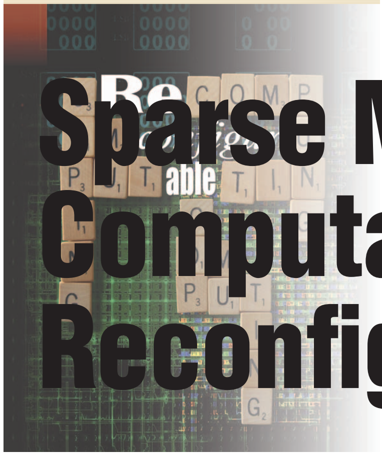

Sparse Matrix Computations on Reconfigurable Hardware

Viktor K. Prasanna, University of Southern California
Gerald R. Morris, US Army Engineer Research and Develpoment Center

Using a high-level-language to hardware-description-language compiler and some novel architectures and algorithms to map two well-known double-precision floating-point sparse matrix iterative-linear-equation solvers—the Jacobi and conjugate gradient methods—onto a reconfigurable computer achieves more than a twofold speedup over software.

Researchers at the US Army Engineer Research and Development Center and the University of Southern California are focusing on algorithms and architectures to facilitate high-performance, reconfigurable computer-based scientific computing. Examples of this research include IEEE Std. 754 floating-point units,[1] molecular dynamics kernels,[2] linear-algebra routines,[3] and sparse matrix solvers.[4] Mapping two sparse matrix solvers onto an FPGA-augmented reconfigurable computers (RC) demonstrated more than a twofold speedup over software.

FIELD-PROGRAMMABLE GATE ARRAYS

Ross Freeman invented the FPGA in the 1980s.[5] These semiconductor devices contain programmable logic elements, interconnections, and I/O blocks, which end users configure to implement complex digital-logic circuits. For RCs, the focus is static random-access memory (SRAM)-based FPGAs, which can be *reprogrammed* using a configuration bitstream.

The traditional FPGA design flow creates a *hardware description language* representation of the design. A synthesis tool translates the HDL into netlist files, which are essentially text-based descriptions of the schematic. Target-specific place-and-route (PAR) and bit-generation tools use netlists to create a configuration bitstream. Simulation at each design stage verifies the functionality. In theory, designers can place any digital logic circuit on an FPGA. In practice, area, clock rate, and I/O are the primary constraints.

RECONFIGURABLE COMPUTERS

First proposed by Gerald Estrin[6] in 1960, the RC is a "fixed plus variable structure" computer that can be "temporarily distorted into a problem-oriented special-purpose computer." The RC languished in relative obscurity for more than 30 years. However, the FPGA has precipitated a reawakening, and RCs that use general-purpose processors (GPPs) and FPGAs as the fixed-plus-variable structure have recently become available. The fine-grained resolution of FPGAs allows reconfiguring the hardware for the specific problem at hand. For applications that have some combination of large-strided or random data reuse, streaming, parallelism, or computationally intensive loops, RCs can achieve higher performance than GPPs.

High-level-language-to-HDL compilers provide features such as pipelined loops and parallel code blocks that allow migrating FPGA-based development out of the hardware design world and into the HLL programming world. The goal is to create deeply pipelined, highly parallelized designs without, as SRC Computers' CEO Jon Huppenthal terms it, "a hardware buddy." Huppenthal made this comment at the ARCS reconfigurable computing conference in August 2005. In concept, researchers can, for example, use C to develop an

Published by the IEEE Computer Society

algorithm and then compile it into a hardware design. In practice, they often must use a hybrid approach involving both HLL and HDL.

Sparse matrix performance

Applications involving sparse matrices can experience significant performance degradation on GPPs. The classic example is *sparse matrix-vector multiply* (SMVM), which has a high ratio of memory references to floating-point arithmetic operations and suffers from irregular memory access patterns. Further, the n-vector, $\mathbf{x}$, cannot fit in the GPP cache for large n, so there may be little chance for data reuse.

Over the past 30 years, researchers have tried to mitigate the poor performance of sparse matrix computations through various approaches such as reordering the data to reduce wasted memory bandwidth,[7] modifying the algorithms to reuse the data,[8] and even building specialized memory controllers.[9] Despite these efforts, sparse matrix performance on GPPs still depends on the matrices' sparsity structure.

In contrast, the runtime of pipelined FPGA-augmented designs, which have single-cycle memory access, does not depend upon the matrix's sparsity structure. Thus, if researchers develop pipelined architectures and extract enough parallelism to realize runtime speedups, they can show that sparse matrix computations have an affinity for RCs.

Floating-point reduction problem

Reductions, which occur frequently in scientific computing, are operations such as accumulation and dot product that input one or more n-vectors and reduce them to a single value. A binary tree of pipelined floating-point cores is a high-performance parallel architecture that accepts input vectors every clock cycle and, after the pipeline latency, emits one result every clock cycle. To accumulate, say, eight numbers, we can use a binary tree with four adders in the first stage, two adders in the second stage, and a single adder in the third stage. However, because of FPGA area constraints, we can only build relatively small trees.

Therefore, we must translate large parallel reductions into a sequence of smaller reductions and reduce the stream of values that are subsequently produced. Consider the dot product architecture that Figure 1 shows. We partition the n-vectors, $\mathbf{x}$ and $\mathbf{y}$, into k-vectors, $\mathbf{u}$ and $\mathbf{v}$. At each clock edge, one pair of k-vectors enters the k-width dot product unit. When the pipeline fills, the partial dot products, d_j, stream out, one value per clock cycle.

The adder accumulates the values in this sequentially delivered vector to produce the dot product, $(\mathbf{x},\mathbf{y})$. Unfortunately, since the adder is pipelined, the loop introduces a multicycle stage. Further, to avoid intermingling, the system must flush the adder after each vector. These stalls result in poor performance and can lead to buffer overruns.

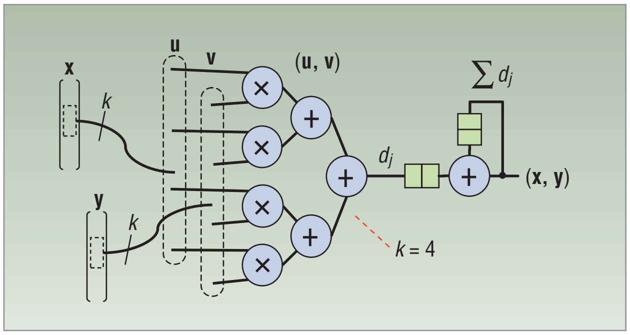

Figure 1. Reduction problem example. The dot product architecture consists of the largest k*-width dot product unit that will fit on the FPGA, followed by a looped adder accumulator. Because the adder is pipelined, the naive adder loop introduces a multicycle pipeline and causes buffer overruns.*

```
1:  algorithm  CG ($A$, $\mathbf{x}$, $\mathbf{b}$)
2:      $\mathbf{x}^{(0)} \leftarrow \mathbf{x}_0$
3:      $\mathbf{p}^{(0)} \leftarrow \mathbf{r}^{(0)} \leftarrow \mathbf{b} - A\mathbf{x}^{(0)}$
4:      $\delta \leftarrow 0$
5:      while ($\Delta$ is too big) do
6:          $\mathbf{q} \leftarrow A\mathbf{p}^{(\delta)}$
7:          $\alpha \leftarrow (\mathbf{r}^{(\delta)}, \boldsymbol{r}^{(\delta)})/(\mathbf{p}^{(\delta)}, \mathbf{q})$
8:          $\mathbf{x}^{(\delta+1)} \leftarrow \mathbf{x}^{(\delta)} + \alpha\mathbf{p}^{(\delta)}$
9:          $\mathbf{r}^{(\delta+1)} \leftarrow \mathbf{r}^{(\delta)} - \alpha\mathbf{q}$
10:         $\beta \leftarrow (\mathbf{r}^{(\delta+1)}, \mathbf{r}^{(\delta+1)})/(\mathbf{r}^{(\delta)}, \mathbf{r}^{(\delta)})$
11:         $\mathbf{p}^{(\delta+1)} \leftarrow \mathbf{r}^{(\delta+1)} + \beta\mathbf{p}^{(\delta)}$
12:         $\delta \leftarrow \delta + 1$
13:     end while
14: end algorithm
```

Figure 2. Conjugate gradient algorithm. The loop calculates the next value of **x** *(estimated solution),* **r** *(residual), and* **p** *(search direction). Each iteration yields a better* **x** *by "walking downhill" in the A-orthogonal (conjugate) direction given by vector* **p***. A convergence test, as idealized by the while clause at line 5, causes the CG algorithm to terminate.*

Thus, solving the reduction problem requires reducing multiple sets of sequentially delivered floating-point vectors without stalling the pipeline or imposing unreasonable buffer requirements.

CONJUGATE GRADIENT SOLVER

The conjugate gradient (CG) method shown in Figure 2, developed by Magnus Hestenes and Eduard Stiefel in 1952,[10] is the best-known iterative method for numerically solving linear equations, $A\mathbf{x} = \mathbf{b}$, whenever A is a symmetric positive-definite (SPD) sparse matrix. As Figure 2 shows, a plot of $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$, where A is an order n SPD matrix,

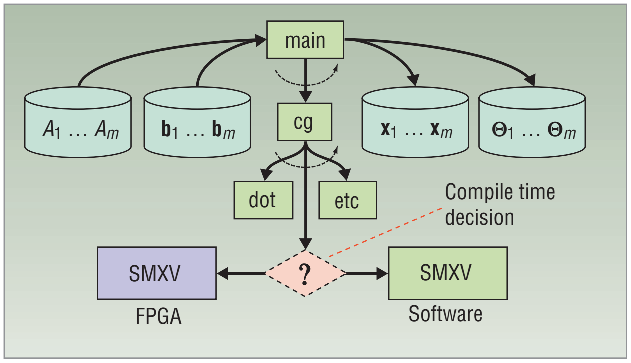
main
$A_1 \ldots A_m$
$\mathbf{b}_1 \ldots \mathbf{b}_m$
cg
$\mathbf{x}_1 \ldots \mathbf{x}_m$
$\Theta_1 \ldots \Theta_m$
dot
etc
Compile time decision
SMXV
FPGA
?
SMXV
Software

Figure 3. Conjugate gradient design. A compile-time decision selects the software-only or FPGA-based version of SMVM.

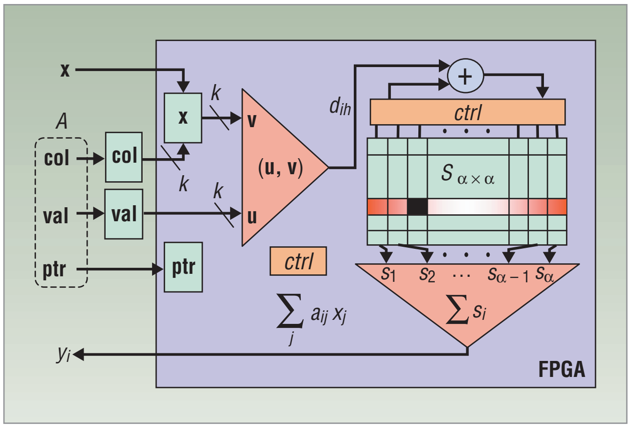

Figure 4. Sparse matrix-vector multiply module. The FPGA-based architecture consists of a k-width dot product core, an α-stage pipelined adder, a partial summation array, S, an output accumulator core, and some on-chip and local memory banks.

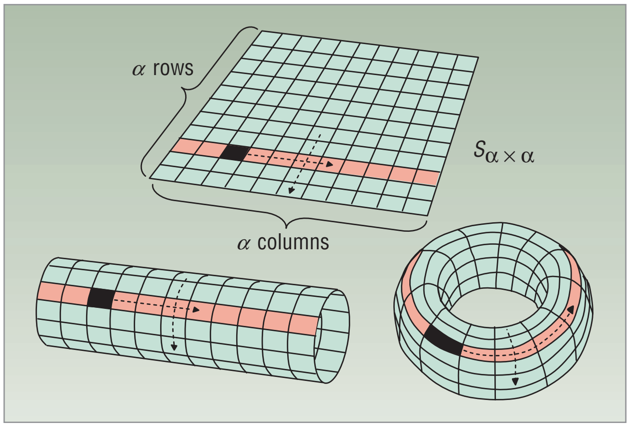

Figure 5. Toroidal access pattern of S. Wrap the α × α array top-to-bottom to produce a cylinder, then end-to-end to produce a torus.

yields an $(n + 1)$-dimensional concave-up parabolic surface. The $\mathbf{x}$ value that minimizes $f(\mathbf{x})$ corresponds to the solution to $A\mathbf{x} = \mathbf{b}$, that is, the $\mathbf{x}$ value at the lowest point in the "bowl" is the solution.

High-level CG design

A profile of CG[4] shows that it spends more than 95 percent of the execution time in SMVM (line 6 of the CG algorithm), so we targeted this module for the FPGA.

Figure 3 shows our high-level CG design. The **main** routine measures how long it takes for CG to solve each set of input equations, $A_i\mathbf{x}_i = \mathbf{b}_i$. The result vectors, $\mathbf{x}_i$, and performance statistics, Θ_i, are written to output files. Since the A matrix is invariant during the entire CG calculation, the FPGA-based SMVM pulls a copy of A one time and stores it in local memory for subsequent iterations. Amortization of the matrix transfer cost across all iterations of CG is a key design feature.

FPGA-based matrix-vector multiply

Figure 4 shows a block diagram of our FPGA-based SMVM architecture. The diagram represents the input sparse matrix, A, in compressed sparse row (CSR) format via the three vectors: **val**, the row-wise matrix values; **col**, the column index of each value; and **ptr**, the position in the **val** vector where each row begins. The basic algorithm for each row calculates a series of partial dot products, d_{ih}, and reduces them to the single value,

$$y_i = \sum_h d_{ih} = \sum_j a_{ij} x_j .$$

The k-width dot product core accepts two double-precision floating-point k-vectors every clock cycle. The **u** inputs from **val** correspond to the next k elements of the A matrix. The corresponding k values from **col** ensure that the matching k elements of **x** are sent to the **v** inputs. After the latency, the core emits a sequential stream of partial dot products.

To reduce the partial dot products, the architecture has an α-stage pipelined adder and a constant-sized α-row by α-column partial summation array, S. A round-robin scheduling algorithm guarantees an α-cycle interval between subsequent references to the same memory location in S. The binary tree output accumulator reduces completed rows of S to produce the components of vector **y**.

The easiest way to envision the round-robin partial summation algorithm is to view the toroidal access pattern of the S array shown in Figure 5. The accumulation of a given input vector is restricted to a specific row—such as the red row—within the S array.

Even if there are more than α elements in the input vector, the major circumference of the torus—the number of columns—is α, thereby ensuring that any previous data at a given location—the black square, for example—has already traversed the adder pipeline and been written back by the time the adder again references that location. If we must reduce a series of small vectors, the minor circumference of the torus—its number of rows—ensures that by the time the algorithm needs to reuse a row, the previous row contents already have been sent to the accumulator and the row initialized to zero. This toroidal access pattern makes S appear to be an infinite two-dimensional array, which can handle arbitrary sets of sequentially delivered vectors without stalling the pipeline.

JACOBI SOLVER

Researchers can use the Jacobi iterative method to solve linear equations, $A\mathbf{x} = \mathbf{b}$, whenever A is a diagonally dominant (DD) matrix. Substituting $A = L + U + D$ into $A\mathbf{x} = \mathbf{b}$ yields the vector form of the Jacobi iteration,

$$\mathbf{x}^{(\delta+1)} \Leftarrow \mathrm{D}^{-1}\,[\mathbf{b} - (L + U)\,\mathbf{x}^{(\delta)}]$$

where L is the lower triangular matrix, U is the upper triangular matrix, D is the diagonal matrix, and δ is the iteration index. The Jacobi component form is given by

$$x_i^{(\partial+1)} \Leftarrow \frac{1}{a_{ii}}\left[b_i - \sum_{j=1:j\neq i}^{j=n} a_{ij}x_j^{(\partial)}\right]. \qquad (1)$$

High-level Jacobi design

The high-level Jacobi design in Figure 6 resembles the high-level CG design, except the entire sparse matrix Jacobi (SJAC) algorithm—a double-precision implementation of the standard Jacobi algorithm—is implemented as either a software-only module or an FPGA-based module. A compile-time decision selects the appropriate version of SJAC. As with CG, the **main** routine measures how long it takes to solve each set of linear equations, then saves the results in output files for subsequent comparison. Again, the FPGA module pulls a copy of matrix A one time and stores it in local memory for subsequent iterations.

FPGA-based sparse matrix Jacobi

Figure 7 shows a block diagram of our FPGA-based SJAC solver. The dot product core accepts two input k-vectors, one per clock cycle. The **u** input corresponds to the next k elements of A; notice that the a_{ii} term is ignored, as required by Equation 1. The corresponding k values from **col** ensure that the matching k elements of $\mathrm{x}^{(\delta)}$ are sent to the **v** input. After the latency, the core emits a stream of dot products, one per clock cycle. The core has an α-stage pipelined subtracter and an α-row by α-column partial summation array S. Each array row is initially set to have b_i in column one and zero in all other columns; as the dot products stream in, they are subtracted from b_i as required by Equation 1. The α-input binary tree output accumulator reduces completed rows of S. The accumulator output is multiplied by the stored $1/\,a_{ii}$ values to produce the $\mathrm{x}^{(\delta\,+\,1)}$ values.

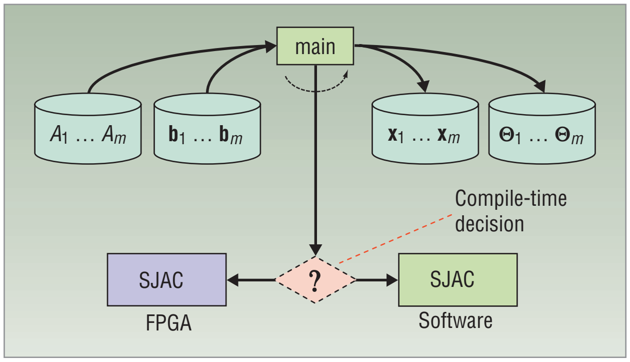
main
$A_1 \ldots A_m$
$\mathbf{b}_1 \ldots \mathbf{b}_m$
$\mathbf{x}_1 \ldots \mathbf{x}_m$
$\Theta_1 \ldots \Theta_m$
Compile-time decision
?
SJAC
FPGA
SJAC
Software

Figure 6. Jacobi design. A compile-time decision selects the software-only or FPGA-based version of SJAC.

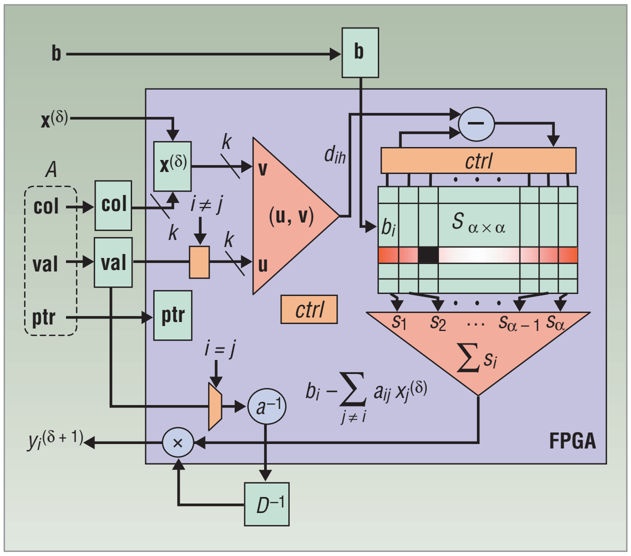

Figure 7. Sparse matrix Jacobi module. The FPGA-based architecture consists of a k-width dot product core, a modified accumulator, a divider, an output multiplier, and some on-chip and local memory banks.

IMPLEMENTATION AND RESULTS

A comparison of our FPGA-augmented implementations with off-the-shelf software-only implementations and algorithms shows that the FPGA-augmented versions achieve greater than a twofold speedup over software.

Target RC and implementation

We used an SRC-6 MAPStation[11] as the target RC. It has dual 2.8-GHz Xeon GPPs with a 512-Kbyte cache and 1 Gbyte of RAM. The MAP Series MPC processor contains two Xilinx Virtex II 6000 FPGAs running at 100 MHz. Each FPGA has 288 Kbytes of on-chip BRAM. Six banks of local memory provide an additional 24 Mbytes of memory. For the FPGA modules, we used the SRC Carte C compiler v2.1 and Xilinx ISE v7.2; for the software modules, we used the Intel C compiler v8.1.

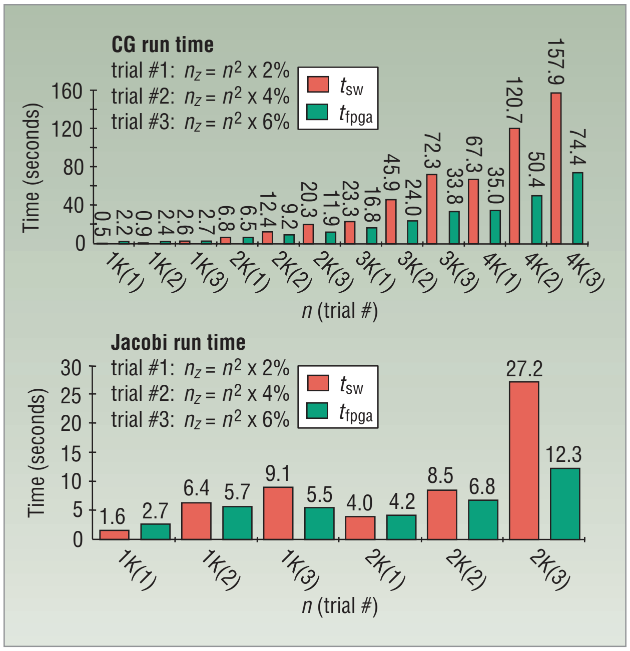

Figure 8. Runtime comparison. The FPGA-based CG achieves a speedup of up to 2.4 over software, and the FPGA-based Jacobi achieves a speedup of up to 2.2 over software.

We used VHDL and the IEEE Std. 754 double-precision floating-point cores[1] to implement the dot product unit and output accumulator. We used Synplify Pro v8.1 to synthesize them and integrated them into the Carte environment as user-defined macros.

Since CG is difficult to implement properly,[12] we used an off-the-shelf implementation from the Sparskit[13] library as a baseline. We also used the optimized software SMVM that came with Sparskit. We did not want to build a poor software CG implementation and then claim to beat it using an RC.

The Jacobi implementation is based on an off-the-shelf algorithm.[14] As with CG, we wanted to base our comparison on a solver recognized within the community. We used the target RC to build CG for matrices up to order $n = 4{,}096$, and we used Jacobi for matrices up to order $n = 2{,}048$. The limiting factor was the number of simultaneous local memory reads. It was necessary to store some vectors, such as x, in the FPGA block memories.

Future RCs will likely have a larger number of local memory banks. If we can put all the vectors into local memory banks, we can expect the implementation to handle significantly larger problems.

Description of test matrices

For each matrix order, 1000, 2000, 3000, and 4000, we generated three SPD matrices that have sparsity values of two, four, and six percent. For example, the two-percent sparsity test matrix for the $n = 1000$ case contains $n_z = n^2 \times 2$ percent $= 10^6 \times 0.02 = 20$ K nonzero entries. The resulting 12 SPD sparse matrices were used as test inputs for the two versions of CG.

For each matrix order, 1000 and 2000, we generated three DD matrices that also have sparsity values of two, four, and six percent. We used the resulting six matrices as test inputs for the two Jacobi versions.

Test results

To capture the entire system behavior, including data transfer time to and from the FPGA-based modules, we instrumented the main routines with microsecond-resolution timers. We ran both versions of the CG and Jacobi solvers using the previously described matrices to capture the entire application's wall-clock runtime.

Figure 8 compares the wall-clock runtime of the FPGA-augmented versions to the software-only versions. For the 1K(*) cases, which fit in the Xeon's 512-Kbyte cache, the software-only version of CG showed the best performance. However, for the remaining test cases, the FPGA-augmented version of CG outperforms software. For the *K(1) cases, the software-only version of Jacobi offered the best performance. For the remaining test cases, the FPGA-augmented Jacobi version ran faster than software.

RELATED RESEARCH

Sreesa Akella and colleagues[15] described a CSR-format SMVM kernel for the SRC-6 reconfigurable computer. Their implementation employs parallel floating-point *multiply accumulator* (MAC) cores. As with our work, they compared the wall-clock runtime of a software version to the wall-clock runtime of the FPGA-augmented version. They also used the Carte HLL- to-HDL compiler to develop the FPGA-based modules. Unfortunately, their implementation "is still 2 to 2.55 slower than software."

Michael deLorimier and André DeHon[16] described an FPGA-only design of a CSR-format SMVM. Their JHDL-based design uses MAC processing elements and a bidirectional-ring communication mechanism. They estimated 1.5 Gflops for a single FPGA. We converted our CG wall-clock runtime results into Gflops. Since each of the n_z nonzero elements in the A matrix is multiplied by the corresponding element in the **x** vector, then added to the other products in a row, we have n_z double-precision floating-point multiply operations and approximately n_z–1 floating-point add operations per SMVM. Given the number of iterations, *iter*, and the wall-clock runtime t_{fpga}, we can approximate CG performance as

$$\text{GFLOPS} = \frac{(2n_z - 1) \times iter}{t_{\text{fpga}} \times 10^9}$$

Table 1 shows the results, which are based on the wall-clock runtime of our complete CG implementation on actual hardware. In contrast, deLorimier and DeHon based their results on post-PAR estimates of SMVM performance and thus might not have included all costs.

In our research at USC, we have mapped several kernels, including dense matrix-vector multiply, onto an FPGA-augmented Cray XD1.[3] The VHDL-based DMVM design uses a dot product tree followed by a serial reduction circuit. Based on wall-clock runtime, it achieves a sustained floating-point performance of 262 Mflops. Obviously, the variable latency of that reduction circuit, for the SMVM case, precludes a direct mapping into an HLL-to-HDL environment because loops containing variable-latency components cannot be pipelined.

In their investigation of RC benchmark suites, Melissa Smith and colleagues[17] mapped CG onto an SRC-6 RC using parallel MAC units and the Carte HLL-to-HDL compiler. This effort compares the wall-clock runtime of the FPGA-augmented version with the software-only version. Their FPGA-augmented implementation was more than two times slower than software.

Yousef El-Kurdi and colleagues[18] described an FPGA-based SMVM design for the banded matrices associated with finite element methods. The architecture is a linear array of processing elements minimized via a novel striping scheme. They estimated a sustained performance of 1.5 Gflops based on post-PAR statistics, which might not have included all costs.

FUTURE WORK

The number of local memory banks needed to provide the parallelism associated with high-performance FPGA kernels presents a recurring limitation. We expect next-generation RCs to have a significantly larger number of memory banks. The soon-to-be-released SRC-7, for example, supports 20 simultaneous memory reads, as opposed to the six simultaneous reads in the SRC-6. In addition, the deeply pipelined floating-point cores used on FPGAs will make unlikely the 100-fold speedups that have been demonstrated for integer applications. However, tenfold overall speedups might be possible.

In our view, current related work demonstrates that performance estimates based on post-PAR statistics might be a bit optimistic because they do not include all costs. When possible, future performance comparisons should be based on actual runtimes on real hardware.

The most obvious future work will be to reconsider the current designs by moving the on-chip stores into the local memory banks and to increase the data path width via parallelism. These two considerations should result in significant speedups and accommodate much larger matrices.

Reconfigurable computers that combine GPPs with FPGAs are now available. The FPGAs can be configured to become, in effect, application-specific coprocessors. Additionally, developers can use HLL-to-HDL compilers to program RCs using traditional HLLs.

Table 1. Approximate CG performance.

Case	n_z	*Iter*	t_{fpga}	Gflops
1K(1)	19,890	4,679	2.2	0.00
1K(2)	39,914	4,377	2.4	0.15
1K(3)	59,808	4,591	2.7	0.20
2K(1)	78,940	10,687	6.5	0.26
2K(2)	160,076	11,170	9.2	0.39
2K(3)	239,840	11,735	12.0	0.47
3K(1)	180,116	19,320	16.8	0.41
3K(2)	363,922	17,813	24.0	0.54
3K(3)	543,344	18,769	33.8	0.60
4K(1)	322,990	28,224	35.0	0.52
4K(2)	639,130	24,519	50.4	0.62
4K(3)	959,680	25,945	74.4	0.67

Our FPGA-augmented designs achieved more than a twofold wall-clock runtime speedup over software. Given that the software-only and FPGA-augmented versions use the same off-the-shelf code and algorithm, are compiled with the same compiler, run on the same platform, and use the same input sets, the comparisons accurately indicate the improvements attributable to FPGA-based acceleration. Despite the limitations in current-generation RCs, our work and related research efforts provide strong evidence that FPGA-augmented RCs could be the next wave in the quest for higher floating-point performance. ■

Acknowledgments

Our work was supported by the US National Science Foundation under award no. CCR-0311823, in part by award no. ACI-0305763, and in part by the Department of Defense High-Performance Computing Modernization Program.

References

1. G. Govindu, R. Scrofano, and V.K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," *Proc. Int'l Conf. Eng. Reconfigurable Systems and Algorithms*, CSREA Press, 2005, pp. 137-148.
2. R. Scrofano et al., "A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers," *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines*, IEEE CS Press, 2006, pp. 23-32.
3. L. Zhuo and V.K. Prasanna, "High-Performance Linear Algebra Operations on Reconfigurable Systems," *Proc. Supercomputing 2005*, IEEE CS Press, 2005, p. 2.
4. G.R. Morris, R.D. Anderson, and V.K. Prasanna, "A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer," *Proc. 14th IEEE*

Symp. Field-Programmable Custom Computing Machines, IEEE CS Press, 2006, pp. 3-12.

5. Xilinx, "How Xilinx Began;" www.xilinx.com/company/ history.htm, 2006.
6. G. Estrin, "Organization of Computer Systems—the Fixed Plus Variable Structure Computer," *Proc. Western Joint Computer Conf.*, Western Joint Computer Conf., 1960, pp. 33-40.
7. E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proc. 24th Nat'l Conf. ACM*, ACM Press, 1969, pp. 157-172.
8. E.J. Im, K.A. Yelick, and R. Vuduc, "SPARSITY: An Optimization Framework for Sparse Matrix Kernels," *Int'l J. High-Performance Computing Applications*, vol. 18, no. 1, 2004, pp. 135-158.
9. L. Zhang et al., "The Impulse Memory Controller," *IEEE Trans. Computers*, vol. 50, no. 11, Nov. 2001, pp. 1117-1132.
10. M. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *J. Research Nat'l Bureau of Standards*, vol. 49, no. 6, 1952, pp. 409-436.
11. SRC Computers, "General-Purpose Reconfigurable Computing Systems;" www.srccomp.com.
12. D.M. O'Leary, "Methods of Conjugate Gradients for Solving Linear Systems," *A Century of Excellence in Measurements, Standards, and Technology: A Chronicle of Selected NBS/NIST Publications, 1901-2000*, NIST Special Publication 958, 2001, pp. 81-85.
13. Y. Saad, "SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations;" www-users.cs.umn.edu/~saad/software/ SPARSKIT/sparskit.html.
14. R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed., SIAM, 1994.
15. S. Akella et al., "Sparse Matrix Vector Multiplication Kernel on a Reconfigurable Computer," *Proc. 9th Ann. High-Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, 2005; www.ll.mit.edu/HPEC/agendas/proc05/ HPEC05_Open.pdf.
16. M. deLorimier and A. DeHon, "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," *Proc. 2005 ACM/SIGDA 13th Int'l Symp. Field-Programmable Gate Arrays*, ACM Press, 2005, pp. 75-85.
17. M.C. Smith, J.S. Vetter, and S.R. Alam, "Investigation of Benchmark Suites for High-Performance Reconfigurable Computing Platforms," *Proc. 9th Ann. Military and Aerospace Programmable Logic Devices Int'l Conf.*, NASA, 2006; www.klabs.org/mapld06/abstracts/index.html.
18. Y. El-Kurdi, W.J. Gross, and D. Giannacopoulos, "Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs," *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines*, IEEE CS Press, 2006, pp. 293-294.

***Viktor K. Prasanna** is Charles Lee Powell Chair in Engineering and a professor of electrical engineering and a professor of computer science at the University of Southern California. His research interests include high-performance computing, parallel and distributed systems, and network computing and embedded systems. Prasanna received a PhD in computer science from Pennsylvania State University. He is a Fellow of the IEEE. Contact him at prasanna@ usc.edu.*

***Gerald R. Morris** is a computer scientist at the US Army Engineer Research and Development Center, Information Technology Laboratory. His research interests include high-performance computing, reconfigurable computing, and acceleration technologies such as ClearSpeed and Cell BE. Morris received a PhD in electrical engineering from the University of Southern California. Contact him at gerald.r.morris@erdc.usace.army.mil.*