Data Processing on Modern Hardware

Jens Teubner, TU Dortmund, DBIS Group
jens.teubner@cs.tu-dortmund.de

Summer 2017
Part VI

Graphics Processors (GPUs)

I adopted some of this material from a slide set of René Müller (now with IBM Research).
While **general-purpose CPUs** increasingly feature “multi-media” functionality,

**graphics processors** become increasingly **general-purpose**.
Graphics Pipeline

1. **App** → **API** → **Front-End**
   - **vertices** → **Transform & Lighting**
   - **Geometry & Primitive Assembly** → **Rasterization**

2. **Frame Buffer**
   - **Raster & Operations**
   - **Fragment Coloring & Texture**
     - **Scissor**
     - **Alpha**
     - **Stencil**
     - **Depth**
     - **Test**

**connectivity information**
Some tasks in the pipeline lend themselves to in-hardware processing.

- Embarrassingly parallel
- Few and fairly simple operations
- Hardly need to worry about caches, coherency, etc.

Early cards did the end of the pipeline in hardware; today’s cards can do much more.
The programmability of GPUs has improved dramatically.

- **hard-coded fix-function pipeline**
- customization through **parameters**
- **programmable shaders**
  - vertex shader
  - geometry shader
  - fragment shader (fragment: pixel)
- “general-purpose” GPUs (GPGPUs)

**Today:** C-like languages (e.g., CUDA, OpenCL)
General-Purpose GPUs (GPGPUs)

Original GPU design based on graphics pipeline not flexible enough.
→ geometry shaders idle for pixel-heavy workloads and vice versa
→ **unified model** with general-purpose cores

**Thus:** Design inspired by CPUs, but different

Rationale: Optimize for **throughput**, not for **latency**.
CPUs vs. GPUs

CPU: task parallelism
- relatively heavyweight threads
- 10s of threads on 10s of cores
- each thread managed explicitly
- threads run different code

GPU: data parallelism
- lightweight threads
- 10,000s of threads on 100s of cores
- threads scheduled in batches
- all threads run same code
  → SPMD, single program, multiple data
Threads on a GPU

To handle 10,000s of threads efficiently, keep things simple.

- Don’t try to reduce latency, but hide it.
  - Large thread pool rather than caches
    (This idea is similar to SMT in commodity CPUs \(\rightarrow\) slide 139.)

- Assume data parallelism and restrict synchronization.
  - Threads and small groups of threads use local memories.
  - Synchronization only within those groups (more later).

- Hardware thread scheduling (simple, in-order).
  - Schedule threads in batches (\(\sim\) “warps”).
OpenCL Computation Model

- Host system and **co-processor** (GPU is only one possible co-processor.)
- Host triggers
  - data copying
    - host ↔ co-processor,
  - invocations of **compute kernels**.
- Host interface: **command queue**.
A traditional loop

```c
for (i = 0; i < nitems; i++)
    do_something(i);
```

becomes a data parallel kernel invocation in OpenCL ($\map$):

```c
status = clEnqueueNDRangeKernel (  
    commandQueue,
    do_something_kernel,...,&nitems,...);
```

```c
__kernel void do_something_kernel(...) {
    int i = get_global_id(0);
    ...
}
```
Kernel Invocation

**Idea:** Invoke kernel for each point in a problem domain

- e.g., $1024 \times 1024$ image, one kernel invocation per pixel; → 1,048,576 kernel invocations (“work items”).
- Don’t worry (too much) about task → core assignment or number of threads created; **runtime** does it for you.
- Problem domain can be 1-, 2-, or 3-dimensional.

- Can pass global parameters to all work item executions.
- Kernel must figure out work item by calling `get_global_id()`.
OpenCL defines a **C99-like** language for compute kernels.

- Compiled **at runtime** to particular core type.
- Additional set of built-in functions:
  - Context (*e.g.*, `get_global_id()`); synchronization.
  - Fast implementations for special math routines.

```c
__kernel void square (__global float *in,
                      __global float *out)
{
    int i = get_global_id(0);
    out[i] = in[i] * in[i];
}
```
Work Items and Work Groups

Work items may be grouped into work groups.

- Work groups ⋐ scheduling batches.
- Synchronization between work items only within work groups.
- There is a device-dependent limit on the number of work items per work group (can be determined via clGetDeviceInfo ()).
- Specify items per group when queuing the kernel invocation.
- All work groups must have same size (within one invocation).

E.g., Problem space: 800 × 600 items (2-dimensional problem).
→ Could choose 40 × 6, 2 × 300, 80 × 5, ... work groups.
Example: NVIDIA GPUs

NVIDIA GTX 280

- 10 Thread Processing Clusters
- $10 \times 3$ Streaming Multiprocessors
- $10 \times 3 \times 8$ Scalar Processor Cores
  - More like ALUs (→ slide 259)
- Each Multiprocessor:
  - 16k 32-bit registers
  - 16 kB shared memory
  - up to 1024 threads
  - (may be limited by registers and/or memory)

source: www.hardwaresecrets.com
Inside a Streaming Multiprocessor

- 8 Scalar Processors (Thread Processors)
  - single-precision floating point
  - 32-bit and 64-bit integer
- 2 Special Function Units
  - sin, cos, log, exp
- Double Precision unit
- 16 kB Shared Memory

- Each Streaming Multiprocessor: up to 1,024 threads.
- GTX 280: 30 Streaming Multiprocessors
  \[ \rightarrow 30,720 \text{ concurrent threads} \]
Inside a Streaming Multiprocessor: nVidia Fermi

- 32 “cores” (thread processors) per streaming multiprocessor (SM)
- but fewer SMs per GPU: 16 (vs. 30 in GT200 architecture)
- 512 “cores” total
- “cores” now double-precision-capable

Source: nVidia Fermi White Paper
Scheduling in Batches

- In SM threads are scheduled in units of 32, called **warps**.

- **Warp**: Set of 32 parallel threads that start together at the same program address.

- For memory access warps are split into **half-warps** consisting of 16 threads.

- Warps are scheduled with zero-overhead.

- Scoreboard is used to track which warps are ready to execute.

- GTX 280: 32 warps per multiprocessor (1024 threads)

- newer cards: 48 warps per multiprocessor (1536 threads)
- **SIMT**: Single Instruction, Multiple Threads
- All threads execute the same instruction.
- Threads are split into warps by increasing thread IDs (warp 0 contains thread 0).
- At each time step scheduler selects warp ready to execute (i.e., all its data are available).
- nVidia Fermi: dual issue; issue two warps at once\(^a\)

\(^a\)no dual issue for double-precision instr.
GPUs implement **fine-grained multithreading**

But:
- Scheduling decisions here affect **entire warps**.
- GPUs have **more functional units** (scalar processors).
- Functional units cannot be scheduled arbitrarily

The above illustration is somewhat misleading in that regard.
Warps and Latency Hiding

Some runtime characteristics:

- Issuing a warp instruction takes **4 cycles** (8 scalar processors).
- Register write-read latency: **24 cycles**.
- Global (off-chip) memory access: $\approx 400$ cycles.

Threads are executed **in-order**.

- **Hide latencies** by executing other warps when one is paused.
- Need **enough warps** to fully hide latency.

*E.g.*,

- Need $24/4 = 6$ warps to hide register dependency latency.
- Need $400/4 = 100$ instructions to hide memory access latency. If every 8th instruction is a memory access, $100/8 \approx 13$ warps would be enough.
Resource Limits

Ideally: 32 warps per multiprocessor (1024 threads)

But: Various resource limits

- limited number of 32-bit registers per multiprocessor
  
  \[ E.g.: \text{11 registers per thread, 256 threads/items per work group.} \]
  
  CUDA compute capability 1.1: \text{8,192 registers per multiprocessor.}
  
  \[ \rightarrow \text{max. 2 work groups per multiprocessor} \quad (3 \times 256 \times 11 > 8192) \]

- 48 kB shared memory per multiprocessor (compute cap. 2.0)
  
  \[ E.g.: \text{12 kB per work group} \]
  
  \[ \rightarrow \text{max. 4 work groups per multiprocessor} \]

- 8 work groups per multiprocessor; max. 512 work items per work group

- Additional constraints: branch divergence, memory coalescing.

Occupyancy calculation (and choice of work group size) is complicated!
Work Groups (NVIDIA: “Blocks”)

Work Groups (on NVIDIA GTX 280):

- Work group can contain up to 512 threads
- A work group is scheduled to exactly one SM
  - Central round-robin distribution
  - Remember: Synchronization and collaboration through shared memory only within work group
- Each SM can execute up to 8 work groups
  - Actual number depends on register and shared memory usage
  - Combined shared memory usage of all work groups ≤ 16 kB

⚠ Characteristics of one particular piece of hardware, not part of the OpenCL specification!
Within a warp, all threads execute same instructions.

→ What if the code contains branches?

```c
if (i < 42)
    then_branch();
else
    else_branch();
```

- If one thread enters the branch, all threads have to execute it.
  → Effect of branch execution discarded if necessary.
  ~ Predicated execution (↗ slide 111).

- This effect is called branch divergence.

- **Worst case:** all 32 threads take a different code path.
  → Threads are effectively executed sequentially.
OpenCL Memory Model

- **Compute Device**
  - **Private Memory**
    - Work Item 1
    - Work Item 2
  - **Local Memory**
  - **Global Memory**

- **Host**
  - **Host Memory**
NVIDIA/Cuda uses a slightly different terminology:

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>Cuda</th>
</tr>
</thead>
<tbody>
<tr>
<td>private memory</td>
<td>registers</td>
</tr>
<tr>
<td>local memory</td>
<td>shared memory</td>
</tr>
<tr>
<td>global memory</td>
<td>global memory</td>
</tr>
</tbody>
</table>

On-chip memory is **significantly** faster than off-chip memory.
Like in CPU-based systems, GPUs access **global memory** in chunks (32-bit, 64-bit, or 128-bit segments).

→ Most efficient if accesses by threads in a half-warp **coalesce**.

*E.g.*, NVIDIA cards with compute capability 1.0 and 1.1:

- Coalesced access → 1 memory transaction

- Misaligned → 16 memory transactions (2 if comp. capability ≥ 1.2)
Coalescing Example

Example to demonstrate coalescing effect:

```c
__kernel void
copy (__global unsigned int *din,
     __global unsigned int *dout,
     const unsigned int offset)
{
    int i = get_global_id(0);
    dout[i] = din[i + offset];
}
```

⚠️ **Strided access** causes similar problems!
Shared memory (OpenCL: “local memory”):

- **fast** on-chip memory (few cycles latency)
- throughput: **38–44 GB/s per multiprocessor(!)**

- partitioned into **16 banks**
  - 16 threads (1 **half-warp**) can access shared memory simultaneously **if and only if** they all access a different bank.
  - Otherwise a **banking conflict** will occur.

- Conflicting accesses are **serialized**
  - (potentially significant) **performance impact**
Bank Conflicts to Shared Memory

stride width: 1 word

→ no bank conflicts

© Jens Teubner · Data Processing on Modern Hardware · Summer 2017
Bank Conflicts to Shared Memory (cont.)

Stride width: 2 words

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 8
- Thread 9
- Thread 10
- Thread 11
- Thread 12
- Thread 13
- Thread 14
- Thread 15

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 8
- Bank 9
- Bank 10
- Bank 11
- Bank 12
- Bank 13
- Bank 14
- Bank 15

→ 2-way bank conflicts

Stride width: 4 words

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 8
- Thread 9
- Thread 10
- Thread 11
- Thread 12
- Thread 13
- Thread 14
- Thread 15

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 8
- Bank 9
- Bank 10
- Bank 11
- Bank 12
- Bank 13
- Bank 14
- Bank 15

→ 4-way bank conflicts

© Jens Teubner · Data Processing on Modern Hardware · Summer 2017
Broadcast reads do not lead to a bank conflict.

- All threads must read the same word.
Thread Synchronization

Threads may use built-in functions to synchronize within work groups.

- **barrier**(*flags*) Block until all threads in the group have reached the barrier. Also enforces memory ordering.

- **mem_fence**(*flags*) Enforce memory ordering: all memory operations are committed before thread continues.

```c
for (unsigned int i = 0; i < n; i++)
{
    do_something();
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

If barrier occurs in a **branch**, same branch must be taken by **all threads** in the group (danger: deadlocks or unpredictable results).
Synchronization Across Work Groups

To synchronize across work groups,

- use **in-order** command queue and queue multiple kernel invocations from the host side
  → Can also queue **markers** and **barriers** to the command queue.

or

- use OpenCL **event mechanism**.
  → Can also synchronize host ↔ device and kernel executions in **multiple command queues**.

To wait on host side until all queued commands have been completed, use `clFinish(command queue)`. 
To summarize,

- GPUs provide **high degrees of parallelism** that can be programmed using a **high-level language**.

**But:**

- GPUs are not simply “multi-core processors.”
- Unleashing their performance requires significant efforts and great care for details.

Also note that

- GPUs provide lots of **Giga-FLOPS**.
  - But rather few applications really need raw GFLOPS.