NVIDIA Tegra와 Tesla GPU에서의 CPU-GPU 데이터 전송성능 연구

권오경*, 구기범*
*한국과학기술정보연구원 슈퍼컴퓨팅본부
okkwon@kisti.re.kr, gibeom.gu@kisti.re.kr

A Performance Study on CPU-GPU Data Transfers of NVIDIA Tegra and Tesla GPUs

Oh-Kyoung Kwon*, Gibeom Gu*
*National Supercomputing Center,
Korea Institute of Science and Technology Information

요 약

최근 HPC, 인공지능에서 GPU 성능이 향상되면서 사용이 보편화되고 있지만 GPU 프로그래밍은 난이도 측면에서 여전히 큰 장애물이다. 특히 호스트(host) 메모리와 GPU 메모리를 따로 관리해야 하는 어려움 때문에 편의성과 성능 측면에서 연구가 활발히 진행되고 있으며, 다양한 CPU-GPU 메모리 전송 프로그래밍 방법들이 제시되고 있다. 본 연구는 NVIDIA Tegra 장치들과 NVIDIA SMX 기반 V100 GPU 카드에서 CPU-GPU 데이터 전송 기법별로 성능비교를 하고자 한다. 특히 NVIDIA Tegra 장치는 CPU와 GPU 통합메모리를 제공하고 있어서 CPU-GPU 메모리 전송방법의 관점에서 기존 GPU 장치와 다른 성능 특징을 보여준다. 성능비교를 위한 실험 워크로드는 HPC 응용프로그램에서 빈번하게 사용하는 2차원 행렬 전치 예제를 사용하였다. 실험을 통해 각 GPU 장치별로 CPU-GPU 메모리 전송 방법에 따른 GPU 커널 성능차이, 페이지 잠긴 메모리와 페이지 가능 메모리의 전송 성능차이, 마지막으로 전체 성능비교를 하였다.

1. 서론

PCI 익스프레스(PCIe) 장치 등에 장착된 GPU 카드는 CPU 및 호스트 메모리와 물리적으로 분리가 되어 있다. 그렇기 때문에 GPU 카드에서 계산을 수행하기 전에 호스트 메모리에서 GPU 메모리로 데이터를 복사하고, 수행이 끝나면 다시 호스트 메모리로 결과를 복사해서 그 내용을 확인한다. 이 과정에서 GPU와 호스트 메모리를 모두 관리해야 하는 불편함과 PCIe 등을 통한 CPU-GPU 이동 부하로 인해 GPU 메모리와 호스트 메모리를 통합 관리하기 위한 연구가 많이 되고 있다[1]. 대표적으로 소프트웨어 수준에서 GPU와 호스트 메모리간 제로카피(zero copy)와 통합메모리(unified memory) 기술이 소개되어 있다. 두 기술은 호스트와 GPU 메모리를 연결(mapping)해서 명시적으로 메모리 복사 명령을 하지 않아도 GPU 커널에서 호스트 메모리를 바로 사용할 수 있게 한다. 사용성의 관점에서 보면 편리한 기술이지만 PCIe 상에서 이동 부하가 여전히 존재하므로 성능 면에서 큰 장점을 발휘하기 어렵다.

한편 최근 Apple M1, NVIDIA Tegra 등 CPU와 GPU 및 통합메모리 등을 하나의 대형 실리콘 패키지에 묶는 SoC(System on a Chip) 제품들이 많이 나오고 있다. 해당 제품들은 GPU 계산 시 CPU와 물리적으로 동일한 메모리를 사용하기 때문에 PCIe 장치 기반 GPU 카드보다 메모리 전송 성능이 개선된다. 그리고 위에서 언급한 제로카피 및 통합메모리 기술을 사용하는 데에 적합한 구조를 갖고 있기도 하다.

본 연구는 위에서 소개한 여러 GPU 장치들을 CPU-GPU 데이터 전송 기법별로 성능을 비교한다. 이때 워크로드는 HPC 응용프로그램에서 일반적인 유형인 행렬 전치를 사용한다. R. S. Santos 연구에서 PCIe 기반 GPU 카드에 대해 세 가지 GPU 메모리 프로그래밍 기법을 비교한 적이 있지만, NVIDIA Tegra 장치와 같은 CPU, GPU 통합 칩에서의 성능 분석은 본 연구가 처음이다[1].

2. CPU-GPU 데이터 전송 프로그래밍 기법 비교

이 장에서는 NVIDIA GPU 환경에서 네 가지 형태의 CPU-GPU 데이터 전송 프로그래밍 기법을 설명한다. 첫 번째는 GPU 환경에서 가장 일반적인 데이터 전송 방법(**페이지 가능 메모리 전송**)으로, CPU와 GPU 메모리를 각각 독립적으로 할당한 후 데이터

를 전송한다. CPU는 malloc 함수로 호스트의 페이지 가능(pageable) 메모리 영역을 할당하고, GPU는 그래픽 메모리 영역을 cudaMalloc 함수로 할당한다. 그리고 CPU와 GPU 사이의 데이터 전송에는 cudaMemcpy 함수 등을 사용한다.

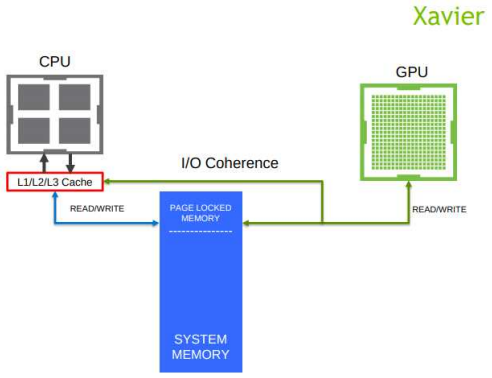
두 번째는 CPU와 GPU 메모리를 독립적으로 할당하되, 호스트 메모리를 페이지 잠긴(pinned 혹은 page locked) 메모리로 할당하여 전송하는 방법(**페이지 잠긴 메모리 전송**)이다. 호스트 메모리 할당에는 cudaMallocHost 함수를 사용하는데, malloc 함수로 할당되는 페이지 가능(pageable) 메모리 영역보다 더 많은 대역폭을 제공받을 수 있어서 데이터를 전송할 때 성능향상을 기대할 수 있다. 하지만 과도하게 많은 페이지 잠긴 메모리 영역을 할당하면 페이지 가능한 영역이 급격하게 줄어들어 전체 시스템 성능에 영향을 미칠 수 있다.

세 번째 방법은 **제로카피**를 활용한 CPU와 GPU 메모리간 데이터 전송이다. 이 방법은 CPU와 GPU가 공통으로 cudaHostAlloc 함수를 사용해서 메모리를 한번만 할당하면 명시적인 이동 명령 없이 각각의 장치가 바로 메모리를 접근할 수 있다. 이 때 cudaHostAlloc 함수는 페이지 잠긴 메모리 영역을 할당한다.

마지막으로 **통합메모리** 기술을 활용한 CPU와 GPU 메모리간 데이터 전송 방법이 있다. 이 기술도 제로카피와 유사하게 데이터에 대한 하나의 연결고리를 제공하므로 별도의 이동 명령을 수행하지 않아도 된다. 하지만 제로카피는 cudaHostAlloc 함수로 페이지 잠긴 영역만 할당하기 때문에 프로그램 성능이 페이지 잠긴 영역이 어느 정도 남아 있는지에 따라 달라질 수 있는 반면, 통합메모리는 cudaMallocManaged 함수를 사용하며 데이터 접근 속도가 빨라질 수 있게 메모리와 실행 영역을 분리하는데 초점이 맞춰져 있다[2].

3. I/O 캐쉬 일관성

NVIDIA Tegra는 CPU와 GPU가 물리적으로 메모리를 공유하지만, TX2와 Xavier는 CPU와 GPU 메모리 캐시(cache) 일관성(coherency) 방법이 달라서 제로카피 성능이 달라질 수 있다. TX2는 CPU와 GPU 메모리 캐시에 대해 하드웨어 일관성이 보장되지 않기 때문에 GPU 커널에서 메모리를 접근할 때 미스(miss)가 발생하면 메인 메모리에서 데이터를 가져와야 한다. 반면 Xavier는 (그림 1)처럼 하드웨어 I/O 일관성이 보장되므로[3][4], 페이지 잠긴 메모리



(그림 1) 하드웨어 I/O 일관성 개념도 [4]

영역과 CPU 캐시(L1, L2, L3 캐시)에 대한 캐시 일관성을 수행하여 메인 메모리 접근을 최소화할 수 있다. 즉, GPU 커널에서 제로카피를 활용하여 메모리에 접근할 때 TX2보다 성능향상을 기대할 수 있다.

4. 실험 워크로드

실험 워크로드는 HPC 응용프로그램에서 빈번하게 사용하는 실수(float) 타입의 2차원 행렬 전치(transpose) 계산을 사용한다. 행렬전치를 위한 GPU 커널은 GPU 전역(global) 메모리 접근 수에 따른 성능 차이를 확인하기 위해서 (그림 2)처럼 두 가지를 구현하였다. 첫 번째 커널은 GPU 전역 메모리를 바로 접근하는 Naive (matrixTransposeNaive) 커널, 두 번째 커널은 GPU 전역 메모리 접근을 줄이기 위해 GPU 공유(shared) 메모리 최적화를 구현한 SM(matrixTransposeSM) 커널이다.

각 GPU 커널을 수행하기 위한 워크로드는 2장에서 소개한 4가지 CPU-GPU 데이터 전송 기법과 GPU 커널 2개에 대한 총 8가지 조합을 (그림 4)의 순서로 실행한다. 1. GPU 메모리 내 입력 및 출력 데이터를 초기화한다. 이 때 페이지 가능 메모리 전송과 페이지 잠긴 메모리 전송 기법은 cudaMemset 함수를 이용하고 나머지는 할당된 변수에 데이터를 직접 쓴다(write). 2. 호스트 메모리에서 GPU 메모리로 행렬을 복사한다. 이 단계는 페이지 가능 전송과 페이지 잠긴 메모리 전송 방법에만 해당된다. 3. GPU 커널을 수행한다. 제로카피와 통합메모리 기법은 수행할 필요가 없다. 4. 커널 실행결과를 호스트 메모리로 복사한다. 2번과 마찬가지로 페이지 가능 전송과 페이지 잠긴 메모리 전송 방법에서만 이 단계가 실행된다. 5. 페이지 잠긴 메모리에서 페이지 가능 메모리로 복사한다. 이 단계는 TX2에서만 필요하다. 다음 단계인 CPU에서 코드 수행 시 페이지 잠긴

```
__global__ void matrixTransposeNaive(float* a, float* b)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index_in  = j * sizeX + i;
    int index_out = i * sizeY + j
    b[index_out] = a[index_in];
}
__global__ void matrixTransposeSM(float*  a, float* b) {
    __shared__ float mat[BLOCK_SIZE_Y][BLOCK_SIZE_X];
    int bx = blockIdx.x * BLOCK_SIZE_X;
    int by = blockIdx.y * BLOCK_SIZE_Y;
    int i  = bx + threadIdx.x; int j  = by + threadIdx.y;

    int ti = by + threadIdx.x; int tj = bx + threadIdx.y;

    if(i < sizeX && j < sizeY)
      mat[threadIdx.y][threadIdx.x] = a[j * sizeX + i];
    __syncthreads();
    if(ti < sizeY && tj < sizeX)
      b[tj * sizeY + ti] = mat[threadIdx.x][threadIdx.y];
}
```

(그림 2) 행렬 전치 GPU 커널 코드

```
1. GPU 메모리 초기화 (pre-process)
2. 호스트(CPU) 메모리에서 GPU 메모리로 복사 (HtoD copy)
3. GPU 커널 실행 (kernel)
4. GPU 메모리에서 호스트 메모리로 복사 (DtoH copy)
5. 페이지 잠긴 메모리에서 페이지 가능 메모리로 복사
   (memcpy)
6. 결과 검증 (post-process)
```

(그림 3) 워크로드 수행 순서

메모리를 읽을(read)때 성능 저하가 발생하여 페이지 가능 메모리로 복사하는 것으로 코드를 수정하였다. 6. 마지막으로 CPU에서 결과를 검증한다. 3번처럼 제로카피와 통합메모리 기법은 수행하지 않는다.

5. 실험 결과 분석

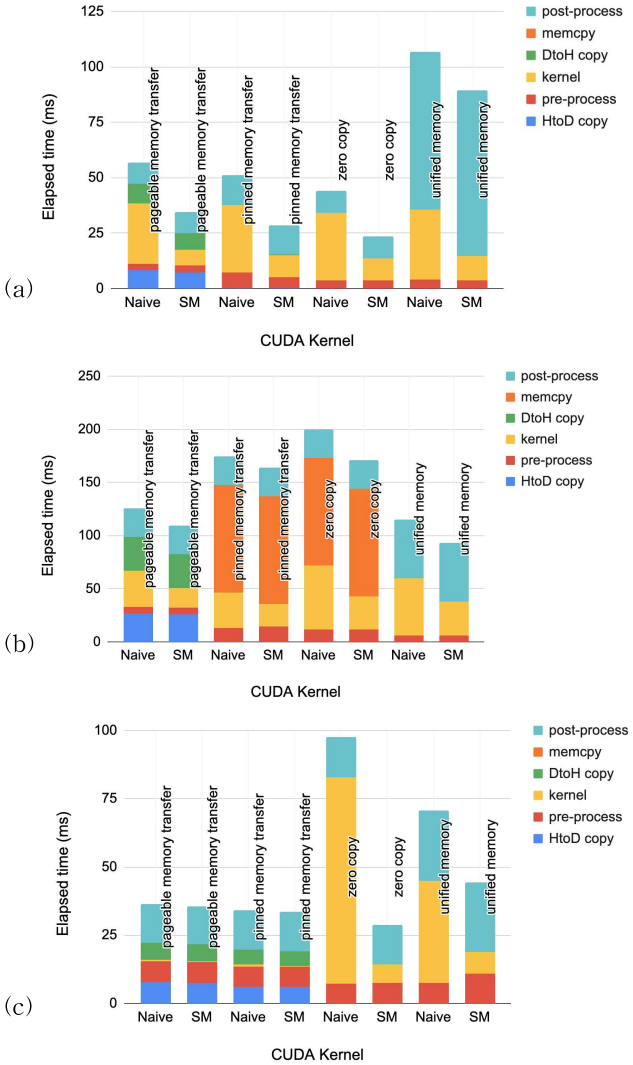
4장에서 소개한 총 8가지 워크로드 조합을 (표1)에 나열한 3 종류의 GPU 장치에서 수행하였다. 여기서 V100-SMX2는 PCIe가 아닌 SMX2 장치 기반에서 데이터 전송이 수행되는데 단일 GPU에서는 PCIe 대비 CPU-GPU 전송 성능 차이는 없다. (그림 4)에 표출한 결과를 다음 세 가지 성능비교 분석을 하고자한다: 1. CPU-GPU 메모리 전송 방법에 따른 GPU 커널 성능 차이. 2. 페이지 잠긴 메모리와 페이지 가능 메모리의 전송성능 차이. 3. 전체(overall) 성능비교.

먼저 GPU 장치별로 데이터 전송 기법에 따른 GPU 커널 성능차이를 살펴보자(그림 4 노란색). Xavier는 기법별로 커널 간 성능차이가 거의 없는 반면, TX2와 V100-SMX2에서는 성능차이를 확인할

<표 1> 실험에 사용한 GPU 장치 및 CUDA 버전

GPU 장치	메모리 사양 (크기, 대역폭)	CUDA	아키텍처
TeslaV100 SMX2	HBM2 (16GB, 900.1GB/s)	11.0	Volta
TX2	LPDDR4 (8GB, 59.7GB/s)	10.2	Pascal
Xavier	LPDDR4x (32GB, 136.5GB/s)	10.2	Volta

수 있다. 제로카피와 통합메모리 기법은 Xavier의 경우 I/O 일관성에 대한 하드웨어 지원으로 인해 GPU 커널 실행 시 메모리 접근에 대한 부하가 거의 발생하지 않고, TX2와 V100-SMX2는 GPU 커널 실행 시 캐시 내 페이지 폴트(page fault)가 계속 발생하여 GPU 캐시쪽으로 메모리를 계속 가져오게 되어 성능 저하가 발생한다. 특히 TX2가 V100-SMX2 대비 전송 성능차이가 확연히 드러나는데 TX2는 통합메모리



(그림 4) 각 GPU 장치별로 CPU-GPU 데이터 전송 방법에 따른 성능 비교 (행렬크기 4096x4096): (a) Xavier (b) TX2 (c) V100-SMX2

내부 이동인 반면 V100-SMX2는 SMX2를 통한 전송이기 때문이다. 또한 V100-SMX2 경우 Naive 커널에서 제로카피 사용 시 커널 실행이 통합메모리 기법보다 확연하게 느림을 확인할 수 있는데, 그 이유는 GPU 전역 메모리 접근시 제로카피가 페이지 폴트 발생시 최적화가 안되어 있기 때문이다. NVIDIA는 통합메모리 기법에서 Pascal 이후 아키텍처는 페이지 폴트에 대한 하드웨어 지원이 이루어지고 있다[5]. 반면 SM 커널은 전역 메모리 사용이 많이 줄어서, 통합메모리 기법 최적화로 인한 성능 개선이 크게 없다. 한편 TX2는 V100-SMX2와 다르게 커널 실행 시 제로카피와 통합메모리 기법 관련 커널 성능차이가 크게 없었다.

다음으로 페이지 잠긴 메모리와 페이지 가능 메모리의 전송 성능차이를 살펴보자. 페이지 잠긴 메모리 방법을 사용할 때 호스트에서 GPU 메모리로의 데이터 전송은 Xavier, TX2, V100-SMX2는 각각 약 48배, 164배, 1.26배, GPU 메모리에서 호스트로의 데이터 전송은 각각 72배, 242배, 1.14배 향상되었다. 페이지 잠긴 메모리 방법을 사용할 경우, Xavier와 TX2에서 사용자가 명시적으로 데이터를 전송할 때의 속도는 확실히 빠르지만, CPU에서 프로그램(pre-process와 post-process)을 실행하거나 GPU 커널에서 메모리를 접근할 때 더 느려지는 것을 확인할 수 있다.

마지막으로 전체 성능비교 분석을 해보자. Xavier는 데이터 전송 기법 중 제로카피가 가장 빠른 성능을, 통합메모리가 가장 느린 성능을 보여준다. 제로카피의 Naive, SM 커널은 페이지 가능 메모리 전송 방법의 Naive, SM 커널에 비해 각각 1.28배, 1.47배 빠른 속도를 보여준다. 통합메모리 방법은 CPU로 결과확인(post-processing, 그림 4 청록색)을 할 때 성능 저하가 많이 발생하는데, 이는 소프트웨어 최적화 문제로 보인다. TX2는 페이지 가능 메모리 전송과 통합메모리 방법이 가장 빠른 성능을 보여주며, 제로카피 방법은 페이지 폴트를 처리하는 루틴의 최적화가 문제인 것으로 보인다. 또한 페이지 잠긴 메모리 전송과 제로카피 방법은 앞서 언급한 것처럼 페이지 가능 메모리 영역으로 복사(memcpy, 그림 4 주황색)하는 과정이 추가되어 성능저하가 많이 된다. V100-SMX2는 Naive 커널의 경우에는 페이지 잠긴 메모리 전송이, SM 커널일 때에는 제로카피 방법이 가장 성능이 좋다. Naive 커널은 페이지 잠긴 메모리 전송이 페이지 가능 메모리 전송 방법 대비 1.07배, SM 커널은 제로카피가 페이지 가능 메모리 전송 방법 대비 1.24배 빠른 속도를 보여준다. SM 커널은 Naive 커널에 비해 전역 메모리 접근 횟수가 적기 때문에 커널 실행 시 부하가 많이 줄었다.

6. 결론

본 연구는 V100-SMX2 GPU 카드, NVIDIA Tegra Xavier, TX2 장치에서 4가지 CPU-GPU 데이터 전송 기법별로 2차원 행렬 전치 예제를 사용해서 성능비교를 하였다. 먼저, Xavier는 I/O 일관성 지원을 통해 데이터 전송 방법과 상관없이 GPU 커널 성능차이가 거의 없었고, TX2와 V100-SMX2는 제로카피와 통합메모리 기법에서 커널 간 성능차이가 발생하였다. 또한 페이지 잠긴 메모리를 사용하면 페이지 가능 메모리 보다 CPU-GPU 사이의 빠른 데이터 전송속도를 확인할 수 있었다. 마지막으로 Xavier, TX2, V100-SMX2는 제로카피, 페이지 가능 메모리 전송, 페이지 잠근 메모리 전송 방법이 가장 빨랐다.

이 논문은 대한민국 정부(과학기술정보통신부)의 재원으로 한국연구재단 슈퍼컴퓨터개발선도사업의 지원을 받아 수행된 연구임 (과제번호 : 2020M3H6A1084857)

참고문헌

[1] R. S. Santos et al. Performance Evaluation of Data Migration Methods Between the Host and the Device in CUDA-Based Programming, Information Technology: New Generations pp 689-700, 2016

[2] CUDA C++ Programming Guide, “https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html”

[3] CUDA for Tegra, “https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html“

[4] Anshuman Bhat, CUDA on Xavier, GTC 2018, “http://on-demand.gputechconf.com/gtc/2018/presentation/s8868-cuda-on-xavier-what-is-new.pdf”

[5] Nikolay Sakharnykh, EVERYTHING YOU NEED TOKNOW ABOUT UNIFIED MEMORY, GTC 2018, “https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf”