CS133 Lab1 report

Section1:

- **The results comparing the sequential version with the two parallel versions on three different problem sizes (1024^3, 2048^3, and 4096^3, change in " lib/gemm.h"). If you get significantly different speedup numbers for the different sizes, please explain why.**

For the size 1024:

Sequencial Gemm:

Time: 3.47097 s

Perf: 0.618699 GFlops

Gemm parallel:

Time: 0.023445 s

Perf: 91.5967 GFlops

Gemm blocked parallel:

Time: 0.044076 s

Perf: 48.7223 GFlops

For the size 2048:

Sequencial GEMM:

Time: 59.5108 s

Perf: 0.288685 GFlops

gemm parallel:

Time: 0.195019 s

Perf: 88.0933 GFlops

Gemm blocked parallel:

Time: 0.195317 s

Perf: 87.9589 GFlops

For the size 4096:

Sequencial GEMM:

Time: 734.154 s

Perf: 0.187207 GFlops

gemm parallel:

Time: 3.4446 s

Perf: 39.8998 GFlops

Gemm blocked parallel:

Time: 1.45949 s

.lPerf: 94.1694 GFlops

Speedup = 1-thread execution time / n-thread execution time

For Gemm Parallel:

1024:148x speedup and 91.5967 GFlops.

2048: 305.15x speedup and 88.0933 GFlops

4096: 213.1x speedup and 39.8998 GFlops

For Blocked Gemm Parallel:

1024:78.7x speedup and 48.7223 GFlops

2048:304.68x speedup and 87.9589 GFlops

4096: 503x speedup and 94.1694 GFlops

Section 2:

omp-blocked only: For the problem size 4096^3, please quantify the impact of each optimization, including the block size selection and optimizations you performed.

Loop permutation ➔ 103x speed up increase 25.1 GFlops increase.

By adding Parallelization + static scheduling ➔ 365x speedup increase and 58.3 GFlops increase.

By adding Block sizing (64) ➔35x speedup increase and 10.8 GFlops increase.

Loop permutation+ Parallelization+ static scheduling+ blocking ➔ 503x speedup and 94.1694 GFlops

I chose to use block sizing of 64 which is the optimal close the cache size, if I changed the size by increasing or decreasing, I get lower performance. This is why I believe that 64 is an optimal block sizing.

Section 3:

omp-blocked only: For the problem size 4096^3, please report the scalability of your optimized code with different numbers of threads, including 1, 2, 4, etc, and discuss the result. (hint: please find out how many threads are supported on m5.2xlarge).

Based on AWS specification provided in this link:
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html

The instance m5.2xlarge has 4 default CPU cores, and each core support 2 threads. So, in total the instance support 4*2 =8

For 1 thread: Time: 6.73185 s
Perf: 20.4162 GFlops

For 2 threads: Time: 3.25126 s
Perf: 42.2725 GFlops

For 4 threads: Time: 1.6887 s
Perf: 81.3875 GFlops

For 8 threads: Time: 1.4303 s
Perf: 96.0907 GFlops

For 16 threads: Time: 1.43403 s
Perf: 95.8407 GFlops

As we increase the number of threads, the performance of my optimized Blocked-omp increases until it reaches the number of threads (8) supported by the instance m5.2xlarge, then we see a significant decrease (specifically for 16 threads). I believe when we create more threads than what the system can support; we actually oversubscribe the system which results in decreased performance.

Section4: Discussing the result

From these results, we can see that the influence of loop permutation optimization and parallelization using OpenOmp have a huge impact on speeding up the calculation of the matrices even with the size 4096^3. The blocking technique has a significant impact on the speedup but not on the magnitude of loop permutation and penalization using parallelization using OpenOmp. This due, I believe to the fact that loop permutation already benefited from the cache locality which can also be provided by the blocking technique.