ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ НА ЯЗЫКЕ MC#

Ю. Сердюк

Резюме:

Представлено расширение языка MC# конструкциями для для программирования графических процессоров. Данное расширение основано на библиотеке GPU.NET, реализованной на языке C# и включающей в себя JIT-компилятор для GPU и набор функций, соответствующих функциям базовой библиотеки CUDA.

Приведены пример программы на MC#, предназначенный для исполнения на графическом процессоре, а также тестовые результаты исполнения MC#-программ на системах с GPU, включая кластерные системы.

1. ВВЕДЕНИЕ

Усложнение архитектуры вычислительных устройств для высокопроизводительных вычислений требует в настоящий момент применения целой группы технологий/библиотек для их программирования. Так, для программирования кластерных установок, имеющих узлы с графическими процессорами (компании NVIDIA), требуется применение связки технологий MPI + Pthreads + CUDA. Если, одновременно, приложения используют универсальные многоядерные процессоры, то к вышеприведенному списку могут добавляться библиотеки типа OpenMP, Intel TBB и т.п. Частично проблему унификации разноообразных программных средств, применяемых для программирования гетерогенных параллельных архитектур, решает библиотека OpenCL, но она достаточно сложна в использовании (описание стандарта OpenCL 1.1 содержит более 300 стр.) и имеет свои ограничения.

В данной работе представлено расширение асинхронной модели программирования, реализованной в языке MC#, конструкциями и средствами, которые позволяют программировать графические процессоры не выходя за рамки указанной модели. Это расширение значительно прощает программирование гетерогенных параллельных архитектур, одновременно обеспечивая достаточную эффективность исполнения MC#-программ на графических процессорах.

2. РАСШИРЕНИЕ ЯЗЫКА MC# ДЛЯ ПРОГРАММИРОВАНИЯ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ

Язык программирования MC# [1] является расширением языка C# и предназначен для разработки параллельных, распределенных приложений. Параллельные приложения предназначены для исполнения на многоядерных процессорах, а распределенными приложениями называются программы, предназначенные для исполнения на вычислительных системах с распределенной памятью (кластерах).

Основным механизмом параллельности в языке MC# являются

а)*асинхронные*(**async**) методы, вызов которых приводит к порождению нового локального потока в рамках которого выполняется тело данного метода, и назначаемого обычно на отдельное ядро, и

б)*перемещаемые*(**movable**) методы, вызов которых аналогичен вызову async-методов, за исключением того, что новый поток может быть порожден на удаленной машине — обычно, на наименее загруженном узле кластера.

Средствами взаимодействия между параллельными процессами (**async**- и **movable**- методами) в языке MC# являются *каналы*(**channels**) и *обработчики* (**handlers**): по каналам может быть послан произвольный набор (скалярных)значений/объектов, которые могут быть получены (возможно, с некоторой предобработкой) посредством обработчика.

Базовая идея адаптации системы программирования MC# к вычислительным архитектурам на базе графических процессоров состоит во введении дополнительного модификатора gpu, который должен быть использован в определении метода, предназначенного для исполнения на GPU. В частности, этим модификатором должны быть помечены все методы, которые вызываются из основного gpu-метода, который, в свою очередь, вызывается из хост-программы (т.е., программы, исполняющейся на основном процессоре).

Общая идеология программирования графических процессоров на языке MC# совпадает с идеологией технологии CUDA. Параметры конфигурации графического процессора устанавливаются программистом с использованием директивы *#gpuconfig* препроцессора языка MC#. Пример задания такой директивы приведен ниже:

```
#gpuconfig ( Device = myNumber, GridSize = ( 1, N ),
             BlockSize = ( BLOCK_SIZE, BLOCK_SIZE ) )
```

Эта директива обрабатывается компилятором языка MC#, который генерирует соответствующие вызовы функций библиотеки GPU.NET, устанавливающие параметры графического процессора.

Чтобы освободить программиста от необходимости явно программировать передачу данных из основной памяти в память графического устройства и обратно, при вызове gpu-метода к аргументам вызова,

являющимся массивами, могут применяться модификаторы *in* и *out*: модификатор *in* (применяемый по умолчанию) для некоторого массива означает, что этот массив должен быть скопирован в память графического устройства до непосредственного вызова gpu-метода, а модификатор *out* для массива означает, что он должен быть скопирован из памяти графического устройства в основную память по окончании вызова gpu-функции. Тем самым, в языке MC# модификатор *out*, применяемый к аргументам-массивам gpu-функции, имеет другую семантику по сравнению с его аналогом в языке C#.

Сам вызов gpu-методов в MC#-программах является синхронным, т.е., после вызова gpu-функции исполнение вызывающей программы блокируется до тех пор, пока не завершится исполнение вызванной функции на графическом устройстве. Соответственно, при использовании в MC#-программе нескольких графических устройств соответствующие gpu-методы должны вызываться из параллельных потоков, которые порождаются в MC#-программах путем вызова async-методов. Наконец, исполнение распределенных MC#-программ на кластере с узлами, оснащенными несколькими графическими устройствами, требует использования соответствующей иерархии методов: movable, async- и gpu-методов.

Ниже представлен полный текст программы на языке MC# для сложения двух векторов целых чисел с использованием GPU:

```
using System;
public class VectorAddition {

 public static void Main ( String[] args ) {

 int N = Convert.ToInt32 ( args [ 0 ] );       //  Length of vectors
 Console.WriteLine ( "N = " + N );

 int[] A = new int [ N ];
 int[] B = new int [ N ];
 int[] C = new int [ N ];

 Random rand = new Random();

 for ( int i = 0; i  N; i++ ) {
  A [ i ] = rand.Next();
  B [ i ] = rand.Next();
 }

 #gpuconfig ( BlockSize = ( N ) )
 vecadd ( A, B, out C );

 Console.WriteLine ( "A [ 0 ] = " + A [ 0 ] + " " +
                "B [ 0 ] = " + B [ 0 ] + " " +
                "C [ 0 ] = " + C [ 0 ]        );
}

 public static gpu vecadd ( int[] A, int[] B, int[] C ) {

 int i = ThreadIndex.X;
 C [ i ] = A [ i ] + B [ i ];

 }

}
```

С помощью директивы *#gpuconfig* программист обязан указывать параметры графического устройства специфичные только для его программы. Всем остальным таким параметрам будут присвоены стандартные значения по умолчанию (в частности, значение по умолчанию параметра Device равно 0).

3. ТЕСТОВЫЕ РЕЗУЛЬТАТЫ

Тестирование проводилось на 2-ухузловом кластере под управлением ОС Linux, где каждый узел кластера был оснащен двумя графическими картами Tesla C1060. Ниже представлен график для программы перемножения двух матриц размером 18432 x 18432 с элементами типа float. График показывает зависимость времени исполнения программы (в секундах) от количества используемых графических устройств.

В качестве межузловой сети на данном кластере применялась сеть Infiniband. Для использования данного вида сети быстрой передачи данных в MC#-программах, был сконфигурирован драйвер IPoIB (Internet Protocol over Infiniband) и был применен протокол SDP (Socket Direct protocol). Таким образом, сам код MC#-программ никак не изменяется и не требует перекомпиляции при использовании различных типов сетей передачи данных.

Рис. 1

Сам алгоритм перемножения матриц является портированием на C# соответствующего алгоритма из примеров, включенных в NVIDIA CUDA SDK, и использует разделяемую паиять (shared memory). В GPU.NET, массивы, размещаемые в разделяемой памяти, объявляются в виде (параметризованных) статических массивов. Пример такого объявления приведен ниже:

```
[StaticArray ( SIZE ) ]
private static Shared1Dfloat A_submatrix;
```

Для (барьерной) синхронизации потоков в рамках одного блока используется функция *SyncThreads* класса *CudaRuntime*, а для получения индексов потоков внутри блока и индексов блока внутри решетки блоков используются статические переменные *ThreadIndex* и *BlockIndex*.

В качестве второго тестового примера использовалась программа N-Queens, вычисляющая количество всех возможных различных расстановок ферзей на шахматной доске размером N x N, где ферзи попарно не атакуют друг друга. Применялся алгоритм, учитывающий симметрии при вычислении количества расстановок, и адаптированный к GPU посредством перевода исходного рекурсивного алгоритма в итеративный. Массивы, в которых сохранялась текущая конфигурация ферзей, размещались в разделяемой памяти, а список "заданий" (частичных конфигураций), который должен быть обработан одним GPU, располагался в памяти графического устройства.

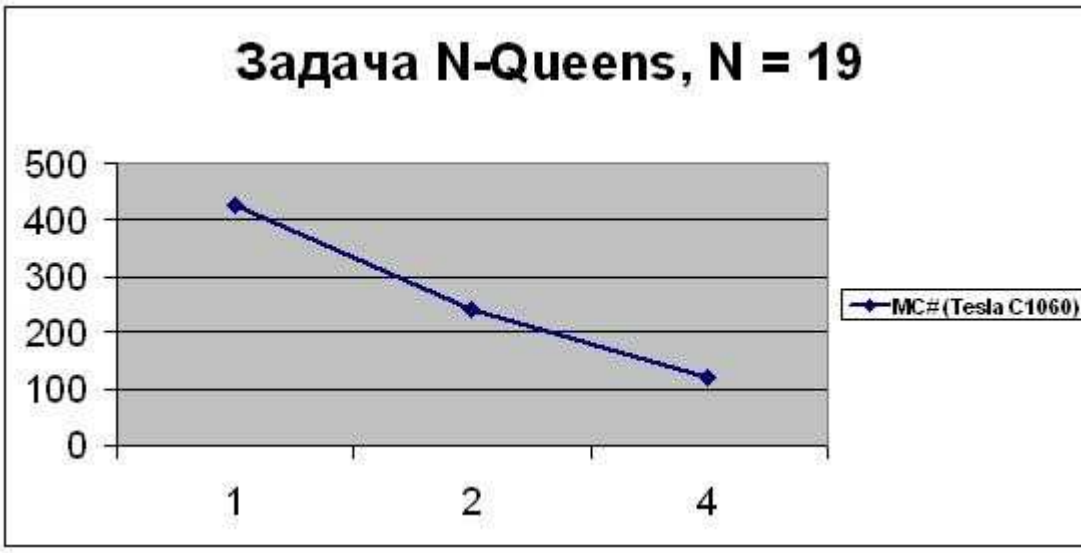

Рис. 2

Ниже на графике показано время решения задачи (в сек.) для N = 19 в зависимости от количества используемых графических устройств (в соответствии с архитектурой Tesla C1060, на каждом таком устройстве задача разделялась между 240 графическими ядрами):

4. ЗАКЛЮЧЕНИЕ

В данной работе было представлено расширение языка MC# конструкциями для программирования графических процессоров. Попытки разработки средств программирования графических процессоров с использованием языка C# предпринимались в разных проектах. Так, в проекте CUDA.NET [2] хост-программу можно писать на языке C#, однако, для написания программы, исполняющейся на GPU, по-прежнему, необходимо использовать язык C. Суть проекта Accelerator [3] компании Microsoft состоит во введении специальных объектов -"параллельных массивов" и операций над ними, которые можно использовать в программах на языке C#. Данные операции транслируются в код, исполняемый на графическом устройстве. При данном подходе, программист не имеет возможности написать собственную функцию, предназначенную для исполнения на GPU, а потому круг задач, решаемых с помощью данного подхода, является достаточно узким.

Тестовые результаты исполнения MC#-программ на системах с GPU показали, что на специального рода задачах (например, включающих операции с матрицами) производительность MC#-программ в несколько раз выше, чем производительность программ на языке C, оттранслированных с помощью оптимизирующих компиляторов и исполняемых на универсальных многоядерных процессорах. Ниже приведен сравнительный график времени исполнения программ перемножения матриц (в сек.), реализованных на MC# и на языке C с применением OpenMP, в зависимости от размера матриц. Программа на C + OpenMP исполнялась на машине с 8 физическими ядрами (два 4-ехядерных процессора Intel Xeon X5570 2.93 GHz):

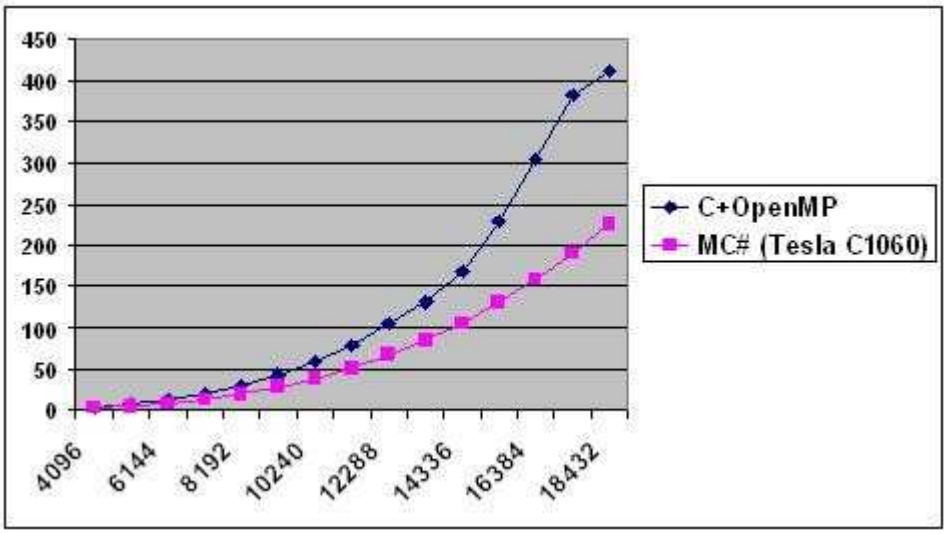

Рис. 3

Следует отметить, что на текущий момент в JIT-компиляторе для GPU не реализован этап оптимизации порождаемого кода для графического процессора, что является одним из направлений дальнейшего развития библиотеки GPU.NET, включенной в состав системы программирования MC#.

Текущую версию системы MC# можно найти на сайте www.mcsharp.net. Автор выражает благодарность Александру Петрову и Илье Виноградову за помощь в проведении экспериментов.

ЛИТЕРАТУРА.

1. А.В. Петров, Ю.П. Сердюк "Система параллельного распределенного программирования MC# 2.0", - Вычислительные методы и программирование, т.9, N 1, стр. 158 - 168.
2. http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx
3. D. Tarditi, S. Puri, and J. Oglesby "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses", - Microsoft Research Technical Report, MSR-TR-2005-184, October 2006.