Faster symmetric matrix multiplication with ThunderKittens

Laker Newhouse
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA, USA
lakern@mit.edu

Dakota Goldberg
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
dakotag@mit.edu

Ricardo Ruiz
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
ruizr@mit.edu

***Abstract*—The Muon optimizer shows promise for speeding up training in machine learning by setting all gradient singular values to 1, using iteration called Newton-Schulz. But on an NVIDIA H100, PyTorch only captures 60% of this iteration's possible throughput. Using the ThunderKittens framework, we design a new kernel for the symmetric matrix multiplication that appears in Newton-Schulz. Our kernel achieves 1510 TFLOPS compared to PyTorch's 774 TFLOPS on 8192x8192 symmetric matrices, translating to an overall speedup in Newton-Schulz of up to 30%. We discuss how one could distribute the kernel, then conclude with thoughts on ThunderKittens framework.**

Code available at https://github.com/Arongil/thunder

1. Introduction

Training machine learning models takes a lot of compute. The most popular optimizer people use for training is Adam, or its variant AdamW, but there is a new optimizer called Muon that may have better sample efficiency. For GPT-2 scale transformer language models, Muon trains to the same loss as a tuned AdamW using 30% less data [1]. Muon is built on ideas developed by Bernstein et al. [2, 3]. The main difference from Adam is that the gradient is sent through the map

$$G \mapsto UV^\top, \tag{1}$$

where $G = U\Sigma V^\top$ is the singular value decomposition. Rather than compute an SVD, one can apply this map efficiently on a GPU by iterating G under certain polynomials, known as a Newton-Schulz iteration.

The purpose of this paper is to present a kernel that speeds up the Newton-Schulz iteration. The idea is to replace the general matrix multiplication $XX^\top$ with a special symmetric matrix multiplication kernel, which we call *symmul*. Because the result of $XX^\top$ is symmetric, symmul computes over only the lower triangular half of the output matrix and copies the results to the upper triangular half.

In Section 2, we describe the Newton-Schulz iteration to show why it is useful to build a specialized symmul kernel. We additionally derive the most FLOP-efficient polynomial to use. In Section 3, we present the symmul kernel. The section begins with a primer on the ThunderKittens framework, then describes the choices we made to design the kernel and gives benchmarks. In Section 4, we describe next steps for our kernel, including initial thoughts on how to distribute over sharded matrices in clusters of GPUs. In Section 5, we conclude with some thoughts on the ThunderKittens framework.

2. What is Newton-Schulz?

Newton-Schulz is an algorithm that iteratively pushes the singular values of a matrix closer to 1 [4]. It does so by applying an odd polynomial, such as $p(x) = \frac{3}{2}x - \frac{1}{2}x^3$. Odd polynomials act directly on the singular values:

$$f(G) = Uf(\Sigma)V^\top$$

for $G = U\Sigma V^\top$. Therefore Newton-Schulz lets us to apply any function $f : \mathbb{R} \to \mathbb{R}$ to the singular values of a matrix, as long as we can approximate f with compositions of odd polynomials. Equation 1 acts on the singular values via the map $x \mapsto 1$. One polynomial that approximates this function when iterated is $p(x) = \frac{3}{2}x - \frac{1}{2}x^3$, but we can choose other polynomials, too.

As a precursor to benchmarking Newton-Schulz using our kernel compared to using PyTorch, this section calculates the FLOPs that Newton-Schulz requires for different polynomial degrees. We also derive the optimal polynomial choice. As far as we know, our FLOP ratio analysis is novel.

For concreteness, suppose X is $n \times m$, where $n < m$. If $m > n$, we can transpose X. Our goal is to minimize the

FLOPs we need to lift a singular value of 0.001 to at least 0.9, while limiting to send singular values close to 1. We set this goal because achieving a 900x inflation factor for small singular values approximates the map from 1. And we allow some $\epsilon > 0$ in the range $[1-\epsilon, 1+\epsilon]$ that singular values are sent to in the limit, typically $\epsilon = 0.3$.

When iterating a polynomial on an input $x \ll 1$, the linear coefficient controls how quickly x increases. To go from 0.001 to 0.9, if the coefficients were $1.5x - 0.5x^3$ it would take 18 iterations; if the coefficients were $2x - x^3$ it would take only 11. If we use fifth degree polynomials, we can cause a 1000x inflation with just 5 iterations using $4x - 7x^3 + 3.3x^5$. But every power we use means more matrix multiplications. We will now consider the right balance.

Let $A = (XX^\top)$. To apply a degree $2k+1$ odd polynomial to X, we need to compute $(c_0 + c_1 A + \cdots + c_k A^k)X$. For $k \geq 1$, the FLOPs required at each stage are as follows:

1) Computing A requires $\frac{1}{2}(2n^2m)$ FLOPs, saving 50% thanks to symmetry.
2) Computing A^l for $1 < l \leq k$ can be done sequentially for $\frac{1}{2}(2n^3)$ FLOPs each, thanks to symmetry.
3) Adding up $c_0 + c_1 A + \cdots + c_k A^k$ requires $O(n^2)$ FLOPs, which we neglect.
4) Multiplying $(c_0 + c_1 A + \cdots + c_k A^k)X$ requires $2n^2m$ FLOPs.

Therefore the total FLOPs to apply one iteration of a polynomial of degree $2k+1$ in Newton-Schulz is

$$3n^2m + (k-1)n^3. \tag{2}$$

Let us consider the case of $n = 4096$ and $m = 16384$, which are viable dimensions for the MLP in a commercial transformer model. The TFLOPs required to apply a polynomial of degree $2k+1$ are

$$0.847 + 0.0685(k-1).$$

For reference, recall that an H100 can reasonably attain 760 TFLOPS for general matrix multiplication (GEMM) of bf16 values, out of a theoretical maximum of 989 TFLOPS.

We still have a choice over coefficients. Because the linear coefficient controls growth near zero, we list below the growth rate per TFLOP. We use the highest linear coefficient possible before the polynomial diverges no matter our choice of higher order terms:

1) Degree 3: best polynomial is $2.5x - 2.3148x^3$, for growth factor 2.5/(0.847 TFLOPs) = 2.95 / TFLOP.
2) Degree 5: best polynomial is $4x - 7x^3 + 3.3x^5$, for growth factor 4/(0.916 TFLOPs) = 4.37 / TFLOP.
3) Degree 7: best polynomial $5x - 14.6x^3 + 16.5x^5 - 5.86x^7$, for growth factor 5/(0.984 TFLOPs) = 5.08 / TFLOP.

But this growth rate is not quite right. Iterating a polynomial twice will add the TFLOPs but multiply the growth from the linear coefficient. To make a single metric that is preserved under composition, we take the log base 2 of the linear coefficient to make it additive under iteration. Then the *log growth rate ratio* for degrees 3, 5, and 7 are 1.561 / TFLOP, 2.127 / TFLOP, and 2.345 / TFLOP.

The analysis indicates that the most effective polynomial degree to use is 7. For this degree, the symmetric matrix multiplication saves 0.42 TFLOPs, turning what would have been 1.40 TFLOPs into 0.98 TFLOPs. Therefore symmul has the potential for a 30% speedup.

3. ThunderKittens implementation

This section outlines the important design principles of ThunderKittens (Section 3.1), describes our implementation of symmul (Section 3.2), and benchmarks the new kernel (Section 3.3).

3.1. ThunderKittens: A Guide for the Perplexed

ThunderKittens is a DSL built on top of CUDA intended for programming tensor kernels on NVIDIA's most powerful GPUs. ThunderKittens provides a simpler level of abstraction than raw CUDA while achieving comparable performance on important kernels like matrix multiply and forward attention. We chose to use ThunderKittens to explore modern GPU programming techniques beyond CUDA and to facilitate development of our kernels. This section explores the major differences that set ThunderKittens apart from CUDA.

ThunderKittens is fundamentally designed for large tensor workloads, and this focus is made apparent in its compute model. In ThunderKittens, the fundamental unit of computation is a 256-element (16 by 16) square matrix tile. As a fundamental unit, tiles replace scalar registers and compose shared and global memory. This choice bakes matrix tiling into ThunderKittens, making matrix kernels easier to read. It also aligns better with the GPU hardware as tensor cores operate on tiles instead of scalar registers.

ThunderKittens exposes a framework for producer-consumer workflows, called prototypes. A prototype is defined by the different stages of computation, either load-compute-store-finish (lcsf) or load-compute-finish (lcf); our project uses the lcf prototype. Prototypes use a warp-specialized producer-consumer model: producer warps load some data into shared memory, which is then computed on by consumer warps. This process iterates until the workload is compete. Coordination between warps is handled by driver kernels using semaphores, which signal when inputs to a phase are ready for processing. The driver kernel sets up shared memory, maintains intermediate state, and shares semaphores. This behavior is abstracted away from the programmer, who is not meant to edit the driver kernel directly.

ThunderKitten prototypes are programmed using template metaprogramming. The user defines two structures: a layout and a kernel template. The layout defines the structure of the kernel's operand matrices and of its shared memory. The kernel template provides functions to define the behavior of producer and consumer warps. For producers, the template defines a load function to write into shared memory. Consumers require definitions of compute and finish functions. Both require their own setup functions, as well as one global setup function common to both warp types. The setup functions run once per loop iteration and specify control flow for the kernel; this allows load, compute, and finish functions to be agnostic of iteration index. The setup functions receive mutable iteration state variables from the driver kernel, allowing them to set iteration bounds and setup addresses according to iteration index.

3.2. Symmul in ThunderKittens

This section describes how we wrote our symmul kernel. Suppose A is $n \times m$. Like for the example implementation of GEMM in ThunderKittens, our symmul kernel uses 128x256 blocks with a base tile size of 64x64. All values are in bf16 except the accumulator state, which is float32.

What sets symmul apart from GEMM is that it computes over only half the blocks. For any matrix multiplication, we begin by assigning to each SM a static list of output blocks that it is responsible for computing. In GEMM, we assign SMs in an order that allows for coalesced memory accesses. In symmul, coalescing loads becomes trickier because of the jagged diagonal. Nonetheless, we use the same coalesced pattern, which we hypothesize is helpful below the diagonal.

Within the load-compute-store framework, the place we assign block coordinates $(\mathsf{row}, \mathsf{col})$ to an SM is in common_setup. If $\mathsf{row} < \mathsf{col}$, we designate the block as upper triangular. Rather than shut off the SM entirely, we iterate the SM forward to whatever is the next block in its static assignment of blocks.

Under this scheme, it is important that all SMs are assigned a similar amount of blocks in the lower triangular part of the matrix. Otherwise some SMs will rapidly run out of work, idling. One solution—which we use—is to initialize one SM per block, taking advantage of the GPU's scheduler to ensure any SM with upper triangular coordinates stops and is repurposed to another block. A more sophisticated method could be to map the linear task_id directly to lower triangular blocks. One could devise an ordering that yields coalesced loads for blocks below the diagonal. We do not explore this second approach.

When symmul is launched, it first assigns block coordinates to each SM. It also assigns the number of compute iterations. Since we accumulate over increments of size 64 according to the base tile, the number of tensor core instructions we need—and hence the number of compute iterations—is $m/64$.

Next, producer and consumer warps work in parallel. Producer warps load in rows of A and columns of B to set up the outer product. Consumer warps feed these outer products into the tensor cores to accumulate results.

When a consumer warps finishes, it stores its result twice because of symmetry. First it stores its output in the lower triangular coordinate normally. Second, it transposes the output and then writes it to the transposed coordinate. Implementing this transposed write required hunting down many devils in the details, including creating temporary register tiles and shared memory tiles.

In reflection, ThunderKittens primitives mean the final kernel is short. But the primitives are built out of a long chain of templates and types, triggering many times that we needed to trace through the source code to figure out how to make simple changes. Once a kernel is written and correct, however, that same structure means ThunderKittens automatically gets lots of little things right, from memory access patterns to keeping tensor cores fed.

3.3. Benchmarking in PyTorch

Calling our kernel from PyTorch is easy:

```
import symmul
import pytorch

dev = "cuda"
bf32 = torch.bfloat32
size = (4096, 4096)

A = torch.ones(size, device=dev, dtype=bf32)
B = A.T.contiguous()
C = torch.zeros(size, device=dev, dtype=bf32)

symmul.symmul(A, B, C)
```

We compare to two baselines: PyTorch's GEMM and cuBLAS's symmetric rank k update command, syrk. This second kernel is meant to implement an XX^T operation. We test on the problem size 8192 by 8192.

TABLE 1. PERFORMANCE COMPARISON OF MATRIX MULTIPLICATION KERNELS ON VARIOUS INPUT SIZES IN TFLOPS

Kernel (TFLOPS)	**1024x1024**	**4096x4096**	**8192x8192**
PyTorch's GEMM	126	743	774
cuBLAS's SYRK	41	N/A	N/A
ThunderKittens' GEMM	165	789	775
ThunderKittens' SYMMUL	143	1041	1510

To evaluate our custom kernels, ensure correctness, and measure performance against existing implementations, we wrote a simple PyTorch benchmarking framework. We use PyBind11 to expose our symmul and matmul ThunderKittens kernels, which lets us invoke them as PyTorch functions in local testing and benchmarking scripts.

Running our symmetric multiply kernel alongside the base implementations highlights the speedups our kernel attains. On a 8192×8192 input size, our implementation consistently runs at 1510 TFLOPS—more than the max throughput of an H100, which is 989 TFLOPs, because we are maintaining the denominator of how many FLOPs would be required for GEMM. Comparatively, the optimized ThunderKittens GEMM kernel reaches 775 TFLOPS for the same input size.

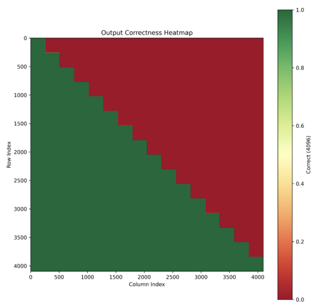

Figure 1. Current correctness results for symmul

4. Next Steps

4.1. Ensuring correctness on arbitrary inputs

At time of writing, our kernel works correctly when the input matrix A is all ones. Our kernel has incorrect behavior in the transpose-and-store step. Though this step seems simple, ThunderKittens hides a lot of complexity that we must contend with at this stage. For one, our matmul kernel uses blocks of size 128×256. Two consumer warpgroups do all the computation, each handling a "wide tile" of size 64×256. Each of four warps is responsible for holding a 16×256 sliver of this wide tile in its registers. Coordinating each of these warps collaboratively is easy—as long as the warps each hold rows. But when we transpose, the rows become columns. Communicating these stores into global memory becomes a tricky task, because ThunderKittens swizzles matrix layouts under the hood. Perhaps there is a way to communicate that the layout should swap to column format. Then the transpose-and-store step would become simple and elegant in code. We have confirmed the rest of the code runs correctly. Namely, we are able to compute results on only the lower triangular half of the output (Figure 1).

4.2. Distributing Newton-Schulz

It is often in the interest of modern machine learning applications to train large models (i.e., later-generation GPT-scale models), which tend to have weight matrices too large to fit into the memory of a single GPU. This challenge requires one to design a training setup that distributes the model's weights across multiple GPUs, which are ideally communicating over a high-speed channel, while maximally maintaining parallel computation and salvaging throughput.

A popular approach to this problem is tensor-parallel sharding, which divides the model's layers across multiple GPUs then combines the partial results computed by each machine in parallel [5]. This approach is efficient for operations that can be split across GPUs for simultaneous computation, such as basic matrix multiplication, which can be divided into independent subprocess blocks.

Newton-Schulz orthogonalization, however, requires iterative updates with global dependencies, making sharding significantly more complex. Finding effective ways to distribute this operation is essential to efficiently training large models with the Muon optimizer. Therefore, an important next step toward achieving accelerated training of large models with the Muon optimizer is writing custom Newton-Schulz distributed kernels using NVIDIA Collective Communications Library (NCCL) primitives.

We initially planned to develop versions of our symmetric multiply kernels that support tensor-parallel training and test the extent to which we could maintain throughput when running `modded-nanogpt` [6] with our kernel bindings. To prepare for this, we spent some time experimenting with NCCL and the `modded-nanogpt` pipeline. Finding custom kernel development with ThunderKittens to be more involved than we expected, we ultimately decided to focus our energy toward optimizng the symmul kernel and leave the distributed component as an opportunity for future work.

There is no native support for NCCL in ThunderKittens prototypes. Given ThunderKittens' stated goal of facilitating kernel development, we see this as a significant area for potential future work. We see two potential entry points for NCCL instructions: kernel templates and prototype drivers. Using NCCL in kernel templates would give the programmer more direct control of inter-device communication. Given that the prototype model abstracts away similar memory management and scheduling, there is a case for abstracting inter-device communication to prototype drivers. If we had more time, we could have experimented with a high-level extension to ThunderKittens prototype drivers with NCCL primitives.

5. Conclusion

We have shown that it is possible to make a faster symmetric multiplication kernel, which can further accelerate the Muon optimizer and ultimately speed up training in machine learning.

There are more optimization opportunities we have not explored. Our work focused on leveraging symmetry to avoid unnecessary computation, but we have yet to examine ways to exploit the symmetry of X and $X^\top$ when loading data into shared memory. There are likely tricks that would make a symmetric matrix multiplication kernel even faster.

Working with ThunderKittens brought both great challenge and great reward. Because the framework is so new, its documentation is sparse. We had to look through the entire codebase, going line-by-line through core files, to understand how everything works. Once we began to overcome the steep learning curve, we appreciated the ease of writing kernels. We have yet to fully explore the framework's capabilities or understand its nuances.

Acknowledgments

Thank you to the teaching staff in 6.S894 for unusually pedagogical labs and your enthusiasm for the subject! We are also grateful to Ben Spector for helping us learn the ropes at ThunderKittens early in the project.

References

[1] Keller Jordan et al. *Muon: An optimizer for hidden layers in neural networks*. 2024. URL: https://kellerjordan.github.io/posts/muon/.

[2] Jeremy Bernstein and Laker Newhouse. “Old Optimizer, New Norm: An Anthology”. In: *Workshop on Optimization for Machine Learning*. 2024.

[3] Jeremy Bernstein and Laker Newhouse. *Modular Duality in Deep Learning*. 2024. URL: https://arxiv.org/abs/2410.21265.

[4] Åke Björck and C. Bowie. “An Iterative Algorithm for Computing the Best Estimate of an Orthogonal Matrix”. In: *SIAM Journal on Numerical Analysis* (1971).

[5] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2020. arXiv: 1909.08053 [cs.CL]. URL: https://arxiv.org/abs/1909.08053.

[6] Jordan Keller. *Modded NanoGPT*. Accessed: 2024-12-10. 2024. URL: https://github.com/KellerJordan/modded-nanogpt.