SDAccel Development Environment User Guide

Features and Development Flows

UG1023 (2015.4) February 16, 2016

XILINX

ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
02/16/2016	2015.4	Added details for RTL kernel support.
10/22/2015	2015.3	Added information on new XOCC enhancements and new device support.
09/15/2015	2015.1.5	Updated information on supported devices.
07/15/2015	2015.1.3	Added new section on dependency removal for pipelining.
06/29/2015	2015.1.2	Added new section on Pipes and new appendix on SDAccel support for OpenCL built-in functions.
05/26/2015	2015.1.0	Initial Xilinx release.

Table of Contents

Introduction

Software is at the foundation of application specification and development. Whether the end application is targeted towards entertainment, gaming, or medicine, most products available today began as a software model or prototype that needed to be accelerated and executed on a hardware device. From this starting point, the software engineer is tasked with determining the execution device to get a solution to market and to achieve the highest possible degree of acceleration possible.

One traditional approach to accomplish this task has been to rely on processor clock frequency scaling. On its own, this approach has entered a state of diminishing returns, which has in turn led to the development of multi-core and heterogeneous computing devices. These architectures provide the software engineer with the possibility to more effectively trade-off performance and power for different form factors and computational loads. The one challenge in using these new computing architectures is the programming model of each device. At a fundamental level, all multi-core and heterogeneous computing devices require that the programmer rethink the problem to be solved in terms of explicit parallelism.

Recognizing the programming challenge of multi-core and heterogeneous compute devices, the Khronos Group industry consortium has developed the OpenCL™ programming standard. The OpenCL specification for multi-core and heterogeneous compute devices defines a single consistent programming model and system level abstraction for all hardware devices that support the standard. For a software engineer this means a single programming model to learn what can be directly used on devices from multiple vendors.

As specified by the OpenCL standard, any code that complies with the OpenCL specification is functionally portable and will execute on any computing device that supports the standard. Therefore, any code change is for performance optimization. The degree to which an OpenCL program needs to be modified for performance depends on the quality of the starting source code and the execution environment for the application.

Xilinx is an active member of the Khronos Group, collaborating on the OpenCL specification, and supports the compilation of OpenCL programs for Xilinx® FPGA devices. The Xilinx SDAccel™ development environment is used for compiling OpenCL programs to execute on a Xilinx FPGA device.

There are some differences between compiling a program for execution in an FPGA and a CPU/GPU environment. The following chapters in this guide describe how to use the SDAccel development environment to compile an OpenCL program for a Xilinx FPGA. This book is intended to document the features and usages of the SDAccel development environment. It is assumed that the user already has a working knowledge of OpenCL API. Though it includes some high level OpenCL concepts, it is not intended as an exhaustive technical guide on the OpenCL API. For more information on the OpenCL API, see the OpenCL specification available from the Khronos Group, and the OpenCL API introductory videos available on the Xilinx website.

Platform and Memory Model

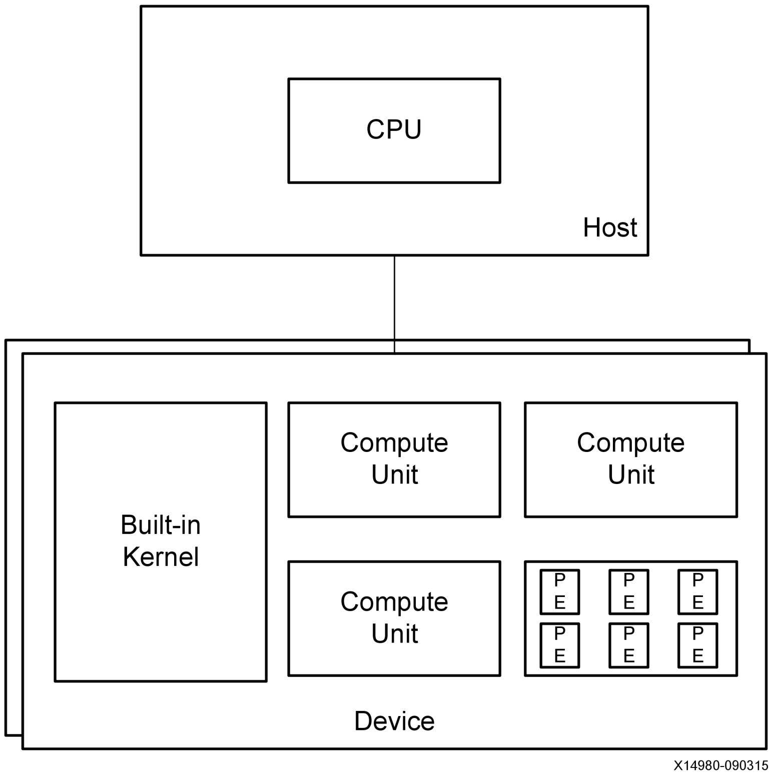
The OpenCL™ standard describes all hardware compute resources capable of executing OpenCL applications using a common abstraction for defining a platform and the memory hierarchy. The platform is a logical abstraction model for the hardware executing the OpenCL application code. This model, which is common to all vendors implementing this standard, provides the application programmer with a unified view from which to analyze and understand how an application is mapped into hardware. Understanding how these concepts translate into physical implementations on the FPGA is necessary for application optimization.

This chapter provides a review of the OpenCL platform model and its extensions to FPGA devices. It explains the mapping of the OpenCL platform and memory model into an SDAccel™ development environment-generated implementation.

OpenCL Platform Model

The OpenCL™ platform model defines the logical representation of all hardware capable of executing an OpenCL program. At the most fundamental level all platforms are defined by the grouping of a processor and one or more devices. The host processor, which runs the OS for the system, is also responsible for the general bookkeeping and task launch duties associated with the execution of parallel programs such as OpenCL applications. The device is the element in the system on which the kernels of the application are executed. The device is further divided into a set of compute units. The number of compute units depends on the target hardware for a specific application. A compute unit is defined as the element in the hardware device onto which a work group of a kernel is executed. This device is responsible for executing the operations of the assigned work group to completion. In accordance to the OpenCL standard division of work groups into work items, a compute unit is further subdivided into processing elements. A processing element is the data path in the compute unit, which is responsible for executing the operations of one work item. A conceptual view of this model is shown in the following figure.

Figure 2–1: **OpenCL Platform Model**



An OpenCL platform always starts with a host processor. For the case of platforms created with Xilinx® devices, the host processor is an x86-based processor communicating to the devices using a PCIe™ solution. The host processor has the following responsibilities:

- Manage the operating system and enable drivers for all devices.
- Execute the application host program.
- Set up all global memory buffers and manage data transfer between the host and the device.
- Monitor the status of all compute units in the system.

In all OpenCL platforms, the host processor tasks are executed using a common set of API functions. The implementation of the OpenCL API functions is provided by the hardware vendor and is referred to as the runtime library.

The OpenCL runtime library, which is provided by the hardware vendor, is the logical layer in a platform that is responsible for translating user commands described by the OpenCL API into hardware specific commands for a given device. For example, when the application programmer allocates a memory buffer using the clCreateBuffer API call, it is the responsibility of the runtime library to keep track of where the allocated buffer physically resides in the system, and of the mechanism required for buffer access. It is important for the application programmer to keep in mind that the OpenCL API is portable across vendors, but the runtime library provided by a vendor is not. Therefore, OpenCL applications have to be linked at compile time with the runtime library that is paired with the target execution device.

The other component of a platform is the device. A device in the context of an OpenCL API is the physical collection of hardware resources onto which the application kernels are executed. A platform must have at least one device available for the execution of kernels. Also, per the OpenCL platform model, all devices in a platform do not have to be of identical type.

OpenCL Devices and FPGAs

In the context of CPU and GPU devices, the attributes of a device are fixed and the programmer has very little influence on what the device looks like. On the other hand, this characteristic of CPU/GPU systems makes it relatively easy to obtain an off-the-shelf board. The one major limitation of this style of device is that there is no direct connection between system I/O and the OpenCL™ kernels. All transactions of data are through memory-based transfers.

An OpenCL device for an FPGA is not limited by the constraints of a CPU/GPU device. By taking advantage of the fact that the FPGA starts off as a blank computational canvas, the user can decide the level of device customization that is appropriate to support a single application or a class of applications. In determining the level of customization in a device, the programmer needs to keep in mind that kernel compute units are not placed in isolation within the FPGA fabric.

FPGA devices capable of supporting OpenCL programs consist of the following:

- Connection to the host processor
- I/O peripherals
- Memory controllers
- Interconnect
- Kernel region

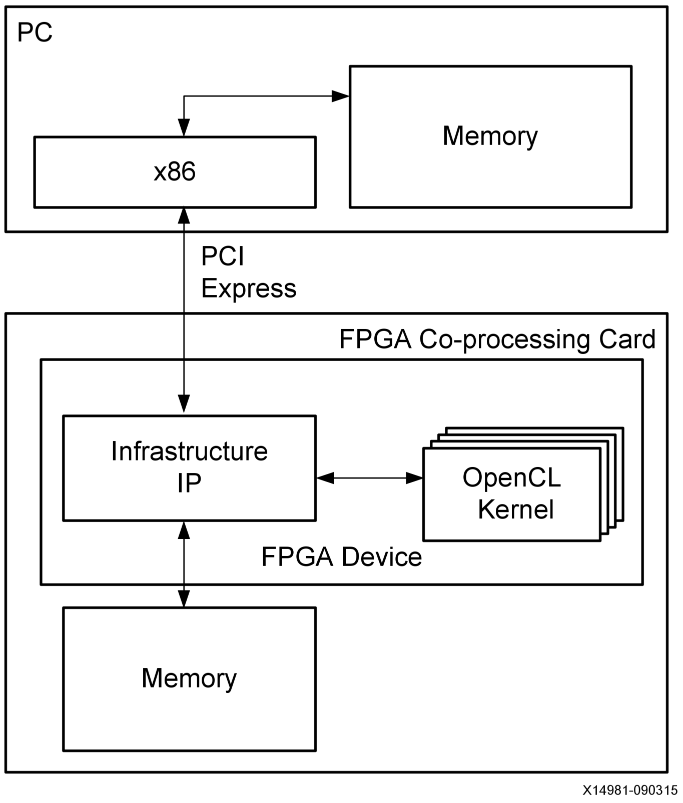
The creation of FPGA devices requires FPGA design knowledge and is beyond the scope of capabilities for the SDAccel™ development environment. Devices for the SDAccel environment are created using the Xilinx Vivado® Design Suite for FPGA designers. The SDAccel environment provides pre-defined devices and allows users to augment the tool with third party created devices. A methodology guide describing how to create a device for the SDAccel development environment is available upon request from Xilinx.

The devices available in the SDAccel environment are for Virtex®-7 and Kintex®-7 FPGAs. These devices are available in a PCIe form factor. The PCIe form factor for Virtex-7 and Kintex-7 devices assumes that the host processor is an x86-based processor and that the FPGA is used for the implementation of compute units.

PCIe Reference Device

The PCIe™ base device has a distributed memory architecture, which is also found in GPU accelerated compute devices. This means that the host and the kernels access data from separate physical memory domains. Therefore, the developer has to be aware that passing buffers between the host and a device triggers memory data copies between the physical memories of the host and the device. The data transfer time must be accounted for when determining the best optimization strategy for a given application. A representative example of this type of device is shown in the following figure.

***Figure 2–2:* PCIe Base Device**



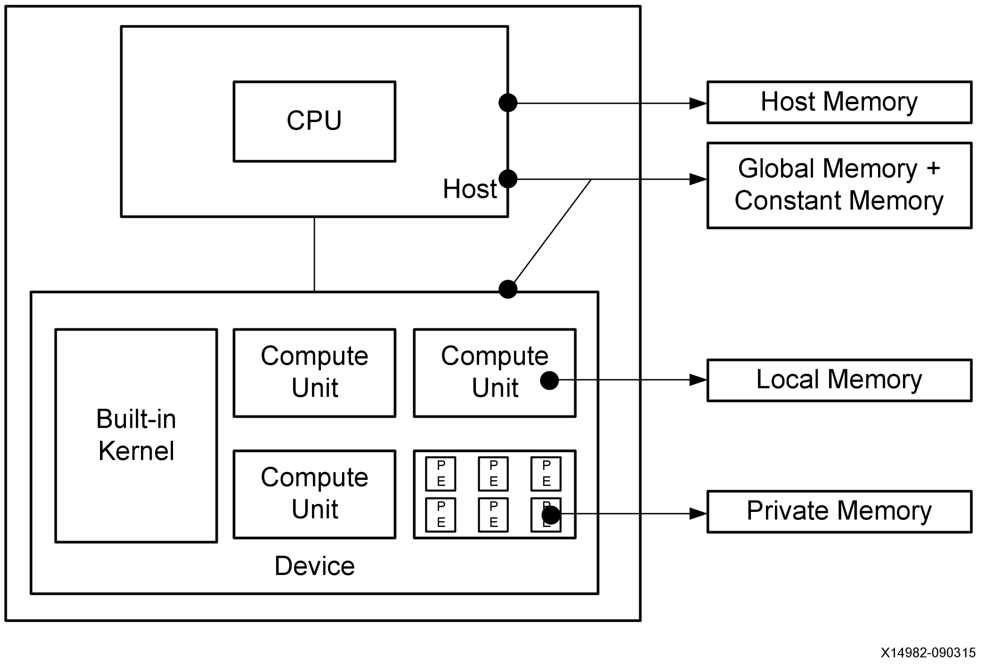
The main characteristics of devices with a PCIe form factor are as follows:

- The x86 processor in the PC is the host processor for the OpenCL™ application.
- The infrastructure IP provided as part of the device is needed for communication to the host over the PCIe core and to access the DDR memories on the board.
- Connecting OpenCL kernels to IP other than infrastructure IP or blocks generated by the SDAccel™ development environment is not supported.
- Kernels work on data in the DDR memory attached to the FPGA device.

OpenCL Memory Model

The OpenCL™ API defines the memory model to be used by all applications that comply with the standard. This hierarchical representation of memory is common across all vendors and can be applied to any OpenCL application. The vendor is responsible for defining how the OpenCL memory model maps to specific hardware. The OpenCL memory model is shown overlaid onto the OpenCL device model in the following figure.

Figure 2–3: **OpenCL Memory Model**



The memory hierarchy defined in the OpenCL specification has the following levels:

- Host Memory
- Global Memory
- Constant Global Memory
- Local Memory
- Private Memory

Host Memory

The host memory is defined as the region of system memory that is only visible and accessible 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 located in this space. Any data needed by a kernel must be transferred into global memory so that it is accessible by a compute unit.

Global Memory

The global memory is defined as the region of system memory that is accessible to both the OpenCL™ host and device. The host is responsible for the allocation and deallocation of buffers in this memory space. There is a handshake between host and device over control of the data stored in this memory. The host processor transfers data from the host memory space into the global memory space. Then, when a kernel is launched to process the data, the host loses access rights to the buffer in global memory. The device takes over and is capable of reading and writing from the global memory until the kernel execution is complete. Upon completion of the operations associated with a kernel, the device turns control of the global memory buffer back to the host processor. Once it has regained control of a buffer, the host processor can read and write data to the buffer, transfer data back to the host memory, and deallocate the buffer.

Constant Global Memory

Constant global memory is defined as the region of system memory that is accessible with read and write access for the OpenCL™ host and with read-only access for the OpenCL device. As the name implies, the typical use for this memory is to transfer constant data needed by kernel computation from the host to the device.

Local Memory

Local memory is defined as the region of system memory that is only accessible to the OpenCL™ device. The host processor has no visibility and no control on the operations that occur in this memory space. This memory space allows read and write operations by the work items within the same compute unit. This level of memory is typically used to store and transfer data that must be shared by multiple work items.

Private Memory

Private memory is the region of system memory that is only accessible by a processing element within an OpenCL™ device. This memory space can be read from and written to by a single work item.

For devices using an FPGA device, the physical mapping of the OpenCL memory model is the following:

- Host memory is any memory connected to the host processor only.
- Global and constant memories are any memory that is connected to the FPGA device. These are usually memory chips that are physically connected to the FPGA device. The host processor has access to these memory banks through infrastructure in the FPGA base device.
- Local memory is memory inside of the FPGA device. This memory is typically implemented using block RAM elements in the FPGA fabric.
- Private memory is memory inside of the FPGA device. This memory is typically implemented using registers in the FPGA fabric in order to minimize latency to the compute data path in the processing element.

Compilation Flow

The Xilinx® SDAccel™ development environment is used for creating and compiling OpenCL™ applications onto a Xilinx FPGA. This tool suite provides a software development environment for algorithm development and emulation on x86 based workstations, as well as deployment mechanisms for Xilinx FPGA devices.

The compilation of OpenCL applications into binaries for execution on an FPGA does not assume nor require FPGA design knowledge. A basic understanding of the capabilities of an FPGA is necessary during application optimization in order to maximize performance. The SDAccel environment handles the low-level details of program compilation and optimization during the generation of application specific compute units for an FPGA fabric. Therefore, using the SDAccel environment to compile an OpenCL program does not place any additional requirements on the user beyond what is expected for compilation towards a CPU or GPU target.

This chapter explains how an OpenCL program is compiled for execution on an FPGA by the SDAccel environment.

Starting the SDAccel Development Environment

The SDAccel™ development environment is a command line-based tool suite for compiling OpenCL™ programs into a Xilinx® FPGA device. This tool suite works in a batch mode in which the user invokes the tool with a command file as in the following command:

```
sdaccel <command file>
```

The command file is a text file which specifies a sequence of commands for the SDAccel environment to execute during the process of compiling an OpenCL application. An example command file for the SDAccel environment is shown below.

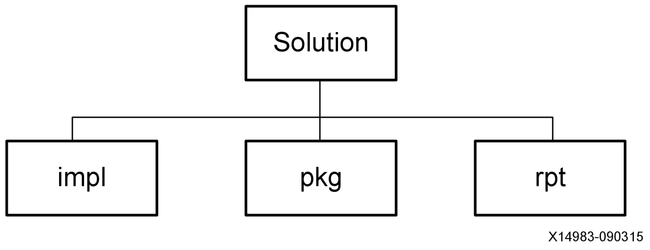
```
# Define a solution name for the current run of SDAccel
create_solution -name mysolution

# Set the target device for the current solution
add_device "xilinx:adm-pcie-7v3:1ddr:2.1"

# Set any special host compilation flags
set_property -name host_cflags -value {-g -Wall -D FPGA_DEVICE} -objects [current_solution]

# Add host code
add_files test-cl.c

# Create an OpenCL Kernel Binary Container
create_opencl_binary mmult1

# Select the execution region within the target device
set_property region "OCL_REGION_0" [get_opencl_binary mmult1]

# Create a kernel
create_kernel -type clc mmult

# Add code to the kernel
add_files -kernel [get_kernels mmult] mmult1.cl

# Create a compute unit
create_compute_unit -opencl_binary [get_opencl_binary mmult1] -kernel [get_kernels mmult] \
   -name myinstance

# Compile application for CPU based emulation
compile_emulation

# Compile the application for hardware emulation
compile_emulation -flow hardware

# Run the CPU based emulation
run_emulation

# Run the hardware emulation
run_emulation -flow hardware

# Generate system performance estimates
report_estimate

# Compile the application to run on an FPGA
build_system

# Package the solution
package_system

# Run the application in hardware
run_system
```

The commands in the script above cover all of the main compilation tasks in the SDAccel environment. Each command is explained in detail throughout this guide.

Creating a Solution

The first step in any SDAccel™ development environment design is to create a solution. A solution is a directory structure and application representation used by the SDAccel environment to compile a design. The directory structure, which is common to all SDAccel environment designs, is shown in the following figure.

Figure 3–1: **SDAccel Environment Directory Structure**



The command to start a new application is as follows:

```
create_solution -name mysolution
```

The `create_solution` command creates the environment in which an OpenCL™ application can be developed and compiled. In addition to generating the solution object for the SDAccel environment to compile the application, the purpose of the solution is to define the device on which the compiled application is executed. A solution can only have one target device. Compiling the same application for multiple devices requires the application programmer to run the SDAccel environment with one solution definition per board. A target device is added to a solution by the following command:

```
add_device "xilinx:adm-pcie-7v3:1ddr:2.1"
```

A complete list of devices available in the SDAccel environment is available in SDAccel Environment Supported Devices.

Adding Host Code

An OpenCL™ application is composed of a host code base and a kernel code base. The host code base is the user code that runs on a processor and utilizes the OpenCL runtime API to enqueue and launch kernels to complete a specific computational workload. The SDAccel™ development environment compiles host code onto an x86 processor for application development and emulation. For deployment, the SDAccel environment compiles the host code for the host processor specified in the target device. The SDaccel environment supports devices with x86 based host processors.

The host code for an application is specified by the following:

```
add_files test-cl.c
```

By default, the `add_files` command marks files to be compiled onto the host processor. The path to the source file can be relative to the current directory or absolute. Compilation of host code files happens in memory, which means the SDAccel environment does not require the user to modify how their source code is stored in any way.

Creating a Kernel

The computational foundation of every OpenCL™ application is a kernel. A kernel defines the code that will be accelerated and executed in parallel to complete a given application. The SDAccel™ development environment supports solutions with N kernels. Each kernel can be targeted for execution on a processor or the FPGA fabric depending on the resources available in the device chosen to execute the application. In addition, a kernel can be specified in either the OpenCL C kernel language or in C/C++ that is suitable for the Vivado® High-Level Synthesis (HLS) tool.

A kernel is created in the context of an SDAccel solution by the following:

```
create_kernel -type clc mmult
```

The `create_kernel` command takes as arguments the name of the kernel and the language in which it is specified. The name of the kernel must match the name of the top kernel function in the source code for which the programmer wants the SDAccel environment to create a custom compute unit. The type parameter is used to tell the compiler which language front end to use in processing the application code. The choice of kernel language depends on the preference of the application programmer. The SDAccel environment compiles both OpenCL C and C/C++ for Vivado HLS with the same compiler.

Expressing a Kernel in OpenCL C

The SDAccel™ environment supports the OpenCL™ language constructs and built in functions from the OpenCL 1.0 embedded profile. The following is an example of an OpenCL kernel for matrix multiplication that can be compiled with the SDAccel environment.

```
__kernel __attribute__ ((reqd_work_group_size(16,16,1)))
void mult(__global int* a, __global int* b, __global int* output)
{
  int r = get_local_id(0);
  int c = get_local_id(1);
  int rank = get_local_size(0);
  int running = 0;
  for(int index = 0; index < 16; index++){
    int aIndex = r*rank + index;
    int bIndex = index*rank + c;
    running += a[aIndex] * b[bIndex];
  }
  output[r*rank + c] = running;
  return;

}
```

Expressing a Kernel in C/C++

The kernel for matrix multiplication can be expressed in C/C++ code that can be synthesized by the Vivado® HLS tool. For kernels captured in this way, the SDAccel™ development environment supports all of the optimization techniques available in Vivado HLS. The only thing that the user has to keep in mind is that expressing kernels in this way requires compliance with a specific function signature style.

```
void mmult(int *a, int *b, int *output)
{
#pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=b offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=b bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  const int rank = 16;
  int running = 0;
  int bufa[256];
  int bufb[256];
  int bufc[256];
  memcpy(bufa, (int *) a, 256*4);
  memcpy(bufb, (int *) b, 256*4);

  for (unsigned int c=0;c<rank;c++){
    for (unsigned int r=0;r<rank;r++){
      running=0;
      for (int index=0; index<rank; index++) {
  #pragma HLS pipeline
        int aIndex = r*rank + index;
        int bIndex = index*rank + c;
        running += bufa[aIndex] * bufb[bIndex];
      }
      bufc[r*rank + c] = running;
    }
  }

  memcpy((int *) output, bufc, 256*4);
  return;
}
```

The preceding code example is the matrix multiplication kernel expressed in C/C++ for Vivado HLS. The first thing to notice about this code is the function signature.

```
void mmult(int *a, int *b, int *output)
```

This function signature is almost identical to the signature of the kernel expressed in OpenCL C. It is important to keep in mind that by default, kernels captured in C/C++ for HLS do not have any inherent assumptions on the physical interfaces that will be used to transport the function parameter data. HLS uses pragmas embedded in the code to direct the compiler as to which physical interface to generate for a function port. For the function to be treated as a valid OpenCL kernel, the ports on the C/C++ function must be reflected on the memory and control interface pragmas for HLS.

The memory interface specification is

```
#pragma HLS INTERFACE m_axi port=<variable name> offset=slave bundle=<interface name>
```

The control interface specification is

```
#pragma HLS INTERFACE s_axilite port=<variable name> bundle=<interface name>
```

Detailed information on how these pragmas are used is available in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

IMPORTANT: *Global variables in HLS C/C++ kernels are not supported.*

Adding Kernel Code to a Solution

As in the case of host code, the SDAccel™ development environment requires that the user add all source and header files that are associated with a kernel defined in the solution. This task is accomplished by the `add_files` command with an additional kernel parameter.

The code for a kernel in the solution is specified by the following command.

```
add_files -kernel [get_kernels mmult] mmult1.cl
```

The `add_files` command has the same behavior as with host code. The only difference is that code added with a `-kernel` flag is associated with a specific kernel object in the current solution. The path to the source file can be relative to the current directory or absolute. Compilation of kernel code files happens in memory, which means the SDAccel environment does not require the user to modify how their source code is stored in any way.

Creating the Xilinx OpenCL Compute Unit Binary Container

The main difference between targeting an OpenCL™ application to a CPU/GPU and targeting an FPGA is the source of the compiled kernel. Both CPUs and GPUs have a fixed computing architecture onto which the kernel code is mapped by a compiler. Therefore, OpenCL programs targeted for either kind of device invokes just in time compilation of kernel source files from the host code. The API for invoking just in time kernel compilation is as follows:

```
clCreateProgramWithSource(…)
```

In contrast to a CPU or a GPU, an FPGA can be thought of as a blank computing canvas onto which a compiler generates an optimized computing architecture for each kernel in the system. This inherent flexibility of the FPGA allows the developer to explore different kernel optimizations and compute unit combinations that are beyond what is possible with a fixed architecture. The only drawback to this flexibility is that the generation of a kernel specific optimized compute architecture takes a longer time than what is acceptable for just in time compilation. The OpenCL standard addresses this fundamental difference between devices by allowing for an offline compilation flow. This allows the user to generate libraries of kernels that can be loaded and executed by the host program. The OpenCL API for supporting kernels generated in an offline compilation flow is as follows:

```
clCreateProgramWithBinary(…)
```

The SDAccel™ development environment uses an offline compilation flow to generate kernel binaries. To maximize efficiency in the host program and allow the simultaneous instantiation of kernels that cooperate in the computation of a portion of an application, Xilinx has defined the Xilinx® OpenCL Compute Unit Binary format `.xclbin`. The `xclbin` file is a binary library of kernel compute units that will be loaded together into an OpenCL context for a specific device. This format can hold either programming files for the FPGA fabric or shared libraries for the processor. It also contains library descriptive metadata, which is used by the Xilinx OpenCL runtime library during program execution.

The library metadata included in the `xclbin` file is automatically generated by the SDAccel environment and does not require user intervention. This data is composed of compute unit descriptive information that is automatically generated during compute unit synthesis and used by the runtime to understand the contents of an `xclbin` file.

The `xclbin` file is created using the following:

```
create_opencl_binary -device <device name> <binary name>
```

An SDAccel solution can generate N binary files. The appropriate number of binary files depends on the application and the amount of hardware resource time multiplexing that is desired. The SDAccel environment assumes that all compute units stored within a binary can be executed concurrently on a given device. The name given during the invocation of the `create_opencl_binary` command defines the name of the `xclbin` file to be used by the host program. For the matrix multiplication example, the compiled kernel library `xclbin` is the following:

```
mmult.xclbin
```

The `create_opencl_binary` command does not launch the generation of any compute unit in the SDAccel solution. This command sets up the SDAccel solution with the target kernel library containers that are created later in the compilation flow.

Choosing the Kernel Execution Target

The SDAccel™ development environment has the ability to compile kernels into compute units for execution on devices in an OpenCL™ device. The valid devices supported by the SDAccel environment are processors and the FPGA fabric. The SDAccel environment does not make a decision on where a compute unit is to be implemented. This decision is left to the application programmer and is communicated to the tool as part of the binary definition with the `create_opencl_binary` command.

The first stage of defining where compute units will be implemented is to define the area of a device that is targeted by a given `xclbin` file. By default, the SDAccel environment targets the first available processing region in a device for the implementation of compute units. For devices with more than one region for compute units, the user can select a specific region with the following command:

```
set_property region <region name> [get_opencl_binary <binary name>]
```

The name of a region in the property described above is provided as part of the device. Because the SDAccel environment can handle the generation of multiple xclbin binary containers per solution, the user must annotate this property with the name of the binary file to which it applies.

The second stage of defining where a kernel will be executed is to instantiate compute units in the system. A compute unit is the custom hardware data path that the SDAccel environment generates to implement the functionality in the kernel code. The SDAccel environment supports the generation of up to N compute units for any kernel and there is no limit on the number of kernels and compute units that an application can have. The limitation on compute units which can be executed concurrently is placed by the device developer who defines how many of the available device resources can be dedicated to the implementation of compute units. The SDAccel environment issues a warning if the number of compute units packed into a single binary container exceeds the computational resources available in the target device. A compute unit for a kernel is defined by the following command:

```
create_compute_unit -opencl_binary [get_opencl_binary <binary name>]
                    -kernel [get_kernels <kernel name>]
                    -name <instance name>
```

This command creates a compute unit for a specific kernel into the specified binary container. The `name` field for a compute unit instance name is optional. If the application programmer only wants one compute unit for a given kernel, then the `name` field can be omitted from the `create_compute_unit` command. For cases where more than one compute unit is required for a kernel, the `create_compute_unit` command can be repeated as many times as necessary. The only requirement is that the `name` field is used with a unique name every time the `create_compute_unit` command is used for the same kernel. Also, a compute unit instance name cannot be used more than once per binary container.

The specifics of how a kernel is compiled and synthesized into a custom compute unit depend on the region of the device selected during the definition of the binary container. The `create_compute_unit` command does not launch the compilation and synthesis of kernels into compute units and binary containers. This is the last solution setup command in the SDAccel environment.

Building the System

Compilation of an application for execution on a Xilinx® enabled OpenCL™ device through the SDAccel™ development environment is called the build system step. This step goes beyond compilation of host and kernel code and is also responsible for the generation of custom compute units for all of the binary containers in the solution.

The following is the command for this step in the flow:

```
build_system
```

For each binary container in the solution, the `build_system` command takes inventory of all of the compute units that must be generated and their execution region in the OpenCL device. The `build_system` command completes after all compute units and binary containers defined in a solution have been created.

The compilation method used for each kernel by the `build_system` command is dependent on the user selected kernel execution target. The `build_system` commands invokes different flows for kernels targeted at a processor and kernels targeted at the FPGA fabric.

Build Flow for Compute Units Targeting the FPGA Fabric

The SDAccel™ development environment generates custom logic for every compute unit in the binary container. Therefore, it is normal for this build step to run for a longer period of time than the other steps in the SDAccel application compilation flow.

The steps in compiling compute units targeting the FPGA fabric are as follows:

1. Generate a custom compute unit for a specific kernel.
2. Instantiate the compute units in the OpenCL™ binary container.
3. Connect the compute units to memory and infrastructure elements of the target device.
4. Generate the FPGA programming file.

The generation of custom compute units for any given kernel code utilizes the production proven capabilities of the Xilinx® Vivado® High-Level Synthesis (HLS) tool, which is the compute unit generator in the SDAccel environment. Based on the characteristics of the target device in the solution, the SDAccel environment invokes the compute unit compiler to generate custom logic that maximizes performance while at the same time minimizing compute resource consumption on the FPGA fabric. Automatic optimization of a compute unit for maximum performance is not possible for all coding styles without additional user input to the compiler. Kernel Optimization discusses the additional user input that can be provided to the SDAccel environment to optimize the implementation of kernel operations into a custom compute unit.

After all compute units have been generated, these units are connected to the infrastructure elements provided by the target device in the solution. The infrastructure elements in a device are all of the memory, control, and I/O data planes which the device developer has defined to support an OpenCL application. The SDAccel environment combines the custom compute units and the base device infrastructure to generate an FPGA binary which is used to program the Xilinx device during application execution.

IMPORTANT: *The SDAccel environment always generates a valid FPGA hardware design, but does not generate an optimal allocation of the available bandwidth in the control and memory data planes. The user can manually optimize the data bandwidth utilization by selecting connection points into the memory and control data planes per compute unit.*

Package System

The final stage of SDAccel™ application compilation is to package the generated system for execution and deployment on FPGA based accelerator cards to be run outside of the SDAccel environment. The command for this step is the following:

```
package_system
```

The output of the `build_system` command is stored in the following location:

```
<solution name>/pkg
```

All files necessary to execute the application on a Xilinx® OpenCL™ device outside of the SDAccel environment are provided at this location.

Run System

After the application is compiled and packaged by the SDAccel™ development environment, the user has the choice of launching the application from within the SDAccel environment or running it on a Linux console. To run the application from within the SDAccel environment the command use the following:

```
run_system -args <command line arguments>
```

The `run_system` command executes the compiled user application with any command line arguments added by the `args` flag. This command is only valid for applications compiled against PCIe® based devices and requires the PCIe based accelerator card to be connected to the development workstation.

Application Emulation

One of the key differences between compiling for a CPU/GPU and an FPGA is the lack of fixed computation architectures in the FPGA. The SDAccel™ development environment leverages the inherent flexibility of the FPGA to generate a kernel specific compute architecture that allows the programmer to fine tune an application not only for performance, but also for power. While there are many benefits to executing an application on an FPGA, this execution device does introduce a design challenge not found in CPU/GPU development.

The challenge presented by FPGA devices is that the development and execution of an application occur on two distinct systems. Application development happens on the application programmer's development workstation, and deployment can occur on remote servers loaded with FPGA based accelerator cards. The difference in environment during an application's life cycle introduces the need for emulation before the application is deployed.

This chapter describes how the SDAccel environment enables emulation of an OpenCL™ application targeted to an FPGA. In the context of the compilation flow described in Compilation Flow, the commands in this chapter are to be executed before the `build_system` command.

CPU Emulation

In the context of the SDAccel™ development environment, application emulation on a CPU is the same as the iterative development process that is typical of CPU/GPU programming. In this type of development style, a programmer continuously compiles and runs an application as it is being developed.

Although partitioning and optimizing an application into kernels is integral to OpenCL™ development, performance is not the main goal at this stage of application development in the SDAccel environment. The main goal of CPU emulation is to ensure functional correctness and to partition the application into kernels.

For CPU-based emulation, both the host code and the kernel code are compiled to run on an x86 processor. The programmer model of iterative algorithm refinement through fast compile and run loops is preserved with speeds that are the same as a CPU compile and run cycle. The steps to enable this development style in the SDAccel environment are the following:

- Compile for emulation
- Run the emulation

Compile for CPU Emulation

The CPU based emulation flow in the SDAccel™ development environment is only concerned with the functional correctness of an application. Therefore, for the purpose of checking functionality, all kernels in an application are compiled into compute units that are executed as threads on an x86 based processor. The SDAccel environment leverages the functional portability of the OpenCL™ standard to abstract the details of compiling for emulation from the application programmer. The same host code that will be deployed when the application runs on the FPGA based accelerator card can be used for all of the emulation flows in the SDAccel environment.

Compiling the solution for CPU-based emulation is executed by the following command:

```
compile_emulation
```

Because this emulation is based on all parts of the application executing on an x86 processor, the compilation time is very fast and similar to what is expected when compiling a program for CPU/GPU execution.

Run CPU Emulation

After the application is compiled using the `compile_emulation` command, it can be executed within the SDAccel™ environment to verify functional correctness. The command to execute the CPU-based emulation in the SDAccel environment is the following:

```
run_emulation -args <command line arguments>
```

The `run_emulation` command executes the CPU emulation of an OpenCL™ application. This command has an additional `args` parameter, which is optional. The purpose of the `args` parameters is to enable the application programmer to pass any command line options to the host program which would also be used during application deployment.

The `run_emulation` command runs the host code program to completion and invokes the Xilinx® OpenCL runtime library as well as all necessary kernels to complete the functionality. This command does not do any checks on correctness of the application execution. It is the responsibility of the application programmer to verify the results of the program.

IMPORTANT: *The SDAccel environment has no concept of what a user application does or what constitutes correct behavior. The programmer must check application results for correctness.*

Hardware Emulation

The SDAccel™ development environment generates at least one custom compute unit for each kernel in an application. This means that while the CPU emulation flow is a good measure of functional correctness, it does not guarantee correctness on the FPGA execution target. Before deployment, the application programmer should also check that the custom compute units generated by the tool are producing the correct results.

The SDAccel environment has a hardware emulation flow, which enables the programmer to check the correctness of the logic generated for the custom compute units. This emulation flow invokes the hardware simulator in the SDAccel environment to test the functionality of the logic that will be executed on the FPGA compute fabric. The steps in the hardware emulation flow are the following:

- Compile for hardware emulation
- Run the hardware emulation

Compile for Hardware Emulation

The FPGA system created by the SDAccel™ development environment for a given application consists of device infrastructure and custom compute units packed into the Xilinx® OpenCL™ Binary Container. The current hardware emulation flow is based on checking the correctness of the compute units generated by the SDAccel environment. Correctness of the device is the responsibility of the device developer and is beyond the current scope of the SDAccel environment.

Compiling the solution for hardware emulation is executed by the following command:

```
compile_emulation -flow hardware
```

Hardware emulation requires the SDAccel environment to generate the custom logic for each compute unit in the system. It is expected for this step in the SDAccel flow to take several minutes.

Run Hardware Emulation

After the application is compiled for hardware emulation, it can be executed within the SDAccel™ environment to verify functional correctness of the custom compute units. The command to execute the CPU based emulation in the SDAccel environment is the following:

```
run_emulation -flow hardware -args <command line arguments>
```

The `run_emulation` command with the `hardware flow` qualifier executes the hardware emulation of an OpenCL™ application. This command has an additional `args` parameter, which is optional. The purpose of the `args` parameters is to enable the application programmer to pass any command line options to the host program which would also be used during application deployment.

The `run_emulation` command runs the host code program to completion and invokes the Xilinx® OpenCL runtime library as well as all necessary kernels to complete the functionality. It is important to keep in mind that the compute units implementing the kernel operations are executed within a hardware simulator for this type of emulation. Therefore, the time it takes to complete this command is directly proportional to the amount of data transferred from the host to the kernels and the number of operations in a compute unit.

The `run_emulation` command does not do any checks on correctness of the application execution. It is the responsibility of the application programmer to verify the results of the program.

IMPORTANT: *The SDAccel environment has no concept of what a user application does or what constitutes correct behavior. The programmer must check application results for correctness.*

Performance Estimate

The generation of FPGA programming files is the step in the SDAccel™ development environment with the longest execution time. It is also the step in which the execution time is most affected by the target hardware device and the number of compute units placed on the FPGA fabric. Therefore, it is essential for the application programmer to have an understanding of the performance of the application before running it on the target device. This chapter introduces the profiling capabilities in the SDAccel environment.

Generating the System Performance Estimate Report

The system performance estimate in the SDAccel™ development environment takes into account the target hardware device and each compute unit in the application. Although an exact performance metric can only be measured on the target device, the estimation report in the SDAccel environment provides an accurate representation of the expected behavior. For the benefits of system level estimation before system implementation to be seen, the generation of the system performance estimate report precedes the `build_system` command in the compilation flow. The command to generate the system performance estimate report is:

```
report_estimate
```

The system performance estimate report is stored in:

```
<solution name>/rpt
```

Analyzing the Performance Estimate Report

The performance estimate report generated by the `report_estimate` command provides information on every binary container in the application, as well as every compute unit in the design. The structure of the report is:

- Target device information
- Summary of every kernel in the application
- Detailed information on every binary container in the solution

The following example report file represents the information that is generated by the `report_estimate` command.

```
-------------------------------------------------------------------------------
Design Name:      baseline_solution
Target Platform:  1ddr
Target Board:
Target Clock:     200MHz
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
Kernel Summary

Total number of kernels: 1

+-----------------------+------------+----------------------+-----------------+---------------+
| Kernel Name           | Type       | Target               | OpenCL Library  | Compute Units |
+-----------------------+------------+----------------------+-----------------+---------------+
| smithwaterman         | clc        | fpga0:OCL_REGION_0   | test            | 1             |
+-----------------------+------------+----------------------+-----------------+---------------+
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
OpenCL Binary = test

Kernels mapped to = clc_region

Timing Information (MHz)
+-----------------+-----------------+--------------------+-------------------------+
| Compute Unit    | Kernel Name     | Target Frequency   | Estimated Frequency     |
+-----------------+-----------------+--------------------+-------------------- ----+
| K1              | smithwaterman   | 239.808            | 273.973                 |
+-----------------+-----------------+--------------------+----------------- -------+

Latency Information (clock cycles)
+-----------------+-----------------+------------------+------------+------------+------------+
| Compute Unit    | Kernel Name     | Start Interval   | Best Case  | Avg Case   | Worst Case |
+-----------------+-----------------+------------------+------------+------------+------------+
| K1              | smithwaterman   | 166424 ~ 209264  | 166423     | 187843     | 209263     |
+-----------------+-----------------+------------------+------------+------------+------------+

Area Information
+-----------------+-----------------+------------+------------+------------+------------+
| Compute Unit    | Kernel Name     | FF         | LUT        | DSP        | BRAM       |
+-----------------+-----------------+------------+------------+------------+------------+
| K1              | smithwaterman   | 1775       | 2923       | 1          | 9          |
+-----------------+-----------------+------------+------------+------------+------------+
```

Design and Target Device Summary

All design estimate reports begin with an application summary and information about the target hardware. Device information is provided by the following section of the report:

```
-------------------------------------------------------------------------------
Design Name:      baseline_solution
Target Platform:  1ddr
Target Board:
Target Clock:     200MHz
-------------------------------------------------------------------------------
```

For the design summary, the only information you provide is the design name and the selection of the target device. The other information provided in this section is the target board and the clock frequency.

The target board is the name of the board that runs the application compiled by the SDAccel™ development environment. The clock frequency defines how fast the logic runs for compute units mapped to the FPGA fabric. Both of these parameters are fixed by the device developer. These parameters cannot be modified from within the SDAccel environment.

Kernel Summary

The kernel summary section lists all of the kernels defined for the current SDAccel™ solution. Following is an example kernel summary:

```
Kernel Summary

Total number of kernels: 1

+-----------------------+------------+----------------------+-----------------+---------------+
| Kernel Name           | Type       | Target               | OpenCL Library  | Compute Units |
+-----------------------+------------+----------------------+-----------------+---------------+
| smithwaterman         | clc        | fpga0:OCL_REGION_0   | test            | 1             |
+-----------------------+------------+----------------------+-----------------+---------------+
```

Along with the kernel name, the summary provides the execution target and the OpenCL™ binary container where the compute unit of the kernel is stored. Also, because there is a difference in compilation and optimization methodology for OpenCL C and C/C++ source files, the type of kernel source file is specified.

The kernel summary section is the last summary information in the report. From here, detailed information on each compute unit binary container is presented.

Timing Information

The detail section for each binary container begins with the execution target of all compute units. It also provides timing information for every compute unit. As a general rule, an estimated frequency that is higher than that of the device target means that the compute unit will run in hardware. If the estimated frequency is below the target frequency, the kernel code for the compute unit needs to be further optimized for the compute unit to run correctly on the FPGA fabric. Following is an example of this information:

```
OpenCL Binary = test

Kernels mapped to = clc_region

Timing Information (MHz)
+-----------------+-----------------+--------------------+-------------------------+
| Compute Unit    | Kernel Name     | Target Frequency   | Estimated Frequency     |
+-----------------+-----------------+--------------------+-------------------- ----+
| K1              | smithwaterman   | 239.808            | 273.973                 |
+-----------------+-----------------+--------------------+----------------- -------+
```

The importance of the timing information is the difference between the target and the estimated frequencies. As stated in Platform and Memory Model, compute units are not placed in isolation into the FPGA fabric. Compute units are placed as part of a valid FPGA design that can include other components defined by the device developer to support a class of applications.

Because the compute unit custom logic is generated one kernel at a time, an estimated frequency that is higher than the device target provides confidence to the developer using the SDAccel™ environment that there will not be any problems during the creation of the FPGA programming files.

Latency Information

The latency information presents the execution profile of each compute unit in the binary container. When analyzing this data, it is important to keep in mind that all values are measured from the compute unit boundary through the custom logic. In-system latencies associated with data transfers to global memory are not reported as part of these values. Also, the latency numbers reported are only for compute units targeted at the FPGA fabric. Following is an example of the latency report:

```
Latency Information (clock cycles)
+-----------------+-----------------+------------------+------------+------------+------------+
| Compute Unit    | Kernel Name     | Start Interval   | Best Case  | Avg Case   | Worst Case |
+-----------------+-----------------+------------------+------------+------------+------------+
| K1              | smithwaterman   | 166424 ~ 209264  | 166423     | 187843     | 209263     |
+-----------------+-----------------+------------------+------------+------------+------------+
```

The latency report is divided into the following fields:

- Start internal
- Best case latency
- Average case latency

The start interval defines the amount of time that has to pass between invocations of a compute unit for a given kernel. This number sets the limit as to how fast the runtime can issue application ND range data tiles to a compute unit.

The best and average case latency numbers refer to how much time it takes the compute unit to generate the results of one ND range data tile for the kernel. For cases where the kernel does not have data dependent computation loops, the latency values will be the same. Data dependent execution of loops introduces data specific latency variation that is captured by the latency report.

Area Information

Although the FPGA can be thought of as a blank computational canvas, there are a limited number of fundamental building blocks available in each FPGA device. These fundamental blocks (FF, LUT, DSP, block RAM) are used by the SDAccel™ development environment to generate the custom logic for each compute unit in the design. The number of each fundamental resource needed to implement the custom logic in a compute unit determines how many compute units can be simultaneously loaded into the FPGA fabric. Following is an example of the area information reported for a compute unit:

```
Area Information
+-----------------+-----------------+------------+------------+------------+------------+
| Compute Unit    | Kernel Name     | FF         | LUT        | DSP        | BRAM       |
+-----------------+-----------------+------------+------------+------------+------------+
| K1              | smithwaterman   | 1775       | 2923       | 1          | 9          |
+-----------------+-----------------+------------+------------+------------+------------+
```

Kernel Optimization

The OpenCL™ standard guarantees functional portability but not performance portability. Therefore, even though the same code can run on every device supporting the OpenCL API, the performance achieved varies depending on coding style and capabilities of the underlying hardware. Optimizing for an FPGA using the SDAccel™ tool chain requires the same effort as code optimization for a CPU/GPU. The one difference in optimization for these devices is that in a CPU/GPU, the programmer is trying to get the best mapping of an application onto a fixed architecture. For an FPGA, the programmer is concerned with guiding the compiler to generate optimized compute architecture for each kernel in the application.

This chapter introduces the methodology by which the SDAccel environment handles some of the key constructs and attributes for the OpenCL kernel language. It also presents kernel optimization techniques available in the SDAccel development environment.

IMPORTANT: *Attributes are programmer hints to the OpenCL™ compiler for performance optimization. It is up to each individual compiler to follow or ignore the attribute. All attributes used by the SDAccel compiler that are not part of the OpenCL standard start with the tag* `xcl`.

SDAccel compiler defines a macro `__xilinx__` *that can be used to conditionally include SDAccel specific attributes in a kernel.*

```
#ifdef __xilinx__
__attribute__((xcl_pipeline_loop))
#endif
for (int i = 1; i < N; i++) {
localS2[i] = s2[i];
}
```

Work Group Size

The workgroup size in the OpenCL™ standard defines the size of the ND range space that can be handled by a single invocation of a kernel compute unit. In CPU and GPU based devices, the attribute on the kernel to define the workgroup size is typically optional. The runtime sets the workgroup size based on device characteristics and launches parallel threads on the hardware to complete the application.

In the case of an FPGA implementation, the specification of the workgroup size is optional but highly recommended. The attribute is recommended for performance optimization during the generation of the custom logic for a kernel.

Starting with the following kernel code:

```
__kernel __attribute__((reqd_work_group_size(4,4,1)))
void mmult32(__global int* A, __global int* B,
             __global int* C)
{
  // 2D Thread ID
  int i = get_local_id(0);
  int j = get_local_id(1);
  __local int Blocal[256];
  int result=0, k=0;
  Blocal[i*16 + j] = B[i*16 + j];
  barrier(CLK_LOCAL_MEM_FENCE);

  for(k=0;k<16;k++) result += A[i*16+k]*B_local[k*16+j];

  C[i*16+j] = result;
}
```

The SDAccel™ compiler generates custom logic to implement kernel functionality with or without the required workgroup size attribute. To generate an accelerator capable of handling an ND range problem space tile, the SDAccel compiler converts the kernel code into a form that is acceptable to the kernel compiler, which is based on the Vivado® High-Level Synthesis (HLS) tool. The code transformation starts with the function signature and the workgroup size:

```
__kernel __attribute__((reqd_work_group_size(4,4,1)))
void mmult32(global int *A,global int *B,global int *C)
{
  . . .
}
```

This code is transformed into:

```
__kernel void mmult32(global int* A,global int* B,global int* C)
{
localid_t id;
int B_local[16*16];
    for(id[2]=0;id[2]<1;id[2]++)
        for(id[1]=0;id[1]<4;id[1]++)
            for(id[0]=0;id[0]<4;id[0]++){
. . .
}
…
}
```

The three `for` loops introduced by the SDAccel compiler into the kernel code are necessary to handle the three-dimensional space of the ND range. The programmer has to keep in mind that at runtime the compute unit in the FPGA fabric can only handle as many threads as the generated hardware can handle. After the implementation is generated, there is no way to dynamically reallocate device resources at runtime.

The loop nest introduced by the SDAccel compiler can have either variable or fixed loop bounds. By setting the `reqd_work_group_size` attribute, the programmer is setting the loop boundaries on this loop nest. Fixed boundaries allow the kernel compiler to optimize the size of local memory in the compute unit and to provide latency estimates. If the workgroup size is not specified, the SDAccel compiler assumes a large size for the local memory, which can hinder the number of compute units that can be instantiated in the FPGA fabric. In addition, the latency numbers for the kernel are not calculated by the `report_estimate` command, because latency is assumed to be data dependent.

The OpenCL standard `reqd_work_group_size(x, y, z)` attribute can only support maximize work group size of 4096 with all three dimensions combined (x*y*z <= 4096). The SDAccel compiler allows kernels to implement larger work group size with an SDAccel attribute `xcl_max_work_group_size(x,y,z)`. Following is an example of the attribute:

```
__kernel __attribute__ ((xcl_max_work_group_size(8192, 8192, 1)))
void mmult(__global int* a, __global int* b, __global int* output)
{
…
}
```

Barriers

Barriers are the OpenCL™ kernel language constructs used to ensure memory consistency. When a kernel is launched, there is no guarantee on the order of execution for all work items in a kernel. Work items can be executed sequentially, in parallel, or in a combination of sequential and parallel execution. The execution start of a work item depends on the underlying hardware device. Because order of execution cannot be guaranteed across multiple devices, the OpenCL standard introduces the concept of barriers to create check points for memory consistency.

By definition, a barrier states that all work items must arrive at the barrier before any work item is allowed to proceed with the rest of the computation task. This ensures memory consistency and prevents any work item from overwriting data required by a different work item before it has been read. A barrier in OpenCL kernel code is expressed as:

```
__kernel __attribute__ reqd_work_group_size(4,4,1)
void mmult32(global int *A, global int *B, global int *C)
{
    // 2D Thread ID
    int i = get_local_id(0);
    int j = get_local_id(1);
    __local int B_local[16*16];

        async_work_group_copy(B_local,B,256, 0);

    barrier(CLK_LOCAL_MEM_FENCE);
    for(k=0;k<16;k++)
        result += A[i*16+k] * B_local[k*16+j];

    C[i*16+k] = result;
}
```

The SDAccel™ compiler transforms the barrier call into:

```
__kernel void mmult32(global int* A, global int* B, global int* C)
{
    localid_t id;
    int B_local[16*16];

    for(id[2]=0;id[2]<1;id[2]++)
        for(id[1]=0;id[1]<4;id[1]++)
            for(id[0]=0;id[0]<4;id[0]++) {
                int i = get_local_id(id,0);
                int j = get_local_id(id,1);
                if(i == 0 && j == 0)
                    async_work_group_copy(B_local,B,256, 0);
            }

    for(id[2]=0;id[2]<1;id[2]++)
        for(id[1]=0;id[1]<4;id[1]++)
            for(id[0]=0;id[0]<4;id[0]++) {
                int i = get_local_id(id,0);
                int j = get_local_id(id,1);
                for(k=0;k<16;k++)
```

```
                        result += A[i*16+k] * B_local[k*16+j];

                    C[i*16+k] = result;
                }
}
```

The size of the work group determines the boundaries on both loop nests. The SDAccel™ compiler executes loop nests in sequential order, which preserves the memory consistency properties of a barrier.

Loop Unrolling

Loop unrolling is the first optimization technique available in the SDAccel™ compiler. The purpose of the loop unroll optimization is to expose concurrency to the compiler. This is an official attribute in the OpenCL™ 2.0 specification.

For example, starting with the code:

```
/* vector multiply */
kernel void
vmult(local int* a, local int* b, local int* c)
{
  int tid = get_global_id(0);
  for (int i=0; i<4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx] * c[idx];
  }
}
```

The execution profile of the example is marked by four loop boundaries. This profile is similar to the following figure:

Table 6–1: **Default Execution Profile of a Loop**

iter0	iter1	iter2	iter3
Read b[0]	Read b[1]	Read b[2]	Read b[3]
Read b[0]	Read c[1]	Read c[2]	Read c[3]
*	*	*	*
Write a[0]	Write a[1]	Write a[2]	Write a[3]

In the execution profile shown in the previous figure, loop iterations execute on the same hardware elements in a purely sequential manner. Even if there is a possibility of running computations from multiple iterations in parallel, everything is executed sequentially because there is no concurrency being exposed to the compiler. This type of loop execution consumes the least amount of FPGA fabric resources. In this type of execution, not all of the hardware elements of the custom kernel logic are working all the time, which increases latency and decreases performance. The execution profile of the loop can be improved by using the loop unroll attribute with an unroll factor:

```
kernel void
vmult(local int* a, local int* b, local int* c)
{
  int tid = get_global_id(0);
  __attribute__((opencl_unroll_hint(2)))
  for (int i=0; i<4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx] * c[idx];
  }
}
```

The code above tells the SDAccel compiler to unroll the loop by a factor of two. This results in two loop iterations instead of four for the compute unit to complete the operation. The execution profile of this code is as shown in the following figure.

Table 6–2: **Loop Execution Profile after Unroll by a Factor of 2**

iter 0	iter 1
Read b[0]	Read b[2]
Read c[0]	Read c[2]
Read b[1]	Read b[3]
Read c[1]	Read c[3]
*	*
*	*
Write a[0]	Write a[2]
Wrate a[1]	Write a[3]

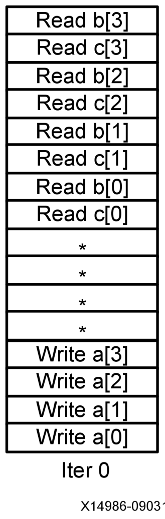
By enabling the SDAccel compiler to reduce the loop iteration count to two, the programmer exposes more concurrency to the compiler. The newly exposed concurrency reduces latency, improves performance, and consumes more FPGA fabric resources.

The last hint that you can provide the SDAccel compiler with this attribute is to unroll the loop completely. The syntax for the fully unrolled version of the code example is as shown below:

```
kernel void
vmult(local int* a, local int* b, local int* c)
{
  int tid = get_global_id(0);
  __attribute__((opencl_unroll_hint))
  for (int i=0; i<4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx] * c[idx];
  }
}
```

Fully unrolling the loop nest results in a single iteration with the execution profile shown in the following figure.

Figure 6–1: **Loop Execution Profile after Complete Unrolling**



In this case, all of the possible concurrency in the loop nest is exposed to the compiler. The SDAccel compiler analyzes the data and control dependencies of the unrolled loop nest and automatically makes parallel all operations that can be executed concurrently.

Loop Pipelining

Although loop unrolling exposes concurrency, it does not address the issue of keeping all elements in a kernel logic busy at all times, which is necessary for maximizing application throughput. Even in an unrolled case, loop control dependencies can lead to sequential behavior. The sequential behavior of operations results in idle hardware and a loss of performance.

Xilinx addresses the issue of idle hardware resources by introducing a vendor extension on top of the OpenCL™ 2.0 specification for loop pipelining. The Xilinx attribute for loop pipelining is:

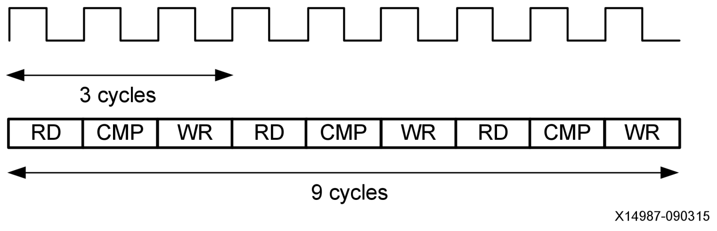
```
xcl_pipeline_loop
```

To understand the effect of loop pipelining on performance, consider the following code example:

```
kernel void
foo(...)
{
  ...
  for (int i=0; i<3; i++) {
    int idx = get_global_id(0)*3 + i;
    op_Read(idx);
    op_Compute(idx);
    op_Write(idx);
  }
  ...
}
```

This kernel code has no attributes and is executed sequentially per the order of operations stated in the code fragment. The execution profile of this code is as shown in the following figure:

Figure 6–2: **Execution Profile without Pipelining**



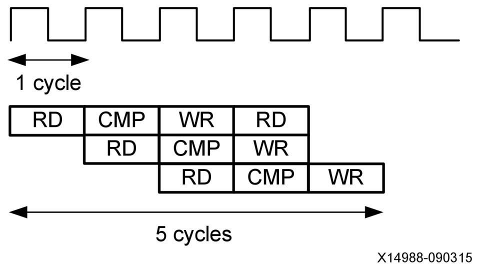
Assuming each operation in the loop takes one clock cycle, one loop iteration takes three clock cycles, and the complete loop takes nine clock cycles to complete. Although the execution is functionally correct, the implementation is not maximizing performance because the read, compute, and write stages of the example are not always busy.

The pipeline attribute serves as a command to the SDAccel™ compiler to maximize performance and minimize the idle time of any stage in the generated logic. The example code with the loop pipeline attribute looks like:

```
kernel void
foo(...)
{
  ...
  __attribute__((xcl_pipeline_loop))
  for (int i=0; i<3; i++) {
    int idx = get_global_id(0)*3 + i;
    op_Read(idx);
    op_Compute(idx);
    op_Write(idx);
  }
  ...
}
```

The execution profile after loop pipelining is as shown in the following figure.

Figure 6–3: **Execution Profile after Loop Pipelining**



The figure shows that each stage of the loop body is kept busy in a pipelined fashion. When analyzing the impact of loop pipelining, it is important to keep in mind the two cycle numbers shown in the Figure. The 1 clock cycle measurement is the amount of time that must pass between the start of loop iteration I and loop iteration I + 1. Without using unrolling to expose all possible concurrency, the best possible number that can be achieved for this metric is 1. The 5 clock cycle number is the overall latency to completely execute the loop nest. By adding loop pipelining to this example, the latency of the implementation is decreased by 45% without any code modification or additional hardware.

Work Item Pipelining

Work item pipelining is the extension of loop pipelining to the kernel work group. The syntax for the attribute for this optimization is:

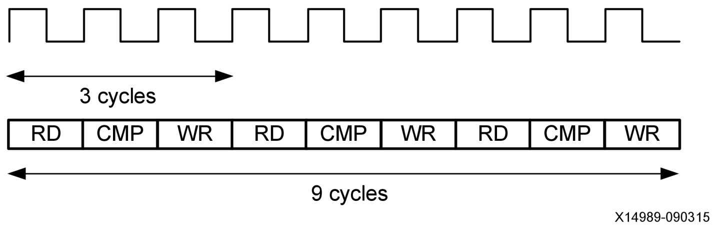
```
xcl_pipeline_workitems
```

An example of code where work pipelining can be applied is:

```
kernel
__attribute__ ((reqd_work_group_size(3,1,1)))
void foo(...)
{
  ...
    int tid = get_global_id(0);
    op_Read(tid);
    op_Compute(tid);
    op_Write(tid);
  }
  ...
}
```

To handle the `reqd_work_group_size` attribute, the SDAccel™ compiler automatically inserts a loop nest to handle the three-dimensional characteristics of the ND range. As a result of the loop nest added by the SDAccel compiler, the execution profile of this code is the same as that of an unpipelined loop (see the following figure).

Figure 6–4: **Work Item Execution without Pipelining**

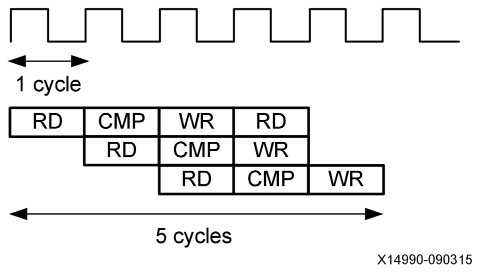


The work item pipeline attribute can be added to the code as:

```
kernel
__attribute__ ((reqd_work_group_size(3,1,1)))
void foo(...)
{
  ...
  __attribute__((xcl_pipeline_workitems)) {
    int tid = get_global_id(0);
    op_Read(tid);
    op_Compute(tid);
    op_Write(tid);
  }
  ...
}
```

As in the case of loop pipelining, the resulting hardware is kept busy all of the time, which maximizes performance, and results in the execution profile, as shown in the following figure:

Figure 6–5: **Work Item Execution with Pipelining**



Memory Architecture Optimization

Along with the compiler improvements available through the use of attributes, you can also increase the performance of an OpenCL™ application by modifying the memory bandwidth exposed from the system to the kernel. In the case of kernels mapped into the FPGA logic, The SDAccel™ development environment generates the custom hardware for the kernel computation and the connections to system-level memory. This key difference between the SDAccel and OpenCL compilers for fixed architectures such as a GPU allows you to take full advantage of the parallel nature of the FPGA architecture and the power benefits derived from it.

There are three memory optimizations available in the SDAccel compiler:

- Multiple Memory Ports per Kernel
- Adjustable Bitwidth for Memory Ports in a Kernel
- On-Chip Global Memories

Multiple Memory Ports per Kernel

The default behavior of the SDAccel™ compiler is to generate functionally correct FPGA kernels that consume the least amount of FPGA resources. This behavior produces the most area efficient kernels, but might not always achieve the application throughput requirement. The area efficient behavior of the SDAccel compiler is demonstrated by the following example:

```
__kernel
void example (__global float16 *A, __global float16 *B, __global float16 *C)
{
    int id = get_global_id(0);
    C[id] = A[id] * B[id];
}
```

The code above uses the float16 vector data type to encapsulate the multiplication of 16 values from buffer A by 16 values from buffer B. All 16 results are then stored into the appropriate locations in buffer C. The hardware implementation generated by the SDAccel compiler without additional help from the developer has a single port to global memory through which all accesses to buffers A, B, and C are scheduled. Although float16 is a vector type, the SDAccel compiler uses the base type float to determine the size of the generated memory interface port. In this case, the memory port is generated with a 32-bit width. During execution, the width of the port forces sequential access to global memory to fetch individual elements from buffers A and B to compute the result C.

The first technique to improve the performance of this code is to use the `pipeline_work_items` attribute from Work Item Pipelining. This compiler hint tells the SDAccel compiler that you want the multiplication operation in the work item to be scheduled as fast as possible to maximize use of the computation element. The application of work item pipelining is shown below:

```
__kernel
void example (__global float16 *A, __global float16 *B, __global float16 *C)
{
    __attribute__((xcl_pipeline_workitems)) {
     int id = get_global_id(0);
     C[id] = A[id] * B[id];
    }
}
```

Scheduling the multiplication operation to happen as quickly as possible requires simultaneous access to buffers A and B. The issue with this requirement is that the default behavior of the SDAccel compiler creates a single physical memory port for the kernel. The single physical port creates a bottleneck that forces sequential accesses to buffers A and B. The SDAccel compiler detects this system-level issue and alerts you with a warning message displayed on the console.

The message highlights that the throughput of the kernel is limited by the available memory bandwidth. The SDAccel compiler will not error out on this issue and only generates the warning message. In cases such as this, the SDAccel compiler automatically lowers the throughput of the application until an improved constraint set is implemented in hardware. This provides you with a valid working system that can be analyzed for performance bottlenecks.

One way of increasing the memory bandwidth available to a kernel is to increase the number of physical connections to memory that are attached to a kernel. Proper implementation of this optimization requires your knowledge of both the application and the target compute device. Therefore, the SDAccel compiler requires direct user intervention to increase the number of physical memory ports in a kernel. The SDAccel command to increase the number of physical memory ports available to the kernel is:

```
set_property max_memory_ports true [get_kernels <kernel name>]
```

The `max_memory_ports` property tells the SDAccel compiler to generate one physical memory interface for every global memory buffer declared in the kernel function signature. This command is only valid for kernels that have been placed into binaries that will be executed in the FPGA logic. There is no effect on kernels executing in a processor.

For the example in this chapter, the kernel function signature is:

```
void example (__global float16 *A, __global float16 *B, __global float16 *C)
```

which contains global memory buffers A, B, and C. The SDAccel command sequence to generate a kernel with three physical memory ports is:

```
create_kernel -type clc example
set_property max_memory_ports true [get_kernels example]
```

The first command in the sequence creates a kernel example in the current solution. The property on the kernel guides the SDAccel compiler to create a physical memory port for each buffer in the kernel code. This property applies to all compute units created for the example kernel and that are placed into a binary targeted for the FPGA logic. Each compute unit for the example kernel will have parallel access to buffers A, B, and C as a result of this kernel property.

Adjustable Bitwidth for Memory Ports in a Kernel

In addition to increasing the number of memory ports available to a kernel, you have the ability to change the bitwidth of the memory port. The benefit of modifying the bitwidth of the memory interface depends on the computation in the kernel. For this example, the core computation is:

```
C[id] = A[id] * B[id]
```

where A, B and C are of type float16. A float16 is a 16 element vector data type created from the fundamental C data type float. The SDAccel™ compiler uses the fundamental C data type when determining the default bitwidth of a memory interface. In this case, the memory interfaces have a bitwidth of 32 bits. Therefore, the kernel requires 16 memory transactions to read enough data to complete the computation of a single work item. You can override the default behavior of the SDAccel compiler with the following kernel property command:

```
set_property memory_port_data_width <bitwidth> [get_kernels <kernel name>]
```

The bitwidths currently supported by the SDAccel compiler are 32, 64, 128, 256, and 512 bits. In cases where your defined bitwidth does not match the bitwidth declared in the kernel source code, the SDAccel compiler handles all data width conversions between the physical interface size and the data type in the kernel source code. This optimization is only supported for kernels mapped for execution in the FPGA logic.

The command sequence to enable the memory port bitwidth optimization is:

```
create_kernel -type clc example
set_property max_memory_ports true [get_kernels example]
set_property memory_port_data_width 512 [get_kernels example]
```

The first command in the sequence creates a kernel example in the current solution. The property on the kernel guides the SDAccel compiler to create a physical memory port for each buffer in the kernel code. This property applies to all compute units created for the example kernel and that are placed into a binary targeted for the FPGA logic. Each compute unit for the example kernel will have parallel access to buffers A, B, and C as a result of this kernel property. In addition, the second property specified on the kernel sets the bitwidth of every physical memory port to 512 bits. As in the case of the `max_memory_port` property, changes to the `memory_port_data_width` property affect all compute units generated for a given kernel. The memory data width change reduces the number of memory transactions per buffer from 16 to 1 for every work item invocation.

IMPORTANT: *Changes to the bitwidth of a kernel physical memory interface must always be specified after all memory ports have been generated. Kernels accessing more than one global memory buffer should set the max_memory_ports property before the bitwidth property is set.*

On-Chip Global Memories

Another memory architectural optimization that is available in the SDAccel™ development environment deals with global memories that are used to pass data between kernels. In cases where the global memory buffer used for inter-kernel communication does not need to be visible to the host processor, the SDAccel environment enables you to move the buffer out of DDR based memory and into the FPGA logic. This optimization is called on-chip global memory buffers and is part of the OpenCL™ 2.0 specification.

The on-chip global memory buffer optimization makes use of the block memory instances embedded in the FPGA logic to create a memory buffer that is only visible to the kernels accessing the buffer. The following code example illustrates the usage model for global memory buffers that is suitable for the on-chip global memory buffer optimization.

```
// Global memory buffers used to transfer data between kernels
// Contents of the memory do not need to be accessed by host processor
global int g_var0[1024];
global int g_var1[1024];

// Kernel reads data from global memory buffer written by the host processor
// Kernel writes data into global buffer consumed by another kernel
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void input_stage (global int *input)
{
  __attribute__((xcl_pipeline_workitems)) {
  g_var0[get_local_id(0)] = input[get_local_id(0)];
  }
}

// Kernel computes a result based on data from the input_stage kernel
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void adder_stage(int inc)
{
  __attribute__ ((xcl_pipeline_workitems)) {
  int input_data, output_data;
  input_data = g_var0[get_local_id(0)];
  output_data = input_data + inc;
  g_var1[get_local_id(0)] = output_data;
  }
}

// Kernel writes the results computed by the adder_stage to
// a global memory buffer that is read by the host processor
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void output_state(global int *output)
{
  __attribute__ ((xcl_pipeline_workitems)) {
  output[get_local_id(0)] = g_var1[get_local_id(0)];
}
}
```

In the code example above, the input_stage kernel reads the contents of global memory buffer input and writes them into global memory buffer `g_var0`. The contents of buffer `g_var0` are used in a computation by the `adder_stage` kernel and stored into buffer `g_var1`. The contents of `g_var1` are then read by the `output_stage` kernel and stored into the output global memory buffer. Although both `g_var0` and `g_var1` are declared as global memories, the host processor only needs to have access to the input and output buffers. Therefore, for this application to run correctly the host processor must only be involved in setting up the input and output buffers in DDR based memory. Because buffers `g_var0` and `g_var1` are only used for inter-kernel communication, the accesses to these buffers can be removed from the system-level memory bandwidth requirements. The SDAccel environment automatically analyzes this kind of coding style to infer that both `g_var0` and `g_var1` can be implemented as on-chip memory buffers. The only requirements on the memories are that all kernels with access to the on-chip memory are executed in the FPGA logic and that the memory has at least 4096 bytes. For example, in an array of int data type, the minimum array size needs to be 1024.

IMPORTANT: *On-chip global memory optimization is an automatic optimization in the SDAccel environment that is applied to memory buffers that are only accessed by kernels executed in the FPGA logic.*

Pipes

The OpenCL™ 2.0 specification introduces a new memory object called pipe. A pipe stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. Pipes can be used to stream data from one kernel to another inside the FPGA device without having to use the external memory, which greatly improves the overall system latency.

In the SDAccel™ development environment, pipes must be statically defined at file scope. Dynamic pipe allocation using the OpenCL 2.x clCreatePipe API is not currently supported. The depth of a pipe must be specified by using the xcl_reqd_pipe_depth attribute in the pipe declaration. The valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
```

IMPORTANT: *A given pipe, can have one and only one producer and consumer in different kernels.*

Pipes can be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode or using Xilinx extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode. The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions. Please see *The OpenCL C Specification, Version 2.0* from Khronos Group for more details on these built-in functions. Other pipe built-in functions are not currently supported in the SDAccel environment.

The following are the function signatures for the pipe functions, where gentype indicates the built-in OpenCL C scalar integer or floating-point data types.

```
int read_pipe (pipe gentype p, gentype *ptr)
int write_pipe (pipe gentype p, const gentype *ptr
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
uint get_pipe_num_packets (pipe gentype p)
uint get_pipe_max_packets (pipe gentype p)
```

The following is a complete OpenCL C example using pipes to pass data from one processing stage to the next using non-blocking read_pipe() and write_pipe() built-in functions:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(512)));
// both read_pipe and write_pipe are non-blocking. Keep trying until they succeed.
// Stage 1 kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void input_stage(__global int *input)
{
  int write_status = -1;
  while (write_status != 0)
    write_status = write_pipe(p0, &input[get_local_id(0)]);
}
// Stage 2 kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void adder_stage(int inc)
{
  int input_data, output_data;
  int read_status = -1;
  int write_status = -1;
  while (read_status != 0)
    read_status = read_pipe(p0, &input_data);
  output_data = input_data + inc;
  while (write_status != 0)
    write_status = write_pipe(p1, &output_data);
}
// Stage 3 kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void output_stage(__global int *output)
{
  int read_status = -1;
  while (read_status != 0)
    read_status = read_pipe(p1, &output[get_local_id(0)]);
}
```

The following is a complete OpenCL C example using pipes to pass data from one processing stage to the next using blocking read_pipe_block() and write_pipe_block() functions:

```
 pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(512)));
// Stage 1
kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void input_stage(__global int *input)
{
  write_pipe_block(p0, &input[get_local_id(0)]);
}
// Stage 2
kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void adder_stage(int inc)
{
  int input_data, output_data; read_pipe_block(p0, &input_data);
  output_data = input_data + inc;
  write_pipe_block(p1, &output_data);
}
// Stage 3
kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void output_stage(__global int *output)
{
   read_pipe_block(p1, &output[get_local_id(0)]);
}
```

Partitioning Memories Inside of Compute Units

One of the advantages of the FPGA over other compute devices for OpenCL™ programs is the ability for the application programmer to customize the memory architecture all throughout the system and into the compute unit. By default, The SDAccel™ compiler generates a memory architecture within the compute unit that maximizes local and private memory bandwidth based on static code analysis of the kernel code. Further optimization of these memories is possible based on attributes in the kernel source code, which can be used to specify physical layouts and implementations of local and private memories. The attribute in the SDAccel compiler to control the physical layout of memories in a compute unit is `array_partition`.

```
__attribute__((xcl_array_partition(<partition type>,
                                   <partition factor>,
                                   <array dimension>)))
```

The `array_partition` attribute implements an array declared within kernel code as multiple physical memories instead of a single physical memory. The selection of which partitioning scheme to use depends on the specific application and its performance goals. The array partitioning schemes available in the SDAccel compiler are cyclic, block, and complete.

Cyclic Partitioning

Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The process of separating the original array into smaller physical memories is similar to the process of dealing a deck of cards among N players. The attribute for cyclic partitioning is

```
__attribute__((xcl_array_partition(cyclic,
                                   <partition factor>,
                                   <array dimension>)))
```

When the `array_partition` attribute is used in cyclic mode, the partition factor specifies among how many physical memories or players to split the original array in the kernel code. The array dimension parameter specifies on which dimension of the array to apply the portioning. The SDAccel™ development environment supports arrays of N dimensions and can modify the physical mapping of an array on any dimension.

For example, consider the following array declaration:

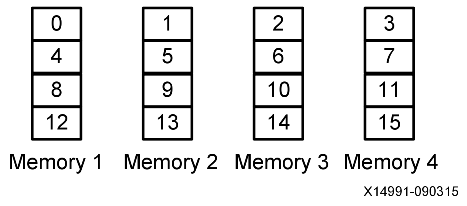
```
__local int buffer[16];
```

The array named buffer stores 16 values that are 32-bits wide each. Cyclic partitioning can be applied to this buffer as follows:

```
__local int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));
```

The cyclic partitioning attribute above tells the SDAccel environment to distribute the contents of buffer among four physical memories. The allocation of buffer elements, which follows the card dealing paradigm, is shown in the following figure.

Figure 7–1: **Physical Layout of Buffer After Cyclic Partitioning**



This attribute increases the immediate memory bandwidth for operations accessing the array buffer by a factor of four. All arrays inside of a compute unit in the context of the SDAccel environment are capable of sustaining a maximum of two concurrent accesses. By dividing the original array in the code into four physical memories, the resulting compute unit can sustain a maximum of eight concurrent accesses to the array buffer.

Block Partitioning

Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside of the compute unit. This type of partitioning is expressed in the kernel code by the following attribute:

```
__attribute__((xcl_array_partition(block,
                                   <partition factor>,
                                   <array dimension>)))
```

When the `array_partition` attribute is used in block mode, the partition factor specifies the number of elements from the original array to store in each physical memory. The number of physical memories created by this optimization is given by the division of the original array size by the block size stated in the attribute. For example, consider the following array declaration:

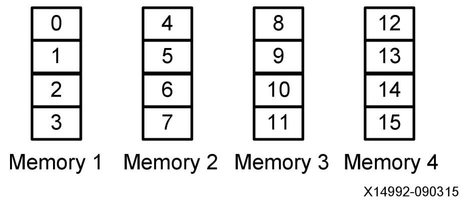
```
__local int buffer[16];
```

This is the same array as in the cyclic partitioning example. Apply block partitioning as follows:

```
___local int buffer[16] __attribute__((xcl_array_partition(block,4,1)));
```

Because the size of the block is four, the SDAccel™ development environment will generate four physical memories with the data distribution shown in the following figure.

Figure 7–2: **Physical Layout of Buffer After Block Partitioning**



For this example, both block and cyclic partitioning result in the same number of physical memories. Both schemes deliver a theoretical maximum of eight concurrent memory transactions. The choice of which to employ depends on the data access pattern of the compute unit. The SDAccel environment will always build the partitioning scheme stated by the attribute, but does not guarantee maximum concurrency of memory operations. The SDAccel environment schedules memory operations by taking into account both the physical layout of data and the access pattern described in the algorithm to ensure fidelity to the intent of the code and avoid data corruption issues. Therefore, depending on the functionality captured in the kernel, one partitioning scheme might deliver higher performance than the other at the same cost in terms of FPGA resources.

Complete Partitioning

Complete partitioning is the physical implementation of a buffer in the kernel code as independent registers within the custom logic of the compute unit. This type of partitioning is expressed in the kernel code by the following attribute:

```
__attribute__((xcl_array_partition(complete, <array dimension>)))
```

When the array partition attribute is used in complete mode, the array in the kernel code is broken up into individual registers that are embedded into the data path of the compute unit. The number of individual registers created by the SDAccel™ development environment corresponds to the number of entries in the array declaration. For example, consider the following array declaration:

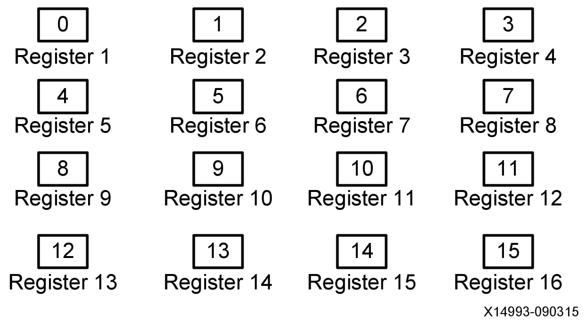
```
__local int buffer[16];
```

This array, which is the same one used in the examples for block and cyclic partitioning, can be qualified for complete partitioning as follows:

```
__local int buffer[16] __attribute__((xcl_array_partition(complete,1)));
```

The resulting memory layout into individual registers is shown in the following figure.

Figure 7–3: **Physical Layout of Buffer After Complete Partitioning**



For this example, complete partitioning results in 16 independent registers. While this creates an implementation with the highest possible memory bandwidth, it is not applicable to all applications. The way in which data is accessed by the kernel code through either constant or data dependent indexes affects the amount of supporting logic that the SDAccel environment has to build around each register to ensure functional equivalence with the usage in the original code. As a general best practice guideline for the SDAccel environment, the complete partitioning attribute is best suited for arrays in which at least one dimension of the array is accessed through the use of constant indexes.

Multiple DDR Banks

For applications requiring very high bandwidth to the global memory, devices with multiple DDR banks can be targeted so that kernels can access all available memory banks simultaneously. For example, SDAccel includes a device support archive (DSA) xilinx:adm-pcieku3:2ddr:2.1 that supports two DDR banks.

In order to take advantage of multiple DDR banks, users need to assign CL memory buffers to different banks in the host code as well as configure XCL binary file to match the bank assignment in compilation script or xocc command line.

Bank assignment in host code is supported by Xilinx vendor extension. The following code snippet shows the header file required as well as assigning a CL memory buffer to DDR bank0:

```
#include <CL/cl_ext.h>
…
int main(int argc, char** argv)
{
…
  int a[DATA_SIZE];
  cl_mem input_a;
  cl_mem_ext_ptr_t input_a_ext;

  input_a_ext.flags = XCL_MEM_DDR_BANK0;
  input_a_ext.obj = a;
  input_a_ext.param = 0;

  input_a = clCreateBuffer(context,
    CL_MEM_READ_ONLY  | CL_MEM_USE_HOST_PTR | CL_MEM_EXT_PTR_XILINX,
    sizeof(int)*DATA_SIZE,
    &input_a_ext,
    NULL);
…
}

cl_mem_ext_ptr_t is a struct as defined below:
  typedef struct{
    unsigned flags;
    void *obj;
    void *param;
  } cl_mem_ext_ptr_t;
```

- Valid values for flags are `XCL_MEM_DDR_BANK0`, `XCL_MEM_DDR_BANK1`, `XCL_MEM_DDR_BANK2`, and `XCL_MEM_DDR_BANK0`.
- `*obj` is the pointer to the associated host memory allocated for the CL memory buffer if `CL_MEM_USE_HOST_PTR` flag is passed to `clCreateBuffer` API.
- `*param` is reserved for future use. Always assign it to 0 or NULL.

For XCL binary compiled in script flow, `map_connect` command is used to connect different global memory ports to different DDR banks. The code snippet below creates one memory port for each kernel pointers and then connect pointer 0 and 2 to DDR bank0 and pointer 1 to DDR bank1.

```
set_property max_memory_ports true [get_kernel mmult]
map_connect  -opencl_binary [get_opencl_binary bin_mmult] \
    -src_type "kernel" -src_name "mmult0"       -src_port "M_AXI_GMEM0" \
    -dst_type "core"   -dst_name "OCL_REGION_0" -dst_port "M00_AXI"

map_connect  -opencl_binary [get_opencl_binary bin_mmult] \
    -src_type "kernel" -src_name "mmult0"       -src_port "M_AXI_GMEM1" \
    -dst_type "core"   -dst_name "OCL_REGION_0" -dst_port "M01_AXI"

map_connect  -opencl_binary [get_opencl_binary bin_mmult] \
    -src_type "kernel" -src_name "mmult0"       -src_port "M_AXI_GMEM2" \
    -dst_type "core"   -dst_name "OCL_REGION_0" -dst_port "M00_AXI"
```

For XCL binary compiled using xocc, in script flow, DDR bank configuration is passed to the xocc command as extra parameters (`--xp` option). The command line example below creates one memory port for each kernel pointers and then connect pointer 0 and 2 to DDR bank0 and pointer 1 to DDR bank1.

```
xocc --xdevice xilinx:adm-pcie-ku3:2ddr:2.1 \
    --xp misc:map_connect=add.kernel.mmult_1.M_AXI_GMEM0.core.OCL_REGION_0.M00_AXI \
    --xp misc:map_connect=add.kernel.mmult_1.M_AXI_GMEM1.core.OCL_REGION_0.M01_AXI \
    --xp misc:map_connect=add.kernel.mmult_1.M_AXI_GMEM2.core.OCL_REGION_0.M00_AXI \
    --max_memory_ports all \
    mmult1.cl
```

Application Debug in the SDAccel Environment

There are four steps to debugging applications in The SDAccel™ development environment:

1. Prepare Host Application for Debug
2. Prepare Kernel Code for Debug in CPU Emulation Flow
3. Launch GDB in SDAccel Environment to Debug Host and Kernel Programs
4. Launch GDB Standalone to Debug Host Program with Kernel Running on FPGA

The SDAccel environment supports host program debugging in all flows and kernel debugging in the CPU emulation flow with integrated GDB in command line mode.

Prepare Host Application for Debug

The host program needs to be compiled with debugging information generated in the executable by adding the `–g` option to the `host_cflags` property in the SDAccel™ compilation script:

```
set_property -name host_cflags -value {-g} -objects [current_solution]
```

Prepare Kernel Code for Debug in CPU Emulation Flow

Kernel code can be debugged together with the host program in the CPU emulation flow using integrated GDB. Debugging information needs to be generated first in the binary container by passing the `–g` option to the `kernel_flags` property on the kernel before it can be debugged.

```
set_property -name kernel_flags -value {-g} -objects [get_kernels smithwaterman]
```

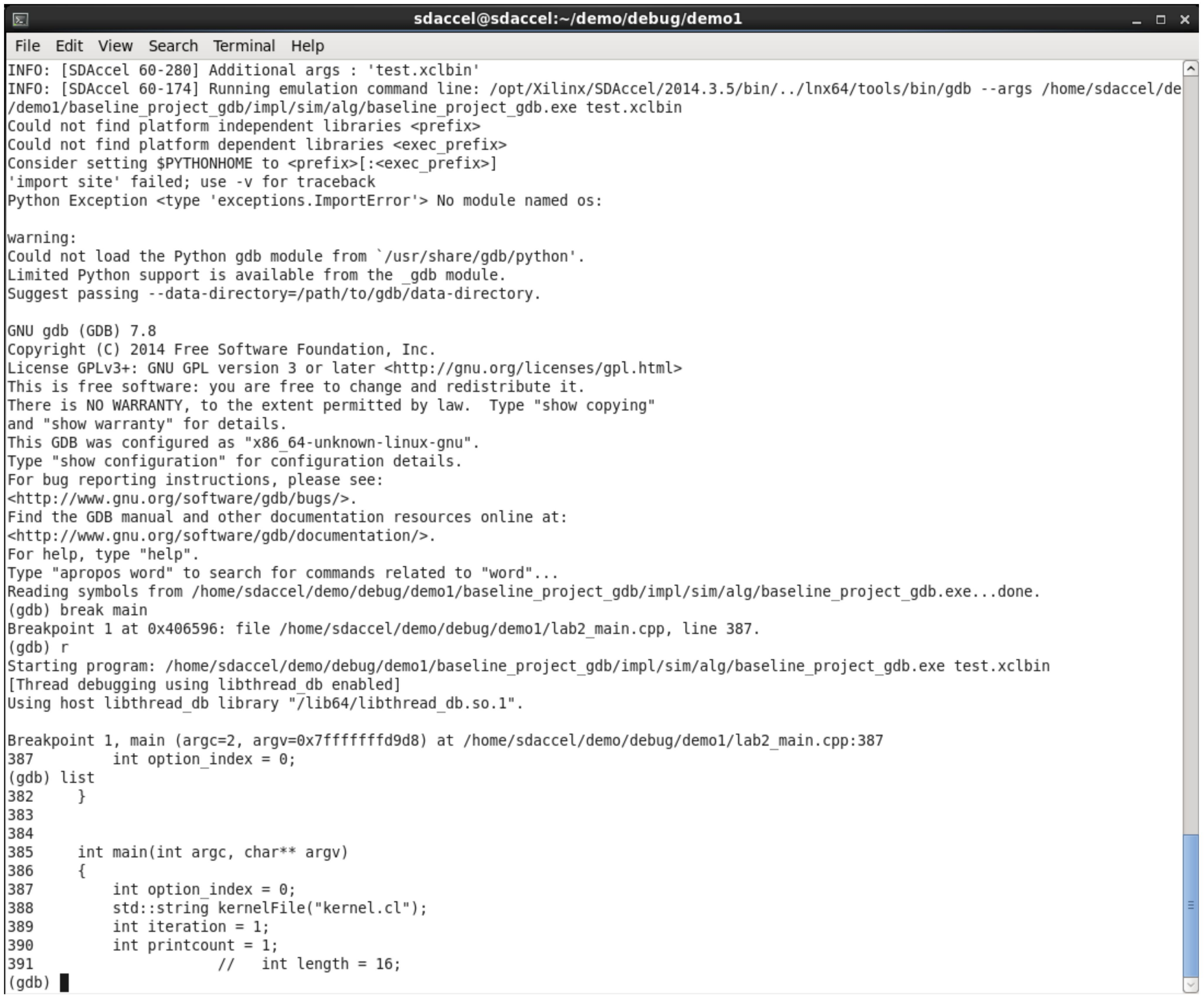
Launch GDB in SDAccel Environment to Debug Host and Kernel Programs

After host and kernel compilation are completed, GDB can be launched from the SDAccel™ environment by passing the `–debug` option to the run_`emulation` command. Below are examples for the CPU and hardware emulation flows respectively.

```
run_emulation -debug -flow cpu -args "test.xclbin"
run_emulation -debug -flow hardware -args "test.xclbin"
```

The following figure shows the GDB terminal window after the `run_emulation` command is executed. Refer to the GDB project web site on GDB commands and usages.

***Figure 8–1:* GDB Terminal Window**


sdaccel@sdaccel:~/demo/debug/demo1
File Edit View Search Terminal Help

```
INFO: [SDAccel 60-280] Additional args : 'test.xclbin'
INFO: [SDAccel 60-174] Running emulation command line: /opt/Xilinx/SDAccel/2014.3.5/bin/../lnx64/tools/bin/gdb --args /home/sdaccel/de
/demo1/baseline_project_gdb/impl/sim/alg/baseline_project_gdb.exe test.xclbin
Could not find platform independent libraries <prefix>
Could not find platform dependent libraries <exec_prefix>
Consider setting $PYTHONHOME to <prefix>[:<exec_prefix>]
'import site' failed; use -v for traceback
Python Exception <type 'exceptions.ImportError'> No module named os:

warning:
Could not load the Python gdb module from `/usr/share/gdb/python'.
Limited Python support is available from the _gdb module.
Suggest passing --data-directory=/path/to/gdb/data-directory.

GNU gdb (GDB) 7.8
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /home/sdaccel/demo/debug/demo1/baseline_project_gdb/impl/sim/alg/baseline_project_gdb.exe...done.
(gdb) break main
Breakpoint 1 at 0x406596: file /home/sdaccel/demo/debug/demo1/lab2_main.cpp, line 387.
(gdb) r
Starting program: /home/sdaccel/demo/debug/demo1/baseline_project_gdb/impl/sim/alg/baseline_project_gdb.exe test.xclbin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0x7fffffffd9d8) at /home/sdaccel/demo/debug/demo1/lab2_main.cpp:387
387         int option_index = 0;
(gdb) list
382     }
383
384
385     int main(int argc, char** argv)
386     {
387         int option_index = 0;
388         std::string kernelFile("kernel.cl");
389         int iteration = 1;
390         int printcount = 1;
391                 //   int length = 16;
(gdb) 
```

X15006-091115

The kernel function can be debugged like a regular C function inside GDB. Note that GDB might ask "`Make breakpoint pending on future shared library load?`" when you set a breakpoint on kernel functions. Answer `y`, and press **Enter**. See the following example.

```
(gdb) break smithwaterman
Function "smithwaterman" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (smithwaterman) pending.
(gdb) r
```

Launch GDB Standalone to Debug Host Program with Kernel Running on FPGA

When running the SDAccel™ host program with the kernel running on an FPGA, the host program is generally packaged using the `package_system` command and often generated on a different machine. GBD can be launched standalone to debug the host program in this case if the host machine has the same version of the SDAccel development environment installed, and the host program was built with debug information (built with the `-g` flag).

Below are the instructions:

1. Set up your SDAccel development environment by following instructions in the *SDAccel Development Environment Installation and Licensing Guide* (UG1020).
2. Launch GDB from the SDAccel installation on the host program as in the following example:

```
<SDAccel_install_dir>/lnx64/tools/bin/gdb --args host.exe test.xclbin
```

Use printf() for Debugging Kernels

The SDAccel™ development environment supports OpenCL™ `printf()` kernel built-in function in all development flows: CPU emulation, hardware emulation, and running kernel in actual hardware.

Below is a kernel example of using `printf()` and the output when the kernel is executed with global size of 8:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);

     printf("Hello world from work item %d\n", idx);
     a[idx] = idx;
}
```

Output is as follows:

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```

Host Application Profiling in the SDAccel Runtime

The SDAccel™ runtime automatically collects profiling data on host applications. After the host application finishes execution, the profile summary is saved in both HTML and CSV formats in the solution report directory or working directory. This can be reviewed in a web browser or Microsoft Excel or OpenOffice.

- When running CPU emulation using the `run_emulation` command in SDAccel environment, the profile summary reports are as follows:

  ```
  <solution_name>/rpt/profile_summary_cpu_em.html
  <solution_name>/rpt/profile_summary_cpu_em.csv
  ```

- When running hardware emulation using the `run_emulation` command in the SDAccel environment, the profile summary reports are as follows:

  ```
  <solution_name>/rpt/profile_summary_hw_emu.html
  <solution_name>/rpt/profile_summary_hw_emu.csv
  ```

- When running an application on hardware the using the `run_system` command in the SDAccel environment, the profile summary reports are as follows:

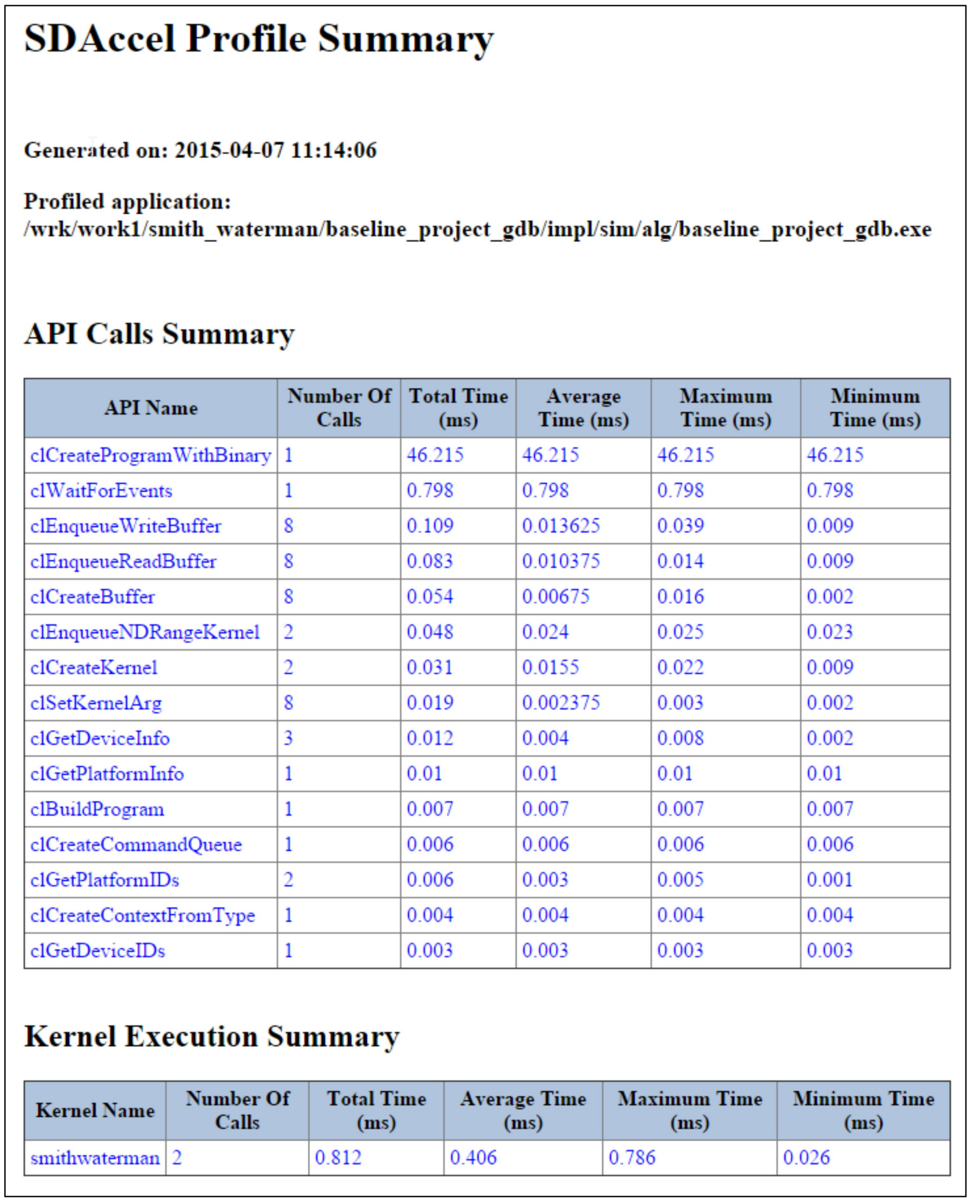
  ```
  <solution_name>/rpt/profile_summary_hw.html
  <solution_name>/rpt/profile_summary_hw.csv
  ```

- When running an application standalone in all flows outside the SDAccel environment, the following profile summary reports are generated in the directory where the application is executed:

  ```
  <working_directory>/sdaccel_profile_summary.html
  <working_directory>/sdaccel_profile_summary.csv
  ```

The following figures show the SDAccel Profile Summary in a web browser:

Figure 9–1: **SDAccel Profile Summary - Part 1**



SDAccel Profile Summary

Generated on: 2015-04-07 11:14:06

Profiled application:
/wrk/work1/smith_waterman/baseline_project_gdb/impl/sim/alg/baseline_project_gdb.exe

API Calls Summary

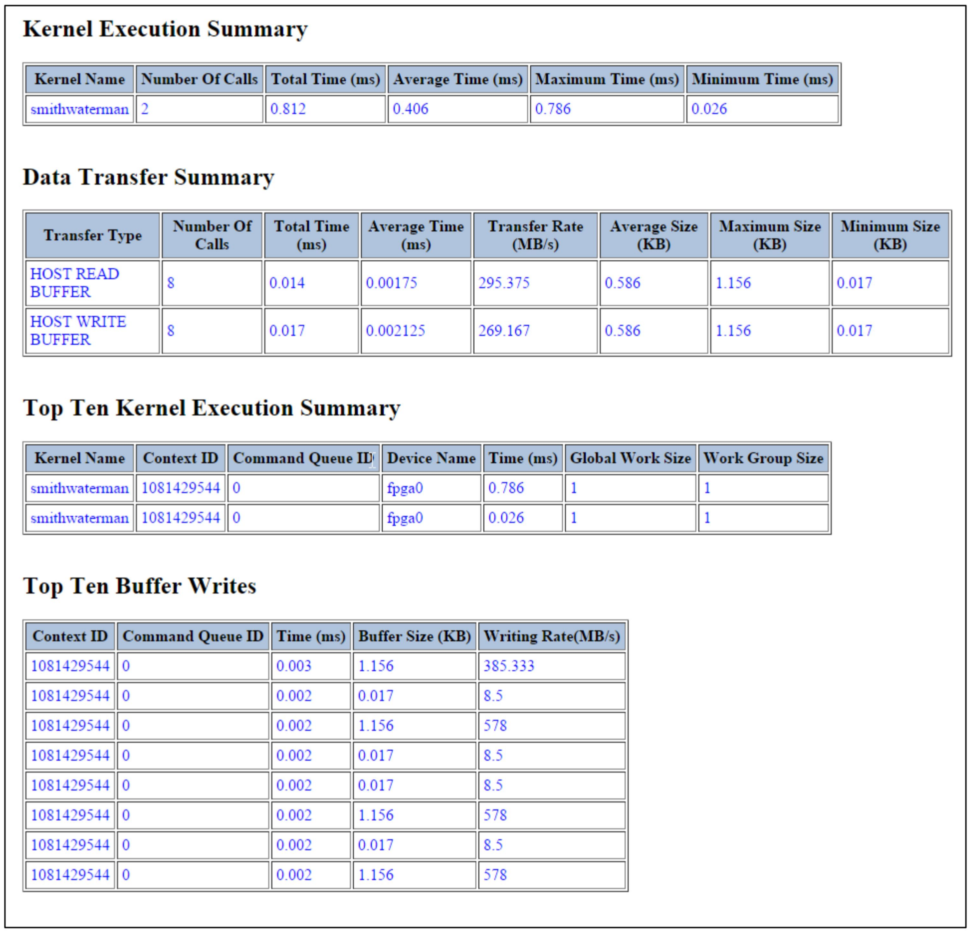
API Name	Number Of Calls	Total Time (ms)	Average Time (ms)	Maximum Time (ms)	Minimum Time (ms)
clCreateProgramWithBinary	1	46.215	46.215	46.215	46.215
clWaitForEvents	1	0.798	0.798	0.798	0.798
clEnqueueWriteBuffer	8	0.109	0.013625	0.039	0.009
clEnqueueReadBuffer	8	0.083	0.010375	0.014	0.009
clCreateBuffer	8	0.054	0.00675	0.016	0.002
clEnqueueNDRangeKernel	2	0.048	0.024	0.025	0.023
clCreateKernel	2	0.031	0.0155	0.022	0.009
clSetKernelArg	8	0.019	0.002375	0.003	0.002
clGetDeviceInfo	3	0.012	0.004	0.008	0.002
clGetPlatformInfo	1	0.01	0.01	0.01	0.01
clBuildProgram	1	0.007	0.007	0.007	0.007
clCreateCommandQueue	1	0.006	0.006	0.006	0.006
clGetPlatformIDs	2	0.006	0.003	0.005	0.001
clCreateContextFromType	1	0.004	0.004	0.004	0.004
clGetDeviceIDs	1	0.003	0.003	0.003	0.003

Kernel Execution Summary

Kernel Name	Number Of Calls	Total Time (ms)	Average Time (ms)	Maximum Time (ms)	Minimum Time (ms)
smithwaterman	2	0.812	0.406	0.786	0.026

X15005-091115

Figure 9–2: **SDAccel Profile Summary - Part 2**



Kernel Execution Summary

Kernel Name	Number Of Calls	Total Time (ms)	Average Time (ms)	Maximum Time (ms)	Minimum Time (ms)
smithwaterman	2	0.812	0.406	0.786	0.026

Data Transfer Summary

Transfer Type	Number Of Calls	Total Time (ms)	Average Time (ms)	Transfer Rate (MB/s)	Average Size (KB)	Maximum Size (KB)	Minimum Size (KB)
HOST READ BUFFER	8	0.014	0.00175	295.375	0.586	1.156	0.017
HOST WRITE BUFFER	8	0.017	0.002125	269.167	0.586	1.156	0.017

Top Ten Kernel Execution Summary

Kernel Name	Context ID	Command Queue ID	Device Name	Time (ms)	Global Work Size	Work Group Size
smithwaterman	1081429544	0	fpga0	0.786	1	1
smithwaterman	1081429544	0	fpga0	0.026	1	1

Top Ten Buffer Writes

Context ID	Command Queue ID	Time (ms)	Buffer Size (KB)	Writing Rate(MB/s)
1081429544	0	0.003	1.156	385.333
1081429544	0	0.002	0.017	8.5
1081429544	0	0.002	1.156	578
1081429544	0	0.002	0.017	8.5
1081429544	0	0.002	0.017	8.5
1081429544	0	0.002	1.156	578
1081429544	0	0.002	0.017	8.5
1081429544	0	0.002	1.156	578

X15004-091115

The profile summary includes a number of useful statistics for your OpenCL™ application. This can provide you with a general idea of the functional bottlenecks in your application. The profile summary consists of six sections:

- **API Calls Summary** - This section displays the profile data for all OpenCL host API function calls executed in the host application.
- **Kernel Execution Summary** - This section displays the profile data for all kernel functions scheduled and executed by the host application.
- **Data Transfer Summary** - This sections displays the profile data for all read and write transfers between the host and device memory via PCIe® link.
- **Top Ten Kernel Execution Summary** - This section displays the profile data for top 10 kernels in terms of execution time. The table is ordered from the longest to shortest execution time.
- **Top Ten Buffer Writes** - This section displays the profile data for top 10 write transfers from the host to the device memory in terms of execution time. The table is ordered from the longest to shortest execution time.
- **Top Ten Buffer Reads** - This section displays the profile data for top 10 read transfers from the device memory to the host memory in terms of execution time. The table is ordered from the longest to shortest execution time.

Xilinx OpenCL Compiler (xocc)

The Xilinx® OpenCL™ Compiler (xocc) is a standalone command line utility for compiling an OpenCL kernel supporting all flows in the SDAccel™ environment. It provides a mechanism for command line users to compile their kernels without writing any Tcl script, which is ideal for compiling host applications and kernels using a makefile.

Following are details of xocc command line format and options.

Syntax:

```
xocc [options] <input_file>
```

Options	Valid Values	Description
`--xdevice <arg>`	Supported acceleration devices by Xilinx and third-party board partners	Required Set target Xilinx device. The following devices are installed with the SDAccel environment: xilinx:adm-pcie-7v3:1ddr:2.1: ADM-PCIE-7V3 card xilinx:adm-pcie-ku3:1ddr:2.1: ADM-PCIE-KU3 card with one DDR3 interface xilinx:adm-pcie-ku3:2ddr:2.1: ADM-PCIE-KU3 card with two DDR3 interfaces
`-t <arg>`	[`sw_emu` | `hw_emu` | `hw`]	Specify a compile target. • `sw_emu`: CPU emulation • `hw_emu`: Hardware emulation • `hw`: Hardware Default: `hw`
`-o <arg>`	File name with `.xo` or `.xclbin` extension depending on mode	Optional Set output file name. Default: `a.xo` for compile mode `a.xclbin` for link and build mode
`-c`	N/A	Optional Run xocc in compile mode and generate `.xo` file. ***NOTE:*** Without the `-c` or `-l` option, xocc is run in build mode, and an `.xclbin` file is generated.

Options	Valid Values	Description
`-l`	N/A	Optional Run xocc in link mode, link to `.xo` input files, and generate `.xclbin` file. ***NOTE:*** Without the `-c` or `-l` option, xocc is run in build mode, and an `.xclbin` file is generated.
`-k <arg>`	Kernel to be compiled from the input `.cl` or `.c/.cpp` kernel source code	Required for C/C++ kernels Optional for OpenCL kernels Compile/build only the specified kernel from the input file. Only one `-k` option is allowed per command. ***NOTE:*** When an OpenCL kernel is compiled without the `-k` option, all the kernels in the input file are compiled.

`--nk <arg>`	`<kernel_name>:`
	`<compute_units>`
	
	(for example, `foo:2`)
		N/A in compile mode
		
		Optional in link mode
		
		Instantiate the specified number of compute units for the given kernel in the `.xclbin` file.
		
		Default: One compute unit per kernel

`--max_memory_ports <arg>`	[`all` | `<kernel_name>`]	Optional Set the maximum memory port property for all kernels or a given kernel.
`--memory_port_data_width <arg>`	[`all` | `<kernel_name>`]:`<width>`	Set the specified memory port data width for all kernels or a given kernel. Valid width values are 32, 64, 128, 256, and 512.
`--report <arg>`	Supported report types [estimate]	Generate a report type specified by `<arg>`. estimate: Generate estimate report in `report_estimate.xtxt`
`-D <arg>`	Valid macro name and definition pair `<name>=<definition>`	Predefine name as a macro with definition. This option is passed to the openCL preprocessor.
`-I <arg>`	Directory name that includes required header files	Add the directory to the list of directories to be searched for header files. This option is passed to the SDAccel compiler preprocessor.
`-h`	N/A	Print help.
`-s`	N/A	Save intermediate files.

Options	Valid Values	Description
`-g`	N/A	Generate code for debugging.
`-j <arg>`	Number of parallel jobs	Optional Specify the number of parallel jobs to pass to Vivado implementation.
`--lsf <arg>`	`bsub` command line to pass to LSF cluster ***NOTE:*** This argument is required.	Optional Use IBM Platform Load Sharing Facility (LSF) for Vivado implementation.
`input file`	OpenCL or C/C++ kernel source file	Compile kernels into a `.xo` or `.xclbin` file depending on the xocc mode.

Following is a makefile example showing how to compile host applications outside the SDAccel environment, and use xocc to compile kernels for CPU and hardware emulation as well as targeting hardware.

IMPORTANT: *g++ from SDAccel installation directory is required to compile host applications outside SDAccel environment.*

`<SDAccel_installation_path>/lnx64/tools/gcc/bin/g++`

```
VPATH = ./

#supported flow: cpu_emu, hw_emu, hw
FLOW=hw
HOST_SRCS = main.c
HOST_EXE = hello_world
HOST_CFLAGS = -DFPGA_DEVICE -g -Wall -I${XILINX_OPENCL}/runtime/include/1_2
HOST_LFLAGS = -L${XILINX_OPENCL}/runtime/lib/x86_64 -lxilinxopencl

KERNEL_SRCS = hello_world.cl
KERNEL_DEFS =
KERNEL_INCS =
#set target device for XCLBIN
#Set target Xilinx device. Devices below are installed with SDAccel installation:
XDEVICE=xilinx:adm-pcie-7v3:1ddr:2.0
KEEP_TEMP=
KERNEL_DEBUG=

CC = g++
CLCC = xocc
CLCC_OPT = --xdevice ${XDEVICE} -o ${XCLBIN} ${KERNEL_DEFS} ${KERNEL_INCS}

ifeq (${FLOW},cpu_emu)
    CLCC_OPT += -t sw_emu
    XCLBIN = hello_world_cpu_emu.xclbin
else ifeq (${FLOW},hw_emu)
    CLCC_OPT += -t hw_emu
    XCLBIN = hello_world_hw_emu.xclbin
else ifeq (${FLOW},hw)
    XCLBIN = hello_world_hw.xclbin
    CLCC_OPT += -t hw
endif

ifeq (${KEEP_TEMP},1)
    CLCC_OPT += -s
endif

ifeq (${KERNEL_DEBUG},1)
    CLCC_OPT += -g
endif
```

```
HOST_ARGS = ${XCLBIN}
OBJECTS := $(HOST_SRCS:.c=.o)

.PHONY: all

all:

run : host xbin
  ${HOST_EXE} ${HOST_ARGS}

host: ${HOST_EXE}

xbin : ${XCLBIN}

${HOST_EXE}: ${OBJECTS}
  ${CC} ${HOST_LFLAGS} ${OBJECTS} -o $@

${XCLBIN}:
  ${CLCC} ${CLCC_OPT} ${KERNEL_SRCS}

%.o: %.c
  ${CC} ${HOST_CFLAGS} -c $< -o $@

clean:
${RM} ${HOST_EXE} ${OBJECTS} ${XCLBIN}
```

Chapter 11

RTL Kernels

Interface Requirements

The following signals and interfaces are required on the top level of an RTL block

- Clock.
- Active Low reset.
- 1 or more AXI memory mapped (MM) master interfaces for global memory.

  You are responsible for partitioning global memory spaces. Each partition in the global memory becomes a kernel argument. The memory offset for each partition must be set by a control register programmable via the AXI MM Slave Lite interface.
- One and only one AXI MM slave lite I/F for control interface.

  Offset 0 of the AXI MM slave lite must have the following signals:
  - Bit 0: start signal - The kernel starts processing data when this bit is set.
  - Bit 1: done signal - The kernel asserts this signal when the processing is done.
  - Bit 2: idle signal - The kernel asserts this signal when it is not processing any data.
- One or more AXI Stream interfaces for streaming data between kernels.

Packaging RTL Kernel

There are three steps to packaging an RTL block as an RTL kernel for SDAccel™ applications:

1. Package the RTL block as Vivado® IP.
2. Create a kernel description XML file.
3. Package the RTL kernel into a Xilinx object file.

Package RTL Block as Vivado IP

The RTL block must be packaged as a Vivado IP using Vivado 2015.1_sda version. See *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118) for details on IP packaging in Vivado.

Create Kernel Description XML File

A kernel description XML file needs to be manually created for the RTL IP to be used as an RTL kernel in SDAccel environment. The following is an example of the kernel XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="0">
<kernel name="input_stage" language="ip"
vlnv="xilinx.com:hls:input_stage:1.0" attributes=""
preferredWorkGroupSizeMultiple="0" workGroupSize="1">
<ports>
<port name="M_AXI_GMEM" mode="master" range="0x3FFFFFFF" dataWidth="32"
portType="addressable" base="0x0"/>
<port name="S_AXI_CONTROL" mode="slave" range="0x1000" dataWidth="32"
portType="addressable" base="0x0"/>
<port name="AXIS_P0" mode="write_only" dataWidth="32" portType="stream"/>
</ports>
<args>
<arg name="input" addressQualifier="1" id="0" port="M_AXI_GMEM"
size="0x4" offset="0x10" hostOffset="0x0" hostSize="0x4" type="int*" />
<arg name="__xcl_gv_p0" addressQualifier="4" id="" port="AXIS_P0"
size="0x4" offset="0x18" hostOffset="0x0" hostSize="0x4" type=""
memSize="0x800"/>
</args>
</kernel>
<pipe name="xcl_pipe_p0" width="0x4" depth="0x200" linkage="internal"/>
<connection srcInst="input_stage" srcPort="p0" dstInst="xcl_pipe_p0"
dstPort="S_AXIS"/>
</root>
```

The following table describes the format of the kernel XML in detail:

Table 11–1: **Kernel XML Format**

Tag	Attribute	Description
<root>	versionMajor	Set to 1 for the current release of SDAccel
	versionMinor	Set to 0 for the current release of SDAccel
<kernel>	name	Kernel name
	language	Always set it to "ip" for RTL kernels

	vlnv	Must match the vendor, library, name, and version attributes in the component.xml of an IP. For example, If component.xml has the following tags:
		`<spirit:vendor>xilinx.com</spirit:vendor>`
		`<spirit:library>hls</spirit:library>`
		`<spirit:name>test_sincos</spirit:name>`
		`<spirit:version>1.0</spirit:version>`
		the vlnv attribute in kernel XML will need to be set to:
		`xilinx.com:hls:test_sincos:1.0`

	attributes	Reserved. Set it to empty string.
	prefreredWorkGroupSizeMultiple	Reserved. Set it to 0.
	workGroupSize	Reserved. Set it to 1.

Tag	Attribute	Description
<port>	name	Port name. At least an AXI MM master port and an AXI MM slave port are required. AXI stream port can be optionally specified to stream data between kernels.
	mode	• For AXI MM master port, set it to "master". • For AXI MM slave port, set it to "slave". • For AXI Stream master port, set it to "write_only". • For AXI Stream slave port, set it "read_only".
	range	The range of the address space for the port.
	dataWidth	The width of the data that goes through the port, default is 32 bits.
	portType	Indicate whether or not the port is addressable or streaming. • For AXI MM master and slave ports, set it to "addressable". • For AXI Stream ports, set it to "stream".
	base	For AXI MM master and slave ports, set to "0x0". This tag is not applicable to AXI Stream ports.
<arg>	name	Kernel argument name.
	addressQualifier	Valid values: 1: global memory 2: local memory 3: constant memory 4: pipe
	id	Only applicable for AXI MM master and slave ports. The ID needs to be sequential. It is used to determine the order of kernel arguments. Not applicable for AXI Stream ports.
	port	Indicates the port that the arg is connected to.
	size	Size of the argument. The default is 4 bytes.
	offset	Indicates the register memory address.
	type	The C data type for the argument. E.g. int*, float*
	hostOffset	Reserved. Set to 0x0.
	hostSize	Reserved. Set to 0x4
	memSize	Not applicable to AXI MM master and slave ports. For AXI Stream ports, memSize sets the depth of the FIFO created for the AXI stream ports.
The following tags specify additional information for AXI Stream ports. They are not applicable to AXI MM master or slave ports.		

Tag	Attribute	Description
<pipe>	For each pipe in the compute unit, the compiler inserts a FIFO for buffering the data. The pipe tag describes configuration of the FIFO.	
	name	This specifies the name for the FIFO inserted for the AXI Stream port. This name must be unique among all pipes used in the same compute unit.
	width	This specifies the width of FIFO in bytes. For example, 0x4 for 32-bit FIFO.
	depth	This specifies the depth of the FIFO in number of words.
	linkage	Always set to "internal".
<connection>	The connection tag describes the actual connection in hardware either from the kernel to the FIFO inserted for the PIPE or from the FIFO to the kernel.	
	srcInst	Specifies the source instance of the connection.
	srcPort	Specifies the port on the source instance for the connection.
	dstInst	Specifies the destination instance of the connection.
	dstPort	Specifies the port on the destination instance of the connection.

Package RTL Kernel into Xilinx Object File

The final step is to package the RTL IP and the kernel XML together into a Xilinx object file (`.xo`) so it can be used by the SDAccel compiler. The following is the command example in Vivado 2015.1_sda. The final RTL kernel is in the `test.xo` file.

```
package_xo -xo_path test.xo -kernel_name test_sincos -kernel_xml
kernel.xml -ip_directory ./ip/
```

Using RTL Kernel

RTL kernels can be created in an SDAccel application by using the commands below. Note the type for `create_kernel` command is `ip` and the file name for kernel source is the `.xo` file generated from the Package RTL Kernel step. RTL kernels are not supported in CPU emulation flow. They are supported in hardware emulation flow and build system flow.

```
create_kernel test_sincos -type ip add_files -kernel
[get_kernels test_sincos] "test.xo"
```

Availability

To learn more about the SDAccel development environment, visit www.xilinx.com/products/design-tools/software-zone/sdaccel.html where you will find video tutorials, documentation, and links to the SDAccel development environment-qualified Alliance members. To access the capabilities of the SDAccel development environment, please contact your local Xilinx sales representative.

Command Reference

SDAccel™ is a command line based environment for the development of OpenCL™ applications targeted at Xilinx® FPGA devices. This appendix provides a reference for every command in SDAccel.

Solution Creation Commands

create_solution

The SDAccel™ environment is a solution-based tool. All applications compiled by the SDAccel environment must be part of a solution. The `create_solution` command defines a solution within the SDAccel development environment.

```
create_solution -name <solution name> -dir <solution directory> -force
```

Options

`-name` - defines the name of the solution in the SDAccel environment. The solution name can be any name defined by the user. It is used to create the top level working directory where the application will be compiled.

`-dir` - allows the user to specify the directory where the SDAccel solution is to be created. This setting is optional. By default, the SDAccel environment creates the solution in the current directory from where the tool was invoked.

`-force` - tells the SDAccel environment to completely delete and overwrite the solution directory specified by the name option if it already exists.

Examples

Create a new solution name mysolution in the temp directory and delete any previous work for mysolution if it exists.

```
create_solution -name mysolution -dir /temp -force
```

Add Target Device Commands

add_device

The `add_device` command defines the target device for which the application is compiled. The name of the device is determined by the device provider for the SDAccel™ environment.

A list of devices included in the SDAccel environment is available in SDAccel Environment Supported Devices.

```
add_device <device_name>
```

Custom devices can be targeted by setting the device_repo_paths property on the current solution: `set_property device_repo_paths /path/to/custom_dsa [current_solution]`

Examples

```
add_device xilinx:adm-pcie-7v3:1ddr:2.1
```

Host Code Commands

add_files

The `add_files` command adds user code host application files to the current solution in the SDAccel™ environment. The files listed by the `add_files` command are used from their current location in the programmer's environment. Files are not copied nor stored in the SDAccel solution.

```
add_files <file name>
```

Examples

```
add_files foo.c
```

Host Code Properties

file_type

The `add_files` command handles both header and source files in the same way. For proper compilation of the user application, the SDAccel™ environment needs to know which files to treat as header files during the code compilation phase. The `file_type` property allows the user to specify which files to treat as header files. The `get_files` command that is attached to this property defines which file in the solution the `file_type` property is used with.

```
set_property file_type "c header files" [get_files <file name>]
```

Examples

```
set_property file_type "c header files" [get_files "foo.h"]
```

host_cflags

The `host_cflags` property defines any additional compiler flags that must be set during the compilation of the host code. These flags are passed as is by the SDAccel™ development environment to the code compiler. The code compiler used by the SDAccel environment to process the host code depends on the design phase of the application and the host processor type in the target device.

```
set_property -name host_cflags -value "<compiler flags>" -objects [current_solution]
```

Examples

```
set_property -name host_cflags -value "-g -D XILINX" -objects [current_solution]
```

Kernel Definition Commands

create_kernel

The `create_kernel` command defines a kernel within the context of the current solution. The name of the kernel must match the name of the kernel function in the source code whose operations will be compiled into a custom compute unit for execution on the FPGA fabric.

```
create_kernel <kernel name> -type <kernel source code type>
```

Options

`-type` - defines the language in which the kernel has been captured. The valid types in the SDAccel™ development environment are `clc` for kernels expressed in OpenCL™ C kernel language and `c` for kernels expressed in C/C++ that can be consumed by the Vivado® HLS tool.

Examples

```
create_kernel test -type clc
```

Kernel Code Commands

add_files -kernel

The `add_files` command with the kernel option adds user kernel code files to the current solution in the SDAccel™ development environment. Because the SDAccel environment supports more than one kernel per solution, the `add_files` command also needs to specify the kernel in the solution that the source files correspond to. The correspondence between source files and kernels in the solution is achieved by the `get_kernels` command.

```
add_files -kernel [get_kernels <kernel name>] <kernel source file name>
```

Examples

```
add_files -kernel [get_kernels test]test.cl
```

Kernel Properties

kernel_flags

The `kernel_flags` property defines any additional compiler flags that must be set during the compilation of kernel code. Kernel flags are set on a per kernel basis and each kernel can be compiled with different flags. The pairing between compiler flags and the kernel object in the SDAccel™ solution is through the use of the `get_kernels` command.

```
set_property -name kernel_flags -value  "<compiler
flags>" -objects [get_kernels <kernel name>]
```

Examples

```
set_property kernel_flags "-D" [get_kernels test]
```

max_memory_ports

The `max_memory_ports` property is used for memory architecture optimization. This property informs the SDAccel™ development environment to create one physical memory port per global memory buffer defined in the kernel code. This property applies to all compute units generated for a given kernel.

```
set_property max_memory_ports true [get_kernels <kernel name>]
```

Examples

```
set_property max_memory_ports true [get_kernels test]
```

memory_data_port_width

The `memory_data_port_width` property is used for memory architecture optimization. This property informs the SDAccel™ development environment to override the fundamental bitwidth of a buffer data type and implement an interface of size port width. The supported interface sizes in the SDAccel environment for memory interfaces are 32, 64, 128, 256, and 512 bits. The setting of this property applies to all compute units generated for a given kernel.

```
set_property memory_data_port_width <port width> [get_kernels <kernel name>]
```

Examples

```
set_property memory_data_port_width 512 [get_kernels test]
```

Binary Container Definition Commands

create_opencl_binary

The SDAccel™ environment creates precompiled binaries that contain the custom compute units, which will be executed concurrently on the FPGA fabric. A binary is compiled to run on a specific device of the device. The name of the binary can be anything the user wants.

```
create_opencl_binary -device <device name> <binary name>
```

Options

`-device` - defines which device in the target device will be used to execute the OpenCL™ binary container.

Examples

```
create_opencl_binary -device "xilinx:adm-pcie-7v3:1ddr:2.01" test_binary
```

Binary Container Properties

region

The `region` property is an optional property that helps further define where on a device the current binary container is executed. All devices supported by the SDAccel™ development environment have at least one region. By default, the SDAccel environment chooses the first computation region available for the device selected during the definition of the binary container.

```
set_property region <region name> [get_opencl_binary <binary name>]
```

Examples

```
set_property region "OCL_REGION_0" [get_opencl_binary test_binary]
```

Compute Unit Defnition Commands

create_compute_unit

The `create_compute_unit` command creates an instance of the custom logic required to execute the operations of a given kernel. Compute units are created within the context of a binary container. A binary container can contain multiple compute units for the same kernel.

```
create_compute_unit -opencl_binary [get_opencl_binary <binary name>]
                    -kernel [get_kernels <kernel name>]
                    -name <compute unit instance name>
```

Options

`-opencl_binary` - defines which binary container the current compute unit is to be created in.

`-kernel` - defines the kernel in the solution from which the compute unit is created.

`-name` - is an optional command to distinguish between compute units in the system. If a kernel only has one compute unit, the name tag is optional. If a kernel has multiple compute units in the system, each compute unit must have a unique name. The name of the compute unit is used internally by the Xilinx® OpenCL™ runtime library to schedule the execution of kernel operations. This name is not exposed to the application programmer in any way and is not referred to in the application host code.

Examples

```
create_compute_unit -opencl_binary [get_opencl_binary test_binary]
                    -kernel [get_kernels test]
                    -name myinstance
```

Emulation Commands

compile_emulation

The `compile_emulation` command compiles the current solution for execution in the emulation environment provided by the SDAccel™ development environment. There are two emulation modes supported in the SDAccel environment. These are CPU and hardware. The CPU emulation is the default emulation environment and allows the programmer a fast turnaround for compile, debug, and run iterations while the application functionality is being developed. The hardware emulation flow executes the custom logic generated for each compute unit inside of a hardware simulator to provide the application programmer with assurance that the compiler built the correct logic before testing it on the FPGA.

```
compile_emulation -flow {cpu | hardware}
```

Options

`-flow` - determines if the application is to be compiled for CPU or hardware emulation. CPU emulation is the default. Therefore this flag is only required when selecting compilation for the hardware emulation flow.

Examples

- CPU Emulation

  ```
  compile_emulation
  ```

- Hardware Emulation

  ```
  compile_emulation -flow hardware
  ```

run_emulation

The `run_emulation` command executes the compiled application in either the CPU or hardware emulation environments. The arguments flag provides a mechanism to provide command line arguments to the host program executable. The `run_emulation` command must be preceded by a `compile_emulation` command.

```
run_emulation -flow {cpu | hardware} -args <command line arguments>
```

Options

`-flow` - allows the user to determine if the application will be executed in either the CPU or the hardware emulation environments. The default is to execute all applications using CPU based emulation. Therefore, this option is only needed when specifying that an application must be executed in the hardware emulation environment.

`-args` - enables the passing of command line arguments to the host program executable. Any option that would be entered at the command line with the host program executable can be used here.

Examples

- CPU Emulation

  ```
  run_emulation
  ```

- Hardware Emulation

  ```
  run_emulation -flow hardware
  ```

Reporting Commands

report_estimate

The `report_estimate` command generates a compute unit level estimate on the application performance of the FPGA resources required to implement the functionality. It is a good practice to run this command and analyze its results before leaving the emulation part of the design cycle and moving to the FPGA system compilation step.

```
report_estimate
```

Examples

```
report_estimate
```

System Compilation Commands

build_system

The `build_system` command compiles each binary container in the current solution into programming binaries for the FPGA. This is the step responsible for generating custom logic and allocating FPGA resources to specific compute units. It is typical for this step to take several hours. This is the longest running step in the SDAccel™ flow. Therefore, a best practice recommendation is for the application programmer to run the emulation steps and analyze the output of the `report_estimate` command for correctness before proceeding to this step.

```
build_system
```

Examples

```
build_system
```

package_system

The `build_system` command creates the binaries for the host code and the compute units to be executed on FPGA based accelerator cards. The `package_system` command completes the binary creation process by placing all generated binaries in a central location for the application programmer to use for deployment. All binaries suitable for execution on the FPGA accelerator card are placed in this location:

```
<solution name>/pkg
```

```
package_system
```

Examples

```
package_system
```

System Execution Commands

run_system

The `run_system` command allows the user to execute the application on the FPGA based accelerator card from within the SDAccel™ environment. This is identical to taking the compiled binaries in the `pkg` directory and executing them in a shell outside of the SDAccel environment.

```
run_system -args <command_line_arguments>
```

Examples

```
run_system -args "test.xclbin"
```

SDAccel Environment Supported Devices

SDAccel™ solutions are compiled against a target device. A device is the combination of board and infrastructure components on which the kernels of an application is executed. Applications compiled against a target device can only be executed on the board associated with the target device. Devices can be provided by SDAccel ecosystem partners, FPGA design teams, and Xilinx. Contact your Xilinx representative for more information about adding third-party devices in the SDAccel environment. The following table shows the devices included in the SDAccel environment:

Device	**Board**	**FPGA Device**	**Board Vendor**
xilinx:adm-pcie-7v3:1ddr:2.1	ADM-PCIE-7V3	Virtex®-7 690T	Alpha Data
xilinx:adm-pcie-ku3:1ddr:2.1	ADM-PCIE-KU3	Kintex® UltraScale™ KU060 with 1 DDR bank	Alpha Data
xilinx:adm-pcie-ku3:2ddr:2.1	ADM-PCIE-KU3	Kintex UltraScale KU060 with 2 DDR banks	Alpha Data

IMPORTANT: *XCL_PLATFORM environment variable is deprecated. Please do not use it for new designs.*

Appendix C

OpenCL Built-In Functions Support in the SDAccel Environment

The OpenCL™ C programming language provides a rich set of built-in functions for scalar and vector operations. Many of these functions are similar to the function names provided in common C libraries but they support scalar and vector argument types. The SDAccel™ development environment is OpenCL 1.0 embedded profile compliant. The following tables show descriptions of built-in functions in OpenCL 1.0 embedded profile and their support status in the SDAccel environment.

Work-Item Functions

Function	Description	Supported
get_global_size	Number of global work items	Yes
get_global_id	Global work item ID value	Yes
get_local_size	Number of local work items	Yes
get_local_id	Local work item ID	Yes
get_num_groups	Number of work groups	Yes
get_group_id	Work group ID	Yes
get_work_dim	Number of dimensions in use	Yes

Math Functions

Function	Description	Supported
acos	Arc Cosine function	Yes
acosh	Inverse Hyperbolic Cosine function	Yes
acospi	acox(x)/PI	Yes
asin	Arc Cosine function	Yes
asinh	Inverse Hyperbolic Cosine function	Yes
asinpi	Computes acos (x) / pi	Yes
atan	Arc Tangent function	Yes
atan2(y, x)	Arc Tangent of y / x	Yes
atanh	Hyperbolic Arc Tangent function	Yes

Function	Description	Supported
atanpi	Computes atan (x) / pi	Yes
atan2pi	Computes atan2 (y, x) / pi	Yes
cbrt	Compute cube-root	Yes
ceil	Round to integral value using the round to +ve infinity rounding mode.	Yes
copysign(x, y)	Returns x with its sign changed to match the sign of y.	Yes
cos	Cosine function	Yes
cosh	Hyperbolic Cosine function	Yes
cospi	Computes cos (x * pi)	Yes
erf	The error function encountered in integrating the normal distribution	Yes
erfc	Complementary Error function	Yes
exp	base- e exponential of x	Yes
exp2	Exponential base 2 function	Yes
exp10	Exponential base 10 function	Yes
expm1	exp(x) - 1.0	Yes
fabs	Absolute value of a floating-point number	Yes
fdim(x, y)	x - y if x > y, +0 if x is less than or equal to y.	Yes
fma(a, b, c)	Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.	Yes
fmax(x, y) fmax(x, float y)	Returns y if x is less than y, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN	Yes
fmin(x, y) fmin(x, float y)	Returns y if y less than x, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.	Yes
fmod	Modulus. Returns x - y * trunc (x/y)	Yes
fract	Returns fmin(x - floor(x), 0x1.fffffep-1f). floor(x) is returned in iptr	Yes

Function	Description	Supported
frexp	Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2exp.	Yes
hypot	Computes the value of the square root of x2 + y2 without undue overflow or underflow.	Yes
ilogb	Returns the exponent as an integer value.	Yes
ldexp	Multiply x by 2 to the power n.	Yes
lgamma	Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.	Yes
lgamma_r	Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.	Yes
log	Computes natural logarithm.	Yes
log2	Computes a base 2 logarithm	Yes
log10	Computes a base 10 logarithm	Yes
log1p	loge(1.0+x)	Yes
logb	Computes the exponent of x, which is the integral part of logr |x|.	Yes
mad	Approximates a * b + c. Whether or how the product of a * b is rounded and how supernormal or subnormal intermediate products are handled is not defined. mad is intended to be used where speed is preferred over accuracy30	Yes
modf	Decompose a floating-point number. The modf function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by iptr.	Yes
nan	Returns a quiet NaN. The nancode may be placed in the significand of the resulting NaN.	Yes
nextafter	Next representable floating-point value following x in the direction of y	Yes

Function	Description	Supported
pow	Computes x to the power of y	Yes
pown	Computes x to the power of y, where y is an integer.	Yes
powr	Computes x to the power of y, where x is greater than or equal to 0.	Yes
remainder	Computes the value r such that r = x - n*y, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x.	Yes
remquo	Floating point remainder and quotient function.	Yes
rint	Round to integral value (using round to nearest even rounding mode) in floating-point format.	Yes
rootn	Compute x to the power 1/y.	Yes
round	Return the integral value nearest to x rounding halfway cases away from zero, regardless of the current rounding direction.	Yes
rsqrt	Inverse Square Root	Yes
sin	Computes the sine	Yes
sincos	Computes sine and cosine of x. The computed sine is the return value and computed cosine is returned in cosval.	Yes
sinh	Computes the hyperbolic sine	Yes
sinpi	Computes sin (pi * x).	Yes
sqrt	Computes square root.	Yes
tanh	Computes the tangent.	Yes
tanpi	Computes tan(pi * x).	Yes
tgamma	Computes the gamma.	Yes
trunc	Round to integral value using the round to zero rounding mode.	Yes
half_cos	Computes cosine. x must be in the range -216… +216. This function is implemented with a minimum of 10-bits of accuracy	Yes
half_divide	Computes x / y. This function is implemented with a minimum of 10-bits of accuracy	Yes
half_exp	Computes the base- e exponential of x. implemented with a minimum of 10-bits of accuracy	Yes

Function	Description	Supported
half_exp2	The base- 2 exponential of x. implemented with a minimum of 10-bits of accuracy	Yes
half_exp10	The base- 10 exponential of x. implemented with a minimum of 10-bits of accuracy	Yes
half_log	Natural logarithm. implemented with a minimum of 10-bits of accuracy	Yes
half_log10	Base 10 logarithm. implemented with a minimum of 10-bits of accuracy	Yes
half_log2	Base 2 logarithm. implemented with a minimum of 10-bits of accuracy	Yes
half_powr	x to the power of y, where x is greater than or equal to 0.	Yes
half_recip	Reciprocal. Implemented with a minimum of 10-bits of accuracy	Yes
half_rsqrt	Inverse Square Root. Implemented with a minimum of 10-bits of accuracy	Yes
half_sin	Computes sine. x must be in the range -2^16... +2^16. implemented with a minimum of 10-bits of accuracy	Yes
half_sqrt	Inverse Square Root. Implemented with a minimum of 10-bits of accuracy	Yes
half_tan	The Tangent. Implemented with a minimum of 10-bits of accuracy	Yes
native_ cos	Computes cosine over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ divide	Computes x / y over an implementation-defined range. The maximum error is implementation-defined	Yes
native_ exp	Computes the base- e exponential of x over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ exp2	Computes the base- 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.	No

Function	Description	Supported
native_exp10	Computes the base- 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.	No
native_ log	Computes natural logarithm over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ log10	Computes a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.	No
native_ log2	Computes a base 2 logarithm over an implementation-defined range.	No
native_ powr	Computes x to the power of y, where x is greater than or equal to 0. The range of x and y are implementation-defined. The maximum error is implementation-defined.	No
native_ recip	Computes reciprocal over an implementation-defined range. The maximum error is implementation-defined.	No
native_ rsqrt	Computes inverse square root over an implementation-defined range. The maximum error is implementation-defined.	No
native_ sin	Computes sine over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ sqrt	Computes inverse square root over an implementation-defined range. The maximum error is implementation-defined.	No
native_ tan	Computes tangent over an implementation-defined range. The maximum error is implementation-defined	Yes

Integer Functions

Function	Description	Supported
abs	|x|	Yes
abs-diff	|x-y| without modulo overflow	Yes
add_sat	x+y and saturate result	Yes
hadd	(x+y) >> 1 without modulo overflow	Yes
rhadd	(x+y+1) >> 1. The intermediate sum does not modulo overflow.	Yes
clz	Number of leading 0-bits in x	Yes
mad_hi	mul_hi(a,b)+c	Yes
mad24	(Fast integer function.) Multiply 24-bit integer then add the 32-bit result to 32-bit integer	Yes
mad_sat	a*b+c and saturate the result	Yes
max	The greater of x or y	Yes
min	The lessor of x or y	Yes
mul_hi	High half of the product of x and y	Yes
mul24	(Fast integer function.) Multiply 24-bit integer values a and b	Yes
rotate	result[indx]=v[indx]<<i[indx]	Yes
sub_sat	x - y and saturate the result	Yes
upsample	result[i] = ((gentype)hi[i] << 8|16|32) | lo[i]	Yes

Common Functions

Function	Description	Supported
clamp	Clamp x to range given by min, max	Yes
degrees	radians to degrees	Yes
max	Maximum of x and y	Yes
min	Minimum of x and y	Yes
mix	Linear blend of x and y	Yes
radians	degrees to radians	Yes
sign	Sign of x	Yes
smoothstep	Step and interpolate	Yes
step	0.0 if x < edge, else 1.0	Yes

Geometric Functions

Function	Description	Supported
clamp	Clamp x to range given by min, max	Yes
degrees	radians to degrees	Yes
cross	Cross product	Yes
dot	Dot product only float, double, half data types	Yes
dstance	Vector distance	Yes
length	Vector length	Yes
normalize	Normal vector length 1	Yes
fast_distance	Vector distance	Yes
fast_length	Vector length	Yes
fast_normalize	Normal vector length 1	Yes

Relational Functions

Function	Description	Supported
isequal	Compare of x == y.	Yes
isnotequal	Compare of x != y.	Yes
isgreater	Compare of x > y.	Yes
isgreaterequal	Compare of x >= y.	Yes
isless	Compare of x < y.	Yes
islessequal	Compare of x <= y.	Yes
islessgreater	Compare of (x < y) || (x > y).	Yes
isfinite	Test for finite value.	Yes
isinf	Test for +ve or -ve infinity.	Yes
isnan	Test for a NaN.	Yes
isnormal	Test for a normal value.	Yes
isordered	Test if arguments are ordered.	Yes
isunordered	Test if arguments are unordered.	Yes
signbit	Test for sign bit.	Yes
any	1 if MSB in any component of x is set; else 0.	Yes
all	1 if MSB in all components of x is set; else 0.	Yes

Function	Description	Supported
bitselect	Each bit of result is corresponding bit of a if corresponding bit of c is 0.	Yes
select	For each component of a vector type, result[i] = if MSB of c[i] is set ? b[i] : a[i] For scalar type, result = c ? b : a.	Yes

Vector Data Load and Store Functions

Function	Description	Supported
vloadn	Read vectors from a pointer to memory.	Yes
vstoren	Write a vector to a pointer to memory.	Yes
vload_half	Read a half float from a pointer to memory.	Yes
vload_halfn	Read a half float vector from a pointer to memory.	Yes
vstore_half	Convert float to half and write to a pointer to memory.	Yes
vstore_halfn	Convert float vector to half vector and write to a pointer to memory.	Yes
vloada_halfn	Read half float vector from a pointer to memory.	Yes
vstorea_halfn	Convert float vector to half vector and write to a pointer to memory.	Yes

Synchronization Functions

Function	Description	Supported
barrier	All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier.	Yes

Explicit Memory Fence Functions

Function	Description	Supported
mem_fence	Orders loads and stores of a work-item executing a kernel	Yes
read_mem_fence	Read memory barrier that orders only loads	Yes
write_mem_fence	Write memory barrier that orders only stores	Yes

Async Copies from Global to Local Memory, Local to Global Memory Functions

Function	Description	Supported
async_work_group_copy	Must be encountered by all work-items in a workgroup executing the kernel with the same argument values; otherwise the results are undefined.	Yes
wait_group_events	Wait for events that identify the async_work_group_copy operations to complete.	Yes
prefetch	Prefetch bytes into the global cache.	No

PIPE Functions

Function	Description	Supported
read_pipe	Read packet from pipe	Yes
write_pipe	Write packet to pipe	Yes
reserve_read_pipe	Reserve entries for reading from pipe	No
reserve_write_pipe	Reserve entries for writing to pipe	No
commit_read_pipe	Indicates that all reads associated with a reservation are completed	No
commit_write_pipe	Indicates that all writes associated with a reservation are completed	No
is_valid_reserve_id	Test for a valid reservation ID	No
work_group_reserve_read_pipe	Reserve entries for reading from pipe	No

Function	Description	Supported
work_group_reserve_write_pipe	Reserve entries for writing to pipe	No
work_group_commit_read_pipe	Indicates that all reads associated with a reservation are completed	No
work_group_commit_write_pipe	Indicates that all writes associated with a reservation are completed	No
get_pipe_num_packets	Returns the number of available entries in the pipe	Yes
get_pipe_max_packets	Returns the maximum number of packets specified when pipe was created	Yes

Pipe Functions enabled by the cl_khr_subgroups extension

Function	Description	Supported
sub_group_reserve_read_pipe	Reserve entries for reading from a pipe	No
sub_group_reserve_write_pipe	Reserve entries for writing to a pipe	No
sub_group_commit_read_pipe	Indicates that all reads associated with a reservation are completed	No
sub_group_commit_write_pipe	Indicates that all writes associated with a reservation are completed	No

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

References

1. *SDAccel Development Environment Tutorial: Getting Started* (UG1021)
2. SDAccel Development Environment web page
3. Vivado® Design Suite Documentation
4. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
5. *Vivado Design Suite User Guide: High level Synthesis* (UG902)
6. *SDAccel Development Environment Installation and Licensing Guide* (UG1020)
7. Khronos Group web page: Documentation for the OpenCL standard
8. Alpha Data web page: Documentation for the ADM-PCIE-7V3 Card
9. Pico Computing web page: Documentation for the M-505-K325T card and the EX400 Card

Please Read: Important Legal Notices