acceleware
PROCESSING SUPERPOWER

OpenCL on FPGAs for GPU Programmers

Introduction

The aim of this whitepaper is to introduce developers who have previous experience with general-purpose computing on **graphics processing units** (GPUs) to parallel programming targeting Altera® **field-programmable gate arrays** (FPGAs) via the Open Computing Language (OpenCL™) framework.

This paper provides a brief overview of OpenCL, highlights some of the underlying technology and benefits behind Altera FPGAs, then focuses on how OpenCL kernels are executed on Altera FPGAs compared to on GPUs. This paper also presents the key differences in optimization techniques for targeting FPGAs.

Overview of OpenCL

OpenCL is an open standard framework for parallel programming that executes across heterogeneous processors, such as GPUs and FPGAs, as well as **central processing units** (CPUs) and **digital signal processors** (DSPs). The standard uses a subset of ISO C99 with added extensions for supporting parallelism, and supports both data and task-based parallel programming models.

Heterogeneous Programming

The basis of the OpenCL platform model is a **host** connected to one or more **devices**, such as FPGAs and GPUs, possibly from multiple device vendors. The host is responsible for device memory management, transferring data to devices, queuing work for devices, and error-management.

Data Parallelism and Kernels

Data parallelism is a form of parallelism across multiple processors that is achieved when each processor performs identical tasks on different pieces of distributed data. Data-parallel portions of an algorithm are executed on devices as **kernels**, which are C functions with some restrictions and a few language extensions. The host launches kernels across a 1D, 2D, or 3D grid of **work-items** to be processed by the devices. Conceptually, work-items can be thought of as individual processing threads, that each execute the same kernel function. Work-items have a unique index within the grid, and typically compute different portions of the result. Work-items are grouped together into **work-groups**, which are expected to execute independently from one another.

Memory Hierarchy – Global, Local, Constant, and Private

The OpenCL kernels have access to four distinct memory regions distinguished by access type and scope. **Global** memory allows read-and-write access from all work-items across all work-groups. **Local** memory also provides read-and-write access to work-items, however it is only visible to other work-items within the same work-group. **Constant** memory is a region of read-only memory accessible to all work-items, thus immutable during kernel execution. Lastly, **private** memory provides read-and-write access visible to only individual work-items.

acceleware
PROCESSING SUPERPOWER

CUDA vs. OpenCL Terminology

Developers with experience in NVIDIA's CUDA parallel computing architecture for GPUs may notice similarities between CUDA concepts and OpenCL, albeit with a different naming convention. Table 1 presents some equivalent CUDA terms:

Table 1: OpenCL vs CUDA Terminology

OpenCL	CUDA
Work-Item	Thread
Work-Group	Thread Block
Multi-dimension Range (NDRange)	Grid
Global / Constant Memory	Global / Constant Memory
Local Memory	Shared Memory
Private Memory	Local Memory

Altera FPGA Architectures

FPGAs are highly configurable integrated circuits that can be customized and tailored for specific processing needs. An FPGA is composed of large numbers of small building blocks: **adaptive logic modules** (ALMs), **digital signal processing** (DSP) blocks, and **memory blocks**, woven together amongst programmable routing switches. In addition to these FPGA fabric building blocks, additional features may be incorporated, such as high-speed transceivers, PCI Express® (PCIe®) interfaces, multiple memory controller interfaces, or even ARM® Cortex®-A9 processors, as part of a **system on a chip** (SoC) solution. For these SoC solutions, the ARM processor can serve as the OpenCL host, driving the FPGA in a self-contained manner.

For GPUs, kernels are compiled to a sequence of instructions that execute on fixed hardware processors. Typically, these hardware processors are composed of cores that are specialized for different functions, therefore some may be unused based upon kernel instruction requirements. On the other hand, kernels on FPGAs are compiled to custom processing pipelines built up from the programmable resources on the FPGA (for example ALMS, DSP, and memory blocks). By focusing hardware resources only on the algorithm to be executed, FPGAs can provide better performance per watt than GPUs for certain applications.

ALMs can be configured to implement desired logic, arithmetic, and register functions. Variable-precision DSP blocks with hardened single-precision floating-point capabilities implement commonly used logic and arithmetic functions in a power or performance optimized manner. Half or double precision is also possible, with the resources in the FPGA being used to accomplish the additional resolution. The total amount of memory blocks embedded within Altera FPGAs range from a few hundred kilobytes to tens of MBs, offering even higher bandwidth and lower latencies. Memory controllers allow access to larger amounts of external memory, and currently support a variety of memory standards, such as quad-data-rate (QDR) or double-data-rate (DDR) SDRAM.

Executing OpenCL Programs on Altera FPGAs

The key difference between kernel execution on GPUs versus FPGAs is how parallelism is handled. GPUs are "single-instruction, multiple-data" (SIMD) devices – groups of processing elements perform the same operation on their own individual work-items. On the other hand, FPGAs exploit pipeline parallelism – different stages of the instructions are applied to different work-items concurrently.

SIMD Parallelism (GPU)	**1** A	**1** B	**1** C	**1** D	**1** E	**4** A	**4** B	**4** C	**4** D	**4** E
	2 A	**2** B	**2** C	**2** D	**2** E	**5** A	**5** B	**5** C	**5** D	**5** E
	3 A	**3** B	**3** C	**3** D	**3** E	**6** A	**6** B	**6** C	**6** D	**6** E
Clock Cycle	1	2	3	4	5	6	7	8	9	10
Pipeline Parallelism (FPGA)	1 **A**	2 **A**	3 **A**	4 **A**	5 **A**	6 **A**				
		1 **B**	2 **B**	3 **B**	4 **B**	5 **B**	6 **B**			
			1 **C**	2 **C**	3 **C**	4 **C**	5 **C**	6 **C**		
				1 **D**	2 **D**	3 **D**	4 **D**	5 **D**	6 **D**	
					1 **E**	2 **E**	3 **E**	4 **E**	5 **E**	6 **E**

Figure 1: Comparison of SIMD Parallelism Versus Pipeline Parallelism

Figure 1 above illustrates the difference between these two methods; using a simplified example involving six work-items (1-6) executing a kernel with five stages (A-E), with the SIMD approach handling three work items at a time. In this example, both parallelization methods finish in the same amount of total time, however the throughputs are slightly different; if additional work-items are to be processed, the SIMD parallelism approach would continue to complete three work items every 5 cycles, whereas the pipeline parallelism approach would average competition of one work item each cycle.

The difference in parallelization strategies employed by GPUs and FPGAs can be mapped directly back to the underlying hardware. GPUs consist of hundreds of simple processing "cores", which can each handle their own work-items. A number of GPU cores execute the same instruction in lock-step with one another as a SIMD unit of fixed size (sometimes called a **wavefront** or **warp**). On an FPGA, each OpenCL kernel is compiled to a custom circuit. The operations within the kernel are implemented using the ALMs and DSPs, forming sub-circuits that are wired together according to the data flow of the algorithm. Load/store units allow access to global, local, and constant memory. Figure 2 illustrates circuit synthesis for a simple OpenCL kernel. As a result, all algorithmic operations of the kernel are allocated dedicated hardware resources on the FPGA, prior to kernel execution. During execution, work-items step through each stage of the kernel one at a time, however due to the fact that each stage has dedicated hardware, multiple work-items may be passing through the circuit at any given moment, thus yielding pipeline parallelism.

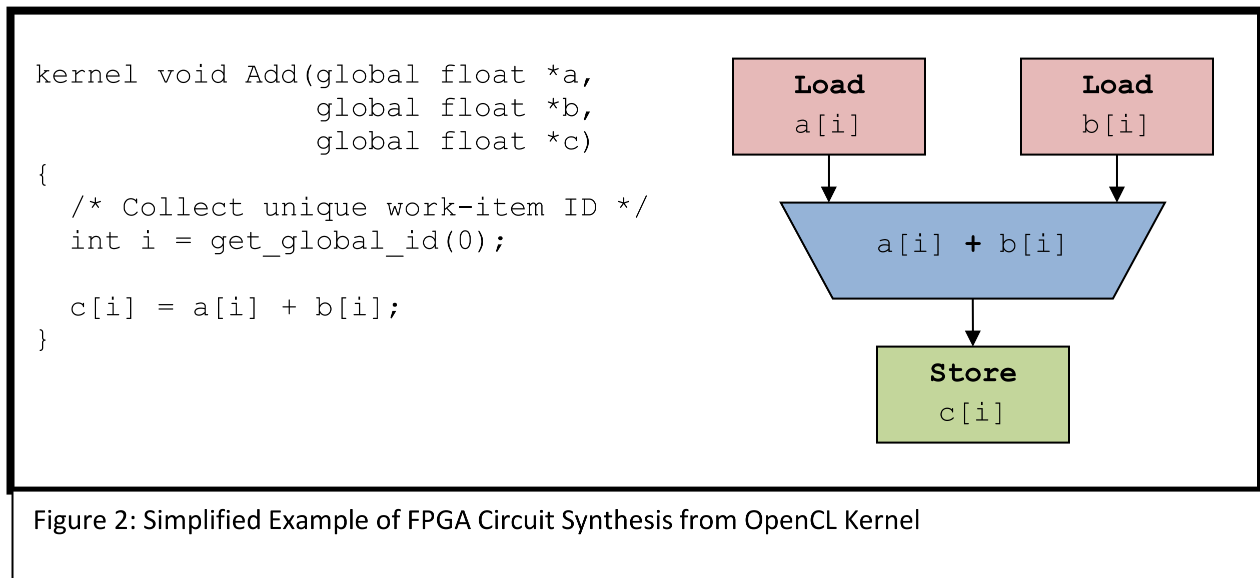

Figure 2: Simplified Example of FPGA Circuit Synthesis from OpenCL Kernel

One consequence of the different parallelization methods is how branching is handled. When branching occurs on a GPU, it is still necessary for all work-items within the same SIMD unit to correctly execute the various branches. However, because the SIMD unit as a whole operates on a single instruction at a time, all code-paths taken by the individual work-items must be executed one after another, with individual work-items disabled or enabled based upon how they evaluated the branching condition. As a result, encountering a branching condition with N options could potentially result in execution time equal to the sum of execution times for all N options (for N up to the SIMD width). Branching is less of an issue on FPGAs because all code-paths are already established in hardware. All branch options can be executed concurrently or even computed speculatively in some cases to allow overlap with branch condition computation. Figure 3 illustrates a simple example of how branching is handled for the two different parallelization methods. Each of the three work items execute the same first stage (A), but then branch and execute the next three stages conditionally (B,C, and D).

SIMD Parallelism (GPU)	1 A	1 B_1	1 C_1	1 D_1	IDLE					
	2 A	IDLE			2 B_2	2 C_2	2 D_2	IDLE		
	3 A	IDLE						3 B_3	3 C_3	3 D_3
Clock Cycle	1	2	3	4	5	6	7	8	9	10
Pipeline Parallelism (FPGA)	1 A	2 A	3 A							
		1 $B_{1,2,3}$	2 $B_{1,2,3}$	3 $B_{1,2,3}$						
			1 $C_{1,2,3}$	2 $C_{1,2,3}$	3 $C_{1,2,3}$					
				1 $D_{1,2,3}$	2 $D_{1,2,3}$	3 $D_{1,2,3}$				

Figure 3: Compare branching behavior - SIMD vs. pipeline parallelism

Key Differences in Optimizing OpenCL for FPGAs

While optimization on GPUs mainly involves modifying code to effectively utilize the fixed underlying hardware, the configurable nature of FPGAs presents a different set of challenges.

Throughput vs. Resource Usage Tradeoffs

Given that OpenCL kernels are compiled to hardware circuits of fixed size, it may mean that a lot of remaining FPGA resources are not used. One method for improving performance on FPGAs is to create multiple copies of the kernel pipelines. Pipelines can execute independently from one another, and performance can scale linearly with the number of copies. Replication is handled in Altera OpenCL by setting the num_compute_units kernel attribute.

In some instances, it may be better to incur some performance penalties by reducing the resources required for a single pipeline so that additional copies can fit onto the FPGA. One method to reduce resources is to cut back on the number of circuit-expensive operators, such as the cosine function; possibly computing the required values on the host and passing them into the FPGA as parameters to the kernel instead.

Conversely, if additional FPGA resources are still available, performance may be improved by investing additional resources on the pipeline. One example of this would be to unroll for loops. The removal of loop counter or loop testing logic is a benefit shared on both GPUs and FPGAs, however the added benefit on FPGAs is that throughput can be increased. Figure 4 illustrates that each work-item accumulates four values then writes to output. In the loop version, one adder is used and throughput is roughly 0.25 work-items per clock cycle. In the unrolled version, four adders are used, however one work-item can complete per clock cycle. Altera OpenCL for FPGAs facilitates loop unrolling via the #pragma unroll directive, and works for loops of known size as well as loop with data-dependent trip counts.

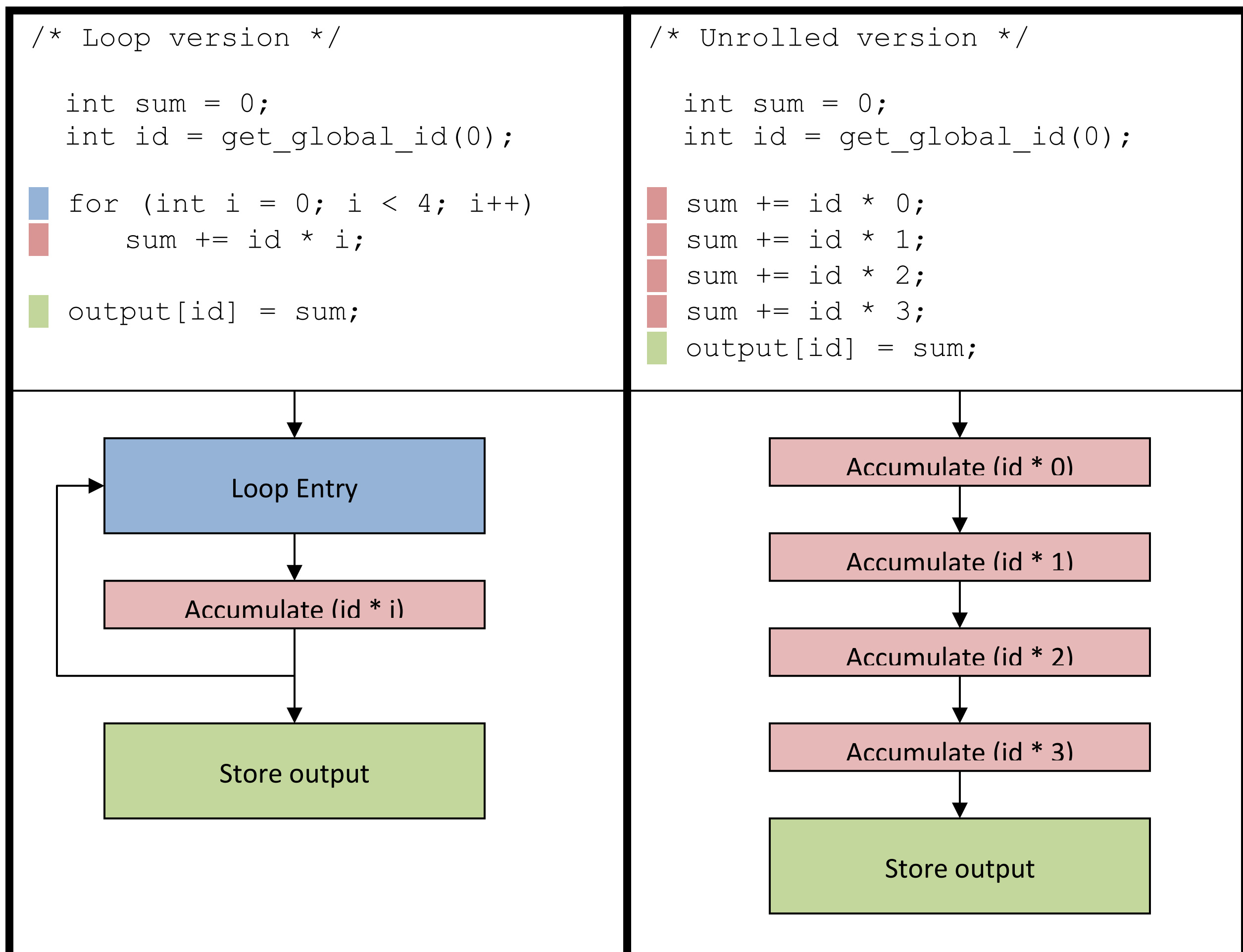

Figure 4: Loop Unrolling Example

A different method of utilizing resources similar to replication is kernel vectorization. Whereas replication makes N exact copies of the kernel pipeline, kernel vectorization maintains a single pipeline where each work-item then does N times as much work. By dealing with larger units of work, kernel vectorization can even reduce the number of loads and stores (each load / store is larger). Kernel vectorization is essentially SIMD parallelization on an FPGA!

Attempting to find ideal settings for replication, vectorization and loop unrolling can be a challenging task, however the Altera OpenCL compiler can optimize these settings via its resource-driven optimizer, which explores multiple design points and chooses the highest performance design.

Memory / Input-Output Subsystems

Unlike GPUs, which provide access to homogenous global (external) memory, Altera FPGAs can support heterogeneous memory interfaces with a variety of external memory types – DDR3 synchronous dynamic random access memory (SDRAM), DDR2 SDRAM, DDR SDRAM, and QDR II static random access memory (SRAM). Multiple memory interfaces may be configured on a single FPGA board, therefore it is important to direct which interface should be used for individual buffers, which can be done via attributes. When dealing with multiple memory banks, data is normally interleaved between banks, however certain applications may see improved performance by manually guiding which bank buffers should reside on.

Another optimization possible on Altera FPGAs not currently present on GPUs is to take advantage of I/O channels and kernel channels (OpenCL 2.0 pipes). Kernel channels allow kernels to transfer data to one another via a first-in-first-out (FIFO) buffer, without the need for host interaction. Traditionally, GPU kernels that want to pass data to one another may do so by issuing reads and writes to global memory combined with some method of synchronization. Performance and power efficiency gains are achieved by the removal of these intermediate reads and writes. Altera FPGAs also extend the idea of kernel channels even further to allow I/O interfaces (I/O pipes), which allow kernels to access directly from a streaming interface without host interactions, known as IO channels. Effectively the host configures the data pipeline and then steps out of the data path. Figure 5 illustrates a kernel being executed on three sets of data coming from an I/O source. Significant time savings are possible because the FPGA communicates directly with the I/O source, and no longer needs the host to serve as a middle-man.

Host	Load GPU$_1$	Launch$_1$		Read GPU$_1$	Load GPU$_2$	Launch$_2$		Read GPU$_2$	Load GPU$_3$	Launch$_3$		Read GPU$_3$
GPU			Kernel$_1$				Kernel$_2$				Kernel$_3$	
I/O	Read$_1$			Write$_1$	Read$_2$			Write$_2$	Read$_3$			Write$_3$
Time	1	2	3	4	5	6	7	8	9	10	11	12
Host	Setup I/O Channels		Launch									
FPGA				Kernel$_1$	Kernel$_2$	Kernel$_3$						
I/O				Read / Write$_1$	Read / Write$_2$	Read / Write$_3$						

Figure 5: Example Highlighting I/O Channel Benefits

Single Work-Item Execution (OpenCL Tasks)

SIMD-based parallel processing is ideal for dealing with loops where there are no dependencies across iterations of the loop - usually parallelization can occur by simply mapping work-items to individual loop iterations. In most real-world applications, data-dependencies are inherent to the structure of the algorithm, and cannot be removed easily. Traditionally, GPU programmers must rely on relatively complicated constructs involving resources shared by work-items in a work group along with synchronization primitives in order to express computations correctly. Alternatively, GPU programmers could choose to have the data-dependent section of work handled by only a single work-item (also called an OpenCL task), however this hampers parallelization and overall performance due to the idleness of other processing cores. Pipeline-parallel devices such as FPGAs have less of an issue dealing with single work-items – single work-items are actually the unit of work in the pipeline anyways! In fact, parallel pipelines can achieve additional performance by pipelining iterations of a loop containing loop carried dependencies – launching the next iteration as soon as loop dependencies have been completed. This scheduling is handled primarily by the compiler; however loop pipelining performance can also be improved by software developer in a number of ways – removing some dependencies, simplifying dependence complexity and relaxing dependence. Removing dependencies, for example by using simple access patterns results in faster launch times for the next iteration. Similar results occur when avoiding expensive operations when computing loop-carried values. Relaxing dependence increases the number of iterations between generation and use of a value, which means that the immediate next iteration can be launched sooner. Setting the kernel attribute “task” informs the Altera OpenCL compiler that the kernel will only run with a single work item. Figure 6 illustrates a simple example of loop-pipelining.

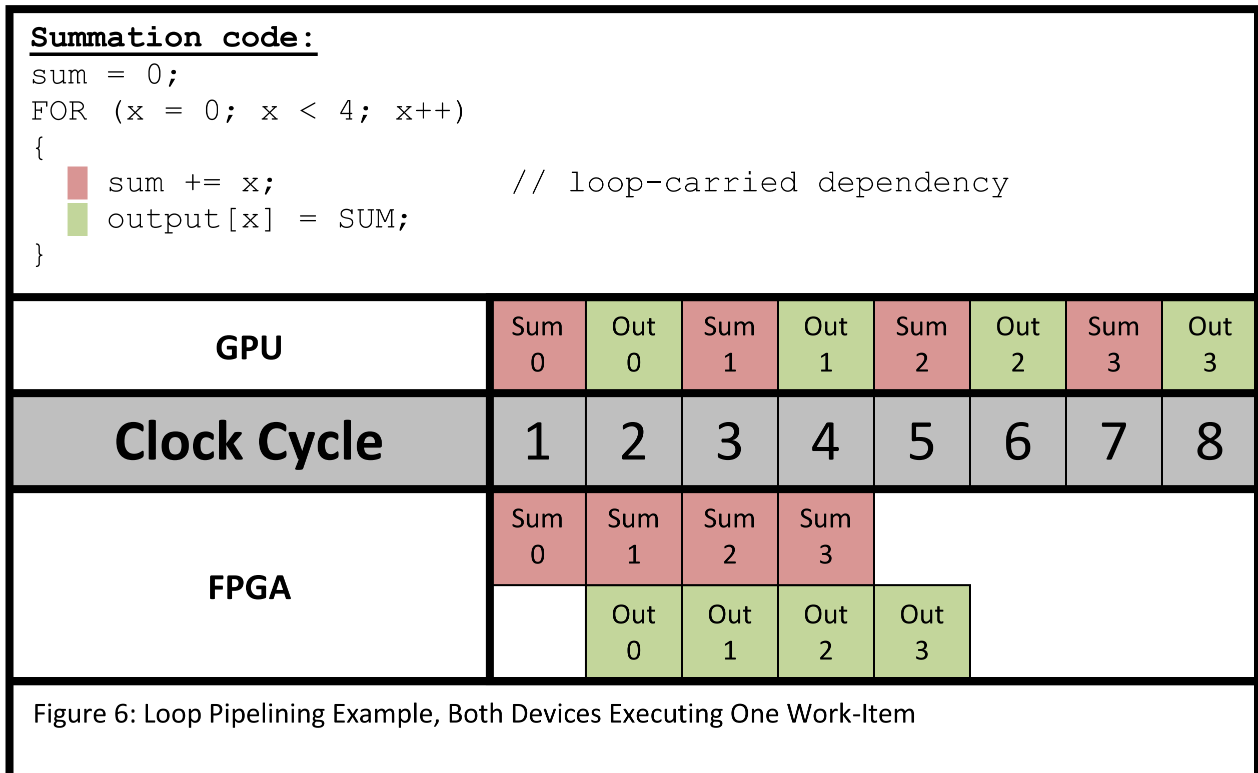

Figure 6: Loop Pipelining Example, Both Devices Executing One Work-Item

Development Process Differences

Kernel development for FPGAs is slightly different than traditional GPU development in that the hardware is created for the specific functions being implemented. FPGA circuit synthesis is a computationally-intensive task, and consumes much more time than compiling kernels into a series of instructions to be executed on a GPU. To reduce development time and more closely resemble the traditional software friendly environment, the recommended approach for FPGA kernel development is to work in stages – syntax debugging, functional debugging, application debugging, and final performance tuning. Each stage is simply a modification of a compiler attribute. During syntax debugging, the Altera compiler simply ensures proper OpenCL code syntax. For functional debugging, the OpenCL kernels are run on an x86 based host to rapidly verify functional correctness of the kernel. During application debugging, an optimization report is produced, providing information about throughput, memory transaction efficiency, as well as potential pipeline stalls. This feedback can be used to modify the kernel accordingly to tune for performance at the functional level, but also on a virtual FPGA fabric that will show how changes in the code will improve performance when the FPGA is actually built. During final performance tuning phase, a fully optimized FPGA circuit is generated and the profiler can be run to get detailed reports on kernel performance.

Table 2: FPGA Development Stages

Stage	Time	Execution Device
Syntax Debugging	Seconds	None
Functional Debugging	Seconds	Emulated kernels on x86 host
Application Debugging	Minutes	Virtual FPGA fabric

Final Performance Tuning	Hours	Optimized circuit on FPGA with profiler

Conclusions

OpenCL is a parallel programming framework that is currently supported by devices, such as GPUs, CPUs, and FPGAs. Kernels are code-portable across different platforms, but not performance-portable, meaning that the same code will execute on various devices, however the underlying parallelism mechanism and hardware features must be taken into consideration when attempting to optimize performance. Of course, since the FPGA creates a custom hardware implementation, code is completely portable between different FPGA families and generations without any modifications or new optimization requirements. Understanding the differences between SIMD parallelism and pipeline parallelism, and taking advantage of FPGA features, such as heterogeneous memory support, channels, and loop pipelining are keys to unlocking high performance-per-watt solutions.