ПРОГРАММИРОВАНИЕ

УДК 004.432.2, 004.435, 004.422.837

ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОВ ПРИ ПОМОЩИ РАСШИРЯЕМЫХ ЯЗЫКОВ

А.В. Адинец

PROGRAMMING GRAPHICS PROCESSORS WITH EXTENSIBLE LANGUAGES

A.V. Adinetz

В статье рассматривается система программирования ГПУ NUDA, созданная на основе расширяемого языка Nemerle. Она даёт программисту контроль над процессом переноса программы на ГПУ, переложив механическую работу на компилятор. Макросы и аннотации облегчают перенос программ на ГПУ и повышают их производительность без ущерба для размера и читаемости исходного кода. На ряде архитектур ГПУ и задач удалось добиться повышения производительности в несколько раз по сравнению с исходным вариантом.

Ключевые слова: языки программирования, параллельное программирование, расширяемое программирование, метапрограммирование, графические процессоры, ГПУ.

This paper presents a GPU programming system, NUDA, built on top of an extensilbe language, Nemerle. NUDA provides control over porting application to GPU, while making the compiler do the mechanical work. Macros and annotations simplify the porting process, and increase efficiency without sacrificing code size and readability. On a number of problems and architectures, use of NUDA increased performance several times compared to initial implementation.

Keywords: programming languages, Parallel programming, Extensible programming, metaprogramming, GPU, GPGPU.

Введение

Графические процессорные устройства (ГПУ) в настоящее время активно используются для решения вычислительных задач. Использование ГПУ позволяет получить ускорение на 1 – 2 порядка по сравнению с традиционными вычислительными системами и достигнуть при этом значительно больших показателей производительности на один процессор, энергоэффективности и производительности в расчете на стоимость. Активно создаются гетерогенные кластеры на основе ГПУ. Например, в ноябрьском списке «Топ-500» за 2010 год [1] 3 из 5 самых мощных вычислительных систем построены на основе ГПУ NVidia, а всего в списке 11 систем на основе ГПУ.

В связи с этим актуальной является проблема создания высокоуровневых средств программирования для систем на основе ГПУ. В настоящее время для решения вычислительных задач на ГПУ преимущественно используются низкоуровневые средства разработки:

CUDA [2], Brook+ и OpenCL [3]. Несмотря на то, что все они являются расширениями языка C, использование низкоуровневых инструментов для программирования ГПУ сопряжено с трудностями. Создаваемые с их помощью программы являются громоздкими и зачастую трудными для чтения. Из перечисленных выше инструментов только OpenCL позволяет создавать переносимые программы; однако и на этом языке оптимизированные версии программ для разных ГПУ будут сильно отличаться.

В последнее время разработка высокоуровневых средств программирования ГПУ ведется достаточно активно. Наиболее популярным подходом является добавление в программу на языке Fortran или C директив размещения данных и компиляции блоков кода для исполнения на графическом процессоре. По этому пути пошли создатели компилятора PGI [4] и CAPS HMPP [5]. Система директив PGI является более краткой и высокоуровневой; в CAPS HMPP директивы длиннее, однако они позволяют осуществлять более тонкий контроль за использованием ускорителя. Однако ни та, ни другая система директив в настоящее время не является стандартной и понимается только компиляторами данных производителей. Если производительность сгенерированных программ оказалась низкой, или если программа задействует нестандартный паттерн работы с ГПУ, единственной возможностью программиста является переход на низкоуровневое средство. Возможности добавить свою директиву преобразования или воспользоваться лишь частью функциональности системы у программиста нет. Наконец, далеко не всегда удобно создавать программу для ГПУ на последовательном языке; часто удобнее сразу записывать программу в параллельном виде.

В данной статье рассматривается подход с использованием парадигмы *расширяемого программирования* [6] с целью создания программ для систем на основе ГПУ. Расширяемый язык программирования — это такой язык программирования, синтаксис и семантика которого могут быть расширены или изменены для облегчения написания программы. Вклад данной статьи можно кратко обозначить следующим образом:

- Предложена идея использования расширяемых языков для создания систем программирования высокопроизводительных вычислений;
- Идея продемонстрирована на примере системы расширений NUDA (Nemerle Unified Device Architecture), предназначенной для программирования ГПУ на языке Nemerle;
- Система апробирована на целом ряде модельных задач, и во многих из них позволила достигнуть ускорения в несколько раз по сравнению с исходным вариантом без увеличения размера исходного кода.

Статья организована следующим образом. В разделе 1 дается краткий обзор расширяемых языков и их особенностей на примере языка Nemerle. В разделе 2 описывается предложенная система расширений NUDA. В разделе 3 приводятся результаты использования системы NUDA для решения задач на ГПУ. В разделе 4 обсуждаются полученные результаты и направления дальнейшей работы. Наконец, в разделе 5 подводится итог результатов работы.

1. Расширяемые языки программирования

Расширяемый язык — это такой язык программирования, синтаксис и семантика которого могут быть изменены и расширены для облегчения написания программы; при этом должны соблюдаться ряд условий. Во-первых, расширение должно выполняться без изменения компилятора языка, что предполагает наличие в компиляторе механизма плагинов. Во-вторых, расширения должны восприниматься одинаково всеми компиляторами, и как следствие, должен существовать интерфейс компилятора, которыми могут пользоваться расширения. Наконец, средства поддержки расширений должны быть предусмотрены в

синтаксисе языка. В настоящее время существует небольшое количество расширяемых языков: LISP с его диалектами [7], Seed7, Nemerle [8], xoc [9]. В последнем случае описывается механизм расширений языка C, при этом сами расширения пишутся на языке Zeta.

По сравнению с традиционными языками, расширяемые языки обладают рядом преимуществ. Строить системы программирования на основе расширяемых языков проще, и они по определению являются открытыми. В рамках расширяемого языка можно строить многоуровневые модели программирования. Например, сначала реализуется просто обёртка над низкоуровневым средством, затем — удобная система аннотаций для программирования ускорителя, и наконец — система автоматического распараллеливания программ. Программист получает, с одной стороны, больший контроль над используемыми преобразованиями программ, а с другой — возможность программировать на низком уровне в случае необходимости. Вследствие открытости системы программист получает возможность использовать преобразования, полезные только для специфических задач, и которые поэтому не имеет смысл реализовывать в самом компиляторе. Открытость и расширяемость системы позволит задействовать ресурсы сообщества разработчиков и осуществлять таким образом последовательное наращивание функциональности. Наличие фиксированного ядра означает, что для программирования для новых моделей или архитектур требуется изучение сравнительно небольшого объёма расширений. Таким образом, облегчается изучение новых архитектур и моделей. Наконец, код уже созданных приложений может быть обработан при помощи новых расширений, что облегчит его перенос на уже существующие архитектуры.

Для работ, описываемых в данной статье, был выбран язык Nemerle, по двум причинам: во-первых, он является относительно стабильным и развитым, и во-вторых, он больше остальных похож на традиционные языки программирования, используемые для высокопроизводительных вычислений. В своей основе Nemerle является. NET-языком, похожим на C#; основным синтаксическим отличием является объявление типа переменной после её имени, через двоеточие. Тип локальных переменных выводится автоматически на основании её инициализатора и выражений, в которых она используется; поэтому явно тип объявляется только для параметров функций. Основным средством расширения языка Nemerle является *макрос* — аналог функции, которая исполняется на этапе компиляции, принимает на вход фрагменты кода (и, возможно, константы) и возвращает новый фрагмент кода. После исполнения макроса возвращаемый фрагмент кода подставляется на место вызова макроса. Для создания фрагментов кода используется механизм *квазицитирования* — фрагмент кода как объект представляется его записью на языке Nemerle, заключённой между скобками `<[` и `]>`. Внутри квазицитирования можно использовать выражение `$expr` */*$*/* для подстановки результата вычисления выражения `expr` в фрагмент кода. Если результатом выражения является список фрагментов кода, то используется оператор `..$expr` */*$*/*. Аналогичные конструкции внутри сопоставителя в операторе **match** позволяют выполнять разбор кода. С помощью объявления **syntax** можно задать для вызова макроса более удобный синтаксис. Внутри макроса можно обращаться к стандартному интерфейсу компилятора, например, для генерации ошибок или получения типа выражения. возможностях языка приведена в `man-nemerle`.

Несмотря на свою простоту, механизм макросов в языке Nemerle является достаточно мощным. Например, операторы условного перехода и циклов реализованы при помощи макросов через оператор сопоставления и вложенные функции, или замыкания. Стандартная библиотека макросов Nemerle содержит реализацию асинхронных методов и блоков, регулярных выражений и контрактов на параметры и возвращаемые значения функций. Макросы могут применяться не только к фрагментам кода, но также и к классам и их членам.

2. Система расширений NUDA

NUDA [10] — это система расширений языка Nemerle, предназначенная для программирования ГПУ. Входящие в NUDA расширения можно условно разделить на следующие группы:

- Макросы, функции и типы данных, реализующие вложение подмножества OpenCL в язык Nemerle;
- Макросы и аннотации, предназначенные для переноса на ГПУ кода и данных;
- Аннотации, предназначенные для преобразования циклов и повышения производительности программ;
- Прочие макросы и аннотации.

Помимо расширений, NUDA включает набор обычных библиотечных функций и классов, осуществляющих интерфейс с системой времени исполнения OpenCL, а также вспомогательные операции по работе с кодом на языке Nemerle.

Работа системы расширений NUDA организована следующим образом. На этапе компиляции срабатывают макросы NUDA, выполняются преобразования кода, а также генерация кода для ГПУ. Если компиляция проходит без ошибок, сгенерированный код на OpenCL сохраняется в создаваемую .NET-сборку в виде атрибутов. Во время исполнения этот код компилируется при помощи системы времени выполнения OpenCL и исполняется на ГПУ.

Цикл nfor является макросом, позволяющим компактно записывать тесно вложенные гнёзда циклов. Общий вид полной формы цикла **nfor** вместе с кодом, в который он преобразуется по умолчанию, изображён на рис. 1. Существует также *сокращённая форма*, изображённая там же. Шаг по умолчанию равен 1; если же вместо диапазона указано только одно скалярное значение, n, то параметр цикла изменяется от 0 до $n-1$ включительно.

```
/* nfor macro ... */
nfor((i1, ..., iN) in (a1 <> b1 :/ c1, ..., aN <> bN :/ cN))
  S;
/* ... is expanded into */
for(mutable i1 = a1; i1 <= b1; i1 += c1)
  ...
  for(mutable iN = aN; iN <= bN; iN += cN)
    S;

/* short form is also possible */
nfor((i, j, k) in (n, n, n))
  a[i, j, k] += b[i, j, k];
```

Рис. 1. Общая (и сокращённая) форма цикла **nfor** и результат его трансляции

Для работы с памятью ГПУ используются специальные типы данных. Для работы с глобальной памятью служит тип `nuarrayXd[T]` (X — размерность массива), или *NUDA-массивы*, для изображений `nuimageXd` для локальной памяти `nulocalarrayXd` и для константной памяти `nuconstarrayXd`. Последние два могут использоваться только в программах, исполняемых на ГПУ. Эти типы данных эмулируют функциональность .NET-массивов и реализуют доступ к элементам по индексам, а также получение информации о ранге и размере по каждому из измерений.

Выделение памяти под массивы для ГПУ реализуется путём применения макросов к операции выделения массива в Nemerle. Массивы в локальной памяти могут выделяться только

в коде на стороне ГПУ при помощи макроса **nulocal**; размер массива по каждому из измерений должен быть константой. NUDA-массивы и изображения могут выделяться только на стороне хоста. Для этого служат макросы **nunew** и **nuimg** соответственно, генерирующие обращение к системе времени выполнения OpenCL для выделения памяти устройства. С каждым потоком хоста ассоциирован свой номер устройства (по умолчанию 0), на котором и выделяется память под массив. Управление глобальной памятью и памятью изображений осуществляется при помощи сборки мусора. Массивы константной памяти существуют только на стороне ГПУ; на стороне хоста они представляются как обычные NUDA-массивы.

NUDA-массивы и изображения содержат методы для копирования данных между ними и .NET-массивами соответствующего типа и ранга. Кроме того, в NUDA-массивах и изображениях предусмотрена возможность доступа к данным на стороне хоста и автоматической синхронизации между ними. Синхронизация осуществляется лениво: на сторону ГПУ массив копируется при вызове ядра, если он мог быть изменён на хосте. Обратно массив копируется при обращении к его элементу на стороне хоста. Память под такие массивы на стороне ГПУ выделяется всегда, а на стороне хоста — только когда там производится обращение к данным.

Точкой входа в исполнение программы на ГПУ является *NUDA-ядро*. Это статический метод класса, помеченный макросом **nukernel**. Ядро должно возвращать тип **void**, а типами его параметров должны быть простые типы данных, изображения или NUDA-массивы. Для метода-ядра генерируется, во-первых, *метод-заглушка* для его вызова на хосте, и во-вторых, код ядра OpenCL. Метод-заглушка выполняет установку параметров ядра, синхронизацию массивов и запуск ядра. В параметры заглушки, помимо параметров ядра, входят также номер устройства, размер сетки потоков и размер блока потоков. Заглушка возвращает управление только после завершения исполнения ядра на ГПУ. Вызов NUDA-ядра осуществляется только при помощи макроса **nucall**, который дополнительно указывает номер устройства и параметры сетки потоков. Пример объявления ядра и его вызова приведён на рис. 2. По сути, пара макросов **nucall** и **nukernel** аналогична нотации `<<<...>>>` и `__global__`-функциям в CUDA. Если на стороне ГПУ требуется вызвать пользовательскую функцию, к ней применяется макрос **nucode**; при этом сохраняется возможность вызова еҷ на стороне хоста.

```
/* kernel declaration */
nukernel arrayBy2(b : nuarray2d[float], a : nuarray2d[float]) : void {
  def i = globalId(0); def j = globalId(1);
  b[i, j] = 2.0f * a[i, j];
}
/* ... */
def a = nunew array(n, n) : array[2, float];
def b = nunew array(n, n) : array[2, float];
/* ... here init array a ... */
/* kernel invocation */
nucall(0, [n, n], [1, 128]) arrayBy2(b, a);
```

Рис. 2. Пример объявления ядра в NUDA и его вызова

Чаще всего в качестве ядра для ГПУ используется тело цикла, поэтому хотелось бы иметь макрос для переноса цикла на ГПУ. В Nemerle это можно легко реализовать; такой макрос реализован в NUDA и называется **nuwork**. Он применяется к циклам **nfor** и группам из нескольких таких циклов. В качестве обязательных параметров он принимает размер блока потоков, по одному целочисленному выражению (не обязательно константному) по каждому из измерений. Макрос **nuwork** на основании текущего контекста и анализа кода тела цикла определяет набор переменных, которые используются в теле цикла, но определе-

ны вне него. Из них составляется список параметров ядра. Тело ядра формируется из тела цикла, вычисления индексов цикла через глобальный номер потока, а также условия, позволяющего корректно исполнять цикл даже в том случае, когда глобальный размер сетки не делится на размер группы потоков. На место цикла подставляется вызов макроса **nucall**, осуществляющий вызов сгенерированного ядра. Пример использования **nuwork** приведён на рис. 3; он решает ту же задачу, что и пример на рис. 2. Размер программы меньше за счёт отсутствия необходимости обявлять параметры ядра. Кроме того, в данном примере можно легко заменить типы массивов на изображения; если ядро выделено явно, это делать сложнее.

```
/* array allocation and initialization */
nuwork(1, 128) nfor((i, j) in (n, n))
  b[i, j] = 2.0f * a[i, j];
```

Рис. 3. Использование макроса **nuwork** для переноса цикла на ГПУ

Код на языке OpenCL генерируется из кода Nemerle-методов, помеченными макросами **nukernel** и **nucode**. Перед собственно генерацией кода выполняется раскрытие макросов. Для этого используется специальная функция, не раскрывающая стандартные макросы.

В сгенерированном OpenCL-коде сначала идут объявления типов, затем — прототипы функций, и наконец, код функций. При этом базовые типы языка Nemerle транслируются в базовые типы OpenCL. Указательные типы OpenCL представлены в Nemerle специальными типами `ptr[T]`, `globalptr[T]` и `localptr[T]`. Типы-массивы транслируются в структуры, содержащие размер массива и указатель на его данные. Типы-кортежи транслируются в структуры, содержащие поля кортежей. Типы-структуры транслируются в структуры OpenCL.

Прототипы сгенерированных функций в OpenCL получаются из прототипов функций в Nemerle заменой Nemerle-типов на соответствующие OpenCL-типы. При передаче в функции-ядра массивы разделяются на параметры, соответствующие указателям на данные и размерам массивов, которые передаются отдельно. В прологе кода ядра они собираются в структуры, описанные выше. Кроме того, в нестатические функции-члены структур добавляется первый параметр **this**, а возвращаемым типом конструкторов структур становится тип самих структур.

В подмножестве кода Nemerle, транслируемом в OpenCL, запрещено использование блочных операторов на уровне выражений. Таким образом, структура транслируемого кода Nemerle почти полностью соответствует структуре кода на C. Трансляция выполняется как процесс рекурсивной генерации строк кода на OpenCL по дереву кода на Nemerle. Кортежные объявления переменных транслируются в объявление нескольких переменных. Операции с массивами транслируются по-разному в зависимости от класса памяти массива. Наконец, вызовы функций также транслируются по-разному для обращений к статическим и нестатическим функциям, а также стандартным функциям OpenCL. Для ядра, изображенного на рис. 2, будет сгенерирован код ядра OpenCL, изображенный на рис. 4.

Иногда к выражению, чаще всего к циклу, требуется применить несколько преобразований подряд, при этом часть результата предыдущего преобразования подаётся на вход следующему. Просто макросы здесь не подходят, поскольку преобразование требуется применять не ко всему результату предыдущего преобразования, а к его «главной» части. Конечно, можно главную часть вычленять в самом коде макроса — но как определить, что является главным? Если в результате преобразования одного цикла получается несколько новых — к какому из них применить очередное преобразование?

```
kernel arrayBy2(global float* b_d, int b_l0, int b_l1,
  global float* a_d, int a_l0, int a_l1) {
  array2d_g_float_t a; a.d = a_d; a.l[0] = a_l0; a.l[1] = a_l1;
  array2d_g_float_t b; b.d = b_d; b.l[0] = b_l0; b.l[1] = b_l1;
  int i = get_global_id(0);
  int j = get_global_id(1);
  b.d[i * b.l[0] + j] = 2.0f * a[i * a.l[0] + j];
}
```

Рис. 4. Пример ядра OpenCL, сгенерированного для ядра NUDA

В этом случае используется *механизм аннотаций*. Каждая аннотация, помимо параметров и преобразуемого выражения, также принимает цепочку аннотаций для дальнейшего применения. Эта цепочка аннотаций применяется к «главной» части результата применения текущей аннотации. Для применения цепочки аннотаций к выражению служит макрос **annot**, пример использования которого показан на рис. 5. Сначала в гнезде будет изменён порядок циклов (**permut**), затем применена развёртка (**unroll**) и, наконец, будет выполнен тайлинг цикла (**tilem**). Такой механизм позволяет выражать сложные преобразования через последовательность аннотаций, производящих относительно простые преобразования.

```
annot(tilem(8, 8), dmine(2, 2), permut(2, 1))
nfor((i, j) in (m, n)) c[j, i] = a[i] * b[j];
```

Рис. 5. Пример применения к циклу нескольких аннотаций

Аннотация **dmine** выполняет *глубокую развёртку* цикла **nfor**. Параметрами **dmine** являются размеры блока развёртки для каждого из измерений цикла, и они должны быть константами. В результате преобразования создаётся необходимое число копий тела цикла, которые затем перемешиваются: сначала выполняется первый оператор каждой из копий, затем второй и т.д. Если внутри тела цикла встречается другой цикл, параметры которого не зависят от номера итерации внешнего цикла, то содержимое его тела также перемешивается. Таким образом, глубокая развёртка позволяет увеличить количество однородных команд внутри наиболее вложенных циклов. Это приводит к улучшению шаблонов доступа в ОЗУ, и позволяет задействовать векторные команды для векторных архитектур. В экспериментах использование **dmine** в ряде случае позволило повысить производительность в несколько раз. Однако глубокая развёртка имеет и обратную сторону: слишком большой размер блока приводит к неэффективному использованию регистров. Пример использования аннотации **dmine** приведён на рис. 6.

Прочие аннотации Приведём описание ещё некоторых аннотаций, реализованных в NUDA:

- **inline** — выполняет полную развёртку цикла с постоянным числом итераций;
- **nudevs** — исполняет каждую итерацию цикла на свочм NUDA-устройстве в отдельном потоке; удобно при программировании нескольких ГПУ;
- **peel** — позволяет отделить несколько итераций в начале и конце цикла от основного блока тела цикла;
- **permut** — меняет порядок измерений цикла;

```
/* dmine-annotated loop ... */
dmine(2) nfor(i in n) {
  mutable r = 0.0f;
  def aa = a[i];
  nfor(j in n) {def bb = b[j]; r += f(aa, bb);}
  c[i] = r;
}

/* is transformed into ... */
/* ... main loop ... */
nfor(i1 in n :/ 2) {
  mutable r0 = 0.0f; mutable r1 = 0.0f;
  def aa0 = a[i1]; def aa1 = a[i1 + 1];
  nfor(j in n) {
    def bb0 = b[0]; def bb1 = b[1];
    rr0 += f(aa0, bb0); rr1 += f(aa1, bb1);
  }
  c[i1] = r0; c[i1 + 1] = r1;
}
/* .. and tail */
nfor(i in n / 2 * 2 <> n - 1) {
  mutable r = 0.0f;
  def aa = a[i];
  nfor(j in n) {def bb = b[j]; r += f(aa, bb);}
  c[i] = r;
}
```

Рис. 6. Использование аннотации **dmine**

- **tile**, **tilem** — тайлинг цикла; размер блока не обязан быть константой времени компиляции;
- **unroll** — простая развёртка циклов, без перемешивания итераций и преобразования вложенных циклов.

С полным списком аннотаций в `nuda` можно ознакомиться в [10].

Параметры командной строки могут быть задействованы в NUDA-программах при помощи макроса **config**, который применяется к объявлению переменной или поля класса. Поддерживаются параметры строкового типа, а также простых типов; для последних используются стандартные функции .NET для преобразования строки в число. Значение параметра по умолчанию берётся из инициализатора в объявлении; в случае его отсутствия параметр должен быть указан в командной строке. Поддерживаются как длинные, так и короткие имена параметров. Для инициализации из переменных среды используется макрос **envfig**.

3. Вычислительные эксперименты

Тестирование системы NUDA проводилось на ряде модельных задач, а также на задаче поиска коллизий для урезанной хэш-функции MD5 методом грубой силы. Тестовые модельные задачи и их характеристики приведены в табл. 1. Параметры систем, использовавшихся для тестирования, приведены в табл. 2. Тестирование проводилось в 2 этапа. На первом этапе выполнялась адаптация задачи для ГПУ с использованием макроса **nuwork**, а также подбирался оптимальный размер блока потоков. На втором этапе программа оптимизировалась при помощи аннотаций **dmine** и **inline**. При этом измерялось повышение производительности, а также определялся предполагаемый размер кода ядра при условии, что оно создавалось бы вручную. Производительность вычислялась исходя из времени исполнения

ядра на ГПУ и не включает время копирования данных. Значения производительности и ускорения приведены в табл. 3. График роста размера кода для каждой из архитектур по сравнению с базовым вариантом приведён на рис. 7. Заметим, что размер кода указан после преобразований NUDA; размер исходного кода остаётся таким же. Полный код примеров, а также использованные аннотации можно найти в дистрибутиве NUDA [10].

Таблица 1

Задачи, используемые для тестирования NUDA

Имя	Название	Размер задачи	Размер кода
imgconv	Фильтрация изображений	4096 × 4096, фильтр 3 × 3	9 строк
nbody	Задача N тел	65536	27 строк
sgemm	Умножение матриц $A^T B$, 32 бит	2048 × 2048	10 строк
dgemm	Умножение матриц $A^T B$, 64 бит	2048 × 2048	10 строк

Таблица 2

Системы, используемые для тестирования NUDA-программ

Параметр	Система **tesla**	Система **fermi**	Система **ati**
ГПУ	NVidia Tesla C1060	NVidia Tesla C2050	AMD Radeon 5830
Пиковая производительность, 32 (64) бит	622 (90) ГФлоп/с	1030 (515) ГФлоп/с	1792 (358.4) ГФлоп/с
Реализация OpenCL	NVidia CUDA 3.1		AMD Stream SDK 2.2

Из всех задач, наименьший рост производительности достигается на задаче N тел, главным образом благодаря её высокой производрительности неоптимизированного варианта. Далее, производительность базовых вариантов всех задач на архитектуре NVidia Tesla C2050 выше — за счёт наличия аппаратной кэш-памяти. Тем не менее, и там фильтрацию изображений удачтся ускорить почти в 2 раза, а умножение матриц — в 4 – 6 раз. Максимальная достигаемая производительность на ГПУ AMD сравнительно невелика; очевидно, текущая реализация OpenCL не может организовать кэширование обращений в память по указателям. Тем не менее, и там удаётся получить ускорение в 1,5 – 4 раза, в зависимости от задачи. Ни в одной из реализаций не используется работа с локальной разделяемой памятью, т.к. соответствующие преобразования пока не реализованы в NUDA. Тем не менее, на задаче **dgemm** на ГПУ Tesla C2050 удаётся добиться эффективности решения задачи 40%. Это значение производительности находится на уровне NVidia CUBLAS 3.1, и всего лишь на 20 процентных пунктов хуже, чем самая эффективная на сегодняшний день версия, реализованная в NVidia CUBLAS 3.2.

Эффективность NUDA также тестировалась на задаче поиска коллизий для урезанной хэш-функции MD5 методом грубой силы. Наиболее вычислительно ёмкой частью данной задачи является вычисление результатов последовательного применения хэш-функции к исходному значению. Эта операция, в свою очередь, применяется к большому количеству значений параллельно, и для этого задействуются графические процессоры. Задача практически не взаимодействует с ОЗУ, все вычисления происходят на регистрах. Тем не менее, за

Таблица 3

Производительность системы NUDA на различных тестовых задачах

Задача	Система	Базовая производительность, ГФлоп/с	Оптимизированная производительность, ГФлоп/с	Ускорение, раз
imgconv	**tesla**	8	40	$\times 5$
imgconv	**fermi**	97	187	$\times 1,93$
imgconv	**ati**	8,8	13,5	$\times 1,53$
nbody	**tesla**	129	154	$\times 1,19$
nbody	**fermi**	328	419	$\times 1,28$
nbody	**ati**	77	176	$\times 2,28$
sgemm	**tesla**	11,34	85,91	$\times 7,57$
sgemm	**fermi**	77	445	$\times 5,78$
sgemm	**ati**	54	234	$\times 4,33$
dgemm	**tesla**	17,84	43,5	$\times 2,44$
dgemm	**fermi**	52	216	$\times 4,15$
dgemm	**ati**	28	111	$\times 3,96$

счёт использования аннотаций NUDA, а именно глубокой развёртки цикла, отправляемого на ГПУ, как видно из рис. 8, на ГПУ AMD удалось повысить производительность почти в 4 раза. В терминах целочисленных операций, эффективность использования ресурсов ГПУ AMD составила 95%.

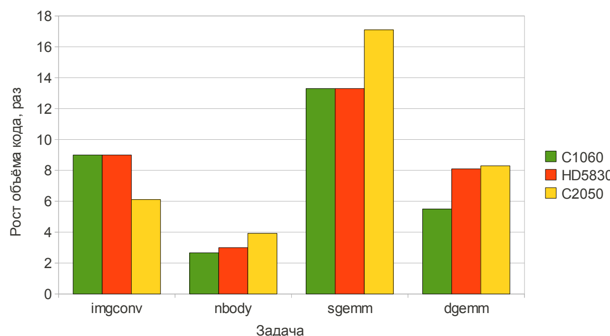

Рис. 7. Увеличение объчма кода при (предполагаемой) ручной оптимизации по сравнению с базовым вариантом

4. Обсуждение и дальнейшая работа

При помощи макросов в рамках системы NUDA для расширяемого языка Nemerle удалось реализовать функциональность программирования ускорителей и преобразования циклов при помощи аннотаций. Для ряда задач это позволило получить значительное увеличение производительности без увеличения размера исходного кода. Заметим, что изначально язык Nemerle не был особым образом предназначен для параллельного программирования — он просто является расширяемым языком. Тем не менее, в итоге удалось создать систему,

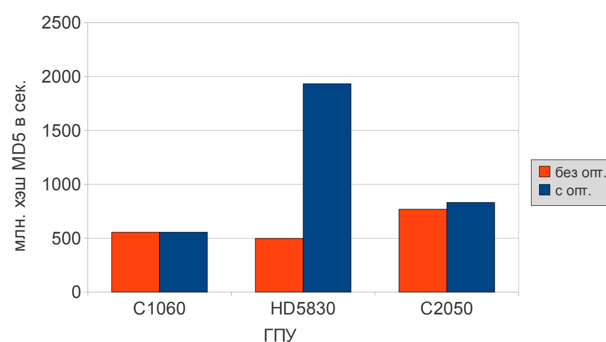

Рис. 8. Производительность вычисления хэш-функции MD5, реализованного с помощью системы NUDA, оптимизациями и без, на различных ГПУ

по функционалу сопоставимую с коммерческими решениями, такими как PGI Accelerator или CAPS HMPP. Объчм кода системы NUDA (без тестов) составляет всего 11 тыс. строк. Объчм кода коммерческой системы, такой как CAPS HMPP, скорее всего, намного больше. Таким образом, продемонстрирована эффективность использования расширяемых языков для создания систем для прогарммирования высокопроизводительных архитектур, в частности ГПУ.

Тем не менее, функциональность системы NUDA требует дальнейшего расширения. Прежде всего, следует создать аннотации для автоматизации работы с другими уровнями иерархии памяти ГПУ, в особенности разделяемой общей памяти. Далее, актуальной задачей является автоматический подбор оптимизирующих преобразований — как на основе моделей, так и на основе эмпирического поиска — и это тоже одно из направлений дальнейших исследований. Также имеет смысл разрабатывать расширения, облегчающие программирование систем из нескольких ГПУ. Наконец, важным направлением будет создание расширений для решения на ГПУ нетривиальных задач — например, задач с рекурсивным параллелизмом, или задач, требующих использования сложных примитивов синхронизации.

Кроме того, Nemerle не является идеальным расширяемым языком для высокопроизводительных вычислений. Прежде всего, получение семантической информации внутри макросов затруднено. В частности, информация об объявленных переменных и типах выражений становится доступной только в процессе развёртки макросов. Если же требуется и получить информацию, и сохранить макросы в дереве исходного кода, требуется использовать обходные пути, которые не всегда работают. Во-вторых, все функции в Nemerle-программе транслируются независимо, и нет возможности получить код одной функции из другой функции. Как следствие, реализация преобразований типа встраивания функций или генерации специальных версий функций для определённых наборов параметров затруднены. В-третьих, для инициирования преобразований требуется применять макросы; возможность инициирования преобразований через определённые шаблоны кода отсутствует. Соответственно, нет возможности выполнить преобразования, добавляющие новую семантику без добавления синтаксиса — как добавление в язык массивного программирования. Наконец, в языке отсутствует гибкий механизм назначения атрибутов вершинам дерева кода и переменным. Всё это говорит о том, что для высокопроизводительных вычислений потребуется разработать более гибкий расширяемый язык программирования. Но для этого сначала потребуется попробовать решить при помощи NUDA и Nemerle более широкий круг задач — чтобы определить требования, предъявляемые к новому языку.

5. Заключение

В статье рассказано о парадигме расширяемого программирования, её преимуществах и возможностях использования для программирования ГПУ. Была описана система NUDA, представляющая собой набор расширений языка Nemerle для программирования ГПУ и преобразования исходного кода программ. Разработанная система была апробирована на ряде задач, для которых позволила добиться увеличения производительности в несколько раз без увеличения объёма исходного кода. При этом, если бы соответствующий код писался вручную, его объъм вырос бы в несколько раз, а читабельность программы значительно снизилась бы.

Проведённые исследования показывают перспективность использования расширяемых языков для программирования высокопроизводительных вычислений. Однако существующие в настоящее время языки не являются для этого достаточно гибкими. Для получения реальных преимуществ потребуется разрабатывать новые расширяемые языки, а для этого требуется исследовать возможность решения существующих задач при помощи языка Nemerle с целью выработки требования к новым инструментам. Что и является основным направлением дальнейшей работы.

Статья рекомендована к публикации программным комитетом международной научной конференции «Параллельные вычислительные технологии 2011».

Литература

1. TOP 500 List — November 2010. URL: http://top500.org/list/2010/11/100 (дата обращения 13.02.2011).
2. NVidia Corporation. NVidia CUDA C Programming Guide, Version 3.2.
3. Khronos Group. The OpenCL Specification, version 1.1, document revision 33. URL: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf (дата обращения 13.02.2011).
4. The Portland Group. PGI Accelerator Programming Model for Fortran & C, version 1.3. URL: http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf (дата обращения 13.02.2011).
5. Caps Enterprise. CAPS HMPP Workbench User Guide, version 2.3.1.
6. Wilson, G.V. Extensible Programming for the 21st Century / G.V. Wilson // ACM Queue. – January 2005. – V. 2. – P. 48 – 57.
7. Seibel, P. Practical Common Lisp / P. Seibel. – Apress, 2005.
8. Nemerle Homepage. URL: http://nemerle.org/ (дата обращения 13.02.2011).
9. Xoc, an extension-oriented compiler for systems programming / R. Cox, T. Bergan, A.T. Clements, F. Kaashoek, E. Kohler // Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII). – Washington, 2008. – P. 244 – 254.
10. Adinetz, A.V. NUDA Project web site / A.V. Adinetz, P. Shvets, V. Sitchikhin. – URL: http://nuda.sf.net/ (дата обращения 13.02.2011).

Андрей Викторович Адинец, кандидат физико-математических наук, мл.н.с., МОТЭФ ОИЯИ, г. Дубна, мл.н.с., НИВЦ МГУ имени М.В. Ломоносова, adinetz@gmail.com.

Поступила в редакцию 2 марта 2011 г.