GPGPU

General Purpose Graphics Processing Unit

Thomas Swift
Sean Brennan
Andrew Wong
Outline

- GPGPU Overview
  - Terminology
  - Graphics Pipeline
  - CPU vs GPU
- CUDA
- NVIDIA's Kepler & AMD's GCN Architectures
- PTask
GPUs in Action
Graphics Processing Unit

- Traditionally used for 3D rendering, but now also used for large computations
  - Originally special function units with specialized HW & ISAs
  - Good at applying same operation to large number of independent elements, in parallel
- Why GPUs?
  - High performance/throughput for massively parallel computations
  - Much higher arithmetic capability and memory bandwidth than even high-end CPUs
Terminology

- Streaming Processor abstraction
  - Programming model designed to abstract away all graphics terminology of GPU
  - Stream
    - Ordered set of data
  - Kernel
    - Function applied element-wise to a set of streams and that outputs one or more streams
- SIMD - Single Instruction, Multiple Data
  - SPMD - Single Program, Multiple Data
Graphics Pipeline

- A series of generation and processing stages
  - Connected by stream-entities
- Processing stages are programmable
  - Shader functions used to alter appearance of graphical output.
Vertex Generation

- VG - prefetches vertex and texture data from memory and constructs a stream of vertex data.
Vertex Processing

- VP - programmable operation on each vertex (e.g. computing projection from 3D space to screen).
Primitive Generation

- **PG** - groups vertices into ordered streams of primitives.

- **PP** - Produces zero or more output primitives.

![Diagram](image-url)
● FG - performs rasterization through sampling. Distance from camera and other parameters are saved
Fragment Processing

- FP - simulates light interaction to determine color and opacity.
Pixel Operations

- PO - calculates image pixel values based on distance and obstructions.
Hardware Multithreading

- Threads stall due to memory accesses
- Hardware contains multiple execution units to perform instructions from other threads when a thread stalls.
# CPU vs GPU

## CPUs
- Parallelism through time multiplexing
- Emphasis on low memory latency
- Allows wide range of control flows + control flow optimizations
- Low-latency caches that allow performant out-of-order execution
- Very high clock speeds

## GPUs
- Parallelism through space multiplexing
- Emphasis on high memory throughput
- Very control flow restricted
- High-latency caches that tend to be read-only
- Mid-tempo clock speeds

## Other GPU Points
- Power efficient for large parallel operations
- Inexpensive on a TFLOP basis
- Difficult to program - but getting better!
CUDA

- **Compute Unified Device Architecture**
- Platform and programming model by Nvidia
- Introduced in 2006 w/ GeForce 8800 GTX
- First architecture targeted at general purpose use
  - CUDA C provides high-level language familiar to most programmers
  - ALUs built for more general types of computation
  - *Unified Shader Model* improves use of GPU resources
Unified Shader Models

- **Unified Shader Architecture**: all GPU units designed to handle any shader
- **Unified Shader Model**: all shaders have similar instruction set
- Unified *Model* does NOT require Unified *Architecture*!
- Advantages over "classical" model:
  - more dynamic and flexible use of GPU resources
  - open to different workflows
    - both of these make USA/USM well-suited to GPGPU programming
Programming in CUDA

- CUDA C: good old C + a few new functions, structs, and primitives
- Kernel functions: __global__ and __device__
  - __global__: code executed on GPU from CPU
  - __device__: code executed on GPU from other GPU functions
- Grid abstraction: *spatial multiplexing*
  - grid → blocks → threads
  - grid (2D) + block (3D) = 5 degrees of indexing freedom
- SIMD paradigm: **Single Instruction Multiple Data**
  - well-suited to data-parallel tasks
  - conditionals are costly and should be avoided
CUDA Workflow

1. allocate data in main memory and on GPU

2. move data from MM → GPU

3. issue kernel over given block + thread count

```c
15 // allocate arrays on host
16 a_h = (float *)malloc(size);
17 b_h = (float *)malloc(size);
18
19 // allocate array on device
20 cudaMalloc((void **) &a_d, size);

26 // copy data from host to device
27 cudaMemcpy(a_d, a_h, sizeof(float) * N, cudaMemcpyHostToDevice);

34 // Part 2 of 2. Call incrementArrayOnDevice kernel
35 incrementArrayOnDevice <<< nBlocks, blockSize >>>(a_d, N);
```
CUDA Workflow

4. kernels execute in parallel
   ```c
   __global__ void incrementArrayOnDevice(float *a, int N) {
   // calculate index to work over; increment
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < N)
   a[idx] = a[idx] + 1.f;
   }
   ```

5. copy data from GPU → MM
   ```c
   // Retrieve result from device and store in b_h
   cudaMemcpy(b_h, a_d, sizeof(float) * N, cudaMemcpyDeviceToHost);
   ```
CUDA Memory Layout

- Per-thread registers
  - Very fast; lifetime of thread
- Per-block shared memory
  - Very fast; lifetime of block
- All-block global memory
  - ~100x slower than shmem
  - High throughput achieved through coalescing
- Per-thread local memory
  - Gotcha! as slow as global
- All-block constant memory
  - Half-warp broadcast reduces bandwidth
- All-block texture memory
  - Useful when exploiting spatial locality
Global memory achieves high throughput through coalescing

- Works under certain global memory access patterns
- **Half-warp coalescing**: accesses by all threads in a half-warp (16 threads) are coalesced
  - threads must access 32, 64, or 128-bit data types
  - data accessed must be properly word-aligned
  - threads must access words of coalesced access in sequence

- Responsibility is placed upon CUDA programmer
  - no coalescing → huge hit to memory throughput

- Can use CUDA Profiler to track amount of coalesced/non-coalesced accesses
CUDA Streaming

- Task parallelism can be achieved through *streaming*
  - stream: queue of tasks to be performed on GPU
- Asynchronous copies hide latency of memory movement
  - requires page-locked ("pinned") memory
    - ruins virtual memory abstraction
    - steps on other processes' toes
- Later versions support two streaming memory accesses
CUDA Applications

- **CUBLAS**: CUDA Basic Linear Algebra Subprograms
  - Adapted from well-known FORTRAN BLAS package
  - Three tiers of routines
    - level 1: vector scaling, inner product
    - level 2: matrix-vector products, matrix triangularization
    - level 3: matrix-matrix multiplication

- **CUFFT**: CUDA Fast Fourier Transforms
  - 1D, 2D, and 3D transforms for real-valued and complex data

---

```
Matrix Size (N x N)
```

- CUBLAS-Zgemm vs MKL-Zgemm

```
Log2(size)
```

- cuFFT Single Precision
  - CUFFT vs MKL

---

* cuBLAS 4.1 on Tesla M2090, ECC on
* MKL 10.2.3, "YAN FT72-87015 Xeon x5680 Six-Core @ 3.33 GHz
* Performance may vary based on OS ver. and motherboard config.
NVIDIA
Kepler GK 110 Architecture

- Dynamic Parallelism
  - Without ANY CPU help, GPU can
    - dynamically create new worker threads
    - synchronize among thread results
    - schedule worker threads
  - Advantages:
    - GPU can adapt to varying amounts and types of parallel workloads (choose optimal # threads and program parameters)
    - CPU can perform other tasks in the meanwhile
    - avoids CPU-GPU data transfers
Dynamic Parallelism Example

Dynamic Parallelism

*Makes GPU Computing Easier & Broadens Reach*

Too coarse | Too fine | Just right
Kepler: New Features Cont'd

- **Hyper-Q**
  - Multiple streams (cores/threads/processes) can run work on a single GPU at the same time, using separate HW work queues
  - Prevents streams from blocking each other due to false dependencies

- **GPUDirect**
  - Allows multiple GPUs on the same machine/network to share data directly without using the CPU or main memory
  - RDMA feature allows third-party devices such as SSDs to directly access GPU memory
  - Greatly improves message passing performance
Hyper-Q Example

Left: only (C,P) and (R,X) can run concurrently
Right: all 3 streams can run concurrently
Kepler Full Chip Diagram

15 SMX Units (Streaming Multiprocessors)

+ 6 64-bit memory controllers
SMX Unit

- 192 single-precision CUDA cores
- 64 double-precision CUDA cores
- 32 special function units (SFU)
- 32 load/store units (LD/ST)
- 4 warp schedulers, 2 instruction dispatch units each
- 64KB memory
- 48KB read-only data cache
SMX Core Architecture

- Quad Warp Scheduler
  - 32 threads / warp and 2 instruction dispatch units / warp scheduler
  - 4 warps can be executed concurrently; 2 instructions at a time
- 255 registers per thread
- Shuffle Instruction
  - Allows any thread in a warp to read registers of any other thread in the same warp in a single step, instead of going through shared memory with separate LD/ST insts.
- 64KB memory split between shared memory & L1 cache
- 48KB read-only cache for constant data
  - can be managed automatically by the compiler, or manually by the programmer
AMD GCN Architecture

- Tighter CPU-GPU integration
  - Virtual Memory: supports 4KB pages
    - could allow CPU & GPU to share single address space in the future
  - GCN includes I/O MMU, which maps GPU addresses to CPU addresses
  - 64B cache lines
- Cache Coherency: data shared between cores through L2 cache, instead of having to synchronize by flushing to memory
Compute Unit Architecture

4 SIMD Units (10 wavefronts each), 16KB R/W L1-D cache, 32KB L1-I cache per 4 CU's, 64KB Local Data Share (LDS)
Compute Units (CU's)

- Basic computational building block
- 4 SIMD units; each has PC and IB for 10 wavefronts
- Each cycle, single SIMD picked via RR to issue up to 5 instructions
  - instructions must be of different types, from different wavefronts
- SIMD executes in parallel across multiple wavefronts, instead of in parallel within a single wavefront
- 16KB R/W L1-D cache & 32KB L1-I cache per 4 CU's
  - LRU replacement
- Local Data Share (LDS): 64KB memory used for intra-work-group synchronization
L1-D cache per CU, L1-I cache per 4 CU's, L2 cache partitioned and shared by all CU's
Cache Hierarchy

- L1-D cache per CU; L1-I cache per 4 CU's
  - write-through
  - data written to L2 cache at end of wavefront instruction, or at a barrier
  - work-group coherency
- L2 cache shared among all CU's and partitioned into one slice per memory channel
  - write-back
  - absorbs L1-D cache misses
  - synchronizes among different wavefronts --> global coherency
- All caches use LRU replacement policy
GPGPU Challenges

- GPGPUs are more than just I/O devices
  - First class computational devices
  - Fairness and isolation guarantees
- Preempting GPUs is not easy
  - Large number of parallel operations
- Limited interface
  - Drivers are black boxes
  - Existing OS/kernel interfaces use ioctl
- Memory can be disjoint
Additional Motivation

- Data movement tied to algorithms
- High-level languages (CUDA) are hard to use.
- New applications that require OS support
  - encrypted file systems
  - gesture detection
Scheduling Bottlenecks

- GPU accelerated tasks can impair seemingly unrelated tasks
  - GPU work causes system pauses
- CPU work interferes with GPU throughput

![Mouse Move Frequency](chart.png)

- Mouse Events / second
- Time (milliseconds)
- Chart showing frequency of mouse moves with and without GPU work.
Scheduling Bottlenecks (cont.)

No OS support $\rightarrow$ No isolation

GPU benchmark throughput

- invocations per second

- Higher is better

- no CPU load
  - 1200
  - 1000
  - 800
  - 600
  - 400
  - 200
  - 0

- high CPU load
  - 400
GPGPU Support

User-mode

Libraries

Application

GPGPU API

Shader/Kernel

Language Int.

GPU Runtime (CUDA, OpenCL, DirectX)

OS

ioctl

Hardware

GPU
PTask

1. GPUs under a single resource manager for fairness and isolation guarantees
2. Simplify development for accelerators/GPGPUs by introducing a programming model that manages devices, performs I/O, and deals with disjoint memory spaces.
3. Create an environment that allows modular and fast code
Dataflow Programming

- Modularity and efficiency
matrix gemm(A, B) {
    matrix AxB = new matrix();
copyToDevice(A);
copyToDevice(B);
invokeGPU(gemm_kernel, A, B, AxB);
copyFromDevice(AxB);
return AxB;
}

matrix AxBxC(A,B,C) {
    return gemm(gemm(A, B), C);
}
Matrix Multiplication (Cont.)

```
matrix AxBxC(A,B,C) {
    matrix AxB = new matrix();
    matrix AxBxC = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    copyToDevice(C);
    invokeGPU(gemm_kernel, A, B, AxB);
    invokeGPU(gemm_kernel, AxB, C, AxBxC);
    copyFromDevice(AxBxC);
    return AxBxC;
}
```

Not modular!
Dataflow Programming

- Allows modularity and efficiency
- Graph structured computation model
  - Units of computation are vertices
  - Vertices have data sources and data sinks (ports)
  - Channels connect ports
- Dataflow managed by the OS
ptask:
gemm(A,B)

matrix A

Channel

port:A

Channel

port:B

Channel

matrix AxB

matrix B

Channel

port:C

matrix AxB
- Dataflow is managed by the OS.
- PTasks (vertices) are computation units.
- Ports connected by channels (edges).
- Both modular and efficient.
Case Study: Gesture Recognition

- Computationally intensive
  - Ideal for GPGPU acceleration
- Latency requirements
- Largely data parallel
- Multiple user-kernel memory copies
- Follows dataflow paradigm
> capture | xform | filter | detect &
- **Handcode**
  - remove unnecessary data copying
- **Modular**
  - similar to Handcode but condensed into one process
- **Pipes**
  - > capture | xform | filter | detect &
- **PTasks**
  - Uses the PTask API

<table>
<thead>
<tr>
<th>impl</th>
<th>fps</th>
<th>tput (MB/s)</th>
<th>lat (ms)</th>
<th>user</th>
<th>sys</th>
<th>gpu</th>
<th>gmem</th>
<th>thrds</th>
<th>ws-delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core2-Quad host-based real-time</td>
<td>20.2</td>
<td>35.8</td>
<td>36.6</td>
<td>78.1</td>
<td>3.9</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>GTX580 handcode real-time</td>
<td>30</td>
<td>53.8</td>
<td>10.5</td>
<td>4.0</td>
<td>5.3</td>
<td>21.0</td>
<td>138</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>unconstrained</td>
<td>138</td>
<td>248.2</td>
<td>–</td>
<td>2.4</td>
<td>6.4</td>
<td>41.8</td>
<td>138</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>modular real-time</td>
<td>30</td>
<td>53.8</td>
<td>12.2</td>
<td>6.0</td>
<td>8.1</td>
<td>19.4</td>
<td>72</td>
<td>1</td>
<td>0.8 (1%)</td>
</tr>
<tr>
<td>unconstrained</td>
<td>113</td>
<td>202.3</td>
<td>–</td>
<td>5.7</td>
<td>8.6</td>
<td>55.7</td>
<td>72</td>
<td>1</td>
<td>0.9 (1%)</td>
</tr>
<tr>
<td>pipes real-time</td>
<td>30</td>
<td>53.8</td>
<td>14.4</td>
<td>6.8</td>
<td>16.6</td>
<td>18.9</td>
<td>72</td>
<td>3</td>
<td>45.3 (58%)</td>
</tr>
<tr>
<td>unconstrained</td>
<td>90</td>
<td>161.9</td>
<td>–</td>
<td>12.4</td>
<td>24.6</td>
<td>55.4</td>
<td>76</td>
<td>3</td>
<td>46.3 (59%)</td>
</tr>
<tr>
<td>ptask real-time</td>
<td>30</td>
<td>53.8</td>
<td>9.8</td>
<td>3.1</td>
<td>5.5</td>
<td>16.1</td>
<td>71</td>
<td>7</td>
<td>0.7 (1%)</td>
</tr>
<tr>
<td>unconstrained</td>
<td>154</td>
<td>275.3</td>
<td>–</td>
<td>4.9</td>
<td>8.8</td>
<td>65.7</td>
<td>79</td>
<td>7</td>
<td>1.4 (2%)</td>
</tr>
</tbody>
</table>
Gestural Interface Performance

Metric relative to handcode

- runtime
- user
- sys
- gpu-util
- latency

- handcode
- modular
- pipes
- ptask
References

- **Managing Accelerators: the Operating System Perspective** K. Shantonu.
- **CUDA by Example** J. Sanders and E. Kandrot. April 2011.
- **NVIDIA Kepler GK110 Architecture** Nvidia Corporation. 2012.
- **AMD GCN Architecture** AMD. June 2012.
PTask API

sys_open_graph    Create/open graph
sys_open_port    Create/open port
sys_open_ptask    Create/open a ptask
sys_open_channel    Create and bind a channel
sys_open_template    Create/open a template
sys_push    Write to a channel/port
sys_pull    Read from a channel/port
sys_run_graph    Run a graph
sys_terminate_graph    Terminate graph
sys_set_ptask_prio    Set ptask priority
sys_set_geometry    Set iteration space
PTask Scheduling

- First-available
- Fifo
- Priority
- Data-aware