VRIJE
UNIVERSITEIT
BRUSSEL

GAUSSIAN ELIMINATION

on GPU, with Naive and Optimization Approaches

Chenhua Li

August 27, 2025

Master in Applied Computer Science

1 GPU Info

1.1 Personal Desktop Computer

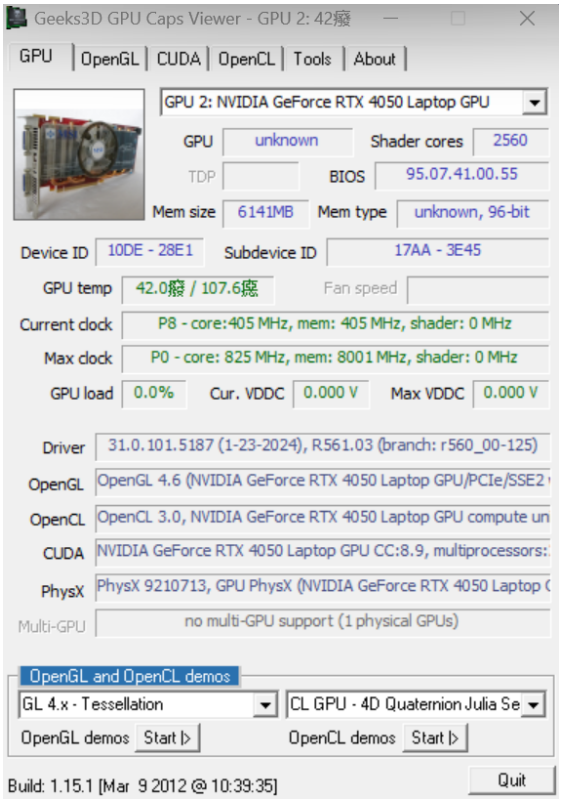

Figure 1: GPU Information

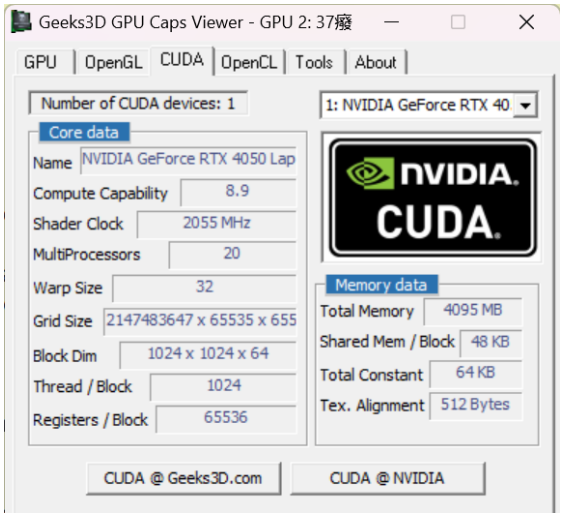

Figure 2: GPU CUDA Information

The key specifications of the graphics system are as follows:

- **Model**: NVIDIA GeForce RTX 4050 Laptop GPU
- **Multiprocessors**: 20 SMs (Streaming Multiprocessors)
- **CUDA Cores**: 2560 total (128 CUDA cores per SM)
- **Memory**: 4095 MB GDDR6
- **Shader Clock**: 2055 MHz
- **Warp Size**: 32 threads
- **Threads per Block**: 1024

1.2 ETRO Farm

The ETRO Farm is a more powerful system:

- **Model**: NVIDIA A100 80GB PCIe
- **Memory**: 81920 MB (80 GB) HBM2e
- **Architecture**: Ampere
- **Multiprocessors**: 108 SMs (Streaming Multiprocessors)
- **CUDA Cores**: 6912 total (64 FP32 cores per SM)
- **Tensor Cores**: 432 (3rd generation)
- **Memory Bandwidth**: ~2 TB/s
- **Compute Capability**: 8.0

1.3 Theoretical Peak Performance

The theoretical peak performance of the GPU for RTX 4050 can be calculated as:

$$\text{FLOPS} = \text{Number of CUDA Cores} \times \text{Clock Frequency} \times \text{Operations per Clock} \tag{1}$$

$$\text{FLOPS} = 2560 \text{ cores} \times 2.055 \text{ GHz} \times 2 \text{ FP32 operations/cycle} = 10.52 \text{ TFLOPS (FP32)} \tag{2}$$

2 Problem Description

Gaussian elimination, as one of the most known methods to solve a system of linear equations, is chosen as a continuation from Parallel Programming course to run experiments to see potential speedup when running on GPU (and when specific GPU-related optimization techniques are used).

Systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where:

- $\mathbf{A}$ is an $n \times n$ coefficient matrix
- $\mathbf{x}$ is the n-dimensional unknown vector we aim to solve
- $\mathbf{b}$ is the n-dimensional right-hand side vector

The algorithm consists of two phases:

1. Forward Elimination: Creates an upper triangular matrix by eliminating variables

   The algorithm transforms the system from:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{3}$$

   Into:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\ddots \\ a'_{nn}x_n &= b'_n \end{aligned} \tag{4}$$

2. Back Substitution: Solves for variables starting from the last equation (from bottom to top)

Algorithm 1 Sequential Gaussian Elimination

```
0: function SequentialGE(A, b)
0:     for k ← 0 to n − 2 do
0:         r ← FindMaxPivotRow(k)
0:         if k ≠ r then
0:             Swap rows k and r in A, b
0:         end if
0:         for i ← k + 1 to n − 1 do
0:             l ← A[i][k]/A[k][k]
0:             for j ← k + 1 to n − 1 do
0:                 A[i][j] ← A[i][j] − l · A[k][j]
0:             end for
0:             b[i] ← b[i] − l · b[k]
0:         end for
0:     end for
0:     return BackSubstitution(A, b)
0: end function=0
```

3 Experimental Setup

The experiments were performed using OpenCL on an NVIDIA GPU and Etro Farm (key info specified in the GPU info section), with each task focusing on different factors impacting the performance.

In the experiments, the following metrics are measured:

- GPU runtime
- Computational performance in GOps/s
- Memory throughput in GB/s

4 Implementation

Naive GPU Implementation (gaussianElimination.cpp) processes one pivot column at a time, as kernel keeps launching, it handles fewer and fewer rows, synchronization is added between the kernels

Algorithm 2 Naive GPU Gaussian Elimination

function NaiveGpuGE(A, b)
  Transfer A, b to GPU memory
  for $k \leftarrow 0$ to $n-2$ **do**
    Launch kernel with $(n-k-1)$ threads
    Kernel: Each thread i processes row $k+1+i$
      $factor \leftarrow A[row][k]/A[k][k]$

    for $j \leftarrow k$ to $n-1$ **do**
        $A[row][j] \leftarrow A[row][j] - factor \cdot A[k][j]$

    end for
      $b[row] \leftarrow b[row] - factor \cdot b[k]$
    Synchronize all threads
  end for
  Launch BackSubstitutionKernel(A, b)
  Transfer result x from GPU
  return x
end function=0

Optimized Implementation (optimizedGaussianElimination.cpp) used blocked and coalesced memory access strategies.

The blocked strategy caches frequently accessed pivot row data in shared memory, allowing workgroups to cooperatively load and reuse this data rather than accessing global memory. This should reduce memory traffic and takes advantage of the GPU's fast on-chip memory hierarchy.

Algorithm 3 Blocked GPU Gaussian Elimination with Shared Memory

function BLOCKEDGPUGE(A, b, W)
  Transfer A, b to GPU memory
  for $k \leftarrow 0$ to $n-2$ **do**
    $workgroups \leftarrow \lceil (n-k-1)/W \rceil$
    Launch kernel with $workgroups \times W$ threads
    Kernel: Allocate shared memory $pivot\_cache[n-k+1]$
      Cooperatively load pivot row into $pivot\_cache$
      $pivot\_cache[n-k] \leftarrow b[k]$ {Cache RHS value}
      Barrier synchronization
      $row \leftarrow k+1+global\_id$

    if $row < n$ **then**
        $factor \leftarrow A[row][k]/pivot\_cache[0]$

      for $j \leftarrow k$ to $n-1$ **do**
          $A[row][j] \leftarrow A[row][j] - factor \cdot pivot\_cache[j-k]$

      end for
        $b[row] \leftarrow b[row] - factor \cdot pivot\_cache[n-k]$

    end if
    Synchronize all workgroups
  end for
  Launch BACKSUBSTITUTIONKERNEL(A, b)
  Transfer result x from GPU
  return x
end function=0

The coalesced strategy takes a different approach by restructuring the parallelization pattern to achieve better memory bandwidth utilization. Instead of assigning entire rows to threads, it maps threads to individual matrix elements, ensuring that adjacent threads access consecutive memory locations. This maximizes the efficiency of memory transactions and better exploits the GPU's memory subsystem, though it comes with the trade-off of some redundant computations as multiple threads may calculate the same elimination factors.

Algorithm 4 Coalesced GPU Gaussian Elimination

```
function CoalescedGpuGE(A, b)
    Transfer A, b to GPU memory
    for k ← 0 to n − 2 do
        total_elements ← (n − k − 1) × (n − k)
        Launch kernel with total_elements threads
        Kernel: Map thread to matrix element
            row_offset ← global_id ÷ (n − k)
            col_offset ← global_id mod (n − k)
            row ← k + 1 + row_offset
            col ← k + col_offset

        if row < n and col < n then
                factor ← A[row][k]/A[k][k]
                A[row][col] ← A[row][col] − factor · A[k][col]

            if col_offset = 0 then {First thread of row}
                    b[row] ← b[row] − factor · b[k]

            end if

        end if
        Synchronize all threads
    end for
    Launch BackSubstitutionKernel(A, b)
    Transfer result x from GPU
    return x
end function=0
```

5 Result and Discussion

Verifications from GPU and CPU confirm the correctness of the result. We directly compared the CPU and GPU outcomes.

Although the allowed difference was set to 10^{-3}, the errors were found to be below 10^{-5}.

5.1 Naive vs. Optimization Methods

5.1.1 Runtime Comparison

According to Figure 3, we observe that coalesced performs the fastest. Notably, blocked method performs worse than naive for 512×512 matrices (0.87×). The blocked method scales better at larger matrices.

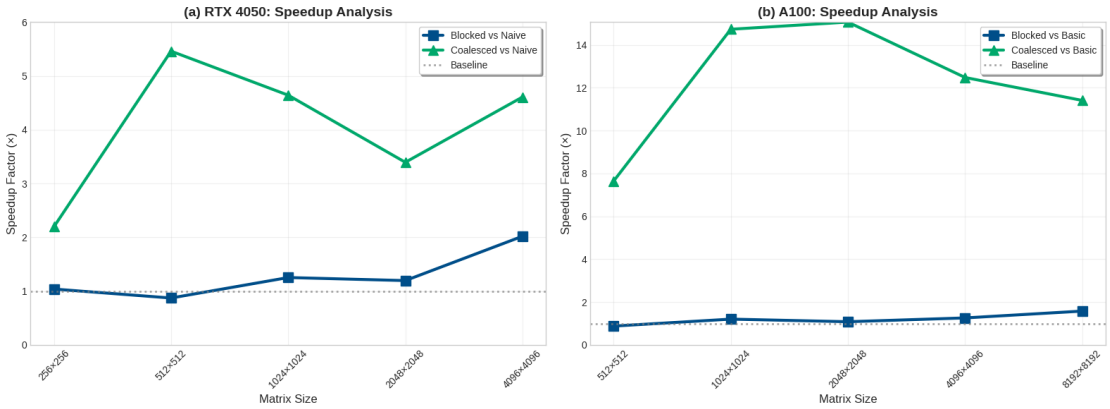

Figure 3: GPU-vs-GPU Speedup between Optimization vs. Naive Method

The intended optimization we aim for to have in the blocked approach is to cache the pivot row in shared memory, however, this approach maps thread to rows, resulting in coarse-grained parallelism, while coalesced maps threads to individual elements. And in cases where matrix size is not sufficiently large, the cost of computational overhead outweighs the benefit it brings even.

From the experimental data and some analysis of the algorithm itself, it is not hard to observe and deduce the Gaussian Elimination is a memory-bound algorithm.

The Operational Intensity is calculated from the the ratio of floating-point operations (FLOPs) to total data movement (memory traffic in bytes), which is about $(n/3)/(8n)0.042FLOPS/Byte$, much less than the RTX 4050 Threshold, which is $3.5FLOPS/Byte$. Thus optimize with regard to memory access patterns turns out to be much more efficient.

5.1.2 Memory Bandwidth Comparison

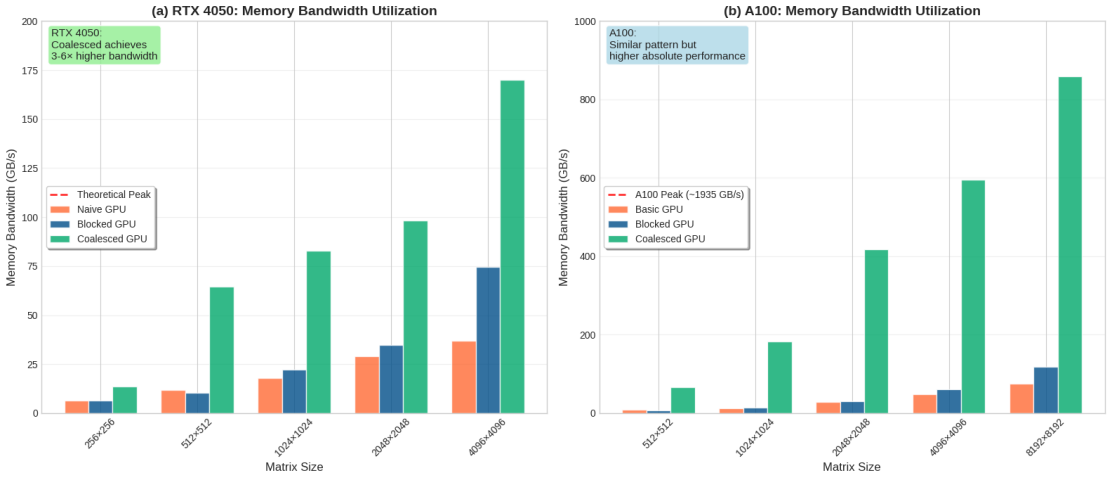

Figure 4: Bandwidth Comparison between Optimization vs. Naive Method

Since we know gaussian elimination is a memory-bound algorithm. Memory bandwidth uti-

lization is the key metric, from Figure 4 we see that coalesced approach consistently outperforms blocked and naive methods by a significant factor.

5.2 Roofline Model

To get the roofline model, the key is to reach the compute-bound region.

Inspired from previous methods in the mini project(add arithmetic operations manually by increasing loop count), similar approach is applied here in the `forwardEliminationCoalescedIntensive` kernel. After observing simply increasing loop count will result in automatic optimization of the compiler, tiny modifications are applied (factor * (j * $1e-10f$)) and scaling operations (*= $0.999999f$), these modifications prevent compiler optimization without tempering the correct results. These operations created computational work that the GPU must execute, transforming a memory-bound operation into a more compute-bound one.

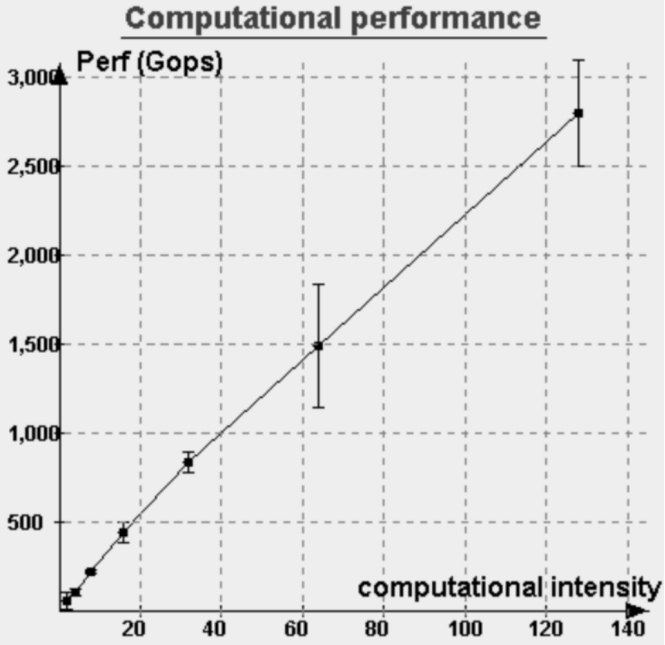

Figure 5: Microbenchmark Roofline Model on RTX 4050

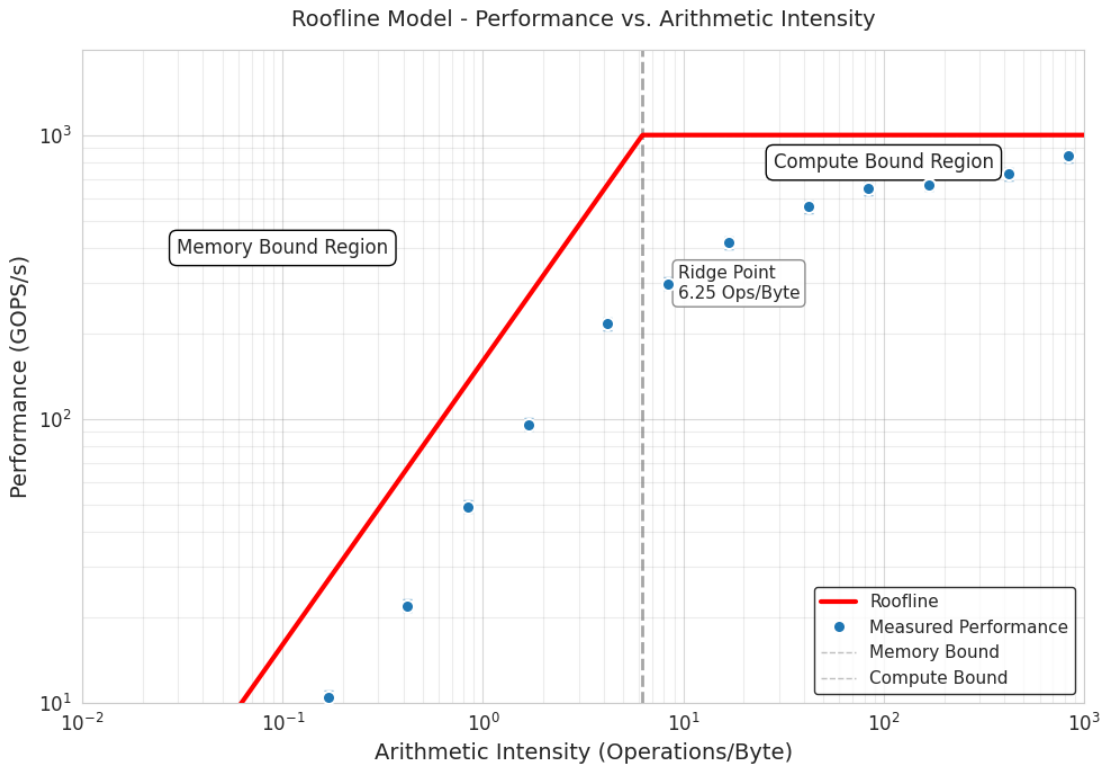

Figure 6: Roofline Model on RTX 4050 Based on Measurements

According to Figure 6, the ridge point occurs at approximately 6.25 Operations/Byte. In the memory-bound region, performance scales linearly with arithmetic intensity. After the ridge point, performance plateaus at approximately 800 $GOPS/s$. Compared with the theoretical peak performance from the benchmarking tool, where peak performance is 2800 $GOPS/s$. This indicates that while the RTX 4050 has theoretical performance boost potential to still exploit. However, it could be due to the inherent unparallelizable operation within gaussian elimination itself, as some operations are unable to parallize in the scope of this specific problem.

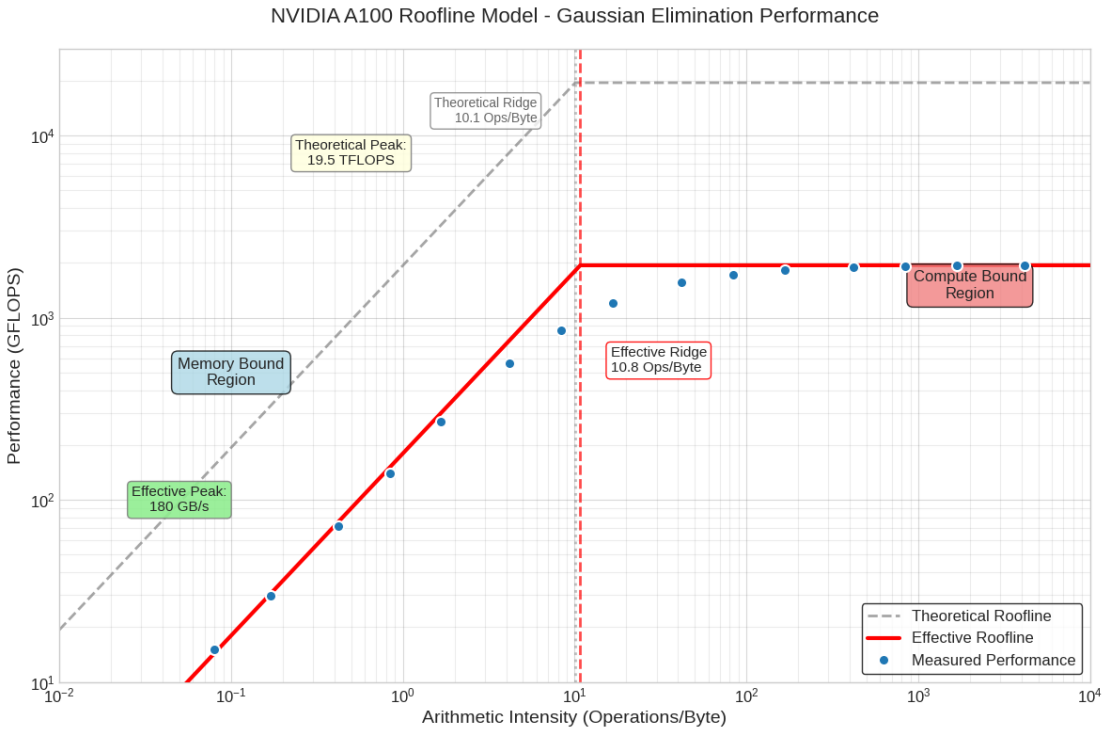

Figure 7: Roofline Model on A100

The theoretical ridge point on A100 is calculated by the formula:

Ridge Point = Peak Performance (FLOPS) ÷ Peak Bandwidth (Bytes/s)

Where the GPU information is retrieved from [1], 19.5 TFLOPS (FP32) and 1,935 GB/s memory bandwidth, giving the 10.1 Ops/Byte theoretical ridge point. In measurement as shown in Figure 7, it also reaches similar ridge point, at approximately 10.8 Ops/Byte

The A100 has higher ridge point than RTX 4050, indicates better balance between compute and memory capabilities.

6 Conclusion

The experiments confirmed the correctness of the GPU implementations, with numerical errors consistently below 10^{-5}.

Among the three methods, the coalesced implementation achieved the highest performance, reaching up to two orders of magnitude speedup over the naive baseline, while the blocked method was less effective at smaller sizes but scaled better at larger matrices. The results confirm that Gaussian Elimination is fundamentally memory-bound, and optimizations that improve memory access patterns (such as coalesced accesses) yield the most significant performance gains.

The roofline analysis on RTX 4050 showed a ridge point around 6.25 FLOPs/Byte and a performance plateau near 800 GFLOPS, well below the theoretical peak of 2800 GFLOPS, highlighting limits imposed by algorithmic dependencies. On the A100, the ridge point was measured near 10.8 FLOPs/Byte, close to the theoretical 10.1 FLOPs/Byte, reflecting a better balance between compute and memory capabilities.

References

[1] *NVIDIA A100 Tensor Core GPU Data Sheet*. Tech. rep. PDF – 3 pages. NVIDIA Corporation, June 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

Appendix: Raw data

Table 1: Gaussian Elimination Performance on RTX 4050 (Naive, Blocked, Coalesced)

Matrix Size	Implementation	GPU Time (μs)	CPU Time (μs)	Speedup vs CPU	GFLOPS	Bandwidth (GB/s)
256x256	Naive	10,736	31,215,	2.91x	0.527	6.25
	Blocked	10,337	30,442,	2.94x	0.550	6.49
	Coalesced	4,890	30,171,	6.17x	1.16	13.72
512x512	Naive	45,420	245,248	5.40x	0.991	11.82
	Blocked	52,095	244,091	4.69x	0.86	10.31
	Coalesced	8,319	242,310	29.13x	5.41	64.54
1024x1024	Naive	241,128	1,948,653	8.08x	1.489	17.81
	Blocked	192,514	1,954,006	10.15x	1.86	22.31
	Coalesced	51,949	6,049,457	116.45x	6.91	82.68
2048x2048	Naive	1,185,108	99,845,618	84.25x	2.42	28.99
	Blocked	990,994	111,107,757	112.12x	2.89	34.67
	Coalesced	349,154	101,217,294	289.89x	8.21	98.41
4096x4096	Naive	7,457,144	892,688,707	119.71x	3.07	36.86
	Blocked	3,692,310	872,280,134	236.24x	6.21	74.45
	Coalesced	1,618,551	943,153,224	582.71x	14.16	169.83

Table 2: Gaussian Elimination Performance on A100 (Basic, Blocked, Coalesced)

Implementation	Matrix Size	GPU Time (μs)	Speedup	GFLOPS	Bandwidth (GB/s)
Basic	512x512	61,185	4.48x	0.735	8.78
	1024x1024	345,008	6.31x	1.040	12.45
	2048x2048	1,239,927	14.00x	2.313	27.71
	4096x4096	5,770,480	24.08x	3.973	47.64
	8192x8192	29,234,550	37.99x	6.271	75.22
Blocked	512x512	69,602	3.81x	0.65	7.71
	1024x1024	286,524	7.45x	1.25	14.99
	2048x2048	1,145,112	14.59x	2.50	30.01
	4096x4096	4,579,150	29.13x	5.01	60.03
	8192x8192	18,516,113	57.63x	9.90	118.76
Coalesced	512x512	8,000	33.28x	5.63	67.11
	1024x1024	23,420	89.14x	15.33	183.39
	2048x2048	82,307	203.06x	34.84	417.46
	4096x4096	462,358	289.00x	49.58	594.51
	8192x8192	2,562,162	414.81x	71.55	858.27

Table 3: Roofline Model Results for RTX 4050 (512×512)

Loop Count	CPU Time (μs)	Original GPU Time (μs)	Intensive GPU Time (μs)	Original GFLOPS	Intensive GFLOPS
1	241,160	8460	11207	5.32	4.02
2	244,615	8996	8597	5.00	10.47
5	239,767	7732	10249	5.82	21.95
10	240,862	8135	9185	5.53	48.99
20	247,061	10737	9432	4.19	95.42
50	242,436	7933	10372	5.67	216.94
100	239,951	7874	15011	5.72	299.79
200	243,011	10592	21556	4.25	417.53
500	244,048	9713	40270	4.63	558.75
1000	239,775	11979	69303	3.76	649.34
2000	243,907	10821	134611	4.16	668.61
5000	242,421	11745	308229	3.83	730.00
10000	239,971	11176	534441	4.03	842.03

Table 4: Roofline Model Results for A100 (1024×1024)

Loop Count	Operational Intensity	GFLOPS	Bandwidth (GB/s)	GPU Time (μs)
1	0.08	14.77	176.76	24,298
2	0.17	29.87	178.73	24,031
5	0.42	71.87	171.98	24,973
10	0.84	142.25	170.20	25,235
20	1.67	267.80	160.21	26,808
50	4.18	560.86	134.21	32,001
100	8.36	868.40	103.90	41,336
200	16.72	1189.01	71.13	60,380
500	41.79	1548.23	37.05	115,927
1000	83.58	1723.22	20.62	208,309
2000	167.15	1828.25	10.94	392,684
5000	417.89	1894.89	4.53	947,188
10000	835.77	1918.59	2.30	1,870,973
20000	1671.55	1930.88	1.16	3,718,124
50000	4178.87	1938.39	0.46	9,259,300