Parallel Processing with Graphics Processors

Rui Wang & Hequn Zhang

March.6.2012

Introduction

- What is CUDA?

Compute Unified Device Architecture (**CUDA**) is NVIDIA's parallel computing architecture, and it is the computing engine in NVIDIA graphics processing units(GPUs)

Background

Computing is evolving from "central processing" on the CPU to "co-processing" on the CPU and GPU. To enable this new computing paradigm, NVIDIA invented the CUDA parallel computing architecture that is now shipping in GeForce, ION, Quadro, and Tesla products, representing a significant installed base for application developers.

Background

In the consumer market:

nearly every major consumer video application has been, or will soon be, accelerated by CUDA, including products from Adobe, Sony , Elemental Technologies and MotionDSP.

In the area of scientific research:

CUDA has been received in the area of scientific research. For example, CUDA now accelerates AMBER, a molecular dynamics simulation program used by more than 60,000 researchers in academia companies worldwide to accelerate new drug discovery.

In the financial market:

Numerix and CompatibL announced CUDA support for a new counterparty risk application and achieved an 18X speedup. Numerix is used by nearly 400 financial institutions.

Application of CUDA

Graphical application:

In the computer game industry, CUDA is used in game physics calculations, physical effects like debris, smokes, fire and fluids.

Non-graphical application:

CUDA has also been used to accelerate non-graphical applications in computational biology, cryptography and other fields by an order of magnitude or more. An example of this is the BOINC distributed computing client.

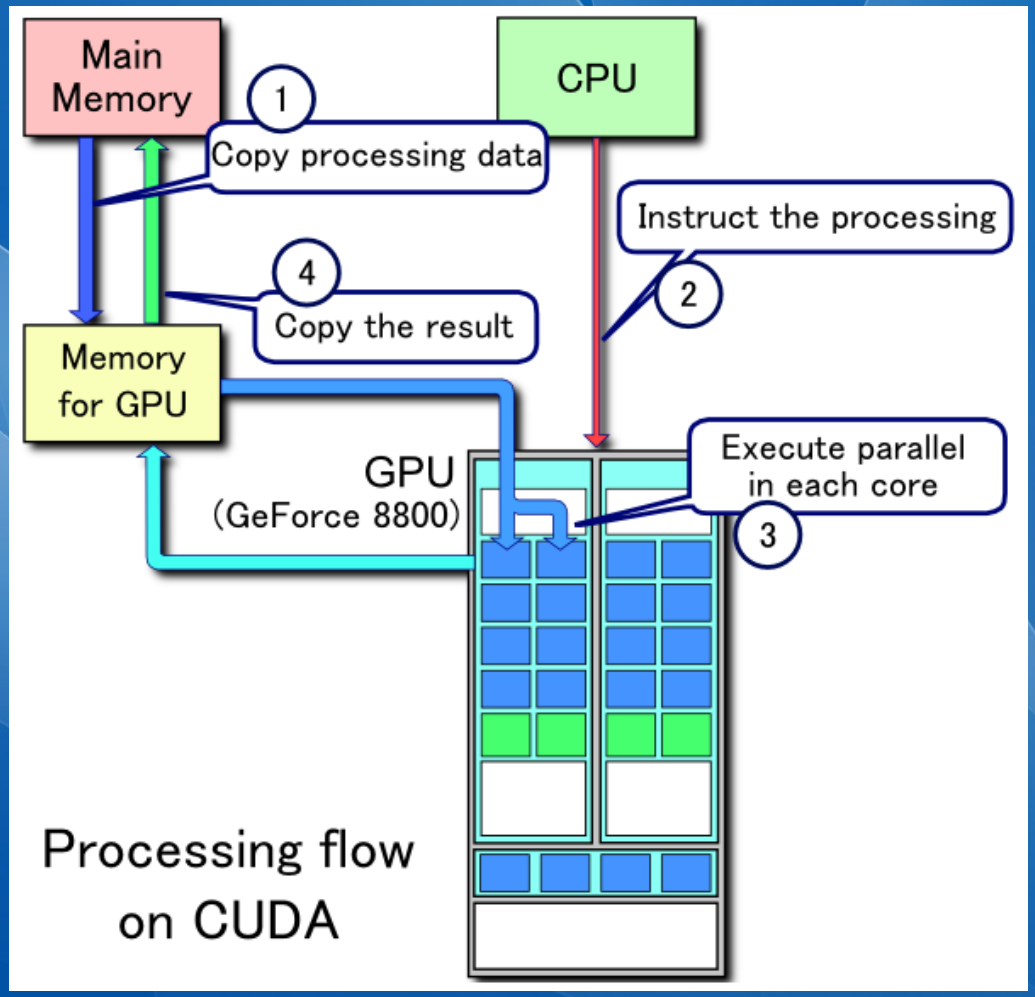
Application of CUDA

With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for CUDA, including image and video processing, computational biology and chemistry, fluid dynamics simulation, CT image reconstruction, and much more.

Advantages

- CUDA has several advantages over traditional general-purpose computation on GPUs using graphics APIs:
  - Scattered reads – code can read from arbitrary addresses in memory.
  - Shared memory – CUDA exposes a fast shared memory region that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
  - Faster downloads and readbacks to and from the GPU.
  - Full support for integer and bitwise operations, including integer texture lookups.

Processing flow on CUDA:

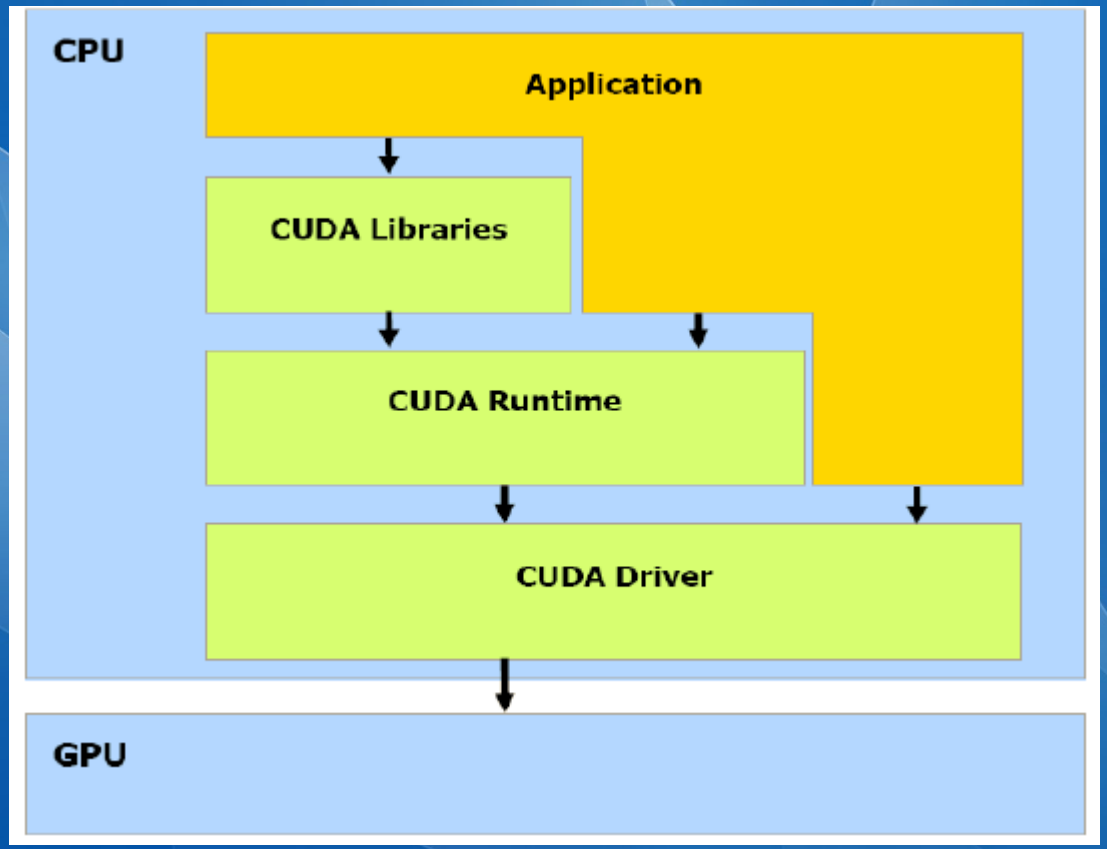


The CUDA Architecture

CUDA is a framework which works in a modern massively-parallel environment. CUDA-enabled graphics processors operate as co-processors within the host computer. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer.

To perform useful work, data must be transferred between the memory space of the host computer and CUDA device (s).

The CUDA Architecture



The CUDA software stack

The CUDA Architecture

CUDA provides general DRAM memory:

the ability to read and write data at any location in DRAM, just like on a CPU. CUDA has a parallel data cache with very fast general read and write access, that threads use to share data with each other.

When programmed with CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU (host). The host and the device maintain their own DRAM, the host memory and device memory, respectively.

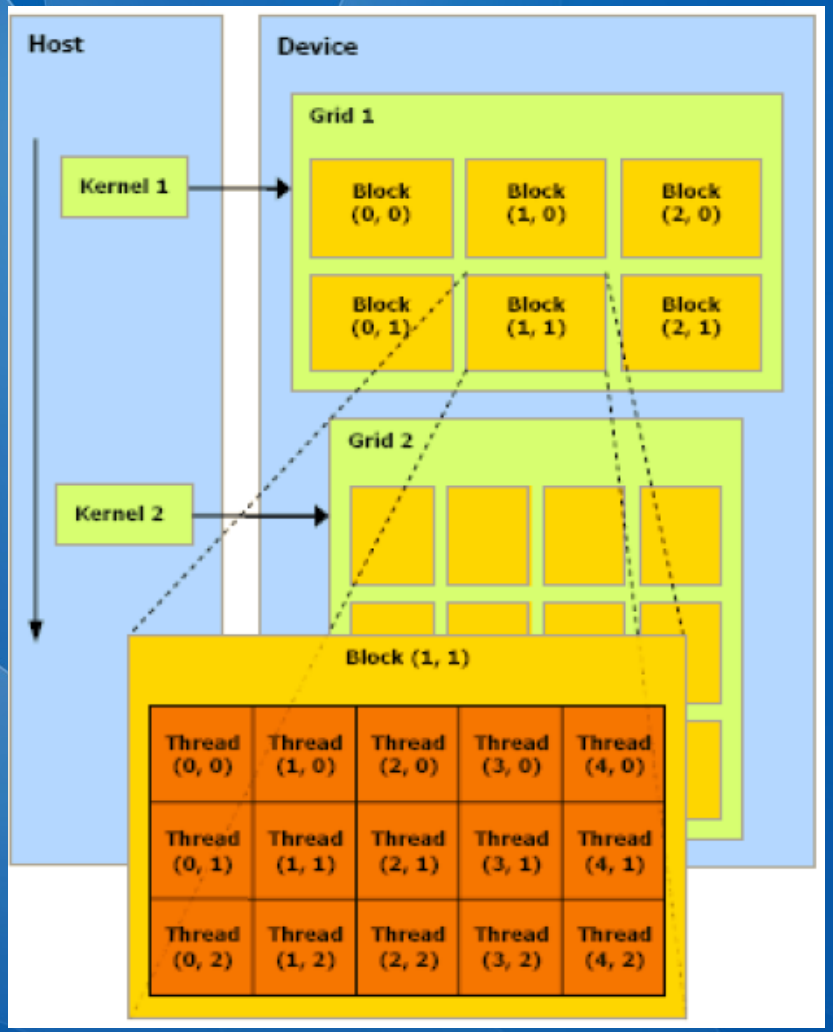
The CUDA Architecture

A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses.

Each thread is identified by its **thread ID**, which is the thread number within the block.

The blocks of same dimensionality and size that execute the same kernel can be batched together into **a grid of blocks**.

Each block is identified by its **block ID**, which is the block number within the grid.

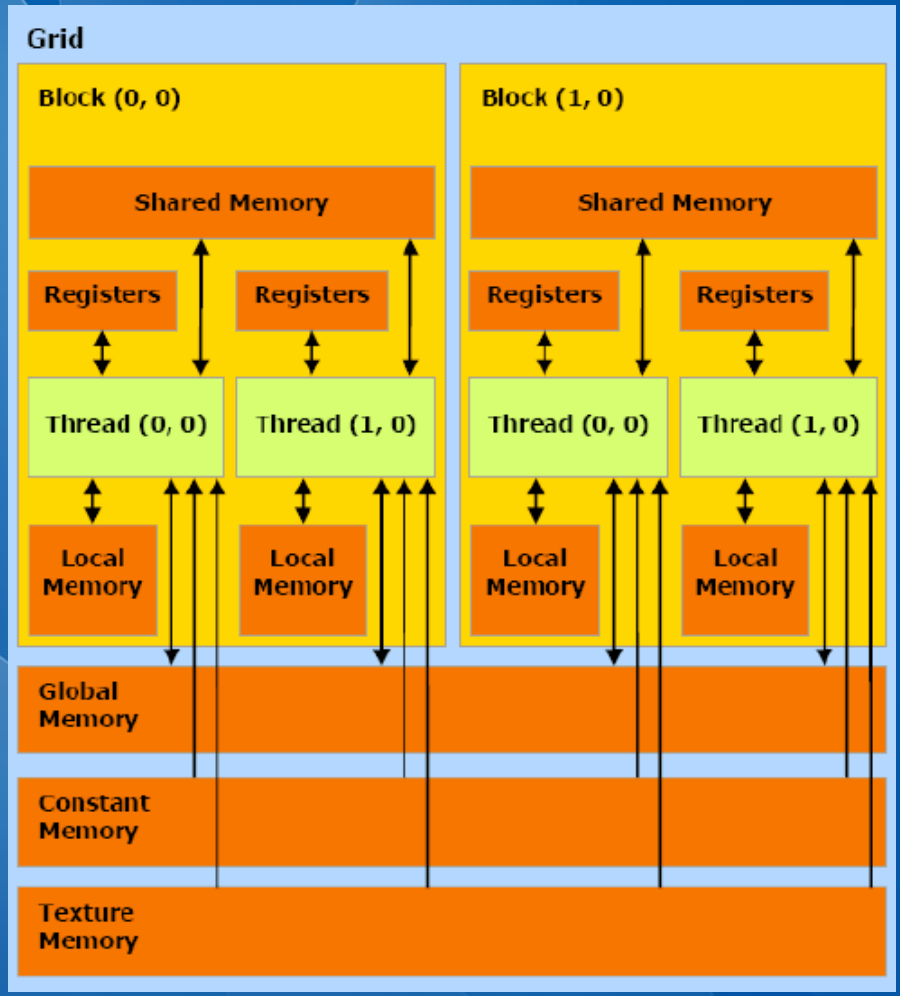


Thread batching

The CUDA Architecture

A thread that executes on the device has only access to the device's DRAM and on-chip memory through the following memory spaces:

- read-write per-thread registers
- read-write per-thread local memory
- read-write per-block shared memory
- read-write per-grid global memory
- read-only per-grid constant memory
- read-only per-grid texture memory



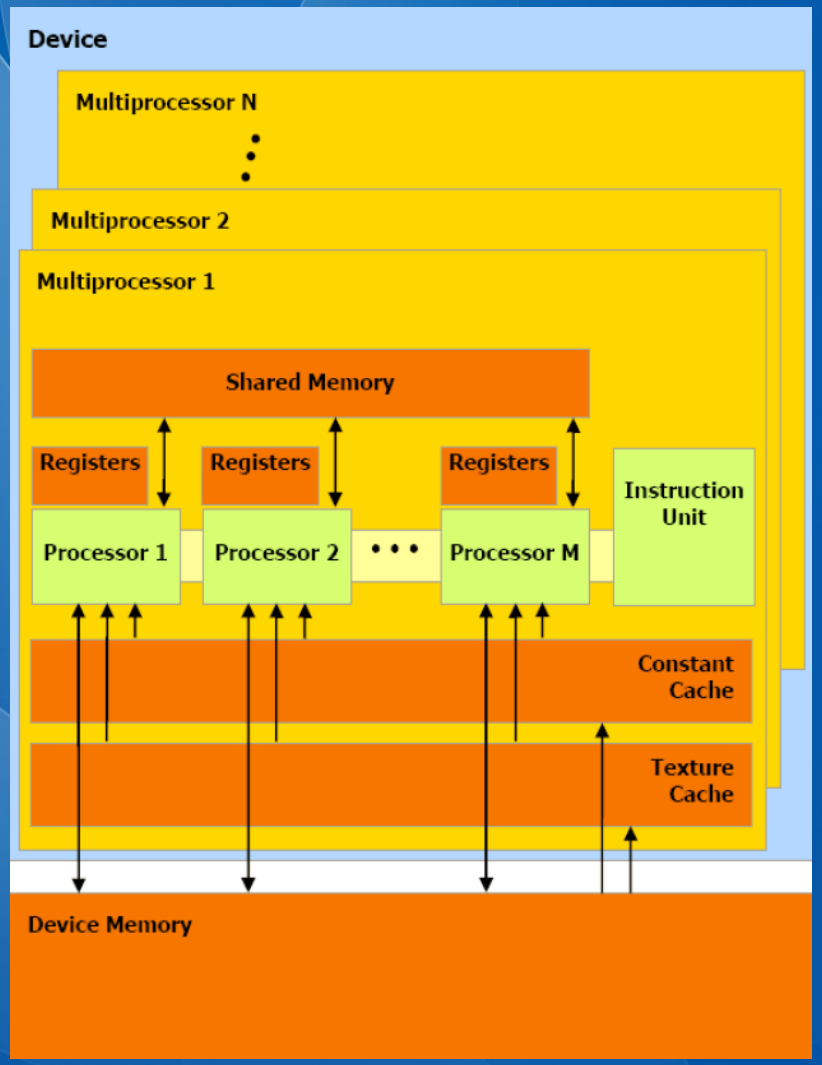
The Memory model

The CUDA Architecture

The device is implemented as a set of multiprocessors .

Each multiprocessor has on-chip memory of the four following types:

- one set of local 32-bit registers per processor.
- a parallel data cache (shared memory) that is shared by all the processors and implements the shared memory space.
- a read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory.
- a read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.



The hardware model

The CUDA Architecture

- **The local and global memory spaces** are implemented as read-write regions of device memory and are not cached. Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering.

The CUDA Architecture

- **A grid of thread blocks** is executed on the device by executing one or more blocks on each multiprocessor using time slicing:
  Each block is split into SIMD(Single Instruction, Multiple Data architecture) groups of threads called **warps**; each of these warps contains the same number of threads, called the **warp size**,
  and is executed by the multiprocessor in a SIMD fashion; **a thread scheduler** periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources.

CUDA and OpenCL

- CUDA and OpenCL offer two interfaces for programming GPUs, both presenting similar features but through different programming interfaces.

CUDA and OpenCL

* What is OpenCL?

OpenCL(Open Computing Language) is an open standard for parallel programming using Central Processing Units (CPUs), GPUs, Digital Signal Processors (DSPs), and other types of processors.

CUDA and OpenCL

CUDA	OpenCL
CUDA is a proprietary API and set of language extensions that works only on NVIDIA's GPUs.	OpenCL(Open Computing Language) is an open standard for parallel programming using Central Processing Units (CPUs), GPUs, Digital Signal Processors (DSPs), and other types of processors.
CUDA is developed to develop the hardware on which it executes, so one may expect it to better match the computing characteristics of the GPU, offering more access to features and better performance.	OpenCL promises a portable language for GPU programming, capable of targeting very dissimilar parallel processing devices. An OpenCL kernel can be compiled at runtime, which would add to an OpenCL's running time. This just-in-time compile may allow the compiler to generate code that makes better use of the target GPU.

CUDA and OpenCL

- They both draw the same conclusions:

if the OpenCL implementation is correctly tweaked to suit the target architecture, it performs no worse than CUDA.

Because the key feature of OpenCL is portability, the programmer is not able to directly use GPU-specific technologies, unlike CUDA.

CUDA and OpenCL

- CUDA is more acutely aware of the platform upon which it will be executing, because it is limited to NVIDIA hardware, and therefore, it provides more mature compiler optimisations and execution techniques

- Furthermore, the CUDA compiler displayed more mature compilation techniques. Therefore, the developer is required to add in the optimisations manually to the kernel code. This is indicative of the maturity of the CUDA toolkit versus the newer OpenCL toolkits. It is likely in the future that this gap will be closed as the toolchains mature.

The end

Thanks a lot!