다중 희소 행렬-행렬 곱셈 하드웨어 가속기 연구

김태형[1], 조영필[2]
[1] 한양대학교 컴퓨터소프트웨어학과 (미래자동차-SW 융합전공) 석사과정
[2] 한양대학교 컴퓨터소프트웨어학과 교수

dks2333@hanyang.ac.kr, ypcho@hanyang.ac.kr

Study on Multiple sparse matrix-matrix multiplication hardware accelerator

Tae-Hyoung Kim[1], Yeong-Pil Cho[2]
[1]Dept, of Computer and Software (Automotive-Computer Convergence), Hanyang University
[2]Dept. of Computer Software, Hanyang University

요 약

희소 행렬은 대부분의 요소가 0 인 행렬이다. 이러한 희소 행렬-행렬 곱셈을 수행할 경우 0 인 데이터 또한 곱셈을 수행하니 불필요한 연산이 발생한다. 이러한 문제를 해결하고자 행렬 압축 알고리즘 또는 곱셈의 부분합의 수를 줄이는 연구들이 활발히 진행 중이다. 하지만 현재의 연구들은 주로 단일 행렬 연산에 집중되어 있어 FPGA(Field Programmable Gate Array)와 특정 용도로 사용하는 가속기에서는 리소스를 충분히 활용하지 못해 비효율적이다. 본 연구는 FPGA 의 모든 리소스를 사용하여 다중 희소 행렬 곱셈을 수행하는 아키텍처를 제안한다.

1. 서론

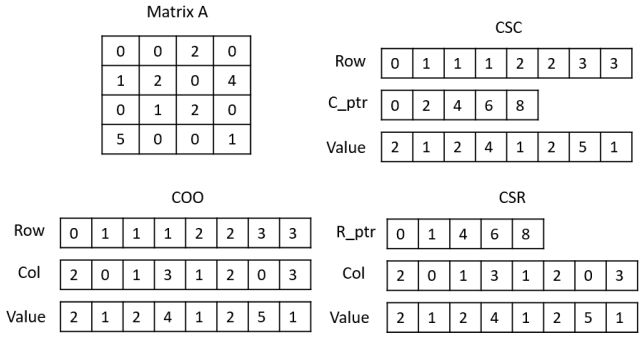
SpGEMM(General Sparse Matrix-Matrix Multiplication)은 과학적 컴퓨팅, 그래프 이론 및 네트워크 분석, 딥러닝[1], 컴퓨터 비전[2] 등 다양한 방면에서 사용된다. 특히 딥 러닝 분야인 컴퓨터 비전, 자연어 처리[3], 음성 인식과 같은 시스템에서 SPGEMM 연산이 사용되고 관련 분야로 지속적으로 연구되고 있다. 희소 행렬이란 행렬 내부 대부분의 요소가 0 으로 구성된 행렬을 말하며, 이러한 특성으로 인해 저장 공간 낭비 및 0 인 요소를 곱하는 과정에서 연산 오버헤드가 발생한다. 최근에는 이러한 단점을 해결하고자 희소 행렬 압축 알고리즘들이 등장하였고, 다양한 플랫폼(ASIC, FPGA, GPU)에서 효율적으로 행렬 간 곱셈을 해결하고자 한다. 이러한 연구는 단일 희소 행렬-행렬 곱셈을 최적화하는 데 중점을 둔다. 하지만 구독형 방식인 클라우드 FPGA 인 AWS F1[4]이 등장함에 따라 동일한 비용을 지불하는데 일부분의 리소스만 사용하게 되면 비효율적이고, 실제 FPGA 장비를 구매한 경우도 마찬가지로 모든 리소스를 사용하지 않으면 리소스 낭비가 심하다. 이 부분을 해결하고자 하나의 행렬 곱셈을 처리하는 것이 아닌 모든 FPGA 리소스를 사용하여 다중 행렬 곱셈을 처리하는 모듈이 필요하다.

본 연구에서는 하드웨어로 구현된 다양한 희소 행렬-행렬 곱셈 작업들을 소개하고 FPGA 에 기반한 다중 희소 행렬 곱셈 아이디어를 소개한다.

2. 배경 지식

2.1 희소 행렬 압축 방식

희소 행렬-행렬 곱셈을 수행할 경우 일반적인 해결 방법으로는 희소 행렬을 0 이 아닌 데이터만 저장하여 처리하는 방식이 있다. 그림 1 과 같이 희소 행렬 압축 형식으로는 크게 3 가지가 존재한다.



(그림 1) 희소 행렬 압축 방식

COO(Coordinate List)[5]는 좌표리스트라는 뜻으로 Row, Col 그리고 Value 형식으로 저장하는 방식이다.

CSR(Compressed Sparse Row)[5]는 Value, Col, 그리고 각 행의 시작 위치를 나타내는 Row_pointer 로 이루어져 있다. 마지막으로 CSC(Compressed Sparse Column)[5]는 Value, Row, 그리고 각 열의 시작 위치를 나타내는 Col_pointer 를 메타데이터로 가진다.

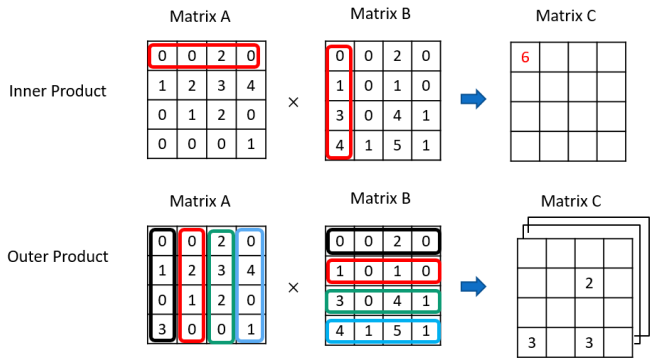
2.2 행렬 곱셈 방식

행렬 곱셈 방식으로는 대표적으로 Inner Product, Outer Product 가 존재한다. Inner Product 는 두 행렬의 각 성분이 대응되는 곱의 합으로 두 행렬 a 와 b 가 주어졌을 경우 행렬 c 의 각 성분은 다음과 같이 계산된다.

$$c_{ij} = \sum_{k=0}^{n} a_{ik}\, b_{kj} \quad (1)$$

Outer Product 는 행렬 a 의 열 벡터와 행렬 b 의 행 벡터를 곱하는 식으로 진행된다. 결과 행렬 c 는 다음과 같이 표현된다.

$$C = a \otimes b = ab^T \quad (2)$$



(그림 2) 행렬 곱셈 방식

그림 2는 식으로 표현된 Inner Product 와 Outer Product 를 실제 행렬 형식으로 표현하였다.

3. 관련 연구

Pal 외 9인은 Outer Product 를 사용한 희소 행렬 곱셈 가속기인 OuterSPACE[6]를 ASIC(Application Specific Integrated Circuit)형태로 설계하였다. 해당 설계는 두 번째 행렬의 희소성을 고려하지 않아 많은 0 값의 요소를 곱하고, 부분합 결과를 정렬할 때 버블 정렬을 사용하여 부분합이 많아질 경우 비효율적이다.

Zhang 외 3 인은 기존 Outer Product 의 입력 또는 출력 데이터 재사용의 비효율성으로 인해 DRAM 액세스가 빈번하게 일어난다는 점을 지적한다. 이를 해결하고자 입력 행렬을 압축하고 최적의 병합 순서를 결정하기 위해 Huffman Tree Scheduler 를 설계하여 부분합 개수를 줄이도록 설계하였다. 그들이 설계한 SpArch[7]는 부분 합의 개수를 줄이는데 성공하였지만 부분 곱의 수는 줄일 수 없다.

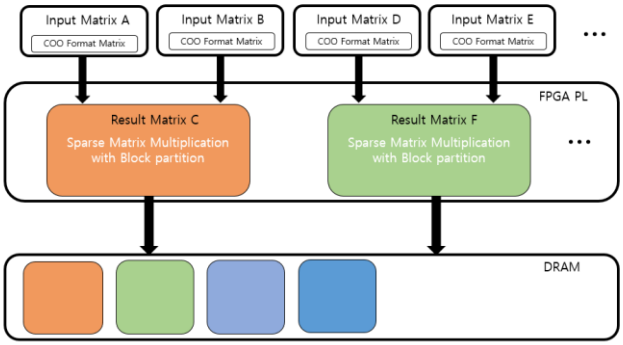
Hojabr 외 4 인은 FPGA 기반 희소 행렬 곱셈 가속기인 SpaGhetti[8]를 설계하였다. 그들은 큰 전체 행렬을 더 작은 블록으로 나눠 처리하여 OuterSPACE 의 출력 데이터 재사용의 비효율성을 해결하였다. 하지만 SpaGhetti 는 작은 블록으로 나누어 처리할 때 분할된 블록에 대해 희소성을 고려하지 않아 연산의 오버헤드가 존재한다. 또한 SpArch 와 마찬가지로 부분 행렬의 수는 줄이지만 부분 곱의 수는 줄이지 않았다.

Hesam 외 4 인은 DNN 에서 희소 행렬 곱셈을 효율적으로 처리한 HIRAC[9]를 설계하였다. HIRAC 는 Software-Hardware 공동 설계로 Software 부분에서 희소 행렬을 조밀한 행렬로 변환하고 Hardware 에서는 행렬 간 곱셈을 수행한다. Quick Sort 를 사용하여 행렬 간 곱셈을 수행할 경우 같은 사이클에 최대한 많은 부분 곱을 더하는 방식으로 설계하여 데이터 처리량을 높였다. 하지만 Software 에서 생성된 연산 오버헤드는 평가에서 고려하지 않았고, 큰 행렬을 부분 행렬을 부분 행렬로 분할할 경우 부분 행렬의 희소성을 고려하지 않았다.

앞서 구현된 디자인은 모두 단일 희소 행렬-행렬 곱셈 모듈들이다. Zhuang 외 7 인은 딥 러닝 애플리케이션에서 발생하는 다양한 크기의 밀도 높은 행렬-행렬 곱셈을 병렬 처리하는 CHARM[10]을 설계하여 기존 단일 대형 가속기를 사용하였을 때보다 높은 처리량을 보였다. 그러나 CHARM 은 일반적인 행렬 곱에 관련된 아키텍처이므로 희소 행렬에 대한 처리가 존재하지 않아 많은 0 인 요소로 인해 연산 오버헤드가 발생할 수 있다.

4. 구현

본 연구에서는 다중 희소 행렬-행렬 곱셈을 제안한다. 그림 3 은 전체 아키텍처를 보인다.

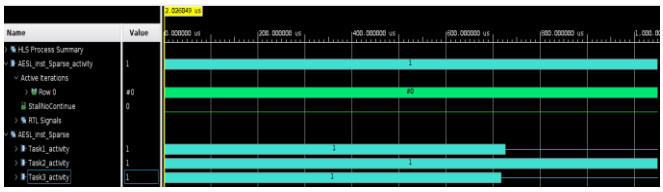


(그림 3) 다중 희소 행렬-행렬 곱셈 아키텍처

먼저 DRAM 으로부터 COO 형식으로 행렬을 받아온다. 출력 데이터 재사용과 병렬 처리를 위해 입력 행렬을 작은 블록으로 나누어 Inner Product 로 곱셈을 수행한다. 또한 병렬 처리를 극대화하기 위해 작은 블록을 더 작은 블록으로 나누어 곱셈을 진행한다. 나누어진 블록의 크기는 처음 입력 행렬들의 크기에 따라 최적의 크기가 존재한다. 행렬을 너무 작게 나누게 되면 여러 작은 행렬의 부분 곱을 다시 큰 행렬

로 더하는 과정에서 오버헤드가 증가하여 전체적인 계산 시간이 증가할 수도 있고, 작은 행렬을 FPGA 의 BRAM 에서 가져오는 빈도가 증가할 수 있다. 본 연구는 희소 행렬의 크기에 적합한 최적의 부분 행렬의 크기를 미리 계산하여 구현하였다.

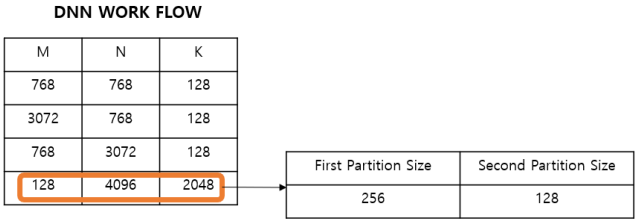
FPGA 는 출력 값에 데이터 의존성이 존재하지 않으면 동시에 여러 작업을 수행할 수 있다. 이점을 이용하여 다양한 입력 행렬에 대한 출력 행렬을 다수 생성 가능하다. 그림 4 는 Vivado 시뮬레이션 툴을 사용하여 희소 행렬-행렬 곱셈이 병렬적으로 처리되는 프로세스를 보여준다. FPGA 시스템은 컴파일 타임에 모든 리소스의 할당과 구성이 결정되므로 처리 시간을 줄이기 위해 입력 행렬 중 가장 큰 데이터를 기준으로 최적의 블록 크기를 선택하는 것이 전체 아키텍처 실행 시간을 줄일 수 있다.



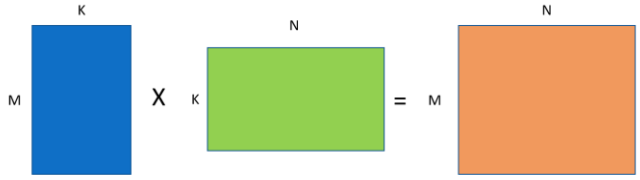
(그림 4) 작업간 병렬처리

5. 평가

데이터 세트로는 Sparkle[11]에서 DNN WORK FLOW 의 행렬 크기를 가져와 실험을 진행하였다. 일반적으로 희소성이 높으면 Latency 가 빠르고 희소성이 낮으면 느리다. 본 연구의 평가 기준으로는 입력 행렬의 희소성 비율을 40%로 설정하고 실험을 진행하였다.



(그림 5) 하위 행렬 크기 선택



(그림 6) DNN 행렬 크기

입력 행렬의 크기 중 가장 큰 행렬이 일반적으로 행렬 간 곱셈을 수행할 경우 실행시간이 길기 때문에 해당 행렬을 기준으로 최적의 블록 크기를 정한다. 이와 같이 설계하면 모든 행렬-행렬 곱들은 가장 큰 행렬-행렬 곱셈이 종료되기 전에 완료된다. 표 1 는 DNN WORK FLOW 에서 행렬의 크기에 따른 희소 행렬 곱셈을 수행할 경우 실행 시간을 나타낸다. 표에서 가장 큰 행렬의 실행 시간이 높은 것을 알 수 있다. 그림 5, 6 은 DNN WORK FLOW 와 가장 큰 행렬의 첫 번째 분할과 두 번째 분할 시 블록 크기를 나타낸다.

본 연구에서는 Zynq Ultrascale+ FPGA 를 사용하여 리소스를 최대한 사용하여 처리량을 향상하고자 한다. HLS 툴을 사용하여 합성을 진행하였고, 클럭 주파수는 100MHz 로 설정하였다. 표 2 는 Zynq Ultrascale+ FPGA 에서 합성된 단일 행렬-행렬 곱셈의 리소스 사용량을 보여준다. 단일 디자인으로는 전체 FPGA 리소스 사용량의 6%만 사용하여 리소스 낭비가 발생한다. HLS 는 #pragma dataflow 지시문을 사용하여 데이터 의존성이 없는 모듈들을 동시에 병렬적으로 처리할 수 있다. 이를 활용하면 표 2 에서 나타난 것처럼 본 연구에서 사용하는 Zynq Ultrascale+ FPGA 기준으로 13 개의 디자인을 병렬적으로 처리할 수 있다.

M	N	K	Latency(Cycle)
768	768	128	23738
3072	768	128	94953
768	3072	128	94953
128	4096	2048	258969

<표 1> DNN WORK FLOW 실행 시간

	LUT(K)	FF(K)	DSP	BRAM Tile
Zynq Ultrascale+	230	461	1728	312
Our Design	16.278	14.382	131	23
Our Design x 13	211.64(91%)	186.966(40.3%)	1703(98.5%)	299(95.8%)

<표 2> FPGA 리소스 사용량

6. 결론 및 향후 연구

본 연구에서는 희소 행렬-행렬 곱셈에 대한 기존 연구를 소개하고 다중 희소 행렬-행렬 곱셈 연구가 필요한 이유를 설명한다. Zynq Ultrascale+ FPGA 의 리소스를 최대한 활용하면, 동일한 시간 동안 기존의 단일 희소 행렬-행렬 곱셈 연산보다 13 배 많은 다중 행렬 곱셈을 수행할 수 있다. 향후 연구에는 기존 설계에서의 리소스 불균형을 처리하고, 구현한 로직을 실제 FPGA 장비에 프로그래밍 하여 테스트 예정이다.

이 논문은 과학기술정보통신부의 재원으로 정보통신기획평가원(No. 2020-0-01840, 스마트폰의 내부데이터

접근 및 보호 기술 분석)과 한국연구재단(No. NRF-2022R1A4A1032361, Processing-in-Memory 보안 기술 개발)의 지원을 받아 수행된 연구임

참고문헌

[1] Lin, Dian-Lun, and Tsung-Wei Huang. "Accelerating large sparse neural network inference using GPU task graph parallelism." *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2021): 3041-3052.

[2] Xie, Xinfeng, et al. "Exploiting sparsity to accelerate fully connected layers of cnn-based applications on mobile socs." *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (2017): 1-25.

[3] Wu, Yonghui, et al. "Google's neural machine translation system: Bridging the gap between human and machine translation." *arXiv preprint arXiv:1609.08144* (2016).

[4] https://aws.amazon.com/ko/ec2/instance-types/f1/

[5] Dongarraxz, Jack, et al. "A sparse matrix library in C++ for high performance architectures." *Proc. 2nd Object Oriented Numerics Conf.* 1994.

[6] Pal, Subhankar, et al. "Outerspace: An outer product based sparse matrix multiplication accelerator." *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[7] Zhang, Zhekai, et al. "Sparch: Efficient architecture for sparse matrix multiplication." *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020

[8] Hojabr, Reza, et al. "Spaghetti: Streaming accelerators for highly sparse gemm on fpgas." *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021.

[9] Shabani, Hesam, et al. "Hirac: A hierarchical accelerator with sorting-based packing for spgemms in dnn applications." *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023.

[10] Zhuang, Jinming, et al. "CHARM: C omposing H eterogeneous A ccele R ators for Matrix Multiply on Versal ACAP Architecture." *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2023.

[11] Xu, Shiyao, et al. "Sparkle: A high efficient sparse matrix multiplication accelerator for deep learning." *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022.