Лабораторна робота №3

**Тема**: CUDA: Типи пам’яті, атомарні операції та синхронізація потоків.

**Мета**: Одержати практичні навички використання різних типів пам’яті 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.

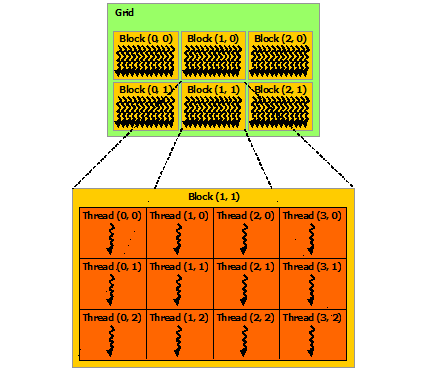


Рис. 1. Організація блоків в CUDA

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.

**Типи пам’яті**

Існує кілька різних типів пам’яті, до яких програма CUDA має доступ. Для кожного окремого типу пам’яті існують компроміси, які необхідно враховувати при розробці програми.

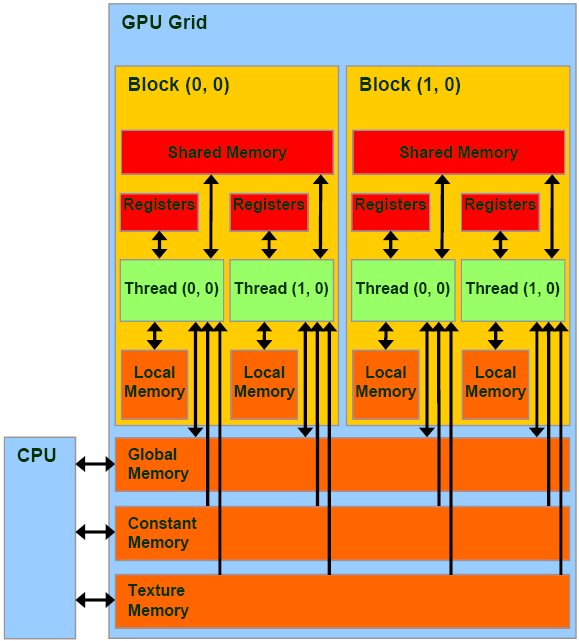


Рис. 2. Модель пам’яті в CUDA

***Register***

Скалярні змінні, які оголошені в області видимості функції ядра за замовчуванням зберігаються в пам’яті регістрів. Доступ до регістрової пам'яті дуже швидкий, але кількість регістрів, доступних на блок, обмежена.

Масиви, оголошені у функції ядра, також зберігаються в регістрах, але лише коли доступ до елементів масиву виконується за допомогою постійних індексів (це означає, що індекс, який використовується для доступу до елемента в масиві, не є змінною, а тому індекс можна визначити під час компіляції).

Змінні регістру є приватними для потоку. Потоки в одному блоці отримають приватні версії кожної змінної регістру. Такі змінні існують лише до тих пір, поки існує потік. Як тільки потік завершить виконання, змінна з регістру не може бути доступною.

Змінні, оголошені в регістровій пам'яті, можна як читати, так і записувати всередині ядра. Читання та запис у таку пам'ять не потрібно синхронізувати.

***Local***

Будь-яка змінна, яка не може поміститися в регістри, буде розміщена в локальній пам’яті. Локальна пам'ять має таку саму затримку доступу, що й глобальна пам'ять (тобто повільна).

Як і регістри, локальна пам'ять є приватною для потоку. Кожен потік повинен ініціалізувати вміст змінної, що зберігається в локальній пам'яті, перш ніж її використовувати. Не можна покладатися на інший потік (навіть у тому самому блоці) для ініціалізації локальної пам’яті, оскільки даний тип пам’яті також є приватним для потоку. Після завершення виконання потоку локальна змінна стає недоступною.

CUDA не надає користувачу спеціального модифікатора, щоб явно оголосити змінну в локальній пам’яті, але компілятор автоматично помістить оголошення змінних у цю пам’ять за таких умов:

* Масиви, доступ до яких здійснюється за допомогою індексів, що зберігаються у змінних. Тобто компілятор не може визначити індекси під час компіляції.
* Великі структури або масиви, які займатимуть забагато місця в регістрі.
* Будь-яка оголошена змінна, що перевищує кількість регістрів для даного ядра.

Змінні, оголошені в локальній пам'яті, можна як читати, так і записувати всередині ядра. Читання і запис у таку пам'ять також не потрібно синхронізувати.

***Shared***

Змінні з модифікатором *\_\_shared\_\_* зберігаються в спільній пам’яті. Доступ до спільної пам’яті дуже швидкий (приблизно в 100 разів швидше, ніж до глобальної пам’яті), хоча кожен потоковий мультипроцесор має обмежену кількість адресного простору спільної пам’яті.

Спільна пам'ять має бути оголошена в межах функції ядра, але існує в межах блоку (на відміну від регістру або локальної пам'яті, яка існує в межах потоку). Після завершення виконання блоку доступ до спільної пам’яті, визначеної в ядрі, неможливий.

Спільну пам'ять можна як читати, так і записувати в ядрі. Модифікація спільної пам’яті повинна бути синхронізована, якщо ви не гарантуєте, що кожен потік матиме доступ лише до пам’яті, з якої інші потоки в блоці не будуть читати або записувати. Синхронізація потоків в блоці досягається за допомогою функції *\_\_syncthreads()* у функції ядра.

***Global***

Змінні з модифікатором *\_\_device\_\_* і оголошені в глобальній області (поза межами функції ядра), зберігаються в глобальній пам’яті. Затримка доступу до глобальної пам’яті дуже висока (приблизно в 100 разів повільніше, ніж до спільної пам’яті), але глобальної пам’яті набагато більше, ніж спільної (цей розмір обмежений к-стю пам’яті самої відеокарти). На відміну від регістрової, локальної та спільної пам’яті, глобальну пам’ять можна зчитувати та записувати за допомогою C-функції *cudaMemcpy*.

Глобальна пам'ять існує в межах програми та доступна для всіх потоків усіх ядер. *Потрібно бути обережним під час читання та запису в глобальну пам’ять, оскільки виконання потоку не може бути синхронізовано між різними блоками*. Єдиний спосіб забезпечити синхронізацію доступу до глобальної пам’яті – це розділяти задачу на різні ядра та синхронізувати на хості між їх викликами.

Глобальна пам’ять оголошується в основному процесі за допомогою *cudaMalloc* і звільняється в основному процесі за допомогою *cudaFree*.

***Constant***

Змінні з модифікатором *\_\_constant\_\_* оголошуються в постійній пам’яті. Як і глобальні змінні, постійні змінні повинні бути оголошені в глобальній області (поза межами будь-якої функції ядра). Константні змінні спільно використовують глобальну пам’ять (пам’ять пристрою), але, на відміну від глобальної пам’яті, існує лише обмежена кількість постійної пам’яті, яку можна оголосити (64 КБ для всіх обчислювальних можливостей).

Затримка доступу до постійної пам’яті значно менша, ніж до глобальної пам’яті, оскільки постійна пам’ять кешується, але на відміну від глобальної пам’яті, постійна пам’ять не може бути записана з ядра. Ця пам’ять доступна тільки для читання всередині ядра, що дає можливість використовувати кешування.

Постійна пам’ять може бути записана головним процесом за допомогою функції *cudaMemcpyToSymbol* і зчитана з неї за допомогою функції *cudaMemcpyFromSymbol*.

Як і глобальна пам'ять, постійна пам'ять існує в межах програми. До неї можуть отримати доступ усі потоки всіх ядер, і значення не змінюватиметься під час викликів ядра, якщо воно явно не змінено головним процесом.

***Коротке резюме по типах пам’яті***

Регістри є найшвидшим типом пам'яті. Подібно до регістрів на ЦП, вони автоматично розподіляються для основних змінних у обчисленнях.

Однак вони видимі лише для кожного потоку, тому вони не підходять для зв’язку на рівні пристрою.

Локальна пам'ять також є пам'яттю, яка є видимою лише для кожного потоку та використовується, коли неможливо використовувати швидші регістри.

Спільна пам'ять дещо повільніша, ніж регістри та видима для всіх потоків у блоці. Однак цього недостатньо, оскільки ми розглядаємо потенційно величезну кількість вхідних даних, яка може бути набагато більшою за максимальний розмір блоку, тобто 1024 потоки.

Глобальна пам'ять, з іншого боку, видима для кожного потоку в пристрої, але також значно повільніша, оскільки вона не розташований безпосередньо на SM.

Постійна пам’ять описує обмежений обсяг глобальної пам’яті лише для читання з особливо швидкою кеш-пам’яттю для рівномірного читання, а текстурна пам’ять має додаткові можливості, які відображають текстури та зображення в звичайних графічних API.

**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 1024 елементи. Значення масиву відповідають індексам елементів ([1, 2, 3, …]).
3. Написати ядро для множення двох матриць. Розмір матриць 8 на 8.

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://developer.nvidia.com/blog/using-shared-memory-cuda-cc/>
2. <https://icl.utk.edu/~mgates3/docs/cuda.html>