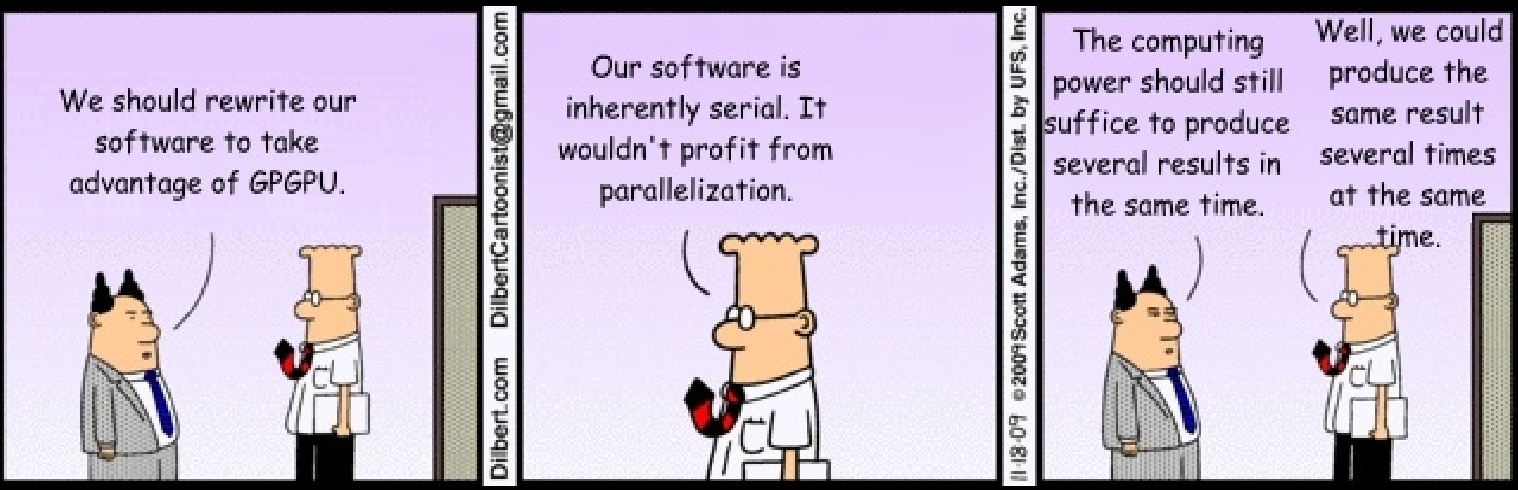
Open CL



Jens Ogniewski
Information Coding Group

Questions

Which types of synchronization does OpenCL offer? Wich not?

What is memory coalescing? How can it be achieved?

What characteristics must a function / an algorithm have to be suitable as an OpenCL kernel?

Outline

Introduction

Language Overview

Open CL for NVIDIA GPUs

Dos and donts

An example

Introduction

- Language aimed for parallel architectures
  - Programmer defines explicitly where and how parallelism occurs
  - Aimed towards SIMD processing
  - Task parallelism?
- Heterogeneous = Hardware independent (mostly)
  - Optimized code still needs intimidate knowledge of used architecture (GPU vs Cell vs hyperthreaded CISC)
- Based on C99 (with modifications)
- First initiated by Apple, still subject to changes

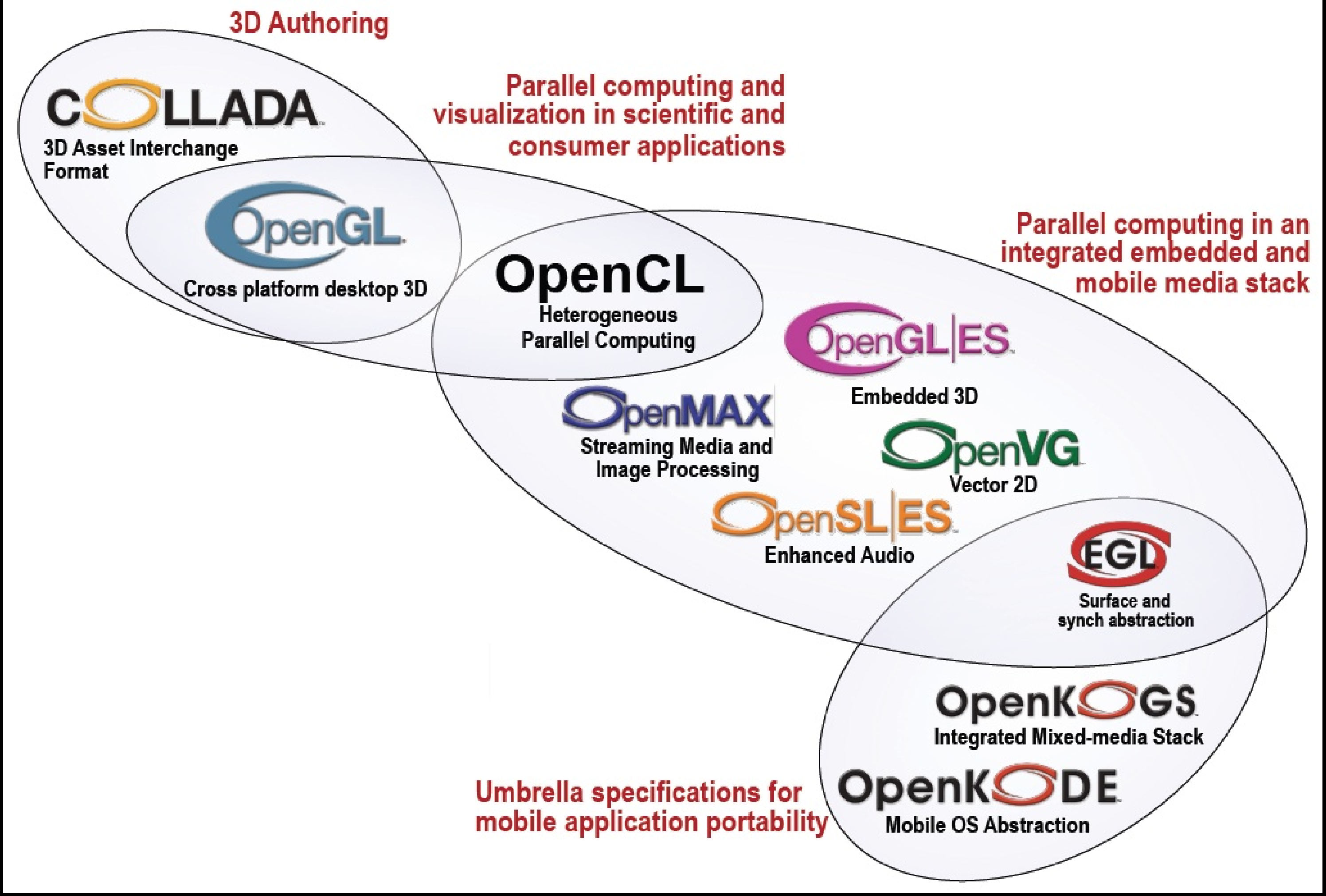
Introduction

KHRONOS GROUP

Over 100 companies creating authoring and acceleration standards

Board of Promoters

Introduction



The Language

- Based on C99, but:
  - No function pointers (will come later?)
  - No pointers to pointers in function calls (=> no multidimensional arrays, could use image types instead of)
  - No recursion
  - No arrays with dynamical length
  - No bitfields
- Optional:
  - Pointers with length <32 bit
  - Writing support for 3D images
  - Double and half types
  - Atomic functions

The Language

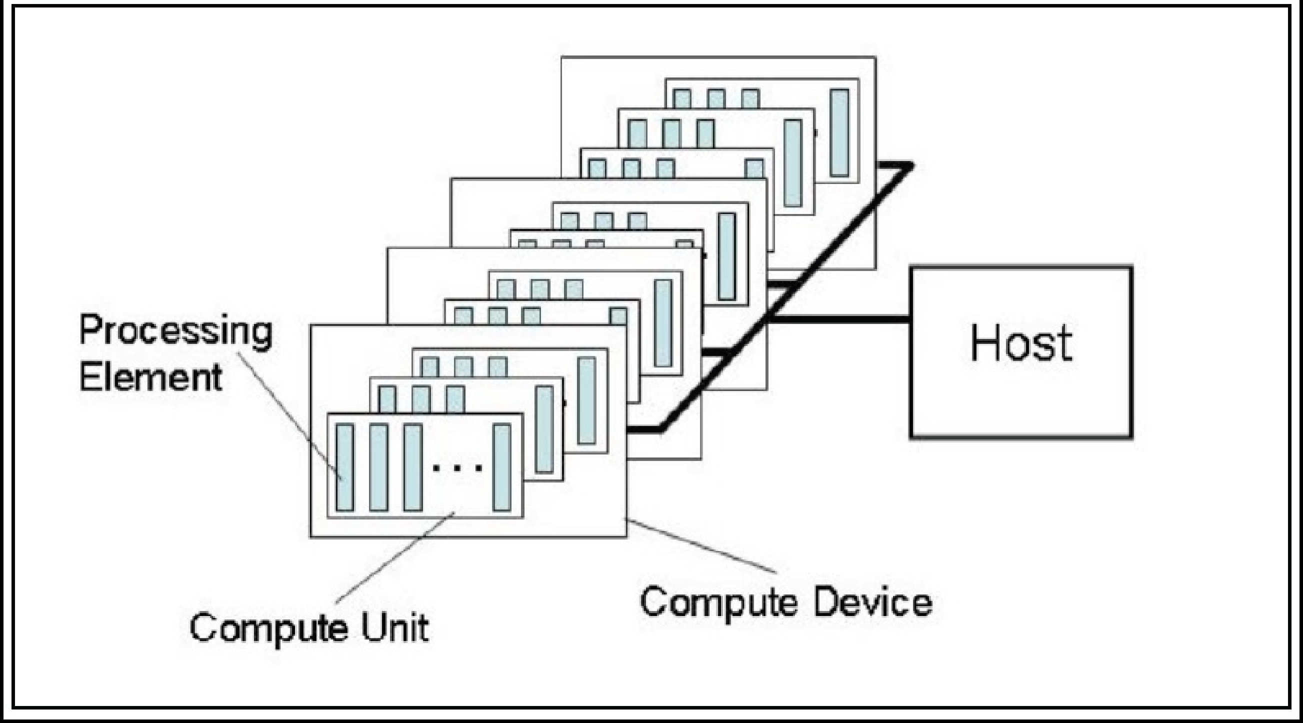
- But instead:
  - Integrated functions for reading / writing 2D images and reading 3D images
  - Converting functions incl. explicit rounding and saturation
  - math.h, all functions with different precisions
  - Vector support (2-, 3- and 4-dimensional)

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

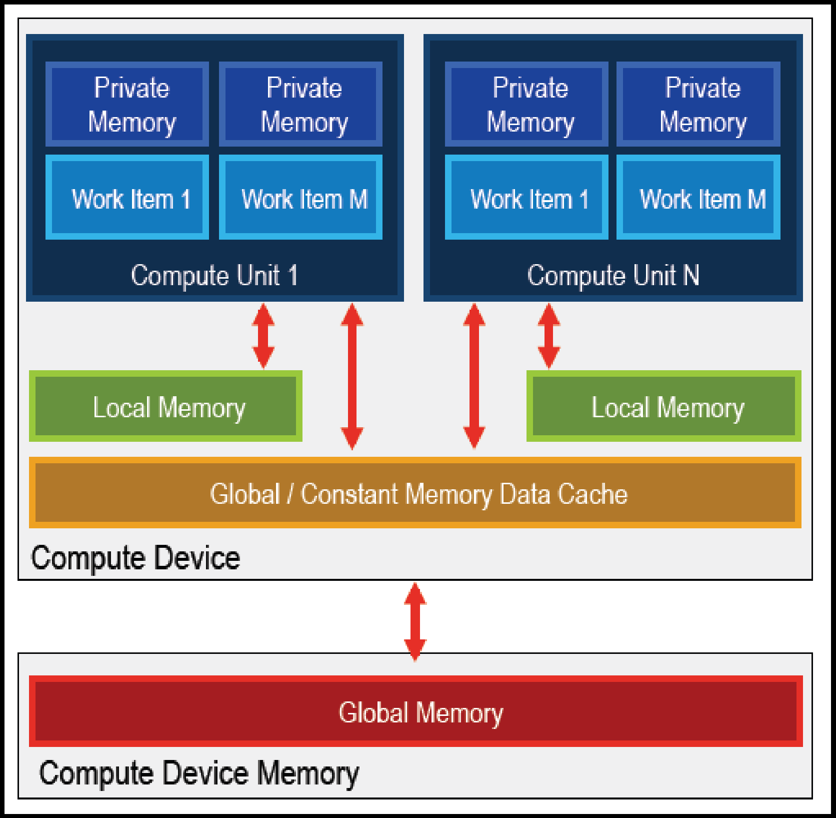
- Mix of OpenCL and OpenGL possible
  - Can share data structures and variables (without copying)
  - API functions available

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. shader)
    - Each compute device: one or several processing elements



Memory model model

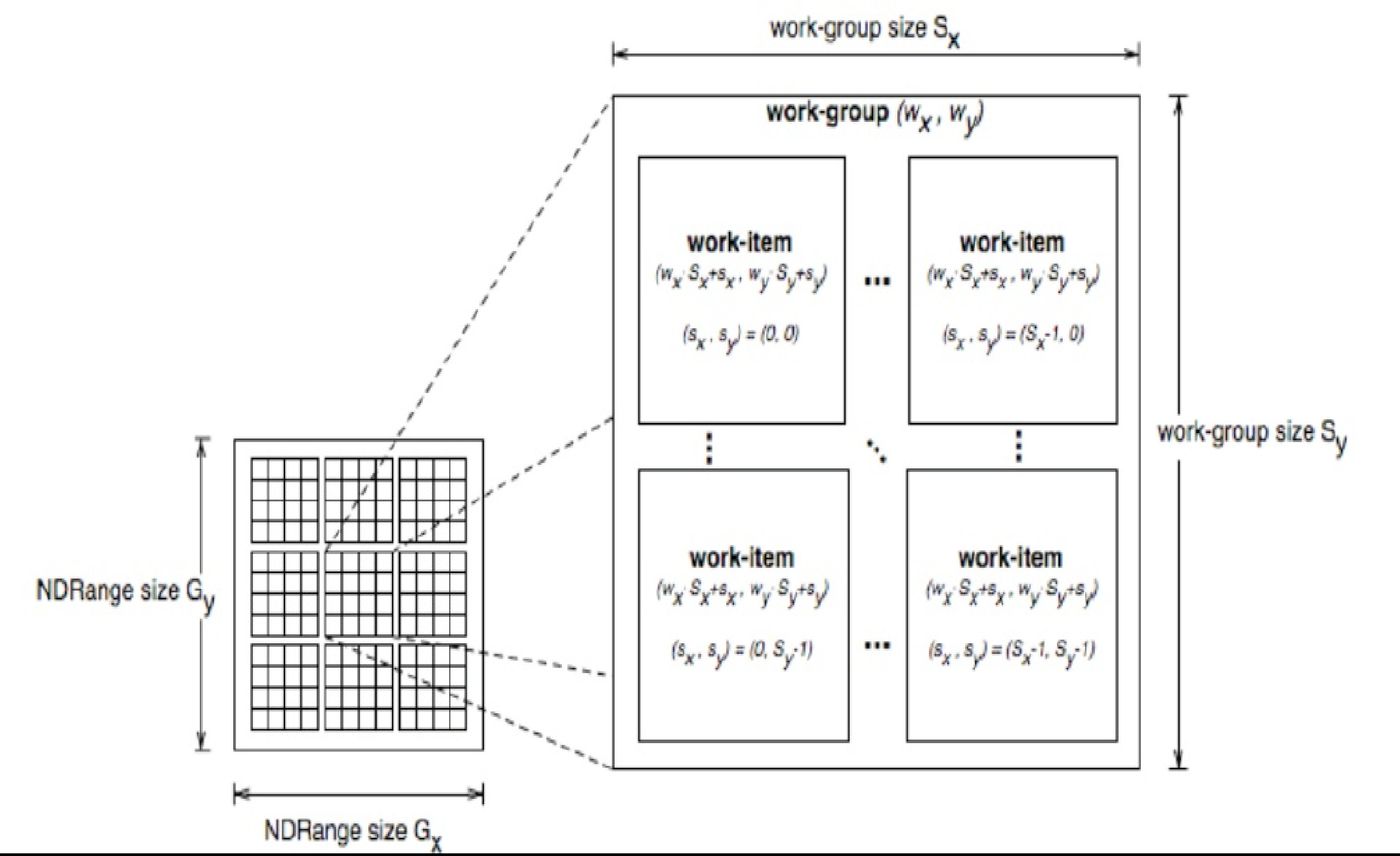


- Private memory
  - Only accessible by one processing element (e.g. register)
- Local memory
  - Only accessible by one compute unit
  - Copying betw. global/constant memory and local memory has to be done explicitly!
- Global / Constant memory
  - Accessible by one compute device
  - Also used by compiler for variables that don't fit into private memory

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 compute 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



Execution model

- How does that work?
  - Functions which return position of work-item in workgroup or globally or position of workgroup in global dimensions
  - Can be used to calculate offset

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 workgroup passed this command and wrote back all changes from private memory to local or global memory
  - All work items have to pass the barrier => not allowed inside branches!
- No synchronization between workgroups!

Synchronization

- Between host and compute device(s):
  - Using queues
  - Available for tasks (`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

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)
  - CUDA local memory: in global memory (variables that don't fit in the register file)
  - Global memory, constant memory: same in CUDA and OpenCL

OpenCL for NVIDIA GPUs

- 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

OpenCL for NVIDIA GPUs

- Example architecture
  - 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
  - Max. 768 resident work items per compute unit
  - Max. 512 work items / work group

  - Support for atomic functions on 32byte words in global memory

- NVIDIA NVS 290 (your machine): 2 compute units

OpenCL for NVIDIA GPUs

- Example architecture
  - 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
  - Max. 768 resident work items per compute unit
  - Max. 512 work items / work group

  - Support for atomic functions on 32byte words in global memory

- NVIDIA GTS 250 (Olympen): 16 compute units

Dos & Donts

- When to use GPGPU?
  - Parallel algorithm
  - Instruction mix: little memory access, much computation
  - Data, not task parallel

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

Dos & Donts

- Avoid:
  - Branches
  - Double precision (at least for now)
  - Memory access: recomputation might lead to faster results
  - Memory bank conflicts (betw. work groups)
  - Private memory in global memory

Dos & Donts

- What else?
  - Use vector intrinsics: to make sure that SIMDs are used correctly and most efficiently
  - Coalescing memory: enforce memory access aligned by 16 words for 16 work items (64 and 128 bytes optimal)
  - Prefer constant to global memory, since it is cached
  - Use local memory as buffer for global memory
  - 16 or 32 work items per work group optimal
  - try to have more than 192 resident work items / compute unit to hide memory accesses
  - Use auto synchronization to avoid barriers
  - Try to reuse kernels as much as possible => compilation expensive
  - Watchdog timer: might disrupt computation

An example

An example

An example: c function

```
for (i=0;i<height*width*4;i++) {
   int tmp = (*(imageData+i))*(*(lightData+i))/128;
   if (tmp>255) tmp = 255;
   *(outputImage+i) = (unsigned char)tmp;
}
```

An example: kernel function

```
imageShader( __global unsigned char *i_image,
      __global unsigned char *i_lightmap,
      __global unsigned char *o_image)
{
   int i = get_global_id(0);

   o_image[i] = convert_uchar_sat(i_image[i]
         * i_lightmap[i]/128);
}
```

An example: init function (1/2)

```
cl_context cxGPUContext;
cl_command_queue commandQueue;
cl_kernel gpgpuKernel;

int init_OpenCL(const char* source_path) {
   cl_int ciErrNum = CL_SUCCESS;
   char *source;
   cl_device_id device;
   size_t program_length;

   cxGPUContext = clCreateContextFromType(0,
      CL_DEVICE_TYPE_GPU, NULL, NULL, &ciErrNum);
```

An example: init function (2/2)

```
device = oclGetDev(cxGPUContext, 0);
commandQueue = clCreateCommandQueue(cxGPUContext,
   device, 0, &ciErrNum);
source = oclLoadProgSource(source_path,
   "// No header needed!\n", &program_length);
cl_program cpProgram =
   clCreateProgramWithSource(cxGPUContext, 1,
   (const char **)&source, &program_length,
   &ciErrNum);
ciErrNum = clBuildProgram(cpProgram, 0, NULL, NULL,
   NULL, NULL);
gpgpuKernel = clCreateKernel(cpProgram, "imageShader",
   &ciErrNum);
free(source);
}
```

An example: kernel call (1/2)

```
i_image = clCreateBuffer(cxGPUContext,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, width *
   height * 4 * sizeof(cl_char), image, &ciErrNum);
i_lightmap = clCreateBuffer(cxGPUContext,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, width *
   height * 4 * sizeof(cl_char), lightmap, &ciErrNum);
o_image = clCreateBuffer(cxGPUContext,
   CL_MEM_READ_WRITE, width * height * 4 *
   sizeof(cl_char), NULL, &ciErrNum);
clSetKernelArg(gpgpuKernel, 0, sizeof(cl_mem),
   (void *) &i_image);
clSetKernelArg(gpgpuKernel, 1, sizeof(cl_mem),
   (void *) &i_lightmap);
clSetKernelArg(gpgpuKernel, 2, sizeof(cl_mem),
   (void *) &o_image);
```

An example: kernel call (2/2)

```
localWorkSize[0] = 1;
globalWorkSize[0] = width * height * 4;
ciErrNum = clEnqueueNDRangeKernel(commandQueue,
   gpgpuKernel, 1, NULL, globalWorkSize,
   localWorkSize, 0, NULL, NULL);
clEnqueueReadBuffer(commandQueue, o_image, CL_TRUE, 0,
   width * height * 4 * sizeof(cl_char), result, 0,
   NULL, NULL);

clReleaseMemObject(i_image);
clReleaseMemObject(i_lightmap);
clReleaseMemObject(o_image);
```

Results

- CPU: 18783 us
- GPU: 32946 us

Results

```
localWorkSize[0] = 16;
globalWorkSize[0] = width * height * 4;
ciErrNum = clEnqueueNDRangeKernel(commandQueue,
   gpgpuKernel, 1, NULL, globalWorkSize,
   localWorkSize, 0, NULL, NULL);
```

- CPU: 18783 us
- GPU, optimized: 15776 us

Why? Coalesced memory!

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

  http://www.khronos.org/opencl/sdk/1.0/docs/man/xhtml/

Further readings

- 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