Assignment -1

#include <bits/stdc++.h>

#include <omp.h>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Node structure for the binary tree

struct Node {

int val;

Node\* left;

Node\* right;

Node(int v) {

val = v;

left = right = NULL;

}

};

// Sequential Depth First Search (DFS) traversal of the binary tree

void sequentialDFS(Node\* root) {

if (!root) return;

stack<Node\*> s;

s.push(root);

while (!s.empty()) {

Node\* curr = s.top();

s.pop();

cout << curr->val << " ";

if (curr->right)

s.push(curr->right);

if (curr->left)

s.push(curr->left);

}

cout << endl;

}

// Sequential Breadth First Search (BFS) traversal of the binary tree

void sequentialBFS(Node\* root) {

if (!root) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* curr = q.front();

q.pop();

cout << curr->val << " ";

if (curr->left)

q.push(curr->left);

if (curr->right)

q.push(curr->right);

}

cout << endl;

}

// Parallel Depth First Search (DFS) traversal of the binary tree

void parallelDFS(Node\* root) {

if (!root) return;

stack<Node\*> s;

s.push(root);

#pragma omp parallel

{

while (!s.empty()) {

Node\* curr;

#pragma omp critical

{

curr = s.top();

s.pop();

}

cout << curr->val << " ";

#pragma omp single

{

if (curr->right)

s.push(curr->right);

if (curr->left)

s.push(curr->left);

}

}

}

cout << endl;

}

// Parallel Breadth First Search (BFS) traversal of the binary tree

void parallelBFS(Node\* root) {

if (!root) return;

queue<Node\*> q;

q.push(root);

#pragma omp parallel

{

while (!q.empty()) {

Node\* curr;

#pragma omp critical

{

curr = q.front();

q.pop();

}

cout << curr->val << " ";

#pragma omp single

{

if (curr->left)

q.push(curr->left);

if (curr->right)

q.push(curr->right);

}

}

}

cout << endl;

}

// Function to measure the time taken by a traversal function

double measureTime(void (\*function)(Node\*), Node\* root) {

auto start = chrono::high\_resolution\_clock::now();

function(root);

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

return duration.count();

}

signed main() {

// Prompting the user to enter the value for the root node

int rootVal;

cout << "Enter the value for the root node: ";

cin >> rootVal;

Node\* root = new Node(rootVal);

// Queue to store nodes whose children need to be entered

queue<Node\*> pendingNodes;

pendingNodes.push(root);

// Prompting the user to enter values for each node and constructing the binary tree

while (!pendingNodes.empty()) {

Node\* curr = pendingNodes.front();

pendingNodes.pop();

int leftVal, rightVal;

cout << "Enter the value for the left child of " << curr->val << " (-1 if none): ";

cin >> leftVal;

if (leftVal != -1) {

curr->left = new Node(leftVal);

pendingNodes.push(curr->left);

}

cout << "Enter the value for the right child of " << curr->val << " (-1 if none): ";

cin >> rightVal;

if (rightVal != -1) {

curr->right = new Node(rightVal);

pendingNodes.push(curr->right);

}

}

// Menu-driven program to allow the user to choose traversal methods

bool flag = true;

while(flag) {

cout << "\n<-------MENU------>" << endl;

cout << "1. For Sequential DFS and Parallel DFS" << endl;

cout << "2. For Sequential BFS and Parallel BFS" << endl;

cout << "3. For Exit" << endl;

int ch;

cout << "Enter Your Choice: ";

cin >> ch;

switch(ch) {

case 1: {

cout << "\nSequential DFS : ";

double timeSequentialDFS = measureTime(sequentialDFS, root);

cout << "Time taken for Sequential DFS: " << timeSequentialDFS << " seconds" << endl;

cout << "\nParallel DFS : ";

double timeParallelDFS = measureTime(parallelDFS, root);

cout << "Time taken for Parallel DFS: " << timeParallelDFS << " seconds" << endl;

if(timeParallelDFS < timeSequentialDFS) cout << "\nParallel DFS performing better than Sequential DFS." << endl;

else cout << "\nSequential DFS is performing better than Parallel DFS" << endl;

break;

}

case 2: {

cout << "\nSequential BFS : ";

double timeSequentialBFS = measureTime(sequentialBFS, root);

cout << "Time taken for Sequential BFS: " << timeSequentialBFS << " seconds" << endl;

cout << "Parallel BFS : ";

double timeParallelBFS = measureTime(parallelBFS, root);

cout << "Time taken for Parallel BFS: " << timeParallelBFS << " seconds" << endl;

if(timeParallelBFS < timeSequentialBFS) cout << "\nParallel BFS performing better than Sequential BFS." << endl;

else cout << "\nSequential BFS is performing better than Parallel BFS" << endl;

break;

break;

}

case 3: {

cout << "\nThank You!!" << endl;

flag = false;

break;

}

default: {

cout << "\nInvalid choice, Try again" << endl;

break;

}

}

}

return 0;

}

Assignment -2

#include <bits/stdc++.h>

#include <ctime>

#include <omp.h>

using namespace std;

// Function to perform sequential bubble sort

void bubbleSortSeq(vector<int>& arr)

{

int n = arr.size();

for (int i = 0; i < n-1; i++)

{

for (int j = 0; j < n-i-1; j++)

{

if (arr[j] > arr[j+1]) swap(arr[j], arr[j+1]);

}

}

cout << "First 10 values in array after Sequential Bubble sort: ";

for(int i =0; i<10; i++)

{

cout << arr[i] << " ";

}

cout << endl;

}

// Function to perform parallel bubble sort

void bubbleSortPar(vector<int>& arr)

{

int n = arr.size();

#pragma omp parallel

{

for (int i = 0; i < n-1; i++)

{

#pragma omp for

for (int j = 0; j < n-i-1; j++)

{

if (arr[j] > arr[j+1]) swap(arr[j], arr[j+1]);

}

}

}

cout << "\nFirst 10 values in array after Parallel Bubble sort: ";

for(int i =0; i<10; i++)

{

cout << arr[i] << " ";

}

cout << endl;

}

// Function to merge two sorted subarrays

void merge(vector<int>& arr, int left, int middle, int right) {

int n1 = middle - left + 1;

int n2 = right - middle;

vector<int> L(n1), R(n2);

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int j = 0; j < n2; j++)

R[j] = arr[middle + 1 + j];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Function to perform sequential merge sort

void mergeSortSeq(vector<int>& arr, int left, int right) {

if (left < right) {

int middle = left + (right - left) / 2;

mergeSortSeq(arr, left, middle);

mergeSortSeq(arr, middle + 1, right);

merge(arr, left, middle, right);

}

}

// Function to merge two sorted subarrays in parallel

void parallelMerge(vector<int>& arr, int left, int middle, int right) {

int n1 = middle - left + 1;

int n2 = right - middle;

vector<int> L(n1), R(n2);

#pragma omp parallel for

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

#pragma omp parallel for

for (int j = 0; j < n2; j++)

R[j] = arr[middle + 1 + j];

int i = 0, j = 0, k = left;

#pragma omp parallel sections

{

#pragma omp section

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

#pragma omp section

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

#pragma omp section

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

}

// Function to perform parallel merge sort

void mergeSortPar(vector<int>& arr, int left, int right) {

if (left < right) {

int middle = left + (right - left) / 2;

#pragma omp parallel sections

{

#pragma omp section

mergeSortPar(arr, left, middle);

#pragma omp section

mergeSortPar(arr, middle + 1, right);

}

merge(arr, left, middle, right);

}

}

// Function to measure execution time of sorting algorithms

double measureTime(void (\*sortFunction)(vector<int>&), vector<int>& arr) {

double start = omp\_get\_wtime();

sortFunction(arr);

double end = omp\_get\_wtime();

return end - start;

}

// Function to measure execution time of sorting algorithms with parameters

double measuretime(void (\*sortFunction)(vector<int>&, int, int), vector<int>& arr, int left, int right) {

double start = omp\_get\_wtime();

sortFunction(arr, left, right);

double end = omp\_get\_wtime();

return end - start;

}

// Main function

signed main() {

int size;

int min\_range, max\_range;

cout << "Enter the size of the array greater than 100: ";

cin >> size;

cout << "Enter the minimum range for random numbers: ";

cin >> min\_range;

cout << "Enter the maximum range for random numbers: ";

cin >> max\_range;

vector<int> arr(size);

srand(time(0));

for (int i = 0; i < size; ++i) {

arr[i] = rand() % (max\_range - min\_range + 1) + min\_range;

}

bool flag = true;

while(flag)

{

cout << "\n<-------MENU------>"<<endl;

cout << "1. For Comparison result between Parallel and Sequential Bubble sort" << endl;

cout << "2. For Comparison result between Parallel and Sequential Merge sort" << endl;

cout << "3. For Exit"<<endl;

int ch;

cout << "Enter Your Choice: ";

cin >> ch;

switch(ch)

{

case 1:

{

vector<int> arr\_bubble = arr;

cout << "\nFirst 10 values in array before sorting: ";

for(int i =0; i<10; i++)

{

cout << arr[i] << " ";

}

cout << endl;

double sequential\_bubble\_time = measureTime(bubbleSortSeq, arr\_bubble);

cout << "Sequential Bubble Sort Time: " << sequential\_bubble\_time << " seconds" << endl;

vector<int> arr\_parallel\_bubble = arr;

double parallel\_bubble\_time = measureTime(bubbleSortPar, arr\_parallel\_bubble);

cout << "Parallel Bubble Sort Time: " << parallel\_bubble\_time << " seconds" << endl;

if(sequential\_bubble\_time < parallel\_bubble\_time) cout << "\nSequential Bubble Sort is performing better than Parallel Bubble sort." << endl;

else cout << "\nParallel Bubble Sort is performing better than Sequential Bubble sort." << endl;

break;

}

case 2:

{

vector<int> arr\_merge = arr;

cout << "\nFirst 10 values in array before sorting: ";

for(int i =0; i<10; i++)

{

cout << arr[i] << " ";

}

cout << endl;

double sequential\_merge\_time = measuretime(mergeSortSeq, arr\_merge, 0, size - 1);

cout << "First 10 values in array after Merge sort: ";

for(int i =0; i<10; i++)

{

cout << arr\_merge[i] << " ";

}

cout << endl;

cout << "Sequential Merge Sort Time: " << sequential\_merge\_time << " seconds" << endl;

vector<int> arr\_parallel\_merge = arr;

double parallel\_merge\_time = measuretime(mergeSortPar, arr\_parallel\_merge, 0, size - 1);

cout << "\nFirst 10 values in array after Merge sort: ";

for(int i =0; i<10; i++)

{

cout << arr\_parallel\_merge[i] << " ";

}

cout << endl;

cout << "Parallel Merge Sort Time: " << parallel\_merge\_time << " seconds" << endl;

if(sequential\_merge\_time < parallel\_merge\_time) cout << "\nSequential Merge Sort is performing better than Parallel Merge sort." << endl;

else cout << "\nParallel merge Sort is performing better than Sequential Merge sort." << endl;

break;

}

case 3:

{

cout << "Thank You!!" << endl;

flag = false;

break;

}

default:

{

cout << "Invalid choice, Try again" << endl;

break;

}

}

}

return 0;

}

Assignment -3

Assignment -3

#include <bits/stdc++.h>

#include <limits.h>

#include <stdlib.h>

#include <omp.h>

#include <chrono>

#include <functional>

using std :: chrono :: duration\_cast;

using std :: chrono :: high\_resolution\_clock;

using std :: chrono :: milliseconds;

using namespace std;

// sequential functions, declarations

void sequential\_average(vector<int>& array);

void sequential\_max(vector<int>& array);

void sequential\_min(vector<int>& array);

void sequential\_sum(vector<int>& array);

// parallel computing functions, declarations

void parallel\_average(vector<int>& array);

void parallel\_max(vector<int>& array);

void parallel\_min(vector<int>& array);

void parallel\_sum(vector<int>& array);

// utility functions, declaration

template <typename T> ostream& operator << (ostream& console, const vector<T>& array);

void populating\_random\_values(vector<int>& array);

void initialize\_array(vector<int>& array, int N);

// function to analyze the performance

int analysis(std :: function<void()> function);

int main(void) {

int N = 0, sequential\_execution = 0, parallel\_execution = 0;

vector<int> array;

omp\_set\_num\_threads(16);

bool flag = true;

while (flag){

// Menu driven program ...

cout << "-------------------------- Menu -------------------------------" << endl;

cout << "1. Sequential and Parallel Min Computation " << endl;

cout << "2. Sequential and Parallel Max Computation " << endl;

cout << "3. Sequential and Parallel Sum Computation " << endl;

cout << "4. Sequential and Parallel Average Computation " << endl;

cout << "5. Exit " << endl;

cout << "---------------------------------------------------------------" << endl;

int choice = -1;

cout << "Enter the choice : ";

cin >> choice;

switch(choice)

{

case 1:

initialize\_array(array, N);

populating\_random\_values(array);

cout << "displaying first 10 numbers " << array;

sequential\_execution = analysis([&] { sequential\_min(array); });

parallel\_execution = analysis([&] { parallel\_min(array); });

cout << "Sequential Min : " << sequential\_execution << " ms" << endl;

cout << "Parallel Min (16): " << parallel\_execution << " ms" << endl;

cout << "speed up " << (float)(sequential\_execution / (float) parallel\_execution) << " ms" << endl;

break;

case 2:

initialize\_array(array, N);

populating\_random\_values(array);

cout << "displaying first 10 numbers " << array;

sequential\_execution = analysis([&] { sequential\_max(array); });

parallel\_execution = analysis([&]{ parallel\_max(array); });

cout << "Sequential Max : " << sequential\_execution << " ms" << endl;

cout << "Parallel Max (16): " << parallel\_execution << " ms" << endl;

cout << "speed up " << (float)(sequential\_execution / (float) parallel\_execution) << " ms" << endl;

break;

case 3:

initialize\_array(array, N);

populating\_random\_values(array);

cout << "displaying first 10 numbers " << array;

sequential\_execution = analysis([&] { sequential\_sum(array); });

parallel\_execution = analysis([&] { parallel\_sum(array); });

cout << "Sequential Sum : " << sequential\_execution << " ms" << endl;

cout << "Parallel Sum (16): " << parallel\_execution << " ms" << endl;

cout << "speed up " << (float)(sequential\_execution / (float) parallel\_execution) << " ms" << endl;

break;

case 4:

initialize\_array(array, N);

populating\_random\_values(array);

cout << "displaying first 10 numbers " << array;

sequential\_execution = analysis([&] { sequential\_average(array); });

parallel\_execution = analysis([&] { parallel\_average(array); });

cout << "Sequential Sum : " << sequential\_execution << " ms" << endl;

cout << "Parallel Sum (16): " << parallel\_execution << " ms" << endl;

cout << "speed up " << (float)(sequential\_execution / (float) parallel\_execution) << " ms" << endl;

break;

case 5:

flag = false;

break;

default:

cout << "Invalid choice !!! " << endl;

break;

}

}

return 0;

}

void initialize\_array(vector<int>& array, int N) {

cout << "Enter the size of the array : ";

cin >> N;

array.resize(max(5, N));

return;

}

// Min, Max, Sum, Average Sequential Functions ---------------------------

void sequential\_average(vector<int>& array) {

cout << "------ Sequential Average -----" << endl;

long long int N = (long long int)array.size();

long long int sum = 0;

float average = 0;

for (int i = 0; i < N; i++)

sum += array[i];

average = (sum / N);

cout << "Sequential Average " << average << endl;

}

void sequential\_max(vector<int>& array) {

cout << " ----- Sequential Max ------" << endl;

int N = array.size();

long long int max\_value = LONG\_MIN;

for (int i = 0; i < N; i++)

if (array[i] > max\_value)

max\_value = array[i];

cout << "Sequential max : " << max\_value << endl;

}

void sequential\_min(vector<int>& array) {

cout << " ----- Sequential Min ------" << endl;

int N = array.size();

long long int min\_value = LONG\_MAX;

for (int i = 0; i < N; i++)

if (array[i] < min\_value)

min\_value = array[i];

cout << "Sequential min : " << min\_value << endl;

}

void sequential\_sum(vector<int>& array) {

cout << " ----- Sequential Sum ----- " << endl;

int N = array.size();

long long int sum = 0;

for (int i = 0; i < N; i++)

sum += array[i];

cout << "Sequential sum : " << sum << endl;

}

// ---- Min, Max, Sum, Average Parallel Functions ------------------------

void parallel\_average(vector<int>& array) {

cout << " ----- Parallel Average -----" << endl;

long long int N = (long long int)array.size();

long long int sum = 0;

float average = 0;

#pragma omp parallel for reduction(+:sum)num\_threads(16)

for (int i = 0; i < N; i++)

sum += array[i];

average = (sum / N);

cout << "Parallel Average :" << average << endl;

}

void parallel\_max(vector<int>& array) {

cout << " ----- Parallel Max ----- " << endl;

int N = array.size();

long long int max\_value = LONG\_MIN;

#pragma omp parallel for reduction(max:max\_value) num\_threads(16)

for (int i = 0; i < N; i++)

if (array[i] > max\_value)

max\_value = array[i];

cout << "Parallel Max : " << max\_value << endl;

}

void parallel\_sum(vector<int>& array) {

cout << " ----- Parallel Sum -----" << endl;

int N = array.size();

long long int sum = 0;

#pragma omp parallel for reduction(+:sum)num\_threads(16)

for (int i = 0; i < N; i++)

sum += array[i];

cout << "Parallel Sum : " << sum << endl;

}

void parallel\_min(vector<int>& array) {

cout << " ----- Parallel Min ----- " << endl;

int N = array.size();

long long int min\_value = LONG\_MAX;

#pragma omp parallel for reduction(min:min\_value) num\_threads(16)

for (int i = 0; i < N; i++)

if (array[i] < min\_value)

min\_value = array[i]

cout << "Parallel Min : " << min\_value << endl;

}

int analysis(std :: function<void()> function) {

// subtract end from start timepoints and cast it to required unit, nanoseconds, microseconds, milliseconds etc.

auto start = high\_resolution\_clock::now();

// Executing function

function();

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(end - start);

return duration.count();

}

// utility functions

template <typename T> ostream& operator << (ostream& console, const vector<T>& array) {

int N = array.size();

for (int i = 0; i < min(10, N); i++)

console << array[i] << " ";

console << endl;

return console;

}

void populating\_random\_values(vector<int>& array) {

int N = array.size();

for (int i = 0; i < N; i++)

array[i] = 1 + (rand() % 1000); // feeding the array values between 1 and 1000 randomly

return;

}

Assignment -4

Matrix Multiplication

Code:

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cstdio>

#include <ctime>

#include <stdio.h>

#include <stdlib.h> #include <time.h>

// Kernel code

\_\_global\_\_ void calc\_prod\_cuda(int\* A, int\* B, int\* C, int rows\_a, int cols\_a, int rows\_b, int cols\_b) {

// get row, column from block and therd index int g = blockIdx.x \*

blockDim.x + threadIdx.x;

int col = g / rows\_a, row = g % rows\_a;

// calcuate prod for a cell C[row \* rows\_b + col]

= 0; for (int i = 0; i < cols\_b; i++) {

C[row \* cols\_b + col] += A[row \* cols\_a + i]\*B[i \* cols\_b + col];

}

}

// serial prouduct method

void calc\_prod\_serial(int \* A, int\* B, int\* C, int rows\_a, int cols\_a, int rows\_b, int cols\_b) {

// traverse rows

for (int i=0; i < rows\_a; i++) { // traverse

column for (int j=0; j < cols\_b; j++) {

// calcuate prod for a cell C[i \* cols\_b +j] = 0;

for (int k=0; k < cols\_b; k++) {

C[i \* cols\_b + j] += A[i \* cols\_a + k] \* B[k \* cols\_b + j];

}

}

}

}

void initialize\_matrix(

int \*host\_a, int \*host\_b, int \*host\_prod, // Host matrices

int rows\_a, int cols\_a, // dimenstin of A

int rows\_b, int cols\_b // dimensions of B

) {

printf("Initializing matrix..\n");

//initialize A, B

for (int i = 0; i < rows\_a \* cols\_a; i++) {

host\_a[i] = i;

}

for (int i = 0; i < rows\_b \* cols\_b; i++) { host\_b[i] = i+i;

}

printf("Matrix initialized\n"); fflush(stdout); }

// function of print matrix

void display\_matrix(int \*matrix, int rows, int cols) {

for (int i = 0; i < rows; i++) { for (int j = 0; j < cols;

j++) {

printf("%d ", matrix[i \* cols + j]);

}

printf("\n");

}

}

// gpu matrix multiplication function void calculate\_cuda(

int \*host\_a, int \*host\_b, int \*host\_prod, // Host matrices int rows\_a, int

cols\_a, // dimenstin of A int rows\_b, int cols\_b, // dimensions of B

int rows\_prod, int cols\_prod, // dimensions of prod

bool show\_product

) {

// initialize matrix on device int \*device\_a, \*device\_b,

\*device\_prod; printf("\nCalculating PARALLEL..\n");

// Allocate on device

cudaMalloc((void\*\*) &device\_a, rows\_a \* cols\_a \* sizeof(int)); cudaMalloc((void\*\*) &device\_b,

rows\_b \* rows\_b \* sizeof(int)); cudaMalloc((void\*\*) &device\_prod, rows\_prod \* cols\_prod \*

sizeof(int));

// Copy host to device cudaMemcpy(

device\_a, host\_a, rows\_a \* rows\_b \* sizeof(int),

cudaMemcpyHostToDevice

);

cudaMemcpy(

device\_b, host\_b,

rows\_b \* cols\_b \* sizeof(int), cudaMemcpyHostToDevice

);

// Define grid and block dimensions dim3 blockDim(cols\_b);

dim3 gridDim(rows\_a);

clock\_t start\_time = clock();

// multiply

calc\_prod\_cuda <<<gridDim, blockDim>>> ( device\_a, device\_b,

device\_prod,

rows\_a, cols\_a,

rows\_b, cols\_b

);

// Copy the result back to the host

cudaMemcpy( host\_prod, device\_prod,

rows\_prod \* cols\_prod \* sizeof(int),

cudaMemcpyDeviceToHost

);

if (show\_product) { printf("\nProduct is:\n");

display\_matrix(host\_prod, rows\_prod, cols\_prod);

}

printf(

"\nProduct calculated in %f seconds\n",

(double)(clock() - start\_time) / CLOCKS\_PER\_SEC

);

fflush(stdout);

cudaFree(device\_a); cudaFree(device\_b);

cudaFree(device\_prod);

}

// serial matrix multiplication function void calculate\_serial(

int \*host\_a, int \*host\_b, int \*host\_prod, // Host matrices int rows\_a, int

cols\_a, // dimenstin of A int rows\_b, int cols\_b, // dimensions of B

int rows\_prod, int cols\_prod, // dimensions of prod

bool show\_product

) {

clock\_t start\_time = clock(); printf("\nCalculating Serial..\n");

calc\_prod\_serial( host\_a, host\_b, host\_prod,

rows\_a, rows\_b,

rows\_b, cols\_b

);

if (show\_product) { printf("\nProduct is:\n");

display\_matrix(host\_prod, rows\_prod, cols\_prod);

}

printf(

"\nProduct calculated in %f seconds\n",

(double)(clock() - start\_time) / CLOCKS\_PER\_SEC

);

fflush(stdout); }

void free\_matrix(int \*host\_a, int \*host\_b, int \*host\_prod) {

// free memory free(host\_a); free(host\_b);

free(host\_prod);

}

int main() { int i=1; while (true) {

if (i==1) {

int rows\_a, cols\_a, rows\_b, cols\_b, see\_prod;

printf("\nEnter dimensions of Matrix: "); scanf("%d", &rows\_a);

cols\_a = cols\_b = rows\_b = rows\_a;

printf("\nDo you want to see prouct? "); scanf("%d", &see\_prod);

printf("\n");

int \*A, \*B, \*prod;

// matrix size int rows\_prod = rows\_a;

int cols\_prod = cols\_b;

// allocate on host

A = (int\*) malloc (rows\_a \* cols\_a \* sizeof(int)); B = (int\*) malloc (rows\_b \*

cols\_b \* sizeof(int)); prod = (int\*) malloc (rows\_prod \* cols\_prod \* sizeof(int));

initialize\_matrix( A, B, prod,

rows\_a, cols\_a,

rows\_b, cols\_b

);

calculate\_cuda( A, B, prod, rows\_a,

cols\_a, rows\_b, cols\_b, rows\_prod,

cols\_prod, see\_prod

);

calculate\_serial( A, B, prod,

rows\_a, cols\_a, rows\_b, cols\_b,

rows\_prod, cols\_prod,

see\_prod

);

free\_matrix(A, B, prod);

} else { break;

}

printf("Enter 1 to calculate again? "); scanf("%d", &i);

}

}

Output:

**Vector Addition**

Code:

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <ctime>

#include <iostream> #include <time.h> using

namespace std;

\_\_global\_\_ void add(int\* A, int\* B, int\* C, int size) { int tid = blockIdx.x \*

blockDim.x + threadIdx.x;

if (tid < size) { C[tid] = A[tid] + B[tid];

}

}

void add\_serial(int \*A, int \*B, int\*C, int size) { for (int i=0; i<size; i++)

{

C[i] = A[i] + B[i];

}

}

void initialize(int\* vector, int size) { for (int i = 0; i < size;

i++) {

vector[i] = rand() % 10;

}

}

void print(int\* vector, int size) { for (int i = 0; i < size;

i++) {

cout << vector[i] << " ";

}

cout << endl;

}

int main() { int i = 1; while (i == 1) {

int N = 4; int\* A, \* B, \* C;

int vectorSize;

cout << "\nEnter size of Vector: "; cin >> vectorSize;

size\_t vectorBytes = vectorSize \* sizeof(int);