Design Heuristics for Mapping Floating-Point Scientific Computational Kernels onto High Performance Reconfigurable Computers

Justin L. Rice, Khalid H. Abed, Gerald R. Morris
Department of Computer Engineering, Jackson State University, Jackson, MS, USA
Email: {justin.l.rice, khalid.abed, gerald.r.morris}@jsums.edu

***Abstract*—Because of the increasing need to develop efficient high-speed computational kernels, researchers have been looking at various acceleration technologies. One approach is to use field programmable gate arrays (FPGAs) in conjunction with general purpose processors to form what are known as high performance reconfigurable computers (HPRCs). HPRCs have already been shown to work well for both fixed-point and integer calculations. Floating-point calculations are a different matter; obtaining speedups has been somewhat elusive. This article, after introducing the three primary HPRC development flows, takes a detailed look at "the three p's," which addresses the crucial relationship among performance, pipelining, and parallelism. It also examines "the FPGA design boundary," which addresses some of the heuristics that allow developers to determine which application modules can be mapped onto the FPGAs. These ideas are illustrated by way of a simple floating-point application that is mapped onto a contemporary HPRC. This article expands upon earlier work by including details on how to map customized intellectual property cores into an HPRC environment via a hybrid development flow.**

***Index Terms*—high performance reconfigurable computer (HPRC), field programmable gate array (FPGA), algorithm mapping**

I. Introduction

The idea of a "fixed plus variable structure" reconfigurable computer (RC) has been around since 1960 and is attributed to Estrin [10]. However, the technological limitations of that era, such as bulky motherboards, manual wiring harness, etc., thwarted the development of the RC. In addition, most of the research during the following decades was focused on the general purpose processor (GPP). The revival of the RC was precipitated by Freeman's invention of the field programmable gate array (FPGA) in 1984 [36]. Less than a decade later, in 1991, Algotronix introduced the CHS2x4, which is considered to be the first commercially available RC [18]. In 1996, Seymour R. Cray's start-up company, SRC Computers, Inc., developed the SRC-6, which is arguably the first commercially successful high performance reconfigurable computer (HPRC) [29]. Commercial HPRCs from vendors such as SRC Computers, SGI, and Cray have ushered in a new era in the field of HPRC research.

This article, which is an expanded version of [28], describes "the three p's," which addresses the crucial relationship among performance, pipelining, and parallelism, and it examines "the FPGA design boundary," which considers heuristics that allow developers to determine which application modules can be mapped onto the FPGAs. This expanded version also includes details on how to map intellectual property (IP) cores into an HPRC development environment via the hybrid design flow.

The rest of this article is organized as follows: Section II is a brief look at related research. Section III describes the principal HPRC application development approaches. Section IV gives a simple example that illustrates why mapping floating-point applications onto HPRCs can be challenging. Section V, which is really the focus of the article, takes a look at various HPRC application design considerations. Section VI expands upon the earlier work by mapping IP cores into the HPRC development environment via the hybrid design approach. Section VII identifies potential future work and provides a conclusion.

II. Related Research

Researchers have had success in mapping some kernels to FPGAs, especially in the area of fixed-point and integer calculations. Taher et al. describe an implementation of a generic wavelet filter on the SRC-6 HPRC [32]. This filter facilitates the implementation of a wide range of discrete wavelet transform filters and outperforms conventional software by a factor of at least 10.5. El-Ghazawi et al. implement an HPRC version of an automatic wavelet-based dimension reduction algorithm for preprocessing of hyperspectral imagery, which garnered an order of magnitude speedup over software [9].

Buell et al. successfully port the Defense Advanced Research Project Agency Benchmark 5, which matches short bit strings against very long bitstreams, onto a contemporary HPRC. This was significant in that the benchmark itself measures the performance of high productivity computing systems, and the HPRC used in that research achieved maximum performance [4]. Using a simple application that computes the ratio of two fifth-degree polynomials, Kindratenko et al. provide an excellent tutorial on the use of the SRC-6 HPRC [20].

Catanzaro and Nelson show that using higher radices for floating-point representations can result in up to a 30 percent smaller area-time product while "delivering

equal worst-case and better average-case numerical accuracy" [5]. Wang et al. develop a library of variable-precision floating-point cores that allow a developer to use something other than the 32-bit or 64-bit precision afforded by GPPs [34]. Van Court and Herbordt recommend considering appropriate arithmetic precision, latency hiding, and using all of the FPGA chip resources (it does not make sense to throw away available computational capacity) [33].

For applications that require rigorous adherence to IEEE 754 floating-point arithmetic, Govindu et al. present a library of deeply pipelined, parameterizable floating-point cores that are able to achieve a frequency of up to 170 MHz on a Xilinx Virtex-II Pro [11]. During a panel session at Supercomputing 2006, El-Ghazawi et al. raise the question "Is high-performance reconfigurable computing the next supercomputing paradigm?" [8]. HPRCs have even entered into the embedded systems world. In April 2007, Lockheed Martin chose SRC Computers, Inc. to provide embedded HPRC systems for the U.S. Army's TRACER program [17].

Despite these advances, application development for HPRCs still poses a number of challenges, especially for floating-point scientific applications, where obtaining a speedup can sometimes be elusive. The application described in this article is one example, and there are others. In their research on HLL-HDL transformation, Böhm and Hammes did not achieve a speedup for a Gauss Seidel iterative solver because of the loop-carried dependence associated with floating-point cores [2]. They state that "this and other floating-point macros have not been integrated in the MAP compiler yet." Akella et al. did not speed up their floating-point sparse-matrix vector multiply (SMVM) kernel [1] because the floating-point accumulators in the Carte 2.1 compiler do not allow for fully pipelined inner-loop accumulation.[1] They conclude that "performance is still about 2-2.55 times slower than software."

Despite the complexity, some researchers have mapped floating-point algorithms onto HPRCs and shown promising results. DeLorimier et al. describe a scalable SMVM implementation on modern FPGAs and show that it can sustain high throughput and near-peak floating-point performance [7]. Morris and Prasanna achieve better than a twofold speedup for an IEEE 754 double-precision floating-point sparse matrix iterative solver and estimate that the same design on a next-generation HPRC could achieve a six-fold speedup [24]. Kindratenko et al. obtain a tenfold speedup for a double-precision floating-point implementation of a two-point correlation function [19]. Clearly, there are circumstances under which speedups can be obtained, even for floating-point applications.

III. Application Development Flows

An application targeted for an HPRC can be divided into two sets of modules: software (SW) modules and FPGA modules. SW modules, which are written in a traditional high-level language (HLL), are used to produce binary executables targeted for GPPs. Developers employ standard software development tools such as editors, compilers, debuggers, linkers, etc., to design, debug, and produce the executable. FPGA modules are used to create the configuration bitstreams that are loaded onto the FPGAs to produce the reconfigurable application-specific processors. These modules can be written using a standard hardware description language (HDL) such as Verilog [16] or VHDL [15], an enhanced HLL such as Carte C [29] or JHDL [3], or a combination of HDL and HLL. Developers employ FPGA development tools and, in the enhanced HLL-based cases, specialized HLL-HDL compilers to produce the FPGA configuration bitstream. Each design flow introduces certain challenges, which are described below. Development of software modules is not the focus of this article, so *only the development of FPGA modules is covered in the sections that follow*.

A. HDL Development Flow

The HDL-based flow is illustrated in Fig. 1. The FPGA module design entry is accomplished using a standard HDL. Vendor-specific IP cores are used to implement features such as GPP and memory interfaces. The design is submitted to the synthesis tool (SYNTH), which produces a netlist. The IP core and FPGA module netlists are submitted to the place and route (PAR) tool, which produces a target-specific circuit description. Finally, a bitstream is produced by the bit generation tool (BITGEN) and loaded onto the FPGA. The resulting application is executed in a cooperative manner by the GPP and FPGA. There has been little success using the HDL-based design flow to accelerate scientific applications. There is significant coupling between the software and FPGA modules, i.e.,

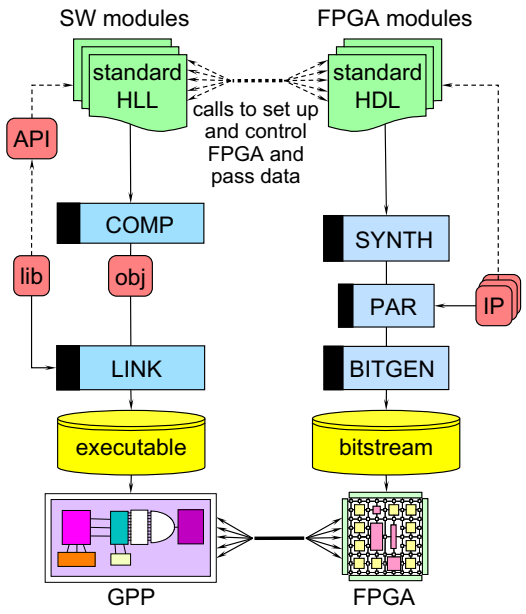

Fig. 1. HDL-Based Flow

[1] According to SRC Computers, this issue is addressed in the next generation SRC-7 system and the Carte 3.x compiler.

the software module must make application programmer interface (API) calls to access and control the FPGA module. There is also the complexity of hardware design and its impact on developer productivity. The net result is that this primitive development flow is not suitable for mainstream HPRC application development.

B. HLL Development Flow

Estrin acknowledges the need for "higher level languages for man-machine communication" in his seminal work on RCs [10]. Both private industry and the research community have responded to this need by developing enhanced HLLs to program HPRCs. In addition to Carte-

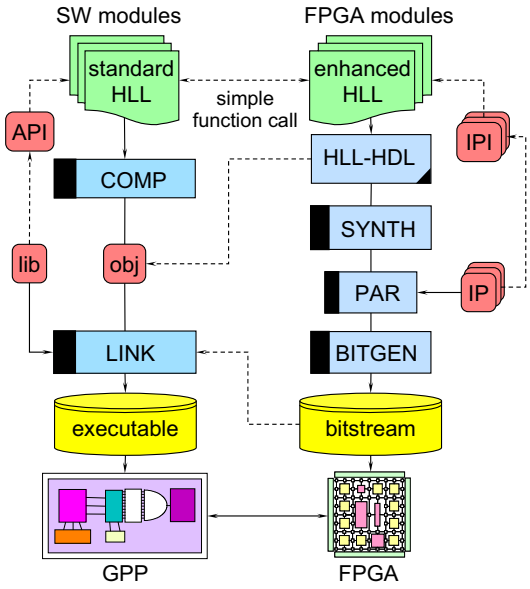

Fig. 2. HLL-Based Flow

C and JHDL, which were mentioned above, other popular HLL-based FPGA module development languages include Handel-C [6] and Mitrion-C [22]. These enhanced HLLs and the associated HLL-HDL compilers typically support pipelined loops, parallel code sections, synchronization primitives, communication channels (or streams), and other features that allow development of high performance FPGA kernels. As Fig. 2 indicates, the HLL-based flow starts with a design entry that is coded in an enhanced HLL. The intellectual property interface (IPI) allows HLL access to the vendor-supplied IP. The HLL is fed into an HLL-HDL compiler, which emits an HDL that is processed by the standard FPGA tool chain to produce the configuration bitstream. The HLL-based approach is certainly better than the HDL-based flow because there is minimal coupling between the software and FPGA modules, and the coding in an HLL allows for increased developer productivity. However, there are still challenges associated with this flow. The vendor IPI limits flexibility by restricting the developer to vendor-specific IP cores. Furthermore, this development flow can be somewhat misleading; even though it uses an HLL and looks like software, it is not software and should not be approached as such. Lastly, because of the deep pipelining needed to achieve high performance, it is often difficult to speed up floating-point computations.

C. Hybrid Development Flow

The hybrid-based development flow, shown in Fig. 3, is essentially the same as the HLL-based flow except it allows for an extensible IPI that accommodates vendor IP

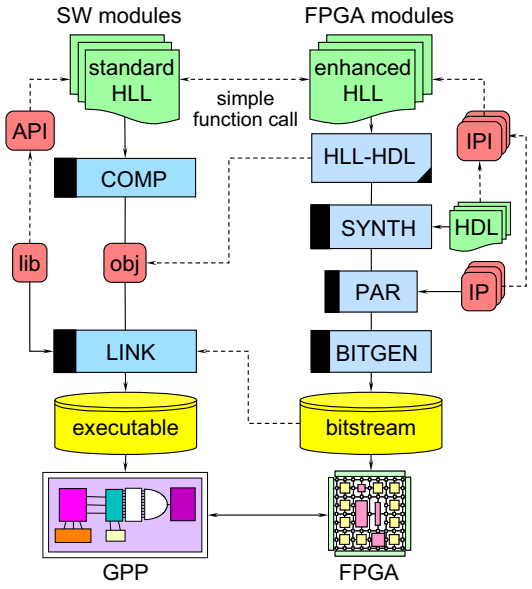

Fig. 3. Hybrid-Based Flow

cores, user-defined HDL designs, and even third-party IP cores. These features make this approach more flexible than the HLL-based flow and will help to bridge the gap between hardware developers and scientists. The hardware developers would be responsible for the development of a library of customized IP cores and their associated HLL interface; scientists would reference the cores via a simple software function call. The specifics of the hybrid approach vary depending upon the target HPRC and compiler; for example, the Carte compiler refers to this as "integrating user macros" [30], but the general concepts are the same. The challenges associated with this approach include all those associated with the HLL-based flow coupled with the need for both hardware design and integration of the HDL and IP cores. Section VI provides more details on this integration process for a modern HPRC development environment.

IV. Mapping Problem

Dividing work between the GPP and the FPGA is not a trivial task. Despite the advances in HLL-HDL compilers, the technology still remains far from a recompile and go approach. Taking existing software codes and compiling them on an HPRC via an HLL-based flow will not generally yield a speedup and may even result in a slowdown. For example, Park's attempt to accelerate the Blowfish algorithm resulted in a fortyfold slowdown when implemented using DIME-C [26]. Developing or porting

applications to HPRCs is still an art form, which relies heavily on the skill and experience of the developer. To illustrate the problem, the runtime of a software-only version of a simple quadratic equation solver is compared with the runtime of a naive HPRC implementation of the same algorithm.

A. Software-Only Version

The pseudo code for the software-only main routine is shown in Fig. 4, and the software-only version of the quadratic equation solver is idealized in Fig. 5. These

```
1: procedure MAIN(n)
2:     start ← now
3:     allocate(a_n, b_n, c_n, x_2n)
4:     load (n, a, b, c)
5:     SWQE(n, a, b, c, x)
6:     elapsed ← now − start
7:     outputResults(elapsed, n, x)
8: end procedure
```

Fig. 4. Software-Only Main Routine

routines were coded in C in a straightforward manner and, to obtain maximum performance, compiled at -O3 using the Intel 8.1 compiler on an SRC-6 HPRC. Note, that the software version only used the Xeon GPP on the SRC-6, not the FPGAs. In this pseudo code, n is the number of equations to be solved, vectors $\mathbf{a}_n, \mathbf{b}_n$, and $\mathbf{c}_n$ are the coefficients for each equation, and vector $\mathbf{x}_{2n}$, which is twice as long as the other vectors, holds the solution pair for each equation.

```
1: procedure SWQE(n, a_n, b_n, c_n, x_2n)
2:     for i in [0, n) do
3:         x_2i ← (−b_i + √(b_i^2 − 4a_i · c_i))/(2a_i)
4:         x_2i+1 ← (−b_i − √(b_i^2 − 4a_i · c_i))/(2a_i)
5:     end for
6: end procedure
```

Fig. 5. Software-Only Quadratic Equation Solver

B. Naive HPRC Version

The pseudo code for the HPRC main routine is shown in Fig. 6, and a naive FPGA-augmented version of the quadratic equation solver is shown in Fig. 7. The

```
1: procedure MAIN(n)
2:     start ← now
3:     cache_aligned_allocate(a_n, b_n, c_n, x_2n)
4:     load (n, a, b, c)
5:     allocate_FPGA(m)
6:     NaiveQE(n, a, b, c, x, m)
7:     elapsed ← now − start
8:     outputResults(elapsed, n, x)
9: end procedure
```

Fig. 6. HPRC Main Routine

FPGA module (MAP function in SRC parlance) was developed from the software code by adhering to the *minimal* requirements of the Carte compiler. The MAP number parameter, m, was added to the call interface. As suggested by lines 3-5, direct memory access (DMA) was used to load the coefficient vectors ($\mathbf{a}, \mathbf{b}, \mathbf{c}$) into on-board memory (OBM) arrays ($\mathbf{A}, \mathbf{B}, \mathbf{C}$). The loop was modified to read the coefficients from OBM arrays and to put the

```
1: procedure NAIVEQE(n, a_n, b_n, c_n, x_2n, m)
2:     declare obm(A_n, B_n, C_n, DE_n)
3:     dma(A, a, n)   // DMA values into OBM banks
4:     dma(B, b, n)
5:     dma(C, c, n)
6:     for i in [0, n) do
7:         D_i ← (−B_i + √(B_i^2 − 4A_i · C_i))/(2A_i)
8:         E_i ← (−B_i − √(B_i^2 − 4A_i · C_i))/(2A_i)
9:     end for
10:    dma(x, DE, 2n)   // DMA results back to GPP
11: end procedure
```

Fig. 7. Naive HPRC Quadratic Equation Solver

solutions (x_1, x_2) into a dual-column OBM array $\mathbf{DE}$. Finally, as suggested by line 10, a striped (two-column) DMA was used to send the results back to the $\mathbf{x}$ vector in the Xeon GPP memory. The MAP function code was compiled by Carte 2.1 with the default IEEE 754 SRC 64-bit floating-point macros and Xilinx tool chain settings, and the associated software modules were compiled at -O3 with the Intel 8.1 compiler.

C. Results

The main routines were instrumented with a μs-resolution timer to capture the wall clock runtime, and both versions were used to solve multiple sets of quadratic equations. A shell script executed each version 100 times for each of the test sizes in order to obtain average runtimes. Table I shows the results of this simple exper-

TABLE I
AVERAGE WALL CLOCK RUNTIME [MS]

n	t_S	σ_S	t_H	σ_H	*slowdown*
2	0.17	0.00	307.8	15.7	1782.8
16	0.18	0.00	304.0	6.6	1704.4
128	0.24	0.11	305.0	6.0	1248.3
1024	0.67	0.01	304.2	6.1	451.3
16384	8.63	0.27	303.1	5.5	35.1
262144	137.40	5.63	334.0	4.6	2.4
523776	258.02	0.77	369.9	4.2	1.4

iment. Column n is the number of quadratic equations solved in each run. The columns labeled t_S and t_H are the average wall clock runtimes for the SW and naive HPRC versions. Columns σ_S and σ_H are the standard deviations (respectively) for the 100 runs, and the column labeled *slowdown* shows how much slower the HPRC version ran compared with the software version. The HPRC version is significantly slower than software, even for the large test cases (the last case corresponds to the

largest set of 64-bit floating-point numbers that will fit in a single SRC-6 OBM bank). Results such as these may lead one to conclude that certain algorithms are not easily mapped or simply do not perform well when mapped onto reconfigurable hardware. The next section describes some application design considerations that may help when deciding how to map, or if one should map, algorithms onto HPRCs.

V. Design Considerations

This section takes a detailed look at "the three p's," which highlights the crucial relationship among performance, pipelining, and parallelism. It also examines "the FPGA design boundary," which addresses some of the heuristics that allow developers to determine which application modules can be mapped onto the FPGAs.

A. The Three P's

FPGA clock rates are in the 100s of MHz range, whereas GPP clock rates are on a GHz scale. Given this order-of-magnitude advantage on the part of GPPs, it is clear something must be done at the design level to compensate, i.e., that certain guidelines must be followed in order for an FPGA to compete with a GPP. As suggested by Fig. 8, the performance of an algorithm on an FPGA is proportional to the extent to which it is pipelined and parallelized. This multiplicative effect, which is known as the three p's, expresses the important relationship among performance, pipelining, and parallelism [23]. Specifically, both operations and datapaths must be pipelined and parallelized. In mapping an algorithm onto an HPRC,

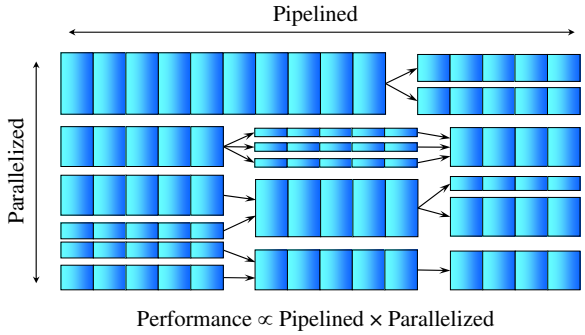

Fig. 8. The Three P's

failure to either pipeline or parallelize the kernel generally results in poor performance. Section IV included a naive HPRC implementation of the quadratic formula. That experiment showed that a naive HLL-based flow approach might not provide a speedup (in that particular case, there is actually a significant slowdown). The same algorithm will now be considered in light of the three p's in order to pipeline the implementation and extract available parallelism. Even though an HLL-based flow will eventually be used, an HDL-based flow will initially be assumed. Three steps are involved: (a) start with the algorithm and make appropriate component substitutions, (b) create a data flow graph (DFG) to uncover operation-level parallelism

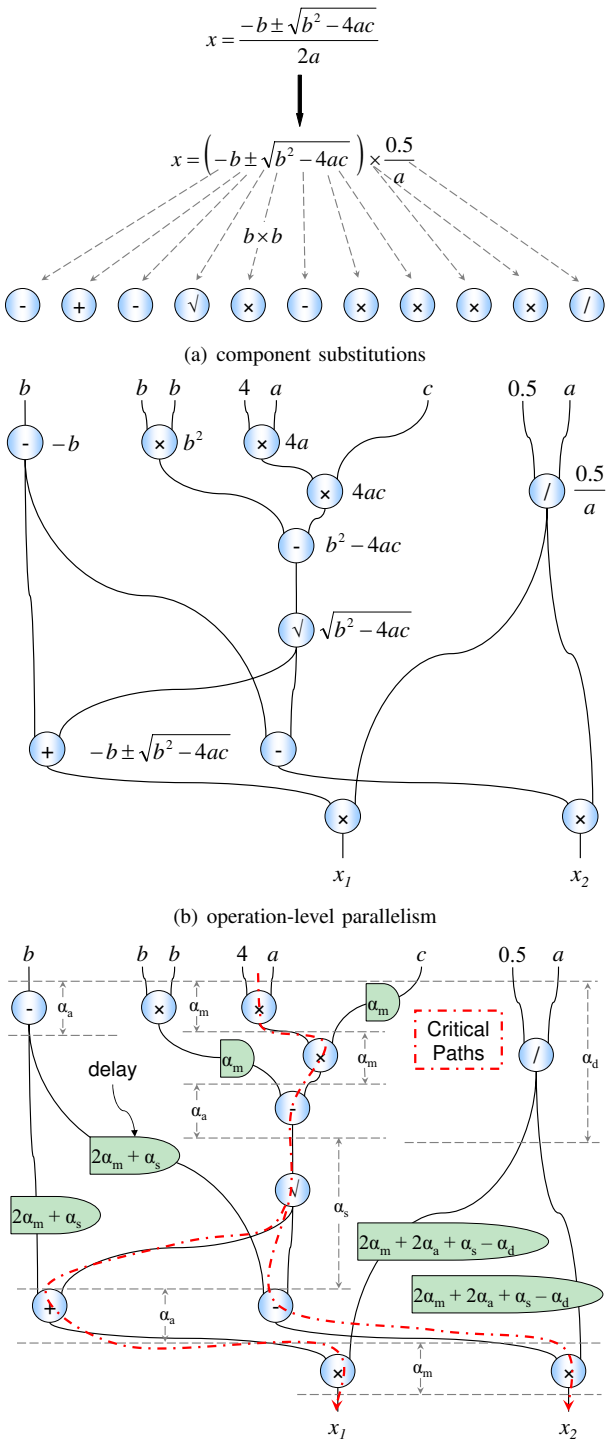

Fig. 9. Pipelining and Parallelizing

(OLP), and (c) equalize DFG path lengths to create a fully pipelined design. Some of these capabilities have been incorporated into HLL-HDL compilers, but the thought process for an HDL-based flow is still useful in an HLL-based environment.

First, it is imperative to start with the algorithm, not the code, as shown in Fig. 9(a). Component substitutions based on size, latency, clock rate, etc., are made. In the example, two dividers and one multiplier are replaced

with one divider and two multipliers, which completely hides the long latency of the output dividers. To avoid the use of a power function macro, b^2 is calculated as $b \times b$.

Second, as shown in Fig. 9(b), a DFG is created to account for true data dependencies that may occur and to uncover available parallelism. There is a four-way OLP at the input and a two-way OLP at the output of the quadratic equation DFG.

Third, in the pipelining portion of this approach shown in Fig. 9(c), each path length must be equalized in order to synchronize the data. Equalizing the path lengths begins by annotating each of the components with its latency, e.g., the divider has a latency of α_d. Then, the longest subpaths (critical paths) are identified. Finally, delay registers are added. The result is a fully pipelined and parallelized quadratic equation datapath, which has a latency of $\alpha_q = 3\alpha_m + 2\alpha_a + \alpha_s$.

```
1:  procedure StreamOutQE(n, a_n, b_n, c_n, x_2n, m)
2:      declare obm(A_n, B_n, C_n, DE_n)
3:      declare stream(S1, S2)
4:      dma(A, a, n)   // DMA values into OBM
5:      dma(B, b, n)
6:      dma(C, c, n)
7:      parbegin
8:        for i in [0, n) do   // parallel w DMA block
9:            a2 ← 0.5/A_i     // break into atomic ops
10:           bb ← B_i · B_i     // to be parallelized
11:           ac ← A_i · C_i     // and pipelined
12:           mb ← −B_i     // by HLL-HDL compiler
13:           ac4 ← 4 · ac
14:           D ← bb − ac4
15:           sqr ← √D
16:           bPsqr ← mb + sqr
17:           bMsqr ← mb − sqr
18:           put_stream(S1,bPsqr · a2)   // 2 streams to
19:           put_stream(S2,bMsqr · a2)   // dma block
20:       end for
21:       block
22:         stream_dma(x,S1,S2,2n)     // DMA to x
23:       endblock
24:     parend
25: end procedure
```

Fig. 10. "Stream Out" HPRC Quadratic Equation Solver

Ideally, several of these fully pipelined datapaths should be used in parallel to implement the FPGA module. In practice, the limited number of concurrent OBM bank accesses in the target HPRC precluded implementing more than one datapath. This violates the three p's, so (not too surprisingly) an actual speedup was not obtained. Nonetheless, to illustrate the concepts, these ideas will still be used to implement an improved solver. When using an HLL-based approach, one should break up monolithic algebraic expressions into the discrete assignments shown in the DFG, e.g. Fig. 9(b). The subsequent parallelization and pipelining can often be handled by the HLL-HDL compiler. In the following examples, based on the quadratic equation solver presented earlier, two different input/output (I/O) approaches are used: 1) streaming DMA for output, and 2) streaming DMA for input. The idea is to overlap communication with computation. Note that the target HPRC does not support simultaneous streaming DMA to and from the GPP.

Stream Out HPRC Version: In lines 4-6 of the pseudo code in Fig. 10, the FPGA module employs the same DMA input approach used in the naive implementation shown in Fig. 7. However, the output uses a dual-streaming DMA approach. The loop at line 8 and the output block at line 21 operate in parallel, as suggested by the **parbegin** construct at line 7. Additionally, loop computations are broken up into atomic operations to allow the HLL-HDL compiler to pipeline and parallelize the loop body. The bottom of the loop produces a pair of streams, S1 and S2, which are consumed by the output DMA on line 22. In the naive approach, the loop latency of the actual code was 169 clock cycles. In this approach, which exhibits a 4×2 OLP, the loop latency has dropped to 120 clock cycles because the HLL-HDL compiler finds more parallelism.

```
1:  procedure StreamInQE(n, a_n, b_n, c_n, x_2n, m)
2:      declare obm(DE_n)     // only need result array
3:      declare stream(SA, SB, SC)
4:      parbegin
5:        block
6:          stream_dma(SA, a, n)     // stream DMA
7:          stream_dma(SB, b, n)     // coefficients
8:          stream_dma(SC, c, n)     // via SA, SB, SC
9:        endblock
10:       for i in [0, n) do   // parallel w DMA block
11:           get_stream(A, SA)   // each loop has
12:           get_stream(B, SB)   // new A, B, C
13:           get_stream(C, SC)
14:           a2 ← 0.5/A     // break into atomic ops
15:           bb ← B · B     // as before
16:           ac ← A · C     // to assist
17:           mb ← −B     // HLL-HDL compiler
18:           ac4 ← 4 · ac
19:           D ← bb − ac4
20:           sqr ← √D
21:           bPsqr ← mb + sqr
22:           bMsqr ← mb − sqr
23:           D_i ← bPsqr · a2   // store result
24:           E_i ← bMsqr · a2   // in OBM banks
25:       end for
26:     parend
27:     dma(x, DE, 2n)   // DMA results back to GPP
28: end procedure
```

Fig. 11. "Stream In" HPRC Quadratic Equation Solver

Stream In HPRC Version: In line 27 of the pseudo code shown in Fig. 11, the FPGA module uses the same DMA output approach used in the naive implementation. However, the input uses three streaming DMAs, which are done in parallel, as suggested by lines 6-8. The loop

computations are done in parallel with the streaming input, so the loop appears to be operating with scalar values of **A**, **B**, and **C** during each iteration. In this approach, which also exhibits a 4×2 OLP, the loop latency of the actual code has dropped to 113 clock cycles.

Results: As with the naive case presented earlier, these two implementations were run multiple times on multiple data sets. Table II shows the results of the experiments. As expected, the single datapath violated the three p's, so there was not an overall speedup. Nonetheless, these

TABLE II
AVERAGE WALL CLOCK RUNTIME [MS]

n	*software*	*"Stream Out"*	*"Stream In"*
2	0.17	304.5	304.4
16	0.18	303.4	303.3
128	0.24	304.2	305.3
1024	0.67	303.2	304.1
16384	8.63	310.5	310.1
262144	137.40	334.8	333.7
523776	258.02	375.0	374.6

experiments are useful in that they illustrate a basic approach for designing a pipelined, parallelized datapath. In addition, they show that both datapath parallelism and OLP are needed. Of particular note is that the pipeline length dropped from 169 cycles in the naive case to 120 cycles and 113 cycles for the Stream Out and Stream In cases, respectively. The performance issue will be taken care of via the next generation of HPRCs, which have (among other improvements) a larger number of OBM accesses per clock cycle, i.e., the ability to have multiple datapaths executing in parallel.

B. FPGA Design Boundary

Determining the "FPGA design boundary," i.e., determining which application modules should be mapped onto FPGAs, is not straightforward [23]. As with many engineering disciplines, one must also rely on heuristics derived from empirical observation. Some areas to be considered when determining the FPGA design boundary are itemized below.

- The three p's
- Overall speedup (Amdahl's Law)
- Expected resource utilization
- Control/memory intensive vs. compute intensive
- Monolithic modules
- Available bandwidth
- Data reuse
- Algorithm design stability
- Algorithm efficiency ...

The three p's: Perhaps the most important heuristic is the three p's previously described. As shown in the experiments, if a module cannot be pipelined and parallelized, then it is unlikely to achieve high performance when mapped onto an FPGA. Even if a module is three p's compliant, it still needs to have enough data to keep the pipelines filled, i.e., to amortize pipeline latency across multiple problems. Thus, a corollary FPGA design consideration is to ensure the length of the data stream is sufficiently large.

Overall speedup (Amdahl's Law): The objective of mapping algorithms to HPRCs is to obtain a speedup relative to the performance of a GPP. Overall speedup can be quantified via Amdahl's Law [13]

$$s_o = \frac{1}{1 - f_e + f_e/s_e}$$

where s_o is the overall speedup, f_e is the fraction of the system to be enhanced, and s_e is the speedup of the portion to be enhanced. This particular law serves as a fundamental basis for design decisions. For example, suppose the estimated speedup for the edge detection portion of a target tracking system was a thousandfold, i.e., an FPGA implementation of the edge detection kernel was estimated to run an incredible 1000 times faster than an equivalent software module. Further suppose edge detection accounted for five percent of the runtime. With a thousandfold speedup, intuition says the FPGA-based edge detection kernel would yield a significant overall speedup. However, after applying Amdahl's Law,

$$s_o = \frac{1}{1 - 0.05 + 0.05/1000} = 1.05$$

it is seen that the overall speedup associated with the FPGA-based kernel is hardly worth the effort, and that in this case, fast is not always *fast*. This relationship between overall speedup and the fraction of a system that can take

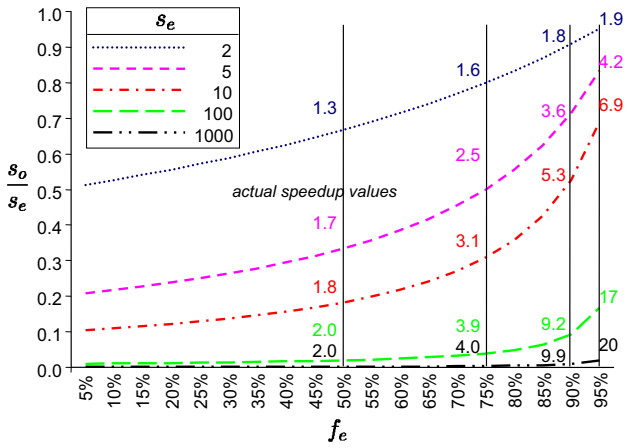

Fig. 12. Fast Is Not Always *Fast*

advantage of the speedup is depicted in Fig. 12. The x-axis represents the fraction of the system to be enhanced (f_e), and the y-axis is the overall speedup normalized to the speedup of the portion to be enhanced ($\frac{s_o}{s_e}$). The plotted lines represent the actual speedup values for five scenarios, $s_e \in (2, 5, 10, 100, 1000)$. The vertical lines represent the overall speedup values for $f_e \in (0.5, 0.75, 0.9, 0.95)$. Notice that for $f_e = 50$ percent, the overall speedup values are only 2.0 for $s_e = 100$ and $s_e = 1000$. The lowly $s_e = 2$ does nearly as well at 1.3. If 75 percent of the system could take advantage of a thousandfold speedup, the overall speedup value will only be 4.0. Even if a whopping 95 percent of the system could

use the thousandfold speedup, the overall speedup would only be 20. Thus, the overall speedup will be minimal, especially for large values of s_e, unless the module to be placed on the FPGA consumes a significant part of the overall runtime. Clearly, the use of Amdahl's Law in determining the FPGA design boundary is important.

Expected resource utilization: Another important FPGA design boundary consideration is the expected FPGA resource utilization of the candidate module. Since floating-point IP cores can be quite large, the developer needs to determine if the candidate will even fit on the FPGA. The developer also needs to consider the needed local memory capacity, number of simultaneous memory accesses, anticipated clock rate in the light of complex routing, etc.

Control/memory vs. compute intensive: It is also imperative to consider whether an algorithm is control/memory intensive or compute intensive. The control aspect is similar to the branching problem in a GPP, and the memory aspect is similar to a GPP where accessing memory data takes a considerable amount of time compared with arithmetic operations. Harkins et al. illustrate the importance of this concept when they show that sorting algorithms do not perform very well on an HPRC [12].

Monolithic module: Another design consideration is that hardware cannot call hardware. If the candidate FPGA module contains procedure calls, they have to be inlined or the module cannot be considered as a viable candidate. Obviously, this will be impacted by the available FPGA resources.

Available bandwidth: The GPP to FPGA bandwidth also deserves attention. Obviously, the FPGA memory access and processing time should be less than the GPP memory access and processing time. According to Herbordt et al., when they discuss latency hiding, a design should try to overlap computation with communication [14]. This might minimize the effects of bandwidth limitations. A closely related issue is data reuse, to be discussed next.

Data reuse: Algorithms that have a significant potential for data reuse may be suitable FPGA module candidates. Morris and Prasanna use this principle to speed up two well-known iterative solvers [25]. This is similar to methods used by the GPP where frequently used data are stored in nearby memory such as general-purpose registers or cache.

Algorithm design stability: Since mapping an algorithm to an FPGA is not the easiest of tasks, it is imperative to make sure that the algorithm is as stable as possible. If the algorithm is altered while in the midst of a hardware implementation process, one could easily discover that the new algorithm no longer fits onto the FPGA, or that it can no longer deliver on the promised speedup.

Algorithm efficiency: Another application design consideration is to make sure an efficient algorithm is employed. For example, Cramer's rule, which has exponential complexity, $O(e \cdot (n+1)!)$, might run faster if implemented on an FPGA. However, Gaussian elimination, with complexity, $O(n^3)$, is a much more efficient algorithm. In this case, one would use a software solution rather than map the inefficient algorithm onto an FPGA.

VI. Hybrid-Based Implementation

The strict HLL-based approach depicted in Fig. 2 can not be used in all circumstances. Perhaps the developer must use a proprietary IP core or maybe the HLL-HDL compiler cannot generate HDL that meets timing or resource constraints, etc. In such cases, the hybrid approach depicted in Fig. 3 might offer a suitable solution. This section describes how to map IP cores into an HPRC development environment.

A. Conceptual Overview

In the simplest case, as depicted in Fig. 13, the entire kernel has been implemented as an IP core, and the FPGA

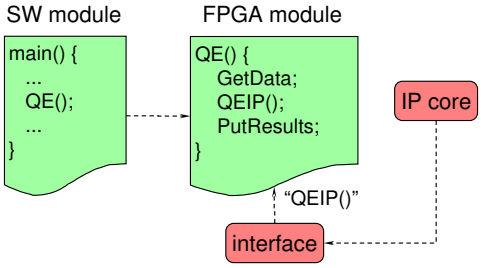

Fig. 13. IP-Only FPGA Module

module is essentially a call to the IP core. The interface is a "wrapper" that allows the IP core to appear as a software call. The FPGA module for the simple case might look something like the pseudo code shown in Fig. 14. The

```
1:  procedure WrappedQE(n, a_n, b_n, c_n, x_2n, m)
2:      interface QEIP(A, B, C, X1, X2)    // wrapper
3:      declare obm(DE_n)    // result array
4:      declare stream(SA, SB, SC)    // input streams
5:      parbegin
6:        block
7:          stream_dma(SA, a, n)    // stream DMA
8:          stream_dma(SB, b, n)    // coefficients
9:          stream_dma(SC, c, n)    // via SA, SB, SC
10:       endblock
11:       for i in [0, n) do    // parallel w DMA block
12:           get_stream(A, SA)    // each loop has
13:           get_stream(B, SB)    // new A, B, C
14:           get_stream(C, SC)
15:           QEIP(A, B, C, D_i, E_i)    // IP core solves it
16:       end for
17:     parend
18:     dma(x, DE, 2n)    // DMA results back to GPP
19: end procedure
```

Fig. 14. "Wrapped QE" HPRC Quadratic Equation Solver

key points are that all the computation is done in the IP core, and that access to the IP core looks like a software call because of the interface mechanism.

In the general case, as depicted by Fig. 15, only a part of the FPGA module is implemented via IP cores. The rest of the module is implemented using the enhanced HLL. In this latter example, the developer uses a divider IP core but implements the rest of the FPGA module in HLL. If one were to incorporate a divider IP core into the "Stream In" code described earlier, then the FPGA

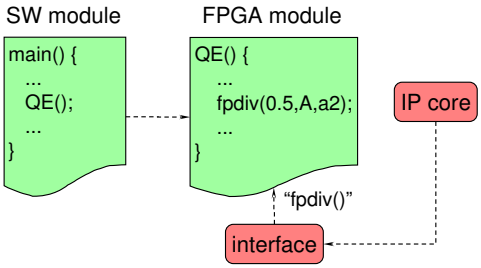

Fig. 15. IP-plus-HLL FPGA Module

module might look something like the pseudo code shown in Fig. 16. In the simple wrapper case, the IP core does the entire computation; in the general case, the IP only does part of the computation. A key point is that the scope

```
1: procedure DivQE(n, a_n, b_n, c_n, x_2n, m)
2:     interface fpdiv(x, y, z)    // wrapper for z = x/y
3:     declare obm(DE_n)    // only need result array
4:     declare stream(SA, SB, SC)
5:     parbegin
6:       block
7:         stream_dma(SA, a, n)    // stream DMA
8:         stream_dma(SB, b, n)    // coefficients
9:         stream_dma(SC, c, n)    // via SA, SB, SC
10:      endblock
11:      for i in [0, n) do    // parallel w DMA block
12:        get_stream(A, SA)    // each loop has
13:        get_stream(B, SB)    // new A, B, C
14:        get_stream(C, SC)
15:        fpdiv(0.5, A, a2)    // call fpdiv
16:        bb ← B · B    // everything
17:        ac ← A · C    // else same as
18:        mb ← −B    // before to
19:        ac4 ← 4 · ac    // allow
20:        D ← bb − ac4    // HLL-HDL compiler
21:        sqr ← √D    // to parallelize
22:        bPsqr ← mb + sqr    // and pipeline
23:        bMsqr ← mb − sqr    // the module
24:        D_i ← bPsqr · a2    // store result
25:        E_i ← bMsqr · a2    // in OBM banks
26:      end for
27:    parend
28:    dma(x, DE, 2n)    // DMA results back to GPP
29: end procedure
```

Fig. 16. "Wrapped Div" HPRC Quadratic Equation Solver

of the IP functionality is irrelevant. One must still create an interface to the IP core and then call it from within the HLL-based FPGA module.

B. IP Core Development

For this research effort, the authors created the IP core from VHDL source code. In the general case, this may not be true, i.e., a properly tested and documented off-the-shelf core could be used. The derivation of the divider IP is briefly summarized in this section: Using an offline engineering workstation (not the SRC-6 HPRC), a Xilinx ISE project [35] was built based upon the Rice-Govindu floating-point divider VHDL source code [27], [11]. The floating-point divider top-level interface, which is called an entity in VHDL, is idealized in Fig. 17. A complete

```
entity divIP is
  -- 58-stage pipelined IEEE floating-point divider
  port (
    x    :  in   std_logic_vector (63 downto 0);
    y    :  in   std_logic_vector (63 downto 0);
    clk  :  in   std_logic;
    z    :  out  std_logic_vector (63 downto 0)
  );
end entity divIP;
```

Fig. 17. Divider VHDL Interface

description of the entity is not needed; one must simply note that the two inputs, x and y, are 64-bit quantities, there is a clock, and there is a 64-bit output, z. The details of the implementation, which is called an architecture in VHDL, are also not needed. The important point is that the entity and its architecture are ultimately synthesized into a 58-clock cycle pipelined IEEE 64-bit floating-point divider IP core entitled divIP.edn. A VHDL test bench and set of input vectors were created so the floating-point divider code could be rigorously tested using the ModelSim VHDL simulation environment [21]. The divider VHDL code was subsequently synthesized using Synplify Pro [31], and the resulting electronic design interchange format (EDIF) netlist file (IP core) was then uploaded to the SRC-6 for integration into the HPRC development environment.

C. IP Core Interface

The discussion in this section centers around the Carte compiler and SRC-6 HPRC that were used in this research. If one were to use another compiler or target

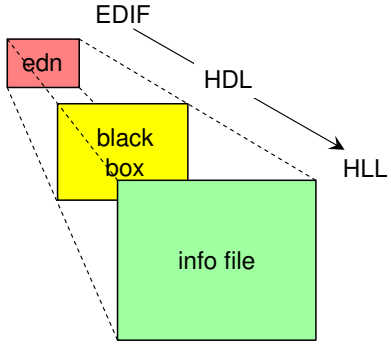

Fig. 18. EDIF to HDL to HLL

HPRC, the details would be different, but the concepts would be the same. The basic mapping process is hierarchical, as suggested by Fig. 18. It is, in some sense, a reverse engineering of the EDIF file back to HDL, and then to HLL. The starting point is the EDIF edn netlist file. While not quite accurate, one can think of the EDIF as being a nongraphical representation of the schematic diagram. A key point is that the hardware design encapsulated by the EDIF has a name and some set of input ports and output ports that 1) must be made visible to the FPGA HDL tool chain, and 2) must be mapped onto an HLL procedure.

The *blackbox*, as depicted in Fig. 19, is a Verilog module that describes the input and output ports and serves as an HDL interface description.

```
module divIP(x,y,clk,z);
      input    [63:0] x;
      input    [63:0] y;
      input    clk;
      output   [63:0] z;
endmodule
```

Fig. 19. Blackbox File

The *info file* is primarily used to map the IP core name onto an HLL procedure name and the IP port names onto HLL procedure parameter names. The info file also

```
BEGIN_DEF "divIP"
   MACRO = "fpdiv";
   EXTERNAL = NO;
   PIPELINED = YES;
   LATENCY = 58;
   INPUTS = 2:
      I0 = FLOAT 64 BITS (x[63:0])
      I1 = FLOAT 64 BITS (y[63:0])
      ;
   OUTPUTS = 1:
      O0 = FLOAT 64 BITS (z[63:0])
      ;
   IN_SIGNAL : 1 BITS "clk" = "CLOCK";

   DEBUG_HEADER = #
      void divIP__dbg(double x, double y, double *z);
   #;

   DEBUG_FUNC = #
      void divIP__dbg(double x, double y, double *z) {
         *z = x / y;
      }
   #;
END_DEF
```

Fig. 20. Info File

contains a "debug" implementation of the IP core. This is a software-only functional equivalent of the IP core. The Carte environment uses the debug code during early development to verify FPGA module functionality without having to go through the (usually time-consuming) FPGA tool chain, i.e., it executes a software-only functional equivalent of the FPGA module. A representation of the info file is shown in Fig. 20. The MACRO line maps the IP core name "divIP" onto a software procedure name "fpdiv." The information about divIP being a 58-stage pipeline helps the HLL-HDL compiler optimize loops. The input and output ports are mapped onto 64-bit floating point data type parameters, and the clock input signal is hidden from the caller. Remember, the idea is to make the IP core hardware design look like a software call, so the hardware notions of ports, bits, and clocks are abstracted away. Finally, the info file includes an implementation of the debug functionality.

D. Comparison

For comparison purposes, two versions of the quadratic equation solver were implemented: the software-only version shown in Fig. 5, and the hybrid version shown in Fig. 16. The pseudo code for the respective main routines are shown in Fig. 4 and Fig. 6. Note that the latter code calls DivQE rather than NaiveQE. As with the codes presented in Section V, these two implementations were run multiple times using multiple data sets. Table III shows the results of these experiments. As before, the single datapath violated the three p's, so there was not an overall speedup. However, as will be shown in the next section, this does not invalidate the research results.

TABLE III
AVERAGE WALL CLOCK RUNTIME [MS]

n	*software*	*"Wrapped Div"*
1024	0.69	303.7
16384	8.63	309.4
262144	137.6	334.5
523776	259.9	373.4

E. Discussion of Performance on Next-Generation HPRC

The Kiviat diagram in Fig. 21 compares the capability of the SRC-6 HPRC used in this research with the next-generation SRC-7 HPRC. The footprint of the SRC-7 system swamps the footprint of the SRC-6. With nearly an order-of-magnitude increase in available user logic, threefold increase in OBM bandwidth, 50 percent increase in clock speed, two orders-of-magnitude increase in local memory, more than a threefold increase in peak 64-bit GFLOPS, etc., it is obvious that this next generation HPRC will facilitate significant improvements in floating-point application performance. In the research described in this article, for example, it would be possible to put at least two quadratic equation datapaths onto the FPGAs. If one factors in the 1.5 clock speed improvement, this translates into a threefold performance boost. Add to this the ability to store much larger data sets, and it is clear that the performance deficiencies of the HPRC described in this article are a temporary problem that will be overcome via the next-generation technology. The objective of the

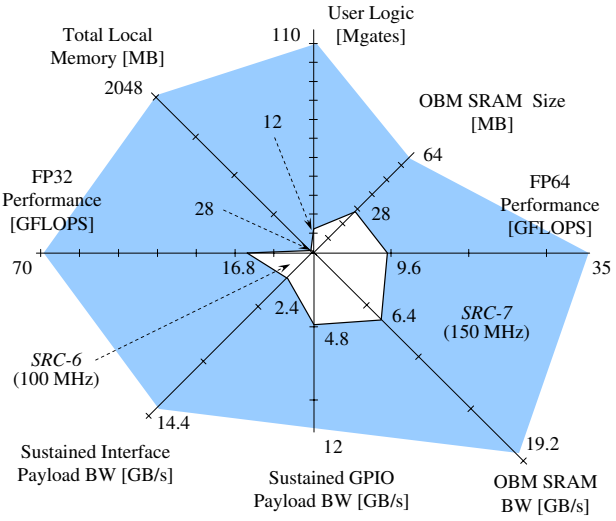

Fig. 21. Capabilities of SRC-6 vs. SRC-7

research was to show how one would port a computational kernel into an HPRC environment. That objective has been met.

VII. Future Work and Conclusion

A. Future Work

Floating-point computational kernels can sometimes be accelerated when mapped onto the FPGAs of modern HPRCs. Such mapping is currently a manual process that relies upon the skill and experience of the designer. This article summarizes some of the known design heuristics under one umbrella; as new rules of thumb are discovered, the authors plan to "add them to the list" as it were. The long-term goal is for tomorrow's compilers to automate these design heuristics and eliminate some of the trial-and-error associated with mapping scientific applications onto HPRCs and other heterogeneous architectures. Jackson State University has recently acquired a multinode SRC-7 cluster with an Infiniband interconnection network. The vendor has completed the initial installation of this state-of-the-art HPRC cluster and is nearly finished with the remaining field-level upgrades. As previously noted, the increased number of memory banks, clock rates, and the FPGA area of these new machines will allow for significant design improvements and a corresponding improvement in speedup. The authors plan to revisit some of their earlier research efforts and map computational kernels such as conjugate gradient, etc., onto these next-generation HPRCs. The authors also plan to look at a number of real-world parallel scientific codes and attempt to speed them up by mapping them onto the SRC-7 cluster. The goal, aside from speeding up the applications, is to further refine the design heuristics (what works and what does not work) relative to mapping floating-point applications onto HPRCs.

B. Conclusion

The overall goal of using an HPRC is to obtain better performance than can be achieved via traditional software. At the current time, obtaining this performance edge is somewhat elusive for floating-point applications. This article summarizes many of the heuristics that the authors and other researchers have uncovered. In particular, the article describes the three p's, the FPGA design boundary, and other factors developers must consider when mapping an application onto an HPRC system. A simple example was used to illustrate the concepts. The article included a section dealing with the mapping of customized IP cores into an HPRC development environment. The jury is still out on whether HPRCs will become part of mainstream high performance computing. However, given the positive results from various research efforts, the expected advancements in FPGA technology, the likelihood of additional compiler optimizations, and the documented improvements in next-generation HPRC performance, one can anticipate a larger role for HPRCs.

Acknowledgments

This work was supported in part by the U.S. Army Engineer Research and Development Center research contract W912HZ-06-C-0057, "High Performance Computational Design of Novel Materials," in part by the National Science Foundation award HRD-0602740, "Louis Stokes Alliances for Minority Participation," and in part by the Department of Defense High Performance Computing Modernization Program. Special thanks to Robert Moak for his help with some of the graphics.

References

[1] S. Akella, M. C. Smith, R. T. Mills, S. R. Alam, R. F. Barrett, and J. S. Vetter. Sparse matrix vector multiplication on the SRC MAPstation. In *Proceedings of the 9th Annual High Performance Embedded Computing Workshop*, Lexington, MA, September 2005.

[2] W. Böhm and J. Hammes. A transformational approach to high performance embedded computing. In *Proceedings of the 8th Annual High Performance Embedded Computing Workshop*, pages 39–40, Lexington, MA, September 2004.

[3] Brigham Young University. JHDL: FPGA CAD tools. www.jhdl.org.

[4] D. A. Buell, S. Akella, J. P. Davis, E. A. Michalski, and G. Quan. The DARPA boolean equation benchmark on a reconfigurable computer. In *Proceedings of the 7th Military and Aerospace Programmable Logic Devices Conference*, Washington, DC, September 2004.

[5] B. Catanzaro and B. Nelson. Higher radix floating-point representations for FPGA-based arithmetic. In *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 161–170, Napa, CA, April 2005.

[6] Celoxica Ltd. DK Design Suite. www.celoxica.com.

[7] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 75–85, Monterey, CA, 2005.

[8] T. El-Ghazawi, D. Bennett, D. Poznanovic, A. Cantle, K. Underwood, R. Pennington, D. Buell, A. George, and V. Kindratenko. Is high-performance reconfigurable computing the next supercomputing paradigm? In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 71–78, Tampa, FL, 2006.

[9] T. El-Ghazawi, E. El-Araby, A. Agarwal, J. LeMoigne, and K. Gaj. Wavelet spectral dimension reduction of hyperspectral imagery on a reconfigurable computer. In *Proceedings of the 7th Military and Aerospace Programmable Logic Devices Conference*, Washington, DC, September 2004.

[10] G. Estrin. Organization of computer systems–the fixed plus variable structure computer. In *Proceedings of the Western Joint Computer Conference*, pages 33–40, San Francisco, CA, May 1960.
[11] G. Govindu, R. Scrofano, and V. K. Prasanna. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms*, pages 137–148, Las Vegas, NV, June 2005.
[12] J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang. Performance of sorting algorithms on the SRC 6 reconfigurable computer. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, pages 295–296, Singapore, December 2005.
[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Francisco, CA, 3rd edition, 2003.
[14] M. C. Herbordt, T. V. Court, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *Computer*, 40(3):50–57, March 2007.
[15] Institute of Electrical and Electronics Engineers. *IEEE Std 1076-1993: IEEE Standard VHDL Language Reference Manual*. IEEE, New York, NY, 1994.
[16] Institute of Electrical and Electronics Engineers. *IEEE Std 1364-2001: IEEE Standard Verilog Hardware Description Language*. IEEE, New York, NY, 2001.
[17] V. Jackson. SRC Computers chosen by Lockheed Martin for U.S. Army program. In *SRC Computers, Inc. - News*, August 2007.
[18] T. Kean. Algotronix history. www.algotronix.com/people/tom/album.html, June 1998.
[19] V. V. Kindratenko, R. J. Brunner, and A. D. Myers. Mitrion-C application development on SGI Altix 350/RC100. In *Proceedings of the 15th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 23–25, Napa, CA, April 2007.
[20] V. V. Kindratenko, C. P. Steffen, and R. J. Brunner. Accelerating scientific applications with reconfigurable computing: Getting started. *Computing In Science and Engineering*, 9(5):70–77, September-October 2007.
[21] Mentor Graphics, Inc. Modelsim. www.model.com.
[22] Mitrionics Inc. Mitrion-C. www.mitrionics.com.
[23] G. R. Morris. *Mapping Sparse Matrix Scientific Applications onto FPGA-Augmented Reconfigurable Supercomputers*. Ph.D.E.E. dissertation, University of Southern California, December 2006.
[24] G. R. Morris and V. K. Prasanna. A hybrid approach for accelerating a sparse matrix Jacobi solver using an FPGA-augmented reconfigurable computer. In *Proceedings of the 9th Military and Aerospace Programmable Logic Devices Conference*, Washington, DC, September 2006.
[25] G. R. Morris and V. K. Prasanna. Sparse matrix computations on reconfigurable hardware. *Computer*, 40(3):58–64, March 2007.
[26] S. J. Park. Reconfigurable computing for HPC computational science. In *Proceedings of the 2007 HPCMP User Group Conference*, Pittsburgh, PA, June 2007.
[27] J. L. Rice. Hardware implementation of an IEEE 754 standard 64-bit floating-point division core on an SRC reconfigurable computer. M.S.C.E. thesis, Jackson State University, May 2008.
[28] J. L. Rice, K. C. Pace, M. D. Gates, G. R. Morris, and K. H. Abed. High performance reconfigurable computer application design considerations. In *Proceedings of the IEEE Southeast Conference 2008*, Huntsville, AL, April 2008.
[29] SRC Computers, Inc. General purpose reconfigurable computing systems. www.srccomp.com.
[30] SRC Computers, Inc. *SRC C Programming Environment v2.1 Guide*. SRC Computers, Inc., Colorado Springs, CO, 2005.
[31] Synplicity, Inc. Synplify Pro. www.synplicity.com/products/synplifypro.
[32] M. Taher, E. El-Araby, A. Agarwal, T. El-Ghazawi, K. Gaj, J. L. Moigne, and N. Alexandridis. Effective implementation of a generic wavelet filter on a hybrid reconfigurable computer. In *Proceedings of the 6th Military and Aerospace Programmable Logic Devices Conference*, Washington, DC, September 2003.
[33] T. Van Court and M. C. Herbordt. Requirements for any HPC/FPGA application development tool flow. In *Proceedings of the 4th Annual Boston Area Computer Architecture Workshop*, pages 55–59, Kingston, RI, February 2006.
[34] X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 137–148, Napa, CA, April 2006.
[35] Xilinx, Inc. Design tools center. www.xilinx.com/products/design_resources/design_tool.
[36] Xilinx, Inc. New technology and a new way of working. In *History of Xilinx*. www.xilinx.com/company/xilinxstory/history.htm, 2006.

AUTHOR BIOGRAPHIES

Justin L. Rice is a computer engineer enrolled in the Ph.D. program at Louisiana Tech University, Ruston, LA. His research interests include reconfigurable computing and field programmable gate arrays. Rice received his B.S. and M.S. in Computer Engineering (summa cum laude) from Jackson State University. Rice is a National Science Foundation Louis Stokes Alliances for Minority Participation Bridge to the Doctorate Fellow and a Student Member of the IEEE.

Khalid H. Abed is Assistant Professor of Computer Engineering at Jackson State University, Jackson, MS. His research interests include high performance reconfigurable computing, field programmable gate arrays, very large scale integrated circuit design, and digital signal processing. He received his B.S., M.S., and Ph.D. in Electrical Engineering from Wright State University. Abed has written more than 20 publications in IEEE journals and conferences and is a technical reviewer for several IEEE journals and conferences. He has received funding from sources such as the National Science Foundation, the Department of Defense, and the Army Research Office. Abed is a Senior Member of the IEEE.

Gerald R. Morris is a researcher at the Department of Defense High Performance Computing Modernization Program Major Shared Resource Center at the U.S. Army Engineer Research and Development Center, Vicksburg, MS. He is also Adjunct Professor of Computer Engineering at Jackson State University, Jackson, MS. His research interests include high-performance and reconfigurable computers. Morris received his B.S. in Electrical Engineering (summa cum laude) from the Ohio State University, M.S. in Computer Engineering from the Air Force Institute of Technology, and Ph.D. in Electrical Engineering from the University of Southern California. He has published extensively and is a Senior Member of the IEEE.