


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”

Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Introduction to GPU programming

01 Lecture
General purpose computing on GPU

Bastrakov S.I.
Software department

Contents

- Why GPU?
- Overview of GPU architecture
- Overview of GPU programming technologies
- Heterogeneous computing on CPUs and GPUs

Why GPU?

Heterogeneous computing and GPGPU

- **Heterogeneous (hybrid) computing** is computing using different kinds of computational hardware.
- Most popular kinds of hardware:
  - CPUs;
  - Intel Xeon Phi coprocessors;
  - GPUs;
  - specialized processors (DSP and others);
  - FPGAs.
- Most popular heterogeneous combination is currently **CPU + GPU**.
- **GPGPU** is general-purpose computing on GPU.

Typical GPGPU applications

- Physical simulation
- Visual effects
- Financial computing
- Computational biology and chemistry

- Examples:

http://www.nvidia.com/object/cuda_showcase_html.html

http://developer.amd.com/zones/openclzone/pages/openclappexamples.aspx

Peak performance of CPUs and GPUs

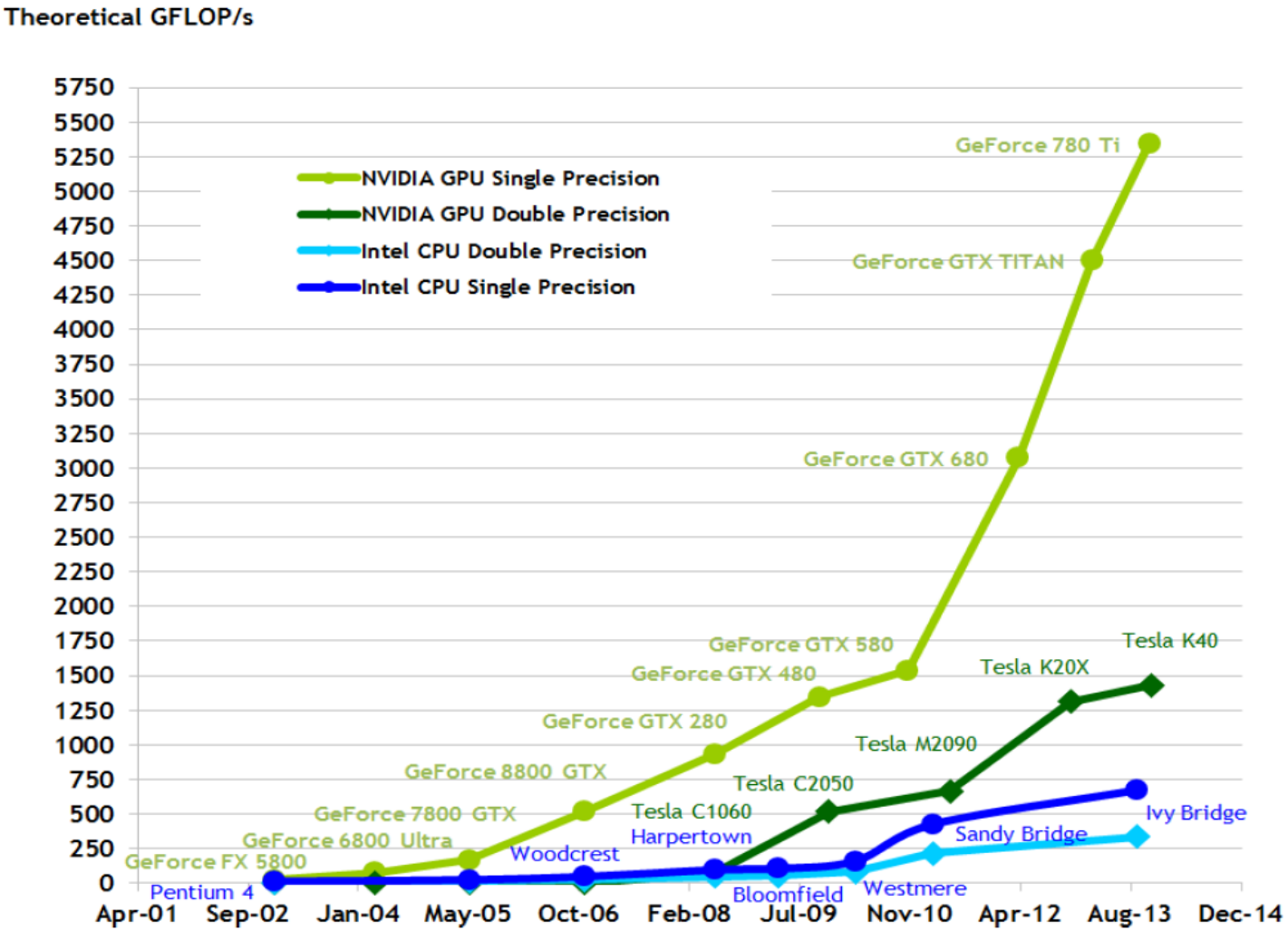


Image source: NVIDIA CUDA C Programming Guide v. 6.5

Memory bandwidth of CPUs and GPUs

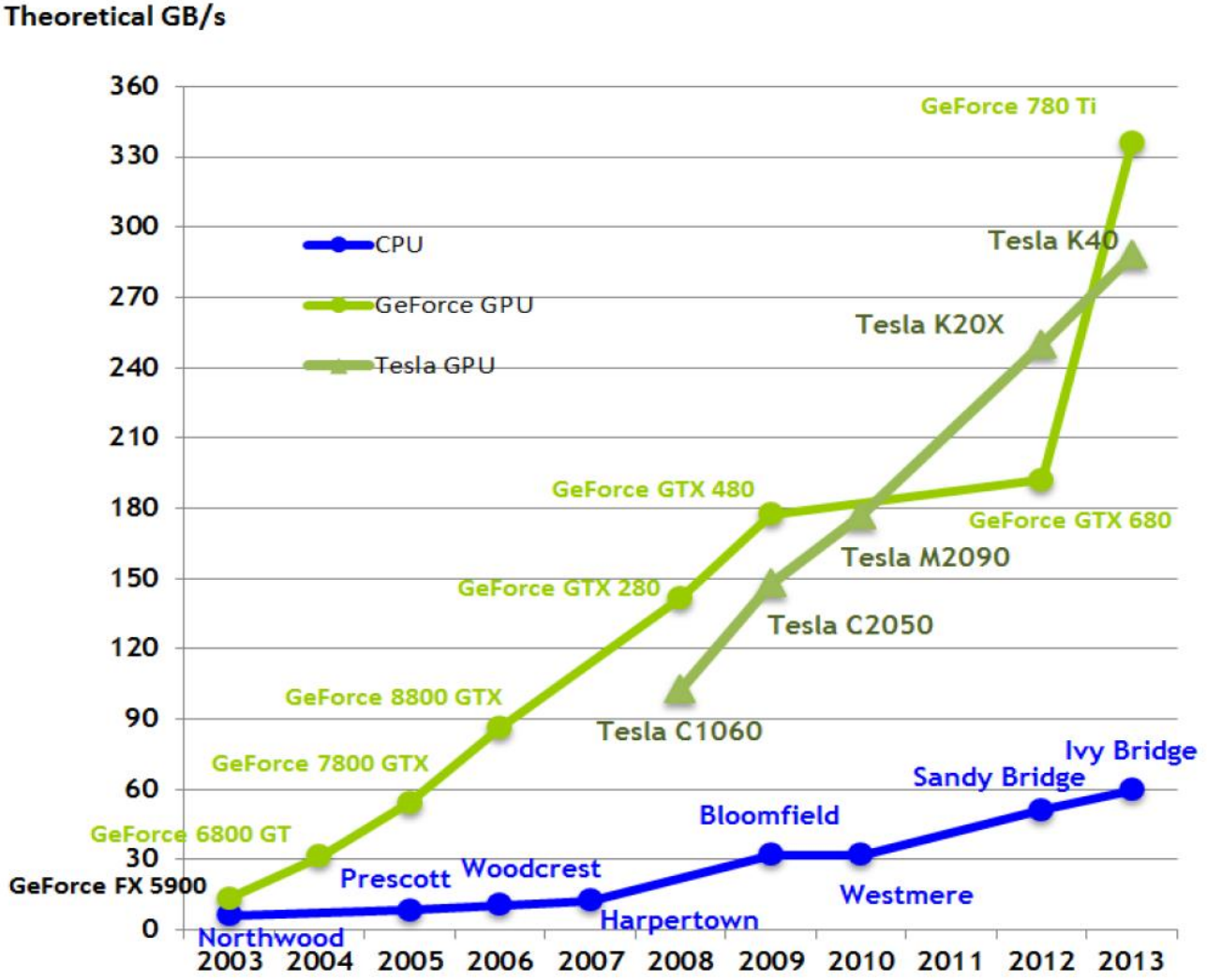


Image source: NVIDIA CUDA C Programming Guide v. 6.5

Modern GPUs

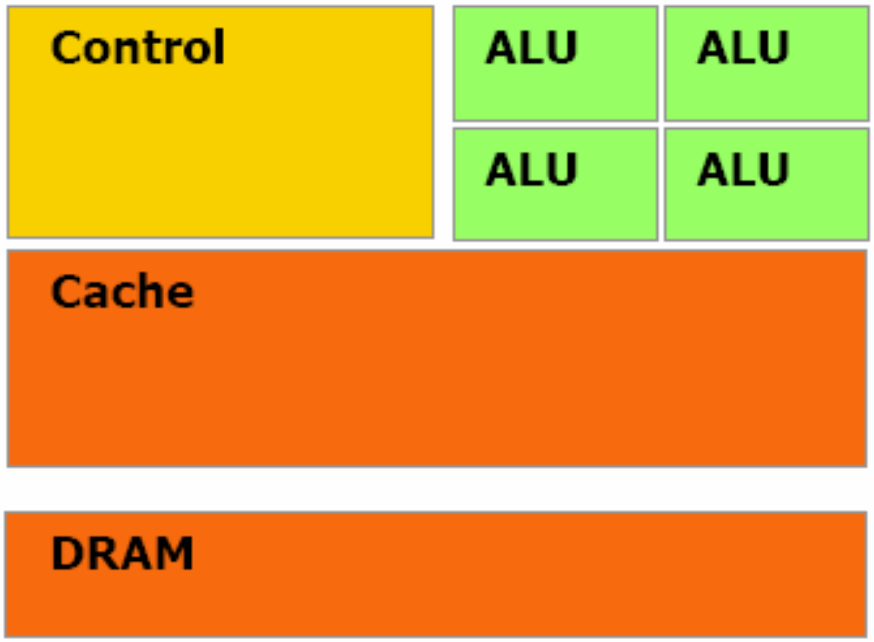
- Modern GPUs are **massively parallel processors**.
- High performance and memory bandwidth.
- Fits for many classes of computationally intensive problems.
- Advanced programming languages and tools.

Why not GPU?

- Not all problems can be parallelized on large amount of threads.
- 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.

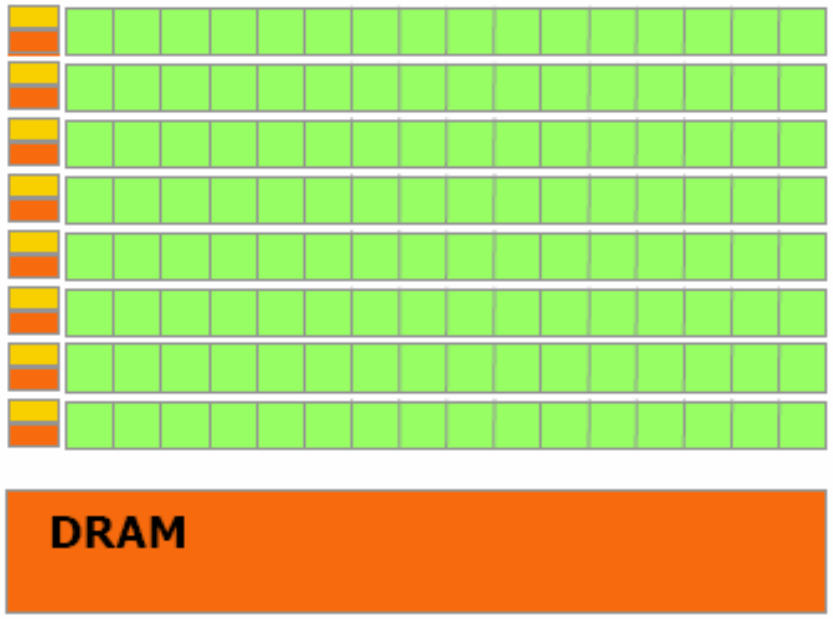
Overview of GPU architecture

Architecture of CPUs and GPUs



CPU

"cache-oriented"



GPU

"cache-miss oriented"

Image source: NVIDIA CUDA C Programming Guide v. 6.5

Architecture of CPUs and GPUs

- 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:
  - 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 GPU architecture

- Consists 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.
- Extremely lightweight threads, hardware thread scheduler.

Multiprocessor of NVIDIA Fermi architecture

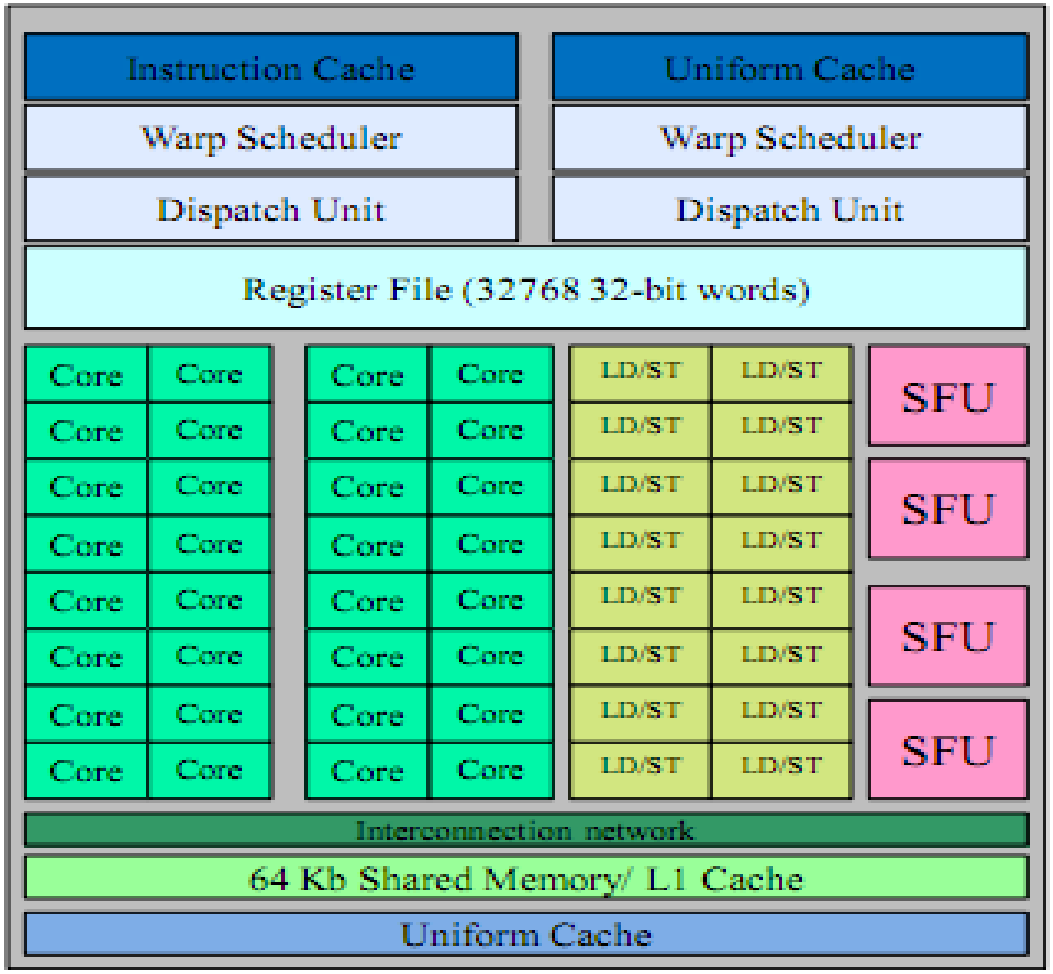


Image Source: А.В. Боресков, А.А. Харламов «Архитектура и программирование массивно-параллельных вычислительных систем»

Overview of GPU programming technologies

Stream computing model

- 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.
  - Example: sum of two vectors of length N, kernel body adds a pair of numbers, kernel is called N times.
- GPU programming technologies use stream computing model.

Shading languages

- 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)

Metaprogramming tools BrookGPU и Sh

- Metaprogramming: a program is written in a high-level language that is later automatically translated to shading languages.
- Much more simple 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+.

NVIDIA CUDA and AMD Stream

- NVIDIA CUDA: February 2007.
- AMD Stream Computing: November 2007.
- Have similar structure:
  - Low level: GPU assembly, memory management.
  - High level: stream extensions of C language, high performance libraries, profiling and debugging tools.
- Advantages:
  - Programming in extensions of C language.
  - Access to hardware features and low-level optimization.
- Disadvantages:
  - Programming requires some knowledge of architecture.
  - Porting is not easy.

OpenCL

- **OpenCL – Open Computing Language.**
- Open standard for heterogeneous computing developed by Khronos Group in collaboration with vendors.
- First version: November 2008.
- Supported by Apple, NVIDIA, AMD, Intel, IBM, and others.
- 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

- Standard for heterogeneous computing on multicore CPUs and GPUs.
- Concept is similar to OpenMP (and particularly #pragma offload for Xeon Phi)
- Limited access to low-level optimization and memory management
- Ideally suited for quick porting of large applications
- Performance-critical parts might be later implemented and optimized using CUDA

Heterogeneous computing on CPUs and GPUs

Motivation

- Heterogeneous systems are becoming more and more popular:
  - **APU** (*accelerated processing unit*) combine CPU and GPU cores.
  - **Intel Xeon Phi** (*MIC, many integrated cores*) coprocessors contain about 60 cores.

Motivation

- 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

Motivation

- 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

- Load balancing is required because of different nature of hardware being used:
  - Performance
  - Data transfers: CPU cores use shared memory, need data transfers between CPU and GPU or GPU and another GPU.
- Inefficient load balancing can cause data transfer overheads to ruin performance gains.

Summary

- GPUs are massively parallel processors that are well suited for data parallel applications
- GPU programming technologies mainly use stream computing model
- Most popular technologies are CUDA, OpenCL, OpenACC
- Heterogeneous computing becomes more popular
- Load balancing needs to be address thoroughly

Summary

- Appropriate algorithms that allow high degree of parallelism need to be chosen for implementation on GPU
- Typical scheme of problem decomposition is:
  - decomposition into large amount (thousands or more) of sub-problems that can be solved independently;
  - decomposition into small amount of sub-problems that can be solved independently, decomposition of each sub-problem into hundreds of smaller problems.
- High computational density is preferable.
- Usually CPUs are used for sequential parts of an application and GPUs are used for massively parallel part.

References

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

Authors

- Bastrakov S.I.,
  Assistant of the Software department of CMC faculty.
  bastrakov@vmk.unn.ru