

chapter 5 note

dongyeon



Uniformly Fine - Grained Data Parallel Computing for Machine Learning Algorithms

0. Introduction

* Paragraph 1. → 헤더처럼 보면

- Modern Computers의 GPU는 Powerful 텐서 general-purpose (성능적)이며 massively parallel numerical processor (다수의 계산 핵심 Processor)로 전문화되었다.
- 많은 machine learning algorithms의 경우 현재 GPU를 사용하기에 적합하다.
- 이러한 알고리즘을 살펴보기 위해, Uniformly fine - grained data Parallel Computing을 다룬다. 이를 GPU와 CPU 같은 computer architecture의 clustering과 regression clustering이라는 두가지 machine learning algorithm에 대해 살펴본다.
- 우리는 algorithm과 data structure의 험구적인 design과 CPU와 GPU사이 관계된 주제 문제에 대해 논의한다.
- CPU와 GPU와 같은 아키텍처의 특성 형상은 모두 CPU만을 이용한 regression clustering과 비슷하다.
- GPU, CPU 같은 architecture의 speed up 효과뿐만 아니라 better cost performance, energy performance 효과도 내용

고려한 multicore의 특성과 Parallel, algorithm

* Paragraph 2. → CPU의 발전과 함께

무엇?

변화의 필요성

- CPU의 Computing Power는 소형화와 증가하는 clock 주파수의 영향을 받아 수령률이 증가하였다.
- Hardware 가속 기술 컴퓨팅, 파이프라인은 CPU의 Computing Power를 증가시켰다.
- Frequency increases speeded up CPU even more directly
- miniaturization이 면적을 점령하고 부하를 즐겁게 하고, Power consumption의 nonlinearity 증가로 인해 increasing the frequency가 Endless 가속화의 양을 제한하는 원인이다. 이러한 소형화 과정의 물리적 한계가 마침내 떠오른 단점이다.
- 따라서 이러한 문제로 인해 CPU의 clock frequency를 늘리지 않고 multiple CPU cores를 same silicon die를 packing하는 경향을 만들었다.
- Parallel computing은 supercomputing을 위한 optional computational architecture이나 general computing을 위한 mainstream choice (주류 선택)으로 등장했다.

- Computing architecture 및 algorithm fundamental은 디자인의 경계
Algorithm design, 디자인 design, Programming strategies은 새로운 challenge
을 제시합니다.

* 이 chapter에서는 Uniformly fine-grained data Parallel Algorithms을 학습할 수 있는
방법론과 같은 디자인 원칙 general design principle을 살펴봅니다.

* Paragraph 3 → GPU의 등장과 발전, CPU와의 차이

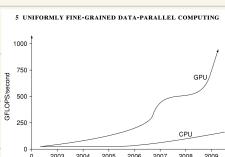
- GPU는 Graphic 처리의 특성 때문에 CPU가 multicore parallel computing을
하기에 어려워 다른 사용된다.
- GPU의 specialized architecture를 흡사한 numerical computing을 위한
노드를 갖으나 다른 mixed result를 드러내었으며 (Benchmark, graph 툴킷 등)
GPU의 graphic interface를 통해 쉽게 다른 scientist들이 unintuitive하게
GPU의 복잡한 numerical algorithm을 programming 할 때는 graphic processing
- GPU의 복잡한 numerical algorithm을 programming 할 때는 graphic processing
을 위한 specialized architecture 때문에 매우 어렵습니다.
 - 'numerical computation'은 graphics programming interface를 통해 programming
하거나反之 'graphics operation'을 통해 하는 경우
- GP-GPU: 최근 GPU는 graphic 기능뿐만 아니라 special purpose General
Purpose Uniform multicore computing devices로 활용되며 GP-GPU
라고 한다.

- (1) C-language interface인 NVIDIA's CUDA는 also available
now for programming GPU
 - 다른 graphic programming interface인 다른 API가 있는
경우 CUDA는 GPU 처리의 general-purpose Computing을 쉽게
할 수 있습니다.

- 일반 CPU, GPU 처리의 차이

◎ GFLOPS: Giga Floating-point operations per second

- numerical processing을 위한 CPU multicore 사용을 다른 GP-GPU
를 사용하는 경우 더 benefit한 hardware benefit GFLOPS을 제공
할 수 있습니다.



*Paragraph 4

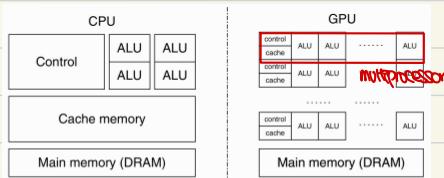
- GPU의 특성은 multiple-data and multiple-thread (MDMT) Processor. GPU는 much more computational power for general-purpose Computing, lower equipment cost, lower power consumption을 가진다.

ex) NVIDIA GPU : a few hundred dollar, 300W power, can solve numerical problems about 40 or more times faster than CPU

* 이 점에서는 Uniformly fine-grained data-parallel Computing 과 같은 GPU 기반 컴퓨팅에 대한 일반적인 design principle을 살펴본다.

- 이 점에서는 Clustering 과 regression clustering에 대해 살펴보자.

1. Overview of a GP-GPU



→ GPU는 algorithm, data structure, programming strategies를 살펴 볼 수 있다.

- GPU는 two-level의 계층 구조를 갖는 아키텍처이다.
 - QoS. → multiple multiprocessor
 - → multiple lightweight data-parallel cores.

* Paragraph 1. - GP-GPU의 특징과 원리

① Multiple multiprocessor

- multiprocessor는 대체 lightweight data-parallel Core를 가지고 있다.
- 동일한 multiprocessor 내의 Cores는 다른 Thread 사이에 짧은 면적 대로인 cache를 공유한다 (16k ~ 32k)
- 다른 multiprocessor 내에서는 Cores는 GPU의 global memory를 통해 Communication 한다. GPU의 global memory는 cache된다. 이로 인해 access latency time을 가진다.
- 같은 블록 내에서 병렬로 실행된다.
- 같은 블록 내에서 병렬로 thread들은 서로无关하게 같은 multiprocessor에서 실행된다.

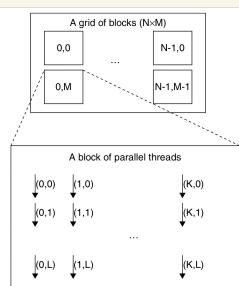


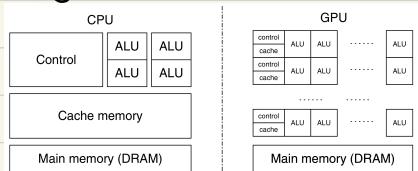
Figure 5.3 Computing threads are organized into block

- 여러가지의 스트림 별을 통합한 몇몇 프로세서로 표기하는 **설비플랫폼** 방식으로 통합하여 메모리 예상스스로 시간을 숨길 수 있다
→ 대기 시간동안 다른 별과의 스트림을 섞어
ALU(예상부호)
- NOT, AND, OR

* Paragraph 2 - the block has three lines of code by step

ALV (અસ્તુતીનામ)

- HEIGHT, AND, OR



- 각 멀티프로세서는 같은 Arithmetic Logic Units를 갖는다 (Core 핵심 개념)
 - 같은 멀티프로세서의 다른 멀티프로세서는 같은 ALU를 쓴다

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

- multicore processor에 있는 일부 processor 부문장치는 다른 A를 만족하는데 task 1을 수행
 - 나머지 processor는 아무것도 하지 않고 대기
 - 이제 다른 도입부의 반대로 기다린 processor들이 task 2를 수행, task 1을 수행하는 processor들을 대기

→ 이러한 예시는 PROCESSOR의 활용도를 높이기 위해 포함되어야 하는 고급 실행 불록의 수를 최대화하는데 Code를 최적화하는 것이 중요성을 말한다.

* Paragraph 3 - Grid. Block. thread 관계 설명

② Threads block

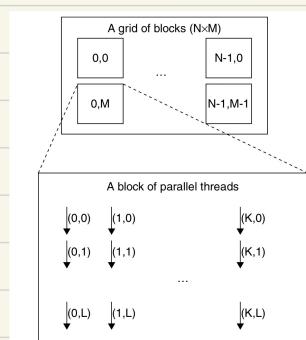


Figure 5.3 Computing threads are organized into blocks.

- GPU가 Computing threads를 토해 어떤 index를 가진 thread block을 처리할지 알다.
 - 같은 threads block를 Block ID를 가진 같은 threads grid를 카운한다.
 - threads grid는 실행의 단위이다.
 - 칸별로 양이 threads block를 위한 multiprocessor> 버퍼가 사용될 수 있도록 한다가 원래는 칸별이다.

- NVIDIA는 thread block의 유효한 수의 threads 계산하고, 유효한 thread block을 찾는다. CUDA Occupancy Calculator Spreadsheet을 사용한다.
 - 소스는 block ID와 thread ID를 data index로 data에 접근하여 parallel computing을 구현하는 dataset이다.

*Paragraph 4 - CUDA 를 통한 사용자 미한 설명

- 다음처럼 NVIDIA-CUDA Interface로 GPU graphic interface에 Direct API가 없는 C Programmer들은 GPU Programming이 가능하다.
- CUDA는 프로그래머가 GPU 처리를 험하게 한다. CPU 메모리와 GPU 메모리 사이 데이터를 복사하고 Parallel Computing thread를 관리하고, GPU에서 노드의 실행을 시작하는 것을 즐기 위한 편리한 API를 제공한다.
- 같은 C programmer, CUDA는 프로그래머 C에서 CPU, GPU or both 둘다 활용될지 어떤지를 지정할 수 있도록 한다.
- 마지막으로 GPU에 대한 간접한 설명이며, 이 협력에서 주로 Machine Learning 알고리즘을 GPU에서 실행하기 위해 디자인 하는 것이다.

2 Uniformly Fine-Grained Data-Parallel Computing on GPU

2-0. Introduction

- 모든 algorithm은 GPU에서 잘 실행되는 것은 아님
- Uniformly fine-grained data-parallel algorithm은 실행될 수 있는 몇 가지 알고리즘을 살펴보며 이는 GPU 환경에서 높은 성능을 보일을 증명한다.

2-1. Data Parallel Computing

*Paragraph 1 - thread들이 data Parallel Computing 방법

- Data Parallelism은 very common Parallel Computing Paradigm이다.
- $X = \sum_{j=0}^{J-1} d_j$ - Partition of dataset
- $T = \sum_{j=0}^{J-1} Proc(d_j)$ - Computation procedure Proc 대로 병렬적으로 적용
- $T = \sum_{j=0}^{J-1} Proc(d_j)$ - J-th Computing thread 합당

→ Data Parallelism은 대체로 같이 표현: threads : Proc(d_j)

→ 각각의 Computing thread는 data of one thread의 Proc 연산을 수행한다.

thread가 다른 thread에 의해
影响하는 경우

- 01 form many different threads \rightarrow 资源共享共享할 수 있다.
• \therefore thread는 block 안에资源共享, Parallel하게 되어야 가능하다.
- NVIDIA's GPU는 \rightarrow Computing Architecture에서 broadcasting
 - Some local/shared memory는 같은 thread끼리资源共享
 - reading + writing共享, shared parameters는 같은 thread끼리资源共享 가능
- \rightarrow threads : proc (shared.read-only_params, d)
- 예 방식이 되도록 경우 다양한 유형의 parameter Code or Constant 형태로 제공된다. GPU는 자체 processor들은 blocking 할 때만 read 가능하다.
- 예상 동작은 Point-to-point parallel computing structure에서 different thread들은 write해야 하는 범위를 겹친다.
 \rightarrow 이와는 특성을 Share NO WRITE Variables (SNOWV)라고 한다

* Paragraph 2

ex) Simple Summation task over N numbers

$$S = \sum_{i=0}^{N-1} a_i - \text{번번이 } J\text{를 사용하여 } N\text{개의 } a_i\text{를 더함}$$

① Step 1.

- dataset을 J개의 partitions로 나누기
- each partitions의 summation 계산 $(0+1)M-1$
- $S_j = \sum_{i=jM}^{(j+1)M-1} a_i$, where $M = N / J$
- M 은 partition에 있는 데이터 개수
 $a_i = \sum_{j=M}^{(j+1)M-1} a_i$
- 각 partition의 summation은 separate computing thread로 향해 수행되며, different thread는 written variables를 겹친다. SNOWV를 적용함

② Step 2.

- Step 1의 computing threads는 각각 수행된 결과를 summation을 해
- total summation 계산 $S = \sum_{j=0}^{J-1} S_j$

정답이 뭐가 됩니까?

- $\sigma_{1,1}, \dots, \sigma_{1,m}$ on X_1 ,
- $\sigma_{2,1}, \dots, \sigma_{2,m}$ on X_2 ,
- ...
- $\sigma_{L,1}, \dots, \sigma_{L,m}$ on X_L .

- data parallel computing Segments를 만들기 위해
 ↳ general Statistical estimator을 위한
SUFFICIENT STATISTIC 개념을 알 수 있다.

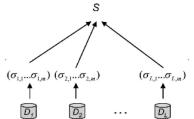
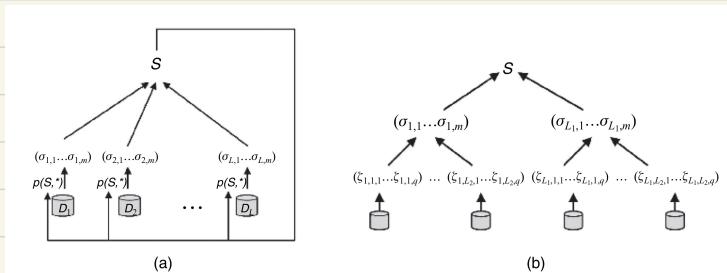


Figure 5.4 Distributed computing by communicating sufficient statistics.

- estimator는 계산하는 목적으로 sufficient statistic을 알면, sample data는 본래 추가적인 information은 필요하지 않다.

- 우리는 whole dataset을 partitions로 나누면 $\rightarrow X_1, \dots, X_L$
 → 각각의 편집은 그 자체가 확률에 각 Partition에 대해서 a set of sufficient statistic을 갖는다.
- SE sufficient statistic들은 그 자체로 계산하는데 쓰여짐
 → sufficient statistic 이외의 information은 필요하지 않음
- THIS basic idea는 k-mean clustering이나 hierarchical clustering
 또는 computing task tree를 생성하는 모범적인 방식으로 사용될 수 있다.



- (b)와 같은 recursion tree에서는 data들은 계층으로 풀고 턱은 단위로 나누는다.
- Iteration과 recursion은 combine될 수 있다.
 each distributed location의 data 편집은 별도로 더 활용되고 퍼포먼스
 distributed computing이 가능
- sufficient statistic 개념은 dataset 수와 상관없이 Constant
- ol chapter에서 k-mean과 regression clustering 위 틀에 대한 설명
- same principle of communicating sufficient statistic is used by Mapreduce

- Map phase: maps a computation to a distributed dataset on many computers and collects the results from all processes
- Reduce: reduce the results to the final result

2.2 Uniformly Fine-Grained Data-Parallel Design

* Paragraph 1 - Uniformly fine-grained data-parallel MIMD

- 동일한 행정작업 단위는 모든 Processor가 동시에 할 수 있는 multiprocessor의 Processor 단위로 이루어져야 한다.
- multiprocessor의 각 Processor마다 동일한 다른 단위를 통해 같은 경우 일련 Processor가 처리해야 하는 multiprocessor의 환경에서 처리됨
- GPU에서 실행되는 program의 vector, coalesced and well-aligned memory access pattern은 memory bandwidth의 높은 활용률을 달성하는 데 핵심이다. → code structure memory access가 환경으로 아울러져 함
- multiprocessing은 GPU board의 global memory의 다른 접근 시각을 통한 데이터 사용
- memory에 대한 scattered writes는 GPU 성능을 저하시키고 문제가 된다.
 - 다음 장의 많은 algorithm은 GPU를 살피는데 이를 제시한다.
 - 이 chapter에서 GPU를 살피는 프로그램은 Uniformly fine-grained data parallel algorithm을 characterize 한다.
- Uniformly fine-grained data Parallel algorithm
 - Uniformity: Uniformity of executions
 - Uniformity of data partitioning → 병렬화는 매우 낮은 단위에서 이루어짐
 - fine-granularity: Parallelization is implemented at the lowest level
 - : Individual data access request + memory unit 이하로
하는 경우 instruction은 multiprocessor마다 별도로
실행된다 → memory access와 Instruction이 동시에
된다
 - 다른 single memory access가 같은 Processor
단위에 수 있는 경우 문제는 없다.
 - 다른 memory access latency를 고려해

Computation, data access 헷갈리면 → ex) $int d_1 = \text{Eigen}(\text{Eigen} \rightarrow \text{연산} \rightarrow \text{행렬})$
 $(\text{data access}) \rightarrow \text{data access latency}$

↑
 본래된 data access phaseset Computation Phase² 가 되는 건은 not good design
 Computation을 data access로 헷갈리면 행렬로 생각하는 걸 헷갈리게 됨
 이를 위해 fine-grained data Parallel computation or coarse grained separation 본래 data access 와 computation 헷갈리게 만들기

* Paragraph2 - caused memory access

- GPU에서 C++ Programming 터키 에디. 알고리즘은 Uniformly fine-grained data parallelism을 허용하는 모든 Coalesced memory access patterns를 지원하는 Code를 작성하는데 매우 유용하다 → 특히 두 가지로 보여지는 algorithmic DE partition parallelization과 NVIDIA의 GPU에서 Coalesced memory access는 Program의 성능을 상당히 향상시킨다.
 - 특히 Uniformly fine-grained data Parallelism은 each data partition가 $\frac{1}{N} \times N$ 의 size를 갖는다. 각 데이터 허브는 면적이 충분하며, 각 허브는 가능한 모든 memory access cycle로 동시에 필요한 data partition의 access를 한다.
↓
각 thread 허브를 지향하는 많은 access 대상 (fine-grained)

* Paragraph 3

(e) $S = \sum_{i=0}^{N-1} a_i$ - we wish to design uniformly fine-grained data parallel program to perform this task!

- example 1 ~~에서~~ 128 lines of cache filled by the byte streams from each $d_j = \sum_{i=0}^{2^{(3j+1)-1}}$ → this is not fine-grained → ~~단일한 일정한 index 주소로 주출~~
 - example 2 ~~에서~~ Uniformly fine-grained requirement을 만족시키고자
we choose the partitions at the individual value level $d_j = \sum_{i=0}^3$
 - thread PROCm - Simple summation on data d_j
 - $\text{PROC}_m(d_j) = S_m + d_j$
 - Parallel thread ~~에서~~单一 data request를 보내면, 128 byte의 Cache line
많은 Parallel threads or ~~에서~~ 캐시에 훈련된다
 - $d_0 \dots d_7$ data ~~에서~~ 캐시에 load → ~~data~~ ~~에서~~ ~~캐시에~~ 훈련된다
 - data requests from parallel threads PROC₀ ... PROC₇ are all satisfied
 - each thread는 즉시 소비될 가장 빠른 양의 데이터만 프로세스한다

* Uniformly fine grained data Parallel Computing

- multprocessors 1개의 memory access cycle에서 최대 16 contiguous Bytes 를 global memory block으로부터 연속을 수 있음
fine-grained data 단위 Processor cycle의 활용률을 높이 흐리게 유지 가능
- data partition 단위 size 충분 \Rightarrow cache에 수렴되는 단위 Partition 활용

Uniformly

- 각 task는 연속된 필요한 data partition 개수를 기반한 흐로 memory access cycle 동안 처리 가능
- 같은 단위의 단위는 같은 모든 threads의 proc 단위로 통일화되어 수행되는 시간이 짧도록 해줌

Coarse - grained

- Cache memory line (16 bytes)
are filled by $d_1 = \sum_{j=1}^{(32)N-1} d_j$

Fine - grained

$$\cdot Pm(t_0) = S_m + = d_1$$

3. The k-means clustering algorithm

3-1. Uniformly Fine - Grained Data Parallelism in k-means

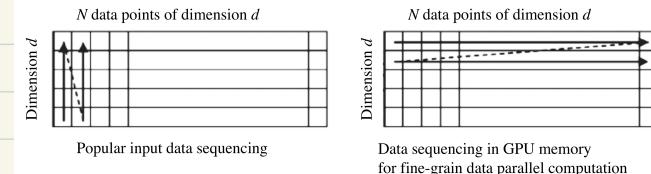
* Paragraph 1 - k means clustering의 sufficient statistics

- Dataset은 thread block 크기 일정하는 data point 수를 가진 블록으로 분할되며, 그때 block 크기는 NVIDIA GPU 사용에 맞는 수준에 따라 달라집니다.
- 다른 data block들은 distributed dataset으로 처리된다.
- 같은 datasets의 각 thread block에서는 수준의 sufficient statistics는 GPU or CPU에서 일정한 단위로 segmentation되어 각각 계산된다.
- 이것은 단계 계산하는 것이 좋지만, 이 경우에는 global communication이 발생함.

* Paragraph 2 - k-Mean clustering의 parallelism 과정

- first we need to look at k-means algorithm as a SIMD parallel or SIMD fashion으로 수행될 수 있다. → 모든 data가 단일 인덱스에 저장되는 경우
- 각 thread는 단일 data point에서 모든 중심점과의 거리를 구하고, 최소 거리를 가지 헤더는 centroid를 찾으며, data point를 그 centroid의 member로 할당하는 역할을 수행한다.
- centroid들은 ~~are~~ multiprocessor의 read-only variables로 되며, 모든 processor가 이를 공유한다. (SIMD)
- K-Mean clustering에서 가장 비용 소모가 큰 part는 바로 인덱스 centroid를 찾는 membership으로 할당하는 part이다.
→ 따라서 할당 part의 예선을 GPU에서 병렬로 처리한다.
- GPU에서 계산된 membership vector는 CPU가 중심을 update하는 있도록 각 iteration마다 GPU와 CPU를 연동한다.

- * Paragraphs - K-Mean clustering at Fine-grained data parallel
 - thread block이 있는 thread data 를 KB 단위로 기반으로 병렬 처리되며, fine-grained data-parallel computation이 가능해진다.



- thread Block의 thread가 각각 헤당하는 data 를 동시에 처리할 때
이전에 헤당한 블록은 Global memory의 어느 chunk에 있는지 알게 된다.
ie block 간의 데이터가 혼란된 경우
- 원래 synchronous computing 스텝에서, 각 thread가 헤당하는 data 허브의 모든
이(Coarse-grained)는 병렬된 memory access에 대해서

Algorithm (2) 설명 \rightarrow Fine-grained을 이용한 k-means

4. K-Means Regression Clustering Algorithm

4-0. Basic Concept of Regression Clustering

- Regression clustering (RC) is a combination of clustering and regression
- clustering is unsupervised learning
- regression is supervised learning which aims to fit all the training data with only single function
- Regression clustering은 각각의 Part of dataset 을 하나의 multiple regression function에 대응시킨다.

- Regression clustering은 2가지 단계로 나눠서 실행된다.
 - clustering : 각각의 데이터에 대해서, clustering label을 할당한다.
 - regression : 각 cluster에 속하는 데이터에 대해 supervised learning과 같은 방식으로 regression function을 계산한다
- \rightarrow 각 regression function은 cluster centroid로 각 cluster의 데이터를 가장 잘 대표하는 regression function으로 한다.

- Centroid based k-means clustering과 비슷하게 regression cluster의 k-means의 centroids는 more complex data model로 대체된다.

- normal regression & clustering 차의 차이점

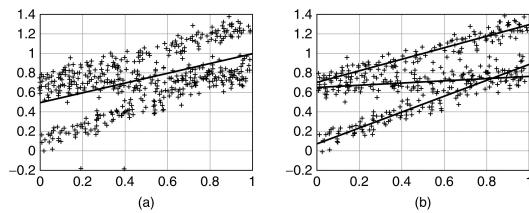
$\Phi = \{f_1, f_2, \dots, f_m\}$: a family of a function \rightarrow regression function들의 집합

$e(\cdot)$: a loss function

$$f^{opt} = \arg \min_{f \in \Phi} \sum_{i=1}^N e(f(x_i), y_i)$$

$$e(f(x), y) = \|f(x) - y\|_p$$

\hookrightarrow loss function을 통해 각각의 regression function을 찾는다.



- (a)는 각각의 데이터가獨立하는
집단에 속하는 경우, (b)는 각각의
cluster에 Partition을 했고, 각 partition
에 대해서 다른 function을 하게
시켜는 경우 더욱 효과적이다.

\rightarrow 이것이 RCE의 idea이다.

- Regression은 각 cluster에 대해서 model을 적용해 주고, 반면에 clustering은 model을 적용해 주고, 각 cluster에 대해서 모델을 적용할 수 있도록 데이터를 분리해 준다.
 \rightarrow 그 이유 algorithm that integrates 서로 같은 Common object function이다

RC Algorithm의 특징

- RC Algorithm은 Centroid based clustering algorithm과 거의 동일한
a set of regression functions $M = \{f_1, \dots, f_k\} \subset \Phi$ 로 대체된다.
- objective function은 k-means의 objective function과 유사하다.
data point와 Center 간의 거리와 regression error를 더해
 $e(f(x), y) = \|f(x) - y\|^2,$

$$d((x, y), M) = \min_{f \in M} (e(f(x), y))$$

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 $\arg \min_j \{e(f_j(x), y) | j = 1, \dots, k\}$. 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 1: initialize the regression function

Step: Φ data (x_i, y_i) 에 대해, k 개의 regression function과 error를
이 가중치로 function f 를 찾고 ($\arg \min_j \dots$) j member를 할당
 \rightarrow 이 단계에서 data가 k 개의 partitions로 나누어진다.

Step 2: $f^{opt} = \arg \min_{f \in \Phi} \sum_{i=1}^N e(f(x_i), y_i)$ - 다음 objective function을 maximize하는
regression function을 다시 찾는다.

Step 4: Group data에 대해 Step 1. 과정을 반복
- k-mean과 유사한 차이점은, centroid regression functions 대신하고,
centroid가 아닌 regression function의 error를 사용하는 점이다.

4-1. Fine grained data Parallel Structures in k-means RC on a GPU

· k-means clustering은 어떤가? k-means RC 또한 25 iteration 뒤에 two phases를 거친다.

- membership calculation + calculating the new function

① membership Calculation Step

- all k functions은 25 threads로 각각 실행되는 힘을 고려해 미개별화 사용되는 모든 멀티프로세스의 쪽수 대로 나눠준다.

- each thread는 자신의 data pointer를 통해, cluster membership을 계산하기 된다.

② Calculating the new center function Step *

- regression 단계에서는 다중 회귀를 수행하는 힘

- 각 군집에 속하는 data point들은 MSE 값이 가장 낮아서 그 군집에 속하는지에 의해 center-function을 update하는데 사용된다.

- 각 군집의 cluster center function을 계산해 놓음

- 그는 각 군집의 cluster에 속한 data 개수

$$A_j = \begin{bmatrix} x_{i_1} \\ x_{i_2} \\ \dots \\ x_{i_{L_j}} \end{bmatrix}, b_j = \begin{bmatrix} y_{i_1} \\ y_{i_2} \\ \dots \\ y_{i_{L_j}} \end{bmatrix}$$

- A는 cluster에 속한 데이터의 행렬로 이루어진 matrix
- bj는 cluster에 속한 데이터의 행렬로 이루어진 vector

→ 각각의 cluster의 수로 center function은 $c_j = (A_j^T A_j)^{-1} A_j^T b_j$

→ $AX=b$ 를 만족하는 X 는 \rightarrow 양 해할 수 있음

- Cluster의 data point가 많으면 C_j 를 계산하는 비용이 상당하게 된다.

→ 따라서 이 연산 과정을 GPU의 Parallel 환경에 활용한다.

- Fine-grained data structure를 활용하기 위해 C_j 를 위한 연산을 더 자세히 살펴보자

$$A_j^T A_j = \sum_{l=1}^{L_j} x_{i_l}^T * x_{i_l}$$

$$A_j^T b_j = \sum_{l=1}^{L_j} x_{i_l}^T * b_j$$

→ 다음 방식들은 계산 단계의 data Parallel 구조를 보여주며, 모든은 1번의

↑ 개별적 인 허가를 많은 것은 허가하고 다른 연산들이 병행할 수 있다.

↑ 병렬화 가능

- 한 번의 cluster의 data는 동일한 크기를 가진 하나의 chunk data를 통해
도는 각 data chunk가 multiprocessor에서 실행되는 한 개의 thread block
에 할당된다.
- Cache를 포함한 해당 multiprocessor에서 실행되는 하나의 thread block는
각각 b_i 가 shared memory에 load된 상태로 $x_{i_1}^T * x_{i_1}$ and $x_{i_1}^T * b_j$, $\forall j \in [k]$
fine-grained computation을 수행된다.

5. Implementations and Performance Comparisons

5-0. Introduction

- 주제 논문에서는 Algorithm 자체가 아닌 GPU에 의한 성능 개선 방안으로, randomly generated dataset을 사용
- 모든 비교를 위한 각 Iteration의 cost는 계산하기에 충분하며, BE experiment에서 iteration 회수는 제한을 두지 않음
- reported time은 Open GPU API 툴 및 계산 시간을 포함한 모든 Iteration의
wall clock time이다. but initializing the dataset은 다른 time은 포함하지
않다.
- CPU 및 GPU 버전 모두 동일한 단계로 처리하는 동일한 centroid를 생성
 \rightarrow 다른 GPU 구현의 algorithm 동작률을 확인 가능

5-1. CPU-Only Implementation of k-means