CS267 Homework 1 Part 2: Optimize Matrix Multiplication

Jingbo Wu
Dept. of EECS, UC Berkeley
BWRC
cuiusoewin@gmail.com

Ashvin Nair
Dept. of EECS, UC Berkeley
BAIR
nair.ashvin@gmail.com

Abstract

In this report, we discuss our experience parallelizing matrix multiplication on multiple cores of NERSC's Cori cluster. We achieve an average peak performance of 10.0% using OpenMP on top of our serial optimizations. We needed to use a different loop ordering, intermediate buffers for each thread, and varying the block size and number of threads based on problem size to achieve our final performance.

1 Approach

Our approach broadly was to use OpenMP to parallelize the matrix multiplication to different threads, and then arrange the computation so that there was minimal interference between different threads. In this section we discuss the changes we implemented to optimize parallel performance.

1.1 Loop Reordering

While the loop order has some effect in the serial case, it is extremely important in the parallel case because writing to similar locations in arrays in different threads introduces false sharing and results in extreme slowdowns. Therefore it is vital to order the loops so that the innermost loop operates on a single block of C. We then allow OpenMP to parallelize the outer two loops with the following directive:

```
#pragma omp parallel for collapse(2)
```
[*1*]

1.2 Intermediate Buffers

To further prevent any communication between threads, we create a buffer for each thread to accumulate its required updates to C. After a thread computes a block, it writes the final update back to C. This also improves the spatial locality of writes: instead of writing to an $M \times N$ block of an lda $\times$ lda array, we directly write to a $M \times N$ array. The final write is less likely to cause cache conflicts because we do not need to access A or B at that stage. We found that using buffers was a key change: it improved performance about $2\times$.

1.3 Varying Block Size and Thread Count

Finally, we found that the performance of each problem size varies significantly with block size and thread count. For the smaller problems (≤ 320), we use block size 32, and for larger problems we use block size 64. We found that the default OpenMP settings overestimates the number of threads required and instead let the thread count scale based on the problem size according to:

```
int num_threads = min(64, lda * 2 / block_size);
```
[*2*]

We arrived at this strategy through a combination of empirical results and intuition. A problem of size lda with block size S splits into $\lceil \frac{\text{lda}}{S} \rceil^2$ blocks. We would hope that we can run this many threads independently and accumulate the result. However, in practice, we found a linear scaling of threads was faster. Parallelizing beyond 2 threads per core did not improve performance.

2 Performance

Figure 1 summarizes the measured performance of the optimization methods we attempted. The following methods are included:

- Single-thread: our serial code without modifications. This method is relatively consistent among various problem sizes and achieves 1.5% average peak performance.
- Reordered Parallel: adding the OpenMP directive and reordering loops to avoid false sharing. This method is still slow on smaller problem sizes (possibly because of threading overhead) and achieves 2.2% average peak performance.
- Resizing Parallel: controlling the number of threads allows us to get better performance on all problem sizes, with 5.1% average peak performance.
- Final: adding a buffer to accumulate updates for each thread and further tuning the block size and number of threads brings us to our final 10.0% average peak performance.

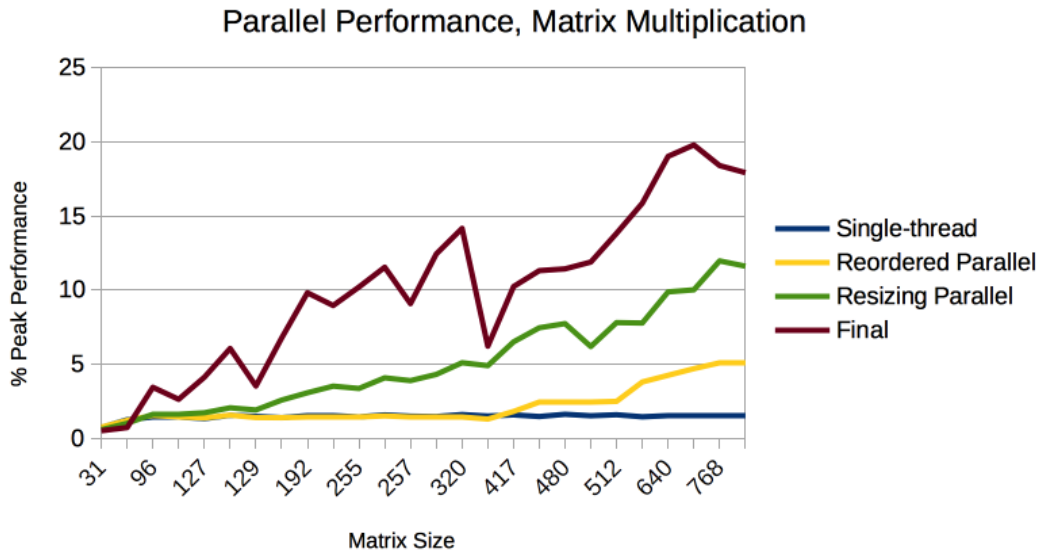

Figure 1: Percent of peak performance achieved by different optimization methods.

Figure 2 illustrates a subset of the same optimizations running on another machine (Gauss) with a Haswell i7 processor. The machine's CPU shares the same architecture as Cori including cache sizing and latency, but runs at a higher clock frequency and contains only 8 cores. The performance on smaller problem sizes matches the Cori performance, suggesting that there is issues related to the cache still left to be solved. On larger problems, Cori exceeds the performance on Gauss by taking advantage of its extra cores.

3 Analysis

Carefully tweaking the single-core code for matrix multiplication significantly changed the speedup for multi-core execution. Reducing inteference between threads and chunking the problem into a reasonable size per thread allowed us to reach our final performance. Our method is about $2\times$ slower than the corresponding BLAS routine, which is heavily optimized for many years for this task.

The performance curve of our final method is somewhat irregular and deserves further consideration. The performance reaches a local maximum at size 320, using a block size of 32 and therefore a team of 20 threads for OpenMP. With a block size of 32, the entire computation (including data from all 3

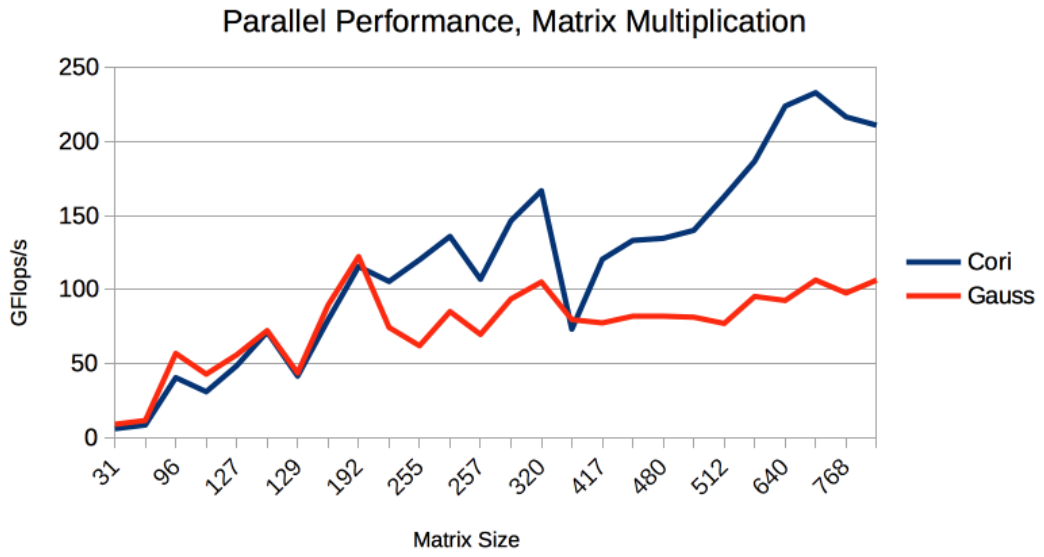

Figure 2: GFlops/s achieved on different platforms.

matrices) can fit into the L1 cache. The shift to using a block size of 64 (a problem which fits into L2 cache but not L1) for the larger matrices is one reason for the performance drop. However, we found that using a block size of 32 for the next set of sizes actually slows them down. This effect is seen in the results for Gauss, which only has 8 cores (16 hyper-threaded). After size 192, which fills up the CPU by using 12 threads, there is a performance drop-off and the extra threads are effectively executed serially. Instead, by increasing the block size, we allow the computation to remain parallel among cores.

4 Team Contribution

Jingbo did most of the work to speed up the serial code, including all of the optimizations included in the final implementation. Ashvin attempted some blocking and automatic vectorization which proved less effective and mainly investigated the parallel section. We met several times to discuss strategies for both sections.