15745 Final Report: Hardware-aware Compilation of Tensor Programs for Low-Precision Computing

Jinqi Chen
jinqic@andrew.cmu.edu

Zhibo Chen
zhiboc@andrew.cmu.edu

https://www.andrew.cmu.edu/user/zhiboc/15745-fa24/

1 Abstract

Inspired by some insights from Ladder[1], we investigated how hardware-aware optimizations impact the performance of generalized matrix multiply operations on GPU with quantized data types. We designed and implemented a novel compilation strategy for FP16 x INT4 kernel in TVM and compare its performance with the Marlin baseline. We find that our compiled kernel is able to better utilize memory bandwidth and is able to achieve a 2.x to 3.x performance speedup for small and medium sized matrix multiplication. We also discovered 0.5x speed up (slowdown) for large and real-world LLM matrix multiplication. As a stretch goal, we investigated the huge-precision computation in the case of large integer multiplications leveraging memory optimization on GPU and compared with CPU.

2 Introduction

2.1 Problem Setup

A majority of computations on GPU is bound by memory bandwidth. The core ALU operations happen on register files, while the data is initially transferred from CPU to the global memory. Before each computation, data has to be copied from global memory to shared memory, and from memory to register files. After a thread finishes computing, the data has to be moved back from the register files to shared memory, and then to global memory. The amount of time it takes to move data around usually surpasses the amount of time it takes to actually do the computation. So having a notion of coordinated data movement is crucial for optimizing GPU performance.

Our background work, Ladder, is trying to optimize the GPU performance by leveraging this particular

[1]Wang, L., Ma, L., Cao, S., Zhang, Q., Xue, J., Shi, Y., Zheng, N., Miao, Z., Yang, F., Cao, T., Yang, Y., & Yang, M. (2024). **LADDER: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation**. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024)*. USENIX Association. https://www.usenix.org/conference/osdi24/presentation/wang-lei

memory hierarchy. Their key observations are two folds, first, there are discretized data types that need to be computed but there may not be an exact matching hardware for the computation. For example, the user may ask GPU to compute multiplications between `int8` tensors and `fp32` tensors, but the GPU does not have integer - floating point multiplication points but only floating point - floating point multiplication points, so data of type `int8` needs to be converted to `fp32` before the multiplication. Second, the conversion traditionally happen in register files, i.e. the integer data needs to be brought all the way through shared memory to register files and then the conversion is applied. Wang et al. proposes different ways of transforming data across memory hierarchies and implement this through a compiler.

With the rising trend of quantization in deep learning workloads, more and more mixed-precision kernels are used in machine learning training pipelines that demand both efficient gradient calculations and reduced memory footprint. We focus on a specific kernel in our work, FP16xINT4. Leveraging TVM and insights from Ladder, our goal is to design a high-performance compilation technique for this kernel. Beyond low-precision (few bits) matrix multiplication, we also leverage the hardware-aware memory optimization insight and investigate the performance of large integer multiplication (many bits) on GPU.

2.2 Approach

We initially follow the compilation pipeline of Ladder, which compiles and transforms the tensor program through a series of hardware-aware memory operations and optimizations. We then analyze the performance of our compiled FP16xINT4 kernel, where in each step, we carefully analyze its performance bottleneck, and iterate on our design. We will further discuss our method in Section 4.

2.3 Contribution

In this work, we present a novel compilation technique of low-precision kernels on GPUs, specifically FP16xINT4, and bring the insights of hardware-aware optimizations. We compare the performance of our compiled kernel against the state-of-the-art Marlin baseline. Our observations and design principles can be applied to a wide family of low-precision kernels, including FP16xINT8, INT1$\times$INT8 etc. Our solution has important impact in the field of quantized machine learning systems, and provides valuable insights for TIR kernel optimization.

3 Background and Related Work

3.1 GPU Background

NVIDIA GPUs comprise an array of Streaming Multiprocessor (SM) elements that share a DRAM memory, known as Global Memory (GMEM) and an L2 cache. Each processing block in SM includes a warp scheduler, a Register File (RF), and an L0 instruction cache. The warp is the basic scheduling unit in CUDA. It usually comprises of 32 threads, which execute instructions in lockstep to ensure efficient utilization of GPU resources.

Tensor Cores are dedicated hardware units for accelerating matrix-multiply-accumulate (MMA) operations, pivotal in deep learning workloads. The latest Hopper GPU architecture supports a wide range of tensor cores for mixed-precision computation, such as INT8 (E4M3, E5M2), INT4 etc.

3.2 Mixed-Precision Inference on LLMs

LLMs typically involve billions of parameters, making them computationally expensive to run. Full-precision (FP32) operations, while accurate, require substantial memory bandwidth and processing power, which limits throughput and scalability. Mixed-precision inference addresses this challenge by representing weights and activations in lower precisions like FP8 and INT4, while selectively using higher precision, such as FP32 for accumulation or other critical calculations.

Mixed-precision inference can significantly accelerate LLM computations, reduce memory footprints, and decrease energy consumption without sacrificing model accuracy. This is especially vital for deploying LLMs in real-world applications. Moreover, advancements in hardware, such as Tensor Cores in NVIDIA's Hopper architecture, have been specifically optimized for mixed-precision operations, which helps quantized LLMs achieve their full potential.

3.3 TVM and TensorIR

Directly writing CUDA code for kernels like GEMM from scratch can be quite convoluted. Compilers such as TVM offers a high-level intermediate representation (IR) called TensorIR[2](TIR) that abstracts away the low-level details of CUDA programming, like error-prone indexing, datatype conversion, data alignment, ptx assembly, and allowing developers to focus on algorithm design and optimization strategies. TensorIR provides an explicit modeling of memory hierarchies and threading improves data reuse and reduces latency, which are critical for GPU performance. By leveraging tensor intrinsics automatically, TensorIR optimizes

[2]Feng, S., Lin, W., Ye, Z., Yu, Y., Hou, B., Shao, J., Zheng, L., Jin, H., Lai, R., & Chen, T. (2022). **TensorIR: An abstraction for automatic tensorized program optimization**. *arXiv*. https://doi.org/arXiv:2207.04296

operations like GEMM (General Matrix Multiplication), achieving performance on par with or exceeding hand-optimized libraries.

3.4 Previous Work

Recent advancements in low-precision computation have highlighted innovative strategies for optimizing mixed-precision kernels. The MARLIN[3] approach (Frantar et al., 2024) addresses the challenges of INT4 to FP16 conversions by employing binary manipulation techniques and asynchronous memory operations to streamline dequantization directly into Tensor Core-compatible layouts, which achieves near-optimal performance in batched LLM kernels.

Complementing this, the LADDER framework (Wang et al., 2024) proposes a hardware-aware compiler that systematically supports custom low-bit precision data types through tensor transformations. By separating data storage and computation, LADDER achieves efficient alignment with hardware memory hierarchies and accelerates mixed-precision operations. It introduces scheduling primitives and a hardware-aware optimization policy to address the performance challenges of aligning low-bit data access with coarse-grained memory systems.

4 My Method

There are many factors that can influence the kernel performance, thus must be carefully addressed when designing kernel optimizations. For example, data reuse can improve cache performance, carefully handling memory access patterns can reduce memory bank conflicts, efficiently partitioning workloads across computation units such as streaming multiprocessors (SMs), warps, and threads can effectively leverage high-speed on-chip resources, using software pipelining to overlap memory transfers with computation can hide the latency of memory loads, and so on.

In the sections below, we will discuss how we integrate these design principles in our compilation strategy for the FP16xINT4 kernel, and how these principles help improve performance.

4.1 Multi-Level Tiling

We follow the multi-level tiling strategy laid out by (cite) CUTLASS: efficient GEMM, which is a hierarchical tiling approach closely aligned with the CUDA programming model. In order to emphasize the scope

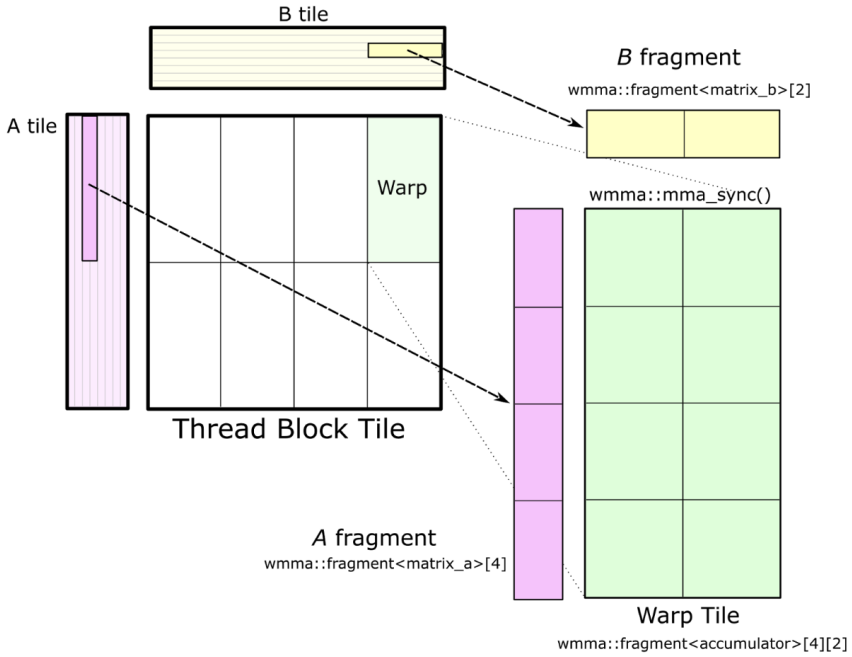
[3]Frantar, E., Castro, R. L., Chen, J., Hoefler, T., & Alistarh, D. (2024). **MARLIN: Mixed-Precision Auto-Regressive Parallel Inference on Large Language Models**. *arXiv*. https://doi.org/10.48550/arXiv.2408.11743

of execution among computation units, we use the execution scopes developed in TIR, like `T.kernel()`, `T.cta()`, `T.warp()`, `T.thread()`, which presents a more concise view of the kernel.

1. **Thread Block Tile**: We divide the original matrices A (MxK) and B (NxK) along their spatial dimensions M,N into subtiles of size BLK_M and BLK_N respectively, so that each thread block is responsible for computing a tile of (BLK_MxBLK_N) in the output matrix C. Furthermore, we divide the reduction dimension K into subtiles of size BLK_K, so that in each stage, we load a tile of A (BLK_MxBLK_K) and B (BLK_NxBLK_K) into shared memory. This can maximize data reuse and reduce global memory accesses.

2. **Warp Tile**: We further partition shared memory among warps, so that each warp is responsible for loading and computing a subtile of 16x16 in the output matrix C in parallel. We divide the spatial dimension BLK_M and BLK_N and reduction dimension BLK_K by 16 to calculate computation stages.

3. **Thread Tile**: Since each warp hold a 16x16 tile, we further partition it into subtiles of size 8 and assign them to 32 threads for warp-level MMA (Matrix Multiply Accumulate). This fully utilizes the potential of tensor cores for high throughput computation.



4

[4]Source: `https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/`

4.2 Tensor Cores

The NVIDIA Hopper architecture supports a wide range of data types for tensor cores, like FP32, FP16, BF16, INT8, INT4, FP8 etc. In the FP16xINT4 GEMM kernel, however, we cannot choose INT8 or INT4 tensor cores due to precision loss of the FP16 input, despite their higher computation speed. Thus, we choose FP16 warp-level MMA of shape m16n8k16. This incurs an additional burden of casting INT4 input to FP16 at some point in the kernel, which we shall discuss in the next section.

Since casting alters the data layout, we cannot utilize any 16x16 WMMA operations that relies on a fixed layout in the compiler. Our method is to directly specify and pass in the registers the starting offsets of A (8 elements), B (4 elements), and C (4 elements) respectively, following the MMA layout as described here [5]. This therefore forces us to forgo warp-level logical memory and rely solely on local physical memory.

4.3 Local Dequantization

In our work, we focus on standard weight quantization on matrix B, instead of group quantization. Assume 2 INT4 are packed into and represented by 1 INT8 throughout the article. Since naive type-cast from INT4 to FP16 is slow, we leverages CUDA's fast `lop3` instruction for logical operation on 3 inputs. We use the following ptx assembly in TIR, which efficiently dequantize 8 INT4 elements following the interleaved numeric conversion method in CUTLASS [6]. Observe that for 8 consecutive elements 0,1,2,3,4,5,6,7, the dequantized order by the above instruction is 4,0,6,2,5,1,7,3.

```
static constexpr uint immLut = (0xf0 & 0xcc) | 0xaa;
static constexpr uint BOTTOM_MASK = 0x000f000f;
static constexpr uint FP16_TOP_MAGIC_NUM = 0x64006400;
static constexpr uint MEDIAN_NUM = isSigned ? 0x64086408 : 0x64006400;
for (int i = 0; i < (N / 2); i++)
    asm volatile("lop3.b32 %0, %1, %2, %3, %4;\\n"
                 : "=r"(h[i])
                 : "r"(i4s >> (4 * i)), "n"(BOTTOM_MASK), "n"(FP16_TOP_MAGIC_NUM), "n"(immLut));
    asm volatile("sub.f16x2 %0, %1, %2;\\n" : "=r"(h[i]) : "r"(h[i]), "r"(MEDIAN_NUM));
```

There are multiple places where dequantization from INT4 to FP16 can happen. For example, a naive way is to dequantize INT4 in shared memory, but loading FP16 from shared memory to registers would

[5] Source: https://docs.nvidia.com/cuda/parallel-thread-execution/#warp-level-matrix-fragment-mma-16816-float

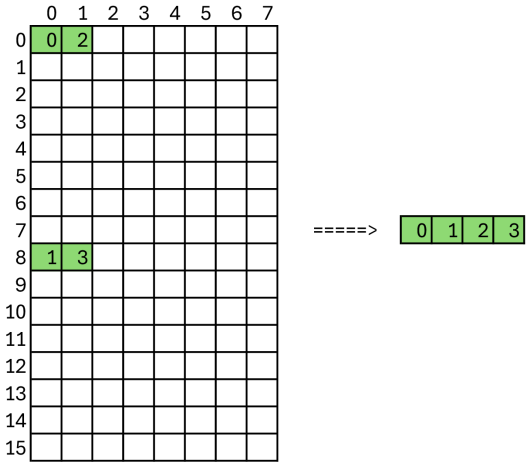
[6] Source: https://github.com/NVIDIA/FasterTransformer/blob/main/src/fastertransformer/cutlass_extensions/include/cutlass_extensions/interleaved_numeric_conversion.h

then increase memory traffic. Dequantizing in local memory would save memory bandwidth and allow faster execution of `lop3`, but it presents new challenges, as the change of data layout due to dequantization would break the MMA layout requirement. There are two ways to handle the break. The first method introduced by Ladder is to dequantize in registers and load back to shared memory for layout permutation. The second approach which we come up with is to load data into registers according to a permuted data layout, dequantize locally and directly use for MMA. The second approach is generally faster than the first one, because it avoids the overhead of extra data movement.

4.4 Shared-to-Register Layout

As mentioned above, we need to load a permuted layout of B into registers. This can happen from global to shared memory, or from shared to local memory. Emperical evidence suggests that both are similar in performance. Since loading a tile of B (16x8) from shared memory to registers requires 4 elements per thread, and there is no effective vectorized ptx instruction to achieve this (`ldmatrix` requires 8 elements), we decide to perform the permuted memory load here.

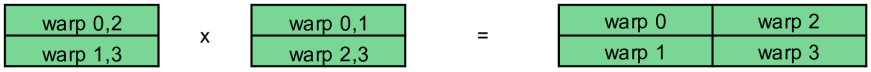
In the following, we will explain how the permutation happens by using ij to denote the (i,j) element in matrix B's 16x16 full form (i.e. where each INT4 element is stored) and use thread 0 as an example. Based on MMA layout, thread 0 should take 00 01 80 81 08 09 88 89 in registers for MMA. Based on the dequantization layout transformation, prior to dequantization, thread 0 should take elements 08 00 88 80 09 01 89 81. We discover that the best way of packing two consecutive INT4 together is row-based stride 8 packing. We load the packed INT8 from shared memory to local according to the following layout.



4.5 Warp Layout

Warp-level tiling can significantly improve the performance of the kernel, because all threads in a warp execute in a lockstep which reduces synchronization overhead, and memory accesses to the shared memory can be coalesced to avoid bank conflicts. A tricky part is how to partition the workload among warps.

In our kernel, we have 4 warps per thread block. There are three ways of partitioning such that each warp computes a subtile of the result. The most balanced one is shown in the figure below, where we assign warp 0,2 and warp 1,3 to the same subtile of A respectively, and warp 0,1 and warp 2,3 to the same subtile of B respectively. We can also assign warp 0,1,2,3 to the same subtile of A, and assign warp 0,1,2,3 to four different subtiles of B, or vice versa. This is a hyperparameter worth tuning in the next section.



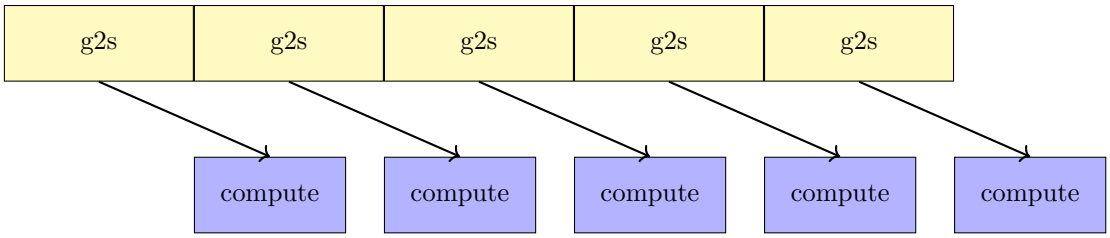
4.6 Hyperparameter Search

Hyperparameter tuning is a critical step of compiling kernels for high performance. Fine-tuning parameters such as tiling sizes, warp and thread partitions, memory access patterns, vectorization size etc. can align with hardware's unique constraints and specifications, like number of SMs, tensor cores, and cuda cores, memory bandwidth and sizes etc. In our kernel, we find the most critical tuneable hyperparameters are thread block tiling size (BLK_M, BLK_N, BLK_K) and warp partitioning strategy. We use a grid search to find the best hyperparameters that maximize the kernel performance, and results are shown in section 5.3.

4.7 Memory Load Pipelining

Prefetching can be used to hide the latency of memory loads. In each cycle, instead of waiting for loading A,B tiles into shared memory to complete to start computation, we can issue asynchronous copy operations and use software pipelining to load a few cycles ahead, and store the loaded data in a buffer.

There are two kinds of asynchronous copy in Hopper architecture: async group mechanism and mbarrier-based mechanism, and we choose the async group mechanism. The `cp.async` instruction bypasses L1 cache and register file when loading into shared memory, which makes overlapping computation and data movement easier. We also create buffers of size (pipeline_depth, xxx) to store the loaded data. At iteration i of the main loop, producer G2S stores in the ((i + pipeline_depth - 1) % pipeline_depth)-th buffer, and consumer reads from the (i % pipeline_depth)-th buffer for computation. In practice, we find that a pipeline of depth 2 suffices to achieve the best performance.



5 Experiment and Results

5.1 Experiment 1: FP16xINT4 GEMM performance

We set up our experiments on H100 GPUs, and profile on various configurations of matrix sizes. Input matrix A is FP16 (size MxK), matrix B is INT4 (size NxK), and output matrix C is FP16 (size MxN). We carried out the absolute CUDA time measurement using the torch profiler. The diagrams below shows the absolute speedup of our compiled kernel performance compared with the Marlin baseline.

We carefully picked five different benchmarks to measure different aspects of the kernel performance.

1. **Small Matrices** test cases evaluate cache-friendly scenarios where matrices fit entirely in L1/L2 cache for low-latency performance.

2. **Medium Matrices** test cases evaluate GPU shared memory utilization and thread block synchronization.

3. **Large Matrices** test cases stress test global memory bandwidth and compute efficiency for large workloads.

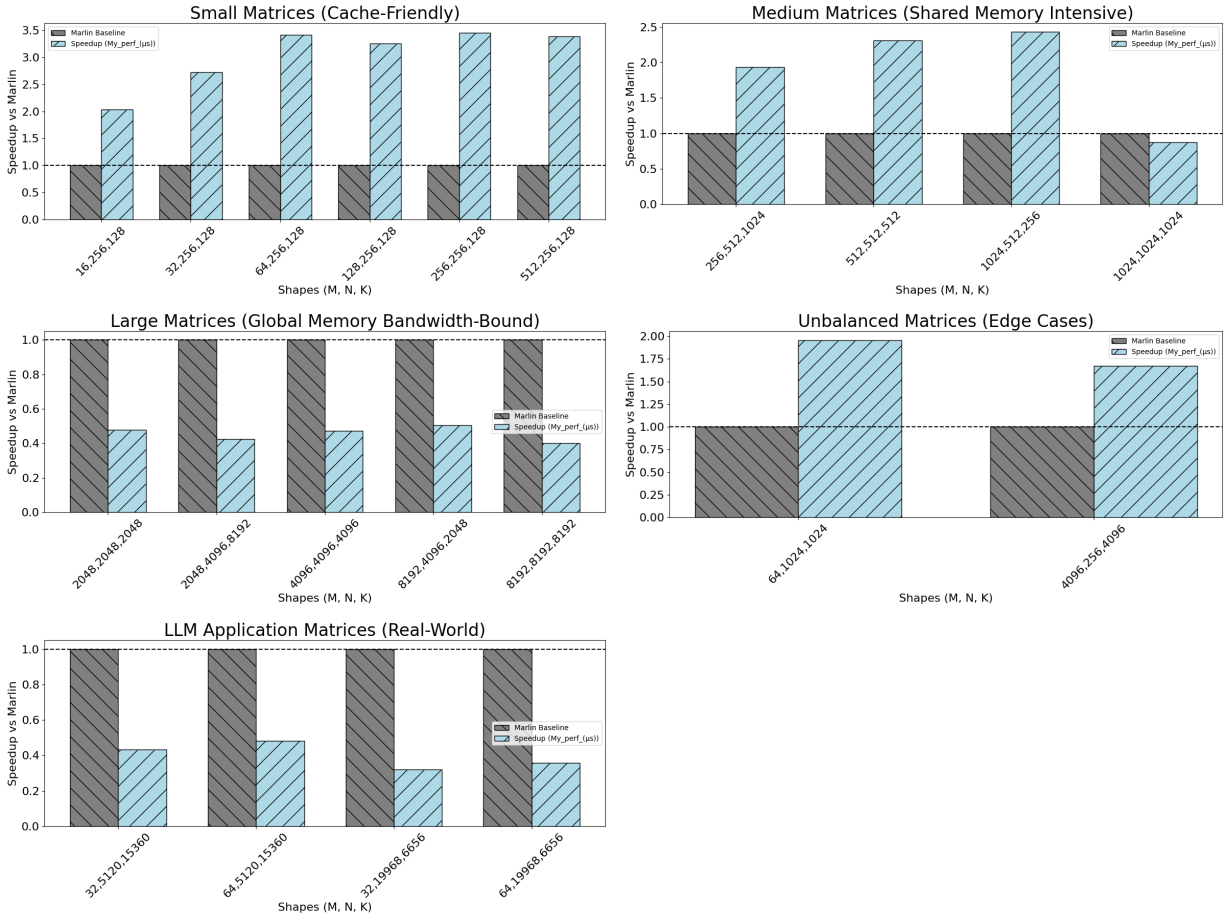
4. **Unbalanced Matrices** test cases test non-square matrices with extreme aspect ratios.

5. **LLM Application Use Cases** reflect matrix sizes in real-world LLM layers.

We summarize the main points using a table.

Benchmark name	Aspects under Test
Small Matrices	Cache-Friendly
Medium Matrices	Shared Memory Intensive
Large Matrices	Global Memory Bandwidth-Bound
Unbalanced Matrices	Edge Cases
LLM Application Use Cases	Real-world Applications, Transformer Models

Here are the results on each set of test cases. On the horizontal axis is the M, N, K size, and the vertical axis is the relative speedup of our compiled kernel vs Marlin baseline.

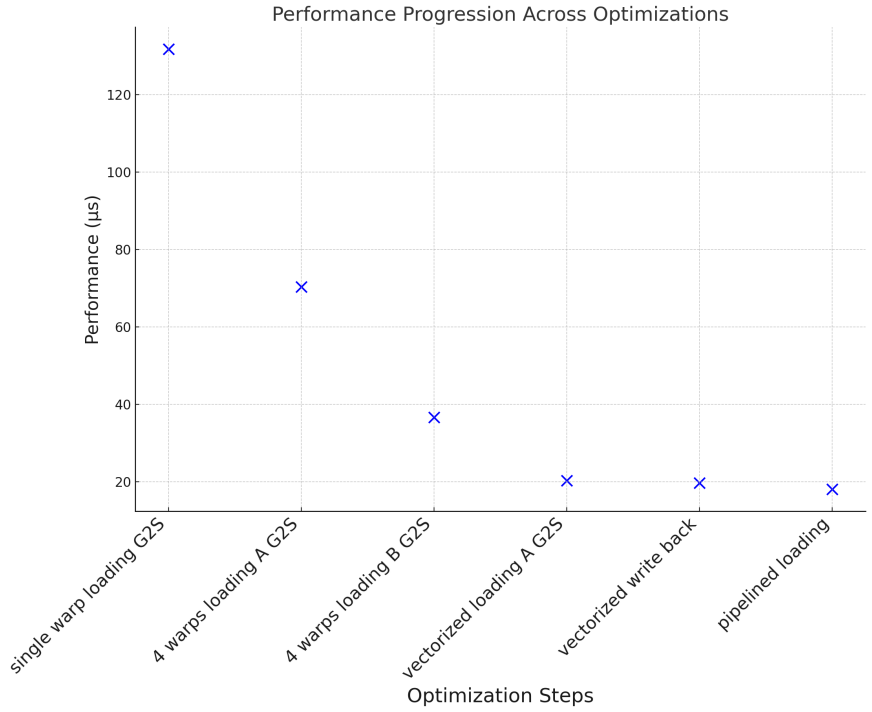


We find that we can outperform the Marlin baseline by 2.x to 3.x times for small and medium sized matrices. However, for large and real-world LLM matrix multiplication, we find a 0.5x speed up (slowdown) compared to the baseline. This is potentially due to different choices of number of warps and saturation of GPU resources in thread blocks.

5.2 Analysis effect of each step of optimization

We evaluate individual optimization strategies that improves the performance of the kernel. In the following diagram, the horizontal axis shows the name of the optimization strategy that we applied and the vertical axis shows the running time of this optimization. These changes are implemented sequentially and the decreasing trend of the data can be seen as our journey in getting a better performance.

The diagram identifies a few key optimization steps impacting performance: use warps to parallel load data from global to shared memory, use vectorized load, and use pipelining to hide memory latency.



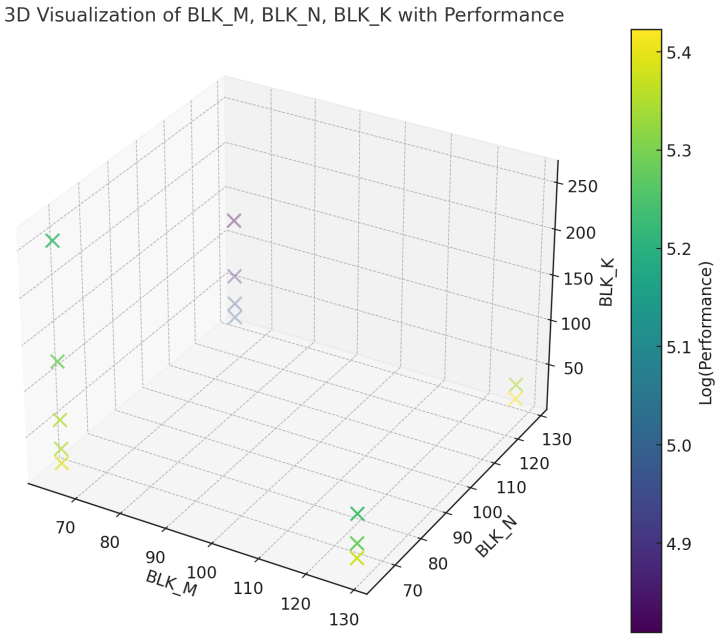
5.3 Hyperparameter search

We analyzed the effect of hyperparameter, specifically the thread block tiling size, on our performance. We gained this insight by the third paper in our assigned reading list (recent research in optimizations), where facebook talks about how they searched for hyperparameters in their server clusters. [7]

The parameter we searched include two values along each `BLK_M` and `BLK_N` dimensions, and several values along the `BLK_K` dimension. We tested two values of `BLK_M` and `BLK_N` (64, 128) and `BLK_K` values from (16, 32, 64, 128, 256). This choice is due to the constraint of the size of shared memory.

This search tells us that the optimal configuration is `BLK_M = 64, BLK_N = 64, BLK_K = 128`, with a warp partition of 2,2.

[7] HHVM Performance Optimization for Large Scale Web Services. ICPE '23, April 15–19, 2023, Coimbra, Portugal © 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 979-8-4007-0068-2/23/04. . . https://doi.org/10.1145/3578244.3583720



6 Stretch Goal (Big integer multiplication)

While matrix multiplication involves many multiplication of small numbers, big integer multiplication involves multiplication of large numbers. However, the basic idea is still the same, large numbers are broken into individual multiplications of small numbers and then accumulated together. We investigated the performance improvement of large integer multiplication on GPU compared to a brute force algorithm on CPU. Previous work by Dieguez et al. [8] suggests that the FFT-based method (which is used by GMP) doesn't offer high precision results for extremely large integers. What they instead have shown is that the traditional method of multiply-accumulate with parallel carry propagation is more suitable to get precise results.

Essentially, multiplying an m-digit by an n-digit integer results in an $m+n$-digit integer. The method is to compute a matrix of size $m \times n$ where each element is the product of the corresponding digits in the two numbers. The final result is then the sum along diagonals of the matrix, where the extra digits are carried over to the next diagonal.

6.1 Algorithm

Bruteforce. Given two large integers as byte arrays, we have implemented a brute force algorithm that does traditional multiply-accumulate with sequential carry propagation, starting from the least significant

[8] Dieguez AP, Amor M, Doallo R, Nukada A, Matsuoka S. Efficient high precision integer multiplication on the GPU. The International Journal of High Performance Computing Applications. 2022;36(3):356-369. https://doi.org/10.1177/10943420221077964

digit.

CUDA V1. Then we implemented a CUDA-based multiplication algorithm that uses the COOPERATIVE GROUP feature of the NVIDIA GPUs. We launch a single thread for each digit in the output. At each lock step, each thread computes a multiplication of the corresponding digits in the two input numbers, adds the current carry to it, stores it as the result, and update the carry of the next digit to the overflow result of the addition. This process is repeated until there are no more numbers to add and no more carries to propagate. By sequentially assigning threads to digits, this utilizes data locality and memory bandwidth.

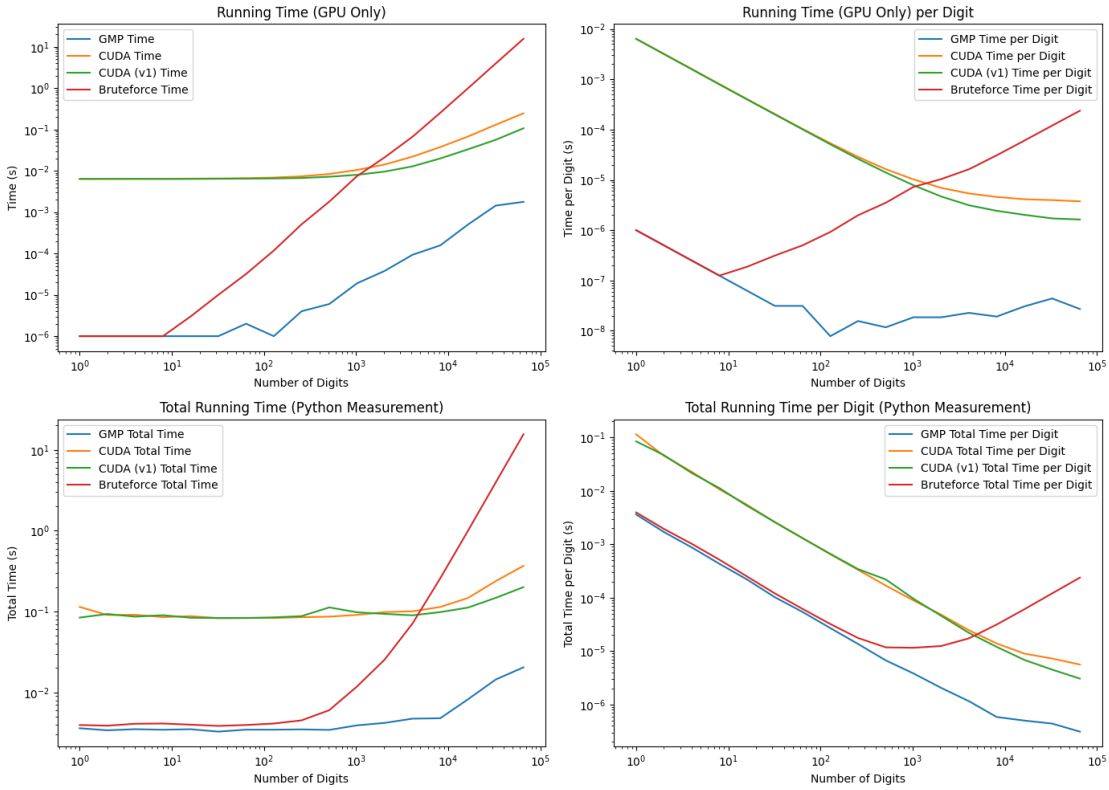
CUDA V2. V1 has a limitation. CUDA has a maximum number of threads per cooperative group (65535). And thus, we cannot use V1 to multiply numbers with more than 65535 digits in the result. We implemented a V2 version that uses software pipelining. We utilize a moving window of active output digits, starting from the lowest significant bit. Once a threads finish computing, it will move onto compute the next digit. The most significant thread propagates the carry along the way. This process repeats until all digits are computed.

GMP.

We also compare our CUDA implementation with the GMP library, which is a state-of-the-art library for arbitrary precision arithmetic. We use the `mpz_mul` function in GMP to multiply two large integers.

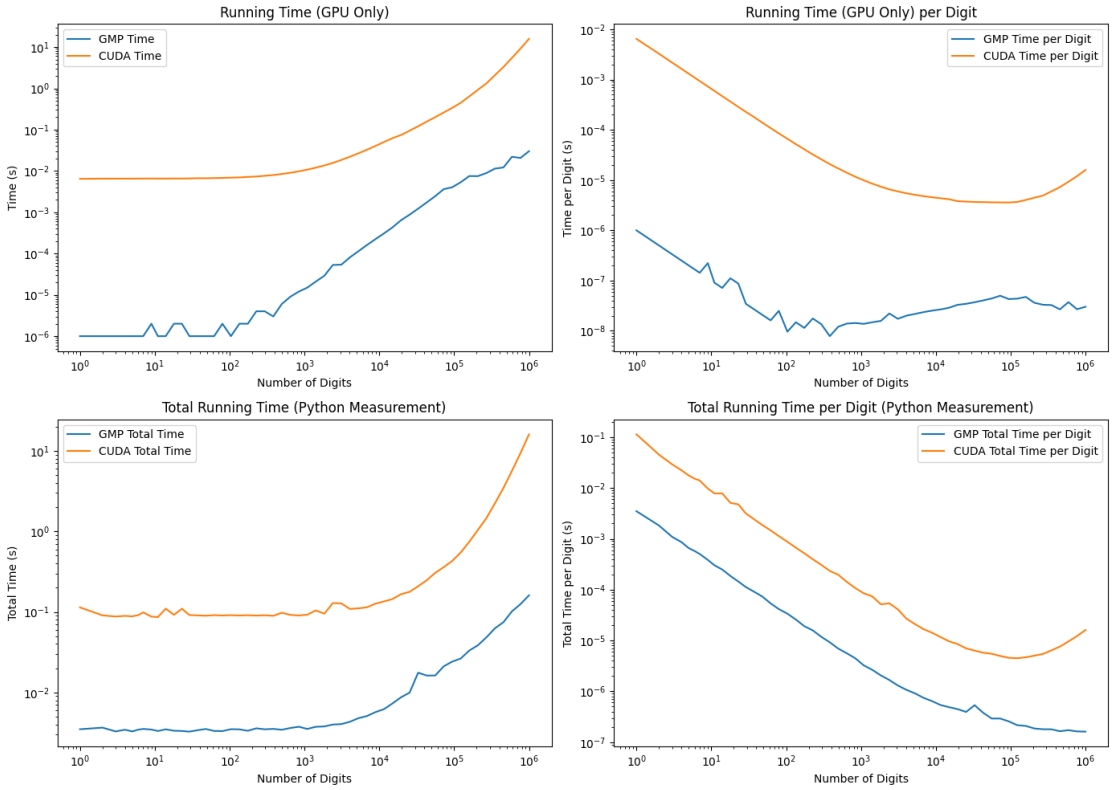
6.2 Experiment Results

In the experiment, a digit is defined as a byte. So that we can have efficient loading and storing of the data. We randomly generate two digits of size 2^n where n starts ranges from 1 to 65535. We compare the running time (GPU Only) which is the running time excluding the time to read and copy data, and the total time (python measurement) which is the end-to-end time measured in Python. Since it takes more time to compute digits of larger sizes, we also measure the running time per digit. The diagrams are plotted on a log scale.



We see that the CUDA implementations is able to outperform the brute force implementation on CPU starting with digits of size 1000, and is able to achieve a 77x speed up for v1 and a 42x speed up for v2. However, the overall performance is below GMP. For the computation time per digit, the bruteforce version steadily rises after more than 10 digits, while the CUDA version continues to decrease.

The pipelined version is certainly slower than the unpipelined version, but is able to handle larger numbers. We show the performance of the pipelined version (compared to GMP) in the following diagram.



We see that both CUDA and GMP is taking more time, but the CUDA V2 version's performance when the number of digits exceeds 60000 starts to uptick and will eventually be much higher. We suspect this is due to the synchronization overhead of the cooperative group.

7 Surprises and Lessons Learned

The lessons we learned are two-folds.

First, the address passed into MMA by 8 threads is used for collective decision on their offset in 8x8 matrix. Although we pass in address offsets according to result of `ldmatrix`, it is not the final values held in each thread's registers. We discover this when debugging local memory on the kernel's CUDA codegen.

Second, we learned some ways to analyze performance bottleneck. We found that hyperparameter tuning is critical for performance optimization. Roofline analysis also helps by comparing current performance with theoretical peak performance.

8 Conclusion and Future Work

We researched on the hardware-aware compilation of low-precision kernels, specifically the FP16 x INT4 GEMM kernel, which is critical in quantized machine learning applications. We presented a compilation

technique that utilizeds multi-level tiling, tensor cores, local dequantization, shared-to-register layout, warp layout, memory load pipelining, and hyperparameter search to achieve the best performance possible. On small and medium sized matrices, we can achieve better performance by utilizing the hardware resources on GPU more efficiently than the current state-of-the-art. However, we do discover some limitations in our approach that the performance is not as good for large and real-world LLM matrix multiplication. We conjecture that this performance degradation is due to the warp-size we picked during the construction of our kernel. This is left as future work to investigate.

As a stretch goal, we investigated the performance of large integer multiplications which is opposed to low-precision and have huge precision (0.4M bytes) by leveraging memory optimizations and computation on GPU compared to CPU. We find that the hardware-aware technique on GPU is able to achieve thousands of times speed up compared to CPU by parallelizing the computation, but still underperformance the state-of-art CPU libraries, which is carefully engineered to use some advanced methods that reduces the algorithmic complexity.

In future work, we plan to bring more hardware-aware techniques into the design of TIR and TVM, which can enable developers to write high-performance kernels more easily.

9 Distribution of Credit

Credits are distributed 50-50 between the two authors.