# Lab 1: CPU w/ OpenMP: General Matrix Multiplication (GEMM)

Name: Dennis Zang
UID: 704766877

(Results)

As we can see, we have performed very significant speedup by utilizing parallelism using OpenMP. To test the program under different workloads, we simply modified the line “const int kN = 4096;” in “gemm.h” for different matrix sizes. Listed below are the outputs and performance metrics of “./gemm”.

(“./gemm sequential”)
(kN = 1024)

```
Problem size: 1024 x 1024 x 1024
Initialize matrices a and b

Run sequential GEMM
Time: 3.47966 s
Perf: 0.617154 GFlops
```

(kN = 2048)

```
Problem size: 2048 x 2048 x 2048
Initialize matrices a and b

Run sequential GEMM
Time: 62.2281 s
Perf: 0.276079 GFlops
```

(kN = 4096)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run sequential GEMM
Time: 738.641 s
Perf: 0.18607 GFlops
```

(“./gemm parallel”)
(kN = 1024)

```
Problem size: 1024 x 1024 x 1024
Initialize matrices a and b

Run parallel GEMM with OpenMP
Time: 0.022507 s
Perf: 95.414 GFlops
```

(kN = 2048)

```
Problem size: 2048 x 2048 x 2048
Initialize matrices a and b

Run parallel GEMM with OpenMP
Time: 0.19088 s
Perf: 90.0035 GFlops
```

(kN = 4096)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with OpenMP
Time: 3.44096 s
Perf: 39.942 GFlops
```

(“./gemm parallel-blocked”)
(kN = 1024)

```
Problem size: 1024 x 1024 x 1024
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 0.046112 s
Perf: 46.571 GFlops
```

(kN = 2048)

```
Problem size: 2048 x 2048 x 2048
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 0.206438 s
Perf: 83.2205 GFlops
```

(kN = 4096)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 1.17189 s
Perf: 117.28 GFlops
```

(Optimizations for “parallel-blocked”)

Compared to the simple “parallel” case, in which we only performed a loop permutation between j and k and a little bit of caching, the gist of “parallel-blocked” is that we performed much more caching in order to exploit cache-hits as much as possible. Below is a clean copy of the program in “omp-blocked.cpp”:

```
#include "gemm.h"
#include <cstring>
#include <cstdlib>
#define BLOCK_SIZE 256
void GemmParallelBlocked(const float a[kI][kK], const float b[kK][kJ], float c[kI][kJ]) {
  for (int i = 0; i < kI; ++i) {
    std::memset(c[i], 0, sizeof(float) * kJ);
  }
  for(int ai = 0; ai < kI; ai += BLOCK_SIZE) {
    int aiiMax = (ai + BLOCK_SIZE) > kI? (kI - ai) : BLOCK_SIZE;
    for(int k = 0; k < kK; k += BLOCK_SIZE) {
      int kkMax = (k + BLOCK_SIZE) > kK? (kK - k) : BLOCK_SIZE;
      float aCache[BLOCK_SIZE][BLOCK_SIZE];
      #pragma omp parallel for schedule(guided) num_threads(8)
      for(int aii = 0; aii < aiiMax; ++aii) {
        for(int kk = 0; kk < kkMax; ++kk) {
          aCache[aii][kk] = a[ai + aii][k + kk];
        }
      }
      #pragma omp parallel for schedule(guided) num_threads(8)
      for(int bj = 0; bj < kJ; bj += BLOCK_SIZE) {
        int bjjMax = (bj + BLOCK_SIZE) > kJ? (kJ - bj) : BLOCK_SIZE;
        float bCache[BLOCK_SIZE][BLOCK_SIZE];
        for(int kk = 0; kk < kkMax; ++kk) {
          for(int bjj = 0; bjj < bjjMax; ++bjj) {
            bCache[kk][bjj] = b[k + kk][bj + bjj];
                        }
        }
        for(int aii = 0; aii < aiiMax; ++aii) {
          for(int kk = 0; kk < kkMax; ++kk) {
            float aConst = aCache[aii][kk];
          float *bCacheRow = bCache[kk];
      for(int bjj = 0; bjj < bjjMax; ++bjj) {
       c[ai + aii][bj + bjj] += aCache[aii][kk] * bCacheRow[bjj];
            }
          }
        }
      }
    }
  }
}
```

As we can see, we generally took the “parallel” code from “omp.cpp” (same order of i, k, and j and some frequently-used pointer variables/constants cached before the float multiplication loops). However, the main difference here is of course the matrix blocking, in which for each block in a, we cached 8 blocks of b so we can perform 8 multiplications at once (we would also cache from the same block row at once so that we can make use of temporal locality and hopefully get more cache hits this way).

The block size was chosen based mainly on experimentation. Originally, I would expect that a block size of 32x32 would be optimal, as 32 x 32 x 4 (the number of bytes to represent a float) x 8 threads would add up to 32768 bytes, or about 32KB (which is the total amount of L1d memory provided by the AWS server, as given by the lscpu command). However, this was quickly proven incorrect, as this number would only give a relatively small Gflops performance (smaller than the “parallel” case). As I was able to increase the block size some more, I eventually capped out around a block size of 256x256 (in which the total memory would total around 256 x 256 x 4 x 8 = 2097152 bytes, or 2097KB). This probably proves that the L2 and L3 caches also play an important role (1024KB and 33792KB respectively), but we must also balance out the frequency at which cache misses and loads would occur.

Below are the improvements for each optimization:

*(caching of local variables in lines 41 and 42): NaN

There doesn’t seem to be an improvement at all. The compiler probably optimized this code automatically anyways, as we compiled this project with the g++ -O3 flag.

*(caching a local, contiguous copy of each block)

The performance jumped quite a bit, from about 55GFlops to about 80 at the time I had tried it. The main reason I manually copied values from the original matrix to a contiguous sub-block is so that I can try to “force” the program to cache the block values into the L1d memory.

(Scalability with Number of Threads)

(1 thread)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 7.1925 s
Perf: 19.1086 GFlops
```

(2 threads)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 3.70989 s
Perf: 37.0467 GFlops
```

(4 threads)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 1.79189 s
Perf: 76.7004 GFlops
```

(8 threads)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 1.57942 s
Perf: 87.0189 GFlops
```

(16 threads)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 1.82589 s
Perf: 75.2724 GFlops
```

(32 threads)

```
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run blocked parallel GEMM with OpenMP
Time: 1.94476 s
Perf: 70.6713 GFlops
```

This doesn’t come too much as a surprise, as we already knew from lscpu that the AWS has 8 virtual CPUs (4 physical cores, with 2 threads each). As the number of threads increases from 1 to 8, the performance would definitely go up as more work is being done in parallel. However, if we were to increase from 8, then we would see that the performance would go down. This would be because of the overhead of context-switching, as we are running more threads than the m5.2xlarge server can handle at once.

(Other Challenges)

An interesting thing to note (that isn’t shown in the above code but in the comments in the source code of “omp-blocked.cpp”) is that dynamic allocation is slower only in some cases. I originally wrote two versions of the program (one that uses malloc and the current version that simply allocates a 256 x 256 array on the stack). The older code worked with malloc a lot more efficiently than with on-stack memory.