석 사 학 위 논 문

범용 목적 GPU 프로파일링 기반 다중 작업 배치 기법

Multi-Task Placement Techniques based on Profiling General Purpose GPUs

숙명여자대학교 대학원

컴퓨터과학과 컴퓨터과학전공

김 세 진

석 사 학 위 논 문

범용 목적 GPU 프로파일링 기반 다중 작업 배치 기법

Multi-Task Placement Techniques based on Profiling General Purpose GPUs

숙명여자대학교 대학원

컴퓨터과학과 컴퓨터과학전공

김 세 진

석 사 학 위 논 문

범용 목적 GPU 프로파일링 기반 다중 작업 배치 기법

Multi-Task Placement Techniques based on Profiling General Purpose GPUs

지도교수 김 윤 희

이 논문을 공학 석사 학위 논문으로 제출함

2020년 12월

숙명여자대학교 대학원

컴퓨터과학과 컴퓨터과학전공

김 세 진

김세진의 공학 석사 학위 청구 논문을 인준함

범용 목적 GPU 프로파일링 기반 다중 작업 배치 기법

Multi-Task Placement Techniques based on Profiling General Purpose GPUs

2020년 12월

심사위원장 (인)

위 원 (인)

위 원 (인)

숙명여자대학교 대학원

목 차

표 목 차

그 림 목 차

국 문 요 약

GPU (Graphics Processing Unit)의 빠른 병렬 연산 능력은 그래픽 연산뿐 아니라 머신 러닝 (ML: Machine Learning), 고성능 컴퓨팅 (HPC: High Performance Computing) 응용 등에서 널리 사용된다. 이에 따라 데이터 센터 및 클라우드 환경에서 GPU 기반 인프라 서비스를 제공하기 시작했다. 실제 GPGPU (General Purpose GPU) 응용은 GPU 친화적인 응용과 달리 낮은 GPU 활용도를 보인다. 이로 인해 높은 가격을 가지는 GPU의 자원 활용도를 높이기 위해 서로 다른 응용들을 동시에 돌리고자 하는 요구가 발생하였다. 그러나 GPU 상에서의 다중 작업은 공유하는 자원에 대해 경쟁이 일어나 성능 저하가 발생할 수 있다. 또한 GPU 다중 작업 기술에 대한 연구가 CPU 기반 다중 작업에 비해 미비하므로 GPU 자원 및 응용 특성에 기반한 다중 작업 배치 기법이 필요하다.

본 연구에서는 연산 단위인 SM (Streaming Multiprocessor)을 기준으로 다중 SM 집합의 공유 자원 및 SM 내부 자원 계층을 고려한 다중 작업 배치 기법을 제안한다. 다중 SM 집합 단계에서는 응용 프로그램 특성 및 자원 사용 패턴 기반의 학습을 통한 다중 작업 배치 기법에 대해 소개한다. 또한, SM 내부 단계에서는 응용 프로그램의 SM 내부 자원 사용 패턴을 기반으로 한 다중 작업 배치 기법을 제안한다. 실험 결과는 존재하는 다른 접근 방법과 비교하여 해당 방법의 우수성을 보인다. 이는 본 논문에서 제안하는 기법이 자원의 경쟁을 최소화하여 자원의 활용도를 높이고 결과적으로 전체 워크로드의 처리량을 높이며 개별 성능도 보존할 수 있음을 보인다.

주제어: GPU, 다중 작업 배치, 자원 관리, 자원 공유, 응용 특성 분석

I. 서 론

1. 연구 배경

GPU(Graphics Processing Unit)는 수천 개의 프로세싱 코어를 통해 대량의 병렬 처리 연산을 수행한다. 이러한 빠른 병렬 연산 능력을 그래픽 연산뿐 아니라 일반 컴퓨팅 영역으로 확장한 것을 GPGPU(General-Purpose GPU)라 한다. 이는 신호 처리, 데이터 마이닝, 생체 의학 시뮬레이션, 검색 및 게임 등의 딥 러닝(DL: Deep Learning), 고성능 컴퓨팅(HPC: High Performance Computing)과 같은 폭 넓은 분야에서 적용된다. 특히, 최근 몇 년 동안 증가하는 컴퓨팅 수요로 인해 GPU는 성능 요구사항을 충족하고 총 소유 비용(TCO: Total Cost of Ownership)을 절감하고 있다.

1년에 두 번 가장 빠른 슈퍼컴퓨터를 발표하는 TOP500[6]은 고성능 컴퓨팅의 최신 트렌드를 추적할 수 있도록 한다. 2020년 6월에 발표된 자료에 따르면 전체 500개의 슈퍼컴퓨터 중 141대의 컴퓨터가 GPU를 가속기로 채택하고 있으며, 최상위 10대의 컴퓨터 중에서는 7대가 GPU를 가지고 시스템을 구성한다. 또한, 이는 2017년의 약 71대, 2014년의 43대와 비교하여 꾸준히 증가하는 추세를 보여준다. 이에 따라 사설 데이터 센터 (예시: 구글의 [1])와 여러 대규모 공용 클라우드 (예시: Amazon EC2 [2], Nimbix [2], Peer1 Hosting [3], Microsoft Azure [4])는 GPU 클라우드를 지원 하기 위해서 GPU 기반 인프라 서비스를 제공하기 시작했다.

대규모 응용 프로그램은 컴퓨팅 집약적이므로 고가의 GPU에 크게

의존한다. 사용자의 입장에서 클라우드의 GPU 인스턴스는 일반 인스턴스보다 거의 10배에 해당하는 비용이 소요된다. 수 만개의 GPU로 구성된 클러스터를 관리하는 데이터 센터 및 클라우드 서비스 제공자는 GPU를 효율적으로 활용하기 위해 클러스터 스케줄러를 사용한다. 이들이 스케줄링을 통해 달성하고자 하는 주요 목표는 자원 활용률은 높이고, 실행 시간은 감소시키며, 높은 시스템 처리량을 달성하는 것이다. 이와 같이 사용자 및 서비스 제공자 입장에서 자원 활용도를 높이는 것은 중요하다.

점점 더 많은 자원이 GPU로 통합되는데 반하여, 단일 응용 프로그램이 항상 모든 자원을 완전히 활용할 수 없다. 일례로, 현재 TOP500 [6] 목록의 2위를 차지하고 있는 Summit은 GPU 친화적인 LINPACK [7] 벤치마크를 실행할 때에는 CPU 노드에서 달성한 최고 성능의 65%를 향상한다. 이에 반해, HPCG [8] 벤치마크에 대해서는 최고 성능의 1.5%만 달성하며 실제 응용프로그램에 대해서는 훨씬 낮은 GPU 사용률을 보이는 것을 알 수 있다 [9]. 따라서 자원 활용도를 높이기 위한 방안으로써 응용 프로그램의 공동 수행이 제안되었다.

이에 따라 GPU 활용률을 향상시키기 위해 GPU 다중 작업(multi-tasking)은 학계와 업계에서 모두 주목을 받기 시작했다. CPU 서버에서 동시 수행을 통해 자원 활용도를 높이면서 처리량을 최대화하고자 하는 연구들은 많이 존재한다 [10-12]. CPU에서 응용 프로그램은 코어, 공유 캐시 및 주 메모리 대역폭을 두고 경쟁한다. 하지만 GPU에서의 동시 수행은 호스트와 디바이스 간 메모리 복사, GPU 전역 메모리 시스템, 공유 메모리 시스템, 컴퓨팅 유닛 등 CPU와 다른 요소에 대해서 경쟁이 일어나므로 이전 연구들이 적용되지 않는다. GPU 응용의 동시 실행을 달성하기 위해서 제안된 단순한 방법은

Left-over 전략을 취하는 것이다. 이는 한 응용에 가능한 리소스를 최대한으로 할당하고 남는 자원을 다른 커널에 할당하는 전략이다. 그러나 단순한 Left-over 전략으로는 자원 활용을 최적화하지 못하고 응용을 단독으로 실행한 것과 비교하여 공동 실행 성능에 저하가 나타날 수 있다. 따라서 GPU 클라우드에서 자원 경쟁으로 인하여 발생한 간섭은 (1) 사용자가 사용하지 않는 자원에 대한 요금이 청구되고, (2) 예측할 수 없는 서비스 수준이 발생할 수 있다는 문제를 발생시킬 수 있다.

GPU 다중 작업에는 시간 다중화 (Temporal multi-tasking) 기법과 공간 다중화 (Spatial Multi-tasking) 기법이 존재한다. 전자는 GPU 커널이 GPU에서 순차적으로 실행되며, 후자는 여러 커널이 GPU 내의 분할된 공간을 최대한으로 활용하도록 한다. 높은 GPU 사용률 및 처리량을 달성하기 위해서는 두 가지 방법을 함께 사용하여 다중의 SM이 공유하는 자원 및 SM 내부 자원을 모두 활용해야 한다. 본 논문을 이를 각각 Intra-SM, Inter-SM이라고 명명한다.

본 논문에서는 GPU의 공유 자원 활용도를 높이기 위하여 범용 목적의 응용을 대상으로 Inter-SM / Intra-SM 두 계층의 자원 사용 특성에 기반한 다중 작업 배치 기법을 제안한다. Inter-SM 계층의 공동 실행을 위하여 응용 프로그램 특성 및 자원 사용 패턴을 고려한 기계 학습 모델을 개발한다. 학습을 활용하여 Inter-SM 자원을 공유하는 스케줄링 기법을 소개하고, 응용 프로그램의 Intra-SM 자원 사용 패턴을 기반으로 한 스케줄링 기법을 제시한다. 각 스케줄링 기법의 성능을 평가하기 위해 최신 연구의 스케줄러와 비교를 진행하였을 때, 본 논문의 기법이 자원 활용도를 높임으로써 응용 프로그램의 성능을 향상시킬 수 있음을 보인다.

2. 연구의 기여

본 연구의 최종목표는 GPU의 공유 자원 활용도를 높이기 위하여 GPU 클러스터 환경에서 범용 목적 응용 프로그램인 HPC 및 DL 등의 응용 프로그램을 대상으로 응용프로그램의 프로파일링에 기반한 다중 작업 배치 기법을 제안한다. 본 논문의 주요 기여 내용은 다음과 같다.

- GPU 클러스터에서 응용들 (HPC, DL)의 자원 (GPU, GPU Memory, PCIe) 사용 특성이 다름을 분석하고 이를 통해 서로 다른 간섭이 발생할 수 있으며 프로파일링 및 모델링의 필요성을 설명한다.
- 수집한 응용의 프로파일링 데이터를 사용하여 응용 간의 간섭을 기계 학습을 사용하여 예측한다.
- 기계 학습 모델을 기반으로 한 간섭 인식 스케줄러인 Co-ScheML 을 제안한다.
- 다양한 GPGPU 응용을 대상으로 SM 내부 자원 사용량 및 런타임 수행 특성이 이질적이며, 이전 연구가 효율적인 자원 분배 방법을 찾을 수는 있으나 동시 수행에서 발생할 수 있는 자원의 경쟁은 예측할 수 없음을 보여준다.
- 응용의 개별 실행 특성에 따라 응용을 분류하며, 이를 바탕으로 한 성능 포화 지점 및 동시 실행 특성을 관찰한다.
- 관찰에 기반하여 Intra-SM 자원의 낮은 활용도 문제를 해결하기

위한 스케줄러인 K-Scheduler를 제안한다.

3. 논문의 구성

본 논문의 구성은 다음과 같다. 2장에서는 GPU의 다중 작업 배치 기법을 위한 배경을 포괄적으로 제시한다. 3장에서는 Inter-SM 자원 공유를 위한 기계 학습 모델 기반 간섭 인식 스케줄러인 Co-scheML을 소개한다. 4장에서는 Intra-SM 자원 공유를 위한 응용 특성 기반 스케줄러인 K-Scheduler를 설명한다. 마지막으로 6장에서는 결론을 맺는다.

II. 연구 배경

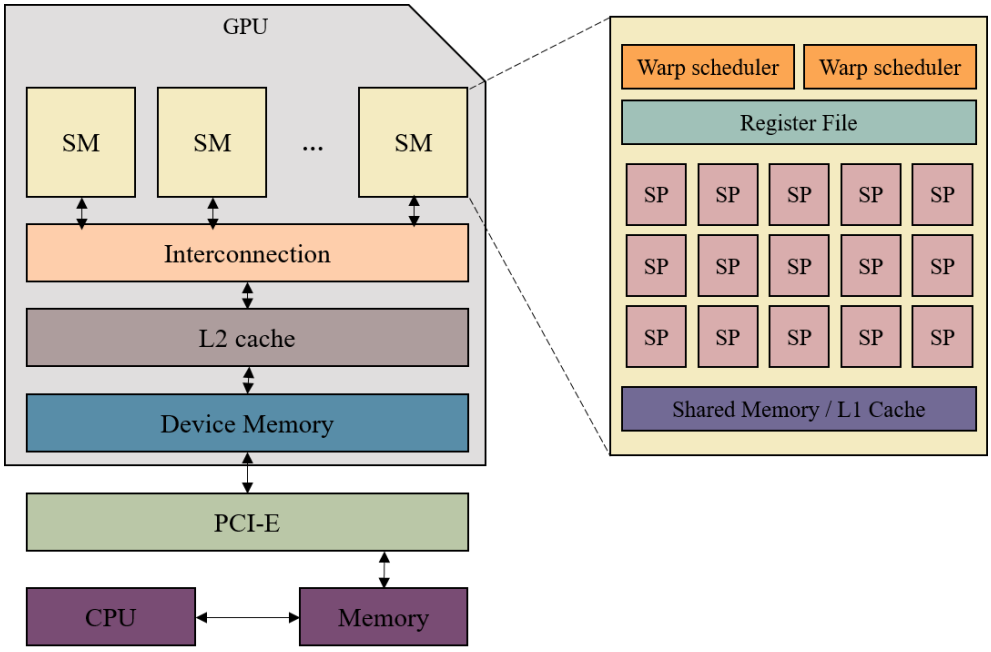
1. GPU 실행 모델

일반적으로, GPU 프로그램은 호스트에서 실행되는 호스트 코드와 GPU 디바이스에서 실행되는 디바이스 코드로 이루어져 있다. 호스트 코드는 CPU 연산과 CPU-GPU 간 데이터 통신을 나타내는 메모리 복사 (copy)를 포함하며, 디바이스 코드는 GPU 상에서 수행되는 함수의 단위인 커널(kernel)로 나타낸다. 프로그래머는 커널 코드를 작성하기 위해서 SIMT(Single Instruction Multiple Threads) 모델을 사용한다. SIMT 프로그래밍 모델에서, 스레드(thread)는 실행의 기본 단위이다. 커널 호출과 동시에 수천 개의 스레드를 생성하여 각 스레드는 GPU 상에서 동일한 코드를 병렬적으로 처리하게 된다. GPU 프로그래밍 모델에서 스레드는 계층적으로 스레드 블록(Thread block: TB)으로 그룹화 되며, 여러 개 의 스레드 블록이 모여 그리드(Grid)를 형성한다.

NVIDIA GPU 하드웨어에는 워프(Warp)로 불리어지는 또 다른 스레드의 그룹이 있다. 이는 GPU 프로그래밍 모델에 직접적으로 노출된 개념은 아니지만, 프로그래머는 GPU 연산 처리가 워프 단위로 동시에 처리되는 특성을 활용하여 GPU 프로그램을 최적화 할 수 있다. 최근의 GPU에서 워프는 32개의 스레드로 구성된다. GPU 상에서 동시에 동작하는 워프(혹은 스레드 블록)의 개수는 레지스터의 개수, 공유 메모리 크기, 최대 스레드 블록 개수와 같은 GPU 자원에 의해서

제한된다.

2. GPU 구조



<그림 1> GPU의 구조

그림 1은 GPU 시스템의 전체적인 구조를 보여준다. 각 GPU는 PCI-E 버스를 통해서 CPU와 연결된다. GPU의 내부에는 스트리밍 멀티프로세서 (Streaming Multiprocessor: SM)와 L2 캐시, GPU DRAM이 있으며 SM은 상호 연결 네트워크를 통해 디바이스 메모리를 공유한다. SM은 CPU의 코어와 비슷한 역할을 하는 주요 실행 단위이다.

각 SM은 스트리밍 프로세서 (Streaming Processor: SP), 공유 메모리, 레지스터, L1 캐시로 구성된다. SM 내부에는 하나 이상의 워프 스케줄러를 가지고 있어 명령을 가져오고, 디코딩하고, 실행한다. 각 워프 스케줄러는 워프 크기와 동일한 32 개의 ALU를 관리한다.

SM은 워프에서 다른 워프로 전환할 때에 컨텍스트 전환 오버헤드 없이 전환할 수 있다. 이로 인해 워프 스케줄러는 메모리 작업 등의 이유로 하나의 워프가 중단되었을 때 다른 워프로 스위칭 하여 워프의 지연 시간을 숨길 수 있다. 이렇게 하나의 SM이 동시에 관리하는 워프를 '활성화 워프'라 하며 '활성화 워프'의 개수는 커널 함수의 자원 사용량 및 하드웨어 제약에 의해 정해진다. 하지만 모든 '활성화 워프'가 반드시 다음 명령을 발행할 수 있는 것은 아니다. 배리어(barrier) 혹은 이전 명령의 결과를 기다려야 하는 등의 이유로 '활성화 워프'가 명령을 발행하지 못한다면 이를 '실행 지연'(stall)이라고 한다. 이와 반대로 명령을 발행 할 수 있다면 이를 '적격 워프'(eligible warp)라고 한다. 즉, 활성화 워프 수는 '실행 지연'된 워프 수와 '적격 워프' 수의 합이라고 할 수 있다.

3. GPU 다중 작업

최신의 GPU는 단일 GPU 하드웨어에서 여러 GPU 커널을 동시에 실행할 수 있도록 하여 GPU 하드웨어에 대한 공간 다중화를 가능하게 한다. NVIDIA의 Hyper-Q 기술 [13] 및 AMD의 대기열 기반 다중 프로그래밍 기술이 [15] 이에 해당한다. NVIDIA의 GPU 프로그래밍

API인 CUDA는 GPU 실행에 대한 컨텍스트를 캡슐화 한 CUDA Context를 생성하면서 시작하는데, Hyper-Q 기술은 같은 CUDA Context에 있는 kernel들만을 동시에 수행 가능하도록 한다. 즉, 다른 응용의 커널은 동시 수행이 불가능하므로 NVIDIA에서는 Multiple Process Service(MPS)[16] 를 제공하여 여러 응용이 동시 수행가능하도록 하였다. 한편, MPS는 한 커널에 가능한 리소스를 최대한으로 할당하고 남는 자원을 다른 커널에 할당하는 단순한 Left-over 전략을 취한다. 그러므로 앞 커널이 자원을 많이 사용하고 시간이 오래 소요된다면, 뒤의 커널은 블로킹(blocking) 된다.

Warped-slicer [17]는 최신의 Intra-SM 공유 체계이며, 동시 커널 간의 스레드 블록 분할을 결정하기 위해 워터-필링(Water-filling) 알고리즘을 사용한다. 이는 활성 스레드 블록에 대한 성능을 나타낸 확장성 곡선(Scalability curve)을 기반으로 SM 내에서의 스레드 블록 파티션을 결정한다. 이 때, 오프라인 및 온라인 프로파일링을 통해 각 커널의 성능 저하를 최소화하는 스레드 블록 파티션을 스위트 포인트 (Sweet point) 로써 식별한다.

SmCompactor[18]는 실제 하드웨어에서 Intra-SM 공유를 가능하게 하는 스레드 블록 기반 스케줄링 프레임워크이다. 이는 먼저 커널의 정적 및 동적 기능을 프로파일링한 후 이 정보에 따라 스레드 블록을 특정 SM에 매핑하는 디스패치 정보 (Dispatch information)를 결정한다. 디스패치 정보는 영구 스레드 모델 (Persistent thread model)에 따라 커널을 변환하는 커널 변환 모듈에 통합되고, 이는 각 스레드 블록을 원하는 SM에 제출할 수 있도록 하여 응용의 Intra-SM 다중 작업을 가능하게 한다.

4. 클러스터 오케스트레이션 플랫폼에서의 GPU 공유

가상 환경에서 GPU를 공유하기 위해 물리 GPU를 가상화한다. NVIDIA는 가상 머신(Virtual Machine: VM)에서 GPU를 사용 가능하게 하기 위해 NVIDIA vGPU 기술을 제공한다. 하이퍼바이저 (Hypervisor) 레이어에 함께 설치되는 NVIDIA vGPU 소프트웨어는 각 VM에 vGPU를 생성하여 여러 VM이 물리 GPU를 공유하도록 한다. OpenStack[19]은 vGPU 스케줄링을 제공하며, 이는 사용자가 요구한 vGPU 양에 맞게 vGPU 인스턴스를 생성한다. 하이퍼바이저가 GPU의 제어 경로(Control path)를 관리하여 GPU 드라이버가 vGPU에 직접 접근할 수 있게 한 NVIDIA의 Mediated pass-through 기술은 기존의 기술과 비교하여 성능 오버헤드를 줄였다. 하지만 서비스 제공자는 여전히 GPU를 위한 새로운 스케줄링 메커니즘이 아닌, 기존의 필터 기반 스케줄러가 사용자의 요구에 맞게 vGPU를 할당하는 방식을 사용한다.

한편, 호스트 시스템의 커널 호출 (Kernel call) 을 사용하여 오버헤드를 줄인 소프트웨어 컨테이너의 사용이 늘어나고 있다. GPU를 사용하는 컨테이너 응용이 증가함에 따라 YARN[20], Kubernetes[21]와 같은 컨테이너 오케스트레이션 플랫폼에서도 GPU를 제공하기 시작했다. 하지만 YARN, Kubernetes 모두 GPU 스케줄링에 있어 GPU를 단순히 확장된 자원으로 고려하여 개별 컨테이너는 GPU를 독점적으로 사용하게 하므로 GPU의 자원 낭비를 초래한다.

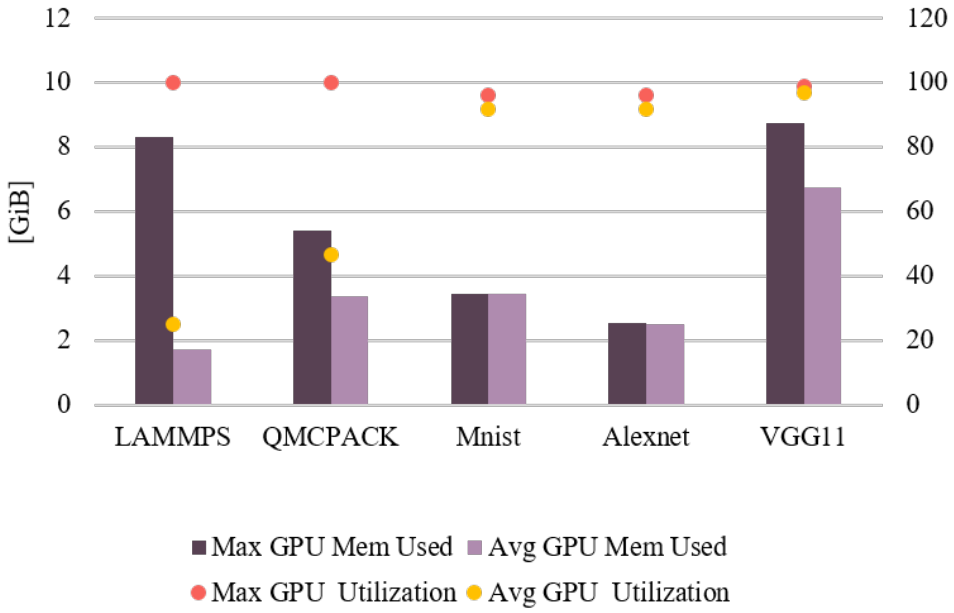
III. 기계 학습 모델 기반 Inter-SM 자원 공유 다중 작업 배치 기법

1. 연구 동기

1) GPU 자원의 오버 커밋

GPU 클라우드 환경에서 GPU를 가상화하고자 하는 노력들이 있었다[22-24]. 이러한 노력에도 불구하고 CPU와 GPU의 자원 통합 성능 간에는 상당한 불균형이 존재하므로 그 격차를 메우는 것은 여전히 어려운 과제이다. 그림 2를 통해 이 문제를 간략히 설명한다. 그림 1은 HPC 응용인 LAMMPS[25], QMCPACK[25]과 DL 모델인 Mnist, Alexnet, VGG11[26]을 NVIDIA TITAN XP GPU 및 i7-5820K CPU 환경에서 실행하였을 때의 GPU 활용도(GPU utilization)와 메모리 사용량(GPU memory used)을 보여준다. GPU 메모리 자원의 경우 CPU와 같은 수준의 메모리 가상화 기술을 제공하지 못하므로 실행 중인 응용의 메모리 사용량 합이 실행 환경의 메모리보다 크다면 Out Of Memory(OOM) 실패가 발생한다. 이를 방지하기 위해서 사용자는 응용의 메모리 요구사항을 제출할 때에 최대 사용량을 요구한다. GPU 컴퓨팅 자원의 경우에도 사용자는 응용의 성능 보장을 위해 GPU 활용도의 최대치를 요구한다. 하지만 그림 2와 같이 평균 값과 최대 값 사이의 차이는 GPU 메모리의 경우 평균 약 54%, GPU 계산 자원의

경우 평균 약 51%를 달성한다. 그러므로 사용자의 요구사항대로 자원을 할당한다면 GPU 자원이 낭비될 수 있다. 하지만 자원의 낭비를 방지하기 위해 평균치를 고려한 스케줄링이 이루어진다면 각 응용이 자원을 최대로 많이 사용하는 구간이 겹침으로써 OOM 실패 및 성능 저하의 문제가 일어날 수 있다. 본 실험결과는 GPU 응용의 프로파일링 및 모델링을 통해 자원의 오버 커밋은 방지하되, OOM 실패 및 심각한 성능저하가 일어나지 않도록 하는 것이 필요함을 보여준다.



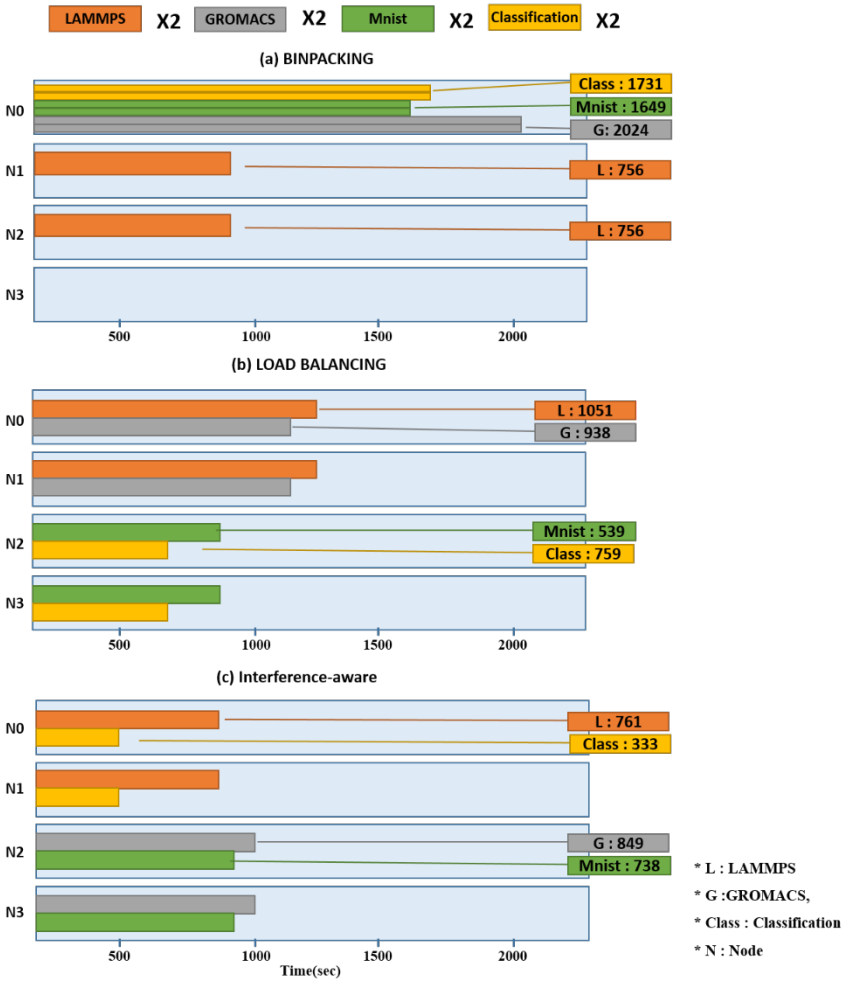
<그림 2> GPU 응용의 자원 오버 커밋 그래프

2) 간섭을 고려한 스케줄링의 필요성

그림 3은 작업의 풀에 LAMMPS, GROMACS, Mnist, Classification

[27] 응용이 각각 두 개씩 있다고 가정하였을 때, 세 개의 정책에 따른 스케줄링을 비교하는 샘플 실험을 보여준다. 그림 3-(a)는 최대 메모리 사용량을 고려한 빈 패킹(Bin packing) 스케줄링 레이아웃이다. 이는 작업을 배치하기에 충분한 가용 자원이 있는 노드 중 가장 자원 사용량이 높은 노드를 선택하여 사용되는 노드의 수를 최소화하는 정책이다. 그림 3-(b)는 단순히 평균 GPU 활용도 만을 고려하여 부하를 분산시키는 방법인 로드 밸런싱 (Load balancing) 기법을 보여준다. 이는 최대 GPU 활용도를 가지는 응용과 최소 GPU 활용도를 가지는 응용을 함께 배치한다. 그림의 하단은 간섭을 고려한 스케줄링 레이아웃이다. 각 응용의 쌍들을 동시 실행한 뒤 단독으로 실행한 시간과 비교하여 간섭 값을 계산한다. 얻어진 간섭 값을 기반으로 하여 간섭 값이 최소화 되는 쌍을 선택하여 실행한다. 전체 워크로드의 수행 시 간섭을 고려한 스케줄링을 각각 빈 패킹 전략과 로드 밸런싱 전략과 비교하였을 때 전체 수행 시간(Makespan)이 2.38배, 1.23배 줄어든 것을 알 수 있다. 각 응용의 성능을 비교해보아도 LAMMPS의 성능이 빈 패킹 전략보다 약 0.6%, Mnist의 수행 성능이 로드 밸런싱 전략과 비교하여 약 36% 성능이 악화된 것을 제외하면 LAMMPS, GROMACS, Mnist, Classification 성능이 각각 최대 38%, 138%, 123%, 419% 향상하였다. 결과적으로 간섭 인지 전략을 스케줄링에 적용하였을 때 응용의 평균 작업 완료 시간(Average job completion time)이 다른 정책과 비교하여 74%, 22% 향상된다(670.25s vs 1168.75s vs 821.75s). 이 결과는 최대 메모리 사용량 만을 고려한 빈 패킹 방식이 사용 가능한 자원을 모두 사용하지 않고, 응용 간 발생할 수 있는 간섭에 대한 고려 없이 노드 수만을 최소화 시켜 성능이 나빠짐을 보여준다. 또한, 노드 간의 부하를 분산시켜 단순하게 간섭을 축소하려는 방법은 빈 패킹 방식보다는 성능이 좋아지지만 성능 향상에

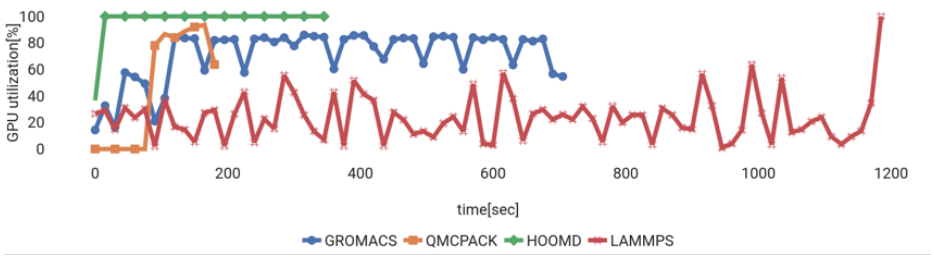
제약이 있는 것을 보여준다. 이를 통해 사용 가능한 자원을 최대한으로 사용하며 간섭을 최소화 시키는 스케줄러를 설계 해야함을 동기 부여 받았다.



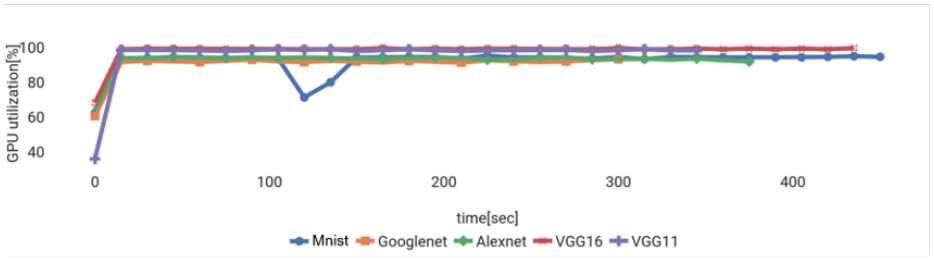
<그림 3> 스케줄링 정책에 따른 작업 배치

2. 응용의 런타임 GPU 자원 사용 특성 분석

1) GPU 활용도



(a) GPU utilization pattern of HPC applications



(b) GPU utilization pattern of DL training applications

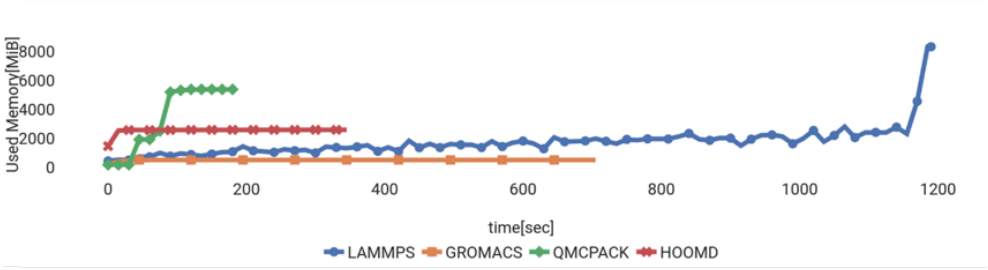
(c) GPU utilization pattern of DL inference applications

<그림 4> 응용의 GPU 활용도 패턴

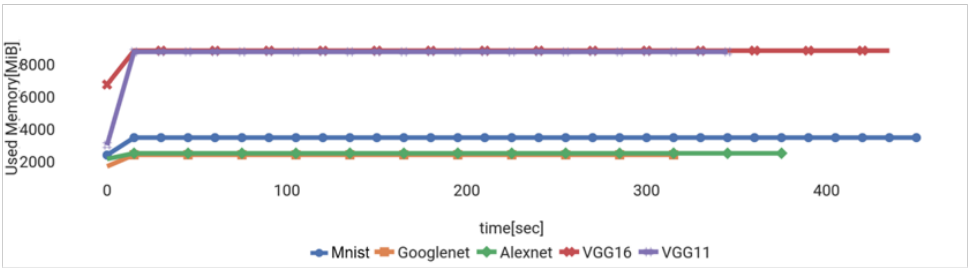
그림 4는 응용의 범주 별 시간에 따른 GPU 활용도를 보여준다. 그림 4-(a)와 같이 HPC 범주의 응용은 HOOMD를 제외하고, 모두 동적인 GPU 활용도를 보인다. QMCPACK 과 LAMMPS의 경우 응용의 수행 후반부에서 GPU 활용도가 급격하게 증가하는 모습을 보인다. 이들은 수행 동안 호스트-디바이스 간 메모리 전송이 빈번하게 일어나 활용도가 급격하게 증가하기 이전에는 낮은 활용도를 보인다. GROMACS의 경우 전반적으로 높고 비교적 일정한 활용도를 보이지만 수행 중간에 메모리 전송과 같은 CPU 작업으로 활용도가 감소하는 순간이 존재한다. 딥 러닝의 훈련(DL training) 작업은 그림 4-(b)와 같이 정적이며 높은 GPU 활용도를 보여준다. 딥 러닝 모델의 종류와 함께 최적의 파라미터를 찾는 하이퍼 파라미터(Hyper-parameter)는 모델의 수렴, 정확도 등에 영향을 미치므로 딥 러닝 관련 연구에서 중요하게 다뤄진다. 이는 학습률, 학습 단계 수, 배치 사이즈 등을 포함한다. 하지만, 본 연구에서는 응용 간의 간섭에 초점을 두고 있기 때문에 고정된 파라미터를 사용하였다. 파라미터를 조절하여 실험을 진행 했을 때, GPU 활용도 수치는 변화하였지만 정적이고 높은 활용도를 가지는 특성은 유지하는 것을 발견하였다. 한편, 그림 4-(c)는 딥 러닝 추론 (DL inference) 작업의 GPU 활용도를 그래프로 나타낸 것이다. 딥 러닝 훈련 작업 보다는 값의 변화가 크게 일어나지만 최대 변화율이 약 10% 로, 최대 변화율이 약 90%인 HPC 범주의 응용보다는 비교적 정적인 특성을 보인다. 특히, 최대 GPU 활용도가 20% 미만으로써 낮은 활용도를 보여준다. 이를 통해 딥 러닝 추론 작업의 계산 강도가 높지 않음을 알 수 있으며 GPU의 계산 자원을 많이 활용하지 않아 이로 인해 유휴한 GPU 컴퓨팅 자원이 많이 발생할 수 있음을 알 수 있다. 딥 러닝 추론 작업도 딥 러닝 훈련 작업과 마찬가지로 파라미터를 조절하여도 정적이며 낮은 활용도를 갖는 특성은

유지되는 것을 실험을 통해 확인하였다. 이와 같은 실험 결과를 고려하였을 때, 평균 혹은 최대 GPU 활용도를 가지고 GPU 응용의 특성을 나타내기에는 한계가 있음을 알 수 있다.

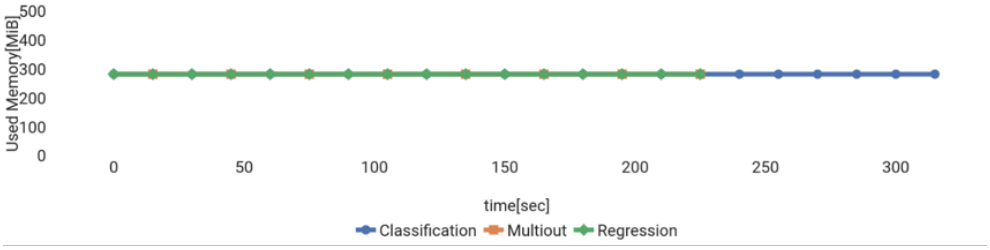
2) GPU 메모리 사용량



(a) GPU memory usage pattern of HPC applications



(b) GPU memory usage pattern of DL training applications

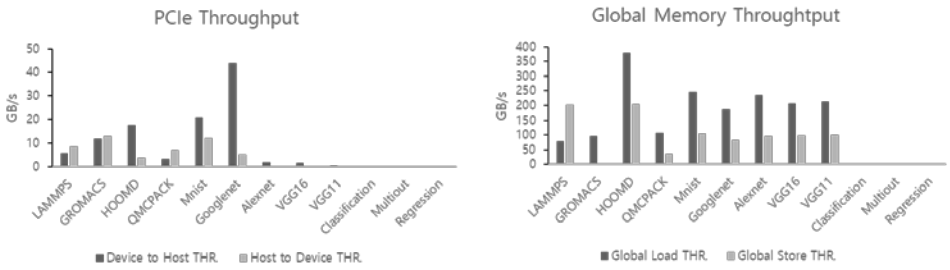


(c) GPU memory usage pattern of DL inference applications

<그림 5> 응용의 메모리 사용 패턴

그림 5는 각 응용의 메모리 사용 패턴을 그래프로 나타낸 결과이다. 각 응용 별 범주에 따라 GPU 활용도와 비슷한 양상을 보인다. HPC 응용은 메모리 사용량의 변화율이 매우 커 메모리 사용량 예측이 어렵다. QMCPACK의 경우 메모리 사용량이 계단식으로 변화하는 패턴을 보이며, LAMMPS의 경우 메모리 사용량이 점진적으로 증가하는 패턴을 보이다 응용의 수행 마지막 부분에서 메모리 사용량의 최대치가 등장한다. 딥 러닝 훈련 작업의 경우에는 한 번 메모리를 잡으면 더 이상 요청 및 해제하지 않고 사용하여 정적인 메모리 사용 패턴을 보인다. 이 때 각 딥 러닝 모델 별로 사용하는 메모리 양이 달랐으며, 실험결과 하이퍼 파라미터를 조절하면 사용하는 메모리 양은 달라지지만 정적인 패턴은 그대로 유지되는 것으로 확인하였다. 딥 러닝 추론 작업의 경우에도 정적인 메모리 사용 패턴을 보이나 사용하는 메모리 자원의 양이 다른 응용과 비교하여 매우 적은 것을 알 수 있다. GPU 활용도가 높아지면 대부분 사용하는 GPU 메모리 양도 증가하지만 GROMACS와 같은 경우 GPU 활용도는 높지만 사용하는 GPU 메모리 양은 약 0.5GB로 적었다. 딥 러닝의 경우 여러 하이퍼 파라미터를 조절하였을 때 메모리 사용량을 예측하거나, 1초 내외의 짧은 프로파일링을 통해서 메모리 사용량을 예측하는 이 전 연구들이 있었다. 하지만 본 실험의 결과와 같이 동적인 사용량을 보이는 HPC 응용들은 계단식 혹은 하이 피크(high peak) 등의 패턴을 보이기 때문에 이전 연구의 방법을 적용한 메모리 사용량 예측은 어렵다. 그러므로 프로파일링 및 모니터링을 통한 예측과 동적 스케줄링이 필요하다.

3) 전역 메모리 및 PCI-E 처리량



<그림 6>응용의 PCI-E 및 전역 메모리 처리량

GPU에서 커널로 인해 발생하는 처리량은 전역 메모리 처리량, PCI-E 처리량이 있다. 전역 메모리 처리량은 커널의 로드 또는 저장 요청과 캐시의 적중률 (Hit ratio) 및 적중 실패율(miss ratio)과 관련이 있다. 만약 SM 내부의 캐시인 L1 캐시에서 데이터 적중이 일어나게 된다면, 이 처리량은 높아지게 될 것이다. PCI-E 처리량은 호스트 메모리인 DRAM에 접근하기 위해 PCI express 버스에서의 데이터 이동 양에 관련된 통계 이다. 그러므로 전역 메모리의 처리량은 높을수록, PCI-E의 처리량은 낮을수록 성능이 좋고, 대기 시간이 적다. 그림 x를 통해 각 응용의 처리량을 보여준다. 딥 러닝 추론 작업은 전역 메모리 처리량 도 낮지만 PCI-E 처리량도 매우 낮은 것을 확인할 수 있다. HPC 응용의 경우 각 응용의 종류에 따라 다양한 PCI-E 및 전역 메모리 처리량을 보인다. 한편 딥 러닝 훈련 작업의 경우 전역 메모리 처리량에 있어서 모두 비슷한 처리량을 보여주는 반면, PCI-E 처리량에 있어서는 각 모델마다 다른 양상을 보이는 것을 확인할 수

있다. CNN과 Googlenet은 PCI-E 자원을 많이 사용하고, 디바이스와 호스트 간 데이터 이동이 많이 일어나 처리량이 크지만 Alexnet, VGG11, VGG16 모델은 PCI-E 처리량이 매우 낮다. 이를 통해 딥러닝 훈련 작업의 경우 각 모델마다 CPU 와의 통신이 많을 수도, 적을 수도 있다는 것을 확인할 수 있다. 또한, PCI-E 처리량이 높다는 것은 호스트 메모리에 버스를 통해 접근해야 하므로 대기 시간이 발생할 수 있는 것을 의미하지만, 동시에 GPU 커널과 호스트-디바이스 간 메모리 작업이 중첩 되어 수행 가능하므로 동시 수행의 장점을 취할 수 있다고도 얘기할 수 있다.

3. 간섭 예측을 위한 기계 학습 모델링

직접 모든 응용의 쌍들을 수행시켜 본 후 간섭 값을 유도해낸다면 가장 정확하고 최적의 결과를 낼 수 있다. 하지만 이는 응용의 개수가 많아질수록, 응용의 수행 시간이 길수록 소요되는 시간이 기하급수적으로 증가한다. 예를 들어 N개의 응용이 있을 때, $\frac{N * (N-1)}{2}$ 개의 쌍을 수행해봐야 한다. 3.2 절에서 살펴본 자원과 함께 GPU에서 응용의 동시 실행에 영향을 미치는 하드웨어 특성 들이 간섭에 미치는 영향을 고려하여 모든 응용 쌍들을 수행해보지 않아도 오프라인 프로파일링 정보만을 통해 간섭을 예측하도록 하였다.

1) 프로파일링을 위한 자원 사용 메트릭 정의

우리는 실제 공동 실행 시 성능에 영향을 미치는 자원들을 정의한다. 수집한 메트릭들은 공동 실행 시 간섭 방지를 예측하기 위해 사용한다. 각 메트릭은 NVIDIA 프로파일러 도구인 Nvprof[28], Nsight[29]를 사용하여 수집하였다. 표 1은 프로파일링 중에 획득한 각 관련 자원에 대한 메트릭의 세부정보를 나타낸다.

GPU 평균 활용도는 응용의 하나 이상의 커널이 GPU에서 실행 된 시간의 평균을 의미한다. 평균 SM 효율성은 GPU의 SM에서 하나 이상의 워프가 활성화 된 시간의 평균을 백분율로 나타낸다. 워프 평균 효율성은 SM에서 워프 당 활성화된 스레드 수의 평균을 의미한다. IPC(Instruction Per Cycle)는 사이클 당 실행된 명령어의 수이다. 평균 점유율은 SM에 지원되는 워프의 최대 개수에 대하여 사이클 당 활성 워프의 평균을 의미한다. 최대 GPU 메모리 사용량은 응용 프로그램 실행 시 사용하는 최대 GPU 메모리 양이다. 평균 GPU 메모리 사용량은 응용 프로그램 실행 동안 사용한 GPU 메모리의 평균 값을 나타낸다. 평균 GPU 메모리 활용도는 응용의 실행 중 일정 기간을 기준으로 하여 GPU 메모리를 읽거나 쓰는 시간의 백분율 값을 평균 낸 것이다. 장치-호스트 처리량은 PCI-E를 통하여 GPU 메모리에서 CPU 메모리로 이동하는 데이터의 처리량을 나타낸다. 호스트-장치 처리량은 CPU 메모리에서 GPU 메모리로 이동하는 데이터의 처리량을 나타낸다. 전역 메모리 로드 처리량은 L1 캐시 및 L2 캐시의 트랜잭션을 포함하는 처리량이다. 이 메트릭은 캐시 히트를 반영하며, 전역 메모리 저장 처리량 또한 L1, L2 캐시와 관련이 있다. 또한, 각 응용 들의 수행 시간과 응용 수행 시 입력 값을 기록하였다. 이는 같은 응용일지라도 사용한 입력 값 및 실행 파라미터에 따라 프로파일링 정보가 달라질 수 있기 때문이다. 같은 구성을 가진 응용이 제출된다면 이전에 수집된

프로파일링 정보를 사용할 수 있다.

<표 1> 간섭 모델링을 위한 자원 메트릭

메트릭	
평균 GPU 활용도	평균 GPU 메모리 활용도
평균 SM 효율성	디바이스-호스트 처리량
평균 워프 효율성	호스트-디바이스 처리량
IPC	전역 메모리 로드 처리량
평균 점유율	전역 메모리 저장 처리량
최대 메모리 사용량	수행 시간
평균 메모리 사용량	GPU 코어 사용량
	수행 시간

2) 간섭 모델링

Interference modeling을 위해서 우리는 선형 회귀 (Linear regression), 의사 결정 트리 (Decision tree), 랜덤 포레스트 (Random forest)에 해당하는 세 가지 머신 러닝 모델을 구축하였다. 우리는 모델 구축을 위해서 3절에서 사용한 총 12개의 응용을 사용하였는데, 이들의 쌍을 모델링 하였으므로 총 144개의 데이터를 사용하였다. 우리는 모델의 입력으로 각 응용의 메트릭 조합을 사용하였다. 이 때, 각 메트릭이 사용하는 단위 및 스케일링(Scaling)이 다르므로 메트릭 값을 정규화 하였다. 모델의 출력 값은 간섭 값인데, 이는 동시 실행되었을 때의 시간과 응용이 단독으로 수행되었을 때의 시간을 비교하여 비율로 나타낸 값이다. 우리는 모델 생성을 위해 모든

응용의 쌍들을 수행시켜본 뒤 수행 시간을 구하여 간섭 값을 계산하였으며, 보다 정확한 수행 시간 측정을 위해 세 번의 실험 결과의 평균 값을 사용하였다. 또한, 모델의 정확도 향상 및 성능 평가의 신뢰성 향상을 위해서 5겹 교차 검증(5-fold cross validation) 사용하였다. 이는 전체 데이터 셋을 5개 분할한 뒤 이 서브 셋 중 4개는 학습 데이터로, 1개는 검증 데이터로 지정하여 이를 5번 반복한다.

표 2 는 세 개의 머신 러닝 모델링을 사용하였을 때의 평균 제곱 오차 (Mean Square Error: MSE) 값과 R 제곱(R-squared: R2)을 보여준다. MSE는 모델이 예측한 값과 실제 값의 차이를 보여주며, 0에 가까울수록 정확도가 높음을 나타낸다. R2는 회귀 모델의 검증 척도로써, 1에 가까울수록 모델의 설명력이 높다. 랜덤 포레스트 모델이 가장 좋은 성능을 나타낸 것을 확인할 수 있었다. 그래서 랜덤 포레스트 모델을 추후 실험에서 사용하였다.

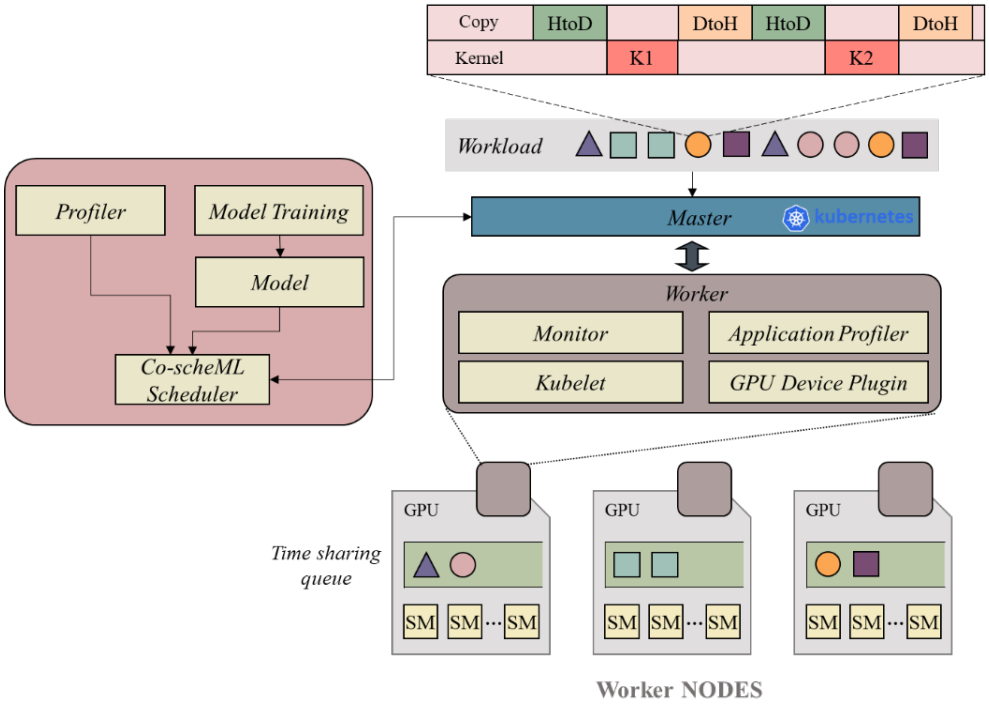
<표 2> 회귀 모델들의 평균 제곱 오차 및 R 제곱

	선형 회귀	의사 결정 트리	랜덤 포레스트
MSE	0.0546	0.0269	0.0222
R-Squared	0.6946	0.8500	0.8758

4. 설계 및 구현

본 절에서는 본 논문에서 제안하는 Co-scheML의 전체 구조 설계와 동적으로 도착하는 응용에 대해 GPU 클러스터에서 노드 별로 스케줄링하는 방법을 설명한다.

1) 시스템 구조



<그림 7> Co-scheML의 전체 구조

Co-scheML의 전체적인 시스템 설계는 그림 7과 같다. 프로파일 저장소 (Profile repository)에는 응용이 단독으로 수행되었을 때의 프로파일링 정보가 응용의 이름 및 입력 데이터로 라벨링되어서 저장된다. 메트릭들은 시계열 기반 데이터베이스인 influxDB에 저장되며 프로파일링 단계는 오프라인으로 진행된다. Co-scheML 는 프로파일 저장소와 모델 (Model)로부터 간섭 값을 요청하고 이에 따라 스케줄링한다. 응용이 수행되는 동안 모니터 (Monitor)에서 모니터링이 계속 진행되는데, 이는 응용의 진행과정을 모니터링하고, 프로파일링 정보도 업데이트하여 정확도를 높인다.

2) 스케줄러

Co-scheML의 스케줄러는 큐 (Queue)에 대기 중인 작업이 없을 때는 유휴한 GPU로 인해 낭비되는 자원이 없고, 응용이 GPU를 독점적으로 사용해서 최대의 성능을 낼 수 있도록 작업을 가능한 많은 노드에 분산시킨다. 큐에서 대기하는 응용이 생기면 응용이 GPU를 독점적으로 사용함으로써 낭비되는 자원을 활용하기 위해 응용을 쌍으로써 실행한다. 이 때, 각 응용의 쌍 마다 간섭 값이 다르므로 이를 고려하여 스케줄링한다. 우리는 탐욕 알고리즘 (Greedy algorithm)에 따라 최소의 간섭 값을 갖는 쌍을 선택한다.

해당 스케줄러는 쿠버네티스 디폴트 스케줄러 및 확장 스케줄러[30]를 확장 수정하여 구현하였다. 쿠버네티스 디폴트 스케줄러는 사용자가 응용을 제출하여 큐에 응용이 도착하거나 실행 중인 응용이 종료하는 이벤트가 발생하였을 때, Co-scheML 스케줄러에게 대기 열에 존재하는

응용이 스케줄링 될 노드에 대한 필터링을 요청한다. 그러므로 본 스케줄링 알고리즘은 새로운 응용이 제출되었을 때 또는 응용의 실행이 끝났을 때 호출된다. Co-scheML의 스케줄링 알고리즘은 그림 8과 같다. 쿠버네티스 디폴트 스케줄러로부터 스케줄링할 응용, 클러스터에 존재하는 GPU 노드의 리스트, 현재 큐에 대기 중인 응용의 리스트 전달받으면 Co-scheML은 노드 할당의 결과를 반환한다. Find_idle_nodes() 함수는 gpu_nodes에 있는 노드 중 실행 중인 응용이 없는 노드를 리턴한다. 만약 유휴한 노드가 있다면 이 노드 중 하나를 응용에게 할당해준다(lines 4-5). 만약 이러한 node가 없다면 find_exclusively_executing_apps() 함수를 호출하여 gpu_nodes에서 독점적으로 GPU를 사용하면서 혼자 수행중인 응용을 찾는다. 이러한 응용이 있다면 혼자 수행중인 모든 single_apps와 큐에서 대기하고 있는 pending_apps를 대상으로 *calculate_interference*() 함수를 통해 모든 응용 쌍의 간섭 값을 계산한다. 간섭 값을 구하기 위해서 각 응용의 메트릭을 프로파일 저장소에 요청한다(lines 18-19). 쿼리를 통해 얻은 메트릭 결과를 입력 값으로 하여 3.3절의 간섭 모델로부터 간섭 값을 리턴 받는다(line 22). 리턴 받은 값에 대해서 오름차순으로 정렬하고, *find_selected_pairs()* 함수를 호출하여 탐욕 알고리즘에 따라 함께 수행할 쌍을 선택한다(lines 10-11). *Find_selected_pairs()*는 정렬된 쌍을 대상으로 해당 쌍의 응용들이 아직 선택되지 않았을 경우 동시 수행이 가능한지 확인한다. 이는 프로파일링 정보 및 모니터링 정보를 가지고 OOM을 예측한다(lines 26-28). 대상 응용이 최소 간섭 값부터 선택하며 동시 수행될 수 있는 쌍들의 리스트에 포함되는지 확인한다(lines 12). 포함된다면 find_node 함수를 통해 해당 쌍의 s_app이 수행중인 노드를 찾아 이 노드를 selected_node로써 반환해준다(lines 13-14).

Algorithm 1 Co-scheML Scheduler

```
1: procedure Co-scheML(App, gpu_nodes, pending_apps)
2: selected_node ← ∅
3: idle_nodes ← find_idle_nodes(gpu_nodes)
4: if idle_nodes ≠ ∅ then
5:     selected_node ← idle_nodes[0]
6: else
7:     single_apps ← find_exclusively_executing_apps(gpu_nodes)
8:     if single_apps ≠ ∅ then
9:         app_pairs ← calculate_interference(single_apps, pending_apps)
10:        sorted_pairs ← sort_by_interference_val(app_pairs)
11:        selected_pairs ← find_selected_pairs(sorted_pairs)
12:        if App ∈ selected_pairs then
13:            selected_node ← find_node(App, selected_pairs)
14:        endif
15:    endif
16: endif
17: return selected_node
18: procedure calculate_interference( single_apps, pending_apps)
19: for s_app in single_apps do
20:    for p_app in pending_apps do
21:        s_metrics ← query_profile_repository(s_app)
22:        p_metrics ← query_profile_repository(p_app)
23:        interference_val ← query_ML_model(s_metrics, p_metrics)
24:        app_pairs← append(s_app, p_app, interference_val)
25:    endfor
26: endfor
27: return app_pairs
28: procedure find_selected_pairs (app_pairs)
29: for pair in app_pairs do
30:    if selected(pair.s_app) ≠ true and selected(pair.p_app) ≠ true then
31:        if can_co-schedule(pair.s_app, pair.p_app)  then
32:            selected_pairs ← append(pair)
33:        endif
34:    endif
35: endfor
36: return selected_pairs
```

<그림 8> Co-scheML 스케줄러의 알고리즘

5. 실험 및 결과

1) 실험 방법

가) 실험 환경

<표 3> 쿠버네티스 클러스터 환경 정보

	Node Details	
	CPU(master)	GPU(node)
Architecture	Intel® Core™ i7-5820K	Nvidia GeForce Titan Xp D5x
Core Clock	3.30GHz	1.58GHz
Num of Cores	6	3840
Mem. Size	32GB	12GB
Threading API	-	Nvidia CUDA 10.0
PCIe bandwidth	-	32GB/s
OS	Ubuntu 16.04.6 LTS	Ubuntu 16.04.6 LTS

<표 4> GPU 노드의 환경 정보

Static Environment features			
Memory Size	11.91 GB	Warps per SM	64
GPU speed	1582 MHz	Thread blocks per SM	32
GPU architecture	Pascal architecture	Shared Memory per SM	96KB
PCIe bandwidth	32GB/s	Threads per SM	2048

쿠버네티스 기반 사설 GPU 클러스터를 사용하여 평가를 진행한다. 실험 환경은 표 3 및 표 4와 같다. 클러스터는 표 3과 같이 마스터 노드 (Master node) 1개와 3 개의 워커 노드 (Worker node) 로 구성되어 있다. 워커 노드인 GPU node에는 NVML[31] 기반 자원 모니터링 구성 요소인 모니터가 존재하고, 이는 5초마다 influx-DB[32]에 메트릭을 기록한다. 마스터 노드에는 쿠버네티스 디폴트 스케줄러, Co-scheML 스케줄러 및 모델이 존재한다. Co-scheML 스케줄러는 쿠버네티스의 스케줄러 확장 메커니즘을 사용하였다.

나) 실험 워크로드

<표 5> 워크로드 순서에 따른 워크로드의 특성

Workload sequence	Characteristics of workload
0	높은 GPU 활용도 (HOOMD, Mnist, Googlenet, VGG11, VGG16, GROMACS)
1	높은 메모리 활용도 (Mnist, Googlenet, VGG16, VGG11)
2	높은 PCI-E 처리량 (LAMMPS, GROMACS, HOOMD, QMCPACK, Mnist, Googlenet)
3	딥 러닝 훈련 응용
4	딥 러닝 추론 응용
5	HPC 응용
6	딥 러닝 응용
7,8,9	무작위로 선정한 응용

12개의 실제 응용을 선정하였다. 이는 Nvidia GPU Cloud(NGC)[25]에서 4개의 HPC 응용(LAMMPS, GROMACS, QMCPACK, HOOMD)을 사용하였으며 표준 입력 값으로 수행된다. 딥 러닝 훈련 작업으로 mnist, googlenet, alexnet, vgg16, vgg11[26] 총 5개의 CNN 모델을 사용하였다. 딥 러닝 추론 작업으로는 DJINN workload suite[27]의 샘플 응용들을 사용하였다. 딥 러닝 작업들은 모두 Tensorflow를 사용하여 GPU에서 실행되며 Docker 컨테이너 컨테이너화 하였다.

10개의 워크로드를 선정하였다. 특징이 정해진 7개의 워크로드와

임의의 3개 워크로드로 나누었다. 사용된 워크로드의 특성은 표 5와 같다. 또한, 스케줄러의 민감도 (Sensitivity)는 응용의 도착 간격을 다르게 하여 평가한다[33-34]. 이 때, 부하를 조절하기 도착 interval을 위해 15초, 30초, 60초로 지정하며 각각을 light loaded, medium loaded, heavy loaded로 명명한다. 기본 태스크 밀도(task density)는 medium loaded로 한다.

다) 평가 지표

- Average Job Completion Time(JCT) 는 각 작업이 제출된 시간부터 완료되는 시간을 평균 낸 값이다.
- Makespan은 워크로드의 모든 응용이 끝나는 시간이다.
- Speedup은 응용이 co-schedule되었을 때 각 응용의 수행 시간을 단독으로 수행되었을 때의 시간으로 정규화 한 값이며 이는 0부터 1사이의 값을 갖고, 1에 가까울수록 단독으로 수행하였을 때와 비교하여 성능의 차이가 없음을 의미한다.

라) 비교 대상

비교 대상 스케줄러는 다음과 같다. 먼저 최대 메모리 사용량 기반의 빈 팩 스케줄러 (Binpack scheduler)와 간섭을 고려한 스케줄러인 로드 밸런스 (Loadbalance), Mystic [33] 스케줄러이다.

로드 밸런스 스케줄러는 평균 GPU 활용도를 기반으로 하여 가장 큰 GPU 활용도를 가지는 응용과 가장 작은 GPU 활용도를 나타내는 응용을 동시 수행할 쌍으로 선택한다. Mystic 스케줄러는 [33] 논문에서 제안한 응용의 메트릭끼리의 유사도를 계산하여 유사도가 낮은 순으로 스케줄링한다.

2) 스케줄링 성능

본 절에서는 다양한 워크로드에 따라 각 스케줄링 방법의 성능에 미치는 영향을 분석한다. 이 때의 작업 부하는 medium으로 지정하였다. 그림 9-(a)는 총 10개의 워크로드에 대한 평균 JCT를 보여준다. 3개의 워크로드를 제외한 모든 워크로드에서 Co-scheML이 가장 짧은 JCT를 보여주며 전체 워크로드에 대한 JCT 평균은 Co-scheML이 844.09초, 빈 팩 스케줄러가 1089.32초, 로드 밸런스가 943.05초, mystic이 967.94초로 성능이 각각 약 30%, 11%, 15% 성능 향상을 보였다. 전체 워크로드에 대한 Makespan은 그림 9-(b)를 통해 확인할 수 있다.

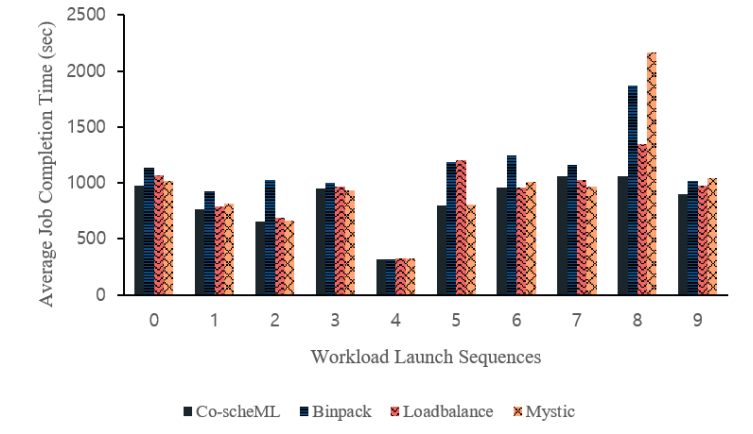
Makespan 측면에서 총 10개의 워크로드에 대하여 3개의 워크로드를 제외한 모든 워크로드에서 Co-scheML이 가장 좋은 성능을 보였다. 전체 워크로드에 대한 Makespan의 평균은 Co-scheML, 빈 팩 스케줄러, 로드 밸런스, Mystic이 각각 1705.1초, 2156.1초, 2027.3초, 2031.8초로 Co-scheML의 Makespan이 26%, 18%, 19% 감소하였다. 1, 3, 5, 7번 워크로드에 대해서는 Makespan과 average JCT 사이에 trade-off가 존재하였다. 예를 들면, 7번 워크로드의 경우 Co-scheML의 평균 JCT는 1060초로 성능이 가장 좋았던

Mystic(970초)과 비교하여 성능이 약 9% 감소하였다. 하지만 Makespan은 각각 2411초, 2931초로 Co-scheML의 성능이 약 18% 증가하였다. 4번 워크로드는 average JCT, Makespan 각각 2%, 7%의 성능 하락이 있는 워크로드로 딥 러닝 추론 작업으로만 구성된 워크로드다. 딥 러닝 추론 작업은 자원을 적게 사용하는 응용으로, 각 응용 간에 간섭이 적게 발생하므로 간섭을 고려하지 않아도 동시 수행의 이득을 보는 것으로 확인하였다. 그러나, Co-scheML이 좋은 성능을 보인 워크로드는 2번, 8번, 9번 워크로드로 average JCT는 각각 평균 약 22%, 69%, 12% 가, Makespan은 평균 약 24%, 44%, 48% 향상되었다. 해당 워크로드의 특징은 자원을 많이 사용하는 응용으로 구성된 워크로드로써, 자원을 많이 사용하는 응용들은 서로 간의 간섭이 많이 발생하므로 Co-scheML이 이들을 스케줄링 할 때 가장 큰 장점을 보인다.

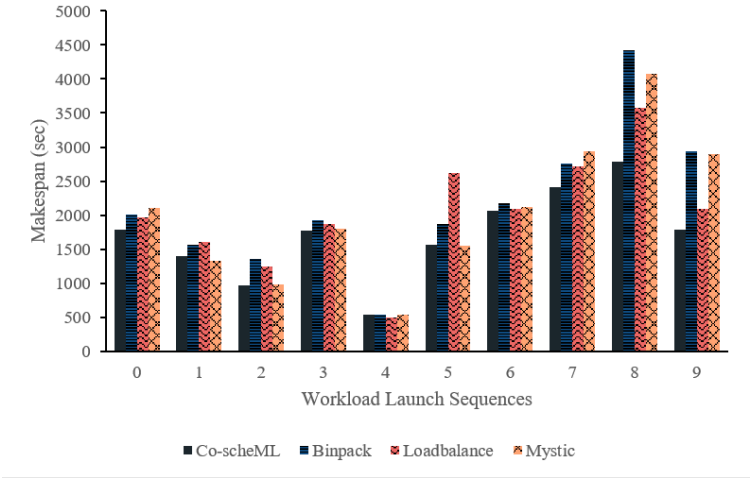
그림 10은 각 워크로드 별로 Speedup을 비교한 결과이다. 각 스케줄러 별로 78%, 56%, 67%, 75%의 Speedup을 보여 준다. 로드 밸런스 스케줄러는 평균 GPU 활용도를 고려하여 공정하게 계산 자원을 나눠 쓰므로 Speedup은 좋았으나 전체 자원 사용은 고려하지 않아 Makespan 및 average JCT에서 충분한 성능을 보이지 않았다. Mystic 스케줄러의 경우 비교 대상 스케줄러 중에서 가장 좋은 성능을 보였지만 유사도를 기반으로 간섭 예측을 하여 Speedup도 충분히 높일 수 없었으며, OOM을 고려하지 않았으므로 평균 JCT와 Makespan에 대해서 Co-scheML보다 나쁜 성능을 보인다.

그림 11은 각 노드당 스케줄러별로 GPU 활용도를 나타낸 그래프이다. 실험에서 사용한 전체 12개의 응용을 각각 두 번씩 실행하여 워크로드는 총 24개의 작업으로 구성되어있으며, 이들의 실행

순서는 임의로 생성하였다. 워크로드전체 워크로드 수행동안 Co-scheML은 평균 GPU 활용도는 78%였으며, 로드 밸런스는 59%, Mystic은 67%으로 Co-scheML이 각각 32%, 16% GPU 활용도를 향상시켰다. Co-scheML은 간섭을 고려하여 각 응용 간에 서로 상호 보완적인 자원을 사용하며 결과적으로 GPU 활용도를 높일 수 있었다.

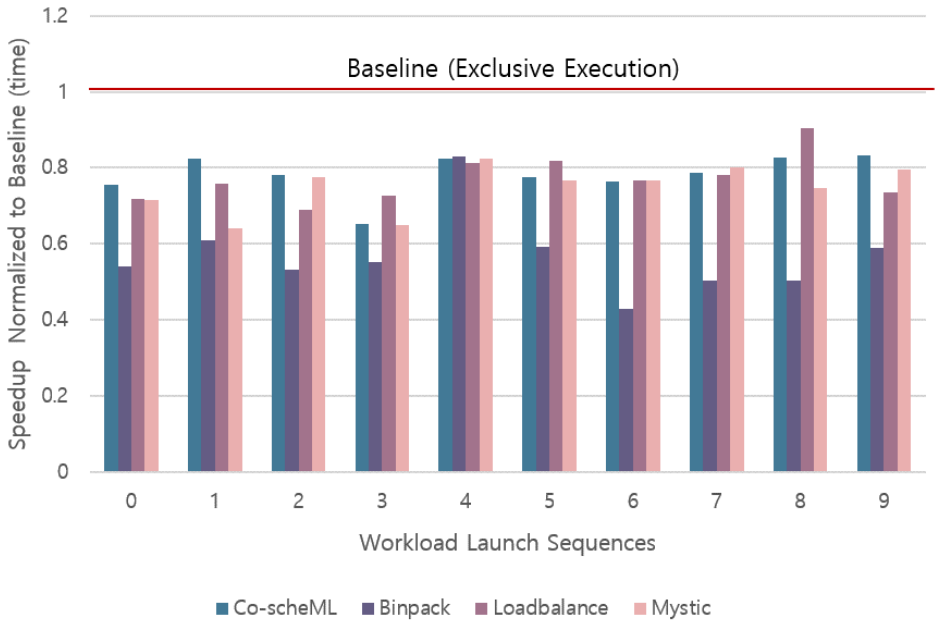


(a) Average JCT

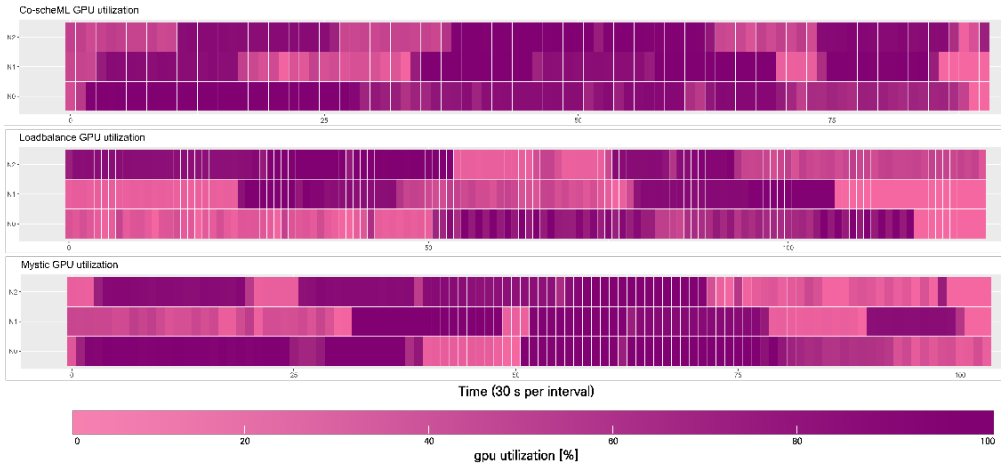


(b) Makespan

<그림 9> 워크로드 및 스케줄러에 따른 평균 JCT와 Makespan

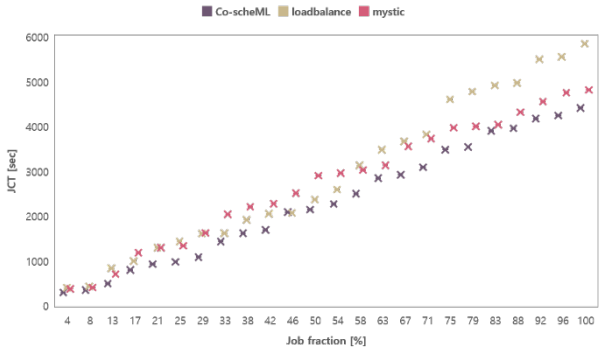


<그림 10> 워크로드 및 스케줄러에 따른 Speedup

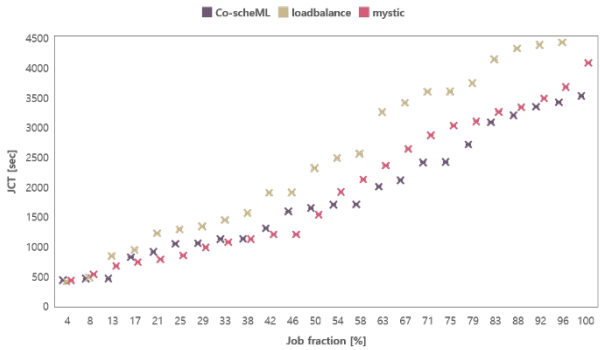


<그림 11> 스케줄러 별 GPU 활용도

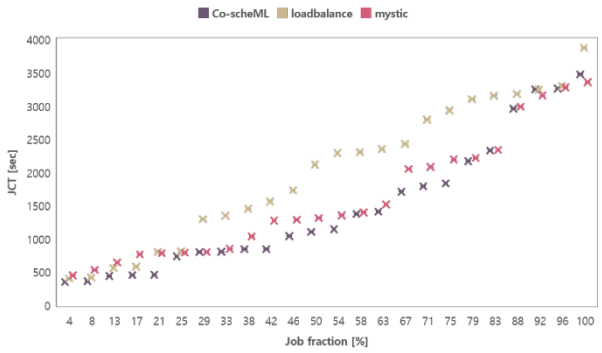
3) 민감도 분석



(a) Light loaded



(b) Medium loaded



(c) Heavy loaded

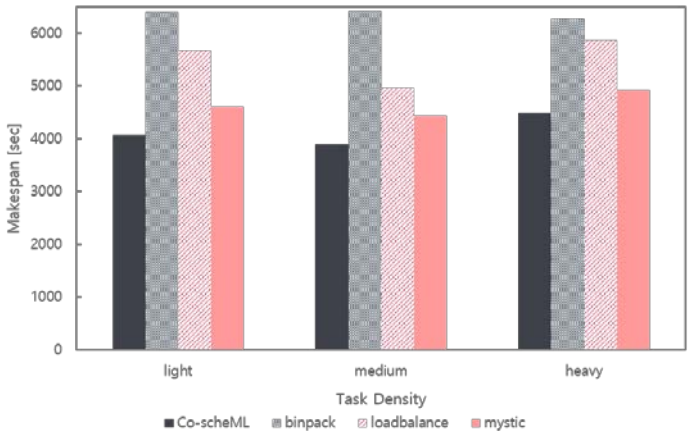
<그림 12> 다양한 서버 부하에 따른 평균 JCT 비교

본 절에서는 동일한 워크로드에 대하여 서버에 부하를 다르게 주었을 때 각 스케줄링 방법을 평가한다. 그림 X는 각 작업 부하에 따른 평균 JCT를 나타낸다. 워크로드의 전체 응용에 대해서 JCT로 sorting하였다. 그림 12-(a)는 heavy loaded일 때, 12-(b)는 medium loaded일 때, 12-(c)는 light loaded일 때를 보여준다. heavy loaded일 때는 중간 값 기준으로 Co-scheML의 평균 JCT가 로드 밸런스 스케줄러, Mystic에 비교하여 각각 12%, 32% 낮았으며 medium loaded일 때는 43%, 3% 낮았고, light loaded일 때는 96%, 18% 낮았다.

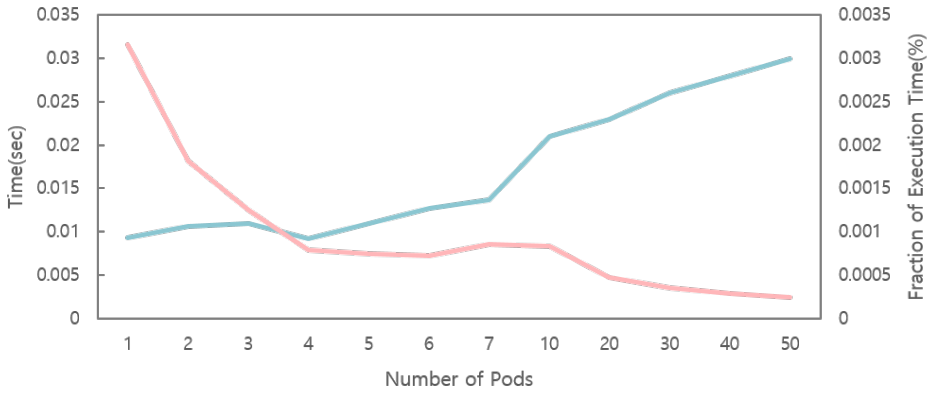
그림 13은 각 서버 부하에 따른 Makespan을 보여준다. 전체 작업 부하에 대해서 Co-scheML은 빈 팩 스케줄러, 로드 밸런스 스케줄러, Mystic과 비교하여 Makespan이 1.53배, 1.32배, 1.12배 성능이 좋았다. 시스템의 부하가 늘어날수록 모든 스케줄러의 성능이 나빠지는 경향을 보였다. 한편 빈 팩 스케줄러 스케줄러가 가장 작업 부하에 영향을 크게 받지 않고 비슷한 성능을 보였지만 성능이 가장 좋지 않았으며 Co-scheML은 모든 작업 부하에 대해서 가장 좋은 평균 JCT 및 Makespan를 보여줌을 확인할 수 있다.

4) 스케줄링 오버헤드

빈 팩 스케줄러 스케줄러는 팟 (Pod)의 개수와 상관 없이 스케줄러 오버헤드가 약 1.001초로 동일하다. 빈 팩 스케줄러 스케줄러는 모든 노드의 가용한 메모리 양과 현재 실행되어야 하는 팟이 요구하는 메모리 양을 비교하여 판단을 내리므로 노드의 개수에 따라 오버헤드가 비례적으로 증가한다. 본 실험에서는 3개의 노드를 사용하였으므로 약 1초의 오버헤드가 존재하였다.



<그림 13> 다양한 서버 부하에 따른 Makespan 비교



<그림 14> Co-scheML의 스케줄링 오버헤드

본 논문에서 구축한 스케줄러는 그림 14의 파란색 선과 같이 팟의 개수에 따라 간섭 값을 비교할 대상이 많아지므로 오버헤드가 선형적으로 증가한다. 하지만 50개의 팟이 존재하더라도 0.003초 이하의 스케줄링 시간이 소요된다. 그림 14의 주황색으로 나타나는

선은 전체 수행 시간 대비 오버헤드를 보여주는데, 이는 0.0003% ~ 0.0035%의 백분율을 가짐으로써 실행 시간 오버헤드는 사소하며 빈 팩 스케줄러 스케줄러보다 낮은 오버헤드가 존재함을 알 수 있다.

또한, HTTP 요청으로 인한 통신 오버헤드가 존재하는데 이는 평균 약 6초의 시간이 소요된다. 이는 쿠버네티스 스케줄러에 공통적으로 포함되는 오버헤드로써 쿠버네티스 디폴트 스케줄러, 빈 팩 스케줄러 스케줄러, 본 논문에서 제안한 스케줄러에 모두 존재한다.

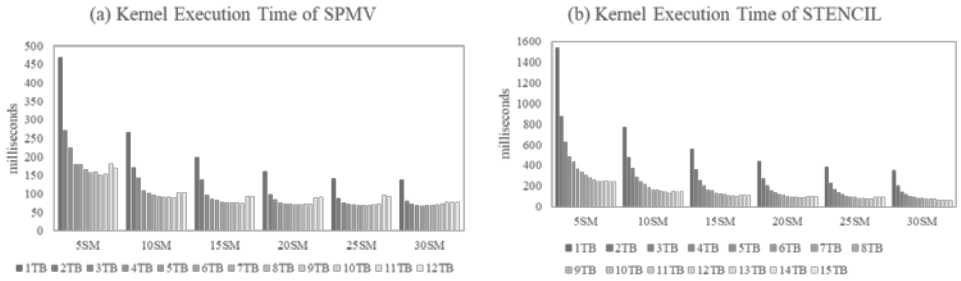
IV. Intra-SM 자원 공유 다중 작업 배치 기법

1. 연구 동기

1) Intra-SM 자원 할당량에 따른 응용 성능 분석

그림 x는 응용 SPMV[37]와 STENCIL[37]을 대상으로 각 응용에 할당해주는 자원의 양에 따른 성능을 측정한 결과이다. 해당 실험은 총 30개의 SM을 가진 NVIDIA Titan Xp GPU 상에서 진행하였다. 이는 각 SM 별 활성 스레드 블록의 개수와 활성 SM 수를 조절하여 각 응용에 할당된 자원의 양을 조절하였을 때 커널 수행 시간이 달라짐을 보여준다.

그림 15-(a)는 자원 할당 양에 따른 SPMV의 커널 수행 시간을 보여주는 그래프이다. SPMV 30SM에 1개의 스레드 블록을 활성화 시켰을 때에는 커널 수행시간이 137ms 이었으며, 30SM에 4개의 스레드 블록을 활성화 함으로써 할당된 자원의 양을 증가시켰을 때 약 68ms 으로 시간이 감소하였다. 하지만, 12개의 스레드 블록을 활성화 시키면 커널 수행 시간이 77ms로 오히려 시간이 증가한 것을 확인할 수 있다. 해당 응용은 정적 프로파일링 정보인 레지스터 개수를 기준으로 하여 한 SM에 12개의 스레드블록까지 활성화 할 수 있으나, 12개의 스레드 블록을 모두 할당해주지 않고 4개의 스레드 블록만 활성화해주어도 원하는 성능을 얻을 수 있다.



<그림 15> SPMV / STENCIL의 활성 SM 개수 SM 당 제출된 스레드 블록 개수에 따라 달라지는 커널 실행 시간

반면 그림 15-(b)의 STENCIL은 30개의 활성 SM을 할당해 주었을 때 활성 스레드 블록의 개수를 1개로 지정하였을 때는 커널 수행 시간이 348ms, 4개로 지정하였을 때는 115ms, 12개로 지정하였을 때는 59ms가 소요되었다. SM 내 가용한 자원 양을 고려하여 최대 개수인 15개의 스레드 블록을 스케줄링 하였을 때는 59ms가 소요되었다. STENCIL 또한 SPMV처럼 하드웨어의 제약 사항 이전에 성능이 포화 되지만 그 시점이 SPMV보다는 더 많은 자원을 할당해줘야 함을 알 수 있다. 이를 통해 각 응용 마다 활성 SM 및 스레드 블록 개수에 따라 성능이 포화 되는 지점이 다른 것을 보여준다.

2) 커널의 이질성

표 6은 NVIDIA TITAN XP GPU 및 i7-5820K CPU 환경에서 각

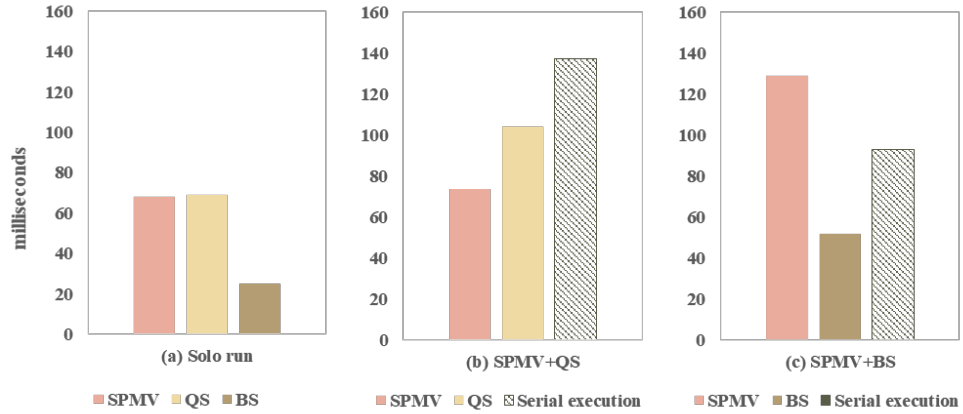
응용의 Intra-SM 자원 사용량 및 런타임에 일어나는 실행 지연의 이유를 나타내고 있다. 예를 들어 QS 응용의 경우 15360개의 레지스터를 사용함으로써 하나의 스레드 블록 만을 활성화 하여도 SM 전체의 약 23%의 레지스터를 사용함으로써 가장 많은 레지스터 사용량을 보이지만 공유 메모리는 전혀 사용하지 않는 것을 확인할 수 있다. 반면에 CUTCP는 4019 바이트의 공유 메모리 사용량을 가짐으로써 1개의 스레드 블록이 SM 내의 전체 공유 메모리의 약 10%를 사용하지만 레지스터는 전체 레지스터의 5%도 사용하지 않는다. 따라서 QS는 레지스터에 대한 하드웨어 제약으로 인하여 최대 활성 스레드 블록 수가 제한되며, CUTCP는 공유 메모리에 대한 하드웨어 제약으로 인하여 제한된다. 이를 통해 각 커널이 고갈시키는 Intra-SM 자원은 이질적이며, 단일 커널 실행은 SM 내부의 자원 활용도를 낮출 수 있다는 것을 알 수 있다.

커널의 런타임 동작 또한 이질성을 가진다. BlackScholes 응용 및 SPMV 응용의 경우, 대부분 실행 지연이 메모리 요구 지연 (memory dependency)으로 인하여 발생하지만, QS의 경우 메모리 요구 지연은 0.15%에 불과하며 대부분의 실행 지연이 실행 의존성 (execution dependency) 및 기타 지연으로 인해 발생하는 것을 알 수 있다. 그림 16-(a) 는 표 5의 응용 SPMV, QS, BS를 단독으로 수행시켰을 때의 커널 시간 결과 그래프이다. 그림 16-(b)는 SPMV를 QS와 함께 수행한 결과를, 그림 16-(c)는 SPMV를 BS와 함께 수행한 결과를 나타내며 가장 오른쪽 막대는 응용을 동시 수행하지 않고, 순차적으로 수행했을 때의 수행 시간을 보여준다. 같은 SPMV 응용을 실행하지만, 어떤 응용과 동시 수행하는 가에 따라 동시 수행 성능이 달라지는 것을 확인할 수 있다. SPMV 를 QS와 함께 수행하였을 때, 전체 수행시간이

순차 실행 시간보다 약 30% 향상했다. 하지만 같은 실행 지연이 발생하는 SPMV를 BS와 함께 수행하게 되면, 동시 실행 시간이 순차적으로 실행한 시간보다 30% 증가하여 성능이 오히려 나빠진 것을 확인할 수 있다. 이와 같이 서로 다른 자원을 사용하는 응용 또는 서로 다른 이유로 중단되는 응용을 함께 실행하여 SM 내부의 자원 활용도를 높일 수 있다.

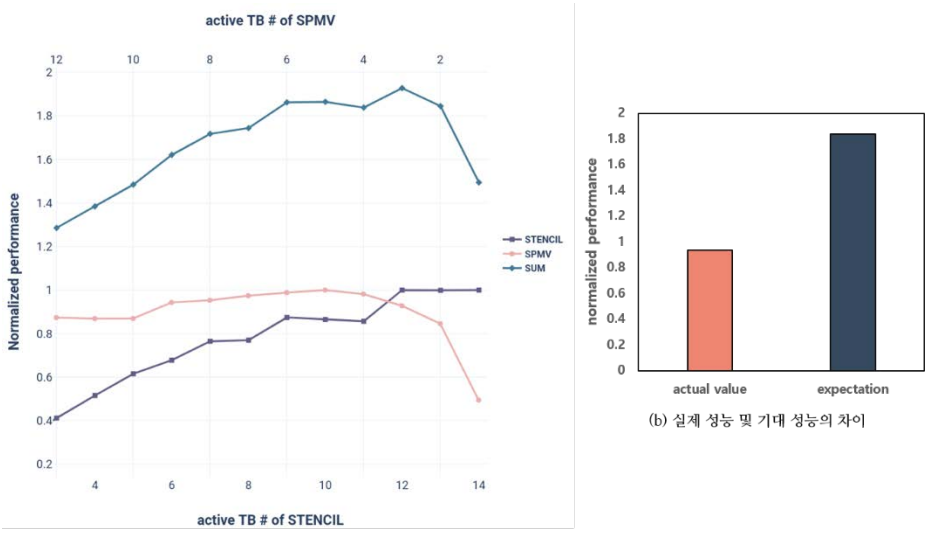
<표 6> 벤치마크 별 Intra-SM 자원 사용량 및 런타임 실행 지연의 이유

	Static Resource		**Runtime Stall Reason (%)**		
Application	Registers /Block	SMem /Block (byte)	Execution dependency	Memory dependency	Others
LavaMD (LM) [36]	7168	7200	94.3%	0.15%	4.54%
BlackScholes (BS) [35]	2944	0	6.36%	73.81%	3.88%
CUTCP [37]	3328	4019	18.23%	0.91%	52.74%
STENCIL [37]	4096	1000	10.45%	59.08%	7.88%
SPMV [37]	5148	0	15.59%	61.31%	15.48%
LBM [37]	4800	0	2.41%	14.9%	46.26%
FDTD3d (FT) [35]	5120	1500	13.86%	26.25%	38.6%
QuasiRandom Generator (QS) [35]	15360	0	22.43%	0.87%	43.26%
Needleman-Wunsch (NW) [36]	656	2180	29.17%	37.16%	6.43%
Hotspot3D (HW) [36]	8192	0	11.86%	78.18%	13.29%
DXTC (DX) [35]	4032	2048	46.11%	11.08%	3.90%
Binomial Options (BO) [35]	4096	516	13.03%	0.00%	27.94%
CP [37]	4224	0	35.96%	0.02%	33.42%
SGEMM (SG) [37]	5376	512	53.14%	13.38%	4.94%
Reduction (RD) [39]	4608	2000	31.12%	35.69%	3.06%
Covariance (COV) [38]	5632	0	2.10%	96.56%	0.68%
Syr2k (SY) [38]	6144	0	0.35%	71.45%	2.12%
Convolution-3D (CONV) [38]	5120	0	20.12%	16.55%	30.17%



<그림 16> 동시 수행 커널에 따른 다중 작업 성능 변화

3) 현존하는 다중 작업 배치 기법의 한계



(a) 성능에 대한 확장성 곡선

(b) 실제 성능 및 기대 성능의 차이

<그림 17> Warped-slicer의 확장성 곡선 및 성능 차이

2.3절에서 설명한 Warped-slicer는 최신의 Intra-SM 자원 공유 시스템이다. 이는 우수한 자원 분배 방법을 찾을 수 있지만 응용을 단독으로 실행되었을 때의 자원 할당량에 따른 성능만을 고려하여 한계가 있음을 그림 17을 통해 설명한다.

그림 17-(a)는 STENCIL 응용과 SPMV 응용의 확장성 곡선을 나타낸다. 각 응용의 성능 포화 지점은 4.1.1절에서 설명한 바와 같다. 각 응용을 Warped-slicer의 확장성 곡선에 대입하여 하드웨어 자원의 제약은 충족하면서 동시 수행의 성능을 최대화 할 수 있는 sweet spot을 식별한다. Sweet spot으로 대표되는 최적의 활성 스레드 블록 개수의 조합을 찾으면, STENCIL과 SPMV 각각 12개와 3개의 활성 스레드 블록을 가지며 이 때의 순차 수행 시간 대비 정규화된 성능은 1.93을 나타낸다. 하지만 그림 17-(b)와 같이 실제 성능은 기대 성능과 다르게 0.94의 성능을 나타낸다. 각 커널을 단독으로 수행한 결과만을 토대로 동시 수행의 성능을 예측하였기 때문에 함께 수행한 응용에서 발생할 수 있는 자원 경쟁이 반영되지 않았으며 이로 인해 기대 성능보다 실제 성능에서 성능 손실이 크다는 것을 확인할 수 있다. 따라서 응용 간 자원 배분의 방법만을 고려할 뿐 아니라 응용의 동시 수행에서 발생할 수 있는 자원의 경쟁 및 그 정도를 예측하여 실제 성능을 최대화할 수 있는 응용의 조합을 찾는 것이 중요하다.

2. 응용의 SM 내부 자원 사용 특성 분석

응용의 효율적인 다중 작업을 위해서는 SM 내의 자원을 효과적으로 분할하며 동시 수행의 성능을 최대화 할 수 있는 응용의 조합을 식별하여야 한다. 시간이 지날수록 대상 응용프로그램의 수가 많아지고, GPU 내의 자원의 수가 증가하므로 이 모든 조합을 탐색하는 것은 불가능하다. 그러므로 본 논문에서는 응용의 개별 실행 특성에 따라 응용을 분류하며, 각 응용의 성능이 포화 되는 시점 및 동시 실행하는 특성을 확인하고자 한다.

1) 응용의 분류

본 논문의 응용 분류는 각 응용 실행에 있어 발생할 수 있는 실행 지연의 이유를 사용한다. 주요 실행 지연의 이유는 다음과 같다.

- 실행 의존성 (Execution dependency) : 명령에 필요한 입력이 아직 준비가 되지 않아 일어나는 실행 지연을 의미한다.
- 명령어 페치 (Instruction fetch) : 다음 어셈블리 명령이 페치되기를 기다리는 중을 의미한다.
- 메모리 의존성 (Memory dependency) : 로드 및 저장과 관련된 자원이 사용 가능하지 않거나, 해당 자원이 완전히 사용 중일 때

발생하는 지연이다.

- 동기화 (Synchronization) : __syncthreads() CUDA API 호출로 인해서 워프가 블록 됨을 의미한다.
- 텍스쳐 캐시 (Texture cache) : L1 캐시가 완전히 사용 중이어서 워프가 대기한다.
- 기타 (Others): 이 외의 다른 자원들에서 충돌되거나 의존성이 발생하는 것을 포함한다.

각 실행 지연의 이유를 Nvprof를 통하여 프로파일링한다. 각 사이클 마다 워프에 실행 지연이 일어났다면, 각 이유에 해당하는 카운트를 증가시켜 백분율로 나타내며, 각 사이클 마다 적격 워프의 개수를 프로파일링한다. 이 때 최소, 평균, 최대 값의 분산이 크지 않으므로 평균 값을 사용하였다. 각 응용 별 실행 지연 이유의 백분율 및 사이클 별 적격 워프의 수(Eligible warps per cycle: EPC)는 표 6에서 보여준다.

본 논문은 메모리 의존성이 전체 지연의 백분율 중 S% 이상을 차지한다면 메모리 집약(Memory) 응용으로 분류한다. 텍스쳐 캐시로 인한 실행 지연이 백분율 중 S% 이상을 차지한다면 L1 캐시 집약(L1 cache) 응용으로 분류하며, 이 외의 응용들은 계산 집약(Compute) 응용으로 분류한다. S는 상수로써 실험 환경에 따라 달라질 수 있으며, 본 논문의 실험 환경에서는 이를 33%로 지정하였다. 이에 따라 분류한 결과는 표 7의 타입과 같다.

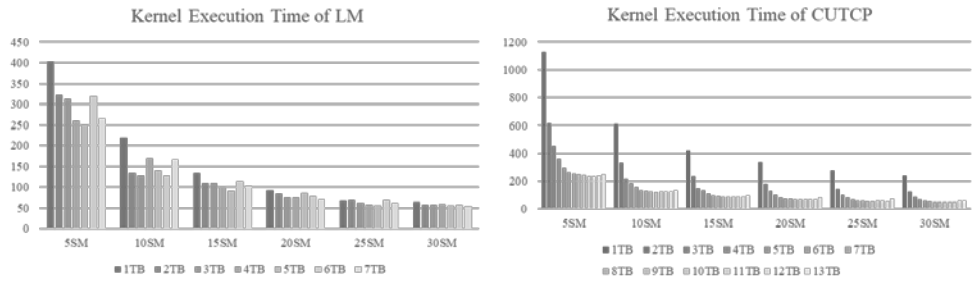
<표 7> 벤치마크 응용의 실행 지연 이유와 이에 따른 응용 분류

	Runtime Stall Reason (%)							
App.	Execution dependency	Intruction fetch	Memory dependency	Others	Synch-ronization	Texture cache	Eligible warps/cycle	Type
LM	94.3%	0.53%	0.15%	4.54%	0.46%	0%	0.17	Compute
BS	6.36%	1.89%	73.81%	3.88%	0%	0.2%	4.04	Memory
CUTCP	18.23%	13.83%	0.91%	52.74%	7.31%	0%	5.67	Compute
STENCIL	10.45%	1.99%	59.08%	7.88%	12.31%	0%	5.68	Memory
SPMV	15.59%	0.92%	61.31%	15.48%	0%	5.01%	0.72	Memory
LBM	2.41%	1.03%	14.9%	46.26%	0%	46.26%	0.59	L1 Cache
FT	13.86%	3.69%	26.25%	38.6%	17.3%	0%	0.55	Compute
QS	22.43%	2.12%	0.87%	43.26%	0%	0%	10.06	Compute
NW	29.17%	4.94%	37.16%	6.43%	20.82%	0.23%	0.31	Memory
HW	11.86%	1.30%	78.18%	13.29%	0.00%	0.01%	2.32	Memory
DX	46.11%	7.43%	11.08%	3.90%	20.27%	0%	3.75	Compute
BO	13.03%	9.25%	0.00%	27.94%	43.05%	0%	5.93	Compute
CP	35.96%	27.29%	0.02%	33.42%	0.00%	0%	3.41	Compute
SG	53.14%	3.46%	13.38%	4.94%	20.21%	0%	4.02	Compute
RD	31.12%	4.97%	35.69%	3.06%	12.89%	0%	0.53	Memory
COV	2.10%	0.57%	96.56%	0.68%	0%	0%	0.08	Memory
SY	0.35%	0.24%	71.45%	2.12%	0%	25.79%	0.12	Memory
CONV	20.12%	1.34%	16.55%	30.17%	0%	31.64%	0.86	L1 Cache

2) 정적 자원 할당에 따른 응용 수행 특성

그림 18은 계산 집약 응용에 해당하는 LM과 CUTCP 응용의 할당 자원 량에 따른 성능을 나타낸 그래프이다. 이들은 정수 및 부동 소수점 연산과 같은 계산 량이 많은 응용이다. 해당 응용들은 메모리 연산보다

계산 연산이 많기 때문에 실행 의존성, 동기화 및 명령어 페치에 의한 실행 지연이 발생한다.



<그림 18> LM / CUTCP의 활성 SM 개수 SM 당 제출된 스레드 블록 개수에 따라 달라지는 커널 실행 시간

하지만 같은 계산 집약 응용이라도 사이클 당 적격 워프의 수 (eligible warp per cycle: EPC) 에 따라서 자원 할당에 따른 성능 포화 지점이 다르다. 그림 18-(a)은 LM의 active SM / TB 별 커널 수행 시간을 보여준다. LM의 EPC는 0.17로써, 실행 지연이 많이 일어나 각 사이클 당 적격인 워프의 수가 평균 0.17인 응용이다. 이는 많은 active TB 수를 할당해주어도 실행 지연으로 인하여 성능 이득이 없는 것을 확인할 수 있다. 그림과 같이 30SM-1TB의 자원을 할당해 주었을 때의 커널 수행 시간은 64ms이며, 30SM-2TB를 할당해 주었을 때는 커널 수행 시간이 58ms로 2개의 스레드 블록을 활성화할 때까지는 성능 향상이 일어난다. 하지만, 30SM-4TB를 할당하면 58ms로 더 많은 자원을 할당해줘도 성능 향상이 일어나지 않는 것을 확인할 수 있다. 반대로 CUTCP는 프로파일링 결과 5.67의 EPC를 보인다. 그림

18-(b)에서 CUTCP는 30개의 SM을 활성화 하였을 때, 1TB를 할당해주면 약 237ms, 2TB를 할당해주면 약 123.974ms, 4TB를 할당해주면 약 69ms로 거의 선형적으로 성능이 향상한다. 10TB를 할당해주면 약 46ms로 성능이 향상되지만, 12개의 TB를 할당하면 57ms로 오히려 성능이 나빠지는 것을 확인할 수 있다. CUTCP는 약 10개 스레드 블록을 활성화 하면 성능이 포화 됨으로써 EPC가 크기 때문에 활성 스레드 블록 수가 많아져도 실행 지연의 영향을 많이 받지 않고 자원을 많이 할당한 만큼 성능이 향상 되는 것을 알 수 있다.

그림 15의 STENCIL 및 SPMV는 표6에서 해당 응용의 범주가 Memory로 명시된 응용들이다. 메모리 집약 응용들은 전역 메모리 영역으로 높은 처리량 및 활용률을 보인다. 이들은 메모리 요청이 많기 때문에 워프 가 다음 명령을 발행하지 못할 가능성이 높다. 특히, GPU DRAM에 대한 메모리 요청은 400~600 사이클이 소요되어 한 사이클 이내에 이동이 가능한 레지스터 및 공유 메모리에 비해 실행 지연을 더 많이 일으킬 수 있다.

메모리 집약 응용들도 EPC 값이 다르면 자원 할당 양이 증가 함에 따라 커널 수행 성능이 포화 되는 시점이 다르다. 그림 15는 SPMV와 STENCIL의 활성 SM / TB 별 성능을 보여준다. 4.1.1절에서 설명한 것과 같이 SPMV는 비교적 적은 자원 양으로도 최적에 가까운 성능을 달성한다. 하지만 STENCIL은 성능이 포화 되기 위해서는 더 많은 자원을 할당해주어야 한다. 이는 SPMV는 EPC가 0.72이기 때문에 더 많은 스레드 블록 할당해주어도 실행 지연으로 인하여 성능의 향상의 이득을 볼 수 없기 때문이다. 반면, STENCIL은 EPC가 5.68이기 때문에 많은 자원을 할당해주면 성능 향상을 기대할 수 있다.

관찰 #1 사이클 당 적격 워프의 수 (EPC)가 적은 커널은 많은 양의 자원을 할당해 주어도 실행 지연으로 인하여 성능 향상의 이득을 볼 수 없다.

3) 시스템 성능 요구량에 따른 응용의 동시 수행 특성

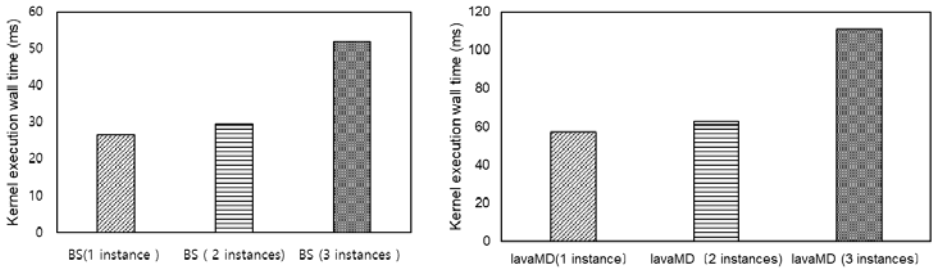
응용이 실행되는 환경에 따라 데이터 전송 및 계산에 대한 성능 공급량이 다르며 각 응용의 요구량 또한 다르다. 커널의 실행에 있어 필요한 데이터는 커널이 시작하기 이전에 호스트 메모리에서 디바이스 메모리로 전송된다. 커널의 실행 중에 필요한 데이터는 GPU DRAM에 미리 저장된 값을 읽어오게 된다. GPU는 CPU의 메모리 성능과 비교하면 상당히 빠른 속도의 메모리 전송 속도를 보여준다. CPU가 지원하는 메모리 모듈 중 가장 최신의 메모리 모듈인 DDR4가 64 bit의 버스 폭을 사용하는 것에 비해, GPU의 최신 메모리 모듈인 GDDR6X는 384 bit의 버스 폭을 사용하기 때문이다. 하지만 각 하드웨어마다 가용 대역폭이 다르며, 이를 넘어서게 되면 메모리 접근 시간이 응용 수행에 병목 지점이 된다.

초당 부동소수점 연산 횟수 (Float point Operations Per Second: FLOPS)는 GPU의 성능을 표현하는데 사용되는 지표이다. 딥 러닝 및 HPC와 같은 과학 연산에서 대부분의 계산들이 부동소수점으로 이루어져 있으므로 해당 지표에 따라 연산의 성능이 결정된다. 각 하드웨어마다 제공하는 연산 능력이 다르며, 이를 초과하면 컴퓨팅 성능이 병목 지점이 되어 커널 수행 시간이 증가하게 된다.

메모리 대역폭은 메모리 클럭 (Clock)과 버스 폭 (Bus width)의 곱으로써 결정되며, 일정 시간 동안 해당 버스를 통해 전송될 수 있는 데이터의 양을 뜻한다. FLOPS 수치는 덧셈기와 곱셈기 숫자의 합을 최대 동작 주파수를 곱해 계산하며, 이를 10^9 의 단위로 나타낸 것을 GFLOPS라고 한다. 본 실험 환경인 TITAN XP의 메모리 대역폭은 547.6GB/s 이며, GFLOPS는 단정밀도 (Single-precision)는 12.15 Tflops를, 배정밀도 (Double-presicion)는 379.7 Gflops 까지 지원한다.

하나의 응용은 보통 공급 성능을 모두 사용하지 않지만 두 개, 세 개의 커널 인스턴스를 동시에 수행하면 가용 대역폭 및 계산 량은 수요를 충족시키기에 충분하지 않은 경우가 발생한다. 그림 19는 하나의 인스턴스 당 266GB/s의 메모리 대역폭을 사용하는 BS 커널의 인스턴스 개수를 늘렸을 때의 성능과 하나의 인스턴스 당 200.26GFLOPS를 필요로 하는 LavaMD의 인스턴스 개수를 늘렸을 때의 성능을 보여준다. 그림 19-(a)와 같이 BS 두 개의 인스턴스를 동시 수행하였을 때는 각 인스턴스의 요구 대역폭 양의 합이 하드웨어의 공급 성능을 초과하지 않기 때문에 1개의 인스턴스를 수행하였을 때와 큰 차이가 나지 않는 것을 알 수 있다. 하지만 3개의 인스턴스를 동시 수행하여 대역폭 수요가 가용 대역폭 양을 초과하는 순간 동시 수행 성능이 급격하게 나빠지는 것을 알 수 있다. LavaMD의 인스턴스 수를 조절한 그림 19-(b) 또한 2개의 인스턴스를 동시 수행하여 공급 성능 이하의 부동소수점 연산을 수행하였을 때는 한 개의 인스턴스를 수행하였을 때와 시간 차이가 크게 나지 않으며 순차 실행과 비교해 거의 2배의 성능을 내는 것을 확인할 수 있다. 그러나 3개의 인스턴스 동시 수행을 통해 공급 성능을 초과하게 되면 부동소수점 연산에서 병목

현상이 일어나며 커널 수행 시간이 증가하게 된다. 이를 통해 하드웨어가 응용의 대역폭 및 계산 량 요구를 충족시킬 수 없다면 해당 자원이 포화 되므로 자원에 대한 경쟁이 일어나고 결과적으로 동시 수행의 성능이 순차 수행과 비슷해지거나 혹은 나빠진다.



(a) BS의 인스턴스 수 조절에 따른 커널 수행 시간 (b) LavaMD의 인스턴스 수 조절에 따른 커널 수행 시간

<그림 19> BS 및 LavaMD의 인스턴스 수 조절에 따른 커널 수행 시간

관찰 #2 각 응용의 누적 DRAM 대역폭 사용량 및 계산 량(GFLOPS) 이 하드웨어 성능 공급량을 초과하면 동시 수행의 성능 이득을 기대할 수 없다.

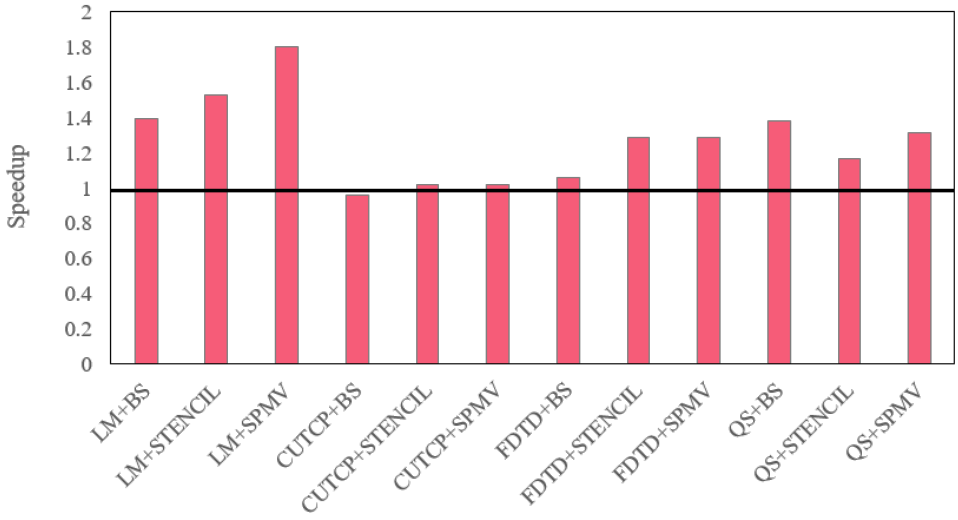
4) 응용 분류에 따른 동시 수행 특성

각 그룹의 응용 간에 페어링하고 해당 응용들을 동시 수행한 결과를 통해 응용 분류에 따른 동시 수행 특성을 분석한다. 각 그래프에서 플러스 (“+”) 앞의 응용은 첫 번째 그룹에 속하는 응용이며, 플러스 뒤의 응용은 두 번째 그룹에 속하는 응용이다. 해당 결과는 두 응용을

순차적으로 수행했을 때의 시간을 동시 수행 시간으로 정규화한 성능 향상도(speed up)를 나타낸다. 즉 1보다 크다면 순차 실행과 비교하여 성능 이득이 있으며, 1보다 작거나 같으면 성능 이득이 없거나 오히려 성능이 나빠짐을 의미한다.

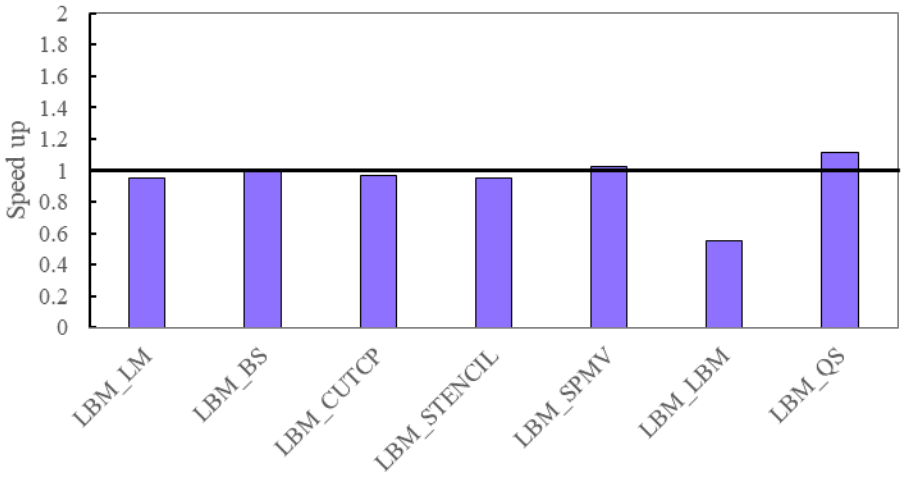
가) 계산 집약 응용과 메모리 집약 응용의 멀티태스킹

계산 집약 응용과 메모리 집약 응용을 함께 수행한 결과는 그림 20에 보여진다. 계산 집약 응용과 메모리 집약 응용을 함께 수행했을 때의 수행 시간은 순차 실행 수행 시간보다 평균 약 27% 단축되었다. 가장 성능이 좋아진 쌍은 LM+SPMV로, 성능이 약 80% 향상했다. 이는 LM과 SPMV의 EPC가 각각 0.17, 0.72로 각각 실행 의존성, 메모리 의존성으로 인하여 실행 지연이 많이 일어나는 응용이기 때문이다. EPC가 낮으므로 같은 응용 상에서는 실행 지연이 일어나 많은 자원을 주어도 성능 향상이 없지만 다른 자원을 사용하는 응용을 함께 배치한다면 서로의 실행 지연을 상호 보완 하여 성능 향상의 기회가 있음을 알 수 있다. 이와 같은 이유로 EPC가 가장 작은 LM과 함께 수행한 쌍들은 모두 약 40% 이상의 성능향상을 얻을 수 있었다. 반대로, CUTCP는 가장 EPC가 높은 응용으로 자원을 많이 할당해줌으로써 활성 스레드 블록 수를 증가시키면 커널 수행시간이 좋아졌다. 하지만 다중 작업의 성능과 순차 실행의 성능에 큰 차이가 나지 않았다. 이는 CUTCP만으로도 실행 지연이 많이 일어나지 않고 자원을 효율적으로 사용하며 워프를 계속 발행하므로 다른 응용의 워프를 수행할 기회가 많지 않기 때문이다.



<그림 20> 계산 집약 응용과 메모리 집약 응용의 다중 작업 성능

나) L1 캐시 집약 응용과 타 범주 응용 간의 멀티태스킹



<그림 21> L1 캐시 집약 응용과 타 범주 응용 간의 다중 작업 성능

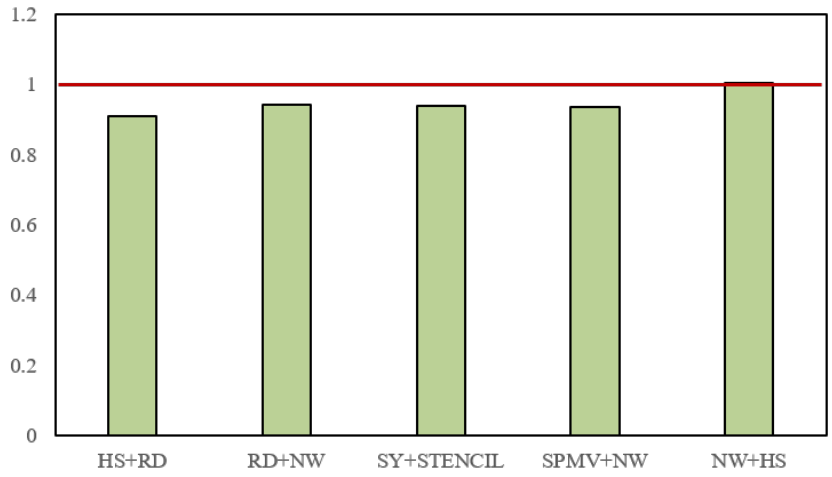
그림 21과 같이 L1 캐시 집약 응용의 경우 대부분의 응용들과 동시 수행했을 때 순차 실행과 비슷한 혹은 나쁜 결과를 보인다. 이는 이미 L1 캐시 집약 응용에게 L1 캐시의 용량을 다 사용할 만큼의 자원을 할당하였기 때문에, 다른 응용이 L1 캐시를 사용한다면 L1 캐시가 포화되기 때문이다. QS의 경우, L1 캐시 트랜잭션이 0에 가깝기 때문에 동시 수행하였을 때 약 11%의 성능 향상이 일어난 것을 알 수 있다. 반면, L1 캐시를 많이 쓰는 응용인 LBM을 두 개의 인스턴스로 실험하면 L1 캐시에 대한 경쟁으로 인하여 멀티태스킹 성능이 순차 실행보다 45% 감소하였다. 이를 통해 L1 캐시 집약 응용은 L1 캐시를 적게 사용하는 응용과 동시 수행할 때만 동시 수행의 이득을 기대할 수 있음을 알 수 있다. 이는 시뮬레이터를 사용한 [17] 논문에서도 관찰한 결과로, 실제 하드웨어에서도 시뮬레이터와 동일하게 동작하는 것을 관찰하였다.

관찰 #3 L1 캐시 집약 응용은 L1 캐시 트랜잭션 수가 기준점을 넘기는 응용과 동시 수행한다면, 다중 작업의 성능 이득을 기대할 수 없다.

다) 메모리 집약 응용 간의 멀티태스킹

그림 22은 메모리 집약 응용 간 동시 수행한 결과를 보여주는 그래프이다. 모든 쌍이 순차 실행과 동일한 혹은 나쁜 성능을 보인다.

표 8은 각 쌍의 DRAM 처리량의 합을 보여준다. NW+HS 쌍이 가장 좋은 성능을 보여 순차 실행보다 약 3%의 성능이 향상 하였으며, 가장 성능이 나쁜 HS_RD 쌍은 오히려 순차 성능보다 약 9% 성능이 감소하였다. 해당 결과를 통해 모든 쌍들의 DRAM 처리량의 합이 시스템이 제공하는 대역폭을 초과하지 않지만 순차 실행과 비슷한 혹은 더 나쁜 성능을 보여 동시 수행의 이득이 없는 것을 알 수 있다. 이는 DRAM 대역폭 외에도 캐시 및 공유 메모리 등 다른 메모리 시스템에서 자원의 경쟁이 일어나기 때문이라고 유추할 수 있다.



<그림 22> 메모리 집약 응용 간의 다중 작업 성능

	HS+RD	RD+NW	SY+STENCIL	SPMV+NW	NW+HS
sum of dram throughput (GB/s)	376.951	96.991	390.922	429.715	441.12

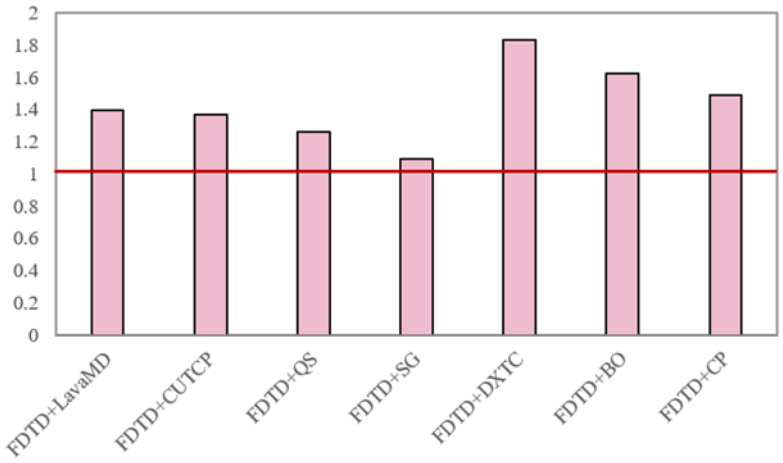
<표 8> 다중 작업 쌍별 DRAM 처리량의 합

관찰 #4 메모리 집약 응용 간 다중 작업은 순차 실행 이하의 성능을 보인다.

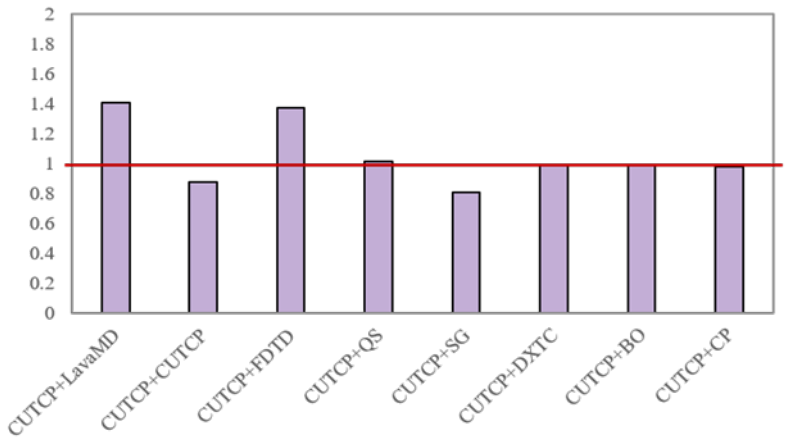
라) 계산 집약 응용 간의 멀티태스킹

그림 23는 계산 집약 응용 간 동시 수행한 결과를 보여준다. 이 실험의 결과를 통해 계산 집약 응용 간 동시 수행은 성능 이득의 기회가 있음을 알 수 있다. 하지만 모든 쌍에 있어서 순차 실행보다 성능이 향상하는 것은 아니다. 그림 23-(a)는 계산 집약 응용인 CUTCP와 그 외의 다른 계산 집약 응용과의 다중 작업 성능을 보여준다. LavaMD 및 FDTD 쌍을 뺀 나머지 응용에서 순차 실행과 비슷한 혹은 나쁜 성능을 보이는 것을 알 수 있다. 이는 CUTCP가 EPC가 가장 높은 응용 중 하나이므로 CUTCP 단독으로 수행하였을 때 실행 지연이 거의 일어나지 않으므로 동시 수행을 통해 감출 실행 지연이 많지 않기 때문임을 알 수 있다. 그러므로 CUTCP는 EPC가 1보다 작아 실행 지연이 많이 일어나는 LavaMD 및 FDTD와의 다중 작업에서만 성능이 향상하였다. 그림 23-(b)는 계산 집약 응용이며 EPC가 1보다 작은 FDTD와 그 외의 다른 계산 집약 응용 간의 다중 작업 성능을 보여준다. 이는 FDTD에서 발생한 실행 지연을 다른 응용과의 동시 수행을 통해 감출 수 있기 때문에 성능이 향상함을 알 수 있다. 이를 통해 두 응용이 모두 EPC가 크다면, 동시 수행을 통해 감출 실행 지연이 서로 많지 않기 때문에 성능 이득이 없지만, 한 응용이라도 EPC가 작다면 감출 실행 지연이 생기므로 동시 수행에 있어서 성능 향상을 기대할 수 있는 것을 관찰할 수 있다.

관찰 #5 계산 집약 응용 중 EPC가 기준점을 넘는 응용을 EPC가 기준점보다 큰 응용과 함께 실행하면 동시 수행을 통해 성능 향상을 기대할 수 없다.



(a) FDTD 와 계산 집약 응용의 다중 작업 성능



(b) CUTCP 와 계산 집약 응용의 다중 작업 성능

<그림 23> FDTD 및 CUTCP와 계산 집약 응용의 다중 작업 성능

3. 설계 및 구현

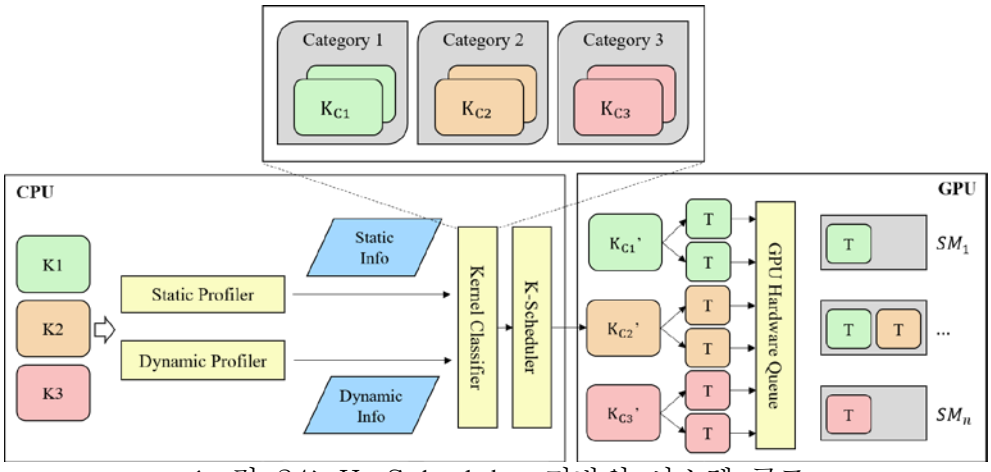
본 절에서는 Intra-SM 자원의 낮은 활용률 문제를 해결하기 위한 자원 공유 실행 프레임워크를 소개한다. 본 논문에서 제안하는 커널 스케줄러의 이름을 K-Scheduler라고 명명한다. 4.3.1 절에서는 K-Scheduler를 적용한 전체 구조 설계를 설명한다. 4.3.2절에서는 4.2절에서 관찰한 결과를 토대로 세부 규칙을 정의하고 이를 기반으로 한 K-Scheduler를 소개한다.

1) K-Scheduler 기반의 시스템 구조

전체적인 시스템 설계는 그림 24와 같다. 각 응용 마다 우세하게 사용하는 SM 내부 자원이 있으며, 서로 다른 우세 자원을 갖는 응용 간 함께 배치한다면 응용 공동 실행의 이득을 얻을 수 있다. 커널이 사용하는 SM 내부 자원을 포함하는 정적 프로파일링 정보는 커널 레벨에서 컴파일 타임에 NVIDIA CUDA Compiler (NVCC) [40]를 활용한 정적 프로파일러 (Static Profiler)로부터 획득한다. GPGPU 응용은 캐시 및 전역 메모리 대역폭 등의 자원을 공유하며, 서로 각기 다른 이유로 실행 지연이 일어나므로 정적 프로파일링 뿐 아니라 동적 프로파일링도 진행해야 한다. 동적 프로파일링 정보는 응용의 런타임에 NVProf를 활용한 동적 프로파일러 (Dynamic Profiler) 를 통해 획득한다. 획득한 프로파일링 정보를 활용하여 응용을 분류하고, 분류된 응용을 기반으로 스케줄링하는 K-Scheduler를 개발한다.

이를 위해 앞서 개발한 프로파일러로부터 수집된 정적 및 동적

프로파일링 정보를 활용하여 커널들을 미리 정의한 카테고리로 분류한다. 예를 들어 그림 24에서 K1 커널이 제출되었을 때 커널 분류기 (Kernel Classifier) 를 거쳐 커널 분류 정보를 포함한 K_{C1}으로 변형된다. 커널 분류 정보와 프로파일링 정보를 사용하여 K-Scheduler에서는 세부 규칙에 기반한 스케줄링을 진행한다. 그림 24와 같이 제출된 커널 K_{C1}이 스케줄러를 거쳐 GPU에 제출 될 때, 할당해 줄 자원의 양과 수행될 SM 배치 정보를 주입 받아 커널 K_{C1}'으로 변형된다. 이를 통해 자원 경쟁을 최소화하며 동시 수행된다. SM 내부 자원을 공유하며 자원 활용도를 최대화 하기 위해 커널은 스레드 블록 단위로 관리한다. 그림 24와 같이 자원의 양 및 배치 정보를 주입 받은 커널의 각 스레드 블록은 GPU의 하드웨어 큐에 태스크로 매핑되어 대기하며, 하드웨어 큐는 SM 배치 정보에 따라 스레드 블록을 해당 SM에 제출 한다.



<그림 24> K-Scheduler 기반의 시스템 구조

2) 자원 할당량 변화에 따른 성능 포화 지점 식별

본 절에서는 다중 응용이 동일한 SM에서 실행될 때 각 응용에 대한 자원 분배를 찾기 위한 방법을 소개한다. 4.2.2절에서 관찰한 바와 같이 GPGPU 응용은 할당해주는 자원의 양을 증가시켰을 때, 일정 시점에 도달하면 더 많은 자원을 할당하여도 성능이 더 이상 증가하지 않거나 오히려 성능이 나빠질 수도 있다. 또한, 자원 할당량 변화에 대한 성능 포화 지점은 응용의 특성에 따라 다르다. 다중 작업의 목적은 각 응용의 성능이 독립적으로 실행하였을 때와 큰 차이가 없음을 보장하면서 전체 처리량을 향상시키는 것에 있다. 그러므로 다중 작업에서 동시 수행의 이점을 극대화 하기 위해서는 각 응용의 성능 손실을 허용 가능한 수준으로 제한하고, 한 SM 내에서 최대한 많은 응용이 동시 수행하도록 해야한다. 이를 위해서는 자원 할당량을 증가시켜도 성능 향상이 되지 않는 혹은 미미한 성능 향상이 있는 성능 포화 지점을 식별하는 것이 중요하다.

본 연구에서는 각 응용에 할당해주는 자원의 양을 늘렸을 때 일정 비율의 성능 향상이 일어나지 않는다면, 성능의 포화 지점에 도달하였다고 판단하여 자원을 더 이상 할당해주지 않는다. 벡터 Perf는 각 응용의 활성 SM 및 TB 개수에 따른 성능을 저장하며, $Perf_{i,j}$ 는 i개의 활성 SM 및 j개의 활성 스레드 블록 수를 지정하였을 때의 성능을 나타낸다. $Rate_{tb}$ 는 성능이 포화 되는 지점을 판단하는 비율이며, 성능 손실을 허용하는 정도를 결정한다. 이 값이 작을수록 성능 손실을 적게 허용함을 의미한다. w는 윈도우 크기를 의미한다. 특정 응용의 경우 활성 스레드 블록 개수를 하나 증가시켰을 때는 성능이 많이

증가하지 않았지만 2개 또는 3개를 늘렸을 때 성능의 이득이 큰 경우가 있으므로 윈도우 크기만큼 보는 것으로 한다. 다음의 방정식을 만족하면 i개의 활성 SM 개수에 따른 성능이 포화 되는 스레드 블록의 개수를 j로 판단한다.

$$\mathrm{Perf}_{i,j} * (1 + \mathrm{Rate}_{tb})^{w} \geq \mathrm{Perf}_{i,j+w} \ , \qquad w = 1, 2, \dots, W\_SIZE$$

이는 w를 1부터 윈도우 크기인 W_SIZE까지 증가시킨다. i개의 활성 SM에 대해서 활성 스레드 블록의 개수가 j 에서 j+w로 증가하였을 때 일정 비율 (Rate_{tb}) 만큼 성능이 향상되는지 확인한다. 모든 w 값에서 위의 식을 만족한다면, 해당 윈도우에서 일정 비율만큼 성능이 향상되지 않은 것이므로 더 이상 자원을 주는 것이 큰 의미가 없다고 판단하여 성능이 포화 되는 스레드 블록 수를 j개로 판단한다.

해당 알고리즘을 적용하여 포화 지점을 찾기 위해서는 활성화 SM 개수 별, 활성화 스레드 블록 개수 별 성능 값이 필요하다. 자원 할당량에 따른 성능의 측정을 위해 [17] 에서 제안된 프로파일링 방법론을 사용하면, 작은 프로파일링 오버헤드를 통해 성능 값을 얻을 수 있다.

3) K-Scheduler

컴파일 시간에 결정되는 정적인 자원의 사용 양 뿐 아니라 복잡한 캐시 접근 패턴, 실행 패턴, 데이터 의존성 등을 반영하여 자원의 경쟁을 최소화하는 스케줄링 기법이 필요하다. 본 절에서는 동시 수행의 성능을 높이기 위해서 정적 자원 사용량 및 런타임 실행 패턴을 함께

고려하여 이질적인 여러 커널을 하나의 SM에 배치하는 K-Scheduler의 알고리즘에 대해서 설명하고, 해당 알고리즘이 적용되기 위해서 필요한 세부 규칙을 정의한다.

가) 알고리즘

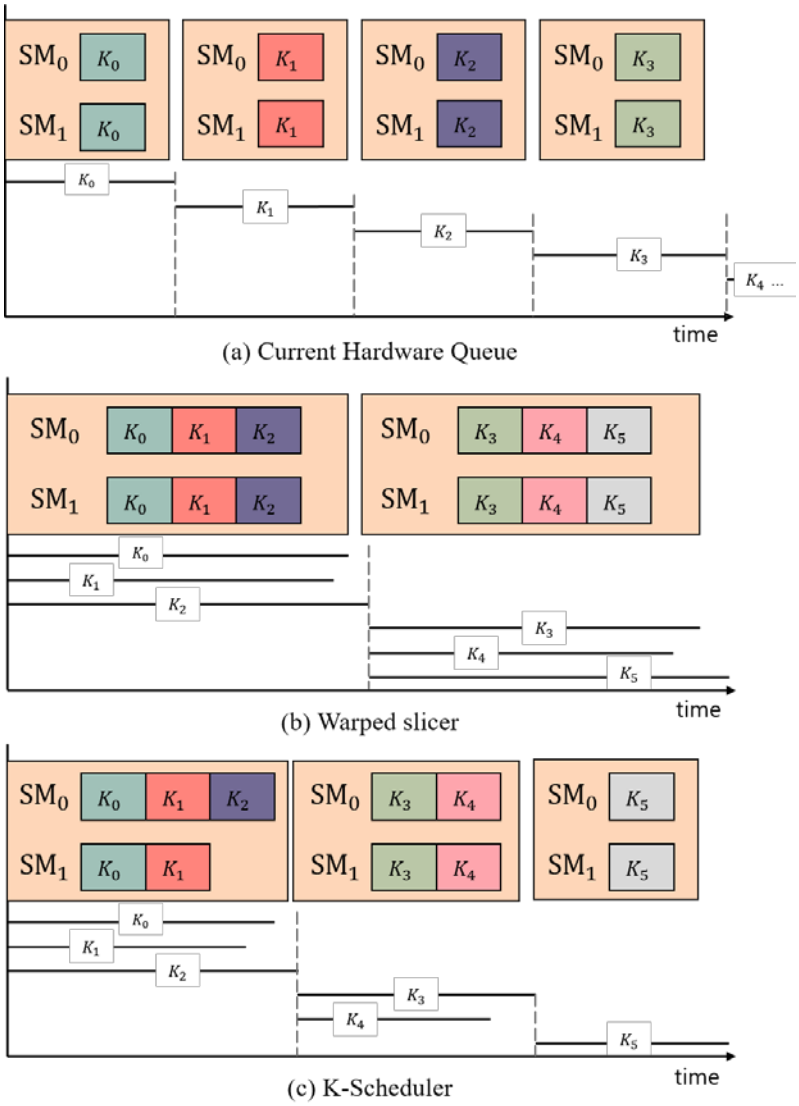
K-Scheduler의 전체 알고리즘은 그림 25와 같다. 해당 알고리즘의 입력은 성능 포화 지점 식별 및 분류가 완료된 커널의 집합인 워크로드이다. 입력으로 들어온 커널들을 대상으로 정렬을 실행한다. 정렬에 있어 L1 캐시 집약 응용이 가장 큰 우선 순위를 가지며, 메모리 집약 응용이 그 다음 우선 순위를, 계산 집약 응용이 가장 작은 우선 순위를 가진다. 같은 범주 안의 응용이어서 같은 우선 순위를 가진다면, 긴 작업을 우선으로 하여 정렬한다 (line: 2). SK_List에 존재하는 커널이 없을 때까지 반복하여 동시 수행할 응용의 조합을 찾는다 (line: 3). 해당 반복에서 찾을 응용의 조합을 CK라 하며 공집합으로 초기화 한다(line: 4). CK에 들어갈 커널을 찾기 위해 SK_List의 원소인 SK^i 를 대상으로 세부 규칙 # 1,2,3,4,5 를 만족하는지 확인하고, 모든 세부 규칙을 만족하여 동시 수행할 커널로 판단되면 SK_List에 존재하던 SK^i를 CK로 이동한다(line: 5-9). For 반복문이 마치게 되면 다중 작업할 커널의 조합인 CK가 구성되므로 이를 CK_List에 추가한다 (line: 10). 알고리즘의 출력은 동시 수행할 커널의 조합의 리스트인 CK_List이며, 해당 리스트의 순서 및 조합대로 다중 작업을 실행한다.

K-Scheduler()

1. **Input** : $Workload = \{K^1,\ K^2, \dots, K^k\}$
2. SK_List ← *sort_kernels* (Workload) ▷ SK_List = $\{SK^1,\ SK^2, \dots, SK^k\}$
3. **while** SK_List is not empty **do**
4. $CK \leftarrow \emptyset$
5. **for each** SK^i in SK_List **do**
6. **if** rule #1 **and** rule #2 **and** rule #3,4,5 is satisfied **then**
7. move SK^i to CK
8. **end if**
9. **end for**
10. add CK to CK_List
11. **end while**
12. **Output :** CK_List ▷ CK_List = $\{CK_1 = \{SK^1, \dots, SK^m\},\ CK_2, \dots, CK_n\}$

<그림 25> K-Scheduler 알고리즘

그림 26은 K-scheduler의 스케줄링 방식을 묘사하고 및 기존 스케줄링 방법과의 비교를 보여준다. 그림 26-(a)는 기존 하드웨어 큐에서 지원하는 단독 실행의 방식이며 그림 26-(b)는 Warped-slicer 방식이고, 그림 26-(c)는 K-Scheduler의 방식이다. Warped-slicer의 동시 수행 커널의 개수는 항상 K개로 고정되어 있으며, 본 그림의 스케줄링 레이아웃은 K를 3으로 지정한 것이다.



<그림 26> 스케줄링 방식 비교

나) 세부 규칙 #1

세부 규칙 #1은 정적 프로파일링 정보를 고려하여 GPU에 충분한 정적 리소스가 있는지 확인한다. 4.2.1절에서 설명한 바와 같이 하나의 SM에 활성화 될 수 있는 최대 스레드 블록 수는 GPU의 정적 리소스 제약 (레지스터의 개수, 공유 메모리의 크기, 최대 스레드 블록 개수 제한) 에 의해 결정된다. 따라서 세부 규칙 #1에서는 K^i 에 대한 스케줄링을 진행함에 있어 현재까지 동시 수행되도록 선택한 커널의 집합이 $CK = \{CK^1, CK^2, \dots, CK^m\}$ 이며, m은 현재까지 선택된 커널의 개수일 때 정적 프로파일링 정보에 기반하여 K^i가 CK 집합의 원소로 추가될 수 있을지 다음과 같이 판단한다:

$$\sum_{j=1}^{m} CK_{reg}^{j} + K_{reg}^{i} < \mathrm{MAX_{REG}}$$

$$\sum_{j=1}^{m} CK_{smem}^{j} + K_{smem}^{i} < \mathrm{MAX\_SMEM}$$

$$\sum_{j=1}^{m} CK_{tb}^{j} + K_{tb}^{i} < \mathrm{MAX\_TB}$$

첫 번째 식을 통하여 현재까지 선택된 커널이 요청한 레지스터의 개수의 합과 K^i 가 요청한 레지스터 개수인 $\mathrm{K}_{reg}^{\mathrm{i}}$ 의 합이 현재 하드웨어가 제공할 수 있는 최대 레지스터 개수를 MAX_REG를 초과하지 않는지 확인한다. 두 번째 식은 선택된 커널의 공유 메모리 크기 합과 K^i가 사용하는 공유 메모리 크기인 K_{smem}^{i}를 더한 값이 최대 공유 메모리 크기인 MAX_SMEM 보다 작은지 확인한다. 세 번째 식은

CK 집합의 원소인 커널들의 지정된 스레드 블록 개수의 합과 K^i 의 지정된 스레드 블록 개수를 합하였을 때 스레드 블록의 최대 개수인 MAX_TB 미만인지 확인한다. 세 식을 모두 만족하면 세부 규칙 #1을 만족하여 K^i 가 정적 프로파일링 정보만을 고려하였을 때 CK의 커널들과 동시 수행 가능함을 의미한다.

다) 세부 규칙 #2

세부 규칙 #2는 관찰 #2에 따라 시스템 성능 요구량을 고려하여 GPU에 충분한 가용 대역폭 및 FLOPS 공급량이 있는지 확인한다. 앞에서 설명한 바와 같이 현재까지 선택된 커널의 집합을 CK이며, K^i에 대한 스케줄링을 진행할 때 시스템의 이론상 DRAM 대역폭 및 GFLOPS 공급 성능을 해당 커널을 스케줄링 했을 때 초과하지 않는지 다음 식을 통하여 확인한다:

$$\sum_{j=1}^{m} CK_{BW}^{j} + K_{BW}^{i} < \text{MAX\_BW}$$

$$\sum_{j=1}^{m} CK_{GFLOPS}^{j} + K_{GFLOPS}^{i} < \text{MAX}_{\text{GFLOPS}}$$

첫 번째 식을 통하여 현재까지 선택된 커널들의 대역폭 요구량을 합한 값과 K^i를 수행하는 데 있어 필요한 대역폭 요구량을 더하였을 때 하드웨어가 제공하는 DRAM 대역폭을 초과하지 않는지 확인한다. 또한 두 번째 식을 통하여 현재 커널과 선택된 커널들의 요구 계산 성능이

시스템이 제공하는 GFLOPS 공급량 미만일 때 동시 수행하도록 스케줄링 한다. 세부 규칙 #2의 만족 여부 확인을 통해 두 식을 모두 만족하는지 확인하여 커널들의 시스템 성능 요구량과 시스템의 공급량을 비교하여 동시 수행 여부를 결정한다.

라) 세부 규칙 #3, 4, 5

관찰 #3, 4, 5를 기반으로 세부 규칙 #3, 4, 5를 다음과 같이 정의한다.

세부 규칙 #3: L1 캐시 집약 응용은 L1 캐시 트랜잭션 수를 기준으로, 기준점 이하의 트랜잭션 수를 가지는 응용과 동시 수행한다.

세부 규칙 #4: 메모리 집약 응용 간에는 동시 수행하지 않는다.

세부 규칙 #5: 계산 집약 응용 중 EPC 가 기준점 (Max threshold)을 넘는 응용은 EPC가 기준점 (Base threshold)보다 큰 응용과 함께 수행하지 않도록 한다.

세부 규칙 #3, 4, 5가 적용되는 방법은 그림 27의 알고리즘을 통해 보여준다.

현재까지 선택된 커널의 집합을 CK이며, K^i 에 대한 스케줄링을 진행할 때, K^i 의 범주에 따라 스케줄링을 진행한다(line: 2). 먼저 K^i 가 L1 캐시 집약 응용이라면, *L1_in_CK*() 함수를 통해 CK에 L1 캐시 집약 응용이 있는지 확인하고, *L1_in_CK*()가 참이라면 L1 캐시 집약 응용 간에는 함께 배치될 수 없으므로 해당 알고리즘은 거짓을 반환한다(line:

3-7). K^i 가 L1 캐시 집약 응용이 아니라면, 가장 먼저 CK에 L1 캐시 집약 응용이 있는지 확인한다. L1 캐시 집약 응용이 있다면 L1 캐시 트랜잭션을 기준으로 L1 캐시 집약 응용과 함께 배치될 수 있는지 검사하는 *Can_placed_with_l1*(K^i) 을 호출하며, L1 캐시 트랜잭션이 기준선을 초과하여 다중 작업이 불가능하다면 세부 규칙 #3에 따라 해당 알고리즘은 거짓을 반환한다 (line: 9-10, 16-17). 이로써 공동 수행에 있어 성능이 가장 나빠질 수 있는 L1 캐시 집약 응용이 가장 높은 우선 순위를 가지고 스케줄링 된다. K^i가 메모리 집약 응용이라면, *M_in_CK*() 함수를 통해 CK에 메모리 집약 응용이 있는지 확인한다. 해당 함수가 참을 반환 하였다면, 세부 규칙 #4에 따라 메모리 집약 응용 간 동시 수행은 허용되지 않으므로 본 알고리즘은 거짓을 반환한다 (line: 11-13). K^i 의 범주가 Compute 라면, C_*over_max_in_CK*() 를 통해 EPC 가 기준점 (Max threshold) 을 넘는 응용이 CK에 있는지 확인한다. 해당 조건이 만족하였을 때 세부 규칙 #5에 따라 K^i_{epc} 가 BASE_THRSHLD를 초과한다면, 거짓을 반환한다 (line 18-19). 기준점 (Max threshold)을 넘는 응용은 없지만, 기준점 (Base threshold) 을 넘는 커널이 있다면 세부 규칙 #5에 따라 K^i_{epc}가 MAX_THRSHLD를 초과하는지 검사하며 초과한다면 거짓을 반환한다(line 20-21). 위 세부 조건을 모두 통과하였다면 K^i는 CK의 커널들과 자원의 경쟁을 최소화하면서 동시 수행의 이득을 얻을 수 있으므로 참을 반환한다.

Rule #3,4,5

1. **Input :** K^j, CK
2. **switch** K_{cat}^i **do**
3. **case** L1 cache
4. **if** *L1_in_CK*() **then**
5. **return** false ▷ Rule #3
6. **case** Memory
7. **if** *L1_in_CK*() **and** not *Can_placed_with_l1*(K^i) **then**
8. **return** false ▷ Rule #3
9. **else if** *M_in_CK*() **then**
10. **return** false ▷ Rule #4
11. **case** Compute
12. **if** *L1_in_CK*() **and** not *Can_placed_with_l1*(K^i) **then**
13. **return** false ▷ Rule #3
14. **else if** C_*over_max_in_CK*() **and** K_{epc}^i> BASE_THRSHLD **then**
15. **return** false
16. **else if** C_*over_base_in_CK*() **and** K_{epc}^i> MAX_THRSHLD **then**
17. **return** false ▷ Rule #5
18. **return** true

<그림 27> 세부 규칙 #3, 4, 5 적용 알고리즘

4. 실험 결과

1) 실험 방법

가) 실험 환경

실험은 3.5.1절 표3 및 표4의 GPU 노드의 환경에서 진행하며, smCompactor[18]의 스레드 블록 기반 스케줄링 프레임워크 위에 해당 K-Scheduler를 구현한다.

나) 실험 워크로드

실험에 사용된 응용들은 표 x와 같으며 NVIDIA CUDA Sample[35], Rodinia GPU benchmark suite[36], Parboil benchmark[37], Polybench[38], SHOC benchmark[39] 의 벤치마크이다. 모든 응용들은 표준 데이터 입력 셋을 사용하여 수행되었다.

해당 벤치마크들을 대상으로 9개의 워크로드를 선정하였다. 각 워크로드의 특성은 표 9와 같다.

<표 9> 워크로드 순서 별 커널의 특성

Workload sequence	Characteristics of workload
W1	Compute intensive applications
W2	Memory intensive applications
W3	Compute intensive applications + Memory intensive applications (1 : 1)
W4	Compute intensive applications + Memory intensive applications (2 : 1)
W5	Applications with large number of EPC (EPC > 1)
W6	Applications with small number of EPC (EPC < 1)
W7	Applications with large number of EPC + Applications with small number of EPC (1:1)
W8	Applications with large number of EPC + Applications with small number of EPC (1:2)
W9	Applications with large number of EPC + Applications with small number of EPC (2:1)

다) 평가 지표

- 가중 속도 향상 (Weighted speedup) [17] : 전체 수행 시간을 순차 수행 시간으로 정규화 한 값이며, 높을수록 좋은 성능을 보인다.

- 평균 반환 시간 (ANTT: Average normalized Turnaround Time) [42] : 각 응용의 반환 시간 (Turnaround time)의 평균이며, 낮을수록 좋은 성능을 보인다.
- 공정성 (Fairness) [43] : $\frac{\text{Min speedup}}{\text{Max speedup}}$ 최소 speedup과 최대 speedup의 차이를 보여주며, 0부터 1사이의 값을 갖고, 1에 가까울수록 공정함을 의미한다.

라) 비교 대상

비교 대상 스케줄러는 다음과 같다. 각 응용에 동일한 양의 SM 내부 자원을 분배해주는 스케줄러인 Even partitioning 스케줄러와 2.3절 및 4.1.3절에서 설명한 이전 작업인 Warped Slicer[17]와 비교한다.

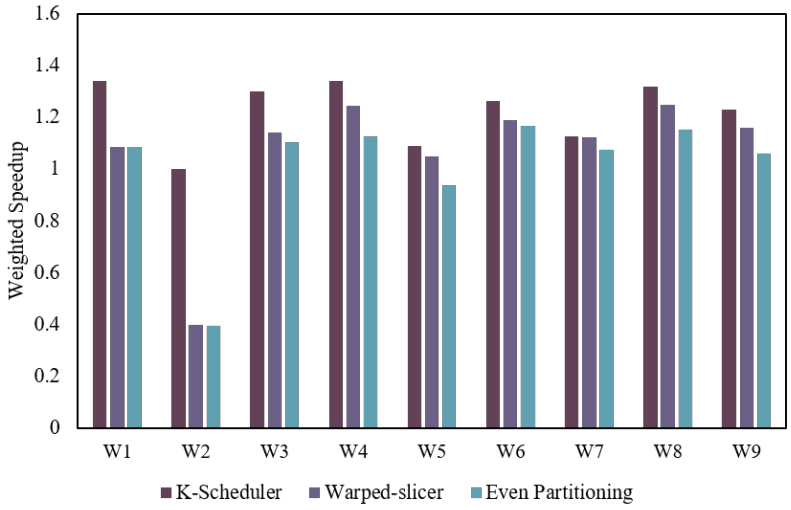
2) 스케줄링 성능

가) 가중 속도 향상 비교

그림 28은 K-scheduler, Warped slicer, Even partitioning의 가중 속도 향상을 보여주며, 순차 수행 시간은 1로 표현된다. 먼저 Even partitioning은 기존의 순차 수행하는 하드웨어 큐의 방식 보다는 자원 사용률을 높여서 평균 약 8%의 성능 향상을 보인다. Warped-slicer는

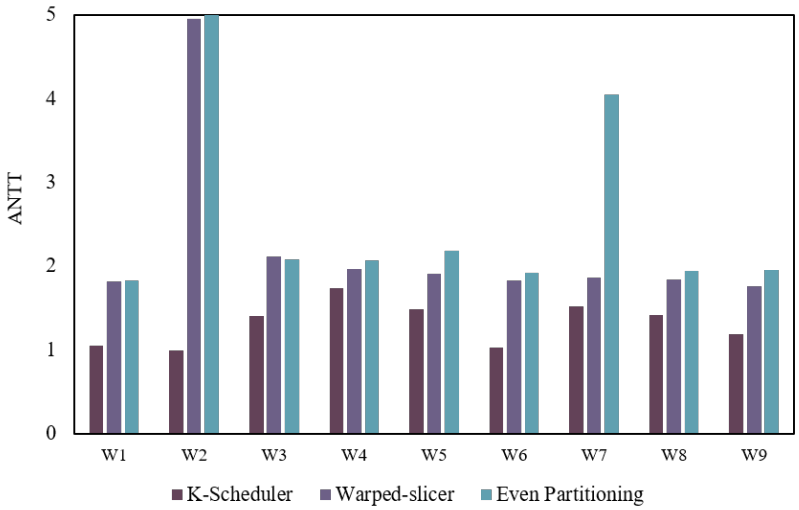
각 응용의 자원 할당량에 따른 성능 손실을 인식하여 스케줄링 하여 Even partitioning 방법보다 성능이 향상되어 평균 약 1.15의 가중 속도 향상을 보여준다. 하지만 이는 각 응용 간의 자원의 경쟁을 고려하지 않았기 때문에 이론과 다른 실제 성능을 보였으며 성능 향상의 한계가 있음을 보여준다. 한편, 응용의 자원 할당량에 따른 성능 및 응용 간 자원 경쟁을 최소화 하는 K-Scheduler는 순차 실행보다 31%의 성능 향상을 보여주며 가장 좋은 성능을 보여준다.

전체 워크로드 중 워크로드 1은 계산 집약 응용으로 구성된 워크로드이다. 해당 워크로드에 대해서 Even partitioning과 Warped slicer의 성능 차이가 거의 나지 않음으로써 계산 집약 응용 간의 다중 작업은 자원의 분배하는 방법보다는 어떤 응용과 함께 동시 수행하는지 정하는 것이 더 중요하다는 것을 알 수 있다. 본 연구의 스케줄러는 계산 집약 응용 간에 동시 수행하면 성능의 이득을 얻을 수 있지만 동시 수행으로 인하여 자원의 경쟁이 생길 수 있음을 인지하고 스케줄링 하므로 해당 워크로드에서 성능을 최대화할 수 있었다. 워크로드 2는 메모리 집약 응용으로만 이루어진 워크로드로써, 메모리 집약 응용 간의 다중 작업은 동시 수행의 이득이 없음을 보여준다. K-Scheduler는 메모리 집약 응용 간 동시 수행하지 않으므로 순차 수행과 같은 성능을 보인다. 워크로드 7의 경우 EPC가 작은 응용과 EPC가 큰 응용이 1:1로 이루어진 워크로드로써, 스케줄러 간의 성능차이가 많이 나지 않는다. EPC가 큰 응용이 EPC가 작은 응용과 함께 수행되어 동시 수행의 이득을 얻는다. 한편, 이미 워크로드에서 각 응용이 같은 비율로 잘 섞여 있으므로 K-Scheduler는 Warped slicer 및 Even partitioning의 성능과 비교하여 성능 향상할 기회가 많지 않음을 보인다.



<그림 28> 각 스케줄러의 가중 속도 향상 비교

나) 평균 반환 시간 비교



<그림 29> 각 스케줄러의 ANTT 비교

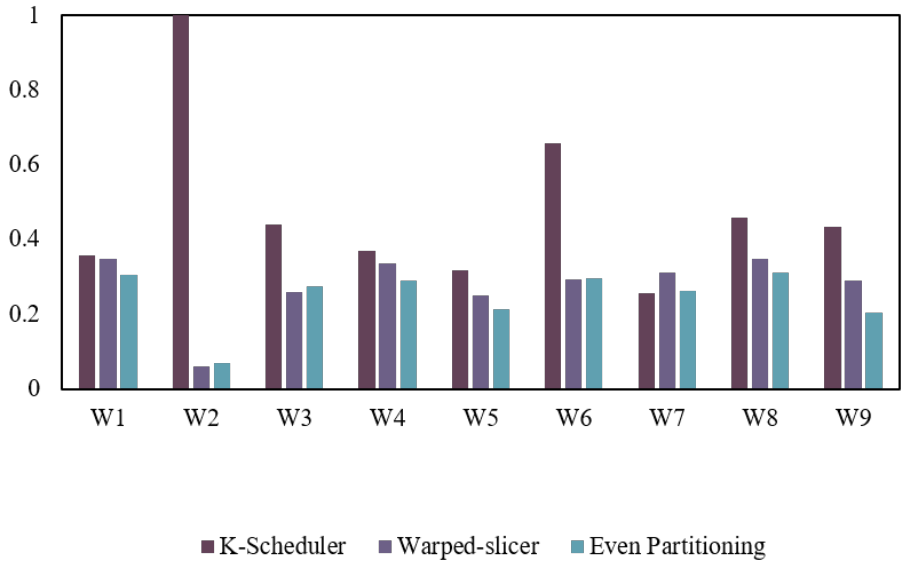
그림 29와 같이 ANTT의 측면에서 K-Scheduler는 평균 1.31의 ANTT를 보임으로써 개별 응용에 대한 QoS (Quality of Sevice)를 보장하는 반면, Warped slicer는 2.22의 ANTT를, Even partitioning은 2.56의 ANTT를 보인다. 특히, 앞 절에 서 언급한 워크로드 7의 경우 K-Scheduler가 Warped slicer와 Even partitioning과 비교하여 Weighted speedup 은 각각 약 0.3%, 5% 만 향상 되어 전체 워크로드의 성능 향상은 크지 않았지만 ANTT는 각각 22%, 166% 향상 되어 스케줄링의 필요성을 보인다.

본 연구의 스케줄러는 자원의 경쟁을 최소화하므로, 전체 워크로드에 대해서 성능을 향상 시킬 뿐 아니라 하나의 응용에 대해서도 독립적으로 실행할 때와 비교하여 성능 저하가 크게 되지 않도록 한다. 이로 인해 사용자는 빠른 응답을 얻을 수 있다. 그러나 Warped slicer는 Sweet spot 식별을 통해서 워크로드의 전체 성능을 높이고자 했으나, 동시 수행으로 인한 자원 경쟁을 고려하지 않아 성능 향상에도 한계가 있었으며 각 응용의 측면에서 서비스 응답 시간은 보장되지 않는다. 또한, Even partitioning은 정적 자원만 고려하여 균등하게 배분하는 스케줄링을 하였으므로 전체 성능이 순차 실행과 비교하여 크게 향상되지 않았으며 개별 응용 측면에서는 2.56배 slowdown 되어 응답 시간이 크게 지연되는 것을 보여준다.

다) 공정성 비교

공정성은 1에 가까울수록 각 응용의 Speedup 간에 편차가 크지 않음을 의미한다. 그림 30과 같이 K-Scheduler의 공정성은 0.3,

0.27의 공정성을 보여주는 Warped slicer 및 Even partitioning에 비교하여 0.47로써, 각 응용 간에 보다 공정한 자원 공유가 가능함을 보여준다. Warped slicer의 경우 2개의 쌍으로 다중 작업을 실행하면, 공정성이 약간 높아지나 가중 속도 향상 및 ANTT 측면에서 성능이 좋지 않다. 워크로드 7에서는 하나의 쌍이 수행 시간이 긴 응용과 짧은 응용이 함께 선택되어 공정성이 낮아졌지만 해당 워크로드를 제외한 모든 워크로드에서 K-Scheduler가 가장 공정한 결과를 보여준다.



<그림 30> 각 스케줄러의 공정성 비교

V. 관련 연구

1. Inter-SM 자원 공유 스케줄링

GPU 클러스터 및 클라우드 서버의 자원 활용을 높이기 위해 GPU 공유 기술이 많이 소개되고 있다. Diab [44]의 경우, 여러 사용자들이 GPU 자원을 공유하며 작업을 동시에 실행 가능한 시스템을 제안한다. CUDA API를 가로채 두개의 커널이 실행가능하도록 제공한다. 그러나 이러한 스케줄링 방법은 자원 사용 특성이 반복적이거나, 뚜렷한 형태를 보이는 응용 대상으로만 가능하다. [45-46]은 GPU 자원을 일정한 크기로 분류하고 비용 트리를 구축하여 실행하는 컨테이너에 할당하며 이에 따라 GPU를 지정하여 제공하는 방식으로 공유하고 있다. 그러나 이는 실행 중인 작업의 최소 자원 요구량 만을 고려하기 때문에 공동 실행할 경우 성능 저하가 발생할 수 있다. 우리는 머신 러닝 응용 뿐만 아니라 자원 사용 패턴이 뚜렷하지 않은 HPC 응용을 대상으로 하여 GPU를 공유한다. 또한 실제 응용의 여러 자원 사용 정보를 수집하여 이를 기반으로 공동 실행 하므로 자원 경쟁을 피할 수 있다. [47]은 GPU 활용도, PCI-E 대역폭, 메모리 사용량을 예측하여 QoS를 보장하며 에너지 효율성을 최소화하였다. 딥 러닝 워크로드는 정적인 자원 사용량을 보여서 예측가능하지만, HPC 응용과 같이 동적으로 자원 사용량이 계속 변하는 응용은 예측하기 어려우므로 본 논문에서는 프로파일링 및 모니터링을 통한 자원 프로비저닝을 진행하였다. 또한, 본 논문에서의 목표는 에너지 효율성 보다는 성능에 있으며 성능을

높이기 위해서는 GPU 활용도, PCI-E 대역폭, 메모리 사용량으로는 부족하다.

2. Inter-SM 다중 작업의 간섭 인지 스케줄링

GPU에서 여러 응용을 실행 시킬 수 있는 기술들이 등장하면서, 이러한 환경에서 공동 실행 시 발생할 수 있는 자원 경쟁 회피를 위한 스케줄링 기법이 많이 소개되고 있다. [33]에서는 응용 공동 실행을 위해 CF(Collaborate Filltering) 기반 간섭 인식 스케줄러를 제안한다. 간섭에 영향을 미치는 메트릭들을 프로파일링 하고, 새로운 응용의 경우 경량 프로파일링 후 CF를 사용하여 예측한다. 이는 각 응용 쌍의 간섭 값을 구하는 방법이 메트릭 벡터의 유사도에 기반한 방법으로, 유사도가 낮으면 간섭이 낮다고 판단하였다. 각 메트릭마다 간섭에 영향을 미치는 정도가 달라 가중치가 다를 수 있으나 이를 반영하지 않고 단순히 유사도를 구하였으며 GPU 응용 동시 실행에 있어서 발생할 수 있는 문제인 OOM 실패를 고려하지 않았다. [34]는 딥 러닝 응용의 특성을 분석하고 이에 따라 성능 예측 모델링을 진행하였으며 QoS인지 스케줄러를 제안하였다. 이는 딥 러닝 도메인에 대한 특별한 지식을 필요로 하므로 HPC 응용과 같은 모든 응용에는 적용 불가능하다. Xu [48]는 GPU에서 실행하는 응용의 특성들의 피쳐 (feature)로 정의하여 머신 러닝 기반 간섭 인지 스케줄러를 구현하였다. 단순한 응용인 경우 커널의 길이에 따라 성능이 많은 영향을 받지만, 머신 러닝 응용의 경우에는 간섭이 달라질 수 있음을 보였다. 그러나 실제 평가를 자원 사용 패턴이 반복적인 머신 러닝 응용 대상으로 하여 수행하였기 때문에

영향을 받는 자원들의 추가적인 피쳐들의 정의가 필요하며 VM을 대상으로 하여 컨테이너 환경을 위한 다른 메트릭의 정의가 필요하다. 또한, 간단한 라운드 로빈 (Round-robin) 스케줄러를 구현하였으므로 간섭 값을 클러스터 스케줄링 방식에 적용할 방법이 필요하다. [49] 논문은 GPU 서버가 있는 클러스터 환경에서 DRL 모델을 사용한 작업 배치 프레임워크를 제안한다. 학습은 작업자 id, CPU, GPU을 입력으로 하여 학습시켜, 여러 응용이 공존할 때 낮은 수준의 간섭으로 작업을 배치한다고 설명한다. 그러나 CPU, GPU의 사용량 값으로 간섭의 영향을 줄이기에는 고려해야할 자원의 세부 정보가 더 필요하다. [50] 논문에서는 간섭 값을 고려하고, max-pair 알고리즘을 사용하여 전체 워크로드의 수행 시간을 줄였다. 하지만 이는 새로운 응용이 들어온다면 그 때마다 모든 pair의 쌍을 수행해야 하므로 시간이 너무 오래 걸린다는 단점이 있다. 본 논문에서는 프로파일링 정보만 있으면 모든 쌍을 수행해 보지 않아도 간섭 값이 예측 가능하도록 하였다.

3. Intra-SM 다중 작업 스케줄링

NVIDIA 가 공간 다중화 기법을 지원하는 Hyper-Q 기술[13]을 소개한 이후, 이를 활용하여 효과적으로 공간 다중화 하는 방법에 대한 연구가 계속되고 있다. 공간 다중화 기법에는 SM의 부분 집합을 분할하여 쓰는 방법[51-53], SM 내부의 공간을 분할하여 쓰는 방법으로 나누어진다[17,42-43].

[51] 논문은 SM의 부분 집합을 효율적으로 나눠 쓰기 위해 응용을

계산 집약과 메모리 집약 응용으로 분류한다. 이전의 논문들은 DRAM 처리량만으로 응용을 분류했다면, 이 논문은 off-SM 대역폭도 고려하여 응용을 분류하였다. 또한, 메모리 집약 응용은 적은 SM 개수를 할당해줘도 비슷한 성능을 낼 수 있음에 착안하여 전력 절약할 수 있는 전원 모드 (Power mode)를 제안하였다. [52]는 화이트 박스 방식 및 블랙 박스 방식을 결합하여 성능 모델을 만들어서, 각 응용이 독립적으로 수행하였을 때와 비교하여 SM 부분 집합을 분할하여 동시 실행했을 때의 성능을 예측한다. 이는 DRAM 대역폭 활용도를 예측하고, 이를 기반으로 공정성 달성 및 QoS를 달성하는 다중 작업 스케줄러를 제안하였다. [53]은 사용자 대상 쿼리의 QoS를 달성하기 위해서 작업 성능 예측을 하며 이에 따라 자원의 경합을 인식하는 자원 할당을 한다. 또한, 작업의 진행을 계속적으로 모니터링 하여 예측된 값 보다 작업의 진행에 지연이 있다면 이를 보상하기 위해 할당 해주는 계산 자원의 양을 증시킨다. 이를 위해 오프라인 방법 뿐 아니라 온라인 방법론을 도입하여 런 타임 시스템을 구축한다. SM 부분 집합을 분할하는 다중 작업 실행 기법은 Intra-SM 자원을 공유하는 방법과 비교하여 자원 활용도 및 처리량 측면에서 성능 향상의 한계가 존재한다[43].

[17] 논문은 SM 내에서 리소스를 효율적으로 나눠 쓰는 법에 대해서 연구하였다. 킬로 워프 명령어 당 L2 캐시 미스 (L2 miss per kilo warp instruction)를 활용하여 응용을 분류 했으며, 워터-필링 (Water-filling) 알고리즘을 활용하여 두 응용이 자원을 공유할 때 가장 효율적인 분배 방법을 찾았다. [43] 은 SM 내 자원 공유를 위해 스레드 블록 단위로 컨텍스트를 전환하도록 하며, 자원을 공정하게 분할하기 위해서 정적 자원 사용량 및 동적 컴퓨팅 사이클을 사용한 스케줄링 알고리즘을 제안하였다. 하지만, 이의 컴퓨팅 사이클을 사용한 동적

자원 할당 방법은 실행 시간의 추정에 있어서 선형 증가를 가정하였다. 본 논문에서의 관찰을 통해, 자원 할당량에 따라 성능이 선형적으로 증가하지 않으며 성능이 포화하는 구간이 존재하므로 이를 인지하는 자원 할당 방법이 필요함을 시사한다. [17,43] 논문은 Intra-SM 자원을 분배하는 방법론에 대해서 소개하였으나, 함께 배치된 작업 간에 일어날 수 있는 간섭에 대해서 고려하지 않았으므로 자원의 경쟁으로 인하여 동시 수행 성능이 나빠질 수 있다. [42] 는 SM 내 자원 공유에 있어서 자원을 분배하는 방법론만으로는 동시 커널 간 간섭을 해결하지 못하기 때문에 이론적 성능과 실제 성능이 다를 수 있음을 지적하였다. 해당 논문에서는 간섭을 해결하기 위해서 균형 잡힌 메모리 요청 제출, 개별 커널에서 실행 하는 메모리 명령 제한을 제안하였다. 이는 메모리 요청 및 메모리 명령을 제어하기 위해서 추가적인 하드웨어 및 하드웨어 변경이 필요하므로 실제 하드웨어에서는 구현 불가능하지만, 본 논문의 방법은 실제 하드웨어에서 동작하면서 간섭을 최소화하고자 하였다.

[17,42-43,51-53] 모두 시뮬레이터에서 실험하여 실제 하드웨어 실험 결과와는 차이가 있을 수 있다. 또한, 선점을 사용한 동적인 SM 공유 방법이 제안되고 있으나 [41], 현재 하드웨어에서는 지원하지 않고 모두 시뮬레이션을 통해 그 효과를 입증하였다. [18] 논문은 SM 및 SM 내부 리소스를 공유하기 위해 스레드 블록 기반 스케줄링 프레임워크를 제안하였다. 해당 논문은 실제 하드웨어에서 Intra-SM 자원을 공유가능하도록 하여 SM 내부 자원의 활용률을 높였으나, 각 커널의 자원 경쟁을 인지하는 배치 방법에 대해서 고려하지 않았다.

VI. 결론

본 연구에서는 범용 목적 GPU 환경에서 응용 프로그램 특성 및 자원 사용 패턴을 반영한 다중 작업 배치 기법을 제안한다.

GPU 클러스터에서 각 응용 별 Intra-SM 자원 사용 특성이 다름을 분석하고 이로 인해 발생할 수 있는 간섭을 모델링 및 프로파일링을 통해 예측한다. 수집한 응용 프로그램의 프로파일링 정보를 통해 기계 학습 모델을 학습시키며, 이를 기반으로 한 간섭 인식 스케줄러인 Co-scheML을 제안하였다. 실험을 통해 Co-scheML은 GPU 클러스터 상에서 응용 간의 간섭을 최소화하고, 전체 클러스터의 활용도를 향상시키는 것을 보인다.

또한, 다양한 GPGPU 응용을 대상으로 Intra-SM 자원 사용량 및 런타임 수행 특성이 이질적이며, 자원 경쟁의 예측이 필요로 함을 보여준다. 응용의 개별 실행 특성에 따라 응용을 분류하며, 이를 바탕으로 동시 실행 특성을 관찰한다. 관찰을 기반으로 K-Scheduler를 소개하였다. 실험 결과는 기존의 Intra-SM 다중 작업 기법과 비교하여 K-Scheduler가 전체 워크로드의 성능을 향상시킬 뿐 아니라 개별 성능도 보존하는 것을 보여준다.

향후 연구로는 동적으로 도착하는 태스크에 대해서 K-Scheduler를 적용할 수 있도록 향상시킬 것이다. 또한, 두 개의 개별 기법을 하나의 일반적인 프레임워크로 통합하여 Inter-SM 및 Intra-SM 자원을 모두 활용함을 통해 GPU의 전체 자원의 활용도를 높이고자 한다.

참 고 문 헌

[1] J. Dean and L. A. Barroso, "The tail at scale," Communications of the ACM, vol. 56, no. 2, pp. 74–80, 2013.

[2]Amazon, "Amazon ec2 elastic gpus," 2017, https://aws.amazon.com/ec2/Elastic-GPUs/.

[3]Nimbi, https://www.nimbix.net/cloud-computing-nvidia/

[4]Peer1 hosting, http://www.peer1hosting.co.uk/hosting/gpu-servers

[5]Microsoft Azure, https://docs.microsoft.com/en-au/azure/virtual-machines/windows/sizes-gpu

[6]TOP 500, https://www.top500.org/

[7]Dongarra, Jack J., Piotr Luszczek, and Antoine Petitet. "The LINPACK benchmark: past, present and future." Concurrency and Computation: practice and experience 15.9 (2003): 803-820.

[8]Dongarra, Jack, and Michael A. Heroux. "Toward a new metric for ranking high performance computing systems." Sandia Report, SAND2013-4744 312 (2013): 150.

[9]Allen, Tyler, Xizhou Feng, and Rong Ge. "Slate: Enabling Workload-Aware Efficient Multiprocessing for Modern GPGPUs."

2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2019.

[10]Menezes, Evandro, et al. "CPU utilization measurement techniques for use in power management." U.S. Patent No. 6,845,456. 18 Jan. 2005.

[11]Sanbonmatsu, David M., et al. "Who multi-tasks and why? Multi-tasking ability, perceived multi-tasking ability, impulsivity, and sensation seeking." PloS one 8.1 (2013): e54402.

[12]Appelbaum, Steven H., Adam Marchionni, and Arturo Fernandez. "The multi-tasking paradox: Perceptions, problems and strategies." Management Decision (2008).

[13]Hyper-Q technology, https://forums.developer.nvidia.com/t/hyper-q-technology/34198

[14]Rogers, Phil, and A. Fellow. "Heterogeneous system architecture overview." Hot Chips Symposium. 2013.

[15]Schulte, Michael J., et al. "Achieving exascale capabilities through heterogeneous computing." IEEE Micro 35.4 (2015): 26-36.

[16]NVIDIA Multi-Process Service, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[17]Xu, Qiumin, et al. "Warped-slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming."

2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, 2016.

[18]QICHEN, CHEN. Optimizing GPU System for Efficient Resource Utilization of General Purpose GPU Applications in a Multitasking Environment. Diss. Seoul National University graduate school, 2020

[19]Openstack – support virtual GPU resources, https://specs.openstack.org/openstack/nova-specs/specs/queens/implemented/add-support-for-vgpu.html

[20]Yarn, https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/UsingGpus.html

[21]Kubernetes, https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

[22]Gupta, Vishakha, et al. "GViM: GPU-accelerated virtual machines." Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing. 2009.

[23]Shi, Lin, et al. "vCUDA: GPU-accelerated high-performance computing in virtual machines." IEEE Transactions on computers 61.6 (2011): 804-816.

[24]Gupta, Vishakha, et al. "Pegasus: Coordinated scheduling for virtualized accelerator-based systems." 2011 USENIX Annual Technical Conference (USENIX ATC'11). Vol. 31. 2011.

[25]Nvidia GPU Cloud(NGC),

https://ngc.nvidia.com/https://ngc.nvidia.com/

[26]Tensorflow CNN benchmarks, https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks

[27]DJINN, https://github.com/LLNL/DJINN

[28]NVIDIA profiler, https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[29]NVIDIA Nsight systems, https://developer.nvidia.com/nsight-systems

[30]Kubernetes, https://github.com/kubernetes/kubernetes

[31]NVML, https://developer.nvidia.com/nvidia-management-library-nvml

[32]InfuxDB, https://www.influxdata.com/

[33]Ukidave, Yash, Xiangyu Li, and David Kaeli. "Mystic: Predictive scheduling for gpu based cloud servers using machine learning." 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2016

[34]Chen Z, Quan W, Wen M, Fang J, Yu J, Zhang C, Luo L. Deep Learning Research and Development Platform: Characterizing and Scheduling with QoS Guarantees on GPU Clusters. IEEE Transactions on Parallel and Distributed Systems. 2019 Jul 29;31(1):34-50.

[35]NVIDIA CUDA Sample, https://docs.nvidia.com/cuda/cuda-samples/index.html

[36]Che, Shuai, et al. "Rodinia: A benchmark suite for heterogeneous computing." 2009 IEEE international symposium on workload characterization (IISWC). Ieee, 2009

[37]Stratton, John A., et al. "Parboil: A revised benchmark suite for scientific and commercial throughput computing." Center for Reliable and High-Performance Computing 127 (2012).

[38]Polyhedral Benchmark suite, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[39]Danalis, Anthony, et al. "The scalable heterogeneous computing (SHOC) benchmark suite." Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. 2010.

[40]NVCC, https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

[41]Park, Jason Jong Kyu, Yongjun Park, and Scott Mahlke. "Dynamic resource management for efficient utilization of multitasking gpus." Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. 2017.

[42]Dai, Hongwen, et al. "Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls." 2018 IEEE International Symposium on High Performance Computer

Architecture (HPCA). IEEE, 2018.

[43] Wang, Zhenning, et al. "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing." 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016.

[44] Diab, Khaled M., M. Mustafa Rafique, and Mohamed Hefeeda. "Dynamic sharing of GPUs in cloud systems." 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. IEEE, 2013.

[45] Gu, Jing, et al. "GaiaGPU: Sharing GPUs in Container Clouds." 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom). IEEE, 2018.

[46] Song, Shengbo, et al. "Gaia Scheduler: A Kubernetes-Based Scheduler Framework." 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom). IEEE, 2018.

[47] Thinakaran P, Gunasekaran JR, Sharma B, Kandemir MT, Das CR. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In2019 IEEE International

Conference on Cluster Computing (CLUSTER) 2019 Sep 23 (pp. 1-13). IEEE.

[48]Xu, Xin, et al. "Characterization and prediction of performance interference on mediated passthrough GPUs for interference-aware scheduler." 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19). 2019.

[49]Bao, Yixin, Yanghua Peng, and Chuan Wu. "Deep Learning-based Job Placement in Distributed Machine Learning Clusters." IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE, 2019.

[50]Wen Y, O'Boyle MF, Fensch C. MaxPair: enhance OpenCL concurrent kernel execution by weighted maximum matching. InProceedings of the 11th Workshop on General Purpose GPUs 2018 Feb 24 (pp. 40-49).

[51]Zhao, Xia, Zhiying Wang, and Lieven Eeckhout. "Classification-driven search for effective sm partitioning in multitasking gpus." Proceedings of the 2018 International Conference on Supercomputing. 2018.

[52]Zhao, Xia, Magnus Jahre, and Lieven Eeckhout. "HSM: A Hybrid Slowdown Model for Multitasking GPUs." Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020.

[53]Zhang, Wei, et al. "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators inZhang, Wei,

et al. "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters." Proceedings of the ACM International Conference on Supercomputing. 2019. datacenters." Proceedings of the ACM International Conference on Supercomputing. 2019.

ABSTRACT

Multi-Task Placement Techniques based on Profiling General Purpose GPUs

Sejin Kim
Department of Computer Science
The Graduate School
Sookmyung Women's University

Graphics Processing Units (GPU) are widely used not only in graphic processing but also in machine learning (ML) and high performance computing (HPC) for massive parallel computing power. Recently, data centers and cloud service providers have provided GPU-accelerated infrastructures. Actual GPGPU (General Purpose GPU) applications are not sufficiently utilized, unlike GPU-friendly applications. This led to multi-task placement of applications with different requirements to increase utilization for expensive GPUs. However, multi-tasking on GPUs may result in poor performance

because of contention for shared resources. In addition, as studies on GPU multi-tasking techniques are insufficient compared to CPU multi-tasking, multi-task placement techniques utilizing GPU resource usage patterns and application characteristics are required.

This study proposes multi-task placement techniques taking into account both shared resources of multiple SM level and internal SM resources level according to computing unit, SM. Multiple SM level introduces multi-task placement technique applying machine learning based on application characteristics and resource usage patterns. In addition, SM internal level proposes multi-task placement technique derived by internal resource usage patterns of applications. Experimental results demonstrate excellence of proposed techniques compared to other existing approaches. This shows that the techniques increase GPU resource utilization by minimizing contention of shared resources. As a result, the overall throughput of workloads is increased, while retaining individual performance.

Key words: GPU, Multi-Task Placement, Resource Management, Application Profiling