Programming the Graphics Processors (GPU) in MC# language

1. Introduction

MC# programming language is an extension of the object-oriented language C# and is intended for developing applications running on multicore processors and on computational clusters with distributed memory. In this case, applications are developed by tools of C# language and specific constructs of MC# language only without any other formalisms as MPI, OpenMP etc.

An extension of MC# language for supporting of graphics processors lies within a single asynchronous parallel programming model adopted in MC# language. In particular, to the ***async***-methods which are intended to run on separate cores of multicore processors and to the ***movable***-methods which are intended to run on separate nodes of clusters, are added so called ***gpu***-methods which are intended to run on graphics processors (about ***gpu***-methods see Section 4 of given document).

General ideology of graphics processors programming in MC# language coincides with the ideology of CUDA technology and knowledge of it are assumed for successful GPU programming in MC#. In particular, before a ***gpu***-method invoking a programmer must establish the parameters of graphics processor configuration by creating and setting values of special object of *GpuConfig* class. The *GpuConfig* class and its methods are described in Section 3 of given document, and specific CUDA tools which can be used in ***gpu***-methods, are presented in Section 5.

Section 6 of given document is devoted to using of shared memory in MC# programs which are intended to run on GPU, and Section 7 describes a current state of GPUMath library – a library of mathematical functions which can be used in ***gpu***-methods.

A constituent part of MC# programming system which supports the graphics processors is a GPU.NET – a library implemented in C# and which includes

1) JIT-compiler for GPU and
2) collection of functions which correspond to the basic functions of CUDA library.

The using of MC# language to program the graphics processors is much more simple than using of the basic CUDA technology. In particular, a programmer doesn't need to program explicitly the data transfer from host memory to GPU memory and back in MC# language. In fact, this task is solved by the MC# compiler by generation of the calls of corresponding functions of GPU.NET library which implements the data copying.

All components of MC# programming system, including components that support GPU, are implemented in C# language and so can be running both under Windows and Linux operating systems. In latter case, free accessible Mono system (www.mono-project.com) is used as an implementation of .NET platform.

MC# language can be integrated in Microsoft Visual Studio 2008/2010. This allows to develop and to run MC# programs for GPU within Studio. At this, both for Windows and Linux, the presence of the installed CUDA system at the machine is supposed.

MC# programming system supports all types of Nvidia GPU processors.

2. An example of GPU programming in MC# language

A basic structure of MC# program which is supposed to be run on graphics processor, consist of

1) ***gpu***-function (method) for running within one thread on GPU,
2) description of graphics processor configuration giving, in particular, the structure and number of threads which will be run on GPU.

Gpu-method in a program is declared by placing ***gpu*** modifier instead of the return type. Defining a GPU configuration is performed by creating of object of *GpuConfig* class and setting parameters for it.

Below a full program in MC# language for integer vector addition using GPU is given. In that program, vectors A and B and the result vector C have length N. The given number serves as size of block of threads running on GPU, so i^{th} thread performs addition of ^{ith} components A[i] and B[i] of input vectors correspondingly.

```
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 ];

 }
}
```

Let's note some features of the above program which will be made more exact in the following sections:

1. To use functions from the GPU.NET library, which is included to the MC# programming system, a statement

   using GpuDotNet.CUDA

   must be included to the program.

2. The basic methods for setting the configuration parameters of graphics processors are thr following
   - SetDeviceNumber,
   - SetGridSize,
   - SetBlockSize.

3. The ***gpu***-methods must be developed in correspondence with CUDA ideology. In particular, they can use CUDA-specific functions as ThreadIndex, BlockIndex, BlockSize, GridSize, SyncThreads, GetClock et al.

4. ***Gpu***-method is invoked as an *extension*-method of *GpuConfig* class; i.e., it is invoked with respect to the object of given class. In compliance with the restrictions of .NET platform, the *extension*-methods can be invoked only from the static class and so ***gpu***-methods can be declared only in the classes declared with the ***static*** modifier.

5. ***Gpu***-method itself must be declared as ***static*** also and all the functions has been invoked from it as well.

6. A graphics processor has own memory and so all the arguments of ***gpu***-methods which are arrays are copied implicitly from the main memory to the GPU memory before ***gpu***-method starting and back after it terminating.

7. A call of ***gpu***-method is synchronous: the computational thread from which this ***gpu***-method has been called blocks until the ***gpu***-method is terminated.

3. *GpuConfig* class and its methods

To run a ***gpu***-method in a program, a programmer must declare a configuration of the (virtual) graphics processor which will be used to launch the method. This configuration consist of

1) graphical device number (if there are several GPUs on the machine; default device number is 0),
2) structure of array of computational threads which are defined using CUDA-notions of "grid size" and "block size".

To set a graphics processor configuration, at first it is necessary to create an object of *GpuConfig* class without parameters:

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

To set the parameters of this object, there are several static methods:

1) SetDeviceNumber (int n)

   − the setting of graphical device number on the machine; a default value is 0.

2) SetBlockSize (int X),
   SetBlockSize (int X, int Y),
   SetBlockSize (int X, int Y, int Z)

   − the setting of sizes of computational threads block; a default value of each of the parameters is 1.

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

   − the setting of sizes of computational threads grid; a default value of each of the parameters is 1.

If several graphical devices are used in a program, it is necessary to create one's own object of *GpuConfig* class for each of them. Typically, it is performing by the way of the same kind in the several threads which has been running on the host processor. A number of the threads equals to the number of GPUs on the machine. An example of using of several GPUs in the program can be found in the distribution of the MC# system (see *MatrixMult_ManyDevices* program).

4. Gpu-methods

Under the common asynchronous programming model accepted in MC# language, the selected methods (functions) can be marked by some modifier which indicates an execution place of given method when it is invoked:

- ***movable*** modifier indicates that the given method can be executed on a remote machine (node of cluster),
- ***async*** modifier indicates that the given method can be executed locally on a some core of multicore processor,
- ***gpu*** modifier indicates that the given method can be executed on a graphics processor.

But, in fact, ***gpu***-methods have three differences from the ***async***- and ***movable***-methods:

1) while when you invoke a ***movable***- or ***async***- method it is launched only one copy of the given method, then when you invoke a ***gpu***-method it is launched as many copies of the given method in concurrent threads as have been defined in the description of GPU configuration;
2) while the calls of ***movable***- and ***async***-methods are *asynchronous*, i.e., a computational thread from which these methods have been called continues his work after calling, then a call of the ***gpu***-method is *synchronous* – a caller blocks until the all copies of the ***gpu***-method are completed;
3) while within ***movable***- or ***async***-methods addressing to the public fields (values) of the corresponding object and to the (public) fields of another objects is possible, then within the ***gpu***-methods
   − own (local) variables,
   − arguments passed to the ***gpu***-method as parameters,
   − constant values of the classes,
   − arrays in the shared memory (see about it in Section 6)

   are accessible only.

At now, MC# language supports a restricted set of data types that the input parameters of ***gpu***-methods may have:

1) for scalar values this is ***int***, ***float*** and ***doubles*** types,

2) for arrays this is only the one-dimensional arrays of the elements with ***int***, ***float*** and ***doubles*** types.

A ***gpu***-method is an *extension*-method of *GpuConfig* class, so it may be invoked relatively to an object of that class only:

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

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

Due to restrictions of .NET platform, the class from which the *extension*-methods may be called (in our case that is ***gpu***-methods), must be declared as ***static***.

Besides that, each ***gpu***-method itself must be declared with the ***static*** modifier as well as the all methods that are invoked from it.

A ***gpu*** modifier should be ascribed only to the principal method (*global*-function in CUDA terminology) which is run on the graphics processor. All auxiliary functions that are invoked from the principal method doesn't need a ***gpu*** modifier. These auxiliary functions (*device*-functions in CUDA terms) may return ***int***, ***float*** and ***double*** scalar values only.

It is possible to invoke a several different ***gpu***-methods (graphical kernels in CUDA terms) relatively to one object of *GpuConfig* class.

When a ***gpu***-method is invoked, all arrays which are the input parameters of it, are copied implicitly from the main memory to the GPU memory. When the ***gpu-***method is terminated, they are copied back. So these arrays can be considered conceptually as a shared memory for CPU and GPU.

5. CUDA-features in gpu-methods

The MC# language supports some features that can be used in ***gpu***-methods and that are similar to features used in the *global*- and *device*-functions of the original CUDA-technology.

These features of MC# language are divided on 3 groups:

1) features to get the values concerning to the hierarchy of computational threads,
2) features for thread synchronization within the blocks of threads,
3) features to get the values of clock counter.

The first group of the above features includes:

1) ThreadIndex.X, ThreadIndex.Y and ThreadIndex.Z properties which define an index of current thread along the each of the dimensions of three-dimensional (in general) block;
2) BlockSize.X, BlockSize.Y and BlockSize.Z properties which define a size of the block of computational threads along the the each dimension;
3) BlockIndex.X, BlockIndex.Y properties which define an index of block to which the current thread belongs within the grid of blocks;
4) GridSize.X, GridSize.Y properties which define a size of grid of block of computational threads along the each of the dimensions.

For each of these properties, the type of the returned value is ***int***.

The function *SyncThreads* of *CudaRuntime* class is a tool to synchronize the computational threads within a block of threads:

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

To get the values of the clock counter which may be used to measure the execution time of fragments of code within the ***gpu***-methods, there is a *GetClock* function which is similar to the *clock* function from the original CUDA library:

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

The returned value of *GetClock* function has the ***int*** type.

6. Using the shared memory

The computational threads within the same block have an access to a so called "shared memory" which is assigned to that block. The arrays stored in the shared memory are labeled by "__shared__" qualifier in the original CUDA technology.

In MC# language, these arrays are declared as the static generic arrays of *Shared1D* type. In the current implementation, such arrays can be one-dimensional only and their elements may have ***int***, ***float*** or ***double*** type.

The size of these arrays can be given by constants only using *StaticArray* attribute. This attribute must precedes the array declaration.

An example of declaration of the array allocated in the shared memory follows below:

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

The additional examples of declaration and using of arrays in the shared memory can be found in MC# distribution (see, for example, *MatrixMult* program).

7. Mathematical functions in gpu-methods

The MC# programming system has as a component part the *GpuMath* library of mathematical functions intended for running on a GPU. This library implements some subset of *single* and *double* precision functions of original CUDA mathematical functions libraries including the intrinsics library.

A general form of invoking of these functions is the following:

```
GPUMath.function_name ( arguments );
```

At current time, *GPUMath* library includes the following functions:

1) Single precision (float)
   - sqrtf: calculate the nonnegative square root of the input argument,
   - sinf: calculate the sine of the input argument (measured in radians),
   - cosf: calculate the cosine of the input argument (measured in radians),
   - log2f: calculate the base 2 logarithm of the input argument,
   - exp2f: calculate the base 2 exponential of the input argument,
   - fabsf: calculate the absolute value of the input argument.
2) Double precision (double)
   - sqrt: calculate the nonnegative square root of the input argument,
   - sin: calculate the sine of the input argument (measured in radians),
   - cos: calculate the cosine of the input argument (measured in radians),
   - fabs: calculate the absolute value of the input argument.
3) Single precision intrinsics
   - __logf: calculate the fast approximate base *e* logarithm of the input argument,
   - __expf: calculate the fast approximate base *e* exponential of the input argument.