Matrix transposition tests

For GPU Computing Homework 1

Emmanuele V. Coppola[a]

[a] *emmanuele.coppola@studenti.unitn.it 247540*

April, 2024

UNIVERSITÀ DI TRENTO

1. Introduction

The objective of this report is to analyze the cache performance and the CPU bandwidth for matrix transposition to assess the impact of cache and compiler optimization. Link of the relative github

2. Theory

The matrix transposition is a simple problem of taking elements of a matrix and change the position of it's elements by inverting the index of columns and rows. This simple algorithm can be used to test the performance of a given system as analogue for artificial intelligence matrix operations. Thanks to this properties the tests done on this algorithm will bring bring analogue results on real applications.

One of the several test done is the use of different access patterns like the Block matrix access pattern in respect to the standard linear access pattern, with the difference as shown in the pseudo code below.

```
#A is the matrix to transpose
def Transpose_Matrix_Linear(A,n):
    B = [n][n] # new matrix n*n
    for i=0 until n:
        for h=0 until n:
            B[i][h] = A[h][i]
    return B

def
   Transpose_Matrix_Block(A,n,block_dim):
    B = [n][n]
    for i=0 until n:
        for h=0 until n:
            end_k = (i+1) + block_dim #
               stopping indexes for the
               block
            end_z = (h+1) + block_dim
            for k=0 until end_k or n:
                for z=0 until end_z or
                   n:
                    B[i][h] = A[h][i]

    return B
```

Other variables to tweak to analyze the performance of the algorithm are:

- change datatype(in this report are tested the float, the double and the integer datatype)
- change the optimization algorithm used by the compiler
- test the same algorithm on different hardware

3. Methodology and Experimental Setup

This study employed two laptops with the following specifications:

- Operating System: GNU/Linux distribution Pop!OS
- Kernel Version: 6.8.0
- GCC Compiler: Version 11.4
- Variations in CPU and memory architecture

The first device is an HP laptop equipped with 8GB of DDR4 RAM operating at 2133 MHz on a 64-bit bus. This setup results in a theoretical maximum memory transfer bandwidth of 34.128 GB/s = $\frac{2133 \times 2 \times 64}{8 \times 10^3}$. The laptop is powered by an Intel Core i3-7100U processor, which includes a three-tier cache architecture: 128KiB L1 unified cache, 512KiB L2 unified cache, and 3MiB L3 unified cache.

The second device is an MSI laptop with 32GB of dual-channel DDR5 memory, running at 5600

MHz on a 64-bit bus. The theoretical maximum transfer bandwidth here is 179.2 GB/s $= \frac{5600\times2\times2\times64}{8\times10^3}$. It features an Intel Core i7-13700H processor, which supports a complex cache architecture tailored to its 14 cores—comprising 6 Performance Cores and 8 Efficiency Cores. For this study, we focus solely on the Performance Cores cache: 256KiB for data (L1), 512KiB for instructions (L1), 4MiB unified (L2), and 24MiB mixed use (L3).

The experimental procedures involve implementing the test algorithm in C. A bash script was utilized to execute various combinations of algorithmic optimizations and data type three times, with matrices of different dimensions and either with or without cache analysis via the cachegrind tool (version 3.22). For the block access pattern was used a block size of 8.

The execution time was determined by capturing the CPU clock cycles at both the beginning and the end of the main loop's execution. This method specifically excluded the cycles spent on memory allocation and result printing to ensure accuracy. Knowing the CPU's clock cycles per second, I then calculated the actual execution time.

For the MSI laptop, tests were conducted using matrices up to 15x15 in dimension. In contrast, the HP laptop was tested with matrices no larger than 10x10.

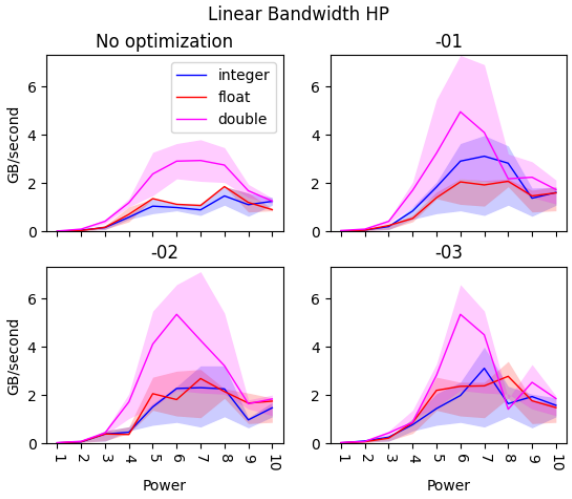

Figure 1: 4 charts one for each optimization flag used by the compiler, each test is done 3 times, for each power of 2 used for the matrix. The colored shadow represent the maximum and minimum value in the tests, the line is the sum of the 2 , halved

4. Results

For all subsequent analyses, we will use the double data type as the default. This decision is based on the observations from Figure 1, where the effects of compiler optimizations and the block algorithm are not only more evident but also exhibit similar trends.

Upon analyzing the data, it becomes clear that for both computers, the actual bandwidth falls significantly short of the RAM's maximum theoretical transfer speed. Notably, the combination of the "-O3" optimization flag and the block access pattern yielded the highest performance improvement. This enhancement was particularly pronounced in the MSI Laptop, as demonstrated in Figure 3.

Infobox **4-1**

The bandwidth was calculated with the following formula, considering 1 read and 1 write:

$$\frac{(2^{\text{power}} \cdot 2^{\text{power}}) \cdot 2 \cdot \text{data type dimension}}{\text{seconds of execution} \cdot 10^9}$$

The bandwidth on both laptops increases and reaches its peak at matrix dimensions between $2^6 *$

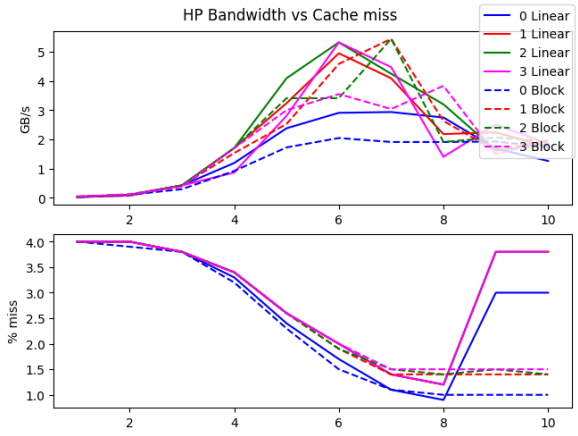

Figure 2: Comparison between bandwidth and first layer cache misses of the HP Laptop in respect of the power of 2 of the matrix dimension

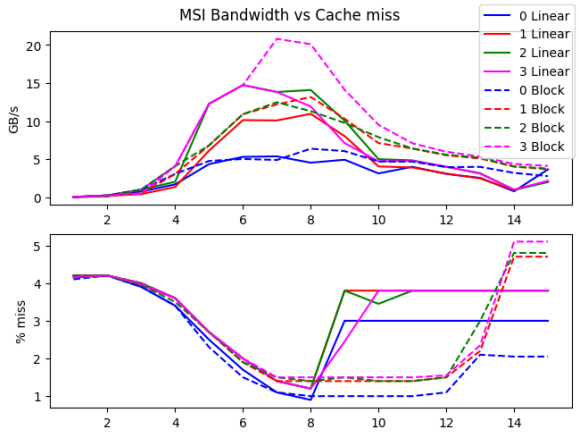

Figure 3: Comparison between bandwidth and first layer cache misses of the MSI Laptop in respect of the power of 2 of the matrix dimension

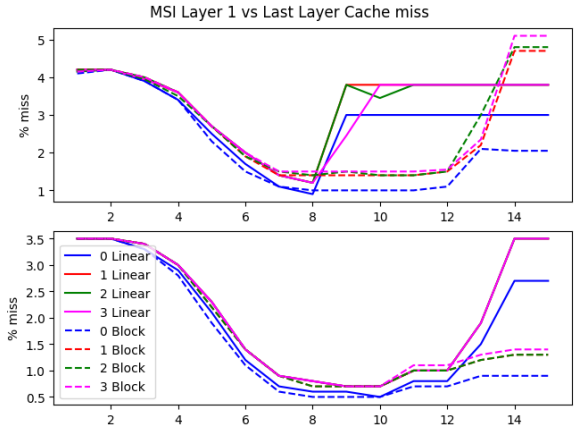

Figure 4: Comparison between caches miss between the first layer and last layer cache of the MSI Laptop

2^6 and $2^7 * 2^7$ for the HP laptop, and $2^8 * 2^8$ for the MSI laptop.

Additionally, an important observation is the difference in the percentage of cache misses between the linear and block access patterns. In tests using the linear access pattern, the percentage of cache misses doubles for both laptops upon reaching a matrix size of $2^9 * 2^9$. In contrast, the block access pattern maintains stability at larger dimensions, indicating its effectiveness in managing cache efficiency.

Interestingly, the bandwidth graph for the MSI laptop shows a noticeable "knee" point at the $2^{10} * 2^{10}$ dimension.

5. Discussion

5.1. Optimization Flags

As anticipated, achieving higher bandwidths was influenced not only by the block access pattern but also significantly enhanced by the use of optimization flags.

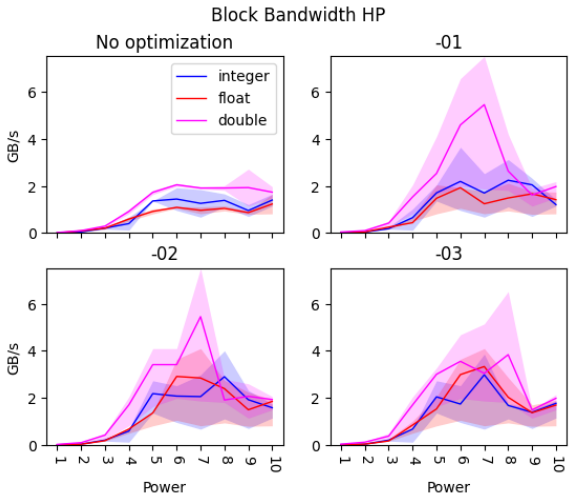

Figure 5: All block bandwidth experiment done with the HP Laptop

The -O1 compile flag alone managed to double the bandwidth achieved, surpassing even the block algorithm without optimization. This improvement is likely attributable to the heuristic optimizations such as "-fguess-branch-probability" and other minor yet impactful instruction optimizations like "-fforward-propagate", which help merge the results of simple instructions.

Conversely, the -O2 flag did not substantially enhance bandwidth performance due to the simple nature of the algorithm. This flag aims to optimize loops more aggressively with options like "-ffinite-loops", which removes smaller loops, and "-fstore-merging", which optimizes memory use for data sizes smaller than a word. The absence of "goto" instructions in the code also meant that certain optimizations under -O2 were not triggered.

The -O3 flag, known for its aggressive optimization techniques, did increase bandwidth noticeably, though not for the HP Laptop. The lackluster performance on the HP Laptop could be due to the integrated "-fvect-cost-model=dynamic" flag, where the "dynamic" cost model may not have recognized any performance benefits from vectorizing the main loop while in runtime. On the MSI Laptop, however, the loop likely underwent vectorization, explaining the observed performance boost. Additionally, optimizations like "-fversion-loops-for-strides" were not beneficial in this case, as the main algorithm did not utilize patterns like A[i*stride] where the stride is one.

5.2. Linear vs Block

As outlined in the Theory section, the block access pattern can potentially enhance performance,

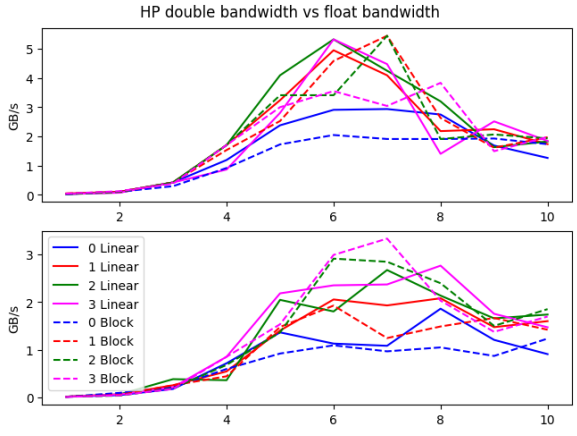

Figure 6: All the block bandwidths experimentation done with the HP Laptop

though this outcome is not guaranteed. For instance, as illustrated in the HP example (see Figure 2), block access patterns often resulted in inferior performance compared to linear access patterns. The block access pattern can introduce significant overhead, particularly in simpler architectures where the instruction and data caches are small and unified, thereby degrading performance. Conversely, in architectures with separate caches for data and instructions, like that of the MSI Laptop, block access patterns can yield improved performance. Another factor could be that the block size was too large, leading to cache eviction when the data was needed again. However, there is reasonable doubt that this was not the main issue, as similar patterns emerged in tests using floats, which occupy only 4 bytes of memory, as opposed to 8 bytes used by doubles [6].

The block algorithm is designed to reduce cache misses, a benefit that is clearly observable in both instances. For operations that are larger and more complex, a block access pattern can be highly advantageous.

5.3. Notable points of performance

In the experiments, using double data types helped define a critical matrix dimension for both laptops: between $2^6 * 2^6$ and $2^7 * 2^7$ for the HP, and $2^8 * 2^8$ for the MSI.

This dimension multiplied to the size of the double data type (8 bytes), matches the capacity of the first-layer cache of each laptop, which is 128 KiB for the HP and 512 KiB for the MSI. These sizes, as shown in the result section, represent the point of peak performance for both machines, as the dimensions allow the branch predictor and cache to efficiently manage data. Consideration that is valid for both access patterns. This setup ensures that half of each matrix remains on the CPU, providing ample time for the RAM to manage data loading and storage, and for both matrices to reside in the additional cache layers.

The "knee" point of the bandwidth can be explained because the $2^{10} * 2^{10}$ matrices will where both the input and output matrices fill the L3 cache. This indicates a lower peak operating point, which might shift to a smaller matrix size if using a less efficient operating system.

When the matrix size reaches a single matrix dimension of 24 MB at $2^{14} * 2^{14}$, close to the MSI's L3 cache capacity, the bandwidth starts to plateau at 5 GB/s. This threshold highlights the limits of cache efficiency and memory management under intensive computational load, since now has to cooperate with the RAM, showing a bigger percentage of misses in the last layer of cache as seen in the figure 5.

6. Conclusion

We've observed that even straightforward operations, such as matrix transposition, can benefit from various optimizations, even within the constraints of sequential processing. However, what if these constraints were removed? Given the inherent data parallelism in this problem,where the outcome of each operation is independent of the rest of the data,we can effectively parallelize the task by assigning different blocks to different threads.

Additionally, another avenue for optimization involves determining the ideal block size that aligns with the specific architecture and the operations being performed. This approach ensures that each thread can operate efficiently, maximizing throughput and minimizing resource conflicts or bottlenecks.