Journal of Korea Multimedia Society Vol. 17, No. 10, October 2014(pp. 1189-1197)
http://dx.doi.org/10.9717/kmms.2014.17.10.1189

데이터 플로우 기반 응용들을 위한 GPU 스케줄링 프레임워크

이용빈[†], 김성찬[††]

A GPU scheduling framework for applications based on dataflow specification

Yongbin Lee[†], Sungchan Kim[††]

ABSTRACT

Recently, general purpose graphic processing units(GPUs) are being widely used in mobile embedded systems such as smart phone and tablet PCs. Because of architectural limitations of mobile GPGPUs, only a single program is allowed to occupy a GPU at a time in a non-preemptive way. As a result, it is difficult to meet performance requirements of applications such as frame rate or response time if applications running on a GPU are not scheduled properly. To tackle this difficulty, we propose to specify applications using synchronous data flow model of computation such that applications are formed with edges and nodes. Then nodes of applications are scheduled onto a GPU unlike conventional scheduling an application as a whole. This approach allows applications to share a GPU at a finer granularity, node (or task)-level, providing several benefits such as eliminating need for manually partitioning applications and better GPU utilization. Furthermore, any scheduling policy can be applied in response to the characteristics of applications.

Key words: GPGPU, Scheduling, Synchronous Data Flow

1. 서 론

최근 산업의 발전에 따른 대규모의 문제 해결의 요구는 커져갔고 이에 따라 CPU(Central Processing Unit)와 주변기기의 성능은 급속도로 성장하였다. 그러나 CPU의 성능개선은 한계에 부딪혔고 이를 극복하려는 방안으로 멀티코어 GPU(Graphic Processing Unit)를 이용한 GPGPU(General-Purpose computing on Graphics Processing Units) 기반의 병렬처리 연산이 사용되고 있으며, 최근에는 범용 작업을 위한 다수의 GPU를 결합한 클러스터 시스템에 대한 연구까지 진행되고 있다[1].

CPU에서는 여러 응용의 작업에 대해 선점형 커널을 기반으로 한 문맥 교환(context switching)을 통한 작업 분배가 가능하며 최근에 출시되는 GPU에서는 동적 병렬 처리가 가능해 작업의 스케줄링을 가능케 한다. 하지만 스마트폰이나 태블릿 등의 모바일 시스템에서는 GPU의 구조적 한계로 인해 한 번에 하나의 프로그램(또는 커널)을 비선점형(non-preemptive) 방식으로 실행할 수밖에 없다. 이러한 한계

※ Corresponding Author : Sungchan Kim, Address: (561-756) 567 Baekje-daero, deokjin-gu, Jeonju-si, Korea, TEL : +82-63-270-2411, FAX : +82-63-270-2394, E-mail : sungchan.kim@chonbuk.ac.kr
Receipt date : July 14, 2014, Revision date : Sep. 12, 2014
Approval date : Sep. 14, 2014

[†] Division of Computer Science and Engineering, Chonbuk National University
(E-mail : forever1363@chonbuk.ac.kr)
[††] Division of Computer Science and Engineering, Chonbuk National University

를 가진 모바일 환경에서 만약 실시간 처리를 요구하는 영상출력과 같은 그래픽 응용과 특정 연산 처리를 요구하는 계산 응용이 같이 실행될 경우 계산 응용이 GPU를 장시간 사용하게 될 수 있다. 이에 따라 그래픽 응용의 실시간 처리가 지연되어 프레임률(frame rate) 손실이 발생할 수 있다.

이렇게 다수의 응용들이 GPU를 동시에 사용할 경우 발생하는 비선점형 방식의 모바일 시스템의 한계를 극복하기 위해 각 응용의 작업을 분할하여 처리하는 방식을 생각하였다. 분할작업이 이루어질 경우 응용마다 상황정보(context)가 달라 이를 유지하는데 어려움이 있다. 이를 해결하기 위한 방법으로 각 작업에 대한 데이터 플로우를 그래프를 통해 명시적으로 표현하여 처리할 수 있는 방법을 생각하였고 그중 입력(input)과 출력(output)이 정해지면 노드단위로 표현 및 구현이 가능한 SDF(Synchronous Data Flow) 그래프를 채택하여 사용하였다. 각 응용의 작업들을 SDF 그래프에 맞게 노드단위로 분할하여 처리하는 방식이다. 즉, 하나의 노드의 작업이 완료되면 다른 응용에 대한 노드가 실행 될 수 있어 선점형 스케줄링을 가능케 하는 것 이다. 우리는 이 스케줄링 기법을 이용해 실시간 처리방식인 영상출력 응용과 벡터덧셈 응용을 동시에 실행하도록 하여 수행능력의 장점을 실험하였다.

본 논문의 구성은 다음과 같다. 2장에서 GPU 스케줄링에 대한 기존 연구에 대해 살펴보고 한계가 무엇인지에 대해 소개할 것이다. 3장에서는 GPGPU와 제시하는 스케줄링의 근간이 되는 SDF 그래프에 대해 살펴볼 것이고, 4장에서는 모바일 플랫폼 GPU의 비선점형 방식의 한계를 극복 할 수 있는 SDF 그래프를 이용한 스케줄링 기법을 소개할 것이다. 5장에서는 제안한 스케줄링 기법을 통해 응용들이 동작할 때 어떠한 효과를 얻을 수 있는지 실험을 통해 살펴보고 6장에서 결론을 맺는다.

2. 관련 연구

최근 병렬 처리가 각광받음에 따라 이를 효과적으로 운용할 수 있는 기법에 대한 연구들이 활발하게 진행되고 있다. 연구들은 크게 두 가지로 분류되는데, 특정 응용을 대상으로 한 최적화에 대한 것이 첫 번째이고, 두 번째는 CPU-GPU 이기종 플랫폼에서 범용 스케줄링 기법에 관한 연구들이다. 첫 번째 분류의 연구로서, [2]에서는 트리 검색을 CPU-GPU 플랫폼에서 병렬화하여 이기종 컴퓨팅 자원들을 효율적으로 사용할 수 있는 동적 로드 밸런싱 기법을 제시하였다. 검색 처리율을 최대화하기 위해 GPU에서 수행되는 쓰레드들에 처리할 데이터를 균일하게 분배하였다. [3]에서는 GPU가 설치된 하둡 맵리듀스 플랫폼에서 자연언어처리 분야에서 사용되는 CYK 파서응용을 수행하기 위해 CPU-GPU 의 정적/동적 스케줄링 기법을 제안하였다. 앞에서 언급된 연구들은 특정 응용 분야에 대한 최적화만을 다루고 있어 다양한 응용들에 범용으로 사용할 수 없는 한계점을 가진다.

다음은 두 번째 분류인 범용 스케줄링을 위한 플랫폼 관련 연구들을 살펴본다. 먼저 XKaapi는 다중 CPU와 다중 GPU 구조에서 데이터 플로우 프로그래밍을 이용한 런타임 시스템으로서, 다중 CPU와 다중 GPU를 이용해 각 쓰레드들을 최대로 최적화하여 사용하기 위한 기법으로 데이터 플로우에 따라 태스크를 나누고 나눠진 태스크를 쓰레드를 통해 처리한다[4]. 이때 여러 쓰레드가 하나의 워크아이템에 접근하여 발생할 수 있는 경쟁 조건(race condition)발생 방지를 위해 각 프로세서마다 큐(queue)를 갖고 각각의 큐에 의해 쓰레드가 관리되도록 하였다. 또한 이에 수반되어 나타나는 쓰레드 사용에 있어서의 불균형을 해결할 방안으로 워크 스틸링 알고리즘(work stealing algorithm)을 사용하였다[5].

태스크 분할을 위해서는 동적 스케줄링 기법인 DDF(Dynamic Data Flow) 그래프를 사용하였다. 이 동적 스케줄링 기법은 런타임 시 각 시스템의 로드 밸런싱(load balancing)을 고려해 태스크를 각 프로세서로 분배하는 것이다. 이러한 런타임 시스템은 응용 소프트웨어의 특성 및 시스템 상황에 따른 최적의 실행 환경을 제공하여 성능과 효율성을 극대화할 수 있게 한다[6]. 하지만 이러한 런타임 시스템의 경우 로드 불균형이나 스케줄링 오버헤드가 커질 수 있으며 여전히 두 개 이상의 응용에 대한 병렬 처리에 있어서는 각 태스크마다의 구분이 명확하지 못해 선점형 방식으로 수행되기는 어렵다.

[7]에서 제안된 KernelMerge는 GPU에서 여러 개의 커널을 동시에 동작시킬 수 있도록 만드는 소프트웨어 스케줄러이다. 기존 응용의 코드의 구조를 바꿀

필요 없이 C++코드와 OpenCL API를 이용해 GPU 자원을 나누어 할당함으로 두 개 이상의 커널을 동시에 사용하도록 하는 것이 핵심이다. 이 기술은 두 개 이상의 커널을 같이 실행하도록 할 수 있지만 자원을 분배하여 사용하다보니 각 커널의 수행속도는 느려지는 단점이 있다. 또한 단순히 자원을 할당하여 분배하는 방식이기 때문에 두 개 이상의 서로 다른 응용에 대한 스케줄링은 불가능하다.

[3]에서는 데이터플로우 그래프로 기술된 응용들을 정적 분석을 통해 버퍼 사용을 최소화하기 위한 데이터플로우 그래프 변환 기법을 제시하였다. [3]의 연구는 데이터플로우 기반의 응용들을 대상으로 한다는 점에서 본 연구와 유사하지만 몇 가지 차이점이 있다. 첫째, 본 연구는 다중 응용들이 하나의 GPU를 공유하는 것들 가정하는 반면 [3]은 하나의 응용을 대상으로 한다. 둘째, 본 연구에서는 응용의 지연시간 최적화를 대상으로 삼는 반면 [3]에서는 버퍼 사용량 최소화를 목적으로 한다.

3. 배경 지식

3.1 GPGPU

GPU 구조에서는 Fig. 1처럼 GPU 내부 요소에 여러 계층이 존재한다. 먼저 SM(Streaming Multiprocessor)이라 불리는 프로세서가 GPU에 따라 16-32개씩 들어있고, 각각의 SM은 내부에 SP(Streaming Processor)라 불리는 코어가 32개씩 들어있다[8]. 이 SP는 실제로 쓰레드가 실행되는 최하단의 위치이며 쿠다 코어라고도 불린다. 모든 쓰레드는 각각의 SP에 할당되어 실행되며 각각의 SP는 한 번에 동시 최대 4개의 쓰레드까지 처리가 가능하다. SP의 상위

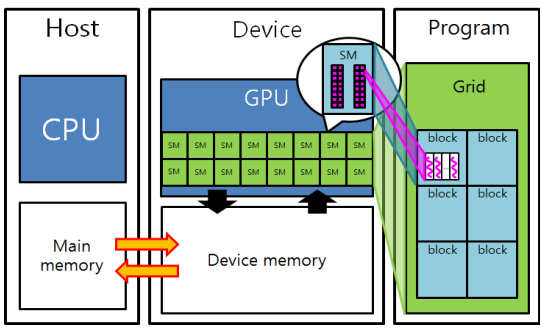


Fig. 1. GPGPU Structure.

계층인 SM은 쓰레드의 소규모 그룹인 쓰레드 블록이 실제 할당되는 하드웨어이다. 쓰레드 블록은 프로그래머가 생성한 전체 쓰레드들 중 일정한 쓰레드 크기 단위로 나누어 놓은 단위이며 이 쓰레드 블록이 모여 전체 쓰레드들을 가지고 있는 그리드를 이룬다. 각각의 블록들은 데이터를 공유하지 않고 개별적으로 동작한다. 그리드와 블록의 크기는 런타임 시 프로그래머가 지정해준 크기대로 결정되며 실험에 GPU에 따라 블록 하나에 할당할 수 있는 최대 쓰레드 개수가 정해져 있다. 블록은 GPU의 SM에 할당되어 실행되며 어떤 블록이 어떤 SM에서 어느 타이밍에 실행되는지는 알 수 없으며 GPU를 포함하고 있는 디바이스와 CPU를 포함하고 있는 호스트는 또한 각각의 메모리가 따로 존재한다.

3.2 Synchronous Data Flow (SDF)

SDF 그래프는 고정된 크기의 입력과 출력만 정해지면 노드단위의 그래프로 응용을 표현 할 수 있다[9]. 이는 데이터 플로우를 표현하기에 적합하며 각 노드별로 독립된 작업이 진행되기 때문에 해당 노드의 입력 값이 정해짐에 따라 독립적으로 수행 될 수 있다. SDF를 이용한 응용 명세의 장점 중의 하나는 설계 제약 조건을 만족하면서 설계 목적을 최적화하기 위한 노드 실행 스케줄을 정적으로 계산할 수 있다는 것이다. 이러한 정적 스케줄링 기법을 통해 그래프의 교착 상태 및 버퍼 오버플로우 등의 명세 단계에서의 오류를 미리 검출할 수 있는데, 이는 임베디드 시스템을 개발하는데 매우 유용한 기능이다[10].

간선마다 노드와 노드간의 입력 데이터의 개수와 출력 데이터의 개수가 정해져 있으며 각 노드는 입력 데이터가 모두 충족될 때 비로소 실행될 수 있다. 입력 데이터와 출력 데이터는 작업이 해당 노드를 떠나는 순간 더 이상 필요로 하지 않는다.

Fig. 2는 SDF 그래프의 한 예를 보여준다. A, B, C는 노드(태스크)를 의미하며 각 노드들은 간선으로 연결되어 있다. 간선 위의 숫자는 출력 노드가 수행되었을 때의 생성되거나 입력 노드가 수행될 때 소모

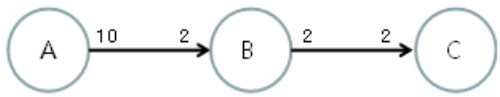


Fig. 2. An example of an SDF Graph.

하는 샘플을 개수를 나타낸다. Fig. 2의 경우 A 노드 수행 시 10개의 샘플이 생성되고 B노드의 경우 입력으로 들어오는 10개의 샘플 중 2개의 샘플을 받아 수행함을 알 수 있다. 이를 통해 A노드 한번 처리 후 B노드와 C노드를 번갈아가며 실행시키는 A5(BC) 또는 A노드 한번 처리 후 B노드에 대한 작업을 연속으로 처리 한 후 C노드의 작업을 수행하는 A5B5C와 같이 노드 간 스케줄링 또한 가능하다. 이때 작업의 특성에 맞게 스케줄링이 되어야 하는데 먼저 노드 사이에 출력 데이터를 저장할 버퍼 메모리가 충분하지 않는다면 두 번째 방법은 불가능 할 것이다. 또한 해당 작업이 실시간 처리를 요구하는 작업이라면 실시간으로 작업 결과가 나오는 첫 번째 방법을 사용해야 할 것이다. 반면 두 번째 방법과 같이 버퍼 메모리에 데이터를 저장해놨다가 한번에 처리하는 방식은 캐시 메모리 적중률(cache memory hit rate)을 높여 빠른 처리가 가능하게 된다. 이와 같이 각 작업의 특성을 고려한 스케줄링이 필요하다.

SDF 그래프에서는 정적 태스크 스케줄링을 사용하여 응용이 실행되기 전에 태스크를 각 프로세서로 분배하며 각 노드의 실행 순서가 컴파일 단계에서 결정된다. 이는 제한된 메모리 요구량을 보존하고 실행 중 필요한 노드에 관한 부가적인 처리를 컴파일 과정에서 수행하여 실행시간을 단축시킬 수 있다.

3.3 문제 정의

본 논문에서의 주요 논제는 선점형 방식의 실행이 불가능한 모바일 환경의 GPU 병렬 처리의 한계에 대한 해결 방안이다. 이를 위해 제시하는 방법은 각 응용을 SDF 그래프로 나누어 스케줄링 하는 것이고, 이를 위해서는 개발자가 사전에 해당 응용을 노드단위로 분할하는 작업이 필요하다. Fig. 1과 같이 CPU 측과 GPU측은 서로 다른 메모리를 가지고 있다. 스케줄러는 CPU측에서 동작하고 계산수행은 GPU에서 동작하기 때문에 두 장치 간 데이터를 서로 전송함으로써 데이터 공유가 이루어진다.

개발자는 SDF 그래프의 형식에 맞게 응용을 구현하여 해당 SDF 그래프 정보를 스케줄러에게 넘겨주어야 한다. 스케줄러에서는 해당 응용의 정보를 저장해 뒀다가 필요시 사용하여 스케줄링을 하여야 한다. 이를 통해 두 개 이상의 응용에 대한 스케줄링이 가능한 시스템 구축이 본 논문의 목표이다.

4. GPU 스케줄링 프레임워크

4.1 스케줄링 프레임워크 알고리즘

기본적으로 각 응용들은 SDF 그래프의 형식에 맞게 노드 단위로 설계 및 구현이 되어 있어야 한다. Fig. 3에서 보듯이 스케줄러에게 정보를 제공하기 위

```
struct Node:
    integer step, input_size, output_size
    character input_type, output_type
procedure Information( ):
    index, total_size, step, characteristic
    Node node_list[step];
```

(a)

```
struct Node:
    integer step, input_size, output_size
    character input_type, output_type
struct Job:
    integer index, total_size, step
    boolean characteristic
    Node node_list[step]
procedure Add_job(size, sdf_information, characteristic):
    if characteristic of job is 1:
        add job to the front of the job_list
    else:
        add job to the back of the job_list
procedure Scheduler(type, index, step):
    deque job_list
    process next node of the job
    update index, step of the job
    if job is finished: remove the job from the job list
```

(b)

Fig. 3. (a) Pseudocode of the Information function and (b) pseudocode of the scheduler.

해 Infomation함수를 별도로 생성하여 SDF 그래프의 노드의 총 개수와 각 노드별 입력과 출력 데이터의 유형과 크기가 정해져 있어야 한다. 입력과 출력 데이터의 유형의 경우 문자 하나로 나타낸다(s=string, i=integer, b=boolean, c=character, f=floating point).

구현된 스케줄러는 CPU에 의해 제어된다. 스케줄러는 각 응용의 Information함수를 통해 응용을 구현하는데 필요한 정보들을 읽어 들인다. 스케줄러 내에서는 동작시킬 응용들의 SDF와 작업 중인 응용이 노드단위로 어느 단계까지 왔는지를 파악하고 있어야 한다. 스케줄러에서 제어하는 응용은 노드단위로 처리될 때마다 스케줄러에 노드의 결과 값과 현재 상태에 대한 정보를 반환해 준다. 반환된 값을 받은 스케줄러는 시스템의 상황을 고려하여 다음 노드의 입력 값이 만족될 경우 다음 노드에 대한 실행 명령을 내린다.

각 응용마다의 특성을 고려한 스케줄링을 위해서는 개발자가 해당 응용이 어떠한 특성을 가지는지를 명시해줘야 한다. 본 논문에서는 실시간처리인 경우와 그렇지 않은 경우 두 가지만을 위한 스케줄링 기법을 다룬다. 개발자는 응용을 SDF 형식으로 구현 시 실시간 처리를 필요로 하는 응용의 경우 1을, 그렇지 않은 경우 0을 반환하는 Get_characteristic() 함수를 구현한다. 스케줄러는 처음 응용이 SDF 형식으로 들어올 때 이 함수를 통해 해당 응용이 어떠한 특성을 가진지 판별할 수 있다. 판별된 응용이 0(실시간 처리 불필요)이라면 해당 응용은 무조건 빨리 처리하면 되는 최선 노력(best effort)으로 처리하고, 1(실시간 처리 필요)이라면 해당 응용은 요구되는 실시간 처리량을 만족시켜야 할 것이다.

Fig. 4는 스케줄러에 대한 의사코드(pseudocode)이다. 기본적으로 하나의 응용에 대한 정보를 Job 구조체를 통해 저장시킨 후 해당 응용을 노드 단위로 job_list에 추가시킨다. 이때 응용의 특성이 실시간 처리 방식인 경우 job_list의 선두에 추가하여 빠른 처리가 가능하도록 만든다. 응용의 구조체 Job은 진행정도 확인을 위한 index와 총 처리해야 할 데이터 수를 나타내는 total_size, SDF 노드의 단계를 나타내는 step, 응용의 특성인 characteristic 변수, 그리고 SDF 노드들을 갖는다. 스케줄러는 job_list의 응용들을 노드 단위로 실행해주며 실행 후 해당 응용의 index와 step을 변경함으로써 진행 상황을 업데이트 해준다. 이때 응용은 노드 단위로 실행을 결정해주기 때문에 스케줄러는 하나의 응용이 최종적으로 끝날 때까지 기다리지 않아도 된다.

Fig. 4는 단순히 라운드 로빈 방식의 스케줄링을 응용하였다. 각 응용마다의 특성 및 중요도를 고려한 스케줄러로 확장할 수 있음을 생각해볼 수 있다.

4.2 영상출력과 벡터덧셈 응용에 대한 GPU 스케줄링

Fig. 4는 두 가지 응용 A1(영상출력), A2(벡터덧셈)이 GPU에서 실행될 경우 스케줄링 되는 예를 보여주고 있다. 각 응용의 작업에 대해 입력과 출력이 분명한 부분을 노드단위로 분할하여 SDF 그래프로 표현하였다. A1은 영상출력 작업을 A(입력), B(연산), C(출력) 세 단계를 거쳐 수행한다고 가정하고 SDF 그래프로 표현한 것으로 세 노드는 모두 입력 샘플 하나를 필요로 하고 출력샘플 하나를 출력한다. A2는 벡터 덧셈으로 한번에 10개의 원소씩 처리하는 경우로 가정하였다. 두 개의 벡터에서 10개의 원소씩 넘겨주는 A, B노드, 실제 덧셈 연산을 수행하는 ADD노드, 마지막으로 결과 값을 병합하는 일을 수행하는 C노드로 이루어져 있다. 이 경우 한번에 10개씩의 원소만 처리하기 때문에 총 원소의 개수에 맞추어 반복된 작업이 필요하다.

(a)에서는 실시간 일괄 처리 방식(real-time batch processing)인 영상출력 응용 A1이 실행되고 있을 때 벡터덧셈 응용인 A2가 실행되는 모습을 보여준다. 영상출력 작업이 명령으로 들어와 있는 상태라 한다면 스케줄러는 작업 명령이 들어오는 대로 실시간으로 작업을 처리할 것이다. 스케줄러는 A1 응용을 노드단위로 실행시키다 중간에 A2 응용이 들어왔을 때 A2 응용으로 전환하여 작업을 처리해준다. 이때 각 노드는 작업 완료 후 완료했음을 스케줄러에 반환해주는 작업이 필요하다. 스케줄러는 반환된 결과와 기존에 저장된 각 응용의 실행순서(execution sequence)를 비교하여 작업을 스케줄링 한다. 이렇게 각 노드는 작업이 종료되고 반환될 때 다음 노드에서 필요로 하는 데이터를 함께 반환하며, 스케줄러는 해당 데이터를 메모리에 저장 시킨 후 다음 작업 명령을 내릴 때 필요 데이터를 입력으로 보내주어 작업을 수행시킨다. Fig. 4의 작업 순서(working order)를 보면 응용 A1작업을 진행하다 블록 시키고

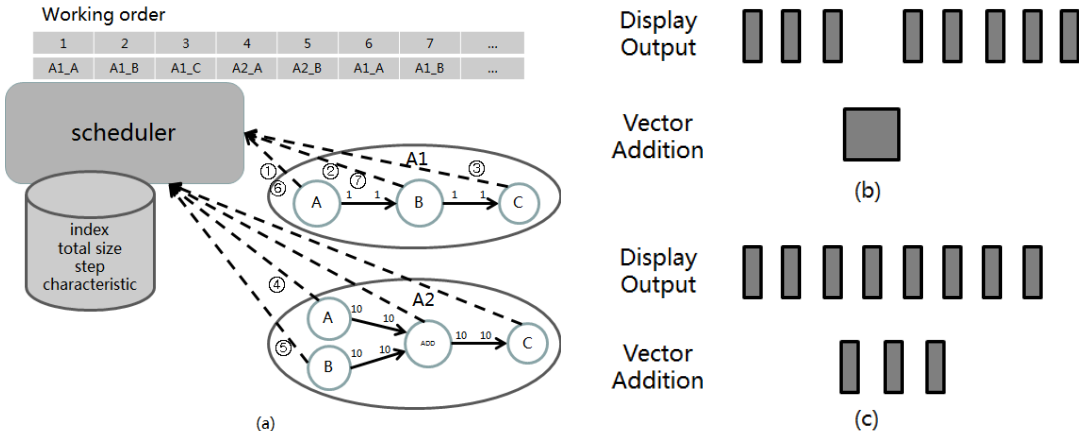


Fig. 4. (a) Scheduling display output, vector addition application based on SDF graph, (b) Scheduled display output, vector addition by application program, (c) Scheduled display output, vector addition by graph node.

A2를 실행 시킨다. 그 후 A1에 대한 작업 요청이 들어오면 A2작업은 마무리되지 못하고 중간에 블록 되고 작업은 A1으로 전환된다. 이렇게 응용 프로그램 작업을 분할함으로서 선점형 방식 수행이 가능하게 된다.

Fig. 4의 (b)는 응용프로그램 단위로 스케줄링 된 영상출력과 벡터덧셈의 작업 흐름을 보여주고 있다. 영상출력이 진행되는 도중 벡터덧셈이 수행되고 이 때 벡터덧셈이 진행되는 동안 영상출력은 블록 되어 프레임률 손실이 발생한다. 반면 그래프노드 단위로 스케줄링 된 (c)의 경우 작업이 분할되어 스케줄링 되기 때문에 두 응용이 필요에 따라 교차로 수행되고 각각의 처리에 관한 응답시간은 감소하여 프레임률 손실을 막는 효율적인 스케줄링이 가능해진다.

본 논문에서 소개한 분할처리 벡터덧셈의 경우 한 번의 연산에 10개의 원소를 처리하였다. 이는 고정된 크기가 아닌 확장 또는 축소가 가능하다. 확장을 통해서는 GPU의 병렬성을 증대시킬 수 있지만 작업 데이터 크기가 커짐에 따라 세밀한 스케줄링은 어렵게 된다. 축소는 반대로 더욱 세밀한 스케줄링을 가능케 하지만 병렬성은 그에 비례하게 떨어지는 단점이 있다. 이렇게 분할하는 단위를 달리함에 따라 시스템 상황에 최적화된 스케줄링이 가능할 수 있음을 알 수 있다.

5. 실험 결과 및 고찰

실험은 Intel(R) Core(TM) i7, 16GB Memory, NVIDIA GeForce GTX 580, Ubuntu 12.04.4 LTS 64Bit 으로 구성된 리눅스 서버에서 수행되었다. OpenCL로 구현된 벡터덧셈과 영상출력 응용을 사용하였고 호스트인 CPU에서 스케줄러가 동작하며 실제 응용에 대한 동작은 디바이스인 GPU에서 동작하도록 하였다. 호스트의 스케줄러에 의해 명령을 받은 디바이스에서 명령 받은 응용에 대한 노드를 실행시키는 구조이다. 3장에서 설명한대로 벡터덧셈은 A, B, ADD, C 네 단계의 노드로 나누었고, 영상출력은 A, B, C 세 단계로 나누어 1초에 24프레임을 처리하는 영상으로 가정하여 약 40(ms)의 시간을 간격으로 프레임 단위 영상처리 요청을 하도록 구현하였다.

Fig. 5는 실험 응용인 벡터덧셈과 영상출력이 어떠한 방법으로 실행되는지를 나타낸 것이다. SDF 그래프 형식으로 구현된 응용이 명령으로 들어오면 스케줄러에서 관리하는 덱(Deque : double-ended queue)에 작업이 노드 단위로 저장된다. 들어온 순서대로 처리하는 라운드 로빈 방식으로 진행하되 중간에 영상출력 작업이 요청이 오면 덱의 제일 앞에 영상출력 작업을 삽입하여 먼저 처리하도록 조정한다. 그 후 OpenCL 실행시간 라이브러리를 이용해 GPU로 실행시킬 작업(Kernel program)을 호스트 프로그램과 함께 컴파일 한 후 실행한다. 이때 커널 실행에 대한 명령을 줄 때 스케줄러는 해당 노드의 정보를 함께 넘겨주어 연산 수행 범위와 단계를 구별해 준다. 스케줄러는 한 노드의 작업이 종료되면 해당 응용의 데이터들을 업데이트 해주어 다음 노드의 작업 수행에 문제가 없도록 한다. 이후 덱에 저장된 다음 노드

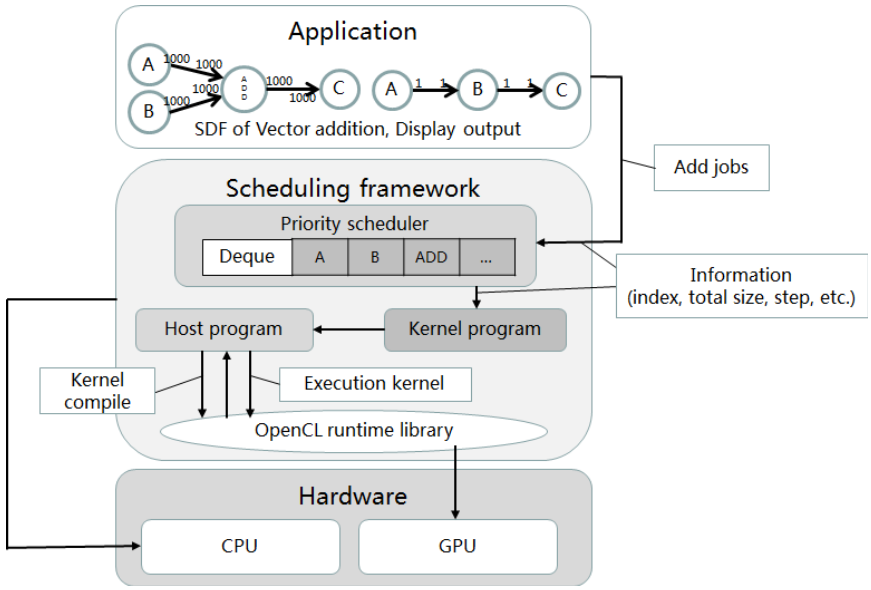


Fig. 5. Overall structure of the proposed scheduling framework.

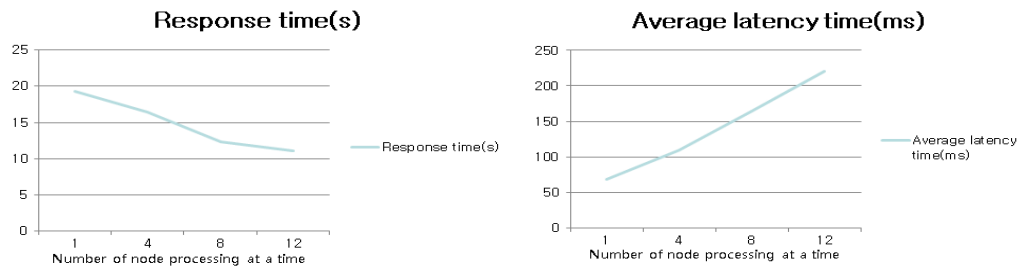
의 작업을 명령으로 주며 이러한 동작을 덱에 저장된 모든 작업이 종료될 때까지 반복적으로 수행한다.

GPU가 많은 양의 데이터를 한 번에 처리한다면 훨씬 더 향상된 속도의 결과를 볼 수 있지만 벡터덧셈과 같이 간단한 연산의 경우 너무 빠른 속도로 처리가 되어 성능차를 확인하기 힘들어 네 개로 나눠진 노드를 한 번에 여러 번 실행 시키는 방식으로 대체하였다.

Fig. 6은 각각 2십만 개의 원소를 갖는 두 벡터의 덧셈과 영상출력 응용을 함께 수행할 때의 영상출력

Number of node processing at a time	Average latency time(ms)	Response time(s)
1	68	19.31
4	109	16.44
8	164	12.36
12	221	11.09

(a)



(b) (c)

Fig. 6. (a) Vector addition by different number of node processing at a time. (b) Frame output latency time, (c) Vector addition response time.

의 지연시간과 벡터덧셈의 응답시간을 나타낸 것이다. 이때 GPU 커널이 한 번에 연산하는 벡터의 원소 개수는 1000이다. 처리하는 노드의 크기 변화에 따른 결과를 보기 위해 다수의 노드를 연속되게 처리하였다. (a)는 한 번에 처리하는 노드의 수의 증가에 따른 프레임 간의 지연시간과 벡터덧셈의 응답시간의 결과이다. (b)는 영상출력 프레임간의 지연시간(latency time)이 얼마나 벌어지게 되는지를 그래프로 보여준다. 노드 하나씩 처리하는 첫 번째 경우에 대해서는 지연 시간이 높지 않지만 한 번에 처리하는 노드의 수가 증가함에 따라 지연시간이 증가함을 볼 수 있다. 반면 벡터덧셈의 응답시간을 나타내는 (c)에서는 한 번에 처리하는 노드의 수가 증가함에 따라 응답시간(response time)은 감소함을 볼 수 있다. 즉, 한 번에 처리하는 벡터덧셈의 연산이 증가함에 따라 영상출력의 프레임률 손실은 증가하지만 벡터덧셈에 대한 응답시간은 더욱 향상됨을 알 수 있다.

이를 통해 응용 프로그램을 분할을 작은 단위로 할 때와 큰 단위로 할 때 응용 프로그램의 특성에 따른 장점과 단점이 있음을 알 수 있다. 응용 프로그램의 특성에 맞춰 적절한 단위로 분할한다면 응용 분할에 의한 성능 향상을 극대화 할 수 있을 것이다.

6. 결 론

본 논문에서는 응용을 선점형으로 처리하지 못하는 모바일 기반의 GPU의 한계에 따른 문제점에 대해 언급하였고 이를 해결하기 위한 방안으로 SDF 그래프에 기반 한 응용 프로그램의 분할에 따른 스케줄링 프레임워크를 제시하였다. 실제적으로 벡터덧셈과 영상출력이라는 두 응용 프로그램에 대한 GPU 스케줄링을 소개했고 제시한 방법을 이용해 실제 GPU 스케줄링이 가능함을 실험을 통해 보여주었다. 마지막으로 응용 프로그램 분할 단위의 크기조정에 따른 성능 향상의 가능성도 제시하였다.

대규모의 문제해결의 요구가 점점 증가에 따라 병렬 처리 기술은 수많은 분야에서 각광받고 있는 지금, 모바일 시스템에서 병렬 처리 기술의 비선점형 스케줄링 한계를 극복할 프레임워크를 제시했다는 것에 큰 의미가 있다.

나아가 이를 근간으로 각 시스템에 적합한 스케줄링 기법과 더불어 CPU와 GPU를 유기적으로 사용할 수 있는 최적화된 스케줄링 기법이 연구 개발 된다면 병렬 처리의 성능을 극대화 시킨 시스템 구축이 가능할 것이다.

REFERENCE

[1] J. Lee, J. Lee and S. Kim, "*Implementation of a GPU Cluster System using Inexpensive Graphics Devices*" Journal of Korea Multimedia Society, Vol. 14, No. 11, pp. 1458-1466, 2011.

[2] D. Shin and J. Park, "A Performance Improvement for Tree Search using Dynamic Load Balancing between CPU and GPGPU," *Journal of Korean Institute of Infomration Technology*, Vol. 10, No. 2, pp. 132-140, 2012.

[3] S.-h.-s. Yi and Y. Yi, "Accelerating Hadoop MapReduce on CPU-GPU Heterogeneous Platforms: A Case Study with CKY Parser," *Journal of KIISE: Computing Practices and Letters*, Vol. 20, No. 6, pp. 329-338, 2014.

[4] T. Gautier, J.V.F. Lima, N. Maillard, and B. Raffin "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," *Proceedings of IEEE 27th International Symposium on Parallel & Distributed Processing,* pp. 1299-1308, 2013.

[5] N.S. Arora, R.D. Blumofe, and C.G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 119-129, 1998.

[6] J. Kim, J. Lee, and W. Choi, "Technology Trends of Runtime Systems to Realize High Performance Computing," *Electronics and Telecommunications Trends*, Vol. 27, Issue 6, pp. 124-133, 2012.

[7] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-Grained Resource Sharing for Concurrent GPGPU Kernels," *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism,* pp. 10-10, 2012.

[8] GeForce GTX 580, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580 (accessed Jul., 14, 2014)
[9] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, Vol 75, Issue 9, pp. 1235-1245, 1987.
[10] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, et al., "Dataflow Analysis for Real-time Embedded Multiprocessor System Design," *Dynamic and Robust Streaming in and between Connected Consumer-electronic Devices*, Vol. 3, pp. 81-108, 2005.

이 용 빈

2014년 2월 전북대학교 컴퓨터공학부 졸업
2014년 3월~현재 전북대학교 컴퓨터공학부 석사과정
관심분야 : 임베디드 시스템, 빅데이터 컴퓨팅, GPU

김 성 찬

1998년 2월 서울대학교 금속공학과 학사
2000년 2월 서울대학교 컴퓨터공학부 석사
2005년 8월 서울대학교 전기컴퓨터공학부 박사

현재 전북대학교 컴퓨터공학부 조교수
관심분야 : 임베디드 시스템, 병렬 컴퓨팅, 빅데이터 컴퓨팅, GPU, SSD