Introduction to GPU Computing with OpenCL

Presentation Outline

- Overview of OpenCL for NVIDIA GPUs
- Highlights from OpenCL Spec, API and Language
- Sample code walkthrough (oclVectorAdd)
- What Next ?
- OpenCL Information and Resources

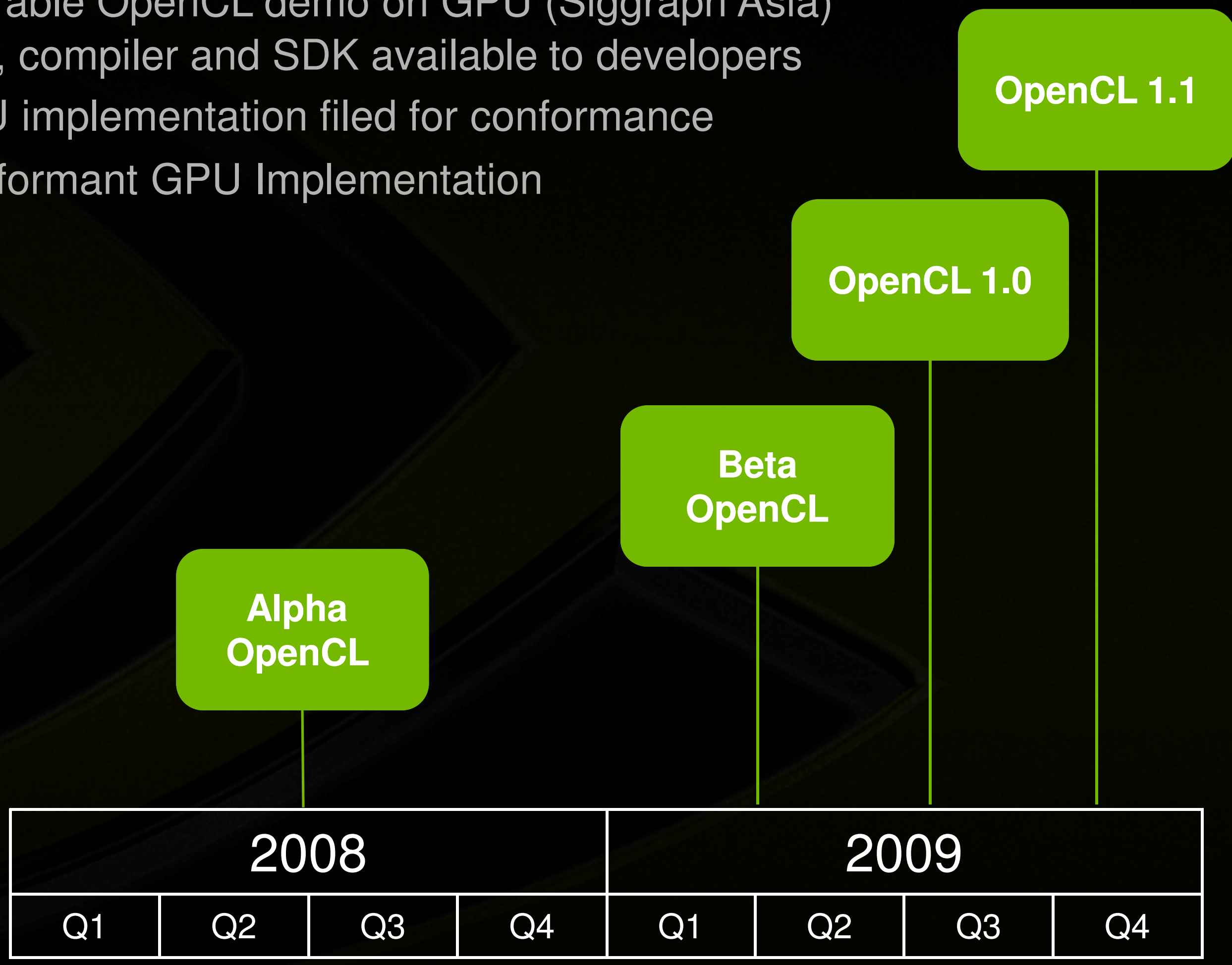
OpenCL™ – Open Computing Language

- Open, royalty-free standard C-language extension
- For parallel programming of heterogeneous systems using GPUs, CPUs, CBE, DSP's and other processors including embedded mobile devices
- Initially proposed by Apple, who put OpenCL in OSX Snow Leopard and is active in the working group. Working group includes NVIDIA, Intel, AMD, IBM…
- Managed by Khronos Group
  (same group that manages the OpenGL std)

Note: The OpenCL working group chair is NVIDIA VP Neil Trevett, who is also President of Khronos Group

NVIDIA's OpenCL Timeline

12 / 2008	1st operable OpenCL demo on GPU (Siggraph Asia)
4 / 2009	Drivers, compiler and SDK available to developers
5 / 2009	1st GPU implementation filed for conformance
6 / 2009	1st Conformant GPU Implementation

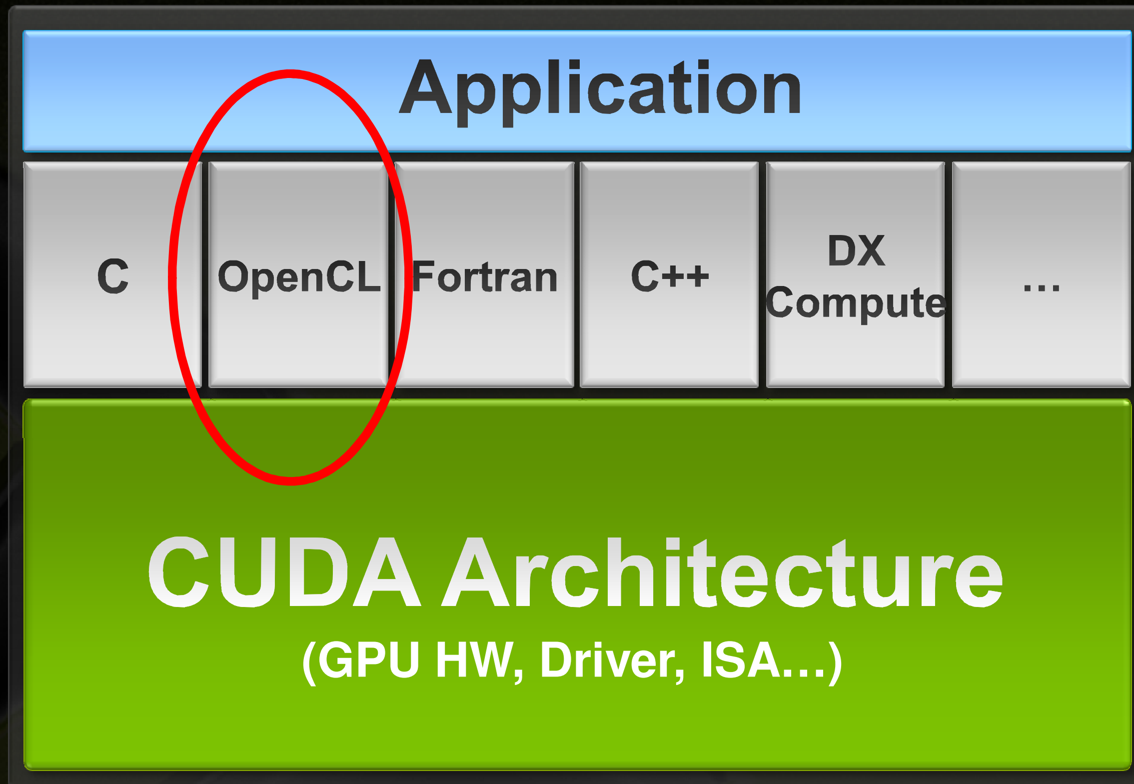


NVIDIA OpenCL Support

- **OS / Platform**
  - 32 and 64 bit Windows XP and Vista (and soon Win 7)
  - 32 and 64 bit Linux (Ubuntu, RHEL, etc)
  - Mac OSX Snow Leopard (indirectly via Apple)
- **IDE's**
  - VS 8(2005) and VS 9(2008) for Windows
  - GCC for Linux
- **Drivers and JIT compiler**
  - In SDK for Alpha & Beta
  - To be packaged with GPU drivers
- **SDK**
  - Source code & white papers for Sample applications (30 presently)
  - Documentation: Spec, Getting Started Guide, Programming Manual, Best Practices (Optimization) Guide

OpenCL & CUDA GPU's

- OpenCL
- C for CUDA
- DirectX Compute
- Fortran (PGI)
- C++
- Java
- Python
- .Net



The CUDA Architecture supports standard languages & APIs to tap the massive parallel computational power of the GPU

OpenCL Language & API Highlights

- **Platform Layer API (called from host)**
  - Abstraction layer for diverse computational resources
  - Query, select and initialize *compute devices*
  - Create *compute contexts* and *work-queues*

- **Runtime API (called from host)**
  - Launch *compute kernels*
  - Set kernel execution configuration
  - Manage scheduling, compute, and memory resources

- **OpenCL Language**
  - Write *compute kernels* that run on a *compute device*
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions
  - Includes rich set of built-in functions, in addition to standard C operators
  - Can be compiled JIT/Online or offline

Kernel Execution Configuration

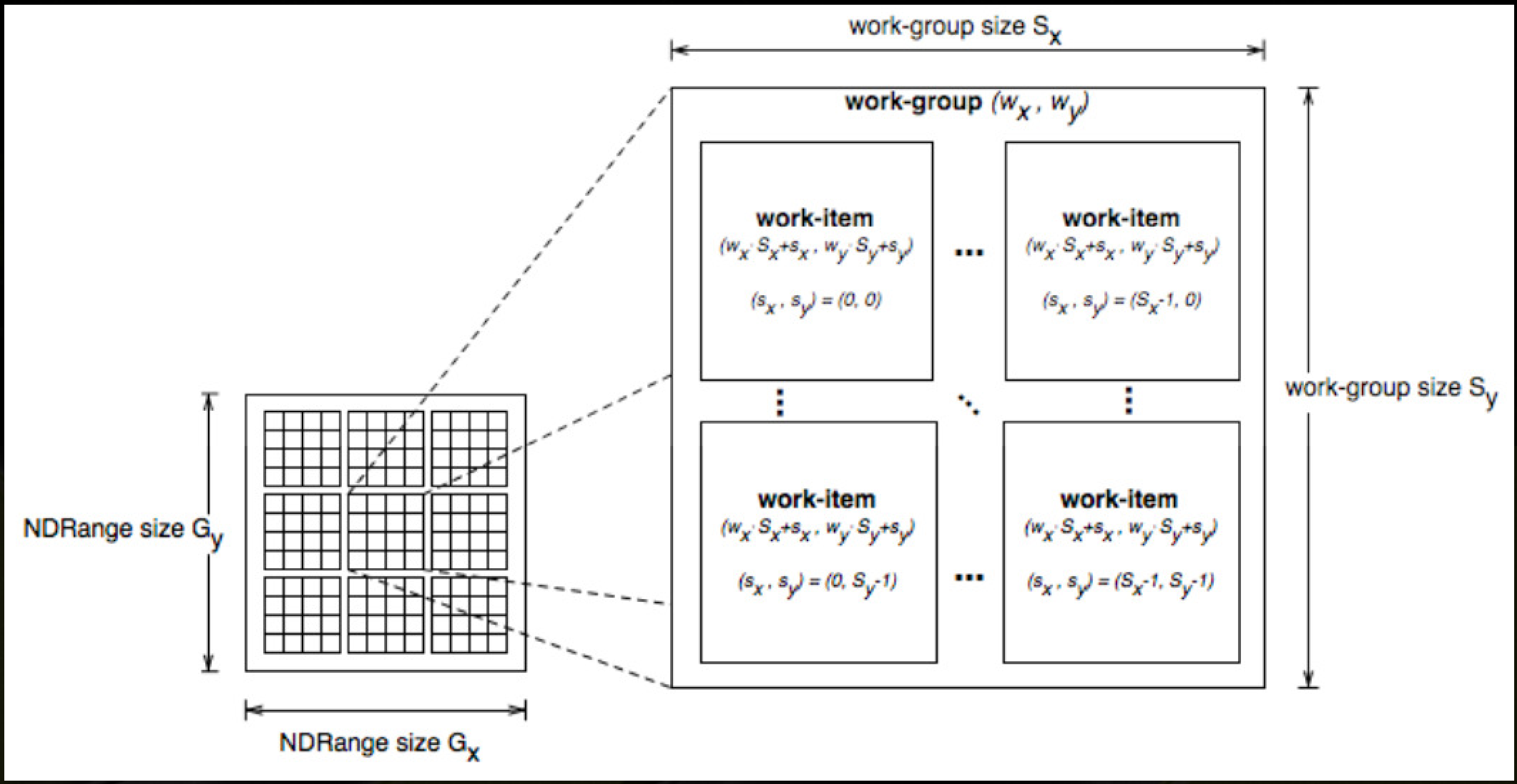
- **Host program launches kernel in index space called NDRange**
  - NDRange (“N-Dimensional Range”) is a multitude of kernel instances arranged into 1, 2 or 3 dimensions
  - A single kernel instance in the index space is called a ***Work Item***
    - Each Work Item executes same compute kernel (on different data)
    - Work Items have unique global IDs from the index space

- **Work-items are further grouped into *Work Groups***
  - Work Groups have a unique Work Group ID
  - Work Items have a unique Local ID within a Work Group

~ Analagous to a C loop that calls a function many times

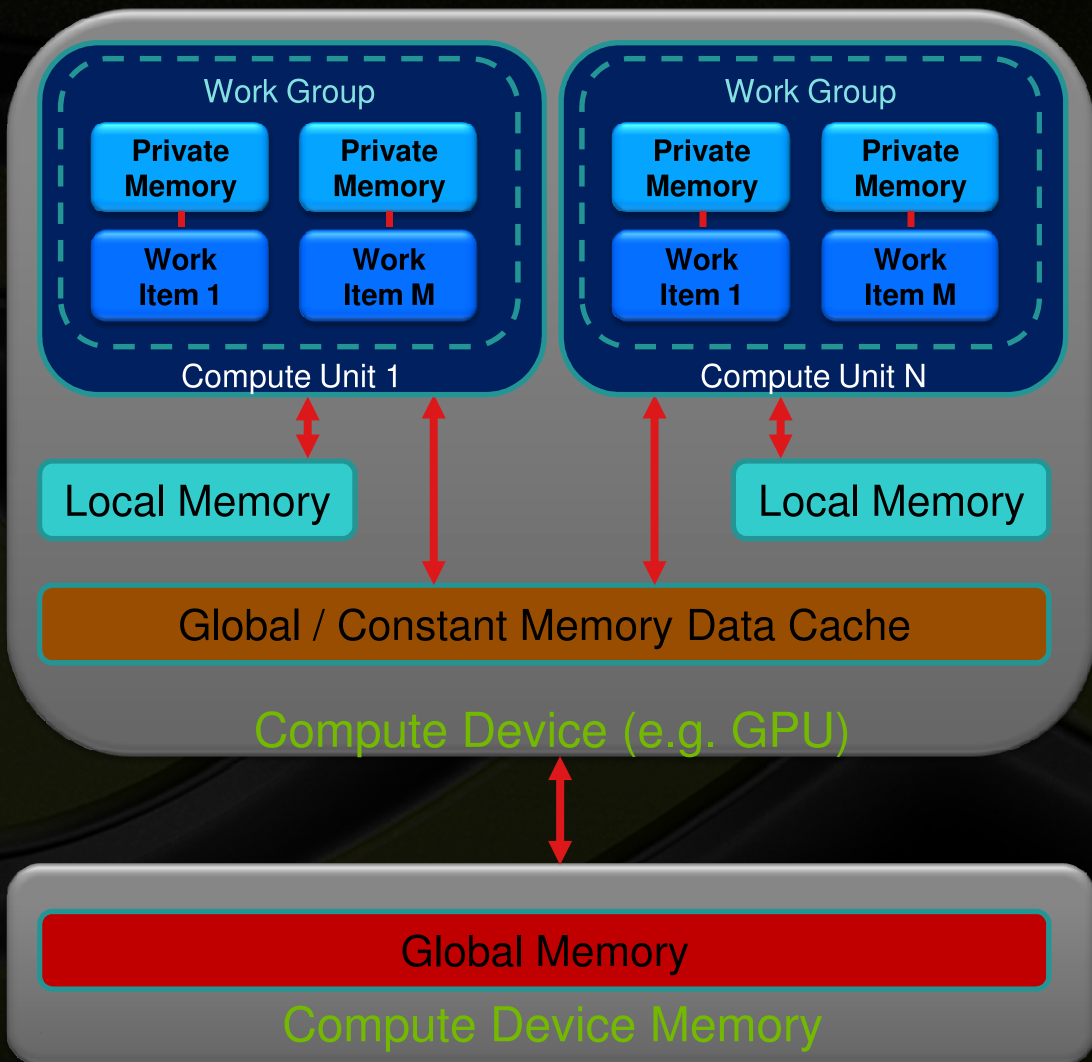
- Except all iterations are called simultaneously & executed in parallel

Kernel Execution Configuration



- Total number of Work Items = G_x * G_y
- Size of each Work Group = S_x * S_y
- Global ID can be computed from Work Group ID and Local ID

OpenCL Memory Model



Private Memory
Read / Write access
For *Work Item* only

Local Memory
Read / Write access
For entire *Work Group*

Constant Memory
Read access
For entire *ND-range*
(All work items, all work groups)

Global Memory
Read / write access
For entire ND-range
(All work items, all work groups)

Basic Program Structure

- Host program
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

  Platform Layer

  - Query compute devices
  - Create contexts

  Runtime

- Compute Kernel (runs on device)
  - C code with some restrictions and extensions

  OpenCL Language

OpenCL Memory Objects

- Buffer objects
  - 1D collection of objects (like C arrays)
  - Scalar & Vector types, and user-defined Structures
  - Buffer objects are accessed via pointers in the compute kernel
- Image objects
  - 2D or 3D texture, frame-buffer, or images
  - Must be addressed through built-in functions
- Sampler objects
  - Describe how to sample an image in the kernel
    - Addressing modes
    - Filtering modes

OpenCL Language Highlights

- Function qualifiers
  - "__kernel" qualifier declares a function as a kernel
- Address space qualifiers
  - __global, __local, __constant, __private
- Work-item functions
  - get_work_dim()
  - get_global_id(), get_local_id(), get_group_id(), get_local_size()
- Image functions
  - Images must be accessed through built-in functions
  - Reads/writes performed through sampler objects from host or defined in source
- Synchronization functions
  - Barriers - All Work Items within a Work Group must execute the barrier function before any Work Item in the Work Group can continue

oclVectorAdd code walkthrough

- **Element-by-element addition of two floating point vectors**

  c[i] = a[i] + b[i] *(where i ranges from 0 to a large #, e.g. **11444777**)*

- **Equivalent C loop**

```c
int iNumElements = 11444777;
// a, b and c are allocated/initalized float arrays of length iNumElements
 for (int i = 0; i < iNumElements; i++)
 {
    c[ i ] = a[ i ] + b[ i ];
  }
```

- **Review *oclVectorAdd* sample from NVIDIA OpenCL SDK**

  For brevity/clarity, error handling, console output and host comparision code is removed here

oclVectorAdd Execution Sequence

- Set Up
  - Set work sizes for kernel execution
  - Allocate and init host data buffers
  - Create context for GPU device
  - Query compute devices (in the context)
  - Create command queue (in the context)
  - Create buffers on the GPU device (in the context)
  - Create and build a program (in the context)
  - Create kernel
  - Set kernel arguments
- Core sequence
  - Copy (write) data from host to GPU
  - Launch kernel in command-queue
  - Copy (read) data from GPU to host… block
- Clean up

Kernel Code

- **Source code for the computation kernel, stored in text file (read from file and compiled at run time, e.g. during app. init)**

```
// OpenCL Kernel Function for element by element vector addition
// *********************************************************************
__kernel void VectorAdd ( __global float* a, __global float* b, __global float* c,
                          __global int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }

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

Host code: Declarations

```
// OpenCL Vars
cl_context cxGPUContext;                 // OpenCL context
cl_command_queue cqCommandQue;           // OpenCL command queue
cl_device_id* cdDevices;                 // OpenCL device list
cl_program cpProgram;                    // OpenCL program
cl_kernel ckKernel;                      // OpenCL kernel
cl_mem cmDevSrcA;                        // OpenCL device source buffer A
cl_mem cmDevSrcB;                        // OpenCL device source buffer B
cl_mem cmDevDst;                         // OpenCL device destination buffer
size_t szGlobalWorkSize;                 // 1D var for Total # of work items
size_t szLocalWorkSize;                  // 1D var for # of work items in the work group
size_t szParmDataBytes;                  // Byte size of context information
size_t szKernelLength;                   // Byte size of kernel code
char* cPathAndName = NULL;               // var for full paths to data, src, etc.
char* cSourceCL = NULL;                  // Buffer to hold source for compilation
int iNumElements = 11444777;             // Length of float arrays to process
```

Host code: Setup

```
// set Local work size dimensions
szLocalWorkSize = 256;

// set Global work size dimensions
// (rounded up to the nearest multiple of LocalWorkSize using C++ helper function)
szGlobalWorkSize = shrRoundUp ((int) szLocalWorkSize, iNumElements);

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

// Init host arrays  using C++ helper functions
shrFillArray ((float*) srcA, iNumElements);
shrFillArray ((float*) srcB, iNumElements);
```

Host code: Context, Device & Queue

```
// Create the OpenCL context on a GPU device
cxGPUContext = clCreateContextFromType (0, CL_DEVICE_TYPE_GPU,
                                         NULL, NULL, NULL);

// Get the list of GPU devices associated with context
clGetContextInfo (cxGPUContext, CL_CONTEXT_DEVICES, 0, NULL, &szParmDataBytes);
cdDevices = (cl_device_id*)  malloc (szParmDataBytes);
clGetContextInfo (cxGPUContext, CL_CONTEXT_DEVICES, szParmDataBytes,
                  cdDevices, NULL);

// Create a command-queue
cqCommandQue = clCreateCommandQueue (cxGPUContext, cdDevices[0], 0, NULL);
```

Host code: Create Memory Objects

```
// allocate the first source buffer memory object
cmDevSrcA = clCreateBuffer  (cxGPUContext, CL_MEM_READ_ONLY,
                             sizeof(cl_float) * iNumElements, NULL, NULL);

// allocate the second source buffer memory object
cmDevSrcB = clCreateBuffer  (cxGPUContext, CL_MEM_READ_ONLY,
                             sizeof(cl_float) * iNumElements, NULL, NULL);

// allocate the destination buffer memory object
cmDevDst = clCreateBuffer (cxGPUContext, CL_MEM_WRITE_ONLY,
                           sizeof(cl_float) * iNumElements, NULL, NULL);
```

Host code: Program & Kernel

```
// Read the OpenCL kernel in from source file using helper C++ functions
cPathAndName = shrFindFilePath(cSourceFile, argv[0]);
cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

// Create the program
cpProgram = clCreateProgramWithSource (cxGPUContext, 1, (const char **)&cSourceCL,
                                       &szKernelLength, NULL);

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

// Create the kernel
ckKernel = clCreateKernel (cpProgram, "VectorAdd", NULL);

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

Host code: Core Sequence

```
// Copy input data to GPU, compute, copy results back
// Runs asynchronous to host, up until blocking read at end

// Write data from host to GPU
clEnqueueWriteBuffer (cqCommandQue, cmDevSrcA, CL_FALSE, 0,
                        sizeof(cl_float) * szGlobalWorkSize, srcA, 0, NULL, NULL);
clEnqueueWriteBuffer (cqCommandQue, cmDevSrcB, CL_FALSE, 0,
                        sizeof(cl_float) * szGlobalWorkSize, srcB, 0, NULL, NULL);

// Launch kernel
clEnqueueNDRangeKernel (cqCommandQue, ckKernel, 1, NULL, &szGlobalWorkSize,
                        &szLocalWorkSize, 0, NULL, NULL);

// Blocking read of results from GPU to Host
clEnqueueReadBuffer (cqCommandQue, cmDevDst, CL_TRUE, 0,
                        sizeof(cl_float) * szGlobalWorkSize, dst, 0, NULL, NULL);
```

Cleanup

```
// Cleanup allocated objects
clReleaseKernel (ckKernel);
clReleaseProgram (cpProgram);
clReleaseCommandQueue (cqCommandQue);
clReleaseContext (cxGPUContext);
clReleaseMemObject (cmDevSrcA);
clReleaseMemObject (cmDevSrcB);
clReleaseMemObject (cmDevDst);
free (cdDevices);
free (cPathAndName);
free (cSourceCL);

// Free host memory
free(srcA);
free(srcB);
free (dst);
```

Console Output

C:\Windows\system32\cmd.exe

```
oclVectorAdd.exe Starting...

# of float elements per Array    = 11444777
Global Work Size                 = 11444992
Local Work Size                  = 256
# of Work Groups                 = 44707

Allocate and Init Host Mem...
clCreateContextFromType...
clGetContextInfo...
clCreateCommandQueue...
clCreateBuffer...
oclLoadProgSource (VectorAdd.cl)...
clCreateProgramWithSource...
clBuildProgram...
clCreateKernel...
clSetKernelArg...
clEnqueueNDRangeKernel...
clEnqueueReadBuffer...

Comparing against Host/C++ computation...

TEST PASSED      (Error Count = 0)

Starting Cleanup...

oclVectorAdd.exe Exiting...
Press <Enter> to Quit
-----------------------------------------------------------
```

What Next ?

- Begin hands-on development with the NVIDIA OpenCL SDK
- Read OpenCL Specification and the extensive materials provided with the OpenCL SDK
- Read and contribute to OpenCL forums at Kronos and NVIDIA

OpenCL Information and Resources

- **NVIDIA OpenCL Web Page**
  http://www.nvidia.com/object/cuda_opencl.html
- **NVIDIA OpenCL Forum**
  http://forums.nvidia.com/index.php?showforum=134
- **NVIDIA Registered Developer Extranet Site**
  https://nvdeveloper.nvidia.com/login.asp
  http://developer.nvidia.com/page/registered_developer_program.html
- **Khronos (current specification)**
  http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf
- **Khronos OpenCL Forum**
  http://www.khronos.org/message_boards/viewforum.php?f=28