ACCELERATING MATRIX MULTIPLICATION

12194114 구현규

**1. OptimizedMatrixMultiplication 함수 코드**

void OptimizedMatrixMultiplication(int\*\* A,

int\*\* B,

int\*\* C,

int size)

{

int block=128;

int loop=2;

for (int sj = 0;sj < size; sj+=block)

{

for (int si = 0; si < size; si+=block)

{

for (int sk = 0; sk < size; sk+=block)

{

for (int i=si;i<si+block;i+=loop\*16)

{

for (int j=sj;j<sj+block;j++)

{

\_\_m512i c[loop];

for (int x=0;x<loop;x++)

c[x]=\_mm512\_loadu\_epi32(&C[j][i+x\*16]);

for (int k=sk;k<sk+block;k++)

{

\_\_m512i b0=\_mm512\_set1\_epi32(B[j][k]);

for (int x=0;x<loop;x++)

{

c[x]=\_mm512\_add\_epi32(c[x],\_mm512\_mullo\_epi32(\_mm512\_loadu\_epi32(&A[k][i+x\*16]),b0));

}

}

for (int x=0;x<loop;x++)

\_mm512\_storeu\_epi32(&C[j][i+x\*16],c[x]);

}

}

}

}

}

}

**2. 함수 코드를 이용한 사용한 방식에 대한 설명**

먼저, 크게 3가지 기능을 사용. Avx, blocking, loop unrolling.

**2-1. avx**

-작은 사이즈의 연산들을 각각처리 하는 것이 아니라 한번에 커다란 연산들로 처리하는것

Avx512버전을 사용, 가장 집중한 것은 교재에서 사용한 자료형을 integer로 바꾸는데 집중.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

교재의 해당코드의 알고리즘 문제는 크게 없음.(행과 열을 반대로 써 넣은 것만 빼면)

1 .코드가 사용하는 avx256 버전에서 double형이 256bit들어갈 수 있는 최대word 수(256/(doble(8-byte)\*1byte(8bit))=4)를 multiplication크기로 설정

2. 행렬 B[j,k) value를 multiplication크기만큼 벡터 복제 저장한 후

🡸해당 코드에서 broadcast가 수행

3. 행렬 &A[k,i]주소+multiplication크기(4)만큼 데이터를 가져와

🡸해당 코드에서 load가 수행

4. 둘을 곱한 후 &C[j,i]주소+multiplication크기(4)에 넣는 것

🡸해당 코드에서 fmadd,store가 수행

가장 큰 이 매커니즘은 틀린 것이 없었다. 그래서 전체적인 구조를 유지 한 후 integer로 대치 할 수 있도록 집중.

for (int i=si;i<si+block;i+=loop\*16)

🡸avx512에서 integer크기를 고려 (512/(integer(4byte)\*8it)=16 주소+16만큼 데이터를 가져와 멀티계산 이때 i는 16만큼 이동해야 데이터 중복이 사라짐.

{

for (int j=sj;j<sj+block;j++)

{

\_\_m512i c[loop];

for (int x=0;x<loop;x++)

c[x]=\_mm512\_loadu\_epi32(&C[j][i+x\*16]);

🡸int32형자료를 총 16개를 가져와 \_m512i형태의 벡터로 저장. 이과정의 경우 저장하려는 C에 원래 값이 있을 수도 있고 C는 반복해서 저장 데이터를 쌓아 가는 것이기 때문에 따로 변수(c[x])를 두어 계산에 용이성을 제공 .

for (int k=sk;k<sk+block;k++)

{

\_\_m512i b0=\_mm512\_set1\_epi32(B[j][k]);

🡸. 행렬 B[j,k] value를 multiplication크기(16)만큼 벡터 복제하는 과정. 기존에 broadcast가 해당 역할을 수행하던것을 set1으로 대치.

for (int x=0;x<loop;x++)

{

c[x]=\_mm512\_add\_epi32(c[x],\_mm512\_mullo\_epi32(\_mm512\_loadu\_epi32(&A[k][i+x\*16]),b0));

🡸.벡터 계산 위의 fmadd함수가 수행하던 것을 두개의 과정(add,mullo)으로 나누어 A의 자료를 순서대로 16개 가져와 B의 복사된 벡터와 곱연산을 한 후 위에서 미리 준비한 c[x]에 더하여 저장.

}

}

for (int x=0;x<loop;x++)

\_mm512\_storeu\_epi32(&C[j][i+x\*16],c[x]);

🡸.c[x]를 C에 맞는 위치[&C[j][i]]+16만큼의 주소에 맞게 저장

}

}

🡺성능 향상 폭

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Avx하나 만으로도 매우 큰 성능 향상을 체감할 수 있었다. 단순히 멀티계산만 하는 것이 아니라 여러 데이터를 한번에 가져오는 것도 한 몫 하는 것으로 예상.

**2-2. Blocking**

-하나의 matrix를 그대로 계산하는 것인 아닌 여러 개의 block으로 나누어 한 번 데이터를 캐쉬에 불러왔을 때 재사용 확률(miss rate낮춤)을 높인다.

int block=128;

for (int sj = 0;sj < size; sj+=block)

{

for (int si = 0; si < size; si+=block)

{

for (int sk = 0; sk < size; sk+=block)

{

🡸block의 크기에 맞게 matrix를 분할하여 출력하는 loop 당연히 block의 크기만큼 sj,si,sk 의 값이 커져야 함. 사실상 이 기능의 핵심 부분이며 아래의 matrix계산부분에서 시작위치와 끝위치를 주고 받으면 loop 도는 것을 제외하면 이외에 더 크게 코드에 관여하지 않는다

for (int i=si;i<si+block;i+=loop\*16)

{

for (int j=sj;j<sj+block;j++)

{

{

for (int x=0;x<loop;x++)

{

}

}

}

}

🡸위 loop서 분할된 matrix의 시작부분(ex..int i=si)을 제공 받아 block의 크기 (i<si+block)만큼 matrix를 계산하는 코드

🡺성능 향상 폭

기존에서 15-16에서 17-18정도의 향상

Avx하나 만으로도 매우 큰 성능 향상을 하였고 avx스스로도 이미 일정 크기만한 데이터를 불러오는 기능을 수행함으로 체감도가 낮은 성능 향상이었다. 블록의 크기는 128로 결정

**2-3. Loop unrolling**

-Loop를 parallelism하게 expose하여 루프 컨트롤 오버헤드를 줄이는 것.

int loop=2;

for (int i=si;i<si+block;i+=loop\*16)

🡸하나의 loop를 여러 loop로 나눈 만큼 나눈 loop만큼 곱하여 하나의 loop크기만큼 움직일 수 있게 한다.

{

for (int j=sj;j<sj+block;j++)

{

\_\_m512i c[loop];

for (int x=0;x<loop;x++)

c[x]=\_mm512\_loadu\_epi32(&C[j][i+x\*16]);

🡸 나눈 loop만큼 원래 c[x]에 C의 matrix를 한번에 저장하는 것을 loop만큼 나누어 오버헤드를 줄인다(원래 32개의 데이터를 한번에 가져와왔다면 loop로 나누어 계산).

for (int k=sk;k<sk+block;k++)

{

\_\_m512i b0=\_mm512\_set1\_epi32(B[j][k]);

for (int x=0;x<loop;x++)

{

c[x]=\_mm512\_add\_epi32(c[x],\_mm512\_mullo\_epi32(\_mm512\_loadu\_epi32(&A[k][i+x\*16]),b0));

}

}

for (int x=0;x<loop;x++)

\_mm512\_storeu\_epi32(&C[j][i+x\*16],c[x]);

🡸 마찬 가지로 나눈 loop만큼 원래 matrix 계산과 저장 과정을 loop만큼 나누어 오버헤드를 줄인다.

}

}

🡺성능 향상 폭

기존에서 17-18정도에서 19정도로 향상

역시 큰 성능 향상은 없었다. 특히 loop를 2보다 크게 늘려면 오히려 성능이 안 좋아지는 경우가 많았다. 추측하는 바는 이미 한번에 여러 데이터를 가져오는데 이것을 여러 번 나누어 계산할 시 이점보다 코드의 사이즈 증가나 레지스터 사용량이 너무 많아지는 문제가 너무 큰 것 같다.

**3. 결론 및 정리**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

18후반대에서 19초가 주로 성능 기록(최고20.7)

🡺avx를 이용한 계산이 큰 역할

🡺나머지 두 기능은 합쳐서 4배정도의 성능 향상 기여

🡺이를 통해 avx는 높은 확률로 성능 향상에 큰 기여를 하고 나머지 두개는 상황과

적절한 배분이 없으면 오히려 성능 향상에 방해

🡺특히 loop unrolling은 자원과 상황을 적절히 고려해야 한다고 생각된다.