[2024-1]

KEASE 기술서

한국형 엑사스케일 응용 SW 개발 환경 프레임워크

GEMM 커널 최적화

2024. 04. 20

숭 실 대 학 교

[개 정 이 력]

개정 일자	버전	설명	개정페이지	작성자	확인자
2024.4.20	1.0	INITIAL DRAFT	전체	정에녹	최재영

목 차

1. 연구의 개요

GEMM optimization 연구에서는 임의의 프로세서에 최적화된 GEMM 커널 루틴 자동 생성 알고리즘을 연구함. 본 세부 연구의 산출물은 아키텍처 추상화를 통한 GEMM 커널 루틴 최적화 알고리즘 및 해당 알고리즘을 통해 생성한 GEMM 커널 루틴이며, 기술의 편의를 위하여 해당 알고리즘이 KNL 프로세서 및 ARM 기반 프로세서에 적용되는 과정을 서술함. 첫 번째로, 단일 코어 환경에 맞춘 최적화 알고리즘 적용 과정을 기술하고 생성된 루틴의 성능을 비교함. 두 번째로, 개발된 루틴을 멀티 코어 환경에 맞추어 확장하고 성능을 비교함. 마지막으로, 이전 세부 연구에서 파악한 ExaMPM의 특징을 활용한 추가적인 GEMM 커널 루틴 최적화 방향성을 검토함.

2. GEMM 커널 최적화

1) GEMM 커널 연구 배경

GEMM 커널을 자동으로 생성해야 하는 이유는 다음과 같음.

(a) GEMM 커널을 생성하는 시간을 절약함. 본 연구개발의 2단계를 고려하면 GPU에 타겟팅한 최적화된 GEMM 커널 루틴을 개발해야 하며, 이를 위해 GPU 아키텍처에 대한 높은 이해가 필요함. 다만 GPU의 세부 아키텍처는 GPU 벤더에서 전혀 공개하지 않으므로, 통상적으로 CPU 제조사에서 제공하는 아키텍처 메뉴얼 수준의 정보를 얻는 것이 불가능함. 프로그램의 실행 결과를 통해 역으로 아키텍처를 유추하여 GPU 아키텍처를 분석하는 리버스 엔지니어링 연구가 활발히 진행되고 있으나, 슈퍼컴퓨터 6호기에 사용될 GPU 아키텍처에 관한 연구가 진행되었는지 확신할 수 없기 때문에 본 연구에서는 리버스 엔지니어링을 수행해야 할 것으로 예상됨. 이에는 많은 시간이 소요되므로, 자동 생성을 통해 6호기 CPU에 루틴을 최적화하는 시간과 노력을 줄이도록 함.

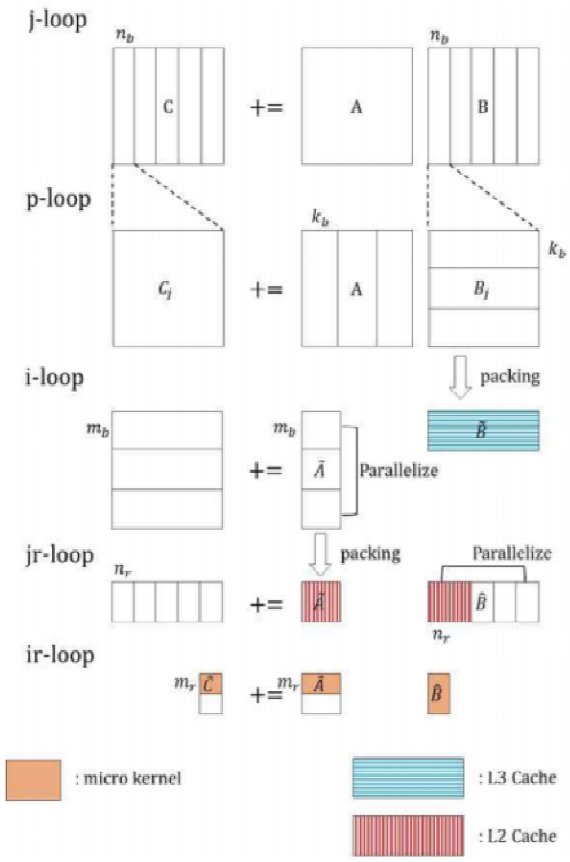
(b) 프레임워크 지원 범위를 확대함. 프레임워크 지원에는 하드웨어뿐만 아니라 소프트웨어적인 지원도 포함함. 위 연구에서 개발 중인 프레임워크는 Kokkos를 기반으로 하므로, 응용 소프트웨어가 최적으로 동작하기 위해서는 해당 응용 소프트웨어가 Kokkos로 작성되어야 함. GEMM 커널 루틴의 자동 생성 모듈은 Kokkos와 독립적으로 활용할 수 있는 여지가 많으므로 Kokkos로 이식되지 않은 응용 소프트웨어일지라도 높은 성능을 보장할 수 있음.

(c) 생성해야 하는 GEMM 커널 루틴의 수가 수동으로 생성하기에는 지나치게 많을 수 있음. 응용의 종류가 많아진 만큼 행렬 곱셈에서 수행할 행렬의 크기 조합도 다양하며, 행렬의 크기가 작으면 작을수록 해당 행렬 크기에 최적화된 독립적인 GEMM 커널 루틴의 필요성이 높아짐. 행렬의 M, N, K가 20 이하인 경우를 위해서는 DGEMM 루틴만 80개가 필요하며, sparse matrix의 경우 데이터 저장 포맷마다 별도의 루틴이 필요함. 최적화 수준을 높이기 위해서는 자동 최적화 기법을 연구하여 각각의 경우에 최적화된 루틴을 자동으로 생성해야만 함.

GEMM 커널 루틴 최적화 연구는 1단계 목표에 따르면 KNL 시스템에서만 수행하는 것이나, 자동 생성 알고리즘이 다수의 아키텍처 기반에서도 높은 성능을 보장함을 보이기 위해 다른 CPU 시스템에서도 연구를 진행하였음. 총 2종류의 CPU 기반 시스템에서 진행하였는데, 하나는 슈퍼컴퓨터 5호기에서 사용되는 프로세서 중 하나인 Intel KNL이고, 다른 하나는 ARMv8 아키텍처 기반 프로세서인 Marvell

ThunderX2임.

행렬 곱셈 (GEMM: GEneral Matrix-matrix Multiplication) 루틴은 Goto 알고리즘으로 구현됨. Goto 알고리즘은 현대 컴퓨터 아키텍처의 메모리 계층 구조를 고려하여 데이터의 재사용성을 높여 행렬 곱셈을 빠르게 수행할 수 있도록 작성된 알고리즘임. 이 알고리즘은 행렬 A, B, C를 블록으로 분할 및 패킹하여 C = α • A • B + β • C 연산을 수행하는 식으로 데이터의 재사용성을 높임. 행렬을 블록 단위로 분할하고, 분할된 행렬을 연산에 용이하도록 데이터의 순서를 변경하고, 해당 행렬을 이용하여 실제 행렬 곱셈 연산을 수행하는 과정으로 나뉨. 이때, 데이터의 순서를 변경하는 작업을 packing이라고 하며, 실제 행렬 곱셈 연산을 수행하는 작업은 micro-kernel에서 수행됨. Micro-kernel은 행렬 곱셈 처리에 유리하도록 변경된 데이터를 이용해 실제 곱셈 연산을 수행해야 하므로, 높은 성능을 위해 숙련된 개발자가 아키텍처 특성을 고려하여 어셈블리어로 작성해야 함.



[그림 1] Goto 알고리즘으로 구현한 GEMM 루틴

Goto 알고리즘을 활용한 GEMM 커널 루틴 최적화의 핵심은 아키텍처에 맞는 적절한 블록 크기 및 아키텍처에 최적화된 micro-kernel 설정임. 특히, micro-kernel은 GEMM 커널 루틴 수행 시간의 95% 이상을 차지하기에, micro-kernel 코드는 숙련된 개발자가 직접 작성하는 것이 일반적일 만큼 최적화 중요도가 높음.

2) 단일 코어 기반 GEMM 커널 최적화

GEMM 커널 루틴 자동 생성을 위해서는 최적화된 micro-kernel 코드의 자동 생성이 핵심이라고 볼 수 있으며, 우선적으로 최적화된 micro-kernel 코드 자동 생성 알고리즘을 제안 및 그 성능을 측정하였음. 최적화된 micro-kernel 코드 자동 생성 알고리즘의 핵심 아이디어는 다음과 같음. "CPU 아키텍처의 복잡성이 높아지고 각 아키텍처마다 가지는 기능이 크게 다르기에, micro-kernel도 아키텍처의 기능을 온전히 활용하도록 보조하는 기능이 선택적으로 추가될 수 있어야 함."

일반적으로 micro-kernel은 오버헤드를 낮추기 위하여 아래 그림과 같이 최대한 간단한 구조를 가지도록 작성되지만, 역으로 micro-kernel이 복잡한 구조를 갖도록 하면 아키텍처의 특성을 더욱 활용하게 되어 성능이 더 높아질 수 있음을 연구를 통해 확인하였음.

Load $\hat{C}$ into r_C
for $i = 0, ..., k_b/k_r - 1$ **do**
  Load $\hat{A}_i$ into r_A
  Load $\hat{B}_i$ into r_B
  $r_C := r_A \times r_B + r_C$
end for
Store r_C into $\hat{C}$

[그림 2] micro-kernel 프로그램 작성 예

Micro-kernel이 CPU 아키텍처의 여러 특성을 활용할 수 있도록 하는 구조를 가지도록 확장하였으며, micro-kernel을 정형화하여 머신러닝을 수행할 수 있도록 하였음. 구체적으로는 '아키텍처적인 특성을 활용할 수 있도록 하는 micro-kernel 코드 최적화 알고리즘의 적용 수치'를 기준으로 micro-kernel을 정형화하였음. 여기서는 micro-kernel 코드 최적화 알고리즘으로 총 4종류의 알고리즘을 활용할 수 있도록 결정하였으며, 해당 4가지 최적화 알고리즘은 [그림 3]과 같음.

Load $\hat{A}_0$ into r_{A_0}
Load $\hat{B}_0$ into r_{B_0}
Load $\hat{C}$ into r_C
for $i = 0, ..., \frac{k_b/k_r}{2} - 1$ **do**
  Load $\hat{A}_{2i+1}$ into r_{A_1}
  Load $\hat{B}_{2i+1}$ into r_{B_1}
  $r_C := r_{A_0} \times r_{B_0} + r_C$
  Load $\hat{A}_{2i+2}$ into r_{A_0}
  Load $\hat{B}_{2i+2}$ into r_{B_0}
  $r_C := r_{A_1} \times r_{B_1} + r_C$
end for
Store r_C into $\hat{C}$

Load $\hat{A}_0$ into r_{A_0}
Load $\hat{B}_0$ into r_{B_0}
Load $\hat{C}$ into r_C
for $i = 0, ..., \frac{k_b/k_r}{2} - 1$ **do**
  Load $\hat{A}_{2i+1}$ into r_{A_1}
  Load $\hat{B}_{2i+1}$ into r_{B_1}
  $r_C := r_{A_0} \times r_{B_0} + r_C$
  Load $\hat{A}_{2i+2}$ into r_{A_0}
  Load $\hat{B}_{2i+2}$ into r_{B_0}
  $r_C := r_{A_1} \times r_{B_1} + r_C$
end for
Store r_C into $\hat{C}$

Prefetch $\hat{C}$ into L1 cache
$r_C := O$
for $i = 0, ..., k_b/k_r - 1$ **do**
  Load $\hat{A}_i$ into r_A
  Load $\hat{B}_i$ into r_B
  $r_C := r_A \times r_B + r_C$
end for
Load $\hat{C}$ into $r_{C'}$
$r_C := r_C + r_{C'}$
Store r_C into $\hat{C}$

Load $\hat{C}$ into r_C
for $i = 0, ..., k_b/k_r - 1 - \alpha$ **do**
  Load $\hat{A}_i$ into r_A
  Load $\hat{B}_i$ into r_B
  $r_C := r_A \times r_B + r_C$
end for
for $i = k_b/k_r - \alpha, ..., k_b/k_r - 1$ **do**
  Prefetch $\hat{A}\_\text{next}_i$ into L1 cache
  Prefetch $\hat{B}\_\text{next}_i$ into L1 cache
  Load $\hat{A}_i$ into r_A
  Load $\hat{B}_i$ into r_B
  $r_C := r_A \times r_B + r_C$
end for
Store r_C into $\hat{C}$

[그림 3] 4가지 최적화 알고리즘

(a) Prefetching matrices: 루프 내에서 추후에 사용할 블록 행렬 A, B를 prefetch하는 최적화 알고리즘임. 블록 행렬 A, B의 load latency를 줄일 수 있으나, prefetch instruction으로 인한 오버헤드 증가 및 cache eviction으로 인한 딜레이가 생길 수 있음.

(b) Preloading matrices: 다음 루프에서 사용할 블록 행렬 A, B를 미리 로드하는 최적화 알고리즘임. 디펜던스로 인한 stall을 줄일 수 있으나, 레지스터 개수 감소로 인한 데이터 재사용성 하락이 생김.

(c) Loading C matrix after multiplication: 블록 행렬 A, B의 곱셈 연산 수행 이후 C를 로드하는 최적화 알고리즘임. C 행렬의 load latency를 숨길 수 있으나, 가용할 수 있는 캐시 크기가 감소함.

(d) Prefetching for next micro-kernel: 이후 수행될 micro-kernel에서 필요한 블록 행렬 A, B 데이터를 prefetch하는 최적화 알고리즘임. 블록 행렬 A, B의 일부 데이터 load latency를 줄일 수 있으나, 코드 길이가 2배로 증가하여 L1i cache miss 발생 확률이 높아짐.

위 알고리즘들은 장단점을 가지므로, 해당 알고리즘의 적용 여부 및 적용의 정도를 수치로 표현하는 방식으로 micro-kernel의 구조를 정형화하였음. 또한, 블록 행렬 A, B, C의 크기와 같이 구조와는 별개로 micro-kernel을 표현할 수 있는 파라미터를 추가하였음. 최종적으로 정형화된 micro-kernel을 구조체 형태로 표현하면 [그림 4]와 같음.

throughout the micro-kernel	target_SIMD_ISA	AVX-512/NEON/SVE/...
	major_A	Row/Col
	major_B	Row/Col
	major_C	Row/Col
	leading_dimension_A	int/ProvidedAsParameter
	leading_dimension_B	int/ProvidedAsParameter
	leading_dimension_C	int/ProvidedAsParameter
	reg_major_C	Row/Col
	k_b	int/ProvidedAsParameter
inside of nano-kernel	m_r	int
	n_r	int
	k_r	int
between nano-kernel	apply_prefetching_A	int
	apply_prefetching_B	int
	apply_preloading_A	int
	apply_preloading_B	int
outside of nano-kernel	apply_load_C_after_mul	T/F
between micro-kernel	prefetch_for_next_micro_kernel	T/F
	preload_for_next_micro_kernel	T/F

[그림 4] 정형화된 micro-kernel을 구조체 형태

3. 단일 코어에서 실험을 통한 테스트

해당 micro-kernel 구조체의 필드를 튜닝 파라미터로 설정하고, 생성된 micro-kernel의 속도를 score로 설정하여 오토 튜닝을 수행함. Intel KNL 시스템 및 Arm ThunderX2 시스템에서 수행하였으며, 환경 조건은 [표 1]과 같음. 오토 튜닝 과정 중 micro-kernel 생성 과정에서 정상적인 micro-kernel을 생성할 수 있도록 target_SIMD는 각 아키텍처에 맞는 값으로 고정하였으며, 그 외의 값은 오토 튜너 과정에서 최적으로 설정하였음.

[표 1] GEMM 커널 생성 환경 설정

	KNL	ThunderX2
compiler	icc 2021.3.0	armclang 22.1
m	600	1000
n	600	1000
k	600	1000
major	column-major	row-major
TransA	No	No
TransB	No	No

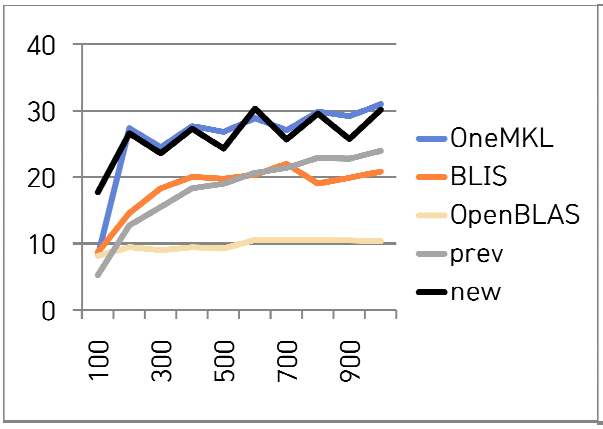
단일 코어 GEMM 성능 테스트오토 튜닝으로 생성된 GEMM 커널을 기존 본 연구실에서 보유하던 수작업으로 최적화한 GEMM 커널과 구조 및 성능을 비교하였음. 기존 GEMM 커널 및 각 환경에서 생성된 GEMM 커널의 micro-kernel 필드 값은 [표 2]와 같음.

[표 2] 기존 GEMM 커널 및 각 환경에서 생성된 GEMM 커널의 micro-kernel 필드 값

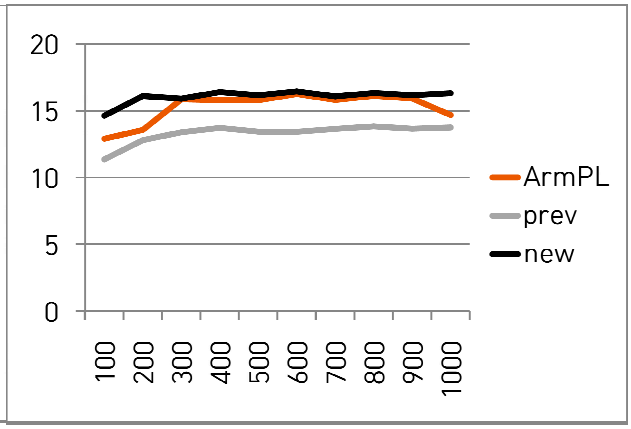
	KNL (prev)	KNL (tuned)	ThunderX2 (prev)	ThunderX2 (tuned)
target_SIMD	AVX-512	AVX-512	NEON	NEON
major_A	Col	Col	Col	Col
major_B	Row	Row	Row	Row
major_C	Col	Col	Row	Row
leading_dimension_A	8	ProvidedAsParameter	6	4
leading_dimension_B	31	24	8	8
leading_dimension_C	ProvidedAsParameter	ProvidedAsParameter	ProvidedAsParameter	ProvidedAsParameter
reg_major_C	Col	Col	Row	Row
kb	>= 600	64	189	208
mr	8	8	6	4
nr	31	24	8	8
kr	1	1	1	1
prefetch_A	Yes (dist=10)	Yes (dist=12)	Yes (dist=20)	No
prefetch_B	Yes (dist=28)	No	Yes (dist=20)	Yes (dist=7)
preload_A	No	Yes (dist=1)	No	Yes (dist=1)
preload_B	No	No	No	Yes (dist=1)
load_C_after_mult	No	Yes	No	Yes
prefetch_for_next_mk	No	No	No	Yes (dist=12)
preload_for_next_mk	No	No	No	No

필드 값의 유의미한 변화가 관측된 지점을 형광색으로 강조하였음. KNL 환경의 기존 GEMM 커널(2번째 열) 및 오토 튜닝으로 생성된 GEMM 커널(3번째 열)을 비교하였을 때 유의미한 차이를 보이는 필드는 5종임. 그 중, preload_A 및 load_C_after_mult 필드의 변화를 보아, Preloading A matrix 및 Loading C matrix after multiplication 최적화 알고리즘이 KNL에서 유의미하게 작용함을 확인할 수 있음. prefetch_B 필드가 No로 변화한 점을 통해 Prefetch 명령어가 B 행렬의 load latency를 hiding하는 것에 전혀 도움을 주지 못했었다는 사실을 알 수 있음. 이와 유사하게, ThunderX2에서도 Preloading matrix 및 Loading C matrix after multiplication 최적화 알고리즘이 유효함을 알 수 있음. Prefetching for next micro-kernel 최적화 알고리즘이 ThunderX2에는 적용되었으나 KNL에는 그렇지 않다는 점은 각 아키텍처마다 적용해야 하는 최적화 알고리즘이 다를 수 있다는 점을 강조함.

위 GEMM 커널의 단일 코어 성능을 측정하였음. 실험은 [표 1] GEMM 커널 생성 환경 조건에서 행렬의 크기를 변경하며 진행하였음. KNL 및 ThunderX2에서의 성능은 각 [그림 5] 및 [그림 6]과 같음. 각 CPU에서 총 5종의 GEMM 커널 간 성능 비교를 수행하였음. 타사 수치 라이브러리로 OneMKL, ArmPL, BLIS, OpenBLAS를 비교군으로 두었으며, 본 연구실에서 보유한 기존 GEMM 커널(prev) 또한 비교군으로 두어 코드 단위의 명확한 비교를 수행할 수 있도록 하였음.



[그림 5] KNL - GEMM 커널 단일 코어 성능 (Gflops)



[그림 6] ThunderX2 - GEMM 커널 단일 코어 성능 (Gflops)

본 연구 결과를 통해 생성된 GEMM 커널은 타 CPU 벤더의 수치 라이브러리 성능과 범줄 만하며, 아직 개선의 여지가 존재함. 우선, 오토 튜너의 전역 최적해 탐색 과정을 검증할 수 있음. 오토 튜너로 생성한 GEMM 커널의 행렬 곱셈 결과 및 속도 검증은 연구원이 직접 수행하나, 오토 튜너가 의도대로 작성되었는지에 대한 검증은 수행하지 않았음. 현재 micro-kernel 최적화 알고리즘으로 추가한 Prefetching for next micro-kernel이 위 두 개의 시스템에서 모두 유의미한 성능향상을 내지 못하는 원인이 단순히 해당 최적화 알고리즘이 해당 시스템에서 효율적이지 못하기 때문인지, 아니면 오토 튜너 작성 과정에서 발생한 실수로 인해 해당 최적화 알고리즘의 성능이 저평가된 까닭인지 검증하고자 함. 또한, 타겟 아키텍처 범위를 넓혀 오토 튜닝 알고리즘에 대한 신뢰성을 높이는 작업이 필요함.

4. 멀티 노드로의 확장

단일 코어 환경의 GEMM 커널 최적화의 핵심은 micro-kernel 루틴의 최적화이고, 멀티 코어 환경의 GEMM 커널 최적화의 핵심은 각 코어로의 작업 분배 최적화임. 멀티 코어 환경에서는 메모리 bandwidth로 인한 딜레이, 스레드 간 작업 동기화로 인한 딜레이, 코어 간 데이터 전달 과정에서 발생하는 딜레이, NUMA 환경에서 remote 메모리 접근으로 발생하는 딜레이를 추가로 고려해야 함. 다만 블록 크기 및 작업 분배 방식에 따른 각 딜레이의 변화량을 예측하는 연구는 아직 진행중임. 다르게 말하자면, 어떠한 작업 분배 방식 및 데이터 공유 알고리즘이 최상의 GEMM 커널을 도출하는지 파악하지 못함. 당장은 최적의 파라미터를 찾기 위해 휴리스틱 서치에 의존하고 있으며, 이는 추후 개선할 예정임.

현재는 휴리스틱 서치를 사용하여 최적의 파라미터를 탐색하기 때문에, 데이터 공유 및 작업 분배 알고리즘을 [그림 7]과 같이 정형화하였음. Blocked matrix multiplication 알고리즘을 코드 상에 적용하여 코어 간 공유하는 데이터의 크기를 블록 크기로 조절할 수 있도록 하였고, C 행렬을 CM * CN개 블록 행렬으로 분해한 후 해당 블록 행렬에 대한 곱셈 작업을 CM * CN개의 코어에 분배하여 작업 분배 방식을 static job scheduling 내에서 부분적으로 정형화하였음. 또한, 작업 실행 루틴 내부에서 쓰레드 간 명시적인 데이터 공유 여부 및 쓰레드 간 작업 동기화 여부를 결정할 수 있도록 하였음. 병렬화 루틴의 정형화된 표현은 아래 [표 3]과 같음. 해당 방식은 dynamic job

scheduling 혹은 batched job scheduling을 수행하지 못하기 때문에, micro-kernel 구조를 확장한 것과 같이 이 또한 추후 구조를 확장하여 추가적인 최적화를 노릴 수 있음.

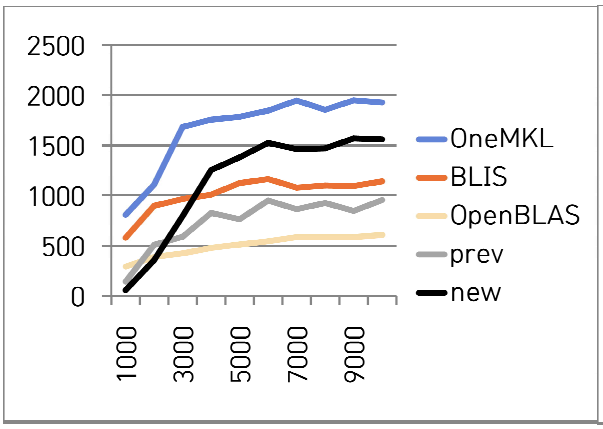
for $\{i, j, k\} = 0 \rightarrow \{M, N, K\}$ step $= \{m_a, n_a, k_a\}$ **do**
  Divide C matrix into $C_M \times C_N$ submatrices
  Create $C_M \times C_N$ jobs corresponding to submatrices
  Distribute each job to $C_M \times C_N$ cores
  Start parallel jobs
  Wait until all jobs end
end for

[그림 7] GEMM 작업 분배에 사용된 알고리즘

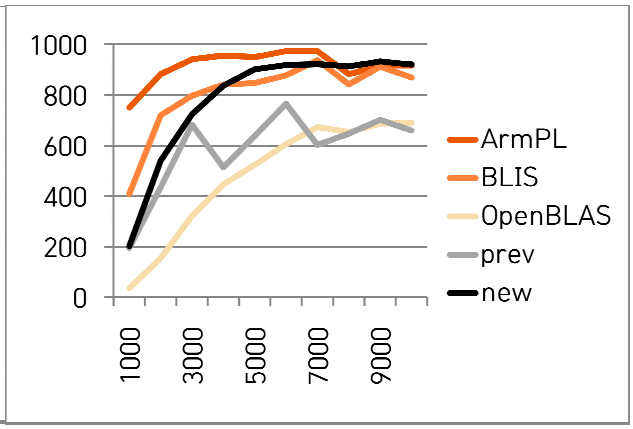
[표 3] 병렬화 루틴의 정형화된 표현

packing	m_r	int
	n_r	int
	m_b	int
	n_b	int
	k_b	int
	inner_loop_order	IJ/JI
	middle_loop_order	IJK/IKJ/JIK/JKI/KIJ/KJI
job_distributing	m_size_per_job	int
	n_size_per_job	int
	k_size_per_job	int
	row_threads	int
	col_threads	int
	share_A_B	Never/OnSameNumaNode/Always
	sync_between_threads	Never/OnSameNumaNode/Always

이를 기반으로 병렬화 기능이 추가된 GEMM 커널 루틴을 개발하였으며, 해당 루틴의 성능을 측정하였음. 병렬화 루틴으로 인한 튜닝 파라미터가 추가되어 일반적인 휴리스틱 서치로 탐색하기에는 탐색 범위가 지나치게 넓어졌기에 micro-kernel의 특징을 고려하여 특정 탐색 범위는 후보에서 제외하였음. 실험은 위 단일 코어 성능 실험 조건에서 OpenMP의 OMP_NUM_THREAD를 추가하여 코어 개수를 명시하였음.



[그림 8] KNL - GEMM 커널 멀티 코어 성능 (Gflops)



[그림 9] ThunderX2 - GEMM 커널 멀티코어 성능 (Gflops)

단일 코어 및 멀티 코어 성능을 통해 신규 개발한 GEMM 커널을 평가하면 다음과 같음. 해당 커널은 단일 코어에서는 CPU 벤더의 수치 라이브러리 성능과 범줄 수 있음. Micro-kernel 최적화 알고리즘 구조 추가를 통해 각 아키텍처의 특징을 활용할 수 있도록 한 점이 성능 향상의 핵심 요소임. 기존 prev와 비교하여 블록 크기와 같은 기존에도 활용하던 튜닝 파라미터는 큰 변화가 없었으나 성능은 크게 향상된 것을 통해 유추하였음. 멀티 코어에서의 GEMM 커널 성능은 타 오픈 소스 라이브러리보다는 명백히 높은 성능을 보이고 있으나, OneMKL 및 ArmPL과 비교하면 부족한 성능을 보임. 멀티 코어에서의 튜닝 성능이 낮다는 것을 알 수 있음. 단일 코어 대비 멀티 코어에서 약세인 원인은 현 GEMM 커널 구조가 멀티 코어에 최적화될 수 있는 형태가 아니기 때문임. Micro-kernel 구조 확장을 통해 단일 스레드 기준 CPU 아키텍처적인 특징을 활용할 수 있도록 하였으나, 행렬 분할 및 패킹 과정에서의 구조 확장은 이루어지지 않았기에 멀티 스레드 기준 CPU 및 메모리의 아키텍처적인 특징을 온전히 활용할 수 없었음.

현재는 병렬화가 하나의 blocked matrix multiplication 단계에 대해서만 이루어졌기에, nested된 병렬화 구조에서의 스레드 팀 간 데이터 공유를 통한 메모리 오버헤드 감소 혜택을 누리지 못함. 멀티 코어에서의 GEMM 최적화 연구를 추가로 진행한다면 단일 코어에서와 같이 멀티 코어에서도 타사의 수치 라이브러리 성능을 능가할 것으로 기대함.