MECHANIKA
CZASOPISMO TECHNICZNE
TECHNICAL TRANSACTIONS
MECHANICS
WYDAWNICTWO POLITECHNIKI KRAKOWSKIEJ
2-M/2010
ZESZYT 8
ROK 107
ISSUE 8
YEAR 107

GRZEGORZ KOLARZ*

UTILIZATION OF NVIDIA CUDA SYSTEM IN THE PROCESS OF SCIENTIFIC COMPUTING

WYKORZYSTANIE SYSTEMU NVIDIA CUDA W PROCESIE OBLICZEŃ NAUKOWYCH

A b s t r a c t

Rapid evolution of markets of computer games and computer animation means that graphic cards are created with the focus on processing large quantities of data necessary for rendering graphics in such applications. Thanks to the utilization of the NVIDIA CUDA software package, it is possible to take advantage of the processing units available on graphic cards, i.e. GPU (Graphic Processing Unit) in scientific calculations. This work presents also the potential of utilization of the processing power of NVIDIA series graphic cards for scientific purposes. It also contains a description of the software environment, which must be met, and presents limitations, which are imposed onto target software under development.

Keywords: parallel processing, graphic cards, CUDA system

S t r e s z c z e n i e

Szybko rozwijające się rynki gier oraz animacji komputerowych sprawiły, że karty graficzne są tworzone z myślą o przetwarzaniu dużej ilości danych na potrzeby renderowania grafiki w tych zastosowaniach. Korzystając z pakietu NVIDIA CUDA istnieje możliwość zaangażowania do procesu obliczeń naukowych jednostek obliczeniowych umieszczonych w kartach graficznych, tzw. GPU (*ang. Graphic Processing Unit*). W niniejszym artykule zaprezentowano możliwości wykorzystania mocy obliczeniowej procesorów kart graficznych z serii NVIDIA do celów naukowych. Zawarto opis środowiska programistycznego, warunków, jakie muszą być spełnione oraz omówiono ograniczenia, które są narzucane na tworzone oprogramowanie.

Słowa kluczowe: przetwarzanie rozproszone, karty graficzne, system CUDA

* MSc. Grzegorz Kolarz, Institute of Applied Informatics, Faculty of Mechanical Engineering, Cracow University of Technology.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

1. Introduction

Evolution of IT technologies and their application in many areas of science mean that the demand for processing power keeps on increasing continuously. There are areas of science where the available processing power is still insufficient (e.g. computer graphics). Increase in the operating frequency of single-core microprocessors used to be one the solution to such a demand for processing power. Currently, it is visible that the said trend is subject to change. Such changes are forced by the problems with heat dissipation and other physical level effects. Such limitations can be bypassed by increasing the number of processors or processing cores in a single machine. Another solution is based on development of computer networks operating under control of specific operating systems, such as MPI or PVM. All these methods utilize the CPU processing power.

On the other hand rapid evolution of markets of computer games and computer animation means that graphic cards are created with the focus on processing large quantities of data necessary for rendering graphics in such applications. There is however also a possibility of taking advantage of the processing units available on the graphic cards, involving the so-called GPU (Graphic Processing Unit) and CUDA framework in general-purpose calculations.

2. Construction of NVIDIA graphic cards

Fig. 1 presents the internal architecture of a G80 type NVIDIA graphic card.

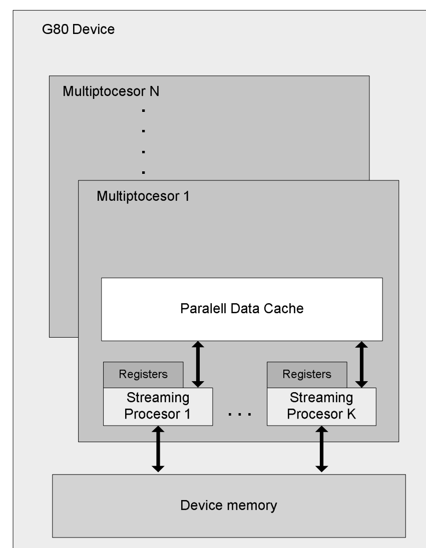

Fig. 1. Simplified model of Architecture of NVIDIA graphic card G80 series

Rys. 1. Uproszczony model architektury karty graficznej NVIDIA serii G80

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

The graphic card core operates at the frequency of 575 MHz and has access to global GDDR3 type memory with the total capacity of 768MB. This memory is used mainly for Scatter and Gatther type operations. Each graphic card comprises 8 multiprocessors. Each multiprocessor further comprises 16 stream microprocessors and has 16 kB of cache memory available (parallel data cache). Memory block of each multiprocessor is freely accessible in stream microprocessors and each processor has also access to the global memory. There is however no way for the multiprocessors to communicate with each other in any other way than via the global memory. Therefore, a complete graphic card has 128 stream processors operating at the frequency of 1350 MHz. Each such processor can execute two MAD type instructions per cycle, i.e. perform operations in the range of multiplication – addition, and a single MUL type operation - i.e. a multiplication. SP operate on 32 floating point data with single precision, in the format compliant with IEEE 754 standard. Even though they are scalar units, they can be connected into vector units, operating in SIMD (Single Instruction Multiple Data) mode [1].

Such a solution is however applied very seldom in graphic cards. Most frequently, vector units are used in such architectures. Utilization of the scalar units is supported by the following justification: three or four scalar units of a stream processor will handle processing a vector or a matrix as well as a vector unit; while in the case of a scalar variable 3/4ths of the processing vector unit would be left wasted.

Parallel Data Cache (PDC) is a special type of multiprocessor memory. It is an indispensable element of each multiprocessor It comprises sixteen memory banks of 1 kB each. It has the total throughput of 32 bits per single processor cycle. This means that all multiprocessors in G80 series graphic cards have the total throughput of approximately 675 Gbit/s. Therefore, PDC memory is almost as fast as registers and can be used as their extension [2].

3. CUDA architecture software development model

Utilization of the processing power available in graphic cards was possible thanks to the fact that the manufacturer of such devices provided a set of software libraries, enabling execution of software code on graphic card processors. Such libraries are based on a processing architecture referred to as CUDA (Compute Unified Device Architecture) [2]. CUDA is a processing environment relying on GPU for general purpose processing. For development of software for NVIDIA CUDA platform, engineers use C language with certain specific extensions.

3.1. Thread organization model

CUDA architecture introduces a new memory organization model. Until now, memory model was based on a linear architecture. A software developer was responsible for providing proper data management, which is depending on the target task at hand (e.g. shifting pointers etc.). The model introduced in CUDA is much different from the traditional memory organization model. It was presented in Fig. 2.

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

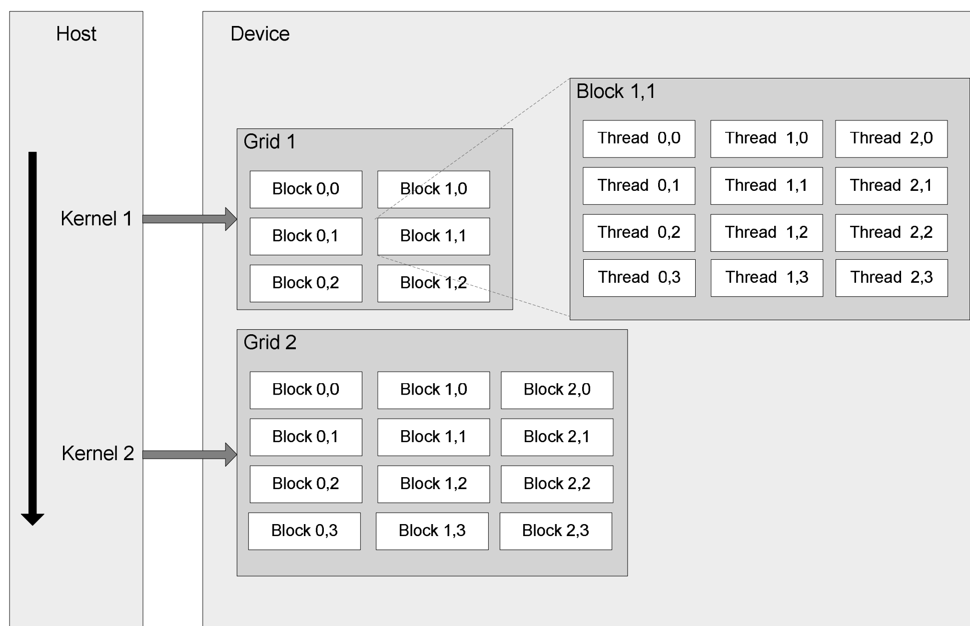

Fig. 2. Thread organization model applied in CUDA system

Rys. 2. Model organizacji wątków zastosowany w systemie CUDA

CUDA allows executing C language kernels directly on underlying hardware. Such functions are processed in parallel by N various threads. Such threads can be further organized in thread blocks. CUDA provides a variable called `threadIdx`, which based on x, y and z coordinates defining location of a thread in a block. Logical organization of individual functions depends on the software developer. Threads inside of a single block can cooperate with each other, exchange information through a shared memory block or synchronize execution [2].

The CUDA architecture allows organizing thread blocks into grids. Just like in the case of individual threads being organized into thread blocks, it is also possible to define multi-dimensional thread block grids. Just like in case of thread blocks, the system provides also a two-dimensional variable `blockIdx`, which allows reading the current location of the given target block in the gird. This means that we can have a two-dimensional thread block grid, where each thread block will represent a three-dimensional thread array. Such a memory organization simplifies management of the calculation processing tasks.

The example shown in Fig. 3 presents practical application of arithmetic operations based on threads.

The function `AdvancedMethod` was pre-pended with a key-word `__global__`, which means that the function may be executed on hardware. The function `AdvancedMethod` receives pointers to arrays located in the global device memory. This function – taking advantage of composite fields in the `blockIdx`, `blockDim` and

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

hreadIdx variables – determines the appropriate portions of data for undertaking further calculations [3].

```
__global__ void AdvancedMetchod(float** A, float** B)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  // do some important operations with A and B
}

int main()
{
  dim3 dimBlock(16, 16);
  dim3 dimGrid(4,4);
  AdvancedMethod <<< dimGrid, dimBlock >>>( A, B );
}
```

Fig. 3. Example of application of arithmetic operations based on threads in CUDA system

Rys. 3. Przykład użycia operacji arytmetycznych na wątkach w systemie CUDA

NVIDIA CUDA operates in the SIMD (Single Instruction Multiple Data) processing mode. Each of the threads in the scope of a single grid executes precisely the same instructions and has the same constant values. Therefore, the method presented above is the suggested approach in terms of data processing.

In order to call the function, it is necessary to set the execution configuration. This is closely related with definition of parameters covering kernel function calls. In such a call, it is necessary to define the grid size (in this case – 4x4) and the size of a thread block attributed to each grid (in this case – 16x16). In this particular case, the function `AdvancedMethod` will be executed 4096 times on 4096 various threads. The formal declaration of the execution configuration is as follows [3]:

```
MethodName<<Dg, Db, Ns>> (parameterList),
```

where:

- `Dg` – type of dim3, and defines the size of the grid, on which individual thread blocks will be executed. `Dg.x * Dg.y` is the number of thread blocks, which will be executed,
- `Db` – type of dim3, and defines the size of a single thread block. The number of threads in a block is calculated as `Db.x * Db.y * Db.x.`
- `Ns` – optional argument. This variable is of `size_t` type and defines the number of bytes in the shared memory block, which is additionally allocated. By default, this variable is assigned value 0.

The CUDA architecture is characterized by low cost (in terms of time) related with thread management. Creating a group of 32 threads with initialization of associated registers and memory occupies just a single processor cycle. Context switching is also very

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

"cheap" in terms of time. This means that at every single time we can select for our calculations a completely different set of threads and the whole operation of switching into a different set of threads costs nothing (in terms of time).

3.2. Function types

As mentioned before, CUDA allows executing C language functions directly on underlying hardware. To make that possible, it is necessary to utilize the keyword `__global__`. When this keyword is used, execution of the given method is possible only from the host level. This function prevents execution of recurrent calls and prohibits declaration of static variables in the function body. Parameters to such functions must be sent through the shared memory, while the function itself must return void type.

Another keyword used for definition of declared functions is the `__device__` qualifier. This one means that the function will be executed on underlying hardware and may be called from another method executed on underlying hardware. Just like the `__global__` qualifier, this one also prevents recurrent calls and prohibits declaration of static variables in the function body.

The keyword `__host__` defines functions which may be executed on the host and may be called from within a different function executed on the host.

3.3. Kinds and types of variables used in applications.

The CUDA architecture, apart from function qualifiers, offers also internal types of variables. Such variable types extend the existing variable types used in standard-defined C language. Such new variable types provide better support in order to enable operation in multi-dimensional environments. Software developers have at their disposal variable types with one, two, three or even four components. The digit at the end of the variable type identifies the number of dimensions. And so, `int1` indicates that this variable has only one component X, while `int2` indicates that such a variable type has two components i.e. X and Y. `Int2` variable type is initialized as follows:

```
int2 var(3,4);
```

While reference to individual values is possible thanks to the following:

```
int1 varX = var.x;

int1 varY = var.y;
```

Three and four dimensional variables are processed in a similar way. Such variable types are also available for various number types: both integer or floating point, signed and unsigned etc. Therefore, a software developer has the following data types at disposal: `charN, ucharN, shortN, ushortN, intN, uintN, longN, ulongN, floatN, double2`, where N = 1,2,3;

There is also an additional, special built-in variable type – `dim3`. It is a variable type based on `uint3`, where variable constructor accepts only one or two parameters. The remaining parameters are assigned a default value of 1 [2].

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

4. Example of CUDA architecture applications – SAXPY algorithm

The SAXPY (Scalar Alpha X Plus Y) algorithm is a typical operation supported in the BLAS (Basic Linear Algebra Subprograms) package, used heavily in vector processors. SAXPY is a combination of a scalar multiplication and addition of vectors. Its mathematical formula is as follows:

$$\mathbf{y} = \alpha * \mathbf{x} + \mathbf{y}$$

where:

α – scalar value,
x, **y** – vectors.

Its typical serial implementation in C language is shown in fig. 4.

```
void saxpy ( int count, int a, int* x, int* y ){
    for (int i = 0; i < count; i++){
        y[i] = a * x[i] + y[i];
    }
}
```

Fig. 4. Serial implementation of SAXPY algorithm

Rys. 4. Strukturalna implementacja algorytmu SAXPY

Such function receives the following input parameters: number of elements in the array, scalar value for multiplication and two vectors (X and Y). Next, in the loop, the function multiplies all the elements by the scalar value. The loop will be executed the number of times corresponding to the array dimension [3].

Its implementation on the CUDA platform is shown in fig. 5:

```
__global__ saxpy(int count, int a, int* x, int* y){
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if ( i<count){
        y[i] = a * x[i] + y[i];
    }
}

int n_blocks = ( n + 255 ) / 256;
saxpy <<< n_blocks, 256 >>> ( n, 3, x, y);
```

Fig. 5. Implementation of SAXPY algorithm in CUDA system

Rys. 5. Implementacja algorytmu SAXPY w systemie CUDA

In this case, the function is called exactly once in each operating thread. For calculation purposes, thread blocks will be created with 256 threads per block. It means that this implementation will be 256 times faster than its serial equivalent.

BPK BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ

5. Conclusions

Software development for NVIDIA hardware prior to introduction of the CUDA API required software developers to have very good, low-level knowledge of graphic card architecture. Additionally, the developed software was closely related with the given architecture and its translation into a different platform was a relatively difficult task. The aforementioned limitations in the majority of cases prevented software developers from utilizing such powerful processing units for general-purpose calculations. The introduction of the CUDA API not only frees software developers from dependence on the particular hardware platform and its drivers, but also introduces a new software development model for such processing units, based on well-known C language. Such a solution does not require learning new concepts or new tools. Thanks to that, all software developers using classic central processing units can also start utilizing GPU for calculation purposes. CUDA as a software development platform manages the thread life cycle, memory passed between the graphic card and the host as well as many more different operation issues.

On the other hand, CUDA is still not perfect. Even though there are an increasing number of solutions using CUDA as a base platform, there are still no solutions to multiple existing problems. Such problems include for example slow operations on double precision values or no support for recurring functions executed on the underlying hardware. Such restrictions mean that the ability to use this platform is very limited in some cases.

There is however no doubt that the NVIDIA CUDA framework provides very universal, effective data processing environment. Knowing well its pros and cons, it is possible to create efficient applications, the purpose of which may substantially exceed the primary purpose of graphic cards i.e. 3D processing and computer games.

References

[1] *NVIDIA CUDA Architecture.* Documentation available on website: http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf, version 1.1, 2009.

[2] *NVIDIA CUDA Programming Guide Version 2.2.* Documentation available on website: http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf, version 2.2, 2009.

[3] NVIDIA CUDA Reference Manual Version 2.2. Documentation available on website: http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/CUDA_Reference_Manual_2.2.pdf, 2009.

BIBLIOTEKA CYFROWA POLITECHNIKI KRAKOWSKIEJ