CSC266 Introduction to Parallel Computing using GPUs
Introduction to Accelerators

Sreepathi Pai
October 11, 2017

URCS
Introduction to Accelerators

GPU Architectures

GPU Programming Models
Outline

Introduction to Accelerators

GPU Architectures

GPU Programming Models
Accelerators

- Single-core processors
- Multi-core processors
- What if these aren’t enough?
- Accelerators, specifically GPUs
  - what they are
  - when you should use them
Timeline

• 1980s
  • Geometry Engines

• 1990s
  • Consumer GPUs
  • Out-of-order Superscalars

• 2000s
  • General-purpose GPUs
  • Multicore CPUs
  • Cell BE (Playstation 3)
  • Lots of specialized accelerators in phones
The Graphics Processing Unit (1980s)

- SGI Geometry Engine
- Implemented the *Geometry Pipeline*
  - Hardwired logic
- Embarrassingly Parallel
  - $O(\text{Pixels})$
  - Large number of logic elements
  - High memory bandwidth
- From Kaufman et al. (2009):
GPU 2.0 (circa 2004)

• Like CPUs, GPUs benefited from Moore’s Law
• Evolved from fixed-function hardwired logic to flexible, programmable ALUs
• Around 2004, GPUs were programmable “enough” to do some non-graphics computations
  • Severely limited by graphics programming model (shader programming)
• In 2006, GPUs became “fully” programmable
  • GPGPU: General-Purpose GPU
  • NVIDIA releases “CUDA” language to write non-graphics programs that will run on GPUs
GPGPU Today

- GPUs are widely deployed as accelerators
- Intel Paper
  - 10x vs 100x Myth
- GPUs so successful that other accelerators are dead
  - Sony/IBM Cell BE
  - Clearspeed RSX
- Kepler K40 GPUs from NVIDIA have performance of 4TFlops (peak)
  - CM-5, #1 system in 1993 was 60 Gflops (Linpack)
  - ASCI White (#1 2001) was 4.9 Tflops (Linpack)

Pictures of Titan and Tianhe 1A from the Top500 website.
Accelerator Programming Models

- CPUs have always depended on co-processors
  - I/O co-processors to handle slow I/O
  - Math co-processors to speed up computation
  - H.264 co-processor to play video (Phones)
  - DSPs to handle audio (Phones)
- Many have been transparent
  - Drop in the co-processor and everything sped up
- Or used a function-based model
  - Call a function and it is sped up (e.g. “decode video”)
- The GPU is not a transparent accelerator for general purpose computations
  - Only graphics code is sped up transparently
- Code must be rewritten to target GPUs
Using a GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.
Introduction to Accelerators

GPU Architectures

GPU Programming Models
The Two (Three?) Kinds of GPUs

- Type 1: Discrete GPUs
  - More computational power
  - More memory bandwidth
  - Separate memory
The Two (Three?) Kinds of GPUs #2

- Type 2: Integrated GPUs
  - Share memory with processor
  - Share bandwidth with processor
  - Consume Less power
  - Can participate in cache coherence
The NVIDIA Kepler

PCI Express 3.0 Host Interface

GigaThread Engine

Memory Controller

L2 Cache

NVIDIA Kepler GK110 Whitepaper
• You must retarget code for the GPU
  • Rewrite, recompile, translate, etc.
• Working set must fit in GPU RAM
• You must copy data to/from GPU RAM
  • “You”: Programmer, Compiler, Runtime, OS, etc.
  • Some recent hardware can do this for you (it’s slow)
NVIDIA Kepler SMX (i.e. CPU core equivalent)
NVIDIA Kepler SMX Details

- 2-wide Inorder
- 4-wide SMT
  - 2048 threads per core (64 warps)
  - 15 cores
  - Each thread runs the same code (hence SIMT)
- 65536 32-bit registers (256KBytes)
  - A thread can use upto 255 of these
  - Partitioned among threads (not shared!)
- 192 ALUs
- 64 Double-precision
- 32 Load/store
- 32 Special Functional Unit
- 64 KB L1/Shared Cache
  - Shared cache is software-managed cache
## CPU vs GPU

<table>
<thead>
<tr>
<th>Parameter</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clockspeed</td>
<td>&gt; 1 GHz</td>
<td>700 MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>GB to TB</td>
<td>12 GB (max)</td>
</tr>
<tr>
<td>Memory B/W</td>
<td>60 GB/s</td>
<td>&gt; 300 GB/s</td>
</tr>
<tr>
<td>Peak FP</td>
<td>&lt; 1 TFlop</td>
<td>&gt; 1 TFlop</td>
</tr>
<tr>
<td>Concurrent Threads</td>
<td>O(10)</td>
<td>O(1000) [O(10000)]</td>
</tr>
<tr>
<td>LLC cache size</td>
<td>&gt; 100MB (L3) [eDRAM]</td>
<td>&lt; 2MB (L2)</td>
</tr>
<tr>
<td></td>
<td>O(10) [traditional]</td>
<td></td>
</tr>
<tr>
<td>Cache size per thread</td>
<td>O(1 MB)</td>
<td>O(10 bytes)</td>
</tr>
<tr>
<td>Software-managed cache</td>
<td>None</td>
<td>48KB/SMX</td>
</tr>
<tr>
<td>Type</td>
<td>OOO super-scalar</td>
<td>2-way Inorder super-scalar</td>
</tr>
</tbody>
</table>
Using a GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.
- Working set must fit in GPU RAM
- You must copy data to/from GPU RAM
  - "You": Programmer, Compiler, Runtime, OS, etc.
  - Some recent hardware can do this for you
- Data accesses should be streaming
  - Or use scratchpad as user-managed cache
- Lots of parallelism preferred (throughput, not latency)
- SIMD-style parallelism best suited
- High arithmetic intensity (FLOPs/byte) preferred
Showcase GPU Applications

- Image Processing
- Graphics Rendering
- Matrix Multiply
- FFT

See “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU” by V.W.Lee et al. for more examples and a comparison of CPU and GPU.
Outline

Introduction to Accelerators

GPU Architectures

GPU Programming Models
## Hierarchy of GPU Programming Models

<table>
<thead>
<tr>
<th>Model</th>
<th>GPU</th>
<th>CPU Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorizing Compiler</td>
<td>PGI CUDA Fortran</td>
<td>gcc, icc, etc.</td>
</tr>
<tr>
<td>“Drop-in” Libraries</td>
<td>cuBLAS</td>
<td>ATLAS</td>
</tr>
<tr>
<td>Directive-driven</td>
<td>OpenACC, OpenMP-to-CUDA</td>
<td>OpenMP</td>
</tr>
<tr>
<td>High-level languages</td>
<td>pyCUDA</td>
<td>python</td>
</tr>
<tr>
<td>Mid-level languages</td>
<td>OpenCL, CUDA</td>
<td>pthreads + C/C++</td>
</tr>
<tr>
<td>Low-level languages</td>
<td>PTX, Shader</td>
<td>-</td>
</tr>
<tr>
<td>Bare-metal</td>
<td>SASS</td>
<td>Assembly/Machine code</td>
</tr>
</tbody>
</table>
“Drop-in” Libraries

- “Drop-in” replacements for popular CPU libraries, examples from NVIDIA:
  - CUBLAS/NVBLAS for BLAS (e.g. ATLAS)
  - CUFFT for FFTW
  - MAGMA for LAPACK and BLAS

- These libraries may still expect you to manage data transfers manually

- Libraries may support multiple accelerators (GPU + CPU + Xeon Phi)
GPU Libraries

- NVIDIA Thrust
  - Like C++ STL, but executes on the GPU
- Modern GPU
  - At first glance: high-performance library routines for sorting, searching, reductions, etc.
  - A deeper look: Specific “hard” problems tackled in a different style
- NVIDIA CUB
  - Low-level primitives for use in CUDA kernels
Directive-Driven Programming

- OpenACC, new standard for “offloading” parallel work to an accelerator
  - Currently supported only by PGI Accelerator compiler
  - gcc 5.0 support is ongoing
- OpenMPC, a research compiler, can compile OpenMP code + extra directives to CUDA
  - OpenMP 4.0 also supports offload to accelerators
  - Not for GPUs yet

```c
int main(void) {
    double pi = 0.0f; long i;

    #pragma acc parallel loop reduction(+:pi)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }

    printf("pi=\16.15f\n",pi/N);
    return 0;
}
```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule(""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print dest-a*b
OpenCL

- C99-based dialect for programming heterogeneous systems
  - Originally based on CUDA
  - nomenclature is different
- Supported by more than GPUs
  - Xeon Phi, FPGAs, CPUs, etc.
- Source code is portable (somewhat)
  - Performance may not be!
- Poorly supported by NVIDIA
CUDA

- “Compute Unified Device Architecture”
- First language to allow general-purpose programming for GPUs
  - preceded by shader languages
- Promoted by NVIDIA for their GPUs
- Not supported by any other accelerator
  - though commercial CUDA-to-x86/64 compilers exist
- We will focus on CUDA programs
CUDA Architecture

- From 10000 feet – CUDA is like pthreads
  - CUDA language – C++ dialect
- Host code (CPU) and GPU code in same file
- Special language extensions for GPU code
- CUDA Runtime API
  - Manages runtime GPU environment
  - Allocation of memory, data transfers, synchronization with GPU, etc.
  - Usually invoked by host code
- CUDA Device API
  - Lower-level API that CUDA Runtime API is built upon
CUDA Limitations

- No standard library for GPU functions
- No parallel data structures
- No synchronization primitives (mutex, semaphores, queues, etc.)
  - you can roll your own
  - only atomic*() functions provided
- Toolchain not as mature as CPU toolchain
  - Felt intensely in performance debugging
- It's only been a decade :)


Conclusions

- GPUs are very interesting parallel machines
- They’re not going away
  - Xeon Phi might pose a formidable challenge
- They’re here and now
  - Your laptop probably already contains one
  - Your phone definitely has one