**Good afternoon everyone**

My name is Vu Dinh Trung. I’m from Vietnam, you can call me Wu Ting Zhong.

**[1] First of all, I would like to extend my thanks to you all for attending my defense today**. Now, I’m going to talk about my thesis: “Speeding up  
Generalized Fuzzy *k*-Means Clustering Using *m* Nearest Cluster Centers Algorithm on GPU.”

**[2]** **As you can see in this outline slice, I’ll first introduce GFKM and parallel GFKM algorithms**. The PGFKM model is mentioned in part 2. The steps of method PGFKM are presented in part 3, 4, and 5. We’ll see some results in part 6. Finally, there are some conclusions and future works.

**[3] Now, I’ll talk about GFKM algorithm and proposed method.**

GFKM algorithm is developed from the fuzzy *k*-means clustering (known as FKM) algorithm, uses a data point’s *M* nearest cluster centers to partition the data set. GFKM has the less computing time and the better clustering quality than method FKM.

The optimal value of *M*, which used to obtain the better clustering result for the method GFKM, is 2.

However, the running time of the GFKM algorithm as well the FKM algorithm grows with the increase of the size and also the dimensionality of the data set. So, we’ll focus on size of data set to implement the parallel GFKM on a GPU.

The proposed method PGFKM is about 3 to 18 times faster than the optimized CPU code version.

**[5] Let’s see PGFKM model.**

The centroids is initialized on CPU, then data points, centroids are copied to device memory for updating membership on GPU.

With small data set, new centroids will be calculated on CPU. With larger data set, new centroids will be calculated on GPU, then the left small data blocks are executed on CPU.

New centroids will be copied back to GPU for checking convergence step. If it is not convergence, then it will loop again from updating membership step. Otherwise, results will be copied back to CPU and finish.

**Now, let’s have closer look at the updating membership step.**

**[7]** In here, there have two levels of loop, one for data points, and other for centroids.

**[8]** On GPU, there is only one level of loop when *N* data points has been dispatched to *N* threads in parallel. Each thread loads a data point into on-chip registers.

If each thread writes less than five values from centroids to shared memory, then threads can read same centroids in shared memory with low latency.

Otherwise, they read same centroids in global memory with high latency.

**[9, 10]** More details are shown in next two slices.

**We’ll move to the calculating new centroids step.**

**[12] When number of data points *N* is small**, number of active threads in parallel is also small, so GPU-based method is inefficient. In other words, the parallel computational power of GPU cannot be fully exploited. In this case, CPU-based method is still used.

**[14, 15, 16] When number of data points *N* is large and the number of cluster representatives *k* is close to *M***, we design algorithms using equation of FKM with the computational complexity O(*Nkd* + *kd*). However, after the previous steps, the data are not coalesced (not adjacent). To achieve coalesced data access, data points are transposed from *N*×*d* to *d*×*N*, and memberships are transposed from *N*×*k* to *k*×*N*, using cuBLAS library, as shown in next two slices.

**[17, 18]** After transposing data points and memberships, we reduce **Ui,jq** for the *j*th cluster as well **Ui,jq \* Xi** for the *j*th cluster and the *x*th dimension on GPU. This may be executed concurrently or interleaved in six different streams in our experimental condition. And then small output block sums are reduced very fast on CPUs.

**[19] When number of data points *N* is large and the number of cluster representatives *k* is not close to *M***, we design algorithms using equation of GFKM with the computational complexity O(*NMd* + *kd*). To reduce un-coalesced data point accesses, the data points are transposed from *N*×*d* to *d*×*N*. To reduce un-coalesced membership accesses, array *NNT* is sorted with keys are cluster indices, and values are point indices and memberships.

**[20] The sorting array *NNT* has two scenarios**: one is using GPU-based counting sort algorithm, and other is using stable sort by keys of Thrust library in CUDA toolkit. For the first scenarios, histogram of *NNT* is calculated first.

**[21]** However, a lots of threads with the same cluster index increase the histogram array conflict with each other.

**[23] Counting sort using atomic functions** is not ensure the stability of point indices in each cluster. This causes the crossover accesses of threads to data points in the global memory and reduces the performance.

**[24] The stable sort by keys of Thrust library** helps avoid cross accesses in reduction functions later.

**Last step is the checking convergence.**

**[27]** **Checking convergence occupies a very small quota**, so it can be designed on both CPU and GPU. On GPU, each thread executes one dimensional of centroid, and each thread block executes for one centroid.

**And now, we’ll see some experimental results.**

**[29] The parallel GFKM algorithm is implemented using CUDA version 7.0**. The experiments are conducted on a PC with an NVIDIA GeForce GTX 760 GPU and an Intel(R) Core(TM) i5-4690 CPU. GTX 760 has six SIMD multi-processors, and each one contains 192 processors and performs at 1.5 GHz. The memory of the GPU is 2GB with the peak bandwidth of 192.2 GB/s. The CPU has four cores running at 3.50 GHz. The main memory is 8 GB with the peak bandwidth of 25.6 GB/s. To show the speedup effect more clearly, the time of the application is measured after the file I/O and the speedup are compared to the optimized CPU codes.

**[31]** **The data set generated from three real images: “Lena,” “Baboon,” and “Peppers,” abbreviated as “LBP” data set**. In this example, the data set consists of *N* = 49,152 data points with *d* = 16. The values *M* = 2, *k* = 8, ε = 10-8, and *max\_iter* = 200 are used for the example. The speedup of four GPU-based GFKM versions is shown in this table. The updating memberships step on GPU is about twelve times faster than on CPU. The calculating centroids step on GPU is about 1.5 times faster than on CPU. Checking convergence does not affect the overall performance. In this example, the running on GPU is about six to seven times faster than on CPU.

**[32] In this example, we perform testing speedup of our proposed methods with various number of data points *N* on the “poker” data set**. The “poker” data set consists of *N* = 1,025,010 instances, with *d* = 10. The values *M* = 2, *k* = 10, ε = 10-8, and *max\_iter* = 200 are used for this example. From this figure, we can see that the speedups of our proposed methods increase when the number of data points increases.

In here, version 1 is better where value *N* smaller than 32000 data points. When value *N* is about from 32000 to one million data points, version 3 becomes more efficient than others. While value k is larger than one million, versions 4 increases stable and becomes the most efficient. The maximum speedup is about twelve times faster in this example.

**[33] On the “poker” data set also, a speedup test is performed for our proposed methods with various number of cluster representatives *k***. The values *N* = 1025010, *M* = 2, ε = 10-8, and *max\_iter* = 200 are used for this example. From this chart, as value *k* is much greater than value M, the speedup of version 2 is dramatically reduced. When value *k* is smaller than 5, version 2 becomes more efficient than others, while value k is larger than 64, versions 1 remains stable and becomes the most efficient. With values k from 5 to 8, version 4 is the best case scenario. Version 3 is better to use where k is from 8 to 64.\

**[34] An evaluation of influence of the data dimension *d* to the speedup of our proposed methods is performed.** The “Synthetic” data set with *N* = 491520, *M* = 2, *k* = 8, ε = 10-8, and *max\_iter* = 200 are used for this example. From this chart, we can see that the speedups of our proposed methods are rapidly decreasing as number of dimension *d* is increasing and *d* is smaller than 16. Speedups are more slowly decreasing as *d* becomes larger and larger. In this example, the lowest speedup archived about 4 to 5.5 times faster, in which, speedup of the version 4 is better than the others’.

**[36]** **In general, the better speedups are archived with the larger number of data points, smaller number of dimensions d and cluster representative *k***. With high-dimensional, the data points cannot be loaded into registers and reduce performance. In future, we should calculate the temporary distance and store in the on-chip registers by loading the data to shared memory, tile by tile and adopting idea of parallel matrix multiplication.

Thank you for your listening to my presentation.

**Single instruction, multiple thread (SIMT)** is a parallel execution model, used in some GPGPU platforms, where multithreading is simulated by SIMD processors.

The G80 NVIDIA GPU architecture introduced the single-instruction multiple-thread (SIMT) execution model where multiple independent threads execute concurrently using a single instruction.

**Single instruction, multiple data (SIMD)** is a class of parallel computers. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.