Systematically Extending a High-Level Code Generator with Support for Tensor Cores

Lukas Siefke
University of Münster
Münster, Germany
lukas.siefke@wwu.de

Bastian Köpcke
University of Münster
Münster, Germany
bastian.koepcke@wwu.de

Sergei Gorlatch
University of Münster
Münster, Germany
gorlatch@wwu.de

Michel Steuwer
The University of Edinburgh
Edinburgh, United Kingdom
michel.steuwer@ed.ac.uk

Abstract

High-level code generators like Halide, Lift, and R**ISE** make a compelling proposition: write programs in a simple high-level language and get high-performing GPU code “for free”. They achieve this feat by restricting the input language to a specific domain (such as image and array processing in Halide) or to a fixed set of flexible parallel patterns (as Lift and R**ISE** do). Implementing high-level code generators that produce high-performance code is challenging, specifically as the target hardware constantly evolves.

In this paper, we discuss how we systematically extend the R**ISE** high-level code generator with support for tensor cores, a specialized hardware feature of recent Nvidia GPUs. We highlight the design of R**ISE** that makes it easily extensible by following a systematic bottom-up approach, that *first*, exposes the imperative tensor core API to the code generator, *then*, raises the abstractions to an internal low-level functional representation, that, *finally*, is targeted by a rewrite process that starts from a high-level functional program.

Our experimental evaluation shows that R**ISE** with support for tensor cores generates code of competitive performance to manually optimized CUDA code, which is only up to 36%, but on average only 10%, slower than Nvidia’s highly optimized `cuBLAS` library, and clearly outperforms any code that does not exploit tensor cores.

1 Introduction

We are in a “new golden age of computer architecture” [5] where performance and efficiency gains are made by specializing the hardware architecture. Hardware changes quickly, as vendors develop specialized hardware solutions to accelerate common or important computational patterns. General Matrix Multiplication (GEMM) is one of the most important computations performed in high-performance computing, and is used extensively in areas such as physics, statistics, and – maybe currently most importantly – machine learning. Recently, many specialized hardware designs for accelerating deep learning workloads, mostly focused on GEMM, have emerged. This includes Google’s TPUs, Graphcore’s AI accelerator, and more [13]. In this paper, we are focusing on Nvidia’s tensor cores that are an addition to their existing GPU architectures, offering support to efficiently perform GEMM computations with low floating-point precision.

While this new hardware promises high-performance gains, programming GPUs manually – even without tensor cores – is error-prone and difficult due to their high degree of parallelism and a complex memory hierarchy that programmers have to manage manually. This leads to a mix of high-level algorithmic ideas, such as tiling a matrix multiplication, with many low-level implementation details making reasoning about the correctness and efficiency of programs difficult. To efficiently exploit tensor cores, we need to adapt the high-level algorithm and perform low-level management as well. We need to break up GEMM into smaller operations each to be performed efficiently by tensor cores, and combining many partial results into the single output. Despite these challenges, it is highly desirable to utilize GPUs with tensor cores for significant performance gains, e.g., up to 4× improved performance for single precision GEMM [9].

A promising attempt to simplify GPU programming are high-level libraries as well as high-level languages and their code generators. While high-level libraries, such as `cuBLAS`, `Thrust`, `Tensorflow`, and `PyTorch` are easy to use and popular, they are also limited as they restrict programmers to a narrow interface. High-level GPU programming languages allow programmers to express more applications, but have the challenge of compiling them to efficient code. Examples of GPU array processing languages are Futhark [6], Dex [11], Accelerate [1], and Halide [12]. None of these languages have added support for tensor cores, despite the considerable performance gains available. Halide has an issue open on GitHub[1] since December 2019 that discusses some restricted implementation strategies, but still no support has been implemented.

GPGPU '22, April 3, 2022, Virtual, Republic of Korea
2022.

[1]https://github.com/halide/Halide/issues/4481

In this paper, we extend the high-level language RISE [4] and its Shine compiler [16] with support for tensor cores. RISE has a multilayered design of different abstraction levels from hardware to high-level algorithms and focuses on extensibility [7]. This facilitates the main idea of this paper: systematically expose tensor cores to non-expert programmers while still generating efficient code.

Our contributions are as follows:

- We present a systematic bottom-up approach to extend the high-level RISE code generator, by:
  1. exposing the tensor core API in the imperative abstraction layer of the code generator;
  2. defining low-level functional tensor core patterns and a translation from them to the imperative level;
  3. providing rewrite rules that enable high-level functional programs to be rewritten into low-level programs that exploit tensor cores.
- We evaluate the code generated by our extended RISE code generator and show performance competitive with handwritten CUDA code exploiting tensor cores. The generated code is up to 36%, and on average only 10% slower than the highly optimized cuBLAS, but much faster than any code that does not exploit tensor cores.

2 Example: GEMM in RISE

In this section, we introduce the RISE language and its compiler Shine using a GEMM example application and a subset of exemplary primitives.

2.1 Expressing GEMM in RISE

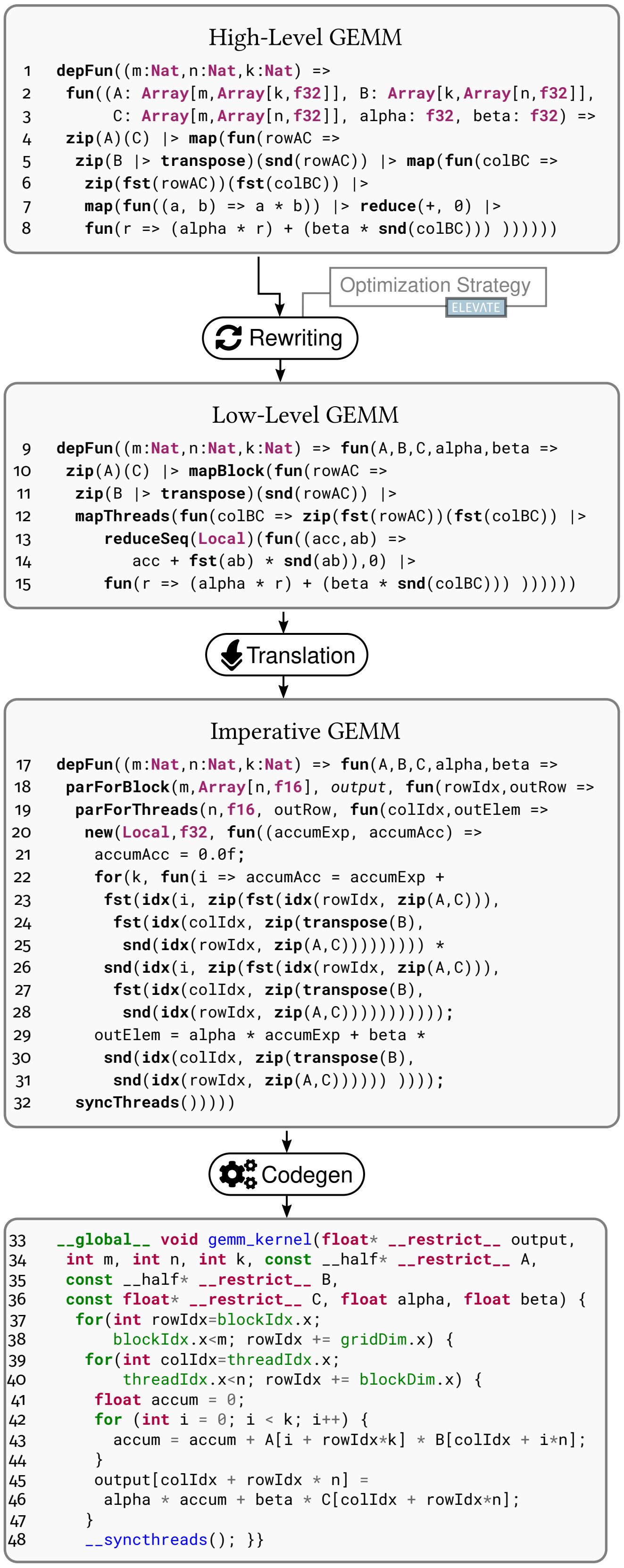
Lines 1–8 in Figure 1 show the Generalized Matrix Multiplication (GEMM) expressed in RISE. This is an example of a high-level program that describes only what to compute, without specifying how to map computation to the hardware. This choice is left to the compiler and allows flexible generation of high-performance code for different hardware from the same high-level program.

GEMM describes the following computation:

$$D = \alpha AB + \beta C$$

Here, matrices A and B are multiplied and scaled with the scalar value α and matrix C scaled by scalar value β is added to form the result matrix D.

In the RISE program, the types of matrices A, B, and C track their dimensions as seen in line 2 and 3. This makes sure that only matrices with matching dimensions are multiplied. The computation is expressed in lines 4–8. The familiar matrix multiplication is expressed in terms of the computational patterns `zip`, `map`, and `reduce`. Focusing first on the matrix multiplication AB, we can read the code as follows: for each row of A and each column of B we compute the dot product. The dot product is expressed in lines 6 and 7 by: pointwise combining row and column using `zip`, then



```
1  depFun((m:Nat,n:Nat,k:Nat) =>
2   fun((A: Array[m,Array[k,f32]], B: Array[k,Array[n,f32]],
3        C: Array[m,Array[n,f32]], alpha: f32, beta: f32) =>
4   zip(A)(C) |> map(fun(rowAC =>
5    zip(B |> transpose)(snd(rowAC)) |> map(fun(colBC =>
6     zip(fst(rowAC))(fst(colBC)) |>
7     map(fun((a, b) => a * b)) |> reduce(+, 0) |>
8     fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

```
9  depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
10  zip(A)(C) |> mapBlock(fun(rowAC =>
11   zip(B |> transpose)(snd(rowAC)) |>
12   mapThreads(fun(colBC => zip(fst(rowAC))(fst(colBC)) |>
13      reduceSeq(Local)(fun((acc,ab) =>
14         acc + fst(ab) * snd(ab)),0) |>
15      fun(r => (alpha * r) + (beta * snd(colBC))) ))))))
```

```
17 depFun((m:Nat,n:Nat,k:Nat) => fun(A,B,C,alpha,beta =>
18  parForBlock(m,Array[n,f16], output, fun(rowIdx,outRow =>
19   parForThreads(n,f16, outRow, fun(colIdx,outElem =>
20    new(Local,f32, fun((accumExp, accumAcc) =>
21     accumAcc = 0.0f;
22     for(k, fun(i => accumAcc = accumExp +
23      fst(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
24       fst(idx(colIdx, zip(transpose(B),
25        snd(idx(rowIdx, zip(A,C))))))))) *
26      snd(idx(i, zip(fst(idx(rowIdx, zip(A,C))),
27       fst(idx(colIdx, zip(transpose(B),
28        snd(idx(rowIdx, zip(A,C)))))))))));
29     outElem = alpha * accumExp + beta *
30      snd(idx(colIdx, zip(transpose(B),
31       snd(idx(rowIdx, zip(A,C)))))) ))));
32   syncThreads()))))
```

```
33 __global__ void gemm_kernel(float* __restrict__ output,
34  int m, int n, int k, const __half* __restrict__ A,
35  const __half* __restrict__ B,
36  const float* __restrict__ C, float alpha, float beta) {
37   for(int rowIdx=blockIdx.x;
38       blockIdx.x<m; rowIdx += gridDim.x) {
39    for(int colIdx=threadIdx.x;
40        threadIdx.x<n; rowIdx += blockDim.x) {
41     float accum = 0;
42     for (int i = 0; i < k; i++) {
43       accum = accum + A[i + rowIdx*k] * B[colIdx + i*n];
44     }
45     output[colIdx + rowIdx * n] =
46      alpha * accum + beta * C[colIdx + rowIdx*n];
47    }
48    __syncthreads(); }}
```

Figure 1. Compilaton of GEMM in RISE.

multiplying each pointwise pair using **map**, and, finally, computing the sum using **reduce**. To generalize this description to GEMM, we scale the computed element with `alpha` and add the element from `C` that has been scaled with `beta` (line 8).

2.2 Compiling = Rewriting + Translation + Codegen

Compiling a high-level program to efficient code is a complex process that involves performing optimizations as well as lowering the abstractions from the high-level functional input to the low-level imperative output. The Shine compiler breaks this process into three distinct steps: *1)* optimizations are applied in a *rewrite process*, transforming the high-level function to a low-level functional program that has all optimization decisions explicitly encoded; *2)* the low-level functional program is *translated* into an imperative representation; and *3)* the target *code is generated*.

Rewriting. The high-level input program is rewritten into a low-level program using semantics-preserving rewrite rules. Many individual rules are combined into larger *optimization strategies*, which can be expressed in a separate strategy language called ELEVATE [4]. The rewriting can either be manually controlled by a performance expert [7], fully automated [15], or an automated process can be guided by the expert [8]. Lines 9–15 in Figure 1 show one possible low-level program for GEMM that has been obtained from the high-level program via rewriting. Generic **map** primitives have been rewritten to **mapBlock** and **mapThreads** which express that each element of the input array has to be processed by a thread block or a single thread, respectively. **mapThreads** is nested inside **mapBlock**, meaning that we describe what every thread block computes in terms of what every thread in the thread block computes. Therefore, the input array can be viewed as being distributed over thread blocks and threads, which then process their share of the data. For GEMM, every thread in a thread block, performs a sequential dot product on the block's row from the A matrix and the thread's column of the B matrix (scaled by `alpha`) and adds the relevant element from the C matrix (scaled by `beta`) to the result. The parameter **Local** in **reduceSeq** indicates that the fast thread local memory of the GPU is used for the accumulator variable.

Translation. To generate CUDA code, low-level programs are deterministically translated from the low-level functional into an intermediate imperative representation. This representation includes parallel for-loops, memory allocations, synchronizations, and fully inlined anonymous functions. Primitives that affect how arrays are indexed, e.g., **transpose**, still exist and are resolved in the final codegen step.

Codegen. During the code generation step, these primitives are compiled into index expressions. The rest of the code generation is straightforward and generates CUDA equivalents to the parallel for-loops and memory allocations.

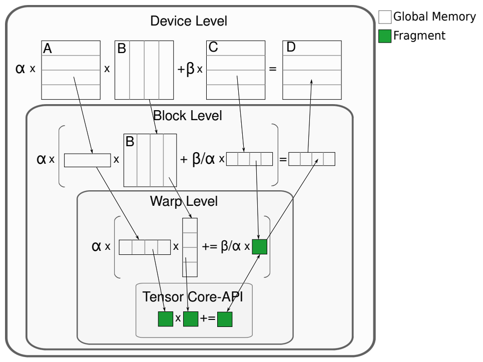


Figure 2. Schematic representation of a naive GEMM on a GPU using tensor cores.

3 Towards Tensor Cores in RISE

Tensor cores are specialized hardware to speed up matrix multiplications. The API for tensor cores enables threads of a single warp to cooperate for using multiple tensor cores concurrently, to perform a *Matrix Multiply and Accumulate* (MMA) computation on three matrices of fixed sizes.

Utilising Tensor Cores for GEMM. To use tensor cores, a GEMM implementation must be broken down into many small MMA computations, using tiling. Figure 2 shows a schema for an example implementation of GEMM, using tensor cores. Instead of computing a dot-product per thread like in Figure 1, the matrices are broken into blocks of multiple rows or columns. Every thread block computes GEMM between a block of A rows, the entire B matrix, and a block of rows in the C matrix. This is implemented by splitting each row and column block into tiles that fit into a warp-wide tensor core computation. This enables each warp to perform a dot product over the matrix tiles (instead of scalar elements) and to add the corresponding C tile.

Programming Tensor Cores. To use tensor cores, tiles must be stored in a specific format in the warp's registers. In CUDA, this format is represented by the `fragment` type.

The left-hand side of Listing 1 shows the fragment type as well as important functions of the tensor core API. A fragment has several attributes that must all be statically known. The kind of the fragment (**FragmKind**) clarifies whether the tile is used as the A, B or accumulator matrix. Dimensions m, n, k specify all the dimensions that are used in a warp-wide tensor core operation. This means that a fragment "knows" about the dimension of the other fragments. Type parameter T is a placeholder for a set of scalar data types that tensor cores can work with, and the layout specifies how a matrix is stored in memory (i.e., row major or column major). Function `mma_sync` performs a warp-wide MMA operation on fragments A, B, C and stores the result in D. Functions `load_matrix_sync` and `store_matrix_sync` are used to convert matrix tiles to and from fragments. They take a reference to the fragment and a

```
template<typename FragmKind, int m, int n, int k,
 typename T, typename Layout=void> class fragment;

void mma_sync(
  fragment<...> &D,
  const fragment<...> &A,
  const fragment<...> &B,
  const fragment<...> &C);
void load_matrix_sync(fragment<...> &A,
  const T* tile, unsigned l_dim, layout_t layout);
void store_matrix_sync(T* tile,
  const fragment<...> &A,
  unsigned l_dim, layout_t layout);
void fill_fragment(
  fragment<...> &A, const T& value);
```

```
Fragment[m: Nat, n: Nat, k: Nat, t: DataType, f: FragmKind]

def mmaFragment(m:Nat, n:Nat, k:Nat, s:DataType, t:DataType,
  A: Exp[Fragment[m,k,n,s,AMatrix], Rd],
  B: Exp[Fragment[k,n,m,s,BMatrix], Rd],
  C: Exp[Fragment[m,n,k,t,Accum], Rd],
  D: Acc[Fragment[m,n,k,t,Accum]]): Comm
def loadFragment(f:FragmKind, m:Nat, n:Nat, k:Nat, t:DataType,
  tile: Exp[Array[m,Array[n,t]], Rd], A: Acc[Fragment[m,n,k,t,f]]): Comm
def storeFragment(m:Nat, n:Nat, k:Nat, t:DataType,
  A: Exp[Fragment[m,n,k,t,Accum],Rd], tile: Acc[Array[m,Array[n,t]]]): Comm
def fillFragment(f:FragmKind, m:Nat, n:Nat, k:Nat, t:DataType,
  A: Acc[Fragment[m,n,k,t,f]], value: Exp[t, Rd]): Comm
```

Listing 1. The tensor core API compared to imperative primitives for tensor cores.

pointer to the tile. The leading dimension is used to load or store matrices that have additional elements between their rows or columns that can be ignored, e.g., when loading tiles. Finally, function `fill_fragment` creates a fragment where every element has the same specified value.

4 Systematically Exposing Tensor Cores

In this section, we show how we develop RISE abstractions for tensor cores in a systematical way. We choose a bottom-up approach and begin by developing low-level imperative primitives that represent the tensor core API. Then, we show how the imperative functionality is brought into the functional world by developing corresponding low-level functional primitives and a translation from functional to imperative. Using this approach, we increase the abstraction level incrementally. This approach significantly simplifies determining the requirements of the next higher abstraction.

4.1 Imperative Primitives

Listing 1 shows the CUDA tensor core API on the left and on the right-hand the newly introduced type and primitives of RISE. Analogously to the CUDA version of a fragment, the fragment type in RISE is constructed from matrix dimensions, the data type of their elements and the fragment kind. The matrix layout is not explicitly stored in the type and instead inferred during code generation. Similarly, the leading dimension (`l_dim` in the CUDA API) is inferred where needed. We do not have to deal with the matrix layout or leading dimension as this information can easily be inferred.

The types of the primitives consist of a list of parameter types and a return type. In RISE `Comm` takes the role of `void`. The `mmaFragment` primitive is generic over matrix dimensions and data types, and accepts the same arguments as the corresponding CUDA function. However, in the types, we differentiate between acceptors (`Acc`) and expressions (`Exp`). Acceptors represent memory locations that are write-only and expressions are evaluated to a value that can only be read if the expression has a read (`Rd`) annotation; otherwise they have to be written into to memory first. Therefore – ignoring the generic parameters – `mmaFragment` takes four arguments: three expressions representing the fragments of matrices A, B, and C, and an acceptor representing the fragment of matrix D in which to store the computed result. The types of the other primitives are designed accordingly.

```
tensorMatMulAdd: {m: Nat} -> {n: Nat} -> {k: Nat} ->
  {s: DataType} -> {t: DataType} ->
  Fragment[m,k,n,s, AMatrix] ->
  Fragment[k,m,n,s, BMatrix] ->
  Fragment[m,n,k,t, Accum] -> Fragment[m,n,k,t, Accum]
asFragment: {m: Nat} -> {n: Nat} -> {k: Nat} ->
  {t: DataType} -> {f: FragmKind} ->
  Array[m, Array[n, t]] -> Fragment[m,n,k,t, f]
asMatrix: {m: Nat} -> {n: Nat} -> {k: Nat} -> {t: DataType} ->
  Fragment[m,n,k,t, Accum] -> Array[m, Array[n, t]]
generateFragment: {m: Nat} -> {n: Nat} -> {k: Nat} ->
    {t: DataType} -> {f: FragmKind} ->
    t -> Fragment[m,n,k,t, f]
mapFragment: {m: Nat} -> {n: Nat} -> {k: Nat} ->
  {t: DataType} -> {f: FragmKind} ->
  Fragment[m,n,k,t,f] -> (t -> t) -> Fragment[m,n,k,t, f]
```

Listing 2. Functional low-level primitives for tensor cores.

4.2 Low-Level Functional Primitives

Our next step is to introduce low-level functional primitives as a target for rewriting, together with a translation to the previously introduced imperative primitives.

Listing 2 shows the tensor core specific functional low-level RISE primitives. In contrast to the imperative primitives, these primitives are functional and, therefore, have a non-`void` return type. Primitive `tensorMatMulAdd` is similar to its imperative counterpart `mmaFragment`. It is generic over matrix dimensions and data types, has parameters for the three input fragments, but returns the computed fragment instead of storing it to memory as its imperative counterpart. Primitives `asFragment` and `asMatrix` are used to convert a 2-dimensional array to and from a fragment type. `generateFragment` creates a fragment that is filled with a single value. `mapFragment` maps a function to every element of a fragment, where the mapping

```
def accT(expr: Phrase[Exp[d,Wr]],
         output: Phrase[Acc[t]]): Phrase[Comm] = expr match {
case tensorMatMulAdd(m,n,k,dt,dtAcc,aMatrix,bMatrix,cMatrix)
  => conT(aMatrix, fun(aMatrix => conT(bMatrix,
  fun(bMatrix => conT(cMatrix, fun(cMatrix =>
    mmaFragment(m, n, k, dt,
     dtAcc, aMatrix, bMatrix, cMatrix, A)))))))
case asFragment(m, n, k, dt, f, tile)
  => conT(tile, fun(tile: =>
   loadFragment(f, m, n, k, dt, tile, A)))
case asMatrix(m, n, k, dt, frag)
  => conT(frag, fun(frag: =>
   storeFragment(m, n, k, dt, frag, A)))
case generateFragment(m, n, k, dt, f, fill)
  => conT(fill, fun(fill =>
   fillFragment(f, m, n, k, dt, fill, A)))
case mapFragment(m, n, k, dt, f, function, input)
  => conT(input, fun(input: =>
   forFragment(m, n, k, dt, f,
     input, A, fun(element => fun(acceptor =>
         accT(function(element), acceptor))))))
... }
```

Listing 3. Translation from functional to imperative.

is cooperatively performed by the threads (lanes) in a warp. This is, for example, used to scale fragments with α or β.

4.3 Translating from Functional to Imperative

We extend the translation from functional to imperative of RISE to support our new primitives. Listing 3 shows one of RISE's two translation function (*accT*), which creates imperative code that evaluates a low-level functional expression and writes the result into an `output` acceptor. For example, to translate **tensorMatMulAdd**, we start by generating the imperative code for each input matrices. For this, we call the second translation function (*conT*). Intuitively, the *conT* takes a functional expression, generates imperative code which evaluates the expression, and passes a handle for the result to a continuation function that integrates the result into the following imperative code. In the generated code for **tensorMatMulAdd**, we pass handles for the generated inputs and the result acceptor to the imperative tensor core primitive.

4.4 Targeting the Tensor Primitives via Rewriting

Finally, we have everything we need to write programs that use tensor cores in low-level RISE. However, the goal is for programmers not to write any low-level code and instead to discover implementations that use tensor cores during rewriting of high-level programs. We, therefore, need rewrite rules that explain how tensor cores can be utilized. Listing 4 shows a rewrite rule that takes a high-level functional program expressing matrix multiplication and rewrites it into an equivalent low-level program that uses tensor cores. The rule makes sure that the input matrices at the beginning of the rewrite rule have types that are compatible with the tensor core primitive. More complex rules in this style are similarly introduced for GEMM and other use cases.

```
aTile: Array[16,Array[16,f16]] |> map(fun(aRow =>
 bTile: Array[16,Array[16,f16]] |> map(fun(bCol =>
  zip(aRow, bCol) |>
  reduceSeq(fun(ac, ab =>
   add(ac, mul(fst(ab), snd(ab)))))(0.0)))))
```

↧

```
tensorMatMulAdd
 (aTile: Array[16,Array[16,f16]] |> asFragment |> toMem(Local))
 (bTile: Array[16,Array[16,f16]] |> transpose
  |> asFragment |> toMem(Local))
 (generateFragment(0.0) |> toMem(Local))
|> toMem(Local) |> asMatrix
```

Listing 4. A rewrite rule that replaces a 16×16 matrix multiplication with a warp-wide tensor core operation.

5 Experimental Evaluation

In this section, we evaluate our abstractions by generating parallel GEMM in CUDA and comparing its performance to manual naive and optimized implementations as well as the state-of-the art cuBLAS library.

Experimental Setup. All measurements were performed on CentOS 7.9.2009 with CUDA 11.1.1, with two different GPUs: A professional level TITAN RTX and a high-end consumer level GeForce RTX 2080 Ti. We compare mixed-precision GEMM (elements of A and B are of half precision floating-point values; other values are single precision) on square matrices, and choose some arbitrary but fix non-zero values for α and β. We assume that the B matrix is transposed in memory. Additionally, cuBLAS requires that the C matrix is transposed as well. All input matrices are initialized with random values. Input dimensions are chosen such that they are dividable by tile sizes, and tile sizes such that they are dividable by the fragment size of 16. With RISE, we generate kernels for every matrix dimension and GPU and try different configurations of thread blocks and tiling sizes from a set of manually chosen values to select the fastest one.

We compare the performance to several other implementations: naive handwritten versions without and with tensor cores (similar to the kernels described in Figure 1 and Figure 2, respectively), an optimized kernel with tensor cores that is taken from the CUDA samples and cuBLAS without and with tensor cores. All these kernels were adapted to match our setting (such as using the same data types). The naive handwritten kernels only work for small to medium-sized matrices, as the simple parallelization method exceed the physical GPU resources for larger matrices.

Results. We measure runtime by executing every kernel 50 times and plot the median floating-point-operations per second in TFLOPS. Our results in Figure 3 show that using tensor cores for GEMM is hugely beneficial and can increase performance up to 7×. This has been observed before, e.g., in [9] with a similar performance trend across input sizes.

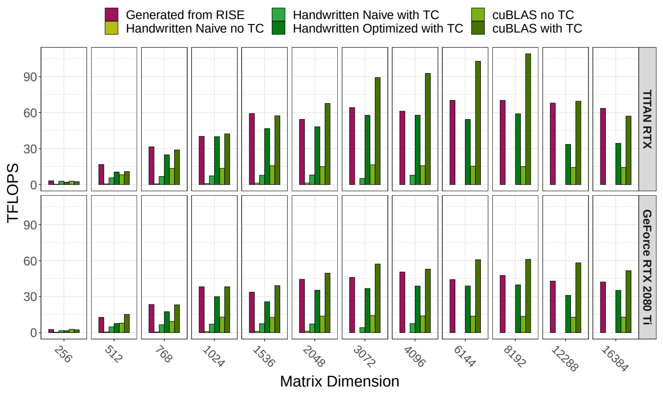


Figure 3. Performance comparison of different GEMM implementations. All matrix sizes are squared. The x-axis shows the matrix dimensions, the y-axis show floating point operations per second in TFLOPS (higher is better).

Our generated kernels are faster than all the handwritten kernels, including the ones exploiting tensor cores. The reason for the generated code being faster than the optimized handwritten kernel is that the generated kernels are easily tuned for different thread block and tiling sizes, while the manual versions assume specific configurations and tuning them, in particular for larger input sizes, requires changes to how the kernels operate. Our generated code clearly outperforms the cuBLAS version that does not use tensor cores, and is at least 64% as fast as the professionally optimized (e.g., exploiting assembly level instructions) cuBLAS implementation with tensor cores. We show that our generated code can even be faster than cuBLAS (by up to 54%), possibly by choosing a better tiling strategy than cuBLAS.

6 Related Work

We already mentioned the functional array languages Accelerate [1], Futhark [6], Dex [11], and Halide [12] in Section 1. These languages are designed to simplify high-performance GPU programming. Nonetheless, to our knowledge, none supports utilizing tensor cores.

Fireiron [3] is a language for describing implementations of matrix multiplication. It supports tensor cores and is able to generate highly performant code that is on par with cuBLAS. However, Fireiron is less generic than our approach and does not support automatic rewriting, making it a tool for expert engineers.

Tensor cores can be utilized for algorithms aside from matrix multiplication. It has been demonstrated that tensor cores can be used to accelerate reductions and scan operations [2, 10] or to implement Fast Fourier Transforms [14]. While these show interesting applications for us to investigate in the future, they focus on manual low-level implementations instead of our higher-level abstractions.

7 Conclusion

In this paper, we have shown how to extend the RISE language to support tensor cores. We demonstrated that its compiler design enables us to systematically develop abstractions by starting with a direct representation of an imperative primitive and step-by-step introducing machinery for higher-level abstractions. Our evaluation shows, that this approach generates high-performance code for matrix multiplication competitive to manually optimized CUDA code and is only up to 36% slower than the highly optimized cuBLAS.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP*.
[2] Abdul Dakkak, Cheng Li, Isaac Gelado, Jinjun Xiong, and Wen-Mei W. Hwu. 2018. Accelerating Reduction and Scan Using Tensor Core Units. arXiv:1811.09736
[3] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodík, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *PACT*.
[4] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. In *ICFP*.
[5] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019).
[6] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely func. GPU-programming with nested parallelism and in-place array updates. In *PLDI*.
[7] Thomas Koehler and Michel Steuwer. 2021. Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs. In *CGO*.
[8] Thomas Koehler, Phil Trinder, and Michel Steuwer. 2021. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings. arXiv:2111.13040
[9] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *IPDPS Workshops*.
[10] Cristóbal A. Navarro, Roberto Carrasco, Ricardo J. Barrientos, Javier A. Riquelme, and Raimundo Vega. 2021. GPU Tensor Cores for Fast Arithmetic Reductions. *IEEE TPDS* 32, 1 (2021).
[11] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the point: index sets and parallelism-preserving autodiff for pointful array prog.. In *ICFP*.
[12] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018).
[13] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2021. AI Accelerator Survey and Trends. In *HPEC*.
[14] Anumeena Sorna, Xiaohe Cheng, Eduardo F. D'Azevedo, Kwai Wong, and Stanimire Tomov. 2018. Optimizing the FFT Using Mixed Precision on Tensor Core Hardware. In *HiPCW Workshops*.
[15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating perf. portable code using rewrite rules: from high-level func. expr. to high-perf. OpenCL code. In *ICFP*.
[16] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design. arXiv:2201.03611