ACCELERATING A TRITON FUSED KERNEL FOR W4A16 QUANTIZED INFERENCE WITH SPLITK WORK DECOMPOSITION

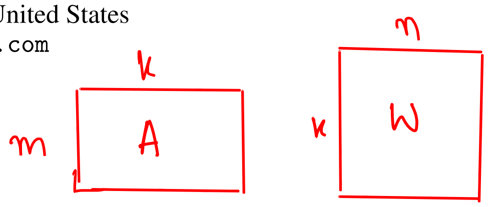
Adnan Hoque
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
adnan.hoque1@ibm.com

Less Wright
Meta AI
less@meta.com

Chih-Chieh Yang
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
chih.chieh.yang@ibm.com

Mudhakar Srivatsa
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
msrivats@us.ibm.com

Raghu Ganti
IBM T.J. Watson Research Center
Yorktown Heights, NY, United States
rganti@us.ibm.com



ABSTRACT

We propose an implementation of an efficient fused matrix multiplication kernel for W4A16 quantized inference, where we perform dequantization and GEMM in a fused kernel using a SplitK work decomposition. Our implementation shows improvement for the type of skinny matrix-matrix multiplications found in foundation model inference workloads. In particular, this paper surveys the type of matrix multiplication between a skinny activation matrix and a square weight matrix. Our results show an average of 65% speed improvement on A100, and an average of 124% speed improvement on H100 (with a peak of 295%) for a range of matrix dimensions including those found in a llama-style model, where $m < n = k$.

Keywords Deep Learning · Foundation Models · Neural Network Quantization · Triton · GPU · Parallel Programming

1 Introduction

There has been a large amount of research performed on designing efficient General Matrix Multiplication (GEMM) kernels [1] [8]. However, there are far fewer publications on how to design high-performance kernels for memory bound computations, such as those commonly found in inference. For inference, matrix multiplications are usually memory bound because of the problem size (m, n and k) creating skinny matmuls where the small dimension is the batch size (i.e. 1-16), along with GPU compute and memory throughput limitations. The implementation we propose shows promising results for this case. Rather than the traditional Data Parallel (DP) decomposition, we leverage a SplitK work decomposition with atomic reductions, and integrate it with dequantization to provide a one step fused dequant and matrix multiply kernel. We implement our kernel in Triton and provide the source code on GitHub [3]. We chose to implement our kernel in Triton, as it provides an easy-to-use interface and also cross-hardware compatibility. Thus, current and future iterations of this kernel can cater to the growing diversity of software/hardware stacks found in workloads across the industry.

2 Method

Our kernel will receive as input an FP16 activation matrix, a packed (quantized) int32 weight matrix, and the scale and zero parameters that will be used to dequantize the weight matrix. The weight matrix is dequantized, scaled and shifted using bitwise operations. Our implementation is tailored to respect GPTQ-style int4 quantization, but the method is relatively general purpose and can be extended to other methods of n-bit quantization. This is then integrated with the optimized SplitK GEMM, an algorithm that first appeared in the CUTLASS library [2]. Our kernel is inspired by the Triton FP16 SplitK implementation [9]. SplitK launches additional thread blocks along the k dimension to calculate partial sums. The partial results from each thread block are then summed using an atomic reduction 1. Thus, our implementation is a fused kernel that performs dequantization, GEMM and atomic reduction in a single kernel launch with notable performance improvements.

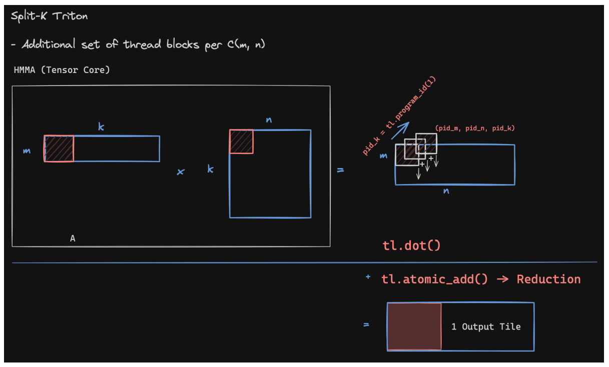


Figure 1: SplitK Thread Block Level

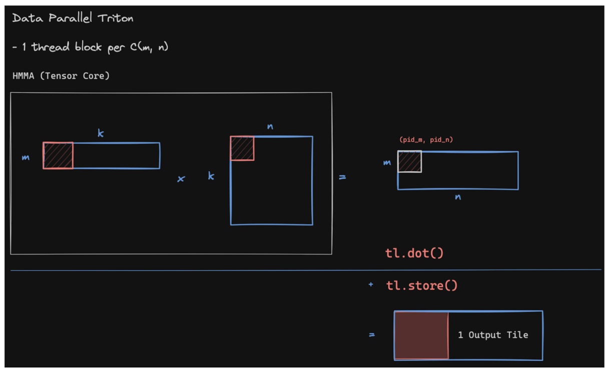


Figure 2: Data Parallel Thread Block Level

2.1 Atomics - Enabling the SplitK strategy

SplitK effectively decomposes the work into finer grained partitions as compared to traditional data parallel block style (tiling), and thus allows for more evenly balanced resource usage of the SMs. On an A100 with our kernel, this results in waves per sm increasing by 61% versus the standard data parallel block tiling. This finer grained decomposition is enabled by the use of atomic operations. In our case, we leverage the atomic add function so that as thread blocks complete their inner accumulation step, which represents a partial portion of the final result, they can safely update their partial results directly to the C output memory. This same output memory will also be updated by other thread blocks working on subsets that will contribute to the final aggregated multiply-accumulate output results for the given C output tile. Thus, ensuring that thread blocks both update the latest results, and have exclusive access while writing their result, is paramount and achieved via atomic adds. Note that there is a tension between the improvements from finer grained SM work distribution, and the overhead of thread blocks contending for exclusive write access to the same output buffer. This effect was seen on an A100 where increasing the SplitK parameter from 4 to 16, resulted in a steady degradation of performance as the matrix sizes increased, presumably due to greater wait times per thread block to get exclusive write access to the same memory output buffer.

→ Atomic operations increase latency.

2.2 Wave Quantization effects - SplitK improves as GPU SM count improves

With our SplitK kernel, we saw proportionate performance gains moving from an A100 to H100 relative to the data parallel block approach. As newer GPU's continue to have both more SM's and each SM at the same time becomes more capable, standard block tiling GEMM's will find steadily greater difficulty in avoiding the effects of wave quantization inefficiency and finding efficient work decomposition tiling sizes. To clarify, wave quantization inefficiency refers to the effect where the final wave of a tiled decomposition may only have a small subset of SM's active, with the remaining SM's idling. This effect is usually mitigated by having oversubscription, where larger amounts of tile production work is assigned to idle SM's. However, newer GPU's have more SM's, which means fewer waves needed to complete the desired tile ouputs/work. In addition, newer GPU's have more powerful SM's, which require larger blocking tile sizes, which also leads to fewer waves needed. Execution schedules with fewer waves are much more prone to wave quantization inefficiency, as the likelihood of total output tiles being oversubscribed and scheduled as an even multiple of SM's rapidly diminishes, thus rendering the final wave likely to use only a small portion of GPU resources. This effect was indirectly shown when moving our fused kernel from A100 to H100 GPUs. The H100 has 33% greater SMs as well as more capable SM's. Intuitively this should result in greater wave quantization inefficiency for data parallel, while SplitK can effectively leverage the SM improvements by simply increasing the SplitK hyperparam (from 4 on A100 to 8 on H100). As a result, compared to the data parallel approach, the average gain with our SplitK kernel on H100 nearly doubled with 124% average performance gain, versus the A100 average gain of 65%.

New GPUs need lesser waves

but higher chances of wave quantization

Explained more here

https://
www.thonking.ai/p/
what-shapes-do-matrix-

2.3 Algorithm

Algorithm 1 SplitK GEMM in Triton

```
1: Initialization of block and thread indices
2: $pid \leftarrow$ tl.program_id$(0)$
3: $pid\_k \leftarrow$ tl.program_id$(1)$
4: $num\_pid\_k \leftarrow$ tl.cdiv$(k, block\_k \times split\_k)$
5: Compute thread tile offsets
6: $offs\_m, offs\_n, offs\_k \leftarrow$ ComputeOffsets$(pid, pid\_k, block\_m, block\_n, block\_k)$
7: Initialize accumulators
8: $acc \leftarrow$ tl.zeros$((block\_m, block\_n)$, dtype $=$ tl.float32$)$
9: Main computation loop
10: for $k \leftarrow 0$ to $num\_pid\_k$ do
11:     Load input tiles and apply quantization
12:     $a, b \leftarrow$ LoadAndDeQuantize$(a\_ptr, b\_ptr, offs\_m, offs\_n, offs\_k)$
13:     Perform matrix multiplication
14:     $acc \leftarrow acc +$ tl.dot$(a, b)$
15:     Update pointers for next iteration
16:     $a\_ptrs, b\_ptrs \leftarrow$ UpdatePointers$(a\_ptrs, b\_ptrs, block\_k, split\_k)$
17: end for
18: tl.atomic_add$(c\_ptrs, acc)$        ▷ Synchronization of partial sums across thread blocks
```

3 Performance

We focused our experiments on M = 1 - 16, as this corresponds to a batch size of 1 - 16, a typical batch size range we might see in LLM inference. As our main contribution is a fused int4 dequantization kernel with a modified decomposition strategy, we explore the performance of the SplitK kernel vs the traditional DP kernel. The key difference is that in the DP decomposition, a single thread block is singly responsible for calculating the aggregate multiply-add accumulation, which produces an output tile in the C matrix. 2 We tested on both NVIDIA A100 and NVIDIA H100.

3.1 SplitK vs Data Parallel Results

We performed our performance analysis on a variety of different GPUs, with M = 1 - 16 and varying n = k. We conducted our tests on the NVIDIA A100 80GB SXM, A100 40GB PCIe and H100 PCIe to get a more comprehensive set of results. For both batch sizes and across all GPUs, SplitK performed much better than the DP decomposition.

3.2 M=1

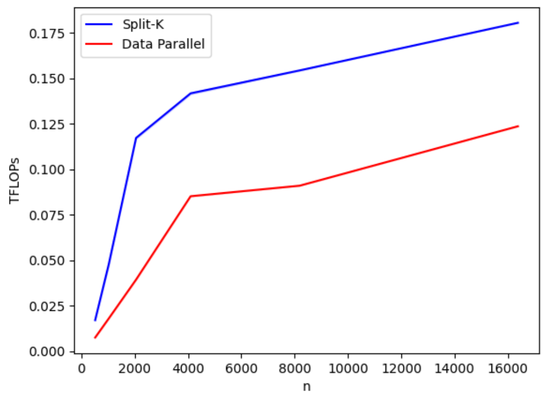


Figure 3: SplitK vs Data Parallel TFLOPS A100 40GB

N	K	SplitK [TFLOPS]	Data Parallel [TFLOPS]
512	512	0.01	0.07
1024	1024	0.04	0.04
2048	2048	0.11	0.08
4096	4096	0.14	0.09
8192	8192	0.15	0.09
16384	16384	0.18	0.12

Table 1: SplitK vs Data Parallel TFLOPS A100 40GB

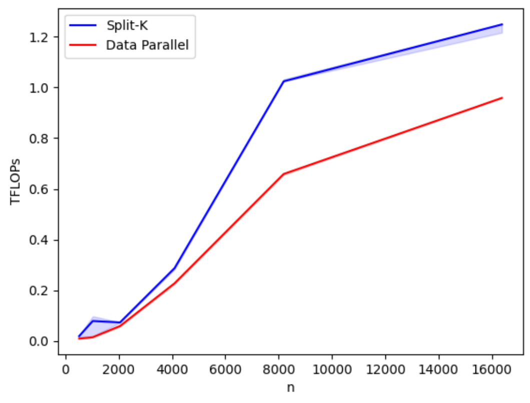


Figure 4: SplitK vs Data Parallel TFLOPS A100 80GB

N	K	SplitK [TFLOPS]	Data Parallel [TFLOPS]
512	512	0.02	0.01
1024	1024	0.01	0.01
2048	2048	0.06	0.04
4096	4096	0.22	0.18
8192	8192	1.03	0.66
16384	16384	1.25	0.96

Table 2: SplitK vs Data Parallel TFLOPS A100 80GB

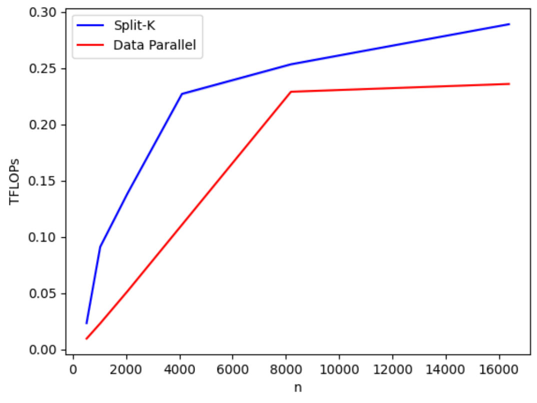
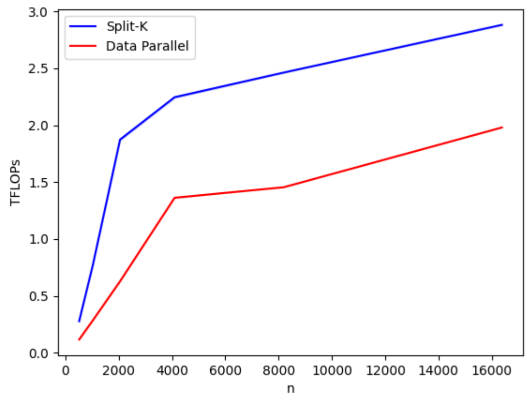


Figure 5: SplitK vs Data Parallel TFLOPS H100

N	K	SplitK [TFLOPS]	Data Parallel [TFLOPS]
512	512	0.28	0.12
1024	1024	0.77	0.28
2048	2048	1.85	0.62
4096	4096	2.25	1.36
8192	8192	2.46	1.45
16384	16384	2.87	1.98

Table 3: SplitK vs Data Parallel TFLOPS H100

3.3 M=16



N	K	SplitK [TFLOPS]	Data Parallel [TFLOPS]
512	512	0.3	0.1
1024	1024	0.8	0.3
2048	2048	1.9	0.6
4096	4096	2.2	1.4
8192	8192	2.5	1.5
16384	16384	2.9	2.0

Table 4: SplitK vs Data Parallel TFLOPS A100 40GB

Figure 6: SplitK vs Data Parallel TFLOPS A100 40GB

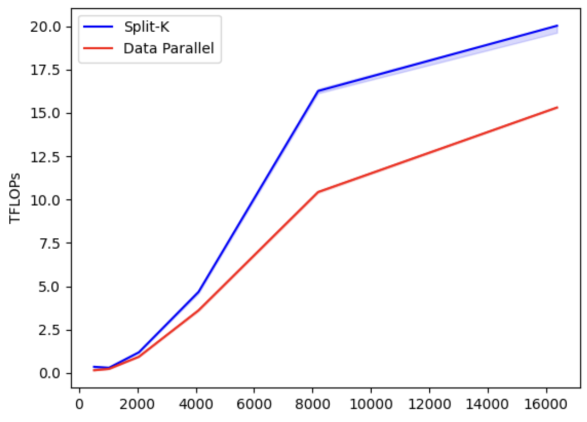


Figure 7: SplitK vs Data Parallel TFLOPS A100 80GB

N	K	SplitK [TFLOPS]	Data Parallel [TFLOPS]
512	512	0.3	0.1
1024	1024	0.3	0.2
2048	2048	1.1	0.9
4096	4096	4.5	3.5
8192	8192	16.3	10.4
16384	16384	20.0	15.3

Table 5: SplitK vs Data Parallel TFLOPS A100 80GB

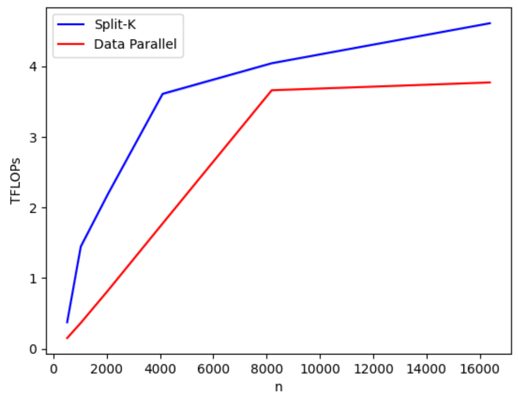


Figure 8: SplitK vs Data Parallel TFLOPS H100

N	K	SplitK [TFLOPS]	Data Parallel [TFLOPS]
512	512	0.4	0.2
1024	1024	1.4	0.2
2048	2048	2.2	0.9
4096	4096	3.6	1.7
8192	8192	4.1	3.7
16384	16384	4.6	3.8

Table 6: SplitK vs Data Parallel TFLOPS H100

The average speedup for the SplitK strategy is 1.24x, 1.14x and 0.64x on the H100, A100 40GB and A100 80GB respectively. We also tested a variety of SplitK settings and found varying optimal tile splitting factors. We found split_k = 4 and and split_k = 8 produced the best results on the A100 80GB and H100 respectively. For these tests, we fixed the tile sizes, number of warps, and number of stages to isolate the impact of SplitK.

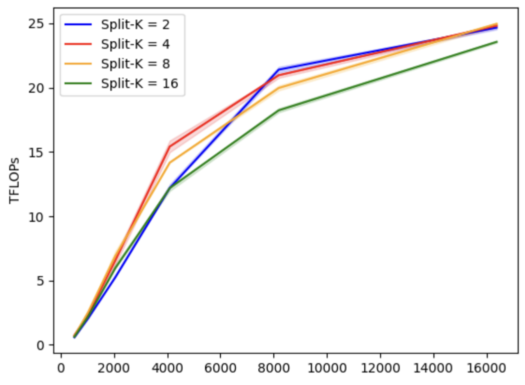


Figure 9: SplitK Comparison of splitting factor, A100

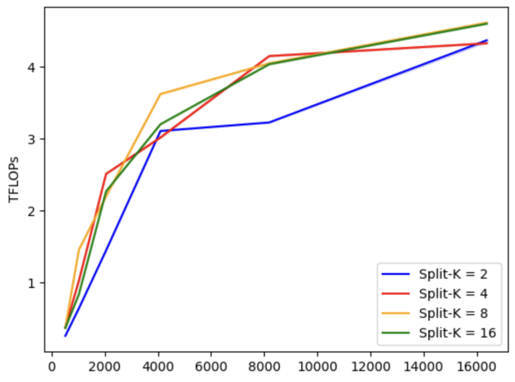


Figure 10: SplitK Comparison of splitting factor, H100

3.4 SplitK vs Data Parallel NVIDIA Nsight Compute

Statistics gathered from NVIDIA Nsight Compute [4], shown below, suggest that the SplitK kernel performs better than the DP Kernel mainly due to increased global memory throughput. These metrics were collected for a single test case where m=16, n= 4096 and k=4096.

Metrics	**SplitK**	**Data Parallel**
Latency	27.90us	52.93us
Global Memory Throughput	313 GB/s	161 GB/s
Grid Size	512	128
Registers	92	150
Shared Memory Usage	102.40KB	167.94KB
Block Limit (Registers)	5	3
Block Limit (SMEM)	5	2
Achieved Occupancy	27.75	7.55
SM Utilization	43.05%	20.75%

Table 7: Nsight Compute Metrics

The SplitK kernel launches 4x more thread blocks than the DP Kernel. Each thread block is responsible for calculating a partial result of an output tile as opposed to a complete tile. The thread blocks in the SplitK kernel then, notably have less work to do compared to the DP Kernel, which leads to better load balancing across SMs. This intuition can be directly confirmed by the 2x improvement in SM utilization.

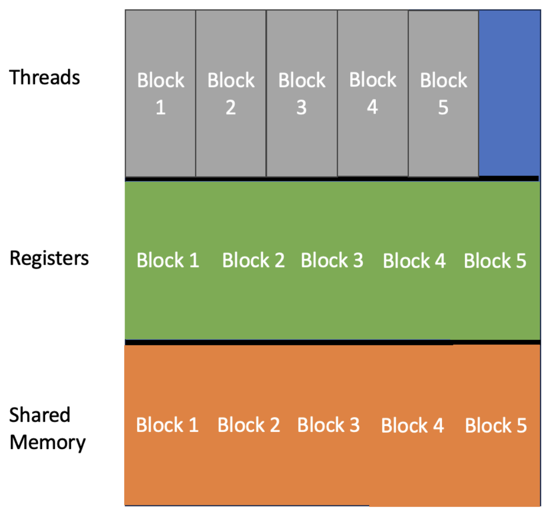


Figure 11: SplitK A100 SM Resource Usage

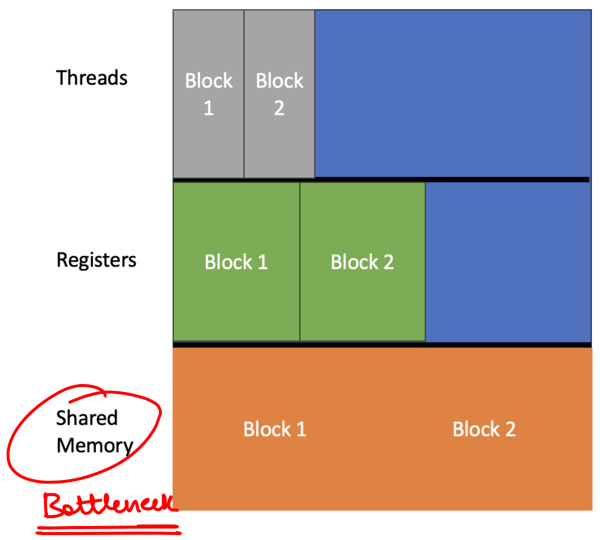


Figure 12: Data Parallel A100 SM Resource Usage

→ If less usage of Registers and shared memory, more threads can work in parallel ⇒ Better occupancy.

We note that the decreased register and shared memory usage in the SplitK kernel results in a nearly 4x improvement in occupancy. The DP kernel, by contrast, is shared memory limited. This means that the shared memory resource requirements of each thread block limits the amount of active thread blocks on an SM. In contrast, the SplitK kernel has thread block limit of 5, a 2.5x improvement due to reduced shared memory and register use. This difference is depicted in 11 and 12. The increase in occupancy and global memory throughput are both important metrics in understanding the latency improvement in the SplitK kernel. The increase in both metrics are roughly proportional to the improvement in latency. We may also use the Warp Scheduler Statistics to further understand the latency hiding characteristics of the SplitK kernel.

Metrics	**SplitK**	**Data Parallel**
Active Warps	4.45	1.21
Eligible Warps	0.67	0.20
Issued Warps	0.43	0.19
Issued IPC Active	1.72	0.75

Table 8: Scheduler Statistics

The SplitK kernel has 2x more warps in the issued slot. This illustrates the algorithms superior ability to hide memory latency. The increase in issued slot utilization is able to increase our compute pipeline utilization. This can be seen by the 1.3x increase in issued instructions per clock cycle (IPC).

3.5 H100 vs A100

We note the following performance specifications of the A100 [5] and H100 [6], as our results indicated a varied set of optimal hyper parameters across tests. These key specifications can be used to help understand those differences.

Feature	**NVIDIA H100 80GB PCIe**	**NVIDIA A100 80GB SXM**	**NVIDIA A100 40GB PCIe**
Architecture	Hopper	Ampere	Ampere
SMs	132	108	108
FP16 Tensor Core	1513 TFLOPS	312 TFLOPS	312 TFLOPS
Memory	80GB HBM3	80 GB HBM2	40GB HBM2
Memory Bandwidth	2.0 TB/s	2.0 TB/s	1.5TB/s
L2 Cache	50MB	40MB	40MB
L1 Cache/SM	256KB	192KB	192KB

Table 9: Comparison of NVIDIA H100 and A100 GPUs

Notably, the smaller form factor A100 sees an increased speedup when using SplitK, compared to the larger form factor (1.14x vs 0.64x).

This is intuitively explained by the fact that on the smaller A100, the memory bandwidth speed is 31% slower, and thus for small problem sizes, the kernels are comparatively more memory bound than when run on the larger (faster memory bandwidth) A100s. The finer decomposition and larger grid of the SplitK algorithm thus provides proportionately greater benefit (1.14x vs 0.64x) relative to the A100 80GB with higher memory bandwidth.

4 Conclusion

We present an optimized Triton kernel implementation for quantized matrix-matrix multiplications in inference workloads, where the problem is memory bound. Our implementation is a fused kernel that performs dequantization and GEMM via SplitK atomic reduction. We benchmark m, n and k values that are relevant for llama-style inference inputs and show an average of 65% speed improvement on A100, and 124% speed improvement on H100 (with a peak of 295%) compared to traditional blocked data parallelization strategies. We provide kernel level analysis to explain the speedups as being driven by a finer grained work decomposition via SplitK algorithm. This results in greater SM occupancy, and thus improved global memory throughput via latency hiding, as well as reduced wave quantization inefficiency. A potential future direction would be to explore the natural successor to SplitK, StreamK. The StreamK [7] decomposition may potentially enable even finer grained optimal work decomposition resulting in additional performance improvements for GEMM workloads.

References

[1] *cuBLAS*. URL: `https://docs.nvidia.com/cuda/cublas/index.html` (visited on 12/22/2023).

[2] *cutlass/media/docs/efficient_gemm.md at main · NVIDIA/cutlass*. GitHub. URL: `https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md` (visited on 12/18/2023).

[3] *foundation-model-stack/triton at triton · foundation-model-stack/foundation-model-stack*. URL: `https://github.com/foundation-model-stack/foundation-model-stack/tree/triton/triton` (visited on 12/22/2023).

[4] *Nsight Compute*. Archive Location: Nsight Compute. URL: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html (visited on 12/20/2023).

[5] *NVIDIA Ampere Architecture*. NVIDIA. URL: https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper (visited on 12/21/2023).

[6] *NVIDIA H100 Tensor Core GPU Architecture Overview*. NVIDIA. URL: https://resources.nvidia.com/en-us-tensor-core (visited on 12/20/2023).

[7] Muhammad Osama et al. *Stream-K: Work-centric Parallel Decomposition for Dense Matrix-Matrix Multiplication on the GPU*. Jan. 9, 2023. arXiv: 2301.03598[cs]. URL: http://arxiv.org/abs/2301.03598 (visited on 12/21/2023).

[8] Vijay Thakkar et al. *CUTLASS*. Version 3.0.0. original-date: 2017-11-30T00:11:24Z. Jan. 2023. URL: https://github.com/NVIDIA/cutlass (visited on 12/22/2023).

[9] *triton/python/triton/ops/matmul.py at main · openai/triton*. GitHub. URL: https://github.com/openai/triton/blob/main/python/triton/ops/matmul.py (visited on 12/21/2023).