GPU 상에서 응용의 실행 특성에 따른 멀티태스킹 성능 분석

김세진O[1] 진계신[2] 염헌영[2] 김윤희[1 1)]
[1]숙명여자대학교 컴퓨터과학과
[2]서울대학교 컴퓨터공학부
wonder960702@gmail.com, charlie.cqc@gmail.com, yeom@snu.ac.kr, yulan@sookmyung.ac.kr

영문제목

Sejin KimO[1] Qichen Chen[2] HeonYoung Yeom[2] Yoonhee Kim[1 1)]
[1]Dept. of Computer Science, Sookmyung Women's University
[2]School of Computer Science and Engineering, Seoul National University

요 약

계산 집약적인 응용을 가속화하기 위해 GPU가 널리 사용됨에 따라 데이터 센터 및 클라우드에서 GPU는 점점 더 많이 채택되고 있다. 여러 응용 프로그램 간에 제한된 GPU 리소스를 공유하는 것은 중요하다. 하지만 GPU 내의 자원을 효과적으로 분할하는 것은 응용의 자원 사용 패턴을 인지하지 않고서는 어렵다. 본 논문은 스레드 블록 기반 스케줄링 프레임워크인 smCompactor를 사용하여 효과적인 동시 수행을 위하여 응용의 실행 패턴 분류한다. 또한, 이를 기반으로 하여 자원의 최대 활용이 가능한 실험을 보여줌으로써 GPGPU 멀티태스킹 스케줄링의 가능성을 제시하는 것을 목표로 한다. 이를 위하여 응용의 실행 특성을 분류하며, 분류된 응용을 토대로 응용 조합의 멀티태스킹 성능을 분석한다.

1. 서 론

GPU는 계산 집약적인 응용을 가속화 하기 위한 장치로 채택되어 왔다. 최근 많은 사람들이 주목하고 있는 고성능 컴퓨팅 응용이나 딥러닝과 같은 분야는 계산량을 많이 필요로 해서 GPU 사용은 점점 더 늘어나고 있다. 이에 따라 클라우드 서비스 제공자들은 GPU 서비스를 제공하기 시작하였으며, 이를 GPU as a Service 라고 한다. 클라우드 서비스 제공자들은 여러 클라이언트의 응용을 효율적으로 실행해야 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 임 의의 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.

본 논문의 목표는 GPU상에서 응용의 실행 시간 특성에 따라 달라지는 멀티태스킹 성능을 분석하는 것으로 한다. GPU 멀티태스킹의 수행 성능은 함께 수행하는 응용의 조합에 따라 달라진다. GPU 자원을 효율적으로 활용하기 위해 동시 수행으로 인하여 성능이 향상되는 조합을 찾는 것은 중요하다. 하지만, 멀티태스킹에 있어 대상 응용프로그램의 수가 증가할수록 가능한 조합의 수가 많아지기 때문에 이들의 조합을 모두 탐색하는 것은 어려운 일이다. 모든 조합을 실행해보지 않고, 동시 수행 성능을 도출하기 위해 각 응용의 개별 실행 특성을 활용하고자 한다. 이를 위해 우리는 각 응용 실행에 있어 발생하는 실행 지연의 이유에 따라 응용의 특성을 분류한다. 또한, 분류한 응용을 대상으로 서로 동시 수행하여 요구 자원에 따른 동시 수행 패턴을 분석하였다. 실험 결과를 통해 다른 범주에 속하는 응용의 조합은 동시 실행의 성능 이득이 크며, 같은 계산 집약 응용이라도 적격 워프의 수가 작다면 동시 수행의 이득을 얻을 수 있는 것을 보여준다. 해당 분석은 효율적인 멀티태스킹 스케줄링에 좋은 가이드를 제공해 줄 수 있다.

2. 배 경

GPU를 하나의 응용이 독점적으로 쓰는 것은 다른 응용으로부터 영향을 받지 않기 때문에 격리(isolation)와 단일 응용의 성능 측면에서 이득을 볼 수 있다. 하지만 GPU 내 자원 양이 증가함에 따라 자원의 낭비가 발생하기 시작했다. 그래서 NVIDIA에서는 Kepler 아키텍처부터 Hyper-Q 기술 [1] 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) [2] 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[2]. 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. 이는 먼저 커널의 정적 및 동적 기능을 프로파일링한 후 해당 정보에 따라 TB를 특정 SM에 매핑하

1) 교신저자

* 본 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2015M3C4A7065646).

표 1 사용된 GPGPU 응용 및 분류 결과

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%	13.53%	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%	15.43%	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%	10.96	Compute

는 디스패치 정보를 결정한다. 이 디스패치 정보는 영구 스레드 모델(persistent thread model)에 따라 커널을 변환하는 커널 변환 모듈에 통합되어 각 TB를 원하는 SM에 제출할 수 있도록 하여 SM 내부 자원을 공유하는 응용의 intra-SM 멀티태스킹을 가능하게 한다. 본 논문에서는 해당 프레임워크를 사용하여 TB를 특정 SM에 지정하여 활성 SM 수 및 각 SM에서의 활성 TB 수를 조절하여 자원을 관리한다.

3. 응용의 실행 시간 특성에 따른 분류

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. (stall_exec_dep)
- 명령어 페치: 다음 어셈블리 명령이 페치되기를 기다리는 중을 의미한다. (stall_inst_fetch)
- 메모리 의존성: 로드 및 저장과 관련된 자원이 사용 가능하지 않거나, 해당 자원이 완전히 사용 중일 때 발생하는 지연이다. (stall_mem_de)
- 동기화: __syncthreads() CUDA API 호출로 인해서 워프가 블록 됨을 의미한다. (stall_sync)
- 텍스쳐 캐시: L1 캐시가 완전히 사용 중이어서 워프가 대기한다. (stall_tex)
- 기타: 이 외의 다른 자원들에서 충돌되거나 의존성이 발생하는 것을 포함한다. (stall_other)

각 실행 지연의 이유를 Nvprof[7]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.

그림 1 계산 집약 응용과 메모리 집약 응용의 멀티태스킹

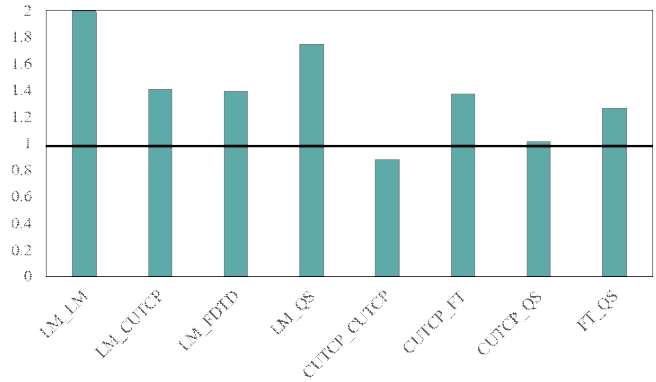

그림 2 계산 집약 응용 간의 멀티태스킹

본 논문은 메모리 의존성이 전체 지연의 백분율 중 1/3 이상을 차지한다면 메모리 집약(Memory) 응용으로 분류한다. 텍스쳐 캐시로 인한 실행 지연이 백분율 중 1/3 이상을 차지한다면 L1 캐시 집약(L1 cache) 응용으로 분류하며, 이 외의 응용들은 계산 집약(Compute) 응용으로 분류한다. 분류한 결과는 표 1의 타입과 같다.

4. 실험 결과

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

4.1. 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[4] 과 Rodinia GPU benchmark suite[5] 로부터 왔다. 모든 응용은 기본 입력 데이터 셋을 사용하여 수행한다.

4.2. 응용 분류에 따른 멀티태스킹 성능 분석

각 그룹의 응용 간에 페어링하고 해당 응용들을 동시 수행한 결과를 보여준다. 각 그래프에서 언더바(“_”) 앞의 응용은 첫 번째 그룹에 속하는 응용이며, 언더바 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.

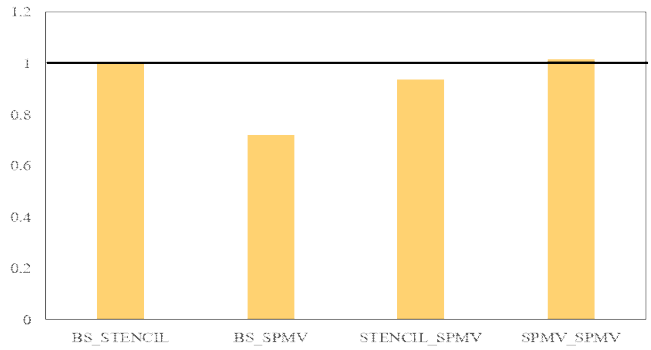

그림 3 메모리 집약 응용 간의 멀티태스킹

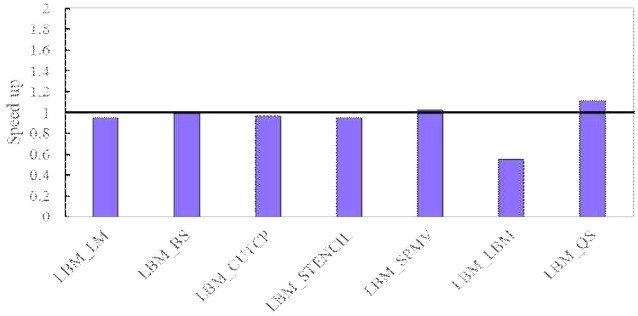

그림 4 L1 캐시 집약 응용과 타 범주 응용 간의 멀티태스킹

4.2.1 계산 집약 응용과 메모리 집약 응용의 멀티태스킹

계산 집약 응용과 메모리 집약 응용을 함께 수행한 결과는 그림 1에 보여진다.

계산 집약 응용과 메모리 집약 응용을 함께 수행했을 때의 수행시간은 순차 실행 수행 시간보다 평균 약 27% 단축되었다. 가장 성능이 좋아진 쌍은는 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 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 좋아졌다. 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.

4.2.2 계산 집약 응용 간의 멀티태스킹

그림 2는 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 성능향상이 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 때문임을 알 수 있다.

4.2.3 메모리 집약 응용 간의 멀티태스킹

그림 3은 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/s 이며, Titan Xp의 메모리 대역폭은 547GB/s 까지 지원을 한다. 그러므로 하드웨어 메모리 대역폭 성능 제공량이 두 응용의 처리량을 모두 지원 하지 못하기 때문에 이와 같은 결과를 보인다.

4.2.4 L1 캐시 집약 응용과 타 범주 응용 간의 멀티태스킹

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 있다.

5. 결 론

본 논문은 응용의 멀티태스킹에 있어 자원을 효과적으로 공유하기 위해 멀티태스킹의 성능을 분석하는 것을 목표로 한다. 응용의 개별 실행 특성으로 응용을 분류하고, 실험을 통해 분류한 응용을 대상으로 동시 실행의 성능을 분석하였다. 우리의 실험 결과는 서로 다른 범주에 속한 응용의 조합이며 EPC가 낮다면 자원의 경쟁이 일어나지 않고 동시 수행할 수 있음을 보였다. 본 논문의 응용 분류 및 분석은 효율적인 멀티태스킹을 하기 위한 스케줄링에 활용될 수 있을 것이다.

향후 연구로써는 해당 분석의 일반화를 통해 스케줄링 알고리즘을 개발하며 이를 TB 기반 프레임워크와 통합하여 다양한 응용을 대상으로 실험을 진행할 예정이다.

참고문헌

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

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

[3] Qichen Chen, et al. “smCompactor: A Workload-aware Fine-grained Resource Management Framework for GPGPUs” Proceedings of the 35th Annual ACM Symposium on Applied Computing. 2021. (accepted)

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

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

[6] 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).

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