CUDA and OpenCL API comparison

Presentation for T-106.5800 Seminar on GPGPU Programming, spring 2010

Sami Rosendahl
sami.rosendahl@digia.com

Contents

- CUDA and OpenCL basics compared
- Development models and toolchains compared
- APIs compared
  - Kernel programming languages and APIs
  - Host APIs
  - API abstraction levels
- Concepts compared (optional)
  - Device models
  - Execution models
  - Memory models
- Summary

CUDA basics

- Proprietary technology for GPGPU programming from Nvidia
- Not just API and tools, but name for the whole architecture
- Targets Nvidia hardware and GPUs only
- First SDK released Feb 2007
- SDK and tools available to 32- and 64-bit Windows, Linux and Mac OS
- Tools and SDK are available for free from Nvidia.

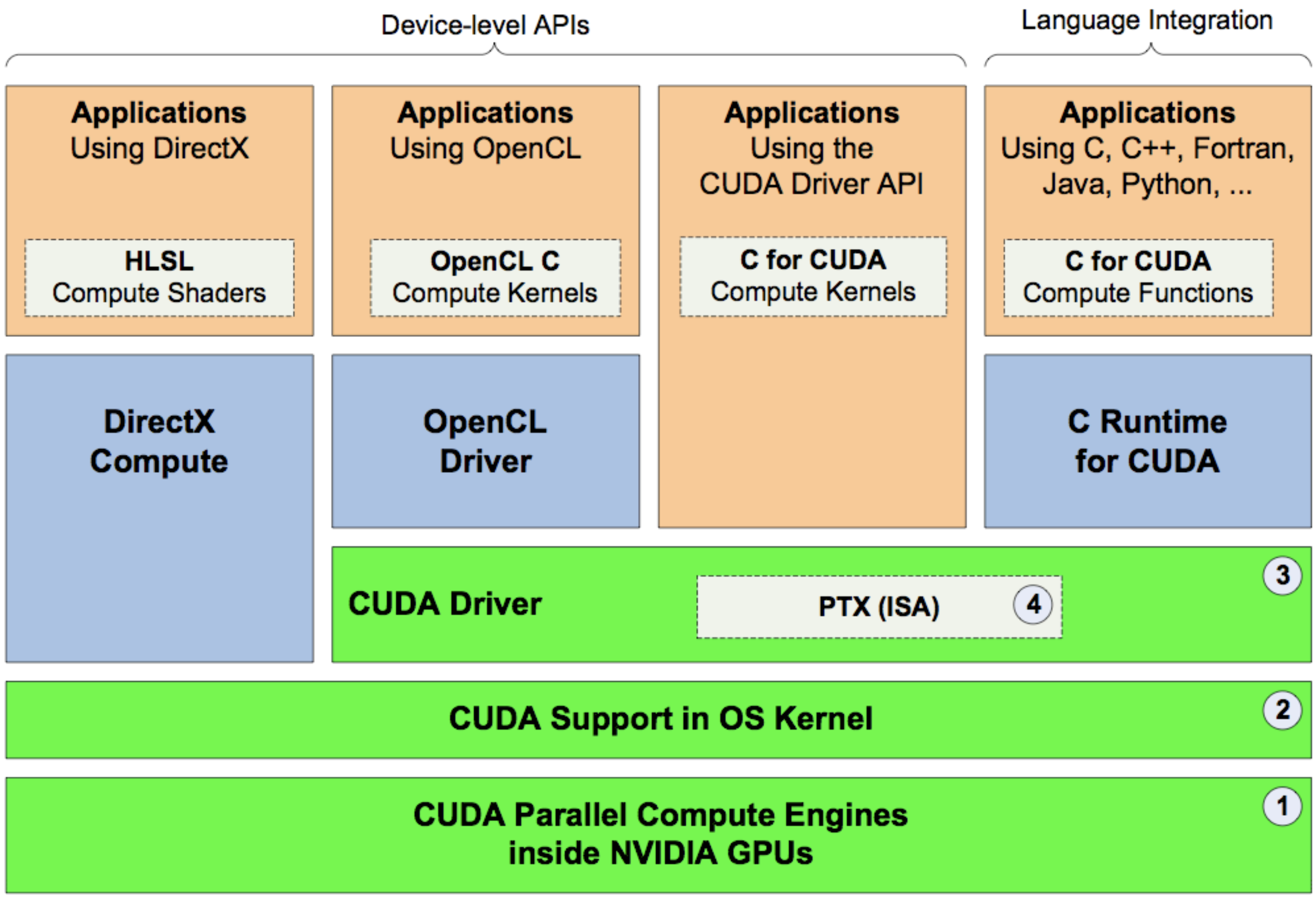
OpenCL basics

- Open, royalty-free standard for parallel, compute intensive application development
- Initiated by Apple, specification maintained by the Khronos group
- Supports multiple device classes, CPUs, GPUs, DSPs, Cell, etc.
- Embedded profile in the specification
- Specification currently at version 1.0, released Dec 2008
- SDKs and tools are provided by compliant device vendors.

Basics compared

	CUDA	OpenCL
What it is	HW architecture, ISA, programming language, API, SDK and tools	Open API and language specification
Proprietary or open technology	Proprietary	Open and royalty-free
When introduced	Q4 2006	Q4 2008
SDK vendor	Nvidia	Implementation vendors
Free SDK	Yes	Depends on vendor
Multiple implementation vendors	No, just Nvidia	Yes: Apple, Nvidia, AMD, IBM
Multiple OS support	Yes: Windows, Linux, Mac OS X; 32 and 64-bit	Depends on vendor
Heterogeneous device support	No, just Nvidia GPUs	Yes
Embedded profile available	No	Yes

CUDA System Architecture [3]



CUDA development model

- A CUDA application consists of *host* program and CUDA *device* program
- The host program activates computation *kernel*s in the device program
- A computation kernel is a data-parallel routine
- Kernels are executed on the device for multiple data items in parallel by device *thread*s
- Computation kernels are written in C for CUDA or PTX
  - C for CUDA adds language extensions and built-in functions for device programming
  - Support for other kernel programming languages is also planned
- Host program accesses the device with either *C runtime for CUDA* or *CUDA Driver API*
  - C runtime interface is higher-level and less verbose to use than the Driver API
  - With C runtime computation kernels can be invoked from the host program with convenient CUDA-specific invocation syntax
  - The Driver API provides more finer grained control
  - Bindings to other programming languages can be built on top of either API
- Device and host code can be mixed or written to separate source files
- Graphics interoperability is provided with OpenGL and Direct3D
- Nivida provides also OpenCL interface for CUDA

CUDA toolchain

- The device program is compiled by the CUDA SDK-provided `nvcc` compiler
- For device code `nvcc` emits CUDA PTX assembly or device-specific binary code
- PTX is intermediate code specified in CUDA that is further compiled and translated by the device driver to actual device machine code
- Device program files can be compiled separately or mixed with host code if CUDA SDK-provided `nvcc` compiler is used
- CUDA custom kernel invocation syntax requires using the nvcc compiler
- Separate compilation can output C host code for integrating with the host toolchain

OpenCL development model

- An OpenCL application consists of *host program* and *OpenCL program* to be executed on the computation *device*
- The host program activates computation *kernel*s in the device program
- A computation kernel is a data-parallel routine
- Kernels are executed on the device for multiple data items in parallel by device *processing elements*
  - Also task-parallel and hybrid models are supported
- OpenCL kernels are written with the OpenCL C programming language
  - OpenCL C is based on C99 with extensions and limitations
  - In addition to the language, OpenCL specifies library of built-in functions
  - Implementations can also provide other means to write kernels
- The host program controls the device by using the OpenCL C API
  - Bindings to other host programming languages can be built on top of the C API
- Graphics interoperability is provided with OpenGL

OpenCL toolchain

- An OpenCL implementation must provide a compiler from OpenCL C to supported device executable code
- The compiler must support standard set of OpenCL defined options
- The kernels can be compiled either *online* (run-time) or *offline* (build-time)
- For online compilation OpenCL C source text is provided by the host program to OpenCL API
- Run time compilation is more flexible for the final application, but may be problematic in some cases
  - Compilation errors need to be extracted through the OpenCL API at development time
  - The kernel source code is included in the application binaries
- The host program is compiled with the default host toolchain and OpenCL is used through its C API

Development models compared

	CUDA	OpenCL
Explicit host and device code separation	Yes *)	Yes
Custom kernel programming language	Yes	Yes
Multiple computation kernel programming languages	Yes	Only OpenCL C or vendor-specific language(s)
Data parallel kernels support	Yes, the default model	Yes
Task parallel kernels support	No, at least not efficiently	Yes
Device program intermediate language specified	Yes, PTX	Implementation specific or no intermediate language used
Multiple programming interfaces	Yes, *including* OpenCL	Only the specified C API with possible vendor extensions
Deep host and device program integration support	Yes, with very efficient syntax	No, only separate compilation and kernel invocation with API calls
Graphics interoperability support	Yes, with OpenGL and Direct3D	Yes, with OpenGL

*) See [6] and [7] as examples of seamless Host/GPGPU programming tools

Toolchains compared

	CUDA	OpenCL
Custom toolchain needed for host program	Yes, if mixed device/host code or custom kernel invocation syntax is used	No
Support for using platform default toolchain for the host program	Yes	Yes, the only option
Run-time device program compilation support	Yes, from PTX (only with the Driver API)	Yes, from OpenCL C source text

C for CUDA kernel programming

- Based on C programming language with extensions and restrictions
  - Curiously the C language standard version used as base is not defined
- Extensions
  - Built-in vector data types, but no built-in operators or math functions*) for them
  - Function and variable type qualifiers
  - Built-in variables for accessing thread indices
  - Intrinsic floating-point, integer and fast math functions
  - Texture functions
  - Memory fence and synchronization functions
  - Voting functions (from CC 1.2)
  - Atomic functions (from CC 1.1)
  - Limited C++ language features support: function and operator overloading, default parameters, namespaces, function templates
- Restrictions
  - No recursion support, static variables, variable number of arguments or taking pointer of device functions
  - No dynamic memory allocation
  - No double precision floating point type and operations (expect from CC 1.3)
  - Access to full set of standard C library (e.g. `stdio`) only in emulation mode
- Numerical accuracy
  - Accuracy and deviations from IEEE-754 are specified
  - For deviating operations compliant, but slower software versions are provided

*) see `C/common/inc/cutil_math.h` in CUDA SDK for SW vector operations

OpenCL C kernel programming

- Based on the C programming language with extensions and restrictions
  - Based on the C99 version of the language standard
- Extensions
  - Built-in first-class vector data types with literal syntax, operators and functions
  - Explicit data conversions
  - Address space, function and attribute qualifiers
  - OpenCL-specific `#pragma` directives
  - Built-in functions for accessing work item indices
  - Built-in math, integer, relational and vector functions
  - Image read and write functions
  - Memory fence and synchronization functions
  - Asynchronous memory copying and prefetch functions
  - Optional extensions: e.g. atomic operations, etc.
- Restrictions
  - No recursion, pointer to pointer arguments to kernels, variable number of arguments or pointers to functions
  - No dynamic memory allocation
  - No double-precision floating point support by default
  - Most C99 standard headers and libraries cannot be used
  - `extern`, `static`, `auto` and `register` storage-class specifiers are not supported
  - C99 variable length arrays are not supported
  - Writing to arrays or `struct` members with element size less than 32 bits is not supported by default
  - Many restrictions can be addressed by extensions, e.g. double-precision support, byte addressing etc.
- Numerical accuracy
  - Accuracy and deviations from IEEE-754 are specified
  - Some additional requirements specified beyond C99 TC2

Kernel programming differences

	CUDA	OpenCL
Base language version defined	No, just "based on C" and some C++ features are supported	Yes, C99
Access to work-item indices	Through built-in variables	Through built-in functions
Address space qualification needed for kernel pointer arguments	No, defaults to global memory	Yes
First-class built-in vector types	Just vector types defined, no operators or functions	Yes: vector types, literals, built-in operators and functions
Voting functions	Yes (CC 1.2 or greater)	No
Atomic functions	Yes (CC 1.1 or greater)	Only as extension
Asynchronous memory copying and prefetch functions	No	Yes
Support for C++ language features	Yes: limited, but useful set of features supported	No

Kernel code example

- Matrix multiplication kernel in C for CUDA and OpenCL C
- See the handout

Host API usage compared

C Runtime for CUDA	CUDA Driver API	OpenCL API
Setup		
	Initialize driver Get device(s) (Choose device) Create context	Initialize platform Get devices Choose device Create context Create command queue
Device and host memory buffer setup		
Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result	Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result	Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result
Initialize kernel		
	Load kernel module (Build program) Get module function	Load kernel source Create program object Build program Create kernel object bound to kernel function
Execute the kernel		
	Setup kernel arguments	Setup kernel arguments
Setup execution configuration Invoke the kernel (directly with its parameters)	Setup execution configuration Invoke the kernel	Setup execution configuration Invoke the kernel
Copy results to host		
Copy results from device memory	Copy results from device memory	Copy results from device memory
Cleanup		
Cleanup all set up above	Cleanup all set up above	Cleanup all set up above

Host APIs code example – launching a matrix multiplication kernel

C runtime for CUDA

```
dim3 threads( BLOCK_SIZE, BLOCK_SIZE );
dim3 grid( WC / threads.x, HC / threads.y );
matrixMul<<< grid, threads >>>( d_C, d_A, d_B, WA, WB );
```

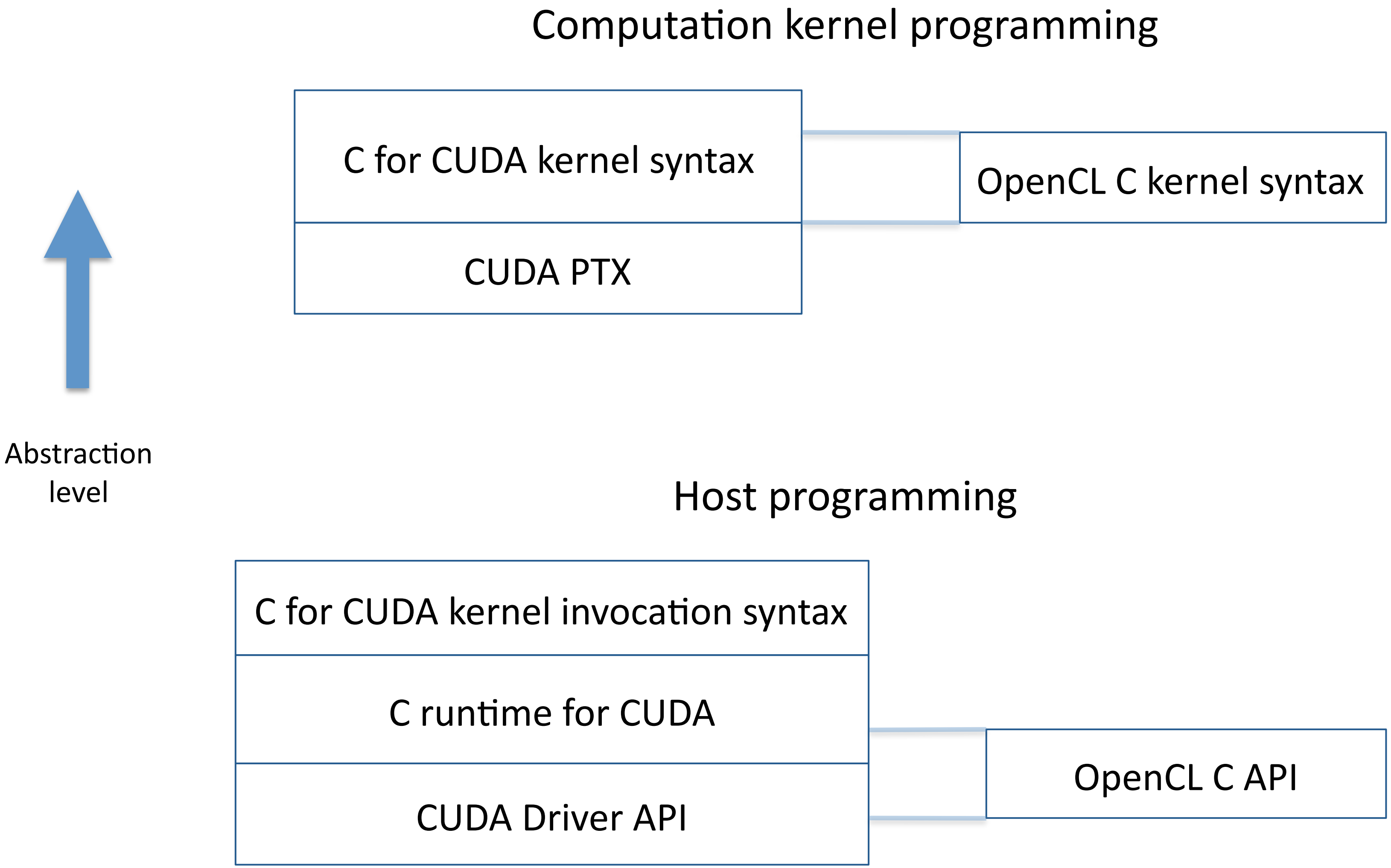
CUDA Driver API

```
cuFuncSetBlockShape( matrixMul, BLOCK_SIZE, BLOCK_SIZE, 1 );
cuFuncSetSharedSize( matrixMul, 2*BLOCK_SIZE*BLOCK_SIZE*sizeof(float) );
cuParamSeti( matrixMul, 0,  d_C );
cuParamSeti( matrixMul, 4,  d_A );
cuParamSeti( matrixMul, 8,  d_B );
cuParamSeti( matrixMul, 12, WA );
cuParamSeti( matrixMul, 16, WB );
cuParamSetSize( matrixMul, 20 );
cuLaunchGrid( matrixMul, WC / BLOCK_SIZE, HC / BLOCK_SIZE );
```

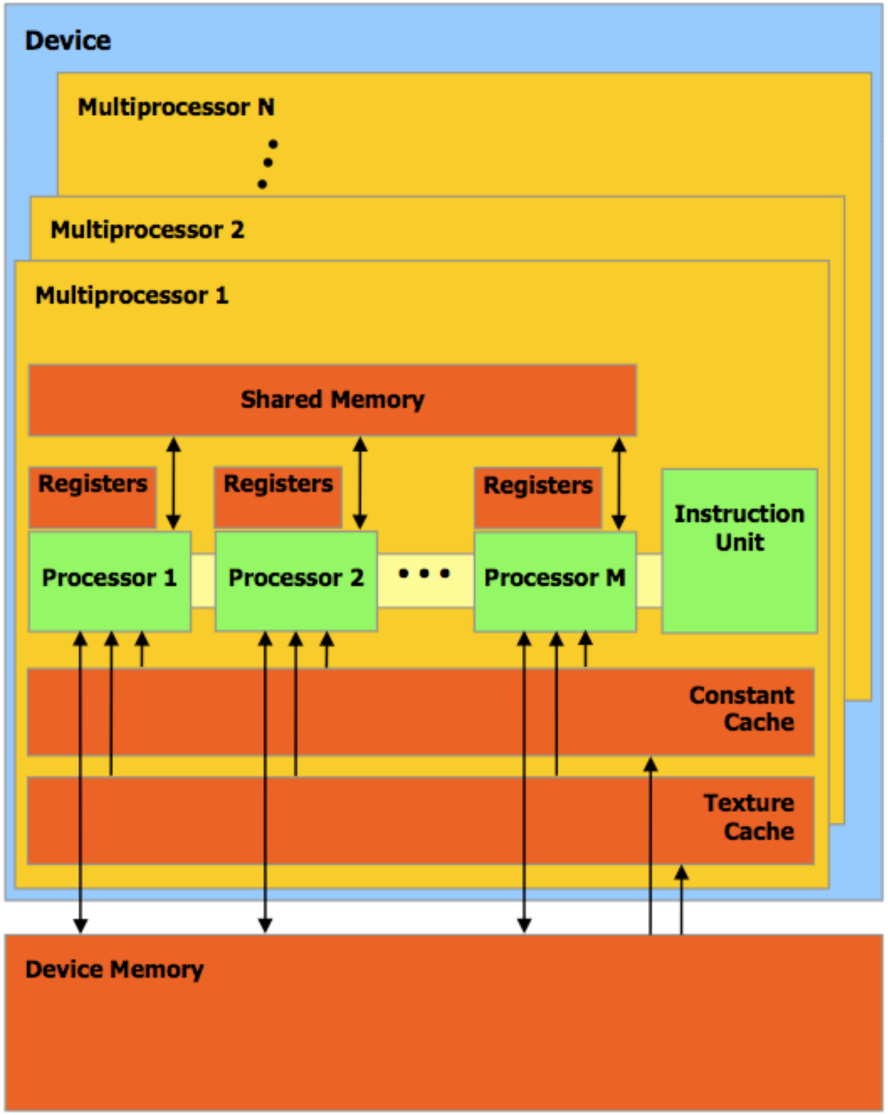
OpenCL API

```
clSetKernelArg( matrixMul, 0, sizeof(cl_mem), (void *) &d_C );
clSetKernelArg( matrixMul, 1, sizeof(cl_mem), (void *) &d_A );
clSetKernelArg( matrixMul, 2, sizeof(cl_mem), (void *) &d_B );
clSetKernelArg( matrixMul, 3, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
clSetKernelArg( matrixMul, 4, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
size_t localWorkSize[]  = { BLOCK_SIZE, BLOCK_SIZE };
size_t globalWorkSize[] = { shrRoundUp(BLOCK_SIZE, WC), shrRoundUp(BLOCK_SIZE, workSize) };
clEnqueueNDRangeKernel( commandQueue, matrixMul, 2, 0, globalWorkSize, localWorkSize,
                                          0, NULL, &GPUExecution );
clFinish( commandQueue );
```

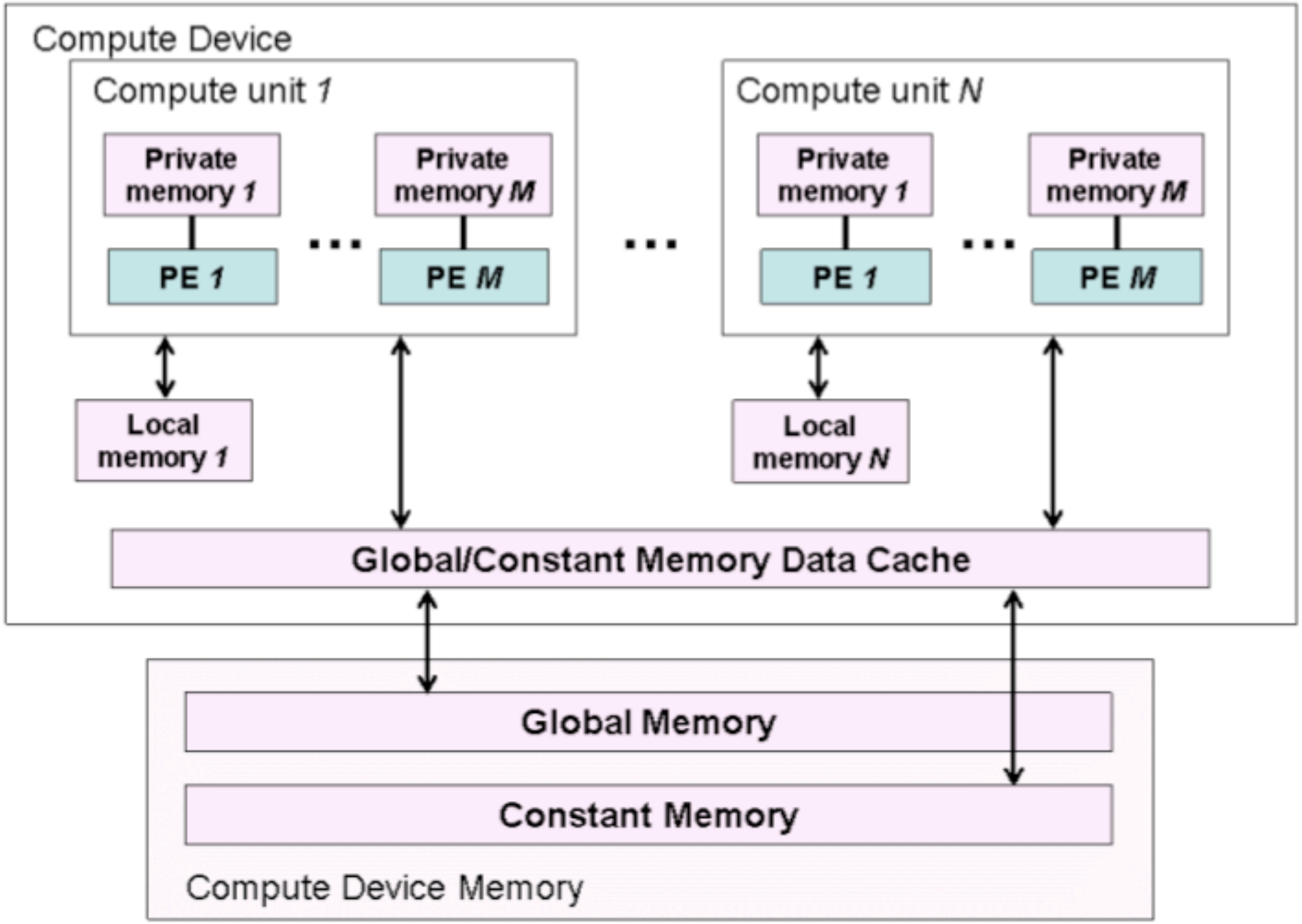
API abstraction levels compared



CUDA device model [1]



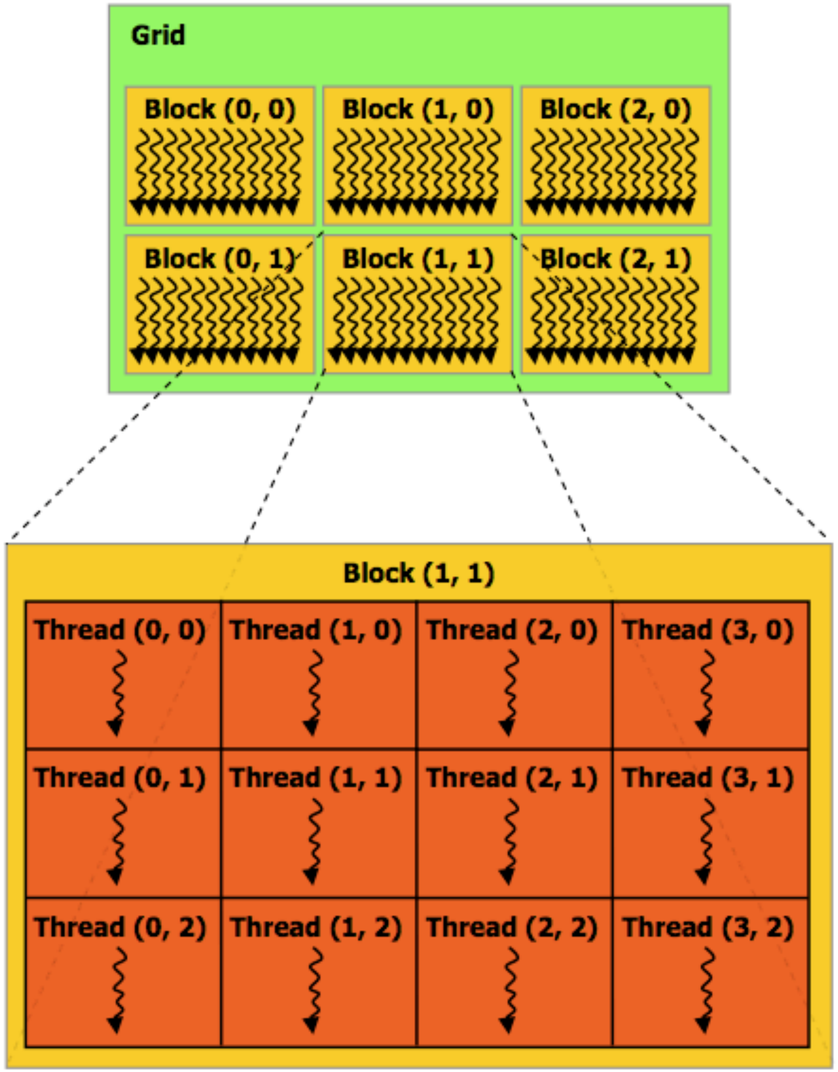
OpenCL device model [4]



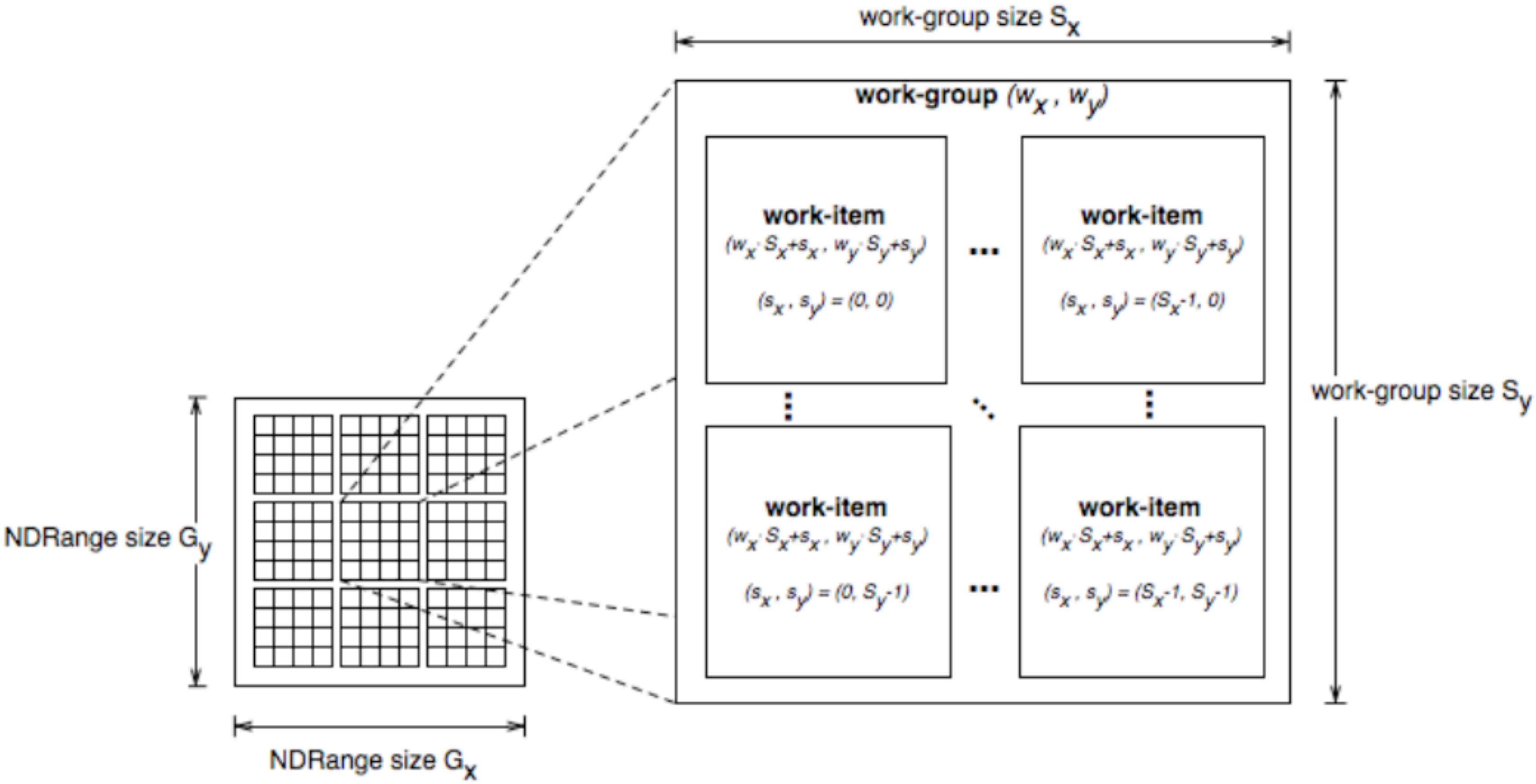
Device models compared

- The models are very similar
- Both are very hierarchic and scalable
- OpenCL model is more generic and uses more generic terminology
  - *Processing Element* instead of *Processor* etc.
- CUDA model is Nvidia-architecture specific
  - Makes Nvidia's *SIMT* execution model explicit

CUDA Execution model



OpenCL execution model [4]



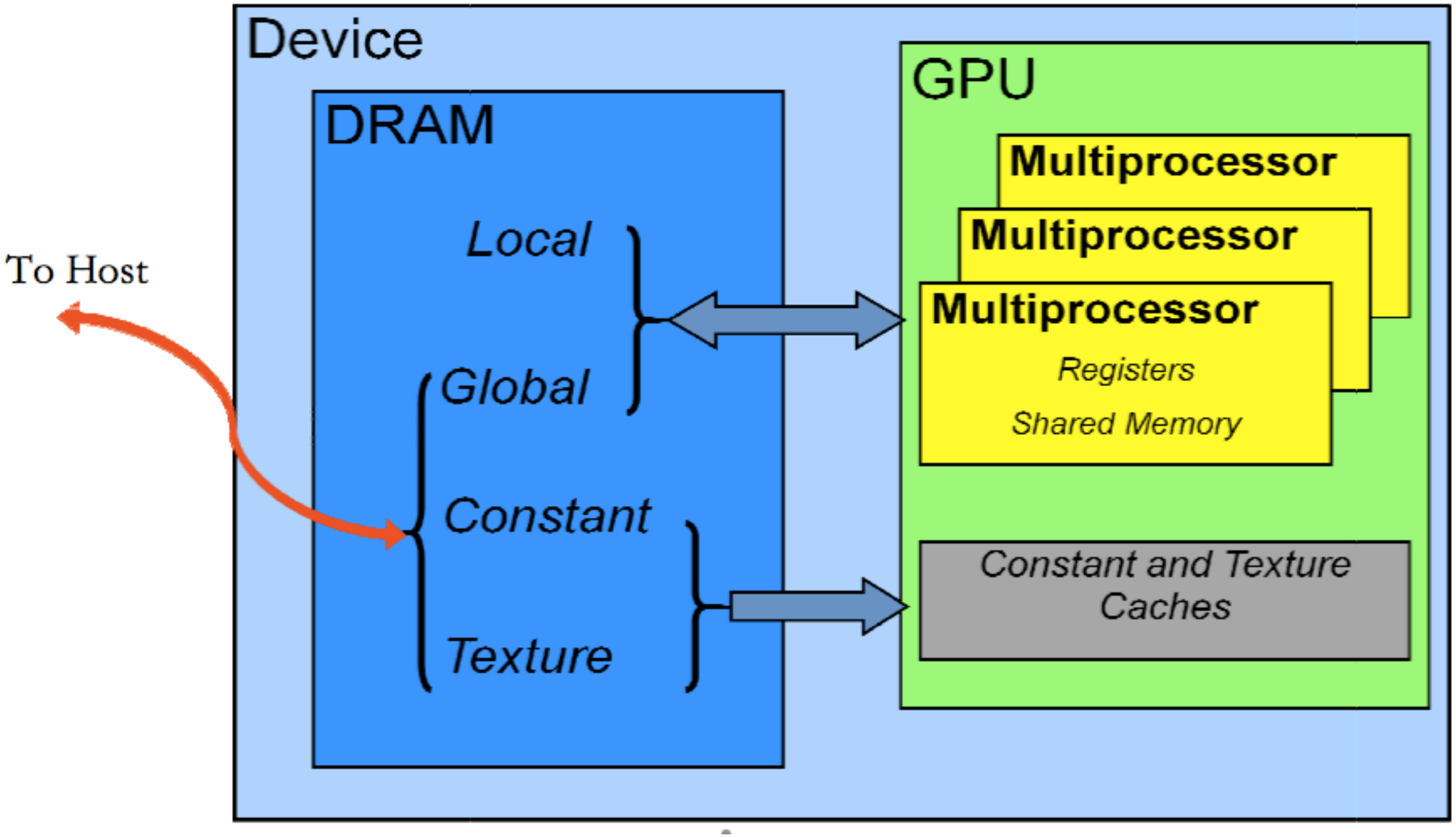
Execution models compared

- Both models provide similar hierarchical decomposition of the computation index space
- Index space is supplied to the device when a kernel is invoked
- In Nvidia's model individual work items are explicitly HW threads
- Both execution model are linked with the device and memory models
- Synchronization is available on Thread block/ work-group level only

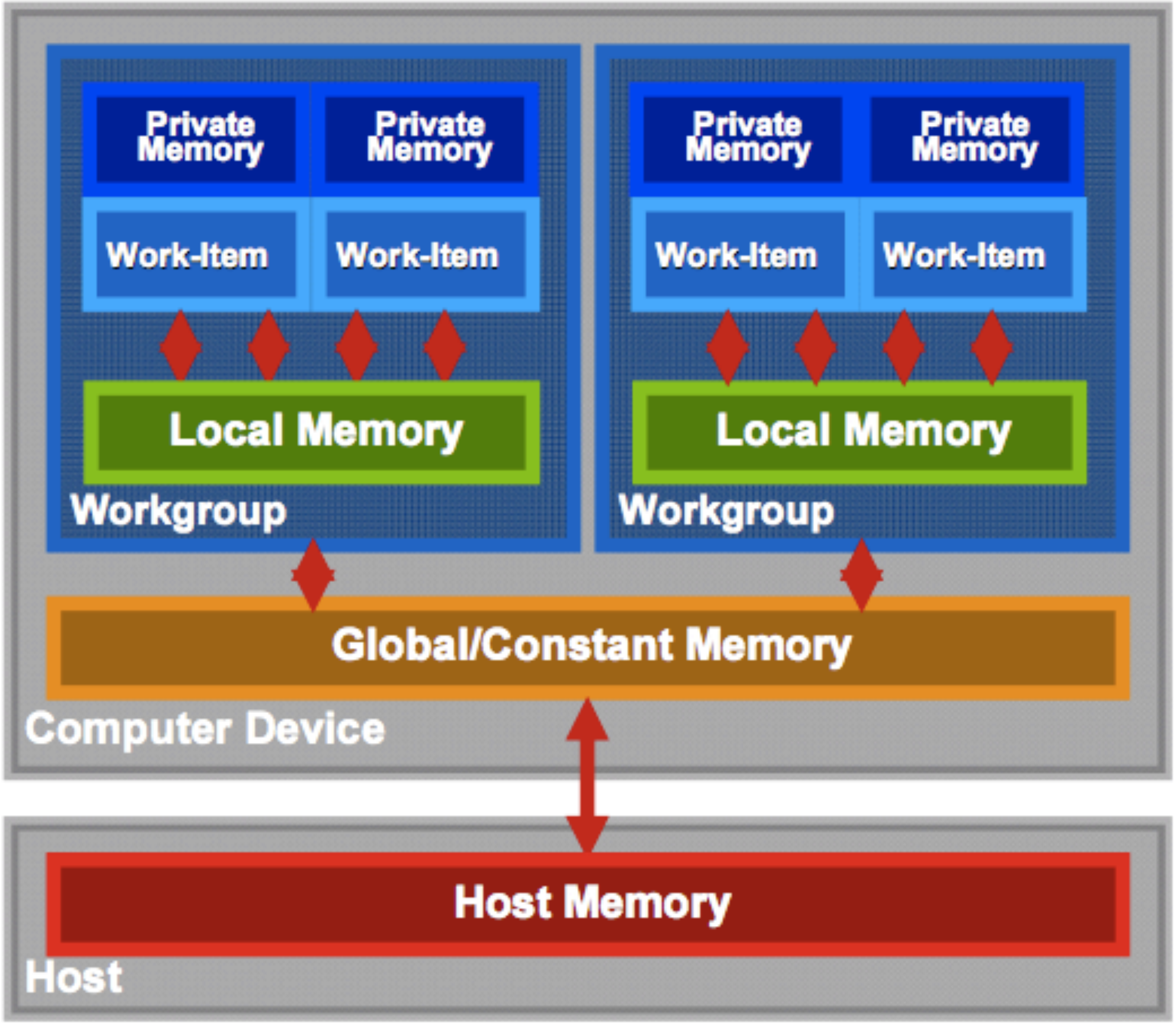
Execution model terminology mapping

CUDA	OpenCL
Grid	NDRange
Thread Block	Work group
Thread	Work item
Thread ID	Global ID
Block index	Block ID
Thread index	Local ID

CUDA Memory model [2]



OpenCL Memory model [5]



Memory models compared

- In both models, host and device memories are separate
- Both models are very hierarchical and need to be explicitly controlled by the programmer
- OpenCL's model is more abstract and provides more leeway for implementation differences
- There's no direct correspondence for CUDA's Local memory in OpenCL
- CUDA defines explicitly what memories are cached and what are not, in OpenCL such details are device-dependent
- In both APIs actual device capabilities can be queried using the API

Memory model terminology mapping

CUDA	OpenCL
Host memory	Host memory
Global or Device memory	Global memory
Local memory	Global memory
Constant memory	Constant memory
Texture memory	Global memory
Shared memory	*Local* memory
Registers	Private memory

Summary

- CUDA and OpenCL are similar in many respects
  - Focused on data-parallel computation model
  - Separate device/host programs and memories
  - Custom, C-based languages for device programming
  - Device, execution and memory models are very similar
  - OpenCL has been implemented on top of CUDA!
- Most differences stem from differences in origin
  - CUDA is Nvidia's proprietary technology that targets Nvidia devices only
  - OpenCL is an open specification aiming to target different device classes from competing manufacturers
  - CUDA is more than just API and programming model specification
  - CUDA has been on the market longer and thus has more support, applications and related research and products available
  - CUDA has more documentation, but it is also more vague.

References

[1] *NVIDIA CUDA™ Programming Guide*, Version 2.3.1 8/26/2009.

[2] *NVIDIA CUDA C Programming Best Practices Guide*, version 2.3, July 2009.

[3] *NVIDIA® CUDA™ Architecture Introduction & Overview*, Version 1.1, April 2009.

[4] *The OpenCL Specification*, Version 1.0 Document Revision 48, Khronos OpenCL Working Group

[5] *OpenCL Overview presentation*, opencl_overview.pdf, Khronos Group, 11.9.2009.

[6] *OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization*, Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann; PPoPP'09
http://www.multicoreinfo.com/research/papers/2009/ppopp09-9-omp2gpu-lee.pdf

[7] *PGI Accelerator Compilers*, The Portland Group
http://www.pgroup.com/resources/accel.htm

[8] CUDA vs. OpenCL discussion in Nvidia user forums
http://forums.nvidia.com/index.php?showtopic=156892