МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

Кафедра дискретного аналізу та інтелектуальних систем

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ
ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №8
“РОЗПАРАЛЕЛЕННЯ МНОЖЕННЯ МАТРИЦЬ CUDA”

Роботу виконав:
Сенюк Віталій Васильович
Студент групи ПМІ-33с

Перевірив:
Доц. Гошко Б.М.
кафедри програмування
Львівського національного
Університету імені Івана Франка

Львів – 2023

Тема: Розпаралелення множення матриць CUDA.

Мета роботи: Написати програму обчислення множення двох матриць на CUDA ядрах.

Завдання

На основі програмно-апаратної архітектури паралельних обчислень CUDA реалізувати один з методів лабораторних 2,3,6,7. Якщо не маєте GPU NVIDIA, то використовуйте OpenCL.

Продемонструвати результати матриць різної розмірності та порівняти з результатами відповідної попередньої лабораторної роботи

Виконання роботи

1. Створюю CUDA ядро яке обчислює множення матриць.

```
const int TILE_SIZE = 16;

__global__ void MatrixMultKernel(int* c, const int* a, const int* b, int m1_rows, int m1_cols, int m2_cols) {
    __shared__ int a_tile[TILE_SIZE][TILE_SIZE];
    __shared__ int b_tile[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int temp_value = 0;
    for (int tile_idx = 0; tile_idx < (m1_cols + TILE_SIZE - 1) / TILE_SIZE; ++tile_idx) {
        if (row < m1_rows && tile_idx * TILE_SIZE + threadIdx.x < m1_cols)
            a_tile[threadIdx.y][threadIdx.x] = a[row * m1_cols + tile_idx * TILE_SIZE + threadIdx.x];
        else
            a_tile[threadIdx.y][threadIdx.x] = 0;

        if (col < m2_cols && tile_idx * TILE_SIZE + threadIdx.y < m1_cols)
            b_tile[threadIdx.y][threadIdx.x] = b[(tile_idx * TILE_SIZE + threadIdx.y) * m2_cols + col];
        else
            b_tile[threadIdx.y][threadIdx.x] = 0;

        __syncthreads();

        for (int i = 0; i < TILE_SIZE; ++i) {
            temp_value += a_tile[threadIdx.y][i] * b_tile[i][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < m1_rows && col < m2_cols)
        c[row * m2_cols + col] = temp_value;
}
```

Замість простого множення, використовую плиточний підхід, який дозволяє зменшити осяг передачі даних та збільшити обсяг кешу, що сприяє зменшеню затримок пам’яті та покращенню загального пристосування до ресурвіс GPU.

2. Створюю метод для генерації матриці.

```
void generateRandomMatrix(int* matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i * cols + j] = rand() % 10;
        }
    }
}
```

3. Розробляю метод для слугує для виділення пам’яті для матриць на пристрої (GPU), копіювання даних із хоста на пристрій, запуск ядра CUDA MatrixMultKernel для виконання множення матриці та копіювання результату назад із пристрою на хост.

```
cudaError_t multiplyWithCuda(int* c, const int* a, const int* b, int m1_rows, int m1_cols,
                             int m2_rows, int m2_cols) {
    int* dev_a = 0;
    int* dev_b = 0;
    int* dev_c = 0;
    cudaError_t cudaStatus;
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!");
        goto Error;
    }
    cudaMalloc((void**)&dev_c, m1_rows * m2_cols * sizeof(int));
    cudaMalloc((void**)&dev_a, m1_rows * m1_cols * sizeof(int));
    cudaMalloc((void**)&dev_b, m2_rows * m2_cols * sizeof(int));
    cudaMemcpy(dev_a, a, m1_rows * m1_cols * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    cudaMemcpy(dev_b, b, m2_rows * m2_cols * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    dim3 blockSize(TILE_SIZE, TILE_SIZE);
    dim3 gridSize((m2_cols + blockSize.x - 1) / blockSize.x, (m1_rows + blockSize.y - 1) / blockSize.y);
    MatrixMultKernel << <gridSize, blockSize >> > (dev_c, dev_a, dev_b, m1_rows, m1_cols, m2_cols);
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "MatrixMultKernel launch failed!");
        goto Error;
    }
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code!");
        goto Error;
    }
    cudaMemcpy(c, dev_c, m1_rows * m2_cols * sizeof(int), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    return cudaStatus;
}
```

4. Запуск програми.

```cpp
int main() {
    srand(time(NULL));

    int matrix_size_limit = 200000;
    int size_factor = 10;

    for (int n = 1; n <= matrix_size_limit; n *= size_factor) {
        int m1_rows = 20 * n;
        int m1_cols = 30 * n;
        int m2_rows = 30 * n;
        int m2_cols = 20 * n;

        int* a_data = new int[m1_rows * m1_cols];
        int* b_data = new int[m2_rows * m2_cols];

        generateRandomMatrix(a_data, m1_rows, m1_cols);
        generateRandomMatrix(b_data, m2_rows, m2_cols);

        int* c = new int[m1_rows * m2_cols]();

        auto start_time = std::chrono::high_resolution_clock::now();

        cudaError_t cudaStatus = multiplyWithCuda(c, a_data, b_data, m1_rows, m1_cols, m2_rows, m2_cols);

        auto end_time = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_elapsed = end_time - start_time;

        if (cudaStatus != cudaSuccess) {
            fprintf(stderr, "multiplyWithCuda failed!");
            return 1;
        }

        printf("Time consumed by multiplying matrix of size %dx%d: %f seconds\n", m1_rows, m2_cols, time_elapsed.count());

        cudaStatus = cudaDeviceReset();
        if (cudaStatus != cudaSuccess) {
            fprintf(stderr, "cudaDeviceReset failed!");
            return 1;
        }

        delete[] a_data;
        delete[] b_data;
        delete[] c;
    }

    return 0;
}
```

5. Тестую та порівнюю час без та із використанням CUDA.

5.1. Мала матриця.

```
Start iteration
Matrix size [20][30]

SyncMatrixAddition time = 00:00:00.0014164

AsyncMatrixAddition time = 00:00:00.0011702 with 2 threads
Acceleration = 1,2103913860878481 Efficiency = 0,6051956930439241

AsyncMatrixAddition time = 00:00:00.0010277 with 4 threads
Acceleration = 1,3782232168920892 Efficiency = 0,3445558042230223

AsyncMatrixAddition time = 00:00:00.0015100 with 8 threads
Acceleration = 0,9380132450331126 Efficiency = 0,11725165562913907
```

5.2. Мала матриця CUDA.

```
matrix of size 20x20: 0.096057 seconds
```

Як можемо бачити, час виконяння більший, і не ефективно використовувати CUDA на малих даних.

5.3. Середня матриця.

```
Matrix size [200][300]

SyncMatrixAddition time = 00:00:00.3302434

AsyncMatrixAddition time = 00:00:00.1918719 with 2 threads
Acceleration = 1,7211660488065215 Efficiency = 0,8605830244032607

AsyncMatrixAddition time = 00:00:00.1444563 with 4 threads
Acceleration = 2,2861128244320255 Efficiency = 0,5715282061080064

AsyncMatrixAddition time = 00:00:00.1247004 with 8 threads
Acceleration = 2,6482946325753565 Efficiency = 0,33103682907191956
```

5.4. Середня матриця CUDA.

```
matrix of size 200x200: 0.053981 seconds
```

У даному випадку, є відчутній приріст порівняно з використанням звичайних cpu потоків. (є велика погрішність між малою та середньою матрицею, так як алгоритм виконується дуже швидко).

5.5.Велика матириця.

```
Matrix size [2000][3000]

SyncMatrixAddition time = 00:11:20.4914434

AsyncMatrixAddition time = 00:06:38.4938911 with 2 threads
Acceleration = 1,7076584073135368 Efficiency = 0,8538292036567684

AsyncMatrixAddition time = 00:04:41.1814213 with 4 threads
Acceleration = 2,4201152418031397 Efficiency = 0,6050288104507849
```

5.6. Велика матриця CUDA.

```
matrix of size 2000x2000: 0.729880 seconds
```

Якщо порівнювати з звичайними потоками, то різниця відчутно більша.

5.7. Дуже велика матриця CUDA.

```
matrix of size 20000x20000: 824.260321 seconds
```

За допомогою cuda зміг помножити матриці, які б виконувались цілі дні на звичайних потоках.

6. Сумуючи даний експеримент, множення матриць у середовищі GPU нам надає велику паралельність та масштабованість, однак присутні обмеження ресурсів пам’яті, також недоліком є те, що потрібно передавати дані між глобальною пам’яттю GPU та системною пам’яттю, що збільшує накладні витрати.

Висновки

У результаті виконання роботи:

1. Дослідив різницю між виконанням алгоритму у CPU та GPU.
2. Дізнався, про плиточний алгоритм множення матриць.

Список використаних джерел

1. Вільний простір інтернету.