



Information Coding / Computer Graphics, ISY, LiTH

# **Lecture 13**

## **OpenCL**

**GPU computing with GLSL**

**OpenGL Compute shaders**



## Lecture questions

- 1) What kind of devices will OpenCL run on?
- 2) What does an OpenCL work group correspond to in CUDA?
- 3) What geometry is typically used for shader-based GPU computing?
- 4) Are scatter or gather operations preferable?  
Why?



Information Coding / Computer Graphics, ISY, LiTH

**Just one more thing...**

**Extensions to CUDA**

**Libraries: cuFFT, cuBLAS...**

**Thrust**

**and others**



# Thrust

## Template library

Templates for common operations

Can simplify your code



### Data Structures

- `thrust::device_vector`
- `thrust::host_vector`
- `thrust::device_ptr`
- Etc.

### Algorithms

- `thrust::sort`
- `thrust::reduce`
- `thrust::exclusive_scan`
- Etc.



## Information Coding / Computer Graphics, ISY, LiTH

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```



Information Coding / Computer Graphics, ISY, LiTH

# **Get pre-optimized implementations for critical standard operations**

**Performance analysis**

**Highly serial summation pararellized by  
reduction - not a "GPU perfect" case!**

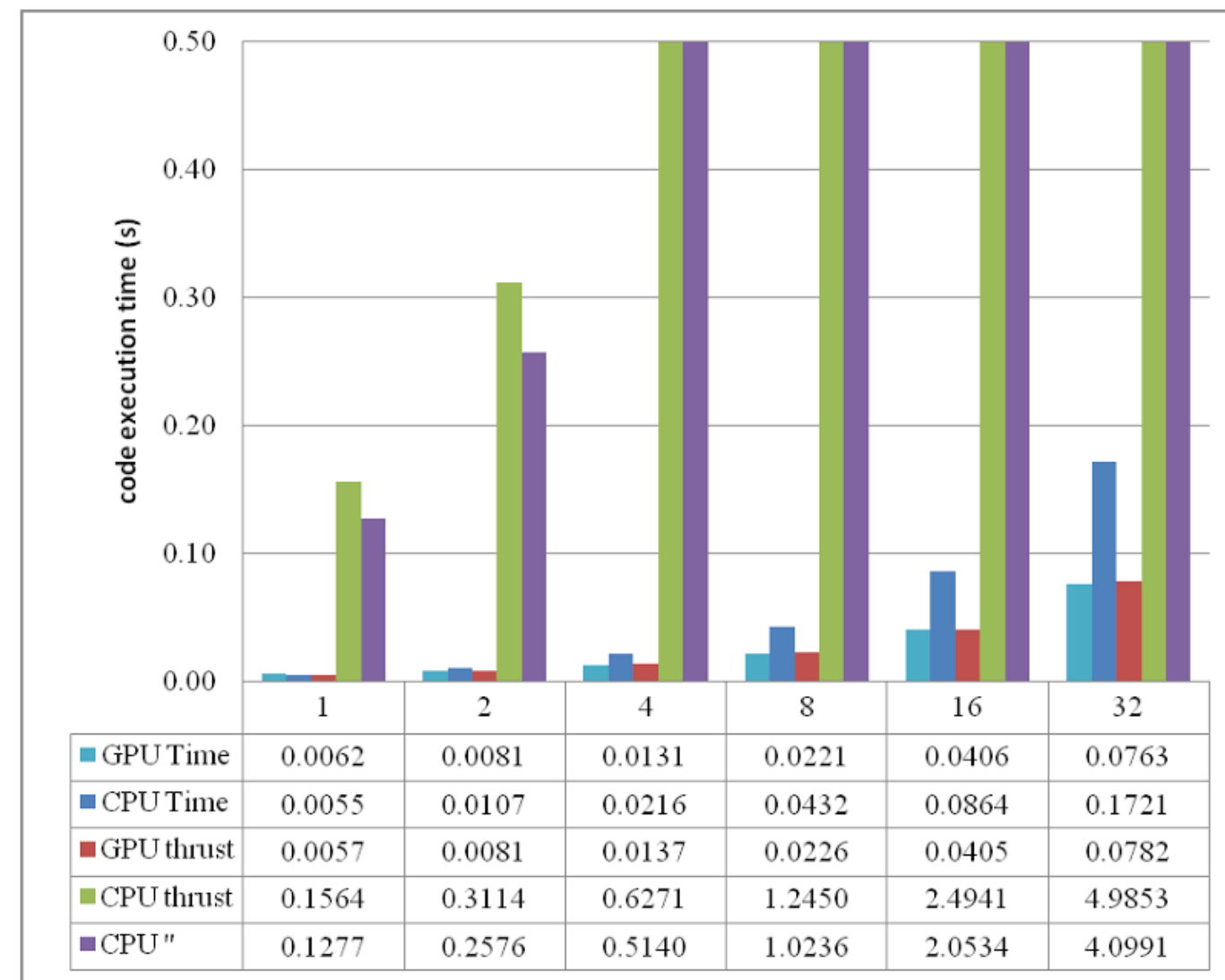


## Performance analysis

**GPU code: Thrust as fast as optimized CUDA code, but simpler!**

**CPU code: Don't touch Thrust for this!**

**(But this was 5 years back - things do change.)**





Information Coding / Computer Graphics, ISY, LiTH

**And that is just *one* alternative way to  
access one particular API...**

**So let's have a look at the alternative APIs as well!**





Information Coding / Computer Graphics, ISY, LiTH

# Introduction to OpenCL

## Open Compute Language





T!

M?

PEL!



**Zlatan efter nya succén:**

**VI ÄR  
REDDO  
FÖR CL**



## Information Coding / Computer Graphics, ISY, LiTH

- **Motivation**
- **Overview**
- **Examples**
- **Performance comparison**





Information Coding / Computer Graphics, ISY, LiTH

# **Origins of OpenCL**

**Initiated by Apple**

**Managed by Khronos group**

**Many supporting parties**

**Many providers**



# Information Coding / Computer Graphics, ISY, LiTH





## Why?

- **The market could not let CUDA rule the world**
  - **Support for other platforms**
    - **Open standard**
  - **Similarity with OpenGL**

**For programming "all" parallel architectures**



Information Coding / Computer Graphics, ISY, LiTH

## **Supported architectures (not complete!)**

**GPU**

**Intel compatible CPUs**

**ARM**

**FPGA**

**CELL**

**Intel Xeon Phi**

**Who decides? Any company making its own OpenCL implementation!**



Information Coding / Computer Graphics, ISY, LiTH

**”Open”?**

**Means *open specification***

**Like OpenGL**

**Many providers making their own  
implementation**

**There is not *one* OpenCL library.**





Information Coding / Computer Graphics, ISY, LiTH

## **No free lunch**

**Model does not fit all architectures**

**One size fits all - platform dependent  
optimizations hard to do**



Information Coding / Computer Graphics, ISY, LiTH

# **OpenCL for GPU Computing**

**Mostly similar to CUDA both in architecture and performance!**

**Messy setup - but you get used to it**

**Kernels similar to CUDA**

**Easier for NVidia to be first with new features**



# OpenCL vs CUDA terminology

## OpenCL

compute unit  
work item  
work group  
local memory  
private memory

## CUDA

multiprocessor (SM)  
thread  
block  
shared memory  
registers

And CUDA local memory =?  
OpenCL local memory (= CUDA shared memory)

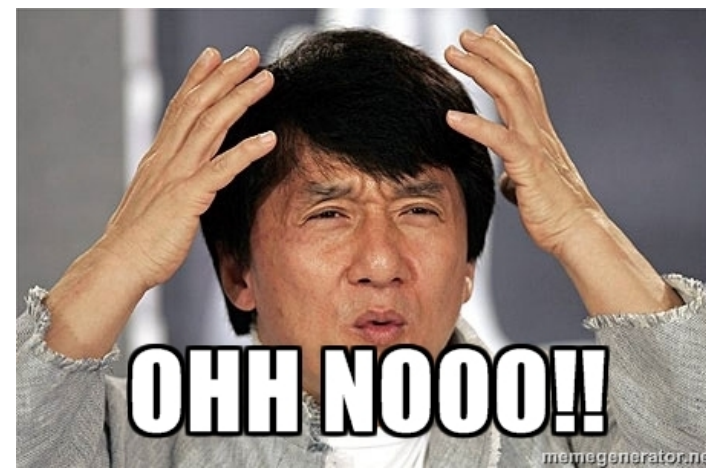


## Oh, that "local memory"...

CUDA local memory = global memory accessible *only by one thread* (like registers but slower)

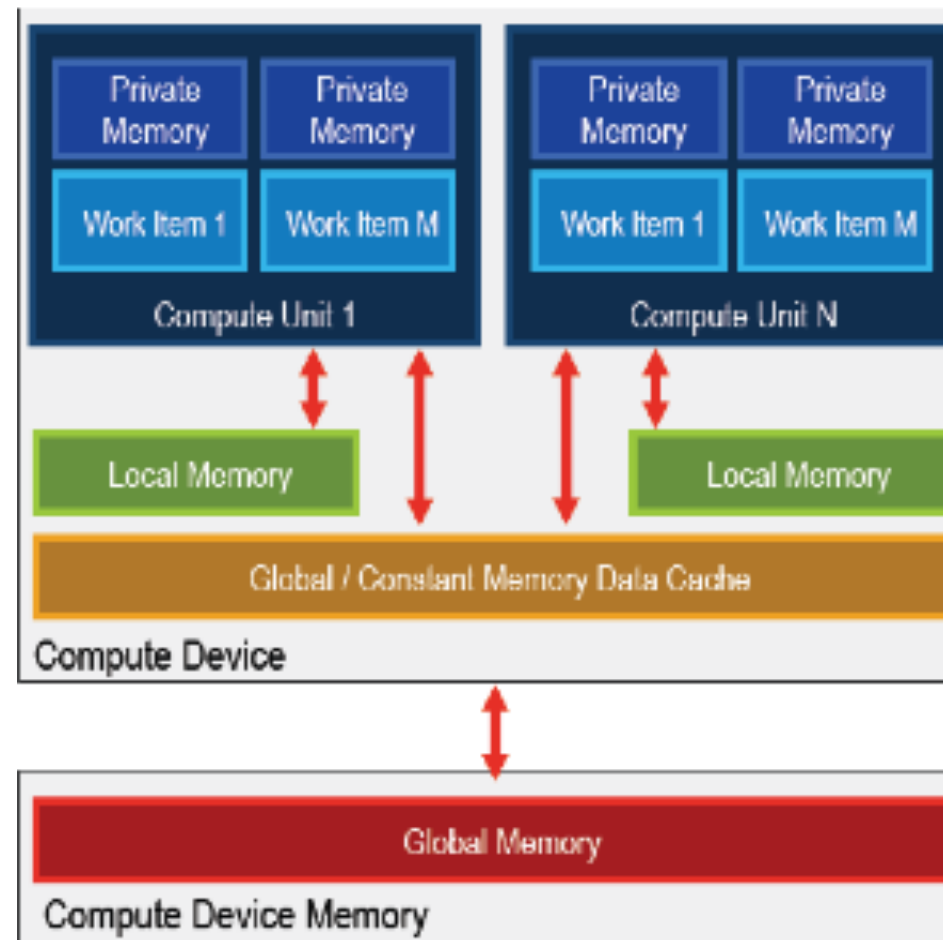
CUDA shared memory = OpenCL local memory = memory local inside the SM, shared within block

Anyone else who thinks this makes sense?





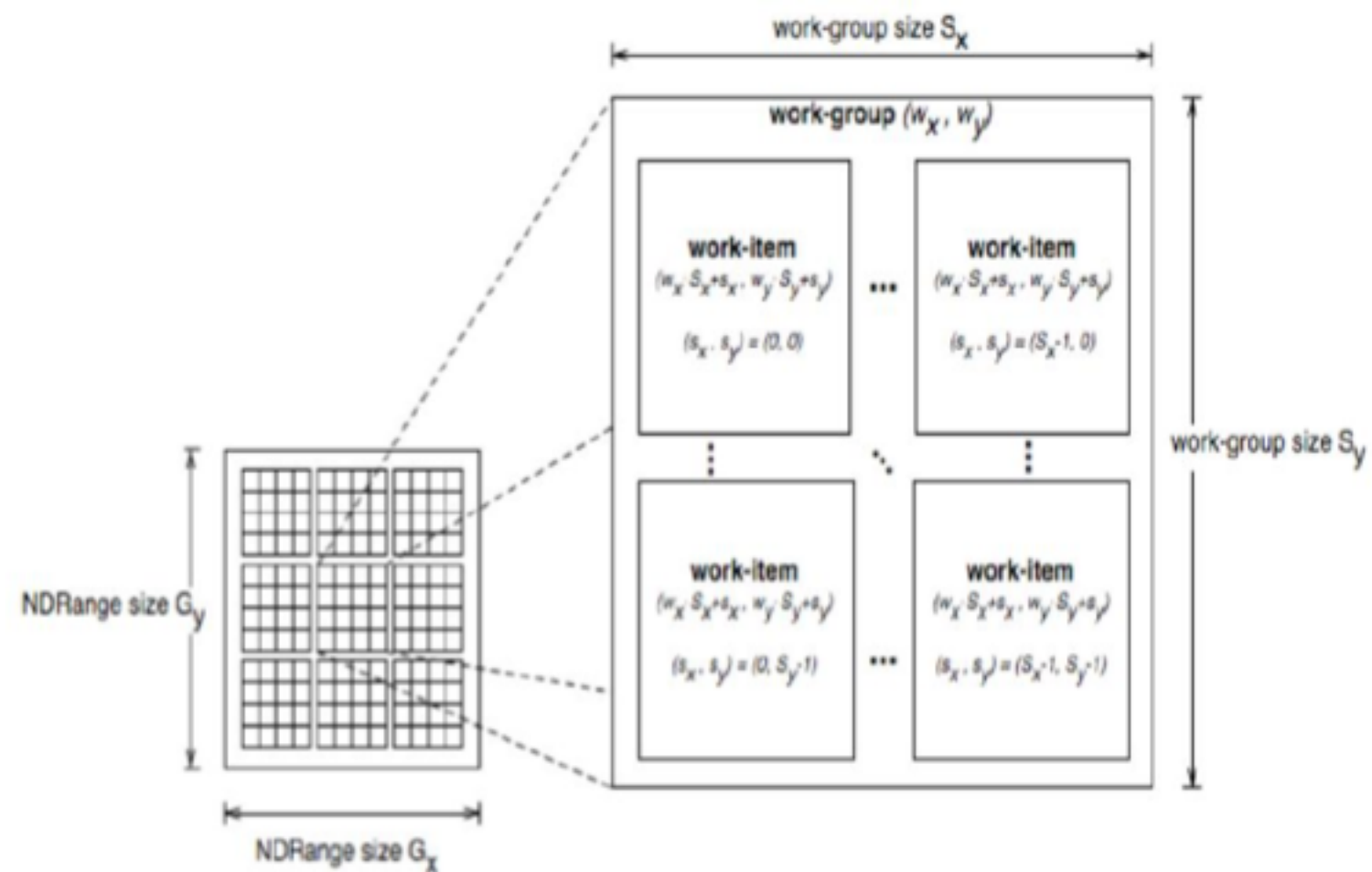
## OpenCL memory model



Been there, done that...



## OpenCL execution model



**Anyone who see "blocks" and "threads"?**



# Synchronization

**Kernels can synchronize within a work group:**

```
barrier(CLK_LOCAL_MEM_FENCE)
```

**No synchronization between work groups. (Do you remember why?)**

**Synchronizes memory access. You choose which kind of memory access to synchronize (global, local).**



# Synchronization

**The host (CPU) can synchronize on global level:**

Available for:

tasks (e.g. `clEnqueueNDRangeKernel`)

Memory (e.g. `clEnqueueReadBuffer`)

events (e.g. `clWaitforEvents`)





Information Coding / Computer Graphics, ISY, LiTH

# **Heterogenous**

**Some differences from CUDA: Designed for heterogenous systems!**

**Several devices may be active at once**

**You can specify which device to launch a task to**

**Query devices and device characteristics**

**Some overhead compared to CUDA, and the reward is flexibility!**



## Language

Based on C99, but:

- ❑ No function pointers
- ❑ No pointers to pointers in function calls  
(=> no multi-dimensional 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



## On the positive side:

- ❑ 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

### Mix of OpenCL and OpenGL possible

- ❑ Can share data structures and variables (without copying)
  - ❑ API functions available



## **How about that setup?**

- 1) Get a list of platforms**
- 2) Choose a platform**
- 3) Get a list of devices**
- 4) Choose a device**
- 5) Create a context**
- 6) Load and compile kernel code**



## **Then we can start working**

**7) Allocate memory**

**8) Copy data to device**

**9) Run kernel**

**10) Wait for kernel to complete**

**11) Read data from device**

**12) Free resources**



## 1-5: Where to run

**Simplified here - might fail!**

```
cl_platform_id platform;  
unsigned int no_plat;  
err = clGetPlatformIDs(1,&platform,&no_plat);
```

```
// Where to run  
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);  
if (err != CL_SUCCESS) return -1;
```

**Context**

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);  
if (!context) return -1;  
commands = clCreateCommandQueue(context, device_id, 0, &err);  
if (!commands) return -1;
```



## 6: Kernel

```
// What to run
program =
clCreateProgramWithSource(context, 1,
(const char **) & KernelSource, NULL,
&err);
if (!program) return -1;

err = clBuildProgram(program, 0, NULL,
NULL, NULL, NULL);
if (err != CL_SUCCESS) return -1;
kernel = clCreateKernel(program, "hello",
&err);
if (!kernel || err != CL_SUCCESS) return -1;
```

```
const char *KernelSource = "\n" \
"__kernel void hello(      \n" \
"  __global char* a,      \n" \
"  __global char* b,      \n" \
"  __global char* c,      \n" \
"  const unsigned int count) \n" \
"{                          \n" \
"  int i = get_global_id(0); \n" \
"  if(i < count)           \n" \
"    c[i] = a[i] + b[i];   \n" \
"}                          \n" \
"\n";
```

**Most programs also load kernels from files**



## 7-8: Get the data in there

```
// Create space for data and copy a and b to device (note that we could also use
clEnqueueWriteBuffer to upload)
input = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
sizeof(char) * DATA_SIZE, a, NULL);
input2 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
sizeof(char) * DATA_SIZE, b, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(char) * DATA_SIZE,
NULL, NULL);
if (!input || !output) return -1;

// Send data
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &input2);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
if (err != CL_SUCCESS) return -1;
```





## 9-10: Run kernel, wait for completion

```
// Run kernel!  
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global,  
&local, 0, NULL, NULL);  
  
if (err != CL_SUCCESS) return -1;  
  
clFinish(commands);
```



## 11-12: Read back data, release

```
// Read result
err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0, sizeof(char) * count,
c, 0, NULL, NULL );
if (err != CL_SUCCESS) return -1;

// Print result
printf("%s\n", c);

// Clean up
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
```



Information Coding / Computer Graphics, ISY, LiTH

## **”Platform” vs ”device”**

**Platform = an OpenCL implementation**

**Device = a chip which the platform supports**



## **Language freedom... sort of**

- + Very easy to call from any language! Anything that can call into a C API can call OpenCL!**
- Kernel code is only C-style (although a specific implementation may choose to support more).**



## **Performance**

**Investigations report remarkably small differences**

**Our research on FFT so far has CUDA up to 2x faster**

**Very hard to compare, due to multiple OpenCL implementations**

**Some report CUDA to be better on NVidia platforms...  
some report a draw even there.**

**Our experience: Usually very close!**



## **Conclusions on OpenCL**

**Don't fear the complex setup phase! The rest is similar to CUDA.**

**Performance tend to be on par with CUDA or almost.**

**Speciality: heterogenous systems!**