Assignment 3

Team_09

Team Members:

Jahnavi Valisetty
Abhishek Vanamala

Lab 1

Introduction

Threads of a block are executed in a groups called warps. There are 32 threads in a warp, each having a consecutive threadIdx.
The 32 threads execute code in a SIMD-like fashion, all together and very quickly. They only execute together when there's no branching within the 32 threads. If there is branching within the 32 threads, the warp will be split up and each branch will be executed one after the other.
Shared memory, like all memories must be read into registers before being operated on. Threads of a warp make requests from shared memory simultaneously. They form some pattern of 32 addresses. Each thread can request a different address and all 32 threads will get their data simultaneously if possible.
Warps are never created across blocks. If a thread block has less than 32 threads, it will actually execute all 32 threads for the warp and ignore or mask out the upper threads.

The delay between threads making a request from shared memory and getting their data may not be a problem at all even when there are bank conflicts. If there are many threads running on an SM, the scheduler can simply switch to another warp while the banks figure out the requested data.
The latency of the worst possible request pattern might be completely hidden and irrelevant.
The only real way to know if bank conflicts are showing an algorithm is to experiment with padding and access patterns.
Threads in different blocks do not cause bank conflicts with each other. Bank conflicts are only an important consideration on the warp level, not the blocks. For instance, if you've a kernel which allocates 32 floats of shared memory, then the second block's shared memory will be placed directly after the first. But the two blocks will have no trouble reading from shared memory at the same time.
If 32 threads of a warp each access a different bank from shared memory, the performance is extremely fast! The easiest thing to do is have every thread in a warp access consecutive words from shared memory.
The performance hit for a few bank conflicts is minimal compared to going to the L2 cache or global memory and the scheduler may be able to hide latency if there's enough blocks in flight at once.

Matrix Transpose Optimization

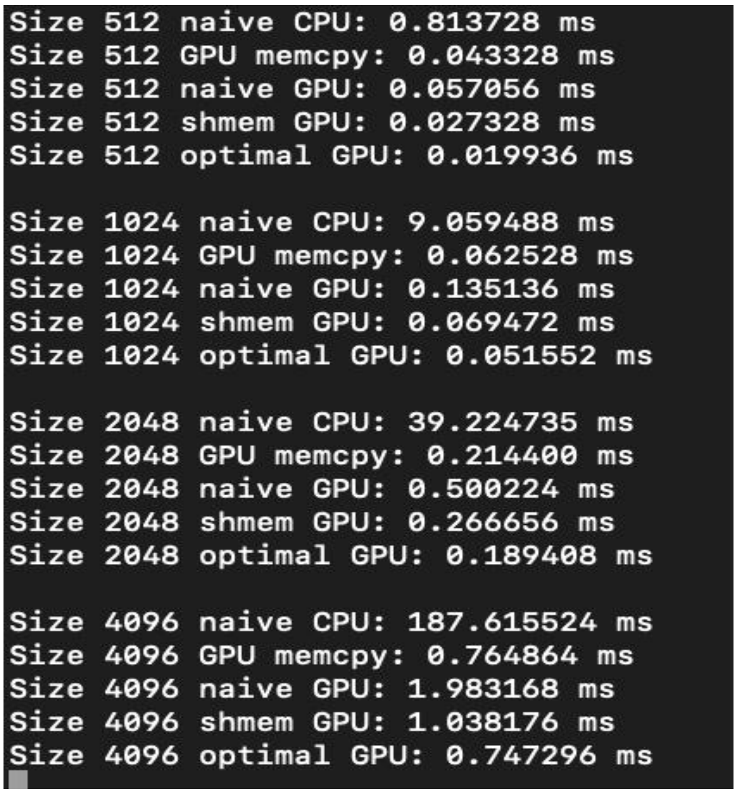
In order to optimize CUDA matrix transpose implementation, we need to avoid bank conflicts and coalescing of read and write operations have to be done. For this we are applying padding to shared memory and using loop unrolling to avoid instruction dependencies.
Each submatrix of dimension 32 x 4 is handled by one warp and each column is handled by a thread so that a warp reads one row per instruction from 32 columns. Hence read is coalesced by considering that input matrix is in row major format. But as each warp will be writing to 32 consecutive rows , the write is not coalesced. As a result, a warp writes to 32 different 128 byte cache lines.

In order to store the 64x64 matrix per block, we will use shared memory in which a warp reads in a row and writes as a column to shared memory. Th read is coalesced as we want to read a row at a time and in order to avoid bank conflicts, we do padding to the shared memory so that a warp reads a row from shared memory and writes to a row to output using the transposed indices which are correct. Then we will be writing into the row again so that the write is coalesced.

The purpose of the shmemTransposeKernel is to demonstrate proper usage of global and shared memory in which we tried to modify transpose kernel to use shared memory. All global reads and writes are coalesced and we minimized the number of shared memory bank conflicts as follows:

We were able to avoid bank conflicts since we were accessing the shared memory stride 65. Submatrix of dimension 64x64 will be stored in shared memory and is padded by a column at the end.
Reading from input is coalesced as every warp reads from 32 consecutive columns as 1 row per instruction. A warp reads and writes as a column in 1 instruction. By padding the shared memory we are removing bank conflicts as there will no threads sharing the same bank.
After which there are no bank conflicts as each warp reads a row from shared memory. By using the transposed indices, we will write a row as a row into output and the write is coalesced as a warp writes 32 consecutive columns as one row per one instruction.

Then the optimalTransposeKernel should be built on top of shmemTransposeKernel. Here we used loop unrolling optimization trick to improve performance. This technique is a loop transformation technique which helps to optimize the execution time of a program where we basically remove or reduce iterations.
As we are unrolling, they could be performed at the same time as they are not dependent on each other. As we are removing the instruction dependencies, the performance is improved as below:



```
Size 512 naive CPU: 0.813728 ms
Size 512 GPU memcpy: 0.043328 ms
Size 512 naive GPU: 0.057056 ms
Size 512 shmem GPU: 0.027328 ms
Size 512 optimal GPU: 0.019936 ms

Size 1024 naive CPU: 9.059488 ms
Size 1024 GPU memcpy: 0.062528 ms
Size 1024 naive GPU: 0.135136 ms
Size 1024 shmem GPU: 0.069472 ms
Size 1024 optimal GPU: 0.051552 ms

Size 2048 naive CPU: 39.224735 ms
Size 2048 GPU memcpy: 0.214400 ms
Size 2048 naive GPU: 0.500224 ms
Size 2048 shmem GPU: 0.266656 ms
Size 2048 optimal GPU: 0.189408 ms

Size 4096 naive CPU: 187.615524 ms
Size 4096 GPU memcpy: 0.764864 ms
Size 4096 naive GPU: 1.983168 ms
Size 4096 shmem GPU: 1.038176 ms
Size 4096 optimal GPU: 0.747296 ms
```

Fig 1. Screenshot of the optimized code which shows the improved performance
As it can be inferred from the screenshot above, the performance level of shmem GPU and optimal GPU increased from before by reducing the time of execution.

Graphs

MATRIX SIZE 512

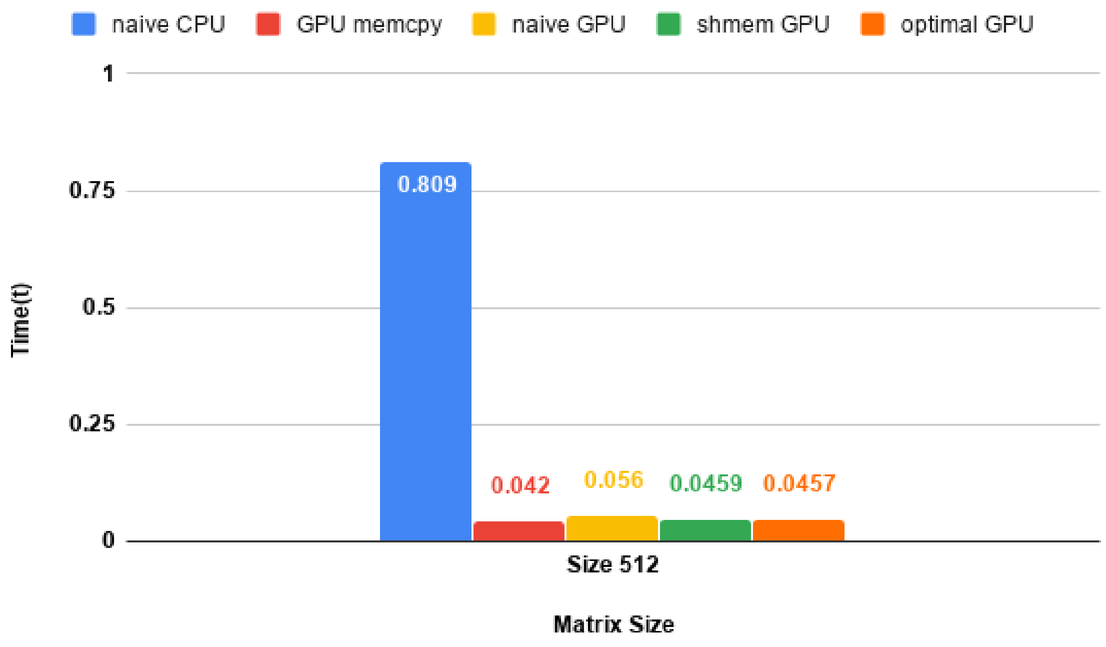


Fig 2. Time taken vs. implementations before performing optimization for matrix size 512

MATRIX SIZE 512

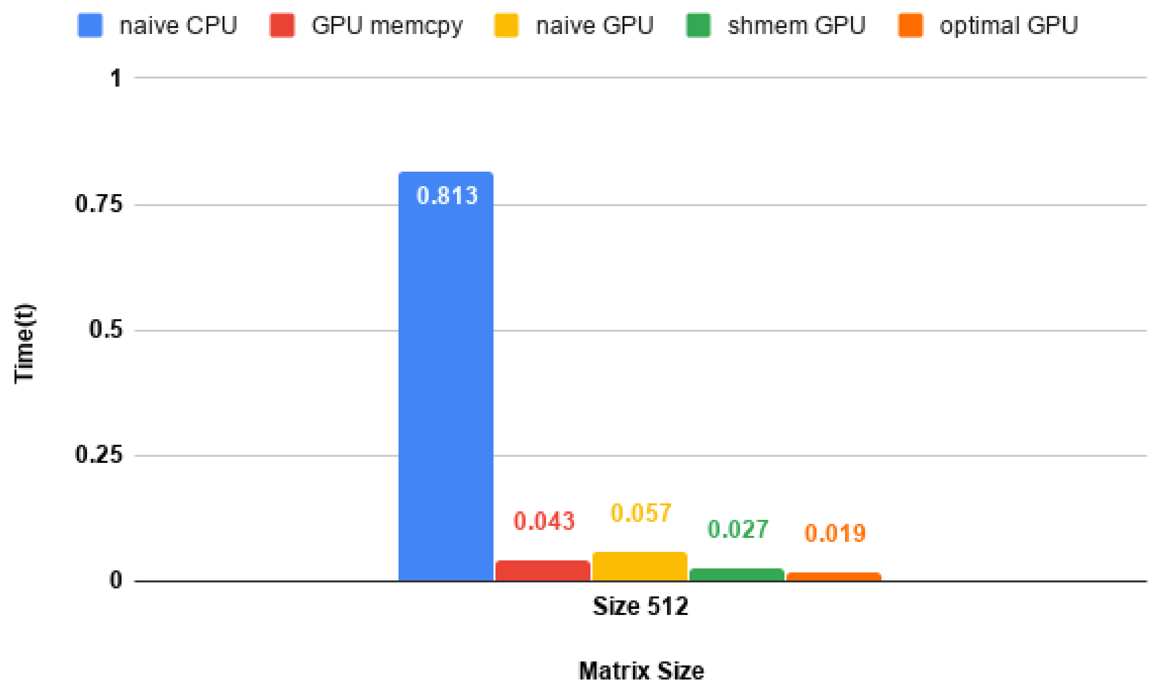


Fig 3. Time taken vs. implementations after performing optimization for matrix size 512

MATRIX SIZE 1024

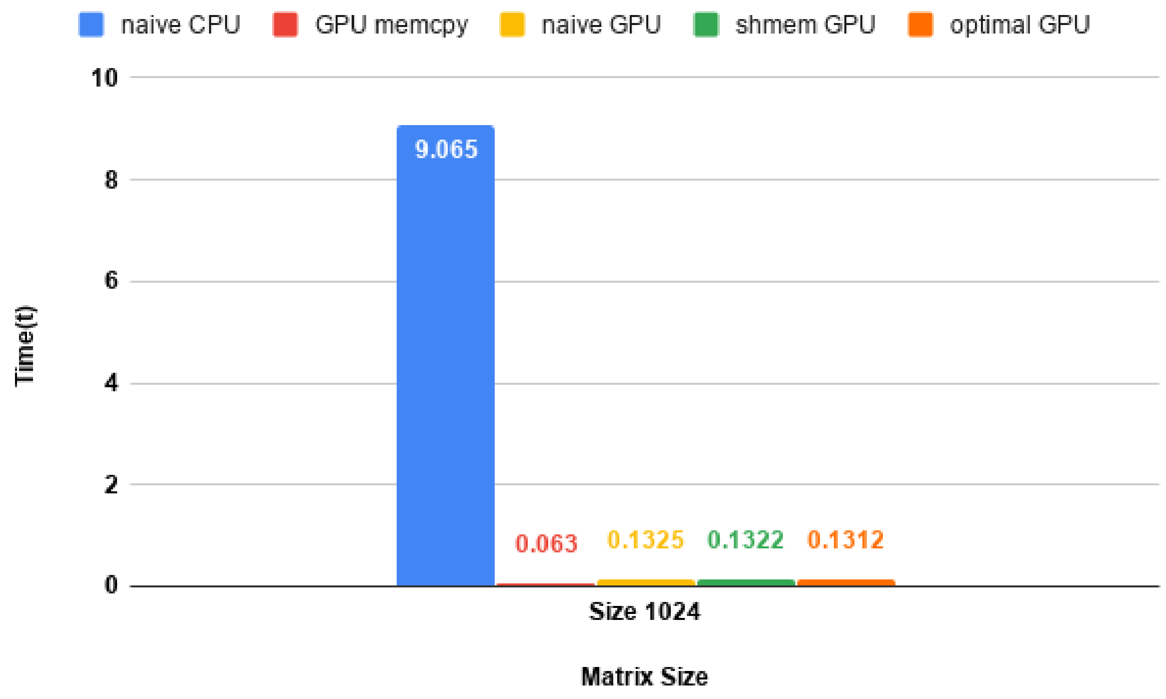


Fig 4. Time taken vs. implementations before performing optimization for matrix size 1024

MATRIX SIZE 2048

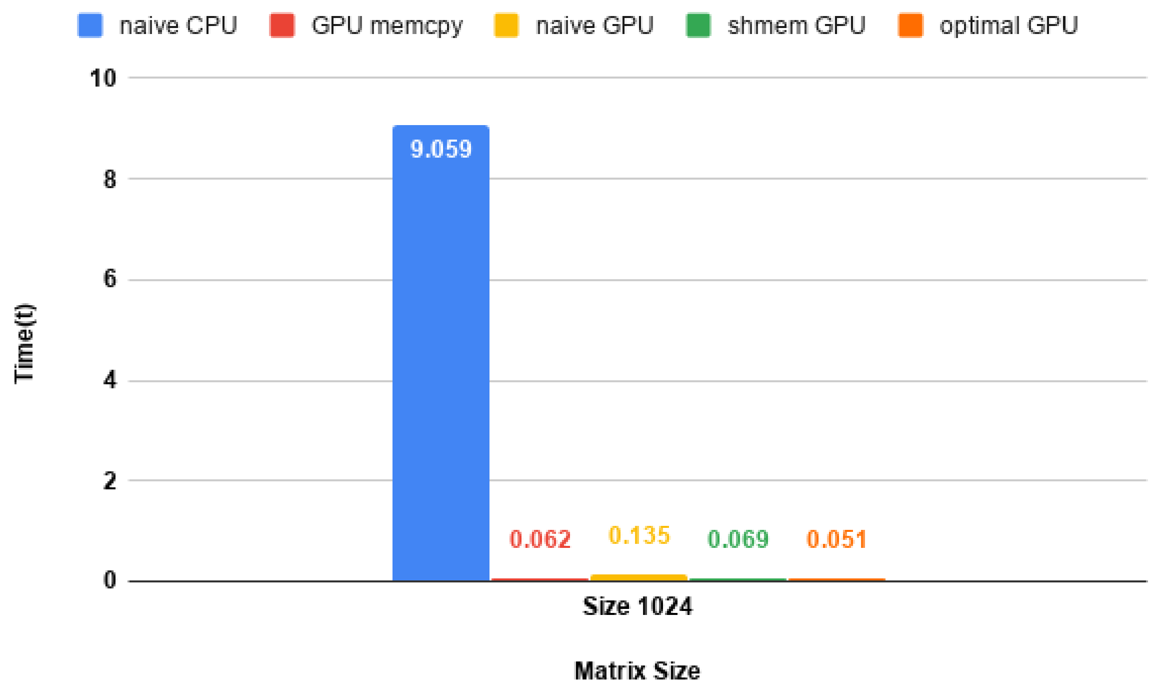


Fig 5. Time taken vs. implementations after performing optimization for matrix size 1024

MATRIX SIZE 2048

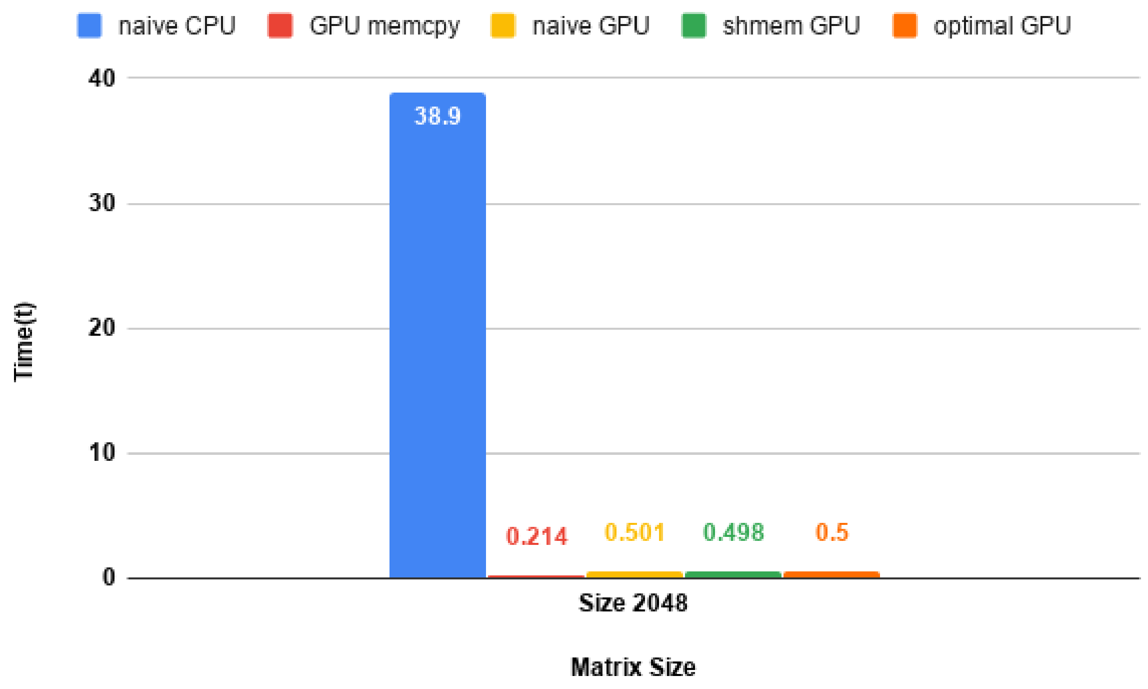


Fig 6. Time taken vs. implementations before performing optimization for matrix size 2048

MATRIX SIZE 2048

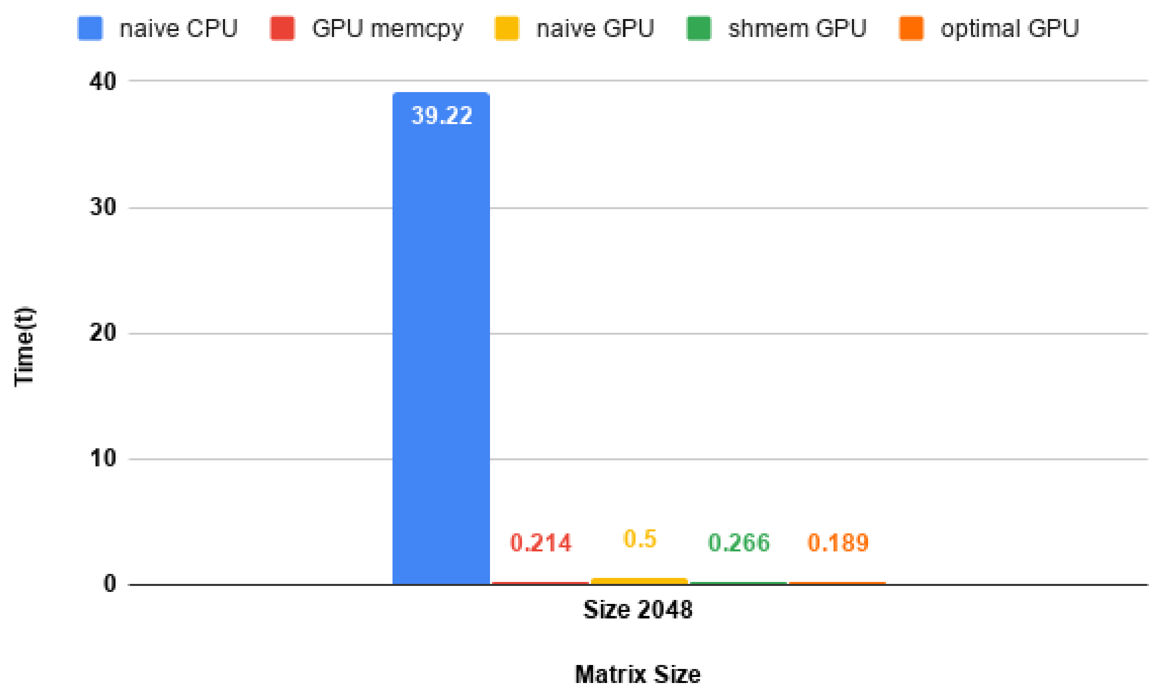


Fig 7. Time taken vs. implementations after performing optimization for matrix size 2048

MATRIX SIZE 4096

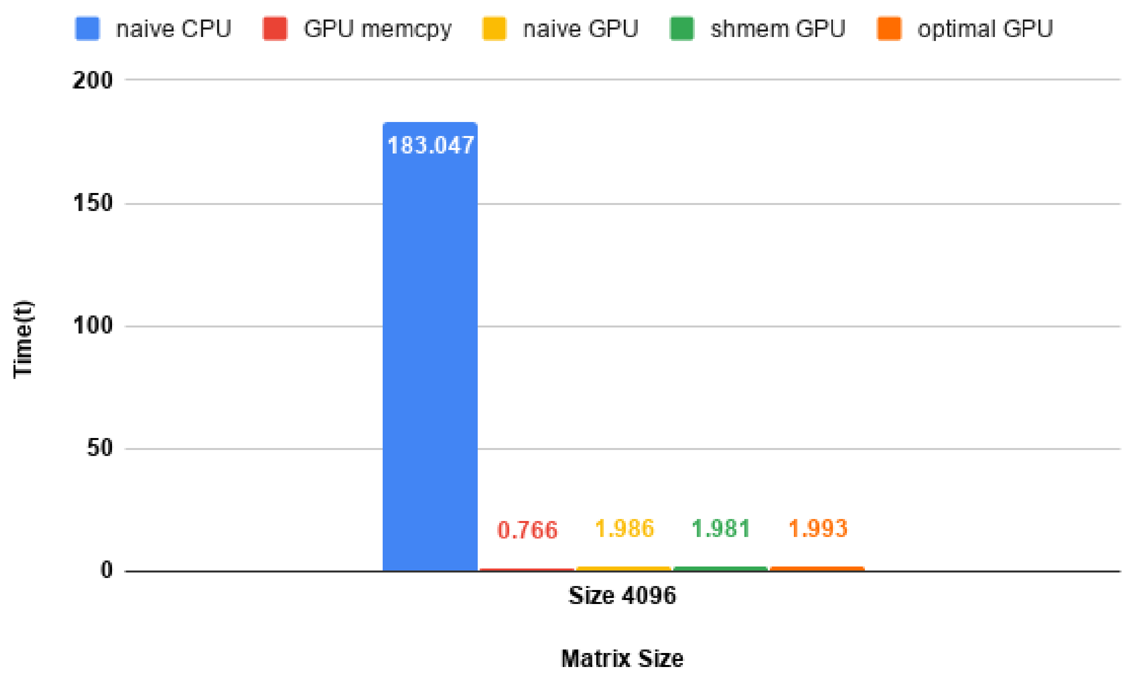


Fig 8. Time taken vs. implementations before performing optimization for matrix size 4096

MATRIX SIZE 4096

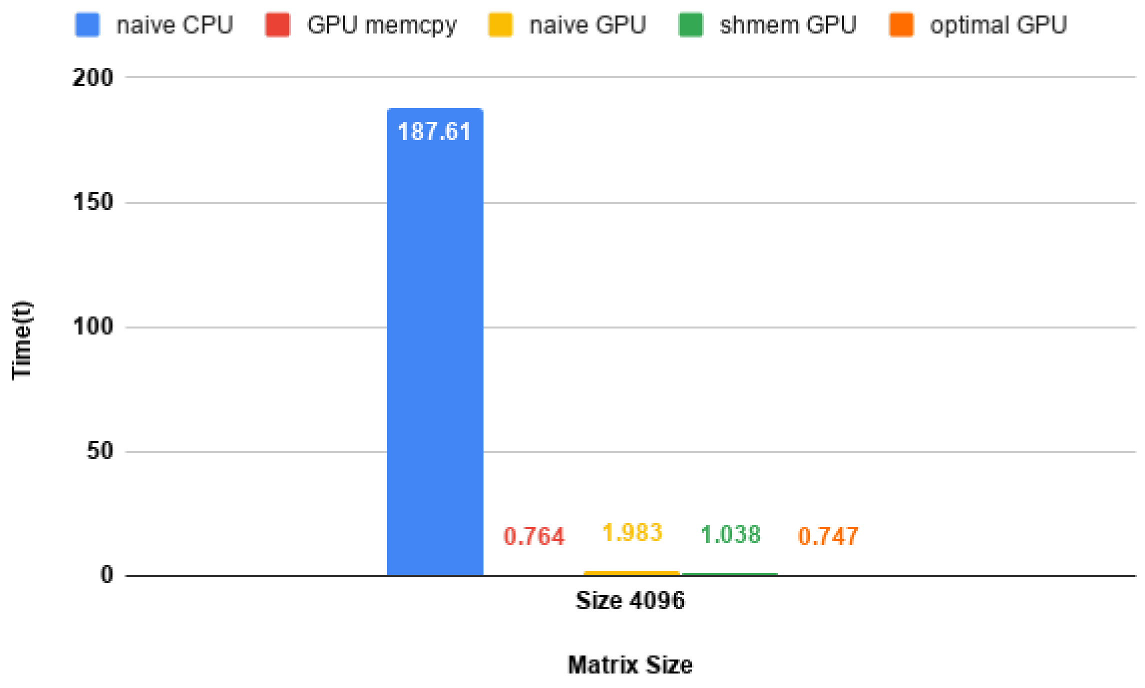


Fig 9. Time taken vs. implementations after performing optimization for matrix size 4096

PERFORMANCE IN TERMS OF GFLOP/s

FOR MATRIX SIZE 512

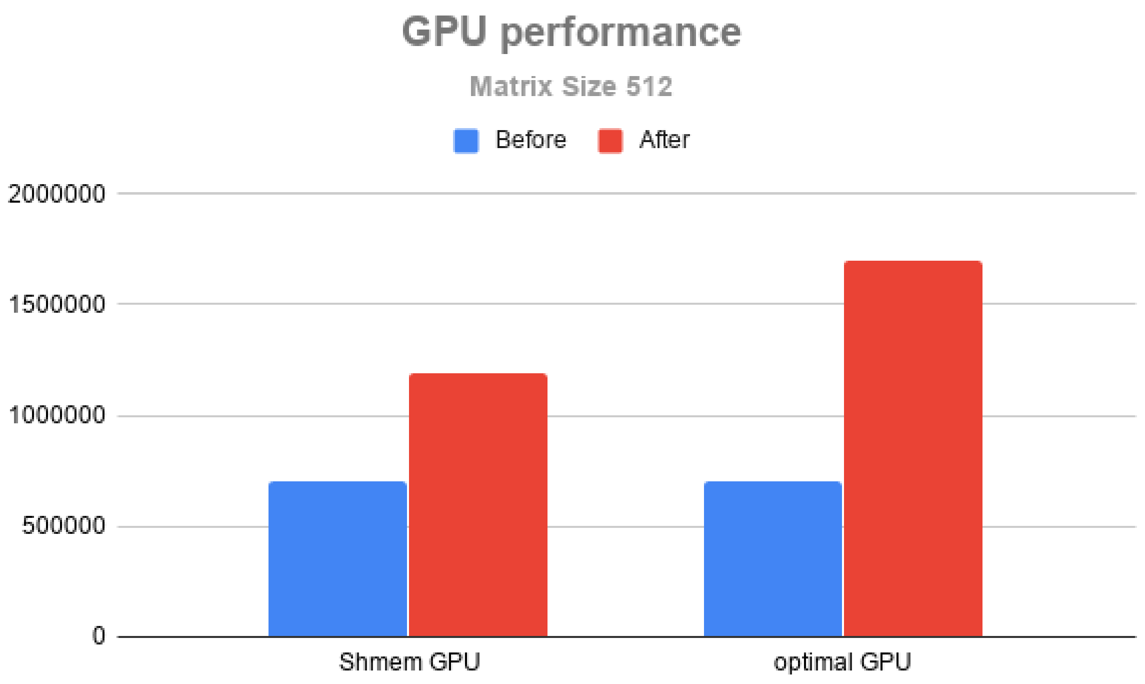


Fig 10: Performance in terms of GFlops for matrix size 512 before and after performing optimization for shared memory GPU and Optimal GPU

FOR MATRIX SIZE 1024

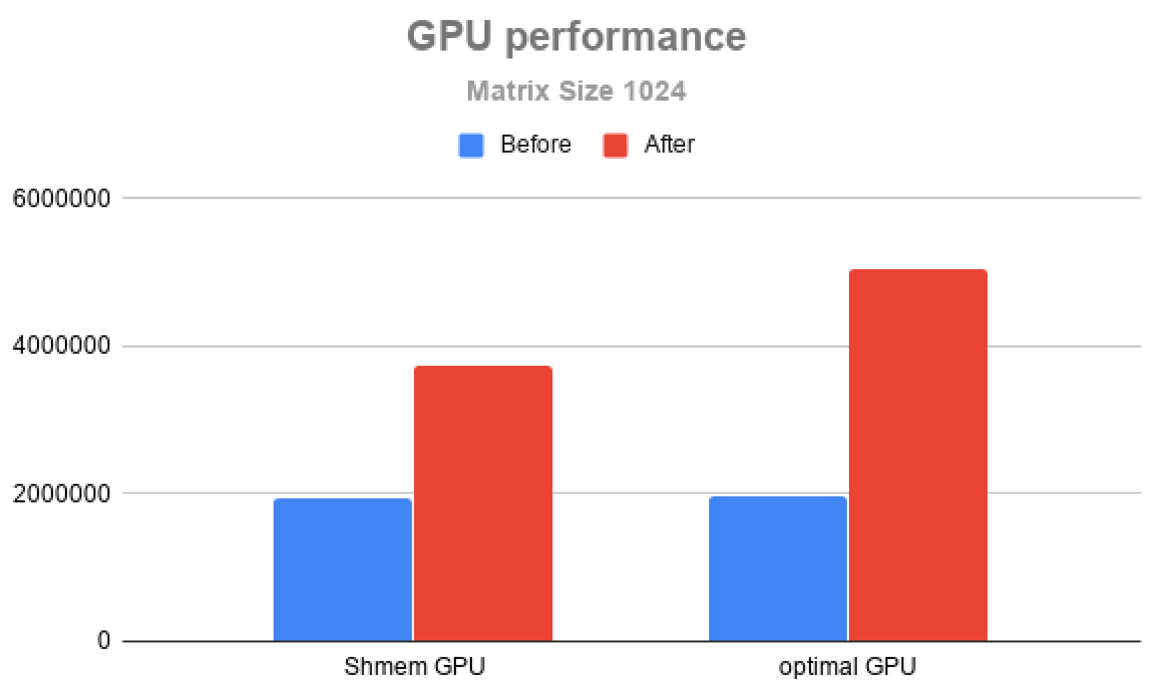


Fig 10: Performance in terms of GFlops for matrix size 1024 before and after performing optimization for shared memory GPU and Optimal GPU

FOR MATRIX SIZE 2048

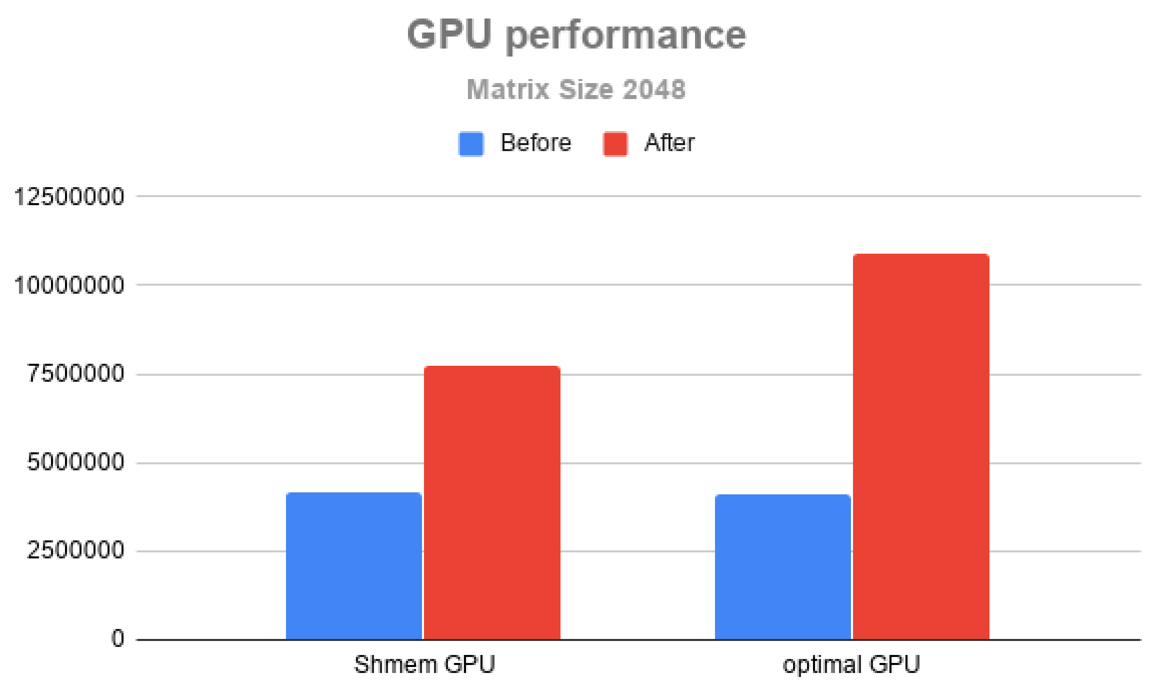


Fig 10: Performance in terms of GFlops for matrix size 2048 before and after performing optimization for shared memory GPU and Optimal GPU

FOR MATRIX SIZE 4096

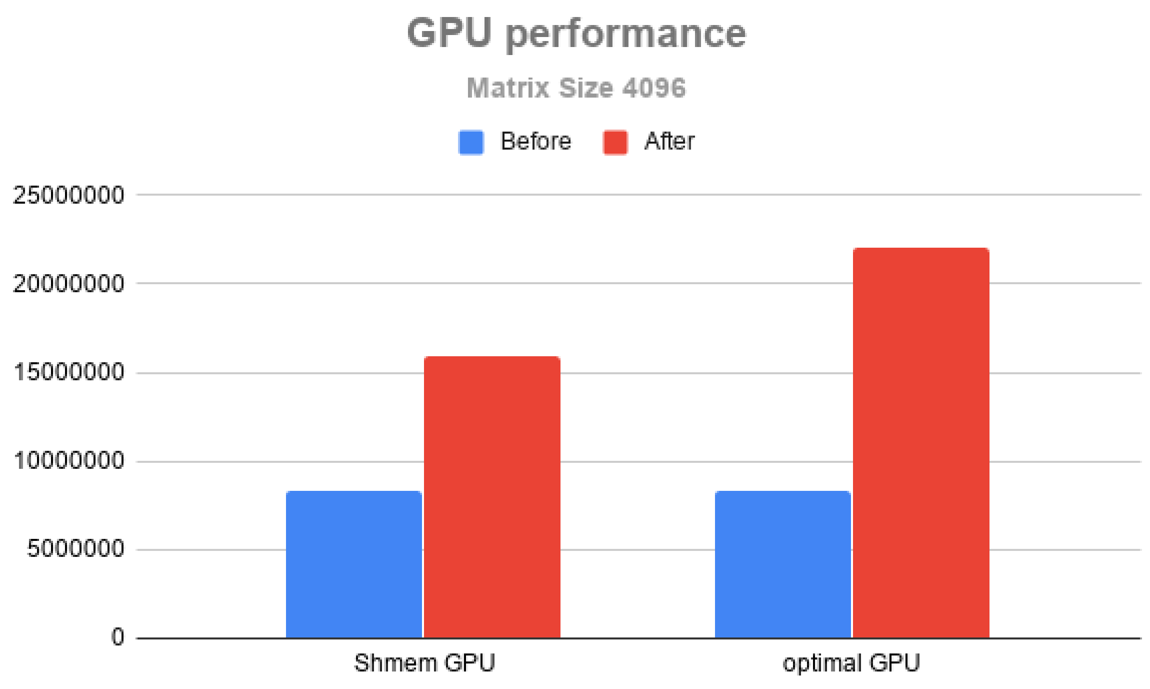


Fig 10: Performance in terms of GFlops for matrix size 4096 before and after performing optimization for shared memory GPU and Optimal GPU

This is how we calculated Gflop/s :

NumOps = 2 * pow(MatrixSIze, 3)
Gflops = e^{-9} * (NumOps/Execution Time)

Where e^{-9} = 0.00012

Instructions for building:

- module load cuda
- make
- sbatch job-transpose