**Applied Parallel Computing HW 1**

**Manisha Agrawal**

**1). A pThreads C program to multiply two square matrices. I am multiplying two 8\*8 matrices using 4 threads. Also I have included a non-threaded routine to test against the threaded routine output. A structure is used to pass arguments to mulArrayPthread function. The members of which are 2 input vectors, output vector and startId.**

/\* ============================================================================

Name : mul\_pthreads.c

Author : Manisha Agrawal

Create a pThreads or C++11 program to multiply two square matrices of arbitrary size.

Fill matrices with random double data. Include a non-threaded routine to multiply

the matrices in order to check. Write a test routine to make sure that both methods

produce the same answer to within a reasonable tolerance. ============================================================================

\*/

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NTHREADS 4

#define ARRAYSIZE 8

typedef struct

{

int startIdx;

double \*\*arrC, \*\*arrA, \*\*arrB;

} mulType;

void \*mulArrayPthread(void \*x)

{

mulType \*p = (mulType \*) x;

int blockSize = ARRAYSIZE/NTHREADS;

int end = (\*p).startIdx + blockSize;

printf("Start=%d\tEnd=%d\n",(\*p).startIdx, end);

int i, j ,k;

for (i=(\*p).startIdx;i<end;i++)

{

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

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

(\*p).arrC[i][j] += (\*p).arrA[i][k]\*(\*p).arrB[k][j];

}

return NULL;

}

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

{

int i, j, k, tids[NTHREADS];

pthread\_t threads[NTHREADS];

pthread\_attr\_t attr;

double \*\*A\_array=(double\*\*)malloc(ARRAYSIZE\*sizeof(double\*));

double \*\*B\_array=(double\*\*)malloc(ARRAYSIZE\*sizeof(double\*));

double \*\*Non\_array=(double\*\*)malloc(ARRAYSIZE\*sizeof(double\*));

double \*\*Thread\_array=(double\*\*)malloc(ARRAYSIZE\*sizeof(double\*));

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

{

A\_array[i]=(double\*)malloc(ARRAYSIZE\*sizeof(double));

B\_array[i]=(double\*)malloc(ARRAYSIZE\*sizeof(double));

Non\_array[i]=(double\*)malloc(ARRAYSIZE\*sizeof(double));

Thread\_array[i]=(double\*)malloc(ARRAYSIZE\*sizeof(double));

}

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

{

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

{

A\_array[i][j] = rand()/(float)RAND\_MAX;

B\_array[i][j] = rand()/(float)RAND\_MAX;

}

}

//Non-threaded routine

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

{

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

{

Non\_array[i][j]=0.0;

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

Non\_array[i][j] = Non\_array[i][j] + (A\_array[i][k] \* B\_array[k][j]) ;

}

}

int index=ARRAYSIZE/NTHREADS ;

mulType \*\*datas = malloc(NTHREADS\*sizeof(mulType));

for (i = 0; i != NTHREADS ; i++)

{

datas[i] = malloc(sizeof(mulType));

datas[i]->startIdx = index\*i;

datas[i]->arrA = A\_array;

datas[i]->arrB = B\_array;

datas[i]->arrC = Thread\_array;

}

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

{

tids[i] = i;

pthread\_create(&threads[i], NULL, mulArrayPthread, (void \*)(datas[i]));

}

/\* Wait for all threads to complete then print global sum \*/

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

{

pthread\_join(threads[i], NULL);

}

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

{

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

{

printf("[%d][%d]:Serial=%f\tThread=%f\n",i,j, Non\_array[i][j], Thread\_array[i][j]);

}

}

/\* Clean up and exit \*/

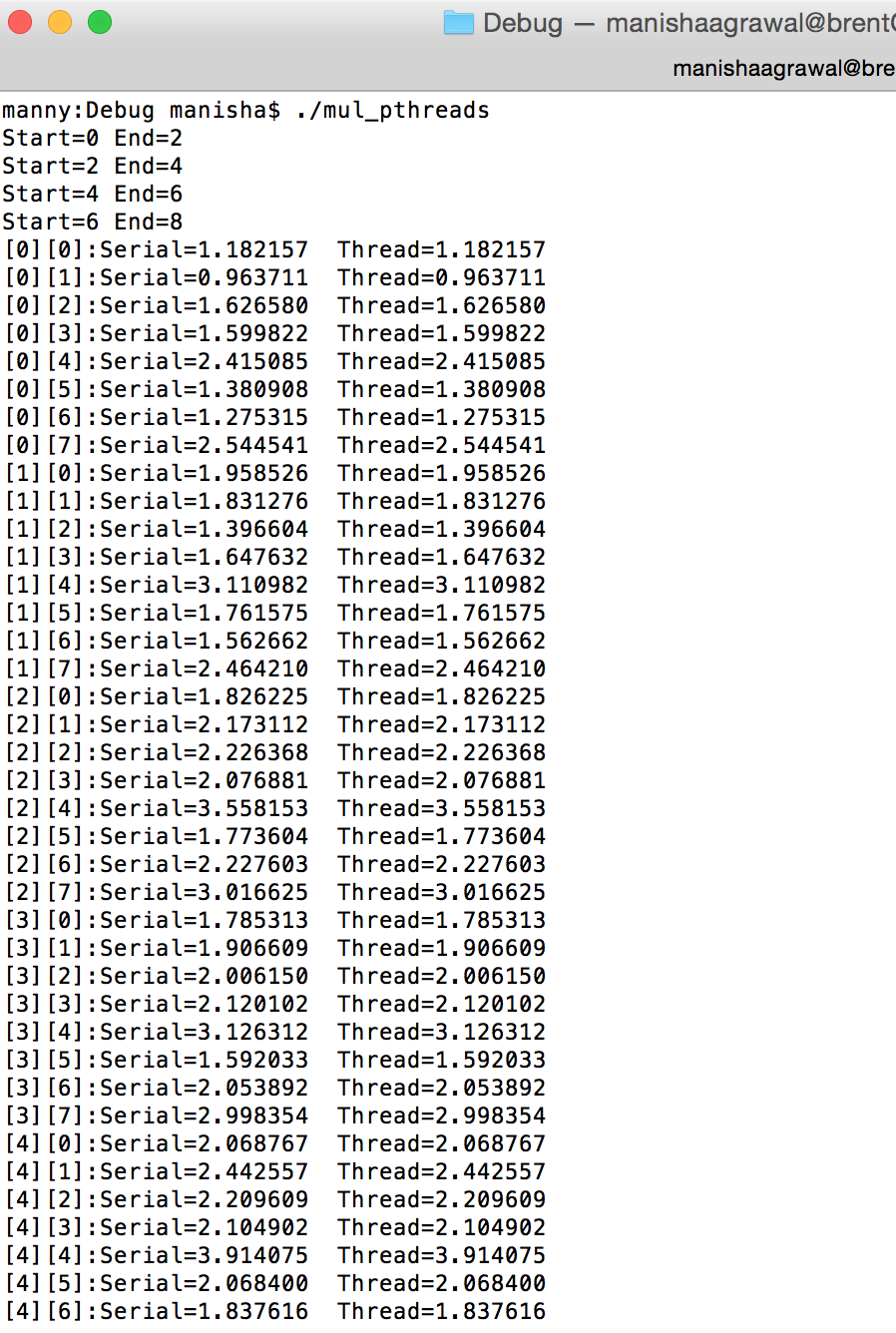
pthread\_attr\_destroy(&attr);

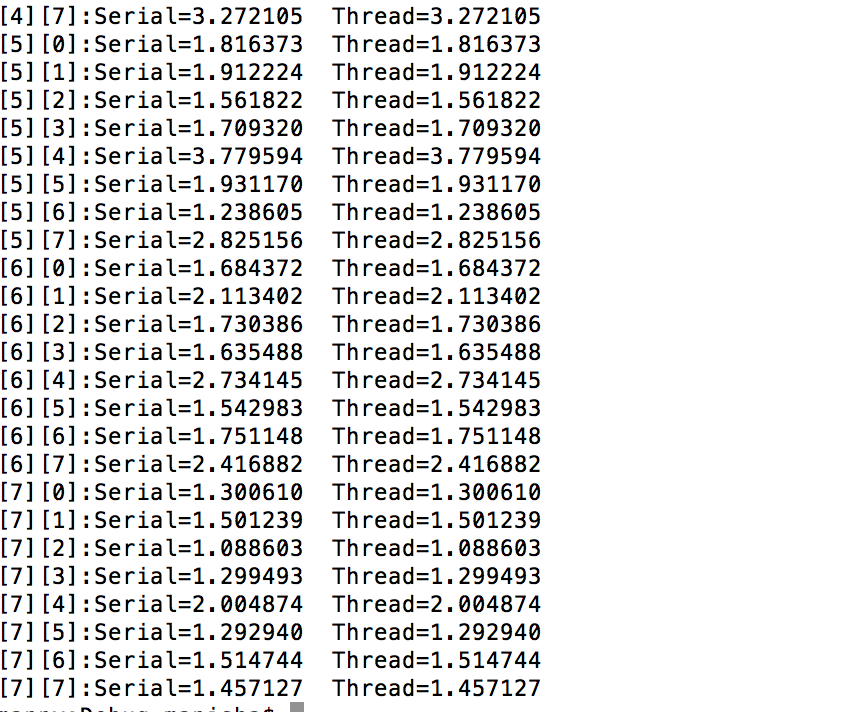
pthread\_exit (NULL);

return 0;

}

**Output**



****

**2). A Cuda program to multiply two square matrices. I am multiplying two 2-D matrices, also I have included a serial routine to test against the parallel routine output. Random floating point values were stored in the 2 input matrix. After the computation of parallel resultant array, each element was compared with their corresponding serial array elements. A root mean square of the error was calculated. From the output we can deduce that as the number of elements increases the error also increases.**

/\*CUDA 2-D Matrix Multiplication\*/

#include <stdio.h>

#include <math.h>

#define TILE\_WIDTH 2

#define WIDTH 100

\_\_global\_\_ void MatrixMul( float \*A\_d , float \*B\_d , float \*C\_d)

{

// calculate thread id

unsigned int col = TILE\_WIDTH\*blockIdx.x + threadIdx.x ;

unsigned int row = TILE\_WIDTH\*blockIdx.y + threadIdx.y ;

C\_d[row\*WIDTH+col] = 0;

for (int k = 0 ; k<WIDTH ; k++ )

{

C\_d[row\*WIDTH + col]+= A\_d[row \* WIDTH + k ] \* B\_d[ k \* WIDTH + col] ;

}

}

// main routine

int main ()

{

float array1\_h[WIDTH][WIDTH] ,array2\_h[WIDTH][WIDTH];

float non\_array\_h[WIDTH][WIDTH] ,M\_result\_array\_h[WIDTH][WIDTH] ;

float \*array1\_d , \*array2\_d ,\*M\_result\_array\_d ; // device array

int i , j ;

//input in host array

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

{

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

{

array1\_h[i][j] =(float)rand()/(float)RAND\_MAX;

array2\_h[i][j] = (float)rand()/(float)RAND\_MAX;

}

}

//Non-threaded routine

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

{

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

{

non\_array\_h[i][j]=0;

for(int k=0; k<WIDTH; ++k)

{

non\_array\_h[i][j] += array1\_h[i][k] \* array2\_h[k][j] ;

}

}

}

//create device array cudaMalloc ( (void \*\*)&array\_name, sizeofmatrixinbytes) ;

cudaMalloc((void \*\*) &array1\_d , WIDTH\*WIDTH\*sizeof (float) ) ;

cudaMalloc((void \*\*) &array2\_d , WIDTH\*WIDTH\*sizeof (float) ) ;

//copy host array to device array; cudaMemcpy ( dest , source , WIDTH , direction )

cudaMemcpy ( array1\_d , array1\_h , WIDTH\*WIDTH\*sizeof (float) , cudaMemcpyHostToDevice ) ;

cudaMemcpy ( array2\_d , array2\_h , WIDTH\*WIDTH\*sizeof (float) , cudaMemcpyHostToDevice ) ;

//allocating memory for resultant device array

cudaMalloc((void \*\*) &M\_result\_array\_d , WIDTH\*WIDTH\*sizeof (float) ) ;

//calling kernel

dim3 dimGrid ( WIDTH/TILE\_WIDTH , WIDTH/TILE\_WIDTH ,1 ) ;

dim3 dimBlock( TILE\_WIDTH, TILE\_WIDTH, 1 ) ;

MatrixMul <<<dimGrid,dimBlock>>> ( array1\_d , array2\_d ,M\_result\_array\_d) ;

//copy back result\_array\_d to result\_array\_h

cudaMemcpy(M\_result\_array\_h , M\_result\_array\_d , WIDTH\*WIDTH\*sizeof(float) ,

cudaMemcpyDeviceToHost) ;

float error = 0;

//print the result array

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Vector Multiplication of %d\*%d matrix\n", WIDTH, WIDTH);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

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

{

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

{

// printf ("%f %f\t",non\_array\_h[i][j], M\_result\_array\_h[i][j] ) ;

**Output**

//Root mean square of the differences

error += pow(M\_result\_array\_h[i][j]-non\_array\_h[i][j], 2);

}

//printf ("\n") ;

}

error = error/(WIDTH\*WIDTH);

error = sqrt(error);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Error:%f\n", error);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

cudaFree(array1\_d);

cudaFree(array2\_d);

cudaFree(M\_result\_array\_d);

return 0;

}



**3). A Cuda program to implement a dot product using shared memory and \_\_syncthreads. Two 1-D arrays are taken as input where random floating point values were stored. . A dot product of the input matrices was also calculated using a serial method to test against the parallel output.**

#include <stdio.h>

#include <cuda\_runtime.h>

#include <stdlib.h>

const int numElements = 1024;

const int threadsPerBlock = 256;

const int blocksPerGrid = (numElements+threadsPerBlock - 1) / threadsPerBlock;

\_\_global\_\_ void cuda\_dot\_kernel(float \*a, float \*b, float \*c)

{

int id = (blockIdx.x \* blockDim.x) + threadIdx.x;

int nextid = gridDim.x \* blockDim.x;

\_\_shared\_\_ float shared\_cache [threadsPerBlock];

int cacheIndex = threadIdx.x;

float temp = 0;

while(id < numElements)

{

temp+= a[id]\*b[id];

id += nextid;

}

shared\_cache[cacheIndex] = temp;

\_\_syncthreads();

//for reduction thread block must be power of 2

for(blockDim.x/2; i!=0; i /= 2)

{

if(cacheIndex < i)

shared\_cache[cacheIndex] += shared\_cache[cacheIndex+i];

\_\_syncthreads();

}

if ( cacheIndex == 0)

{

c[blockIdx.x] = shared\_cache[0];

}

}

// Host main routine

int main(void)

{

int i;

cudaError\_t err = cudaSuccess; // Error code to check return values for CUDA calls

size\_t size = numElements \* sizeof(float);

// Allocate the host input vector A,B,C

float \*h\_A = (float \*)malloc(size);

float \*h\_B = (float \*)malloc(size);

float \*h\_C = (float \*)malloc(blocksPerGrid\*sizeof(float));

// Verify that allocations succeeded

if (h\_A == NULL || h\_B == NULL || h\_C == NULL)

{

fprintf(stderr, "Failed to allocate host vectors!\n");

exit(EXIT\_FAILURE);

}

// Initialize the host input vectors

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

{

h\_A[i] = rand()/(float)RAND\_MAX;

h\_B[i] = rand()/(float)RAND\_MAX;

serial\_product += h\_A[i]\*h\_B[i];

}

//Allocate the device input vector A

float \*d\_A = NULL;

cudaMalloc((void\*\*)&d\_A, size);

// Allocate the device input vector B

float \*d\_B = NULL;

cudaMalloc((void\*\*)&d\_B,size);

// Allocate the device output vector C

float \*d\_C = NULL ;

cudaMalloc( (void\*\*)&d\_C, blocksPerGrid\*sizeof(float) );

// Copy the host input vectors A and B to the device input vectors

err = cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)

{

fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n", cudaGetErrorString(err));

exit(EXIT\_FAILURE);

}

err = cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)

{

fprintf(stderr, "Failed to copy vector B from host to device (error code %s)!\n", cudaGetErrorString(err));

exit(EXIT\_FAILURE);

}

// Launch the Dot Vector CUDA Kernel

cuda\_dot\_kernel<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C);

// Copy device result to the host result

err = cudaMemcpy(h\_C, d\_C, blocksPerGrid\*sizeof(float), cudaMemcpyDeviceToHost);

if (err != cudaSuccess)

{

fprintf(stderr, "Failed to copy result from device to host (error code %s)!\n", cudaGetErrorString(err));

exit(EXIT\_FAILURE);

}

// Free device global memory

cudaFree(d\_C);

cudaFree(d\_B);

cudaFree(d\_C);

float result = 0;

for(int k=0; k<blocksPerGrid; k++)

{

result += h\_C[k];

}

printf("Cuda Dot Product=%f\n", result);

printf("Serial product =%f\n", serial\_product);

// Free host memory

free(h\_A);

free(h\_B);

free(h\_C);

return 0;

}

**Output**

****