Analysis of dense matrix vector multiplication

Ludovico Bessi

Matematica per l'ingegneria
Politecnico di Torino

May 2019

Table of Contents

1 Introduction and tools used

2 Different implementations

3 Comparison with Eigen

4 OpenMP

5 Conclusions

Introduction

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

Introduction

- Kernel: result[i] = result[i] + A[i,j]*vector[i]
- Matrix size changing: $n_1 = 40$, $n_2 = 400$, $n_3 = 1000$

Introduction

- Kernel: result[i] = result[i] + A[i,j]∗vector[i]
- Matrix size changing: $n_1 = 40$, $n_2 = 400$, $n_3 = 1000$
- Different optimization flags

Introduction

- Kernel: result[i] = result[i] + A[i,j]∗vector[i]
- Matrix size changing: $n_1 = 40$, $n_2 = 400$, $n_3 = 1000$
- Different optimization flags
- What is Eigen? How to change the kernel to match its performance?

Tools

- Stream $\rightarrow$ Bandwidth 6000 MB/s

Tools

- Stream $\rightarrow$ Bandwidth 6000 MB/s
- LIKWID: topology and perfctr

Tools

- Stream $\rightarrow$ Bandwidth 6000 MB/s
- LIKWID: topology and perfctr
- Hardware: Interlagos 12-core Opteron processor.

Roofline model

- Performance model used to determine bottlenecks: bandwidth and algorithm performance.

Roofline model

- Performance model used to determine bottlenecks: bandwidth and algorithm performance.
- $P_{max} = f_{max} * \frac{n_{operations}}{n_{cycles}}$

Roofline model

- Performance model used to determine bottlenecks: bandwidth and algorithm performance.
- $P_{max} = f_{max} * \frac{n_{operations}}{n_{cycles}}$
- $P = min(P_{max}, \text{Bandwidth} * \text{Computational intensity})$

Roofline model

- Performance model used to determine bottlenecks: bandwidth and algorithm performance.
- $P_{max} = f_{max} * \frac{n_{operations}}{n_{cycles}}$
- $P = min(P_{max}, \text{Bandwidth} * \text{Computational intensity})$
- Computational intensity is algorithm dependent, the main focus is to maximize it.

Naive

- Simplest possible code, useful for benchmark.

Naive

- Simplest possible code, useful for benchmark.
- $P_{max} = \frac{2.6}{8}\frac{Gflops}{s}$, not exploiting pipelines of sum and product

Naive

- Simplest possible code, useful for benchmark.
- $P_{max} = \frac{2.6}{8}\frac{Gflops}{s}$, not exploiting pipelines of sum and product
- Computational intensity $= \frac{2}{32}\frac{Flops}{Bytes}$

Temporary variable

- Additional temporary variable in outer loop to speed up cache lookup

Temporary variable

- Additional temporary variable in outer loop to speed up cache lookup
- $P_{max} = \frac{2.6}{8} \frac{Gflops}{s}$, still not exploiting pipelines of sum and product

Temporary variable

- Additional temporary variable in outer loop to speed up cache lookup
- $P_{max} = \frac{2.6}{8}\frac{Gflops}{s}$, still not exploiting pipelines of sum and product
- Computational intensity $= \frac{2}{16}\frac{Flops}{Bytes}$

Loop unrolling

- Manually split two loops four by four to exploit pipelines of sum and product

Loop unrolling

- Manually split two loops four by four to exploit pipelines of sum and product
- $P_{max} = 2.6 \frac{Gflops}{s}$

Loop unrolling

- Manually split two loops four by four to exploit pipelines of sum and product
- $P_{max} = 2.6 \frac{Gflops}{s}$
- Computational intensity $= \frac{32}{160} \frac{Flops}{Bytes}$

Loop unrolling

- Manually split two loops four by four to exploit pipelines of sum and product
- $P_{max} = 2.6 \frac{Gflops}{s}$
- Computational intensity $= \frac{32}{160} \frac{Flops}{Bytes}$
- Peaked at assembly code to understand inner functionalities

Comparing the Mflops/s

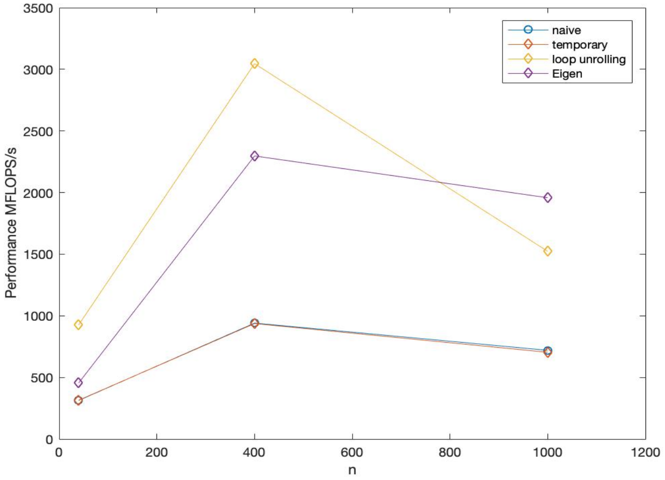
O0	naive	with_tmp	loop unrolling	eigen
n = 40	195	204	313	59
n = 400	328	332	453	70
n = 1000	327	328	445	94

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

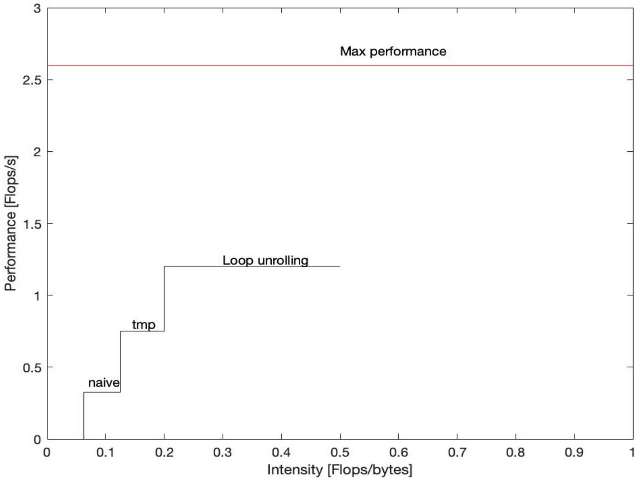
O2	naive	with_tmp	loop unrolling	eigen
n = 40	313	315	926	455
n = 400	941	937	3046	2297
n = 1000	720	704	1523	1958

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

Data visualization



Roofline graph



What is OMP?

- API that supports multi-platform shared memory multiprocessing.

What is OMP?

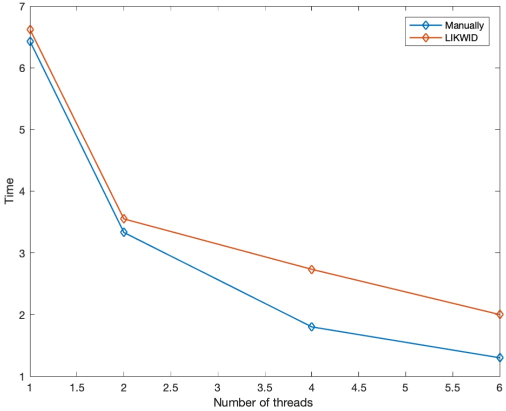
- API that supports multi-platform shared memory multiprocessing.
- Used for parallelism within a multi-core node.

What is OMP?

- API that supports multi-platform shared memory multiprocessing.
- Used for parallelism within a multi-core node.
- Based on threads: processes that can share memory.

Case study

With $n = 50000$, $O3$ flag and different number of threads:



Achievements

- Outperformed Eigen when $n = 40$

Achievements

- Outperformed Eigen when $n = 40$
- Outperformed Eigen when $n = 400$ and $O2$ flag.

Future directions

- Cache aware roof line model

Future directions

- Cache aware roof line model
- SIMD vectorization

Future directions

- Cache aware roof line model
- SIMD vectorization
- Understanding GCC behaviour when matrix is saved in L1