CUDA™ to Unleash Computational Power of GPU

DATE 2011

Iole Moccagatta and Elif Albuz
NVIDIA Corporation
(e-mail: iomoccagatta@nvidia.com, ealbuz@nvidia.com)
Welcome!

Goal: an Introduction to High Performance Parallel Computing with CUDA

Outline:
- The Evolution of GPU Computing
- The CUDA Programming Model
- CUDA Tools and Libraries
- Examples of GPU Computing Applications
- Pointers and Links to Know More
The Evolution of GPU Computing
Fixed-Function Graphics Pipelines

["The Cg Tutorial", R. Fernando and M.J. Kilgard, 2003]

From Fixed-Function To Programmable

- Graphics pipeline stages become programmable processors
  - Support more features that the built-in fixed ones
  - Support developers’ growing sophistication
  - Best candidates are stages whose functions vary with rendering

- Data independence drives towards parallelism
  - Some stage do a lot of operations on completely independent data
  - Data independence is key difference between GPU and CPU

- CPU cache memory vs. GPU datapath and fixed-function logic,
  GPU bandwidth over latency emphasis
  - Rendering algorithms access memory coherently
  - Shaders are compute limited
From Fixed-Function To Programmable (2)

[Diagram showing the process from 3D application or game to framebuffer, highlighting the changes from fixed-function to programmable processes.]

Unified Graphics and Computing Processors

- From programmable to unified pipeline and shader design
  - DirectX 10 introduces a unified instruction set across all shaders
  - A unified GPU shader architecture follows → re-circulating path
    - Deliver the throughput required by new and sophisticated shading algorithms
    - Enable dynamic partition and load balancing
    - Focus engineering effort on one processor vs. three (vertex, pixel, and geometry)
    - Improve cost (performance/mm²) and power (performance/watt) efficiency
  - Reduced number of pipeline stages and change in the flow

[“NVIDIA GeForce 8800 GPU Architecture Overview”, NVIDIA Technical Brief, 2006]
Unified Graphics and Computing Processors (2)

- **Unified programmable processor array**
  - Recirculation of the logical pipeline into threads of the processor array
  - Dynamic partitioning of the array to vertex, geometry, and pixels processing

Stream/Thread Processor
- generalized single-precision FPU
- can operate on vertices, pixels, or any data
- 1024 registers (32-bit)
- multithreaded

Shader Processor = 8 SP
- 4 Texture Module Units (TMU)
  - 1 Texture Address (TA) Unit
  - 2 Texture Filtering (TF) Unit
- 1 Shared L1 cache

ROP and shared resources
- L2 cache
- registers
- I/F to graphic memory (FB)

[“NVIDIA GeForce 8800 GPU Architecture Overview”, NVIDIA Technical Brief, 2006]
GPUs as Parallel Computers

- GPU is optimized for compute-intensive data-parallel processing
  - Same computation executed on many data elements in parallel
  - Many calculations per memory access
- Data processing rather than data caching and flow control

Multicore CPUs vs. many-cores GPUs
The CUDA Programming Model
**Drawbacks of Legacy GPGPU Model: Software Limitations**

- The GPU can only be programmed through a graphics API
- High learning curve
- Graphics API overhead
- Addressing limitations, less programming flexibility
**GPU as a Highly Multithreaded Coprocessor**

- **The GPU is a compute device**
  - A coprocessor to the CPU or host to run data and compute intensive portion of the application
  - Has its own DRAM = device memory
  - Runs many threads in parallel

- **Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads**

- **GPU vs. CPU threads**
  - GPU threads are extremely lightweight
    - very little creation overhead
  - GPU supports 1000s of threads
    - GPU fully efficient when 1000s of threads are running
    - multi-core CPUs can support/needs only a few
CUDA: Compute Unified Device Architecture

- A new hardware and software parallel computing architecture for issuing and managing computations on the GPU
  - No need to go through a graphics API
  - Dedicated features for general computing
    - shared memory, double precision FP, etc.
  - Automatic HW thread manager
    - threads scheduling, thread resources management, etc.

- GPGPU made easy and lightweight
  - GPU accessible to developers through standard programming language
    - CUDA API is an extension to C
  - HW designed for lightweight runtime and driver
  - Access to on-chip shared memory for efficient inter-thread data sharing
CUDA Architectural View: A Set of SIMD Multiprocessors

- The GPU is a set of multiprocessor

- Each multiprocessor is a set of 32-bit processors with a SIMD architecture
  - A multiprocessor executes the same instruction on a group of threads called a warp
  - The number of threads in a warp = warp size
CUDA Architectural View: Memory Model

- **Global, constant, and texture**
  - Set of 32-bit registers per processor
  - On-chip shared memory
    - where the shared memory space resides
  - A read-only constant cache
    - speed up access to the constant memory space
  - A read-only texture cache
    - speed up access to the texture memory space

- **CPU can read/write global, constant, and texture memory**
Best Programming Practices

- **Global memory is un-cached**
  - Much slower access than shared memory

- **Take advantage of fast shared memory**
  - Partition the data set into data subsets that fit into shared memory
  - Handle each data subset with one thread block
    - load the subset from global to shared memory

- **Operate on the subset in shared memory**
  - Each thread can efficiently multi-pass over any data element

- **Copy results from shared to global**
CUDA Programming Model: SPMD + SIMD

- **Grid**: a group of thread blocks that execute a single CUDA program logically in parallel

- **Thread Block**: a group of threads that are executed together and can share memory on a single multiprocessor
  - Synchronized execution for hazard-free shared memory accesses
  - Two threads from two different blocks cannot cooperate

- **Warp**: a group of threads executed physically in parallel (SIMD)

- **Thread**: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
  - GPU vs. CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller
CUDA Programming Model: SPMD + SIMD

- Threads and blocks have IDs (mention this when presenting previous slide)
  - So each thread can decide what data to work on
  - Blocks can be modeled as 1D, 2D or 3D
  - Threads can be modeled as 1D, 2D, or 3D

- Different identification layouts simplify memory addressing when processing multidimensional data (remove, do not mention)
  - Image processing
  - Solving PDEs on volumes
CUDA Execution Model

- Each thread block of a grid is split into warps that get executed by one multiprocessor
  - The device processes only one grid at a time
- Each thread block is executed by only one multiprocessor
  - Shared memory space resides in the on-chip shared memory
  - Registers are allocated among the threads
    - a kernel that requires too many registers will fail to launch
- A multiprocessor can execute several blocks concurrently
  - Shared memory and registers are allocated among the threads of all concurrent blocks
  - User can increase number of blocks that can run concurrently
    - decrease shared memory usage (per block)
    - decrease register usage (per thread)
CUDA Application Programming Interface

- The CUDA API is an extension to the C programming language.

- Language extensions
  - To target portions of the code for execution on the device.

- A runtime library split into
  - A common component providing built-in vector types and a subset of the C runtime library supported in both host and device codes.
  - A host component to control and access one or more devices from the host.
  - A device component providing device-specific functions.
**CUDA Runtime Component**

- **Device management (including multi-device systems)**
  - cudaGetDeviceCount()
  - cudaGetDeviceProperties()
  - cudaChooseDevice()

- **Memory management**
  - cudaMalloc(), cudaMallocPitch(), cudaFree(), cudaMallocArray(), cudaFreeArray()
  - cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyTo/fromArray(), cudaMemcpyTo/fromSymbol()

- **Texture management**

- **Interoperability with OpenGL and Direct3D9**

- **Error handling**
  - cudaGetLastError()

- **Device**
  - Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. $\_\_\sin(x)$)
    - $\_\_\text{pow}$
    - $\_\_\log, \_\_\log2, \_\_\log10$
    - $\_\_\exp$
    - $\_\_\sin, \_\_\cos, \_\_\tan$

- **Texture functions**
  - float4 value = tex2D(myTexRef, i);

- **Synchronization functions**
  - void __syncthreads();
    - Synchronizes all threads in a block
CUDA Runtime Component

- Device management (including multi-device systems)
  - cudaGetDeviceCount()
  - cudaGetDeviceProperties()
  - cudaChooseDevice()

- Memory management
  - cudaMalloc()

- Texture management

- Interoperability with OpenGL and Direct3D9

- Error handling
  - cudaGetLastError()

- Some mathematical functions (e.g. less accurate, but faster)
  - __sin(x)
  - __cos(x)
  - __tan(x)
  - __pow
  - __log, __log2, __log10

- Built-in vector types
- A subset of the C runtime library supported in both host and device codes

- Texture functions
  - float4 value = tex2D(myTexRef, i);

- Synchronization functions
  - void __syncthreads();
    - Synchronizes all threads in a block

A host thread can execute device code on only one device (Multiple host threads required to run on multiple devices)
Example of GPU Computing Programming
CUDA Programming: Vector Addition Kernel

// Pair-wise addition of vector elements, one thread per addition
__global__ void vectorAdd(float* inA, float* inB, float* outC)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    outC[idx] = inA[idx] + inB[idx];
}

- `__global__` defines a kernel function, must return `void`, called from Host
- `__device__` defines a kernel function called from device
- For functions executed on the device:
  - no recursion, no static variable declarations inside the function, no variable number of arguments
- Built-in variables
  - `dim3 gridDim; //Dimensions of the grid in blocks (gridDim.z unused)`
  - `dim3 blockDim; //Dimensions of the block in threads`
  - `dim3 blockIdx; //Block index within the grid`
  - `dim3 threadIdx; //Thread index within the block`
Vector Addition Host Code

float* A = (float*) malloc(N * sizeof(float));
float* B = (float*) malloc(N * sizeof(float));
// ... initialize h_A and h_B

float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float)); // allocate device memory
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice ); // copy host mem. to device
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice );

vectorAdd<<< N/256, 256 >>>( d_A, d_B, d_C); // execute on N/256 blocks of 256 threads each

- Explicit GPU memory allocation/free
  - cudaMalloc(), cudaFree()
- Memory copy from host to device etc
  - cudaMemcpy(), cudaMemcpy2D(),...
CUDA Tools and Libraries
# GPU Computing Applications

## Libraries and Middleware

<table>
<thead>
<tr>
<th>cuFFT</th>
<th>cuBLAS</th>
<th>CULA LAPACK</th>
<th>NPP &amp; cuDPP</th>
<th>Video</th>
<th>PhysX</th>
<th>OptiX</th>
<th>mental ray</th>
<th>Reality Server</th>
<th>3D Web Services</th>
</tr>
</thead>
</table>

## Languages

- C
- C++
- OpenCL
- Direct Compute
- Fortran
- Java and Python

## NVIDIA GPU

NVIDIA GPU with the **CUDA** Parallel Computing Architecture

### Fermi Architecture
(Compute capabilities 2.x)

<table>
<thead>
<tr>
<th>GPU Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce 400 Series</td>
</tr>
<tr>
<td>Quadro Fermi Series</td>
</tr>
<tr>
<td>Tesla 20 Series</td>
</tr>
</tbody>
</table>

### Tesla Architecture
(Compute capabilities 1.x)

<table>
<thead>
<tr>
<th>GPU Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce 200 Series</td>
</tr>
<tr>
<td>GeForce 9 Series</td>
</tr>
<tr>
<td>GeForce 8 Series</td>
</tr>
<tr>
<td>Quadro FX Series</td>
</tr>
<tr>
<td>Quadro Plex Series</td>
</tr>
<tr>
<td>Quadro NVS Series</td>
</tr>
<tr>
<td>Tesla 10 Series</td>
</tr>
</tbody>
</table>

## Applications

- **Entertainment**
- **Professional Graphics**
- **High Performance Computing**

OpenCL is trademark of Apple Inc. used under license to the Khronos Group Inc.
NVIDIA CUDA Libraries

- Accelerate building blocks required by algorithms widely used in GPU computing
  - Most commonly used routines are heavily optimized
  - Optimized for all CUDA-capable hardware
  - Incorporate best practices from the field
  - CUFFT - Fourier Transform Libraries
  - CUBLAS - Basic Linear Algebra Subprograms
  - CUSPARSE and Cusp (3rd party) - Sparse matrix operations
  - NPP - NVIDIA Performance Primitives (image, graph, geometry, etc.)
  - CURAND - Random number generation library
  - Thrust - CUDA library of parallel algorithms
  - Math.h - C99 compliant math library containing IEEE-754 accurate sqrt etc.
NVIDIA Developer Ecosystem

Debuggers & Profilers
- cuda-gdb
- NV Visual Profiler
- Parallel Nsight
- Visual Studio
- Allinea TotalView

GPU Compilers
- C
- C++
- Fortran
- OpenCL
- DirectCompute
- Java
- Python

Parallelizing Compilers
- PGI Accelerator
- CAPS
- HMPP
- mCUDA
- OpenMP

Libraries
- BLAS
- FFT
- LAPACK
- NPP
- Video Imaging
- GPU Lib

Numerical Packages
- MATLAB
- Mathematica
- NI LabView
- pyCUDA

GPGPU Consultants & Training
- ANEO
- GPU Tech

OEM Solution Providers
- acceleware
- GASS
- STONE RIDGE TECHNOLOGY
- EM Photonics
- SCALABLE GRAPHICS
- HP
- SGI
- FUTUS
- Bull
- ATUO
- lenovo
- NEC
- DELL
- HP
- IBM
- SUPERMICRO
- CRAY
Examples of GPU Computing Applications
### Increasing Number of CUDA Applications

<table>
<thead>
<tr>
<th>Tools &amp; Libraries</th>
<th>Already Available</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tools</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA LAPACK Library</td>
<td>CUDA C/C++, PGI Fortran</td>
<td>Nsight Visual Studio IDE</td>
<td>Allinea Debugger</td>
<td>TotalView Debugger</td>
</tr>
<tr>
<td><strong>Libraries</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mathematica</td>
<td>CAPS HMPP Enhancements</td>
<td>Mathworks MATLAB</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Oil &amp; Gas</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMBER, GROMACS, GROMOS, HOOMD, LAMMPS, NAMD, VMD</td>
<td>BigDFT, ABINIT, TeraChem</td>
<td>Quantum Chem Code 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bio-Chemistry</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMBER, GROMACS, GROMOS, HOOMD, LAMMPS, NAMD, VMD</td>
<td>BigDFT, ABINIT, TeraChem</td>
<td>Quantum Chem Code 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bio-Informatics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex Protein Docking</td>
<td>CUDA-BLASTP, GPU-HMER, MUMmer, MEME, CUDA-EC</td>
<td>Protein Docking</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Video &amp; Rendering</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fractions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fraunhofer JPEG2K</td>
<td>OptiX Ray Tracing Engine</td>
<td>mental ray with iray</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Finance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAG: RNGs</td>
<td>Numerix: Counterparty Risk</td>
<td>Scicomp SciFinance</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CAE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoDesk Moldflow</td>
<td>OpenCurrent: CFD/PDE Library</td>
<td>Moldex3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EDA</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Electro-magnetics: Agilent, CST, Remcom, SPEAG</td>
<td>Agilent ADS, Spice Simulator</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 100,000 active GPU computing developers

- Released Product
- Announced Product
- Unannounced Product
Examples of CUDA Applications

- Imaging, Audio, Video
  - Badaboom Media Converter™
    - 1080p MPEG-X, VC-1, RAW 4:2:0 → H.264 BP and MB
    - AC-3, MP2, PCM, AAC → AAC-LC (2 channel)
    - 20x speedup (ex: 2h movie → 20min)
      speedup linear with number of SP
  - Adobe Premiere Pro CS5™
  - Image Processing
    - 200x speedup edge detection, 40x speedup histogram

- Computer Vision
  - GpuCV 1.0.0 rev 600
    - GPU-accelerated OpenCV API
    - Benefit image processing programmers and operator developers
    - Up to 100x speedup

Badaboom Media Converter is trademark of Elemental Technologies
Adobe Premiere Pro CS5 is trademark of Adobe
GpuCV is distributed under the CeCILL-B license
Examples of CUDA Applications (2)

**Graphics**

- **Furry Ball™ GPU rendered for Maya**
  - Dynamic Fur and Hairs, Bump mapping, Reflection, Ambient occlusion, Transparency, etc.
  - 30 to 300x speedup over CPU render
  - HD commercial - render time 30 sec per frame

**Digital Content Generation**

- **AgiSoft PhotoScan™ 3D Modeling**
  - create 3D content from still images
  - image alignment and 3D model is fully automated
  - 20x speedup
  - sample 3D models: [sample01-2.pdf](http://furryball.aaa-studio.cz) [sample02-2.pdf](http://www.agisoft.ru/products/photoscan)
Examples of CUDA Applications (3)

- **Electronic Design Automation**
  - Statistical Static Timing Analysis (SSTA)
    - Monte Carlo based statistical static timing analysis
    - Texas A&M University, USA
    - 260x speedup
  - SCGPSim: SystemC Simulator on GPU
    - FERMAL Lab, Virginia Tech, USA
    - 100x speedup

- **From MATLAB to Mosquito Death Ray**
  - MATLAB GPU Computing™
    - cuFFT and cuBLAS to GPU-accelerate MATLAB operations
    - Integrate CUDA kernels into MATLAB
    - up to 10x speedup
  - Kill malaria-spreading mosquitoes mid-flight
    - mosquito tracking and analysis computed on GPU
Hints for GPU Programming: Think Data Parallel

- GPU is a data-parallel processor
  - Thousands of parallel threads
  - Thousands of data elements to process
  - All data processed by the same program
    - SPMD computation model
  - Contrast with task parallelism and ILP

- Best results when you “Think Data Parallel”
  - Design your algorithm for data-parallelism
  - Use data-parallel algorithmic primitives as building blocks
  - Understand parallel algorithmic complexity and efficiency
Conclusion

- The GPU’s **highly multithreaded architecture** is very suitable for **solving data-parallel problems** and does it with increasingly higher performance.

- CUDA-capable GPUs go one step further by streamlining the programming model and introducing **On-Chip Shared Memory** for efficient inter-thread communication.

- By being a simple **extension to the C programming language**, CUDA is the tool that will allow you to leverage that flexibility and power with minimum learning curve.

- **CUDA libraries** provide benefit of massively parallel optimized code ready to launch on your applications.
Pointers and Links to Know More
CUDA on the Web

  - Includes drivers, compilers, libraries, debuggers, profilers, reference manuals, programming guides, code samples
- And more on CUDA Zone: http://www.nvidia.com/object/cuda_home_new.html
Questions?