MLIRAGE : MLIR for Ampere-Architecture GPGPU Efficiency

Xuan Nguyen
Bachelor of Computer Science, Worcester Polytechnic Institute

Abstract

As the grand scheme for distributed deep learning approaching exascale computing, GPU vendors introduced proprietary micro-architecture for their GPUs to meet such demand. In order to efficiently employ the full capability of such hardware novelties, GPU vendors also ship their products with powerful, hand-tuned API frameworks and libraries. The challenge of such approach is that they require programmers with specialized experience to utilize them to the fullest – and since such libraries were written in lower level abstraction to that of modern programming language for machine learning applications, they restrained productivity with unavoidable boilerplating routines and codes. We mitigated this problem by automating the lowering process and codegen for GPUs – from a high-level framework, down to low-level hardware-specific instruction using the MLIR compiler framework. In this report, we demonstrated the process with automatic generated kernel codes for 2D vector multiplication. Our experiment shows that with large matrix multiplication, our pipline achieve near peak throughput that is comparable to vendor fine-tuned codes

1 Introduction

With the advances in machine learning and deep learning research, the demand for High-Performance Computing (particularly GPGPUs) has also increased. GPUs nowadays come with thousands of parallel processing cores capable of rapidly solving large problems with substantial parallelism. The latest GPU architectures such as Nvidia Turing [3], Ampere [4] consist of Tensor cores [5], [6], which can perform several mixed-precision matrix-multiply and accumulate computations in a single operation. Usually, matrix operations are among the commonly performed operations in the fields of HPC (High-Performance Computing) and ML (Machine Learning), which are highly parallel in nature. The matrix multiplication is the most common out of all the matrix operations performed, which in generalized form is known as GEMM (General Matrix Multiply) and is represented as D = aAB + BC. The ordinary matrix multiplication AB can be performed by setting a to one and C to an all-zeros matrix. GEMM in the field of HPC is used for dense linear algebra [7], [8], earthquake simulation [9], and climate prediction [10]. In ML, GEMM is used for training convolutional neural networks, long short-term memory (LSTM) cells, and natural language processing. CUDA, a parallel computing API created by Nvidia, enables us to execute C, C++, etc., codes on Nvidia GPUs. Having GPUs and CUDA API is not enough; we also need a highly optimized kernel to run on these GPUs to maximize the usage of available resources. In general, GPU vendors provide manually tuned implementations such as cuBLAS [11], cuDNN [12], etc., which contain various deep learning primitives for several different matrix layouts. The problem with these manually tuned implementations is that they need to be tuned for every new architecture that comes into the market, and tuning them requires people with specialization in those domains. The process of generating efficient fine-tuned implementations of deep learning primitives can be automated using polyhedral frameworks. The matrix operations, such as matrix multi- plication, bias addition, etc., can be expressed as affine loop-nests, which can be easily analyzed and transformed using polyhedral frameworks. The advantages of automating the process of generation of an efficient kernel is to achieve portability across hardware architectures, neural network's composition and easy of code debugging and interpretation. The compiler infrastructure parts that we build for optimizations, such as tiling and unroll and jam, are common to both CPUs and GPUs. The compiler framework we are using to build this infrastructure is MLIR (Multi-Level Intermediate Representation) [13, 14, 15].

2 Motivation

Several libraries already exist that include handwritten, highly optimized kernels for GPU. CUDA libraries such as cuDNN [12], and cuBLAS [11] provides highly tuned implementations of deep learning functions/algorithms and basic linear algebra subroutines. These libraries are optimized only for a limited set of matrix layouts, and thus, they don't provide much flexibility. CUB [16] an NVIDIA's library provides software components that are reusable for CUDA programming model,'s each layer such as Warp-wide, Block-wide, and Device-wide primitives for constructing high-performance, maintainable CUDA kernel code. The components provide by CUB give a state-of-the-art performance. CUTLASS [17] is a CUDA C++ template library that contains components to instantiate performant kernels on GPUs. CUTLASS also contains support for mixed-precision computation for Tensor cores. It is not easy for end-users to extend the CUTLASS since its codebase is large and maintaining such a huge codebase is difficult. The performance achieved by CUTLASS [18] for WMMA GEMM is 95/100 of the performance of cuBLAS. cuTensor [19] is a CUDA library provided by NVIDIA for its GPUs, supports several Tensor operations, for example - pointwise operations with pointwise operator fusion support. cuBLAST a library that is a lighter version of CUBLAS, is made available by NVIDIA for basic linear-algebra subroutines dedicated to GEMM, which provides flexibility in supporting more data types for input and compute matrices and more matrix layouts. Halide [20] is a DSL (embedded in the C++) compiler framework for image processing; it is also extended for supporting Tensor cores. The problem with Halide is that it does not support complex data types and the flexibility provided by it is limited; also, it supports only a single combination of memory layouts of the input matrices. Due to the Tensor comprehensions, [21] developed by Vasilache et al., and by using the polyhedral compilation techniques, the Halide compiler generates CUDA kernels for a given mathematical specification of a deep learning graph. Some of the recent works in automatic kernel generation are being done by Somashekaracharya G. Bhaskaracharya et al. [22], and Thomas Faingnaert et al. [23]. Bhaskaracharya et al. [22] in their work used a polyhedral approach like ours to generate efficient CUDA kernels for matrix multiplication using inline assembly instructions. On the other hand, we have used MLIR for generating efficient CUDA kernels. They also proposed an extended approach for generating fused kernels such as matrix multiplication plus ReLU activation or bias addition. In contrast, we have generated kernels only for matmul. Faingnaert et al. [23] framed the problem of manual code generation as a two-language problem. Faingnaert et al. [23] stated that ”efficient kernel generation require either low-level programming, which implies low programmer productivity, or using libraries that only offer a limited set of components.” The author used the LLVM framework [24], and Julia programming language [25] for generating the efficient kernels. The Julia is a dynamic and flexible program- ming language suitable for numeric and scientific computations. The performance of Julia is comparable to traditional statically-typed languages like C and C++. The work of Faingnaert et al. [23] is different from our work in two ways: we have used the affine dialect based on the polyhedral techniques and the MLIR compiler framework [13, 14, 15]; instead, their work is based on the LLVM compiler framework and Julia programming language. The advantage of using MLIR over LLVM is that MLIR provides multiple levels of abstractions like loop ab- stractions, math abstractions, tensor, and vector abstractions. Also, one can define their own operations and abstractions in MLIR suitable for the problem they are trying to solve, which is the biggest difference from LLVM.

3 Background

3.1 The MLIR Infrastructure

Multi-Level Intermediate Representation(MLIR) [13, 14, 15] aims to build reusable, extensible compiler infrastructure and reduce the cost of building domain-specific compilers. MLIR is a hybrid IR that can be used for multiple different requirements, such as: data flow graphs (such as in TensorFlow), kernels in a form suitable for optimization for ML ops, high-performance-computing-style loop optimizations across kernels (loop fusion, loop interchange, multi-level tiling, unroll-and-jam), target-specific operations such as accelerator-specific operations, kernels at different levels of abstractions. The MLIR structure is made up of the following components:

- Operations: this is the basic unit of semantics in MLIR and is commonly referred to as Op. In MLIR, everything is modeled as ops, whether it is instruction, module, or function. They take as input zero or more values called operands and produces zero or more values called results respectively. Figure 1 illustrates the modular concept of MLIR. Figure 2 shows the emitted IR codes for an MLIR operation under multiple abstractions including Tensor (native MLIR type) and MHLO (Tensorflow’s MLIR dialect).

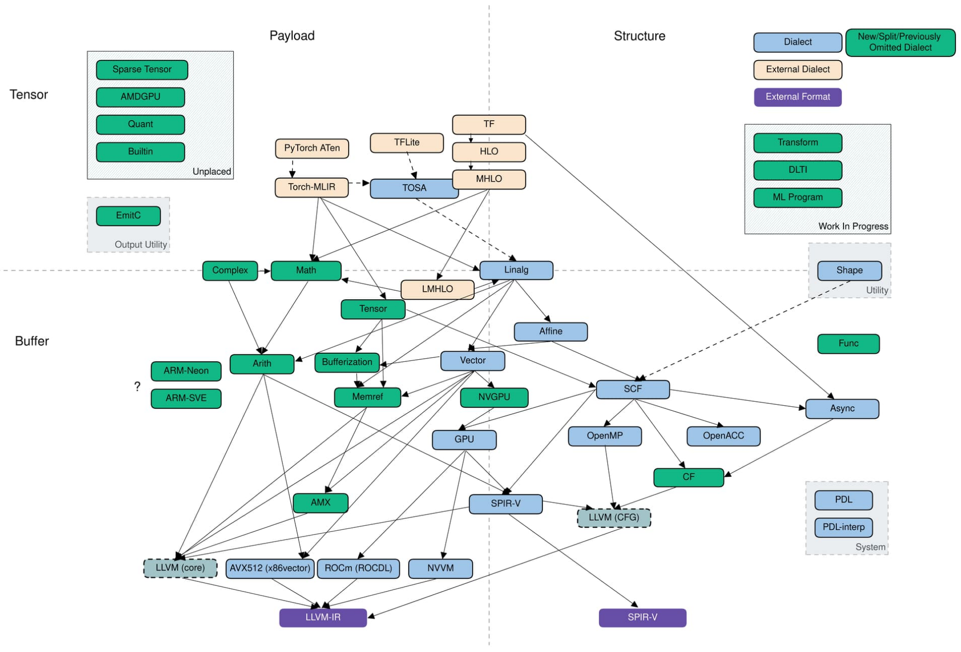


Figure 1: mlir.llvm.org

```
func@main(%arg0: tensor<1x2xi32>, %arg1: tensor<2x1xi32>) {
  %matrixA = alloc() : memref<128x64xf16>
  %matrixB = alloc() : memref<64x128xf16>
  %matrixC = alloc() : memref<128x128xf16>
  // Initialize the matrices.
  linalg.matmul ins(%matrixA, %matrixB: memref<128x64xf16>, memref<64x128xf16>)
                outs(%matrixC: memref<128x128xf16>)
  %result = "mhlo.dot"(%arg0, %arg1) : (tensor<1x2xi32>, tensor<2x1xi32>)
                                      -> tensor<i32>
}
```

Figure 2:

- Attributes: attributes are structured compile-time static information, e.g., integer con- stant values, string data, etc. Each op instance has an open key-value dictionary from string names to attribute values.

- Regions and blocks: a region in MLIR is a list of blocks, and a block contains a list of operations that may further contain regions. The blocks inside the region form a Control Flow Graph (CFG). Each block may have successor blocks to which the control flow may be transferred, and it ends with a terminator op.

- Dialects: a dialect is a logical grouping of operations, attributes, and types. Operations from multiple dialects can coexist at any level of the IR at any point in time. Dialects allow extensibility and provide flexibility that helps in performing specific optimizations and transformations. There are many dialects in the MLIR, but the dialects we have worked with are Affine, GPU, LLVM, SCF, and Standard (std).

- Functions and modules: A module is an operation with a single region containing a single block and terminated by a dummy op that does not transfer the control flow. On the other hand, a function is an op with a single region, with arguments corresponding to function arguments.

3.1.1 Breakdown of our MLIR pipeline

- Affine Dialect: This dialect uses techniques from polyhedral compilation to make depen- dence analysis and loop transformations efficient and reliable. We have performed most of the optimizations and transformations at the level of affine dialect. The description of some ops and their functionalities from the affine dialect that we have used in this document can be referenced from the MLIR documentation available at https://mlir.llvm.org/docs/Dialects/Affine/ : Affine Expressions, Affine Maps, AffineParallelOp (affine.parallel).

- GPU Dialect: This dialect provides middle-level abstractions for launching GPU kernels following a programming model similar to that of CUDA or OpenCL. It provides abstractions for kernel invocations (and may eventually provide those for device management) that are not present at the lower level (e.g., as LLVM IR intrinsics for GPUs). Its goal is to abstract away device- and driver-specific manipulations to launch a GPU kernel and provide a simple path towards GPU execution from MLIR. It may be targeted, for example, by DSLs using MLIR. The dialect uses gpu as its canonical prefix. https://mlir.llvm.org/docs/Dialects/GPU/

- NVVM Dialect: Since we are focusing on Tensor core code generation, we use and extend another Nvidia specific dialect known as NVVM. This dialect provides operations that are directly mapped to the NVPTX back-end in LLVM. https://mlir.llvm.org/docs/Dialects/NVVMDialect/

- LLVM Dialect: The final stage of code generation involves lowering to LLVM IR, from where the LLVM back-end takes control and generates the target code. To model LLVM IR, we use this dialect called the LLVM dialect, the lowest level of abstraction present in MLIR. https://mlir.llvm.org/docs/Dialects/LLVM/

3.2 The Ampere-Architecture Programming Model

GPUs [26] are massively parallel processors, which means many threads execute the same function commonly referred to as kernel in parallel. Since our approach, analysis, and experimentation are only confined to NVIDIA GPUs, we will limit our discussion to NVIDIA GPUs, and the CUDA programming model [27]. The smallest unit of execution is a thread organized in a thread hierarchy. The hardware groups threads into sets of 32 threads called warps. The 32 threads in the same warp execute in a SIMT (Single Instruction Multiple Thread) manners. In other words, these threads within a warp execute the same instruction simultaneously, possibly on different data. The threads are logically grouped into blocks known as thread-block. The set of all thread blocks on the GPU device is called the grid. Just like the threads, the GPU memory is also organized hierarchically. The smallest and fastest type of memory in GPUs is the register file or registers. Each thread in GPU typically has access to 255 registers. The threads within a thread block have their own shared memory set, which they use for communication. The largest capacity memory on the GPUs is global memory, but it also has the highest latency. All threads on the device (GPU) can access the global memory, irrespective of which thread block they belong to. Figure 3 shows the architecture model of Ampere GPUs and figure 4 shows the micro-architectures of general GPUs that also apply to Ampere ones.

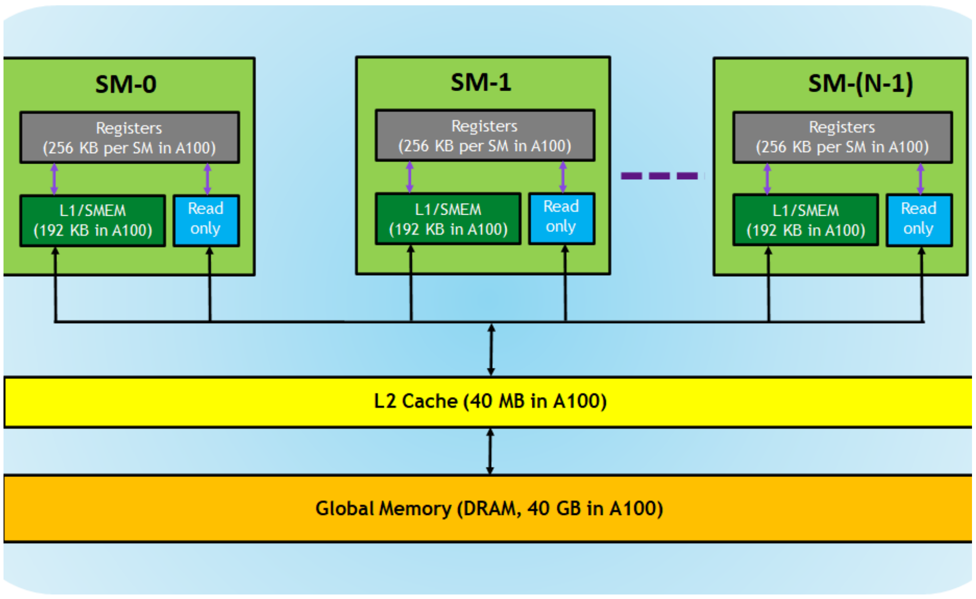


Figure 3: nvidia.com

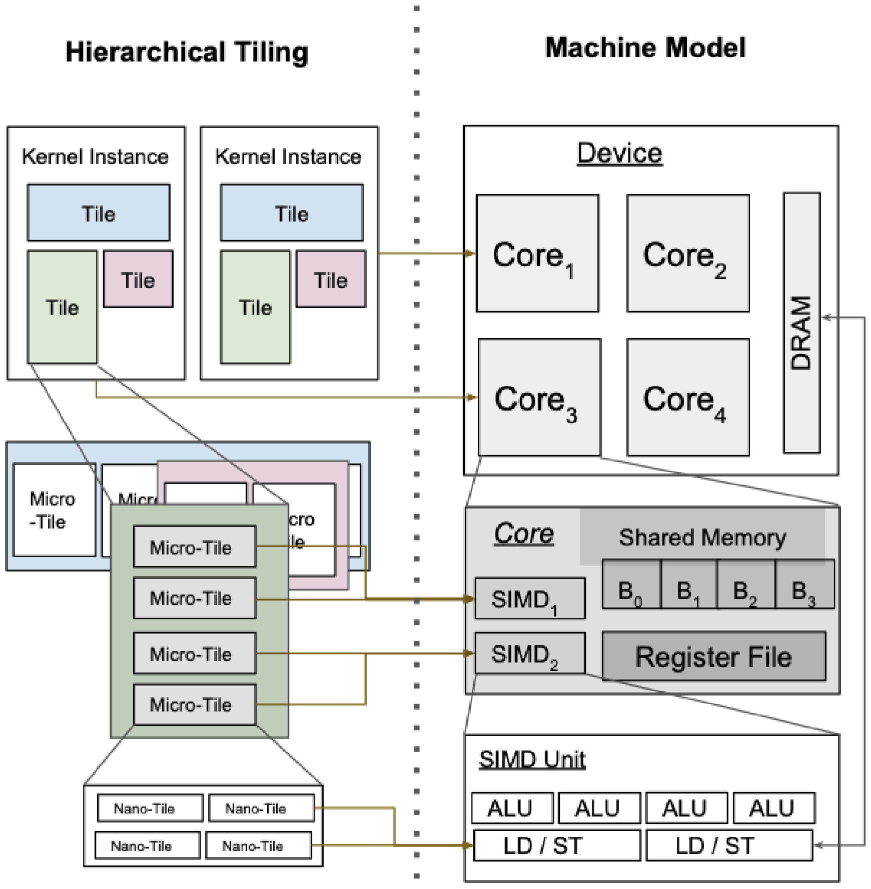


Figure 4: triton-lang.org

3.2.1 Tensor Cores

Tensor cores [5], [6] were first introduced in NVIDIA's Volta architecture [28] in 2017. Tensor cores are programmable matrix-multiply-and-accumulate units that provide a 4x4x4 matrix processing array which performs the operation D = A * B + C, where A, B, C and D are 4×4 matrices. Each Tensor core performs 64 floating-point fused-multiply-add mixed-precision operations per clock. Multiple Tensor cores are used concurrently by a full warp of execution. A warp comprising of 32 threads provide a larger 16x16x16 matrix operation to be processed by the Tensor cores. The Tensor cores operation are exposed by CUDA as warp-level matrix operations in the CUDA C++ WMMA (Warp Matrix Multiply Accumulate) API [29]. Only a limited set of data types are supported by tensor cores such as TF32, FP16, FP64, INT8 for input. We can use Tensor cores through libraries such as NVIDIA's cuDNN [12] library that contains Tensor cores kernels. NVIDIA's cuTENSOR [19] based on CUTLASS [17] contains Tensor-Core-accelerated kernels. LLVM also provides intrinsics which are mapped one-to-one with the functions provided by WMMA API.

4 Pipeline Implementation

Figure 5 displays the IR lowering pipeline, from which the details for each components will be described later in the section.

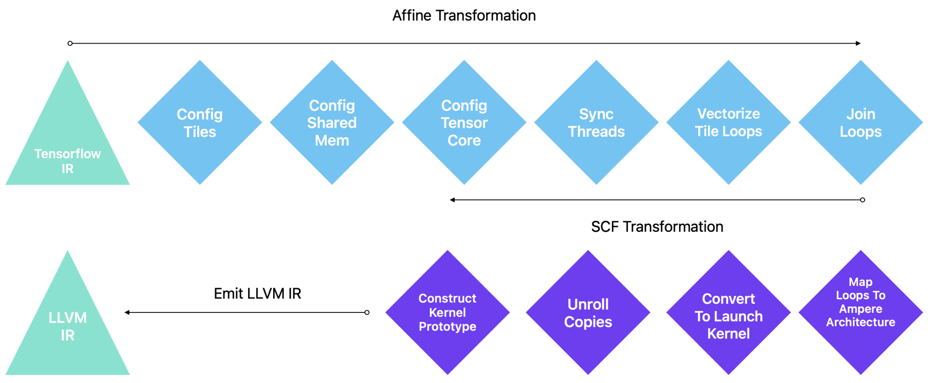


Figure 5: Generated with Apple's Freeform

- Tensorflow IR represents the basic matrix multiplication operation in some higher abstraction level such as MLIR-HLO [30] or TF. The IR is now lowered to affine dialect abstraction level to perform loop optimizations.

- Tiling, also known as blocking, important from parallelism and locality viewpoint, is performed for better data reuse and performance. The key objective of the tiling is to maximize the ratio of computation operations to memory operations. We have performed two-level tiling: thread-block level and warp level. The tile sizes are chosen based on the tile-size selection model discussed in Section 5.

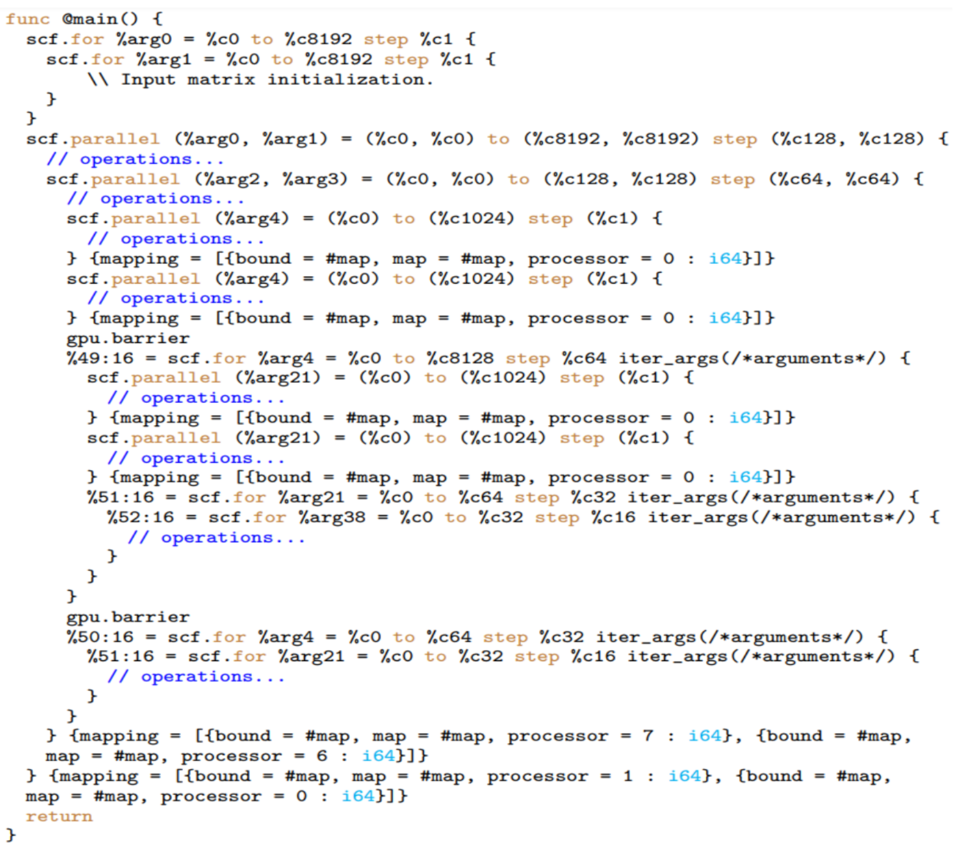
- Since we are generating kernels targeting GPUs, we allocate buffers in GPU's shared memory. After the allocation, the data is copied from GPU's global memory into the shared memory buffers. We have allocated the shared memory buffers only for the input matrices (A and B), and this is a design choice that we have made. The thread-block level tile size determines the shared memory buffer size for both the matrices. The tile sizes are chosen such that the shared memory can be used maximally. The shared memory buffers are used for lower latency and higher bandwidth.

- To take advantage of the Ampere GPU architecture, we generate Tensor cores specific operations such as wmma.load, wmma.store, wmma.mma sync, which replaces the scalar load, store, and compute operations [31].

- The synchronization barriers are inserted in the code to ensure data is available before the computation begins. The synchronization barriers being used in our generated code are actually a thread-block level barrier to synchronize all threads in a thread block after copying the input matrix tiles into the shared memory. The number of synchronization barriers inserted is kept minimal because the barrier causes stall, ultimately degrading performance.

- The vectorization is performed to make use of the SIMD instruction set available in GPU architectures. For doing loop vectorization in MLIR, we have to convert memrefs of f16 (input matrices data type) into memrefs of a vector of f16 and transform the loop bounds and loop bodies accordingly. We have created the vectors, each comprising 8 elements. The vectorization is a crucial part of the solution because it provides a 2x speedup in the performance. We know that memory is divided into banks and successive 32-bit words assigned to successive banks. Also, each bank can service one address per cycle. If multiple simultaneous accesses are made to a bank, then it results in a bank conflict. As a result, the conflicting accesses are serialized. Hence the memory access time increases, and the performance decreases. The solution to this is shared memory padding. The padding is done to ensure simultaneous bank accesses are reduced to the extent possible, thereby reducing the memory bank conflicts. For padding, we have tuned a parameter known as the padding factor, which specifies how much value the memory should be padded. Also, we have padded the k dimension of the matrices. The optimal padding factor turns out to be 8, which gives us the best performance. To map the parallel loop nests to GPU compute hierarchy, we first need to identify and mark the parallel loops in the IR. An affine.for is parallel if all its iterations can be executed in parallel, or we can say if all its iterations are independent of each other. We mark parallel loops as affine.parallel so that during the time of execution, there is a clear separation between the set of instructions that can execute in parallel and the instructions which need to be executed serially. The pass algorithm works as follows: walks over the entire input IR and collects all those affine.for which are parallel, using a utility that checks for all the dependencies inside the iteration space of affine.for. Then, one by one converts all affine.for obtained in the first step to affine.parallel and keep affine.parallel's bounds same as bounds of affine.for. After marking the loops as parallel, we collapse perfectly nested affine.parallel ops into a single n-dimensional affine.parallel op where n is the sum of dimensions of all affine.parallel ops, which are coalesced together. This process happens as follows: Collects all the affine.parallel ops, which are perfectly nested. Collects all the affine expressions corresponding to the lower and upper bound map of affine.parallel ops obtained in the previous step. Collects all the lower and upper bound operands of affine.parallel ops. Creates new lower and upper bound maps using affine expressions obtained in the previous step. Creates new lower and upper bound maps using affine expressions obtained in the last step. Creates new coalesced affine.parallel op using maps and operands obtained in the last two steps, respectively. Finally, we colapse the parallel loop nests helps in mapping the loops to GPU compute hierarchy since all the perfectly nested parallel loops are now grouped into a single op.

- To execute the kernel on the GPU, the parallel loops need to be mapped to GPU entities such as thread blocks, warps, and threads. We greedily map loops starting from outermost to innermost to each GPU entity. Mapping the parallel loop means to specify which loop is distributed over a thread block grid, distributed over a thread block, or distributed over warps. For mapping the parallel loops, we move from the outermost loop to the innermost loop. Each parallel loop in between is mapped to some GPU compute hierarchy. The parallel loops at nesting level one are mapped to the thread block grid. The parallel loops at nesting level two are mapped to the thread block, and the ones at nesting level three are mapped to the warps. All parallel loops which are at nesting level four or more are marked as sequential. The mapping is done by attaching an attribute to each parallel loop. Figure 6 demonstrates in MLIR code how we will be able to achieve the mapping to Ampere-exclusive architecture.



```
func @main() {
  scf.for %arg0 = %c0 to %c8192 step %c1 {
    scf.for %arg1 = %c0 to %c8192 step %c1 {
        \\ Input matrix initialization.
    }
  }
  scf.parallel (%arg0, %arg1) = (%c0, %c0) to (%c8192, %c8192) step (%c128, %c128) {
    // operations...
    scf.parallel (%arg2, %arg3) = (%c0, %c0) to (%c128, %c128) step (%c64, %c64) {
      // operations...
      scf.parallel (%arg4) = (%c0) to (%c1024) step (%c1) {
        // operations...
      } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
      scf.parallel (%arg4) = (%c0) to (%c1024) step (%c1) {
        // operations...
      } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
      gpu.barrier
      %49:16 = scf.for %arg4 = %c0 to %c8128 step %c64 iter_args(/*arguments*/) {
        scf.parallel (%arg21) = (%c0) to (%c1024) step (%c1) {
          // operations...
        } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
        scf.parallel (%arg21) = (%c0) to (%c1024) step (%c1) {
          // operations...
        } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
        %51:16 = scf.for %arg21 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
          %52:16 = scf.for %arg38 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
            // operations...
          }
        }
      }
      gpu.barrier
      %50:16 = scf.for %arg4 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
        %51:16 = scf.for %arg21 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
          // operations...
        }
      }
    } {mapping = [{bound = #map, map = #map, processor = 7 : i64}, {bound = #map,
    map = #map, processor = 6 : i64}]}
  } {mapping = [{bound = #map, map = #map, processor = 1 : i64}, {bound = #map,
  map = #map, processor = 0 : i64}]}
  return
}
```

Figure 6:

- In MLIR's GPU dialect, we have an operation known as LaunchOp, which launches a kernel on the specified grid of thread blocks. The body of the kernel is the single region that this operation contains. The parallel loops mapped to the thread block grid GPU compute hierarchy in the previous section are now converted into a LaunchOp. There are six operands of the LaunchOp, which are grid and block sizes. The grid and block sizes are determined using the thread-block level tile size and warp level tile size. The body region of launch op contains six arguments that are block identifiers and thread identifiers, along the x,y,z dimensions. The loop induction variable, lower and upper bounds of parallel loops mapped to the thread block, and warp in the previous section are also modified based on some computations performed using tile sizes of both the levels and linear thread id.

- The loop unrolling provides better opportunities for instruction scheduling and register tiling. Loop unrolling also helps reduce control overhead, and reduced instruction count due to fewer number compare and branch instructions. The loops are unrolled based on a loop unroll factor, which specifies that by what extent the loop has to be unrolled. In our case, we have unrolled only the copy loops and have unrolled them completely. After unrolling the copy loops, we delay the data copy by moving the copy instructions after the computation instructions. The copies are delayed to hide the latency of loads from global memory [31]. The GPU kernel outlining pass is run now, converting all the GPU dialect's LaunchOp into LaunchFuncOp which launches a function as a GPU kernel on the specified grid of thread blocks. Since we are using the LLVM backend for code generation, we now convert MLIR to LLVM IR. Now, the LLVM backend will generate the target code. Figure 7 demonstrates in MLIR code how we will be able to generate Ampere-specific kernel with mapped loops.

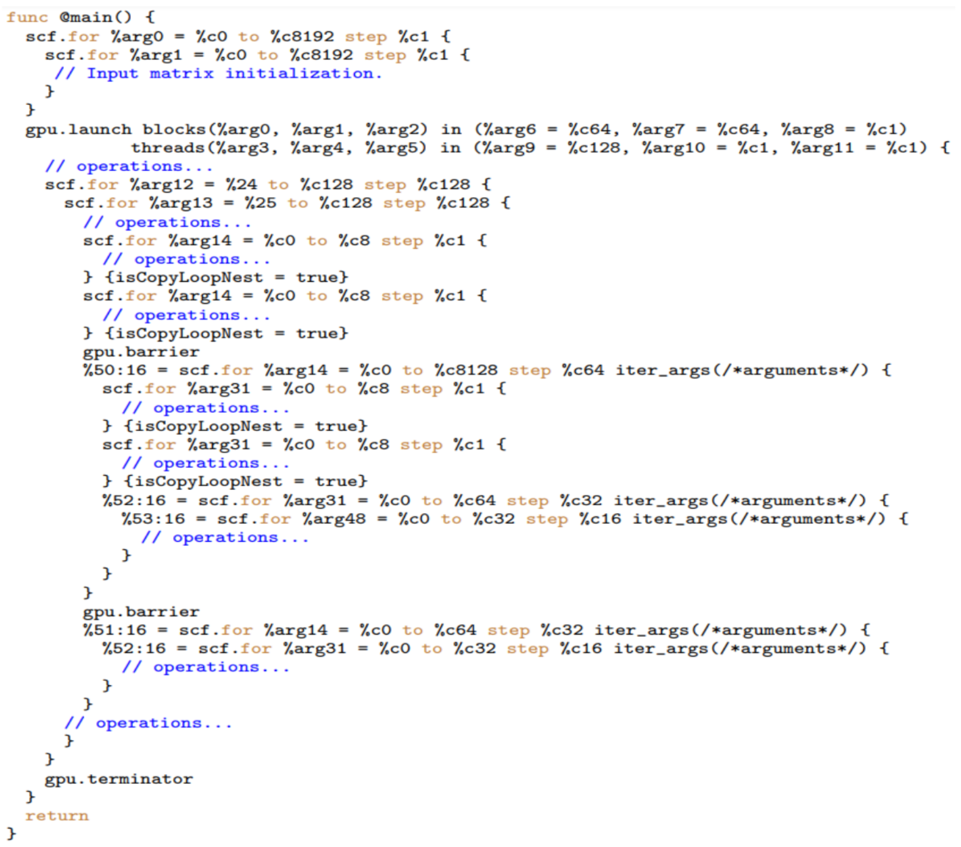

```
func @main() {
  scf.for %arg0 = %c0 to %c8192 step %c1 {
    scf.for %arg1 = %c0 to %c8192 step %c1 {
     // Input matrix initialization.
    }
  }
  gpu.launch blocks(%arg0, %arg1, %arg2) in (%arg6 = %c64, %arg7 = %c64, %arg8 = %c1)
             threads(%arg3, %arg4, %arg5) in (%arg9 = %c128, %arg10 = %c1, %arg11 = %c1) {
    // operations...
    scf.for %arg12 = %24 to %c128 step %c128 {
      scf.for %arg13 = %25 to %c128 step %c128 {
        // operations...
        scf.for %arg14 = %c0 to %c8 step %c1 {
          // operations...
        } {isCopyLoopNest = true}
        scf.for %arg14 = %c0 to %c8 step %c1 {
          // operations...
        } {isCopyLoopNest = true}
        gpu.barrier
        %50:16 = scf.for %arg14 = %c0 to %c8128 step %c64 iter_args(/*arguments*/) {
          scf.for %arg31 = %c0 to %c8 step %c1 {
            // operations...
          } {isCopyLoopNest = true}
          scf.for %arg31 = %c0 to %c8 step %c1 {
            // operations...
          } {isCopyLoopNest = true}
          %52:16 = scf.for %arg31 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
            %53:16 = scf.for %arg48 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
              // operations...
            }
          }
        }
        gpu.barrier
        %51:16 = scf.for %arg14 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
          %52:16 = scf.for %arg31 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
            // operations...
          }
        }
      // operations...
      }
    }
    gpu.terminator
  }
  return
}
```

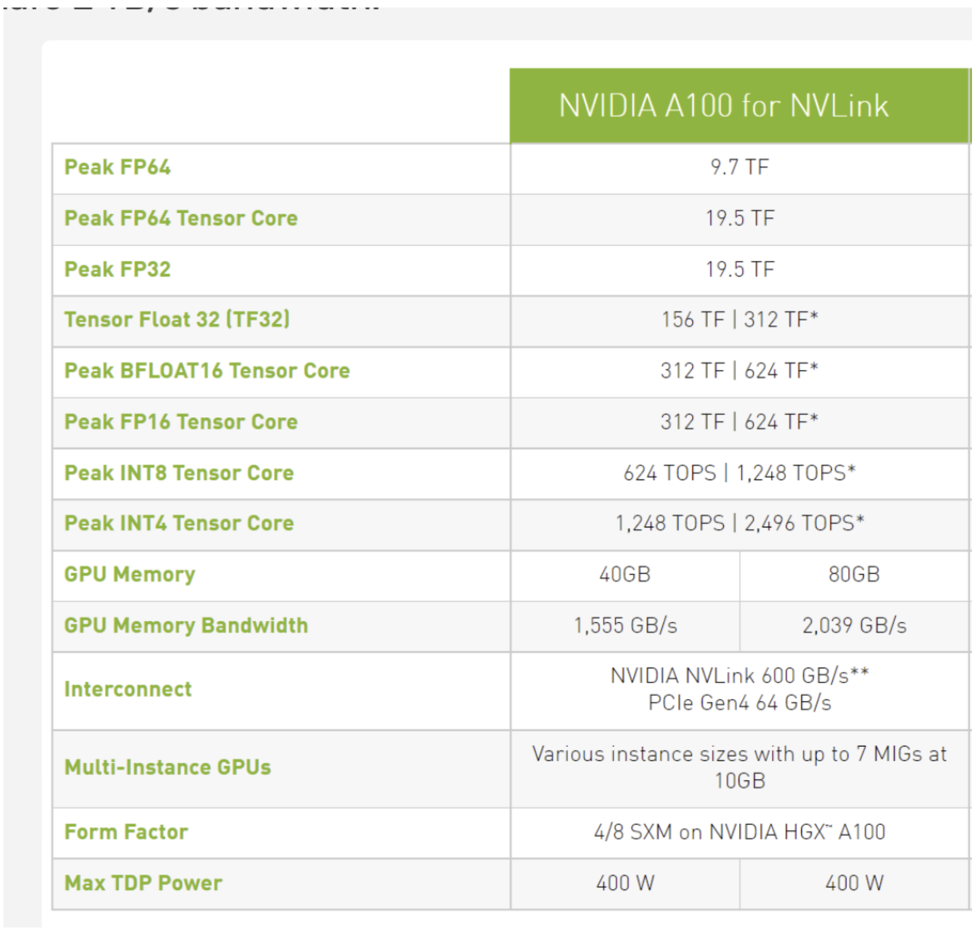
Figure 7:

5 Benchmark

Loop tiling, also known as blocking, is done for better data reuse and, all state-of-the-art implementations of GEMM perform two-level tiling. We have also performed two-level tiling, namely thread-block tiling and, warp-level tiling also known as register tiling. In this section, we first explain the experimental setup, and then we discuss the impact of different levels of tiling on the performance. We have used the following symbols in the subsequent subsections: m, n, and k constitute the problem size in dimensions. Tm, Tn, and Tk is the thread-block level tile size for dimensions m,n, and k, respectively. Wm, Wn, and Wk is the warp level tile size for dimensions m,n, and k, respectively.

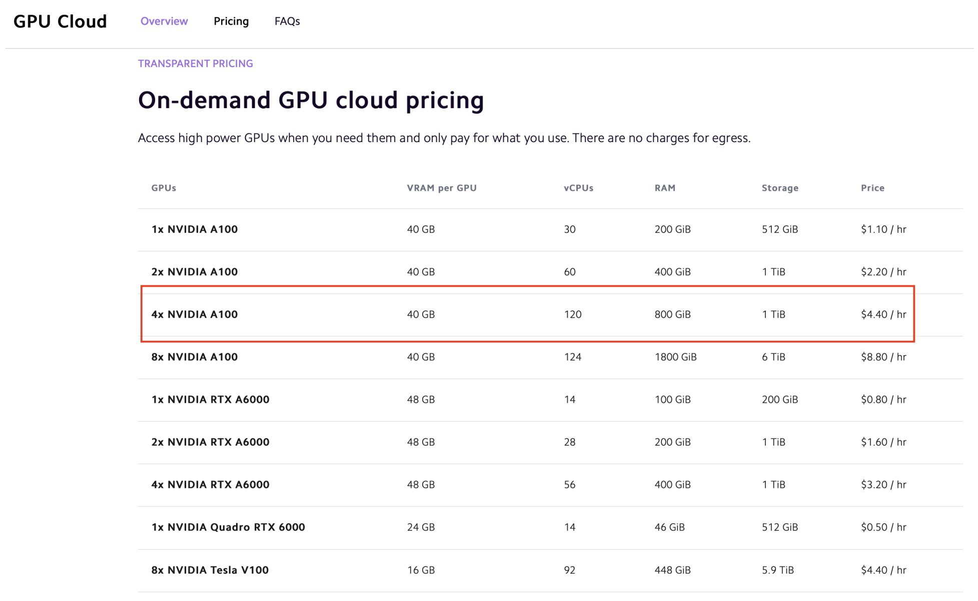
5.1 Hardware

The experiments are done on a Lambda GPU Cloud Computing Platform with 4x NVIDIA A100 (40GB) running on Linux instance which can be access through secure ssh from the local network. We have used NVIDIA SMI to profile the kernel executions and gather the details of performance metrics for analysis. The problem size chosen for multiplication is 8192x8192x8192. The input data is of type FP16, and the accumulator/output is of type FP32. Figure 8 describes the specification of NVIDIA A100-SMX4-40GB units used in our report. Figure 9 briefs about the services offered on Lambda GPU Cloud Computing. We chose a cluster of 4x A100 GPUs as it is economically feasible for our scope of study and budget. Figure 10 shows the result for our pipeling across various tile sizes.



	NVIDIA A100 for NVLink	
Peak FP64	9.7 TF	
Peak FP64 Tensor Core	19.5 TF	
Peak FP32	19.5 TF	
Tensor Float 32 (TF32)	156 TF | 312 TF*	
Peak BFLOAT16 Tensor Core	312 TF | 624 TF*	
Peak FP16 Tensor Core	312 TF | 624 TF*	
Peak INT8 Tensor Core	624 TOPS | 1,248 TOPS*	
Peak INT4 Tensor Core	1,248 TOPS | 2,496 TOPS*	
GPU Memory	40GB	80GB
GPU Memory Bandwidth	1,555 GB/s	2,039 GB/s
Interconnect	NVIDIA NVLink 600 GB/s** PCIe Gen4 64 GB/s	
Multi-Instance GPUs	Various instance sizes with up to 7 MIGs at 10GB	
Form Factor	4/8 SXM on NVIDIA HGX™ A100	
Max TDP Power	400 W	400 W

Figure 8: nvidia.com


GPU Cloud   Overview   Pricing   FAQs

TRANSPARENT PRICING

On-demand GPU cloud pricing

Access high power GPUs when you need them and only pay for what you use. There are no charges for egress.

GPUs	VRAM per GPU	vCPUs	RAM	Storage	Price
1x NVIDIA A100	40 GB	30	200 GiB	512 GiB	$1.10 / hr
2x NVIDIA A100	40 GB	60	400 GiB	1 TiB	$2.20 / hr
4x NVIDIA A100	40 GB	120	800 GiB	1 TiB	$4.40 / hr
8x NVIDIA A100	40 GB	124	1800 GiB	6 TiB	$8.80 / hr
1x NVIDIA RTX A6000	48 GB	14	100 GiB	200 GiB	$0.80 / hr
2x NVIDIA RTX A6000	48 GB	28	200 GiB	1 TiB	$1.60 / hr
4x NVIDIA RTX A6000	48 GB	56	400 GiB	1 TiB	$3.20 / hr
1x NVIDIA Quadro RTX 6000	24 GB	14	46 GiB	512 GiB	$0.50 / hr
8x NVIDIA Tesla V100	16 GB	92	448 GiB	5.9 TiB	$4.40 / hr

Figure 9:

5.2 Result

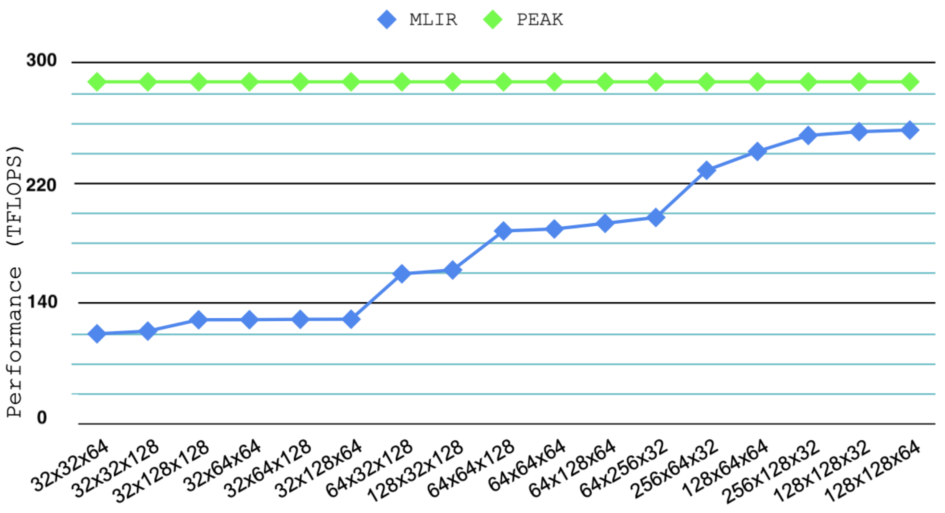


Figure 10:

PEAK performance represents what fine-tuned codes using vendor libraries (cuBLAS) can achieve, our result show that our pipelines seems to lag behind the cuBLAS by quite a large margin. However, as the problems size scales larger, we can see that our implementation offers comparable performance to that of cuBLAS. To help explain this performance residual, we theorized that the as the synchronization transformation are inserted in the code to ensure that all the threads finish copying data and computation for a given iteration before the next iteration begins, since some threads take more time to complete than others; as a result, remaining threads have to wait at the barrier for completion, and the amount of time spent in waiting is known as stall barrier cycles. This created a overhead for small problem size, reducing throughput. Next, we suspect that a large of time is spent on data movement between the memory hierchachy of Ampere architecture for smaller problem set, between tile-blocks and memory regions, leading to low Tensor (FP) functional unit utilization. The data movement happens in this way: Global memory $\rightarrow$ L2 cache $\rightarrow$ L1 cache $\rightarrow$ registers. Since the tiling helps in data reuse, the better the data reuse, the better the performance. As a result, some threads spend more time copying data. In comparison, others spend less time doing computation, due to which a large amount of time is spent waiting on the barrier for all threads to complete. For second-level tiling, also known as register tiling, is to distribute a thread block tile over the appropriate number of warps. While moving the data from shared memory into registers, the warp level tile size plays a key role; hence the padding along with appropriate warp level tile size is must for reducing the bank conflicts. However, this reduces number of registers available also limits the number of warps/threads that can be issues/executed.

5.3 Limitation and Future Work

Due to the lacking verbosity of NVIDIA SMI, many of our theories on the performance residual can not be thoroughly examined and confirmed. As Lambda Cloud can only provide us with an instance of 4x A100 GPUs due to high demand for the server, we can not evaluate our pipeline on a different Ampere system that are available such as A6000, A4500, A100-80GB, etc. Plus, we provide for the automatic kernel generation for matrix multiplication is restricted only to the GPUs comprising Tensor cores. In the future, this work can be extended to make this solution work for the GPUs not having tensor cores and also for the GPUs made available by hardware vendors other than NVIDIA. Our solution tackles the problem of automatic code generation only for matrix multiplication, and it can be further extended for problems such as automatic generation of fused kernels for matrix multiplication plus pointwise operations such as bias addition, ReLU activation, and convolution.

6 Reference

1 Alex Zinenko, Nicolas Vasilache, Stephan Herhut, Mahesh Ravishankar, Geof- frey Martin-Noble. Codegen Dialect Overview.. https://llvm.discourse.group/t/ codegen-dialect-overview/2723.

2 NVIDIA Corporation, CUDA C Programming Guide, 2011.

3 NVIDIA Corporation, Turing architecture, https://images.nvidia.com/aem-dam/ en-zz/Solutions/design-visualization/technologies/turing-architecture/ NVIDIA-Turing-Architecture-Whitepaper.pdf.

4 NVIDIA Corporation, Ampere architecture, https://www.nvidia.com/en-in/ data-center/ampere-architecture/.

5 Jeremy Appleyard and Scott Yokim. “Programming Tensor Cores in CUDA 9,” 2017, https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/.

6 V. Mehta, “Getting started with Tensor Cores in HPC,” 2019, NVIDIA GPU Technology Conference.

7 Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Towards Half-Precision Com- putation for Complex Matrices: A Case Study for Mixed Precision Solvers on GPUs. In 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large- Scale Systems (ScalA), pages 17–24, 2019.

8 Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Re- finement Solvers. In SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 603–613, 2018.

9 Tsuyoshi Ichimura, Kohei Fujita, Takuma Yamaguchi, Akira Naruse, Jack C. Wells, Thomas C. Schulthess, Tjerk P. Straatsma, Christopher J. Zimmer, Maxime Martinasso, Kengo Nakajima, Muneo Hori, and Lalith Maddegedara. A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing. In SC18: Interna- tional Conference for High Performance Computing, Networking, Storage and Analysis, pages 627–637, 2018.

10 Vishal Mehta. "Getting started with Tensor Cores in HPC," 2019, NVIDIA GPU Tech- nology Conference.

11 NVIDIA Corporation, "cuBLAS" 2019, https://docs.nvidia.com/cuda/cublas/ index.html.

12 NVIDIA Corporation, "cuDNN", https://docs.nvidia.com/deeplearning/cudnn/ index.html.

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

14 Chris Lattner and Jacques Pienaar. MLIR Primer: A Compiler Infrastructure for the End of Moore's Law, Compilers for Machine Learning Workshop, CGO 2019.

15 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pien- aar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In 2021 IEEE/ACM In- ternational Symposium on Code Generation and Optimization (CGO), pages 2–14, 2021.

16 NVIDIA Corporation, CUB, https://docs.nvidia.com/cuda/cub/index.html.

17 NVIDIA Corporation, "CUDA templates for linear algebra subroutines," 2019, vol. 21, no.5, pp. 313–348, 1992. https://github.com/NVIDIA/cutlass.

18 NVIDIA Corporation, CUTLASS performance, https://github.com/NVIDIA/cutlass performance.

19 NVIDIA Corporation, "cuTENSOR: A high-performance CUDA library for Tensor prim- itives," 2019, https://docs.nvidia.com/cuda/cutensor/index.html.

20 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fredo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Paral- lelism, Locality, and Recomputation in Image Processing Pipelines. SIGPLAN Not., 48(6):519–530, June 2013.

21 Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary De- Vito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions, Facebook AI Research Technical Report. February 13, 2018.

22 Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. Automatic Kernel Generation for Volta Tensor Cores. CoRR, abs/2006.12645, 2020.

23 Thomas Faingnaert, Tim Besard, and B. D. Sutter. Flexible Performant GEMM Kernels on GPUs. ArXiv, abs/2009.12263, 2020.

24 LLVM Foundation, The LLVM Compiler Infrastructure. https://llvm.org/.

25 The Julia language, 2021. https://julialang.org/.

26 John Nickolls and William J. Dally. The GPU Computing Era. IEEE Micro, 30(2):56-69, 2010.

27 NVIDIA Corporation, https://docs.nvidia.com/ cuda/pdf/CUDACProgrammingGuide.pdf.

28 NVIDIA Corporation, Volta architecture, https://images.nvidia.com/content/ volta-architecture/pdf/volta-architecture-whitepaper.pdf.

29 NVIDIA Corporation, “CUDA Toolkit Documentation,” 2019, https: //docs.nvidia.com/cuda/par thread-execution/index.html warp-level-matrix-fragment-mma-884.

30 Google Brain Team, Tensorflow/MLIR-HLO, https://github.com/tensorflow/ mlir-hlo.

31 Navdeep Kumar Katel, M.Tech. Research Thesis. Automatic Code Generation for GPU Tensor Cores using MLIR.