Лабораторна робота №4

**Тема**: CUDA: Атомарні операції та синхронізація потоків.

**Мета**: Одержати практичні навички використання атомарних операцій та механізмів синхронізації CUDA потоків.

1. Теоретичні відомості

**Організація потоків**

Паралельне виконання програми досягається за рахунок архітектурних особливостей графічних процесорів. CUDA використовує велику кількість окремих потоків для розрахунків. Всі потоки згруповані дотримуючись наступної ієрархії: grid, block, thread.

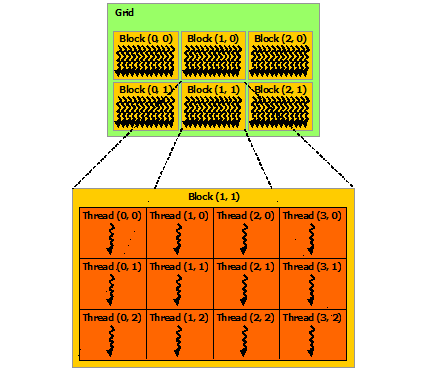


Рис. 1. Організація блоків в CUDA

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.

**Атомарні операції**

Атомарні операції — це операції, які виконуються без втручання будь-яких інших потоків. Припустимо, що у нас є два потоки з іменами A і B. Тепер припустимо, що кожен потік хоче збільшити значення змінної Х на одиницю. Для прикладу, на даний момент значення Х в пам’яті дорівнює 5. Якщо A і B хочуть збільшити значення Х одночасно, кожен потік повинен буде спочатку прочитати значення. Залежно від того, коли відбуваються зчитування, можливо, що і A, і B прочитають значення 5. Після додавання значення 1 і A, і B захочуть записати 6 у комірку пам’яті, що є неправильним! Значення 5 слід було збільшити двічі (один раз для кожного потоку), але замість цього значення було збільшено лише один раз! Це називається race-condition і може статися в будь-якій багатопотоковій програмі, якщо програміст не буде обережним.

Цієї ситуації можна легко уникнути в CUDA. Атомарна операція здатна зчитувати, змінювати та записувати значення назад у пам’ять без втручання будь-яких інших потоків. Атомарні операції в CUDA зазвичай працюють як для спільної, так і для глобальної пам’яті. Атомарні операції в спільній пам'яті зазвичай використовуються для запобігання конкуренції між різними потоками в одному блоці потоку. Атомарні операції в глобальній пам’яті використовуються для запобігання конкуренції між двома різними потоками, незалежно від того, у якому блоці потоку вони знаходяться. Зауважте, що спільна пам’ять зазвичай набагато швидша, ніж глобальна пам’ять.

int atomicAdd(int\* адреса, int val);

Цю функцію *atomicAdd* можна викликати в ядрі. Коли потік виконує цю операцію, адреса пам’яті зчитується, до неї додається значення «val», а результат записується назад у пам’ять. Початкове значення пам’яті за «адресою» розташування повертається до потоку. Багатьом алгоритмам, які вимагають атомарних операцій, не потрібно буде використовувати вихідне значення в місці пам’яті. Повний список атомарних функцій - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>

**Синхронізація потоків**

CUDA забезпечує бар’єр синхронізації для всіх потоків у блоці через метод *\_\_syncthreads()*.

Синхронізація потоків можлива лише між усіма потоками в блоці, але не між усіма потоками, що працюють у сітці. Не дозволяючи синхронізувати потоки між блоками, CUDA дозволяє виконувати декілька блоків на інших потокових мультипроцесорах (SM) у будь-якому порядку. Черга блоків може бути розподілена на будь-який SM без необхідності чекати завершення блоків з іншого SM. Це дозволяє додаткам із підтримкою CUDA масштабуватися між платформами, які мають у своєму розпорядженні більше SM, виконуючи більше блоків одночасно, ніж інші платформи з меншою кількістю SM.

Синхронізація потоків відповідає суворим правилам синхронізації. Усі потоки в блоці повинні досягти точки синхронізації, якщо це потрібно.

**Розподілення потоків**

Коли ядро викликається, середовище виконання CUDA розподілить блоки між SM на пристрої. Максимум 8 блоків (незалежно від платформи) буде призначено кожному SM, якщо є достатньо ресурсів (реєстрів, спільної пам’яті та потоків) для виконання всіх блоків. У випадку, коли на SM недостатньо ресурсів, середовище виконання CUDA автоматично призначатиме менше блоків на SM, доки використання ресурсу не стане нижче максимального на SM. Коли блок призначається SM, він далі ділиться на групи з 32 потоків, які називаються *warp*. Розмір warp залежить від платформи.

2. Хід роботи.

1. Обрати будь-яку з наведених бібліотек CUDA для роботи:

* Мова C, стандартна бібліотека СUDA
* Мова Python, бібліотека [Numba](https://numba.pydata.org/)
* Мова Python/C, бібліотека [Pycuda](https://documen.tician.de/pycuda/tutorial.html)

1. Написати CUDA програму відповідно до варіанту.

**Варіанти виконання:**

1. Написати функцію-ядро Counter. Кожен потік повинен збільшувати значення змінної counter на одиницю. На виході має бути число, яке дорівнює к-сті потоків з якими було викликано ядро. Для прикладу, якщо викликати ядро із 1024 потоками – в результаті змінна counter повинна дорівнювати 1024. Якщо викликати ядро із двома блоками по 512 потоків – кінцевий результат також повинен дорівнювати 1024.
2. Написати ядро для обчислення суми елементів масиву. Розмір масиву 256 елементів. Значення масиву відповідають індексам елементів ([1, 2, 3, …, 256]).
3. Написати ядро для множення двох матриць. Розмір матриць 4 на 4.

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 grid, block, thread, warp?

2. Директива виклику ядра.

3. Типи пам’яті.

4. Ієрархія потоків.

5. Атомарні операції.

6. Cинхронізація потоків.

Література:

1. <https://icl.utk.edu/~mgates3/docs/cuda.html>