NVDSL: Simplifying Tensor Cores with Python-Driven MLIR Metaprogramming

Guray Ozen [1]

Abstract

Exploiting the formidable computational capabilities of modern GPU tensor cores remains a challenging endeavor for developers. Existing programming models like CUDA and OpenCL are ill-suited for the non-SIMT nature of tensor cores, leaving a significant gap in the landscape of GPU programming languages. Vendors have primarily relied on library-based solutions or enhancements to mainstream machine learning frameworks, sacrificing the fine-grained control once afforded by CUDA in the SIMT era.

In this paper, we introduce NVDSL, a Python-embedded domain-specific language that is based on MLIR compiler. NVDSL abstracts away the intricate details of tensor core programming. It allows programmers to efficiently program Hopper's *Warpgroup* (128 threads or 4 warps), enabling users to express sophisticated algorithms, such as *multistage* and *warp specialization*, with remarkable simplicity. We demonstrate its efficacy through two optimized GEMM kernels that achieve cuBLAS-like performance with remarkable code clarity. It is publicly available in upstream MLIR. The tutorial of this work is presented in EuroLLVM24 [1].

1. Introduction

GPUs have gained popularity due to their dazzling performance. Vendors have provided programming languages such as OpenCL and CUDA, which give users full control while handling tedious tasks like generating extensive assembly code. Extensive resources like books, talks, and blog posts have made these languages accessible.

The introduction of tensor cores with the Nvidia Volta architecture revolutionized performance. However, programming tensor cores did not align with the SIMT model, and Nvidia has not exposed tensor core functionalities within the CUDA programming model. Their usage remained confined to Parallel Thread Execution (PTX)(NVIDIA, 2024) assembly, with Nvidia providing the CUTLASS(Thakkar et al., 2023) library or close source cuBLAS library, offering many tensor core recipes by performance engineers. Despite this, programming tensor cores remains challenging for many developers.

We propose *NVDSL*, a Python-like language designed to leverage tensor cores. *NVDSL* allows programmers to efficiently use Hopper's *Warpgroup* (128 threads or 4 warps) to perform matrix multiply-accumulate (MMA) with a simple `D += A @ B` line, while our compiler handles the complexity of generating hundreds of lines of code. This groundbreaking approach enables the creation of high-performance kernels, like those in CUTLASS, in just a few hundred lines of *NVDSL* code.

Our contributions include:

1. `NVVM` dialect as an intrinsic layer in the MLIR compiler; it is very close to the PTX
2. `NVGPU` dialect as an atomic layer in the MLIR compiler; it provides readable abstractions and generates multiple `NVVM` OPs.
3. *NVDSL*, a Python-like language, as an MLIR generator specifically for the `NVGPU` dialect.
4. Performant *multistage* and *warp-specialized* kernels using *NVDSL*, achieving cuBLAS-like performance.

Our article explains `NVVM` and `NVGPU` dialects implementation in MLIR compiler in Section 2, NVDSL Python language in Section 3. We show a *multistage* and *warp specialized* kernels in Section 4 and evaluate them. Section 5 concludes our article.

2. Implementation of Hopper Support in MLIR Compiler

Initially, neither LLVM nor MLIR supported the Hopper GPU, and support for some Ampere architecture features was lacking. In this article, we introduced Hopper GPU

[1]Google Research. Correspondence to: Guray Ozen <gry@google.com>.

Work presented at the ES-FoMo Workshop at ICML 2024, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

[1]https://www.youtube.com/watch?v=V3Q9IjsgXvA

```
def NVVM_MBarrierArriveExpectTxOp : NVVM_PTXBuilder_Op<"mbarrier.arrive.expect_tx">,
  Arguments<(ins LLVM_AnyPointer:$addr, I32:$txcount, PtxPredicate:$predicate)> {
  let assemblyFormat = "$addr `,` $txcount (`,` `predicate` `=` $predicate^)? attr-dict `:` type(operands)";
  let extraClassDefinition = [{
    std::string $cppClass::getPtx() { return std::string("mbarrier.arrive.expect_tx.b64 _, [%0], %1;"); }
}]; }
```

Figure 1. TableGen: Definition of MBarrierArriveExpectTxOp using `BasicPtxBuilder` Interface

```
nvvm.mbarrier.arrive.expect_tx.shared %barrier, %txcount, predicate = %pred : !llvm.ptr<3>, i32, i1
```

Figure 2. Example IR: Using MBarrierArriveExpectTxOp NVVM Op

```
llvm.inline_asm has_side_effects asm_dialect =
      att "@$2 mbarrier.arrive.expect_tx.b64 _, [$0], $1;", "l,r,b" %arg0, %arg1, %arg2
      : (!llvm.ptr, i32, i1) -> ()
```

Figure 3. LLVM IR: Generated IR with inline asm

support in the `NVVM` and `NVGPU` dialects, explained below. We selected the MLIR compiler due to its dialect mechanism. Our work is upstreamed to open-source MLIR.

2.1. NVVM Dialect

The `NVVM` dialect is an intrinsic layer, closely resembling PTX assembly, however it has slightly higher level of abstraction. The Hopper architecture introduces new tensor cores and a data load unit called TMA. Leveraging TMA requires asynchronous transactional barriers (mbarriers). We implemented the related PTX instructions in the `NVVM` dialect.

The `NVVM` dialect is designed to generate LLVM intrinsic; however, LLVM lacks intrinsic for Hopper architecture. We developed the `BasicPtxBuilder` interface in MLIR, which automatically generates PTX by reading the tablegen definition of the `NVVM` Op. This interface generates `inline_asm`, and we have implemented over 40 Ops using it.

Figure 1 shows the OP definition in TableGen using the `BasicPtxBuilder` interface. The tablegn includes the *getPtx* function that returns the PTX string, and the interface automatically figures `inline_asm` OP generation. Figure 1 shows how this Op looks in NVVM IR. Figure 2 shows the generated PTX in the LLVM dialect by the `BasicPtxBuilder` interface. Note that the `NVVM` Ops interface is designed to be close to PTX, facilitating a smooth transition to LLVM intrinsic once they become available.

2.2. NVGPU Dialect

The `NVGPU` dialect serves as a crucial intermediate layer in the MLIR compiler, bridging high-level MLIR dialects (such as `Memref`, `Vector`, and others) to the `NVVM` dialect. We significantly expanded `NVGPU` capabilities and responsibilities in this work to target Hopper architecture. This expansion included the introduction of 15 new Operations (Ops) specifically designed to handle Tensor Memory Accelerator (TMA), memory barriers (mbarriers), and tensor cores. These new Ops are capable of generating multiple lines of `NVVM` dialect code, thereby providing a more abstracted and manageable interface for GPU programming. It's important to note that these conversions from `NVGPU` to `NVVM` are purely mechanical and do not involve complex heuristics or decision-making processes. We can categorize and explain the new NVGPU Ops in three main categories:

2.2.1. Mbarrier Ops

Modern GPU architectures have introduced sophisticated synchronization mechanisms. One such mechanism is the mbarrier object, which allows a thread to track the completion of one or more asynchronous operations by monitoring the current phase of the mbarrier. To facilitate the use of this advanced feature, we have implemented multiple Ops in the `NVGPU` dialect that simplify mbarrier synchronization. Our implementation offers two particularly noteworthy features:

1. We enable the generation of multiple mbarrier objects, allowing for complex synchronization scenarios.
2. We permit accessing multiple mbarrier objects using a single SSA value. This approach is especially beneficial when dealing with multiple mbarriers, particularly within loop structures, as it simplifies code and improves readability.

To illustrate these features, Figure 4 provides a concrete example. In this figure, Line 6 demonstrates the creation of three distinct mbarrier objects. Subsequently, Lines 7, 8, 9 and 10 show how to access the first of these mbarrier objects, showcasing the ease of use our implementation provides.

2.2.2. TMA Ops

Utilizing the Tensor Memory Accelerator (TMA) involves a two-step process:

1. Creating a descriptor on the host
2. Loading or storing data on the device using this descriptor

```
1  !descType = !nvgpu.tensormap.descriptor<tensor = memref<128x64xf16, 3>, swizzle = swizzle_128b,
2                                          l2promo = l2promo_128b, oob = zero, interleave = none>
3  func @main() {
4   %tmaDesc = nvgpu.tma.create.descriptor %memref box[128, 64] !descType
5   gpu.launch
6     %mbar = nvgpu.mbarrier.create -> <num_barriers = 3>
7     nvgpu.mbarrier.init %mbar[%c0], %c1, predicate = %tidx0
8     nvgpu.tma.async.load %tmaDesc[i, j], %mbar[%c0] to %tileA predicate=%tid0 : !descType
9     nvgpu.mbarrier.arrive.expect_tx %mbar[%c0], 16384 predicate=%tid0
10    nvgpu.mbarrier.try_wait.parity %mbar[%c0], %phase, %ticks
11 }
```

Figure 4. Example of using mbarrier and TMA in NVGPU dialect

```
%C = nvgpu.warpgroup.mma.init.accumulator -> !<fragmented = vector<128x128xf32>>
%A = nvgpu.warpgroup.generate.descriptor %tileA : ... -> !<tensor=memref<128x64xf16, 3>>
%B = nvgpu.warpgroup.generate.descriptor %tileB : ... -> !<tensor=memref<64x128xf32, 3>>
%D = nvgpu.warpgroup.mma %A, %B, %C : <tensor = memref<128x64xf16, 3>>, <tensor = memref<64x128xf32, 3>>
                 ->  <fragmented = vector<128x128xf32>>
nvgpu.warpgroup.mma.store %D to %memrefD
```

Figure 5. Example of using Tensor Core in NVGPU dialect

The challenging aspect of this process lies in generating the appropriate load or store instructions based on the descriptor created on the host. To address this challenge, we have designed TMA Ops as types that encapsulate each parameter of the TMA descriptor. This design choice ensures that when we generate load or store instructions in the device code, we have full knowledge of every feature of the descriptor.

Figure 4 provides an example. Line 1 defines the TMA descriptor type, encapsulating all necessary information. Line 4 uses the `nvgpu.tma.create.descriptor` Op to generate the actual TMA descriptor. Finally, Line 8 demonstrates how to load data using this descriptor, showcasing the seamless integration of host-side descriptor creation and device-side data manipulation.

2.2.3. Tensor Core Ops

The introduction of the Hopper architecture brought with it warpgroup-level tensor core instructions, which are critical for achieving peak performance. Recognizing the complexity and importance of these instructions, we have developed abstractions that encapsulate their usage.

Our approach embeds the intricate details and potentially tricky aspects of using tensor core instructions directly within the tensor core Ops. This abstraction allows developers to leverage the power of tensor cores without needing to manage low-level details, thereby reducing the potential for errors and improving code maintainability.

Figure 5 presents an IR example that performs a 128x128x64 MMA using our OPs. A key point to note is that each Op in this example is executed by the entire warpgroup, highlighting the warpgroup-level nature of these operations and demonstrating how our abstraction aligns with the underlying hardware capabilities.

3. `NVDSL`: Python DSL for NVIDIA Hopper GPU Programming

We introduce NVDSL, a Python-based Domain-Specific Language (DSL) designed for generating MLIR operations and targeting the NVIDIA Hopper GPU architecture. NVDSL simplifies GPU kernel programming by abstracting complex hardware features and providing an intuitive, high-level syntax.

Table 1 illustrates the layers of NVDSL and compares them with CUTLASS. While CUTLASS implements entirely in C++, we implement the bottom layers in the MLIR compiler and the outer layers in NVDSL Python. This approach enables metaprogramming, which we elaborate on in the following section. Python layers are responsible for generating MLIR operations using MLIR's Python bindings. It targets MLIR's `NVGPU` dialect for Hopper GPU support and also generates `gpu`, `arith`, and `math` dialects.

CUTLASS	**NVDSL**
Device	**@NVDSL**.mlir_func def gemm(x, y, z): # Setups and Calls Kernel
Kernel	**@NVDSL**.mlir_gpu_launch(...) def gemm_kernel() # Kernel Body

Collective	**Multistage**:
	def prologue () *# has `NVGPU` OPs*
	def mainloop () *# has `NVGPU` OPs*
	def epilogue () *# has `NVGPU` OPs*
	
	Warp Specialized:
	def producer_loop() *# has `NVGPU` OPs*
	def consumer_loop() *# has `NVGPU` OPs*

Atom	`NVGPU` dialect
Thread	
Intrinsic	`NVVM` Dialect

Table 1. Comparison of layers of CUTLASS and NVDSL

3.1. Example: Single Tile MMA

We delve into MMA with $M = 128$, $N = 128$, $K = 64$. Listing 6 illustrates how to use NVDSL to perform this operation seamlessly.

NVDSL provides the `@NVDSL.mlir_func` decorator, allowing users to invoke `gemm_128_128_64` effortlessly from Python, as demonstrated in Line 45-49. This decorator performs just-in-time (JIT) compilation of the generated code and executes the binary, translating *NumPy* arrays into MLIR's memref types. By leveraging both NumPy and NVDSL, one can enjoy the best of both worlds: Python's simplicity and the efficiency of compiled code.

The program start by allocating and copying the input matrices a and b, and the output matrix d, to the GPU. Subsequently, it creates TMA descriptors for efficient data loading and initializes swizzled memory layouts. Within the device kernel, named `gemm_kernel`, NVDSL offers another useful decorator, `@NVDSL.mlir_gpu_launch`, which configures the kernel. Let us break down the process:

1. Create and initialize an asynchronous transactional barrier (`mbarrier`) to synchronize our threads (Lines 17-18).
2. Calculate the shared memory offsets for input matrices (Lines 20-22).
3. TMA Load request from global memory to shared memory by thread-0 (Lines 24-28).
4. Warpgroup waits the completion of the TMA load (Line 30).
5. Perform MMA operation using Tensor Cores in a single line (Line 36).
6. Stored fragmented result registers to the global memory (Line 38).

```
1 @NVDSL.mlir_func
2 def gemm_128_128_64(a, b, d):
3   # GPU alloc/cpy
4   aDev, bDev, cDev = allocCopy(a, b, d)
5    # Create TMA Descriptors
6   sw = NVGPU.TensorMapSwizzleKind.SWIZZLE_128B
7   aTma = TMA([M, K], a.type, swizzle=sw)
8   bTma = TMA([K, 64], b.type, swizzle=sw)
9   aTma.create_descriptor(aDev)
10  bTma.create_descriptor(bDev)
11  szA = get_type_size(aTma.tma_memref)
12  szB = get_type_size(bTma.tma_memref)
13
14  @NVDSL.mlir_gpu_launch(grid=(1),block=(128),smem=sz)
15  def gemm_128_128_64_kernel():
16   t0 = gpu.thread_id(gpu.Dimension.x) == 0
17   # 1. Create & Init mbarrier
18   mbars[0].init(1, predicate=t0)
19   mbars = Mbarriers(number_of_barriers=1)
20   # 2. Find shared memory of operands
21   aSmem = getSmem((M,K),T.f16())
22   bSmem = getSmem((K,N),T.f16(), szA)
23   bSmem2 = getSmem((K,N),T.f16(), szA + szB)
24   # 3. TMA Load for two input matrices
25   aTma.load(aSmem,mbars[0], coords=[0,0], predicate=t0)
26   bTma.load(bSmem,mbars[0], coords=[0,0], predicate=t0)
27   bTma.load(bSmem2,mbars[0],coords=[64,0],predicate=t0)
28   ta_count = szA + (szB * 2)
29   mbars[0].arrive(ta_count, predicate=t0)
30   # 4. All threads wait TMA load completion
31   mbars[0].try_wait()
32   # 5. Initialize input A, B, and accumulator D
33   A= WGMMAMatrix(Descriptor,[M,K],desc=aTma,smem=aSmem)
34   B= WGMMAMatrix(Descriptor,[K,N],desc=bTma,smem=bSmem)
35   D= WGMMAMatrix(Accumulator,shape=[M,N],ty=T.f32())
36   # MMA (F32 += F16 * F16)
37    D += A @ B
38   # 6. Stores fragmented registers to global memory
39   D.store_accumulator(dDev)
40  gemm_128_128_64_kernel()
41  # GPU copy device2host
42  copyD2H(d, dDev)
43
44 ### Use NumPy, call NVDSL, and verify  ###
45 # Create & Init numpy arrays
46 a = np.random.randn(M, K).astype(np.float16)
47 b = np.random.randn(K, N).astype(np.float16)
48 d = np.zeros((M, N), np.float32)
49 # Call NVDSL function
50 gemm_128_128_64(a, b, d)
51 # Verify the resulf of NVDSL
52 ref_d = a.astype(np.float16) @ b.astype(np.float16)
53 np.testing.assert_allclose(d, ref_d, ...)
```

Figure 6. GEMM 128x128x64 using TMA and Tensor Core

4. NVDSL Fast Kernels

The key point of implementing fast gemm kernels is to feed tensor core efficiently. As it is faster than data load, we need overlap tensor core with the data load. In this section we illustrate two method for that.

4.1. Multistage GEMM Kernel

Listing 8 demonstrates a multistage GEMM kernel that efficiently overlaps tensor core computations with data loading, enhancing performance. It uses more shared memory and loads multiple tile of input matrices to shared memory asynchronously using TMA. The number of stage is *ns* variable that's parametric. The GEMM shape and tile sizes are also parametric in the kernel so one code can compute multiple GEMM shapes. In this kernel we program single *Warpgroup*. We split kernel into three section: prologue, main loop and epilogue. Full code is available in [2].

Figure 7 shows the execution pipeline of a Multistage kernel. Here, we select a pipeline value of 3, resulting in 3 slots in shared memory. The kernel manages these slots concurrently.

Prologue Line(15-18) initializes the asynchronous TMA loading of tiles. This function iterates over the stages, invoking `tma_load` for each stage.

Main loop (Line 1-13): function contains the core computation. It begins by initializing the input matrices A, B, and the accumulator D (Lines 5-7). The function then enters a loop over the K dimension, divided by the tile size (Lines 9). At each iteration, the function waits for the current stage to complete using calculates the shared memory offsets for operands A and B and performs MMA operation using Tensor Cores. It also initiates the asynchronous TMA load for the next stage, ensuring computation overlap.

[2]https://github.com/llvm/llvm-project/blob/main/mlir/test/Examples/NVGPU/Ch4.py

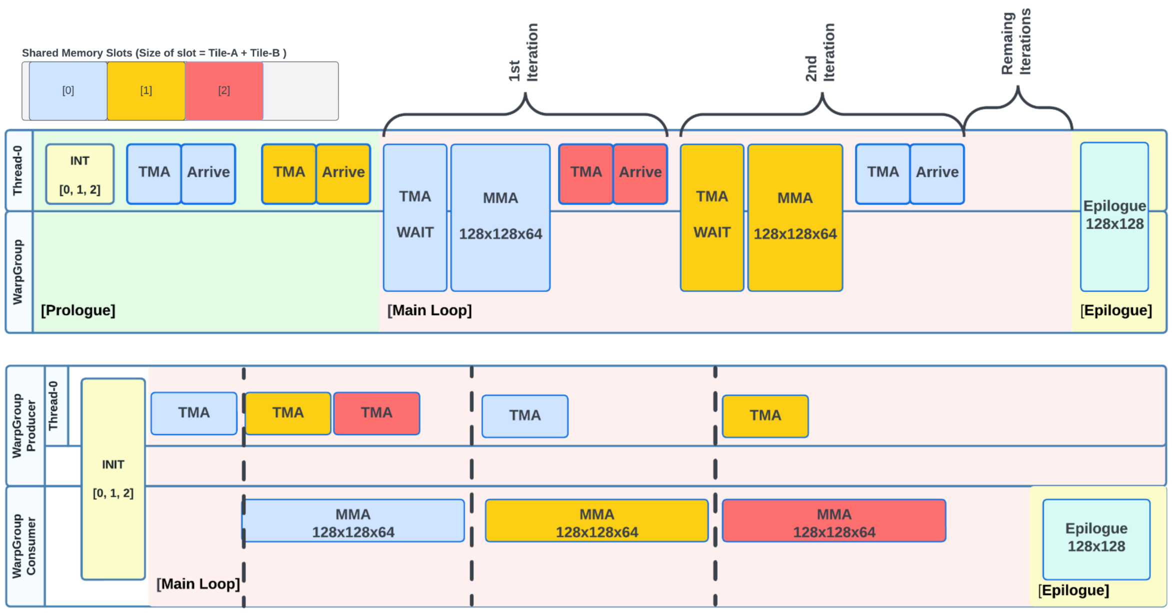

Figure 7. Multistage kernel pipeline (top), and warp specialized kernel pipeline (bottom)

```
def mainloop(mbars, aTma, bTma, numSt, TM, TN, TK):
 begin_b = numSt * get_type_size(aTma.tma_memref)
 size_a = TILE_M * TILE_K * get_type_size(T.f16())
 # Initialize input A, B, and accumulator D
  A= WGMMAMatrix(Descriptor, [TM, TK], desc=aTma)
  B= WGMMAMatrix(Descriptor, [TK, TN], desc=bTma)
  D= WGMMAMatrix(Accumulator, [TM, TN], f32())

 # Start K-Loop
 for iv,[acc,phs] in scf.for_(0,numSt,1,[D.op,False]):
  # Loop body performs MMA and TMA load

 D.update_accumulator(acc)
 return D

def prologue(mbars, aTma, bTma, numSt):
 for iv in scf.for_(0, numSt, 1):
  tma_load(mbars, aTma, bTma, iv, iv, numSt)
  scf.yield_([])


@NVDSL.mlir_gpu_launch(grid=grid,block=block,smem=sz)
def gemm_multistage_kernel():
 # 1. Initialize & Create mbarriers
 mbars = initialize(aTma, bTma, numSt)
 # 2. Async TM load tiles
 prologue(mbars, aTma, bTma, numSt)
 # 3. Main loop
 D = mainloop(mbars, aTma, bTma, numSt)
 # 4. Epilogue
 D.store_accumulator(dDev)
```

Figure 8. Multistage GEMM Host and Kernel code. The `TM, TN, TK` are the tile sizes, `numSt` is number of pipeline stages.

4.2. Warp Specialized GEMM Kernel

In this kernel, we orchestrate two *warpgroups*: a producer and a consumer. The producer warpgroup handles data loading via TMA, while the consumer warpgroup performs MMA using the tensor core and executes the epilogue. The NVDSL `Warpgroup` class provides an elegant way to manage warp-specialized kernels. Full code is available in [3].

```
1  @NVDSL.mlir_gpu_launch(grid=grid, block=block, smem=sz)
2  def gemm_warp_specialized_kernel():
3   # Init Warpgroups
4   producer = Warpgroup(pthread=128, register_size=40)
5   consumer = Warpgroup(pthread=0, register_size=232)
6
7   # Initialize mbarriers and prefetch TMA descriptors
8   mbars = initialize(a_tma, b_tma, ns)
9
10  # Producer Warpgroup performs TMA
11  with producer:
12   producer_loop(mbars, aTma, bTma, prod, ns)
13
14  # Consumer Warpgroup performs MMA, epilogue
15  with consumer:
16   D = consumer_loop(mbars, aTma, bTma, cons, ns)
17   epilogue(D, d_dev)
```

Figure 9. Warp Specialized GEMM Kernel

Figure 7 illustrates the execution pipeline of a Warp Specialized kernel on the bottom. Similar to the Multistage kernel, we select a pipeline value of 3. The main difference here is that the first warpgroup executes only TMA instructions, while the second warpgroup executes tensor core instructions.

Listing 9 shows the kernel code. We initialize two *warpgroups*: `prod` (producer) and `cons` (consumer) (Lines 4-5). The producer *warpgroup* is allocated a register size of 40, as TMA does not use registers, while the consumer *warpgroup* is allocated a register size of 232, reflecting the register-intensive tensor core operations. The `initialize` func-

[3]https://github.com/llvm/llvm-project/blob/main/mlir/test/Examples/NVGPU/Ch5.py

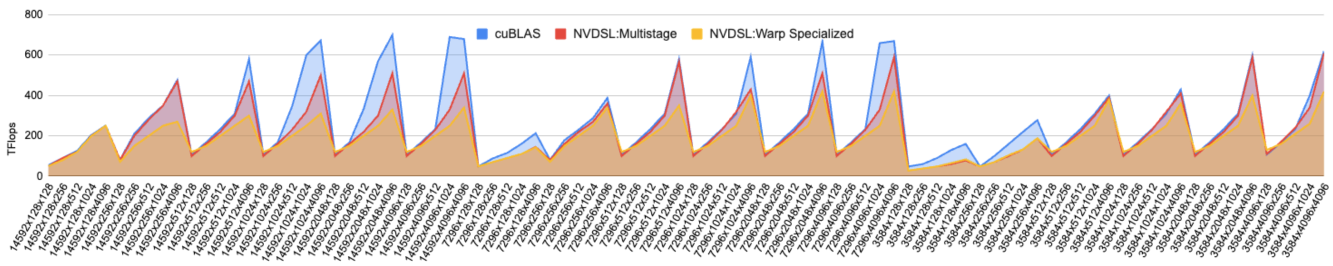

Figure 10. Performance of GEMM

tion sets up the mbarriers (`mbars`) and prefetches the TMA descriptors (Line 7). The producer *warpgroup* runs the `producer_loop` function to manage data loading with TMA (Lines 9-10), encapsulated in a `with prod` block to ensure the correct register size. The consumer *warpgroup* then performs MMA operations and the epilogue to store the results (Lines 12-14), wrapped in a `with cons` block.

5. Evaluation

We conducted experiments on a system equipped with an NVIDIA H100 SXM card, assessing the performance of *multistage* and *warp specialized* kernels in comparison to NVIDIA's cuBLAS, renowned for its out-of-the-box performance in GEMM problems.

Figure 10 presents a performance comparison. Our findings exhibit promising results, closely aligning with those of hand-optimized libraries. By integrating *warp specialization* in a ping-pong manner, facilitated by NVDSL's user-friendly kernel development environment, we anticipate achieving performance parity with cuBLAS. Additionally, the implementation of the cga cluster feature remains a focal point for further optimization. Notably, these advancements are readily achievable through `NVGPU` dialect's existing support, necessitating their integration within NVDSL for kernel development.

6. Related Works

To address tensor core utilization, both library and compiler approaches have been explored. NVIDIA's CUTLASS C++ (Thakkar et al., 2023) offers CUDA C++ template abstractions for high-performance GEMM implementations, but its usage with templates can be daunting, and template compilation can be slow.

From the compiler side the Triton compiler (Tillet et al., 2019) demonstrated the feasibility of a programming language tailored for tensor cores, achieving cuBLAS-like performance for previous GPU architectures. It lacks support for specialized features like warp specialization in NVIDIA Hopper architecture, limiting performance portability. The Halide(Ragan-Kelley et al., 2013) and TVM(Feng et al., 2023) uses scheduling language to express loops, TensorIR(Chen et al., 2018) generalizes high-level loop nest representation. However, it isn't clear that they can support sophisticated kernels like NVDSL does. The (Grover, Kunwar, 2024) from IREE compiler (The IREE Authors, 2019) lets you program thread block just like triton. Therefore, it will have expressibility issue for warp specialization.

7. Conclusion

In this article, we have introduced NVDSL, a Python-based DSL for writing fast GPU kernels. By using NVDSL, programmers can take advantage of GPU hardware features without needing to delve into low-level details. We demonstrated how NVDSL simplifies tensor core usage and warpgroup programming, allowing users to achieve cuBLAS-like performance with concise and readable code. Our implementation leverages the MLIR compiler and introduces support for the NVIDIA Hopper GPU.

Acknowledgements

I would like to express my sincere gratitude to Quentin Colombet, Manish Gupta, Adam Paszke, and Nicolas Vasilache for their valuable contributions to this work. Their thorough code reviews, encouragement, and suggestions significantly improved the quality of this research.

References

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.

Feng, S., Hou, B., Jin, H., Lin, W., Shao, J., Lai, R., Ye, Z., Zheng, L., Yu, C. H., Yu, Y., and Chen, T. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pp. 804–817, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3576933. URL https://doi.org/10.1145/3575693.3576933.

Grover, Kunwar. Turbine Kernels, 2024. URL https://github.com/nod-ai/techtalks/blob/main/C4ML_24_Turbine_Kernels.pdf.

NVIDIA. Parallel Thread Execution ISA Version 8.5, 2024. URL https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL https://doi.org/10.1145/2499370.2462176.

Thakkar, V., Ramani, P., Cecka, C., Shivam, A., Lu, H., Yan, E., Kosaian, J., Hoemmen, M., Wu, H., Kerr, A., Nicely, M., Merrill, D., Blasig, D., Qiao, F., Majcher, P., Springer, P., Hohnerbach, M., Wang, J., and Gupta, M. CUTLASS, January 2023. URL https://github.com/NVIDIA/cutlass.

The IREE Authors. IREE, September 2019. URL https://github.com/iree-org/iree.

Tillet, P., Kung, H.-T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.