

Homework #4

cpe 512

Kyle Ray

November 27, 2017

Contents

[Part 1 2](#_Toc499509685)

[Action 1.1 2](#_Toc499509686)

[Action 1.2 2](#_Toc499509687)

[Action 1.3 3](#_Toc499509688)

[Action 1.4 4](#_Toc499509689)

[Action 1.5 5](#_Toc499509690)

[Action 1.6 6](#_Toc499509691)

[Action 1.7 7](#_Toc499509692)

[Part 2 9](#_Toc499509693)

[Action 2.1 9](#_Toc499509694)

[Action 2.2 10](#_Toc499509695)

[Appendix 11](#_Toc499509696)

[Laplace Source Code Part1 Final 11](#_Toc499509697)

[Laplace Source Code Part 2 Final 13](#_Toc499509698)

[Matrix Multiplication Part 2 OpenAcc Source Code Final 15](#_Toc499509699)

# Part 1

## Action 1.1

**Table 1: Laplace 2D OpenMP Run Times with NP = 1,2,4,8**



## Action 1.2

**Table 2: Laplace 2D OpenAcc Run Times Using Kernels**



The average run time has increased dramatically using only the “#pragma acc kernels” on the main loops.

****

The commented lines “Action 1.2 Change” are the portions of code that were changed for this action item.

## Action 1.3

**Table 3: Laplace 2D OpenAcc Run Times Using Loop Improvement**



This is a significant improvement in execution time compared to the OpenMP version as well as just using the kernels directive in OpenAcc.

****

The commented line “Action 1.3 Change” are the portions of code that were changed for this action item.

## Action 1.4

**Table 4: Laplace 2D OpenAcc Run Times Loop Tuning**



This is a very significant improvement over the OpenMP implementation even when using 8 CPU cores.

****

## Action 1.5

**Table 5: Laplace 2D OpenAcc Run Times Parallel**



It seems that this is the slowest version yet, even slower than the OpenAcc implementation with just the kernels directive.



## Action 1.6

**Table 6: Laplace 2D OpenAcc Run Times Parallel Transfer Reduced**



Like before, when we cut out the many times the data had to be copied to the GPU, we have reached a significant performance increase. This is on par with the changes that were made in Action 1.3. Overall still a significant change from the OpenMP version.



## Action 1.7

**Table 7: Laplace 2D OpenAcc Run Times Parallel Transfer Reduced Tuned**



Out of all the implementations thus far this is the lowest average execution time. With an average of 8.6 seconds over five runs compared to the 34 seconds over five runs for the OpenMP implementation utilizing 8 threads. The rest of the test cases with OpenMP do not even come close to the speed this solution offers. The GPU version seems to fair quite a bit better for this type of problem.



# Part 2

## Action 2.1

**Table 8: Matrix Multiplication Run Time for Square Matrix Size 8000 Serial**



**Table 9: Matrix Multiplication Run Times for Square Matrix Size 8000 with OpenAcc**



When testing the implementation with OpenAcc with smaller array sizes, just to check the output was correct, I was getting much larger execution times than the serial. I’m assuming this is because my implementation is copying the arrays over to the GPU and this incurs some overhead. For much larger array sizes, the tables turn, and the GPU is much faster than the serial version, over 6000 seconds vs. 17 seconds! The speedup that is gained from just adding a few lines of code to the serial implementation is astounding!

****

## Action 2.2

**Table 10: Matrix Multiplication Run Times for Various Square Matrix Sizes**



Looking at this we can see that the serial implementation execution time is growing exponentially with the size of the square matrices. While the MPI implementations are faster they are still growing exponentially just as the serially implementation. The GPU implementation, using OpenAcc however, seems to be increasing in a linear fashion from the results I’ve gathered out of these tests. This is an amazing decrease in execution time, and with a lot less work on the programmer! It is worthwhile to note that there seems to be a slight overhead encountered by copying the data over to the GPU. From other tests that I’ve ran while doing this assignment I was seeing that even with very small matrices that my OpenAcc implementation was running around 4-5 seconds. This can be seen in the 512 and 1024 sized matrix cases where even the serial implementation either is faster or is close to that of the GPU execution time. So, it would be safe to say that for very small matrices the OpenMPI versions would be better and for much larger matrices utilizing the GPU would be ideal.

# Appendix

## Laplace Source Code Part1 Final

/\*

\* Copyright 2017 NVIDIA Corporation

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

to load pgi compiler on dmc.asc.edu first type

module load pgi

to compile type

pgc++ -fast laplace2d\_acc.cpp -o laplace2d\_acc -acc -ta=nvidia -Minfo=accel

this creates the

./laplace2d\_acc

executable.

to run -- use the run\_gpu queue and place the following lines in a

bash script file

#!/bin/bash

./laplace2d\_acc

\*/

/\*

In Final Exam Problem 1 -- add the necessary OpenACC pragmas that are

needed to accomplish the goals of each part.

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <sstream>

#include <math.h>

#include <string.h>

#include "timer.h"

#define NN 4096

#define NM 4096

double A[NN][NM];

double Anew[NN][NM];

int main(int argc, char\*\* argv)

{

const int n = NN;

const int m = NM;

const int iter\_max = 1000;

const double tol = 1.0e-6;

double error = 1.0f;

memset(A, 0, n \* m \* sizeof(double));

memset(Anew, 0, n \* m \* sizeof(double));

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

{

A[j][0] = 1.0;

Anew[j][0] = 1.0;

}

cout << "Jacobi relaxation Calculation: " << n << " x "

<< m << " mesh" << endl;

StartTimer();

int iter = 0;

#pragma acc data copy(A) create(Anew) // Action 1.3 Change

{

while ( error > tol && iter < iter\_max ) {

error = 0.0;

#pragma acc kernels // Action 1.2 Change

{

#pragma acc loop device\_type(nvidia) tile(32,4) // Action 1.4 Change

{

for( int j = 1; j < n-1; j++) {

//#pragma acc loop gang(16), vector(32)

for( int i = 1; i < m-1; i++ ) {

Anew[j][i] = 0.25 \* ( A[j][i+1] + A[j][i-1]

+ A[j-1][i] + A[j+1][i]);

error = fmax( error, fabs(Anew[j][i] - A[j][i]));

}

}

}

#pragma acc loop device\_type(nvidia) tile(32,4) // Action 1.4 Change

{

for( int j = 1; j < n-1; j++) {

//#pragma acc loop gang(16), vector(32)

for( int i = 1; i < m-1; i++ ) {

A[j][i] = Anew[j][i];

}

}

}

}

if (iter % 100 == 0) cout << iter << " " << setprecision(6)

<< fixed << error << endl;

iter++;

}

}

double runtime = GetTimer();

cout << " total: " << runtime/1000 << " s" << endl;

}

## Laplace Source Code Part 2 Final

/\*

\* Copyright 2017 NVIDIA Corporation

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

to load pgi compiler on dmc.asc.edu first type

module load pgi

to compile type

pgc++ -fast laplace2d\_acc.cpp -o laplace2d\_acc -acc -ta=nvidia -Minfo=accel

this creates the

./laplace2d\_acc

executable.

to run -- use the run\_gpu queue and place the following lines in a

bash script file

#!/bin/bash

./laplace2d\_acc

\*/

/\*

In Final Exam Problem 1 -- add the necessary OpenACC pragmas that are

needed to accomplish the goals of each part.

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <sstream>

#include <math.h>

#include <string.h>

#include "timer.h"

#define NN 4096

#define NM 4096

double A[NN][NM];

double Anew[NN][NM];

int main(int argc, char\*\* argv)

{

const int n = NN;

const int m = NM;

const int iter\_max = 1000;

const double tol = 1.0e-6;

double error = 1.0f;

memset(A, 0, n \* m \* sizeof(double));

memset(Anew, 0, n \* m \* sizeof(double));

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

{

A[j][0] = 1.0;

Anew[j][0] = 1.0;

}

cout << "Jacobi relaxation Calculation: " << n << " x "

<< m << " mesh" << endl;

StartTimer();

int iter = 0;

#pragma acc data copy(A), create(Anew) // Action 1.6 Change

{

while ( error > tol && iter < iter\_max ) {

error = 0.0;

#pragma acc parallel loop reduction(max:error) device\_type(nvidia) tile (32, 4) // Action 1.5 Change // Action 1.7 Change

{

for( int j = 1; j < n-1; j++) {

for( int i = 1; i < m-1; i++ ) {

Anew[j][i] = 0.25 \* ( A[j][i+1] + A[j][i-1]

+ A[j-1][i] + A[j+1][i]);

error = fmax( error, fabs(Anew[j][i] - A[j][i]));

}

}

}

#pragma acc parallel loop device\_type(nvidia) tile(32, 4) // Action 1.5 Change // Action 1.7 Change

{

for( int j = 1; j < n-1; j++) {

for( int i = 1; i < m-1; i++ ) {

A[j][i] = Anew[j][i];

}

}

}

if (iter % 100 == 0) cout << iter << " " << setprecision(6)

<< fixed << error << endl;

iter++;

}

}

double runtime = GetTimer();

cout << " total: " << runtime/1000 << " s" << endl;

}

## Matrix Multiplication Part 2 OpenAcc Source Code Final

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

/\* Matrix Matrix Multiplication Program Example -- serial version \*/

/\* September 2017 -- B. Earl Wells -- University of Alabama \*/

/\* in Huntsville \*/

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

// mm\_mult\_serial.cpp

// compilation:

// gnu compiler

// pgc++ mm\_mult\_serial.cpp -o mm\_mult\_serial -O3 -lm

/\*

This program is designed to perform matrix matrix multiplication

A x B = C, where A is an lxm matrix, B is a m x n matrix and

C is a l x n matrix. The program is designed to be a template

serial program that can be expanded into a parallel multiprocess

and/or a multi-threaded program.

The program randomly assigns the elements of the A and B matrix

with values between 0 and a MAX\_VALUE. It then multiples the

two matrices with the result being placed in the C matrix.

The program prints out the A, B, and C matrices.

The program is executed using one or three command line parameters.

These parameters represent the dimension of the matrices. If only

one parameter is used then then it is assumed that square matrices are

to be created and multiplied together that have the specified

dimension. In cases where three command line parameters are entered

then the first parameter is the l dimension, the second the m, and

the third is the n dimension.

To execute:

mm\_mult\_serial [l\_parameter] <m\_parameter n\_parameter>

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <sstream>

#include <stdlib.h>

#include <string.h>

#include <sys/time.h>

#include "timer.h"

#define MX\_SZ 320

#define SEED 2397 /\* random number seed \*/

#define MAX\_VALUE 100.0 /\* maximum size of array elements A, and B \*/

/\* copied from mpbench \*/

#define TIMER\_CLEAR (tv1.tv\_sec = tv1.tv\_usec = tv2.tv\_sec = tv2.tv\_usec = 0)

#define TIMER\_START gettimeofday(&tv1, (struct timezone\*)0)

#define TIMER\_ELAPSED ((long long) (tv3.tv\_usec)+((long long) (tv3.tv\_sec)\*1000000))

#define TIMER\_STOP {gettimeofday(&tv2, (struct timezone\*)0);timersub(&tv2,&tv1,&tv3);}

struct timeval tv1,tv2,tv3;

/\*

This declaration facilitates the creation of a two dimensional

dynamically allocated arrays (i.e. the lxm A array, the mxn B

array, and the lxn C array). It allows pointer arithmetic to

be applied to a single data stream that can be dynamically allocated.

To address the element at row x, and column y you would use the

following notation: A(x,y),B(x,y), or C(x,y), respectively.

Note that this differs from the normal C notation if A were a

two dimensional array of A[x][y] but is still very descriptive

of the data structure.

\*/

float \*a,\*b,\*c;

#define A(i,j) \*(a+i\*dim\_m+j)

#define B(i,j) \*(b+i\*dim\_n+j)

#define C(i,j) \*(c+i\*dim\_n+j)

/\*

Routine to retrieve the data size of the numbers array from the

command line or by prompting the user for the information

\*/

void get\_index\_size(int argc,char \*argv[],int \*dim\_l,int \*dim\_m,int \*dim\_n) {

if(argc!=2 && argc!=4) {

cout<<"usage: mm\_mult\_serial [l\_dimension] <m\_dimension n\_dimmension>"

<< endl;

exit(1);

}

else {

if (argc == 2) {

\*dim\_l = \*dim\_n = \*dim\_m = atoi(argv[1]);

}

else {

\*dim\_l = atoi(argv[1]);

\*dim\_m = atoi(argv[2]);

\*dim\_n = atoi(argv[3]);

}

}

if (\*dim\_l<=0 || \*dim\_n<=0 || \*dim\_m<=0) {

cout<<"Error: number of rows and/or columns must be greater than 0"

<< endl;

exit(1);

}

}

/\*

Routine that fills the number matrix with Random Data with values

between 0 and MAX\_VALUE

This simulates in some way what might happen if there was a

single sequential data acquisition source such as a single file

\*/

void fill\_matrix(float \*array,int dim\_m,int dim\_n)

{

int i,j;

for(i=0;i<dim\_m;i++) {

for (j=0;j<dim\_n;j++) {

array[i\*dim\_n+j]=drand48()\*MAX\_VALUE;

}

}

}

/\*

Routine that outputs the matrices to the screen

\*/

void print\_matrix(float \*array,int dim\_m,int dim\_n)

{

int i,j;

for(i=0;i<dim\_m;i++) {

for (j=0;j<dim\_n;j++) {

cout << array[i\*dim\_n+j] << " ";

}

cout << endl;

}

}

/\*

MAIN ROUTINE: summation of a number list

\*/

int main( int argc, char \*argv[])

{

float dot\_prod;

int dim\_l,dim\_n,dim\_m;

int i,j,k;

/\*

get matrix sizes

\*/

get\_index\_size(argc,argv,&dim\_l,&dim\_m,&dim\_n);

// dynamically allocate from heap the numbers in the memory space

// for the a,b, and c matrices

a = new (nothrow) float[dim\_l\*dim\_m];

b = new (nothrow) float[dim\_m\*dim\_n];

c = new (nothrow) float[dim\_l\*dim\_n];

if(a==0 || b==0 || c==0) {

cout <<"ERROR: Insufficient Memory" << endl;

exit(1);

}

/\*

initialize numbers matrix with random data

\*/

srand48(SEED);

fill\_matrix(a,dim\_l,dim\_m);

fill\_matrix(b,dim\_m,dim\_n);

/\*

output numbers matrix

\*/

//cout << "A matrix =" << endl;

//print\_matrix(a,dim\_l,dim\_m);

//cout << endl;

//cout << "B matrix =" << endl;

//print\_matrix(b,dim\_m,dim\_n);

//cout << endl;

/\*

Start recording the execution time

\*/

//TIMER\_CLEAR;

//TIMER\_START;

//

StartTimer();

// multiply local part of matrix

#pragma acc data copy(a[0:dim\_l\*dim\_m]) copy( b[0:dim\_m\*dim\_n]) copy(c[0:dim\_l\*dim\_n])

{

#pragma acc kernels

{

#pragma acc loop device\_type(nvidia) tile(32,4)

{

for (i=0;i<dim\_l;i++) {

for (j=0;j<dim\_n;j++) {

dot\_prod = 0.0;

for (k=0;k<dim\_m;k++) {

dot\_prod += A(i,k)\*B(k,j);

}

C(i,j) = dot\_prod;

}

}

}

}

}

/\*

stop recording the execution time

\*/

//TIMER\_STOP;

//cout << "C matrix =" << endl;

//print\_matrix(c,dim\_l,dim\_n);

//cout << endl;

double runTime = GetTimer();

cout << "time=" << setprecision(8) << runTime/1000.0

<< " seconds" << endl;

}