COSC3500 Tutorial 3: Experiments in matrix multiplication optimisation

17th August 2021

Matrix Multiply(1): naive method

```cpp
double A[n][n];
double B[n][n];
double C[n][n];

int trials = atoi(argv[1]);
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        A[i][j] = (double) rand() / (double) RAND_MAX;
        B[i][j] = (double) rand() / (double) RAND_MAX;
        C[i][j] = 0;
    }
}

std::vector<int> times;
for (int i = 0; i < trials; i++) {
    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start).count();
    times.push_back(duration);
}
```

Trials done at n = 2048

Runtime

Version	Method	Timing
1	Standard Implementation	242.06s

Matrix Multiply(2): Switch inner loops

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        for (int j = 0; j < n; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Runtime

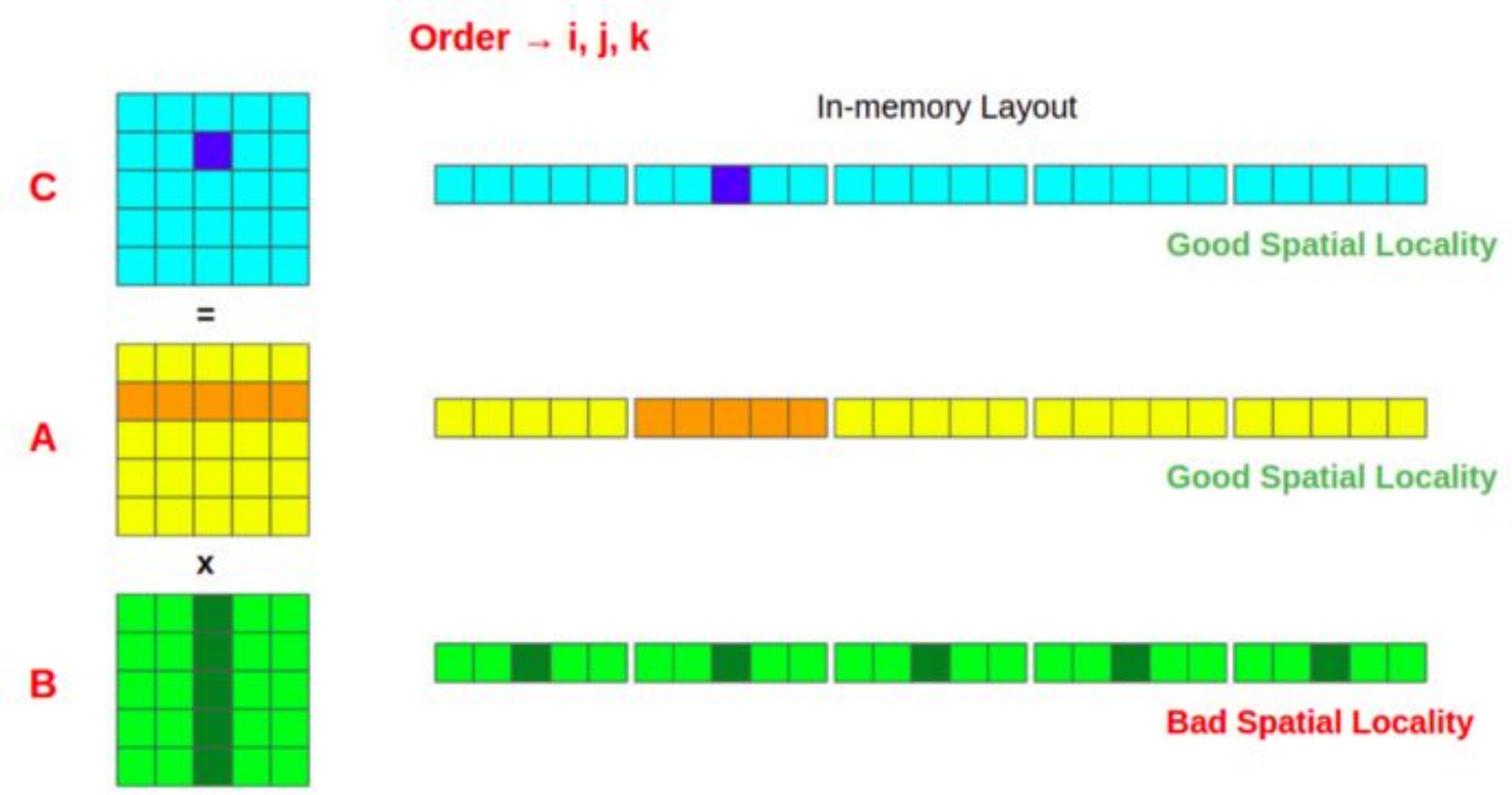
Version	Method	Timing
1	Standard Implementation	242.06s
2	Interchange Loop Order	93.20s

Remember, C uses row-major order for multi-dimensional arrays: first index is row, second index is column!

E.g. arr[i][j] indexes row i, column j.

Spatial Locality case 1

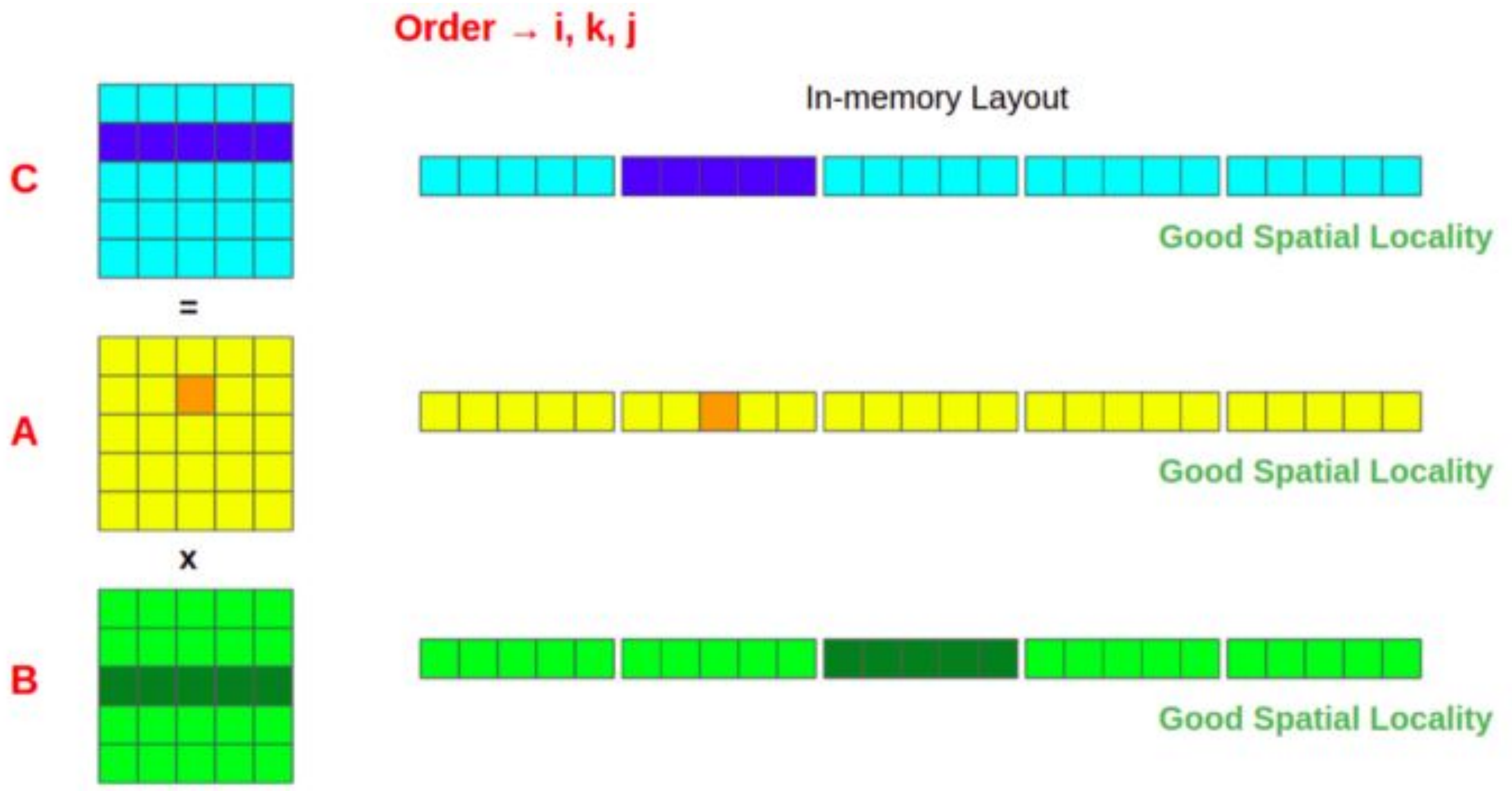
$$C[i][j] \mathrel{+}= A[i][k] * B[k][j]$$
$$i = 1,\ j = 2,\ k = 0..n$$



https://levelup.gitconnected.com/c-programming-hacks-4-matrix-multiplication-are-we-doing-it-right-21a9f1cbf53

Spatial Locality case 2

C[i][j] += A[i][k] * B[k][j]
i = 1, k = 2, j = 0..n



https://levelup.gitconnected.com/c-programming-hacks-4-matrix-multiplication-are-we-doing-it-right-21a9f1cbf53

Valgrind

Usage: *valgrind –tool=cachegrind args*

Example:

```
valgrind --tool=cachegrind ./matrix 1
```

```
==251==
==251== D   refs:       66,180,043,483  (61,391,524,032 rd   + 4,788,519,451 wr)
==251== D1  misses:      4,718,855,645  ( 4,717,280,497 rd   +     1,575,148 wr)
==251== LLd misses:      4,710,839,645  ( 4,709,265,294 rd   +     1,574,351 wr)
==251== D1  miss rate:             7.1% (           7.7%     +           0.0%  )
==251== LLd miss rate:             7.1% (           7.7%     +           0.0%  )
==251==
==251== LL refs:         4,718,857,147  ( 4,717,281,999 rd   +     1,575,148 wr)
==251== LL misses:       4,710,841,133  ( 4,709,266,782 rd   +     1,574,351 wr)
==251== LL miss rate:              1.8% (           1.8%     +           0.0%  )
```

https://stackoverflow.com/questions/20172216/how-do-you-interpret-cachegrind-output-for-caching-misses

Modified Loop Order

```
==259== D   refs:      28,503,779,515  (26,407,360,706 rd   + 2,096,418,809 wr)
==259== D1  misses:       253,851,805  (   252,276,657 rd   +     1,575,148 wr)
==259== LLd misses:       253,776,053  (   252,201,702 rd   +     1,574,351 wr)
==259== D1  miss rate:            0.9% (           1.0%     +           0.1%  )
==259== LLd miss rate:            0.9% (           1.0%     +           0.1%  )
==259==
==259== LL refs:          253,853,307  (   252,278,159 rd   +     1,575,148 wr)
==259== LL misses:        253,777,541  (   252,203,190 rd   +     1,574,351 wr)
==259== LL miss rate:             0.2% (           0.2%     +           0.1%  )
```

Runtime

Version	Method	Timing
1	Standard Implementation	242.06s
2	Interchange Loop Order	93.20s
3	O3 Optimisation Flag	4.31s

Why is –O3 so much faster?

- -O3 flag enables vectorisation by default
- Vectorisation is very powerful and the compiler could easily vectorise this code. Sometimes vectorisation is not possible or needs to be done manually

Matrix Multiply (3): blocking

```
for (int ii = 0; ii < n; ii += b) {
    for (int jj = 0; jj < n; jj += b) {
        for (int kk = 0; kk < n; kk += b) {
            for (int i = 0; i < b; i++) {
                for (int k = 0; k < b; k++) {
                    for (int j = 0; j < b; j++) {
                        C[i + ii][j + jj] += A[i + ii][k + kk] * B[k + kk][j + jj];
                    }
                }
            }
        }
    }
}
```

Animations of the different approaches:

Method 1 - Naive: https://www.youtube.com/watch?v=QYpH-847z0E

Method 2 - Loop interchange: https://www.youtube.com/watch?v=0u2K_dRLhWw

Method 3 - Blocking with loop interchange: https://www.youtube.com/watch?v=aMvCEEBIBto

Runtime

Version	Method	Timing
1	Standard Implementation	242.06s
2	Interchange Loop Order	93.20s
3	O3 Optimisation Flag	4.31s
4	More Cache Optimisation	2.73s

Summary of Methods

- Utilising computer hardware well (Show levels of cache)
- Compiler Flags
- Achieved ~5 times speedup with designing around caching
- Experiment a lot! (We can still do better)