Mapping Floating-Point Kernels onto High Performance Reconfigurable Computers

Gerald R. Morris
U.S. Army Engineer Research and Development Center, Vicksburg, MS, USA
Email: gerald.r.morris@us.army.mil

Khalid H. Abed
Jackson State University/Computer Engineering Department, Jackson, MS, USA
Email: khalid.h.abed@jsums.edu

***Abstract*—Contemporary field programmable gate arrays (FPGAs) combine the fine-grained design capability of the traditional lookup table with the speed of medium-scale and large-scale logic components such as RAM blocks or DSP blocks to provide for significant computational capability from a single FPGA. High performance reconfigurable computers, which typically use FPGAs as computational elements, have been commercially used to accelerate computational kernels. However, the deep pipelines and extensive parallelism needed for FPGAs to compete with GHz-scale general purpose processors make mapping of floating-point kernels a challenging research area. In this paper, we describe some of the progress that has been made towards solving some of these mapping challenges.**

I. Introduction

Modern field programmable gate arrays (FPGAs) have gone beyond the traditional *lookup table (LUT) plus routing* model that characterized their early counterparts. FPGAs now incorporate medium-scale and large-scale logic components such as RAM blocks, DSP blocks, shift registers, on-chip clock controllers, and high-speed I/O blocks within the programmable FPGA interconnection fabric. This combination provides for the fine-grained, gate-level (albeit slow) design capability associated with LUTs, yet allows for the use of the much faster fixed components within a design. One of the results of this modern architecture is the ability to provide for significant computational capability from a single FPGA. High performance reconfigurable computers (HPRCs), which typically use FPGAs as computational elements, have been commercially used to accelerate kernels for both embedded and traditional applications [1], [2]. However, the deep pipelines and extensive parallelism needed for FPGAs (which run at a few hundred MHz) to compete with general purpose processors (which run at a few GHz) make mapping of floating-point kernels a challenging research area. In this article, we describe some of the progress that has been made towards solving some of these mapping challenges. The article is organized as follows. Section II provides background information on HPRCs including a discussion of the high-level language (HLL)-based development flow and the specific platform used in this research. Section III addresses some of the heuristics that allow developers to determine which kernels are good candidates for mapping onto an FPGA. Section IV gives a specific example of how to map hardware description language (HDL)-based components into an HLL-based development flow. Section V illustrates the use of these ideas to map a simple floating-point Jacobi iterative solver onto an HPRC, and Section VI presents the conclusions.

II. Background

A. High performance reconfigurable computers

The reconfigurable computer (RC) was introduced in the 1960s by Estrin [3] as a "fixed plus variable structure computer." However, technological limitations such as hand placement and wiring of components hampered research progress. Freeman's invention of the FPGA [4] in the 1980s generated renewed interest in RC-based research and development. Perhaps the earliest example of a commercially available RC was the Algotronix CHS2x4, which was featured in the international version of BYTE magazine [5]. While not a commercial success, it did set the stage for future efforts. A number of modern HPRCs that combine general purpose processors (GPPs) and FPGAs as the "fixed plus variable structure" are now available. Maxeler Technologies, for example, offers the MAX3 dataflow compute card [6], which was used by JP Morgan to reduce its risk analysis run time from 8 hours to about 4 minutes [2]. SRC Computers [7] offers the MAP processor, which was used by Lockheed Martin in the Synthetic-Aperture Radar (SAR) unit that flies aboard the U.S. Army's MQ-9 Unmanned Air Vehicle (UAV) [8]. Mercury Computer Systems offers FPGA-based compute boards such as the Ensemble MXI-205 [9], which is used in several of their application ready subsystems [10].

B. HLL development flow

As noted above, RCs have been successfully used to speed up applications. However, acceleration of floating-point applications is still challenging. There are a host of contributing issues; the loop-carried dependence associated with pipelined floating-point functional units, for example, makes it difficult to fully pipeline floating-point

doi:10.4304/jcp.8.4.859-873

kernels such as sparse matrix vector multiply. There have been some successes in these application areas [11], [12]. However, mapping kernels onto RCs is still primarily an art form, which relies upon the skill and experience of the developer to craft a customized solution on a case-by-case basis. By way of example, the JP Morgan effort motioned earlier involved the mapping of only two kernels, yet took about 3 years. Mainstream computer users simply can not tolerate such a lengthy development cycle. If HPRCs are to be a part of mainstream computing, development environments must move away from HDL-based *hardware design* toward HLL-based *programming*.

Several companies including Mentor Graphics and SRC Computers [13], [14] have introduced HLL-to-HDL compilers, which allow scientists and engineers to program HPRCs using HLLs (as with traditional computers) rather than employing HDL-based hardware designs (as with traditional FPGA-based circuits). It should be noted that these HLL-to-HDL compilers are not a panacea, especially for floating-point applications. One of the authors' attempts to speed up a simple sparse matrix kernel using an earlier generation HLL-to-HDL compiler resulted in a 10-fold slowdown. Even integer kernels can be problematic; Park's attempt to accelerate the Blowfish kernel showed over a 40-fold slowdown when implemented using the DIME-C HLL-to-HDL compiler [15]. To make MHz-scale FPGAs competitive with GHz-scale GPPs, we must have deeply pipelined, highly parallelized FPGA-based designs. Therefore, modern HLL-to-HDL compilers include enhanced HLL features such as pipelined loops, parallel code blocks, communication channels, synchronization primitives, and intellectual property interfaces (IPI) to access vendor-supplied or (when necessary) user-supplied intellectual property (IP) cores. Note that the IPI allow the IP cores to be "called" as though they were standard parameterized HLL subprograms. Section IV gives a specific example on how to map HDL-based user IP components into an HLL-based development flow.

Fig. 1 illustrates the HLL development flow. The design is partitioned into software modules, which are targeted for execution on the GPP, and hardware modules, which are targeted for execution on the FPGA. During development, the HLL software modules are compiled with a standard HLL compiler to produce object files. The linker uses the object files, library files, hardware module call specification, and the FPGA configuration bitstream (treated as data by the linker) to produce a binary executable. The HLL hardware modules are ingested by the HLL-to-HDL compiler, which emits HDL. This HDL output is used by the standard FPGA tool chain to produce an FPGA configuration bitstream, which when loaded onto the FPGA programs the FPGA with a hardware implementation of the design specified in the HLL code. From the viewpoint of the software module HLL code, the hardware module looks like a simple parameterized subroutine call. Note that the hardware module developer provides an API, e.g., header file, that

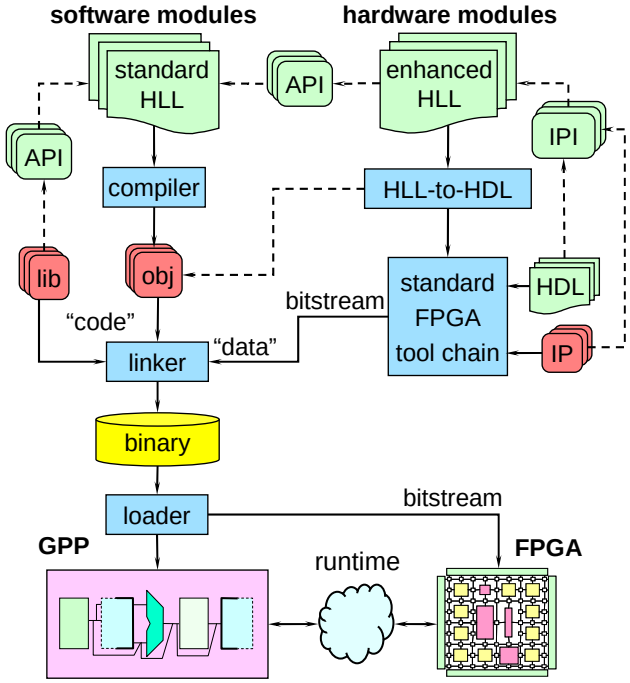


Fig. 1. HLL development flow

describes the call specification. From the viewpoint of the hardware module HLL code, the IP cores also look like subroutine calls. In this case, the vendor-supplied or user-supplied IPI describes the call specification. At the beginning of execution, the configuration bitstream is extracted from the binary executable data segment and loaded onto the FPGA. The GPP then executes the machine code instructions and invokes the FPGA-based kernel as needed via a run time support capability.

C. Description of target HPRC

There are a number of different RC architectures. One of the more popular architectures, as suggested by Fig. 2 and alluded to in Section II-B, is when the FPGA-based processing element (PE) essentially functions as a reconfigurable coprocessor that is invoked by a traditional GPP-based PE. In this model, there is some form of high-speed interconnect between the GPP-based fixed PE and the FPGA-based variable structure PE. It should be noted that other RC node architectures are extant, for example, FPGA as *peer* PE, FPGA as as *sole* PE, etc. In a typical application based on the architectural model shown in Fig. 2, the GPP would marshal a data set into the global common memory of the variable structure PE, and then call the FPGA-based kernel. The FPGA kernel would copy the data into the multibank local memory to allow for multiple simultaneous accesses. In the SRC-7 MAPstation-based [7] system used in our research, for example, there are sixteen local memory banks, which allow the FPGA to simultaneously fetch up to sixteen 64-bit words per FPGA clock cycle. The simultaneous memory access is required to support the parallelism needed to achieve a speedup on an FPGA.

The HPRC used in this research is a cluster of SRC-7 RC compute nodes [16] with a traditional login/control node, as idealized in Fig. 3. According to McGrath [17], the Jackson State University SRC-7 cluster is described as "the world's first InfiniBand-based SRC-7 MAPstation cluster." In Fig. 3, each RC compute node is as idealized by Fig. 2. Note that all InfiniBand MPI communication is handled by the Xeons; the variable structure MAP processors (which contain the FPGAs) serve strictly as reconfigurable coprocessors to accelerate selected kernels within the portion of the application that is running on the given compute node. The first Ethernet port on the login node is used to connect to the outside world over ssh. The second Ethernet port is used as a local (private) interconnect between the login node and all the compute nodes. This allows our researchers to ssh into a compute node if necessary. In addition, we use an NFS-mounted file system (over Ethernet) to export the user's RAID-protected home directory on the login node to all the compute nodes. In a production system, one would most likely use a parallel file system such as pNFS [18].

Each SRC-7 RC node has two 3.0GHz Intel Xeon processors with a 16K L1 cache, 2MB L2 cache, and 6GB RAM. The MAP Series H reconfigurable processor contains two Altera EP2S180F1508C3 FPGAs running at 150MHz. There are sixteen 64-bit-wide banks of local on-board memory (OBM) associated with the FPGAs, which provide up to 64MB of local memory; and there are two 64-bit banks of global common memory (GCM), which provide 2GB of memory that can be read to and written from by both the Xeons and the FPGAs. As noted earlier, the GCM is used primarily as a store wherein the Xeons and FPGAs marshal and retrieve large data sets. The MAP H processor is connected to the Xeon motherboard through a SNAP D interface, which has an 8GB/s bandwidth. To maximize performance, the SRC-7 uses a memory-based mapping rather than an I/O-based mapping. Direct memory access (DMA) is used to move data between the Xeon memory and MAP memory. In addition, the MAP processor has a streaming DMA capability, and an inter/intra-FPGA streaming capability that allows overlapping communication and computation

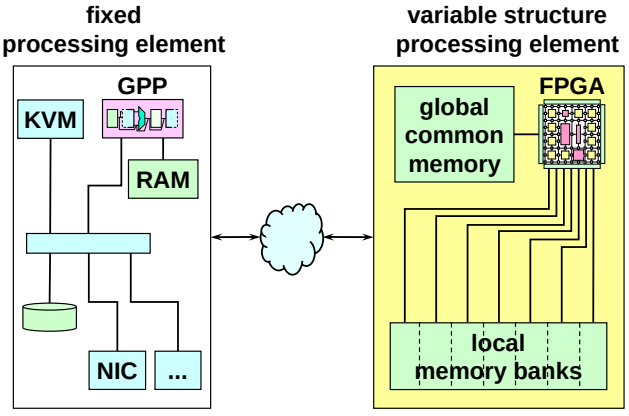


Fig. 2. RC node architecture

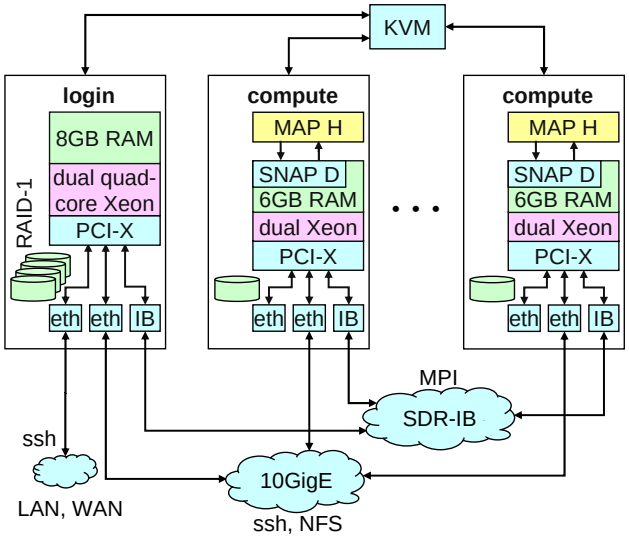


Fig. 3. HPRC cluster

to facilitate parallelism within the MAP processor.

SRC's Carte v4.0 programming environment is tightly integrated with their HPRC hardware. Carte automatically handles interchip communication, I/O pin mapping, and other necessary (but uninteresting) hardware details. This frees the developer to concentrate on mapping the algorithm onto hardware. Carte also directly supports the HLL development flow using either C or FORTRAN. In our research, we have elected to use Carte C. For the software-only components, the Intel C compiler v11.1 was used. Mapping of algorithms onto this novel SRC-7 HPRC system was successfully demonstrated in [19], [20], [21], [22], [23], [24], [25].

III. Design Considerations

This section takes a detailed look at the three p's, which highlights the critical relationship among performance, pipelining, and parallelism. It then examines the FPGA design boundary, which addresses some of the heuristics allowing developers to select candidate kernels that 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, something must be done at the design level for the FPGA to compete with the GPP. As suggested by Fig. 4, the performance of an algorithm on an FPGA is proportional to the extent to which it is pipelined and parallelized.

There are five simultaneous datapaths producing six outputs. Each of the datapaths is also pipelined to allow multiple operations within a data path to overlap. This multiplicative effect, which is known as the three p's, expresses the important relationship among performance, pipelining, and parallelism. Failure to either pipeline or parallelize a kernel generally results in poor performance

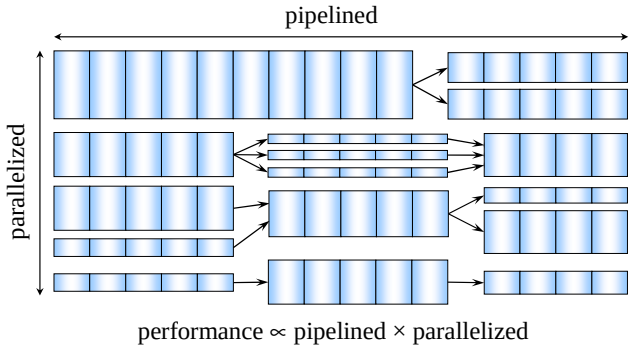


Fig. 4. The three p's

```
1:  algorithm SNIPPET(... parameter list ...)
2:      ⋮
3:      a2 ← 0.5/a     // break into atomic ops
4:      bb ← b · b     // to be parallelized
5:      ac ← a · c     // and pipelined
6:      mb ← −b        // by HLL-HDL compiler
7:      ac4 ← 4 · ac
8:      D ← bb − ac4
9:      sqr ← √D
10:     bPsqr ← mb + sqr
11:     bMsqr ← mb − sqr
12:     x1 ← bPsqr · a2
13:     x2 ← bMsqr · a2
14:     ⋮
15: end algorithm
```

Fig. 5. Quadratic equation algorithm snippet

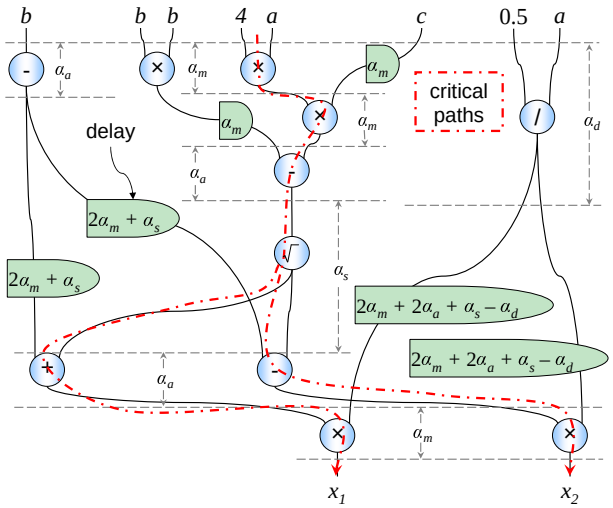


Fig. 6. Pipelined, parallelized quadratic formula

on an RC. We can pipeline and parallelize the quadratic formula [25] as depicted in Fig. 6, where $\alpha_m, \alpha_a, \alpha_s$, and α_d represent the latencies of the floating-point multiplier, adder, square rooter, and divider IP, respectively. A huge advantage of a modern HLL-to-HDL compiler is that it builds a data-flow graph, determines the true data dependences, and automatically calculates delays such that all the parallel paths are of equal length. The developer simply has to "give the compiler a chance" to find the parallelism, e.g., as depicted by the algorithm snippet in Fig. 5. Ideally, several of these fully pipelined datapaths should be used in parallel to implement the FPGA module.

Other techniques associated with parallelizing and pipelining a kernel include performing parallel I/O and overlapping communication with computation. Obviously, the best scenario is when we can crunch the data as they flow by, i.e., when we can create a systolic array. At the other end of the spectrum are those cases, such as an iterative solver, when we must first bring in the data set, then do the computation, and then do the output. The in-between cases include partially or completely overlapping computation with input or output.

B. FPGA design boundary

Determining the FPGA design boundary, i.e., determining which application modules should be mapped onto FPGAs, is not straightforward. As with many engineering disciplines, we must also rely on heuristics derived from empirical observation. Some areas to be considered when determining the FPGA design boundary include, 1) the three p's, 2) expected overall speedup, 3) expected resource utilization, 4) control/memory intensive vs. compute intensive, 5) monolithicity of modules, 6) available bandwidth, 7) opportunities for data reuse, 8) algorithm design stability, 9) algorithm efficiency, and 10) memory access patterns, as follows:

The three p's – Perhaps the most important heuristic is the three p's previously described. 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.

Expected overall speedup – If the objective of mapping a kernel onto an RC is to obtain a speedup relative to the performance of a GPP, then we need to look at the expected *overall* speedup. Overall speedup can be quantified via Amdahl's Law [26]

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

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. Amdahl's Law is often the basis for design decisions and can help us avoid costly design mistakes based on deceptive intuition. For example, suppose the estimated speedup for the postprocessing system were a thousandfold, i.e., an FPGA implementation of the postprocessing kernel were estimated to run an incredible 1000 times faster than an equivalent software module. Intuition says the FPGA-based postprocessing kernel would yield a significant overall speedup. Suppose

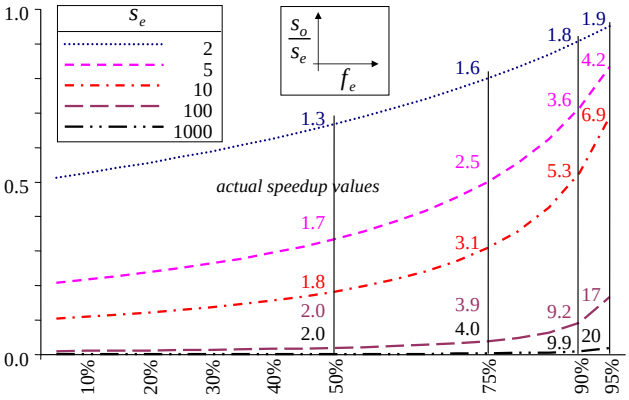


Fig. 7. Fast is not always *fast*

postprocessing only constitutes 10 percent of the runtime. After applying Amdahl's Law,

$$s_o = \frac{1}{1 - 0.10 + 0.10/1000} = 1.11,$$

we see that the overall speedup associated with the FPGA-based kernel is only about 10 percent. This relationship between overall speedup and the fraction of a system that can take advantage of the speedup is depicted in Fig. 7. The abscissa represents the fraction of the system to be enhanced (f_e), and the ordinate represents overall speedup (normalized to the speedup of the portion to be enhanced, i.e., s_o/s_e). The plotted lines represent the normalized speedup values for five scenarios, $s_e \in (2, 5, 10, 100, 1000)$. The vertical lines are the actual overall speedup 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 $s_e = 1000$. If 75 percent of the system could take advantage of a thousandfold speedup, the overall speedup value will only be 4.0. 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 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 important to consider whether an algorithm is control/memory intensive or compute intensive. In hardware, control flow is implemented as multiple hardware paths with a mux at the end. Therefore, a kernel with many control clauses is likely to result in large (and slow) hardware. Harkins et al. illustrate the importance of this concept when they show that comparison-sorting algorithms do not perform well on an HPRC [27].

Monolithicity of modules – 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 [28]. This might minimize the effects of bandwidth limitations. A closely related issue is data reuse, to be discussed next.

Opportunities for data reuse – Algorithms that have a significant potential for data reuse may be suitable FPGA module candidates. We used this principle to speed up two well-known iterative solvers [29]. 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, we might 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, $O(n!)$ (Habgood and Arel notwithstanding [30]), might run faster if implemented on an FPGA. However, Gaussian elimination, with complexity, $O(n^3)$, is a much more efficient algorithm. We should first try a more efficient software solution rather than map inefficient algorithms onto an FPGA.

Memory access patterns – If a candidate kernel has large or irregular stride memory access patterns, then it is more likely to do well on an FPGA, which does not depend upon the memory access patterns like a cache-based GPP. See [24] for a more detailed discussion of this issue.

IV. Integrating HDL-based Components

A. Standard integration approach

In some cases, it is desirable to use HDL-based IP cores in an RC design, e.g., when the enhanced HLL does not provide the needed capability. The standard way in which Carte integrates IP (Carte refers to these IP cores as *user macros*) is by 1) specifying a *header* file to be included by the hardware module code, and 2) specifying a *blackbox* file and an *info* file via Carte environment variables. We have coined the phrase intellectual property interface (IPI) to describe the integration mechanism. As suggested by Fig. 8, the blackbox is the HDL interface to the IP; the info file contains several properties of the IP including a connection between the IP core name and the HLL name; and the header file is the HLL interface. Note that a user

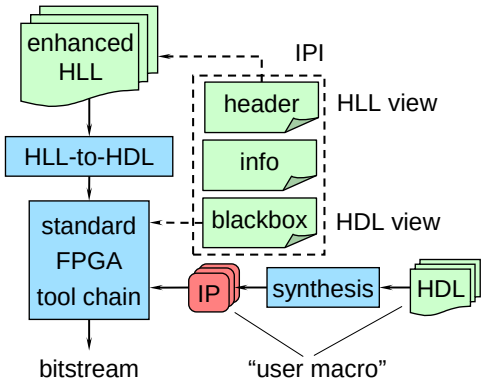


Fig. 8. Intellectual property interface

macro can also be a synthesized IP, i.e., netlist only (no HDL).

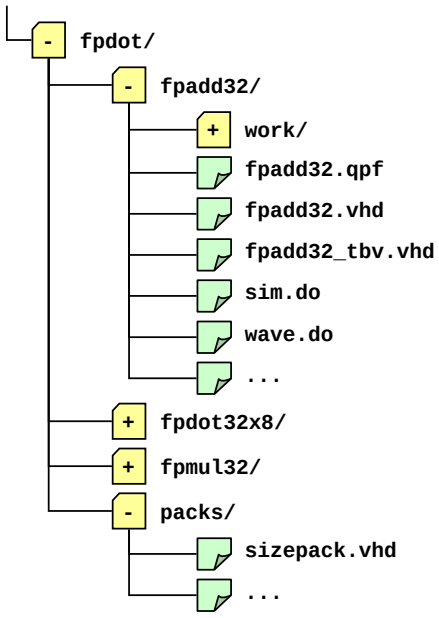
The default Carte mechanism for incorporating user macros does not directly support a multiple file, multiple directory VHDL hierarchy. The following sections give an example of integrating such hierarchical components into the Carte environment. We will use an HDL-based dot product user macro. Note that Carte *can* deal with the generation of a dot product unit at the HLL level; we use it as example because it is an easy kernel to understand, yet can be built in the hierarchical fashion typical of an HDL-based design.

B. Primitive floating-point components

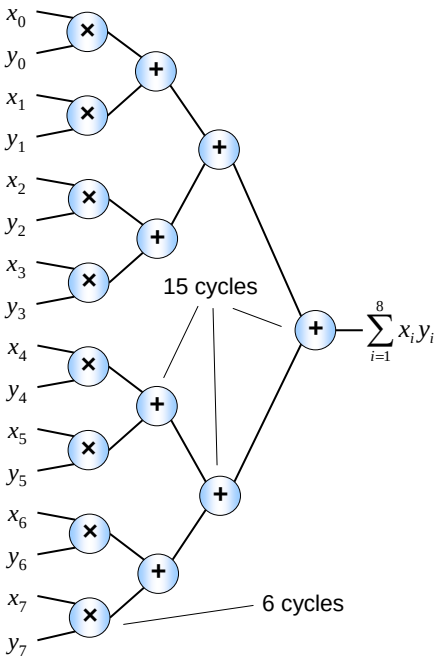
The Quartus [31] Mega-wizard tool allowed us to create RTL VHDL codes using a simple graphical user interface (GUI). We used the tool to generate pipelined, IEEE 32-bit floating-point adder and multiplier components entitled `fpadd32.vhd` and `fpmul32.vhd`, respectively. After testing the designs using ModelSim [32], these primitive components were used to build a pipelined, 32-bit floating-point, 8-input structural VHDL dot product component entitled `fpdot32x8.vhd`.

C. Dot product component

Fig.9(a) is an elided representation of the directory hierarchy for the VHDL-based floating-point dot product. Associated with the adder (for example) are the ModelSim VHDL compiler library directory `work/`, Quartus project file `fpadd32.qpf`, design file `fpadd32.vhd`, test bench file `fpadd32_tbv.vhd`, ModelSim simulation scripts `sim.do` and `wave.do`, support package `../packs/sizePack.vhd`, and several other files. An analogous substructure exists for the multiplier and dot product unit. Fig. 9(b) illustrates the 51-stage pipelined dot product binary tree. Since `fpdot32x8.vhd` was written using a structural VHDL approach, it incorporated the `fpadd32` and `fpmul32` components. The 6-cycle first stage used the multiplier. The 15-cycle second, third, and output stage used the adder. A key mechanism in the creation of `fpdot32x8.vhd` was the use of generate loops and the associated arrays



(a) directory hierarchy



(b) binary tree pipeline

Fig. 9. Dot product

of standard logic vectors. We only had to write 164 lines of VHDL code, since the Quartus Mega-wizard plug-in produced 6K or more lines of highly tuned RTL VHDL code. The architectural hierarchy of this design requires `sizePack.vhd` to be compiled before `fpmul32.vhd` and `fpadd32.vhd`. These latter two have to be compiled before `fpdot32x8.vhd`.

D. Dot product intellectual property interface

The directory structure of the entire RC-based dot product is shown in Fig. 10. We will only mention some of the files in passing, since the purpose here is to describe the IPI. The software module that is executed on the

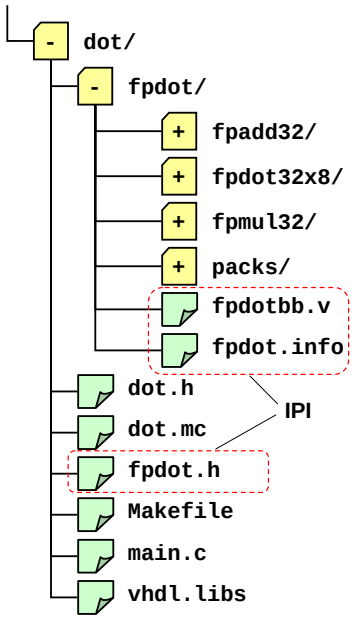


Fig. 10. RC-based dot product

general purpose processor is in `main.c`. It generates some vectors and calls the hardware module code that is represented in the `dot.mc` file. The `dot.h` file is the user-generated API that describes the user interface for the hardware module. The `Makefile`, which we will discuss later, orchestrates the system build. The `vhdl.libs` will also be described later. The `fpdot/` directory is as discussed earlier with two important additions, `fpdotbb.v` and `fpdot.info`. These two files and `fpdot.h` constitute the IPI described in the next section.

1) Header, blackbox and info files: The header file, `fpdot.h`, does not require much elaboration; it simply contains the prototype of the C interface used by the `dot.mc` hardware module code when it calls the dot product IP. Of more interest are the blackbox and info file used to tell the Carte compiler about the VHDL-based IP. The first file, `fpdotbb.v`, is a Verilog blackbox file that describes the interface for the dot product and associated IP. In essence, the blackbox file tells Carte how to deal with the user macros when it begins to synthesize the HDL emitted by the HLL-to-HDL compiler. The blackbox entry for the `fpadd32.vhd` file would be as shown in Fig. 11. Analogous entries would be created for `fpmul32` and `fpdot32x8`. The second file, `fpdot.info`, is the info file that contains information about the user macro that is needed by the Carte HLL-

```
module fpadd32(clk, x, y, z);
    input         clk;
    input  [31:0] x;
    input  [31:0] y;
    output [31:0] z;
endmodule
```

Fig. 11. Blackbox entry for fpadd32

```
BEGIN_DEF "myfpAdd"    // Carte-C name
   MACRO = "fpadd32";  // VHDL name
   EXTERNAL = NO;      // Chap 11, Carte C
   PIPELINED = YES;    // pipelined
   LATENCY = 15;       // 15 clock cycles

   // clk is invisible to C
   IN_SIGNAL : 1 BITS "clk" = "CLOCK";
   INPUTS = 2:         // input parameters
      I0 = FLOAT 32 BITS (x[31:0])
      I1 = FLOAT 32 BITS (y[31:0])
      ;
   OUTPUTS = 1:        // output parameter
      O0 = FLOAT 32 BITS (z[31:0])
      ;
   DEBUG_HEADER = #
      void myfpAdd__dbg(
        float x,
        float y,
        float *z
      );
   #;
   DEBUG_FUNC = #
      void myfpAdd__dbg(
        float x,
        float y,
        float *z) {
         // functional equivalent
         // of VHDL code behavior
         *z = x + y;
      }
   #;
END_DEF
```

Fig. 12. Info entry for fpadd32

to-HDL compiler. This includes such things as latency, name of netlist file, name to be used in the Carte C code, etc. The info file also contains *debug* code that provides a software-only functional equivalent of the IP that is used by Carte when creating a debug build. The debug build is a software-only equivalent of the hardware module designed to test the functionality without requiring a lengthy synthesis, place and route, and bit generation cycle. The info file entry for the `fpadd32.vhd` file, for example, would be as shown in Fig. 12. Analogous entries would be created for `fpmul32` and `fpdot32x8`.

2) Makefile modifications: The Carte development environment includes a `Makefile` that is tailored for each new development. Without going into too much detail, the `Makefile` is different than what we normally see with a software build. Essentially, a Carte `Makefile` defines a set of environment variables and then fires off the Carte compiler to do the work. A default Carte `Makefile` assumes the user macro is a single VHDL file contained in a single directory. As noted above, this is usually not the case. Therefore, we had to do some research to figure out how to coax Carte into dealing with a multiple-file, multiple-directory scenario. Our modified `Makefile` is as shown in Fig. 13. We have included line numbers to facilitate the discussion that follows. The first part of the puzzle is to tell Carte the name of the VHDL library that will contain the compiled

```
01:  FILES = main.c
02:  MAPFILES = dot.mc
03:  BIN = dot
04:
05:  # location of blackbox and info files
06:  MY_BLKBOX = fpdot/fpdotbb.v
07:  MY_INFO = fpdot/fpdot.info
08:
09:  # tell Carte about the default VHDL library
11:  # and directories containing the netlists
12:  MCCFLAGS   +=  -param_file $(PWD)/vhdl.libs \
13:                 -ngo_dir ../fpdot/packs \
14:                 -ngo_dir ../fpdot/fpadd32 \
15:                 -ngo_dir ../fpdot/fpmul32 \
16:                 -ngo_dir ../fpdot/fpdot32x8
17:
18:  # from here on it is just like normal
19:  CC = icc
20:  LD = icc
21:  CFLAGS = -O3 # optimize the C code
22:
23:  # No modifications are required below
24:  MAKIN      ?=  $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
25:  include $(MAKIN)
```

Fig. 13. Makefile

```
set_global_assignment -name VHDL_FILE sizePack.vhd  -library work
set_global_assignment -name VHDL_FILE fpadd32.vhd   -library work
set_global_assignment -name VHDL_FILE fpmul32.vhd   -library work
set_global_assignment -name VHDL_FILE fpdot32x8.vhd -library work
```

Fig. 14. vhdl.libs

netlists. To accomplish this task, we create an auxiliary file called (in this case) `vhdl.libs` and send it to the Carte compiler via the `-param_file` flag, as shown on line 12 of the `Makefile`. As shown in Fig. 14, `vhdl.libs` simply lists each VHDL file and associates it with a VHDL library. During the early stages of a system build, the Carte compiler invokes the Quartus synthesizer to compile the VHDL code in the `fpdot/` directory. At that time, the Carte compiler's default directory is the same as the directory from which the `Makefile` was invoked. Therefore, the full path name of the `vhdl.libs` file is just `$(PWD)/vhdl.libs`, where `PWD` is the environment variable containing the present working directory. The second part of the puzzle is to tell Carte where the compiled netlists are located (the Unix directory, not the VHDL library). For a single netlist, this is usually accomplished via the predefined Carte variable called `MY_NGO_DIR`. Since Carte interprets this environment variable as a single directory name, we cannot use it to pass along multiple directories. Therefore, we use multiple instances of the Carte `-ngo_dir` flag, one for each netlist directory. The complication is that by the time Carte references the MCCFLAGS environment variable to extract the netlist directories, Carte is operating in a different (new) subdirectory it has created under `dot/`. Therefore, we prepend a relative path reference to obtain the full path name of each netlist directory, as shown on lines 13 – 16 of the `Makefile`.

After creating the IPI infrastructure, we simply have to do a `make`, and the Carte compiler produces the desired executable. See [22] and [21] for additional details on this process.

V. Mapping an iterative solver onto an HPRC

To illustrate the use of some of our ideas, we will show the mapping of a simple floating-point sparse matrix Jacobi iterative solver onto an HPRC.

A. Derivation of the Jacobi method

Discussions of the Jacobi iterative method can be found in introductory numerical analysis textbooks such as [33] or [34]. For our simple solver, we assume real-valued matrices and vectors. Let A be an $n \times n$ strictly diagonally dominant matrix; let $\mathbf{x}$ be the unknown n-vector; and let $\mathbf{b}$ be the constant n-vector. To solve $A\mathbf{x} = \mathbf{b}$ iteratively, the next approximate solution, $\mathbf{x}^{(\delta+1)}$, is computed as a function of the current approximation, $\mathbf{x}^{(\delta)}$, where δ is the iteration index. Substituting $A = L + U + D$ into $A\mathbf{x} = \mathbf{b}$, where L is the strictly lower triangular part of A, U is the strictly upper triangular part of A, and D is the diagonal, and manipulating it into the standard iterative form $\mathbf{x}^{(\delta+1)} = f(\mathbf{x}^{(\delta)})$ results in

$$\mathbf{x}^{(\delta+1)} = D^{-1}\left(\mathbf{b} - (L+U)\,\mathbf{x}^{(\delta)}\right), \qquad (1)$$

Combining residual vector, $\mathbf{r}^{(\delta)} = A(\mathbf{x}-\mathbf{x}^{(\delta)})$ with $A\mathbf{x} = \mathbf{b}$ produces

$$\mathbf{r}^{(\delta)} = \mathbf{b} - A\mathbf{x}^{(\delta)}. \quad (2)$$

Substitute $A = L + U + D$ into Equation 2 to obtain

$$\mathbf{r}^{(\delta)} + D\mathbf{x}^{(\delta)} = \mathbf{b} - (L+U)\mathbf{x}^{(\delta)}. \quad (3)$$

Combining Equation 1 and Equation 3 yields

$$\mathbf{x}^{(\delta+1)} = D^{-1}\mathbf{r}^{(\delta)} + \mathbf{x}^{(\delta)}. \quad (4)$$

The Jacobi iterative solver first calculates Equation 2 and then uses the result to calculate Equation 4. This approach yields the residual vector needed for the termination test. A common termination test, often referred to as residual norm, is shown in Equation 5

$$\frac{\|\mathbf{r}^{(\delta)}\|}{\|\mathbf{b}\|} < \varepsilon, \quad (5)$$

where $\|\cdot\| \equiv \|\cdot\|_2$ is the 2-norm, e.g., $\|\mathbf{x}\| = \sqrt{\sum x_i^2}$, and ε is some suitably small value. This is the termination criteria used by our Jacobi iterative solver.

B. Compressed sparse row format

If a matrix of order n has a small number of nonzero values, n_z, then compressed sparse row (CSR) format [35] can reduce both storage and computational requirements. CSR uses vectors **val**, **col**, and **ptr** to store only the nonzero values and identify the row and column indices. Vector **val** is a real vector of length n_z containing the nonzero values obtained via a row-wise matrix traversal. The **col** integer vector is also of length n_z and contains the column index of each nonzero value. The **ptr** integer vector is of length $n+1$ and contains the index in **val** where each matrix row starts, i.e., the first nonzero element of matrix row i is found at index ptr_i of **val**. To facilitate consistent usage, we let $ptr_{n+1} = n_z + 1$. For example, the order $n = 4$ sparse matrix with $n_z = 8$ nonzero values shown in dense format in Fig. 15(a) is shown in CSR format in Fig. 15(b). Consider $ptr_3 = 5$; this indicates that row 3 of the matrix begins at index 5 of **val** and **col**. Notice that $val_5 = a_{33}$ and that $col_5 = 3$ as required.

$$\begin{bmatrix} a_{11} & 0 & a_{13} & a_{14} \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & a_{42} & 0 & a_{44} \end{bmatrix}$$

(a) dense format

	1	2	3	4	5	6	7	8
val	a_{11}	a_{13}	a_{14}	a_{22}	a_{33}	a_{34}	a_{42}	a_{44}
col	1	3	4	2	3	4	2	4
ptr	1	4	5	7	9			

(b) CSR format

Fig. 15. Matrix formats

	1	2	3	4	5
kval	$[a_{11}, a_{13}]$	$[a_{14}, 0]$	$[a_{22}, 0]$	$[a_{33}, a_{34}]$	$[a_{42}, a_{44}]$
kcol	$[1, 3]$	$[4, 1]$	$[2, 1]$	$[3, 4]$	$[2, 4]$
kptr	1	3	4	5	6

Fig. 16. Aligned CSR format

C. Aligned CSR format

To parallelize our Jacobi processor, $k = 8$ values are read from **val** and **col** during each clock cycle. This is accomplished by striping these vectors across multiple memory banks as alluded to in Section II-C. To avoid large multiplexers and the associated degradation in performance discussed in Section III-B, matrix rows are padded with zeros (as needed) to ensure they have an exact multiple of k values. The **ptr** vector is modified to express indices in terms of k-sized groups of data, and the term kn_z (analogous to n_z) is the number of k-groups. This entire process, which is carried out by the software module when it marshals the data for the FPGA-based processor, is known as "k-alignment." This k-alignment process produces scalar kn_z and vectors **kval**, **kcol**, and **kptr**. As before, $kptr_{n+1} \equiv kn_z + 1$. If we assume that $k = 2$, then the sparse matrix represented in Fig. 15(b) could be represented in k-aligned CSR format as shown in Fig. 16. Each bracketed k-group of data represents the contents at a given index across a striped data set. Consider $kptr_4 = 5$; this indicates that matrix row 4 begins at index 5 of striped memory banks **kval** and **kcol**. Notice $kval_5 = [a_{42}, a_{44}]$ (2 banks, 2 values) and that $kcol_5 = [2, 4]$ (2 banks, 2 indices). Also notice that we simply set the column index of the padded 0 values to 1 since it does not really matter anyway

D. Sparse matrix Jacobi algorithm

An algorithm for the Jacobi iterative method is shown in Fig. 17; it deals with CSR format matrices.

```
1:  algorithm SWJACOBI( val, col, ptr, b, x^(δ), δ)
2:      δ ← 0
3:      repeat
4:          for i in [1, n] do
5:              r_i^(δ) ← b_i
6:              for j in [ptr_i, ptr_{i+1}) do
7:                  r_i^(δ) ← r_i^(δ) − val_j · x_{col_j}^(δ)
8:                  if col_j .EQUATION i then aii ← val_j
9:              end for
10:             x_i^(δ+1) ← r_i^(δ)/aii + x_i^(δ)
11:         end for
12:         x^(δ) ← x^(δ+1)
13:         δ ← δ + 1
14:     until ‖r^(δ)‖/‖b‖ < ε .OR. δ > δ_max
15: end algorithm
```

Fig. 17. Sparse matrix Jacobi algorithm

values	6	9	4	3	5	1	7	8	2
counts	3	4	2						
sums	19	16	10						

Fig. 18. Streaming accumulator example

E. Streaming accumulator

One of the features of the latest Carte compiler is a streaming accumulator, which can be used to overcome the loop-carried dependence associated with variable-length inner loops like that of the sparse Jacobi method shown in Fig. 17. In previous research efforts, we were forced to build our own HDL-based streaming accumulator [36], so we were quite pleased when the new compiler provided this capability. In some sense, this justifies one of the goals of our research, i.e., to get good ideas to be adopted by hardware and compiler vendors. At any rate, the accumulator accepts two input FIFO streams, **values** and **counts**, and emits an output stream, **sums**. The relationship between these three streams is perhaps best understood by way of a simple example. In Fig. 18, the **values** stream represents the three vectors, $\mathbf{v1} = [6, 9, 4]$, $\mathbf{v2} = [3, 5, 1, 7]$, and $\mathbf{v3} = [8, 2]$. The three values in the **counts** stream indicate the number of values in each vector, i.e., $|\mathbf{v1}| = 3$, $|\mathbf{v2}| = 4$, and $|\mathbf{v3}| = 2$. The streaming accumulator uses **values** and **counts** to compute the three vector sums, $s1 = \sum v1_i = 19$, $s2 = \sum v2_i = 16$, and $s3 = \sum v3_i = 10$ and send them to the **sums** FIFO stream. The streaming accumulator generally requires four parallel sections as suggested by the pseudo-code snippet shown in Fig. 19. In the Jacobi processor, which we will see momentarily, the **values** stream corresponds to the series of partial dot products emitted, one per clock cycle, by a fully pipelined $8 \cdot 8$ dot product unit. The **counts** stream corresponds to the number of partial dot products per matrix row. The **sums** stream contains the full dot products, $\sum a_{ij} \cdot x_j^{(\delta)}$, needed to compute the residual vector shown in Equation 2.

```
algorithm WHATEVER(···)
  ···
  parBegin
    p1:  // feed values stream
    ···
    V_FIFO ← ···

    p2:  // feed counts stream
    ···
    C_FIFO ← ···

    p3:  // streaming accumulator
    S_FIFO ← Σ_STREAM(V_FIFO, C_FIFO)

    p4:  // consume sums stream
    ··· ← S_FIFO
    ···
  parEnd
  ···
end algorithm
```

Fig. 19. Streaming accumulator usage

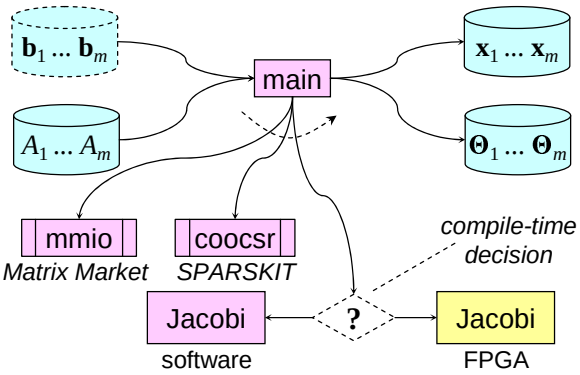


Fig. 20. High-level Jacobi design

F. High-level Jacobi design

The high-level design for the Jacobi iterative solver is shown in Fig. 20. It consists of four major components: a main routine and matrix support libraries; several strictly diagonally dominant sparse matrices, $A_1 \ldots A_m$; the software or hardware (FPGA-based) Jacobi iterative solver; and the output result and statistics files, $\mathbf{x}_1 \ldots \mathbf{x}_m$ and $\Theta_1 \ldots \Theta_m$. The $\mathbf{b}_i$ vectors are shown as inputs, but for the experiments in this research they are generated from a known $\mathbf{x}$ vector at run time. The main routine is a driver program, which essentially measures how long it takes for Jacobi to solve each set of equations. The coordinate-format matrices are read in using the Matrix Market I/O library [37] and converted to CSR format using Saad's SPARSKIT library [35]. The software Jacobi kernel implementation is based on the algorithm shown in Fig. 17, and the FPGA-based Jacobi kernel will be described later. A compile-time decision selects either the software or FPGA-based version of Jacobi. At run time, main reads in each coordinate-format matrix, converts it to CSR format, and uses a known $\mathbf{x}$ vector to generate $\mathbf{b}$. It then invokes the selected Jacobi kernel sending matrix, A ($\mathbf{val}, \mathbf{col}$, and $\mathbf{ptr}$), starting point $\mathbf{x^{(0)}}$, and constant vector $\mathbf{b}$. After convergence, Jacobi stores the result and returns. The main routine writes the solution to the results file; it also writes the input matrix name, number of iterations, and wall clock execution time to the statistics file and then terminates.

G. Jacobi design considerations

This section parallels Section III and shows how the specific criteria were applied to the Jacobi design.

The three p's – The Jacobi iterative method is inherently parallelizable. Given sufficient hardware, one could do all the dot products in parallel. In the actual design, we were able to fully pipeline all inner loops and to parallelize the dot product unit, i.e., to adhere to the three p's.

Expected overall speedup – To determine what fraction of the run time was consumed by the Jacobi kernel, `gprof` was used to profile the software version of the code. Not too surprisingly, given that the main routine is only a driver, the Jacobi kernel consumed over 96 percent

of the run time for all data sets, i.e., $f_e = 0.96$. Thus, assuming even a modest value $s_e = 2$, an overall speedup $s_o = 1/(0.04 + 0.48) = 1.9$ is anticipated.

Expected resource utilization – In the case of the Jacobi solver, the limiting factor was the amount of block RAM (BRAM) in the FPGA fabric, with the number of OBM banks running a close second. Despite these limitation, we were able to build an 8-wide parallel Jacobi datapath capable of handling sparse matrices up to order, $n = 8$K, with up to approximately 4M nonzero entries.

Control/memory vs. compute intensive – Our Jacobi processor employs design features that minimize the number of control clauses, e.g., the use of k-aligned CSR format precludes the need for muxes at the dot product tree input. Furthermore, unlike a GPP memory hierarchy, the HPRC memory organization does not penalize irregular memory accesses [24]. Therefore, on an FPGA, Jacobi appears to be primarily compute intensive.

Monolithicity of modules – In the case of the Jacobi processor, the Carte compiler provided IPI for some of the lower level routines such as square root. Therefore, the entire Jacobi kernel was effectively monolithic and suitable for mapping onto an FPGA.

Available bandwidth – In the Jacobi processor, we were able to reduce the bandwidth requirement by marshaling the data into GCM banks and then using DMA to load the OBM and BRAM memory used during the FPGA-based computation.

Opportunities for data reuse – In the case of the Jacobi iterative solver, the matrix A and vector $\mathbf{b}$ are reused during every iteration. This had the effect of amortizing the transfer costs across all iterations.

Algorithm design stability – The Jacobi iterative solver has been around for a long time; obviously we did not anticipate any algorithm design modifications.

Algorithm efficiency – In the case of the Jacobi method, we acknowledge that the efficiency rule is being violated. Jacobi is *significantly* slower than other solvers. However, our paper is not suggesting that Jacobi is the best way to solve a set of equations nor is it trying to demonstrate the speedup of a particular application. Our primary purpose is to illustrate the mapping process.

Memory access patterns – Since the Jacobi processor uses a sparse matrix, it clearly demonstrates an irregular memory access pattern and is a good mapping candidate.

H. Parallelized Jacobi algorithm

The parallelized, pipelined algorithm for performing the Jacobi iteration on HPRC hardware is shown in Fig. 21. The algorithm operates in three phases: *input*, *compute*, and *output*.

1) Input phase: During *input*, three parallel blocks use direct memory access (DMA) to input the problem data from GCM. Lines 2–5 bring in the k-aligned CSR-format $\mathbf{kval}$ and $\mathbf{kcol}$ and store them, stripe-8, in the OBM banks. The striping of matrix values across multiple memory banks allows *compute* to read eight values per clock. Lines 6–9 and 10–13 bring in $\mathbf{kptr}$, $\mathbf{b}$, $\mathbf{d}$ ($1/a_{ii}$ values), and $\mathbf{x}^{(\delta)}$ and store them in BRAM arrays.

```
1:  algorithm HWJAC(kval, kcol, kptr, b, d, x^(δ), δ)
2:      parBegin  // only two GCM banks
3:          BUF_DMA_GCM1:OBM (kval, stripe-8)
4:          BUF_DMA_GCM2:OBM (kcol, stripe-8)
5:      parEnd
6:      parBegin  // so parallel DMA limited to
7:          STREAM_DMA_GCM1:BRAM (kptr)
8:          STREAM_DMA_GCM2:BRAM (b)
9:      parEnd
10:     parBegin  // two vectors at a time
11:         STREAM_DMA_GCM1:BRAM (d)
12:         STREAM_DMA_GCM2:BRAM (x^(δ))
13:     parEnd
14:     δ ← 0
15:     b2 ← 1/||b||_2
16:     repeat
17:         x1 ← ... ← x9 ← x^(δ)
18:         parBegin
              p1:  // feed values stream
19:           for i in [1, kn_z] do
20:             a ← (a1 ··· a8) stripe-8 from kval_i
21:             j ← (j1 ··· j8) stripe-8 from kcol_i
22:             x ← (x1_j1 ··· x8_j8)
23:             V_FIFO ← dot8Tree (a, x)
24:           end for
              p2:  // feed counts stream
25:           for i in [1, n] do
26:             C_FIFO ← kptr_{i+1} − kptr_i
27:           end for
              p3:  // streaming accumulator
28:           S_FIFO ← Σ_STREAM(V_FIFO, C_FIFO)
              p4:  // residual & next approximation
29:           for i in [1, n] do
30:             dotN ← S_FIFO
31:             r ← b_i − dotN
32:             x_i^(δ+1) ← r/d_i − x9_i
33:             MAC (r, r, r2)  // r2 ← Σ r_i^2
34:           end for
35:         parEnd
36:         x^(δ) ← x^(δ+1)
37:         r2b2 ← √r2 · b2
38:         δ ← δ + 1
39:     until r2b2 < ε .OR. δ > δ_max
40:     BUF_DMA_BRAM:GCM2 (x^(δ))
41: end algorithm
```

Fig. 21. Hardware Jacobi algorithm

2) Compute phase: Lines 16–39 constitute the *compute* phase. App. V-E provides details on Carte's new streaming accumulator. As mentioned, the need for an HLL-based streaming accumulator was identified by earlier research, e.g., [38], [36], [39], and that the compiler vendor subsequently provided that capability. Since the FPGAs do not have multiport memory to support nine address and data buses on a single memory bank, and since parallel sections $\mathbf{p}_1 \ldots \mathbf{p}_4$ operate simultaneously, independent banks are needed to avoid a multicycle pipeline. Therefore, line 17 creates nine copies of $\mathbf{x}^{(\delta)}$; eight for the dot product tree, and one to calculate $x_i^{(\delta+1)}$. Parallel section $\mathbf{p}_1$ (lines 19–24) is a fully pipelined $8 \cdot 8$ dot product unit. Each clock cycle it consumes the next eight a_{ij} values from $\mathbf{kval}$ and the matching eight values from $\mathbf{x}^{(\delta)}$, and outputs the resulting partial dot products (dot8s) to the V_{FIFO} stream. Parallel section $\mathbf{p}_2$ (lines 25–27) calculates the number of dot8s for

each row and sends them to the C_{FIFO} stream. Parallel section $\mathbf{p}_3$ (line 29) is the streaming accumulator that consumes the V_{FIFO} and C_{FIFO} streams, computes the n dot products, $\text{dotN}_i = \sum_j a_{ij} x_j^{(\delta)}$ for all i, and feeds the results into the S_{FIFO} stream. Parallel section $\mathbf{p}_4$ (lines 29–34) consumes the dotNs from S_{FIFO} and uses them to calculate the residual vector and next approximate solution. Via a multiply-accumulate (MAC), $\mathbf{p}_4$ also calculates $\sum r_i^2$ needed for the convergence test. Lines 36–38 complete the iteration; line 39 is the termination test.

3) Output phase: During *output*, the converged $\mathbf{x}^{(\delta)}$ value is DMAed to GCM as shown on line 40.

I. Detailed description of Jacobi processor

If the algorithm in Fig. 21 is implemented and compiled with the Carte compiler, it produces the FPGA-based Jacobi processor idealized in Fig. 22. The main routine is instrumented with a microsecond-resolution timer. The timer is started as the first possible executable statement and ended as the last possible executable statement in order to capture wall clock execution time. The main routine k-aligns the input data and marshals it into GCM where it is subsequently DMAed into either OBM or BRAM by the FPGA. OBM banks are used to store **kcol** and **kval**, while BRAM is used to store **kptr**, **d**, **b**, the multiple copies of $\mathbf{x}^{(\delta)}$, and $\mathbf{x}^{(\delta+1)}$. Recall, the processor has three phases: *input*, *compute*, and *output*. As suggested by the circular arrow in Fig. 22, *compute* consists of three subphases: *update*, which creates the nine copies of $\mathbf{x}^{(\delta)}$; *iterate*, which computes the residual vector and next approximate solution; and *test*, which determines

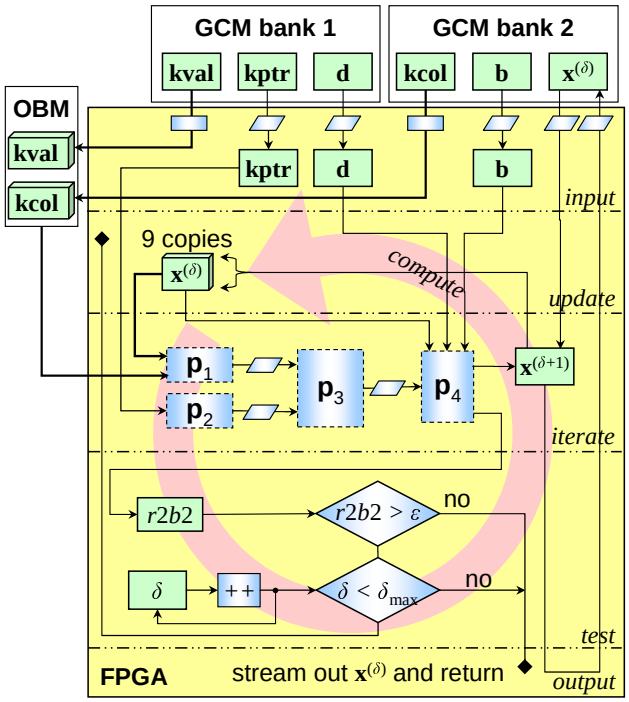


Fig. 22. Jacobi processor block diagram

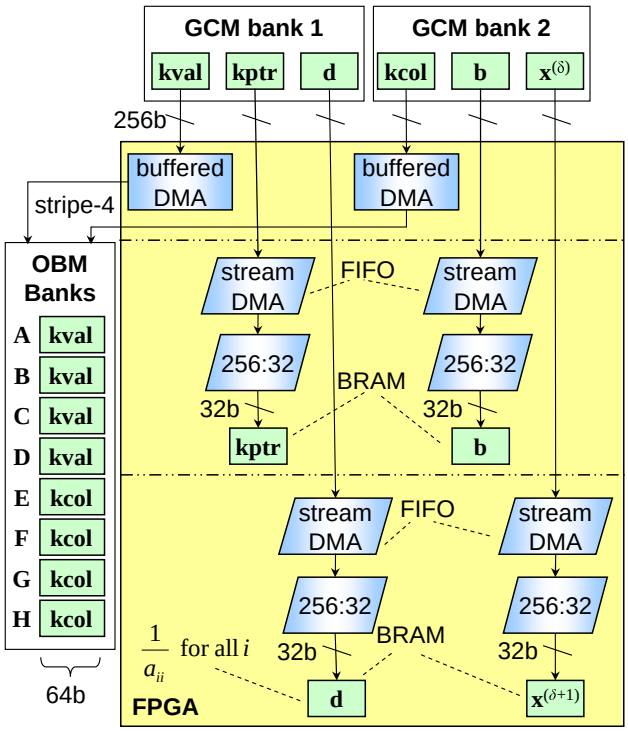


Fig. 23. Jacobi input phase

if the Jacobi iteration should terminate. As previously described, the processor has three phases: *input*, *compute*, and *output*. The following sections describe each of the Jacobi processor phases.

1) Input phase: As shown in Fig. 23, the *input* phase consists of three parallel sections. In the first parallel section, the Jacobi processor uses buffered DMA to read 256-bit words representing the **kval** and **kcol** vectors from the GCM and store them, stripe-4, in OBM banks. The result is four 64-bit packed representations of **kval** and four 64-bit packed representations of **kcol**. Each 64-bit value is later unpacked to produce two 32-bit values. In effect, **kval** and **kcol** are stored stripe-8, as indicated in lines 3–4 of Fig. 21. In the second parallel section, the Jacobi processor uses streaming DMA and a 256-bit to 32-bit stream width converter to store the **kptr** and **b** vectors in BRAM arrays on the FPGA. In the third parallel section, the processor again uses streaming DMA and a stream width converter to store the **d** ($1/a_{ii}$ for all i), and $\mathbf{x}^{(0)}$ vectors in BRAM. Note that the latter vector is stored in the $\mathbf{x}^{(\delta+1)}$ array to simplify loop entry.

2) Compute phase: The principal features of the *compute* phase are shown in Fig. 24. This block diagram does not show the trivial *update* phase (see Fig. 22). It does show the four parallel sections that constitute the *iterate* subphase, and the relatively simple calculations associated with convergence in the *test* subphase.

Values stream: Parallel section $\mathbf{p}_1$ is an $8 \cdot 8$ dot product unit. Notice that it reads in the next four 64-bit packed values from **kval** and **kcol** and splits them into the corresponding eight 32-bit values and eight 32-

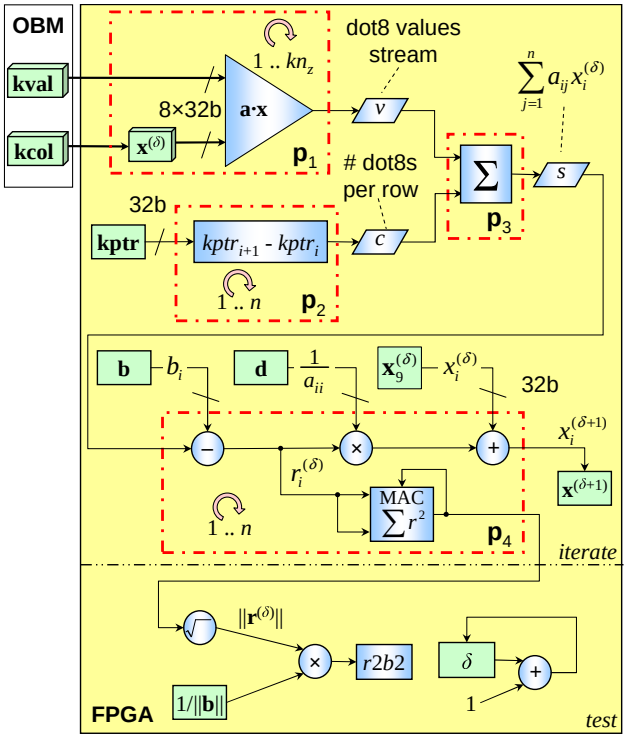


Fig. 24. Jacobi compute phase

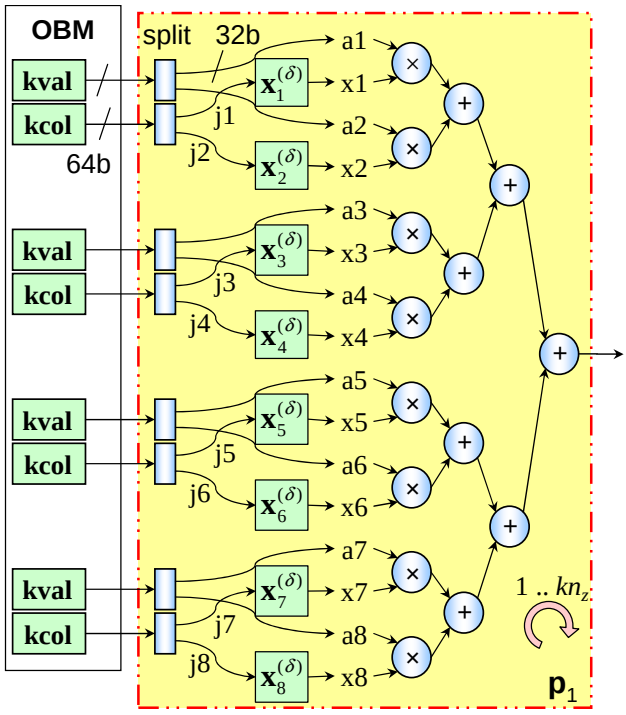


Fig. 25. Dot product unit

bit indices. The split **kval** values correspond to the A matrix values, and the split **kcol** indices are used to find the matching values from $\mathbf{x}^{(\delta)}$. These pairs are applied to the eight leaf-node multipliers of the fully pipelined binary tree dot product unit as detailed in Fig. 25. This latter diagram illustrates most clearly why multiple copies of $\mathbf{x}^{(\delta)}$ are needed. The dot product unit must read eight pairs of values on every clock cycle. Striping allows parallel access to the A values, but each $\mathbf{x}^{(\delta)}$ resides at a different memory location. Since the FPGA does not support multiport memory, the choices are 1) multiple copies of the vector, or 2) multiple clock cycles for each memory access. For performance purposes, option 1 was chosen. The output of the dot product unit is inserted into the values FIFO stream that will eventually be consumed by the streaming accumulator. It is important to note that the dot product unit does not need to deal with row boundaries; it simply emits a stream of partial dot products. As alluded to in App. V-B, the k-alignment associated with the marshaling process on the software side ensures each matrix row has an exact multiple of eight values (zero padding being used as necessary).

Counts stream: Parallel section $\mathbf{p}_2$ simply computes the number of dot8s per matrix row. This calculation is inserted into the counts FIFO stream that will eventually be consumed by the streaming accumulator.

Streaming accumulator: Parallel section $\mathbf{p}_3$ computes the full dot products, $\text{dotN}_i = \mathbf{a}_i \cdot \mathbf{x}^{(\delta)}$ associated with each matrix row and inserts them into the sums FIFO stream that will be consumed by parallel section $\mathbf{p}_4$.

Residual and next approximation: Parallel section $\mathbf{p}_4$ consumes the sums FIFO stream and uses it, in conjunction with the other inputs, to calculate the residual and next approximate solution vectors. Notice that it also uses a multiply-accumulate (MAC) unit to calculate $\sum r^2$ for the residual norm calculation.

Convergence test criteria: This section of hardware does not produce a meaningful value until the four parallel sections mentioned above have completed. It then computes the residual norm ratio $r2b2 = \|\mathbf{r}^{(\delta)}\|/\|\mathbf{b}\|$ and updates the number of iterations. These are needed to perform the termination test shown on line 39 of Fig. 21.

J. Results

1) Description of test problems: To create test matrices, Matgen [40] was used. This tool accepts an input file describing the features of the matrix (including diagonal dominance) and writes the output in Matrix Market coordinate format. Three sets of matrices were used for the tests. Each set consists of $8 \cdot 3 = 24$ matrices. There are eight matrix orders, $1,000, 2,000, \ldots, 8,000$. For each order, three matrices having sparsity percentages of approximately two, four, and six percent were generated, that is, $n_z \approx 0.02n^2$, $n_z \approx 0.04n^2$, and $n_z \approx 0.06n^2$, respectively. These are referred to as trial (1), trial (2), and trial (3), respectively. The $\mathbf{b}$ vector was generated as $\mathbf{b} = A\mathbf{x}_s$, where the solution vector, $\mathbf{x}_s$, consists of all 1,000s. The initial value of the solution vector, $\mathbf{x}^{(0)}$, consisted of all zeros. Since 32-bit floating-point data were used, the termination threshold was set at $\varepsilon = 5 \cdot 10^{-6}$. The maximum iteration count was set at $\delta_{\max} = 20,000$.

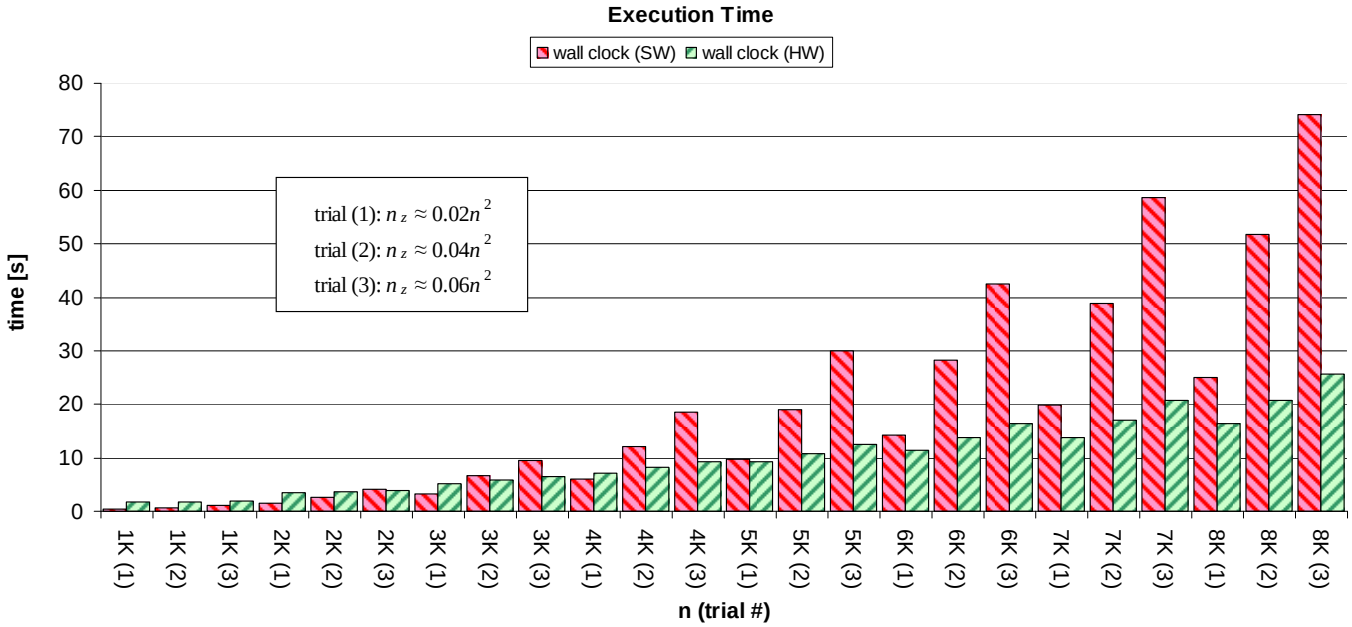


Fig. 26. Run time comparison

2) Test results: For all 72 software and 72 hardware test cases, Jacobi was run on an unloaded system and terminated with the expected solution, $\mathbf{x}^{(\delta)} \approx \mathbf{x}_s$. The average wall clock times for both Jacobi versions are shown in Fig. 26. The wall clock values are the average times for the three sets of 24 matrices. As expected, by virtue of the data set sizes, trial (1) takes less time than trial (2), which takes less time than trial (3). In addition, the hardware execution times for larger data sets are significantly lower than software execution times. The explanation for this phenomenon is simple. When the data size exceeds the cache limits of the GPP then the GPP performance suffers. See [24] for the supporting research. Notice that several of the hardware cases show a nearly threefold speedup over software.

VI. Conclusion

Mapping floating-point kernels on FPGA-based HPHCs via HLL-to-HDL compiler technology can still be a formidable task that relies mostly on the skill and intuition of individual developers. In this article we looked at some HPRC application design considerations including the three p's, expected overall speedup, expected resource utilization, control/memory intensive vs. compute intensive, monolithicity of modules, available bandwidth, opportunities for data reuse, algorithm design stability, algorithm efficiency, and memory access patterns. We also showed a simple example of how to interface HDL-based IP cores into an HLL-based design flow. By way of a more complete sparse matrix Jacobi iterative solver, we illustrated the challenging floating-point mapping process while simultaneously showing that such a mapping can result in a significant speedup compared with an equivalent software implementation. Our HPRC-based sparse matrix Jacobi iterative solver demonstrates a nearly threefold wall clock run time speedup when compared with a software implementation. If FPGA-based computational units are to be part of mainstream scientific computing, more research is necessary to simplify the mapping process. Ideas from research such as this should continue to be incorporated into tools to facilitate a more automated mapping process.

Acknowledgments

This work was supported in part by the DoD High Performance Computing Modernization Program under contract numbers W912HZ-(08-C-0073, 09-C-0108, and 10-C-0107), "High Performance Computational Design of Novel Materials," in part by Army Research Office HBCU/MSI grant number W911NF-07-1-0527, and in part by the U.S. Army Engineer Research and Development Center.

References

[1] Jane's Information Group, "Tactical reconnaissance and counter-concealment enabled radar (TRACER)(United States)," *Jane's Electronic Mission Aircraft*, 2011.
[2] M. Feldman, "JP Morgan buys into FPGA supercomputing," *HPCwire (www.hpcwire.com/hpcwire/2011-07-13)*, July 2011.
[3] G. Estrin, "Organization of computer systems – the fixed plus variable structure computer," in *Proceedings of the Western Joint Computer Conference*, San Francisco, CA, USA, May 1960, pp. 33 – 40.
[4] Xilinx, Inc., "Company history," *Funding Universe (www.fundinguniverse.com/company-histories)*, 2011.
[5] D. Fountain, "Algotronix: The first custom computer," *BYTE*, September 1991.
[6] Maxeler Technologies, "Maxeler Technologies hardware," *www.maxeler.com/content/hardware)*, 2011.
[7] SRC Computers, LLC, "General purpose reconfigurable computing systems," *www.srccomp.com/products/mapstationworkstations.asp*, 2010.
[8] M. B. Tellez, "System-level approach wins for UAV radar payload designs," *COTS Journal*, April 2011.

[9] Mercury Computer Systems, Inc., "Ensemble MXI-205 Xilinx V5 FPGA AMC Module," *www.mc.com/products/boards/ensemble_mxi205_xilinx*, 2011.
[10] ——, "Application ready subsystems," *www.mc.com/products/ars*, 2011.
[11] L. Zhuo and V. K. Prasanna, "High performance linear algebra operations on reconfigurable systems," in *Proceedings of the ACM/IEEE SuperComputing 2005 Conference*, Seattle, WA, USA, November 2005, pp. 2 – 13.
[12] ——, "Design tradeoffs for BLAS operations on reconfigurable hardware," in *Proceedings of the 34th International Conference on Parallel Processing*, Oslo, Norway, June 2005, pp. 78 – 86.
[13] Mentor Graphics, "DK Design Suite," *www.mentor.com/products/fpga/handel-c/dk-design-suite*, 2010.
[14] SRC Computers, LLC, "Carte programming environment," *www.srccomp.com/techpubs/carte.asp*, 2010.
[15] S. J. Park, "Reconfigurable computing for HPC computational science," in *Proceedings of the 2007 HPCMP User Group Conference*, Pittsburgh, PA, USA, June 2007, p. www.hpcmo.hpc.mil/UGC2007/UGC_2007_Agenda.pdf.
[16] SRC Computers, LLC, "SRC scalable Systems & Servers," *www.srccomp.com/products/scalable.asp*, 2011.
[17] D. McGrath, "Reconfigurable cluster computing installation could be a first," *COTS Journal*, April 2009.
[18] Panasas, "Parallel file system for HPC storage," *www.panasas.com*, 2011.
[19] A. R. Anderson, G. R. Morris, and K. H. Abed, "Achieving true parallelism on a high performance heterogeneous computer via a threaded programming model," in *Proceedings of the IEEE SoutheastCon 2011*, Nashville, TN, USA, March 2011, pp. 283 – 286.
[20] A. N. Malone, G. R. Morris, and K. H. Abed, "FPGA-based implementation of Horner's rule on a high performance heterogeneous computer," in *Proceedings of the IEEE SoutheastCon 2011*, Nashville, TN, USA, March 2011, pp. 277 – 282.
[21] N. S. Peay, G. R. Morris, and K. H. Abed, "Integrating Quartus Wizard-based VHDL floating-point components into a high performance heterogeneous computing environment," in *Proceedings of the IEEE SoutheastCon 2011*, Nashville, TN, USA, March 2011, pp. 413 – 417.
[22] G. R. Morris and K. H. Abed, "Mapping hierarchical multiple file VHDL kernels onto an SRC-7 high performance reconfigurable computer," in *Proceedings of the High Performance Computing Modernization Program Users Group Conference 2010*, Schaumburg, IL, USA, June 2010, pp. 524 – 533.
[23] G. R. Morris, R. Y. McGruder, and K. H. Abed, "Accelerating a sparse matrix iterative solver using a high performance reconfigurable computer," in *Proceedings of the High Performance Computing Modernization Program Users Group Conference 2010*, Schaumburg, IL, USA, June 2010, pp. 517 – 523.
[24] K. H. Abed and G. R. Morris, "Improving performance of codes with large/irregular stride memory access patterns via high performance reconfigurable computers," in *Proceedings of the High Performance Computing Modernization Program Users Group Conference 2009*, San Diego, CA, USA, June 2009, pp. 422 – 429.
[25] J. L. Rice, K. H. Abed, and G. R. Morris, "Design heuristics for mapping floating-point scientific computational kernels onto high performance reconfigurable computers," *Journal of Computers*, vol. 4, no. 6, pp. 542 – 553, June 2009.
[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufman, 2003.
[27] 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*, Singapore, December 2005, pp. 295 – 296.
[28] 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*, vol. 40, no. 3, pp. 50 – 57, March 2007.
[29] G. R. Morris and V. K. Prasanna, "Sparse matrix computations on reconfigurable hardware," *Computer*, vol. 40, no. 3, pp. 58 – 64, March 2007.
[30] K. Habgood and I. Arel, "A condensation-based application of cramer's rule for solving large-scale linear systems," *Journal of Discrete Algorithms*, vol. 10, pp. 98 – 109, January 2012.
[31] Altera Corporation, "Quartus II v9.1," www.altera.com.
[32] Mentor Graphics, "ModelSim-Altera Edition," www.altera.com.
[33] R. S. Varga, *Matrix Iterative Analysis, Second Edition*. Springer, 2009.
[34] E. Isaacson and H. B. Keller, *Analysis of Numerical Methods*. John Wiley & Sons, 1966.
[35] Y. Saad, "SPARSKIT: A basic tool-kit for sparse matrix computations (version 2)," *www-users.cs.umn.edu/~saad/software/SPARSKIT*, 2009.
[36] G. R. Morris, R. D. Anderson, and V. K. Prasanna, "An FPGA-based application-specific processor for efficient reduction of multiple variable-length floating-point data sets," in *Proceedings of the 17th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Steamboat Springs, CO, USA, September 2006, pp. 323 – 330.
[37] NIST, "Matrix Market," *math.nist.gov/MatrixMarket*, June 2004.
[38] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Transactions On Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377 – 1392, October 2007.
[39] G. R. Morris, L. Zhuo, and V. K. Prasanna, "High-performance FPGA-based general reduction methods," in *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, April 2005, pp. 323 – 324.
[40] Sourceforge, "Matgen," *www.matgen.sourceforge.net*, 2007.

Gerald R. Morris is a researcher at the U.S. Army Engineer Research and Development Center, Scientific Computing Research Center, Vicksburg, MS. He is also Adjunct Professor of Computer Engineering at Jackson State University, Jackson, MS, and Adjunct Professor of Computer Science at Mississippi State University, Mississippi State, MS. His research interests include high performance computing and mapping of algorithms onto alternative computational technologies. Morris received the B.S. in electrical engineering 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.

Khalid H. Abed is Professor of Computer Engineering at Jackson State University, Jackson, MS. His research interests include high performance heterogeneous computing, field programmable gate arrays, very large scale integrated circuit design, and digital signal processing. He received the B.S., M.S., and Ph.D. in electrical engineering from Wright State University. Abed has numerous 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.