OpenCL day 1: GPU hardware and the programming model

Cosmin Oancea and Troels Henriksen

January 28, 2019

Introduction and Course Contents

Hardware Trends

The GPU Architecture

The OpenCL Programming Model

Debugging and Profiling OpenCL

Coalesced Memory Accesses

Programming Exercises

Introduction and Course Contents

Hardware Trends

The GPU Architecture

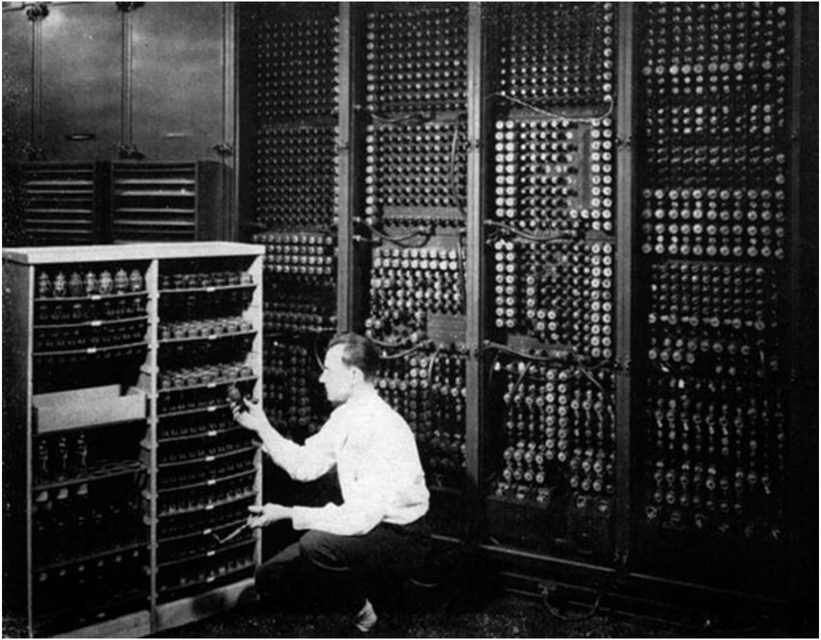
The OpenCL Programming Model

Debugging and Profiling OpenCL

Coalesced Memory Accesses

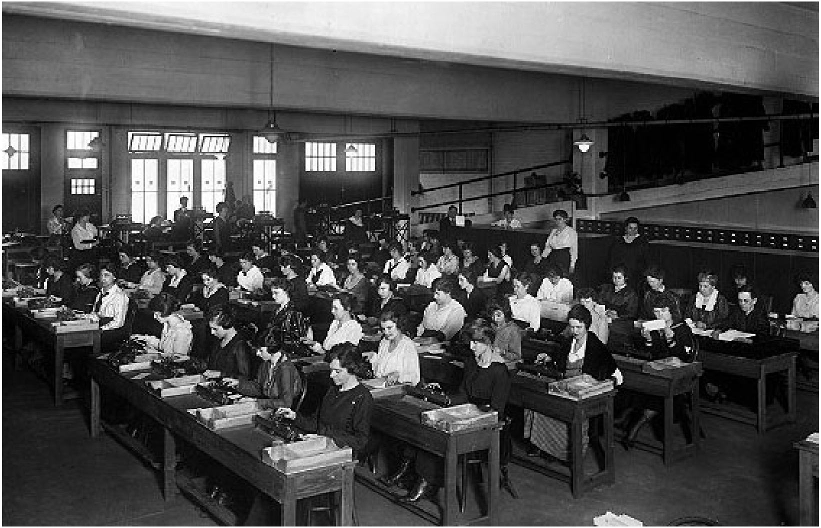
Programming Exercises

The first computers were not this

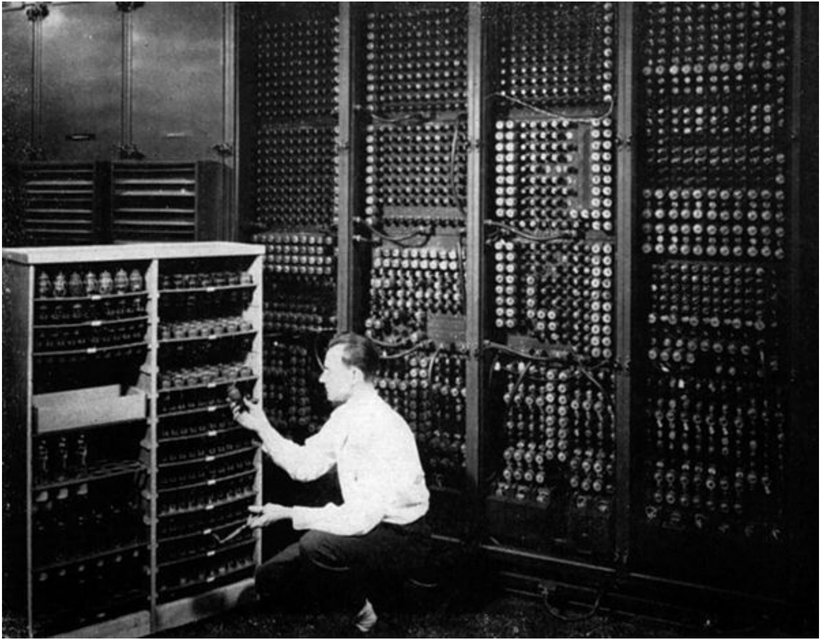


But this

And if you had a larger problem



But then they started looking like this



Then this

Then this

Then this

Then this

Then this

Then, from around 2005

Then, from around 2005

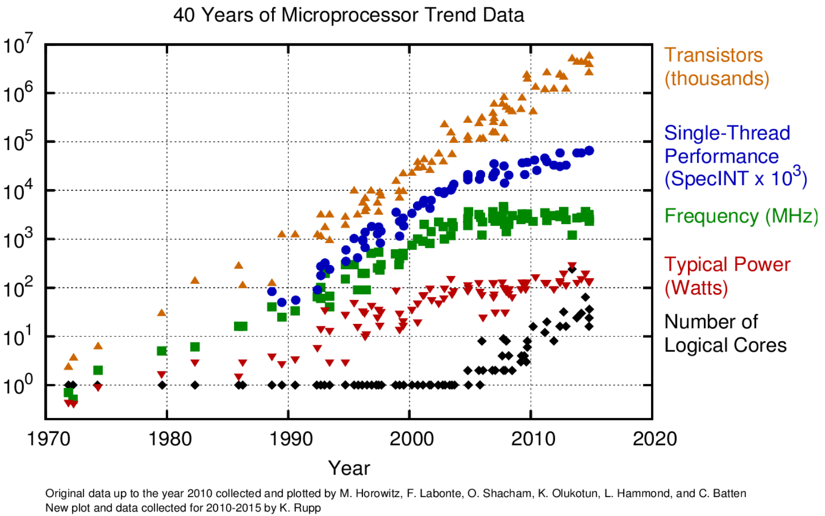
Then, from around 2005

Improvements in *sequential performance* stalled, although computers still got smaller and faster.

What Changed?

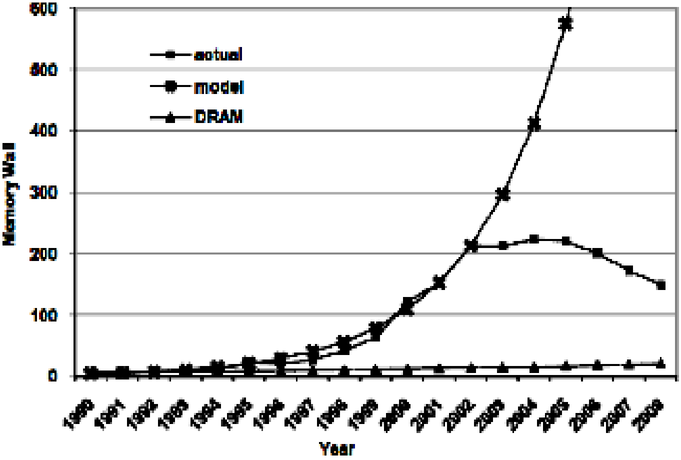
- *Power complexity* $P_{dynamic} \sim Freq^3$, preventing us from increasing processor frequency.
- *Memory wall*, ever-increasing performance gap between processor and memory (which means that *memory* becomes bottleneck, not processor speed).

CPU progress



Addressed with *more cores.*

The Memory Wall



Memory Wall = processor cycles/memory cycles

Addressed with caches (not scalable) and *latency hiding*.

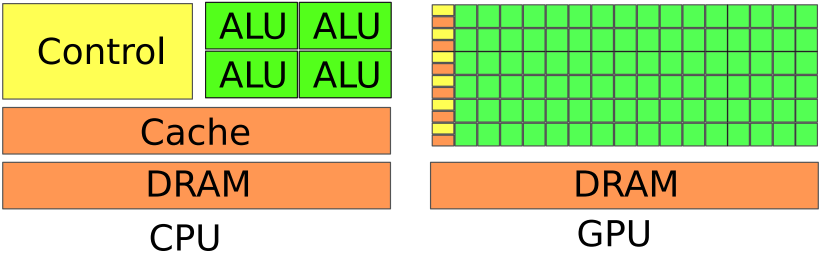
This is why GPUs are useful

The design of GPUs directly attacks these two problems.

- **Frequency scaling** becomes less of an issue because we can instead use thousands of (slower) cores.
- The **memory wall** is partially circumvented by using faster and smaller memory, but mostly by *latency hiding*. With tens of thousands of threads, we can probably find something else to do while some threads are waiting for memory!

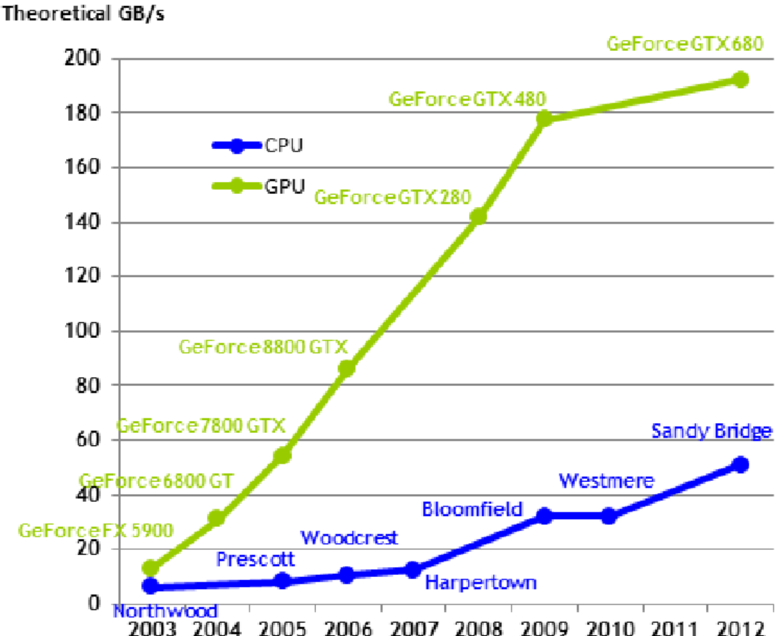
Ultimately, GPUs do *throughput processing*, and operations have (relatively) high latency.

CPUs compared to CPUs

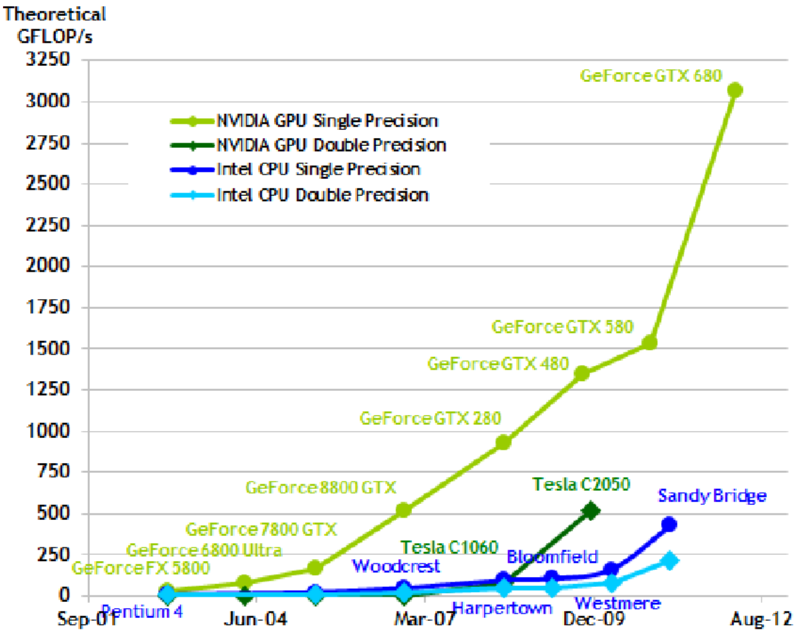


- GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens of thousands* of threads.
- GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc.
- Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*.

GPUs and Memory



GPUs and GFLOPS



Introduction and Course Contents

Hardware Trends

The GPU Architecture

The OpenCL Programming Model

Debugging and Profiling OpenCL

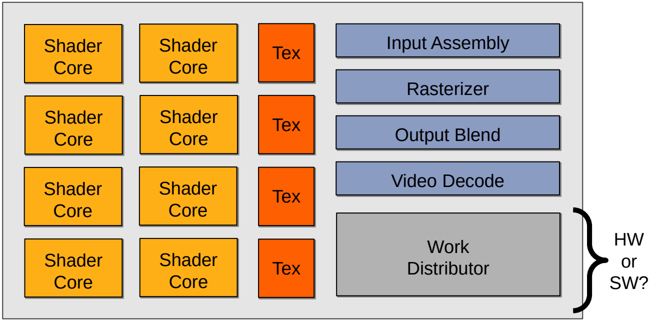
Coalesced Memory Accesses

Programming Exercises

The following slides are taken from the presentation *Introduction to GPU Architecture* by Ofer Rosenberg of AMD.

What’s in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



A diffuse reflectance shader

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Shader programming model:

Fragments are processed independently,
but there is no explicit parallel programming

Compile shader

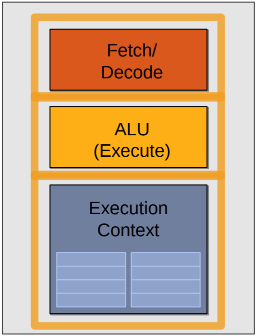
1 unshaded fragment input record

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

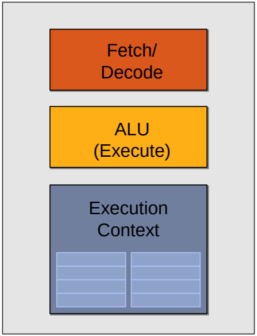
1 shaded fragment output record

Execute shader



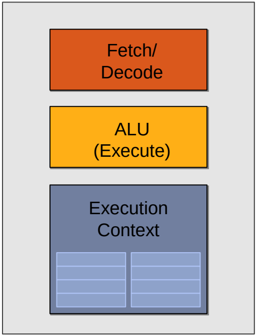

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader



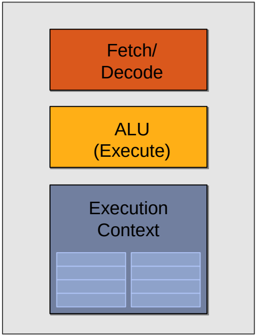

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader



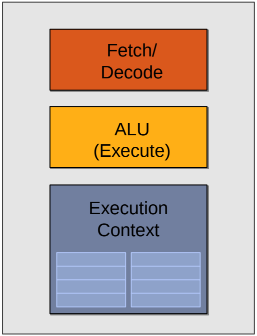

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader



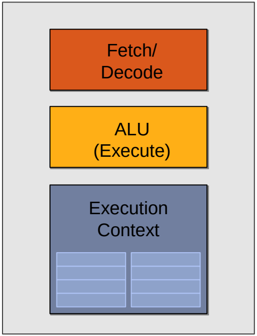

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

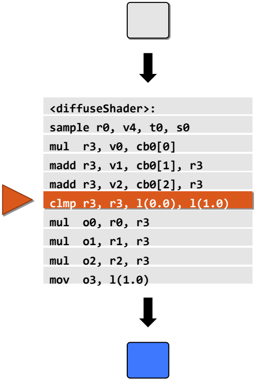
Execute shader




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

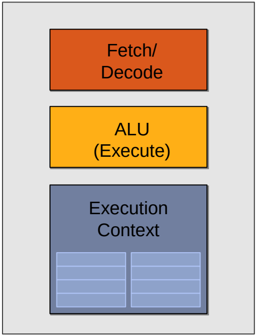
Execute shader





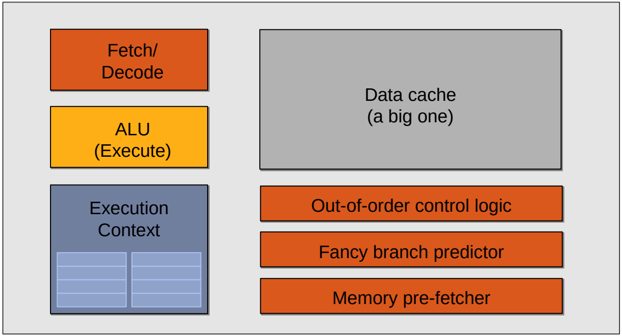
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader

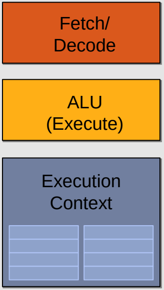



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

“CPU-style” cores



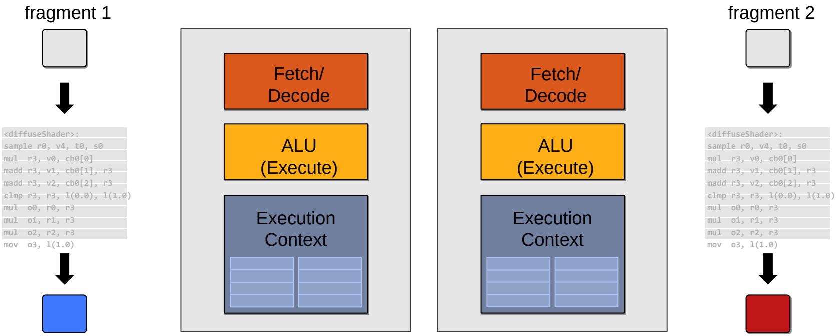
Slimming down



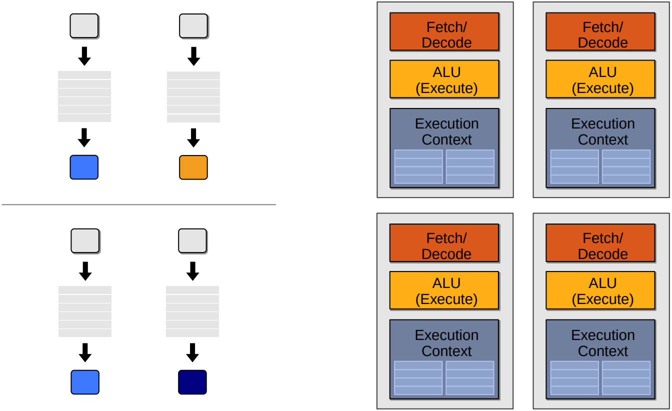
Idea #1:

Remove components that help a single instruction stream run fast

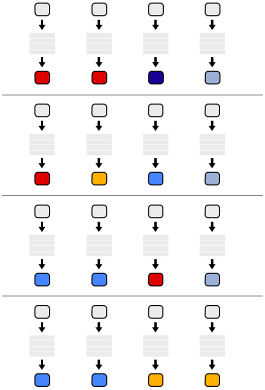
Two cores (two fragments in parallel)

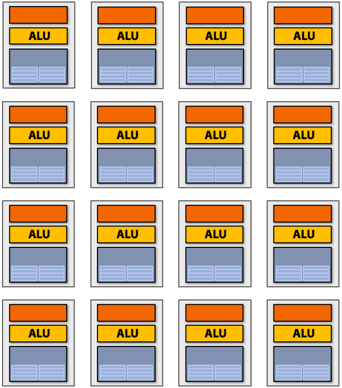


Four cores (four fragments in parallel)



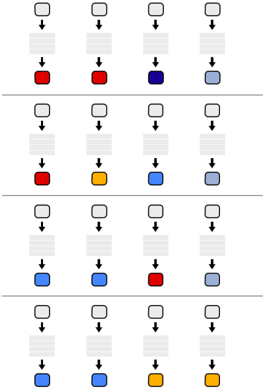
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

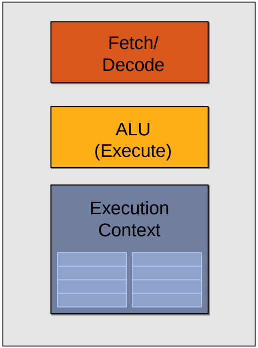
Instruction stream sharing



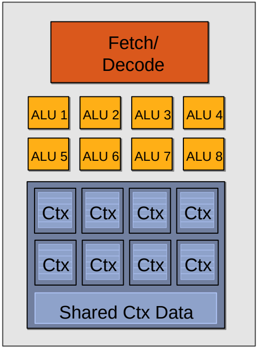
But ... many fragments should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Recall: simple processing core



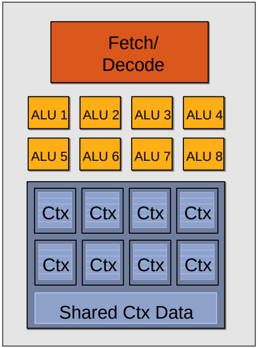
Add ALUs



Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader



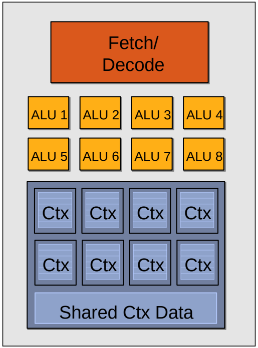

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using

scalar ops on scalar registers

Modifying the shader

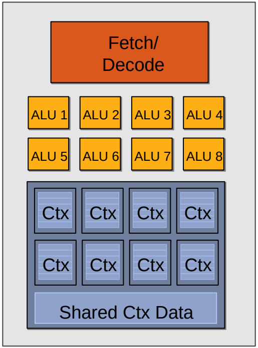


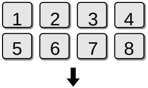

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```

New compiled shader:

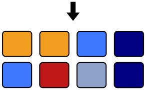
Processes eight fragments using vector ops on vector registers

Modifying the shader

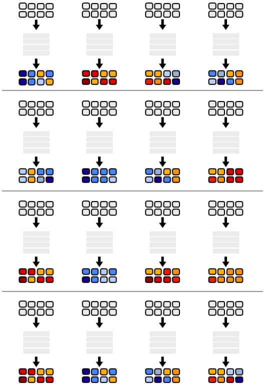




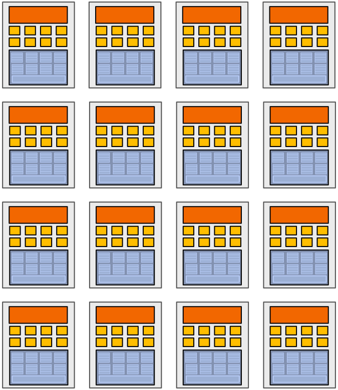

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```



128 fragments in parallel

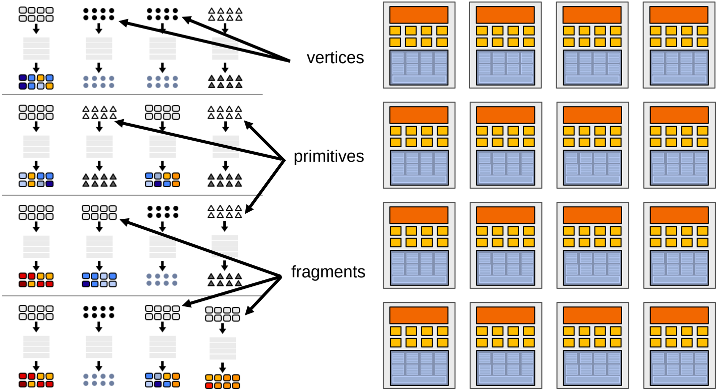


16 cores = 128 ALUs

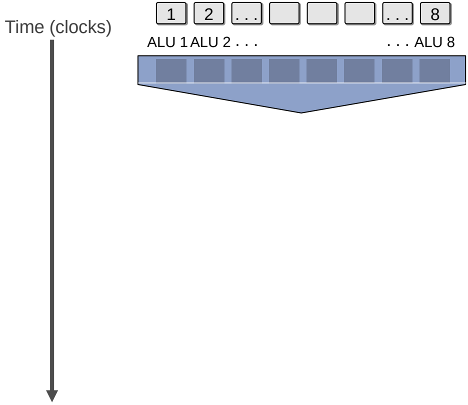


, 16 simultaneous instruction streams

128 [vertices/fragments primitives OpenCL work items] in parallel

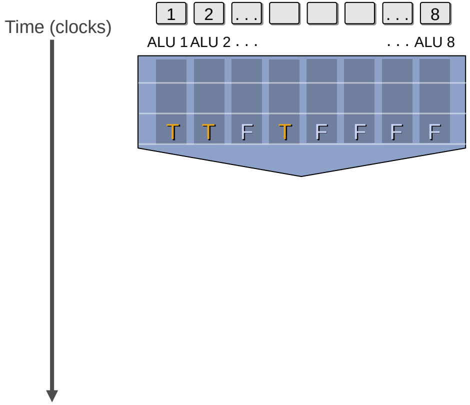


But what about branches?



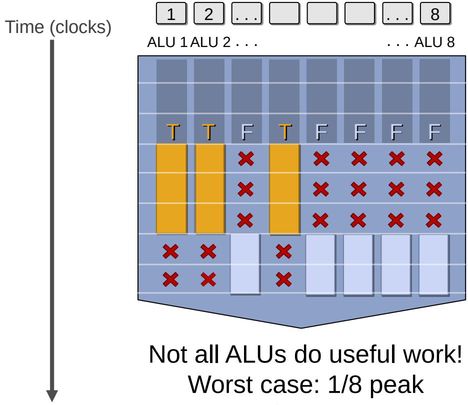

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

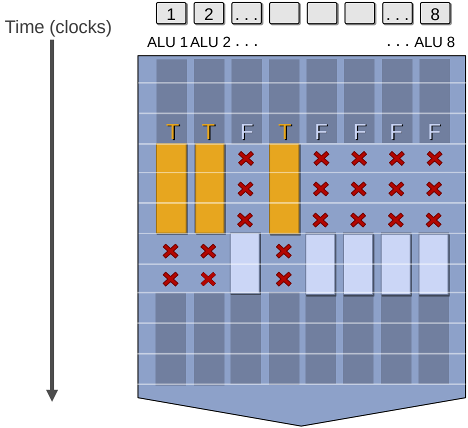
But what about branches?



Not all ALUs do useful work!
Worst case: 1/8 peak performance

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

But what about branches?

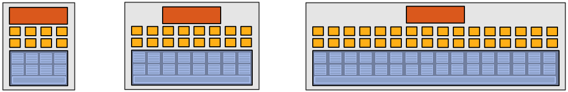



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce (“SIMT” warps), ATI Radeon architectures (“wavefronts”)



In practice: 16 to 64 fragments share an instruction stream.

Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100’s to 1000’s of cycles

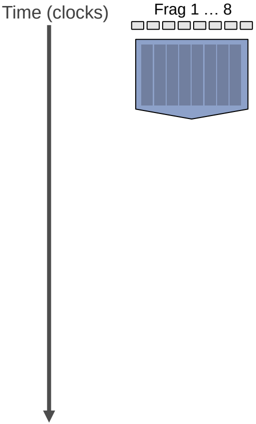
We’ve removed the fancy caches and logic that helps avoid stalls.

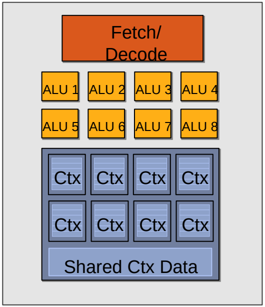
But we have LOTS of independent fragments.

Idea #3:

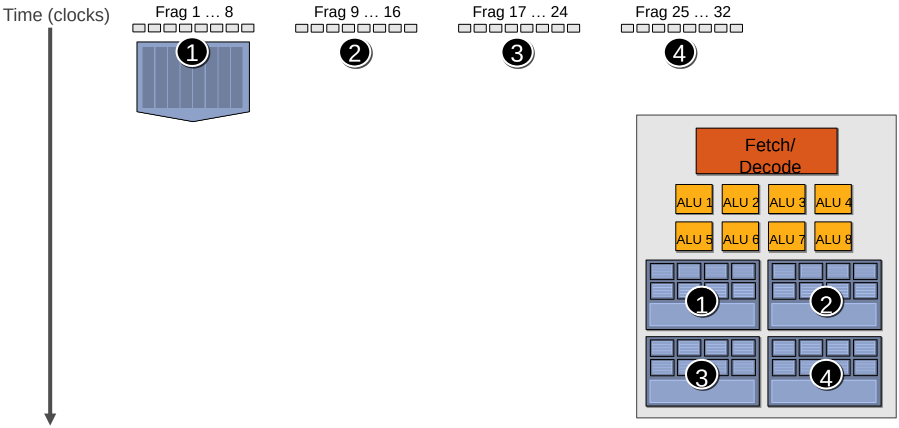
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

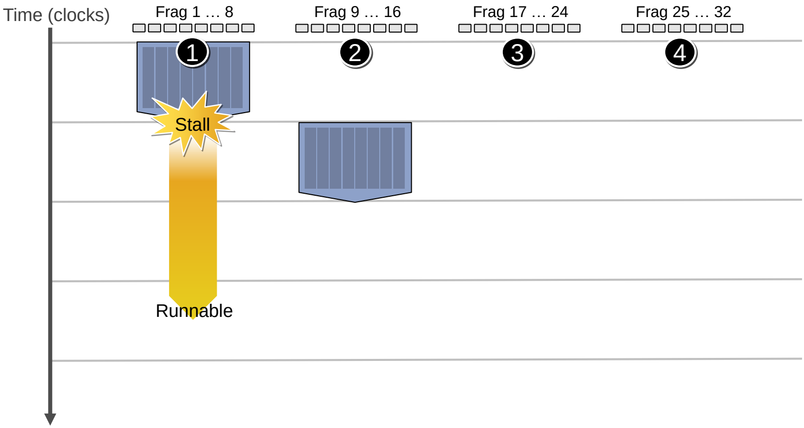




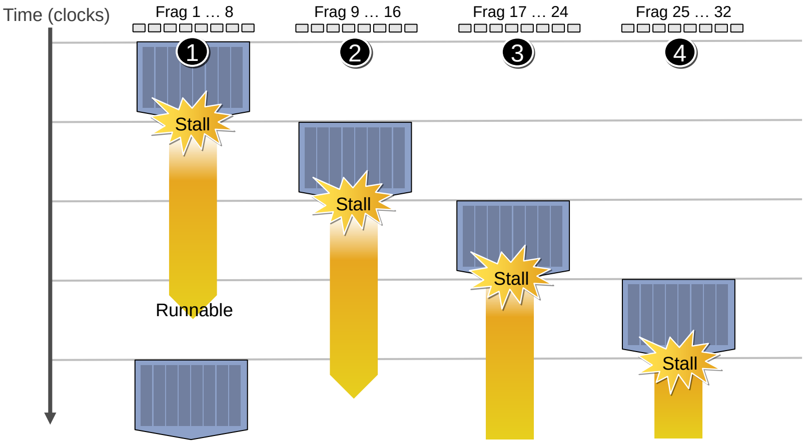
Hiding shader stalls



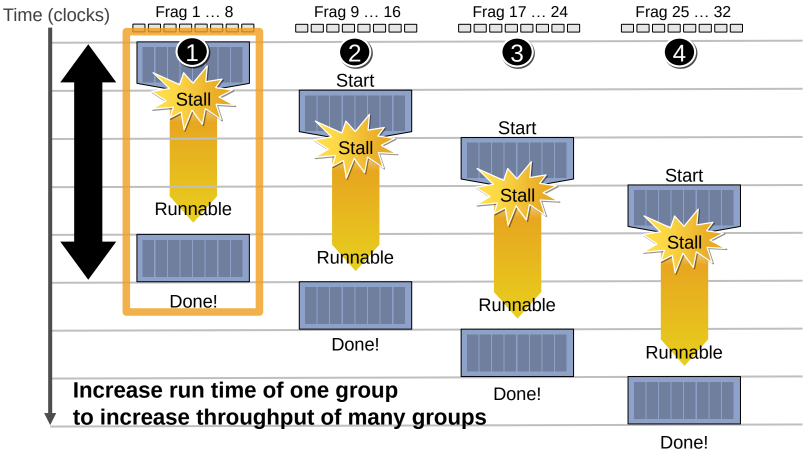
Hiding shader stalls



Hiding shader stalls

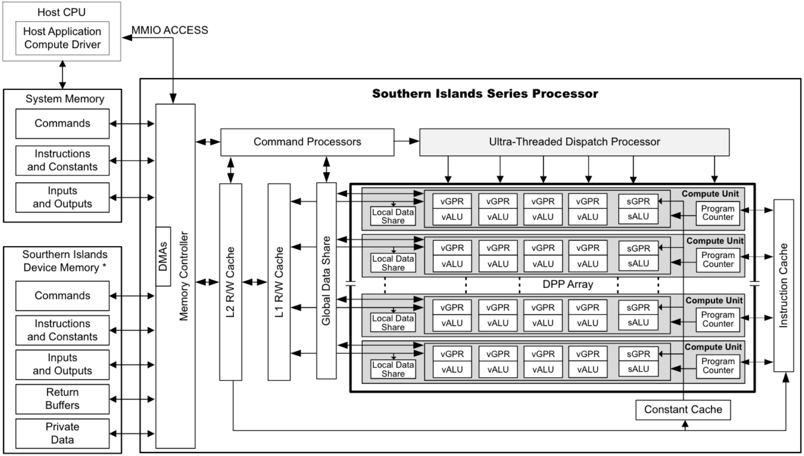


Throughput!



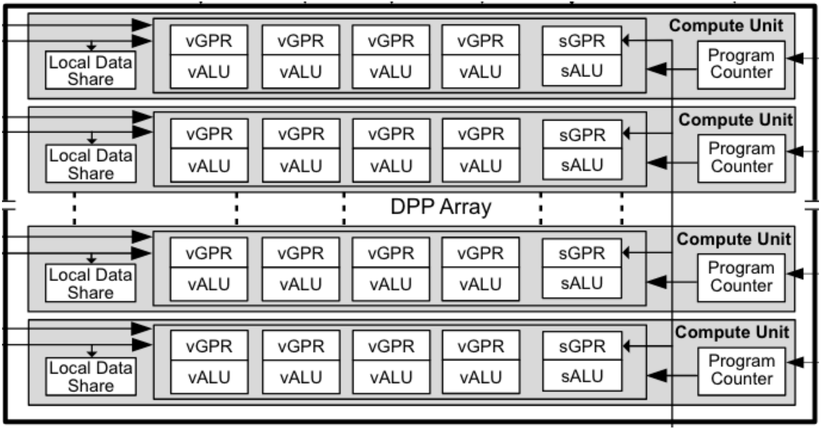
Increase run time of one group to increase throughput of many groups

The GPU we will be using: Radeon HD 7800[1]



[1] https://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

Zooming in on the Compute Units



- Each vector-ALU executes a *wavefront* of 64 work-items over four clock cycles.
- Many wavefronts in flight at once to hide latency.

Introduction and Course Contents

Hardware Trends

The GPU Architecture

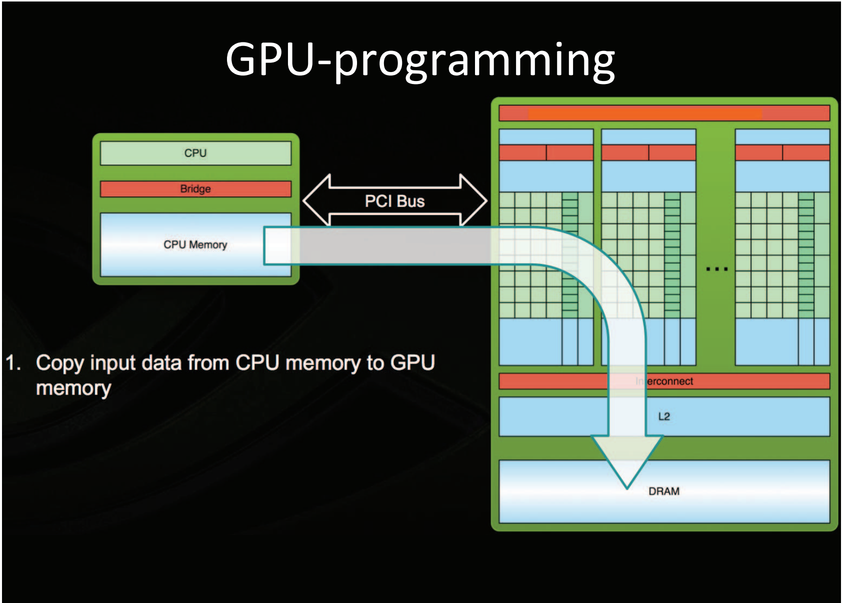
The OpenCL Programming Model

Debugging and Profiling OpenCL

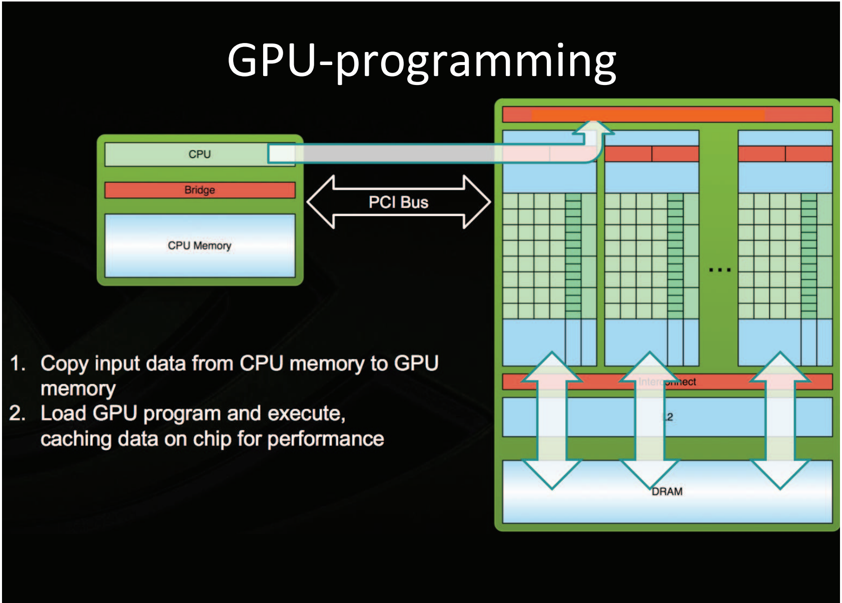
Coalesced Memory Accesses

Programming Exercises

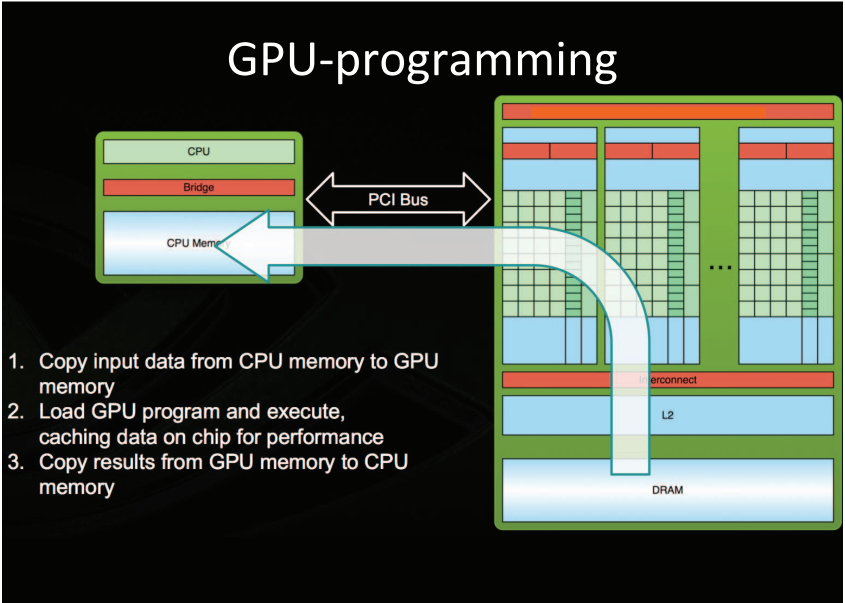
GPU programming



GPU programming



GPU programming



OpenCL for this course

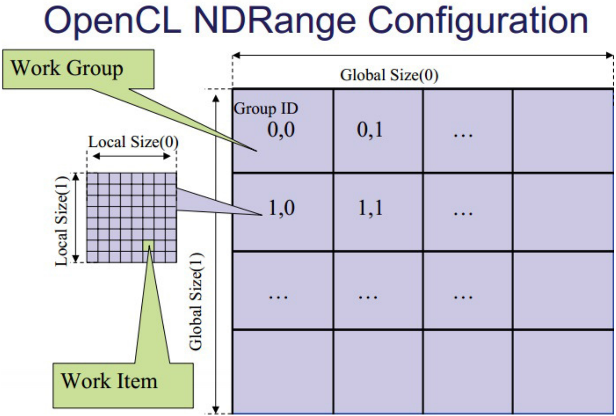
- OpenCL is a standard C API for programming GPUs and other "accelerators".
- OpenCL is very low-level and very boilerplate-heavy.
- Any real application will build domain-specific abstraction layers on top.
- Since we want to teach you *actual* OpenCL, we can't do that, but we will use a small library of abbreviations and helpers: `clutils.h`
- OpenCL comprises ordinary code running on the *host* (CPU), which calls API functions to direct the *device* (e.g. GPU).

OpenCL

`https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/`

OpenCL is an SIMT model

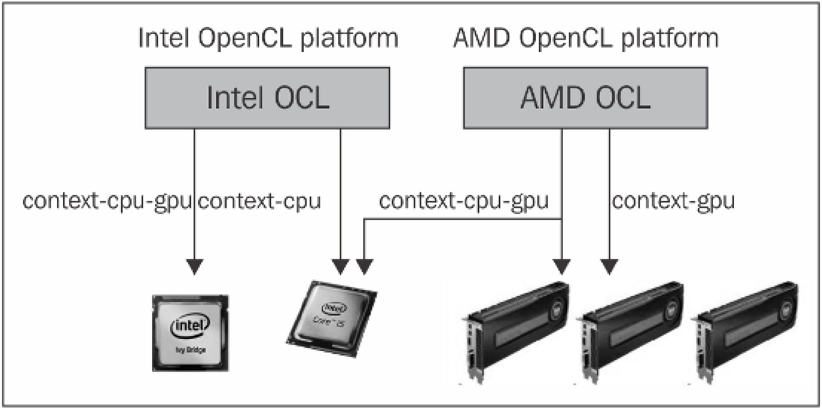
Single Instruction Multiple Threads means we provide a *sequential function* that is executed in parallel by multiple threads ("work items" in OpenCL).



Threads are arranged in *workgroups*, which form an *NDRange* (often called *grid*).

OpenCL Platforms and Devices

A *platform* is more like a *vendor* (technically, an OpenCL backend or driver). Each platform provides access to zero or more *devices*.



To use OpenCL, we must pick a *platform*, then one of its *devices*, use that to create a *context*, and then a *command queue* to which we can finally enqueue device operations.

Listing available devices (Day1/devices.c)

```
cl_int clGetPlatformIDs
  (cl_uint num_entries,
   cl_platform_id *platforms,
   cl_uint *num_platforms)
```

```
cl_uint num_platforms;

// Find the number of platforms.
OPENCL_SUCCEED(
  clGetPlatformIDs(0, NULL, &num_platforms));

printf("Found %d platforms\n",
       (int)num_platforms);
```

The OPENCL_SUCCEED() macro translates OpenCL error codes to strings and aborts the process in case of error. Proper error handling is inherently application-specific and left as a very boring exercise.

```c
// Make room for them.
cl_platform_id *all_platforms =
  calloc(num_platforms, sizeof(cl_platform_id));

// Fetch all the platforms.
OPENCL_SUCCEED(
  clGetPlatformIDs(num_platforms,
                   all_platforms,
                   NULL));

for (unsigned int i = 0; i < num_platforms; i++) {
  ...
}
```

```c
cl_int clGetPlatformInfo
  (cl_platform_id platform,
   cl_platform_info param_name,
   size_t param_value_size,
   void *param_value,
   size_t *param_value_size_ret)
```

```c
size_t req_bytes;
char *name;

// How much space do we need for the platform name?
OPENCL_SUCCEED(
  clGetPlatformInfo(all_platforms[i],
                    CL_PLATFORM_NAME,
                    0, NULL,
                    &req_bytes));
```

```
// Allocate space for the name and fetch it.
name = malloc(req_bytes);
OPENCL_SUCCEED(
  clGetPlatformInfo(all_platforms[i],
                    CL_PLATFORM_NAME,
                    req_bytes, name,
                    NULL));

printf("Platform %d: %s\n", i, name);

free(name);
```

```
// Now let us print the names of all the devices,
// first we count how many of them exist.
cl_uint num_devices;
OPENCL_SUCCEED(
  clGetDeviceIDs(all_platforms[i],
                 CL_DEVICE_TYPE_ALL,
                 0, NULL,
                 &num_devices));

// Then we make room for them.
cl_device_id *platform_devices =
  calloc(num_devices, sizeof(cl_device_id));

// Then we fetch them.
OPENCL_SUCCEED(
  clGetDeviceIDs(all_platforms[i],
                 CL_DEVICE_TYPE_ALL,
                 num_devices, platform_devices,
                 NULL));
```

```
for (unsigned int j = 0; j < num_devices; j++) {
  // How much space do we need for the device name?
  OPENCL_SUCCEED(
    clGetDeviceInfo(platform_devices[j],
                    CL_DEVICE_NAME,
                    0, NULL, &req_bytes));

  // Allocate space for the name and fetch it.
  name = malloc(req_bytes);
  OPENCL_SUCCEED(
    clGetDeviceInfo(platform_devices[j],
                    CL_DEVICE_NAME,
                    req_bytes, name, NULL));

  printf("\tDevice %d: %s\n", j, name);
  free(name);
}
```

OpenCL in Visual Studio

Warning

Neither Cosmin nor I are Windows users and we are entirely unexperienced with Visual Studio.

- Ensure the AMD OpenCL SDK is installed.
- After creating a new project, edit its properties and set...
  1. *C/C++→SDL checks* to No.
  2. *C/C++→Additional Include Directories* add `C:\Program Files (x86)\AMD APP SDK\3.0\include`.
  3. *Linker→Additional Library Directories* add `C:\Program Files (x86)\AMD APP SDK\3.0\lib`.
  4. *Linker→Input→Additional Dependencies* add `OpenCL.lib`.
- All but step 1 can be done by using the `AMDOpenCL.props` property sheet in the Git repository.
- Make sure you are doing a 64-bit build (VS calls this "x64").

Obtaining a `cl_command_queue` (clutils.h)

Assuming variables `platform_index` and `device_index`.

```
cl_uint num_platforms;
OPENCL_SUCCEED(
  clGetPlatformIDs(0, NULL, &num_platforms));
cl_platform_id *all_platforms =
  (cl_platform_id*)
  calloc(num_platforms, sizeof(cl_platform_id));
OPENCL_SUCCEED(
  clGetPlatformIDs(num_platforms,
                   all_platforms,
                   NULL));

assert(platform_index < num_platforms);
cl_platform_id platform =
  all_platforms[platform_index];
```

```
cl_uint num_devices;
OPENCL_SUCCEED(
  clGetDeviceIDs(platform,
                 CL_DEVICE_TYPE_ALL,
                 0, NULL,
                 &num_devices));
cl_device_id *platform_devices =
  (cl_device_id*)
  calloc(num_devices, sizeof(cl_device_id));
OPENCL_SUCCEED(
  clGetDeviceIDs(platform,
                 CL_DEVICE_TYPE_ALL,
                 num_devices,
                 platform_devices,
                 NULL));

assert(device_index < num_devices);
cl_device_id device = platform_devices[device_index
```

```
cl_context clCreateContext
  (cl_context_properties *properties,
   cl_uint num_devices,
   const cl_device_id *devices,
   void *pfn_notify (...),
   void *user_data,
   cl_int *errcode_ret)
```

```
cl_context_properties properties[] = {
  CL_CONTEXT_PLATFORM,
  (cl_context_properties)platform,
  0
};

cl_int error;
cl_context ctx =
  clCreateContext(properties, 1, &device,
                  NULL, NULL, &error);
OPENCL_SUCCEED(error);
```

```
cl_command_queue clCreateCommandQueue
  (cl_context context,
   cl_device_id device,
   cl_command_queue_properties properties,
   cl_int *errcode_ret)
```

```
cl_command_queue queue =
  clCreateCommandQueue(*ctx, *device, 0, &error);
OPENCL_SUCCEED(error);
```

Using `clutils.h`, all of the above can be replaced with:

```
cl_context ctx;
cl_command_queue queue;
cl_device_id device;
opencl_init_command_queue_default
  (&device, &ctx, &queue);
```

Rot-13 in OpenCL (Day1/rot13.c)

Rot-13 is a cutting edge encryption algorithm. In C, it is:

```
void rot13(char *out, char *in, int n) {
  for (int i = 0; i < n; i++) {
    if (i < n) {
      if (in[i] >= 'a' && in[i] <= 'z') {
        out[i] = (in[i] - 'a' + 13) % 26 + 'a';
      } else {
        out[i] = in[i];
      }
    }
  }
}
```

Here restricted to operate on lowercase ASCII only to ensure readable output.

Loading OpenCL programs

We obtain an OpenCL *program* by passing its source (written in OpenCL C) to `clBuildProgram()`. Lots of boilerplate again; let's just use `clutils.h`:

```
cl_program program =
  opencl_build_program(ctx, device,
                       "kernels/rot13.cl",
                       "");
```

OpenCL C is a cut-down dialect of C with many restrictions:

- No function pointers.
- No recursion.
- Limited standard library.
- No memory allocation.
- No printing to the screen.
- *Etc*...

Kernel functions (Day1/kernels/rot13.cl)

An OpenCL C program contains kernel functions that serve as entry points:

```
// Rot-13 for lowercase ASCII.
kernel void rot13(global char *out,
                  global char *in,
                  int n) {
  int gtid = get_global_id(0);
  if (gtid < n) {
    if (in[gtid] >= 'a' &&
        in[gtid] <= 'z') {
      out[gtid] = (in[gtid] - 'a' + 13)
                  % 26 + 'a';
    } else {
      out[gtid] = in[gtid];
    }
  }
}
```

Accesing kernels (Day1/rot13.c)

To launch a kernel on the GPU from the host, we first use `clCreateKernel()` with the `cl_program` object we got back:

```
cl_kernel rot13_k =
  clCreateKernel(program, "rot13", &error);
OPENCL_SUCCEED(error);
```

- Now we can ask the GPU to run the kernel.
- Except that GPUs have their own separate memory, so we have no data for the kernel to work on!

Allocating GPU memory

```
cl_mem clCreateBuffer(cl_context context,
                      cl_mem_flags flags,
                      size_t size,
                      void *host_ptr,
                      cl_int *errcode_ret)
```

```
char *string = "Hello, World!\n";
cl_int n = strlen(string);

cl_mem input =
   clCreateBuffer(ctx,
                  CL_MEM_READ_ONLY
                  | CL_MEM_COPY_HOST_PTR,
                  n, string, &error);

cl_mem output =
  clCreateBuffer(ctx, CL_MEM_WRITE_ONLY,
                 n, NULL, &error);
```

Passing arguments to the kernel

Remember that `rot13_k` object? We finally get to use it.

```
clSetKernelArg
  (rot13_k, 0, sizeof(cl_mem), &output);
clSetKernelArg
  (rot13_k, 1, sizeof(cl_mem), &input);
clSetKernelArg
  (rot13_k, 2, sizeof(cl_int), &n);
```

Reminder on `Day1/kernels/rot13.cl`:

```
kernel void rot13(global char *out,
                  global char *in,
                  int n) {
  ...
}
```

Launching a kernel

When launching a kernel, we must specify the layout of the grid:

- The number of dimensions (1, 2, or 3).
- The size of each workgroup in each dimension.
- The total number of threads in each dimension (which must be divisible by the workgroup size in that dimension).

For our rot-13, we want a 1D grid with one thread per input, rounded up to the workgroup size.

```
size_t local_work_size[1] = { 256 };
size_t global_work_size[1] = {
  div_rounding_up(n, local_work_size[0])
  * local_work_size[0]
};
```

Workgroup size is a tunable parameter, but we'll always pick 256 for now.

clEnqueueNDRangeKernel()

```
cl_int clEnqueueNDRangeKernel
  (cl_command_queue command_queue,
   cl_kernel kernel,
   cl_uint work_dim,
   const size_t *global_work_offset,
   const size_t *global_work_size,
   const size_t *local_work_size,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

```
clEnqueueNDRangeKernel(queue, rot13_k,
                       1,
                       NULL,
                       global_work_size,
                       local_work_size,
                       0, NULL, NULL);
```

More on command queues

- Enqueuing a command is *asynchronous*. It might start executing immediately, soon, or not at all.
- Use `clFinish()` to ensure that all operations have finished:

```
OPENCL_SUCCEED(clFinish(queue));
```

This is also where execution errors are typically reported.

Reading results back from the GPU

```
cl_int clEnqueueReadBuffer
  (cl_command_queue command_queue,
   cl_mem buffer,
   cl_bool blocking_read,
   size_t offset, size_t cb, void *ptr,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

```
char *output_string = malloc(n + 1);
output_string[n] = '\0'; // Ensure 0-termination.
clEnqueueReadBuffer(queue, output, CL_TRUE,
                    0, n,
                    output_string,
                    0, NULL, NULL);

printf("Result: %s\n", output_string);
```

Introduction and Course Contents

Hardware Trends

The GPU Architecture

The OpenCL Programming Model

Debugging and Profiling OpenCL

Coalesced Memory Accesses

Programming Exercises

Debugging with `oclgrind`

- `https://github.com/jrprice/Oclgrind`
- Makes itself available as an OpenCL platform that runs your kernels in an error-checking interpreter.
- A lot like `valgrind`.
- Fairly slow, so use it on reduced workloads.

```
$ oclgrind ./rot13
Using platform: Oclgrind
Using device: Oclgrind Simulator

Invalid read of size 1 at global memory address 0
    x1000000000000e
   Kernel: rot13
   Entity: Global(14,0,0) Local(14,0,0) Group(0,0,0)
     %1 = load i8, i8 addrspace(1)* %arrayidx, align
      1, !dbg !20
   At line 5 of input.cl:
     if (in[gtid] >= 'a' && in[gtid] <= 'z') {
...
```

Profiling with Wall Clock Time

Just like how you profile anything else.

```
// Current wall time in microseconds.
static int64_t get_wall_time(void);
```

Use it like this:

```
int64_t before = get_wall_time();

...

clFinish(ctx);

int64_t after = get_wall_time();

printf("Took %d microseconds\n",
       (int)(after-before));
```

The `clFinish()` call is crucial as otherwise the device may still be working (remember that most enqueuings are *asynchronous*).

Profiling with Events

An event is an object that communicates the status of an OpenCL command. Whenever we enqueue something in a command queue, we can get an event object back.

```
cl_int clEnqueueNDRangeKernel
  (cl_command_queue command_queue,
   cl_kernel kernel,
   cl_uint work_dim,
   const size_t *global_work_offset,
   const size_t *global_work_size,
   const size_t *local_work_size,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

Retrieving Information from Events

```
cl_int clGetEventInfo
  (cl_event event,
   cl_event_info param_name,
   size_t param_value_size,
   void *param_value,
   size_t *param_value_size_ret)
```

```
cl_int clGetEventProfilingInfo
  (cl_event event,
   cl_profiling_info param_name,
   size_t param_value_size,
   void *param_value,
   size_t *param_value_size_ret)
```

The latter only works if CL_QUEUE_PROFILING_ENABLE was passed to clCreateCommandQueue().

Values for `cl_profiling_info`

`CL_PROFILING_COMMAND_QUEUED`
: When the command was queued.

`CL_PROFILING_COMMAND_SUBMIT`
: When the command was sent to the device.

`CL_PROFILING_COMMAND_START`
: When the command started executing.

`CL_PROFILING_COMMAND_END`
: When the command finished executing.

- All produce a value of type cl_ulong.
- clGetEventProfilingInfo () returns CL_PROFILING_INFO_NOT_AVAILABLE if the information is not available (yet)

Example of Profiling with Events

```
cl_event write_e;
clEnqueueWriteBuffer(queue, to, CL_FALSE,
                     0, n,
                     from,
                     0, NULL, &write_e));

...

cl_ulong start, end;

clGetEventProfilingInfo
  (write_e, CL_PROFILING_COMMAND_START,
   sizeof(start), &start, NULL);
clGetEventProfilingInfo
  (write_e, CL_PROFILING_COMMAND_START,
   sizeof(end), &end, NULL);
```

Event Profiling versus Wall Clock Profiling

- Event profiling is **much more fine-grained** and lets us see the per-operation runtime.
- Measuring per-operation with wall clock would require us to `clFinish()` after every operation, which is very slow because it prevents pipelining.
- Wall clock profiling tells us about **overall application performance**. We generally cannot just sum the runtimes for each event, since the commands may overlap in time, and the events do not count host-based overheads.
- **Ideally, use both.**

However, neither of these approaches will tell us *why* something is slow...

Introduction and Course Contents

Hardware Trends

The GPU Architecture

The OpenCL Programming Model

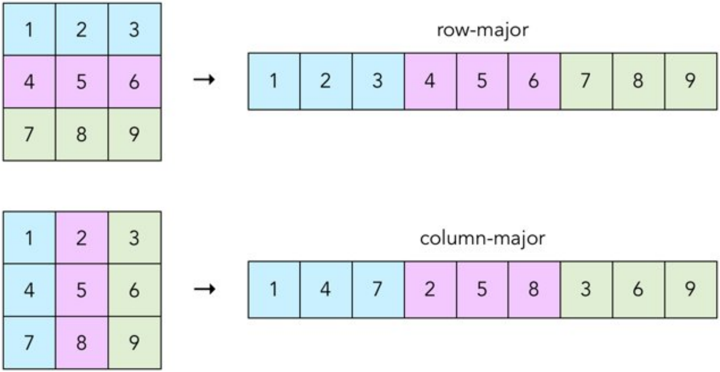
Debugging and Profiling OpenCL

Coalesced Memory Accesses

Programming Exercises

Summing the rows of a matrix

Consider summing the rows/columns of a 10000×10000 row-major matrix on CPU and GPU:



Performance

```
for (int row = 0; row < n; row++) {
  cl_int sum = 0;
  for (int col = 0; col < n; col++) {
    sum += matrix[row*n+col];
  }
  sums[row] = sum;
}
```

On the GPU, we assign one iteration of the outer loop to each thread.

Summing rows on CPU	22025μs
Summing columns on CPU	741225μs
Summing rows on GPU	60461μs
Summing columns on GPU	6169μs

Why does this go so badly?

The reason is our memory access pattern – specifically, our loads are not *coalesced*.

Memory Coalescing

All threads within each consecutive 16-thread gang should simultaneously access consecutive elements in memory to maximise memory bus usage.

- If neighboring threads access widely distant memory in the same clock cycle, the loads have to be *sequentialised*, instead of all fulfilled using one (wide) memory bus operation.
- The HD 7800 has a memory bus width of 256 bits, so only using 32 bits per operation exploits an eight of the bandwidth.

The accesses specifically

Table: Current accesses - this is worst case behaviour!

Iteration	**Thread** 0	**Thread** 1	**Thread** 2	...
0	`matrix`[0]	`matrix`[n]	`matrix`[$2n$]	...
1	`matrix`[1]	`matrix`[$n+1$]	`matrix`[$2n+1$]	...
2	`matrix`[2]	`matrix`[$n+2$]	`matrix`[$2n+2$]	...

Table: These are the accesses we want

Iteration	**Thread** 0	**Thread** 1	**Thread** 2	...
0	`matrix`[0]	`matrix`[1]	`matrix`[2]	...
1	`matrix`[n]	`matrix`[$n+1$]	`matrix`[$n+2$]	...
2	`matrix`[nc]	`matrix`[$2n+1$]	`matrix`[$2n+2$]	...

This is the exact opposite of what we are usually taught for CPUs!

Introduction and Course Contents

Hardware Trends

The GPU Architecture

The OpenCL Programming Model

Debugging and Profiling OpenCL

Coalesced Memory Accesses

Programming Exercises

Profiling rot-13 with wall clock and events

- `Day1-exercises/rot13-profile-simple.c`
- `Day1-exercises/rot13-profile-events.c`

Try profiling both one and multiple kernel launches. What do you observe? What if you call `clFinish()` after every kernel invocation? What if you also count the cost of copying from the CPU to the GPU?

Reversing a string in parallel

Write an OpenCL for reversing a string. Base it heavily on the Rot-13 program. Create your own Visual Studio project for it as well.

Load balancing (`Day1-exercises/fibfact.c`)

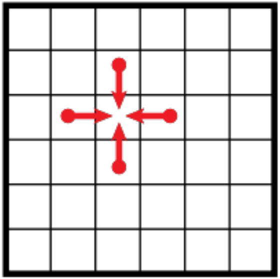
Finish the program, which is supposed to do the equivalent of:

```
void f (int k, float *out, int *ns, int *op)
for (int i = 0; i < k; i++) {
  int n = ns[i];
  int x;
  if (op[i] == 1) {
    x = fib(n);
  } else {
    x = fact(n);
  }
  out[i] = x;
}
```

- Where `fact()` and `fib()` are the usual factorial and Fibonacci functions.
- How fast does it run for various contents of `ns` and `ops`? Can you make it faster by preprocessing these arrays?

Implementing Game of Life (`Day1-exercises/life-arrays.c`)

Conway's Game of Life is a simple 2D cellular automaton ("stencil") that is embarassingly parallel. Each cell is updated based on the value of its neighbours.



Using image objects for Game of Life (`Day1-exercises/life-images.c`)

- GPUs have special hardware for textures, and this can be used whenever we need 2D arrays with spatial locality (like in Game of Life).
- Instead of `clCreateBuffer()`, use `clCreateImage()`, and in the kernel use the `image2d_t` type.
- Implement this as a 2D kernel in `Day1-exercises/life-images.c`.
- Main challenge: understand the OpenCL documentation and figure out how to represent our information in a colour channel.

Help for image objects

```
cl_mem clCreateImage
  (cl_context context,
   cl_mem_flags flags,
   const cl_image_format *image_format,
   const cl_image_desc *image_desc,
   void *host_ptr,
   cl_int *errcode_ret)
```

This is probably the best image format for us:

```
cl_image_format format =
  { .image_channel_order = CL_RGBA,
    .image_channel_data_type = CL_UNSIGNED_INT8
  };
```

Image objects inside kernels

Inside the kernel we can use these functions to read/write elements:

```
unsigned int4 read_imageui(image2d_t image,
                           sampler_t sampler,
                           int2 coord)

void write_imageui(image2d_t image,
                   int2 coord,
                   unsigned int4 color)
```

E.g.

```
write_imageui(img,
              (int2)(x,y),
              (uint4)(r,g,b,a))
uint4 v = read_imageui(img, sampler, (int2)(x,y));
// v.s0, v.s1, v.s2, v.s3
```

Matrix multiplication

- Implement matrix multiplication as a 2D kernel with one thread per element of the output matrix.
- **Spoiler alert:** you will find that it is slow. Why?

Cosmin will eventually tell you how to make it less slow.