gpu computing with cuda: a hands-on tutorial

christiaan p. gribble
department of computer science
grove city college

pittsburgh perl workshop
11 october 2008

image source: nvidia.com
agenda

• gpu computing
  – what is it?
  – why use it?
  – what about cuda?

• compute unified device architecture
  – programming model
  – key abstractions
  – basic techniques
what is gpu computing, anyway?
gpu

• graphics processing unit
• evolution
  – simple accelerators
  – fixed-function rendering pipeline
  – programmable shading
  – manycore general-purpose processing
g80 architecture

- thread execution manager
- several multiprocessors
  - thread processors
  - on-chip shared memory
  - constant + texture cache
- globally-accessible dram
execution manager

- low-overhead threads
  - handles thread creation
  - issues threads for execution
  - handles context switches
- opportunities for latency hiding
- hides nasty details
multiprocessor

- multiple thread processors
- on-chip memory
  - set of 32-bit registers
  - parallel data cache
    - accessible from all thread processors
    - r/w access: 4-6 cycles
  - read-only constant cache
  - read-only texture cache

![Diagram of multiprocessor architecture]

mp0
mp1
mp2
mp15

DRAM
dram

- typically 256 MB to 1 GB
- globally accessible
- r/w access time
  - 400-600 cycles
  - 100x slower than PDC
- no hw caching
gpu computing

- using gpu architectures to solve non-traditional (ie, non-graphics) problems
- possible applications
  - computational science
  - image processing
  - scientific visualization

image source: nvidia.com
ok, but why use gpus?
parallel computing

• golden age
  – 1980s, early 1990s
  – wide variety of hw, algorithms, languages, & programming models
  – focus on data parallel computing

• dark age
  – commercial & research activity subsided
  – commodity processors gained momentum
  – focus shifted to commodity clusters
dawn of the gpu

- current gpus
  - massively multithreaded
  - highly programmable
- outperform cpus
  - speedups of 10x-100x
  - gflops/dollar
Dawn of the GPU

- Current GPUs
  - Massively multithreaded
  - Highly programmable
- Outperform CPUs
  - Speedups of 10x-100x
  - GFLOPs/dollar

For the right applications
supercomputing on a budget: more !, less $
so what about cuda?
nvidia’s cuda

• compute unified device architecture
  – co-designed hw & sw architecture
  – hw execution environment
  – sw dev tools & environment
  – scalable parallel programming model

• design goals
  – scalability: 100s of cores, 1000s of threads
  – extensibility: enable cpu + gpu systems
  – usability: programmer focus on algorithms
programming model

- GPU as compute coprocessor
  - highly multithreaded
  - local DRAM
  - SPMD-style processing
- Exposed via extensions to C/C++
programming model

• gpu as compute coprocessor
  – highly multithreaded
  – local dram
  – spmd-style processing

• exposed via extensions to C/C++
programming model

• gpu as compute coprocessor
  – highly multithreaded
  – local dram
  – spmd-style processing

• exposed via extensions to C/C++

gpu == compute also maps well to multicore cpus
execution model

• parallel execution via kernels
  – many threads execute in each kernel
  – one kernel executed at a time

• key abstractions
  – many lightweight threads
  – hierarchy of concurrent threads
  – lightweight sync primitives
  – cooperation via shared memory
thread hierarchy

- thread
- warp
- block
- grid
thread hierarchy

- thread
- warp
- block
- grid
thread hierarchy

- thread
- warp
- block
- grid
thread hierarchy

- thread
- warp
- block
- grid
memory model

- per-thread
- per-block
- per-device

thread \to\!

\iffalse

\begin{itemize}
\item per-thread
\item per-block
\item per-device
\end{itemize}

\fi
memory model

- **per-thread**

- **per-block**

- **per-device**
memory model

- per-thread
- per-block
- per-device
basic concepts

• memory management
  – allocate & release
  – data movement

• kernels & functions
  – invocation
  – qualifiers

• variables
  – vector types
  – qualifiers
memory management

• host manages device memory
• allocate & release

int nbytes = 1024*sizeof(int);
int* dptr = 0;
cudaMalloc((void**)&dptr, nbytes);
cudaMemset(dptr, 0, nbytes);
cudaFree(dptr);

cudaMalloc(void** ptr, size_t nbytes)
cudaMemset(void*  ptr, int value, size_t count)
cudaFree(void* ptr)
data movement

cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind dir)

• blocks calling thread
• copies memory according to
  enum CudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
kernels

• C function with some restrictions
  – no access to host memory
  – no variable arg lists
  – no recursion
  – no static variables
  – must return \texttt{void}

• args automatically transferred from host to device
kernel invocation

- call syntax

  \[ \text{kernel} \lll \text{dim3 } dG, \text{ dim3 } dB \rrl (\ldots); \]

- configuration
  - \( dG \rightarrow \) dim & size of grid (in blocks)
  - \( dB \rightarrow \) dim & size of blocks (in threads)
functions

• __global__
  – called from host, executed on device
  – must return void

• __device__
  – called from device, executed on device
  – cannot be called from host

• __host__
  – called from host, executed on host
  – default if unspecified
variables

- **__device__**
  - stored in global memory, high latency
  - accessible to all threads
- **__shared__**
  - stored in shared memory, very low latency
  - accessible to all threads within same block
- unqualified
  - scalar/vector types stored in registers
  - spilled to local memory if necessary
vector types

• integral types
  - char, short, int, long
  - one to four elements
  - unsigned varieties (uchar, etc.)
  - dim3 $\rightarrow$ uint3

• floating-point types
  - float[1..4]
  - double[1..2]
void inc_cpu(int* a, int N) {
    int i;

    for (i = 0; i < N; i++)
        a[i] = a[i] + 1;
}

int main()
{
...

inc_cpu(a, N);
...
}

gpu program

__global__ void inc_gpu(int* a, int N) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < N)
        a[i] = a[i] + 1;
}

int main()
{
...

    dim3 dB(nthreads);
    dim3 dG(ceil(N/(float)dB));
    inc_gpu<<<dG, dB>>>(a, N);
...
}
resources

• cuda zone
  – site for cuda developers

• cuda 2.0 documentation
  – quick start guide
  – programming guide
  – reference manual
  – …

• nvision 08 conference materials
acknowledgements

- pghpw.org
- robert blackwell
- nvidia professor partnership program
thank you

contact information

cpgribble@gcc.edu
724.450.1535 (office)
724.450.4031 (fax)

http://www2.gcc.edu/dept/comp/faculty/gribblecp/