GPU computing - Homework 1

Luca Cazzola - 248716 - luca.cazzola-1@studenti.unitn.it

April 2024

1 Problem description

It is asked to implement a simple algorithm able to compute the **transpose** of a given non-symmetrical squared matrix. Consider an arbitrary matrix $X^{[n,m]}$ having n rows and m columns, the transpose $t(X)$ is a matrix $Y^{[m,n]}$ such that it's columns are the X's rows.

1.1 Problem assessment

Algorithm 1 - Naive matrix transposition

```
Input: X
parameters: matrix-size                          ▷ matrix-size = n or m
for i = 0 to matrix-size do                      ▷ parse over rows
    for j = i + 1 to matrix-size do              ▷ parse over columns
        swap X(i, j) with X(j, i)                ▷ swap elements
    end for
end for
return X                                         ▷ return transposed matrix
```

Note: Minor changes have been applied to the pseudo-code for better visualization and understanding

X being squared implies $t(X)$ is equivalent to computing the symmetry along the main diagonal, for this reason elements on the main diagonal don't need to be moved. The provided solution has time complexity of $O(n^2 - n)$ and I believe it's not possible to find a lower one since all elements of the matrix needs to be parsed (a different approach might be 3.1). That being said there's room for improvement during runtime if the **cache behaviour** is taken into account :

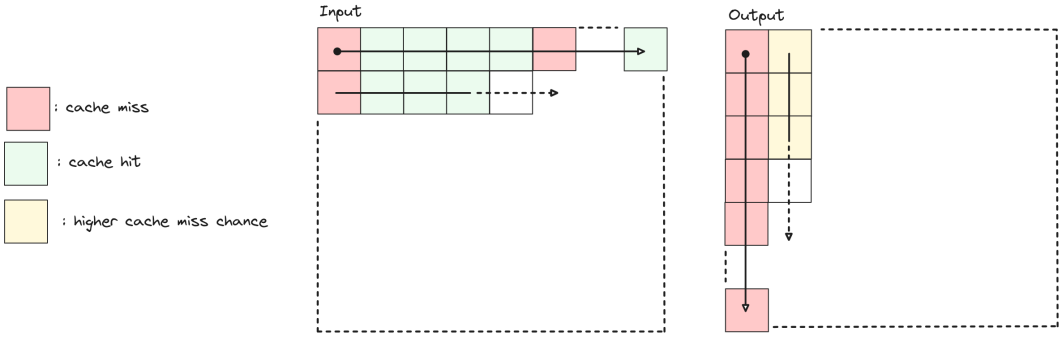


Figure 1: The cache perspective

Parsing element by element following the row-column order can be not so efficient for the cache because C follows **row-major ordering**. Taken into account the case of a matrix: element $X(i, j)$ and $X(i+1, j)$ can be found in consecutive memory addresses, while $X(i, j)$ and $X(i, j+1)$ are actually 1 row of elements (in bytes) apart. Each time a read operation is performed on the main memory a **cache line** of **consecutive** elements is moved in cache instead, to make future accesses faster.

Taking into account the matrix transposition problem and assuming for simplicity that read and write operations are done on 2 separate matrices X and Y what happens is that :

- **While reading X** : reading the first element will result into a miss since the matrix has never been accessed. Next accesses will be faster due to caching until the cache line is exhausted.
- **While writing Y** : the entire first column will consist of cache misses since elements are stored in row-major order and not column-major. When parsing the second column elements might still be available in cache, but that's less likely to happen because in the mean time the entire first column has been parsed.

Following the **spatial and temporal locality principles**, which generally lead to a more efficient cache usage the following implementation is proposed :

Algorithm 2 - Block-based matrix transposition

```
Input: X
parameters: matrix-size block-size      ▷ block-size = same as matrix-size, but related to blocks
blocks-per-row = matrix-size / block-size
for diagonal = 0 to blocks-per-row do                    ▷ parse over the main diagonal blocks first
    for i = 0 to block-size do
        for j = i + 1 to block-size do
            row-index = i + (diagonal * block-size)
            column-index = j + (diagonal * block-size)
            swap X(row-index, column-index) with X(column-index, row-index)
        end for
    end for
end for

for block-row = 0 to block-per-row do                    ▷ parse the rest of the matrix
    for block-column = block-row +1 to block-per-row do
        for i = 0 to block-size do
            for j = 0 to block-size do
                row-index = i + (row-index * block-size)
                column-index = j + (column-index * block-size)
                swap X(row-index, column-index) with X(column-index, row-index)
            end for
        end for
    end for
end for

return X                                                 ▷ return transposed matrix
```

Note: Minor changes have been applied to the pseudo-code for better visualization and understanding

X is divided into **blocks** of defined size and operations (swaps in our case) are performed more locally within elements belonging to symmetrical blocks (with respect to the main diagonal). This allows a more localized job into the space and time domains. Blocks belonging to the main diagonal are evaluated first respect to the others because they represent a sub-case, in which swaps are performed inside a single block (and not a pair of blocks). Blocks are still accessed in row-column order after the diagonal is parsed.

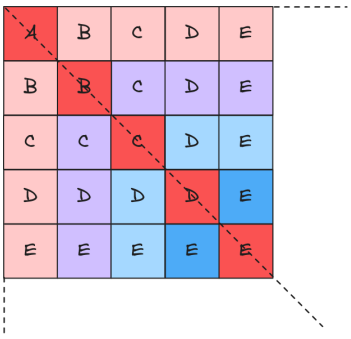


Figure 2: Visual : block swapping

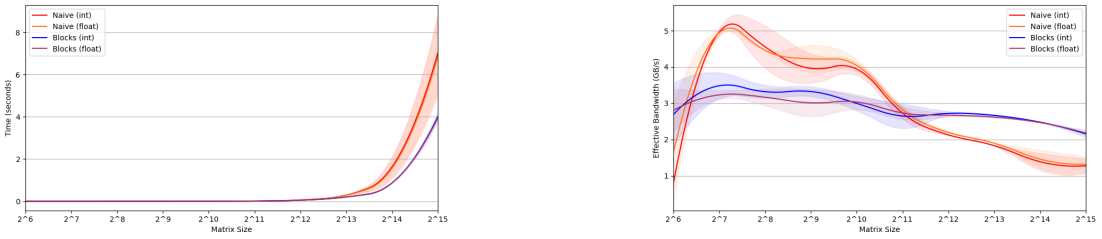
Swaps are performed between blocks with matching color and letter

2 Experimental setup and analysis - CPU

Performance evaluation has been done on my own device. Here follows some of the hardware and software specifics:

- **Notebook** : Dell XPS 15 7590
- **CPU** : Intel i7-9750H, 2.6GHz base - 4.5GHz max, cores : 6
- **RAM** : 2 × 8GB SODIMM DDR4 - 2667 MT/s
- **Cache** : 64 Bytes cache line
  - **L1** : 384 kB - 8-way Set-associative
  - **L2** : 1536 kB - 4-way Set-associative
  - **L3** : 12288 kB - 16-way Set-associative
- **OS** : Ubuntu 20.04

2.1 Performance analysis



(a) Execution time

(b) Effective bandwidth

Figure 3: Naive vs. Blocks transposition

The two algorithms have been evaluated on matrices of exponentially increasing size, taking into account both int32 and float32 data type. Block size is set to 2^4 since into a cache line of 64 bytes at most 16 consecutive int32 or float32 elements can fit. The graphs clearly show a **not significant difference in handling the two data types**. It's also evident that the Naive approach is more time consuming as the matrix grows. The block based approach also shows a more stable effective

bandwidth, sign of a overall better memory usage. To prove what's been stated in 1.1 regarding the cache contribution to this boost in performances here follows the related analysis :

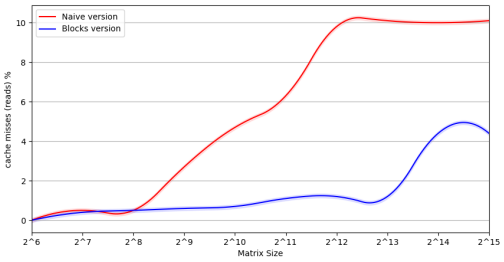


Figure 4: Naive vs. Blocks - cache miss (Data)

By running a **Cachegrind simulation** on the two algorithms it's evident how the Naive version starts suffering in performances right after the matrix size 2^8 step, which is the moment X starts to **fill L1 cache size** :

- L1 size = 384 kB $\sim$ 96k float32 elements - capacity exceeded in between 2^8 and 2^9 steps
  Note : remember that "matrix size" refers to the number of elements in a row/column.
  Ex: At step 2^9, X contains 2^{18} elements.

The Naive approach then stabilizes upon surpassing the 2^{11} step, which again correspond to the filling of both L1 and L2 cache levels.

- L1 + L2 size = $384 + 1536 = 1920$ kB $\sim$ 480k float32 elements - capacity exceeded in between 2^{11} and 2^{12} steps

Performance drops are of course present in the block based approach, but occurs later due to better usage of the cache. One might argue that the cache miss % could be biased due to some differences in the algorithms implementations, especially related to the 2 additional for loops present in the block version which interfere in the Cachegrind counting of total reads

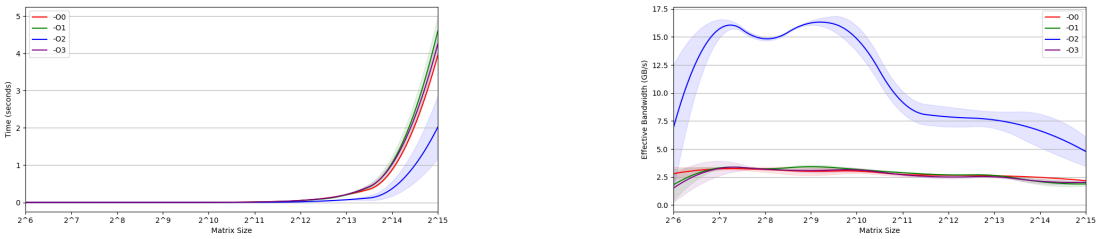
$$\text{total read misses } \% = \frac{\text{L1 read misses} + \text{L2 read misses}}{\text{total reads}} * 100$$

Total reads becomes higher due to just presenting more variables updates in the Blocks version respect to the Naive one. Anyway this can be excluded noticing that regardless of the total number of memory reads the Blocks version always performs significantly fewer misses.

Matrix Size	**NAIVE**		**BLOCKS**	
	L1 Read Misses	**L2 Read Misses**	**L1 Read Misses**	**L2 Read Misses**
$\mathbf{2^6}$	11	0	9	0
$\mathbf{2^7}$	1,031	0	1,196	0
$\mathbf{2^8}$	4,219	0	5,111	0
$\mathbf{2^9}$	79,552	0	25,752	0
$\mathbf{2^{10}}$	549,915	0	110,919	0
$\mathbf{2^{11}}$	2,355,172	565,518	445,457	248,956
$\mathbf{2^{12}}$	9,533,753	8,569,073	1,750,886	1,390,138
$\mathbf{2^{13}}$	39,382,289	35,564,330	6,825,333	5,850,780
$\mathbf{2^{14}}$	158,896,368	138,295,179	152,459,246	24,930,779
$\mathbf{2^{15}}$	637,092,856	566,952,987	609,925,648	100,592,536

Table 1: Counting cache misses - Naive vs. Blocks

All the previously shown data was gathered using no compiler optimization level -O0. Let's see if performances changes increasing the optimization level :



(a) Execution time

(b) Effective bandwidth

Figure 5: Compiler optimization levels comparison (Block version)

For such a small program the increasing compile time is not a problem at all. Performance wise it's evident how the optimization level -O2 offers the most benefit to the Blocks versions. Perhaps the more aggressive **loop optimization** and **data prefetching** (which are the most relevant optimization features to improve on such a problem) provided by -O2 have a better balance between simplicity and aggressiveness with respect to -O0, -O1 and -O3.

3 Parallelization and future directions

The Block version of matrix transposition would really benefit from **parallel computation** since each pair of blocks can be evaluated in a totally independent way respect to another. This will surely be the first thing I'll try when moving the problem on the GPU.

3.1 Alternative designs

Another way I've tried to approach the problem was by instead of physically swapping each element to just set the addresses of X's first row as pointers of Y. This would be a $O(n)$ time complexity solution, which on the other hand would require later to access Y always with n-striding access pattern, which is not beneficial at all from the cache perspective since Y is physically still structured as X was (row-major) but accesses in a different pattern. Led to that reasoning I didn't further develop the idea.

GPU computing - Homework 2

Luca Cazzola - 248716 - luca.cazzola-1@studenti.unitn.it

June 2024

4 Moving on the GPU

Algorithm 3 - Block-based matrix transposition on GPU

Input: matrix X (on device)
parameters: SIZE (size of X's side), BLK_SIZE (size of block in X to parse)

define 2 blocks on **shared memory** A and B of size BLK_SIZE^2

$xA = \text{blockIdx.x} \times \text{BLK\_SIZE} + \text{threadIdx.x}$ ▷ x,y base offset w.r.t. X of an element in A
$yA = \text{blockIdx.y} \times \text{BLK\_SIZE} + \text{threadIdx.y}$
$xB = \text{blockIdx.y} \times \text{BLK\_SIZE} + \text{threadIdx.x}$ ▷ x,y base offset w.r.t. X of an element in B
$yB = \text{blockIdx.x} \times \text{BLK\_SIZE} + \text{threadIdx.y}$

if blockIdx.x > blockIdx.y **then** ▷ parse upper non-diagonal block
  for $j = 0$ **to** BLK_SIZE **step** blockDim.y **do** ▷ copy to shared memory
    for $i = 0$ **to** BLK_SIZE **step** blockDim.x **do**
      $A[(\text{threadIdx.y} + j) \cdot \text{BLK\_SIZE} + \text{threadIdx.x} + i] = X[(yA + j) \cdot \text{SIZE} + xA + i]$
      $B[(\text{threadIdx.y} + j) \cdot \text{BLK\_SIZE} + \text{threadIdx.x} + i] = X[(yB + j) \cdot \text{SIZE} + xB + i]$
    end for
  end for
  __syncthreads() ▷ wait for A and B to fill
  for $j = 0$ **to** BLK_SIZE **step** blockDim.y **do** ▷ write back to global coalesced
    for $i = 0$ **to** BLK_SIZE **step** blockDim.x **do**
      $X[(yA + j) \cdot \text{SIZE} + xA + i] = B[(\text{threadIdx.x} + i) \cdot \text{BLK\_SIZE} + \text{threadIdx.y} + j]$
      $X[(yB + j) \cdot \text{SIZE} + xB + i] = A[(\text{threadIdx.x} + i) \cdot \text{BLK\_SIZE} + \text{threadIdx.y} + j]$
    end for
  end for
else if blockIdx.x == blockIdx.y **then** ▷ parse diagonal block
  same schema as non-diagonal case using only one block among A or B
end if

return X ▷ return transposed matrix

Note: Minor changes have been applied to the pseudo-code for better visualization and understanding

This is more or less a 1 to 1 translation of the block-based matrix transposition described in (1.1), but implemented with **CUDA**. The most important aspect of this solution is the usage of **shared memory** which is ×100 faster than the global memory (on no-cached accesses) and enables **coalesced writes** (on global). Each thread block is in charge of copying 2 symmetrical blocks from X (one per side w.r.t the main diagonal) inside the shared memory, referred as A (superior block) and B (inferior block). Once the copy is completed for both blocks each thread block proceeds copying A and B inside X following column major ordering on reads and row major on writes. Notice also that write addresses on X of A and B are swapped.

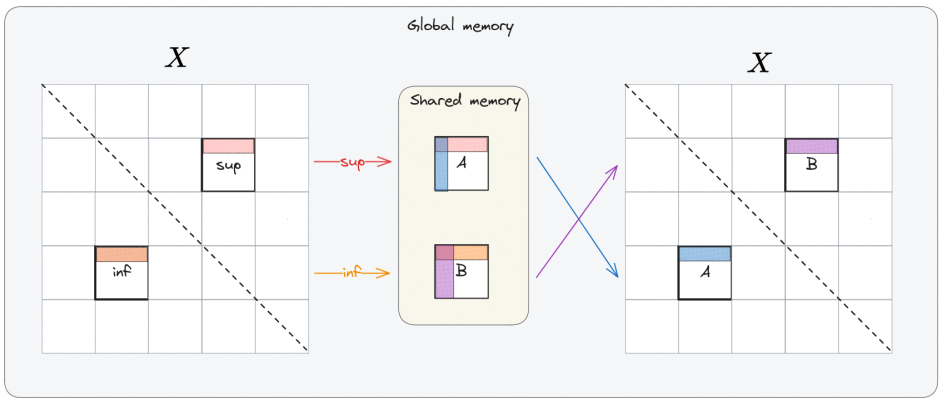


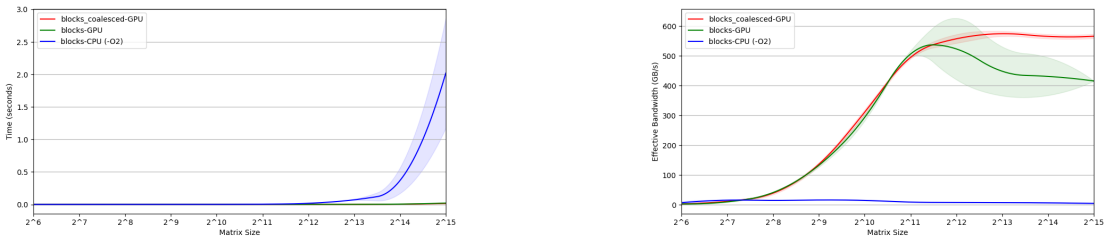
Figure 6: block-based transpose on GPU

5 Experimental setup and analysis - GPU

Performance evaluation has been done on the **University of Trento DISI department cluster**, which is equipped with many GPUs. For this experiment I've used a **NVIDIA A30**. here follows some of the card's specifics (relevant to the analysis):

- **architecture** : Ampere — 8.0 compute capability
- **stream multiprocessors** : 56 — 64 CUDA Cores/SM
- **Global mem. size** : 24062 MBytes
- **L2 cache size** : 25165824 bytes
- **Shared mem.** : 167936 bytes/SM — 49152 bytes/block
- **Memory bandwidth** : 933 GB/s

5.1 Performance analysis



(a) Execution time

(b) Effective bandwidth

Figure 7: blocks version (CPU) vs. blocks-naive (GPU) vs. blocks-coalesced (GPU) matrix transpose

Comparing the CPU and GPU versions is basically meaningless for large enough matrices since values are expressed in totally different scales. Resulting curves are flat one w.r.t the other in both plots and the recorded increment in performances for a $2^{15} \times 2^{15}$ matrix is $\sim 120\times$.

For reference, in addition to the kernel proposed in (4) I've implemented another one which does the same job, but without the use of shared memory. Both solutions effective bandwidth linearly increase up until the 2^{11} step, after that the non-coalesced version (green) drops in performances and becomes more unstable as the L2 cache gets saturated :

- 2^{11} step means that $2^{22} \cdot 4 = 16777216$ bytes allocated $\implies$ L2 cache is filled in the next step as $2^{24} \cdot 4 = 67108864 >$ L2 cache size (5).

This drop doesn't occur in the coalesced version, as the global memory accesses don't penalize as much. The coalesced version seems to have reached a plateau after surpassing the 2^{12} step, while the non-coalesced one is in a decreasing trend which is getting more and more stable.

Kernel	**% of Theoretical BW Peak**
Block-based **naive**	$44.53\% \pm 0.1\%$
Block-based **coalesced**	$60.61\% \pm 0.7\%$
$+16.08\%$ achieved with the application of shared memory	

Table 2: Performance at the 2^{15} matrix size step.

5.2 Parameters

The above data have been obtained by running kernels with the following setting :

- **grid dim.** : always large enough so that the entire matrix is covered (further discussed at 6) .
- **thread block dim.** : 8×32
- **matrix block size** : 32×32

That's the parameter setting which provided me the best results considering that the matrix block size can't exceed 64×64 since maximum 49152 bytes of shared memory can be allocated per thread block. Sure fact is that keeping the (thread block dim. $<$ block size) is beneficial as not all threads in a block can run in parallel, which makes less costy to just shift the thread with an index instead of switching the context of many threads [1]. I've tried squared thread blocks which performed fine and rectangular thread blocks such as 32×8 which performed significantly worse, but I've always came beck to the 8×32 setting. My guess to this fact is that such a shape is more beneficial from an L1 caching perspective, as elements are cached sooner.

6 Conclusion and future directions

I'm aware that one choice which might be a limiting factor to my implementation (4) is choosing to keep the grid size always as big as needed ($\frac{\text{matrix size}}{\text{block size}}$). That's because especially for bigger matrices it might be more efficient to make each thread block parse additional X's blocks and avoid the allocation of too many thread blocks. It's also true that such an implementation would need the addition of 2 more outer for loops, so I preferred to keep things simple and didn't really explore that possibility.

Another thing to note is that my implementation works **in-place** on the input matrix, without the need of initializing further memory (as in the CPU version 1.1). This design choice is beneficial from a memory perspective at the cost of requiring more runtime (worsening effective bandwidth), as once X is allocated there's no need to initialize twice the memory to store the result.

Bank conflicts-wise this application is safe, as by default devices with compute capability $\geq 3.x$ have bank sizes of 4 bytes [2]. Since this application works with float32 values and each thread processes its data independently there's no risk.

One direction I'd take if I had to further expand this project is to look at finer grain, reasoning on what could improve at **warp level**.

6.1 Code

Code available on GitHub

References

[1] Mark Harris. An efficient matrix transpose in cuda c/c++. https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/.

[2] Mark Harris. Using shared memory in cuda c/c++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.