GPU Computing 2nd deliverable

Ettore Saggiorato
Universita' di Trento
Trento, Italy
ettore.saggiorato@studenti.unitn.it

The code of this project is available on GitHub[1].

I. Problem Description

Develop an algorithm to transpose non-symmetric matrices of size $N \times N$, where N is a power of 2, for both single-threaded CPU and CUDA implementations. Evaluate the CPU implementation's performance by measuring the effective bandwidth under different compiler optimization levels (-O0, -O1, -O2, and -O3), and analyze the cache behavior to assess the implementation's efficiency. For the CUDA implementation, measure the effective bandwidth to evaluate its performance.

II. Algorithms and Kernels

The matrix data is defined as a 1D array of size $N \times N$, where the element at position i, j is stored at the index $i \times N + j$. The matrix is stored in row-major order[1]. The CUDA implementation uses a 2D grid of blocks and threads to transpose the matrix.

```
**let** result ← Matrix shape.y × shape.x
**for** $i$ ← 0 to shape.x **do:**
    **for** $j$ ← 0 to shape.y **do:**
        result[j][i] ← result[i][j]
**return** result
```

Algorithm 1: Simple Transposition

```
**let** result ← Matrix shape.y × shape.x
**let** $N$ ← shape.x
**let** $M$ ← shape.y
**for** (ii ← 0) **to** N **by** *block* **do:**
    **for** (jj ← 0) **to** M **by** *block* **do:**
        **for** ($i$ ← ii **to** min(ii+block, N)) **do:**
            **for** ($j$ ← jj **to** min(jj+block, M)) **do:**
                result[j][i] ← data[i][j]
**return** result
```

Algorithm 2: Block Matrix Transposition

```
**let** x ← blockIdx.x ⋆ tile_dim + threadIdx.x
**let** y ← blockIdx.y ⋆ tile_dim + threadIdx.y
**let** width ← gridDim.x ⋆ tile_dim[2]
**for** $i$ ← 0 **to** tile_dim **by** block_rows **do:**
    odata$[(y + i) * \text{width} + x]$ ← idata$[(y + i) * \text{width} + x]$
```

Algorithm 3: CUDA Copy

```
**for** $i$ ← 0 **to** tile_dim **by** block_rows **do:**
    odata$[\text{width} * x + (y + i)]$ = idata$[(y + i) * \text{width} + x]$;
```

Algorithm 4: CUDA Transpose Naive

```
**set** shared tile[tile_dim][tile_dim]
**for** $i$ ← 0 **to** tile_dim **by** block_rows **do:**
    tile$[\text{tIdx}.y + i][\text{tIdx}.x]$ ← idata$[(y + i) * \text{width} + x]$
syncthreads
**for** $i$ ← 0 **to** tile_dim **by** block_rows **do:**
    odata$[(y + i) * \text{width} + x]$ = tile$[\text{tIdx}.x][(\text{tIdx}.y + i)]$;
```

Algorithm 5: CUDA Transpose Shared

```
**set** shared tile[tile_dim][tile_dim+**1**]    // Same as CUDA Transpose Shared, but with padding to avoid bank conflicts
```

Algorithm 6: CUDA Transpose Shared No Conflicts

A. CPU Transposition

Algorithm 1 is straightforward and easy to implement. However, due to its implementation it is not cache-friendly, as it does not exploit the cache hierarchy.

Algorithm 2 divides the matrix into blocks of size block × block and transposes each block. This approach improves the cache behavior by reducing the number of cache misses.

B. CUDA Transposition

The performance of various transposition algorithms is evaluated using the Algorithm 3 kernel as a baseline. This straightforward implementation only utilizes global memory and exploits memory coalescing, meaning all threads read and

[1]matrix[i][j] notation is used instead of matrix$[i \times \text{shape.y} + j]$ due to space constraints and readability concerns.

[2]x,y and `width` are common in all CUDA implementations, they will be omitted in other pseudocodes.

write to contiguous memory locations. However, it does not leverage shared memory.

The Algorithm 4 is a naive transposition implementation that also uses only global memory. While it exploits memory coalescing during read operations, it does not do so during writes. The strided memory access pattern during writes results in suboptimal performance and poor effective bandwidth, as noted in [2].

The Algorithm 5 improves upon the naive implementation by using shared memory. It extracts a 2D tile of a multidimensional array from global memory in a coalesced fashion into shared memory, and then have contiguous threads stride through the shared memory tile, without losing any performance. After memory loading, the threads need to syncronize as they will read strided memory locations from shared memory. This is done so that all of them can write to global memory in a coalesced fashion. This approach removes any striding while accessing global memory and improves the effective bandwidth. However, it is still not optimal due to bank conflicts. Bank conflicts occur when multiple threads in a warp access different addresses within the same memory bank, causing serialized access and reducing throughput. The presence of bank conflicts can be detected using the CUDA profiler or through exclusion, as detailed in [3], but this analysis is omitted here for brevity.

Finally, Algorithm 6 includes padding in shared memory ensures that memory accesses within a warp are spread across different memory banks, preventing bank conflicts.

III. Performance metrics

On the cpu runs time is registered using the `std::chrono` library, while on CUDA with `cudaEvent`. The effective bandwidth is calculated as the ratio between the number of bytes read and written and the time taken to perform the operation using the formula:

$$\text{eb} = \text{effective bandwidth} = \frac{\text{Br} + \text{Bw}}{\text{t\_routine}} \quad (1)$$

That in the case of matrix transpose is equal to:

$$\text{eb} = \frac{\text{size}_A \times \text{size}_B \times \text{dtype}_{\text{bytes}}}{\text{time} \times 10^9} \quad (2)$$

GPU's theoretical bandwidth is calculated as:

$$\frac{\text{mem\_clock\_rate} \times 10^3 \times \left(\frac{\text{memory\_bus\_width}}{8}\right) \times 2}{10^9} \quad (3)$$

and is specified in the Table 3.

IV. Experimental setup

A. Hardware

The experiments were performed on a system equipped with a robust CPU cooler, ensuring that thermal throttling was effectively prevented. Table 1, Table 2, Table 3 detail the CPU, memory and graphics card specifications respectively.

Model	AMD Ryzen 5 5600X
Architecture	x86
Clock Speeds	3.7 GHz base, 4.6 GHz boost
Cache Levels	L1 384 KB, L2 3 MB, L3 32 MB
Cores, Threads	6, 12

Table 1: CPU Specs.

Type	DDR4
Size	16 GB
Speed	3200 MHz
Memory Channels	Dual Channel
DOCP/AMP/XMP	DOCP 3200 MHz

Table 2: RAM Specs.

Device 0	NVIDIA GeForce GTX 1060 6GB
CUDA Driver/Runtime	12.4 / 12.5
CUDA Capability	6.1
Total global memory	6043 MBytes
Multiprocessors, Cores/MP	10, 128, 1280 CUDA Cores
Memory Clock rate	4004 Mhz
Memory Bus Width	192-bit
Total shared memory per block	49152 bytes
Warp size	32
Max # of threads per block	1024
Max size of a thread block (x,y,z)	(1024, 1024, 64)
Theoretical memory bandwidth	192.2 GBps

Table 3: GPU Specs.[3]

V. CPU Runs

To improve the accuracy of the measurements and reduce the OS noise, each experiment was repeated multiple times, in function of the specific fun, to have ~1s run time, and the average value was calculated. The matrix size was varied from 2^1 to 2^{10}, and the block size was set to 16. All experiments were done with float data type. Results are presented in the next section.

[3]Informations from NVidia's deviceQuery tool [4].

A. *Performance Trends*

Taking a look at Figure 1 we can notice that:

- No optimization exhibits the worst performance, which is expected as it applies no optimization techniques.
- O1 optimization shows a similar curve to no optimization but with slightly better performance. Both methods perform comparably.
- O2 and O3 optimizations demonstrate a more significant improvement globally.
- The simple method outperforms the block method for small matrices ($\leq 2^6$), while the block method performs better for larger matrices. This difference persists despite similar cache hit and miss rates for both methods. This may be due to overhead of block management.

B. *Cache Comparison*

Comparing cache characteristics between O3 simple and block optimizations from Figure 3, Figure 4, and Figure 5:

- Both methods exhibit similar instruction, data, and unified cache behavior.
- However, there's a noticeable difference in cache misses, particularly in the D1 miss rate, with O3 simple at 3.3% and O3 block at 1.3% when $N = 2^8$.

1) ***Decrease in Cache References:*** From Figure 6 we can observe that as optimization levels increase, cache references decrease. This reduction is attributed to more efficient code resulting from higher optimization levels, leading to fewer memory accesses. This decrease is evident in D references. Note that the D1 miss rate increases as the matrix size increases and as the optimization level increases. This increase is primarily in the write cache, suggesting that prefetching may be beneficial.

2) ***Similar Cache Miss:*** For matrices with size $\leq 10^6$, all optimization levels exhibit similar cache miss behavior. However, differences become more pronounced for matrices with size $> 10^7$.

3) ***Performance Variation between Simple and Block Methods:*** The performance difference between the simple and block methods stems from various factors:

- Cache Line Utilization: The block method operates on smaller blocks that fit within a single cache line, leading to better cache line utilization. In contrast, the simple method may access larger contiguous regions, resulting in more cache misses.
- Spatial Locality: The block method exhibits better spatial locality as it accesses data in a more localized manner within each block. Conversely, the simple method may display poorer spatial locality, especially for larger matrices.

Overall, the difference in miss rates between the simple and block methods for larger matrices is a combination of cache line utilization, spatial locality, and loop optimizations.

VI. CUDA

The GPU runs were repeated 100 times to minimize OS and system noise. The grid size was calculated dynamically based on the matrix size, while the block size was kept static. Multiple block dimensions were tested: $[16 \times 4]$, $[16 \times 8]$, $[16 \times 16]$, $[32 \times 4]$, $[32 \times 8]$, $[32 \times 16]$, to assess kernel sensitivity to block size. Figure 7 illustrates the effective bandwidth performance of the different kernels, demonstrating their sensitivity to block size. Additional plots are available in the repository. The naive implementation, due to its lack of shared memory usage, is less affected by block size variations.

The speed-up relative to Algorithm 2 is approximately an order of magnitude, and about 4 times faster compared to Algorithm 4. This highlights the significant performance gains achievable with GPUs for parallelizable matrix operations.

When comparing Algorithm 5 and Algorithm 6, it is evident that bank conflicts significantly impacted the performance of the former kernel.

VII. Future work

When comparing multiple matrix sizes, the shared implementation without bank conflicts proves to be faster than a simple copy kernel without shared memory. Introducing a comparison with a shared-memory copy kernel would enhance this analysis.

Further investigation into the `diagonal block reordering` kernel proposed by NVidia[5] could provide valuable insights. A comparison with existing implementations and an exploration of optimal usage scenarios would be beneficial.

All implementations discussed here spawn $\frac{\text{shape.x}}{\text{TILE\_DIM}} \times \frac{\text{shape.y}}{\text{TILE\_DIM}}$ blocks, each with $\text{TILE\_DIM} \times \text{BLOCK\_ROWS}$ threads. However, if the number of threads exceeds the maximum concurrently running threads of the GPU, performance may suffer due to queuing. Therefore, exploring a kernel that maximizes the utilization of concurrent threads, evenly distributes tasks among threads, maintains coalesced memory access, and avoids shared memory bank conflicts could be interesting.

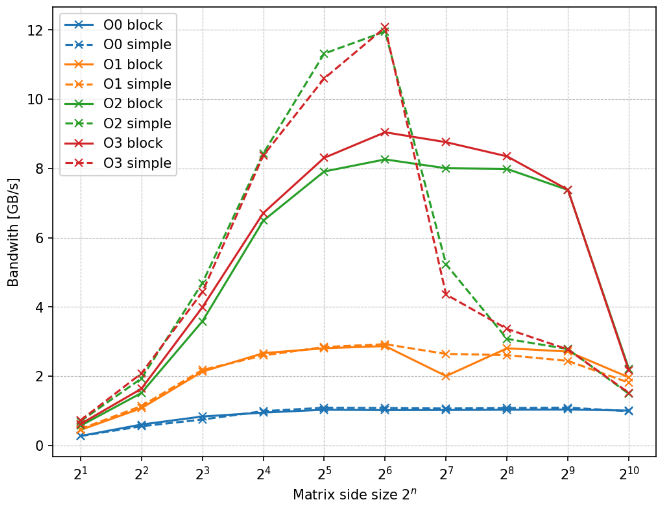

Figure 1: CPU - Effective bandwidth (float)

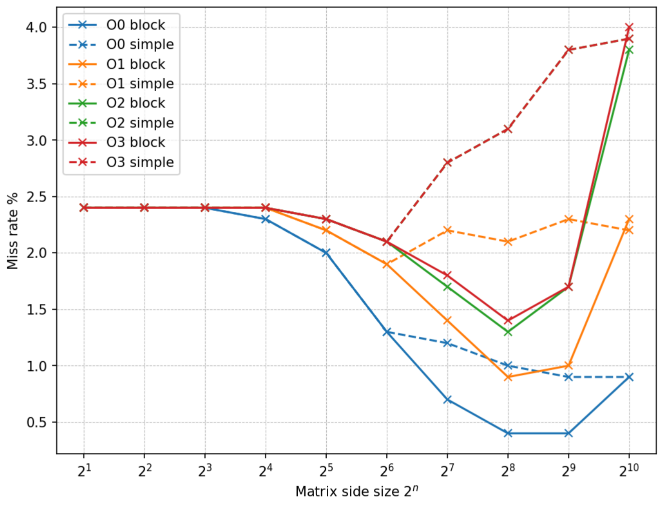

Figure 4: CPU - D1 miss rate

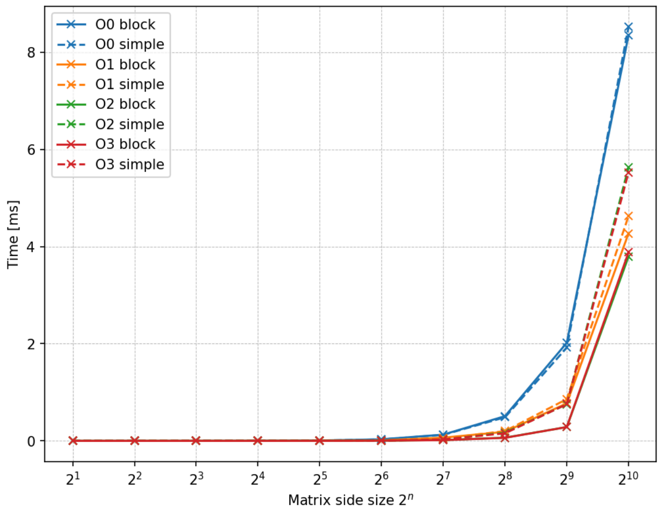

Figure 2: CPU - Execution time (float)

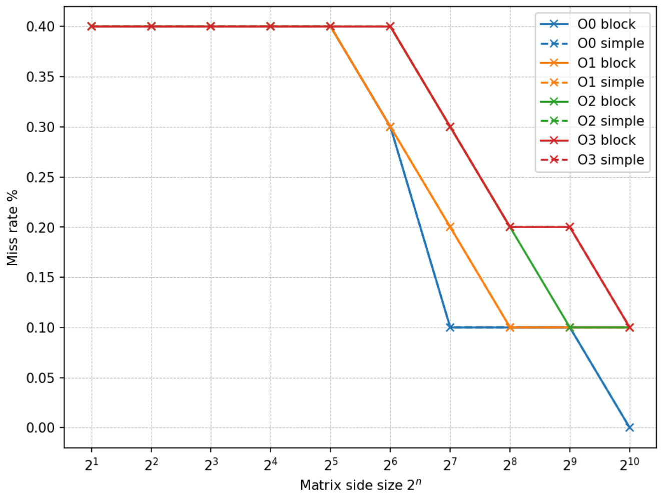

Figure 5: CPU - LL miss rate

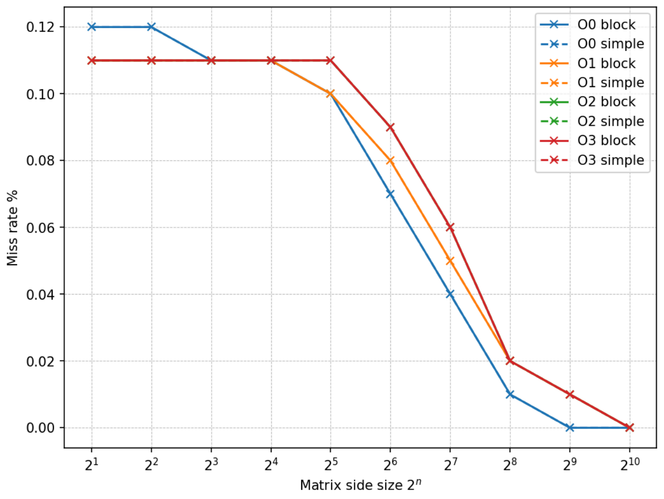

Figure 3: CPU - L1 miss rate

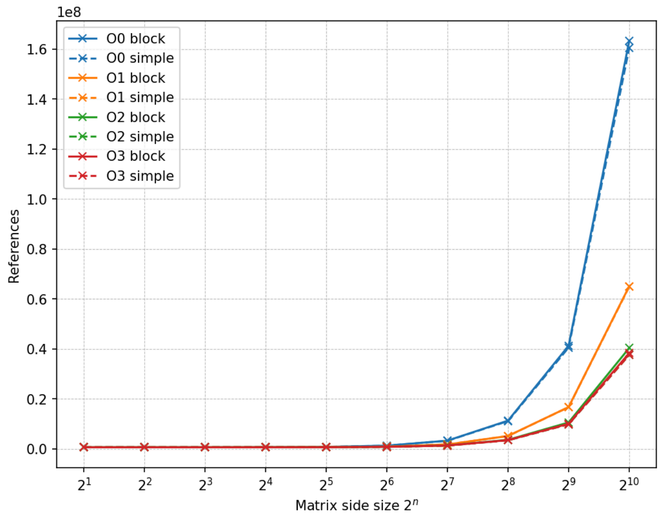

Figure 6: CPU - D references

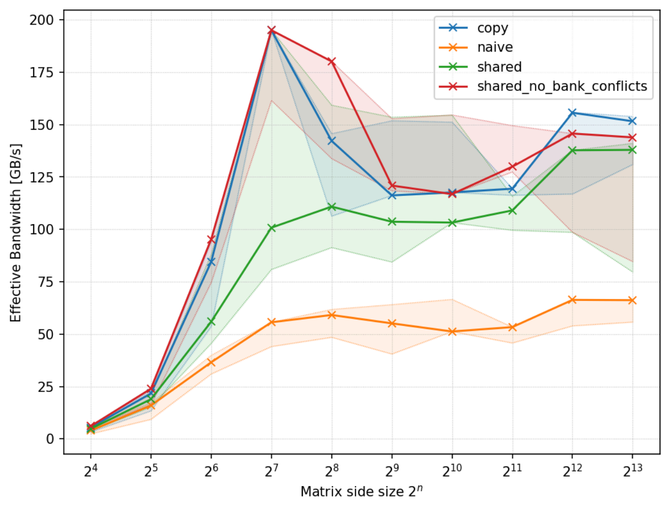

Figure 7: CUDA - Effective bandwidth - Performance of matrix transpose implementations (copy, naive, shared, shared_no_bank_conflicts). Shaded areas show the min-max range from multiple grid size dimensions, with size 32x8 highlighted.

References

[1] S. - Ettore Saggiorato, “GPU Computing.” [Online]. Available: https://github.com/sa1g/gpu-computing

[2] M. Harris, “How to Access Global Memory Efficiently in CUDA C/C++ Kernels.” [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[3] M. Harris, “An Efficient Matrix transpose in CUDA C/C++.” [Online]. Available: https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/

[4] N. Corporation, “CUDA Samples - Device Query.” [Online]. Available: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/1_Utilities/deviceQuery

[5] G. Ruetsch and P. Micikevicius, “Optimizing Matrix Transpose in CUDA.” [Online]. Available: https://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf