УДК 681.3

*А.Е. Дорошенко, К.А. Жереб*

# РАЗРАБОТКА ВЫСОКОПАРАЛЛЕЛЬНЫХ ПРИЛОЖЕНИЙ ДЛЯ ГРАФИЧЕСКИХ УСКОРИТЕЛЕЙ С ИСПОЛЬЗОВАНИЕМ ПЕРЕПИСЫВАЮЩИХ ПРАВИЛ

Использование графических ускорителей позволяет достичь высокой производительности, однако требует от разработчика низкоуровневого программирования и знания деталей аппаратной и программной платформы. В работе предложен подход к автоматизации разработки приложений для графических ускорителей, основанный на использовании парадигмы переписывающих правил.

## Введение

В настоящее время параллельное программирование становится все более актуальным при разработке программ для стандартных, широкодоступных компьютеров, а не только для специализированных высокопроизводительных систем. Этому способствует бурное развитие многоядерных процессоров, которые сейчас устанавливаются в большинстве настольных и мобильных компьютеров [1]. Но есть еще одно направление параллельного программирования, которое получило особенное развитие в последнее время. Речь идет о программировании общецелевых задач для графических ускорителей [2].

Рыночные требования привели к бурному развитию графических ускорителей, в результате чего их вычислительная мощность на данный момент значительно превышает возможности обычных процессоров. Поэтому возник интерес к использованию графических ускорителей для решения задач, не связанных напрямую с обработкой графики [3–4]. Первоначально для этого использовались средства программирования графических задач, не приспособленные для произвольных вычислений. В последнее время интерес к использованию графических ускорителей в качестве средств высокопроизводительных вычислений поддерживается усилиями ведущих разработчиков аппаратуры. Так, компания NVidia представляет платформу для вычислений на графическом ускорителе CUDA [5]. Аналогично, компания AMD выступила с инициативой Stream [6].

Такие платформы облегчают реализацию различных задач на графических ускорителях, поскольку предоставляют модель программирования, которая более приспособлена к разработке произвольных вычислений, по сравнению со средствами программирования графики. Тем не менее, разработка приложений для графических ускорителей остается достаточно сложной задачей. Разработчик должен быть знаком с устройством графического ускорителя, понимать особенности работы его компонент. Модель программирования, предоставляемая выбранной платформой, является новой по сравнению как с последовательным программированием, так и с многопоточным программированием для многоядерных процессоров. Многие задачи требуют достаточно низкоуровневой реализации и оптимизации. Все это приводит к необходимости автоматизации процесса разработки.

В данной работе предложен подход к автоматизации разработки приложений для графических ускорителей, на примере платформы CUDA. Для этого используется система переписывающих правил Termware [7–9]. Предложенный подход позволяет как переходить от последовательных программ к параллельным, выполняющимся на графическом ускорителе, так и оптимизировать программы для CUDA. При этом возможно распараллеливание программ, изначально написанных на языке

C# для платформы Microsoft .NET, с сохранением возможности использования всех средств платформы .NET.

Данная работа продолжает исследования, начатые в работах [10–11], по автоматизации процесса проектирования и разработки эффективных параллельных программ. Особенность данной работы – использование новой аппаратной платформы для параллельных вычислений, графических ускорителей, что позволяет добиться значительного повышения производительности по сравнению с использованием многоядерных процессоров общего назначения.

Область исследований, связанная с автоматизацией разработки приложений для графических процессоров, в настоящее время активно развивается. При этом рассматриваются как задачи перехода от последовательных к параллельным программам, так и задачи оптимизации существующих параллельных программ с использованием возможностей графических ускорителей. Так, в работе [12] рассмотрен автоматический переход от многопоточной программы, реализованной с использованием технологии OpenMP [13], к реализации данной программы на платформе CUDA. Работа [14] описывает платформу для оптимизации циклов в программах для графических ускорителей. Разработаны системы для автоматического распараллеливания и оптимизации программ из конкретной предметной области, например, data mining [15] или обработка изображений [16]. Авторы работы [17] описывают библиотеку высокоуровневых структур данных для графических ускорителей. Также разрабатываются платформы программирования графических ускорителей, предоставляющие более высокоуровневые средства по сравнению с CUDA, такие как hiCUDA [18] и BSGP [19].

В отличие от существующих работ по данной тематике, в данной работе рассмотрена автоматизация перехода на платформу CUDA с высокоуровневого языка C#. Это позволяет использовать возможности платформы Microsoft .NET, которая в настоящее время широко используется для разработки приложений в различных областях. Кроме того, использование переписывающих правил для описания распараллеливающих и оптимизирующих преобразований позволяет легко добавлять новые преобразования.

Материал данной работы организован следующим образом. В разделе 1 описаны особенности платформы CUDA, влияющие на разработку приложений для этой платформы. В разделе 2 описан автоматизированный процесс распараллеливания программы с целью исполнения на графическом ускорителе. Раздел 3 содержит некоторые примеры оптимизирующих преобразований, автоматизированные с помощью платформы Termware. Работу завершают выводы и направления дальнейшей работы.

## 1. Особенности платформы CUDA

**1.1. Аппаратные составляющие.** Современные графические ускорители поддерживают высокую степень параллелизма, специально приспособленную для выполнения графических задач. Использование возможностей графических ускорителей для общецелевых вычислений требует от разработчика понимания особенностей аппаратной платформы и модели программирования CUDA [5].

Общая схема устройства графического ускорителя показана на рис. 1. Графическое устройство (device) содержит несколько мультипроцессоров, а также общую для них графическую память. Каждый мультипроцессор содержит несколько вычислительных ядер (скалярных процессоров), а также один управляющий блок, поддерживающий многопоточное исполнение. В результате количество вычислительных ядер (а значит, и степень возможного параллелизма) оказывается существенно выше, чем у общецелевых многоядерных процессоров.

Графические ускорители поддерживают несколько разных видов памяти, отличающихся по объему, скорости доступа и особенностям реализации. Самая быстрая память – регистры вычислительных ядер; однако их количество в каждом ядре ограничено.

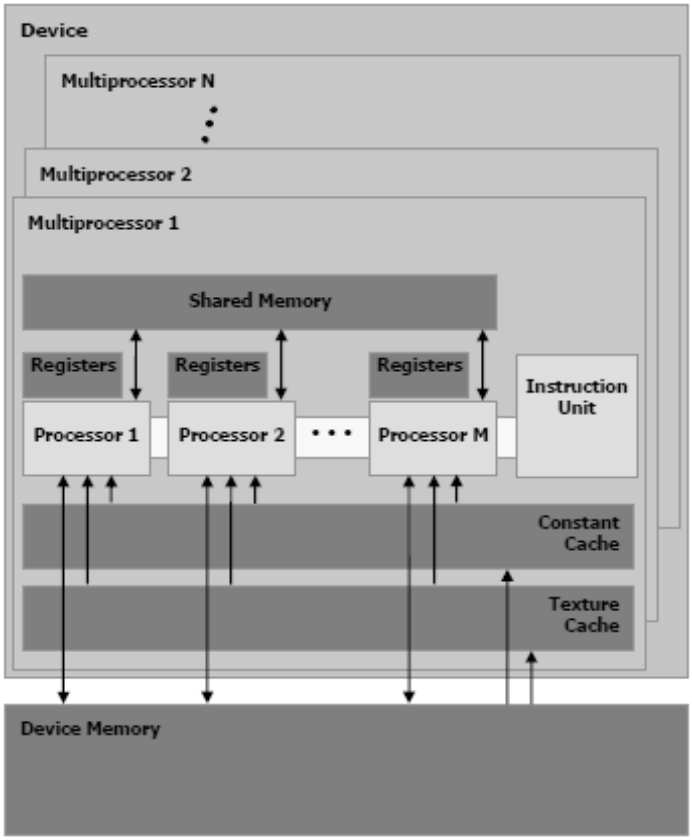


Рис. 1. Основные аппаратные компоненты графического ускорителя

Кеш данных, или разделяемая память (shared memory) поддерживает произвольный доступ из любого вычислительного блока, однако имеет ограниченный размер и требует явной синхронизации доступа. Есть также два специфических вида памяти – кеш констант и кеш текстур. Они поддерживают только чтение, но имеют больший объем по сравнению с shared памятью. Разница между текстурной и константной памятью заключается в том, что текстурная память поддерживает специальные режимы доступа, полезные для графических задач. Всем мультипроцессорам доступна также графическая память устройства. Она является самой большой по объему и самой медленной памятью, доступной графическому ускорителю.

**1.2. Структура параллельной программы.** Параллельная программа для графических ускорителей состоит из большого количества потоков. Особенности аппаратного обеспечение, а именно большое количество вычислительных ядер, позволяют использовать очень мелкозернистый параллелизм, вплоть до выделения отдельного потока для каждого элемента данных.

Для исполнения на графическом ускорителе потоки объединяются в блоки. Каждый блок выполняется на одном мультипроцессоре. Различные блоки, соответствующие одной или нескольким программам, по возможности распределяются равномерно по доступным мультипроцессорам.

При исполнении на мультипроцессоре, потоки объединяются в так называемые warp’ы по 32 потока. На каждом шаге многопоточный планировщик выбирает один из доступных warp’ов. Далее очередная инструкция для потоков этого warp’а выполняется одновременно на вычислительных ядрах. Следует отметить, что наличие ветвлений в потоках понижает производительность, поскольку каждый из вариантов исполнения в таких случаях выполняется последовательно, а остальные потоки warp’а ожидают выполнения.

**1.3. Программная модель CUDA.** Программа для графического ускорителя,

написанная на CUDA, разделяется на две части: код для исполнения на графическом устройстве (device code) и код для исполнения на обычном процессоре (host code). Код для исполнения на устройстве представлен в виде ядер (kernel). Ядра могут быть написаны как на специальном низкоуровневом языке (PTX), так и на расширении языка C (C for CUDA). В последнем случае ядро имеет вид функции языка C, описывающей поведение одного потока.

Платформа CUDA предлагает два разных способа вызова ядер. Более простой способ, C for CUDA, является расширением языка C. Вызов ядра при этом похож на вызов функции C, но с передачей дополнительных параметров. Эти параметры определяют размер блока, а также количество одновременно исполняемых блоков в сетке (grid). Каждый поток имеет доступ к своему номеру в блоке и номеру блока в сетке, что позволяет различным потокам работать с разными данными.

Более сложным способом вызова ядер является driver API. В этом случае ядра необходимо предварительно скомпилировать в бинарный формат, после чего используются функции для загрузки и выполнения ядер. Driver API является более сложным в разработке и может привести к появлению ошибок, так как требует ручного выполнения многих действий, которые автоматизированы в C for CUDA. Пример таких действий – передача параметров ядра. С другой стороны, driver API дает разработчику больший контроль над выполнением программы.

Оба способа вызова CUDA предоставляют функции для работы с графической памятью. Обычно работа программы состоит из таких шагов: инициализация структур данных в графической памяти, копирование из обычной памяти в графическую входных данных, вызов ядра (или нескольких ядер, или многократный вызов одного ядра), копирование результатов из графической памяти, освобождение графической памяти.

**1.4. Программирование графического ускорителя из .NET.** Платформа CUDA на данный момент поддерживает только язык C, поэтому для использования возможностей графического ускорителя из .NET программ необходимы дополнительные компоненты. К подобным компонентам относится библиотека CUDA .NET [20]. Эта библиотека позволяет .NET коду выполнять ядра CUDA, используя возможности, близкие к driver API.

Таким образом, для использования CUDA .NET необходимо сначала реализовать ядро CUDA, что делается средствами C for CUDA. После этого необходимо написать C# код для вызова ядра. Так как CUDA .NET использует более сложный driver API, от разработчика требуется корректная реализация многих низкоуровневых деталей. В частности, при передаче параметров необходимо иметь представление об их размещении в памяти и корректно передать размеры и относительные смещения параметров. Необходимость выполнения этих, довольно сложных, операций вручную затрудняет разработку и повышает вероятность ошибки.

Кроме CUDA .NET, для разработки приложений для .NET можно использовать также проект Accelerator от Microsoft Research [21]. Этот проект предоставляет набор функций, которые вызываются из C# и выполняются на графическом ускорителе. Он не требует низкоуровневой разработки на языках, не принадлежащих к семейству .NET. Однако его возможности существенно ограничены, так как любой алгоритм необходимо представить в виде комбинации доступных функций. Это затрудняет использование Accelerator во многих практических классах задач. Кроме того, проект более не развивается и поэтому не поддерживает новейшие достижения разработчиков графических ускорителей.

## 2. Переход к вычислениям на графическом ускорителе

**2.1. Общая схема преобразования.** В данной работе используется метод автоматизированного распараллеливания кода для выполнения на графическом процессоре, основанный на использовании переписывающих правил. В качестве входных данных для такого преобразования используется последовательный код на языке C#. Этот код преобразуется в представление в

виде термов с использованием анализатора (парсера) языка C#. В качестве входных данных может использоваться также формальное представление алгоритма, например, из системы ИПС [10], однако этот случай в работе не рассматривается.

В исходной (последовательной) программе выделяются фрагменты, в которых происходят интенсивные вычисления. Такие фрагменты целесообразно исполнять на графическом процессоре, что позволит существенно повысить производительность программы в целом. При этом обычно такой код составляет лишь незначительную часть всего кода программы.

После выделения перспективных участков кода к ним применяются разработанные правила Termware, которые переводят последовательный код в параллельный код для исполнения на графическом ускорителе. Преобразованный код состоит из двух частей. Код, описывающий алгоритм, переводится в представление, соответствующее языку C for CUDA. Этот код представлен в виде ядра CUDA, сохраняется в файл с расширением .cu и компилируется в бинарный формат (.cubin) с использованием компилятора NVCC от NVidia. После генерации кода для CUDA возможно также применение оптимизирующих преобразований.

Кроме алгоритмического кода, генерируется служебный код на C#, который вызывает сгенерированное ядро CUDA с использованием средств CUDA .NET. Результатом преобразования является программа, преобладающая часть кода которой по-прежнему реализована на C# и может использовать все возможности платформы .NET. При этом критические по производительности участки программы исполняются на графическом ускорителе с повышенной производительностью.

**2.2. Система Termware.** Для осуществления всех преобразований используется платформа переписывающих правил Termware, подробно описанная в [7–11]. Termware предназначена для описания преобразования над термами, т. е. выражениями вида $f(t_1, \ldots, t_n)$. Для задания преобразований используются правила Termware, т. е. конструкции вида `source [condition] -> destination [action]`.

Здесь `source` – исходный терм (образец для поиска), `condition` – условие применения правила, `destination` – преобразованный терм, `action` – дополнительное действие при срабатывании правила. Каждый из 4 компонентов правила может содержать переменные (которые записываются в виде `$var`), что обеспечивает общность правил. Компоненты `condition` и `action` являются необязательными. Они могут исполнять произвольный процедурный код, в частности использовать дополнительные данные о программе.

Применение правила происходит следующим образом: сначала находится подтерм входного терма (дерева программы), который подходит под `source`. Далее проверяется условие применения (если оно присутствует). Если условие выполняется, происходит замена `source` на `destination`. При этом переменные в destination заменяются соответствующими значениями из source. Также выполняется действие `action` (если оно присутствовало).

Каждое преобразование задается системой правил, т.е. набором правил, которые последовательно применяются к данному терму (дереву программы). Порядок применения правил определяется стратегией. В систему Termware встроены несколько основных стратегий, таких как TopDown, BottomUp, FirstTop. Кроме того, возможно создание дополнительных стратегий.

**2.3. Пример распараллеливания.** В качестве примера использования описанного подхода рассмотрим задачу распараллеливания алгоритма битонической сортировки (bitonic sort, [22]). Последовательная реализация алгоритма имеет следующий вид:

```
for (k = 2; k <= ARR_SIZE; k <<= 1)
{
 for (j = k >> 1; j > 0; j = j >> 1)
 {
     for (int i = 0; i < ARR_SIZE;
i++)
     {
         int ixj = i ^ j;
         if ((ixj) > i)
         {
             if ((i & k) == 0 &&
```

```
toSort[i] > toSort[ixj])
            {
                swap(toSort,i,ixj);
            }
            if ((i & k) != 0 &&
toSort[i] < toSort[ixj])
            {
                swap(toSort,i,ixj);
            }
        }
    }
 }
}
```

Для обработки этого кода средствами Termware он сначала преобразуется в терм с помощью анализатора языка C#. При этом создается следующий терм:

```
For(Assignment(k,2),k<=ARR_SIZE,
Assignment(k,k<<1),
[For(Assignment(j,k>>1),
j>0,Assignment(j,j>>1),
[For(Assignment(i,0),i<=ARR_SIZE,
Increment(i),
[DeclarationAssign-
ment(ixj,int,bit_xor(i,j)),
 If(ixj>i,[
     If(And(Equals(i      &      k,0),
ArrayElement(toSort,i) >
ArrayElement(toSort,ixj)),
[MethodCall(swap,toSort,i,ixj)]),
     If(And(NotEquals(i    &    k,0),
ArrayElement(toSort,i) <      ArrayE-
lement(toSort,ixj)),      [MethodCall
(swap,toSort,i,ixj)])
])])])])
```

**2.4. Выбор участка кода для распараллеливания.** Первым шагом перехода от последовательной программы *K* параллельной программе с использованием графического ускорителя является выбор критичных по производительности участков кода, которые будут распараллелены. Во многих случаях такие участки определяются циклической конструкцией (цикл типа for или while) или вызовом метода.

Выбор кода для распараллеливания может осуществляться по-разному в зависимости от специфики задачи. Возможен выбор необходимого кода вручную, на основании знаний и опыта разработчика. Возможно также использование внешнего инструментария, такого как профилировщик. В этих случаях информация о месте применения распараллеливающих преобразований задается разработчиком через пользовательский интерфейс. Возможно указание терма, к которому применяется преобразование, или непосредственно участка кода (в этом случае система определяет необходимый терм автоматически, используя информацию о связях между термами и элементами кода). Возможна также автоматизированная оценка сложности различных участков алгоритма, описанная в работе [11].

Независимо от способа выбора кода для распараллеливания, этот код помечается специальной меткой, для использования в дальнейших правилах. Так, для цикла for это сводится к добавлению подтерма-метки _MARK_CUDA к терму, обозначающего циклическую конструкцию. Эта метка указывает место применения дальнейших правил, а также задает имя, идентифицирующее данный участок кода (на случай, если преобразование можно применить в нескольких местах). После добавления метки терм приобретает вид

```
For($init, $condition, $step, $body,
_MARK_CUDA(bitonic_kernel))
```

В примере алгоритма битонической сортировки кандидатом для распараллеливания является внутренний цикл (по i). Внешние циклы имеют более сложную структуру, что препятствует их распараллеливанию средствами CUDA.

**2.5. Структура преобразования.** После добавления метки, выделяющей участки кода для преобразования, становится возможным применение правил Termware для перехода к параллельному коду. Первое правило, которое применяется в данном случае, задает общую структуру преобразования: перевод алгоритмического кода в ядро CUDA, а также генерацию служебного кода для вызова этого ядра средствами CUDA .NET. Это правило имеет следующий вид:

```
For(Assignment($idx,0),$idx<$size,
Increment($idx),
$body,_MARK_CUDA($name))            ->
[_InitCuda,_InitCudaKernel($name),
_CallCudaKernel($name,$size,$params)]
[GetExtraParams($body,$params,$idx);
GenerateKernel1D($name,      $params,
$body, $idx)]
```

Оно срабатывает для циклов типа for, причем только специального вида – фактически цикл со счетчиком. (Внешние циклы в алгоритме битонической сортировки не имеют такого вида, поэтому к ним данное преобразование неприменимо).

Кроме того, дополнительное условие срабатывания правила – наличие метки _MARK_CUDA.

Срабатывание правила приводит к замене цикла на 3 терма: `_InitCuda` – инициализация платформы CUDA; `_InitCudaKernel` – инициализация данного ядра; и `_CallCudaKernel` – собственно вызов ядра. Эти термы являются заготовками для служебного кода, вызывающего ядро CUDA. Кроме того, используются средства Termware для вызова двух процедурных методов. Метод `GetExtraParams` вычисляет список параметров ядра, т.е. переменных, которые используются в теле цикла, но не определяются в нем. (Этот метод можно было бы реализовать системой правил, однако в данном случае процедурная реализация оказывается более простой и наглядной). Метод `GenerateKernel1D` создает из тела цикла ядро CUDA; этот метод использует отдельную систему правил для перехода от кода на C# к коду на C for CUDA.

После срабатывания данного правила происходит промежуточный этап – перемещение созданных термов-заготовок для служебного кода. Терм `_CallCudaKernel` никуда не перемещается, поскольку он должен находиться в том же месте в коде, где был исходный цикл. Однако термы `_InitCuda` и `_InitCudaKernel` должны использоваться однократно, в начале программы. Поэтому используется система правил, которая переводит эти термы в начало программы (т.е. в начало метода Main или специальный метод, отведенный для инициализации). Кроме того, эта система устраняет повторения: если распараллеливание происходит в нескольких местах кода, дубликаты термов `_InitCuda` и `_InitCudaKernel` устраняются. В результате остается один терм `_InitCuda`, за которым следуют термы `_InitCudaKernel` (по одному для каждого уникального ядра).

**2.6. Преобразование алгоритмического кода в код для CUDA.** Как уже упоминалось, для преобразования алгоритмического кода в ядро CUDA используется метод `GenerateKernel1D ($name,$params,$body,$idx)`. Этот метод реализован на C# и вызывается средствами Termware как действие (action) при срабатывании правила. Метод сводится к созданию терма, соответствующего новому файлу и вызову системы правил для создания содержимого этого файла.

Имя метода содержит суффикс 1D, что указывает на использование одномерных структур данных. Параметрами метода являются: `$name` – имя ядра; `$params` – список параметров ядра; `$body` – исходное тело цикла, которое преобразуется в код ядра и `$idx` – исходный индекс цикла.

В ходе работы метода создается новый исходный файл в проекте, с именем, совпадающим с именем ядра и расширением .cu (исходный файл C for CUDA). В этот файл добавляются необходимые для CUDA файлы заголовков, а затем – код ядра, который получается преобразованием исходного терма, `_CudaKernel1D($name, $params, $body, $idx)` с использованием разработанной системы правил. В самом общем виде эта система имеет следующий вид:

```
1. _CudaKernel1D($name,   $params,
   $body,         $idx)        ->
   _CudaFunctionKernel($name,
   $params,
   [_PrepareIdx($idx,x),_CsToCuda(
   $body)])
2. _CudaFunctionKernel($name,
   $params,       $body)       ->
   Function([ExternC,__global__],
   void,$name,$params,$body)
3. _PrepareIdx($idx,$coor) ->  De-
   clarationAssignment(
   $idx,int,_CudaIdx($coor))
4. _CudaIdx($coor)              ->
   Dot(blockIdx,$coor)           *
   Dot(blockDim,$coor)           +
   Dot(threadIdx,$coor)
```

Первое правило задает общую структуру ядра – специальное описание функции, с телом, состоящим из определения индекса, за которым следует тело исходного цикла. Правило 2 реализует описание функции, которая является ядром CUDA. Эта функция имеет модификатор `__global__` , не возвращает значения (void), а ее имя и список параметров определяется данными, переданными в метод `GenerateKernel1D`. Кроме того, добавляется модификатор `extern "C"` , необходимый для вызова ядра из CUDA .NET. (Вы-

зов осуществляется по имени ядра, и использование такого модификатора требует от компилятора сохранить имя в простом виде, без добавления информации, характерной для C++). Правила 3 и 4 определяют индекс исходного массива из параметров запуска ядра (индекс блока в сетке и потока в блоке). Поскольку в данном случае ядро работает с одномерным массивом, единственный индекс определяется соотношением

```
i = blockIdx.x * blockDim.x +
threadIdx.x;
```

Вышеприведенная общая система правил дополняется специфическими правилами, которые переводят код из C# в C for CUDA. Эти правила работают с термом `_CsToCuda( $body)`. Такие правила могут быть двух видов: технические, которые переводят определенные конструкции C# в соответствующие конструкции C, и оптимизационные, которые вносят изменения в структуру кода с целью повышения производительности. Правила первого типа достаточно простые и одинаковы для различных программ. Особый интерес представляют именно оптимизационные преобразования, которые являются специфичными для каждой задачи. Следует отметить, что оптимизационные преобразования имеет смысл выделить в отдельные системы правил, которые применяются отдельно от процесса распараллеливания. Некоторые примеры оптимизационных преобразований рассмотрены в разделе 3.

Для алгоритма битонической сортировки, в силу его простоты, практически не требуются специальные преобразования (как технические, так и оптимизационные). Можно упомянуть лишь одно необходимое преобразование, связанное с вызовом метода swap в исходном коде. В данном случае этот метод преобразуется в функцию с модификатором `__device__` , которая также выполняется на графическом ускорителе (на самом деле функции такого рода встраиваются в код ядра на этапе компиляции).

В результате работы метода `GenerateKernel1D` получается следующий код ядра:

```
extern "C" __global__ void
bitonic_kernel(int * toSort, int j,
int k)
{
    int i = threadIdx.x + blockDim.x
* blockIdx.x;
    int ixj = i ^ j;
    if ((ixj) > i)
    {
        if ((i & k) == 0 && toSort[i]
> toSort[ixj])
        {
            swap(toSort,i,ixj);
        }
        if ((i & k) != 0 && toSort[i]
< toSort[ixj])
        {
            swap(toSort,i,ixj);
        }
    }
}
```

Далее этот код компилируется в бинарный формат (.cubin) с использованием компилятора NVCC. В таком виде он становится доступным для использования средствами CUDA .NET.

**2.7. Генерация служебного кода.** Служебный код для вызова полученного ядра генерируется из трех термов `_InitCuda`, `_InitCudaKernel` и `_CallCudaKernel`. Первые два терма, отвечающие за инициализацию платформы и данного ядра, генерируют достаточно простой и стандартный код:

```
// Init CUDA, select 1st device.
CUDA cuda = new CUDA(0, true);

// load module, select kernel
string path = Path.Combine (
Environment.CurrentDirectory,
"bitonic_kernel.cubin");
cuda.LoadModule(path);
CUfunction bitonic_kernel =
cuda.GetModuleFunction
("bitonic_kernel");
```

Этот код инициализирует платформу CUDA, выбирает для вычислений первое из доступных графических устройств. Затем загружается сгенерированный бинарный модуль, и из него выбирается (по имени) ядро.

Правила для вызова ядра являются более сложными. Сам вызов ядра состоит из нескольких шагов:

1) копирование входных данных в графическую память;

2) задание параметров ядра;

3) собственно вызов ядра;

4) копирование результатов работы из графической памяти;

5) освобождение структур данных CUDA.

При этом различные шаги выполняются в разных местах программы и различное количество раз. Шаги 2 и 3 выполняются для каждого вызова: в случае задачи битонической сортировки они оказываются внутри двух циклов (по k и по j) и поэтому выполняются многократно. Шаг 1 выполняется после того, как входные данные проинициализированы в C# коде, и до вызова ядра. При этом рассматриваются только массивы данных (`toSort` в алгоритме сортировки), а не единичные переменные (`j` и `k`). Аналогично, шаг 4 выполняется после того как произведены вычисления на графическом ускорителе, до того как их результаты используются в коде на C#.

В общем случае, возможно многократное выполнение шагов 1 и 4 – либо для разных массивов, либо для одного массива, если он используется в коде между вызовами ядра (или различных ядер). Для задачи битонической сортировки реализуется более типичный сценарий, когда массив копируется в графическую память и из нее однократно.

Шаг 5 – освобождение структур данных CUDA – происходит после последнего использования этих структур в шагах 1–4. В примере с сортировкой это происходит сразу после копирования данных из графической памяти (шаг 4).

Таким образом, система правил для генерации вызова ядра устроена следующим образом. Сначала исходный терм `_CallCudaKernel($name,$size,$params)` разбивается на 5 термов:
`_CopyHostToDevice($params)`, `_SetParameters($name,$params)`, `_ExecuteKernel($name,$size)`, `_CopyDeviceToHost ($params)` и `_FreeCuda($params)`. Далее терм `_CopyHostToDevice` поднимается «вверх», т.е. он переставляется местами с термами, в которых не происходит запись в массив данных. В результате он оказывается сразу после записи входных данных в массив. Аналогично, термы `CopyDeviceToHost` и `_FreeCuda` перемещаются вниз.

Далее каждый из термов расшифровывается. Для `_CopyHostToDevice` получается достаточно простой код, поскольку CUDA .NET предоставляет удобные методы для работы с массивами. Полученный код имеет вид

```
CUdeviceptr dev_toSort =
cuda.CopyHostToDevice<int>(toSort);
```

Аналогично термы `CopyDeviceToHost` и `_FreeCuda` преобразуются в следующий код:

```
cuda.CopyDeviceToHost<int>(
dev_toSort, toSort);
cuda.Free(dev_toSort);
```

Терм `_SetParameters` преобразуется в код, который задает значение каждого параметра и его размер в памяти. Размеры параметров вычисляются автоматически, что позволяет избежать ошибок при ручном определении размеров. Сгенерированный код имеет вид:

```
cuda.SetParameter(bitonic_kernel,  0,
(uint) dvalues.Pointer);
cuda.SetParameter(bitonic_kernel,
IntPtr.Size, (uint)j);
cuda.SetParameter(bitonic_kernel,
IntPtr.Size + 4, (uint)k);
cuda.SetParameterSize(bitonic_kernel,
(uint) (IntPtr.Size + 8));
```

В результате сам вызов ядра (терм `_ExecuteKernel`) генерирует достаточно простой код:

```
cuda.SetFunctionBlockShape
(bitonic_kernel, MAX_THREAD, 1, 1);
cuda.Launch(bitonic_kernel, ARR_SIZE
/ MAX_THREAD, 1);
```

В этом коде появляется дополнительный параметр `MAX_THREAD`, определяющий количество потоков в блоке.

Таким образом, использование систем переписывающих правил позволяет полностью автоматизировать процесс создания служебного кода для вызова ядер CUDA средствами CUDA .NET. Тем самым разработчик может концентрировать внимание на особенностях алгоритма, а не на технических средствах его реализации. Кроме того, исключаются ошибки, возможные при создании этого служебного кода вручную.

**2.8. Сравнение производительности.** Для оценки эффекта от применения распараллеливающих преобразований были проведены измерения скорости работы алгоритма битонической сортировки в

двух версиях: исходная реализация (CPU) и распараллеленная с использованием CUDA (GPU). Результаты измерений приведены в табл. 1.

Таблица 1. Сравнение производительности

| Размер | GPU, с | CPU, с | Ускорение |
|---|---|---|---|
| 1048576 | 0.12 | 3.3 | 27.5 |
| 2097152 | 0.26 | 8.98 | 34.53846 |
| 4194304 | 0.5 | 16 | 32 |
| 8388608 | 0.98 | 32.34 | 33 |
| 16777216 | 2.42 | 75 | 30.99174 |

Из результатов видно, что распараллеливание дает существенный эффект: производительность повышается в 30 раз. Отметим, что этот результат достигается при автоматизированном преобразовании, и для его реализации от разработчика не требуется знаний платформы CUDA или CUDA .NET.

## 3. Оптимизация кода для CUDA

**3.1. Особенности оптимизации CUDA.** Для программ, выполняющихся на графическом ускорителе и написанных на CUDA, оптимизация имеет особое значение. Достаточно типичной является ситуация, когда производительность неоптимизированной программы в десятки раз отличается от производительности ее оптимизированного аналога. Так как именно повышение производительности является причиной перехода на вычисления на графическом ускорителе, то оптимизация – важная часть жизненного цикла программ на CUDA.

При этом оптимизация приложений, написанных на CUDA – достаточно сложный процесс и требует понимания архитектуры CUDA, взаимосвязи различных аппаратных компонент графического ускорителя. Сам алгоритм должен поддерживать очень широкий параллелизм, чтобы задействовать возможности современных графических процессоров. Кроме того, его реализация должна учитывать многие мелкие детали, такие как размещение данных в памяти, порядок доступа к ним, использование оптимальных типов данных и операций и т. д.

Учитывая все вышеперечисленные особенности, особую важность приобретает задача автоматизации процесса оптимизации программ для CUDA. В данном разделе описаны примеры автоматизированных преобразований в виде правил Termware, направленных на повышение производительности приложения.

**3.2. Пример: иерархическое параллельное суммирование.** Рассмотрим оптимизирующие преобразования для задачи нахождения суммы большого массива данных. Используется иерархический алгоритм суммирования: на каждом этапе элементы массива разбиваются на пары и суммируются независимо друг от друга. Это позволяет получить высокую степень параллелизма.

Простейшая реализация данного алгоритма на CUDA имеет следующий вид:

```
extern "C" __global__ void
parallelAdd ( int * inData, int *
outData )
{
      int tid = threadIdx.x;
      int i = blockIdx.x * blockDim.x
+ threadIdx.x;
      outData[i]=inData[i];
      __syncthreads ();
      for ( int s = 1; s <
blockDim.x; s *= 2 ) {
      if ( tid % (2*s) == 0 )
      outData [i] += outData [i + s];
      __syncthreads ();
      }
}
```

**3.3. Использование разделяемой (shared) памяти.** Часто производительность программ для CUDA зависит в основном от правильной реализации доступа к памяти, а не от эффективности самих вычислений. В частности, в примере с параллельным суммированием все операции производятся с глобальной памятью (которая реализована на видеопамяти). Ее преимущества заключаются в том, что она доступна всем процессорам, а также с ней можно работать непосредственно из обычного кода. Однако скорость доступа к ней существенно меньше, чем к более быстрым видам памяти.

Более быстрой альтернативой глобальной памяти является так называемая разделяемая (shared) память. Она расположена на каждом процессоре, поэтому ско-

рость доступа к ней максимальна. Ее недостатками являются малый размер (порядка 16К в современных ускорителях), а также то, что доступ к ней возможен лишь в пределах одного блока, при этом требует синхронизации. Тем не менее, если алгоритм позволяет использовать небольшое локальное хранилище данных, целесообразно использовать shared память. (Алгоритм битонической сортировки, рассмотренный в разделе 2, такими свойствами не обладает).

Обычно работа с shared памятью выполняется следующим образом: данные из обрабатываемого участка глобальной памяти копируются в shared память (этот процесс также является параллельным – каждый поток копирует свою часть данных). После этого выполняется синхронизация потоков командой `__syncthreads ()`. Далее вся работа идет уже в shared памяти. Когда необходимо записать результаты в глобальную память, также проводится синхронизация и результаты вычислений по частям копируются из shared в глобальную память.

Для перехода к использованию shared памяти в задаче параллельного суммирования можно использовать следующую систему правил:

1. `Function($x, $y, parallelAdd,$z,[$b1:$b2]) -> Function($x, $y, parallelAdd1, $z,[SharedDecl:$b1:CopyBack($b2)])`
2. `SharedDecl -> ArrayDeclaration (localData,int, BLOCK_SIZE, __shared__)`
3. `Assignment (ArrayElement (outData,i), ArrayElement(inData,i)) -> Assignment (ArrayElement (localData ,tid),ArrayElement(inData,i))`
4. `PlusAssignment(ArrayElement (outData,i),ArrayElement(outData,i+s)) -> PlusAssignment (ArrayElement (localData, tid),ArrayElement (localData, tid+s))`
5. `CopyBack([$x:$y]) -> [$x:CopyBack($y)]`
6. `CopyBack(NIL) -> If (Equals(tid,0),Assignment( ArrayElement(outData,Dot (blockIdx,x)),ArrayElement (localData,0)))`

Правила 1 и 2 добавляют объявление локального массива shared памяти. Правило 3 заменяет копирование напрямую в выходной массив копированием в локальный массив. Правило 4 аналогично заменяет суммирование в глобальной памяти на суммирование в shared памяти. Правила 5 и 6 добавляют в конце ядра копирование в выходной массив результата вычисления.

В результате действия системы правил получается следующее модифицированное ядро:

```
extern "C" __global__ void
parallelAdd1 ( int * inData, int *
outData )
{
      __shared__ int localData
[BLOCK_SIZE];
      int tid = threadIdx.x;
      int i = blockIdx.x * blockDim.x
+ threadIdx.x;
      localData [tid] = inData [i];
      __syncthreads ();
      for ( int s = 1; s <blockDim.x;
s *= 2 ) {
      if ( tid % (2*s) == 0 )
      localData [tid] += localData
[tid + s];
      __syncthreads ();
      }
      if ( tid == 0 )
      outData[blockIdx.x] = localData
[0];
}
```

**3.4. Уменьшение количества ветвлений.** Еще одним недостатком исходной реализации алгоритма параллельного суммирования является большое количество ветвлений. В коде используется условие цикла:

```
if ( tid % (2*s) == 0 )
```

Это условие часто меняется для соседних потоков (в частности, для s=1 все соседние потоки идут по разным ветвям). Поскольку в архитектуре графического ускорителя предусмотрено большее количество вычислительных блоков по сравнению с управляющими, такое ветвление вынуждает запускать потоки последовательно, а не одновременно.

Возможно уменьшение степени ветвления, за счет перераспределения вычислений между потоками. В исходной реализации, вычисления выполнялись на потоках с номерами, кратными степеням

двойки (на первом шаге – четные номера, на втором – кратные 4, и т.д.). Рассмотрим преобразование, которое переносит вычисления на последовательные потоки (на первом шаге – первая половина потоков, на втором – первая четверть и т.д.). Преобразование состоит из единственного правила:

```
If(Equals(tid % (2*s),0),  PlusAs-
signment( ArrayElement (localData,
tid) ,ArrayElement(localData,tid+s)))
-> [DeclarationAssignment(index,int,2
* s * tid),If(index<Dot(blockDim,x),
PlusAssignment( ArrayElement( local-
Data,index), ArrayElement( localData,
index+s)))]
```

Это правило заменяет предыдущее условия на такой код:

```
int index = 2 * s * tid;
if ( index < blockDim.x )
data [index] += data [index + s];
```

**3.5. Конфликты доступа к shared памяти.** После применения предыдущего правила количество ветвлений существенно понижается. Однако теперь недостатком являются конфликты доступа к shared памяти. Особенностью этой памяти является то, что она разделена на 16 банков, при этом доступ соседних потоков к одному банку осуществляется последовательно, а к различным банкам – одновременно. В варианте реализации, полученном в предыдущем пункте, все данные сохраняются в первых элементах массива, которые лежат в одном банке. Поэтому все обращения к элементам массива оказываются последовательными.

Чтобы избежать такой ситуации, можно провести преобразование, заключающееся в изменении порядка суммирования. Ранее на первом этапе суммировались соседние элементы, потом удаленные на 2 и т.д. Перейдем к другому порядку суммирования: на первом этапе будем суммировать максимально удаленные элементы и далее будем уменьшать расстояние между элементами. Для этого используется следующая система правил:

1. 
```
For(DeclarationAssignment(s,int
,1),
s<Dot(blockDim,x),MulAssignment
(s,2), $body) ->
For(DeclarationAssignment(s,int
, Dot(blockDim,x)/2),s>0,
RShiftAssignment(s,1),$body)
```

2. 
```
If(Equals(tid % (2*s),0),$body)
-> If(tid < s,$body)
```

Эта система правил применяется к программе, использующей shared память, из пункта 3.3. Первое правило заменяет параметры цикла: расстояние между суммируемыми элементами уменьшается, а не увеличивается. Второе правило заменяет условие суммирования. Преобразованный код имеет вид:

```
for ( int s = blockDim.x / 2; s > 0;
s >>= 1 )
{
      if ( tid < s )
      localData [tid] += localData
[tid + s];
      __syncthreads ();
}
```

Заметим, что при использовании данного преобразования вводятся две оптимизации: уменьшение конфликтов при доступе к банкам памяти и снижения количества ветвлений. В этом смысле преобразование является более эффективным, чем рассмотренное в п. 3.4. Однако данное преобразование в большей мере использует знания о характере задачи (т.е. о возможности проводить суммирование в другом порядке).

**3.6. Изменение параметров вызова ядра.** Преобразование из п. 3.5 позволило избавиться от ветвлений и конфликтов по доступу к памяти, но при этом понизилась эффективность использования потоков. Половина из них вообще не производит вычислений (поскольку для них не выполняется условие `tid < s`).

Поэтому можно модифицировать ядро, исключив потоки, которые не выполняют вычислений. В новой реализации, первое суммирование производится еще на стадии копирования данных в shared память. Поэтому в нем участвуют все потоки. Правила для выполнения этого преобразования имеют вид:

1. 
```
DeclarationAssignment(
i,int,Dot(blockIdx,x) *
Dot(blockDim,x) +
Dot(threadIdx,x)) ->
DeclarationAssign-
ment(i,int,2*Dot(blockIdx,x) *
Dot(blockDim,x) +
Dot(threadIdx,x))
```

2. 
```
Assignment (ArrayElement (
localData,tid), ArrayElement
(inData,i)) -> Assignment (
```

```
      ArrayElement(localData,tid),
      ArrayElement(inData,i)
      +ArrayElement(inData,i+
      Dot(blockDim,x)))
   3. CallKernel(
      Shape(ARR_SIZE/BLOCK_SIZE,
      BLOCK_SIZE),parallelSum,[inData
      ,outData]) -> CallKernel(
      Shape(ARR_SIZE/(2*
      BLOCK_SIZE),BLOCK_SIZE),
      parallelSum,[inData,outData])
```

Правило 2 определяет, что при копировании в shared память происходит также суммирование. При этом получается, что один блок может обработать вдвое больший объем информации. Поэтому правило 1 задает изменение индекса во входном массиве, а правило 3 – изменение параметров вызова ядра (поскольку каждый блок обрабатывает вдвое больший объем данных, размер сетки уменьшается вдвое). В результате код ядра меняется следующим образом:

```
int i = 2*blockIdx.x * blockDim.x +
threadIdx.x;
localData [tid] = inData [i] + inData
[i+blockDim.x];
```

**3.7. Развертывание цикла и устранение лишних синхронизаций.** Одним из недостатков исходной реализации, который так и не был устранен после всех преобразований, является достаточно большое количество синхронизаций (на каждом шаге суммирования). Для первых итераций эти синхронизации являются необходимыми, так как разные потоки могут исполняться на разных warp'ах. Но на последних итерациях, когда количество вычисляющих потоков не превосходит 32 (размер warp'а), синхронизации происходят автоматически. Поэтому можно избавиться от явного вызова `__syncthreads ()`. Кроме того, в этом случае можно развернуть цикл и убрать ветвление по признаку `tid < s`.

Для этого используется следующая система правил:

```
   1. For(DeclarationAssignment(s,int
      ,Dot(blockDim,x)/2),s>0,
      RShiftAssignment(s,1),$body) ->
      [For(DeclarationAssignment(s,
      int,Dot(blockDim,x)/2),s>32,
      RShiftAssignment(
      s,1),$body),_Unroll($body)]
   2. _Unroll(If(tid<s,$body)) ->
      If(tid<32,_Unroll($body))
   3. _Unroll[$x:$y] ->
      [_Unroll($x):_Unroll($y)]
   4. _Unroll(PlusAssignment($x,$y))
      ->
      _Unroll(PlusAssignment($x,$y),3
      2)
   5. _Unroll(PlusAssignment($x,$y),$
      n) -> [PlusAssignment(
      $x,_Replace($y,$n)),_Unroll(
      PlusAssignment( $x,$y),$n/2)]
      [$n>1]
   6. _Unroll(PlusAssignment($x,$y),1
      ) -> PlusAssignment(
      $x,_Replace($y,1))
   7. _Replace(ArrayElement(localData
      ,tid+s),$n) -> ArrayElement(
      localData,tid+$n)
   8. _Unroll(SyncThreads) -> NIL
```

В этой системе правило 1 заменяет предел цикла с 0 до 32, а также добавляет за циклом заготовку для развернутого варианта. Правило 2 удаляет проверку `tid < s` и добавляет проверку на срабатывание развернутого цикла `tid < 32`. Правила 3–7 определяют замену тела цикла (суммирование) на развернутый вариант, который повторяется для значений 32, 16, 8, 4, 2, 1. В результате правило 8 удаляет явные вызовы синхронизационной функции.

После применения правил код ядра принимает следующий вид:

```
extern "C" __global__ void
parallelAdd5 ( int * inData, int *
outData )
{
      __shared__ int localData
[BLOCK_SIZE];
      int tid = threadIdx.x;
      int i = 2*blockIdx.x *
blockDim.x + threadIdx.x;
      localData [tid] = inData [i] +
inData [i+blockDim.x];
      __syncthreads ();
      for ( int s = blockDim.x / 2; s
> 32; s >>= 1 )
      {
            if ( tid < s )
            localData [tid] +=
localData [tid + s];
            __syncthreads ();
      }
      if ( tid < 32
      {
            localData [tid] +=
localData [tid + 32];
            localData [tid] +=
localData [tid + 16];
            localData [tid] +=
localData [tid + 8];
            localData [tid] +=
localData [tid + 4];
            localData [tid] +=
```

```
localData [tid + 2];
            localData [tid] +=
localData [tid + 1];
      }
      if ( tid == 0 )
      outData [blockIdx.x] =
localData [0];
}
```

Заметим, что для `tid > 0` некоторые вычисления в развернутом цикле являются избыточными, поскольку используется в конечном счете только значение `localData[0]`. Это произошло из-за того, что было исключено условие `tid < s`. Для обычных платформ программирования такие избыточные вычисления привели бы к понижению производительности. Однако для графического ускорителя это не происходит: из-за большого количества вычислительных блоков можно выполнять даже ненужные вычисления, если это упрощает работу управляющего блока как и происходит в данном случае.

**3.8. Сравнение производительности.** Для оценки эффективности различных преобразований приведем данные сравнения производительности различных вариантов реализации. Измерялось время 100-кратного исполнения алгоритма. Также для сравнения приведено время исполнения последовательной версии алгоритма. Результаты приведены в табл. 2.

Таблица 2. Сравнение различных способов оптимизации

| Вариант | Время исполнения, мс |
|---|---|
| Исходный | 625 |
| Shared память | 300 |
| Без ветвлений | 186 |
| Без конфликтов | 77 |
| Без простаивания | 44 |
| Развертывание | 30 |
| Последовательный | 578 |

Из результатов видно, что исходная реализация была весьма неэффективной (и даже менее производительной, чем последовательная версия). Каждое из преобразований давало вполне ощутимый эффект по повышению производительности. В итоге после применения всех преобразований получили программу, которая в 20 раз эффективнее исходной.

## Выводы

В работе рассмотрен подход к автоматизации процесса разработки приложений для графических ускорителей, основанный на использовании системы переписывающих правил Termware. Рассмотрены преобразования, которые приводят как к распараллеливанию программ с использованием платформы CUDA, так и к оптимизации параллельных программ для графических ускорителей. Применение разработанных преобразований дает существенный эффект: производительность приложений повышается в десятки раз. При этом автоматизированное осуществление преобразований позволяет упростить разработку и понизить вероятность ошибки разработчика.

Дальнейшие исследования в данном направлении предполагают разработку дополнительных преобразований для распараллеливания и оптимизации кода для графических ускорителей, а также оценку их эффективности на различных примерах. Также предполагается поддержка других технологий программирования для графических ускорителей, таких как AMD Stream.

Авторы выражают благодарность Н. Котюку за помощь в проведении вычислительных экспериментов.

1. *Эндрюс Г.* Основы многопоточного, параллельного и распределенного программирования. – М.: ИД ”Вильямс”, 2003. – 512 с.
2. *Ryoo S., Rodrigues C.I., Baghsorkhi S.S., Stone S.S., Kirk D.B., and Hwu W.W.* Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Salt Lake City, UT, USA, February 20–23, 2008). PPoPP '08. ACM, New York, NY. – P. 73–82.
3. *Fatahalian K., Sugerman J., and Hanrahan P.* Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware, 2004. – P. 133–137.

4. *General-Purpose* Computation Using Graphics Hardware. http://www.gpgpu.org .
5. *NVidia* CUDA technology. http://www.nvidia.com/cuda .
6. *AMD* (ATI) Stream technology. http://www.amd.com/stream .
7. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications, Fundamenta Informaticae. – 2006. – Vol. 72, N 1–3. – P. 95–108.
8. *TermWare*. – http://www.gradsoft.com.ua/products/termware_rus.html .
9. *Дорошенко А.Е., Шевченко Р.С.* Система символьных вычислений для программирования динамических приложений // Проблемы программирования. – 2005. – № 4. – С. 718–727.
10. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Формализованное проектирование эффективных многопоточных программ // Проблемы программирования. – 2007. – № 1. – С. 17–30.
11. *Дорошенко А.Е., Жереб К.А., Яценко Е.А.* Об оценке сложности и координации вычислений в многопоточных программах // Проблемы программирования. – 2007. – № 2. – С. 41–55.
12. *Lee S., Min S., and Eigenmann R.* OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Raleigh, NC, USA, February 14–18, 2009). PPoPP '09. ACM, New York, NY. – P. 101–110.
13. *OpenMP* specification. http://openmp.org/wp/ .
14. *Baskaran M., Bondhugula U., Krishnamoorthy S., Ramanujam J., Rountev A., and Sadayappan P.* A compiler framework for optimization of affine loop nests for gpgpus. In Proceedings of the 22nd Annual international Conf. on Supercomputing (Island of Kos, Greece, June 07–12, 2008). ICS '08. ACM, New York, NY. – P. 225–234.
15. *Ma W. and Agrawal G.* A compiler and runtime system for enabling data mining applications on gpus. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Raleigh, NC, USA, February 14 – 18, 2009). PPoPP '09. ACM, New York, NY. – P. 287–288.
16. *Allusse Y., Horain P., Agarwal A., and Saipriyadarshan C.* GpuCV: an open source GPU-accelerated framework for image processing and computer vision. In Proceeding of the 16th ACM International Conf. on Multimedia (Vancouver, British Columbia, Canada, October 26–31, 2008). MM '08. ACM, New York, NY. – P. 1089–1092.
17. *Lefohn A.E., Sengupta S., Kniss J., Strzodka R., and Owens J.D.* Glift: Generic, efficient, random-access GPU data structures. ACM Trans. Graph. 25, 1 Jan. 2006. – P. 60–99.
18. *Han T.D. and Abdelrahman T.S.* hiCUDA: a high-level directive-based language for GPU programming. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (Washington, D.C., March 08 – 08, 2009). GPGPU-2, Vol. 383. ACM, New York, NY. – P. 52–61.
19. *Hou, Q., Zhou, K., and Guo, B.* BSGP: bulk-synchronous GPU programming. In ACM SIGGRAPH 2008 Papers (Los Angeles, California, August 11 – 15, 2008). SIGGRAPH '08. ACM, New York, NY. – P. 1–12.
20. *CUDA* .NET http://www.gass-ltd.co.il/en/products/cuda.net/ .
21. *Microsoft* Research Accelerator Project http://research.microsoft.com/en-us/downloads/648909e1-cb85-46c4-9a94-3cca55971b1d/ .
22. Bitonic Sort Algorithm. http://www.itl.nist.gov/div897/sqg/dads/HTML/bitonicSort.html .

Получено 12.06.2009

***Об авторах:***

*Дорошенко Анатолий Ефимович*,
доктор физико-математических наук,
профессор, заведующий отделом теории
компьютерных вычислений Института
программных систем НАН Украины,

*Жереб Константин Анатольевич*,
аспирант Физико-технического учебно-
научного центра НАН Украины.

***Место работы авторов:***

Институт программных систем
НАН Украины,
03680, Киев-187,
Проспект Академика Глушкова, 40.
Тел.: (044) 526 1538.
e-mail: dor@isofts.kiev.ua

Физико-технический учебно-научный
центр НАН Украины,
03142, Киев-142,
Бульвар Вернадского, 36.
Тел.: (044) 424 3025.
e-mail: zhereb@gmail.com