HW2 Thread/OpenMP Study for CS553

Coen Petto

***Abstract*—This study looks at how thread count affects the performance of a parallel computation algorithm. I used OpenMP pragmas on GEMM to see what kind of speedups I could get.**

I. Introduction

THIS writeup is about my attempt to speed up GEMM using OpenMP. This was performed on MACKEREL the CS machine with an i9-13900k and 24cores/32threads. I added two pragmas to the code:

```
#pragma omp parallel for private(j, k)
    shared(C, A, B, alpha, beta)
for (i = 0; i < _PB_NI; i++) {
  for (j = 0; j < _PB_NJ; j++)
    C[i][j] *= beta;
  for (k = 0; k < _PB_NK; k++) {
    #pragma omp simd
    for (j = 0; j < _PB_NJ; j++)
      C[i][j] += alpha * A[i][k] * B[k][j];
  }
}
```

Listing 1. GEMM with pragmas

The outer loop gets parallelized across threads. This is effective because each row in GEMM can be processed without depending on others. The inner loop uses SIMD to vectorize operations at the instruction level which allows for the process of multiple data points in parallel. I tested this setup with different thread counts and N sizes to see where things improve and where they don't. The next section goes into my observations.

II. Results

Both tables show results for varying thread counts across GEMM. Table I shows average runtime (in milliseconds) for small N, and Table II shows average runtime (in seconds) for large N. My script creates a much larger CSV file, but I restricted the table to the most interesting thread counts since Threads 17-31 just reinforce trends that are observable without their presence.

Analysis:

- Besides a few exception(s) more threads = more performance. This becomes increasingly obvious as N grows larger as well. Where in early increases to threads 2 to 4 or 4 to 8 a rough 2x speedup can be observed. So as the workload scales, the available threads can be more fully utilized, and the benefits of parallel execution outweigh the synchronization and management costs.
- For the smallest problem size (e.g., N = 32), when the thread count is increased to 32 there is actually a performance loss compared to 16 threads. I believe this is most likely due to the absurd thread overhead where each data point sits on its own thread, and having to join 32 individual single computation threads is much slower than 8 or even 4 threads that perform 4 or 8 iterations each. To clarify the 32 individual threads would be executed simultaneously, but their actual execution time is a joke compared to how long it takes to wrangle them back into

TABLE I
Average Runtime (in milliseconds) for Small Problem Sizes and Varying Threads

N	Threads	Avg Time (ms)
32	no openmp	0.000 47
	1	0.000 81
	2	0.000 46
	4	0.000 33
	8	0.000 38
	16	0.000 39
	32	0.000 50
64	no openmp	0.004 34
	1	0.005 23
	2	0.002 75
	4	0.0014
	8	0.000 91
	16	0.001 69
	32	0.000 85
128	no openmp	0.040 71
	1	0.045 03
	2	0.023 09
	4	0.011 55
	8	0.005 92
	16	0.005 57
	32	0.003 53
512	no openmp	2.114 19
	1	2.765 41
	2	1.437 54
	4	0.724 03
	8	0.388 21
	16	0.311 52
	32	0.172 91
1024	no openmp	156.9721
	1	217.9414
	2	116.6105
	4	60.1103
	8	28.7119
	16	24.9597
	32	14.9104

TABLE II
AVERAGE RUNTIME (IN SECONDS) FOR LARGE PROBLEM SIZES AND VARYING THREADS

N	Threads	Avg Time (s)
2048	no openmp	2.263 063 2
	1	2.740 232 7
	2	1.712 989 1
	4	0.893 682
	8	0.554 609 7
	16	0.293 251 1
	32	0.217 017 2
4096	no openmp	23.046 109 1
	1	28.219 747 9
	2	17.331 069 1
	4	9.129 137 7
	8	6.823 659 4
	16	5.264 412
	32	3.575 467 9

one final output array.

- When using Pragmas and OpenMP with 1 thread there is a noticeable performance loss compared to executing the code unmodified and relying on GCC to parallelize whatever seems fit. From HW1 I can tell you that this code is versioned, but seems to execute sequentially when untouched. Anyways, this introduces an interesting behavior. OpenMP has overhead that negatively affects performance, and is detrimental when it isn't used to properly exploit multiple threads. In short, if you use OpenMP you better be using multiple threads.
- It appears the least important step in thread count occurs between 8 to 16. Not sure why this happens.... ChatGPT claims, "While increasing from 1 to 8 threads yields strong runtime reductions, the improvement from 8 to 16 threads is modest. Interestingly, performance improves significantly again at 32 threads. This is likely due to the i9-13900K's hybrid architecture, where 8 performance and 16 efficiency cores (supporting 32 threads total) see better utilization at higher thread counts. Gains between 8 and 16 threads are limited by overhead, but larger workloads benefit from full core saturation and hyper-threading."

III. CONCLUSION

OpenMP parallelization shows clear performance gains as problem size increases, with larger N values benefiting significantly from increased thread counts. For small inputs, threading overhead can actually hurt performance, especially with excessive thread counts. Moderate thread counts (e.g., 4–8) offer the best balance for mid-sized problems, while full core utilization shines in large workloads. Ultimately, effective parallelization depends on choosing the right thread count to match the problem scale.

IV. COMMAND REFERENCE AND SAMPLE OUTPUT

A. Machine Info

Command:

```
python3 tuning2.py --machine-info
```

This command will display CPU information for your current machine.

Example Output:

```
(base) mackerel:~/CS553/[CS553] HW2 Coen
    Petto$
python3 tuning2.py --machine-info

Processor: 13th Gen Intel(R) Core(TM)
    i9-13900K
Number of physical cores: 24
Number of logical CPUs: 32
Threads per core: 1
```

B. Explore Without OpenMP

Command:

```
python3 tuning2.py --read-environment
    openmp_env.txt --explore-polybench gemm.c
    --explore-N "32,64,128,512,1024,2048,4096"
```

This command will perform a basic search across different matrix sizes N on GEMM without OpenMP or pragmas.

Example Output:

```
(base) hanoi:~/CS553/[CS553] HW2 Coen Petto$
python3 tuning2.py --read-environment
    openmp_env.txt --explore-polybench gemm.c
    --explore-N "32,64,128,512,1024,2048,4096"

Compiling for N = 32...
Running 10 experiments for gemm_32...
N = 32 - Min: 4e-06, Max: 7e-06, Avg: 5.2e-06
Compiling for N = 64...
Running 10 experiments for gemm_64...
N = 64 - Min: 3.5e-05, Max: 7.3e-05, Avg:
    5.33e-05
Compiling for N = 128...
Running 10 experiments for gemm_128...
N = 128 - Min: 0.00041, Max: 0.000588, Avg:
    0.0004645
Compiling for N = 512...
Running 10 experiments for gemm_512...
N = 512 - Min: 0.020585, Max: 0.024792, Avg:
    0.0223645
Compiling for N = 1024...
Running 10 experiments for gemm_1024...
...
```

C. Explore Threads

Command:

```
python3 tuning2.py --read-environment
    openmp_env.txt --explore-polybench
    gemm_pragma.c --explore-N
    "32,64,128,512,1024,2048,4096"
    --explore-threads
```

This command will perform an intensive search across different values of N on GEMM with OpenMP using the specified pragmas in the given C file. For each N, it explores 0 to Max threads (as defined in `openmp_env.txt`).

Example Output:

```
(base) hanoi:~/CS553/[CS553] HW2 Coen Petto$
python3 tuning2.py --read-environment
    openmp_env.txt --explore-polybench
    gemm_pragma.c --explore-N
    "32,64,128,512,1024,2048,4096"
    --explore-threads

Compiling for N = 32...
Exploring threads for binary gemm_32...
OMP_NUM_THREADS value: 32
Exploring with 0 threads...
Thread count 0: Min: 4e-06 Max: 5e-06 Avg:
    4.4e-06
Exploring with 1 threads...
Thread count 1: Min: 7e-06 Max: 1.1e-05 Avg:
    8.9e-06
Exploring with 2 threads...
Thread count 2: Min: 5e-06 Max: 7e-06 Avg:
    5.2e-06
Exploring with 3 threads...
Thread count 3: Min: 3e-06 Max: 4e-06 Avg:
    3.6999999999999997e-06
Exploring with 4 threads...
Thread count 4: Min: 3e-06 Max: 4e-06 Avg:
    3.5e-06
Exploring with 5 threads...
Thread count 5: Min: 3e-06 Max: 3e-06 Avg:
    3e-06
...
```