Parallel Blocked Matrix Multiplication Report

David Akeley

1 Parallel Speedup

This table compares the measured **GFlops** for the sequential version, the naively parallelized version, and my optimized block-parallel version. All three were benchmarked for square matrix multiplication, with sizes 1024, 2048, and 4096 tested. Due to the excruciating slowness of the two unoptimized matrix multipliers, I only ran each benchmark once – noise may affect these results.

Square matrix size	1024	2048	4096
Sequential	0.5887	0.2821	0.1911
Parallel	1.589	0.9875	0.8700
Parallel Blocked	117.02	118.85	118.91

The parallel blocked implementation performs about the same on all three matrix sizes (there is a very slight degredation for smaller matrices, probably due to constant startup costs). Meanwhile, the two slower algorithms get progressively slower as matrix size increases. This is what you'd expect from non-cache-aware algorithms; performance decreases when they are given more memory to thrash through.

Note: one thing that disturbs me about these results is that the performance of the parallel-blocked implementation is consistently slower by 15 GFlops when it's run after the sequential or naive parallel algorithm, compared to when it's run by itself, as it was in benchmarks reported in later sections of this report. This has happened every time I've tried to run this sequential-versus-blocked benchmark, and I cannot explain it.

2 Parallel-Blocked Thread Scalability

I got near perfect scaling from one to four threads, with a much smaller increase from four to eight threads. This is probably because the AWS instance provides only four physical cores, which are split into 8 threads with hyperthreading. Because my code has such high arithmetic intensity, there isn't much parallelism left for hyperthreading to exploit. (In fact, on experiments with my personal 4 core, 8 thread laptop, I got consistently better performance with 4 threads than with the default 8, probably due to increased cache pressure with 8 threads).

Each GFlops result reported is the mean of five trial runs.

Cores	GFlops
1	26.14
2	53.71
4	107.93
8	133.25

3 Optimizations Done

I performed several optimizations on my parallel blocked matrix multiplier implementation. All benchmarks in this section are based on the average (mean) GFlops over 5 consecutive trials. I list the optimizations done in roughly most-to-least effective order.

Although you ask for a brief report; I performed so many optimizations that I had to use several pages to list them all. The uninterested reader may skip to the table at the end of this section quantifying the results of these optimizations.

3.1 Tiling

I break the output matrix into 128-by-128 tiles. Each tile is filled during one parallel task, which are assigned dynamically to threads by incrementing a single shared atomic counter representing the next tile to be filled (the tiles are numbered in lexicographical order). Delegating work based on the output matrix minimizes synchronization overhead: aside from the atomic counter and possible false sharing near the borders of tiles, no thread will ever be delayed due to concurrent writes.

I also tested 64-by-64 tiles and 256-by-256 tiles. Both were slower than 128-by-128.

Table of GFlops by tile size and square matrix size

Matrix Size	1024	2048	4096
Tile Size			
64	121.91	125.65	113.18
128	130.52	133.00	133.17
256	116.64	123.07	125.11

3.2 Vectorization – Fused Multiply Add (FMA)

To calculate dot products (part of the process of multiplying two blocks of the input matrices), I use Intel FMA3 instructions. This allows me to replace 32 instructions (16 memory reads, 8 multiplications, 8 additions) with 3 instructions (2 vector memory reads, 1 vector fused multiply-add). This FMA3 instruction conceptually adds 8 more partial sums (+=$a_{i+0}b_{i+0} + a_{i+1}b_{i+1} + ... + a_{i+7}b_{i+7}$) to the dot product in a single step. (Except that a short addition tree is needed at the end, to add together the 8 partial sums in 8 lanes of a vector register to form the final dot product).

3.3 Dot Product Parallelization

The dependency chain of the dot product operation is long: unless an addition tree is used, each subsequent sum depends on the result of the previous one. This wastes the processor's throughput; the latency of waiting for an addition to finish prevents some floating point units from being utilized.

In order to better exploit single-thread parallelism, I calculate four dot products at once, cycling the execution of FMA instructions between each of the four. In theory, work on the other three dot products can be done while waiting for the result of the result of one dot product's FMA to be available.

As side effects of this optimization, I unroll the inner loop of the matrix tile multiplication function by 4, I save memory reads (the left hand input for the four parallel dot products is always the same, and memory reads for it can be shared), and I vectorize the 8 lane add tree better. It's possible that these side effects are actually responsible for the speedup observed from this optimization.

3.4 Temporary Sum Matrix

Filling in one tile of the output matrix requires summing up $k/TileSize$ sub-matrices, each produced by multiplying two tiles of the two input matrices. Naive implementations tend to use the output matrix itself as scratch space for summing these matrices. I instead use a thread-private buffer for this, writing only the final sum to the output matrix. The speedup observed from this may be from better locality and/or minimized false sharing.

As additional micro-optimizations, I avoid zero initializing the temporary sum by treating the first iteration of a tile-filling job as a special case using C++ template programming (I write to the temporary sum directly instead of using an add-assign operation). I use a similar technique in the last iteration to avoid having to copy from the temporary sum to the output matrix – I write to the output matrix directly in the last iteration. I did not try to quantify the effects of this micro-optimization.

3.5 RHS Transpose Future

This optimization is kind of hard to explain. The right-hand side (RHS) input matrix is in the "wrong" order for performance, which is especially problematic for vectorization. This suggests that when multiplying two matrix tiles, we should transpose the RHS tile. The problem with this is that this wastes some GFlops – the floating point units sit idle during this memory-intensive operation.

Instead of transposing the entire RHS tile at once, I interleave the work of transposing the tile with the work of calculating one tile matrix product. This means that the bulk of transpose work can be done while floating point instructions are in-flight. I provide a double buffer system: while calculating one tile matrix product, the `rhs_transpose` buffer is assumed to hold the transpose of the input RHS tile matrix, while the `rhs_transpose_future` buffer is gradually filled with the transpose of the *next* tile matrix product's RHS input. Swapping the buffers after each tile matrix product is done maintains the `rhs_transpose` invariant for the next tile matrix product.

Because of how fragile and error-prone this optimization sounds, I tested this optimization with a debug build containing assertions that (sort of) check this optimization's validity. This optimization also depends on the first/last iteration template – this optimization is useless on the initial iteration, and should not be done for the final iteration of computing a tile matrix product.

3.6 Quantifying effects

To quantify the effects of these optimizations (except for tiling, which was tested separately), I provided preprocessor flags that control whether each optimization was in effect. I then built 16 different versions of the parallel blocked matrix multiplier, one version for each possible combination of optimizations, and a final version (p16) that was a naive rewrite that performed no other optimizations aside from tiling.

This table ranks the average GFlops of the builds, averaged over five $4096 \times 4096 \times 4096$ matrix multiplications.

GFlops	133.31	121.71	118.83	113.79	105.65	105.55	92.30	88.12
Build Number	p0	p8	p2	p4	p12	p10	p6	p14
Fused Multiply Add	y	y	y	y	y	y	y	y
Dot Product Parallel.	y	y		y	y			
Tmp Sum Matrix	y	y	y			y		
RHS Transpose Future	y		y	y			y	
GFlops	30.96	30.83	30.24	29.97	12.59	12.59	12.49	12.31
Build Number	p1	p5	p13	p9	p3	p7	p11	p15
Fused Multiply Add								
Dot Product Parallel.	y	y	y	y				
Tmp Sum Matrix	y			y	y		y	
RHS Transpose Future	y	y			y	y		

Note: The naive (p16) build achieved only 2.52 GFlops.

4 Discussion

I'm really happy with the results.[1] Version p0, the official version with all optimizations, performs best, validating my optimization decisions.

By far the most significant optimization done, besides tiling, is the use of fused multiply-add instructions. There's a clear performance gulf between builds using FMA (even numbered, except p16) and builds not using FMA – the best non-FMA build achieves less than $\frac{1}{4}$ the speed of my official build. I remember one of the TAs telling me that manual vectorization may actually slow down my code by interfering with compiler auto-vectorization. At least with the compiler flags provided by the default makefile, this appears not to be true. I'm glad I ignored that advice and stubbornly vectorized my code manually anyway.

The remaining three optimizations have a less dramatic impact, but all of them seem "worth it". Disabling any one of them costs at least 10 GFlops, and disabling all three incurs a $\frac{1}{3}$

[1]As anyone who's played Secret Hitler with me, I'm super irrationally power-hungry, and there's nothing like seeing 133 GFlops at my command to satisfy that urge. Can't wait to get started on GPU programming!

performance drop. (See build p14).

I'm surprised by how much of an effect the dot product parallelization and RHS transpose future optimizations had. I did not think instruction reordering would have such a significant impact for modern out-of-order processors, and expected the two to be micro-optimizations at best. I was particularly pessimistic about the RHS transpose future optimization, since it actually slightly reduced performance on my work laptop; it didn't show any positive effect until my final benchmark on the real AWS instance. Even then, its effect is the smallest among all optimizations, which figures, since it was the hardest to implement and reason about.

To be honest, I didn't "iterate" when writing my code like they tell you to in CS 31. I lounged about in my living room and banged out the entire p0 build one evening, adding every optimization my Pinkie Sense told me to add before testing the code even once. My first working build gave me a satisfactory 250 GFlops performance on my gaming desktop.[2] I then successively removed optimizations in order to have something to write about. Since I wasn't too motivated to optimize my intentionally-slowed-down builds, it's possible that these results oversell the effectiveness of my optimizations – maybe if I made a more honest effort to optimize before resorting to the extreme tactics I described here, the performance delta of said tactics wouldn't be so great. I'm particularly perplexed by the 5X performance difference between builds p15 and p16, which are conceptually the same (no optimizations besides tiling) – somehow the fact that I wasn't so motivated writing p16 resulted in a real performance difference.

I actually have no idea how my classmates are getting really respectable performance, presumably without using the extreme optimization tactics I've employed that were not taught or suggested in class. (In fact, I've heard many submissions have considerably better performance than mine, which makes me worried that I've missed something obvious). It's possible that if I warmed up to OpenMP more and studied it earnestly like my classmates, I might have found out OpenMP could do some of the manual optimizations I performed (or at least auto-vectorization). I personally don't like how "magical" OpenMP seems, and my first attempt at parallelization with tasks led to abysmal performance. So, I kept interaction with OpenMP to a bare minimum in this assignment – just a single `#pragma omp parallel` directive. I used homemade constructs anywhere else I had to deal with parallelism.

[2]I was understandably disappointed by my AWS instance when I finally had access to it.

5 Appendix: Benchmark Code

5.1 benchmark.sh

```
#!/bin/bash
($1 parallel-blocked 2>&1 && $1 parallel-blocked 2>&1 && \
$1 parallel-blocked 2>&1 && $1 parallel-blocked 2>&1 && \
$1 parallel-blocked 2>&1) | python3 avg_gflops.py
```

5.2 avg_gflops.py

```
#!/usr/bin/python3

import sys

samples = 0
total = 0.0

for line in sys.stdin:
    if line.startswith("Diff:"):
        print(line, file=sys.stderr)

    if line.startswith("Perf:"):
        n = float(line.split()[1])
        total += n
        samples += 1
        sys.stderr.write("Got sample %f GFlops\n" % n)
        sys.stderr.flush()

print(total/samples)
```

6 Appendix: Tile (Block) Size Testing Log

Nomenclature: `./gemm-MatrixSize-TileSize`

```
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-1024-64
Got sample 121.968000 GFlops
Got sample 121.857000 GFlops
Got sample 122.252000 GFlops
Got sample 121.815000 GFlops
Got sample 121.650000 GFlops
121.9084
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-1024-128
Got sample 130.944000 GFlops
Got sample 130.936000 GFlops
Got sample 129.257000 GFlops
Got sample 131.040000 GFlops
Got sample 130.427000 GFlops
130.5208
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-1024-256
Got sample 115.116000 GFlops
Got sample 117.016000 GFlops
Got sample 116.565000 GFlops
Got sample 116.730000 GFlops
Got sample 117.793000 GFlops
116.644
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-2048-64
Got sample 125.748000 GFlops
Got sample 125.655000 GFlops
Got sample 125.389000 GFlops
Got sample 125.847000 GFlops
Got sample 125.597000 GFlops
125.6472
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-2048-128
Got sample 132.472000 GFlops
Got sample 132.072000 GFlops
Got sample 133.354000 GFlops
Got sample 133.563000 GFlops
Got sample 133.569000 GFlops
133.006
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-2048-256
Got sample 123.288000 GFlops
Got sample 123.058000 GFlops
Got sample 123.409000 GFlops
Got sample 122.630000 GFlops
Got sample 122.989000 GFlops
123.07480000000001
```

```
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-4096-64
Got sample 113.283000 GFlops
Got sample 113.174000 GFlops
Got sample 113.676000 GFlops
Got sample 113.361000 GFlops
Got sample 112.410000 GFlops
113.1808
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-4096-128
Got sample 133.260000 GFlops
Got sample 133.245000 GFlops
Got sample 133.259000 GFlops
Got sample 132.987000 GFlops
Got sample 133.114000 GFlops
133.173
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./benchmark.sh ./gemm-4096-256
Got sample 122.753000 GFlops
Got sample 125.752000 GFlops
Got sample 125.490000 GFlops
Got sample 126.159000 GFlops
Got sample 125.374000 GFlops
125.10560000000001
```

7 Appendix: Thread Scalability Testing Log

```
ubuntu@ip-172-31-0-100:~/matrix/lab1$ OMP_NUM_THREADS=1 ./benchmark.sh ./gemm
Got sample 26.022300 GFlops
Got sample 26.037400 GFlops
Got sample 26.315500 GFlops
Got sample 26.347500 GFlops
Got sample 25.978000 GFlops
26.140140000000002
ubuntu@ip-172-31-0-100:~/matrix/lab1$ OMP_NUM_THREADS=2 ./benchmark.sh ./gemm
Got sample 53.478700 GFlops
Got sample 54.055500 GFlops
Got sample 53.406200 GFlops
Got sample 54.097200 GFlops
Got sample 53.534300 GFlops
53.714380000000006
ubuntu@ip-172-31-0-100:~/matrix/lab1$ OMP_NUM_THREADS=4 ./benchmark.sh ./gemm
Got sample 107.592000 GFlops
Got sample 108.381000 GFlops
Got sample 107.854000 GFlops
Got sample 108.016000 GFlops
Got sample 107.800000 GFlops
107.9286
ubuntu@ip-172-31-0-100:~/matrix/lab1$ OMP_NUM_THREADS=8 ./benchmark.sh ./gemm
Got sample 133.302000 GFlops
Got sample 133.048000 GFlops
Got sample 133.255000 GFlops
Got sample 133.310000 GFlops
Got sample 133.350000 GFlops
133.253
```

8 Appendix: Sequential - Parallel Benchmarks Log

```
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./gemm-4096 sequential parallel parallel-blocked
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run sequential GEMM
Time: 719.346 s
Perf: 0.191061 GFlops
Max delta squared: 1.3411e-07

Run parallel GEMM with OpenMP
Time: 157.982 s
Perf: 0.869966 GFlops
Max delta squared: 1.3411e-07

Run blocked parallel GEMM with OpenMP
Time: 1.15582 s
Perf: 118.91 GFlops
Max delta squared: 2.15173e-05

ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./gemm-2048 sequential parallel parallel-blocked
Problem size: 2048 x 2048 x 2048
Initialize matrices a and b

Run sequential GEMM
Time: 60.8947 s
Perf: 0.282124 GFlops
Max delta squared: 3.35276e-08

Run parallel GEMM with OpenMP
Time: 17.3974 s
Perf: 0.987499 GFlops
Max delta squared: 3.35276e-08

Run blocked parallel GEMM with OpenMP
Time: 0.14455 s
Perf: 118.851 GFlops
Max delta squared: 2.92063e-06
ubuntu@ip-172-31-0-100:~/matrix/lab1$ ./gemm-1024 sequential parallel parallel-blocked
Problem size: 1024 x 1024 x 1024
Initialize matrices a and b

Run sequential GEMM
Time: 3.64775 s
```

```
Perf: 0.588714 GFlops
Max delta squared: 8.3819e-09

Run parallel GEMM with OpenMP
Time: 1.35128 s
Perf: 1.58922 GFlops
Max delta squared: 8.3819e-09

Run blocked parallel GEMM with OpenMP
Time: 0.018351 s
Perf: 117.023 GFlops
Max delta squared: 4.10713e-07
```

9 Appendix: Optimization Testing Log

Nomenclature: `./gemm-pBitfield`

Bit 16 Set: Use naive blocked implementation.

Bit 8 Set: Disable `rhs_transpose_future` (don't interleave transposing the next block of the right-hand matrix with calculating dot products for the current block).

Bit 4 set: Disable `tmp_sum` (use output matrix as scratch space directly).

Bit 2 set: Disable parallel dot products (calculate 1 dot product at a time instead of 4).

Bit 1 set: Disable vectorization and fused multiply-add instructions (some vectorized SSE instructions are still used).

```
ubuntu@ip-172-31-0-100:~/matrix/lab1$ python3
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> for i in range(17):
...  print("Benchmark ./gemm-p%d"%i)
...  status = os.system("./benchmark.sh ./gemm-p%d"%i)
...
Benchmark ./gemm-p0
Got sample 133.494000 GFlops
Got sample 133.147000 GFlops
Got sample 133.120000 GFlops
Got sample 133.417000 GFlops
Got sample 133.381000 GFlops
133.3118
Benchmark ./gemm-p1
Got sample 30.951800 GFlops
Got sample 30.959400 GFlops
Got sample 30.958100 GFlops
Got sample 30.949400 GFlops
Got sample 30.960600 GFlops
30.955859999999994
Benchmark ./gemm-p2
Got sample 118.717000 GFlops
Got sample 118.770000 GFlops
Got sample 118.535000 GFlops
Got sample 118.510000 GFlops
Got sample 119.629000 GFlops
118.83219999999999
Benchmark ./gemm-p3
Got sample 12.630800 GFlops
Got sample 12.550500 GFlops
Got sample 12.519800 GFlops
Got sample 12.625800 GFlops
Got sample 12.624000 GFlops
12.59018
Benchmark ./gemm-p4
Got sample 113.758000 GFlops
Got sample 113.756000 GFlops
Got sample 113.887000 GFlops
Got sample 113.631000 GFlops
Got sample 113.895000 GFlops
113.78540000000001
Benchmark ./gemm-p5
Got sample 30.845100 GFlops
Got sample 30.843900 GFlops
Got sample 30.813500 GFlops
Got sample 30.811000 GFlops
Got sample 30.814200 GFlops
30.82554
Benchmark ./gemm-p6
Got sample 92.223100 GFlops
Got sample 92.014300 GFlops
Got sample 92.661200 GFlops
Got sample 92.223500 GFlops
Got sample 92.385200 GFlops
92.30145999999999
Benchmark ./gemm-p7
Got sample 12.589200 GFlops
Got sample 12.601100 GFlops
Got sample 12.572500 GFlops
Got sample 12.580200 GFlops
Got sample 12.588800 GFlops
12.586359999999999
Benchmark ./gemm-p8
Got sample 121.718000 GFlops
Got sample 121.716000 GFlops
Got sample 121.576000 GFlops
Got sample 121.812000 GFlops
Got sample 121.710000 GFlops
121.7064
Benchmark ./gemm-p9
Got sample 29.990300 GFlops
Got sample 29.952000 GFlops
Got sample 29.961800 GFlops
Got sample 29.974300 GFlops
Got sample 29.957300 GFlops
29.96714
Benchmark ./gemm-p10
Got sample 105.391000 GFlops
Got sample 105.420000 GFlops
Got sample 105.327000 GFlops
Got sample 105.511000 GFlops
Got sample 106.081000 GFlops
105.546
Benchmark ./gemm-p11
Got sample 12.497400 GFlops
Got sample 12.477400 GFlops
Got sample 12.505800 GFlops
Got sample 12.466100 GFlops
Got sample 12.505300 GFlops
```

```
12.490400000000001
Benchmark ./gemm-p12
Got sample 105.686000 GFlops
Got sample 105.821000 GFlops
Got sample 105.423000 GFlops
Got sample 105.544000 GFlops
Got sample 105.753000 GFlops
105.6454
Benchmark ./gemm-p13
Got sample 30.272300 GFlops
Got sample 30.238700 GFlops
Got sample 30.205000 GFlops
Got sample 30.243400 GFlops
Got sample 30.233200 GFlops
30.238520000000005
Benchmark ./gemm-p14
Got sample 87.656800 GFlops
Got sample 88.165200 GFlops
Got sample 88.030000 GFlops
Got sample 88.100900 GFlops
Got sample 88.659500 GFlops
88.12248
Benchmark ./gemm-p15
Got sample 12.300100 GFlops
Got sample 12.303100 GFlops
Got sample 12.330600 GFlops
Got sample 12.322500 GFlops
Got sample 12.311100 GFlops
12.313480000000002
Benchmark ./gemm-p16
Got sample 2.525580 GFlops
Got sample 2.520450 GFlops
Got sample 2.522780 GFlops
Got sample 2.520780 GFlops
Got sample 2.505260 GFlops
2.51897
```