

# POLITECNICO DI TORINO



BACHELOR OF SCIENCE IN MATHEMATICAL ENGINEERING

BACHELOR DEGREE THESIS

## Analysis of matrix vector product

Supervisors:  
Prof. Stefano Berrone  
Dr. Federico Tesser

Candidate:  
Ludovico Bessi

July 2019

*Ai miei genitori*

## **Abstract**

Comparison between different self-made implementations of the dense matrix vector product and Eigen's library implementation, using different optimizations flags on a single core.

After getting to the theoretical limit of performance on our machine, I addressed scalability by running the code in parallel using OpenMP.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	STREAM . . . . .	4
1.2	LIKWID . . . . .	5
1.3	Eigen . . . . .	7
1.4	Roof-line model . . . . .	7
<b>2</b>	<b>Naive implementation</b>	<b>10</b>
2.1	Different optimization flags comparison . . . . .	11
2.2	Decreasing matrix dimension . . . . .	11
2.2.1	n = 44 . . . . .	11
2.2.2	n = 400 . . . . .	11
<b>3</b>	<b>Temporary variable</b>	<b>12</b>
3.1	Different optimization flags comparison . . . . .	12
3.2	Decreasing matrix dimension . . . . .	13
3.2.1	n = 44 . . . . .	13
3.2.2	n = 400 . . . . .	13
<b>4</b>	<b>Loop unrolling</b>	<b>14</b>
4.1	Different optimization flags comparison . . . . .	15
4.2	Decreasing matrix dimension . . . . .	15
4.2.1	n = 44 . . . . .	15
4.2.2	n = 400 . . . . .	15
<b>5</b>	<b>Eigen implementation and comparison</b>	<b>16</b>
<b>6</b>	<b>Parallel computations using OpenMP</b>	<b>18</b>
<b>7</b>	<b>SIMDY FORSE</b>	<b>19</b>
<b>8</b>	<b>Conclusions and future directions</b>	<b>20</b>

# Chapter 1

## Introduction

In this first chapter I will introduce the main ideas to understand three different implementations of the code, plus the tools used to analyze them. Let  $A$  be a  $n \times n$  dense matrix and  $v$  be a vector of  $n$  entries. We define the **kernel** as the following operation:

$$result[i] = result[i] + A[i, j] * vector[i]$$

The kernel is indeed very simple, so there is not much mathematical machinery that can be employed to speed up the calculation.

However, I will show how to write it in slightly different ways to exploit the cache and pipe dimension.

When using the g++ compiler, we can set different optimization flags such as: O0, O1, O2 and O3.

O0 is the default one, and no optimization is done. From here, the higher the number the better the compiler optimizes the code.

The code comparison will then be done with different optimization flags and different size of matrices. The reason for the latter is that there exists a number  $n \in \mathbb{N}$  such that all our data will be saved in the Cache, thus speeding up the algorithm.

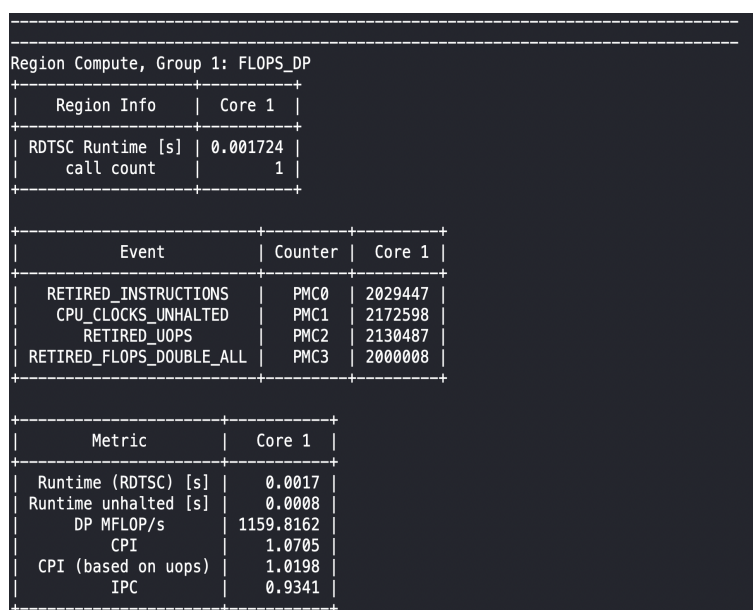
### 1.1 STREAM

STREAM is a simple synthetic benchmark program that measures sustainable memory bandwidth in MB/s and the corresponding computation rate for vector kernels.

Having an estimate of bandwidth is very important because it lets us calculate the performance of our code relatively to the hardware. By running a python script, I calculated that our bandwidth on average is: 6000 MB/s.

## 1.2 LIKWID

LIKWID is a set of command line tools supporting software developers, benchmarkers and application users to get the best performance on a given system. One of the main tool is **likwid-perfctr**, that offers access to hardware performance counters and derives metrics from the raw counts. By running **likwid-perfctr**, we can profile our code. The following screenshot is an example of an output:



```
-----
Region Compute, Group 1: FLOPS_DP
+-----+-----+
| Region Info | Core 1 |
+-----+-----+
| RDTC Runtime [s] | 0.001724 |
| call count | 1 |
+-----+-----+

+-----+-----+-----+
| Event | Counter | Core 1 |
+-----+-----+-----+
| RETIRED_INSTRUCTIONS | PMC0 | 2029447 |
| CPU_CLOCKS_UNHALTED | PMC1 | 2172598 |
| RETIRED_UOPS | PMC2 | 2130487 |
| RETIRED_FLOPS_DOUBLE_ALL | PMC3 | 2000008 |
+-----+-----+-----+

+-----+-----+
| Metric | Core 1 |
+-----+-----+
| Runtime (RDTSC) [s] | 0.0017 |
| Runtime unhaltd [s] | 0.0008 |
| DP MFLOP/s | 1159.8162 |
| CPI | 1.0705 |
| CPI (based on uops) | 1.0198 |
| IPC | 0.9341 |
+-----+-----+
```

A brief explanation of all the metrics:

- **RETIRED\_INSTRUCTIONS**: instructions completely executed between two clocktick event samples.
- **CPU\_CLOCKS\_UNHALTED**: number of cycles where CPU was not halted.
- **RETIRED\_UOPS**: number of low-level hardware operations.
- **RETIRED\_FLOPS\_DOUBLE\_ALL**: number of completely executed floating point operation per second
- **Runtime (RDTSC) [s]**: counts the number of cycles since reset.
- **Runtime unhaltd [s]**: counts the number of cycles since reset when CPU was not halted.
- **DP MFLOP/s**: number of double precision MFLOP/s.

- **CPI**: number of cycles per instruction retired.
- **CPI (based on uops)**: Calculation of CPI based on uops.
- **IPC** : number of instructions per cycles

Below some useful formulas that provide more insights into how these numbers are related to each other:

- $DP\ MFLOP/s = 1.0E-06 * (RETIRE\_FLOPS\_DOUBLE\_ALL) / time$
- $CPI = CPU\_CLOCKS\_UNHALTED / RETIRE\_INSTRUCTIONS$
- $CPI\ (based\ on\ uops) = CPU\_CLOCKS\_UNHALTED / RETIRE\_UOPS$
- $IPC = RETIRE\_INSTRUCTIONS / CPU\_CLOCKS\_UNHALTED$

To understand and use the performance properties of the hardware, it is important to know the machine's topology: **likwid-topology** gives us information about:

- **Thread topology**: how threads are concurrently executed and split in subprocesses.
- **Cache topology**: how processors share the cache hierarchy.
- **Cache properties**: detailed information about all cache levels.

Below an example of the output with information about HW and thread topology:

```

CPU name:      AMD Opteron(tm) Processor 6344
CPU type:      AMD Interlagos processor
CPU stepping:  0
*****
Hardware Thread Topology
*****
Sockets:      2
Cores per socket: 12
Threads per core: 1
-----
HWThread      Thread      Core      Socket      Available
0              0              0          0            *
1              0              1          1            *
2              0              2          0            *
3              0              3          1            *
4              0              4          0            *
5              0              5          1            *
6              0              6          0            *
7              0              7          1            *
8              0              8          0            *
9              0              9          1            *
10             0             10         0            *
11             0             11         1            *
12             0             12         0            *
13             0             13         1            *
14             0             14         0            *
15             0             15         1            *
16             0             16         0            *
17             0             17         1            *
18             0             18         0            *
19             0             19         1            *
20             0             20         0            *
21             0             21         1            *
22             0             22         0            *
23             0             23         1            *
-----
Socket 0:      ( 0 2 4 6 8 10 12 14 16 18 20 22 )
Socket 1:      ( 1 3 5 7 9 11 13 15 17 19 21 23 )

```

Another picture with information on cache levels:

```

*****
Cache Topology
*****
Level:          1
Size:           16 kB
Type:           Data cache
Associativity:   4
Number of sets: 64
Cache line size: 64
Cache type:     Non Inclusive
Shared by threads: 1
Cache groups:   ( 0 ) ( 2 ) ( 4 ) ( 6 ) ( 8 ) ( 10 ) ( 12 ) ( 14 ) ( 16 ) ( 18 ) ( 20 ) ( 22 ) ( 1 ) ( 3 )
                ( 5 ) ( 7 ) ( 9 ) ( 11 ) ( 13 ) ( 15 ) ( 17 ) ( 19 ) ( 21 ) ( 23 )
-----
Level:          2
Size:           2 MB
Type:           Unified cache
Associativity:   16
Number of sets: 2048
Cache line size: 64
Cache type:     Non Inclusive
Shared by threads: 2
Cache groups:   ( 0 2 ) ( 4 6 ) ( 8 10 ) ( 12 14 ) ( 16 18 ) ( 20 22 ) ( 1 3 ) ( 5 7 ) ( 9 11 ) ( 13 15 )
                ( 17 19 ) ( 21 23 )
-----
Level:          3
Size:           6 MB
Type:           Unified cache
Associativity:   48
Number of sets: 2048
Cache line size: 64
Cache type:     Non Inclusive
Shared by threads: 6
Cache groups:   ( 0 2 4 6 8 10 ) ( 12 14 16 18 20 22 ) ( 1 3 5 7 9 11 ) ( 13 15 17 19 21 23 )

```

Lastly, likwid let us change the maximum and minimum frequency of our CPU. To have the least amount of oscillations when running our code, we set both at 2.6GHz, which is the maximum frequency allowed by our machine.

## 1.3 Eigen

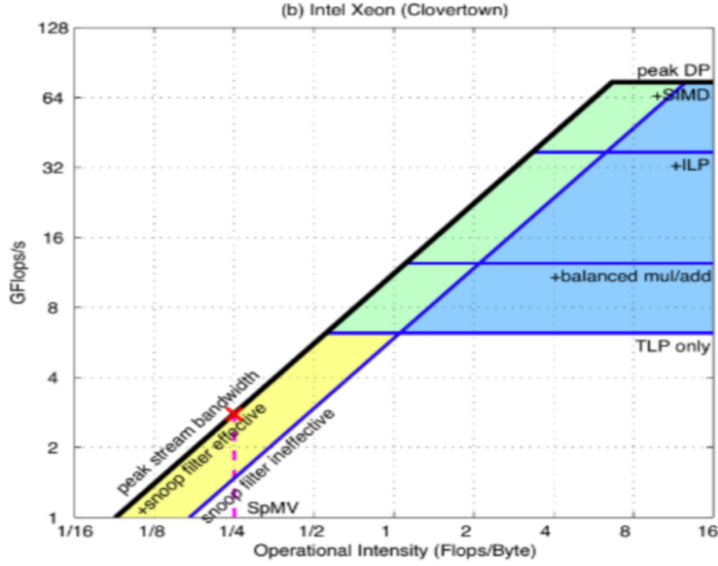
Our main goal is to see how our code measures against Eigen's. Eigen is a library written in C++ which provides algorithms for fast matrix algebra, such as: dense and sparse product of matrix and vectors, matrix decomposition, space transformations.

This library is widely used by many projects and is the result of the work of experts starting at least from 2009. Given this premises, it seems very difficult to get close to their performance. However, even matching it will be a great accomplishment and would show that there is still work to be done to improve it.

## 1.4 Roof-line model

The Roofline model is a performance model used to provide estimates of a given computer kernel. It can be visualized by plotting floating point performance as a function of arithmetic intensity. Below an example of such a plot taken by [4].





This model is used to determine the two main bottlenecks: bandwidth and core. It works under the assumption that data is already saved in cache and there is no latency. It depends on empirical measurements, so it is inherently subject to error. It plots the performance of the code as dependent variable of computational intensity, below I will explain how to calculate such values using the tool described at the beginning of the chapter.

The peak performance can be calculated using the following formula, where  $n$  is the number of cores,  $F$  is the number of floating point instructions for cycle,  $v$  is the clock speed and  $S$  is the SIMDY feature, for now we are not concerned about this last term.

$$P_{peak} = n * F * S * v$$

We can also define the maximum performance reached by our machine with the following formula, where  $f_{max}$  is the maximum frequency of the CPU of our machine,  $\frac{n_{operations}}{n_{cycles}}$  is the number of operations per cycle.

$$P_{max} = f_{max} * \frac{n_{operations}}{n_{cycles}}$$

In our case, the result is then just  $2.6 \frac{Gflops}{s}$ . However, this is just the theoretical limit: our performance  $P$  actually is:

$$P = \min(P_{max}, \text{Bandwidth} * \text{computational intensity})$$

In this formula, computational intensity is a measure of floating-point operations (FLOPs) performed by a given code (or code section) relative to the amount of memory accesses (Bytes) that are required to support those

operations.

We shall see how it is indeed the deciding factor when evaluating different algorithms, because in this framework we ideally want to build an algorithm such as that the value **Bandwidth \* computational intensity** reaches  $P_{max}$ .

With this idea in mind, I developed three different implementations of the dense matrix vector product:

- **Naive implementation:** the classic implementation without much thought, still useful to benchmark against to check progress.
- **Temporary variable:** using a temporary variable to always keep in cache a value to lessen the number of fetches from memory.
- **Loop unrolling:** splitting up the loop manually to exploit pipe levels, that means keeping the pipe occupied avoiding downtime.

With the last implementation, we managed to get to the theoretical limit. After that, we addressed scalability. That is, we looked at how our kernel worked in parallel using OpenMP.

## Chapter 2

# Naive implementation

The naive implementation is just that, the simplest code one can develop to solve this problem, below the .C file:

```
1 for(int i = 0; i < n; i++){
2     result[i] = 0;
3     for(int j = 0; j < n; j++){
4         result[i] += matrix[i][j]*vector[j];
5     }
6 }
```

The computational intensity in this case is  $\frac{2 \text{ Flops}}{32 \text{ Bytes}}$ .

Before blindly applying the formulas seen in chapter one, there are some key insights to understand. until we get to the final implementation, we are not exploiting pipelines of sum and product. In short, this means that we are not allowing the processor to execute these basic operations in parallel. It follows that  $P_{max}$  is limited by a factor of 8, because the size of the pipe for sum and of product are 4 each. We then have that  $P_{max} = \frac{2.6 \text{ Gflops}}{8 \text{ s}}$ , and this is a major bottleneck of our program, because **Bandwidht \* computational intensity** =  $6 * \frac{1}{16} = 0.375 \frac{\text{Gflops}}{\text{s}}$ . We will tackle the bottleneck in chapter 4.

Using likwid-topology, we find the following sizes for the cache:

- L1: 16 kB
- L2: 2 MB
- L3: 6 MB

We are going to focus our attention on the L1 and L2 cache levels, which are the fastest. In short: if the elements we are working with are saved there, we can access them faster than usual. For dense matrices, L2 is actually faster than L1. I am going to repeat the same analysis with two different matrices  $A_1$  and  $A_2$  of size respectively  $n_1 = 40$  and  $n_2 = 400$ . The first matrix can be saved in L1, the second one in L2 and the one with  $n = 1000$  is saved in memory.

## 2.1 Different optimization flags comparison

I will now analyze the code compiling it with different flags: O0, O1, O2 and O3 and  $n = 1000$ .

I am going to focus on the DP Mflops/s, the higher the number the faster the result.

**O0** 328 Mflops/s

**O1** 740 Mflops/s

**O2** 720 Mflops/s

**O3** 761 Mflops/s

## 2.2 Decreasing matrix dimension

By changing the matrix dimension, we get different and interesting results.

### 2.2.1 $n = 44$

**O0** 213 Mflops/s

**O1** 370 Mflops/s

**O2** 345 Mflops/s

**O3** 368 Mflops/s

### 2.2.2 $n = 400$

**O0** 328 Mflops/s

**O1** 948 Mflops/s

**O2** 941 Mflops/s

**O3** 936 Mflops/s

## Chapter 3

# Temporary variable

We can improve the code described in chapter two, introducing a temporary variable in the following way:

```
1 for(int i = 0; i < n; i++){
2     result[i] = 0;
3     tmp = result[i];
4     for(int j = 0; j < n; j++){
5         tmp = tmp + matrix[i][j]*vector[j];
6     }
7     result[i] = tmp;
8 }
```

At first glance, It may seem we are just wasting precious cache space with another double variable but it turns out this change doubles the computational intensity with respect to the first algorithm.

The key take away is that our kernel does not change even with the addition of the *tmp* variable: this time we can skip the fetching and loading of this variable, because when entering the second for loop it is already in the cache. This means that the computational intensity becomes  $\frac{1}{8} \frac{Flops}{Bytes}$ . However, we still have not solved the pipeline's problem described in the previous chapter. This means that even though we doubled the computational intensity our bottleneck is still  $P_{max}$  due to the coefficient  $\frac{1}{8}$ . Nevertheless, by compiling the code we still manage to improve, as seen in the next section.

### 3.1 Different optimization flags comparison

I will now analyze the code compiling it with different flags: O0, O1, O2 and O3 and  $n = 1000$ .

**O0** 328 Mflops/s

**O1** 344 Mflops/s

**O2** 704 Mflops/s

**O3** 766 Mflops/s

## **3.2 Decreasing matrix dimension**

### **3.2.1 n = 44**

**O0** 229 Mflops/s

**O1** 276 Mflops/s

**O2** 357 Mflops/s

**O3** 348 Mflops/s

### **3.2.2 n = 400**

**O0** 332 Mflops/s

**O1** 345 Mflops/s

**O2** 937 Mflops/s

**O3** 961 Mflops/s

## Chapter 4

# Loop unrolling

With this code, I finally manage to exploit the sum and product pipelines, thus reaching  $P_{max} = 2.6$  GHz, then the bottleneck must be in the other term, which depends on the computational intensity of the algorithm. The code can be found below:

```
1  for(int i = 0; i < n; i = i + 4){
2      result[i] = 0;
3      result[i+1] = 0;
4      result[i+2] = 0;
5      result[i+3] = 0;
6      tmp1 = result[i];
7      tmp2 = result[i+1];
8      tmp3 = result[i+2];
9      tmp4 = result[i+3];
10     for(int j = 0; j < n; j = j + 4){
11         tmp1 += matrix[i][j]*vector[j];
12         tmp1 += matrix[i][j+1]*vector[j+1];
13         tmp1 += matrix[i][j+2]*vector[j+2];
14         tmp1 += matrix[i][j+3]*vector[j+3];
15
16         tmp2 += matrix[i+1][j]*vector[j];
17         tmp2 += matrix[i+1][j+1]*vector[j+1];
18         tmp2 += matrix[i+1][j+2]*vector[j+2];
19         tmp2 += matrix[i+1][j+3]*vector[j+3];
20
21         tmp3 += matrix[i+2][j]*vector[j];
22         tmp3 += matrix[i+2][j+1]*vector[j+1];
23         tmp3 += matrix[i+2][j+2]*vector[j+2];
24         tmp3 += matrix[i+2][j+3]*vector[j+3];
25
26         tmp4 += matrix[i+3][j]*vector[j];
27         tmp4 += matrix[i+3][j+1]*vector[j+1];
28         tmp4 += matrix[i+3][j+2]*vector[j+2];
29         tmp4 += matrix[i+3][j+3]*vector[j+3];
30     }
```

```

31 result[i] = tmp1;
32 result[i+1] = tmp2;
33 result[i+2] = tmp3;
34 result[i+3] = tmp4;
35 }

```

The pipeline for the sum and multiplication is constantly filled with the trick of splitting the loop 4 by 4. It can be seen as a generalization of the code with the temporary variable.

This time, the computational intensity is  $\frac{32 \text{ Flops}}{160 \text{ Bytes}}$ . This requires more explanation: in the numerator we have 32 operations, calculated just by looking at the code. In the denominator we get 160 by seeing that we fetch and load  $(16 + 4)$  double elements, thus getting 160 bytes. Then the operational intensity is  $\frac{1 \text{ Flops}}{5 \text{ Bytes}}$ , multiplied by the Bandwidth we get  $1.2 \frac{\text{Gflops}}{\text{s}}$ .

## 4.1 Different optimization flags comparison

I will now analyze the code compiling it with different flags: O0, O1, O2 and O3 and  $n = 1000$ .

**O0** 446 Mflops/s

**O1** 1544 Mflops/s

**O2** 1523 Mflops/s

**O3** 1538 Mflops/s

## 4.2 Decreasing matrix dimension

### 4.2.1 $n = 44$

**O0** 324 Mflops/s

**O1** 973 Mflops/s

**O2** 1011 Mflops/s

**O3** 991 Mflops/s

### 4.2.2 $n = 400$

**O0** 453 Mflops/s

**O1** 2346 Mflops/s

**O2** 3046 Mflops/s

**O3** 2935 Mflops/s



## Chapter 5

# Eigen implementation and comparison

Let's now see how the the Eigen library is doing on the same hardware.

Let's start with  $n = 1000$  as before.

**O0** 125 Mflops/s

**O1** 2000 Mflops/s

**O2** 2037 Mflops/s

**O3** 2048 Mflops/s

With  $n = 44$ :

**O0** 93 Mflops/s

**O1** 604 Mflops/s

**O2** 591 Mflops/s

**O3** 421 Mflops/s

With  $n = 400$ :

**O0** 145 Mflops/s

**O1** 4072 Mflops/s

**O2** 4340 Mflops/s

**O3** 4352 Mflops/s

Cose che notato di cui vorrei parlare:

Se la matrice sta tutta in L1 va peggio che tutta in L2, perché? infatti  $n=44$  fa schifo sia il nostro che eigen ma eigen più schifo  $n = 400$  cioè che sta in L2 va meglio che  $n = 1000$ , che sta comunque tutta in L2, perché? Poca differenza tra O2 e O3, perché? Spiegare perché superiamo P del modello introducendo brevemente roofline cache model

## Chapter 6

# Parallel computations using OpenMP

## Chapter 7

SIMDY FORSE

## Chapter 8

# Conclusions and future directions

blablabla What can be done better? Splitting the matrix in blocks

# Bibliography

- [1] Jan Treibig, Georg Hager, Gerhard Wellein, *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.*
- [2] Mark bull, *A Short Introduction to OpenMP*
- [3] Gaël Guennebaud, Benoît Jacob, *Eigen v3*
- [4] Samuel Williams, Andrew Waterman, David Patterson, *An insightful visual performance model for floating point programs and multicore architectures*