Программирование графических процессоров (GPU) на языке MC#

1. Введение

Язык программирования MC# 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). Программирование приложений, при этом, производится исключительно средствами языка C# и специфическими средствами языка MC# и не требует применения каких-либо иных средств, таких как MPI, OpenMP и т.п.

Расширение языка MC# для поддержки программирования графических процессоров (GPU) лежит в рамках единой асинхронной модели программирования, принятой в языке MC#. 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, добавляются так называемые ***gpu***-методы – методы, которые предназначены для исполнения на графическом процессоре (о ***gpu***-методах см. Раздел 4 данного документа).

Общая идеология программирования графических процессоров на языке MC# совпадает с идеологией технологии CUDA, знание которой предполагается для программирования GPU на MC#. В частности, перед вызовом ***gpu***-метода программист должен установить параметры конфигурации графического процессора путем создания и задания значений специального объекта класса GpuConfig. Класс GpuConfig и его методы описаны в Разделе 3 данного документа, а специфические средства CUDA, используемые в рамках ***gpu***-методов, представлены в Разделе 5.

Раздел 6 данного документа посвящен использованию разделяемой памяти (shared memory) в MC#-программах, предназначенных для исполнения на GPU, а в Разделе 7 описано текущее состояние библиотеки GPUMath – библиотеки математических функций, которые можно использовать в ***gpu***-методах.

Составной частью системы программирования MC#, поддерживающей графические процессоры, является библиотека GPU.NET, реализованная на языке C# и включающая в себя

1) JIT-компилятор для GPU и
2) набор функций, соответствующих базовым функциям библиотеки CUDA.

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

GPU и обратно – эта задача решается компилятором языка MC#, который генерирует вызовы соответствующих функций библиотеки GPU.NET, реализующих копирование данных.

Поскольку все составные компоненты 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 система Mono (www.mono-project.com).

Для ОС Windows имеется интеграция языка MC# в систему разработки Microsoft Visual Studio 2008/2010, что позволяет в рамках нее разрабатывать и исполнять MC#-программы для GPU. При этом, как в случае ОС Windows, так ОС Linux, предполагается наличие на машине установленной системы CUDA (http://developer.nvidia.com/cuda-downloads).

Графическими процессорами, поддерживаемыми в MC#, являются все типы GPU компании Nvidia.

2. Пример программирования GPU на языке MC#.

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

1) ***gpu***-функции (метода), которая будет исполняться в рамках одного потока на GPU, и
2) описания конфигурации графического процессора, задающего, в частности, структуру и количество потоков, запускаемых на GPU.

Gpu-метод в программе определяется путем задания для него модификатора ***gpu*** на месте типа возвращаемого значения. Определение конфигурации GPU производится с помощью создания объекта класса GpuConfig и задания его параметров.

Ниже представлен полный текст программы на языке MC# 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 блока потоков, запускаемых на GPU – соответственно, i-ый поток выполняет сложение i-ых компонентов A[i] и B[i] исходных векторов.

```
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 ];

 }
}
```

Отметим некоторые особенности этой программы, которые будут уточнены в следующих разделах:

1. Для использования средств библиотеки GPU.NET, включенной в состав системы программирования MC#, в программе должен быть использован оператор

   using GpuDotNet.CUDA.

2. Основными методами, устанавливающими параметры конфигурации графического процессора, являются
   - SetDeviceNumber,
   - SetGridSize,
   - SetBlockSize.

3. ***Gpu***-метод должен быть написан в соответствии с идеологией CUDA. В частности, в нем могут применяться специфические CUDA-функции, такие как ThreadIndex, BlockIndex, BlockSize, GridSize, SyncThreads, GetClock и др.

4. ***Gpu***-метод вызывается как extension-метод класса GpuConfig, т.е., он вызывается относительно созданного объекта данного класса. В соответствии с ограничениями платформы .NET, extension-методы могут вызываться только из статических классов, а потому ***gpu***-методы могут объявляться только в классах, объявленных с модификатором ***static***.

5. Сам ***gpu***-метод также должен быть объявлен как ***static***, также как и все функции, вызываемые из него.

6. Поскольку графический процессор имеет собственную память, то все массивы, являющиеся аргументами ***gpu-***метода, копируются неявно из основной памяти в память GPU перед исполнением ***gpu***-метода, и копируются обратно в основную память после завершения работы этого метода.

7. Вызов ***gpu***-метода является *синхронным*, т.е., выполнение вычислительного потока откуда он был вызван, блокируется до тех пор, пока этот метод не закончит свою работу на GPU.

3. Класс GpuConfig и его методы

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

1) номер графического устройства (если имеется несколько GPU на данной машине; номер устройства по умолчанию – 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 необходимо вначале создать объект класса GpuConfig без параметров:

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

Задание его параметров производится с помощью 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 – 1.

3) SetGridSize (int X),
SetGridSize (int X, int Y)

− задание размеров решетки вычислительных потоков; значение по умолчанию каждого из параметров – 1.

При использовании в одной программе нескольких графических устройств, для каждого из них должен быть создан собственный объект класса GpuConfig. Обычно это делается однотипным образом в нескольких потоках, запускаемых на основном процессоре, по количеству имеющихся GPU. Пример использования нескольких GPU можно найти в дистрибутиве системы MC# (программа MatrixMult_ManyDevices).

4. Gpu-методы

В рамках единой асинхронной модели программирования, принятой в языке MC#, отдельным методам (функциям) программы может быть приписан модификатор, указывающий место исполнения данного метода при его вызове:

- модификатор ***movable*** указывает, что данный метод может быть исполнен на удаленной машине (узле кластера),
- модификатор ***async*** указывает, что данный метод может быть исполнен локально на другом ядре многоядерного процессора,
- модификатор ***gpu*** указывает, что данный метод может быть исполнен на графическом процессоре.

Однако, ***gpu***-методы имеют три отличия от ***async***- и ***movable***-методов:

1) тогда как при вызове ***movable***- или ***async***-метода запускается только одна копия этого метода, то при вызове ***gpu***-метода на графическом процессоре запускается столько копий этого метода в параллельных потоках, сколько их определено в описании конфигурации GPU;
2) тогда как вызовы ***movable***- и ***async***-методов являются *асинхронными*, т.е., вычислительный поток, их вызвавший, продолжает после вызова свою работу, то вызов ***gpu***-метода является *синхронным* – вызвавший вычислительный поток блокируется до тех пор, пока на графическом процессоре не закончится исполнение всех запущенных копий ***gpu***-метода;
3) тогда как внутри ***movable***- и ***async***-методов может происходить обращение к общим полям (значениям) объекта, к которому они относятся, а также к полям других объектов, то внутри ***gpu***-методов доступны только
   - собственные (локальные) значения переменных,
   - аргументы, переданные ***gpu***-методу, в качестве параметров,
   - константные значения классов,
   - массивы в разделяемой памяти (см. о них в Разделе 6).

На данный момент, в MC# поддерживается ограниченный набор типов данных, которые могут иметь входные параметры ***gpu***-методов:

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

Gpu-метод является *extension*-методом класса GpuConfig; т.е., он может вызываться только относительно объекта этого класса:

```
GpuConfig gpuconfig = new GpuConfig();
gpuconfig.SetBlockSize ( N );
gpuconfig.vecadd ( A, B, C );

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

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

Кроме того, сами ***gpu***-методы должны быть объявлены с модификатором ***static***, также как и все методы, вызываемые по цепочке, из них.

Модификатор ***gpu*** должен быть приписан только главному методу (global-функции в терминах CUDA), запускаемому на графическом процессоре – всем вспомогательным функциям, вызываемым из него, такой модификатор не нужен. Эти вспомогательные функции (*device*-функции – в терминах CUDA) могут возвращать только скалярные значения типов ***int***, ***float*** и ***double***.

В рамках одного объекта класса GpuConfig, может вызываться несколько различных ***gpu***-методов (графических ядер (kernels) – в терминах CUDA).

При вызове ***gpu***-метода, массивы, являющиеся его входными 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 общую память для CPU и GPU.

5. 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# разбиты на 3 группы:

1) средства для получения значений, связанных с иерархией вычислительных потоков;
2) средства для синхронизации потоков внутри блока;

3) средства для получения значения счетчика временных тактов.

Средствами для получения значений, связанных с иерархией вычислительных потоков, являются:

1) свойства ThreadIndex.X, ThreadIndex.Y и ThreadIndex.Z, определяющие индекс текущего потока по каждой из 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 индекс блока, к которому относится текущий поток, в рамках 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***.

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

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

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

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

Тип возвращаемого значения – ***int***.

6. Использование разделяемой памяти

Вычислительные потоки, относящиеся к одному и тому же блоку, имеют доступ к так называемой “разделяемой памяти” – массивам, помеченным квалификатором “__shared__” в оригинальной технологии CUDA.

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

Размер этих массивов может задаваться только константными значениями при помощи специального атрибута StaticArray, который должен предшествовать самому определению массива.

Пример определения массива в области разделяемой памяти приведен ниже:

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

Пример определения и использования массивов в разделяемой памяти можно найти в дистрибутиве системы MC#, программа MatrixMult.

7. Математические функции в gpu-методах

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

Общий формат обращения к этим функциям имеет вид:

```
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 : вычисление логарифма 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 натуральному основанию *e*,
   - __expf : быстрое приближенное возведение в степень по натуральному основанию *e*.