AMD **Accelerated** Parallel Processing TECHNOLOGY

SAMPLE

Basic Debugging

1 Introduction

This is a stand-alone OpenCL sample for the novice programmer. It is a sample that demonstrates the basic OpenCL debugging techniques:

1. How to use KernelAnalyzer for debugging kernel compilation errors.
2. How to use `printf` inside a kernel.
3. How to use CodeXL to debug API errors or kernel functions.

2 AMD APP KernelAnalyzer

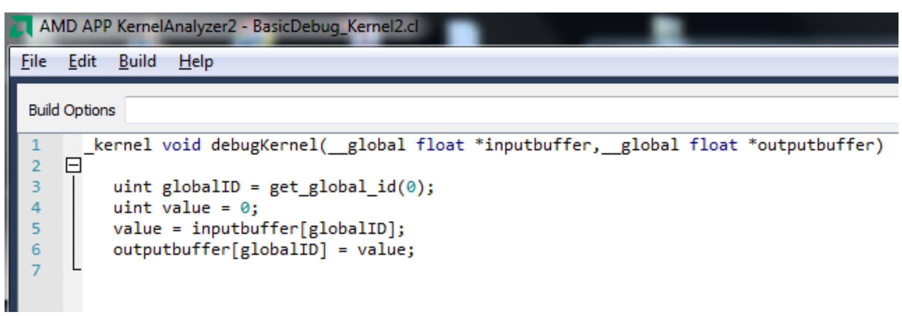
2.1 Overview

The AMD APP KernelAnalyzer is an OpenCL kernel code analysis tool for GPU applications. It contains an OpenCL compiler, a text editor, an assembly code window, and a statistics viewer.

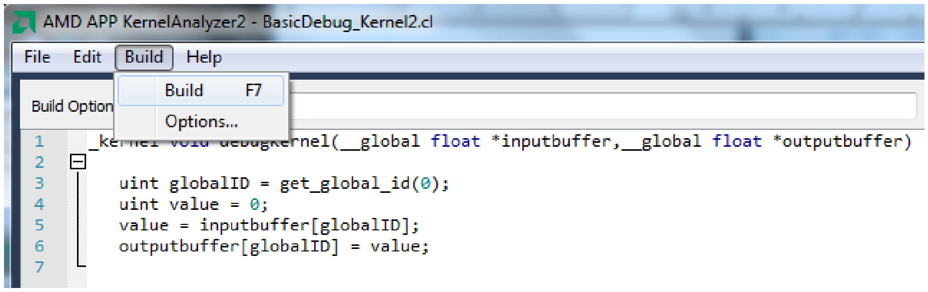
2.2 Getting Started

The following steps describe how to check for kernel compile-time errors.

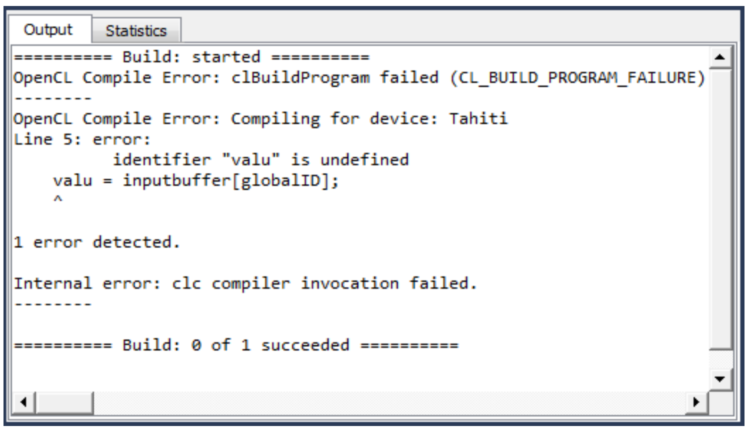
1. Run AMD APP KernelAnalyzer.
2. Open the OpenCL source file.
   This example uses `BasicDebug_Kernel2.cl`.



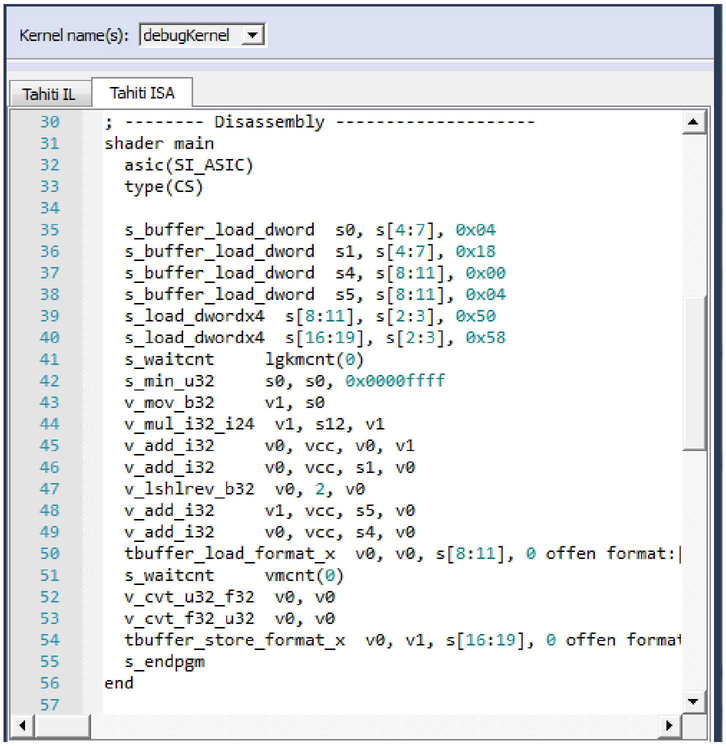
3. Select “Build” from the Build menu to compile the OpenCL kernels.



4. The compiler output is shown in the Output window. This is useful for debugging compilation errors.

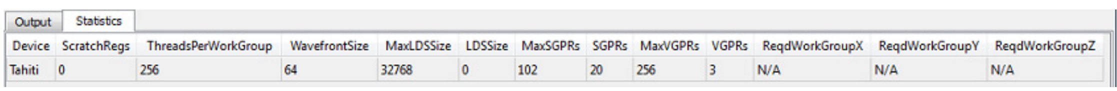


5. If the compilation is successful, the ISA codes of the targeted GPU will be shown in the assembly window.




```
; -------- Disassembly --------------------
shader main
  asic(SI_ASIC)
  type(CS)

  s_buffer_load_dword  s0, s[4:7], 0x04
  s_buffer_load_dword  s1, s[4:7], 0x18
  s_buffer_load_dword  s4, s[8:11], 0x00
  s_buffer_load_dword  s5, s[8:11], 0x04
  s_load_dwordx4  s[8:11], s[2:3], 0x50
  s_load_dwordx4  s[16:19], s[2:3], 0x58
  s_waitcnt     lgkmcnt(0)
  s_min_u32     s0, s0, 0x0000ffff
  v_mov_b32     v1, s0
  v_mul_i32_i24  v1, s12, v1
  v_add_i32     v0, vcc, v0, v1
  v_add_i32     v0, vcc, s1, v0
  v_lshlrev_b32  v0, 2, v0
  v_add_i32     v1, vcc, s5, v0
  v_add_i32     v0, vcc, s4, v0
  tbuffer_load_format_x  v0, v0, s[8:11], 0 offen format:|
  s_waitcnt     vmcnt(0)
  v_cvt_u32_f32  v0, v0
  v_cvt_f32_u32  v0, v0
  tbuffer_store_format_x  v0, v1, s[16:19], 0 offen format
  s_endpgm
end
```

6. The Statistics tab shows some statistical information of a kernel on a particular GPU. Moving the mouse over a table header shows a tool tip explaining the meaning of data in that column.



Device	ScratchRegs	ThreadsPerWorkGroup	WavefrontSize	MaxLDSSize	LDSSize	MaxSGPRs	SGPRs	MaxVGPRs	VGPRs	ReqdWorkGroupX	ReqdWorkGroupY	ReqdWorkGroupZ
Tahiti	0	256	64	32768	0	102	20	256	3	N/A	N/A	N/A

3 Using `printf` Inside a Kernel

The built-in `printf` function writes output to an implementation-defined stream, such as stdout, under control of the string pointed to by a format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated, but otherwise ignored. The `printf` function returns when the end of the format string is encountered.

The following steps are a guide to using the `printf` function.

1. Function prototype : `int printf(constant char * restrict format, …)`
2. `printf` output synchronization:

   Calling `clFinish` on a command queue flushes all pending output by `printf` in previously enqueued and completed commands to the implementation-defined output stream. In case `printf` is executed from multiple work-items concurrently, there is no guarantee of ordering with respect to written data.

3. Differences between the C and the OCL version of printf.
    a. Since `format` is in the constant address space, it must be resolvable at compile time; thus, it cannot be dynamically created by the executing program.
    b. OpenCL C adds the optional `vn` vector specifier to support printing of vector types.
    c. In OpenCL C, `printf` returns 0 if it was executed successfully; otherwise, it returns 1.
4. More information can be found in section 6.12.13) of *The OpenCL Specification, v 1.2.*

4 Implementation Details

This sample shows how to use the function `printf` in an OpenCL kernel to export some information for debug purposes.

4.1 Kernel Code

```
__kernel void printfKernel(__global float *inputbuffer)
{
    uint globalID = get_global_id(0);
    uint groupID = get_group_id(0);
    uint localID = get_local_id(0);
    if(10 == globalID)
{
        float4 f = (float4)(inputbuffer[0], inputbuffer[1], inputbuffer[2],
            inputbuffer[3]);
        printf("Output vector data: f4 = %2.2v4hlf\n", f);
    }
    __local int data[256];
    data[localID] = localID;
    barrier(CLK_LOCAL_MEM_FENCE);
    if(0 == localID)
    {
        printf("\tThis is group %d\n",groupID);
        printf("\tOutput LDS data:  %d\n",data[0]);
    }
    printf("the global ID of this thread is : %d\n",globalID);
}
```

4.2 Code Interpretation

1. Get global ID, local ID and group ID of every thread:

    ```
    uint globalID = get_global_id(0);
    uint groupID = get_group_id(0);
    uint localID = get_local_id(0);
    ```

2. When debugging the kernel, define temporary variables and do some calculations; then, output some information for a specific thread (use vector type here):

    ```
    if(10 == globalID)
    {
        float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
    printf("f4 = %2.2v4hlf\n", f);
    }
    ```

The meaning of format string `2.2v4hlf` is:

`V4` – Specifies that the `f` conversion specifier applies to a vector argument.
Since the vector type is float4, use `v4` here.

`hl` – Specifies that the `f` conversion specifier applies to a float4 argument.

3. Sometimes we use local memory in the kernel. When debugging the kernel, output some local memory information for a specific thread:

```
if(0 == localID)
{
printf("\tThis is group %d\n",groupID);
printf("\tOutput LDS data:  %d\n",data[0]);
}
```

4. To know the calculation process or calculate the sequence of the threads, output some information according to the global ID (`globalID` is private value here).

```
printf("the global ID of this thread is : %d\n",globalID.
```

5 Using CodeXL to Debug API Errors or Kernel Functions

CodeXL is an OpenCL and OpenGL debugger. It brings together the GPU and CPU compute tools to enable faster and more robust development of OpenCL and OpenGL accelerated applications, specifically for Heterogeneous Compute application development and APUs.

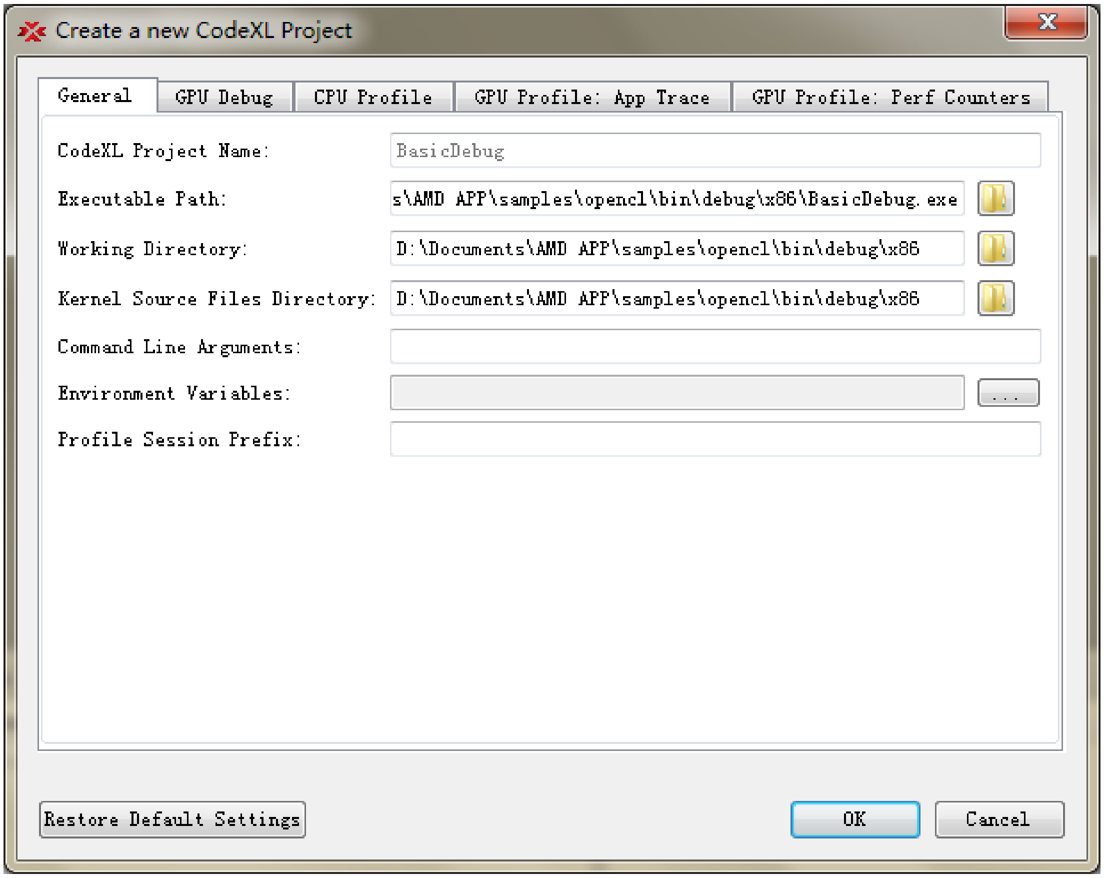
CodeXL will be available in three versions:

1. Plug-in to Microsoft® Visual Studio®.
2. Stand-alone software package for the Windows platform.
3. Stand-alone software package for Linux environments.

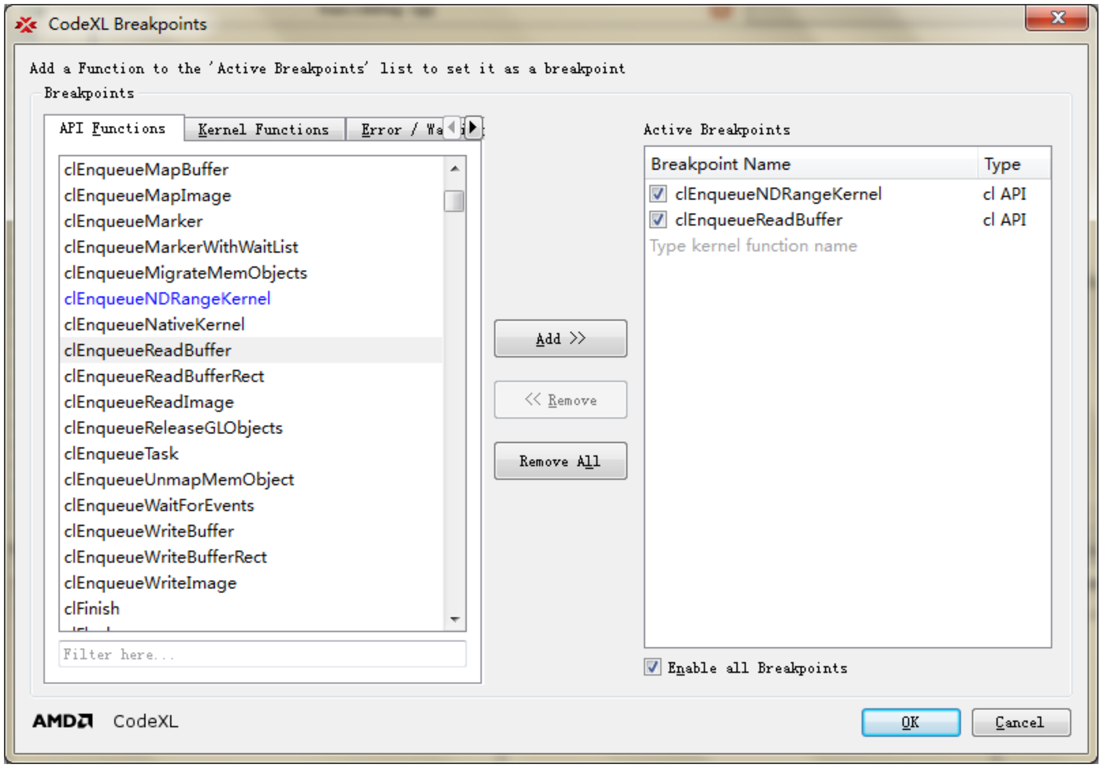
The AMD website for more information is:
http://developer.amd.com/tools/hc/CodeXL/pages/default.aspx

To use CodeXL:

1. Install AMD CodeXL.
2. On the CodeXL Home Page. Select "Create a New Project" to bring up the New Project Wizard.



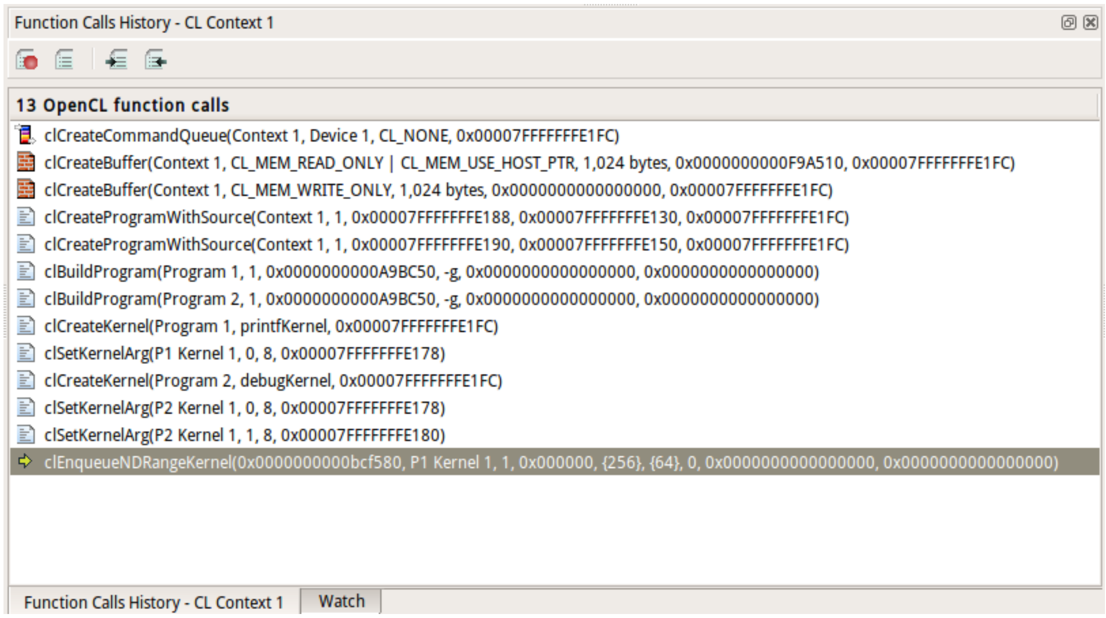
3. Host OpenCL API debugging:
   a. The Breakpoint dialog lets you choose OpenCL and OpenGL API function breakpoints, as well as kernel function name breakpoints.
   b. Add an API Functions breakpoint.



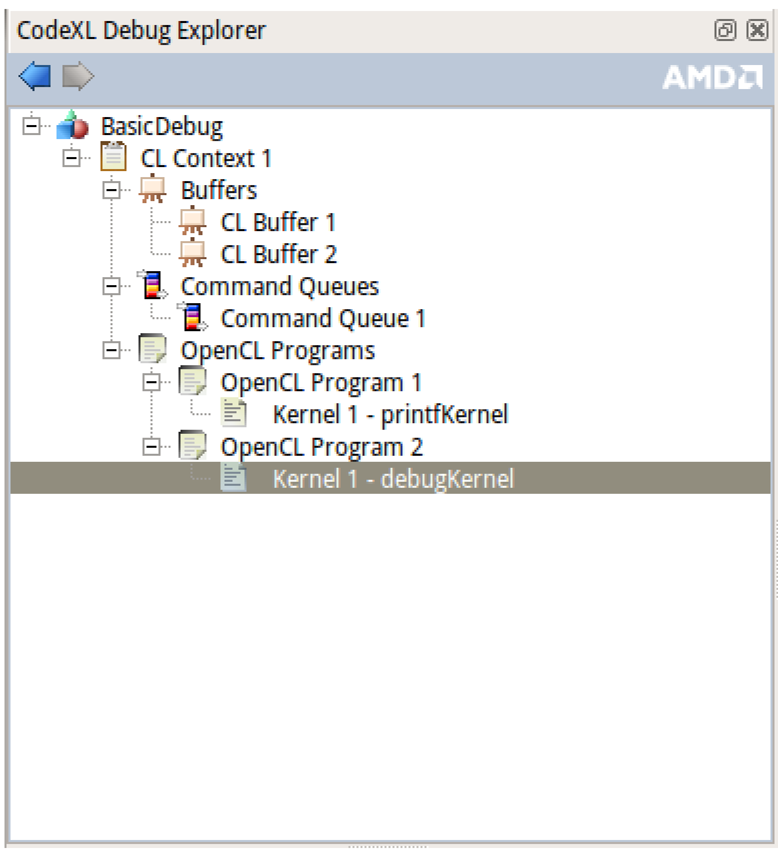
c. Continue to run the program. It will stop at the breakpoint

```
//set kernel args.
status = clSetKernelArg(kernel1, 0, sizeof(cl_mem), (void *)&inputBuffer);

//create debug kernel
cl_kernel kernel2 = clCreateKernel(program2, "debugKernel", &status);
if (status != CL_SUCCESS)
{
  std::cout<<"Error: Creating kernel failed!"<<std::endl;
  return Failed;
}
//set kernel args.
status = clSetKernelArg(kernel2, 0, sizeof(cl_mem), (void *)&inputBuffer);
status = clSetKernelArg(kernel2, 1, sizeof(cl_mem), (void *)&outputBuffer);


size_t global_threads[1];
size_t local_threads[1];
global_threads[0] = GlobalThreadSize;
  local_threads[0] = GroupSize;

//execute the kernel
status = clEnqueueNDRangeKernel(commandQueue, kernel1, 1, NULL, global_threads, local_threads, 0, NULL, NULL);
if (status != CL_SUCCESS)
{
  std::cout<<"Error: Enqueue kernel onto command queue failed!"<<std::endl;
  return Failed;
}
status = clFinish(commandQueue);
```

d. The Function Calls History” window shows a history of the API calls and the parameters:

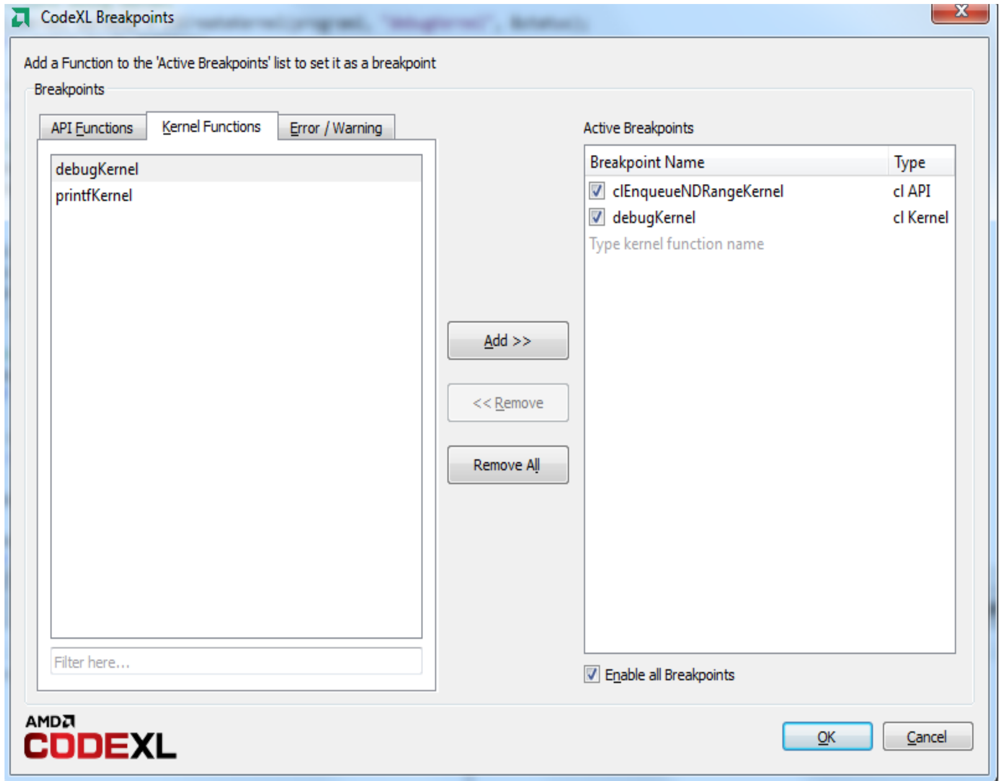


e. CodeXL Explorer. Expand the Context tree showing the buffers, queues, programs, and kernels created in that context.

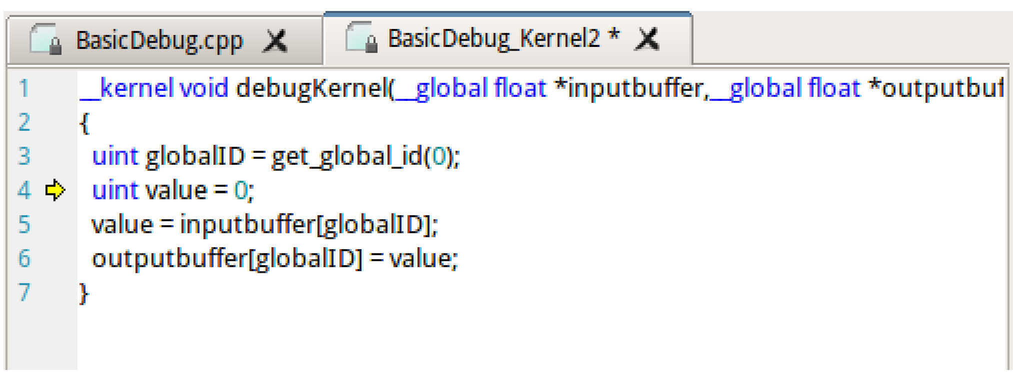


4. Kernel debugging.

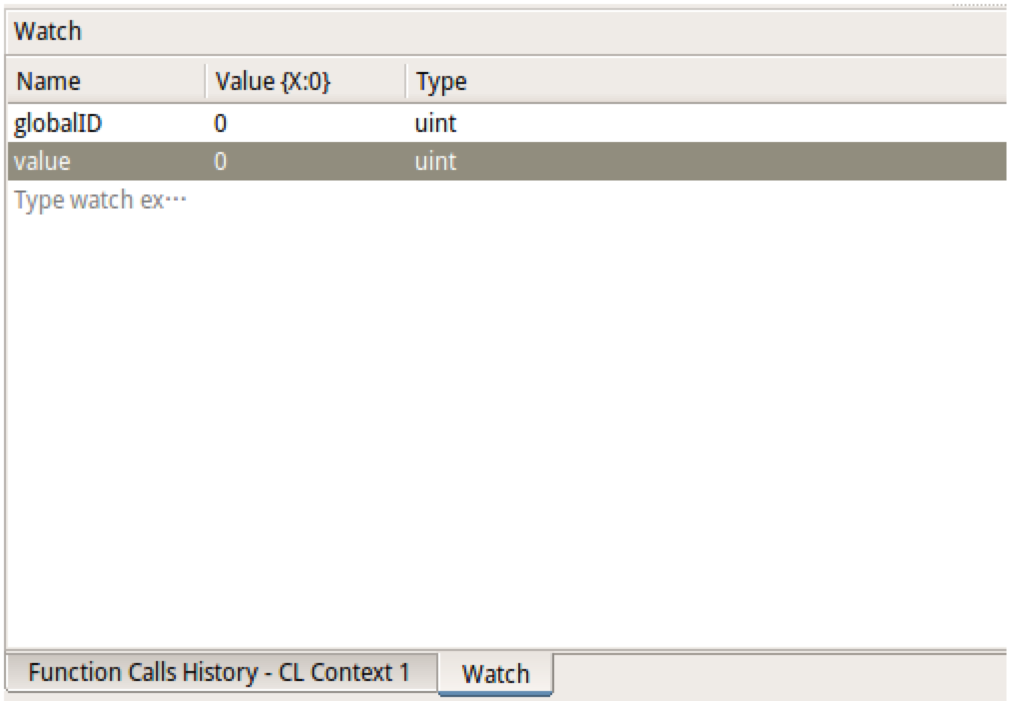
   a. After building the OpenCL program, set a breakpoint in the kernel.



   b. Continue to run the program until it stops inside the kernel.

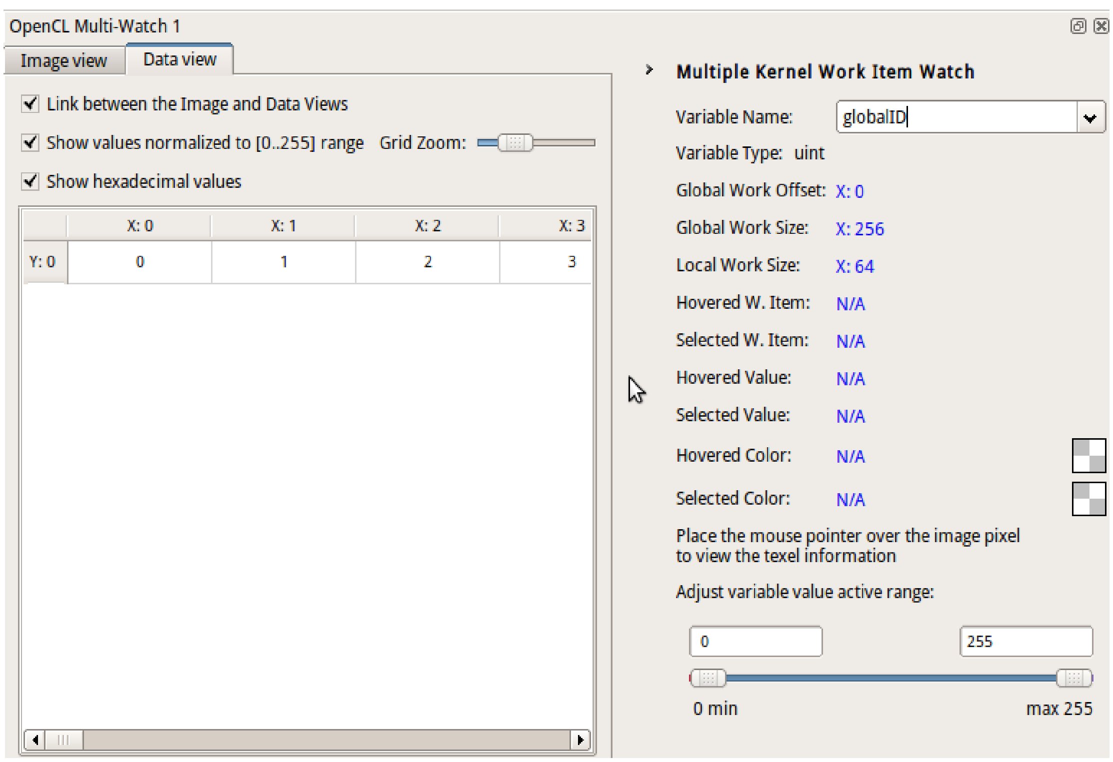


   c. Track the value of the variables with the “Watch” window. First watch for work item 0.



d. Use the work-item tool to switch to item 1. The variables in the watch window are updated in real-time.

e. See the value of `globalID` of every thread. Check this with the “OpenCL Multi-Watch” window.



Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For AMD Accelerated Parallel Processing:
URL: **developer.amd.com/appsdk**
Developing: **developer.amd.com/**
Support: **developer.amd.com/appsdksupport**
Forum: **developer.amd.com/openclforum**