Report Homework 1

Alex Pegoraro 227642
alex.pegoraro@studenti.unitn.it

October 25, 2023

Link to Git repository: `https://github.com/AlphaNightLight/parallel_homework1`

1 Array addition and vectorization

This problem asks us to use implicit parallelism tecniques to improve the performance of a program that compute element-wise addition of two arrays. In this report I will first focus on a description of the sequential program, then I will apply explicit vectorization to it through AVX intrinsics, and finally I will show that simply playing with compiler flags we can achieve an even better performance.

All the programs will be tested on my local machine with scripts to automate the process. Finally, I will convert the mesurament into plots, and comment them.

1.1 The sequential program

The first attempt to solve the exercise is presented in the file `array_addition.cpp`, composed by three functions. Its `main` is responsable for collecting the time results for different dimensions of the arrays, and storing them in a .csv file. Furthermore, to avoid spikes due to fluctuation of CPU resources I decided to reuse the same dimension 4 times and perform an arithmetic average over its times.

The function `array_addition` takes as an input the size of the arrays, to allocate them dynamically and provide an initialization with random numbers. It also prints debugging information: the initial arrays, the result and the execution time. These information are not necessary for the final goal, so I have commented them in the release version.

The core routine of the program, is `routine1`, that uses the arrays (and their dimension) to perform the sum, with a very simple `for` statement. This is the operation we are interested to improve. To mesure its time, I used the functionalities of the `<chrono>` library offered by C++.

1.2 AVX Intrinsics

In the program `array_addition_AVX.cpp` I used AVX intrinsics to manually exploit the vector register exposed by an Intel architecture. The source code looks quite similar to the previous one, in fact the only modifications are the substitution of `routine1` with `routine2` and a change in the CSV filename.

The first thing to do in order to use AVX is to declare vector variables. In this case I decided to use `__m256`, a type that will be mapped to a vector register containing 8 floating point values.

Hence, the step of the `for` will no more be 1, but `i` can be incremented by 8 units per cycle[1]. Inside the loop, the vector variables are loaded with the value of the arrays, another variable will contain the result of their addition and such a result need to be stored back in the final array. All these operation are carried out with intrinsics. Particoular attention should be done in the compilation of such a code, in fact to be able to use vectorization we must tell it the compiler throw a particoular flag: `-mavx`.

For this section, I used as my main source the paper: "Vectorization and Parallelization of Loops in C/C++ Code"[2] by Xuejun Liang, Ali A. Humos, and Tzusheng Pei.

1.3 Autovectorization

Vectorization can also be achieved with no modification applied to the `.cpp` file. In such a case, I leave the dirty job to the compiler, that automatically converts the code thanks to some specified flags. In particoular I tried to activate the third optimization level adding `-O3` to the `g++` command, as among the other things it also perform auto-vectorization. To be sure that the code really gets vectorized I asked for a feedback regarding vectorization thanks to `-fopt-info-vec`. With such a configuration the compiler gave a positive answer:

```
array_addition.cpp:109:12: note: loop vectorized
```

In such a situation, I can obtain a new executable simply compiling `array_addition.cpp` with the specified options. It worth to notice that the output program will be very different from the first one, even thaught they have been compiled from the same source code.

1.4 Automatic testing

In order to facilitate the experiment reproducibility, I decided to create three bash scripts. The one called `compile.sh` compiles `array_addition.cpp` in two different executables, one with no optimization specified and one with the autovectorization flags, and `array_addition_AVX.cpp` with the flag `-mavx`. Another script called `benchmark.sh` execute these 3 programs, taking also care to reneme the output of the autovectorized code[3]. The last script, `clean.sh` is in charge to delete all the executable and the CSV files, leaving the workspace with only essenial elements.

The last ingredient of the experiment, is `plot_results.m`, a GNU Octave script to plot the results from the CSV. I decided to create two plots for the first exercise. The first one (Figure 1) shows the size of the arrays in the x axis, in a logarithmic scale, and the average execution time on the y axis, in seconds. This representation has the advantage of showing us the real temporal difference between our mesures, but due to the logarithmic scale on only one axis lines are deformed into an exponentials. So, I created a second plot in wich times are reported in logarithmic scale as well (Figure 2). In such a case, the function returns to be a line as it would be in a normal plot with liear scales.

1.5 Results

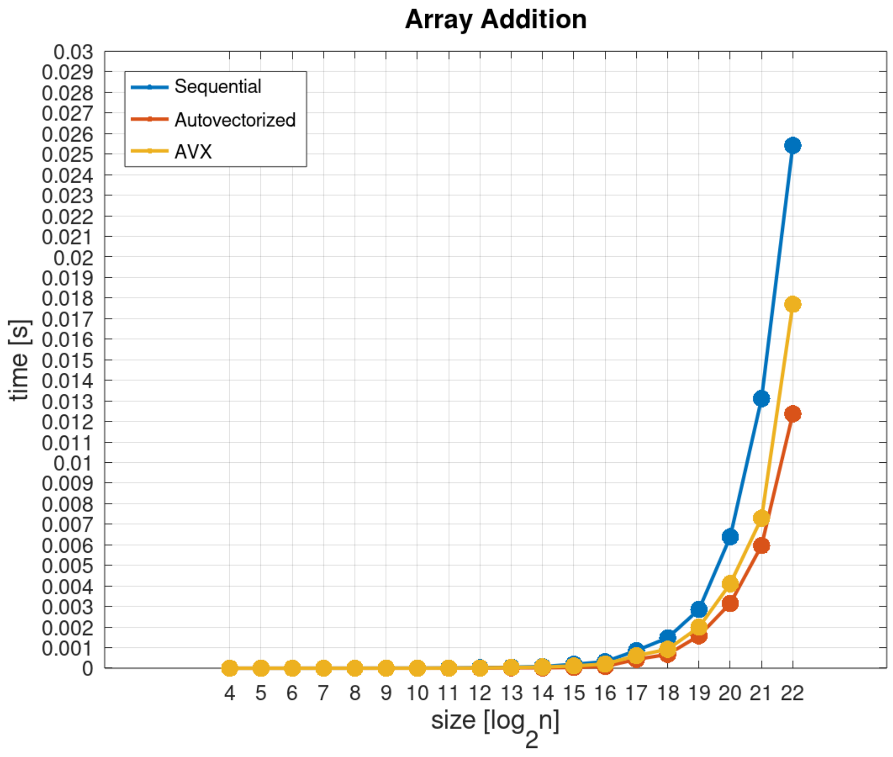
I wasn't able to connect to the cluster due tu VPN problems, so I run the test script on my machine. Its CPU is an Intel(R) Core(TM) i3-2370M, 2.40 GHz, and the RAM is a 4GB DDR3.

[1]In normal cases it would be necessary to ensure that we are not overcoming the boundary of he aray in the last iteration, but in our benchmark we will have only power of 2 tarting from 16, so all the sizes will be a multiple of 8.

[2]`https://www.jsums.edu/robotics/files/2016/12/FECS17_Proceedings-FEC3555.pdf`

[3]This is necessary to avoid an overwrite caused by the purely sequential program, as it has been compiled by the same source and so it aims to create a file with the same name.

Figure 1: A plot of the array addition routines' performance. The size of the array is in logaritmic scale of 2, while the execution time is in linear scale.



Looking at Figure 2, we first notice that, as expected, the execution time is linear with respect to the dimension of the arrays, in all the three analyzed routines. Selecting the sequential program as the baseline, the AVX vector registers gave a meaningful improvement in performance, but the compiler flags overcomes both of them. In a simple algorithm like this (at the end, nothing more tha a `for` loop) I expected my explicit vectorization to be close to the optimal, and I think the significant gap between it and the autovectorized one is due to the fact that `-O3` doesn't limit itself to vectorization, but probably improves other aspect of the source code I'm neglecting.

2 Matrix copy via block reverse ordering

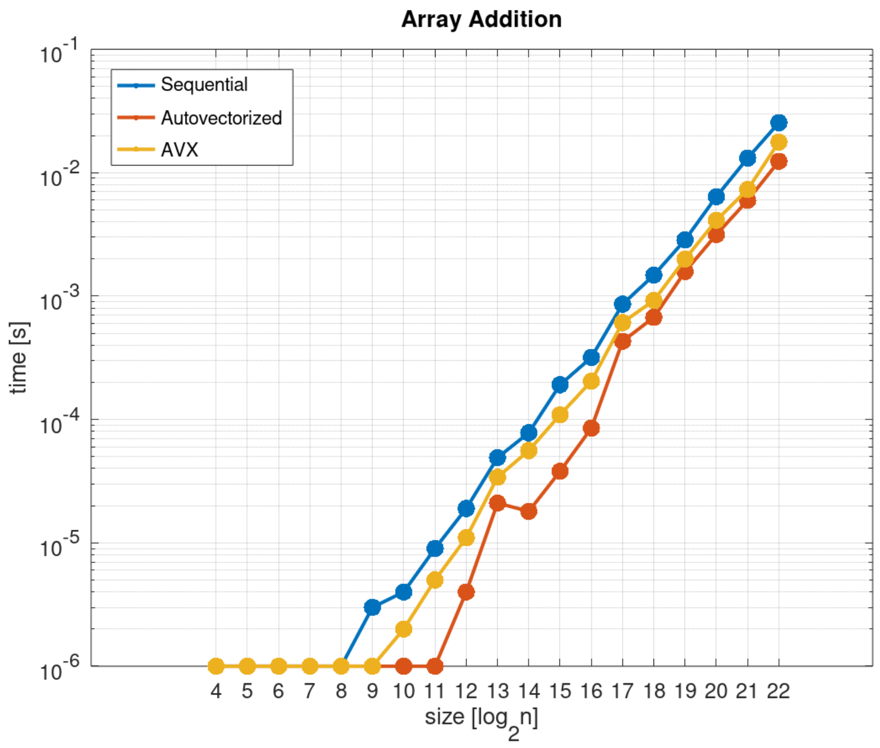
In the second problem, the goal is to split a matrix into blocks of size b, and copy these blocks in the reversed order into another matrix. As for the previous exercise, the first section will be dedicated to the sequential program. In the second one, I will try different access patterns with the intent of exploit the cache as much as possible. Finally, I will play with compiler flags.

2.1 The sequential program

The first code I will analyze for this problem, is `matrix_copy.cpp`. Its structre mimics the one of the first exercise, with a `main` function that collects the time mesuraments, average them and store them in a CSV, a `matrix_copy` funztion that dynalically allocates the matrices and a `routine1` that performs the core computation. It is important to notice that `n`,the size of the matrix, will never change, the only variable parameter is `b`, the size of the block.

The variable `a = n / b;` is the number of blocks per line. It allowes to express the $n \times n$ input

Figure 2: A plot of the array addition routines' performance. Both the execution time and the array size are in logaritmic scale.



matrix as a $a \times a$ block-matrix whose elements are $b \times b$ matrices. Hence, the iteration of the matrix will be a little bit unusual. It consists of 4 nested `for` loops, the outer two with variables `i` and `j` will navigate the blocks, while the inner two will access the elements of each block throw `ib` and `jb`. The opertion we have to perform to invert the block-matrix would be:

$$\mathtt{O}[\mathtt{a}-1-\mathtt{i}][\mathtt{a}-1-\mathtt{j}] = \mathtt{M}[\mathtt{i}][\mathtt{j}];$$

But all the elements of this pseudo-matrix are matrices themself, whose size is `b`, so the code line becomes:

$$\mathtt{O}[(\mathtt{a}-1-\mathtt{i})*\mathtt{b}+\mathtt{ib}][(\mathtt{a}-1-\mathtt{j})*\mathtt{b}+\mathtt{jb}] = \mathtt{M}[\mathtt{i}*\mathtt{b}+\mathtt{ib}][\mathtt{j}*\mathtt{b}+\mathtt{jb}];$$

2.2 Improve cache usage

In C++ matrices are stored in a row major order, that means they are memorized as a sequence of their row vectors. A consequence is that access the elements by columns can be much slower than access them by row, as they are not consecutive. This situation is even worse in dynamic matrices, because every row is allocated independetly and so different rows can be in complete different parts of the Heap.

I tried to print the order in wich elements are accessed in the debug commented lines of `routine1`. This showed me a pattern in wich the program continuously jump between the cells. However observing the code I noted that the `for` loops can be swapped without affecting the correctness of the result[4] but only affectiong the access pattern. This suggested me to reorder the

[4]I mean that every elements of M will go to the right place of O regardless of the nesting order of the loops.

loops in a way that the matrix is accessed by rows instaed of single separated cells. The result is `matrix_copy_prefetch.cpp`, a copy-and-paste of the first solution but with different nesting.

I thaught that it would be interesting to see for comparison the worst possible access pattern, in this case the one by columns, so I implemented it in `matrix_copy_prefetch_reversed.cpp`, that is just another copy-and-paste of `matrix_copy.cpp` with a different order of iterative statements.

2.3 Autovectorization

The previous exercise thaught me that autovectorization outperfors the explicit use of AVX. For this reason I directly moved to compiler options to avoid another modification of the source. I again chose `-O3 -fopt-info-vec` flags as they provide better result compared to other solutions. Also this time I recieved positive notifications from g++:

```
matrix_copy.cpp:143:17:  note:  loop vectorized
matrix_copy.cpp:33:12:  note:  loop vectorized
matrix_copy_prefetch.cpp:143:17:  note:  loop vectorized
matrix_copy_prefetch.cpp:33:12:  note:  loop vectorized
```

2.4 Automatic testing

I managed to update the testing scripts as follows. In `compile.sh` I added instructions to compile the three source codes with no flags as well as the serial and the cache-friendly ones with vectorization flags[5]. `benchmark.sh` executes the programs and resolve conflicts in output files' naming, and `clean.sh` removes all unnecessary files.

The plot script was updated a well. This time I'm plotting the execution times over the size of the block, with the size of the matrix fixed at 4096. As before, on the x axis we need a logaritmic scale, but this time we have no problems in maintain the y axis linear as the trends we observe are mostly horizontal. So, the plots I provided for this exercise are Figure 3 in wich I drawed all the implementations, and Figure 4 in wich I removed the cache-unfriendly version to have a better zoom on the others.

2.5 Results

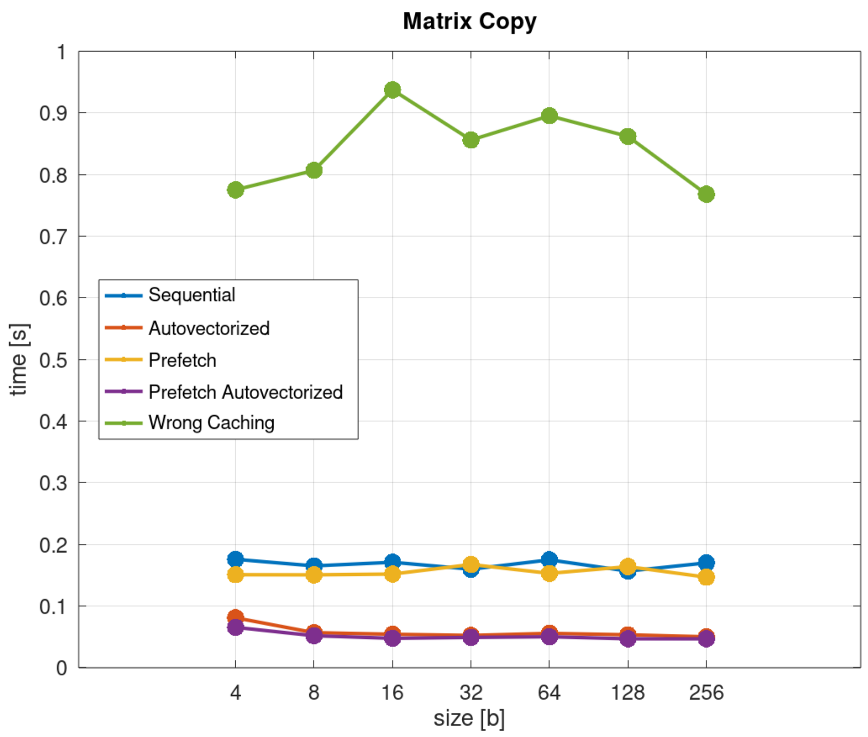
The main purpose of Figure 3 is to show how important is the access pattern in this exercise. In fact, the cache-unfriendly implementation is way slower than the other ones, so much that to interpret the information given by the data generate by other solutions, it must be removed from the plot switchong to Figure 4.

As for the previous exercise, the plot helps in identifying the trend, an horizontal line in this case. This did not surpised me: unlike before, we are not modifying the size but the way the data structure is subdivided. Hence, the dimension of the problem remains fixed (to 4096^2 in this case) and so it should be the execution time. Another aspect that catches the eye, is that the sequential and the prefetch routines intersects each other. I suppose it is because their timing is quite similar, and so even a minimal fluctuation can result in the slower overcoming the faster. In the plot I choose to include in this report the two autovectorized lines never cross each other, but in some tests they do it as well.

The last thing I can discuss, is that the main improvement on the code was given by the vectorization rather then the cache usage. It is a good news, as it means that the original access

[5]The cache-unfriendly version is just for comparison purposis, it's not meant to be used and improved.

Figure 3: A plot of the matrix copy routines' performance. The block size is in logaritmic scale of 2, while the time is in linear scale. The size of the matrix is fixed at 4096×4096 elements.



pattern of `matrix_copy.cpp` was not so bad as I thaught, because its performance is much coser to the cache-friendly one rather then the cache-unfriendly one.

2.6 Bandwidth

The theorical memory bandwidth peak is given by the subsequent formula:

$$Bandwidth_{ideal} = memory_{clock} \times memory_{interface} \times TDR$$

Where the components involved are the memory clock rate, the memory inrterface and the transfer data rate respectivley. Reading the specifics of my machine I found that it has 1300 MHz memory clock rate, a 8 bytes DIMM iterface, and a TDR of 2 since it is a DDR3.

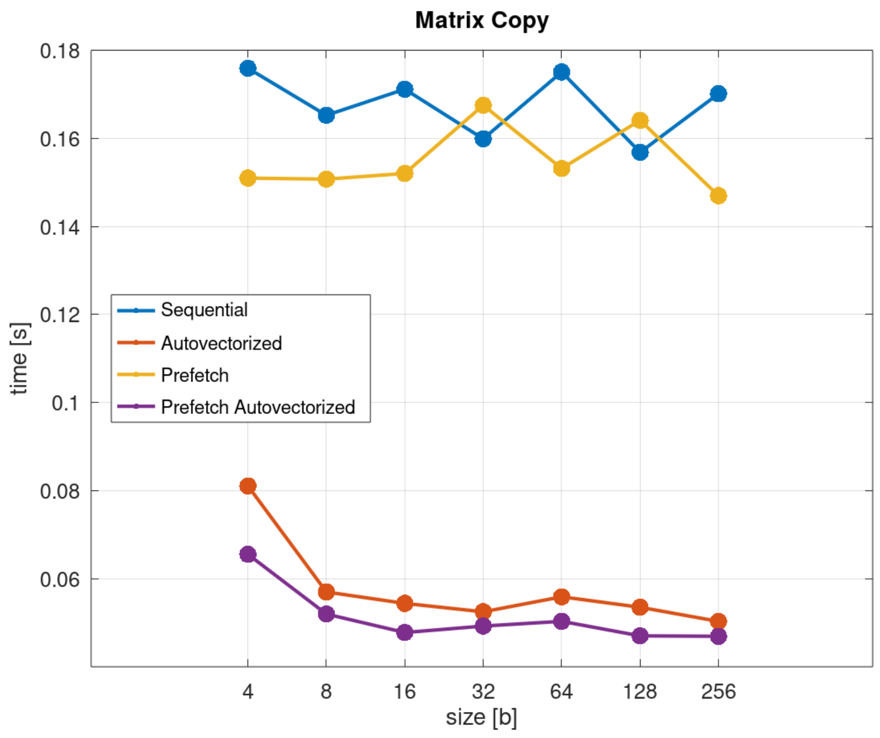
$$Bandwidth_{ideal} = 1,3 \times 10^9 \times 8 \times 2 = 20,8\ GB/s$$

This result should be compared to the actual performance obtained by the program, in which we take into account the amount bytes read and written by the routine and the effective execution time it takes.

$$Bandwidth_{effective} = ((Br + Bw) \div 10^9) \div t_{routine}$$

In this particoular instance of the problem, the matrix dimension is fixed at 4096 floats, that means a single matrix has a weight of $4096 \times 4096 \times 4$ bytes. This number represents both the read and the written date. For the execution time of the routine, I take as representatives the executions of `matrix_copy.cpp` with no flag and `matrix_copy_prefetch.cpp` with optimization flags, both

Figure 4: A plot of the matrix copy routines' performance, with the wrong cached program removed. The block size is in logaritmic scale of 2, while the time is in linear scale. The size of the matrix is fixed at 4096 × 4096 elements.



considered in the case of the largest block. These values can be read in the respective CSV file, an are 0,170 s and 0,047 s respectivley. Plugging them into the formula we obtain:

$$Bandwidth_{serial} = ((2 \times 4096 \times 4096 \times 4) \div 10^9) \div 0,170 = 0,790\ GB/s$$

$$Bandwidth_{improved} = ((2 \times 4096 \times 4096 \times 4) \div 10^9) \div 0,047 = 2,856\ GB/s$$

Unfortunatley, in both the cases there's an effective bandwidth quite smaller than the optimal one, and for this reason I won't add it into the plot as I don't want an eccessive zoom out. This is due to the fact that my computer has some issues in mamory management (it's a problem I was already aware), in fact before running the simulation I asked the task manager the memory usage, and it was about 89% only for system applications. Hence, I woulden't impute the very low performance the program implementation but to the fact that most of the bandwidth was already occupied by other processes.

Apart from this issue, I would like to emphasize that there's also a big difference between the bandwidth of the serial program and the one of the optimized one, meaning that indeed an improvement has taken place, even if it didn't brough the application close to the peak.

2.7 Final remarks

What I lerned more from this project, is that the proper usage of the compiler and its flags can have a much higher impact on the performance than what I thought.