ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 4(22), 2014, с. 3–17

УДК 004.382.2, 004.432

Ю. П. Сердюк

Программирование графических процессоров (GPU) на языке MC#

Аннотация. В статье рассматриваются базовые принципы и средства программирования графических процессоров (graphical processor units – GPU) на языке MC#, являющимся расширением языка C#. Приводится пример программы на языке MC#, предназначенной для исполнения на GPU, и разбираются средства задания конфигурации GPU, средства работы с разделяемой памятью и использование CUDA-средств в MC#-программах. В заключение, перечисляются нерешенные вопросы и пути дальнейшего развития системы программирования MC# для GPU.

Ключевые слова и фразы: Параллельное программирование, графические процессоры, разделяемая память..

Введение

Язык параллельного программирования MC# [1] является расширением объектно-ориентированного языка C# и предназначен для разработки приложений, исполняющихся как на многоядерных процессорах, так и на вычислительных системах с распределенной памятью (кластерах). В частности, он поддерживает разработку приложений для гибридных систем на базе GPU (graphical processor unit) – систем, включающих в себя основной процессор и ускорители на основе GPU компании Nvidia. При этом, разработка приложений проводится исключительно с помощью специфических средств языка MC# и не требует применения дополнительных инструментов или библиотек, таких как OpenMP, MPI, OpenCL и т.п.

Расширение языка MC# для поддержки программирования графических процессоров [2] лежит в рамках единой модели параллельного программирования, принятой в этом языке. В частности, к *async*-методам, которые предназначены для исполнения на отдельных ядрах многоядерного процессора, и *movable*-методам, которые

предназначены для исполнения на отдельных узлах кластера, в расширении языка MC# для GPU добавляются так называемые *gpu*-методы – методы, которые предназначены для исполнения на графическом процессоре (о *gpu*-методах см. раздел 2 данной статьи).

Общая идеология программирования графических процессоров на языке MC# совпадает с идеологией технологии CUDA, знание которой предполагается для программирования GPU на языке MC#. В частности, перед вызовом gpu-метода программист, в соответствии с технологией CUDA, должен определить параметры конфигурации графического процессора – задать номер графического устройства, размеры решетки и блоков вычислительных потоков и т.п. В программах на языке MC# это реализуется путем создания экземпляра объекта класса GpuConfig и задания его свойств. Класс *GpuConfig* и работа с ним описаны ниже в разделе 2.

Специфические средства CUDA, которые возможно использовать в gpu-методах, представлены в разделе 3 данной статьи. Так, например, в этом разделе демонстрируется, как можно использовать разделяемую память (shared memory) в MC#-программах, предназначенных для исполнения на GPU, а также даны сведения о библиотеке GPUMath – библиотеке математических функций, которые можно использовать в *gpu*-методах.

Составной частью системы программирования MC#, отвечающей за поддержку графических процессоров, является библиотека GPU.NET, реализованная полностью на языке C# и включающая в себя:

(1) JIT (Just-In-Time)-компилятор для GPU компании Nvidia,
(2) набор функций, соответствующих базовым функциям библиотеки CUDA.

Также, за поддержку GPU отвечает компилятор языка MC#, который распознает в тексте программы *gpu*-методы и порождает код, необходимый для вызова соответствующих функций непосредственно на GPU.

Использование языка MC# для программирования графических процессоров значительно упрощает их использование по сравнению с применением базовой технологии CUDA. В частности, при программировании на языке MC# программист освобожден от необходимости явно программировать передачу данных из основной памяти в память графического устройства и обратно – эта задача решается компилятором языка MC#, который генерирует вызовы соответствующих

функций библиотеки GPU.NET, реализующих копирование данных. Аналогичный механизм, получивший название Unified Memory [3], появился только в последних версиях библиотеки CUDA.

Поскольку все составные части системы программирования MC#, включая компоненты, поддерживающие GPU, написаны на языке C#, то программы на языке MC# могут исполняться как под ОС Windows, так и под ОС Linux, где, в последнем случае, в качестве реализации платформы .NET используется свободно доступная программная система Mono (www.mono-project.com).

Систему программирования MC# возможно интегрировать в систему разработки Microsoft Visual Studio, что позволяет в рамках последней разрабатывать и исполнять MC#-программы для GPU. При этом, как и в случае ОС Windows, так и ОС Linux, предполагается наличие на машине установленной системы CUDA.

Графическими процессорами, поддерживаемыми в системе программирования MC#, являются все типы GPU компании Nvidia, включая последние модели Kepler K20 и K40.

1. Пример программирования GPU на языке MC#

Базовая структура программы на языке MC#, предназначенной для исполнения на графическом процессоре, состоит из:

(1) описания конфигурации графического процессора, в котором, в частности, задается количество (параллельных) потоков, запускаемых на GPU, и параметры их объединения в блоки и решетку, и
(2) *gpu*-функции (метода), которая будет исполняться в рамках одного вычислительного потока на GPU.

Gpu-метод в программе задается путем приписывания его определению модификатора *gpu* синтаксически располагающегося на месте типа возвращаемого значения (см. ниже пример *gpu*-функции *vecadd*). Определение конфигурации GPU производится с помощью создания объекта класса *GpuConfig* и задания его параметров. Ниже представлен полный текст простой программы на языке MC#, предназначенной для сложения двух векторов целых чисел с использованием GPU. В этой программе, исходные векторы A и B, а также результирующий вектор C имеют длину N. Данное число служит размером (одномерного) блока потоков, запускаемых на GPU – соответственно, i-ый поток выполняет сложение i-ых компонентов $A[i]$

```
using System;
using GpuDotNet.Cuda;
public static class VectorAddition {
public static void Main ( String[] args )
{
    int     N = Convert.ToInt32 ( args [ 0 ] );
    Console.WriteLine ( "N="  + N );
        int[]      A = new int [ N ];
        int[]      B = new int [ N ];
        int[]      C = new int [ N ];
        for ( int i = 0; i < N; i++ ) {
            A [ i ] = i;
            B [ i ] = i + 1;

        GpuConfig gpuconfig = new GpuConfig();
        gpuconfig.SetBlockSize ( N );
        gpuconfig.vecadd ( A, B, C );
        for ( int i = 0; i < N; i++ )
            Console.WriteLine ( C [ i ] );
    }
    public static gpu vecadd ( int[] A, int[] B, int[] C ) {
        int     i = ThreadIndex.X;
        C [ i ] = A [ i ] + B [ i ];
    }
}
```

и $B[i]$ исходных векторов. Также в данной программе предполагается, что длина векторов N не превосходит размера блока потоков, допускаемого для конкретного графического устройства, на котором предполагается исполнение данной программы.

На примере этой программы, отметим некоторые ключевые особенности MC#-программ, предназначенных для исполнения на GPU, которые будут детализированы в последующих разделах:

(1) Для исполнения на GPU, требуется использование библиотеки GPU.NET, включенной в состав системы программирования MC#. Ссылка на эту библиотеку задается с помощью оператора *using*:

```
        using GpuDotNet.Cuda;
```

(2) Основными методами, устанавливающими параметры конфигу-

рации графического процессора, являются:

- *SetDeviceNumber* (номер графического устройства),
- *SetGridSize* (размеры решетки потоков),
- *SetBlockSize* (размеры блоков потоков).

Некоторые из параметров конфигурации GPU имеют значения по умолчанию (например, номер графического устройства на машине по умолчанию равен 0), а потому вызов некоторых методов задания конфигурации может быть опущен.

(3) *Gpu*-должен быть написан в соответствии с идеологией CUDA. В частности, в нем предполагается использование специфических CUDA-средств, таких как *ThreadIndex, BlockIndex, BlockSize, GridSize, SyncThreads, GetClock* и др.

(4) *Gpu*-метод рассматривается как *extension*-метод (в терминологии языка C#) класса *GpuConfig,* а потому он вызывается относительно некоторого объекта данного класса. В соответствии с ограничениями языка C#, *extension*-методы могут вызываться только из статических классов, а потому *gpu*-методы могут располагаться только в классах, объявленных пользователем с использованием модификатора *static*.

(5) Сам *gpu*-метод также должен быть объявлен как *static*. также как и все функции, вызываемые из него (к которым модификатор *gpu* уже не применяется).

(6) Поскольку графические процессоры текущего поколения имеют собственную память, отличную от памяти основного процессора, то все массивы, являющиеся аргументами *gpu*-метода, копируются неявно из основной памяти в память GPU перед тем, как будет вызван *gpu*-метод, и копируются обратно в основную память после завершения работы этого метода.

(7) В текущей реализации MC#, вызов *gpu*-метода является синхронным (в отличие от *async*- и *movable*-методов), т.е., выполнение основного вычислительного потока, из которого был вызван *gpu*-метод, блокируется до тех пор, пока этот метод не закончит свою работу на GPU.

2. Класс *GpuConfig* и *gpu*-методы

Перед вызовом *gpu*-метода, программист должен определить в MC#-программе конфигурацию (виртуального) графического процессора, на котором этот запуск будет осуществлен. В состав этой конфигурации входят:

(1) номер графического устройства (в случае, если имеется несколько GPU на данной машине; по умолчанию, номер устройства равен 0),
(2) структура поля вычислительных потоков, задаваемая с помощью CUDA-понятий «размер решетки» (grid size) и «размер блока» (block size).

Для задания конфигурации графического процессора, программист создает объект класса *GpuConfig* без параметров:

```
1    GpuConfig gpuconfig = new GpuConfig();
```

Задание параметров этого объекта производится путем вызова статических методов:

(1) SetDeviceNumber (*int* n)
— задание номера графического устройства на машине; значение по умолчанию – 0.

(2) SetBlockSize (*int* X),
SetBlockSize (*int* X, *int* Y),
SetBlockSize (*int* X, *int* Y, *int* Z)
— задание размеров блока вычислительных потоков; значение по умолчанию каждого из параметров есть 1 (напомним, что, в соответствии с идеологией CUDA, блоки потоков не могут быть более, чем 3х-мерными)

(3) *SetGridSize* (*int* X),
SetGridSize (*int* X, *int* Y)
— задание размеров решетки вычислительных потоков; значение по умолчанию каждого из параметров есть 1 (аналогично, в соответствии с идеологией CUDA, решетки потоков не могут быть более, чем 2х-мерными).

Для использования в одной MC#-программе нескольких графических устройств, для каждого из них должен быть создан собственный объект класса GpuConfig. Обычно это делается однотипным образом в каждом из нескольких потоков, запускаемых на основном процессоре, по количеству GPU, используемых в программе. В дистрибутиве системы программирования MC# можно найти пример использования нескольких GPU в одной программе – программу перемножения матриц MatrixMult_ManyDevices на

нескольких графических устройствах одновременно. В рамках единой модели параллельного программирования, принятой в языке MC#, отдельным методам (функциям) в программе может быть приписан модификатор, указывающий место исполнения данного метода при его вызове:

- модификатор *movable* указывает, что данный метод предназначен для исполнения на удаленной машине (узле кластера),
- модификатор *async* указывает, что данный метод предназначен для локального исполнения на отдельном ядре многоядерного процессора,
- модификатор *gpu* указывает, что данный метод предназначен для (локального) исполнения на графическом процессоре.

Однако, *gpu*-методы имеют следующие отличия от *async*- и *movable*-методов, что обусловлено спецификой архитектуры GPU (и некоторыми особенностями текущей реализации системы программирования MC#):

(1) при вызове *movable*- или *async*-метода, запускается только одна новая копия этого метода, а при вызове *gpu*-метода на графическом процессоре запускается столько копий этого метода в параллельных потоках, сколько этих потоков определено в описании конфигурации GPU;
(2) вызовы *movable*- и *async*-методов являются асинхронными, т.е., вычислительный поток, их вызвавший, продолжает свою работу после вызова; вызов же *gpu*-метода является синхронным – исполнение вызвавшего вычислительного потока блокируется до тех пор, пока на графическом процессоре не закончится исполнение всех запущенных копий этого *gpu*-метода;
(3) внутри *movable*- и *async*-методов может происходить обращение к полям (значениям) всех объектов, созданных на данной машине в процессе исполнения программы, а внутри же *gpu*-методов доступны только
    - собственные (локальные) переменные,
    - аргументы, переданные *gpu*-методу, в качестве параметров,
    - константные значения классов,
    - массивы в разделяемой памяти (см. о них в разделе 3);

    т.е., в рамках *gpu*-методов не поддерживается объектно-ориентированная парадигма программирования.

Также, текущая реализация системы программирования MC# поддерживает только ограниченный набор типов данных, которые могут иметь константы и переменные (включая массивы) в *gpu*-методах:

(1) для скалярных значений – это типы *int*, *float* и *double*,
(2) для массивов – это только одномерные массивы с элементами типов *int*, *float* или *double*.

gpu-метод является *extension*-методом класса *GpuConfig*, и потому он может вызываться только относительно объекта этого класса:

```
1    GpuConfig gpuconfig = new GpuConfig();
2    gpuconfig.SetBlockSize ( N );
3    gpuconfig.vecadd ( A, B, C );
4
5         .  .  .
6
7    public static gpu vecadd ( int[] A, int[] B, int[] C ) {
8
9              < Тело метода >
10 }
```

В силу ограничений платформы .NET, класс, из методов которого вызываются *extension*-методы, т.е., в нашем случае – gpuметоды, сам должен быть объявлен как статический (static).

Кроме того, по правилам программирования на языке MC#, сами *gpu*-методы должны объявляться с модификатором static, также как и все методы, вызываемые транзитивно из них. Следует отметить, что модификатором *gpu* отмечаются только те методы, которые являются главными при запуске на GPU, т.е., запускаются относительно объекта класса *GpuConfig*. Все методы (функции), которые, по цепочке, могут вызываться из них, таким модификатором не помечаются, а потому рассматриваются в качестве обычных функций, которые могут возвращать значения. Эти вспомогательные функции (device-функции в терминах CUDA) могут возвращать только скалярные значения типов *int*, *float* или *double*.

Относительно одного объекта класса *GpuConfig*, может как многократно вызываться некоторый *gpu*-метод, так и несколько различных *gpu*-методов (графических ядер (kernels) в терминах CUDA).

Важной особенностью программирования GPU на языке MC# является то, что при вызове *gpu*-метода, массивы, являющиеся его хходными аргументами, неявно копируются из основной памяти в память GPU, а после завершения работы *gpu*-метода – в обратном направлении. Таким образом, логически эти массивы можно рассматривать как находящиеся в общей памяти для CPU и GPU. Тем самым, в этом механизме реализуется понятие «Unified Memory», появившееся в последних версиях системы CUDA.

3. CUDA-средства в *gpu*-методах

В языке MC# реализован ряд средств, которые могут быть использованы в *gpu*-методах, и которые являются аналогами средств, доступных для применения в global- и device-функциях оригинальной технологии CUDA.

Эти средства языка MC# подразделяются на три группы:

(1) средства для получения параметров поля вычислительных потоков,
(2) средства для синхронизации потоков внутри блока,
(3) средства для получения значения счетчика временных тактов.

Средствами для получения параметров поля вычислительных потоков являются:

(1) свойства *ThreadIndex.X*, *ThreadIndex.Y* и *ThreadIndex.Z*, определяющие порядковый номер (индекс) текущего потока относительно каждой их координат, в общем случае, трехмерного блока;
(2) свойства *BlockSize.X*, *BlockSize.Y*, *BlockSize.Z*, определяющие размер блока потоков по каждой из координат;
(3) свойства *BlockIndex.X*, *BlockIndex.Y*, определяющие порядковый номер (индекс) блока, к которому относится текущий поток, в структуре решетки блоков;
(4) свойства *GridSize.X*, *GridSize.Y*, определяющие размер решетки блоков вычислительных потоков по каждой из координат.

Тип возвращаемого значения каждого из этих свойств – *int*. Пример использования некоторых из этих свойств можно найти в разделе 2 данной статьи в программе сложения двух векторов.

Средством синхронизации вычислительных потоков внутри блока потоков является функция *SyncThreads*, относящаяся к встроенному классу *CudaRuntime*:

```
1     CudaRuntime.SyncThreads();
```

В соответствии с семантикой CUDA, исполнение этой инструкции в некоторой точке *gpu*-метода, означает приостановку выполнения текущего вычислительного потока до тех пор, пока до аналогичной точки не дойдут остальные копии данного *gpu*-метода (вычислительные потоки), относящиеся к одному и тому же блоку потоков. Пример использования *SyncThreads* можно найти в дистрибутиве системы MC#, в программе для перемножения матриц с использованием разделяемой памяти (shared memory).

Для получения значения счетчика временных тактов, позволяющего замерять время исполнения фрагментов кода gpuметодов, имеется функция *GetClock* – аналог функции *clock* библиотеки CUDA. Ее вызов имеет вид:

```
1     CudaRuntime.GetClock();
```

Тип возвращаемого значения этой функции – *int*.

Вычислительные потоки, относящиеся к одному и тому же блоку, имеют доступ к так называемой «разделяемой памяти» (shared memory), организованной обычно в виде массивов, помеченных квалификатором «_ _shared_ _» в оригинальной технологии CUDA. «Разделяемая память» является одним из основных средств графических процессоров, позволяющих достигать ими очень высокой производительности по сравнению с обычными процессорами на определенном круге задач.

На языке MC#, массивы, размещаемые в разделяемой памяти, задаются в виде статических массивов параметризованного (generic) типа Shared1D. В текущей реализации MC#, элементы таких массивов могут быть типов *int*, *float* и *double*, а сами массивы могут быть только одномерными.

Пример определения массива, размещаемого в области разделяемой памяти, приведен ниже:

```
1     private const int BLOCK_SIZE = 16;
2     [StaticArray ( BLOCK_SIZE * BLOCK_SIZE ) ]
3     private static Shared1D<double> A;
```

Размер таких массивов может задаваться только константными значениями при помощи специального атрибута StaticArray, который должен предшествовать самому определению массива. Количество и размер массивов, размещаемых в разделяемой памяти, ограничивается физическими размерами такой памяти, имеющейся на каждом конкретном типе графического устройства.

Классическим примером использования разделяемой памяти является алгоритм перемножения матриц, использующий этот вид памяти GPU, код которого можно найти в дистрибутиве системы программирования MC#.

Для использования в MC#-программах, исполняющихся на GPU, математических функций, в систему программирования MC# текущей версии включена библиотека GPUMath, в которой реализовано подмножество математических функций библиотек CUDA точности single и *double*, включая библиотеку внутренних функций (intrinsics).

Общий формат обращения к этим функциям имеет вид:

```
1    GPUMath.имя_функции ( аргумент );
```

На данный момент, в составе библиотеки GPUMath имеются функции:

(1) Single precision (float):
- sqrtf : вычисление квадратного корня,
- sinf : вычисление синуса,
- cosf : вычисление косинуса,
- log2f : логарифм по основанию 2,
- exp2f : возведение в степень по основанию 2,
- fabsf : вычисление абсолютного значения.

(2) Double precision (double):
- sqrt :вычисление квадратного корня,
- sin : вычисление синуса,
- cos : вычисление косинуса,
- fabs : вычисление абсолютного значения.

(3) Single precision intrinsics:
- _ _logf : быстрое приближенное вычисление логарифма по основанию e,
- _ _expf : быстрое приближенное возведение в степень по основанию e.

Пример использования математических функций в *gpu*-методах можно найти в реализации программы из области финансовой математики – вычисления стоимости европейских опционов по методу Блэка-Шоулза, включенной в дистрибутив системы программирования MC#.

Появление в последних версиях библиотеки CUDA средств раздельной компиляции, когда *global*- и *device*-функции могут компилироваться раздельно с дальнейшей сборкой (линкованием) в единый исполняемый на GPU модуль, позволит в последующих версиях системы MC# поддержать полный список математических функций, которые оригинально доступны в системе CUDA (см. об этом более подробно в следующем разделе).

4. Направления развития системы программирования MC# для GPU

Графические процессоры сравнительно недавно заняли место в области высокопроизводительных вычислений. А потому в течение нескольких последних лет наблюдалось быстрое усовершенствование архитектур таких процессоров и наращивание их возможностей как в плане дальнейшего повышения их производительности, так и в повышении удобства их программирования.

Соответствующим образом развивается и базовая технология CUDA, предназначенная для программирования GPU. Только за последние 2 года сменилось несколько версий этой библиотеки, и каждая из них представляла собой серьезный шаг в развитии средств программирования GPU. За это время появились и новые инструменты в этом классе, среди которых следует отметить технологию OpenACC [4], являющуюся аналогом системы OpenMP, но в применении к графическим процессорам.

Отметим следующие важные направления развития системы программирования MC# для GPU, практическая реализация которых позволит сделать систему MC# более удобной для использования и более мощной в плане компактного выражения основных приемов программирования графических процессоров.

(1) *Повышение гибкости управления перемещением данных из основной памяти в память GPU и обратно.*

Часто встречаются задачи, в которых приходится многократно вызы-

вать один и тот же или несколько *gpu*-методов. При этом копирование данных из основной памяти в память GPU и обратно требуется только в начале и конце такой последовательности вызовов. Типичным примером такой задачи является LU-разложение матриц. В систему MC# для GPU предполагается включить новые средства управления копированием данных, которые позволяют более компактно записывать такого рода фрагменты программ и эффективно их исполнять на GPU. В частности, предполагается исследовать возможности взаимодействия этих новых средств со средством Unified Memory последних версий библиотеки CUDA.

(2) *Реализация динамического параллелизма.*

До появления последних моделей графических процессоров компании Nvidia и версии 5.0 библиотеки CUDA, на GPU было сильно затруднено программирование рекурсивных алгоритмов из-за невозможности вызова из *global*-функций (в терминах CUDA) другой или той же самой *global*-функции. Последние модели GPU и выпущенные недавно версии библиотеки CUDA поддерживают теперь такую возможность, которая получила название «dynamic parallelism». В языке MC# предполагается реализовать аналогичные возможности для динамического параллелизма, в частности, возможность рекурсивного программирования. Это еще более расширит круг задач, которые можно эффективно решать на графических процессорах на языке MC#.

(3) *Подключение базовых библиотек CUDA.*

Текущая версия системы программирования MC# для GPU поддерживает только очень ограниченное подмножество (математических) функций, имеющихся в нескольких базовых библиотеках CUDA. Это сильно затрудняет программирование на MC# сложных алгоритмов, где необходимы, например, функция быстрого преобразования Фурье и другие аналогичные функции, которые программист должен в данном случае запрограммировать сам. Появление в последних версиях библиотеки CUDA возможностей раздельной компиляции *global*- и *device*-функций позволит подключить к системе MC# любую стандартную CUDA-библиотеку, такую как *cuBLAS*, *cuFFT* и другие.

(4) *Программирование в терминах глобального поля вычислительных потоков.*

В силу архитектурных особенностей графических процессоров, программист должен разрабатывать свою программу в терминах 3х-уровневой структуры вычислительных процессов: поток – блок потоков – решетка потоков. Выбор параметров такой структуры является одним из ключевых решений при разработке приложения для GPU и, в конечном итоге, во многом определяет эффективность исполнения приложения на графическом процессоре. В некоторых случаях, имеется возможность предоставить программисту средства для программирования GPU в терминах одноуровневой структуры – (однородного) поля вычислительных потоков. Предполагается в рамках системы MC# для GPU исследовать возможность автоматического преобразования программы, написанной в терминах глобального поля вычислительных потоков, в программу, реализованную в терминах 3х-уровневой системы, пригодной для непосредственного исполнения на графическом процессоре.

Заключение

Современные суперкомпьютеры теперь уже не представимы без универсальных ускорителей типа GPU или Xeon Phi. Специфическая архитектура графических процессоров, в частности, быстрая локальная память, позволяет исполнять на них некоторые приложения в десятки раз быстрее по сравнению с обычными процессорами. Но эта же специфичность архитектуры GPU в несколько раз усложнила программирование для них. Поэтому, в настоящий момент, очень актуальны исследования, направленные как на совершенствование уже имеющихся средств для программирования GPU, так и на поиск новых парадигм их использования. В рамках проекта MC# для GPU (www.mcsharp.net), решаются оба вида таких задач с целью сделать язык MC# практическим средством решения реальных задач на GPU в таких областях, как обработка изображений, финансовая математика, машинное обучение и др.

Список литературы

[1] Петров А. В., Сердюк Ю. П., «Система параллельного программирования MC# 2.0», *Вычислительные методы и программирование*, **9** (2008), с. 1–11, URL http://num-meth.srcc.msu.ru/zhurnal/tom_2008/v9r201.html ↑ 3.

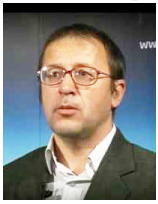
[2] Сердюк Ю. П., «Программирование графических процессоров на языке MC#», *Тр. Международной суперкомпьютер. конф. «Научный сервис в сети Интернет: суперкомпьютерные центры и задачи»* (Россия, Абрау-Дюрсо, сент. 2010), URL http://agora.guru.ru/abrau2010/pdf/293.pdf ↑ 3.

[3] D. Negrut, R. Serban, A. Li, A. Seidell, "Unified Memory in CUDA 6: A Brief Overview", *Dr. Dobb's Journal*, 2014, Sept., URL http://www.drdobbs.com/parallel/unified-memory-in-cuda-6a-brief-overvie/240169095 ↑ 5.

[4] R. Farber, "Easy GPU Parallellism with OpenACC", *Dr. Dobb's Journal*, 2012, June, URL http://www.drdobbs.com/parallel/easy-*gpu*-parallelism-withopenacc/240001776 ↑ 14.

Рекомендовал к публикации *к.т.н. Е. П. Куршев*

Об авторе:



Юрий Петрович Сердюк

Старший научный сотрудник ИПС РАН. Занимается вопросами параллельного программирования для многоядерных и кластерных систем, включая системы с ускорителями различного типа.

e-mail: yury@serdyuk.botik.ru

Образец ссылки на эту публикацию:

Ю. П. Сердюк. *Программирование графических процессоров (GPU) на языке MC#* // Программные системы: теория и приложения: электрон. научн. журн. 2014. Т. 5, № 4(22), с. 3–17.

URL http://psta.psiras.ru/read/psta2014_4_3-17.pdf

Yury Serdyuk. *Programming the graphics processors (GPU) in MC# language.*

Abstract. MC# is an extension of the object-oriented language C#. It intended for developing applications running on multicore processors and on clusters with distributed memory. Given paper presents the basic principles and tools for programming the graphics processors (GPUs) in MC# language. The sample program in MC# intended to run on GPU is presented. Further we describe the tools for establishing the parameters of GPU configuration and for using of shared memory in MC# programs. In the final section we outline the current problems under investigation and the directions for further improvement of MC# programming system for GPU. (*In Russian*).

Key Words and Phrases: Parallel programming, graphics processors (GPU), shared memory.