특집 01

AVX-512를 활용한 인텔 차세대 프로세서에서의 효과적인 프로그래밍 방법

최재영 · 김래현 · 임록택 (숭실대학교)

목 차

1. 서 론

마이크로프로세서의 클럭 증가에 따른 전력 소모와 그에 따른 발열에 대한 제약으로 인해 고성능 컴퓨팅 아키텍처는 점차 낮은 클럭의 다수 코어로 이루어진 칩 병렬화 구조로 변화되고 있다. 이런 추세에 발맞춰 인텔은 2001년 DP (Dual processor) 및 MP (Multi processor, 4개 이상) 구조의 서버용 Intel Xeon 프로세서를 시작으로 2017년 현재 최대 24개 코어 프로세서까지 출시하였다. 뿐만 아니라 2012년 Many Integrated Core (MIC) 구조의 Xeon Phi Knights Corner (KNC)를 시작으로 2016년 Xeon Phi Knights Landing (KNL)을 출시하였다. 인텔 Xeon Phi 프로세서는 최대 72개의 코어로 구성되어 있으며, Vector Processing Unit (VPU)을 통해 최대 512-bit 길이의 강력한 벡터 연산 기능을 지원한다. 2017년에 출시된 Intel Xeon Skylake Scalable Processors (Skylake-SP)도 Advanced Vector Extensions 512 (AVX-512) 명령어를 지원한다.

인텔 차세대 아키텍처의 성능을 최대한 이끌어내기 위해서는 대상 프로세서의 구조를 정확히 파악하고, 이를 바탕으로 효과적인 알고리즘을 구현해야 한다. 벡터 연산 기능을 효과적으로 활용하기 위해 벡터화가 용이한 형태로 알고리즘을 수정함과 동시에 Single Instruction Multiple Data (SIMD) 명령어를 적극적으로 사용해야 한다. 또한 다중 레벨 캐시 구조에 맞게 캐시 재활용을 최대화하여 주된 성능 저하 요인인 대역폭을 최소화하여야 한다. 뿐만 아니라 환경 변수와 컴파일 옵션 등 추가적으로 고려해야 할 요소가 다수 존재한다. 일반 사용자에게 있어 이러한 요소들을 모두 고려하여 프로그램을 작성하고 최적화하는 것은 매우 어려울 뿐 아니라 많은 시간을 소요하게 된다.

본 논문에서는 일반 행렬 곱셈 알고리즘의 구현 결과를 바탕으로 인텔 차세대 아키텍처를 효

과적으로 사용하는 방법을 제시하고자 한다. 2장에서는 인텔 Xeon Phi Knights Landing (KNL) 프로세서의 하드웨어, 소프트웨어적 특성에 대해 설명한다. 3장에서는 인텔 차세대 아키텍처를 효과적으로 활용하기 위한 Advanced Vector Extensions 512 (AVX-512) 명령어 집합을 소개하고, 배정밀도 행렬곱셈루틴인 DGEMM (Double precision GEneral Matrix Multiply) 알고리즘 구현 결과를 바탕으로 최적의 캐시 활용법과 병렬화에 관련된 환경 변수 설정에 대해 설명한다. 마지막 4장에서는 연구 내용을 종합한 결론으로 마무리한다.

2. 인텔 차세대 매니코어 프로세서의 구조

인텔 Xeon Phi Knights Landing (KNL) 프로세서 7250은 68개의 1.4GHz 코어들로 이루어져 있는 매니코어 프로세서 아키텍처이다. 총 34개의 타일이 2D 매쉬 구조로 연결되어 있는 형태이며, 각 타일은 2개의 코어와 4개의 VPU들로 구성되어 있다. 타일 내부의 2개 코어는 개별적으로 32KB 크기의 L1 캐시를 가지고 있으며, 1MB 크기의 L2 캐시를 공유한다. 각 코어에는 2개의 VPU가 있어 32개의 512-bit 벡터 레지스터를 제어하며 각 VPU는 사이클마다 16개의 단정밀도 연산 혹은 8개의 배정밀도 연산을 수행한다.

KNL의 가장 중요한 특징 중 하나는 DDR4 SDRAM과 Multi-Channel DRAM (MCDRAM)으로 구성된 계층적 메모리 구조이다. KNL은 on-package 메모리로 최대 450GB/s의 대역폭을 가지는 16GB 크기의 MCDRAM를 제공하며, platform 메모리로 최대 90GB/s의 대역폭을 가지는 DDR4 SDRAM 6개 슬롯을 제공한다. MCDRAM은 용도에 따라 캐시 모드, 플랫 모드, 하이브리드 모드의 세 가지 메모리 모드로 활용할 수 있다. 본 연구에서는 MCDRAM을 직접적으로 제어하기 위해 플랫 모드로 구동하였다.

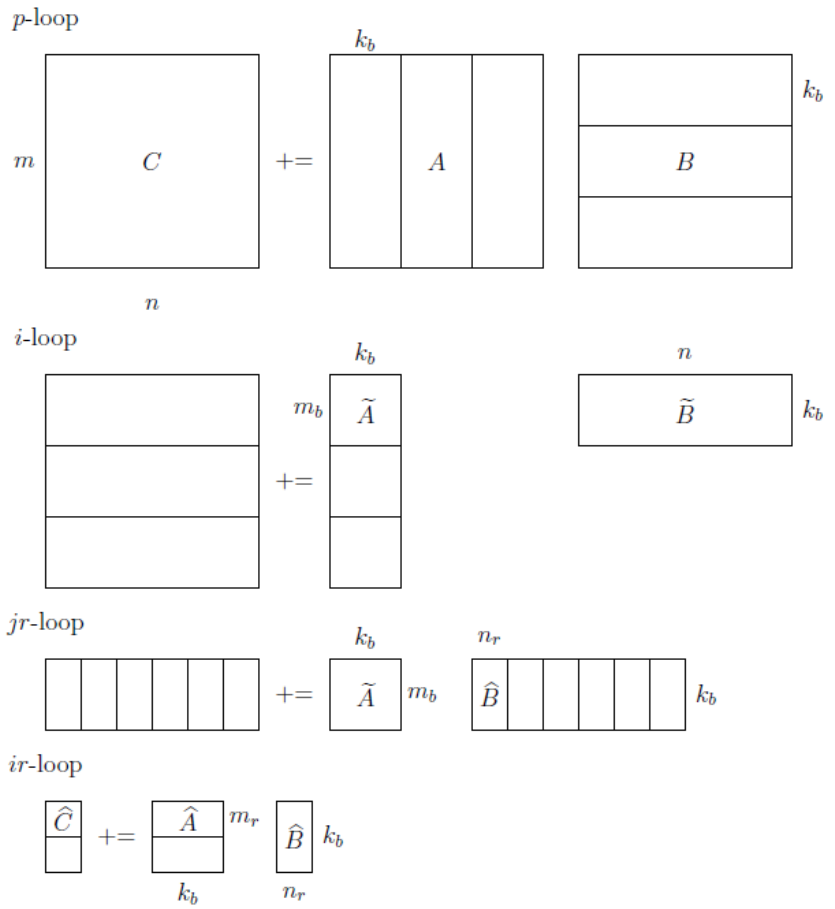
KNL은 독립적으로 부팅가능한 첫 제온 파이 시리즈 프로세서로 CentOS 계열 OS를 지원한다. 본 연구는 CentOS 버전 7.2.1511 (커널 3.10.0-327.13.1)의 OS에서 진행되었다. GEMM 알고리즘의 성능을 비교하기 위해 Intel Math Kernel Library (MKL)과 BLAS-like Library Instantiation Software(BLIS)[8]의 두 BLAS 라이브러리를 활용하였다. MKL 버전은 Intel Parallel Studio XE 2017 Update 1, BLIS 버전은 0.2.2이다.

3. 최적화 방법

본 논문에서는 인텔 차세대 매니코어 프로세서 KNL에서 DGEMM 알고리즘을 구현하고 그 결과를 바탕으로 최적 변수 탐색에 대한 기준을 제시하고 환경 변수 설정과 병렬화 효율성에 대해 설명한다. 분석을 위해 구현한 DGEMM 알고리즘은 배정밀도 밀집행렬간 곱셈 알고리즘으로 바깥 부분의 블록화된 행렬 곱셈 영역와 실제 연산을 수행하는 고효율 내부 커널로 이루어져 있다. 내부 커널은 AVX-512 명령어 집합을 사용

〈표 1〉 Blocked GEMM 알고리즘의 매개변수 설명

매개변수	설명
m	행렬 A, C 의 행 개수
n	행렬 B, C 의 열 개수
k	행렬 A의 열과 행렬 B의 행 개수
k_b	p-loop의 블록킹 매개변수
m_b	i-loop의 블록킹 매개변수
n_r	jr-loop의 블록킹 매개변수
m_r	ir-loop의 블록킹 매개변수



(그림 1) Blocked GEMM 알고리즘

하여 구축하였으며, C 언어를 사용해 블록화된 행렬 곱셈 영역을 구현하였다.

블록화된 행렬 곱셈은 그림 1과 같은 형태로 진행되며, 그림 1에 사용된 매개 변수들은 표 1에 설명되어 있다. 내부 커널은 그림 1의 ir-loop에서 $\hat{C}+=\hat{A}\times\hat{B}$ 연산 부분을 담당한다.

3.1 AVX-512 명령어 집합

KNL은 각 코어의 벡터 레지스터를 조작하기 위해 SSE, AVX 등 여러 명령어 집합를 지원한다. 그 중 Advanced Vector Extensions 512 (AVX-512) 명령어 집합은 가장 강력한 벡터 기반 연산 명령어 집합으로 인텔 Xeon Phi 시리즈와 인텔 Xeon Scalable 프로세서를 포함한 인텔 차세대 아키텍처에서 지원된다. AVX-512 명령어는 어셈블리로 구성된 약간 더 높은 수준의 매크로로, 레지스터 주소를 통한 명시적 할당이 필요한 어셈블리 언어와는 달리 일반적인 변수를 활용할 수 있다. 즉, 일반적인 사용자의 경우 일반적인 함수와 같은 형태로 제공되는 AVX-512 명령어 집합을 활용함으로써 쉽고 빠르게 어셈블리 언어 만큼의 계산 효율을 가져갈 수 있다.

AVX-512 명령어의 기본 연산 단위는 512-bit

길이의 캐시 라인으로 8개의 배정밀도 혹은 16개의 단정밀도 연산이 한 번에 이루어진다. 따라서 AVX-512 명령어를 효과적으로 활용하려면 대상 알고리즘을 512-bit 캐시 라인에 맞게 수정할 필요가 있다. OMP SIMD directive을 통해 컴파일 과정에서 자동으로 SIMD 명령어를 사용할 수 있지만 직접적인 명령어 활용에 비해 그 성능이 떨어지는 모습을 보인다.

AVX-512 명령어 집합에는 단일 곱셈-누산기(Fused multiply-add)를 포함한 일반적인 산술 명령어와 로드, 스토어, 프리패치와 같은 레지스터 조작 명령어가 포함되어 있다. 사용 가능한 명령어와 작동 기작은 인텔 Intrinsic Guide[4]에서 확인할 수 있다. 그림 2는 AVX-512 명령어를 활용한 단일 곱셈-누산 알고리즘이다. 일반적인 C 코드와 같이 벡터 레지스터 변수를 선언하고 AVX-512 명령어를 활용하여 연산을 수행한다. 그림 3은 그림 2의 코드에 의해 실제로 생성되는 어셈블리 코드이다. AVX-512 명령어와 어셈블리 명령어가 일대일로 대응되는 모습을 확인할 수 있다.

AVX-512 명령어를 활용하여 내부 커널을 구성하여 실험한 결과 [6], 어셈블리어로 구현된 인텔 MKL DGEMM 커널에 버금가는 계산 성능을 보일 수 있다.

3.2 레지스터 및 캐시 사용 기준

KNL의 각 코어에는 32개의 512-bit 벡터 레지스터가 존재한다. 레지스터 사용량은 내부 커널의 구조가 결정하는데, $\hat{C} += \hat{A} \times \hat{B}$ 연산에서 레지스터에 $m_r \times n_r$ 크기의 $\hat{C}$와 n_r 길이의 $\hat{B}$의 한 행이 저장된다. 한 벡터 레지스터에는 8개의 배정밀도 수가 저장되므로 레지스터 사용량 num_r은 다음과 같은 식으로 계산할 수 있다.

```
register __m512d a, b, c;
a = _mm512_load_pd(A);
b = _mm512_load_pd(B);
c = _mm512_load_pd(C);
c = _mm512_fmadd_pd(a, b, c);   // c += a * b
_mm512_store_pd(C, c);
```

(그림 2) AVX-512 명령어를 활용한 단일 곱셈-누산 알고리즘

```
vmovaps          256(%rsp), %zmm1
vmovaps          320(%rsp), %zmm2
vmovaps          384(%rsp), %zmm3
vfmadd231pd      %zmm1, %zmm2, %zmm3
vmovaps          %zmm3, 384(%rsp)
```

(그림 3) 그림 2의 코드로 생성된 어셈블리 코드

$$num_r = (m_r + 1) \times \left\lceil \frac{n_r}{8} \right\rceil$$

최적의 레지스터 사용량에 대해서는 전체를 사용해야 한다는 주장과 50%만 사용해야 한다는 주장이 있다. Goto와 van de Geijn[1]은 프리패치시킨 데이터를 저장할 공간을 확보해야 한다는 이유로 50%의 사용을 주장하였다. 반면에 Jaffers et al.[5]은 전체 벡터 레지스터를 활용해야 높은 성능을 얻을 수 있다고 주장하였다. 뿐만 아니라 Heinecke et al.[3]은 이전 세대의 인텔 매니코어 아키텍처 나이츠 코너(KNC)에서 모든 벡터 레지스터를 활용해 내부 커널을 구성하였으며, 그 결과 이론적 최대 성능의 약 90%까지 얻었다. 우리는 KNL에서 위의 모든 경우를 적용하여 커널을 구성하여 실험하였으며, 모두 사용하는 경우에 50%만 사용하는 경우보다 더 좋은 결과를 얻었다[6]. 뿐만 아니라 언롤링을 적용하여 레지스터 사용량을 높일수록 성능이 향상되는 것을 확인하였다.

다음으로 캐시 사용량은 각 캐시에 들어가는 블록의 크기가 결정하는데 L2 캐시의 경우 i-loop의 $\widetilde{A}, \hat{B}, \hat{C}$ 블록들이 들어가며, L1 캐시의 경우 jr-loop의 $\hat{A}, \hat{B}, \hat{C}$ 블록들이 들어간다. 다만 L1 캐시의 경우 캐시 만료가 가능하다는 전제 하에 $\hat{B}$ 블록의 크기만 고려할 수 있으나 KNL의 경우 캐시 만료 명령어를 지원하지 않기 때문에 $\hat{A}, \hat{B}, \hat{C}$ 블록들 모두의 크기를 고려하였다. 즉 L1, L2 캐시 사용량 $usage_{L1}, usage_{L2}$은 다음과 같은 식으로 나타낼 수 있다.

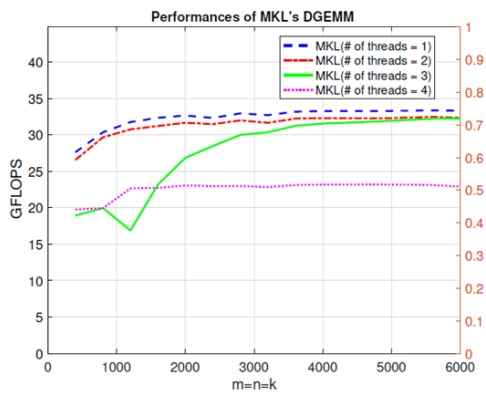
$$usage_{L1} = (m_r \times k_b + n_r \times k_b + m_r \times n_r) \times (8bytes)$$
$$usage_{L2} = (m_b \times k_b + n_r \times k_b + m_r \times n_r) \times (8bytes)$$

캐시 활용 관련 논의의 핵심 쟁점은 크게 두 가지로 좁힐 수 있다. 첫째는 각 레벨 캐시의 최적 사용량 문제이다. Gunnels et al.[2]은 각 캐시 레벨에서 이루어지는 행렬 곱셈의 구성요소 중에서 가장 큰 블록의 크기가 해당 캐시 크기의 대부분을 차지해야 한다고 주장하였다. 반면 Goto와 van de Gejin[1]은 캐시 방출(cache eviction)를 방지하기 위해 가장 큰 블록의 크기가 캐시 크기의 절반 이하가 되어야 한다고 주장하였다. 우리는 실험을 통해 최적의 L2 캐시 사용량을 탐색하였다[6].

다음으로 L1 캐시 블록킹의 효용성에 대한 논의이다. 종래의 블록킹 기법의 경우 모든 캐시 레벨에 대해 블록킹이 이루어지는 것을 기본 전제로 캐시 사용량, 블록의 형태에 대해 논의하였다. 하지만 최근 KNC[3, 7]나 Loongson 프로세서[9]에 대한 연구에 따르면 계산 성능을 충분히 이끌어내기엔 L1 캐시의 크기가 너무 작기 때문에 L1 캐시 블록킹의 효용이 떨어진다고 한다. 실험을 통해 확인한 결과 L2 캐시의 절반정도를 사용하는 것이 가장 효과적임을 확인할 수 있었으며 그 기준은 다음 식과 같다.

$$(m_b \times k_b + n_r \times k_b + m_r \times n_r) \times (8bytes) \approx 512KB$$

싱글 코어 기준 모든 경우 50%(512KB)를 조금 넘어가는 선에서 가장 높은 결과를 얻을 수 있었고, 650KB를 넘어가는 순간 큰 성능 저하가 나타났다. 즉 L2 캐시의 경우 512KB 기준 그 상하범위에서 최적 매개변수를 탐색하는 것이 효과적임을 확인할 수 있었다. 또한 KNL의 경우 L1 캐시 블록킹의 효용성이 떨어지는 모습을 보였다. 매개 변수 조정을 통해 L1 캐시 재활용 횟수를 늘려가며 성능을 측정한 결과 성능이 감소하는 모습을 확인할 수 있었다.



(그림 4) 코어당 쓰레드 생성량에 따른 MKL 성능 테스트

3.3 환경 변수 설정

KNL의 각 코어는 Hyperthreading을 통해 최대 4개의 쓰레드를 생성할 수 있다. 모든 성능을 이끌어내기 위해선 2개 이상의 쓰레드 생성이 필수였던 이전 프로세서 KNC와는 달리 KNL은 코어당 1개, 2개, 4개의 쓰레드만으로도 최대 성능을 이끌어낼 수 있다[5]. 최적의 쓰레드 생성량은 구현하는 알고리즘마다 차이를 보이므로, 구현 과정 중에 최적의 쓰레드 생성량을 확인하는 것이 중요하다. 그림 4는 쓰레드 생성량을 바꿔가며 MKL DGEMM 커널을 테스트한 결과이고, 다음과 같은 환경 변수를 이용하여 쓰레드 생성을 조절하였다.

```
$ export KMP_AFFINITY = COMPACT
$ export KMP_HW_SUBSET =
'1C,nT' with n=1,2,3,4
```

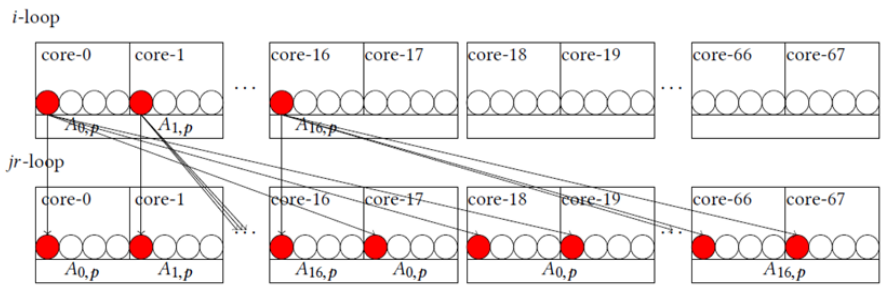
그 결과 MKL DGEMM 커널의 경우 쓰레드 1개만으로도 가장 높은 성능을 얻었다. 이러한 결과를 바탕으로 쓰레드 생성량 기준을 코어당 1개로 하고 DGEMM 알고리즘을 구현하였으며, 최적화 과정 단계마다 최적의 쓰레드 생성량을 확인하였다.

본 연구에서 구현한 DGEMM 알고리즘의 주된 병렬 구간은 i-loop와 jr-loop으로 구성된 nested loop이다. Nested 병렬 구간의 경우 쓰레드를 물리적 코어에 할당하는 방식에 따라 그 병렬화 성능이 크게 변한다. 특히 KNL의 경우 타일 기반 구조로 1MB 크기의 L2 캐시를 한 타일의 두 코어가 공유하는 형태이므로 환경 변수 설정에 보다 민감하다.

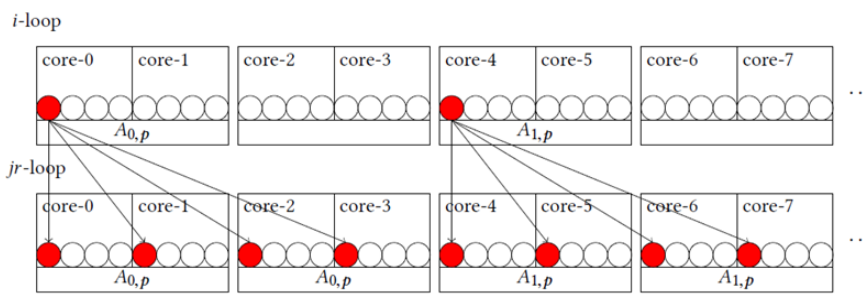
```
$ export KMP_AFFINITY = SCATTER
$ export OMP_PLACES = CORES
```

KNL에서는 다음의 두 환경 변수를 이용하여 쓰레드를 할당한다. KMP_AFFINITY 환경 변수는 OMP_PLACES 환경 변수에 우선하여 적용된다. 만약 두 환경 변수가 동시에 선언될 경우 KMP_AFFINITY 환경 변수 설정에 따라가게 된다. 먼저 KMP_AFFINITY는 인텔 OpenMP 런타임 라이브러리에서 제공하는 환경 변수로 쓰레드를 물리적 코어에 할당하는 부분을 제어한다. 일반적으로 KMP_AFFINITY=SCATTER로 설정하며, 이 경우 각 쓰레드는 최대한 떨어져서 할당되며 모든 코어에 할당되기 전에는 한 코어에 다중으로 할당되지 않는다. 즉 68개의 쓰레드를 생성하면 68개의 코어에 각각 1개씩의 쓰레드가 할당되며, 이후 추가로 생성하는 쓰레드는 앞서 68개의 코어에 할당된 순서를 따라 할당된다.

다음으로 OMP_PLACES는 OpenMP Affinity에서 제공하는 환경변수로 마찬가지로 쓰레드 할당을 담당한다. OMP_PLACES="{0:8:1}"과



(그림 5) KMP_AFFINITY=SCATTER 환경에서의 작업 분배



(그림 6) OMP_PLACES=CORES 환경에서의 작업 분배

같이 명시적으로 할당하거나 OMP_PLACES=CORES와 같이 미리 정의된 규칙을 활용할 수 있다. OMP_PLACES=CORES로 설정할 경우에는 쓰레드를 코어마다 할당하게 된다. 즉 KMP_AFFINITY=SCATTER 설정과 마찬가지로 68개의 쓰레드를 생성하면 68개의 코어에 각각 1개씩의 쓰레드가 할당되게 된다.

68개의 쓰레드를 생성할 경우 KMP_AFFINITY=SCATTER와 OMP_PLACES=CORES가 동일한 CPU map을 갖는 것처럼 보이지만 내포 병렬 구간의 작업 분배에 있어 이 둘은 큰 차이가 있다. 그림 5는 KMP_AFFINITY=SCATTER 환경에서 작업 분배 형태를, 그림 6은 OMP_PLACES=CORES 환경에서 task 분배 형태를 각각 도식화한 그림이다.

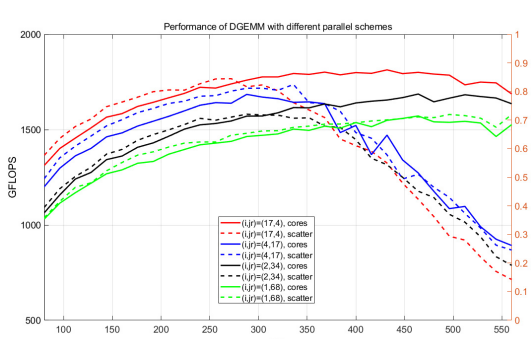
KMP_AFFINITY=SCATTER의 경우 내포 병렬 구간의 바깥 루프인 i-loop에 할당한 쓰레드 개수만큼의 타일에서 작업 수행에 서로 다른 $\widetilde{A}$ 블록을 요구하게 된다. 이러한 작업 분배 방식은 L2 캐시를 비효율적으로 활용하게 하며, 성능 저하를 일으키는 원인이 된다. 반면 OMP_PLACES=CORES의 경우 모든 타일에서 그 내부의 두 코어는 같은 $\widetilde{A}$ 블록을 공유하여 작업을 수행한다. 이 경우 두 코어가 같은 $\widetilde{A}$ 블록을 공유함으로써 타일 내부의 L2 캐시를 보다 효율적으로 활용할 수 있게 된다. 다만 jr-loop에 홀수 개의 쓰레드를 할당한다면 적어도 2개의 타일에서 서로 다른 $\widetilde{A}$ 블록을 요구하게 되어 L2 캐시의 공유 환경을 깨트리게 된다.

본 실험에서는 병렬화 조합 $(i, jr) = (17,4)$, $(4,17)$, $(2,34)$, $(1,68)$에 대해 KMP_AFFINITY=SCATTER, OMP_PLACES=CORES 환경

에서 실험하였으며 고정된 크기의 행렬 곱셈에서 매개변수 k_b만을 변경시키면서 DGEMM 커널의 병렬화 성능을 테스트하였다. 앞서 언급한 바와 같이 다른 변수들이 고정되어 있을 경우 k_b가 L2 캐시 사용량을 결정하게 된다. 따라서 k_b에 따른 성능 변화를 관찰함으로써 각 환경 설정에 따른 L2 캐시 사용 양상을 확인하였다. 실험 결과는 그림 7과 같다. 실선은 OMP_PLACES=CORES 환경에서의 결과이며 점선은 KMP_AFFINITY=SCATTER 환경에서의 결과이다.

먼저 병렬 조합 $(i,jr)=(17,4),\ (2,34)$의 경우 앞서 설명한 바와 같이 OMP_PLACES=CORES 환경에서 L2 캐시를 보다 효율적으로 활용하는 모습을 확인할 수 있다. KMP_AFFINITY=SCATTER 환경에서는 $k_b=256$ 근처에서 가장 좋은 성능을 보이고 이후 성능이 크게 감소하였으나 OMP_PLACES=CORES 환경에서는 $k_b=432$ 근처에서 가장 좋은 결과를 얻었으며 피크 이후 성능 감소도 완만한 모습을 보인다.

반면 병렬 조합 $(i,jr)=(4,17),\ (1,68)$의 경우 두 환경에서의 결과가 크게 차이나지 않다는 것을 확인하였다. 먼저 $(i,jr)=(4,17)$의 경우 jr-loop에 홀수 개의 쓰레드가 할당된 상황이다. 앞서 언급한 바와 같이 이러한 경우 2개의 타일에서 서로 다른 $\widetilde{A}$ 블록을 요구하게 되며, 이는 L2 공유 환경을 해치게 된다. 이 경우 $\widetilde{A}$ 블록 공유 문제가 전체 34개의 타일 중 단 2개에서만 일어났음에도 불구하고 전체 성능에 큰 영향을 끼쳤다. 따라서 내포 구간 병렬화에 있어 쓰레드 할당에 주의하여야 한다. 다음으로 $(i,jr)=(1,68)$로 쓰레드를 할당할 경우 내포 구조가 아니므로 동일한 CPU map을 바탕으로 작업이 분배된다. 따라서 두 환경 사이의 양상이 동일한 모습을 보인다.



(그림 7) 병렬 조합 및 환경 설정에 따른 병렬 성능 비교

4. 결 론

본 논문에서는 AVX-512를 활용하여 구현한 배정밀도 밀집행렬곱셈 알고리즘(DGEMM)을 통해 인텔 차세대 매니코어 프로세서인 나이츠 랜딩(KNL)의 특성을 파악하였다. 그리고 이를 바탕으로 레지스터 및 캐시 활용과 환경 변수 설정에 관한 최적화 방법을 제시하였다. 내부 커널 실험 및 캐시 블록킹 테스트를 통해 KNL에 알맞은 레지스터 및 캐시 활용의 기준을 유도하였다. 이러한 기준을 바탕으로 알고리즘의 설계 및 최적화 과정에서 보다 효율적으로 매개변수를 탐색하고 결정할 수 있을 것이다. 또한 여러 환경 변수에 대한 분석과 테스트를 통해 내포 병렬 구간을 보다 효과적으로 병렬화하기 위한 환경을 제시하였다. 대부분의 과학 계산 알고리즘에 내포 병렬 구간이 존재하는 만큼 보다 효과적인 병렬화가 가능하게 된다. 2017년 3분기에 출시된 Intel Xeon Skylake Scalable Processors도 AVX-512 명령어를 지원하기 시작하였고, 앞으로 출시되는 인텔 프로세서들에서도 이를 활용하여 보다 효과적인 프로그래밍이 가능할 것이다.

감사의 글

본 연구는 2017년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행되었습니다(No. R0117-17-0001).

참 고 문 헌

[1] Goto, K., van de Geijn, R.A. “Anatomy of high-performance matrix multiplication”, ACM Transactions on Mathematical Software (TOMS) 34(3), 12 (2008)

[2] Gunnels, J.A., Henry, G.M., Van De Geijn, R.A. “A family of highperformance matrix multiplication algorithms.”, In: International Conference on Computational Science, pp. 51-60. Springer (2001)

[3] Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A.G., Chrysos, G., Dubey, P. “Design and implementation of the linpack benchmark for single and multi-node systems based on Intel Xeon Phi Coprocessor” In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp.126-137. IEEE (2013)

[4] “Intel Intrinsics Guide.” Software.intel.com. (2018). [online] Available at: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ [Accessed 22 Mar. 2018].

[5] Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann (2016)

[6] Lim, R., Lee, Y., Kim, R., Choi, J. “An Implementation of matrix-matrix multiplication on the Intel KNL processor with AVX-512.” In: Cluster Computing (Submitted)

[7] Peyton, J.L. “Programming dense linear algebra kernels on vectorized architectures.” Master's thesis, The University of Tennessee, Knoxville (2013)

[8] Van Zee, F. G., van de Geijn, R. A. “BLIS: A Framework for Rapidly Instantiating BLAS Functionality” In: ACM Trans. Math. Softw., 41(3), pp.1-33. ACM (2015)

[9] Xianyi, Z., Qian, W., Yunquan, Z. “Model-driven level 3 BLAS performance optimization on Loongson 3A processor” In: Parallel and Distributed Systems, 2012 IEEE 18th International Conference, pp. 684-691. IEEE (2012)

저 자 약 력

최 재 영

이메일 : choi@ssu.ac.kr

- 1984년 서울대학교 제어계측공학과 (공학사)
- 1986년 미국 남가주대학교 전기공학과 (석사)
- 1991년 미국 코넬대학교 전기공학부 (박사)
- 1992년 1월~1994년 2월 미국 국립 오크리지연구소 연구원
- 1994년 3월~1995년 2월 미국 테네시 주립대학교 연구교수
- 1995년 3월~현재 숭실대학교 컴퓨터학부 교수
- 관심분야 : HPC 컴퓨팅, 시스템 소프트웨어, 로봇 미들웨어

김 래 현

이메일 : Kim.rh3169@gmail.com

- 2016년 서울대학교 수학교육과 (학사)
- 2016년 서울대학교 계산과학 연합전공 (학사)
- 2018년 서울대학교 수리과학과 (석사)
- 2018년~현재 숭실대학교 연구원
- 관심분야 : 계산 과학, 수치 해석, 수치 선형대수, 알고리즘

임 록 택

이메일 : rokt.lim@gmail.com

- 2007년 서울대학교 물리천문학부 천문학전공 (학사)
- 2014년 서울대학교 협동과정 계산과학전공 (박사)
- 2014년~2015년 서울대학교 박사 후 연구원
- 2015년~2016년 Nanyang Technological University, 연구원
- 2016년~현재 숭실대학교 연구원
- 관심분야 : 수치해석, 유한요소해법, 고성능계산