Compiling Software Code to FPGA-based Application Accelerator Processors

David Gardner *and* **Doug Johnson**, *Celoxica, Inc.*

ABSTRACT: *We present a programming environment that enables developers to quickly map and implement compute-intensive software functions as hardware accelerated components in an FPGA. As part of the application acceleration subsystem, these reconfigurable FPGA devices provide parallel processing performance that can deliver super linear speed up for targeted applications. This paper will illustrate how supercomputer developers can quickly harness this potential using their familiar design methodologies, languages and techniques.*

KEYWORDS: Programming, Programming Tools, FPGA, Algorithm, C-Synthesis, XD1, Accelerated Computing, Reconfigurable Computing

1. Introduction

Software engineers typically write code using C and C++ and compile the code to general-purpose processors (GPPs) or digital signal processors (DSPs) for execution. These high-level languages provide a compact and portable way for software engineers to develop high-performance applications that run on processors. However, the computational requirements of algorithms are forcing application developers to look at new paradigms to partition code that run on both processors and dedicated hardware acting as co-processors. These co-processors can be field-programmable gate array devices (FPGAs) that accelerate the complex functions of algorithms in reconfigurable hardware.

Traditionally, complex FPGA designs have been architected and implemented using hardware description languages (HDLs) such as VHDL and Verilog HDL. The use of C and C++ for hardware design facilitates the partitioning of resources between software (SW) and hardware (HW), and fosters both hardware-software co-design and code reuse. Celoxica's C-based hardware design language, Handel-C, and the GUI-based DK Design Suite fuses system verification, HW/SW co-design and Handel-C language synthesis into a single flow targeting programmable logic implementations allowing users to have an immediate access to efficient hardware implementations of C-based functional descriptions.

The C algorithm used as an example in this paper is a search kernel whereby a stream of unsorted, 4-dimensional points are checked for intersection against a set of hypercubes with an equal number of dimensions. This work demonstrates that by using a Cray XD1 Supercomputer augmented with FPGAs the search kernel is capable of executing at a speed over 113 times greater than the standard high-end processor. [1]

2. FPGA Programming Environment Using C

To provide designers with the capability to program the FPGAs, the DK Design Suite was developed by Celoxica as a software-centric set of tools and a methodology to develop applications for the FPGAs that are based on C-algorithms rather than VHDL or Verilog

HDL. The design methodology enables designers to quickly and efficiently accelerate software algorithms and system bottlenecks in parallel hardware. The sequential C algorithm is migrated to a parallel hardware implementation using the DK Design Suite and accurate high performance hardware code is automatically generated. By using Celoxica's implementation process, coding remains at the C level throughout, providing a common language and methodology for HW and SW design. Using this high-level language implementation flow the programmer also benefits of from fast simulation for HW-SW co-verification.

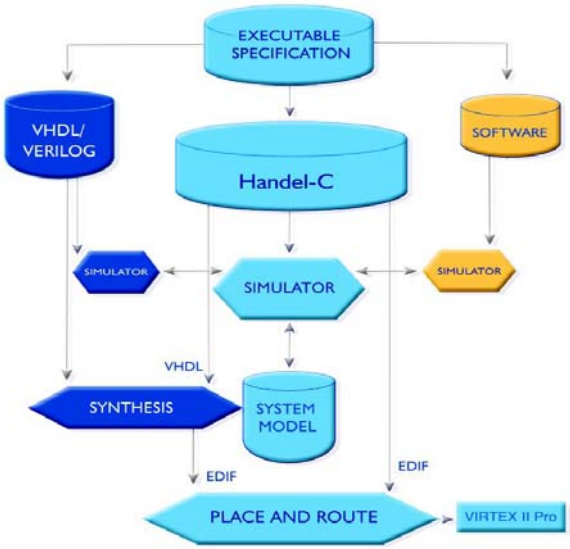

Figure 1: DK Design Suite Flow for FPGAs

Figure 1 shows the design flow used to develop the search kernel application described in this paper. DK supports a mixed abstraction level modelling and simulation environment based on Handel-C, C and C++ source code are supported by DK. The C/C++ functions are imported as externals, providing a clear separation between parts of the design compiled to a processor and those functions in Handel-C intended for direct implementation on the FPGA.

After verifying the characteristics and functionality of the system at a high level (C/C++), individual modules are selected for hardware implementation using Handel-C. The hardware descriptions in Handel-C can be simulated within the completely specified system model with the following advantages:

- During early architectural design space exploration, functional verification with mixed language descriptions enables the designer to consider many different block-based partitioning possibilities.

- Successive refinement and interactive transformation of relevant software algorithms within a system module to functions for hardware implementation allows simulation at differing levels of detail with consequent speed implications through very fast behavioral (C/C++) and fast cycle accurate (Handel-C) simulations.

- A test environment can be written at a high level of abstraction in C/C++ and developed into sophisticated models, incorporating input from and output to other modelling and simulation tools such as Matlab.

- The same environment may be used throughout the stages of a hardware module design. [2]

3. Handel-C Language

Handel-C is a subset of ISO-C (ANSI-C) with the necessary constructs added for hardware design (see Figure 2). Handel-C algorithms are coded in a sequential software style with a 'par' construct to implement parallelism. A channel 'chan' statement allows for communication and synchronization between parallel branches of the program and channels function across multiple clock domains with semantics based on unbuffered, synchronous send and receive.

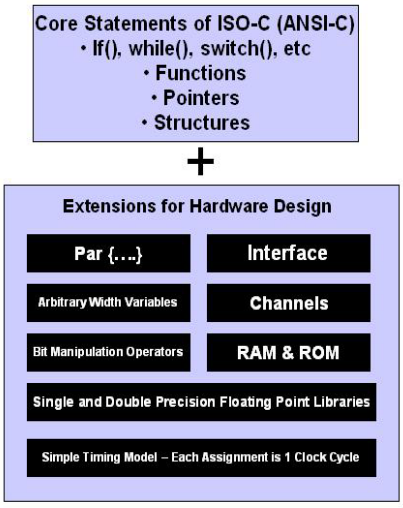

Figure 2: Extensions to ANSI-C for Hardware Design

Handel-C's level of design abstraction is above register transfer level (RTL) but below behavioral. In Handel-C each assignment infers a register and takes one clock cycle to complete, so it is not a fully behavioral language in terms of timing. However, this simple expression of scheduling in Handel-C gives the designer full control over the clock cycle accuracy of the implementation. This allows for fast and efficient exploration of different design architectures.

A significant advantage of this approach is the determinism of the result. The source code completely describes the scheduling and the most complex expression determines the clock period. This is in contrast with HDL synthesis where tools make significant use of constraint attributes.

Handel-C is intended for the design of synchronous logic and is particularly suited to the implementation of complex algorithms in FPGAs. Some of the advantages of such an approach include:

- No requirements to describe an explicit state machine. The state machine is automatically generated from an algorithmic description of the circuit in terms of parallel and sequential blocks of code. By allocating one clock cycle per assignment and using par and seq constructs nested in any way in combination with software flow control statements, Handel-C can create state machines of virtually limitless size.

- Automatic scheduling of parallel and sequential blocks of code. The code following a group of parallel blocks of code is scheduled only after that whole parallel block has completed.

- Channels for communications and scheduling. Channels provide communications between parallel blocks of code, even if they are in different clock domains. They also provide scheduling or synchronization as the receiving and transmitting blocks of code can only proceed after the communication has completed.

- Assignment and delay statements take one clock cycle where combinatorial expressions are computed between clock edges.

- Automatic generation of clocks, clock enables and resets for the synthesized logic. [3]

4. Handel-C for High Level Design and Low Level Control

Handel-C allows the designer to describe the behavior of the intended hardware in the same sense as a software programmer describes the intended behavior of a processor executing a program. This is fundamentally different from using a standard C/C++ syntax to describe the structure of a hardware implementation.

Handel-C is used for clock cycle accurate modelling and high-level design of hardware. If the system design has been done in C then the translation to Handel-C is greatly simplified because of the similar syntax, and most importantly, the same level of abstraction. Moreover developing parallel and pipelined behavior from a sequential start point is intuitive and allows for rapid design space exploration. Handel-C's level of abstraction also allows low-level control over designs. This means that clock accurate timing is implicit in the language and combinational logic may be implemented using data types that represent ports, busses and signals.

5. Developing Applications on Cray's XD1 using the DK Design Suite

For this research, the Cray XD1 supercomputer was chosen as the platform to benchmark the search kernel algorithm running on an AMD Opteron processor against the algorithm running on an FPGA. Figure 3 shows the architecture of the Cray XD1 supercomputer. The FPGAs are a part of the application acceleration system built into the XD1. [4]

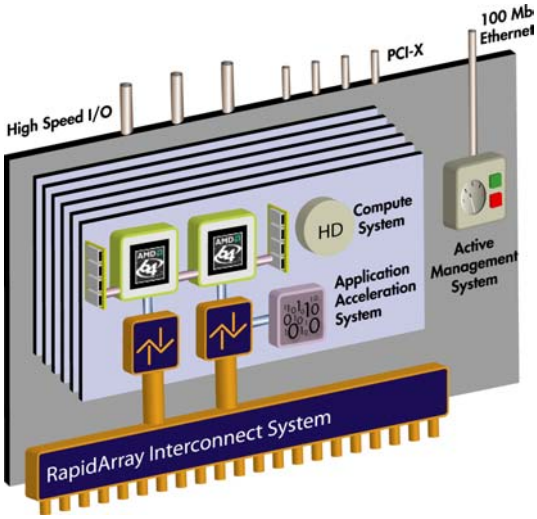

Figure 3: Architecture of the Cray XD1 Supercomputer

The XD1 incorporates a design that directly connects the AMD Opteron compute processors with Xilinx Virtex FPGAs over high bandwidth, low latency links and integrates the FPGAs through hardware, operating system and communications management software. The Rapid Array interconnect directly connects the FPGA to the AMD Opteron processors through a bi-directional bus, with 3.2 GB per second bandwidth (1.6 GB per second in each direction). The application acceleration system includes 16 MB of Quad-Data Rate (QDR) Static RAMs directly connected to the FPGA through a 12.8 GB per second interface. This large bandwidth enables the FPGA to stay busy while the host Opteron processor concurrently transfers data to and from the FPGA system at 3.2 GB per second. Figure 4 depicts the architecture of the FPGA, memory interfaces and the high-speed interconnect.[5]

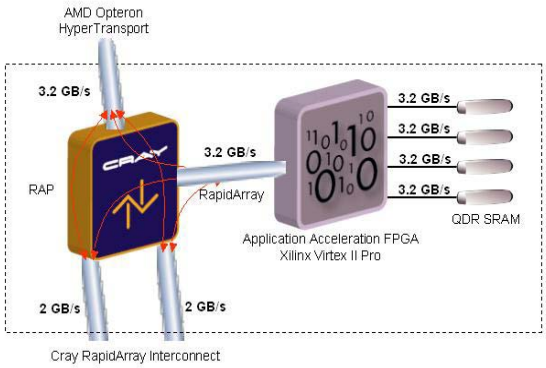

Figure 4: XD1 Application Acceleration System

Figure 5 illustrates the Handel-C-based design flow within DK for the Cray XD1 that was used to develop the search kernel algorithm application. In this flow, user applications are written directly in Handel-C or ported to Handel-C from software ANSI C code or other high level languages such as Matlab M-code. Each software environment has predefined test benches and scripts to allow the designer to concentrate on the algorithm development in Handel-C. Design, debugging, simulation, and verification can all be performed prior to RTL and EDIF (Electronic Design Interchange Format) netlist compilation and synthesis using Celoxica's Handel-C compiler and synthesis tool embedded in DK Design Suite. Xilinx XST uses the EDIF netlist generated from DK to place and route (PAR) the design. Optionally, further verification can be done on the design using RTL simulators such as Mentor Graphic's ModelSim or Aldec's Riviera. [6]

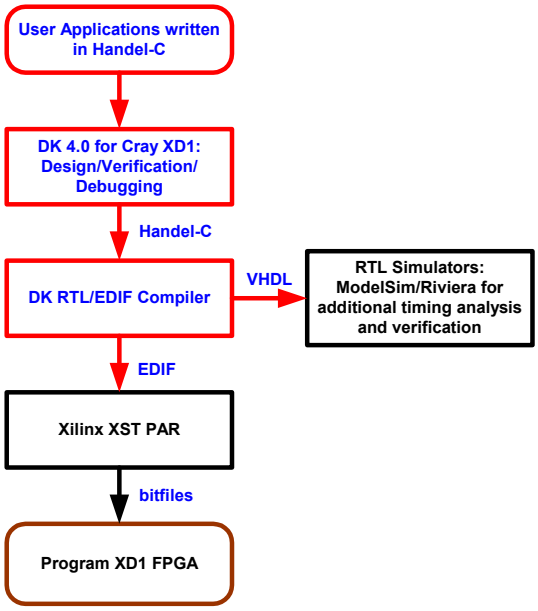

Figure 5:Handel-C Design Flow with DK4.0 for XD1

6. Search Kernel Algorithm Example

A search kernel algorithm was chosen to show the benefits of accelerated computing using FPGA technology. This algorithm has a set of *p* points and a set of *h* hypercubes, both of which having *d* dimensions. The system identifies each point-hypercube combination where the point lies completely within the hypercube. The points are effectively streamed into the search algorithm and it is not practical to sort them. Consequently, many well-known range search optimizations cannot be used. For the purposes of this research, the number of points will be limited to around 10^7. For this research, the number of hypercubes, h, was set at 64 and the number of dimensions was set at 4. All of the point and hypercube parameters are represented by 16-bit positive integers. The result of the search kernel should be represented as bit-vectors where each bit represents a hit/miss flag for every point-hypercube combination. These resulting bit-vectors must be written back into the processor's main memory. [1]

The probability that a given point will intersect a given hypercube can substantially skew the acceleration in favor of the FPGA or processor depending on whether the probability is high or low respectively. To understand this characteristic, consider how the processor operates. As the processor evaluates each point-hypercube combination it does so by evaluating the dimensions sequentially, one at a time. So if the first dimension of the point does not intersect with the first dimension of the hypercube then the processor can quickly move on to the next point-hypercube combination. However, if there is a

match on the first dimension then the processor must evaluate the second dimension and so on. Consequently the processor will achieve a higher throughput when the probability of intersection is low. In contrast, the FPGA is capable of checking each of the four dimensions of a given point-hypercube combination in parallel and so the time to evaluate each point-hypercube combination is constant, regardless of the probability of intersection. To balance the effect of this factor, sets of points & hypercubes are chosen so that the probability of a given point-hypercube combination is equally likely to miss as it is to hit. That is, the probability that a point will intersect a hypercube on all four dimensions is 50%.

7. Test Configuration

Both software and hardware implementations of the search kernel were created. In both cases an effort was made to write the code cleanly and efficiently. However, no optimization "tricks" were employed that might result in obfuscated code that would be difficult to maintain. The processor code is written in C and compiled using the gcc compiler with the –O3 switch. The FPGA code is written in Handel-C and compiled with the DK Design Suite compiler.

The processor code is executed on a Cray XD1 Supercomputer equipped with an AMD 2.2GHz Opteron and a Xilinx Virtex II/Pro FPGA that is connected to the Opteron via the HyperTransport link. The HyperTransport link is moving 16-bit words at 400MHz, which yields an effective bandwidth between the processor and FPGA of approximately 3.2GB/s bi-directional. For each test run the test harness software running on the host processor goes through five phases, which are:

1. Create a set of input data for the points and hypercubes, selecting values such that there is a 50% probability of point-hypercube intersection.
2. Check for point-hypercube intersections using the FPGA
3. Check for point-hypercube intersections using the Opteron
4. Validate the FPGA & Opteron results against one another
5. Report results

In order to determine acceleration the execution times steps 2 & 3 are measured independently. Steps 2 & 3 both start execution with the same set of point & hypercube data in the same memory space, and they each write the results back into the processor's main memory as an array of bit-vectors.

8. Hypercube Range Search Implementation

Considering that the number of points is very large compared to the number of hypercubes, the best implementation for the kernel is one where the hypercube information is loaded into the FPGA first, after which the points are streamed into the FPGA, checked for intersection against the hypercubes, and the results streamed back to the processor memory.

8.1 Input data

Since it is known that the FPGA will receive data at the rate of approximately 1.6GB/s, and that the data structure for each point consists of four, 16-bit values, the maximum rate at which points can be streamed is 200 Mega-points/s. Targeting an FPGA clock speed of 200 MHz implies a processing rate of one point per clock.

8.2 Output data

One bit is needed to indicate a hit/miss condition for each point-hypercube pair. Because the HyperTransport link is 1.6GB/s from the FPGA to the processor, it can be shown that the bandwidth will support 64 (8 * 1.6 / 0.2) bits of result data per clock cycle. Assuming that the system is processing one point per clock cycle, we see that the bandwidth supports checking a point against 64 hypercubes in each clock cycle.

The result of the above analysis is that the search kernel implemented in the FPGA is optimally sized at 64, 4-dimensional hypercubes and that it compares a single point against each dimension of each hypercube simultaneously to produce a 64-bit result set every clock cycle. That is to say that on every clock cycle the FPGA receives a single 4-dimensional point from the processor and writes a 64-bit vector back to the processor.

The kernel configuration described above yields an optimum throughput of 200 million, 4-dimensional point–hypercube intersection tests per second, assuming that the FPGA clock speed is 200MHz. In this research, a previous generation Xilinx Virtex-II FPGA is used and the actual clock speed is 140MHz. This means that the FPGA search kernel is capable of performing (140,000,000 * 4 * 2 * 64 = 71,680,000,000) approximately 72 billion, 16-bit integer comparison operations per second.

Finally, we can calculate the theoretical time to process a given set of points. Considering that the number of points of much greater than the number of hypercubes, the time it takes to load the hypercube

information into the FPGA is omitted and assume that the time to process the points is effectively the best-case time of the hardware. In this research, a set of 128k points is processed 500 times. From this information we can calculate run time in hardware of at least (500 * 128 * 1024 / 140,000,000) or 0.47 seconds.

8.3 Performance Results

Using testing process outlined above, Table 1 shows the performance metrics were captured for 5 runs.

Run Number	FPGA (ms)	Opteron Time (ms)	FPGA Acceleration Factor
1	1,060	119,470	112.7
2	1,060	119,480	112.7
3	1,060	119,480	112.7
4	1,050	119,490	113.8
5	1,050	119,480	113.8
Average	**1,056**	**119,480**	**113.1**

Table 1: FPGA vs. Opteron

The conclusion is that by using the XD1 configured with a Xilinx Virtex-II FPGA to execute the search kernel algorithm, an acceleration of up to 113X over an AMD 2.2GHz Opteron is easily achievable (see Figure 6).

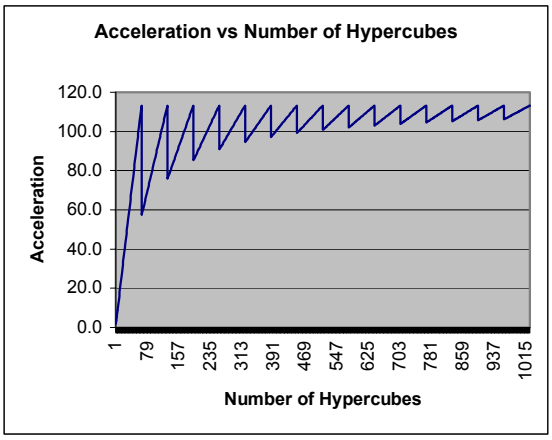

Figure 6: Acceleration of Search Kernel Algorithm Versus Number of Hypercubes

8.4 Your Mileage May Vary

As with any benchmark, there are many ways to skew the results, both positively and negatively. We have tried to be fair in this research by using a point-hypercube intersection probability of 50% and by using code that is written in a way that is easy to maintain. However, there are also other factors which one should keep in mind when considering the possible acceleration in another system:

- This design is running at 140 MHz in the FPGA. Using this FPGA with additional pipelining or perhaps using a current-generation FPGA such as the Virtex-4, the clock frequency could be increased to a maximum of 200MHz. At that rate the peak acceleration factor would be increased from 113 to 162.

- As the number of hypercubes increases the acceleration factor will vary. On the processor side, the additional processing time is directly related to the number of hypercubes. If 100 hypercubes are used instead of 64 the average runtime on the Opteron would likely be 1.56 (100/64) times longer. However, since the FPGA uses a 64-hypercube kernel, for any number of hypercubes from 65 to 128 the execution time would effectively double. Again, assuming 100 hypercubes we would then expect the acceleration factor to be 88 (113.1 * (100/64/2))

- As the number of dimensions increases, the kernel will need to be called more times – once for each set of 4 dimensions. Also, the results from each call will need to be merged. More research is needed to determine the best way to handle this step - either in an FPGA or a processor. [1]

9. Conclusion

This research has demonstrated a kernel for implementing a multi-dimension range search algorithm in an FPGA that offers a peak acceleration of over 113X compared to a high-end Opteron processor. The implementation using Celoxica's programming environment produced an efficient hardware implementation in a very rapid timeframe. The implementation maintained a familiar software development paradigm, while giving access to this acceleration capability that can be leveraged by reconfigurable hardware.

The search kernel algorithm described in this paper is one possible high performance algorithm that can be implemented using Celoxica's programming environment. There are a multitude of other applications where Celoxica's C-based compiler technology and Cray's FPGA-augmented supercomputers can be used to accelerate today's most challenging computational algorithms.

References:

[1] David Gardner, “Hardware Acceleration of a Multi-Dimensional Range Search”, Celoxica Inc., Austin, Texas, USA
[2] Dr. Stephen Chappell, Chris Sullivan, “Using the Handel-C High Level Language for Field Programmable System on Chip (FPSoC) Design and Implementation”, Celoxica Ltd., Oxford, UK
[3] Handel-C Users Manual, Celoxica, Ltd.
[4] Cray XD1 Datasheet, Cray Incorporated, Seattle, WA. http://cray.com/downloads/Cray_XD1_Datasheet.pdf
[5] Presentation, “The Cray XD1,” Cray Incorporated
[6] Long Dai, David Gardner, Phil Keene, “White Paper: Celoxica DK4.0 for Cray XD1 FPGA”, Celoxica, Inc.

About the Authors:

David Gardner is Senior System Architect, Celoxica Inc., Austin TX. David can be reached by phone at (512) 795-8170 or by email at david.gardner@celoxica.com. Doug Johnson is Business Development Manager, Celoxica Inc., Redondo Beach, CA. Doug can be reached by phone at (310) 543-2468 or by email at doug.johnson@celoxica.com.