Exploring Implicit Parallelism and Explicit Parallelism with OpenMP

Davide Facchinelli
Mat: 237302
Email: davide.facchinelli@studenti.unitn.it

***Abstract*—This paper examines optimizing matrix transposition using GCC flags for implicit parallelism and OpenMP for explicit parallelism. The objective is to compare optimized codes with the sequential baseline in order to measure performance improvement and highlight the impact of compiler techniques and parallelization. There will also be a brief part regarding the optimization of Symmetry Check function.**

I. Introduction

A. Importance of the problem

The question that may arise is why we would need complex codes and libraries to get a matrix transposed? Eventually, transposing a matrix is a trivial computation that can be simply carried out by accessing the matrix stored in row-major order with a column-major order pattern (that is what libraries such as BLAS [1] do). However, the problem resides in the time needed to access a matrix in that way: the amount of cache misses is considerably impact-full (the chunk of memory loaded together with the first element of the first matrix is always skipped), the spatial locality is no more helpful, the CPU has to load each time the value from the memory, which is much slower than loading it from the cache.
Matrix Transposition, by now, is a widely used technique representing the core of numerous applications such as Image Processing, Signal Processing, Climate Modeling, Bioinformatics, Routing optimization over the Internet, Neural Networks, and many others. Given the range of applications that use it, we can understand how important it is to have optimized logic performing this operation.
A very similar argument can be made for Matrix Symmetry Check, which is important by itself, and in some cases it can avoid carrying out useless Matrix Transposition.

B. Outline of the project

The initial idea of the project was to optimize Matrix Transposition and Matrix Symmetry Check together; however, given that the cases in which randomly allocated matrices are actually symmetric are very low, I decided to split the problem into two parts by allocating random matrices for the Matrix Transposition problem and by allocating symmetric matrices for the Matrix Symmetry Check problem. This allowed me to study the problem separately and use Implicit Parallelization (that rely on techniques provided by compilers and libraries, in our case the GCC compiler) and Explicit Parallelization (provided by OpenMP) to optimize the two different tasks.

II. State of the Art

A. Main solutions and current research

In current research, a great deal of effort is dedicated to Cache Aware and Cache Oblivious algorithms. This is due to the fact that in a program, the operation that requires the most of the time is the load instruction that takes data from the memory and loads it into the CPU (or cache); so this is a challenge that we encounter in nearly every optimization problem (by simply doing a Chache Oblivious research you can get a lot of results, these are some of the ones I read: [6], [5]). One of the simplest techniques in this field is called blocking and it involves dividing, in our case the matrix, into blocks of elements that perfectly fit in the cache dimension so that it is possible to maximize the speedup given by spatial locality. Another technique, called padding, (also this one largely discussed, I most looked at this reference paper: [4]) involves adding to the matrix useless data in order to align the matrix dimension and avoid false sharing among threads: the entire line is entirely assigned to a threads, avoid more threads accessing the same line.
Another challenge, often discussed, is the communication and synchronization between threads: an operation usually needed when the matrix is divided into smaller chucks and computations are carried out independently (that may happen when using techniques such as blocking or tiling). This problem is often faced using OpenMP, which gives the possibility both to parallelize the code and to use a directive to create some sort barriers that cannot be removed until all threads are done.
There are also more advanced research ([3]) that started no so long ago with the advent of Non-Volatile memories and the need of having algorithms for a very large amount of data and systems. I tried to have a look on this topic, but I did not developed anything about it given the relatively small amount of data we have to work with, and the complexity of the code; still I wrote a little about it since I found it very interesting.

B. Objective of the project

The objective of this project was already mentioned in the Introduction, however, I want to point out that in relation to the state of the art I will try to look at the potential of Vectorization and Unrolling, which are near to the idea of blocking.

III. Contribution and Methodology

The directives of the project were about, as already said, exploring Implicit and Explicit parallelization techniques.

For what concerns implicit parallelization, I decided to look at the effect of some of the flags that the compiler offered. In broad terms, the main flags generally used by GCC to optimize a code are -O0, -O1, -O2 and -O3 (there are also other flags that focus on optimizing compilation time instead of execution time, here is the complete set [2]). The level -O0 does not apply any type of optimization, while level -O3 applies a huge set of flags; moreover, note that each level adds a set of optimization flags to the previous level. The problems I encountered with using flags are two. The first is that I had the idea to draw a baseline execution time by using the -O0 flag and then add to it other flags to look at their effect on the execution time. However, as documented here [2], besides removing all the flags, -O0 does not even allow to manually add any type of flag, so it was not possible for me to use the compilation flag -O0 as baseline and study the effect of single flags, which was a thing I really wanted to do with two specific flags: -funroll-loops and -ftree-loop-vectorize, for that reason I wrote a code that tries to perform exactly these two operations for both the Matrix Transposition and the Matrix Symmetry Check operations. Here below [1] I reported the pseudo code of the vectorization of the matrix transposition code in blocks of 4*4. I also wrote the code for blocks of 8*8 elements, but it is more complicated and thus less intuitive to explain here. The idea behind vectorization is to use intrinsics functions that allowed me to manually load some chunks of data in the SIMD registers of the CPU and perform on it equal operations. The second problem I had with optimization flags was in the Symmetry Check code with flags -O1 or higher. A characteristic of these flags is that they remove unused code (or death code); in other words, they do not compute the parts that are not used; this was the case for the Symmetry Check function: it was returning an integer as result that was never used. I found the solution by printing the result without affecting the time measure. For what concerns explicit parallelization with OpenMP, in general, I tried to find the best combination of flags that returned the lowest execution time of the code. The most problematic part was in the Symmetry Check Parallelization. I structured the code in a way that when a non-symmetric cell is found, a shared variable is modified, and eventually the code terminated. This could lead to some race conditions, so I tried three different methods to avoid them: use atomic writing or critical section, both I combination with a local variable reduction. The problem is that critical sections introduce a significant overhead in the execution time (due to faster threads being stopped until all the other reach that point), so the option I adopted was to use atomic writing. In this way the threads could go on executing without having to wait each others.

Algorithm 1 Matrix Transposition with Blocking

```
Require: row0, row1, row2, row3 (__m128 registers with 4 elements each)
1: tmp0, tmp1 ← Combine lower/higher parts of row0 and row1
2: tmp2, tmp3 ← Combine lower/higher parts of row2 and row3
3: col0, col1 ← Combine lower/higher parts of tmp0 and tmp2
4: col2, col3 ← Combine lower/higher parts of tmp1 and tmp3
5: return col0, col1, col2, col3 (__m128 registers with 4 elements each)
```

IV. Experiment and System Description

A. Description of the used platforms

All the tests I made were run on my Windows machine (for convenience reasons), but the final results I present below are taken from the execution of the codes in the university cluster (which is a much more controlled and stable environment). If needed, it is possible to look at all the data that I collected in the link to the Google Sheet that I put in the last section. There you can also find the link to the git repository with the readme.md file that contains the reproducibility instruction for both the Windows machine and the university cluster architecture.

B. Project setup

In the repository you can find 12 files (6 for the Matrix Transposition and 6 for the Symmetry Check) plus an additional MainScriptC.pbs file that can be used to run all the codes in a single shot on the cluster. Each of the 12 files contains a different technique that I used (the same in both the problems):

- sequential (with eventual flags),
- explicit unrolling,
- explicit vectorization in blocks of 4,
- explicit vectorization in blocks of 8,
- parallelization with OpenMP (and eventual directives),
- effect of different number of threads.

Here is a list with some things that are important to notice for the correct code setup and comprehension:

- At the top of each file it is possible to find the declaration and description of all the functions that will be used in it (their definition is done at the end of the file).
- If needed, in all the files, there is the possibility to remove the comment to the function that prints the matrices to look if they are actually transposed (or symmetric). If the matrices are too big to visually check the result there is the possibility to remove the comment to the "matrix_actually_transposed" function that checks the correctness of the operation automatically.
- For the unrolled codes, in order to increase/decrease the unroll level, you have to modify the comments in the function definition (this process is better explained in the function itself).
- For the OpenMP codes there is the need to modify the comments (in the transposition/symmetry check function) in order to use the different directives.
- The result for each execution is the average of 50 executions.

V. Result and Discussion

A. *Presentation and Analysis of Matrix Transposition*

In this section I present the results that I get by optimizing the Matrix Transposition algorithm. The 4 methods plotted in the graph [1] (in logarithmic scale) are: the baseline execution time given by the sequential code (with flag -O0), the Implicit Optimization (with flags -O1 -floop-interchange), the Explicit Vectorization in blocks of 8 (written by me) and the OpenMP with the directives collapse(2) and static schedule (with 8 threads). The choice of using 8 threads was not casual: I noticed that on the cluster there was to possibility to add much more threads, and for big matrices, this could have been a good idea (you can see it from the data I collected in the link below); however, this was not the case for small matrices so, I decided to use 8 threads, which is a number of threads that is much more easy to find on a common machine.

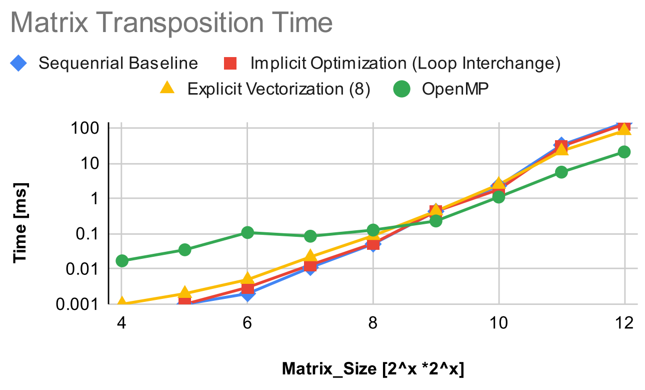

Fig. 1. Matrix Transposition Time

From the graph we can observe that OpenMP, for small matrices, is much more slower than the other algorithms. This is due to the fact that creating and closing threads takes some time, and with small amount of data, that time overweights the optimization given by the parallelization. This is also confirmed by the fact that with bigger matrices OpenMP shows much better performances with respect to the other codes (recall that the graph has a logarithmic scale).
It is possible to observe a similar behavior for the Explicit Vectorized code: it starts performing worst than the others, but it ends up with slightly better performances. This is due to the additional operations needed to load the data in the SIMD registers and store it back at the end, that with smaller matrices, represent a relatively bigger workload.
For what concerns the Implicit Optimized code we can arguably notice some difference with the Baseline code, however that is not unexpected since the objective of it was only to understand the effect of the -floop-interchange optimization flag; still I want to point out that this is the one plotted because this was the best Implicit Optimized code that I could reach (To see the proof of that at the end of the report you can find the link to all the data that I collected).

We can also compare the three presented solutions in terms of speedup: by using the sequential code as Baseline we can understand what is the advantage, in time percentage, that we get by using different of the techniques. The formula used to compute it is:

$$Speedup = \frac{BaseTime - TestTime}{BaseTime} \cdot 100\% \qquad (1)$$

In the graph [2] below, we can see plotted the results given by the speedup approach. The graphs presents results only for

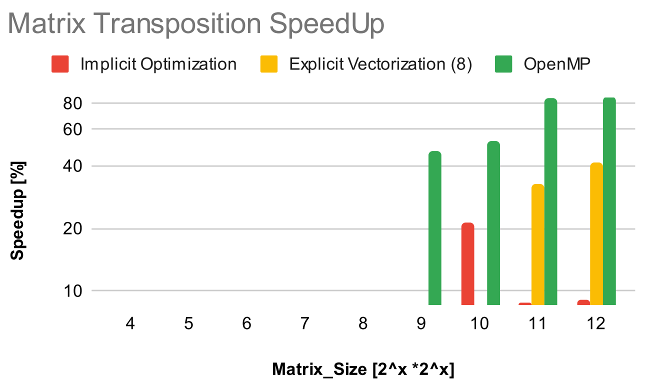

Fig. 2. Matrix Transposition Speedup

matrices bigger than 256*256, which means that, with other algorithms, we get better result than the baseline, only when the data processed is greater than a certain amount. From the graph we can observe, again, how OpenMP, after the threshold gives best results w.r.t the others algorithms.

B. *Presentation and Analysis of Symmetry Check*

As previously mentioned, I divided the problem of matrix Transposition and Matrix Symmetry Check, and now I will open a little parasynthesis to explain the result given by the Matrix Symmetry Check. The procedure that I adopted was the same as the Matrix Transposition part, looking at both Implicit and Explicit Optimization.
In the graph [3] I presented the Sequential Baseline, the Implicit Optimization (with unrolling), the Explicit Unrolling in blocks of 8 (written by me) and the OpenMP optimization with the directive: Collapse(2), Reduction (for the IsSymmetric variable) and static scheduling (with 8 threads). We can again observe that OpenMP start with general worst performance but still it is the best choice for bigger matrices. The rest of the algorithms, the more the matrix size increases, the more uniformed results give back. We can also say that the Implicit Optimization with unrolling gives back better result that the Explicit optimization written by me.

C. *Peak Performance Comparison of the Windows Machine*

For sake of simplicity, I studied the performance of my PC, and not the one of the cluster. The Peak Performance of a memory architecture, also known as theoretical bandwidth,

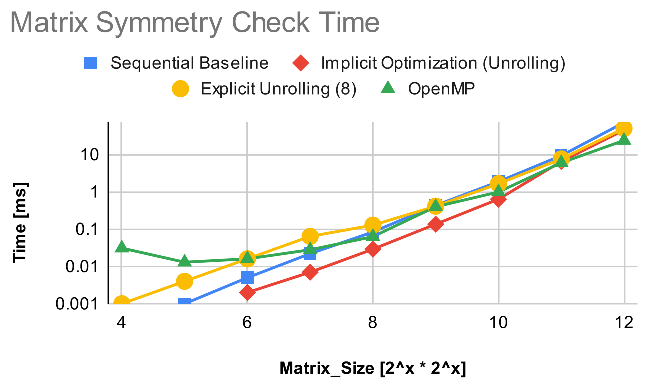

Fig. 3. Matrix Symmetry Check Time

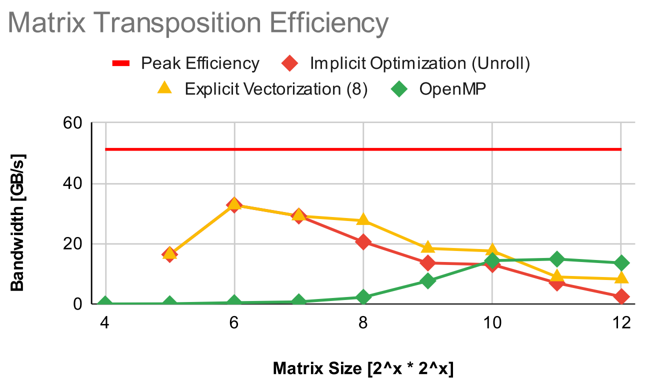

Fig. 4. Matrix Transposition Efficiency

is defined as the maximum possible data transfer rate, from memory to CPU, that a memory technology can handle assuming perfect conditions (No delay, perfectly pipe-lined architecture, ...). The formula I used to compute it is:

$$\begin{aligned} Bandwidth &= ClockSpeed \cdot 2 \cdot BusWidth \cdot Channels \\ &= 3200\frac{MT}{s} \cdot 64bit \cdot 2 = 409600\frac{MT \cdot bit}{s} \\ &= 409600\frac{MHz \cdot bit}{s} = 51.200GB \end{aligned} \tag{2}$$

Notice that the reason why the *ClockSpeed* is doubled is that many architectures mount DDR memories, capable of transferring data both at rising and falling edges of the clock: for each clock cycle two transfer operations are accomplished. However, that is not always the case, so to get to know your memory type you can use the CPU-Z software. It will also show the rest of the data that you need to compute your theoretical bandwidth.
Now I can compare the best results that I get throughout the project with the top performance of my computer to better understand what my machine can offer, note that the result will computed as percentage of bandwidth utilized:

$$Efficency = \frac{MeasuredBandwidth}{IdealBandwidth} \cdot 100[\%] \tag{3}$$

where

$$MeasuredBandwidth = \frac{DataTransferred}{10^9 \cdot Time}[\frac{GB}{s}] \tag{4}$$

Nevertheless, we can already state that due to some straightforward reasons (such as the interested code not being the only process running on the machine, part of the resources being allocated exclusively to the OS, Codes not being perfect, ...), we will never be able to reach the top performances of an architecture. The graph [4] represents the theoretical pick performance of my machine against the efficiency of my codes.

As expected we do not reach the top performance (represented by the red line) although we can notice that OpenMP start off by using very little of the whole bandwidth and increases with time. Viceversa, with the Implicit Optimization of the sequential code and the Explicit Vectorization we get better performance with small matrices, but worse results with bigger ones. The difference between the Implicit Sequential Optimized code and Explicit Vectorized code lies in the fact that the vectorization uploads from the memory blocks of 8*8 values, while the unrolling takes only 8*1 blocks of value repeated times.

VI. Conclusion

The project successfully explored the effect of Implicit and Explicit optimization. By using GCC compiler flags and OpenMP directives, it was possible to understand the best combination of each of them that allowed me to get the best performances, and by manually writing the Vectorization code it was possible to understand what is the real effect of this kind of optimization.
Although far from achieving peak performance (or the State of Art complexity level), optimized implementation utilized a significant portion of the system bandwidth.
The whole project and the findings highlighted the value of using optimization techniques and the power of OpenMP.

VII. Git and Instruction for Reproducibility

Here you can find the link to all codes and also the link to all the data I collected.

- You can find the Git Repository **here** with all the instruction to reproduce the code both on the cluster and on a windows machine.
- You can find the Google Sheet **here** with all the results collected throughout the project.

References

[1] BLAS Contributors. Blas documentation. https://www.netlib.org/blas/, 2020. Accessed: 24-11-2024.

[2] GCC Contributors. Gcc optimization flag documentation. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2020. Accessed: 10-11-2024.

[3] GZhenhua Cai, Jiayun Lin, Fang Liu, Zhiguang Chen, Hongtao Li. Nvm based caching. https://ieeexplore.ieee.org/document/9443933, 2020. Accessed: 25-11-2024.

[4] IEEE. Padding free bank conflict resolution for cuda-based matrix transpose algorithm. https://ieeexplore.ieee.org/document/6888709, 2014. Accessed: 25-11-2024.

[5] Kamen Yotov, Thomas Roeder, Keshav Pingali, Fred G. Gustavson. Cache-oblivious and cache-aware comparison. https://www.researchgate.net/publication/221257357_An_experimental_comparison_of_cache-oblivious_and_cache-conscious_programs, 2007. Accessed: 25-11-2024.

[6] Lars Arge, Michael A. Bender, Erik Demaine, Charles Leiserson and Kurt Mehlhorn. Cache-oblivious and cache-aware algorithms. https://drops.dagstuhl.de/storage/16dagstuhl-seminar-proceedings/dsp-vol04301/DagSemProc.04301.1/DagSemProc.04301.1.pdf, 2005. Accessed: 25-11-2024.