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 method PGFKM’s steps 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.

**[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.

**[29]**