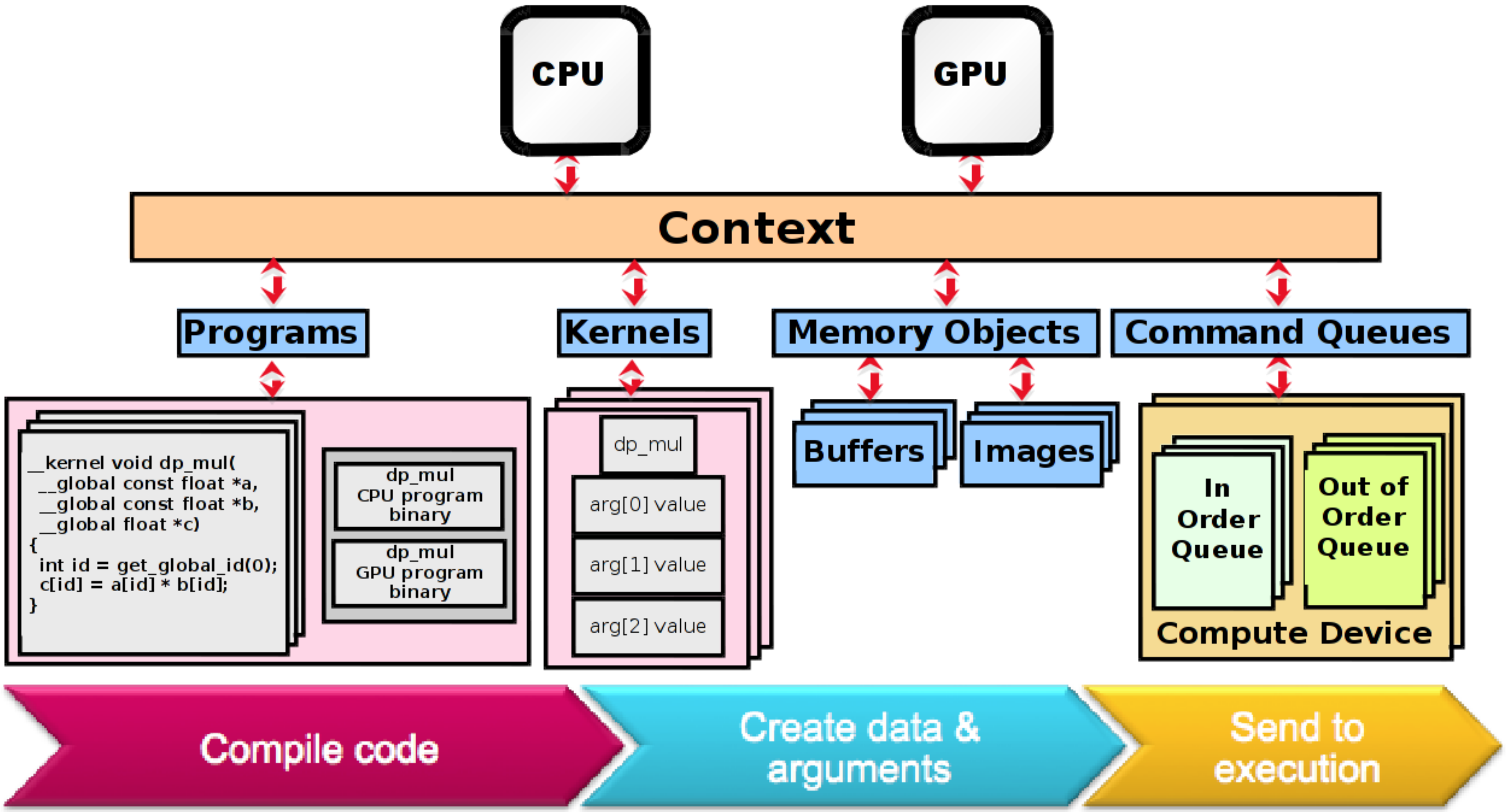
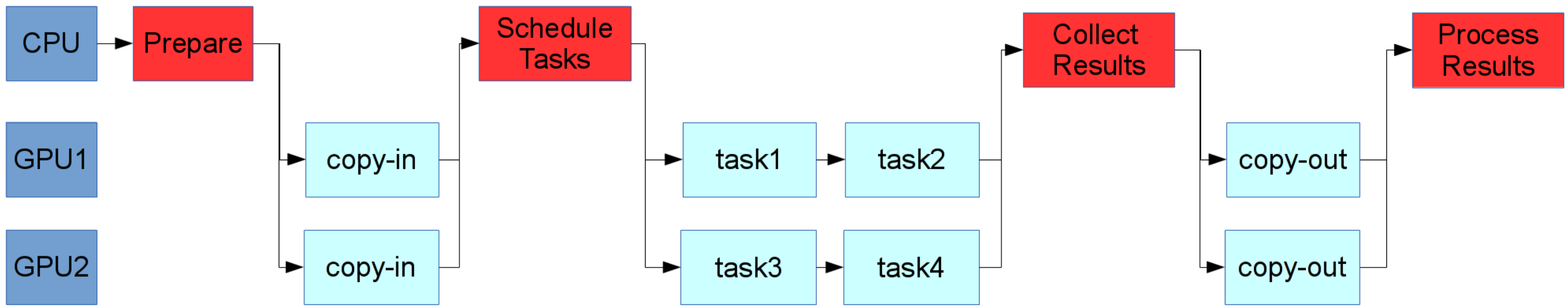
Karlsruhe Institute of Technology

GPU Programming Model



Basics of OpenCL and CUDA programming models

Execution Flow



- All GPUs are treated individually and controlled by CPU thread
  - GPU cores in case of double-core cards
- Initialize contexts
- Send data to internal GPU memory
- Execute one or more task (so called **kernels**) on GPUs
- Collect results back in system memory

OpenCL/CUDA Kernel

Karlsruhe Institute of Technology

The idea is to replace loops with functions (**kernels**) executing at each point in problem domain

```
void mul(...) {
  int i;
  for (i = 0; i < n; i++)
    out[i] = 2 * in[i];
}
```

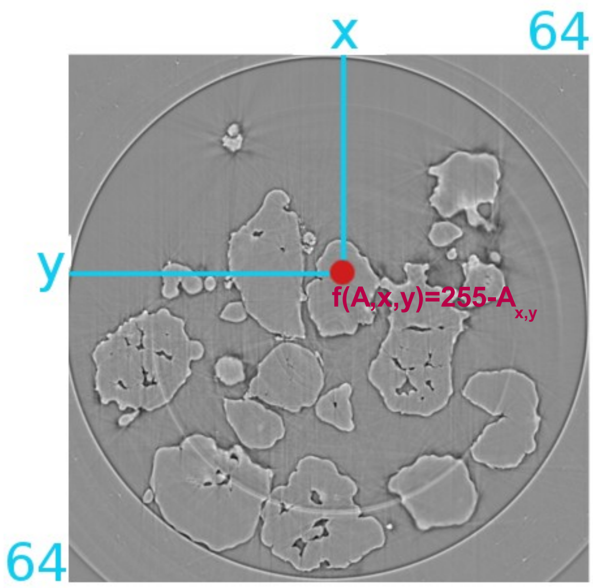
```
__kernel mul(...) {
      int  i = get_global_id(0);
      out[i] = 2 * in[i];
}
```

get_global_id(0)

10

Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Output	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50

Writing a Kernel



a) **Define a Grid of work-items (threads)**
- We select dimensionality (1D, 2D, or 3D) and number of threads along each dimension
- Number of threads is not restricted by available hardware. The OpenCL engine will execute as many threads in parallel as permited by hardware and will sequentialize others.

b) **Define a kernel function**
- Which locates data offsets using provided thread coordinates and perform computations

c) **Schedule kernel** on some data

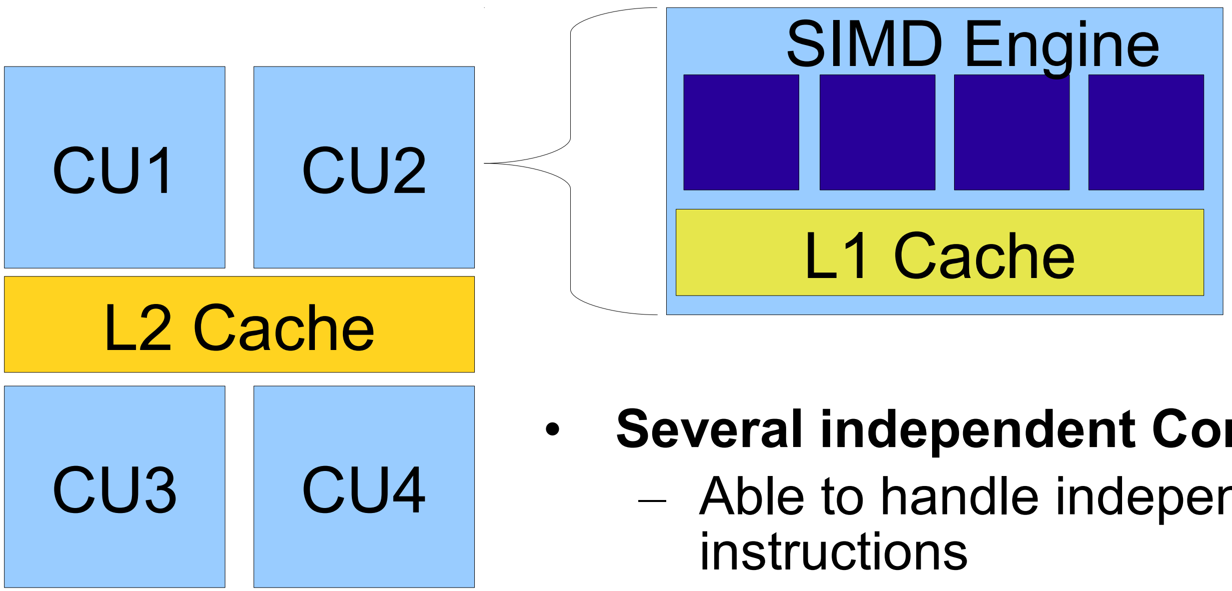
OpenCL Kernel

```
__kernel void invert(int width, __global float *res,
                     __global const float *img) {
    int i = get_global_id(0) + get_global_id(1) * width;
    res[i] = 255 - img[i];
}
```

CUDA Kernel

```
__global__ void invert(int width, float *res,
                       const float *img) {
    int i = threadIdx.x + threadIdx.y * width;
    res[i] = 255 - img[i];
}
```

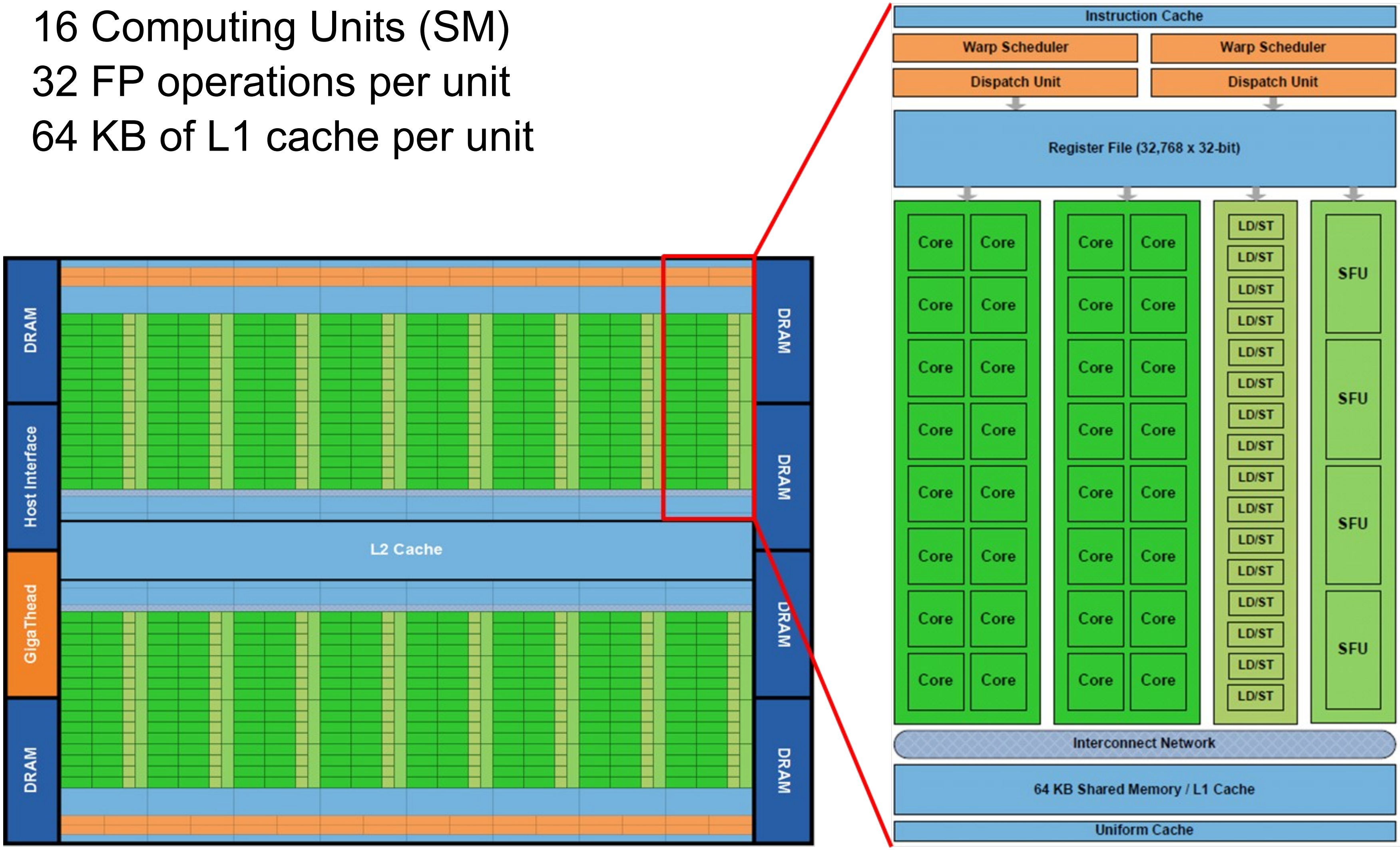
Hardware View



- **Several independent Compute Units**
  - Able to handle independent flow instructions
  - Share Global Memory and L2 Cache
- **SIMD Engine**
  - Issue a single instruction on multiple data items
  - Share L1 Cache

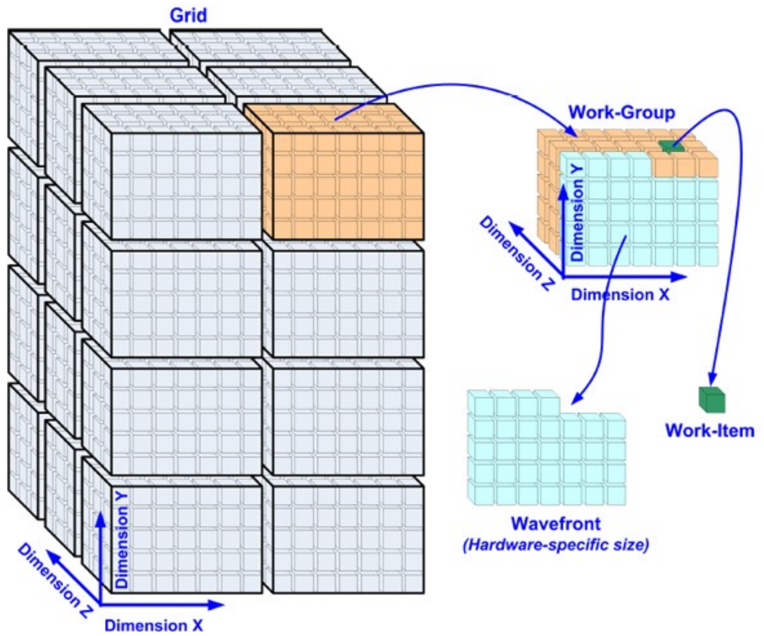
NVIDIA Fermi

- 16 Computing Units (SM)
- 32 FP operations per unit
- 64 KB of L1 cache per unit



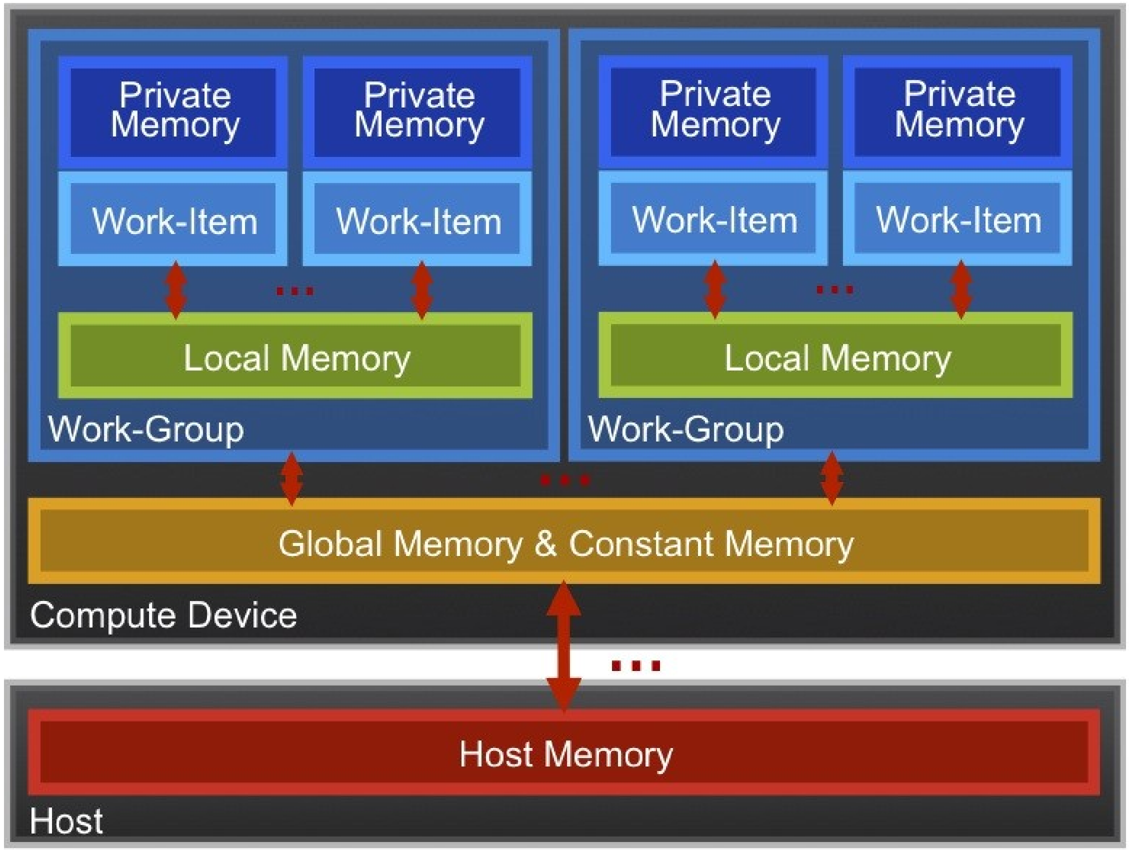
Task Grid

The idea is to allow efficient communication of work-items using L1 cache



- Task grid is split in **work-groups** (blocks in CUDA terminology)
- Work-group is executed on one of the Compute Units and may use the **local memory** (shared memory) of this unit to exchange data
- Work-group size is not limited by the width of SIMD engine (but limited by available register space). Each instruction is executed in several steps. This is used to hide memory latencies.
- CU always execute a block of work-items in parallel. This block is called **warp** (NVIDIA) or wavefront (AMD) and may be less when actual size of SIMD engine.
- The Compute Unit may schedule multiple work groups simultaneously

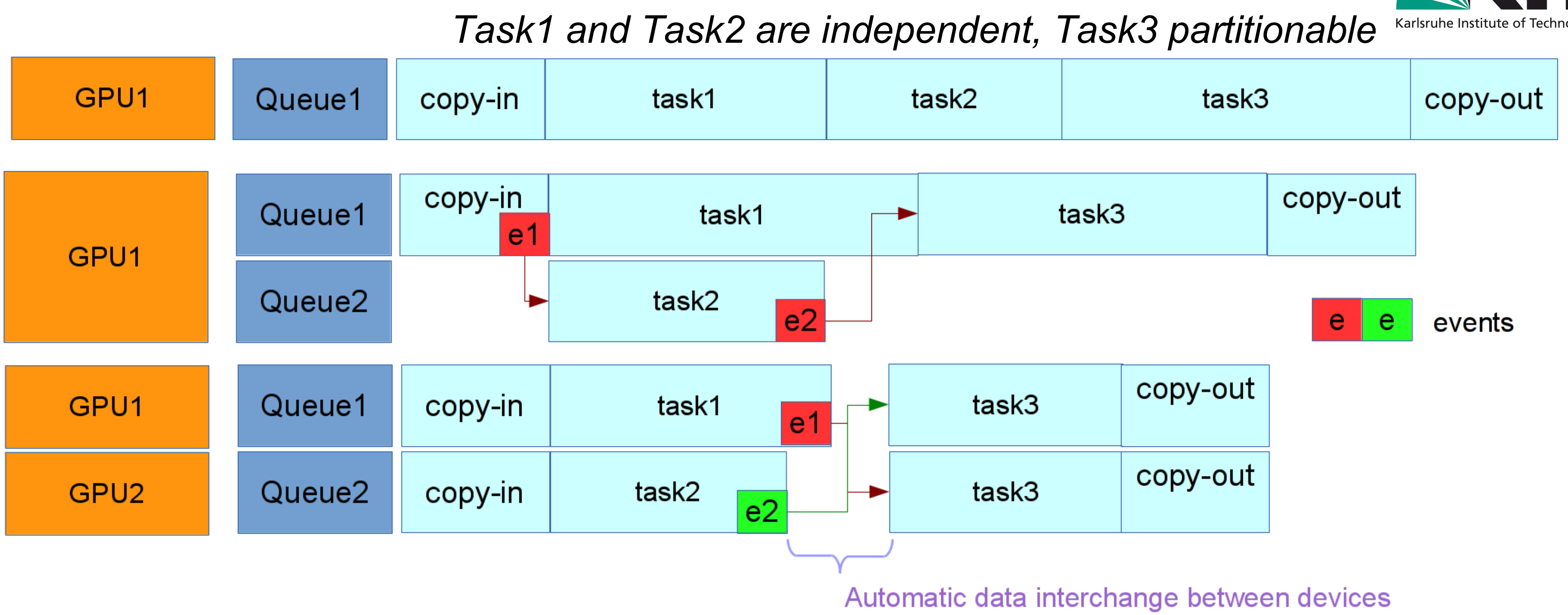
Memory model



- **Host Memory**
  6 GB/s (PCIe x16 gen2) to 12 GB/s (PCIe x16 gen3)
- **Global Memory**
  100 – 300 GB/s with latencies up to 1000 clocks
- **Local Memory**
  1 – 2 TB/s (total) with latencies below 100 clocks
- **Registers**
  - private to work-items

Complex memory hierarchy consisting of 4 levels and with each level one order of magnitude faster when previous!

Execution Model

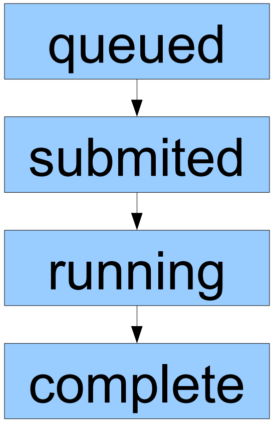


OpenCL **Queue** (CUDA Stream) and **Events** are synchronization primitives used to:

- Write asynchronous host code by scheduling multiple commands to the queue and waiting for completion
- Better utilize GPU resources while scheduling small Grids (supported only by some architectures)
- Handling of multiple GPU devices
- Get profiling informatation

Queues and Events

- **Events**
  - Each device command will trigger an event when finished
  - Device commands may have dependencies and will be only submited to device when all specified events are triggered. This is used to synchronize queues.
- **Queues**
  - Automatically distributed between available GPUs
  - Support In-order (default) and out-of-order execution
  - May synchronize using Events
- **Host code may**
  - Wait until complete queue finishes (*clFinish*)
  - Wait until some events are triggered (*clWaitForEvents*)
  - Register a callback on event (*clSetEventCallback*)
  - Create and trigger custom events to handle synchronization between host and queues (*clCreateUserEvent*, *clSetUserEventStatus*)
  - Get timestamps when device command entered each of 4 possible states (*clGetProfilingInfo*)



Building OpenCL Application

- Detect available platforms and devices
- Initialize OpenCL context for selected devices
- Compile OpenCL kernels for selected devices
- Create command queues
- Copy data to device memory
- Perform computations using one or several kernels
- Copy results back to system memory
- Clean-up

Creating Context

```
cl_int err;
cl_platform_id platform = 0;
cl_uint num_devices;
cl_device_id devices[16] = {0};
cl_command_queue queues[16] = {0};
cl_context ctx = 0;

err = clGetPlatformIDs(1, &platform, NULL);
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 16, devices, &num_devices);
ctx = clCreateContext(props, num_devices, &device, NULL, NULL, &err);

for (i = 0; i < num_devices; i++) {
        queues[i] = clCreateCommandQueue(ctx, devices[i], CL_QUEUE_PROFILING_ENABLE, &err);
}
```

Initializing all GPU devices from the first OpenCL platform and creating 1 command queue per device

Types of devices:

- CL_DEVICE_TYPE_CPU
- CL_DEVICE_TYPE_GPU
- CL_DEVICE_TYPE_ACCELERATOR

clGetDeviceInfo may be used to get device configuration

Compiling the Kernels

Loading kernel code

```
size_t len;
FILE * f = fopen("application.cl", "r");
fseek(f, 0, SEEK_END); len = ftell(f); fseek(f, 0, SEEK_SET);
char *source = (char*)malloc(len + 1);
fread(source + strlen(source), 1, len, f);
fclose(f);
```

Building code

```
const char *build_flags = "-cl-mad-enable";
cl_program app = clCreateProgramWithSource(ctx, 1, (const char**)&source, &len, &err);
err = clBuildProgram(app, num_devices, devices,  build_flags, NULL, NULL);
```

Instantiating kernels

```
cl_kernel kernel = clCreateKernel(app, "invert", &err);
```

OpenCL specification defines a number of build flags controlling allowed optimizations. The OpenCL platform may support additional flags defining the compiler behavior.

In case of threaded host code, a dedicated instance of cl_kernel may be needed for each queue in the context

Reporting Build Errors

```
size_t size
char build_log[4096];
cl_build_status build_status;

err = clBuildProgram(app, num_devices, devices,  build_flags, NULL, NULL);

do {
        err = clGetProgramBuildInfo(app, device, CL_PROGRAM_BUILD_STATUS,
                                    sizeof(build_status), &build_status, NULL);
} while (build_status == CL_BUILD_IN_PROGRESS);

if (build_status != CL_BUILD_SUCCESS) {
        err = clGetProgramBuildInfo(app, device, CL_PROGRAM_BUILD_LOG,
                                    sizeof(build_log) - 1, &build_log, &size);
        puts(build_log);
}
```

Building code

Waiting until build is complete

Printing build log in case of error

Different compilation results may be produced for each device

Managing Device Memory

```
float in[size * size];
float out[size * size];
```

Load source data on the host

```
cl_mem dev_in = clCreateBuffer(ctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    size * size * sizeof(float), in, &err);

cl_mem dev_out = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
                    size * size * sizeof(float), NULL, &err);
```

Allocate device memory and load source buffer

Execute computations...

Synchronous or asynchronous

```
err = clEnqueueWriteBuffer(queue, dev_out, CL_FALSE, 0,
                    size * size * sizeof(float), out, 0, NULL, NULL);
```

Copy results back to host

You don't need to specify at which devices to allocate memory, OpenCL will handle this automatically

clEnqueueReadBuffer may be used to send data to device after device memory was allocated

clEnqueueReadBufferRect and **clEnqueueWriteBufferRect** may be used to transfer parts of multidimensional array

Executing Kernels

Karlsruhe Institute of Technology

```
const size_t global_size[] = {size, size};
const site_t *local_size = NULL;
```

Define grid size, the work-group size determined automatically

```
clSetKernelArg(kernel, 0, sizeof(int), &size);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_in);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_out);
```

Set kernel arguments

Grid Dimension

Schedule the kernel

```
err = clEnqueueNDRangeKernel(queue, kernel, 2, 0, global_size, local_size, 0, NULL, &event);
```

events to wait

Wait for completion

```
clWaitForEvents(event);
```

Get kernel runtime in nanoseconds

```
cl_ulong start, end;
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL); +=
printf(“Kernel was executed in %lu ns\n”, end - start);
```

```
__kernel void  invert(int width, __global float *res,  __global const float *img) {
        int i = get_global_id(0) + get_global_id(1)  * width;
        res[i]  = 255 - img[i];
 }
```

CUDA Equivalent

```
float in[size * size];
float out[size * size];

float *dev_in, *dev_out;
cudaMalloc(&dev_in, size * size * sizeof(float));
cudaMalloc(&dev_out, size * size * sizeof(float));
cudaMemcpy(dev_in, in, size * size * sizeof(float), cudaMemcpyHostToDevice);

Dim3 blocks(16,16);
Dim3 grid(M/16, N/16);
invert<<<grid,blocks>>>(size, dev_out, dev_in);

cudaMemcpy(out, dev_out, size * size * sizeof(float), cudaMemcpyDeviceToHost);
```

Load source data on the host

Allocate device memory and copy source buffer

Define the grid and run the kernel.

Get results back

```
__global__ void invert(int width, float *res, const float *img) {
        int i = threadIdx.x + threadIdx.y * width;
        res[i] = 255 - img[i];
 }
```

Using PyOpenCL

```
import pyopencl as cl
import numpy

in = numpy.random.rand(size * size).astype(numpy.float32)
out = numpy.empty_like(a)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx, properties=cl.command_queue_properties.PROFILING_ENABLE)

src = open("application.cl", "r").read()
program = cl.Program(ctx, kernelSrc).build()

dev_in = cl.Buffer(ctx, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR, hostbuf=in)
dev_out = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, out.nbytes)

LocalWorkSize = ( 16, 16, )
globalWorkSize = ( size, size, )
event = program.invert(queue, global_size, local_size, size, dev_out, dev_in)

event.wait()
cl.enqueue_read_buffer(queue, out, dev_out).wait()

print("GPU execution time: %g ns" % (event.profile.end - event.profile.start))
```

Load source data on the host

Create context and queue

Building kernels

Allocating device memory

Define the grid and run the kernel.

Wait completion and get results back