, 1).to(Warp)
    .tile(64,  1).unroll // copy in two steps
    .tile(2,   1).to(Thread).layout(ColMajor)
    .done).storageLayout(ColMajor).noSync
//--- move B to SH -----------------------------------//
  .move(MatMul.B, SH, Move(B:8x128)(GL->SH)(Block)
    .tile(8, 16).to(Warp)
    .tile(8,  4).unroll
    .tile(1,  1).to(Thread).layout(ColMajor)
    .done).storageLayout(RowMajor).pad(4)
///// WARP-LEVEL ////////////////////////////////////////
  .tile(64,32).to(Warp)
///// THREAD-LEVEL //////////////////////////////////////
  .tile((4,32),(4,16)).to(Thread)
    .layout(ColMajor).swizzle(swizz)
  .split(1).unroll
// move A and B to RF--(omit Move details for brevity)-//
  .move(MatMul.A, RF, Move.tile(4,1).unroll.done)
  .move(MatMul.B, RF, Move.tile(1,4).unroll.done)
//--- perform computation using FMA ------------------//
  .tile(1,1).unroll.done//MatMul(1,1,1)(RF,RF,RF)(Thread)
```

```
val cudaWMMASample = MatMul(M,N,K)(GL,GL,GL)(Kernel)
///// BLOCK-LEVEL ////////////////////////////////////////
 .tile(128, 128).to(Block)
 .accumulateIn(FR, Init  // init: WMMA-Fragment for C
     .tile(64,32).to(Warp)
     .tile(16,16).unroll.done,
   Move(C:128x128)(FR->GL)(Block) // Epilog in 2 steps via SH
     .move(Move.src, SH, Move // Step 1: FR -> SH
       .tile(64,32).to(Warp)
       .tile(16,16).unroll.done)
     .tile(16, 128).to(Warp)  // Step 2: SH -> GL
     .tile(1,  128).unroll
     .tile(1,    4).to(Thread).done)
 .split(128).sync.unroll
//--- move A to SH -----------------------------------//
 .move(MatMul.A, SH, Move(A:128x128)(GL->SH)(Block)
   .tile(16,128).to(Warp)
   .tile(2, 128).unroll
   .tile(1,   8).to(Thread).done).noSync.pad(8)
//--- move B to SH -----------------------------------//
 .move(MatMul.B, SH, Move(B:128x128)(GL->SH)(Block)
   .tile(128,16).to(Warp)
   .tile(128, 2).unroll
   .tile(8,   1).to(Thread).layout(ColMajor).done).pad(8)
///// WARP-LEVEL ////////////////////////////////////////
 .tile(64, 32).to(Warp)
 .split(16).unroll
//--- fill WMMA fragments for A and B----------------//
 .move(MatMul.A, FR, Move.tile(16, 16).unroll.done)
 .move(MatMul.B, FR, Move.tile(16, 16).unroll.done)
//--- perform WMMA computation ----------------------//
 .tile(16,16).done // MatMul(16,16,16)(FR,FR,FR)(Warp)
```

Listing 3: Fireiron strategies expressing optimized matrix multiplication implementations targeting the Maxwell architecture (top) and using Tensor Cores by targeting executable WMMA specs (bottom).

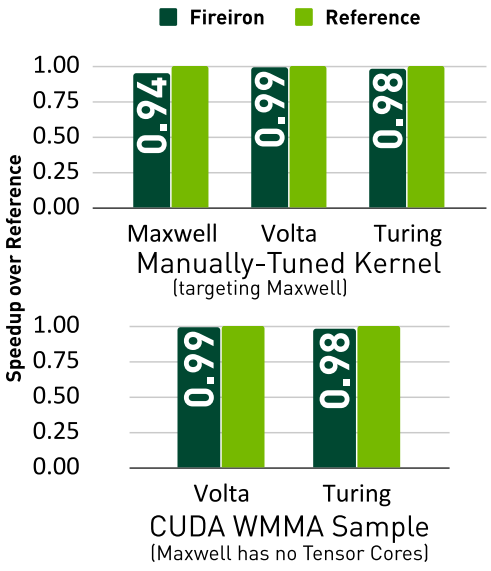


Figure 9: Comparing Fireiron generated code against two references. We achieve the same performance while requiring significantly less line of code.

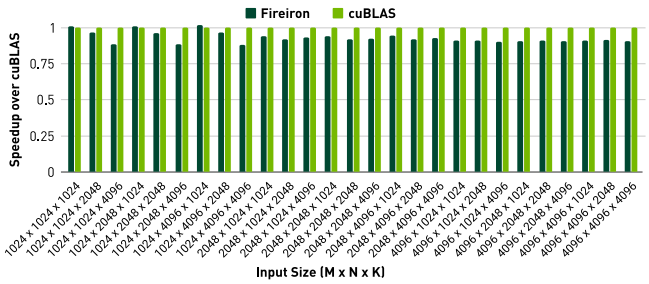


Figure 10: Comparing Fireiron-generated code against cuBLAS on large input matrices, both use Tensor Cores.

Hypothesis C: *Fireiron-generated code achieves performance close to expert-tuned code.*

Figure 9 shows the performance of our generated kernels using the `maxwell` schedule (left) and the `wmma` schedule (right) shown in Listing 3 compared to the reference kernels executed on multiple architectures. Here, we achieve exactly the same performance on Volta and Turing and come very close on the Maxwell architecture compared to the handwritten references while requiring significantly less lines of code.

cuBLAS provides the best implementations available written in optimized SASS assembly. It contains multiple differently optimized implementations and chooses one including tile sizes at runtime depending on the input sizes and hardware architecture based on internal heuristics. For a fair comparison, we use two parameterized strategies (one more suited for smaller, one for larger inputs) allowing to explore tile sizes (powers of two: 2^4–2^8), and report the best performance. Figure 10 shows the speedup compared to cuBLAS for large inputs. Here, we exactly match the performance in three

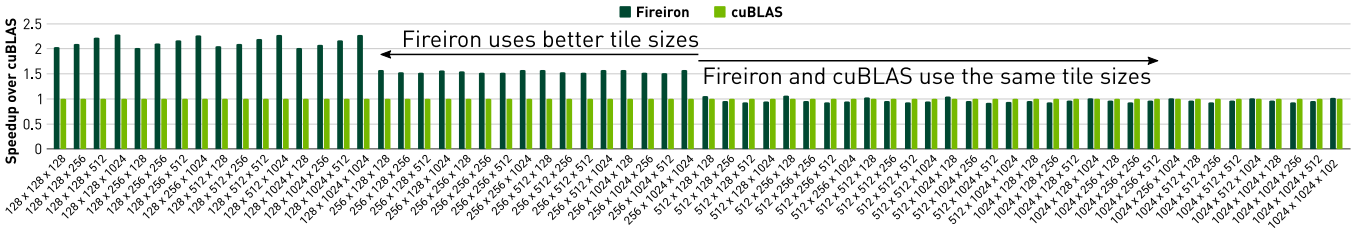


Figure 11: Comparing Fireiron-generated against cuBLAS on small input matrices, both use Tensor Cores.

cases and on average, we achieve 93.1% of the cuBLAS performance with minimum of 88.3% in one case and a maximum of 101% in two cases. This shows that Fireiron generates code performing close to the practically achievable peak.

Hypothesis D: *Fireiron's scheduling language is capable of expressing implementation strategies that generate code which outperforms the state-of-the-art.*

We were able to define strategies outperforming the manually optimized cuBLAS code by more than 2×.

Figure 11 shows the performance achieved compared to cuBLAS using small input sizes. We are able to significantly outperform cuBLAS on the smallest input sizes because there we use better tile sizes: We generally found a tile size 16×16 in the M and N dimensions and 64 in the K dimension, computed by two warps per block, to perform best. cuBLAS also chose 64 in the K dimension, but larger sizes for the M and N dimensions, which reduces the available parallelism.

Our high-level scheduling language allowed easy experimentation with different tile sizes. Changing tile sizes in a Fireiron schedule is simple (it requires changing two lines of code) whereas tile size exploration is a tedious and time-intensive process when kernels are developed in low-level assembly. There, changing tile sizes requires the adjustment of multiple complex index expressions throughout the whole kernel.

6 RELATED WORK

Fireiron is inspired by Halide [22, 23], TVM [7] and other scheduling languages [3, 6, 32], but no existing framework treats data movements as a first-class concept or allows to target Tensor Cores using `mma.sync`. Fireiron is also inspired by SPIRAL [21] in decomposing high-level specs using top-down transformations, however both SPIRAL and Spiral in Scala [19] automatically apply built-in rewrite rules, rather than giving control to the programmer.

MLIR [4, 14], (using the `linalg`-dialect), and CUTLASS [13, 16] achieve high matrix multiplication performance on GPUs by decomposing it similar to Fireion. However, CUTLASS provides a fixed set of hard-coded implementation strategies and neither provides a high-level scheduling language for experimenting with different decompositions.

Lift [10, 26], Multi-dimensional Homomorphisms [25], Tensor-Comprehension [29], and Futhark [12] are also aiming at high-performance code generation. In contrast to fixed compute specifications as used in Fireiron, they allow users to specify computations using a (functional) programming languages. We consider expressing computations in a programming language in the future.

Diesel [9], NOVA [8], and PPCG [31] make heavy use of the polyhedral model for optimization. Fireiron generates nested affine loops and might therefore profit using polyhedral techniques too. Additionally Fireiron strategies are similar to polyhedral schedule trees [30].

Auto-Tuning approaches including Halide's auto-tuners [1, 23], OpenTuner [2], ATF [24], and program synthesis techniques such as SwizzleInventor [20] aim to automatically develop optimized code using design space exploration. We aim to automatically synthesize Fireiron strategies in the future but in its current version it is designed as a tool for human performance experts.

7 CONCLUSION

In this paper we introduced Fireiron, a data-movement-aware scheduling language for GPUs. Treating data movements as first class concepts allows the precise description of high-performance GPU kernels as Fireiron strategies. We introduced specifications for both computations and data movements and decompositions to partially implement and map them to the multi-layered compute and memory hierarchies. Defining low-level PTX assembly as well as macro-instructions like WMMA as executable specs allows us to flexibly target specialized hardware like Tensor Cores.

Using different matrix multiplication implementations, we showed that Fireiron is able to express optimizations used in hand-tuned kernels written by experts. The code we generate generally matches the performance of hand-tuned implementations and experts are able to use Fireiron to improve the state-of-the-art by outperforming vendor libraries by more than 2×.

ACKNOWLEDGMENTS

We thank Martin Lücke and Thomas Koehler for their support with comparing against TVM and Halide. We also thank Julien Demouth for his support with targeting Tensor Cores. This work has been supported in part by the NSF Grants ACI OAC–1535191, FMitF CCF-1918027, OIA-1936731, by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352), the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01, grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm. The first author was financially supported by an NVIDIA Graduate Fellowship.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12. https://doi.org/10.1145/3306346.3322967

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*. 303–316. https://doi.org/10.1145/2628071.2628092

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

[4] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* abs/2003.00532 (2020).

[5] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (2007), 291–312.

[6] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Citeseer.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom, June 12-13, 2014*. 8–13. https://doi.org/10.1145/2627373.2627375

[9] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 42–51. https://doi.org/10.1145/3211346.3211354

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*. 100–112. https://doi.org/10.1145/3168824

[11] Halide. 2020. MatMulGenerator. https://github.com/halide/Halide/blob/master/apps/cuda_mat_mul/mat_mul_generator.cpp

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 556–571. https://doi.org/10.1145/3062341.3062354

[13] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. 2018. Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs. *CoRR* abs/1808.07984 (2018).

[14] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020).

[15] NVIDIA. 2020. CUDA WMMA Sample Kernel - CUDA Tensor Core GEMM. https://github.com/NVIDIA/cuda-samples/blob/master/Samples/cudaTensorCoreGemm/cudaTensorCoreGemm.cu

[16] NVIDIA. 2020. CUTLASS: Fast Linear Algebra in CUDA C++. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[17] NVIDIA. 2020. Programming Tensor Cores. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

[18] NVIDIA. 2020. PTX - MMA Instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma

[19] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*. 125–134. https://doi.org/10.1145/2517208.2517228

[20] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 65–78. https://doi.org/10.1145/3297858.3304059

[21] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[22] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 519–530. https://doi.org/10.1145/2491956.2462176

[24] Ari Rasch and Sergei Gorlatch. 2019. ATF: A generic directive-based auto-tuning framework. *Concurr. Comput. Pract. Exp.* 31, 5 (2019).

[25] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *PACT*. IEEE, 354–369.

[26] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 74–85.

[27] TVM. 2020. How to optimize GEMM on CPU. https://docs.tvm.ai/tutorials/optimize/opt_gemm.html

[28] TVM. 2020. How to optimize matmul with Auto TensorCore CodeGen. https://docs.tvm.ai/tutorials/optimize/opt_matmul_auto_tensorcore.html

[29] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[30] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule Trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.

[31] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *TACO* 9, 4 (2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[32] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491