OpenCL Basics

Parallel Computing on GPU and CPU

23. März 2011 | Willi Homberg

Agenda

- Introduction
- OpenCL architecture
  - Platform model
  - Execution model
  - Memory model
  - Programming model
- Platform layer
- Runtime
- Language, compiler
- Example: Vector addition

Introduction

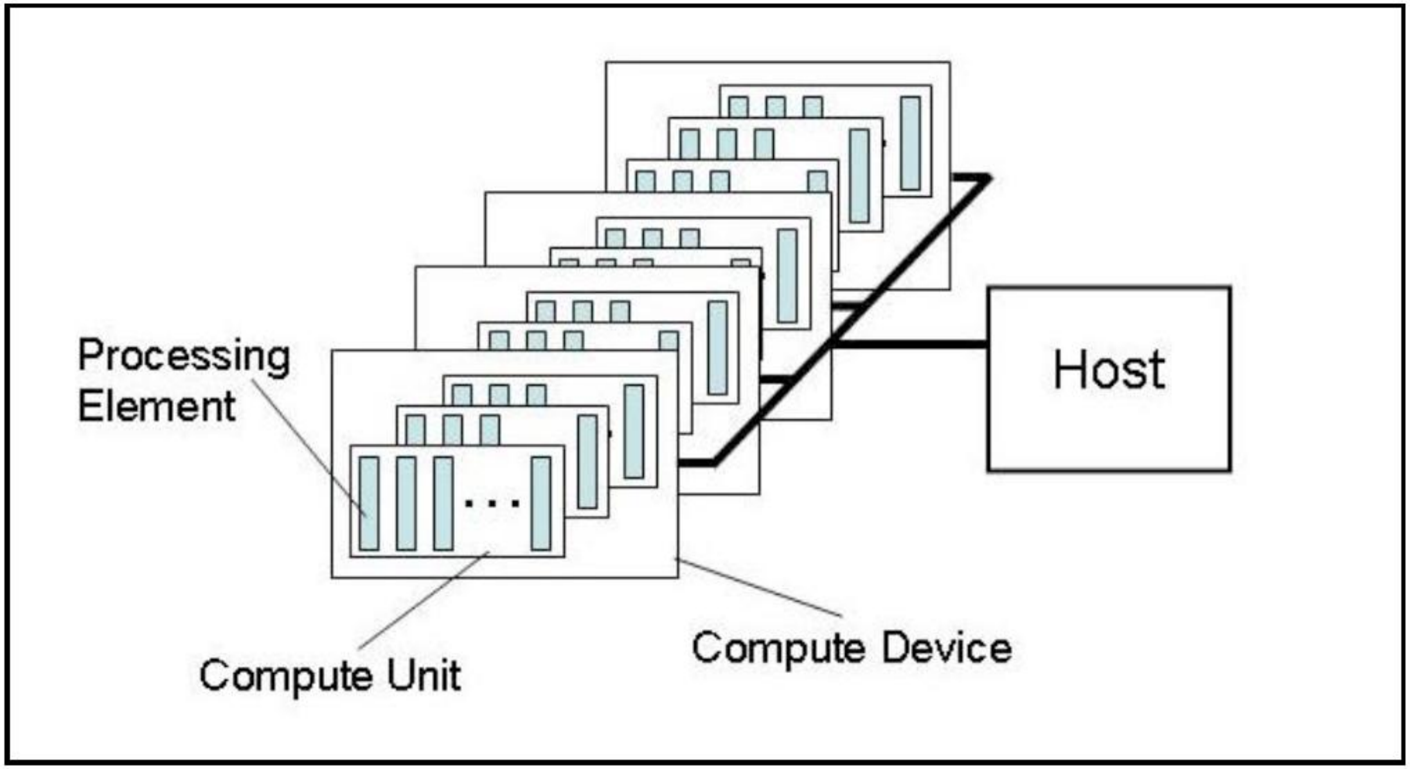
- OpenCL – Open Computing Language
  - Open, royalty-free standard
  - For cross-platform, parallel programming of modern processors
  - An Apple initiative
  - Approved by NVIDIA, Intel, AMD, IBM, ...
  - Specified by the Khronos group
- Intended for accessing heterogeneous computational resources
  - CPUs (Intel, AMD, IBM Cell BE, ...)
  - GPUs (Nvidia GTX & Tesla/Fermi, AMD/ATI 58xx, …)
  - Future processors with integrated graphics chip (Sandy Bridge, Llano)
- Difference to CUDA or CAL/IL
  - Code hardware agnostic, portable

OpenCL Information

- Khronos OpenCL - http://www.khronos.org/opencl
  - Specification and headers files
  - Online manual pages, quick reference card
- MacResearch OpenCL tutorials - http://www.macresearch.org/opencl
- NVIDIA devel. zone: OpenCL - http://developer.nvidia.com/object/opencl.html
  - Best practices guide: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf
- AMD/ATI – ATI Stream Computing:
  - http://ati.amd.com/technology/streamcomputing/opencl.html
  - SDK - http://developer.amd.com/gpu/ATIStreamSDK
- IBM OpenCL for Linux on Power - http://www.alphaworks.ibm.com/tech/opencl

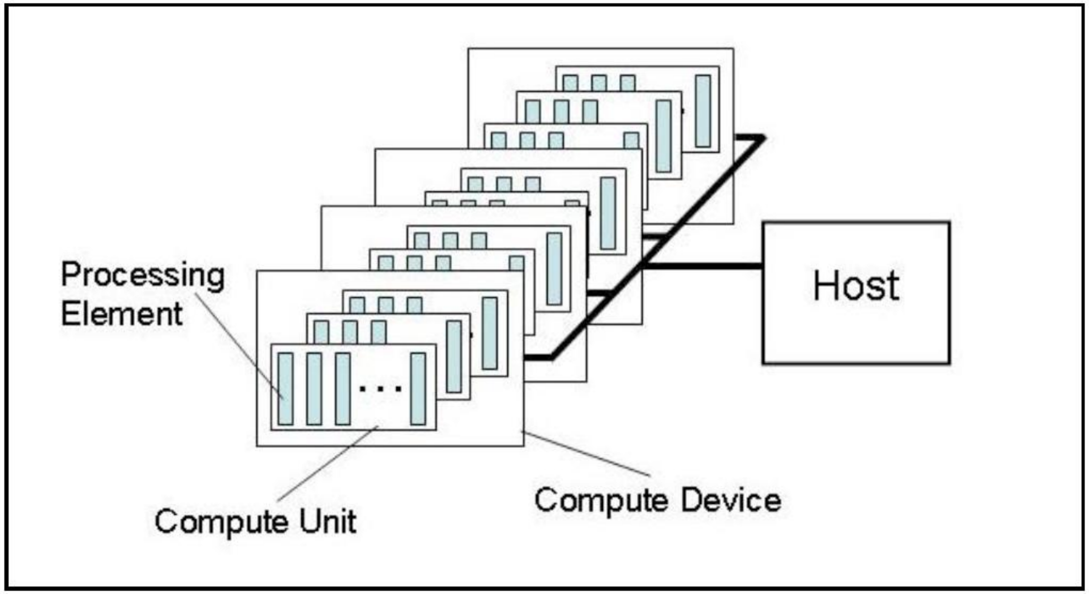
OpenCL Architecture: Platform Model

- A host is connected to one or more (possibly heterogeneous) OpenCL devices
  - A device is divided into compute units (CUs)
    - CUs are subdivided into processing elements (PEs)



OpenCL Architecture: Platform Model

- An OpenCL application runs on the host
  - Submits commands to execute on the PEs within a device
  - PEs execute a single stream of instructions as
    - SIMD units
      - In lockstep with a single stream of instructions
    - SPMD units
      - PEs maintain own program counter



JuGiPSy – Juelich's GPU System

- NVIDIA Tesla 1070
  - 4 x Tesla C1060 GPU
  - 4 x 4 GB DDR3 memory
  - 4 x 102 GB/s memory bandwidth
  - 4 TFLOPS single-precision
  - 346 GFLOPS double-precision
  - 700W
- Host System (2x)
  - Xeon E5430@2,66 GHz quad-core
  - 32 GB memory, 32 KB data cache
  - 1 TB disk

Bandwidth:	MB/s
Host to Device:	2157.0
Device to Host:	1886.4
Device to Device:	73458.1

2x PCIe 16x

Transtec 1000R S1070

JUGIPSY

- Network
  - External interface: jugipsy.zam.kfa-juelich.de
  - Internal private 1 GE network:
    - gipsy1, gipsy2
    - each connected to 2 GPUs
- Software
  - OpenSuSE 11.1
  - NVIDIA
    - NVIDIA_GPU_Computing_SDK (CUDA-3.1)
  - AMD / ATI
    - ati-stream-sdk-v2.2-lnx64

JUDGE – Juelich Dedicated GPU Environment

Heterogeneous cluster: 60 TFLOPS peak

54 IBM System x iDataPlex dx360 m3

each:

- 2 NVIDIA Tesla M2050 (Fermi)

  1,15 GHz (448 cores), 3 GB memory
  - 148 GB/s memory bandwidth
  - 1,03 TFLOPS single-precision
  - 515 GFLOPS double-precision
  - 225W
- 2 Intel Xeon X5650(Westmere)

  6-core processor 2,66 GHz

  96 GB memory

Bandwidth:	MB/s
Host to Device:	3422.5
Device to Host:	2911.1
Device to Device:	85636.2

JUGIPSY: OpenCL Devices

```
PLATFORM NAME :  ATI Stream          PLATFORM VERSION : OpenCL 1.0 ATI-Stream-v2.1 (145)
PLATFORM PROFILE : FULL_PROFILE
PLATFORM VENDOR : Advanced Micro Devices, Inc.
1 devices found supporting OpenCL
 Device name :Intel(R) Xeon(R) CPU           E5430  @ 2.66GHz
 Max Compute Units : 4
 Amount of Global Memory : 3221225472 bytes
 Amount of Local Memory : 32768 bytes
 Max Work Group Size : 1024
 Max Work Item Dimensions : 3
 Max Work Items on dimemsion 0 : 1024
 Max Work Items on dimemsion 1 : 1024
 Max Work Items on dimemsion 2 : 1024
================================================
PLATFORM NAME :  NVIDIA CUDA      PLATFORM VERSION : OpenCL 1.0 CUDA 3.0.1
PLATFORM PROFILE : FULL_PROFILE
PLATFORM VENDOR : NVIDIA Corporation
2 devices found supporting OpenCL
 Max Compute Units : 30
 Amount of Global Memory : 4294770688 bytes
 Amount of Local Memory : 16384 bytes
 Max Work Group Size : 512
 Max Work Item Dimensions : 3
 Max Work Items on dimemsion 0 : 512
 Max Work Items on dimemsion 1 : 512
 Max Work Items on dimemsion 2 : 64 ....................................
```

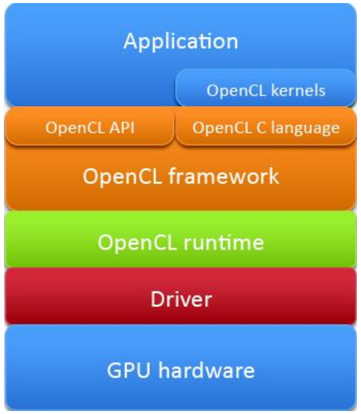
JUDGE: OpenCL Devices

```
Device 0: "Tesla M2050"
  CUDA Driver Version:                           3.20
  CUDA Runtime Version:                          3.20
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 2817982464 bytes
  Multiprocessors x Cores/MP = Cores:            14 (MP) x 32 (Cores/MP) = 448 (Cores)
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.15 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads
                                                 can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                Yes
  Device is using TCC driver mode:               No
Device 1: "Tesla M2050"
```

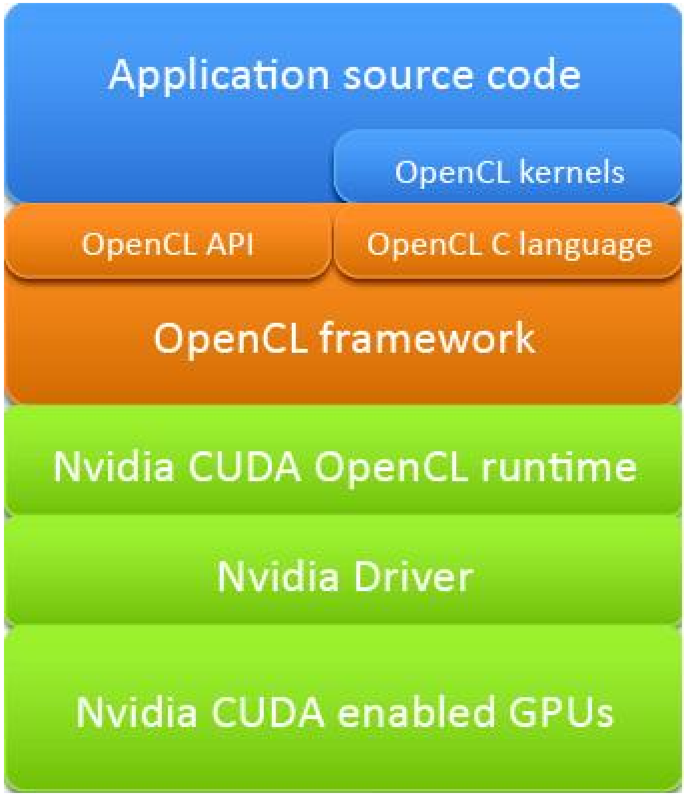
OpenCL Architecture: Execution Model

- Compute Kernel
  - Runs on the OpenCL device
  - Basic unit of executable code (similar to a C function)
  - Data-parallel or task-parallel
- Compute Program
  - Collection of compute kernels and internal functions
  - Analogous to a dynamic library
- Applications
  - Run on the host
  - Queue compute kernel execution instances
  - Instances are executed in-order or out-of-order
  - Events are used to implement appropriate order of execution

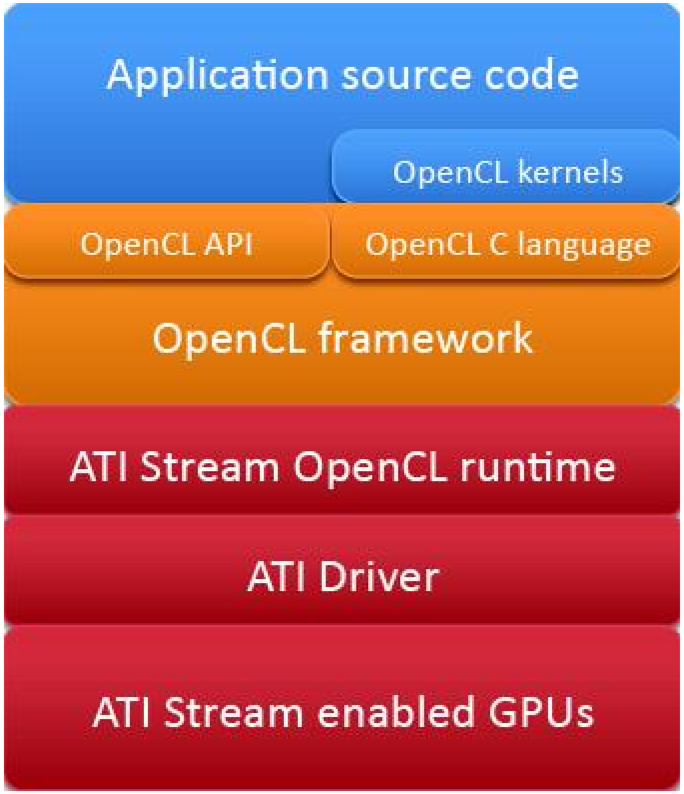
OpenCL Stack (GPUs)



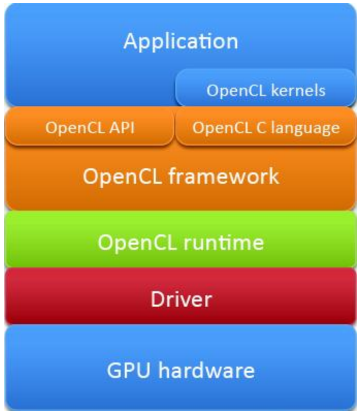
NVIDIA-GPU



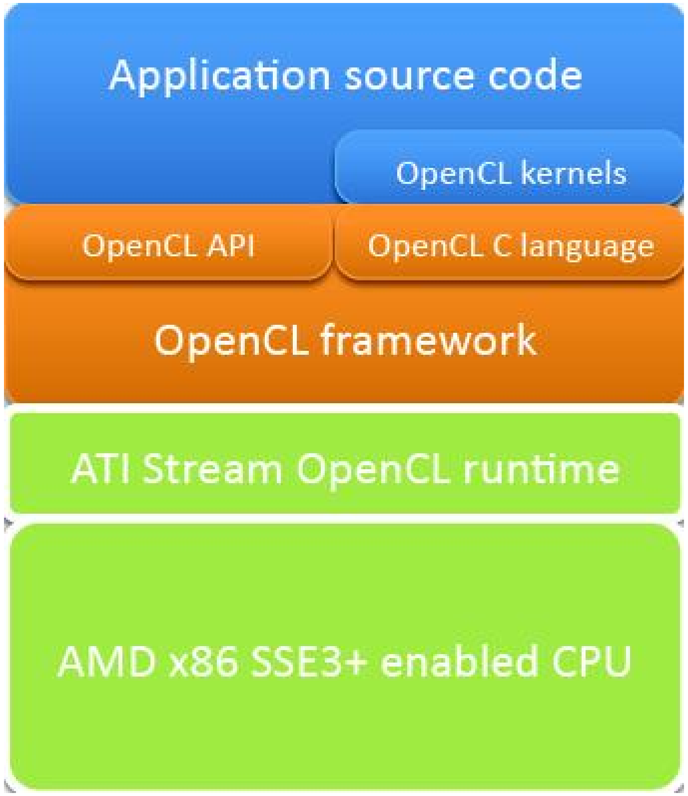
AMD-GPU



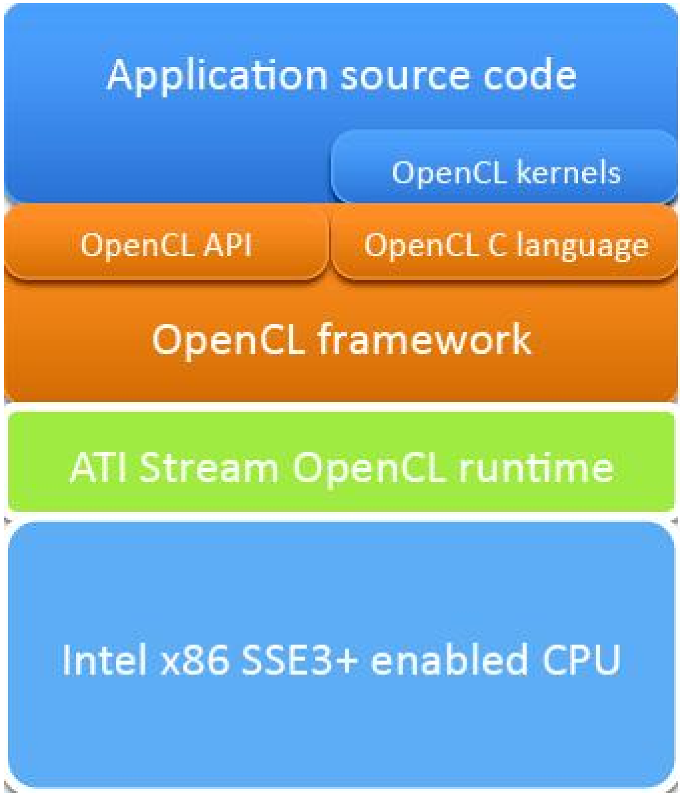
OpenCL Stack (CPUs)



AMD-CPU



Intel-CPU



OpenCL Architecture: Execution Model

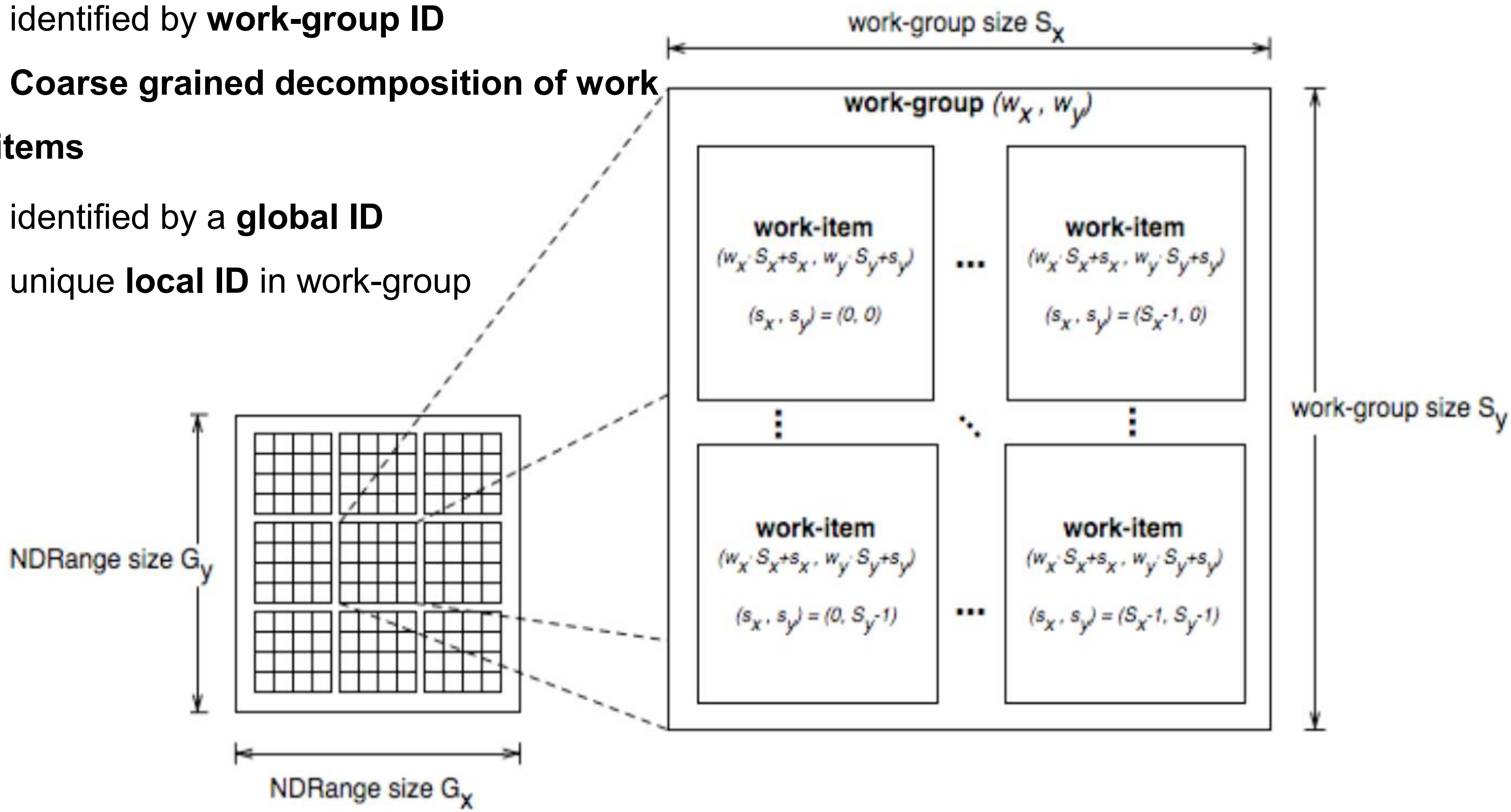
Program execution occurs in two parts

- Host program executes on the host
  - Defines contexts for kernels and manages execution
  - Data parallel programming model
    - NDRange index space (1-, 2- or 3-dimensional)
  - Task parallel programming model

- Kernels execute on one or more devices
  - An instance of the kernel is called **work-item** (CUDA: thread)
    - Executes for each point in index space
  - Organized in **work-groups** (CUDA: blocks)

OpenCL Architecture: Execution Model

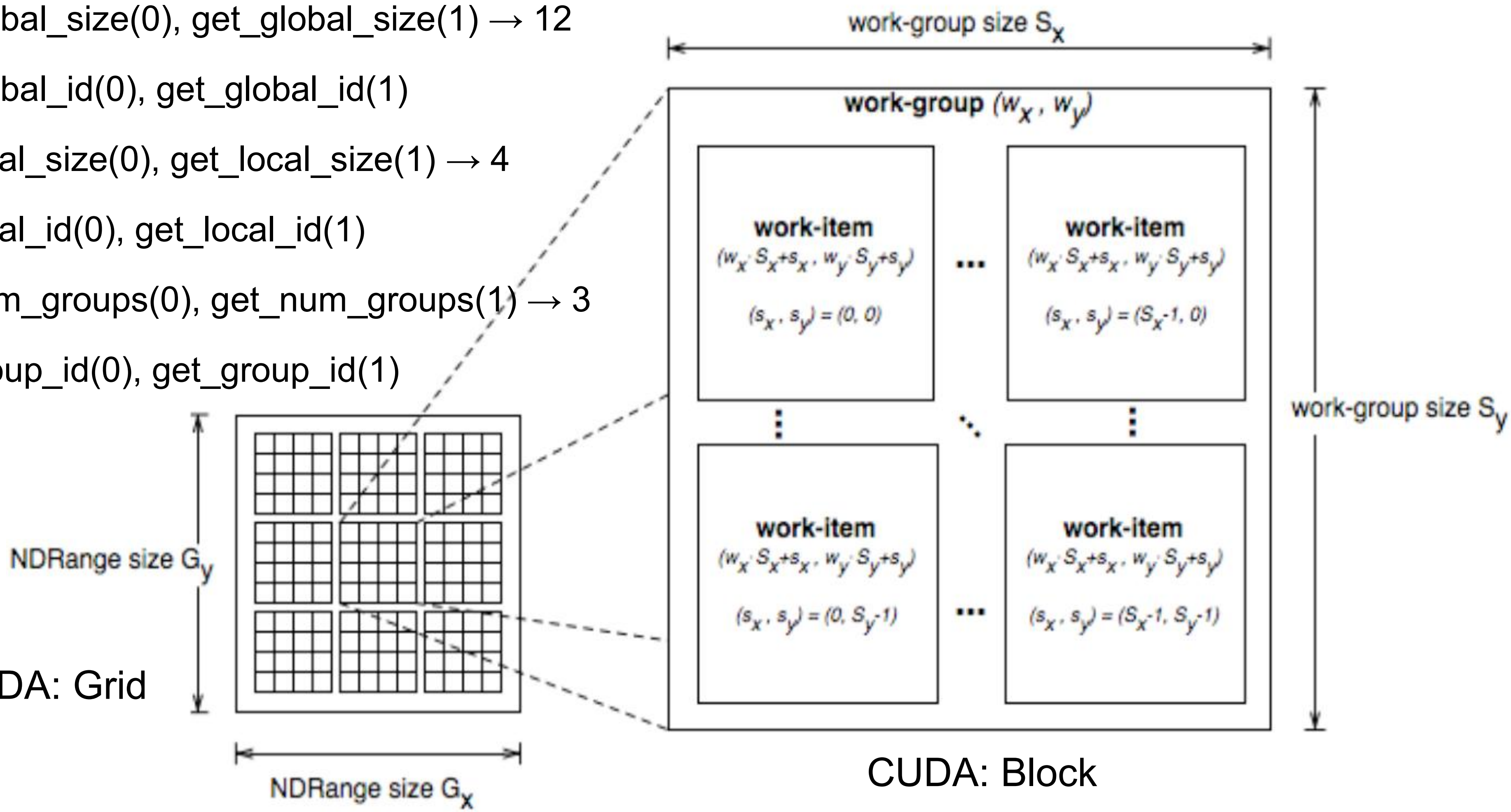
- **Work-groups**
  - identified by **work-group ID**
  - **Coarse grained decomposition of work**
- **Work-items**
  - identified by a **global ID**
  - unique **local ID** in work-group



OpenCL Architecture: Execution Model

Built-in functions: get_work_dim() → 2

- get_global_size(0), get_global_size(1) → 12
- get_global_id(0), get_global_id(1)
- get_local_size(0), get_local_size(1) → 4
- get_local_id(0), get_local_id(1)
- get_num_groups(0), get_num_groups(1) → 3
- get_group_id(0), get_group_id(1)



CUDA: Grid

CUDA: Block

Kernel: Vector Addition

```
__kernel void
vec_add (__global const float *a,
         __global const float *b,
         __global float *c)
{

    int gid = get_global_id(0);

    c[gid] = a[gid] + b[gid];

}
```

float *a=	0	1	2	3	4	5	6	7
float *b=	7	6	5	4	3	2	1	0

__kernel void vec_add(...)

float *c=	7	7	7	7	7	7	7	7
gid=	0	1	2	3	4	5	6	7

Global Work Size Gx=8

OpenCL Architecture: Execution Model

- Host defines context for execution of the kernels
- Context includes
  - Devices
    - Collection of OpenCL devices to be used by the host
  - Kernels
    - OpenCL functions that run on OpenCL devices
  - Program objects
    - Program source and executable that implement the kernels
  - Memory objects
    - Visible to host and devices
    - Values that can be operated on by instances of a kernel

OpenCL Architecture: Execution Model

- Command queues
  - Data structure to coordinate execution of kernels on the devices
  - Commands are placed in queue and scheduled onto the devices
    - Kernel execution command (execute a kernel)
    - Memory commands (transfer data)
    - Synchronisation commands
  - Scheduling properties in-order, out-of-order
  - Event objects control execution
    - Refer to kernel execution or memory commands
    - Coordinate execution between host and devices and between commands
  - Multiple queues with a single context
    - Run concurrently, no synchronisation

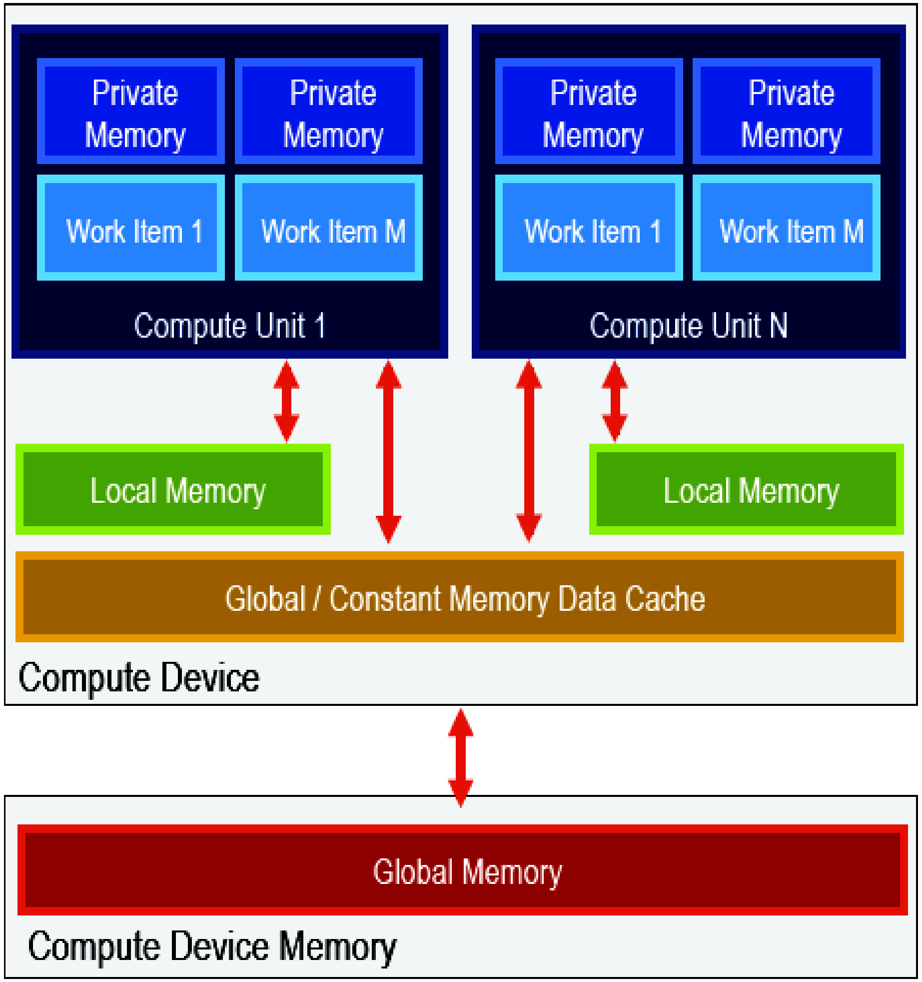
OpenCL Architecture: Memory Model

- Work-items executing a kernel have access to four memory regions
  - Global memory – read/write access to all work-items
  - Constant memory – remains constant during the execution of the kernel
  - Local memory – region local to a work-group (CUDA: shared)
  - Private memory – region private to a work-item (CUDA: local)

	Global	Constant	Local	Private
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access

OpenCL Architecture: Memory Model

- Multiple distinct address spaces
- Address qualifiers
  - __private
  - __local
  - __constant
  - __global
- Relaxed consistency
  - Local memory in a single work-group
  - Global memory at a work-group barrier
- Host creates memory objects in global memory
- Memory transfer blocking or non-blocking
- Host may map a region into its address space



OpenCL Architecture: Programming Model

- Data-parallel execution
  - Sequence of instructions applied to multiple elements of a memory object
  - Index space defines how the data maps onto work-items
  - No one-to-one mapping between work-item and element in memory object
  - Programmer defines total number of work-items to be executed in parallel
    - Explicit model – division into work-groups is specified by programmer
    - Implicit model – division into work-groups is managed by OpenCL
- Task-parallel execution
  - Compute devices such as CPUs can also execute task-parallel compute kernels
  - Executes as a single work-item per core
  - A compute kernel written in OpenCL
  - A native C / C++ function

OpenCL Platform Layer

- Implements platform-specific features allowing applications for
  - Querying OpenCL platforms
    - clGetPlatformIDs: provides list of available platform
    - clGetPlatformInfo: provides specific information about platform
  - Getting device information
    - clGetDeviceIDs: provides list of available devices
    - clGetDeviceInfo: provides specific information about device
  - Creating OpenCL contexts
    - clCreateContext: creates an OpenCL context with one or more devices (for managing command-queues, memory, kernels)
    - clCreateContextFromType: creates context from a device type
    - clGetContextInfo: provides information about a context

OpenCL Runtime, Buffer Objects, Program Objects

- API calls for executing a kernel, reading or writing a memory object
  - Command Queues
    - clCreateCommandQueue: creates queue on a specific device
  - Memory objects
    - clCreateBuffer: creates a buffer object
    - clEnqueueReadBuffer: enqueue read from a buffer to host memory
    - clEnqueueWriteBuffer: enqueue write to a buffer from host memory
  - Program objects
    - clCreateProgramWithSource: creates program object for context and loads the source code from text strings
    - clCreateProgramWithBinary: creates program object for context and loads binary into the program object

OpenCL Runtime, Program Objects, Kernel&Event Objects

- Program objects (cont'd)
  - clBuildProgram: builds (compiles & links) a program executable for all devices in context associated with program
  - clCreateKernel: creates a kernel object
  - clSetKernelArg: set the value for a specific argument of a kernel
- Executing kernels
  - clEnqueueNDRangeKernel: enqueues a command to execute a kernel using a given range of work-items
  - clEnqueueTask: enqueues a command to execute a kernel using a single work-item
- Flush and Finish
  - clFlush: issues all previously queued commands in queue
  - clFinish: blocks until all queued commands have completed

OpenCL Programming Language

- Derived from ISO C99
- A few restrictions
  - Recursion, function pointers, functions in C99 standard headers, ...
- Built-in Data Types
  - Scalar and vector data types, structs, pointers, data-type conversion functions, …
- Built-in Functions (see reference card)
  - work-item functions
  - math.h
  - read and write image
  - relational
  - geometric functions
  - synchronization functions

Example: Vector Addition

Compute c = a + b (a, b, and c are float vectors of length N)

```
void VectorAddHost(const float* a, const float* b, float* c, int N){
    int i;
    for (i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

OpenCL Host program:

- Query compute devices
- Create context
- Create memory objects
- Compile and create kernel
- Issue commands to command-queue
- Synchronization of commands
- Clean up resources

Platform, Device, Context

```
cl_platform_id cpPlatform;              // OpenCL platform
cl_device_id cdDevice;                  // OpenCL device
cl_context cxGPUContext;                // OpenCL context
cl_command_queue cqCommandQueue;        // OpenCL command queue
cl_int ciErr1;                          // Error code var

// Get an OpenCL platform
ciErr1 = clGetPlatformIDs(1, &cpPlatform, NULL);
// Get a GPU device
ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
// Create the context
cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);

// Create a command-queue
cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
```

Command-queue, buffer memory objects

```
int N = 11444777;                          // Length of float arrays (odd # for illustration)
void *srcA, *srcB, *dst;                   // Host buffers for OpenCL test
size_t szGlobalWorkSize;                   // 1D var for Total # of work items
size_t szLocalWorkSize;                    // 1D var for # of work items in the work group

// set Global and Local work size dimensions
szLocalWorkSize = 256;
// GlobalWorkSize rounded up to the nearest multiple of the LocalWorkSize
szGlobalWorkSize = ((N/(int)szLocalWorkSize)+1)*(int)szLocalWorkSize;

// Allocate host buffers
srcA = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
srcB = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
dst = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
```

Command-queue, buffer memory objects (cont'd)

```
cl_mem cmDevSrcA;          // OpenCL device source buffer A
cl_mem cmDevSrcB;          // OpenCL device source buffer B
cl_mem cmDevDst;           // OpenCL device destination buffer
```

```
//  Allocate buffer memory objects
cmDevSrcA = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr1);
cmDevSrcB = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
cmDevDst = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
                sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
```

Build program and create kernel

```
const char* cSourceFile = "VectorAdd.cl";          // kernel source file
cl_program cpProgram;                              // OpenCL program

// Read the OpenCL kernel in from source file
char** program_src;
std::vector<std::string> prog_lines;
readCLFile (prog_lines, cSourceFile);
program_src = (char**) malloc (sizeof(char*) * prog_lines.size());
for (unsigned int i = 0; i < prog_lines.size(); i++)
    program_src[i] = (char*) prog_lines[i].c_str();

// Create the program
cpProgram = clCreateProgramWithSource(cxGPUContext, prog_lines.size(),
                                     (const char **)program_src, NULL, &ciErr1);
// Build the program (compile & link)
ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
```

Build program and create kernel

```
cl_kernel ckKernel;                                   // OpenCL kernel
// Create the kernel
ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);

// Set the Argument values
ciErr1  = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&cmDevSrcA);
ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&cmDevSrcB);
ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&cmDevDst);
ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&N);

// Asynchronous write of data to GPU device
ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcA, CL_FALSE, 0,
                 sizeof(cl_float) * szGlobalWorkSize, srcA, 0, NULL, NULL);
ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcB, CL_FALSE, 0,
                 sizeof(cl_float) * szGlobalWorkSize, srcB, 0, NULL, NULL);
```

Build program and create kernel

```
//  Launch kernel (1-dimensional range)
 ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
                    &szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);

// Synchronous/blocking read of results, and check accumulated errors
 ciErr1 = clEnqueueReadBuffer(cqCommandQueue, cmDevDst, CL_TRUE, 0,
                    sizeof(cl_float) * szGlobalWorkSize, dst, 0, NULL, NULL);

// Cleanup allocated objects
if(ckKernel)clReleaseKernel(ckKernel);
if(cpProgram)clReleaseProgram(cpProgram);
if(cqCommandQueue)clReleaseCommandQueue(cqCommandQueue);
if(cxGPUContext)clReleaseContext(cxGPUContext);
if(cmDevSrcA)clReleaseMemObject(cmDevSrcA);
if(cmDevSrcB)clReleaseMemObject(cmDevSrcB);
if(cmDevDst)clReleaseMemObject(cmDevDst);
```

Kernel: Handle extra indices

```
__kernel void
vec_add (__global const float *a,
         __global const float *b,
         __global float *c,
         int N))
{
    // get index into global data array
    int gid = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for standard/serial C code)
    if (gid >= N)
    {
        return;
    }
    // add the vector elements
    c[gid] = a[gid] + b[gid];
}
```

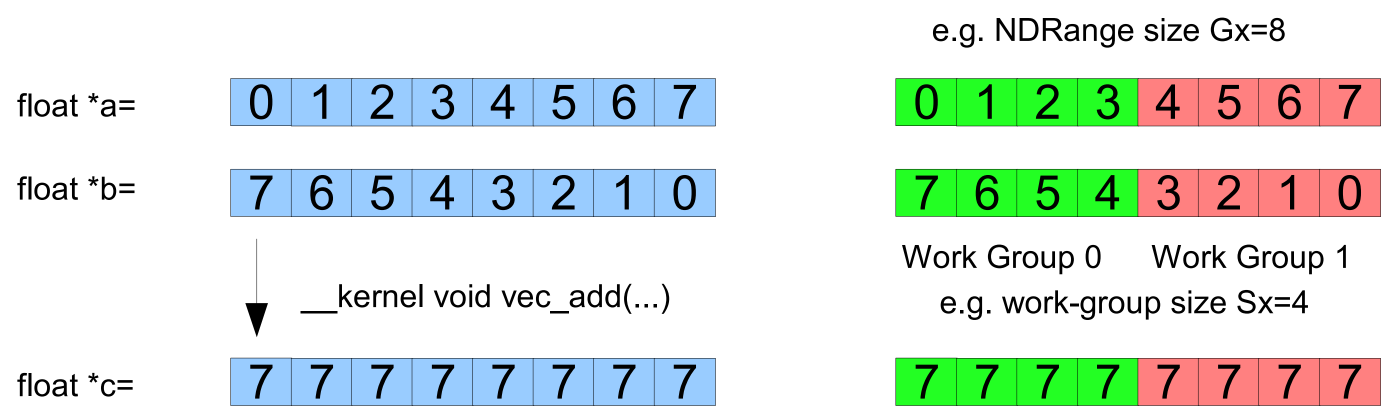
Array length **N** = 11444777
Global Work Size = 11444992
Local Work Size = 256
\# **of Work Groups** = 44707 (**last** work-group deals only with **215** elements)

Kernel: Use Work-Item Built-in Functions

```
__kernel void
vec_add (__global const float *a,
         __global const float *b,
         __global float *c,
         int N))
{
    // get index into global data array
//  int gid = get_global_id(0);
    int gid = get_local_size(0)*get_group_id(0)
            + get_local_id(0);

    // add the vector elements
    c[gid] = a[gid] + b[gid];
}
```

Array length **N**	= 8
Global Work Size	= 8
Local Work Size	= 4
# of Work Groups	= 2



JUGIPSY: Getting started

- See login message
  - Functionality tests:
    - /usr/bin/nvidia-smi
    - /srv/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/deviceQuery
    - /srv/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/bandwidthTest
  - For a start, copy the tutorials to your own home directory …
    - Develop own projects from SDK tutorials
  - NVIDIA:
    - 30 OpenCL tutorials, 71 CUDA tutorials
  - ATI:
    - 37 OpenCL tutorials, 14 CAL tutorials
  - Mind environment, e.g.LD_LIBRARY_PATH sequence:
    - **module load cuda - module switch cuda ati-stream-sdk**

JUDGE: Getting started

Functionality tests:

- cd /localhome/partec/NVIDIA_GPU_Computing_SDK_3.2.16
  - C/bin/linux/release/deviceQuery
  - C/bin/linux/release/bandwidthTest
- For a start, copy the tutorials to your own home directory …
  - Develop own projects from NVIDIA SDK tutorials (28 OpenCL, 79 CUDA)
- Account application:
  - https://dispatch.fz-juelich.de:8812/account_ident/back=/RESSOURCEN
- Environment: source NV_env.sh:

```
export CUDA_HOME=/opt/cuda
export CUDA_INSTALL_PATH=/opt/cuda
export
CUDA_SDK_HOME=/localhome/partec/NVIDIA_GPU_Computing_SDK_3.2.16
export PATH=${CUDA_HOME}/bin:${CUDA_HOME}/computeprof/bin:$PATH
export LD_LIBRARY_PATH=${CUDA_HOME}/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=${CUDA_SDK_HOME}/lib:$LD_LIBRARY_PATH
export CUDA_VERSION=3.2.16
```

Matrix Multiplication

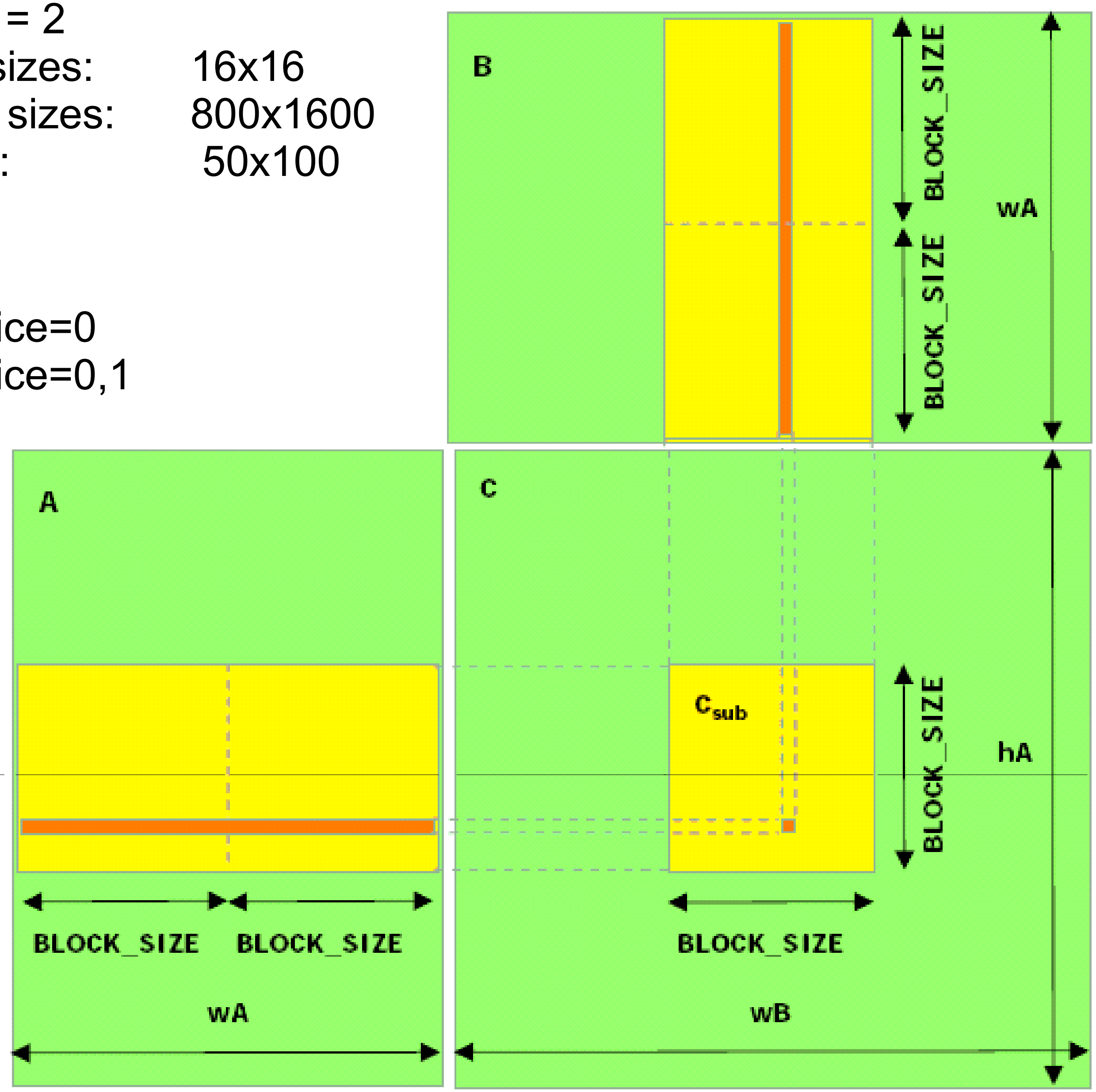
Matrix sizes:
wA = 800
hA = 1600
wB = 800

Dimensions = 2	
Local work sizes:	16x16
Global work sizes:	800x1600
Group sizes:	50x100

```
./oclMatrixMul -sizemult=10 -device=0
./oclMatrixMul -sizemult=10 -device=0,1
```

Device 0

Device 1



Results

./oclMatrixMul -sizemult=10 -device=0
./oclMatrixMul Starting... Device 0: Device Tesla T10 Processor
Using Matrix Sizes: A(800 x 1600), B(800 x 800), C(800 x 1600)

Running Computations on 1 - 1 GPU's...

MM, Throughput = **175.4612 GFlops/s**, Time = **0.01167 s**, Size = 2048000000, NumDevsUsed = 1, Workgroup = 256
  Kernel execution time on GPU 0   : 0.01164 s

Comparing results with CPU computation... PASSED

./oclMatrixMul -sizemult=10 -device=0,1
./oclMatrixMul Starting...Device 0: Device Tesla T10 Processor   Device 1: Device Tesla T10 Processor
Using Matrix Sizes: A(800 x 1600), B(800 x 800), C(800 x 1600)

Running Computations on 1 - 2 GPU's...

MM, Throughput = **175.4431 GFlops/s, Time = 0.01167 s**, Size = 2048000000, NumDevsUsed = 1, Workgroup = 256
  Kernel execution time on GPU 0   : 0.01166 s

MM, Throughput = **347.5211 GFlops/s**, **Time = 0.00589 s**, Size = 2048000000, NumDevsUsed = 2, Workgroup = 256
  Kernel execution time on GPU 0   : 0.00587 s
  Kernel execution time on GPU 1   : 0.00586 s

Comparing results with CPU computation... PASSED

Mitglied der Helmholtz-Gemeinschaft

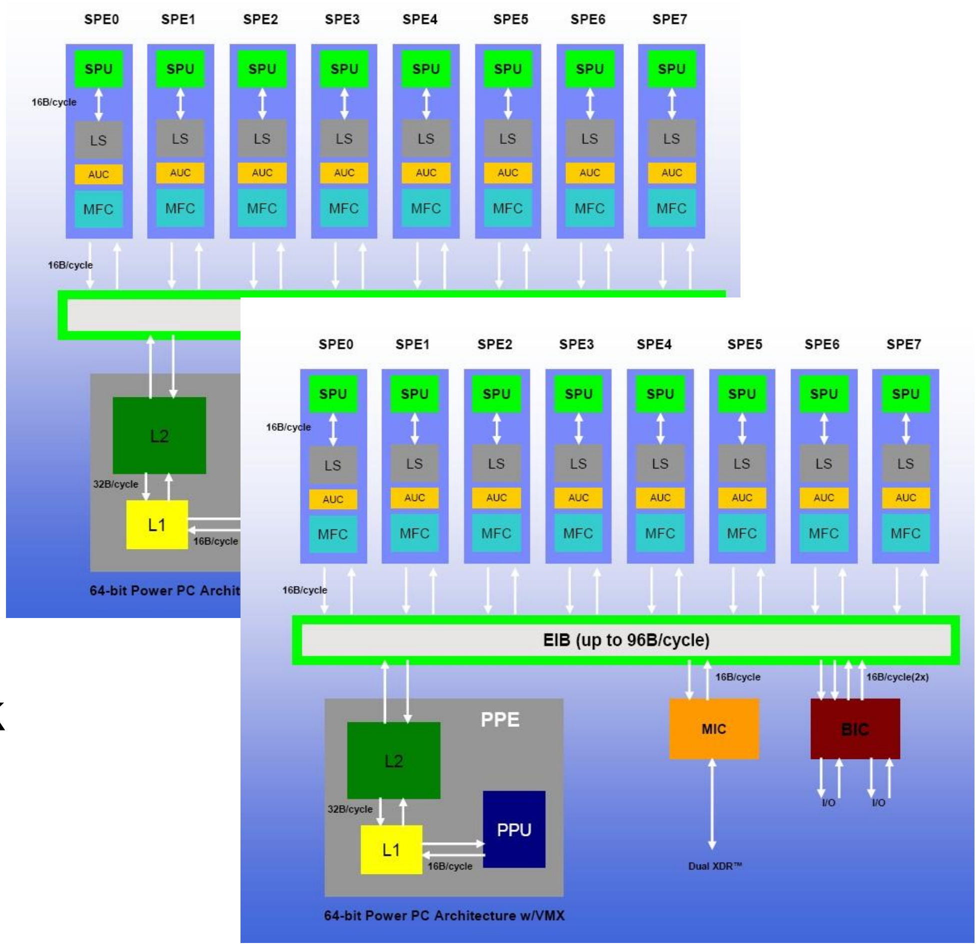
JÜLICH FORSCHUNGSZENTRUM

OpenCL Basics
Extra slides

23. März 2011 | Willi Homberg

Cell Cluster JUICEnext

- 35 QS22 blades @ 3.2 Ghz
  - 2 Cell processors (CBE)
  - 8 GB Memory
- 4x InfiniBand adapter (SDR)
- Frontend: x86 compatible
- Interconnect switch: IB (Cisco)
- Peak performance:
  - 1 blade: 217,6 GFLOPS peak (64 bit float)
  - 35 blades: 7 TFLOPS peak (64 bit float)



JUICEnext: Getting started

PLATFORM NAME : IBM
PLATFORM VERSION : OpenCL 1.0 200912040043
PLATFORM PROFILE : FULL_PROFILE
PLATFORM VENDOR : IBM
PLATFORM EXTENSIONS :
2 devices found supporting OpenCL

Device name :ACCELERATOR PowerXCell8i processor
Device version :OpenCL 1.0 200912040043
Device profile :EMBEDDED_PROFILE
Device vendor :IBM
Device extensions :cl_khr_byte_addressable_store
Max Compute Units : 16
Amount of Global Memory : 1939271680 bytes
Amount of Local Memory : 248832 bytes
Max Work Group Size : 256
Max Work Item Dimensions : 3
Max Work Items on dimemsion 0 : 256
Max Work Items on dimemsion 1 : 256
Max Work Items on dimemsion 2 : 256

Device name :ACCELERATOR PowerXCell8i processor
Device version :OpenCL 1.0 200912040043
Device profile :EMBEDDED_PROFILE
...

JUICEnext: Getting started

OpenCL environment:

access IBM SDK: /srv/Software/OpenCL_Cell
samples: /srv/Software/OpenCL_Cell/samples

e.g. copy sample to your $HOME directory:

```
cd /srv/Software/OpenCL_Cell/samples
cp -a blackscholes $HOME
```

then:

```
qsub -I
cd $HOME/blackscholes/ppc
make
./bsop --help
```