Raman Research Institute
Bangalore

A Report On

Development of Signal Processing Algorithms using OpenCL for Viretx 7 FPGA

Submitted By

Pradeep Singh

pdeepsingh91@gmail.com / pradeepsingh7890@live.com

CONTENTS

1. ABSTRACT

OpenCL™ is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software.

In this report, an attempt is made to understand the OpenCL interface to the new Xilinx Vivado Design Suite which uses Xilinx's own implementation of the OpenCL specification.

The report is organised as follows. In the first section, a brief introduction to OpenCL is presented with more preference to the host and kernel programming aspect of OpenCL by focussing on the various new data structures and functions used in OpenCL.

Then the use of OpenCL in the field of FPGA is discussed with respect to Altera and Xilinx. Xilinx Vivado HLS is then used to run the various codes like Vector Addition, N-Point DFT and the 8-Point FFT.

We have also mentioned the problems and errors that we have faced during this project along with the solutions. And, we have also tried to give a path for the future, what can be done and what are the possible steps/measure can be taken in order to move this project further.

The performance estimates of the synthesized RTL codes are used to compare with those of built-in Xilinx FFT IP core using the Vivado Design Suite.

2. INTRODUCTION TO OpenCL

OpenCL defines a single consistent programming model based on C99 and a system level abstraction for parallel computing (using task and data parallelism) of heterogeneous platforms that consist of devices like CPUs, GPUs, DSPs, FPGAs etc. from various vendors.

The OpenCL programming standard is maintained by the Khronos Group, a not-for-profit industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices.

OpenCL has three distinctive advantages over other forms of programming namely

● Portability - Vendors who provide OpenCL compliant hardware also provide their own compilers. This enables the use of a single piece of OpenCL code to be run on multiple devices from different vendors.

● Vector Processing – Usually vector processing of various platforms are vendor specific like SSE, PTX and Altivec. But OpenCL overcomes this disadvantage by standardizing the vector processing.

● Parallel Programming – The concept of Workgroups and Work items have given a new dimension to parallel programming with the scope for full-task parallelism.

● Generally, there are several benefits for software developers and system designers to use OpenCL to develop code for FPGAs:

- **Simplicity and ease of development**: Most software developers are familiar with the C programming language, but not low-level HDL languages. OpenCL keeps you at a higher level of programming, making your system open to more software developers.

- **Code profiling**: Using OpenCL, you can profile your code and determine the performance-sensitive pieces that could be hardware accelerated as kernels in an FPGA.

- **Performance**: Performance per watt is the ultimate goal of system design. Using an FPGA, you’re balancing high performance in an energy-efficient solution.

- **Efficiency**: The FPGA has a fine-grain parallelism architecture, and by using OpenCL you can generate only the logic you need to deliver one fifth of the power of the hardware alternatives.

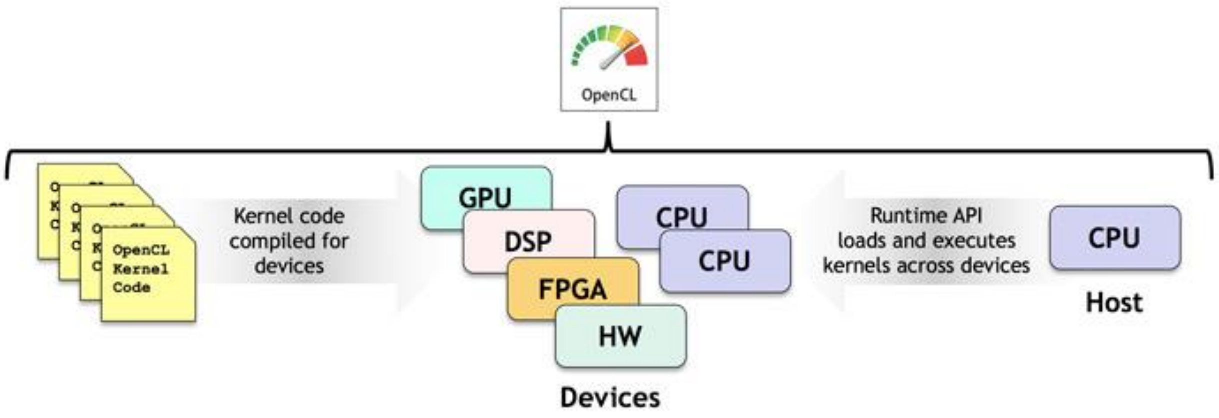
- **Heterogeneous systems**: With OpenCL, you can develop kernels that target FPGAs, CPUs, GPUs, and DSPs seamlessly to give you a truly heterogeneous system design.

- **Code reuse**: The holy grail of software development is achieving code reuse. Code reuse is often an elusive goal for software developers and system designers. OpenCL kernels allow for portable code that you can target for different families and generations of FPGAs from one project to the next, extending the life of your code.

Today, OpenCL is developed and maintained by the technology consortium Khronos Group. Most FPGA manufacturers provide Software Development Kits (SDKs) for OpenCL development on FPGAs.

OpenCL – Portable Heterogeneous Computing

- OpenCL = Two APIs and Two Kernel languages
  - C Platform Layer API to query, select and initialize compute devices
  - OpenCL C and (soon) OpenCL C++ kernel languages to write parallel code
  - C Runtime API to build and execute kernels across multiple devices.
- One code tree can be executed on CPUs, GPUs, DSPs, FPGA and hardware
  - Dynamically balance work across available processors


OpenCL
OpenCL Kernel Code
Kernel code compiled for devices
GPU
DSP
FPGA
HW
CPU
CPU
Devices
Runtime API loads and executes kernels across devices
CPU
Host

2.1 OpenCL Memory Model:-

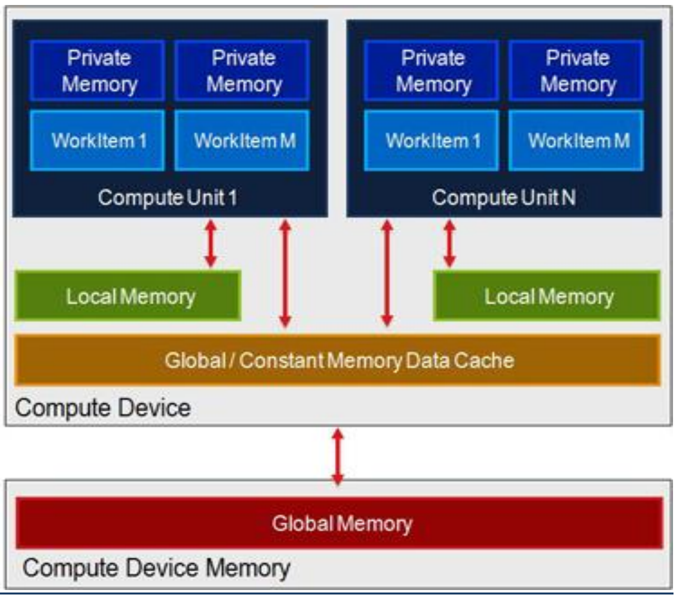


Figure 2.1 - Block Diagram of the memory model used in OpenCL

OpenCL memory hierarchy is comprised of five parts as depicted in figure 1.1.

a. **Host Memory:** - Memory region which is visible and accessible only to the host processor. The host processor has full control of this memory space and can read and write from this space without any restrictions. Kernels cannot access data in this memory space. Any data to be accessed must be transferred to the global memory for the compute units to access them.

b. **Global Memory:** - This memory space is accessible to both the host and devices. Host is responsible for allocation and deallocation of memory space. Initially the host transfers data from the host memory to global memory. When a kernel is launched to access the data, host loses its access rights and the device takes over. After processing the data, the control is returned back to the host which then transfers back the data to host memory and deallocates the memory buffer.

c. **Constant Memory:** - This memory space provides both read and write access to the host whereas providing only read access to the devices. This space is mostly utilized to provide constant data required for kernel computation.

d. **Local Memory:**- Memory region which is visible and accessible only to the device. The host processor has no read and write access control; allows read and write access to all the compute units within a device. This type of memory is used to share data between multiple compute units. Work items can access local memory 100x faster compared to global/constant memory.

e. **Private Memory:** - This space is accessible by particular processing element of a device. Multiple work items which belong to same processing element can obtain the read and write access. This is the memory space with the least access time and the lowest memory size. The Access speed is far greater than local memory and global/constant memory.

In the context of FPGA the memory model refers to the following parameters

i. **Host Memory:** - Any memory that is directly connected to the host processor.

ii. **Global Memory:** - Any external memory device connected to the FPGA through a physical interface.

iii. **Local Memory:** - Memory within the FPGA mostly RAM blocks.

iv. **Private Memory:** - Memory within the FPGA created by utilizing the registers in order to minimize the latency.

2.2 OpenCL Programming Model:-

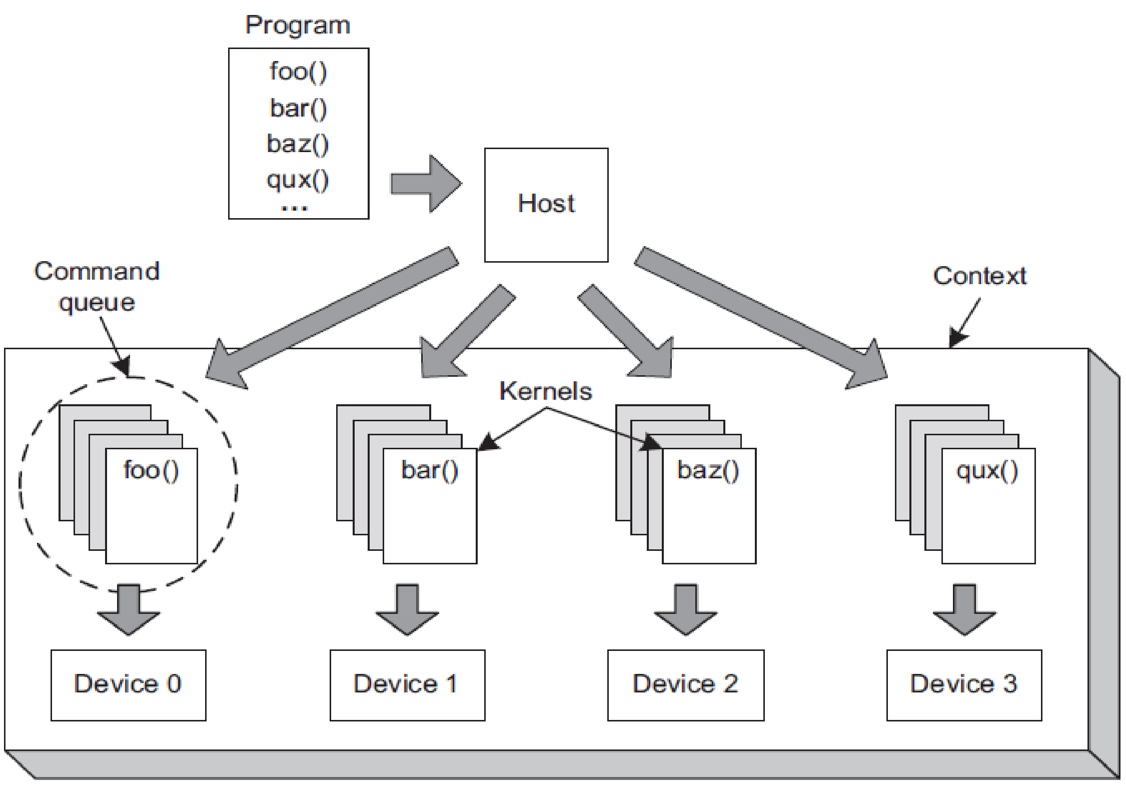


Figure 2.2 - Programming Model used in OpenCL

OpenCL programming model consists of two main components with their own coding patterns namely Host (Host Programming) and Device (Kernel Programming).

1.HOST - It is the main CPU to which other devices are connected and the main program is executed. The host can call the devices to perform various computations from the host code using kernels. It can be connected to multiple devices as depicted in fig 1.2. Host code is generally C/C++ code, only the time critical part of the code is implemented using OpenCL in the form of multiple kernels which run on the devices.

2.DEVICE – They are mostly GPUs, FPGAs which are used to execute kernels sent by the host. Each kernel has a specific task which is broken down into a set of parallel tasks to be implemented on compute devices as shown in fig 1.2.

The following is the list of identifiers used in OpenCL

Platform	vendor specific OpenCL implementation
Device	set of devices that belong to a platform
Context	group of devices selected to work together
Kernel	function to be executed on the evice
Command Queue	specifies kernel execution order to the devices
Work Item	single instance of a kernel

Table 2.1

2.2.1 Host Programming

Host Programming in OpenCL makes use of six new data structures namely platforms, devices, contexts, programs, kernels, and command queues.

- **Accessing Platforms**

Function Signature:

*cl_int clGetPlatformIDs (cl_uint num_entries, cl_platform_id *platforms, cl_uint *num_platforms)*

a. *platforms* places platform ids in the platforms' memory reference.
b. *num_platforms* is the reference for number of available platforms.
c. *num_entries* is maximum number of platforms user is interested in detecting.

- **Obtaining Platform Info**

Function Signature:

*cl_int clGetPlatformInfo (cl_platform_id platform, cl_platform_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)*

a. *param_value* - where the parameter value is stored.
b. *param_name* - desired parameter name
c. *param_value_size* - size of the value of the parameter

- **Accessing Installed Devices**

Function Signature:

*cl_int clGetDeviceIDs (cl_platform_id platform, cl_device_type device_type, cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)*

a. *devices* constructs cl_device_id structure to represent a device.
OpenCL device types include the following

Device Type	Information
CL_DEVICE_TYPE_ALL	Identifies all devices associated with platform
CL_DEVICE_TYPE_DEFAULT	Identifies devices associated with default type
CL_DEVICE_TYPE_CPU	Identifies the host processor
CL_DEVICE_TYPE_GPU	Identifies a device containing a GPU
CL_DEVICE_TYPE_ACCELERATOR	Identifies an external accelerator device

- **Obtaining Device Information**

Function Signature:

*cl_int clGetDeviceInfo (cl_device_id device, cl_device_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)*

The list of parameters that could be obtained are as follows.

Parameter	Purpose
CL_DEVICE_NAME	Returns the name of the device
CL_DEVICE_VENDOR	Returns the device’s vendor
CL_DEVICE_EXTENSIONS	Returns the device’s supported extensions
CL_DEVICE_GLOBAL_MEM_SIZE	Returns the size of the device’s global memory
CL_DEVICE_ADDRESS_BITS	Returns the size of the device’s address space
CL_DEVICE_AVAILABLE	Returns whether the device is available
CL_DEVICE_COMPILER_AVAILABLE	Returns whether the implementation provides a device compiler

- **Exploring Contexts**

In OpenCL a context refers to a set of devices that are selected to work together. Contexts enable the creation of command queues which allow hosts to send kernel to the devices. Creating contexts using devices of different platforms is not supported.

The following figure represents the concept of contexts in OpenCL.

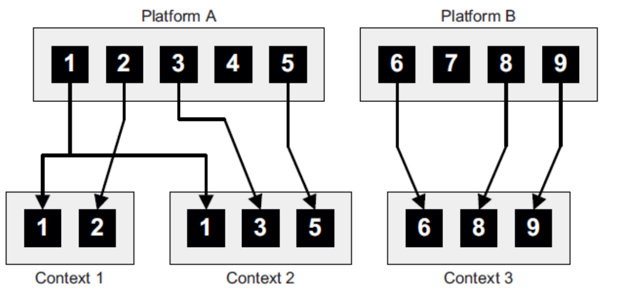


Figure 1.3 - Contexts in OpenCL

- **Creating Contexts**

Function Signature:

*cl_context clCreateContext (const cl_context_properties *properties, cl_uint num_devices, const cl_device_id *devices, (void CL_CALLBACK *notify_func)(...), void *user_data, cl_int *error)*

a. *cl_context_properties* – must identify an array of names and values whose last element must be zero

b. CL_CALLBACK accepts a call-back function as an argument which is invoked when error occurs during context operations.

• Obtaining Context Information

Function Signature:

clGetContextInfo (cl_context context, cl_context_info param_name, size_t param_value_size, void param_value, size_t *param_value_size_ret)*

The list of parameters that can be obtained are as follows.

Parameter	Purpose
CL_CONTEXT_NUM_DEVICES	Returns the number of devices in the context
CL_CONTEXT_DEVICES	Returns the devices in the context
CL_CONTEXT_PROPERTIES	Returns the property array associated with context
CL_CONTEXT_REFERENCE_COUNT	Returns the reference count of the context
CL_CONTEXT_D3D10_PREFER _SHARED_RESOURCES_KHR	Returns whether Direct3D shared resources will be accelerated more than unshared resources

• Creating Programs

OpenCL supports creating programs using both the source and the binary files using the following two commands respectively.

1. *clCreateProgramWithSource*
2. *clCreateProgramWithBinary*

Function Signature:

*clCreateProgramWithSource (cl_context context, cl_uint src_num, const char **src_strings, const size_t *src_sizes, cl_int *err_code)*

a. *src_num* – number of strings to expect
b. *src_sizes* – identifies size of each string

Steps to create cl_program from text file

i. Determine size of kernel.cl.
ii. Read the file content into a buffer.
iii. Use the buffer to create *cl_program*.

Function Signature:

*clCreateProgramWithBinary (cl_context context, cl_uint num_devices, const cl_device_id *devices, const size_t *bin_sizes, const unsigned char **bins, cl_int *bin_status, cl_int *err_code)*

The functionality is similar to clCreateProgramWithSource except that it sources from a binary file rather than text files.

● Building Programs
Function Signature:

*clBuildProgram (cl_program program, cl_uint num_devices, const cl_device_id *devices, const char *options, (void CL_CALLBACK *notify_func)(...), void *user_data)*

a. The function compiles and links a *cl_program* for devices associated with the platform.

● Obtaining Program Information
Function Signature:

*clGetProgramInfo (cl_program program, cl_program_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)*

a. *cl_program_info* is an enumerated data type which includes CL_PROGRAM_SOURCE that concatenates all of the program source buffer into one string that contains all of kernel functions.

b. When receiving errors, it is advisable to go through this file. Only way to understand what occurred during program build process.

c. Dynamic memory allocation for the log is preferred.

● Creating Kernels (Constructing cl_kernel from cl_program)
Function Signature:

*clCreateKernelsInProgram (cl_program program, cl_uint num_kernels, cl_kernel *kernels, cl_uint *num_kernels_ret)*

a. New *cl_kernel*s are placed in *kernels* array.
b. *num_kernels_ret* – number of available kernels
c. For single kernels *clCreateKernel* function could be used.

Function Signature:

*clCreateKernel (cl_program program, const char *kernel_name, cl_int *error)*

● Obtaining Kernel Information
Function Signature:

*clGetKernelInfo (cl_kernel kernel, cl_kernel_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)*

The parameters that can be obtained are as follows.

Parameter	Purpose
CL_KERNEL_FUNCTION_NAME	Returns the name of the function from which kernel was formed
CL_KERNEL_NUM_ARGS	Returns the number of input arguments accepted by the kernel's associated function
CL_KERNEL_REFERENCE_COUNT	Returns the number of times the kernel has been referenced

CL_KERNEL_CONTEXT	Returns the context associated with the kernel
CL_KERNEL_PROGRAM	Returns the program from which the kernel was created

- Collecting Kernels in Command Queue

Command is a message sent from the host which tells the device to perform certain operations. Data transfer operations may convey data to and from device, but commands in command queue move in only one direction.

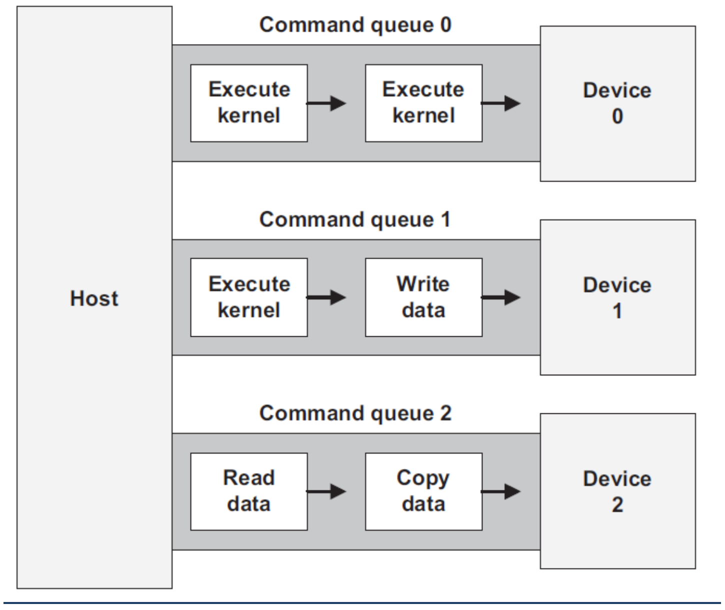


Figure 1.4 - Representation of Command Queues in OpenCL

- Creating Command Queue

Function Signature:

*clCreateCommandQueue(cl_context context, cl_device_id device, cl_command_queue_properties properties, cl_int *err)*

a. Returns a *cl_command_queue* whose reference count can be increased or decreased by *clRetainCommandQueue* and *clReleaseCommanQueue*
b. Normally executes in FIFO order.

c. For out of order execution we could use CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

- Enqueuing Kernel Execution commands

Function Signature:

*clEnqueueTask(cl_command_queue queue, cl_kernel kernel, cl_uint num_events, const cl_event *wait_list, cl_event *event)*

a. cl_command_queue - sends command to a specific device.
b. cl_kernel - contains the OpenCL function to be executed.

When this function is called, kernel execution function is sent to command queue, device will execute the kernel once it processes the command. No separate function to execute the kernel is required.

- Setting Kernel Arguments

Function Signature

*clSetKernelArg (cl_kernel kernel, cl_uint index, size_t size, const void *value)*

a. index - order of the kernel argument
b. *value - pointer to the data that will be transferred to the kernel function.

This can take the following forms:

i.Pointer to primitive data - To Transfer simple primitives to the device.
ii.Pointer to memory object - Transfer complex data.
iii.Pointer to sampler object - Transfers an object that defines how images are read.
iv.NULL - tells the device to allocate a local memory space for kernel argument, No kernel argument is received from the host.

- Creating Buffer Objects

Buffer objects package data.

Function Signature

*clCreateBuffer (cl_context context, cl_mem_flags options, size_t size, void *host_ptr, cl_int *error)*

a. returns a cl_mem that wraps around the data identified b host_ptr argument.
b. The options parameter configure the object characteristics such as whether buffer data is read only or write only.

- Creating Sub buffer Objects
  Function Signature

clCreateSubBuffer (cl_mem buffer,
cl_mem_flags flags, cl_buffer_create_type type,
*Const void *info, cl_int *error)*

a. Used when one kernel has to subset of data required by other kernel.

We avoid the discussion of image objects as it is beyond the scope of this project.

- Obtaining Information about Buffer Objects
  Function Signature

clGetMemObjectInfo (cl_mem object, cl_mem_info param_name,
*size_t param_value_size, void *param_value,*
*size_t *param_value_size_ret)*

The first three arguments provide input:
a. The memory object, a name that identifies the type of data we are requesting, and the amount of data we are requesting.
 The last two arguments are output arguments, in which the function returns the data we are requesting and the size of the returned data.

- Read/Write Data Transfer

EnqueueReadBuffer(cl_command queue queue, _mem buffer, cl_bool blocking, ze_t offset,	eads data from a buffer bject to host memory

ze_t data_size, void *ptr, cl_uint um_events, onst cl_event *wait_list, cl_event event)	
EnqueueWriteBuffer(cl_comman _queue_queue, _mem buffer, cl_bool blocking, ze_t offset, ze_t data_size, const void *ptr, _uint num_events, const _event *wait_list, _event *event)	/rites data from host nemory to a buffer bject
EnqueueReadImage(cl_comman _queue queue, _mem image, cl_bool blocking, onst size_t origin[3], const size_t egion[3], ze_t row_pitch, size_t ice_pitch, oid *ptr, cl_uint num_events, onst cl_event *wait_list, cl_event event)	eads data from an nage object to host nemory
EnqueueWriteImage(cl_comman _queue queue, _mem image, cl_bool blocking, onst size_t origin[3], const size_t egion[3], ze_t row_pitch, size_t ice_pitch, onst void * ptr, cl_uint um_events, onst cl_event *event_wait_list, _event *event)	/rites data from host nemory to an image bject

EnqueueReadBufferRect(cl_com nand_queue_queue, _mem buffer, cl_bool blocking, onst size_t buffer_origin[3], onst size_t host_origin[3], onst size_t region[3], size_t uffer_row_pitch, ze_t buffer_slice_pitch, size_t ost_row_pitch, ze_t host_slice_pitch, void *ptr, _uint num_events, const _event *wait_list, _event *event)	eads a rectangular portion f data from a buffer bject to host memory
EnqueueWriteBufferRect(cl_com nand_queue queue, _mem buffer, cl_bool blocking, onst size_t buffer_origin[3], onst size_t host_origin[3], onst size_t region[3], size_t uffer_row_pitch, ze_t buffer_slice_pitch, size_t ost_row_pitch, ze_t host_slice_pitch, void *ptr, _uint num_events, const _event *wait_list, _event *event)	/rites a rectangular portion f data from host nemory to a buffer bject

● Mapping Memory Objects
Instead of using the read/write operations presented earlier, we can map a memory object on a device to a memory region on the host. Once this map is established, we can read or modify the memory object on the host using pointers or other memory operations.
Commands such as clEnqueueMapBuffer, clEnqueueMapImage help on this regard.

● Data Partitioning
Enables better distribution of processing load on various computing devices.
Function Signature

clEnqueueNDRangeKernel(cl_command_queue queue, cl_kernel kernel,
*cl_uint work_dims, const size_t *global_work_offset,*
*const size_t *global_work_size, const size_t *local_work_size,*
*cl_uint num_events, const cl_event *wait_list, cl_event *event)*

a. work_dimswork_dims—The number of dimensions in the data
b. global_work_offset—The global ID offsets in each dimension
c. global_work_size—The number of work-items in each dimension
d. local_work_size—The number of work-items in a work-group, in each dimension

Now that we have discussed the various aspects of host programming, let us consider the kernel programming part.

2.2.2 Kernel Programming

Some of the basic rules for Kernel Programming are :-

a. Every Kernel declaration must start with __kernel.

b. Every Kernel must return void.

c. All pointers passed to a kernel must be preceded by an address space qualifier like
__global, __constant, __local, __private. This convention is necessary in allocating the memory space as discussed earlier in OpenCL memory model section.

d. All data types are similar to C/C++, except for accessing double data type, for which we should load the extension cl_khr_fp64

● Scalar Data types

Datatype	Purpose
bool	A Boolean condition: true (1) or false (0)
char	Signed two's complement 8-bit integer
unsigned char/ uchar	Unsigned two's complement 8-bit integer

short	Signed two’s complement 16-bit integer
unsigned short/ ushort	Unsigned two’s complement 16-bit integer
int Signed	two’s complement 32-bit integer
unsigned int/ uint Unsigned	two’s complement 32-bit integer
long Signed	two’s complement 64-bit integer
unsigned long/ ulong Unsigned	two’s complement 64-bit integer
half 16-bit	floating-point value, IEEE-754-2008 conformant
float 32-bit	floating-point value, IEEE-754 conformant
intptr_t	Signed integer to which a void pointer can be converted
uintptr_t	Unsigned integer to which a void pointer can be converted
ptrdiff_t	Signed integer produced by pointer subtraction
size_t	Unsigned integer produced by the size of operator
void	Untyped data

- Byte Order

Two ways the determine whether the byte order is little endian or big endian

i. calling clGetDeviceInfo with param value CL_DEVICE_ENDAIN_LITTLE in the host code.

ii. using #if def __ENDIAN_LITTLE__ in the kernel program.

- Vector Data types

OpenCL kernels support vector data types which contain multiple elements of the same type. Each time an operation is performed on a vector, all the elements of the vector are operated on simultaneously.

The following list depicts the vector types supported in OpenCL standard 1.0

Vector type	Description
charn	Vector containing n 8-bit signed two’s complement integers
ucharn	Vector containing n 8-bit unsigned two’s complement integers
shortn	Vector containing n 16-bit signed two’s complement integers
ushortn	Vector containing n 16-bit unsigned two’s complement integers

intn	Vector containing n 32-bit signed two’s complement integers
uintn	Vector containing n 32-bit unsigned two’s complement integers
longn	Vector containing n 64-bit signed two’s complement integers
ulongn	Vector containing n 64-bit unsigned two’s complement integers
floatn	Vector containing n 32-bit single-precision floating-point values

3. OpenCL in FPGAs

In recent years, processor manufacturers have hit a hardware roadblock, wherein the scope for increase in operating frequencies has reduced considerably. This has meant that several manufacturers like Intel are moving towards adding multiple cores and enhanced instruction sets. The parallel computing speeds up the program execution without requiring faster clock frequencies.

Current trend is the execution of code chunks called threads on multiple cores to achieve parallelism. Several companies run their software on GPUs, when they should be using FPGAs instead; and at the same time, others stick to FPGAs and ignore GPUs completely. The main reason is that converting CUDA to VHDL, or Verilog to CPU intrinsic, is really time consuming. Another reason can be seen in the amount of investment put on a certain technology.

One of the solutions to these predicaments is Open Computing Language i.e. OpenCL framework enables us to write a single program that runs across heterogeneous platforms like CPUs, GPUs, FPGAs, DSPs, and other processors. OpenCL allows parallel computing through task based and data based parallelism.

The main topic of discussion is why FPGAs should be programmed using OpenCL?
FPGAs are inherently parallel, perfectly fitting the capabilities of OpenCL. In addition to this the pipelining concepts of FPGA provides an alternative addition to the traditional data and task parallelism. The major advantage of OpenCL in FPGAs is the level of abstraction offered by OpenCL, which provides a language similar to C/C++ to program FPGAs.This allows any software programmer to work with FPGAs without any knowledge about the low level HDL coding practises of FPGA designers.

Most the points stacked in favour of OpenCL compared to other languages have been discussed in the Introduction section.

With all the advantages discussed above let us discuss how different vendors have embraced OpenCL.

Altera have released an exclusive SDK for OpenCL. The Altera SDK for OpenCL is in full production release, making Altera the first FPGA Company to have a solution that conforms to the OpenCL specification. The Altera SDK for OpenCL supports a variety of host CPUs, including the embedded ARM® Cortex®-A9 processor cores in SoC devices, the IBM Power Series processors, and a standard x86 CPU. The Altera SDK for OpenCL supports scalable solutions on multiple FPGAs and multiple boards as well as a variety of memory targets, such as DDR SDRAM for sequential memory accesses, QDR SRAM for random memory accesses, or internal FPGA memory for low-latency memory access. Half-precision as well as single- and double-precision floating point is also supported.

Xilinx Vivado HLS suite version 201   4.3 and higher have support for OpenCL. The latest release from Xilinx is the SdAccel design suite which solely focuses on OpenCL framework implementation on FPGAs manufactured by them.

It is important to note each of the FPGA vendors like Xilinx, Altera have customized the host code of the OpenCL implementation in order to provide an easier alternative coding style for their customers. These customizations are basically, abstraction functions which perform the tasks of multiple lines of code written in the OpenCL standard host code.

We could consider a snippet of Xilinx host code below to execute a kernel
hls_run_kernel ("vadd", a, LENGTH, b, LENGTH, hw_c, LENGTH);

This API has the following signature:
*void hls_run_kernel(const char *KernelName, ScalarType0 *Arg0, int size0, ScalarType1 *Arg1, int size1, …)*

Where:
- Arg0 is the first argument of the OpenCL API C kernel.
- ScalarType0 is the type of the first argument of the OpenCL API C kernel.
- Size0 is the number of data samples to be read or written.

Kernel coding aspect in FPGA resembles the OpenCL standard discussed earlier.

4. METHODOLOGY

Our goal was to develop Signal Processing Algorithms using OpenCL on Viretx 7 FPGAs. Since, FFT and DFT are the most basic and robust Algorithms to develop and to get started, we decided to test OpenCL and Virtex 7 with FFT& DFT.

With the implementation of FFT in mind, we started out with a bottom up approach and took the following steps:-

a. We familiarized ourselves with the concept of host and kernel programming through various sources which are indexed in the References.

b. We started with the implementation of 16 bit Vector addition kernel, an example project on VIVADO HLS 2015.1.

c. Once we were comfortable with the Vivado HLS & Vivado software, steps that need to be taken to compile, synthesis, co-simulation and implementation, we started writing our own code for DFT.

d. Since, our project was all about FFT & DFT, we also made our self-familiar with DFTs and FFTs. And, also the various algorithm's to compute DFT.

e. We started with Two point DFT and were able to develop and simulate the 2 point DFT and also compared the result.

f. For the next stage we wished to explore the parallel programming capabilities of OpenCL with the implementation of N point DFT algorithm.

g. We implemented a single dimension kernel with work group size 2*N capable of computing both the real and imaginary terms of the DFT output. We simulated the code for 1024 point complex input DFT and also verified the results.

h. Our next step was to implement FFT Algorithm. We started with taking a small 2 point FFT. And, later implemented an 8 point FFT.

i. The final step was the comparison of results between the Xilinx Radix-2 FFT IP Core and RTL synthesized from our 8 point FFT code

5. Problems/ Errors:-

During this projects we faced the following problems:

1. The inbuilt pipeline command definitions such as xcl_pipeline_workitems used in the example code, were not being detected by the Xilinx compiler.

We discovered that the pipeline commands were supported in the VIVADO HLS 2015.2 version.

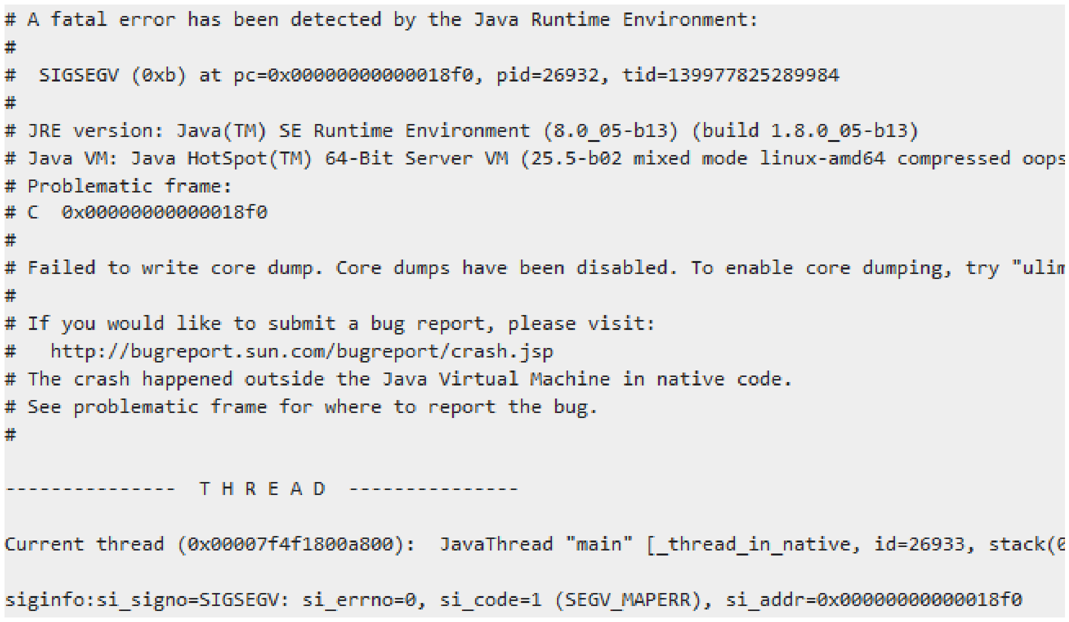
2.The attributes to the function async_work_group_copy were not matching the function definition arguments.

Using only the global memory for storing and modifications of the kernel arguments.

3. We were using Vivado Design Suite 2015.1. Vivado runs, but Vivado HLS crashes on start without any error.

After a while I found out that there is error. I have removed last line in "/Xilinx/Vivado_HLS/2015.1/bin/vivado_hls" file and ran it as root and error file appeared).

The error was SIGSEGV in main thread.

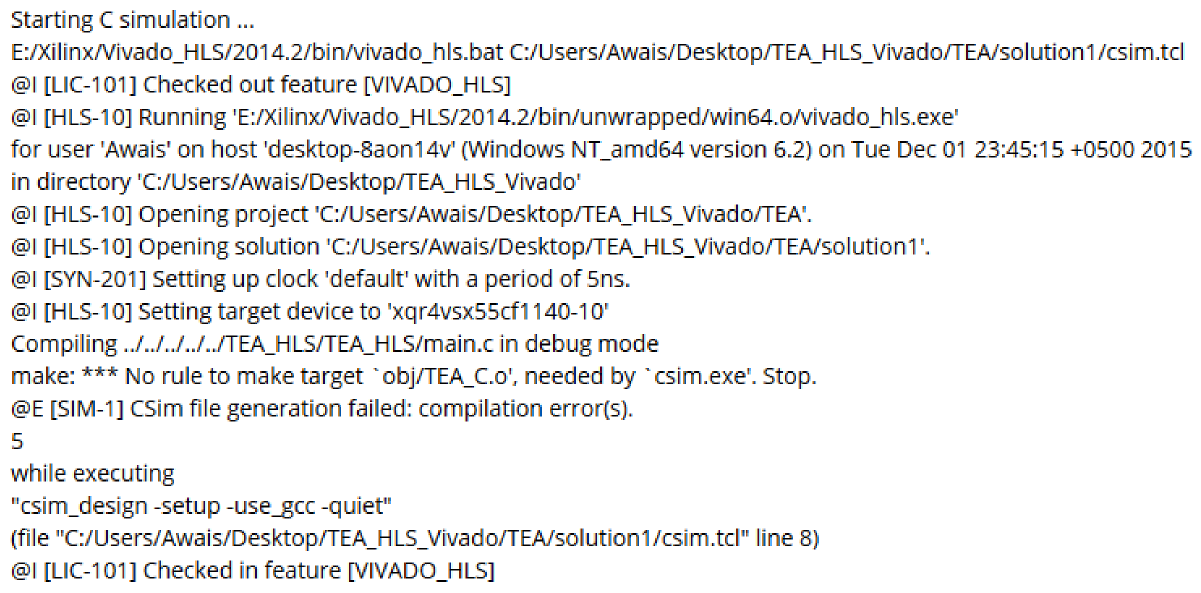


```
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00000000000018f0, pid=26932, tid=139977825289984
#
# JRE version: Java(TM) SE Runtime Environment (8.0_05-b13) (build 1.8.0_05-b13)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (25.5-b02 mixed mode linux-amd64 compressed oops
# Problematic frame:
# C  0x00000000000018f0
#
# Failed to write core dump. Core dumps have been disabled. To enable core dumping, try "ulim
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.sun.com/bugreport/crash.jsp
# The crash happened outside the Java Virtual Machine in native code.
# See problematic frame for where to report the bug.
#

---------------  T H R E A D  ---------------

Current thread (0x00007f4f1800a800):  JavaThread "main" [_thread_in_native, id=26933, stack(

siginfo:si_signo=SIGSEGV: si_errno=0, si_code=1 (SEGV_MAPERR), si_addr=0x00000000000018f0
```

We found a solution in Xilinx forum. We just add export SWT_GTK3=0 at the top of vivado_hls file.

4. While doing simulation, we were getting error of "csim_design -setup -use_gcc -quiet".

Below is complete log of simulation.



```
Starting C simulation ...
E:/Xilinx/Vivado_HLS/2014.2/bin/vivado_hls.bat C:/Users/Awais/Desktop/TEA_HLS_Vivado/TEA/solution1/csim.tcl
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'E:/Xilinx/Vivado_HLS/2014.2/bin/unwrapped/win64.o/vivado_hls.exe'
for user 'Awais' on host 'desktop-8aon14v' (Windows NT_amd64 version 6.2) on Tue Dec 01 23:45:15 +0500 2015
in directory 'C:/Users/Awais/Desktop/TEA_HLS_Vivado'
@I [HLS-10] Opening project 'C:/Users/Awais/Desktop/TEA_HLS_Vivado/TEA'.
@I [HLS-10] Opening solution 'C:/Users/Awais/Desktop/TEA_HLS_Vivado/TEA/solution1'.
@I [SYN-201] Setting up clock 'default' with a period of 5ns.
@I [HLS-10] Setting target device to 'xqr4vsx55cf1140-10'
Compiling ../../../../../TEA_HLS/TEA_HLS/main.c in debug mode
make: *** No rule to make target `obj/TEA_C.o', needed by `csim.exe'. Stop.
@E [SIM-1] CSim file generation failed: compilation error(s).
5
while executing
"csim_design -setup -use_gcc -quiet"
(file "C:/Users/Awais/Desktop/TEA_HLS_Vivado/TEA/solution1/csim.tcl" line 8)
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

As it was recommended on Xilinx, that this error is probably because we were using the old version of Vivado HLS. As a result of which we upgraded our Software and subsequently we were able to get rid of this error.

5. Since, we were new to Vivado HLS, we were interested in generating RTL code from compute intensive sections of a c code, so we took a simple code below, and synthetized with Vivado_HLS, I got no errors/warnings, but the report shows 0.0 timing estimation and no resource utilization.

We thought we were missing something or does it have to do with the coding style?

We did the same for another section of the code and it reports timing and utilization. Then, why there was so much uncertainty with resource utilization.

Code in C:

Example 1, with no results what so ever.

```
int main ( )
{
    int i, index, rev;
    int NumBits;
    for ( i=rev=0; i < NumBits; i++ )
    {
        rev = (rev << 1) | (index & 1);
        index >>= 1;
    }

    return rev;
}
```

EXAMPLE 2:

We took another function below and after synthesis in Vivado_HLS it reports a latency of 194 clock cycles and a utilization of 320 LUTS.

```
#define ROT32(x,n)    ((x << n) | (x >> (32 - n)))

int main( )
{
    long W[80];
    int i;
    //long temp, A, B, C, D, E, W[80];

    for (i = 16; i < 80; ++i) {
    W[i] = W[i-3] ^ W[i-8] ^ W[i-14] ^ W[i-16];

#ifdef USE_MODIFIED_SHA
    W[i] = ROT32(W[i], 1);
#endif
        //printf( "W[%i] = %li \n", i, W[i]);
    }
  return 0;
}
```

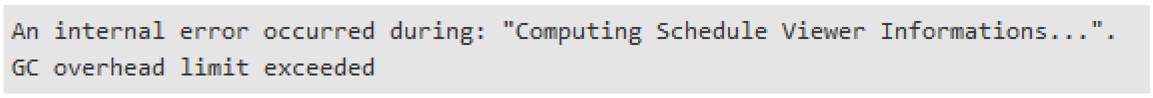
Later, we found out the solution to this problem,

1- int main() is the C TB function so you need to have another top level function otherwise you can't have a C TB to test your design; usually main() will call top() and top is the C function converted to HDL RTL.

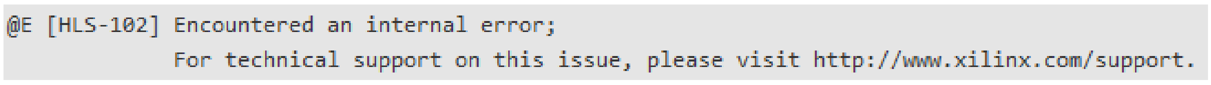
2- our design returns 0 so we managed to create an IP that does spend some clock cycles doing something.

6. Vivado crashes randomly during simulation

During our initial days Vivado was crashing randomly.



```
An internal error occurred during: "Computing Schedule Viewer Informations...".
GC overhead limit exceeded
```



```
@E [HLS-102] Encountered an internal error;
             For technical support on this issue, please visit http://www.xilinx.com/support.
```

This error is triggered due to the Java engine memory (heap) utilization being larger than the limits that are set

The limits that are set are based on environment variables of the OS.

The suggestion to resolving these issues then is to increase the memory heap size as necessary using the following methods:

Windows Environment Variable:

right click on "My Computer"

click on "Advanced System settings"

click on the "Environment Variables..." button.

create a new variable called "XIL_CS_JVMARGS" and set the value to "-Xmx2048M".

7. After we were done with C simulation and co-simulation of our C code. But when we used to get back to C simulation, the console message come up as follows:

```
@I [SYN-201] Setting up clock 'default' with a period of 5ns.
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Setting target device to 'xc7vx690tffg1158-2'
@E [HLS-101] 'csim_design': Unknown option '-compiled_library_dir'.
@E [HLS-101] 'csim_design': Unknown option 'D:/Compiled_Lib_15.1'.
FORMAT
csim_design [OPTIONS]
-O
-argv <string>
-clean
-ldflags <string>
-mflags <string>
-setup
command 'csim_design' returned error code
while executing
"csim_design -compiled_library_dir "D:/Compiled_Lib_15.1" -quiet"
(file "E:/HLS_projects/matrix_inv/solution4/csim.tcl" line 8)
invoked from within
"source E:/HLS_projects/matrix_inv/solution4/csim.tcl"
invoked from within
"hls::main E:/HLS_projects/matrix_inv/solution4/csim.tcl"
("uplevel" body line 1)
invoked from within
"uplevel 1 hls::main {*}$args"
(procedure "hls_proc" line 5)
invoked from within
"hls_proc $argv"
```

After putting it in forums, it was suggested to update our software. So, this error was reduced by upgrading our software to Vivado 15.2.1 version from 15.1

8. One of an issue with Vivado HLS 2015.2 regarding cosimulation using hls::stream as input and output interface is that

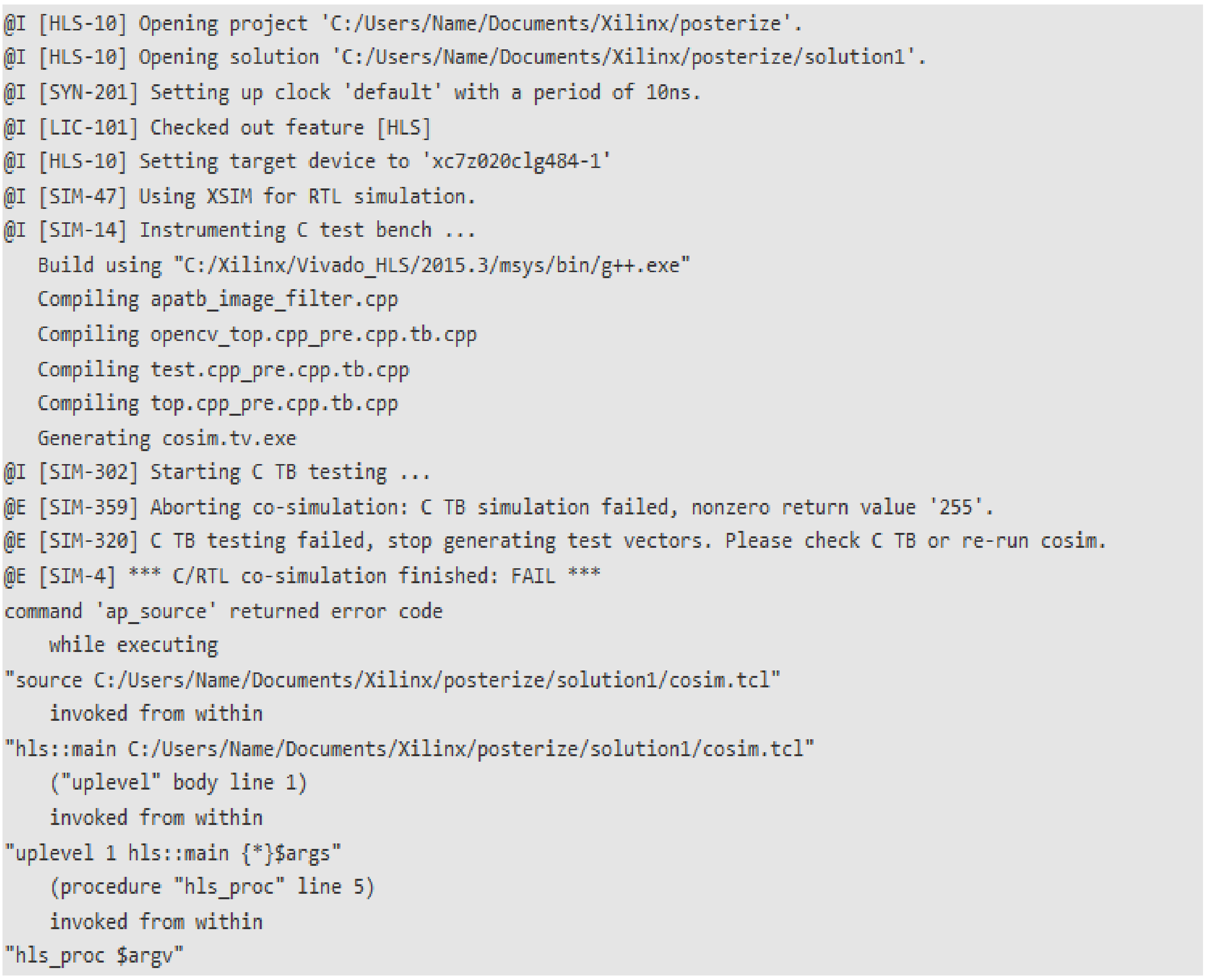
C-Testbench works fine, also C-Synthesis.

When we used to try to start the cosimulation using xsim after *##run all* in console, simulation does process and nothing else happens. Had it running for some hours but I doubt it takes so long to simulate a design with ~200 FFs and ~375 LUTs.

The Progress window shows a process bar Vivado HLS C/RTL Cosimulation with the words begin...... underneath.

This problem was solved by again upgrading to newer version of Vivado

When I run the C/RTL co-simulation, I get the following error:

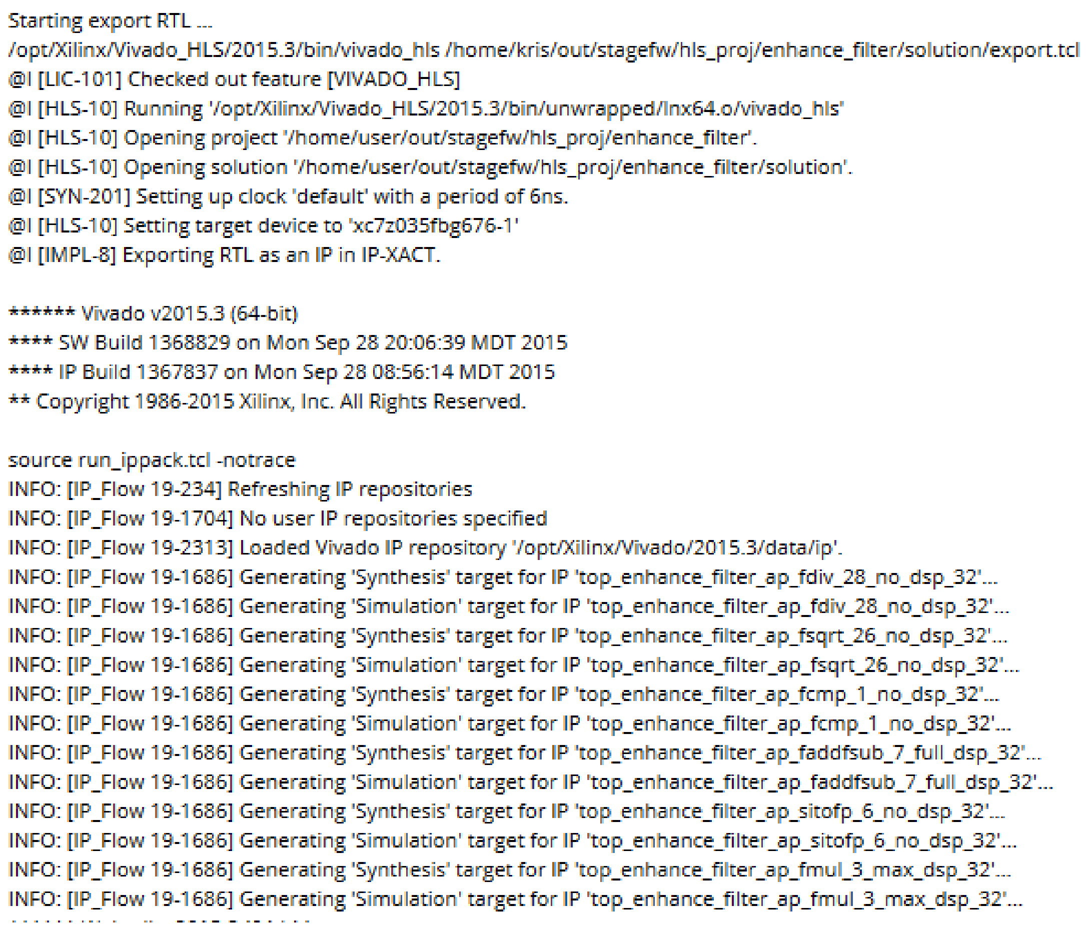


```
@I [HLS-10] Opening project 'C:/Users/Name/Documents/Xilinx/posterize'.
@I [HLS-10] Opening solution 'C:/Users/Name/Documents/Xilinx/posterize/solution1'.
@I [SYN-201] Setting up clock 'default' with a period of 10ns.
@I [LIC-101] Checked out feature [HLS]
@I [HLS-10] Setting target device to 'xc7z020clg484-1'
@I [SIM-47] Using XSIM for RTL simulation.
@I [SIM-14] Instrumenting C test bench ...
   Build using "C:/Xilinx/Vivado_HLS/2015.3/msys/bin/g++.exe"
   Compiling apatb_image_filter.cpp
   Compiling opencv_top.cpp_pre.cpp.tb.cpp
   Compiling test.cpp_pre.cpp.tb.cpp
   Compiling top.cpp_pre.cpp.tb.cpp
   Generating cosim.tv.exe
@I [SIM-302] Starting C TB testing ...
@E [SIM-359] Aborting co-simulation: C TB simulation failed, nonzero return value '255'.
@E [SIM-320] C TB testing failed, stop generating test vectors. Please check C TB or re-run cosim.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
command 'ap_source' returned error code
    while executing
"source C:/Users/Name/Documents/Xilinx/posterize/solution1/cosim.tcl"
    invoked from within
"hls::main C:/Users/Name/Documents/Xilinx/posterize/solution1/cosim.tcl"
    ("uplevel" body line 1)
    invoked from within
"uplevel 1 hls::main {*}$args"
    (procedure "hls_proc" line 5)
    invoked from within
"hls_proc $argv"
```

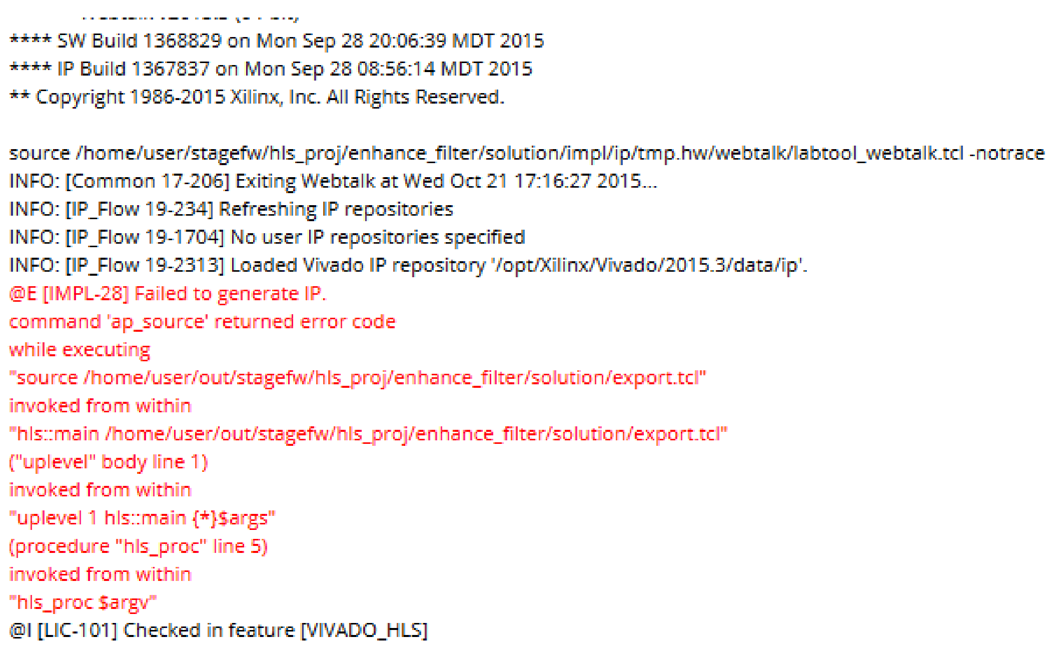
After running the same code for couple of time, we were able to figure it out that it was crashing right at the start of the application because the path of the code was not defined correctly.

9. When we started using the updated version of Vivado 2015.3

Everything was great for the HLS builds, however one of our HLS modules passes CSim and Cosim but fails on on Exporting RTL, below is the message got:



```
Starting export RTL ...
/opt/Xilinx/Vivado_HLS/2015.3/bin/vivado_hls /home/kris/out/stagefw/hls_proj/enhance_filter/solution/export.tcl
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running '/opt/Xilinx/Vivado_HLS/2015.3/bin/unwrapped/lnx64.o/vivado_hls'
@I [HLS-10] Opening project '/home/user/out/stagefw/hls_proj/enhance_filter'.
@I [HLS-10] Opening solution '/home/user/out/stagefw/hls_proj/enhance_filter/solution'.
@I [SYN-201] Setting up clock 'default' with a period of 6ns.
@I [HLS-10] Setting target device to 'xc7z035fbg676-1'
@I [IMPL-8] Exporting RTL as an IP in IP-XACT.

****** Vivado v2015.3 (64-bit)
**** SW Build 1368829 on Mon Sep 28 20:06:39 MDT 2015
**** IP Build 1367837 on Mon Sep 28 08:56:14 MDT 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/opt/Xilinx/Vivado/2015.3/data/ip'.
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'top_enhance_filter_ap_fdiv_28_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'top_enhance_filter_ap_fdiv_28_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'top_enhance_filter_ap_fsqrt_26_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'top_enhance_filter_ap_fsqrt_26_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'top_enhance_filter_ap_fcmp_1_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'top_enhance_filter_ap_fcmp_1_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'top_enhance_filter_ap_faddfsub_7_full_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'top_enhance_filter_ap_faddfsub_7_full_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'top_enhance_filter_ap_sitofp_6_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'top_enhance_filter_ap_sitofp_6_no_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Synthesis' target for IP 'top_enhance_filter_ap_fmul_3_max_dsp_32'...
INFO: [IP_Flow 19-1686] Generating 'Simulation' target for IP 'top_enhance_filter_ap_fmul_3_max_dsp_32'...
```



```
**** SW Build 1368829 on Mon Sep 28 20:06:39 MDT 2015
**** IP Build 1367837 on Mon Sep 28 08:56:14 MDT 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

source /home/user/stagefw/hls_proj/enhance_filter/solution/impl/ip/tmp.hw/webtalk/labtool_webtalk.tcl -notrace
INFO: [Common 17-206] Exiting Webtalk at Wed Oct 21 17:16:27 2015...
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/opt/Xilinx/Vivado/2015.3/data/ip'.
@E [IMPL-28] Failed to generate IP.
command 'ap_source' returned error code
while executing
"source /home/user/out/stagefw/hls_proj/enhance_filter/solution/export.tcl"
invoked from within
"hls::main /home/user/out/stagefw/hls_proj/enhance_filter/solution/export.tcl"
("uplevel" body line 1)
invoked from within
"uplevel 1 hls::main {*}$args"
(procedure "hls_proc" line 5)
invoked from within
"hls_proc $argv"
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

Actually fixing the new warnings in the new Vivado seems to work for this error that we were able to do within 1-2 days by editing our code.

6. PROGRAMMING USING VIVADO HLS

In the section that follows, all the OpenCL programs that were implemented using the Xilinx Vivado HLS are briefly explained to present an understanding of the Xilinx's implementation of OpenCL.

6.1 Vector Addition

Out of the many example projects part of the Xilinx Vivado HLS, there is only one example project that uses an OpenCL kernel. This example code performs the basic vector addition operation. The host and the kernel code are given below.

Host Code:

```
#include <stdio.h>
#include "vadd.h"
int main (int argc, char** argv)

{
    int errors=0, i;
    int a[LENGTH],b[LENGTH],hw_c[LENGTH],swref_c[LENGTH];

 // Fill our data sets with pattern and compute their sum for reference.

for(i = 0; i < LENGTH; i++)
{
        a[i] = i;
        b[i] = i;
        swref_c[i]= a[i] + b[i];
        hw_c[i] = 0;
    }

 // Execute the OpenCL kernel function to allow C simulation
    hls_run_kernel("vadd", a, LENGTH, b, LENGTH, hw_c, LENGTH );

// Verify the results match the pre-computed value
 // Return a value of 0 if the results are correct (required for RTL
cosimulation)


for (i=0; i<LENGTH; i++)
    {
        int diff = hw_c[i] != swref_c[i];
        if(diff) {
            printf("error - hw_c[%d]: %d differs from sw reference swref_c[%d]:
%d\n",i,    hw_c[i], i, swref_c[i]);
            errors+=diff;
        }
    }

  printf("There are %d error(s) -> test %s\n", errors, errors ? "FAILED" :
"PASSED");
    return errors;

}
```

Kernel Code:

```
#include <clc.h>                                                    // needed for OpenCL
kernels
#include "vadd.h"                                                   // header for this
example;

__kernel void __attribute__ ((reqd_work_group_size(LENGTH, 1, 1)))

vadd(__global int* a,
     __global int* b,
     __global int* c)

{
    __local int a_local[LENGTH];
    __local int b_local[LENGTH];
    __attribute__((xcl_pipeline_workitems))
    {
        async_work_group_copy(a_local,a,LENGTH,0);
        async_work_group_copy(b_local,b,LENGTH,0);
        barrier(CLK_LOCAL_MEM_FENCE);

        int idx = get_global_id(0);
        c[idx] = a_local[idx] + b_local[idx];

      }

}
```

This is the simplest code that can be used to understand about the structure of OpenCL programming model.
A brief explanation of the various parts of this code is given below.

The host code is pretty straightforward and is written using standard C constructs. The most important part of this code is the execution of the Kernel function. This is done using the hls_run_kernel function provided by Xilinx as a part of their abstraction level introduced to host programming. Vivado HLS provides the following function signature to execute the
OpenCL API C kernel:

void hls_run_kernel(const char *KernelName, ScalarType0 *Arg0, int size0, ScalarType1 *Arg1, int size1, …)

• **Arg0** is the first argument of the OpenCL API C kernel.
• **ScalarType0** is the type of the first argument of the OpenCL API C kernel.
• **Size0** is the number of data samples to be read or written.

In this code, the name of the kernel is “**vadd**” and 16 values must be read from both the inputs, a and b whereas the output is stored in hw_c.

Hence, the function call is,

hls_run_kernel ("vadd", a, LENGTH, b, LENGTH, hw_c, LENGTH)

Due to an additional level of abstraction introduced by Xilinx, the usual flow of creating data structures namely platforms, devices, contexts, command queues etc. is not followed. Instead, a single function call is used to execute the kernel.

The kernel code uses the header file “**clc.h**”.
This header file is required to use the OpenCL constructs in Vivado HLS and includes the definition of all the functions, data types and keywords used in OpenCL. It is sufficient to include only this header file.

The kernel function always begins with the keyword __*kernel* followed by the function signature. OpenCL uses attributes to provide hints to the compiler and these attributes are specified using the keyword __*attribute*.

In this example, the workgroup size required by the kernel is specified as an attribute. The function signature includes three variables a, b and c which are stored in the global memory. Inside the function, two local variables, a_local and b_local are used.

The attribute xcl_pipeline_workitems is used to pipeline the work items and is an optimization technique provided in OpenCL. The function async_work_group_copy is used to copy data from the global memory to the local memory. Since the updated local variables are used for further computation, *barrier (CLK_LOCAL_MEM_FENCE)* is used to make sure all the work items are synchronized. Execution cannot proceed further until all the work items have finished executing *barrier (CLK_LOCAL_MEM_FENCE)*.

The last two lines of the kernel perform the actual computation. *get_global_id(0)* returns the global id of the first work group. Depending upon the size of the work group mentioned using attribute *reqd_work_group_size*, the variable *idx* will have a range of values like in a 'for' loop in C and can be used to execute a statement parallely as many times. Thus the last statement effectively computes the sum of a and b and stores it in hw_c.

NOTE:

1. If work groups are utilized in the kernel i.e., get_global_id, get_local_id, get_global_size etc. then the attribute reqd_work_group_size must be used with the proper parameters specifying the number of work items required. If the attribute is not specified, the compilation fails and Xilinx does not specify where the error lies. If the parameters to the attribute are not proper, it can cause a Segmentation error.

2. 'Proper' parameters depend on the kernel code and is usually specified according to the number of times that particular work group items need to be executed.

6.2 8-Point FFT

Implementation of a FFT algorithm using OpenCL normally requires either running multiple kernel functions or using recursion.

Since both these features are currently not supported on the Xilinx Vivado HLS platform, the length of the FFT was chosen to be 8 (the least value allowed by the built-in Xilinx FFT IP core) and the code was developed to use a single kernel.

Host Code:

```
//8 point fft host code

#include<stdio.h>
#include <math.h>

int main()
{

   int i,length=8,sign=1;
    int twid[8]={0,0,0,0,0,0,0,0};//output
   int x[8]={1,4,5,2,3,7,7,8};//input
    int y[8]={0,0,0,0,0,0,0,0};

// Masking part. We wanted to mask 32 bit input, in such a manner
that we are taking on 8 bit at input.

int masking(int x[i] );
{                                         // Example value: 0x01020304
   for (i=0; i<8 ; i++ )
   {
   int byte1 = (x[i] >> 24);              // 0x01020304 >> 24 is 0x01 so
                                          // no masking is necessary
   int byte2 = (x[i] >> 16) & 0xff;      // 0x01020304 >> 16 is 0x0102 so
                                        // we must mask to get 0x02
   int byte3 = (x[i] >> 8)  & 0xff;      // 0x01020304 >> 8 is 0x010203 so
                                     // we must mask to get 0x03
   int byte4 = x[i] & 0xff;             // here we only mask, no shifting
                                     // is necessary
    y[i] = byte1;

}
}

   hls_run_kernel("fft",x,8,length,1,twid,8);
   for(i=0;i<2*length;i++)
          printf("%d /t %d \n",x[i],twid[i]);
   return 0;

}


}
```

In this host code, we have created a main function which take 8 bit fixed point data as input. And, for output we have created an array which have zero as its parameter.

Since, the input is 32 bits, we wanted to take only 8 bit data at the inputs we have used masking for this purpose. In masking function we have use the traditional technique of masking using AND, OR gates.

Following is the explanation of how we went about masking in our Host Code:

A mask defines which bits you want to keep, and which bits you want to clear.

Masking is the act of applying a mask to a value. This is accomplished by doing:

- Bitwise ANDing in order to extract a subset of the bits in the value
- Bitwise ORing in order to set a subset of the bits in the value
- Bitwise XORing in order to toggle a subset of the bits in the value

Let's say we want to extract some bits from higher no of bits, this is how we can do this:

Below is an example of extracting a subset of the bits in the value:

```
Mask:   00001111b
Value:  01010101b
```

Applying the mask to the value means that we want to clear the first (higher) 4 bits, and keep the last (lower) 4 bits. Thus we have extracted the lower 4 bits. The result is:

```
Mask:   00001111b
Value:  01010101b
Result: 00000101b
```

Masking is implemented using AND:

```
uint8_t stuff(...) {
  uint8_t mask = 0x0f;   // 00001111b
  uint8_t value = 0x55;  // 01010101b
  return mask & value;
}
```

Here is a very simple example which will help us to understand masking easily:

Extracting individual bytes from a larger word. We define the high-order bits in the word as the first byte. We use two operators for this, &, and >> (shift right). This is how we can extract the four bytes from a 32-bit integer:

```
void more_stuff(uint32_t value) {             // Example value: 0x01020304
    uint32_t byte1 = (value >> 24);           // 0x01020304 >> 24 is 0x01 so
                                              // no masking is necessary
    uint32_t byte2 = (value >> 16) & 0xff;    // 0x01020304 >> 16 is 0x0102 so
                                              // we must mask to get 0x02
    uint32_t byte3 = (value >> 8)  & 0xff;    // 0x01020304 >> 8 is 0x010203 so
                                              // we must mask to get 0x03
    uint32_t byte4 = value & 0xff;            // here we only mask, no shifting
                                              // is necessary
    ...
}
```

Notice that you could switch the order of the operators above, you could first do the mask, then the shift. The results are the same, but now you would have to use a different mask:

Kernel Code: (for 8 point FFT)

```
//8 point fft kernel code

# include <clc.h>

__kernel __attribute__((reqd_work_group_size(1,2,2))) void fft(__global
float2* Y,int N,__global float2* Y3)
{
int i;

      float2 twiddleFactors1[8];
   float2 twiddleFactors2[8];
   float2 Y1[8],Y2[8];

   for( i= 0;i<N/2;i++)

{
          Y1[2*i] = Y[2*i] + Y[2*i+1];
          Y1[2*i+1] = Y[2*i] - Y[2*i+1];

   }

      int j = get_global_id(1);
        Y1[2*j] = Y[2*j] + Y[2*j+1];
           Y1[2*j+1] = Y[2*j] - Y[2*j+1];

           float param1 = - 3.14159265359f * 2 * (2*j) / (float)N;
           float c1, s1 = sincos(param1, &c1);

             twiddleFactors1[2*j] = (float2)(c1, -s1);
             twiddleFactors2[2*j] = (float2)(s1, c1);

           float param2 = - 3.14159265359f * 2 * (2*j+1) / (float)N;
        float c2, s2 = sincos(param2, &c2);

          twiddleFactors1[2*j+1] = (float2)(c2, -s2);
             twiddleFactors2[2*j+1] = (float2)(s2, c2);


        for ( i=0; i<2; i++)

    {

          {
```

```
     Y2[j+i*N/2] = Y1[j+(i*4)] +
(float2)(dot(Y1[j+(i*4)+2],twiddleFactors1[2*j]),dot(Y1[j+(i*4)+2],twiddle
Factors2[2*j]));
           Y2[j+2+i*N/2] = Y1[j+(i*4)] -
(float2)(dot(Y1[j+(i*4)+2],twiddleFactors1[2*j]),dot(Y1[j+(i*4)+2],twiddle
Factors2[2*j]));

    }

 }


{
   int m = get_global_id(2);
for ( i=0; i<2; i++)

{

     {
           Y3[m+i*2] = Y2[m+i*2] +
(float2)(dot(Y2[m+i*2+4],twiddleFactors1[m+i*2]),dot(Y2[m+i*2+4],twidd
leFactors2[m+i*2]));
           Y3[m+i*2+4] = Y2[m+i*2] -
(float2)(dot(Y2[m+i*2+4],twiddleFactors1[m+i*2]),dot(Y2[m+i*2+4],twidd
leFactors2[m+i*2]));

          }

        }

    }
}
```

In this code, the kernel makes use of three work groups with sizes 1,2,2 respectively.

Each work group is used to perform exactly one stage of the 8-Point FFT. Vector operations are used to compute both real and imaginary values together.

The first stage uses get_global_id(0);
The second stage uses get_global_id(1);
The third stage uses get_global_id(2).
The work items in each work group execute parallelly.
All the three programs were successfully simulated and synthesized to obtain the RTL codes using the Xilinx Vivado HLS.

7. Future:

1. Exploit Parallelism: Use multiple kernels

In order to get better efficiency and performance, we need to use Multiple Kernels. Which is not supported in the latest version of Vivado, so as of now using Multiple Kernels can't be done. But, surely if in future, Xilinx Vivado extended their support for multiple Kernels, this should be our way forward.

Other option that we have is to use as many Work groups as possible. Work group is also called as threads. A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler. We can use as many threads/work groups as possible. Since, we want to exploit parallelism, we should always look for using as many threads/work items as possible.

But, since in this project we implemented small rather a basic algorithms, there is no requirements of using many threads. In fact we can't use many threads as we don't have that amount of computation required to be done.

But, in future we can certainly exploit the idea of using as many work items as possible if we have to do large amount of computation work.

2. Libraries

Libraries that are used in writing program also play an important role in making your kernel much simple and easier, and much more efficient. There are libraries which directly uses the single-precision values/data as compared to the default libraries of OpenCL or HLS. Such, libraries can be found easily and can be implemented it project.

Example:

One such library is CMAKE Library. CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.

Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment.

Documentation for CAMKE can be found at **cmake.org**. CMAKE uses only single prescion values by default. Whereas the default or the normal library of HLS called hls_math uses the double-precision. So it convert the float to a double, run the much more complex double-precision function, and then convert the result to a float. In the latter case, our implementation ended up utilizing more resources.

So, this can also be a good way to approach further, use libraries which can do things simply and easily without using much resources.

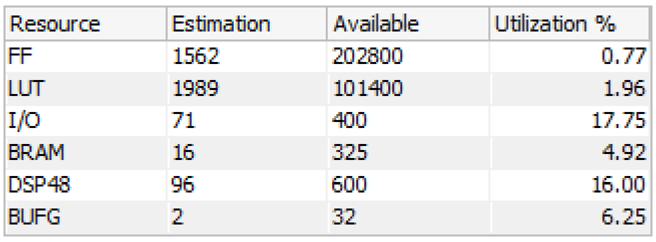
6. RESULTS

RTL code was synthesized from the 8 FFT Point OpenCL code on Vivado HLS. This RTL code was packaged in the form of an IP core and transported the IP catalogue of Vivado Design Suite. The synthesis and implementation of the IP core was performed on Vivado suite with following listings

Product Family	**--**	**virtex7**
Target Board	**--**	**xc7vx330tffv1157-1**

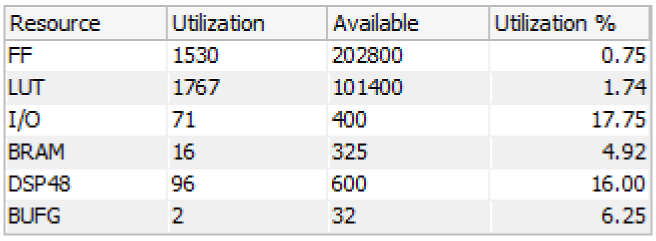
8 POINT FFT IP CORE WITH RADIX-2 IMPLEMENTATION FROM XILINX

Post Synthesis:



Resource	Estimation	Available	Utilization %
FF	1562	202800	0.77
LUT	1989	101400	1.96
I/O	71	400	17.75
BRAM	16	325	4.92
DSP48	96	600	16.00
BUFG	2	32	6.25

Post Implementation:

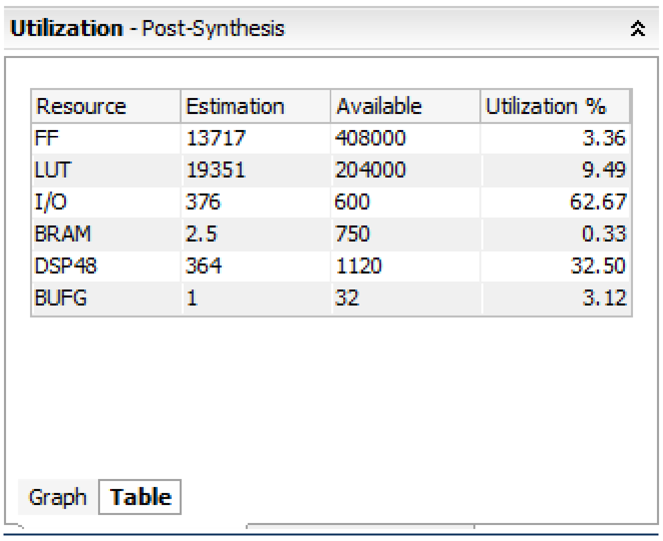


Resource	Utilization	Available	Utilization %
FF	1530	202800	0.75
LUT	1767	101400	1.74
I/O	71	400	17.75
BRAM	16	325	4.92
DSP48	96	600	16.00
BUFG	2	32	6.25

IP CORE SYNTHESIZED USING OPENCL CODE

Post Synthesis:

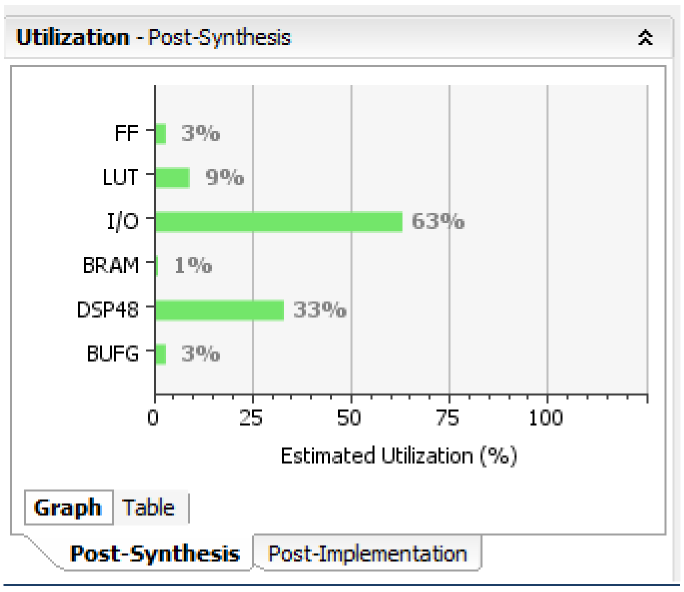
Table Format



Utilization - Post-Synthesis

Resource	Estimation	Available	Utilization %
FF	13717	408000	3.36
LUT	19351	204000	9.49
I/O	376	600	62.67
BRAM	2.5	750	0.33
DSP48	364	1120	32.50
BUFG	1	32	3.12

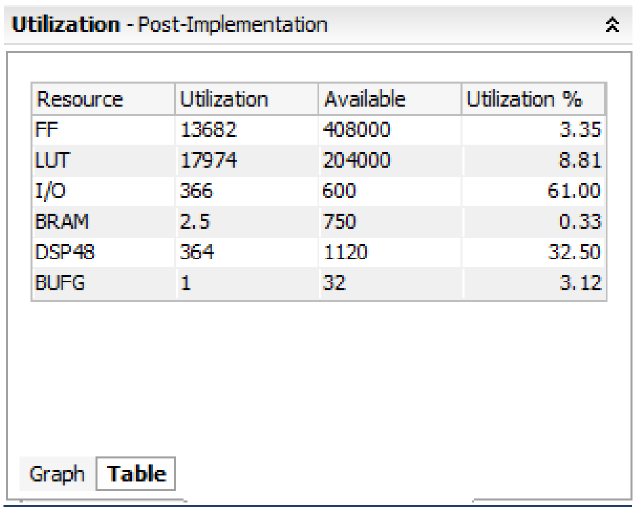
Graph Format



Utilization - Post-Synthesis

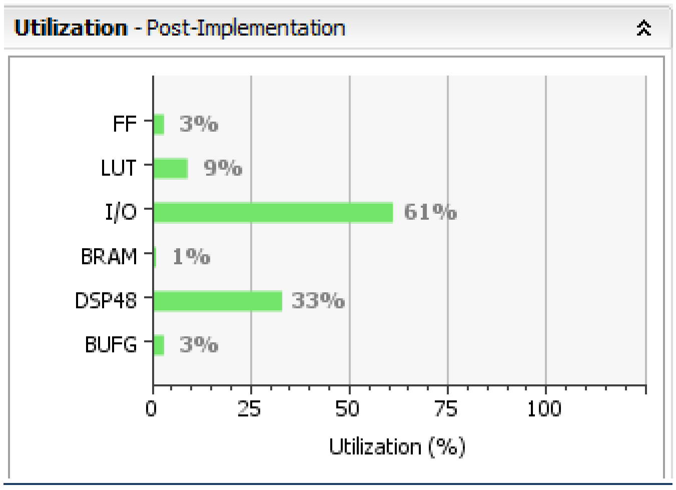
Post Implementation:

Table Format


Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	13682	408000	3.35
LUT	17974	204000	8.81
I/O	366	600	61.00
BRAM	2.5	750	0.33
DSP48	364	1120	32.50
BUFG	1	32	3.12

Graph Format



7. CONCLUSION:

The resource utilization of synthesized 8 point FFT IP core was compared with the inbuilt Xilinx example IP core. The results are improved to large extent, by cutting down the resource utilization by half of what was achieved earlier.

But, still results clearly suggest that the higher resource utilization by the IP core synthesized using OpenCL code, and can be optimized more to get better performance and resource utilization.

In order to distinctively arrive at the performance of OpenCL code, we suggest a timing analysis to derive the computing time in each of the test cases.

8. REFERENCES

[1] Mathew Scarpino, “OpenCL In Action: How to accelerate graphics and computation”, Manning Publications Co., ISBN: 9781617290176

[2] OpenCL on Khronos Group - https://www.khronos.org/opencl/

[3] OpenCL Reference Page on Khronos Group - https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/

[4] OpenCL 1.0 API and C Language Specification (revision 48, October 6, 2009)

[5] UG-902 Vivado Design Suite User Guide - High-Level Synthesis

[6] Xilinx SDAccel: A Unified Development Environment for Tomorrow’s Data Center

[7] OpenCL Reference Page on Stream Computing Group - http://streamcomputing.eu/consultancy/by-technology/opencl/