딥 러닝 연산을 위한 GPU/CPU 성능 분석*

최진서○ 강동현
창원대학교 컴퓨터공학과
jinseo@gs.cwnu.ac.kr, donghyun@changwon.ac.kr

GPU/CPU Performance Analysis for Deep Learning Computation

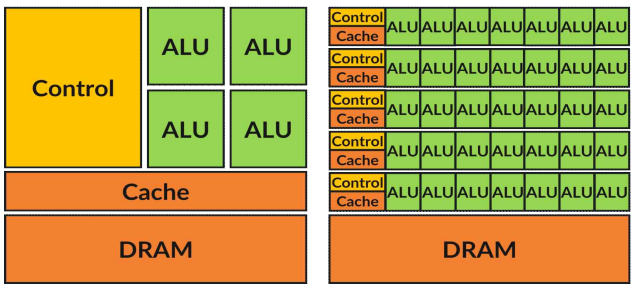
Jinseo Choi○ Donghyun Kang
Dept. of Computer Engineering, Changwon National University

요 약

최근, 딥 러닝 모델의 높은 정확도를 달성하기 위해 학습모델의 규모가 점차 증가하고 있다. 이에 따라 학습모델의 훈련에 요구되는 연산량이 증가하면서 연산 처리시간 또한 증가하는 문제가 존재한다. 이를 해결하기 위해 일반적으로 병렬처리에 특화된 GPU를 사용하여 학습모델의 훈련을 가속한다. 이에 본 논문에서는 CPU와 GPU의 성능 차이를 분석하기 위해 행렬 곱 연산 수행시간을 비교하는 실험을 진행했다. 이를 위해 먼저 범용 GPU 프로그래밍 모델인 CUDA를 분석한 후, CPU 연산 최적화 라이브러리를 비교군으로 설정하여 실험을 설계하였다. 실험 결과, 동일한 행렬 곱 연산에서 GPU를 사용할 경우, CPU 대비 최대 485배 높은 처리성능이 향상되는 결과를 확인할 수 있었다.

1. 서 론

최근 딥 러닝(Deep learning) 기술이 주목받으면서, 높은 정확도를 달성하기 위해 학습모델의 규모가 점차 증가하고 있다[1]. 이에 따라 증가하는 연산량으로 인해 일반적으로 병렬처리에 특화된 GPU(Graphic Processing Unit)를 사용하여 학습모델의 훈련을 가속한다[2]. GPU는 그래픽 처리에 특화된 특성상 여러 작업을 동시에 처리하기 위해 다수의 코어로 구성된다. 그림 1은 CPU와 GPU의 하드웨어 구조의 차이를 보여준다[3]. (a)에서 CPU는 소수의 고성능 멀티코어(Multi-core)가 복잡한 연산을 수행할 수 있도록 구성되어 있지만, (b)의 GPU는 매니코어(Many-core) 그룹으로 구성되어 연산을 병렬적으로 처리할 수 있다. 이러한 특성으로 인해, 더 높은 처리성능을 보인다.



(a) CPU (b) GPU

그림 1. CPU와 GPU의 구조

따라서, 이러한 GPU의 연산 성능을 범용 계산에 사용하기 위한 GPGPU(General Purpose Graphic Processing Unit)를 도입하여 기존에 CPU가 수행하는 태스크(Task)를 GPGPU가 수행하여 성능을 향상하는 연구가 활발하게 진행되고 있다[4-6]. 대표적인 GPGPU 프로그래밍 모델로는 Nvidia 사의 CUDA와 Khronos 그룹의 OpenCL이 존재한다[7,8].

본 논문에서는 실험을 통해 CPU와 GPU의 병렬처리 성능을 비교한다. 또한 CPU 연산 최적화 기술이 적용된 라이브러리를 살펴보고 GPU의 연산 처리성능을 분석한다. 이를 통해 연산량이 증가할수록 GPGPU를 사용할 때, 최대 485배 더 높은 처리성능을 보여주는 것을 확인할 수 있었다.

본 논문의 구성은 다음과 같다. 2장에서는 연구 배경과 CPU 연산 최적화 기술에 대해 살펴보고 3장에서는 실험 환경과 CPU와 GPU의 성능 비교하는 실험을 설계한다. 이후 4장에서 결과를 분석하고 5장에서 결론을 맺는다.

2. 연구배경

GPGPU 프로그래밍 모델은 CPU와 GPU를 모두 사용하는 이기종 모델(Heterogeneous Model)이다[3]. 이는 크게 CPU와 시스템 메모리를 포함하여 호스트(Host)로 정의하고 GPU와 메모리를 디바이스(Device)로 정의한다.

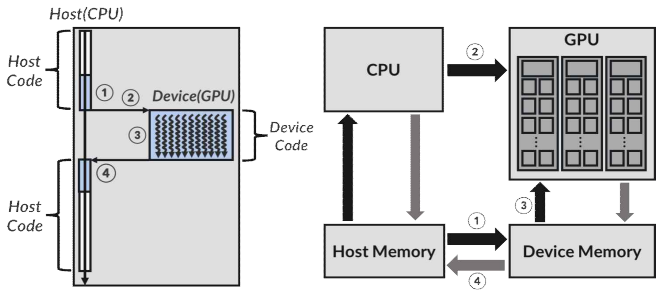


그림 2. GPGPU의 동작 흐름

그림 2는 CUDA 프로그래밍 모델의 동작 흐름을 보여준다. 프로그램 구성은 크게 CPU에서 동작하는 호스트 코

* 이 논문은 2021년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. NRF-2021R1I1A3047006).

드와 GPU에서 동작하는 디바이스 코드로 구분된다. CPU와 GPU는 개별적인 메모리 공간을 사용한다. 따라서 먼저 데이터를 GPU에서 처리하기 위해 디바이스 메모리 공간을 할당한 후, 호스트 메모리에서 디바이스 메모리로 데이터를 복사한다(①). 이후, 디바이스 코드에 의해 CPU에서 GPU로 사전에 정의된 연산을 수행하도록 요청한다(②). 이 과정에서 GPU에서 생성될 쓰레드의 개수를 지정하여 전달한다. GPU는 디바이스 메모리에 할당된 데이터를 로드하여 요청받은 연산을 수행한 후 결괏값을 반환한다(③). 마지막으로 반환된 데이터는 호스트 메모리에 복사된다(④). 이러한 GPGPU의 처리 과정에서 데이터를 복사하고 이동하는 시간이 병렬 연산시간보다 길어진다면, 상대적으로 데이터 이동이 적은 CPU에서 처리하는 것이 더 효율적일 수 있다.

2.1 CUDA 실행구조

CUDA는 Nvidia사에서 제공하는 GPGPU 프로그래밍 모델로 CUDAToolkit을 통해 고수준 프로그래밍 API를 지원한다. CUDA는 하나의 명령으로 여러 개의 쓰레드를 생성하여 병렬처리 하는 SIMT(Single Instruction, Multiple Thread) 구조로 구성되어있다. 이를 위해 CUDA 루틴은 커널(Kernel)이라는 단위로 동작한다. 커널은 다수의 쓰레드 블록(Thread Block)으로, GPU에서 SM(Streaming Multiprocessor)에 할당된다. SM은 쓰레드 모임인 워프(Warp)로 구성되어 있다. 워프는 GPU에서 작업 최소단위로 사용되며, 보통 32개의 쓰레드가 1 워프로 지정되어 있다. 사용자는 커널마다 병렬 처리할 쓰레드 블록과 각 블록에 할당될 쓰레드 수를 지정할 수 있다. 커널이 실행되면 각 쓰레드가 내부 동작을 동시에 처리한다. 이를 통해 고차원 행렬 곱셈과 같은 높은 연산량을 요구하는 선형대수 연산에서 CPU보다 높은 처리성능을 보인다[3].

3. 실험

본 장에서는 CPU와 GPU의 병렬처리 성능 비교를 위한 실험환경과 실험 설계에 관해서 설명한다. 또한 GPU와 성능 비교를 위한 CPU 비교군의 종류와 특성에 관해 설명한다.

3.1 실험 환경

표 1은 실험환경을 나타낸다. CUDA는 11.7버전을 사용하였다. CPU는 물리 코어의 수가 10개인 프로세서 2개를 사용하였으며 GPU는 물리 코어가 2560개로 구성된 GPU를 사용하였다.

표 1 실험환경

운영체제	Ubuntu 20.04 LTS
CPU	Intel® Xeon® Gold 5215 @2.50GHz ×2
# of CPU Cores	10+10
GPU	Nvidia® Tesla® T4 16GB
# of GPU Cores	2560
메모리(RAM)	64GB

3.2 실험 설계

CPU와 GPU의 비교실험은 성능 측정에 많이 사용되는 행렬 곱 연산 수행시간을 측정하여 비교한다. GPU의 경우, CUDA를 사용하여 GPU 쓰레드가 각 행렬의 원소를 병렬적으로 계산하는 GPU 쓰레드 구조에 최적화된 행렬 곱 연산을 구현했다. 커널에 할당될 쓰레드는 한 개의 쓰레드 블록당 1024개의 쓰레드를 배치하였으며 입력되는 행렬의 크기에 따라 커널에 할당되는 쓰레드 블록의 개수를 행렬 곱의 연산횟수만큼 지정하였다.

비교군의 CPU의 경우, Pthread를 사용한 행렬 곱 연산을 구현하였으며, 성능이 가장 높게 측정된 64개 쓰레드를 사용한 수행시간으로 비교하였다. 또한 최적화된 선형대수 연산 라이브러리를 사용한 행렬 곱 연산을 사용하였다. MKL(Math Kernel Library)는 Intel 사에서 개발한 수치 연산 라이브러리로, BLAS를 지원한다[9]. BLAS(Basic Linear Algebra Subprograms)는 선형대수 연산을 수행하는 로우레벨(Low-level) 루틴을 명세한 규약이다[10]. 이는 연산성능 최적화를 위해 개발된 것으로 많은 수학 연산 라이브러리가 BLAS를 지원한다. 또한 MKL은 Intel 프로세서에 특화된 자체적인 최적화와 자동 병렬화를 제공한다[9].

ATLAS(Automatically Tuned Linear Algebra Software)는 C와 Fortran에서 BLAS API를 제공하는 수치 연산 라이브러리다[11]. 이와 같이 GPU 행렬 곱과 Pthread와 BLAS를 지원하는 라이브러리를 사용한 행렬 곱 수행시간을 측정하여 비교・분석한다.

4. 실험결과

실험은 크기가 각각 4096, 8192로 구성된 정방행렬 곱셈 결과를 측정하였다. 또한 행렬의 각 원소에 float 32형 난수를 생성하여 부동소수점 연산을 수행하도록 구성했다. 그림 3과 그림 4는 각 행렬 곱 수행시간 비교 결과를 보여준다. 그림 3에서 GPU를 사용하는 CUDA로 구현한 연산의 수행시간이 가장 빠른 결과를 보여준다. 이는 행렬 곱 연산의 횟수에 비례하는 수천 개 이상의 쓰레드가 병렬적으로 동작함으로써 높은 성능을 보인다.

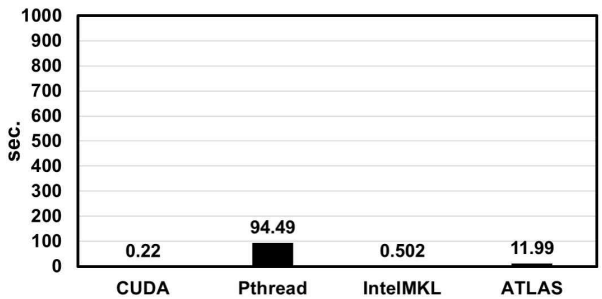


그림 3. 4096 행렬 곱 수행시간 비교

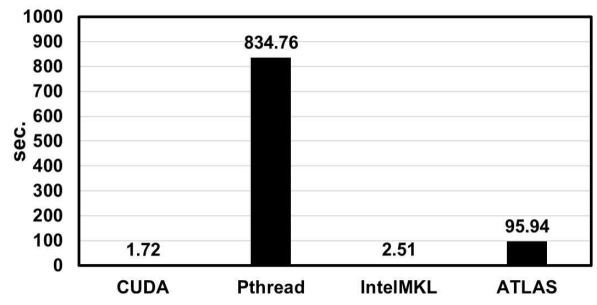


그림 4. 8192 행렬 곱 수행시간 비교

MKL로 구현한 연산의 경우, 두 번째로 높은 성능을 보이며 CUDA 대비 1.2배만큼 낮은 성능을 보였다. 이는 MKL의 프로세서 최적화 및 자동 병렬화에 따른 성능 향상 결과로 분석된다. 실제 성능 모니터링 시 19개의 쓰레드가 생성되어 병렬처리를 수행하는 것을 확인할 수 있었다.

ATLAS의 경우 CUDA 대비 54.5배, MKL 대비 23.8배만큼 낮은 성능을 보였다. 성능 모니터링 시 ATLAS는 1개의 쓰레드만 동작하는 것을 확인하였다. 이는 BLAS의 저수준 최적화에 따른 성능으로 분석된다.

반면, Pthread로 구현한 연산성능의 경우, 멀티쓰레드를 사용하여 병렬처리를 수행하지만, ATLAS 대비 7.8배 느린 가장 낮은 성능을 보였다. 이는 행렬 곱을 처리하기 위해 고수준에서 다수의 쓰레드가 CPU에 경쟁적으로 접근하여 반복문을 처리할 때 발생하는 오버헤드로 인한 결과로 분석된다.

그림 4의 경우 8192개의 행과 열로 구성된 정방행렬 곱 연산의 수행 결과를 보여준다. 이는 4096 행렬 곱 대비 연산량이 8배 증가한다. 따라서 ATLAS를 사용한 연산성능의 경우 기존 4096 행렬 연산 결과 대비 실행시간이 8배 증가한 결과를 보인다. Pthread의 경우 기존 대비 수행시간이 8.8배 증가하는 결과를 보였다. 이는 개별 쓰레드가 처리해야할 연산량이 증가하면서 물리 코어수보다 많은 쓰레드가 코어를 선점하는 경쟁상태가 지속됨에 따라 발생하는 지연시간에 따른 추가적인 오버헤드로 분석된다.

CUDA는 기존 대비 7.8배만큼 실행시간이 증가하였다. 이는 처리 해야 할 행렬 연산량에 비례하여 병렬 처리되는 쓰레드의 수가 증가한 것으로 분석된다. 또한 비교군 중 가장 높은 성능을 보였다. 특히, MKL의 경우 기존 대비 5배만큼 실행시간이 증가하였으며 CUDA 대비 1.4배만큼 낮은 성능을 보였다. 이는 비교군 중 가장 낮은 실행시간 증가율을 보였으나 CUDA와 성능 비교의 경우 기존 4192 행렬 대비 0.2배 더 낮은 성능 차이를 보였다. 성능 모니터링 시 병렬 처리되는 쓰레드의 수는 증가하지 않는 것을 확인할 수 있었다. 이러한 결과는 향후, MKL 프로세서 최적화 기법을 분석하는 연구를 진행하고자 한다.

5. 결 론

본 논문에서는 GPU를 사용하여 태스크를 병렬처리 하기 위한 프로그래밍 모델인 CUDA에 대해 분석하였다. 또한 CPU와 연산 처리성능을 비교하기 위해 행렬 곱 수행시간을 측정하는 실험을 통해 CPU 기반 선형대수 최적화 라이브러리와 CUDA의 처리성능을 비교하는 실험을 진행하였다. 분석 결과, CUDA로 구현한 행렬 곱 연산성능이 CPU 대비 최대 485배 높은 처리성능을 보였다. 또한 CPU 기반 MKL의 경우 CUDA 대비 최대 1.2배만큼 낮은 성능으로, 다른 CPU 기반 비교군 대비 높은 성능을 보였다. 이는 MKL의 자동 병렬화와 프로세서 최적화로 인한 결과로 분석된다. 향후, MKL의 최적화 기법을 분석하고 CUDA 기반 선형대수 최적화 라이브러리인 CUBLAS를 적용했을 때 대규모 행렬 곱 성능을 비교하는 연구를 진행하고자 한다.

참 고 문 헌

[1] C. Sun, A. Shrivastava, S. Singh, A. Gupta, “Revisiting Unreasonable Effectiveness of Data in Deep Learning Era”, International Conference on Computer Vision, pp. 843-852, 2017.

[2] S. Shi, Q. Wang, P. Xu, X. Chu, “Benchmarking State-of-the-Art Deep Learning Software Tools”, International Conference on Cloud Computing and Big Data, pp. 99-104, 2016.

[3] CUDA Toolkit Documentation, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, (accessed 2022, October, 19)

[4] 최환석, 박진석, 이창건, “GPGPU를 이용한 큰 정수 곱셈 고속화”, 한국정보과학회 학술발표논문집, pp. 1574-1576, 2015.

[5] 김대희, 박능수, “GPGPU를 활용한 Spark 성능 향상, 한국정보과학회 학술발표논문집, pp. 31-32, 2016.

[6] P. Bhatotia, R. Rodrigues, A. Verma, “Shredder: GPU-Accelerated Incremental Storage and Computation”, USENIX Conference on File and Storage Technologies, Vol. 14, 2012.

[7] CUDA, [Online]. Available: https://developer.nvidia.com/cuda-toolkit, (accessed 2022, October, 19)

[8] OpenCL, [Online]. Available: https://www.khronos.org/opencl/, (accessed 2022, October, 19)

[9] Intel Math Kernel Library, [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-mkl-dpcpp-developer-reference/top.html, (accessed 2022, October, 19)

[10] J. J. Dongarra, J. Croz, S. Hammarling, I. S. Duff, “A Set of Level 3 Basic Linear Algebra Subprograms”, ACM Transactions on Mathematical Software, Vol. 16, No. 1, pp. 1-17, 1990.

[11] R. C. Whaley, A. Petitet, “Minimizing Development and Maintenance Costs in Supporting Persistently Optimized”, Software: Practice and Experience, Vol. 35, No. 2, pp. 101-121, 2005.