ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 51, No. 7, pp. 583-591, 2024. 7
https://doi.org/10.5626/JOK.2024.51.7.583

Hierarchical N:M Sparsity를 통한 DNN 모델 가속화

(Accelerating DNN Models via Hierarchical N:M Sparsity)

유 승 민 † 이 하 윤 † 신 동 군 ††
(Seungmin Yu) (Hayun Lee) (Dongkun Shin)

요 약 N:M sparsity pruning은 NVIDIA의 Sparse Tensor Core 기술을 활용하여 심층 신경망을 압축하는 강력한 기법입니다. 하지만 이 기법은 하드웨어에 제한으로 인해 압축률이 고정되고 불필요한 입력 데이터 접근이 늘어나며 불균형한 중요 파라미터의 분포를 해결하지 못한다. 본 논문은 vector sparsity를 먼저 적용한 후 N:M sparsity를 적용하는 Hierarchical N:M(HiNM) sparsity를 제안하고, 이 sparsity를 위해 설계된 2-axis channel permutation (2CP) 기법을 제안한다. 실험 결과, HiNM sparsity은 기존 N:M sparsity보다 2배 높은 압축률에서 latency가 평균적으로 37% 감소하였다.

키워드: NVIDIA GPU, 희소 텐서 코어, N:M 희소성, 채널 순열 알고리즘, 프루닝

Abstract N:M sparsity pruning is an effective approach for compressing deep neural networks by leveraging NVIDIA's Sparse Tensor Core technology. Despite its effectiveness, this technique is constrained by hardware limitations, leading to fixed compression ratios and increased access to unnecessary input data, and does not adequately address the imbalanced distribution of essential parameters. This paper proposes Hierarchical N:M (HiNM) sparsity, where vector sparsity is applied prior to N:M sparsity for various-levels of sparsity. We also introduce a novel permutation technique tailored for HiNM sparsity, named 2-axis channel permutation (2CP). The experimental results showed that HiNM sparsity achieves a compression ratio twice that of traditional N:M sparsity while reducing latency by an average of 37%.

Keywords: NVIDIA GPU, Sparse Tensor Core, N:M sparsity, channel permutation, pruning

1. 서 론

딥 러닝 모델이 지속적으로 발전함에 따라, 산업계는 모델의 정확도를 향상시키기 위해 모델 크기를 확대해왔다. 예를 들어, 컴퓨터 비전 분야에서 사용되는 DALL-E[1]는 약 120억개의 파라미터를 사용하고, 자연어 처리에서는 사용되는 대규모 언어 모델인 GPT-3[2]는 1750억개 이상의 파라미터를 사용한다. 그러나 모델의 무분별한

· 이 논문은 2024년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)

† 비 회 원 : 성균관대학교 전자전기컴퓨터공학과 학생
sminy67@gmail.com
lhy920806@skku.edu
†† 종신회원 : 성균관대학교 소프트웨어학과 교수
(Sungkyunkwan Univ.)
dongkun@skku.edu
(Corresponding author임)

논문접수 : 2023년 10월 10일
(Received 10 October 2023)
논문수정 : 2024년 4월 16일
(Revised 16 April 2024)
심사완료 : 2024년 4월 24일
(Accepted 24 April 2024)

정보과학회논문지 제51권 제7호(2024. 7)

확장은 메모리 요구 사항과 계산 복잡성을 증가시켜 많은 비즈니스 응용 분야에서 상당한 부담이 되고 있다. 이 문제를 완화하기 위해, 기존 연구[3,4,5]들은 pruning 기법을 제안해왔다.

Pruning 기법은 딥 러닝 모델의 가중치 요소들의 중요도를 표현하고 중요하지 않은 가중치를 제거하여 모델의 정확도를 해치지 않고 경량화 한다. 이 기법은 pruning을 통해 나타나는 sparsity 패턴에 따라 모델의 정확도와 계산 속도 사이에 트레이드오프를 발생시킨다. 예를 들어, element-wise pruning 기법[5]은 각 가중치 요소를 개별적으로 제거함으로써 모델의 정확도를 유지한다. 그러나 이 방법은 불규칙한 sparsity 패턴을 야기하여, 가중치 요소에 해당하는 입력을 인덱싱하는 과정을 복잡하게 만든다. 이는 메모리 접근의 불규칙성을 초래하고 모델의 전체적인 계산 속도를 느리게 하는 결과를 가져온다.

최근 NVIDIA의 Sparse Tensor Core (STC)[6]는 N:M sparsity 패턴이라고 불리는 새로운 pruning 패턴의 연산을 지원하기 시작했다. N:M sparsity는 가중치의 동일한 output (row) channel에서 연속적인 M개의 가중치 요소들을 하나의 stripe로 활용하며, stripe 내의 임의의 N개를 제외한 나머지 가중치 요소들을 제거한다. 이는 stripe이라고 불리는 구조적 패턴 내에서 불규칙한 패턴을 가지므로 기존의 element-wise pruning 기법[5]과 같이 정확도 감소를 최소화한다. 한편, element-wise pruning 패턴과 달리, N:M sparsity은 STC의 하드웨어 기반 인덱싱을 통해 가중치 요소에 해당하는 입력에 접근한다. 이는 하드웨어에서 복잡한 인덱싱 시스템을 처리함으로써 불규칙한 메모리 접근을 줄일 수 있다.

그림 1은 N과 M이 각각 2와 4로 설정된 N:M sparsity 패턴을 적용한 가중치의 저장 방식과 STC에서 Sparse Matrix Multiplication (SpMM) 연산 과정을 설명한다. 이 때, 색상은 입력 채널에 따라 다르게 표현한다. N:M sparsity 패턴은 stripe에서 선택된 가중치 요소의 위치를 2비트 인덱스로 저장한다. 그림1(a)를 보면, 첫 번째 행의 첫 stripe에서 세 번째와 네 번째 가중치 요소가 선택되고, 각 가중치 요소의 위치는 2비트 인덱스인 2와 3으로 표현된다. 이후 SpMM 연산을 위해 GPU의 글로벌 메모리에서 압축된 가중치, NM 인덱스, 입력이 공유 메모리를 통해 STC 내의 레지스터로 로드된다. 그림 1(b)에서는 압축된 가중치, 인덱스, 입력이 블록 단위로 공유 메모리에 로드되어 GPU의 병렬성을 활용하고, STC에서는 하드웨어 기반 인덱싱을 통해 가중치 요소에 해당하는 입력 값이 선택된다. 이후, 행렬 곱 누적 (MMA) 연산을 통해 최종 출력 값이 반환된다.

그러나, N:M sparsity는 NM 인덱스와 pruning 패턴으로 인해 세 가지 문제를 가지고 있다. 첫 번째 문제는 STC가 인덱싱을 통해 입력에 접근하면서 인덱싱 오버헤드를 줄이지만, 불필요한 입력 요소도 레지스터로 읽어야 한다는 것이다. 그림1(b)를 보면, STC에서 SpMM 연산에 필요한 입력 요소의 수는 4개이지만, STC는 총 8개의 입력 요소를 레지스터에 로드 한다. 이로 인해 불필요한 메모리 접근이 증가하게 된다.

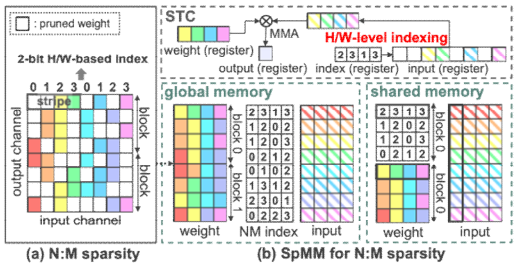

그림 1 N:M 희소성와 이를 통한 SpMM 연산 과정
Fig. 1 N:M sparsity and SpMM operation process

두 번째 문제는 STC에서 하드웨어 기반 인덱싱을 통해 사용되는 인덱스가 2비트로 고정되어 있어, 오직 2:4 sparsity만 지원된다는 점이다. 최근 연구[7]에서는 2:8이나 2:16과 같은 높은 압축 비율을 가진 N:M sparsity 패턴으로 딥 러닝 모델을 pruning 했을 때도 정확도 손실이 거의 없다는 것이 실험을 통해 입증되었다. 그러나 이러한 sparsity 패턴은 압축된 가중치의 위치를 인덱스로 표현하기 위해 2비트 이상이 필요하기 때문에, STC의 하드웨어 기반 인덱싱의 혜택을 받을 수 없다.

마지막 세 번째 문제는 uniform sparsity 문제이다. 이 문제는 N:M pruning 기법이 모든 stripe에서 동일한 수의 가중치 요소를 제거하는데, 이는 stripe 간 중요 가중치 요소의 분포 불균형에 취약하다. Stripe는 한 출력 채널에서 연속된 M개의 가중치 요소로 구성되어, stripe에 따라 중요 가중치 요소의 수가 변할 수 있다. 그럼에도 N:M sparsity는 모든 stripe에서 일괄적으로 동일한 수의 가중치 요소가 제거하며, 이로 인해 중요한 가중치 요소가 제거되거나 중요하지 않은 가중치 요소가 선택되는 경우가 발생한다. 이는 궁극적으로 딥 러닝 모델의 정확도를 저하시키게 된다.

본 논문에서는 N:M sparsity와 관련된 문제점들을 극복하기 위해 Hierarchical N:M (HiNM) sparsity 패턴을 제안하고, 이에 적합한 GPU 커널을 개발하였다. HiNM sparsity 패턴은 N:M sparsity 패턴을 적용하기 전에 vector-wise (VW) sparsity 패턴을 적용함으로써, 이중 레벨의 sparsity 구조를 형성한다. VW pruning 기법은 가중치를 벡터 단위로 그룹화하고, 중요도가 낮

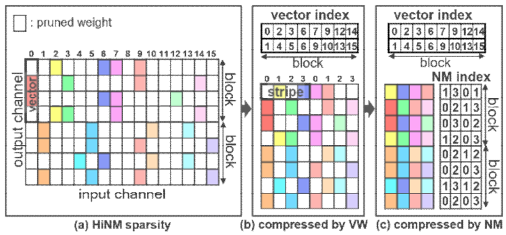

그림 2 계층적 N:M 희소성
Fig. 2 Hierarchical N:M sparsity

은 벡터를 제거한 뒤, 벡터 인덱스로 각 벡터의 위치를 저장한다. 이 방법은 GPU의 글로벌 메모리에서 공유 메모리로 데이터를 로드할 때 필요한 입력 요소만 선택적으로 읽도록 하여, 불필요한 메모리 접근을 줄일 수 있다.

HiNM sparsity 패턴은 VW pruning을 통해 불필요한 가중치를 사전에 제거함으로써 N:M pruning을 수행할 때 stripe 내에서 중요하지 않은 가중치의 선택을 방지한다. 이는 기존 N:M sparsity의 한계를 넘어서서 가중치의 중요도에 따라 더 효율적인 가중치 압축이 가능하게 하며, 이로 인해 메모리 접근 효율성과 연산 속도가 향상될 수 있다. 또한, 벡터 인덱스를 활용함으로써 다양한 압축 비율을 지원할 수 있으며, STC의 하드웨어 기반 인덱싱과 결합하여 높은 성능을 달성할 수 있다.

그림 2는 HiNM sparsity 패턴의 압축 방법을 상세히 설명한다. 그림 2(a)에서 HiNM sparsity 패턴을 보여주고 있다. HiNM sparsity는 먼저 그림 2(b)에서 보이듯이 VW sparsity를 통해 가중치를 압축하고, 벡터 인덱스로 살아남은 벡터의 위치를 표현한다. 이어서 그림 2(c)에서는 N:M sparsity를 적용하여 가중치를 추가로 압축하고, NM 인덱스는 각 stripe 내에서 선택된 가중치 요소의 위치를 나타낸다. HiNM sparsity 패턴은 최종적으로 벡터 및 NM 인덱스와 함께 압축된 가중치를 메모리에 저장한다.

HiNM sparsity 패턴은 서로 다른 채널 축에 대하여 sparsity 패턴을 가지기 때문에 축에 따라 서로 다른 목적으로 pruning 해야 한다. HiNM pruning기법은 입력 채널 축에 따라 벡터 단위로 중요한 가중치 요소들이 모이도록 해야 하고, 출력 채널 축에 따라 stripe 간 중요한 가중치 요소가 골고루 분산되도록 해야 한다. 이러한 문제를 대처하기 위해 본 논문에서는 HiNM sparsity에 대한 2-axis Channel Permutation (2CP) 기법을 제안한다. 이 기법은 출력 채널의 순서와 입력 채널의 순서를 변경하여 가중치의 중요성에 따라 더 효율적으로 가중치 요소들을 벡터 단위로 그룹화하고, N:M sparsity 패턴의 stripe 간 중요 가중치 분포의 균형을 개선하는 데 목적을 둔다.

2CP 기법은 두 주요 단계로 구성된다. 첫 번째 단계인 output channel permutation (OCP)에서는 출력 채널의 순서를 재배열하여 중요도가 낮은 가중치 요소들이 벡터 단위로 효율적으로 그룹화되도록 한다. 이 조정은 VW sparsity 과정에서 제거되는 가중치의 중요도를 최소화하고 중요 가중치의 보존 가능성을 높인다. 두 번째 단계인 input vector permutation (IVP)에서는 남아 있는 벡터들의 위치, 즉 벡터 인덱스의 순서를 조정하여 N:M sparsity 단계에서 stripe 간의 가중치 제거가 균형 있게 이루어지도록 한다. 이러한 조정은 uniform sparsity 문제를 해결하고 중요 가중치의 잘못된 제거를 줄이는 데 기여한다.

본 논문에서는 다양한 크기의 가중치 행렬을 대상으로 NM과 HiNM의 지연 시간(latency)을 비교 분석한다. 실험 결과에 따르면, 50% 압축률을 가진 NM에 비해 75% 압축률을 가진 HiNM이 평균적으로 37% 낮은 latency를 달성하는 것으로 나타났다. 또한, ResNet 모델에서의 2CP 기법의 성능 평가를 위해 accuracy를 측정하였다. 실험에 따르면, ResNet18 모델과 ResNet50 모델에서 2CP 기법을 적용하기 전과 후의 accuracy 비교에서 각각 5.12%와 4.12%의 향상되었다.

본 논문은 총 여섯 개의 장으로 구성되어 있다. 두 번째 장에서는 배경과 관련 연구를 리뷰하여 HiNM과 2CP 기법의 연구 맥락과 필요성을 설명한다. 세 번째 장에서는 HiNM의 메모리 저장 방식과 GPU에서의 연산 방식을 상세히 서술한다. 네 번째 장에서는 2CP 기법의 OCP 및 IVP 기법을 설명한다. 다섯 번째 장에서는 HiNM을 활용한 GPU kernel의 성능 및 2CP 기법의 효율성을 실증적으로 증명하는 실험 결과를 제시한다. 마지막 여섯 번째 장에서는 연구의 주요 발견과 결론을 도출한다.

2. 배경 및 관련 연구

2.1 Pruning 기법

Pruning 기법은 가중치를 제거하는 단위에 따라 fine-grained sparsity와 coarse-grained sparsity로 분류할 수 있다. Fine-grained sparsity[7]는 개별 가중치를 선택적으로 제거하는 unstructured pruning 방식을 채택하여, 모델의 압축 비율을 높이면서도 정확도를 유지할 수 있다. 그러나 이 방식의 sparsity 패턴은 불규칙적이기 때문에, GPU에서의 SpMM 연산 처리 과정에서 복잡한 인덱스 구조가 필요하게 되며, 이는 불규칙한 메모리 접근을 유발하여 높은 압축률에도 불구하고 지연 시간을 증가시킬 수 있다.

반면, coarse-grained sparsity [8] 가중치를 채널 단

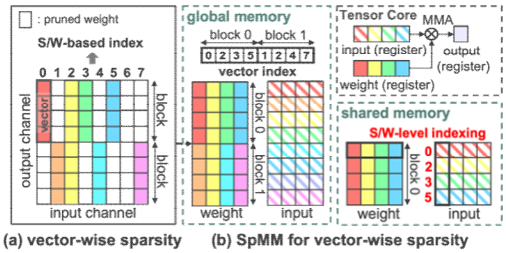

그림 3 벡터 기반 희소성을 통한 SpMM 연산

Fig. 3 SpMM operation with vector-wise sparsity

위나 필터 단위와 같은 더 큰 단위로 구조화하여 제거하는 structured pruning 방식을 사용한다. 이 접근법은 GPU에서의 SpMM 연산을 위한 인덱스의 양을 상당히 줄일 수 있어 연산 효율을 높인다. 하지만, 중요한 가중치 요소가 중요하지 않은 가중치 요소들과 함께 제거될 위험이 있어 fine-grained sparsity에 비해 모델의 정확도가 저하될 가능성이 있다. 이러한 트레이드-오프는 pruning 기법을 선택할 때 고려해야 할 중요한 요소로 작용한다.

최근 연구[3,4]에서는 unstructured pruning방식과 structured pruning 방식의 절충점으로, 가중치의 입력 채널 축을 따라 벡터 단위로 그룹화하고, 중요도가 낮은 벡터를 제거하는 vector-wise (VW) pruning방식이 제안되었다. 이 기법은 살아남은 벡터들의 위치를 벡터 인덱스로 저장하고, 이 인덱스를 통해 GPU의 글로벌 메모리로부터 공유 메모리로 입력을 로드 할 때 입력 요소를 효율적으로 불러온다. 그러므로, fine-grained sparsity에 비해 인덱싱 오버헤드를 줄이면서, structured pruning 방식보다 모델의 정확도 저하도 완화한다.

본 논문에서는 GPU의 병렬 처리 능력을 최대화하기 위해 각 블록마다 동일한 수의 벡터를 제거함으로써 GPU에서의 load balancing을 조정한다. 이를 통해 병렬성이 극대화되고, 전체적인 연산 효율이 향상된다.

그림 3는 GPU에서 VW sparsity 패턴을 활용한 Sparse Matrix Multiplication (SpMM) 연산 방식을 보여준다. 이 그림에서는 입력 채널 축을 따라 벡터를 묶어 소프트웨어 인덱싱을 통해 공유 메모리로 입력을 로드 하는 과정이 설명되어 있다. 이 과정에서 입력 요소들은 가중치 벡터들이 동일한 입력 요소들을 공유하기 때문에 최대한 재사용 가능하며, 공유 메모리에서의 VW sparsity 패턴을 통한 SpMM 연산은 기존의 행렬곱 연산(GEMM)과 유사하게 진행된다. 이러한 설계는 GPU 연산의 효율성을 높이는 데 중요한 역할을 한다.

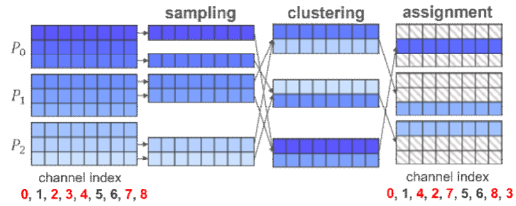

그림 4 채널 교환 기법의 일반적인 과정

Fig. 4 General process of the channel permutation method

2.2 Channel Permutation 기법

Channel Permutation 기법은 가중치의 중요도를 기반으로 하여, 중요하지 않은 가중치 요소들이 VW sparsity 패턴이나 N:M sparsity 패턴에 잘 부합하도록 채널의 순서를 재정렬하는 전 처리 방법이다. 이 기법은 목표 축의 채널들을 벡터 또는 stripe의 크기에 따라 파티션을 나누고, 파티션 간 채널 이동을 통해 전체 채널의 순서를 재정렬한다.

Channel permutation의 절차는 샘플링, 클러스터링, 할당의 세 단계로 구성된다. 샘플링 단계에서는 파티션 내 일부 채널을 추출한다. 이 후, 클러스터링 단계에서는 선택된 채널 수가 파티션 수를 초과할 경우, 클러스터링을 통해 같은 파티션에 속할 채널들을 그룹화한다. 마지막으로 할당 단계에서는 클러스터 된 채널 그룹을 적절한 파티션에 배치한다. 그림 4는 벡터의 크기가 3인 경우, VW sparsity 패턴을 위한 channel permutation 과정을 예시로 보여주고 있다. 이 과정은 3개의 출력 채널을 각각의 파티션으로 나누고, 샘플링 과정에서 각 파티션에서 2개의 채널을 선택한다. 이후 클러스터링 과정을 거쳐 적절한 파티션에 채널을 할당하고 위 과정을 반복 수행하여 점진적으로 최적의 채널 순서를 찾아가게 된다.

최근 연구에서 VW sparsity 패턴을 위한 output channel permutation 기법[3]이 제안되었다. 이 기법은 모든 채널에 대해 균형 잡힌 k-means 알고리즘을 사용하여 유사한 중요도 분포를 가진 채널들을 그룹화하고, 중요도에 따라 가중치를 벡터 단위로 묶는다. 그러나 이 기법은 한 번의 과정으로 모든 채널의 최적의 순서를 찾기 때문에, 출력 채널들을 정확하게 그룹화하는 데 어려움이 있다.

N:M sparsity 패턴을 위한 input channel permutation 기법[9]은 입력 채널의 순서를 변경하여 stripe 내에서 중요하지 않은 가중치 요소의 수를 균형 있게 배치한다. 이는 stripe 크기에 따라 입력 채널들을 파티션으로 나누고, 임의의 두 파티션에서 채널을 한 개씩 선택한다.

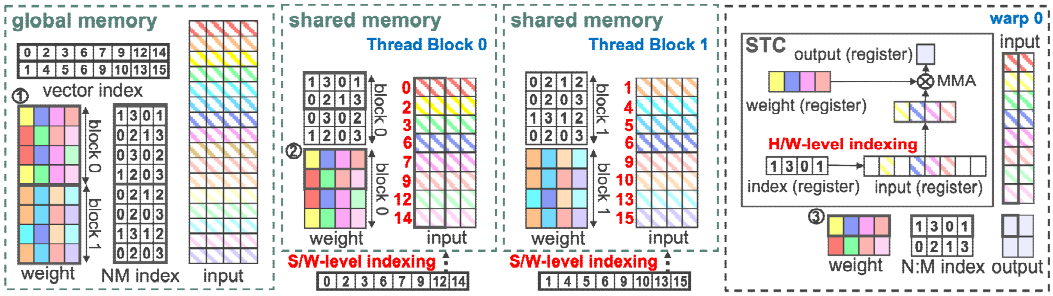

그림 5 HiNM 희소성 기반 SpMM 연산
Fig. 5 HiNM sparsity-based SpMM

이후, 추출된 두 개의 채널을 바꾸어 각 파티션에 할당하였을 때, stripe 내 불필요한 가중치의 수가 균형이 맞춰지면 두 채널의 파티션을 바꾸고 아닌 경우는 원래대로 되돌리게 된다. 이 기법은 오직 2개의 채널만 추출하여 점차 최적의 채널 순서를 찾아가기 때문에 최적화 과정에서 지역 최소값에 빠질 위험이 있다.

3. Hierarchical N:M Sparsity

3.1 HiNM Sparsity를 위한 SpMM 연산

CUDA 프로그래밍 모델은 GPU 커널을 thread block, warp, thread로 세분화하여 관리한다. 각 thread block은 여러 warp로 구성되며, 이들은 NVIDIA GPU의 Streaming Multiprocessor (SM)에서 실행된다. Warp는 32개의 thread로 구성되며, CUDA에서 연산을 수행하는 기본 단위로 작동한다. CUDA 프로그래밍에서는 이러한 계층적 구조를 고려한 설계가 중요하다.

NVIDIA GPU의 메모리 계층은 글로벌 메모리, 공유 메모리, 레지스터로 구성된다. 글로벌 메모리는 모든 SM에 의해 공유되는 대용량 메모리이지만 접근 시간이 긴 단점이 있다. 반면, 공유 메모리는 각 SM에 할당되어 각 thread block의 warp들에게 공유된다. 이 후, 실제 연산은 각 쿠다 코어 혹은 텐서 코어에서 레지스터를 통해 warp 단위로 진행된다.

그림 5은 HiNM sparsity 기반의 SpMM 연산 과정을 보여주며, 이 연산은 메모리 계층 구조와 CUDA 프로그래밍 모델을 함께 고려해야 한다. HiNM sparsity 패턴의 메모리 저장 방식에 따라, 가중치는 글로벌 메모리에 벡터 및 NM 인덱스를 통해 압축된 형태로 저장되어 있다. 글로벌 메모리에서 각 블록에 해당하는 가중치는 공유 메모리에 로드되고 입력은 해당 블록의 소프트웨어 기반 인덱싱을 통해 공유 메모리에 로드 된다. 이 후 공유 메모리에 로드 한 데이터는 thread block에 할당된다. 그림 5의 ①를 보면, 가중치는 두 개의 블록으로

그림 6 **mma.sp** 명령어
Fig. 6 **Mma.sp** instruction

나뉘어져 있고, 각 블록은 서로 다른 thread block들에 할당되어 각각 공유 메모리에 로드되는 것을 볼 수 있다.

공유 메모리에서는 thread block의 warp의 개수에 따라 각 warp가 할당 받는 데이터를 결정하게 된다. 예를 들어 그림5의 ②에서는, 한 개의 thread block이 2개의 warp로 구성되어 있으며, 첫 번째 warp는 첫 두 행을 처리하고, 두번째 warp는 다음 두 행을 처리하도록 할당되었다. 그림 5의 ③에서, 해당 warp는 공유 메모리에서 레지스터로 **mma.sp** 명령어가 요구하는 데이터를 로드하고, **mma.sp** 명령어를 통해 STC에서 연산을 수행한다.

mma.sp 명령어는 STC에서 warp 단위로 N:M sparsity 기반 SpMM 연산을 수행하는 명령어이다. 그림 6는 **mma.sp** 명령어의 예시를 보여주는데, m16n8k32는 16×16 크기의 가중치와 32×8 크기의 입력을 한 개의 warp가 레지스터로 로드한다는 것을 의미한다. 이 때, 가중치의 열의 개수와 입력의 행의 개수가 다른 이유는 입력을 STC 내에서 하드웨어 인덱싱을 통해 선택하기 때문이다.

3.2 GPU 커널 구현

본 논문에서는 CUDA 프로그래밍 모델의 계층적 구조를 고려하여 thread block에 할당되는 block tile, warp에 할당되는 warp tile, 그리고 **mma.sp** 명령어가 요구하는 크기의 mma tile이라는 세 가지 주요 tile을 활용한다.

그림 7은 HiNM 기반 SpMM 연산을 수행하는 GPU 커널의 일부를 의사코드 형식으로 보여준다. 가중치와 입력을 글로벌 메모리로부터 공유 메모리로 로드 할 때는 vectorized load를 통해, 메모리 대역폭을 최적화한다 (코드라인 3~4). 반면, H/W index를 공유 메모리로 로드 할 때는 각 16개의 index(2bit)가 INT형으로 저장

```
1  for (int block_tile_k = 0; block_tile_k < NumBlockTile_K; block_tile_k++) {        Block Tile
2    // global to shared (vector load)
3    *((int4 *)&A_shared[shared_A_idx] = *((int4 *)&A[global_A_idx];
4    *((int4 *)&B_shared[shared_B_idx] = *((int4 *)&B[vec_idx[global_B_idx];
5    nm_idx_shared[shared_nm_ptr] = nm_idx[global_nm_ptr];
6    for (int warp_tile_k = 0; warp_tile_k < NumWarpTile_K; warp_tile_k++) {       Warp Tile
7      // shared to register (ldmatrix)
8      #pragma unroll
9      for (int mma_m = 0; mma_m < NumMMATile_M; mma_m++) {
10       asm volatile ("ldmatrix.sync.aligned.x4.m8n8.shared.b16 {%0, %1, %2, %3}, [%4];"
11                     : "=r"(A[0]), "=r"(A[1]), "=r"(A[2]), "=r"(A[3]) : "r"(tile_A_shared_ptr));
12       uint32_t E = nm_idx_shared[reg_nm_ptr];
13       for (int mma_n = 0; mma_n < NumMMATile_N; mma_n++) {                       MMA Tile
14         asm volatile ("ldmatrix.sync.aligned.trans.x4.m8n8.shared.b16 {%0, %1, %2, %3},
15                       [%4];" : "=r"(B[0]), "=r"(B[1]), "=r"(B[2]), "=r"(B[3]) : "r"(tile_A_shared_ptr));
16         asm volatile (
17           "mma.sp.sync.aligned.m16n8k32.row.col.f16.f16.f16.f16\n\t"
18           "{%0, %1}, {%2, %3, %4, %5}, {%6, %7, %8, %9}, {%10, %11}, %12, 0x0;"
19           : "=r"(D[0]), "=r"(D[1]) : "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
20           "r"(B[0]), "r"(B[1]), "r"(B[2]), "r"(B[3]), "r"(D[0]), "r"(D[1]), "r"(E));
```

그림 7 HiNM 기반 SpMM 연산의 의사코드
Fig. 7 pseudocode of HiNM based SpMM

되어 있어서 한 블록 당 가져오는 H/W index의 크기가 상대적으로 작기 때문에 vectorized load를 활용하지 않는다 (코드라인 5). 가중치와 입력을 공유 메모리에서 레지스터로 전달할 때는 **ldmatrix** 명령어를 이용한다. **ldmatrix** 명령어는 **mma.sp** 명령어와 동일하게 warp 단위로 동작하는 명령어로, **mma.sp** 명령어에서 요구하는 데이터 레이아웃에 따라 레지스터에 불러올 수 있다. 세부적으로, **mma.sp** 명령어는 가중치 행렬은 row major로, 입력은 column major로 처리한다. 이 때문에, 입력을 column major로 바꾸기 위해, **ldmatrix** 명령어에 “trans”라는 동작을 추가하여 이를 맞춰준다 (코드라인 10~11, 14~15). 이후, **mma.sp** 명령어를 사용하여, STC에서 SpMM 연산을 수행한다 (코드라인 16~20).

4. 2-axis Channel Permutation

본 논문에서 제안하는 2-axis Channel Permutation (2CP) 기법은 출력 채널의 순서를 변경하는 output channel permutation (OCP)와 입력 채널의 순서를 변경하는 input vector permutation (IVP)로 이루어져 있다.

그림 8은 2CP 기법의 과정을 설명한다. 그림 8의 위쪽의 처음 행렬은 벡터의 크기가 2일 때, HiNM pruning 기법을 적용한 가중치 행렬을 나타내고 있다. 그리고 행렬의 요소의 수치는 각 가중치 요소의 중요도를 나타내고 진하게 표현된 가중치 요소가 중요한 가중치 요소임을 나타내고 있다. 2CP 기법의 OCP 기법은 중요하지 않은 가중치 요소들이 벡터 단위로 모이도록 순서를 변경하기 때문에 기존 가중치 행렬보다 제거되는 중요도 수치가 낮아진 것을 볼 수 있다. 하지만 그림8을 보면 OCP 기법만 적용한 후 N:M sparsity를 적용하여 압축하게 되면 stripe 간 uniform sparsity 문제를 해결하지 못하여 여

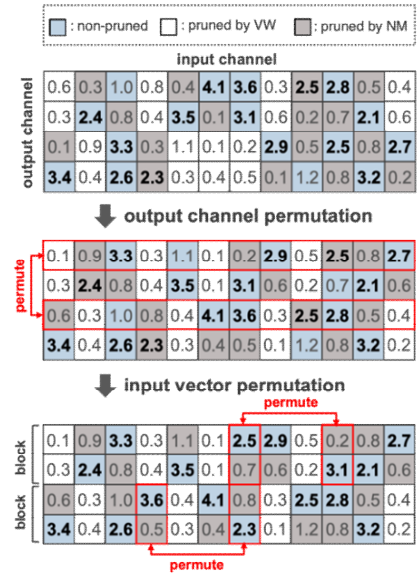

그림 8 2-axis 채널 교환 기법
Fig. 8 Two-axis channel permutation method

전히 중요한 weight가 제거되는 모습을 볼 수 있다.

한편, IVP 기법은 VW pruning 기법에서 선택되어 살아남은 벡터들의 위치, 즉 벡터 인덱스의 순서를 변경하여 uniform sparsity 문제를 해결한다. 벡터 인덱스는 출력 채널들을 벡터 크기만큼 묶은 블록 단위로 이루어져 있기 때문에 같은 블록에 있는 벡터들 간 IVP 기법을 적용한다. 그림8의 하단에서 볼 수 있듯이, IVP를 적용한 후의 가중치 행렬은 중요도가 높은 가중치가 제거되지 않는 결과를 보여준다.

4.1 Output Channel Permutation (OCP) 기법

OCP는 출력 채널의 순서를 재정렬하여, 중요하지 않은 가중치 요소들이 벡터 단위로 모이도록 한다. 이러한 재정렬은 VW pruning을 진행 시 중요한 가중치 요소를 최대한 보존하게끔 한다.

그림 9에서 보듯이 OCP 기법에서는 먼저, 각 파티션에서 무작위 출력 채널을 하나씩 선택하여 추출하는 샘플링 과정을 거친 후, 추출된 출력 채널들과 파티션 간의 모든 조합에 대한 비용 행렬을 만든다. 이 비용은 출력 채널이 파티션에 속하였을 때, VW pruning에 의해 제거되는 요소들의 합으로 측정한다. 이것은 추출된 출력 채널과 파티션 조합이 중요하지 않은 가중치 요소들을 얼마나 많이 벡터 단위로 묶을 수 있는 지를 의미한다. 그림 9을 보면, 각 파티션 P_i에서 한 개의 출력 채널 c_i을 샘플링 하고 비용 함수를 통해 비용 행렬을 계산한다. 그림에서는 두 번째 파티션과 세 번째 채널 간

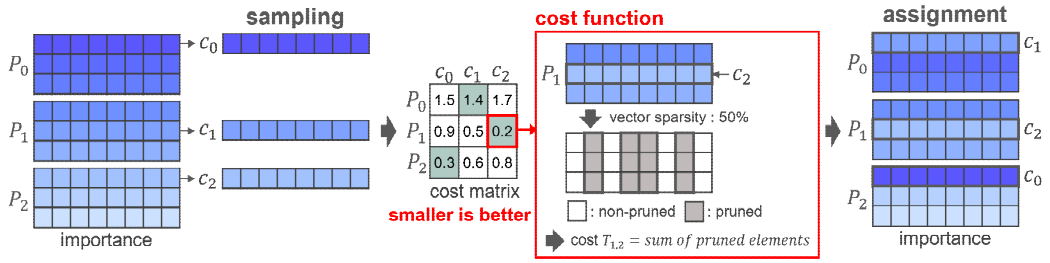

그림 9 출력 채널 교환의 과정

Fig. 9 Process of output channel permutation

의 비용을 구하는 예시를 보여준다.

마지막 할당 과정에서는 계산된 비용 행렬에 헝가리안 알고리즘을 사용하여 채널들과 파티션의 조합이 최소 비용을 가질 때를 찾고 이 때 찾아진 조합으로 채널을 파티션에 할당하게 된다. 이 최적화 과정은 반복적으로 수행된다. 그림 9에서 비용 행렬이 헝가리안 알고리즘을 통해 비용이 최소가 되는 조합을 초록색으로 표현하였다.

OCP 기법의 이점은 파티션과 출력 채널 사이의 모든 경우의 수를 비용 행렬로 나타내고 최소 비용이 되는 조합을 찾기 때문에 점진적을 비용이 줄어드는 방향으로 최적화가 가능하다. 또한, 샘플링 과정에서 모든 파티션 마다 한 개의 채널을 추출하기 때문에, 지역 최소값에 빠질 위험이 적다.

4.2 Input Vector Permutation (IVP) 기법

딥 러닝 모델에서 출력 채널과 입력 채널의 순서를 동시에 변경하게 되면 레이어 간 채널의 순서를 동일하게 맞추기 위한 추가적인 연산을 진행해야 한다. 예를 들어, n번째 레이어의 가중치의 출력 채널의 순서를 바꾸게 되면 n+1번째 레이어의 가중치의 입력 채널의 순서를 이에 맞추어 변경해야 한다. 하지만, n+1번째 레이어의 가중치의 입력 채널 역시 input channel permutation에 의해 순서가 변경되었기 때문에 이 두 순서를 맞춰줄 필요성이 있다.

반면, 2CP 기법은 먼저 가중치를 VW pruning 방식으로 압축한 후 input channel permutation을 진행하게 된다. 이로 인해 2CP 기법 중 IVP 기법은 가중치의 입력 채널의 순서를 변경하는 것이 아닌 벡터 인덱스의 순서만을 변경하기 때문에 레이어 간 채널의 순서를 맞추기 위한 추가적인 연산이 필요하지 않다.

또한, IVP 기법은 벡터 인덱스의 순서를 조정하기 때문에, 블록 별로 인덱스의 순서를 다르게 가져갈 수 있다. 이는 입력 채널보다 작은 벡터 단위로 최적화할 수 있기 때문에 N:M sparsity의 uniform sparsity 문제를 기존보다 효과적으로 해결할 수 있다.

그림 10은 input permutation을 채널 단위로 진행할

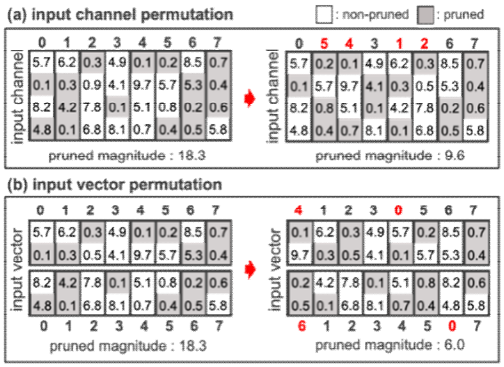

그림 10 입력 채널 단위 교환과 입력 벡터 단위 교환

Fig. 10 Input channel permutation and input vector permutation

때와 벡터 단위로 진행하였을 때의 차이점을 보여주고 있다. 그림 10(a)에서 입력 채널의 순서를 변경할 때는 입력 채널은 4개의 가중치 요소를 한 번에 이동시켜야 하지만, 그림 10(b)의 IVP 기법에서는 벡터를 구성하는 2개의 가중치 요소만 이동시키기 때문에 stripe 간 중요 가중치를 더욱 고르게 분포할 수 있다.

5. 실 험

실험에서는 ResNet 모델[10]을 대상으로 제안하는 기법의 정확도와 지연 시간을 측정하였다. VW pruning에 사용한 벡터의 크기는 32를 사용했다.

우선 제안한 permutation 기법이 정확도를 얼마나 향상시키는 지 확인하고, 이어서 지연 시간 측면에서 HiNM pruning 기법이 N:M pruning 기법 대비 얼마나 우수한 지 측정하였다.

5.1 2-axis Channel Permutation의 efficacy

본 논문에서는 2CP 알고리즘의 성능을 측정하기 위해 efficacy라는 단위를 정의한다. Efficacy는 다음과 같은 수식으로 나타난다.

$$Efficacy = \frac{|W_s^p|-|W_s|}{|W_{opt_s}|-|W_s|} \quad (1)$$

이 때, $|W_{opt_s}|$는 벡터나 stripe와 같은 sparsity 패턴에 대한 제약도 고려하지 않고 각 출력 채널 마다 동일한 개수의 가중치 요소를 제거하여 출력 채널 간 균형만을 고려한 기법을 적용했을 때의 가중치 합이다. 이것은 HiNM pruning 기법을 통해서 달성 가능한 최적의 결과를 가정하기 위해 사용하였다. $|W_s^p|$는 가중치에 2CP 기법을 먼저 적용하고 HiNM pruning을 적용하였을 때의 중요도 합을 나타내고, $|W_s|$는 가중치에 2CP 기법을 사용하지 않고 HiNM pruning을 적용하였을 때의 중요도 합을 나타낸다. 결국, Efficacy는 2CP 기법이 중요하지 않은 가중치 요소들을 얼마나 잘 벡터 단위로 묶는지와 stripe 간 중요 가중치 분포의 균형을 맞추는 지를 측정하는 지표이다.

실험에는 *pytorch*[11]에서 제공하는 사전에 학습된 ResNet 18 모델의 각 레이어의 가중치를 사용하였다. 실험에서 OCP, IVP, 2CP는 각각 가중치에 HiNM pruning을 진행하기 전에 어떤 permutation 기법을 사용하였는지를 나타낸다.

실험 결과, OCP만 적용한 경우 모든 레이어의 평균 efficacy는 13.8%에 불과했다. 이는 OCP 기법만으로는 N:M sparsity의 uniform sparsity 문제를 해결하지 못한다는 것을 확인할 수 있다. 반면, IVP 가법만 적용했을 때의 모든 레이어의 평균 efficacy는 45.9%로, OCP 기법만 적용한 경우에 비해 높게 측정되었다. 이는 IVP 기법이 블록 단위로 진행되므로 각 stripe의 uniform sparsity 문제를 상당히 완화시킬 수 있는 것을 보여준다. 마지막으로, 2CP 알고리즘 전체를 적용하였을 때 모든 레이어의 평균 efficacy는 59.8%로 측정되었다. 이 결과는 OCP와 IVP가 함께 사용될 때 서로의 효과를 상호 보완하여 효율성을 높인다는 것을 확인할 수 있다.

5.2 2CP 기법을 이용한 HiNM pruning의 정확도

본 논문은 2CP 기법을 HiNM을 적용한 모델에 활용하였을 때, 정확도를 측정하기 위해 ResNet18과 ResNet50 모델을 사용했다. 데이터셋은 ImageNet을 사용하였고 모델의 모든 가중치를 2CP 기법을 통해 채널의 순서를 재정렬 시킨 뒤, 다양한 압축 비율로 HiNM pruning을 진행했다. 이후, 모델을 50 epoch 만큼 finetuning을 했다.

그림 12는 다양한 압축률에서 ResNet18 모델과 ResNet50모델의 정확도를 나타낸다. 이 때, dense는 압축하지 않은 모델의 정확도를 나타내고, HiNM w/ 2CP와 HiNM w/o 2CP 는 각각 2CP 기법을 활용하였을 때와 활용하지 않았을 때의 정확도를 나타낸다. 실험 결과, 75% 압축률에 대하여, ResNet18모델과 ResNet50 모델에서 2CP 기법을 적용한 경우가 2CP 기법을 적용하지 않은 경우에 비해 각각 5.12%, 4.21%만큼 정확도가 향상되었다. 그 결과, 압축하지 않은 모델과 정확도가 각각 0.85%와 1.09% 밖에 차이 나지 않는다.

5.3 HiNM 기반 SpMM 연산의 지연 시간

본 논문은 GPU에서 HiNM sparsity기반 SpMM 연산의 지연 시간을 측정하기 위해 NVIDIA의 A100 GPU 1대에서 성능을 측정하였다. 가중치의 크기와 입력의 크기는 각각 1024×K와 K×1024로 설정하였고 K는 1024, 2048, 4096으로 설정하였다. HiNM 기법의 성능을 cuBLAS 및 N:M sparsity 기반 SpMM 연산과 비교했다. cuBLAS는 텐서 코어에서 기존 GEMM 연산 API인 *cublasGemmEx*를 사용하였고, N:M sparsity와 HiNM sparsity는 Sparse Tensor Core에서 1장과 3장에서 설명한 SpMM 연산 커널을 구현하였다. 이 때, 모든 데이터 타입은 FP16을 사용하였다. HiNM sparsity에서 vector sparsity는 50%로 설정하여, 2단계의 pruning 후에 sparsity는 75%가 되며, 32와 64의 2 가지 vector의 크기에 따라 성능을 측정하였다.

그림 12은 왼쪽부터 *cuBLAS*를 사용한 GEMM 연산, N:M sparsity 기반 SpMM 연산, 벡터의 크기(32, 64)로 사용한 HiNM sparsity기반 SpMM 연산의 지연 시간을 다양한 K값에 따라 나타낸다. 먼저, N:M sparsity에서 지연 시간은 *cuBLAS*에 비해 약 33% 감소하였다. 반면 HiNM sparsity기반 SpMM 연산 시간은 *cuBLAS*에 비해 벡터의 크기가 32일 때는 약 49%, 64일 때는 약 53% 감소하였다. 위 실험을 통해, HiNM sparsity를

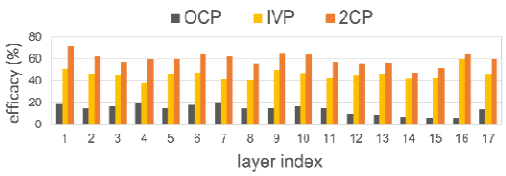

그림 11 channel permutation의 efficacy 비교
Fig. 11 Comparison of channel permutation efficacy

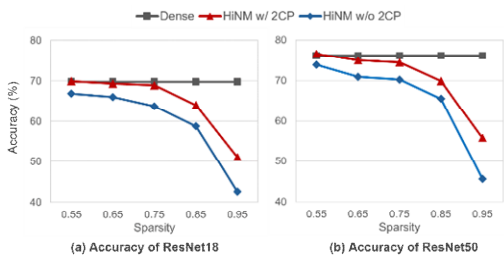

그림 12 ResNet 모델들의 정확도
Fig. 12 Accuracy of ResNet models

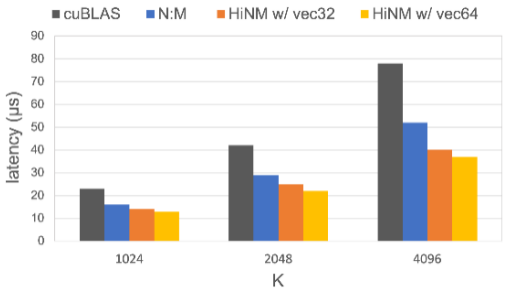

그림 13 계층적 N:M 희소성의 지연 시간
Fig. 13 Latency of hierarchical N:M sparsity

활용하였을 때, 다양한 압축률을 제공하면서 Sparse Tensor Core를 활용할 수 있는 가능성을 제시한다.

6. 결 론

본 논문은 N:M sparsity의 고정된 압축률과 모든 N:M stripe가 uniform sparsity를 가지는 문제점을 hierarchical N:M (HiNM) sparsity와 2-axis Channel Permutation (2CP) 알고리즘을 통해 해결했다. 특히, 소프트웨어 기반 vector index와 하드웨어 기반 NM index를 함께 사용하는 HiNM sparsity를 위한 SpMM 연산을 위한 GPU 커널을 제시하였다.

References

[1] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation." *International Conference on Machine Learning. PMLR*, 2021.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners." *Advances in neural information processing systems*, Vol. 33, pp. 1877-1901. 2020.

[3] Y. Tan, K. Han, K. Zhao, X. Yu, Z. Du, Y. Chen, Y. Wang, and J. Yao, "Accelerating Sparse Convolution with Column Vector-Wise Sparsity." *Advances in Neural Information Processing Systems*, Vol. 35, pp. 30307-30317. 2022.

[4] C. Park, M. Park, H. J. Oh, M. Kim, M. K. Yoon, S. Kim, and W. W. Ro, "Balanced Column-Wise Block Pruning for Maximizing GPU Parallelism." *Proc. of the AAAI Conference on Artificial Intelligence*. Vol. 37. No. 8. 2023.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv*:1510.00149. 2015.

[6] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, "Accelerating sparse deep neural networks." *arxiv preprint arxiv*:2104.08378. 2021.

[7] W. Sun, A. Zhou, S. Stuijk, R. Wijnhoven, A. O. Nelson, H. Li, and H. Corporaal, "DominoSearch: Find layer-wise fine-grained N: M sparse schemes from dense neural networks." *Advances in neural information processing systems*, Vol. 34, pp. 20721-20732. 2021.

[8] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks." *Proc. of IEEE International Conference on Computer Vision*. 2017.

[9] J. Pool, and C. Yu, "Channel permutations for N: M sparsity." *Advances in Neural Information Processing Systems*, Vol. 34, pp. 13316-13327. 2021.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition." *Proc. of the IEEE conference on computer vision and pattern recognition*. 2016.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, G., …S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *In Advances in Neural Information Processing Systems*, Vol. 32, pp. 8024 - 8035. 2019.

유 승 민
2022년 건국대학교 수학과(학사)
2022년~현재 성균관대학교 전자전기컴퓨터공학과 석사과정. 관심분야는 GPU 커널 최적화, 딥 러닝 모델 압축, 딥 러닝 컴파일러

이 하 윤
2017년 성균관대학교 컴퓨터공학과(학사)
2017년~현재 성균관대학교 전자전기컴퓨터공학과 석박통합과정. 관심분야는 Processing-in-Memory, 딥 러닝 컴파일러, 딥 러닝 모델 압축

신 동 군
1994년 서울대학교 계산통계학과(학사)
2000년 서울대학교 전산과학과(석사)
2004년 서울대학교 컴퓨터공학부(박사)
2004년~2007년 삼성전자 소프트웨어연구소 책임연구원. 2007년~현재 성균관대학교 소프트웨어대학 교수. 관심분야는 컴퓨터구조, 운영체제, 머신러닝