ISSN 1975-8359(Print) / ISSN 2287-4364(Online)
The Transactions of the Korean Institute of Electrical Engineers Vol. 67, No. 2, pp. 270~276, 2018
http://doi.org/10.5370/KIEE.2018.67.2.270

OpenCL을 활용한 이기종 파이프라인 컴퓨팅 기반 Spark 프레임워크

Spark Framework Based on a Heterogenous Pipeline Computing with OpenCL

김 대 희* · 박 능 수†
(Daehee Kim · Neungsoo Park)

Abstract - Apache Spark is one of the high performance in-memory computing frameworks for big-data processing. Recently, to improve the performance, general-purpose computing on graphics processing unit(GPGPU) is adapted to Apache Spark framework. Previous Spark-GPGPU frameworks focus on overcoming the difficulty of an implementation resulting from the difference between the computation environment of GPGPU and Spark framework. In this paper, we propose a Spark framework based on a heterogenous pipeline computing with OpenCL to further improve the performance. The proposed framework overlaps the Java-to-Native memory copies of CPU with CPU-GPU communications(DMA) and GPU kernel computations to hide the CPU idle time. Also, CPU-GPU communication buffers are implemented with switching dual buffers, which reduce the mapped memory region resulting in decreasing memory mapping overhead. Experimental results showed that the proposed Spark framework based on a heterogenous pipeline computing with OpenCL had up to 2.13 times faster than the previous Spark framework using OpenCL.

Key Words : HPC, GPGPU, OpenCL, Apache spark, Parallel computing

1. 서 론

최근 머신러닝과 사물인터넷 등 Industry 4.0 기반 기술의 확산으로 빅-데이터를 처리하는 기술의 수요와 관심이 증대되고 있다. 그러나 하둡(Hadoop)과 같은 분산처리 프레임워크는 맵-리듀스(Map-reduce) 연산으로 빅-데이터를 처리할 수 있지만 디스크 입출력(I/O) 오버헤드로 인해 성능 한계를 보이고 있다[1]. 이러한 하둡의 단점을 보안하고 이를 고속으로 처리하기 위해 인-메모리 기반으로 처리하는 Spark 프레임워크가 등장하였다[2, 3]. 하지만 Spark 프레임워크 만으로 늘어나는 데이터를 고속으로 처리하기 힘들고 더 높은 처리 성능을 얻기 위해 SparkJNI [4], SparkCL[5], IBMSparkGPU 등과 같이 Spark 프레임워크에 GPGPU (General Purpose computing on Graphics Processing Unit)를 융합한 연구가 활발히 진행되고 있다. Spark 프레임워크에서 GPGPU를 활용하는 것은 매우 어렵고 복잡한 과정이기 때문에 대부분의 연구가 GPGPU를 쉽게 활용하는 것에 초점을 두고 있다. 따라서 Spark 프레임워크에서 GPGPU를 보다 효율적으로 활용하여 성능을 향상시킬 수 있는 프레임워크 기술 연구가 필요한 시점이다. 본 논문은 OpenCL(Open Computing Language)을 통해 GPGPU를 활용하는 Spark 프레임워크를 분석하고 이를 기반으로 전체 성능을 개선하고자 한다.

OpenCL 기반 Spark 프레임워크의 내부 연산과정에는 Java 메모리 영역과 Native 메모리 영역 간의 전송(CPU 처리), 호스트와 GPU 사이의 데이터 전송(DMA 처리), GPU 커널 연산 등이 포함이 된다[6]. 일반적으로 데이터의 크기가 커질수록 Java 메모리 영역과 Native 메모리 영역 간의 전송과 호스트-디바이스 통신 오버헤드 또한 커지게 된다. 이를 개선하기 위해 호스트-디바이스 통신을 GPU 커널 연산과 중첩시켜 은닉시키는 파이프라인 기법을 적용 한다[7, 8]. GPGPU 파이프라인 기법을 적용하기 위해서는 호스트와 디바이스간의 통신 버퍼를 mapped-memory로 구성해야한다. mapped-memory는 메모리 매핑과정을 통해 형성되는데 GPU 처리 전에 GPU 연산 준비 과정에서 수행된다. 일반적인 GPGPU 연산은 GPU 연산 준비 과정을 거친 후 GPU 연산이 이루어지기 때문에 이를 제외하고 순수 GPU 계산시간만으로 성능을 평가한다[9]. 하지만 OpenCL 기반 Spark 프레임워크는 Transformation 및 Action 연산과 같은 Spark 연산을 GPU에 할당하고 처리하기 위해 GPU를 호출하는데 이때 GPU 연산 준비 과정을 수행하게 된다. 따라서 OpenCL 기반 Spark 프레임워크의 전체 성능을 평가하기 위해서는 GPU 연산 준비 과정을 성능 측정에 포함해야 한다. 기존의 OpenCL 기반 Spark 프레임워크는 GPU 연산 준비 과정에 단순 버퍼 할당만을 포함하고 GPGPU 파이프라인 기법을 적용하면 mapped-memory 과정까지 포함한다. 특히 메모리 매핑과정은 단순 버퍼 할당시간에 비하여 매우 크다. 따라서 데이터 크기만큼 메모리를 매핑하여 처리하기 때문에 처리할 데이터가 커질수록 GPGPU 파이프라인 기법을 적용한 OpenCL 기반 Spark 프레임워크의 GPU 연산 준

† Corresponding Author : Dept. of Computer Science and Engineering, Konkuk University, Korea.
E-mail: neungsoo@konkuk.ac.kr
* Dept. of Computer Science and Engineering, Konkuk University, Korea.
Received : January 4, 2018; Accepted : January 10, 2018

비 시간이 더 크게 증가하여 기존에 비해 오히려 성능이 감소할 수 있다.

OpenCL 기반 Spark 프레임워크에는 Java 메모리 영역과 Native 메모리 영역간의 전송을 호스트(CPU)가 처리한다. 호스트-디바이스 간의 통신과 GPU 커널 처리 중에는 CPU가 유휴자원으로 낭비가 된다[10]. 따라서 본 논문에서는 GPGPU 파이프라인 기법의 호스트-디바이스 간의 통신과 GPU 커널이 동시 연산되는 중간에 CPU가 다음 호스트-디바이스 간 통신을 위한 데이터를 전송하기 위한 Java-to-Native 간의 전송을 중첩시키는 이기종 파이프라인 컴퓨팅 기법을 제안한다. 특히 Native영역에 호스트-디바이스 간의 전송 버퍼를 이중 버퍼로 구성된 스위칭 듀얼 버퍼 기법으로 구현하여 버퍼를 재사용하면서 메모리 매핑 영역을 감소시켜 OpenCL기반 Spark 프레임워크의 전체 성능을 향상시키고자 한다.

제안한 이기종 파이프라인 기반 Spark 프레임워크의 성능을 평가하기 위하여 Spark 벤치마크 프로그램을 이용하여 실험을 수행하였다. 다양한 크기의 데이터 세트로 실험한 결과 OpenCL 기반 Spark 프레임워크에 GPGPU 파이프라인 기법만을 적용한 프레임워크는 GPU 실행시간은 감소하였으나 GPU 연산 준비 시간 내의 mapped-memory 시간이 증가하여 총 연산 시간이 기존에 비해 최대 34% 증가하였다. 본 연구에서 제안하는 이기종 파이프라인 기법을 적용하면 Java 메모리 영역과 Native 메모리 영역 간의 전송을 다른 DMA 통신과 GPU 커널 연산과 중첩시켜 성능을 향상시킬 수 있다. 또한 스트림 개수가 늘어날수록 스위칭 이중버퍼의 크기를 줄이므로 메모리 매핑 영역이 감소한다. 따라서 32개 스트림을 이용할 경우 GPU 연산 준비 시간이 최대 58% 감소하였고 기존 OpenCL 기반 Spark 프레임워크에 비교하여 전체 처리 성능이 최대 2.13배 그리고 단순 파이프라인 기법을 적용하였을 때 보다 2.84 배 성능 향상되었음을 확인하였다.

본 논문의 구성은 2장에서 GPGPU를 활용한 Spark 프레임워크를 분석하고, 3장에서는 분석 결과를 토대로 스위칭 듀얼 버퍼 기법으로 구현한 이기종 파이프라인 컴퓨팅 기법에 대해 설명한다. 4장에서는 실험 결과를 통해 개선된 프레임워크를 분석하고 5장에서 결론으로 마무리 한다.

2. GPGPU를 활용한 Spark 프레임워크 분석

2.1 OpenCL 기반 Spark 프레임워크

인-메모리 기반으로 분산 처리하는 Spark 프레임워크는 디스크 기반 맵-리듀스로 연산하는 하둡의 단점을 보완하여 현재 많이 활용되고 있다. 하지만 처리할 데이터 크기가 점점 커짐에 따라 더 높은 처리 성능을 요구되고 있어 이를 위하여 GPGPU를 Spark에 접목시킨 다양한 연구가 진행되고 있다. 그러나 Spark는 JVM(Java Virtual Machine)에서 동작하기 때문에 GPU를 활용하기 위해서는 반드시 JNI(Java Native Interface)를 통해 Native 언어를 처리해야 한다[11]. JNI를 이용하여 GPU 프로그래밍을 하는 것은 매우 어렵고 복잡한 과정이므로 대부분의 연구가 GPGPU를 보다 쉽게 활용하는 것에 초점을 두고 있다. 특히 OpenCL 기반 Spark 프레임워크는 Spark 사용자가 GPU 프로그래밍을 위해 추가적인 설정 없이 간단한 함수만 호출하여 사용할 수 있도록 지원하고 있다. 그림 1은 OpenCL 기반 Spark 프레임워크의 실행 모델을 묘사한 것이다. Java-to-OpenCL 프레임워크는 Java에서 JNI를 이용하여 Native 언어를 제어하고 OpenCL 라이브러리 연결을 추상화하여 Spark 프레임워크 내에서 GPU 사용을 간단하게 지원하는 역할을 한다. RDDs(Resilient Distributed Datasets)를 설계하는 Transformation과 Action 연산을 GPU로 병렬 처리함으로써 성능을 향상시키며 MapCL, MapCLPartition, ReduceCL 등 GPU를 사용하는 RDDs 연산을 지원한다.

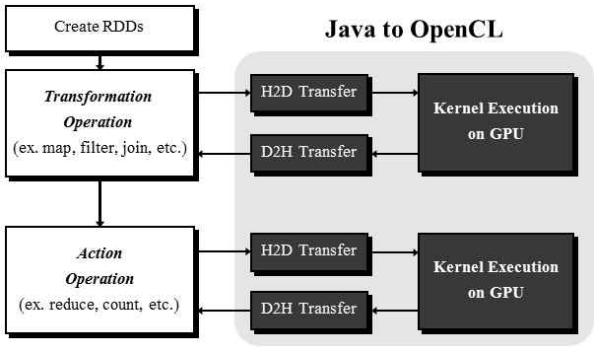


그림 1 OpenCL 기반 Spark 프레임워크
Fig. 1 Spark framework based on OpenCL

2.2 OpenCL 기반 Spark 프레임워크 분석

현재 OpenCL 기반 Spark 프레임워크의 내부 연산과정을 살펴보면 호스트와 GPU 사이의 일련의 데이터 전송 및 처리 과정으로 진행된다. Spark에서 GPU를 활용하기 위해서는 기본적으로 그림 2의 위와 같이 총 7단계의 과정이 필요하다. GPU 연산 준비 과정을 마친 후 Spark 데이터를 GPU가 사용할 수 있도록 Java 힙 메모리 영역에서 Native 영역으로 데이터를 복사하고, GPU 연산 결과를 Spark가 활용할 수 있도록 Java 힙 메모리 영역으로 데이터를 복사한다. GPGPU 연산은 GPU 연산 준비 과정을 거친 후 GPU 연산이 이루어진다. GPU 연산 준비 과정은 디바이스 메모리에 버퍼 생성과 같은 사전 작업이 포함된다. 또한 호스트 CPU와 디바이스 GPU가 함께 접근할 수 있도록 하는 mapped-memory를 할 경우 호스트와 디바이스 메모리를 매핑(Mapping)하는 과정이 추가적으로 발생한다. 그리고 GPU는 데이터를 모두 전송 받은 후 GPU 연산 처리를 하고 그 결과 값을 전부 반환 한 후 다음 태스크의 연산을 수행할 수 있다. 일반적으로 GPU 연산 성능을 평가할 때 GPU 연산 준비 과정은 연산 전에 이루어지므로 이를 제외하고 순수 계산시간만으로 성능을 평가한다. 그러나 Spark 프레임워크에서 GPU를 활용한다는 것은 위의 그림 1에서 보았듯이 Transformation 혹은 Action 연산을 GPU로 처리한다는 의미이다. 즉, OpenCL 기반 Spark 프레임워크는 Spark 태스크 단위로 일부 연산을 GPU로 처리하는 것이기

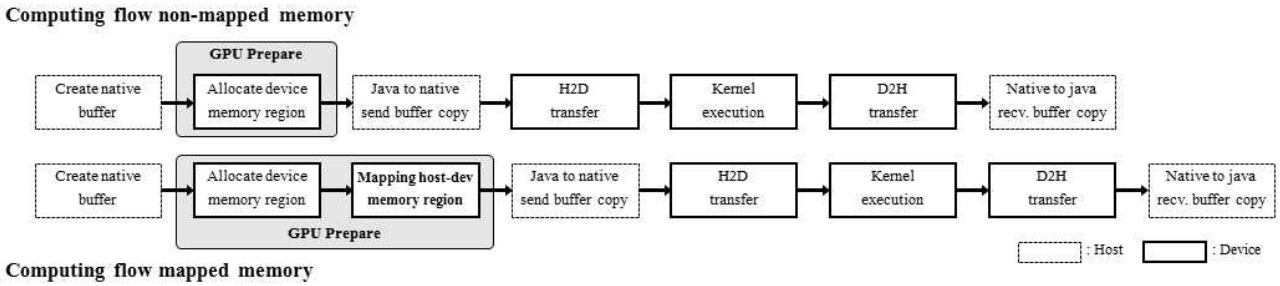


그림 2 OpenCL 기반 Spark 프레임워크 연산 흐름도
Fig. 2 Computing flow in Spark+OpenCL framework

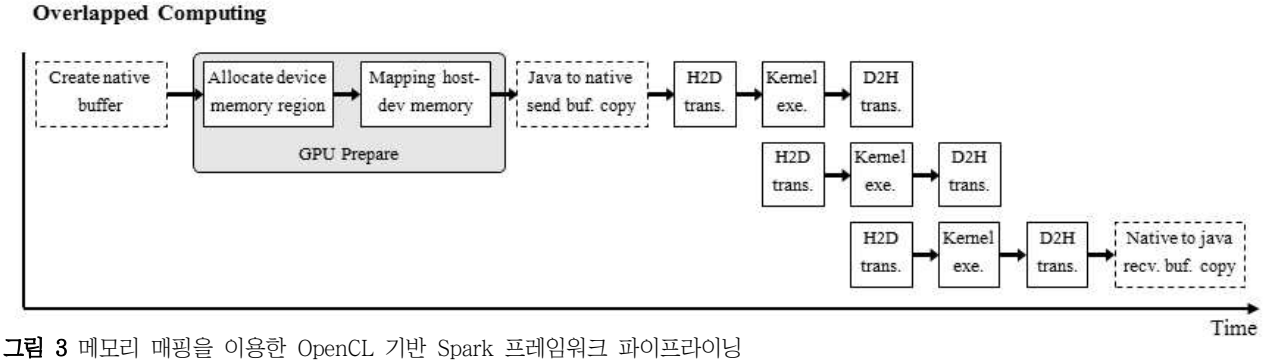


그림 3 메모리 매핑을 이용한 OpenCL 기반 Spark 프레임워크 파이프라이닝
Fig. 3 Pipelining Spark+OpenCL framework using a mapped memory

표 1 GPU 연산 준비 시간 비교
Table 1 The comparison of GPU prepare time

Size (million)	1	5	10	20	40
Non-mapped (ms)	1.6	3.6	6.6	13.8	25.3
Mapped (ms)	4.7	20.5	40.8	79.2	156.7

표 2 실험 환경
Table 2 The experimental environment

NVIDIA System		
OS		Ubuntu 14.04 LTS 64 bit
Spark Version		1.3.1
Host-Device Interface		PCIe 3.0 x16 (16GB/s)
Host	CPU	Intel i7-4790 3.60GHz
	Memory	16 GB
Device	GPU	GeForce GTX 960
	Memory	2 GB

때문에 태스크 마다 GPU를 호출하여 매번 GPU 연산 준비 과정이 필요하다.

일반적인 GPU 연산 성능 평가 방법과 달리 OpenCL 기반 Spark 프레임워크는 GPU 연산 준비 과정이 성능 평가 시 반드시 포함하게 된다. 표 1은 데이터 크기가 증가함에 따라 GPU 연산 준비 시간을 측정한 것이다. 메모리 매핑과정이 포함된 GPU 연산 준비 과정 시간이 처리할 데이터 크기가 점점 커질수록 메모리 매핑 과정이 포함되지 않은 GPU 연산 준비 과정 시간보다 선형적으로 더 크게 증가하는 것을 볼 수 있다.

3. GPGPU를 활용한 이기종 파이프라인 컴퓨팅 기반 Spark 프레임워크

3.1 파이프라인 기반 Spark 프레임워크

OpenCL 기반 Spark 프레임워크는 Spark 연산 중간에 다른 디바이스를 사용하기 때문에 호스트-디바이스 통신이 전체 연산 과정에 포함된다. 호스트-디바이스 통신 오버헤드가 데이터의 크기가 커질수록 전체 연산 성능에 큰 영향을 끼칠 수 있다. 이러한 호스트-디바이스 통신 오버헤드를 은닉시켜 성능을 향상시키기 위해 파이프라인 기법을 적용한다. 그림 3은 다중 스트림기반 파이프라인을 적용하여 계산과 통신이 분할되어 중첩 연산되는 것을 보여준다.

메모리를 매핑 하지 않고 진행하는 기존의 Spark+OpenCL 프레임워크와 달리 다중 스트림 기반에 파이프라인 기법을 구현하기 위해서는 Host-to-Device와 Device-to-Host에 있는 DMA (Direct Memory Access) 통신과 GPU 커널 계산 간의 중첩을 시켜야 하는데, 이때 그림 2의 메모리 매핑을 포함하는 계산 과정으로 처리를 해야 한다. 그러나 표 1과 같이 메모리 매핑 영역

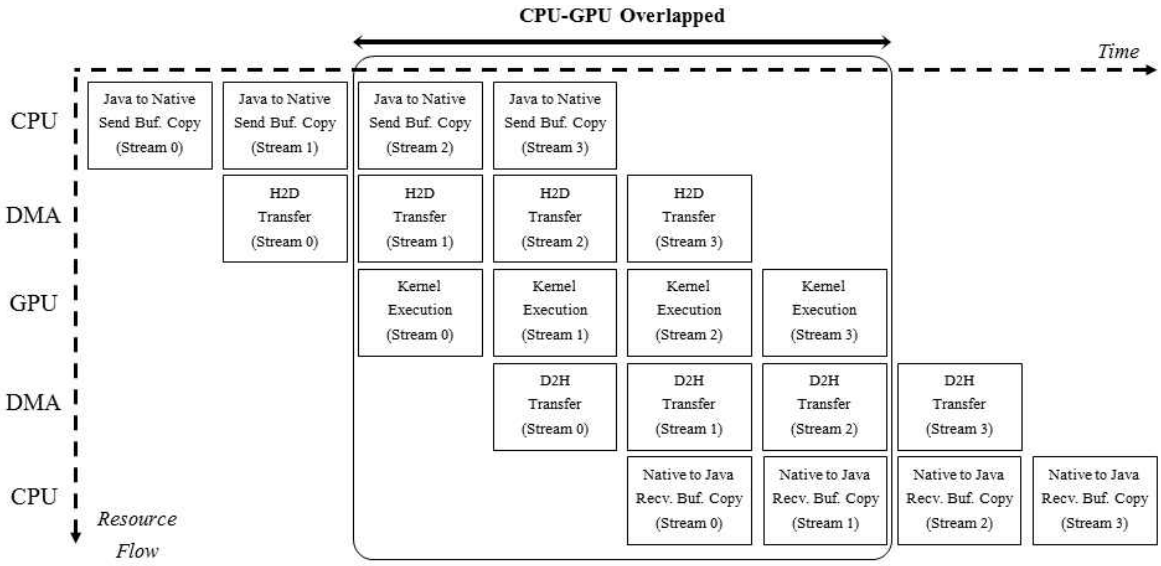


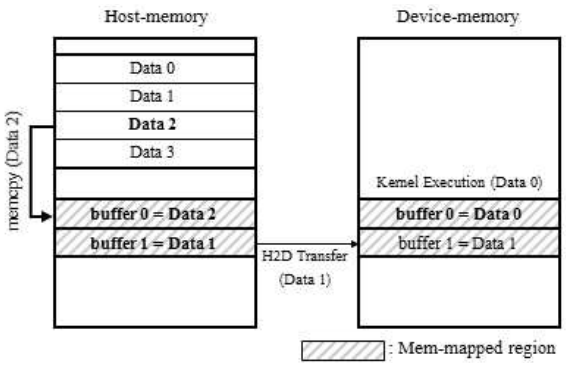
그림 4 CPU-GPU 이기종 컴퓨팅 기반 Spark 프레임워크
Fig. 4 The Spark+OpenCL framework based on CPU-GPU heterogenous computing

이 커질 경우 GPU 연산 준비 과정의 오버헤드가 증가하는 문제가 생기게 된다. 본 논문에서는 GPU 연산 준비 시간 오버헤드를 줄이고 효율적으로 파이프라인을 구현하기 위해 CPU-GPU 이기종 컴퓨팅(Heterogeneous Computing) 기반 파이프라인 기법을 제안하고자 한다.

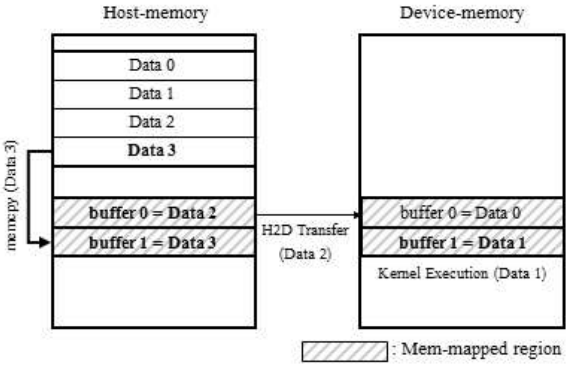
3.2 CPU-GPU 이기종 파이프라인 컴퓨팅 기반 Spark 프레임워크

Spark 프레임워크에서 GPU를 활용하기 위해서는 Java 영역에서 Native 영역으로 데이터를 복사하고 이를 DMA 통신으로 GPU로 전달하게 된다. 이 과정에 일반적인 파이프라인 기법을 적용하면 DMA 통신을 하고 GPU 커널을 연산할 때 CPU가 유휴 자원으로써 낭비 된다. 본 논문에서는 DMA 통신과 GPU 커널 연산이 중첩되는 동안 Java와 Native 영역 사이에 데이터 복사를 CPU가 지원하는 이기종 컴퓨팅을 제안한다. 그림 4는 제안한 CPU-GPU 이기종 컴퓨팅 기반 파이프라인 기법을 표현한 것으로 시간에 따른 자원 소모 흐름을 보이고 있다. 그림 4와 같이 Stream 3이 CPU에서 Java-to-Native 전송 버퍼 복사가 실행되는 중에, Stream 2는 H2D 전송(DMA), Stream 1은 Spark Kernel 연산(GPU), 그리고 Stream 0는 D2H 전송(DMA)이 중첩되어 CPU-GPU 간에 이기종 파이프라인 컴퓨팅 처리가 이루어지도록 되어 있다.

Spark+OpenCL 프레임워크에 파이프라인 기법을 적용하면 메모리 매핑 오버헤드가 증가하는 문제를 발견하였다. 본 논문에서는 DMA 통신을 위한 메모리 매핑 버퍼를 이중 버퍼로 구성된 스위칭 듀얼 버퍼로 구성하여 그 크기를 줄였다. 스위칭 듀얼 버퍼는 한 쪽 버퍼영역이 처리될 때 다른 쪽 버퍼의 데이터를 새로운 데이터로 갱신하여 버퍼를 재사용한다. 예를 들어, 그림 5는 4개의 다중 스트림을 이용하여 파이프라인을 할 때 스위칭 듀얼



(a) During H2D transfer using buffer1



(b) During H2D transfer using buffer0

그림 5 스위칭 듀얼 버퍼를 적용한 이기종 컴퓨팅 예제
Fig. 5 Example of an overlapped heterogeneous computing using the switching dual buffer

버퍼를 이용하여 데이터를 처리하는 것을 묘사한 것이다. 스위칭 듀얼 버퍼를 사용하면 2분의 1 크기의 매핑된 메모리 영역으로 데이터를 스위칭(Switching)하여 버퍼를 재사용하기 때문에 실제 메모리 매핑 영역이 기존에 비해 절반으로 줄어들어 GPU 연산 준비 오버헤드를 감소시킨다. 스위칭 듀얼 버퍼의 크기는 다음과 같이 구할 수 있다.

$$Switching\ Buffer\ size = 2 \times \frac{Data\ size}{No.\ of\ Streams} \quad (1)$$

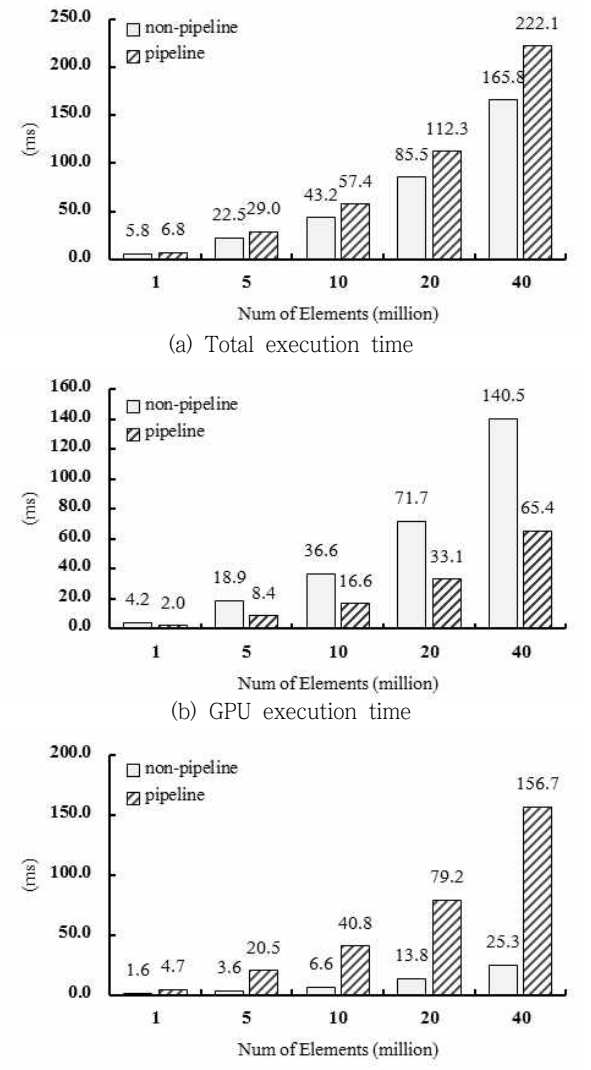
따라서 스트림의 개수를 증가시킬 경우 전달하는 데이터의 크기가 감소하고 스위칭 듀얼 버퍼의 크기도 줄게 되어 GPU 연산 준비 시간을 절약할 수 있으며 많은 통신 시간을 중첩을 시킬 수 있다. 또한 스위칭 듀얼 버퍼 기법은 매핑 영역을 재사용하기 때문에 디바이스 메모리 한계를 넘어 더 많은 데이터를 처리할 수 있는 장점도 있다.

4. 실험 및 고찰

본 실험은 성능 측정을 위해 Spark에서 제공하는 파이(Pi) 벤치마크 프로그램을 이용하였다. 파이를 계산하기 위한 데이터 세트의 크기는 4천만 개, 2천만 개, 1천만 개, 5백만 개, 1백만 개 등 다양한 크기로 성능을 측정하였다. Java-to-OpenCL 프레임워크 특성 상 AMD 시스템에서 빌드된 라이브러리를 이용하여 NVIDIA 시스템에서 실험을 진행하였다. 실험이 진행된 환경은 표 2와 같다.

그림 6은 기존 Spark+OpenCL 프레임워크와 파이프라인 기법을 적용한 Spark+OpenCL의 성능을 평가한 결과이다. 그림 6(a)는 데이터 세트 크기가 증가함에 따라 전체 실행 시간을 측정한 것으로 처리해야 할 데이터 세트 크기가 커짐에 따라 파이프라인 기법을 적용한 방법이 기존 방법과 비교하여 최대 34% 시간이 더 증가한 것을 볼 수 있다. 파이프라인 기법 적용 시 오히려 성능이 저하되는 원인을 찾기 위해 GPU 연산 준비 시간과 실행 시간을 나누어 측정해보았다. 그림 6(b)는 GPU 실행 시간만 측정한 것으로 처리해야 할 데이터 세트 크기가 커짐에 따라 실행 시간이 감소하여 오버헤드가 발생하지 않는 것을 알 수 있다. 그림 6(c)는 GPU 연산 준비 시간만 측정한 것으로 파이프라인 적용 기법이 기존에 비교하여 크게 증가하였다. 기존의 파이프라인을 적용하지 않은 Spark+OpenCL은 GPU 디바이스에 메모리 할당만 하고 호스트와 메모리 매핑 과정이 없어 더 빠르게 준비가 된다. 반면 파이프라인 기법을 적용 시 메모리 매핑 영역 크기가 증가함에 따라 매우 큰 폭으로 증가하는 것을 알 수 있다.

본 논문에서는 메모리 매핑으로 발생하는 GPU 연산 준비 오버헤드를 줄이기 위해 스위칭 듀얼 버퍼 기법을 제안하였다. 그림 7는 스위칭 듀얼 버퍼 기법을 적용하였을 때 GPU 연산 준비 시간과 실행 시간을 나타낸다. 처리할 데이터 세트 크기가 4천만 개 일 때 스트림 개수에 따른 GPU 성능 변화를 보여준다. 스트림 개수가 2개일 때는 단순 파이프라인 기법과 동일하게 처리되므로 GPU 연산 준비시간이 크게 증가한 것을 볼 수 있다. 하지



(a) Total execution time

(b) GPU execution time

(c) GPU prepare time

그림 6 데이터 크기에 따른 Spark GPU 실행 시간

Fig. 6 The spark GPU times according to the various data sets

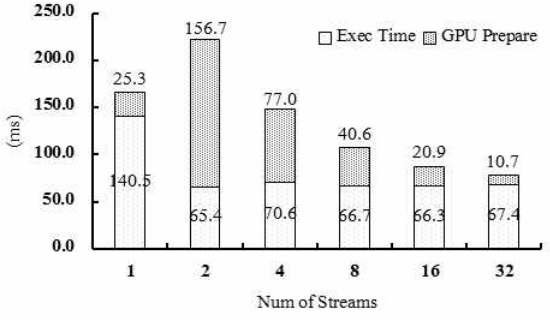


그림 7 이기종 파이프라인 적용 총 GPU 시간

Fig. 7 GPU prepare + execution time of Spark+OpenCL based on heterogeneous pipeline computing

만 스트림 개수가 늘어날수록 메모리 매핑 영역이 감소하여 32개 스트림을 이용할 경우 GPU 연산 준비 시간이 최대 58% 줄어드는 것을 볼 수 있다. 따라서 32개의 스트림을 이용할 경우 기존에 비해 2.13배 그리고 단순 파이프라인 기법을 적용하였을 때 보다 2.84배의 성능 향상을 보였다.

5. 결 론

현재 OpenCL 기반 Spark 프레임워크는 GPU를 활용하는 것이 매우 간편한 장점이 있다. 하지만 내부 연산과정을 살펴보면 Spark의 Java와 Native 메모리 영역간의 데이터 전송과 호스트와 GPU 사이의 데이터 전송 및 GPU 처리가 일련의 과정으로 진행된다. 따라서 계산-통신 파이프라인 기법을 적용하여 성능을 개선시키고자 하였다. 하지만 호스트와 디바이스 사이의 DMA 통신을 위한 메모리 매핑과정에 오버헤드가 발생하여 오히려 성능이 저하되는 현상이 발견되었다. 일반적인 GPGPU 연산은 메모리 매핑과정이 GPU 연산 준비 과정에 포함되지만 성능 측정 시 포함되지 않는다. 그러나 OpenCL 기반 Spark 프레임워크는 Spark 연산 중간에 GPU를 활용하기 때문에 태스크를 처리할 때마다 GPU 연산 준비 과정이 필요하여 메모리 매핑 오버헤드는 전체 성능에 큰 영향을 끼친다. 또한 호스트 CPU는 호스트와 GPU 사이의 데이터 전송 및 GPU 연산 기간에는 유휴자원으로 낭비가 되었다. 본 논문에서는 CPU 유휴시간을 줄이기 위하여 CPU에서 처리되는 Java와 Native 메모리 영역간의 전송을 DMA 통신과 GPU 연산과 함께 중첩 처리하는 이기종 파이프라인 기법을 제안한다. 또한 DMA통신 버퍼를 스위칭 듀얼 버퍼 기법으로 구현하여 버퍼를 재사용하고 메모리 매핑 크기를 줄여 메모리 매핑 시간을 감소시켰다. 제안한 OpenCL을 활용한 이기종 파이프라인 기반 Spark 프레임워크의 성능을 평가하기 위하여 Spark 파이(Pi) 벤치마크를 이용하였고 다양한 데이터 세트 크기로 실험을 진행하였다. 실험 결과 처리할 데이터 세트 크기가 4천만 개일 때 스트림 개수에 따라 다른 성능 변화를 보였다. 스트림 개수가 증가할수록 메모리 매핑 영역이 감소하여 32개 스트림을 이용할 경우 GPU 연산 준비 시간이 57.7% 감소하고 기존 OpenCL 기반 Spark 프레임워크에 비해 최대 2.13배, 단순 파이프라인 기법을 적용하였을 때 보다 2.84 배 성능 향상되었음을 확인하였다.

감사의 글

이 논문은 2017년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. 2017R1D1A1B03033128).

References

[1] Jeffrey Dean and Sanjay Ghemawat, “MapReduce: simplified data processing on large clusters,” Communications of the ACM. 51(1). pp. 107-133. 2008.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. pp. 2-2. 2012.

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, “Spark: cluster computing with working sets,” Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. pp.10-10. 2010.

[4] Tudor Alexandru Voicu, “SparkJNI: A Reference Design for a Heterogeneous Apache Spark Framework,” M.S. Thesis, the Delft University of Technology, 2016.

[5] Oren Segal, Pilip Colangelo, Nasibeh Nasiri, Zhuo Qian and Martin Margala, “SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters,” arXiv preprint arXiv:1505. 01120, 2015.

[6] Oren Segal, Pilip Colangelo, Nasibeh Nasiri, Zhuo Qian and Martin Margala, “Aparapi-Ucores: Ahigh level programming framework for unconventional cores,” High Performance Extreme Computing Conference(HPEC). pp. 1-6. 2015.

[7] Diego Caballero, Sara Royuela and Roger Ferrer, “Optimizing Overlapped Memory Accesses in User-directed Vectorization,” In Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 393-404. 2015.

[8] Toshiya Komoda, Shinobu Miwa and Hiroshi Nakamura, “Communication Library to Overlap Computation and Communication for OpenCL Application,” In Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW). pp. 567-573. 2012.

[9] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker and Byung-Gon Chun, “Making sense of performance in data analytics frameworks,” 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp.293-307. 2015.

[10] Naila Farooqui, “Runtime specialization for heterogeneous CPU-GPU platforms,” Ph.D. Dissertation, the Georgia Institute of Technology, 2016.

[11] Max Grossman, Shams Imam and Vivek Sarkar, “HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators,” Proceedings of the Principles and Practices of Programming on The Java Platform. pp. 2-15. 2015.

[12] NVIDIA, “CUDA C PROGRAMMING GUIDE.“ Technical

Report, September 2015.
[13] AMD, “AMD APP SDK OpenCLTM User Guide.“ Technical Report, Advanced Micro Devices(AMD), August 2015.
[14] AMD, “AMD APP SDK OpenCLTM Optimization Guide.” Technical Report, Advanced Micro Devices (AMD), August 2015.

저 자 소 개

김 대 희 (Daehee Kim)

2015년 건국대학교 컴퓨터공학과(학사)
2017년 건국대학교 컴퓨터공학과(석사)
2017년~현재 건국대학교 컴퓨터공학과 (박사과정)
관심분야 : GPGPU, OpenCL, HPC 등

박 능 수(Neungsoo Park)

1991년 연세대학교 전기공학과(학사)
1993년 연세대학교 대학원 전기공학과 (석사)
2002년 미국 University of Southern California 전기공학과(공학박사)
2002년~2003년 삼성전자 책임연구원
2003년~현재 건국대학교 컴퓨터공학과 교수
관심분야 : 컴퓨터구조, 임베디드 시스템, 병렬시스템, HPC, GPGPU 컴퓨팅, 빅-데이터 처리, 멀티미디어 컴퓨팅 등