GPU Computing with CUDA

Part 2: CUDA Introduction

Dortmund, June 4, 2009
SFB 708, AK "Modellierung und Simulation"

Dominik Göddeke
Angewandte Mathematik und Numerik
TU Dortmund

dominik.goeddeke@math.tu-dortmund.de // http://www.mathematik.tu-dortmund.de/~goeddeke
Acknowledgements

• Slides based on previous courses by
  • Mark Harris, Simon Green, Gregory Ruetsch (NVIDIA)
  • Robert Strzodka (MPI Informatik)
  • Dominik Göddeke (TU Dortmund)

• ARCS 2008 GPGPU and CUDA Tutorials
  http://www.mathematik.tu-dortmund.de/~goeddeke/arcs2008/
• University of New South Wales Workshop on GPU Computing with CUDA
  http://www.cse.unsw.edu.au/~pls/cuda-workshop09/
CUDA on one slide

- **Parallel computing architecture and programming model**
  - Unified hardware and software specification for parallel computing

- **Massively hardware multithreaded**
  - GPU = dedicated many-core co-processor

- **General purpose programming model**
  - User launches batches of threads on the GPU (application controlled SIMD program structure)
  - Fully general load/store memory model (CRCW)
  - Simple extension to standard C
  - Mature(d) software stack (high-level and low-level access)

- **Not another graphics API**
  - Though graphics API interoperability possible
Outline

• CUDA parallel hardware architecture

• CUDA programming model

• Code walkthrough

• Libraries

• Tool chain and OpenCL

• Tesla compute hardware
Hardware overview (GTX 280)

Streaming Processor (SP)
Thread Processor (TP)

- Multi-banked Register File
- FP / Integer
- Special Ops

Streaming Multiprocessor (SM)

- Special Function Unit (SFU)
- Double Precision
- TP Array Shared Memory

Main Memory

- Thread Manager

30 SMs per chip
Thread processor

- Floating point / integer unit
- Single precision, „almost“ IEEE-754
- Move, compare, logic, branch
- Local register file
- Essentially rather an ALU than a processor core
Multiprocessor

- Eight thread processors
- SFU for transcendental
- One double precision unit (fully IEEE-754 compliant)
- 16kB shared memory
- Shared instruction unit and instruction cache
- Maintains up to 768 threads simultaneously, hardware scheduler with zero-overhead context switching
- GTX 280: 30 multiprocessors
## Double precision support

<table>
<thead>
<tr>
<th>Feature</th>
<th>NVIDIA Tesla T10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>IEEE 754</td>
</tr>
<tr>
<td>Rounding modes for FADD and FMUL</td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
</tr>
<tr>
<td>Denormal handling</td>
<td>Full speed</td>
</tr>
<tr>
<td>NaN support</td>
<td>Yes</td>
</tr>
<tr>
<td>Overflow and Infinity support</td>
<td>Yes</td>
</tr>
<tr>
<td>FMA</td>
<td>Yes</td>
</tr>
<tr>
<td>Square root</td>
<td>Software with low-latency FMA-based convergence</td>
</tr>
<tr>
<td>Division</td>
<td>Software with low-latency FMA-based convergence</td>
</tr>
<tr>
<td>Reciprocal estimate accuracy</td>
<td>24 bit</td>
</tr>
<tr>
<td>Reciprocal sqrt estimate accuracy</td>
<td>23 bit</td>
</tr>
<tr>
<td>$\log_2(x)$ and $2^x$ estimates accuracy</td>
<td>23 bit</td>
</tr>
</tbody>
</table>
Memory subsystem

- Several memory partitions
- Each partition has its own 64-pin connection
- Supports up to 4 GB memory
- Arbitrary load/store model (concurrent read concurrent write)
- But: Arbitrary value is written: All CRCW hazards are avoided or better, placed in the programmer’s responsibility
- GTX 280: 8 memory partitions, 512-pin connection
Scalable processor array

- **Scalable design**
  - Multiprocessors form scalable processor array
  - Different price-performance regimes
  - Varying number of multiprocessors (automatically scaling execution)
  - Varying number of memory partitions

- **Different clock domains**
  - Core clock (instructions)
  - SPA clock (compute, typically 2x core)
  - Memory: > 1 GHz DDR (> 2 GHz effective)
## Compute capabilities

- **deviceQuery SDK sample**

```
Device 0: "GeForce GTX 280"
  Major revision number: 1
  Minor revision number: 3
  Total amount of global memory: 107344144 bytes
  Number of multiprocessors: 30
  Number of cores: 240
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 16384 bytes
  Total number of registers available per block: 16384
  Warp size: 32
  Maximum number of threads per block: 512
  Maximum sizes of each dimension of a block: 512 x 512 x 64
  Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
  Maximum memory pitch: 262144 bytes
  Texture alignment: 256 bytes
  Clock rate: 1.30 GHz
  Concurrent copy and execution: Yes

Device 1: "GeForce 8600 GT"
  Major revision number: 1
  Minor revision number: 1
  Total amount of global memory: 536870912 bytes
  Number of multiprocessors: 4
  Number of cores: 32
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 16384 bytes
  Total number of registers available per block: 8192
  Warp size: 32
  Maximum number of threads per block: 512
  Maximum sizes of each dimension of a block: 512 x 512 x 64
  Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
  Maximum memory pitch: 262144 bytes
  Texture alignment: 256 bytes
  Clock rate: 1.19 GHz
  Concurrent copy and execution: Yes
```
Memory spaces

- **Global memory**
  - Read/write
  - 100s of MB
  - Very slow (600+ cycles)

- **Texture memory**
  - Physically the same as global
  - Read-only
  - Cached for streaming throughput (2D neighborhoods)
  - Built-in filtering and clamping
Memory spaces

- **Constant memory**
  - Read-only
  - 64kB per chip
  - Very fast (1-4 cycles)

- **Shared memory**
  - Read/write
  - 16kB per multiprocessor
  - Very fast if DRAM bank conflicts are avoided

- **Registers**
  - Read/write
  - 16K per multiprocessor (8K on G8x and G9x)
  - Fastest
Compiling CUDA for GPUs

C/C++ CUDA Application

NVCC

CPU Code

PTX Code

PTX to Target Translator

GPU

Target device code
Outline

• CUDA parallel hardware architecture

• CUDA programming model

• Code walkthrough

• Libraries

• Tool chain and OpenCL

• Tesla compute hardware
Some design goals

• Scale to 100s of cores, 1000s of parallel threads

• Let programmers focus on parallel algorithms
  • *not* mechanics of a parallel programming language
  • C for CUDA plus runtime API

• Enable heterogeneous systems (i.e., CPU+GPU)
  • CPU & GPU are separate devices with separate DRAMs
Key parallel abstractions in CUDA

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads
Hierarchy of concurrent threads

- **Parallel kernels composed of many threads**
  - All threads execute the same sequential program

- **Threads are grouped into thread blocks**
  - Threads in the same block can cooperate

- **Threads/blocks have unique IDs**

- **Thread blocks are arranged in a grid**
### Device Code

```c
// Compute vector sum C = A+B

// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```
// Compute vector sum $C = A + B$
// Each thread performs one pair-wise addition

__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run $N/256$ blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
Synchronization of blocks

- Threads within block may synchronize with **barriers**
  
  ... Step 1 ...
  __syncthreads();
  ... Step 2 ...

- Blocks can **coordinate via atomic memory operations**
  - e.g., increment shared queue pointer with **atomicInc()**

- **Implicit barrier between dependent kernels**
  
  vec_minus<<<nblocks, blksize>>>(a, b, c);
  vec_dot<<<nblocks, blksize>>>(c, c);
What is a thread?

• **Independent thread of execution**
  • Has its own PC, variables (registers), processor state, etc.
  • No implication about how threads are scheduled

• **CUDA threads might be physical threads**
  • As on NVIDIA GPUs

• **CUDA threads might be virtual threads**
  • Might pick 1 block = 1 physical thread on multicore CPU as in MCUDA
What is a thread block?

• **Thread block = virtualized multiprocessor**
  • Freely choose processors to fit data
  • Freely customize for each kernel launch

• **Thread block = a (data) parallel task**
  • All blocks in kernel have the same entry point
  • But may execute any code they want

• **Thread blocks of kernel must be independent tasks**
  • Program valid for *any interleaving* of block executions
Blocks must be independent

• Any possible interleaving of blocks should be valid
  • Presumed to run to completion without pre-emption
  • Can run in any order
  • Can run concurrently OR sequentially

• Blocks may coordinate but not synchronize
  • Shared queue pointer: OK
  • Shared lock: BAD … can easily deadlock

• Independence requirement gives scalability
  • And makes hardware realisation manageable
Levels of parallelism

- **Thread parallelism**
  - Each thread is an independent thread of execution

- **Data parallelism**
  - Across threads in a block
  - Across blocks in a kernel

- **Task parallelism**
  - Different blocks are independent
  - Independent kernels
Memory model

Thread

Per-thread Local Memory

Block

Per-block Shared Memory
Memory model

Sequential Kernels

Kernel 0

Kernel 1

Per-device Global Memory
Memory model

Host memory

cudaMemcpy()

Device 0 memory

Device 1 memory
Memory model

- Each thread can
  - 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 host can
  - Read/write global,
  - Constant, and
  - Texture memory (stored in DRAM)
Using per-block shared memory

• Variables shared across block
  ```
  __shared__ int *begin, *end;
  ```

• Scratchpad memory
  ```
  __shared__ int scratch[blocksize];
  scratch[threadIdx.x] = begin[threadIdx.x];
  // ... compute on scratch values ...
  begin[threadIdx.x] = scratch[threadIdx.x];
  ```

• Communicating values between threads
  ```
  scratch[threadIdx.x] = begin[threadIdx.x];
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```
Warps and half-warps

A thread block consists of 32-thread warps.

A warp is executed physically in parallel (SIMD) on a multiprocessor.

A half-warp of 16 threads can coordinate global memory accesses into a single transaction called coalescing.
Memory transaction coalescing

• Single most important performance tuning step

```c
__global__
void saxpy_with_stride(int n, float a, float *x, float *y, int stride)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n)
        y[i * stride] = a * x[i * stride] + y[i * stride];
}
```

![Graph showing performance comparison between single and double precision against stride]
Launch configuration

- Extended function invocation syntax for parallel kernel launch
  `KernelFunc<<<500, 128>>>(...);`

- 1D, 2D or 3D grids
- 1D, 2D or 3D blocks

- Allocate shared memory per kernel
  `knl<<<..,..shb>>>(...);`
C for CUDA: Minimal extensions

- **Declaration specifiers to indicate where things live**
  
  ```c
  __global__ void KernelFunc(...);  // kernel callable from host
  __device__ void DeviceFunc(...);  // function callable on device
  __device__ int GlobalVar;        // variable in device memory
  __shared__ int SharedVar;        // in per-block shared memory
  ```

- **Extend function invocation syntax for parallel kernel launch**
  
  ```c
  KernelFunc<<<500, 128>>>(...);    // 500 blocks, 128 threads each
  KernelFunc<<<500, 128, 1024>>>(...); // ... 1024B shared memory per block
  ```

- **Special variables for thread identification in kernels**
  
  ```c
  dim3 threadIdx;    dim3 blockIdx;    dim3 blockDim;
  ```

- **Intrinsics that expose specific operations in kernel code**
  
  ```c
  __syncthreads();       // barrier synchronization
  ```
GPU math and runtime libraries

- **Standard mathematical functions**
  
sinf, powf, atanf, ceil, min, sqrtf, etc.

- **Atomic memory operations**
  
  atomicAdd, atomicMin, atomicAnd, atomicCAS, etc.

- **Texture accesses in kernels**
  
  texture<float,2> my_texture;  // declare texture reference

  float4 texel = texfetch(my_texture, u, v);
Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - Pointer arithmetic possible, not allowed to take address
    `cudaMalloc()`, `cudaFree()`

- Explicit memory copy for host ↔ device, device ↔ device
  `cudaMemcpy()`, `cudaMemcpy2D()`, ...

- Texture management
  `cudaBindTexture()`, `cudaBindTextureToArray()`, ...

- OpenGL & DirectX interoperability
  `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

Summary

- **CUDA = C + a few simple extensions**
  - Makes it easy to start writing basic parallel programs

- **Three key abstractions:**
  1. Hierarchy of parallel threads
  2. Corresponding levels of synchronization
  3. Corresponding memory spaces

- Supports massive parallelism of many-core GPUs
Outline

- CUDA parallel hardware architecture
- CUDA programming model
- Code walkthrough
- Libraries
- Tool chain and OpenCL
- Tesla compute hardware
Code walkthrough

• Live code walkthrough (simpleCUDA.cu)

• More demos later: CUDA Performance Tips and Tricks
Outline

- CUDA parallel hardware architecture
- CUDA programming model
- Code walkthrough
- Libraries
- Tool chain and OpenCL
- Tesla compute hardware
Libraries

- CUBLAS
  - (subset of) level 1, 2 and 3 BLAS
  - C and Fortran bindings

- CUFFT
  - Modeled after FFTW (plan interface)

- SpMV
  - Early beta, but promising speedups

- CUDPP
  - CUDA data-parallel programming primitives
  - Reduce, scan, ...

- Ship with CUDA toolkit, easy to use
Matrix size vs. GFLOP/s
CUBLAS: CUDA 2.0, Tesla C1060 (10-series GPU)
ATLAS 3.81 on Dual 2.8GHz Opteron Dual-Core
Heterogeneous DGEMM

- Xeon Quad-core 2.8 GHz, MKL 10.3
- Tesla C1060 GPU (1.296 GHz)
- GPU + CPU
SpMV

**Single Precision**

- Tesla C1060 GPU: CSR
- Tesla C1060 GPU: HYB

**Double Precision**

- Intel Xeon Quad 2.3 GHz
- Tesla C1060 GPU: CSR
- Tesla C1060 GPU: HYB
Outline

• CUDA parallel hardware architecture
• CUDA programming model
• Code walkthrough
• Libraries
• Tool chain and OpenCL
• Tesla compute hardware
Profiler and debugger

- CUDA profiler
  - Query hardware performance counters
  - GUI frontend
  - Linux: console-style interface via environment variables also available

- Debugger
  - cudagdb in CUDA 2.2 beta
CUDA and OpenCL

C for CUDA Application

NVCC

C for CUDA Kernels

CUDACC

CUDA object files

Linker

Combined CPU-GPU Code

Rest of C Application

CPU Compiler

CPU object files

CPU-GPU Executable
CUDA and OpenCL

Entry point for developers who prefer high-level C

Entry point for developers who want low-level API (CUDA driver API)

Shared back-end compiler and optimization technology
Different programming styles

- **C for CUDA**
  - C with parallel keywords
  - C runtime that abstracts driver API
  - Memory managed by C runtime (familiar malloc, free)
  - Generates PTX
  - Low-level “driver” API optionally available

- **OpenCL**
  - Hardware API - similar to OpenGL and CUDA driver API
  - Memory managed by programmer
  - Generates PTX
• CUDA parallel hardware architecture
• CUDA programming model
• Code walkthrough
• Libraries
• Tool chain and OpenCL
• Tesla compute hardware
## Tesla GPU computing products

<table>
<thead>
<tr>
<th></th>
<th>Tesla S1070 1U System</th>
<th>Tesla C1060 Computing Board</th>
<th>Tesla Personal Supercomputer (4 Tesla C1060s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs</td>
<td>4 Tesla GPUs</td>
<td>1 Tesla GPU</td>
<td>4 Tesla GPUs</td>
</tr>
<tr>
<td>Single Precision Perf</td>
<td>4.14 Teraflops</td>
<td>933 Gigaflops</td>
<td>3.7 Teraflops</td>
</tr>
<tr>
<td>Double Precision Perf</td>
<td>346 Gigaflops</td>
<td>78 Gigaflops</td>
<td>312 Gigaflops</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB / GPU</td>
<td>4 GB</td>
<td>4 GB / GPU</td>
</tr>
</tbody>
</table>

Dominik Göddeke | TU Dortmund