Pipelined Data Path for an IEEE-754 64-Bit Floating-Point Jacobi Solver

Gerald R. Morris[1] and Viktor K. Prasanna[2]
Department of Electrical Engineering
University of Southern California, Los Angeles, CA
{grm,prasanna}@usc.edu

Abstract

Solving linear equations is essential for certain embedded applications such as adaptive beam forming and synthetic aperture radar. When direct methods like Cholesky factorization are not viable, it becomes necessary to use an iterative approach. Even when the convergence of the basic iterative methods like Jacobi or Gauss-Seidel cannot be guaranteed, they are often used as preconditioners for more advanced methods like generalized minimum residual (GMRES) [1]. This paper presents a *data path* for an IEEE-754 64-bit floating-point Jacobi iterative solver. The data path component is written in VHDL; it is deeply pipelined and parallelized for efficient execution on field-programmable gate arrays (FPGA). The data path circuit is implemented using a Xilinx Virtex-II Pro as the target; size, clock speed, and peak GFLOPS statistics are presented.

Introduction

Within the high performance embedded computing domain, FPGAs are no longer restricted to their traditional application space as substitutes for application-specific integrated circuits (ASIC). Contemporary research spans a broad range of topics such as application design and synthesis [2], benchmarking [3], variable-precision floating-point operations [4], and higher radix floating-point representations [5]. The mega gate counts, arithmetic capability, and other features of modern FPGAs have precipitated research into general-purpose linear algebra routines [6], as well as floating-point kernels from specific problem domains such as molecular dynamics [7]. FPGAs may soon have an order of magnitude peak floating-point performance over general-purpose processors [8]. SRC, Inc., offers a compact, high-end embedded computer that has FPGA-based acceleration capability [9]. FPGA-based floating-point computational kernels are becoming a reality within the high performance embedded computing domain.

The Jacobi Iterative Method

The Jacobi iterative method for solving $A\mathbf{x} = \mathbf{b}$ involves splitting $n \times n$ matrix, A, into lower triangular, upper triangular, and diagonal matrices and casting the resulting equation in the form of an iteration, $\mathbf{x}^{(\delta+1)} \Leftarrow f(\mathbf{x}^{(\delta)})$, where δ is the iteration index. Plugging $A = L + U + D$ into $A\mathbf{x} = \mathbf{b}$ yields the vector form of the Jacobi iteration:

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

The Jacobi iteration can also be expressed in point form:

$$x_i^{(\delta+1)} \Leftarrow \frac{1}{a_{ii}}\left(b_i - \sum_{j=1|j\neq i}^{n} a_{ij}x_j^{(\delta)}\right) \qquad (2)$$

Equation 2 can be trivially vectorized "since the updates could in principle be done simultaneously" [10]. As such, it is an ideal candidate for implementation within an FPGA.

Design

To motivate the discussion, there is a simplified block diagram of the *complete* Jacobi iterative solver shown in Figure 1. This paper presents the design of the *binary tree data path* shown in the center of the figure. For simplicity, a data path width, $k = 4$, is shown.

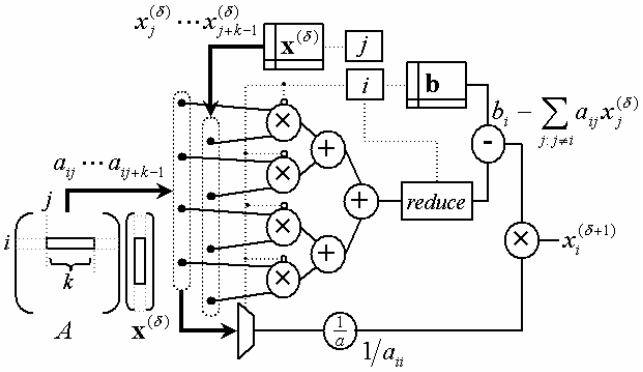

Figure 1: Jacobi Iteration Block Diagram

During initialization, the $\mathbf{b}$ and $\mathbf{x}^{(\delta)}$ vectors are stored internally. Then, the first k-vector (a *subrow*) of row $\mathbf{a}_1$ is read in and held at the multiplier inputs along with the corresponding k-vector of $\mathbf{x}^{(\delta)}$ that was fetched from storage. On the next clock, the pipeline ingests the next subrow of $\mathbf{a}_1$, and so forth until the entire row has been dispatched. At this point, row $\mathbf{a}_2$ begins to enter the data path, etc., until all rows of A are processed. The binary tree reduces the input k-vectors and emits one partial value per clock. The partial sums for row i are accumulated by the *reduce* circuit to generate the Equation 2 summation. This sum is subtracted from b_i and the result is fed into the multiplier with $1/a_{ii}$ to produce $x_i^{(\delta+1)}$. Counter i controls the data path and sets $a_{ii}x_i^{(\delta)} = 0$; i also selects the a_{ii} values into the divider, the b_i values into the subtracter, and tells *reduce* when a new row has started. Counter j ensures that the proper k-vector is read from the $\mathbf{x}^{(\delta)}$ store.

[1] Supported in part by the DoD High Performance Computing Modernization Program (HPCMP).

[2] Supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763.

Two *reduce* circuits are presented in [11], the subrow approach is in [6], and the floating-point units (FPU) are described in [12]. This paper presents the binary tree data path. One of the challenges of testing the data path design was that the tools do not support IEEE-754 representation. Scrofano's *float2hex*, and *hex2float* utilities [7] were modified to generate test vectors for the Model*Sim* VHDL testbench codes.

Experiments

Using the Xilinx ISE 6.3.03, Syn*plicity* Synplify Pro 7.7.1, and Model Technology Model*Sim* XE II 5.8c development tools, with Xilinx Virtex-II Pro as target devices, the data paths were implemented in VHDL. Because of I/O and slice limitations, $k = 8$ is the widest data path that fits on the targets ($k = 16$ will *almost* fit on the xc2vp100). Extensive optimizations such as reentrant routing and floor planning were not used. A 10 ns timing constraint was placed at the top of the design hierarchy. The elided Model*Sim* diagram in Figure 2 shows a latency of 52 clock cycles, which is consistent with the analytical result: 10 cycles for the multiplier stage, and 14 cycles for each of the lg(k) adder stages. Each k-vector in $\mathbf{a}_1$ and the corresponding k-vector in $\mathbf{x}^{(\delta)}$ results in a single tree output value. If the tree output values are added together, one gets the desired summation.

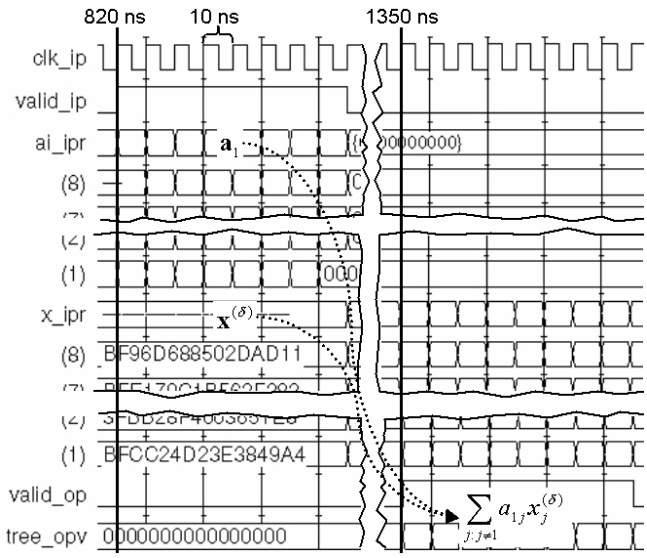

Figure 2: Latency

Xilinx place and route reports show the data path has the size and clock speed characteristics listed in Table 1.

Table 1: Place & Route Statistics

device	*n*	*I/O*	*slices*	*clk*
xc2vp100-6	8	581	20527	9.351 ns
xc2vp70-6	8	581	20188	9.696 ns

If the vector length, $n >> k$, then the pipeline latency can be ignored when calculating peak GFLOPS. There are k multipliers, and k - 1 adders. This corresponds to 15 IEEE 64-bit floating-point operations per clock cycle. The data path can run at 9.35 ns, so there is a peak rating of 1.60 GFLOPS for the given technology. As larger and faster FPGAs are available, this number will rise.

Future Work

Design a complete Jacobi iteration circuit using a subrow design as in [6], coupled with the reduction techniques described in [11]. Design a large sparse matrix version.

Acknowledgments

The authors thank HPCMP computational scientists Tom Oppe, Bill Ward, and Alvaro Fernandez for clarifying some theoretical issues. The authors also thank Ron Scrofano for his *float2hex* and *hex2float* utilities.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Boston: PWS Publishing Company, 1985.

[2] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the streams-C C-to-FPGA compiler: an applications perspective," *2001 ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays*, Feb 2001.

[3] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, and H. Spaanenberg, "A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions," *2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, Feb 2005.

[4] M. Leeser and X. Wang, "Variable precision floating-point division and square root," *8th Annual High Performance Embedded Computing Workshop*, Sep 2004.

[5] B. Catanzaro and B. Nelson, "Higher radix floating-point representations for FPGA-based arithmetic," *13th IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2005.

[6] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," *2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, Feb 2005.

[7] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware," *International Conference on Engineering Reconfigurable Systems and Algorithms*, Jun 2004.

[8] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," *2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, Feb 2004.

[9] SRC Computers, Inc., http://www.srccomp.com

[10] 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, 1994.

[11] G. R. Morris, L. Zhuo, and V. K. Prasanna, "High-performance FPGA-based general reduction methods," *13th IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr 2005.

[12] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," *11th Reconfigurable Architectures Workshop*, Apr 2004.