TEAM 34: Sungjae Lee sungjael@andrew.cmu.edu
Chen Wang chenw@andrew.cmu.edu

Loop Unrolling for Matrix Multiplication

Optimization goal:

- Hardware resources being optimized toward? (cache? SIMD? multicore?)
  Less instructions
- What is the specification of the hardware you are optimizing for?
  Any hardware.

Optimization process:

- Data considerations

We thought looping through all the iteration when multiplying would take too many unnecessary loop-related instructions. First we decide to compute 3 matrix for each iteration to reduce the loop-related instructions and iterations to be decreased by 1/3; Then we found a flag funroll-loops which automatically unroll the loop in complier level which gave us better performance.

On the other hand, there is down side of using Loop Unrolling since the code gets longer, size of the code gets larger. But for this assignment we thought sizing wasn't the matter, so we stick to it.

Another down side is that, If we happen to be using a vectorizing compiler, loop unrolling can interfere with the vector optimizations.

Optimization results:

- Performance before optimization

```
./matrix_mul -I ../matrix_mul_02.dat
Test Case 1     0.154053 milliseconds
Test Case 2     0.00292969 milliseconds
Test Case 3     0.0478516 milliseconds
Test Case 4     0.0500488 milliseconds
Test Case 5     353.472 milliseconds
```

- Performance after optimization

```
./matrix_mul -i ../matrix_mul_02.dat
Test Case 1     1.71997 milliseconds
Test Case 2     0.105957 milliseconds
Test Case 3     0.141113 milliseconds
Test Case 4     0.10791 milliseconds
Test Case 5     289.871 milliseconds
```

20% improvement

TEAM 34: Sungjae Lee sungjael@andrew.cmu.edu
Chen Wang chenw@andrew.cmu.edu

Cache Blocking for Matrix Multiplication

Optimization goal:

- Hardware resources being optimized toward? (cache? SIMD? multicore?)
  Cache.
- What is the specification of the hardware you are optimizing for?
  32kb*4 L1 cache

Optimization process:

- Data considerations
  Cache is a temporary storage area where frequently accessed data can be stored for rapid access. However, with original code, it gives too many compulsory misses and capacity misses.
  In order to improve it we divide the matrix by blocks for better spatial locality.
  We decide to set block as 50 which gave us the best result.

  Total misses compare to cache blocking vs na ïve is 10:1;

  We were not sure how much it would improve the speed, but we knew that will be significant drop on cache miss.

Optimization results:

- Performance before optimization

```
./matrix_mul -i ../matrix_mul_02.dat
Test Case 1        0.114014 milliseconds
Test Case 2        0.00195312 milliseconds
Test Case 3        0.0119629 milliseconds
Test Case 4        0.0119629 milliseconds
Test Case 5        477.169 milliseconds
```

- Performance after optimization

```
./matrix_mul -i ../matrix_mul_02.dat
Test Case 1        0.161865 milliseconds
Test Case 2        0.00292969 milliseconds
Test Case 3        0.0400391 milliseconds
Test Case 4        0.0400391 milliseconds
Test Case 5        281.079 milliseconds
```

59% improvement

TEAM 34: Sungjae Lee sungjael@andrew.cmu.edu
Chen Wang chenw@andrew.cmu.edu

Transpose for Matrix Multiplication

Optimization goal:

- Hardware resources being optimized toward? (cache? SIMD? multicore?)
  - Cache & Multicore
- What is the specification of the hardware you are optimizing for?
  32kb*4 L1 cache
  - CPU Name: Intel(R) Xeon(R) CPU W3520 @ 2.67GHz
  - # of physical cores: 4
  - # of virtual cores: 8

Optimization process:

- Data considerations

In matrix_mul.cpp code, matrices are stored in "row major" order, meaning that if we look in global memory for our matrix we will see it stored in a one-dimensional array from the first row, followed by the second row, etc… Therefore, in order to read the column of the second matrix "sq_matrix_2", we need to move the pointer "sq_dimension" steps to get only one value in the column. If we transpose the matrix first, then the matrix multiplication part can read data from transposed matrix with minimum pointer moves. Yet we still need to read columns in the first place, this task can be speeded up by parallelization because transposing for each element is independent.

- Parallelization considerations

Because the reading of matrix columns takes a lot of pointer moving steps, which is time consuming, we transpose the matrix by distributing this work among all available threads (8 threads).

Another block of time consuming work is the multiplication over values in columns and rows. Workload sharing is also enabled to distribute the work over all available threads.

- Method
  We made local variable and set as transpose of sq_matrix_2. Then we add those values to calculate for the results in nested loop.

- To be noticed:
  The cache efficiency was very similar to that of cache blocking methods. However, using both methods did not help the performance because both optimizations are related to cache that it does not sum up the performance.

Optimization results:

- Performance before optimization

```
./matrix_mul -i ../matrix_mul_02.dat
Test Case 1        0.154053 milliseconds
Test Case 2        0.00317383 milliseconds
Test Case 3        0.0119629 milliseconds
Test Case 4        0.0119629 milliseconds
Test Case 5        476.654 milliseconds
```

- Performance after optimization

```
./matrix_mul -i ../matrix_mul_02.dat
Test Case 1        1.00684 milliseconds
Test Case 2        0.941162 milliseconds
Test Case 3        0.917969 milliseconds
Test Case 4        0.918945 milliseconds
Test Case 5        280.923 milliseconds
```

58% improvement

Auto Vectorization for KMeans

Optimization goal:

- Hardware resources being optimized toward?

SIMD (Single Instruction, Multiple data) is optimized toward in the omp version of kmeans code.

- What is the specification of the hardware you are optimizing for?

The CPU model of ghc clusters is "Intel(R) Xeon(R) CPU W3520" and the level of SSE supported is "SSE4.2".

Optimization process:

- Data considerations

In SSE4.2 supported instruction set, the data can be stored in 128 bits, so for float type (4 bytes), the SIMD capacity can be used up to 4 times.

- Parallelization considerations: example code that is vectorizable is described below:
  - Euclid_dist_2:

According to the code omp_kmeans.c, the multicore has been utilized to parallelize the computation of method "find_nearest_cluster" for the large set of objects. However, within the thread running "find_nearest_cluster", the "euclid_dist_2" of objects to clusters are computed. Single instruction "-" is to be executed between points twice to compute square of distance in each dimension. Single instruction "*" is to be executed multiple times (by for loop) on each dimension. Therefore, SIMD capability can be utilized by vectorize the "for loop" of dimension in "euclid_dist_2".

- Method:

To utilize SIMD, auto-vectorization compilers are available in GCC. We enable it by making changes in Makefile and add flags of SIMD when source code is compiled.

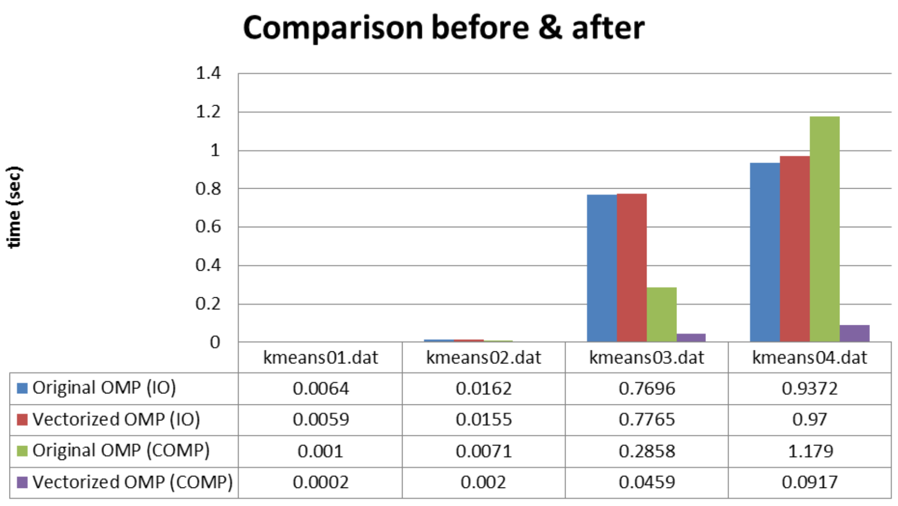
$(SIMDFLAGs) = -msse2 –msse3 –O3 –ffast-math –ftree-vectorize –ftree-vectorizer-verbose=2 -lm

Optimization results:

- By enabling "-ftree-vectorizer-verbose=2" in Makefile and compiling the code, we can see following hints:
  - "omp_kmeans.c:71: note: vectorized 4 loops in function."
- Performance **before** & **after** optimization:

We test all four kmeans01 ~ kmeans04.dat for the case the number of clusters equals to 3 (n = 3). We compare the IO and Computation time separately between the original omp code and the vectorized version. Finally, we compute the total speedup for all tests in total time (IO + computation time).

- Comparison of I/O time and Computation time before & after.



	kmeans01.dat	kmeans02.dat	kmeans03.dat	kmeans04.dat
🟦 Original OMP (IO)	0.0064	0.0162	0.7696	0.9372
🟥 Vectorized OMP (IO)	0.0059	0.0155	0.7765	0.97
🟩 Original OMP (COMP)	0.001	0.0071	0.2858	1.179
🟪 Vectorized OMP (COMP)	0.0002	0.002	0.0459	0.0917

- Speedup:

Summation over 4 tests	I/O Time	Computation Time	Total Time	Speedup
Original OMP	1.7294	1.4729	3.2023	1.678618231
Vectorized OMP	1.7679	0.1398	1.9077	

TEAM 34: Sungjae Lee sungjael@andrew.cmu.edu
Chen Wang chenw@andrew.cmu.edu

Kmeans Optimization using OpenMP TASK Construct

Optimization goal:

Task construct is a new feature with OpenMP 3.0. It defines an explicit task, which will be executed by the encountering thread in a multi-core environment.

- What is the specification of the hardware you are optimizing for?

Optimization process

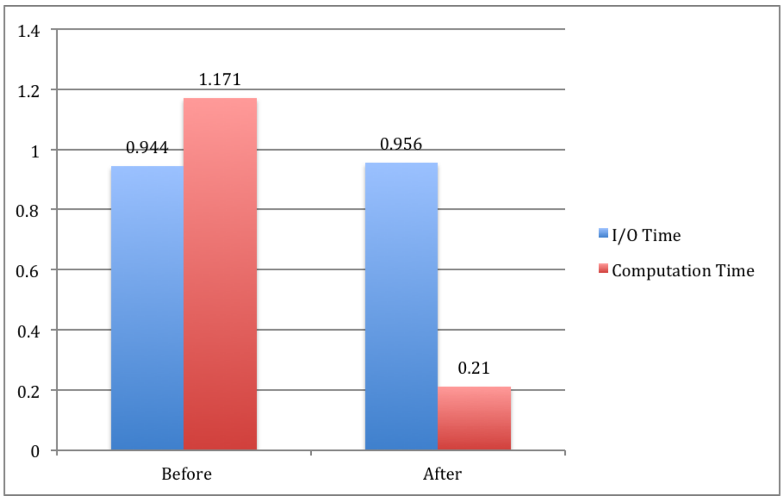
- Data considerations: throughout the given Kmeans implementation, there exists much openMP optimization around computational expensive loops. But, during the second stage of the algorithm, the loop of updating new cluster centers for each data point increases linearly with the number of data points. For large dataset to be clustered, e.g. kmeans04.dat with 48.8k data points, the loop may need to call function find_nearest_neighbor for 48.8k times.

```
#pragma omp parallel \
                shared(objects,clusters,membership,local_newClusters,local_newClusterSize)
        {
            int tid = omp_get_thread_num();
            #pragma omp for \
                    private(i,j,index) \
                    firstprivate(numObjs,numClusters,numCoords) \
                    schedule(static) \
                    reduction(+:delta)
            for (i=0; i<numObjs; i++) {
                /* find the array index of nestest cluster center */
        #pragma omp task
                index = find_nearest_cluster(numClusters, numCoords,
                                    objects[i], clusters);
                /* if membership changes, increase delta by 1 */
                if (membership[i] != index) delta += 1.0;
                /* assign the membership to object i */
                membership[i] = index;
                /* update new cluster centers : sum of all objects located
                   within (average will be performed later) */
                local_newClusterSize[tid][index]++;
                for (j=0; j<numCoords; j++)
                    local_newClusters[tid][index][j] += objects[i][j];
            }
        } /* end of #pragma omp parallel */
```

- Parallelization considerations: in order for OpenMP FOR directive to work, the action inside the loop should be pre-deterministic, however, in the code segment shown above; the function *find_nearest_cluster* passes distinct data point each time. Meanwhile, function *find_nearest_cluster* is perfectly independent for each data point so that it does not suffer loop carried dependence and is parallelizable.

Optimization results:

- Performance before&after optimization: We use the kmeans04.dat for illustrative benchmark and run the algorithms (k=3) for 30 iterations: although I/O time has slight increase, we see computation time drops significantly by over 500%.



TEAM 34: Sungjae Lee sungjael@andrew.cmu.edu
Chen Wang chenw@andrew.cmu.edu

Function Inlining for KMeans

Optimization goal:

- Hardware resources being optimized toward? (cache? SIMD? multicore?)
  Less instruction
- What is the specification of the hardware you are optimizing for?
- Intel(R) Xeon(R) CPU W3520 @ 2.67GHz
- 32kb*4 L1 cache

Optimization process:

- Data considerations
  We saw that find_nearest_cluster() is called my main commutating loop a lot.
  So we thought if we inline calculating Euclid_dist_2() which is inside of find_nearest_cluster(), it would increase the speed in big time.
  However; we could not observe any performance improvement.
  The only reason we think of it is that vectorization optimization might not be go well with function inlining.

Optimization results:

- Performance before optimization

```
Input file:         kmeans04.dat
numObjs               = 488565
numCoords             = 8
numClusters         = 3
threshold           = 0.0010
I/O time                       =        0.9301 sec
Computation timing =         0.0821 sec
```

- Performance after optimization

```
Input file:         kmeans04.dat
numObjs               = 488565
numCoords             = 8
numClusters         = 3
threshold           = 0.0010
I/O time                       =        0.9823 sec
Computation timing =         0.1163 sec
```

Contribution

Optimizations handled by Sung Jae

Loop_unrolling
Cache blocking
Function Inlining
OMP parallel

Optimizations handled by Chen

Transposing the matrix
Vectorization
OMP parallel
OpenMP TASK Construct

Sung Jae started with matrix_multiplication and Chen did the kmeans, and after each day, both looked at the others code and tried different optimizations to see if it works.