KMeans Clustering Algorithm Parallelization With CUDA Task 3

Presenting a Working Solution for Various Devices:

In this task, I am proposing an implementation of a parallel K-means algorithm based on Sequential C++ code, called SKmeans. The aim is so that we can use it as a benchmark to evaluate the performance of our CUDA implementation. Before introducing our Skmeans algorithm, we first give a short description of the SKmeans (Sequential K-means) algorithm in Section A, and then we give the detailed evaluation of the two implementations side by side to compare the speed up for Section B.

Section A

A SKmeans Algorithm:

The clustering algorithm K-means is one of the most popular partitioning clustering algorithms. The main idea in the algorithm is to define K centroids (one for each cluster). These centroids should be placed in a cunning way because different locations cause different results. In the SKmeans algorithm, the objective function J is defined as in (1). SKmeans aims to minimize the objective function, i.e., a squared error function. In (1), J refers to the distance index of N data objects from the corresponding cluster centroids, n x (1≤n≤N) indicates a data point, and k c (1≤k≤K) specifies the cluster centroid. 2 nkx c− is the distance measure between n x and k c . In (2), k μ (1≤k≤K) refers to the mean of data points that belong to cluster k, and k N indicates the number of objects belonging to cluster k. In our SKmeans algorithm, the number of clusters K is a user-specified parameter. First, read N objects from the input file. The initial K-centroids are randomly selected, defined as k μ (1 ≤ k ≤ K).

Second, the SKmeans algorithm iteratively assigns each object into the corresponding cluster by the minimum distance. When all objects are assigned, update the K centroids. This process will be repeated until the user-specified threshold is met. The SKmeans algorithm is shown in Fig. 1

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} ||x_n - c_k||^2 \qquad (1)$$

$$\mu_1 = \frac{1}{N_k} \sum_{n \varepsilon c_k} x_n \qquad (2)$$

SKmeans algorithm Implementation

```
Input: number of clusters K, number of data objects N
Output: K centroids
1: Read N objects from the file;
2: Randomly select K points as the initial cluster centroids, denoted as k μ (1≤k≤K);
3: Calculate J in Formula (1), denoted by J';
4: Assign each object n (1≤n≤N) to the closest cluster;
5: Calculate new centroid of each cluster k μ in Formula (2) ;
6: Recalculate J in Formula (1);
7: Repeat steps 3-6 until J'- J < threshold;
8: Output the clustering results: K centroids;
```

SKmeans C++ Implementation:

```cpp
#include <algorithm>
#include <cstdlib>
#include <limits>
#include <random>
#include <vector>

struct Point {
  double x{0}, y{0};
```

```
};

using DataFrame = std::vector<Point>;

double square(double value) {
  return value * value;
}

double squared_l2_distance(Point first, Point second) {
  return square(first.x - second.x) + square(first.y - second.y);
}

DataFrame k_means(const DataFrame& data,
                  size_t k,
                  size_t number_of_iterations) {
  static std::random_device seed;
  static std::mt19937 random_number_generator(seed());
  std::uniform_int_distribution<size_t> indices(0, data.size() - 1);

  // Pick centroids as random points from the dataset.
  DataFrame means(k);
  for (auto& cluster : means) {
    cluster = data[indices(random_number_generator)];
  }

  std::vector<size_t> assignments(data.size());
  for (size_t iteration = 0; iteration < number_of_iterations;
++iteration) {
    // Find assignments.
    for (size_t point = 0; point < data.size(); ++point) {
      double best_distance = std::numeric_limits<double>::max();
      size_t best_cluster = 0;
      for (size_t cluster = 0; cluster < k; ++cluster) {
        const double distance =
            squared_l2_distance(data[point], means[cluster]);
        if (distance < best_distance) {
          best_distance = distance;
          best_cluster = cluster;
        }
      }
      assignments[point] = best_cluster;
    }

    // Sum up and count points for each cluster.
    DataFrame new_means(k);
    std::vector<size_t> counts(k, 0);
    for (size_t point = 0; point < data.size(); ++point) {
      const auto cluster = assignments[point];
      new_means[cluster].x += data[point].x;
      new_means[cluster].y += data[point].y;
      counts[cluster] += 1;
    }

    // Divide sums by counts to get new centroids.
    for (size_t cluster = 0; cluster < k; ++cluster) {
      // Turn 0/0 into 0/1 to avoid zero division.
      const auto count = std::max<size_t>(1, counts[cluster]);
      means[cluster].x = new_means[cluster].x / count;
      means[cluster].y = new_means[cluster].y / count;
    }
  }

  return means;
}
```

You can compile k-means.cpp with `g++ k-means.cpp -lm -O -0 k-means`. *This is not so much about the k-means algorithm itself as it is about comparing the performance of implementations of k-means across various platforms, so we need to know how fast our c++ implementation is. Putting simple* ***time.time()*** *calculations immediately around the function invocations, I get the following results for a dataset of* ***n∈100,100000 points, T=300 iterations and k = clusters, using the average time of 5 runs:***

Let’s see how fast it is!

IMPLEMENTATION	**N = 100**	**N = 100, 000**
C++	**0.00054s**	**0.26804s**

Section B

CUDA

As mentioned at the very beginning how obviously parallelizable k-means is. Why do I think so? Well, let's look at the definition of the *assignment* step:

1. Compute the distance from each point x_i to each cluster centroid μ_j
2. Assign each point to the centroid it is closest to.

What's important to notice here is that each data point xi does *its own thing*, i.e. no information or data is shared across individual data points, except for the here immutable cluster centroids. This is nice, because complexity in parallel programming arises almost exclusively when data needs to be shared. If all we're doing is performing some computation on each point individually, then coding this up on a GPU is a piece of cake.

Things get less rosy when we consider the ***update* step,** where we recompute the cluster centroids to be the mean of all points assigned to a particular centroid. Essentially, this is an average reduction, just that we aren't averaging over all values in the dataset, but doing one reduction over each cluster's respective subset. The simplest way to do such a reduction is to use **an *atomic* counter**, **just we previously did with our Cuda Code.** This is slow since the atomic counter increment will be greatly contended and serialize all threads' accesses. However, it's easy to implement – so let's get to it! Here is the CUDA code:

```
#include <algorithm>

#include <cfloat>

#include <chrono>
```

```
#include <random>

#include <vector>

// A small data structure to do RAII for a dataset of 2-dimensional
points.

struct Data {

  explicit Data(int size) : size(size), bytes(size * sizeof(float)) {

    cudaMalloc(&x, bytes);

    cudaMalloc(&y, bytes);

  }



  Data(int size, std::vector<float>& h_x, std::vector<float>& h_y)

  : size(size), bytes(size * sizeof(float)) {

    cudaMalloc(&x, bytes);

    cudaMalloc(&y, bytes);

    cudaMemcpy(x, h_x.data(), bytes, cudaMemcpyHostToDevice);

    cudaMemcpy(y, h_y.data(), bytes, cudaMemcpyHostToDevice);

  }



  ~Data() {

    cudaFree(x);

    cudaFree(y);
  }

  void clear() {
    cudaMemset(x, 0, bytes);
    cudaMemset(y, 0, bytes);
  }

  float* x{nullptr};
  float* y{nullptr};
  int size{0};
  int bytes{0};
};

__device__ float
squared_l2_distance(float x_1, float y_1, float x_2, float y_2) {
  return (x_1 - x_2) * (x_1 - x_2) + (y_1 - y_2) * (y_1 - y_2);
}

// In the assignment step, each point (thread) computes its distance to
each
// cluster centroid and adds its x and y values to the sum of its closest
// centroid, as well as incrementing that centroid's count of assigned
points.
__global__ void assign_clusters(const float* __restrict__ data_x,
                                const float* __restrict__ data_y,
                                int data_size,
                                const float* __restrict__ means_x,
                                const float* __restrict__ means_y,
                                float* __restrict__ new_sums_x,
                                float* __restrict__ new_sums_y,
                                int k,
                                int* __restrict__ counts) {
  const int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= data_size) return;

  // Make global loads once.
  const float x = data_x[index];
  const float y = data_y[index];

  float best_distance = FLT_MAX;
  int best_cluster = 0;
  for (int cluster = 0; cluster < k; ++cluster) {
    const float distance =
        squared_l2_distance(x, y, means_x[cluster], means_y[cluster]);
    if (distance < best_distance) {
      best_distance = distance;
      best_cluster = cluster;
    }
  }

  // Slow but simple.
  atomicAdd(&new_sums_x[best_cluster], x);
  atomicAdd(&new_sums_y[best_cluster], y);
  atomicAdd(&counts[best_cluster], 1);
}

// Each thread is one cluster, which just recomputes its coordinates as the mean
// of all points assigned to it.
__global__ void compute_new_means(float* __restrict__ means_x,
                                  float* __restrict__ means_y,
                                  const float* __restrict__ new_sum_x,
                                  const float* __restrict__ new_sum_y,
                                const int* __restrict__ counts) {

  const int cluster = threadIdx.x;

  // Threshold count to turn 0/0 into 0/1.

  const int count = max(1, counts[cluster]);

  means_x[cluster] = new_sum_x[cluster] / count;

  means_y[cluster] = new_sum_y[cluster] / count;

}


int main(int argc, const char* argv[]) {

  std::vector<float> h_x;

  std::vector<float> h_y;


  // Load x and y into host vectors ... (omitted)


  const size_t number_of_elements = h_x.size();


  Data d_data(number_of_elements, h_x, h_y);


  // Random shuffle the data and pick the first

  // k points (i.e. k random points).

  std::random_device seed;
  std::mt19937 rng(seed());

  std::shuffle(h_x.begin(), h_x.end(), rng);

  std::shuffle(h_y.begin(), h_y.end(), rng);

  Data d_means(k, h_x, h_y);



  Data d_sums(k);



  int* d_counts;

  cudaMalloc(&d_counts, k * sizeof(int));

  cudaMemset(d_counts, 0, k * sizeof(int));



  const int threads = 1024;

  const int blocks = (number_of_elements + threads - 1) / threads;



     for   (size_t   iteration   =   0;   iteration   <   number_of_iterations;
++iteration)  {

    cudaMemset(d_counts, 0, k * sizeof(int));

    d_sums.clear();



    assign_clusters<<<blocks, threads>>>(d_data.x,

                                         d_data.y,
                                                d_data.size,
                                                d_means.x,
                                                d_means.y,
                                                d_sums.x,
                                                d_sums.y,
                                                k,
                                                d_counts);
    cudaDeviceSynchronize();
    compute_new_means<<<1, k>>>(d_means.x,
                                d_means.y,
                                d_sums.x,
                                d_sums.y,
                                d_counts);
    cudaDeviceSynchronize();
  }
}
```

This is largely unoptimized CUDA code that makes no effort to come up with an efficient parallel algorithm to perform the reduction (we'll get to one in a bit). I'll compile this with nvcc `nvcc gpu-kmeans.cu -lm gpu_kmeans` and run on a recent NVIDIA Geforce GTX GPU. And?

IMPLEMENTATION	**N = 100**	**N = 100, 000**
C++	**0.00054s**	**0.26804s**

CUDA	0.00956s	**0.0752s**

Interesting! Running a GPU for 100 data points is a little like launching a space rocket to get from the living room to the kitchen in your house: totally unnecessary, not using the full potential of the vehicle and the overhead of launching itself will outweigh any benefits once the rocket, or GPU kernel, is running. On the other hand, we see that GPUs simply *scale* so beautifully across data when looking at the 100k experiment. Whereas the assignment step in our previous CPU algorithm scaled linearly w.r.t. the number of observations in our dataset, the span complexity of our GPU implementation stays constant and only the overall work increases. That is, adding more data does not alter the overall execution time (in theory). Of course, this only holds if you have enough threads to assign one to each point. In my experiments here I will assume such favorable circumstances.

Now, even though we can be quite happy with this speedup already, we haven't really invested much effort into this. **Using atomic operations** is somewhat cheating and definitely does not use the full capacity of GPUs, since the 100,000 threads we launch ultimately have to queue up behind each other to make their increments. Also, there is one particularly awful line in the above code that makes the implementation slow and that is somewhat easy to fix, without requiring deep algorithmic changes. It's this one here:

```
const float distance =

    squared_l2_distance(x, y, means_x[cluster], means_y[cluster]);
```

I'm not talking about the function call, but about the global memory loads means_x[cluster] and means_y[cluster]. Having every thread go to global memory to fetch the cluster means is inefficient. One of the first things we learn about GPU programming is that understanding and utilizing the memory hierarchy in GPUs is essential to building efficient programs, much more so than on CPUs, where compilers or the hardware itself handle register allocation and caching for us. The simple fix for the above global memory loads is to place the means into shared memory and have the threads load them from there. The code changes are quite minor. First in the assign_clusters kernel:

```
_global__ void assign_clusters(const float* __restrict__ data_x,
                                const float* __restrict__ data_y,
                                int data_size,
                                const float* __restrict__ means_x,
                                const float* __restrict__ means_y,
                                float* __restrict__ new_sums_x,
                                float* __restrict__ new_sums_y,
                                int k,
                                int* __restrict__ counts) {
  // We'll copy means_x and means_y into shared memory.
  extern __shared__ float shared_means[];

  const int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= data_size) return;

  // Let the first k threads copy over the cluster means.
  if (threadIdx.x < k) {
    // Using a flat array where the first k entries are x and the last k
are y.
    shared_means[threadIdx.x] = means_x[threadIdx.x];
    shared_means[k + threadIdx.x] = means_y[threadIdx.x];
  }

  // Wait for those k threads.
  __syncthreads();

  // Make global loads once.
  const float x = data_x[index];
  const float y = data_y[index];

  float best_distance = FLT_MAX;
  int best_cluster = 0;
  for (int cluster = 0; cluster < k; ++cluster) {
    // Neatly access shared memory.
    const float distance = squared_l2_distance(x,
                                               y,
                                               shared_means[cluster],
                                               shared_means[k + cluster]);
    if (distance < best_distance) {
      best_distance = distance;
      best_cluster = cluster;
    }
  }

  atomicAdd(&new_sums_x[best_cluster], x);
  atomicAdd(&new_sums_y[best_cluster], y);
  atomicAdd(&counts[best_cluster], 1);
}
```

Then in the kernel launch:

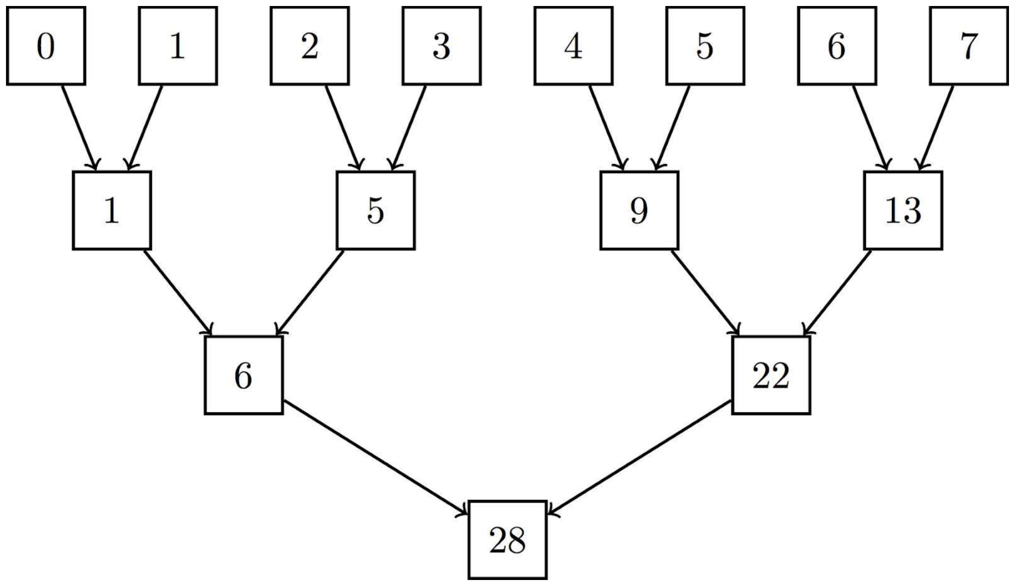
```
int main(int argc, const char* argv[]) {
  const int threads = 1024;
  const int blocks = (number_of_elements + threads - 1) / threads;
  const int shared_memory = d_means.bytes * 2;
  // ...
     for (size_t iteration = 0; iteration < number_of_iterations;
++iteration) {
    assign_clusters<<<blocks, threads, shared_memory>>>(d_data.x,
                                                          d_data.y,
                                                          d_data.size,
                                                          d_means.x,
                                                          d_means.y,
                                                          d_sums.x,
                                                          d_sums.y,
                                                          k,
                                                          d_counts);
    // ...
  }
}
```

Easy. Is the improvement noticeable? Let's see:

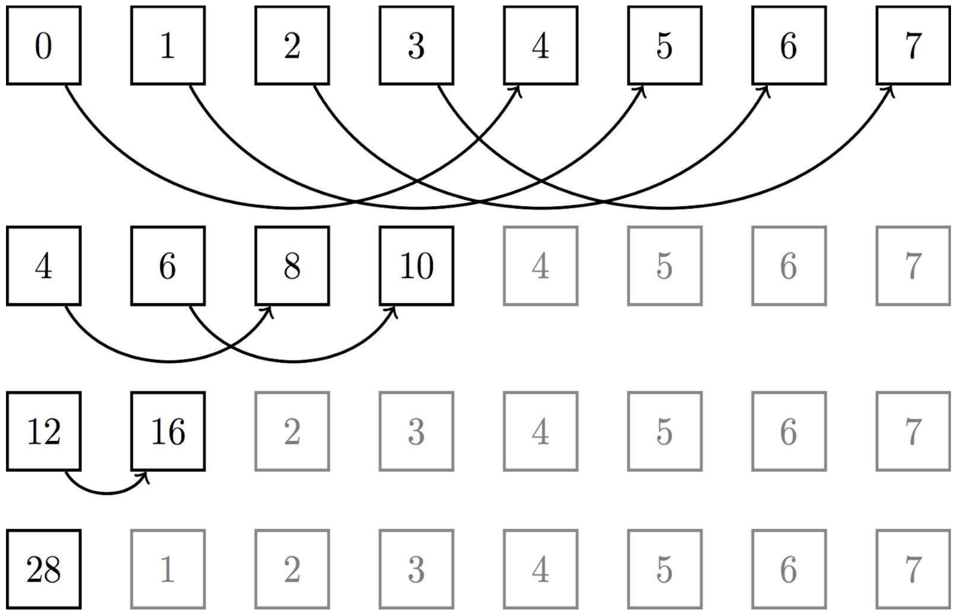
IMPLEMENTATION	**N = 100**	**N = 100, 000**
C++	**0.00054s**	**0.26804s**
CUDA	0.00956s	**0.0752s**
CUDA 2	0.00878s	**0.0611s**

Indeed, ***But we're still doing atomic increments****, so let's think a bit more.*

Serial reductions like the averaging operation we are performing during the update step scale linearly (we need to touch each observation once). However, parallel reductions can be implemented efficiently with only **log n** steps using a tree-reduction. Conceptually, you can think of a tree-reduction like this:



though practically speaking, we implement it more like so:



The work complexity is still the same (n-1 addition operations), but the span complexity is only logarithmic. For ***large n,*** this benefit is enormous. One thing to note about the practical implementation of this tree reduction is that it ***requires two kernel launches. The first performs block-wise reductions, yielding the sum of all values for each block. The second then launches one more thread block to reduce those block-wise sums into a single, overall sum (this assumes we have enough threads and stuff).***

Now, the tricky thing in our case is that we can't just average over all our data. Instead, for each cluster, we have to only average over the points assigned to that cluster. There's a few ways we could solve this problem. One idea would be to sort the data by their assignment, so that points in the same cluster are next to each other in memory, then do one standard reduction per segment.

The approach I picked is a bit different. Essentially, I wanted to do more work in the same kernel. So my idea was the following: keep a shared memory segment in each thread block and for each cluster and each thread, check if the thread is assigned to the cluster and write the thread's value into the shared memory segment if yes, otherwise write a zero in that place. Then do a simple reduction. Since we also need the count of assigned points per cluster, we can also map values to zeros and ones and reduce over those in the same sweep to get the cluster counts. This approach has both very high shared memory utilization and overall occupancy (we're doing lots of work in each block and in many blocks). Here is the code for the "fine", per-block reduction (it's quite a lot):

```
_global__ void fine_reduce(const float* __restrict__ data_x,
                            const float* __restrict__ data_y,
                            int data_size,
                            const float* __restrict__ means_x,
                            const float* __restrict__ means_y,
                            float* __restrict__ new_sums_x,
                                  float* __restrict__ new_sums_y,
                                  int k,
                                  int* __restrict__ counts) {
  // Essentially three dimensional: n * x, n * y, n * counts.
  extern __shared__ float shared_data[];

  const int local_index = threadIdx.x;
  const int global_index = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_index >= data_size) return;

  // Load the mean values into shared memory.
  if (local_index < k) {
    shared_data[local_index] = means_x[local_index];
    shared_data[k + local_index] = means_y[local_index];
  }

  __syncthreads();

  // Assignment step.

  // Load once here.
  const float x_value = data_x[global_index];
  const float y_value = data_y[global_index];

  float best_distance = FLT_MAX;
  int best_cluster = -1;
  for (int cluster = 0; cluster < k; ++cluster) {
    const float distance = squared_l2_distance(x_value,
                                               y_value,
                                               shared_data[cluster],
                                               shared_data[k + cluster]);
    if (distance < best_distance) {
      best_distance = distance;
      best_cluster = cluster;
    }
  }

  __syncthreads();

  // Reduction step.

  const int x = local_index;
  const int y = local_index + blockDim.x;
  const int count = local_index + blockDim.x + blockDim.x;
  for (int cluster = 0; cluster < k; ++cluster) {
the  // Zeros if this point (thread) is not assigned to the cluster, else
    // values of the point.
    shared_data[x] = (best_cluster == cluster) ? x_value : 0;
    shared_data[y] = (best_cluster == cluster) ? y_value : 0;
    shared_data[count] = (best_cluster == cluster) ? 1 : 0;
    __syncthreads();

    // Tree-reduction for this cluster.
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
      if (local_index < stride) {
        shared_data[x] += shared_data[x + stride];
        shared_data[y] += shared_data[y + stride];
        shared_data[count] += shared_data[count + stride];
      }
      __syncthreads();
    }

    // Now shared_data[0] holds the sum for x.

    if (local_index == 0) {
      const int cluster_index = blockIdx.x * k + cluster;
      new_sums_x[cluster_index] = shared_data[x];
      new_sums_y[cluster_index] = shared_data[y];
      counts[cluster_index] = shared_data[count];
    }
    __syncthreads();
  }
}
```

Note that we perform the assignment and block-wise reduction in the same kernel. Then, we do a coarse reduction to sum the block-wise values into a final, single value:

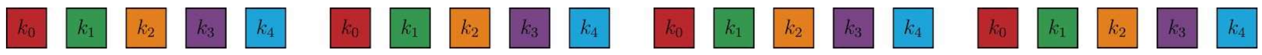
```
__global__ void coarse_reduce(float* __restrict__ means_x,
                              float* __restrict__ means_y,
                              float* __restrict__ new_sum_x,
                              float* __restrict__ new_sum_y,
                              int k,
                              int* __restrict__ counts) {
  extern __shared__ float shared_data[];

  const int index = threadIdx.x;
  const int y_offset = blockDim.x;
  // Load into shared memory for more efficient reduction.
  shared_data[index] = new_sum_x[index];
  shared_data[y_offset + index] = new_sum_y[index];
  __syncthreads();

  for (int stride = blockDim.x / 2; stride >= k; stride /= 2) {
    if (index < stride) {
      shared_data[index] += shared_data[index + stride];
stride]; shared_data[y_offset + index] += shared_data[y_offset + index +
    }
    __syncthreads();
  }

  // The first k threads can recompute their clusters' means now.
  if (index < k) {
    const int count = max(1, counts[index]);
    means_x[index] = new_sum_x[index] / count;
    means_y[index] = new_sum_y[index] / count;
    new_sum_y[index] = 0;
    new_sum_x[index] = 0;
    counts[index] = 0;
  }
}
```

Recall that in the fine reduction step, each block produces one sum per cluster. As such, You can visualize the input to the coarse reduction like this:


k_0 k_1 k_2 k_3 k_4 k_0 k_1 k_2 k_3 k_4 k_0 k_1 k_2 k_3 k_4 k_0 k_1 k_2 k_3 k_4

The goal is to combine all these values into k sums, by summing up the individual, block-wise cluster sums (i.e. have one sum for each ki). To do so, we simply stop the reduction at stride k, as you can see in the code above.

The last step is to launch the kernels of course:

```
// * 3 for x, y and counts.

const int fine_shared_memory = 3 * threads * sizeof(float);

// * 2 for x and y. Will have k * blocks threads for the coarse reduction.
```

```
const int coarse_shared_memory = 2 * k * blocks * sizeof(float);
// ...
for (size_t iteration = 0; iteration < number_of_iterations; ++iteration)
{
  fine_reduce<<<blocks, threads, fine_shared_memory>>>(d_data.x,
                                                       d_data.y,
                                                       d_data.size,
                                                       d_means.x,
                                                       d_means.y,
                                                       d_sums.x,
                                                       d_sums.y,
                                                       k,
                                                       d_counts);
  cudaDeviceSynchronize();

  coarse_reduce<<<1, k * blocks, coarse_shared_memory>>>(d_means.x,
                                                         d_means.y,
                                                         d_sums.x,
                                                         d_sums.y,
                                                         k,
                                                         d_counts);
```

```
    cudaDeviceSynchronize();
}
```

With all this effort and algorithmic stuff, we'd hope to have some gains from this, right? (Or if not, we'd call it a commendable scientific effort). Let's see:

IMPLEMENTATION	**N = 100**	**N = 100, 000**
C++	**0.00054s**	**0.26804s**
CUDA	0.00956s	**0.0752s**
CUDA 2	0.00878s	**0.0611s**
CUDA 3	0.00822s	**0.0171s**

Now, that's fast! Our CUDA implementation is nearly 16 times faster than plain c++.

Conclusion

So is this the best we can do? Definitely not. I imagine someone with more GPU programming experience could get even more out of it. I'm certain that the right intrinsics and loop unrolling tricks sprinkled around could speed up the implementation even more.

In conclusion, I have to say that GPU programming with CUDA is a lot of fun. I've realized that dealing with such highly parallel code and hundreds of thousands of threads requires a whole new set of techniques. Even simple reductions like the

ones we used here have to be approached completely differently than on serial CPUs. But getting speedups like the ones we saw is definitely amazing. I feel like I've gained a much greater appreciation for the algorithms that make my neural networks run 10x faster on a single GPU than even on a 32-core CPU. But there's so much more to explore in this space. Apparently convolutions are supposed to be pretty fast on GPUs?.