GPT-2 Report

GPU Compute Primitives

Parallel Attentionists

Luke You Minseob Shin Kevin Peercy Luke Kwiatkowski

Introduction

Parallelized operations sound simple on a first pass, and are an obvious staple of the work we do for transformer architecture development and generation. These operations span everything from normalizing items, multiplying forward matrices, and breaking down our inputs into processable components. The tools for parallel work, however, have extreme nuances and opportunity for improvement. This milestone focuses on implementation and analysis of different methods, namely shared memory, reduction, cubLAS, and tensor core integration within our matrix multiplication and data loading. The goals of each strategy focus on two central themes, memory load reduction, and increasing parallel component percentage.

Both these goals are obviously central to speeding up our architecture entirely, but the exact gains we get, and the tradeoffs from each method, is something we have to test and analyze after trying to implement. Our group collectively designed the framework for functional kernels in every version of method sensible for the work done, and this report highlights the gains that were made from said methods. Our hope is to select the best results, and use those implementations, or even combine a couple of tools, to formulate the highest performing transformer going forward.

Baseline Profiling Report

We utilized NVIDIA A40

```
Name:                          NVIDIA A40
Total global memory:           47608692736
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    1740000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     84
```

Figure 1. NVIDIA A40 Resources

CUDA API Summary

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
59.0	110,377,868	115	959,807.5	210,692.0	1,002	25,960,966	2,481,393.7	cudaDeviceSynchronize
34.2	63,935,586	3	21,311,862.0	8,298,968.0	19,827	55,616,791	29,995,952.4	cudaMemcpy
2.7	5,096,709	12	424,725.8	5,816.5	201	1,800,141	616,148.6	cudaFree
1.3	2,472,993	1	2,472,993.0	2,472,993.0	2,472,993	2,472,993	0.0	cudaFreeHost
1.0	1,868,579	1	1,868,579.0	1,868,579.0	1,868,579	1,868,579	0.0	cudaGetDeviceProperties_v2_v12000
0.6	1,214,832	1	1,214,832.0	1,214,832.0	1,214,832	1,214,832	0.0	cudaMallocHost
0.6	1,053,698	171	6,162.0	3,136.0	2,754	336,126	25,740.6	cudaLaunchKernel
0.4	742,995	6	123,832.5	114,834.0	3,457	315,878	117,419.2	cudaMalloc
0.1	190,957	810	235.7	220.0	100	741	97.6	cuGetProcAddress_v2
0.0	42,038	1	42,038.0	42,038.0	42,038	42,038	0.0	cudaMemset
0.0	24,548	18	1,363.8	386.0	310	11,531	2,740.5	cudaEventCreateWithFlags
0.0	8,596	18	477.6	366.0	250	2,084	423.4	cudaEventDestroy
0.0	3,918	3	1,306.0	1,353.0	1,142	1,423	146.3	cuInit
0.0	2,676	3	892.0	311.0	181	2,184	1,120.8	cuModuleGetLoadingMode
0.0	861	2	430.5	430.5	290	571	198.7	cudaGetDriverEntryPoint_v11030

Figure 2. Baseline API Times

CUDA GPU Memory Time Summary

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
86.6	55,529,666	2	27,764,833.0	27,764,833.0	864	55,528,802	39,264,181.5	[CUDA memcpy Host-to-Device]
12.8	8,203,836	1	8,203,836.0	8,203,836.0	8,203,836	8,203,836	0.0	[CUDA memcpy Device-to-Host]
0.6	406,144	1	406,144.0	406,144.0	406,144	406,144	0.0	[CUDA memset]

Figure 3. Baseline Memory Times

CUDA GPU Kernel Summary

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
78.3	86,203,508	49	1,759,255.3	1,610,399.0	428,864	25,956,529	3,565,786.9	matmul_forward_kernel
7.6	8,386,205	12	698,850.4	698,799.5	697,984	700,543	768.7	att_kernel
7.6	8,373,783	12	697,815.3	697,599.0	696,063	700,991	1,602.4	preatt_kernel
4.7	5,131,130	25	205,245.2	205,696.0	190,944	211,776	3,742.3	layernorm_forward_kernel
1.0	1,095,840	12	91,320.0	91,536.0	89,792	92,256	798.1	permute_kernel
0.3	367,935	12	30,661.3	30,656.0	30,399	30,944	144.5	softmax_forward_kernel
0.3	340,736	12	28,394.7	28,352.0	28,224	28,544	88.8	unpermute_kernel
0.1	96,641	24	4,026.7	4,064.5	3,521	4,544	396.6	residual_forward_kernel
0.1	95,489	12	7,957.4	7,936.0	7,808	8,128	111.2	gelu_forward_kernel
0.0	6,688	1	6,688.0	6,688.0	6,688	6,688	0.0	encoder_forward_kernel

Figure 4. Baseline Kernel Times

The most time-consuming kernel in our baseline implementation is the matmul_forward_kernel. This is due to the overhead of the memory operations caused by the bottleneck in the memory access pattern of directly accessing the global memory.

Since the matmul_forward_kernel takes a significant amount of the total time, we will go more in depth to analyze why that is.

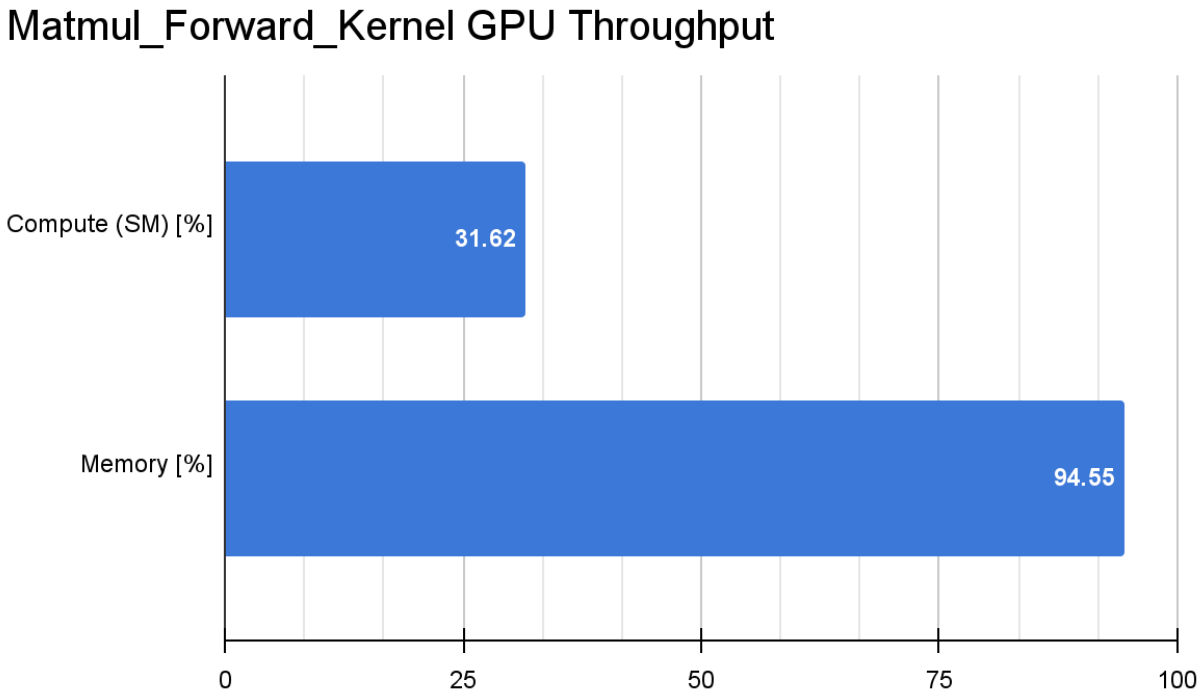


Figure 5. Baseline Matmul Throughput

In the GPU Throughput chart, we could observe that the kernel is utilizing greater than 80.0% of the available memory performance of the device. To further improve the performance, the work will likely need to be shifted from the memory to the compute unit. In other words, there is a bottleneck in the memory. Supporting this, we could see that on average, only 4.0 of 32 bytes transmitted per sector are utilized by each thread for global loads from L1/TEX, only 16.0 of the 32 bytes for global stores to L1/TEX. Hence, we observe that memory utilization is not coalesced.

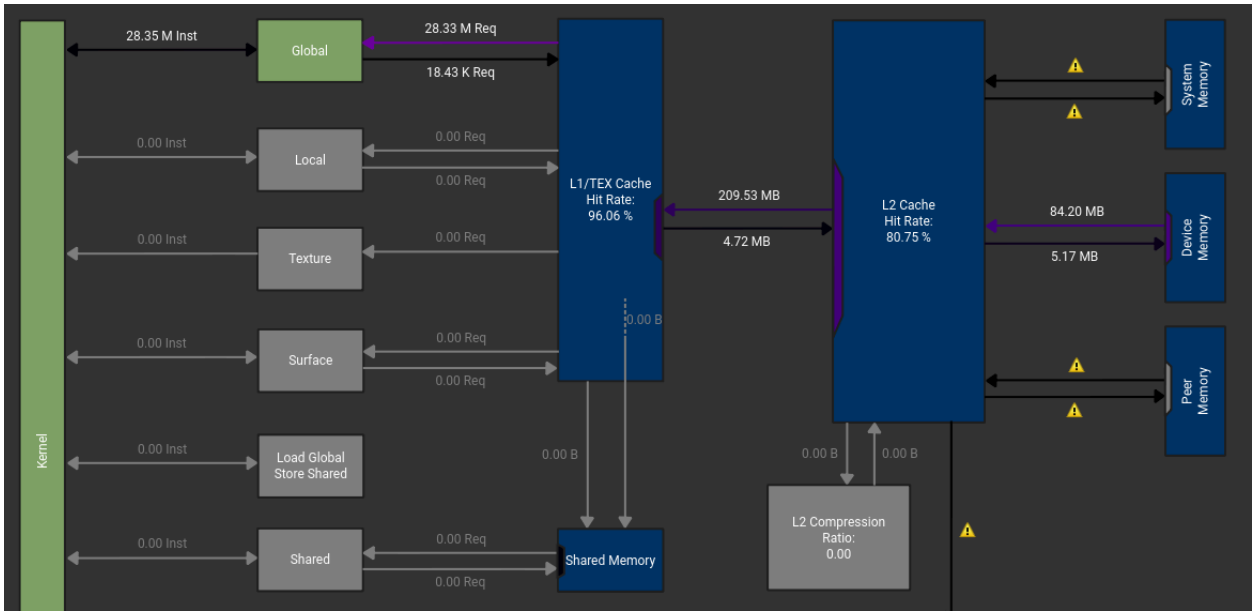


Figure 6. Baseline Memory Chart

From the memory chart, we can also see that we are requesting 28.33M from global memory. This will be important for comparison to our optimizations.

Other kernels such as residual_forward, gelu_forward, and encoder_forward don't take up a significant amount of the total time. However, we will look at how they could be further optimized.

Residual_forward_kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of the device, having a latency issue. This is due to the very low number of eligible warps and the high stall time of active warps. Analyzing the warp state, we were able to identify that on average each warp spends 86.6 cycles (out of 134.7 warp cycles per executed instruction) being stalled waiting for a scoreboard dependency on a L1/TEX operation. Supporting this, L1/TEX hit rate was found to be low of 21.54%. Similarly, gelu_forward and encoder_forward kernels had a problem of non-coalesced memory access, achieving a low cache hit rate of below 30%.

This baseline profiling results suggests that we will have to optimize memory access and leverage data locality by coalescing the memory access, using a joint shared memory, and a thread registers. Once we reduce the memory access overheads, we will focus on optimizing the computational overhead of the kernels.

Joint Shared Memory and Register Tiling for Matrix Multiplication

Luke You

The use of shared memory is a common optimization tactic that is especially relevant in areas of frequent reuse. It is significantly faster than global memory which is often a bottleneck in highly parallelized scenarios, as was the case for our kernels. This is largely attributed to its closer proximity to the compute cores. In fact, it's often implemented using the same hardware as L1 cache in modern GPUs. It however comes with two big caveats: it's limited in size and can only be shared between threads in the same block. In the case of matrix multiplication, this results in the need to split up parts of the input matrices into smaller tiles and load them into shared memory one at a time across multiple iterations as the rows and columns are traversed. This adds overhead but is still faster than utilizing global memory alone due to the high reuse in this type of kernel.

Figure 1. Matrix Multiplication Visualization

To be more specific about the number of reuses, remember that any given element (i, j) in the output uses the corresponding row 'i' in matrix A and column 'j' in matrix B. This also means that every element in the same row of the output matrix uses the same row in matrix A, and every element in the same column of the output matrix uses the same column in matrix B. This means that in a non-tiling algorithm, the number of global memory reuses of any given input element is as large as the width or height of the input matrix. With the use of shared memory, we can reduce that number by a factor of whatever the width or height of an output tile is. This is because all elements in a row or column of the output tile will use the same row or column in shared memory, reducing the number of global memory accesses by the same amount.

Register tiling is another optimization which involves moving chunks of memory to registers which provide near instantaneous access to data. This optimization may seem counterintuitive at first since registers are local to each thread, making it impossible to "share" memory with others to allow reuse. However, it's important to remember that matrix multiplication kernels are often memory-bound, meaning we can actually gain performance if we enable register reuse by making each thread do more work to compute multiple output elements.

We implemented joint shared memory and register tiling for matmul_forward_kernel and various attentioning kernels. For all of them, we utilized a tiling size of 32x32 and a coarsening factor of 8.

Note that in our implementation of joint shared memory and register tiling, we decided to use a 2D thread block across the x and y dimensions instead of a 1D thread block in just the y dimension. We felt that increasing the width of the output tile (and consequently shared memory tile) this way would allow for better reuse of elements from matrix A without increasing the coarsening factor. This meant a single output tile looked something closer to the following figure:

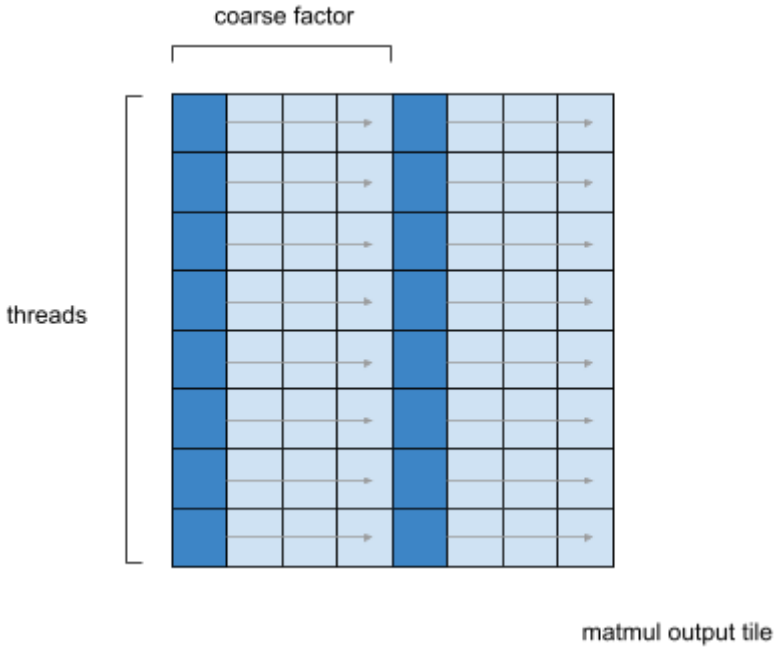


Figure 2. Joint Shared Memory and Register Tile

During the implementation of this optimization, we also made a few changes to the kernel launch configurations which included some that improved parallelism over the baseline. To make it easier to isolate the performance with and without optimization, we decided to include a profiling result that only included the launch configuration changes:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
92.1	86,270,229	49	1,760,616.9	1,610,688.0	429,184	26,005,061	3,572,573.8	matmul_forward_kernel
5.5	5,126,465	25	205,058.6	205,152.0	190,304	209,791	3,772.0	layernorm_forward_kernel
1.6	1,454,432	12	121,202.7	121,216.0	121,152	121,248	31.9	preatt_kernel
0.2	231,744	12	19,312.0	19,312.5	19,136	19,392	67.5	softmax_forward_kernel
0.2	207,906	12	17,325.5	17,296.5	17,184	17,568	117.8	att_kernel
0.1	96,575	12	8,047.9	8,048.0	7,968	8,128	61.7	permute_kernel
0.1	95,805	24	3,991.9	4,031.5	3,392	4,512	438.6	residual_forward_kernel
0.1	94,815	12	7,901.3	7,888.0	7,776	8,032	87.9	gelu_forward_kernel
0.1	60,576	12	5,048.0	5,040.0	5,024	5,088	27.7	unpermute_kernel
0.0	6,657	1	6,657.0	6,657.0	6,657	6,657	0.0	encoder_forward_kernel

Figure 3. Baseline Kernel with Improved Launch Configurations Times

And finally, the results from our fully optimized implementation are shown below:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
79.6	23,879,840	49	487,343.7	442,784.0	140,544	6,437,920	881,207.4	matmul_forward_kernel
17.1	5,135,294	25	205,411.8	205,920.0	190,944	210,368	3,633.3	layernorm_forward_kernel
0.8	231,551	12	19,295.9	19,296.0	19,199	19,424	69.7	softmax_forward_kernel
0.7	201,762	12	16,813.5	16,800.0	16,672	16,992	80.0	att_kernel
0.7	195,840	12	16,320.0	16,320.0	16,256	16,448	49.2	preatt_kernel
0.3	100,769	12	8,397.4	8,416.0	8,288	8,512	71.6	permute_kernel
0.3	100,383	12	8,365.3	8,368.5	8,224	8,543	110.1	gelu_forward_kernel
0.3	98,083	24	4,086.8	4,112.0	3,424	4,705	475.1	residual_forward_kernel
0.2	60,578	12	5,048.2	5,056.0	5,024	5,088	19.7	unpermute_kernel
0.0	6,656	1	6,656.0	6,656.0	6,656	6,656	0.0	encoder_forward_kernel

Figure 4. Joint Shared Memory and Register Tiling Times

Comparing our kernels, we can see that the time the MatMul kernel took went from 86,203,508 to just 23,879,840. This is over a 3x speedup just by smartly utilizing resources. Our attention kernels also saw a significant speedup and now only take up a small portion of the total execution time.

In order to better understand where the performance improvement came from, we took a look through the profiling results generated via Nsight Compute and found the following:

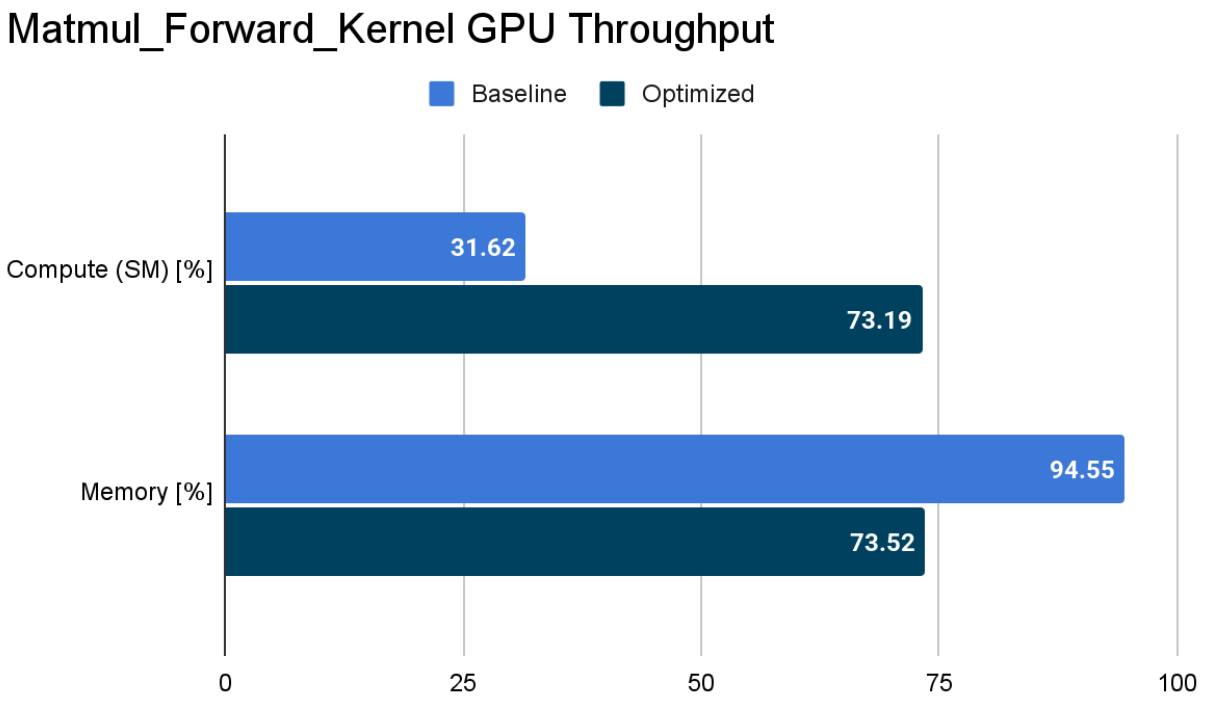


Figure 5. Matmul Throughput Statistics

We can see our compute throughput improved dramatically from 31.62% for the baseline to 73.19%. This is likely thanks to the use of shared memory and registers which allows for much faster memory access when compared to global memory, meaning threads no longer have to wait as long for data retrieval

before continuing with computation. Consequently, memory throughput was also reduced simply because global memory accesses have been reduced thanks to reuse within other types of memory. One thing we found interesting however was that the values are now both about 25-30% away from 100%, suggesting that there is another bottleneck elsewhere in the system, but pinpointing the source is difficult.

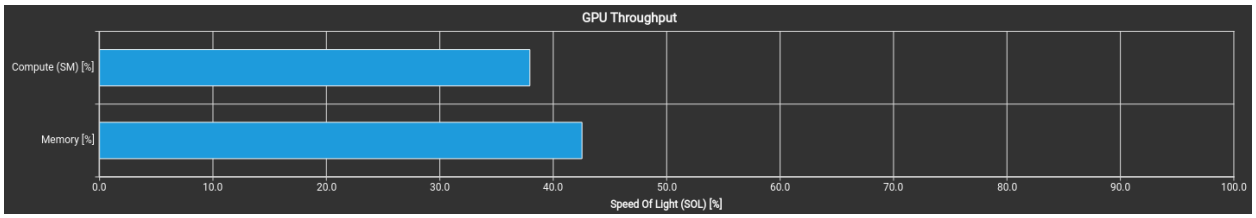


Figure 6. Attention Throughput Statistics

We also see more balanced compute and memory throughput for our attention kernel, suggesting that we are again no longer memory bound. Curiously, we see an even greater deviation from 100% throughput here when compared to the matmul kernel.

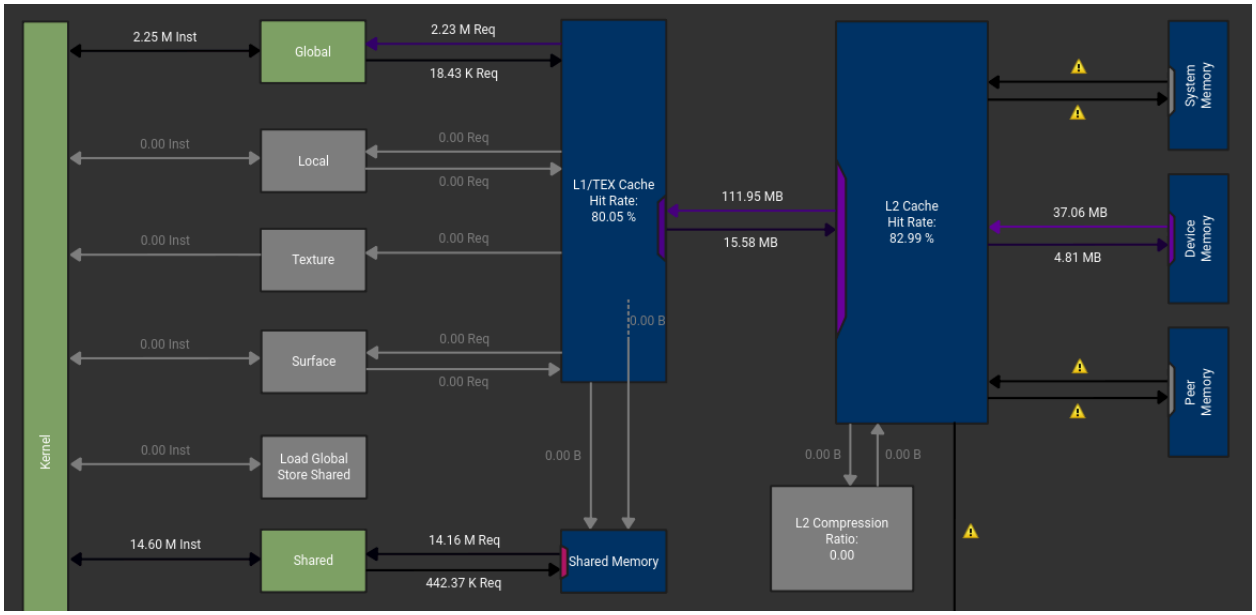


Figure 7. Joint Register and Shared Memory Matrix Multiplication Memory Chart

Looking at the memory chart, we can see that shared memory is now used about 7x as much as global memory. There is also additional reuse from registers, but it is not shown in this chart. Additionally, our kernel only read 2.23M from global memory compared to 28.33M from the baseline implementation. Both these statistics help explain our increased compute throughput and reduced memory throughput. Interestingly, we can see that we had 14.16M reads from and 442.37K writes to shared memory. Dividing these values gives us $14.16/0.44237 \approx 32$ which is exactly the same size as our tile width and height. This supports our earlier calculations for shared memory reuse.

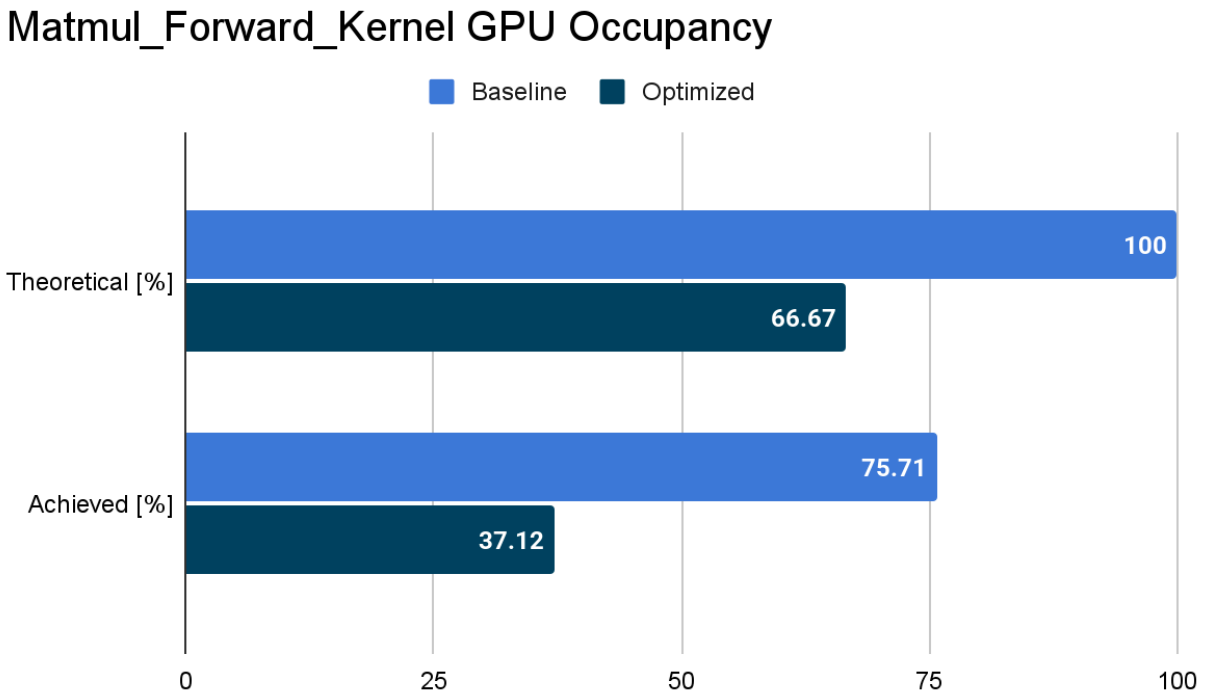
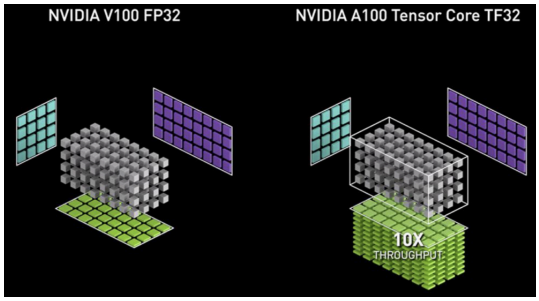


Figure 8. Matmul Occupancy Statistics

Another interesting observation we made was our theoretical and achieved occupancy. Our optimized kernel saw nearly half the achieved occupancy of our baseline kernel. This could stem from a variety of factors. For one, the increased usage of thread synchronization means each warp needs to wait for all other warps in the block to finish before continuing with computation. The increased amount of shared memory and register usage could also put additional strain on the SM's resources, potentially limiting the number of blocks that can run on a single SM. Despite these additional constraints, we thought it was interesting that this optimization still made such a significant improvement to the total execution time. It just goes to show how badly memory throughput can bottleneck a kernel.

Tensor Core for Matrix Multiplication



Kevin Peercy

Tensor cores for matrix multiplication functions as a specialized library access on our end, which abuses the hardware tensor units for the exact function $D = A * B + C$ given 'fragments' which are, for us, the tiles of the matrix. These fragments represent tiles of the input matrices that are distributed across threads in a warp for efficient parallel processing. We implemented Tensor Cores using NVIDIA's WMMA API with TF32 precision, which provides significant computational throughput improvements on the A40 GPU. Our implementation combines the WMMA framework with a tiling strategy that processes matrices in 16×16×8 chunks, corresponding to the WMMA_M, WMMA_N, and WMMA_K dimensions respectively.

We combine the provided framework with our tried and true matrix tiling method of cycling down the rows of Matrix A, and the columns of Matrix B. For this milestone, we did not combine shared tiling with the tensor loads, and instead just used the tensor loading functions: wmma_load and then wmma_sync for the output. We load fragments directly from global memory into tensor core registers using wmma::load_matrix_sync, and perform the calculation with wmma::mma_sync, and store the results back in global memory with wmma::store_matrix_sync. There are clear points of improvement regarding memory access, but it still achieves gains through the specialized hardware and provides the speedups we are looking for.

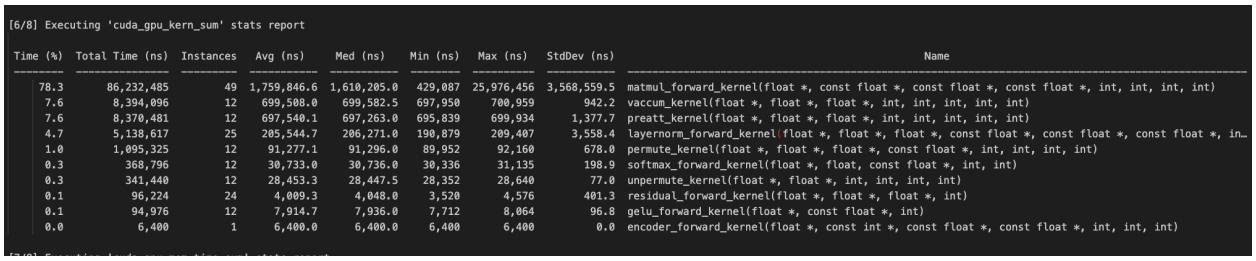
Our tensor core implementation achieves performance gains mainly through two mechanisms. First the A40 GPU provides 74.8 TFLOPs of TF32 tensor core compute throughput, compared to 37.4 TFLOPs for standard FP32 operations, giving us a theoretical 2× speedup in compute capability under the same stress. Secondly, by loading fragments directly into tensor core registers without intermediate shared memory staging, we reduce memory transactions, and the tensor core's optimized data distribution reduces the register use. However, our current implementation has limitations in its memory access pattern. Loading fragments directly from global memory without shared memory caching means we don't benefit from data reuse across multiple warps in the same block, and certain matrix layouts may result in non-coalesced access patterns.

Each warp under tensor core algorithms compute one tile for us, and the tile size is implementation dependent but also subject to the following constraints:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
⋮	⋮	⋮	⋮
__nv_bfloat16	__nv_bfloat16	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8

Figure 1. Supported Tensor Matrix Sizes

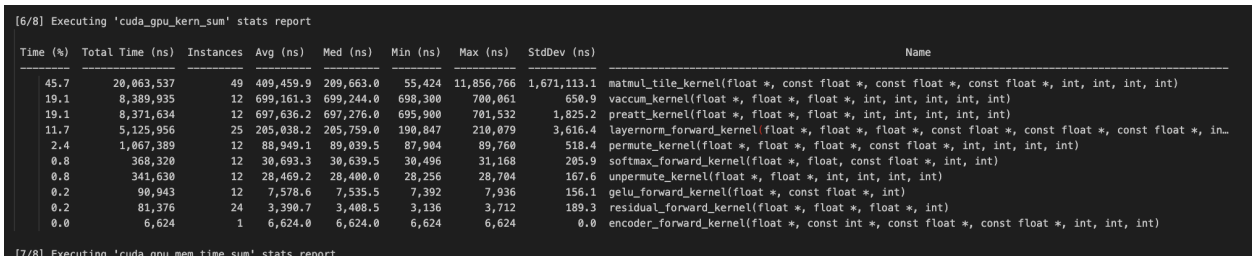
We chose to use the tf32 inputs and float accumulator, which we also pushed to its max tile size of 16x16 outputs, which are simply mapped to one block for us. Our grid launches used 1 thread warp in a block per tile to make indexing simpler. Below we will be analyzing numerical results from our profiling, and draw conclusions about optimal strategy and whether we implemented the tensor cores well.



[6/8] Executing 'cuda_gpu_kern_sum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
78.3	86,232,485	49	1,759,846.6	1,610,205.0	429,087	25,976,456	3,568,559.5	matmul_forward_kernel(float *, const float *, const float *, const float *, int, int, int, int)
7.6	8,394,096	12	699,508.0	699,582.5	697,950	700,959	942.2	vaccum_kernel(float *, float *, float *, int, int, int, int, int)
7.6	8,370,481	12	697,540.1	697,263.0	695,839	699,934	1,377.7	preatt_kernel(float *, float *, float *, int, int, int, int, int)
4.7	5,138,617	25	205,544.7	206,271.0	190,879	209,407	3,558.4	layernorm_forward_kernel(float *, float *, float *, const float *, const float *, const float *, in…
1.0	1,095,325	12	91,277.1	91,296.0	89,952	92,160	678.0	permute_kernel(float *, float *, float *, const float *, int, int, int, int)
0.3	368,796	12	30,733.0	30,736.0	30,336	31,135	198.9	softmax_forward_kernel(float *, float, const float *, int, int)
0.3	341,440	12	28,453.3	28,447.5	28,352	28,640	77.0	unpermute_kernel(float *, float *, int, int, int, int)
0.1	96,224	24	4,009.3	4,048.0	3,520	4,576	401.3	residual_forward_kernel(float *, float *, float *, int)
0.1	94,976	12	7,914.7	7,936.0	7,712	8,064	96.8	gelu_forward_kernel(float *, const float *, int)
0.0	6,400	1	6,400.0	6,400.0	6,400	6,400	0.0	encoder_forward_kernel(float *, const int *, const float *, const float *, int, int, int)

Figure 2. Baseline Kernel Times



[6/8] Executing 'cuda_gpu_kern_sum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
45.7	20,063,537	49	409,459.9	209,663.0	55,424	11,856,766	1,671,113.1	matmul_tile_kernel(float *, const float *, const float *, const float *, int, int, int, int)
19.1	8,389,935	12	699,161.3	699,244.0	698,300	700,061	650.9	vaccum_kernel(float *, float *, float *, int, int, int, int, int)
19.1	8,371,634	12	697,636.2	697,276.0	695,900	701,532	1,825.2	preatt_kernel(float *, float *, float *, int, int, int, int, int)
11.7	5,125,956	25	205,038.2	205,759.0	190,847	210,079	3,616.4	layernorm_forward_kernel(float *, float *, float *, const float *, const float *, const float *, in…
2.4	1,067,389	12	88,949.1	89,039.5	87,904	89,760	518.4	permute_kernel(float *, float *, float *, const float *, int, int, int, int)
0.8	368,320	12	30,693.3	30,639.5	30,496	31,168	205.9	softmax_forward_kernel(float *, float, const float *, int, int)
0.8	341,630	12	28,469.2	28,400.0	28,256	28,704	167.6	unpermute_kernel(float *, float *, int, int, int, int)
0.2	90,943	12	7,578.6	7,535.5	7,392	7,936	156.1	gelu_forward_kernel(float *, const float *, int)
0.2	81,376	24	3,390.7	3,408.5	3,136	3,712	189.3	residual_forward_kernel(float *, float *, float *, int)
0.0	6,624	1	6,624.0	6,624.0	6,624	6,624	0.0	encoder_forward_kernel(float *, const int *, const float *, const float *, int, int, int)

Figure 3. Tensor Core Matrix Multiplication Kernel Times

Above are outputs from pre-tensor installation top, and post-tensor implementation for just matmul_forward bottom. The improvement from 86 million -> 20 million nanoseconds is a speedup of over 400%, and clearly shows the extent of tensor's optimal hardware utilization for compute capability.

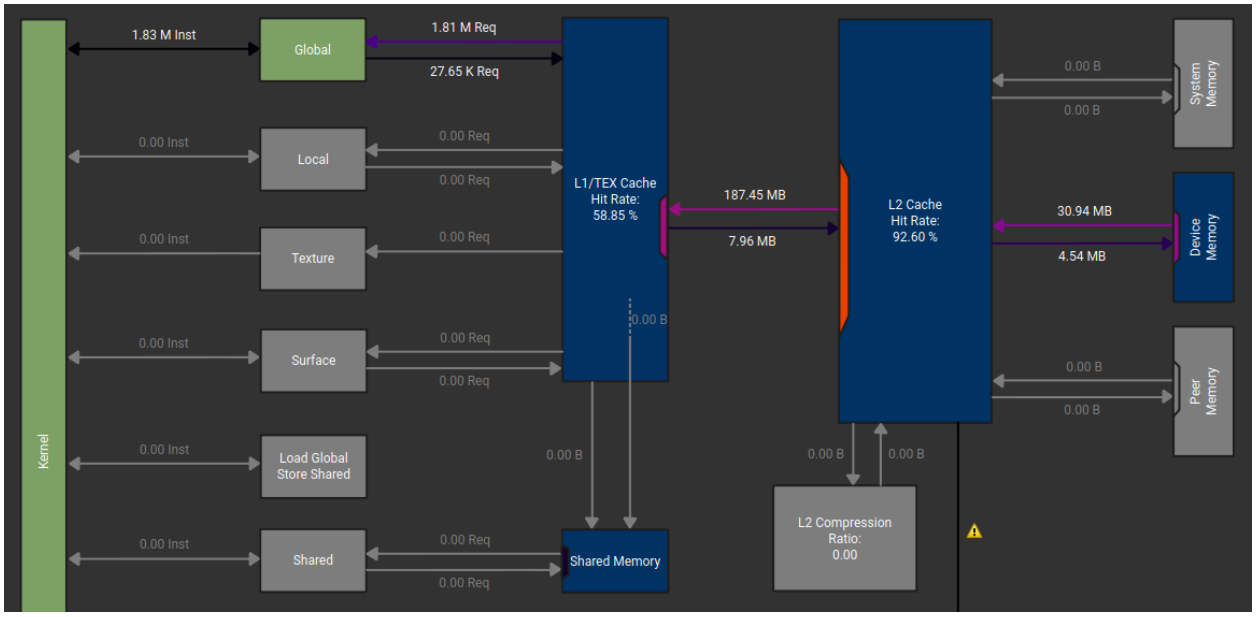


Figure 4. Tensor Core Matrix Multiplication Memory Chart

Above is the memory stream chart, which gives insight into the complete absolution of shared memory, and also a fairly weak L1 cache hit rate, with our blocks only covering a small fragment of our output. Our implementation of tensor cores for this milestone had no external focus on the memory optimizing aspect of it, and that is clearly reflected with our data chart showing no shared memory use, and just a straight pipeline from Kernel to Global.

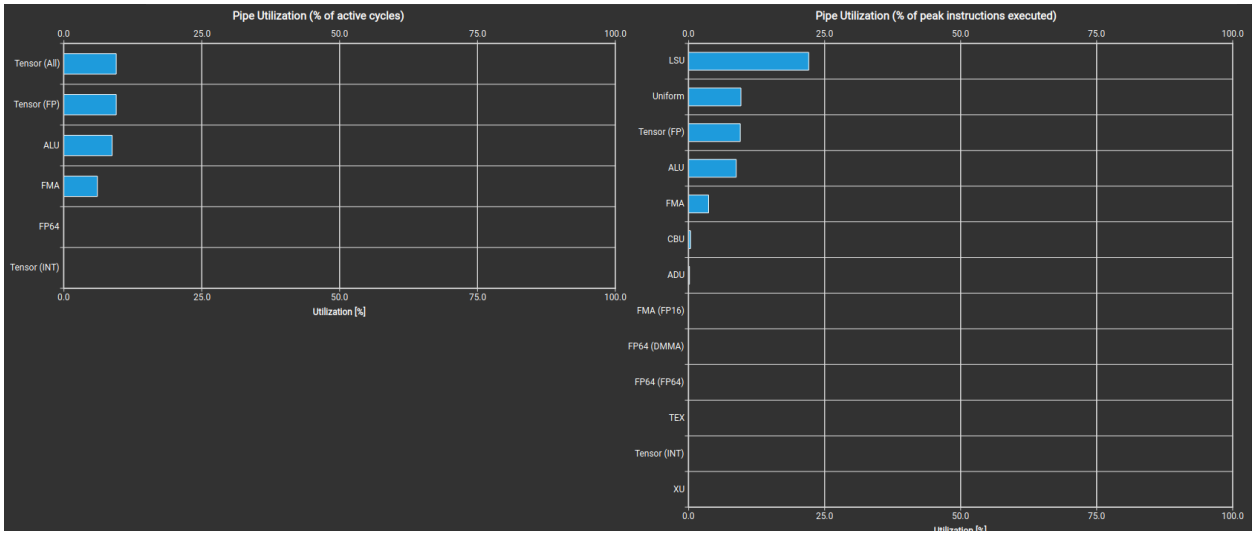


Figure 5. Pipe Utilization Reference

The above utilization chart was massively insightful into the drawbacks of our tensor algorithm. We can see ALU operations formatting a solid amount of our work, when that should be just the bias additions compared to the massive matrix multiplications happening, tile by tile. The ratios of matrix operations to just the bias additions should be nowhere near this close and show a drawback of tensor loading as we perform it. The other stark problem is the number of load store instructions executed, as we do our bias addition again with loading the data from our global mem, and storing the offset figure back into it. Memory should be offloaded better than we see here, and the tensor cores do not provide the pipelines for clearing those setbacks.

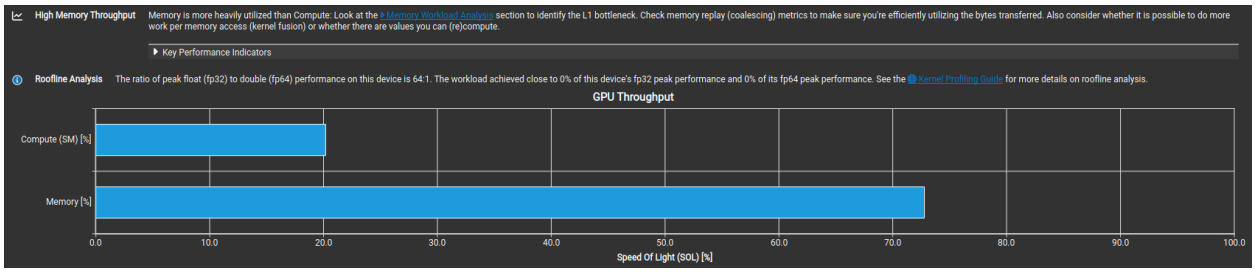


Figure 6. Speed of Light Reference

Breakdown of findings:
The switch to tensor core hardware in our matrix multiplications was an obvious benefactor for performance, as the size of the grid we had to launch was drastically reduced, and along with that was quicker algorithms and more parallelized functionality to the discretion of the core library. That was the obvious and intended result of our work. On the other side, we lost the memory optimizations that other implementation methods may have focused more on, and that is still the biggest factor for performance as we are still memory bound. The memory charts all displayed a lackluster ratio of compute to memory utilization, as we were continuously bottlenecked by the loading and offloading of our data and how the registers for tensor cores work, not even mentioning the poor bias contribution to our formulas.

If we want to truly make use of the tensor cores, we will need to work around our loading and bias methodology, as at the moment, we are only tapping into the hardware gains while another system utilizes, for example, cuBLAS; which already has us beat with a level of insight into optimal memory functions paired with tensor cores.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
0.6	138,433	12	11536.1	11552.0	11360	11776	115.3	att_kernel
0.6	126,048	12	10504.0	10496.0	10400	10688	82.0	preatt_kernel

Figure 7. Tensor Core Attention Times

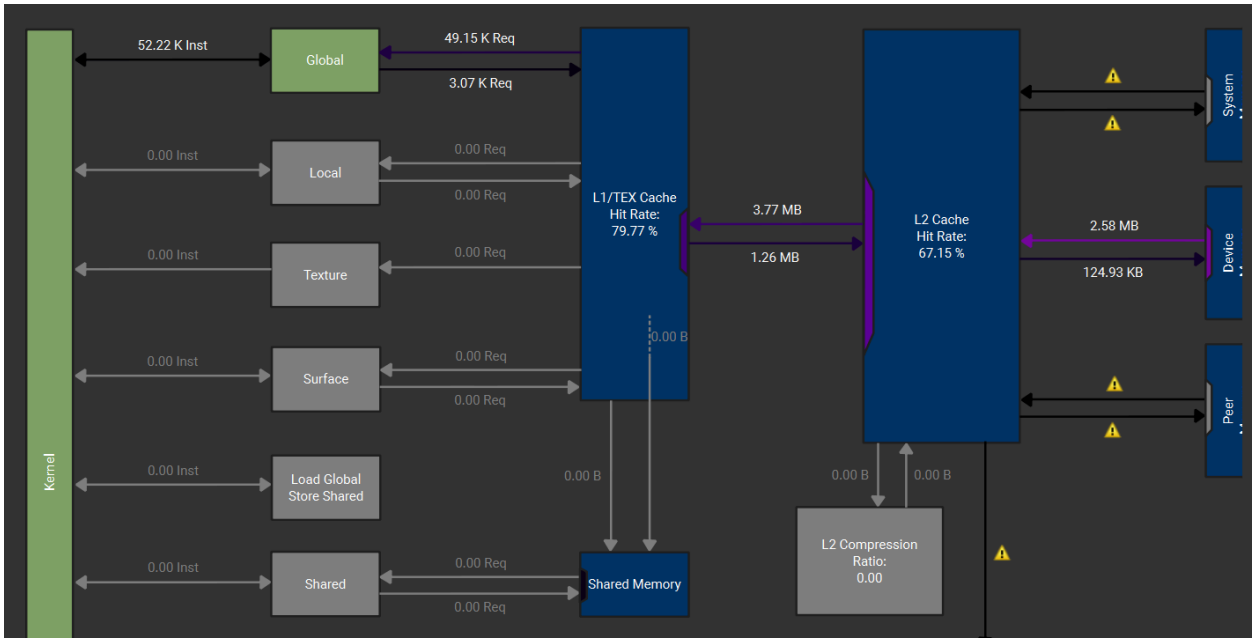


Figure 8. Tensor Core Attention Memory Chart

Similarly to the matmult kernel this uses a very basic implementation that uses no shared memory and has a relatively low L1 cache hit rate.

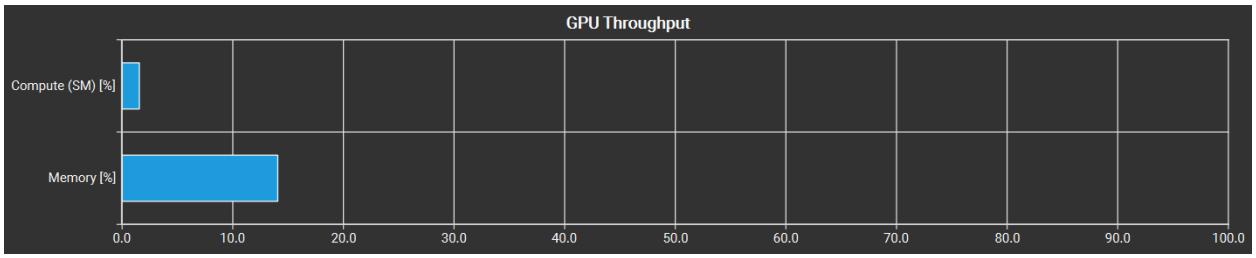


Figure 9. Attention Speed of Light Baseline

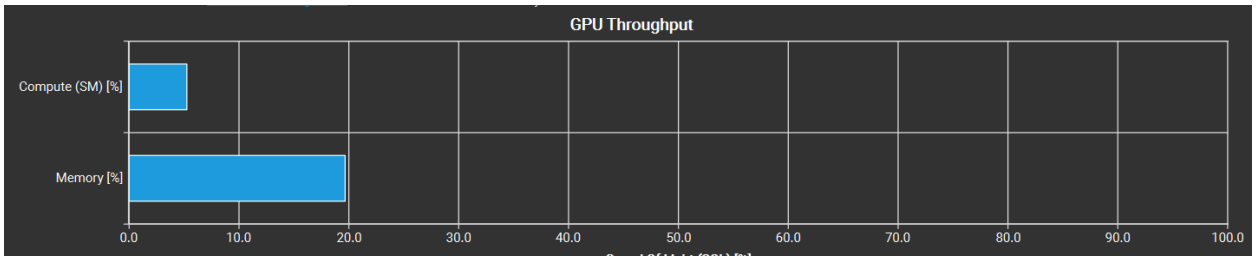


Figure 10. Tensor Core Attention Speed of Light

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	
53.1	27,024,122	49	551,512.7	500,385.0	155,584	7,260,615	994,514.5	matmul_forward_kernel(float *, const
16.5	8,388,102	12	699,008.5	698,752.5	698,208	700,192	680.7	vaccum_kernel(float *, float *, floa
16.4	8,371,594	12	697,632.8	697,073.0	696,000	701,505	1,715.3	preatt_kernel(float *, float *, floa
10.1	5,122,726	25	204,909.0	204,832.0	191,489	209,216	3,532.8	layernorm_forward_kernel(float *, fl
2.1	1,080,772	12	90,064.3	90,096.5	89,120	91,424	648.5	permute_kernel(float *, float *, flo
0.7	368,769	12	30,730.8	30,688.0	30,560	31,072	153.4	softmax_forward_kernel(float *, floa
0.7	342,113	12	28,509.4	28,496.0	28,320	28,704	136.8	unpermute_kernel(float *, float *, i
0.2	103,040	12	8,586.7	8,576.0	8,384	8,832	129.7	gelu_forward_kernel(float *, const f
0.2	98,528	24	4,105.3	4,176.0	3,520	4,736	473.0	residual_forward_kernel(float *, flo
0.0	6,625	1	6,625.0	6,625.0	6,625	6,625	0.0	encoder_forward_kernel(float *, cons

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	
76.1	27,024,119	49	551,512.6	500,801.0	155,776	7,254,086	993,583.6	matmul_forward_kernel(float *, cons
14.4	5,125,858	25	205,034.3	205,440.0	191,456	209,248	3,567.8	layernorm_forward_kernel(float *, f
4.2	1,484,899	12	123,741.6	123,712.0	123,104	124,416	380.0	softmax_forward_kernel(float *, flo
3.0	1,081,537	12	90,128.1	90,256.5	88,832	90,976	658.6	permute_kernel(float *, float *, fl
1.0	341,761	12	28,480.1	28,480.0	28,320	28,609	109.3	unpermute_kernel(float *, float *,
0.4	133,025	12	11,085.4	11,088.0	10,913	11,264	114.4	vaccum_kernel_tensor_core(float *,
0.4	126,658	12	10,554.8	10,528.5	10,464	10,689	77.1	preatt_kernel(float *, float *, flo
0.3	103,583	12	8,631.9	8,576.0	8,512	8,960	122.2	gelu_forward_kernel(float *, const
0.3	98,241	24	4,093.4	4,144.0	3,424	4,736	521.9	residual_forward_kernel(float *, fl
0.0	6,656	1	6,656.0	6,656.0	6,656	6,656	0.0	encoder_forward_kernel(float *, con

Figures 11 & 12. Baseline and Tensor core attention times

This above 60x speedup in these 2 types of matrix multiplication is likely due to the headsize being 64 which fits very well into the tensor core tile size (16x16x8) as well as the general speedup from tensor cores' more efficient multiplication. But this couldn't explain it so easily so we looked at the instruction statistics and it became clearer.

Metric	Current
LDG	786432.00
FFMA	393216.00
MOV	80384.00
IADD3	80064.00
UIADD3	73728.00
BRA	67680.00
ISETP	62656.00
IMAD	56544.00
UMOV	12288.00
PLOP3	12288.00
ULDC	6240.00
STG	6144.00
S2R	1536.00
EXIT	352.00
LOP3	96.00

Metric	Current
IMAD	96960.00
IADD3	67776.00
LD	49152.00
LEA	31296.00
HMMA	24576.00
SHF	17664.00
ISETP	13248.00
BRA	11328.00
LOP3	8640.00
S2R	7488.00
WARPSYNC	6912.00
ULDC	6912.00
CS2R	6336.00
ST	3072.00
MOV	2496.00
IMNMX	768.00
CALL	768.00
BSYNC	768.00
BSSY	768.00
EXIT	384.00
S2UR	192.00

Figures 13 & 14 Baseline and Tensor Core instruction counts in attention

The tensor core implementation executes around 4x less instructions overall and further replaces expensive FFMA operations with HMMA ones. The most common instructions in the tensor core implementation are using integers further explaining the speedup.

`cuBLAS Utilization

Parallel Attentionists

NVIDIA cuBLAS is a GPU-accelerated library for accelerating AI and HPC applications. It contains highly optimized Basic Linear Algebra APIs that are simple to use for drop-in hardware acceleration.

We will compare our current progress against NVIDIA cuBLAS to illustrate how much speedup we were able to achieve so far in respect to state-of-the-art NVIDIA GPU linear algebra kernels. To start, we utilized cuBLAS for our matrix multiplication. Note that matrix multiplication may be the most important algorithm for the transformer architectures as they spend most of their FLOPs inside the matrix multiplication kernels both during training and inference.
For the matrix multiplication forward, we perform

$$Input * Weight^T + Bias = Output$$

where $Input(B, T, C)$, $Weight(OC, C)$, $Bias(OC)$, and $Output(B, T, OC)$

We utilized cublas<t>gemmStridedBatched() to efficiently do this. The cuBLAS library uses column-major storage, and 1-based indexing. Since C and C++ use row-major storage, we have to think about the parameters that we are passing into the function call.

$Input * Weight^T = Output$ is the expression that we want the cuBLAS library to receive. In order to achieve this, we pass in $(Weight^T)^T * Input^T = Output^T$. The cuBLAS library will interpret this expression in column-major order and compute the desired output. Note that to convert from row-major to column-major order, you essentially need to transpose the matrix.

Let's take a look at the profiling result from the matrix multiplication utilizing the cuBLAS library.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
36.8	1,860,326	36	51,675.7	50,303.0	26,880	78,143	20,953.3	void cutlass::Kernel2<cutlass_80_tensorop_s1688gemm_64x64_16x6_tn_align1>(T1::Params)
26.7	1,349,611	1	1,349,611.0	1,349,611.0	1,349,611	1,349,611	0.0	void cutlass::Kernel2<cutlass_80_tensorop_s1688gemm_256x64_16x4_tn_align1>(T1::Params)
13.7	694,900	12	57,908.3	58,223.0	55,711	60,350	1,277.1	void cutlass::Kernel2<cutlass_80_tensorop_s1688gemm_64x64_16x6_tn_align4>(T1::Params)
7.1	361,338	48	7,527.9	6,943.5	4,833	12,160	2,609.2	matmul_forward_kernel

Figure 1. cuBLAS Utilized Matrix Multiplication Kernel Profile

Note that the cuBLAS utilized a parameterizable kernel with different dimensionality for matrix multiplication. Total runtime adds up to 4,266,175 ns. This demonstrates NVIDIA's state-of-the-art matrix multiplication optimization and how much time was put into composing the cuBLAS library. The cuBLAS library was able to achieve 5.59x speedup compared to our Joint Shared Memory and Register Tiling optimized kernel, 4.70x speedup compared to our Tensor Core Matrix Multiplication, and 20.21x speedup compared to our baseline implementation.

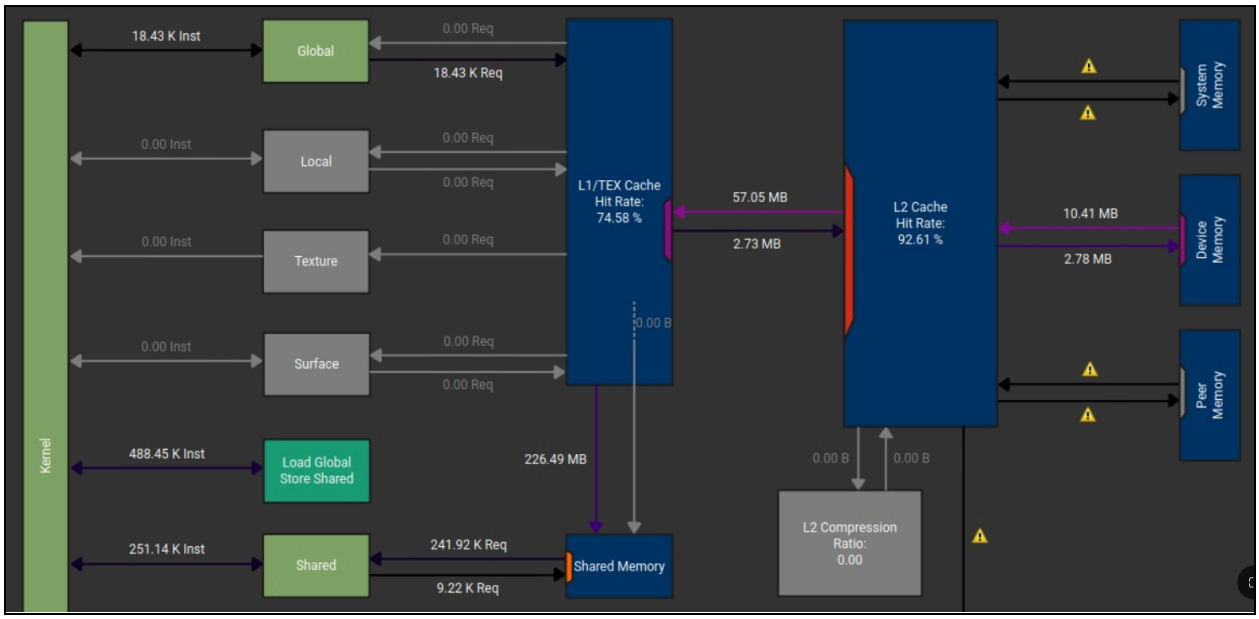


Figure 2. Matrix Multiplication cuBLAS Memory Chart

In Nsight Compute, we observed an incredible amount of data reuse that the cuBLAS utilized matrix multiplication was able to achieve. Notice the data transfer from L1/TEX Cache to Shared Memory. Most of the data were fetched from L1 Cache to Shared Memory, resulting in immense speedup. We believe this was done by hyperoptimizing the memory access patterns for the maximum data reuse along with the advanced optimizations.

Compared to our joint register and shared memory tiling implementation of the matrix multiplication, shown in Figure, the cuBLAS issues significantly less memory instructions from the kernel, resulting in 22.23x less memory access in total. Similarly, the cuBLAS issues less memory instructions than the tensor core matrix multiplication kernel, as shown in Figure, with 2.41x less memory access in total. This demonstrates the effectiveness of the data reuse and memory access patterns of the cuBLAS library and provides an upper limit on how much more we could optimize our kernels.

The attention algorithm also heavily relies on matrix multiplication. So we utilized the cuBLAS library for our attention kernels as well.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
3.0	150,942	12	12,578.5	12,576.0	12,512	12,703	62.8	ampere_sgemm_128x128_tn
3.0	149,500	12	12,458.3	12,416.0	12,319	12,864	156.4	ampere_sgemm_128x128_nn

Figure 3. cuBLAS Utilized Attention Kernel Profile

This resulted in around 1.31x, 0.91x speedup compared to joint shared memory and register tiling and tensor core utilization respectively. This result is surprising as our tensor core implementation for attention algorithm actually outperformed the cuBLAS library. We believe this is due the fact that we were able to fine-tune the parameters for the tensor core utilization.

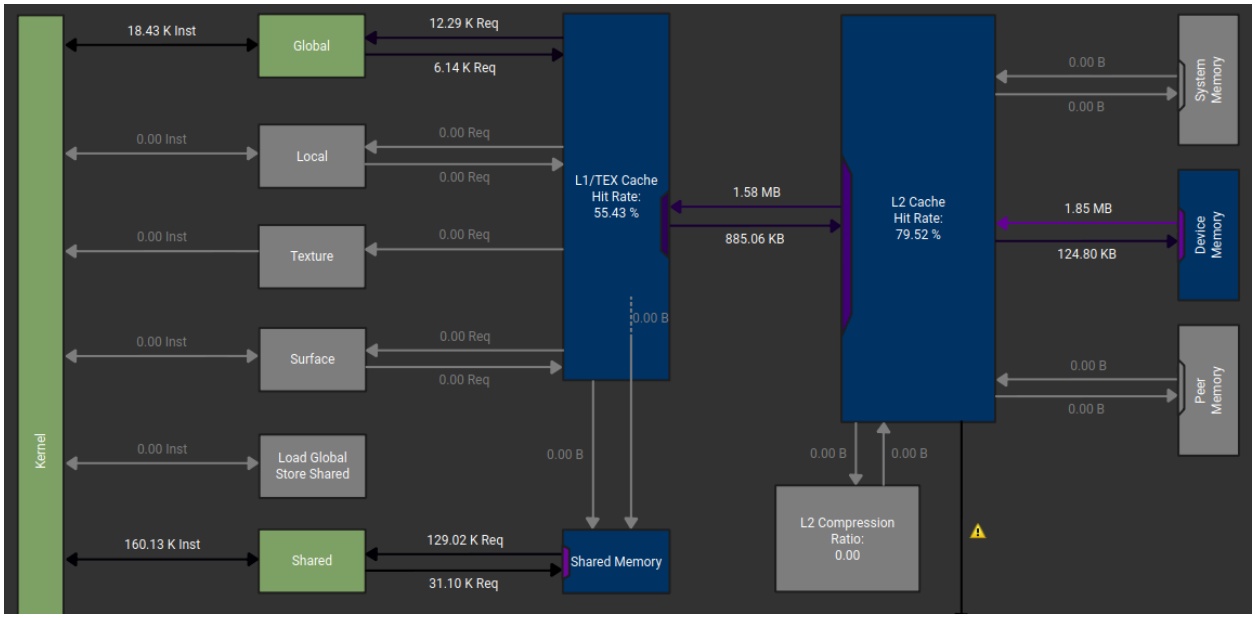


Figure 4. cuBLAS Utilized Attention Kernel Memory Chart

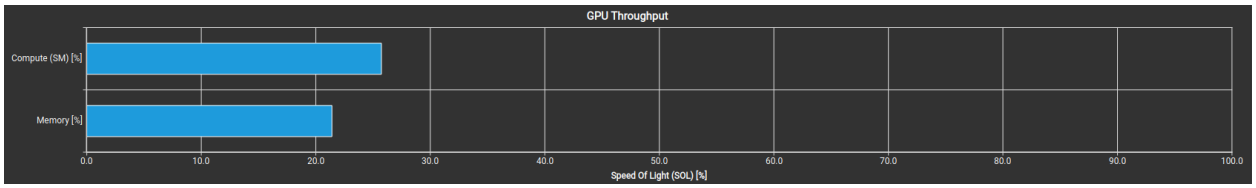


Figure 5. cuBLAS Utilized Attention Kernel Speed of Light

Here we could see that cuBLAS actually becomes compute-bound whereas the tensor core implementations of the attention kernel are perfectly balanced, as shown in Figure 9.

Parallel Reduction

Minseob Shin

The reduction operator is a type of operator that is commonly used in parallel programming to reduce the elements of an array into a single result and we will use this optimization technique to speed up a couple of our kernels. Observing our kernels, we were able to identify that layernorm_forward_kernel and softmax_forward_kernel compute a sum of values. Computing a sum is one of the classic optimizations achieved by reduction operation, as it reduces a set of input values to one value (in our case, this will be the sum). Note that this is possible because computing the sum is commutative and, importantly, associative.

There are two reasons for the reduction optimization

1) The parallel reduction operator reduces the steptime from O(N) to O(log N) by organizing the computation as a binary tree structure rather than a sequential scan.
2) The parallel reduction algorithm is work-efficient, meaning that it requires the same amount of work as a sequential algorithm.

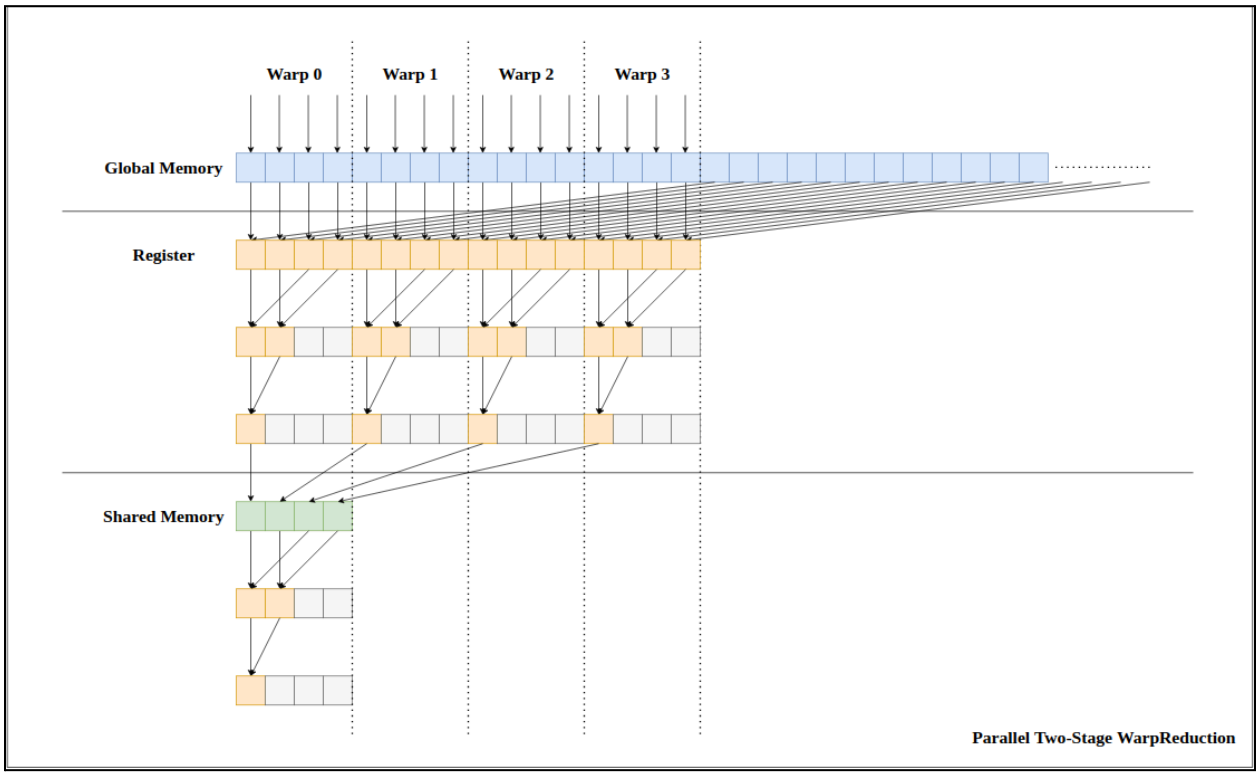


Figure 1. Two-Stage Warp Reduction Algorithm

For N input values, the number of operations is

$$\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \ldots + 1 = N - 1$$

Let's take a look at our baseline implementation for both layernorm_forward and softmax_forward kernel.

Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
5,133,523	25	205,340.9	205,472.0	190,752	214,880	3,975.6	layernorm_forward_kernel
1,483,196	12	123,599.7	123,456.0	123,071	124,576	469.1	softmax_forward_kernel

Figure 2. Baseline Implementation Profile

Our baseline implementation follows naive implementation where each thread individually computes the sum of the input columns. Layernorm_forward_kernel lets each thread loop through the input column values three times. Each for computing the mean, variance, and reciprocal standard deviation. Similarly, softmax_forward_kernel consists of three major loops that only compute through their own position. This is due to the decoder-only architecture of GPT-2. This results in a total runtime of 5,133,523 ns for layernorm_forward_kernel and 1,483,196 ns for softmax_forward_kernel.

```
float sum = 0.0f;
for (int c = 0; c < C; ++c) {
    sum += inp[b * T * C + t * C +
c];
}
```

Figure 3. Naive Summation

Our optimized implementation follows Figure 1. Our implementation of parallel reduction is the two-stage warp[1] reduction. It first loads the sum of two (or more) input values from global memory to registers. Once values are loaded to registers, each warp performs warp level reduction using warp shuffle functions[2] to broadcast the intermediate values from one thread to another. All warps store their computed value to shared memory and synchronization. Finally, one warp performs warp reduction to acquire the single result. This implementation of parallel reduction is highly efficient because it utilizes registers and removes unnecessary overhead of thread block synchronization. Note that fully computed intermediate values needed for the next set of computation were loaded into shared memory by thread 0 to broadcast them across the thread within the block. Also, we chose to implement thread coarsening[3] along with parallel reduction due to the possibility of non-coarsened thread blocks being serialized by the hardware, as shown in Figure 5. After applying the parallel reduction optimization, we were able to obtain 36.5x speed up in layernorm_forward_kernel and 11.4x speed up in softmax_forward_kernel.

Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
140,578	12	11,714.8	11,712.0	11,648	11,872	64.6	softmax_forward_kernel
129,055	25	5,162.2	5,023.0	4,672	5,888	438.2	layernorm_forward_kernel

Figure 4. Optimized Implementation Profile

[1] A warp is a group of 32 threads that are scheduled and executed together as a single unit on an NVIDIA GPU
[2] A warp shuffle function in CUDA (and GPU programming) is a special operation that allows threads within the same *warp* to directly exchange register values with each other
[3] Thread coarsening is a CUDA optimization technique where each thread computes multiple outputs rather than a single output

```
for (int tile = tx; tile <= own_pos; tile +=
BLOCK_DIM) {
    maxval = fmaxf(maxval, inp[base + tile]);
}
```

Figure 5. Parallel Reduction Max Computation

In Nsight Compute, we observed the number of FLOPs (Floating-point operations) to verify that parallel reduction is indeed work-efficient. Let's look at baseline and optimized profiles for instance.

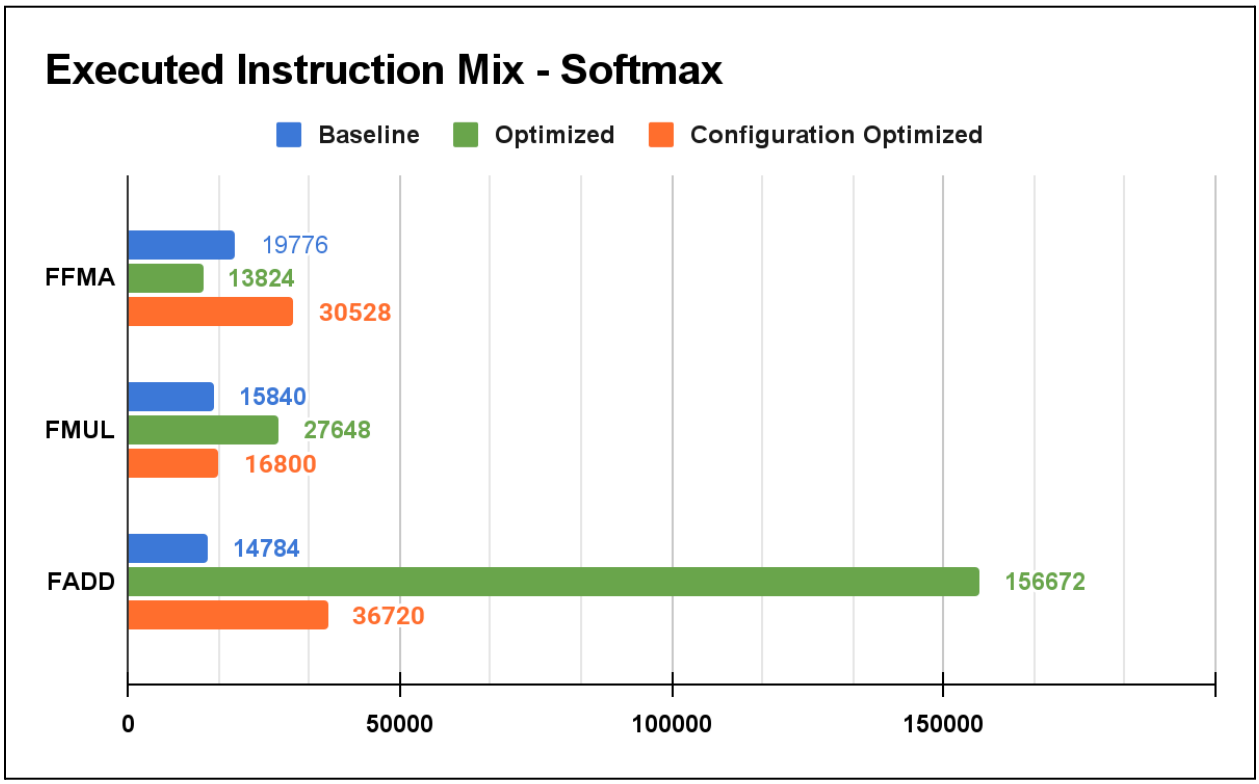


Figure 6. Softmax Floating-Point Instruction Statistics

We could observe that the FADD significantly increased from 24,197 to 156,672, increasing over 6x times. We believed that this increase was not reasonable, and concluded that the reduction was not as parallel as we wanted it to be due to the large number of inputs and limited device resources. In order to compensate for this, we changed the kernel launch configuration. For the configuration optimization, we changed the BLOCK_DIM from 256 to 32 to increase the coarse factor by 8 and removed the overhead from two-stage warp reductions. This results in a total number of FADD of 36,720, which is a reasonable increase from the baseline implementation as parallel reduction uses slightly more FADD to reduce the computed values from each block to the final value.

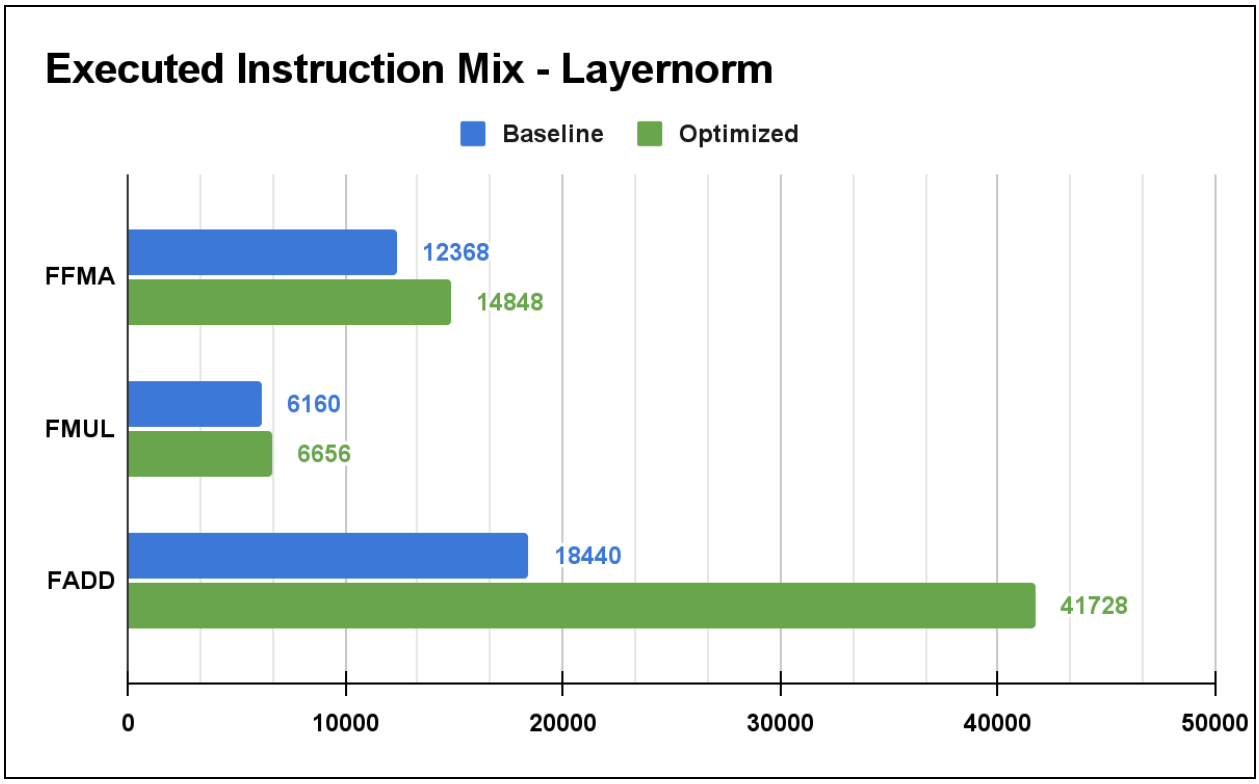


Figure 7. Layernorm Floating-Point Instruction Statistics

For the same reason, we could observe a similar increase in FADD for layernorm_forward_kernel, as it increases from 18,440 to 41,728. Note that we didn't include Configuration Optimized statistics because our optimized configuration turned out to be the best configuration for layernorm_forward_kernel. To conclude, this demonstrates that the parallel reduction algorithm is work-efficient and can result in a significant amount of speed up for specific use cases.

Optimization Proposal

From our optimization, we still noticed that our kernels are not optimal. To further improve our kernels, we propose the following optimizations to reduce the computation and memory redundancy.

To begin, we propose utilizing a KV cache. A KV cache is a memory structure used in transformer models, particularly for large language models (LLMs), to store the key and value tensors from self-attention layers. Utilizing a KV cache will significantly reduce the time needed for self-attention.

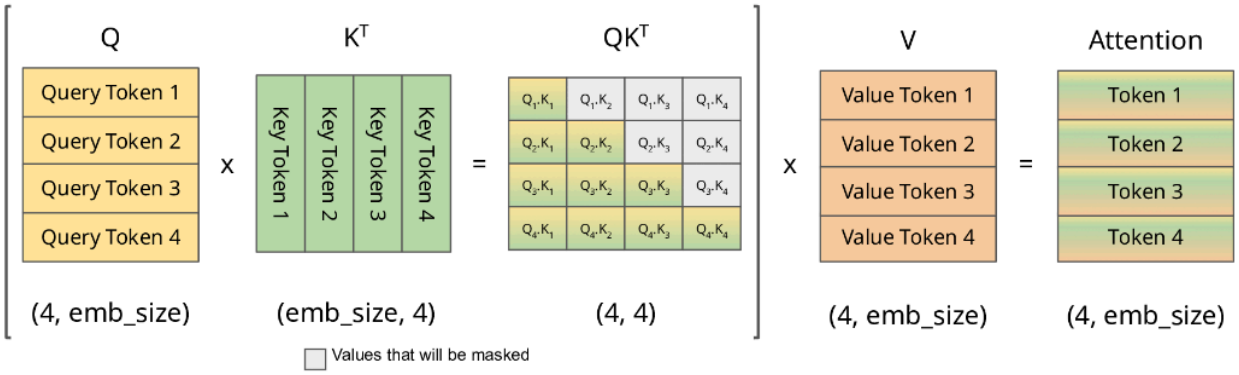


Figure 1. Attention algorithm for GPT-2

A KV Cache allows us to focus on only calculating the attention for the new token because a KV cache caches previous keys and values. This is a significantly less computation needed for attention. We will use the device's global memory for the caching purpose, and even though this increases our memory latency, we believe that reducing the computation redundancy is more important for the performance. This is because of the fact that we have to recompute key and value matrices every step. This results in $O(n^2)$ computation work. However, with a KV cache, we eliminate the need to recalculate the key and value matrices and reduce the computation work of $O(n)$. Given the parameter size, we believe this will result in significant speedup.

For the better customization of our kernels, we plan to utilize NVIDIA CUTLASS.

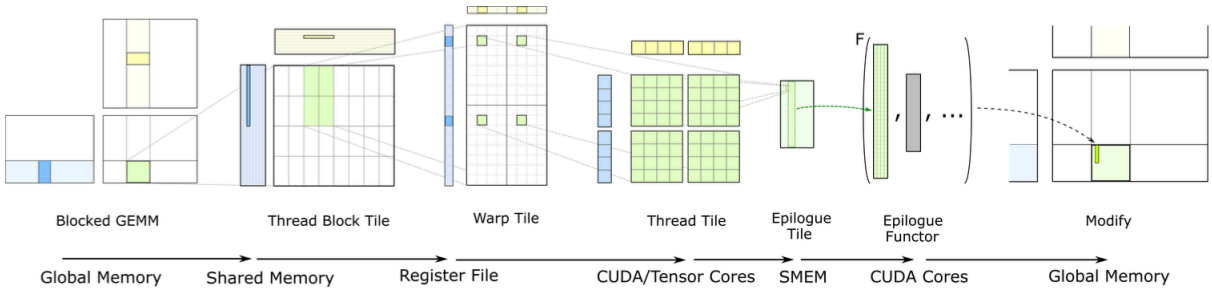


Figure 2. The Hierarchical GEMM Computation Embodied by CUTLASS

CUTLASS decomposes GEMM into modular components at thread, warp, and threadblock levels, allowing us to specialize tiling sizes, data types, and algorithmic policies at each level. This flexibility enables fine-grained performance tuning across the parallelization hierarchy to maximize throughput for our specific problem sizes. Our main use for CUTLASS will be similar to cuBLAS utilization, and will be mainly utilized for matrix multiplication and attention. Also CULASS uses software pipelining and split-k parallelism and we believe this will be a good way to reinforce the learning on these concepts, while learning how to utilize open-source API for possible future projects.

Let's change our perspective to the broader system.

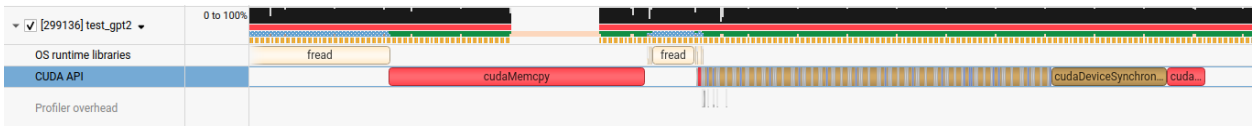


Figure 3. Nsight Systems Timeline

Observing the timeline of our system in Nsight Systems, we could notice that memory transfer to the kernel takes an immense amount of time and happens at once, meaning that computation doesn't happen after all the data is transferred to the kernels. This is inefficient because single computation for threads often doesn't require the kernel to have the entire data; threads only need the specific data for their part of computation. Our solution to this problem is to pipeline (asynchronous data transfer using CUDA Streams) the computation. We plan to pipeline the data transfer and computation to overlap the CPU and GPU execution. In order to achieve this, we will closely observe the data amount needed for the single pass of the GPT-2 execution and modify gpt2.cuh to pipeline the system without breaking it.

Another thing we see is the overhead of the cuda device synchronization and extraneous stores/loads. We can assist in this problem through kernel fusion. With our GPT2 test, the functionality through our kernels for the forward pass have a repetitive pattern shown in the function calls below.

```
// now do the forward pass
layernorm_forward(l_ln1, l_ln1_mean, l_ln1_rstd, residual, l_ln1w, l_ln1b, B, T, C);
matmul_forward(scratch, l_ln1, l_qkvw, l_qkvb, B, T, C, 3*C);
attention_forward(l_atty, l_qkvr, l_att, scratch, B, T, C, NH);
matmul_forward(l_attproj, l_atty, l_attprojw, l_attprojb, B, T, C, C);
residual_forward(l_residual2, residual, l_attproj, B*T*C);
layernorm_forward(l_ln2, l_ln2_mean, l_ln2_rstd, l_residual2, l_ln2w, l_ln2b, B, T, C);
matmul_forward(l_fch, l_ln2, l_fcw, l_fcb, B, T, C, 4*C);
gelu_forward(l_fch_gelu, l_fch, B*T*4*C);
matmul_forward(l_fcproj, l_fch_gelu, l_fcprojw, l_fcprojb, B, T, 4*C, C);
residual_forward(l_residual3, l_residual2, l_fcproj, B*T*C);
```

Figure 4. GPT-2 Forward Pass Kernel Calls

Layernorm_forward is followed both times with Matmul_forward, with the output from layernorm_forward serving as the input token embeddings for matmul_forward immediately following. We also have the combination of matmul_forward followed by residual_forward as can be seen in the final 2 calls, and also following our attention call. The most extreme implementation of this idea would be

layernorm through residual, with a swappable attention or gelu call in the middle. This would add in complexity with grid sizings, and with shared memory and such, so we do not propose this formally as a goal. The layernorm + matmul can be achieved, and lets our output double as the load for our input into matrix multiplication, and a similar sentiment can be applied on the matmul into residual sequence.