УДК 004.4

С.Д. Погорілий, О.А. Верещинський

СТВОРЕННЯ МЕТОДИКИ ПРОЕКТУВАННЯ ЗАСТОСУВАНЬ ДЛЯ ПРОГРАМНО-АПАРАТНОЇ ПЛАТФОРМИ CUDA

Проаналізовано програмно-апаратні платформи, призначені для масивно-паралельних обчислень на відеоадаптерах. Описано принципи створення програм для платформи NVidia CUDA. Запропоновано методику роботи з динамічно розподіленою пам'яттю на основі шаблону «Стратегія» з використанням нотації UML. Відзначено взаємозв'язок шаблонів паралельного програмування із математичним апаратом систем алгоритмічних алгебр. Описано підхід до проектування алгоритмів на основі шаблону «Команда». Реалізовано програмний інтерфейс, на основі якого створено паралельну версію алгоритму Данцига.

Вступ

GPGPU – обчислення загального призначення з використанням графічних процесорів (GPU), які на сьогоднішній день є високопродуктивними багатоядерними процесорами, що характеризуються надвисокою обчислювальною потужністю та великою пропускною здатністю. Спочатку – спеціально призначені для комп'ютерної графіки і незручні для прикладного програмування, графічні процесори сьогодні є універсальними паралельними пристроями з підтримкою цілком доступних програмних інтерфейсів та стандартних мов програмування, таких як C та C++ [1, 2].

У графічних API повністю відсутня підтримка взаємодії між пікселями, що обробляються паралельно, які в принципі не дуже потрібно для графіки, але для обчислювальних задач є досить бажаним. Іншим суттєвим недоліком є відсутність підтримки операції типу *scatter* (розподілити, рис. 1). Найпростішим прикладом операцій такого типу є побудова гістограм за вхідними даними, коли кожен новий елемент призводить до зміни наперед невідомого компонента (чи компонентів) гістограми.

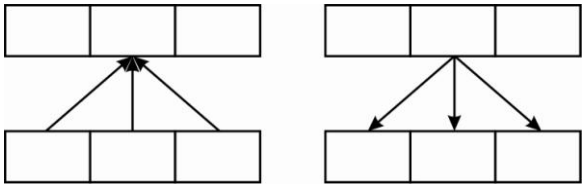

Рис. 1. Операції типу scatter

Це пов'язано з тим, що в графічних API шейдер може здійснювати запис тільки в наперед визначене місце, оскільки для фрагментного шейдера наперед визначається, який саме фрагмент він буде опрацьовувати, і він може записати лише значення даного фрагмента.

Ще одним «спадковим» недоліком є те, що розробка проходить одночасно двома мовами: для CPU та для GPU відповідно.

Всі вищенаведені обставини ускладнюють використання GPGPU і накладають ряд серйозних обмежень на алгоритми. Тому цілком природно виникла необхідність у створенні засобів розробки GPGPU програм, вільних від цих обмежень та орієнтованих на розв'язування складних задач.

Як такі засоби виступають NVidia CUDA, OpenCL [3] та DX11 Compute Shaders [4].

OpenCL (Open Computing Language) – відкритий стандарт для паралельного програмування гетерогенних систем [3], який є першим відкритим безкоштовним стандартом для кросплатформенного паралельного програмування сучасних процесорів персональних комп'ютерах, серверах і портативних та вбудованих пристроях. OpenCL значно збільшує швидкість і гнучкість для широкого спектру застосувань (від ігор і розваг до наукових і медичних програм).

OpenCL створюється групою багатьох передових компаній і установ під назвою Khronos Group.

У фреймворк OpenCL входять мова програмування, яка базується на стандарті C99, та інтерфейс програмування (API), що забезпечує паралельність на рівні інструкцій та на рівні даних і є реалізацією техніки GPGPU. OpenCL – повністю відкритий стандарт, його використання доступне на базі вільних ліцензій. Мета OpenCL полягає у тому, щоб доповнити OpenGl і OpenAL, які є відкритими галузевими стандартами для тривимірної комп'ютерної графіки і звуку.

DirectX Compute – технологія, створена компанією Microsoft на базі бібліотеки Direct3D. Для розв'язування задач використовуються шейдери, але є можливість використання специфічних для DirectX API.

NVidia CUDA. Запропонована компанією NVidia технологія CUDA помітно полегшує створення GPGPU-застосувань. Вона не використовує графічних API і вільна від обмежень, притаманних такого роду API [5, 6].

При цьому слід чітко розуміти принципову різницю між потоками на CPU та на GPU:

– потоки на GPU мають дуже низьку «собівартість» створення, керування та використання (контекст потоку мінімальний, всі регістри розподілені наперед);

– для ефективного використання GPU слід використовувати багато тисяч окремих потоків, тоді як для CPU достатньо 10 – 20 штук.

Зазвичай типова програма з використанням CUDA має наступний вигляд:

1) виділення пам'яті на GPU;
2) копіювання даних між CPU та GPU;
3) запуск ядра (технологія Fermi дозволяє запуск кількох ядер одночасно);
4) копіювання результатів даних між GPU та CPU;
5) звільнення виділеної пам'яті на GPU.

Для обробки масиву даних створюється багато потоків, і як наслідок, кожен потік обробляє один елемент вхідних даних. Усі ці потоки виконуються паралельно, потік може отримати інформацію про себе через набір змінних середовища.

Важливим моментом є те, що хоча даний підхід і є подібним на роботу із SIMD-моделлю, але є принципові відмінності (компанія NVidia використовує термін SIMT – Single Instruction Multiple Thread). Потоки розбиваються на групи по 32 штуки, які називаються ***варпами*** (warp). Тільки потоки в межах одного варпа виконуються фізично одночасно. Потоки з різних варпів можуть перебувати на різних стадіях виконання програми. При цьому керування варпами прозоро для програміста виконується самим GPU. Розподіл потоків насправді відображає ще один із основних прийомів використання CUDA–декомпозицію вихідної задачі на окремі підзадачі, які можуть бути розв'язані незалежно одна від одної.

У зв'язку з усім вищезгаданим метою роботи є створення методики проектування застосувань для програмно-апаратної платформи CUDA на основі існуючого промислового інструментарію та застосування UML і шаблонів паралельного програмування.

Реалізація підходу до виділення пам'яті на базі шаблону «Стратегія»

Робота з динамічно розподіленою пам'яттю. При використанні динамічно розподіленої пам'яті у програмах, створених мовами C та C++, зазвичай використовується один з методів виділення пам'яті (memory allocation) – функція `malloc/free` (C) чи `new/delete` (C++). Команда new забезпечує виклик конструкторів за замовчуванням для кожного екземпляра класу, під який виділяється пам'ять. Після завершення роботи з даними виділену пам'ять слід звільнити відповідно командою `free` або `delete`. Якщо цього не зробити, виникне класична проблема – протікання пам'яті (memory leak) – фактично пам'ять не звільнюється, а тому не може бути заново розподілена диспетчером [7, 8].

Технологія NVidia CUDA дозволяє напряму використовувати ресурси відеокарти за допомогою набору API-функцій, які зокрема, забезпечують і доступ до відеопам'яті. Щоб виділити необхідний обсяг слід скористатися функцією `cudaMalloc` (чи однією з подібних). Аналогічно,

пам'ять звільняється командою `cudaFree`. Доступ до даних, розміщених на відео карті 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 обсяг пам'яті та скористатися 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 вказаних проблем дозволяють так звані «розумні» вказівники (smart pointers). Як приклад можна навести `shared_ptr` із стандартної бібліотеки STL чи 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 про створення деякого уніфікованого інтерфейсу розумного вказівника, який дозволив би однаково працювати з даними, незалежно від їх розміщення, приховуючи реалізацію низькорівневих операцій, які стали б прозорими для користувача (тут і надалі, користувач – розробник, що використовує можливості створеного вказівника).

Можна сформувати низку вимог до створюваного інтерфейсу (надалі – UnifiedPtr).

Автоматизація виділення та звільнення пам'яті, що дозволить наступні операції:

```
// Вказівник для виділення пам'яті в «купі» для 10-ти цілих чисел
HeapUnifiedPtr<int> h(10);
// Вказівник для виділення пам'яті на відеокарті для 10-ти цілих чисел
DeviceUnifiedPtr<int> d (10);
// Власне, виділення пам'яті
h.Allocate();
d.Allocate();
// Явне звільнення пам'яті в змінній h,
h.Deallocate();
```

Реалізація прозорого копіювання даних з використанням різних типів UnifiedPtr, що зробить можливим наступні присвоювання:

```
HeapUnifiedPtr h1, h2;
DeviceUnifiedPtr d1, d2;
// Присвоювання працюють коректно
h2 = h1;   // Host  =>  Host
d1 = h2;   // Host  =>  Device
d2 = d1;   // Device  =>  Device
h1 = d2;   // Device  =>  Host
```

Як описано вище, на UnifiedPtr накладається низка вимог, сформованих зі сторони функціоналу. Таким чином постає проблема настроювання компонента. Нехай, наприклад, треба створити вказівник для виділення динамічної області з глибоким копіюванням для одно поточного середовища, а також вказівник для виділення пам'яті на відео карті з підрахунком посилань для багато поточного середовища.

Виникає проблема створення досить гнучкого інтерфейсу, який міг би дати користувачу можливість настроювання окремих компонентів, і водночас мав розумні розміри.

Реалізація всього можливого функціоналу під оболонкою деякого універсального інтерфейсу не є вдалим вибором, оскільки призведе до створення громіздких компонентів (яскравим прикладом в даному випадку є клас `std::string` із стандартної бібліотеки C++, переповнений функціоналом та дублюванням операцій). Програми, що будуть використовувати такі класи працюють набагато повільніше, аніж аналогічні програми, розроблені самотужки [2].

Але однією із 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 напи-

сання синтаксично правильного коду, який може зруйнувати основні зв'язки всередині бібліотеки.

Один з 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 і налагоджену поведінку.

В такому випадку логічно очікувати появи численних реалізацій типу:

- `HeapDeepCopySingleThreadPtr`
- `DeviceRefCountMultiThreadPtr`

Для такого розв'язку проблеми характерним є різкий ріст кількості різних варіантів проектних рішень. Тільки два вищезгаданих класи можуть легко призвести о 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 – наяву:

– масштабованість даного рішення практично нульова. Щоб додати новий тип даних потрібно повністю його описувати;

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

– 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 в плачевний стан.

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.

Шаблон «Стратегія» (Strategy)

Стратегія – шаблон поведінки об'єктів [5, 7].

Шаблон визначає сімейство алгоритмів, інкапсулює їх та робить взаємозамінними. Стратегія дозволяє змінювати алгоритми незалежно від клієнтів, що їх використовують.

Учасники (рис. 2):

Strategy – Стратегія.

Оголошує спільний для всіх підтримуваних алгоритмів інтерфейс.

Concrete Strategy – конкретна стратегія.

Реалізує алгоритм, який використовує інтерфейс, оголошений стратегією.

Context – контекст.

Конфігурується об'єктом класу Concrete Strategy;

зберігає вказівник на Strategy;

може оголосити інтерфейс, який дозволить стратегії отримати доступ до даних контексту.

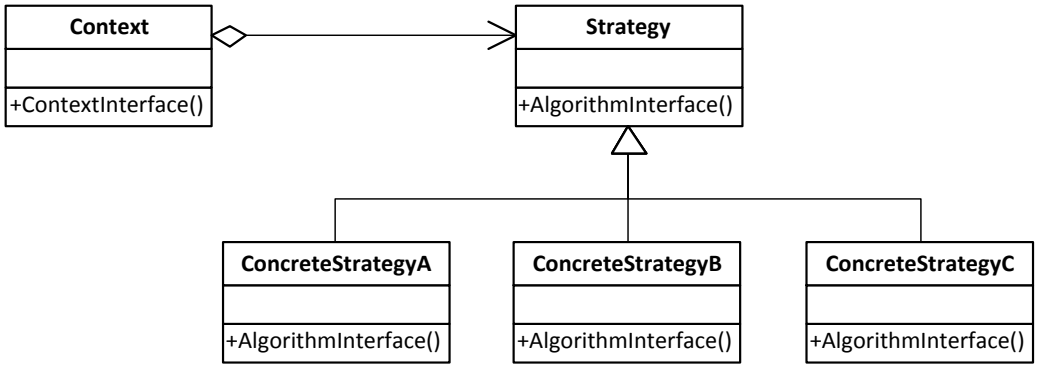

Рис. 2. Діаграма класів, що відображає шаблон «Стратегія»

Результати застосування шаблону «Стратегія» (позитивні та негативні):

– спорідненість алгоритмів;

– альтернатива породженню підкласів;

– за допомогою стратегій можна уникнути використання операторів розгалуження;

– вибір реалізації на етапі ***виконання/компіляції;***

– клієнти повинні «знати» про різні стратегії;

– обмін інформацією між клієнтом та стратегією;

– збільшення кількості об'єктів;

– реалізація стратегій, як шаблонних параметрів.

В C++ для конфігурації стратегій можна використати шаблонні параметри. Клас конфігурується стратегією у момент інстанціювання. При такому використанні стратегій відпадає необхідність у абстрактному класі для визначення інтерфейсу. Крім того передача у вигляді шаблона дозволяє статично зв'язати стратегію з контекстом, підвищуючи таким чином ефективність програми.

У роботі запропоновано використання класів стратегій для настроювання функціоналу на етапі компіляції програми (інформація про стратегії передається у вигляді шаблонних параметрів). Тип успадковує свої стратегії, успадковуючи таким чином необхідний функціонал [7].

Такий підхід призводить до автоматичної генерації компілятором необхідних ієрархій класів. Перевагою є відсутність динамічного поліморфізму, а, як наслідок, – пов'язаних з цим затримок.

Декомпозиція UnifiedPtr на стратегії

Стратегія виділення пам'яті. Призначення цієї стратегії – динамічне виділення пам'яті. Таким чином вона постачає набір функцій:

```
void * AllocateMemory(size_t)
void * AllocateMemory(size_t,
size_t)
void DeallocateMemory(void*)
```

для централізації інтерфейсу всі стратегії виділення пам'яті успадковуються від класу Allocator (фактично, це не обов'язково, достатньо реалізувати дві вищеописані функції.

Додатково виділено дві групи розподілювачів пам'яті (рис. 3): лінійні розподілювачі (пам'ять виділяється одним нероздільним шматком в адресному просторі, всі дані розміщені послідовно) та матричні розподілювачі (при виділенні на відеоадаптері пам'яті під матрицю кожен рядок вирівнюється за адресою, кратною 64 або 128 байт, що значно пришвидшує роботу з глобальною пам'яттю).

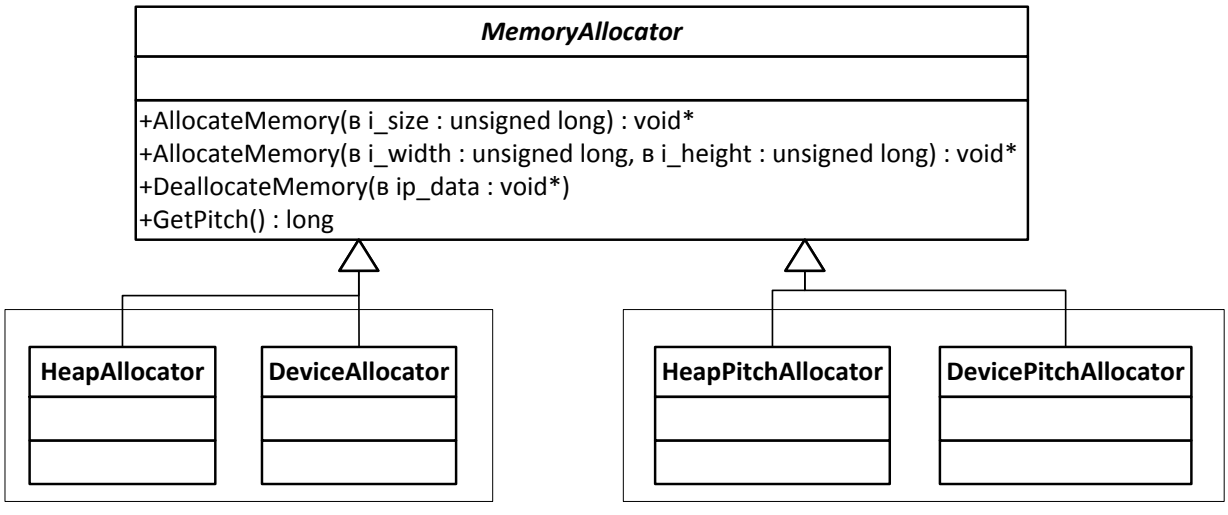

Рис. 3. Діаграма класів, що відображає стратегію виділення пам'яті

Стратегія копіювання даних. Стратегія копіювання даних тісно пов'язана з методом виділення пам'яті та стратегією зберігання інформації.

Для створення UnifiedPtr доцільно створити деякий універсальний клас, який реалізуватиме глибоке копіювання даних. Необхідну функцію компілятор може вибирати сам на основі інформації, закладеної в шаблонні параметри. Таким чином мовою C++ можна записати наступну сигнатуру для функції копіювання:

```
template
   <
  Class Allocator1,
  Class Allocator2
  >
void Copy(void * ip_src, void *
ip_dest, size_t i_size)
```

Такий вигляд дозволяє уникнути операторів розгалуження у тексті підпрограми і полегшує копіювання даних між різними пристроями.

Стратегії зберігання даних. Стратегія зберігання реалізує логіку зберігання та копіювання даних.

Розглянемо існуючі стратегії та можливість їх застосування до UnifiedPtr.

Стратегія глибокого копіювання (DeepCopyStorage Policy).

Кожен екземпляр класу містить вказівник на свою ділянку пам'яті (рис. 4). При копіюванні відбувається повне (глибоке) копіювання даних. При знищенні об'єкта пам'ять звільняється. До плюсів такого підходу можна віднести відносну простоту реалізації. Суттєвим мінусом є значні накладні витрати на копіювання та зберігання інформації. Стратегію показано на рисунку.

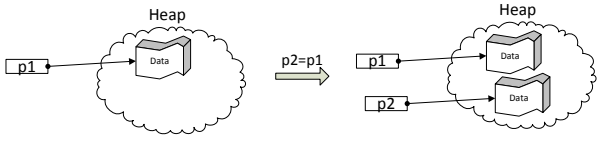

Рис. 4. Схематичне зображення глибокого копіювання даних

Для UnifiedPtr дана стратегія є цілком застосовною, оскільки не накладає жодних обмежень, що легко реалізується через клас CopyFunctor.

Стратегія володіння (Ownership Strategy).

Ресурсом (у даному випадку – пам'яттю) володіє лише один об'єкт (рис. 5). При присвоювання відбувається передача права власності. Пам'ять звільнюється лише один раз при знищенні екземпляра, який володіє ресурсом у поточний момент часу.

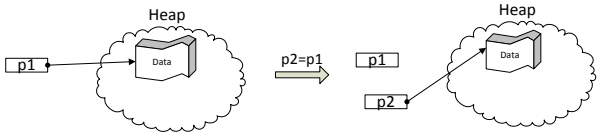

Рис. 5. Схематичне зображення стратегії володіння даними

Така стратегія значно зменшує затрати, пов'язані з копіюванням інформації. Але, оскільки в нашому випадку, спосіб звільнення пам'яті жорстко зв'язаний із способом її виділення і задається ще на етапі компіляції, то копіювання можливе лише в межах одного пристрою.

Стратегія підрахунку посилань (Reference Count Strategy)

Окрім власне корисної інформації в динамічній пам'яті зберігається також лічильник посилань (рис. 6). Пам'ять звільняється тоді, коли лічильник стає рівний нулю. Таким чином копіювання призводить до збільшення лічильника на одиницю.

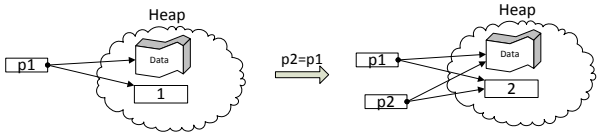

Рис. 6. Схематичне зображення стратегії підрахунку посилань

Для використання з UnifiedPtr пропонується провести деяку модифікацію. Будемо зберігати два лічильники посилань (рис. 7). При копіюванні між CPU та відеокартою виконується глибоке копіювання, а після чого можна використовувати класичний підхід підрахунку посилань.

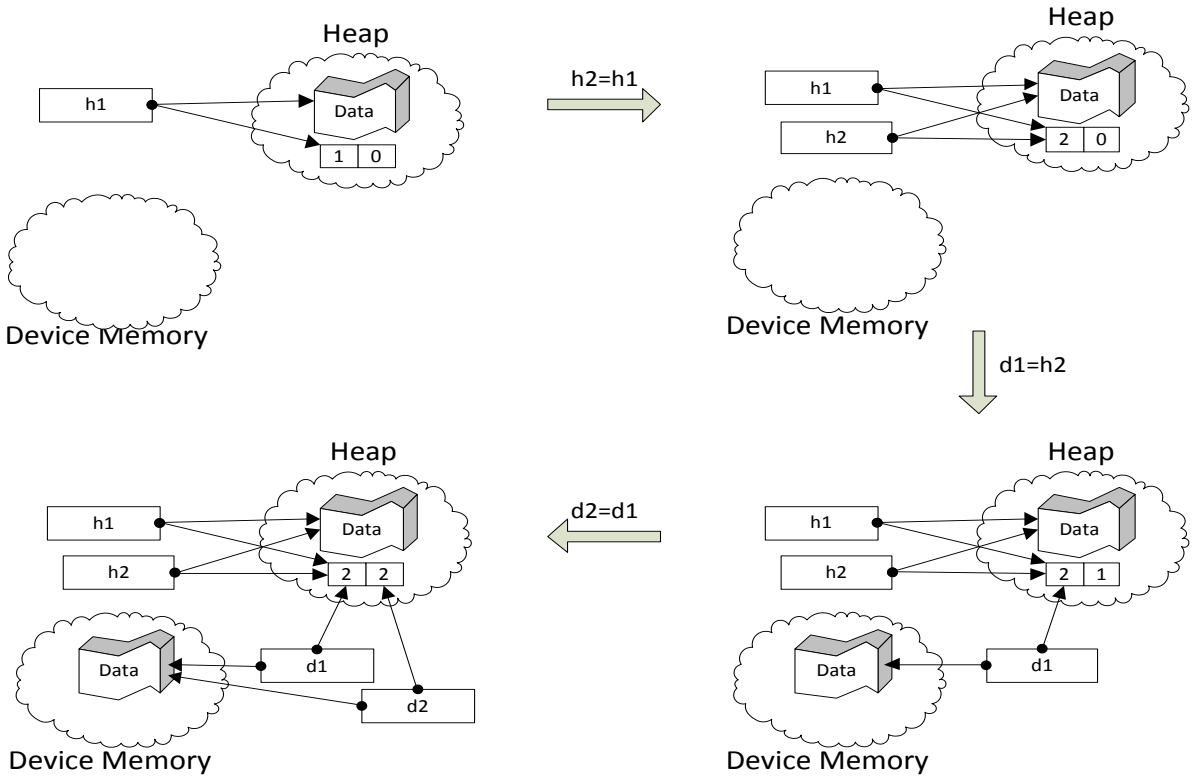

Рис. 7. Стратегія підрахунку посилань, адаптована для CUDA

Інкапсуляція операцій в об'єктах для декомпозиції задачі

Шаблон «Команда». «Команда» (інші назви – оператор, функтор), як і «Стратегія» – шаблон поведінки об'єктів. Він дозволяє інкапсулювати запит як об'єкт, дозволяючи тим самим задавати параметри клієнта для обробки відповідних запитів, ставити запити в чергу, протоколювати і підтримувати операцію «Відмінити» [5, 7].

Інколи необхідно посилати об'єктам запити, нічого не знаючи про те, виконання якої операції необхідно та хто є адресатом запиту. Шаблон проектування «Команда» дозволяє надсилати запити деяким об'єктам програми, перетворюючи сам запит в об'єкт. Його можна передавати та зберігати як і будь-який інший об'єкт. В основі лежить абстрактний клас, який оголошує інтерфейс для використання операції. У найпростішій формі цей інтерфейс містить єдину функцію Execute (виконати операцію). Конкретні підкласи містять інформацію про те, хто є адресатом, і що саме треба виконувати. У адресата є набір даних, необхідних для успішного виконання операції.

Учасники (згідно з рис. 8Рис. 8):

Command – команда.

Оголошує інтерфейс для виконання операції.

ConcreteCommand – конкретна команда.

Визначає зв'язок між об'єктом адресатом та дією;

реалізує операцію Execute шляхом виклику відповідних операцій.

Client – клієнт.

Створює об'єкт класу ConcreteCommand та встановлює його одержувача.

Invoker – ініціатор.

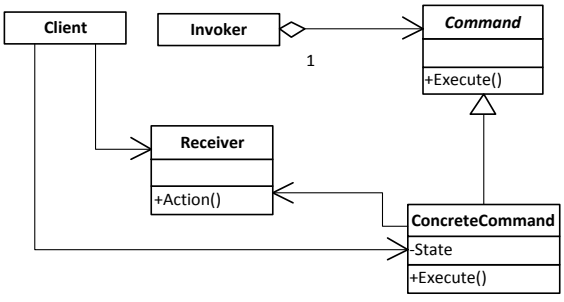

Рис. 8. Структура шаблону «Команда»

Звертається до команди для виконання запиту.

Receiver – одержувач запиту.

Містить інформацію про способи виконання операції. У ролі одержувача може виступати будь-який клас.

У результаті використання «Команди» розривається зв'язок між об'єктом, що ініціює виконання операції та об'єктом, що містить операцію, необхідну для її виконання. Оскільки команди – це справжні об'єкти, то допускаються довільні маніпуляції з їх участю, розширення, тощо. Як наслідок із простих команд можна збирати складніші, об'єднуючи їх в ланцюжки операцій. Для додавання нових операцій не потрібно змінювати вже існуючі класи.

Зв'язок шаблону «Команда» та систем алгоритмічних алгебр (САА) В.М. Глушкова

САА – потужний математичний апарат для формалізації алгоритмів, запропонований В.М. Глушковим. Основною перевагою САА є спрямування на проектування схем алгоритмів і можливість представлення довільного алгоритму у вигляді алгебраїчної формули. Розвинені методи формальних перетворень САА дозволяють проводити оптимізацію схеми, абстрагуючись від конкретної програмної реалізації. Крім того САА цілком відповідають основним принципам структурованого програмування [9].

Наступним кроком було розширення математичного апарату на випадок багатопроцесорних архітектур, за рахунок введення додаткових операцій, спрямованих на паралельну обробку даних. Введення нових операцій призвело до виникнення модифікованих САА (САА-М).

Як відомо САА оперують поняттям операторів, пов'язуючи їх у цілісні алгоритми з допомогою формульного запису. Використовуючи шаблон «Команда» для представлення таких «примітивних» алгоритмів можна реалізовувати схеми, подані у вигляді САА, створюючи макрокоманди на рівні об'єктів (окремі оператори можна об'єднувати у послідовності). Таким чином, САА виконують роль абстракції від конкретної платформи, зосереджуючись на особливостях роботи алгоритму.

Перенесення схеми на конкретну робочу платформу покладається на окремі оператори, які виступають деяким проміжним етапом в даному випадку.

Як приклад, розглянемо дуже просту схему, записану з використанням нотації САА:

$$Scheme = Step1 \times \underset{Cond}{(} Step2 \vee Step3) \qquad (1)$$

Наступна ієрархія класів дозволяє представити дану схему. На рис. 9 нижні два рівні представляють абстракції, призначені для уніфікації роботи з класами. На вищому рівні ієрархії (внизу рисунка) різними кольорами показано по дві конкретні реалізації кожного оператора. Крім того ніщо не забороняє змішувати різнотипні реалізації, дозволяючи таким чином отримати вісім можливих програмних представлень наведеної схеми.

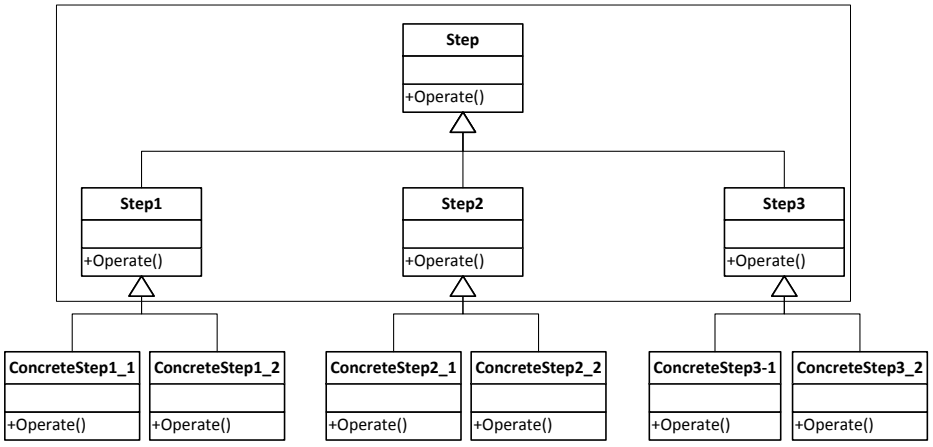

Рис. 9. Ієрархія класів для реалізації простої САА схеми

Тоді програмний код буде мати наступний вигляд:

```
Step *step1, *step2, *step3;
// Ініціалізація даних та команд
...
// Виконання схеми
Step1->Operate();
If (Condition)
Step2->Operate();
else
Step3->Operate();
```

Крім того, умовний оператор також можна представити у вигляді команди, повністю «автоматизувавши» таким чином виконання алгоритму. Можна добитися навіть зовнішньо настроювання алгоритму з допомогою файлу конфігурації чи набору команд користувача. Тобто на вхід програми поступає САА схема, за якою збирається настроєний користувачем алгоритм. Слід зауважити, що такий підхід потребує чіткої декомпозиції задачі на окремі підзадачі (не обов'язково примітивні).

Таким чином, можна використати схеми, записані з допомогою САА для реалізації алгоритмів допомогою технології NVidia CUDA. Для цього слід використати механізм шаблонів C++:

```
template<class Operation>
__global__ void Kernel(Operation
i_operation)
  {
  i_operation.Operate();
  }
```

Вищенаведена функція – узагальнене ядро CUDA, яке абсолютно не залежить від реалізації оператора. Функція `Operate()` в свою чергу є тілом функтора і може бути як віртуальною, так і статичною.

Програмна реалізація стратегій

Стратегія виділення пам'яті. Для виділення пам'яті використовується клас Allocator. Він об'єднує дві функції – виділення та звільнення пам'яті. В роботі реалізовано чотири підкласи:

`HeapAllocator` – стандартне виділення динамічної пам'яті можливостями мови C++. Власне в програмі використовується метод виділення malloc/free, що дозволяє уникнути необхідності в конструкторі за замовчуванням, а також деяких накладних витрат, пов'язаних із початковою ініціалізацією даних;

`DeviceAllocator` – виділення пам'яті на відео карті з використанням інтерфейсу CUDA;

`PitchDeviceAllocator` – виділення пам'яті на відео карті з використанням інтерфейсу CUDA з використанням автоматичного вирівнювання даних в пам'яті, що призводить до покращення швидкодії при читанні даних;

`PitchHeapAllocator` – відповідний розподілювач пам'яті для центрального процесора. Оскільки вирівнювання не є настільки важливим у даному випадку, то відбувається просто емуляція функціоналу.

У подальшому клас `Allocator` може бути легко розширено. Наприклад, можна реалізувати роботу з текстурною пам'яттю відеокарти, розміщення даних у стеку програми, тощо.

Стратегія копіювання даних. Для забезпечення легкого і прозорого копіювання інформації створено клас `CopyFunctor`, який має один статичний метод `Copy`. Для інстанціювання класу використовується інформація про два класи типу `Allocator`:

```
template
  <
  class AllocationPolicy1,
  class AllocationPolicy2
  >
class CopyFunctor
```

У залежності від вхідної комбінації типів компілятор автоматично вибирає необхідну функцію. На даному етапі представлено 4 перезавантажених методи для копіювання типу:

host – host,

host – device,

device – device,

device – host,

які відповідають всім можливим комбінаціям нащадків класу `Allocator`.

Стратегія зберігання даних. В роботі реалізовано вищеописані типи стратегій зберігання даних. Для уніфікації коду та уникнення повторювань всі стратегії у свою чергу є нащадками неабстрактного базового класу `StoragePolicy`. Розглянемо його інтерфейсну частину.

```
  template
    <
    typename T,
    class AllocationPolicy
    >
  class Storage : public
AllocationPolicy
    {
    public:
      void SetSize(size_t i_size);
      size_t GetSize() const;
      bool Empty() const;
      void Allocate();
      void Deallocate();
      T* GetImpl();
    };
```

Стратегія реалізує загальні функції для задання розміру необхідного буфера, виділення та звільнення пам'яті.

Власне функція GetImpl() забезпечує прямий доступ до буфера пам'яті, покладаючи таким чином відповідальність за роботу з буфером на користувача.

У роботі реалізовано програмно три вищеописані стратегії зберігання даних. У нотації UML ми отримаємо наступну діаграму (рис. 10).

Оскільки базова стратегія вже реалізує необхідні функції для роботи з пам'яттю, класи-нащадки мають реалізувати лише ту логіку, яка специфікується даним типом, делегуючи виконання більш загальних операцій базовому класу, який за необхідності може передати їх далі по ієрархії.

Таким чином стратегії зберігання реалізовують лише набір необхідних вузькоспеціалізованих конструкторів та операторів присвоювання. Вони власне інкапсулюють основні відмінності між класами, які між іншим, також встановлюються компілятором на основі інформації про базові типи даних.

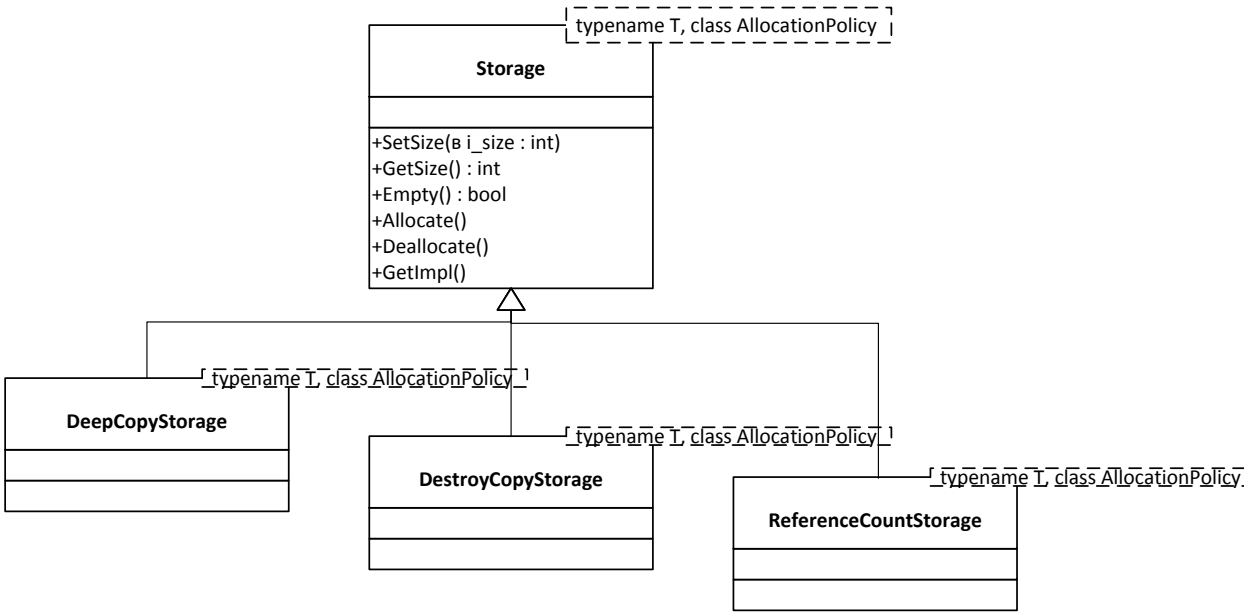

Рис. 10. Діаграма, що відображає ієрархію стратегій зберігання даних

Опис інтерфейсу UnifiedPtr

На основі описаних стратегій реалізовано клас `UnifiedPtr`, який дозволяє однаково працювати з даними розміщеними як на відео карті, так і в оперативній пам'яті комп'ютера. Далі наведено заголовок класу:

```
  template
    <
    typename T,
    class AllocationPolicy,
    template <class, class> class
StoragePolicy
    >
    class UnifiedPtr : public
StoragePolicy<T, AllocationPolicy>
```

Як видно, клас повністю специфікується трьома шаблонними типами, успадковуючи функції стратегій. Таким чином, для настроювання поведінки класу достатньо змінити будь-яку з стратегій, а для розширення слід просто описати відповідну частину функціоналу. Наприклад, для додавання функцій, що забезпечать правильну роботу класу в багатопоточному середовищі процесора можна створити нову стратегію зберігання даних `MultithreadingStoragePolicy` і передати її як шаблонний параметр. Таким чином значну роботу виконує компілятор, який при інстанціюванні шаблонного типу генерує відповідні ієрархії класів.

На діаграмі рис. 11 показано реальну ієрархію класів, що створюється компілятором. Як бачимо, на основі вже реалізованих типів даних можна створити дванадцять нових типів, які легко зробити взаємозамінними, настроювати та підтримувати.

Створений програмний інтерфейс успішно пройшов низку перевірок основного функціоналу, для чого було використано бібліотеку UnitTest++.

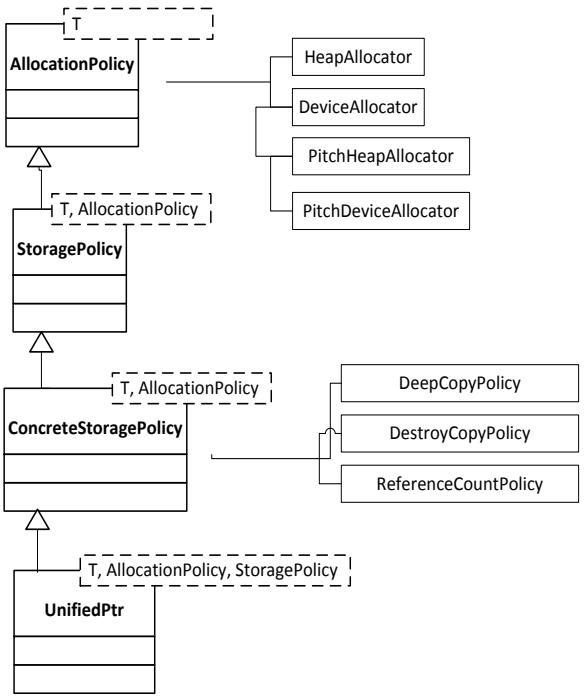

Рис. 11. Схема класу UnifiedPtr

Реалізація базового класу-оператора

Шаблон «Команда» дозволяє створити узагальнене ядро CUDA. Для можливості вибору відповідного алгоритму можна застосувати принципи успадкування та поліморфізму.

У роботі запропоновано підхід, що ґрунтується на можливостях статичного поліморфізму, а саме шаблонних параметрах C++. Прийом, що застосовується носить назву CRTP – рекурентний шаблонний паттерн, який дозволяє використовувати клас нащадок при інстанціюванні батьківського класу. Наприклад,

```
template<class TDerived>
class Base {};
class Derived : public
Base<Derived> {};
```

Як не дивно, така конструкція не викликає зациклювання компілятора (дозволено стандартом C++99) і водночас відкриває цікаві можливості. Зокрема, в роботі використано підхід, що дозволяє емулювати віртуальність функції, визначаючи необхідну адресу на етапі компіляції.

На рис. 12 наведено відповідну діаграму UML.

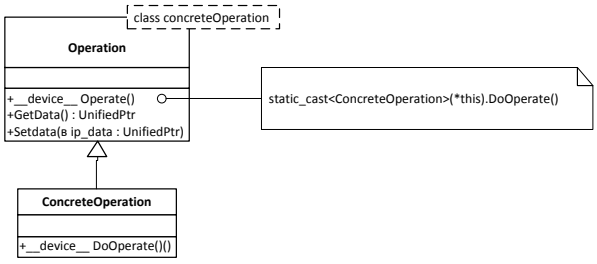

Рис. 12. Схема класу-оператора з підтримкою статичного поліморфізму

Як видно з рисунка, всередині класу застосовується складне приведення типів та виклик невіртуальної функції. Таким чином клас-нащадок зобов'язаний реалізувати функцію `_DoOperate()`.

Як вже було сказано, перевагою такого підходу є, безумовно, краща швидкодія. До мінусів можна віднести необхідність перекомпіляції ядра при використанні нового оператора. Втім цього недоліку можна уникнути створивши набір ядер, інстанційованих різними операторами.

У роботі створено власне базовий каркас для оператора, який слід успадковувати при створенні нових алгоритмів. Як перспективу розвитку слід зауважити можливість об'єднання команд в ланцюжки, створення об'єктів-умов, об'єктів-циклів і т. д. Як вже було сказано, це дає можливість перенести алгоритми, записані за допомогою абстрактних засобів (САА) на конкретну програмну реалізацію.

Приклад реалізації алгоритму Данцига для платформи CUDA

Паралельна схема алгоритму в нотації САА. Одним із найефективніших методів пошуку всіх найкоротших шляхів у зваженому орієнтованому графі є алгоритм Данцига. Його складність становить $O(n^3)$, де n – кількість вершин графа [10].

Вихідними даними для алгоритму є матриця вагових коефіцієнтів. Ідея полягає у послідовному обчисленні з допомогою рекурентної процедури підматриць найкоротших шляхів D_m зростаючої розмірності m x m. Кожна така матриця, фактично, є матрицею найкоротших шляхів підграфа з вершинами від 0 до *m-1*.

Оскільки алгоритм має квадратичну складність, то він є перспективним з точки зору розпаралелювання. Розглянуто можливість розпаралелювання алгоритму для систем із спільною пам'яттю.

Паралельна версія алгоритму описується наступною САА-схемою:

$$
\begin{aligned}
Dancig = \\
= \underset{m>N}{\{} \left(\overset{2N}{\underset{l=1}{\vdots}} (PSub1(m,2N,l) \times B_{2N}(l)) \right) \times \\
\times \left(\overset{2N}{\underset{l=1}{\vdots}} (PSub2(m,2N,l) \times B_{2N}(l)) \right) \times \\
\times \left(\overset{2N}{\underset{l=1}{\vdots}} (PSub3(m,2N,l) \times B_{2N}(l)) \right) \times \\
\times (++m)\}.
\end{aligned}
\qquad (2)
$$

Завдяки використанню шаблонів можна застосувати дану схему для реалізації програми на платформі CUDA.

Результати тестування алгоритму

На рис. 13 показано графік для порівняння часових характеристик роботи алгоритму з використанням технології CUDA та з використанням багатопоточності у системі з чотирьохядерним процесором. Дані було отримано для вибірки графів з кількістю вершин від 100 до 1000.

Як видно з графіка, для графів низької розмірності (до 500 вершин) багатопоточна версія є виграшною за часом. При зростанні навантаження (збільшення кількості вершин) версія для відеоадаптера дає виграш, який поступово зростає з ростом розмірності графа. Для пошуку найкоротших шляхів для 1000 вершин необхідно близько 900 мс на відеоадаптері, що у 2.5 рази швидше за програму для CPU.

Такий хід графіків цілком пояснюється специфікою алгоритму Данцига, який працює з ітеративним набором

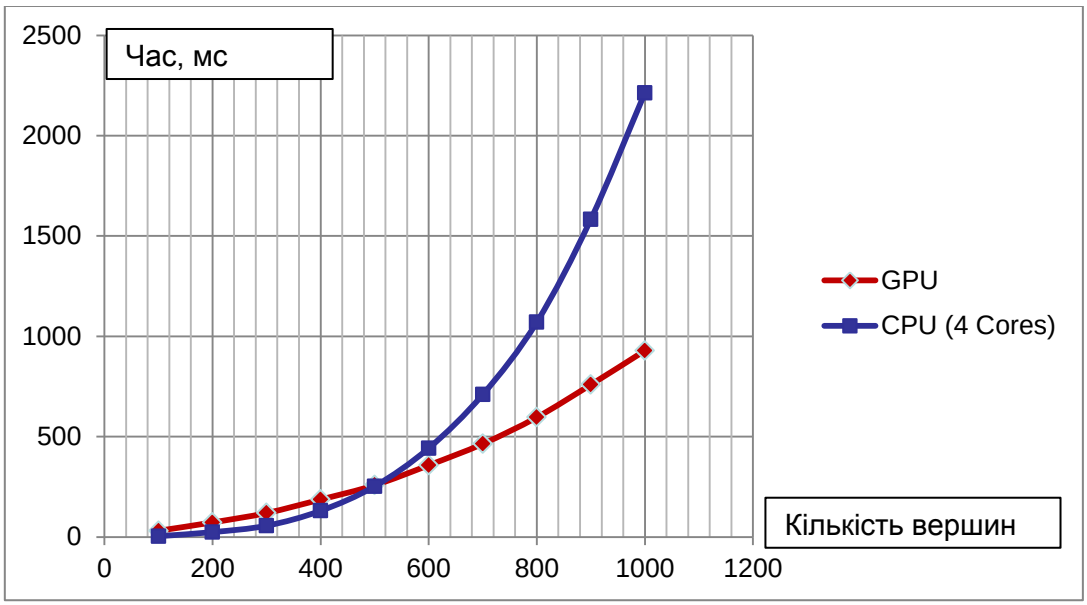

Рис. 13. Порівняння часових характеристик програм для GPU та CPU

матриць зростаючої розмірності (1x1, 2x2, 3x3 і т. д.). Таким чином для графів малої розмірності завантаження відеоадаптера не є оптимальним, отже його ресурс не використовується повністю. При переході до великих розмірностей графів більшу частину роботи алгоритму GPU повністю завантажений обчисленнями, що й зумовлює виграш у часі.

Як видно з графіка, час роботи алгоритму зростає повільніше для GPU ніж для CPU, що дає змогу зробити висновок про доцільність його використання для графів високої розмірності.

Висновки

У роботі запропоновано підхід до реалізації уніфікованого інтерфейсу для роботи із динамічно розподіленою пам'яттю (діаграма класів на рис. 12), який базується на використанні шаблону проектування «Стратегія». Здійснено розклад класу на основні стратегії та розглянуто можливість їх застосування для програм, призначених для платформи CUDA. Виділено наступні класи стратегій:

- стратегія виділення пам'яті;
- стратегія копіювання даних;
- стратегія зберігання даних.

Описано метод створення об'єктів-операторів на основі шаблону «Команда», який дозволяє реалізувати абстрактні схеми алгоритмів за рахунок використання можливостей статичного поліморфізму.

Запропоновані підходи представлено за допомогою UML-нотації, яка є універсальним засобом проектування як ієрархій класів, так і діаграм потоків, що відображають роботу програми.

Для використання описаних у роботі підходів створено бібліотеку мовою C++, яку використано для програмної реалізації алгоритму Данцига, представленого у вигляді абстрактної схеми за допомогою САА Глушкова. Алгоритм реалізований для відеоадаптера дає виграш у часових характеристиках приблизно у 2.5 рази порівняно з чотирьохядерним процесором Intel Core 2 Quad, що ефективну підтверджує застосовність запропонованого методу проектування програм та перспективність його розвитку.

1. *Боресков А.В., Харламов А.А.* Основы работы с технологией CUDA. – М.: ДМК Пресс, 2010. – 232 с.
2. *Sanders J., Kandrot E.* CUDA by Example: An Introduction to General-Purpose GPU Programming. – Addison-Wesley Professional, 2010. – 312 p.

3. *The* Khronos Group: Open Standards, Royalty Free, Dynamic Media Techno-logies. http://www.khronos.org/opencl/
4. *Direct* Compute. http://www.nvidia.com/object/cuda_directcompute.html
5. *Александреску А.* Современное проектирование на C++. Серия C++ in Depth.: Пер. с англ. – М.: Издательский дом «Вильямс», 2008. – 336 с.
6. *Погорелый С.Д., Бойко Ю.В., Трибрат М.И., Грязнов Д.Б.* Анализ методов повышения производительности компьютеров с использованием графических процессоров и апаратно-програмной платформы CUDA // Математичні машини і системи. – 2010. – № 1. – С. 40–54.
7. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.* Приемы объекто-ориентированого проектирования // Паттерны проектирования. – СПб: Питер, 2009. – 366 с.
8. *Саттер Г.* Решение сложных задач на C++. Серия C++ in Depth.: Пер. с англ. – М.: Издательский дом «Вильямс», 2008. – 400 с.
9. *Ющенко Е.Л., Цейтлин Г.Е., Грицай В.П. и др.* Многоуровневое структурное проектирование программ // Теоретические основы, инструментарий.– М.: Финансы и статистика, 1989. – 342 с.
10. *Майника Э.* Алгоритмы оптимизации на сетях и графах. – М.: Мир, 1981. – 324 с.

Одержано 22.03.2013

Про авторів:

Погорілий Сергій Демьянович,
доктор технічних наук,
професор,

Верещинський Олег Андрійович,
аспірант другого року навчання.

Місце роботи авторів:

Київський національний університет
імені Тараса Шевченка,
01601, вул. Володимирська, 60,
м. Київ, Україна.
Тел. +38 (093) 080 7975.
E-mail: sdp@univ.net.ua,
oleg.vereshchynsky@gmail.com