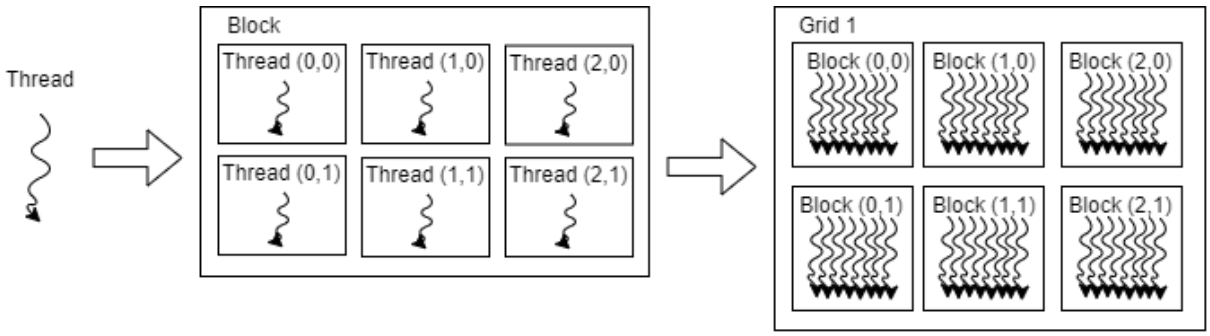
1. 요약

최근 기계학습의 대중화로, GPGPU(General-Purpose computing on GPU)를 많이 활용한다. 이는 컴퓨터 그래픽스만을 위한 계산을 한 GPU가, CPU의 일반적인 연산에 적용할 수 있는 기술이다. 또한 학습 데이터의 크기가 커짐에 따라 GPU 스케쥴링을 통해 최대 효율을 내는 연구들이 있었으나 간섭 요인으로 인한 GPU 성능 저하는 고려되지 못했다. 본 실험은 GPU 성능 저하를 실제로 확인하고, 가능하다면 원인까지 분석을 하도록 한다.

2. 서론

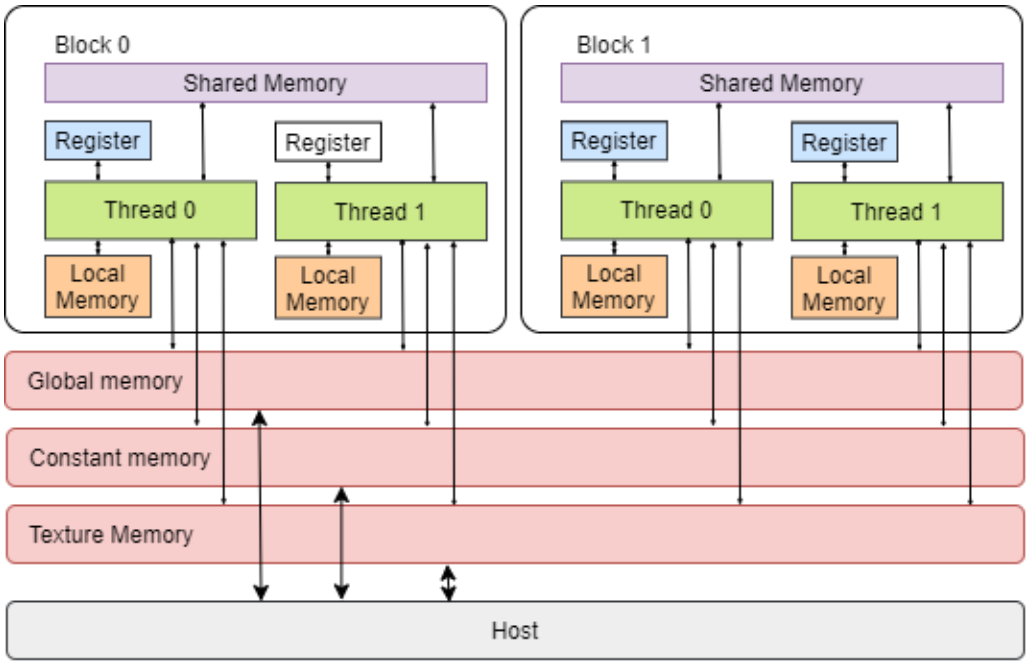
2.1 제안배경 및 필요성

기본적으로 GPU의 구조는 수많은 멀티 코어를 갖고 있어 병렬처리에 능해서 연산 처리 속도를 높일 수가 있다. CPU의 코어 개수가 16개 정도에 미친다면, GPU(RTX 3090)의 코어 개수는 10000개에 육박한다. 여기서 수행하는 병렬 처리 알고리즘을 코드로 짤 수 있게 해주는 GPGPU 기술인 CUDA(Compute Unified Device Architecture) 가 있다[1]. CUDA에서는 그리드(grid), 블록(block), 쓰레드(thread)를 이용하여 처리 영역을 구분한다. 쓰레드는 최소 명령어 처리 단위이고, 블록은 쓰레드들의 묶음이다. 그리고 그리드는 블록들이 모여서 생긴다. 여기서 병렬 실행되는 명령의 모음을 kernel이라고 한다. 쓰레드들을 묶어서 스케쥴링하는 단위를 warp라고 한다.



CUDA Memory는 Register, Local Memory, Global Memory, Shared Memory, Constant memory, Texture Memory를 제공한다. Global Memory는 그래픽카드에 장착된 DRAM을 의미한다. 접근 속도가 가장 느리다. Register는 로컬변수로, CPU의 레지스터와 비슷하게 접근 속도가 가장 빠른 메모리이다. 그런데 기본적으로 CPU에서 여러 개의 task를 실행할 때(A, B), A task가 진행되고 있는 상태에서, 다음 우선 순위의 B task가 실행되어야 할 때, A task의 상태 및 정보(context)는 register에 저장하고, B task의 context로 바꾸는 행위를 context switching이라고 한다. context switching이 잦아지게 되면, overhead가 발생하게 되어서 성능이 저하된다. 그러나 GPU는 각 쓰레드마다 다량의 register를 지니고 있기 때문에 context switching을 할 때, overhead가 적다는 장점이 있다. Local memory는 각 쓰레드들이 사용하는 register의 양이 많으면 사용한다. register의 사용량이 많아지면, 값들을 local memory에 저장하고 해당하는 register들을 다르게 이용한다. 이 local memory를 많이 사용하게 되면, GPU의 성능이 저하될 수 있다. 그래서 CUDA에서 thread당

register 사용량을 정할 수 있는데, 만약 register 사용량을 적게 세팅한다면 local memory 사용량이 늘어나고, 이는 성능 저하로 이어질 수 있을 것이다. Shared Memory는 블록의 쓰레드들끼리만 공유하는 메모리이다. 즉, A 블록의 쓰레드들은 B 블록의 shared memory에 접근할 수 없다. 이는 크기가 작고, 접근 속도가 빠르다. Constant Memory는 하드웨어에 구현된 크기가 작은 읽기 전용 메모리이다. 이는 캐싱을 지원해서 최초의 메모리는 Global memory에서 가져오기 때문에 속도가 느리지만, 이후에는 캐싱된 값을 이용하기 때문에 접근 속도가 빠르다. 만약 쓰레드들이 같은 주소값을 호출한다면, GPU는 Constant memory로 읽기를 한 번만 하고 값을 뿌려주는 방식으로 성능 향상이 생길 수 있을 것이다. Texture memory는 constant memory와 비슷하게 읽기 전용 메모리이며, 캐싱을 지원한다. Global memory 위에 구현되지만 캐싱이 되기 때문에, 특정 경우 성능향상을 이끌어 낼 수 있다. 또한 이는 공간지역성을 갖는 그래픽 작업에 많이 쓰인다.



2.2 목표

최근 클라우드에서 컴퓨팅을 구축하는 메커니즘으로 컨테이너의 인기가 증가하고 있다. 일관되고 신뢰성 있는 성능 제공을 위해 사용자에게 할당한 자원의 독립성에 대한 연구가 이루어지고 있다. 그런데 최근 한 연구에서 컨테이너 자원 사용량에 포함되지 않는 커널 계층의 연산으로 자원 독립성이 훼손되는 경우가 있었다. [2] 두 컨테이너가 있을 때, 네트워크 스택과 관련된 연산 오버헤드가 한쪽에서만 발생하는 경우, 각 컨테이너는 이 오버헤드를 제외한 절반의 자원을 받기 때문에, 컨테이너의 자원 독립성을 무너뜨린다. 이는 네트워크 스택에서 발생하는 상황인데, GPU 컨테이너 역시 비슷한 issue가 발생할 가능성이 있다. 예를 들면, 여러 개의 kernel이 메모리를 참조해야 하는 상황에서, context switching으로 인한 cache를 오염시키는 경우가 생길 수 있다. cache가 오염되면 cache hit 횟수가 줄어들어서 local memory에 접근을 자주하게 되고, 각 이용자

들의 성능이 감소할 수도 있다. 이 현상을 확인해보고자 한다.

또한 gpgpu를 이용한 연구에서 persistent kernel의 활용이 활발하다. [3] persistent kernel이 실행되면 GPU의 연산 처리 단위인 SM에서 hyper threading이 불가능해지기 때문에 다른 kernel의 성능저하에 더 큰 영향을 끼칠 수 있는 것으로 의심된다. 다수의 컨테이너 중 하나의 컨테이너가 persistent kernel을 실행시키는 상황을 가정하여 각각의 kernel들을 persistent kernel과 함께 실행시켰을 때와 일반 kernel과 동시 실행시켰을 때 성능 저하의 차이에 대해서 비교해보려고 한다.

실험할 때 유의할 점은, GPU는 동일 명령어를 여러 개의 쓰레드가 동시에 실행하는 SIMT (single instruction multiple threads) 모델을 사용하는데, 이는 하나의 instruction만 실행가능한 단점이 있다. 쓰레드의 일부가 분기를 하거나 루프를 하면 다른 쓰레드들은 대기를 하는데, 병렬처리의 장점이 퇴색된다. 그렇기 때문에, code optimization이 필요하다. 혹은 비교군이 동등한 optimization level에서 진행해야 한다.

이러한 성능 저하를 확인하고, 다른 연구에서 이를 해결하면 병렬 처리 및 다중 kernel 실행에 있어서 GPU의 최대 성능을 이끌어 내고, 더 나아가 다중 GPU에서의 성능 저하현상 그리고 또 다른 연구에서의 도움이 될 수 있을 것이다.

2.3 실험 구성 및 결과 요약

memory access가 큰 matrix multiplication 과 memory access는 적지만 연산을 많이 하는 code를 이용해 각 kernel들의 성능저하를 확인하고 원인을 분석한다. 그 결과 성능 저하의 원인은 memory bandwidth의 bottleneck 현상 및 낮은 cache hit ratio가 영향을 끼친다고 추정할 수 있었다.

실행시킨 persistent kernel의 개수를 달리하며 kernel의 성능을 구해본 결과, persistent kernel과 일반 kernel이 다른 kernel에 대해 거의 동일한 성능 저하를 갖게 하였다.

3. 관련 연구 및 기술현황

3.1 클라우드상 GPU 활용 관련 시장 현황

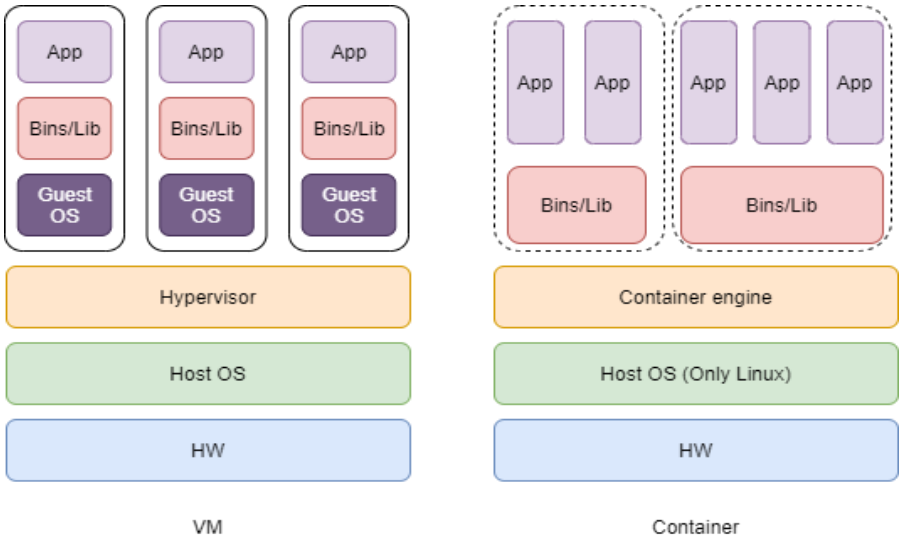
최근 클라우드에서 GPU가 많이 활용되고 있다. 기계학습과 같은 연산들을 빠르게 실행 가능한 GPU 자원을 서버에서 제공해주는 형태이다. 실제로 NVIDIA 수익의 40%가 데이터 센터의 수입이고, Alibaba, AWS, Baidu, Google, IBM, Microsoft, Oracle, Tencent 등 많은 대기업들 역시 클라우드 서비스를 제공하고 있다.[4][5]

클라우드 컴퓨팅은 사용자가 자원을 직접 구성할 필요 없이, 시스템 자원을 필요시 서버를 통해 바로 제공 가능한 것을 말한다. 이를 위해서 가상화가 필요한데, 가상화는 독립적인 코어와 메

모리를 갖는 여러 대의 가상머신들이 물리적 자원을 할당 받아 사용가능케 하는 것이다.

3.2 클라우드에서의 가상화 기술 활용

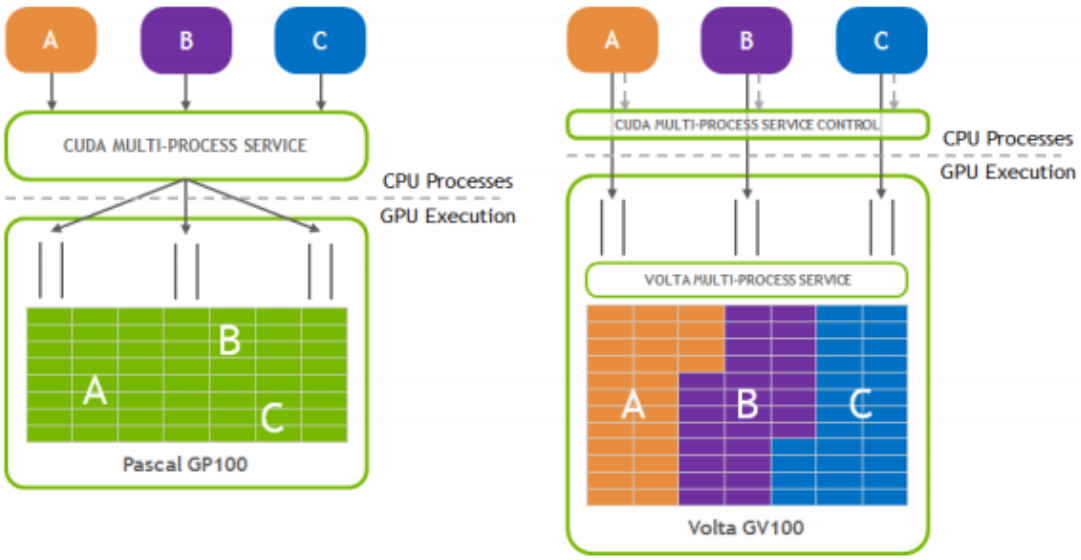
가상화는 단일 물리 하드웨어 시스템에서 여러 개의 환경이나 자원을 생성할 수 있는 기술이다. VM(Virtual Machine)은 컴퓨팅 환경을 가상으로 구현한 것인데, 구조가 물리적 컴퓨팅 환경과 비슷하다. VM은 여러 개가 동시에 작동할 수 있고, 서로 다른 환경을 가질 수 있는데, 이를 관리하는 관리자를 하이퍼바이저(Hypervisor)라고 한다. 이 하이퍼바이저가 VM들에게 자원을 분배하고 하드웨어와 VM간의 I/O를 담당한다. 컨테이너(container)는 분리된 독립적 컴퓨팅 환경을 말한다. 기존의 방식은 각 VM에 게스트 OS가 존재했으나, 컨테이너에서는 각각의 OS가 필요 없고, 호스트 OS를 공유하는 방식으로 실행된다. VM은 서버를 여러 개를 만들 수 있게 해주었다면, 컨테이너는 하나의 서버에 여러 개의 어플리케이션을 이용할 수 있게 해주었다. 즉, VM은 하드웨어 수준의 가상화를 진행하였고, 컨테이너는 소프트웨어 수준의 가상화를 한 것이다. 그래서 컨테이너는 자원 사용량이 적고, 성능이 빠르며 상대적으로 관리가 편하다.



3.3 NVIDIA의 멀티 프로세싱을 위한 기술 MPS와 MIG

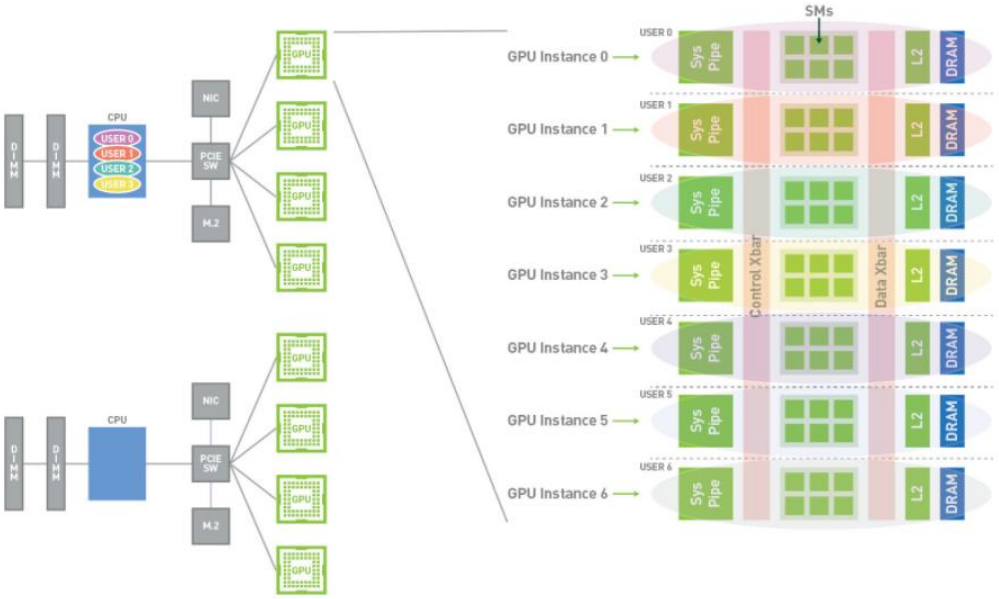
컨테이너들이 동시에 GPU에 올라갈 수 있도록 MPS(Multi Process Service) 기술이 있다[5]. MPS는 한개의 GPU 위에 다수의 프로세스가 동시에 작동 가능하도록 하는 서비스이다. 기존에는 여러 프로세스가 GPU에서 메모리 접근을 동일한 context에서 해야 했다. 여기서 돌아가면서 메모리를 접근하는데, 관리해주는 존재가 없어서 서로 간의 메모리가 침범 가능했다. 그래서 성능이 저하되는 모습이 보였는데, Volta MPS는 여러 프로세스가 GPU에 접근 시 각각 자원에 직접 접근 후에 사용을 한다. 그렇기 때문에 이들 간의 메모리 구분을 위해 virtual memory address space를 구분해주었다. 그래서 각 프로세스가 메모리 침범 없이 동시에 실행이 가능했고, 이를 관리하는

MPS서버가 존재한다. 그래서 GPU context switching 오버헤드가 줄어들었다.



이 기술의 한계는 MPS에서 각 프로세스들은 결국 동일 GPU의 virtual memory address space를 이용하는 것이기 때문에, out-of-range write, 혹은 out-of-range read로 다른 프로세스에게 비정상적으로 쓰기를 당하거나, 읽기를 당할 수 있다. 이 경우에 에러가 발생하는지 알 수 없다. 또한 MPS는 스케쥴링과 에러 보고 리소스(error reporting resource)도 공유한다. 그래서 특정 클라이언트가 예외사항이 생기면 다른 모든 클라이언트들에게 보고되지만, 어떤 클라이언트가 오류를 발생시켰는지 알 수 없다. 또한 이 오류를 일으킨 프로세스가 종료된다면, 다른 프로세스들도 종료될 수 있다.

그래서 NVIDIA에서는 MIG(Multi-Instance GPU)를 도입했다[7]. 이는 A100 GPU를 최대 7개의 독립된 GPU 인스턴스로 분할할 수 있다. MIG를 사용하면 각 인스턴스의 프로세서가 독립된 메모리 시스템을 갖게 된다. 즉, 캐시나 메모리 컨트롤러, DRAM 주소 버스는 모두 개별 인스턴스에 고유하게 할당된다.



이로 인해 각각의 사용자가 위에서 언급한 캐시 오염 등과 같은 상황에 노출되지 않고, 다른 사용자와 상관없이 최적의 자원을 할당 받을 수 있다. 여기에 VM, 컨테이너 등 자원의 독립성이 보장되어야 하는 서비스가 제공될 수 있다. 이는 A100 GPU에서만 지원되며, Linux에서만 지원된다.

비슷하게, GPU 위에 가상 클러스터를 올릴 때, 사용하지 않는 메모리는 binding을 해제하고 사용하는 메모리를 binding하는, 동적 binding을 이용해 독립적인 메모리를 할당하고 간섭으로 인한 성능저하를 줄인 연구가 있다. [8]

4. 실험 과정

4.1 요약

첫 번째 실험으로, memory access가 큰 matrix multiplication 과 memory access는 적지만 연산을 많이 하는 code를 이용해 각 kernel들의 성능저하를 확인하고 원인을 분석한다.

두 번째 실험으로, Persistent kernel의 개수를 증가해가며 각각의 kernel에 어느 정도의 성능저하를 끼치는지 확인해본다.

4.2 실험 환경

OS: Ubuntu 20.04.2 LTS

CPU: Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz

GPU: Quadro P4000

CUDA Version: 10.1

NVIDIA-DRIVER Version: 418.67

4.3 구현 및 설명

제안서에 따르면 cache가 오염되어 hit ratio가 감소하여 성능이 저하되는 현상을 확인하고, memory bandwidth 공유로 인한 성능 저하 현상을 확인해보려고 했다. 그러기 위해서는 memory access가 적고, core연산이 많은 code와, memory access가 많은 code를 준비해서 동시에 kernel을 가동시켜야 한다. 그래서 memory access가 많은 matrix multiplication code와, memory access가 적고 core 연산이 많은 작은 크기의 array내에서 사칙연산만 반복하는 code를 이용해 실험했다.

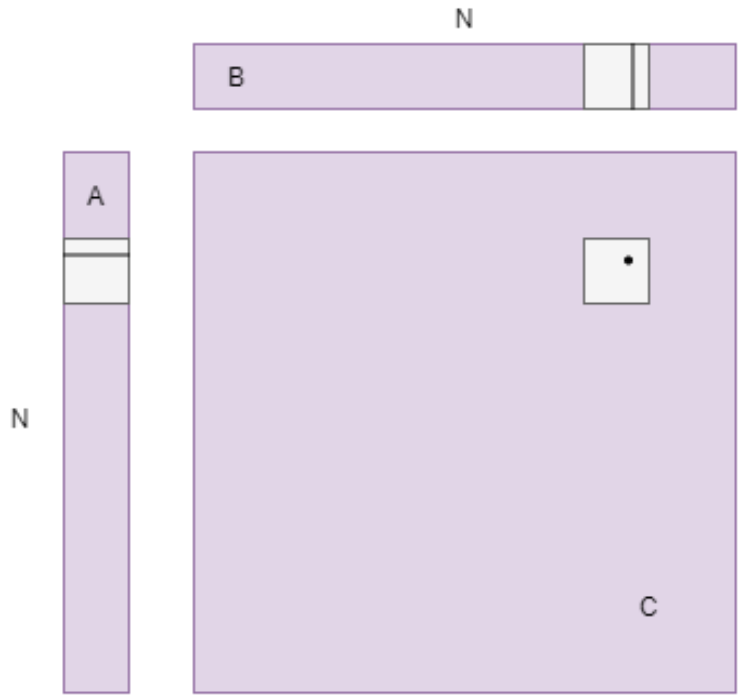
그리고 나서 실험 목표에 맞는 환경을 맞추기 위해 mps를 사용했다. 단일 gpu에 하나의 mps를 올리기 위해 gpu의 compute mode를 'Exclusive Process'로 맞춰주었고, 'nvidia-cuda-mps-control -d'를 이용해 가상 mps deamon을 켜준다.

각각의 code에 존재하는 kernel (function <<< >>>)이 실행 될 때, cudaEventRecord()로 기록한 뒤, kernel이 실행이 완료되기 까지의 시간을 측정한다.

4.3.1 mm_normal.cu (AB = C)

A, B 행렬을 cudaMalloc으로 메모리 할당을 해준다. 그리고 나서 seed를 랜덤으로 준 뒤에 각 행렬을 랜덤의 숫자들을 넣어 완성해준다. cudaDeviceSynchronize()를 이용해 이 과정이 완성할 때까지 cpu에게 대기를 시킨다. 그리고나서 dimGrid와 dimBlock을 이용해 연산에 사용할 Block의 개수와 Thread 개수를 정해준 뒤에, function<<<dimGrid, dimBlock>>> 으로 kernel을 실행시킨다. matrix가 연산되는 방법은 [그림1]과 같다. 이 때, 이 kernel이 실행되기 직전과 후에 cudaEventRecord()로 기록해준다. 그 후에 cudaEventElapsedTime()을 이용해 kernel이 작동한 시간을 알아내어 출력시킨다.

[그림 1]



4.3.2 mm_shared.cu (AB = C, use shared memory)

A, B 행렬을 shared memory를 활용해 계산한다. shared memory를 이용한 계산은 [그림 1]에서 A, B의 작은 단위의 사각형을 shared memory에 저장한 뒤에 계산을 위해 matrix의 데이터를 global memory에 접근하는 대신에 shared memory에 접근해서 값을 가져온다. 그 외의 과정은 4.3.1과 같다.

4.3.3 high_core.cu (core intensive kernel)

memory access를 줄이고 core 연산에 좀더 중점을 둔 code이다. random으로 array를 만든 뒤에, 커널 내에서 사칙연산 횟수를 많은 횟수로 반복한다. 여기서 단순 계산의 경우 compile 과정에서 optimization될 가능성이 있기 때문에 각 연산이 변수끼리 dependency하게 연산을 만들 필요가 있었다. 그리고나서 배열의 순서를 한번씩 바꿔주는 방법으로 또 새로운 연산으로 느껴지게끔 만든다.

이러한 kernel들을 동시에 실행시키는 방법으로 성능저하를 확인해본다. malloc과 같은 과정들을 배제하고 kernel을 동시에 실행시키기 위해 function<<< >>> 전에 getchar()를 이용해 y를 입력받아야 실행이 되게 하였다.

두 번째로 persistent kernel로 인한 성능 저하를 알아보기 위한 실험에서는 mps환경을 사용하지 않는다. "echo quit | nvidia-cuda-mps-control"로 종료한 뒤에, compute mode 역시 "nvidia-smi -c 0"으로 Default로 변경한다. 이렇게 환경을 설정한 뒤에, persistent kernel[9]을 N개 실행시킨 뒤에 각각의 kernel들의 성능저하가 얼마나 일어나는지 확인한다.

5. 결과 분석

shared memory를 사용하지 않은 global memory만을 이용한 matrix multiplication kernel을 G-mem, shared memory를 사용한 matrix multiplication kernel을 S-mem이라 정의한다.

Memory intensive kernel에서 matrix row, column size는 4096이며, grid당 block을 4개, block당 thread를 64개로 설정했다. Core intensive kernel의 경우에는 grid당 block을 16개, thread를 256개로 설정했다.

5.1 mps 환경에서 kernel의 성능저하 확인

표 1, 표 2는 1개의 kernel(G-mem, S-mem)만이 실행되었을 때의 성능을 100% performance로 잡았다.

표 1

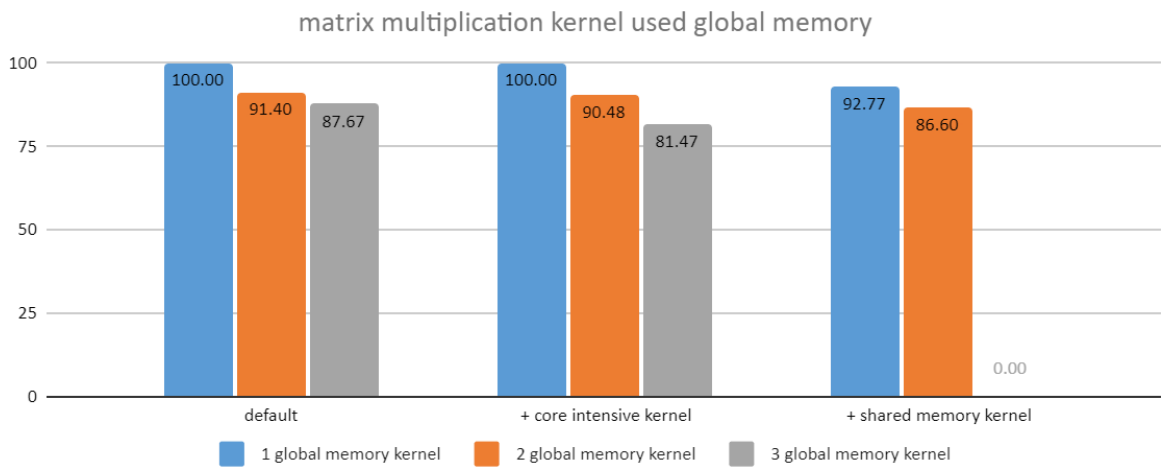


표 2

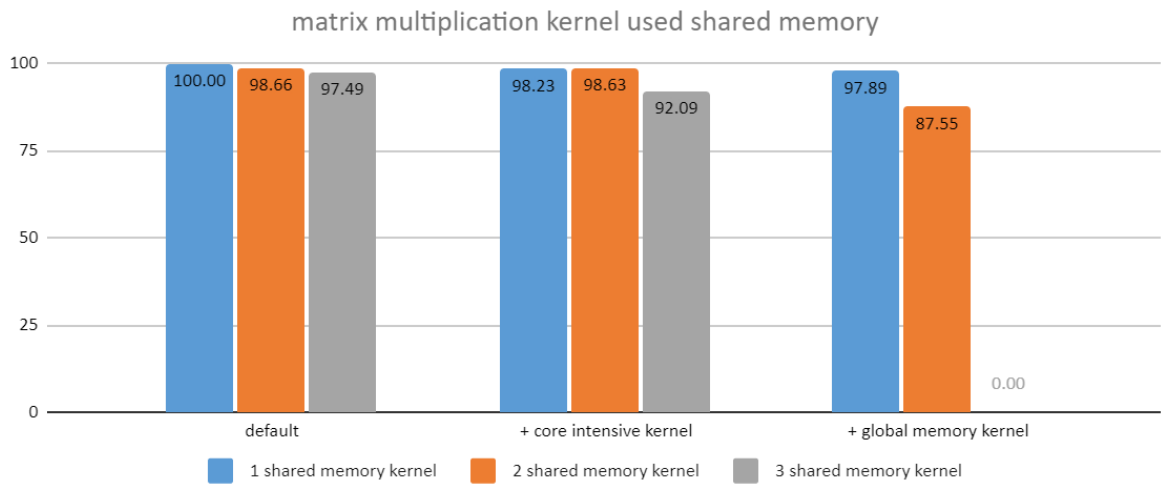


표 1은 G-mem의 개수를 늘려가며 실험한 결과이다. x축은 차례대로 G-mem만을 실행했을 경우, core intensive kernel을 하나 추가로 실행했을 경우, S-mem을 하나 추가로 실행했을 경우이다. 표 2는 S-mem의 개수를 늘려가며 실험한 결과이다. x축은 차례대로 S-mem만을 실행했을 경우, core intensive kernel을 하나 추가로 실행했을 경우, G-mem을 하나 추가로 실행했을 경우이다.

G-mem의 경우에는 개수가 늘어날수록 성능이 현저하게 떨어지는 것을 확인할 수 있었다. S-mem의 경우에는 개수가 늘어나도 크게 성능변화가 없는데, 우선 여기서 global memory에 접근하는 bandwidth의 한계로 인해 성능이 저하될 가능성이 있다. 연산과정은 거의 동일하나, shared memory의 사용 여부가 차이가 나기 때문이다.

그래서 memory access가 적고 연산을 많이 하는 core intensive kernel을 함께 실행해서 결과를 보았다. 그 결과 G-mem의 성능이 1개, 2개일때는 크게 변화가 없었다. 이는 S-mem역시 같은 결과가 나왔다. 이 결과로 memory access의 문제로 인한 것이 가능성이 더 높아졌다. 각 kernel이 3개씩 작동했을 때에는 memory문제를 동반한 연산처리를 할 자원부족으로 인한 큰 성능저하가 이루어진 것으로 추측된다.

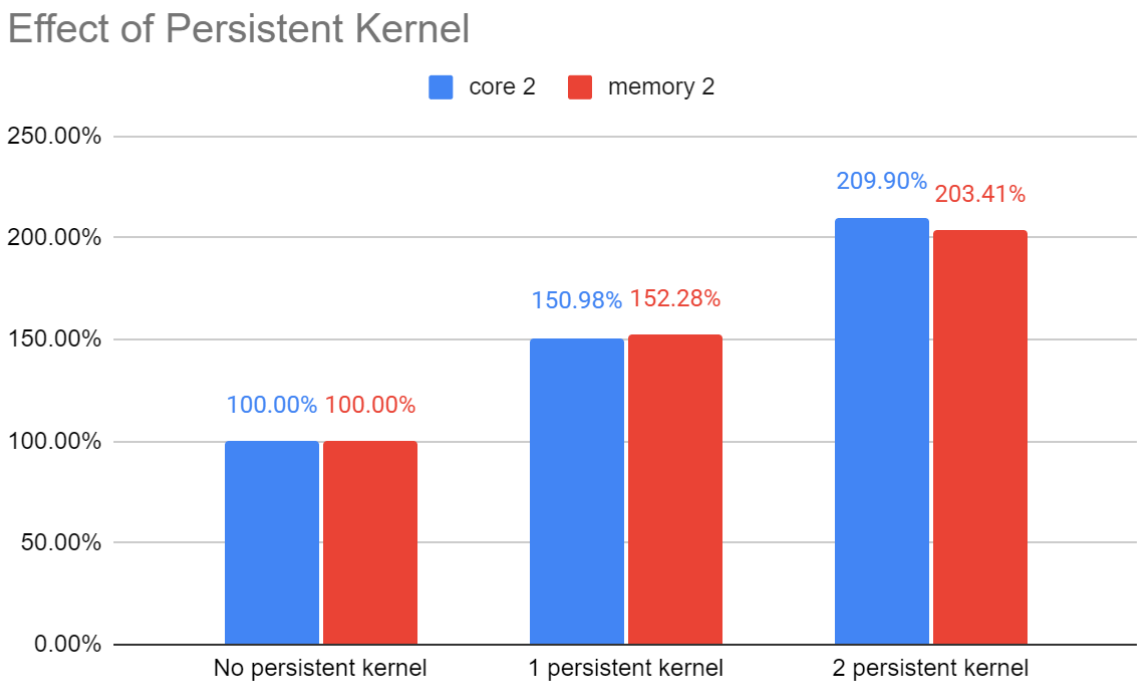
다음으로, cache hit ratio의 문제를 확인하기 위해. G-mem과 S-mem을 같이 실행시킨 결과를 확인한다. G-mem은 함께 실행시켰을 때, S-mem에 비해 크게 감소된, kernel을 두 개 실행시킨 성능을 보인다. 그에 반해 S-mem는 성능변화가 매우 작다. 이것은 상대적으로 global memory access를 많이 하는 G-mem의 cache hit ratio에 문제가 생겼기 때문에 S-mem는 변화가 작지만, G-mem의 경우 S-mem을 추가로 실행시킨 것과, G-mem을 추가로 실행시킨 것이 비슷한 성능이 나온 것으로 추정 가능하다.

그래서 kernel이 동시에 작동할 때, 공동 memory에 접근하는 과정에서 문제가 발생하거나, cache에 변화가 생기면서 hit ratio가 떨어져서 성능이 저하됨을 유추할 수 있다.

5.2 일반 커널과 persistent kernel을 통한 각각의 성능저하 비교

5.2.1 Effect of persistent kernel

표 3



core연산이 많은 kernel과 memory연산이 많은 kernel이 각각 두 개씩 실행되었을 때, persistent kernel로 인한 성능저하를 나타낸 그래프이다. 본 그래프를 보아 persistent kernel이나 kernel을 하나 더 실행했을 때나 성능저하가 거의 동일한 수준이라고 볼 수 있다.

5.2.2 Persistent kernel과 memory intensive kernel간의 성능 저하

표 4

high_core.cu \ persistent (ms)	0	1	2
1	13533.85666	27783.44553	42966.27656
2	27712.24946	41838.73589	58168.23359
3	41343.25709	57592.57002	73891.55469

표 5

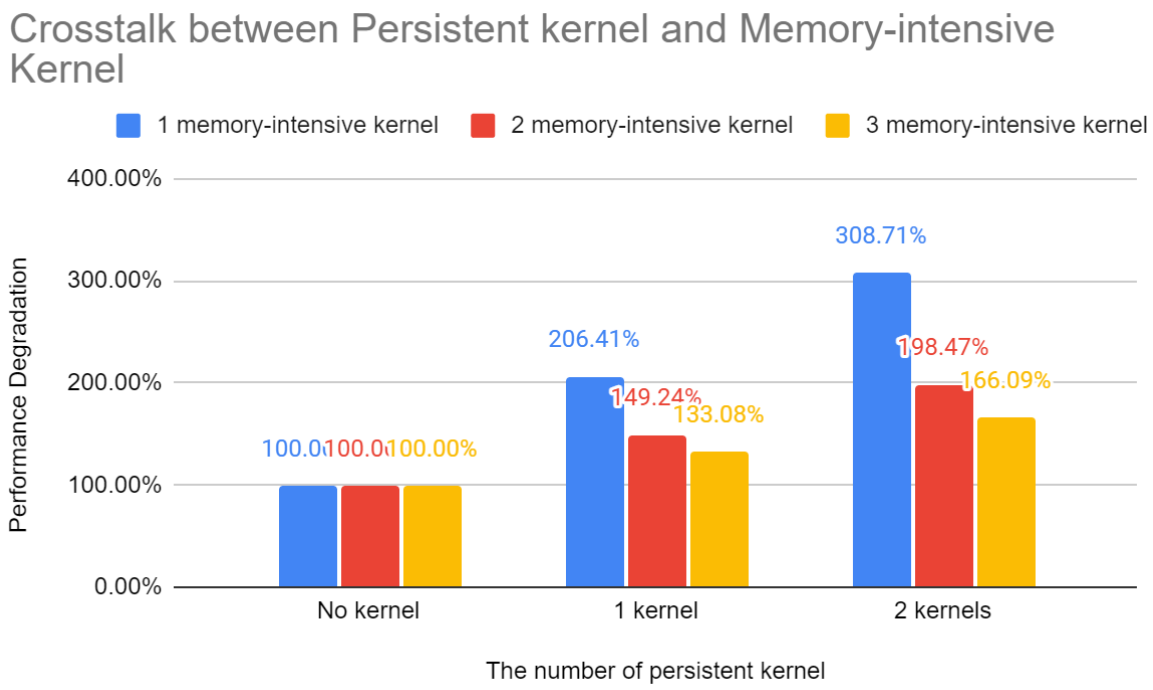


표 4, 표 5는 Persistent kernel과 memory 연산이 많은 G-mem과의 성능 저하를 확인한 표와 그래프이다. 표 5의 Y축의 값은 (N 개의 persistent kernel을 함께 실행시켰을 때의 시간) / (persistent kernel 이 없었을 때의 실행 시간) 을 의미한다.

표 4의 우상향 대각선과 표 5의 수치를 봤을 때, 시간의 거의 동일한 것을 보아 persistent kernel을 실행시켜 놓은 것과 kernel하나를 추가한 것이 거의 동일한 수준의 성능저하를 갖게 됨을 알 수 있다.

5.2.3 Persistent kernel과 core intensive kernel간의 성능 저하

표 6

Mm_normal.cu \ persistent (ms)	0	1	2
1	11679.13325	24597.12109	36846.69488
2	24106.37196	36709.97157	49034.9349
3	36054.5068	48818.85475	61197.94951

표 7

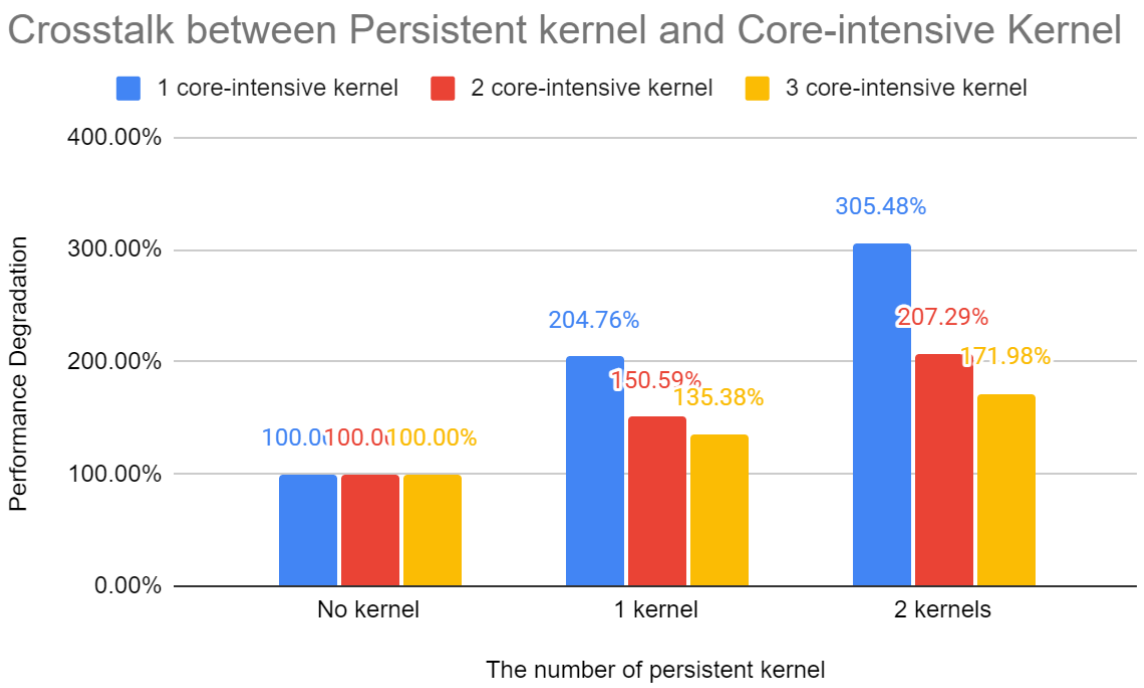


표 6, 표 7은 Persistent kernel과 core 연산이 많은 core intensive kernel(high_core.cu)과의 성능저하를 확인한 표와 그래프이다. 표 7의 Y축의 값은 (N 개의 persistent kernel을 함께 실행시켰을 때의 시간) / (persistent kernel 이 없었을 때의 실행 시간) 을 의미한다.

표 6의 우상향 대각선과 표 7의 수치를 봤을 때, 시간의 거의 동일한 것을 보아 persistent kernel을 실행시켜 놓은 것과 kernel하나를 추가한 것이 거의 동일한 수준의 성능저하를 갖게 됨을 알 수 있다.

6. 결론 및 소감

memory access가 큰 matrix multiplication 과 memory access는 적지만 연산을 많이 하는 code를 이용해 각 kernel들의 성능저하를 확인하고 원인을 분석했다. 그 결과 성능 저하의 원인은 memory bandwidth의 bottleneck 현상 및 낮은 cache hit ratio가 영향을 끼친다고 추정할 수 있었다.

Persistent kernel을 사용하는 것이 일반 kernel을 사용하는 것에 비해 다른 kernel의 성능저하에 더 큰 영향을 끼치지 않고 동등한 수준의 영향을 끼친다는 결과가 나왔다.

그동안 GPU 클러스터 기술의 발전은 병렬 프로세싱을 쉽고 편리하게 제공하기 위해 심플한 API를 통한 HW자원의 자동적인 활용을 추구해 왔으나, 여러 사용자가 공유하게 되면서 사용자 간의 간섭으로 인한 성능 저하 상황은 간과되어왔다. 그렇기 때문에, GPU 클러스터에서 활용될 수 있는 간섭 요인을 고려한 자원 통합 관리 기술의 연구가 필요한데, 이 연구의 기반으로 이용할 수 있을 것이다.

NVIDIA mps에서 cache 구조나 작동 과정에 대한 documents를 찾기 힘들어서 매우 아쉬웠다.

7. 참고문헌

[1] NVIDIA CUDA DOCS https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#arrive_wait

[2] Khalid, Junaid, et al. "Iron: Isolating Network-based CPU in Container Environments." 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018.

[3] K. Gupta, J. A. Stuart and J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads," 2012 Innovative Parallel Computing (InPar), 2012

[4] NVIDIA QUARTERLY REVENUE TREND https://s22.q4cdn.com/364334381/files/doc_financials/annual/2021/Rev_by_Mkt_Qtrly_Trend_Q421.pdf

[5] NVIDIA Cloud Computing https://www.nvidia.com/ko-kr/data-center/gpu-cloud-computing/

[6] NVIDIA MPS https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[7] NVIDIA MIG https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#changelog

[8] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis

C.M. Lau, Yuqi Wang, Yifan Xiong, Bin Wang "HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees" 14th USENIX Symposium on Operating Systems Design and Implementation, 2020

[9] https://github.com/Susoon/persistent_kernel_example_code.git