CLWrap

Nonsense free control of your GPU

John Colvin

Why bother?

- CPUs are *way* more complicated than necessary for a lot of computational tasks.

- They are optimised for serial workloads, parallelism has been somewhat tacked on the side.

- GPUs are purpose-built to run massively parallel computations at maximum speed, low cost and low power consumption.

- Piggy-backing on the games industry's economies of scale

- Everybody already has one (or two, or three…)

GP-GPU programming

- NVIDIA CUDA
- Kronos OpenCL
- Compute shaders (various APIs)
- Or just do compute using normal shaders

- Open standard

- Designed to work with any heterogeneous system, not just CPU host + GPU device.

- Drivers are available for all major GPUs and even CPUs (can target the CPU as the compute device).

- Has been somewhat overshadowed by CUDA, party due to a confusing model and intimidating C API.

Primitives of OpenCL

- `platform` : The driver, e.g. one from Intel for your CPU, one from NVIDIA for your GPU. Always a shared entry point between them: selecting is part of the OpenCL API. All the following are create w.r.t. one `platform`.

- `device` : E.g. one GPU.

- `context` : A group of devices.

- `kernel` : The entry point to some GPU code, compiled seperately for each `device` + `context` pair.

- `command_queue` : Execution queue for one device, put a kernel on here to get it run.

- `buffer / image` : data accessible from kernels. Bound to `contexts`, the driver is responsible for making sure it is available on any given device when needed.

The C API

Let’s just try and report the names
of the GPUs we’ve got…

```
// assume have already got a platform: platform_id
size_t nGpus;
cl_int err = clGetDeviceIDs(platform_id,
    CL_DEVICE_TYPE_GPU, 0, NULL, &nGpus);
if (err != CL_SUCCESS) { /* handle error */ }

cl_device_id *gpus = malloc(nGpus * sizeof(cl_device_id));
if (!gpus) { /* handle error */ }

err = clGetDeviceIDs(platform_id,
    CL_DEVICE_TYPE_GPU, nGpus, gpus, NULL);
if (err != CL_SUCCESS) { /* handle error */ }

char name[256] = {'\0'};
for (size_t i = 0; i < nDevices; ++i)
{
    err = clGetDeviceInfo(gpus[i],
            CL_DEVICE_NAME, 255, name, NULL);
    if(err != CL_SUCCESS) { /* handle error */ }
    puts(name);
}
```

Eww

Let’s try again in D

```
// assume have already got a platform, platform_id
size_t nGpus;
cl.int_ err = cl.getDeviceIDs(platform_id,
    cl.DEVICE_TYPE_GPU, 0, null, &nGpus);
if (err != cl.SUCCESS) { /* handle error */ }

cl.device_id *gpus = cast(cl.device_id*)
    malloc(nGpus * sizeof(cl.device_id));
if (!gpus) { /* handle error */ }

err = cl.getDeviceIDs(platform_id,
    cl.DEVICE_TYPE_GPU, nGpus, gpus, NULL);
if (err != cl.SUCCESS) { /* handle error */ }

char[256] name = '\0';
for (size_t i = 0; i < nDevices; ++i)
{
    err = cl.getDeviceInfo(gpus[i],
            cl.DEVICE_NAME, 255, name, NULL);
    if(err != cl.SUCCESS) { /* handle error */ }
    puts(name);
}
```

```
import clWrap, std.stdio, std.algorithm, std.conv, std.array;

int main(string[] args)
{
    if (args.length != 2 || args[1].empty)
    {
        stderr.writeln("Error: please provide platform name");
        return 1;
    }

    auto platform = getChosenPlatform(args[1]);
    auto devices = platform.getDevices(cl.DEVICE_TYPE_GPU);
    devices.map!(getInfo!(cl.DEVICE_NAME))
        .joiner("\n").writeln;

    return 0;
}
```

Tldr;

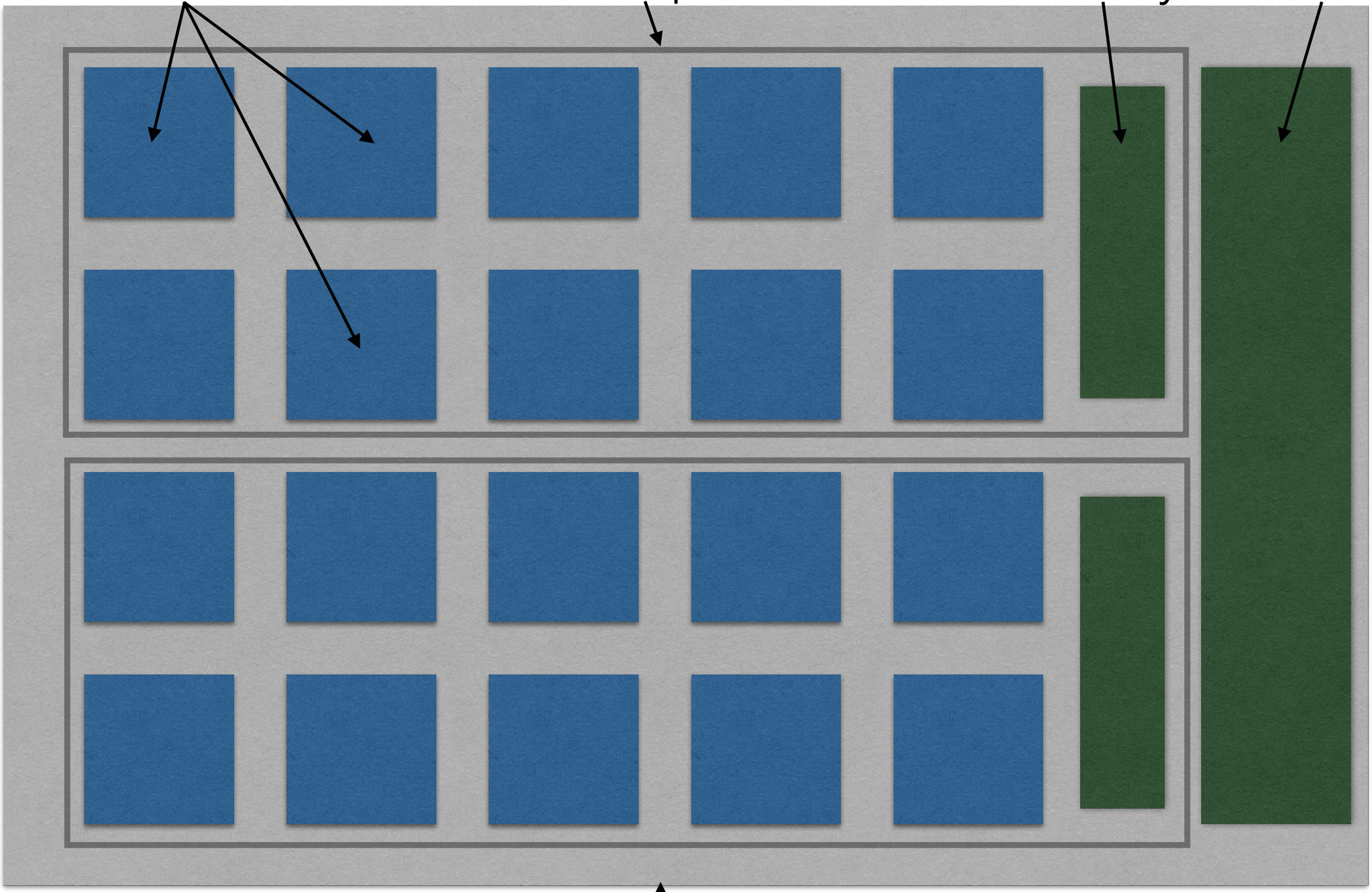
OpenCL, threads and data:

Work items

Work Group

Local Memory

Global Memory



Device

NDRanges

(x,y): global id
(x,y): local id

(0,0) (0,0)	(1,0) (1,0)	(2,0) (2,0)	(3,0) (3,0)	(4,0) (0,0)	(5,0) (1,0)	(6,0) (2,0)	(7,0) (3,0)
(0,1) (0,1)	(1,1) (1,1)	(2,1) (2,1)	(3,1) (3,1)	(4,1) (0,1)	(5,1) (1,1)	(6,1) (2,1)	(7,1) (3,1)
(0,2) (0,0)	(1,2) (1,0)	(2,2) (2,0)	(3,2) (3,0)	(4,2) (0,0)	(5,2) (1,0)	(6,2) (2,0)	(7,2) (3,0)
(0,3) (0,1)	(1,3) (1,1)	(2,3) (2,1)	(3,3) (3,1)	(4,3) (0,1)	(5,3) (1,1)	(6,3) (2,1)	(7,3) (3,1)
(0,4) (0,0)	(1,4) (1,0)	(2,4) (2,0)	(3,4) (3,0)	(4,4) (0,0)	(5,4) (1,0)	(6,4) (2,0)	(7,4) (3,0)
(0,5) (0,1)	(1,5) (1,1)	(2,5) (2,1)	(3,5) (3,1)	(4,5) (0,1)	(5,5) (1,1)	(6,5) (2,1)	(7,5) (3,1)

OpenCL C

- The language in which you write your device code.
- A restricted C with some extra builtins and funny type qualifiers.
- Compiled at *runtime* for the exact target hardware.
- pointers are tagged with `global` or `local` to specify which memory they point to.
- No pointers to pointers, everything is flat.
- a function annotated with `kernel` is an entry point, think like a `main`.

Some boring kernels:

```
kernel void myKernel(global float *input, float b)
{
    size_t i = get_global_id(0);
    input[i] = exp(input[i] * b);
}

kernel void myOtherKernel(global float *input,
    global float *output) {
    size_t idx = get_global_id(0) * get_global_size(1)
        + get_global_id(1);
    output[idx] = 1.0f / input[idx];
}

kernel void myStrangeKernel(global float *input,
    global float *output) {
    //assuming random_integer defined elsewhere
    size_t i_in = random_integer(0, get_global_size(0))
        + get_global_offset(0);
    size_t i_out = get_global_id(0);
    output[i_out] = input[i_in];
}
```

clWrap architecture

- Layer 1: Take the OpenCL C API and make it strictly typed

- Layer 2: Take this strictly typed layer and layer a more D-appropriate API on top

- Layer 3: Go to town. High level abstractions to enable people to easily execute code on co-processors. Based on Layer 2.

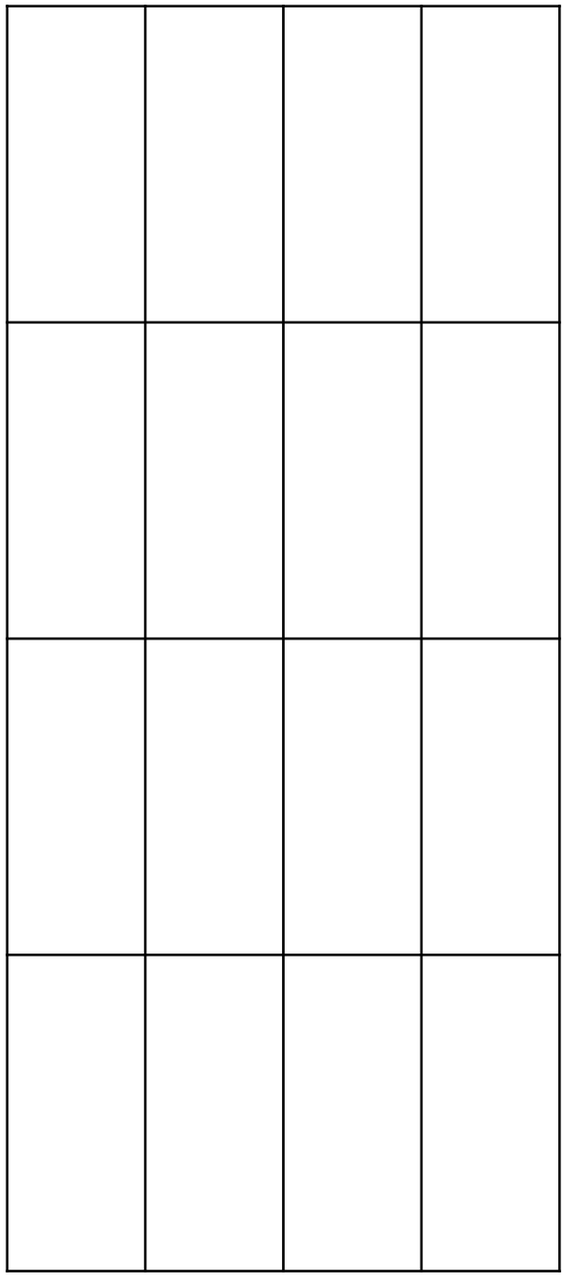
Let’s do it live!

Notes:

- The OpenCL API is quite big and it's only getting bigger
- The lack of assumptions and the implied choice that this gives you is both OpenCL's main strength and biggest weakness
- `clWrap` doesn't aim *or need* to be a comprehensive D-like wrapper of every possible use-case
- It sits on top of the C API, streamlining common tasks.
- The user can drop down to the C API at any point without inconvenience or fear of invalidating future use of `clWrap` functions.

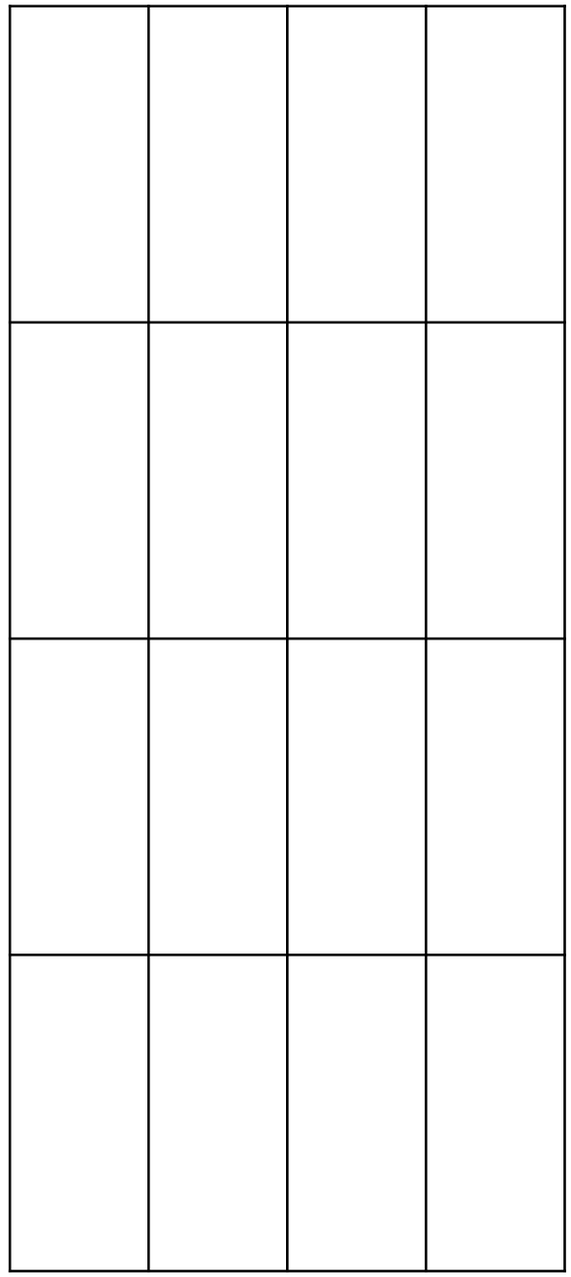
clMap

Data



1 to 1 mapping
of elements to
work items

Work Items



```
auto myKernel = CLKernelDef!("someKernel",
        CLBuffer!float, "input",
        float, "b")
(q{
    input[gLinId] = exp(cos(input[gLinId] * b));
});

void main()
{
    auto output = new float[](1000);
    auto input = iota(1000).map!(to!float).array;

    auto inputBuff = input
        .sliced(100, 10)
        .toBuffer(cl.MEM_COPY_HOST_PTR);

    inputBuff.clMap!myKernel(3.4f).enqueue;

    inputBuff.read(output).writeln;
}
```

Why not OpenCL 2.x?

- NVIDIA: only recently started supporting OpenCL 1.2, who knows how long until 2.0 (2.1? 2.2?)
- Derelict-CL: only supports 1.2
- Will probably move to support it anyway, the opportunities are too good to pass up.

What's next?

- Massive cleanup. A product v.s. an accretion of ideas
- Testing, testing…
- Get information from people about the things that really annoy them in OpenCL and GPU programming in general, fix them.
- Make D the obvious choice for cross-platform GPU control.
- Make general-purpose GPU usage a normal thing to do.

```
inputBuff.clMap!(q{
        *p0 = cos(*p0 * p1 + p2)
    })(3.4, 42)
    .read(output)
    .writeln;
```

```
inputBuff.clMap!(
    (p0, p1, p2) => cos(p0 * p1 + p2)
    )(3.4, 42)
    .read(output)
    .writeln;
```

?