УДК 004.3

С. Д. Погорілий, Д. Ю. Вітель, О. А. Верещинський
Київський національний університет імені Тараса Шевченка
Радіофізичний факультет
просп. Академіка Глушкова, 4-г, 01033Київ, Україна

Новітні архітектури відеоадаптерів. Технологія GPGPU. Частина 2

Детально розглянуто основні принципи роботи зі спільною та розподіленою пам'яттю в технології NVidia CUDA. Описано шаблони взаємодії потоків і проблеми глобальної синхронізації. Проведено порівняльний аналіз основних технологій, що використовуються в підході GPGPU — Nvidia CUDA, OpenCL, Direct Compute.

Ключові слова: *General-purpose graphics processing units (GPGPU), NVidia Compute Unified Device Architecture (CUDA), DirectCompute, OpenCL, Single Instraction Multile Data (SIMD), Single Instraction Multiple Threads (SIMT), шейдери, Message Passing Interface (MPI), ієрархічна організація потоків, модель синхронізації, модель пам'яті, гетерогенні розподілені системи, динамічний паралелізм, динамічне виділення пам'яті, архітектури відеоадаптерів (Tesla, Fermi, Kepler).*

Спільна пам'ять

Спільна пам'ять організована у вигляді банків (banks) так, що послідовні 32-бітні слова належать послідовним банкам пам'яті. Одночасний доступ до елементів пам'яті, що належать одному банку, генерує банк-конфлікт — збільшення кількості тактів для зчитування даних. Останніх слід уникати, належним чином організовуючи доступ до ресурсів у програмі.

Для Tesla-архітектури їхня кількість рівна 16, і пропускна спроможність кожного з них рівна 32 біти за 2 такти. Запит із warp-групи до спільної пам'яті розбивається на 2 незалежних запити (одна для кожної half-warp-групи). Тому для двох потоків з різних half-warp-груп банк-конфлікти відсутні.

Поширеними шаблонами доступу до спільної пам'яті є наступні.

1. Послідовний доступ (Strided access) — доступ до елементів масиву в спільній пам'яті, при якому кожен потік у warp-групі читає 1 елемент з індексом, що рівний індексу потоку.

2. Масовий доступ (Broadcast access) — декілька потоків формують запит до одного елемента у спільній пам'яті.

Для Tesla-архітектури банк-конфлікти виникають у наступних випадках:

1) за наявності 32-бітового послідовний доступу до масиву з half-warp-групи за послідовними індексами, коли перший індекс не кратний 164;

2) за наявності 8-, 16-бітового послідовного доступу;

3) за наявності послідовного доступу до елементів з розміром, більшим за 32 біти.

Fermi-архітектура має 32 банки спільної пам'яті з пропускною спроможністю 32 біти за 2 такти. При цьому банк-конфлікти відсутні при доступі до слова в межах одного 32 біт-сегмента (що належить 1 банку). У цьому випадку зчитаний банк транслюється усім потокам (broadcast access).

З появою архітектури Kepler з'явилася можливість обирати режим роботи банків спільної пам'яті. Як і для архітектури Fermi, кількість останніх залишилася рівною 32, проте пропускна спроможність кожного збільшилася до 64 біт за 2 такти. Існує два режими роботи: 64-бітний режим — послідовні слова розміром у 64 біти асоціюються з відповідними банками спільної пам'яті (якщо декілька потоків створюють запит на данні в межах одного 64-бітного сегмента, то банк-конфлікти не створюються); 32-бітний режим — послідовні слова розміром у 32 біти асоціюються з відповідними банками спільної пам'яті. Рис. 1 демонструє деякі шаблони доступу до спільної пам'яті.

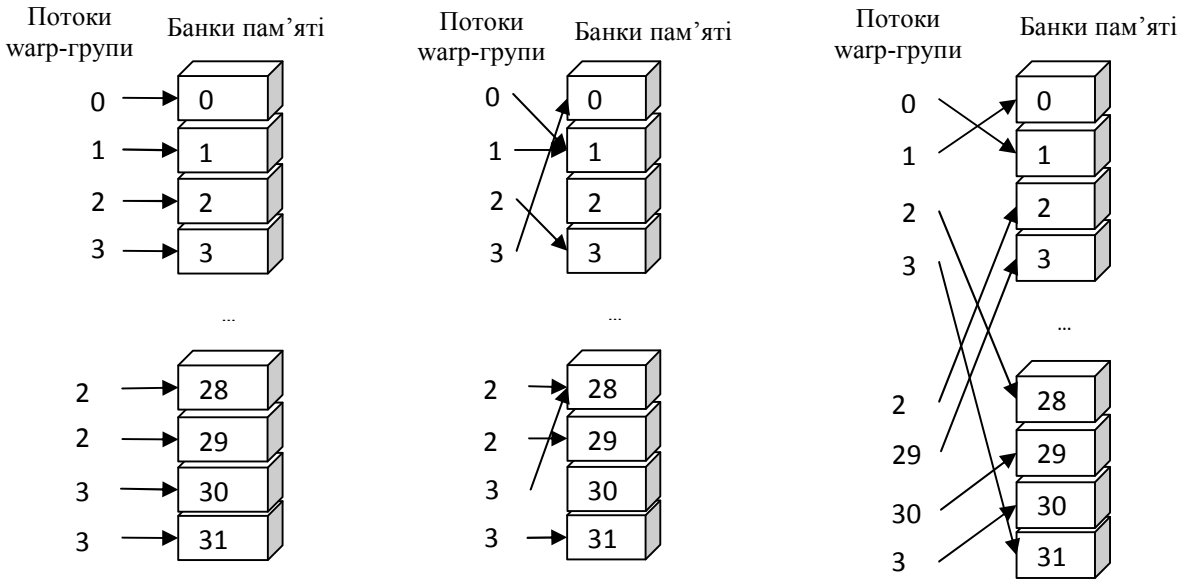

Рис. 1. Варіанти доступу до спільної пам'яті

Шаблони взаємодії потоків у NVidia CUDA

Для максимального використання апаратних можливостей системи ряд операцій у CUDA-програмі виконується асинхронно та паралельно. До них належать:

— код на CPU та GPU (асинхронний запуск ядра);

— код на CPU та копіювання даних у пам'ять GPU за допомогою операцій з суфіксом `Async` або розміром, що менший за 64 КБ;

— код на CPU та копіювання даних між двома адресами в пам'яті GPU;

— код на CPU та функції встановлення області пам'яті на відеоадаптері (`memSet`).

Існує можливість переведення виконання даних операцій у синхронний режим за допомогою змінної оточення `CUDA_LAUNCH_BLOCKING`.

Також існують наступні можливості:

— виконання ядра паралельно із копіюванням даних з CPU RAM до глобальної пам'яті GPU (можливість з'явилась у Tesla CC 1.1). Вимагає використання page-locked пам'яті [1–4];

— паралельне виконання ядер на одному відеоадаптері (можливість з'явилася з появою Fermi-архітектури);

— паралельне виконання копіювання з page-locked CPU RAM-пам'яті до GPU-пам'яті та копіювання із GPU-пам'яті до page-locked CPU RAM (можливість з'явилась з появою Fermi-архітектури).

Синхронізація між кодом CPU та GPU-операціями може виконуватись явно або неявно. Для явної синхронізації обчислень використовується поняття CUDA-потоку (*stream*). CUDA-потік — набір інструкцій, що виконуватимуться послідовно в порядку додавання інструкцій до нього. Проте, різні потоки можуть виконуватися паралельно та не упорядковано. CUDA-потік створюється за допомогою API-функції `cudaStreamCreate` та знищується за допомогою `cudaStreamDestroy` (призводить до очікування всіх операцій потоку). Існує ряд API-функцій, що дозволяють провести очікування GPU: `cudaDeviceSynchronize`, `cudaStreamWaitEvent`. Наступний приклад демонструє використання потоків:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size,
size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i *
size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Інший варіант взаємодії CPU та GPU — використання функцій оберненого виклику (*callbacks*). Такі функції можуть бути зареєстровані в коді CPU за допомогою методу `cudaStreamAddCallback`.

Також CUDA визначає поняття події (*event*) — умови, що стає істинною, коли всі команди в потоці до неї вже виконались.

Взаємодія потоків на GPU може відбуватися за допомогою наступних механізмів.

1. Синхронізація потоків у блоці (`_syncthreads`, `_syncthreads_count` тощо).

```
void CUDART_CB MyCallback(void *data){
    printf("Inside callback %d\n", (int)data);
}
...
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size, cudaMemcpy-
HostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devP-
trIn[i], size);
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size, cudaMemcpy-
DeviceToHost, stream[i]);
    cudaStreamAddCallback(stream[i], MyCallback, (void*)i, 0);
}
```

2. Атомарні функції (`atomicXor, atomicOr, atomicAnd` тощо).

3. Функції голосування потоків у warp-групі (`__all, __any, __ballot`).

4. Функції перемішування потоків у warp-групі (`__shfl, __shfl_up, __shfl_down, __shfl_xor` з'явились у Kepler-архітектурі) — функції, що дозволяють обмінюватися даними між потоками в одній warp-групі.

Проблема міжблокової синхронізації та можливі шляхи її вирішення

Зрозуміло, що при розв'язуванні складних задач «блочна» синхронізація не є достатньою, може виникнути ситуація, коли блоки виконують різні операції паралельно, і один блок для подальшого виконання повинен дочекатися результатів роботи іншого блока. В такому випадку виникає необхідність штучної міжблокової синхронізації. Для забезпечення такого механізму можна скористатися двома методами.

1. Введення змінних-флагів у глобальній пам'яті та циклічних перевірок для синхронізації. У даному випадку можуть виникнути проблеми з порожніми циклами, оскільки оптимізатор може просто видалити непотрібний на його думку код. Така синхронізація безумовно буде працювати, щоправда звертання до глобальної пам'яті є однією з найбільш латентних операцій.

2. Розбиття складного ядра на простіші — декомпозиція на рівні ядер. Як було зазначено раніше, код ядра завантажується на відеоадаптер лише один раз, отже не буде постійних накладних витрат, пов'язаних з викликом. Щоправда, такий підхід ще більше звужує коло задач, які можна покласти на архітектуру CUDA.

Динамічний паралелізм та виділення пам'яті

З появою Kepler-архітектури з'явилася технологія динамічного паралелізму. Така технологія надає можливість GPU виконувати розрахунки з більшою продуктивністю шляхом виклику одного ядра з іншого. В такій взаємодії CPU не використовується, і тому нема втрат часу на взаємодію GPU та CPU (рис. 2).

Динамічне виділення пам'яті полягає у використанні стандартних CPU-подібних функцій `malloc` та `free` в коді на GPU. Час життя виділеної таким чи-

ном пам'яті дорівнює часу життя CUDA-контексту, і тому може використовуватись у послідовному визові серії ядер.

Разом два даних підходи зменшують навантаження на CPU, звільняючи його для іншої роботи.

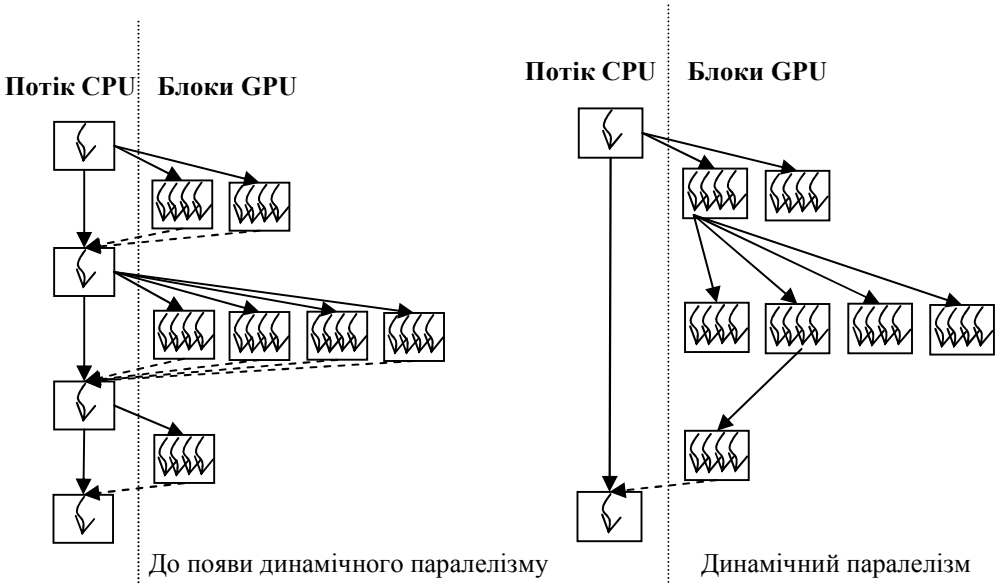

Рис. 2. Виконання багатоетапного мультипотокового завдання

Взаємодія потоків у системах з розподіленою пам'яттю. Гетерогенні кластерні системи на основі відеопадптерів

Основний спосіб взаємодії потоків у розподілених системах описується шаблоном передачі повідомлень (Message Passing), ідея якого полягає в тому, що для доступу до даних деякого потоку з групи реалізується певний інтерфейс чи протокол обміну. Фактично, використовуючи цей інтерфейс, один потік може запросити відповідні данні чи дію у іншого потоку. При цьому інший потік має реалізувати певну поведінку для обслуговування запитів до нього. В такій схемі окремо можна виділити синхронну та асинхронну взаємодії. Виділення в системі головного потоку (або арбітра), або певної їхньої сукупності, дозволяє створити синхронізацію, що позбавлена тупикових ситуацій.

Частковим випадком взаємодії Message Passing є клієнт-серверна взаємодія, яка реалізується в таких технологіях як MPI [5, 6], Windows Communication Foundation (WCF), Web Services, Win Services. Основою реалізації даного шаблону є механізм рознімів (sockets). Цей вид взаємодії інтегрований в деякі мови програмування (наприклад F# (MailboxProcessor) [7], Scala (Actors), ErLang). Реалізація шаблону Message Passing фактично є реалізацією скінченного автомату.

Розглянемо шаблон Message Passing детальніше. Можна виділити наступні сутності даного шаблону: клієнт-потік — мережевий потік, що відсилає повідомлення; сервер-потік — потік, що приймає повідомлення і виконує певну дію; повідомлення — певні данні, що сформовані за певним (наприклад, мережевим) протоколом. Уникнути тупикових ситуацій у даній схемі можливо за рахунок на-

явності того факту, що певний ресурс належить лише сервер-потоку, що, в свою чергу, може володіти чергою повідомлень на обробку. В цьому випадку багато клієнтів можуть відправити паралельно велику кількість запитів на обробку даних, а сервер-потік обробить ці запити послідовно, не порушуючи стан даних.

Далі розглянемо основні принципи побудови застосування для гетерогенних систем на основі відеоадаптерів. Такі системи можуть відтворювати наступні рівні паралелізму:

— рівень процесорних ядер одного вузла мережі;

— рівень вузлів мережі;

— рівень відеоадаптера.

Слід зазначити, що у багатьох випадках не доцільно використовувати усі ці рівні разом. Так, наприклад, при створенні застосування, що виконується на наборі відеоадаптерів у мережі (2 і 3 рівень разом), використання додаткових ядер центрального процесора в задачі разом із ядрами відеоадаптера не є доцільним, оскільки останні є повільнішими, і така взаємодія потребує додаткових засобів синхронізації. Центральний процесор в даному випадку може виступати як арбітр і посередник між мережею і відеоадаптером.

CUDA 5.0 дозволяє відмовитися від прямої участі процесора навіть у цій взаємодії, що наштовхує на думку про створення спеціалізованих мереж з комп'ютерів, у яких взагалі відсутні процесори. CUDA 5.0 презентує підхід *віддаленого прямого доступу* до пам'яті (Remote Direct Memory Access, RDMA), при якому в роботі такої розподіленої системи приймають участь лише відеоадаптери і спеціалізовані мережеві карти (рис. 3).

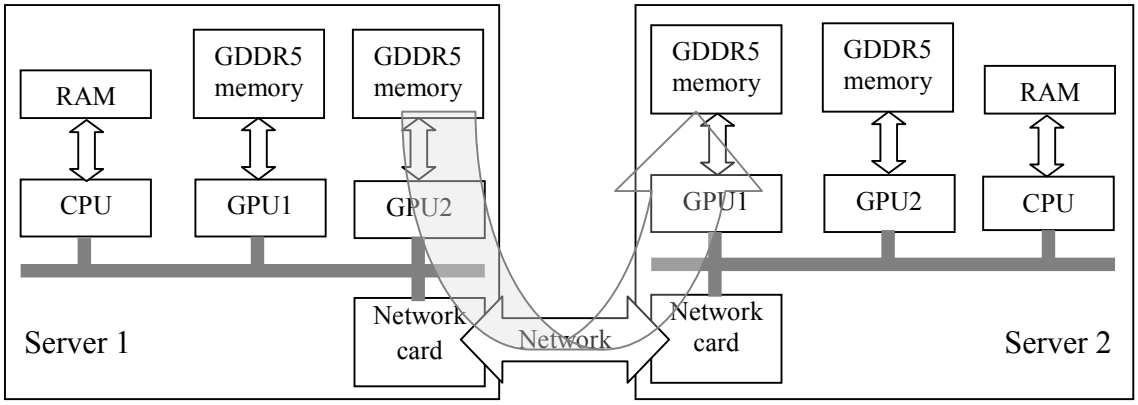

Рис. 3. Мережева взаємодія адаптерів у CUDA 5.0

Основні принципи створення застосування для гетерогенної системи з відеоадаптерами полягають у наступному.

1. Розбити ділянки даних, що обробляє алгоритм, на частини, які будуть належати відповідному вузлу мережі (тобто створити відображення: об'єкт пам'яті → вузол мережі).

2. Кожну таку ділянку слід розбити на частини, що відповідатимуть одному потоку виконання (робочому елементу) на відеоадаптері (відображення: ділянка пам'яті → потік відеоадаптера).

3. При використанні технології CUDA-версії, нижче за 5.0, існуватиме 2 копії об'єктів у програмі. Одна копія зберігається в оперативній пам'яті, а інша в пам'яті GPU. Відповідно, за необхідності передати данні з одного вузла мережі в інший, необхідно скопіювати актуальні данні з відеоадаптера в RAM, передати мережею в RAM віддаленого вузла і скопіювати дані з RAM віддаленого вузла в пам'ять його відеоадаптера.

4. Для синхронізації на рівні вузла мережі використовується CPU як арбітр (зазвичай виконання запускається на GPU асинхронно, а CPU в необхідний момент може почати очікування завершення виконання інструкцій на ньому). Для синхронізації на рівні вузлів мережі слід використовувати певний мережевий протокол взаємодії. При використанні MPI цього легко досягти використовуючи відповідну API-функцію.

OpenCL

OpenCL — це модель виконання, що підтримує *програмні моделі* паралелізму за даними та завданнями. Також підтримується змішана (гібридна) програмна модель.

Програмна модель паралелізму за даними визначає певне обчислення в термінах певної послідовності інструкцій, що застосовуються до множини об'єктів пам'яті. Індексний простір, визначений OpenCL, дозволяє створити відображення певного робочого елемента (PE) (потоку) пристрою на певний елемент чи область пам'яті з яким він буде працювати. Проте у строго визначеній моделі паралелізму за даними таке відображення є відображенням один до одного. OpenCL-специфікація не накладає такого строгого обмеження і дозволяє визначати співвідношення типу багато до багатьох (проте в цьому випадку необхідно дбати про синхронізацію доступу до об'єктів).

Також OpenCL підтримує ієрархічну модель паралелізму за даними, що проявляється у наступному: розподіл між робочими групами елементів виконання може або жорстко керуватися програмно, або визначатися імплементацією OpenCL-специфікації.

У моделі паралелізму за завданнями кожен примірник ядра, що виконується в межах OpenCL-пристрою існує незалежно від будь-якого індексного простору. Це еквівалентно його виконанню на обчислювальній одиниці в межах робочої групи, яка має лише один елемент виконання. Використання паралелізму можливе в цій моделі за рахунок:

— наявності векторних типів даних OpenCL-пристрою;

— виконання декількох ядер-функцій одночасно;

— виконання ядер, що специфічні для даного типу пристроїв, які мають ортогональну модель до OpenCL.

Модель синхронізації OpenCL. Існує два типи синхронізації в OpenCL:

1) рівня робочої групи;

2) рівня команд черги в одному контексті виконання.

Синхронізація робочих елементів на рівні однієї робочої групи здійснюється за допомогою операції типу «бар'єр». Механізм синхронізації робочих груп відсутній.

На рівні черги команд існують наступні точки синхронізації:

1) *бар'єр* — команда, яка гарантує, що всі раніше поставлені в чергу команди закінчили своє виконання і стан об'єктів, що виник після цього, є видимим для усіх інших наступних команд після неї. Синхронізація можлива лише в межах однієї командної черги;

2) *очікування події*. Всі функції API OpenCL, що ставлять в чергу на виконання певну команду, повертають подію, яка ідентифікує цю команду, а також об'єкти пам'яті з якими остання працює. Наступна команда очікування цієї події гарантує, що всі оновлення зазначених об'єктів пам'яті будуть доступні у наступних командах черги.

Модель пам'яті OpenCL. Робочі елементи виконання ядра мають доступ до чотирьох різних областей пам'яті.

1. Глобальна пам'ять. Ця області пам'яті дозволяє читання/запис для всіх PE у всіх групах. PE можуть читати з або писати в будь-який елемент об'єкта пам'яті даного типу. З іншого боку читання і запис можуть кешуватись залежно від можливостей OpenCL-пристрою.

2. Постійна пам'ять: області глобальної пам'яті, що залишається постійною протягом виконання ядра. Головний пристрій виділяє пам'ять і ініціалізує об'єкти, що розташовані в ній.

3. Локальна пам'ять: область пам'яті доступна тільки на рівні робочої групи. Ця область пам'яті може бути використана для виділення змінних, що є загальними для всіх PE, які працюють в одній групі. Локальна область пам'яті може бути відображена на ділянку глобальної пам'яті.

4. Приватна пам'ять: область пам'яті, що доступна тільки на рівні PE. Змінні, визначені в одному такому елементі, не доступні для іншого.

Табл. 1 описує можливості виділення пам'яті керуючою програмою та ядром, тип виділення (статична (часу компіляції) або динамічна (часу виконання)) і тип дозволеного доступу.

Таблиця 1.

	Глобальна	**Постійна**	**Локальна**	**Приватна**
Головний пристрій	Динамічна з доступом на читання і запис	Динамічна з доступом на читання і запис	Динамічна без доступу	Без можливості виділення і доступу
OpenCL-пристрій	Без можливості виділення з доступом на читання і запис	Статична з доступом лише на читання	Статична з доступом на читання і запис	Статична з доступом на читання і запис

Застосування, що виконується головним пристроєм, використовує API OpenCL для створення об'єктів у глобальній пам'яті та пам'яті для постановки в чергу команд, що працюють з ними.

Модель пам'яті головного пристрою та OpenCL-пристрою у більшості випадків реалізації є незалежними один від одного. Перша із зазначених моделей визначається поза OpenCL-специфікацією і не обмежена умовами останньої. Проте в деяких випадках необхідно визначити принципи взаємодії цих моделей. Ця взаємодія відбувається одним з двох способів: шляхом явного копіювання даних або шляхом відображення об'єктів з однієї пам'яті в іншу.

Явне копіювання передбачає наявність у черзі команди для передачі даних між пам'яттю OpenCL-пристрою та пам'яттю головного пристрою. Така команда передачі даних може бути блокуючою або неблокуючою.

Виклик блокуючої функції OpenCL зупиняє виконання потоку інструкцій головним пристроєм до моменту, коли асоційовані з нею ресурси пам'яті можуть бути безпечно використані (в сенсі цілісності даних). У випадку ж неблокуючого виклику функції потік інструкцій продовжує оброблятися незалежно від стану пам'яті, що асоційована з викликом. У цьому випадку потрібна додаткова синхронізація головного пристрою для можливості роботи з даним блоком пам'яті.

Метод відображення (mapping/unmapping) дозволяє співставляти регіони пам'яті пристроїв і також існує у формі блокуючої і неблокуючої взаємодії.

Direct Compute

Поява DirectX 11.0 надала можливість створювати обчислювальні шейдери, що призначені для виконання алгоритму, який допускає можливість розпаралелювання за даними на відеоадаптері, і є основою технології Direct Compute [8, 9]. При цьому для обміну даними між CPU та GPU використовують ті самі методи, що й для інших видів шейдерів. Створення та використання таких паралельних програм є можливим лише для тих відеоадаптерів, які апаратно підтримують DirectX 10.0 (10.1) та 11.0.

Програма, що використовує DirectCompute, розподіляє паралельну роботу між групами потоків, запускаючи на виконання потокові групи для вирішення поставленої задачі. Розмір таких груп встановлюється програмним чином, як і їх кількість. Розмірність таких груп і їхні сітки варіюється від одного до трьох, надаючи можливість індексувати потоки та групи відповідною кількістю чисел. Кожен виконуваний потік має доступ до своїх індексів, аналізуючи які, може визначити яку ділянку пам'яті слід обробляти.

Два потоки, що належать одній групі, можуть мати доступ до спільної ділянки пам'яті (shared memory), латентність якої набагато менша за латентність глобальної пам'яті. Проте її об'єм — обмежений, а доступ на читання-запис декількома потоками з однієї групи потребує синхронізації.

У багатьох задачах, що розв'язуються за допомогою відеоадаптерів, існує проблема синхронізації усіх потоків у всіх групах. Щоб зрозуміти чому не існує такої синхронізації, слід розглянути процес виконання груп потоків. Кожна група розбивається на підгрупи (warp) кількістю 32 потоки. В кожен момент часу один потоковий мультипроцесор (SM) виконує лише одну підгрупу певної групи потоків. Крім того, група потоків завжди виконується на одному мультипроцесорі, перемикання виконання однієї підгрупи потоків на іншу відбувається за рахунок латентних операцій звернення до різних видів пам'яті. Таким чином, щоб отримати пікову продуктивність відеоадаптера, слід навантажувати корисною роботою всі потокові мультипроцесори, тобто створювати таку кількість груп, яка відповідає їхній кількості.

Проблема глобальної синхронізації виникає для тих алгоритмів, які рекурсивно отримують нові дані, використовуючи старі. Якщо б така проблема вирішувалася за допомогою механізмів мови HLSL, створені застосування втратили б мас-

штабованість. Крім того, виникала б можливість запирання потоків (deadlock). Проте така проблема вирішується шляхом використання CPU як арбітра, що для різних груп потоків створює глобальні точки синхронізації.

У процесі свого виконання потік може використовувати різні види пам'яті, в комірках яких час життя інформації може залежати від часу життя самого потоку, часу життя його групи або часу життя програми CPU, що зарезервувала ці комірки. Розміщення змінної в одній із зазначених раніше видів пам'яті визначається програмістом і має досить високий вплив на продуктивність застосування. Табл. 2 ілюструє основні види змінних, які доступні при написанні програми для відеоадаптера.

Таблиця 2.

Тип змінної	**Межі доступу**	**Час життя**	**Види доступу**	**Місце знаходження**
Local scalar	потік	потік	читання/запис	on-chip(registers)
Local array	потік	потік	читання/запис	off-chip
Shared	група	група	читання/запис	on-chip(RAM)
Constant	глобальний	CPU	читання	on-chip(cache)
Shader Resource View	глобальний	CPU	читання	off-chip
Unordered ResourceView	глобальний	CPU	читання/запис	off-chip

Для того, щоб отримати максимальну швидкодію алгоритму, що виконується на відеоадаптері, його реалізація має відповідати ряду вимог, таких як:

1) мінімізація обміну даними між CPU та GPU через PCI;
2) мінімізація доступу до глобальної пам'яті;
3) об'єднанння (coalesced) доступу до глобальної пам'яті;
4) максимальне використання спільної пам'яті;
5) мінімізація кількості використовуваних локальних змінних;
6) групування даних, що використовуються разом, у структури;
7) кратність сумарної кількості потоків у групі має бути 32 (warp length);
8) рівність кількості груп кількості мультипроцесорів;
9) наявність більш ніж одного warp у групі;
10) зменшення кількості розгалужень у warp-і, що використовують індекс потоку до мінімуму;
11) використання оптимізації компіляції.

Висновки

Аналізуючи тенденції розвитку GPGPU-технологій, слід зазначити, що GPU стає все більш незалежним пристроєм по відношенню до CPU. Це демонструють такі технології як динамічний паралелізм, динамічне виділення пам'яті, RDMA. Як результат, слід очікувати появу універсальної архітектури GPU-CPU, що об'єднає функціональні можливості цих пристроїв.

Кожна із розглянутих реалізацій GPGPU має як свої переваги, так і недоліки. Так, CUDA C, що створена фірмою NVidia, підтримується відеоадаптерами лише цієї фірми. Direct Compute, у свою чергу, підтримує ширший набір апаратних платформ. Крім того, ця технологія інтегрована в DirectX, що дозволяє достатньо прос-

то використовувати GPU в цілях обчислень математичних алгоритмів одночасно з процесом рендерінгу зображень. Недоліком є можливість використання цієї технології лише в програмній платформі Windows. OpenCL є найбільш універсальною технологією, що, з одного боку, є перевагою, а з іншого — веде до втрати специфічності для кожної платформи і, як наслідок, до меншого виграшу у швидкодії.

Назва	Фірма виробник	Підтримувані апаратні платформи	Підтримувані програмні платформи
CUDA C	NVidia	NVidia GPUs	Windows / Unix / Linux
Open CL	Khronous group	NVidia, ATI GPUs	Windows / Unix / Linux
Direct Compute	Microsoft	NVidia, ATI GPUs	Windows

Усі зазначені реалізації підходу GPGPU мають досить подібні моделі створення та виконання паралельних застосувань. CUDA C дозволяє створювати код, що виконуватиметься на відеоадаптері за розширення мови C додатковими операторами та функціями. Створений *.cu-файл компілюється або в проміжний код, або в набір інструкцій машинною мовою. Аналогічно, Direct Compute використовує мову HLSL (High-level shader language) і компілятор fxc. OpenCL використовує GLSL.

1. *CUDA C* Programming Guide [Електронний ресурс]. — Режим доступу: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. — Дата доступу 5.11.2012. — Nvidia Corporation.

2. *Боресков А.В.* Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов. — ДМК-Пресс, 2010. — 232 с.; ISBN 978-5940745785.

3. *Sanders Jason.* CUDA by Example: An Introduction to General-Purpose GPU Programming / Jason Sanders, Edward Kandrot. — Addison-Wesley Professional. Ann Arbor, Michigan (USA). — July 2010. — 312 p.; ISBN 978-0131387683

4. *Kaufmann Morgan.* CUDA Application Design and Development, Rob Farber / Morgan Kaufmann. — Waltham, Massachusetts (USA). — November 14, 2011. — 336 p.; ISBN 978-0123884268.

5. *Богачёв К.Ю.* Основы параллельного программирования / Богачёв К.Ю. — М.: БИНОМ. Лаборатория знаний, 2003. — С. 232–292.

6. *Погорілий С.Д.* Дослідження паралельних версій алгоритму Флойда-Уоршала для SMP- та MPP-архітектур / С.Д. Погорілий, М.І. Трибрат, Д.Ю. Вітель // Математичні машини та системи. — 2011. — № 3.

7. *Don Syme.* Expert F# / Don Syme, Adam Granicz, Antonio Cisternino. — Berkeley: Apress. 2007. — 639 p.

8. *Sherrod Allen.* Game Development with Microsoft DirectCompute / Allen Sherrod. —Course Technology PTR. — December 9, 2011. — 496 p.; ISBN 978-1435458468.

9. *DirectCompute* Expert Roundtable Discussion [Електронний ресурс]. — Режим доступу: http://channel9.msdn.com/blogs/gclassy/directcompute-expert-roundtable-discussion. — Дата доступу 5.11.2012. — Microsoft Corporation.

Надійшла до редакції 25.02.2013