M3239.005400 데이터사이언스를 위한 컴퓨팅 2 (001)

Homework #2

Due : 2024/10/13 (Sun)

2024-28413 이정현

1. Theoretical Peak Performance of CPU

(a) 실습 서버의 계산 노드에 장착된 CPU 모델명

Answer : Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz

(b) 실습 서버의 계산 노드에 장착된 CPU 개수 (socket 수)

Answer : 2

(c) 실습 서버의 계산 노드에 장착된 CPU base clock frequency / boost clock frequency

Answer :

Base clock frequency : 2.10GHz

Boost clock frequency : 3.20GHz

Base와 Boost 두 종류가 존재하는 이유는 전력 효율성과 성능최적화를 위해서이다.

Boost clock frequency는 굉장히 heavy workload를 다룰 때 사용하게 되는데, 이때 processor의 열 조건이 허용되어야 한다.

(d) 실습 서버의 계산 노드에 장착된 CPU 하나의 physical / logical core 개수

Answer : physical = 16개 **/** logical = 32개

Theoretical peak performance를 위해서는 physical core 개수를 사용해야한다. Logical core 들은 hyper-threading과 같은 기술로 physical core을 늘리는 가상의 개념이기 때문에 몇몇 multi-thread task에 대해서는 성능이 좋아질 수 있지만 실제 CPU의 연산 능력을 향상시키지는 않는다. 즉, 실제 CPU FP32 연산의 성능은 physical core 개수와 연관이 있다.

(e) 실습 서버의 계산 노드에 장착된 CPU에서 AVX512 instruction을 사용하는 경우 하나의 코어가 한 clock cycle에 몇개의 FP32 연산을 수행할 수 있는가

Answer :

Intel(R) Xeon(R) Silver 4216 CPU 는 1개의 AVX512 FMA (Fused Multiply-Add) unit을 가지고 있다.

512비트를 지원하는 AVX512는 16개(= 512/32)의 FP32 숫자를 담을 수 있고, 하나의 clock cycle 당 FMA는 하나의 AVX512 instruction을 실행할 수 있다. FMA는 두개의 연산(multiply, add)을 하는 것으로 계산되기 때문에 16*2 = 32,

즉 하나의 core는 한 clock cycle에 32개의 FP32 연산을 수행한다.

(f) 실습 서버의 계산 노드의 theoretical peak performance를 GFLOPS 단위로 답하라.

Answer : Theoretical peak performance를 구하기 위해서 곱해야하는 값들은 다음과 같다.

-CPU (socket) 개수 : 2

-Socket의 core 개수 : 16

-Base clock frequency : 2.10 GHz

-FP32 operations per clock cycle : 32

$$R_{peak} = 2(sockets) * 16(cores/socket) * 2.10(GHz) * 32(FP32ops/cycle)$$

$$R_{peak} = 2150.4 \text{ GFLOPS}$$

2. Matrix Multiplication using Pthread

-병렬화 방식에 대한 설명

가장 먼저, thread들을 어떤 방법으로 배정할지가 중요했다.이번 과제에서 행렬들은 row-wise (행 우선) 메모리로 사용되고 있었다. 행렬 A와 B가 곱해지는 과정에서 A의 row와 B의 column 사이의 연산이 이루어진다는 사실을 고려했을 때, cache efficiency를 극대화하기 위해서 A의 row들을 전부 thread한테 넘겨주고, B를 부분적으로 나눠서 곱셈결과를 C에 업데이트 해나가는 방법을 선택하였다.

A행렬의 row를 기준으로 thread를 나눴기 때문에, 이번 과제 코드에서 thread 사이의 의존성은 존재하지 않고, synchronization 과정도 필요하지 않다.

이번 과제에서는 3가지 병렬화 방법을 적용해보았다.

(1) Cache Blocking

Cache miss를 줄이기 위해서 각 thread 안에서 blocking 방법을 적용하였다. (BLOCK_SIZE는 128로 설정)

(2) Accumulation buffer for results

위에서 사용한 cache blocking의 효과를 더욱 극대화하기 위해서 [BLOCK_SIZE * BLOCK_SIZE] 크기의 buffer를 사용해서 부분 곱셈 결과를 저장했다가 한번에 C행렬에 쓰는 방법을 선택하였다. 곱셈마다 C행렬에 접근하는 것이 아닌, 블록 단위로 접근하게 되기 때문에 C행렬의 memory access를 효과적으로 할 수 있다는 큰 장점이 있다.

(3) Loop reordering

아래 코드는 실제 블록단위 곱셈결과가 buffer에 작성되는 과정에 대한 코드이다.

```
//access in i->k->j order.
//much more efficient memory access
for(int i=ii; i<max_i; i++){
  for(int k=kk; k<max_k; k++){
    for(int j=0; j< max_j-jj; j++){ //iterate over BLOCK_SIZE
      temp_buff[(i-ii)*BLOCK_SIZE + j] += A[i*K + k] * B[k*N + (jj + j)];
    }
  }
}
```

일반적인 i,j,k순서가 아닌 i,k,j 순서로 루프를 배치하였다. 이 순서를 이용하면 가장 안쪽의 루프가 반복될 때 A행렬이 reference 되는 위치는 바뀌지 않는다는 것을 확인할 수 있다. (A[i*K + k] 는 j 루프에 영향받지 않음)

또한, B행렬도 j 루프 안에서 row를 따라서 연속적으로 access 되고있는 것을 확인할 수 있다. 이 순서대로 짜여진 코드는 Cache miss가 줄어들고, 더욱 효율적이다.

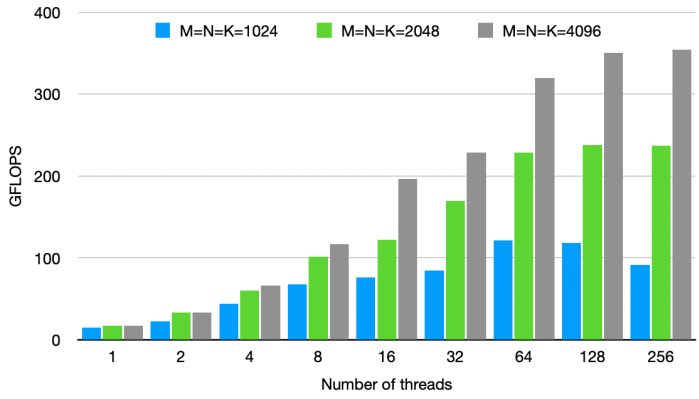
```
int rows_per_thread = M / num_threads;
int start_row_idx = rank * rows_per_thread;
int end_row_idx;

if(rank == num_threads-1){ //last thread. end_row is M.
  end_row_idx = M;
}
else { //not the last thread
  end_row_idx = (rank+1) * rows_per_thread; //set end row to start of next thread.
}
```

```
//set the end row for the blocks. this code handles leftovers as well
int max_i, max_j, max_k;
if(ii+BLOCK_SIZE < end_row_idx){max_i = ii+BLOCK_SIZE;}
else{max_i = end_row_idx;}
if(jj+BLOCK_SIZE < N) {max_j = jj+BLOCK_SIZE;}
else{max_j = N;}
if(kk+BLOCK_SIZE < K) {max_k = kk+BLOCK_SIZE;}
else{max_k = K;}
```

또한, A의 row 개수가 thread 개수의 배수가 아닐때도, 행렬의 차원이 BLOCK_SIZE의 배수가 아닐때도 오류없이 작동할 수 있도록 구현하였다. 이는 ./run_validation.sh의 결과가 모두 VALID 로 출력되는 것으로 확인할 수 있었다.

-스레드를 1개부터 256개까지 사용했을때의 행렬곱 성능 측정

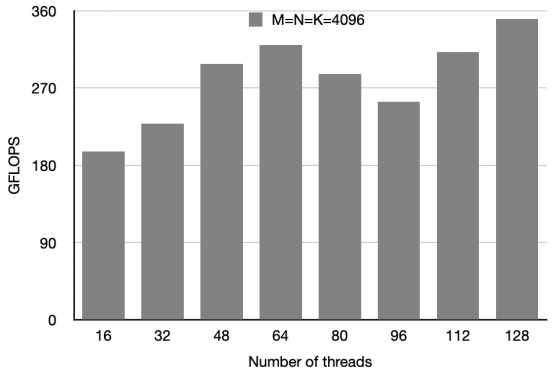


첫 실험으로는 thread의 개수를 2배씩 늘리면서 3가지 크기의 행렬에 대한 곱셈 성능을 측정해보았다. (BLOCK_SIZE는 128로 고정)

1024x1024 크기의 행렬에 대해서는 thread 개수가 64개까지 늘어날때는 GFLOPS가 꾸준히 증가하다가 이후 감소하는 양상을 보였다.

더 큰 크기의 행렬(2048, 4096)에 대해서는 thread 개수가 64개까지 늘어날때는 GFLOPS 증가폭이 매우 컸으나, 이후는 증가폭이 대폭 감소한 모습을 확인할 수 있었다.

우선 thread가 64개까지 늘어날 때 지속적인 성능 향상이 일어났던 이유는 thread의 개수가 core의 개수보다 적기 때문이라고 생각할 수 있다. (하나의 계산노드에는 logical core 64개) 이때는 thread가 늘어나도 바로 사용되지 않는 core를 배정해줄 수 있기 때문에 연산 성능이 무조건 증가하게 된다. 그러나 thread가 64개를 넘어서는 순간, 연산 성능은 감소하거나(1024), 향상 폭이 대폭 감소한다(2048, 4096). Thread가 64개보다 많아지면 여러개의 thread들이 하나의 core을 배정받기 위해서 경쟁하는 상황이 벌어진다. 이 과정에서 이전 상황에는 존재하지 않았던 overhead가 발생하게 된다. 이에 따라서 작은 크기의 행렬에 대해서 연산 성능은 감소하고, 큰 크기의 행렬에 대해서는 성능 증가 폭이 줄어들게 되는 것이다.



두번째 분석으로는 크기가 4096x4096인 행렬에 대해서 thread를 16개씩 증가시킨 그래프를 그려보았다. 눈여겨 볼만한 점은 64~128 구간이다. Thread 개수가 64일때와 128일때만 비교하면 후자의 결과가 더 좋지만, 그 사이 thread 개수들일때 성능이 항상 증가하지는 않는다는 점이다. 이 현상을 유추해보자면, thread의 개수가 계산노드의 logical core(64) 의 배수가 아닌 경우 코어를 배정받기 위한 알고리즘이 더 복잡해 진다는 것을 예상 해볼 수 있었다. (Overhead 증가, 오히려 성능 감소)

-가장 높은 성능을 보이는 스레드 개수에서 행렬곱 성능은 1번에서 계산한 peak performance와 비교했을때 어떤지

M,N,K를 모두 4096으로 설정한 실험에서는 위 그래프와 같이 256개의 thread를 사용하는 것이 가장 높은 성능을 보여주었다.

[*수치상으로는 64개, 128개 thread 사용하였을 때와 엄청 큰 차이는 없음. 위에서 답변한 것 처럼 thread 개수가 64를 넘어가는 순간부터 thread간의 경쟁으로 인한 overhead가 발생하는데, 크기가 큰 행렬에서는 발생하는 overhead보다 연산 성능 향상이 조금 더 커서 결과적으로는 약간의 성능 향상이 일어난 현상으로 보여진다. 문제에서 가장 높은 성능을 보이는 thread 개수와 비교하라고 명시되어 있기 때문에, 256개를 사용한 결과와 비교한다.]

[**이 실험 세팅으로 1024x1024 행렬의 곱셈에서 이미 thread가 64개일 때 최고 성능을 보여주었기 때문에 이보다 작은 크기의 행렬 곱셈에서는 64개 thread를 이용하는 것이 최고 성능을 보여줄 가능성이 높다.]

256개의 thread를 사용해서 4096x4096 행렬들의 곱셈을 할때의 성능은 353.750122 GFLOPS 였다. 1번에서 계산한 peak performance의 16.45% 정도 성능을 보여주었다.

위 코드를 개선하고자 할때 고려할 수 있는 몇개의 사항들은 다음과 같다.

(개선 방법)

1. 행렬 크기에 따른 BLOCK_SIZE 정의.

-현재는 128로 고정한 채 컴파일을 했지만, 곱셈을 하는 행렬 크기에 따라 블록 크기를 유동적으로 조절한다면 더욱 최적화된 cache blocking 방법이 적용된 코드를 실행할 수 있을 것이다. 행렬의 크기와, 행/열 개수의 약수 등을 적절히 사용해서 지정한다면 효과적일 것으로 예상한다.

이는 사용자가 수동으로 수정하거나, 컴파일러 수준에서 가능한 최적화이고, 더 나아가 오토튜닝의 영역에서도 tunable parameter로 볼 수 있는 변수이다.

2. Loop Unrolling

-실제로 곱셈이 이루어지는 루프들 중 가장 안쪽 루프에 대해서 unroll을 시도해볼 수 있다. Unroll의 횟수는 직접 수정하거나, 오토튜닝을 거치는 등의 방법이 있다. Unroll한 코드는 Instruction-level-parallelism 관점에서 컴파일러가 최적화할 수 있는 선택지가 조금 더 많아지는 결과를 가져올 수 있다.

3. Vectorization

-계산노드의 CPU는 AVX512 instruction을 지원하는데, SIMD 연산을 지원하는 함수들을 사용해서 vectorization을 추가할 수 있다. (_mm512_loadu_ps, _mm512_fmadd_ps 등등) 이런 연산들은 한번에 여러 element에 동시 연산을 가능하게 해주기 때문에 병렬화에 많은 도움이 될 것이다.

-정확성 : ./run_validation.sh 10개 모두 VALID

-성능 : ./run_performance.sh 실행결과 234GFLOPS.

```
shpc106@login3:~/skeleton/hw2$ ./run_performance.sh
srun: job 827694 queued and waiting for resources
srun: job 827694 has been allocated resources
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.586688 sec
Calculating...(iter=1) 0.613172 sec
Calculating...(iter=2) 0.580675 sec
Calculating...(iter=3) 0.608844 sec
Calculating...(iter=4) 0.617166 sec
Calculating...(iter=5) 0.625527 sec
Calculating...(iter=6) 0.620722 sec
Calculating...(iter=7) 0.536627 sec
Calculating...(iter=8) 0.512747 sec
Calculating...(iter=9) 0.565740 sec
Validating...
Result: VALID
Avg. time: 0.586791 sec
Avg. throughput: 234.221401 GFLOPS
```