알고리즘\_과제\_4

(mkl을 이용한 matrix multiplication)



학부:컴퓨터학부

학번:20162518

출석번호 : 156번

이름 : 최승서

**<소스코드>**

#define min(x,y) (((x) < (y)) ? (x) : (y))

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include "mkl.h"

/\* Consider adjusting LOOP\_COUNT based on the performance of your computer \*/

/\* to make sure that total run time is at least 1 second \*/

#define LOOP\_COUNT 10

#define N 5000

float randomRange() { // -1~1 사이의 실수값 난수를 생성

return (2 \* ((double)rand() / RAND\_MAX))- 1;

}

void print\_index(double a[], int m, int p){

for(int i=0; i<(m\*p); i++){

if((i%(m)) == 0)

printf("\n");

printf("%.3f\t", a[i]);

}

}

void init\_index(double a[], int m, int n){

for (int i = 0; i < (m\*n); i++)

a[i] = 0.0;

}

int main()

{

double \*A, \*B, \*C;

int m, n, p, i, j, k;

double alpha, beta;

double sum;

double s\_initial, s\_elapsed;

printf ("\n This example measures performance of rcomputing the real matrix product \n"

" C=alpha\*A\*B+beta\*C using a triple nested loop, where A, B, and C are \n"

" matrices and alpha and beta are double precision scalars \n\n");

m = N, p = N, n = N;

printf (" Initializing data for matrix multiplication C=A\*B for matrix \n"

" A(%ix%i) and matrix B(%ix%i)\n\n", m, p, p, n);

alpha = 1.0; beta = 0.0;

printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n"

" performance \n\n");

A = (double \*)mkl\_malloc( m\*p\*sizeof( double ), 64 );

B = (double \*)mkl\_malloc( p\*n\*sizeof( double ), 64 );

C = (double \*)mkl\_malloc( m\*n\*sizeof( double ), 64 );

if (A == NULL || B == NULL || C == NULL) {

printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");

mkl\_free(A);

mkl\_free(B);

mkl\_free(C);

return 1;

}

printf (" Intializing matrix data \n\n");

srand((unsigned)time(NULL));

for (i = 0; i < (m\*p); i++) {

A[i] = randomRange();

}

printf("\n\nA Matrix\n");

//print\_index(A,m,p);

for (i = 0; i < (p\*n); i++) {

B[i] = randomRange();

}

printf("\n\nB Matrix\n");

//print\_index(B,m,p);

init\_index(C,m,p);

printf (" \n\n\nMaking the first run of matrix product using triple nested loop\n"

" to get stable run time measurements \n\n");

for (i = 0; i < m; i++) {

for (j = 0; j < n; j++) {

sum = 0.0;

for (k = 0; k < p; k++)

sum += A[p\*i+k] \* B[n\*k+j];

C[n\*i+j] = sum;

}

}

init\_index(C,m,p);

printf (" Measuring performance of matrix product using triple nested loop \n\n");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*algo 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

s\_initial = dsecnd();

for (i = 0; i < m; i++) {

for (j = 0; j < n; j++) {

sum = C[n\*i+j];

for (k = 0; k < p; k++)

sum += A[p\*i+k] \* B[n\*k+j];

C[n\*i+j] = sum;

}

}

s\_elapsed = (dsecnd() - s\_initial);

printf("\n\n algo\_1 C Matrix\n");

//print\_index(C,m,p);

printf("\n\nalgo 1 속도 : %.lf\n\n", (2\*pow((m\*n),3))/(s\_elapsed \* 1000));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

init\_index(C,m,p);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*algo\_2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

s\_initial = dsecnd();

for(k=0; k<m; k++){

for(i=0; i<n; i++){

sum = A[p\*i+k];

for(j=0; j<p; j++){

C[n\*i+j] += sum \* B[n\*k+j];

}

}

}

s\_elapsed = (dsecnd() - s\_initial);

printf("\n\n algo\_2 C Matrix\n");

//print\_index(C,m,p);

printf("\n\nalgo 2 속도 : %.lf\n\n", (2\*pow((m\*n),3))/(s\_elapsed \* 1000));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

init\_index(C,m,p);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*algo3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

sum=0;

s\_initial = dsecnd();

for(i = 0; i<n; i++){

for(k=0; k<n; k++){

sum =A[k+i\*n];

for(j=0; j<n; j++)

C[j+i\*n] += sum \* B[j+k\*n];

}

}

s\_elapsed = (dsecnd() - s\_initial);

printf("\n\n algo\_3 C Matrix\n");

// print\_index(C,m,p);

printf("\n\nalgo 3 속도 : %.lf\n\n", (2\*pow((m\*n),3))/(s\_elapsed \* 1000));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

init\_index(C,m,p);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*algo 4\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

s\_initial = dsecnd();

cblas\_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, p, alpha, A, p, B, n, beta, C, n);

s\_elapsed = (dsecnd() - s\_initial);

printf("\n\n algo\_4 C Matrix\n");

//print\_index(C,m,p);

printf("\n\nalgo 4 속도 : %.lf\n\n", (2\*pow((m\*n),3))/(s\_elapsed \* 1000));

printf (" Deallocating memory \n\n");

mkl\_free(A);

mkl\_free(B);

mkl\_free(C);

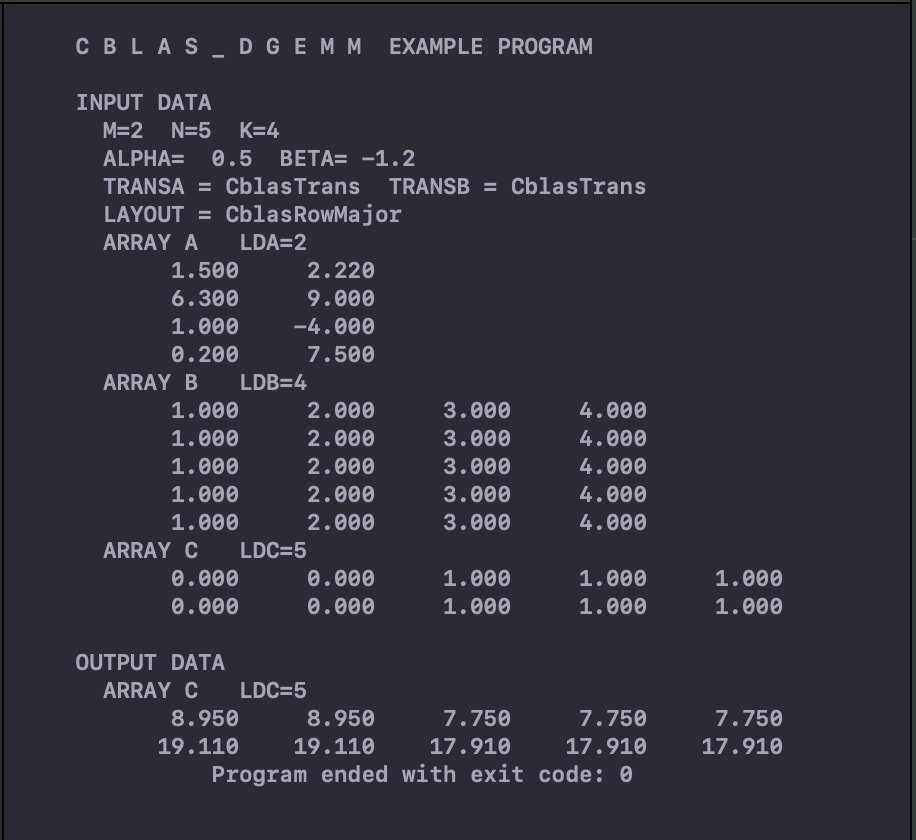
printf (" Example completed. \n\n");

return 0;

}

1.

ppt에 명시된 실행 결과



2.

A) matrix\_multiplication.c에서 i-j-k 루프(내적 계산)으로 행렬 곱셉 루틴이 동작하는 Algo1입니다.

for (i = 0; i < m; i++) {

for (j = 0; j < n; j++) {

sum = C[n\*i+j];

for (k = 0; k < p; k++)

sum += A[p\*i+k] \* B[n\*k+j];

C[n\*i+j] = sum;

}

}

B)ALGO1을 수정하여 k-i-j 루프(외적 계산)으로 동작하는 Algo2를 구현해 보았습니다.

for(k=0; k<m; k++){

for(i=0; i<n; i++){

sum = A[p\*i+k];

for(j=0; j<p; j++){

C[n\*i+j] += sum \* B[n\*k+j];

}

}

}

내적계산으로 동작하는 Algo1과는 다르게 Algo2는 k루프부터 돌기 때문에 외적계산으로 동작하는 구조입니다.

3.

과제에 명시되어 있는 Algo3 코드에 약간의 오류가 존재하여 저의 방식으로 소스코드를 수정하여 알고리즘을 구현해 보았습니다.

for(i = 0; i<n; i++){

for(k=0; k<m; k++){

sum =A[k+i\*n];

for(j=0; j<p; j++)

C[j+i\*n] += sum \* B[j+k\*n];

}

}

또 이 코드에서 i-루프와 j-루프가 바뀌었을때 코드는

for(j = 0; j<p; j++){

for(k=0; k<m; k++){

for(i=0; i<n; i++)

C[k+j\*n] += A[i+k\*n] \* B[j+i\*n];

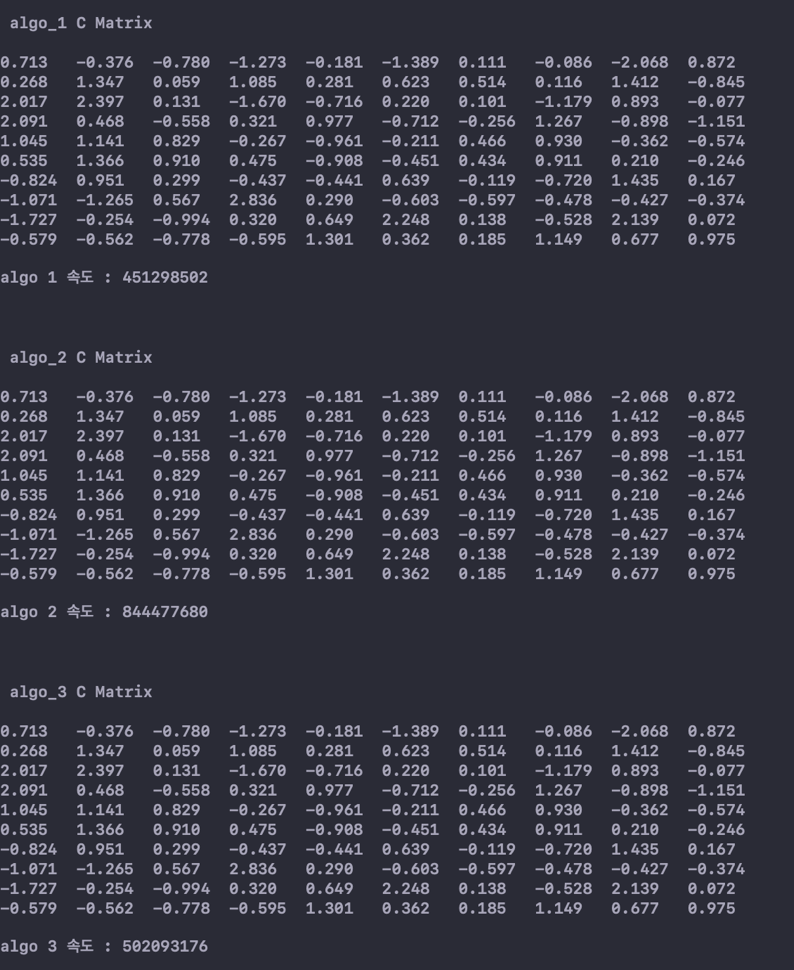
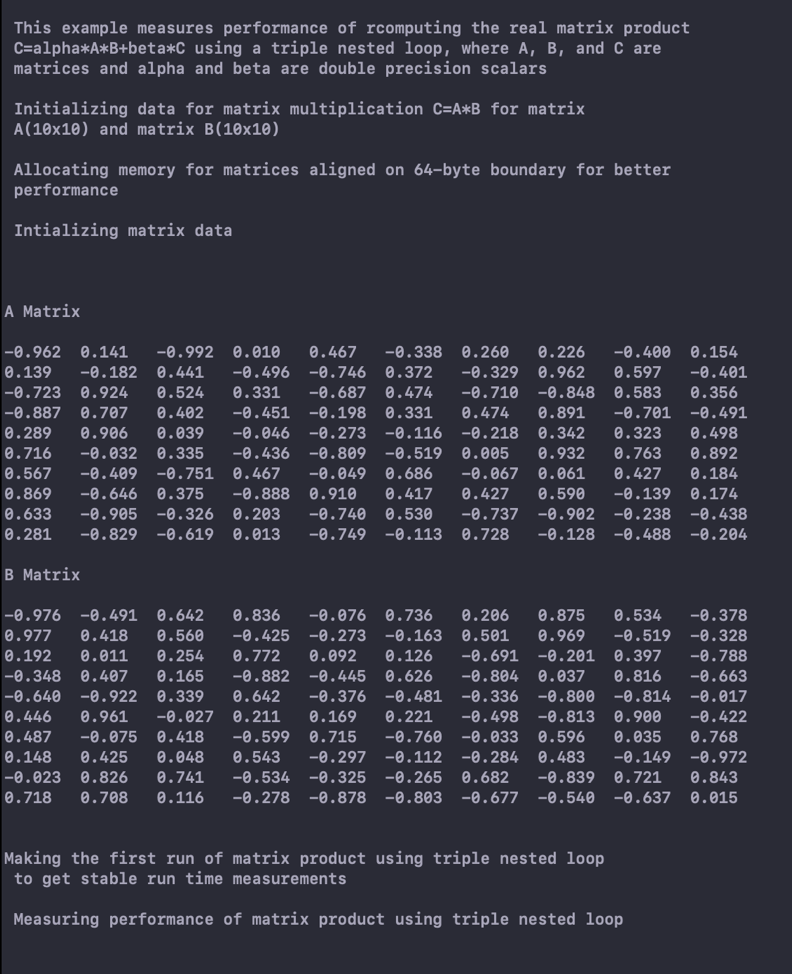
}

}

이렇게 되는데 Algo3은 j값이 변함에 따라 B,C행렬의 열이 변하기 때문에 다음 index를 찾아 가면 되지만, 바뀐 알고리즘은 i값에 따라 A,C행렬의 행이 변하는 구조라서 현재 위치보다 10칸 떨어진 index를 찾아가야하기 때문에 효율성이 떨어집니다.

4.

10X10 행렬 A,B,C를 만들어 -1~1사이의 난수값을 넣은 후, 각 3개의 알고리즘 결과를 출력하여, 같은 결과를 출력하는지 확인해보도록 하겠습니다.



위의 실행화면을 보면 무작위로 생성된 값을 넣은 행렬 A,B의 곱셈 연산을 위에 명시된 3개의 알고리즘을 통해 계산결과를 출력하여 3개의 알고리즘이 같은 결과를 생성함을 확인하였습니다.

5.

dgemm\_with\_timing.c에 있는 루틴을 Algo4라고 하겠습니다. Algo4는 cblas\_dgemm이라는 함수인데 mkl헤더에 포함되어 있는 mkl함수로 mkl.h를 include하면 사용할 수 있습니다. 4개의 행렬 곱셈 알고리즘중에 가장 뛰어난 속도를 보여줬습니다.

6.

먼저 과제 출력물에는 행렬의 크기 1000~ 10000까지 Algo1,2,3,4의 속도를 비교하라고 명시되있지만

행렬의 크기 5000이상은 실행시간이 너무 많이 소요되는 관계로 생략하겠습니다.

또한 Algo4의 속도가 독보적으로 빠른 관계로 아래 그래프에서의 비교는 무의미하다고 판단하여

그래프에서는 제외 하였습니다.

결론부터 말씀드리자면 각 알고리즘의 속도는 Algo4>Algo3>Algo2>Algo1이 되는데 속도 측정방법은

Matrix\_multiplication.c에서 성능측정 원리를 이용하여 밀리 세컨드를 구한 후 공식을

을 이용하여 구하였습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 2000 | 3000 | 4000 |
| algo1 | 469360255475992 | 2801873096109470 | 8853418306846080 | 18380818207683600 |
| algo2 | 84944290719618 | 7951017920439180 | 23858955641544500 | 56195339650666300 |
| algo3 | 864013354925831 | 8096863331460060 | 23858955641544500 | 58649975981549500 |
| algo4 | 19668749337537200 | 372922053120216000 | 1153945687124520000 | 3332972571076630000 |

7.

앞서 각기 다른 4개의 알고리즘을 비교해 서로 성능차이가 발생하는것을 확인하였습니다.

이는 행렬은 column(열)row(행)로 구성되어 있는데 우리가 사용하는 메모리 구조는 1차원입니다. memory cache는 근처에 있는것을 자주 사용할 것이다’라는 가정에 따라 열을 움직이며 연산을 하는것 보다, 행을 움직이며 연산을 실행하게 되면 메모리에 접근할때 매우 비효율적이게 되므로 속도가 떨어지는 것입니다.

따라서 각각의 알고리즘이 “어떤 루프를 이용하여 연산을 해나가냐”에 따라 성능 차이가 발생하게 됩니다.

Mkl함수를 이용한 algo4를 제외한 나머지 3개의 알고리즘을 비교해 보면 먼저 비슷한 성능을 보여준 algo3와 algo2를 확인해 보겠습니다.

Algo3 와 algo2는 각각 i-k-j루프와 i-j-k루프로 동작하는데 이 2개의 알고리즘이 Algo1보다 빠른 속도를 보여준 이유는 algo3은 변수 j로 인해 열이 움직이며 동작하고,

Algo2는 i-j-k 루프 즉 행과 열이 하나씩 움직이며 동작하기 때문입니다. 위에 설명 드린대로 행을 움직이며 계산하게 되면 메모리접근이 비효율적이므로 더 많은 시간을 소요하게 되는데, 그러므로 모든 계산이 행을 움직이며 동작하는 Algo1은 셋중에 가장 비효율적인 메모리 접근으로 인하여 가장 느린 속도를 가지게 되는 것입니다.

Algo4는 mkl헤더를 이용하여 별도의 과정을 거치어 압도적인 속도를 내는데 루프를 이용하여 계산하는 Algo1,2,3과는 비교 대상이 안된다고 판단하였습니다.