Λογισμικό και προγραμματισμός υψηλής επίδοσης

Εργασία 3

Contents

Question 1 1
α) 1
β) 3
γ) 3

Question 1

α)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#define N 512

// Total run time
double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
}
// Random matrix values
void initializeMatrix(float* mat, int n) {
    for (int i = 0; i < n * n; ++i) {
        mat[i] = (float)rand() / (float)RAND_MAX;
    }
}
// Compute
void complexMatMulKernel(const float* A, const float* B, const float*
C, const float* D,
                         float* E, float* F, int n) {

// Omp offload to GPU
    #pragma omp target teams distribute parallel for collapse(2) \
    map(to:A[0:n*n], B[0:n*n], C[0:n*n], D[0:n*n]) map(from:E[0:n*n],
F[0:n*n])

    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
```

```
// Initialize to zero before calculations
            float sumAC = 0.0f;
            float sumBD = 0.0f;
            float sumAD = 0.0f;
            float sumBC = 0.0f;


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

// retrieve values
                float a = A[row * n + k];
                float b = B[row * n + k];
                float c = C[k * n + col];
                float d = D[k * n + col];

// Calculate A,B,C,D
                sumAC += a * c;
                sumBD += b * d;
                sumAD += a * d;
                sumBC += b * c;
            }

// Calculate and store in E,F
            E[row * n + col] = sumAC - sumBD;
            F[row * n + col] = sumAD + sumBC;
        }
    }
}
int main() {
    double start_time = get_time();

// Random matrix
    srand(time(NULL));
    size_t size = N * N * sizeof(float);

    // Starting in cpu before transfering to gpu
    float* A = (float*)malloc(size);
    float* B = (float*)malloc(size);
    float* C = (float*)malloc(size);
    float* D = (float*)malloc(size);
    float* E = (float*)malloc(size);
    float* F = (float*)malloc(size);
    if (!A || !B || !C || !D || !E || !F) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    // Initialize matrixes
```

```
    initializeMatrix(A, N);
    initializeMatrix(B, N);
    initializeMatrix(C, N);
    initializeMatrix(D, N);

    // Running on cpu to offlaod
    complexMatMulKernel(A, B, C, D, E, F, N);


    free(A);
    free(B);
    free(C);
    free(D);
    free(E);
    free(F);

    double end_time = get_time();
    printf("Matrix size: %d x %d\n", N, N);
    printf("Total execution time: %.4f seconds\n", (end_time -
start_time) / 1000000.0);
    return 0;
}
```

β)

Οι υπολογισμοί παραμένουν ίδιοι με την cuda υλοποίηση της εργασίας 2. Οι αλλαγές μας έχουν να κάνουν με την λογική του omp, δηλαδή πλέον δεν χρειαζόμαστε εμείς να γράψουμε χειροκίνητα τις θέσεις μνήμης που θα καταλαμβάνουν τα δεδομένα στην gpu, αυτό γίνεται αυτόματα από τον compiler. Επιπλέον, δεν ορίζουμε το block size, απλά γράφουμε teams και ο compiler αυτόματα δημιουργεί ομάδες thread και τους αναθέτει εργασία. Δεν χρειάζεται να δημιουργήσουμε critical section, αφού το κάθε νήμα υπολογίζει διαφορετικό στοιχείο των μητρώων εξόδου, άρα δεν πειράζουν κοινές μεταβλητές. Τέλος, απλά ορίζουμε που να δώσει η cpu τα δεδομένα και μετα από που να τα πάρει. Ουσιαστικά η όλη λογική είναι η εξής γραμμή:

```
#pragma omp target teams distribute parallel for collapse(2) \
    map(to:A[0:n*n], B[0:n*n], C[0:n*n], D[0:n*n]) map(from:E[0:n*n],
F[0:n*n])
```

γ)

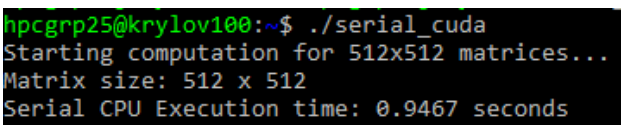
Για τον σειριακό κώδικα χρησιμοποιούμε αυτόν της εργασίας 2.

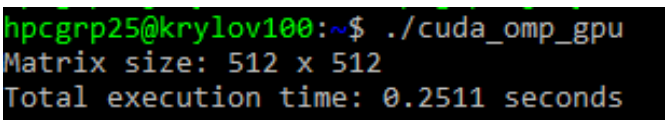
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

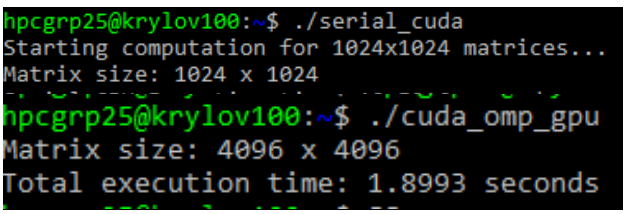
```c
#include <sys/time.h>

#define N 1024

double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
}

void initializeMatrix(float* mat, int n) {
    for (int i = 0; i < n * n; ++i) {
        mat[i] = (float)rand() / RAND_MAX;
    }
}

void complex_matrix_multiply(const float* A, const float* B, const
float* C, const float* D,
                             float* E, float* F, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            float sumAC = 0.0f;
            float sumBD = 0.0f;
            float sumAD = 0.0f;
            float sumBC = 0.0f;

            for (int k = 0; k < n; k++) {
                float a = A[i * n + k];
                float b = B[i * n + k];
                float c = C[k * n + j];
                float d = D[k * n + j];

                sumAC += a * c;
                sumBD += b * d;
                sumAD += a * d;
                sumBC += b * c;
            }

            E[i * n + j] = sumAC - sumBD;
            F[i * n + j] = sumAD + sumBC;
        }
    }
}

int main() {
    srand(time(NULL));
    size_t size = N * N * sizeof(float);
```

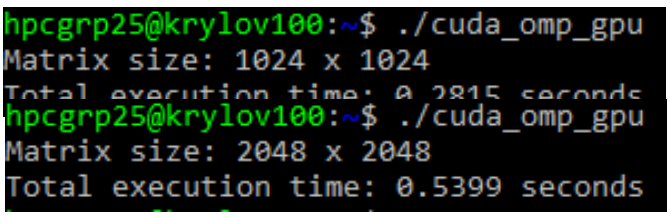
```
    float* A = (float*)malloc(size);
    float* B = (float*)malloc(size);
    float* C = (float*)malloc(size);
    float* D = (float*)malloc(size);
    float* E = (float*)malloc(size);
    float* F = (float*)malloc(size);

    if (!A || !B || !C || !D || !E || !F) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    initializeMatrix(A, N);
    initializeMatrix(B, N);
    initializeMatrix(C, N);
    initializeMatrix(D, N);

    printf("Starting computation for %dx%d matrices...\n", N, N);
    double start_time = get_time();

    complex_matrix_multiply(A, B, C, D, E, F, N);

    double end_time = get_time();
    double execution_time = (end_time - start_time) / 1000000.0;

    printf("Matrix size: %d x %d\n", N, N);
    printf("Serial CPU Execution time: %.4f seconds\n",
execution_time);

    free(A); free(B); free(C); free(D); free(E); free(F);
    return 0;
}
```

Πραγματοποιούμε πειράματα για N=512,1024, 2048, 4096 (για την cpu υλοποίηση έως 2048, καθώς ο χρόνος εκτέλεσης είναι πολύ μεγάλος):









Από τα πειράματα μας κατανοούμε πως πράγματι η απόδοση της gpu δεν συγκρίνεται σε καμία περίπτωση με σειριακό κώδικα που τρέχει σε cpu. Από τα αποτελέσματα της άσκησης 2, ακόμη και αν χρησιμοποιήσουμε και τους 24 πυρήνες της cpu με omp και πάλι παρατηρούμε ότι δεν συγκρίνεται με την gpu υλοποίηση το οποίο είναι λογικό, αφού ουσιαστικά ανταγωνίζεται χιλιάδες εξειδικευμένους πυρήνες. Το ενδιαφέρον είναι ότι συγκριτικά με την cuda υλοποίηση, η omp είναι αισθητά πιο αργή όσο αυξάνεται το μέγεθος του προβλήματος (για N>2048). Αυτό συμβαίνει πολύ απλά επειδή δεν γίνεται η αυτόματη δημιουργία block και κατανομή εργασίας στα νήματα να ανταγωνιστεί την εξειδικευμένη cuda, όπου ορίζουμε ρητά το κατάλληλο block size ανάλογα την gpu μας και επιπλέον έχουμε ρητή διαχείριση μνήμης αντί για αυτόματη. Συμπερασματικά, προφανώς η omp version είναι universal και για άλλες κάρτες γραφικών που δεν είναι nvidia και κανονικά θέλουν Open cl η' κάποιο άλλο library, άρα δεν χρειάζεται να γράφουμε κώδικα 2 φορές και επιπλέον είναι πιο απλή αφού δεν χρειάζεται να διαχειριστούμε την μνήμη και τα block.