

A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer

Gerald R. Morris and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA
{grm, prasanna}@usc.edu

Richard D. Anderson
Department of Computer Science
Jackson State University
Jackson, MS
richard.d.anderson@jsums.edu

Abstract

Supercomputer companies such as Cray, Silicon Graphics, and SRC Computers now offer reconfigurable computer (RC) systems that combine general-purpose processors (GPPs) with field-programmable gate arrays (FPGAs). The FPGAs can be programmed to become, in effect, application-specific processors. These exciting supercomputers allow end-users to create custom computing architectures aimed at the computationally intensive parts of each problem. This report describes a parameterized, parallelized, deeply pipelined, dual-FPGA, IEEE-754 64-bit floating-point design for accelerating the conjugate gradient (CG) iterative method on an FPGA-augmented RC. The FPGA-based elements are developed via a hybrid approach that uses a high-level language (HLL)-to-hardware description language (HDL) compiler in conjunction with custom-built, VHDL-based, floating-point components. A reference version of the design is implemented on a contemporary RC. Actual run time performance data compare the FPGA-augmented CG to the software-only version and show that the FPGA-based version runs 1.3 times faster than the software version. Estimates show that the design can achieve a 4 fold speedup on a next-generation RC.

1. Introduction

1.1. Background

The *conjugate gradient* (CG) method, which was discovered by Hestenes and Stiefel [8], is used to solve systems of equations, $Ax = b$. Naive implementations failed on large systems of equations and CG fell out of favor. However, Reid's work lead to renewed interest in the algorithm [15], and Krylov subspace methods such as CG were affirmed as one of the top 10 algorithms of the twentieth century [21].

The *reconfigurable computer* (RC), which was proposed by Estrin [6], is a "fixed plus variable structure" (F+VS) computer that can be "temporarily distorted into a problem oriented special purpose computer." The advent of the microprocessor forced the RC into relative obscurity. However, the FPGA has precipitated a reawakening, and companies like Cray, Silicon Graphics, and SRC Computers (SRC) now offer RCs that use GPPs and FPGAs as the F+VS structure [3, 17, 19].

1.2 FPGA-based computational kernels

FPGA-based floating-point computational kernels targeting specific problem domains such as molecular dynamics have been demonstrated [10, 22, 16], as have general-purpose floating-point scientific kernels such as linear algebra routines [5, 26, 12, 4]. Underwood argues that by 2009, FPGAs will have an order of magnitude peak floating-point performance over GPPs [20]. These efforts have set the stage for research into floating-point scientific computing on modern RCs.

1.3. Scientific computing on RCs

For floating-point scientific computing, modern RCs are still immature. For example, they do not have enough local memory banks to support highly parallel computations. Several companies such as Celoxica, Mitronics, and SRC Computers, have HLL-to-HDL compiler technology that allows FPGA-based development using high-level languages [2, 11, 18]. These products do a good job with integer and fixed-point applications such as digital signal processing. For floating-point applications, however, current RCs and HLL-to-HDL compilers make programmer access to the FPGAs range from moderately difficult to nearly impossible. Clearly, additional research on the mapping of floating-point scientific kernels onto RCs is necessary.

1.4. Overview of this paper

This research effort employs a hybrid development approach wherein an HLL-to-HDL compiler is used in conjunction with custom-built, VHDL-based, floating-point intellectual property (IP) cores to map an IEEE-754 64-bit floating-point CG design onto an FPGA-augmented RC. Section 2 provides a brief overview of CG, discusses some of the desirable features of a “good” implementation, and identifies the software that was selected for this research. Section 3 provides a brief overview of the hybrid development process. Section 4 presents high-level designs for the software-only and FPGA-augmented CG versions. Section 5 details the FPGA-based portion of the design. Section 6 gives a brief description of the RC hardware used during this investigation and describes the implementation. Section 7 compares the actual run time performance of the two CG versions, and estimates the performance on a next generation RC. Section 8 presents the conclusions.

2. Conjugate gradient

2.1. Overview of CG

CG is probably the most widely known iterative method for solving linear equations, $A\mathbf{x} = \mathbf{b}$, whenever A is a real, square, symmetric, positive-definite (RSSPD), sparse matrix. CG-related literature resources are ubiquitous—any text dealing with sparse linear equations would include a discussion of CG, e.g., [24, 1]. A Google™ search for “conjugate gradient” resulted in over 1.3 million hits, and an Amazon.com search yielded 65 books having “conjugate gradient” in the title. Therefore, this section is not a rigorous mathematical treatment of CG; however, it gives enough of the intuition such that the algorithm can be understood. Fundamentally, $CG(f(\mathbf{x}), \mathbf{x}_0)$, is an iterative algorithm for finding the nearest local minimum of function, $f(\mathbf{x})$, where \mathbf{x} is an n -vector of independent variables, and \mathbf{x}_0 is a starting point. If one takes the quadratic form of vector \mathbf{x} ,

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T \mathbf{x} + c, \quad (1)$$

and matrix A is an RSSPD matrix, it can be shown via “a little bit of tedious math” that $f(\mathbf{x})$ is minimized by the solution to $A\mathbf{x} = \mathbf{b}$ [9]. A plot of $f(\mathbf{x})$, when A is a 2×2 RSSPD matrix, yields a bowl-shaped parabolic surface. The \mathbf{x} value at the lowest point in this bowl is the solution to $A\mathbf{x} = \mathbf{b}$. It is also the local and global minimum, which can therefore be found using CG. For matrices having $n > 2$, an $(n + 1)$ -dimensional parabolic surface corresponding to $f(\mathbf{x})$ can be imagined. Intuitively, one can “walk downhill to find the lowest point.” Unlike the method of steepest descent, which uses the local gradient for going downhill,

and can therefore end up taking multiple steps in the same direction, CG uses A -orthogonal (conjugate) search directions that facilitate more rapid convergence.

The algorithm shown in Figure 1 is a synthesis of those given in [9, 24, 1, 8]. In the absence of ill-conditioned ma-

```

1:   $\mathbf{x} = \mathbf{x}_0$            ! approximate solution
2:   $\mathbf{r} = \mathbf{b} - A\mathbf{x}$       ! residual
3:   $\mathbf{p} = \mathbf{r}$            ! search direction
4:   $\delta_n = \delta_c = \mathbf{r}^T \mathbf{r}$  ! scratch pad
5:  while ( $\Delta$  is too big)
6:     $\mathbf{q} = A\mathbf{p}$            ! need  $\mathbf{q}$ 
7:     $\alpha = \delta_n / \mathbf{p}^T \mathbf{q}$  ! and  $\alpha$  again
8:     $\mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$     ! new solution
9:     $\mathbf{r} = \mathbf{r} - \alpha \mathbf{q}$     ! new residual
10:    $\delta_c = \delta_n$         ! keep current and
11:    $\delta_n = \mathbf{r}^T \mathbf{r}$     ! new scratch values
12:    $\mathbf{p} = \mathbf{r} + \frac{\delta_n}{\delta_c} \mathbf{p}$  ! new direction

```

Figure 1. Conjugate gradient algorithm

trices, roundoff error, etc., each loop iteration yields a better \mathbf{x} by “walking” in the direction derived from A and \mathbf{p} . CG typically uses some distance measure to test for convergence, hence the terminology “ Δ is too big” on line 5.

2.2. Features of a good CG implementation

Clearly, a “good” implementation has to be both correct and efficient. As noted in Section 1.1, CG initially fell out of favor because of naive implementations. Furthermore, CG should allow the user to specify convergence criteria such as type of convergence test, tolerances, and maximum number of iterations. A good CG implementation should also allow for preconditioning and residual correction. Concerning lines 2 and 6 of the CG algorithm, a) most of the work takes place in matrix-vector multiply, b) CG does not actually need the matrix, only the matrix-vector product, and c) the matrix is invariant across iterations. Therefore, in addition to the features already mentioned, a good implementation should a) use a highly optimized matrix-vector multiply, b) insulate itself from having to know about the matrix by using callbacks, reverse-communication, or some similar technique, and c) allow matrix-vector multiply to save a copy of matrix A (if reading the matrix is costly).

2.3. Selected software CG

The best course of action for this research effort was to find an existing CG implementation suitable for high-performance use. This approach has, as British philosopher and mathematician, Bertrand Russell, once wrote, “all the advantages of theft over honest toil.” Saad’s SPARSKIT li-

library [25], which is used to implement sparse matrix scientific applications, includes a CG module that exhibits many of the features mentioned in Section 2.2. For example, the SPARSKIT reverse communication approach makes it very easy to add a preconditioner without modifying the design. This reverse communication mechanism also makes it easy to substitute a more efficient matrix-vector multiply routine without impacting the design. SPARSKIT is already written, debugged, and being used in actual scientific applications. Therefore, SPARSKIT was selected as the software base from which the CG designs in this investigation were formulated.

3. Hybrid development process

Several vendors now offer HLL-to-HDL compilers to improve programmer productivity. Since traditional HLLs do not have mechanisms for expressing parallelism, HLL-to-HDL compiler vendors have taken one of three approaches, 1) extend an existing HLL – as with Celoxica’s *Handel-C*, 2) create a new HLL – as with Mitronics’ *Mitrion-C*, or 3) use a standard HLL but include pragmas to guide the compiler – as with SRC’s *Carte* compiler. Independent of the mechanism, the goal is deeply pipelined, highly parallelized hardware. Therefore, vendors also provide features such as pipelined loops, communication channels, synchronization primitives, and application programmer interface (API) calls to access vendor-supplied IP cores.

During a typical RC development effort, one partitions the design into *software modules*, which are targeted for execution on GPPs, and *hardware modules*, which are targeted for execution on FPGAs. Software modules that call hardware modules include some vendor-specific calls to control/use the FPGA, e.g., the Cray XD1 *fpga.load* call reconfigures the FPGA.

Some HLL-to-HDL development environments such as Celoxica’s DK Design Suite (DK) support a hybrid development approach. This hybrid approach allows the developer to use a hardware description language such as VHDL to create customized IP cores and import them into the HLL environment. As a result, the developer can use all the vendor HLL features such as parallel code blocks, pipelined loops, and channels, yet still have HLL access to the customized IP cores. Figure 2 illustrates the hybrid approach. The software modules are compiled with the normal software compiler to produce object files. The hardware module HLL code is compiled with the HLL-to-HDL compiler to produce input to the synthesis module. The user HDL code is also given as input to the synthesis module. Vendors provide some mechanism allowing the HLL-to-HDL compiler and/or linker to obtain visibility into the user’s IP cores, e.g., DK’s *interface declaration* or *Carte*’s *info* file.

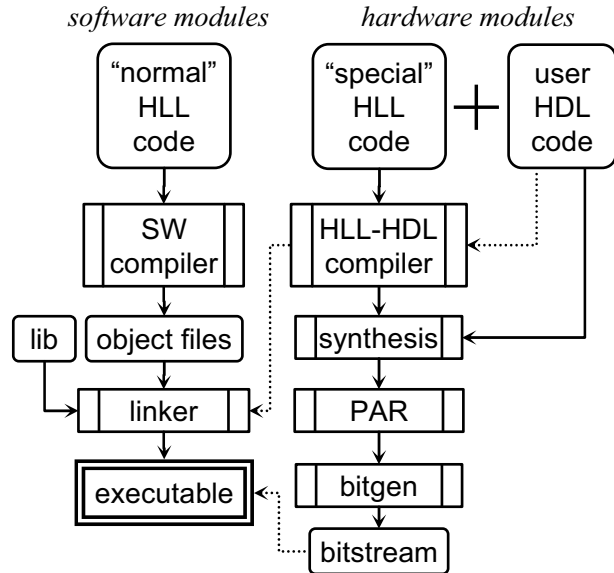


Figure 2. Hybrid development process

The netlists produced by synthesis are fed into the place and route (PAR) module. PAR output feeds bitgen to produce the FPGA logic configuration bitstream. The linker ingests the object files, library files, IP core interface information, and produces the executable.

4. High-level CG designs

4.1. Compressed sparse row format

Before presenting the CG designs, a brief description of compressed sparse row (CSR) format is in order. CSR format is often used when a matrix has a relatively small number of non-zero values, $n_z \ll nm$, where n_z is the number of non-zeros, n is the number of rows, and m is the number of columns. CSR employs three vectors, **val**, **col**, and **ptr**, to store only the non-zero values and identify the row and column indices.

- val** The non-zero values as the matrix is traversed row-wise. Vector **val** is of length, n_z .
- col** The column index of each non-zero value. Thus, if $val(k) = a_{ij}$ then $col(k) = j$. Vector **col** is of length, n_z .
- ptr** The location in **val** where each matrix row starts, i.e., the first element of row i is $val(ptr(i))$. The range of index values for row i is $ptr(i) \leq j < ptr(i + 1)$, i.e., row i contains $len_i = ptr(i + 1) - ptr(i)$ non-zero values. To keep the len calculations consistent, $ptr(n + 1) = n_z + 1$. Vector **ptr** is of length, $n + 1$.

An example depicting the CSR format for a 4×4 sparse matrix is shown in Figure 3.

$$A = \begin{bmatrix} 5 & 0 & 0 & 2 \\ 0 & 8 & 0 & 0 \\ 1 & 0 & 6 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \quad \begin{array}{l} n=4 \\ \mapsto \\ n_z=6 \end{array}$$

<i>i</i>	1	2	3	4	5	6
val	5	2	8	1	6	3
col	1	4	2	1	3	4
ptr	1	3	4	6	7	
<i>len</i>	2	1	2	1		

$n_z + 1$

Figure 3. CSR format

4.2. Software-only CG design

The software-only CG shown in Figure 4 consists of three major components; a **main** routine, some MatrixMarket elements [14], and the SPARSKIT library. In an actual operational scenario, the MatrixMarket elements, **coocsr** routine, and the part of **main** on the left side of Figure 4 would correspond to the producer of the linear equations being solved, e.g., the simulation code. The rest of the **main** routine, and the **cg**, **amux**, and **ddot** routines on the right side of the figure would correspond to the solver itself.

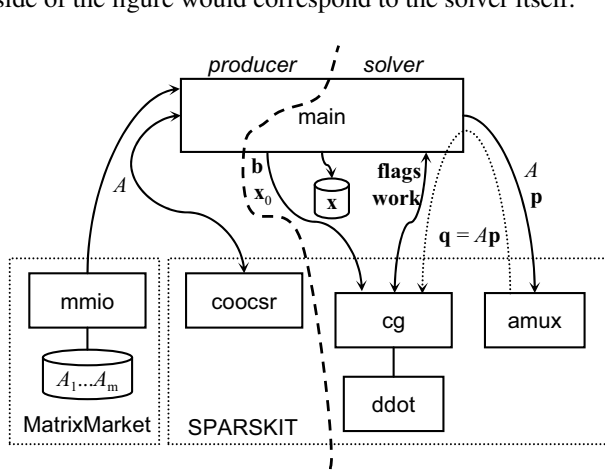


Figure 4. Software-only CG design

The **main** routine is a simple driver program, which essentially measures how long it takes for CG to solve a set of linear equations. The MatrixMarket portion consists of several RSSPD sparse matrices ($A_1 \dots A_m$) stored in MatrixMarket coordinate format, and the associated **mmio** input-output routines. The SPARSKIT **coocsr** routine converts the coordinate format into CSR format. The **cg** routine is a reverse-communication-enabled implementation of the CG iterative method. The **ddot** and **amux** routines are dot product and sparse matrix-vector multiply.

At run time, **main** reads in the matrix, converts it to CSR format, and uses a known x vector to generate b . It

then initializes a set of **flags**, some **work** space, and invokes the **cg** routine sending b , starting point x_0 , **flags**, and **work**; matrix A is not passed to **cg**. Whenever **cg** needs a matrix-vector product, it sets the appropriate request value in **flags**, uses **work** to communicate the p vector back to the **main** routine, and returns. The **main** routine loop examines the **flags** to determine the course of action (*done*, *matrix-vector multiply*, or *error*). If **cg** needs a matrix-vector product, **main** sends matrix A and vector p to the **amux** sparse matrix-vector multiply routine. When **amux** returns the $q = Ap$ matrix-vector product, **main** uses **work** to communicate q back to **cg**. The **main** routine sets the appropriate reentry indicator in **flags** and calls **cg**, which continues with the iteration. When **cg** is done, it sets the *done* value in **flags**, uses **work** to communicate x back to the **main** routine, and returns. The **main** routine writes the solution to a text file, displays the input matrix name, number of iterations, wall-clock run time, and then terminates. Since a known x vector was used to generate b , verification of the solution is trivial.

4.3. FPGA-augmented CG design

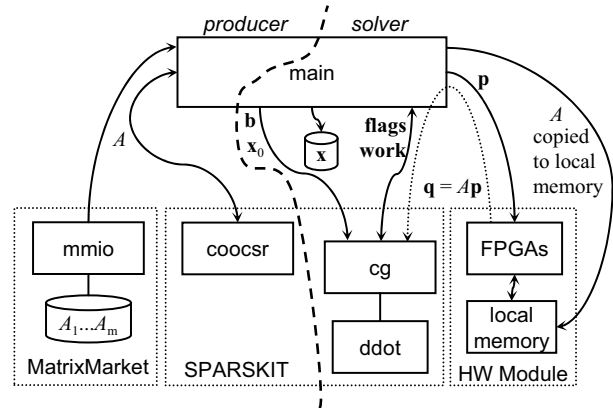


Figure 5. FPGA-augmented CG

The FPGA-augmented CG is shown in Figure 5. It consists of four major components; the **main** routine, the MatrixMarket elements, the SPARSKIT library, and a hardware module. The software components are as described in Section 4.2. The hardware module implements a specialized sparse matrix-vector multiply as a replacement for the **amux** routine.

The run time operation is virtually identical to the software-only design but with one important distinction. Since the A matrix is invariant during the entire CG calculation, the hardware module pulls a copy of matrix A exactly one time, stores it in its local memory and uses it for subsequent iterations. Amortization of the A matrix transfer cost across all iterations of CG is a key design feature.

5. Hardware module design

5.1. Module overview

A profile of the software-only version of CG showed that it spent over 95% of the execution time in sparse matrix-vector multiply. As noted in [4], the other operations “have little impact on performance.” Thus, the hardware module is a specialized sparse matrix-vector multiply circuit. The

Table 1. Design parameters

Parameter	Description
k	dot product data path width
α_v	vendor adder loop latency
α_m	multiplier IP core latency
α_a	adder IP core latency
n_{max}	max number of matrix rows
$n_{z_{max}}$	max number of non-zeros

parameters shown in Table 1 specify the design characteristics. As shown in Figure 6, the design consists of two pipelined cooperating FPGAs, **F1** and **F2**; and a set of local memory banks to hold the **val**, **col**, and **jptr** vectors. The **F1** configuration includes a VHDL-based, $k \times k$ dot product IP core; on-chip arrays to store the **p** and **ptr** vectors; an output channel to send the dot products over to **F2**; an input channel to receive the q_i products from **F2**; an output direct memory access (DMA) channel to send the q_i back to GPP memory; and some control circuitry. The **F2** configuration includes a partial summation unit; an n -row by α_v -column partial summation array, S ; an input channel to receive the dot products from **F1**; an output channel to send the q_i products back to **F1**; a VHDL-based, α_v -input accumulator IP core; and a controller.

5.2. VHDL-based IP cores

To meet timing constraints, simplify the hardware module code, and reduce compilation time, pipelined, VHDL-based IP cores were developed using the 64-bit IEEE floating-point adder and multiplier IP cores described in [7].

The *dot product* core accepts two 64-bit floating-point k -vectors every clock cycle. After the pipeline is filled, it emits one 64-bit floating-point dot product every clock cycle. There are $\lg k$ non-leaf levels in a full binary tree having k leaf nodes. As noted in Table 1, α_m and α_a are the latencies of the floating point IP cores used in this design, so the latency of the dot product core is $\alpha_m + \alpha_a \lg k$ clock cycles. Figure 7(a) shows an example for $k = 4$.

The *accumulator* core accepts α_v 64-bit floating-point values every clock cycle. After the pipeline is filled, it produces one 64-bit floating-point sum every clock cycle. In

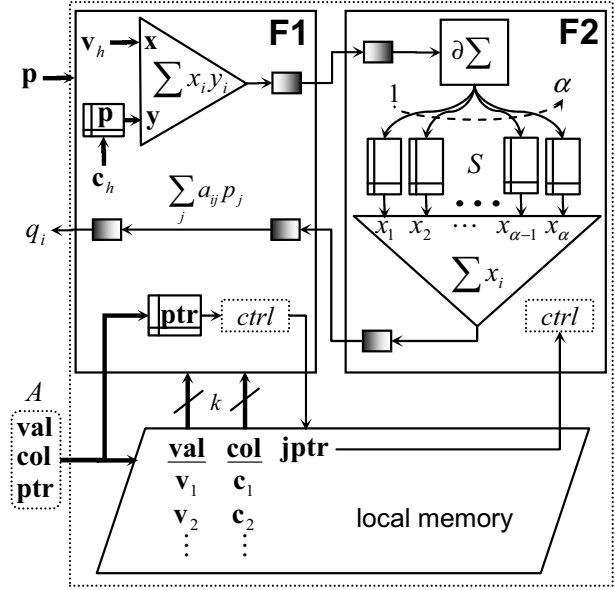


Figure 6. HW module block diagram

the general case, α_v may not be a power of two, and delay units are needed to synchronize the pipeline stages. There are $\lceil \lg \alpha_v \rceil$ non-leaf levels in a binary tree having α_v leaves, and the adder IP latency is α_a , so the latency is $\alpha_a (\lceil \lg \alpha_v \rceil)$ cycles. Figure 7(b) shows an example for $\alpha_v = 6$.

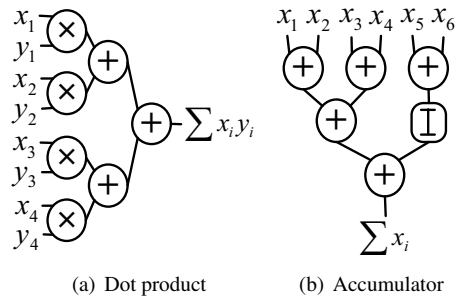


Figure 7. VHDL-based IP cores

5.3. Partial summation unit

A simple loop, as in Figure 8, to accumulate the serially delivered dot products is inefficient because of the pipelined floating-point units. A solution is to use the partial summation unit shown in Figure 6. The idea is to have an n -row by α_v -column partial summation array, S , and a meta-data array, **jptr**, which associates an index with each dot product. The j^{th} incoming dot product “belongs” to $q(jptr(j))$ and is added to $S(jptr(j), j \bmod \alpha_v)$. This scheduling approach guarantees an α_v -cycle interval between subsequent

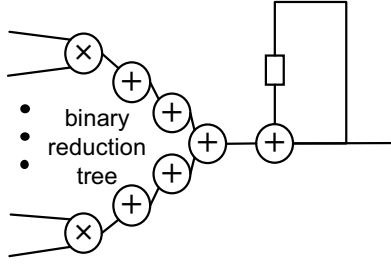


Figure 8. Naive adder loop

writes to the same memory location and allows a pipelined loop. Summation

$$q_i = \sum_{j=1}^{j=\alpha_v} S(i, j)$$

is the i^{th} element of the output vector.

5.4. Hardware module operation

The hardware module has four operational sequences; startup, input, execute, and output as described herein.

During the *startup sequence*, which occurs once per CG run, the CSR format sparse matrix A is DMAed from GPP memory and stored in the hardware module. The **val** and **col** vectors are stored in the local memory banks, and **ptr** is stored in block RAM (BRAM) memory on **F1**. To allow the dot product unit to fetch k values from **val** and k values from **col** every clock cycle, the vectors are striped across the local memory banks. The number of simultaneous local memory bank reads, and the width of the banks, are a constraint that the developer must consider when designing RC-based applications. The SRC-6, for example, only allows six simultaneous 64-bit reads. To conserve memory bank resources in this design, multiple **col** values are packed into each local memory bank. In Figure 9, for example, four elements of **col** are packed into a single 64-bit local memory word, and four additional local memory banks are used for the **val** vector. To simplify routing, and allow for higher clock rates, each **val** memory bank is attached to exactly one leaf node in the dot product unit. Thus, zeros are padded at the end of each matrix row that does not have a multiple of k values. This process is called k -alignment. The idea, for $k = 4$, is shown in Figure 9.

During the *input sequence*, which happens once per CG iteration, **p** is stored in BRAM memories in **F1**. Since the dot product unit must fetch k different values each clock cycle, k copies of **p** are created in parallel, one for each y_i input of the dot product unit, as shown in Figure 9. As noted in Section 1.3, there are simply not enough local memory banks to support highly parallel execution. The decision in

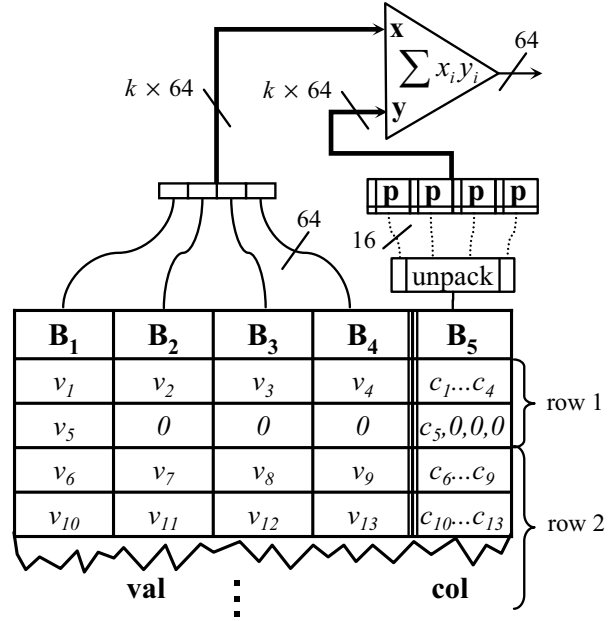


Figure 9. Local memory banks

this implementation to create k copies of **p** is admittedly not scalable; in future RCs, one could map these copies of **p** onto local memories, which have megawords of storage.

At the beginning of the *execute sequence*, the first k elements of **val**, $v_1 \dots v_k$, are placed in registers at the **x** input lines of the dot product unit. Simultaneously, the corresponding k -elements of **col**, $c_1 \dots c_k$, are used as addresses into the **p** memories, as shown in Figure 9. This causes the appropriately paired a_{ij} and p_j values to enter the dot product unit at the same time. On the next clock cycle, the second k -group is processed, and so forth. The dot products are sent to **F2**, where the partial summation unit does a partial summation into the S array.

During the *output sequence*, which occurs once per iteration, the accumulator unit in **F2** sums each S row to produce the q_i , which are sent to **F1** and then DMAed back to GPP memory.

6. Implementation

6.1. Description of target RC

From a practical perspective, SRC's Carte programming environment is more tightly integrated than Cray's or SGI's. Carte automatically handles inter-chip communication and other necessary (but uninteresting) details. Thus, the developer can concentrate on mapping the algorithm onto hardware. Carte also directly supports the hybrid development model. Thus, an SRC-6 Model T210-0 MAPStation running Carte v2.1 was used as the target RC. The SRC-6 has

two 2.8GHz Intel Xeon processors with 512KB cache and 1GB RAM as the fixed structure. The variable *MAP Series MPC* processor contains two Xilinx Virtex II 6000 FPGAs (xc2v6000) running at 100MHz. Each of these FPGAs has 144, 2KB BRAMs and 144 *mult18x18* integer multipliers. There are six banks of *on-board memory* (OBM) associated with the FPGAs. Each bank has a 64-bit word width and a depth of nearly 0.5M words. With six banks, the FPGAs can read or write up to 48 OBM bytes per clock cycle. The MAP is connected to the Xeon motherboard through a memory interface, so DMA is used to move data between the OBM and GPP memory. The MAP has a streaming DMA capability, and an inter/intra-FPGA streaming capability that allows computation to overlap communication and facilitates parallelism.

6.2. Partial summation unit: postscript

Before describing the implementation, a few words about the partial summation unit specified in Section 5.3 are needed. In Carte parlance, a simple floating-point adder loop creates a “loop-carried dependence” on the accumulation variable. For the implementation at hand, this corresponds to a 14 cycle/iteration penalty, which is intolerable. The SRC floating-point accumulator macro, *fp_accum_64* does not allow a reset/read except when entering/exiting a loop. Since the application needs to accumulate one set of dot products per matrix row, a nested loop is needed. However, Carte flushes inner loop pipelines, which significantly reduces the performance of this application. Finally, the VHDL-based accumulator IP cores described in [13] will not work in pipelined loops since the latency depends upon the number of elements in the input stream. In short, existing accumulation solutions fail for this application. Hence, the partial summation unit is needed.

6.3. Implementation summary

The two CG designs in Section 4.2, and Section 4.3 were coded for the SRC-6 platform. The same set of files were used in both implementations to ensure a valid side-by-side comparison. The hardware module is not needed for the software-only version, and the **amux** routine is not needed for the FPGA-augmented version; the **main** routine was altered slightly to accommodate both versions.

The bulk of the effort was in implementing the hardware module, which used the parameters shown in Table 2. The k value is constrained by the six local memory banks in the MAP. Since dot product fetches k values from **val** and k values from **col** every clock cycle, $k = 4$ is the widest data path that will fit on the target platform. The α_v value is based on the “clocks per iteration” inner loop summary of Carte. The α_m and α_a latency values are documented

Table 2. Implementation parameters

Parameter	Value
k	4
α_v	14
α_m	10
α_a	14
n_{max}	2,048
nz_{max}	262,144

in [7]. The n_{max} value is constrained by the 144 BRAMs used to implement the partial summation array on **F2**. An $n_{max} \times \alpha_v = 2,048 \times 14$ array of 64-bit words requires $8 \times 14 = 112$ BRAMs, so $n_{max} = 2,048$ is the largest matrix order that can be processed on the target RC. The nz_{max} value is constrained by the **ptr** values. Since the 16-bit unsigned integer, ptr_i , is the index of the i^{th} k -group in the k -aligned **val** and **col** arrays, then 2^{16} is the number of possible k -groups, and $nz_{max} = 2^{16} \times k = 262,144$ is the largest number of non-zero values that can be processed on the MAP Series MPC. After several false starts, and with a considerable amount of effort in the code and test phase, a reference implementation was produced. The Carte compiler was able to pipeline every loop, and meet both timing and areas constraints. Table 3 shows post PAR statistics for both FPGAs.

Table 3. Post PAR statistics

	F1	F2
slices	14,335	20,602
mult18x18	95	0
BRAM	36	112
clock	9.993ns	9.996ns

7. Results

7.1. Description of test matrices

To compare the two CG versions under a number of different conditions, the nine RSSPD sparse matrices in Table 4 were used as test inputs to both CG versions. The three trial 1 matrices for $n \in (1000, 1500, 2000)$ each contain approximately 12,000 non-zero elements. Each trial 1 data set can fit in the 512KB cache of the Xeon processor. The three trial 2 matrices contain approximately 55,000 non-zero elements (slightly larger than the 512KB cache). Trial 3 matrices contain approximately 250,000 non-zero elements (about five times larger than the Xeon cache). These test matrices are relatively dense and have an order, n , which is not that large. However, the resource constraints of current

RCs limited the sizes that were possible, and it was obvious that an RC could only compete if the data would not fit in GPP cache. Therefore, the decision was made to use matrix sizes that would fit on the FPGA and not fit in GPP cache. As newer RCs become available, one can expect larger matrix sizes to be possible.

Table 4. Test matrices

Trial	KB _{max}	n _z		
		n=1000	n=1500	n=2000
1	166	12,528	12,418	13,834
2	572	54,694	51,832	55,386
3	2,512	253,274	251,462	254,066

A brief description of the matrix-generation algorithm, and proof that it is correct, is in order. Given a real, non-singular, lower triangular $n \times n$ matrix, L , a new matrix, $A = LL^T$, is generated. The following proof shows that A is an RSSPD matrix as required by the CG algorithm. That A is real and square (RS) follows trivially from $A = LL^T$. Symmetry (S) requires $A = A^T$, i.e., $LL^T = (LL^T)^T$. For any two matrices, M_1 and M_2 ,

$$(M_1 M_2)^T = M_2^T M_1^T, \quad (2)$$

so $(LL^T)^T = (L^T)^T L^T = LL^T$, and A is symmetric. A is positive definite (PD) if, $\forall \mathbf{x} \neq 0 \mid \mathbf{x}^T A \mathbf{x} > 0$. The proof by contradiction starts by assuming $A = LL^T$ is *not* positive-definite, i.e.,

$$\exists \mathbf{x} \neq 0 \mid \mathbf{x}^T LL^T \mathbf{x} \leq 0. \quad (3)$$

Using Equation 2, and letting $\mathbf{v} = L^T \mathbf{x}$, yields,

$$[\mathbf{x}^T L] L^T \mathbf{x} = [(L^T \mathbf{x})^T] L^T \mathbf{x} = \mathbf{v}^T \mathbf{v} \leq 0. \quad (4)$$

However,

$$\mathbf{v}^T \mathbf{v} = (\|\mathbf{v}\|_2)^2, \quad (5)$$

where $\|\mathbf{v}\|_2 = \sqrt{\sum_i v_i^2}$ is the 2-norm. Equation 4 contradicts Equation 5, since $(\|\mathbf{v}\|_2)^2 > 0$. Therefore, Equation 3 is false, and $A = LL^T$ is positive-definite.

7.2. Test conditions

The \mathbf{b} vector is generated as $\mathbf{b} = A\mathbf{x}_h$, where \mathbf{x}_h is an n -vector consisting of all 100's. The initial guess, \mathbf{x}_0 , is an n -vector consisting of all 0's. The convergence test used is given by,

$$\frac{\|\mathbf{r}\|_2}{\|\mathbf{b} - A\mathbf{x}_0\|_2} \leq 10^{-9}$$

where $\|\mathbf{r}\|_2$ is the 2-norm of the residual, and $\|\mathbf{b} - A\mathbf{x}_0\|_2$ is the 2-norm of the initial residual, as shown on line 2 of Figure 1. In all 18 cases, CG was run on an unloaded system, terminated normally, and had a solution vector, $\mathbf{x} = \mathbf{x}_h$.

7.3. Test results

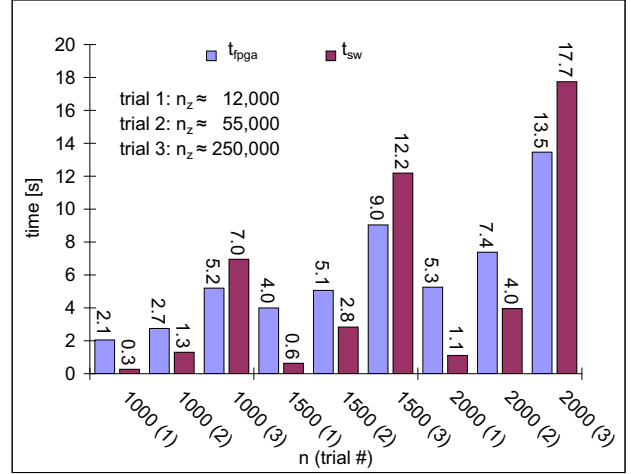


Figure 10. Run time comparison

Figure 10 shows the wall-clock run time of the FPGA-augmented version and the software-only version of CG. For the trial 1 and trial 2 test cases, which fit (or nearly fit) in the 512KB cache of the Xeon processor, the FPGA-augmented CG run time is higher than the run time of the software-only version. However, for trial 3 test cases, which do not fit in Xeon cache, the FPGA-augmented CG run time is lower than the run time of the software-only version. Trial 3 demonstrates a speedup greater than 1.3 for all three matrix sizes. Table 5 shows the iterations and run times from trial 3. In large scientific simulations, where CG is run thou-

Table 5. Iterations and run times

n	n _z	iter	t _{sw}	t _{fpga}
1,000	253,274	4,348	6.96	5.20
1,500	251,462	7,770	12.19	9.04
2,000	254,066	10,968	17.74	13.46

sands of times, saving 4 seconds per CG call can significantly reduce overall simulation time.

7.4. SRC-7 performance estimates

The 1.3 speedup indicates that the FPGA-based design is sound. In this section, the expected performance of this design running on the soon-to-be-released SRC-7 RC is conservatively estimated. Table 6 contains some of the specifications for the MAP Series MPH/RL processor used in the SRC-7 [19, 23]. In this CG design, n_{max} is limited by available BRAM on **F2**, and k is limited by the number of simultaneous OBM reads. In the MPH/RL processor, $k = 8$ is possible since there are 20 available simultaneous OBM

Table 6. MAP processor specifications

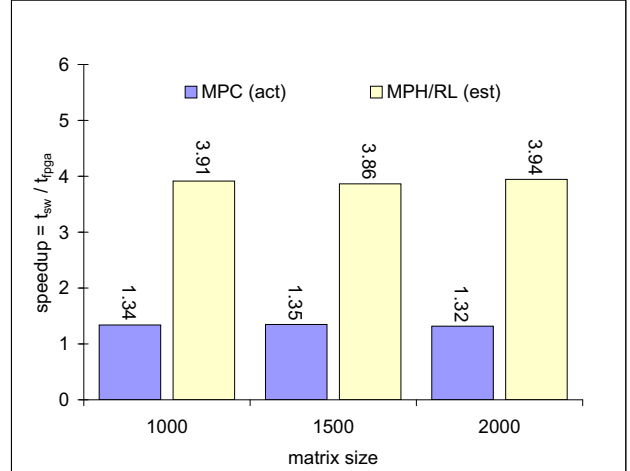
	MAP Series	
	MPC	MPH/RL
FPGA	xc2v6000	xc2vp100
Slices	33,792	44,096
Clock	100MHz	150MHz
BRAMs	144	444
mult18x18	144	444
simultaneous OBM reads	6	20
sustainable payload BW	2.8GB/s	14.4GB/s

reads, not all of which are being used (8 for `val`, 2 for `col`, and 1 for `jptr`). To estimate run time for an MPH/RL-based RC, simple scaling will be used. As shown in Table 6, the MAP Series MPH/RL runs at 150MHz. The $k = 8$ design parameter allows the dot product unit to ingest twice as many values per clock cycle. This means that the MPH/RL matrix-vector multiply executes $2 \times 1.5 = 3$ times faster than in the MPC. The overall run time consists of time spent in matrix-vector multiply, and time spent elsewhere. To determine percentage of time spent in matrix-vector multiply, the *oprofile* package was used to profile the software-only version of CG. The profile report showed that 97.5% of the time was spent in the `amux` routine. Thus, the MPH/RL-based CG runs $3 \times 0.975 + (1 - 0.975) = 2.95$ times faster than the MPC-based CG. A comparison of Table 3 and Table 6 shows that there are sufficient mult18x18's to build the additional four floating-point multipliers. Table 6 also shows there is enough BW to handle the increased throughput. The MPC speedups shown in Figure 11 are based on the actual run times shown in Figure 10, and the MPH/RL speedups are based on the estimation technique just presented.

From Table 3 and Table 2, for $n_{max} = 2,048$, a total of 112 BRAMs were used. In Carte, hardware arrays are allocated in lengths that are a power of 2, e.g., *double b[1000]*, actually causes a 1024×64 -bit group of BRAM blocks to be used. Therefore “legal” values for n_{max} are also powers of 2. Thus, $n_{max} = 4,096$, which requires $2 \times 112 = 224$ BRAMs, is the largest size that will fit on the MPH/RL processor. Clearly, the SRC-7 will not only be faster, but it will also be able to handle larger matrices.

8. Conclusion

This research has shown that implementing sophisticated double-precision floating-point kernels on FPGA-augmented RCs can result in higher performance when compared to a software-only implementation. The neces-

**Figure 11. Actual and estimated speedup**

sary use of IP cores and a hybrid development process to meet timing constraints confirms the authors’ view that “For floating-point applications, however, current RCs and HLL-to-HDL compilers make programmer access to the FPGAs range from moderately difficult to nearly impossible.” Actual run times for an FPGA-augmented version of CG are 1.3 times faster than the software-only version when the input data sets are too large to fit in the cache of the Xeon GPP. Conservatively estimated run times show a 4 fold speedup on the soon-to-be-released SRC-7.

Acknowledgments

This work was supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763. This work was also supported in part by the Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP). The authors thank HPCMP computational scientist Tom Oppe for his assistance with the theoretical work; SRC Computers’ staff members Dan Poznanovic, et al, for their technical insight; and colleague Ron Scrofano for his myriad contributions.

References

- [1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [2] Celoxica Ltd. DK Design Suite. <http://www.celoxica.com>.

- [3] Cray Inc. Cray XD1. <http://www.cray.com>.
- [4] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *FPGA'05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 75–85, Monterey, CA, February 2005.
- [5] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *FPGA'05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 86–95, Monterey, CA, February 2005.
- [6] 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.
- [7] 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 (ERSA'05)*, Las Vegas, NV, June 2005.
- [8] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.
- [9] J. R. Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, Edition 1 $\frac{1}{4}$. Unpublished draft, <http://www.cs.cmu.edu/~jrs/>.
- [10] G. Lienhart, A. Kugel, and R. Manner. Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulations. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 182–191, Napa, CA, April 2002.
- [11] Mitrionics Inc. Mitrion-C. <http://www.mitrionics.com>.
- [12] G. R. Morris and V. K. Prasanna. An FPGA-based floating-point Jacobi iterative solver. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'05)*, pages 420–427, Las Vegas, NV, December 2005.
- [13] 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 (FCCM'05)*, Napa, CA, April 2005.
- [14] NIST. Matrix Market. <http://math.nist.gov/MatrixMarket>, June 2004.
- [15] J. K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear equations. In J. K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 231–254. Academic Press, New York, NY, 1971.
- [16] R. Scrofano and V. K. Prasanna. Computing Lennard-Jones potentials and forces with reconfigurable hardware. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'04)*, pages 284–290, Las Vegas, NV, June 2004.
- [17] Silicon Graphics Inc. SGI RASC Technology. <http://www.sgi.com/products/rasc/>.
- [18] SRC Computers Inc. Carte Programming Environment. <http://www.srccomp.com/SoftwareElements.htm>.
- [19] SRC Computers, Inc. MAP Processor Specifications. <http://www.srccomp.com/HardwareSpecs.htm>.
- [20] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *FPGA'04: Proceedings of the 2004 ACM/SIGDA Twelfth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2004.
- [21] H. A. van der Vorst. Krylov Subspace Iteration. *Computing in Science & Engineering*, 2(1):32–37, January 2000.
- [22] C. Wolinski, F. Trouw, and M. Gokhale. A preliminary study of molecular dynamics on reconfigurable computers. In *Proceedings of The International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'03)*, Las Vegas, NV, June 2003.
- [23] Xilinx Inc. Documentation and literature. <http://www.xilinx.com/support/library.htm>.
- [24] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1985.
- [25] Y. Saad. *SPARSKIT: a basic tool kit for sparse matrix computations*. <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html>, June 1994.
- [26] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *FPGA'05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 63–74, Monterey, CA, February 2005.