Introduction to OpenCL

Open Compute Language


Zlatan efter nya succén:
VI ÄR REDO FÖR CL

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

Origins of OpenCL

Initiated by Apple

Managed by Khronos group

Many supporting parties

Many providers


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

Why?

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

For programming ”all” parallel architectures

Supported architectures (not complete!)

GPU

Intel compatible CPUs

ARM

FPGA

CELL

Intel Xeon Phi

Who decides? Any company making its own OpenCL implementation!

”Open”?

Means *open specification*

Like OpenGL

Many providers making their own implementation

There is not *one* OpenCL library.

No free lunch

Model does not fit all architectures

One size fits all - platform dependent optimizations hard to do

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	**CUDA**
compute unit	multiprocessor (SM)
work item	thread
work group	block
local memory	shared memory
private memory	registers

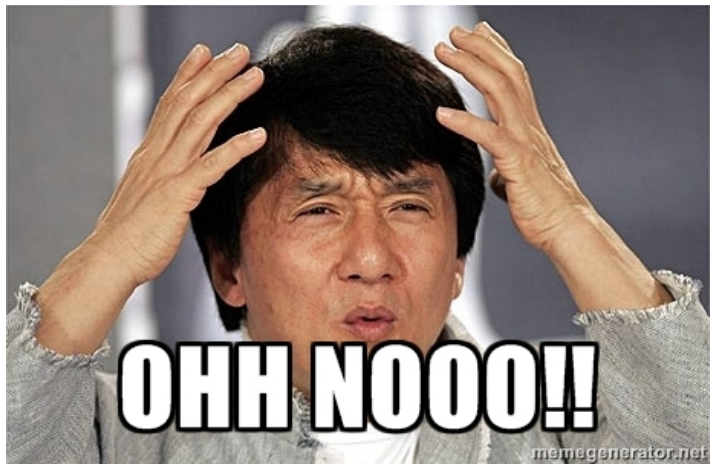
And CUDA local memory =?
OpenCL local memory (= CUDA shared memory)

Oh, that "local memory"...

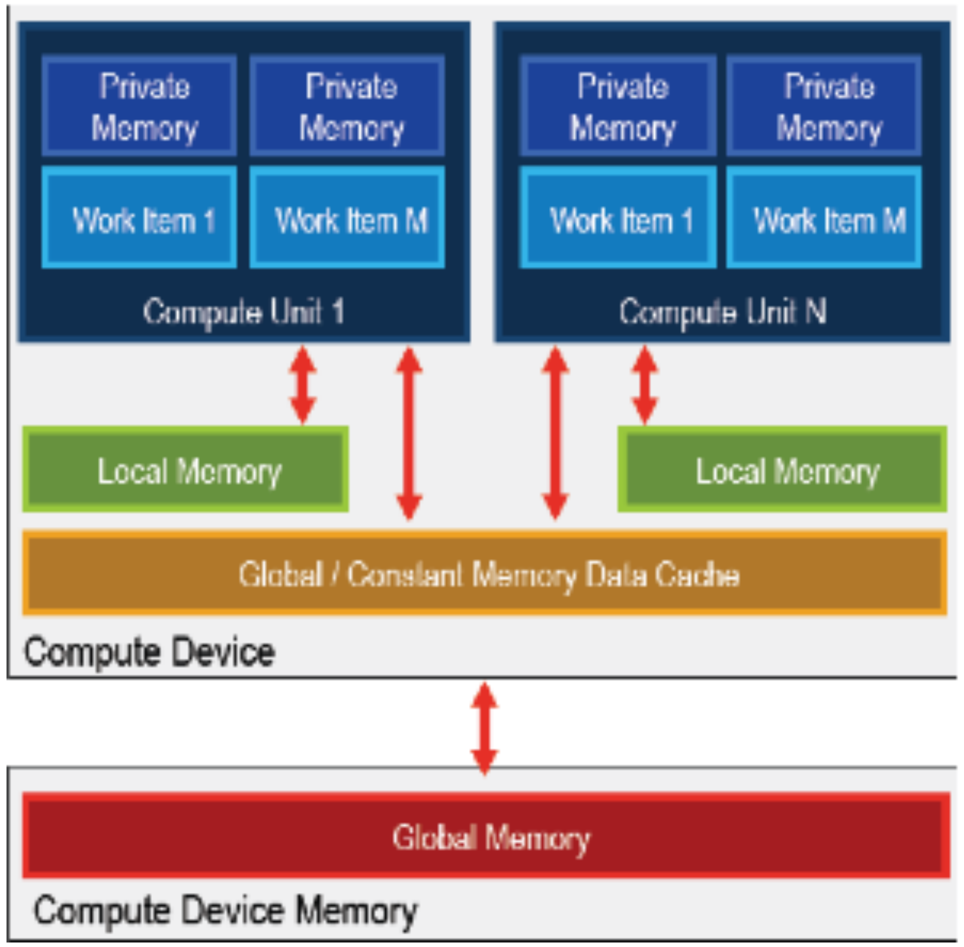
CUDA local memory = global memory accessible *only by one thread* (like registers but slower)

CUDA shared memory = **OpenCL local memory** = memory local inside the SM, shared within block/work group

Anyone else who thinks this makes sense?

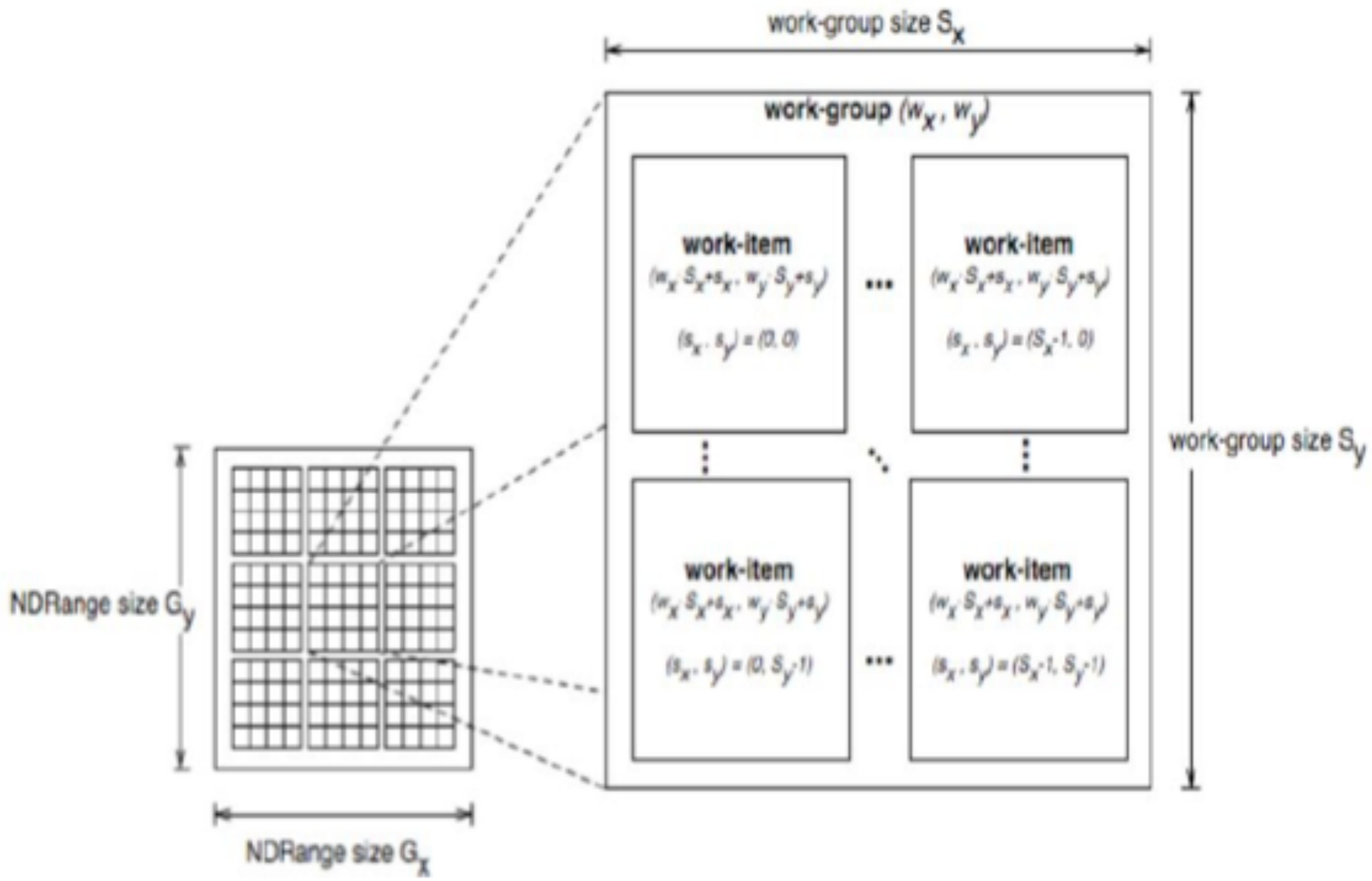


OpenCL memory model



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)
```

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

The host (CPU) can synchronize on global level:

Available for:
tasks (e.g. clEnqueueNDRangeKernel)
Memory(e.g.clEnqueueReadBuffer)
events (e.g. clWaitforEvents)

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!

Example using local (shared) memory:

```
 * Rank sorting in sorting OpenCL

__kernel void sort(__global unsigned int
*data, __global unsigned int *outdata, const
unsigned int length)
{
 unsigned int pos = 0;
 unsigned int i, b;
 unsigned int val;
 unsigned int this;

 unsigned int __local buf[128];

 // loop until all data is covered

 this = data[get_global_id(0)];
for (b = 0; b < length; b += 128)
{
 // Get data
 buf[get_local_id(0)] = data[get_local_id(0) + b];

 // Synch
 barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

 //find out how many values are smaller
 for (i = 0; i < 128; i++)
  if (this > buf[i]) // data[b + i])
   pos++;

 // Synch
 barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
}

outdata[pos] = this;
}
```

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);
```

”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!

+ Based on C99. Similar to CUDA.

- Kernel code is only C-style (although a specific implementation may choose to support more). C++ in 2.2.

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!