Лабораторна робота №2

**Тема**: CUDA: Ядро та компіляція програми

**Мета**: Одержати практичні навички написання та компіляції CUDA-програм.

1. Теоретичні відомості

**Програмна модель**

Стандартною мовою програмування для написання додатків на CUDA є мова C. Програму на CUDA можна логічно розділити на дві частини, перша частина (керуюча) виконується на CPU, друга частина (обчислювальна) виконується на GPU. CPU в термінології CUDA називається host, GPU – device. Частина коду, яка повинна виконуватися на GPU, називається ядром (kernel), вона описується у вигляді функції.

Розширення мови С, що входять у CUDA складаються з:

* специфікаторів функцій, які описують, де буде виконуватися функція і звідки вона може бути викликана;
* специфікаторів змінних, що задають тип пам'яті, який використовується для даної змінної;
* директив, що використовуються для запуску ядра і задає як дані, так і ієрархію потоків;
* вбудованих змінних, що містять інформацію про поточний потік;
* runtime, що містить в собі додаткові типи даних.

Програми зберігаються у файлі з розширенням *.cu*.

Всі програмні коди компілюються використовуючи CUDA API. Спочатку компілюється код, що призначений виключно для центрального процесора, а інший код, призначений для графічного процесора, компілюється в проміжну мову PTX (щось схоже до байт-коду в Java) для виявлення можливих помилок. Після чого, компілюється в «зрозумілу» для CPU/GPU мову.

**Специфікатори (модифікатори)**

Перед функціями в *.cu* файлі можуть стояти наступні модифікатори:

* *\_\_device\_\_* – означає, що функція виконується тільки на відеокарті. З програми, що виконується на звичайному процесорі (хості), її викликати не можна;
* *\_\_global\_\_* – функція – початок вашого обчислювального ядра. Виконується на відеокарті, але запускається з хоста;
* *\_\_host\_\_* – виконується і запускається тільки з хоста (тобто звичайна функція C).

При цьому модифікатори *\_\_host\_\_* і *\_\_device\_\_* можуть бути використані разом (це означає, що відповідна функція може виконуватися як на GPU, так і на CPU – відповідний код для обох платформ буде автоматично згенерований компілятором).

Модифікатори *\_\_global\_\_* і *\_\_host\_\_* не можуть бути використані разом. Детальніше можна почитати [тут](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-extensions).

Функції, що виконуються на GPU, мають певні обмеження: вони не можуть містити рекурсії, не можуть мати змінне число вхідних аргументів, не можуть містити статичні змінні, а також не можна отримати [адресу](https://uk.wikipedia.org/wiki/%D0%92%D0%BA%D0%B0%D0%B7%D1%96%D0%B2%D0%BD%D0%B8%D0%BA) такої функції.

**Нові змінні та типи даних**

В розширення мови C додані наступні спеціальні змінні:

* *gridDim* – розмір сітки (grid) (тип dim3);
* *blockDim* – розмір блоку (тип dim3);
* *blockIdx* – індекс поточного блоку в grid (тип uint3);
* *threadIdx* – індекс поточного потоку в блоці (тип uint3);
* *warpSize* – розмір warp (тип int).

Також додаються 1 / 2 / 3 / 4 – мірні вектори з базових типів: char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4 і так далі.

**Директива виклику ядра**

Для запуску ядра на GPU використовується наступна конструкція:

kernelName <<<Dg,Db,Ns,S>>> (args)

Тут *kernelName* – ім'я (адреса) відповідної *\_\_global\_\_* функції,

*Dg* – змінна (або значення) типу dim3, що задає розмірність grid (в блоках - gridDim),

*Db* – змінна (або значення) типу dim3, що задає розмірність блоку (в потоках - blockDim),

*Ns* – змінна (або значення) типу size\_t, що задає додатковий обсяг спільної пам'яті, яка повинна бути динамічно виділена (до вже статично виділеної shared-пам'яті; параметр не є обов'язковим),

*S* – змінна (або значення) типу cudaStream\_t, що задає потік (потік CUDA), в якому має викликатися ядро, за замовчуванням використовується потік 0. Через *args* позначено аргументи виклику функції *kernelName*.

Множину потоків у блоці та блоків у сітці (grid) можна задавати у вигляді 1 / 2 / 3 - мірних векторів. Розміри сітки та максимально можлива кількість потоків напряму залежать від відеокарти, яка використовуєтья.

Також в мову С додана функція *\_\_syncthreads,* яка здійснює синхронізацію всіх потоків блоку. Управління з неї буде повернуто тільки тоді, коли всі потоки даного блоку викличуть цю функцію. Тобто, коли весь код, що йде перед цим викликом, вже виконано. Ця функція дуже зручна для організації безконфліктної роботи зі [спільною пам'яттю](https://devblogs.nvidia.com/using-shared-memory-cuda-cc/).

**Як потік знає над якими даними йому працювати?**

Згадаємо як організовані блоки та потоки в CUDA.

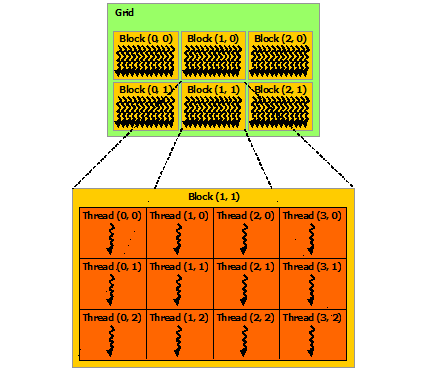


Рис. 1. Організація блоків в CUDA

Припустимо, що потрібно зробити деякі операції над зображенням (зберігається у змінній fox) розміром *400х400*пікселів. Зображення можна розділити на ділянки (блоки) по *10х10* пікселів і для опрацювання кожної ділянки запустити окремий потік.

dim3 threads(10, 10) // розмір ділянки зображення

dim3 blocks(40, 40) // к-сть блоків для обробки зображення; 10\*40 = 400

process\_image\_kernel<<<blocks, threads>>> (fox)

Оскільки параметри (див. Нові змінні та типи даних), що передаються у ядро однакові для всіх потоків, то кожен блок повинен сам визначити свою позицію на зображенні.

const int positionX = blockDim.x \* blockIdx.x + threadIdx.x;

const int positionY = blockDim.y \* blockIdx.y + threadIdx.y;

Обчислені змінні є відповідними координатами x та y вхідного зображення.

**Приклад програми для додавання двох чисел**

#include <iostream>

\_\_global\_\_ void Addition(int \*a, int \*b, int \*c) {

\*c = \*a + \*b;

}

int main() {

int a, b, c;

int \*dev\_a, \*dev\_b, \*dev\_c;

int size = sizeof(int);

cudaMalloc((void\*\*)&dev\_a, size);

cudaMalloc((void\*\*)&dev\_b, size);

cudaMalloc((void\*\*)&dev\_c, size);

a = 100, b = 400;

cudaMemcpy(dev\_a, &a, sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, &b, sizeof(int), cudaMemcpyHostToDevice);

Addition<<<1, 1>>>(dev\_a, dev\_b, dev\_c);

cudaMemcpy(&c, dev\_c, size, cudaMemcpyDeviceToHost);

cudaFree(&dev\_a);

cudaFree(&dev\_b);

cudaFree(&dev\_c);

printf("%d\n", c);

return 0;

}

Скомпілювати програму можна наступною командою:

nvcc file.cu

Як видно із лістингу коду вище, написання CUDA програми складається з кількох кроків:

1. Написання функції ядра
2. Оголошення необхідних змінних та [виділення пам’яті](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1g37d37965bfb4803b6d4e59ff26856356) для них на GPU
3. [Копіювання](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1gc263dbe6574220cc776b45438fc351e8) даних з пам’яті хоста в пам’ять GPU
4. Запуск ядра
5. Копіювання даних з пам’яті GPU в пам’ять пристрою.

2. Хід роботи.

1. Відкрити <https://colab.research.google.com/drive/1xTHL3CnpErVdNY9AKZmj29sXHLw3KlXD?usp=sharing> та виконати інструкції.
2. Написати CUDA програму відповідно до варіанту.

**Варіанти виконання:**

1. Написати ядро для обчислення суми двох векторів. Вектори розміром 2048 елементів.
2. Написати ядро для обчислення суми трьох векторів. Вектори розміром 4096 елементів.

3. Структура звіту лабораторної роботи.

* Титульна сторінка.
* Тема та мета роботи.
* Короткі теоретичні відомості.
* Лістинг коду та екрани із результатами виконання програми.
* Висновки.

4. Контрольні запитання

1. Що таке CUDA?

2. Основні системні вимоги для інсталяції СUDA?

3. Переваги GPU над CPU?

4. Що таке grid, block, thread, warp?

Список літератури:

1. CUDA Runtime API Docs - <https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>
2. CUDA Programming Guide - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels>