Lab 6: Caches

Matrix Multiplication Optimization

Objectives

This lab demonstrates some basic ideas in cache optimization and demonstrates it's importance in real world tasks.

Source code

The files on Learn should contain the following files:

src/
- benchmark_block.c
- benchmark_mul.c
- matrix.c
- matrix.h
- **matrix_mul.c** ← This is the file you will be editing
- test_mul.c

Makefile

Part 1 – the problem

An implementation of basic matrix multiplication has been given, the matrix multiplication algorithm used is an $O(n^3)$ algorithm – but practical aspects of current computer architecture have larger influences on performance.

At what point does this naïve implementation deviate from $O(n^3)$ performance? Why?

The executables used in this lab can be all built with the command “make”

What does the following tell us?

```
./benchmark_mul 50
```

Part 2 – data reorganization

We have seen that a naïve implementation of matrix multiplication suffers from certain problems. One way to try to optimize the implementation may be to rearrange the data.

Finish the implementation (in **matrix_mul.c**)

```
void matrix_mul_transposed(double *res, double *a, double *b, int n)
```

Test your solution by compiling with “make” and running

```
./test_mul
```

This will verify you have a correct solution. Afterwards you can benchmark the new implementation against the old one again with:

```
./benchmark_mul 50
```

How does transposing the second matrix(b) change the performance of the program? Why is this?

Part 3 – loop blocking

A well known **cache-aware** optimization is loop blocking. Where we restrict the working set of memory to fit in one of the CPU's caches. For matrix multiplication that means we restrict ourselves to work with a small (square) block in each of the matrices by limiting the extent of each loop. (We end up with 6 loops instead of 3).

Finish the implementation (in **matrix_mul.c**)

```
void matrix_mul_blocked(double *res, double *a, double *b, int n,
int block)
```

Test your solution by compiling with “make” and running (again)

```
./test_mul
```

Once you have verified your implementation is correct, you can again benchmark your solution (as before)

```
./benchmark_mul 50
./benchmark_block
```

In addition we have provided a second benchmark program to find the best block size. Run the **bechmark_block** program to determine the best block size. What is the optimal block size, and why?

Extensions

Can you optimize (one of) the matrix_mul routines even further? Are there other ways to rearrange the data to improve upon both loop blocking and transpose approaches? Can you use SSE instructions to futher speed up the algorithm?