Matrix transposition tests

For GPU Computing Homework 2

1st Emmanuele Virginio Coppola

emmanuele.coppola@studenti.unitn.it 247540

***Abstract*—A continuation of the work done in the first deliverable**

I. Introduction

The objective of this report is to analyze the performance of a sequential algorithm for matrix transposition and to explore the potential improvements achievable through the use of parallel algorithms. Project available at: https://github.com/EmmanuelEngineer/GpuComputing

II. Theory

As defined in the previous report, the transposition of a matrix is a simple problem but can be implemented in various ways. In the previous report, we analyzed sequential algorithms for performing a transposition.

In this report, to transpose the matrix, we can use algorithms that leverage natively parallel hardware such as graphics cards. Graphics cards are specialized hardware for matrix manipulation. They feature a hierarchy of components, from streaming multiprocessors and warps to a specialized memory hierarchy from global memory to shared memory and caches. There are various algorithms that exploit this particular architecture, and we will focus:

1) Naive Tile algorithm
2) transposed coalesced

The Naive algorithm is a simple algorithm that, similar to block alghorithm of the previous report, takes a tile from global memory and simply transposes it directly in memory without further steps.

Algorithm 1 Naive Tile Alghorithm

```
$A$ ←Input Matrix
$B$ ← Output Matrix
$x$ and $y$ element of the Matrix
$width$ ← matrix width $j$ starts from 0 until $j$ minor than tile dimension $j$ increases
$B[x * width + (y + j)] \leftarrow A[(y + j) * width + x]$
```

=0

The transposed coalesced algorithm addresses the issue of accessing non-contiguous data. If we examine two elements of the same matrix that have the same x position but different y positions, although they are logically adjacent elements, they are actually separated by a ”row width” in memory, leading to multiple request global memory accesses. This because for each requested data, global memory provides a fixed-size memory row of adjacent elements to reduce further requests. If the provided row is smaller than the distance in memory space between the two data elements, the memory will be accessed twice. To mitigate this problem, the transposed coalesced algorithm first copies all possible data into what is defined as shared memory and then only later copies the transposed data back to memory.

Algorithm 2 Transpose Coalesced

```
$A$ ←Input Matrix
$B$ ← Output Matrix
$T$ ← Temporary Matrix
$k$ ← A fixed value    ▷ This value is given when calling the alghorithm
$z$ ← A fixed value    ▷ This value is given when calling the alghorithm
$x$ and $y$ element of the Matrix
$width$ ← matrix width $j$ starts from 0 until $j$ minor than tile dimension $j$ increases
$T[z + j][k] \leftarrow A[(y + j) * width + x]$
$j$ starts from 0 until $j$ minor than tile dimension $j$ increases
$B[(y + j) * width + x] \leftarrow T[k][z + j]$
```

The algorithm maximizes the use of a multi-core architecture by assigning specific memory portions to each multiprocessor, which in Nvidia's software abstraction are defined as blocks.

III. Methodology and Experimental Setup

A. Equipment

The test were done on 2 machines with:

- Same GPU vendor (Nvidia)
- Same cuda Runtime version(12.1)

One of the machines is one node of the Universitá di Trento cluster ”Marzola” using one Nvidia A30. The other machine is a MSI laptop with a mobile RTX 4070 locked at the maximum 90w of power consumption. The key differences in these two machines use for this study are listed in the following table:

Given the task of a matrix transpose the expected total bandwidth for each of these 2 machines are:

- 64 GB/s = $\frac{2000 \times 2 \times 128}{8 \times 10^3}$ for the 4070-M
- $933,12$ GB/s = $\frac{1215 \times 2 \times 3072}{8 \times 10^3}$ for the A30

	4070-M	A30
Global Memory (MB)	7933	24169
Cuda Cores	4608	3584
Memory Clock (Mhz)	2000*	1215
Total memory Bandwidth**	128-bit	3072-bit
L2 Cache size (MB)	33	25
Warp size	32	32

TABLE I
DATA OBTAINED WITH THE NVIDIA PROVIDED TEST CODE SAMPLES
***is the only value that was not obtained from cuda device prop because it returns 8000 as a value**
***** this value is given as a sum of the bandiwidth of all memory controllers***

B. Implementation detail

A problem that can occur when using this specific architecture and implementing the use of shared memory is the occurrence of specific cases of bank conflicts. In Nvidia's architecture, certain tile sizes in shared memory can lead to continuous bank conflicts when accessing memory. One solution applied to avoid this problem is to use an algorithm where the shared memory, i.e., the temporary memory discussed in the theory section of the report, is of size tile by tile + 1.

C. Golden Standard

To provide a reference for the implementation, each test included a comparison with a simple matrix copy. Therefore, all the schemes presented in the following sections will have the maximum y limit as the maximum measure obtained in the copy.

D. Experiment procedure

As in the previous report, a bash file was used to perform the experiments, which automatically iterates over the testing parameters and then saves the results. These results are then analyzed using a Python script, which also enables the visualization of graphs.

The only differences are:

- each datapoint is the result of the average of 100 tests
- the dimension of the tile, is added as a parameter, it can vary between 16,32 and 64

IV. RESULTS

In the test the roof line represents the maximum bandwidth obtained from the golden-test i.e. the copy. As we can see from Fig.1 for the resulting value obtained from the 4070 is much bigger than the calculated theoretical maximum of 64 GB/s. Probably there is some other value that was wrongly reported in the code sample, since for the Marzola tests the value are lower than the theoretical maximum (as expected) Fig.3. Even if the calculation are cannot be trusted completely, we can in Fig.2 see that there are some recognizable behavior in relation between each other. In fact as we can see that for the integer datatype there is a strong decline just after the 2^{11} mark and for the double datatype, at the same point we are already well below $50GB/s$. This is caused from the filling of the L2 cache since $2^{11} * 2^{11} * 8$ bytes is well over the 32 MB of L2 cache.

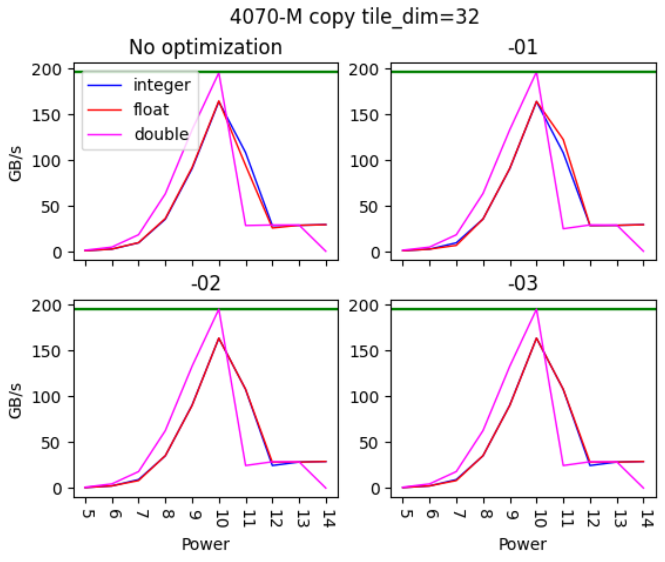

Fig. 1. Enter Caption

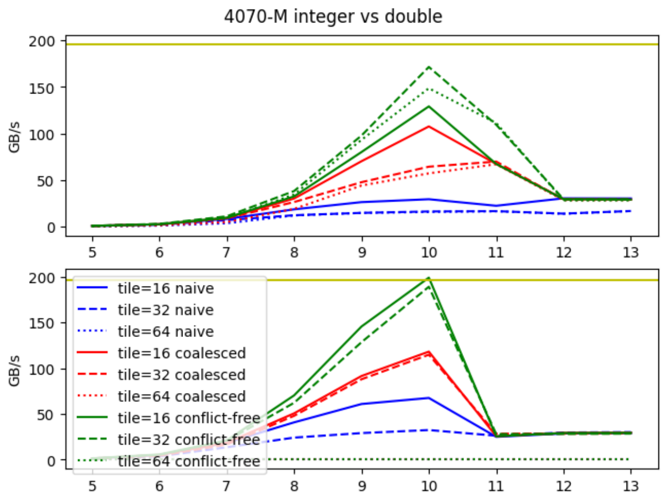

Fig. 2. Global test on the different types of transpose operation. Up is used the integer datatype, down the double

Another interesting behavior is the effect of the different matrix Transpose algorithms showing the difference in performance and the impact of the tile dimension. We can see that for most of the algorithms bigger tiles does not usually correspond to a higher bandwidth. This an be easily explained from an hardware perspective. In the GPU architecture we have a strong hierarchy between the processing components and the memory. The implementation of these algorithms divides the matrix in tiles giving each tile to each sub-unit of the GPU streaming which has at it's bottom layer the so called warps that contains the cores. Having a tile dimension of 32 we have that each tile is processed only by one Warp only. Plus that Warp is fully saturated because since each core can move one datum at a time and we have to avoid bank conflicts.

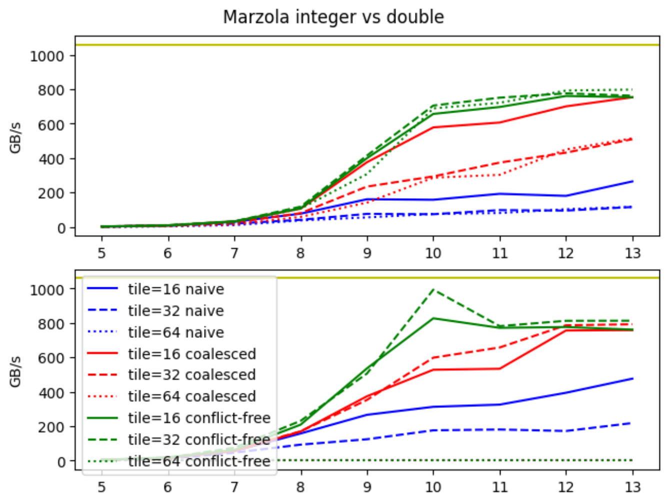

Fig. 3. A30 result scores

V. Conclusions

As we see in the 3 graph(since is more reliable) we can see that we have a pretty high bandwidth at the platou of the results, but there is at least a 20% of bandwidth to reach. From an alghorithm stand point we could try to reach that peaks trying to further optimize bank conflicts and so on, or if we can, we could use a multi GPU approach trying to put each GPU in the optimal condition like in the 2^{10} situation.

References

[1] By https://developer.nvidia.com/blog/author/mharris/Mark Harris,An Efficient Matrix Transpose in CUDA C/C++ https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/