Apache TVM 기반 HPC 워크로드의 오토튜닝 성능 최적화 연구

권오경[1]
[1]한국과학기술정보연구원 슈퍼컴퓨팅본부
okkwon@kisti.re.kr

A Study on Auto-Tuning Performance Optimization of HPC Workloads Based on Apache TVM

Oh-Kyoung Kwon[1]
[1]National Supercomputing Center,
Korea Institute of Science and Technology Information

요 약

최근 고성능 컴퓨팅(HPC) 환경에서는 이기종 컴퓨팅 자원의 보편화로 인해 다양한 하드웨어 아키텍처에서의 프로그램 최적화가 필수적이다. 이에 따라 수작업 없이도 하드웨어에 맞춤형 최적화를 자동으로 수행할 수 있는 오토튜닝 기술의 중요성이 커지고 있다. 본 연구에서는 Apache TVM을 활용하여 HPC 워크로드에서의 성능 최적화 가능성을 탐구하였다. 밀집 행렬 연산(GEMM)과 희소 행렬 연산(SpMM)을 주요 워크로드로 설정하고, 국가슈퍼컴퓨터센터의 5호기 누리온에서 실험을 진행하였다. GEMM 연산에서는 누리온 KNL과 Skylake 노드에서 각각 209배, 87배의 성능 향상을 확인하였고, SpMM 연산에서는 BCSR 포맷 기준으로 최대 117배, 10배의 성능 향상이 이루어졌다. 본 연구는 Apache TVM이 인공지능 분야를 넘어 HPC 워크로드에서도 성능 최적화에 유효한 도구임을 시사하며, 향후 다양한 이기종 컴퓨팅 환경에의 적용 가능성을 제시한다.

1. 서론

최근 고성능 컴퓨팅(HPC) 환경은 이기종 컴퓨팅 자원의 보편화로 인해 다양한 하드웨어 아키텍처에서 프로그램을 최적화해야 하는 필요성이 급증하고 있다. 기존의 단일 하드웨어 환경에서 벗어나, CPU, GPU, FPGA 등 여러 컴퓨팅 자원이 혼재된 이기종 환경에서 효율적으로 성능을 발휘하려면 각 자원에 맞춘 맞춤형 최적화가 요구된다. 그러나 이러한 최적화 작업은 고도로 복잡하며, 시간이 많이 소요되기 때문에 수작업으로 이를 수행하기는 매우 어렵다. 이에 따라 이기종 환경에서 자동으로 성능을 최적화해주는 오토튜닝(Auto-tuning) 기술에 대한 연구의 필요성이 갈수록 커지고 있다.

이기종 고성능 컴퓨팅 환경에서 인공지능 분야에서 활발히 사용되고 있는 오토튜닝 플랫폼 중 하나로 Apache TVM이 있다. Apache TVM은 딥러닝 추론 모델을 위한 최적화 컴파일러로서, 모델의 레이어에서 발생하는 행렬 또는 텐서 연산을 자동으로 최적화하는 기능을 제공한다. 2018년 Chen et al.에 의해 처음 소개된 이후, Apache 오픈소스 플랫폼을 기반으로 지속적으로 발전해 왔으며, 최근에는 상용 클라우드 플랫폼에도 널리 적용되고 있다. Apache TVM은 CPU, GPU 등 다양한 계산 환경에서 기존의 수치 및 딥러닝 라이브러리와 동등하거나 이를 능가하는 성능을 목표로 한다. 또한 이기종 시스템에서 각각의 하드웨어에 맞춘 라이브러리를 수동으로 개발하고 최적화하는 부담을 크게 줄여준다.

본 연구에서는 이러한 Apache TVM이 HPC 워크로드에서도 효과적으로 적용될 수 있는 가능성을 탐구하고자 한다. 특히 밀집(dense) 행렬 연산과 희소(sparse) 행렬 연산을 주요 HPC 워크로드로 삼아 Apache TVM을 통해 성능 최적화를 시도할 예정이다. 여기서 밀집 행렬 연산은 밀집 행렬과 밀집 행렬 곱 연산인 GEMM, 희소 행렬 연산은 희소 행렬과 밀집 행렬 곱 연산인 SpMM 연산으로 성능 최적화를 수행한다. 실험은 국가슈퍼컴퓨터센터의 5호기 누리온의 CPU 환경에서 진행되며, 이를 통해 Apache TVM이 인공지능뿐만 아니라 HPC 워크로드에서도 성능을 향상시킬 수 있는지를 검증하는 것을 목표로 한다.

2. Apache TVM에서 HPC 워크로드 최적화

Apache TVM은 다양한 하드웨어에서 효율적으로 연산을 최적화할 수 있도록 고안된 프레임워크이며, 이 과정에서 Tensor Expression을 이용하여 연산을 추상화하고, 이를 바탕으로 다양한 최적화 기법을 적용할 수 있다. 본 절에서는 HPC에서 사용하는 빈번한 두가지 워크로드인 GEMM과 SpMM 연산을 Apache TVM에서 제공하는 Tensor Expression을 이용하여 정의한다. GEMM 연산 관련해서는 여러 가지 성능 최적화 방법들을 살펴보고자 한다. 또한, Apache TVM에서는 특별한 코딩 없이 성능 가속이 가능한 오토튜닝 기능이 있는데 SpMM 연산을 통해 확인해보고자 한다.

2.1 GEMM 연산

GEMM 연산 실험은 1024x1024 사이즈 밀집 행렬과 1024x1024 사이즈 밀집 행렬에 대해서 단일 정밀도(Single Precision) 연산으로 수행한다. Apache TVM은 그림 1과 같이 파이썬 언어를 이용해서 Tensor Expression을 표현한다. 이 코드는 아직 하드웨어에 특화된 최적화가 적용되지 않은 상태이며, 연산 속도는 하드웨어에 따라 최적화되지 않을 수 있다.

Apache TVM은 이렇게 정의된 Tensor Expression을 최적화된 코드로 변환하는 과정을 제공한다. 이를 위해 TVM은 스케줄링(Scheduling)이라는 개념을 사용하여 최적화 기법을 적용한다. 스케줄링 단계에서 TVM은 연산을 하드웨어의 특성에 맞게 효율적으로 처리할 수 있도록 다양한 기법을 사용한다.

본 연구에서 다음 최적화 기법들을 사용하였다.

- 1. Blocking(블로킹): 큰 연산을 작은 블록으로 나누어 캐시를 효율적으로 활용한다.
- 2. Vectorization(벡터화): 데이터를 벡터 단위로 처리하여 많은 데이터를 한 번에 처리할 수 있다.
- 3. Loop Permutation(루프 변형): 루프 구조를 변형하여 메모리 접근에 대한 최적화를 수행한다.
- 4. Array Packing(자료구조 최적화): 다차원 배열 저장 방식을 재정렬하여 이를 1차원 배열에 저장한 후 순차적으로 접근할 수 있도록 하여 메모리 접근 성능을 향상한다.
- 5. Write Cache 최적화: 블로킹 최적화 후 결과 데이터는 블록 단위로 쓰게 되지만, 접근 패턴은 순차적이지 않다. 이때 결과 데이터를 순차적인 캐시 배열을 사용하여 저장하고 성능을 향상한다.
- 6. Parallelization(병렬화): 여러 연산을 동시에 처리할 수 있도록 하여 성능을 극대화한다.

위에서 정의한 Tensor Expression을 최적화 코드로 변환하기 위해 아래 그림처럼 스케줄링 기법을 이용하여 병렬화 및 벡터화를 수행할 수 있다.

```
M = K = N = 1024
dtype = "float32"
dev = tvm.device("llvm", 0) # Random generated tensor for testing
a = tvm.nd.array(numpy.random.rand(M, K).astype(dtype), dev)
b = tvm.nd.array(numpy.random.rand(K, N).astype(dtype), dev)
c = tvm.nd.array(numpy.zeros((M, N), dtype=dtype), dev) # output
# Algorithm
k = te.reduce_axis((0, K), "k")
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
C = te.compute((M, N), lambda m, n: te.sum(A[m, k] * B[k, n], axis=k), name="C")
```

(그림 1) GEMM 연산 Vanilla 코드: baseline

```
s = te.create_schedule(C.op)
mo, no, mi, ni = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(kaxis,) = s[C].op.reduce_axis
ko, ki = s[C].split(kaxis, factor=kfactor)
s[C].reorder(mo, no, ko, ki, mi, ni)
```

(그림 2) GEMM 연산 블로킹 코드: blocking

```
s[C].vectorize(ni)
```

(그림 3) GEMM 연산 벡터화: vectorization

```
# re-ordering
s[C].reorder(mo, no, ko, mi, ki, ni)
s[C].vectorize(ni)
```

(그림 4) GEMM 연산 루프변형: loop permutation

```
k = te.reduce_axis((0, K), "k")
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
# re-write the algorithm slightly.
packedB = te.compute( (N / bn, K, bn),
      lambda bigN, k, littleN: B[k, bigN * bn + littleN],
      name="packedB" )
C = te.compute( (M, N), lambda m, n:
      te.sum(A[m, k] * packedB[n // bn, k,
      tvm.tir.indexmod(n, bn)], axis=k),
      name="C", )
```

(그림 5) GEMM 연산 자료구조 최적화: array packing

```
# Allocate write cache
CC = s.cache_write(C, "global")
mo, no, mi, ni = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
# Write cache is computed at no
s[CC].compute_at(s[C], no)
# New inner axes
mc, nc = s[CC].op.axis
(kaxis,) = s[CC].op.reduce_axis
ko, ki = s[CC].split(kaxis, factor=kfactor)
s[CC].reorder(ko, mc, ki, nc)
s[CC].vectorize(nc)
# unroll kfactor loops
s[CC].unroll(ki)
```

(그림 6) GEMM 연산 Write Cache 최적화: write cache

```
# parallel
s[C].parallel(mo)
```

(그림 7) GEMM 연산 병렬화: parallelization

그림 2에서 split 함수는 블로킹을 적용하는 것이며, 이를 통해 CPU 캐시를 효율적으로 사용할 수 있게 한다. 그림 3에서 vectorize 함수로 데이터를 벡터 단위로 처리하여 계산 효율을 높인다. 그림 4에서 reoder 함수를 통해서 이중 루프의 바같쪽과 안쪽 순서를 바꿔서 메모리 접근 패턴을 최적화한다. 그림 5에서 B의 다차원 배열을 packedB로 저장 방식을 재정렬한 뒤 메모리 접근 성능을 향상한다. 그림 6에서 cache_write와 compute_at 함수를 이용해 결과 데이터를순차적인 캐시 배열에 준비하여 계산한다. 마지막으로 그림 7의 parallel 함수는 병렬화 기법을 적용하여 여러 코어에서 연산을 동시에 처리하도록 한다.

2.2 SpMM 연산

```
M, N, K, density = 512, 512, 512, 0.2
X_np = np.random.randn(M, K).astype("float32")
W_sp_np = sp.random(N, K, density=density, format="csr",
dtype="float32")
W_data = te.placeholder(shape=W_sp_np.data.shape,
dtype=str(W_sp_np.data.dtype))
W_indices = te.placeholder(shape=W_sp_np.indices.shape,
dtype=str(W_sp_np.indices.dtype))
W_indptr = te.placeholder(shape=W_sp_np.indptr.shape,
dtype=str(W_sp_np.indptr.dtype))
X = te.placeholder(shape=X_np.shape, dtype=str(X_np.dtype))
Y = topi.nn.sparse_dense(X, W_data, W_indices, W_indptr)
```

(그림 8) CSR 포맷을 사용한 SpMM 연산을 위한 코드

```
M, N, K, density = 512, 512, 512, 0.2
BS_R = 32, BS_C = 4
W_sp_np = random_bsr_matrix(N, K, BS_R, BS_C,
density=density, dtype="float32")
W_data = te.placeholder(shape=W_sp_np.data.shape,
dtype=str(W_sp_np.data.dtype))
W_indices = te.placeholder(shape=W_sp_np.indices.shape,
dtype=str(W_sp_np.indices.dtype))
W_indptr = te.placeholder(shape=W_sp_np.indptr.shape,
dtype=str(W_sp_np.indptr.dtype))
X = te.placeholder(shape=X_np.shape, dtype=str(X_np.dtype))
Y = topi.nn.sparse_dense_sp_rhs(X, W_data, W_indices,
W_indptr)
```

(그림 9) BCSR 포맷을 사용한 SpMM 연산을 위한 코드

SpMM 연산 실험은 512x512 사이즈 희소 행렬과 1024x1024 사이즈 밀집 행렬에 대해서 Single Precision 정밀도 연산으로 수행한다. 희소행렬 포맷은 CSR과 BCSR를 사용한다[2]. CSR은 가장 보편적으로 사용되는 포맷으로 0이 아닌 실제 데이터와 컬럼(Column) 인덱스와 열(Row)에 대한 포인터를 저장한다. 이때 실제 데이터를 접근하기 위해서는 컬럼 인덱스를 통한 간접 인덱스를 사용해야 하기 때문에 캐쉬 미스가 빈번하게 발생하여 성능저하가 많이 발생한다. 이에 대해서 블락킹 최적화을 통해 캐쉬 미스 발생을 보완한 포맷이 BCSR이다[3].

TVM은 AutoTVM이라는 오토튜닝 프레임워크를 통해 다양한 최적화 공간을 탐색한다. AutoTVM은 수작업으로 최적화를 수행하는 대신, 여러 하드웨어 환경에 맞춰 최적의 스케줄을 찾는 작업을 자동으로 수행한다. AutoTVM은 다양한 하이퍼파라미터(예: 블록 크기, 병렬화 정도, 벡터화 여부 등)를 테스트하여 성능을 평가하고, 가장 적합한 스케줄을 자동으로 선택한다. 여기서 SpMM 연산 실험을 AutoTVM을 이용해서 성능 가속이 어느정도 달성할 수 있는 확인할 예정이다.

3. HPC 워크로드 성능 실험 결과

Apache TVM은 다양한 하드웨어에서 최적의 성능을 제공하기 위해 Tensor Expression을 사용하여 연산을 추상화하고, 스케줄링을 통해 하드웨어에 맞춘 최적화 기법을 자동으로 적용한다. 본 절에서는 Apache TVM을 통해 HPC 워크로드에 대한 최적화 방법을 탐구하며, 이를 통해 국가센터5호기 누리온에서 성능을 극대화할 수 있는 가능성을 확인하고자 한다.

3.1 GEMM 연산 실험

<표 1> 누리온 Intel KNL에서 GEMM 실험 (단위 sec.)

최적화 방법	최적화 아키텍처 옵션	
	llvm	llvm -mcpu=KNL
numpy with MKL	0.108827	0.108827
바닐라(baseline)	20.471076	22.43128
blocking	0.879994	0.54048
vectorization	0.924878	0.514267
loop permutation	0.593474	0.19186
array packing	0.51566	0.165799
write cache	0.50015	0.106985
parallel	0.557257	0.138088

<표 2> 누리온 Intel Skylake에서 GEMM 실험 (단위 sec.)

최적화 방법	최적화 아키텍처 옵션	
	llvm	llvm -mcpu=core-avx2
numpy with MKL	0.010268	0.010268
바닐라(baseline)	3.33175	3.473729
blocking	0.294072	0.316435
vectorization	0.323944	0.289974
loop permutation	0.115781	0.082009
array packing	0.108054	0.083326
write cache	0.092516	0.039868
parallel	0.012113	0.04408

2.1절에서 소개된 최적화 방법을 이용하여 GEMM 연산 실험을 누리온 KNL 노드(표 1)와 Skylake 노드(표 2)에서 각각 수행하였다. 또한 Apache TVM 내 device 함수를 이용하여 하드웨어별 최적화 옵션을 다르게 설정해 비교하였다. 성능 가속화의 상대적 비교를 위해 Intel MKL로 컴파일된 최적화된 numpy 라이브러리로 동일한 연산을

수행하였다.

누리온 KNL과 Skylake 노드에서 바닐라 코드 대비 각각 최대 209배, 87배의 성능 향상이 확인되었다. 최종 가속화는 Intel MKL 수치 라이브러리와 유사한 결과임을 확인할 수 있었다. 또한 하드웨어 최적화 옵션(mcpu)을 추가하여 수행한 경우, KNL 및 Skylake의 AVX-512 명령어군을 활용할 수 있어 벡터화가 조금 더 잘 이루어졌음을 확인할 수 있었다. 블로킹 최적화를 통해 이미 벡터 최적화가 컴파일러 수준에서 자동으로 이루어졌기 때문에, 명시적으로 벡터화를 한 경우에 비해 매우 큰 성능 향상은 없었다. 성능 향상에 가장 큰 영향을 미친 방법은 블로킹을 통한 벡터화였으며, 그다음으로 중요한 요소는 메모리 레이아웃을 변경해 메모리 접근성을 향상시키는 루프 변형 최적화 방법이었다.

3.2 SpMM 연산 실험

<표 3> 누리온 Intel KNL에서 SpMM 실험 (단위 sec.)

	BCSR		CSR	
	llvm	llvm -mcpu=core-avx2	llvm	llvm -mcpu=core-avx2
Baseline	126.146	123.62	144.571	144.582
Tuned	42.362	11.974		

<표 4> 누리온 Intel Skylake에서 SpMM 실험 (단위 sec.)

	BCSR		CSR	
	llvm	llvm -mcpu=core-avx2	llvm	llvm -mcpu=core-avx2
Baseline	28.19	27.693	29.995	29.891
Tuned	0.646	0.235		

2.2절에서 소개된 최적화 방법을 이용하여 SpMM 연산 실험을 누리온 KNL 노드(표 3)와 Skylake 노드(표 4)에서 각각 수행하였다. CSR 포맷은 AutoTVM 내 오토튜닝이 구현되지 않아 실험이 불가능했으며, BCSR 포맷만으로 오토튜닝 성능 테스트를 진행할 수 있었다.

BCSR 포맷을 기준으로 최적화되지 않은 코드(baseline)와 비교했을 때, 누리온 KNL과 Skylake 노드에서 오토튜닝을 통해 각각 최대 약 117배, 10배의 성능 향상이 확인되었다. CSR 포맷을 BCSR 포맷으로 변환했을 때, 최적화되지 않은 코드는 각각 1.17배, 1.08배의 성능 향상만 있었는데, 이는 블로킹을 통한 최적화가 이루어지지 않아 벡터화가 제대로 이루어지지 않았기 때문이다.

4. 결론 및 향후 연구

본 연구를 통해 Apache TVM이 고성능 컴퓨팅(HPC) 워크로드에서도 효과적으로 성능 최적화를 제공할 수 있음을 확인하였다. 특히, 밀집 행렬 연산(GEMM)과 희소 행렬 연산(SpMM)을 대상으로 한 실험에서 각각의 성능이 크게 향상되었으며, 오토튜닝 기능을 통해 수작업 없이도 각 하드웨어에 맞춘 최적화가 가능하다는 점이 입증되었다. 누리온 KNL과 Skylake 노드에서의 성능 향상은 각각 최대 209배와 87배에 달했으며, BCSR 포맷을 활용한 SpMM 연산에서도 117배, 10배의 성능 향상이 확인되었다. 이러한 결과는 Apache TVM이 제공하는 최적화 기법이 HPC 환경에서도 유효하게 작동할 수 있음을 보여준다.

또한, 최적화 과정에서 블로킹 및 벡터화가 중요한 성능 향상 요소로 작용하였으며, 메모리 접근성을 개선하는 루프 변형 최적화 역시 긍정적인 영향을 미쳤다. 본 연구는 Apache TVM의 오토튜닝 기술이 인공지능뿐만 아니라 다양한 HPC 워크로드에서도 성공적으로 적용될 수 있음을 시사한다.

향후 연구에서는 GPU 환경에서의 성능 최적화를 중점적으로 탐구할 계획이다. Apache TVM은 현재 CPU 기반 최적화에서 우수한 성능을 보였지만, GPU와 같은 병렬 처리에 강한 아키텍처에서도 최적의 성능을 발휘할 수 있을지에 대한 연구가 필요하다. 이를 통해 다른 이기종 환경에서 Apache TVM이 성능 향상을 극대화할 수 있는 방법을 모색하고, 더 다양한 HPC 워크로드에 적용할 수 있는 범용적인 최적화 솔루션으로 발전시킬 계획이다.

감사의 글

이 성과는 과학기술정보통신의 재원으로 한국연구재단의 지원을 받아 수행된 연구입니다 (RS-2023-00283799).

참고문헌

[1] Tianqi Chen, et al. "TVM: an automated end-to-end optimizing compiler for deep learning ", OSDI'18: Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation

[2] Sun, H. et al. Selective Protection for Sparse Iterative Solvers to Reduce the Resilience Overhead. SBAC-PAD, Porto, Portugal, September 2020.

[2] Asanovic, K. et al. A view of the parallel computing landscape. Commun. ACM 2009, 52, 56 - 67.