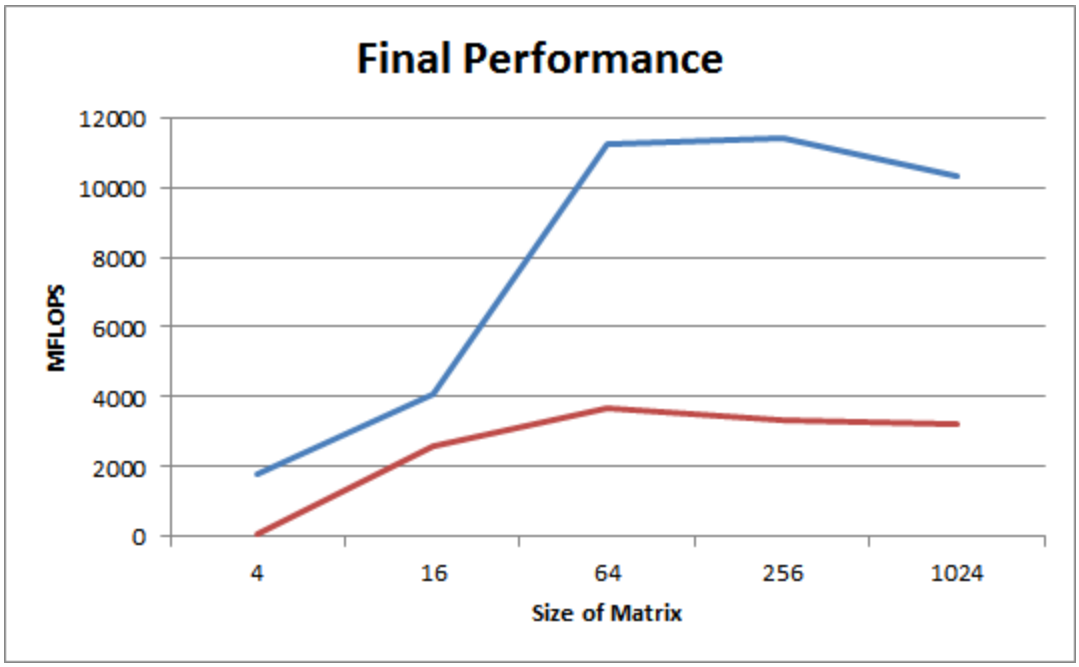
Project 1 Final Submissions

Yang Li
Shiqi Wu
Solomon Sia

1) Final MFLOPS plot:



Size	4	16	64	256	1024
MFLOPS	1794.450	4080.680	11207.476	11381.750	10301.133

2) Description of 5-10 Optimizations

a) **Major Bottleneck:** Overhead cost for multithreading is too significant for small matrix sizes.

Reason for Specific Optimization: Removes unnecessary overhead cost of creating threads, which slows down small computations.

Summary of Optimization: When matrix size is too small, e.g. below size 8, our program runs the calculation in a single thread.

Quantitative Evidence of Optimization Affecting Performance: For matrix size of 4, MFLOPS performance improved from 91 MFLOPS to 1794 MFLOPS.

Plausible hypothesis for how the optimization would affect performance: The optimization worked as expected, removing unnecessary overhead.

b) **Major Bottleneck:** L1 Cache misses become very significant for matrix sizes of 64 and above, slowing down our code from miss fetching.

Reason for Specific Optimization: By using clever block sizes to fit in our L1 cache, we improve spatial and temporal locality and reduce cache miss rate.

Summary of Optimization: We rewrote the loops from a single naive loop to a double loop, where the inner loop operated on matrices of size 32.

Quantitative Evidence of Optimization Affecting Performance: For the 64x64 matrix, performance improved from 3686 MFLOPS to 6130 MFLOPS. L1 Cache misses reduced from 39,000 to 2,600.

Plausible hypothesis for how the optimization would affect performance: Reducing L1 cache misses reduces the cost of memory fetch, hence speeding up performance since memory is a bottleneck.

c) **Major Bottleneck:** Insufficient instruction level parallelization

Reason for Specific Optimization: We felt that the restrict keyword would give the compiler more freedom to optimize the order of instructions.

Summary of Optimization: We implemented the __restrict__ keyword on the matrices that were passed as arguments.

Quantitative Evidence of Optimization Affecting Performance: The restrict keyword improved performance across all matrix sizes. E.g. for a matrix of size 64, performance increased from 939.6 MFLOPS to 1832 MFLOPS.

Plausible hypothesis for how the optimization would affect performance: When memory fetches take a long time, improved instruction level parallelization helps to reduce the cost of a cache miss by hiding it and performing other instructions in the meantime. As our hit rate improves, the effects of the restrict keyword may become less.

d) **Major Bottleneck:** L2 Cache misses are starting to be significant for large matrix sizes, slowing down our code once again from miss fetch.

Reason for Specific Optimization: Our single loop improved the L1 cache hit rate, but large matrix sizes above N=128 were affecting our L2 hit rate.

Summary of Optimization: We implemented a second outer loop that separated matrices into blocks of size 128 to improve L2 hit rate.

Quantitative Evidence of Optimization Affecting Performance: It doesn't seem to have affected performance.

Plausible hypothesis for how the optimization would affect performance: L2 cache misses are not the bottleneck, although the number is increasing, the relative time taken to fetch them is small compared to other things such as L1 compulsory misses.

e) **Major Bottleneck:** Cost of loops and false speculation are costing many cycles.

Reason for Specific Optimization: We believe that the best way to speed up the program is to optimize the individual instructions. One of these methods is loop unrolling and loop optimization manually to reduce the number of branches.

Summary of Optimization: Hand coding the loops and unrolling them reduced the number of branches and false speculation, speeding up the code.

Quantitative Evidence of Optimization Affecting Performance: Even for small matrices of size 4, MFLOPS increased from 895 to 1794. For matrices of size 64, MFLOPS increased from 6130 to 11207. Number of cycles per processor reduced from 249,000 to 136,000.

Plausible hypothesis for how the optimization would affect performance: We thought it would improve performance by reducing cycles, and it did.

3) List of Compiler flags tried and their impact on performance

-ftree-vectorize

-funroll-loops:

After set size of loop to a known number at compiler time, the -funroll-loops flag increase MFLOPS by 1.5 times, for all sizes of matrices (i.e size of 4, 16, 64, 256, 1024).

-no-as-needed

4) Any additional interesting observations

Recursion: We attempted a cache oblivious algorithm that recursively divided the matrix into smaller sizes. While we were able to match the cache aware algorithm, we were unable to come up with further improvements.

Strassen Algorithm: We couldn't get a promised speedup of n^(2.804) instead of n^(3), probably because our computational overhead was too great. We hope in the future to effectively implement Strassen's Algorithm as that would likely lead to a high promised speedup, especially over very large matrices.

1) Current optimization performance

Note: Miss rate = miss count / sum (fhGETS, fhGETX, hGETS, hGETX, mGETS, and mGETXIM)

Size	MFLOPS	Thread ID	CPI	L1 Data Miss Count	L2 Miss Count	L3 Miss Count
16	3599	1	0.668	23	25	
		2	0.48	21	23	
		3	0.48	23	24	
		4	0.48	18	20	
		Average	0.74	21.25	23	8
				0% Miss Rate	100% Miss Rate	8.7% Miss Rate
64	8608	1	0.467	656	33	
		2	0.495	700	116	
		3	0.48	700	116	
		4	0.48	700	116	
		Average	0.48	689	95	20
256	7600	1	0.56	4.376M	27K	
		2	0.56	4.376M	27K	
		3	0.56	4.376M	26K	
		4	0.56	4.376M	27K	
		Average	0.56	4.376M	27K	2531
				5.05% miss rate	0.61% miss rate	2.36% miss rate
1024	6640	1	0.64	279.5M	67.0M	
		2	0.63	279.5M	66.7M	
		3	0.56	279.5M	19.0M	
		4	0.64	279.5M	40.6M	
		Average	0.61	279.5M	48.2M	1.4M
				12.59%miss rate	5.99%miss rate	0.73%miss rate

2) Optimization 1: blocking and multithreading

For matrix of size 64 and more, we use blocking and multithreading optimization approach. As the application runs on a 4-core system, we use 4 thread to maximize parallelism and to minimize overhead. i.e. L1 cache size = 32kb, L2 cache size = 256 kB. Let B be the block size, the total space to store matrices A, B and C is 3*(B*8byte/double)^2 < 32kB. Then B < 13. Since B is a power of 2, the maximum value of B is 8. Similarly, for L2 cache, B < 35, and the maximum power of 2 value of B is 32. We choose the block size to be 16 as a result of compromise, since block size of 8 reduce L1 miss rate but increases performance overhead, and block size of 32 reduce L2 cache miss rate.

Optimization 2: auto-vectorization

We use compiler flag -ftree-vectorize and the “restricted” keyword to enable auto-vectorization. By adding the “_restricted_” keyword to pointers A, B and C in the function argument, we allow the compiler to treat all array references of different arrays as independent. With this modification, the MFLOP of the baseline code increases moderately from 2759 to 3250 for the multi-thread version. To fully exploit the compiler’s auto-vectorization feature, we will need to further transform the loop structure in matmul().

3) We are expecting a 4 times speedup for the multithreaded application over the single thread. Such speedup is observed only when size of matrix exceeds 64. For a matrix of size 16, the multithreaded application is about twice as fast as a single thread; while for a matrix of size 4, multithread is significantly slower than the single thread. This is because overhead is required to implement multithread matrix multiplication, and therefore for small size matrices, the benefit of multithreaded computation cannot overcome the overhead of creating new threads. In conclusion, we can approximate a perfect strong scaling speedup for matrices of large sizes because the overhead of creating new threads becomes insignificant.

Latest fastest single thread

4	731
16	2199
64	2283
256	1906
1024	1928

You do matmul for single and multithreaded types.
Jsub –l -- ./matmul –s 64

To run:

Make gensim
Cd sim-<whatever>
Jsub –l --./ run_script.sh

To check solution:

At top directory pa1, run

./scripts/zout_parser.py ./sim-<timestamp>

Restrict keyword
http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html
sse2 basics and intrinsics
https://www.cs.drexel.edu/~jjohnson/2007-08/fall/cs680/lectures/sse.pdf
compiler flags:
-funroll-loops for loop unrolling
-ftree-vectorize auto use of SSE
-ftree-vectorize-verbose=2

Basic tests and running ability:
Multi Thread Matrix (Naive implementation).

Size	MFLOPS	Cycles	Instr	L1D Miss	L2DMiss	L3Miss
4	91.451	4101+()		24+18+17 +17	26+19+18 +19	4
16	2566.570	9352+ ()				6
64	3686	412722*4				20
256	3321	29568803* 4				2526
1024						

Single Thread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	895	419	696	0	1	0

16	1137	21107	34721	1	1	0
64	939.6	1634902	2126368	38477	15	12
256	834.9	117744244	134678641	16866312	16859664	2470
1024						

2) Restrict keyword:
Multi Thread Matrix.

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	91.451	4101+()		24+18+17 +17	26+19+18 +19	4
16	2846	8434+ (6433*3)				6
64	3690	214697+41 2722*3				20
256	3321	19113156+ 29568803* 3				2526
1024						

Single Thread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	1217.662	308	657	0	0	0
16	1444.632	16615	31140	1	1	0
64	1832.641	838224	1872424	38478	16	16
256						
1024						

3) Blocking 8 keyword:
Multi Thread Matrix.

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	87.361	4101+999*3	600*3+3244	19*4+5	20*4+7	4
16	2759	8434+ 5552*3	12581+10K*3	20*4+3	20*4+5	6
64	4989.392	305K*4	616K*4	1380*4	33+88+87+91	20
256	4963.9	19.8M*4	39M*4	131.4K*4	73K	2526
1024						

Single Thread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	719.846	521	823	0	1	0
16	1255.101	19124	38472	1	2	0
64	1273	1.2M	2.45M	5250	16	16
256	1249	78.7M	156M	522.5K	287.8K	2472
1024						

4) Blocking8 + restrict keyword
Note: Currently restrict increases cycle and instruction performance for the first thread but not for the other threads? Must be the implementation wrong.
As cache misses decrease, benefit of restrict function will increase.
MultiThread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	87.361	4101+999*3	600*3+3244	19*4+5	20*4+7	4
16	2759	7878+ 5552*3	12581+10K*3	20*4+3	20*4+5	6
64	4989.392	305K*3 + 248K	616K*3 + 567K	1380*4	33+88+87+91	20

256	4975.46	19.8M*3+ 15.8M	39M*3+36 M	131.4K*4	73K	2526
1024						

Single Thread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	917	409	823	0	1	0
16	1597	15124	35535	1	2	0
64	1627	0.94M	2.26M	5250	16	16
256	1605	61.2M	144M	491.8K	287.8K	2461
1024						

4) Add flags -funroll, and -ftree-vectorize
Note: Increases mflops for size 16 from 2759MFLOPS to 3250MFLOPS for the multiThread, marginally improves performance. Why? Reduce loop conditionals, esp since size 16 is similar to block size of 8.
MultiThread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4	87.361	4301+999* 3	600*3+324 4	19*4+5	20*4+7	4
16	3250	7878+ 5552*3	12581+10 K*3	20*4+3	20*4+5	6
64	5033	305K*3 + 248K	616K*3 + 567K	1380*4	33+88+87 +91	20
256	4999.46	19.8M*3+ 15.8M	39M*3+36 M	131.4K*4	73K	2526
1024						

Single Thread

Size	MFLOPS	Cycles	Instr	L1D Mis	L2DMiss	L3Miss
4						
16						
64						
256						
1024						

```
/*****************************************************************
* EE282 Programming Assignment 1:
* Optimization of Matrix Multiplication
*
* Updated by: mgao12    04/14/2014
*****************************************************************/

// This is the matrix multiply kernel you are to replace.
// Note that you are to implement C = C + AB, not C = AB!

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 4

/*****************************
* Single-threaded
*****************************/
#if 1
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

typedef struct {
  const double *A;
  const double *B;
  double *C;
  int dim;
  int row_begin;
  int row_end;
  int col_begin;
  int col_end;
```

```
} thread_arg;

void matmul(int N, const double* A, const double* B, double* C) {
  int block_size = 8;

  int i, j, k, i0, j0, k0;
  for (i0=0; i0<N; i0+=block_size){

     for (j0=0; j0<N; j0+=block_size){

        for (k0=0; k0<N; k0+=block_size){


           for (i = i0; i < MIN(i0+block_size, N); i++){
              for (j = j0; j < MIN(j0+block_size, N); j++){
                 for (k = k0; k < MIN(k0+block_size, N); k++){
                    C[i*N + j] += A[i*N + k] * B[k*N + j];
                 }
              }
           }
        }
     }
  }
}
#endif

/*****************************
* Multi-threaded
*****************************/
#if 0
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

typedef struct {
  const double *A;
  const double *B;
  double *C;
  int dim;
  int row_begin;
  int row_end;
  int col_begin;
  int col_end;
} thread_arg;
```

```
void * worker_func (void *arg) {
  thread_arg *targ = (thread_arg *)arg;

  const double *A = targ->A;
  const double *B = targ->B;
  double *C = targ->C;
  int dim = targ->dim;
  int row_begin = targ -> row_begin;
  int row_end = targ -> row_end;
  int col_begin = targ -> col_begin;
  int col_end = targ -> col_end;

  int block_size = 8;

  int i0, j0, k0;
  int i, j, k;
  for (i0 = row_begin; i0 < row_end; i0 += block_size) {
     for (j0 = col_begin; j0 < col_end; j0 += block_size) {
        for (k0 = 0; k0 < dim; k0 += block_size) {
           for (i = i0; i < MIN(i0+block_size, row_end); i++){
              for (j = j0; j < MIN(j0+block_size, col_end); j++){
                 for (k = k0; k < MIN(k0+block_size, dim); k++){
                    C[i*dim + j] += A[i*dim + k] * B[k*dim + j];
                 }
              }
           }
        }
     }
  }
  return NULL;
}

void matmul(int N, const double* A, const double* B, double* C) {
  pthread_t workers[NUM_THREADS - 1];
  thread_arg args[NUM_THREADS];
  int stripe = (N+1) / 2;
  int i;

  args[0] = (thread_arg){ A, B, C, N, 0, stripe, 0, stripe};
  pthread_create(&workers[0], NULL, worker_func, &args[0]);

  args[1] = (thread_arg){ A, B, C, N, 0, stripe, stripe, N};
  pthread_create(&workers[1], NULL, worker_func, &args[1]);
```

```
  args[2] = (thread_arg){ A, B, C, N, stripe, N, 0, stripe};
  pthread_create(&workers[2], NULL, worker_func, &args[2]);

  args[3] = (thread_arg){ A, B, C, N, stripe, N, stripe, N};
  worker_func(&args[NUM_THREADS - 1]);

  for (i = 0; i < NUM_THREADS - 1; i++) {
    if (pthread_join(workers[i], NULL) != 0) {
      fprintf(stderr, "Fail to join thread!");
      exit(1);
    }
  }
}
#endif
```