COMPARISON BETWEEN RESEARCH DATA PROCESSING CAPABILITIES OF AMD AND NVIDIA ARCHITECTURE-BASED GRAPHIC PROCESSORS

V. A. Dudnik, [*] ***V. I. Kudryavtsev, S. A. Us, M. V. Shestakov***

National Science Center "Kharkov Institute of Physics and Technology", 61108, Kharkov, Ukraine

(Received January 26, 2015)

A comparative analysis has been made to describe the potentialities of hardware and software tools of two most widely used modern architectures of graphic processors (AMD and NVIDIA). Special features and differences of GPU architectures are exemplified by fragments of GPGPU programs. Time consumption for the program development has been estimated. Some pieces of advice are given as to the optimum choice of the GPU type for speeding up the processing of scientific research results. Recommendations are formulated for the use of software tools that reduce the time of GPGPU application programming for the given types of graphic processors.

PACS: 89.80.+h, 89.70.+c, 01.10.Hx

1. INTRODUCTION

The development of various architectures of graphic processors attained its maximal variety in the nineties of the last century. At that time, a great many companies at the computer graphics market (S3 Graphics, Matrox, 3D Labs, Cirrus Logic, Oak Technolog, Realtek, XGI Technology Inc., Number Nine Visual Technologies, etc.) made overtures concerning the architectures of graphical processors. However, by the present time, as a result of stiff competition, out of a variety of the proposed architectures two architectures of operating companies NVIDIA and AMD have taken the key positions at the market. Yet, when considering modern GPU structures of the AMD and NVIDIA processors, one can notice that there is more similarity than difference between them (and this being despite a severe architecture competition). The reasons for this outcome of GPU-processor development are similar to the results of the development of other high technology products (automobiles, aircraft, etc.), because in the process of engineering design improvement we always have the interchange, borrowing and use of most happy ideas with the result that the competing companies arrive at very similar engineering solutions.

2. *AMD* AND *NVIDIA* PROCESSOR STRUCTURE

The similarity between the architectures of AMD and NVIDIA processors can be explained by the fact that from the outset the GPU microarchitecture was organized quite differently than that of ordinary CPU. At the very beginning of the GPU development, the graphics tasks implied an independent parallel data processing, and therefore, the GPU architecture, unlike the CPU architecture, was multithreaded right from the start. Besides, the fundamental principles of the GPU organization were initially general for video accelerators of all manufacturers, because they had a single target task, namely, shader programs. Therefore, the general GPU structure of different manufacturers differed only slightly. The differences concerned the details of microarchitecture realization. The internal organization of the GPU is similar for both the AMD architecture and the NVIDIA architecture, i.e., it consists of a few tens (30 for NVIDIA GT200, 20 for Evergreen, 16 for Fermi) of central processing elements referred to in the NVIDIA nomenclature as Streaming Multiprocessors, and in the ATI terminology as SIMD Engine (miniprocessors). They can simultaneously perform a set of computational processes, i.e., threads. Each miniprocessor has a local storage of size 16 KB for GT200, 32 KB for Evergreen and 64 KB for Fermi (in actual fact, this is a programmed L1 cache). The local storage is common for all the threads executed in the miniprocessor. Apart from the local storage, the miniprocessor has also another storage area, being approximately four times larger in storage capacity in all the architectures under consideration. It is shared in equal parts by all the executive threads, these are registers for storing the temporary variable and intermediate computation results. Each miniprocessor has a great number of computation modules (8 for GT200, 16 for Radeon and 32 for Fermi), but all of them can carry out the same instruction with one and the same software address. However, the operands can be different in this case, one's own for different threads. For example, the instruction "to

[*]Corresponding author. E-mail address: vladimir-1953@mail.ru

add the contents of two registers" is executed simultaneously by all computing devices, but the registers are used different. If however, because of program branching, the threads came apart in their way of running the code, then the so-called serialization takes place. That is, not all computation modules are involved, because the threads supply different instructions for execution, while the computation module block can carry out, as stated above, only the instruction with one address. The computational efficiency falls down in this case with reference to the maximum capability. Another peculiarity of the GPU in comparison with the CPU is the absence of stack. On account of a great many simultaneously executing threads, the GPU does not provide the stack, which might store the function parameters and local variables (simultaneously running 10000 threads call for an enormous storage). Therefore, there is no recursion in the GPU, and instead of the call, the functions are substituted during compilation into the GPGPU program code. The AMD have obtained a full-scale support for general-purpose computations, starting with the Evergreen family, where also Direct X II specifications were first realized. Of importance is the from the brginning use of the AMD technology named VLIW (Very Long Instruction Word). The AMD graphics processors have the same number of registers as, for example, GT200 have, the difference being that these are 128-bit vector registers (e.g., using one single-cycle instruction, the number a1xb1+a2xb2+a3xb3+a4xb4 or the two-dimensional vector (a1xb1+a2xb2, a3xb3+a4xb4) can be computed). Until recently, NVIDIA supported only scalar simple instructions operating on scalar registers and realizing a simple classical RISC, but since 2009 the NVIDIA programmers have presented a further development of the CUDA platform, namely, the architecture FERMI, and further, Kepler. The support of one-precision and double-precision floating point computations realized in a new architecture, was one of the key demands of the high-performance computing market.

3. DIFFERENCES BETWEEN *AMD* AND *NVIDIA* MICROSTRUCTURES

Here we mention the principal potentialities of advanced technical solutions of NVIDIA company, which specified the main differences between GPU architectures proposed at present by AMD and NVIDIA. These differences are due to the initial tendency of using the NVIDIA graphic processors not only for processing the computer graphics. The NVIDIA also marketed its architecture as a means for high-performance computing. That implied both a high speed of computation operations and a high reliability combined with high programmability. Therefore, the last realizations of GPU by NVIDIA have incorporated the possibilities of finding and correcting errors in the operating storage and cache-memory subsystems. That provided the fault tolerance and performance reliability of computational algorithms comparable with the CPU. Essential modifications have been made in the memory structure of NVIDIA graphic processors. The appearance of L2-cache in the FERMI architecture has radically (by a factor of a few tens) accelerated operations with memory, thereby enabling one to organize the internal memory manager, which is sufficiently rapid to allocate the memory by the machine language C memory control functions (malloc/free) for each thread in the course of operation. Paralleling of global videomemory access has been improved and the general address space has been realized for all the memory of CPU and GPU, thereby making it possible to unite in a single address space all the memory visible for the computational thread. Besides, the changes in the memory structure have provided a way of using recursion functions (Fig.1).

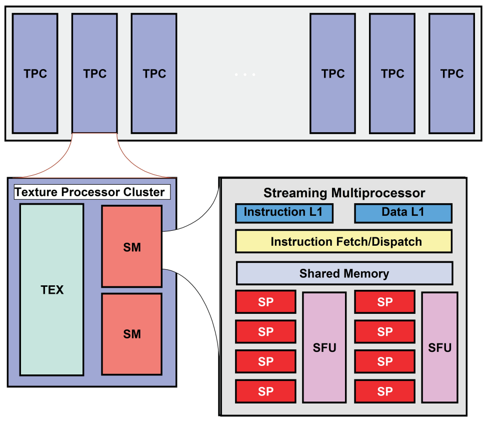

Fig.1. *NVIDIA GPU structure*

The simultaneous optimized execution of several kernels, realized in the NVIDIA architecture, permits organization of simultaneous running of several CUDA functions of the same application provided that one CUDA function cannot fully load the computational capability of the GPU (as the GPU analogue of the multitask mode for multi-core CPUs). Two interfaces for copy operations, realized in the NVIDIA GPU, have provided practically twice as fast data exchange due to simultaneous copying of data from the memory of CPU multiprocessors to the GPU and vice versa (from the GPU memory to the CPU memory). Unlike the NVIDIA GPU, new models of AMD video adapters differ from the previous versions practically only in quantitative characteristics. They are still based on the dated architecture Graphics Core Next (GCN) Tahiti. This architecture forms the basis for all present-day engineering solutions of the company, and even the latest graphic chip Hawaii differs from Tahiti only in a greater number of execution devices, and in some computational power modifications (e.g., as a support of a greater number of simultaneously executed instruction streams), as well as in support of some options Direct X 11.2, and in the improved technology AMD Power Tune. The present-day AMD graphic processors still have only one global read-write memory, and many different sources of texture memory and constant memory, both being read-only memories. The peculiarity of the constant memory is the availability of caching at the access for all GPU streams to one data area (operates as quickly as the registers do). Another peculiarity of the texture memory of AMD graphic processors is the read caching (from 8 Kb on a per data flow processor basis), and also, the access to the memory in real coordinates. Though the L1-L2 cache sizes in the NVIDIA and AMD cards are approximately similar, this being evidently due to optimality requirements from the point of view of game graphics, the latency of access to these caches is essentially different. The access latency for the NVIDIA is greater, and the texture caches in the NVIDIA GeForce first of all aid to reduce the load on the memory bus rather than immediately accelerate the data access. This is not noticeable in graphics programs, but is of importance for general-purpose programs. Meanwhile, in the AMD Radeon the latency of the texture cache is lower, but the latency of miniprocessors local memory is higher. The following example can be offered. For optimum matrix multiplication on the NVIDIA cards it is best to use the local memory, loading it with the matrix, one block at a time, while in the AMD case, it is better to rely on a low-latency texture cache, reading the matrix elements when required. But it appears to be a rather fine optimization and for the algorithm already adapted in principle to the GPU. (Fig.2).

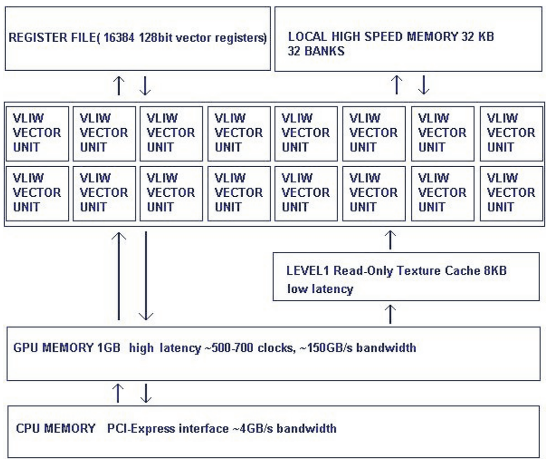
REGISTER FILE(16384 128bit vector registers)
LOCAL HIGH SPEED MEMORY 32 KB 32 BANKS
VLIW VECTOR UNIT
LEVEL1 Read-Only Texture Cache 8KB low latency
GPU MEMORY 1GB high latency ~500-700 clocks, ~150GB/s bandwidth
CPU MEMORY PCI-Express interface ~4GB/s bandwidth

Fig.2. *AMD GPU structure*

The AMD uses its own instruction placement format in the computer code. They are arranged not in succession (according to the program listing), but by sections. First goes the section of conditional transfer instructions, which comprise references to the sections of continuous arithmetic instructions corresponding to different transition paths. They are called VLIW bundles. These sections comprise only arithmetic instructions with the data from the registers or the local memory. This organization simplifies the control over the instruction flow and its delivery to the executing devices. It is all the more reasonable, considering that the VLIW instructions are rather large in size. The AMD structure also provides sections for the memory access instructions. In the NVIDIA GPU the instructions are arranged according to the listing (for possibilities of debugging, testing and optimization). When programming the NVIDIA GPU, this permits the use of the methods developed when writing programs for the CPU.

4. DIFFERENCES BETWEEN *AMD* AND *NVIDIA* IN THE SOFTWARE FOR *GPU* APPLICATION DEVELOPMENT

The supporting software in using Radeon products for fast computations is still essentially behind the development of the hardware (unlike the NVIDIA situation). The AMD-manufactured OpenCL-compiler once too often produced an erroneous code or refused to compile the code from the correct source code. And only recently a release with a higher operational capability has appeared. The absence of function libraries is also typical of the AMD. For example, there are no sine, cosine and exponent for double-precision real numbers. For programming the applications for the AMD GPU, the firm-specific technologies Compute Abstraction Layer (CAL) and Intermediate Language (IL) have been designed. The CAL technology serves for writing the code interacting with the GPU and is executed by the CPU, while the IL technology permits writing the code, which will be executed directly by the GPU. The code for the AMD GPU is designed as shaders. Below we give an example of the program code for the AMD IL.

```
 We calculate 0.z being the global flow
 identifier(uint)
umad r0.__z_, r0.wwww, cb0[0].yyyy,
r0.zzzz
 ; We save the first part of the data in the
 register
 ftoi r1.x___, vWinCoord0.xxxx
  mov r1._y__, r0.zzzz
  mov  r1.__z_, cb[0].xxxx
  mov  r1.___w, l0.yyyy ;
We calculate the output buffer shift g[] umul
r0.__z_,r0.zzzz,l0.wwww
 ; Save the first part of the data in the
 storage
mov g[r0.z+0].xyzw, r1.xyzw
; Load the texture data i0 ;
Preliminarily, we transfer the coordinates
 to float and add 0.5
itof r0.xy__, r0.xyyy
 add r0.xy__, r0.xyyy, l0.zzzz
sample_resource(0)_sampler(0)_aoffimmi(0
,0,0) r1, r0
; Save the
second part of the data in the
 storage
 mov g[r0.z+1].xyzw, r1.xyzw
; Exit from the main function endmain
; Program code termination
end
```

The IL code is looks like the assembly code, but there is no sense in trying to make use of optimization procedures typical of the assembler program (rearrangement of independent operations, precomputation of constant operators), because this is a pseudoassembler program, and only the IL compiler can carry out a correct optimization of the code. The usage of the second constituent of the AMD development tools, i.e., Compute Abstraction Layer (CAL), is necessary for organization of the interaction between the CPU-executed program parts with the computing procedures executed by the GPU:

- Driver initialization
- Obtaining information about all supported GPU
- Memory allocating and copying
- Compilation and loading of the GPU kernel
- Kernel run for execution
- Operation synchronization with CPU

Unlike the NIVIDIA CUDA, which has both the Run-time API and Driver API, the AMD need only the Driver API. Below we give the program code fragment for the CAL, which executes data copying into the GPU memory:

```
unsigned int pitch;
 unsigned char* mappedPointer;
unsignedchar*dataBuffer; char*dataBuffer;
 CALresult result =
calResMap(
 (CALvoid**)&mappedPointer, &pitch, resource,
  0 );
unsigned int width;
 unsigned int height;
  unsigned int elementSize = 16;
  if( pitch > width ) {
    for( uint index = 0; index < height;
     ++index )
    {    memcpy( mappedPointer  + index *
         pitch * elementSize,
         dataBuffer     + index * width *
          elementSize,
          width * elementSize );
    }
} else {
    memcpy( mappedPointer, dataBuffer,
     width * height * elementSize );
}
```

It can be seen that the program for the CAL represents the program written in the language similar to C, and comprises specific subroutine call sequences. However, irrespective of a rigid binding to the Windows 7 platform, the use of API Direct Compute 5.0 seems more promising for programming Radeon video cards, because it is much simpler than OpenCL and is expected to be more stable.

5. DEVELOPMENT TOOLS FOR *NVIDIA GPU*

The hardware performance capabilities of the NVIDIA GPU architecture have made it possible to develop the software solution for improving the processes of creating and checking of CUDA applications - NVIDIA NEXUS. The C++ support is the most important peculiarity of the NVIDIA software development tools. Furthermore, the interaction is provided between the mechanisms of graphics processing and the tools for executing general-purpose computations. For data processing through the CUDA mechanism, the Open GL library can be used; as well the OpenCL is available for the development of applications. Below, by way of example, we give the program code fragment for NVIDIA CUDA:

```
#include <stdio.h>
#include <stdio.h>
#include <assert.h>

#include <assert.h>
#include <cuda.h>

#include <cuda.h>
int main(void) int main(void)
{

float *a_h, *b_h; // pointers to host memory

float *a_h, *b_h; float *a_d, *b_d;
// pointers to device memory
float *a_d, *b_d;
int i;
int i;
// allocate arrays on host
a_h = (float *)malloc(sizeof(float)*N);

a_h = (float*)malloc(sizeof(float)*N);

b_h = (float *)malloc(sizeof(float)*N);

b_h = (float *)malloc(sizeof(float)*N);
// allocate arrays on device
cudaMalloc((void **) &a_d, sizeof(float)*N);

cudaMalloc((void **) &a_d, sizeof(float)*N);

cudaMalloc((void **) &b_d, sizeof(float)*N);

cudaMalloc((void **) &b_d, sizeof(float)*N);
// send data from host to device: a_h to a_d

cudaMemcpy(a_d, a_h, sizeof(float)*N,
cudaMemcpyHostToDevice);
```

```
cudaMemcpy(a_d, a_h, sizeof(float)*N,
cudaMemcpyHostToDevice);
// copy data within device: a_d to b_d
cudaMemcpy(b_d, a_d, sizeof(float)*N,
cudaMemcpyDeviceToDevice);
cudaMemcpy(b_d, a_d, sizeof(float)*N,
cudaMemcpyDeviceToDevice);

// Defect deletion
MedianFilter(b_d);
// Data retrieval from device: b_d to b_h
cudaMemcpy(b_h, b_d, sizeof(float)*N,
udaMemcpyDeviceToHost);
cudaMemcpy(b_h, b_d, sizeof(float)*N,
cudaMemcpyDeviceToHost);
// cleanup
// Resource deallocation
free(a_h);

free(b_h);

free(a_h);

free(b_h);

cudaFree(a_d);cudaFree(b_d);
cudaFree(a_d); cudaFree(b_d);
```

It can be seen that the program for NVIDIA GPU is very similar in its form and structure to the customary programs written in C++ for the CPU. But, perhaps, the most essential novelties realized in NVIDIA NEXUS are the possibilities of full-rate application debugging into the GPU. Previously, it has been necessary to use the GPU emulation program for debugging. The use of NVIDIA NEXUS for Microsoft Visual Studio will make it possible to avoid the majority of the existing problems of debugging, and thus, to increase the speed of application development. Let us consider in more detail the new debugging features of NEXUS. The NEXUS debugger supports the code debugging by CUDA C and HLSL directly on the GPU hardware in the working space of Visual Studio 2008, and includes the following performance capabilities:

- **CUDA home page** provides complete information about CUDA run state in the user application. Users can filter and obtain detailed information on the exception cases, breakpoints, facts added to the database, MMU errors, and also, can easily switch over for problem debugging.
- **CUDA Warp Watch**provides a more efficient technique of resident threads navigation and threads state visualization at the point of warp.
- **Supports graphics and GPU computing.**A simple debugging of shaders or the programs of scientific and technical computations directly into the GPU.
- **Parallel-aware**- debugging of applications using thousands of the data processing streams (or graphic primitives).
- **Source breakpoints** - breakpoint at any points (using the hardware evaluation of conditions breakpoints);
- **Memory inspection**- direct control and image of the GPU memory with the use of Visual Studio Memory Window.
- **Data breakpoints** - writing breakpoint wherever in the memory;
- **Memory Checker**- halting on running-out of the allocated memory.
- **Trace** - journaling of effects and events executed by CPU and GPU in the chosen correlated line. It includes:
  1.CUDA C, DX10, Open GL and Cg API calls;
  2.GPU - Host memory transfers;
  3.GPU workload executions;
  4.CPU core, thread and process events;
  5.Custom user events – Mark custom events or time ranges using a C API.

A further tool is the NEXUS Analyzer that supports tracing and profiling of GPU applications; acquisition and analysis of information on the level of core efficiency, including hardware productivity counters.

6. COMPARISON BETWEEN THE GRAPHIC CAPABILITIES OF *NVIDIA* AND *AMD* ARCHITECTURE

The comparison between the AMD and NVIDIA architecture capabilities of data displaying shows that mainly they are similar in both the rate and quality. Among most useful capabilities incorporated into the graphic architectures we may point out the technology, which provides a combined connection of several displays to the personal computer and, thus, permits their use as a single large display consisting of several parts. In NVIDIA, this technology is called the multimonitor SLI, and the similar technology of AMD is known as Eyefinity. Comparing the capabilities of Eyefinity and multimonitor SLI, it is worthy of note that the AMD multimonitor appears a more widespread and more advanced technology. Unlike the multimonitor NVIDIA SLI technology, which is provided only for rather expensive video accelerators of professional series Quadro, the AMD Eyefinity technology is maintained practically for all AMD-made display cards. Besides, the AMD Eyefinity capabilities of controlling the inline monitors are of larger scale than those of NVIDIA. And another thing, the Eyefinity offers the possibility of connecting more monitors to one video accelerator. In AMD graphic processors (starting with RV870) the image output unit has been updated in such a way that the chip supports the imaging on the output devices in quantity 6 pieces inclusive. The number of supported monitors depends on a specific card capability and can be three or six (through the monitor interface DisplayPort). Multimonitor configurations can operate in the clone and desktop extend modes. One large image can be composed of several monitors; this is applicable for both theWindows desktop visual display, and the full-screen video and 3D applications. This special feature is supported in the operating systems Windows Vista, Windows 7 and Windows 8, Linux, and also, in other OS (Fig.3).

Fig.3. *Multimonitor configurations in the clone and desktop extend modes*

The AMD has announced the team work with display manufacturers, in particular, with Samsung Company. They come out with special versions of monitors having screens 23″ in size, supported by the 1920x1080 pixels resolution, to the interfaces DisplayPort, DVI and VGA, and also, a it has very narrow screen frame, 7 to 8 mm in size. The use of multimonitor connection may appear very efficient in many cases, where the operation of large-size monitors permits one both to raise labor productivity and to reduce the performance time. These are the work with large electronic circuit diagrams and complicated equipment drawings, the program development, the generation and analysis of data represented as large-size tables, etc. It should be noted that the cost of a large monitor composed of several monitors of smaller size turns out to be several orders of magnitude lower than the cost of a ”monolithic” (one-piece) monitor; and that may essentially extend the field of application of these monitors.

7. CONCLUSIONS

In view of the foregoing, some general recommendations on the use of graphic accelerators AMD and NVIDIA can be formulated:

- To attain the maximum graphic performance capabilities with minimum expenses (support of multiscreen monitors, 3D graphics formation), it is more reasonable to use the AMD architecture.
- To create programs relying on the GPGPU potentialities, it is preferable to use NVIDIA video accelerators that offer more powerful,different-level and versatile means for program development and debugging.

References

1. 1. A. Zubinsky. NVIDIA Cuda: unification of graphics and computations. (In Russian). 8 May. 2007. http://itc.ua/node/27969
2. D. Luebke. Graphics CPU-not only for graphics(In Russian). http://www.osp.ru/os/2007/02/4106864/
3. 3. David Luebke, Greg Humphreys. How GPUs Work.. IEEE Computer, February, 2007. IEEE Computer Society, 2007.

СРАВНЕНИЕ ВОЗМОЖНОСТЕЙ ПО ОБРАБОТКЕ РЕЗУЛЬТАТОВ НАУЧНЫХ ИССЛЕДОВАНИЙ ДЛЯ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ АРХИТЕКТУР AMD И NVIDIA

В. А. Дудник, В. И. Кудрявцев, С. А. Ус, М. В. Шестаков

Сделано сравнительное описание возможностей аппаратных и программных средств двух наиболее распространённых современных архитектур графических процессоров (AMD и NVIDIA). Особенности и различия архитектур GPU иллюстрированы примерами фрагментов программ GPGPU. Приведена также сравнительная оценка временных затрат на их разработку. Даны советы по оптимальному выбору типа GPU для ускорения обработки результатов научных исследований. Сформулированы рекомендации по использованию программных инструментов, позволяющих сократить время разработки GPGPU-приложений для этих типов графических процессоров.

ПОРІВНЯННЯ МОЖЛИВОСТЕЙ З ОБРОБКИ РЕЗУЛЬТАТІВ НАУКОВИХ ДОСЛІДЖЕНЬ ДЛЯ ГРАФІЧНИХ ПРОЦЕСОРІВ АРХІТЕКТУР AMD І NVIDIA

В. О. Дуднік, В. І. Кудрявцев, С. О. Ус, М. В. Шестаков

Зроблено порівняльний опис можливостей апаратних і програмних засобів двох найбільш поширених сучасних архітектур графічних процесорів (AMD і NVIDIA). Особливості і відмінності архітектури GPU ілюстровані прикладами фрагментів програм GPGPU. Приведена також порівняльна оцінка часових витрат на їх розробку. Дані поради з оптимального вибору типу GPU для прискорення обробки результатів наукових досліджень. Сформульовані рекомендації по використанню програмних інструментів, що дозволяють скоротити час розробки GPGPU-додатків для цих типів графічних процесорів.