POLITECNICO DI TORINO

Bachelor of Science in Mathematical Engineering

Bachelor degree thesis

Analysis of matrix vector product

Supervisor:
Prof. Stefano Berrone
Dr. Federico Tesser

Candidate:
Ludovico Bessi

July 2019

Ai miei genitori

Abstract

We made a comparison between a self-made implementation of the matrix vector product and the Eigen's library implementation. We found out that on equal hardware power our code runs Faster?

Contents

Chapter 1

Introduction

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:

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

. All the calculations are made having in mind the roofline model. We used LIKWID and STREAM to profile the code and gather insights.
STREAM is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for vector kernels. LIKWID is a set of command line tools supporting software developers, benchmarkers and application users to get the best performance on a given system.

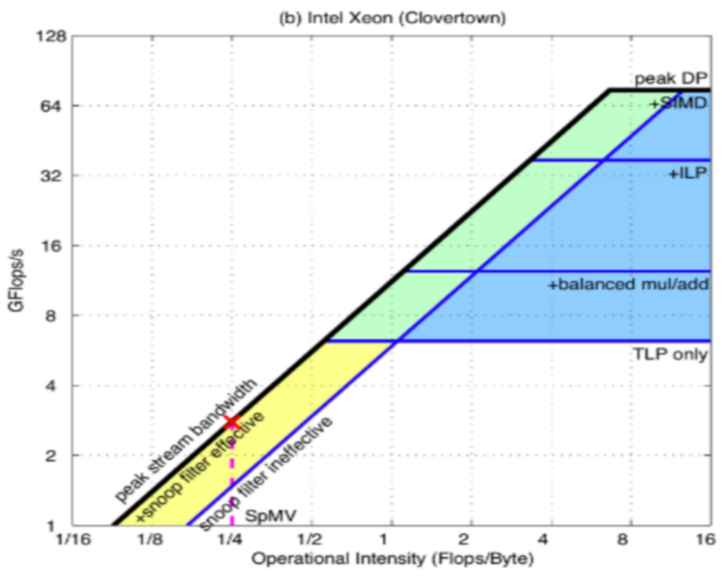
We went through different implementation of our kernel to get to the theoretical limit of our implementation using one core.

- Naive implementation
- Using a temporary variable to speed up cache lookup
- Loop unrolling to exploit sum and multiplication pipelines

With the last implementation, we managed to get to the theoretical limit. After that, we adressed scalability. That is, we looked at how our kernel worked in parallel on the supermegacomputer of politenico (inserire specifiche)

1.1 Roofline 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]

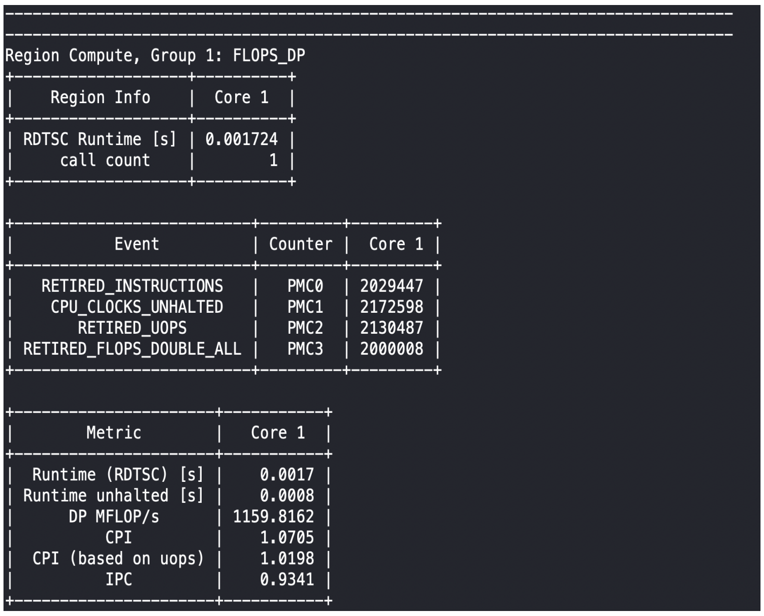


1.2 Eigen

Our main goal is to see how our code measures against Eigen code. 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.

1.3 LIKWID

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  |
+-------------------+----------+
| 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 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 unhalted [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*(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

Chapter 2

Naive implementation

At first we started with the simplest thing one could think of:

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

In this case: blablabla calcoli su P_peak computational intesity e grafico

Chapter 3

Using a temporary variable to speed up cache lookup

Chapter 4

Loop unrolling to exploit sum and multiplication pipelines

Chapter 5

Parallel computations

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*