Introduction to OpenCL on TI Embedded Processors

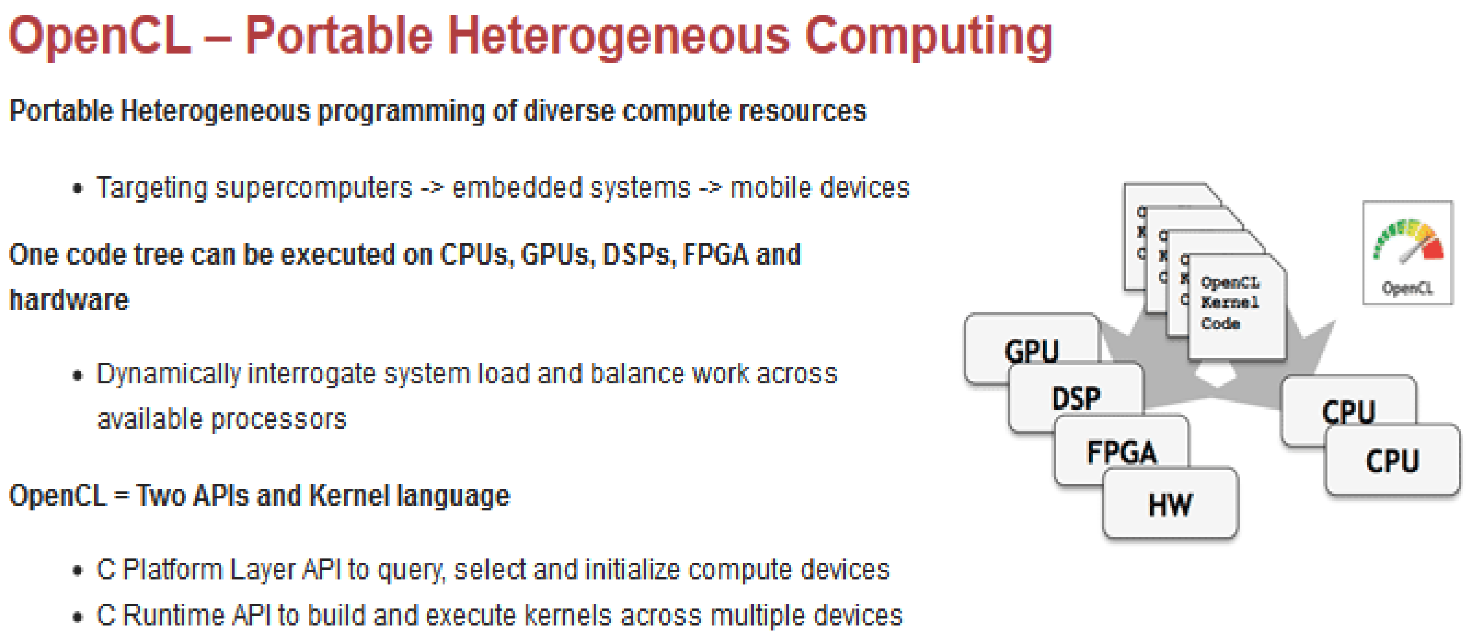
Agenda

- OpenCL Overview
- Why and When to Use OpenCL on TI Embedded Processors
- Processor SDK OpenCL Examples
- TI Design Example Code Walkthrough

OpenCL Overview

Introduction to OpenCL

OpenCL Parallel Language for Heterogeneous Model



OpenCL – Portable Heterogeneous Computing

Portable Heterogeneous programming of diverse compute resources

- Targeting supercomputers -> embedded systems -> mobile devices

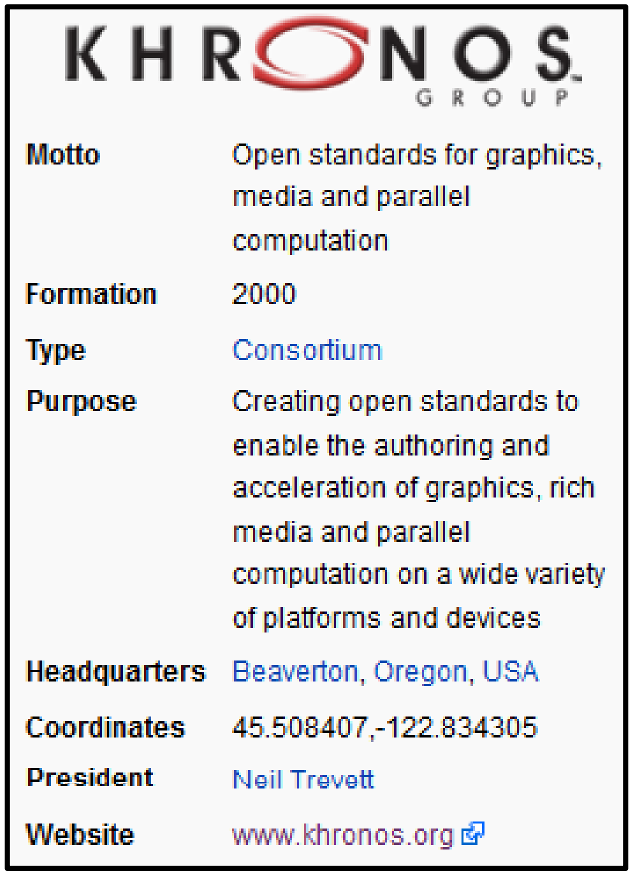
One code tree can be executed on CPUs, GPUs, DSPs, FPGA and hardware

- Dynamically interrogate system load and balance work across available processors

OpenCL = Two APIs and Kernel language

- C Platform Layer API to query, select and initialize compute devices
- C Runtime API to build and execute kernels across multiple devices

OpenCL Kernel Code
OpenCL
GPU
DSP
FPGA
HW
CPU
CPU



KHRONOS GROUP

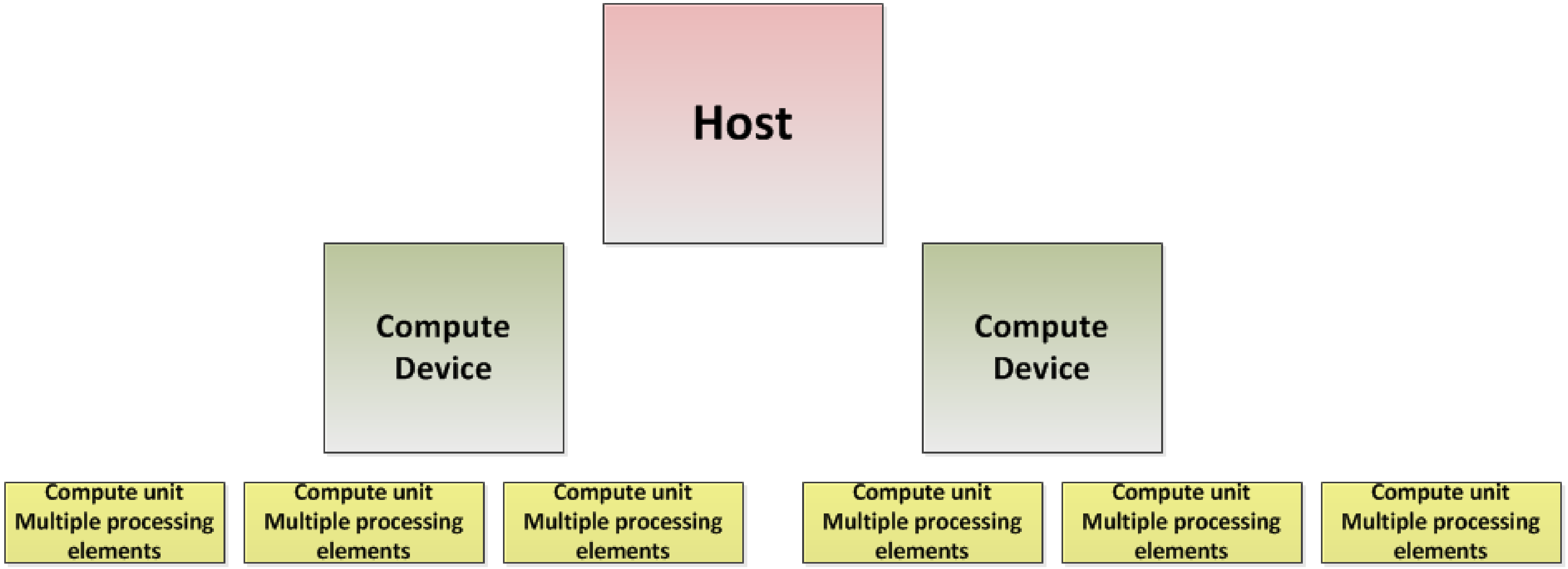
Motto	Open standards for graphics, media and parallel computation
Formation	2000
Type	Consortium
Purpose	Creating open standards to enable the authoring and acceleration of graphics, rich media and parallel computation on a wide variety of platforms and devices
Headquarters	Beaverton, Oregon, USA
Coordinates	45.508407,-122.834305
President	Neil Trevett
Website	www.khronos.org

- The content of this slide originates from the OpenCL standards body Khronos.
- AM57x has the ARM Cortex-A15 as a host, and DSP cores as accelerators.
- The TI OpenCL implementation is compliant with OpenCL 1.1

Benefits of Using OpenCL on TI Processors

- Easy porting between devices
- No need to understand memory architecture
- No need to worry about MPAX and MMU
- No need to worry about coherency
- No need to build/configure/use IPC between ARM and DSP
- No need to be an expert in DSP code, architecture, or optimization

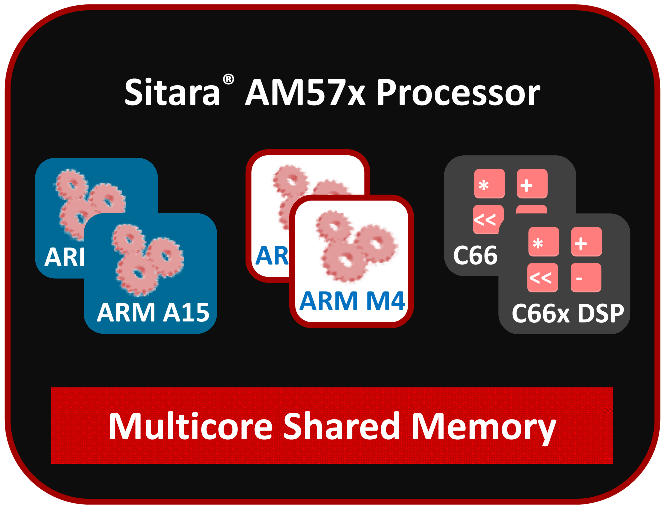
OpenCL Platform Model

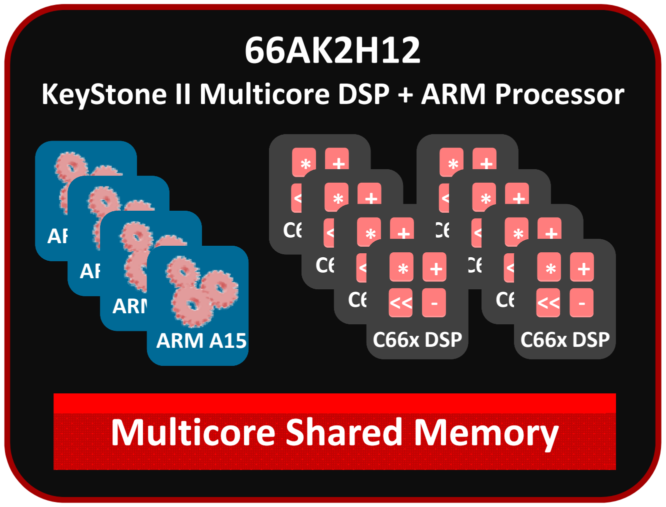

Host
Compute Device
Compute Device
Compute unit Multiple processing elements
Compute unit Multiple processing elements
Compute unit Multiple processing elements
Compute unit Multiple processing elements
Compute unit Multiple processing elements
Compute unit Multiple processing elements

- A host is connected to one or more OpenCL compute devices.
- An OpenCL compute device is a collection of one or more compute units.
- Each compute unit may have multiple processing elements.

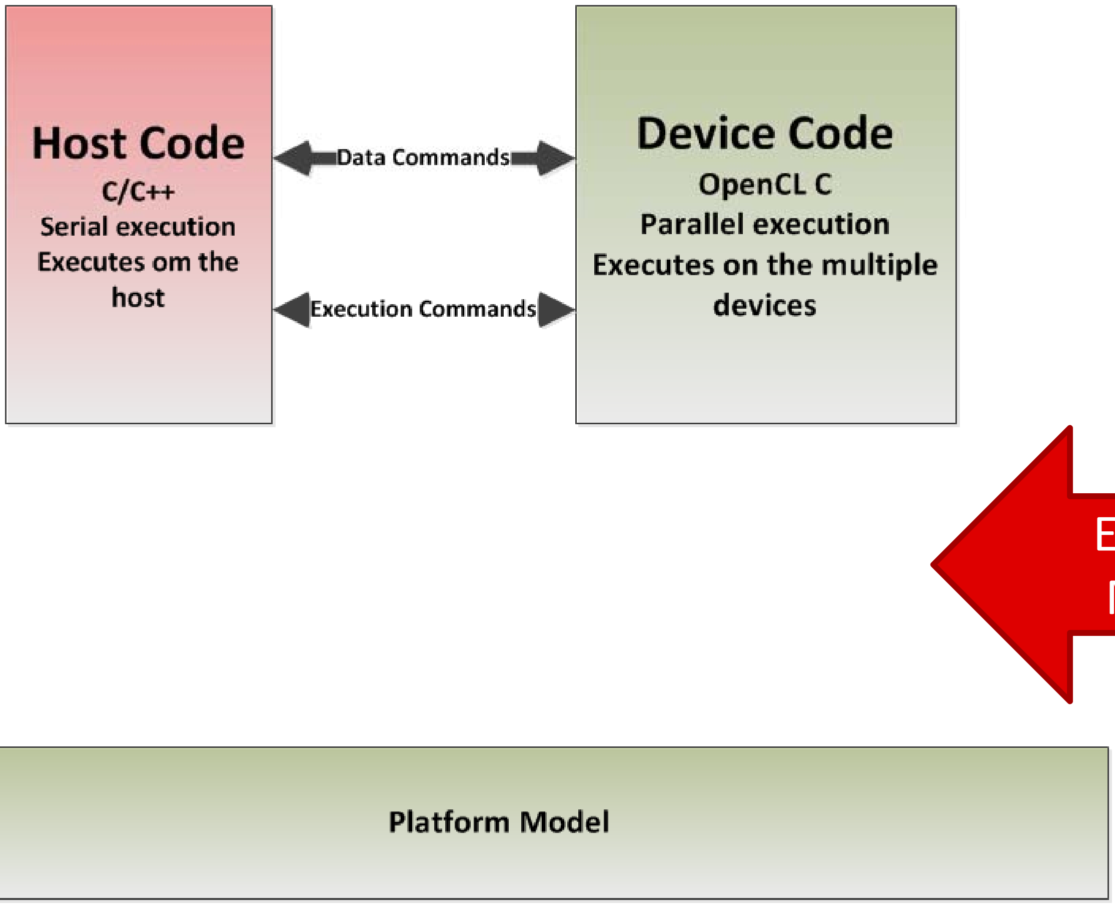
OpenCL TI Platform Model

- **ARM Cortex-A15 is the host:** Commands are submitted from the host to the OpenCL devices (execution and memory move).
- **All C66x DSP CorePacs are OpenCL compute devices. Each DSP core is a compute unit.**
  An OpenCL device is viewed by the OpenCL programmer as a single virtual processor. This means that the programmer does not need to know how many cores are in the device. OpenCL runtime efficiently divides the total processing effort across the cores.
  NOTE: AM57x and 66K2H12 have the same OpenCL code.





OpenCL Applications Model

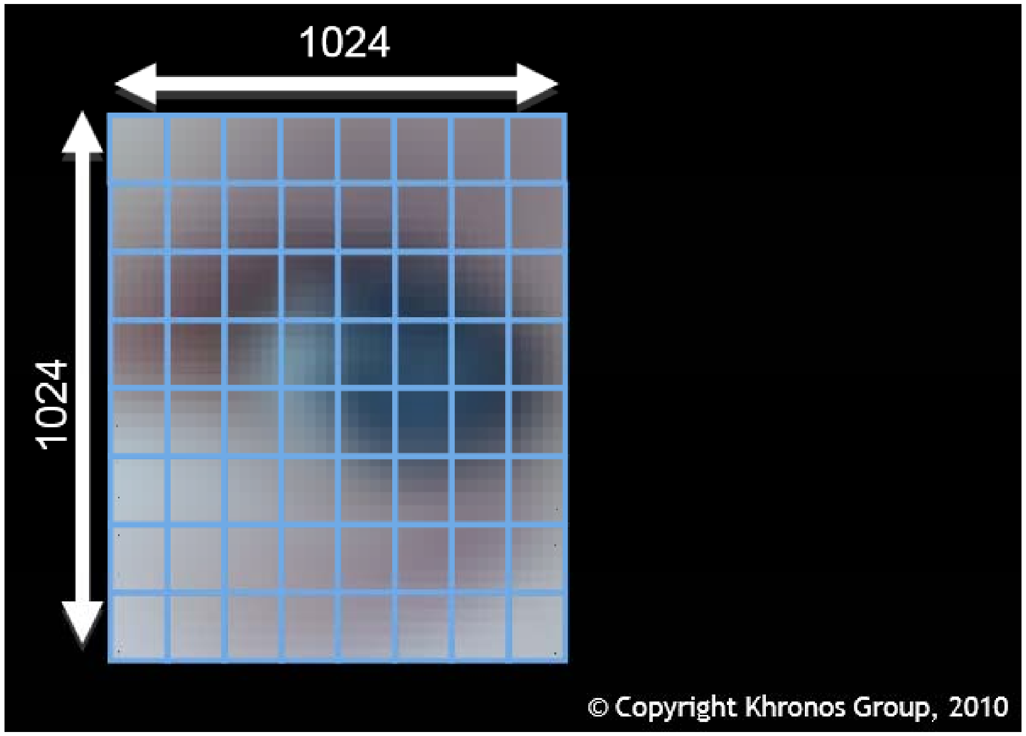


Serial Code	Host
Parallel Code	Multiple DSP cores
Serial Code	Host
Parallel Code	Multiple DSP cores

Execution Model
Memory Model

Texas Instruments

OpenCL Execution Model

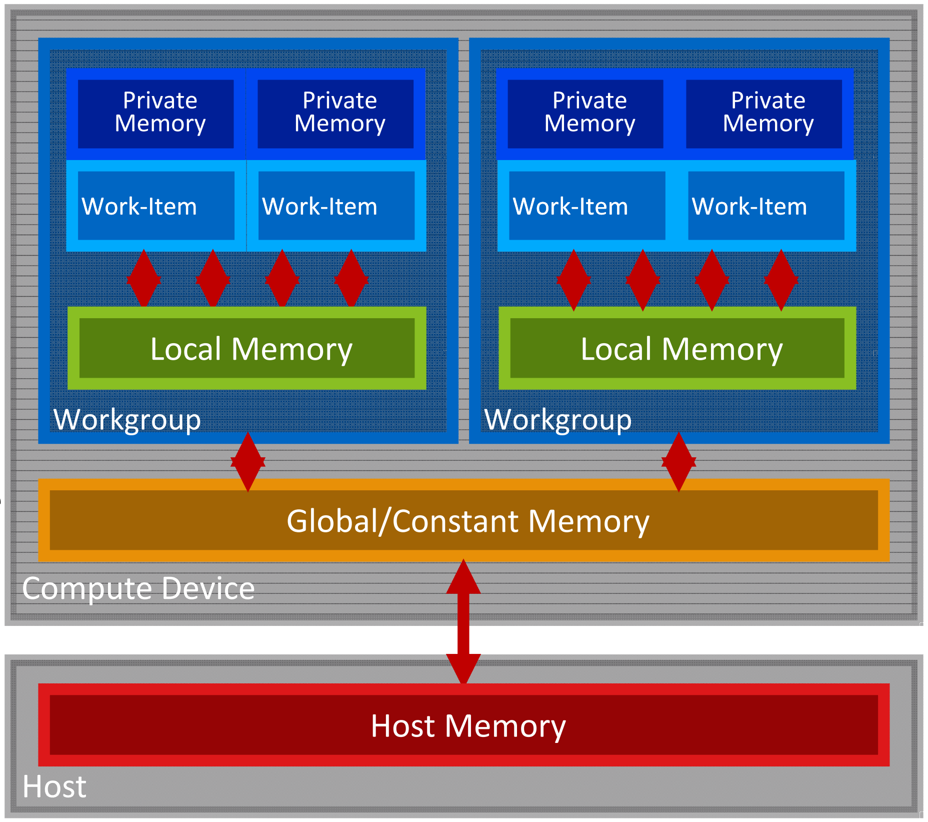


Context Define device and state	**Host**
Processing Algorithm One or more kernel(s)	**Compute Device** One or more Compute Unit(s)
Work Group	**Compute Unit** One or more Compute Element(s)
Work Item	**Compute (Processing) Element**

Work items => Work group

OpenCL Memory Model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a workgroup, local to a compute unit (core)
- Global/Constant Memory
  - Shared across all compute units (cores) in a compute device
- Host Memory
  - Attached to the Host CPU
  - Can be distinct from global memory
    - Read / Write buffer model
  - Can be same as global memory
    - Map / Unmap buffer model



Memory management is explicit;
Commands move data from
host -> global -> local *and* back.

OpenCL Execution Model

Definitions

Context

Device

Command queue

Global buffers

Build Kernels

Get source from file (or part of the code) and compile it at run-time

OR

Get binaries, either as stand-alone .out or from a library

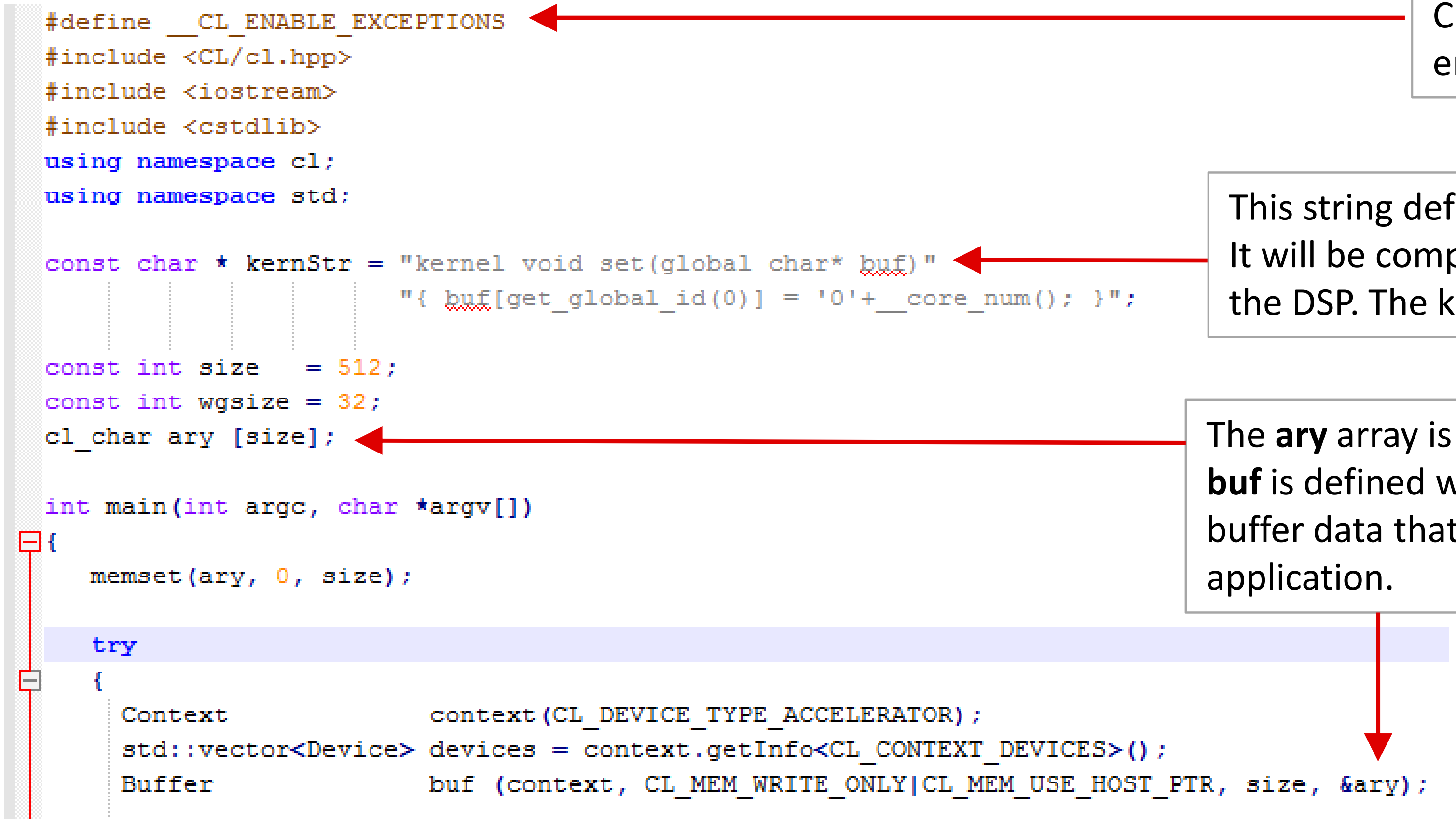
Manipulate Memory & Buffers

Move data and define local memory

Execute

Dispatch all work items

Simple Function Code Walkthrough



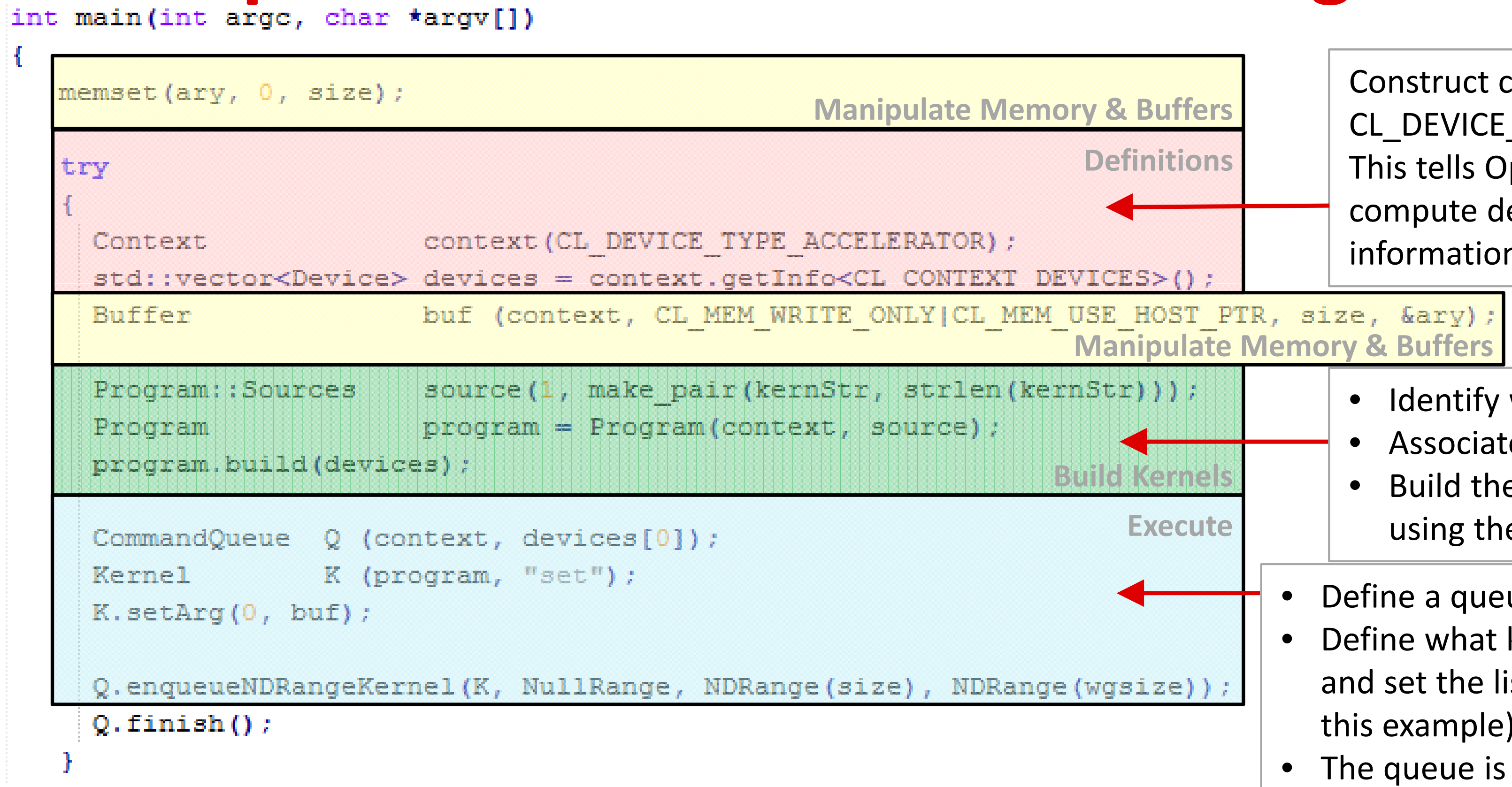
```
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
#include <iostream>
#include <cstdlib>
using namespace cl;
using namespace std;

const char * kernStr = "kernel void set(global char* buf)"
                       "{ buf[get_global_id(0)] = '0'+__core_num(); }";

const int size   = 512;
const int wgsize = 32;
cl_char ary [size];

int main(int argc, char *argv[])
{
   memset(ary, 0, size);

   try
   {
     Context               context(CL_DEVICE_TYPE_ACCELERATOR);
     std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
     Buffer                buf  (context, CL_MEM_WRITE_ONLY|CL_MEM_USE_HOST_PTR, size, &ary);
```

The OpenCL include file CL_ENABLE_EXCEPTIONS enables C++ class checking.

This string defines the kernel.
It will be compiled for the DSP and runs on the DSP. The kernel name is **set**.

The **ary** array is defined in the host memory. **buf** is defined with a pointer to **ary**, which is buffer data that is already allocated by the application.

Simple Function Code Walkthrough



```
int main(int argc, char *argv[])
{
    memset(ary, 0, size);
    try
    {
        Context               context(CL_DEVICE_TYPE_ACCELERATOR);
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
        Buffer                buf (context, CL_MEM_WRITE_ONLY|CL_MEM_USE_HOST_PTR, size, &ary);

        Program::Sources      source(1, make_pair(kernStr, strlen(kernStr)));
        Program               program = Program(context, source);
        program.build(devices);

        CommandQueue    Q (context, devices[0]);
        Kernel          K (program, "set");
        K.setArg(0, buf);

        Q.enqueueNDRangeKernel(K, NullRange, NDRange(size), NDRange(wgsize));
        Q.finish();
    }
```

Construct context CL_DEVICE_TYPE_ACCELERATOR is DSP. This tells OpenCL the architecture of the compute device. **getinfo** returns information about the device or devices.

- Identify where the kernel(s) is defined.
- Associate the program with the kernel.
- Build the program for the devices using the right code generation tools.

- Define a queue to the device.
- Define what kernel is sent to the device and set the list of arguments (only one in this example).
- The queue is connected to the device and the kernel is compiled and set. Start execution by calling the enqueue function. NDRange class provides the dimensions.

Why and When to Use OpenCL

Introduction to OpenCL

Using OpenCL on TI DSP Devices

- HPC machines with large numbers of computational units – no issue. Use OpenCL or CUDA or similar.
- For devices like 66AK2H12, where there are 4 ARM A15 cores and 8 DSP C66x cores:
  - 8 DSPs process many signal-processing algorithms.
  - Some of the ARM cores can be on a separate compute device.
    - Not supported currently
- Rule of thumb: Use OpenCL when high processing power is needed. Compare it to the overhead associated with dispatching DSP execution.
  - The example NULL (from the release examples that are discussed later) provides the overhead that is associated with execution of null program by the DSP.

Using OpenCL on TI Sitara Devices

- For devices like AM57x where there is 1-2 ARM (1.5G) cores and 1-2 DSP C66x (600 MHZ) cores:
  - ARM Cortex-A15 is high-performance processor.
    - But it is not as efficient as DSP for some algorithms.
  - Consider the overhead that is associated with building the OpenCL structure and the run-time compiling of the kernel.
    - There is directive that keeps the previous compiled binaries in cache between calls.
- Rule of thumb … Use OpenCL when the following are true:
  - The same kernel runs many (infinite) times (the overhead is negligible) and the ARM can execute other functions at the same time.
  - Kernel involves complex processing algorithms, especially if real-time is a consideration.
- Benchmark your code with and without OpenCL and compare.

Processor SDK OpenCL Examples

Introduction to OpenCL

OpenCL in Processor SDK Linux Release

- OpenCL implementation is part of the Processor SDK Linux perspective.
- TI standard file system has several OpenCL examples: **`/usr/shared/ti/examples/opencl`**

OpenCL Examples in Processor SDK Linux File System

```
root@am57xx-evm:/usr/share/ti/examples/opencl# ls -ltr
-rwxr-xr-x    1 root      root     2450 Aug 26 12:30 make.inc
-rwxr-xr-x    1 root      root      548 Aug 26 12:30 Makefile
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 vecadd
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 simple
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 platforms
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 ooo_callback
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 offline_embed
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 offline
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 null
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 matmpy
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 float_compute
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 edmamgr
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 dsplib_fft
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 ccode
drwxr-xr-x    2 root      root     4096 Aug 26 12:55 buffer
```

Executing OpenCL Examples: ccode

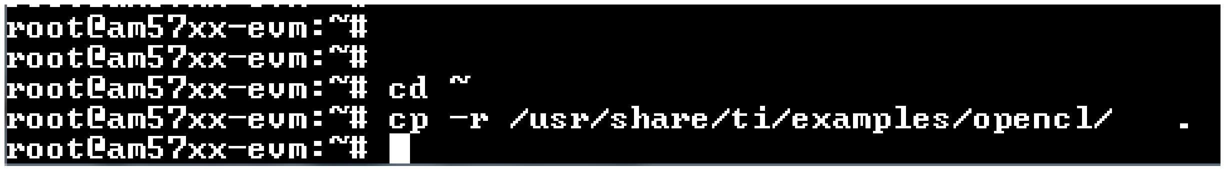
```
root@am57xx-evm:/usr/share/ti/examples/opencl#
root@am57xx-evm:/usr/share/ti/examples/opencl# cd ccode
root@am57xx-evm:/usr/share/ti/examples/opencl/ccode# ls -ltr
-rw-r--r--    1 root     root          2107 Aug 26 12:30 oclwrapper.cl
-rw-r--r--    1 root     root        377656 Aug 26 12:30 main.o
-rw-r--r--    1 root     root          6544 Aug 26 12:30 main.cpp
-rw-r--r--    1 root     root          6376 Aug 26 12:30 ccode.obj
-rw-r--r--    1 root     root          2036 Aug 26 12:30 ccode.c
-rw-r--r--    1 root     root           171 Aug 26 12:30 Makefile
-rwxr-xr-x    1 root     root         22524 Aug 26 12:30 ccode
root@am57xx-evm:/usr/share/ti/examples/opencl/ccode# ./ccode
[  540.955345] NET: Registered protocol family 41
Success!
root@am57xx-evm:/usr/share/ti/examples/opencl/ccode#
```

Executing OpenCL Examples: vecadd

```
root@am57xx-evm:/usr/share/ti/examples/opencl/ccode# cd ../vecadd
root@am57xx-evm:/usr/share/ti/examples/opencl/vecadd# ls
Makefile           main_map_prof.cpp  main_prof.cpp
main.cpp           main_md.cpp        vecadd
main.o             main_md.o          vecadd_md
root@am57xx-evm:/usr/share/ti/examples/opencl/vecadd# ./vecadd
DEVICE: TI Multicore C66 DSP

Offloading vector addition of 8192K elements...

Kernel Exec : Queue  to Submit: 7 us
Kernel Exec : Submit to Start : 68 us
Kernel Exec : Start  to End   : 32176 us

Success!
```

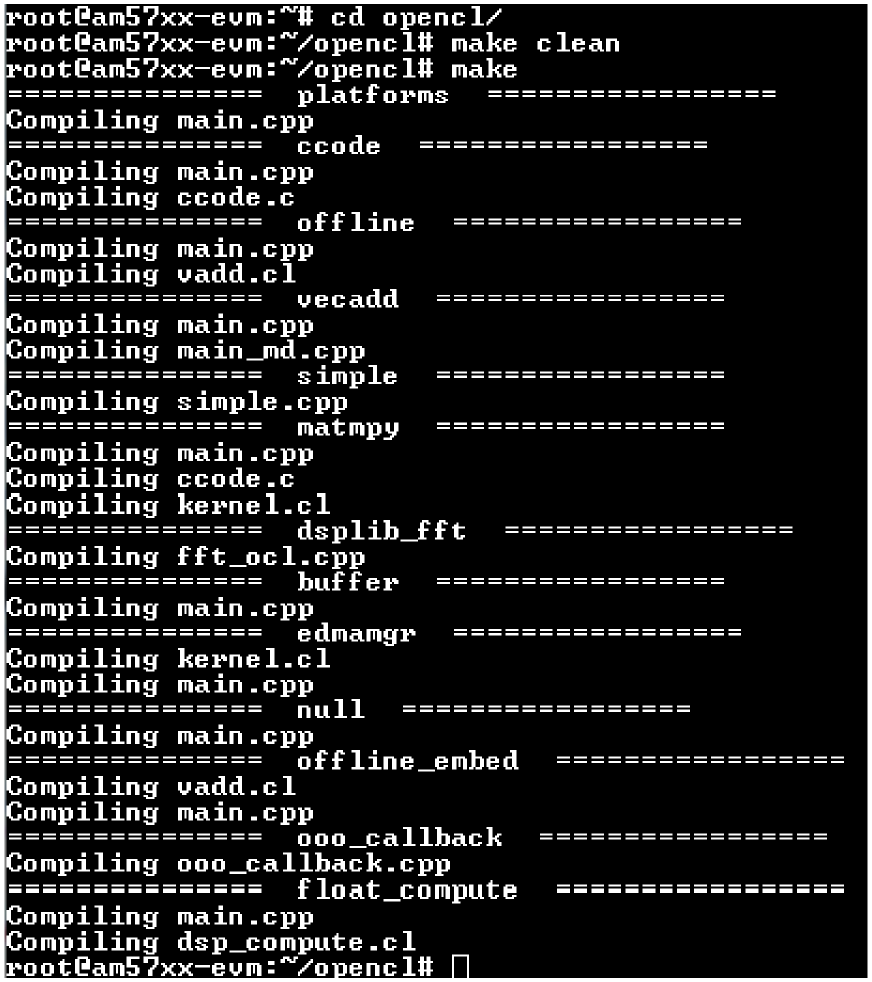
Building OpenCL Examples

- Copy the OpenCL examples directory into your home directory.



```
root@am57xx-evm:~#
root@am57xx-evm:~#
root@am57xx-evm:~# cd ~
root@am57xx-evm:~# cp -r /usr/share/ti/examples/opencl/    .
root@am57xx-evm:~#
```

- Go to /opencl directory, do **make clean** and then **make**. All directories will be built.
- Next, run any of the projects by going to the project directory and running the executable.

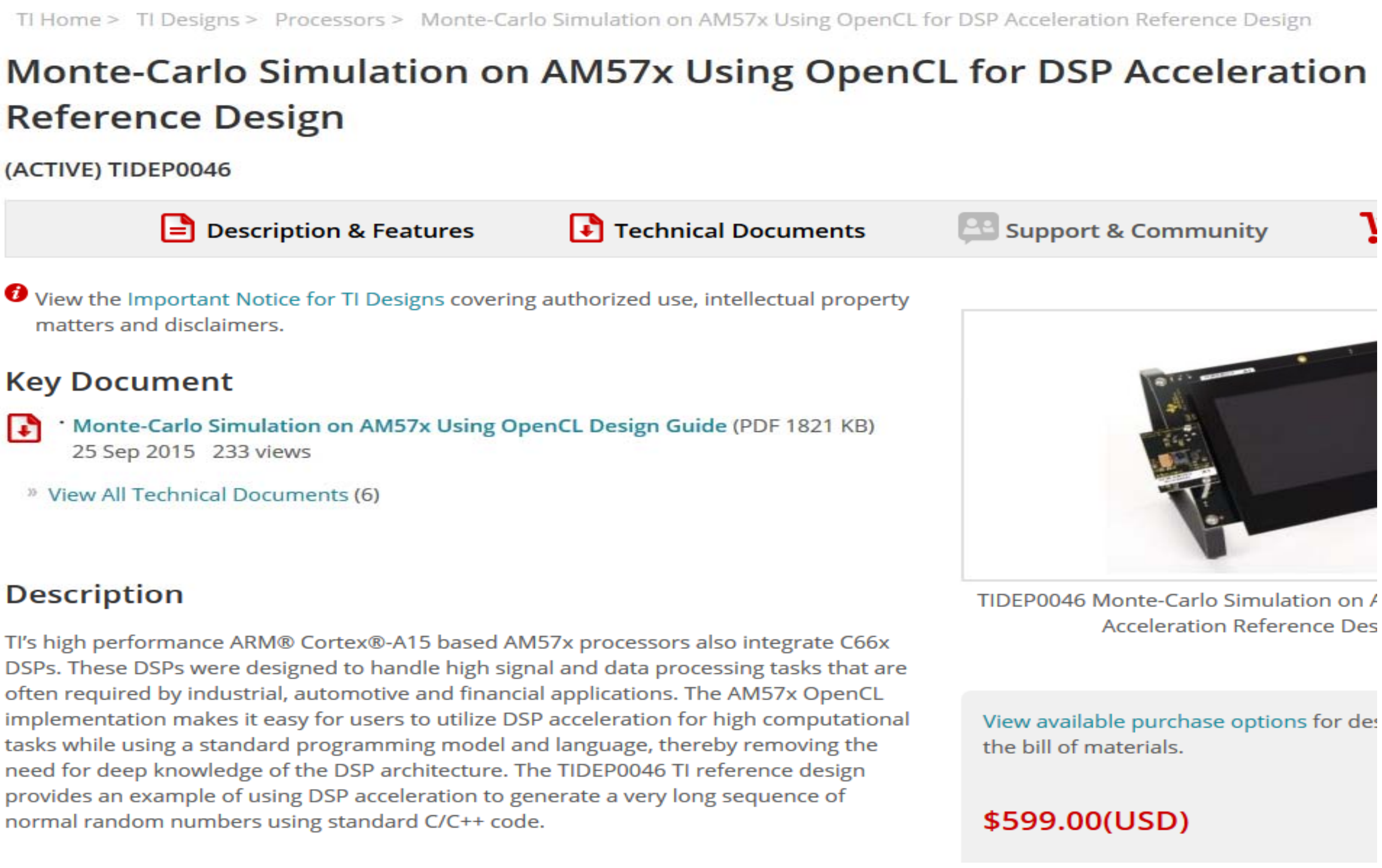


```
root@am57xx-evm:~# cd opencl/
root@am57xx-evm:~/opencl# make clean
root@am57xx-evm:~/opencl# make
===============  platforms  =================
Compiling main.cpp
===============  ccode  =================
Compiling main.cpp
Compiling ccode.c
===============  offline  =================
Compiling main.cpp
Compiling vadd.cl
===============  vecadd  =================
Compiling main.cpp
Compiling main_md.cpp
===============  simple  =================
Compiling simple.cpp
===============  matmpy  =================
Compiling main.cpp
Compiling ccode.c
Compiling kernel.cl
===============  dsplib_fft  =================
Compiling fft_ocl.cpp
===============  buffer  =================
Compiling main.cpp
===============  edmamgr  =================
Compiling kernel.cl
Compiling main.cpp
===============  null  =================
Compiling main.cpp
===============  offline_embed  =================
Compiling vadd.cl
Compiling main.cpp
===============  ooo_callback  =================
Compiling ooo_callback.cpp
===============  float_compute  =================
Compiling main.cpp
Compiling dsp_compute.cl
root@am57xx-evm:~/opencl#
```

TI Design Example Code Walkthrough

Introduction to OpenCL

OpenCL TI Design: www.ti.com/tool/TIDEP0046



TI Home > TI Designs > Processors > Monte-Carlo Simulation on AM57x Using OpenCL for DSP Acceleration Reference Design

Monte-Carlo Simulation on AM57x Using OpenCL for DSP Acceleration Reference Design

(ACTIVE) TIDEP0046

View the Important Notice for TI Designs covering authorized use, intellectual property matters and disclaimers.

Key Document

- Monte-Carlo Simulation on AM57x Using OpenCL Design Guide (PDF 1821 KB)
  25 Sep 2015 233 views

» View All Technical Documents (6)

Description

TI's high performance ARM® Cortex®-A15 based AM57x processors also integrate C66x DSPs. These DSPs were designed to handle high signal and data processing tasks that are often required by industrial, automotive and financial applications. The AM57x OpenCL implementation makes it easy for users to utilize DSP acceleration for high computational tasks while using a standard programming model and language, thereby removing the need for deep knowledge of the DSP architecture. The TIDEP0046 TI reference design provides an example of using DSP acceleration to generate a very long sequence of normal random numbers using standard C/C++ code.

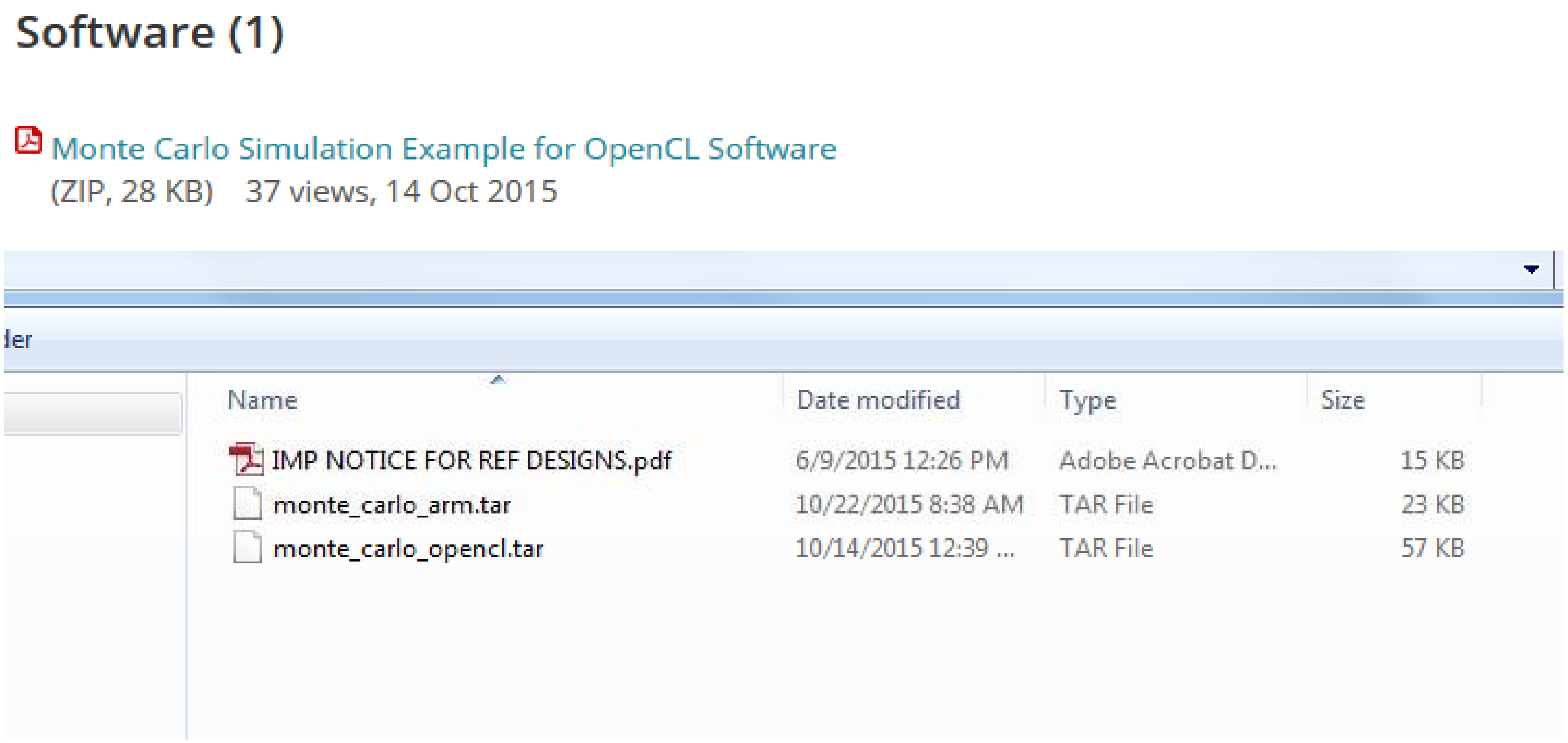
TIDEP0046 Monte-Carlo Simulation on A
Acceleration Reference Des

View available purchase options for des
the bill of materials.

$599.00(USD)

OpenCL TI Design: Resources

http://www.ti.com/tool/TIDEP0046

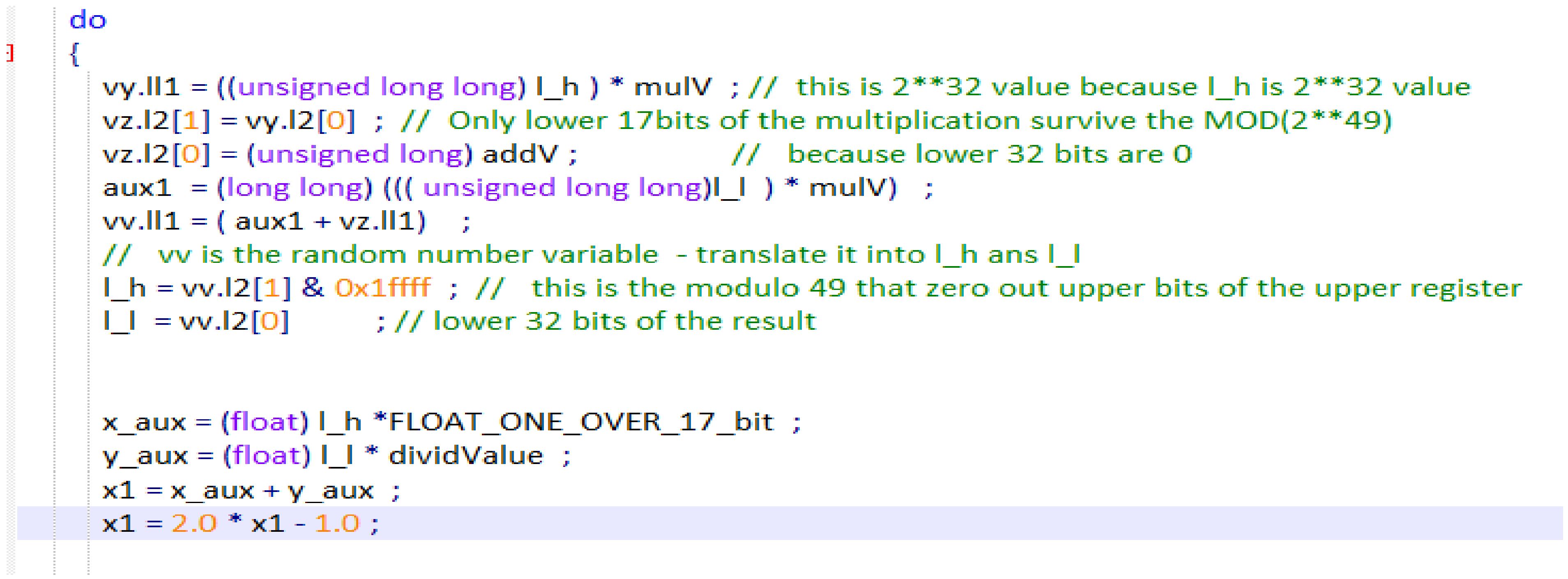


Software (1)

Monte Carlo Simulation Example for OpenCL Software
(ZIP, 28 KB) 37 views, 14 Oct 2015

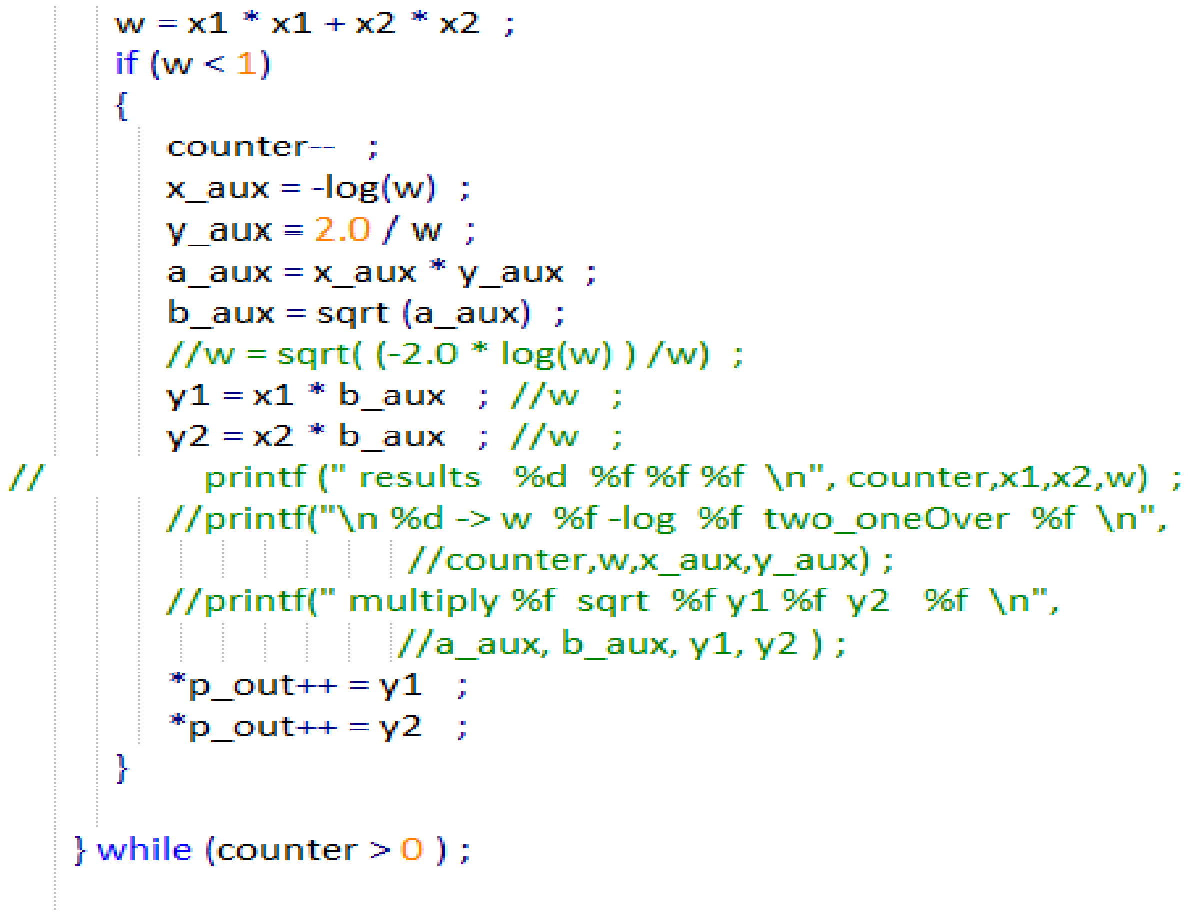
Name	Date modified	Type	Size
IMP NOTICE FOR REF DESIGNS.pdf	6/9/2015 12:26 PM	Adobe Acrobat D...	15 KB
monte_carlo_arm.tar	10/22/2015 8:38 AM	TAR File	23 KB
monte_carlo_opencl.tar	10/14/2015 12:39 ...	TAR File	57 KB

The Algorithm



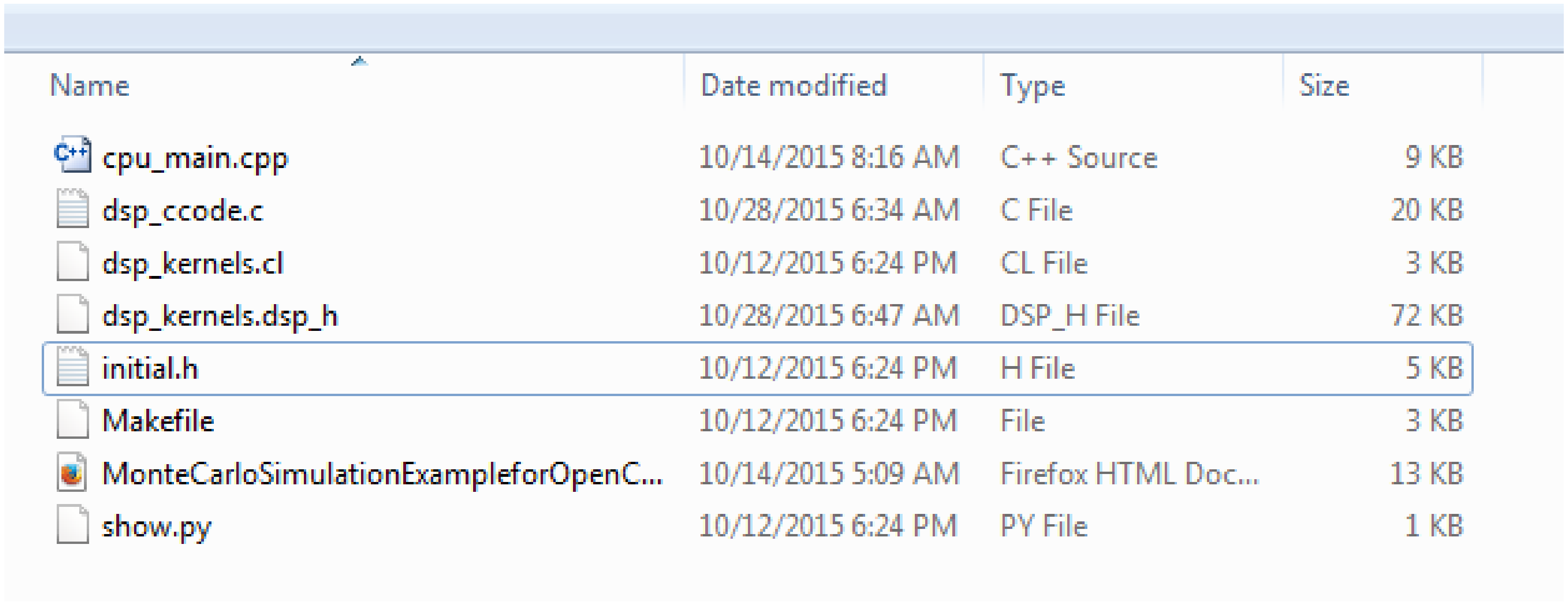
```
do
{
    vy.ll1 = ((unsigned long long) l_h ) * mulV  ; //  this is 2**32 value because l_h is 2**32 value
    vz.l2[1] = vy.l2[0]  ;  //  Only lower 17bits of the multiplication survive the MOD(2**49)
    vz.l2[0] = (unsigned long) addV ;                  //   because lower 32 bits are 0
    aux1  = (long long) ((( unsigned long long)l_l  ) * mulV)   ;
    vv.ll1 = ( aux1 + vz.ll1)    ;
    //   vv is the random number variable  - translate it into l_h ans l_l
    l_h = vv.l2[1] & 0x1ffff  ;  //   this is the modulo 49 that zero out upper bits of the upper register
    l_l  = vv.l2[0]          ; // lower 32 bits of the result


    x_aux = (float) l_h *FLOAT_ONE_OVER_17_bit  ;
    y_aux = (float) l_l * dividValue  ;
    x1 = x_aux + y_aux  ;
    x1 = 2.0 * x1 - 1.0 ;
```

The Algorithm



```
    w = x1 * x1 + x2 * x2  ;
    if (w < 1)
    {
        counter--   ;
        x_aux = -log(w)  ;
        y_aux = 2.0 / w  ;
        a_aux = x_aux * y_aux  ;
        b_aux = sqrt (a_aux)  ;
        //w = sqrt( (-2.0 * log(w) ) /w)  ;
        y1 = x1 * b_aux   ;  //w   ;
        y2 = x2 * b_aux   ;  //w   ;
//          printf (" results   %d  %f %f %f  \n", counter,x1,x2,w)  ;
        //printf("\n %d -> w  %f -log  %f  two_oneOver  %f  \n",
                    //counter,w,x_aux,y_aux) ;
        //printf(" multiply %f  sqrt  %f y1 %f  y2   %f  \n",
                    //a_aux, b_aux, y1, y2 ) ;
        *p_out++ = y1   ;
        *p_out++ = y2   ;
    }

} while (counter > 0 ) ;
```

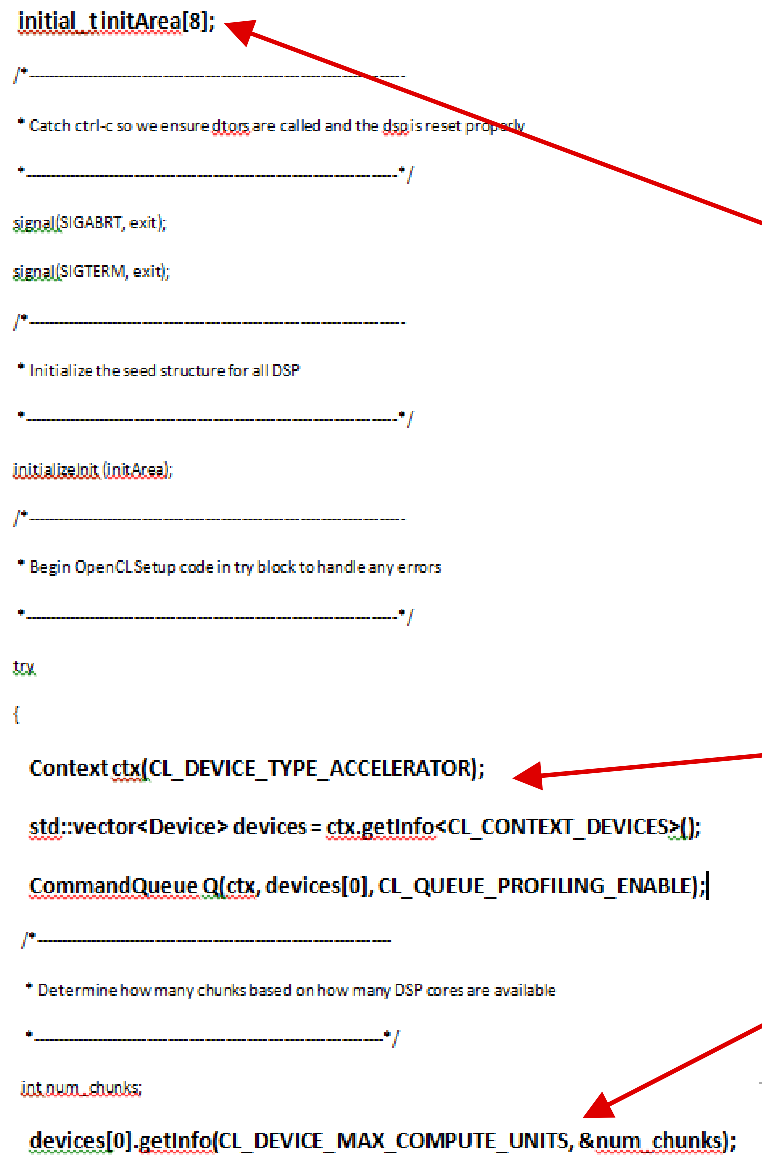
Code Walkthrough



Name	Date modified	Type	Size
cpu_main.cpp	10/14/2015 8:16 AM	C++ Source	9 KB
dsp_ccode.c	10/28/2015 6:34 AM	C File	20 KB
dsp_kernels.cl	10/12/2015 6:24 PM	CL File	3 KB
dsp_kernels.dsp_h	10/28/2015 6:47 AM	DSP_H File	72 KB
initial.h	10/12/2015 6:24 PM	H File	5 KB
Makefile	10/12/2015 6:24 PM	File	3 KB
MonteCarloSimulationExampleforOpenC...	10/14/2015 5:09 AM	Firefox HTML Doc...	13 KB
show.py	10/12/2015 6:24 PM	PY File	1 KB

Cpu_main.cpp

Context and host memory

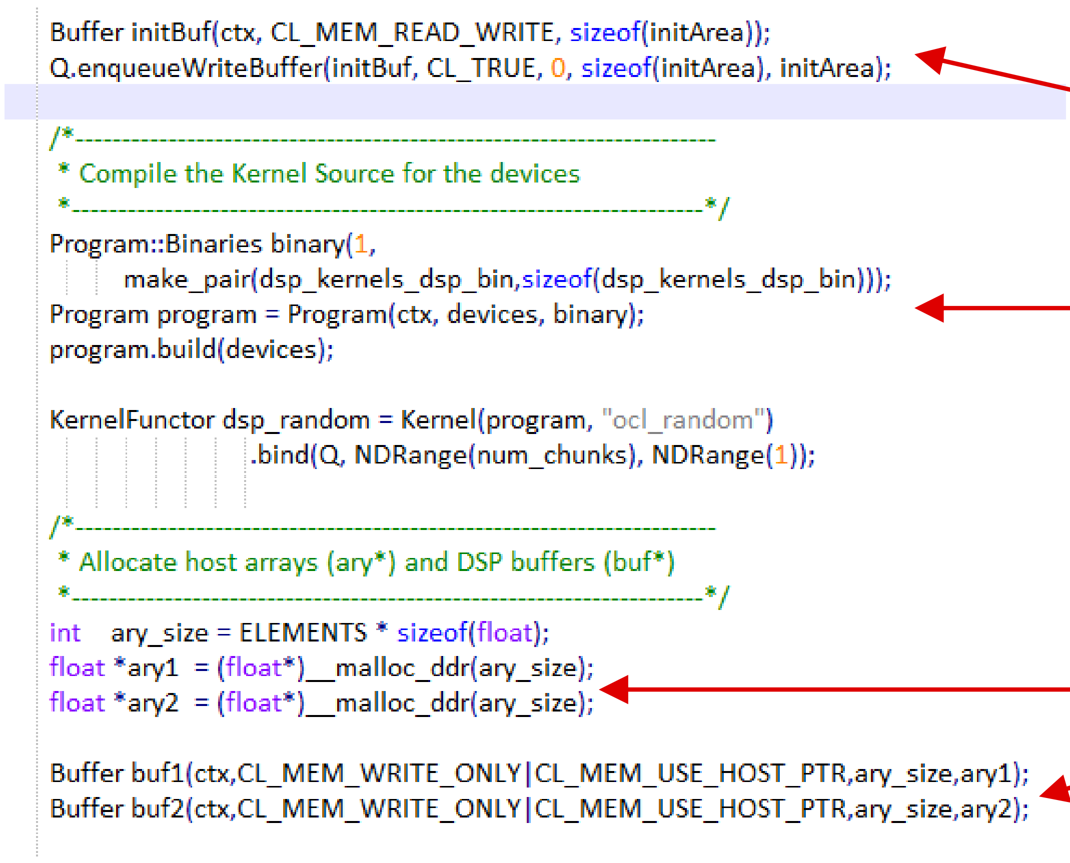


```
initial_t initArea[8];
/*-----------------------------------------------------------------
 * Catch ctrl-c so we ensure dtors are called and the dsp is reset properly
 *----------------------------------------------------------------*/
signal(SIGABRT, exit);
signal(SIGTERM, exit);
/*-----------------------------------------------------------------
 * Initialize the seed structure for all DSP
 *----------------------------------------------------------------*/
initializeInit (initArea);
/*-----------------------------------------------------------------
 * Begin OpenCLSetup code in try block to handle any errors
 *----------------------------------------------------------------*/
try
{
  Context ctx(CL_DEVICE_TYPE_ACCELERATOR);
  std::vector<Device> devices = ctx.getInfo<CL_CONTEXT_DEVICES>();
  CommandQueue Q(ctx, devices[0], CL_QUEUE_PROFILING_ENABLE);
 /*-----------------------------------------------------------------
 * Determine how many chunks based on how many DSP cores are available
 *----------------------------------------------------------------*/
 int num_chunks;
  devices[0].getInfo(CL_DEVICE_MAX_COMPUTE_UNITS, &num_chunks);
```

Host memory

Define context, device and queue

Cpu_main.cpp



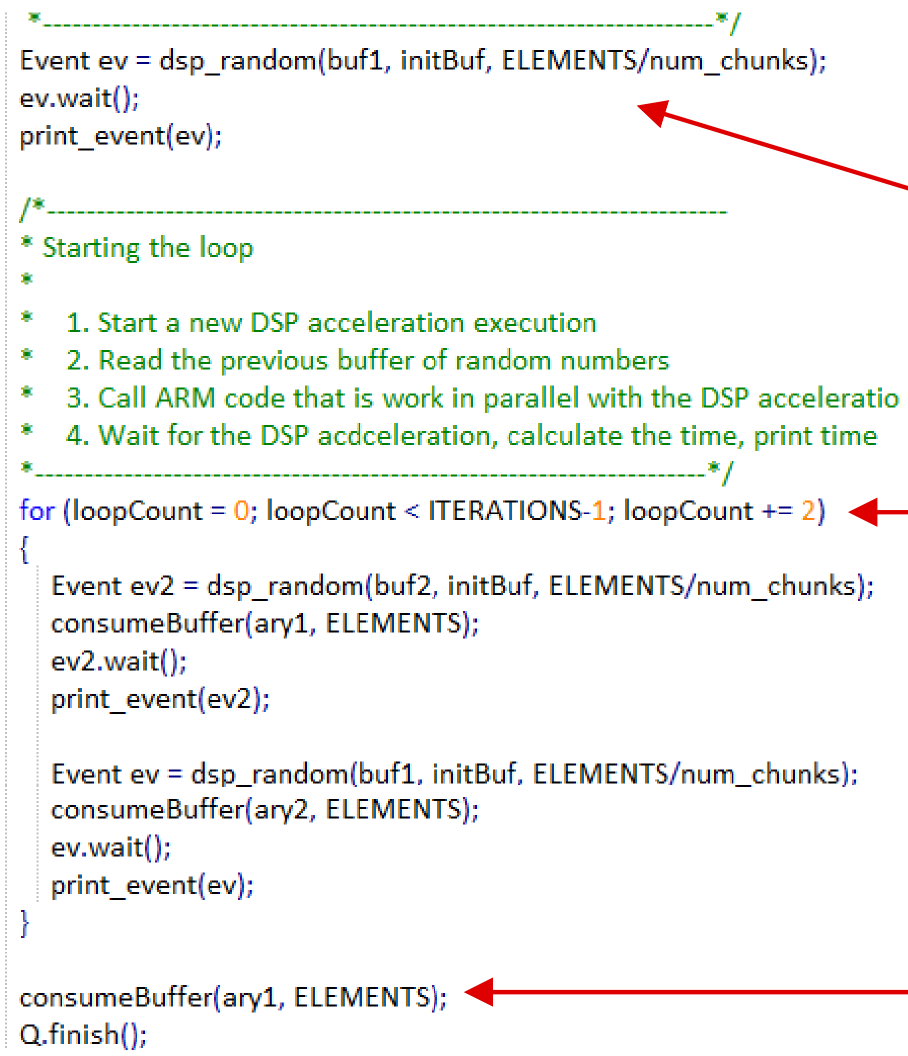
```
Buffer initBuf(ctx, CL_MEM_READ_WRITE, sizeof(initArea));
Q.enqueueWriteBuffer(initBuf, CL_TRUE, 0, sizeof(initArea), initArea);

/*-----------------------------------------------------------------------
 * Compile the Kernel Source for the devices
 *----------------------------------------------------------------------*/
Program::Binaries binary(1,
        make_pair(dsp_kernels_dsp_bin,sizeof(dsp_kernels_dsp_bin)));
Program program = Program(ctx, devices, binary);
program.build(devices);

KernelFunctor dsp_random = Kernel(program, "ocl_random")
                        .bind(Q, NDRange(num_chunks), NDRange(1));

/*-----------------------------------------------------------------------
 * Allocate host arrays (ary*) and DSP buffers (buf*)
 *----------------------------------------------------------------------*/
int    ary_size = ELEMENTS * sizeof(float);
float *ary1  = (float*)__malloc_ddr(ary_size);
float *ary2  = (float*)__malloc_ddr(ary_size);

Buffer buf1(ctx,CL_MEM_WRITE_ONLY|CL_MEM_USE_HOST_PTR,ary_size,ary1);
Buffer buf2(ctx,CL_MEM_WRITE_ONLY|CL_MEM_USE_HOST_PTR,ary_size,ary2);
```

Define a global buffer initBuf and copy the host buffer initArea into it.

Building a kernel from DSP binaries that were compiled from C code before .

Define buffers in the global memory that the DSP and the host can access. OpenCL takes care of coherency.

Cpu_main.cpp



```
 *-------------------------------------------------------------------*/
Event ev = dsp_random(buf1, initBuf, ELEMENTS/num_chunks);
ev.wait();
print_event(ev);

/*-------------------------------------------------------------------
* Starting the loop
*
*    1. Start a new DSP acceleration execution
*    2. Read the previous buffer of random numbers
*    3. Call ARM code that is work in parallel with the DSP acceleratio
*    4. Wait for the DSP acdceleration, calculate the time, print time
*-------------------------------------------------------------------*/
for (loopCount = 0; loopCount < ITERATIONS-1; loopCount += 2)
{
    Event ev2 = dsp_random(buf2, initBuf, ELEMENTS/num_chunks);
    consumeBuffer(ary1, ELEMENTS);
    ev2.wait();
    print_event(ev2);

    Event ev = dsp_random(buf1, initBuf, ELEMENTS/num_chunks);
    consumeBuffer(ary2, ELEMENTS);
    ev.wait();
    print_event(ev);
}

consumeBuffer(ary1, ELEMENTS);
Q.finish();
```

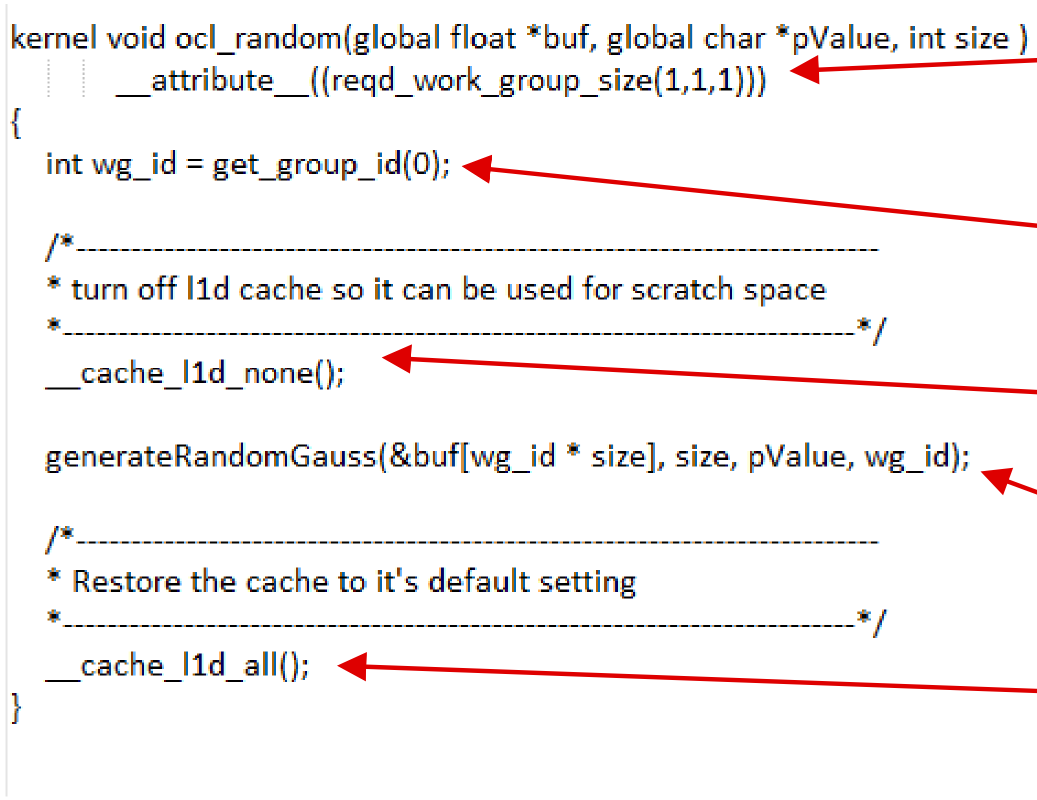
- dsp_random is the kernel that was built previously.
- The code associates it with event.
- ev.wait() waits until the kernel is done.
- The output is in buf1 (global memory that host can access).

The loop uses an explicit ping-pong buffer:

1. Start the kernel with buf2.
2. Process the data in buf1.
3. Wait until the kernel is done.
4. Start the kernel with buf1.
5. Process the data in buf2.
6. Wait until the kernel is done.

Process the last buffer (buf1).

The OpenCL Kernel Code: Dsp_kernel.cl



```
kernel void ocl_random(global float *buf, global char *pValue, int size )
        __attribute__((reqd_work_group_size(1,1,1)))
{
  int wg_id = get_group_id(0);

  /*-------------------------------------------------------------------------
  * turn off l1d cache so it can be used for scratch space
  *-----------------------------------------------------------------------*/
  __cache_l1d_none();

  generateRandomGauss(&buf[wg_id * size], size, pValue, wg_id);

  /*-------------------------------------------------------------------------
  * Restore the cache to it's default setting
  *-----------------------------------------------------------------------*/
  __cache_l1d_all();
}
```

Set the work group size attributes.
Since the C code generates fix number of values in each call, the 1,1,1, values tell the compiler not to try and break the work into multiple work items (work group).

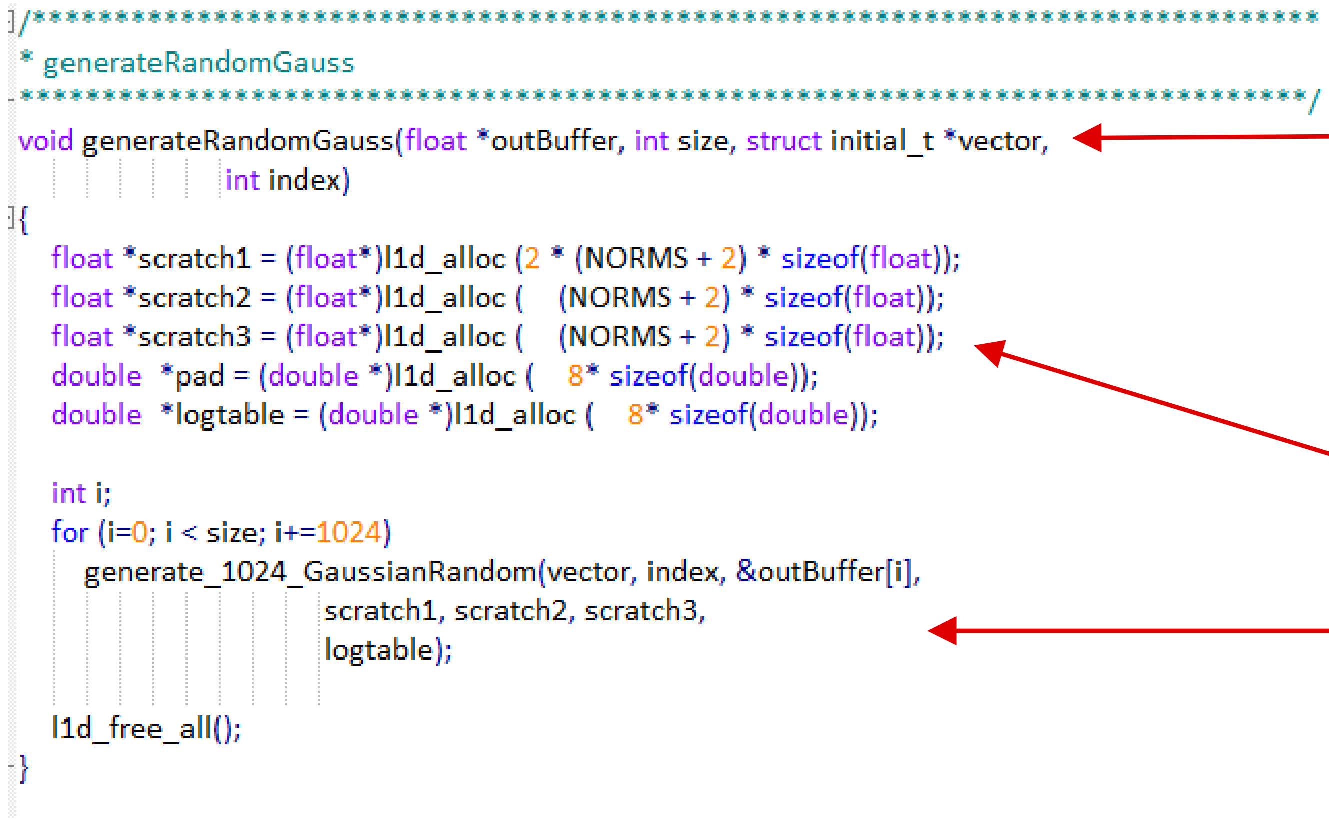
Each DSP gets a single work group. The ID is either 0 or 1 (in AM572 case).

L1D is used as SRAM for intermediate results storage.

Calling a C routine with four arguments (using a standard C call convention).

Return L1D to the original setting.

Dsp_ccode.c



```
/*****************************************************************************
* generateRandomGauss
*****************************************************************************/
void generateRandomGauss(float *outBuffer, int size, struct initial_t *vector,
                         int index)
{
    float *scratch1 = (float*)l1d_alloc (2 * (NORMS + 2) * sizeof(float));
    float *scratch2 = (float*)l1d_alloc (    (NORMS + 2) * sizeof(float));
    float *scratch3 = (float*)l1d_alloc (    (NORMS + 2) * sizeof(float));
    double  *pad = (double *)l1d_alloc (    8* sizeof(double));
    double  *logtable = (double *)l1d_alloc (    8* sizeof(double));

    int i;
    for (i=0; i < size; i+=1024)
        generate_1024_GaussianRandom(vector, index, &outBuffer[i],
                                     scratch1, scratch2, scratch3,
                                     logtable);

    l1d_free_all();
}
```

Standard DSP C function. The file with all the include files and the sub-routines is compiled by the same Makefile that compiles main.cpp, with the DSP code generator compiler.

L1D is used as SRAM and 5 buffers are allocated in L1D. The number of elements in each call of the loop were chosen so that the buffers fit into 32KB L1D.

For More Information

- TI Design: Monte-Carlo Simulation on AM57x Using OpenCL for DSP Acceleration Reference Design
  http://www.ti.com/tool/TIDEP0046

- Processor SDK Product Page
  http://www.ti.com/lsds/ti/tools-software/processor_sw.page

- Processor SDK Training Series
  http://training.ti.com/processor-sdk-training-series

- TI OpenCL Wiki: http://processors.wiki.ti.com/index.php/OpenCL

- For questions regarding topics covered in this training, visit the support forums at the TI E2E Community website.