CSE 591: GPU Programming

Programmer Interface

Klaus Mueller

Computer Science Department
Stony Brook University
Compute Levels

Encodes the hardware capability of a GPU card
• newer cards have higher compute levels
• higher compute levels have more or faster features

Find a card’s compute capability with CUDA runtime API
• `cudaGetDeviceProperties()` returns two fields: major and minor

Compute 1.0
• 8000 series of cards
• does not have atomic (un-interrupted completion) operations

Compute 1.1
• 9000 series of cards
• can overlap data transfer with kernel execution
• non-blocking kernel invocation
• will require more GPU memory (active + loading process)
Compute Levels

Compute 1.2
- GT 200 series of cards
- increased the number of warps from 24 to 32
- some memory management restrictions were also removed

Compute 1.3
- support of limited double-precision operations
- but watch out for significant performance drops
Compute Levels

Compute 2.0

- Fermi hardware
- New: L1 cache (6k-48k): encourages data re-use, locality
- New: shared L2 cache (up to 768k): enables inter-block communication
- ECC (Error Correcting Code): detect and correct single bit errors
- ECC only in Tesla boards, needed in data centers where the radiation of nearby processors could flip bits
- Dual copy engines and streams: run multiple asynchronous processes
- Switchable L1/shared memory (48k/16k or 16k/48k)
- Cache lining can be turned off
- Increase memory banks from 16 to 32: now an entire warp can write simultaneously
Compute Levels

Compute 2.1

- GTX 480
- 48 CUDA cores per SM (instead of 32) but some are only single precision
- 8 instead of 4 single precision special function units
- dual warp dispatcher
- exploits instruction level parallelism (gives super-scalar speedups like Pentium, but needs independent instructions)

New compute capabilities

- 3.0 (680 GTX series) and 3.1 (Tesla K20)
Checking GPU Device Capabilities

Call `cudaGetDeviceProperties()`

There is 1 device supporting CUDA

Device 0: "Quadro FX 570"
  Major revision number: 1
  Minor revision number: 1
  Total amount of global memory: 268107776 bytes
  Number of multiprocessors: 16
  Number of cores: 128
  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: 0.41 GHz
  Concurrent copy and execution: Yes

Test PASSED

Press ENTER to exit...
C for CUDA

- most intuitive
- exposes programming model as a minimal set of operations
- define kernel as a C-function
- compile with nvcc
- runtime API provides various functions
- runtime API is implemented in cudart DLL (prefix cuda)

CUDA driver API

- expert interface
- allows finer control
- define kernels as modules of CUDA binary or assembly code
- runtime API is built on top of CUDA driver API
Compilation with `nvcc`
- any CUDA source file must be compiled with `nvcc`
- produces `PTX` assembler code
- produces `cubin` binary objects
- produces C code (host CPU code)

`PTX` assembler code
- can be ported to different GPUs
- but may include advanced functionality only recent GPUs support
- check the `compute capability`
- `PTX` is backward compatible

`Cubin` binary objects
- specific to a particular GPU model
- generate by recompiling `PTX` (or higher level code)

Linking
- CUDA runtime library (cudart) and CUDA core library (cuda)
More Graphically

C/C++ CUDA Application

float4 me = gx[gtid];
me.x += me.y * me.z;

PTX Code

NVCC

CPU Code

Virtual

Physical

PTX to Target Compiler

ld.global.v4.f32 {f1, f3, f5, f7}, [r9+0];
mad.f32

G80

Target code

GPU

float4 me = gx[gtid];
me.x += me.y * me.z;

ld.global.v4.f32 {f1, f3, f5, f7}, [r9+0];
mad.f32

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign
Debugging Using the Device Emulation Mode

An executable compiled in device emulation mode (`nvcc -deviceemu`)

- runs completely on the host using the CUDA runtime
- no need of any device and CUDA driver
- each device thread is emulated with a host thread

Running in device emulation mode, one can:

- use host native debug support (breakpoints, inspection, etc.)
- access any device-specific data from host code and vice-versa
- call any host function from device code (e.g. `printf`) and vice-versa
- detect deadlock situations caused by improper usage of `__syncthreads`
Emulated device threads execute sequentially,

- simultaneous accesses of the same memory location by multiple threads could produce different results.

Dereferencing device pointers on the host or host pointers on the device

- can produce correct results in device emulation mode
- but will generate an error in device execution mode
A Word of Caution: Floating Point

Results of floating-point computations will slightly differ because of:

• different compiler outputs, instruction sets
• use of extended precision for intermediate results
  - there are various options to force strict single precision on the host
The API is an extension to the C programming language.

It consists of:

- **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 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
Declspecs

• global, device, shared, local, constant

__device__ float filter[N];
__global__ void convolve (float *image) {
    __shared__ float region[M];
    ...
    region[threadIdx] = image[i];
    ___syncthreads() 
    ...
    image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign
Extended C

Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
foo.s

OCG

G80 SASS
foo.sass

CPU Host Code
foo.cpp

gcc / cl

Mark Murphy, “NVIDIA’s Experience with Open64,”
www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc
**Language Extensions: Built-in Variables**

\[
\text{dim3 } \text{gridDim}; \\
\quad \cdot \text{ dimensions of the grid in blocks (gridDim.z unused)}
\]

\[
\text{dim3 } \text{blockDim}; \\
\quad \cdot \text{ dimensions of the block in threads}
\]

\[
\text{dim3 } \text{blockIdx}; \\
\quad \cdot \text{ block index within the grid}
\]

\[
\text{dim3 } \text{threadIdx}; \\
\quad \cdot \text{ thread index within the block}
\]
Common Runtime Component: Mathematical Functions

pow, sqrt, cbrt, hypot

exp, exp2, expm1

log, log2, log10, log1p

sin, cos, tan, asin, acos, atan, atan2

sinh, cosh, tanh, asinh, acosh, atanh

ceil, floor, trunc, round

e tc.

• when executed on the host, a given function uses the C runtime implementation if available

• these functions are only supported for scalar types, not vector types
Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. __sin(x))

- __pow
- __log, __log2, __log10
- __exp
- __sin, __cos, __tan

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign
Host Runtime Component

Provides functions to deal with:

- device management (including multi-device systems)
- memory management
- error handling

Initializes the first time a runtime function is called

A host thread can invoke device code on only one device

- multiple host threads required to run on multiple devices
void __syncthreads();

Synchronizes all threads in a block

Once all threads have reached this point, execution resumes normally

Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

Allowed in conditional constructs only if the conditional is uniform across the entire thread block
Setup CUDA

Compute Unified Device Architecture

• Hardware compatibility: http://www.nvidia.com/object/cuda_gpus.html

• Driver, Toolkit (7.0) and SDK: https://developer.nvidia.com/cuda-toolkit

• Toolkit includes:
  -- Compiler
  -- Development tools
  -- Libraries for scientific computation
    (CUBLAS, CUFFT, CUSPARSE, CURAND, etc.)
  -- User guides and documents
Any source file containing CUDA language extensions must be compiled with NVCC

NVCC is a compiler

• Compile device code
• Invoking the necessary compilers for host code like, g++, cl, ...

Any executable with CUDA code requires dynamic libraries:

• The CUDA runtime library (`cudart`) OR
• The CUDA core library (`cuda`)
Development Tools

NVIDIA Nsight (Windows)

• Visual Studio Based GPU Development Environment
• Debug CUDA C/C++ source code directly on the GPU
• Use the familiar Visual Studio Locals, Watches, Memory and Breakpoints windows
• Integrated analysis tool to isolate performance bottleneck
• support for Visual Studio, Eclipse

CUDA-GDB debugger

for Linux and MacOS
A graphical profiling tool to measure and benchmark performance tracks events with hardware counters on signals in the chip.

Fine Tuning Performance by watching the following metric:

- Coalescing
- Occupancy
- Branch diversity
- Instruction throughput
- Computing / Data transfer ratio
- Share memory and register per thread