https://doi.org/10.22670/knom.2019.22.1.60

다중 프로세스 서비스를 이용한 GPU 응용 동시 실행 성능 분석

김 세 진◆, 오 지 선*, 김 윤 희°

A Execution Performance Analysis of Applications using Multi-Process Service over GPU

Se-Jin Kim◆, Ji-Sun Oh*, Yoonhee Kim°

요 약

Graphical Processing Units(GPUs)는 비교적 정형화된 연산을 병렬적으로 처리함으로써 높은 성능을 제공한다. 기술의 발전에 따라 GPU 환경에서 다양한 응용 실행을 시도하는 General Purpose GPU(GPGPU) 실행환경이 연구되고 있으나, 자원 분배, 스케줄링 등의 GPU 자원을 효율적으로 사용하기에는 아직 제한적이다. 최신의 GPU 구조들은 커널의 동시 실행을 지원하지만 같은 응용 안에서만 동시 실행이 가능하다는 문제점이 있어 NVIDIA는 Multi-Process Service(MPS)를 제안하였다. MPS는 다른 응용에 속한 커널도 동시 실행할 수 있도록 서비스한다. 하지만 응용의 실행 특성 및 동시 실행되는 패턴이 미리 파악되어 있지 않으면 MPS 장점을 최대한으로 취할 수 없다. 본 논문에서는 응용 프로파일링을 통해 응용의 특성을 파악하고, 동시 실행 스케줄링 알고리즘을 적용하여 실험을 진행하였다. MPS의 장점을 최대한으로 활용하기 위해서는 함께 돌릴 응용의 특성을 파악하고, 프로파일링을 통해 동시 실행하는 응용들의 순서를 제어하는 스케줄링 알고리즘이 중요함을 보인다.

Key Words : GPU, Multi-Process Service, GPGPU. kernel scheduling

ABSTRACT

Graphical Processing Units(GPUs) achieve high performance undertaking from relatively uniformed computation in parallel. The technology related to General Purpose GPU(GPGPU) has been enhanced, which provides concurrent kernel execution of multi and diverse applications at the same time, but it is still limited to support resource sharing or planning. NVIDIA recently introduces Multi-Process Service(MPS), which allows kernels from different applications can be execute concurrently. However, the strength of MPS comes along with the characteristics of applications and the order of their execution. This paper shows the performance analysis of diverse scientific applications in real world. Based on the analysis, we prove that it is important to the identify characteristics of co-run applications, and to schedule multiple applications via profiling to maximize MPS functionality.

※이 논문은 2015년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(2015M3C4A7065646)
◆ First Author : Sookmyung Women's University, wonder960702@gmail.com
° Corresponding Author : Sookmyung Women's University, yulan@sookmyung.ac.kr
* Sookmyung Women's University, jsoh8088@gmail.com

논문번호 : KNOM2019-01-13, Received June 22, 2019; Revised July 22, 2019; Accepted August 15, 2019

Ⅰ. 서 론

Graphics Processing Units(GPUs)의 사용을 통해 다양한 응용들이 강력한 처리 성능을 받음으로써 여러 분야에서 이를 사용하는 것이 점점 더 보편화 되고 있다. 이는 응용의 연산 집약적인 부분을 병렬적으로 처리함으로써 처리 속도를 높일 수 있다.

특히 NVIDIA GPU의 Compute Unified Device Architecture(CUDA) 프로그래밍 모델은 커널(kernel)을 GPU에서 연산을 수행하기 위해 동작하는 함수로 정의한다. 이는 여러 개의 스레드가 프로그래머의 정의에 따라 스레드 블록을 이루며 각 스레드 블록은 하나의 Streaming Multiprocessor(SM)에서 실행된다. GPU는 수천 개의 코어에서 스레드를 병렬(parallel) 실행하여 응용들을 가속해왔다[5]. 하지만 GPU의 하드웨어 자원이 증가하면서 스레드 병렬성만으로는 GPU 자원들을 충분히 활용하지 못하는 문제점이 발생하였다[1].

이를 위해 최근의 GPU 구조들은 프로세스(응용)의 독립적인 커널들을 다른 스트림으로 지정함으로써 동시 실행(concurrent)을 제공한다. 하지만 이는 같은 프로세스 안에서만 동시 실행이 가능하다는 한계가 있다. 최근의 연구들은 응용들이 대부분 한 가지 자원만 고갈시키며, 나머지 자원들은 활용하지 않는다는 특성을 보여주고 있다[11]. 따라서 이러한 특성을 가진 응용들의 실행은 GPU의 자원 활용률을 저하한다. 이의 한계를 해결하기 위해서 NVIDIA는 Multi-Process Service(MPS)를 제공한다. MPS는 다른 프로세스의 커널을 하나의 프로세스처럼 보이도록 함으로써 동시 수행이 가능하게 한다.

그러나 응용이 동시 실행되는 패턴이 미리 파악되어 있지 않으면 MPS의 장점을 최대로 취할 수 없다. MPS는 응용의 특성을 고려하지 않고, 단순히 사용자가 제출한 응용들을 동시 실행시키기 때문이다. MPS를 사용하여 동시 수행하여도 응용 특성에 따라 성능의 향상이 없을 수도, 나빠질 수도 있다. 그러므로 응용의 프로파일링을 통해서 응용의 성격 및 동시 실행 패턴을 파악하고, 동시 실행 스케줄링 알고리즘을 통해 응용들의 순서를 제어하여 서비스를 이용하는 것이 중요하다.

스케줄링의 목적은 같은 GPU에서 동시에 수행할 응용을 선택하여 성능을 향상하는 데에 있다. 탐욕 알고리즘을 사용한 스케줄링은 이해 및 구현하기에 쉽지만, 단계마다 이득을 최대화하는 방향으로 진행되므로 전체적으로 최적의 결과를 낼 수 없다. [10] 논문에서는 시스템의 성능을 최적화하기 위해 그래프 기반 알고리즘을 사용하여 동시에 실행할 커널의 쌍을 선택하였다. 태스크 사이의 관계를 그래프로 모델링 하고, 이에 대해 [10] 논문에서 제안한 max matching 알고리즘을 적용하여 최적화를 진행한다.

본 논문에서는 MPS를 사용한 CUDA 과학 응용의 실행 패턴을 분석하며, 각 응용 간의 동시 실행을 통해 MPS의 커널 수행방식과 응용의 특성을 파악한다. 이를 바탕으로 max matching 알고리즘을 적용하여 생성된 가중치 그래프를 사용하여 응용을 스케줄링 한다. 이를 응용의 쌍을 무작위를 선택하는 랜덤 스케줄링 알고리즘 및 탐욕 알고리즘과 비교하여 MPS를 사용하여 응용을 동시 실행할 때 프로파일링을 통한 응용의 순서 제어 스케줄링의 중요성을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 MPS의 구성과 기능을 설명한다. 3장에서는 Max Pair 알고리즘을 소개하며 4장에서는 실험을 통해 MPS의 수행방식과 Max Pair 스케줄링을 평가한다. 5장에서는 결론 및 향후 연구에 관하여 기술한다.

Ⅱ. Multi-Process Service

CUDA 응용은 사용할 하드웨어 자원이 담긴 CUDA context를 생성하며 시작된다. Hyper-Q 기술은 하나의 커널이 모든 GPU 자원을 사용하지 않는다면, 같은 CUDA context에 속한 다른 커널을 동시에 실행시킬 수 있도록 한다[6]. CUDA 모델에서 스레드 병렬성과 함께 동시에 실행될 수 있는 CUDA 작업은 커널, 디바이스-호스트간 메모리 복사작업, CPU에서의 작업이다. Hyper-Q 기술을 가진 GPU의 동시 실행 스케줄러(concurrent scheduler)는 이 작업을 중첩해 같은 context에 속한 여러 프로세스가 GPU 자원을 동시에 활용할 수 있도록 한다. 동시 실행 스케줄러는 커널과 디바이스-호스트 간 메모리 복사작업을 가지는 세 개의 작업이 제출되었을 때, 그림 1과 같이 커널과 메모리 복사작업을 중첩한다. 하지만 다른 context에 속한 커널들은 time-sliced 스케줄러를 통해 그림 1과 같이 순차적으로 수행됨으로써 실행시간이 늘어남을 확인할 수 있다. 그러므로 하나의 context가 충분한 자원을 사용하지 않는다면 GPU의 자원이 낭비될 수 있다.

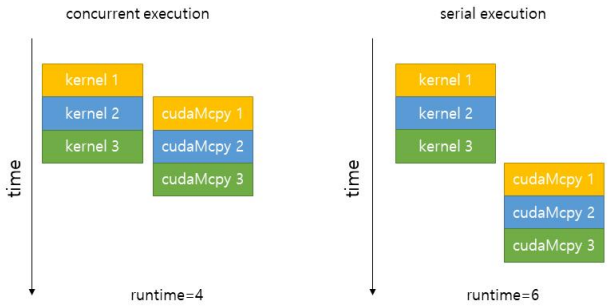

그림 1. 동시 실행과 순차 실행
Fig. 1. Concurrent execution and serial execution

MPS는 다른 CUDA context로부터 온 여러 커널이 MPS 서버를 통해 작업을 제출함으로써 하나의 context로 관리되어 Hyper-Q 기술을 최대한으로 활용할 수 있도록 제공하는 CUDA API이다. GPU 남는 자원을 다른 프로세스의 커널이 사용할 수 있도록 하여 MPI(Message Passing Interface) 작업을 실행하기에 적합하다[2]. MPS의 구조는 그림 2와 같이 서버 프로세스, 클라이언트 프로세스, MPS 컨트롤 데몬 프로세스로 구성되어 작동한다.

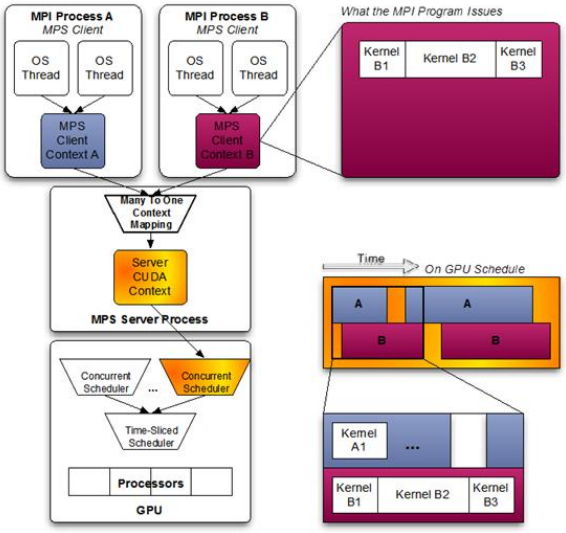

그림 2. MPS 클라이언트-서버 구조[2]
Fig. 2. Client-server architecture of MPS

단일 프로세스는 GPU에서 사용 가능한 자원을 모두 사용하지 않는다. MPS는 서로 다른 프로세스(응용)의 커널과 호스트와 디바이스간의 메모리 복사작업을 중첩해서 시간을 단축하고, GPU 활용률을 높일 수 있다. GPU는 각 프로세스에 저장 공간과 스케줄링 자원을 할당한다. 따라서 여러 프로세스가 GPU에서 실행되면 스케줄링 자원이 GPU에서 스와핑 되어야 한다. 하지만 MPS 서버는 모든 MPS 클라이언트가 GPU의 저장 공간과 스케줄링 자원을 공유하게 하므로 클라이언트를 스케줄링할 때 스와핑 오버헤드를 제거한다.

Ⅲ. MaxPair

응용 간 동시 실행의 연관성을 그래프로 묘사될 수 있다. 각 정점은 각 응용을 나타내고, 간선은 응용이 동시 실행을 통해서 성능이 향상됨을 나타낸다. 각 간선은 가중치를 가지는데, 이는 동시 실행을 통해 얻을 것으로 기대하는 이득을 가리킨다.

1. 그래프 매칭

그래프 이론에서의 매칭은 공통 정점을 가지지 않는 간선의 집합을 의미한다. 그림 3에서 빨간색 선으로 표시된 간선들의 집합이 매칭을 보여준다. 매칭의 결과로써 간선의 집합과 간선으로 연결되지 않은 개별 정점의 집합이 주어진다. 매칭의 결과는 스케줄링을 의미하며, 간선으로 연결된 노드는 동시 실행되며 개별 노드는 독점적으로 GPU를 사용하게 된다.

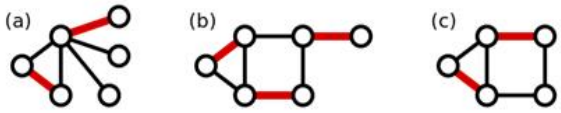

그림 3 그래프 이론에서의 매칭[12]
Fig. 3. Matching in a graph

최적화를 위한 초기 단계로써, 그림 4와 같이 동시 수행했을 때 성능이 저하됨을 의미하는 간선들은 제거한다. 이 스케줄링은 응용들을 동시 실행해서 이득을 얻기 위함이므로, 동시 실행하였을 때 성능이 저하되는 응용의 쌍은 독립적으로 실행함을 의미한다.

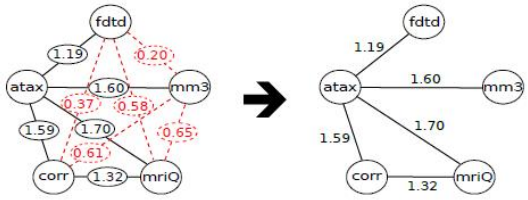

그림 4. 최적화의 초기 단계[10]
Fig. 4. Initial optimization

2. 가중치

알고리즘을 적용하여 그래프 전체의 가중치 합을 최대화한다. 이때 가중치로써 세 가지를 고려할 수 있다. 이는 단일 가중치(unified weight), 상대 속도 향상 가중치(relative speedup), 시간 가중치(time

weight)이다.

1) 단일 가중치(Unified weight) : 모든 간선에 같은 가중치를 주는 것을 의미한다. 매칭을 통해 나오는 결과는 동시 실행을 통해 이득을 수 있는 응용 쌍 개수의 최댓값이다.
2) 상대 속도향상 가중치(Relative speedup weight) : 각 간선에 상대 속도향상을 지정해 준다. 상대 속도향상은 두 응용을 순차적으로 실행한 시간을 동시 실행한 시간으로 나눈 값을 의미한다. 이를 통해 전반적인 속도향상을 기대할 수 있다.
3) 시간 가중치(Time weight) : 각 간선에 순차적으로 응용을 실행했을 때와 비교하여 동시 실행했을 때 절약한 시간을 지정한다. 긴 수행시간을 가지는 응용의 경우, 상대 속도향상이 높지 않아도 절약되는 시간이 클 수 있으므로 전체적인 시간을 절약할 수 있다.

3. 알고리즘

동시 실행할 응용의 쌍을 찾기 위해서 최대 가중치 매칭 알고리즘(maximum weight matching)을 적용한다. 최대 가중치 매칭은 최대 매칭(maximum cardinality matching)의 변형이다.

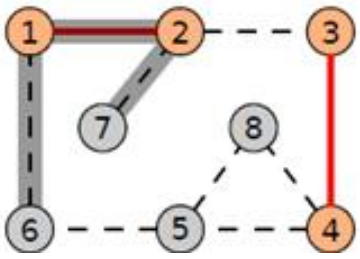

그림 5 Augmenting 경로[10]
Fig. 5. Augmenting path

먼저, 최대 매칭은 가장 간선의 개수가 많은 매칭을 찾는 알고리즘이다. 주어진 그래프에 대해서, 각 노드는 single 또는 matched 노드로 나누어진다. 그림 5에서 주황 색깔 노드는 matched 노드를, 회색 색깔 노드는 single 노드를 의미한다. 각 간선에 대해서 matched 노드 사이의 간선은 matched 간선이라 하며, 이 외의 간선은 unmatched 간선이라 한다. 그림 5에서 빨간색 간선이 matched 간선을, 점선으로 된 검은색 간선이 unmatched 간선을 나타낸다. 경로 중 matched 간선과 unmatched 간선이 번갈아서 등장하는 경로를 alternating 경로라 하며, alternating 경로 중 single 노드에서 시작하여 single 노드에서 끝나는 경로를 augmenting 경로라고 한다. 예를 들어, 그림 5에서 강조된 6-1-2-7의 경로가 augmenting 경로이다. 이 알고리즘은 single 노드에 대하여 augmenting 경로를 반복적으로 찾아서 더는 augmenting path가 없을 때 알고리즘을 중지한다. 이를 변형한 최대 가중치 매칭 알고리즘은 그림 6에서 보여준다. 알고리즘에 대한 자세한 설명은 [10] 에서 참고하였다.

```
INPUT: Graph G, matching M on G;
OUTPUT: A maximum matching M*x on G;
Initialization: All nodes in G are unmatched nodes;
for each unmatched node in G do
    instructions;
    if the vertex is an unvisited node then
        if this vertex has been matched then
            extend augmenting path P;
        else
            found an augmenting path P;
            update M by P⊕M;
        end
    else
        if found a blossom then
            replace the blossom by a super node;
        end
    end
end
```

그림 6. MaxPair 알고리즘[10]
Fig. 6. The MaxPair algorithm

Ⅳ. 실험 및 분석

1. 실험 환경

본 연구에서 MPS의 프로세스 실행 방식의 진행과 성능을 확인하고, max matching을 적용한 스케줄링 방식을 평가하는 실험을 진행한 환경은 표 1과 같다.

표 1. 실험 환경
Table 1. Experimental environment

	CPU	GPU
Architecture	Intel(R) Core(TM) i7- 5820K	Nvidia GeForce Titan Xp D5x
Core clock	3.30GHz	1.58GHz
Num of Cores	6 cores	3840 cores
Memory size	32GB	12GB
Threading API	-	Nvidia CUDA 10.1
Compiler	Gcc 5.4.0	Nvidia C Compiler (NVCC8.0)
OS	Ubuntu 16.04.3 LTS	Ubuntu 16.04.3 LTS

2. 대상 응용

과학 계산 응용인 5개의 응용을 대상으로 실험을 진행하였다. LAMMPS 응용은 분자 동역학 시뮬레

이션을 위한 소프트웨어이다. 본 실험에서는 kokkos package를 사용하였으며, input file은 Leonard Jones 3D melt example[13]을, 격자의 크기를 나타내는 x, y, z 값은 각각 8을 사용하였다. LAMMPS 응용의 총 수행시간은 1196초이며 수행 동안 363MB~8.3GB의 메모리를 사용한다. 이 응용은 호스트와 디바이스간 메모리 복사가 빈번하게 일어나서 GPU 커널이 실행되는 비율을 나타내는 GPU의 평균 활용도(utilization)가 약 25%이다.

GROMACS는 물 분자 데이터 세트를 사용하여 뉴턴의 운동 방정식을 시뮬레이션 한다. 본 실험에서는 Molecular Dynamics simulation을 수행하는 mdrun engine을 수행하였으며, 10000 스텝을 수행하도록 정의해주었으며 입력 데이터는 water GMX50 bare benchmark data[14]를 사용하였다. GROMACS 응용의 경우, 총 수행시간은 636초이며 수행 동안 171MB~537MB의 메모리를 사용한다. 이 응용은 수행시간 동안 빈번히 연산이 실행되어 평균 활용도(utilization)가 69%를 보이며 계산 집약적인 형태를 확인할 수 있다.

QMCPACK 응용은 Quantum Monte Carlo 알고리즘을 실행하는 고성능 전자 구조 코드이다. 실행 시작 후 95초 동안은 연산 작업을 하지 않는 모습을 보이다 이 이후부터 약 120초 동안 평균 활용도가 약 90%로 연산 집약적인 형태를 보인다. 또한, 209MB~5391MB의 메모리를 사용한다. 본 실험에서는 산화니켈에 대해 확산 몬테카를로 기법(Diffusion Monte Carlo)을 적용하는 예제[8]를 사용하였다.

HOOMD 응용은 입자를 시뮬레이션 하는 툴킷이다. 미세 액적 상태 별 모양 고분자의 소산 입자 동역학 시뮬레이션[9]을 벤치마크로 사용하였다. 이 응용은 시작 5초 후부터 평균 GPU 활용도가 100%인 연산 집약적인 응용임을 알 수 있으며, 1575MB~2609MB의 메모리를 사용한다.

3. MPS 오버헤드

본 실험에서 사용한 네 개의 응용을 하나의 GPU에서 독립적으로 수행할 때 MPS를 사용하였을 때와 사용하지 않았을 때의 시간은 표 2와 같다. MPS를 사용하였을 때, 사용하지 않았을 때와 비교하여 수행시간이 HOOMD, GROMACS, LAMMPS, QMCPACK 각각 1초(%), 12초, 5초, 15초 늘어난 것을 확인하였다. 오버헤드가 응용 별로 다르다. 이는 MPS의 프로세스 간 상호작용으로 인한 오버헤드로 추측이 되지만 추후 분석이 필요하다.

표 2. 응용의 수행시간 비교
Table 2. Execution time comparison of each application with MPS

	HOOMD	GROMACS	LAMMPS	QMCPACK
MPS x	345s	636s	1196s	200s
MPS o	346s	641s	1208s	215s

4. 동시 수행 그래프

본 실험에서 세 가지 가중치를 사용하므로, 표 3과 표4는 각각 두 개의 커널 쌍을 수행하였을 때의 상대 속도향상과 수행시간을 보여준다. 응용 간 동시 수행을 할 때의 관계를 나타내는 동시 실행 그래프는 그림 x 와 같다. 이는 상대 속도향상을 가중치로 선택하여 나타낸 그래프이다. 동시 수행하였을 때 성능이 낮아지는 LAMMPS-QMCPACK 간선은 초기화 단계에서 제거한다.

표 3. 응용 동시 실행의 속도향상
Table 3. Speedup of concurrent execution

	LAMMPS	GROMACS	HOOMD	QMCPACK
LAMMPS	OOM	2.29x	1.02x	0.97x
GROMACS	-	1.04x	1.29x	1.06x
HOOMD	-	-	1.08x	1.15x
QMCPACK	-	-	-	1.16x

표 4. 응용 동시 실행 수행시간
Table 4. Execution time of co-execution

	LAMMPS	GROMACS	HOOMD	QMCPACK
LAMMPS	OOM	809s	1531s	1471s
GROMACS	-	1233s	766s	806s
HOOMD	-		640s	486s
QMCPACK	-			370s

5. MPS 수행 패턴

MPS를 사용한 응용의 동시 실행을 통해 MPS의 커널 수행방식을 확인하기 위해, 동시 작업을 GROMACS- GROMACS로 구성하여 실험하였다. GROMACS-GROMACS 작업의 경우 MPS를 사용하지 않고 실행했을 때 1391초, MPS를 사용하였을 때는 1233초 수행시간이 소요된다.

그림 8은 MPS를 사용하지 않았을 때 LAMMPS-GROMACS 실행을 179.5초부터 180.15

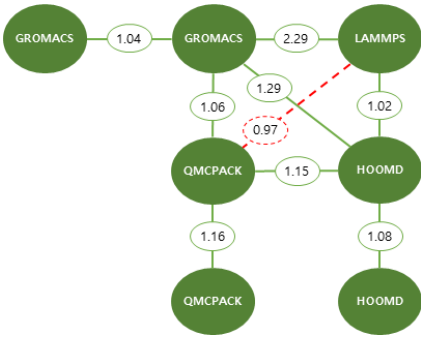

그림 7. 동시 실행 그래프
Fig. 7. Application co-execution graph

초까지 프로파일링한 타임라인이다. 두 프로세스는 다른 CUDA context에 속하여 실행된다. 커널(하늘색 작업)들이 동시에 제출이 되어 time sliced 스케줄러가 시간을 나누어 순차 실행이 되고, 문맥이 교환될 때 스케줄링 자원이 GPU에서 스와핑된다. 호스트-디바이스간 메모리 작업(황토색 작업)과 커널이 중첩되어 실행되기보다는, 커널이 동시 실행되므로 GPU 자원을 두고 경쟁이 일어나 효율성이 떨어지며 수행시간도 길어진다.

그림 9는 MPS를 통해 두 응용을 수행했을 때 173.3초부터 173.9초까지의 타임라인을 나타낸다. 173.43초부터 약 0.15초 간격으로 두 번째로 제출한 프로세스의 메모리 복사작업과 첫 번째로 제출한 프로세스의 커널이 중첩되며, 173.46초부터 약 0.15초 간격으로 첫 번째 프로세스의 메모리 복사작업과 두 번째 프로세스의 커널이 중첩되는 것을 확인할 수 있다. 또한, MPS를 사용하여 각 프로세스의 두 개의 커널을 실행하였을 때는 70밀리 초가 소요되었지만, MPS를 사용하지 않았을 때는 같은 커널의 실행에 대해 110밀리 초가 소요되었다. 이를 통해 MPS는 스케줄링 자원의 GPU 스와핑을 제거하여 문맥 교환의 오버헤드를 축소하고, 커널과 메모리 복사작업을 중첩하여 수행시간을 단축할 수 있다는 것을 알 수 있다.

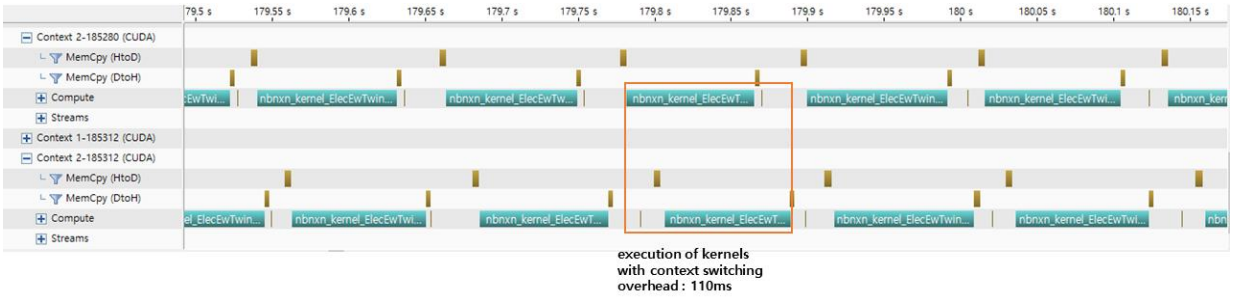

그림 8. MPS를 사용하지 않은 GROMACS-GROMACS 작업의 수행 프로파일
Fig. 8. Execution profile of GROMACS-GROMACS without MPS

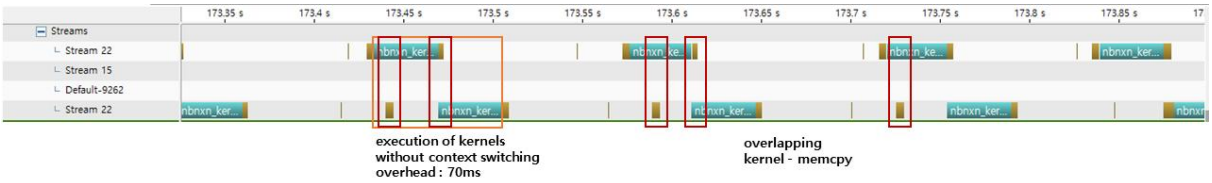

그림 9. MPS를 사용한 GROMACS-GROMACS 작업의 수행 프로파일
Fig. 9. Execution profile of GROMACS-GROMACS with MPS

6. MaxPair 스케줄링 성능 평가

그림 10은 네 개의 응용으로 이루어진 워크로드에 대하여 각 스케줄링 방법에 따른 수행시간을 나타낸 그래프이다. 순차 실행 스케줄링은 모든 응용이 각각 하나의 GPU를 독점적으로 사용하므로 3612초가 걸림으로써 가장 나쁜 성능을 보였다. 알고리즘의 성능을 비교하기 위한 기준 알고리즘으로는 랜덤 알고리즘을 사용하였다. 이는 응용들의 쌍을 무작위로 선택한 뒤 동시에 실행하는 알고리즘으로써, {GROMACS-GROMACS, QMCPACK-QMCPACK, LAMMPS-HOOMD, HOOMD}를 선택하여 3480초가 소요되었다. 각 단계별로 가장 높은 상대속도 향상을 가지는 간선을 선택하는 속도향상 탐욕 알고리즘은 {GROMACS-LAMMPS, QMCPACK-QMCPACK, HOOMD-HOOMD, GROMACS}를

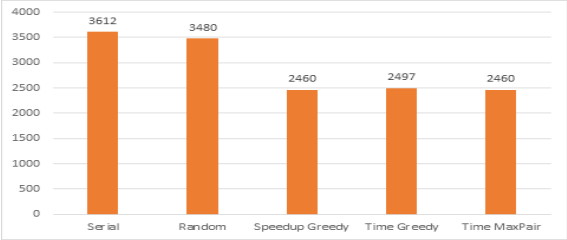

그림 10. 스케줄링 방법에 따른 워크로드 수행시간
Fig. 10. Execution time of workload using different scheduling strategies

선택하여 2460초, 단계별로 가장 시간이 많이 단축되는 쌍을 선택하는 시간 탐욕 알고리즘은 {GROMACS-LAMMPS, GROMACS-HOOMD, QMCPACK, HOOMD, GROMACS}를 선택하여 2497초가 소요된다. MaxPair 알고리즘은 가중치를 시간으로 주었을 때와 상대 속도 향상으로 주었을 때 모두 속도향상 탐욕 알고리즘과 같이 2460초 동안 모든 작업을 수행하였다. 이는 기준 알고리즘인 랜덤 알고리즘보다 성능이 약 41% 향상하였으며, 시간을 가중치로 하여 탐욕 알고리즘을 진행하였을 때보다 시간이 약 1.5% 감소한 것을 알 수 있다. 본 실험에서는 속도향상 탐욕 알고리즘과 MaxPair 알고리즘이 같은 성능을 보였으나, 다른 워크로드에 대해서는 탐욕 알고리즘이 항상 최적의 결과를 선택하지 않아 MaxPair 알고리즘의 성능이 탐욕 알고리즘과 같거나 우수하다[10]. 따라서 MPS를 통해 여러 응용을 동시에 수행할 수 있도록 하였지만, MPS 서버에서 서로 영향을 주는 응용들을 고려하지 않는다면 이의 장점을 최대한으로 활용할 수 없으므로 함께 돌릴 응용의 특성을 파악하고, 프로파일링을 통해 응용의 쌍을 결정하는 스케줄링 알고리즘이 중요하다.

V. 결 론

MPS를 사용하여 응용을 동시 수행하여 스와핑 오버헤드를 제거하고 병렬적으로 수행할 수 있는 메모리 작업과 커널을 중첩함으로써 GPU 자원 활용도를 높일 수 있다. 하지만 MPS는 응용의 특성을 고려하지 않고, 사용자가 제출한 응용들을 동시 수행시키는 역할만 하므로, 응용의 특성 및 동시 수행 패턴을 미리 파악하지 않으면 MPS의 장점을 최대한으로 취할 수 없다. [10] 논문에서 제안한 MaxPair 스케줄링 방법을 MPS 위에서도 적용한 실험을 통해, 이 스케줄링이 MPS 상에서도 잘 동작하며, MPS를 사용하는 장점을 최대화하기 위해서는 응용의 특성을 파악하고, 프로파일링을 통해 동시 실행하는 응용들의 스케줄링이 필요한 것을 확인할 수 있었다.

향후 연구로는 체크 포인팅을 사용하여 다중 작업의 동시 실행을 위한 실행 제어를 지원하는 스케줄러에 관해 연구하고자 한다.

References

[1] Carvalho, Pablo, et al. "Kernel concurrency opportunities based on GPU benchmarks characterization." Cluster Computing (2019): 1-12.

[2] Multi-Process Service, https://docs.nvidia.com/deploy/mps/index.html

[3] LAMMPS, https://lammps.sandia.gov/.

[4] GROMACS, http://www.gromacs.org/

[5] CUDA C Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[6] HyperQ, http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

[7] GPU-accelerated GROMACS, https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/gromacs/

[8] S32 example, https://drive.google.com/open?id=1Kgsm3py0oH4699Cmkg4d5BsZqL_sWoEc

[9] Microsphere benchmark, https://github.com/joaander/hoomd-benchmarks/tree/master/microsphere

[10] Wen, Yuan, Michael FP O'Boyle, and Christian Fensch. "MaxPair: enhance OpenCL concurrent kernel execution by weighted maximum matching." *Proceedings of the 11th Workshop on General Purpose GPUs. ACM*, 2018.

[11] Adriaens, Jacob T., et al. "The case for GPGPU spatial multitasking." *IEEE International Symposium on High-Performance Comp Architecture. IEEE*, 2012.

[12] Matching (graph theory), https://en.wikipedia.org/wiki/Matching_(graph_theory)

[13] Leonard Jones 3D melt example, https://lammps.sandia.gov/inputs/in.lj.txt

[14] water GMX50 bare, ftp://ftp.gromacs.org/pub/benchmarks/water_GMX50_bare.tar.gz

김 세 진 (Se-Jin Kim)

2019년 2월 : 숙명여자대학교 소프트웨어학부 컴퓨터과학 전공 졸업

2019년 3월~현재 : 숙명여자대학교 컴퓨터과학과 석사과정

<관심분야> 클라우드 컴퓨팅, 분산처리, 고성능 컴퓨팅, 이기종 컴퓨팅

오 지 선 (Ji-Sun Oh)

2017년 2월 : 숙명여자대학교 컴퓨터과학부 졸업

2017년 3월~현재 : 숙명여자대학교 컴퓨터과학과 석사과정

<관심분야> 클라우드 컴퓨팅, 분산컴퓨팅, GPU 가상화

김 윤 희 (Yoon-Hee Kim)

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.

<관심분야> 클라우드 컴퓨팅, 워크플로우 제어, 그리드/클라우드 관리