한국컴퓨터정보학회 하계학술대회 논문집 제26권 제2호 (2018. 7)

통합 메모리를 사용하는 NVIDIA 파스칼 GPU에서의 CPU 메모리와 GPU 메모리 간 데이터 통신 분석

신필규O, 홍성수*
O*서울대학교 전기정보공학부
e-mail: pkshin@redwood.snu.ac.krO, sshong@redwood.snu.ac.kr*

Evaluation of the Data Migration between CPU Memory and GPU Memory for a NVIDIA Pascal GPU Using Unified Memory

Philkyue ShinO, Seongsoo Hong*
O*Dept. of Electrical and Computer Engineering, Seoul National University

● 요 약 ●

통합 메모리는 CPU 메모리와 GPU 메모리 간의 데이터 통신을 개발자에게 투명하게 내재적으로 수행하는 소프트웨어 런타임 환경으로 개발자에게 CPU 메모리와 GPU 메모리가 통합된 하나의 메모리로 보이게 해준다. 통합 메모리는 장점에도 불구하고 아직 널리 사용되지 못하고 있는데 그 이유는 내재적으로 수행되는 데이터 통신의 오버헤드가 큰 것으로 알려져 있기 때문이다. 하지만 이 데이터 통신이 구체적으로 어떻게 이루어지고 오버헤드는 어떻게 발생하는지 분석한 연구는 아직 존재하지 않는다. 우리는 NVIDIA 사의 최신 GPU 마이크로아키텍처 중 하나인 파스칼을 사용하는 GPU를 대상으로 하여, 통합 메모리를 사용할 시 데이터 통신이 이루어지는 조건과 GPU 응용의 수행시간에 데이터 통신이 끼치는 영향을 실험을 통해 분석한다. 실험 결과 통합 메모리의 오버헤드는 두 가지 원인 때문에 발생한다. 첫째, 통합 메모리를 사용하면 CPU 또는 GPU가 데이터에 접근할 때마다 이 데이터는 CPU 또는 GPU 메모리로 옮겨지고 옮겨진 데이터는 제거된다. 따라서 재사용할 데이터도 제거되어 추가적인 데이터 통신이 발생하고, 이 데이터 통신의 지연시간은 GPU 응용의 수행시간에 더해진다. 둘째, 통합 메모리를 사용하면 데이터 통신과 커널들이 서로 다른 스트림에 할당되어도 동시에 수행되지 못한다. 따라서 GPU 응용의 수행시간은 동시에 수행되던 데이터 통신과 커널의 수행시간만큼 증가한다.

키워드: 그래픽 처리 장치(GPU), 통합 메모리(unified memory), 데이터 통신(data migration)

I. Introduction

GPU는 데이터 병렬 처리에 강점을 가진 하드웨어로 그래픽 응용뿐만 아니라 심층 학습과 같은 광범위한 GPGPU 응용에 사용되고 있다. 이러한 GPU 응용들은 데이터 집약적인 특성이 있으므로 CPU 메모리와 GPU 메모리 사이의 데이터 통신은 개별 GPU에서 성능에 있어 매우 중요한 요소 중 하나다. CPU와 CPU 메모리 사이 또는 GPU와 GPU 메모리 사이와 비교해 CPU 메모리와 GPU 메모리 사이의 데이터 통신은 작은 대역폭을 지녀 병목현상을 일으킬 수 있기 때문이다 [4].

전통적인 GPU 프로그래밍 모델에서는 이 데이터 통신을 개발자가 명시적으로 관리하고 최적화한다. 하지만 이는 개발자에게 GPU의 메모리 아키텍처를 고려하여 프로그래밍해야만 하는 부담을 준다. NVIDIA와 AMD는 최근에 개발자의 이런 부담을 덜어줄 수 있는 통합 메모리를 제안했다. 통합 메모리는 CPU 메모리와 GPU 메모리 간의 데이터 통신을 사용자에게 투명하게 내재적으로 수행하는 소프트웨어 런타임 환경으로 개발자에게 CPU와 GPU 메모리가 통합된 하나의 메모리로 보이도록 해준다.

통합 메모리는 장점에도 불구하고 내재적으로 수행되는 데이터 통신의 오버헤드가 큰 것으로 알려져 아직 널리 사용되지 못하고 있다. 하지만 이 오버헤드가 어떤 상황에서 어떻게 발생하는지 분석한 연구는 아직 없다. 게다가 NVIDIA 사의 GPU는 가장 많이 사용되는 개별 GPU임에도 불구하고 이와 관련된 문서도 공개되어 있지 않다.

기존 연구들은 이런 상황에서 NVIDIA GPU에 대한 정보를 가지고 있는 NVIDIA 사의 연구원들에 의해 주로 수행되었다. 그들은 통합 메모리의 오버헤드로 GPU 페이지 폴트 발생 시 GPU의 연산기가 멈추는 것에 초점을 맞췄다. 여기서 GPU 페이지 폴트란 GPU가 GPU 메모리에 없는 데이터에 접근했을 때 발생하는 폴트로 CPU 메모리에서 GPU 메모리로 이 데이터를 옮기는 작업을 촉발한다. 그들의 연구는 GPU 응용 수행 중 GPU 페이지 폴트를 처리하는

누적시간을 줄이는 방향으로 이루어졌다[1, 2, 3, 4, 5]. 이 연구들은 단위 시간 동안 발생하는 GPU 페이지 폴트 횟수를 줄이는 연구들과 한 번의 GPU 페이지 폴트 처리에 걸리는 시간을 줄이거나 숨기는 연구들로 분류된다.

첫 번째 분류 연구의 핵심 아이디어는 앞으로 GPU가 이용할 데이터를 GPU 메모리로 미리 이동시키고 앞으로 이용하지 않을 데이터는 GPU 메모리에서 내보내는 것이다[1, 2, 3]. 두 번째 분류 연구의 핵심 아이디어는 GPU 페이지 폴트 발생 시에도 GPU 연산기가 멈추지 않도록 하거나[4], GPU 페이지 폴트 처리를 빨리할 수 있도록 GPU 메모리 아키텍처를 개선하는 것이다[5]. 하지만 이 연구들에서 GPU 페이지 폴트 발생 시 GPU 연산기의 수행이 멈추는 것이 GPU 응용의 수행시간에 어떤 상황에서 어떻게 영향을 주는지 설명되어 있지 않다.

본 논문에서는 NVIDIA 사의 최신 GPU 마이크로아키텍처 중 하나인 파스칼을 사용하는 GPU를 대상으로 하여, 통합 메모리를 사용할 시 CPU 메모리와 GPU 메모리 사이에서 데이터 통신이 이루어지는 조건과 GPU 응용의 수행시간에 이 데이터 통신이 끼치는 영향을 분석한다. 우리는 이를 위해 두 가지 실험을 수행한다. 첫 번째 실험에서는 데이터 통신과 커널들이 연속적으로 수행되는 상황에서 CPU에서 수행되는 함수와 GPU에서 수행되는 커널이 접근하는 데이터를 바꾸면서 데이터 통신과 커널의 수행시간을 측정한다. 두 번째 실험에서는 데이터 통신과 커널들을 여러 스트림에 할당하여 동시에 수행될 수 있도록 한 상황에서 위와 같은 것을 측정한다.

우리의 실험은 Ubuntu 17.10과 CUDA (Compute Unified Device Architecture) 9.2에서 동작하는 GTX 1080 Ti GPU에서 이루어졌다. 실험 결과 통합 메모리의 오버헤드는 두 가지 원인 때문에 발생한다. 첫째, 통합 메모리를 사용하면 CPU 또는 GPU가 데이터에 접근할 때마다 이 데이터는 CPU 또는 GPU 메모리로 옮겨지고 옮겨진 데이터는 제거된다. 따라서 재사용할 데이터도 제거되어 추가적인 데이터 통신이 발생하고, 이 데이터 통신의 지연시간은 GPU 응용의 수행시간에 더해진다. 둘째, 통합 메모리를 사용하면 데이터 통신과 커널들이 서로 다른 스트림에 할당되어도 동시에 수행되지 못한다. 따라서 GPU 응용의 수행시간은 동시에 수행되던 데이터 통신과 커널의 수행시간만큼 증가한다.

본 논문의 나머지는 다음과 같이 구성된다. 2장에서는 본 논문의 이해를 돕기 위해 CUDA 프로그래밍 모델에서의 CPU와 GPU 메모리 간 데이터 통신을 설명한다. 3장에서는 우리가 수행한 실험들과 그 결과를 설명한다. 그리고 4장에서 결론을 맺는다.

II. Data Migration between the CPU and GPU Memory on CUDA

이 장에서는 나머지 장들에 대한 이해를 돕기 위해 먼저 전통적인 CUDA 프로그래밍 모델과 명시적으로 이루어지는 CPU와 GPU 메모리 간의 데이터 통신을 설명한다. 그리고 통합 메모리와 내재적으로 이루어지는 데이터 통신을 설명한다. 마지막으로 데이터 통신에 걸리는 지연시간을 숨길 수 있도록 CUDA에서 제공되는 스트림이라는 메커니즘에 대해 설명한다.

1. Traditional CUDA programming model and explicit data migration

CUDA는 NVIDIA GPU에서 범용 컴퓨팅을 수행하기 위한 병렬 컴퓨팅 플랫폼이자 프로그래밍 모델이다. CUDA API를 사용하여 작성된 CUDA 프로그램은 CUDA 라이브러리와 디바이스 드라이버를 통해 GPU를 사용하면서 수행된다.

전통적인 CUDA 프로그램은 크게 세 단계를 거쳐 수행된다. 첫 번째 단계에서는 CPU 메모리에 있는 데이터를 GPU 메모리로 옮긴다. 이는 cudaMemcpy()라는 CUDA API를 통해 이루어진다. 두 번째 단계에서는 GPU 메모리에 있는 데이터를 사용하여 연산을 수행한다. 수행하는 연산은 C나 C++ 언어의 함수로 표현되는데, 이를 커널이라고 한다. 개발자는 커널을 수행시킬 때 스레드 수와 블록 수를 지정하여야 한다. 스레드 수는 동시에 커널을 수행하는 인스턴스의 수를 의미하고 블록 수는 동시에 수행되는 인스턴스 묶음의 수를 의미한다. 세 번째 단계에서는 연산 수행 결과를 다시 cudaMemcpy()를 사용하여 GPU 메모리에서 CPU 메모리로 옮긴다.

Program 1: Single stream

```
1:  kernel vecAdd(int * a, int * b, int * c, int n)
2:      i = blockIdx.x * blockDim.x + threadIdx.x
3:      if(i < n)
            c[i] = a[i] + b[i]
4:  end kernel
5:
6:  procedure MAIN
7:      ...
8:      vectorAdd<<<numBlocks,blockSize>>>(a,b,c,n)
9:      for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + a[i]
10:     vectorAdd<<<numBlocks,blockSize>>>(a,b,c,n)
11:     for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + a[i]
12:     for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + b[i]
13:     vectorAdd<<<numBlocks,blockSize>>>(a,b,c,n)
14:     for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + c[i]
15:     vectorAdd<<<numBlocks,blockSize>>>(a,b,c,n)
16:     vectorAdd<<<numBlocks,blockSize>>>(a,b,c,n)
17:     ...
18: end procedure
```

Fig. 1. Pseudo code for experiment 1

2. Unified memory and implicit data migration

통합 메모리는 CPU 메모리와 GPU 메모리 간의 데이터 통신을 개발자에게 투명하게 내재적으로 수행하는 소프트웨어 런타임 환경으

로 개발자에게 CPU와 GPU 메모리가 통합된 하나의 메모리로 보이도록 해준다. 통합 메모리를 이용한 CUDA 프로그래밍에서 데이터 통신은 다음과 같이 이루어진다. 개발자는 cudaMallocManaged()라는 CUDA API를 이용하여 데이터를 할당하고 CPU에서 수행되는 함수와 GPU에서 수행되는 커널에서 이 데이터에 접근한다. 그러면 CPU 또는 GPU에서 자신의 메모리에 없는 데이터에 접근했을 때 페이지 폴트가 발생하고, 페이지 폴트 핸들러에서 통합 메모리 소프트웨어 런타임 환경을 통해 페이지 단위로 데이터 통신을 수행한다.

3. Hiding data migration latency

CUDA에서는 CPU 메모리와 GPU 메모리 사이의 데이터 통신 시간 동안 GPU에서 커널을 수행할 수 있도록 해주는 스트림이라는 메커니즘을 제공한다. 하나의 스트림에서 수행되는 데이터 통신과 커널들은 연속적으로 수행된다. 하지만 서로 다른 스트림에서 수행되는 데이터 통신과 커널들은 동시에 수행될 수 있다. 개발자가 데이터 통신과 커널들을 데이터 의존성을 고려하여 서로 다른 스트림에 지정하면, GPU에서 데이터 통신과 커널들을 스케줄링하여 가용 자원이 있을 때 동시에 수행한다.

Program 2: Multi stream

```
1:  kernel vecAdd(int * a, int * b, int * c, int n)
2:      i = blockIdx.x * blockDim.x + threadIdx.x
3:      if(i < n)
            c[i] = a[i] + b[i]
4:  end kernel
5:
6:  procedure MAIN
7:      ...
8:      vectorAdd<<<numBlocks,blockSize, stream0>>>(a,a,r1,n);
9:      vectorAdd<<<numBlocks,blockSize, stream1>>>(b,b,r2,n);
10:     vectorAdd<<<numBlocks,blockSize, stream2>>>(a,b,r3,n);
11:     for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + r1[i];
12:     for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + r2[i];
13:     for(int i = 1; i < n; i++)
            sum[i] = sum[i - 1] + r3[i];
14:     ...
15: end procedure
```

Fig. 2. Pseudo code for experiment 2

III. An Evaluation of the Data Migration in Unified Memory

우리는 통합 메모리를 사용할 시 CPU 메모리와 GPU 메모리 사이에서 데이터 통신이 이루어지는 조건과 GPU 응용의 수행시간에 이 데이터 통신이 끼치는 영향을 분석하기 위해 두 가지 실험을 수행한다. 이 장에서는 그 실험과 결과를 설명한다.

1. Experimental setup

우리는 Ubuntu 17.10과 CUDA 9.2에서 동작하는 GTX 1080 Ti GPU에서 두 가지 실험을 수행한다. 첫 번째 실험에서는 데이터 통신과 커널들이 연속적으로 수행되는 상황에서 CPU에서 수행되는 함수와 GPU에서 수행되는 커널이 접근하는 데이터를 바꾸면서 데이터 통신과 커널의 수행시간을 측정한다. 두 번째 실험에서는 데이터 통신과 커널들을 여러 스트림에 할당하여 동시에 수행될 수 있도록 한 상황에서 위와 같은 것을 측정한다.

우리는 이 실험들을 위해 CUDA 프로그램들[1]을 작성하여 NVIDIA Visual Profiler로 프로파일링한다. 우리는 첫 번째 실험을 위해서 Fig. 1의 의사코드를 통합 메모리를 사용한 경우와 전통적인 CUDA 프로그래밍 모델을 사용한 경우의 두 가지 프로그램으로 작성한다. 두 번째 실험을 위해서는 Fig. 2의 의사코드를 마찬가지로 두 가지 프로그램으로 작성한다.

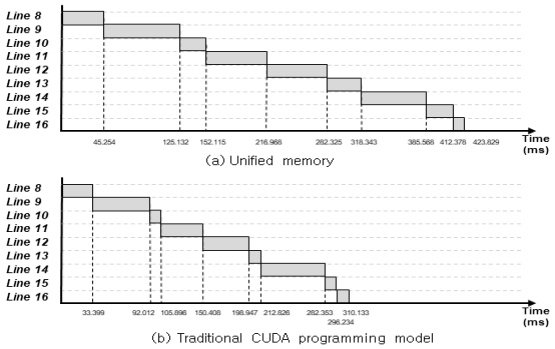

Fig. 3. Execution time of programs per line written in pseudo code Fig.1

2. Single stream

Fig. 3은 Fig. 1의 의사코드를 작성한 프로그램을 수행한 결과다. 8번째 줄의 의사코드가 시작된 시점을 0으로 하여 16번째 줄의 의사코드가 종료된 시점까지 코드 라인별 수행시간을 측정하였다.

우리는 이 실험 결과로부터 연속적으로 데이터 통신과 커널들이 수행되는 상황에서 통합 메모리의 오버헤드는 추가적인 데이터 통신 때문에 발생한다는 것을 알아냈다. 이는 두 가지 관측에 근거한다. 첫째, 통합 메모리를 사용하였을 때 CPU 또는 GPU가 데이터에 접근할 때마다 이 데이터는 CPU 또는 GPU 메모리로 옮겨지고 옮겨진 데이터는 제거된다. 따라서 재사용할 데이터도 제거되어 추가적인 데이터 통신이 발생한다. (a)의 라인 8, 10, 13, 15, 16의 수행시간을 보면, 각 라인의 수행 전에 CPU에서 수행되는 함수에서 접근한 데이터에 종속적이다.

둘째, 똑같은 데이터를 옮긴다면 통합 메모리와 전통적 CUDA 프로그래밍 모델은 수행시간에 차이가 없다. (a)의 라인 13과 (b)의 라인 8은 수행시간은 비슷하다. (a)의 라인 14와 (b)의 라인 14도 마찬가지다.

1) 이 프로그램들의 소스 코드는 https://github.com/Hozzu/KSCI2018 에서 받을 수 있다.

3. Multi stream

Fig. 4는 Fig. 2의 의사코드를 작성한 프로그램을 수행한 결과다. 8번째 줄의 의사코드가 시작된 시점을 0으로 하여 13번째 줄의 의사코드가 종료된 시점까지 코드 라인별 수행시간을 측정하였다.

우리는 이 실험 결과로부터 통합 메모리를 사용할 때 서로 다른 스트림에 할당된 데이터 통신과 커널들이 거의 동시에 수행되지 못한다는 것을 알아냈다. 라인 8, 9, 10은 데이터 의존성이 없는데도 불구하고 (b)와 다르게 (a)에서 거의 동시에 수행되지 않는다.

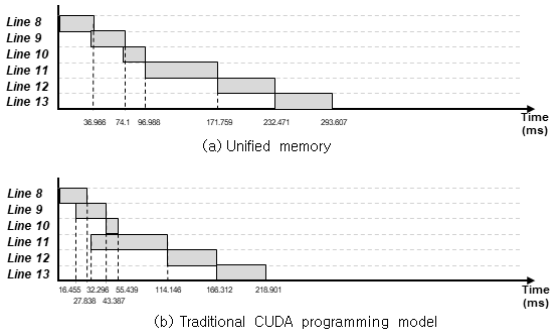

Fig. 4. Execution time of programs per line written in pseudo code Fig. 2

IV. Conclusion

우리는 NVIDIA 파스칼 GPU를 대상으로 하여, 통합 메모리를 사용할 시 CPU 메모리와 GPU 메모리 사이에서 데이터 통신이 이루어지는 조건과 GPU 응용의 수행시간에 이 데이터 통신이 끼치는 영향을 실험을 통해 분석하였다. 실험 결과 통합 메모리의 오버헤드는 두 가지 원인 때문에 발생한다. 첫째, 통합 메모리를 사용하면 CPU 또는 GPU가 데이터에 접근할 때마다 이 데이터는 CPU 또는 GPU 메모리로 옮겨지고 옮겨진 데이터는 제거된다. 따라서 재사용할 데이터도 제거되어 추가적인 데이터 통신이 발생하고, 이 데이터 통신의 지연시간은 GPU 응용의 수행시간에 더해진다. 둘째, 통합 메모리를 사용하면 데이터 통신과 커널들이 서로 다른 스트림에 할당되어도 동시에 수행되지 못한다. 따라서 GPU 응용의 수행시간은 동시에 수행되던 데이터 통신과 커널의 수행시간만큼 증가한다.

REFERENCES

[1] Agarwal, Neha, et al. "Page placement strategies for GPUs within heterogeneous memory systems." ACM SIGPLAN Notices 50.4, 2015, 607-618.

[2] Agarwal, Neha, et al. "Unlocking bandwidth for GPUs in CC-NUMA systems." HPCA, 2015.

[3] Rhu, Minsoo, et al. "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design." MICRO, 2016.

[4] Zheng, Tianhao, et al. "Towards high performance paged memory for GPUs." HPCA, 2016.

[5] Vogel, Pirmin, et al. "Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs." ACM TECS 16.5s, 2017, 154.