GPU Computing
Homework 2: Matrix Transposition with CUDA

Murtas Cristian
248025
cristian.murtas@studenti.unitn.it
GitHub Repository

Abstract

This report shows the results obtained by the implementation of different algorithms for matrix transposition on a GPU using CUDA. The first algorithm is a naive implementation that uses global memory to store the matrix and performs the transposition. The second algorithm uses shared memory to reduce the number of global memory accesses. Lastly, the third algorithm uses shared memory and avoid bank conflicts. These algorithms have been tested using different kernel configurations to find the best parameters. DIRE QUALCOSA SUI RISULTATI.

1. Problem Description

The goal of this homework is to implement different algorithms to transpose non-symmetric matrices of size $N \times N$. The transpose of a matrix is an operation that flips the matrix over its diagonal [4]. The program should use an input parameter to set the size of the matrix, for instance, **transpose 12** will work on a $2^{12} \times 2^{12}$ matrix. We are also asked to find the best kernel configuration in terms of grid and block size, and to compare the performance of the different algorithms, evaluating the effective bandwidth.

2. CUDA Algorithms

2.1. Naive Transposition

In the *NaiveTranspose* algorithm, we simply read the element at position (row, col) from the source matrix and write it to the destination matrix at position (col, row). Note that while the accesses to the *SRC* matrix are coalesced, the accesses to the *DST* matrix are not, indeed the writes to *DST* have a stride equal to the width of the matrix.

Algorithm 1 Naive Matrix Transposition

```
1: procedure NAIVETRANSPOSE(src, dst, width)
2:     row ← blockIdx.x × blockDim.x + threadIdx.x
3:     col ← blockIdx.y × blockDim.y + threadIdx.y
4:     if row ≤ width && col ≤ width then
5:         dst[row × width + col] = src[col × width + row]
6:     end if
7: end procedure
```

2.2. Transposition with Coalesced Memory

This algorithm exploits the usage of the shared memory to store a tile of the matrix, avoiding the large strides trough the global memory.

Algorithm 2 Matrix Transpose with Shared Memory

```
1: procedure TR_COALESCED(src, dst, width)
2:     tile[TILE_DIM][TILE_DIM]
3:     row ← blockIdx.x × TILE_DIM + threadIdx.x
4:     col ← blockIdx.y × TILE_DIM + threadIdx.y
5:     if row ≤ width && col ≤ width then ▷ copy to shared memory
6:         tile[tIdx.y][tIdx.x] = src[col × width + row]
7:     end if
8:     __syncthreads()
9:     row ← blockIdx.x × TILE_DIM + threadIdx.x
10:    col ← blockIdx.y × TILE_DIM + threadIdx.y
11:    if row ≤ width && col ≤ width then
12:        dst[col × width + row] = tile[tIdx.y][tIdx.x]
13:    end if
14: end procedure
```

2.3. Transposition with Coalesced Memory and No Bank Conflicts

A wrong access pattern to the shared memory can lead to bank conflicts, which can slow down the execution of the kernel. To avoid this conflict we can pad the shared memory with an additional column.

Algorithm 3 Matrix Transpose with Shared Memory with no Bank Conflicts

1: **procedure** TR_COALESCED($src, dst, width$)
2: $tile[TILE\_DIM][TILE\_DIM+1]$
3: $row \leftarrow blockIdx.x \times TILE\_DIM + threadIdx.x$
4: $col \leftarrow blockIdx.y \times TILE\_DIM + threadIdx.y$
5: **if** $row \leq width$ && $col \leq width$ **then** ▷ copy to shared memory
6: $tile[tIdx.y][tIdx.x] = src[col \times width + row]$
7: **end if**
8: $\_\_syncthreads()$
9: $row \leftarrow blockIdx.x \times TILE\_DIM + threadIdx.x$
10: $col \leftarrow blockIdx.y \times TILE\_DIM + threadIdx.y$
11: **if** $row \leq width$ && $col \leq width$ **then**
12: $dst[col \times width + row] = tile[tIdx.y][tIdx.x]$
13: **end if**
14: **end procedure**

3. Experimental Setup

The entire experiment has been conducted on the Marzola cluster of the University of Trento, which is equipped with NVIDIA A30 GPUs [3]. Furthermore, the code has also been tested on a local machine equipped with an NVIDIA RTX 3070 [1]. For both setup, the information about the GPUs have been retrieved using the *cudaGetDeviceProperties* function [2].

4. Results

The algorithms have been first tested with a matrix of size 1024×1024 and a fixed kernel configuration with 4096 blocks each containing 256 threads. The results, shown in table 1, clearly highlight how in this case the access pattern matters. Indeed, having a global coalesced access is a key point to achieve a good performance.

Effective Bandwidth (GB/s)		
Algorithm	RTX 3070	A30
Naive	280.49	285.14
Coalesced	372.41	523.79
Coalesced no bank conflicts	386.32	601.03

Table 1. Effective Bandwidth of the different algorithms

5. Conclusions

References

[1] GigaByte. Rtx 3070 specifications. https://www.gigabyte.com/Graphics-Card/GV-N3070GAMING-OC-8GD-rev-20/sp#sp, 2020.

[2] NVIDIA. Cuda documentation: cudagetdeviceproperties. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE_1g1bf9d625a931d657e08db2b4391170f0, 2020.

[3] NVIDIA. Nvidia a30 datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/products/a30-gpu/pdf/a30-datasheet.pdf, 2022.

[4] Wikipedia. Transpose. https://en.wikipedia.org/wiki/Transpose.