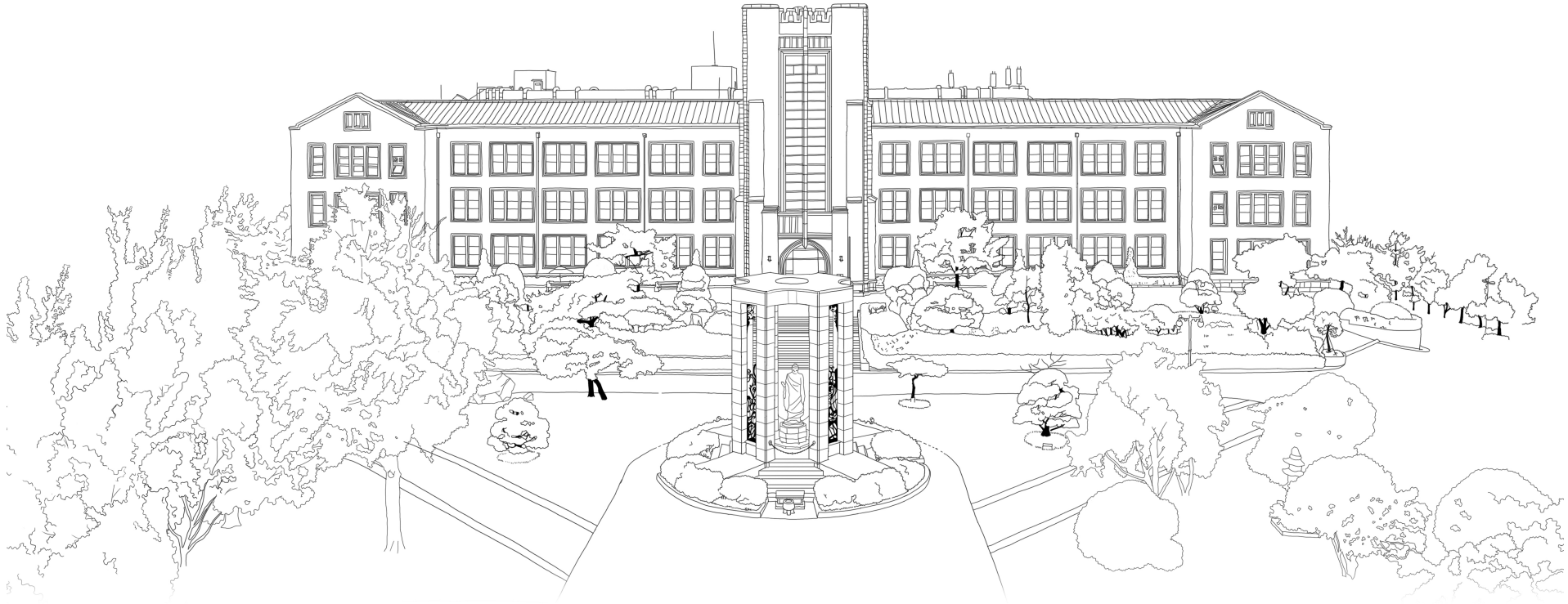


Uniformly Fine-Grained Data-Parallel Computing for Machine Learning Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

BigDataLab
Dept. of Multimedia Engineering at Dongguk University

dongguk UNIVERSITY

Contents

Introduction

❖ Overview

- **The development of Graphic Processing Unit (GPU)**
  - ➢ The GPU of modern computers has evolved into a powerful, general-purpose, massively parallel numerical processor
    - ✓ The numerical computation in a number of machine learning algorithms fits well on the GPU
    - ✓ To identify such algorithms, they present $uniformly\ fine-grained\ data-parallel\ computing$ & illustrates GPU and CPU mixed computing architecture
    - ✓ Example) k-means clustering, k-means regression clustering
  - ➢ We will discuss key issues
    - ✓ Design of the algorithms
    - ✓ Computation partitioning between a CPU and a GPU
  - ➢ Performance gains on a CPU and GPU mixed architecture are compared with the architecture implemented completely on a CPU

❖ Development CPU

- The development of CPU
  - ➢ The computing power of the CPU has increased a lot in the past few decades
  - ➢ Physical limit of the CPU change the industry to go in the direction of packing multiple CPU cores onto the same silicon die
  - ➢ Parallel computing emerged from an optional computational architecture into a mainstream choice for general computing
  - ➢ As a result of this fundamental change in the computing architecture, algorithm design, data structure design and programming strategies have to be changed

Introduction

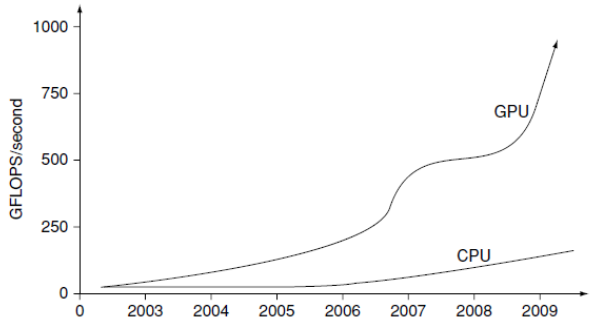
❖ GP-GPU

- **Original GPU**
  - ➢ GPUs employed multicore parallel computing long before CPUs did
    - ✓ have special architecture because of the nature of graphics computations
  - ➢ some efforts were made to use GPU's special architecture for more general-purpose numerical computing
    - ✓ Written code for the graphics interface was very unintuitive
    - ✓ Difficult to program general-purpose numerical algorithms on a GPU because of its specialized architecture

- **General-Purpose GPU (GP-GPU)**
  - ➢ GPU has developed rapidly from a special GPU into a GP-GPU
    - ✓ GP-GPU has made the mapping of numerical computing task onto the new GPU architecture much easier
    - ✓ NVIDIA's CUDA which is an augmented C interface is also available now for programming GPUs for generally-skilled programmers

❖ GP-GPU (cont'd)

- Advantages of GP-GPU
  - ➢ numerical processing using a GP-GPU is better than using CPU



The performance increase of a GPU and a CPU

- ✓ Saving hardware cost
- ✓ Saving power consumption per GFLOPS (Giga FLOating-point operation Per Second)
- ✓ GPU use 300W of power and can solve numerical problems about 40 more times faster than CPU (at 50W)

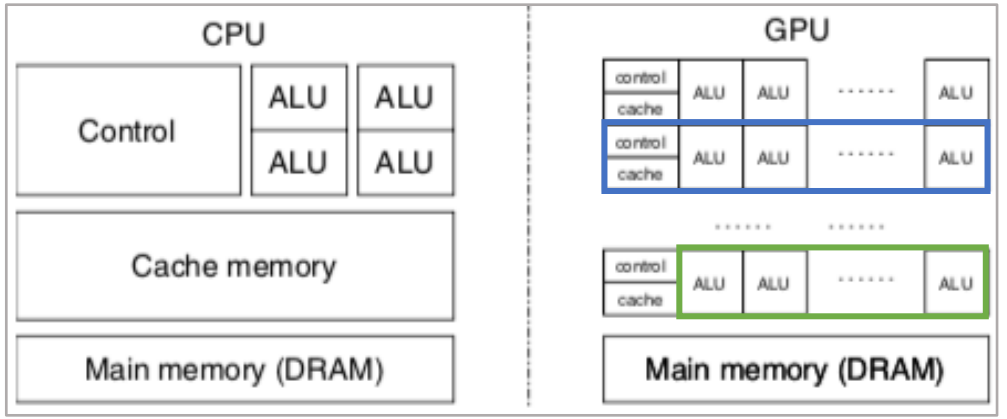
➢ This chapter presents a general design principle for GPU based computing that they call $uniformly\ fine-grained\ data-parallel\ computing$

Overview of a GP-GPU

❖ The structure of GP-GPU

- Multiple multiprocessor
  - ➢ A GPU consists of a two-level hierarchical architecture
    - ✓ Multiprocessor has multiple lightweight data-parallel cores

    - ✓ multiprocessor sharing the same program logic
  - ➢ Shared memory (Cache)
    - ✓ Shared by the cores within the same multiprocessor
    - ✓ Low access-latency
  - ➢ Global memory (Main memory)
    - ✓ Shared by the cores in different multiprocessor
    - ✓ Much longer access latency than the cache



The structure of CPU & GPU

Overview of a GP-GPU

❖ The structure of GP-GPU (cont'd)

- **Grid, Block and Thread**
  - ➢ Thread
    - ✓ The thread is the execution unit of the process
    - ✓ Computing threads on a GPU are organized into thread blocks
    - ✓ Each thread up to three dimensional indices (Thread ID)
  - ➢ Block
    - ✓ Many thread blocks are organized into a thread grid
    - ✓ Each block has indices (Block ID)
  - ➢ Grid
    - ✓ Unit of launch for execution
    - ✓ Enough thread blocks are necessary to keep all the multiprocessor busy

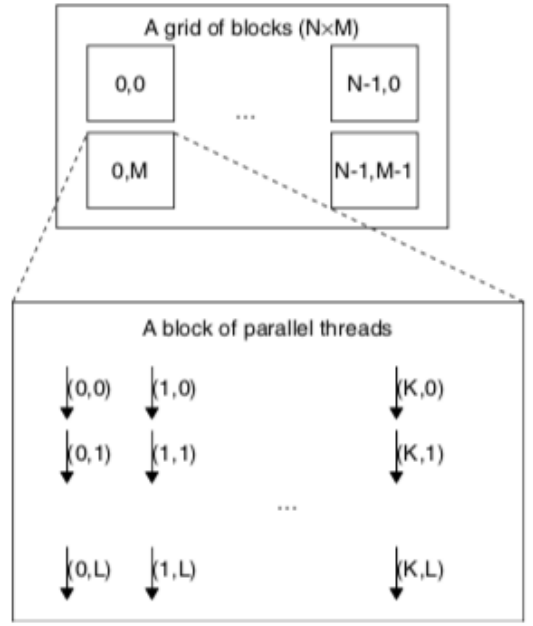


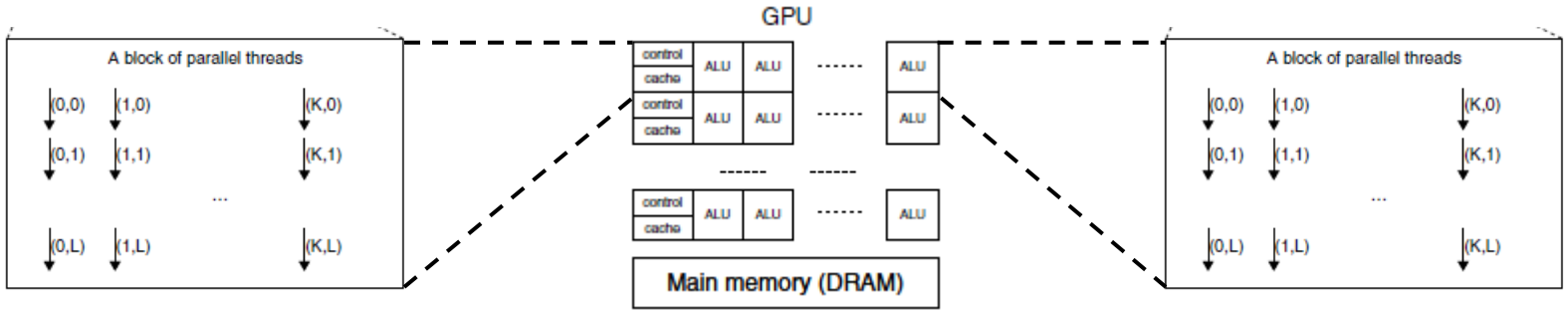
Figure 5.3 Computing threads are organized into blocks.

- Threads are linked to data for data-parallel by mapping the block & thread ID to data indices

Overview of a GP-GPU

❖ The structure of GP-GPU (cont'd)

- **Thread Block & GPU**
  - ➢ The threads within the same block are always executed on the same multiprocessor throughout their lifetime
  - ➢ Multiple blocks of threads can be loaded to the same multiprocessor and executed in a multiplexed fashion to hide the memory access latencies



The first thread block | Multiple multiprocessors | The second thread block

  - ➢ Synchronization

```
If A is true then
   Do task 1;
Else
   Do task 2;
```

✓ Some processor execute task 1 → the remaining processors are left in idle (synchronization)

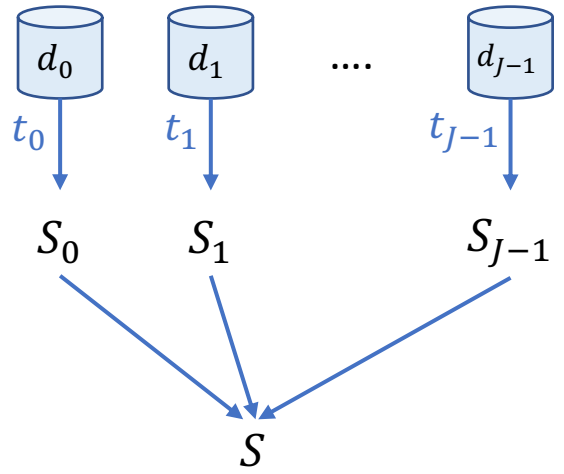
✓ Carefully designing the code to minimize the number of steps that have to be included into a conditional execution block

❖ Data-Parallel Computing

- **Concept of data-parallel computing**
  - ➢ Data parallelism is a very common parallel computing paradigm
  - ➢ data-parallelism
    - ✓ $X = \{d_j\}_{j=0}^{J-1}$: partition of a dataset
    - ✓ $Proc$: computation procedure executed by each thread
    - ✓ $T = \{t_j\}_{j=0}^{J-1}$: computing threads
    - ✓ $thread_j = proc(d_j)$: jth computing thread executing the procedure $proc$ on d_j
    - ✓ All threads can go in parallel without blocking because they don't share anything (data parallelism)
  - ➢ SNWV (Share No Write Variables)
    - ✓ Different threads in the data-parallel computing structure share no variables that have write permission
    - ✓ $thread_j = proc(shared\_read\_only\_parameters,\ d_j)$ → NVIDIA's GPU
    - ✓ Also be embedded in the code as "constant" that all processors can read concurrently

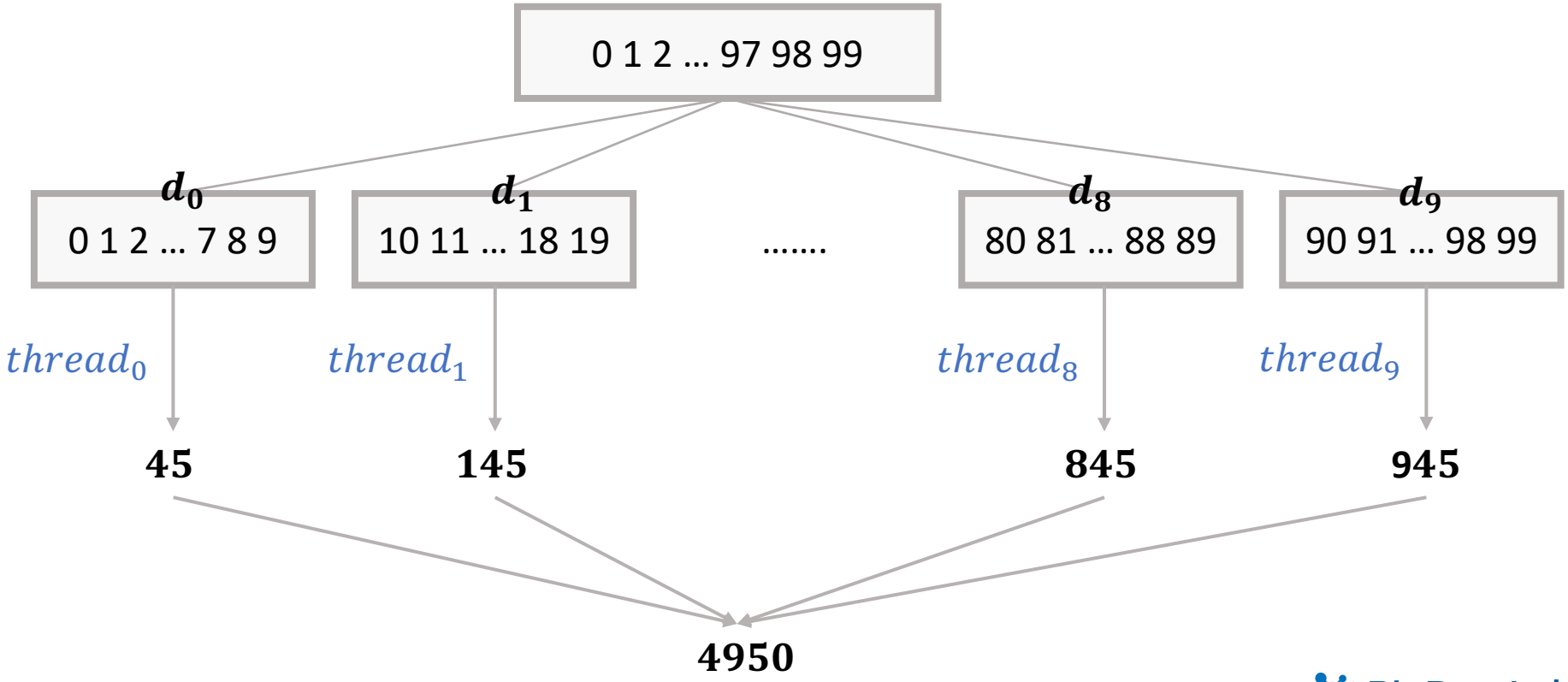
❖ Data-Parallel Computing (cont'd)

- **Example of data-parallel computing**
  - ➢ $S = \sum_{i=0}^{N-1} x_i$: summation over N numbers
  - ➢ Divide the problem into two steps
  - ➢ First step (data parallel)
    - ✓ Divide the dataset into J segments → Calculate the summation of each segment
    - ✓ $d_j = \{x_i\}_{jM}^{(j+1)M-1}$, where $M = N/J$
    - ✓ $S_j = \sum_{i=jM}^{(j+1)M-1} x_i$
    - ✓ Summation over each partition S_j is done by a separate computing thread
    - ✓ Different threads share no written variables, satisfying SNWV
  - ➢ Second step
    - ✓ The total summation S is calculated by the outputs from all threads in step 1
    - ✓ $S = \sum_{j=0}^{J-1} S_j$



❖ Data-Parallel Computing (cont'd)

- Example of data-parallel computing
  - ➢ The number of thread(J) : 10
  - ➢ The number of data(N) : 100



❖ Uniformly Fine-Grained Data-Parallel Design

- **Concept of Uniformly Fine-Grained Data-Parallel Design**
  - ➢ When all processors execute the same instruction, the highest utilization of the processors is achieved
  - ➢ Characterize the programs well suited for GPUs as uniformly fine-grained data-parallel algorithms
  - ➢ Uniformity
    - ✓ uniformity of instructions executed on the same multiprocessor
    - ✓ uniformity of data partitioning over the array of processor
  - ➢ Fine granularity
    - ✓ parallelization is implemented at the lowest level
    - ✓ Instructions are executed on the multiprocessors at the same time while individual data access requests are made to the memory
    - ✓ This will hide the memory access latency better
  - ➢ Having long, separated data access phases and computation phases is not a good design

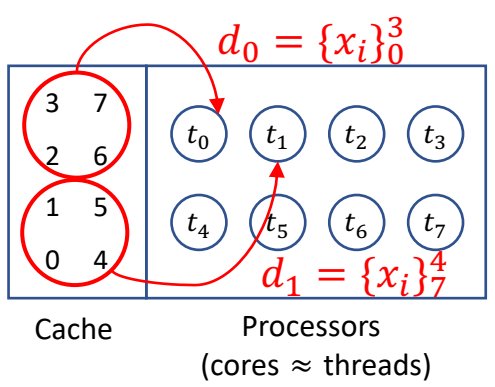
Fine-grain : Pseudocode for 100 processors	Coarse-grain : Pseudocode for 2 processors

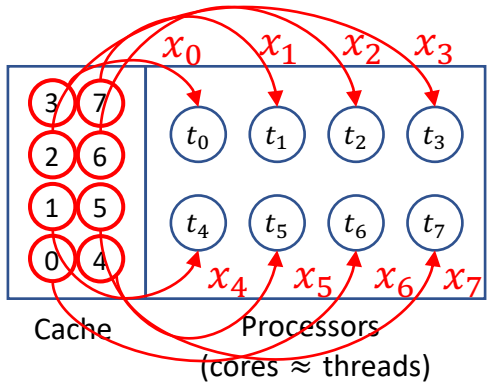
`void main()`
`{`
`  switch (Processor_ID)`
`  {`
`  case 1: Compute element 1; break;`
`  case 2: Compute element 2; break;`
`  case 3: Compute element 3; break;`
`  .`
`  .`
`  .`
`  case 100: Compute element 100;`
`  break;`
`  }`
`}`
	`void main()`
	`{`
	`  switch (Processor_ID)`
	`  {`
	`  case 1: Compute elements 1-50;`
	`  break;`
	`  case 2: Compute elements 51-100;`
	`  break;`
	`  }`
	`}`

Computation time - 1 clock cycle	Computation time - 50 clock cycles

❖ Uniformly Fine-Grained Data-Parallel Design (cont'd)

- **Example of uniformly fine-grained data-parallel computing**
  - ➢ A multiprocessor can read from up to 128 contiguous bytes from global memory in one memory access cycle
  - ➢ problem : $S = \sum_{i=0}^{N-1} x_i$: summation over N numbers
  - ➢ $d_j = \{x_j\}$, not $d_j = \{x_i\}_{jM}^{(j+1)M-1}$
    - ✓ This satisfy the uniformly fine-grained requirement
  - ➢ $proc_m(x_j)$: $S_m \mathrel{+}= x_j$





At time t_1: $proc_1(x_1), proc_1(x_1), \dots, proc_7(x_7)$
At time $t_{1+\delta}$: $proc_1(x_{M+1}), proc_1(x_{M+2}), \dots, proc_7(x_{2M})$

All processors are always busy & low memory access delay

The k-Means Clustering Algorithm

❖ Concept of k-Means Clustering Algorithm

- **Pseudocode**
  - ➢ $X = \{x_i | i = 1, \ldots, N\}$: **dataset**
  - ➢ k : the number of cluster, $\mathrm{M} = \{m_j | j = 1, \ldots, k\}$: k cluster centroids
  - ➢ $S_j \subset X$: the subset of x that are closer to m_j

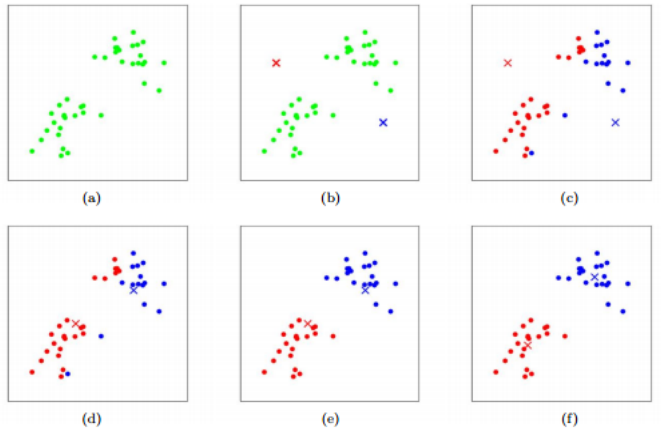
The *k*-means Algorithm

Step 1: Initialize all centroids (randomly or based on some heuristic)

Step 2: Associate each data point with the nearest centroid. This step partitions the dataset into k disjoint subsets (*Voronoi Partition*)

Step 3: Calculate the centroids to maximize the objective function $\sum_{j=1}^{k} \sum_{x \in S_j} ||x - m_j||^2$, which is the total squared distance from each data point to the nearest centroid

Repeat steps 2 and 3 until there are no more changes in the membership of the data points (proven to converge in a finite number of steps)



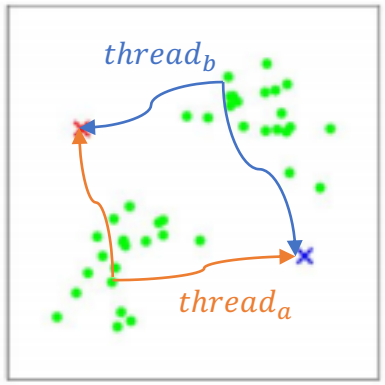
  - ➢ This algorithm guarantees convergence to only a local optimum of the objective function

$$Obj_{KM}(X, M) = \sum_{j=1}^{k} \sum_{x \in S_j} ||x - m_j||^2,$$

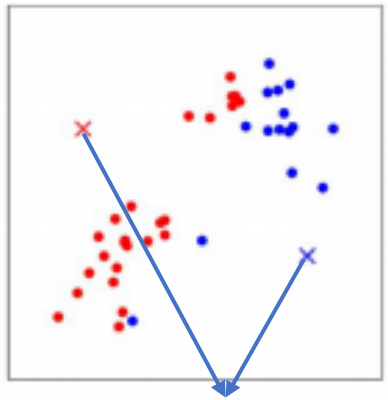
The k-Means Clustering Algorithm

❖ Data Parallelism in k-Means

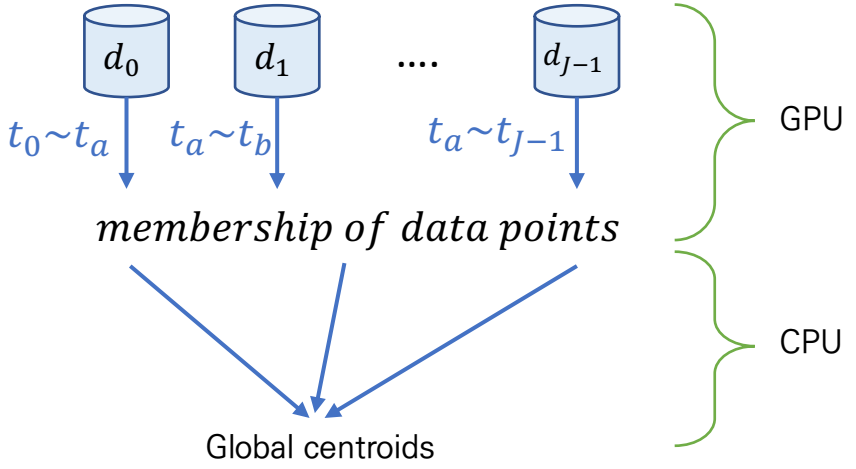
- **Data-Parallel design**
  - ➢ A dataset is partitioned into blocks with the number of data points matching the thread block size
  - ➢ The step 2 of k-Means data parallel and can be performed in an SNWV.
    - ✓ Each thread evaluates the distances from a data point to all the centroids (Parallel)
    - ✓ Each thread assigns the membership of the data points (Parallel)
    - ✓ The centroids are loaded to all the multiprocessors as read-only variables and shared (SNWV)
  - ➢ Membership vector calculated on the GPU is transferred to the CPU to update the centroids



Each thread calculates the distance from a data to all the centroids



Centroids are shared by all the threads (Read-Only variables)

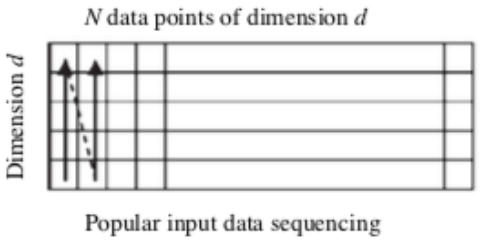


The k-Means Clustering Algorithm

❖ Fine-Grained Data-Parallelism in k-Means

- **Fine-Grained Data-Parallel design**
  - ➢ Fine grained data-parallel computation is achieved by arranging the data values of all threads in a thread block to work in a synchronized fashion
    - ✓ The column of input datasets is naturally one datapoint
    - ✓ The input data should be arranged
    - ✓ The data points that are accessed by all threads in block in the same clock cycle are consecutive in memory

Data Parallelism



3	7	7	3	3	7	7	3
2	6	6	2	2	6	6	2
1	5	5	1	1	5	5	1
0	4	4	0	0	4	4	0

d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7

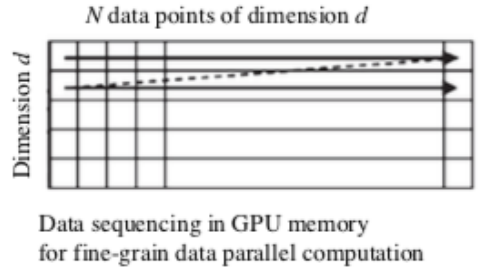
3	7
2	6
1	5
0	4

t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7

Cache

Processors (cores ≈ threads)

Uniformly Fine-Grained Data-Parallelism



3	7	7	3	3	7	7	3
2	6	6	2	2	6	6	2
1	5	5	1	1	5	5	1
0	4	4	0	0	4	4	0

d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7

3	7
7	3
3	7
7	3

t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7

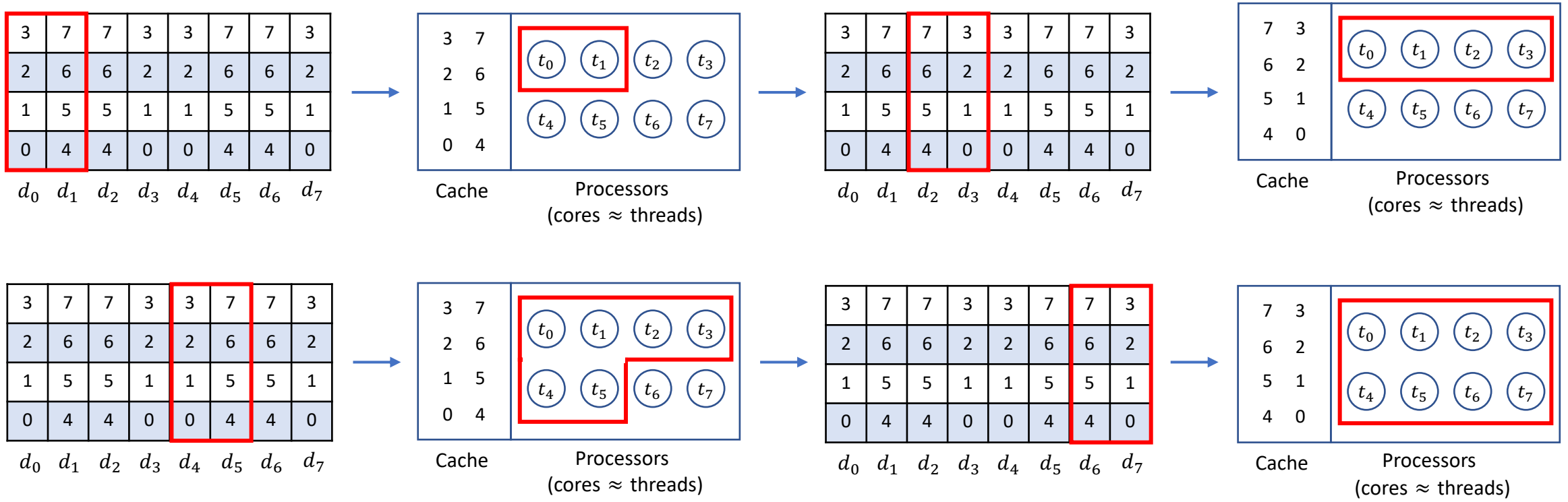
Cache

Processors (cores ≈ threads)

The k-Means Clustering Algorithm

❖ Fine-Grained Data-Parallelism in k-Means (cont'd)

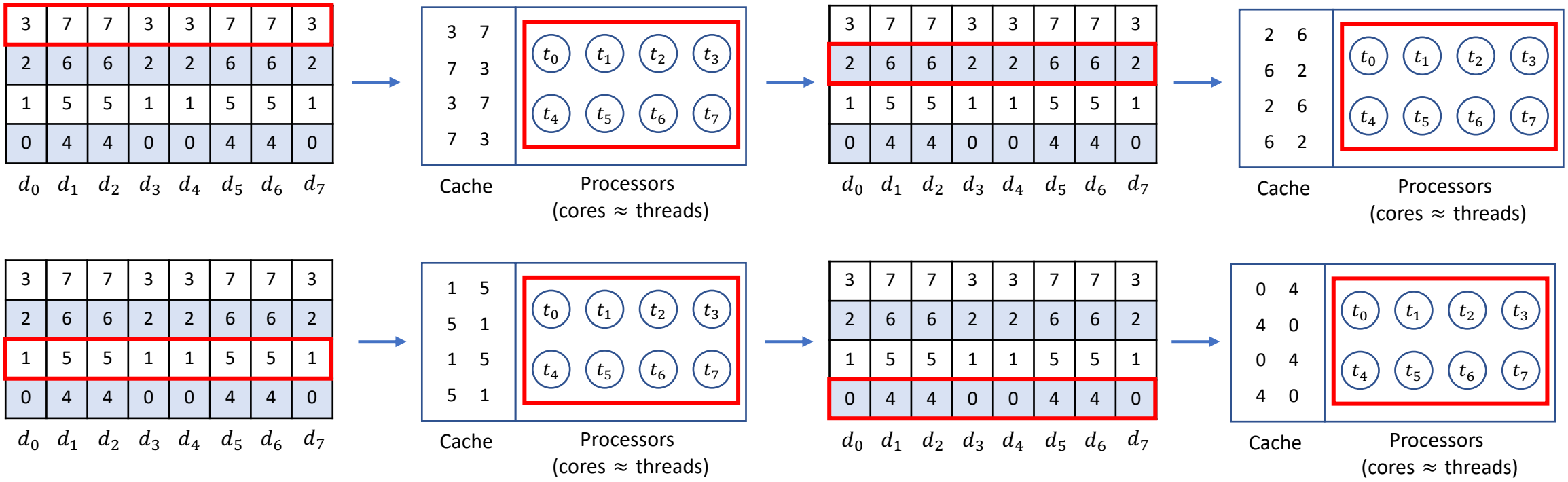
- **Fine-Grained Data-Parallel design**
  - ➢ Process of Coarse-Grained Data-Parallel
  - ➢ All running thread calculate the distance from the point(whole dimension) to centroids



The k-Means Clustering Algorithm

❖ Fine-Grained Data-Parallelism in k-Means (cont'd)

- **Fine-Grained Data-Parallel design**
  - ➢ Process of Fine-Grained Data-Parallel
  - ➢ All running thread calculate the distance from the point(one dimension) to centroids



The k-Means Clustering Algorithm

❖ Fine-Grained Data-Parallelism in k-Means (cont'd)

- **Fine-Grained vs Coarse-Grained**
  - ➢ Coarse-Grained Data-Parallel
    - ✓ A lot of data is allocated to the thread at one time
    - ✓ Many processors are in idle
    - ✓ The latency of access to global memory is high
  - ➢ Fine-Grained Data-Parallel
    - ✓ A small amount of data is allocated to the thread at one time
    - ✓ Cache accesses global memory while all processors are working
    - ✓ Trade off the latency of access to global memory
    - ✓ The highest utilization of the processors is achieved

The k-Means Clustering Algorithm

❖ Fine-Grained Data-Parallelism in k-Means (cont'd)

- **Fine-Grained Data-Parallel Pseudocode**
  - ➢ Pseudocode for each thread to calculate the membership of the data points
  - ➢ It is called in each iteration of k-means

Algorithm 12: Assign Cluster Membership to Data Points

Input: Floating-point storage *data*[.][.] for the transposed data array
Input: Floating-point storage *centroids*[.][.] for the centroids
Input: Integer storage *membership*[.] for the calculated membership of data points
Input: Integer parameters *data_size*, *num_dims*, *num_clusters*

```
thread_id ← blockDim * blockIdx + threadIdx
total number of threads TN ← blockDim * gridDim
For i = thread_id to data_size − 1 with step size TN do
    min_distance ← FLT_MAX
    cidx ← 0
    For j = 0 to num_clusters − 1 do
        distance ← 0
        For dim = 0 to num_dims − 1 do
            distance ← distance+ (data[dim][i] − centroids[j][dim])
        If dist < min_distance then
            min_distance ← distance
            cidx ← j
    membership[i] ← cidx
Synchronize Threads
```

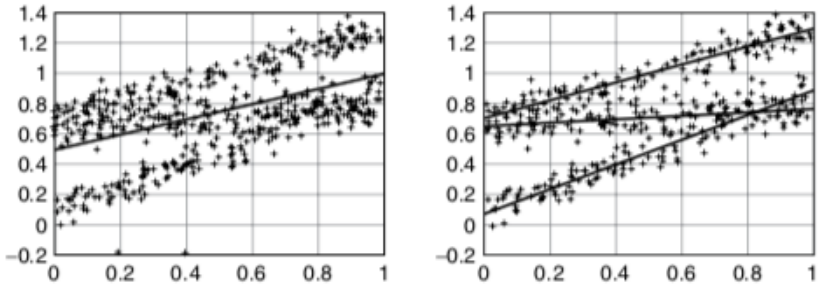
- ➢ Input
  - ✓ $data[dimension][index]$
  - ✓ $centroids[index][dimension]$
  - ✓ $membership[index]$
  - ✓ $Parameters: data\_size, num\_dim, num\_cluster$
- ➢ Output : the membership vector
- ➢ variables

gridDim – the total number of blocks in a grid
blockIdx – the index of a block inside its grid
blockDim – the total number of threads in a block
threadIdx – the index of a thread in its block
FLT_MAX – the maximum value of a floating point (hardware property)
i – data index
j – cluster index
dim – dimension index
cidx – the index of the closest centroid found so far

The k-Means Regression Clustering Algorithm

❖ Concept of the k-Means Regression Clustering Algorithm

- **Regression Clustering(RC)**
  - ➢ Regression clustering(RC) is a combination of clustering and regression



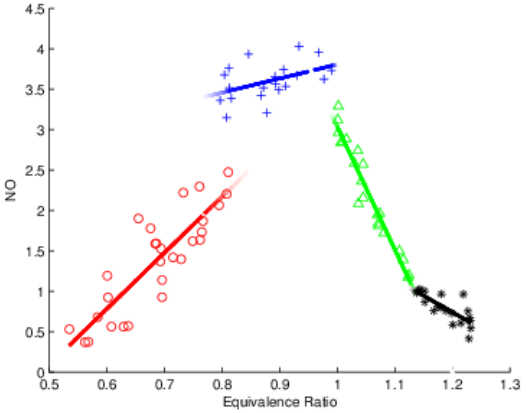
  - ✓ When the dataset contains a mixture of very different response characteristics, regression is not effective
  - ✓ RC allows multiple regression function, each fitting part of the data
  - ➢ RC is performed in two steps
    - ✓ Clustering : the data points are assigned clustering labels
    - ✓ Regression : a regression function is fit to the data in each cluster
    - ✓ Each regression function of cluster is “cluster centroid”
  - ➢ Compared with the centroid-based k-means, the centroids in k-mean RC are more complex data models

The k-Means Regression Clustering Algorithm

❖ Concept of the k-Means Regression Clustering Algorithm (cont'd)

- **The process of Regression Clustering(RC)**
  - ➢ k centroids → a set of regression functions $M = \{f_1, \ldots, f_k\} \subset \phi$
  - ➢ RC-k-means has the following steps

Step 1: Initialize the regression functions.
Step 2: Associate each data point (x, y) with the regression function that provides the best approximation $\arg\min_j \{e(f_j(x), y) | j = 1, \ldots, k\}$. This step partitions the dataset into k partitions.
Step 3: Recalculate the regression function on each data partition that maximizes the objective function (see Equation 5.2).
Step 4: Repeat steps 2 and 3 until no more data points change their membership.

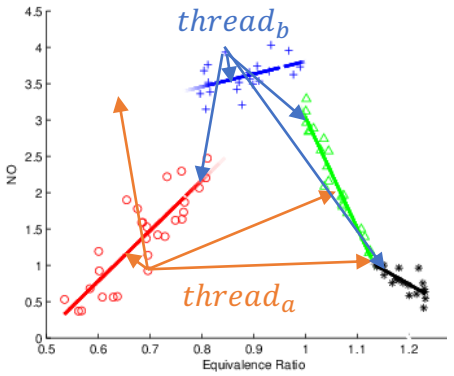


  - ➢ Step 2
    - ✓ $e(f(x), y) = \|f(x) - y\|^2$
    - ✓ Associate each data point with the nearest regression function and partition the dataset into k partitions
    - ✓ Same with the calculate membership vector step in centroid-based k-means
  - ➢ Step 3
    - ✓ Maximize objective function $f^{opt} = \arg\min_{f \in \Phi} \sum_{i=1}^{N} e(f(x_i), y_i)$

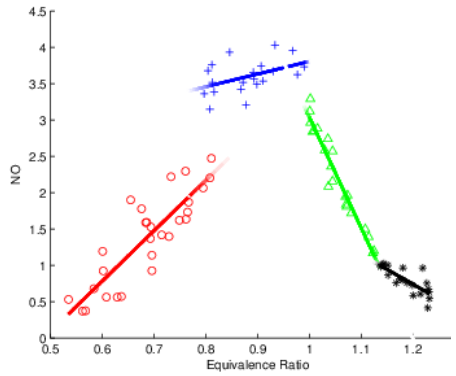
The k-Means Regression Clustering Algorithm

❖ Fine-Grained Data-Parallel Structures in k-Means RC Algorithm

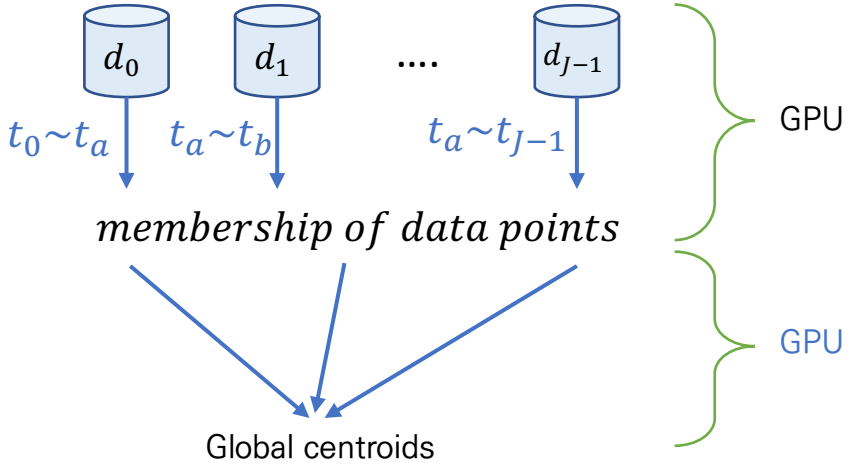
- k-means RC also has two phases in each iteration
- Membership calculation step
  - ➢ It is similar with membership calculation step in centroid-based k-means
  - ➢ Just two items are different
    - ✓ Calculate the distance from "point to point" to "point to function"
    - ✓ The centroids(regression function parameters) are loaded to all the multiprocessors as read-only variables and shared (SNWV)



Each thread calculates the distance from a data to all the regression functions



functions are shared by all the threads (Read-Only variables)



The k-Means Regression Clustering Algorithm

❖ Fine-Grained Data-Parallel Structures in k-Means RC Algorithm (cont'd)

- **Calculating the new center-functions step**
  - assume that we are doing a simple multivariate linear regression
  - Regression function : $y = c_j^T x$, c : coefficient of regression function in jth cluster
  - The data points in each cluster are used to update the center-function they are associated with by a MSE regression for the best fit to the data in the cluster
  - Calculate coefficient c_j
    - ✓ L_j : the number of data points assigned into the jth cluster
    - ✓ A_j : data vectors in the jth cluster, B_j: corresponding y-values
    - ✓ $A_j = \begin{bmatrix} x_1 \\ \cdots \\ x_{L_j} \end{bmatrix}$, $b_j = \begin{bmatrix} y_1 \\ \cdots \\ y_{L_j} \end{bmatrix}$
    - ✓ When $A_j c_j = b_j$, we can calculate c_j using normal equation $c_j = (A_j^T A_j)^{-1} A_j^T b_j$
    - ✓ If dimension of x is 5,

$$A_j = \begin{bmatrix} a & b & c & d & e \\ & & \cdots & & \\ f & g & h & i & j \end{bmatrix}, \quad b_j = \begin{bmatrix} a \\ \cdots \\ b \end{bmatrix}, \quad c_j = \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}$$

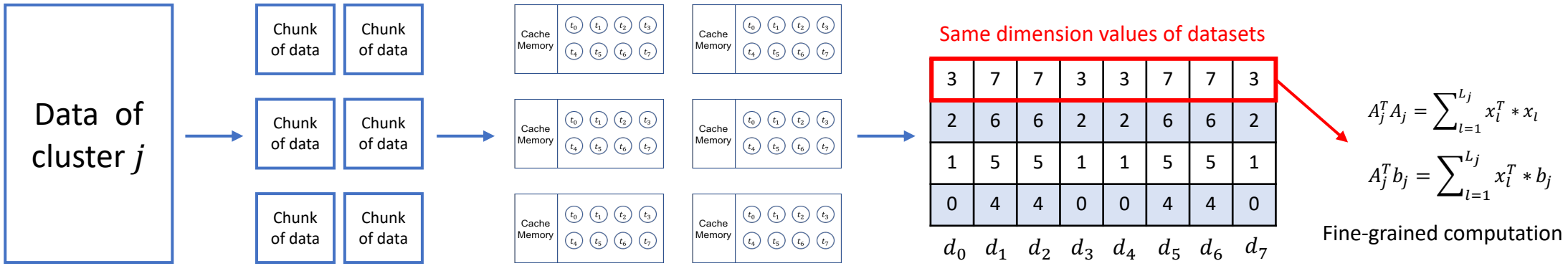
The k-Means Regression Clustering Algorithm

❖ Fine-Grained Data-Parallel Structures in k-Means RC Algorithm (cont'd)

- **Calculating the new center-functions step**
  - ➢ Calculate coefficient c_j in parallel
    - ✓ $c_j = (A_j^T A_j)^{-1} A_j^T b_j$
    - ✓ Rewrite the matrix multiplication and the vector multiplication as summations over the data points

$$A_j^T A_j = \sum_{l=1}^{L_j} x_l^T * x_l \qquad A_j^T b_j = \sum_{l=1}^{L_j} x_l^T * b_j$$

    - ✓ These show the data-parallel structure of the computing tasks and work can be naturally partitioned by breaking up the summation over 1 to L_j



Implementations and Performance Comparisons

❖ CPU-Only Implementation of k-Means

- MineBench : high-performance multi-threaded data-mining package
- Author's own highly optimized k-means package

Table 5.1. *Performance comparison between MineBench and our optimized CPU implementation. N – number of data points; D – dimensionality; k – number of clusters; M – number of iterations run.*

Dataset				MineBench time(s)			Optimized time(s)			Speedups (×)		
N	*D*	*k*	*M*	1c	4c	8c	1c	4c	8c	1c	4c	8c
2M	2	100	50	154	39	19	36	10	5	4.2	4.0	3.8
2M	2	400	50	563	142	71	118	30	16	4.8	4.7	4.6
2M	8	100	50	314	79	40	99	26	13	3.2	3.1	3.0
2M	8	400	50	1214	304	152	354	89	45	3.4	3.4	3.4
4M	2	100	50	308	78	39	73	20	10	4.2	4.0	3.7
4M	2	400	50	1128	283	142	236	60	30	4.8	4.7	4.7
4M	8	100	50	629	159	80	197	51	27	3.2	3.1	3.0
4M	8	400	50	2429	609	304	709	179	91	3.4	3.4	3.3

➢ Author's optimized CPU code for k-means runs several times faster than MineBench

Implementations and Performance Comparisons

❖ GPU-Accelerated k-Means & k-Means clustering Algorithm

- HPL CPU – author's optimized CPU-only implementations
- HPL GPU – author's GPU-accelerated version

Table 5.2. *Speedups compared with the CPU versions running on 1 CPU core (HPLC and HPLG are our own CPU and GPU implementations).*

Dataset				Time (s)			Speedups (×)	
N	*D*	*k*	*M*	MineBench	HPLC	HPLG	MineBench	HPLC
2M	2	100	50	154	36	1.45	106	25
2M	2	400	50	563	118	2.16	261	55
2M	8	100	50	314	99	2.48	127	40
2M	8	400	50	1214	354	4.53	268	78
4M	2	100	50	308	73	2.88	107	25
4M	2	400	50	1128	236	4.36	259	54
4M	8	100	50	629	197	4.95	127	40
4M	8	400	50	2429	709	9.03	269	79

Performance of K-means on GPU

Table 5.3. *Performance of k-means RC algorithm.*

Dataset				Time (s)		
N	*D*	*k*	*M*	CPU	GPU	Speedups (×)
262144	4	100	50	59.4	0.95	62.5
262144	4	200	50	112	1.19	94
262144	4	400	50	216	1.67	129
524288	4	100	50	120	1.80	66.5
524288	4	200	50	224	2.22	101
524288	4	400	50	433	3.00	144
1048576	4	100	50	243	3.50	69.5
1048576	4	200	50	456	4.23	108
1048576	4	400	50	874	5.66	154

Performance of K-means RC on GPU

- Achieved an average of 190x speedup over MineBench running on single core
- Achieved a 49x speedup over author's own optimized CPU implementations

Conclusions

❖ Conclusions about this chapter

- GP-GPU has advanced rapidly and has been applied to a variety of scientific computing algorithms
- Uniformly Fine-Grained Data-Parallel computing paradigm has a high-performance parallel implementations of computing tasks on a GP-GPU
- Not all algorithms fit well on a GPU, before programming an algorithm on a GPU, discovering uniformly fine-grained data parallelism in the algorithm is very important
- GPU provides significant speedups for intensive numerical computing at relatively low hardware cost and energy cost

Thank You!