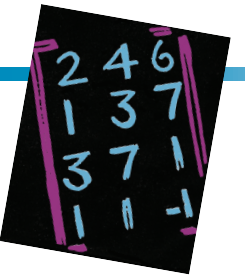
Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu



ACCELERATING SCIENTIFIC APPLICATIONS WITH RECONFIGURABLE COMPUTING

GETTING STARTED

By Volodymyr V. Kindratenko, Craig P. Steffen, and Robert J. Brunner

High-performance reconfigurable computing combines the advantages of the coarse-grain parallel processing provided in conventional multiprocessor systems with the fine-grain parallel processing available in field-programmable gate arrays.

Reconfigurable computing[1] based on the combination of conventional microprocessors and field-programmable gate arrays (FPGAs) has reached a point at which we can substantially accelerate select scientific kernels with the ease of a C/Fortran style of programming. It wasn't until the late 1990s that FPGAs achieved sufficient gate density and functional capability to support the nontrivial, double-precision, floating-point operations that many scientific kernels require, and only recently have high-level languages (HLLs) and code development tools become available that hide the complexity involved in a typical FPGA design implementation cycle. Consequently, high-performance reconfigurable computing (HPRC)—or reconfigurable supercomputing—is a relatively recent technology that rapidly evolves with new systems coming online and new software being written.

Traditional high-performance computing (HPC) vendors, such as SGI and Cray, have introduced several commercial HPRC products, but newcomers, including SRC Computers and Nallatech, have also emerged with viable solutions. All these systems consist of a traditional computer based on general-purpose processors and a separate "accelerator" component built around an FPGA. Although similar in their basic concepts, these individual solutions differ in the accelerator component's design, the coupling between the accelerator and the general-purpose computer system, and the access and control software.

Several tools can compile code written in an HLL directly into the hardware circuitry description. Dividing code between the general-purpose processor and the FPGA accelerator isn't a trivial task[2] and is still the software developer's responsibility. However, once the code-partitioning decision is made, the code developer can implement the algorithm's hardware side on the selected FPGA via the appropriate toolset and the software side on the general-purpose processor via conventional code development techniques. Developers have successfully ported numerous codes to various reconfigurable supercomputing platforms, including molecular dynamics,[3] linear algebra solvers,[4] and bioinformatics (http://sourceforge.net/projects/mitc-openbio), to name a few.

Today, reconfigurable computing technology is easier to use than you might think—at least for problems that map well to a particular HPRC architecture. In this introductory article, we demonstrate the steps necessary to implement a simple computational kernel on an SRC-6 MAP Series E reconfigurable processor. In a future article, we'll present an actual scientific computational kernel and show how we achieved a substantial speedup on the SRC-6 platform.

The SRC-6 Reconfigurable Computer

The SRC-6 MAPstation used in this work consists of a standard dual 2.8-GHz Intel Xeon motherboard and a MAP Series E processor interconnected with a 1.4-Gbyte/second low-latency four-port Hi-Bar switch (see Figure 1). The SNAP Series B interface board connects the CPU board to the Hi-Bar switch; SNAP plugs directly into a CPU board's dual in-line memory module (DIMM) slot.

The MAP Series E processor module contains two user-controlled FPGAs, one control FPGA, and the associated memory. For the six banks (A through F) of on-board memory (OBM), each bank is 64 bits wide and 4 Mbytes deep, for a total of 24 Mbytes. The programmer is explicitly responsible for application data transfer to and from these memory banks via vendor-supplied programming macros invoked from within the FPGA application. An additional 4 Mbytes of dual-ported memory is dedicated to data transfer between the two FPGAs, but this memory can also be used as two additional OBM banks (G and H).

The two user FPGAs in the MAP Series E are Xilinx Virtex-II Pro XC2VP100 FPGAs; each contains 88,192

 Copublished by the IEEE CS and the AIP 1521-9615/07/$25.00 ©2007 IEEE

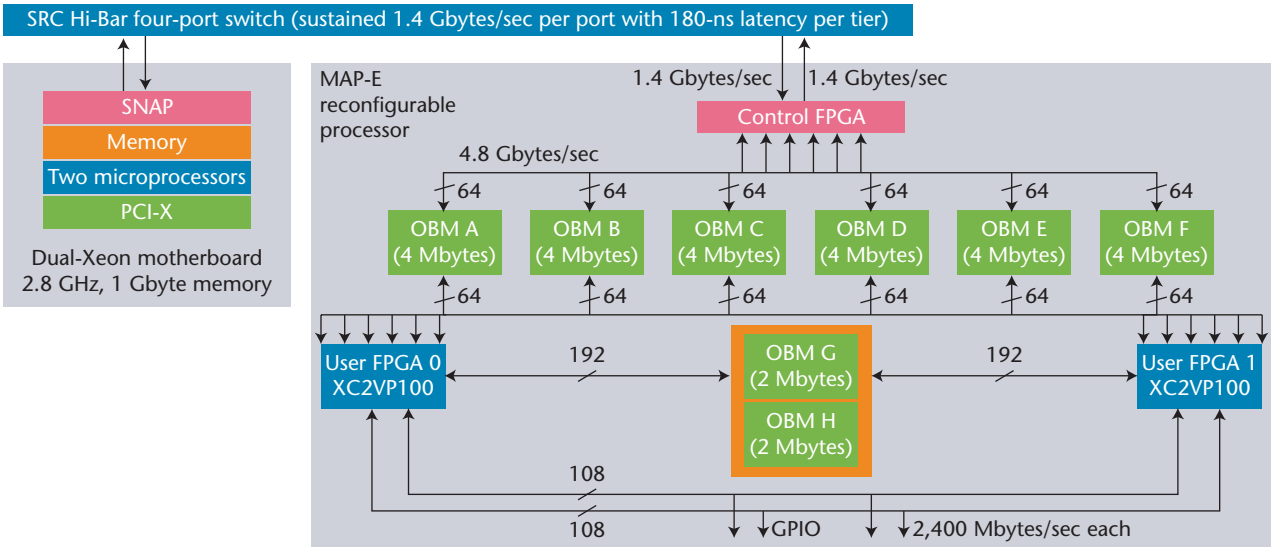


Figure 1. SRC-6 MAPstation architecture. The MAP Series E processor connects to a commodity dual-Xeon motherboard via the Hi-Bar switch.

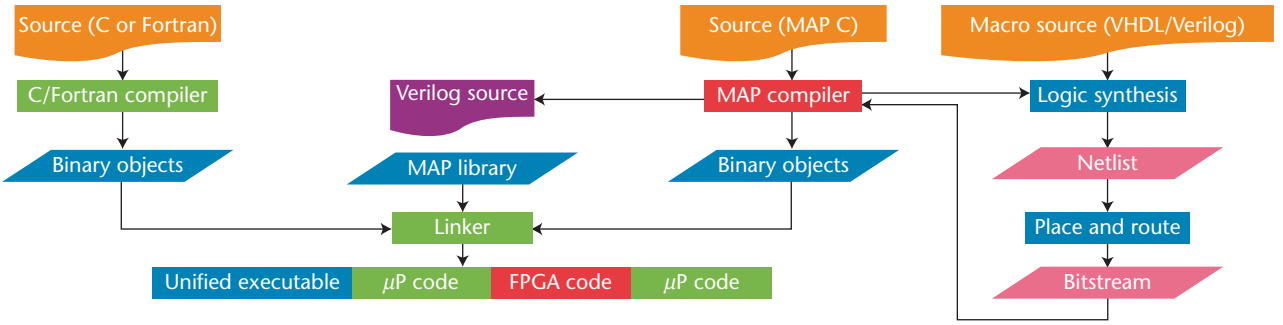


Figure 2. Outline of the SRC Carte code compilation flow. The CPU side of the algorithm is compiled with the Intel icc compiler whereas the FPGA side is compiled to a hardware description language via the MAP C compiler and converted into an FPGA configuration bitstream using Xilinx ISE.

four-input lookup tables, 88,192 flip flops, 444 dedicated 18 × 18 integer multipliers, and 999 Kbytes of internal dual-ported block RAM. These FPGA elements aren't directly visible to the programmer, but are interconnected via the programmer's MAP C algorithm code, the SRC Carte programming environment tools, and the Xilinx FPGA place and route tools. The SRC programming environment enforces the static FPGA clock rate of 100 MHz.

As Figure 2 shows, the Carte programming environment for the SRC-6 MAPstation is highly integrated: a single `makefile` can generate all the compilation targets. The two main compilation targets are a software-emulated debug version that verifies functional correctness and a final version that contains the embedded circuit bitmap that runs on the actual hardware FPGA. The Intel icc compiler generates both the CPU-only debug executable and the CPU side of the combined CPU/MAP executable. `make-file` invokes the SRC MAP compiler to produce the FPGA design's hardware description for the final combined CPU/MAP target executable and passes this intermediate hardware description to the Xilinx Integrated Synthesis Environment's (ISE's) place and route tools, which produce the FPGA bit file. Finally, the linker is invoked to combine the CPU code and the FPGA hardware bit files into a unified executable. Fortunately, `make debug` and `make hw` are the only two build commands the programmer has to learn; they hide the underlying complexity.

First Program

Programming an SRC-6 reconfigurable computer is similar to programming any other HPC system: achieving the highest performance possible requires exploiting abun-

	/*main_cpu.c*/	/*main_map.c*/
01	/*main_cpu.c*/	/*main_map.c*/
02	#include <stdlib.h>	#include <stdlib.h>
03		#include <libmap.h>
04		
05	#define SZ 1048576	#define SZ 1048576
06		
07	void ratval5(double *X, double *R, int sz);	voidratval5(double X[], double R[], int sz, int mapnum);
08		
09	int main (int argc, char *argv[])	int main (int argc, char *argv[])
10	{	{
11		int nummap=0;
12		
13	double *X=(double *)malloc(SZ* sizeof(double));	double *X=(double *)Cache_Aligned_Allocate(SZ * sizeof(double));
14	double *R=(double *)malloc(SZ* sizeof(double));	double *R=(double *)Cache_Algined_Allocate(SZ * sizeof(double));
15		
16	for (int i = 0; i < SZ; i++)	for (int i = 0; i < SZ; i++)
17	X[i]=(float)rand()/rand();	X[i]=(float)rand()/rand();
18		
19		map_allocate(1);
20		ratval5(X, R, 0, nummap);
21		
22	unsigned long long start, stop;	unsigned long long start, stop;
23	_asm_ _volatile_("rdtsc":"=A"(start):);	_asm_ _volatile_("rdtsc":"=A"(start):));
24		
25	ratval5(X, R, SZ);	ratval5(X, R, SZ, nummap);
26		
27	_asm_ _volatile_("rdtsc":"=A"(stop):);	_asm_ _volatile_("rdtsc":"=A"(stop):);
28	unsigned long long t = stop – start;	unsigned long long t = stop – start;
29		
30		map_free(1);
31		
32	printf("CPU ticks %lli, time %f seconds\n",	printf("CPU ticks %lli, time %f seconds\n",
33	t, (float)t/2800733000);	t, (float)t/2800733000);
34		
35	free(X);	Cache_Aligned_Free((char*)X);
36	free(R);	Cache_Aligned_Free((char*)R);
37	}	}

Figure 3. Microprocessor implementation. The left column shows the original main.c subroutine, and the right column shows main.c modified to use the SRC-6 MAP reconfigurable processor.

dant resources while working around any limited ones. As a specific example, consider a simple application that evaluates the following rational function for a million values of x:

$$R(x) = \frac{P(x)}{Q(x)} = \frac{p_0 + p_1x + p_2x^2 + p_3x^3 + p_4x^4 + p_5x^5}{q_0 + q_1x + q_2x^2 + q_3x^3 + q_4x^4 + q_5x^5}.$$

Reference C Implementation

The left columns of Figures 3 and 4 show a reference microprocessor implementation of the entire application. The compute kernel subroutine, `ratval5`, is called from the main subroutine (line #25 in Figure 3); accepts a pointer to an array, **X**, containing double-precision floating-point values of x for which we evaluate the rational function, a pointer to the array for storing the results, **R**, and the size of the input and output arrays, sz. The subroutine iterates through the array of x values, computes the $P(x)$ and $Q(x)$ polynomials using Horner's rule,[5] divides $P(x)$ by $Q(x)$, and stores the results in the output array.

We use the RDTSC (read time stamp counter) microprocessor instruction to count the number of CPU clock cycles necessary to execute the kernel subroutine (lines 23 and 27 in Figure 3). We then divide the number of clock cycles by the CPU frequency (2,800,733,000 Hz happened to be the frequency of the microprocessor

used in our system) to calculate the total execution time (line #33).

MAP C Implementation

Only a few lines of additional code need to be added to the main C routine to enable the reconfigurable processor's use. First, we include the `libmap.h` header file (line #3) to provide the reconfigurable MAP processor library function prototypes. We then change the kernel subroutine's function declaration to comply with the MAP C language (line #7) and add an extra variable, `nummap` (line #11). Before the application can use the MAP reconfigurable processor, it must be allocated via the `map_allocate()` library function call (line #19). At this point, we can also instruct the system to configure the FPGA with an appropriate bitstream by calling the FPGA subroutine once with dummy data (line #20). Although this step isn't strictly necessary and requires merely one call, we chose to implement it this way to decouple the FPGA bitstream configuration from the actual calculations. This simplifies the timing analysis—the FPGA configuration is done only once per kernel used and generally takes roughly 100 milliseconds.

The actual call to the FPGA-based subroutine (line #25) closely resembles the original microprocessor implementation. The Carte preprocessor and compiler will automatically generate the wrapper code to replace the call to this subroutine with a call to its FPGA-based implementation (without the developer's involvement). When the algorithm is executed, we release the MAP processor back to the system via the `map_free` function call (line #30).

Note that we use both a different memory allocation subroutine, `Cache_Aligned_Allocate` instead of `malloc` (lines #13 and 14), and also a different memory deallocation subroutine, `Cache_Aligned_Free` instead of `free` (lines #35 and 36). These subroutines are part of the Carte API, and they ensure memory alignment at 64-bit word boundaries, which is important for Direct Memory Access (DMA) transfers of data from main RAM to the MAP processor's OBM.

The MAP C implementation of the computational kernel subroutine (right column of Figure 4) is more complex than the original code (left column). They're algorithmically the same, but several additions are needed to transfer data in and out of the MAP processor. In essence, the programmer is responsible for cache memory management—something taken for granted on a conventional microprocessor architecture. This process has several required steps. First, we must declare which memory banks will be used (lines #6 through 9 in Figure 4, right column). Next, we issue instructions for the controller FPGA to transfer data from the system memory to the corresponding OBM banks on the MAP processor board (line #22) and wait until these transfers are completed (line #23). Each OBM bank is capable of holding 4 Mbytes of data, but because we'll process 8 Mbytes at once, we instruct the DMA engine to distribute the data across two OBM banks (line #22), modify the loop array access instruction to fetch the data from an appropriate bank (line #27), and modify the result store instruction (line #31). The calculation code looks like standard C (lines #28 through 30), it just uses data stored in two OBM banks and stores the results in two other banks. Once the calculations are done, we instruct the controller FPGA to transfer the results from the OBM banks to the system memory (lines #34 and 35).

Compiling the SRC-6 MAP C code is somewhat different from compiling a regular microprocessor code, although all we have to do is type `make hw` and wait patiently. For this example algorithm, the entire synthesis, place, and route process for our FPGA design takes more than 30 minutes. In general, though, this process takes even longer because an implementation that uses a substantial fraction of the resources in an FPGA typically takes many hours to place and route. Another result of the compilation process is the production of extensive diagnostic output by the compiler and `makefile` suite, which we now analyze.

Performance Analysis

One of the Carte compiling environment's useful outputs is the loop pipelining analysis report:

```
###########################################
############ INNER LOOP SUMMARY #############
loop on line 25:
clocks per iteration: 1
pipeline depth: 170
###########################################
```

This report tells us that the compiler encountered a loop in line 25 and successfully pipelined it. After the initial 170 clock cycles (the pipeline's depth), each iteration of the loop will bring in a new set of values and produce a new result. With this information, we can calculate the loop's overall execution time:

```
01 /*subroutine.c*/                                  /*subroutine.mc*/
02                                                   #include <libmap.h>
03
04 void ratval5(double *X, double *R, int sz)       void ratval5(double X[], double R[], int sz, int mapnum)
05 {                                                 {
06                                                    OBM_BANK_A(AL, double, MAX_OBM_SIZE)
07                                                    OBM_BANK_B(BL, double, MAX_OBM_SIZE)
08                                                    OBM_BANK_C(CL, double, MAX_OBM_SIZE)
09                                                    OBM_BANK_D(DL, double, MAX_OBM_SIZE)
10
11   float p0=0.434f;      float q0=0.595f;          float p0=0.434f;      float q0=0.595f;
12   float p1=-0.3434f;    float q1=0.34152f;        float p1=-0.3434f;    float q1=0.34152f;
13   float p2=3.4545f;     float q2=-1.4653f;        float p2=3.4545f;     float q2=-1.4653f;
14   float p3=-0.0045f;    float q3=3.2323f;         float p3=-0.0045f;    float q3=3.2323f;
15   float p4=-22.344f;    float q4=0.67578f          float p4=-22.344f;    float q4=0.67578f;
16   float p5=-0.4542f;    float q5=0.112f;          float p5=-0.4542f;    float q5=0.112f;
17
18   int i;                                          int i;
19
20                                                   if (!sz) return;
21
22                                                   DMA_CPU(CM2OBM,AL,MAP_OBM_stripe(1,"A,B"),X,1,sz*8,0);
23                                                   wait_DMA(0);
24
25   for(i=0; i<sz;i++)                              for(i=0; i<sz;i++)
26   {                                               {
27       const double x=X[i];                            const double x=(i%2==0)?AL[i/2]:BL[i/2];
28       double P=p0+x*(p1+x*(p2+x*(p3+x*(p4+x*p5))));   double P=p0+x*(p1+x*(p2+x*(p3+x*(p4+x*p5))));
29       double Q=q0+x*(q1+x*(q2+x*(x3+x*(x4+x*q5))));   double Q=q0+x*(q1+x*(q2+x*(q3+x*(q4+x*q5))));
30       R[i]=P/Q;                                       double val=P/Q;
31                                                       if(i%2==0)CL[i/2]=val; else DL[i/2]=val;
32   }                                               }
33
34                                                   DMA_CPU(OBM2CM,CL,MAP_OBM_stripe(1,"C,D"),R,1,sz*8,0);
35                                                   wait_DMA(0);
36 }                                                 }
```

Figure 4. Kernel subroutine. Compare the left column (the original `subroutine.c`) with the right column (`subroutine.mc`, implemented in the MAP C programming language targeting the SRC-6 MAP E reconfigurable processor).

$$T_{compute} = \frac{N + D + S_{loop}}{F} = \frac{1,048,576 + 170 + 14}{100,000,000 s^{-1}} = 0.0104876 s,$$

where N is the number of loop iterations, D is the pipeline depth, S_{loop} is the loop startup time expressed in the number of clock cycles, and F is the operational frequency of the FPGA.

We can also calculate the transfer time for moving data between the main memory and OBM storage. DMA instructions, lines #22 and #34 in Figure 4's code, force the DMA engine to operate at the full bandwidth—that is, after some initial DMA engine startup time, two 64-bit-wide words are transferred between the main memory and the OBM banks on each FPGA clock cycle. Thus, it takes

$$T_{din} = T_{dout} = \frac{M + S_{DMA}}{F} = \frac{1,048,576 / 2 + 5,300}{100,000,000 s^{-1}} = 0.00529588 s$$

to transfer the entire data set in or out of the OBM, where M is the number of 128-bit words transferred, and S_{DMA} is the DMA engine startup time measured in the number of FPGA clock cycles. On average, we find that S_{DMA} = 5,300 clock cycles.

The overall execution time of our algorithm running on the MAP processor is the sum of four components: $T_{compute}$,

T_{din}, T_{dout}, and $T_{startup}$—the time it takes to pass control from the CPU to the MAP processor. We measured that on average $T_{startup}$ = 0.00013 seconds. Thus, we estimate that our algorithm's overall execution time on the MAP processor will be 0.02120936 seconds.

Resources Utilization

Another useful piece of information the compiler produces is the place and route summary:

```
################################################
########## PLACE AND ROUTE SUMMARY ##########
No. of Slice Flip Flops: 29,807 out of 88,192 33%
No. of 4 input LUTs: 25,507 out of 88,192 28%
No. of occupied Slices: 22,621 out of 44,096 51%
No. of Block RAMs: 1 out of 444 1%
No. of MULT18X18s: 66 out of 444 14%
freq = 100.7 MHz
################################################
```

This reports the level to which our program uses various FPGA resources. The place and route summary tells us that our design uses 22,621 out of 44,096 slices (a slice is a small set of basic building blocks used as a basic unit area when determining an FPGA-based design's size), 66 out of 444 hardware multipliers, and one out of 444 block RAMs—three distinct types of logic "islands" in the "sea" of connectors. The report tells us that the design is capable of running at up to 100.7 MHz.

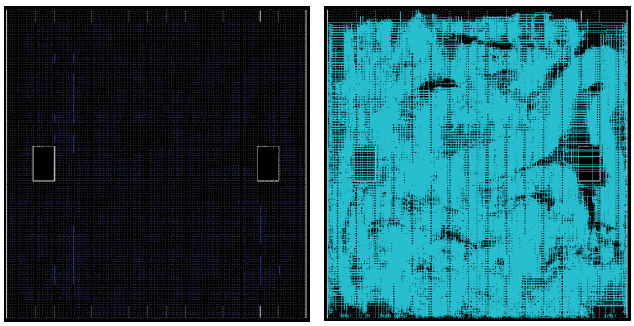
With the help of the FPGA editor (one of the many tools included in Xilinx ISE), we can also visualize the FPGA resources that the design uses. Figure 5a shows the placement of the components used in the design, and Figure 5b shows the connections made between them, with the route superimposed on a rectangular grid of switches. Although extensive design analysis and editing capabilities exist in Xilinx ISE, their use generally requires a deep knowledge of the hardware design principles—something that computational science application developers needn't be concerned with while using high-level programming languages such as MAP C.

The place and route summary specifies that our design uses less than half the FPGA. As a result, we can consider implementing multiple parallel computations to fill in the remaining space and speed up the calculations, a subject we'll return to later.

Runtime Performance

We compile our reference C implementation in version 8.1 of the Intel C compiler, with the architecture-specific processor optimizations turned on (-O3 -tpp7 –xW). When we execute our application's microprocessor implementation, the number of CPU clock cycles it takes to execute the kernel varies from one run to another. The differences can be as much as a few hundred thousand clock cycles—the CPU executes other tasks while we perform our tests—but, on average, the reference implementation takes 66,958,113 clock cycles on a 2.8-GHz CPU, for an elapsed mean time of 0.024 seconds.

When we execute the MAP-based algorithm, it takes on average 64,732,678 CPU clock cycles (0.023 seconds) to give control to the MAP and wait for the calculations to finish until control returns to the CPU. This time is close to the predicted performance (based on the pipeline and DMA transfer analysis) of 0.021 seconds.



(a) (b)

Figure 5. Component design. For the right column of Figure 4, we can visualize (a) the components and (b) the FPGA design's routes and connections.

MAP Code Optimization

Although our algorithm's MAP-based implementation runs and produces correct results, it doesn't really outperform the reference C implementation. Can we improve the overall performance at all? Have we learned anything while writing this first implementation that we can use to improve performance?

The first step in addressing these questions is to understand how well we used the system resources, such as I/O bandwidth, memory, and FPGA logic. Let's look at how we can apply unused resources to speed up the calculations. We can also examine implementation or algorithm modifications that might lead to better performance.

System Resource Usage

As we noted earlier, DMA instructions (lines #22 and #34 in Figure 4) force the DMA engine to operate at full bandwidth. Therefore, we've reached the system's data transfer bandwidth limit (main RAM to FPGA OBM).

But as we noticed earlier, this algorithm implementation uses less than half the available space on the FPGA;

thus, we can implement an additional rational function computation that can execute in parallel with the one already implemented. This is easy to achieve using the concept of parallel sections, which the MAP C language supports. The approach will work particularly well because the input (and output) data is stored in two separate OBM banks, so we can have noninterfering access to the data needed for executing two sets of simultaneous calculations. (This is a very important architecture consideration. In the MAP processor, a critical constraint is the number of simultaneous accesses to operands from local storage.) Once this approach is implemented, we will have cut the compute loop's overall execution time in half:

$$T_{compute} = \frac{N/2 + D + S_{loop}}{F} = \frac{1{,}048{,}576/2 + 170 + 14}{100{,}000{,}000 s^{-1}} = 0.00524472 s.$$

Adding the data transfer times and the MAP subroutine startup time to the modified loop execution time, we arrive at a new execution time of 0.01596648 seconds, which is roughly 1.5 times faster than the C implementation's execution time. With this approach, however, we'll also hit the FPGA resource limit (our new design will use up all slices available on the chip) in addition to the data transfer bandwidth saturation.

Implementation Restructuring

So far, we've observed that there are sufficient FPGA resources to implement two calculations in parallel, with each calculation generating one 64-bit result (for a total of 128 bits of data per single FPGA clock cycle). We also know that the DMA engine transfers data at the same rate: 128 bits per FPGA clock cycle. Therefore, we can restructure the code to fully overlap the calculations with the data transfer, and thus effectively hide one operation behind another. This is easy to implement by using the concept of streams as supported in the MAP C language: results produced in one section of the code are consumed simultaneously in another section (with some minimal latency) without the need for explicit intermediate storage. Because $T_{dout} > T_{compute}$, this implementation's overall execution time will be the sum of the data transfer operations and the MAP subroutine startup time: 0.01072176 seconds, which is roughly 2.2 times faster than the reference C implementation. Figure 6 shows the final MAP subroutine that implements this and the previous code modifications.

This implementation's measured execution time is 0.0125 seconds, compared to the 0.0107 seconds predicted. Compared with the microprocessor-based implementation's 0.024 seconds, we've improved the computational time by a factor of 1.9. At this stage, however, we can't improve execution time any further because we've run into a fundamental hardware limitation: I/O bandwidth. We use the full bandwidth to transfer in the data required to perform the calculations, and we also use the full bandwidth to transfer out the results, while overlapping calculations with data output. Interestingly, we haven't exhausted all the MAP processor's FPGA resources—we aren't even using the second user FPGA.

Software development for an FPGA-based system is an iterative process. The first, and perhaps most obvious, step is to identify an appropriate computational kernel—not all algorithms are appropriate for HPRC architectures. The next step is to implement the selected kernel "as is" and analyze its performance. Depending on this step's outcome, we might opt to continue with the chosen algorithm and just work on improving its performance by removing bottlenecks, overlapping data transfer with computations, pipelining loops, and parallelizing the code. Alternatively, we might choose to use an entirely different algorithm that's more appropriate for a given problem or architecture. The two driving forces in making this decision are the selected algorithm's performance characteristics and the system resources necessary to support its implementation. Eventually, space, memory, or bandwidth limitations will prevent any additional performance improvements. With experience, developers can recognize these application performance characteristics before code implementation, thereby speeding up the overall process.

With what might seem like a considerable effort, the HPRC system we described in this article produced a 1.9x speedup. A natural question, therefore, is whether the effort justifies the reward? After all, several other articles state significant performance boosts of 10x, 100x, or even 1,000x by using FPGA-based systems. But in reality, not all algorithms or applications lend themselves to FPGA implementations that can easily outperform fast microprocessors. Other factors that work against the FPGA are a limited I/O bandwidth between the main system memory

```
01  void ratval5(double X[], double R[], int sz, int mapnum)
02  {
03      OBM_BANK_A (AL, double, MAX_OBM_SIZE)
04      OBM_BANK_B (BL, double, MAX_OBM_SIZE)
05      OBM_BANK_C (CL, double, MAX_OBM_SIZE)
06      OBM_BANK_D (DL, double, MAX_OBM_SIZE)
07
08      Stream_64 S0, S1;
09
10      float p0=0.434f;                float q0=0.595f;
11      float p1=-0.3434f;              float q1=0.34152f;
12      float p2=3.4545f;               float q2=-1.4653f;
13      float p3=-0.0045f;              float q3=3.2323f;
14      float p4=-22.344f;              float q4=0.67578f;
15      float p5=-0.4542f;              float q5=0.112f;
16
17      if (!sz) return;
18
19      DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A,B"),
        X, 1, sz*sizeof(double), 0);
20      wait_DMA (0);
21
22     #pragma src parallel sections
23     {
24        #pragma src section
25        {
26        int i;
27        double x, P, Q;
28
29        for (i = 0; i < sz/2; i++)
30        {
31        x = AL[i];
32        P = p0+x*(p1+x*(p2+x*(p3+x*(p4+x*p5))));
33        Q = q0+x*(q1+x*(q2+x*(q3+x*(q4+x*q5))));
34        put_stream_dbl(&S0, P / Q, 1);
35        }
36      }
37
38     #pragma src section
39     {
40       int i;
41       double x, P, Q;
42
43       for (i = 0; i < sz/2; i++)
44       {
45         x = BL[i];
46         P = p0+x*(p1+x*(p2+x*(p3+x*(p4+x*p5))));
47          Q = q0+x*(q1+x*(q2+x*(q3+x*(q4+x*q5))));
49         put_stream_dbl(&S1, P / Q, 1);
50       }
51     }
52
53     #pragma src section
54     {
55        stream_dma_cpu_dual(&S0, &S1, STREAM_TO_
        PORT, CL, DMA_C_D, R, 1, sz*sizeof(double));
56     }
57    }
58  }
```

Figure 6. Final MAP C implementation of the computational kernel. In this implementation, two rational functions are computed at the same time, and the calculations overlap with the data transfer.

and the MAP processor and the need to copy data to OBM banks. In a future article, we'll use a different algorithm to demonstrate that we can achieve significant performance improvements for many computational kernels, in spite of these disadvantages.

References

1. M.B. Gokhale and P.S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer, 2005.
2. V. Kindratenko, "Code Partitioning for Reconfigurable High-Performance Computing: A Case Study," *Proc. Eng. Reconfigurable Systems and Algorithms* (ERSA 06), CSREA Press, 2006, pp. 143–149.
3. V. Kindratenko and D. Pointer, "A Case Study in Porting a Production Scientific Supercomputing Application to a Reconfigurable Computer," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 06), IEEE CS Press, 2006. pp. 13–22.
4. G. Morris and V. Prasanna, "Sparse Matrix Computations on Reconfigurable Hardware," *Computer*, vol. 40, no. 3, 2007, pp. 58–64.
5. W. Press et al., *Numerical Recipes in C: the Art of Scientific Computing*, 2nd ed., Cambridge Univ. Press, 1997, pp. 173–176.

Volodymyr V. Kindratenko is a senior research scientist at the National Center for Supercomputing Applications at the University of Illinois. His research interests include scientific, high-performance, and reconfigurable computing. Kindratenko has a D.Sc. in analytical chemistry from the University of Antwerp, Belgium. Contact him at kindr@ncsa.uiuc.edu.

Craig P. Steffen is a senior research scientist at the National Center for Supercomputing Applications at the University of Illinois. His research interests include programming unusual processor architectures, genome bioinformatics, and data archiving. Steffen has a PhD in high-energy physics from Indiana University. Contact him at csteffen@ncsa.uiuc.edu.

Robert J. Brunner is an assistant professor in the Department of Astronomy and a research scientist at the National Center for Supercomputing Applications at the University of Illinois. His research interests include black holes, cosmology, and high-performance computing. Brunner has a PhD in astrophysics from the Johns Hopkins University. Contact him at rb@astro.uiuc.edu.