# Algorithm Engineering
# Project 2

Lasse Espeholt - 20093223
Kasper Nielsen - 20091182

March 16, 2013

**Implementation code and test results:** `http://github.com/kasper0406/AlgEng/project2`

## Contents

## 1 Introduction

In this project we have chosen to implement matrix multiplication. We will investigate how hardware architecture affects the running time of different algorithms, and try to optimize the performance of these algorithms with respect to the hardware.

## 2 Algorithms and data structures

In this section we will describe the different algorithms and memory layouts we have used.

### 2.1 Simple multiplication

#### 2.1.1 Row-based layout

We started out implementing the simple conventional $O(n^3)$ matrix multiplication algorithm where we stored the matrices using a row based layout.

Expectations of cache faults here....

We do not expect any significant amount of branch mispredictions.

#### 2.1.2 Combined row-based and column-based layout

In order to improve the number of cache faults, we have tried to use a column base layout in the right operand in the multiplication. We expect this to give us a bit better cache performance. This approach has the drawback of limiting a matrix only to be used on one side of a multiplication.

### 2.2 Recursive multiplication

For exploiting more kinds of layouts we implemented the recursive algorithm.

#### 2.2.1 Z-curve layout

The first idea was to use a Z-curved layout. One major advantage of this layout is that improves locality on all levels of the recursive multiplication. The drawback is that the index calculations are time consuming. On x86 we get approximately 50 bitwise operations each time. However, this can be improved by incrementally constructing the Z-curve numbers. The upcoming Intel Haswell architecture has support for bit permutations which will improve the situation.

In the recursive algorithm we used 4*4 as a base case for switching to the naive algorithm.

Hvorfor faar vi saa mange cache faults?

Lav dem inkrementielt?

#### 2.2.2 Tiled layout

A tiled layout is a compromise between locality and performance of index calculations. When the recursive algorithm reaches blocks of the same size as the tiles, it switches to the naive algorithm. And the naive algorithm functions without any modifications.

## 2.3 Special instructions

...

## 2.4 Parallelization

### 2.4.1 Combined row-based and column-based layout

The naive algorithm which is used for this combined is easily parallelizable because we can assign intervals of rows for each thread. The result matrix is stored in a row-based layout so writing is separated so cache thrashing should not be a problem.

...

# 3 Benchmarks

## 3.1 Test setup

Our program was written in C/C++ and compiled with Clang. All memory allocations were cache line boundary aligned. For parallelization we used C++11 cross-platform library `<thread>`.

All benchmarks were performed on a Linux desktop which has 4 GB ram and a Core i3 550 CPU with the following specification:

- 2 * 3.2 GHz (no Turbo boost)
- 2 * 32 KB L1 instruction cache
- 2 * 32 KB L1 data cache
- 2 * 256 KB L2 cache
- Shared 4 MB L3 cache
- 64 byte cache lines
- Inclusive caches
- The associativity for the cache levels are 8, 8 and 16 respectively

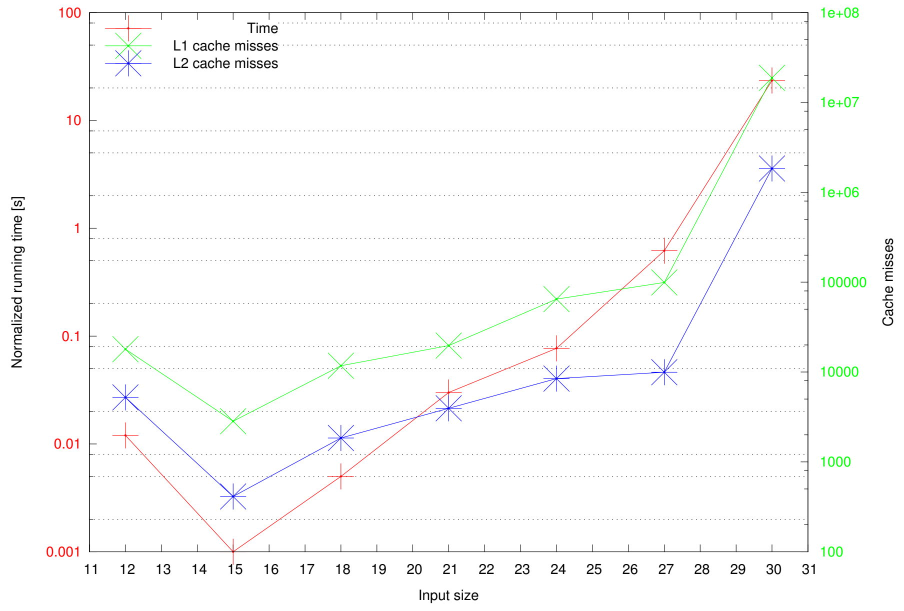
L2 cache faults and L3 cache faults were measured using Intel Performance Monitor Counter while branch mispredictions were measured using PAPI.

All tests were performed 5 times and the median was selected. The data in the matrices were randomly generated double precision floating points with an uniform distribution. The range of the data was 0 to 10,000.

## 3.2 Simple multiplication

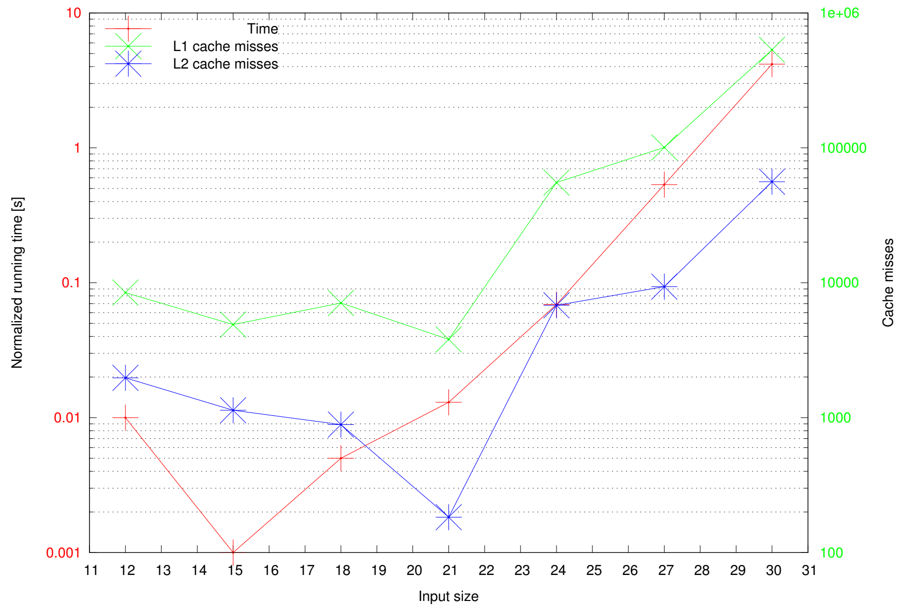
### 3.2.1 Row-based layout

3.2.1



### 3.2.2 Combined row-based and column-based layout

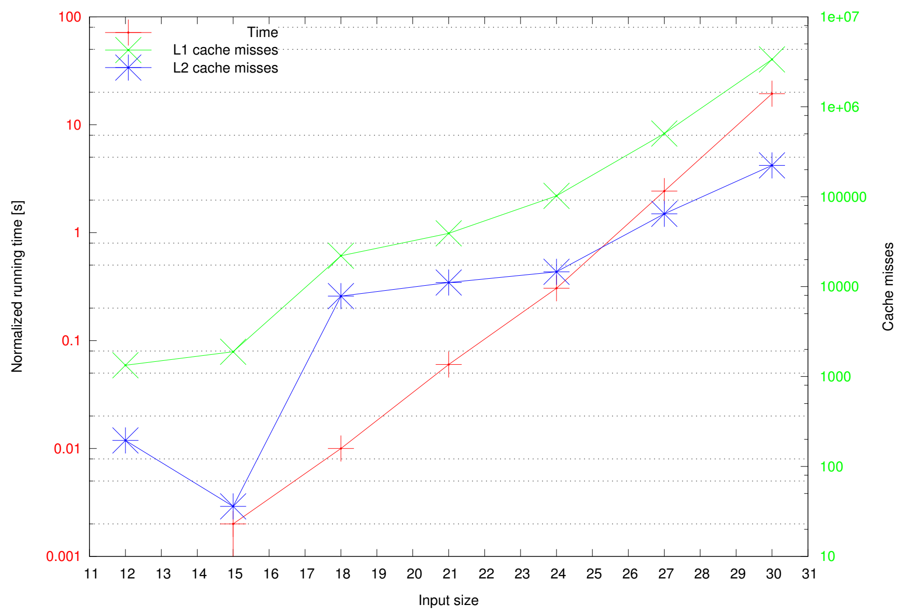
3.2.2



## 3.3 Recursive multiplication

### 3.3.1 Z-curve layout

3.3.1


Time
L1 cache misses
L2 cache misses
Normalized running time [s]
Cache misses
Input size
1e+07
1e+06