2.8 < 3..? Implementing the Strassen Algorithm on GPU

Elijah Baraw (ebaraw), Tanay Bennur (tbennur)

April 2025

1 Summary

We implemented a variant of the Strassen Matrix Multiplication Algorithm on GPU, which computes $C = A \cdot B$ in $O(n^{2.807})$ instead of the $O(n^3)$ achieved by standard matrix multiplication algorithms. We performed extensive memory optimizations and created a custom kernel to improve our implementation. We tested our implementation against cuBLAS, the proprietary NVIDIA linear algebra library, and outperformed it on square matrices with dimension ≥ 4096, with a maximum speedup of 1.24x on matrices with dimension 16384. We tested our implementation on RTX 2080 and V100 GPUs, and found similar speedups for both device types.

2 Background

2.1 Elementary Matrix Multiplication

In the elementary matrix multiplication of $C = A \cdot B$, each output element of C is computed by taking the dot product a row of A and a column of B. This has a cost of $O(N^3)$.

If we divide each of our matrices into four sub-matrices, each of dimension $\frac{N}{2} \times \frac{N}{2}$ as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

We find that we can compute the result matrix, C, as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix}$$

This blocked computation is equivalent to the elementary matrix-multiplication computation and involves 8 submatrix multiplications of size $\frac{N}{2}$. It has total cost $8 \cdot (\frac{N}{2})^3 = N^3$.

Elementary matrix-multiplication is highly parallelizable on GPUs, demonstrating high data parallelism and low divergence, and existing multiplication kernels achieve high memory throughput and use almost all of the GPU's available compute. However, for large matrices, the GPU's compute and memory-throughput capacity are quickly exceeded due to the aforementioned cubic cost. The only way to overcome this is with a fundamentally more efficient algorithm.

2.2 The Strassen Algorithm

The key observation of the Strassen algorithm is that we can compute intermediate terms:

$$\begin{aligned}
M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22});\\
M_2 &= (A_{21} + A_{22}) \times B_{11};\\
M_3 &= A_{11} \times (B_{12} - B_{22});\\
M_4 &= A_{22} \times (B_{21} - B_{11});\\
M_5 &= (A_{11} + A_{12}) \times B_{22};\\
M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12});\\
M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}),
\end{aligned}$$

And use these intermediate terms to compute C:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

Notice that we do one multiplication for each of seven intermediate terms, reducing the number of submatrix multiplications from 8 to 7. We introduce more matrix-additions, but adding these matrices has cost $O(N^2)$, which is less influential asymptotically. With this reduction of one matrix multiplication, total cost becomes $7 \cdot (\frac{N}{2})^3 + O(n^2) = \frac{7}{8}N^3 + O(n^2)$.

This alone doesn't provide an asymptotic improvement. However, if we call the Strassen algorithm recursively, then the total work of multiplying two $N \times N$ matrices becomes $N^{\lg 7} \approx N^{2.807}$, which is an asymptotic improvement over the elementary matrix-multiplication algorithm [3].

2.3 Strassen Variants

For our project, we wanted to find a modern variant of the Strassen-Algorithm and implement it on GPU architectures. We eventually found a paper from 2023 called "Pebbling Game And Alternative Basis For High Performance Matrix Multiplication" [4] which we pursued for its low memory overhead.

This pebbled algorithm had only been implemented with SIMD parallelism on a CPU. We wanted to extend this to GPUs - matrix multiplication is very computationally expensive and would benefit greatly from the parallelism a GPU provides. The workload was inherently data-parallel, as matrix multiplication is independent across rows. It was also highly local, as matrices tend to be contiguously stored. However, it was riddled with data dependencies, making a fully parallel GPU implementation very complex. There were also several opportunities for low-level optimizations, as our baseline (cuBLAS) was already highly optimized.

3 Approach

3.1 Algorithm

We surveyed existing Strassen algorithm variants, and their implementations on parallel machines, to identify a variant which had low memory overhead. As mentioned above, we eventually settled on a pebbled variant of the Strassen algorithm [4]. The algorithm is:

Algorithm 1 Pebbled Strassen Algorithm

```
1: $(A_{11}, A_{12}, A_{21}, A_{22}) \leftarrow A$ {Partition $A$ into four submatrices}
2: $(B_{11}, B_{12}, B_{21}, B_{22}) \leftarrow B$ {Partition $B$ into four submatrices}
3: $A_{22} \leftarrow A_{12} - A_{21} + A_{22}$ {Basis transformation}
4: $B_{22} \leftarrow B_{12} - B_{21} + B_{22}$
5: $\text{temp}_1 \leftarrow -A_{11} + A_{22}$ {Bilinear phase}
6: $\text{temp}_2 \leftarrow -B_{11} + B_{22}$
7: $A_{11} \leftarrow A_{11} \cdot B_{11}$
8: $B_{11} \leftarrow -B_{12} + B_{22}$
9: $\text{temp}_1 \leftarrow \text{temp}_1 \cdot B_{12}$
10: $M_7 \leftarrow \text{temp}_1$
11: $B_{12} \leftarrow A_{21} + A_{22}$
12: $A_{21} \leftarrow A_{21} \cdot \text{temp}_2$
13: $M_3 \leftarrow A_{21}$
14: $\text{temp}_2 \leftarrow B_{21} + B_{22}$
15: $B_{12} \leftarrow B_{12} \cdot \text{temp}_2$
16: $M_5 \leftarrow B_{12}$
17: $\text{temp}_2 \leftarrow -A_{12} + A_{22}$
18: $A_{12} \leftarrow A_{12} \cdot B_{21}$
19: $M_2 \leftarrow A_{12}$
20: $A_{22} \leftarrow A_{22} \cdot B_{22}$
21: $M_4 \leftarrow A_{22}$
22: $\text{temp}_2 \leftarrow \text{temp}_2 \cdot B_{11}$
23: $M_6 \leftarrow \text{temp}_2$
24: $A_{11} \leftarrow A_{11} + A_{12}$
25: $A_{21} \leftarrow A_{21} + \text{temp}_2$
26: $A_{22} \leftarrow -A_{12} - A_{22}$
27: $A_{22} \leftarrow A_{22} + B_{12}$
28: $A_{22} \leftarrow A_{22} + \text{temp}_2$
29: $A_{12} \leftarrow B_{12} - \text{temp}_1$
30: $A_{12} \leftarrow A_{12} - A_{22}$ {Inverse basis transformation}
31: $A_{21} \leftarrow -A_{21} + A_{22}$
32: $C \leftarrow \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ {Combine results into matrix $C$}
33: return $C$
```

Of note is that this algorithm reduces the storage overhead, based on an exhaustive search which used pebbling to maximally-reuse intermediate storage once its result wasn't needed. Also note that stores to $M_{\{1...7\}}$ only indicate how components of the original Strassen algorithm are computed, not how they are actually computed or stored in our implementation.

For this project, we targeted 32-bit floats on V100 and RTX 2080 architectures. All profiling was done on the RTX 2080 due to its availability, and all figures except 11 are for 2080 machines. We wrote our implementation with the cuBLAS CUDA library, using the incredibly helpful work [2] of Simon Boehm (a performance engineer at Anthropic) for benchmarking and organizational purposes.

Matrices map very well to GPU architectures. Sub-matrices can be mapped to thread blocks, with each thread (and warp) being mapped to a smaller rectangular chunk of elements. We did not explicitly control this mapping for the majority of our code, as our implementation used cuBLAS (which handles the exact mapping internally). However, we did explicitly define this mapping for our double addition kernel, which we will discuss below. Our initial implementation did not alter the original serial algorithm, although we eventually made modifications to reduce memory overhead and fuse operations.

3.2 Naive Implementation

The pebbled Strassen algorithm requires several in-place matrix multiplications, which cannot be handled by `cublasSgemm`. We counteracted this by introducing an temporary variable storage, `temp3`, which was used to hold results of $A \cdot = B$, and following each matrix multiply with a `cudaMemcpy`. For example, the line $A_{22} \leftarrow A_{22} \cdot B_{22}$ became:

$$\text{temp}_3 \leftarrow A_{22} \cdot B_{22}; A_{22} \leftarrow \text{temp3}$$

Below are the GPU usage breakdown for running both a recursive and non-recursive variant, created using `nsys`:

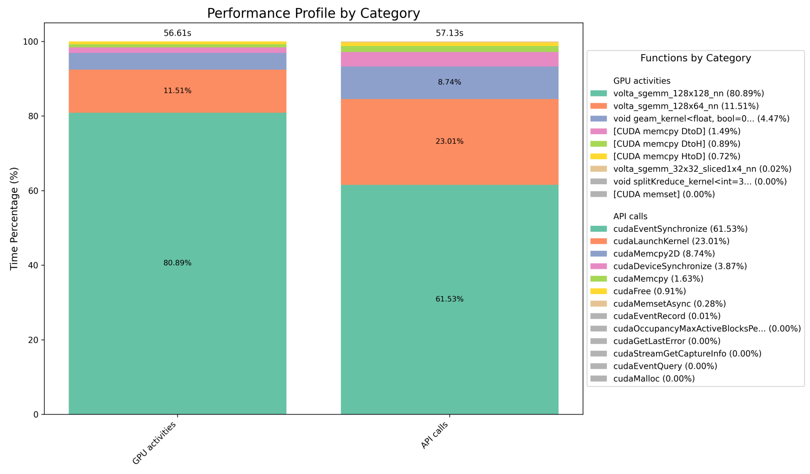

Figure 1: Naive Implementation Breakdown, Non-Recursive

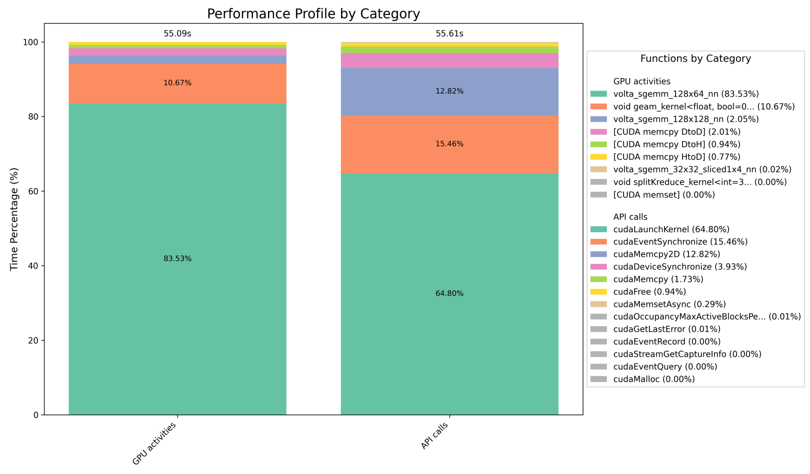

Figure 2: Naive Implementation Breakdown, Recursive

From this breakdown, we can see that recursion inside our Strassen variant allows us to substitute more multiplications for additions, increasing the relative percentage of `sgeam` from 4.47% to 10.67%. We can also see that internal memory copying (`[CUDA memcpy DtoD]`) takes 1.49% of the GPU runtime for non-recursive and 2.01% for recursive due to the overhead of in-place multiplication. We used this breakdown to improve our runtime - deciding to first accelerate the additions and then reduce mem-copies.

3.3 Custom Addition Kernel

The breakdown of our recursive implementation convinced us to write a custom CUDA kernel which computes $C = \alpha A + \beta B + C$. Implementing this computation in cuBLAS takes two calls, leading to unnecessary data movement. We designed a custom double-addition kernel to fuse this operation. Our kernel was parameterized in terms of four constants: TM, TN, BM, BN. Each thread computes the result for a $TN \times TM$ subsection of C, and each block computes the result for a $BN \times BM$ subsection of C. We set these kernel parameters with a brute-force search of the following parameter space:

$$TM \in \{1, 2, 4, 8\}$$

$$TN \in \{4, 8, 16, 32\}$$

$$BM \in \{32, 64, 128\}$$

$$BN \in \{32, 64, 128\}$$

We recorded the wall-clock time of each configuration, and ran `ncu` for more in-depth information about the compute/memory throughput and occupancy of each kernel configuration. The results of our search are presented below:

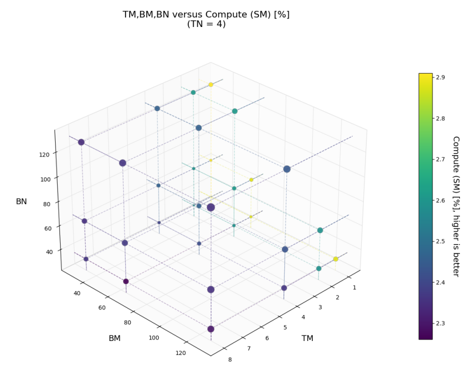

Figure 3: Tuning Results: SM Utilization

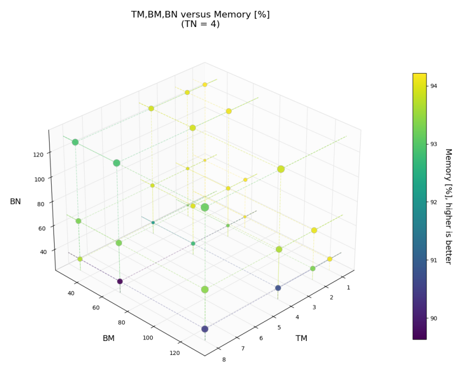

Figure 4: Tuning Results: Memory Utilization

We found that $TN = 4$ was optimal for all sizes. The figures above visualize all parameter combinations, with values of the other three parameters (TM, BM, BN) on the x, y and z axes respectively. We see that our kernel uniformly demonstrates extremely low compute utilization (at only 2.9% for the best configuration), but its memory utilization ranges from $< 90\%$ to $> 94\%$ depending on configuration. This meant we should choose a kernel with good memory utilization and performance.

We also measured performance (wall-clock time) across different matrix sizes:

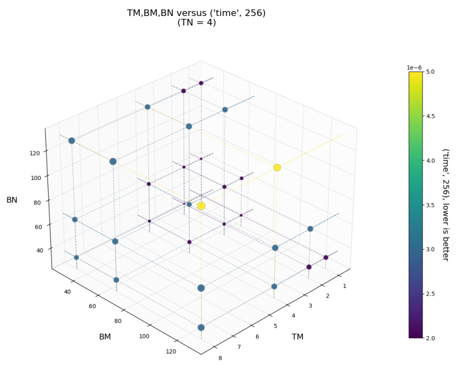

Figure 5: Tuning Performance: $N = 256$

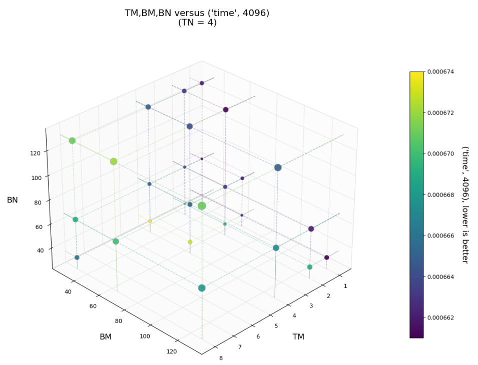

Figure 6: Tuning Performance: $N = 4096$

The optimal configuration was independent of problem sizes, and had low TM, $TN = 4$ and $BN = BM = 32$. Based on our experimental results, we settled on the following configuration: $TM = 1, TN = 4, BM = 32, BN = 32$.

We made a few other optimizations to our kernel, like float4 loading and intelligent memory hierarchy utilization. These were inspired by the optimizations made by Simon Boehm in his Matrix Multiplication Kernel article[1]

Our tuned custom double-addition kernel yielded an improvement in performance across all matrix sizes. Below is the breakdown of the algorithm when running the new kernel:

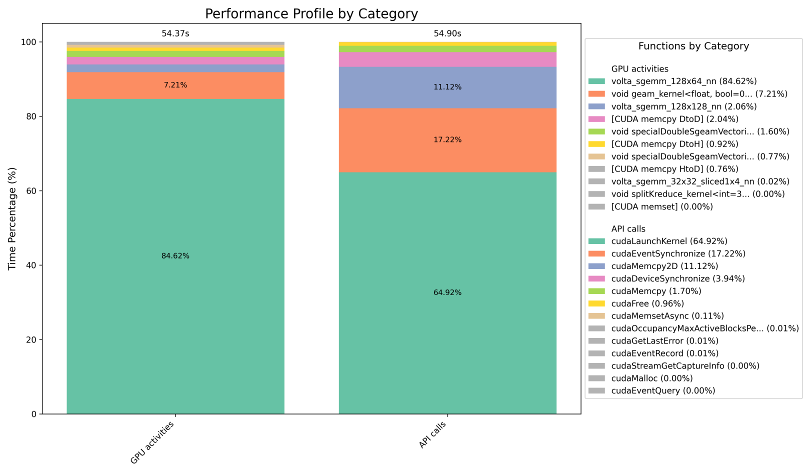

Figure 7: Implementation with Custom Add

The runtime has decreased, and the percent of time spent doing additions on the GPU went from 10.67% to $7.21 + 1.60 = 8.81\%$. Our final double-add kernel was 60% faster than the equivalent two cubLAS calls. We then turned our attention to reducing mem-copies.

3.4 Reduced Memory Copying

Our next improvement was to reduce the memory copying overhead. The Strassen variant we implemented relied on the multiplication $A \cdot = B$, which normally requires a `cudaMemcpy`. After careful consideration and correctness testing, we removed six of the seven mem-copies by swapping pointers and rearranging memory instead of copying. This gave a large performance improvement:

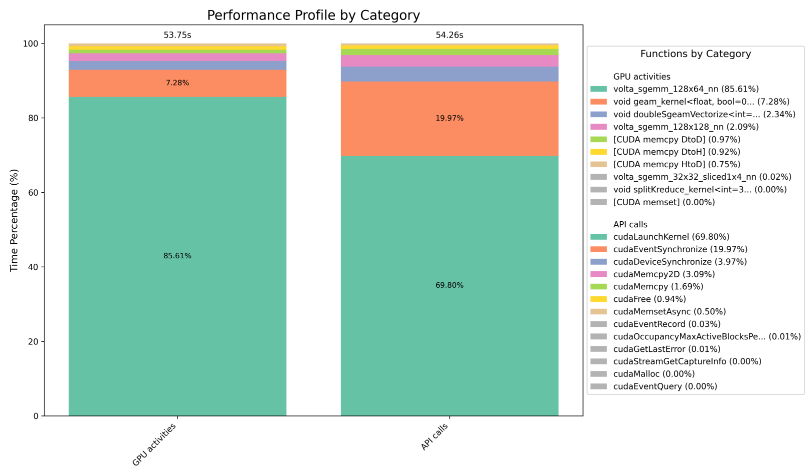

Figure 8: Implementation with Reduced Mem-copies

The `DtoD` mem-copies have been reduced from 2.04% to 0.97%, and this corresponded to a $\approx 1\%$ improvement in performance.

3.5 Dead Ends

3.5.1 Multi-Stream

Some submatrix operations can theoretically be done simultaneously, so we tried using multiple streams to improve performance. Unfortunately, the data dependencies endemic to the pebbled Strassen algorithm made this difficult and capped our implementation at 2 consecutive streams. Even with 2 streams, there was minimal inter-stream overlap because of the latency to launch the kernels and contention for GPU resources. The two `volta_sgemm_128x64_nn` calls shown in Figure 9 occur on different streams but have minimal overlap - we only observe overlap at the *end* of one stream's kernel, when it starts releasing GPU resources. There were also issues balancing the work between the two streams, which can be seen in Figure 9, where one stream has 64.8% of the usage and the other has 29.7% of the usage. These factors combined meant that the performance gain from streams wasn't worth the overhead of creating a new stream and cuBLAS handle.

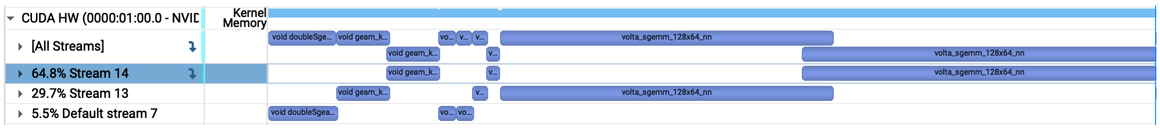

Figure 9: Multi-Stream Approach, Nsight Systems

3.5.2 Prefetching

We experimented with prefetching, but it had a negligibly negative effect on performance. We believe this is due to our high memory throughput - we already had 94% memory throughput, and there is simply no latency to hide.

3.5.3 Device Pointers

The cuBLAS APIs for matrix addition (`sgeam`) and matrix multiplication (`sgemm`) take in pointers to constants α and β, when computing $C = \alpha A + \beta B$ and $C = \alpha AB + \beta C$ respectively. By default, these pointers are located on the host (CPU), limiting asynchronous launch of library routines. Since we know the constants required for Pebbled-Strassen at compile time, we created on-device globals and passed pointers to them. This reduced latency from host-to-device mem-copies, but the overhead from switching pointer modes from on-host to on-device pointers was larger than the improvement from on-device pointers.

4 Results

4.1 Summary

We implemented, optimized and extensively profiled the pebbled Strassen Algorithm. We reduced our memory overhead through intelligent reuse and recomputation and experimented with the memory hierarchy, multiple streams, prefetching and custom kernels to boost perfomance. In the end, we outperformed a cuBLAS baseline on matrices with dimension $N \geq 4096$ on the V100 and RTX 2080 architectures. All things considered, we achieved all of our major goals.

4.2 Performance

We choose to optimize the wall-clock time our implementation took to multiply 2 matrices. We experimented with square matrices with $N \in \{128, 256 \ldots 16, 384\}$. We tested our implementation by ensuring its output matched that of a cuBLAS reference, using the cuBLAS performance (averaged over 50 times) as a baseline time. We then ran our implementation 50 times and recorded the average runtime. We repeated this process for all matrix sizes and constructed runtime and speedup graphs.

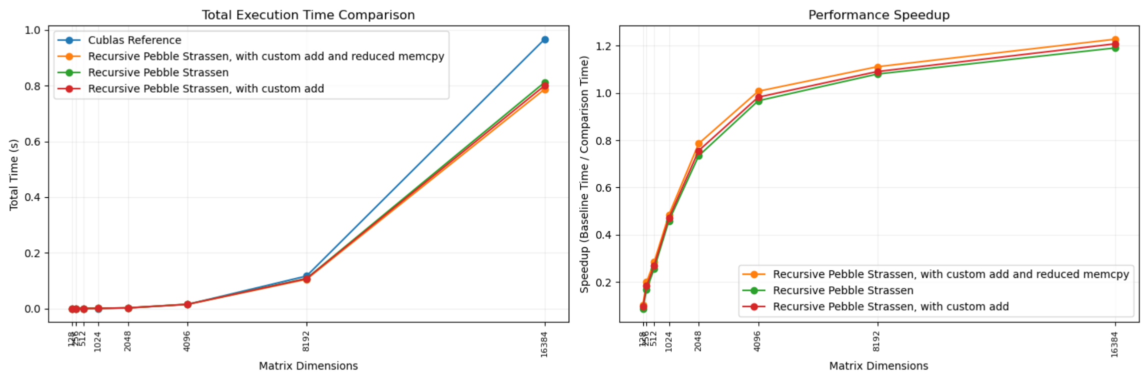

Figure 10: Performance and Speedup on RTX 2080 (GHC Cluster)

On the GHC RTX 2080s, we tested three major versions of our implementation against a cuBLAS implementation. The first version was the basic implementation of the recursive pebbled Strassen algorithm, the second one used a custom double addition kernel and the third one reduced the number of mem-copies. All three implementations surpass the cuBLAS baseline when $N \geq 8192$, but only the final one beats cuBLAS at $N = 4096$.

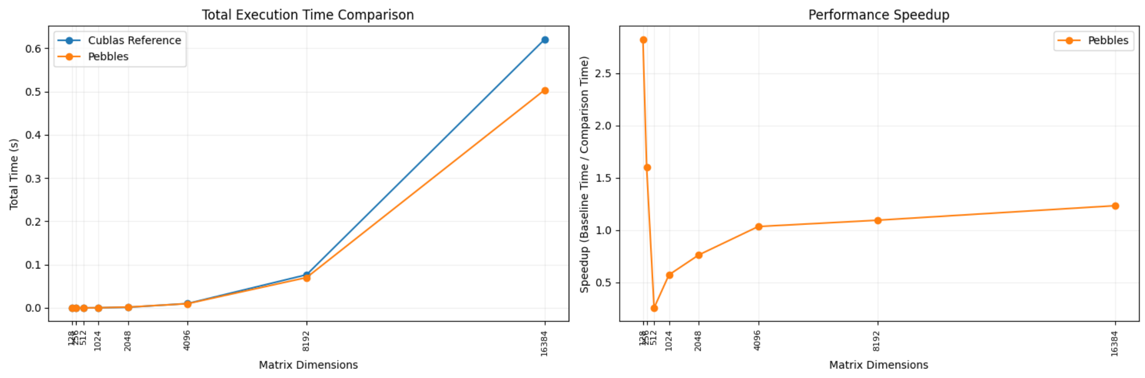

Figure 11: Performance and Speedup on V100 (PSC Cluster)

We only tested the cuBLAS baseline and final implementation (custom kernel and reduced memcpys) on the PSC V100s. We believe that the spike in speedup at $N = 128$ and $N = 256$ are due to warm up costs. That being said, our final implementation consistently outperforms the cuBLAS baseline on $N \geq 4096$ on both 2080's and V100's.

Overall, we performed better on larger inputs than small ones. This was expected. As discussed earlier, the Strassen algorithm is *asymptotically* more efficient, as it replaces 1 of the 8 expensive sub-matrix multiplications with 14 additions instead. At smaller sizes, this efficiency is limited by data dependencies and data movement overhead; for example, in our profiling, multiplying two 128×128 matrices took roughly the same amount of time as adding two matrices of the same size, which is why we have a less-than-one speedup for small sizes. However, this overhead is overpowered by the asymptotic advantage as the matrices scale, improving speedup.

4.3 Implementation Analysis

We effectively reduce computation time for $N \geq 4096$ by taking advantage of the GPU's high throughput. This can be seen in Figure 12, which provides an annotated timeline-view of our final iteration running a single matrix multiply. Nsight Systems (`nsys`) indicates a near-100% "utilization" of the GPU, meaning that the GPU is always running *something* and communication time between the GPU and CPU is **minimal** in comparison to the time it takes to execute kernels. This can also be seen in Figure 8, as the difference between the time spent in GPU Activities and API calls indicates minimal communication beyond copying the matrix to/from the GPU.

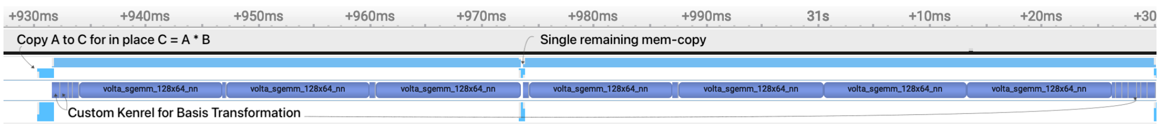

Figure 12: Nsight Profile of Final Iteration

`sgemms`: cuBLAS multiplies

Small unlabled regions in between the `sgemms`: `sgeam`, matrix-adds outlined in Section 3.1

In our current iteration, the vast majority of our implementation's runtime comes on calling cuBLAS functions (see 8: $85.61 + 2.09 = 87.70\%$ on `sgemm`, $7.28 + 2.34 = 9.62\%$ on `sgeam`). We assume that these functions are near-optimal at computing a single matrix multiplication, so focus instead on calling them effectively, fusing operations where possible and minimizing data movement. We tried methods to potentially call these functions more effectively (streaming and prefetching), but this did not provide a performance increase. We also reduced our memory overhead (via removing mem-copies). This leads us to believe that our implementation is as optimized as possible without rewriting cuBLAS.

We used Nsight Compute (`ncu`) for an in-depth exploration of our custom kernel, profiling memory and SM utilization. Our custom kernel achieves a 94% memory throughput and 2.9% SM utilization, indicating near-optimal usage of the limiting resource for matrix addition which is memory throughput. These experimental results, combined with our extensive search of the parameter space leads us to believe that this kernel has been improved as much as possible.

Based on our analysis of our final implementation and our experiments implementing multiple-streams and prefetching (which yielded no or negative improvement), we believe that further improvement in performance would need to come from a fundamental change in algorithm choice–building a new Strassen variant which takes into consideration the constraints and advantages of the GPU. For example, the algorithm we implemented was designed to minimize memory overhead, requiring only one additional $N \times N$ matrix's worth of space (or $\frac{3}{4}$ if we did more mem-copy's). However, this focus on memory overhead led to data dependencies and reduced parallelism that limited our overall performance and the algorithm's speedup.

4.4 Machine Target Choice

Our choice of machine target was appropriate - Matrix Multiplication is a highly parallel algorithm and GPUs are built for highly parallel applications. The paper[4] which we got our variant from implemented their algorithm on multi-core CPUs, but GPUs are more heavily used in industry for large Matrix Multiplications in (primarily) Deep Learning tasks, which is why we pursued them instead. TPUs and systolic arrays would also be appropriate for Matrix Multiplication tasks, but were out of scope for this project.

We selected V100 GPUs because they are prevalently used in training and inference for deep learning models, and the RTX 2080 GPU becuase it's a high-end consumer model which is often used for rendering graphics, either for video games or digital artists/special effects. Both of these domains feature large matrix multiplications and could benefit from our optimizations.

References

[1] Simon Boehm. How to optimize a cuda matmul kernel for cublas-like performance: a worklog, December 2022. Accessed: 2025-04-15.

[2] Simon Boehm. Sgemm cuda, December 2022. Accessed: 2025-04-27.

[3] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Strassen's matrix multiplication on gpus. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 157–164, 12 2011.

[4] Oded Schwartz and Noa Vaknin. Pebbling game and alternative basis for high performance matrix multiplication. *SIAM Journal on Scientific Computing*, 45(6):C277–C303, 2023.

5 Work Distribution

Task	**Elijah%**	**Tanay%**
Research Spike on Strassen Implementation	50	50
Strassen Reference Implementation *Python*	100	0
Pebbled-Strassen Reference Implementation *Python*	100	0
Pebbled-Strassen Baseline Implementation *Cuda*	80	20
Refactor timing and evaluation code	0	100
Evaluate runtime and memory overhead for baselines	50	50
Continue implementing memory overhead improvements	0	100
Explore multi-stream	20	80
Profile multi-stream, explain results	100	0
Explore prefetching	0	100
Improve with custom kernels	20	80
Programmatically tune custom kernel, visualize	80	20
Explore performance on various GPUs (2080, V100)	100	0
Profile with `nsys` and `ncu`	100	0
Generate documentation	100	0
Merge, clean code base	20	80
Graphics for Report	100	0
Report Writing	50	50
Poster	50	50

Table 1: Per-Task Work Breakdown

Elijah%	**Tanay%**
50	50

Table 2: Overall Work Breakdown