

# CS492 Distributed Systems & Algorithms

## Final Report: Mr.CL

20030767 최종욱, 20050145 김준기, 20060080 김민국

2009 년 12 월 24 일

### 1 Introduction

우리는 지난 학기 동안 분산 시스템의 기본적인 구조와 MapReduce, 그리고 분산 시스템 상에서 이용되는 여러 알고리즘에 대해 공부해왔다. 특히나 3 가지 프로젝트를 통해 Hadoop 을 이용한 MapReduce 프레임워크 기반으로 실제 프로그램을 구현해 보면서 이에 관해 더욱 깊은 이해를 할 수 있었다. 이를 통해서 MapReduce 프레임워크가 임의의 규모로 클러스터를 확장시키기 편리하여 대용량의 데이터를 처리하는 데 무척 용이함을 알게 되었다. 우리는 기존에 해결하기 어려웠지만 Hadoop 을 통해 새로운 접근 방법을 시도해볼 수 있는 문제를 찾다가 matrix multiplication 이라는 주제를 선택하였다.

Matrix multiplication 은 과학 연산과 데이터 마이닝 등 여러 분야에서 많이 이용되는 기본적인 연산이며 매우 방대한 크기의 matrix 를 이용해 계산하는 경우가 자주 발생한다. 이 때문에 이를 최적화 시키기 위한 많은 시도가 이루어져 왔는데, 우리는 MapReduce 를 통해서 임의의 대용량의 matrix 를 가지고 multiplication 을 수행하는 알고리즘을 개발하고자 한다.

이렇게 방대한 크기의 matrix 를 다룰 수 있게 하는 것도 중요하지만, 연산 수행에 필요한 시간을 단축키는 노력 또한 함께 이루어져야 한다. 따라서 더 빠르게 계산을 수행할 수 있도록 matrix multiplication 과정에서 GPU 에 의한 가속 기능을 도입하였다. GPU 는 그 성격상 실수와 matrix 연산에 특화되어

일반 CPU 보다 훨씬 더 나은 성능을 보여주기 때문이다.

이러한 아이디어를 바탕으로 하여, 우리는 GPU 를 이용한 분산 시스템 상에서의 Hadoop 을 이용한 matrix multiplication 알고리즘을 설계하였다. 분산 시스템을 통해서 대용량의 데이터에 대한 확장성 (scalability) 을 얻고, GPU 를 통해 연산의 속도 향상을 이룸으로써 이전보다 개선된 matrix multiplication 을 수행하고자 한다.

### 2 Design Overview

#### 2.1 Assumptions

우리가 목표로 하는 시스템의 효과적인 설계와 구현을 위해 약간의 가정이 필요하다.

- Matrix 의 각 원소들은 single precision floating point format 을 통해서 표현된다. CUDA 라이브러리에서 double precision 을 지원하는 하지만 JCublas 와 연동할 때 내부적인 변환에 따른 성능 저하 및 심각한 계산 오류가 발생하였다.<sup>1</sup>
- 각 matrix 는 dense matrix 이다. Sparse matrix 의 경우 모든 원소를 연속적으로 저장하지 않고 개별적인 쌍으로 저장하기 때문에 원소를 접근할 때 index 를 key 로 하는 탐색 작업이

<sup>1</sup>과일럿 테스트 과정에서 무작위로 생성한  $100 \times 100$  행렬을 이용해 곱셈을 했을 때 single precision 으로 입력을 주어 계산하면 한 원소가 250 정도의 값을 가지는데 double precision 의 경우 0 에 매우 가까운 실수가 되어 correctness 판정을 통과하지 못했다.

필요하다. 이것은 GPU 처럼 연속적인 메모리를 할당하여 한꺼번에 pipelining 하는 구조에 적합하지 않으므로 우리는 dense matrix 인 경우만 고려하였다.

## 2.2 Block Approach

Dense matrix multiplication을 수행할 때 I/O 병목에 의해 계산 속도가 제약되는 이유는  $O(n^3)$ 의 곱셈 횟수와  $O(n^2)$ 의 element를 불러오는 횟수가 matrix가 커질수록 큰 차이가 나기 때문이다. 특히 이러한 I/O 병목 문제는 multi-processor 나 네트워크 기반 클러스터로 규모를 확대할수록 더욱 심각해지는데, 이는 프로세서끼리 혹은 클러스터 상의 컴퓨터끼리의 통신 대역폭이 제한되어 있기 때문이다. 따라서 우리는 프로세서의 성능을 최대한으로 활용하기 위해 프로세서가 처리할 데이터가 가능한 한 지속적으로 공급될 수 있도록, 혹은 데이터를 한번 불러왔을 때 가능한 한 많이 반복적으로 연산에 활용하는 알고리즘과 코드를 작성해야 한다.

이 문제를 해결하기 위해 우리는 matrix를 block 단위로 쪼개어 적절히 이를 분산시켜 multiplication을 수행하는 방법을 선택하였다. 만약 각 block들이 적당히 큰 상황이라면, 우리는 각 노드 상에서 이러한 block들의 각 element들을 한 번 불러와 적당한 횟수만큼 multiplication에 다시 이용할 수 있을 것이다. 이때 각 block들은 일반적으로 square matrix지만 아닌 경우도 있을 수 있으므로 알고리즘에 따라서 적절한 block decomposition을 구현하는 것이 중요하다.

### 2.2.1 Outer Product Algorithm

Matrix multiplication을 구현하는 방법은 여러 가지가 있는데, 정의대로 계산하는  $O(n^3)$  짜리와  $O(n^{2.376})$ 의 시간이 걸리는 Strassen 알고리즘 등이 있으나 우리는 MapReduce에서 쓰기 알맞도록 outer product를 응용한 방식을 사용하였다.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} = \begin{bmatrix} 1a & 1d \\ 4a & 4d \\ 7a & 7d \end{bmatrix} + \begin{bmatrix} 2b & 2e \\ 5b & 5e \\ 8b & 8e \end{bmatrix} + \begin{bmatrix} 3c & 3f \\ 6c & 6f \\ 9c & 9f \end{bmatrix} \quad (1)$$

Eq. (1)를 보면 두 matrix의 열과 행 vector들을 각각 outer product를 취하여 나온 matrix들을 모두 더하면 전체 multiplication 결과를 완성할 수 있음을 알 수 있다.

여기서 각 원소들은 하나의 실수가 아니라 submatrix(앞에서 block이라고 표현했던 matrix의 일부)여도 되므로, 우리는 이처럼 block 단위로 쪼개고 다시 합치는 과정을 MapReduce로, 각 block들을 곱하는 과정을 개별 노드에서 GPU로 가속하는 방식을 취하였다.

## 2.3 NVIDIA CUDA

실험에 사용된 클러스터 상의 모든 노드에는 NVIDIA의 그래픽 카드가 장착되어 있다. NVIDIA에서는 CUDA라는 기술을 통해 개발자들이 C언어로 직접 GPU에서 동작하는 프로그램을 만들 수 있도록 하고 있다. 이것을 기반으로 BLAS API<sup>2</sup>를 구현한 CUBLAS를 이용하면 matrix multiplication을 GPU를 이용하여 수행할 수 있다.

## 2.4 Apache Hama Project

프로젝트 초기에는 matrix multiplication 구현 자체에 대한 부담을 줄이고 GPU를 이용한 가속에 집중하기 위해 이미 Hadoop 기반으로 다양한 matrix 연산을 구현하고 있는 Apache 재단의 Hama 프로젝트를 이용하였다. 하지만 다음과 같은 이유로 Hama 프로젝트를 사용하지 않고 직접 Hadoop MapReduce로 구현하게 되었다.

- Hama가 Hadoop 기반 데이터베이스인 HBase를 이용하고 있었고, 아직 HBase가 성

<sup>2</sup>Basic Linear Algebra Subprograms. Vector-Vector, Vector-Matrix, Matrix-Matrix 연산들을 단계별로 정의하고 있다.

능 측면이나 프로젝트 자체의 성숙도가 떨어져 실제 우리가 원하는 규모의 연산을 돌리기에는 적합치 않았다.<sup>3</sup> HBase를 들어내려면 Hama를 거의 처음부터 다시 짜야 한다.

- Hama에서는 matrix의 원소들을 기본으로 double precision으로 처리하고 있는데 여기에 CUDA를 연동시키려면 single precision으로 변환하는 과정이 필요하여 불필요한 성능 저하를 발생시키고 이렇게 동작하도록 Hama의 코드 전체를 바꾸는 것이 용이하지 않았다.
- 곱셈 과정에서 block 쌍들을 모두 별도로 저장하였다가 불러오는데, 이 과정의 I/O complexity가  $O(n^3)$  이어서 새로운 접근 방법이 필요했고 결국 이것이  $O(4n^2)$ 으로 줄어든 위의 outer product algorithm을 새로 구현해야 했다.

### 3 Implementation

TODO: 최종적으로 Hadoop 위에 바로 코딩해서 올린 구조 설명

#### 3.1 Data Model

#### 3.2 MapReduce Execution

##### 3.2.1 Mapper

**Input**

blah blah

**Output**

blah blah

**Mechanism**

blah blah

##### 3.2.2 Reducer

**Input**

blah blah

**Output**

blah blah

**Mechanism**

blah blah

### 4 Performance Analysis

#### 4.1 Pure Java

#### 4.2 JCublas with NVIDIA CUDA

### 5 Conclusion

TODO: 부딪힌 문제점들 설명 (Hama와 HBase의 설치 중 삽질한 것, CUDA 드라이버 관련 삽질한 것, JCublas에서 double/float 삽질한 것 등)

TODO: 앞으로 개선할 점과 계속해서 연구한다면 어떤 것들을 해보고 싶다 등등

---

<sup>3</sup>우리 목표는  $10^4$  이나  $10^5$  정도인데, Hama 개발자 스스로 자신의 블로그에서  $5000 \times 5000$  정도가 한계라고 밝히고 있음. (<http://blog.udanax.org/2009/01/distributed-matrix-multiplication-with.html>)