CS 677: Parallel Programming for Many-core Processors
Lecture 11

Instructor: Philippos Mordohai

Webpage: www.cs.stevens.edu/~mordohai

E-mail: Philippos.Mordohai@stevens.edu

Final Project Presentations

- May 1
  - **Submit PPT/PDF file by 4pm**
  - 8 min presentation + 2 min Q&A
- Counts for 15% of total grade

Final Project Presentations

- Target audience: fellow classmates
- Content:
  - Problem description
    - What is the computation and why is it important?
  - Suitability for GPU acceleration
    - Amdahl’s Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
    - Compare with CPU version

Final Project Presentations

- Content (cont.):
  - GPU Implementation
    - Which steps of the algorithm were ported to the GPU?
    - Work load allocation to threads
    - Use of resources (registers, shared memory, constant memory, etc.)
    - Occupancy achieved
  - Results
    - Experiments performed
    - Timings and comparisons against CPU version

Final Report

- Due May 10 (11:59pm)
- 6-10 pages including figures, tables and references
- Content
  - See presentation instructions
  - Do not repeat course material
- Counts for 20% of total grade
- NO LATE SUBMISSIONS

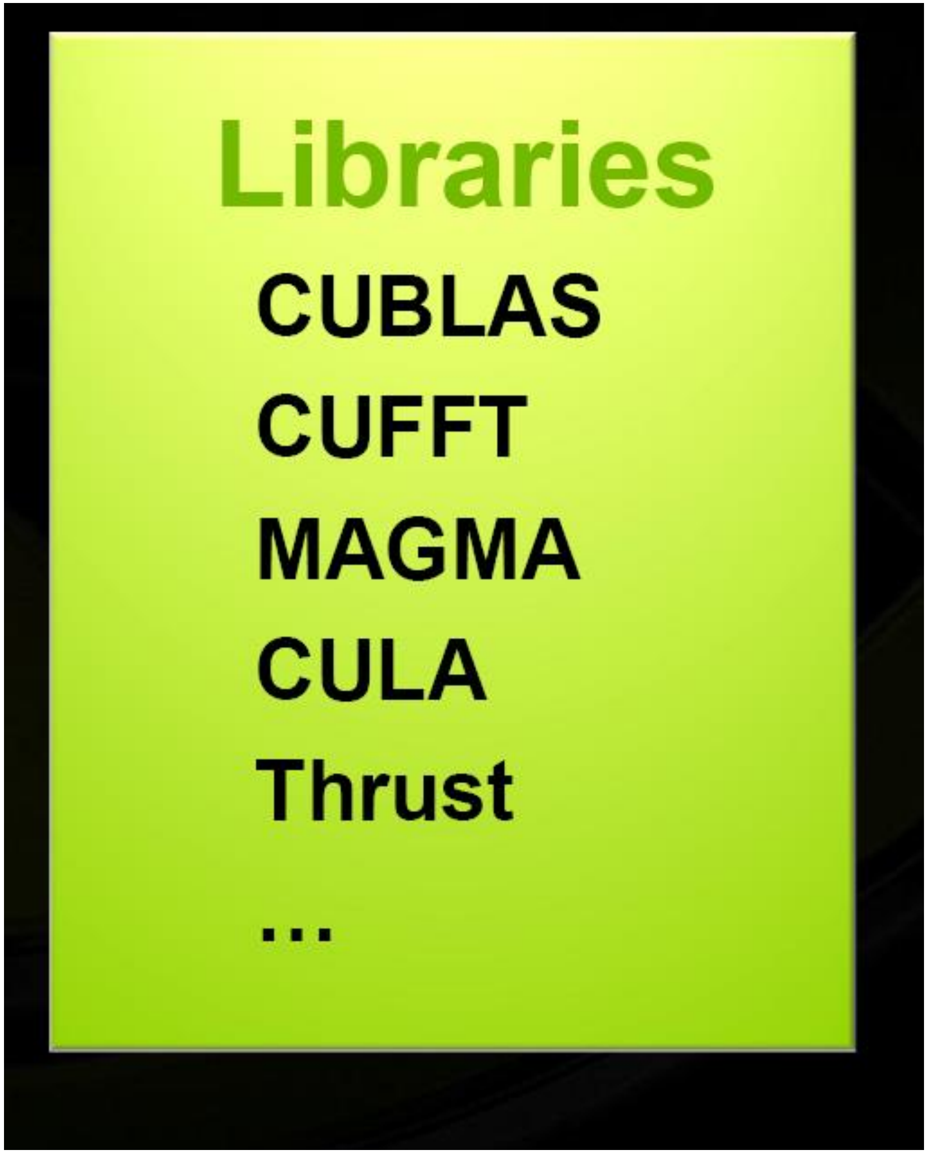
Outline

- More CUDA Libraries
- OpenGL Interface
- Introduction to OpenCL
- Image Convolution Using OpenCL

CUDA Libraries

Joseph Kider
University of Pennsylvania

CIS 565 - Spring 2011



Libraries

CUBLAS

CUFFT

MAGMA

CULA

Thrust

...

CUDA Specialized Libraries: PyCUDA

- PyCUDA lets you access Nvidia‘s CUDA parallel computation API from Python

PyCUDA

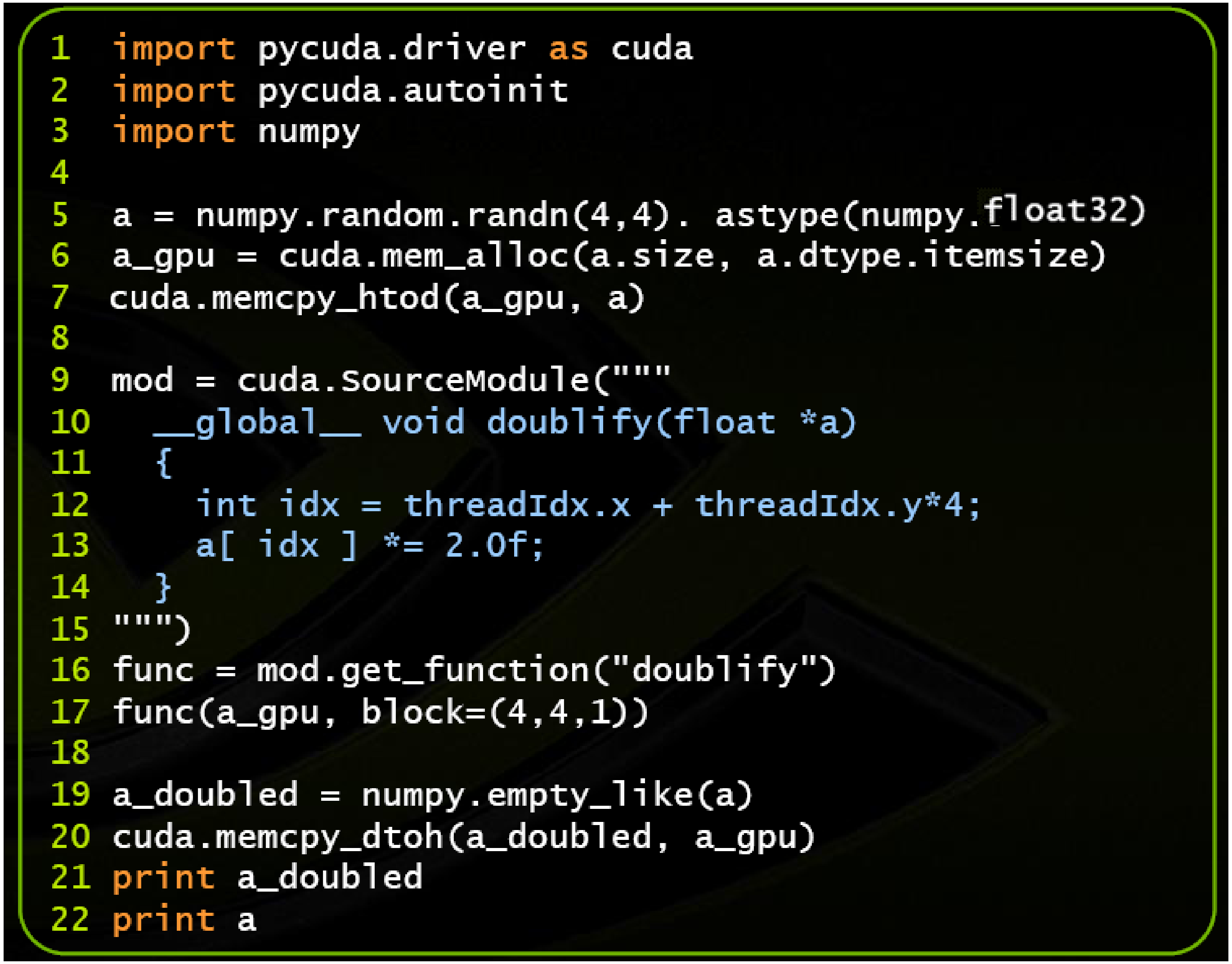
- Third party open source, written by Andreas Klöckner
- Exposes all of CUDA via Python bindings
- Compiles CUDA on the fly
  - CUDA is presented as an interpreted language
- Integrated with numpy
- Handles memory management, resource allocation
- CUDA programs are Python strings
  - Metaprogramming – modify source code on the fly

https://developer.nvidia.com/pycuda

PyCUDA - Differences

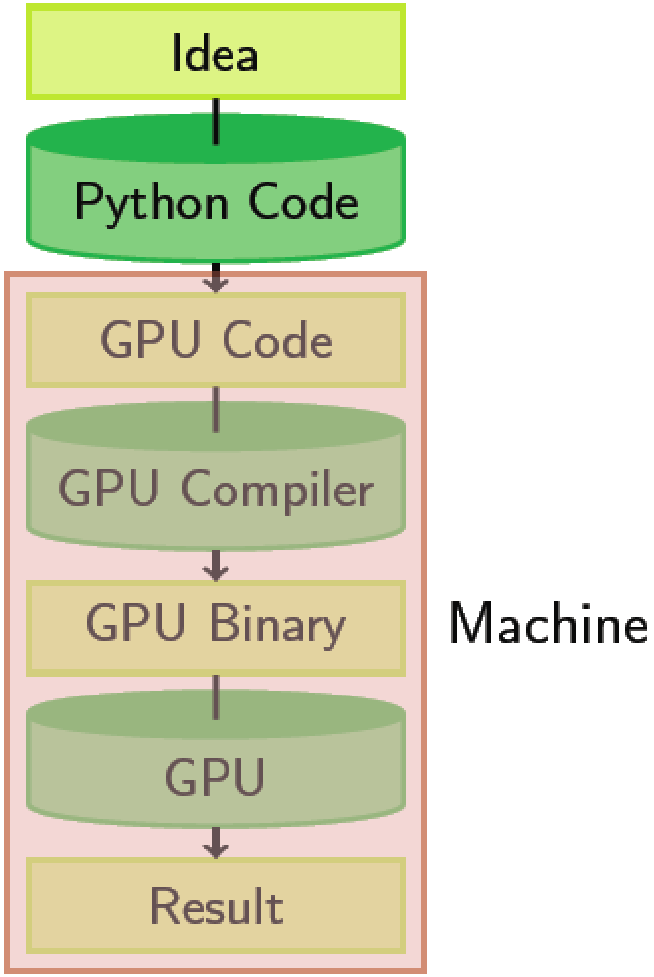
- Object cleanup tied to lifetime of objects
  - Easier to write correct, leak- and crash-free code
  - PyCUDA knows about dependencies, too, so it won't detach from a context before all memory allocated in it is also freed
- Convenience: Abstractions like pycuda.driver.SourceModule and pycuda.gpuarray.GPUArray make CUDA programming even more convenient than with Nvidia's C-based runtime
- Completeness: PyCUDA provides the full power of CUDA's driver API
- Automatic Error Checking: All CUDA errors are automatically translated into Python exceptions
- Speed: PyCUDA's base layer is written in C++

PyCUDA - Example



```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4). astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.size, a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[ idx ] *= 2.0f;
  }
""")
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

CUDA Specialized Libraries: CUDPP

- CUDPP: CUDA Data Parallel Primitives Library
  - CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum (”scan”), parallel sort and parallel reduction

http://cudpp.github.io/

CUDPP - Design Goals

- CUDPP is implemented as 4 layers:
  - The Public Interface is the external library interface, which is the intended entry point for most applications. The public interface calls into the Application-Level API.
  - The Application-Level API comprises functions callable from CPU code. These functions execute code jointly on the CPU (host) and the GPU by calling into the Kernel-Level API below them.
  - The Kernel-Level API comprises functions that run entirely on the GPU across an entire grid of thread blocks. These functions may call into the CTA-Level API below them.
  - The CTA-Level API comprises functions that run entirely on the GPU within a single Cooperative Thread Array (CTA, aka thread block). These are low-level functions that implement core data-parallel algorithms, typically by processing data within shared memory

CUDPP + Thrust

- CUDPP's interface is optimized for performance while Thrust is oriented towards productivity

```
int main(void)
 {
  unsigned int numElements = 32768;

  // allocate host memory
  thrust::host_vector<float> h_idata(numElements);
  // initialize the memory
  thrust::generate(h_idata.begin(), h_idata.end(),
      rand);
```

CUDPP + Thrust

```
// set up plan
CUDPPConfiguration config;
config.op = CUDPP_ADD;
config.datatype = CUDPP_FLOAT;
config.algorithm = CUDPP_SCAN;
config.options = CUDPP_OPTION_FORWARD | CUDPP_OPTION_EXCLUSIVE;

CUDPPHandle scanplan = 0;
CUDPPResult result = cudppPlan(&scanplan, config, numElements,
                            1,0);

if(CUDPP_SUCCESS != result)
{
  printf("Error creating CUDPPPlan\n");
  exit(-1);
}

// Run the scan
cudppScan(scanplan,
          thrust::raw_pointer_cast(&d_odata[0]),
          thrust::raw_pointer_cast(&d_idata[0]),
          numElements);
```

CUDA Specialized Libraries: CUBLAS

- CUDA accelerated BLAS (Basic Linear Algebra Subprograms)

https://developer.nvidia.com/cublas

CUBLAS

- GPU Variant 100 times faster than CPU version
- Matrix size is limited by graphics card memory and texture size
- Although taking advantage of sparse matrices would help reduce memory consumption, sparse matrix storage is not implemented by CUBLAS

CUDA Specialized Libraries: CUFFT

- Cuda Based Fast Fourier Transform Library
- The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets
- One of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing

CUFFT

- Computes parallel FFT on the GPU
- Uses “plans” like FFTW*
  - A plan contains information about optimal configuration for a given transform
  - Plans can prevent recalculation
  - Good fit for CUFFT because different kinds of FFTs require different thread/block configurations

* FFTW is a popular CPU library for FFT

CUFFT

- 1D, 2D and 3D transforms of complex and real-valued data
- Batched execution for doing multiple 1D transforms in parallel
- 1D transform size up to 8M elements
- 2D and 3D transform sizes in the range [2, 16384]
- In-place and out-of-place transforms

CUDA Specialized Libraries: CULA

- CULA is EM Photonics' GPU-accelerated numerical linear algebra library that contains a growing list of LAPACK functions.
- LAPACK stands for Linear Algebra PACKage. It is an industry standard computational library that has been in development for over 15 years and provides a large number of routines for factorization, decomposition, system solvers, and eigenvalue problems.

http://www.culatools.com/

OpenGL Interface

Utah CS 6235
by Mary Hall

OpenGL Rendering

- OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory
  - Vertex buffer objects
  - Pixel buffer objects
- Allows direct visualization of data from computation
  - No device to host transfer
  - Data stays in device memory –very fast compute / viz cycle

  - Data can be accessed from the kernel like any other global data (in device memory)

OpenGL Interoperability

1. Register a buffer object with CUDA
   - cudaGLRegisterBufferObject(GLuintbuffObj);
   - OpenGL can use a registered buffer only as a source
   - Unregister the buffer prior to rendering to it by OpenGL
2. Map the buffer object to CUDA memory
   - cudaGLMapBufferObject(void**devPtr, GLuintbuffObj);
   - Returns an address in global memory
   - Buffer must be registered prior to mapping

OpenGL Interoperability

3. Launch a CUDA kernel to process the buffer
   - Unmap the buffer object prior to use by OpenGL
   - cudaGLUnmapBufferObject(GLuintbuffObj);
4. Unregister the buffer object
   - cudaGLUnregisterBufferObject(GLuintbuffObj);
   - Optional: needed if the buffer is a render target
5. Use the buffer object in OpenGL code

Example from simpleGL in SDK

1. GL calls to create and initialize buffer, then register with CUDA:

```
// create buffer object
glGenBuffers( 1, vbo);
glBindBuffer( GL_ARRAY_BUFFER, *vbo);

// initialize buffer object
unsigned int size = mesh_width * mesh_height * 4 *
  sizeof( float)*2;
glBufferData( GL_ARRAY_BUFFER, size, 0,
  GL_DYNAMIC_DRAW);
glBindBuffer( GL_ARRAY_BUFFER, 0);

// register buffer object with CUDA
cudaGLRegisterBufferObject(*vbo);
```

Example from simpleGL in SDK

2. Map OpenGL buffer object for writing from CUDA

```
float4 *dptr;
cudaGLMapBufferObject( (void**)&dptr, vbo));
```

3. Execute the kernel to compute values for dptr

```
dim3 block(8, 8, 1);
dim3 grid(mesh_width / block.x, mesh_height
  / block.y, 1);
kernel<<< grid, block>>>(dptr, mesh_width,
  mesh_height, anim);
```

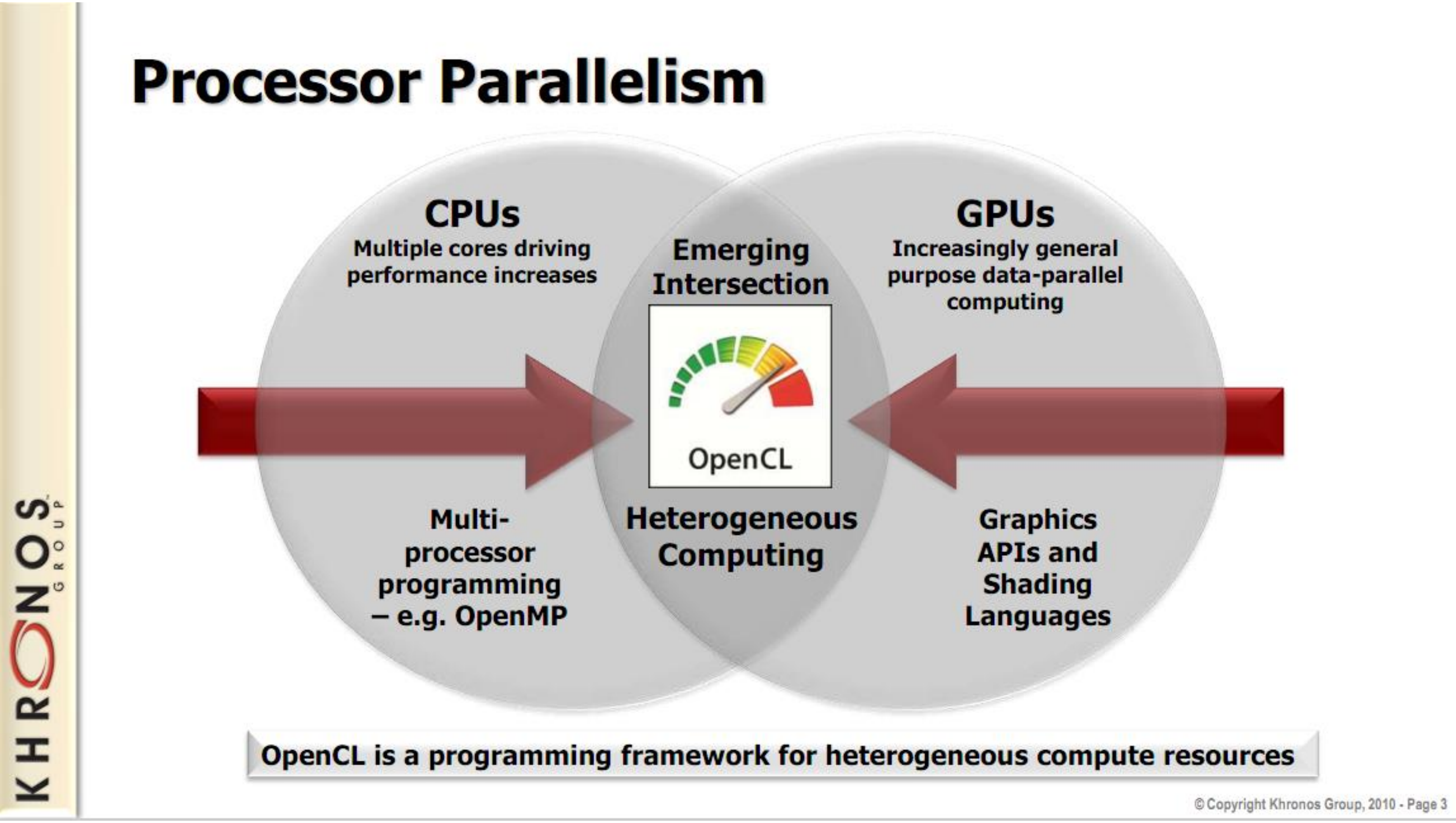
4. Unregister the OpenGL buffer object and return to Open GL

```
 cudaGLUnmapBufferObject( vbo);
```

OpenCL

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

with additional material from
Joseph Kider
University of Pennsylvania
CIS 565 - Spring 2009



Processor Parallelism

CPUs
Multiple cores driving performance increases

Emerging Intersection

OpenCL

GPUs
Increasingly general purpose data-parallel computing

Multi-processor programming – e.g. OpenMP

Heterogeneous Computing

Graphics APIs and Shading Languages

OpenCL is a programming framework for heterogeneous compute resources

Image from: http://www.khronos.org/developers/library/overview/opencl_overview.pdf

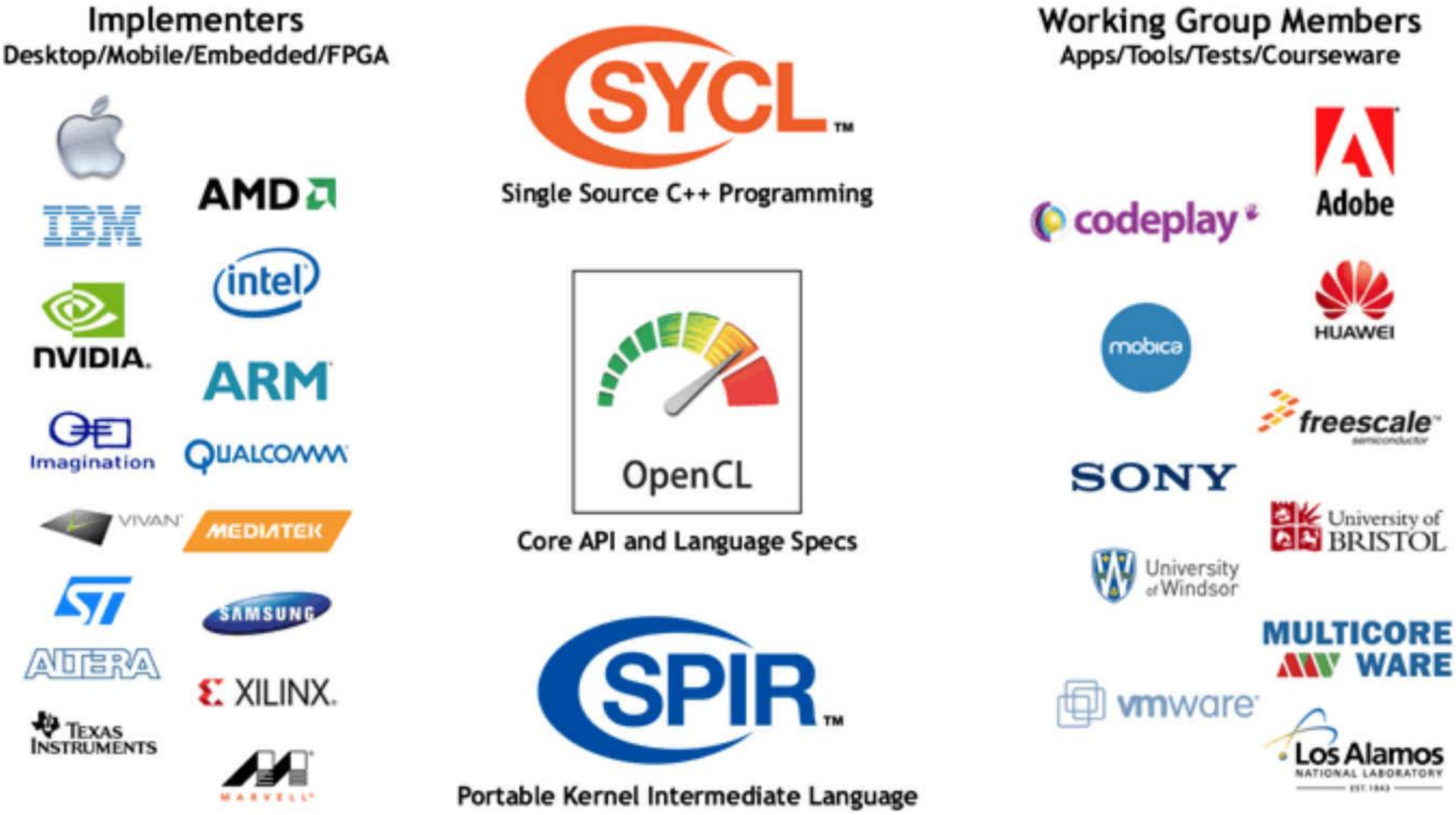
OpenCL

- Open Compute Language
- For heterogeneous parallel-computing systems
- Cross-platform
  - Implementations for
    - ATI GPUs
    - NVIDIA GPUs
    - x86 CPUs
  - Is cross-platform really *one size fits all*?

OpenCL

- Standardized
- Initiated by Apple
- Developed by the Khronos Group

OpenCL Ecosystem



Implementers
Desktop/Mobile/Embedded/FPGA

Single Source C++ Programming

Core API and Language Specs

Portable Kernel Intermediate Language

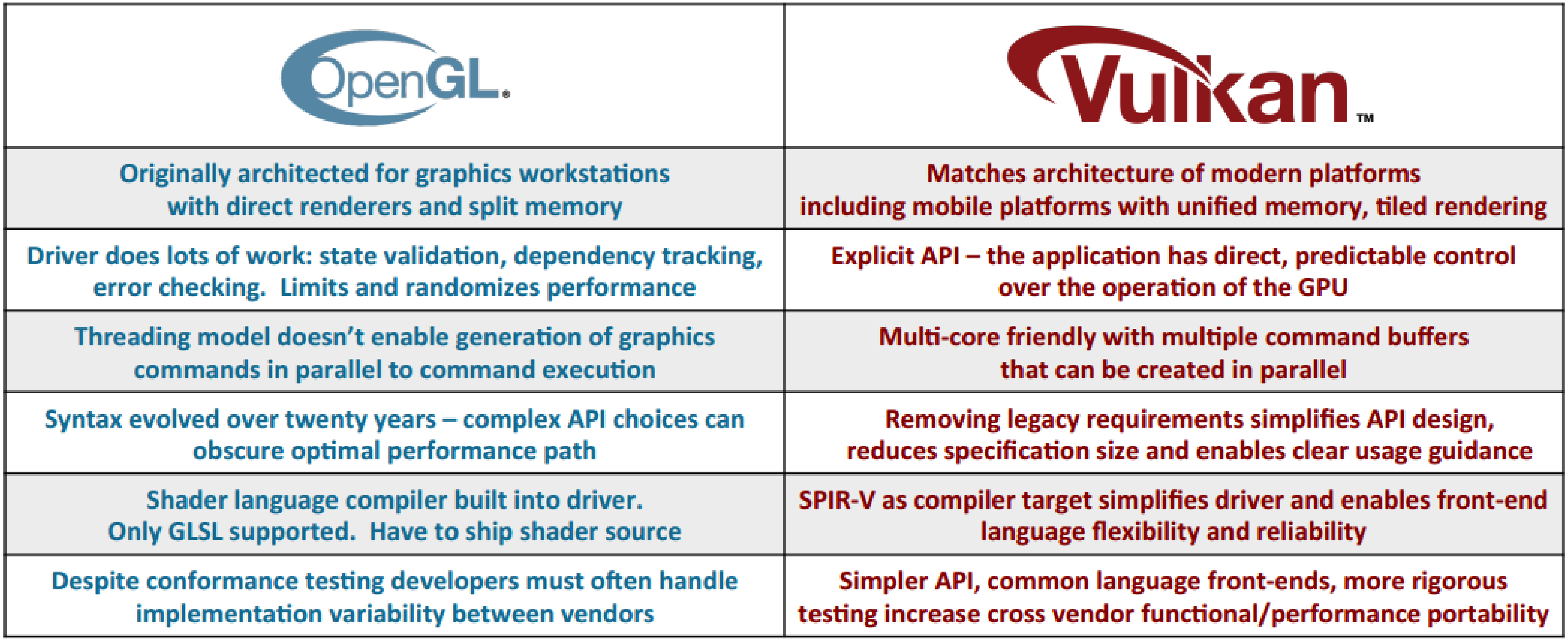
Working Group Members
Apps/Tools/Tests/Courseware

Image from: http://www.khronos.org/opencl/

SPIR

- Standard Portable Intermediate Representation
  - SPIR-V is first open standard, cross-API, intermediate language for natively representing parallel compute and graphics
  - Part of the core specification of:
    - OpenCL 2.1
    - the new Vulkan graphics and compute API

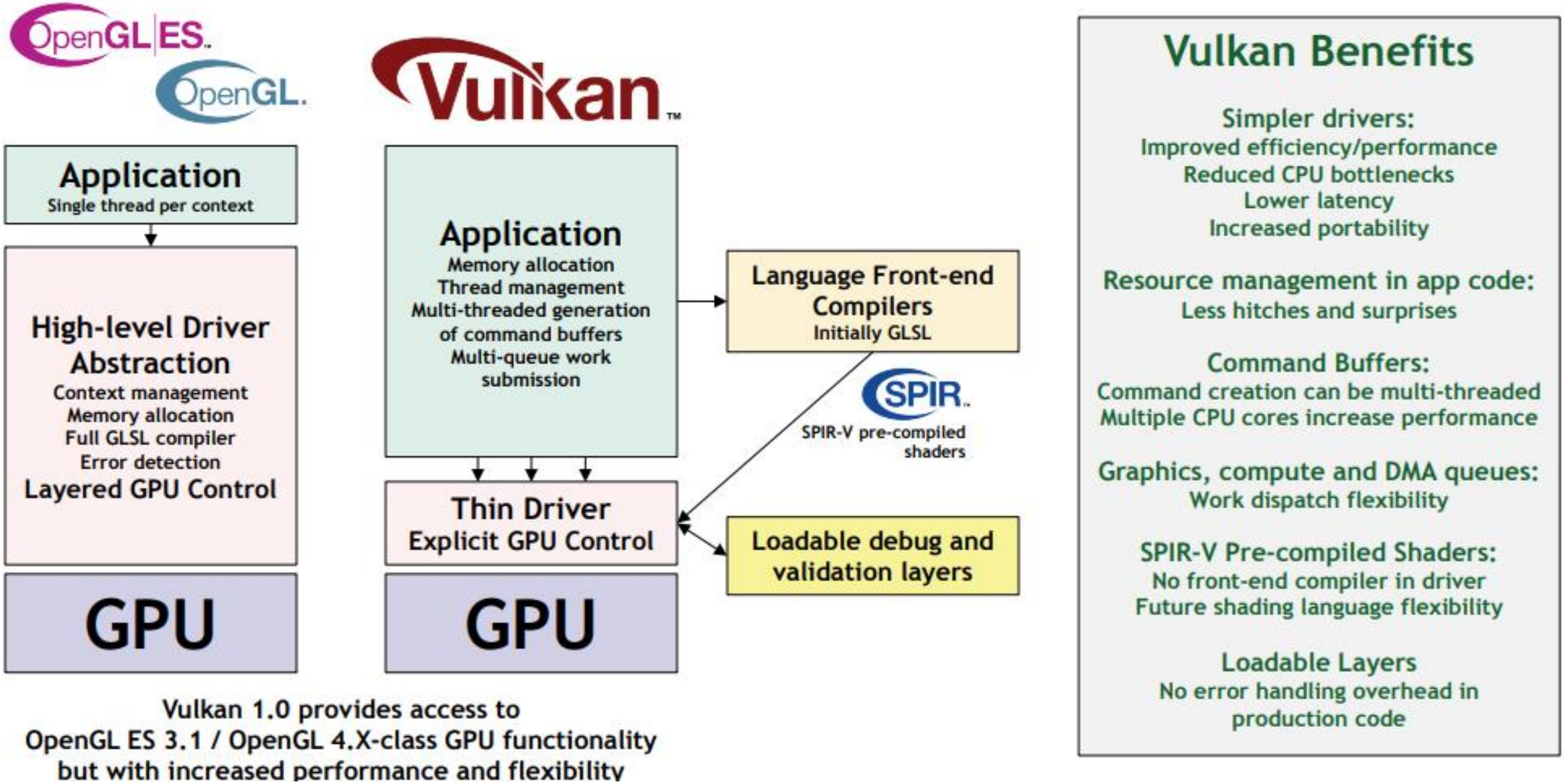
Vulkan



OpenGL.	Vulkan™
Originally architected for graphics workstations with direct renderers and split memory	Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering
Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance	Explicit API – the application has direct, predictable control over the operation of the GPU
Threading model doesn't enable generation of graphics commands in parallel to command execution	Multi-core friendly with multiple command buffers that can be created in parallel
Syntax evolved over twenty years – complex API choices can obscure optimal performance path	Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance
Shader language compiler built into driver. Only GLSL supported. Have to ship shader source	SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability
Despite conformance testing developers must often handle implementation variability between vendors	Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability

Source: www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf

Vulkan



Application
Single thread per context

High-level Driver Abstraction
Context management
Memory allocation
Full GLSL compiler
Error detection
Layered GPU Control

GPU

Application
Memory allocation
Thread management
Multi-threaded generation of command buffers
Multi-queue work submission

Thin Driver
Explicit GPU Control

GPU

Language Front-end Compilers
Initially GLSL

SPIR-V pre-compiled shaders

Loadable debug and validation layers

Vulkan 1.0 provides access to
OpenGL ES 3.1 / OpenGL 4.X-class GPU functionality
but with increased performance and flexibility

Vulkan Benefits

Simpler drivers:
Improved efficiency/performance
Reduced CPU bottlenecks
Lower latency
Increased portability

Resource management in app code:
Less hitches and surprises

Command Buffers:
Command creation can be multi-threaded
Multiple CPU cores increase performance

Graphics, compute and DMA queues:
Work dispatch flexibility

SPIR-V Pre-compiled Shaders:
No front-end compiler in driver
Future shading language flexibility

Loadable Layers
No error handling overhead in production code

Source: https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf

Design Goals of OpenCL

- Use all computational resources in the system
  - GPUs and CPUs as peers
  - Data- and task-parallel computing
- Efficient parallel programming model
  - Based on C
  - Abstract the specifics of underlying hardware
  - Define maximum allowable errors of math functions
- Drive future hardware requirements

OpenCL

- API similar to OpenGL
- Based on the C language
- Easy transition form CUDA to OpenCL

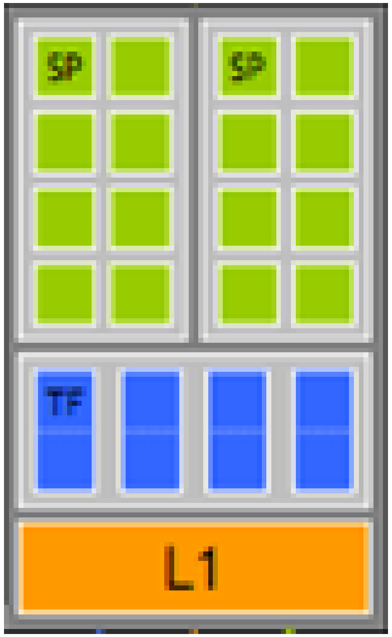
OpenCL and CUDA

- Many OpenCL features have a one to one mapping to CUDA features
- OpenCL
  - More complex platform and device management
  - More complex kernel launch

➢ OpenCL is more complex due to its support for multiplatform and multivendor portability

OpenCL and CUDA

- Compute Unit (CU) corresponds to
  - CUDA streaming multiprocessor (SMs)
  - CPU core
  - etc.
- Processing Element corresponds to
  - CUDA streaming processor (SP)
  - CPU ALU



OpenCL and CUDA

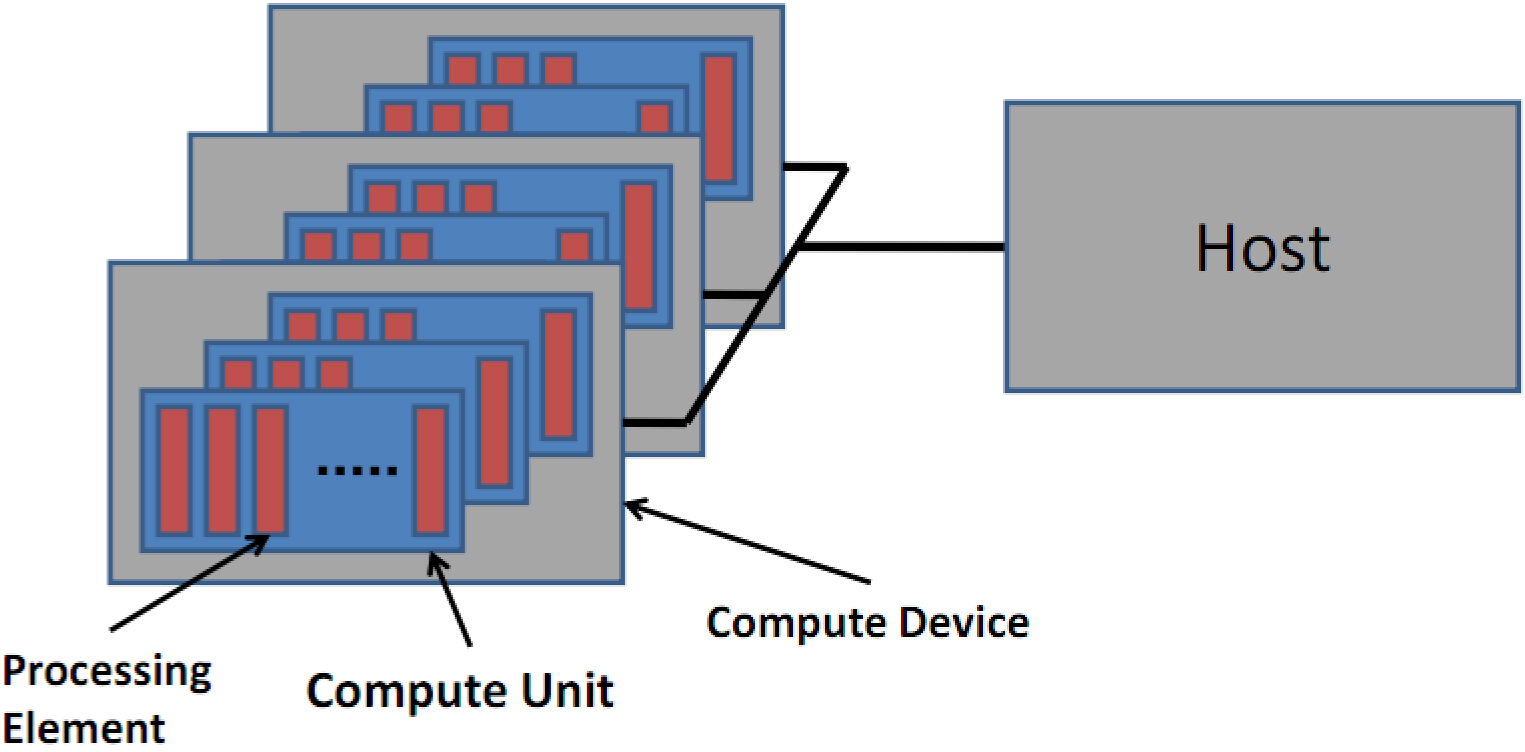


Image from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL and CUDA

CUDA	OpenCL
Kernel	Kernel
Host program	Host program
Thread	Work item
Block	Work group
Grid	NDRange (index space)

OpenCL and CUDA

- Work Item (CUDA thread) - executes kernel code
- Index Space (CUDA grid) - defines work items and how data is mapped to them
- Work Group (CUDA block) - work items in a work group can synchronize

OpenCL and CUDA

- CUDA: `threadIdx` and `blockIdx`
  - Combine to create a global thread ID
  - Example
    - `blockIdx.x * blockDim.x + threadIdx.x`

OpenCL and CUDA

- OpenCL: each thread has a unique global index
  - Retrieve with `get_global_id()`

CUDA	OpenCL
`threadIdx.x`	`get_local_id(0)`
`blockIdx.x * blockDim.x + threadIdx.x`	`get_global_id(0)`

OpenCL and CUDA

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

OpenCL and CUDA

- Recall CUDA:

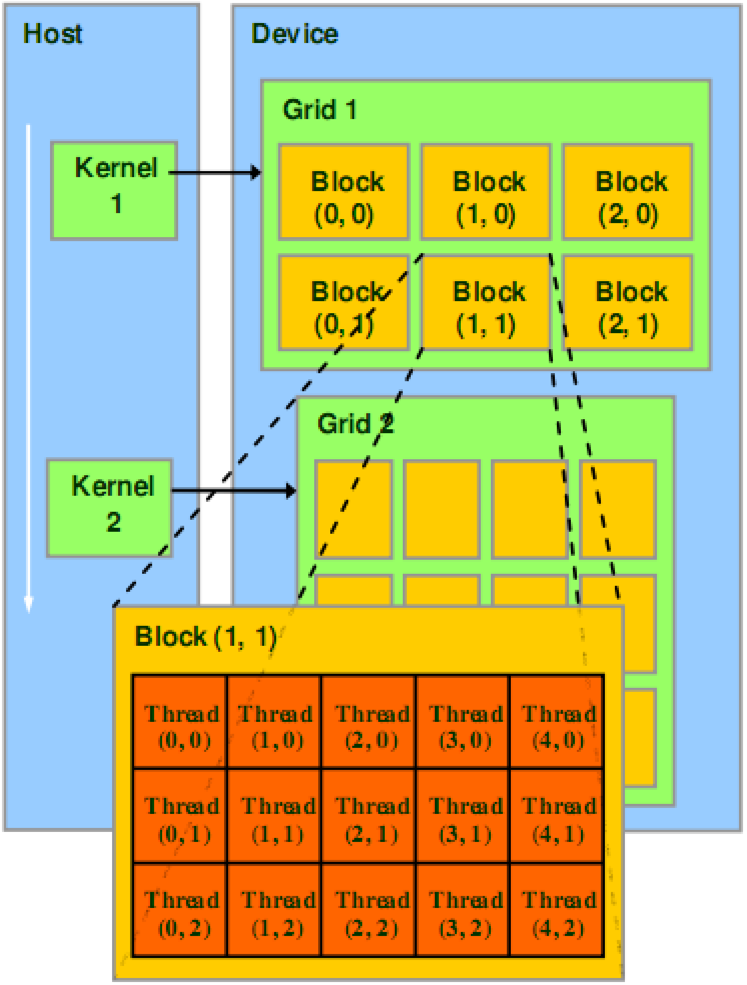
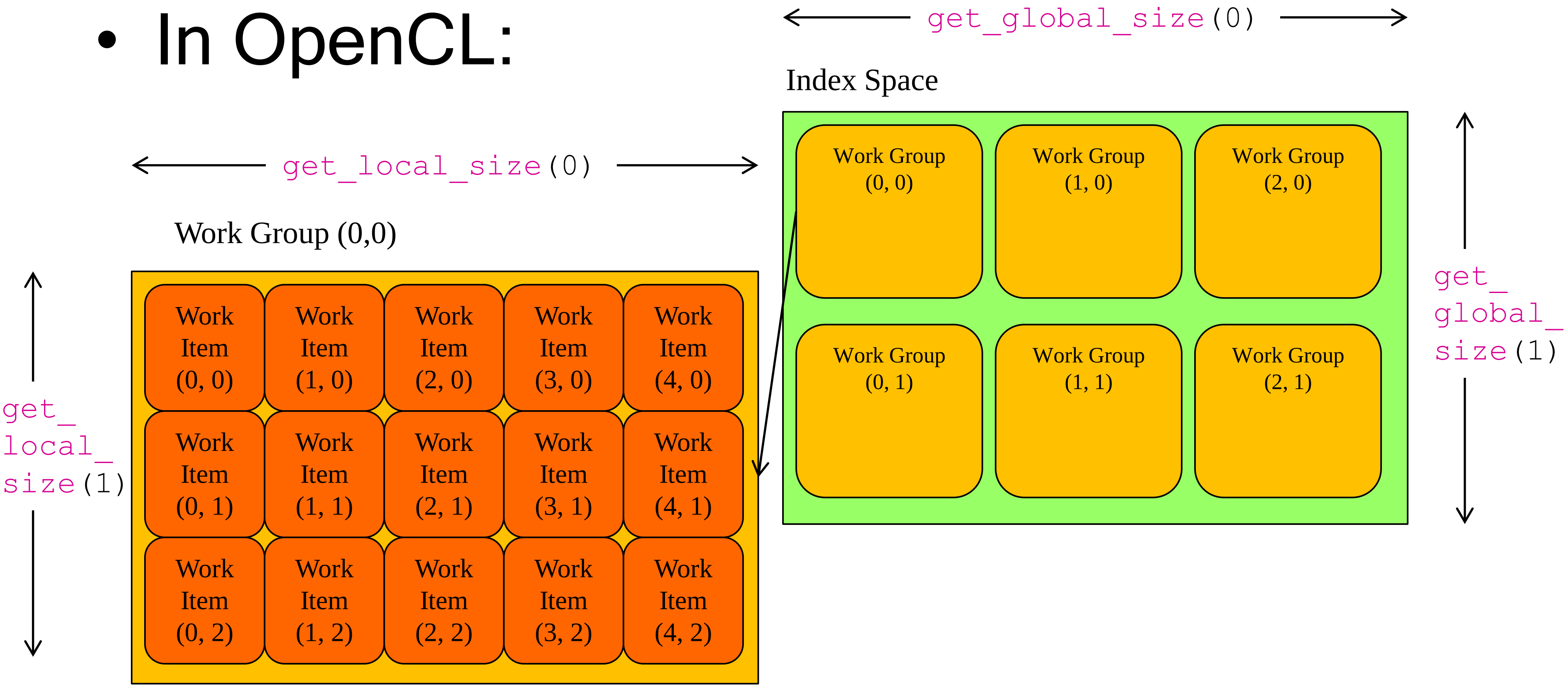


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

OpenCL and CUDA

- In OpenCL:



get_global_size(0)

Index Space

Work Group (0, 0)	Work Group (1, 0)	Work Group (2, 0)
Work Group (0, 1)	Work Group (1, 1)	Work Group (2, 1)

get_global_size(1)

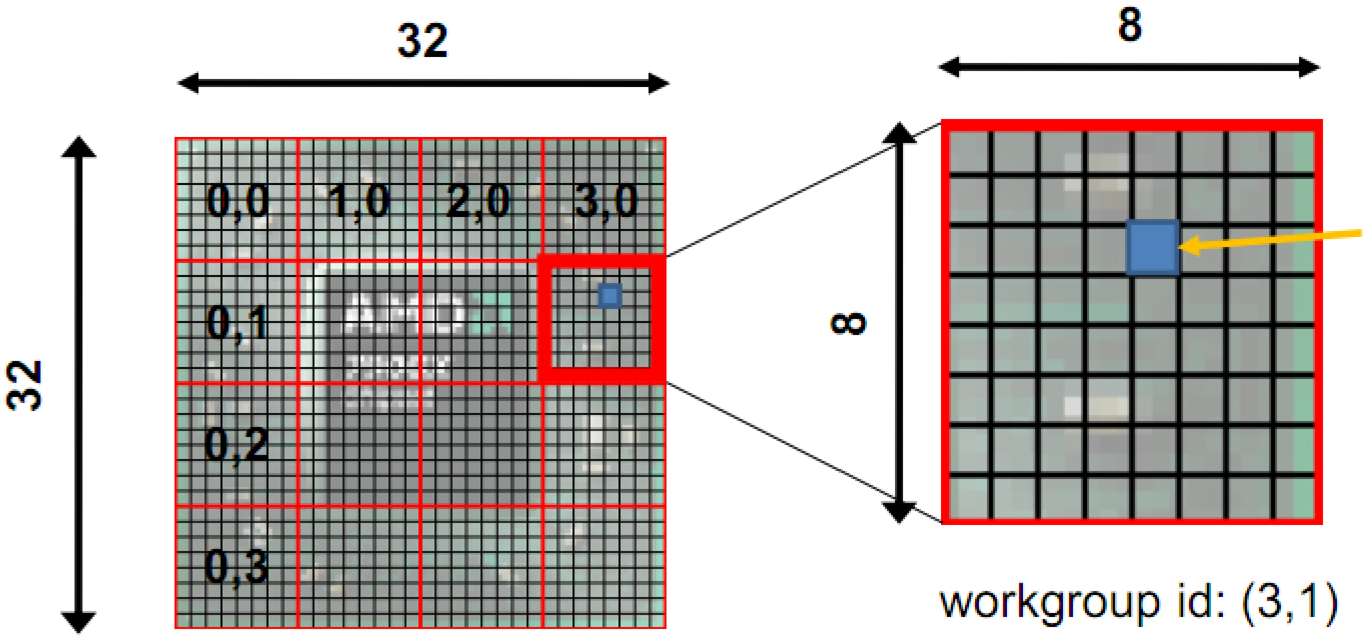
get_local_size(0)

Work Group (0,0)

Work Item (0, 0)	Work Item (1, 0)	Work Item (2, 0)	Work Item (3, 0)	Work Item (4, 0)
Work Item (0, 1)	Work Item (1, 1)	Work Item (2, 1)	Work Item (3, 1)	Work Item (4, 1)
Work Item (0, 2)	Work Item (1, 2)	Work Item (2, 2)	Work Item (3, 2)	Work Item (4, 2)

get_local_size(1)

Kernels: Work-item and Work-group Example



local id: (4,2)
global id: (28,10)

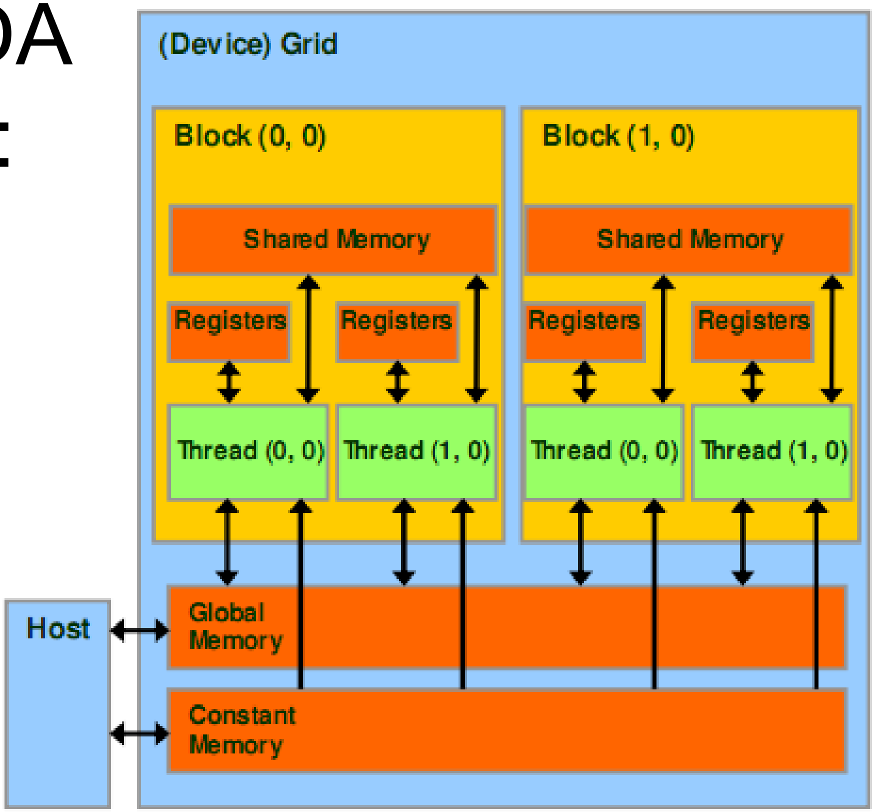
workgroup id: (3,1)
local size: 8x8=64

dimension: 2
global size: 32x32=1024
num of groups: 16

Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL and CUDA

- Recall the CUDA memory model:



OpenCL and CUDA

- In OpenCL:

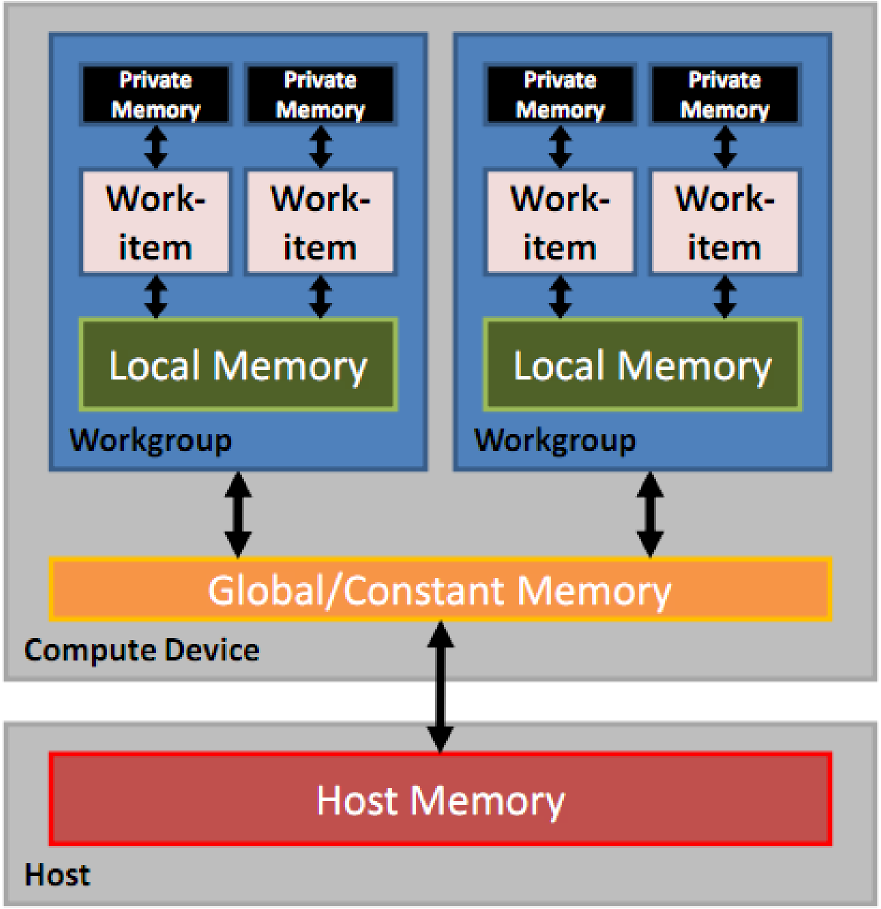
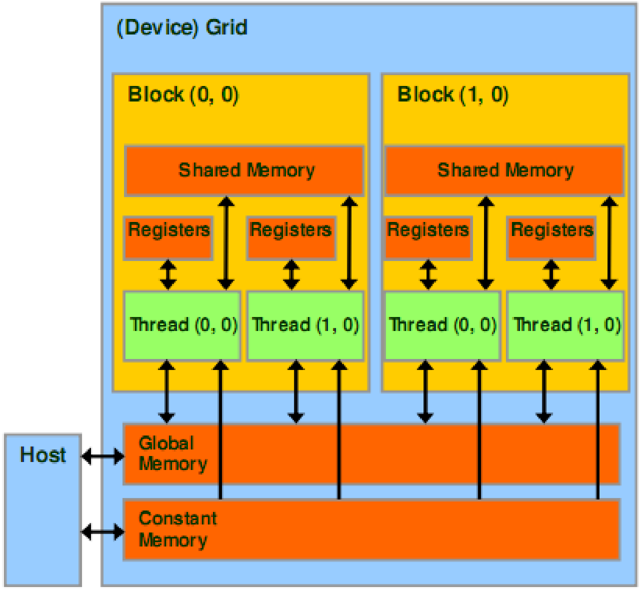
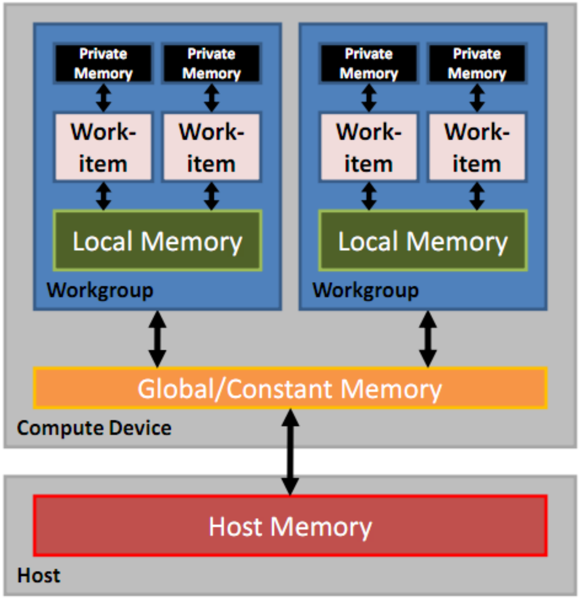


Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL and CUDA

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





OpenCL and CUDA

CUDA	Host Access	Device Access	OpenCL
Global memory	Dynamic allocation; read/write access	No allocation; read/write access by all work items in all work groups; large and slow but may be cached in some devices	Global memory
Constant memory	Dynamic allocation; read/write access	Static allocation; read only access by all work items	Constant memory
Shared memory	Dynamic allocation; no access	Static allocation; shared read/write access by all work items in a work group	Local memory
Local memory	No allocation; no access	Static allocation; read/write access by a single work item	Private memory

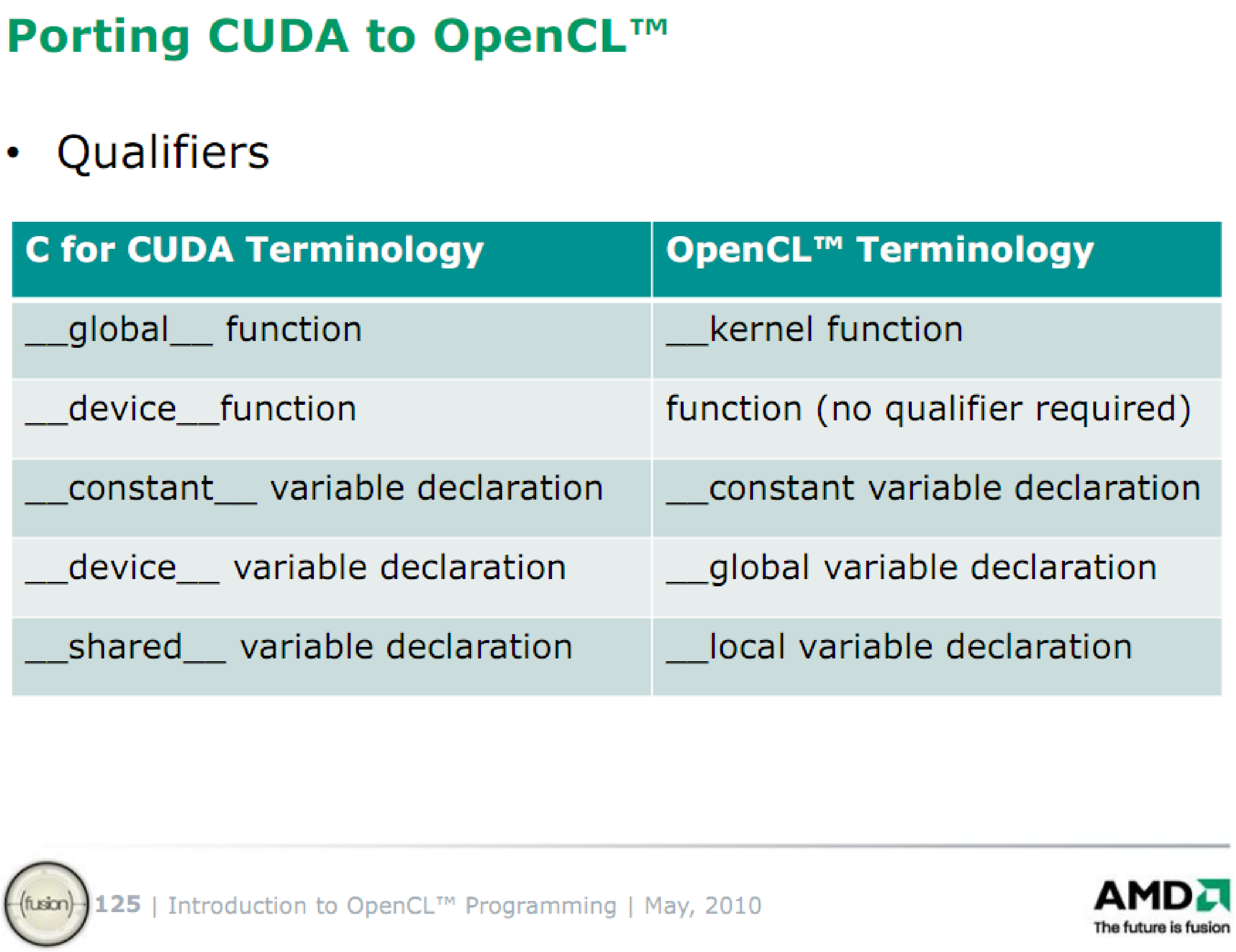
OpenCL and CUDA

CUDA	OpenCL
`__syncthreads()`	`__barrier()`

- Both also have Fences
  - In OpenCL
    - `mem_fence()`
    - `read_mem_fence()`
    - `write_mem_fence()`

OpenCL Fence Examples

- `mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
  - waits until all reads/writes to local and/or global memory made by the calling work item prior to `mem_fence()` are visible to all threads in the work-group
- `barrier(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
  - waits until all work-items in the work-group have reached this point and calls `mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`



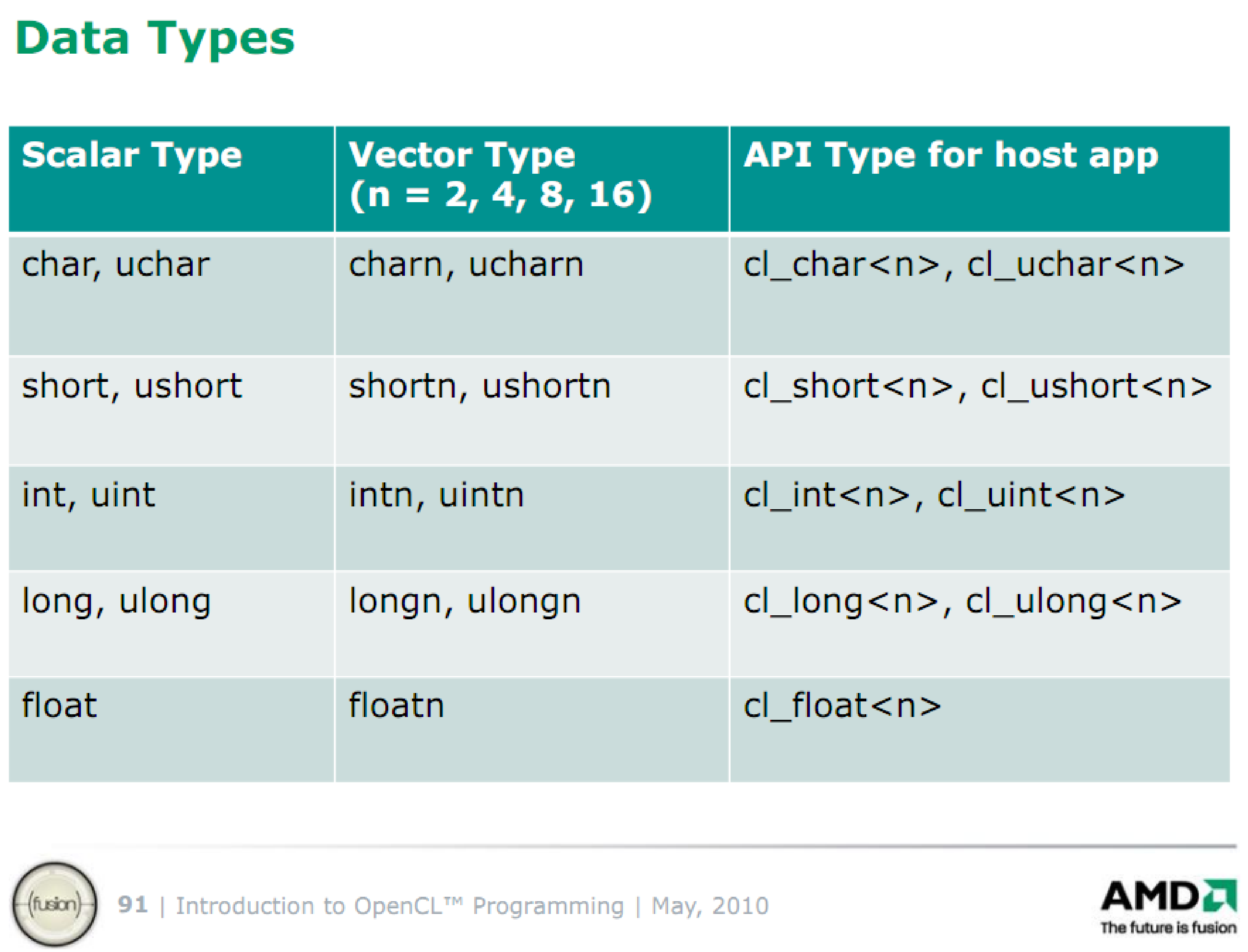
Porting CUDA to OpenCL™

- Qualifiers

C for CUDA Terminology	OpenCL™ Terminology
__global__ function	__kernel function
__device__function	function (no qualifier required)
__constant__ variable declaration	__constant variable declaration
__device__ variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

125 | Introduction to OpenCL™ Programming | May, 2010

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

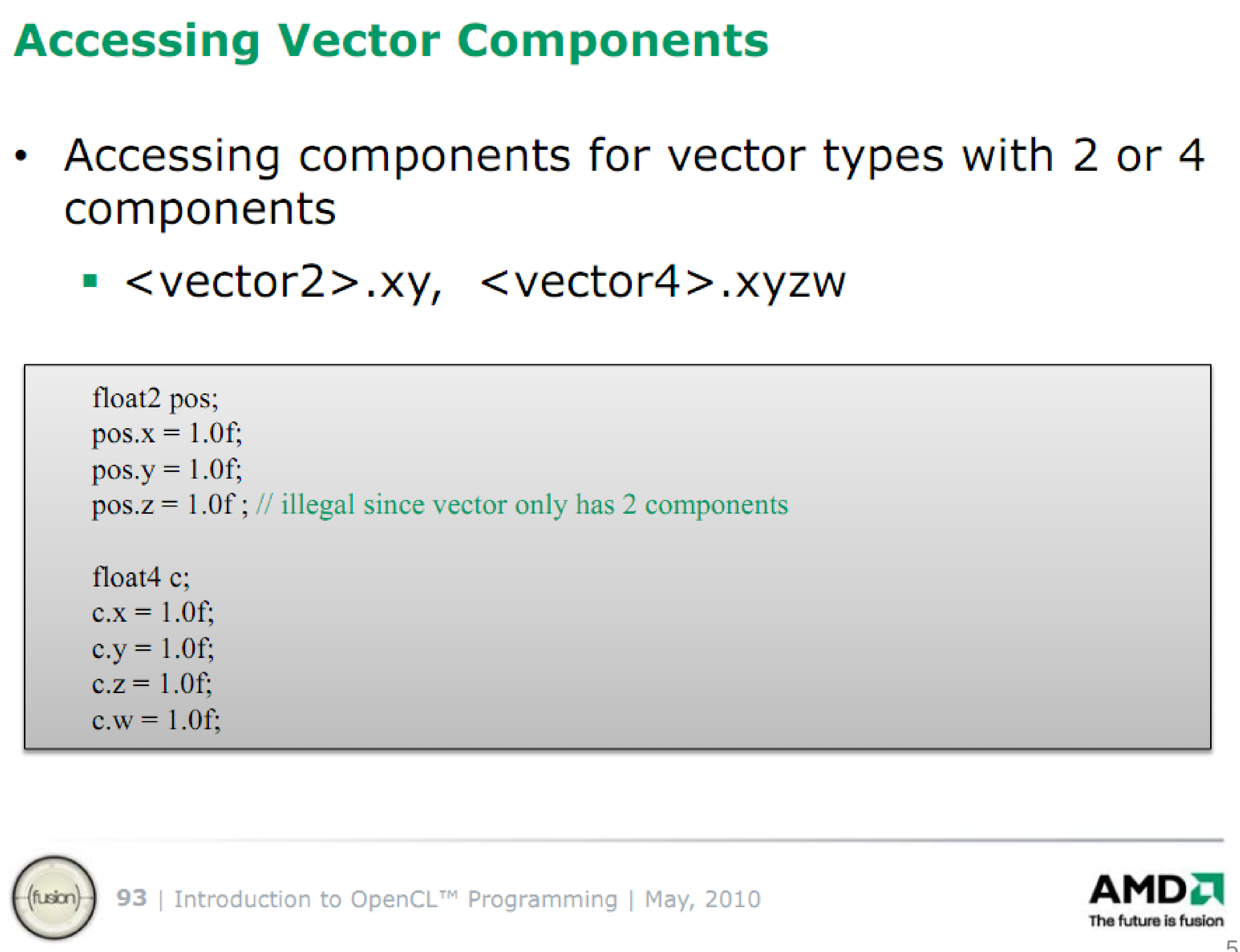


Data Types

Scalar Type	Vector Type (n = 2, 4, 8, 16)	API Type for host app
char, uchar	charn, ucharn	cl_char<n>, cl_uchar<n>
short, ushort	shortn, ushortn	cl_short<n>, cl_ushort<n>
int, uint	intn, uintn	cl_int<n>, cl_uint<n>
long, ulong	longn, ulongn	cl_long<n>, cl_ulong<n>
float	floatn	cl_float<n>

AMD
The future is fusion

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

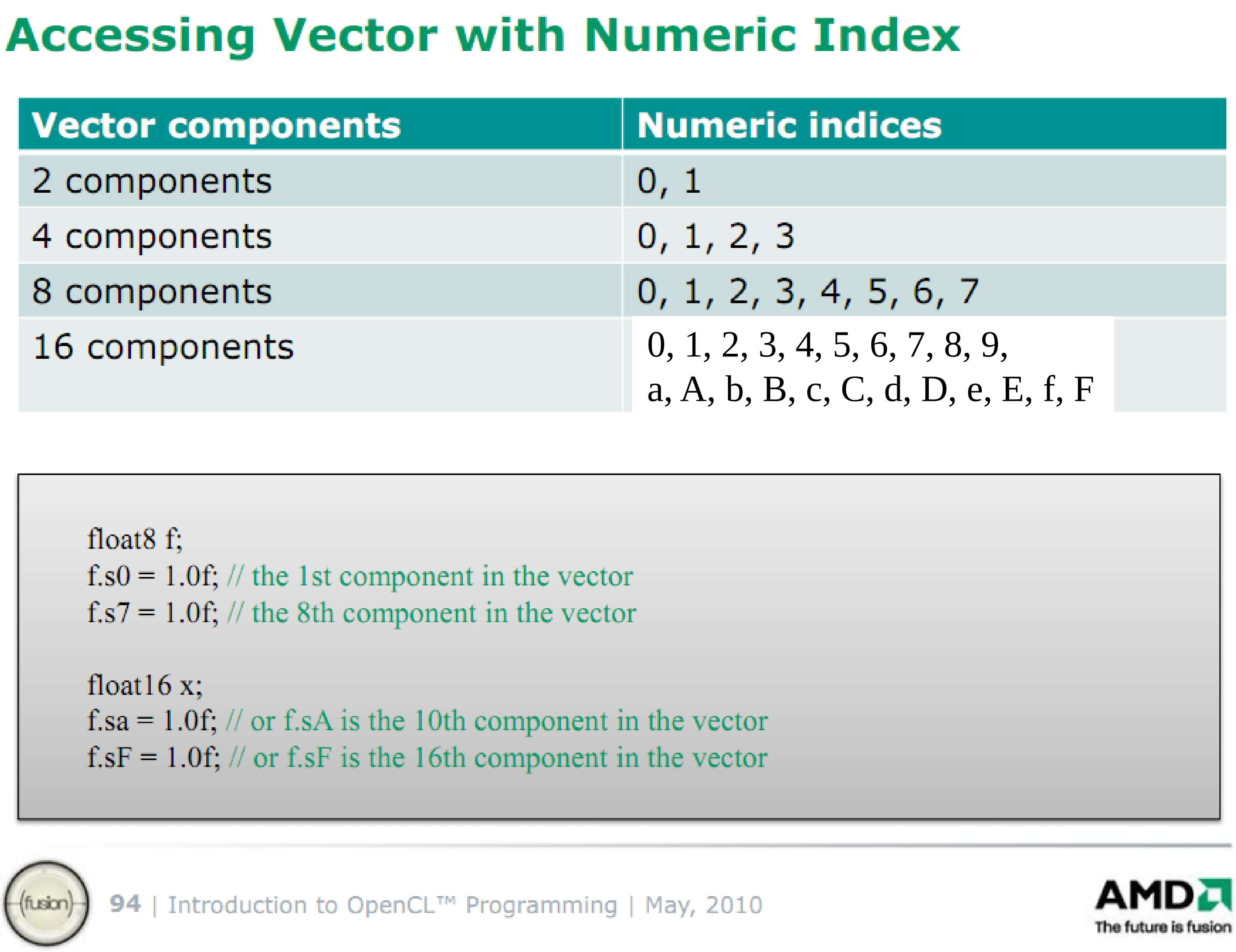


Accessing Vector Components

- Accessing components for vector types with 2 or 4 components
  - <vector2>.xy, <vector4>.xyzw

```
float2 pos;
pos.x = 1.0f;
pos.y = 1.0f;
pos.z = 1.0f ; // illegal since vector only has 2 components

float4 c;
c.x = 1.0f;
c.y = 1.0f;
c.z = 1.0f;
c.w = 1.0f;
```

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

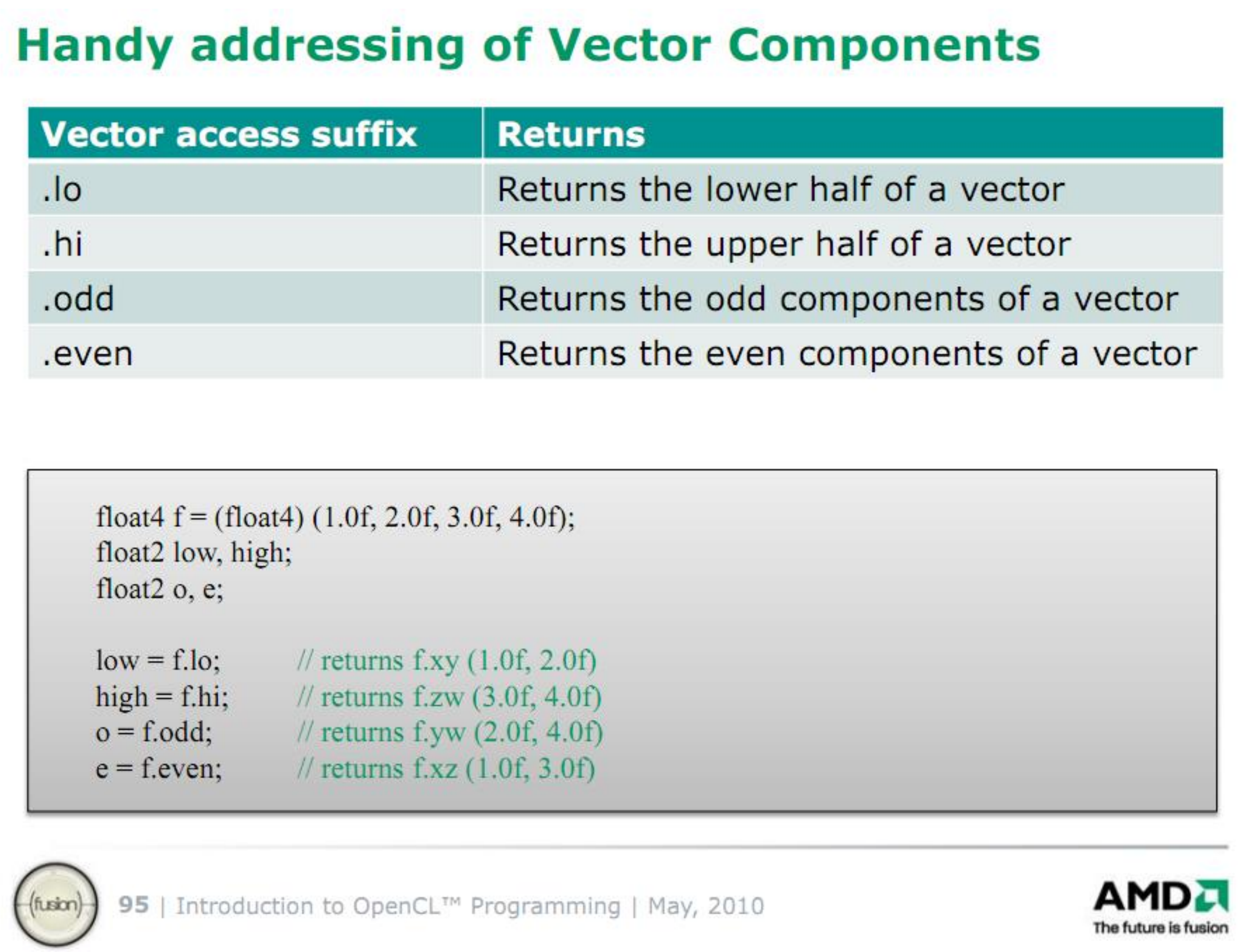


Accessing Vector with Numeric Index

Vector components	Numeric indices
2 components	0, 1
4 components	0, 1, 2, 3
8 components	0, 1, 2, 3, 4, 5, 6, 7
16 components	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

```
float8 f;
f.s0 = 1.0f; // the 1st component in the vector
f.s7 = 1.0f; // the 8th component in the vector

float16 x;
f.sa = 1.0f; // or f.sA is the 10th component in the vector
f.sF = 1.0f; // or f.sF is the 16th component in the vector
```

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx



Handy addressing of Vector Components

Vector access suffix	Returns
.lo	Returns the lower half of a vector
.hi	Returns the upper half of a vector
.odd	Returns the odd components of a vector
.even	Returns the even components of a vector

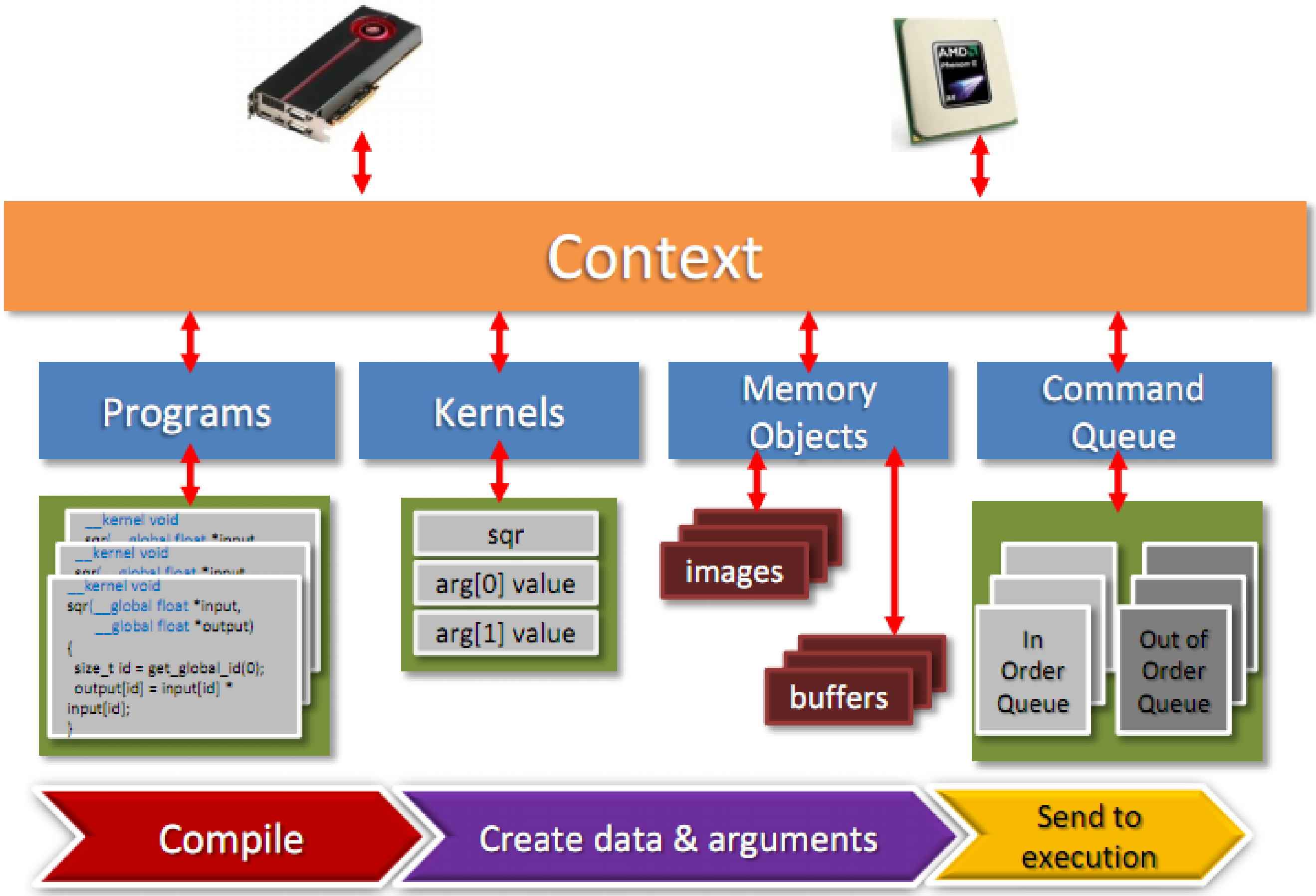
```
float4 f = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
float2 low, high;
float2 o, e;

low = f.lo;         // returns f.xy (1.0f, 2.0f)
high = f.hi;        // returns f.zw (3.0f, 4.0f)
o = f.odd;          // returns f.yw (2.0f, 4.0f)
e = f.even;         // returns f.xz (1.0f, 3.0f)
```

(fusion) 95 | Introduction to OpenCL™ Programming | May, 2010

AMD The future is fusion

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL™ Program Flow



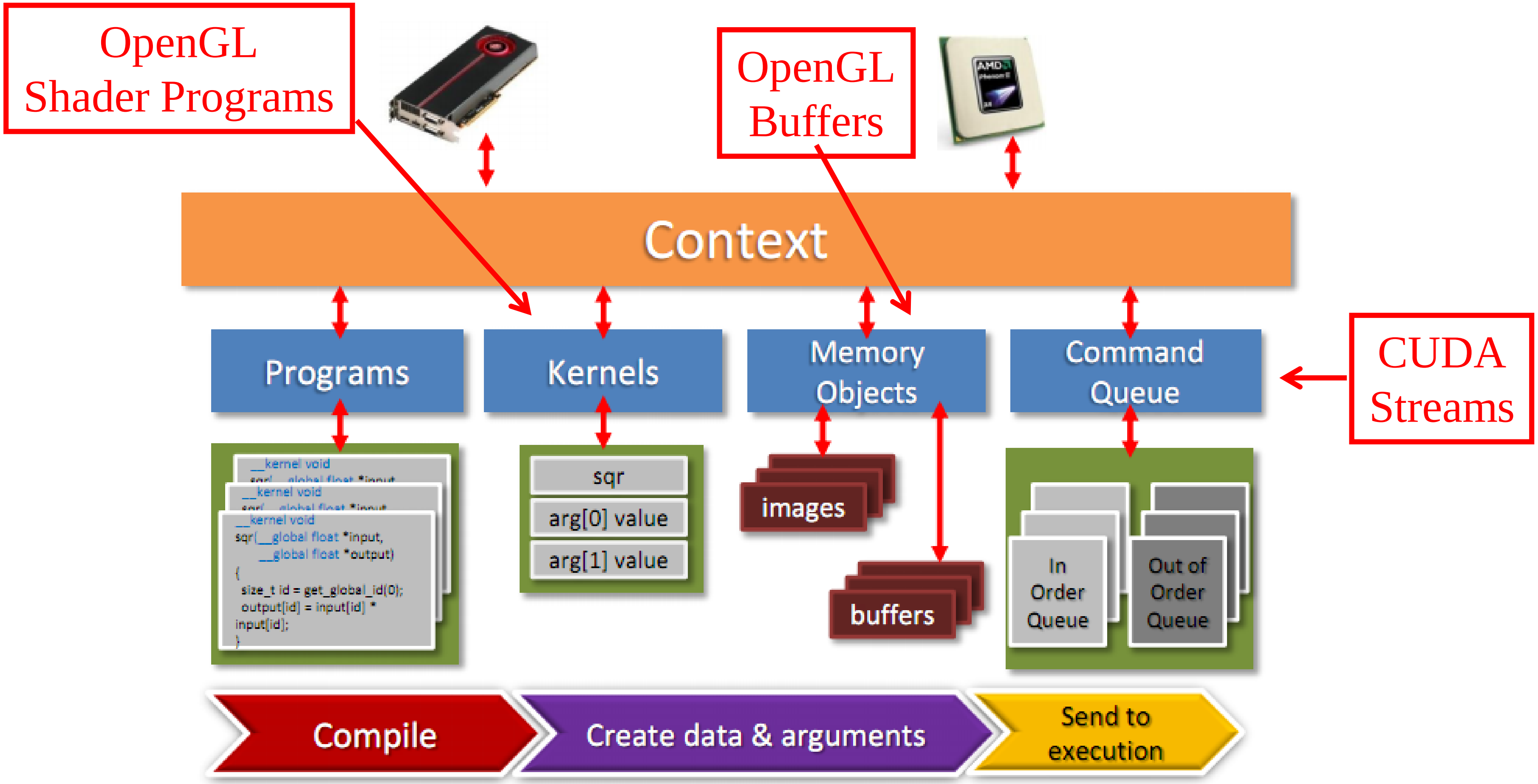
```
__kernel void
sqr(__global float *input,
    __global float *output)
{
  size_t id = get_global_id(0);
  output[id] = input[id] *
input[id];
}
```

46 | Introduction to OpenCL™ Programming | May, 2010

The future is fusion

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL™ Program Flow



OpenGL Shader Programs

OpenGL Buffers

CUDA Streams

Context

Programs

Kernels

Memory Objects

Command Queue

```
__kernel void
sqr(__global float *input,
    __global float *output)
{
 size_t id = get_global_id(0);
 output[id] = input[id] *
input[id];
}
```

sqr

arg[0] value

arg[1] value

images

buffers

In Order Queue

Out of Order Queue

Compile

Create data & arguments

Send to execution

AMD The future is fusion

OpenCL API

- Walkthrough OpenCL **host** code for running `vecAdd` kernel:

```
__kernel void vecAdd(__global const
  float *a, __global const float *b,
  __global float *c)
{
   int i = get_global_id(0);
   c[i] = a[i] + b[i];
}
```

OpenCL API

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
  CL_DEVICE_TYPE_GPU, 0, 0, 0);
```

OpenCL API

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
  CL_DEVICE_TYPE_GPU, 0, 0, 0);
```

Create a context for a GPU

OpenCL API

```
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  0, 0, &nContextDescriptorSize);
cl_device_id aDevices =
  malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  nContextDescriptorSize, aDevices, 0);
```

OpenCL API

```
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  0, 0, &nContextDescriptorSize);
cl_device_id aDevices =
  malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  nContextDescriptorSize, aDevices, 0);
```

Retrieve an array of each GPU

Choosing Devices

- A system may have several devices - which is best?
- The "best" device is algorithm-dependent
- Query device info with: `clGetDeviceInfo(device, param_name, *value)`
  - Number of compute units `CL_DEVICE_MAX_COMPUTE_UNITS`
  - Clock frequency `CL_DEVICE_CLOCK_FREQUENCY`
  - Memory size `CL_DEVICE_GLOBAL_MEM_SIZE`
  - Extensions (double precision, atomics, etc.)
- Pick best device for your algorithm

OpenCL API

```
// create a command queue for first
// device the context reported
cl_command_queue hCmdQueue;
hCmdQueue =
  clCreateCommandQueue(hContext,
    aDevices[0], 0, 0);
```

OpenCL API

```
// create a command queue for first
// device the context reported
cl_command_queue hCmdQueue;
hCmdQueue =
  clCreateCommandQueue(hContext,
    aDevices[0], 0, 0);
```

Create a command queue (CUDA stream) for the first GPU

OpenCL API

```
// create & compile program
cl_program hProgram;
hProgram =
  clCreateProgramWithSource(hContext,
     1, source, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0,
  0);
```

- A program contains one or more kernels. Think dll.
- Provide kernel source as a string
- Can also compile offline

OpenCL API

```
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram,
  “vecAdd”, 0);
```

Create kernel from program

Program and Kernel Objects

- Program objects encapsulate:
  - a program source or binary
  - list of devices and latest successfully built executable for each device
  - a list of kernel objects
- Kernel objects encapsulate:
  - a specific kernel function in a program - declared with the kernel qualifier
  - argument values
  - kernel objects created after the program executable has been built

OpenCL API

```
// allocate host vectors
float* pA = new float[cnDimension];
float* pB = new float[cnDimension];
float* pC = new float[cnDimension];

// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);
```

OpenCL API

```
cl_mem hDeviceMemA = clCreateBuffer(
  hContext,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  cnDimension * sizeof(cl_float),
 pA,  0);

cl_mem hDeviceMemB = /* ... */
```

OpenCL API

```
cl_mem hDeviceMemA = clCreateBuffer(
  hContext,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  cnDimension * sizeof(cl_float),
 pA,   0);

cl_mem hDeviceMemB = /* ... */
```

Create buffers for kernel input. Read only in the kernel. Written by the host.

OpenCL API

```
hDeviceMemC = clCreateBuffer(hContext,
  CL_MEM_WRITE_ONLY,
   cnDimension * sizeof(cl_float),
   0, 0);
```

Create buffer for kernel output.

OpenCL API

```
// setup parameter values
clSetKernelArg(hKernel, 0,
  sizeof(cl_mem), (void
  *)&hDeviceMemA);
clSetKernelArg(hKernel, 1,
  sizeof(cl_mem), (void
  *)&hDeviceMemB);
clSetKernelArg(hKernel, 2,
  sizeof(cl_mem), (void
  *)&hDeviceMemC);
```

Kernel arguments set by index

OpenCL API

```
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue,
  hKernel, 1, 0, &cnDimension, 0, 0, 0,
  0);
// copy results from device back to host
clEnqueueReadBuffer(hContext,
  hDeviceMemC, CL_TRUE, 0,
   cnDimension * sizeof(cl_float),
   pC, 0, 0, 0);
```

OpenCL API

Let OpenCL pick work group size

```
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue,
  hKernel, 1, 0, &cnDimension, 0, 0, 0,
  0);
// copy results from device back to host
clEnqueueReadBuffer(hContext,
  hDeviceMemC, CL_TRUE, 0,
   cnDimension * sizeof(cl_float),
   pC, 0, 0, 0);
```

Blocking read

clEnqueueNDRangeKernel

```
cl_int clEnqueueNDRangeKernel (
        cl_command_queue command_queue,
        cl_kernel kernel,
        cl_uint work_dim,                      <=3
        const size_t *global_work_offset,      NULL
        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)
```

global_work_size must be divisible by local_work_size

OpenCL API

```
delete [] pA;
delete [] pB;
delete [] pC;
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

CUDA Pointer Traversal

```
struct Node { Node* next; }
n = n->next; // undefined operation in OpenCL,
// since ‘n’ here is a kernel input
```

OpenCL Pointer Traversal

```
struct Node { unsigned int next; }
…
n = bufBase + n; // pointer arithmetic is fine, bufBase is
// a kernel input param to the buffer’s beginning
// no pointers between OpenCL buffers are allowed
```

Intro OpenCL Tutorial

Benedict R. Gaster, AMD
Architect, OpenCL™

The “Hello World” program in OpenCL

- Programs are passed to the OpenCL runtime via API calls expecting values of type char *
- Often, it is convenient to keep these programs in separate source files
  - For this tutorial, device programs are stored in files with names of the form name_kernels.cl
  - The corresponding device programs are loaded at runtime and passed to the OpenCL API

Header Files

```
#include <utility>
#define __NO_STD_VECTOR
// Use cl::vector instead of STL version
#include <CL/cl.hpp>

// additional C++ headers, which are agnostic to
// OpenCL.
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
#include <iterator>

const std::string hw("Hello World\n");
```

Error Handling

```
inline void checkErr(cl_int err, const char * name)
{
  if (err != CL_SUCCESS) {
      std::cerr << "ERROR: " << name
            << " (" << err << ")" << std::endl;
      exit(EXIT_FAILURE);
  }
}
```

OpenCL Contexts

```
int main(void)
{
  cl_int err;
  cl::vector< cl::Platform > platformList;
  cl::Platform::get(&platformList);
  checkErr(platformList.size()!=0 ? CL_SUCCESS
      : -1,"cl::Platform::get");
  std::cerr << "Platform number is: " <<
     platformList.size() << std::endl;

  std::string platformVendor;
  platformList[0].getInfo((cl_platform_info)CL_
  PLATFORM_VENDOR,&platformVendor);
  std::cerr << "Platform is by: " <<
     platformVendor << "\n";
```

OpenCL Contexts

```
cl_context_properties cprops[3] =
    {CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platformList[0])(),
    0};
cl::Context context(
    CL_DEVICE_TYPE_CPU,
    cprops,
    NULL,
    NULL,
    &err);
checkErr(err, "Context::Context()");
```

Just pick first platform

OpenCL Buffer

```
char * outH = new char[hw.length()+1];
cl::Buffer outCL(
  context,
  CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
  hw.length()+1,
  outH,
  &err);
checkErr(err, "Buffer::Buffer()");
```

OpenCL Devices

```
cl::vector<cl::Device> devices;
devices =
  context.getInfo<CL_CONTEXT_DEVICES>();
checkErr(devices.size() > 0 ? CL_SUCCESS : -1,
  "devices.size() > 0");
```

In OpenCL many operations are performed with respect to a given context. For example, buffer (1D regions of memory) and image (2D and 3D regions of memory) allocation are all context operations. But there are also device specific operations. For example, program compilation and kernel execution are on a per device basis, and for these a specific device handle is required.

Load Device Program

```
std::ifstream file("lesson1_kernels.cl");
checkErr(file.is_open() ? CL_SUCCESS:-1,
  "lesson1_kernel.cl");
std::string
  prog(std::istreambuf_iterator<char>(file),
  (std::istreambuf_iterator<char>()));
cl::Program::Sources source(1,
  std::make_pair(prog.c_str(),
  prog.length()+1));
cl::Program program(context, source);
err = program.build(devices,"");
checkErr(err, "Program::build()");
```

Kernel Objects

```
cl::Kernel kernel(program, "hello", &err);
checkErr(err, "Kernel::Kernel()");
err = kernel.setArg(0, outCL);
checkErr(err, "Kernel::setArg()");
```

Launching the Kernel

```
cl::CommandQueue queue(context, devices[0], 0,
  &err);
checkErr(err, "CommandQueue::CommandQueue()");
cl::Event event;
err = queue.enqueueNDRangeKernel(
  kernel,
  cl::NullRange,
  cl::NDRange(hw.length()+1),
  cl::NDRange(1, 1),
  NULL,
  &event);
checkErr(err,
  "ComamndQueue::enqueueNDRangeKernel()");
```

Reading the Results

```
event.wait();
err = queue.enqueueReadBuffer(
  outCL,
  CL_TRUE,
  0,
  hw.length()+1,
  outH);
checkErr(err,
  "ComamndQueue::enqueueReadBuffer()");
std::cout << outH;
return EXIT_SUCCESS;
}
```

The Kernel

```
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store
  : enable

__constant char hw[] = "Hello World\n";
__kernel void hello(__global char * out)
{
  size_t tid = get_global_id(0);
  out[tid] = hw[tid];
}
```

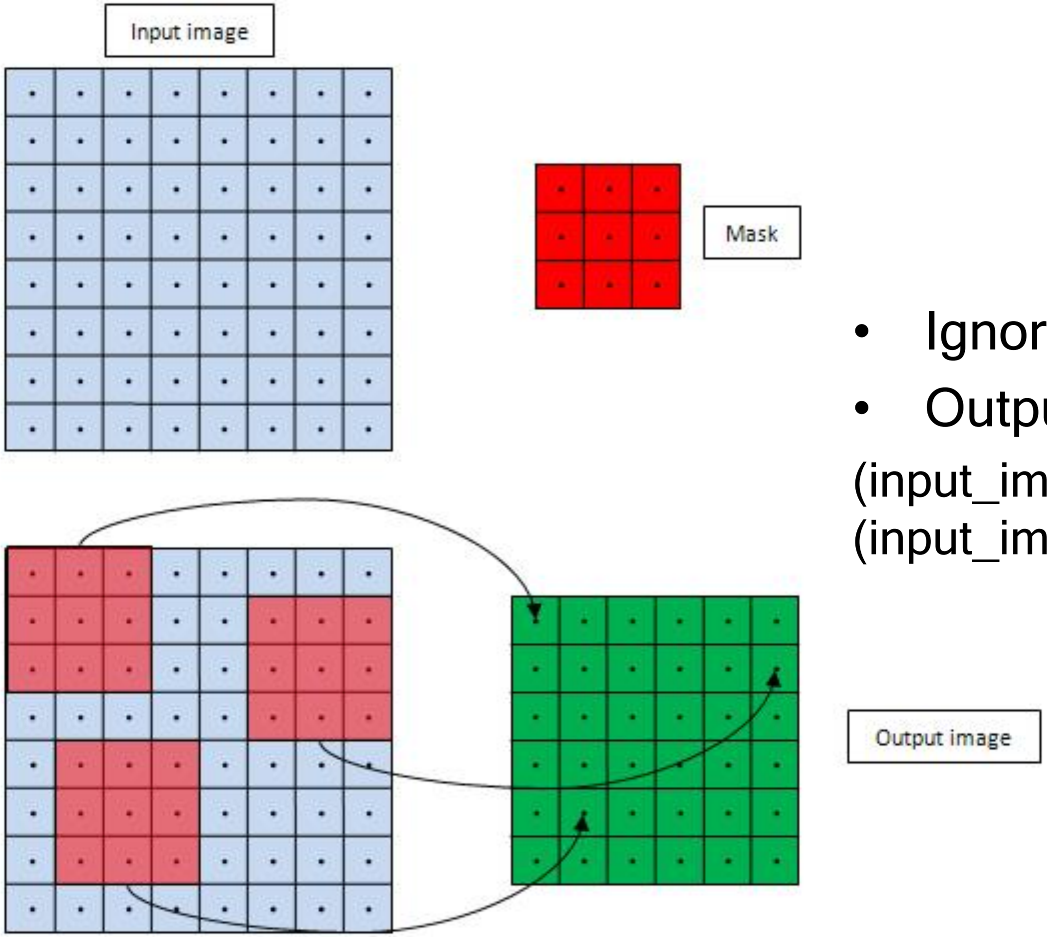
Image Convolution Using OpenCL™

Udeepta Bordoloi,
ATI Stream Application Engineer

10/13/2009

Note: ATI Stream Technology is now called AMD Accelerated Parallel Processing (APP) Technology.

Step 1 – The Algorithm



- Ignore boundaries
- Output size:

(input_image_width – filter_width + 1) by
(input_image_height – filter_width + 1)

C Version

```
void Convolve(float * pInput, float * pFilter, float
  * pOutput, const int nInWidth, const int nWidth,
  const int nHeight,
const int nFilterWidth, const int nNumThreads)
{
  for (int yOut = 0; yOut < nHeight; yOut++)
  {
      const int yInTopLeft = yOut;
      for (int xOut = 0; xOut < nWidth; xOut++)
      {
            const int xInTopLeft = xOut;
            float sum = 0;
```

C Version (2)

```
for (int r = 0; r < nFilterWidth; r++)
{
      const int idxFtmp = r * nFilterWidth;
      const int yIn = yInTopLeft + r;
      const int idxIntmp = yIn * nInWidth +
                  xInTopLeft;
      for (int c = 0; c < nFilterWidth; c++)
      {
            const int idxF = idxFtmp + c;
            const int idxIn = idxIntmp + c;
            sum += pFilter[idxF]*pInput[idxIn];
      }
} //for (int r = 0…
```

C Version (3)

```
            const int idxOut = yOut * nWidth + xOut;
            pOutput[idxOut] = sum;
      } //for (int xOut = 0…
  } //for (int yOut = 0…
}
```

Parameters

```
struct paramStruct
{
  int nWidth; //Output image width
  int nHeight; //Output image height
  int nInWidth; //Input image width
  int nInHeight; //Input image height
  int nFilterWidth; //Filter size is nFilterWidth X
                    //nFilterWidth
  int nIterations; //Run timing loop for nIterations
  //Test CPU performance with 1,4,8 etc. OpenMP threads
  std::vector ompThreads;
  int nOmpRuns; //ompThreads.size()
  bool bCPUTiming; //Time CPU performance
} params;
```

OpenMP for Comparison

```
//This #pragma splits the work between multiple threads
#pragma omp parallel for num_threads(nNumThreads)
for (int yOut = 0; yOut < nHeight; yOut++)
...

void InitParams(int argc, char* argv[])
{
…
// time the OpenMP convolution performance with
// different numbers of threads
   params.ompThreads.push_back(4);
   params.ompThreads.push_back(1);
   params.ompThreads.push_back(8);
   params.nOmpRuns = params.ompThreads.size();
}
```

First Kernel

```
__kernel void Convolve(const __global float * pInput,
  __constant float * pFilter, __global float * pOutput,
  const int nInWidth, const int nFilterWidth)
{
  const int nWidth = get_global_size(0);
  const int xOut = get_global_id(0);
  const int yOut = get_global_id(1);
  const int xInTopLeft = xOut;
  const int yInTopLeft = yOut;

  float sum = 0;
```

First Kernel (2)

```
  for (int r = 0; r < nFilterWidth; r++)
  {
      const int idxFtmp = r * nFilterWidth;
      const int yIn = yInTopLeft + r;
      const int idxIntmp = yIn * nInWidth + xInTopLeft;

      for (int c = 0; c < nFilterWidth; c++)
      {
            const int idxF = idxFtmp + c;
            const int idxIn = idxIntmp + c;
            sum += pFilter[idxF]*pInput[idxIn];
      }
  } //for (int r = 0…
  const int idxOut = yOut * nWidth + xOut;
  Output[idxOut] = sum;
}
```

Initialize OpenCL

```
cl_context context =
  clCreateContextFromType(…,CL_DEVICE_TYPE_CPU,…);

// get list of devices - quad core counts as one device
size_t listSize;
/* First, get the size of device list */
clGetContextInfo(context, CL_CONTEXT_DEVICES, …,
  &listSize);
/* Now, allocate the device list */
cl_device_id devices = (cl_device_id *)malloc(listSize);
/* Next, get the device list data */
clGetContextInfo(context, CL_CONTEXT_DEVICES, listSize,
  devices, …);
```

Initialize OpenCL (2)

```
cl_command_queue queue = clCreateCommandQueue(context,
  devices[0], …);

cl_program program = clCreateProgramWithSource(context,
  1, &source, …);

clBuildProgram(program, 1, devices, …);

cl_kernel kernel = clCreateKernel(program, "Convolve",
  …);

// get error messages
clGetProgramBuildInfo(program, devices[0],
  CL_PROGRAM_BUILD_LOG, …);
```

Initialize Buffers

```
cl_mem inputCL = clCreateBuffer(context,
  CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
  host_buffer_size, host_buffer_ptr, …);

//If the device is a GPU (CL_DEVICE_TYPE_GPU), we can
// explicitly copy data to the input image buffer on the
// device:
clEnqueueWriteBuffer(queue, inputCL, …, host_buffer_ptr,
      …);

// And copy back from the output image buffer after the
// convolution kernel execution.
clEnqueueReadBuffer(queue, outputCL, …, host_buffer_ptr,
      …);
```

Execute Kernel

```
/* input buffer, arg 0 */
clSetKernelArg(kernel, 0, sizeof(cl_mem),
      (void *)&inputCL);
/* filter buffer, arg 1 */
clSetKernelArg(kernel, 1, sizeof(cl_mem),
      (void *)&filterCL);
/* output buffer, arg 2 */
clSetKernelArg(kernel, 2, sizeof(cl_mem),
      (void *)&outputCL);
/* input image width, arg 3*/
clSetKernelArg(kernel, 3, sizeof(int),
      (void *)&nInWidth);
/* filter width, arg 4*/
clSetKernelArg(kernel, 4, sizeof(int),
      (void *)&nFilterWidth);
```

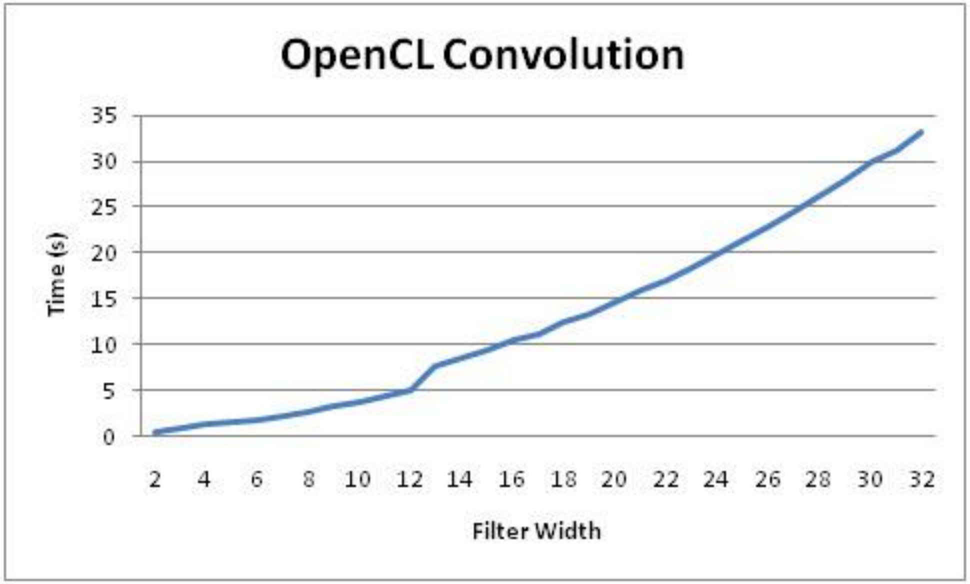
Execute Kernel

```
clEnqueueNDRangeKernel(queue, kernel,
      data_dimensionality, …, total_work_size,
      work_group_size, …);

// release all buffers
clReleaseBuffer(inputCL);
...

// release all resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

Timing

```
clFinish(queue); //Timer Started here();
for (int i = 0; i < nIterations; i++)
      clEnqueueNDRangeKernel(…);
clFinish(queue); //Timer Stopped here();
//Average Time = ElapsedTime()/nIterations;
```

clFinish() call before both starting and stopping the timer ensures that we time the kernel execution activity to its completion and nothing else



On 4-core AMD Phenom treated as a single device by OpenCL

C++ Bindings

```
~~cl_context context =~~
~~        clCreateContextFromType(…,CL_DEVICE_TYPE_CPU,…);~~
cl::Context context = cl::Context(CL_DEVICE_TYPE_CPU);

~~// get list of devices - quad core counts as one device~~
~~size_t listSize;~~
~~/* First, get the size of device list */~~
~~clGetContextInfo(context, CL_CONTEXT_DEVICES, …, &listSize);~~
~~/* Now, allocate the device list */~~
~~cl_device_id devices = (cl_device_id *)malloc(listSize);~~
~~/* Next, get the device list data */~~
~~clGetContextInfo(context, CL_CONTEXT_DEVICES, listSize,~~
        ~~devices, …);~~
std::vector<cl::Device> devices = context.getInfo();
```

See https://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf

C++ Bindings (2)

```
cl::CommandQueue queue = cl::CommandQueue(context, devices[0]);

cl::Program program = cl::Program(context, …);

program.build(devices);

cl::Kernel kernel = cl::Kernel(program, "Convolve");

string str = program.getBuildInfo(devices[0]);

// Buffer init is similar to C version
// using methods of queue
```

Execute Kernel

```
 /* input buffer, arg 0 */
~~clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&inputCL);~~
kernel.setArg(0, inputCL);

/* filter buffer, arg 1 */
~~clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&filterCL);~~
kernel.setArg(1, filterCL);

// etc.


queue.clEnqueueNDRangeKernel(kernel, …, total_work_size,
       work_group_size, …);
```

Loop Unrolling

```
__kernel void Convolve_Unroll(const __global float * pInput,
      __constant float * pFilter, __global float * pOutput,
      const int nInWidth, const int nFilterWidth)
{
      const int nWidth = get_global_size(0);
      const int xOut = get_global_id(0);
      const int yOut = get_global_id(1);
      const int xInTopLeft = xOut;
      const int yInTopLeft = yOut;

      float sum = 0;
      for (int r = 0; r < nFilterWidth; r++)
      {
            const int idxFtmp = r * nFilterWidth;
            const int yIn = yInTopLeft + r;
            const int idxIntmp = yIn * nInWidth + xInTopLeft;
```

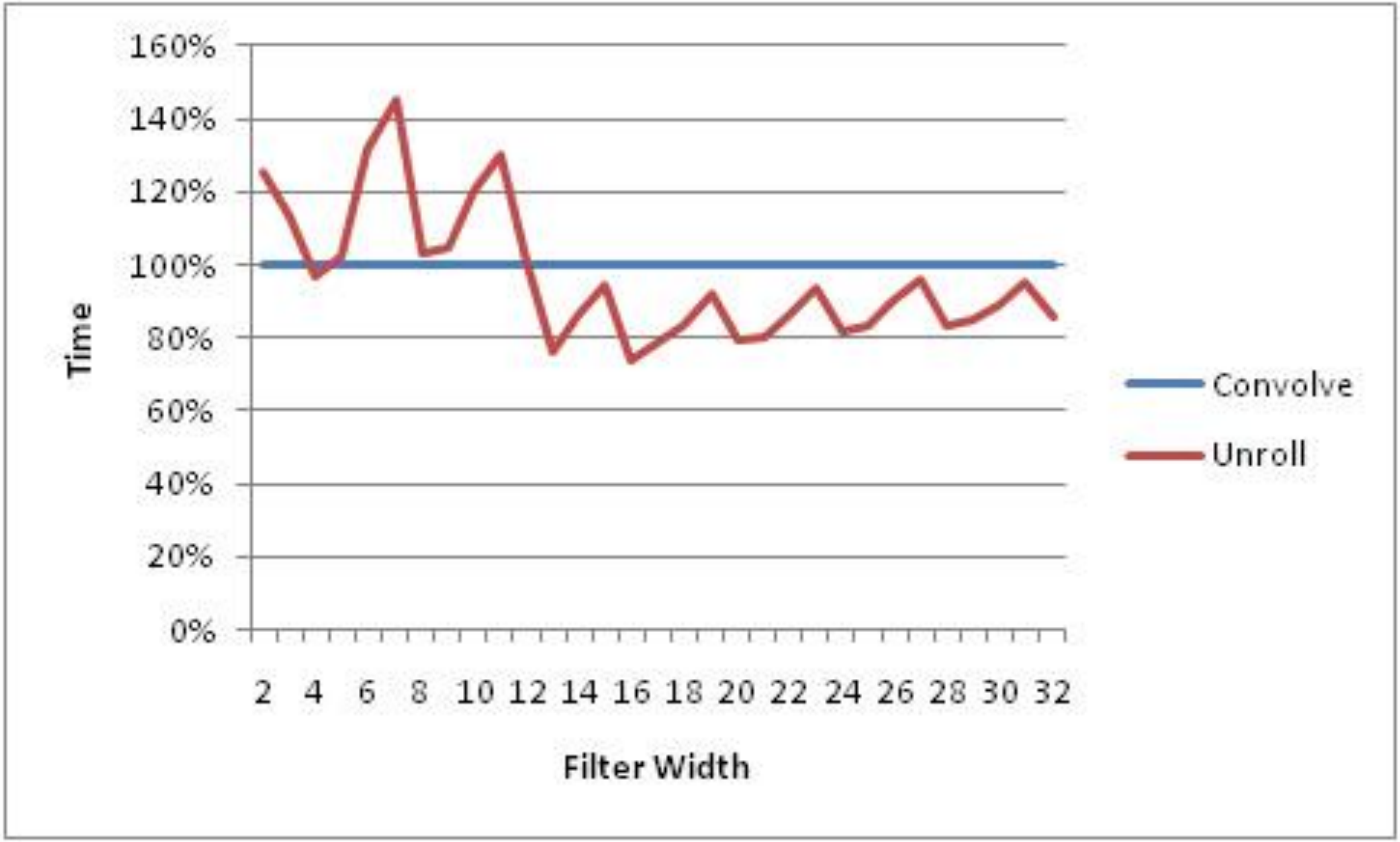
Loop Unrolling (2)

```
int c = 0;
while (c <= nFilterWidth-4)
{
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        sum += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        sum += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        sum += pFilter[idxF]*pInput[idxIn];
        c += 4;
}
```

Loop Unrolling (3)

```
             for (int c1 = c; c1 < nFilterWidth; c1++)
             {
                    const int idxF = idxFtmp + c1;
                    const int idxIn = idxIntmp + c1;
                    sum += pFilter[idxF]*pInput[idxIn];
             }
       } //for (int r = 0…
       const int idxOut = yOut * nWidth + xOut;
       pOutput[idxOut] = sum;
}

// what does this do?
```

Performance



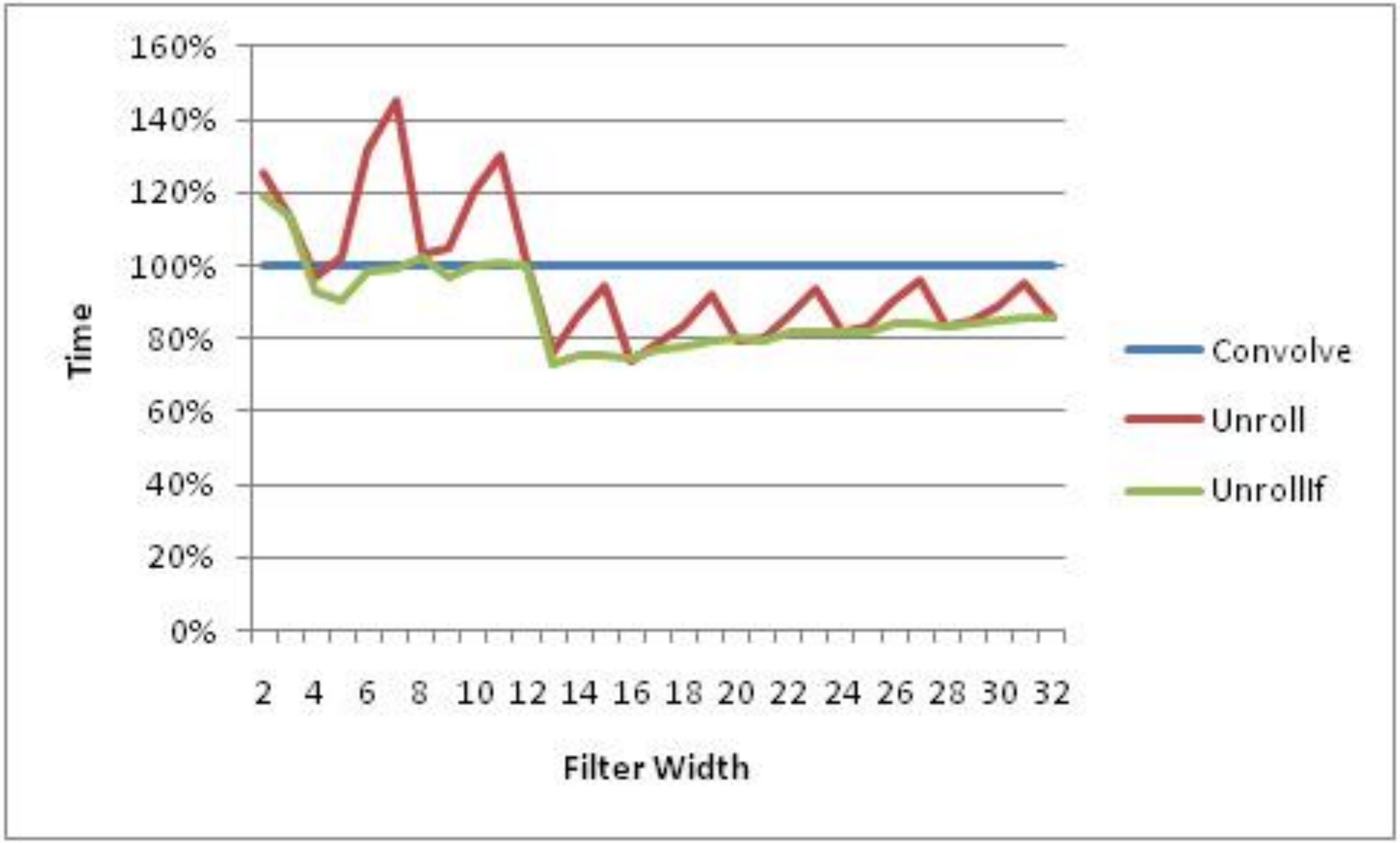
Unrolled Kernel 2 (if Kernel)

```
// last loop
int cMod = nFilterWidth - c;
if (cMod == 1)
{
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
}
else if (cMod == 2)
{
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
        sum += pFilter[idxF+1]*pInput[idxIn+1];
}
```

Unrolled Kernel 2 (2)

```
              else if (cMod == 3)
              {
                    int idxF = idxFtmp + c;
                    int idxIn = idxIntmp + c;
                    sum += pFilter[idxF]*pInput[idxIn];
                    sum += pFilter[idxF+1]*pInput[idxIn+1];
                    sum += pFilter[idxF+2]*pInput[idxIn+2];
              }
       } //for (int r = 0…
       const int idxOut = yOut * nWidth + xOut;
       pOutput[idxOut] = sum;
}
```

Performance



Yet another way to achieve similar results is to write four different versions of the ConvolveUnroll kernel.

The four versions will correspond to (filterWidth%4) equalling 0, 1, 2, or 3.

The particular version called can be decided at run-time depending on the value of filterWidth

Kernel with Invariants

- Loop unrolling did not help when the filter width is low

- So far, kernels have been written in a generic way so that they will work for all filter sizes
- What if we can focus on a particular filter size?
  - E.g. 5×5. We can now unroll the inner loop five times and get rid of the loop condition
  - If we use the invariant in the loop condition, a good compiler will unroll the loop itself
  - `FILTER_WIDTH` can be passed to compiler

Kernel with Invariants

```
__kernel void Convolve_Def(const __global float * pInput,
       __constant float * pFilter, __global float * pOutput,
       const int nInWidth, const int nFilterWidth)
{
       const int nWidth = get_global_size(0);
       const int xOut = get_global_id(0);
       const int yOut = get_global_id(1);
       const int xInTopLeft = xOut;
       const int yInTopLeft = yOut;

       float sum = 0;
       for (int r = 0; r < FILTER_WIDTH; r++)
       {
              const int idxFtmp = r * FILTER_WIDTH;
              const int yIn = yInTopLeft + r;
              const int idxIntmp = yIn * nInWidth + xInTopLeft;
```

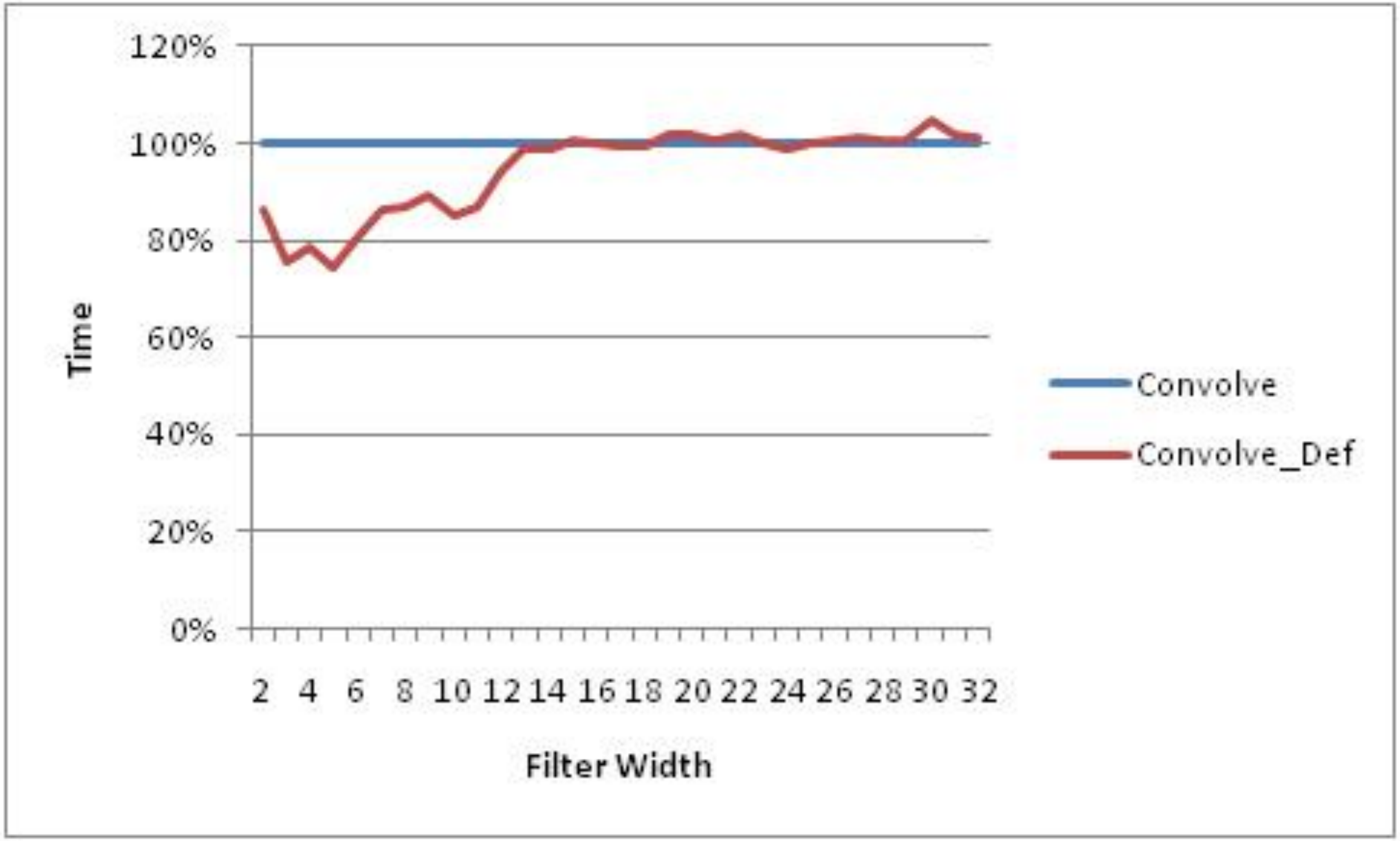
Kernel with Invariants (2)

```
        for (int c = 0; c < FILTER_WIDTH; c++)
        {
            const int idxF = idxFtmp + c;
            const int idxIn = idxIntmp + c;
            sum += pFilter[idxF]*pInput[idxIn];
        }
    } //for (int r = 0…
    const int idxOut = yOut * nWidth + xOut;
    pOutput[idxOut] = sum;
}
```

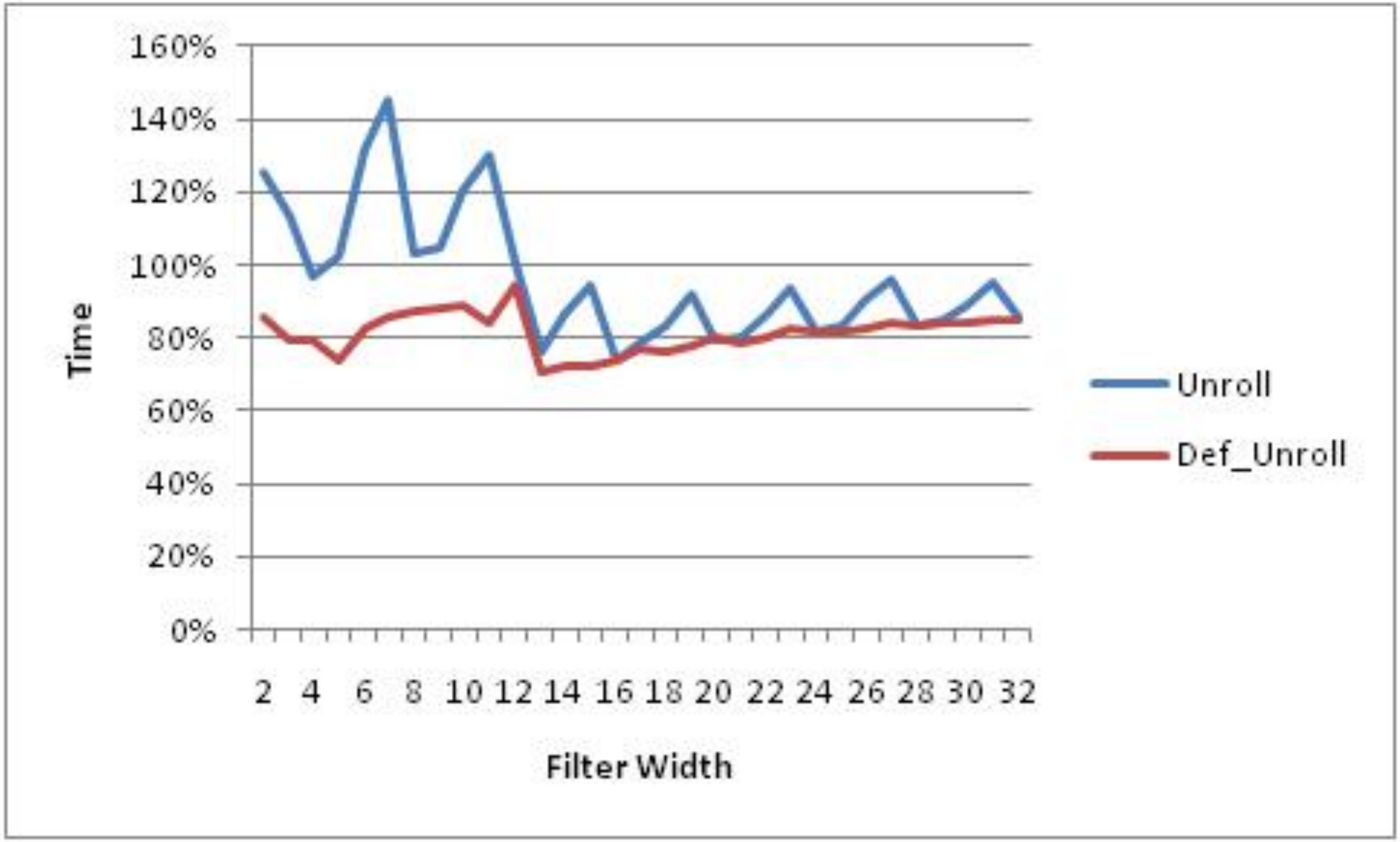
Setting Filter Width

```
// this can be done online and offline

/* create a cl source string */
std::string sourceStr = Convert_File_To_String(File_Name);
cl::Program::Sources sources(1,
       std::make_pair(sourceStr.c_str(), sourceStr.length()));
/* create a cl program object */
program = cl::Program(context, sources);
/* build a cl program executable with some #defines */
char options[128];
sprintf(options, "-DFILTER_WIDTH=%d", filter_width);
program.build(devices, options);

/* create a kernel object for a kernel with the given name */
cl::Kernel kernel = cl::Kernel(program, "Convolve_Def");
```

Performance

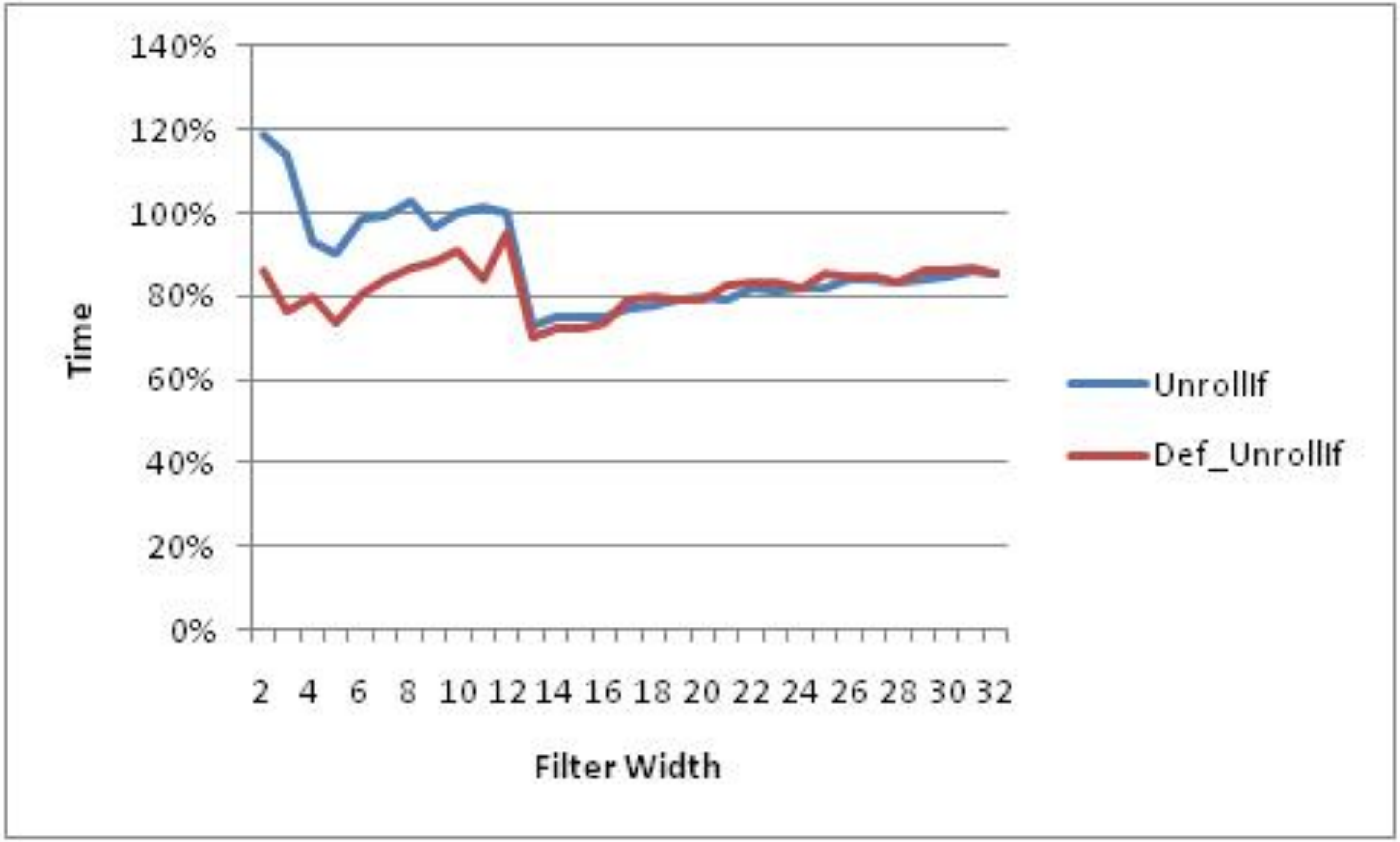


Performance



Performance

Unroll + if on remainder



Vectorization

```
__kernel void Convolve_Unroll(const __global float * pInput,
        __constant float * pFilter, __global float * pOutput,
        const int nInWidth, const int nFilterWidth)
{
        const int nWidth = get_global_size(0);
        const int xOut = get_global_id(0);
        const int yOut = get_global_id(1);
        const int xInTopLeft = xOut;
        const int yInTopLeft = yOut;

        float sum0 = 0; float sum1 = 0;
        float sum2 = 0; float sum3 = 0;

        for (int r = 0; r < nFilterWidth; r++)
        {
                const int idxFtmp = r * nFilterWidth;
```

Vectorization (2)

```
const int yIn = yInTopLeft + r;
const int idxIntmp = yIn * nInWidth + xInTopLeft;

int c = 0;
while (c <= nFilterWidth-4)
{
        float mul0, mul1, mul2, mul3;
        int idxF = idxFtmp + c;
        int idxIn = idxIntmp + c;
        mul0 = pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        mul1 += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        mul2 += pFilter[idxF]*pInput[idxIn];
        idxF++; idxIn++;
        mul3 += pFilter[idxF]*pInput[idxIn];
```

Vectorization (3)

```
            sum0 += mul0; sum1 += mul1;
            sum2 += mul2; sum3 += mul3;
            c += 4;
        }

        for (int c1 = c; c1 < nFilterWidth; c1++)
        {
            const int idxF = idxFtmp + c1;
            const int idxIn = idxIntmp + c1;
            sum0 += pFilter[idxF]*pInput[idxIn];
        }
    } //for (int r = 0…
    const int idxOut = yOut * nWidth + xOut;
    pOutput[idxOut] = sum0 + sum1 + sum2 + sum3;
}
```

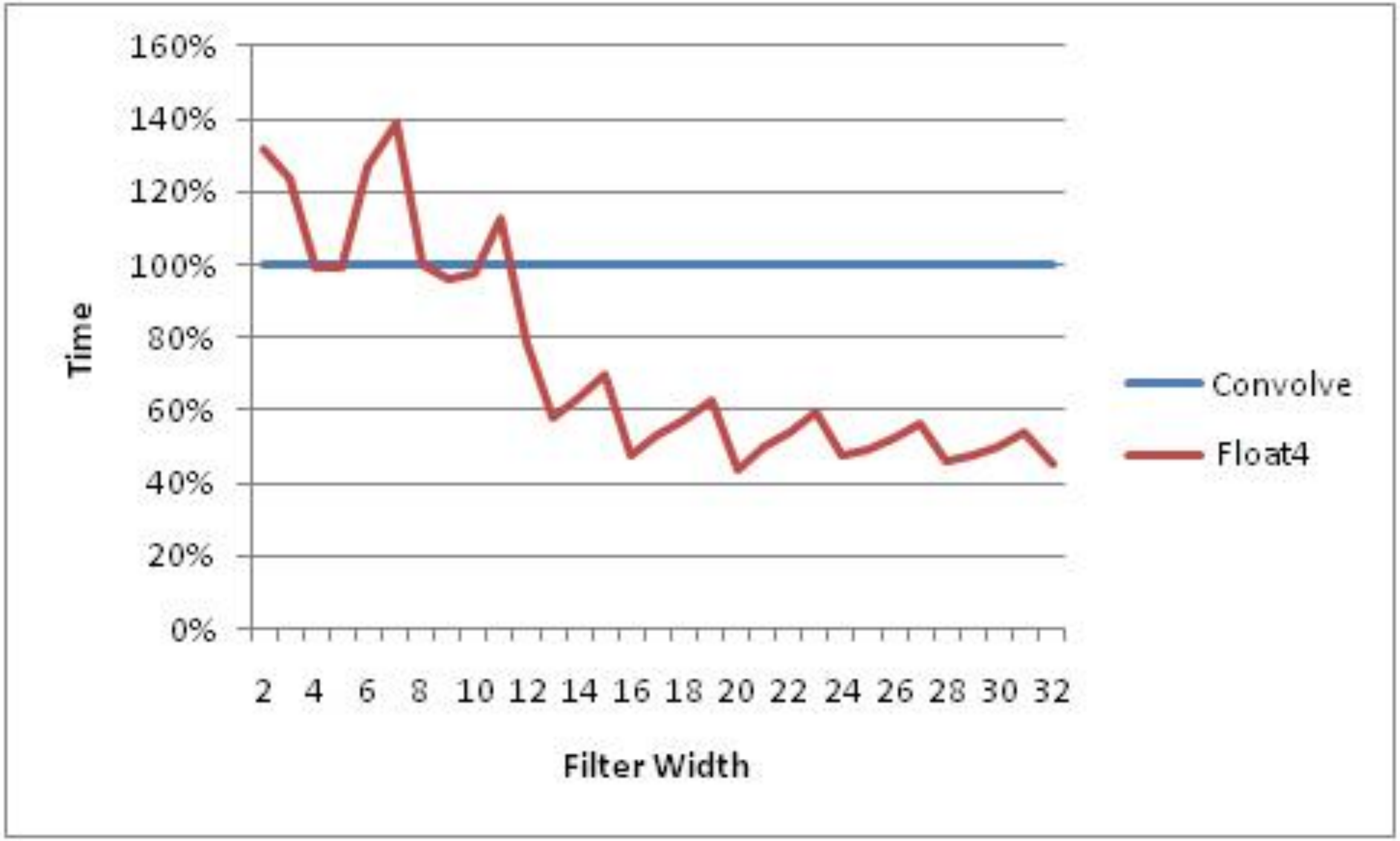
Vectorized Kernel

```
__kernel void Convolve_Float4(const __global float * pInput,
       __constant float * pFilter, __global float * pOutput,
       const int nInWidth, const int nFilterWidth)
{
       const int nWidth = get_global_size(0);
       const int xOut = get_global_id(0);
       const int yOut = get_global_id(1);
       const int xInTopLeft = xOut;
       const int yInTopLeft = yOut;

       float4 sum4 = 0;
       for (int r = 0; r < nFilterWidth; r++)
       {
              const int idxFtmp = r * nFilterWidth;
              const int yIn = yInTopLeft + r;
              const int idxIntmp = yIn * nInWidth + xInTopLeft;
```

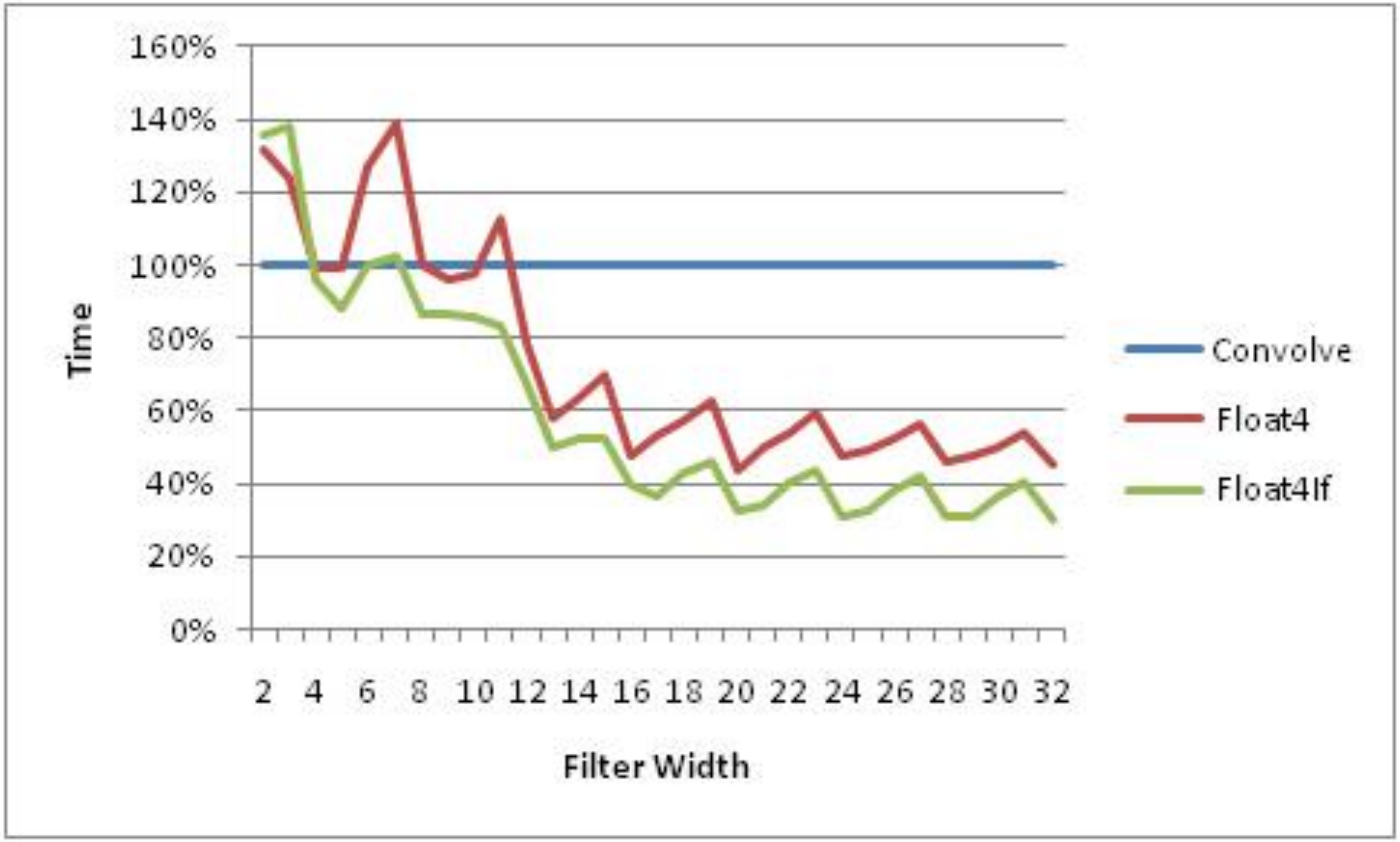
Vectorized Kernel

```
            int c = 0; int c4 = 0;
            while (c <= nFilterWidth-4)
            {
                  float4 filter4 = vload4(c4,pFilter+idxFtmp);
                  float4 in4 = vload4(c4,pInput +idxIntmp);
                  sum4 += in4 * filter4;
                  c += 4;
                  c4++;
}
for (int c1 = c; c1 < nFilterWidth; c1++) { const int idxF =
idxFtmp + c1; const int idxIn = idxIntmp + c1; sum4.x +=
pFilter[idxF]*pInput[idxIn]; } } //for (int r = 0…

const int idxOut = yOut * nWidth + xOut;
pOutput[idxOut] = sum4.x + sum4.y + sum4.z + sum4.w; }
```

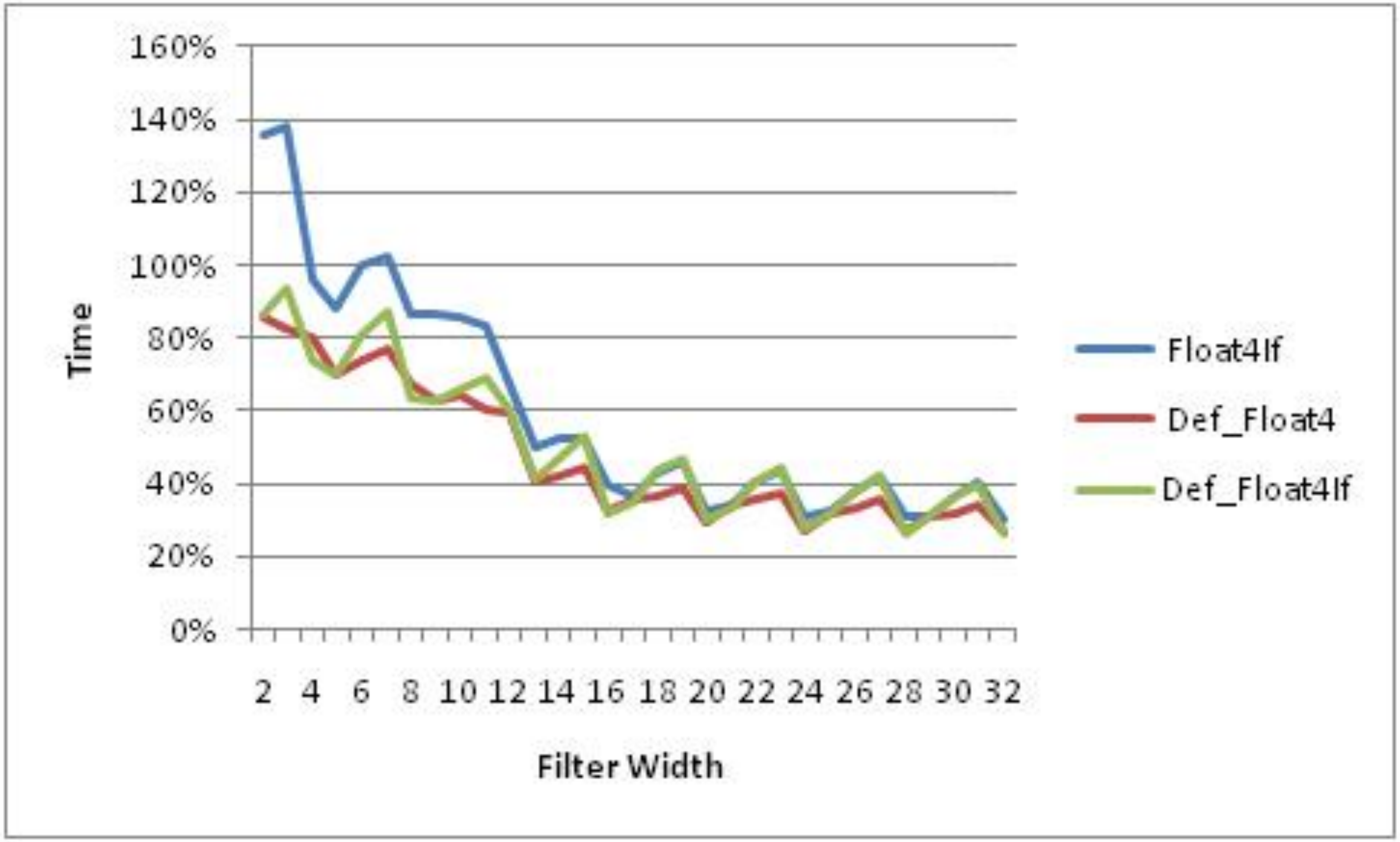
Performance



Performance – if Kernel

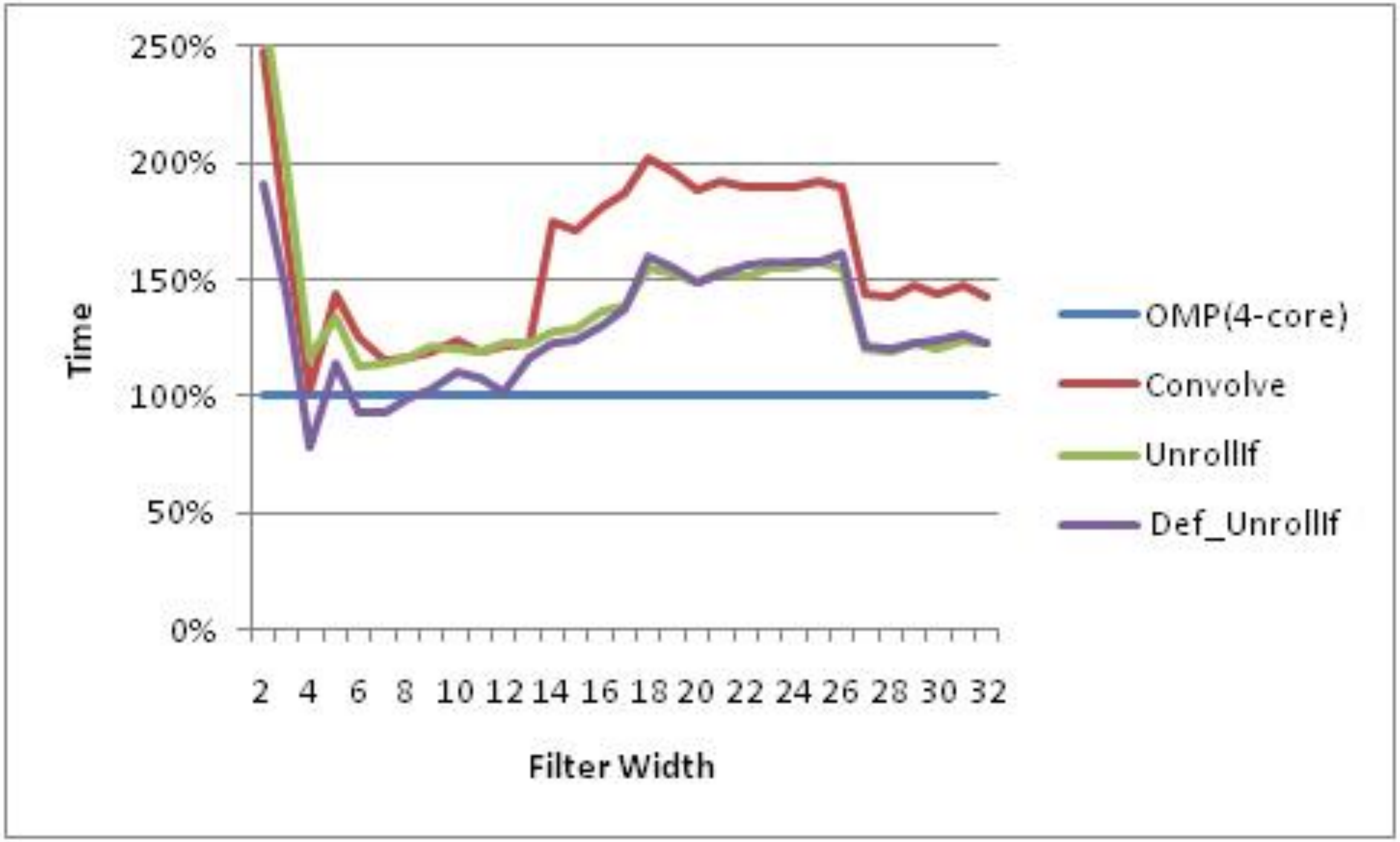


Performance – Kernel with Invariants

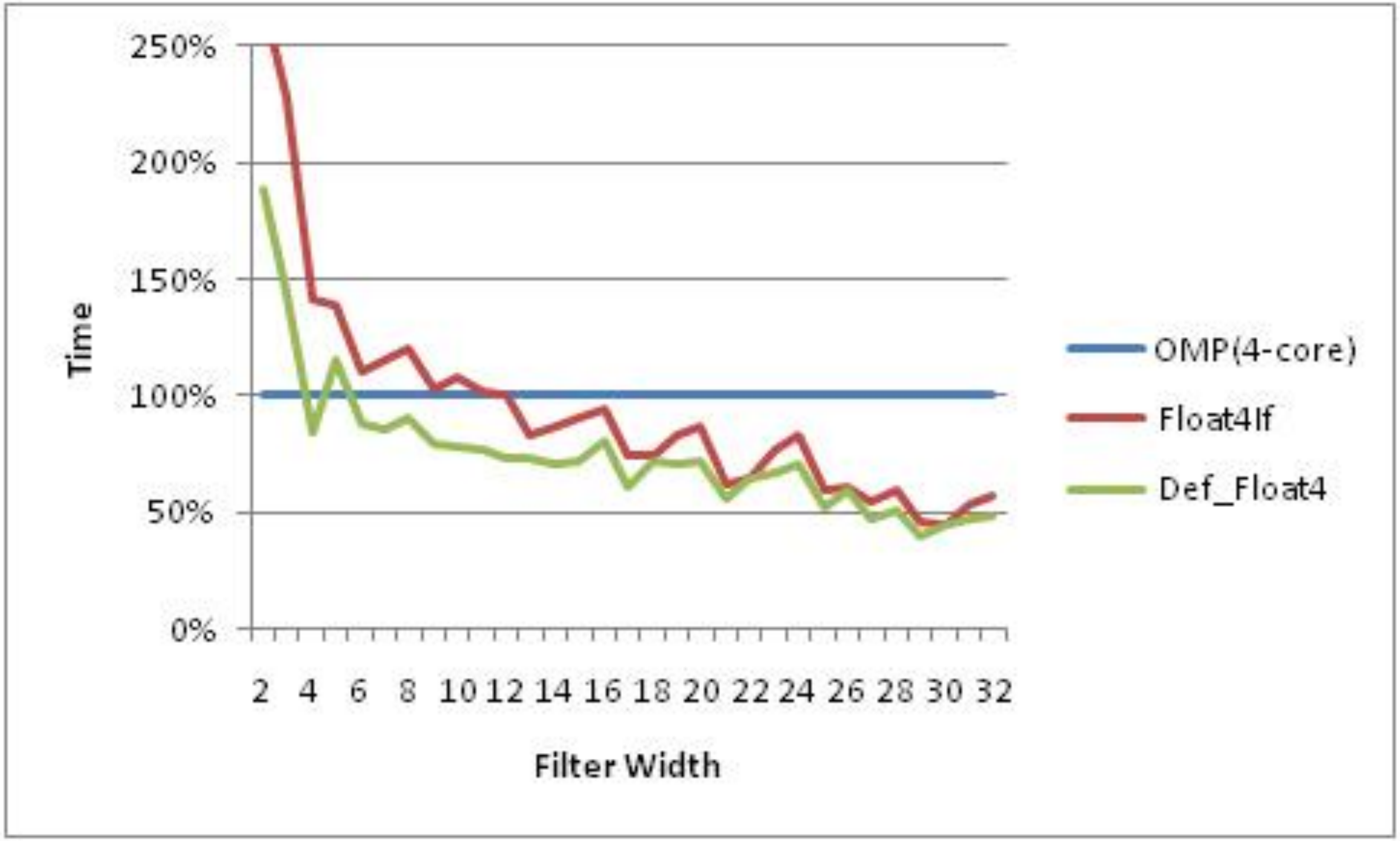


Instead of passing filterWidth as an argument to the kernel, we will define the value for FILTER_WIDTH when we build the OpenCL program object

OpenMP Comparison



OpenMP Comparison


Time
Filter Width
OMP(4-core)
Float4If
Def_Float4