Міністерство освіти і науки України

Львівський національний університет Ім. І. Франка

Факультет прикладної математики та інформатики

Кафедра дискретного аналізу та інтелектуальних систем

Паралельні та розподілені обчислення

Лабораторна робота №8

Програмно-апаратна архітектура паралельних обчислень CUDA

Роботу виконала:

Студентка ПМІ-33

Багінська Маргарита

Прийняв:

доц. Пасічник Т.В.

Львів 2023

Тема: Програмно-апаратна архітектура паралельних обчислень CUDA

Мета: Реалізувати алгоритм множення матриць використовуючи програмно-апаратну архітектуру паралельних обчислень CUDA.

Виконання

- Файл **kernel.h**

```
#ifndef KERNEL_H_
#define KERNEL_H_

void matrixMultiplication(float* A, float* B, float* C, int N);

#endif
```

Файл "kernel.h" містить оголошення функції matrixMultiplication, яка призначена для виконання множення двох матриць, використовуючи технологію CUDA.

- Файл **dev_array.h**

```
#ifndef _DEV_ARRAY_H_
#define _DEV_ARRAY_H_

#include <stdexcept>
#include <algorithm>
#include <cuda_runtime.h>

template <class T>
class dev_array
{
    // public functions
public:
    explicit dev_array()
        : start_(0),
        end_(0)
    {}

    // constructor
    explicit dev_array(size_t size)
    {
        allocate(size);
    }
    // destructor
    ~dev_array()
    {
        free();
    }

    // resize the vector
    void resize(size_t size)
    {
        free();
        allocate(size);
    }

    // get the size of the array
    size_t getSize() const
    {
        return end_ - start_;
    }

    // get data
    const T* getData() const
    {
        return start_;
    }

    T* getData()
    {
        return start_;
    }

    // set
    void set(const T* src, size_t size)
    {
        size_t min = std::min(size, getSize());
        cudaError_t result = cudaMemcpy(start_, src, min * sizeof(T), cudaMemcpyHostToDevice);
        if (result != cudaSuccess)
        {
            throw std::runtime_error("failed to copy to device memory");
        }
    }
    // get
    void get(T* dest, size_t size)
    {
        size_t min = std::min(size, getSize());
        cudaError_t result = cudaMemcpy(dest, start_, min * sizeof(T), cudaMemcpyDeviceToHost);
        if (result != cudaSuccess)
        {
            throw std::runtime_error("failed to copy to host memory");
        }
    }


    // private functions
private:
    // allocate memory on the device
    void allocate(size_t size)
    {
        cudaError_t result = cudaMalloc((void**)&start_, size * sizeof(T));
        if (result != cudaSuccess)
        {
            start_ = end_ = 0;
            throw std::runtime_error("failed to allocate device memory");
        }
        end_ = start_ + size;
    }

    // free memory on the device
    void free()
    {
        if (start_ != 0)
        {
            cudaFree(start_);
            start_ = end_ = 0;
        }
    }

    T* start_;
    T* end_;
};

#endif
```

Файл "dev_array.h" містить клас шаблонів dev_array, який є обгорткою над пам'яттю пристрою (пам'ять GPU) для простої алокації пам'яті, зміни розміру, зчитування та запису. Цей клас допомагає забезпечити легкий та безпечний спосіб керування пам'яттю пристроїв.

- Файл kernel.cu

```cpp
#include <math.h>
#include <iostream>
#include "cuda_runtime.h"
#include "kernel.h"
#include <stdlib.h>
#include "device_launch_parameters.h"

using namespace std;

__global__ void matrixMultiplicationKernel(float* A, float* B, float* C, int N) {

    int ROW = blockIdx.y * blockDim.y + threadIdx.y;
    int COL = blockIdx.x * blockDim.x + threadIdx.x;

    float tmpSum = 0;

    if (ROW < N && COL < N) {
        // each thread computes one element of the block sub-matrix
        for (int i = 0; i < N; i++) {
            tmpSum += A[ROW * N + i] * B[i * N + COL];
        }
    }
    C[ROW * N + COL] = tmpSum;
}


void matrixMultiplication(float* A, float* B, float* C, int N) {

    // declare the number of blocks per grid and the number of threads per block
    // use 1 to 512 threads per block
    dim3 threadsPerBlock(N, N);
    dim3 blocksPerGrid(1, 1);
    if (N * N > 512) {
        threadsPerBlock.x = 512;
        threadsPerBlock.y = 512;
        blocksPerGrid.x = ceil(double(N) / double(threadsPerBlock.x));
        blocksPerGrid.y = ceil(double(N) / double(threadsPerBlock.y));
    }

    matrixMultiplicationKernel << <blocksPerGrid, threadsPerBlock >> > (A, B, C, N);
}

```

Файл "kernel.cu" містить визначення функції **matrixMultiplication**, яка використовує ядро CUDA **matrixMultiplicationKernel** для перемноження двох матриць на GPU. Основні частини коду:

__global__ void matrixMultiplicationKernel(float* A, float* B, float* C, int N):

Це ядро CUDA, яке виконує матричне множення на рівні елементів. Кожен потік відповідає за обчислення одного елемента результуючої матриці C. Ітерація через рядки і стовпці вхідних матриць A та B відбувається всередині цього ядра. Ядро виконується у паралель для кожної комбінації blockIdx та threadIdx, середнє значення цих значень перемножується з розмірами блоку, щоб отримати індекс рядка (ROW) та стовпця (COL).

Функція **void matrixMultiplication(float* A, float* B, float* C, int N):**

керівництво CUDA, яке приймає вказівники на дані матриць A, B та C та розмір матриці N. Функція налаштовує рядки, стовпці блоків на сітці та потоки на блок для керування виконанням ядра. Він використовує блоки та потоки для паралельного обчислення елементів результуючої матриці C.

- Файл matrixmul.cu

```cpp
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>
#include "kernel.h"
#include "dev_array.h"
#include <math.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main()
{
    // Perform matrix multiplication C = A*B
    // where A, B and C are NxN matrices
    int N = 30000;
    int SIZE = N * N;

    // Allocate memory on the host
    vector<float> h_A(SIZE);
    vector<float> h_B(SIZE);
    vector<float> h_C(SIZE);

    // Initialize matrices on the host
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            h_A[i * N + j] = sin(i);
            h_B[i * N + j] = cos(j);
        }
    }

    // Measure time taken by CUDA
    auto start_cuda = high_resolution_clock::now();

    // Allocate memory on the device
    dev_array<float> d_A(SIZE);
    dev_array<float> d_B(SIZE);
    dev_array<float> d_C(SIZE);

    d_A.set(&h_A[0], SIZE);
    d_B.set(&h_B[0], SIZE);

    matrixMultiplication(d_A.getData(), d_B.getData(), d_C.getData(), N);
    cudaDeviceSynchronize();

    d_C.get(&h_C[0], SIZE);
    cudaDeviceSynchronize();

    auto stop_cuda = high_resolution_clock::now();
    auto duration_cuda = duration_cast<milliseconds>(stop_cuda - start_cuda);

    float* cpu_C;
    cpu_C = new float[SIZE];

    // Measure time taken by CPU
    auto start_cpu = high_resolution_clock::now();

    // Now do the matrix multiplication on the CPU
    float sum;
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            sum = 0.f;
            for (int n = 0; n < N; n++) {
                sum += h_A[row * N + n] * h_B[n * N + col];
            }
            cpu_C[row * N + col] = sum;
        }
    }

    auto stop_cpu = high_resolution_clock::now();
    auto duration_cpu = duration_cast<milliseconds>(stop_cpu - start_cpu);

    cout << "[" << N << "]" << "[" << N << "]" << endl;
    cout << "CUDA execution time: " << duration_cuda.count() << " ms = " << duration_cuda.count() * 0.001 << " s" << endl;
    cout << "CPU execution time: " << duration_cpu.count() << " ms = " << duration_cpu.count() * 0.001 << " s" << endl;

    // Release device memory
    cudaFree(d_A.getData());
    cudaFree(d_B.getData());
    cudaFree(d_C.getData());

    // Release host memory
    delete[] cpu_C;

    return 0;
}
```

В цьому файлі реалізовано порівняння часу виконання матричного множення на CPU та GPU (CUDA). Використовуючи структуру chrono, код обчислює час виконання операцій на CPU та GPU.

Результати

Згадаємо результати третьої лабораторної роботи, де ми множили матриці синхронно на паралельно на CPU.

```
[2000][2000] * [2000][2000]
Sync time ~ 00:00:54.6495798
Async time ~ 00:00:09.2754881 with 8 threads
Acceleration ~ 5,891827924397854
Efficiency ~ 0,7364784905497318
```

Тепер результати CUDA:

```
[2000][2000]
CUDA execution time: 153 ms = 0.153 s
CPU execution time: 56636 ms = 56.636 s
```

Під CPU execution time мається на увазі час роботи послідовного алгоритму, але навіть порівняно з часом паралельного алгоритму на результати CUDA вражають.

Збільшимо розмірність.

```
[4000][4000]
CUDA execution time: 1907 ms = 1.907 s
CPU execution time: 545581 ms = 545.581 s
```

Різниця насправді колосальна. Те, що синхронно рахувалось CPU 9 хвилин, на CUDA ядрах виконалось за менш ніж 2 секунди.

Висновки: У результаті виконання лабораторної роботи було реалізовано алгоритм множення матриць використовуючи програмно-апаратну архітектуру паралельних обчислень CUDA.