

Accelerating Scientific Applications Using FPGAs

A physicist uses C-to-hardware tools with the Cray XD1 supercomputer and Xilinx FPGAs to prototype high-performance algorithms.

by Peter Messmer, Ph.D.
Research Scientist
Tech-X Corporation
messmer@txcorp.com

Ralph Bodenner
Senior Software Engineer
Impulse Accelerated Technologies
ralph.bodenner@impulsec.com

Xilinx® FPGAs are increasingly used alongside traditional processors in a variety of hybrid parallel computing systems. In systems such as the Cray XD1 supercomputer, FPGAs play a supporting role by providing massive instruction-level parallelism and by fundamentally changing the way that key algorithms are processed.

Although many scientific applications could benefit from these hybrid systems, the complexity of the hardware programming process reduces the potentially large user community to a few hardware-savvy scientists. The promise of latest-generation software-to-hardware translation tools is to enable domain scientists (chemists, biologists, or physicists) with little or no hardware programming experience to take advantage of FPGA-enhanced systems. In this article, we'll describe the experience of one physicist using such tools to develop an FPGA-accelerated application targeting a Cray XD1 system.

Fortunately, software-to-hardware tools now exist that allow FPGA-based scientific algorithms to be described using familiar, software-oriented methods.

FPGAs for Application Acceleration

Programming software algorithms into FPGA hardware has traditionally required specific knowledge of hardware design methods, including hardware description languages such as VHDL or Verilog. Although these methods may be productive for hardware designers, they are typically not suitable for domain scientists and higher level software programmers. When considering FPGAs for high-performance computing, one challenge has been finding a way to move critical algorithms from a pure software implementation (written in C) into an FPGA, with minimal FPGA hardware design knowledge.

Fortunately, software-to-hardware tools now exist that allow FPGA-based scientific algorithms to be described using familiar, software-oriented methods. Using these tools, an application and its key algorithms can be described in standard C, with the addition of relatively simple library functions to specify inter-process communications. The critical algorithms can then be compiled automatically into HDL representations, which are subsequently synthesized into lower level hardware targeting one or more FPGA devices. Although it still may require a certain level of FPGA knowledge and in-depth hardware understanding to optimize the application for the highest possible performance, the formulation of the algorithm and initial testing can now be left to the domain scientist.

Case Study: The FDTD Algorithm

The finite-difference time-domain (FDTD) algorithm is a well-established tool in computational electrodynamics. Applications of FDTD include propagation problems (such as wave propagation in complex urban environments), photonics problems, high-frequency component modeling, scattering problems (like determination of radar signatures), electromagnetic compatibility

problems (including transmitter-tissue interaction), and plasma modeling.

A particular form of the FDTD algorithm, the Yee-FDTD algorithm, discretizes the electromagnetic field in space and advances both Faraday's and Ampere's laws in time. Figure 1 illustrates this operation for one cell in the computational grid, as well as the spatial location of the field components on a staggered grid. An equivalent update operation must be performed for each field component for all cells on the computational grid.

The spatial grid has to be fine enough to accurately sample the shortest wavelength. In addition, the fastest waves in the system should not propagate more than one grid-cell per timestep, which sets an upper limit on the timestep. FDTD simulations therefore consist generally of large computational grids and a large number of timesteps.

The FDTD algorithm can be summarized as:

```
for i = 0, n steps
  apply boundary condition
  update E field:  $E_{\text{new}} = E_{\text{old}} + c2 * \text{curl } B * dt$ 
  update B field:  $B_{\text{new}} = B_{\text{old}} - \text{curl } E * dt$ 
end
```

The large number of cells and the high degree of parallelism in the computations make the FDTD an ideal candidate to benefit from FPGA hardware acceleration.

This project had two goals: first, we wanted to find a way to move this critical algorithm from a pure software implementation (expressed in C language) into the FPGA. Second, we wanted to investigate if modern software-to-hardware tools would allow a scientist with minimal FPGA hardware design knowledge to port other, similar algorithms to an FPGA-enhanced platform. To achieve these goals, the hardware-savvy engineer participating in this project should mainly function as a consultant.

From Software to FPGA Hardware

The first step in performing a software-to-hardware conversion was to select a target platform. We chose a Cray XD1 super-computer system, featuring FPGA application accelerators on each compute node. Each application accelerator in the XD1 system comprises one or more Xilinx® Virtex™-II Pro or Virtex-4 FPGAs.

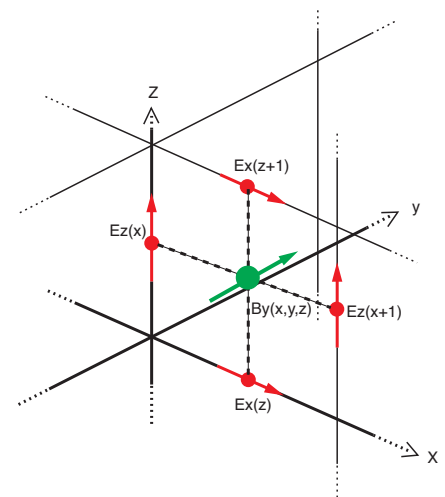


Figure 1 – The Yee-FDTD computational grid. The E field components are centered on the cell edges, the B field components at the cell faces. The dashed line indicates the computation of the y component of curl E, through finite differences, which is used to update B_y .

Handling the application accelerator and loading the FPGA bitstreams was simplified by Cray system tools, making the entire process easier for a software programmer to comprehend.

After choosing the platform, our next step was to select a software-to-hardware translator. We chose Impulse C from Impulse Accelerated Technologies for these experiments because of its emphasis on standard C, its ability to automatically generate and optimize parallel hardware, and its support for the Cray XD1. The Impulse

```

void curlProcess(co_stream inputStream, co_stream outputStream)
{
    co_stream_open(inputStream, O_RDONLY, INT_TYPE(32));
    co_stream_open(outputStream, O_WRONLY, INT_TYPE(32));

    while(!co_stream_eos(inputStream)) {
        co_stream_read(inputStream, &bx, sizeof(co_int32));
        co_stream_read(inputStream, &by, sizeof(co_int32));
        co_stream_read(inputStream, &bz, sizeof(co_int32));
        co_stream_read(inputStream, &bx_plus_y, sizeof(co_int32));
        co_stream_read(inputStream, &bx_plus_z, sizeof(co_int32));
        co_stream_read(inputStream, &by_plus_x, sizeof(co_int32));
        co_stream_read(inputStream, &by_plus_z, sizeof(co_int32));
        co_stream_read(inputStream, &bz_plus_x, sizeof(co_int32));
        co_stream_read(inputStream, &bz_plus_y, sizeof(co_int32));

        ex = (bz_plus_y >>2) - (bz >>2) - (by_plus_z >>2) + (by >>2);
        ey = (bx_plus_z >>2) - (bx >>2) - (bz_plus_x >>2) + (bz >>2) ;
        ez = (by_plus_x >>2) - (by >>2) - (bx_plus_y >>2) + (bx >>2) ;

        co_stream_write(outputStream, &ex, sizeof(co_int32));
        co_stream_write(outputStream, &ey, sizeof(co_int32));
        co_stream_write(outputStream, &ez, sizeof(co_int32));
    }

    co_stream_close(outputStream);
    co_stream_close(inputStream);
}

```

Figure 2 – Summarized C code for the Impulse C hardware process applying the curl operator to the B field.

C tools made it easy to prototype various hardware/software partitioning strategies and to test and debug an application using familiar C development tools.

Partitioning the Application

Identifying the critical sections of the application code is important for any hardware-accelerated algorithm. In our case, the time-consuming part of the FDTD algorithm was the computation of the curl operation, which required several additions and at least one multiplication and division per cell half-update. As it turns out, the updating of individual cells does not depend on the updating of other cells, making this code a good candidate for parallelization.

Impulse C supports the concept of multi-processing, in which multiple parallel processes can be described using the C language and interconnected to form a system of semi-autonomous processing nodes. Various methods of inter-process communication are provided in Impulse C, including streams, shared memory, and signals.

To keep things simple, we started our FPGA-based FDTD algorithm with a plain C implementation of the integer-based curl algorithm, which was isolated into a distinct C-language hardware process. Using Impulse C, this hardware process looks very much like a typical C subroutine, with the addition of Impulse C-stream read and write function calls to provide abstract data communication.

The source code describing this process is summarized in Figure 2.

This splitting of an application into hardware and software processes is relatively straightforward from a programming perspective, but it is also quite communication-intensive; the input and output streams become performance bottlenecks. However, this initial partitioning allowed us to experiment with hardware generation and the hardware/software generation process, as well as to begin understanding how to optimize C code for higher performance.

After desktop software simulation using the Impulse tools, we invoked the software-to-hardware compiler. The automatically generated files included the hardware for the curl function, as well as all required interfaces implementing data streams through Cray's RapidArray interface (see Figure 3). The generated VHDL was then translated into a Virtex-II Pro FPGA bitstream using Xilinx ISE™ tools, following instructions provided by Impulse and Cray. Once again, the Impulse tools abstracted the process, including export to the Xilinx tools, such that we did not need to understand the process in detail. The resulting bitstream and the sources for the software process (which was to run on the Opteron) were transferred to the Cray XD1 platform using tools provided by Cray.

Moving More of the Application to the FPGA

After developing and deploying a single algorithm to the Cray XD1 platform, we were in a position to begin optimizing the larger application. Initial experiments led us to the conclusion that dedicating the application accelerator to computing only the curls would lead to poor performance, given the large amount of data transfer required between the CPU and the FPGA. Fortunately, the high-gate density of the Virtex-II Pro device allowed us to put far more demanding operations onto the FPGA itself.

As a next step, we implemented the entire FDTD algorithm as a hardware process on the FPGA. In this design, the host CPU is responsible only for commu-

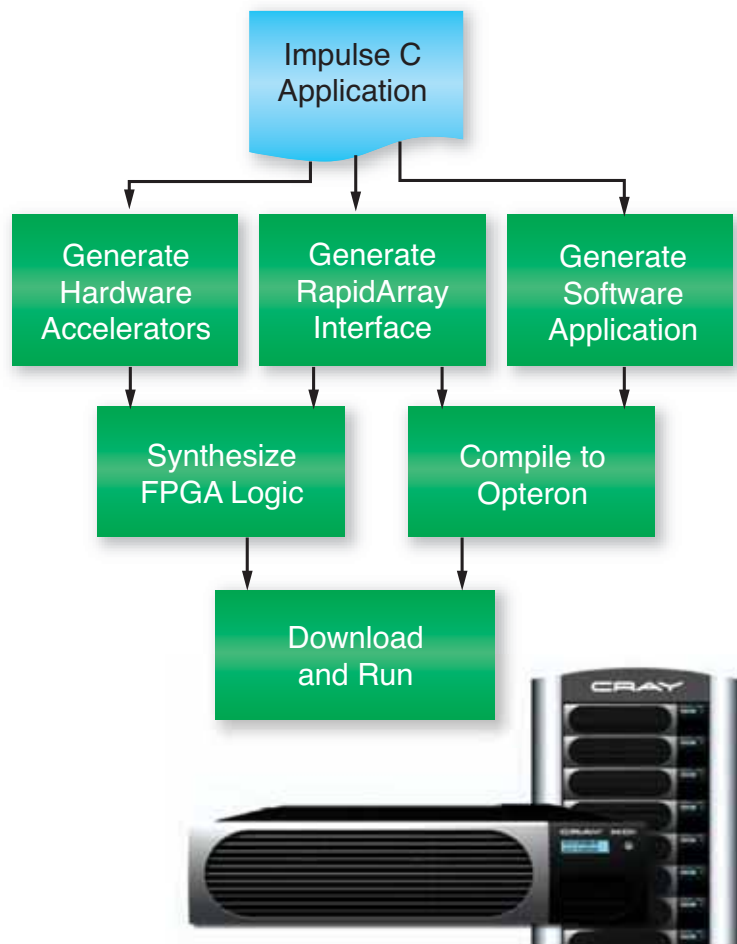


Figure 3 – Impulse C tools allow you to partition an application between hardware and software and generate the required Cray XD1 platform interfaces.

nicating the number of required timesteps to the FPGA and setting the boundary condition in a pre-defined part of the computational domain at each timestep. The FPGA then stores all of the fields and performs both the E- and B-field updates. With this more complete algorithm working in hardware – producing correct and reliable results as observed on the software side – we now have the confidence to try many different partitioning strategies and C-level optimizations to increase overall system performance.

Throughout the optimization process, we treated the generated hardware description as a black box and only performed optimizations on the high-level C source files. The iterative application development, optimization, and proto-

type hardware generation enabled by the Impulse C tools proved to be an enormous time-saver.

Future Optimizations

Now that we have the basic algorithm working on the Cray XD1 platform with its FPGA accelerator – a process that required little more than three weeks of total effort, including the time required to learn the Impulse tools and programming model and to become familiar with the Cray XD1 system – we are now focusing on algorithm optimization. One of the optimizations will be to utilize the high-speed QDR II memories, which will be directly supported in Impulse C using abstract memory block read/block write functions. This will reduce or eliminate the stream communication

bottlenecks that we are currently observing in the system as a whole.

During this optimization phase, we are using more advanced Impulse optimization tools that allow us to analyze the generated logic and automatically generate loop pipelines for increased throughput. What we are learning as we pursue these optimizations is that the high-level language approach to hardware programming typified by Impulse C allows iterative refining of the algorithm and supports the concept of explorative optimization. Interactive design analysis tools, such as the graphical optimizer and cycle-accurate debugger/simulator provided with Impulse C, allow us to estimate the performance of an algorithm before translation and mapping. In this way, creating FPGA accelerators becomes a highly iterative, dynamic process of C coding, debugging, and high-level optimization.

Conclusion

Before the advent of software-to-hardware tools, the creation of an FPGA-accelerated application would have been impractical for all but the most hardware-savvy domain scientists. The Impulse C approach, when combined with the Cray XD1 platform, enabled us to convert prototype algorithms into hardware realizations and to quickly observe results using different methods and styles of C programming. This project has given us confidence that – even as software application programmers – we have the ability to experiment with many different algorithmic approaches and can accelerate scientific applications without relying on the knowledge of an experienced FPGA designer.

For more information on accelerating applications using FPGAs and Impulse C, visit www.impulsec.com.

The authors would like to thank Dave Strenski, Steve Margerm, and others at Cray, Inc. for their assistance and for providing access to the Cray XD1 system, and Roy White at Xilinx for providing access to the ISE tools used in this evaluation project.