POLITECNICO DI TORINO

Bachelor of Science in Mathematical Engineering

Bachelor degree thesis

Analysis and comparison of matrix vector multiplication codes

Supervisors:
Prof. Stefano Berrone
Dr. Federico Tesser

Candidate:
Ludovico Bessi

May 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.
I also addressed scalability by running the code in parallel using OpenMP.

Contents

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 the code in three different ways to extract as much as performance as we can from the hardware.

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 tries to optimize the code.
The code comparison will then be done with different optimization flags and different size of matrices. Changing the dimension of the matrix shows how much difference there is in saving values in the Cache versus memory.

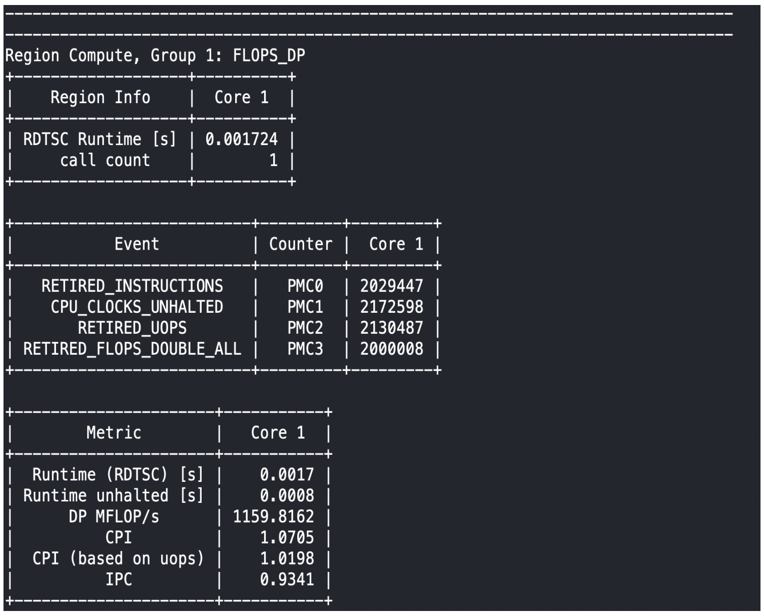
After that, I will address scalability by running the code in parallel using OpenMP.

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 performance monitoring and benchmarking suite. 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 benchmark our code. The following image is an example of an output:



```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Region Compute, Group 1: FLOPS_DP
+-------------------+----------+
|    Region Info    |  Core 1  |
+-------------------+----------+
| RDTSC 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 unhalted [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.
- **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.
- **Runtime (RDTSC) [s]**: the runtime in seconds of the benchmarked section.
- **Runtime unhalted [s]**: the runtime in seconds of the benchmarked section, not considering the halted states of the cpu.
- **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

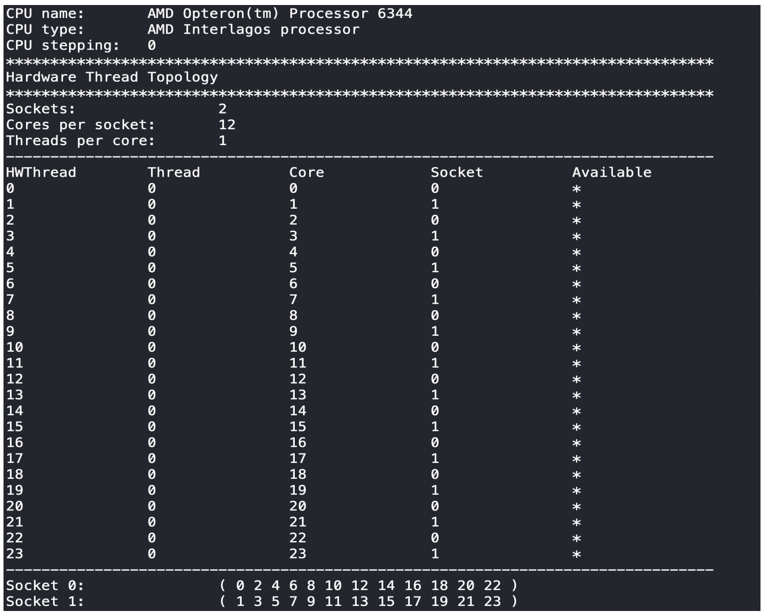
Below some useful formulas that provide more insights into how these numbers are related to each other:

- DP MFLOP/s = 1.0E-06*(RETIRED_FLOPS_DOUBLE_ALL)/time
- CPI = CPU_CLOCKS_UNHALTED/RETIRED_INSTRUCTIONS
- CPI (based on uops) = CPU_CLOCKS_UNHALTED/RETIRED_UOPS
- IPC = RETIRED_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:

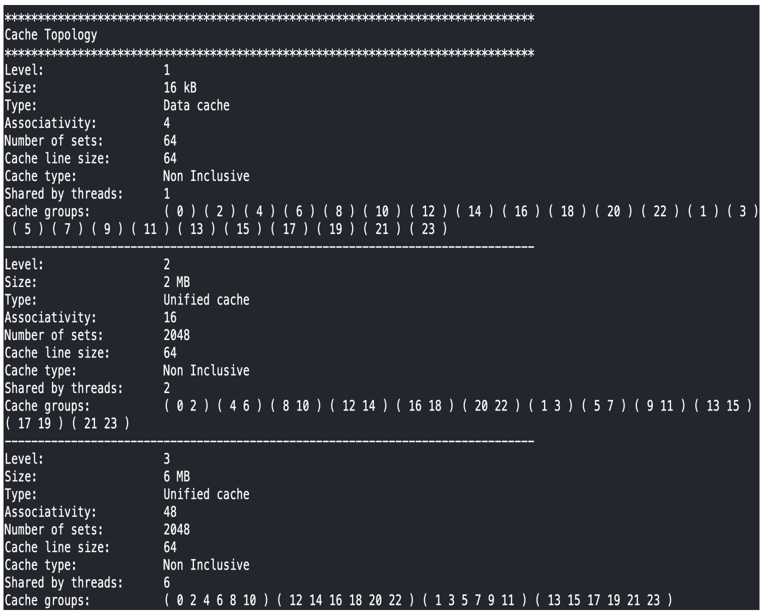
- **Thread topology**: how threads are concurrently executed and split in subprocesses.
- **Cache topology**: how processors share the cache hierarchy.
- **Cache properties**: detailed information about all cache levels.

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:

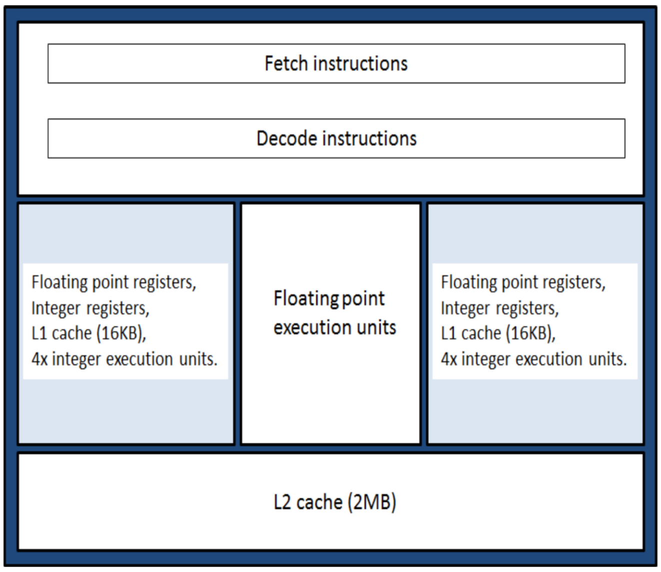


```
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   16 kB
Type:                   Data cache
Associativity:          4
Number 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, they are both set at 2.6GHz, which is the maximum frequency allowed by our machine.

1.3 Hardware specifications

The Interlagos 12-core (for each socket) Opteron processor by AMD's. We The cores are paired into "modules" which share a significant proportion of resources. The figure below gives an overview of which resources are shared between the two cores of a module. More information can be found here: [7]


Fetch instructions
Decode instructions
Floating point registers, Integer registers, L1 cache (16KB), 4x integer execution units.
Floating point execution units
Floating point registers, Integer registers, L1 cache (16KB), 4x integer execution units.
L2 cache (2MB)

1.4 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.

1.5 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.

This model is used to determine the two main bottlenecks: bandwidth and algorithm's perfomance. It works under the assumption that data is "coming from the last level of memory 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 SIMD feature, for now we are not concerned about this last term.

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

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.

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

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 s 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 an algorithm such 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, still useful to benchmark against to check progress.
- **Temporary variable**: using a temporary variable saved in cache 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. Moreover, we manage to get more temporary variables in cache.

After the last implementation, we adressed 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:

```c
for(int i = 0; i < n; i++){
  result[i] = 0;
  for(int j = 0; j < n; j++){
    result[i] += matrix[i][j]*vector[j];
  }
}
```

The computational intensity in this case is $\frac{2}{32}\frac{Flops}{Bytes}$.
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 $P_{max} = \frac{2.6}{8}\frac{Gflops}{s}$, and this is a major bottleneck of our program, because **Bandwitdht** $*$ **computational intensity** $= 6 * \frac{1}{16} = 0.375\frac{Gflops}{s}$. 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 three different matrices of size respectively $n_1 = 40$, $n_2 = 400$, $n_3 = 1000$.
The first matrix can be saved in L1, the second one in L2 and the third one is saved in memory.

Chapter 3

Temporary variable

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

```
for(int i = 0; i < n; i++){
  result[i] = 0;
  tmp = result[i];
  for(int j = 0; j < n; j++){
    tmp = tmp + matrix[i][j]*vector[j];
  }
  result[i] = tmp;
}
```

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 $\frac{2}{16}\frac{Flops}{Bytes}$. 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 $\frac{1}{8}$ of P_{max}. Nevertheless, by compiling the code we still manage to improve, as seen in chapter 5.

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:

```
for(int i = 0; i < n; i = i + 4){
  result[i] = 0;
  result[i+1] = 0;
  result[i+2] = 0;
  result[i+3] = 0;
  tmp1 = result[i];
  tmp2 = result[i+1];
  tmp3 = result[i+2];
  tmp4 = result[i+3];
  for(int j = 0; j < n; j = j + 4){
    tmp1 += matrix[i][j]*vector[j];
    tmp1 += matrix[i][j+1]*vector[j+1];
    tmp1 += matrix[i][j+2]*vector[j+2];
    tmp1 += matrix[i][j+3]*vector[j+3];

    tmp2 += matrix[i+1][j]*vector[j];
    tmp2 += matrix[i+1][j+1]*vector[j+1];
    tmp2 += matrix[i+1][j+2]*vector[j+2];
    tmp2 += matrix[i+1][j+3]*vector[j+3];

    tmp3 += matrix[i+2][j]*vector[j];
    tmp3 += matrix[i+2][j+1]*vector[j+1];
    tmp3 += matrix[i+2][j+2]*vector[j+2];
    tmp3 += matrix[i+2][j+3]*vector[j+3];

    tmp4 += matrix[i+3][j]*vector[j];
    tmp4 += matrix[i+3][j+1]*vector[j+1];
    tmp4 += matrix[i+3][j+2]*vector[j+2];
    tmp4 += matrix[i+3][j+3]*vector[j+3];
  }
```

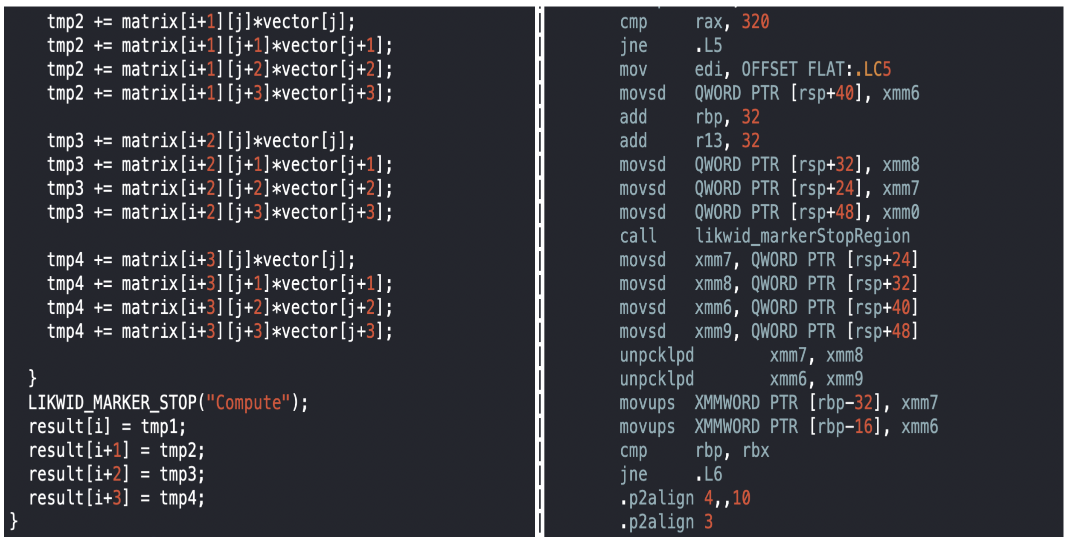
```
    result[i] = tmp1;
    result[i+1] = tmp2;
    result[i+2] = tmp3;
    result[i+3] = tmp4;
}
```

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 $\frac{32}{160}\frac{Flops}{Bytes}$. In the dividend we have 32 operations. In the divisor we get 160 by seeing that we fetch and load $(16 + 4)$ double elements, thus getting 160 bytes. Then the operational intensity is $\frac{1}{5}\frac{Flops}{Bytes}$, multiplied by the Bandwidth we get 1.2 $\frac{Gflops}{s}$.

4.1 Analysis of Assembly code

It is also useful to take a look at how the compiler arranges instructions at a lower level. Below, a side by side comparison of a snippet of C code and Assembly code:



```
    tmp2 += matrix[i+1][j]*vector[j];
    tmp2 += matrix[i+1][j+1]*vector[j+1];
    tmp2 += matrix[i+1][j+2]*vector[j+2];
    tmp2 += matrix[i+1][j+3]*vector[j+3];

    tmp3 += matrix[i+2][j]*vector[j];
    tmp3 += matrix[i+2][j+1]*vector[j+1];
    tmp3 += matrix[i+2][j+2]*vector[j+2];
    tmp3 += matrix[i+2][j+3]*vector[j+3];

    tmp4 += matrix[i+3][j]*vector[j];
    tmp4 += matrix[i+3][j+1]*vector[j+1];
    tmp4 += matrix[i+3][j+2]*vector[j+2];
    tmp4 += matrix[i+3][j+3]*vector[j+3];

  }
  LIKWID_MARKER_STOP("Compute");
  result[i] = tmp1;
  result[i+1] = tmp2;
  result[i+2] = tmp3;
  result[i+3] = tmp4;
}
```

```
        cmp     rax, 320
        jne     .L5
        mov     edi, OFFSET FLAT:.LC5
        movsd   QWORD PTR [rsp+40], xmm6
        add     rbp, 32
        add     r13, 32
        movsd   QWORD PTR [rsp+32], xmm8
        movsd   QWORD PTR [rsp+24], xmm7
        movsd   QWORD PTR [rsp+48], xmm0
        call    likwid_markerStopRegion
        movsd   xmm7, QWORD PTR [rsp+24]
        movsd   xmm8, QWORD PTR [rsp+32]
        movsd   xmm6, QWORD PTR [rsp+40]
        movsd   xmm9, QWORD PTR [rsp+48]
        unpcklpd        xmm7, xmm8
        unpcklpd        xmm6, xmm9
        movups  XMMWORD PTR [rbp-32], xmm7
        movups  XMMWORD PTR [rbp-16], xmm6
        cmp     rbp, rbx
        jne     .L6
        .p2align 4,,10
        .p2align 3
```

The easiest piece to recognize is the call to the function *likwid marker stop*. In our source code, after that call we assign 4 double values to specific entries of the result vector. We can see in the Assembly code that we have 4 **movsd** calls, which moves a scalar double-precision floating-point value to another location.
A few lines after that, we can see the word **cmp** which stands for compare, this is checking the condition on the for loop, if it satisfied it jumps at the beginning again with the call **jne**.

Chapter 5

Eigen implementation and comparison

Let's now see how the the Eigen library is doing on the same hardware. Below four tables that represent the data gathered from the simulation, every number is expressed in MFlops/s.

O0	naive	with_tmp	loop unrolling	eigen
n = 40	195	204	313	59
n = 400	328	332	453	70
n = 1000	327	328	445	94

O1	naive	with_tmp	loop unrolling	eigen
n = 40	302	255	882	464
n = 400	948	345	2346	2339
n = 1000	740	344	1544	1922

O2	naive	with_tmp	loop unrolling	eigen
n = 40	313	315	926	455
n = 400	941	937	3046	2297
n = 1000	720	704	1523	1958

O3	naive	with_tmp	loop unrolling	eigen
n = 40	315	321	915	383
n = 400	936	961	2935	3429
n = 1000	761	766	1538	2024

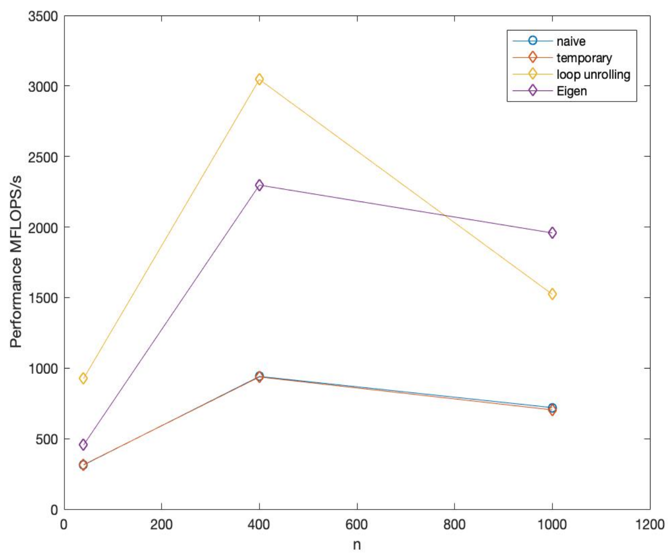
Below, another table containing the theoretical performance $P[\frac{Gflops}{s}]$ achievable for each algorithm:

naive	with_tmp	loop unrolling
0.325	0.325	1.2

Let's see what are the main takeaways from the data.
For $n = 40$, Eigen struggles against our best implementation for every flag by a pretty significant margin.
The code performs better with $n = 400$ than with $n = 40$, which seems strange at first because L1 cache should be faster than L2. However it seems to be a known issue that for dense matrix product the gcc compiler does not to exploit all the performance. I am going to analyze this issue in greater detail in the next section.

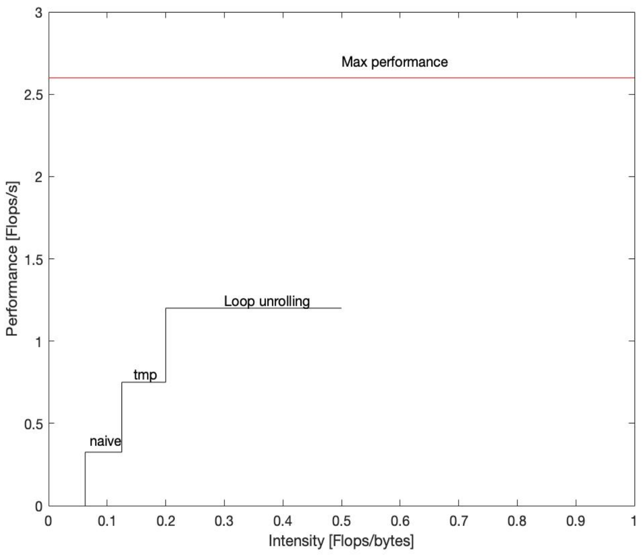
Unsurprisingly, the code runs faster with $n = 400$ than with $n = 1000$. One reason is the fact that the elements of the latter are saved in memory. We see that Eigen performs the best when the compiler setting are turned to the maximum. Otherwise, loop unrolling performs very well.

It is very interesting to see how much performance we gained from the first implementation to the third one. It is summarized in the graph below where the data is taken from the -O2 compilation.



5.1 Roofline graph

The next graph shows the roof line model:



Based on the model we chose, our performance should not have surpassed $1200\frac{MFlops}{s}$. However our code reaches on average a maximum performance of 2935 $\frac{MFlops}{s}$ and Eigen a staggering 3429 $\frac{MFlops}{s}$. The results are different from expected probably because in the first memory levels the roofline is imprecise. Moreover, the operating system may decide to up the frequency of the CPU for a short time if it is deemed useful for the task at hand.

5.2 In-depth analysis of $n = 40$ performance

With $n = 40$, we have the big advantage of saving everything in the L1 cache. However, it is hard to use our metrics to appreciate the gain because we have very small execution time, in the order of 10^{-5}. Moreover, LIKWID performs heavy approximation in its calculations.
I addressed this problem with two changes:

- Artificially increase the execution time with another for loop which does not perform any additional operation.
- Self made calculation of MFlop/s to avoid unwanted approximations.

Even with these changes, we get the following results when running the code with $O3$ flag:

- **n = 40 k = 100**: 3913 MFlops/s
- **n = 400 k = 1**: 3650 MFlops/s

The number of MFlops/s obtained is too large, even accounting for turbo boost capabilities.
The problem stems for the fact that increasing the number of operations does not scale well with the matrix dimension change, even though it approximately should.
This problem still remains open, and further investigations should be made, even regarding the behaviour of the compiler with small dense matrices.

Chapter 6

Parallel computations using OpenMP

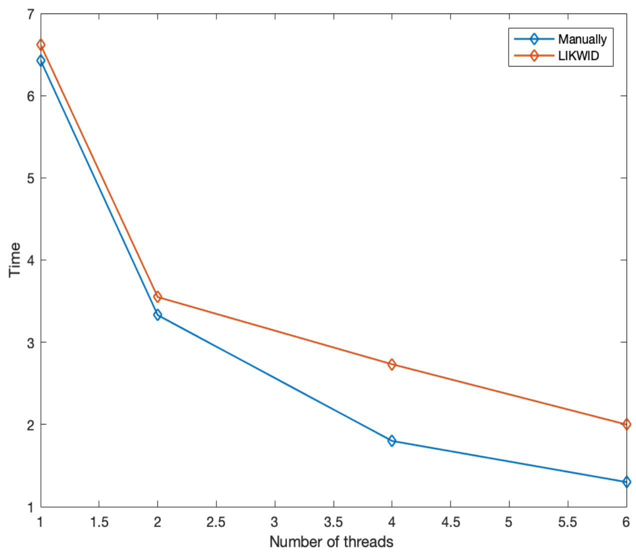
Open Multi-Processing is an application programming interface that supports multi-platform shared memory multiprocessing. It is used for parallelism within a multi-core node.
It is based on threads, which are processes that can share memory.
They communicate with each other via reading and writing shared data. Each thread proceeds through program instructions independently of other threads, that means it is important to ensure that actions on shared variables occur in the correct order.

In many applications, loops are the main source of parallelism. If the iterations can be done in any order, it is possible to share the iterations between threads, for instance: thread #1 does the even iterations and thread #2 does the odd. After the loop is done, we can merge together the result of the two loops. In theory, we managed to do the same task in half the time.

OpenMP is called with a language construct named *pragma*. Pragmas are used for the creation of the parallel region and for specifying the private variables inside such region. Below an example taken from the code:

```
#pragma omp parallel
{
  LIKWID_MARKER_THREADINIT;
  LIKWID_MARKER_START("Compute");
  #pragma omp for private(i,j) collapse(2)
   for(i = 0; i < n; i++){
     for(j = 0; j < n; j++){
        result[i] = 0;
        result[i] += matrix[i][j]*vector[j];
     }
   }
```

The first pragma initializes the pragma region. The second one initializes the parallel for with private variables i and j. The naive code is runwith $n = 50000$, $O3$ flag, with number of threads 1,2,4,6 to appreciate the difference in run time. Below, a graph showing the difference in run time and also how LIKWID calculates the run time:



The time calculated manually at code level is way more precise than the one calculated using LIKWID API's. The reason for this is that calling LIKWID marker functions adds unwanted complexity to the kernel, thus registering a much higher time.

Chapter 7

Future directions

There are still a few topics left to analyze in more detail.
First of all, we should investigate how changing the roof line model to the cache aware roof line model alters the expected result.

Then, we could also employ SIMD vectorization. The idea is to exploit data level parallelism: performing the same operation on multiple data point simultaneously. The GCC compiler already tries to vectorize wherever it is possible but sometimes it is better to explicitly write the operations to exploit the single instruction multiple data paradigm.

Lastly, it would be very interesting to further analyze how GCC behaves with matrix contained in the L1 cache. Closed source compilers like Intel's are expected to perform better in this situation.

Still, we managed to show that the most sophisticated implementation of the kernel significantly outperfmorms Eigen's with $n = 40$, independently of the chosen flag. The same can be said when $n = 400$ and $O2$ flag is chosen.

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.*

[5] Aleksandar Illic, Leonel Sousa, Frederico Pratas, *Cache aware roofline model: Upgrading the loft.*

[6] Hewlett-Packard Development Company, *Loop unroll and jam.*

[7] Numerical Algorithms Group Ltd, *How to make best use of the AMD Interlagos processor.*