Parallel Architectures - Homework 4

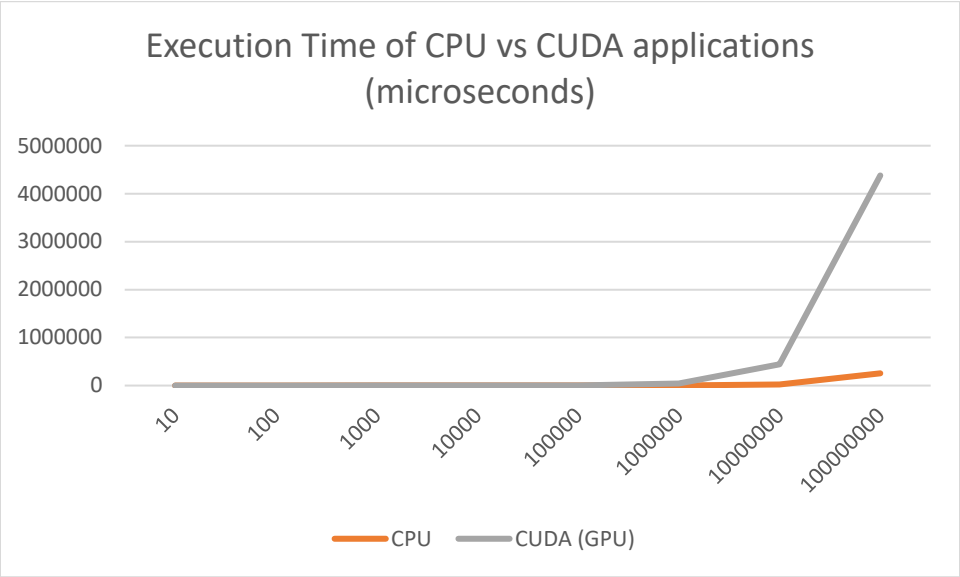
Şamil NART - 110510129

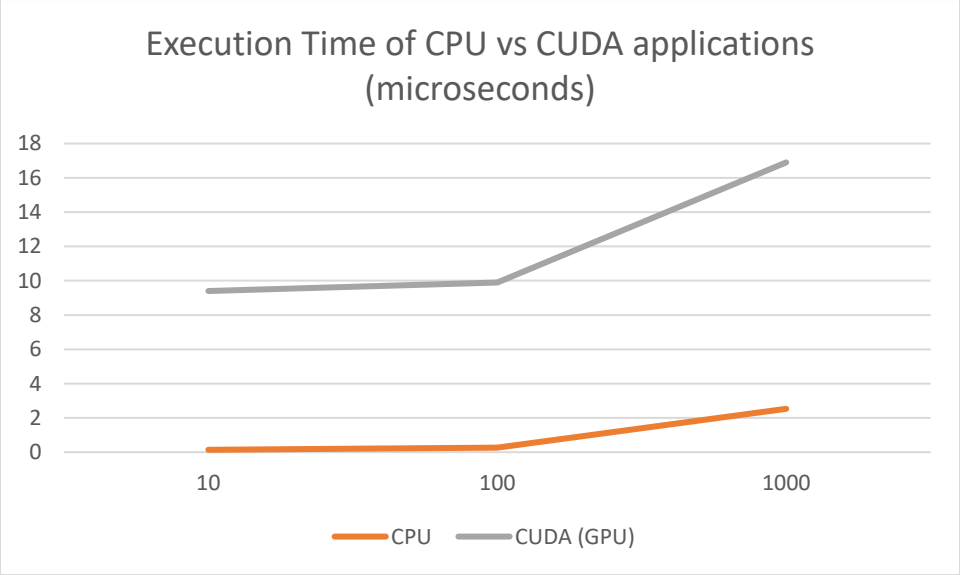
Executing the executables in AGÜ HPC node:

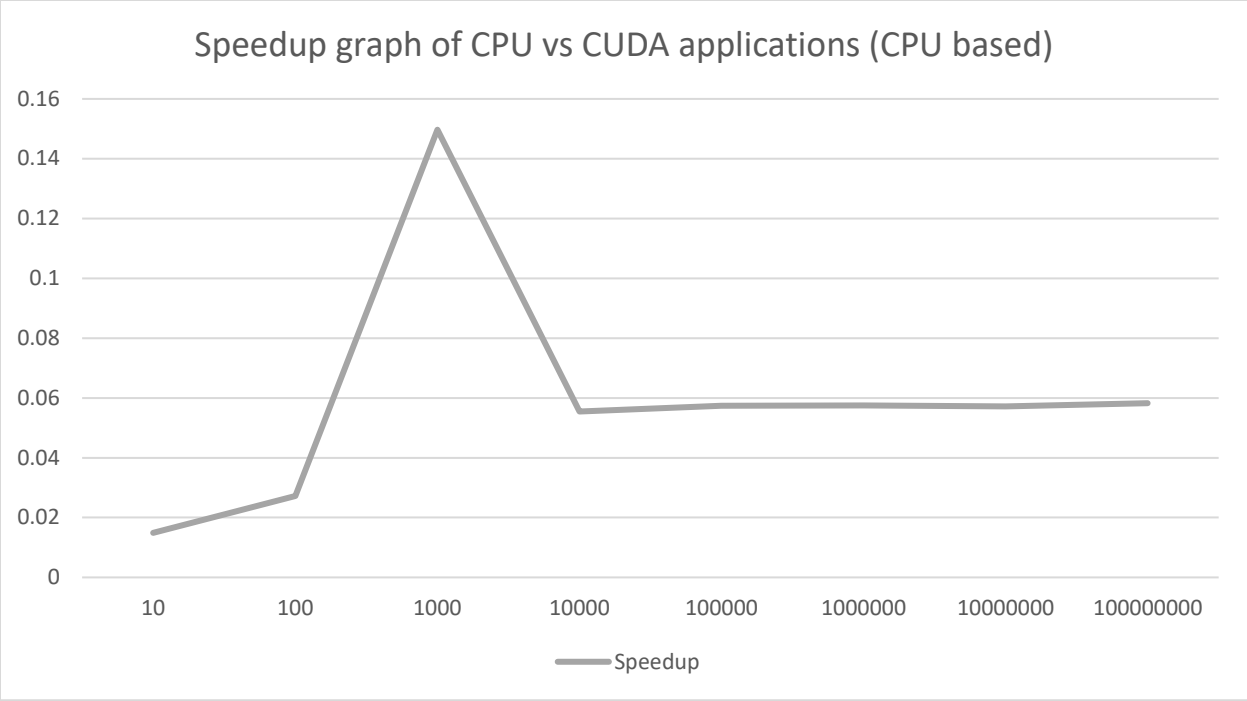
```
gystudent@dgx01:~/snart/cuda_experiment$ ./run.sh
---SINGLE THREADED---
---10.txt---
Result: -26.251198
Time: 0.139342us
---100.txt---
Result: 409.539703
Time: 0.272532us
---1000.txt---
Result: -1067.029541
Time: 2.529141us
---10000.txt---
Result: -2450.917969
Time: 25.300735us
---100000.txt---
Result: 28296.759766
Time: 251.911577us
---1000000.txt---
Result: 53613.007812
Time: 2514.498039us
---10000000.txt---
Result: -5302.839844
Time: 25440.908019us
---100000000.txt---
Result: 376913.281250
Time: 255057.624836us
---CUDA---
---10.txt---
Result: -26.251198
Time: 9.393698us
---100.txt---
Result: 409.539703
Time: 9.892861us
---1000.txt---
Result: -1067.029541
Time: 16.867394us
---10000.txt---
Result: -2450.923828
Time: 456.122023us
---100000.txt---
Result: 28296.800781
Time: 4389.883401us
---1000000.txt---
Result: 53611.230469
Time: 43732.837130us
---10000000.txt---
Result: -5304.196289
Time: 444690.760527us
---100000000.txt---
Result: 376986.531250
Time: 4379581.140791us
gystudent@dgx01:~/snart/cuda_experiment$
```

Methodology: Given dot product algorithm was implemented as a single-threaded CPU application and a NVIDIA CUDA GPU application. Each application ran the algorithm on data with varying sizes 512 times and the average of execution time was taken.

Results:







Discussion: The CPU version of the algorithm acts pretty linear in execution time, spending around 0.01 microseconds per each element in the arrays. For the CUDA version of the algorithm, the input had to be sent to the GPU in 1000 element chunks as CUDA is limited in how many cores can be used to compute concurrently. According to results, CUDA version has an almost exponential increase in execution time having an order of magnitude difference in execution time compared to the CPU version. One conclusion we can reach from the data is, while GPUs are fast at working on big amounts of data in parallel, communication between the CPU and GPU is the bottleneck to our program. Evidently it takes too much time to transfer data from main memory to GPU memory (especially as we do it in 1000 element chunks, increasing count of transfers we have to do) while the CUDA cores only execute one instruction, multiplying two elements. If our algorithm did more processing within CUDA cores, we may see an execution time advantage compared to using the CPU.

Source code: main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

const int TRY_COUNT = 512;

void fetch_data(char *file_name, float **mat1, float **mat2, int *mat_size) {
    FILE *file = fopen(file_name, "r");

    fscanf(file, "SizeA= %d", mat_size);
    float *matA = (float*) malloc(sizeof(float) * (*mat_size));
    for (int i = 0; i < (*mat_size); i++) {
        fscanf(file, "%f", &(matA[i]));
    }
```

```
    fscanf(file, " ");

    fscanf(file, "SizeB= %d", mat_size);
    float *matB = (float*) malloc(sizeof(float) * (*mat_size));
    for (int i = 0; i < (*mat_size); i++) {
        fscanf(file, "%f", &(matB[i]));
    }

    *mat1 = matA;
    *mat2 = matB;
    fclose(file);
}

float dot_product(float *mat1, float *mat2, int mat_size) {
    float sum = 0;
    for (int i = 0; i < mat_size; i++) {
        sum += mat1[i] * mat2[i];
    }

    return sum;
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Please provide file name!");
        exit(1);
    }

    float *mat1, *mat2;
    int mat_size;
    fetch_data(argv[1], &mat1, &mat2, &mat_size);
    double time_total = 0;
    for (int i = 0; i < TRY_COUNT; i++) {
        double start = omp_get_wtime();
        dot_product(mat1, mat2, mat_size);
        double end = omp_get_wtime();
        time_total += end - start;
    }

    printf("Result: %f\n", dot_product(mat1, mat2, mat_size));
    printf("Time: %fus\n", 1000000 * (time_total / ((double) TRY_COUNT)));
    free(mat1);
    free(mat2);
    return 0;
}
```

Source code: par_main.cu

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

```
const int TRY_COUNT = 512;

void fetch_data(char *file_name, float **mat1, float **mat2, int *mat_size) {
    FILE *file = fopen(file_name, "r");

    fscanf(file, "SizeA= %d", mat_size);
    float *matA = (float*) malloc(sizeof(float) * (*mat_size));
    for (int i = 0; i < (*mat_size); i++) {
        fscanf(file, "%f", &(matA[i]));
    }

    fscanf(file, " ");

    fscanf(file, "SizeB= %d", mat_size);
    float *matB = (float*) malloc(sizeof(float) * (*mat_size));
    for (int i = 0; i < (*mat_size); i++) {
        fscanf(file, "%f", &(matB[i]));
    }

    *mat1 = matA;
    *mat2 = matB;
    fclose(file);
}

__global__
void dot_product(float *mat1, float *mat2, float *tmp, int *mat_size, float *result) {
    tmp[threadIdx.x] = mat1[threadIdx.x] * mat2[threadIdx.x];
    __syncthreads();

    if(threadIdx.x == 0) {
        float sum = 0;
        for(int i = 0; i < (*mat_size); i++) {
            sum += tmp[i];
        }
        *result = sum;
    }
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Please provide file name!");
        exit(1);
    }

    float *mat1, *mat2;
    int mat_size;
    fetch_data(argv[1], &mat1, &mat2, &mat_size);
    float *cuda_mat1, *cuda_mat2, *cuda_tmp, *cuda_result;
    int *cuda_mat_size;
    float result_initial_value = 0;
    double time_total = 0;

    float result;
    if (mat_size <= 1000) {
```

```
        cudaMalloc(&cuda_mat1, sizeof(float) * mat_size);
        cudaMalloc(&cuda_mat2, sizeof(float) * mat_size);
        cudaMalloc(&cuda_tmp, sizeof(float) * mat_size);
        cudaMalloc(&cuda_mat_size, sizeof(int));
        cudaMalloc(&cuda_result, sizeof(float));
        cudaMemcpy(cuda_mat1, mat1, sizeof(float) * mat_size, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_mat2, mat2, sizeof(float) * mat_size, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_mat_size, &mat_size, sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_result, &result_initial_value, sizeof(float), cudaMemcpyHostToDevice);

        for (int i = 0; i < TRY_COUNT; i++) {
            double start = omp_get_wtime();
            dot_product<<< 1, mat_size >>>(cuda_mat1, cuda_mat2, cuda_tmp, cuda_mat_size,
cuda_result);
            cudaDeviceSynchronize();
            double end = omp_get_wtime();
            time_total += end - start;
        }

        cudaMemcpy(&result, cuda_result, sizeof(float), cudaMemcpyDeviceToHost);
    } else {
        int mat_size_dummy = 1000;
        cudaMalloc(&cuda_mat1, sizeof(float) * mat_size_dummy);
        cudaMalloc(&cuda_mat2, sizeof(float) * mat_size_dummy);
        cudaMalloc(&cuda_tmp, sizeof(float) * mat_size_dummy);
        cudaMalloc(&cuda_mat_size, sizeof(int));
        cudaMalloc(&cuda_result, sizeof(float));
        cudaMemcpy(cuda_mat_size, &mat_size_dummy, sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_result, &result_initial_value, sizeof(float), cudaMemcpyHostToDevice);

        for (int i = 0; i < TRY_COUNT; i++) {
            double start = omp_get_wtime();
            result = 0;
            for (int cur = 0; cur < mat_size; cur += 1000) {
                cudaMemcpy(cuda_mat1, &(mat1[cur]), sizeof(float) * mat_size_dummy,
cudaMemcpyHostToDevice);
                cudaMemcpy(cuda_mat2, &(mat2[cur]), sizeof(float) * mat_size_dummy,
cudaMemcpyHostToDevice);
                dot_product<<< 1, 1000 >>>(cuda_mat1, cuda_mat2, cuda_tmp, cuda_mat_size,
cuda_result);
                cudaDeviceSynchronize();
                float result_tmp;
                cudaMemcpy(&result_tmp, cuda_result, sizeof(float), cudaMemcpyDeviceToHost);
                cudaMemcpy(cuda_result, &result_initial_value, sizeof(float),
cudaMemcpyHostToDevice);
                result += result_tmp;
            }
            double end = omp_get_wtime();
            time_total += end - start;
        }
    }

    printf("Result: %f\n", result);
    printf("Time: %fus\n", 1000000 * (time_total / ((double) TRY_COUNT)));
```

```
    cudaFree(cuda_mat1);
    cudaFree(cuda_mat2);
    cudaFree(cuda_tmp);
    cudaFree(cuda_mat_size);
    cudaFree(cuda_result);
    return 0;
}
```