# High Performance Computing
# Anthony Trubiano
# Assignment 2 (due March 11, 2019)

**Note:** All computations were performed with an Intel Core i7-8750H Processor with base frequency 2.20 GHz. The maximum main memory bandwidth is 41.8 GB/s. At 16 double precision operations per cycle, the theoretical max flop rate would be about 35.2 GFlops/s. It has $6$ cores and can reach $12$ threads through hyper-threading.

1. **Optimizing matrix-matrix multiplication.** We test two implementations of matrix-matrix multiplication, a row based and column based approach. We then implement a blocking approach and parallelize this to compare to the naive approaches. The results are summarized in the following tables.

| | Row Based | | Column Based | |
|---|---|---|---|---|
| $N$ | Flop Rate | Memory Rate | Flop Rate | Memory Rate |
| 16 | 3.72 | 59.6 | 6.75 | 108 |
| 256 | 0.5 | 7.8 | 6.9 | 110 |
| 1024 | 0.26 | 4.22 | 5.3 | 84 |
| 1984 | 0.22 | 3.54 | 4.6 | 73 |

**Table 1:** The measured flop rate (GFlops/s) and memory access rate (GB/s) for the naive matrix-matrix multiplication codes, MMult1.cpp, ran on $N \times N$ matrices, using the -O3 compiler flag.

From this table, we see the column based approach outperforms the row based approach significantly. This is because the matrices are stored by column, so in the column based approach, we get cache hits most of the time whereas the row based approach gets constant cache misses, except in the case of small matrices. We note a significant drop in performance around $N = 850$, where the columns can no longer fit in L1 cache.

| | Block Size 32 | | Block Size 32 - Parallel | |
|---|---|---|---|---|
| $N$ | Flop Rate | Memory Rate | Flop Rate | Memory Rate |
| 32 | 6.73 | 107 | 4.2 | 68.5 |
| 320 | 6.97 | 111 | 36.6 | 585 |
| 1120 | 6.61 | 106 | 39 | 631 |
| 1984 | 6.60 | 100 | 37.5 | 600 |

**Table 2:** The measured flop rate (GFlops/s) and memory access rate (GB/s) for the block matrix-matrix multiplication codes, MMult1.cpp, ran on $N \times N$ matrices, using the -O3 compiler flag.

Next, we implemented blocked matrix-matrix multiplication and experimented with block size. We find that performance does not drop off in the $N = 800$ region, remaining mostly constant up to $N = 2000$. Experimenting with block size in multiples of $4$, we find that performance increases up to block size $32$, then begins to decrease again. The processor has an L1 cache with $32,000$ bytes, and we can compute that three $32 \times 32$ matrices of doubles takes $24,576$ bytes. This is the biggest block size that can be stored in L1 cache, hence the performance drop after. We then parallelize using this block size and see improvements up to $6$ times for the larger matrix size, using 12 threads (hyper-threaded). This is on par with the theoretical peak flop-rate.

2. **OpenMP version of 2D Jacobi/Gauss-Seidel Smoothing.** Here we implement an OpenMP version of the Jacobi and red-black Gauss-Seidel methods to solve a linear system arising from the discretization of the 2D Laplacian. Setting the right hand side vector equal to $1$, we apply our schemes to the Poisson equation in a box, with zero Dirichlet boundary conditions. We measured the time taken to complete $50,000$ iterations of each method for variable matrix size $N$, and variable numbers of threads. The results are in the following table:

| | Jacobi | | | | Gauss-Seidel | | | |
|---|---|---|---|---|---|---|---|---|
| | Number of Threads | | | | Number of Threads | | | |
| $N$ | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 |
| 25 | 0.16 | 0.12 | 0.1 | 0.1 | 0.19 | 0.15 | 0.14 | 0.14 |
| 50 | 0.61 | 0.37 | 0.26 | 0.27 | 0.62 | 0.36 | 0.23 | 0.23 |
| 100 | 2.4 | 1.38 | 0.88 | 0.88 | 2.31 | 1.23 | 0.67 | 0.64 |
| 200 | 9.9 | 5.47 | 3.3 | 3.1 | 9.11 | 4.7 | 2.46 | 2.3 |
| 400 | 39.4 | 21.5 | 12.6 | 12.1 | 36.2 | 18.5 | 9.5 | 8.3 |

**Table 3:** Time taken (in seconds) for the Jacobi and Gauss-Seidel algorithms to perform $50,000$ iterations on an $N \times N$ matrix, for $1$, $2$, $4$, and $8$ threads.

From this table, we see that parallelizing leads to minimal improvement for small matrices, in the range of $N = 25$ to $N = 50$. As the matrices become larger, $N > 100$, we see that doubling the number of threads roughly halves the amount of time the algorithm takes, up to $4$ threads. After $4$ threads we see diminishing returns. This machine has $6$ cores, and gets $12$ threads through hyper-threading, put the performance increase after $6$ threads is minimal.