ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 48, No. 6, pp. 604-611, 2021. 6
https://doi.org/10.5626/JOK.2021.48.6.604

GPU의 효율적인 자원 활용을 위한 동시 멀티태스킹 성능 분석

(Performance Analysis of Concurrent Multitasking for Efficient Resource Utilization of GPUs)

김 세 진 [†] (Sejin Kim)　　진 계 신 [††] (Qichen Chen)　　염 헌 영 [†††] (HeonYoung Yeom)　　김 윤 희 [††††] (Yoonhee Kim)

요 약 계산 집약적인 응용을 가속화하기 위해 GPU(Graphics Processing Unit)가 널리 사용됨에 따라 데이터 센터 및 클라우드에서 GPU는 점점 더 많이 활용되고 있다. 여러 응용들의 동시 실행 요청이 있을 때 GPU 자원을 효율적으로 공유하도록 하는 연구는 아직 충분하지 않다. 또한, GPU 내의 자원을 효과적으로 공유하는 것은 응용의 자원 사용 패턴을 인지하지 않고서는 어렵다. 본 논문은 응용의 실행 패턴에 기반한 응용 분류법을 제시하고 자원 할당량 증가에도 성능이 향상되지 않는 이유를 런타임 특성에 따라 설명한다. 또한, 스레드 블록 기반 스케줄링 프레임워크인 smCompactor를 사용하여 분류된 응용을 기반으로 응용 조합의 동시 멀티태스킹 특성을 분석한다. 이를 통해 자원의 효율적인 활용이 가능한 응용의 조합을 파악한다. 응용 실행 특성을 고려하여 GPU상 멀티태스킹 실험을 진행한 결과, 기존 동시 실행 방법인 NVIDIA의 MPS와 비교하여 평균 28% 이상의 성능 향상을 보였다.
키워드: GPU, 멀티태스킹, 응용 분류, 스케줄링, smCompactor

Abstract As Graphics Processing Units (GPUs) are widely utilized to accelerate compute-intensive applications, their application has expanded especially in data centers and clouds. However, the existing resource sharing methods within GPU are limited and cannot efficiently handle several requests of concurrent cloud users' executions on GPU while effectively utilizing the available system resources. In addition, it is challenging to effectively partition resources within GPU without understanding and assimilating application execution patterns. This paper proposes an execution pattern-based application classification method and analyzes run-time characteristics: why the performance of an application is saturated at a point regardless of the allocated resources. In addition, we analyze the multitasking performance of the co-allocated applications using smCompactor, a thread block-based scheduling framework. We identify near-best co-allocated application sets, which effectively utilize the available system resources. Based on our results, there was a performance improvement of approximately 28% compared to NVIDIA MPS.
Keywords: GPU, multitasking, application classification, scheduling, smCompactor

· 본 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2015M3C4A7065646, 2021R1A2C1003379)
· 이 논문은 2020 한국소프트웨어종합학술대회에서 'GPU 상에서 응용의 실행 특성에 따른 멀티태스킹 성능 분석'의 제목으로 발표된 논문을 확장한 것임

† 비 회 원 : 숙명여자대학교 컴퓨터과학과 석사과정
wonder960702@gmail.com
†† 비 회 원 : 서울대학교 컴퓨터공학과 학생
charlie.cqc@gmail.com
††† 종신회원 : 서울대학교 컴퓨터공학과 교수
yeom@snu.ac.kr
†††† 종신회원 : 숙명여자대학교 소프트웨어학부 교수
(Sookmyung Women's Univ.)
yulan@sookmyung.ac.kr
(Corresponding author임)

논문접수 : 2021년 2월 16일
(Received 16 February 2021)
논문수정 : 2021년 3월 17일
(Revised 17 March 2021)
심사완료 : 2021년 3월 17일
(Accepted 17 March 2021)

Copyright©2021 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지 제48권 제6호(2021. 6)

1. 서 론

GPU는 계산 집약적인 응용을 가속화하기 위한 장치로 채택된다. 최근 많은 사람들이 주목하고 있는 고성능 컴퓨팅 응용이나 딥러닝과 같은 분야는 많은 계상량 필요로 해서 GPU 사용은 점점 더 늘어나는 추세이다. 이에 따라 클라우드 서비스 제공자들은 GPU 서비스를 제공하기 시작하였으며, 이를 GPU as a Service 라고 한다.

GPU의 내부는 다중의 스트리밍 멀티프로세서(Streaming Multiprocessor: SM)로 구성된다. 한편, GPU에서 실행되는 함수 단위인 커널은 여러 스레드가 모여 스레드 블록(Thread Block: TB)을 구성하고, TB는 각각 임의의 SM에 지정된다. 기술이 발전함에 따라 SM 내 자원의 수는 많아졌지만, Single Instruction Multiple Thread(SIMT) 모델을 채택하는 GPU 연산 특성상 각 응용은 모든 자원을 완전히 활용하지 않으므로 활용도가 낮은 자원들은 낭비될 수 있다는 문제가 발생하였다. 이를 해결하기 위해 여러 커널을 병렬로 실행하는 멀티태스킹이 등장하였다.

각 응용에 할당하는 자원의 양을 늘리면, 우리는 성능이 좋아지기를 기대한다. 이전 연구에서는 각 응용의 SM 수를 전체 SM 개수보다 적게 활성화했을 때 혹은 SM 내의 자원을 모두 사용하는 만큼의 활성 TB 수를 주기 전에 응용의 성능이 포화하는 것을 확인하였다[1-3]. 특히 [1]에 의하면 계산 집약 응용은 자원을 많이 줄수록 성능이 선형적으로 증가하며 메모리 집약 응용은 적은 자원을 줘도 최적에 가까운 성능을 나타냄을 관찰하였다. 하지만 실제 하드웨어 상에서 실험을 진행한 결과 TB 및 SM 두 가지 관점에서 하드웨어의 제한점 이전에 성능이 포화 되며, 각 응용 특성에 따라 포화되는 지점이 다른 것을 확인하였다.

본 논문의 목표는 SM 내의 자원을 효과적으로 분할하여 쓰기 위해 멀티태스킹 특성을 분석하는 것으로 한다. 우리는 각 응용 실행 중 발생할 수 있는 실행 지연의 이유에 따라 응용의 특성을 분류하고, 적격 워프의 수(Eligible warp per cycle: EPC)을 통해 성능이 포화되는 시점을 식별한다. 본 논문의 관찰 결과에 따르면 EPC가 작을수록 많은 자원을 할당해줘도 성능 향상이 없음을 확인하였다. 또한, 분류한 응용을 대상으로 동시 수행하여 요구 자원에 따른 동시 수행 패턴을 분석하였다. 이를 통해 더 많은 자원을 할당해도 성능 향상을 얻지 못하는 응용의 쌍과 GPU 자원을 모두 고갈시키지 않아 동시 수행의 이득을 얻을 기회가 있는 응용의 조합이 스레드 블록 기반 스케줄링 프레임워크인 smCompactor[3]를 사용하여 더 효과적인 동시 수행을 하는 것을 알 수 있다. 해당 분석은 효율적인 멀티태스킹 스케줄링에 좋은 가이드를 제공해 줄 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 연구배경을 설명하고, 3장에서는 연구의 동기를 소개한다. 4장에서는 응용을 실행시간 특성에 따라 분류하며, 5장에서는 응용 분류법에 따라 멀티태스킹 실험을 진행 및 분석한다.

2. 연구배경

NVIDIA에서는 Kepler 아키텍처부터 Hyper-Q 기술[4]을 적용하여 커널의 동시 실행이 가능하도록 하였다. CUDA 프로그램은 GPU 실행에 대한 컨텍스트를 캡슐화한 CUDA context를 생성하면서 시작한다. 해당 기술은 같은 CUDA context에 있는 커널들만을 동시 수행이 가능하도록 한다. 즉, 다른 응용에 속한 커널은 동시 수행이 불가능하다는 한계가 있으며, NVIDIA에서는 Multiple Process Service(MPS)[5]를 제공하여 여러 응용이 동시 수행할 수 있도록 하였다. 이때, 하나의 GPU 상에서 동시에 수행될 수 있는 커널의 개수는 레지스터의 크기, 공유 메모리(shared memory)의 크기 등 하드웨어의 제약에 따라 결정된다. MPS는 left-over 전략을 취하므로 먼저 수행을 시작한 커널이 자원을 모두 사용하지 않는 경우 다음 커널에게 남는 공간에 대해 자원을 할당한다[5]. 이로 인해 앞 커널이 자원을 많이 사용하고 시간이 오래 소요된다면, 뒤 커널이 블로킹(blocking) 될 수 있는 문제가 발생한다.

smCompactor[3]는 스레드 블록(Thread Block: TB) 기반 스케줄링 프레임워크이다. 본 논문에서는 해당 프레임워크를 사용하여 TB를 특정 SM에 지정하여 활성 SM 수 및 각 SM에서의 활성 TB 수를 조절하여 자원을 관리한다.

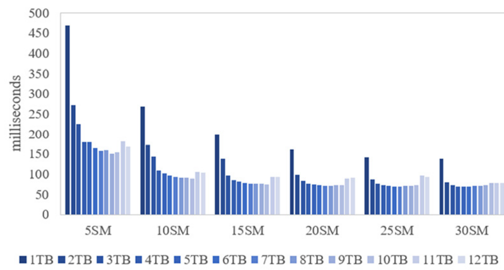
3. 연구동기

그림 1은 표 1의 응용 SPMV와 STENCIL을 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 TB (Thread Block) 수와, 활성 SM 수를 조절하였을 때 커널 수행시간이 달라짐을 보여준다.

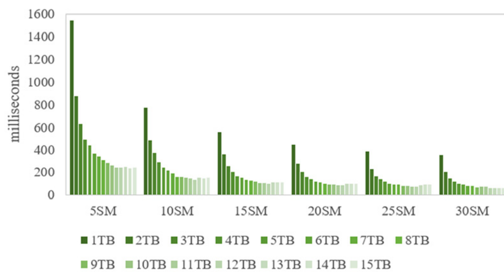
그림 1(a)는 자원 할당 양에 따른 SPMV의 커널 수행 시간을 보여주는 그래프이다. SPMV 30SM에 1TB를 주었을 때는 137ms, 30SM에 4TB를 주었을 때 약 68ms으로 시간이 감소하였다. 하지만, 12TB를 주었을 때는 77ms로 오히려 시간이 증가한 것을 확인할 수 있다. 해당 응용은 정적 프로파일링 정보인 레지스터를 기준으로 하여 한 SM에 12TB까지 할당해 줄 수 있으나, 12TB를 모두 할당해주지 않고 4TB만 할당해주어도 원하는 성능을 얻을 수 있다.

표 1 응용의 실행 지연의 이유와 사이클 당 적격 워프의 수 및 분류 결과
Table 1 Stall reason, eligible warp per cycle and classification result of applications

Application	exec_dep	inst_fetch	mem_dep	other	sync	tex	Eligible warps/cycle	Type
LavaMD (LM)	94.3%	0.53%	0.15%	4.54%	0.46%	0%	0.17	Compute
BlackScholes (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
FDTD3d (FT)	13.86%	3.69%	26.25%	38.6%	17.3%	0%	0.55	Compute
QuasiRandom Generator (QS)	22.43%	2.12%	0.87%	43.26%	0%	0%	10.06	Compute



(a) Kernel Execution Time of SPMV



(b) Kernel Execution Time of STENDIL

그림 1 SPMV 및 STENCIL의 자원 할당량에 따른 성능 변화
Fig. 1 The performance variation per resource allocation size of SPMV and STENCIL

반면 그림 1(b)의 STENCIL은 30SM에 1TB를 주었을 때는 348ms, 4TB를 주었을 때는 115ms, 12TB를 주었을 때는 59ms, SM 내 가용한 자원 양을 고려하여 최대의 TB를 스케줄링한 15TB를 주었을 때는 59ms가 소요되었다. STENCIL 또한 SPMV처럼 하드웨어 제약 이전에 성능이 포화 되지만 그 시점이 SPMV보다는 더 많은 TB를 할당해 줘야 함을 알 수 있다. 이를 통해 각 응용에 모든 자원을 할당해주지 않아도 최적에 가까운 성능을 보여주며, 응용마다 할당한 SM-TB 수에 따라 성능이 포화되는 지점이 다른 것을 보여준다.

4. 응용의 실행 시간 특성에 따른 응용 분류법

응용의 효율적인 멀티태스킹을 하기 위해서는 SM 내의 자원을 효과적으로 분배해야 한다. 시간이 지날수록 대상 응용의 수가 많아지고, GPU 내의 자원의 수가 증가하므로 이 모든 조합을 탐색하는 것은 어렵다. 그러므로 본 논문에서는 응용의 개별 실행 특성에 따라 동시 실행의 성능을 확인하고자 한다.

본 논문의 응용 분류는 각 응용 실행에 있어 발생할 수 있는 실행 지연의 이유를 사용한다. 각 SM에 할당된 TB를 활성 TB라고 하며, GPU 하드웨어 큐는 TB를 구성하는 스레드 중 32개의 스레드를 하나의 단위로 동시에 처리하는데, 이를 워프(warp)라고 한다. 즉, 활성 TB 수를 조절하는 것은 활성 워프 수를 조절하는 것과 같은 맥락이라고 할 수 있다. 하지만 모든 워프가 반드시 다음 명령을 발행할 수 있는 것은 아니다. 다른 워프가 도착할 때까지 배리어(barrier)에서 대기하거나 이전 명령의 결과를 기다리는 등의 이유로 명령을 발행하지 못한다면 이를 워프의 실행 지연이라고 한다. 반대로, 명령을 발행할 수 있다면, 이를 적격 워프(eligible warp)라고 한다. 즉, 활성 수는 실행 지연이 일어난 워프의 수와 적격 워프 수의 합이라고 할 수 있다. 실행 지연이 일어나는 주요 원인은 다음과 같다.

- 실행 의존성: 명령에 필요한 입력이 아직 준비가 되지 않아 일어나는 실행 지연을 의미한다. (exec_dep)
- 명령어 페치: 다음 어셈블리 명령이 페치되기를 기다리는 중을 의미한다. (inst_fetch)
- 메모리 의존성: 로드 및 저장과 관련된 자원이 사용 가능하지 않거나, 해당 자원이 완전히 사용 중일 때 발생하는 지연이다. (mem_de)
- 동기화: __syncthreads() CUDA API 호출로 인해서 워프가 블록 됨을 의미한다. (sync)
- 텍스쳐 캐시: L1 캐시가 완전히 사용 중이어서 워프

가 대기한다. (tex)

- 기타: 이 외의 다른 자원들에서 충돌되거나 의존성이 발생하는 것을 포함한다. (other)

각 실행 지연의 이유를 Nvprof[6]를 통하여 프로파일링한다. 각 사이클마다 워프에 실행 지연이 일어났다면, 각 이유에 해당하는 카운트를 증가시켜 백분율로 나타내며, 각 사이클마다 적격 워프의 개수를 프로파일링한다. 이 때 최소, 평균, 최대 값의 분산이 크지 않으므로 평균 값을 사용하였다. 각 응용별 실행 지연 이유의 백분율 및 사이클 별 적격 워프의 수(Eligible warps per cycle: EPC)는 표 1에서 보여준다.

본 논문은 메모리 의존성이 전체 지연의 백분율 중 S% 이상을 차지한다면 메모리 집약(Memory) 응용으로 분류한다. 메모리 집약 응용은 전역 메모리 영역으로 높은 처리량 및 활용도를 보인다. 이들은 메모리 요청이 빈번하게 일어나기 때문에, 워프가 다음 명령을 발행하지 못하는 메모리 의존성에 의한 실행 지연이 일어난다. 특히, GPU DRAM에 대한 메모리 요청은 400~600 사이클 소요되어 1 사이클 이내에 저장 및 로드가 가능한 레지스터 및 공유 메모리에 비해 지연을 더 많이 일으킬 수 있다. 텍스쳐 캐시로 인한 실행 지연이 백분율 중 S% 이상을 차지한다면 L1 캐시 집약(L1 cache) 응용으로 분류한다. 해당 응용은 L1 cache의 용량을 다 사용할 때까지는 자원 양을 늘리면 성능이 좋아지다가 캐시가 포화되는 시점에서 성능이 나빠지기 시작한다[2]. 이 외의 응용들은 계산 집약(Compute) 응용으로 분류한다. 이들은 정수 및 실수 연산과 같은 계산량이 많은 응용이다. 해당 응용들은 메모리 연산보다 계산 연산이 많기 때문에 실행 의존성, 명령어 페치, 동기화와 같은 실행 지연이 발생한다.

본 실험에서 S는 상수이며, 실험 환경에 따라 달라질 수 있다. 본 논문의 실험 환경에서는 휴리스틱하게 33%로 지정하였다. 분류한 결과는 표 1의 타입과 같다.

그림 1은 같은 메모리 집약 응용에 해당하는 SPMV와 STENCIL의 활성 SM 및 스레드 블록 수에 따른 성능 그래프이다. SPMV는 EPC가 0.72이기 때문에 더 많은 스레드블록을 활성화하여도 실행 지연으로 인하여 성능 향상의 이득을 볼 수 없다. 반면, STENCIL은 EPC가 5.68이기 때문에 활성 스레드블록 수가 많아져도 실행 지연의 영향을 받지 않는다.

계산 집약 응용에 대해서도 EPC의 크기에 따라 같은 특징을 보였다. LM의 경우 30SM-1TB의 자원을 할당해 주었을 때 수행 시간은 64ms이며, 30SM-2TB를 할당해 주었을 때는 수행 시간이 58ms로 성능 향상이 일어난다. 하지만 30SM-4TB를 할당하면 58ms로 더 많은 자원을 할당해줘도 성능 향상이 일어나지 않는 것을 확인할 수 있다. 반대로 5.67의 EPC를 가지는 CUTCP는 30개의 SM을 활성화 하였을 때, 1TB를 할당해주면 약 237ms, 2TB를 할당해주면 약 123.974ms, 4TB를 할당해주면 약 69ms로 거의 선형적으로 성능이 향상한다. 이를 통해 EPC가 큰 응용은 많은 자원을 할당해주면 성능 향상을 기대할 수 있음을 알 수 있다.

5. 실험 결과

본 장에서는 각 범주의 응용 간 페어링 실험을 통해 GPU 멀티태스킹 특성을 파악하고, 해당 쌍들의 성능을 MPS와 비교한다.

5.1 실험 환경

실제 GPU 시스템에서 실험을 진행하며 해당 시스템은 12GB의 GPU DRAM과 30개의 SM을 가진 NVIDIA Titan Xp GPU와 6개의 코어를 가진 Intel Core i7-5820K으로 구성되어 있다. Titan Xp는 각 SM 당 65536개의 레지스터와 64KB의 공유 메모리를 갖는다. 전체 시스템은 Ubuntu 16.04에서 수행된다. NVIDIA 드라이버 버전은 430.64이며, CUDA 버전은 10.0을 사용하였다. 실험에 사용된 응용들은 표 1과 같으며 모두 NVIDIA CUDA Sample[7]과 Rodinia GPU benchmark suite[8], Parboil benchmark[9]로부터 왔다. 모든 응용은 기본 데이터 셋을 사용하여 수행한다.

5.2 응용 분류에 따른 멀티태스킹 성능 분석

각 그룹의 응용 간에 페어링하고 해당 응용들을 동시 수행한 결과를 보여준다. 각 그래프에서 언더바("_") 앞의 응용은 첫 번째 응용 분류에 속하는 응용이며, 언더바 뒤의 응용은 두 번째 응용 분류에 속하는 응용이다. 해당 결과는 두 응용을 순차적으로 수행했을 때의 시간을 동시 수행시간으로 정규화한 성능 향상도(speedup)를 나타낸다. 즉 1보다 크다면 순차 실행과 비교하여 성능 이득이 있으며, 1보다 작거나 같으면 성능 이득이 없거나 오히려 성능이 나빠짐을 의미한다.

5.2.1 계산 집약 응용과 메모리 집약 응용의 멀티태스킹

계산 집약 응용과 메모리 집약 응용을 함께 수행한 결과는 그림 2에 보여진다. 계산 집약 응용과 메모리 집약 응용을 함께 수행했을 때의 수행시간은 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 의존성로 인하여 실행 지연이 많이 일어나는 응용이기 때문이다. EPC가 낮으므로 같은 응용 상에서는 stall이 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

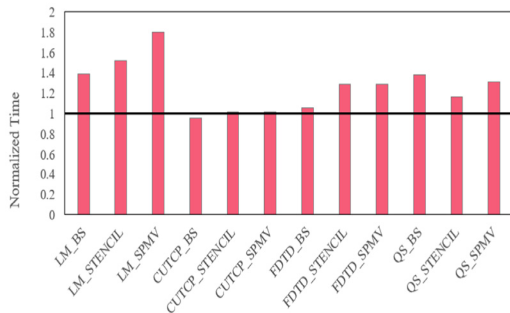


그림 2 계산 집약 응용과 메모리 집약 응용의 멀티태스킹 성능

Fig. 2 A performance comparison between the multitasking compute and memory-intensive applications

가장 작은 LM과 함께 수행한 쌍들은 모두 약 40% 이상의 성능향상을 얻을 수 있었다. 반대로, CUTCP는 가장 EPC가 높은 응용으로 자원을 많이 할당해줌으로써 활성 TB 수를 증가시키면 커널 수행시간이 감소했다. 하지만 멀티태스킹 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.

5.2.2 계산 집약 응용 간의 멀티태스킹

그림 3은 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. 5.2.1절의 실험과 같이 EPC가 가장 낮은 LM과 동시 수행한 경우 40% 이상의 성능 향상이 있었다. 특히, 2개의 LM 인스턴스를 수행하였을 때는 하나의 인스턴스에서 발생한 실행 의존성을 서로 감추기 때문에 성능이 거의 2배 좋아지는 것을 알 수 있다. 한편, EPC가 가장 높은 CUTCP는 EPC가 1보다 낮은 LM과 FDTD와 함께 수행한 경우에만 성능이 좋아졌으며 나머지 경우에는 순차 실행과 비슷한 혹은 나쁜 성능을 보여준다. 이는 동시 수행을 통해 감출 실행 지연이 많지 않기 때문이다.

5.2.3 메모리 집약 응용 간의 멀티태스킹

그림 4는 메모리 집약 응용끼리 동시 수행한 결과를 보여주는 그래프이다. 모든 쌍이 순차 실행과 동일한 혹은 나쁜 성능을 보인다. 특히 가장 EPC가 높은 쌍인 BS_STENCIL은 순차 실행과 비교하여 약 28% 안 좋은 성능을 보였으며, EPC가 가장 낮은 쌍인 SPMV_SPMV는 약 1%만 성능이 좋아졌다. 이를 통해 메모리 집약 응용 간 멀티태스킹 또한 EPC가 높은 응용의 쌍에 대해서 성능이 감소하는 것을 알 수 있다. 또한, 메모리 집약 응용끼리 쌍을 지으면 동시 수행의 이득을 기대할 수 없는데 이는 메모리 대역폭의 포화 때문이다. 예를 들어 STENCIL의 DRAM 처리량은 약 332GB/s, BS는 약 266GB/s이며, Titan Xp의 메모리 대역폭은 547GB/s까지 지원을 한다. 그러므로 하드웨어 메모리 대역폭 성능 제공량이 두 응용의 처리량을 모두 지원하지 못하기 때문에 이와 같은 결과를 보인다.

5.2.4 L1 캐시 집약 응용과 타 범주 응용 간의 멀티태스킹

그림 5와 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

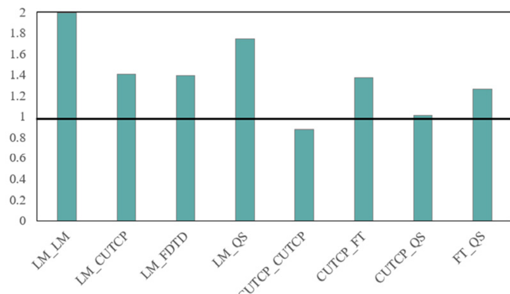


그림 3 계산 집약 응용 간의 멀티태스킹 성능

Fig. 3 A performance comparison between multitasking compute-intensive applications

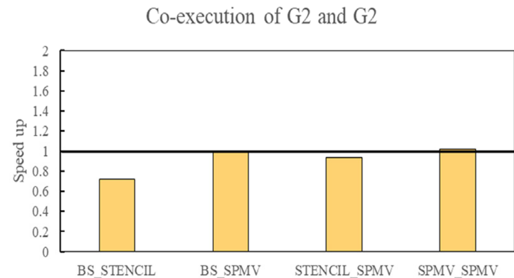


그림 4 메모리 집약 응용 간의 멀티태스킹 성능

Fig. 4 A performance comparison between multitasking memory-intensive applications

그림 5 L1 캐시 집약 응용과 이외의 응용의 멀티태스킹 성능
Fig. 5 A performance comparison between multitasking L1 cache-intensive and other applications

동시 수행의 이득을 기대할 수 있다.

5.2.5 MPS와의 비교

smCompactor를 사용하여 MPS와 비교한 결과는 그림 6과 같다. LM_SPMV 쌍은 각각 계산 집약 응용과 메모리 집약 응용 중 가장 EPC가 낮은 응용이다. 이는 다른 자원을 필요로 하는 응용과 동시 수행시켰을 때 성능 향상의 기회가 가장 큰 응용들이었다. 이 쌍들을 대상으로 smCompactor를 통해 SM 내 자원를 공유하도록 하였을 때, left-over 전략을 택하는 MPS를 사용하여 수행하였을 때보다 성능이 약 32% 좋은 것을 확인할 수 있다. 이는 MPS가 LM과 SPMV와 같이 자원을 많이 주어도 성능 향상이 없는 응용의 특성을 고려하지 않고, 정적인 자원 사용량 만을 고려하여 최대의 자원을 할당하여 동시 수행되기 때문이다. LM_CUTCP 쌍 에서 두 응용은 모두 계산 집약 응용에 속하는 응용으로써, 두 응용의 동시 실행은 메모리 대역폭를 포화시키지 않았으며 LM의 빈번한 실행 지연을 CUTCP가 감추어 순차 실행보다 커널 수행시간이 단축되었다. 특히 해당 응용을 MPS에서 수행하였을 때와 smCompactor에서 수행하였을 때를 비교하면, smCompactor의 성능이 약 28% 향상되었다. 이는 두 응용이 GPU 내부 자원을 모두 고갈시키지 않아 smCompactor를 사용한 효율적인 자원 공유가 가능하기 때문이다. 메모리 집약 응용끼리 함께 수행한 BS+SPMV 쌍은 MPS 보다 약 3%만 성능이 우수했다. smCompactor를 사용하여 스레드 블록 기반 스케줄링을 할 때 한 응용만으로도 메모리자원을 충분히 고갈시키기 때문에 자원을 분할하여 쓰는 것에 큰 이득을 얻지 못하였다. 이를 통해 동시 수행으로 인하여 GPU가 지원하는 메모리 대역폭을 초과하게 된다면, 아무리 자원을 효과적으로 나누어도 큰 효과가 없는 것을 알 수 있다.

본 실험의 결과를 통해서 LM과 같이 더 많은 자원을

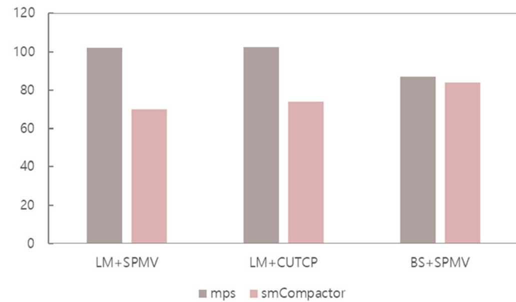


그림 6 smCompactor와 MPS 간의 비교
Fig. 6 A performance comparison between smCompactor and MPS

주어도 성능 향상을 얻지 못하는 응용의 쌍과 GPU 자원을 모두 고갈시키지 않아 동시 수행의 이득을 얻을 기회가 있는 응용의 쌍이 smCompactor가 MPS에 비해 더 효과적인 동시 수행을 하는 것을 알 수 있다. 또한, LM_SPMV과 LM_CUTCP 쌍과 같이 적절한 응용의 조합을 선택한다면 평균 28% 이상의 성능 향상이 있었다. 이는 응용의 특성을 고려하여 동시 실행할 응용을 선택하는 것이 성능을 크게 좌우하므로 성능고려 스케줄링 기법을 정교하고 일반화하여 제시하면 GPGPU용 Intra- SM 멀티태스킹 프레임워크인 smCompactor의 동시실행 성능을 극대화할 수 있음을 알 수 있다.

6. 관련 연구

GPU 멀티태스킹의 방법에는 두 가지가 있다. 타임 슬라이싱 방법으로 시간을 공유하는 방법(time sharing)이 있으며, SM 내부에서 혹은 SM의 부분 집합을 나누어 쓰는 공간 공유(spatial sharing) 방법이 있다.

시간 공유 방법으로 GPU를 공유할 때 발생하는 간섭을 고려한 스케줄링 연구들이 있다. Mystic[10]은 응용의 공동 실행을 위해 CF(Collaborate Filltering) 기반 간섭 인식 스케줄러이다. Bao 외[11] 논문은 GPU 서버가 있는 클러스터 환경에서 DRL model을 사용한 작업 배치 프레임워크를 제안한다. 김광복 외[12] 및 Huangfu 외[13] 논문은 워프 스케줄링 정책을 제안하였다. 시뮬레이터를 활용한 워프 스케줄링을 제안하였으나, 동시 수행에 관한 논의가 아니었다.

NVIDIA가 공간 공유를 지원하는 Hyper-Q 기술을 소개한 이후, 이를 활용하여 효과적으로 공간 공유하는 방법에 관한 연구가 계속되고 있다. Warped-slicer[2] 논문은 SM 내에서 자원을 효율적으로 나눠 쓰는 법에 관해서 연구하였다. 하지만 자원을 공유하는 두 응용의 특성에 따라 동시 실행 성능이 달라지는 것에는 초점을 두지 않았다. [14]는 자원을 공정하게 분할하기 위해서

정적 자원 사용량 및 동적 컴퓨팅 사이클을 사용한 스케줄링 알고리즘을 제안하였다. HSM[1] 논문은 메모리 집약 응용은 적은 SM 개수를 할당해줘도 비슷한 성능을 낼 수 있음에 착안하여 전력 모드를 제안하였다. 이는 SM 및 SM 내부 자원을 공유하는 본 논문과 다른 결과를 보여주며, HSM 및 Warped-slicer 모두 시뮬레이터에서 실험하여 실제 하드웨어 실험 결과와는 차이가 있을 수 있다.

7. 결 론

본 논문은 효율적인 Intra-SM 멀티태스킹을 위하여 응용 분류 및 분석하였다. 실행 지연의 이유를 사용하여 응용을 분류하고, 다른 범주에 속하는 응용 간의 멀티태스킹 및 EPC가 작은 계산 집약 응용 간의 멀티태스킹은 적은 간섭으로 동시 수행할 수 있음을 보이고 스케줄링에 대한 가이드를 제시하였다.

향후 연구로는 해당 분석을 사용하여 일반화를 시켜 스케줄링 알고리즘을 개발할 예정이다. 본 논문의 멀티태스킹 성능 분석은 각 응용의 특성에 기반한 응용 조합의 성능을 예측함에 있어 방향성을 제시할 것이다. 또한, 이를 TB 기반 프레임워크와 통합하여 다양한 응용을 대상으로 실험을 진행할 예정이다.

References

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

[2] 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.

[3] Qichen Chen, et al., "smCompactor: A Workload-aware Fine-grained Resource Management Framework for GPGPUs," *Proc. of the 35th Annual ACM Symposium on Applied Computing*, Mar. 2021.

[4] NVIDIA Hyper-Q technology, http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

[5] NVIDIA Multi Process Service (MPS), https://docs.nvidia.com/deploy/mps/index.html

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

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

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

[9] 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).

[10] 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.

[11] 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.

[12] G. B. Kim, J. M. Kim, and C. H. Kim, "Latency Hiding based Warp Scheduling Policy for High Performance GPUs," *Journal of the Korea Society of Computer and Information*, vol. 24, no. 4, pp. 1-9, Apr. 2019.

[13] Yijie Huangfu, Wei Zhang. (2017). Warp-Based Load/Store Reordering to Improve GPU Time Predictability, *Journal of Computing Science and Engineering*, 11(2), 58-68.

[14] 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.

김 세 진

2019년 숙명여자대학교 소프트웨어학부 졸업(학사). 2021년 숙명여자대학교 컴퓨터과학과 졸업(석사). 관심분야는 고성능 컴퓨팅, 클라우드 컴퓨팅, 데이터 사이언스

진 계 신

2011년 NanJing XiaoZhuang University 컴퓨터과학 및 정보 졸업(학사). 2020년 서울대학교 컴퓨터공학과 졸업(박사). 관심분야는 분산 시스템, 데이터베이스 시스템, 운영체제

염 헌 영

1984년 서울대학교 계산통계학과(학사) 1986년 Texas A&M 컴퓨터공학과(석사) 1992년 Texas A&M 컴퓨터공학과(박사) 1986년~1990년 Texas Transportation institute 시스템 분석가. 1992년~1993년 삼성 데이터 시스템. 1993년~현재 서울대학교 교수. 관심분야는 분산시스템, 스토리지

김 윤 희

1991년 숙명여자대학교 전산학과 졸업(학사). 1996년 Syracuse University 전산학과 졸업(석사). 2000년 Syracuse University 전산학과 졸업(박사). 1991년~1994년 한국전자통신연구원 연구원 2000년~2001년 Rochester Institute of Technology 컴퓨터공학과 조교수. 2001년~2016년 숙명여자대학교 컴퓨터과학부 교수. 2017년~현재 숙명여자대학교 소프트웨어학부 교수. 관심분야는 클라우드 컴퓨팅, 워크플로우 제어, 그리드/클라우드 관리