GPU architecture and its impact on GPGPU programming

Robert Rüger

Institut für **T**heoretische **P**hysik
Goethe-Universität Frankfurt

July 4^{th}, 2012

Contents

Contents

Why use GPUs?

... how to compute faster?

- Increased clock speed?
  ... no increase in many years for physical reasons
- More complex instructions or higher instruction throughput?
  ... complicated and requires many transistors
  ... complex instructions reduce possible clock speeds
- Vectorization?
  ... cheap: SIMD = only one instruction decoder
  ... requires special programming
- Parallelization?
  ... more expensive and powerful than vectorization: MIMD!
  ... requires special programming

Why use GPUs?
... how to compute faster?

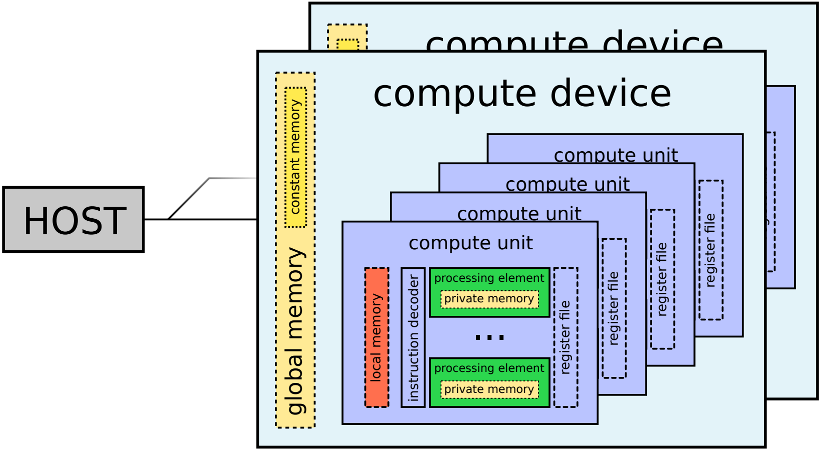
- Combination of vectorization and parallelization!
  ... compromise between cost (SIMD) and flexibility (MIMD)
- Simple cores!
  ... complicated logic (branch prediction, instruction level parallelism, etc.) would have to be replicated for every core
  ... but: other optimizations specific to data parallel calculations
- Clock speed is secondary!

This is essentially a GPU!

Contents

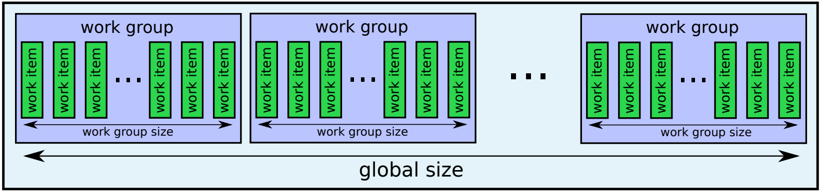
Generic GPU architecture and the OpenCL terminology

... the hardware side



Generic GPU architecture and the OpenCL terminology

... the software side



Direct correspondence to the hardware:

- work groups are assigned to the compute units
- each processing element works on one work item

Generic GPU architecture and the OpenCL terminology

... implicit vectorization: the pros and cons of SIMT

Compute units are essentially vector processors
and should be programmed like that!

Single **I**nstruction **M**ultiple **T**hread

- vectorization hidden as one thread per vector element
- allows simple conditional statements instead of masks
- branches will be executed sequentially, part of the threads NOOPs (there is only one instruction decoder!)

$\Rightarrow$ Keep implicit vector nature in mind and program accordingly!

Generic GPU architecture and the OpenCL terminology

... latency hiding: overallocation and the register file

Lets consider the global memory to be slow! Details later ...
We don't want memory accesses to stall the execution!

Latency hiding

- overallocation: work group size > #processing elements and more than 1 work group per compute unit
- large register file that stores the variables of all work items
- hardware scheduler keeps processing elements busy

In practice: amount of computation "in flight" depends on register file size and kernel register requirements
⇒ Use few registers per work item to give the scheduler freedom!

Generic GPU architecture and the OpenCL terminology

... what about synchronization?

Remember: work group size > #processing elements
Synchronization problem not present in CPU SIMD!

Synchronization among work items of the same work group?

- Yes! (barriers, memory fences, etc.)

Synchronization among different work groups?

- Rather not! Location and sequence of work group execution not specified!
- Theoretically possible via atomic operations in global memory, but generally discouraged ...

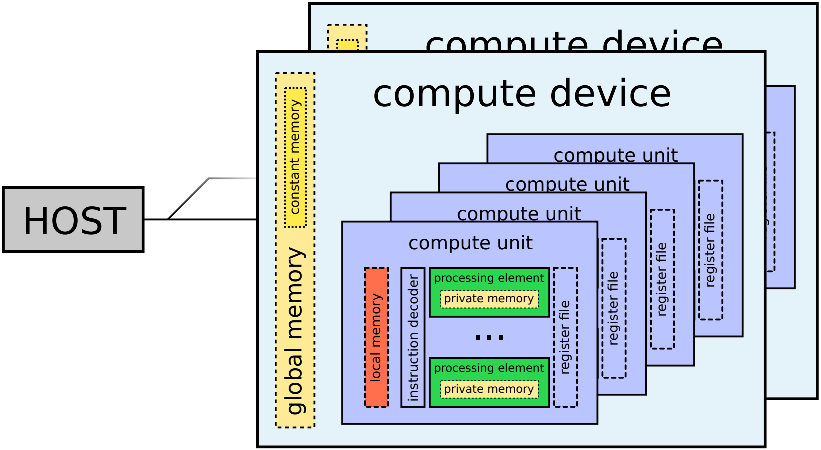
Contents

GPU memory

Warning: OpenCL only specifies the accessibility of the different memories, but not their physical location and therefore speed!

This presentation: Typical case for common GPUs ...

GPU memory



HOST
compute device
compute device
global memory
constant memory
compute unit
compute unit
compute unit
compute unit
local memory
instruction decoder
processing element
private memory
...
processing element
private memory
register file
register file
register file
register file

GPU memory

... host accessible memories

Global memory

- very large, typically gigabytes
- readable and writable by all work items
- state only well defined after kernel has finished
- slow, but much faster with streaming access (coalescing)!
- sometimes cached

Constant memory

- read-only part of the global memory (writable from host)
- often cached
- prefer constant memory for constant values ...

GPU memory

... device-only accessible memories

Local memory

- very fast on-chip memory
- shared among work items within the same work group
- versatile! (explicit global memory cache, etc.)

Private memory

- private to a single work item
- usually physically a part of the global memory, slow!
- will be used to store the work items registers if the register file is exhausted (must be avoided!)

Contents

Summary

... some rules to follow

1. keep the implicit vector nature in mind and program accordingly
2. try to use as few registers as possible
3. keep work groups independent
4. use constant memory, be creative with the local memory
5. avoid using private memory
6. global memory access should be coalesced if possible

Questions?

Thank you for your attention! :-)

Sources

- General Purpose Computing on Graphics Processing Units
  M. Bach, D. Rohr, V. Lindenstruth: HPC, WS11/12 (Univ. of Frankfurt)
  `http://compeng.uni-frankfurt.de/fileadmin/Vorlesungsmaterial/WS11-12/L11_GPGPU.pdf`
- `www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces`
- `www.codeproject.com/Articles/143395/Part-3-Work-Groups-and-Synchronization`