과제 #2

M1522.006700 확장형 고성능 컴퓨팅 (001)
M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

박찬정
서울대학교 전기정보공학부
2023-24013

1 Theoretical Peak Performance of CPU

Fig. 와 같이 계산 노드의 CPU 정보를 얻었다.

(a) Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz

(b) 2개의 CPU가 장착되어 있다.

(c) Base clock frequency는 2.10GHz이고, boost clock frequency는 3.20 GHz이다. Base clock과 boost clock은 각각 CPU가 동작할 수 있는 최소, 최대의 clock이며, CPU는 현재 수행중인 workload가 필요로 하는 연산 성능을 판단하여 해당 범위 내에서 clock speed를 조정한다. CPU가 boost clock frequency로 동작하기 위한 조건으로는, 현재 수행중인 workload가 높은 성능을 필요로 하고, 공급되는 전력이 충분하며, CPU의 온도가 너무 높지 않아야 한다.

(d) Physical core는 32개, logical core는 64개이다. Theoretical peak performance 계산을 위해서는 physical core 개수인 32를 사용해야 하며, 그 이유는 logical core는 하나의 physical core가 동시에 여러 thread를 연산할 수 있도록 하는 기술(i.e. Hyper-Threading, SMT 등)을 적용하여 얻어진 가상의 개념이므로, 실제 연산의 성능은 결국 physical core에 달려 있기 때문이다.

(e) Intel(R) Xeon(R) Silver 4216 CPU 는 1개의 AVX512 FMA 유닛을 갖고 있다. 이를 바탕으로 한 clock cycle당 실행되는 AVX512 instruction의 수는 1개라고 가정하자.
AVX512 instruction이 하나당 수행할 수 있는 FP32 연산의 수는 $512/32 = 16$개이다.
따라서 하나의 core가 한 clock cycle에 수행할 수 있는 FP32 연산의 수는 16개이다.

(f) Theoretical peak performance를 다음과 같이 계산할 수 있다.

$$\begin{aligned} R_{peak} &= \frac{(\#\text{ of FP32 operation})}{(\#\text{ of core})(\#\text{ of clock})} \times (\text{boost clock frequency}) \times (\#\text{ of core}) \\ &= 16 \times 3.2 \times 32 \\ &= 1638.4 \text{ (GFLOPS)} \end{aligned}$$

2 Matrix Multiplication using Pthread

- **병렬화 방식.** 주어진 행렬은 row-wise 방식으로 메모리를 사용한다 (바로 다음 메모리가 같은 row의 다음 element). 이를 고려할 때 행렬 B를 쪼개는 것이 행렬 A에 대한 캐시 활용도를 높일 수 있을 것이라고 판단하였다. 이에 각 thread가 A를 모두 사용하고 B의 일부 column을 사용하여 C의 일부 column을 계산하도록 코드를 작성하였다. 계산 중간에 thread 사이의 synchronization은 일어나지 않으며, 각 thread의 계산이 끝나는 시점이 균일하도록 각 thread에게 C의 column을 균등히 할당하였다. 또한, 한 번에 행렬 C의 32×32 만큼을 계산하도록 하는 macro kernel을 정의하여 연산 과정이 cache-friendly 하도록 하였다.

- **Thread 수에 따른 성능 변화.** Thread를 2배씩 증가시키며 실험한 결과[Fig.], GFLOPS는 증가하다가 다시 감소하는 모습을 보였다. 처음에 증가하는 이유는, physical core의 수보다 thread의 수가 적기 때문에 thread가 증가함에 따라 새로운 physical core가 연산에 참여할 수 있게 되기 때문이다. 64개 thread를 넘어서면 다시 감소하는 모습을 보이는데, 이때부터는 logical core의 수보다 thread의 수가 많아지면서 여러 thread가 core를 두고 경쟁하게 되기 때문이며, thread의 수가 더 증가하면서 경쟁에서 발생하는 overhead가 증가하여 성능이 감소하게 된다.

  최고 성능을 보인 thread 수의 근처에서 추가로 실험해본 결과[Fig.], 훨씬 일관되지 않은 모습을 보였다. 그 이유로는 thread의 수가 logical core의 수의 배수가 아닌 경우 경쟁이 더욱 복잡하게 일어난다는 점과 thread의 수에 비례하여 더 치열해지는 경쟁을 들 수 있다.

- **최적의 Thread 수.** 위 실험에서 확인한 바, 64개의 thread를 사용할 때 최고 성능을 보였다. 이는 CPU의 logical core의 수와 같다. 그 이유로 강력히 추측되는 것은, 연산 코드가 충분히 최적화되지 않아 하드웨어를 최대로 사용하지 못하고 있기 때문에, 두 개 thread에서 수행하는 연산들이 interleave될 여지가 매우 많았기 때문으로 보인다. 연산 코드가 최적화됨에 따라 하드웨어를 최대로 사용하게 되면서, 최고 성능을 보이는 thread의 수는 점차 physical core의 수인 32개로 다가갈 것으로 예상한다.

  개선 방법. 더 높은 성능을 위해 적용할 수 있는 방법은 다음과 같은 것들이 있다.

  - 연산을 수행하는 최하단 커널에 loop-unroll을 적용한다. 이를 통해 루프를 수행하기 위해 기본적으로 존재하는 overhead를 효과적으로 줄일 수 있으며, 컴파일러에게 더 많은 최적화 기회를 제공할 수 있다.
  - AVX512 명령어를 사용한다. 위 문제에서 답하였듯이, AVX512 명령어를 사용하면 하나의 명령어로 16개의 FP32 연산을 한 번에 수행할 수 있으므로 성능을 매우 높일 수 있다. 이는 직전에 언급한 loop-unroll 방법의 상위호환이라고 할 수 있다.
  - Macro kernel과 micro kernel을 분리한다. 보통의 GEMM 라이브러리에서는 cache의 사용과 register의 사용을 최적화하기 위해 여러 단계의 kernel을 정의하여 사용한다. 이번 과제에서는 적당한 크기의 macro kernel과 매우 naive한 micro kernel을 사용하였으나, 프로세서의 캐시를 고려하여 macro kernel의 크기를 설정하고, 매우 최적화된 micro kernel을 사용한다면 성능을 매우 높일 수 있다.
  - 연산 이전에, 주어진 행렬을 연산에 최적화된 배열로 buffer에 복사한다. 이를 통해 micro kernel에서 연산을 더욱 효과적으로 수행할 수 있으며, 연산 이전에 데이터에 접근하게 되므로 연산 수행 시에 cache의 hit-rate가 증가한다.
  - 프로세서가 superscalar 방식으로 동작하는 경우, micro kernel에서 memory load 및 prefetch 명령어와 실제 연산 명령어를 적절한 비율로 섞어주는 기법을 사용할 수 있다. 연산을 수행하는 과정에서도 사용이 끝난 레지스터를 다음에 사용할 값으로 미리 채워주거나, 곧 사용하게 될 데이터를 prefetch 하여 캐시에 유지하도록 하는 등의 명령어를 수행해야 한다. 이러한 명령을 실제 연산 명령어 사이에 일정 주기로 삽입하여 memory access latency를 효과적으로 숨길 수 있다.