# Performance Analysis on Modern CPUs

# Exercise 2

In this exercise, we will explore top-down microarchitecture analysis using a modified matrix multiplication sample from LLVM. The original uses integer matrix multiplication, while our modified version uses **`double`** instead of **`int`** for the matrix elements. We also use a different method to initialize the matrices that will be multiplied together. The original code sample can be found on GitHub at the link below:

https://github.com/llvm-mirror/test-suite/blob/master/SingleSource/Benchmarks/Shootout/matrix.c

Our modified version is in the code listing 1 in the appendix of this document. It implements a naïve matrix multiplication without considering performance. We will analyze its problems with **`perf`** and improve it to make full use of the available hardware resources.

## 1 Setup

The setup for this problem is quite simple. Please download the source code from Indico or copy it from the appendix into a file named `matrix.c` in `/home/$USER/exercise2`. Then, compile it with GCC and run it as shown below

```
$ gcc -O2 -g -o matrix matrix.c
$ time ./matrix
      1.0240e+03       1.0240e+03       2.0000e+00       2.0000e+00
 -5.18490000e+04      -5.1849e+04       5.4096e+04      -3.7166e+06

real    0m5.717s
user    0m5.695s
sys     0m0.012s
```

## 2 Top-Down Microarchitecture Analysis

The program initializes, then multiplies two $1024 \times 1024$ matrices. By default, it does it only once, but the program takes an optional argument to specify a number of repetitions of the multiplication, which will become handy when we optimize the code to make it run for longer to make it easier to analyze by sampling with **`perf`**. Let's start by looking at a simple top-down microarchitecture analysis with **`perf stat`**:

```
$ perf stat --topdown -- ./matrix
      1.0240e+03       1.0240e+03       2.0000e+00       2.0000e+00
 -5.18490000e+04      -5.1849e+04       5.4096e+04      -3.7166e+06

 Performance counter stats for './matrix':

 %  tma_frontend_bound      %  tma_retiring %  tma_backend_bound %  tma_bad_speculation
                  0.2                 16.4                 83.4                  0.1

       5.708644492 seconds time elapsed

       5.685755000 seconds user
       0.012968000 seconds sys
```

We can see that performance is bound by the backend of the CPU. We have two main sub-categories in the backend: core bound, and memory bound. To distinguish between them, let's look further down the hierarchy by counting level 2 events for the backend bound category next.

```
$ perf stat --topdown --td-level 2 -- ./matrix
      1.0240e+03       1.0240e+03       2.0000e+00       2.0000e+00
 -5.18490000e+04      -5.1849e+04       5.4096e+04      -3.7166e+06

 Performance counter stats for './matrix':

 %  tma_heavy_operations %  tma_light_operations %  tma_fetch_latency %  tma_fetch_bandwidth
                     3.0                    13.4                   0.1                   0.1
 %  tma_branch_mispredicts %  tma_machine_clears    %  tma_core_bound  %  tma_memory_bound
                     0.0                    0.0                  34.9                  48.5

       5.748132833 seconds time elapsed
       5.686684000 seconds user
       0.014830000 seconds sys
```

We are memory bound for almost 50% of the CPU cycles, which is indicative of a bad memory access pattern. Another way of looking into this is by asking for just the two metrics we want with **`perf stat`**, or asking for more detailed information:

```
$ perf stat -M tma_core_bound,tma_memory_bound -- ./matrix
      1.0240e+03       1.0240e+03       2.0000e+00       2.0000e+00
 -5.18490000e+04      -5.1849e+04       5.4096e+04      -3.7166e+06

 Performance counter stats for './matrix':

       143,002,093      CPU_CLK_UNHALTED.REF_XCLK        #     34.9 %  tma_core_bound
                                                  #     48.5 %  tma_memory_bound         (74.97%)
        11,989,539      EXE_ACTIVITY.BOUND_ON_STORES                                    (74.97%)
       167,661,031      IDQ_UOPS_NOT_DELIVERED.CORE                                     (74.97%)
         8,225,475      INT_MISC.RECOVERY_CYCLES                                        (66.66%)
     6,369,109,382      EXE_ACTIVITY.1_PORTS_UTIL                                       (66.68%)
       142,980,612      CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE                                (66.70%)
    18,281,790,098      CPU_CLK_UNHALTED.THREAD                                         (75.02%)
     9,362,434,281      CYCLE_ACTIVITY.STALLS_MEM_ANY                                   (75.02%)
    11,939,607,426      UOPS_RETIRED.RETIRE_SLOTS                                       (75.02%)
     9,400,301,925      CYCLE_ACTIVITY.STALLS_TOTAL                                     (75.02%)
     1,991,810,157      EXE_ACTIVITY.2_PORTS_UTIL                                       (75.00%)
    11,948,695,700      UOPS_ISSUED.ANY                                                 (74.98%)

       5.754115617 seconds time elapsed
       5.695839000 seconds user
       0.013835000 seconds sys
```

```
$ perf stat -d -- ./matrix
      1.0240e+03       1.0240e+03       2.0000e+00       2.0000e+00
 -5.18490000e+04      -5.1849e+04       5.4096e+04      -3.7166e+06

 Performance counter stats for './matrix':

          5,708.98 msec task-clock                #    1.000 CPUs utilized
                 9      context-switches          #    1.576 /sec
                 0      cpu-migrations            #    0.000 /sec
             6,221      page-faults               #    1.090 K/sec
    18,240,680,127      cycles                    #    3.195 GHz
    10,843,345,295      instructions              #    0.59  insn per cycle
     1,092,375,551      branches                  #  191.343 M/sec
         1,134,029      branch-misses             #    0.10% of all branches
     3,238,096,765      L1-dcache-loads           #  567.193 M/sec
     1,353,674,537      L1-dcache-load-misses     #   41.80% of all L1-dcache accesses
       135,876,181      LLC-loads                 #   23.800 M/sec
           119,640      LLC-load-misses           #    0.09% of all LL-cache accesses

       5.709588950 seconds time elapsed
       5.682728000 seconds user
       0.016936000 seconds sys
```

We can see that 41.8% of all data cache accesses are missing the first level cache (L1). If you'd like to further explore top-down microarchitecture analysis with **`perf`**, you may also try the commands below to list different metrics and try them with the **`matrix`** binary.

```
$ perf list metricgroup
$ perf list --long-desc --details metric
$ perf list --long-desc --details Topdown
$ perf list --long-desc --details tma
```

For example, to further explore the memory bound category, you can run **`perf stat`** like shown below:

```
$ perf stat -M MemoryBound -- ./matrix 2>&1 | grep tma
       142,741,196      CPU_CLK_UNHALTED.REF_XCLK        #      0.0 %  tma_pmm_bound
                                                 #     15.6 %  tma_l2_bound
                                                 #      0.2 %  tma_dram_bound          (18.64%)
       142,776,180      CPU_CLK_UNHALTED.REF_XCLK        #      0.0 %  tma_store_bound
                                                 #     18.2 %  tma_l3_bound            (10.43%)
       142,767,037      CPU_CLK_UNHALTED.REF_XCLK        #     17.1 %  tma_l1_bound             (18.78%)
         5,578,818      MEM_LOAD_RETIRED.FB_HIT          #    14.64 tma_info_memory_load_miss_real_latency
         ↪  (12.52%)
    12,844,868,438      L1D_PEND_MISS.PENDING_CYCLES     #     1.24 tma_info_memory_mlp        (10.43%)
```

So the bad memory access pattern causes us to frequently miss both L2 and L3 caches in addition to L1.

## 3 Performance Optimization

Now, use the slides for lecture 3 as your reference, and optimize the matrix multiplication code, re-running the top-down microarchitecture analysis at each step. The optimizations to be performed are:

- Optimize memory access pattern to avoid data cache misses
- Parallelize with OpenMP, to make use of all available processors
- Enable vectorization at compile-time, by adding the appropriate compile options
- Enable fused multiply-add instructions to further enhance performance

If you still have time at the end, you may also try the following extra optimizations:

- Use cache blocking to optimize usage of lower level caches
- Collapse one or more of the loop levels to improve OpenMP scheduling
- Use page-aligned and/or contiguous memory allocations for each matrix
- Try different scheduling affinity setups with OpenMP (one vs two sockets)

Use **`perf`** to observe how many of the floating point operations have been vectorized by the compiler. What was the maximum speedup you were able to achieve relative to the initial implementation? What is the total CPU utilization of the optimized matrix multiplication? What is the final percentage of CPU cycles in the retiring category? How do the initial and optimized implementations compare in terms of power usage? How many instructions per cycle did you achieve after optimizations?

## Appendix

```c
#include <float.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

double **mkmatrix(int rows, int cols) {
        int i, j;
        double **m = (double **)malloc(rows * sizeof(double *));
        for (i = 0; i < rows; i++) {
                m[i] = (double *)malloc(cols * sizeof(double));
                for (j = 0; j < cols; j++)
                        m[i][j] = (i < j ? -1.0 : 1.0) * (double) (i + j);
        }
        return (m);
}

void zeromatrix(int rows, int cols, double **m) {
        int i, j;
        for (i = 0; i < rows; i++)
                for (j = 0; j < cols; j++)
                        m[i][j] = 0;
}

void freematrix(int rows, double **m) {
        while (--rows > -1)
                free(m[rows]);
        free(m);
}

double **mmult(int rows, int cols, double **m1, double **m2, double **m3) {
        int i, j, k, val;
        for (i = 0; i < rows; i++) {
                for (j = 0; j < cols; j++) {
                        val = 0;
                        for (k = 0; k < cols; k++) {
                                val += m1[i][k] * m2[k][j];
                        }
                        m3[i][j] = val;
                }
        }
        return (m3);
}

int main(int argc, char *argv[]) {
        int i, n;

        n = ((argc == 2) ? atoi(argv[1]) : 1);

        double **m1 = mkmatrix(SIZE, SIZE);
        double **m2 = mkmatrix(SIZE, SIZE);
        double **mm = mkmatrix(SIZE, SIZE);

        zeromatrix(SIZE, SIZE, mm);

        for (i = 0; i < n; i++)
                mm = mmult(SIZE, SIZE, m1, m2, mm);

        printf("%16.4e %16.4e %16.4e %16.4e\n", mm[0][0], mm[0][1], mm[1][0], mm[1][1]);
        printf("%16.4e %16.4e %16.4e %16.4e\n", mm[0][SIZE - 1], mm[0][SIZE - 1],
                                                mm[SIZE-1][0], mm[SIZE-1][SIZE-1]);

        freematrix(SIZE, m1);
        freematrix(SIZE, m2);
        freematrix(SIZE, mm);
        return (0);
}
```

Listing 1: Simple matrix multiplication benchmark.