Optimization 1 for Matrix Multiplication: Setup no more than 1024 threads per block

- **Optimization goal:**
  - Hardware resources being optimized toward: GPU
  - What is the specification of the hardware you are optimizing for?
    GeForce GTX 1080, 20 Multiprocessors, maximum number of threads per multiprocessor is 2048.

- **Optimization process:**
  - Data & parallelization consideration:
    The example code use only one block and number of sq_dimension * sq_dimension threads. Due to the hardware limitation of GTX 1080, each block can only use 1024(32*32) threads so that test case 4(33*33), and test case 5(1000*1000) cannot be computed. Therefore, we set fixed number 1024(32*32) threads per block as maximum value of GTX 1080, and also set the number of blocks as (length of input matrix * length of input matrix+1023) / 1024. In this way, any size of input matrix can be completely scanned and achieve the matrix multiplication.

- **Optimization results:**
  - Performance before optimization: Test case 3: 0.154 Gflop/s, Test case 4: Failure
  - Performance after optimization: Test case 3: 0.151 Gflop/s, Test case 4: 0.204896 Gflop/s

Optimization 2 for Matrix Multiplication: Use GPU shared memory

- **Optimization goal:**
  - Hardware resources being optimized toward: GPU shared memory.
  - What is the specification of the hardware you are optimizing for? GeForce GTX 1080, Maximum amount of shared memory per multiprocessor: 96KB.

- **Optimization process:**
  - Data & parallelization consideration:
    Although the speed of GTX1080 global memory GDDR5X (448GB/s) has been improved a lot from previous generation product, the speed of shared memory(was 1244GB/s in GTX 280, cannot find the data of GTX 1080) still faster than global memory. Since we use 32*32 threads in each block, it's intuitive to pre-load 32*32*size of float in each block before computing. And while we loaded from global memory to shared memory, we also transferred to 2-D form for better readability and avoid memory bank conflicts. We also used "nvprof" to check "shared_store_transactions_per_request" and "shared_load_transactions_per_request". They are 1.2 and 1.0 respectively which is low enough to prevent damaging the performance.

- **Optimization results:**
  - Performance before optimization: Test case 3: 0.154 Gflop/s, Test case 4: Failure, Test case 5: Failure
  - Performance after optimization: Test case 3: 0.151 Gflop/s, Test case 4: 0.204896 Gflop/s, Test case 5: 537.279 Gflop/s

Optimization 3 for Matrix Multiplication: Use loop unrolling

- **Optimization goal:**
  - Hardware resources being optimized toward: GPU.
  - What is the specification of the hardware you are optimizing for?
    GeForce GTX 1080, 20 Multiprocessors.

- **Optimization process:**
  - Data & parallelization consideration:
    The branch that happens in for loop would take much time for compiler. To optimize loop execution, we apply loop unrolling method to reduce the frequency of branch instructions and increase more independent operation to be scheduled. In CUDA, the instruction (#pragma unroll) can control unrolling of the given loop and improve the ability of compiler to mix instruction and increase instructions executed per cycle.

- **Optimization results:**
  - Performance before optimization: Test case 5: 537.279 Gflop/s
  - Performance after optimization: Test case 5: 580.194 Gflop/s

Optimization 1 for K-means: Setup object threshold for small amount of object

- **Optimization goal:**
  - Hardware resources being optimized toward: CPU and GPU
  - What is the specification of the hardware you are optimizing for?
    Intel Xeon CPU E5-1660 v4, GeForce GTX 1080

- **Optimization process:**
  - Data & parallelization consideration:
    Increase the utilization of CPU to minimize the data transmission time between device and host. While the data is not large enough, the transmission between device and host would take much more time than reducing computing time by GPU. Consider the data may not be always that big, we setup the threshold (only when number of object > 5000, we use GPU parallelized computing) to optimize the efficiency of transmission between device(GPU) and host(CPU).

- **Optimization results:**
  - Performance before optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0102s	0.0228 s	0.6070 s	0.7638 s
Computation time	0.3637 s	0.3417 s	1.1308 s	4.3756 s

  - Performance after optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0113s	0.0189 s	0.5791 s	0.6561 s
Computation time	0.0343 s	0.3464 s	0.4218 s	0.7157 s

Optimization 2 for K-means: Use GPU shared memory

- **Optimization goal:**
  - Hardware resources being optimized toward: GPU shared memory.
  - What is the specification of the hardware you are optimizing for?
    GeForce GTX 1080, Maximum amount of shared memory per multiprocessor: 96KB.

- **Optimization process:**
  - Data & parallelization consideration:
    In example code, the numThreadsPerClusterBlock is 128 which is not enough to compute numObjs for test case 3 and 4, so we use maximum number of threads per block 1024 to apply all the cases.
    As we cannot determine the size of shared memory before compiling, we need to allocate and use shared memory through extern__shared__ in the function of __global__ during execution. While the data need multiple threads visit (ex: membershipChanged and cluster), we compute and find the array index to use shared memory.

- **Optimization results:**
  - Performance before optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0102s	0.0228 s	0.6070 s	0.7638 s
Computation time	0.3637 s	0.3417 s	1.1308 s	4.3756 s

  - Performance after optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0113s	0.0189 s	0.5791 s	0.6561 s
Computation time	0.0343 s	0.3464 s	0.4218 s	0.7157 s

Optimization 3 for K-means: Change array layout to SOA (Struct of Array)

- **Optimization goal:**
  - Hardware resources being optimized toward: GPU thread schedule.
  - What is the specification of the hardware you are optimizing for?
    GeForce GTX 1080, 20 Multiprocessors, maximum number of threads per multiprocessor is 2048.

- **Optimization process:**
  - Data & parallelization consideration:
    By using SOA (Struct of Array), the thread can visit object in sequence which have better cache locality and increase the hit of cache. On the other hand, if we use AOS (Array of Struct), the thread have to load the unused data with low efficiency cache locality, the hit rate could be only 1/32. Therefore, we flip the nest order to change the layout of newClusters to [numClusters][numCoords] which can access the data more efficiently.

- **Optimization results:**
  - Performance before optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0102s	0.0228 s	0.6070 s	0.7638 s
Computation time	0.3637 s	0.3417 s	1.1308 s	4.3756 s

  - Performance after optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0113s	0.0189 s	0.5791 s	0.6561 s
Computation time	0.0343 s	0.3464 s	0.4218 s	0.7157 s

Optimization 4 for K-means: Transfer summing cluster members and clusters size and updating clusters' centroids into cuda parallelized computing

- **Optimization goal:**
  - Hardware resources being optimized toward: GPU
  - What is the specification of the hardware you are optimizing for? GeForce GTX 1080, 20 Multiprocessors, maximum number of threads per multiprocessor is 2048.
- **Optimization process:**
  - Data & parallelization consideration: We noticed that after computing delta, updating cluster size and summing up cluster members for calculating centroids were executed in sequential by CPU. However, most of them can be parallelized by GPU. For updating cluster size and cluster size, it's a mapreduce problem. Since for all of test cases the number of cluster is at most 32, it would take little part of time of whole procedure, we just simply use atomic operation to compute. Although we cannot reduce the computing time, we can save the time of copying data from device(GPU) to host(CPU). For update clusters' centroids, it can be completely parallelized by each cluster and coordinate, so we use 'number of clusters' blocks and 'number of coordinates' threads to span the function.
- **Optimization results:**
  - Performance before optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0102s	0.0228 s	0.6070 s	0.7638 s
Computation time	0.3637 s	0.3417 s	1.1308 s	4.3756 s

  - Performance after optimization:

	Test case 1	Test case 2	Test case 3	Test case 4
numObjs	351	7089	191681	488565
I/O time	0.0113s	0.0189 s	0.5791 s	0.6561 s
Computation time	0.0343 s	0.3464 s	0.4218 s	0.7157 s