Algorithm Engineering
Project 2

Lasse Espeholt - 20093223
Kasper Nielsen - 20091182

March 21, 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.

In order to estimate the number of cache faults, we will for each row in the output matrix try to bound the number of cache faults encountered in order to compute it. Figure 2.1.1 illustrates the layouts and the multiplication process.

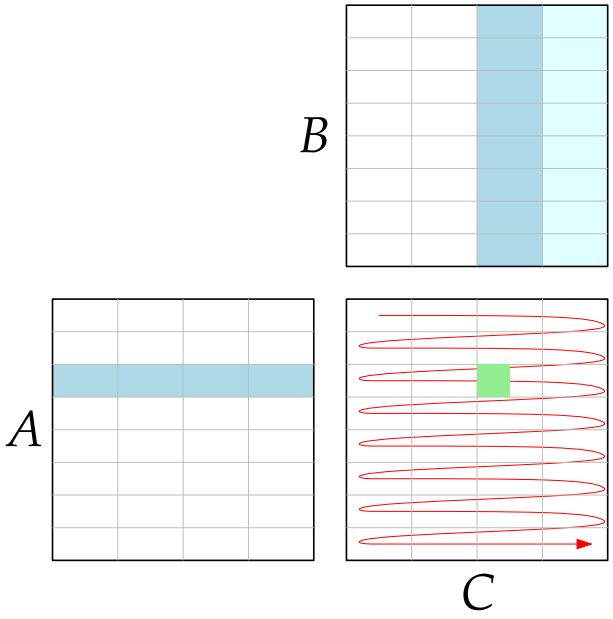


Figure 1: Illustration of multiplication with row-row layout.

We want to compute the matrix product $C = AB$. We do this by filling out all entries in C in a row-by-row and inside a row, column-by-column fashion. Hence, if the matrices are sufficiently small, then the current row of A will never be excluded from cache. In order for this to happen, we should have enough cache to store all the cache lines used in matrix B (to fill the current row). These are the cache lines marked with the darker blue color in Figure 2.1.1. Because the CPU uses an adjacent cache line

prefetcher, we also expect it to load the next cache line, hence we also expect the area marked with light blue to be in cache. If we let $\mathcal{M}$ denote the size of the cache in bytes, then we find that we can reuse the cache lines of C only if

$$(n + n \cdot \underbrace{8}_{\text{8 doubles in a cache line}} \cdot 2) \cdot 8 \leq \mathcal{M}. \iff n \leq \frac{\mathcal{M}}{136}.$$

Since the cache works in a LRU like fashion[citation?], we will get no benefit from the loaded cache lines of the B matrix if the cached elements does not fit. For example, the L2 cache, which is $256kb$, will begin to trash the values when

$$n \leq \frac{256 \cdot 1024}{136} \approx 1927.$$

We will analyze the expected number of cache faults in both situations.

We assume that the cache is reset after the computation of a row.

When computing a row in $C = AB$, we keep a row in A fixed, and vary the columns in B. Hence the row i A will always be in cache, initially causing $\frac{n}{B}$ cache faults. We compute all columns in the result row sequentially. Because our cache is big enough to hold $n + 1$ cache lines, we can hold all cache lines

We do not expect any significant amount of branch mispredictions.

Argue why this makes sense

Insert image illustrating this!

Argue!?

Expectation of cache faults here...

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. However, this problem can be mitigated by conversions which is an $O(n^2)$ operation.

We analyze the number as cache faults as before, but

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 we want to convert a coordinate to the position in the array. However, this can be improved by incrementally constructing the Z-curve numbers or by precomputing offsets at base cases. The upcoming Intel Haswell architecture has support for bit permutations which will improve the situation.

We ended up with precomputing the Z-curve offsets at the base case. That means the only penalty when we want to store or lookup a value is a level of indirection. In the recursive algorithm we used 8*8 as a base case for switching to the naive algorithm.

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. To improve cache locality a bit we used a row-based layout in the tiles for the left operand and a column-based layout in the tiles for the right operand.

2.3 Strassen

Strassen exploits some tricks with matrix multiplication such that it only uses 7 multiplications instead of 8 multiplications. It is a recursive algorithm by nature. The first step of the algorithm is the split the operands into 2*2 blocked matrix.

Instead of actually splitting the matrix into smaller matrices we have used a Z-curve layout. That means that the smaller matrices are able to just point to the data in the parent matrix while still preserving a proper layout themselves.

Strassen needs to have a large base case such that we do not spend too much time on the recursion and to improve cache locality at the base levels. To avoid the overhead of handling matrix multiplication with a level of indirection or by calculating Z-curve indices, we used a row-based and column-based layout at the base case for the left and right operand respectively.

The additions and subtractions in the algorithm only works on left operand block matrices or right operand block matrices. Therefore, we can use a single loop and no index calculations to add and subtract matrices. I.e. Z-curve indices are the same and we only add/subtract row-based base cases with row-based base cases etc.

hukomelse

2.4 SIMD instructions

...

Manglende præcision

2.5 Parallelization

L1 cachen er en write back (synchronized) cache saa det bliver maaske hurtigere ved at allokere stack-vis?

2.5.1 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.

...

2.5.2 Strassen

Strassen was parallelized by starting new threads at each of the 7 multiplications. This was done at 1 or 2 levels depending on the level of parallization we wanted. The last additions/subtractions and combine operations were not parallelized. However, the multiplications uses most of the time.

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 -1 to 1.

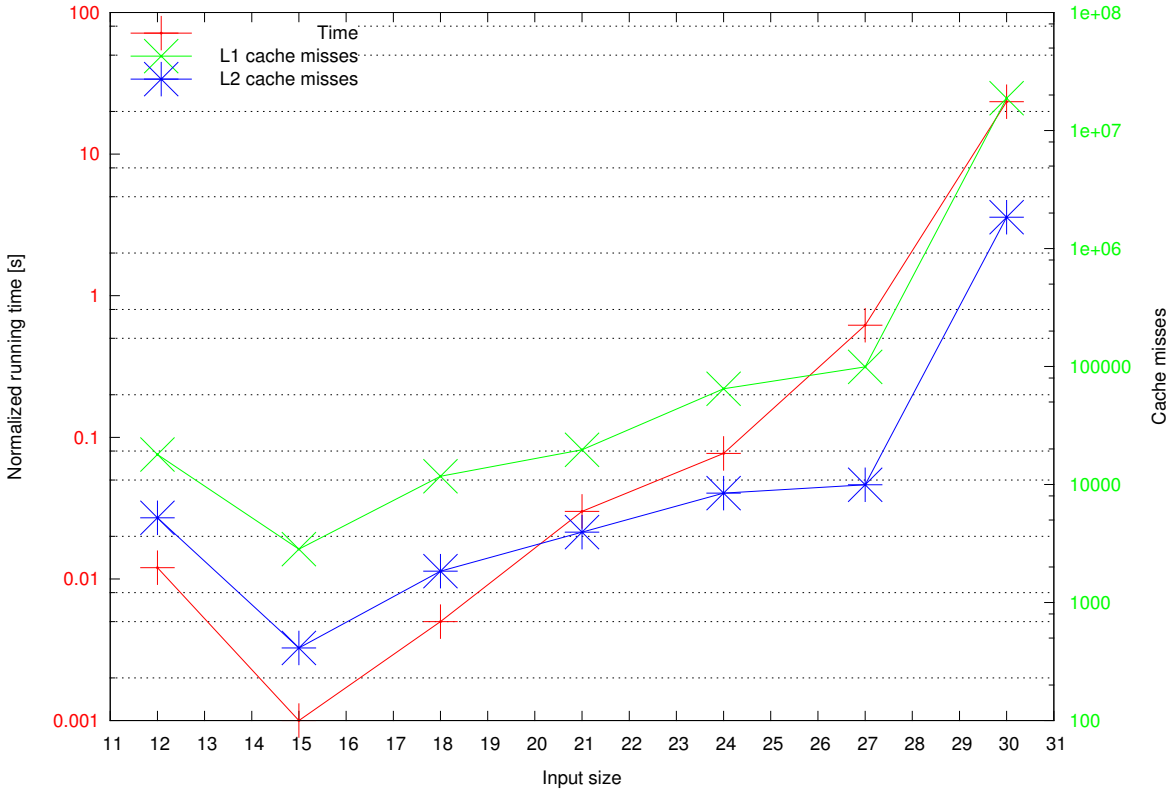
3.2 Simple multiplication

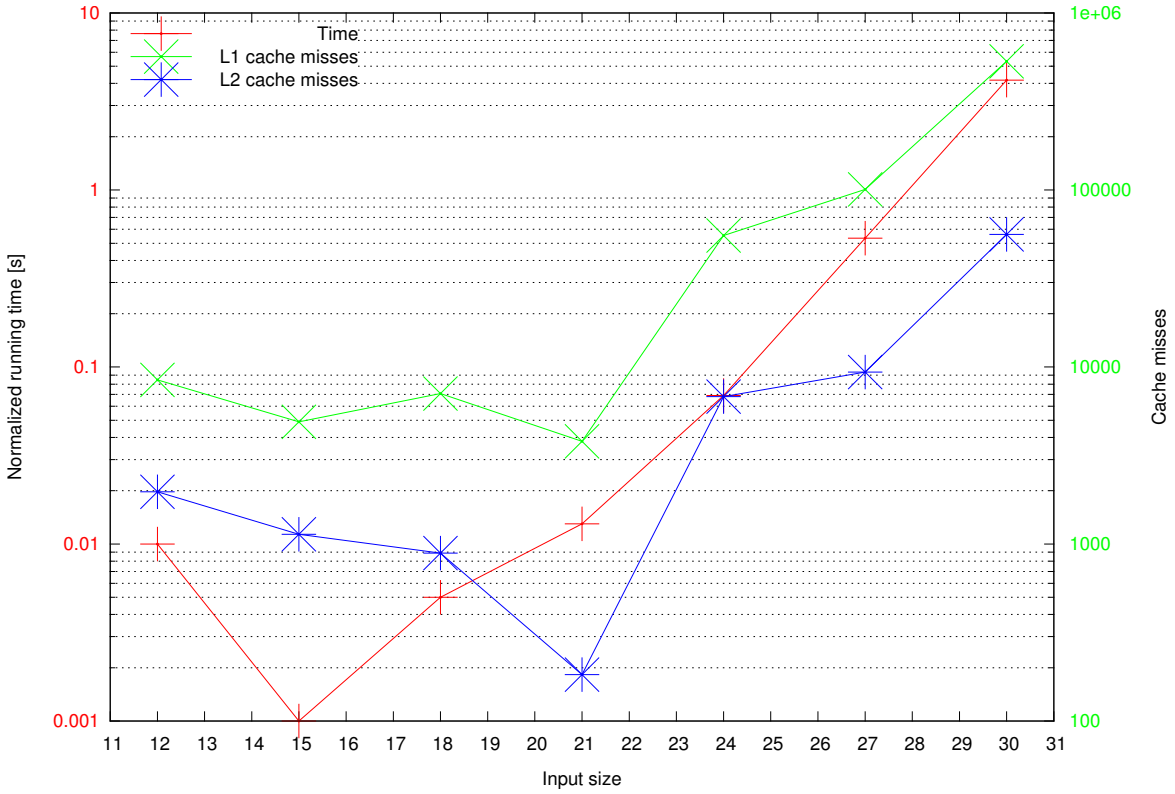
3.2.1 Row-based layout

3.2.1

3.2.2 Combined row-based and column-based layout

3.2.2


Time
L1 cache misses
L2 cache misses
Normalized running time [s]
Cache misses
Input size
100
10
1
0.1
0.01
0.001
1e+08
1e+07
1e+06
100000
10000
1000
100
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31


Time
L1 cache misses
L2 cache misses
Normalized running time [s]
Cache misses
Input size
10
1
0.1
0.01
0.001
1e+06
100000
10000
1000
100
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

3.3 Recursive multiplication

3.3.1 Z-curve layout

3.3.1