Introduction to OpenCL

with GPGPUs

Sandra Wienke, M.Sc.
wienke@rz.rwth-aachen.de

PPCES 2012

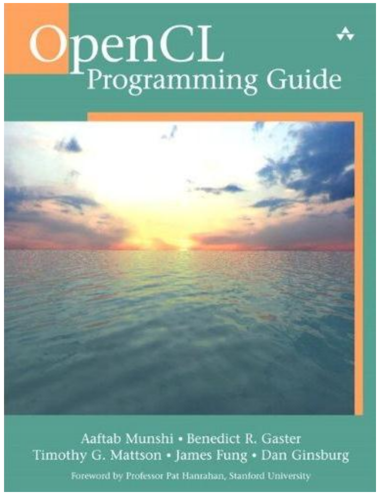
Links

- **General**
  - GPGPU Community: http://gpgpu.org/
  - GPU Computing Community: http://gpucomputing.net/
- **OpenCL**
  - Khronos Group (Specification, Reference Pages,…):

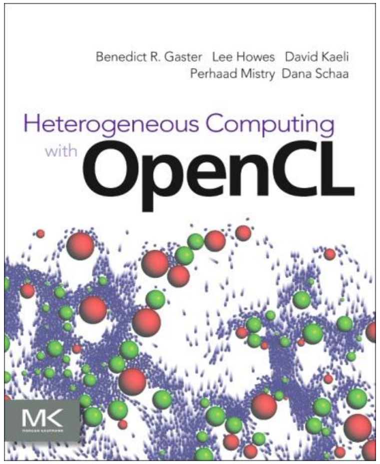
    http://www.khronos.org/opencl/
  - OpenCL + Nvidia http://developer.nvidia.com/opencl
  - OpenCL + AMD: http://developer.amd.com/zones/openclzone
  - OpenCL + Intel: http://software.intel.com/en-us/articles/opencl-sdk/

Books

- **A. Munshi, B. Gaster, T. Mattson, J. Fung, D. Ginsburg: *OpenCL Programming Guide* (2011)**

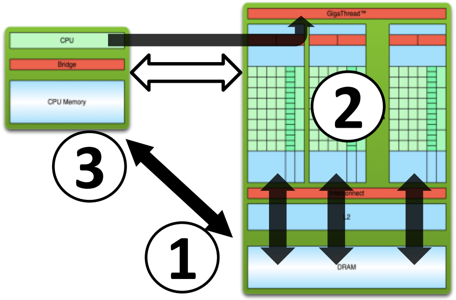


- **B. Gaster, D. Kaeli, L. Howes, P. Mistry, D. Schaa: *Heterogeneous Computing with OpenCL* (2011)**

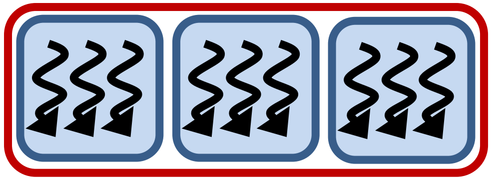


Contents

- **Processing flow**
  - Copy data from host to device
  - Execute GPU code (kernel) in parallel
  - Copy data from device to host



- **CUDA: Kernel executes grid of blocks of threads**
- **CUDA memory hierarchy on GPU**
  - Thread: registers, local
  - Block: shared
  - Grid: global



Paradigm *OpenCL*

= **Open Computing Language**

- **Portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors**
- **Open industry standard by Khronos group**

  AMD, NVIDIA, Intel, Apple, Ericsson, Nokia, IBM, Sony, EA, Freescale, TI,…
- **OpenCL C for Compute Kernels: Derived from ISO C99**
  - Restrictions: recursion, function pointers, …
  - Built-in data types/ functions
- **Timeline**
  - Jun'08: Launched, "strawman"
  - Dec'08: OpenCL 1.0 specification
  - Jun'10: OpenCL 1.1 specification
  - Nov'11: OpenCL 1.2 specification

  → Goal: a new OpenCL every 18 months

Example SAXPY

- SAXPY = Single-precision real Alpha X Plus Y: $\vec{y} = \alpha \cdot \vec{x} + \vec{y}$

```
void saxpyCPU(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f;
  float* x; float* y;
  x = (float*) malloc(n * sizeof(float));
  y = (float*) malloc(n * sizeof(float));

  // Initialize x, y
  for(int i=0; i<n; ++i){
    x[i]=i;
    y[i]=5.0*i-1.0;
  }

  // Invoke serial SAXPY kernel
  saxpyCPU(n, a, x, y);

  free(x); free(y);
  return 0;
}
```

Review: SAXPY for GPUs (CUDA C)

```
__global__ void saxpy_parallel(int n,
   float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (i < n){
        y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[]) {
 int n = 10240;
 float* h_x,*h_y; // Pointer to CPU memory
 // Allocate and initialize h_x and h_y

 float *d_x,*d_y; // Pointer to GPU memory
 cudaMalloc(&d_x, n*sizeof(float));
 cudaMalloc(&d_y, n*sizeof(float));
```

```
cudaMemcpy(d_x, h_x, n * sizeof(float),
  cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, n * sizeof(float),
  cudaMemcpyHostToDevice);

// Invoke parallel SAXPY kernel
dim3 threadsPerBlock(128);
dim3 blocksPerGrid(n/threadsPerBlock.x);
saxpy_parallel<<<blocksPerGrid,
  threadsPerBlock>>>(n, 2.0, d_x, d_y);

cudaMemcpy(h_y, d_y, n * sizeof(float),
  cudaMemcpyDeviceToHost);

cudaFree(d_x);
cudaFree(d_y);
free(h_x);
free(h_y);
return 0;
}
```

Example SAXPY

Outlook: SAXPY for GPUs (OpenCL)

```
#include <stdio.h>
#include <CL/cl.h>
const char* source[] = {
  "__kernel void saxpy_opencl(int n, float a, __global float*
     x, __global float* y)",
  "{",
  "  int i = get_global_id(0);",
  "  if( i < n ){",
  "    y[i] = a * x[i] + y[i];",
  "  }",
  "}"
};
int main(int argc, char* argv[]) {
  int n = 10240;  float a = 2.0;
  float* h_x, *h_y; // Pointer to CPU memory
  h_x = (float*) malloc(n * sizeof(float));
  h_y = (float*) malloc(n * sizeof(float));
  // Initialize h_x and h_y
  for(int i=0; i<n; ++i){
    h_x[i]=i; h_y[i]=5.0*i-1.0;
  }
  // Get an OpenCL platform
  cl_platform_id platform;
  clGetPlatformIDs(1,&platform, NULL);
  // Create context
  cl_device_id device;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device,
     NULL);
  cl_context context = clCreateContext(0, 1, &device, NULL,
     NULL, NULL);
  // Create a command-queue on the GPU device
  cl_command_queue queue = clCreateCommandQueue(context,
     device, 0, NULL);
// Create OpenCL program with source code
  cl_program program = clCreateProgramWithSource(context, 7, source,
     NULL, NULL);
  // Build the program
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
// Allocate memory on device on initialize with host data
  cl_mem d_x = clCreateBuffer(context, CL_MEM_READ_ONLY |
     CL_MEM_COPY_HOST_PTR, n*sizeof(float), h_x, NULL);
  cl_mem d_y = clCreateBuffer(context, CL_MEM_READ_WRITE |
     CL_MEM_COPY_HOST_PTR, n*sizeof(float), h_y, NULL);
  // Create kernel: handle to the compiled OpenCL function
  cl_kernel saxpy_kernel = clCreateKernel(program, "saxpy_opencl",
     NULL);
  // Set kernel arguments
  clSetKernelArg(saxpy_kernel, 0, sizeof(int), &n);
  clSetKernelArg(saxpy_kernel, 1, sizeof(float), &a);
  clSetKernelArg(saxpy_kernel, 2, sizeof(cl_mem), &d_x);
  clSetKernelArg(saxpy_kernel, 3, sizeof(cl_mem), &d_y);
  // Enqueue kernel execution
  size_t threadsPerWG[] = {128};
  size_t threadsTotal[] = {n};
  clEnqueueNDRangeKernel(queue, saxpy_kernel, 1, 0, threadsTotal,
     threadsPerWG, 0,0,0);
  // Copy results from device to host
  clEnqueueReadBuffer(queue, d_y, CL_TRUE, 0, n*sizeof(float), h_y,
     0, NULL, NULL);
  // Cleanup
  clReleaseKernel(saxpy_kernel);
  clReleaseProgram(program);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);
  clReleaseMemObject(d_x); clReleaseMemObject(d_y);
  free(h_x); free(h_y); return 0;
}
```

Contents

Programming model

- **Data parallel**
  - Sequence of instructions is applied to multiple data elements
  - Hierarchical data parallel programming model
- **Task parallel**
  - Single instance of kernel executes e.g. multiple tasks
- **Hybrids possible**

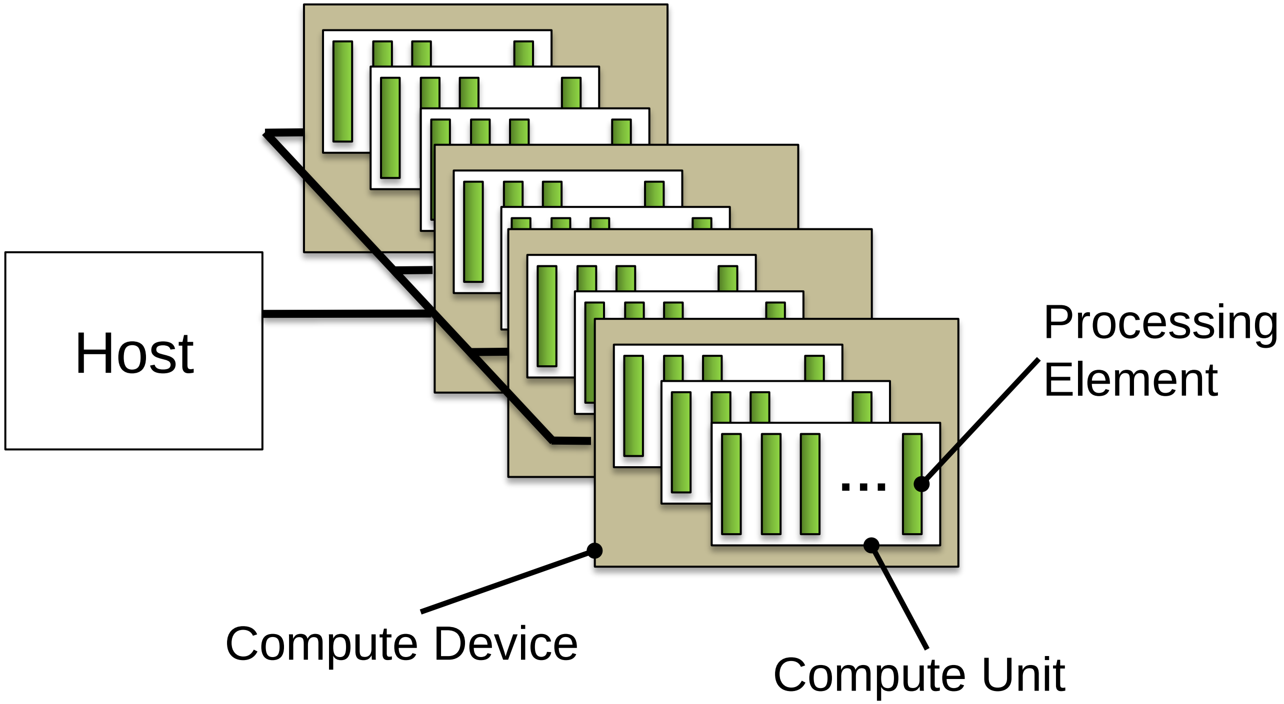
- **SPMD = Single program multiple data**

CUDA	OpenCL
Pointer (host)	Handles (host)
Pointer (device)	Pointer (device)
CUDA C Runtime API less verbose	Verbose (similar to CUDA C Driver API)
Offline compilation	JIT

Contents

Platform Model

CUDA	OpenCL
Core	Processing Element (PE)
Multiprocessor	Compute Unit (CU)
GPU/ Device	Compute Device



Contents

Execution model

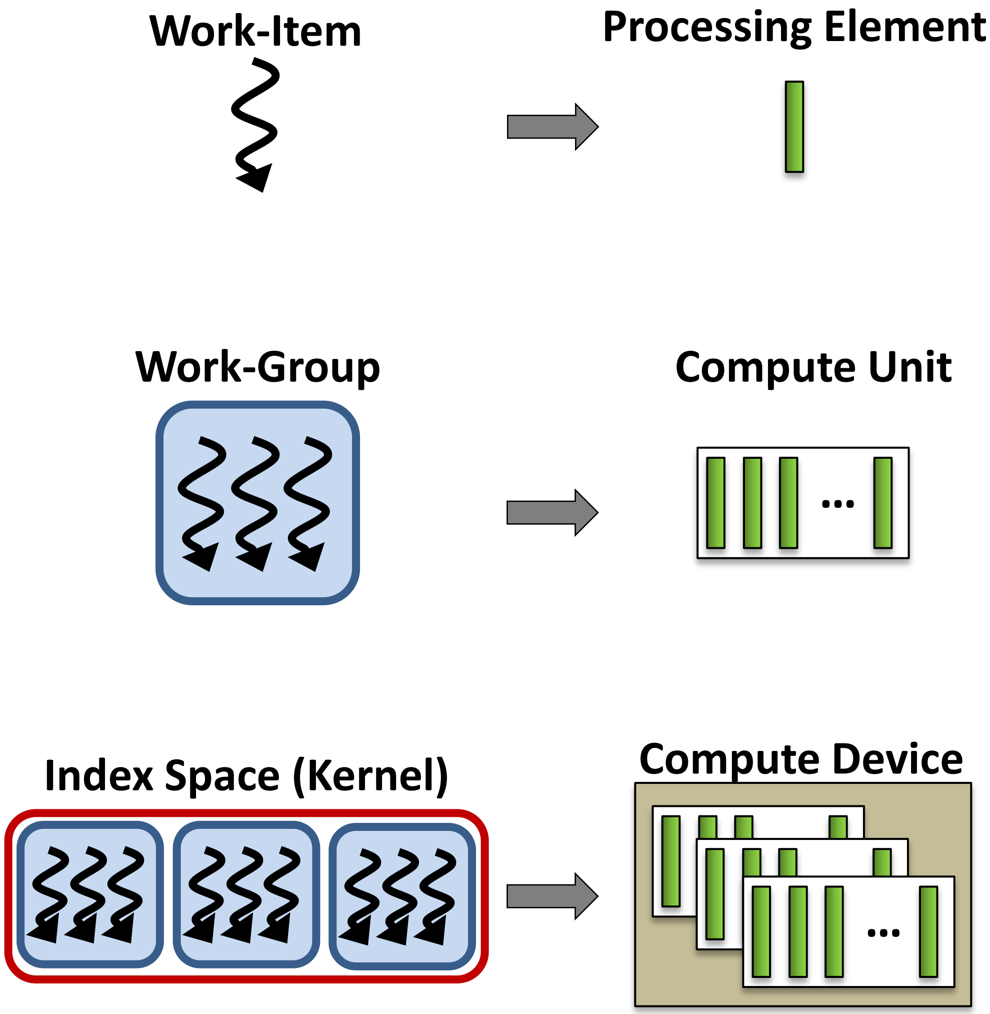
- **Host-directed execution model**
  - Executes kernel on device
- ***Work-items*** **are grouped into** ***work-groups***
  - Synchronization within work-group possible

CUDA	OpenCL
Thread	Work-Item
Block	Work-Group
Grid	Index Space/ NDRange

- **Kernel executes for each work-item within an** ***index space***
  - *NDRange* = n-dimensional index space
- **Indices** (examples for one dimension)

CUDA	OpenCL
`threadIdx.x`	`get_local_id(0)`
`blockIdx.x`	`get_group_id(0)`
`blockIdx.x*blockDim.x+threadIdx.x`	`get_global_id(0)`
`blockDim.x`	`get_local_size(0)`
`gridDim.x`	`get_num_groups(0)`
`blockDim.x*gridDim.x`	`get_global_size(0)`

Execution model



- Each work-item is executed by a processing element
- Each work-group is executed on a compute unit
- Several concurrent work-groups can reside on one compute unit depending on memory resources
- Each kernel is executed on a compute device

Execution model (OpenCL)

green = background information OpenCL

- **Setup GPU (e.g. driver)**
- **Download + install OpenCL libraries**

  Nvidia (GPU), AMD (GPU+CPU), Intel (CPU),… (cf. “Links” section)
- **OpenCL API**

  (cf. “Links” section)
- **Includes**

```
#include <CL/opencl.h>
```

- **Compiling + linking**

  Use default Intel Compiler or *# on our cluster*

```
module switch intel gcc          # on our cluster
$CXX saxpy.cl -lOpenCL           # $CXX: e.g. icpc or g++
```

Execution model (OpenCL)

Kernel code

- Function qualifier: `__kernel`
- IDs: `get_global_id(dim)`, `get_local_id(dim)`, `get_group_id(dim)`
- Save as char array or read in from file

Kernel usage

- Create program handle from kernel code: Just-In-Time (JIT) compilation

```
cl_progam program = clCreateProgramWithSource (context,
   srcSize, source,…, err);
```

```
clBuildProgram(program,…, options,…);
```

```
options, e.g.:
-D name #preprocessor
-cl-fast-relaxed-math
```

- Create kernel handle

```
cl_kernel kernel = clCreateKernel(program,kernelName,err);
```

- Set kernel arguments

```
clSetKernelArg(kernel, argIdx, argSize, argVal);
```

Example SAXPY: Kernel

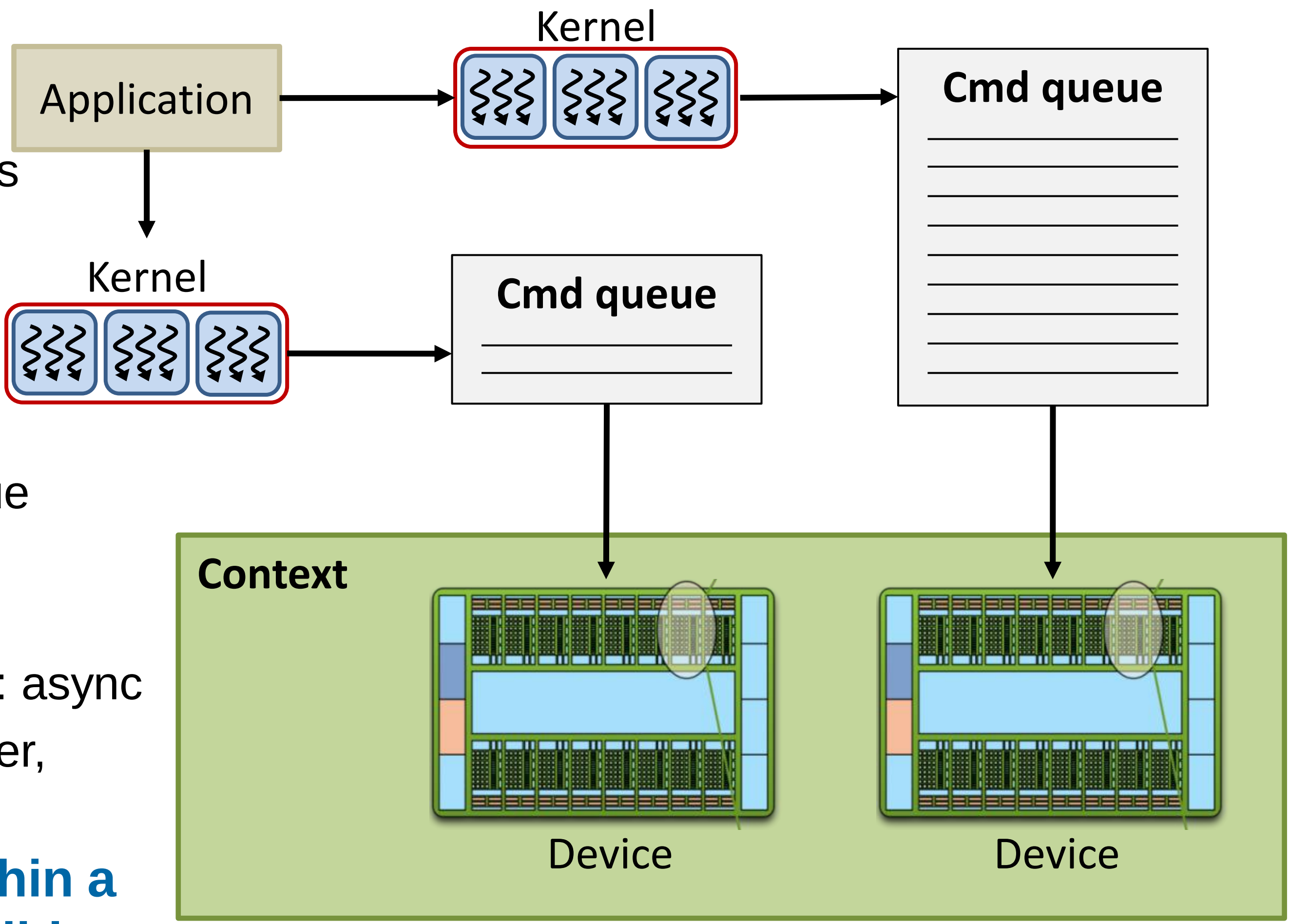
```
const char* source[] = {
  "__kernel void saxpy_opencl(int n, float a,__global const float* x, __global
                                                                float* y)",
  "{",
  "   int i = get_global_id(0);",
  "   if( i < n ){",
  "      y[i] = a * x[i] + y[i];",
  "  }",
  "}"
};
```

Example SAXPY: Kernel usage

```
int main(int argc, char* argv[]) {
  [..]
  // Create OpenCL program with source code
  cl_program program = clCreateProgramWithSource(context, 7, source, NULL, NULL);

  // Build the program (JIT)
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

  // Create kernel: handle to the compiled OpenCL function
  cl_kernel saxpy_kernel = clCreateKernel(program, "saxpy_opencl", NULL);

  // Set kernel arguments
  clSetKernelArg(saxpy_kernel, 0, sizeof(int), &n);
  clSetKernelArg(saxpy_kernel, 1, sizeof(float), &a);
  clSetKernelArg(saxpy_kernel, 2, sizeof(cl_mem), &d_x);
  clSetKernelArg(saxpy_kernel, 3, sizeof(cl_mem), &d_y);
 [..]
}
```

Execution model: Context & Queues

- **Context**
  - Includes i.a. devices
- **Command Queue**
  - Assigned to single device within a context
  - Commands in queue are executed
  - → Scheduling
    - Host ⇔ Device: async
    - Relative: in-order, out-of-order
- **Multiple queues within a single context possible**
- **Synchronization between commands enqueued to command-queue(s) in single context possible**



Execution model (OpenCL)

- **Request platform**

```
cl_platform platform;
clGetPlatformIDs(1, &platform, …)
```

- **Request device**

```
cl_device_id device;
clGetDeviceIDs(platform, type, 1,&device,…)
```

```
type, e.g.:
CL_DEVICE_TYPE_GPU
CL_DEVICE_TYPE_CPU
CL_DEVICE_TYPE_ALL
```

- **Create context**

```
cl_context context = clCreateContext(…,1, &device,…)
```

- **Create command queue**

```
cl_command_queue queue = clCreateCommandQueue(context,
      device, props, err)
```

```
props, e.g.:
CL_QUEUE_OUT_OF_ORDER_
EXEC_MODE_ENABLE
(default: in-order)
```

- **Execute kernel**

```
clEnqueueNDRangeKernel(queue, kernel, numDims,…, gridDims,
      blockDims,…)
(clFinish(queue))
```

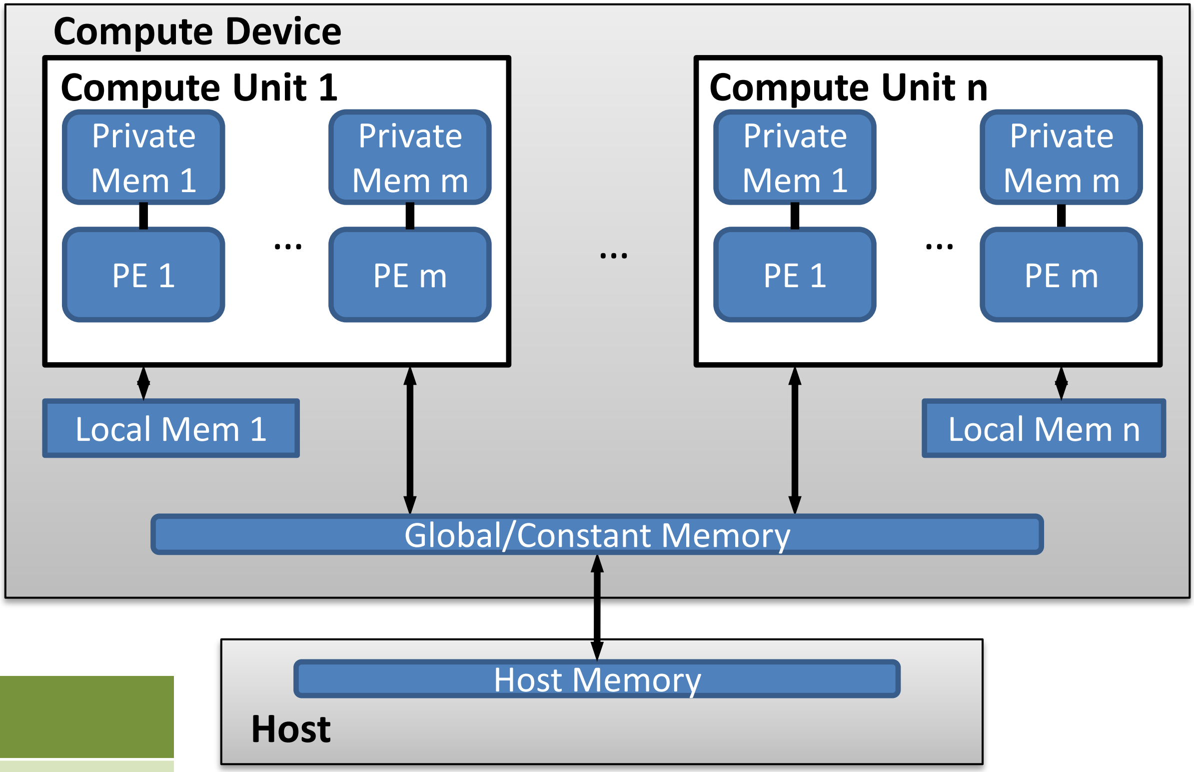
Example SAXPY: Context & Queue

```
int main(int argc, char* argv[]) {
  [..]
   // Get an OpenCL platform
   cl_platform_id platform;
   clGetPlatformIDs(1,&platform, NULL);

   // Create context
   cl_device_id device;
   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
   cl_context context = clCreateContext(0, 1, &device, NULL, NULL, NULL);

   // Create a command-queue on the GPU device
   cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

   // Create kernel, set kernel arguments [..]
   // Enqueue kernel execution
   size_t threadsPerWG[] = {128};
   size_t threadsTotal[] = {n}; // work items

   clEnqueueNDRangeKernel(queue, saxpy_kernel, 1, 0, threadsTotal, threadsPerWG,
     0,0,0);
  [..]
}
```

Contents

Memory model

- **Work-Item**
  - *Private* memory
- **Work-Group**
  - *Local* memory:
- **Kernel/ application**
  - *Constant* memory
  - *Global* memory



CUDA	OpenCL
Local memory	Private memory
Shared memory	Local memory
Constant memory	Constant memory
Global memory	Global memory

Attention! Difference!

Memory model (OpenCL)

- **Kernel code**
  - Address space qualifiers: `__global`, `__constant`, `__local`, `__private`
  - Used in: Variable declarations, function arguments
- **Memory management**

```
cl_mem buf = clCreateBuffer(context, memFlags, bufSize,
pointerToCPUMem, err)
clReleaseMemObject(buf)
```

```
memFlags, e.g.:
CL_MEM_READ_WRITE,
CL_MEM_WRITE_ONLY,
CL_MEM_READ_ONLY,
CL_MEM_COPY_HOST_PTR
```

- **Memory transfer CPU – GPU**

"→" `clEnqueueWriteBuffer(queue, buf, blocking,…, bufSize, pointerToCPUMem,…)`

```
blocking:
CL_TRUE:  blocking,
CL_FALSE: non-blocking
```

"←" `clEnqueReadBuffer(queue, buf, blocking,…, bufSize, pointerToCPUMem,…)`

Example SAXPY: Memory

```
const char* source[] = {
"__kernel void saxpy_opencl(int n, float a,__global const float* x, __global
                                                              float* y)",[..]}
int main(int argc, char* argv[]) {
  [..]
  float* h_x, *h_y; // Pointer to CPU memory
  h_x=(float*) malloc(n*sizeof(float)); h_y=(float*) malloc(n*sizeof(float));
  // Initialize h_x and h_y
  // Create context, command queue, program, kernel

  // Allocate memory on device on initialize with host data
  cl_mem d_x = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
       n*sizeof(float), h_x, NULL);
  cl_mem d_y = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
       n*sizeof(float), h_y, NULL);
  // Execute kernel

 // Copy results from device to host
  clEnqueueReadBuffer(queue, d_y, CL_TRUE, 0, n*sizeof(float), h_y, 0, NULL,
       NULL);
[..]
}
```

Contents

Summary

	CUDA	OpenCL
	Platform model	
*	Core	Processing Element (PE)
**	Multiprocessor	Compute Unit (CU)
***	GPU/ Device	Compute Device
	Execution model	
*	Thread	Work-Item
**	Block	Work-Group
***	Grid	Index Space/ NDRange
	Memory model	
*	Local memory	Private memory
**	Shared memory	Local memory
***	Global memory	Global memory

Stars map hardware, (logical) execution unit and memory space

RWTH AACHEN UNIVERSITY

5 steps for a basic program with OpenCL

```
#include <stdio.h>
#include <CL/cl.h>
const char* source[] = {
  "__kernel void saxpy_opencl(int n, float a, __global float*
     x, __global float* y)",
  "{",
  "  int i = get_global_id(0);",
  "  if( i < n ){",
  "    y[i] = a * x[i] + y[i];",
  "  }",
  "}"
};
int main(int argc, char* argv[]) {
  int n = 10240;  float a = 2.0;
  float* h_x, *h_y; // Pointer to CPU memory
  h_x = (float*) malloc(n * sizeof(float));
  h_y = (float*) malloc(n * sizeof(float));
  // Initialize h_x and h_y
  for(int i=0; i<n; ++i){
    h_x[i]=i; h_y[i]=5.0*i-1.0;
  }
  // Get an OpenCL platform
  cl_platform_id platform;
     NULL);
  cl_context context = clCreateContext(0, 1, &device, NULL,
     NULL, NULL);
  // Create a command-queue on the GPU device
  cl_command_queue queue = clCreateCommandQueue(context,
     device, 0, NULL);
```

1. Define the platform (= devices + context + queues)

2. Create + built the program

```
  // Build the program
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

3. Setup memory objects

```
  // Allocate memory on device on initialize with host data
     CL_MEM_COPY_HOST_PTR, n*sizeof(float), h_y, NULL);
```

4. Define kernel (attach kernel function to arguments)

```
  // Create kernel: handle to the compiled OpenCL function
  clSetKernelArg(saxpy_kernel, 1, sizeof(float), &a);
  clSetKernelArg(saxpy_kernel, 2, sizeof(cl_mem), &d_x);
  clSetKernelArg(saxpy_kernel, 3, sizeof(cl_mem), &d_y);
```

5. Submit commands: move memory objects and execute kernels

```
  // Enqueue kernel execution
     0, NULL, NULL);
  // Cleanup
  clReleaseKernel(saxpy_kernel);
  clReleaseProgram(program);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);
  clReleaseMemObject(d_x); clReleaseMemObject(d_y);
  free(h_x); free(h_y); return 0;
}
```

Contents

Tools

- **Debugger**
  - gDEbugger — GPU (memory) debugger, Windows (currently), integrated in Visual Studio, AMD (currently free of charge)
  - Intel Debugger — CPU debugger, Windows, integrated in Visual Studio, Intel (free of charge)
- **Profiling/ tracing**
  - Visual Profiler — Performance analysis w/ HW counters, NVIDIA (free of charge)
  - Parallel Nsight — API & kernel tracing, Windows, integrated in Visual Studio, NVIDIA (free of charge)
  - VampirTrace — Performance monitoring (tracing), TU Dresden

Libraries

- **ViennaCL** Linear algebra: BLAS, FFT, iterative solvers
- **ArrayFire** E.g. sort, sum

- **Upcoming**
  - MAGMA Dense linear algebra (subset of BLAS, LAPACK)