Implementing FPGA Design with the OpenCL Standard

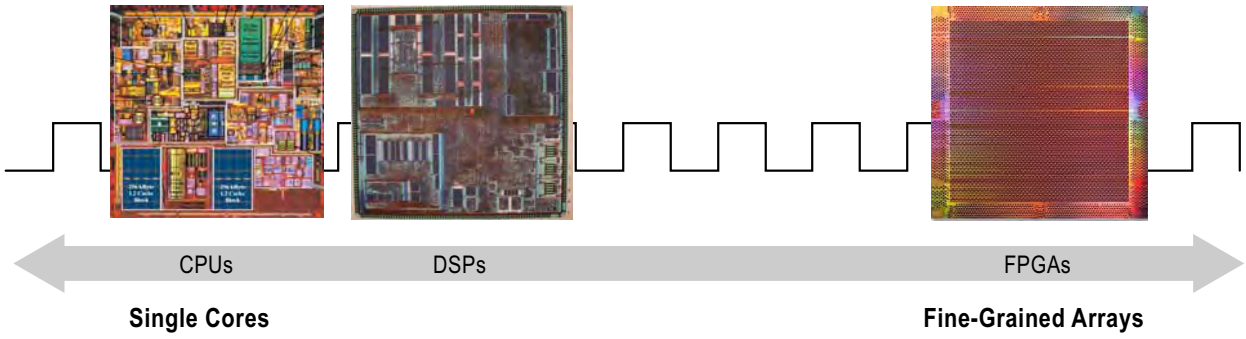
WP-01173-1.0 White Paper

Utilizing the Khronos Group's OpenCL™ standard on an FPGA may offer significantly higher performance and at much lower power than is available today from hardware architectures such as CPUs, graphics processing units (GPUs), and digital signal processing (DSP) units. In addition, an FPGA-based heterogeneous system (CPU + FPGA) using the OpenCL standard has a significant time-to-market advantage compared to traditional FPGA development using lower level hardware description languages (HDLs) such as Verilog or VHDL.

Introduction

The initial era of programmable technologies contained two different extremes of programmability. As illustrated in Figure 1, one extreme was represented by single core CPU and digital signal processing (DSP) units. These devices were programmable using software consisting of a list of instructions to be executed. These instructions were created in a manner that was conceptually sequential to the programmer, although an advanced processor could reorder instructions to extract instruction-level parallelism from these sequential programs at run time. In contrast, the other extreme of programmable technology was represented by the FPGA. These devices are programmed by creating configurable hardware circuits, which execute completely in parallel. A designer using an FPGA is essentially creating a massively-fine-grained parallel application. For many years, these extremes coexisted with each type of programmability being applied to different application domains. However, recent trends in technology scaling have favored technologies that are both programmable and parallel.

Figure 1. Early Spectrum of Programmable Technologies



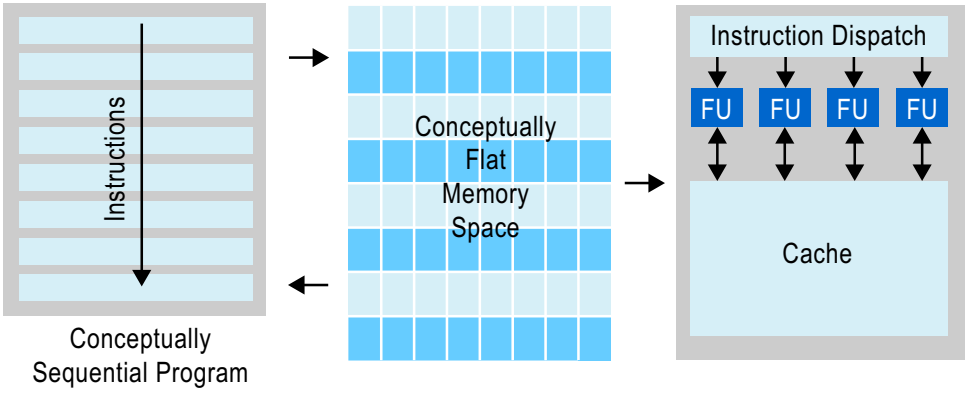
101 Innovation Drive
San Jose, CA 95134
www.altera.com

As the need for performance grew, software-programmable devices that execute a sequential program increasingly relied on two fundamental trends to improve their performance. The first was the scaling of operating frequency as process generations evolved. For a variety of reasons, we can no longer continue the trend of scaling operating voltage lower and increasing operating frequency while maintaining reasonable power densities. This phenomenon known as the "power wall" is driving significant changes to the architecture of all classes of programmable devices.

The second trend that the software programmable devices relied on was the emergence of complex hardware that would extract instruction-level parallelism from sequential programs. As illustrated in Figure 2, a single-core architecture would input a stream of instructions and execute them on a device that might have many parallel functional units. A significant fraction of the processor hardware must be dedicated to extracting parallelism dynamically from the sequential code.

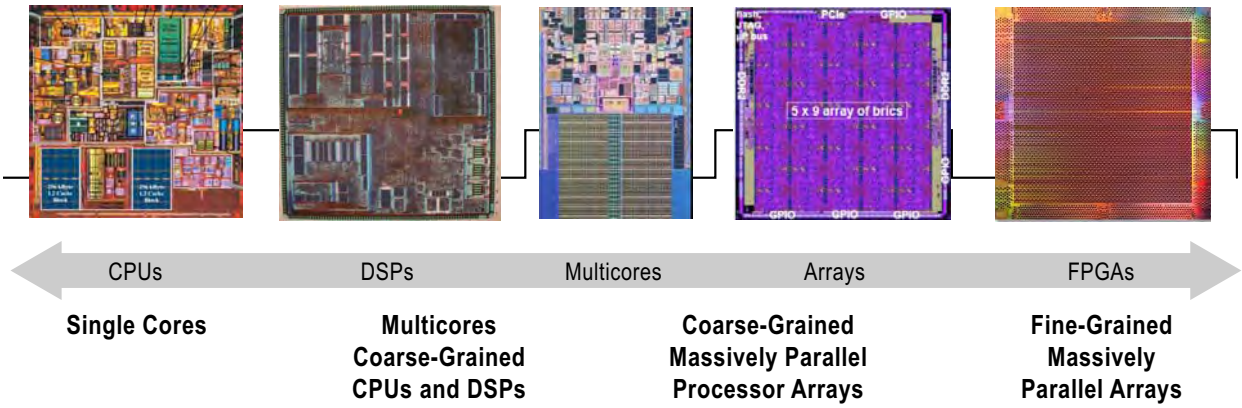
Additionally, hardware attempted to compensate for memory latencies. Generally, programmers create programs without consideration of the processor's underlying memory hierarchy, as if there were only a large, flat, uniformly fast memory. In contrast, the processor must deal with the physical realities of high-latency and limited bandwidth connections to external memory. In order to keep functional units fed with data, the processor must also speculatively prefetch data from external memory into on-chip caches so that the data is much closer to where the computation is being performed. After many decades of performance improvements using these techniques, there have been greatly diminishing returns from these types of architectures.

Figure 2. Single-Core Architectures



Given the diminishing benefits of these two trends on conventional processor architectures, we are beginning to see that the spectrum of software-programmable devices is now evolving significantly, as shown in Figure 3. The emphasis is shifting from automatically extracting instruction-level parallelism at run time to explicitly identifying thread-level parallelism at coding time. Highly parallel multicore devices are beginning to emerge with a general trend of containing multiple simpler processors where more of the transistors are dedicated to computation rather than caching and extraction of parallelism. These devices range from multicore CPUs, which commonly have 2, 4, or 8 cores, to GPUs consisting of hundreds of simple cores optimized for data-parallel computation. To achieve high performance on these multicore devices, the programmer must explicitly code their applications in a parallel fashion. Each core must be assigned work in such a way that all cores can cooperate to execute a particular computation. This is also exactly what FPGA designers do to create their high-level system architectures.

Figure 3. Recent Trend of Programmable and Parallel Technologies



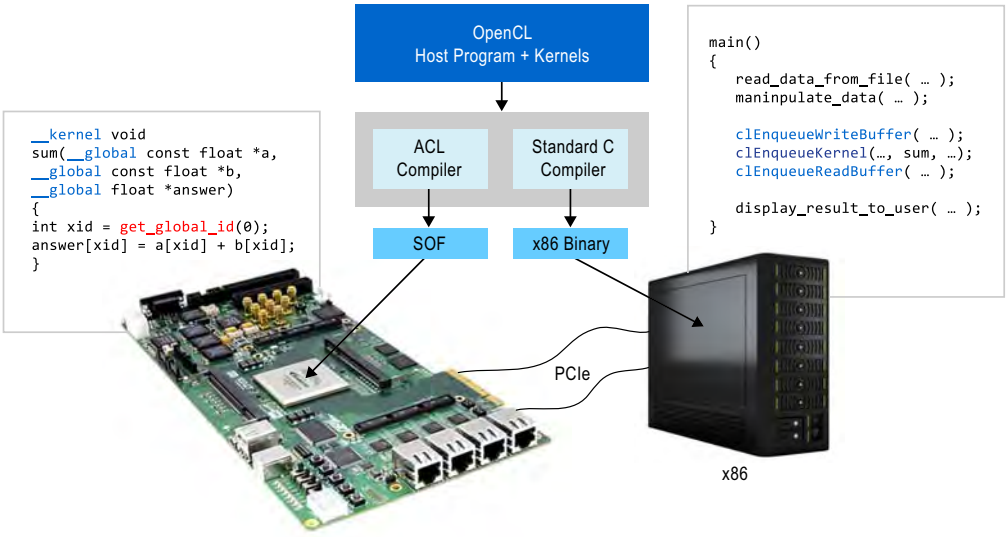
Considering the need for creating parallel programs for the emerging multicore era, it was recognized that there needs to be a standard model for creating programs that will execute across all of these quite different devices. The lack of a standard that is portable across these different programmable technologies has plagued programmers. In the summer of 2008, Apple submitted a proposal for an OpenCL (Open Computing Language) draft specification to The Khronos Group in an effort to create a cross-platform parallel programming standard. The Khronos Group consists of a consortium of industry members such as Apple, IBM, Intel, AMD, NVIDIA, Altera, and many others. This group has been responsible for defining the OpenCL 1.0, 1.1, and 1.2 specifications. The OpenCL standard allows for the implementation of parallel algorithms that can be ported from platform to platform with minimal recoding. The language is based on C programming language and contains extensions that allow for the specification of parallelism.

In addition to providing a portable model, the OpenCL standard inherently offers the ability to describe parallel algorithms to be implemented on FPGAs, at a much higher level of abstraction than hardware description languages (HDLs) such as VHDL or Verilog. Although many high-level synthesis tools exist for gaining this higher level of abstraction, they have all suffered from the same fundamental problem. These tools would attempt to take in a sequential C program and produce a parallel HDL implementation. The difficulty was not so much in the creation of a HDL implementation, but rather in the extraction of thread-level parallelism that would allow the FPGA implementation to achieve high performance. With FPGAs being on the furthest extreme of the parallel spectrum, any failure to extract maximum parallelism is more crippling than on other devices. The OpenCL standard solves many of these problems by allowing the programmer to explicitly specify and control parallelism. The OpenCL standard more naturally matches the highly-parallel nature of FPGAs than do sequential programs described in pure C.

Brief Overview of the OpenCL Standard

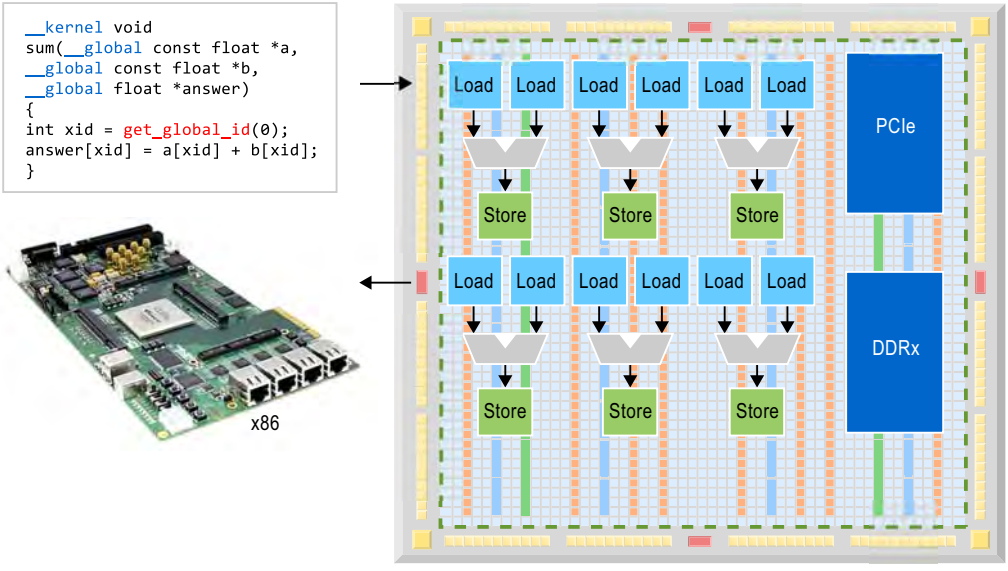
OpenCL applications consist of two parts. The OpenCL host program is a pure software routine written in standard C/C++ that runs on any sort of microprocessor. That processor may be, for example, an embedded soft processor in an FPGA, a hard ARM processor, or an external x86 processor, as depicted in Figure 4.

Figure 4. OpenCL Overview



At a certain point during the execution of this host software routine, there is likely to be a function that is computationally expensive and can benefit from the highly parallel acceleration on a more parallel device: a CPU, GPU, FPGA, etc. This function to be accelerated is referred to as an OpenCL kernel. These kernels are written in standard C; however, they are annotated with constructs to specify parallelism and memory hierarchy. The example shown in Figure 5 performs the vector addition of two arrays, *a* and *b*, while writing the results back to an output array answer. Parallel threads operate on the each element of the vector, allowing the result to be computed much more quickly when it is accelerated by a device that offers massive amounts of fine-grained parallelism such as an FPGA. The host program has access to standard OpenCL application programming interfaces (APIs) that allow data to be transferred to the FPGA, invoking the kernel on the FPGA and transferring the resulting data back.

Figure 5. Example of OpenCL Implementation on an FPGA



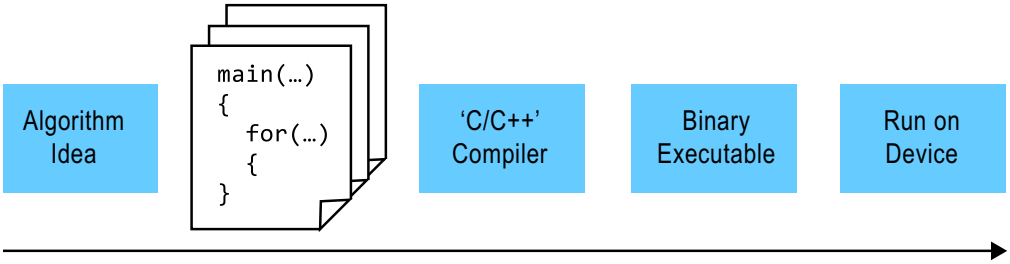
More details of the OpenCL standard can be found on The Khronos Group's website (www.khronos.org/opencl/).

Unlike CPUs and GPUs, where parallel threads can be executed on different cores, FPGAs can offer a different strategy. Kernel functions can be transformed into dedicated deeply pipelined hardware circuits that are inherently multithreaded using the concept of pipeline parallelism. Each of these pipelines can be replicated many times to provide even more parallelism than is possible with a single pipeline. As shown in Figure 5, the vector addition kernel could be implemented by cascading functional units to implement each operation in the OpenCL description and replicated to meet the application's throughput and latency requirements. Although a simple representation has been shown, each functional unit may be deeply pipelined to ensure that the operating frequency of the resulting circuit is fairly high. In addition, the compiler can create the circuitry to manage the communication to the external system. In this example, DDRx controllers and PHYs are connected to the kernel to allow it to access large off-chip arrays with high efficiency. Similarly, PCI Express® (PCIe®) IP is automatically instantiated and connected to the kernel so that an x86 host can communicate with the FPGA accelerator via the OpenCL APIs.

Benefits of Implementing the OpenCL Standard on an FPGA

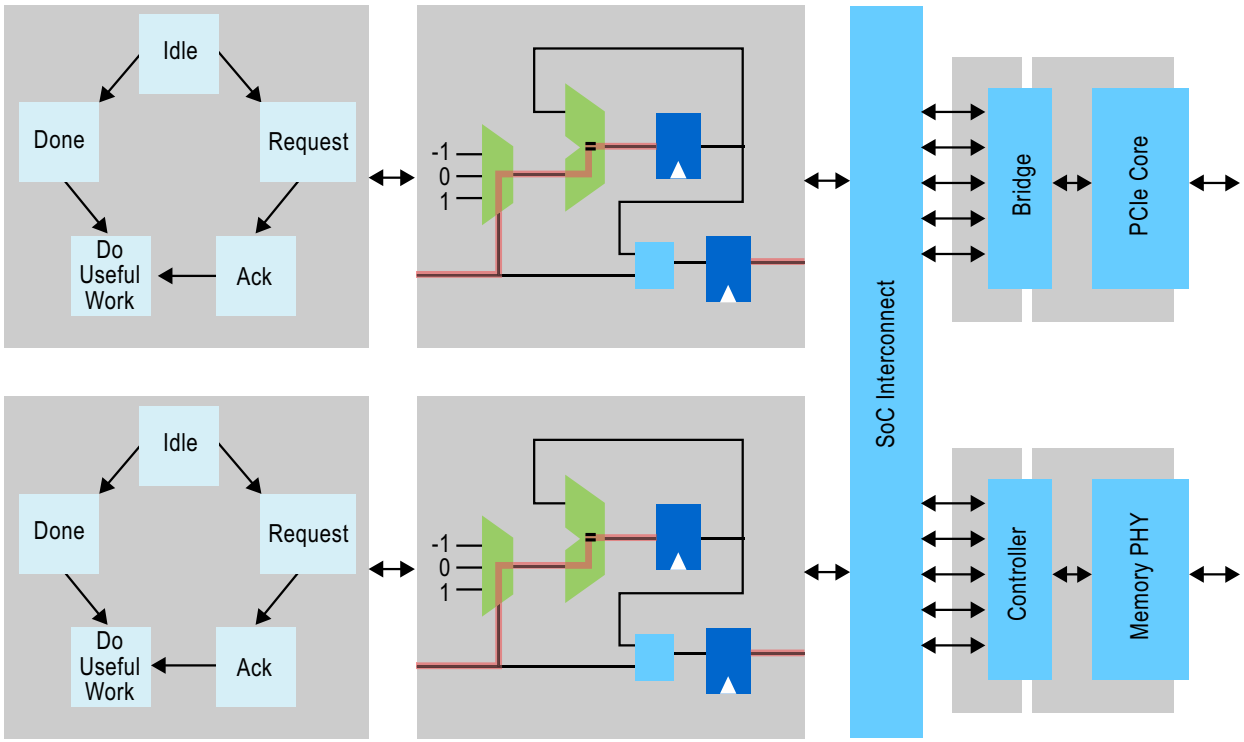
The creation of designs for FPGAs using an OpenCL description offers several advantages in comparison to traditional methodologies based on HDL design. The most significant of these is shown in Figure 6. Development for software-programmable devices typically follows the flow of conceiving an idea, coding the algorithm in a high-level language such as C, and then using an automatic compiler to create the instruction stream.

Figure 6. Software Programmer's View



This approach can be contrasted with traditional FPGA-based design methodologies. Here, much of the burden is placed on the designer to create cycle-by-cycle descriptions of hardware that are used to implement their algorithm. The traditional flow, shown in Figure 7, involves the creation of datapaths, state machines to control those datapaths, connecting to low-level IP cores using system level tools (e.g., SOPC Builder, Platform Studio), and handling the timing closure problems since external interfaces impose fixed constraints that must be met. The goal of an OpenCL compiler is to perform all of these steps automatically for the designers, allowing them to focus on defining their algorithm rather than focusing on the tedious details of hardware design. Designing in this way allows the designer to easily migrate to new FPGAs that offer better performance and higher capacities because the OpenCL compiler will transform the same high-level description into pipelines that take advantage of the new FPGAs.

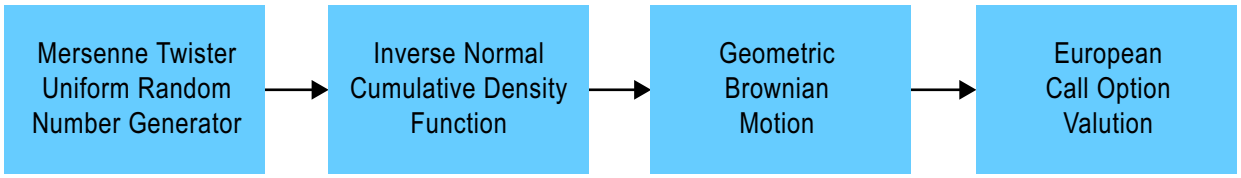
Figure 7. FPGA Design Methodology



Case Study: the Monte Carlo Black-Scholes Method

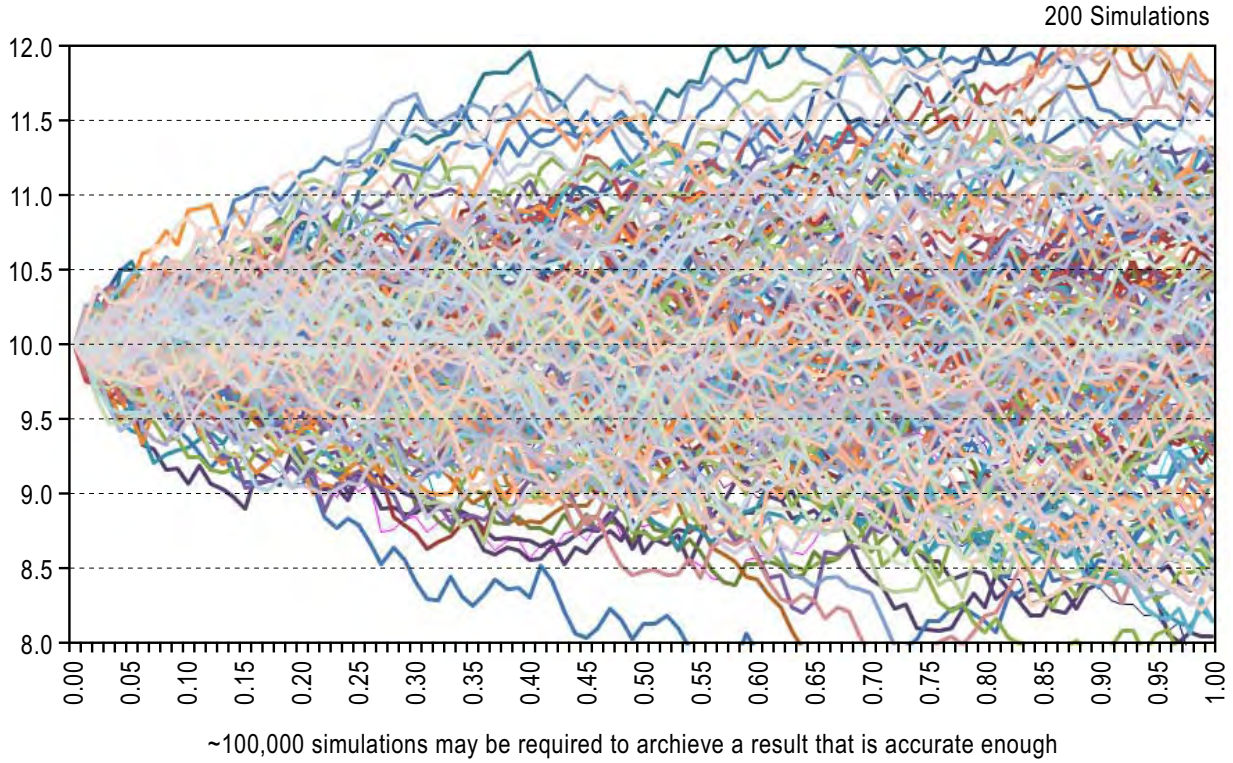
One of the most important benchmarks in financial markets is the computation of option prices via the Monte Carlo Black-Scholes method. The technique is based on conducting random simulation of the underlying stock price and averaging the expected payoff over millions of different paths. An example of such simulations is shown graphically in Figure 8.

Figure 8. Monte Carlo Simulation



The high-level structure of the algorithm that performs this computation is shown in Figure 9. The Mersenne twister random number generator is first used to create values that are uniformly distributed. This sequence of random numbers is fed into an Inverse Normal Cumulative density function to produce a normally distributed sequence. These random numbers are then used to simulate the movement of the stock prices using Geometric Brownian motion. At the end of each simulation path, the call option payoff is recorded and averaged to produce an expected value for the payoff. The entire algorithm can be implemented in approximately 300 lines of OpenCL code that is portable from FPGA to CPU to GPU.

Figure 9. Algorithm Structure



Utilizing an OpenCL framework developed for Altera® FPGAs produces excellent benchmark results, as shown in Table 1. In contrast to a comparable GPU, the OpenCL framework targeting a Stratix® IV FPGA EP4SGX530 exceeds the throughput of both a CPU and a GPU. In addition to greater throughput, FPGA solutions consume one-fifth the power of comparable GPUs when executing the same code, by conservative estimates. This combination of speed and power efficiency slashes energy requirements for compute-intensive applications.

Table 1. Monte Carlo Black-Scholes Results

OpenCL Monte Carlo Black-Scholes	Quad Core Xeon	Comparable GPU	Stratix IV 530
Simulations per second	240M	950M	2,200M
Peak GFLOPS of device	160	500	200

Conclusion

Utilizing the OpenCL standard on an FPGA may offer significantly higher performance and at much lower power than is available today from hardware architectures (CPU, GPUs, etc). In addition, an FPGA-based heterogeneous system (CPU + FPGA) using the OpenCL standard has a significant time-to-market advantage compared to traditional FPGA development using lower level hardware description languages (HDLs) such as Verilog or VHDL. Altera joined The Khronos Group in 2010 and is an active contributor to the standard. To stay up to date on Altera's OpenCL program for FPGAs, please register at www.altera.com/opencl.

Further Information

- Altera's OpenCL Program: www.altera.com/opencl
- The Khronos Group—The OpenCL Standard: www.khronos.org/opencl/

Acknowledgements

- Deshanand Singh, Supervising Principal Engineer, Software and IP Engineering, Altera Corporation

Document Revision History

Table 2 shows the revision history for this document.

Table 2. Document Revision History

Date	Version	Changes
November 2011	1.0	Initial release.