SOFTWARE DEVELOPMENT TOOLS USING GPGPU POTENTIALITIES

V.A. Dudnik*, V.I. Kudryavtsev, T.M. Sereda, S.A. Us, M.V. Shestakov

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

(Received December 3, 2010)

The paper deals with potentialities of various up-to-date software development tools for making use of graphic processor (GPU) parallel computing resources. Examples are given to illustrate the use of present-day software tools for the development of applications and realization of algorithms for scientific-technical calculations performed by GPGPU. The paper presents some classes of hard mathematical problems of scientific-technical calculations, for which the GPGPU can be efficiently used. is possible. To reduce the time of calculation program development with the use of GPGPU capabilities, various dedicated programming systems and problem-oriented subroutine libraries are recommended. Performance parameters when solving the problems with and without the use of GPGPU potentialities are compared.

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

1. INTRODUCTION

As far back as about five years, use of GPU parallel computing resources for the development of applications and realization of algorithms for scientific-technical calculations was quite an exotic technique. To use the GPGPU special versions of graphic drivers, rather specific operating system settings were required, while only general programming aids provided by GPU manufacturing firms were available. The CUDA architecture (NVIDIA product) [1,2,3] has turned out to be especially successful for using the GPU capacity to speed-up scientific-technical calculations. This has given such a vigorous impetus to the development of GPGPU tools that today their capabilities are included as standard not only for providing of all up-to-date graphic adapters but also in the most popular operational systems Windows 7 and Linux. A wide use of GPGPU has stimulated the development of software environment, which speeds up the GPGPU program development and simplifies an access to the GPU computational power from scientific-technical calculation programs.

2. BASIC DIRECTIONS IN ELABORATION OF SOFTWARE DEVELOPMENT TOOLS

The basic directions in elaboration of software development tools include:

- unification of access to GPU hardware and standardization of GPGPU programming languages;
- automatic multisequencing compilers (applying directives);
- creation of problem-oriented tools and libraries for application development;
- enhancement of architecture and GPGPU hardware and software platforms.

Now we consider these directions in greater detail.

3. UNIFICATION OF ACCESS TO GPU HARDWARE AND STANDARDIZATION OF GPGPU PROGRAMMING LANGUAGES

At present the realization of Open CL interface [4,5] is the most remarkable example of programming standardization and provision of hardware-independent GPGPU software development tools. The development of Open CL program interface that provides hardware- and mostly software-independent access to the GPGPU tools has made it possible to simplify the creation of the programs taking advantage of the GPGPU means, to extend considerably the range of their use and also, to increase their lifetime. The Open CL specification determines the program interface for computer programs concerned with parallel computations at various GPUs and central processors (CPU). The programming language based on C99 standard and the application programming interface (API) are incorporated into the Open CL programming language. The Open CL interface provides parallelism at the level of instructions and realizes the GPGPU technique at the level of data. The Open CL standard is an entirely open standard and its usage is not liable to license fees. The fragment of program given below demonstrates Open CL possibilities as for standardization of the process of engaging the computation fragments realization by GPGPU means in the C++ program.

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

```
// Demo OpenCL application to compute a simple vector addition
#include <stdio.h> #include <stdlib.h> #include <CL/cl.h>
// OpenCL source code
const char* OpenCLSource[] = { "__kernel void VectorAdd(__global
int* c, __global int* a,__global int* b)",
  "{",
  "  // Index of the elements to add \n",
  "  unsigned int n = get_global_id(0);",
  "  // Sum the n'th element of vectors a and b and store in c \n",
  "  c[n] = a[n] + b[n];",
  "}"
};
// Some interesting data for the vectors
int InitialData1[20] =
{37,50,54,50,56,0,43,43,74,71,32,36,16,43,56,100,50,25,15,17}; int
InitialData2[20] =
{35,51,54,58,55,32,36,69,27,39,35,40,16,44,55,14,58,75,18,15};
// Number of elements in the vectors to be added
#define SIZE 2048

// Demo OpenCL application MAIN - PART 1
int main(int argc, char **argv) {
   // Two integer source vectors in Host memory
    int HostVector1[SIZE], HostVector2[SIZE];
     // Initialize with some interesting repeating data
    for(int c = 0; c < SIZE; c++)
    {
    HostVector1[c] = InitialData1[c%20];
    HostVector2[c] = InitialData2[c%20];
    }
     // Create a context to run OpenCL on our CUDA-enabled NVIDIA GPU
    cl_context GPUContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                    NULL, NULL, NULL);
     // Get the list of GPU devices associated with this context
    size_t ParmDataBytes; clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, 0, NULL,
    &ParmDataBytes);
    cl_device_id* GPUDevices = (cl_device_id*)malloc(ParmDataBytes);
    clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, ParmDataBytes, GPUDevices, NULL);
    // Create a command-queue on the first GPU device
    cl_command_queue GPUCommandQueue = clCreateCommandQueue(GPUContext,
                  GPUDevices[0], 0, NULL);
// Allocate GPU memory for source vectors AND initialize from CPU memory
    cl_mem GPUVector1 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, HostVector1, NULL);
    cl_mem GPUVector2 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY |
              CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, HostVector2, NULL);
     // Allocate output memory on GPU
    cl_mem GPUOutputVector = clCreateBuffer(GPUContext, CL_MEM_WRITE_ONLY,
              sizeof(int) * SIZE, NULL, NULL);

     // Create OpenCL program with source code
    cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext, 7,
              OpenCLSource, NULL, NULL);
     // Build the program (OpenCL JIT compilation)
    clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL);
    // Create a handle to the compiled OpenCL function (Kernel)
    cl_kernel OpenCLVectorAdd = clCreateKernel(OpenCLProgram, "VectorAdd", NULL);
    // In the next step we associate the GPU memory with the Kernel arguments
    clSetKernelArg(OpenCLVectorAdd, 0, sizeof(cl_mem),(void*)&GPUOutputVector);
    clSetKernelArg(OpenCLVectorAdd, 1, sizeof(cl_mem), (void*)&GPUVector1);
```

```
    clSetKernelArg(OpenCLVectorAdd, 2, sizeof(cl_mem), (void*)&GPUVector2);
    // Launch the Kernel on the GPU
    size_t WorkSize[1] = {SIZE};  // one dimensional Range
    clEnqueueNDRangeKernel(GPUCommandQueue, OpenCLVectorAdd, 1, NULL,
         WorkSize, NULL, 0, NULL, NULL);

// Copy the output in GPU memory back to CPU memory
    int HostOutputVector[SIZE];
    clEnqueueReadBuffer(GPUCommandQueue, GPUOutputVector, CL_TRUE, 0,
               SIZE * sizeof(int), HostOutputVector, 0, NULL, NULL);
    // Cleanup
    free(GPUDevices);
    clReleaseKernel(OpenCLVectorAdd);   clReleaseProgram(OpenCLProgram);
    clReleaseCommandQueue(GPUCommandQueue); clReleaseContext(GPUContext);
```

4.AUTOMATIC MULTISEQUENCING COMPILERS

The compilers of this sort perform automatic analysis of the program structure and the data used in it, as well as they provide division of the computing process into separate parts intended for their parallel processing by x32 or x84 CPU and GPU tools, according to the directives included into the program text. After that, the compiler generates the program code that provides optimization of the program cyclic areas execution by automating the GPU using. The automatic multisequencing compiler FORTRAN PGI 2010 comprises PGI Accelerator FORTRAN and C99 compilers [6] thus providing support of the architecture system Intel and AMD x64 NVIDIA CUDA operating under the control of the operational systems Linux, Mac OS X and Windows. The programming system HMPP may be mentioned as another example of automatic multisequencing compiler realization. The system HMPP comprises the preprocessor-analyzer of program structure prototypes, which are the most optimal for computational speedup with the help of hardware. Then generation of the program code for the GPU is executed in NVIDIA CUDA, AMD CAL/IL or OpenCL languages, and the program code for CPU Intel or AMD in C/C++ or Fortran languages. (Fig.1) below shows the HMPP system structure.

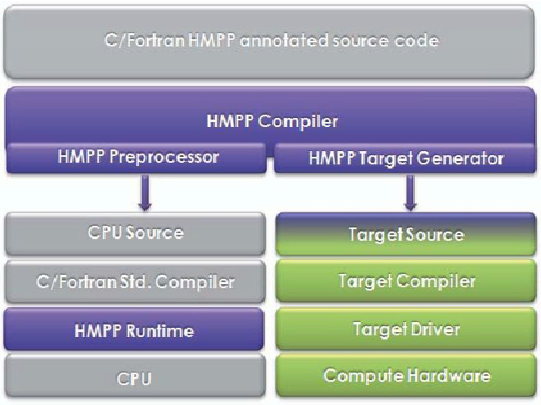

Fig.1 *HMPP system structure.*

5.CREATION OF PROBLEM-ORIENTED MEANS AND LIBRARIES FOR APPLICATION DEVELOPMENT

The enhancement of GPGPU basic means of programming and creation of adequate compilers facilitating application production stimulated the development of numerous libraries for the problem-oriented application development using the GPU hardware possibilities for computation optimization. The main advantage of the problem-oriented libraries is that you need not be acquainted with the GPU architecture features for realization of GPGPU-based applications. CUBLAS and CUFFT, being the earliest realizations of the mentioned libraries, have appeared as early as in the structures of NVIDIA and CUDA. The CUBLAS library was a variant of BLAS (Basic Linear Algebra Subprograms) library, and the CUFFT was a variant of FFT (Fast Fourier Transform) library. Both were optimized for speeding up the computations by NVIDIA CUDA tools. CUBLAS was used for execution of vector and matrix operations of single and double precision. The FFT-based computations were done with the help of CUFFT.

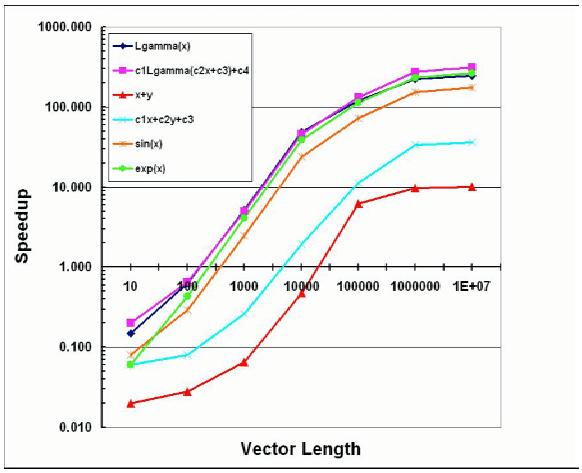

Fig.2 *Speeding up some vector operations as functions of vector dimension.*

CPULib from Tech-X Corporation, one more representative of this direction of development, was

intended for speeding up the operation of the applications created by the use of MATLAB and IDL tools. The plot below illustrates the possibility of speeding up some vector operations as functions of vector dimension (Fig.2): It is obvious that an appreciable acceleration is achieved only the vectors are composed of 1000 and more elements.

6. ENHANCEMENT OF ARCHITECTURE POSSIBILITIES AND BASIC GPGPU HARDWARE AND SOFTWARE MEANS

As an example of enhancement of GPGPU possibilities due to architecture improvement, new Fermi architecture may be mentioned and which has been realized in new GPUs from NVIDIA produced in 2010. The most typical features of new architecture are:

- support of 512 computing kernels;
- support of operations with floating point of double accuracy (precision);
- support of memory with error correction;
- NVIDIA Parallel Data Cache considerably speeding up computations and execution of other functions;
- NVIDIA Giga Thread *tm* technology (various branches of one and the same application may be executed with GPU simultaneously);
- support of Nexus - completely integrated computing media of application development by Microsoft Visual Studio.

It should be mentioned that in the nearest future there will appear one more competitor of NVIDIA Fermi. That is the Intel Larrabee architecture [7]. The base of Larrabee scalar block is the integer logic of Intel Pentium processor. To which was added the support of 64-bit commands of multithread performance, preliminary access and some other functions. The Larrabee vector block consists of 16 32-bit conveyors, each being able to perform both integer commands and single-precision real-valued commands. Double-accuracy real-valued commands may also be executed but with some loss of productivity. Almost all graphic traditional tasks (rasterization, interpolation, alpha-mixing, etc.) are performed not by specialized logic but according to the program by vector blocks. The only exception is the textural filtration executed by hardware.

7. CONCLUSIONS.

It should be mentioned that the use of graphic accelerators as fast calculators continues to expand to various branches of science and engineering, where computer capabilities of processing numerical data are used. In this connection we observe rapid development of programming aids of graphic accelerators as massively parallel processors. Besides, the development of architecture capabilities and basic hardware-software means of GPUs is going on, and the enhancement of their hardware-software resources is largely aimed at of using them as GPGPU means.

References

1. A. Zubinsky. NVIDIA CUDA: unification schedules and calculations. On May, 8, 2007. (http://itc.ua/node/27969).
2. D. David Ljubke. Graphic processors - not only for Graphics. (http://www.osp.ru/os/2007/02/4106864/).
3. D. Alexey V. Boreskoff. Bases CUDA http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html/
4. Michael Wolfe: Understanding the CUDA Data Parallel Threading Model http://www.pgroup.com/lit/articles/insider/v2n1a5.htm
5. Federico Dal Castello. Advanced System Technology, STMicroelectronics, Italy Douglas Miles. The Portland Group: Parallel Random Number Generation Using OpenMP, OpenCL and PGI Accelerator Directives. http://www.pgroup.com/lit/articles/ insider/v2n2a4.htm
6. Don Breazeal, Craig Toepfer: Tuning Application Performance Using Hardware Event Counters in the PGPROF Profiler http://www.pgroup.com/lit/articles/insider/v2n4a3.htm
7. Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan and Pat Hanrahan: Larrabee: A Many-Core x86 Architecture for Visual Computing http://softwarecommunity.intel.com/UserFiles/en-us/File/larrabee_manycore.pdf

СРЕДСТВА РАЗРАБОТКИ ПРОГРАММ, ИСПОЛЬЗУЮЩИХ ВОЗМОЖНОСТИ GPGPU

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

Приведено описание возможностей различных современных программных средств разработки для использования параллельных вычислительных возможностей графических процессоров (GPU). Даны примеры использования существующих программных средств для разработки приложений и реализации алгоритмов научно-технических расчётов, выполняемых средствами графических процессоров (GPGPU). Выделены некоторые классы математически интенсивных задач научно-технических расчётов, для которых возможно эффективное применение GPGPU. Выданы рекомендации по сокращению времени разработки программ научно-технических расчётов, использующих возможности GPGPU для ускорения обработки данных, за счёт применения различных специализированных систем программирования и проблемно-ориентированных библиотек подпрограмм. Показаны примеры сравнения показателей производительности при решении этих задач без применения GPU и с использованием возможностей GPGPU.

ЗАСОБИ РОЗРОБКИ ПРОГРАМ, ЩО ВИКОРИСТОВУЮТЬ МОЖЛИВОСТІ GPGPU

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

Приведено опис можливостей різних сучасних програмних засобів розробки для використання паралельних обчислювальних можливостей графічних процесорів (GPU). Наведені приклади використання існуючих програмних засобів для розробки додатків і реалізації алгоритмів науково-технічних розрахунків, виконуваних засобами графічних процесорів (GPGPU). Виділені деякі класи математично інтенсивних задач науково-технічних розрахунків, для яких можливе ефективне застосування GPGPU. Видані рекомендації по скороченню часу розробки програм науково-технічних розрахунків, що використовують можливості GPGPU для прискорення обробки даних, за рахунок застосування різних спеціалізованих систем програмування і проблемно-орієнтованих бібліотек підпрограм. Показані приклади порівняння показників продуктивності при вирішенні цих завдань без застосування GPU і з використанням можливостей GPGPU.