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

Bug fixes for cuda_kmeans

Problems in the code

- In the original cuda_kmeans.cu code, after finding the nearest cluster for each object, there would be $numClusterBlocks = ceil\left(\frac{numObjs}{numThreadsPerClusterBlock}\right)$ intermediate results. For kmeans03.dat, there are totally 191681 objects and the number of threads per cluster block is set to be 128, so the size of intermediate results is 1498. For kmeans04.dat, the size of intermediate results would be 3817.
- According to the original code, the cuda_kmeans method will call following method to reduce an array of data into one return value. This method will use "numReductionThreads" to reduce the "deviceIntermediates" results within "1" block. However, the "numReductionThreads" is set to be the nextPowerOfTwo of "numClusterBlocks" (2048 for kmeans03.dat, and 4096 for kmeans04.dat) in original code, which exceed the maximum threads per block (1024) in GTX 480.

```
compute_delta <<< 1, numReductionThreads, reductionBlockSharedDataSize >>>
        (deviceIntermediates, numClusterBlocks, numReductionThreads);
```

Bug Fixing

- We first make changes to the number of blocks used to reduce the "deviceIntermediates". For the cases where the size of $numClusterBlocks$ is larger than the 1024 (maximum threads per block), we ceil it as 1024 and add another block to reduce the "deviceIntermediates". (We later change the block size to 512 for maximizing scheduling throughput. It will be descibed in the following pages.)

```
unsigned int orgReductionThreads = nextPowerOfTwo(numClusterBlocks);
const unsigned int numReductionBlocks = orgReductionThreads/1024 + 1;
const unsigned int numReductionThreads = orgReductionThreads>1024?1024:orgReductionThreads;
……
compute_delta <<< numReductionBlocks, numReductionThreads, reductionBlockSharedDataSize >>>
        (deviceIntermediates, numClusterBlocks, numReductionThreads);
```

- Within the method of "compute_delta", we make changes to add up the reduced values from reduced values in all blocks.

```
if ((threadIdx.x == 0) && (blockIdx.x == 0)) {
      deviceIntermediates[0] = intermediates[0];
} else if (threadIdx.x == 0) {
      deviceIntermediates[0] += intermediates[0];
}
```

Performance of original cuda_kmeans.cu after fixing bugs

```
./cuda_main.org -i ~/645/data/kmeans01.dat -n 3 -o
Input file:     /afs/andrew.cmu.edu/usr4/chenw/645/data/kmeans01.dat
numObjs       = 351
numCoords     = 34
```

numClusters = 3
threshold = 0.0010
Loop iterations = 7
I/O time = 0.0062 sec
Computation timing = 0.0492 sec

[chenw@ghc66 kmeans]$ ./cuda_main.org -i ~/645/data/kmeans02.dat -n 3 -o
Input file: /afs/andrew.cmu.edu/usr4/chenw/645/data/kmeans02.dat
numObjs = 7089
numCoords = 4
numClusters = 3
threshold = 0.0010
Loop iterations = 7
I/O time = 0.0144 sec
Computation timing = 0.0450 sec

[chenw@ghc66 kmeans]$ ./cuda_main.org -i ~/645/data/kmeans03.dat -n 3 -o
Input file: /afs/andrew.cmu.edu/usr4/chenw/645/data/kmeans03.dat
numObjs = 191681
numCoords = 22
numClusters = 3
threshold = 0.0010
Loop iterations = 4
I/O time = 0.7763 sec
Computation timing = 0.2039 sec

[chenw@ghc66 kmeans]$ ./cuda_main.org -i ~/645/data/kmeans04.dat -n 3 -o
Input file: /afs/andrew.cmu.edu/usr4/chenw/645/data/kmeans04.dat
numObjs = 488565
numCoords = 8
numClusters = 3
threshold = 0.0010
Loop iterations = 14
I/O time = 0.9196 sec
Computation timing = 0.4730 sec

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

Maximizing Scheduling Throughput for cuda_kmeans

Optimization Goal

- Occupancy: it is the ability of a CUDA kernel to occupy concurrent contexts in a Streaming Multiprocessor
    - Namely, it is the ratio of active warps to the maximum number of warps supported.

Optimization Process

- After compiling the original cuda_kmeans.cu, we notice that there are two kernels in the code, “find_nearest_cluster” and “compute_delta”. By enabling “--ptxas-options=-v” flag when we compile, the command line show us the registers and shared memory usage for these two kernels as follows.

```
ptxas info    : Function properties for _Z20find_nearest_clusteriiiPfS_PiS0_
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 16 registers, 80 bytes cmem[0]
ptxas info    : Compiling entry function '_Z13compute_deltaPiii' for 'sm_20'
ptxas info    : Function properties for _Z13compute_deltaPiii
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 9 registers, 48 bytes cmem[0]
```

- We then use “CUDA_Occupancy_Calculator” to estimate the occupancy of each muti-processor. It can be seen that, the occupancy for kernel “find_nearest_cluster” is only 67% and it can be improved to 100% if we change the number of threads per block to 256. Similar for the kernel “compute_delta”, the occupancy is only 67% with 1024 threads per block and can be improved to 100% if the number of threads per block is set to be 512.

Optimization Results

We compare the computation time between the optimized code with the original cuda_kmeans.cu in the following table.

Table 1 Comparison of Computation Time

(seconds)	kmeans01	kmeans02	kmeans03	kmeans04	Total	Total Speedup
Original Version	0.0496	0.0447	0.2292	0.4144	0.7379	
Scheduling Maximized Version	0.0445	0.0448	0.2011	0.1016	0.392	1.88X

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

SOA vs AOS in cuda_kmeans

Optimization Goal

- Minimizing memory bandwidth to DRAM
    - In the original cuda_kmeans code, the coordinates of objects are stored in the data structure of SOA. It would be very effective in reducing the memory bandwidth if the number of objective is small but the number of coordinates is very large. However, in our testing cases kmeans03.dat and kmeans04.dat where the computation is very slow, the number of coordinates is much less than the number of objects.

Optimization Process

- Change the store of objects from the structure of Struct of Arrays to the structure of Array of Struct.
    - Specifically, we comment all the codes of matrix transposition on objects and clusters.
    - We copy the objects and clusters data to cuda memory using the data structure of AOS
- Making changes in method “euclid_dist_2” to retrieve coordinates in row-wise.

Optimization Results

We run the original version, the scheduling maximized version and the AOS optimized version (with scheduling maximization in it) on the same cluster machine at nearly the same time. The comparison of computation time is shown in the following table.

Table 2 Comparison of Computation Time

(seconds)	kmeans01	kmeans02	kmeans03	kmeans04	Total	Total Speedup
Original Version	0.0613	0.0447	0.2313	0.4344	0.7717	
Scheduling Maximized Version	0.0435	0.0445	0.1999	0.1008	0.3887	1.99
AOS Optimized Version	0.0440	0.0446	0.1735	0.0765	0.3386	2.27

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

Auto Vectorization in cuda_kmeans

Optimization goal:

- Hardware resources being optimized toward?
  - SIMD (Single Instruction, Multiple data) is optimized toward in the cuda version of kmeans code.
- What is the specification of the hardware you are optimizing for?
  - It is said that "Modern graphics processing units (GPUs) are often wide SIMD implementations, capable of branches, loads, and stores on 128 or 256 bits at a time." on wiki, but we did not find the actual vector size for GTX-480.

Optimization process:

- Data considerations

We do not know what is the exact vector size for the core of GTX-480, so we tried to add auto-vectorization flags on Makefile and to see if there is any improvement.

To utilize SIMD, auto-vectorization compilers are available in nvcc. We enable it by making changes in Makefile and add flags of SIMD when source code is compiled.
$(SIMDFLAGs) = –O3

Optimization Results

All the previous optimizations have been included in the auto vectorized version. We can compare all versions of code in the following table.

Table 3 Comparison of Computation Time

(seconds)	kmeans01	kmeans02	kmeans03	kmeans04	Total	Total Speedup
Original Version	0.0510	0.0451	0.2457	0.4705	0.8123	
Scheduling Maximized Version	0.0474	0.0461	0.2172	0.1095	0.4202	1.93X
AOS Optimized Version	0.0453	0.0452	0.1813	0.0791	0.3509	2.31X
Auto Vectorized Version	0.0448	0.0450	0.0810	0.0550	0.2258	3.60X

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

Bug Fixes in Matrix Multiplication

Problems in the code

- In the original matrix_mul.cu code, matrix_multiplication method only calls 1 block (dimGrid(1,1)) to run matrix_mul_kernel. For cases of sq_dimension = 5, 8, 10, 32, there would be no problems because they only call, 25, 64, 100, 1024 threads in that block to run matrix_mul_kernel. However, for cases of sq_dimension = 33, 1000, $sq\_dimension^2$ number of threads running on one block will exceed the maximum number of threads allowed in GTX-480, which is 1024.

Bug Fixing

- So, similarly as what we did in kmeans, we fix the size of block as 1024 and invoke more blocks when the sq_dimension is large.

```
dim3 dimBlock(BLK_SZ, BLK_SZ);
const unsigned int dimSz = (sq_dimension + BLK_SZ - 1) / BLK_SZ;
dim3 dimGrid(dimSz,dimSz);
```

- Accordingly, for the blocks that contain thread id that exceeds the sq_dimension, we avoid running the data reading of matrice because it will read some wrong memory.

```
if ((tx < sq_dimension) && (ty < sq_dimension)) {
        for(int k = 0; k < sq_dimension; k++) {
                sum += sq_matrix_1[ty*sq_dimension + k] * sq_matrix_2[k*sq_dimension + tx];
        }
        sq_matrix_result[ty*sq_dimension + tx] = sum;
}
```

Performance of original matrix_mul.cu after fixing bugs

```
Test Case 1     0.00399218 Gflop/s
Test Case 2     0.00792774 Gflop/s
Test Case 3     0.257604 Gflop/s
Test Case 4     0.250275 Gflop/s
Test Case 5     63.5233 Gflop/s


OK (1 tests)
```

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

Shared Memory Optimization in Matrix Multiplication

Optimization goal

- Take advantage of faster memory bandwidth
  - We need to load the data from DRAM to shared memory before we perform work on those data.

Optimization Process

- Referring to the cache blocking scheme we used in the Project 1, we divide a large matrix into blocks. For the computation of each block, we analyze all the computations that will be done in this block and divide those computation processes by blocks of data they need to reference. Then, the code only loads needed blocks of data once a time and perform all the computations related to those data before it loads other data.

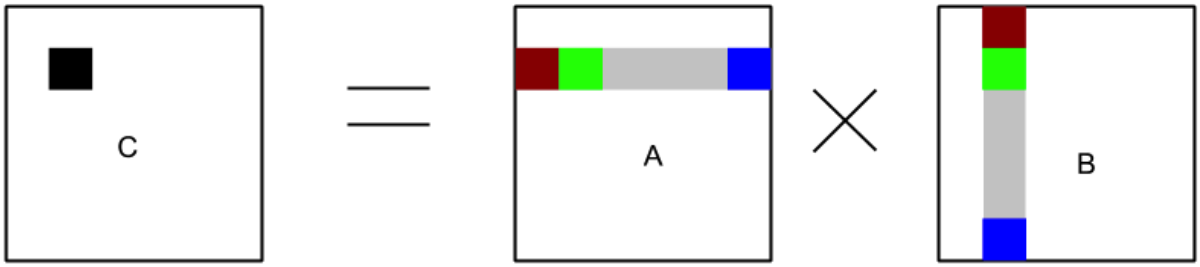

Figure 1 Cache Blocking and Shared Memory Loading

- In Figure 1, as it is shown, if we want to compute the black block of data in matrix C, we need to multiply the horizontal gray stripe in A with the vertical gray strip in B. During the computation, the red block in A needs to multiply red block in B and red block in A will only be used when red block in B is ready. Therefore, we load red blocks in A and B into the shared memory, compute their multiplication at once. Then, we load green blocks and so on, until the blue blocks. Finally, the results those blocks can be summed up as the black block in result matrix.

Performance Improvement

Table 4 Comparison of Computation Throughput

Gflops/s	case 1	case 2	case 3	case 4	case 5	total
Original Version	0.00402569	0.00791323	0.261642	0.278204	63.6513	64.203
Memory Shared Version	0.0039822	0.00787188	0.267649	0.282983	147.525	148.087

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

Scheduling Throughput Maximization in Matrix Multiplication

Optimization goal

- Occupancy: it is the ability of a CUDA kernel to occupy concurrent contexts in a Streaming Multiprocessor
    - Namely, it is the ratio of active warps to the maximum number of warps supported.

Optimization Process

- By enabling "--ptxas-options=-v" flag when we compile, the command line show us the registers and shared memory usage for the kernels "matrix_mul_kernel" as follows:

```
ptxas info    : Function properties for _ZN4cuda17matrix_mul_kernelEPfS0_S0_i
  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 22 registers, 8192 bytes smem, 60 bytes cmem[0]
```

- We set the cache block size now as 32, so the total block contains $32 \times 32 = 1024$ threads per block, which is also the maximum number of threads per block. By using "CUDA_Occupancy_Calculator" to estimate the occupancy of each multiprocessor, tt can be seen that, the occupancy for kernel "matrix_mul_kernel" is only 67% and it can be improved to 100% if we change the number of threads per block to 256 (BLK_SZ = 16), where the shared memory will be reduced to 2048 bytes at the same time.

Optimization Results

This scheduling throughput maximized version include the optimization of memory share.

Table 4 Comparison of Computation Throughput

Gflops/s	case 1	case 2	case 3	case 4	case 5	total
Memory Shared Version	0.00384864	0.00759086	0.260088	0.274292	146.858	147.4
Scheduling Throughput Maximized Version	0.00391403	0.00768303	0.264797	0.279267	157.802	158.4

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

Tile Buffering ---- Advanced Shared Memory

Optimization goal

- Take advantage of faster memory bandwidth
    - We need to load the data from DRAM to shared memory before we perform work on those data.

Optimization Process

- From the Shared Memory optimization, we can see that the block size has been optimized to be set as 16 X 16. However, if we check the CUDA Occupancy calculater, we can see that the shared memory size can further be increased, which means we can load more blocks, make more computations and increase the throughput by reducing the total number of blocks. It can not be hard to infer that loading blocks also generate time costs.

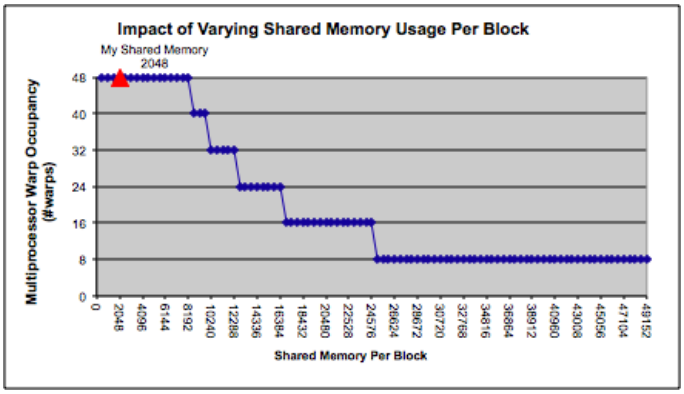

Figure 2 Memory Curve with fixed register number and block size
(Generated by CUDA_GPU_Occupancy_Calculator)

- Therefore, we introduce another constant called “TILE_SZ” to load multiple blocks to at once load multiple blocks in matrix2. We choose to tile buffering matrix2 in row-wise because matrix2 will be read in column-wise and the rows after each block would have been read from DRAM anyway.

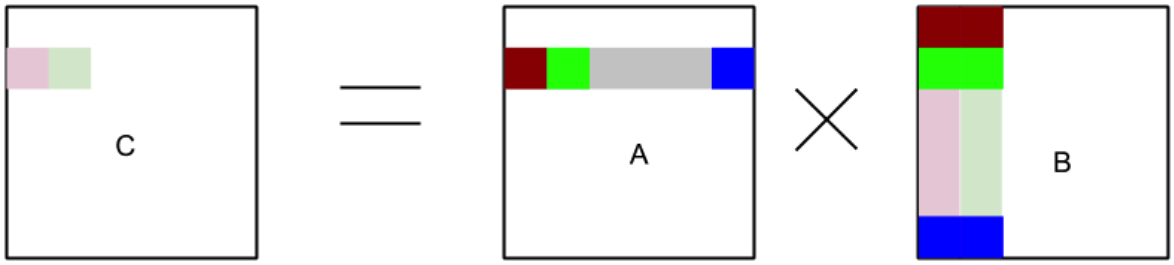

Figure 3 Schematic Graph of Tile buffering in matrix multiplication (TILE_SZ = 2)

- Figure 3 simply shows how tile buffering works in matrix multiplication. Therefore, instead

of computing one block per time, two blocks are computed. Only one block in A will be loaded. Because matrice are stored in row-wise, loading a block in B will read through all rows of that block in B. Therefore, multiple blocks in a row stripe in B are tiled up.

Optimization Results

In order to test how the tile_sz can improve the computational throughput, we test two cases of tile buffering version, TILE_SZ = 2 and TILE_SZ = 4. The computational throughput are compared with scheduling throughput maximized version as follows.

Table 4 Comparison of Computation Throughput

Gflops/s	case 1	case 2	case 3	case 4	case 5	total
Scheduling Throughput Maximized Version	0.00386833	0.00766035	0.260165	0.277008	157.645	158.19
TILE_SZ = 2	0.00389657	0.00765181	0.262239	0.277627	185.819	186.37
TILE_SZ = 4	0.00385745	0.00762532	0.26353	0.281782	206.939	207.5

Contribution

Optimizations handled by Sung Jae

Matrix Multiplication

Optimizations handled by Chen

kmeans

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.