KIPS Trans. Comp. and Comm. Sys.
Vol.10, No.5 pp.123~136
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2021.10.5.123

GPU Memory Management Technique to Improve the Performance of GPGPU Task of Virtual Machines in RPC-Based GPU Virtualization Environments

Jihun Kang†

ABSTRACT

RPC (Remote Procedure Call)-based Graphics Processing Unit (GPU) virtualization technology is one of the technologies for sharing GPUs with multiple user virtual machines. However, in a cloud environment, unlike CPU or memory, general GPUs do not provide a resource isolation technology that can limit the resource usage of virtual machines. In particular, in an RPC-based virtualization environment, since GPU tasks executed in each virtual machine are performed in the form of multi-process, the lack of resource isolation technology causes performance degradation due to resource competition. In addition, the GPU memory competition accelerates the performance degradation as the resource demand of the virtual machines increases, and the fairness decreases because it cannot guarantee equal performance between virtual machines. This paper, in the RPC-based GPU virtualization environment, analyzes the performance degradation problem caused by resource contention when the GPU memory requirement of virtual machines exceeds the available GPU memory capacity and proposes a GPU memory management technique to solve this problem. Also, experiments show that the GPU memory management technique proposed in this paper can improve the performance of GPGPU tasks.

Keywords : GPU Virtualization, GPU Memory, Resource Managements, Cloud Computing, HPC Cloud

RPC 기반 GPU 가상화 환경에서 가상머신의 GPGPU 작업 성능 향상을 위한 GPU 메모리 관리 기법

강 지 훈†

요 약

RPC(Remote Procedure Call) 기반 GPU(Graphics Processing Unit) 가상화 기술은 다수의 사용자 가상머신에게 GPU를 공유하기 위한 기술 중 하나이다. 하지만 클라우드 환경에서 일반적인 GPU는 CPU나 메모리와는 다르게 가상머신의 자원 사용량을 제한할 수 있는 자원 격리(Isolation) 기술을 제공하지 않는다. 특히 RPC 기반 가상화 환경에서는 각 가상머신에서 실행되는 GPU 작업은 멀티 프로세스 형태로 수행되기 때문에 자원 격리 기술의 부재는 자원 경쟁으로 인한 성능 저하 문제를 발생시킨다. 그리고 GPU 메모리 경쟁은 가상머신들의 자원 요구량이 많을수록 성능 저하를 가속화하고 가상머신 사이의 균등한 성능을 보장하지 못하기 때문에 공평성이 저하되는 문제를 발생시킨다. 본 논문에서는 RPC 기반 GPU 가상화 환경에서 사용자 가상머신들의 GPU 메모리 요구량이 가용 GPU 메모리 용량을 초과했을 때 발생하는 자원 경쟁으로 인한 성능 저하 문제 분석하고 이를 해결하기 위한 GPU 메모리 관리 기법을 제안한다. 또한, 실험을 통해 본 논문에서 제안한 GPU 메모리 관리 기법이 GPGPU 작업의 성능을 향상시킬 수 있다는 것을 보여준다.

키워드 : GPU 가상화, GPU 메모리, 자원 관리, 클라우드 컴퓨팅, 고성능 클라우드

1. 서 론

GPU를 범용 연산장치로 사용하는 GPGPU 기술은 GPU에 존재하는 수천개의 연산 코어를 사용해 CPU의 성능 범위를 넘어서는 대규모 고성능 병렬처리를 가능하게 하며, 산술 연산, 암호화, 데이터 분석과 같이 다양한 분야에서 연산 성능의 가속화를 위해 사용된다. 이러한 장점을 클라우드 환경에서도 사용하기 위해 상용 클라우드 서비스에서도 사용자가 고성능 서비스를 운용할 수 있도록 GPU가 할당된 고성능 가상머신을 제공하고 있다[2,3]. GPU는 일반적으로 일부 클라우드 전용 GPU[3,4]를 제외하면 클라우드 환경과 같이 다수의 사용자가 자원을 공유하는 환경을 고려하지 않고 설계되었다. 클라우드 전용 GPU의 경우 각 가상머신이 사용할 수 있는 GPU 메모리가 하드웨어적으로 분할되어 있으며, 각각

† 준 회 원 : 고려대학교 정보대학 4단계 BK21 컴퓨터학교육연구단 박사후과정생
Manuscript Received : September 14, 2020
Accepted : November 2, 2020
* Corresponding Author : Jihun Kang(k2j23h@korea.ac.kr)

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/ licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

의 가상머신들은 GPU 메모리 요구량에 따라 GPU 메모리를 동적으로 할당 받을 수는 없지만 균등하게 할당된 GPU 메모리 영역을 독립적으로 사용하고 GPU 전용 하이퍼바이저가 존재하기 때문에 가상머신 사이의 GPU 메모리 경쟁이 발생하지 않는다. 일반적인 GPU의 경우 다중 GPGPU 작업이 실행될 때 각 GPGPU 작업들은 독립적으로 사용할 수 있는 GPU 메모리 영역의 제한 없이 모든 가상머신이 GPU 메모리를 공유하기 때문에 가용 GPU 메모리를 사용하기 위한 경쟁이 발생하며, 이로 인해 GPU 메모리를 균등하게 사용하는데 제한적이다.

또한, GPGPU 프로그래밍 모델[5,6]에서는 작업에 사용될 데이터를 GPU 메모리에 직접 작성할 수 없기 때문에 메인 메모리에 데이터를 입력한 후 GPU 메모리로 복사하는 방식을 사용한다. 이러한 특징으로 인해 GPU 메모리가 초과 사용되는 환경에서 동시에 실행되는 다수의 GPGPU 작업은 메인 메모리와 GPU 메모리 사이의 데이터 교환이 가용 GPU 메모리가 확보될 때까지 지속적으로 발생하며, 이는 GPGPU 작업을 실행하는 가상머신들이 증가할수록 GPU 메모리가 부족해지면 스래싱(thrashing) 문제와 데이터 교환으로 인해 동시에 실행되는 GPGPU 작업의 성능이 저하된다.

GPGPU 작업의 경우 작업에 사용할 데이터가 전용 메모리, 즉, GPU 메모리에 존재해야 한다. GPGPU 작업에서 GPU는 CPU의 보조 처리기(Coprocessor)로써 작업 전체에서 GPU 연산 부분만을 수행하며 작업의 전반적인 부분은 모두 CPU에서 처리된다. 하지만 CPU가 GPU 메모리에 직접 접근할 수 없기 때문에 GPU 메모리에서 데이터를 직접 읽어오거나 수정할 수 없다. 이러한 특징으로 인해 GPGPU 작업에서는 연산에 사용될 데이터 입력과 연산 결과를 출력하기 위해 GPU 연산 전, 후로 메인 메모리와 GPU 메모리 사이의 데이터 입출력 작업이 발생한다. GPU 연산을 실행할 때 연산에 사용할 데이터가 GPU 메모리에 적재된 후에 GPU 연산을 시작할 수 있기 때문에 GPU 메모리의 할당 순서는 다중 GPGPU 작업 환경에서 각 GPGPU 작업의 성능에 직접적으로 영향을 미친다. 단일 사용자가 GPU를 전부 사용하는 환경에서는 다중 GPGPU 작업을 동시에 호스팅 해도 프로그래머가 각 GPGPU 작업의 GPU 메모리 사용량을 소스코드 수준에서 조절할 수 있지만, 다중 사용자가 단일 GPU를 공유하는 클라우드 환경에서 GPGPU 프로그램을 개발하는 프로그래머는 다른 가상머신 사용자들의 GPU 메모리 사용량을 고려할 수 없으며, 가상머신 사용자는 자신이 호스팅 하는 GPGPU 작업만을 중요시하기 때문에 클라우드 제공자 측면에서 가상머신 사용자들 사이에서 GPU 메모리 경쟁을 최소화하고 GPU 메모리를 균등하게 사용할 수 있도록 GPU 자원을 관리해야 한다.

클라우드 제공자는 물리 자원을 논리적으로 분할하여 가상머신 사용자가 독립적으로 사용할 수 있도록 컴퓨팅 자원을 제공해 주지만 GPU 메모리의 경우 메인 메모리와 비교했을 때 상대적으로 가용 용량이 매우 적다. 평균적으로 메인 메모리는 단일 서버에 1TB 이상 설치할 수 있지만 GPU의 경우 일반적인 단일 서버에 최대 4~16개 정도를 설치할 수 있다. 고성능 GPU의 경우 일반적으로 32GB 정도의 전용 메모리[7]가 내장되어 있으며 각 GPU에 내장된 전용 메모리는 교체할 수 없기때문에 GPU 자체를 교체하지 않는 이상 메인 메모리처럼 용량을 확장할 수 없다. 이러한 자원의 제약으로 인해 다수의 가상머신이 GPU를 공유하는 환경에서는 GPU 메모리는 자원 기아 현상에 빠르게 도달할 가능성이 있으며, 다수의 가상머신이 요구하는 GPU 메모리의 총량이 가용 GPU 메모리 용량을 초과하는 경우 메모리 경쟁이 심화되는 문제를 발생시킨다. GPU 메모리 부족 현상을 해결하기 위한 가장 간단한 방법은 고성능 GPU를 최대한 많이 설치하는 것이다. 하지만 GPU 메모리의 경우 일반적으로 GPU 연산을 수행할 때만 사용되며 메인 메모리와 같이 OS, 백그라운드 서비스와 같이 메모리를 상시 점유하는 경우가 매우 적기 때문에 메인 메모리와 비교하면 상대적으로 유휴 상태일 확률이 더 높다. 이러한 특징을 종합해 볼 때 GPU 메모리 부족 현상을 방지하기 위해 단순하게 고성능 GPU를 많이 사용하는 것은 비용 및 자원 낭비로 이어질 수 있다.

본 논문에서는 RPC 기반 GPU 가상화 환경에서 GPU 메모리의 초과 예약을 지원하기 위한 데이터 사용 시점 기반 GPU 메모리 할당 기법을 제안한다. 제안하는 기법에서는 GPU 메모리의 초과 예약으로 인한 메인 메모리와 GPU 메모리 사이의 데이터 교환과 스래싱으로 인한 오버헤드를 최소화하고 GPU 메모리 부족으로 인한 가상머신 사이의 자원 경쟁을 완화하기 위해 각 가상머신에서 실행하는 GPGPU 작업이 GPU 메모리를 균등하게 사용할 수 있도록 관리한다. 또한, 기존 GPGPU 작업의 흐름처럼 GPGPU 작업에 사용할 메모리 영역을 작업 시작 시 미리 할당하는 방식이 아니라 가용 GPU 메모리 용량을 고려해 GPGPU 작업의 데이터 사용 시점에 따라 데이터 일부를 GPU 메모리에 적재하는 분할 처리 방식을 사용한다. 이를 통해 각 가상머신의 GPU 메모리 사용량을 조절하고 GPU 메모리 초과 예약 상태를 최소화해서 스래싱 문제를 해결하고 GPU 메모리 경쟁을 완화한다. 본 논문의 주요 기여는 다음과 같다.

- 본 논문에서 제안하는 GPU 메모리 할당 기법은 RPC 기반 GPU 가상화 환경에서 가상머신 사이의 GPU 메모리 경쟁을 최소화하고 각 가상머신이 GPU 메모리를 균등하게 사용할 수 있는 GPU 메모리 관리 기법을 제안한다.
- 가상머신의 GPU 데이터 사용 시점을 기반으로 사용하지 않는 GPU 메모리 영역을 회수하여 가용 GPU 메모리를 확보하고 불필요한 데이터로 인한 GPU 메모리 낭비를 최소화한다.
- 다수의 가상머신이 요구하는 GPU 메모리 총량이 가용 GPU 메모리 용량을 초과한 경우 GPGPU 작업에 사용할 전체 데이터의 부분 할당을 통해 GPU 메모리에 GPGPU 작업에서 요구하는 데이터의 부재로 인한 스래싱 오버헤드를 최소화한다.

- 본 논문의 GPU 메모리 할당 기법은 오픈소스 기반 가상화 플랫폼인 Xen[8]을 사용해 가상화 환경을 구성하였으며, 실험을 통해 기존 환경과 비교했을 때 동일한 작업의 실행 시간이 최대 68% 감소하며, 제안하는 기법으로 인한 오버헤드가 매우 적다는 것을 증명한다.

본 논문의 2장에서는 본 논문에서 제안하는 기술을 설명하기 위한 배경 기술들을 설명한다. 3장에서는 실험을 통해 다중 GPGPU 작업을 동시에 실행할 때 GPU 메모리 경쟁으로 인한 성능 저하 문제를 증명하고 제안하는 기법의 주요 목적에 대한 동기부여를 설명한다. 4장에서는 본 논문에서 제안하는 GPU 메모리 관리 기법에 대해 자세히 설명하고 5장에서는 RPC 기반 GPU 가상화 환경에서 본 논문에서 제안한 기법의 효율성과 기존 환경과 비교한 성능을 평가한다. 6장에서는 관련 연구에 대해 설명하고 마지막 7장에서는 본 논문의 결론과 추후 연구를 설명한다.

2. 배경 기술

2.1 GPGPU 프로그래밍 모델

GPGPU 프로그래밍 모델은 GPU를 범용 연산장치로 사용하기 위한 기술이다. GPGPU 프로그래밍 모델에서 GPU는 CPU의 보조 연산장치로 사용되며, CPU를 배제하고 단독적으로 프로그램 코드를 시작하거나 작업을 수행할 수 없다. GPU는 단순한 보조 연산장치이기 때문에 모든 명령어는 CPU로부터 시작하고 CPU의 요청에 따라 프로그램의 GPU 연산 부분만 GPU에서 처리된다.

GPGPU 작업 환경은 이기종 컴퓨팅(Heterogeneous Computing) 환경으로써 CPU와 GPU 모두를 연산장치로 사용한다. GPU는 CPU의 보조 프로세서(Co-Processor)로써 프로그램의 병렬처리 부분을 처리한다. GPGPU 프로그램에서 GPU 연산 부분은 일반적으로 커널 함수라고 부르며, CPU를 통해 프로그램의 전반적인 작업을 수행한 후 프로그램 내부에서 커널 함수가 호출되면 GPU에서 연산을 수행한다.

GPGPU 작업과 일반적인 CPU 작업의 가장 큰 차이점은 작업에 사용될 데이터 관리에 있다. 일반적으로 CPU에서 사용할 데이터는 메인 메모리에서 관리 되지만 GPU의 경우 GPU 연산에 사용될 데이터를 적재하기 위한 GPU 메모리가 별도로 존재한다. 하지만 GPU 메모리는 메인 메모리와 다르게 데이터를 직접 입력할 수 없으며, CPU가 GPU 메모리에 직접 접근할 수 없다. 이로 인해 GPU 연산에 사용될 데이터를 GPU 메모리에 업로드하기 위해서는 메인 메모리에 데이터를 업로드한 후에 GPU 메모리로 복사하는 방식을 사용하며, GPU 작업 결과를 읽어오기 위해서는 GPU 메모리에서 메인 메모리로 작업 결과를 복사해야 작업 결과 데이터에 접근할 수 있다. Fig. 1은 GPGPU 프로그래밍 모델에서 GPGPU 작업의 흐름도를 보여준다.

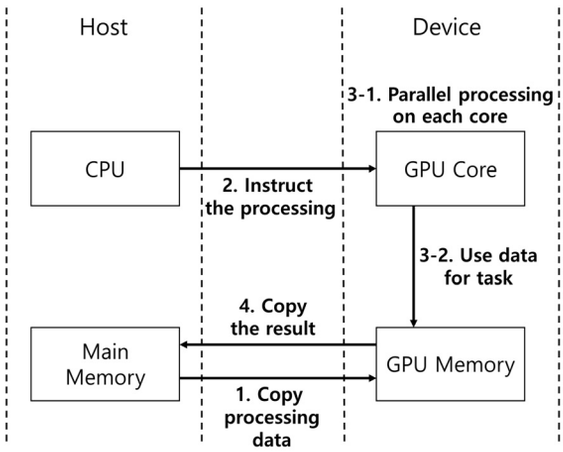


Fig. 1. GPGPU Programming Model

Fig. 1에서 보여주는 것과 같이 GPGPU 프로그래밍 모델에서는 연산장치에 따라 Host와 Device 영역으로 분류된다. 또한, 앞서 설명한 것과 같이 GPU 메모리에 적재되어있는 데이터에 직접 접근할 수 없어서 GPU 연산을 수행하기 전과 후에 메인 메모리와 GPU 메모리 사이의 데이터 전송 작업이 필수적으로 발생하며, 메인 메모리에서 GPU 메모리로 데이터 전송이 완료되어야 GPU 연산을 수행할 수 있고 GPU 메모리에서 메인 메모리로 GPU 작업 결과 복사를 완료해야 GPU 작업의 결과를 확인할 수 있다. 이러한 특성으로 인해 GPU 메모리에 데이터가 전송되는 성능과 GPU 메모리에서 메인 메모리로 작업 결과의 전송 성능은 GPGPU 작업의 성능에 많은 영향을 미치며, 실제 많은 수의 GPGPU 애플리케이션은 전체 수행 시간에서 GPU 연산 시간보다 메인 메모리와 GPU 메모리 사이의 데이터 전송 시간이 더 많은 비중을 차지한다.

2.2 GPU 메모리의 데이터 관리

앞서 설명한 것과 같이 GPGPU 작업을 수행할 때 GPU 연산에 사용될 데이터는 GPU 메모리에 존재해야 한다. 메인 메모리의 경우 자주 사용되지 않는 페이지를 하드디스크에 덤프 파일 형태로 저장하여 가용 메모리 용량을 확보하여 메인 메모리의 초과 사용을 관리하기 위해 가상 메모리 기능을 제공한다. GPU 메모리의 경우 메인 메모리와 비슷하게 가상 GPU 메모리 영역을 제공하는데 GPU 메모리의 경우 가상 GPU 메모리 영역을 위해 메인 메모리를 사용한다. GPU 메모리는 데이터를 적재하기 위해 전용 영역과 공유 영역 두 가지 영역을 관리한다. 전용 영역은 GPU 메모리를 말하며, 공유 영역은 메인 메모리의 일정 부분을 데이터 저장에 사용할 수 있도록 할당된 영역을 말한다.

하지만 공유 영역으로 설정된 메모리 영역은 메인 메모리에 존재하기 때문에 공유 영역에 존재하는 데이터를 GPU가 직접 접근할 수 없으며, GPU 연산이 실행될 때 GPU 연산에 필요한 데이터는 공유 영역에서 전용 영역으로 옮겨와야

한다. 가용 GPU 메모리가 존재하는 경우 공유 영역의 데이터는 전용 영역으로 이동하기만 하면 된다. 하지만 GPU 메모리가 초과 사용되면, 가용 GPU 메모리가 부족해져서 전용 영역과 공유 영역 사이에서 데이터를 교환해야 한다. 단일 GPGPU 작업이 실행되는 일반적인 GPU 연산 환경에서는 GPGPU 작업 하나가 GPU를 독점할 수 있으며, 자원 경쟁자가 존재하지 않기 때문에 충분한 가용 자원을 확보할 확률이 높다. 하지만 단일 GPU를 다수의 가상머신이 공유하는 클라우드 환경에서는 각각의 가상머신을 사용하는 가상머신 사용자들이 GPU를 함께 공유하는 다른 가상머신들의 GPGPU 작업에 대한 정보를 얻을 수 없을뿐더러 다른 가상머신의 자원 사용을 위해 자신의 자원 사용 기회를 양보하는 상황도 발생할 수도 없기 때문에 클라우드 관리자는 가상화 계층 수준에서 각 가상머신의 GPU 메모리 사용량을 균등하게 제한하고 다수의 가상머신이 GPU 메모리의 전용공간과 공유 공간 사이의 데이터 교환을 감소시키고 GPU 메모리를 균등하게 사용할 수 있도록 클라우드 관리자 측면에서 관리해야 한다.

2.3 RPC 기반 GPU 가상화 환경

본 논문에서는 앞서 설명한 것과 같이 RPC 기반 GPU 가상화 환경에서 GPU 메모리의 관리 기법을 제안한다. RPC 기반 GPU 가상화 기법은 단일 GPU를 다수의 가상머신 사이에서 공유할 수 있도록 지원하는 기술이며, API 포워딩(API Forwarding) 방식이라고도 불린다. RPC 기반 GPU 가상화 환경에서 각각의 사용자 가상머신은 GPU를 직접 할당받지 않으며, 수정된 GPU 프로그래밍 API를 사용한다. 사용자 가상머신이 GPU 작업을 수행하면 원래의 GPU 프로그래밍 API 대신 수정된 GPU 프로그래밍 API가 GPU의 실제 소유자에게 RPC 통신을 통해 API의 정보, 데이터들을 전달한 후 작업이 완료되면 해당 사용자 가상머신에게 작업 결과를 반환하며, 일반적으로 GPU는 별도의 가상머신이나 호스트 머신이 직접 할당받고 사용자 가상머신과 GPU 소유자는 내부 통신을 통해 GPU 작업 정보를 교환한다.

Fig. 2는 본 논문의 RPC 기반 GPU 가상화 환경의 전체 구조도이다. 각 사용자 가상머신에는 수정된 GPU 프로그래밍 API가 사용되며, 수정된 GPU 프로그래밍 API는 GPU를 소유한 가상머신 에게 API 정보와 데이터를 전송할 수 있도록 RPC 통신 기능이 내장되어 있다. Fig. 2에서 보여주는 것과 같이 본 논문에서 사용한 RPC 기반 GPU 가상화 환경에서는 별도의 가상머신이(이후 GPGPUvm으로 칭함) 직접 통과 방식(Direct Pass-Through)[9]으로 GPU를 직접 할당받으며 사용자 가상머신이 실행하는 GPU 작업의 API를 전달받아 대신 처리해주는 구조를 갖는다. 본 논문의 환경에서 가상머신들은 GPGPU 프로그래밍 API로 많이 사용되는 OpenCL 기반의 GPGPU 작업을 실행하며, 사용자 가상머신의 GPU 작업을 지원하기 위해 OpenCL의 API를 수정하였다.

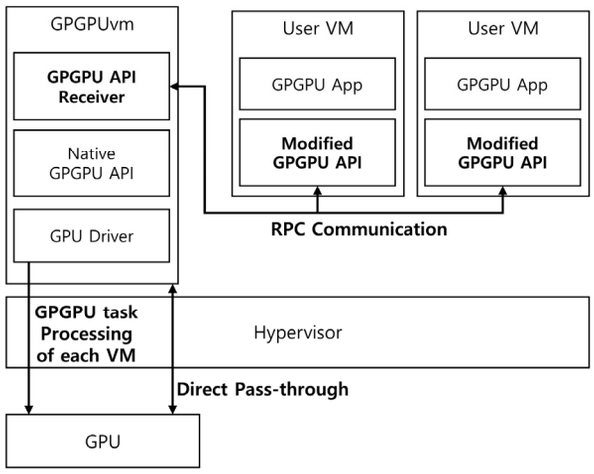


Fig. 2. RPC-based GPU Virtualization Technique

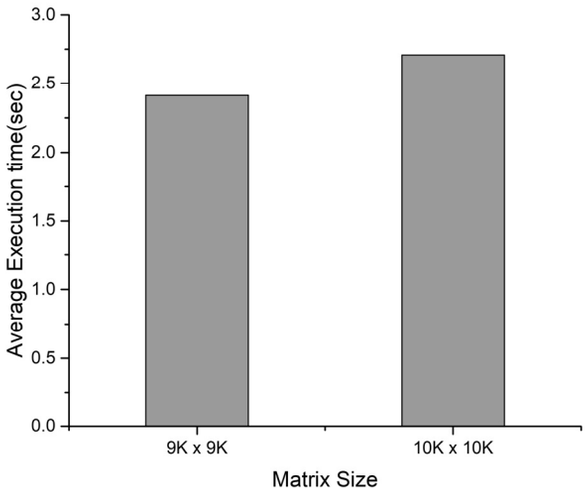
3. RPC 기반 GPU 가상화 환경에서 GPU 메모리 초과 사용으로 인한 GPGPU 작업의 성능 저하

RPC 기반 GPU 가상화 환경은 앞서 설명한 것과 같이 GPU를 실제 할당받고 직접 접근할 수 있는 GPGPUvm이라고 불리는 별도의 가상머신이 존재한다. 사용자 가상머신들이 OpenCL을 사용한 GPGPU 작업을 실행하면 수정된 OpenCL API는 RPC 통신을 통해 GPGPUvm에게 전달된 후 GPGPUvm은 전달받은 API와 작업에 사용될 데이터를 사용해 사용자 가상머신의 GPGPU 작업을 대신 처리한 후 작업 결과를 반환한다. 사용자 가상머신들의 GPGPU 작업이 실행되고 GPGPU vm에게 전달되면 GPGPUvm은 멀티 프로세스 환경에서 각각의 사용자 가상머신이 요청한 GPGPU 작업을 실행한다.

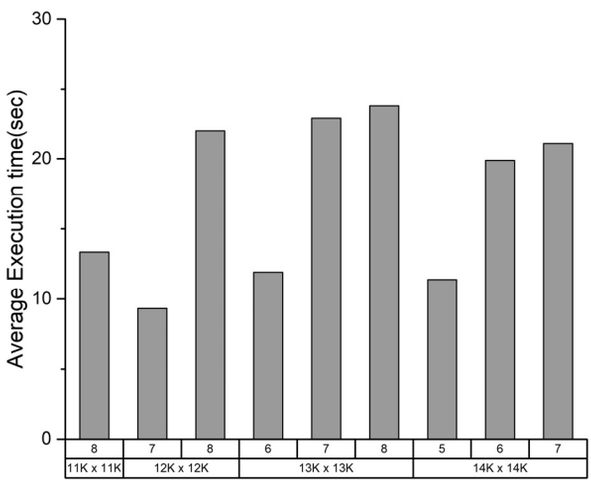
이번 장에서는 RPC 기반 GPU 가상화 환경에서 GPU 메모리의 초과 사용 상황을 발생시키기 위해 최대 8개의 가상머신에서 행렬 곱셈 작업을 동시에 실행하여 성능을 측정한다. 또한, 가용 GPU 메모리가 충분한 경우에 가상머신의 GPGPU 작업의 성능을 측정하여 GPU 메모리 초과 사용으로 인한 성능 저하 문제를 분석한다. 실험을 위한 GPGPU 작업과 사용되는 가상머신의 개수는 Table 1과 같다.

Table 1. Experiment Scenario

GPU memory status	Matrix size	Number of used VMs	Single execution performance
GPU memory sufficient	9K × 9K	8 VMs	0.624
	10K × 10K	8 VMs	0.702
Out of GPU memory	11K × 11K	8 VMs	0.858
	12K × 12K	7 and 8 VMs	0.936
	13K × 13K	6, 7 and 8 VMs	1.045
	14K × 14K	5, 6, 7, and 8 VMs	1.155



(a) GPGPU Task Performance when GPU Memory is Sufficient



(b) Performance when GPU Memory is Overused

Fig. 3. Performance Degradation of GPGPU Task Owing to Increase in GPU Memory usage and Task Size

이번 장의 실험에서는 GPU 메모리가 부족한 경우와 충분한 경우로 나누어 성능을 측정한다. 본 논문에서 사용한 RPC 기반 GPU 가상화 환경은 GPGPU 작업을 실행할 때 사용자 가상머신과 GPGPUvm 사이에 데이터 전송 작업이 발생한다. 본 논문의 실험에서는 가상머신 사이의 GPU 메모리 경쟁을 최대화하기 위해서 데이터 전송 작업이 완료된 후 GPGPU 작업을 동시에 실행시킨다. 실험 결과는 Fig. 3에서 보여준다. 실험 결과에서는 사용자 가상머신과 GPGPUvm 사이의 데이터 전송 작업을 제외한 GPGPU 연산 부분의 성능만을 보여준다.

Fig. 3(a)와 (b)는 각각 GPU 메모리가 충분한 상황과 초과 사용되는 상황에서 가상머신들의 GPGPU 작업 성능을 보여준다. Fig. 3(b)의 x축은 위에서부터 가상머신의 개수, 행렬 크기를 보여준다. Fig. 3(a)와 같이 GPU 메모리가 충분한 경우에는 가상머신이 증가하거나 GPGPU 작업 규모가 커져도

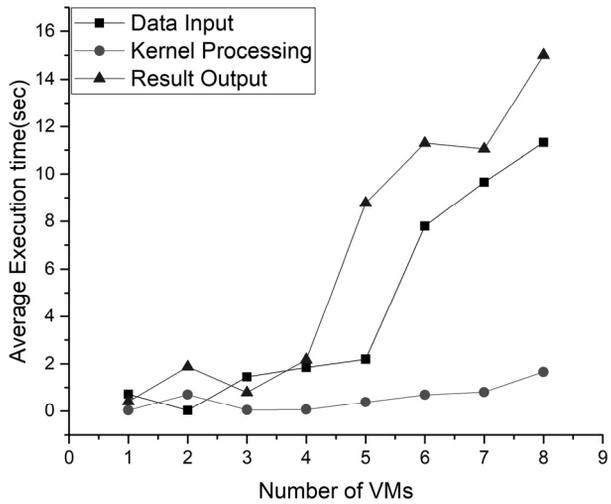


Fig. 4. Detailed Performance of 14K × 14K Matrix Multiplication as the Number of Virtual Machines Increases

급격한 성능 저하는 발생하지 않는다. GPGPU 작업과 동시 실행 중인 가상머신의 개수가 많아지면 자원 사용량이 증가하기 때문에 어느 정도의 성능 저하는 당연한 결과이다. 하지만 Fig. 3(b)에서 보여주는 것과 같이 GPU 메모리가 초과 사용되는 상황에서는 GPGPU 작업의 규모가 커지고, 실행되는 가상머신이 증가할수록 급격한 성능 저하를 보여준다. 성능 저하가 심할 때는 각 가상머신의 GPGPU 작업을 동시에 실행시키지 않고 하나씩 순차적으로 처리할 때보다 성능이 더 좋지 않다.

Fig. 4는 Fig. 3(b)의 실험에서 GPU 메모리를 가장 많이 사용하는 14K × 14K 행렬 곱셈의 성능 자세히 보여준다. Fig. 4에서는 가상머신이 증가함에 따라 GPU 메모리로 데이터 입력, 커널 함수 실행, 결과값 출력 작업 각각의 성능이 어떻게 변하는지 평균 성능을 보여준다. GPU 메모리가 초과 사용되면 GPU 메모리 경쟁상태가 심해지고 GPU 메모리에 데이터를 업로드하거나 결과값을 메인 메모리로 복사할 때도 데이터 입출력 작업에 대한 경쟁이 발생한다. 또한 앞서 설명한 것과 같이 GPGPU 작업을 실행하는 스레드들의 처리 상태에 따라 데이터를 GPU 메모리에 업로드해야하기 때문에 GPU 메모리의 전용 영역과 메인 메모리에 존재하는 공유 영역 사이의 데이터 교환이 증가한다. 이러한 문제는 GPGPU 작업을 실행할 때 스레드에서 GPU 메모리에 적재되어있는 데이터를 읽어올 때 지연을 발생시키며 가상머신과 GPGPU 작업의 데이터의 규모가 증가할수록 성능 저하는 심해진다.

본 논문에서는 앞서 수행한 실험의 결과에서 보여주는 GPGPU 메모리가 초과 사용되는 상황에서 GPGPU 작업의 성능 저하를 방지하기 위한 GPU 메모리 관리 기법을 제안한다. 본 논문에서 제안한 기법은 가용 GPU 메모리 용량과 현재 실행 중인 가상머신의 개수를 고려하여 각 가상머신이 사용할 수 있는 GPU 메모리 용량을 설정한다. 또한, GPGPU 작업에 사용되는 데이터를 GPGPU 작업을 실행하는 스레드

단위로 분할하고 분할된 데이터들을 사용 가능한 GPU 메모리 용량에 맞춰 그룹화한 후 데이터 그룹을 GPU 메모리에 순차적으로 업로드하는 방식을 사용한다. 본 논문에서는 이를 통해 메인 메모리 사이의 데이터 교환과 GPU 메모리 경쟁을 최소화하여 각 가상머신에서 실행되는 GPGPU 작업의 성능을 향상시킨다.

4. RPC 기반 GPU 가상화 환경에서 GPGPU 작업성능을 향상하기 위한 GPU 메모리 관리 기법

이전 장에서 수행한 실험을 통해 GPU 메모리가 초과 사용되면 동시에 실행되는 GPGPU 작업의 성능이 급격하게 저하되는 것을 확인했다. 본 논문에서 제안하는 GPU 메모리 관리 기법의 주요 목적은 가상머신 사이의 GPU 메모리 경쟁을 최소화하여 가상머신이 실행하는 GPGPU 작업의 성능을 향상하는 것이다.

본 논문에서 GPU 메모리의 경쟁을 최소화하기 위해 GPGPU 작업을 수행하는 가상머신의 개수와 가용 GPU 메모리의 용량을 고려한다. 이전 장에서 수행한 실험 결과에서 다수의 가상머신에서 동시에 실행하는 GPGPU 작업들의 데이터를 동시에 GPU 메모리에 업로드하면 GPU 메모리 경쟁으로 인해 GPGPU 작업의 성능이 크게 저하되는 것을 확인할 수 있었다. 본 논문에서는 GPU 메모리의 경쟁을 최소화하기 위해 가상머신의 개수와 가용 GPU 메모리를 고려하여 GPGPU 작업에 사용될 데이터를 분할 업로드 하여 다수의 가상머신이 GPU 메모리를 균등하게 사용할 수 있도록 지원하는 GPU 메모리 관리 기법을 제안한다.

앞서 설명한 것과 같이 본 논문에서 제안하는 GPU 메모리 관리 기법은 RPC 기반 GPU 가상화 환경에서 GPU 메모리의 경쟁을 최소화하기 위해 가상머신에서 실행하는 GPGPU 작업에 사용될 데이터를 분할하고 순차적으로 GPU 메모리에 업로드한다. 이를 통해 GPU 메모리가 초과 사용되는 것을 방지하고 각 가상머신이 사용할 수 있는 GPU 메모리의 용량을 제한하여 각 가상머신이 GPU 메모리를 균등하게 사용할 수 있도록 지원한다. 본 논문에서 제안하는 GPU 메모리 관리 기법은 Fig. 5와 같다.

본 논문에서 제안하는 GPU 메모리 관리 기법은 크게 2가지 요소로 구성된다: 1) GPU 데이터 분할 및 2) GPU 메모리 할당 및 해제. 각 서브 시스템에 대한 자세한 설명은 다음과 같다.

4.1 GPU 데이터 분할 기법

GPU 데이터 분할 기법은 본 논문에서 제안하는 GPU 메모리 할당 기법에서 가장 중요한 역할을 한다. GPU 데이터 분할 기법의 주요 역할은 GPGPU 작업에 사용될 데이터 객체를 분할하고 분할된 각 데이터 객체들을 관리하는 것이다. 앞서 설명한 것과 같이 본 논문에서 제안하는 GPU 메모리 할당 기법에서는 다수의 가상머신이 GPU 메모리를 균등하게 사용할 수 있도록 각 가상머신이 사용할 수 있는 GPU 메모리 용량을 제한한다. 또한, 전용 메모리 영역과 예약 영역 사이의 데이터 교환으로 인한 오버헤드를 최소화하기 위해 가용 GPU 메모리 용량에 적합한 크기의 데이터 객체만을 업로드하도록 데이터 객체를 관리한다. GPGPU 작업에 사용될 데이터 객체를 분할 처리하게 되면 GPU 메모리에 데이터 객체의 일부분만 업로드되기 때문에 커널 함수를 실행하는 스레드들도 분할 처리해야 한다.

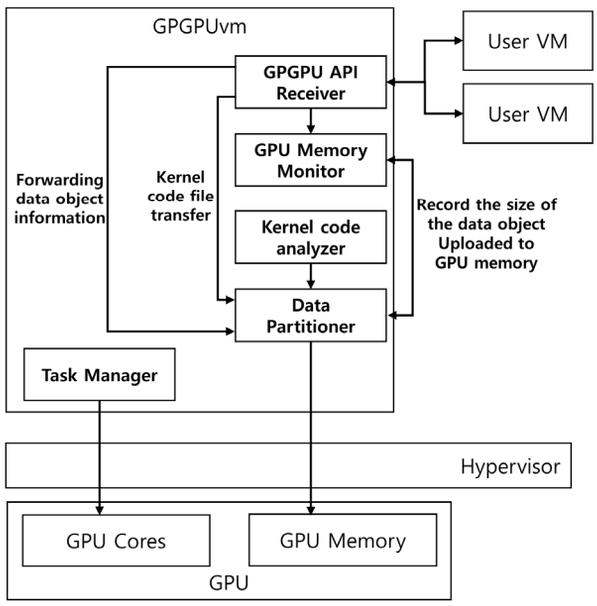


Fig. 5. Overall Structure of the System

데이터 객체의 일부분만 GPU 메모리에 적재된 상태에서 커널 함수를 실행하기 위한 모든 스레드가 한 번에 호출되면 스레드 일부를 제외하고 나머지 스레드들은 잘못된 데이터 객체를 참조하거나 존재하지 않는 데이터 객체의 인덱스에 접근하는 문제를 발생시킬 수 있다. 또한 앞서 설명한 것과 같이 GPGPU 작업에 사용될 데이터는 GPU 메모리에 준비되어 있어야 한다. GPU 연산을 수행하는 스레드가 작업을 시작하면 인위적으로 문맥 교환을 발생시키는 것이 불가능하기 때문에 GPGPU 작업이 정상적으로 완료되기 위해서는 스레드가 작업에 사용할 데이터는 온전하게 GPU 메모리에 존재해야 한다. 본 논문에서 제안한 기법에서는 이러한 특징을 기반으로 GPGPU 작업의 스레드가 정상적으로 작동하도록 지원하면서 데이터 객체를 가장 효율적으로 분할하기 위해 데이터 객체들을 스레드 단위로 분할하는 방법을 사용한다.

데이터의 객체를 분할하기 위해서는 우선 데이터 객체의 유형을 분류해야 한다. 이전 장에서 설명한 것과 같이 GPGPU 작업은 GPU 메모리에 데이터를 직접 작성할 수 없고 GPU 메모리에 저장된 데이터를 직접 읽어올 수 없기 때문에 메인 메모리를 통해 데이터를 입력하거나 출력하는 작업을 수행한다. GPU 메모리에 존재하는 데이터 객체는 입력 데이터 객체와 출력 데이터 객체로 크게 2가지로 분류할 수 있다. 입력

데이터 객체는 GPU 메모리가 할당된 후 메인 메모리에 작성해 놓은 데이터 객체를 복사하기 때문에 유효한 데이터가 존재하며, GPU는 입력 데이터 객체를 사용해 GPU 연산을 수행한다. 출력 데이터 객체는 GPU 연산의 결과를 저장하기 위한 데이터 객체이기 때문에 별다른 데이터를 입력하지 않고 GPU 메모리에 작업 결과값을 저장할 영역만 할당해 놓는다. 일반적으로 스레드들이 데이터를 읽어오기만 하는 입력 데이터 객체와 GPU 연산 결과를 저장하는 출력 데이터 객체는 스레드가 데이터 객체에 접근하는 방법과 목적이 다르기 때문에 데이터 객체를 스레드 단위로 적절하게 분할하기 위해서는 입력 및 출력 데이터 객체를 서로 다른 방식으로 분할해야 한다.

GPGPU 작업은 수천개의 스레드를 통해 대규모 병렬처리를 수행하며, 각각의 스레드는 서로 다른 입력 데이터 객체의 원소 그룹을 사용하여 같은 커널 코드를 실행한다. 일반적으로 GPGPU 작업은 작업 처리 중에 발생하는 스레드 사이의 간섭과 스레드의 대기 시간을 방지하기 위해 각각의 스레드들의 상호 종속성이 발생하지 않도록 설계된다. 즉, 스레드 B가 스레드 A의 작업 결과를 사용해야 하거나 스레드 A와 B가 데이터 객체의 같은 원소를 수정할 수 없도록 설계된다. 따라서 각각의 스레드들은 서로 작업 처리시 연관성이 없고 상호 데이터 작성 순서로 인한 충돌이 발생하지 않는다. 예를 들어 앞서 수행한 실험에서 사용했던 행렬 곱셈의 경우 행렬 A, B의 곱셈 결과를 행렬 C에 저장한다. 행렬 A와 B의 경우 단순 읽기 작업이기 때문에 다수의 스레드가 동시에 접근해도 상관없지만, 행렬 C에는 각각의 스레드가 행렬 곱셈의 결과를 저장해야 하기 때문에 여러 스레드가 데이터 객체의 동일한 인덱스에 접근하면 충돌이 발생한다. 이러한 특징으로 인해 입력 데이터 객체는 여러개의 스레드가 동시에 데이터 객체의 동일한 원소에 접근해도 문제가 발생하지 않지만 출력 데이터 객체의 경우 복수개의 스레드가 동시에 동일한 데이터 객체의 원소에 접근하면 데이터 저장 작업의 충돌이 발생하기 때문에 일반적으로 스레드 1개에 1개의 원소를 독립적으로 사용하도록 구현된다.

앞서 설명한 것처럼 입력 데이터와 출력 데이터의 접근 방식으로 인해 GPGPU 작업에서 사용되는 데이터 객체를 분할하기 위해서는 가상머신의 요청으로 생성된 데이터 객체가 입력 데이터인지 출력 데이터인지 분류하고 해당 커널 함수에서 사용되는 데이터 객체인지 식별해야 한다. 본 논문에서는 데이터 객체의 분류를 위해 RPC를 통해 전달받은 API와 커널 코드에 대한 간단한 분석 작업을 수행한다.

커널 코드 분석 작업에서 가장 먼저 수행해야 할 작업은 GPU 연산에 사용되는 입력 데이터와 연산 결과를 저장하는 출력 데이터에 대한 데이터 객체를 분류하는 것이다. 본 논문에서 제안하는 기법에서는 RPC를 통해 전달받은 커널 함수의 매개변수를 통해 데이터 객체의 이름을 찾아내고 ***clEnqueueWriteBuffer()*** 함수와 같이 메인 메모리에서 GPU 메모리로 데이터를 복사하는 작업의 매개변수를 확인한다. 그리고 앞서 찾은 데이터 객체들의 이름과 비교한 후 커널 함수에서 사용할 입력 데이터 객체를 분류한 후 나머지 데이터 객체들은 출력 데이터 객체로 분류한다. API의 매개변수 순서는 고정적이기 때문에 RPC 통신을 통해 전달받은 API의 이름과 매개변수의 정보를 통해 데이터 객체의 이름은 쉽게 찾을 수 있다. 또한 메인 메모리에서 데이터 복사가 발생하지 않은 객체는 출력 데이터 객체로 분류할 수 있다.

사용자 가상머신에서 요청한 데이터 객체의 분류가 끝나면 데이터 객체를 분할하기에 앞서 입력/출력 데이터 객체의 분할 규모를 정해야 한다. GPGPU 작업이 잘못된 데이터 객체의 원소에 접근하는 것을 방지하면서 데이터 객체를 분할하기 위한 가장 좋은 방법은 GPU 메모리에 업로드할 데이터 객체를 스레드 단위로 분할하는 것이다. 데이터 객체의 분할 규모를 결정하기 위해서는 커널 함수를 실행하기 위해 생성되는 스레드의 전체 개수를 알아야 한다. 본 논문의 기법에서는 커널 함수의 스레드의 개수를 확인하기 위해 OpenCL에서 전역 스레드의 개수를 매개변수로 사용하는 ***clEnqueueNDRangeKernel()***와 같은 커널 함수의 API 매개변수를 검사한다. 커널 함수를 실행하기 위한 스레드의 개수가 커널 함수가 사용할 데이터 객체의 최대 분할값을 나타낸다.

앞서 수행한 데이터 객체의 분할 규모와 입/출력 데이터 객체의 분류 작업의 경우는 커널 코드의 내용 없이 완벽하게 외부에서 수행할 수 있다. 하지만 입력/출력 데이터 분할 작업의 경우 매우 복잡하다. 각각의 스레드가 데이터 객체에 접근하는 패턴은 GPGPU 작업에 따라 서로 상이하기 때문에 각각의 스레드가 데이터 객체에 접근하는 방식을 특징화하는 것이 매우 힘들다. 각각의 스레드가 데이터 객체에 어떻게 접근하는지 확인하기 위해서 결국은 커널 코드의 내용을 알아야 한다.

제안하는 기법에서는 커널 코드에 간단한 분석 작업을 수행하기 위해 OpenCL의 온라인 컴파일 방식을 이용한다. OpenCL은 다양한 플랫폼에서 커널 코드를 작동시킬 수 있다. 동일한 프로그램을 다양한 플랫폼에서 실행시키려면 각 플랫폼에 맞는 바이너리 형태의 파일을 모두 프로그램에 준비해야한다. 하지만 OpenCL은 다양한 플랫폼에 맞는 바이너리를 GPGPU 작업 실행 시점에서 생성할 수 있도록 프로그램 실행 시 커널 코드를 빌드하고 컴파일할 수 있는 온라인 컴파일 방식을 지원한다. 따라서 본 논문에서는 OpenCL의 온라인 컴파일 기능을 이용해 가상머신에서 커널 함수 실행 요청이 발생하면 커널 코드를 대상으로 간단한 분석 작업을 수행한다.

GPGPU 작업에서 사용될 데이터 객체들을 스레드 단위로 분할할 때 각 스레드의 ID는 매우 중요한 참조 변수이다. GPGPU 작업을 설계할 때 프로그래머는 커널 함수를 실행하는 수천개의 스레드에게 데이터를 직접 할당하지 않는다. 커널 함수는 사용할 데이터 객체 전체를 매개변수로 입력받고 커널 함수 내부에서는 현재 스레드의 ID를 조회하고 스레드의 ID 참조하여 각 스레드가 사용할 데이터 객체 원소의 인덱

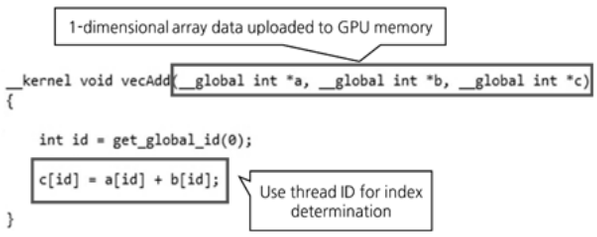


Fig. 6. Correlation between Thread ID and Index of Data Object

스값을 설정하도록 설계된다. Fig. 6은 3개의 데이터 객체를 사용해 덧셈 연산을 수행하는 간단한 커널 코드를 보여준다.

Fig. 6에서 보여주는 것과 같이 각각의 스레드가 사용할 데이터는 스레드의 ID를 기반으로 결정된다. 본 논문의 작업에서도 GPGPU 작업에 사용될 데이터를 스레드 단위로 분할하기 위해 스레드 ID 정보를 사용한다. 데이터 객체를 스레드 단위로 분할하기 위해서는 커널 함수에서 사용하는 데이터 객체 전체에서 각각의 스레드가 데이터 객체의 원소에 접근하기 위해 데이터 객체의 인덱스값을 결정하는 패턴을 찾아야 한다. 즉, 데이터 객체 전체에서 각각의 스레드가 사용할 데이터 일부를 분할하기 위해서는 커널 함수에서 스레드 ID를 기반으로 데이터 객체의 원소에 접근하는 패턴을 찾아야 한다.

본 논문의 커널 코드 분석 작업에서는 스레드의 ID와 커널 함수 내부에서 사용되는 변수를 통해 각각의 스레드가 요구하는 데이터 객체의 인덱스를 추출하여 각각의 스레드가 데이터 객체의 원소에 접근하는 패턴을 분석한다. 커널 함수에서 각각의 스레드가 데이터 객체를 사용할 때 1차원 형태의 배열과 동일한 형태로 사용한다. 커널 함수의 매개변수로 입력 받은 데이터 객체 메인 메모리에서의 형태가 1차원이거나 2차원인 것과 상관없이 GPGPU 연산에 사용되는 데이터 객체의 경우 1차원 데이터 형태로 변환되기 때문에 커널 코드 내부에서 data_object_name[n]과 같이 작성되어 각각의 스레드들은 1차원 인덱스를 사용해 데이터에 직접 접근한다. 데이터 객체를 스레드 단위로 분할할 때에는 각각의 스레드 ID 별로 데이터 객체에 접근하기 위한 1차원 인덱스 값을 찾아야 한다.

커널 코드를 분석 작업은 OpenCL에서 커널 함수를 오브젝트화 하기 위해 커널 코드 파일을 읽어온 후 커널 함수가 호출된 순간 실행된다. 각 스레드의 데이터 객체 접근 패턴을 찾기 위해 우선 커널 코드에서 커널 함수의 매개변수가 호출된 코드 라인을 찾는다. 데이터 객체가 호출되지 않은 코드 라인까지 전부 검색하는 것은 불필요한 분석 작업을 증가시킬 수 있기 때문에 데이터 객체가 호출된 코드 라인을 찾은 후 각각의 스레드가 접근할 데이터 객체의 인덱스를 결정하기 위해 사용된 변수들이 선언되거나 초기화하기 위해 호출된 코드 라인을 찾는 두 단계 분석 작업을 수행한다. 데이터 객체가 호출된 커널 코드 라인을 찾은 후 데이터 객체의 변수화 되어있는 인덱스를 찾고 연산자와 변수를 분류한다. 데이터 객체의 경우 앞서 설명한 것과 같이 1차원 배열 형태로 사용되기 때문에 첨자 연산자(소스 코드의 “[”, “]”와 같은 연산자) 내부의 내용을 분석하면 각 스레드의 데이터 객체 접근 패턴을 확인할 수 있다.

첨자 연산자 사이의 데이터 객체의 인덱스를 추출하면 변수와 산술 연산자를 구분하고 커널 함수의 매개변수를 식별한다. 그리고 마지막으로 그 외의 변수와 변수가 호출된 코드 라인 그리고 첨자 연산자 내부의 산술 연산자의 분석을 통해 각 스레드 ID 마다 데이터 객체의 어떤 원소에 접근하는지 패턴을 분석한다. 스레드 ID는 [0, 0] 부터 차례대로 증가하기 때문에 커널 코드 내부에서 스레드 ID를 조회하는 함수인 ***get_global_id(0), get_global_id(1)***와 매핑된 변수를 통해 스레드의 ID를 추정하고 데이터 객체의 index 값 계산을 위해 사용된 변수와 연산자를 통해 각 스레드 ID 별로 필요로 하는 데이터 객체의 인덱스를 추출할 수 있다.

본 논문에서는 앞서 설명한 방식으로 스레드 ID 별로 스레드가 접근하는 데이터 객체의 인덱스를 추출할 수 있다. 쓰레드 ID별로 데이터 객체의 인덱스가 식별되면 데이터 객체를 스레드 별로 분할하는 작업은 간단하게 처리할 수 있다. 데이터 객체의 분할 작업은 가상머신이 GPU 메모리에 업로드 하기 위해 전송한 데이터 객체가 GPGPUvm에게 전달되면 GPGPUvm은 데이터 객체를 보관하고 있다가 데이터 객체를 GPU 메모리에 업로드하는 API가 요청되었을 때 실행된다.

스레드 단위로 분할된 데이터 객체들을 실제로 GPU 메모리에 업로드할 때 개별적으로 업로드되지 않는다. 데이터 객체와 스레드들은 GPU 메모리에 업로드되고 커널 함수를 실행하기 전에 각 가상머신에게 허용된 GPU 메모리의 용량에 따라 여러 개의 분할된 데이터 객체가 업로드된다. 예를 들어 가상머신이 10MB의 GPU 메모리를 사용할 수 있을 때 각 스레드 단위로 분할된 데이터가 각각 1MB라면 분할된 데이터 객체 10개가 한 번에 GPU 메모리에 업로드 되고 스레드도 10개가 동시에 처리된다. 최근 GPU 메모리의 용량은 일반적으로 GB 단위이기 때문에 각 스레드가 요구하는 데이터 객체의 용량이 매우 큰 경우를 제외하고는 일반적으로 데이터 분할 처리를 수십 번 반복하는 상황이 발생할 확률이 낮다. 또한, 본 논문에서 제안한 데이터 분할 기법은 모든 GPGPU 작업을 분할하지 않는다. GPGPU 작업에서 사용할 데이터 객체의 크기가 사용 가능한 GPU 메모리의 용량보다 작은 경우이다. 이런 경우는 데이터 객체를 분할하지 않고 커널 함수를 실행할 스레드도 분할 처리하지 않는다.

4.2 GPU 메모리 할당 및 해제 기법

스레드 단위로 분할된 각각의 데이터 객체들은 개별적으로 처리되지 않는다. 각 가상머신은 자신이 사용할 수 있는 GPU 메모리 용량에 맞게 분할된 데이터 객체 여러 개를 GPU 메모리에 업로드하고 그에 따라 스레드 그룹을 실행할 수 있도록 커널 함수를 호출할 때 스레드 생성 매개변수를 변경한다. 이러한 방식은 RPC 기반 GPU 가상화 환경 특성상 RPC를 통해 API와 매개변수를 모두 확인할 수 있기 때문에 가능하다. GPU 메모리 할당 기법은 간단한 GPU 메모리 모

니터링을 통해 수집한 정보를 조회하여 해당 가상머신이 GPU 메모리를 얼마나 사용할 수 있는지 확인하고 분할된 데이터 객체의 크기를 확인한다. 그 후에 분할된 데이터 객체의 크기와 사용 가능한 GPU 메모리 용량을 비교하여 분할된 데이터 객체의 그룹 규모를 결정한다. 일반적으로 GPGPU 작업에서 커널 함수를 실행하는 각각의 스레드는 동일한 크기의 데이터를 사용하기 때문에 각각 분할된 데이터 객체는 동일한 크기를 갖고 있기 때문에 GPU 메모리에 업로드할 분할된 데이터 객체의 크기는 비교적 간단하게 결정할 수 있다.

본 논문에서 제안하는 기법에서 RPC를 통해 GPGPU 작업의 요청을 전달받은 GPGPUvm은 GPU 메모리에 업로드할 데이터 객체를 스레드 단위로 분할하고 분할된 객체 그룹을 순차적으로 GPU 메모리에 업로드하기 때문에 커널 함수를 실행하는 스레드들도 여러 번 나눠서 처리된다. 가용 GPU 메모리 용량이 부족하면 GPU 메모리에 데이터 객체 전체를 할당하지 않고 나눠서 할당하기 때문에 GPU 연산에 사용되는 데이터의 크기에 따라 GPU 메모리의 할당과 해제가 여러 번 반복된다. 분할된 데이터 객체는 해당 데이터 객체를 사용하는 스레드의 ID가 낮은 순서대로 GPU 메모리에 업로드 된다. 하지만 본 논문에서 제안하는 GPU 메모리 할당 방식에서는 커널 함수를 여러 번 호출하기 때문에 커널 함수가 호출되고 스레드가 생성될때 마다 스레드 ID가 초기화되기 때문에 커널 함수를 실행시킬 전체 스레드 개수, 현재 실행이 완료된 스레드의 개수 그리고 아직 처리되지 않은 스레드의 개수를 기록하여 GPGPU 작업 처리를 진행하면서 단일 공간의 스레드 ID 처럼 커널 함수를 진행할 수 있도록 커널 함수의 매개변수를 재정의 한다. GPU 메모리 할당 기법은 분할된 데이터 객체를 차례대로 GPU 메모리에 업로드하고 스레드들의 작업이 완료되면 다음 스레드들이 사용할 데이터를 GPU 메모리에 업로드 하기 위해 GPU 메모리에 업로드 했던 데이터를 해제하고 작업 결과를 메인 메모리에 저장한다. 커널 코드는 스레드 ID 순서로 차례대로 실행되기 때문에 연산 결과를 저장하기 위한 출력 데이터 객체의 접근 RPC 통신을 통해 생성된 출력 데이터 객체에 순서대로 저장한다. 본 논문에서 제안한 GPU 메모리 관리 기법을 사용했을 때 GPGPU 작업의 실행 흐름은 Algorithm 1과 같다.

Algorithm 1에서 보여주는 것과 같이 최초 RPC 통신을 통해 가상머신들의 GPGPU 작업 요청이 발생하면 GPU 데이터 분할 기법을 통해 데이터를 분할하고 각각의 데이터를 사용할 스레드 그룹을 분할한다. GPGPU 작업에서 GPU 메모리 입력, GPU 연산 그리고 GPU 메모리 해제 작업은 데이터 객체 전체의 크기/분할된 데이터 객체의 그룹 개수만큼 반복하게 된다. 데이터를 분할하고 GPU 메모리에 데이터를 부분적으로 업로드된 상태에서 GPGPU 작업에서 선언한 모든 스레드를 호출하게 되면 GPU 메모리에 업로드된 데이터에 따라 스레드 대부분은 데이터 객체 원소의 잘못된 인덱스를 참조하여 잘못된 연산 결과를 출력할 것이다. 이로 인해 GPU 메모리에 데이터의 부분업로드를 하기 위해서는 커널 함수를 실행할 스레드의 ID를 기반으로 나눠서 처리해야 한다.

본 논문의 GPU 메모리 할당 기법에서 GPGPU 작업의 데이터 객체는 가상머신이 사용할 수 있는 GPU 메모리 용량 단위로 분할되며, 분할된 데이터 객체는 스레드 ID를 기준으로 차례대로 GPU 메모리에 업로드된다. 예를 들어 가상머신이 사용할 수 있는 GPU 메모리가 50MB라고 가정했을 때 스레드 10개의 ID가 각각 0~9으로 할당되고 각각의 스레드가 사용할 데이터의 용량이 10MB인 경우 GPU 데이터 분할 기법은 데이터 객체를 스레드 단위로 분할한다. 그 후에 GPU 메모리를 할당할 때 50 MB 단위로 GPU 메모리에 데이터 객체 일부를 업로드하고 스레드는 5개씩 처리된다. 분할된 데이터는 스레드의 ID를 기준으로 GPU 메모리에 업로드 된다. 즉, ID가 0인 스레드부터 차례대로 실행된다. 각각의 스레드가 사용할 데이터는 10메가 이기 때문에 총 5개의 스레드가 동시에 실행될 수 있도록 GPU 메모리에 스레드가 사용할 데이터를 준비할 수 있다. 이러한 환경에서 Algorithm 1에서는 스레드 ID 0~4와 5~9를 나눠서 처리하고 각각의 스레드

Algorithm 1

```
1  If User VM requests GPGPUvm for GPGPU task
2    Number of Active VM += 1;
3    Active_VM_List[] =  VM_ID
4    Available GPU memory for each VMs
     = GPU memory Capacity / Number of Active VM
5    if GPU memory allocation operation call
6      wait();
7      if Data upload request to GPU memory
8        Classify the data object as input data;
9      else if
10       Classify the data object as output data;
11     Data_List[Thread_ID][Start_Index][End_Index]
                    = Kernel code analysis;
12     for i= 0 to Number of Thread
13       Partitioned_Data[i][Start_Thread_ID]
                        [End_Thread_ID]
                      = Data_List[Thread_ID][][];
14     if Available GPU memory =< Size of
                    ( Partitioned_Data[j][][] +
                     Partitioned_Data[i + 1][][])
                     / Number of Data Object )
15       break();
16     For j = 0 to Number of Partitioned Data
17       if User VM does not use GPU memory
18         Alloc_GPU_mem( Partitioned_Data[j]
                        [Start_Thread_ID]
                         [End_Thread_ID] );
19         Processing_Kernel();
20         if Kernel processing is complete
21           Free( Partitioned_Data[j][Start_Thread_ID]
                                [End_Thread_ID]  );
```

그룹이 차례대로 처리될 수 있도록 GPU 메모리에 업로드될 데이터의 할당 및 해제 작업을 수행한다.

본 논문의 실험에 사용한 행렬 곱셈 작업은 각각의 스레드들은 데이터 객체의 원소들을 공유한다. 입력 데이터 객체를 읽어올 때 동일한 원소를 읽어오는 스레드가 발생하기 때문에 데이터를 각각의 스레드 단위로 분할하고 GPU 메모리에 업로드할 때 중복 데이터가 발생한다. 본 논문에서는 중복 데이터를 제거하고 GPU 메모리 낭비를 방지하기 위해 각 스레드가 요구하는 데이터 객체의 인덱스를 확인해서 데이터가 GPU 메모리에 중복된 데이터가 업로드되지 않도록 방지한다.

현재 본 논문에서 제안하는 GPU 메모리 할당 기술에서는 앞서 설명한 몇 가지 경우를 제외하고는 각 가상머신이 사용할 수 있는 GPU 메모리 영역을 GPGPU 작업이 요구하는 GPU 메모리양과 상관없이 동등하게 분할한다. 특정 GPGPU 작업의 GPU 메모리 요구량이 GPU 메모리 할당량 보다 작은 경우 요구량에 맞춰서 GPU 메모리를 덜 사용하게 하는 것은 간단하지만 남은 GPU 메모리를 어떤 가상머신에게 추가로 할당하는 것은 전혀 다른 문제이다. GPU 메모리 요구량이 자신에게 할당된 GPU 메모리 용량을 초과한 가상머신이 여러 개 있는 경우 어떤 가상머신에게 GPU 메모리를 추가로 할당할지 결정하기 위해서는 가상머신 사이의 공평성을 고려해야 하며 이는 SLA(Service Level Agreement) 항목, QoS(Quality of Service) 정책 등을 고려한 정밀한 자원 할당 기술이 필요하다. 하지만 본 논문에서 제안한 GPU 메모리 할당 방식은 스레드 단위로 분할된 데이터 객체의 크기를 통해 가상머신이 GPU 메모리에 업로드 할 데이터 객체의 크기를 정할 수 있기 때문에 가상머신의 GPU 메모리 요구량에 맞게끔 GPU 메모리를 할당하는 것도 가능하다.

5. 성능 평가

이번 장에서는 본 논문에서 제안하는 GPU 메모리 관리 기법의 효율성을 평가하기 위한 실험을 수행한다. 실험에서는 다수의 가상머신이 GPGPU 작업을 실행할 때 사용할 수 있는 GPU 메모리를 가용 GPU 메모리 용량에 맞게끔 동일한 크기로 제한하고 GPGPU 작업의 성능을 측정한다. 본 논문에서는 실험을 위해 GPGPU 작업의 규모에 따라 최대 8개의 사용자 가상머신을 사용하며, 각각의 가상머신들은 GPU 메모리 초과 상태를 만들기 위해 다양한 크기의 행렬 곱셈을 수행한다. 이번 장의 실험에서는 각 사용자 가상머신에게 4개의 vCPU와 2GB의 메인 메모리를 할당하였으며, 사용자 가상머신의 GPGPU 작업을 대신 처리하는 GPGPUvm은 메인 메모리 부족 문제를 최소화하기 위해 15GB의 메인 메모리를 할당하고 GPU 메모리가 8GB인 GPU를 할당하였다. 자세한 실험 환경은 Table 2와 같다.

본 논문의 구현에서는 Table 3과 같이 GPU 메모리 사용량과 성능을 참조하여 가용 GPU 메모리의 최대 용량을 10GB로 설정하며, 8개의 가상머신을 실행시킬 때 각 가상머신은 약 1.2GB의 GPU 메모리를 사용할 수 있다. 하지만 본 논문에서는 GPGPU 작업에 사용될 데이터를 분할할 때 데이터의 용량 기반이 아니라 스레드 단위로 분할하기 때문에 데이터를 정확하게 1.2GB 나누는 것은 제약이 있다. 때문에 가상머신이 사용할 수 있는 GPU 메모리의 최대 용량을 1.2GB로 설정하고 남는 GPU 메모리 용량에 대해서는 무시한다. 본 논문에서 제안하는 GPU 메모리 관리 기법은 메모리 초과 상황일 때 GPU 메모리의 전용 영역과 메인 메모리에 존재하는 공유 메모리 영역의 데이터 교환과 GPU 메모리 경쟁을 최소화하기 위해 가상머신들이 사용할 GPU 메모리를 가용 GPU 메모리 영역의 크기에 적합하게 조절한다.

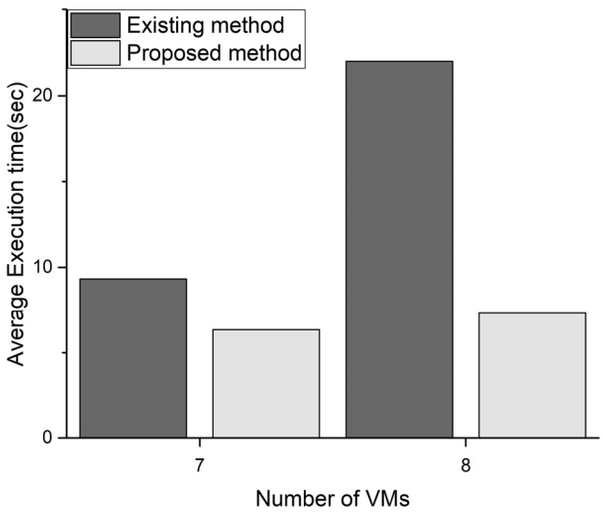
이번 실험의 경우 제한된 GPU 메모리 용량을 사용한 실험이며 본 논문에서 제안한 기법에서는 데이터 분할로 인한 추가 오버헤드와 각 가상머신의 성능이 어느 정도까지 개선되었는지 초점을 둔다. 실험에서는 사용자 가상머신이 사용할 GPU 메모리 용량에 제한을 두지 않은 기존 환경과 본 논문에서 제안한 기법을 사용한 경우의 평균 성능과 각 가상머신의 개별 성능을 측정한다. 각 가상머신의 GPGPU 작업을 최

Table 2. Experiment Environments

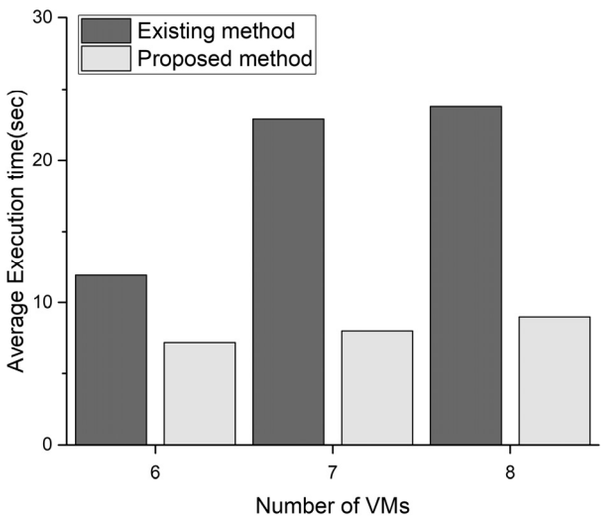
	Host Machine	GPGPUvm	User VM
CPU (vCPU)	Intel Xeon E3-1231V3	4vCPU	4vCPU
Memory	32GB	15GB	2GB
GPU	-	Radeon RX 570 (8GB GPU Memory)	-
OS	Ubuntu 14.04	Windows 7	Windows 7
Hypervisor	Xen 4.4.1	-	-
Calculation Type	-		GPU Matrix Multiplication

Table 3. GPU Memory Usage by Number of VMs and Task Size

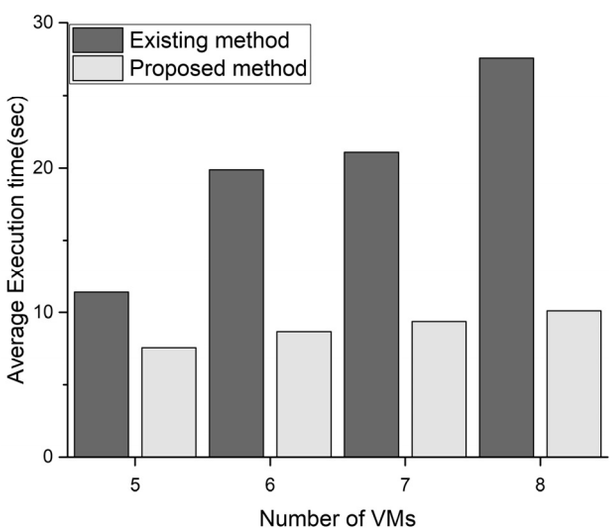
	12K × 12K		13K × 13K		14K × 14K	
Number of VMs	Memory usage (GB)	Average execution time	Memory usage (GB)	Average execution time	Memory usage (GB)	Average execution time
1	1.6093	0.936	1.8887	1.0450	2.1904	1.155
2	3.2186	1.3965	3.7774	1.5760	4.3809	1.88
3	4.8279	1.872	5.6661	2.1633	6.5714	2.2826
4	6.4373	2.1992	7.5548	2.7297	8.7618	4.1107
5	8.0466	2.6206	9.4436	3.2950	**10.9523**	**11.3974**
6	9.6559	5.213	**11.3323**	**11.9240**	13.1428	19.877
7	**11.2652**	**9.32671**	13.221	22.8784	15.3332	21.09
8	12.8746	21.9885	15.1097	23.7691	17.5237	27.5894



(a) Performance of 12K×12K Matrix Multiplication



(b) Performance of 13K×13K Matrix Multiplication



(c) Performance of 14K×14K Matrix Multiplication

Fig. 7. Performance Improvement of GPGPU Tasks through Proposed GPU Memory Management

대한 동시에 실행시키기 위해서 모든 데이터가 GPGPUvm에 전송이 완료된 후에 GPGPU 작업을 실행시킨다. 성능 실험 결과는 Fig. 7과 같다.

Fig. 7에서 보여주는 것과 같이 기존 환경의 경우 GPU 메모리가 초과 사용될 때 성능이 기하급수적으로 저하된다. 예를 들어 Table 3에서 보는것과 같이 12K×12K 행렬 곱셈의 경우 가상머신이 6개가 실행되었을 때와 7개가 실행되었을 때 약 5초에서 약 9초로 2배 가까이 실행 시간이 증가한다. 실험에서 나타난 급격한 성능 저하는 GPU 메모리 경쟁상태가 심해질수록 증가한다. 이는 앞서 설명했던 것과 같이 GPU 메모리 요구량이 증가할수록 전용 메모리 영역과 공유 메모리 영역 사이의 데이터 교환이 증가하기 때문에 발생한다. GPU 메모리는 한정적이지만 데이터를 GPU 메모리에 업로드 요청이 증가하면 전용 메모리 공간과 공유 메모리 공간 사이의 데이터 교환이 많아지기 때문에 이는 GPGPU 작업의 성능에 영향을 미친다.

하지만 Fig. 7에서 보여주는 것과 같이 본 논문에서 제안한 기법은 GPU 메모리 초과 사용을 방지하여 GPU 메모리 경쟁을 완화한다. 이를 통해 GPU 메모리의 전용 영역과 공유 영역 사이의 데이터 교환이 감소하고 각 가상머신의 GPGPU 작업은 데이터 호출로 인한 대기 시간을 최소화 할 수 있다. 그리고 각각의 가상머신이 사용할 수 있는 GPU 메모리 용량을 균등하게 할당하기 때문에 각 가상머신들이 GPU 메모리를 공평하게 사용하도록 지원한다. 또한 GPGPU 작업에 사용될 데이터를 GPU 메모리에 분할해서 업로드하면 GPGPU 작업을 실행하는 스레드도 분할처리 되기 때문에 GPU 코어 경쟁 상태도 간접적으로 피할 수 있다.

실험 결과에서 보여주듯이 본 논문에서 제안한 GPU 메모리 관리 기법은 기존 환경과 비교하여 효과적으로 GPU 메모리 경쟁을 감소시키고 가상머신의 GPGPU 작업의 성능을 향상시킨다. 하지만 Fig. 8과 같이 각각의 가상머신이 요청한

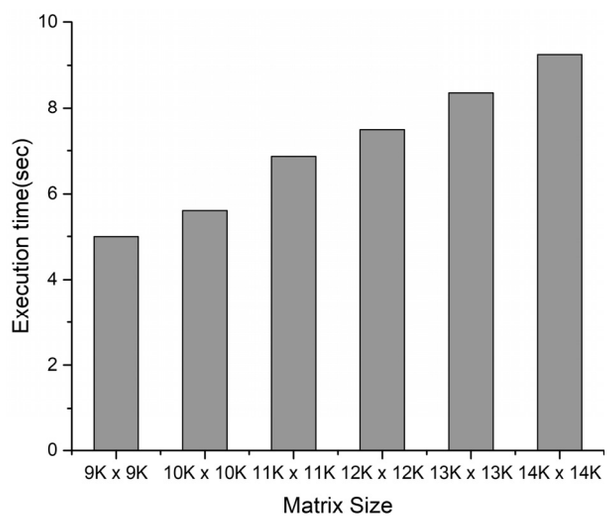


Fig. 8. Performance when Processing Multiple GPGPU Tasks in Sequence

GPGPU 작업을 하나씩 차례대로 처리하는 경우보다 성능이 좋지 않다. 예를 들어 Fig. 7과 8에서 가상머신 8개를 사용한 14K×14K 행렬 곱셈의 성능을 비교하면, 순차적으로 처리한 경우 약 9초의 실행 시간을 갖지만 본 논문에서 제안한 GPU 메모리 관리 기법을 적용한 경우 약 10초의 실행 시간을 갖는다. 본 논문은 각 사용자 가상머신이 GPU 자원을 균등하게 사용해야만 하는 클라우드 환경이기 때문에 GPGPU 작업의 성능을 향상 시키기위한 방법으로 GPGPU 작업들을 차례대로 실행시키는 방법은 고려하지 않는다. 본 논문의 기법은 각 가상머신이 실행하는 GPGPU 작업을 동시에 시작시키고 거의 유사한 시간에 작업이 완료되도록 지원하기 때문에 단순히 차례대로 실행시키는 방법보다 클라우드 환경에 더 적합하다.

6. 관련 연구

GPU 장치를 클라우드 환경에서 효율적으로 공유하기 위한 다양한 기존 연구들이 존재한다. 단일 GPU에서 여러 GPGPU 작업을 동시에 실행하면 GPU 자원 활용률을 향상시킬 수 있다. 그러나 다중 GPGPU 작업 환경은 항상 자원 경쟁이 발생한다. 특히 본 논문의 연구 동기와 같이 GPGPU 작업이 데이터 입출력 작업을 수행할 때 GPU 메모리에 동시에 접근하는 횟수가 증가할수록 자원 경쟁이 심해진다. 또한 가용 GPU 메모리보다 가상머신들의 GPU 메모리 요구량이 큰 경우 이전 장에서 수행했던 실험에서 보여주듯이 GPGPU 작업의 성능이 매우 저하된다. 이전 연구에서는 GPU 메모리 경쟁을 최소화 하기 위해 다양한 기법이 제안되었다.

vCUDA[10]와 rCUDA[11]는 본 연구의 구현 환경과 같은 RPC 기반 GPU 가상화 기법을 제안한 연구이다. GPU가 없는 환경이나 다수의 머신이 단일 GPU를 공유하도록 지원한다. 하지만 기존 RPC 기반 GPU 가상화 기술에서 다수의 GPGPU 작업에 대한 스케줄링은 GPU 장치의 하드웨어 스케줄러에 의존한다. GPGPU 작업을 처리할 때 각 사용자 혹은 머신에서 요청받은 GPGPU API는 FIFO방식으로 처리가 요청되고 API가 GPU에 전달되는 시점부터 작업 처리와 관련된 스케줄링은 GPU에 담당하며, 작업 처리 완료를 대기해야 한다. 또한 기존 기술들은 다수의 사용자 및 머신에게 GPU를 공유하는 것에 초점을 두고 있으며, 단일 GPU 장치에 대해 자원의 초과사용, 균등한 자원 사용에 대한 처리는 관심을 두지 않는다.

GPUswap[12]은 GPU 메모리가 초과 예약 된 경우 GPU 메모리 영역을 메인 메모리 영역으로 확장하고 사용하지 않는 데이터를 메인 메모리로 이동시켜 사용 가능한 GPU 메모리 공간을 확보하기 위한 기법이다. 해당 연구는 여러 GPU 작업이 GPU를 공유할 때 GPU 메모리가 부족해지는 것을 방지하기 위해 GPU 메모리를 가장 많이 사용하는 GPU 응용 프로그램의 데이터 일부분을 메인 메모리로 이동하여 다른 GPU 작업에서 GPU 메모리를 사용할 수 있도록 지원한다. 처리량 지향 GPU 메모리 할당 기법[13]은 GPU 메모리 할당의 병목 현상을 방지하기 위해 여러 GPU 작업이 동시에 실행되는 환경에서 동시성을 향상시키기 위해 동적 GPU 메모리 할당 기술을 제안한다. 동적 GPU 메모리 할당에 Bulk 세마포어를 사용하여 리소스를 동시에 할당 할 수 있다. LD[14]는 GPU 경쟁 조건을 감지하고 확정적(Deterministic) GPU 실행을 가능하게한다. LD는 컴파일러 지원 및 런타임 기술을 사용하여 GPU 리소스 경합을 감지하고 GPU 자원의 비경쟁 실행을 지원한다. GPU 작업의 GPU 메모리 액세스 중요도를 기반으로 GPU 메모리를 관리하는 GPU 메모리 액세스 관리 기법[15]은 3D 장면의 프레임 속도 등 성능값을 기반으로 각 작업의 중요도를 예측하고 중요도에 따라 GPU 메모리 액세스를 스케줄링한다. Mask[16] 및 Mosaic[17]은 메모리 주소 변환으로 인한 오버헤드를 최소화하고 작업이 GPU 메모리에 액세스 할 때 GPU 작업 성능을 최적화한다. 하이브리드 섀도우 페이지 테이블(Shadow page table) 기술[18]은 가상머신의 가상 GPU 메모리와 실제 GPU 메모리 간의 주소 변환 작업을 최적화하여 GPU 작업 성능을 향상시킨다. gScale[19]은 페이지 테이블의 격리를 통해 여러개 가상머신이 GPU 메모리 영역을 동적으로 공유 할 수 있는 기술을 제안한다. 기존 GPU 메모리 관리 기술에는 전체 GPU 가상화 구조를 변경해야 하는 부담이 있다. 또한 제안 된 구조는 특정 GPU 또는 GPU 가상화 시스템 전용이므로 다른 시스템에 적용하는데 제약이 따른다.

앞서 언급한 연구에서는 장치 드라이버 및 운영 체제 커널과 같은 시스템의 여러 구성 요소 변경이 필요하다. 이러한 접근 방식은 여러 구성 요소의 변경으로 인해 시스템 및 클라우드 공급자의 시스템 관리의 복잡성을 증가시킨다. 또한 시스템에 대한 의존도가 높기 때문에 새로운 장치 및 시스템에 대한 적응력이 저하된다. 하지만 본 논문에서 제안한 GPU 메모리 관리 기법은 운영 체제, 하이퍼바이저 및 컴파일러와 같은 시스템 구성 요소에 의존하지 않기 때문에 새로운 GPU 장치에 적용하는데 비교적 간단하다.

7. 결론 및 향후 연구

본 논문에서는 다수의 가상머신이 GPU 메모리를 초과 사용할 때 발생하는 성능 저하 문제를 확인하고 GPU 메모리 경쟁으로 인한 성능 저하를 해결하기 위한 GPU 메모리 관리 기법을 제안한다. 본 논문에서 제안한 GPU 메모리 관리 기법은 가상머신들이 GPGPU 작업에 사용할 데이터를 분할하고 순차 처리하여 GPU 메모리 경쟁 상태를 완화하고 가상머신에서 실행하는 GPGPU 작업의 성능을 향상 시킨다. 또한 데이터를 GPU 메모리에 분할 업로드할 때 GPGPU 작업을 실행하는 스레드들도 분할 처리 되기 때문에 간접적으로 GPU 코어의 경쟁도 완화한다.

본 논문에서 제안한 GPU 메모리 관리 기법은 실험 결과와 같이 다수의 가상머신이 실행하는 GPGPU 작업의 성능을 기존 환경과 향상시킬 수 있었다. 하지만 GPGPU 작업에 사용될 데이터를 분할하고 그룹화하는 추가 작업을 수행하기 때문에 Fig. 7과 같이 여러 개의 GPGPU 작업을 순차적으로 처리하는 경우보다 성능이 좋지 않다. 본 논문의 실험 환경은 GPU 자원의 경쟁 상황을 만들기 위해 다수의 가상머신에서 GPGPU 작업을 동시에 실행한다. 하지만 순차적으로 처리하는 경우 각각의 GPGPU 작업이 차례대로 GPU를 독점하기 때문에 자원 경쟁이 발생하지 않으며 충분한 자원을 사용해 작업을 실행할 수 있다. 하지만 앞서 설명한것과 같이 동시에 요청되는 작업을 순차적으로 처리하기 위해서는 어떤 작업이 먼저 처리되고 어떤 작업이 가장 늦게 처리될지 정해야 한다. 각 가상머신에게 자원을 균등하게 제공해야 하는 클라우드 환경에서 동시에 요청된 작업의 순차적은 처리는 가상머신 사이의 공평성을 저해한다. 때문에 본 논문에서 제안한 데이터 분할 업로드 기법이 클라우드 환경에 더 적합한 기법이다. 또한 기존 클라우드 환경의 CPU 자원처럼 시분할 기반으로 GPGPU 작업을 스케줄링하는 방법도 있지만, 관련 연구에서 설명한 것과 같이 OS, GPU 드라이버, 하이퍼바이저 등 시스템 구성요소의 수정이 필요하기 때문에 시스템 복잡도가 증가하는 문제가 발생한다.

본 논문의 구현과 실험에서는 각 가상머신에서 동일한 규모의 GPGPU 작업을 수행시키고 모든 가상머신이 동일한 크기의 GPU 메모리를 사용한다고 가정한다. 때문에 각 가상머신에서 실행하는 GPGPU 작업에 사용될 데이터를 동일한 크기로 분할하는 방법을 사용할 수 있었다. 하지만 각 가상머신에서 실행하는 GPGPU 작업의 규모가 다르거나 각 가상머신이 사용할 수 있는 GPU 메모리의 용량이 서로 다르거나 실행 중인 가상머신의 개수가 동적으로 변한다면, 데이터 분할에 다양한 요소를 추가해야 한다. 추후 연구로는 실행 중인 가상머신의 개수, 가상머신의 GPU 메모리 요구량과 같은 정보를 고려할 수 있도록 GPU 메모리 관리 기법을 확장할 것이다.

References

[1] Amazon, Amazon EC2 Instance Types [Internet]. https://aws.amazon.com/ec2/instance-types/?nc1=f_ls.

[2] Alibaba Cloud, Elastic GPU Service [Internet], https://hpc.aliyun.com/product/gpu_bare_metal.

[3] NVIDIA, NVIDIA GRID [Internet], https://www.nvidia.com/ko-kr/data-center/virtual-gpu-technology/

[4] AMD, AMD Radeon Pro [Internet], https://www.amd.com/ko/graphics/workstation-virtualization-solutions-csp

[5] AMD, OpenCL: Open Computing Language [Internet], https://www.khron os.org/opencl/.

[6] NVIDIA, CUDA: Compute Unified Device Architecture [Internet], http://www.nvidia.com/object/cuda_home_new. html.

[7] NVIDIA, NVIDIA V100 [Internet], https://www.nvidia.com/ko-kr/data-center/v100/

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03. ACM: New York, NY, USA, 2003, pp.164-177.

[9] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger,G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed I/O," *Intel Technology Journal*, 2006.

[10] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, Vol.61, No.6, pp.804-816, 2012.

[11] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ort, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," *High Performance Computing and Simulation*, pp.224-23, 2010.

[12] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," *ACM SIGPLAN Notices*, Vol.50, No.7, pp.65-77, 2015.

[13] I. Gelado, "Garland M. Throughput-oriented GPU memory allocation," In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pp.27-37, 2019.

[14] P. Li, X. Hu, D. Chen, J. Brock, H. Luo, E. Z. Zhang, and C. Ding, "LD: Low-overhead GPU race detection without access monitoring," *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol.14, No.9, pp.1-25, 2017.

[15] S. Rai and M. Chaudhuri, "Using criticality of GPU accesses in memory management for CPU-GPU heterogeneous multi-core processors," *ACM Transactions on Embedded Computing Systems (TECS)*, Vol.16, No.5s, pp.1-23, 2017.

[16] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *ACM SIGPLAN Notices*, Vol.53, No.2, pp.503-518, 2018.

[17] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a GPU memory manager with application-transparent support for multiple page sizes," In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.136-150, 2017.

[18] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi, and H. Guan, "Boosting GPU Virtualization Performance with Hybrid Shadow Page Tables," *USENIX Annual Technical Conference*, pp.517-528, 2015.

[19] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, S. Jiao, B. He, and H. Guan, "gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space," *USENIX Annual Technical Conference*, pp.579-590, 2016.

강 지 훈

https://orcid.org/0000-0003-4773-6157

e-mail : k2j23h@korea.ac.kr

2011년 배재대학교 게임공학과(학사)

2013년 배재대학교 게임멀티미디어공학과 (석사)

2020년 고려대학교 컴퓨터학과(박사)

2020년 ~ 현 재 고려대학교 정보대학 4단계 BK21 컴퓨터학 교육연구단 박사후과정생

관심분야 : Cloud Computing, GPU Virtualization, HPC Cloud