УДК 004.3

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

Новітні архітектури відеоадаптерів. Технологія GPGPU. Частина 1

Проведено порівняльний аналіз новітніх архітектур відеоадаптерів Tesla, Fermi та Kepler і інструментальних засобів створення застосувань NVidia CUDA. Визначено основні методи роботи з глобальною пам'яттю. Розглянуто методи формування потоків для цих програмно-апаратних платформ.

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).*

Вступ

Основною особливістю сучасних графічних адаптерів, які використовують графічні процесори (Graphical Processing Unit, GPU), є наявність набору потокових мультипроцесорів (Streaming multiprocessor, SM), що використовувалися раніше лише в алгоритмах і задачах, пов'язаних з обробкою графічних зображень. Існують дві основні програмні технології (інтерфейси), що застосовуються програмістами для створення програм такого напрямку: DirectX та OpenGL. Вони використовують пам'ять відеоадаптера для розміщення структур даних, таких як текстури, буфери тощо, визначають конвеєр обробки, кожен етап якого відповідає за свої специфічні дії: растеризацію, інтерполяцію, блендинг, теселяцію тощо; визначають дві *C*-подібні мови програмування, що дозволяють створювати застосування, які виконуються саме на GPU: High level shader language (HLSL), OpenGL shading language (GLSL). Основні принципи зазначених програмних інтерфейсів є досить подібними, проте відрізняються орієнтацією на програмні платформи, що використовуються системою.

Технологія обчислень загального призначення на графічних процесорах (GPGPU) ґрунтується на використанні великої кількості процесорів GPU, що працюють паралельно, для обробки даних за допомогою алгоритмів загального при-

© С. Д. Погорілий, Д. Ю. Вітель, О. А. Верещинський

значення (наукових чи інших, але не обов'язково пов'язаних з обробкою зображень). Можливість проведення таких обчислень реалізується за допомогою наступних інтерфейсів: NVidia CUDA [1–5], DirectCompute [9, 10], OpenCL [6–8] та ATI Stream (пропрієтарна технологія AMD/ATI). Принципи побудови застосувань для них є досить подібними. Потоковий процесор на GPU має простішу структуру, ніж вузол CPU. Тобто такі вузли менш універсальні і виконують менший набір функцій, аніж вузли процесора. Проте, оскільки їхня кількість велика, то для певного набору задач можна досягти суттєвого приросту в швидкодії. Найкращого прискорення вдається досягти для алгоритмів, що підтримують концепцію паралелізму за даними (один паралельний потік обробляє свою область у пам'яті). Тому зазвичай GPGPU застосовують у наступних сферах: обробка зображень (Reduction, Histogram, Fast Fourier Transform, Summed Area Table); обробка відеоданих (Transcode, Digital Effects, Analysis); лінійна алгебра; моделювання (Technical, Finance, Academic, Some Databases) тощо.

Слід зазначити, що кожен із наведених програмних інтерфейсів створення GPGPU-застосувань має свій рівень абстракції по відношенню до апаратної реалізації цільових архітектур. Так, OpenCL є специфікацією, що орієнтована на апаратні прискорювачі різного типу і визначає також у своїй моделі виконання паралелізм за завданнями як один із можливих варіантів. NVidia CUDA позиціонується фірмою-розробником як програмна модель і платформа паралельних обчислень загального призначення. Вона орієнтована виключно на адаптери цієї фірми і має низький рівень абстракції щодо архітектури останніх. Direct Compute — технологія, що розроблена фірмою Microsoft як логічне розширення DirectX. Дане розширення використовує інфраструктуру і основні підходи цієї технології і тому, на відміну від NVidia CUDA, по-перше, орієнтована на операційне середовище Windows; по-друге, орієнтована на застосування професійними програмістами і тяжко адаптується для використання в науковій сфері.

Основні риси інструментальних засобів NVidia CUDA, Direct Compute, Open CL

Кожний із зазначених інтерфейсів визначає наступні поняття: ієрархія потоків (модель виконання), ієрархія пам'яті (модель пам'яті), взаємодія потоків центрального процесора і відеоадаптера та потоків відеоадаптера між собою.

Ієрархія потоків — вид організації потоків, що виконуються на відеоадаптері. Для всіх зазначених технологій потоки організовані у вигляді 3D-блоків (блок (block) у NVidia CUDA, група у DirectCompute, та робоча група (work group) у OpenCL). Кожен такий блок (група), у свою чергу, є елементом 3D-сітки (сітка (grid) — NVidia CUDA, dispatch — DirectCompute, NDRange — OpenCL). Таким чином потік, що виконуватиметься на графічному процесорі, унікально ідентифікується двома векторами — індексом блоку (групи) в сітці блоків і індексом потоку в блоці. Така організація потоків пов'язана з особливістю архітектури самого відеоадаптера. На рис. 1 наведено приклад ієрархічної організації потоків (робочих елементів у OpenCL) для технології OpenCL (розмірність сітки і блоку для спрощення обрана рівною двом), де (w_x, w_y) — індекси робочої групи в сітці, (S_x, S_y) — розміри робочої групи, (s_x, s_y) — індекси робочого елемента в групі,

(F_x, F_y) — зміщення робочої групи в сітці (першого потоку в групі щодо першого потоку в сітці)).

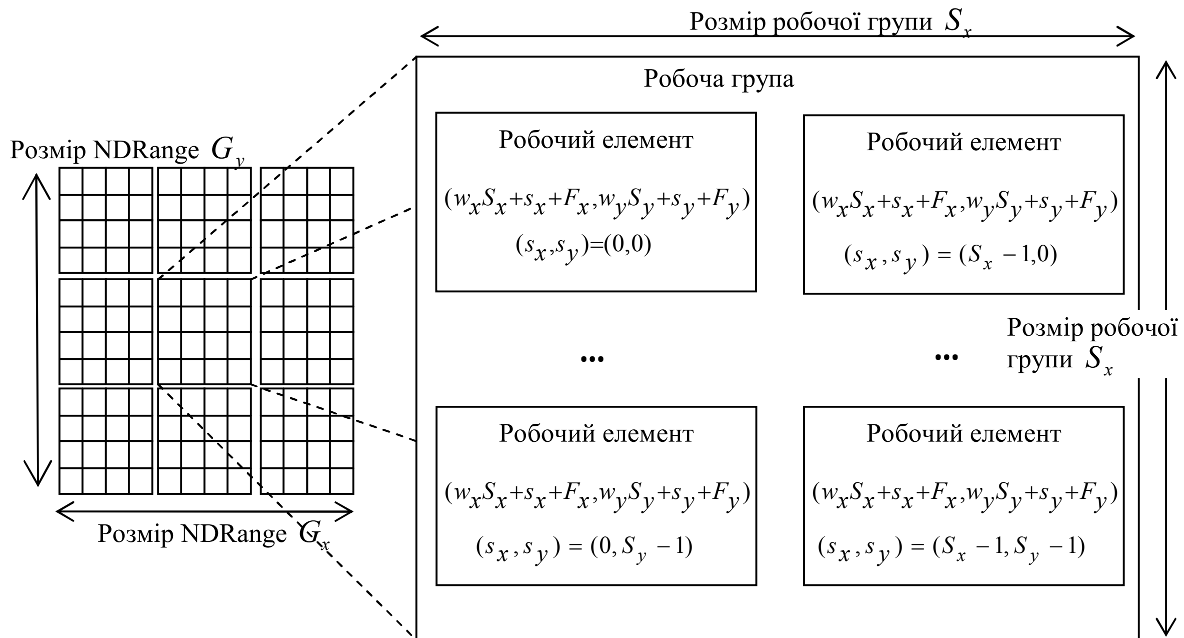

Рис. 1. Модель виконання OpenCL

Кожна із зазначених технологій визначає набір *типів пам'яті*, що використовуються потоками під час обрахунків. До основних типів належать глобальна пам'ять відеоадаптера, константна пам'ять, спільна пам'ять, локальна пам'ять, регістрова пам'ять. Проте деталі роботи із кожним таким типом залежать від обраної технології. Їхній загальний опис представлено в табл. 1.

Таблиця 1

Тип памяті	**Межі доступу**	**Межі існування**	**Час життя**	**Види доступу**	**Місце знаходження**
Глобальна пам'ять	Усі потоки в сітці	Час життя процесу на CPU	Процес CPU	читання/запис	поза кристалом (off-chip)
Локальна пам'ять	Потік-власник	Час життя потоку-власника на GPU	Потік-власник GPU	читання/запис	на кристалі або поза ним (on-chip(cache) або off-chip)
Спільна пам'ять	Блок (група)	Час життя усього блок (групи) на GPU	Блок (група) GPU	читання/запис	на кристалі (on-chip(RAM))
Константна пам'ять	Усі потоки в сітці	Час життя процесу на CPU	Процес CPU	читання	на кристалі (on-chip(cache))
Регістрова пам'ять	Потік-власник	Час життя потоку-власника на GPU	Потік-власник GPU	читання/запис	на кристалі (on-chip(registers))
Текстурна пам'ять	Усі потоки в сітці	Час життя процесу на CPU	Процес CPU	читання	Груповий кеш (SM group cache or off-chip)

Взаємодія потоків CPU та GPU між собою також залежить від типу обраного інтерфейсу програмування. Застосування на GPU запускається асинхронно до коду на CPU. Це означає, що потік виконання останнього не зупиняється і не очікує на результат обробки GPU. Для такого очікування інтерфейси надають додаткові механізми синхронізації. Окрім цього можлива також синхронізація потоків між собою на рівні блоку (групи) потоків. Шаблони синхронізаційної взаємодії та обміну даними розглянуто далі.

Архітектури сучасних GPU від NVidia. NVidia CUDA

Основою архітектури NVIDIA GPU є масштабований масив потокових мультипроцесорів (SM). Коли застосування CUDA на головному процесорі викликає сітку CUDA-ядер, блоки сітки розподіляються поміж наявними мультипроцесорами (рис. 2).

На одному мультипроцесорі може виконуватись набір блоків, проте потоки в одному блоці виконуються завжди на одному мультипроцесорі. Після закінчення виконання певного блоку SM, що звільнився, починає виконувати наступний наявний вільний блок. SM спроектовано за принципами SIMT (Single-Instruction, Multiple-Thread) так, що він може паралельно виконувати сотні потоків. Інструкції також проходять стадію конвеєризації. Проте, на відміну від CPU, вони виконуються послідовно без можливості спекулятивного виконання і прогнозування гілок розгалуження.

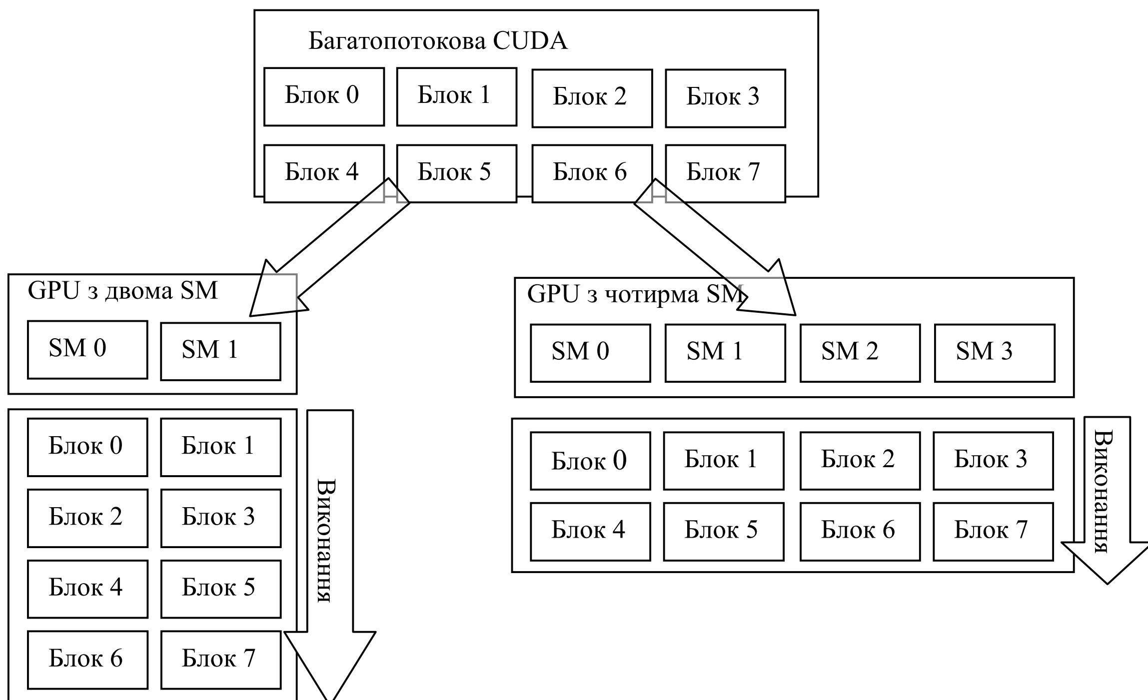

Рис. 2. Процес поділу блоків між потоковими мультипроцесорами

Мультипроцесор оперує групами потоків, що має розмір 32, і називається *warp.* Кожен потік у такій групі виконується паралельно з іншими, починаючи з певної, спільної для групи, адреси команд. Проте кожен з цих потоків має власний лічильник адреси інструкцій і стан регістрів і тому може виконувати іншу послідовність команд (іншу гілку операцій). Warp поділяється на підгрупи *half-warp* та *quarter-warp*. Деякі команди виконуються мультипроцесором саме по відношенню до цих підгруп.

Блоки потоків, що подаються на виконання мультипроцесора, розбиваються на warp послідовно, використовуючи лінійний індекс потоку в блоці. Окремий модуль, що має назву warp scheduler, відповідає за передачу набору інструкцій кожній групі warp за певне число тактів. Warp виконує 32 загальні інструкції в один момент часу паралельно. Проте, якщо в потоці команд зустрічається розгалуження його виконання, мультипроцесор змушений виконувати кожну з гілок послідовно. Це суттєво впливає на продуктивність отриманої реалізації алгоритму і тому в застосуваннях слід мінімізувати кількість розгалужень потоку виконання на рівні групи warp.

Кількість блоків (warp-груп), що можуть оброблятися мультипроцесором, залежить як від кількості регістрів і спільної пам'яті використаного відеоадаптера, так і від реалізації застосування для нього. Ці значення, як і більшість інших параметрів відеоадаптера, визначаються можливостями обчислень (compute capability — CC) цільового пристрою. CC складається з двох частин — головного та другорядного чисел ревізії. Головне число ревізії CC визначає апаратну архітектуру відеоадаптера. На сьогодні існують наступні архітектури:

— Tesla architecture — пристрої з CC рівним 1.x;

— Fermi architecture — пристрої з CC рівним 2.x;

— Kepler architecture — пристрої з CC рівним 3.x

Слід однак зазначити, що фірма NVidia має серії (лінії) відеоадаптерів, що орієнтовані на певні сфери використання:

1) GeForce — лінія відеоадаптерів, що орієнтована на розважальну індустрію. Найбільш потужним на даний час є відеоадаптер GeForce GTX 690, що має архітектуру Kepler. Серія GPU GTX є більш потужною за серію GT;

2) Quadro — серія, що орієнтована на професійних дизайнерів. Надає більше можливостей по обробці відео, зображень і створенню графічних застосувань (ігор, CAD-систем тощо);

3) Tegra — лінія для використання у мобільних рішеннях (мобільних телефонах, смартфонах і т.д);

4) Tesla — серія для використання у високопродуктивних (кластерних) обчисленнях. GPU позбавлені деяких специфічних для графіки функцій і широко застосовуються у науковій сфері. Найпотужніший на даний час відеоадаптер — Tesla K20 — також спроектований за Kepler-архітектурою;

5) NVS.

Зміни в головному номері ревізії CC відображають покрокові вдосконалення ядра GPU та появу додаткових можливостей. У табл. 2 наведено порівняльну характеристику існуючих архітектур [1] з позиції апаратного складу мультипроцесора. Часто в якості порівняльної характеристики потужності GPU-пристроїв виступає загальна кількість їхніх CUDA-ядер. CUDA-ядро — модуль мультипроце-

сора, що відповідальний за різні арифметичні операції. Для виконання спеціальних операцій (наприклад, тригонометричних функцій) над числами з плаваючою точкою використовуються окремі модулі, кількість яких менша. В NVidia GPU підтримується стандарт IEEE 754-2008 (з деякими розбіжностями) по відношенню до таких операцій. Робота з числами подвійної точності значно повільніше виконується аніж над числами одиничної точності. Тому для отримання максимальної продуктивності реалізації в ній часто нехтують точністю обчислень.

Таблиця 2

Архітектура – CC	Апаратний склад SM
Tesla – 1.x, де x (другорядне число ревізії) може бути рівним 0, 1, 2 або 3	• 8 CUDA-ядер для арифметичних операцій • 2 модулі для спеціальних функцій, що працюють на множені чисел з плаваючою точкою одинарної точності (також виконують множення чисел) • 1 warp scheduler • Поява модуля для операцій з плаваючою точкою подвійної точносі при x = 3. • Кеш, що розмішений на чипі, для постійної пам'яті (в динамічній пам'яті GPU) • Texture Processor Clusters — 2 (x ≤ 1) або 3 (x > 1) SM з асоційованим текстурним кешем, що пришвидшує роботу із текстурною пам'яттю відеоадаптера • Локальна та глобальна пам'ять не мають асоційованих кешів
Fermi – 2.x, де x (другорядне число ревізії) може бути рівним 0 або 1	• 32 (x = 0) або 48 (x = 1) CUDA-ядер для арифметичних операцій • 4 (x = 0) або 8 (x = 1) модулі для спеціальних функцій, що працюють на множені чисел з плаваючою точкою одинарної точності (також виконують множення чисел) • 2 warp scheduler • Кеш, що розмішений на чипі, для постійної пам'яті (в динамічній пам'яті GPU) • L1 кеш для локальної та глобальної пам'яті (використовує спільний простір із спільною пам'яттю) • L2 кеш (спільний для усіх SM) для локальної та глобальної пам'яті • Graphics Processor Clusters — 4 SM з асоційованим текстурним кешем, що пришвидшує роботу із текстурною пам'яттю відеоадаптера
Kepler – 3.x, де x (другорядне число ревізії) може бути рівним 0 або 5	• 192 CUDA-ядер для арифметичних операцій • 32 модулі для спеціальних функцій, що працюють на множені чисел з плаваючою точкою одинарної точності (також виконують множення чисел) • 4 warp scheduler • Кеш, що розмішений на чипі, для постійної пам'яті (в динамічній пам'яті GPU) • L1 кеш для локальної та глобальної пам'яті (використовує спільний простір зі спільною пам'яттю) • L2 кеш (спільний для усіх SM) для локальної та глобальної пам'яті • Graphics Processor Clusters — 3 SM з асоційованим текстурним кешем, що пришвидшує роботу із текстурною пам'яттю відеоадаптера

Для постановки на виконання інструкції у групі warp на відеоадаптері з CC 1.x warp scheduler модуль втрачає 4 такти для операцій з числами одиничної точності, 32 такти на операції з числами подвійної точності (x = 3) та 16 спеціальних функцій на операції з числами одиничної точності. Для пристроїв із CC, що вища за 1.x, одночасно на виконання може бути поставлено декілька інструкцій для SM. Це можливо завдяки наявності декількох модулів warp scheduler. Додаткові особливості пристроїв різної архітектури наведено в табл. 3.

Таблиця 3

Архітектура — CC	Tesla — 1.x	Fermi — 2.x	Kepler — 3.x
Розмірність сітки блоків	2	3	
Surface функції	Не підтримує	Підтримує	
Додаткові функції синхронізації	Не підтримує	Підтримує: __syncthreads_count, __syncthreads_and, __syncthreads_or	
Функції голосування потоків у warp	Не підтримує	Підтримує: __ballot	
Funnel shift	Не підтримує		Підтримує
Атомарні операції	Залежно від x, типу цільової пам'яті та типу операнда	Підтримує всі	
Максимальна кількість потоків в блоці	512	1024	
Максимальна кількість блоків на SM	8	16	
Максимальна кількість warp груп	24, при x >1 – 32	48	64
Максимальна кількість 32-бітних регістрів на потік	128	63	255
Максимальна кількість спільної пам'яті	16 КБ	48 КБ	
Кількість банків спільної пам'яті	16	32	

Структурна декомпозиція типової програми для CUDA

CUDA визначає розширення мови C/C++, яке дозволяє створювати код, що виконуватиметься на відеоадаптері, визначаючи C функції-ядра (*kernels*). Ці функції виконуються кожним потоком в ієрархії сітка–блок–потік і визначає ряд контекстних змінних:

1) gridDim — векторна змінна, що зберігає розмірності сітки;
2) blockIdx — векторна змінна, що зберігає індекс блоку в сітці;
3) blockDim — векторна змінна, що зберігає розмірність блоку;
4) threadIdx — векторна змінна, що зберігає індекс потоку в блоці;
5) warpSize — змінна, що зберігає розмір warp групи.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Код, що виконуватиметься на GPU, можна комбінувати з кодом для CPU. Програма обробляється за допомогою `nvcc` компілятора, що проводить пошук дириктив у тексті, визначає частину застосування, що виконуватиметься на відеоадаптері і компілює її. CPU-код обробляється за допомогою компілятора цільової системи. Результатом `nvcc` може бути:

1) cudabin — бінарний файл з кодом для GPU;

2) pfx файл, що містить інструкції на проміжній мові PFX. Під час виконання такі файли обробляються за допомогою JIT (Just in time) компілятора в бінарні коди. Результат JIT керується, а кеш зберігається до моменту зміни цільового відеоадаптера.

Слід зазначити, що CUDA представляє два набори API функцій: CUDA Runtime API та CUDA Driver API. CUDA Runtime API є надбудовою над CUDA Driver API і дозволяє спростити процес побудови застосування для GPU. CUDA Driver API надає низькорівневий інтерфейс керування відеоадаптером. Наприклад, ядро може бути створене окремо, а потім з використанням CUDA Driver API завантажене у відеоадаптер. CUDA Runtime API спрощує даний процес шляхом введення спеціального синтаксису виклику ядра з CPU-програми. Наступний лістинг демонструє приклад ядра, що виконує додавання двох матриць на відеоадаптері.

`MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C)` — виклик ядра з кода CPU, що фактично на низькому рівні перетворюється на процес завантаження ядра в пам'ять GPU та запуск його на виконання з параметрами сітки `(numBlocks, threadsPerBlock)`.

Зазвичай типова програма з використанням CUDA включає такі кроки:

1) виділення пам'яті на GPU;

2) копіювання даних між CPU та GPU;

3) запуск ядра (починаючи з архітектури Fermi можливий запуск кількох ядер одночасно);

4) копіювання результатів даних між GPU та CPU;
5) звільнення виділеної пам'яті на GPU.

Кожен потік CPU прив'язаний до свого CUDA-контексту, а кожен такий контекст асоційований з CUDA-сумісним GPU. Існує набір API, який дозволяє створювати новий контекст, змінювати відеоадаптер для контексту, міняти контексти для потоків. Таким чином, CPU може обслуговувати одразу декілька відеоадаптерів.

Особливості роботи з пам'яттю

Типи пам'яті, що використовуються CUDA-програмою, були наведені в табл. 1. Її виділення можна проводити або лінійно, або за допомогою CUDA-масивів. У першому випадку робота з пам'яттю ведеться за допомогою вказівників. У другому — за допомогою дескриптора, що описує певну ділянку пам'яті. CUDA-масиви зазвичай використовуються для роботи з текстурами (textures) та поверхнями (surfaces). Щоб виділити необхідний об'єм лінійної пам'яті слід скористатися функцією `cudaMalloc` (чи однією з подібних). Аналогічно, пам'ять звільняється командою `cudaFree`. Доступ до даних, що розміщені на відеокарті, може здійснювати лише GPU.

Розглянемо копіювання динамічно розміщених даних. При простому копіюванні вказівників отримуємо так зване «поверхневе» копіювання, яке має сенс лише в межах одного типу пам'яті. В цьому випадку дані безпосередньо не копіюються. Це може призвести до проблем часу виконання при неправильному звільненні пам'яті. Для глибокого копіювання необхідно заново виділити потрібний об'єм пам'яті і скористатись однією з багатьох функцій для копіювання (`memcpy`, `std::copy`, `cudaMemcpy` тощо). Зокрема, копіювання між різними типами пам'яті можливе лише з використанням CUDA API.

Розв'язати деякі з указаних проблем дозволяють так звані «розумні» вказівники (smart pointers). Як приклад можна навести `auto_ptr` із стандартної бібліотеки STL чи `shared_ptr` із бібліотеки Boost. Вони дозволяють проводити вдосконалену політику керування пам'яттю (memory management). Але ці конструкції не розраховані на роботу в середовищі CUDA.

Таким чином, формується завдання на створення деякого уніфікованого інтерфейсу вказівника, який дозволив би однаково працювати з даними, незалежно від їхнього розміщення, приховуючи реалізацію низькорівневих операцій, які стали би прозорими для користувача (тут і надалі, користувач — розробник, що використовує можливості створеного вказівника).

На інтерфейс накладається низка вимог, сформованих зі сторони функціоналу. Таким чином, постає задача настроювання компонента. Наприклад, може виникнути потреба створити вказівник для виділення динамічної області з глибоким копіюванням для однопоточного середовища, а також вказівник для виділення пам'яті на відеокарті з підрахунком посилань для багатопоточного середовища.

Виникає проблема створення достатньо гнучкого інтерфейсу, який міг би дати користувачу можливість настроювання окремих компонентів, і в той же час мав розумні розміри.

Реалізація всього можливого функціоналу під оболонкою деякого універсального інтерфейсу не є вдалим вибором, оскільки призведе до створення громіздких компонентів (яскравим прикладом у даному випадку є клас `std::string` зі стандартної бібліотеки C++, переповнений функціоналом і дублюванням операцій). Програми, що будуть використовувати такі класи, працюють набагато повільніше аніж аналогічні програми, розроблені самотужки.

Але однією з найбільших проблем є втрата *безпеки статичних типів*. Однією із цілей будь-якої архітектури є закладання деяких аксіом «за означенням». В ідеальному випадку більшість обмежень мають накладатися ще на етапі компіляції коду. В межах громіздкого інтерфейсу дуже важко врахувати всі подібні обмеження. Це призводить до збільшення розриву між *семантичною* та *синтаксичною* правильністю програми. Тобто виникає небезпека написання синтаксично правильного коду, який може зруйнувати основні зв'язки всередині бібліотеки.

Один із найочевидніших підходів — фактично, груба сила — полягає в реалізації різних проектних рішень у вигляді окремих класів більш скромного розміру. Кожен з таких класів міг би інкапсулювати набір взаємопов'язаних алгоритмів, які б забезпечували швидку й налагоджену поведінку.

У такому випадку логічно очікувати появи численних реалізацій:

```
HeapDeepCopySingleThreadPtr
DeviceRefCountMultiThreadPtr
```

Для такого розв'язку проблеми характерним є різкий ріст кількості різних варіантів проектних рішень. Тільки два вищезгаданих класи можуть легко призвести до виникнення багатьох нових комбінацій. Перетворення типів викличе ще більшу кількість варіацій, які, рано чи пізно, вийдуть за рамки можливостей програміста та користувача бібліотеки. Як бачимо, подолати експоненційний ріст варіантів за допомогою грубої сили неможливо.

Отже проблеми такого підходу очевидні:

1) масштабованість такого рішення практично нульова. Щоб додати новий тип даних потрібно повністю його описувати;

2) велика частина коду просто копіюється (Copy-Paste);

3) отримані структури даних ніяк не зв'язані між собою, хоча існують деякі логічні операції (наприклад, присвоювання, копіювання), які передбачають існування зв'язку. Фактично, ми нехтуємо можливостями, які дає використання поліморфізму.

Бібліотеки такого типу не потребують значних інтелектуальних зусиль для розробки, але є вкрай негнучкими. Найменша непередбачувана дрібниця може привести всі класи до плачевного стану.

Щоб уникнути значної частки описаних проблем, можна використати спадкування. Це, в свою чергу призведе до створення складних ієрархій, які доведеться описувати вручну. Знову ж таки при розширенні архітектури програми в області алокації пам'яті кількість нових класів зростатиме геометрично.

Глобальна пам'ять

Слід зазначити, що доступ до глобальної пам'яті, яка лінійно виділена, є оптимальним умові, що дані в глобальній пам'яті мають правильне вирівнювання.

Тому в деяких випадках доцільніше використовувати функції `cudaMallocPitch`, `cudaMalloc3D`, що автоматично вирівнюють данні в пам'яті для двовимірних і тривимірних масивів. Для копіювання в цьому випадку застосовуються функції `cudaMemcpy2D`, `cudaMemcpy3D`. Інші API-функції роботи з пам'яттю можна знайти в [1].

Для оптимального доступу до глобальної пам'яті для архітектури Tesla з CC 1.0, 1.1 слід дотримуватися досить жорстких обмежень у застосуванні. Запит з warp-групи може створити мінімум 2 транзакції до пам'яті (для кожної half-warp групи). Для мінімізації кількості транзакцій необхідно щоб:

1) розмір слова, до якого йде запит з потоку групи, був рівний 4, 8 або 16 байт;

2) при розмірі слова, рівному 4, усі 16 слів знаходилися в одному сегменті розміром 64 байт;

3) при розмірі слова, рівному 8, усі 16 слів знаходилися в одному сегменті розміром 128 байт;

4) при розмірі слова, рівному 16, усі 16 слів знаходились у двох послідовних сегментах розміром 256 байт;

5) доступ до слова має бути послідовним, а номер слова в сегменті має відповідати номеру потоку в half-warp групі.

Для доступу до слів, що мають розмір 4 або 8 (16), мінімальна кількість транзакцій рівна 1(2). При цьому кількість транзакцій мінімізується навіть у випадку розгалуження warp-групи. Якщо кількість транзакцій не вдається мінімізувати, то створюється 16 транзакцій розміром у 32 байти, що дуже сповільнює роботу алгоритму. Архітектура глобальної пам'яті була вдосконалена в Tesla CC 1.2, 1.3. Мінімізація кількості транзакцій буде проходити навіть у випадку, коли потоки створюють запит до слів з номерами, що не відповідають їхнім номерам у half-warp групі. Алгоритм побудови транзакцій наступний.

1. Пошук сегмента пам'яті, що необхідний потоку з найменшим індексом у групі активних на читання потоків з half-warp. Розмір сегмента залежить від розміру слова, що намагається отримати потік: 32 для слова в 1 байт, 64 для слова в 2 байти, 128 для слів з розміром 4, 8 та 16 байт.
2. Пошук потоків серед активних на читання в групі, чиї адреси запиту лежать у цьому ж сегменті.
3. Зменшення розміру транзакції. Якщо її розмір рівний 128 байт, і доступ йде лише до першої або другої половини сегмента, то транзакція зменшується до розміру в 64 байт. Аналогічні зменшення проводяться рекурсивно для наступних розмірів. Мінімальний розмір транзакції — 32 байти.
4. Проведення транзакції і видалення потоків з групи активних на читання.
5. Якщо є потоки, що активні на читання, то перехід на пункт 1.

У Fermi-архітектурі доступ до глобальної пам'яті буферизується в кеш-пам'яті (кешується). При цьому існує 2 типи кеш-пам'яті — L1 (локальний для SM), L2 (спільний для усіх SM). Лінія кешу — 128 байт, що відображається на 128 байт глобальної пам'яті. При цьому до L1 можливі лише транзакції розміром у 128 байт, а до L2 — 32 байти. Пропускна спроможність кеш-пам'яті набагато більша за пропускну спроможність глобальної пам'яті. Це значно пришвидшує роботу застосування якщо кількість кеш-промахів мала. Мінімальну кількість транзакцій

вдається отримати у випадках, коли потоки можуть створювати запит до будь-якого слова в сегменті, або запит до одного і того самого слова.

У Kepler-архітектурі доступ до глобальної пам'яті кешується лише на рівні L2 та read-only data кешу. Для того, щоб данні були збережені в read-only data кеші, необхідно використовувати до вказівників модифікатори `const` та `__restrict__`. Це вказує компілятору, що такі вказівники є не тільки не постійні данні, а що й не існує інших посилань на цю ж область пам'яті. Рис. 3 демонструє варіанти доступу до глобальної пам'яті для різних архітектур.

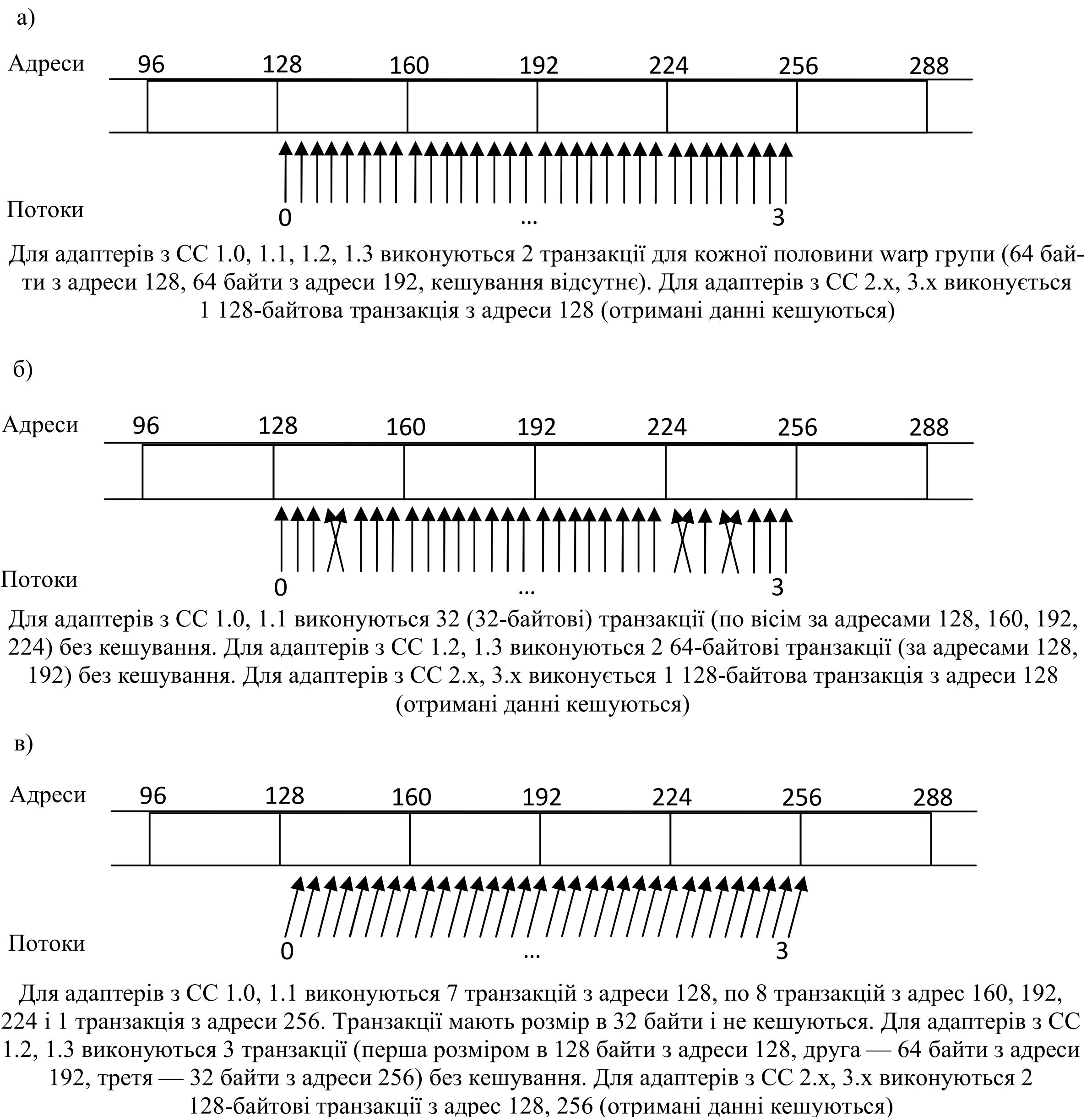

Рис. 3. Варіанти доступу до глобальної пам'яті: а) послідовний доступ з вирівнюванням; б) довільний доступ з вирівнюванням; в) доступ без вирівнювання

Висновки

Аналіз тенденцій розвитку GPGPU-технологій, свідчить про той факт, що GPU стає все більш незалежним пристроєм по відношенню до CPU. Це демонструють такі технології як динамічний паралелізм, динамічне виділення пам'яті, RDMA. Результатом цього процесу є очікування появи найближчим часом універсальної архітектури GPU-CPU, що об'єднає функціональні можливості цих пристроїв.

Розглянуто реалізацію інструментальних засобів технології GPGPU, яка має як свої переваги, так і недоліки. Засоби CUDA C, що створені фірмою NVidia, підтримують відеоадаптерами лише цієї фірми. З іншого боку, цей факт є і перевагою, оскільки, порівняно з іншими реалізаціями, інструментальні засоби CUDA C на низькому рівні підтримують специфічні для NVidia GPU інструкції, що в багатьох випадках може суттєво покращити швидкодію реалізації алгоритму.

Показано, що CUDA C дозволяє створювати код мовою, який є розширенням мови ANSI C99 додатковими операторами та функціями. Створений *.cu-файл компілюється або в проміжний код, або в набір інструкцій машинною мовою. Після компіляції одержаний файл виконуватиметься на відеоадаптері.

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

5. *David B. Kirk.* Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series) / David B. Kirk, Wen-mei W. Hwu. Morgan Kaufmann. — Waltham, Massachusetts, USA. — February 5, 2010. — 280 p. — ISBN 978-0123814722.

6. *OpenCL* Reference Pages [Електронний ресурс] — Режим доступу: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/. — Дата доступу: 9.11.2012. — The Kronos Group Inc.

7. *Heterogeneous* Computing with OpenCL. — 1-st Edition / Benedict Gaster, Lee Howes, David Kaeli [et al.]. — Waltham, Massachusetts, USA. — August 31, 2011. —296 p. — ISBN 978-0123877666.

8. *OpenCL* Programming Guide / Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung. — Addison-Wesley Professional, Boston. — July 23, 2011. — 648 p. — ISBN 978-0321749642.

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

10. *DirectCompute* Expert Roundtable Discussion [Електронний ресурс]. — Режим доступу: http://channel9.msdn.com/blogs/gclassy/directcompute-expert-roundtable-discussion. — Дата доступу: 3.11.2012. — Microsoft.

Надійшла до редакції 26.11.2012