Programming Tensor Cores from an Image Processing DSL

Savvas Sioutas
Eindhoven University of Technology
Eindhoven, The Netherlands
s.sioutas@tue.nl

Sander Stuijk
Eindhoven University of Technology
Eindhoven, The Netherlands
s.stuijk@tue.nl

Twan Basten
Eindhoven University of Technology
TNO - ESI
Eindhoven, The Netherlands
a.a.basten@tue.nl

Lou Somers
Canon Production Printing
Eindhoven University of Technology
Eindhoven, The Netherlands
l.j.a.m.somers@tue.nl

Henk Corporaal
Eindhoven University of Technology
Eindhoven, The Netherlands
h.corporaal@tue.nl

ABSTRACT

Tensor Cores (TCUs) are specialized units first introduced by NVIDIA in the Volta microarchitecture in order to accelerate matrix multiplications for deep learning and linear algebra workloads. While these units have proved to be capable of providing significant speedups for specific applications, their programmability remains difficult for the average user. In this paper, we extend the Halide DSL and compiler with the ability to utilize these units when generating code for a CUDA based NVIDIA GPGPU. To this end, we introduce a new scheduling directive along with custom lowering passes that automatically transform a Halide AST in order to be able to generate code for the TCUs. We evaluate the generated code and show that it can achieve over 5x speedup compared to Halide manual schedules without TCU support, while it remains within 20% of the NVIDIA cuBLAS implementations for mixed precision GEMM and within 10% of manual CUDA implementations with WMMA intrinsics.

CCS CONCEPTS

• **Computer systems organization** → *Embedded systems*; • **Software and its engineering** → **Compilers**; **Domain specific languages**.

KEYWORDS

GPGPUs, tensor cores, Halide, matrix multiplication

ACM Reference Format:
Savvas Sioutas, Sander Stuijk, Twan Basten, Lou Somers, and Henk Corporaal. 2020. Programming Tensor Cores from an Image Processing DSL. In *23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES '20), May 25–26, 2020, Sankt Goar, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3378678.3391880

SCOPES '20, May 25–26, 2020, Sankt Goar, Germany

ACM ISBN 978-1-4503-7131-5/20/05…$15.00
https://doi.org/10.1145/3378678.3391880

1 INTRODUCTION

Matrix multiplication (GEMM) has proven to be an integral part of many applications in the image processing domain [8]. With the rise of CNNs and other Deep Learning applications, NVIDIA designed the Tensor Core Unit (TCU). TCUs are specialized units capable of performing 64 (4x4x4) multiply - accumulate operations per cycle. When first introduced alongside the Volta microarchitecture, these TCUs aimed to improve the performance of mixed precision multiply-accumulates (MACs) where input arrays contain half precision data and accumulation is done on a single precision output array. With the newer Turing architecture, TCUs also support fixed precision MACs as well as more data types compared to the previous generation.

Although TCUs can significantly increase the performance of applications such as DNNs and other tensor contractions whose main workloads can be formulated as matrix multiplications, direct programmability of these units remains either inaccessible to non CUDA experts, or completely hidden behind libraries such as cuBLAS and CUTLASS.

Halide [13] is a Domain Specific Language (DSL) for image processing applications that aims to increase code portability and readability by separating the functional description of an application from its optimization schedule. Using LLVM [10] as a backend compiler, Halide can target various architectures including multi-core CPUs as well as CUDA based GPGPUs. These multi-core CPUs can often act as a host while parts of the code are offloaded into a GPU which acts as an accelerator.

In this work we extend the Halide DSL with the `tensor_core` scheduling directive, along with all necessary lowering and backend compiler passes in order to allow the compiler to automatically utilize the TCUs when asked by the user. To this end, we implement custom lowering passes that replace the parts of the AST that correspond to the traditional matrix multiplication and inject calls to new compiler intrinsics that correspond to tensor operations. Furthermore, using NVVM as a backend, we extend the PTX (Parallel Thread Execution) code generator for each of the new intrinsics. Finally, we demonstrate that through our extensions, the compiler can automatically generate a highly optimized implementation of GEMM without the user having to worry about data types, loop bounds or other scheduling choices. Experimental results show that the performance of the generated code is over 5x faster than

manually tuned Halide schedules without tensor core support, and close to or even faster than NVIDIA cuBLAS implementations.

The rest of this work is organized as follows: Section 2 presents background information on the Halide DSL and the NVIDIA tensor core architecture, focusing on its programmability. Section 3 discusses related work on compiler support for similar architectures across various DSLs. Section 4 introduces the new scheduling directive along with all necessary compiler passes that enable code generation for TCUs in Halide, along with an example optimization schedule that was used for benchmarking. Section 5 evaluates the performance of generated code based on the aforementioned schedule compared to equivalent cuBLAS implementations as well as manually scheduled Halide implementations without tensor core support. Finally, concluding remarks are made in Section 6.

2 BACKGROUND INFORMATION

This section presents key background information on the NVIDIA Tensor Core architecture and its programmability, as well as on the Halide DSL and compilation flow.

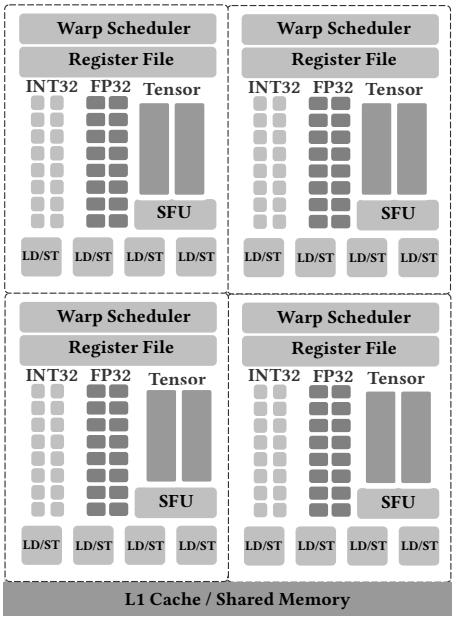

Figure 1: Simplified view of the Turing SM microarchitecture. Each SM sub-core contains 2 Tensor Cores capable of executing 64 multiply/accumulate operations per cycle. Warps in each subcore can utilize these units and can communicate through the shared memory.

2.1 The NVIDIA Tensor Core Architecture

The NVIDIA Tensor Core Unit [7] (TCU) was first introduced alongside the Volta architecture and is also present in the Turing microarchitecture. A simplified model of the Turing Streaming Multiprocessor (SM) microarchitecture can be seen in Figure 1. Each SM contains four sub-cores (processing units). Sub-cores contain two processing units, each capable of executing $4x4x4$ multiply/accumulate per cycle. This translates to 128 operations per cycle for every TCU. On a Turing RTX 2080Ti (TU102, which was used in our experiments) that operates on a 1.635Ghz clock and contains 68 SMs (or 544 TCUs), theoretical tensor core performance reaches 113TOPS. Similar architectures have been introduced by Google [9] (TPU) and Intel [3] (NNP).

NVIDIA provides two dinstinct ways of programming these units: a) Widely used libraries such as cuBLAS [4] and cuDNN [2] have been extended with new kernels that utilize the TCUs to accelerate GEMM performance. CUTLASS [5] (CUDA templates for Linear Algebra Subroutines), another NVIDIA library that built upon C++ in order to enable high-performance in BLAS-like kernels supports code generation for the TCUs as well. b) the CUDA WMMA (Warp level Matrix Multiply and Accumulate) API provides a more direct way for CUDA developers to program these units using specific intrinsics. These intrinsics operate on a new data type called `fragment` which represents the part of the array that will be used in the following TCU instructions and can vary per data type, memory layout of the corresponding array and/or size. Fragments can either be used in load, store or multiply/accumulate instructions (`wmma.load`, `wmma.store` and `wmma.mma` intrinsics respectively). Load and store intrinsics can read/store into the shared or the global memory. In this work, we instead use the NVVM IR intrinsics that correspond to the above instructions, and extend the Halide compiler passes accordingly in order to generate high-performance GEMM kernels.

2.2 The Halide DSL and Compiler

As already mentioned, Halide [13] is an image processing DSL that has proven to be capable of generating high-performance code without obscuring the functionality of the implementation. This is achieved through the separation of algorithmic description and optimization schedule. As an example, consider the code seen in Listing 1, which implements a simple matrix multiplication kernel in Halide.

```
1  Var x("x"), y("y"),xi("xi"),yi("yi");
2  int matrix_size=1024;
3  // Algorithm
4  RDom k(0, matrix_size);
5  C(x, y)  = 0.0f ;
6  C(x, y) += A(k,y) * B(x,k);
7
8  // Schedule
9  C.compute_root().gpu_tile(x,y,xi,yi,32,16);
10 C.update().gpu_tile(x,y,xi,yi,32,16);
```

Listing 1: Example Matrix Multiplication in Halide

In detail, lines 5 and 6 are responsible for the functional behavior of the application and define the relationship between output and input data. Line 5 initializes all elements of array C to zero and then line 6 which is called an update definition (in Halide terms) over the initialization describes the matrix multiplication of arrays A and B (and accumulation in output array C). Lines 9 and 10 dictate the optimization schedule of the implementation and control details such as the loop transformations that will be applied on the generated code, the memory storage and layout for intermediate buffers, the order in which input data is loaded and other architecture specific information, such as the loop dimensions that correspond to the x,

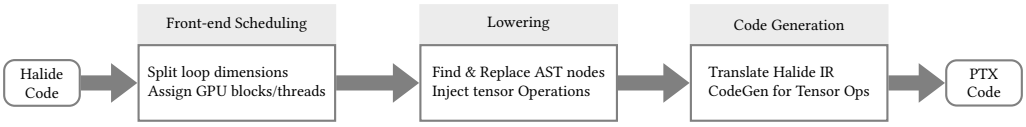

Figure 2: Basic Compilation Flow: The `tensor_core` scheduling directive dictates that the corresponding Halide function should be ported on the TCUs. Calls to custom tensor core intrinsics are injected into the AST during lowering and finally translated into LLVM/NVVM IR before generating the final PTX code.

y and z dimensions of a threadblock on a GPU. It is important to note that the optimization schedule does not affect the functional output of the code. In this specific example, the `gpu_tile` scheduling directive will cause the compiler to tile the `x` and `y` dimensions with tile sizes of 32 and 16 respectively, such that `xi` and `yi` are the inner intra-tile loops and `x`, `y` the outer inter-tile loops. At the same time, the inner (intra-tile) loops will be assigned as CUDA threads while the outer (inter-tile) loops as block dimensions. The same will be done for the update definition of `C`.

As seen in the above example, Halide allows for increased code portability and readability, since: a) the optimizations applied through the scheduling directives cannot change the functional output of the code and b) changing the target platform only requires rewriting the schedule. All scheduling transformations are applied after the initial description has been lowered into an intermediate representation (IR). Halide IR remains platform independent until later stages of the compilation flow where it is translated into LLVM IR for the final target code generation. Specifically, for CUDA based architectures, Halide IR gets mapped into LLVM's NVVM backend to generate PTX code. NVVM IR is a compiler IR (internal representation) based on the LLVM IR which was designed to represent GPU compute kernels (for example, CUDA kernels). For each Halide function/stage a different CUDA kernel is generated through LLVM. In the above example, one kernel would be launched to initialize the output array to zero and another one would be responsible for the actual matrix multiplication.

We will show that by injecting new tensor specific intrinsics as well as properly modifying Halide's PTX code generator (where the translation to LLVM IR occurs for the PTX backend) we can generate high performance code that utilizes the NVIDIA TCUs.

3 RELATED WORK

Imaging DSLs provide an efficient high-level way for developers to generate high-performance implementations without sacrificing code readability and maintainability. Due to this reason, custom backends and extensions for architectures that operate as accelerators have been a popular subject in the context of such languages [11, 13, 14, 16]. Most DSLs use LLVM as a backend to generate code for accelerators. Halide [13] and Tensor Comprehensions [16] are two example languages that translate their IR into LLVM IR in order to generate PTX code for GPU offloading. In a similar fashion, Halide can generate code for the Qualcomm Hexagon DSP with HVX extensions. Other approaches make use of C code generators in these DSLs to support FPGA code generation through HLS (High level synthesis) [12], or DSPs [17]. TVM [1] is a deep learning compiler stack, heavily influenced by Halide IR. It supports various architectures and was recently extended with Tensor Core code generation [15]. However, unlike our approach, TVM instead injects CUDA WMMA intrinsics into C/C++ output code and requires nvcc to compile the final implementation. Our method does not need the code to be linked with any CUDA libraries during design time and thus enables cross compilation. Moreover, TVM requires the user to make scheduling choices for the mapping, while we predefine an efficient schedule that as seen in the benchmarks of Section 5 can achieve high throughput in nearly all cases while remaining generic.

4 TENSOR CORES IN HALIDE

This section presents our extensions to the Halide compiler in order to enable code generation for the NVIDIA TCUs. To this end, we introduce a new `tensor_core` scheduling directive. After our extensions the optimization schedule presented in Listing 1 becomes:

```
1 // Schedule
2 C.compute_root().gpu_tile(x,y,xi,yi,32,16);
3 C.update().tensor_core(A,row,B,col);
```

Listing 2: The new tensor_core directive is enough to generate high performance code for GEMM kernels in supported architectures

As seen in the above listing, the new directive only requires the input arrays and their storage layout. The size of the fragments, as well as the data types and the necessary NVVM intrinsics to generate code are all automatically derived based on the type of the input buffers. In a similar way, the dimensions of the grid to be launched as well as the amount of shared memory to be requested for the implementation are calculated. A simplified view of the compilation steps required to generate TCU code can be seen in Figure 2. In the language front-end, we introduce the `tensor_core` directive which marks the computation of a Halide function to be ported on the TCU. At the same time we split the dimensions of the loop such that we create a 256x256 tile (similar to the CUDA WMMA implementation in [6]) and assign the corresponding dimensions to the CUDA grid dimensions. Finally we propagate the types of the input and output arrays, as well as their layout to the upcoming IR passes. Since this is a new scheduling directive, we need to provide new information during the lowering phase in order for the compiler to know what kind of new transformations should be applied on the code. We first attempt to fix the bounds of the loops that need to be modified to account for the required grid dimensions. To this end, we introduce a new lowering pass during which we transform the AST by injecting IR nodes with calls to tensor operation intrinsics. For the rest of this paper, A, B and C

NVVM Intrinsic	A Layout	B Layout	C Layout	A/B Type	C Type	Halide IR
llvm.nvvm.wmma.m16n16k16.load.a.row.stride.f16.p0i32	row	-	-	float16	-	mma_load
llvm.nvvm.wmma.m16n16k16.load.b.col.stride.u8.p0i32	-	col	-	uint8	-	mma_load
llvm.nvvm.wmma.m16n16k16.load.c.row.stride.f32.p0f32	-	-	row	-	float32	mma_load
llvm.nvvm.wmma.m16n16k16.store.d.col.stride.i32.p0i32	-	-	col	-	int32	mma_store
llvm.nvvm.wmma.m16n16k16.mma.col.col.f32.f32	col	col	-	float16	float32	mma_operation
llvm.nvvm.wmma.m16n16k16.mma.row.col.u8	row	col	-	uint8	int32	mma_operation

Table 1: NVVM WMMA Intrinsic Selection Examples

refer to the arrays/buffers of Listing 1, while a, b, c and d refer to the tensor fragments used in WMMA operations. We introduce an intrinsic for each type of tensor instruction:

(1) `mma_load(a/b/c,pointer,layout,stride,type)` : Loads data from the memory location of `pointer` into the `a/b/c` fragment of a TCU. The memory location may be in either the shared or global memory. The size, type and stride of the fragment is inferred by `type` and `stride`.
(2) `mma_store(pointer,layout,stride,type)` : Stores the `d` fragment data from the TCU registers into the memory location specified by `pointer`, which can point to either the shared or global memory. The size, type and stride of the fragment is inferred by `type` and `stride`.
(3) `mma_operation` : Multiplies the `a` and `b` fragments and accumulates the result (plus `c`) in the `d` fragment.

All of the above arguments that are not present in the `tensor_core` scheduling directive are automatically inferred by the compiler.

We use a modified version of the schedules provided by NVIDIA [6] as an example schedule to showcase the effectiveness of the tensor core architecture in Halide. We focus on the optimized version which automatically makes use of the maximum shared memory available on the platform to store chunks of A, B and C arrays. When this is not possible, a naive implementation that does not make use of the shared memory is provided at the cost of performance. Efficient streaming of A/B/C data to and from the global memory is realized through calls to memcpy. Listing 3 shows an example lowered AST of the schedule generated by the lowering pass for arrays of size 1024×1024, row, column layout for A and B respectively and mixed precision (float16 for A, B and float32 for array C). For simplicity, the pointer and indexing calculation is not shown, but all pointers are typically functions of the surrounding loop iterators. The number of blocks of the launched kernel is equal to the SM count of the GPU. A While node was also implemented in the Halide IR as seen in line 4 of Listing 3 to account for the step size in the original CUDA WMMA code.

Elements of C are first streamed into the shared memory (from the global memory). In case C is zero as in the example of Listing 1 then the call to the initial `memcpy` can be replaced by a `memset`. The same data is then loaded into fragments to be reused in the following multiply/accumulates through a `mma.load.c` instruction. A similar process is repeated for chunks of A and B arrays. Iterating across the global K dimension, fragments a and b are multiplied and accumulated into fragment d, reusing fragments of b against a in the process. Finally, d fragments are stored into the shared memory before being streamed back into the global memory. All of the previous steps are repeated until there are no more tiles to

```
1  gpu_block<CUDA> (output.s1.x.__block_id_x, 0, 68) {
2   allocate __shared[65536] in GPUShared
3   gpu_thread<CUDA> (.__thread_id_x, 0, 256) {
4    while (s0, output.s1.x.__block_id_x, ((s0*8)/64)*(8) < 64, 68){
5     unrolled (s1, 0, 16) {//Stream C into shared memory
6      (float32)mma_memcpy_C_to_shared(ptr_C, ptr_shared)
7     }
8     gpu_thread_barrier()//Sync threads
9     unrolled (s2, 0, 8) {//Load multiple fragments of C
10     (float32)mma_load(c, ptr_shared, row, stride)
11    }
12    gpu_thread_barrier()
13    unrolled (s3, 0, 16) {//loop over global K dimension
14     unrolled (s4, 0, 8) {//Stream A/B chunks into shared memory
15      (float16)mma_memcpy_AB(ptr_A, ptr_B, ptr_shared)
16     }
17     gpu_thread_barrier()
18     unrolled (s5, 0, 4) {
19      unrolled (s6, 0, 2) {//Load fragment A
20       (float16)mma_load(a, ptr_shared, row, stride)
21       unrolled (s7, 0, 4) {//Load fragment B
22        (float16)mma_load(b, ptr_shared, col, stride)
23        (float32)mma_operation()//Multiply/accumulate
24       }
25      }
26     }
27     gpu_thread_barrier()//Sync threads
28    }
29    unrolled (s8, 0, 8) {//Store accumulator into shared memory
30     (float32)mma_store(ptr_shared, row, stride)
31    }
32    gpu_thread_barrier()//Sync threads
33    unrolled (s9, 0, 16) { //stream C back to global memory
34     (float32)mma_memcpy_C_to_global(ptr_C, ptr_shared)
35    }
36    gpu_thread_barrier()//Sync threads
37   }
38  }
39 }
```

Listing 3: Example lowered AST of the computational loop after tensor operations have been injected. The schedule has been adapted from NVIDIA. All loop bounds and types are automatically derived by the compiler.

compute for each block. More specific details for the schedule can be found in the NVIDIA CUDA Samples [6].

At this point the Halide IR along with the newly injected tensor nodes/calls needs to be translated into LLVM/NVVM IR before the final code generation. Proper NVVM intrinsic selection depends on the data type and storage layout of the input/output arrays. All the necessary information was either already propagated by the previous compilation stages or was inferred by the compiler. An example of Halide IR to NVVM intrinsic mapping can be seen in Table 1. All fragments of C/D are 8 elements wide, while fragments of A/B arrays are 4 elements wide when they contain `uint8` elements and 8 elements wide when the data type is `float16`.

Since load and multiply/accumulate operations return a fragment data type (an array of 8 floats or integers depending on the data type of the output), all elements in the accumulator fragment C/D need to be extracted and stored into the local memory after each `wmma.mma.load.c` operation in order to ensure that the accumulator fragment is actually updated after each cycle. For `wmma.mma.store.d` operations, the same elements need to be loaded into registers before they can be used in the instruction. Finally, `wmma.mma` operations are handled by first loading the previous values of the accumulator into registers and then storing them back after each multiply/accumulate. This process is automatically handled during the translation of the Halide IR tensor intrinsics along with all necessary memory allocations.

5 EVALUATION

This section presents the experimental results obtained using the `tensor_core` scheduling directive along with the other extensions presented above.

All experiments were performed on a RTX 2080Ti NVIDIA GPU with 68 SMs. By default CUDA limits the maximum allowed shared memory per block to 48Kb. Bypassing this limit is possible by setting the maximum dynamic shared memory size to 64Kb through the `cuFuncSetAttribute` PTX runtime function which is added to the Halide CUDA runtime before calling the kernel. The average execution time of each benchmark was measured through Halide's `benchmark` subroutine across 100 iterations and 10 samples followed by a device sync.

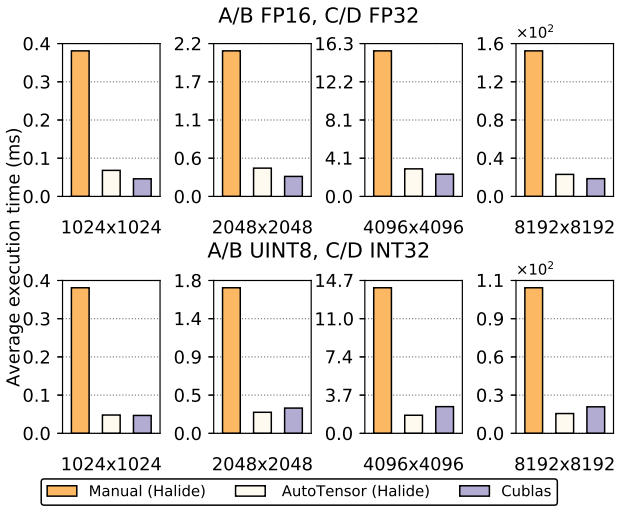

Figure 3: Matrix multiplication average execution time on an NVIDIA RTX 2080Ti GPU, for mixed precision and fixed-point implementations on various problem sizes.

The experimental results obtained on the NVIDIA RTX 2080Ti GPU are shown in Figure 3. All implementations refer to the Halide function of Listing 1. The top rows shows the results across four different problem sizes when the input A/B arrays contain elements of half precision and the output is calculated in full precision, while the bottom row shows the same results on fixed-point implementations. The "manual" bar refers to the fastest Halide optimization schedule for matrix multiplication (that does not use tensor cores), while cuBLAS refers to the results obtained using cuBLAS on CUDA version 10.0. The optimized implementation we provide assumes that the storage layout for array B is column major while the Halide code of Listing 1 requires all arrays to be row major. Due to this reason, we transpose array B before using it in the matrix multiplication. The execution time for this transposition is taken into account for the "AutoTensor" bar of Figure 3. Tensor cores speed up performance by up to 10x on larger problem sizes. The overhead of transposition accounts for less than 10% of the runtime. Moreover, our tensor core Halide implementation remains on average within 29% of the cuBLAS performance for mixed precision matrix multiplications, and it is even faster when using integer data types since cuBLAS cannot yet use tensor operations for Turing architectures: Most BLAS kernels in cuBLAS are optimized for the Volta architecture which did not have integer tensor core support. When both Halide and cuBLAS assume a row, column major layout for arrays A and B and no transposition is needed, Halide can stay within 18% of cuBLAS performance.

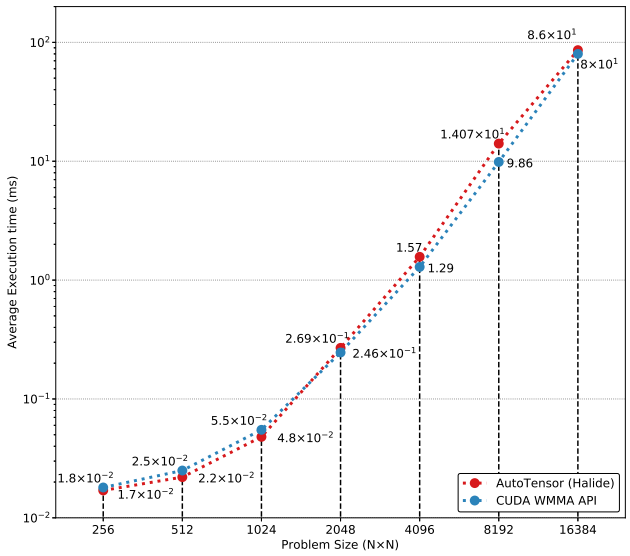

Figure 4: Average execution time of matrix multiplication on a wide range of problem sizes on NVIDIA RTX 2080Ti.

Since standard cuBLAS routines (SGEMM) cannot yet utilize the tensor core units when using int32 data types, we instead compare the performance of our implementations against the hand-optimized fast GEMM CUDA kernel found in [6] with imma (Integer Matrix Multiply Accumulate) intrinsics, specialized for the Turing architecture. Figure 4 shows an extended comparison for Int32 matrix multiplications on the same platform across a wider range of problem sizes. Both implementations assume a row, col storage layout for arrays A and B respectively. We see that the automatic Halide implementation achieves competitive performance (7% slower on average) with the manual CUDA code [6] in nearly all cases.

6 CONCLUSIONS

In this work we presented a new scheduling directive for the Halide DSL along with a set of intrinsics and lowering passes that enable automatic, efficient code generation for matrix multiplications leveraging the Tensor Core units present in the latest CUDA GPU architectures. We evaluated our extensions using a Turing based NVIDIA GPGPU and showed that through our custom intrinsics and passes, Halide can achieve performance competitive with cuBLAS and manual CUDA code on matrix multiplication kernels.

REFERENCES

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[3] Intel Corporation. 2020. *Intel Neural Network Processor*. Retrieved February 14, 2020 from https://www.intel.ai/nervana-nnp/#gs.wby5fr

[4] NVIDIA Corporation. 2008. NVIDIA cuBLAS library. https://developer.nvidia.com/cublas version 10.0.

[5] NVIDIA Corporation. 2017. *CUDA Templates for Linear Algebra Subroutines*. Retrieved February 14, 2020 from https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

[6] NVIDIA Corporation. 2017. *NVIDIA CUDA Samples*. Retrieved February 14, 2020 from https://github.com/NVIDIA/cuda-samples/blob/master/Samples/cudaTensorCoreGemm/cudaTensorCoreGemm.cu

[7] NVIDIA Corporation. 2017. *NVIDIA Tensor Cores*. Retrieved February 14, 2020 from https://www.nvidia.com/en-us/data-center/tensorcore/

[8] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA, USA. http://www.deeplearningbook.org.

[9] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[11] Richard Membarth, Oliver Reiche, Frank Hannig, Jurgen Teich, Mario Korner, and Wieland Eckert. 2016. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (Jan. 2016), 210–224. https://doi.org/10.1109/TPDS.2015.2394802

[12] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article Article 26 (Aug. 2017), 25 pages. https://doi.org/10.1145/3107953

[13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[14] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. 2015. Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU-8)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/2716282.2716290

[15] TVM. 2019. TVM GitHub Repository (Apache-2.0 license). https://github.com/apache/incubator-tvm (commit 9ff44969e3b566a8f1a7a50c327f63a3427984420.

[16] 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

[17] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. 2017. Extending Halide to Improve Software Development for Imaging DSPs. *ACM Trans. Archit. Code Optim.* 14, 3, Article Article 21 (Aug. 2017), 25 pages. https://doi.org/10.1145/3106343

A SOURCES

1. Dependencies

This section describes the process in order to reproduce the results obtained in the Evaluation section of the paper.

(1) *Hardware Dependencies*: A CUDA GPU with tensor cores is required (Turing - sm75). All experiments were conducted on a RTX 2080Ti GPU.

(2) *Software Dependencies*: To build and run the provided source code the following frameworks are required:

- Clang/LLVM 9.0 or higher (for Linux)
- Linux 5.0 (tested on Ubuntu 18.04)
- Make 4.1 or higher
- Git 2.17 or higher
- NVIDIA CUDA driver 10.0 or later

2. Installation

(1) *Acquiring LLVM:*

Linux binaries for LLVM 9.0 along with the matching version of Clang can be found through http://llvm.org/releases/download.html. Both llvm-config and clang must be somewhere in the path.

(2) *Acquiring and building Halide TCU:*

The source code for Halide with TCU support can be found through:

```
$ git clone https://github.com/TUE-EE-ES/HalideTCU.git
```

Point Halide to llvm-config:

```
$ export LLVM_CONFIG=<path to llvm>/build/bin/llvm-config
```

To build Halide:

```
$ cd Halide
$ make
```

3. Benchmarking

This subsection explains the process in order to reproduce the results obtained in Figures 3 and 4.

(1) To reproduce the results of Figure 3 set the `CUDA_SDK` variable and build the mixed precision matmul benchmark along with the necessary transposition. For the int32 matmul run the benchmark in the `mat_mul_int` folder. The output should correspond to the average execution time for each of the three implementations:

```
$ export CUDA_SDK=<path_to_cuda>
$ cd apps
$ cd mat_mul
$ make test MATRIX_SIZE=1024 TRANSPOSE=1
```

(2) To reproduce the Halide runtimes of Figure 4 build the int32 matmul benchmark. For the WMMA runtimes use the imma example code provided by NVIDIA [6]:

```
$ cd apps
$ cd mat_mul_int
$ make test MATRIX_SIZE=1024 TRANSPOSE=0
```

Changing the value of `MATRIX_SIZE` after `make clean` generates an implementation for other problem sizes. All runtimes are expected to have a variation of +- 10% but a similar ratio across each implementation compared to the one seen in the above figures. The new passes and extensions for the TCU code generation can be found in `Halide/src`: `Inject_Tensor_Ops.cpp`, `top_equiv.cpp`, `CodeGen_PTX_Dev.cpp`, `Func.cpp` and smaller additions to other parts of the compiler and runtime.