ENCE360 Lab 5: Caches

Objectives

This lab demonstrates some basic ideas in cache optimization and demonstrates its importance in real world tasks. This lab specifically will focus on optimising matrix multiplication

The aims of this lab are for you to understand:

1. Programming principles that take advantage of computer caching architecture.
2. How to use knowledge of the specific cache layout of a given computer to speed up computation.

Preparation

As well as this handout, `lab5Cache.zip` should contain a `Makefile`, and the source files `benchmark_block.c`, `benchmark_mul.c`, `matrix.c`, `matrix.h`, `test_mul.c`, and `matrix_mul.c`. We will only be editing `matrix_mul.c`.

Once you've completed the tasks below, make sure you also complete the lab quiz.

Function `matrix_mul_basic()`

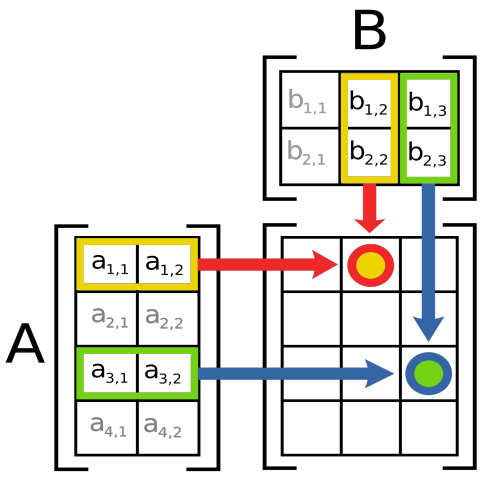

Figure 1: Two matrices are multiplied together to produce a single product matrix. The order of the matrices matters: the rows of the first matrix are combined with the rows of the second to produce the elements of the result.

If you're not familiar with matrix multiplication, it's worth having a look at some examples on the Wikipedia page.

An implementation of basic matrix multiplication has been given. This is an $O(n^3)$ matrix multiplication algorithm – faster algorithms do exist. However, practical aspects of computer architecture have larger influences on performance, so we'll stick with this basic algorithm.

At what point does this naive 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
```

Function `matrix_mul_transposed()`

We have seen that a naive implementation of matrix multiplication suffers from certain problems. One way to try to optimise the implementation may be to rearrange the data. In this case, we transpose one of the matrices.

Finish the implementation (in `matrix_mul.c`) of

```
void matrix_mul_transposed(double *res, double *a, double *b, size_t
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 matrix in this second function change the performance? Why is this?

Function `matrix_mul_blocked`

A well known cache-aware optimization is *loop blocking*. This is 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 nested loops instead of 3).

I **strongly** recommend drawing out a diagram of how the matrices will be subdivided; getting your head around this really does take a picture.

Finish the implementation (in `matrix_mul.c`) of

```
void matrix_mul_blocked(double *res, double *a, double *b, size_t n,
size_t 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
```

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