The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

Introduction to GPU programming

Lecture 1. General purpose computing on GPU

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this lecture is to give an introduction to general purpose computing on GPU (GPGPU) and heterogeneous computing, and explain reasons why computing on GPU became so popular, as well as present an overview of programming technologies that can be used for GPGPU.

ABSTRACT

This lecture considers historical development of GPU architecture that lead to GPGPU, overview of CPU and GPU architectures and GPU programming technologies. We consider a topic of work organization for heterogeneous computing.

BRIEF OVERVIEW

Heterogeneous (hybrid) computing is computing using different kinds of computational hardware. Most popular kinds of hardware are CPUs, Intel Xeon Phi coprocessors, GPUs, specialized processors (DSP and others), FPGAs. Most popular heterogeneous combination is currently CPU + GPU. Term GPGPU is used for general-purpose computing on GPU. Typical areas applications of GPGPU are physical simulation, visual effects, financial computing, computational biology and chemistry.

Modern GPUs are massively parallel processors with high performance and memory bandwidth. GPU architecture fits for many classes of computationally intensive problems. There are advanced programming languages and tools. However, not all problems can be parallelized on large amount of threads due to the following reasons: not all problems fit GPU architecture, code development and optimization is more complicated compared to traditional programming, many problems have computationally intensive subproblems that can be efficiently ported to GPU.

GPU architecture is aimed at computing that is data parallel: each operation is performed for many elements in parallel, big computational density. Compared to CPU is has much less cache and control logic much more computational elements. Memory latency is covered by using large amount of lightweight threads. CPU architecture is gradually becoming more parallel, GPU architecture is gradually becoming more sophisticated.

NVIDIA GPUs consist of streaming multiprocessors (MP), each contains several CUDA-cores and shared memory. In first CUDA devices CUDA-cores were called scalar processors (SP). CUDA-cores of one multiprocessor work as one or several SIMD units. There are extremely lightweight threads and hardware thread scheduler.

GPUs use stream computing model. Data stream is a sequence of uniform elements that can be processed independently. A function that processes one element is called kernel. Kernel body handles one element, kernel is called many times (maybe in parallel), once per each element. Examples of kernels: sum of two vectors of length N, kernel body adds a pair of numbers, kernel is called N times. GPU programming technologies are based on stream computing model.

Shading languages are historically first tool that could be used for GPGPU (although not designed for it). Shading languages are used to write pieces of code that is executed on GPU hardware, so-called shading processors. All programming is done in graphics terms. One of the most popular shading languages is GLSL (OpenGL Shading Language).

The next step in GPU programming technologies was introduction of metaprogramming tools: a program is written in a high-level language that is later automatically translated to shading languages. This model is much simpler to use compared to shading languages. A tradeoff is efficiency, the code is hard to optimize for a specific hardware. Now these tools are not supported, by they have given birth to other technologies: Sh became RapidMind that was consumed by Intel in 2009, improved version of BrookGPU was part of AMD Stream as Brook+.

GPU vendors introduced programming tools in 2007: NVIDIA CUDA in February 2007, AMD Stream Computing in November 2007. They have similar structure: GPU assembly and memory management on the low level, stream extensions of C language, high performance libraries, profiling and debugging tools on the high level. There are clear advantages of such tools: programming in extensions of C language, access to hardware features and low-level optimization. However, there are disadvantages: programming requires some knowledge of architecture, porting is not easy.

Open Computing Language (OpenCL) is an open standard for heterogeneous computing developed by Khronos Group in collaboration with vendors, first version in November 2008. It is supported by Apple, NVIDIA, AMD, Intel, IBM, and others. OpenCL supports a wide range of hardware due to programming in terms of abstract models of hardware. Applications are portable. Performance is not always high, but usually reasonable.

OpenACC is a standard for heterogeneous computing on multicore CPUs and GPUs. The concept is similar to OpenMP (and particularly #pragma offload for Xeon Phi). It provides limited access to low-level optimization and memory management. This tool is ideally suited for quick porting of large applications. Performance-critical parts might be later implemented and optimized using CUDA.

Heterogeneous systems are becoming more and more popular with rise of APUs (accelerated processing unit) that combine CPU and GPU cores and Intel Xeon Phi (MIC, many integrated

cores) coprocessors that contain about 60 cores. Most popular heterogeneous combination is currently CPU + GPU. Peak performance of modern GPUs is largely superior over CPUs. However, a gap is usually smaller on real applications. It makes sense to use both CPUs and GPUs simultaneously. In applications using CUDA or OpenCL host part is usually playing only utility role: initialization, running kernels, data exchanges. It does not place a great computational load on CPU and does not require all CPU cores. Thus, some (or all) CPU cores can be used for computing. Load balancing is required because of different nature of hardware being used. Besides performance difference, CPU cores use shared memory, while one needs data transfers between CPU and GPU or GPU and another GPU. Inefficient load balancing can cause data transfer overheads to ruin performance gains.

FOR STUDENTS

General information about NVIDIA GPU architecture and programming is presented in [1, 2].

REFERENCES

1. Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. – Addison-Wesley Professional, 2010. – 312 p.
2. NVIDIA CUDA C Programming Guide. [http://docs.nvidia.com/cuda/cuda-c-programming-guide/].

INDIVIDUAL WORK

1. Distinguish main architectural differences between CPUs and GPUs.
2. Enlist main GPU programming technologies.
3. Find an example that proves that load balancing is required for efficient heterogeneous computing.