УДК 004.4

ПІДХОДИ ДО ФОРМАЛІЗАЦІЇ ПРОЕКТУВАННЯ ЗАСТОСУВАНЬ В ТЕХНОЛОГІЇ GPGPU

С.Д. ПОГОРІЛИЙ, О.А. ВЕРЕЩИНСЬКИЙ, Д.Ю. ВІТЕЛЬ

Обґрунтовано необхідність створення формалізованих методів проектування алгоритмів, їх програмних реалізацій та дослідження тонкої інформаційної структури програм для систем з масовим паралелізмом, які містять відеоадаптери. Запропоновано та обґрунтовано застосування чотирьох підходів до формалізації проектування застосувань у технології GPGPU: алгеброалгоритмічного, з використанням кольорових мереж Петрі, з використанням об'єктно-орієнтованих шаблонів програмування та з використанням поширених методів функціонального програмування. Проаналізовано переваги застосування модифікованої системи алгоритмічних алгебр Глушкова (САА-М) та алгебри реального часу (RTPA) до розробки GPGPU-застосувань. Розроблено модифікації поширених шаблонів об'єктно-орієнтованого програмування, що враховують специфіку роботи відеоадаптера. Запропоновано декларативний спосіб визначення GPU-обчислення з використанням шаблону MapReduce та функціональних мов програмування. Надано рекомендації щодо практичного використання цих підходів.

ВСТУП

Основною особливістю новітніх графічних відеоадаптерів, які використовують графічні процесори (Graphical Processing Unit, GPU), є наявність набору потокових мультипроцесорів, що використовувалися раніше лише в алгоритмах і задачах, пов'язаних з обробкою графічних зображень. Технологія обчислень загального призначення на графічних процесорах (GPGPU [1–2]) ґрунтується на використанні великої кількості процесорів GPU, що працюють паралельно, для обробки даних за допомогою алгоритмів загального призначення (не обов'язково пов'язаних з обробкою зображень).

На сьогодні найбільш поширені такі архітектури графічних відеоадаптерів фірми NVidia: Tesla, Fermi, Kepler, Maxwell та Pascal. Слід зазначити, що фірма NVidia, крім того, має серію відеоадаптерів Tesla, орієнтованих на використання у високопродуктивних (кластерних) обчисленнях [3–5]. Ці відеоадаптери позбавлені деяких специфічних для графіки функцій і широко застосовуються у науковій сфері. Найпотужніший на цей час відеоадаптер — Tesla K20 — також спроектований за Kepler-архітектурою.

Застосування відеоадаптерів суттєво ускладнює архітектуру сучасних комп'ютерних систем із масовим паралелізмом для проведення обчислень

загального призначення за рахунок:

- створення додаткового рівня паралелізму — рівня відеоадаптерів;
- присутності значної кількості низькорівневих операцій, які залежать від архітектури і конкретної моделі відеоадаптера;
- складної структури системи пам'яті відеоадаптера.

Складність процесу створення застосувань для таких систем вимагає необхідності розробки формалізованих методів проектування алгоритмів, їх програмних імплементацій та дослідження тонкої інформаційної структури програм. Авторами запропоновано чотири підходи до формалізації проектування застосувань в технології GPGPU.

Мета роботи — аналіз підходів проектування та розробки GPGPU-застосувань з використанням математичного апарату мереж Петрі, модифікованої системи алгоритмічних алгебр Глушкова, алгебри реального часу, шаблонів об'єктно-орієнтованого та функціонального програмування

ЗАСТОСУВАННЯ АЛГЕБРОАЛГОРИТМІЧНОГО ПІДХОДУ

Алгоритмічний етап проектування є початковим та найменш технологічно забезпеченим. Але він впливає на подальший процес розробки та визначає її успіх в цілому. Інструментальні засоби цього етапу проектування мають забезпечувати:

- ефективні засоби опису алгоритмів для систем з масовим паралелізмом;
- можливість формальних еквівалентних перетворень алгоритмів з метою їх досліджень та аналізу;
- засоби опису темпоральних та асинхронних фрагментів алгоритмів;
- можливість генерації асоційованих із досліджуваними алгоритмами програм для різних парадигм паралельного програмування;
- можливість моделювання і оцінки часових характеристик роботи алгоритмів до їх використання.

Ефективним методом алгоритмічного проектування і дослідження паралельних алгоритмів є використання алгебр алгоритмів, які дозволяють сформувати формули (схеми) алгоритмів, що залежать від різних параметрів, якими можуть виступати програмно-апаратні платформи, парадигми паралельного програмування тощо.

Такі алгебри дозволяють виконувати еквівалентні трансформації схем паралельних алгоритмів, що ефективно застосовується для подальшого пошуку оптимальних реалізацій.

В роботах [6–8] авторами пропонується і обґрунтовується застосування таких алгебр алгоритмів як:

- модифікована система алгоритмічних алгебр (САА-М) Глушкова;
- алгебра реального часу RTPA (Real Time Process Algebra) [9–10].

Особливості сигнатури операцій кожної із цих алгебр дозволяють зручно записувати схеми алгоритмів для комп'ютерних систем із масовим паралелізмом, які містять у вузлах відеоадаптери, систем реального часу та великих розподілених систем.

RTPA-алгебри [7] створено для нотації алгоритмів в процесі проектування. Основні їх особливості такі:

- можливість специфікації алгоритмів функціонування відкритих систем;
- наявність в сигнатурі операцій можливості обробки асинхронних подій;
- зручність проектування алгоритмічного забезпечення складних розподілених систем;
- наявність операцій опису процесів реального часу.

У [8] проілюстровано проведення ланцюжка еквівалентних трансформацій у CAA-M схеми алгоритму транспорту даних у глобальній мережі (на основі алгоритму Данцига) і сформовано в тому числі дві еквівалентні його схеми. Інтерпретація цих схем така. Перша схема відповідає ситуації з відносно невеликим навантаженням обчислювальних гілок і значною кількістю операції синхронізації після кожної фази обчислень. Друга схема навпаки значною мірою «навантажує» обчислювальні гілки, але суттєво скорочує кількість операцій синхронізації. Врешті-решт продуктивність другої схеми виявляється суттєво вищою.

ЗАСТОСУВАННЯ МЕРЕЖ ПЕТРІ

Інший підхід до моделювання роботи GPGPU-систем надає математичний апарат мереж Петрі [11, 12]. До його переваг належать можливості: проводити темпоральне моделювання, знаходити тупикові ситуації (deadlocks, lifelocks), представляти складні системи у вигляді набору взаємопов'язаних моделей (ієрархічні мережі).

Підхід мереж Петрі висуває наступні задачі:

- Створення моделей роботи сучасних відеоадаптерів, а саме основних принципів їх функціонування в процесі обчислення GPGPU-алгоритмів. (Такі моделі нададуть можливість враховувати обмеження цільової архітектури і формувати більш ефективні застосування на етапі проектування).
- Створення моделей основних шаблонів, що використовуються у розробці GPGPU-застосувань. До останніх належить широко застосовний шаблон MapReduce та шаблони доступу до пам'яті.

Математичний апарат мереж Петрі не накладає жодних припущень відносно предметної області, до якої він застосовується. Це, у свою чергу, надає свободу у процесі створення мереж, проте має і негативний фактор: як результат отримані моделі можуть бути неоптимальними за низкою критеріїв, складними і не ефективними. Тому було визначено набір правил побудови мереж для конкретної предметної області, що пов'язані з представленням потоку виконання (у вигляді мітки без кольору), даних у моделі (у вигляді метаданих алгоритму), декомпозицією моделі на ієрархію мереж, представлення внутрішнього стану мережі, тощо.

Основні моделі роботи відеоадаптера:

- Моделі синхронного та асинхронного копіювання даних із RAM до пам'яті відеоадаптера та в оберненому напрямку. Копіювання даних між відеоадаптерами.
- Модель CUDA-потоків (streams) [13].
- Модель ієрархії потоків. Синхронізація в межах блоку та через CPU (CUDA-потік). Передача даних між потоками.

Було розроблено мережу CUDA-потоків (streams). Ця мережа враховує особливості взаємодії CUDA-потоків — кожен потік містить чергу команд. Проте нова команда в черзі нульового потоку не може бути виконана поки наявні активні команди в інших чергах не закінчать виконання аналогічно з новими командами в не нульових чергах — вони не можуть почати виконуватись, поки є активні команди в нульовій черзі. Мережа на рис. 1 моделює такий вид взаємодії та враховує особливості асинхронного виконання коду на CPU та GPU. Команда в черзі представлена кольором ACTIONS і об'єднує в собі операції запуску CUDA-ядра (kernel), операції з пам'яттю та операції синхронізації CPU та GPU. Черги представлені за рахунок додаткових-місць-сховищ (queue index, ended index, logical queue index, logical queue ended index), що містять інформацію щодо порядоку виконання операцій у CUDA-потоці.

ЗАСТОСУВАННЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ШАБЛОНІВ ПРОГРАМУВАННЯ

Основні шаблони [14], що розглядаються в контексті GPGPU-обчислень:

- *Стратегія* (відокремлення частини функціоналу в окремий клас-стратегію, що дозволяє настроювати роботу програми заміною стратегій).
- *Команда* (інкапсуляція алгоритму в класі, що дозволяє відокремити виклик методу від його виконання) надає можливість створювати взаємозамінні компоненти алгоритму).
- *Активний об'єкт* (відокремлення виклику методу від його виконання у паралельному середовищі (у часі, і програмному потоці) через механізм запитів, що дозволяє використовувати окремий обчислювальний ресурс як співпроцесор.
- *Блокування за областю видимості* (принцип відомий як RAII — Resource Acquisition Is Initialization — захват об'єкта в конструкторі та звільнення в деструкторі классу).
- *Синхронізація на основі стратегій* (використання стратегій для гнучкого настроювання синхронізації на етапі виконання або компіляції програми (використовує блокування за областю видимості).
- *Універсальний вказівник* (абстракція над механізмами доступу до пам'яті, що уніфікує процеси читання/запису/копіювання даних для різних пристроїв (оперативна пам'ять, відеопам'ять, тощо).

Шаблон «Активний об'єкт». Структура та опис

Задача активного об'єкта полягає у відділенні виконання методу від його виклику для підвищення ефективності використання паралелізму та спрощення синхронізованого доступу до об'єкта, що працює у власному програмному потоці. В загальному випадку клієнт, який намагається використовувати функціонал об'єкта також розміщений в окремому потоці.

Шаблон ефективно вирішує наступні задачі:

- Методи, які викликаються паралельно, не мають блокувати виконання всього процесу, погіршуючи таким чином загальну ефективність ін-

ших методів. Наприклад, якщо необхідно провести низку подібних обчислень, то не потрібно очікувати на результат кожного з них, поки він не стане потрібним для подальших операцій.

- Синхронізація доступу до спільних об'єктів має бути простою. В загальному випадку методи, які вимагають синхронізації мають робити це прозоро для користувача, не вимагаючи зовнішнього використання mutex, критичних секцій або інших механізмів. Як наслідок програмний код буде більш близьким саме до предметної області задачі й не буде розв'язувати задачі, не пов'язані з доменною областю.
- Програма має прозоро використовувати наявні програмно-апаратні ресурси. Наприклад за наявності вільного відеоадаптера він має бути завантажений обчисленнями, а операції з низьким пріоритетом можуть бути відправлені для обчислень на інший комп'ютер кластера.

Структуру шаблону зображено на рис. 1.

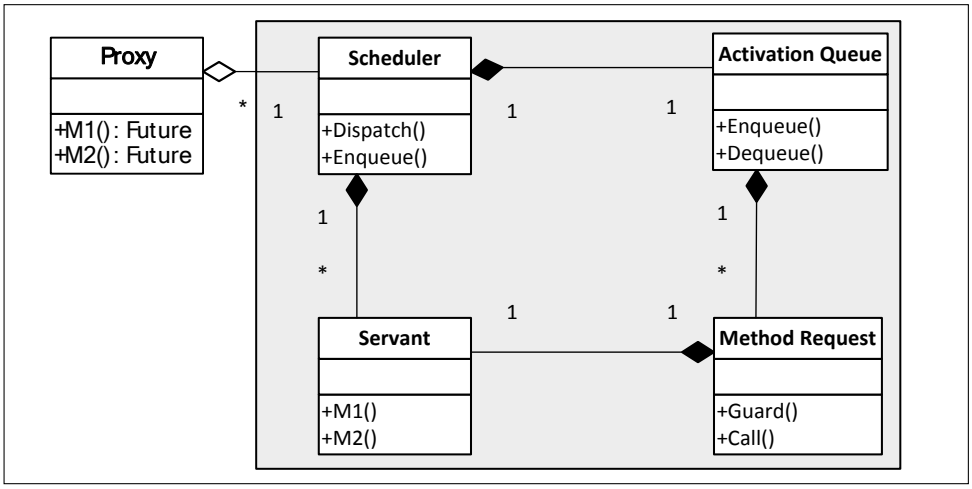

Рис. 1. Структура шаблону «Активний об'єкт»

Proxy — надає інтерфейс для клієнтів, що дозволяє викликати методи активного об'єкта використовуючи сильну типізацію замість відсилання слабко-типізованих повідомлень між потоками. Виклик методу призводить до конструювання запиту на виконання та розміщення його в активаційній черзі.

Method request — запит на виконання, що використовується для передачі контекстної інформації про специфіку виконання (наприклад параметри методу, код, пріоритет, вимоги до апаратної частини, тощо). Також об'єкт запиту містить методи для синхронізації доступу до даних. Для операції кожного типу створюється відповідний підклас запиту, який конструюється Proxy за необхідності.

Activation queue — черга, що використовується для впорядковування запитів на виконання. Також відмежовує клієнтський потік від потоку виконання методів.

Scheduler — планувальник має працювати в окремому потоці для обробки запитів на виконання, які створюються Proxy і потрапляють в активаційну чергу. Планувальник вирішує, який метод буде виконуватися

наступним і які ресурси буде задіяно для обчислень. Таке планування може бути реалізовано за низкою критеріїв — наприклад, порядок виконання може відповідати порядку виклику методів. Також можливо задіяти деяку систему пріоритетів. Можна враховувати вимоги до синхронізації, виконання специфічних умов, завершення інших операцій, тощо. Для синхронізації зазвичай використовується об'єкти запити.

Servant — визначає поведінку і стан активного об'кта. Реалізує методи, визначені в Proxy. Методи обробника викликаються для відповідних запитів від клієнта. Таким чином обробник виконується в тому ж потоці, що й планувальник.

Future — дозволяє клієнту одержати результати обчислень після того, як вони будуть завершені обробником. Якщо доступ до даних буде зроблено до завершення виконання всіх обчислень — це призведе до блокування клієнта. Можливе опитування об'єкта, щоб уникнути блокування.

Асинхронне виконання обчислень з використанням відеоадаптера

Відеоадаптер є одним із програмно-апаратних ресурсів, який дозволяє проводити масивно паралельні обчислення з великими об'ємами даних. Важливою особливістю в цьому випадку є той факт, що відеоадаптер працює з великою кількістю однотипних потоків, які виконують одну й ту ж інструкцію над масивом даних. Зазвичай інструкції достатньо прості. Таким чином відеоадаптер можна розглянути як потужний паралельний математичний співпроцесор. Завдяки активному об'єкту можна організувати його прозоре використання використання.

Як було вказано вище, планувальник перетворює запит на виконання у справжній виклик, використовуючи об'єкт-обробник. Ніщо не забороняє створити реєстр обробників, що надають доступ до різних видів ресурсів. Наприклад, виконання обчислень у мультипоточному середовищі на окремому комп'ютері кластера. Відповідно до ієрархії розробників необхідно створити ієрархію запитів на виконання. Одним із підходів до забезпечення відповідності буде надання планувальникові доступу до деякої абстрактної фабрики запитів.

Розглянемо такий варіант детальніше. Для призначення пріоритетів задачам доцільно ввести список:

```
enum EPriority {
  PRIORITY_NONE     = 0x00,  // задача може бути призупинена
                             // для виконання інших задач
  PRIORITY_LOW      = 0x01,
  PRIORITY_MEDIUM   = 0x02,
  PRIORITY_HIGH     = 0x04,
  PRIORITY_CRITICAL = 0x08, // задача вимагає призупинення
                             // виконання інших задач
  PRIORITY_ANY      = 0xFF
  }
```

Також доцільним буде введення списку доступних обробників:

```
enum EServant {
  SERVANT_CPU,
  SERVANT_GPU,
  SERVANT_CPU_MULTI,
  SERVANT_CPU_REMOTE,
  ...
  }
```

Список можна розширювати при реалізації обробників нового типу.

Під час запуску системи доступні розробники мають бути зареєстрованими у планувальника. Для цього можна використати конфігураційний файл. Також необхідно забезпечити можливість для клієнта зробити запит на наявність конкретних обробників. Це призведе до розширення інтерфейсу планувальника.

На рис. 2 зображено розши-рення структури класів. Як видно з діаграми, під час реєстрації обробника слід вказати пріорите-ти, для яких він підходить. Такі вимоги можуть бути зумовлені можливостями призупинки роботи обробника або латентністю доступу до нього (наприклад доцільною є делегація низько пріоритетних задач віддаленим процесорам, а швидкі паралельні завдання можуть бути оперативно виконані за допомогою відеоадаптера. Пріоритет завдання можна буде вказати під час створення об'єкту-запиту на виконання. Також клієнт має можливість запросити виконання задачі за допомогою конкретного обробника, якщо такий є в наявності.

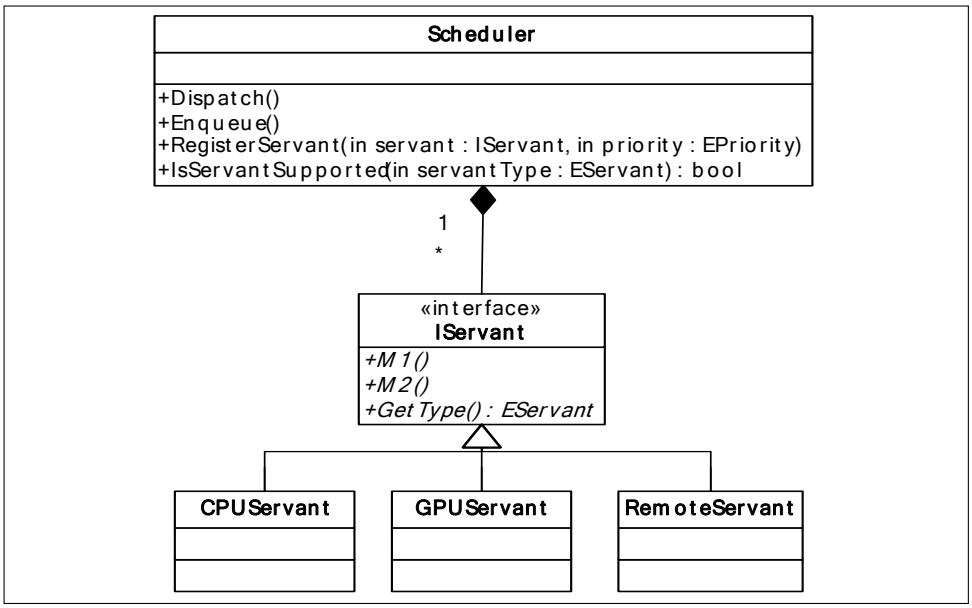

Рис. 2. Розширення структури планувальника для забезпечення підтримки різних обробників

Для забезпечення відповідності запитів на виконання до обробників даних доцільно використати фабрику запитів. Кожен обробник буде додатково реєструвати свою фабрику, яка буде готувати запит у правильній формі

(копіювання даних, специфіка синхронізації, тощо). На рис. 3 зображено схему класів для такого випадку.

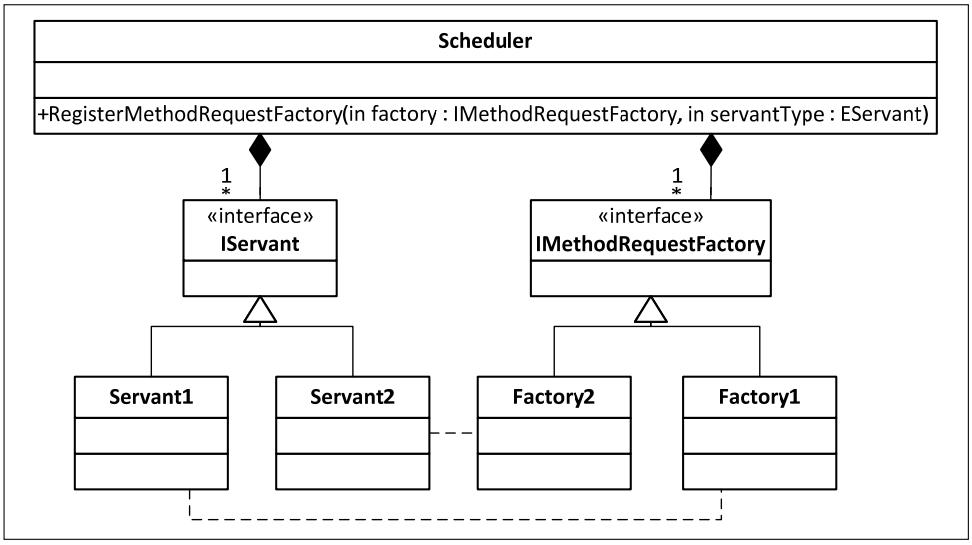

Рис. 3. Використання абстрактної фабрики для забезпечення відповідності запитів на виконання існуючим обробникам

Розглянемо приклад системи обробки даних, яка вимагає частого виконання подібних операцій над масивами даних, наприклад простого додавання числових масивів. Розглянемо необхідний набір класів для реалізації кількох стратегій виконання:

- Формування запитів на виконання. Всі запити можна уніфікувати з використанням універсального вказівника, який було розглянуто у роботі [11]. Його можна виразити приблизно наступним классом:

```
template <class TAllocationPolicy, class TData>
struct AddMethodRequest : public IMethodRequest {
  public UniPtr<TAllocationPolicy, TData> A;
     public UniPtr<TAllocationPolicy, TData> B;
}
```

Через шаблонний параметр TAllocationData можна задати виділення пам’яті для відеоадаптера або для центрального процессора.

- Обробники виконання:

  – послідовне виконання:

```
template <class TData>
class CPUServant : public IServant {
       void Add(TData *A, TData *B, size_t size) {
         TData *C = new TData[size];
         for (size_t i = 0; i < size; ++i)
            C[i] = A[i] + B[i]
       }
 }
```

– паралельне виконання:

```
template <class TData>
class MultiCPUServant : public IServant {
      void Add(TData *A, TData *B, size_t size) {
      size_t threads_count = 4;
      Threads threads[threads_count];
      for(size_t thread_id = 0;
        thread_id < threads_count;++thread_id){
          threads[thread_id] = Thread([](){
            for (size_t i=(thread_id – 1)*size /
    threads;
                i < thread_id * size / threads; ++i)
            C[i] = A[i] + B[i];
          });
        threads[thread_id].start();
      }
      for (auto thread : threads)
        Thread.join();
          }
    }
```

– виконання на GPU:

```
template <class TData>
class MultiCPUServant : public IServant {
     void _global_ _Add(TData * A, TData * B, TData
*C,
                        size_t size) {
       int idx = blockIdx.x * blockDim.x +
threadIdx.x;
       if (idx < size)
         C[idx] = A[idx] + B[idx];
       }
     void Add(TData *A, TData *B, size_t size) {
    ...
      TData *C;
    dim3 dimBlock(blocksize);
    dim3 dimGrid(ceil(size / float(blocksize)));
    _Add<dimGrid, dimBlock>(A, B, C, size);
  }
}
```

Як видно з коду, обробники повністю інкапсулюють у собі логіку обчислень, залежну від середовища виконання. Таким чином, робота клієнта зведеться до виклику функції:

```
Future *result = proxy->Add(A, B, size).
```

Об’єкт Future — накладе блокування на зчитування результату доки обчислення не буде завершено.

ЗАСТОСУВАННЯ ШАБЛОНІВ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ

Розглянемо використання Message Passing агентів (акторів) для розробки керуючого CPU-коду в GPGPU-застосуванні та проаналізуємо можливість декларативного визначення GPU-обчислення.

Парадигма Message Passing. Агенти та актори

Однією з поширених парадигм паралельного програмування є парадигма *message passing* [16,17]. Вона часто застосовується під час створення складних розподілених систем з високим рівнем паралелізму. Реалізація цієї парадигми представлена в мовах програмування в якості акторів (actor) або агентів (agent).

Основні особливості message passing:

- Застосування складається з ізольованих компонентів, що працюють паралельно (в паралельних потоках з пулу потоків (thread pool)). Взаємодія між компонентами йде через обмін повідомленнями за певним протоколом. Мережеві компоненти можуть взаємодіяти з використанням TCP, UDP, HTTP, тощо. Локальні ж компоненти взаємодіють через протокол, що визначений конкретною мовою його реалізації чи бібліотекою.
- Компонент визначає логіку обробки вхідних повідомлень. Останні потрапляють у чергу (queue) і беруться з неї послідовно для обробки.
- Компонент може володіти певними ресурсами і бути їх провайдером для інших. Ресурсом можуть бути: дані в певному форматі в оперативній пам'яті (кортежі, списки, множини, масиви, мапи або хеші, структурні типи); апаратно-програмна платформа (взаємодія через мережу, з базою даних, через USB, COM порт або з відеоадаптером) або їх комбінація (у випадку кешування доступу до бази даних наприклад використовується як данні в пам'яті так і API доступу до бази).
- Компонент має певний стан, що може інкапсулювати ресурс (зазначений вище), або, як у випадку машини станів (state machine), може бути виражений у вигляді певного алгоритму обробки повідомлень, що переводить його в інший стан.
- Інтерфейс між компонентами:
  - PostSync, postAsync — посилає повідомлення до компонента синхронно або асинхронно. У випадку синхронного методу потік, що посилає повідомлення, очікує його одержання перед виконанням наступної операції.
  - ReceiveSync, receiveAsync отримує повідомлення від компонента синхронно чи асинхронно. Асинхронне очікування полягає у тому, що потік як ресурс повертається системі (можливо лише в пулі потоків) і може бути використаний для іншої роботи. Система регіструє функцію оберненого виклику (callback) на подію появи даних від компонента (наприклад, для мережевого компоненту це поява відповідних даних на мережевому інтерфейсі). Під час отримання даних ця функція їх використовує для подальшої обробки, виконуючись у новому або тому самому потоці з пулу.

o TryReceive функції — аналогічні переліченим вище, проте використовують певний таймаут для отримання даних.

- Парадигма тісно пов'язана з поняттям асинхронного виконання.

Використання агентів і акторів в GPGPU-застосуванні

Оскільки відеоадаптер є певним ресурсом, доступ до якого може бути критичним при обчисленнях, то доцільно організувати роботу з ним у вигляді агента, що інкапсулюватиме деталі взаємодії. Переваги такого підходу такі:

- Спрощення інтерфейсу взаємодії з відеоадаптером. Фактично агент може приймати наступні повідомлення:

– Масив даних у відповідній мові програмування. За метаданими масиву (розміром, рангом, ідентифікатором) агент формує набір операцій з відеоадаптером (виділення памяті, її копіюваннння за відповідним напрямом).

– Програма для відеоадаптера у певному форматі. Для CUDA це може бути або NVidia CUDA C або PTX-асемблер. Для DirectCompute та OpenCL — відповідні шейдери. Для деяких мов (наприклад, F#) можлива передача обчислення у вигляді конструкцій цієї ж мови (F# quotations). Таке обчислення містить абстрактне синтаксичне дерево, що може бути трансформоване до формату, що зрозуміле відеоадаптером у відповідній технології. Задачею агенту є компіляція обчислення до певного рівня і збереження результату у кеші з можливістю подальшого доступу до результату.

– Параметри запуску ядра. Сюди належать розмірність ієрархії потоків, ідентифікатор ядра, тощо. Агент запускає на виконання відповідне ядро.

– Запит на результат із синхронізацією, якщо це необхідно.

- Агент є планувальником навантаження на наявні ресурси відеоадаптера. Відповідно він реалізує планування CUDA потоків з урахуванням шаблону «Staged concurrency copy and execute». За наявності декількох відеоадаптерів він надає можливість балансувати навантаження між ними, максимально ефективно використовуючи апаратні ресурси. Агент також містить статистику покриття відеоадаптера обчисленнями в певні часові проміжки.
- Агент надає можливість визначати оптимальну конфігурацію CUDA-ядра, проводячи серію його запусків з відповідними параметрами. Отриману конфігурацію можна повторно використовувати у подальших обчисленнях.
- Агент є серверним компонентом, що може приймати дані від інших агентів, що працюють з такими ресурсами як мережа чи база даних.

Декларативне визначення GPU-обчислення

При розгляді GPU-коду в GPGPU-застосуваннях (алгоритми маршрутизації, нейромережі) було зроблено висновок про досить часте застосування шаблону MapReduce [18], а точніше його деякої модифікації з урахуванням особливості архітектури. Просту модель шаблону подано на рис. 4.

Основні компоненти (розглядаються в контексті GPU-обчислення):

- Splitter — визначає шаблон обходу вхідного масиву даних не переміщуючи при цьому дані в пам'яті. За наявності декількох масивів даних йде визначення шаблонів для кожного з масиву з можливою залежністю обходів один від одного.
- Mapper — за побудованими шаблонами обходу від попереднього етапу отримує необхідні елементи даних, перетворює їх за певним алгоритмом та зберігає в спільну або глобальну пам'ять відеоадаптера.

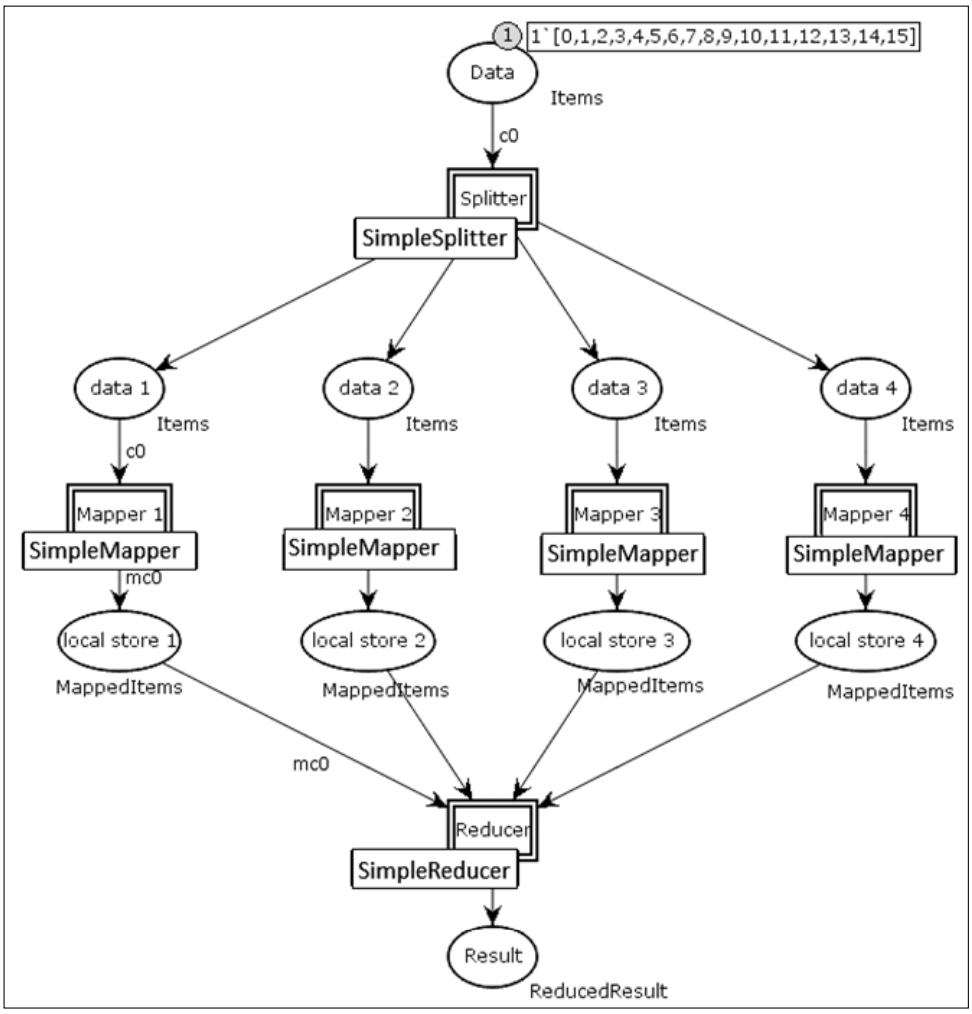

Рис. 4. Мережа Петрі шаблону MapReduce

- Reducer — акумулює дані в паралельних потоках. Спочатку потоки одного блоку опрацьовують дані у своїй спільній пам'яті, а потім йде запис даних у глобальну пам'ять і подальша акумуляція результату.

Ідея декларативного визначення GPU-обчислення полягає у наступному. Існує певний клас (CPU-код) `GpuComputation`, що може надавати наступний інтерфейс:

• `Data(dataArray, patternLambda)` — функція, що дозволяє визначити шаблон обходу масиву `dataArray` за допомогою лямбда-функції `patternLambda`. Остання ж приймає на вхід індекс warp-групи та індекс потоку в групі та має повернути множину (set) індексів у вхідному масиві `dataArray`, доступ до яких необхідно здійснити із вказаного потоку. За метаінформацією про масив `dataArray` та за вказаним лямбда-обчисленням у стані `GpuComputation` формується відповідний код для відеоадаптера (наприклад, PTX для CUDA). Також масив і його шаблон обходу зберігається в контексті `GpuComputation` для участі в наступних операціях.

• `Map(mapLambda)` — дозволяє для вказаних раніше методом `data` масивів виконати обчислення над відповідними елементами в напрямках шаблонів обходу. `mapLambda` приймає по елементу з кожного масиву та, за необхідності, набір індексів, що вказують позицію в обході шаблонів. Результатом `mapLambda` має бути новий елемент, де зберігається результат обчислення визначається наступними методами `GpuComputation`. У стані ж `GpuComputation` формується необхідний GPU-код для обчислення.

• `Reduce(reduceLamdba)` — визначає обчислення-акумулятор результату. Може застосовуватись до результату методу `map` або напряму до масивів даних, що визначені за допомогою `data`. `reduceLamdba` лямбда-функція, яка приймає на вхід як елементи даних попереднього обчислення, так і акумулятор. Як і в попередньому випадку місце збереження результату визначає наступними викликами методів з інтерфейсу `GpuComputation`.

• `ToSharedMemory` — дозволяє вказати, що результат з методів `map` чи `reduce` має бути збережений в спільну пам'ять.

• `ToData(dataArray)` — дозволяє зберегти результат в глобальну пам'ять. Є фінальним методом при побудові GPU-обчислення. Після його виконання обчислення в стані `GpuComputation` є сформованим і може бути відправлене на компіляцію та виконання.

Запропонований підхід формування обчислення не є єдиним. Можлива реалізація, що приховує особливості роботи з архітектурою відеоадаптера (відсутній метод `toSharedMemory`). Наразі йде розробка засобів побудови GPU-обчислення зазначеними методами на функціональній мові програмування F#.

ВИСНОВКИ

1. Створено методологію проектування застосувань у новітній технології GPGPU, яка включає:

• схематологію паралельних алгоритмів у суперкомп'ютерних системах з кількома рівнями паралелізму;

• метод проектування і моделювання роботи систем з масовим паралелізмом на основі відеоадаптерів з використанням математичного апарату мереж Петрі;

- застосування об'єктно-орієнтованих шаблонів програмування, що надають сукупність абстракцій для вирішення широкого класу задач.

2. На основі досліджень алгеброалгоритмічних методів проектування сформовано паралельні регулярні схеми сукупності алгоритмів для різних:

- архітектур суперкомп'ютерних систем;
- парадигм паралельного програмування.

Схеми сформовано на значному рівні абстракції, що дозволяє використовувати розвинуту систему еквівалентних трансформацій алгоритмів і здійснювати пошук оптимальних схем.

Створено моделі в термінах мереж Петрі:

- сучасних відеоадаптерів;
- основних шаблонів, що використовуються при побудові GPGPU-застосувань.

3. Запропоновано низку правил формального представлення виконання GPGPU-застосувань за допомогою математичного апарату мереж Петрі з метою подальшого їх моделювання.

4. Запропоновано використання об'єктно-орієнтованих шаблонів програмування для забезпечення різних сценаріїв функціонування відеоадаптерів. Розглянуто аспекти, що ґрунтуються на поліморфізмі та параметризованих типах (в сенсі C++). Перший випадок надає більше гнучкості на етапі виконання програми, а другий забезпечує збільшення продуктивності за рахунок повністю прекомпільованого коду.

5. Створено програмний інструментарій підтримки названих методів.

ЛІТЕРАТУРА

1. *Погорелый С.Д., Бойко Ю.В., Трибрат М.И., Грязнов Д.Б.* Анализ методов повышения производительности компьютеров с использованием графических процессоров и программно-аппаратной платформы CUDA // Математичні машини і системи. — 2010. — № 1. — С. 40–54.
2. *Погорілий С.Д., Вітель Д.Ю., Верещинський О.А.* Новітні архітектури відеоадаптерів. Технологія GPGPU. Частина 1 // Реєстрація, зберігання і обробка даних. — 2012. — **14**. — № 4. — С. 52–65.
3. *Погорілий С.Д., Вітель Д.Ю., Верещинський О.А.* Новітні архітектури відеоадаптерів. Технологія GPGPU Частина 2 // Реєстрація, зберігання і обробка даних. — 2013. — **15**. — № 1. — С. 71–81.
4. *Anisimov A.V., Pogorilyy S.D., Vitel D.Yu.* About the Issue of Algorithms formalized Design for Parallel Computer Architectures. Applied and Computational Mathematics. — **12**. — № 2. — 2013. — P. 140–151.
5. *Pogorilyy S.D., Gusarov Y., Paralleling A.D.* Of Edmonds-Karp Net Flow Algorithm // Applied and Computational Mathematics. — 2006. — **5**, № 2. — P. 121–130.
6. *Погорілий С.Д., Мар'яновський В.А., Бойко Ю.В., Верещинський О.А.* Дослідження паралельних схем алгоритму Данцига для обчислювальних систем зі спільною пам'яттю // Математичні машини і системи. — 2009. — № 4. — С. 27–37.

7. *Yingxu Wang.* Software Engineering Foundations. A Software Science Perspective. Auerbach Publications, Taylor & Francis Croup, Boca Raton New York, 2008 by Taylor & Francis Group, LLC, P. 217 – 262.
8. *Котов В.Е*. Сети Петри. — М.: Наука, ГРФМЛ, 1984
9. *Погорілий С.Д., Вітель Д.Ю.* Використання мереж Петрі для проектування паралельних застосувань // Проблеми програмування. — 2013. — № 2. — С. 32–41.
10. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д*. Приемы обьектно-ориентированного проектирования. Паттерны проектирования. — СПб: Питер, 2009. — 366 с.
11. *Погорілий С.Д., Верещинський О.А.* Створення методики проектування застосувань для програмно-апаратної платформи CUDA // Проблеми програмування. — 2013. — № 3. — С. 47–60.
12. *Ruben Vermeersch.* Concurrency in Erlang & Scala: The Actor Model. — http://savanne.be/articles/concurrency-in-erlang-scala/.
13. *Messages* and Agents. F# for fun and profit. — http://fsharpforfunandprofit.com/posts/concurrency-actor-model/.
14. *Jeffry Dean, Sanjay Ghemawat.* MapReduce: Simplified Data Processing on Large Clusters. Google, Inc. OSDI 04: 6th Symposium on Operating Systems Design and Implementation.

Надійшла 24.06.2014.