OpenCL

Jens Ogniewski
Information Coding Group

Outline

Introduction (incl. Motivation)

Language Overview (incl. synchronization, system model)

Parallel Programming (incl. debugging, optimization)

Motivation

- OpenCL vs Cuda
  - Cuda faster than OpenCL on NVIDIA GPUs
  - NVIDIA GPUs best GPUs for general processing

=> Use CUDA for high performance computing

=> Use OpenCL everywhere else!

Motivation

- Future portable multimedia systems (e.g. smartphone, pads)
  - More and more multicore systems
  - Including hardware accelerators
  - Enabling high processing power with low energy consumption

- OpenCL might become new standard programming language for embedded systems

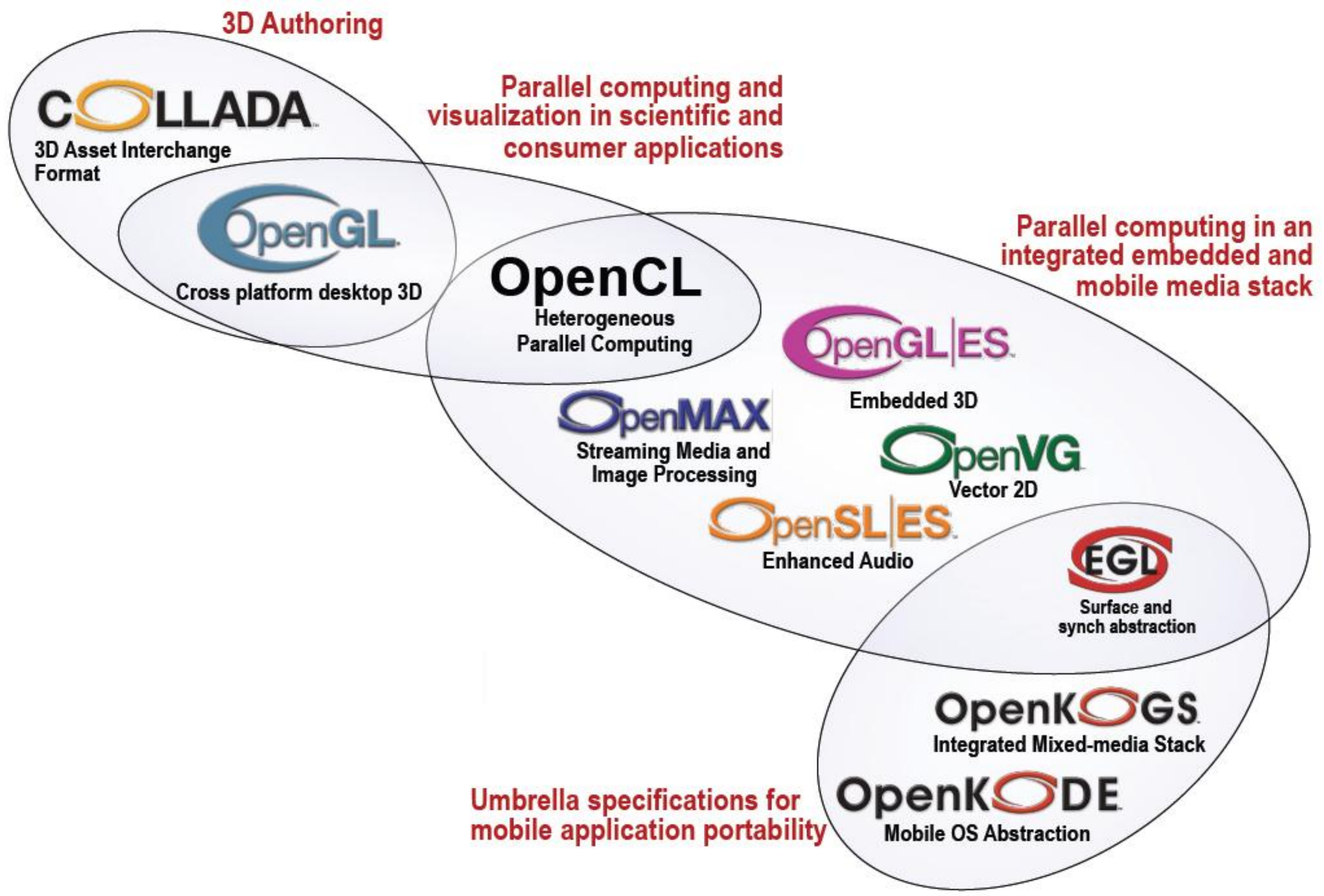
Introduction

- First suggested by Apple
- Standardized by Khronos (as OpenGL)
  - Work in Progess – still subject to changes

Introduction

KHRONOS GROUP
Over 100 companies creating authoring and acceleration standards
Board of Promoters

Introduction


3D Authoring
COLLADA
3D Asset Interchange Format
Parallel computing and visualization in scientific and consumer applications
OpenGL
Cross platform desktop 3D
OpenCL
Heterogeneous Parallel Computing
Parallel computing in an integrated embedded and mobile media stack
OpenGL|ES
Embedded 3D
OpenMAX
Streaming Media and Image Processing
OpenVG
Vector 2D
OpenSL|ES
Enhanced Audio
EGL
Surface and synch abstraction
OpenKOGS
Integrated Mixed-media Stack
OpenKODE
Mobile OS Abstraction
Umbrella specifications for mobile application portability

Introduction

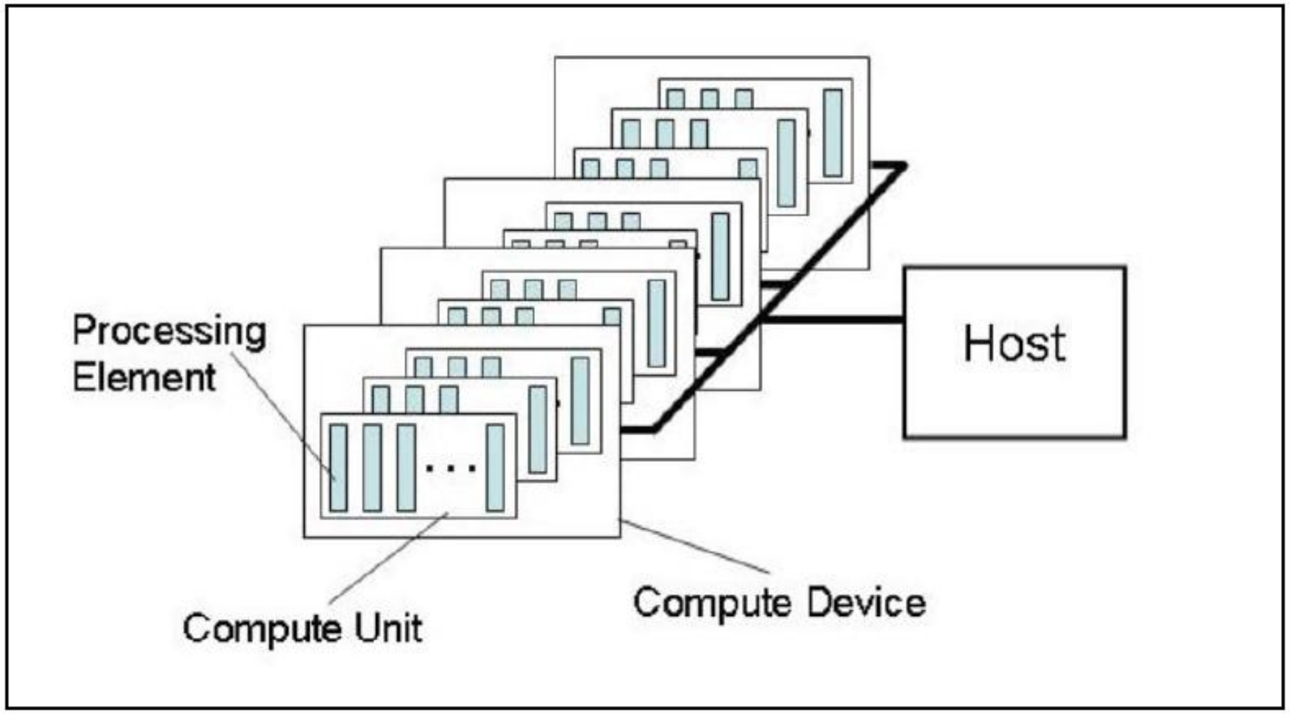
- Language aimed for parallel architectures
  - Programmer defines explicitly where and how parallelism occurs
  - Geared towards data-parallelism
    - Task-parallelism also possible
- Heterogeneous = Hardware independent (mostly)
  - Supports even hardware accelerators
  - Optimized code still needs intimidate knowledge of used architecture (GPU vs Cell vs hyperthreaded CISC)
- Basically a language to program all parallel systems

OpenCL for NVIDIA GPUs

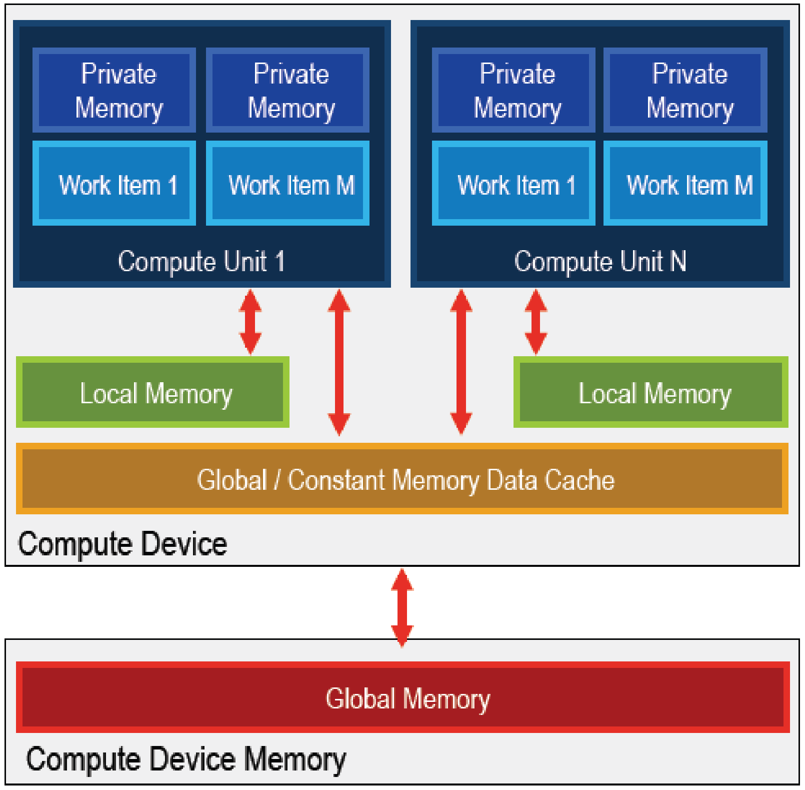
- Different terms (CUDA legacy)
  - Compute unit = multiprocessor
  - Work item = thread
  - Work group = thread block, sometimes also warp
  - Carefully: warp often used with a constant size (e.g. 1 warp = 32 threads)
  - And CUDA local memory ≠ OpenCL local memory (= CUDA shared memory)

Platform model

- One host (e.g. PC), one or several compute devices (e.g. graphic card)
  - Each compute device: one or several compute units (e.g. SIMD multiprocessor)
    - Each compute unit: one or several processing elements



Memory model model



- Private memory
  - Only accessible by one processing element (e.g. register)
- Local memory
  - Only accessible by one compute unit
- Global / Constant memory
  - Accessible by one compute device
  - Copying betw. global/constant memory and local memory has to be done explicitly!

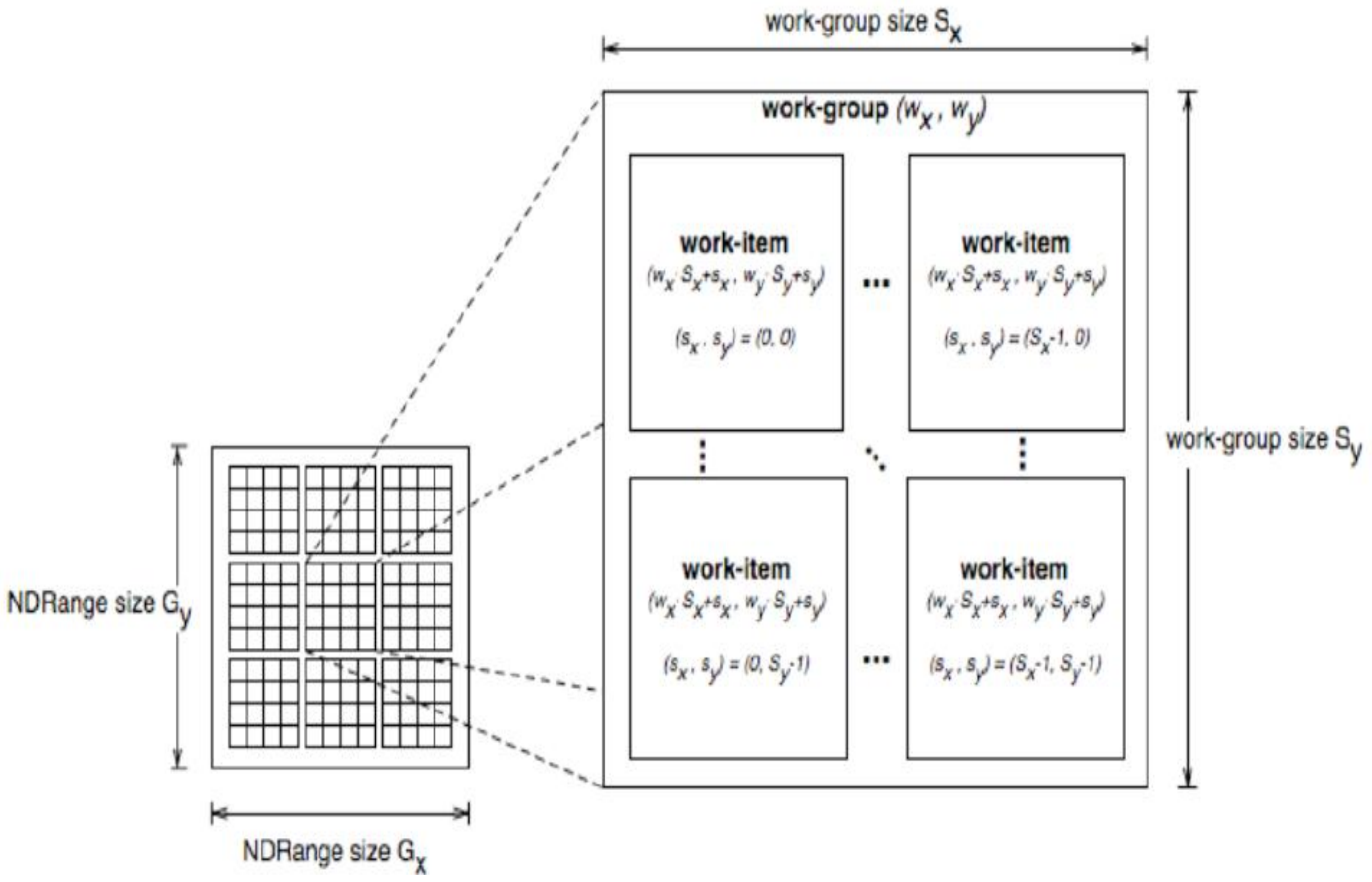
Memory model

- Global memory
  - Allocated by host
  - Passed as parameter during the kernel invocation

- Local memory
  - Have to be marked with `__local`
  - If used for arrays inside kernel: size must be known at compilation time

- Everything else will end up in private memory

Execution model


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, S_y - 1)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, S_y - 1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Execution model

- Kernel
  - (Short) function which will be executed in parallel
  - How many is determined by the global dimensions
  - Each execution is called a work item
- Several work items share one computing unit
  - Called a work group
  - How many is determined by the local dimensions
- Local and global dimensions are given by the programmer
  - Should be chosen carefully depending on application, algorithm and hardware used

Execution model

- Resident work item
  - Has reserved private memory in multiprocessor
  - Does not need to be active, but can become active anytime
- Active work item
  - Executes instruction
- Thread scheduler
  - Can swap out / in resident work items without any overhead
  - Tries to swap out work items waiting for memory access and run other resident work items instead
  - Tries to coalesce memory access inside a workgroup

Synchronization

- Memory
  - Explicit data movement between host and compute device memory
  - Explicit data movement between global/constant and local memory on the device
  - Global/constant memory can be “linked” to host memory during creation

- Task synchronization: only inside a workgroup
  - Using `barrier(CLK_LOCAL_MEM_FENCE)`
  - Execution continues only after all items in the work group passed this command and wrote back all changes from private memory to local or global memory

- No synchronization between workgroups!

Synchronization

- Barrier command
  - Works only inside a workgroup
  - The programmer can optionally select which memory(s) should be synchronized (e.g. `CLK_LOCAL_MEM_FENCE` for local memory)
  - After encountering a barrier, the work item makes sure that all its write accesses to these memory(s) have finished, and then waits until all other work items in the work group has done the same
  - The barrier has to be passed by all work items of the work group
    => don’t put a barrier inside part of your code which cannot be reached by all work items (e.g. inside an if branch) or your program will stall!
  - Synchronization for global memory is possible, but should be avoided due to its long access time. If you are sharing variables between different work items you should put it in local memory anyway to speed-up your kernel.

Synchronization

- Between host and compute device(s):
  - Using queues
  - Available for tasks (e.g. `clEnqueueNDRangeKernel`)
  - Memory (e.g. `clEnqueueReadBuffer`)
  - And events (e.g. `clWaitforEvents`)
- Queues can be in-order or out-of-order
- Several queues in parallel possible, programmer has to take care of synchronization however
- Queue system enables task-parallism
  - e.g. CPU can work on a task while waiting for the results of the GPU

Synchronization

- Since aimed towards heterogenous systems
  - Functions exist to specify where tasks should be executed
  - Need to be called before a task can be executed
  - Basically return a pointer to the device
  - Also, functions exists to determine how many devices are available, and what capability each device has
  - Tasks are compiled at runtime

=> More overhead as CUDA, but that's the price for increased flexibilty

The Language

- Based on C99, but:
  - No function pointers (will come later?)
  - No pointers to pointers in function calls (=> no multidimensional arrays)
  - No recursion
  - No arrays with dynamical length
  - No bitfields
  - Also, no possibility to call a kernel from another kernel
- Optional:
  - Pointers with length <32 bit
  - Writing support for 3D images
  - Double and half types
  - Atomic functions

The Language

- But instead:
  - Integrated functions for reading / writing 2D images and reading 3D images
  - Converting functions incl. explicit rounding and saturation
  - math.h, all functions with different precisions
  - Vector support (2-, 3- and 4-dimensional)

- Available primitive datatypes:
  - Bool, char, int, long, float, size_t, void, unsigned versions as well

- Mix of OpenCL and OpenGL possible
  - Can share data structures and variables (without copying)
  - API functions available

The Language

- Pointers: only useable as kernel arguments

- Inside kernel: no pointers, no memory management functions, the size of all data has to be known at compilation time
  - Saves overhead of a stack

- Workaround
  - Put placeholders for the array sizes in your kernel code
  - During runtime: replace the placeholders with the values you want, before kernel compilation

Example Architecture

- NVIDIA GTS 250 (found in Olympen)
  - 8192 registers / compute unit
  - 16 kb local memory / compute unit
  - 64 kb constant memory (varying global memory size)
  - Max. 16 kbytes private memory / working item
  - Local memory access time: 24 cycles
  - Global memory access time: 400-600 cycles
  - Kernel size limit: 2 million PTX instructions
  - 8 processing elements / compute unit
  - 16 compute units
  - Max. 768 resident work items per compute unit
  - Max. 512 work items / work group
  - Support for atomic functions on 32byte words in global memory

Parallel Programming – Practical Issues

- When to use GPGPU?
  - Parallel algorithm
  - Instruction mix: little memory access, less branches, much computation

  Or if your algorithm can be rewritten to fulfill those criteria without introducing much overhead (little is ok)!

Parallel Programming – Practical Issues

- But how to do that?
  - Find dataparallism
  - Works often: look for loop which doesn't have dependencies between iterations (or which can be rewritten to fulfill this criteria)
  - Dimension of the loop can be used as the global work group size
    - e.g. in image processing the x size is the width and the y size the height of the image

Parallel Programming – Practical Issues

Example:

```
for (i=0; i<n; i++) {
   a[i]=a[i]+b[i]*c[i];
}
```

Trivial solution:

```
__kernel void task1(..)
   a[get_global_id(0)]=
        a[get_global_id(0)]
       +b[get_global_id(0)]
       *c[get_global_id(0)];
}
```

Parallel Programming – Practical Issues

Example:

```
for (i=0; i<n; i++) {
    a[i]=a[i-1]+b[i]*c[i];
}
```

No trivial solution:

- Data dependency to earlier iteration of the loop
- Parallelization difficult, if possible at all

Parallel Programming – Practical Issues

Example:

```
for (i=0; i<n; i++) {
    a[i]=a[i+1]+b[i]*c[i];
}
```

Problem:

- A thread with a higher id may write back his result to ‘a’ before a thread with a lower id read its value from ‘a’ (e.g. thread 3 writes its value back to ‘a’ before thread 2 could read its value from ‘a’)
- Solution: see next slide

Parallel Programming – Practical Issues

Solution: write the output to a different array

```
for (i=0; i<n; i++) {
   d[i]=a[i+1]+b[i]*c[i];
}
```

Trivially:

```
__kernel void task1(..)
   d[get_global_id(0)]=
        a[get_global_id(0)+1]
       +b[get_global_id(0)]
       *c[get_global_id(0)];
}
```

Parallel Programming – Optimization Issues

- Avoid:
  - Branches
  - Double precision (ok for new graphic cards)
  - Memory access: recomputation might lead to faster results
  - Memory bank conflicts
  - Private memory in global memory, fit it into the register file
  - Barrier(), use auto-synchronization techniques if possible
  - Atomic commands if possible

Parallel Programming – Optimization Issues

- What else?
  - Use vector intrinsics: to make sure that SIMDs are used correctly and most efficiently
  - Coalescing memory: enforce aligned memory access
  - Prefer constant to global memory, since it is cached
  - Use local memory as buffer for global memory
  - try to have at least 192 resident work items per compute unit to hide memory accesses
  - Try to reuse kernels as much as possible (avoid compilation)
  - For filters: use similar block sizes in all dimensions to minimize memory access
  - Use low precision functions if possible
  - Use the inbuilt interpolation, but beware of its precision
  - Try to replace if branches by min / max functions
  - Don't forget the CPU – let it work, too!

Parallel Programming – Debugging Issues

- Current Lab environment: Unfortunately limited debugging capability
  - No printout available (if running on GPU)
  - Only few error messages
  - “Black flash”: program crashed
  - Errors in pixels on the display: wrong memory usage, will need to restart eventually
  - Very fast runtime: probably wrong work group size, most likely too big (might even be that the kernel is too big or that it uses too much memory for this work group size)
  - Note: even if you get the correct result back, the execution might have failed (the results might belong to an earlier run of the program)

Parallel Programming – Practical Issues

- Finally: Don't be anxious if your speedup isn’t that high
  - Many research projects reports very high speedups (100 times and more) if using the GPU
  - In reality might be less:
    - Copying between CPU and GPU neglected in report
    - Or serial part of the application ignored
    - Often even comparing GPU to unoptimized CPU code

- Still, you should get a much higher speed for most applications, 4 times (and more) are realistic

Which is great!

Further readings

- Open CL at Khronos

  http://www.khronos.org/developers/library/overview/opencl_overview.pdf

  http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

- NVIDIA and OpenCL

  http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf

  http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

Questions?

Thank you very much!

www.liu.se