

An FPGA-Based Floating-Point Jacobi Iterative Solver

Gerald R. Morris and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
3740 McClintock Ave, EEB 200C
Los Angeles, CA 90089-2562
{grm, prasanna}@usc.edu

Abstract

Within the parallel computing domain, field programmable gate arrays (FPGA) are no longer restricted to their traditional role as substitutes for application-specific integrated circuits—as hardware “hidden” from the end user. Several high performance computing vendors offer parallel reconfigurable computers employing user-programmable FPGAs. These exciting new architectures allow end-users to, in effect, create reconfigurable coprocessors targeting the computationally intensive parts of each problem. The increased capability of contemporary FPGAs coupled with the embarrassingly parallel nature of the Jacobi iterative method make the Jacobi method an ideal candidate for hardware acceleration. This paper introduces a parameterized design for a deeply pipelined, highly parallelized IEEE 64-bit floating-point version of the Jacobi method. A Jacobi circuit is implemented using a Xilinx Virtex-II Pro as the target FPGA device. Implementation statistics and performance estimates are presented.

1. Introduction

Field programmable gate arrays (FPGA) are no longer restricted to their traditional role as substitutes for application-specific integrated circuits. SRC Computers offers the MAP[®] processor, which includes two user-programmable FPGAs, in a number of different configurations [13]. Cray has the XD1[™], which can have six user-programmable FPGAs per chassis [1]. Silicon Graphics just announced their FPGA-based RASC[™] technology [3]. End-user scientists and engineers can, in effect, create customized coprocessors targeting the computationally intensive parts of each problem. The high clock rates, mega gate counts, arithmetic capability, on-chip memory, and other features of modern FPGAs have allowed researchers

to develop FPGA-based floating-point cores like multiplication, division, and square root [10, 5]. Thus, FPGA-based floating-point computational kernels for specific problem domains such as molecular dynamics [16, 12] are now possible, as are general-purpose floating-point kernels, e.g., linear algebra [18, 15]. Underwood gives a convincing argument that by 2009 FPGAs will have an order of magnitude peak floating-point performance over central processing units (CPU) [14].

1.1. FPGA primer

FPGAs are hardware devices that can be configured by end users to implement digital logic circuits. For static random access memory (SRAM) based FPGAs, a *configuration bitstream* is loaded onto the FPGA chip. Small 1-bit wide blocks of SRAM, known as look up tables (LUT), are loaded with the truth tables for the logic functions. The LUT address bits correspond to the logic inputs, and the content at that address corresponds to the logic function value.

The basic unit of area on a Xilinx FPGA is a *slice*, which usually consists of two 4-input LUTs, two flip-flops, some muxes, and other control logic. Modern FPGA fabric contains tens-of-thousands of slices, higher-level logic blocks such as memory, multipliers, even CPUs, and a programmable interconnection network arranged in a rectangular grid pattern. In theory, one could design *any* digital logic circuit and put it onto an FPGA.

Typical FPGA design flow takes a hardware description language (HDL) or schematic representation of a circuit and synthesizes it into netlist files, which are essentially text-based descriptions of the schematic. Netlists are processed by the place and route (PAR) and bit generation tools to produce a configuration bitstream.

1.2. FPGA-based supercomputing

Rapidly solving linear equations, $A\mathbf{x} = \mathbf{b}$, where A is an $n \times n$ real matrix, \mathbf{b} is a real n -vector, and \mathbf{x} is the unknown real n -vector, is essential for a number of parallel applications, e.g., when trying to solve partial differential equations (PDE), where the PDE is discretized into a system of equations that are repeatedly solved at each time step. When direct methods like Gaussian elimination are inappropriate, iterative methods are used.

The embarrassingly parallel nature of the Jacobi iterative method coupled with the sculptural fabric of modern FPGAs make the Jacobi method an ideal candidate for hardware acceleration. This paper describes the design of a parameterized, deeply pipelined, highly parallelized FPGA-based IEEE 64-bit floating-point version of the Jacobi method. Using a Xilinx Virtex-II Pro as the target device, a Jacobi circuit is implemented, and various implementation statistics and performance estimates are presented. Analytical techniques are used to estimate the performance of a large sparse matrix Jacobi circuit.

1.3. Organization of this paper

Section 2 provides a brief overview of the Jacobi iterative method and describes the FPGA-based design. Section 3 presents the testing and implementation of a Jacobi circuit on a Xilinx Virtex-II Pro target device and gives some implementation statistics. Section 4 compares circuit execution time to uniprocessor execution time. Section 5 and Section 6 deal with the sparse matrix case, and Section 7 presents the conclusions.

2. Jacobi design

2.1. Overview of the Jacobi method

To motivate the description of the Jacobi iterative method, an iterative approach is used to solve a quadratic equation. The idea is to compute the next value, $x^{(\delta+1)}$, as a function of the current value, $x^{(\delta)}$, i.e.,

$$x^{(\delta+1)} \leftarrow f(x^{(\delta)}). \quad (1)$$

Transforming quadratic equation, $(x+2)(x-5) = 0$, into an iterative format yields,

$$x^{(\delta+1)} \leftarrow \sqrt{3x^{(\delta)} + 10}.$$

Using a starting value $x^{(0)} = 1.000$, the iteration converges to one of the known roots in 7 iterations.

$$\begin{aligned} x^{(1)} &\leftarrow \sqrt{3 \times 1.000 + 10} &\rightarrow x^{(1)} &= 3.606 \\ &\vdots && \\ x^{(7)} &\leftarrow \sqrt{3 \times 4.999 + 10} &\rightarrow x^{(7)} &= 5.000 \end{aligned}$$

To solve $A\mathbf{x} = \mathbf{b}$ iteratively, it is massaged to look like Equation 1. Let $A = L + U + D$, where L is the lower triangular matrix containing all elements of A below the diagonal, U is the upper triangular matrix containing all elements of A above the diagonal, and D is the diagonal matrix consisting of only the diagonal elements of A . Substituting $L + U + D$ into $A\mathbf{x} = \mathbf{b}$ yields the vector form of the Jacobi iteration,

$$\mathbf{x}^{(\delta+1)} \leftarrow D^{-1} [\mathbf{b} - (L + U)\mathbf{x}^{(\delta)}]. \quad (2)$$

The Jacobi iteration can be expressed in point form as,

$$x_i^{(\delta+1)} \leftarrow \frac{1}{a_{ii}} \left[b_i - \sum_{j=1:j \neq i}^{j=n} a_{ij} x_j^{(\delta)} \right]. \quad (3)$$

The Jacobi method often converges rather slowly compared to the more sophisticated methods. Furthermore, convergence is only guaranteed for weakly diagonally dominant matrices [8]. According to Saad, Jacobi is seldom used as a stand-alone solver [17], but rather as a preconditioner for more advanced iterative methods like conjugate gradient, therefore convergence is not considered in this monograph.

2.2. Idealized design

An idealized Jacobi iteration block diagram is shown in Figure 1. It consists of a binary reduction tree, subtraction unit, divider, and some control units. The first matrix row,

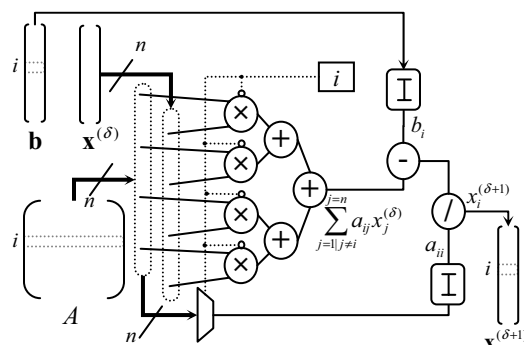


Figure 1. Idealized Jacobi block diagram

\mathbf{a}_1 , is placed at the input lines of the multiplier leaf nodes. Simultaneously, the $\mathbf{x}^{(\delta)}$ vector is placed at the other input lines of the multipliers. On the next clock cycle, the second matrix row, \mathbf{a}_2 , and $\mathbf{x}^{(\delta)}$ vector are ingested. The next cycle deals with the third matrix row and so forth. The binary tree reduces the inputs to produce,

$$\sum_{j=1:j \neq i}^{j=n} a_{ij} x_j^{(\delta)}. \quad (4)$$

Each matrix diagonal element, a_{ii} , is fed into the divider, and each b_i is fed into the subtraction unit. The delay units ensure that a_{ii} and b_i are applied to the division and subtraction units at the correct times. The i counter controls the multipliers so that term, $a_{ii}x_i^{(\delta)}$ is ignored, per Equation 3. The subtraction unit inputs b_i and Equation 4 to produce,

$$b_i - \sum_{j=1:j \neq i}^{j=n} a_{ij}x_j^{(\delta)}. \quad (5)$$

This result, and the appropriate a_{ii} value, enter the divider, which delivers the $x^{(\delta+1)}$ vector, one value per clock cycle.

2.3. Actual design

Figure 2 is a block diagram depicting the actual Jacobi circuit (JAC) design. It consists of a set of block RAM

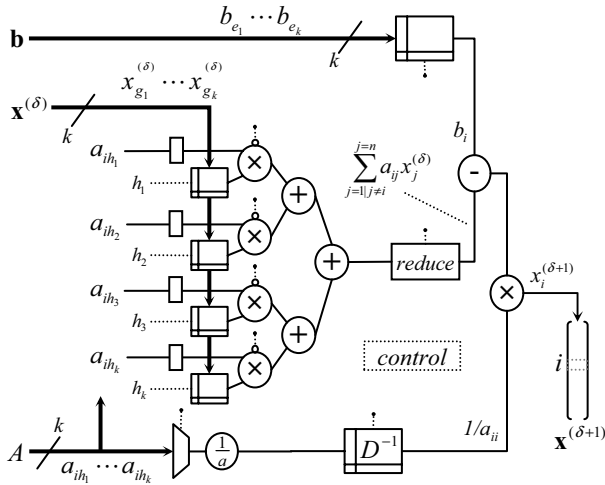


Figure 2. JAC block diagram

(BRAM) stores, a binary reduction tree, a *reduce* unit (described later), a division unit, subtraction unit, multiplication unit, and some control units to orchestrate circuit operation. The design is sculptural; the parameters shown in

Table 1. JAC design parameters

Parameter	Type	Description
n	integer	input vector length
α_m	integer	multiplier latency
α_a	integer	adder latency
α_d	integer	divider latency
k	integer	data path width
m	integer	reduction vector length

Table 1 are used to specify the implementation features. For

example, a data path width, $k = 4$, is shown in Figure 2. The following paragraphs describe how the design works.

Simply increasing the size of the binary reduction tree to deal with larger matrices fails since FPGA resources are fixed; there is an upper bound on the data path width. Whenever $n > k$, it is necessary to multiplex the inputs.

During initialization, the n -vectors, $x^{(\delta)}$, and b , are input, one k -vector per clock cycle, and stored in the BRAMs. Binary tree operation requires k simultaneous values from the $x^{(\delta)}$ vector. Since each leaf node always needs the same subset of values, $x^{(\delta)}$ is strided across the BRAMs.

After the initialization phase, the first k -vector, $a_{11} \dots a_{1k}$, of the first matrix row is placed in registers at the input lines of the leaf-node multipliers. The registers are needed to synchronize the matrix values with the corresponding k elements of the $x^{(\delta)}$ vector that are being fetched from the BRAMs and placed at the other input lines of the multipliers. On the next clock cycle, the second k -vector from the matrix and the matching k elements from the $x^{(\delta)}$ vector are processed. The next cycle deals with the third k -vector and so forth. 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 latency of the multiplication and addition units. Therefore, after $\alpha_m + \alpha_a \lg k$ clock cycles, the first *partial sum* is emitted from the root adder of the binary tree. On the next clock cycle, the next partial sum is emitted, and so forth. The sum of this serially-delivered *reduction vector* corresponds to Equation 4.

A simplistic “adder loop” to sum the reduction vector values fails because of the deeply pipelined floating-point units being employed. Reduction circuits to efficiently handle serially-delivered data are described in [11]. The box labeled, *reduce*, in Figure 2, corresponds to one of these circuits. It inputs the n/k serially delivered reduction vector values and produces the sum given in Equation 4. This sum and the b_i value are fed into the subtraction unit to produce the value shown in Equation 5.

In the idealized design, the division was done at the end of the pipeline. In JAC, the large latency of the division pipeline, α_d , is hidden by doing the division in parallel with the reduction, and storing the result. The $1/a_{ii}$ value is fetched from the D^{-1} BRAM and, with Equation 5, fed into the output multiplier unit. The multiplier delivers the vector, $x^{(\delta+1)}$, one value every n/k clock cycles.

3. Experimental results

The JAC design was coded in the VHDL hardware description language using the parameters shown in Table 2. Using Xilinx ISE 6.3.03i, Mentor Graphics ModelSim XE II 5.8c, and Synplicity Synplify Pro 8.1 tools with a Xilinx Virtex-II Pro XC2VP50 as target device, it was tested and implemented.

Table 2. Implementation parameters

Parameter	Value
n	64
α_m	10
α_a	14
α_d	58
k	8
m	8

3.1. Floating-point IP cores

While not dependent on particular intellectual property (IP) cores, this implementation used the IEEE-754 64-bit floating-point multiply, add, and divide cores described in [5]. Post place and route statistics for these cores are shown

Table 3. IP core statistics

IP Core	Latency [cycles]	f_{max} [MHz]	Area [slices]
multiply	$\alpha_m = 10$	170	935
add	$\alpha_a = 14$	170	1078
divide	$\alpha_d = 58$	140	3015

in Table 3. The latency of the reduction core [9] is given by,

$$\alpha_r = m + 2^{\lceil \lg m \rceil + 1} + (\alpha_a - 1) \lceil \lg m \rceil - 2. \quad (6)$$

3.2. Testing

A C program was developed to generate the VHDL testbench packages used to verify JAC operation. These packages contained the hex representations of the values in the matrices and vectors. VHDL simulation codes dumped hex result values to text files. The testbench packages and ModelSim results were linked into a spreadsheet. Macros were developed to convert between the hex strings and the IEEE-754 64-bit floating point numbers they represent. The spreadsheet parsed the testbench packages and calculated expected results, which were compared with actual results.

3.3. Implementation

After using ModelSim to ensure the design produced correct results, JAC was synthesized by Synplify Pro, and placed and routed by the Xilinx ISE tools. A summary of the post place and route statistics is shown in Table 4.

4. Comparison of JAC to a uniprocessor

In this section the total execution time of the JAC hardware is derived and compared with the expected lower bound on uniprocessor Jacobi software execution time.

Table 4. JAC place and route statistics

Resource	Used	Out Of	%
Multipliers	144	232	62%
BRAMs	95	232	40%
Slices	20,908	23,616	88%
Clock	13ns		

4.1. JAC hardware execution time

The total execution time for the JAC hardware includes data transfer time between the host system and the FPGA plus the circuit execution time. While the latter can be analytically derived, the former must be estimated.

JAC inputs the A matrix and the two vectors, \mathbf{b} and \mathbf{x}^δ . At the end of the iteration it returns vector, $\mathbf{x}^{\delta+1}$. Since a double-precision float is 8 bytes, and $n = 64$, the total data size is,

$$8(n^2 + 3n) = 34,304B. \quad (7)$$

Empirical data from contemporary machines was used to develop a reasonable memory bandwidth value. McCalpin's stream scale data for the top 20 shared memory systems [7], normalized to a single CPU, has a bandwidth of 8.4GB/s. As a check, the stream scale data for PC-compatible systems was also considered. Several of these boxes achieved bandwidth measurements in excess of 4GB/s. Therefore a bandwidth of 4GB/s is assumed, and the estimated data transfer time for the Jacobi circuit is,

$$t_{J_{xf}} = \frac{\text{bytes}}{\text{bandwidth}} = \frac{34,304B}{4 \times 10^9 B/s} = 8.6\mu s. \quad (8)$$

Table 5. JAC pipeline cycles

Action	Cycles	Notes
input $\mathbf{x}^{(\delta)}$	8	n/k
input \mathbf{b}	8	n/k
input $a_{11} \dots a_{1k}$	1	
fetch $x_1 \dots x_k$	1	
traverse tree	52	$\alpha_m + \alpha_a \lg k$
reduce	61	Equation 6
subtract	14	α_a
multiply	10	produces $x_1^{(\delta+1)}$
flush remaining $\mathbf{x}^{(\delta+1)}$	504	$n/k \times (n - 1)$
<i>total</i>	659	

The data shown in Table 5 was derived by following the first element through the pipeline and then calculating the number of clock cycles needed to flush the $n - 1$ remaining $\mathbf{x}^{(\delta+1)}$ vector elements. With a 13ns clock, the 659 cycles corresponds to an execution time of,

$$t_{J_{ex}} = 13 \times 10^{-9} s \times 659 = 8.6\mu s. \quad (9)$$

The total hardware execution time is the sum of the data transfer time in Equation 8, and circuit execution time in Equation 9. Thus, for a single iteration, the JAC hardware takes,

$$tJ_{total} = tJ_{xf} + tJ_{ex} \approx 18\mu s. \quad (10)$$

If the hardware performs I iterations, then the data is only transferred once, and the total hardware execution time is,

$$tJ_{total}(I) = tJ_{xf} + tJ_{ex} \times I \approx 9 + 9I\mu s. \quad (11)$$

4.2. Uniprocessor Jacobi execution time

To calculate a lower bound on uniprocessor Jacobi execution time, one could assume 0 execution time and only consider memory access time (“free cycles–expensive bytes”). Equation 7 shows that a uniprocessor Jacobi requires $\Theta(n^2)$ space for each iteration. Given an average memory access time (AMAT), one could compute lower bound uniprocessor running time as,

$$tU_{min} = \text{AMAT} \times n^2. \quad (12)$$

Empirical data from contemporary machines was used to develop an AMAT value. Table 6 was summarized from information presented in [4]. The 34KB data size of an

Table 6. Average memory access time

Machine	Data Size	
	4KB-16KB	256KB-4MB
HP SC45	1ns	33ns
HP SC40	1ns	39ns
SGI 3900	9ns	29ns
IBM SP3	3ns	45ns

$n = 64$ Jacobi falls between the 16KB and 256KB sizes shown in Table 6. By linearly interpolating these two AMAT columns, and using Equation 12, one can derive Table 7. Using the average value shown in Table 7, the total

Table 7. Lower bound on Jacobi run time

Machine	AMAT	tU_{min}
HP SC45	3.3ns	13.7 μs
HP SC40	3.8ns	15.4 μs
SGI 3900	10.5ns	42.8 μs
IBM SP3	6.1ns	24.8 μs
average	5.9ns	24.2 μs

expected lower bound execution time for a single iteration of a uniprocessor version of Jacobi is,

$$tU_{total} \approx 24\mu s. \quad (13)$$

A double-precision, $n = 64$, C language implementation of the Jacobi method was compiled (Cygwin, bash, gcc, -O3) and executed on a 1.7GHz Intel P4-M CPU having 1MB L2 cache, and a stream scale bandwidth measurement of 2.1GB/s. The average time per iteration for 1000 iterations was 40 μs . Based on this anecdotal evidence, the value shown in Equation 13 seems reasonable. If I iterations are done, each iteration costs the same, since the AMAT calculations already take cache hits/misses into consideration. Therefore the expected multiple iteration uniprocessor execution time is given by,

$$tU_{total}(I) = tU_{total} \times I = 24I\mu s. \quad (14)$$

4.3. Comparison

Based on Equation 10 and Equation 13, the speedup for a single iteration is given by,

$$\frac{tU_{total}}{tJ_{total}} = \frac{24}{18} = 1.3. \quad (15)$$

Based on Equation 11 and Equation 14, the speedup for a large number of iterations ($I \gg 1$), where the data transfer time is negligible, is given by,

$$\frac{tU_{total}(I)}{tJ_{total}(I)} = \frac{24I}{9 + 9I} \approx \frac{24I}{9I} = 2.7. \quad (16)$$

5. Large sparse matrix Jacobi design

This section opens with a brief overview of compressed sparse row (CSR) storage format. But the primary topic is the design for a large sparse matrix Jacobi circuit (SJAC).

5.1. Compressed sparse row format

CSR format is often used when a matrix has a small number of non-zero values, $n_z \ll n^2$. CSR employs three vectors, **val**, **col**, and **ptr**, to store only the n_z non-zero values of a sparse matrix and identify the row and column indices.

val The non-zero values as matrix is traversed row-wise. Vector **val** is of length, n_z .

col The column index of each non-zero value. Thus, if $val_m = a_{ij}$ then $col_m = j$. Vector **col** has length, n_z .

ptr The location in **val** where each matrix row starts. If $val_m = a_{ij}$ then $ptr_i \leq m < ptr_{i+1}$. The number of non-zeros in each row is $len_i = ptr_{i+1} - ptr_i$. To keep calculations consistent, $ptr_{n+1} = n_z + 1$. Thus, vector **ptr** is of length, $n + 1$.

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

$$A = \begin{bmatrix} 5 & 0 & 0 & 2 \\ 0 & 8 & 0 & 0 \\ 1 & 0 & 6 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

$n=4$	val	i	1	2	3	4	5	6
\mapsto	col	val	5	2	8	1	6	3
$n_z=6$	ptr	col	1	4	2	1	3	4
	len	ptr	1	3	4	6	7	\swarrow
		len	2	1	2	1	\swarrow	$n_z + 1$

Figure 3. CSR format

5.2. Sparse matrix design

As shown in Figure 4, SJAC consists of a set of BRAM stores, a binary reduction tree, a reduce unit, division unit, subtraction unit, multiplication unit, and some control units. The essential operation is identical to JAC. The circuit goes

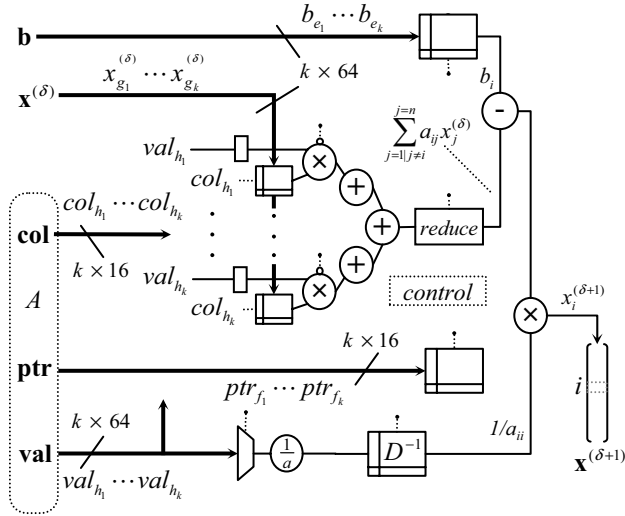


Figure 4. SJAC block diagram

through an initialization phase and then pipelines k values per clock cycle through its k -width data path. The following paragraphs describe how SJAC works. They also assess the impact of sparse matrices on cycle count performance when migrating the design from JAC to SJAC.

In addition to the 64-bit \mathbf{b} , $\mathbf{x}^{(\delta)}$, and \mathbf{val} inputs, SJAC requires the 16-bit \mathbf{ptr} and \mathbf{col} inputs associated with the CSR format sparse matrix.

The two n -vectors, $\mathbf{x}^{(\delta)}$, and \mathbf{b} are brought in, one k -vector per cycle, and stored in BRAM. Binary tree operation requires k simultaneous values from the $\mathbf{x}^{(\delta)}$ vector. A leaf node may need *any* $\mathbf{x}^{(\delta)}$ value, and each BRAM can only deliver one value per clock cycle. Therefore, $\mathbf{x}^{(\delta)}$ cannot be strided across the BRAMs; a complete copy is stored at each leaf node. This is done in parallel and does not affect the number of cycles. During this same time frame, the \mathbf{ptr} ($n+1$)-vector is brought in and stored in BRAM. Thus, the

number of SJAC initialization cycles is the same as JAC.

After initialization, the first k elements, $val_1 \dots val_k$ (a k -group), of the first matrix row are placed in registers at the input lines of the leaf-node multipliers. This synchronizes them with the corresponding k elements of the $\mathbf{x}^{(\delta)}$ vector being fetched from BRAMs using the $col_1 \dots col_k$ addresses. On the next clock cycle, the second k -group from the matrix is processed, and so forth. The circuit still inputs k values per clock cycle and applies them to the binary reduction tree. Thus, the cycle count through the binary reduction tree does not change when going from JAC to SJAC. Furthermore, the operation of the serial reduction unit, division unit, subtraction unit, and output multiplier is identical to JAC. In short, the cycle count performance remains unchanged when going from JAC to SJAC.

6. Comparison of SJAC to a uniprocessor

In this section, the total performance of the SJAC hardware is derived. A highly optimized uniprocessor sparse matrix Jacobian (SMJ) could not be found in the literature, but (as will be shown later) uniprocessor sparse matrix-vector multiply (SMV) performance is a bound on SMJ performance. Additionally, MFLOPS performance data for a highly tuned uniprocessor SMV has already been published [18]. Therefore a performance comparison (based on MFLOPS rather than execution time) is made between SJAC and SMV. Here is an outline of the approach:

1. Estimate MFLOPS performance of SJAC for sparse matrices in [18] using cycle counting and bandwidth.
2. Show that SMV performance is an *upper bound* on SMJ performance.
3. Compare performance of SJAC and SMV.
4. If SJAC can beat SMV, it can certainly beat a uniprocessor SMJ.

6.1. SJAC hardware performance

As in Section 4.1, data transfer time, $t_{J_{xf}}$, and circuit execution time, $t_{J_{ex}}$, are used to calculate overall hardware performance.

Transfer time, as shown in Table 9, is calculated using byte counts and a 4GB/s memory bandwidth. SJAC data transfer involves not only the $3n + n_z$ 64-bit floating-point numbers in \mathbf{b} , $\mathbf{x}^{(\delta)}$, $\mathbf{x}^{(\delta+1)}$, and \mathbf{val} , but also the $n+1 + n_z$ 16-bit integers in \mathbf{ptr} , and \mathbf{col} .

The circuit execution time shown in Table 9 is calculated by multiplying SJAC pipeline cycles by the 13ns clock cycle time. To get SJAC pipeline cycles, Table 5 was modified as shown in Table 8.

Table 8. SJAC pipeline cycles

Action	Cycles	Notes
input $\mathbf{x}^{(\delta)}$	n/k	n elements
input \mathbf{b}	n/k	n elements
input $1^{st} k$ \mathbf{a}_1 vals	1	
fetch \mathbf{x} values	1	
traverse tree	52	$\alpha_m + \alpha_a \lg k$
reduce	use Equation 6	$m = nz_{av}/k$
subtract	14	α_a
multiply	10	produces $x_1^{(\delta+1)}$
flush $\mathbf{x}^{(\delta+1)}$	$(n-1)nz_{av}/k$	$n-1$ elements

When converting the transfer and execution times into MFLOPS, $OP_s = 2n_z$ is used, where OP_s is the number of floating-point operations needed to do a sparse matrix-vector multiply (n_z multiplies and n_z adds). For a large number of SJAC iterations ($I \gg 1$), the transfer time is negligible and one can calculate MFLOPS using only execution time. The results are shown in Table 9. In reality, none of

Table 9. SJAC performance

Matrix	OP_s	tJ_{xf}	tJ_{ex}	MFLOPS	
				$I = 1$	$I \gg 1$
rdist1	188816	263 μ s	176 μ s	430	1072
gemat11	66216	115 μ s	81 μ s	338	816
lns_3937	50814	89 μ s	65 μ s	330	782
sherman5	41586	74 μ s	55 μ s	324	758
mcfe	48764	66 μ s	44 μ s	445	1116
jpwh_991	12054	22 μ s	17 μ s	312	704
bp_1600	9682	17 μ s	14 μ s	304	673
str_600	6558	11 μ s	12 μ s	293	554

these matrices could be used in an actual Jacobi iteration since they have zeros in the diagonal. However, this does not negate their value relative to estimating performance of the Jacobi hardware.

6.2. Uniprocessor Jacobi performance

The summation term in Equation 3 is nearly identical to a matrix-vector multiply. In the sparse case, the $j \neq i$ index constraint means there *could be* one less add and one less multiply in SMJ than in SMV. However, SMJ must subtract the summation from b_i , which cancels the cost of the “missing” addition. The division, which cannot be done in parallel on a uniprocessor, covers the cost of the “missing” multiply. Therefore, uniprocessor SMJ cannot run faster than uniprocessor SMV, i.e.,

$$\text{MFLOPS}_{\text{SMJ}} \leq \text{MFLOPS}_{\text{SMV}}. \quad (17)$$

Zhuo [18] used a highly optimized uniprocessor SMV obtained via SPARSITY [6], with a set of documented matrices [2], to obtain the MFLOPS performance data shown in Table 10. Column n_z is the number of non-zero elements in the matrix, and nz_{av} is the average number of non-zeros in a row. Per Equation 17, these data are used as

Table 10. Uniprocessor SMV performance

Matrix	n	n_z	nz_{av}	MFLOPS
rdist1	4134	94408	23	387
gemat11	4929	33108	7	136
lns_3937	3937	25407	6	104
sherman5	3312	20793	6	85
mcfe	765	24382	32	100
jpwh_991	991	6027	6	25
bp_1600	822	4841	6	20
str_600	363	3279	9	15

upper bound MFLOPS performance estimates for SMJ. The uniprocessor performance of SMV, and hence SMJ, is limited by the minimal locality characteristics of sparse matrix-vector multiply. Contiguous \mathbf{x} vector values are seldom used due to the gaps in the \mathbf{A} matrix. Entire cache lines are often brought in just to obtain a single value. Performing multiple iterations will not solve the problem since the CPU still has to bounce all over memory to get the appropriate values. Therefore, SMJ performance is independent of the number of iterations.

6.3. Comparison

Using the data in Table 9, and Table 10, one can calculate the speedups for a single iteration ($I = 1$) and a large number of iterations ($I \gg 1$), as shown in Table 11. All test

Table 11. Performance comparison

Matrix	Speedup	
	$I = 1$	$I \gg 1$
rdist1	1.1	2.8
gemat11	2.5	6.0
lns_3937	3.2	7.5
sherman5	3.8	8.9
mcfe	4.5	11.2
jpwh_991	12.6	28.5
bp_1600	15.3	33.9
str_600	19.5	36.8

matrices had better performance on SJAC than in software. Matrices with an irregular structure, as depicted in Figure 5, did especially well in SJAC. The structure diagrams depict the full $n \times n$ matrix with a black dot where the matrix has a non-zero, and a white dot where it does not.

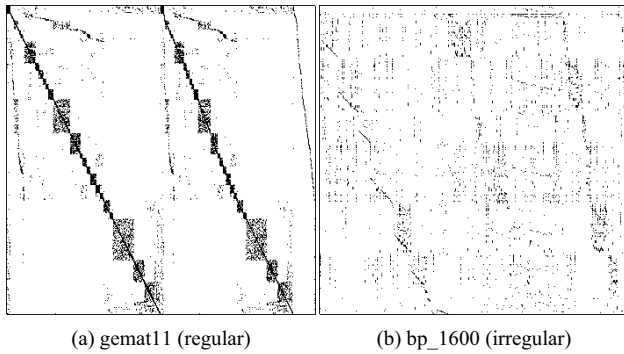


Figure 5. Regular and irregular matrices

7. Conclusion

This research has demonstrated that the design of sophisticated FPGA-based double-precision floating-point kernels is feasible. The paper showed the design and implementation of a parameterized, deeply pipelined FPGA-based IEEE 64-bit floating-point version of the Jacobi iterative method. Performance estimates, which included both data transfer and execution time, showed that the dense Jacobi circuit had a 1.3 speedup, for a single iteration, when compared to uniprocessor implementations. The multiple iteration speedup was 2.7. For a single iteration, a large sparse matrix Jacobi circuit could achieve an estimated speedup of 1.1 to 19.5, when compared to highly optimized uniprocessor implementations. Multiple iteration speedups ranged from 2.8 to 36.8. Sparse matrices having an irregular structure had the biggest speedups. Clearly, FPGA-based acceleration of floating-point scientific kernels should be considered when trying to improve computational performance.

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 High Performance Computing Modernization Program (HPCMP). The authors thank their colleagues Ling Zhuo and Ron Scrofano and HPCMP computational scientists Alvaro Fernandez, Tom Oppe, and Bill Ward for their myriad inputs.

References

- [1] Cray Inc. Cray XD1™. <http://www.cray.com/products/xd1>.
- [2] T. Davis. University of Florida sparse matrix collection. Technical report, <http://www.cise.ufl.edu/research/sparse/matrices>, University of Florida, 2004.
- [3] EE Times. SGI offers application acceleration FPGA module. <http://www.eetimes.com>, September 2005.
- [4] G. R. Morris. The Effect of Cache on Memory Access Time. Technical report, ERDC MSRC, July 2003. <http://www.erdhpc.mil/index.htm>.
- [5] 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)*, Nevada, USA, June 2005.
- [6] E. J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.
- [7] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. <http://www.cs.virginia.edu/stream>, June 2005.
- [8] K. R. James. Convergence of matrix iterations subject to diagonal dominance. *SIAM Journal on Numerical Analysis*, 10(3):478–484, June 1973.
- [9] L. Zhuo. Reduction Circuit Summary. “P-Group” internal presentation, lzhuo@usc.edu.
- [10] M. Leeser and X. Wang. Variable precision floating-point division and square root. In *Proceedings of the 8th Annual High Performance Embedded Computing Workshop*, pages 47–48, Lexington, MA, September 2004.
- [11] 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, April 2005.
- [12] 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*, pages 284–290, Las Vegas, NV, June 2004.
- [13] SRC Computers, Inc. Systems and Servers. <http://www.srccomp.com/Servers.htm>.
- [14] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA Twelfth International Symposium on Field Programmable Gate Arrays*, Napa, CA, February 2004.
- [15] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.
- [16] 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)*, Nevada, USA, June 2003.
- [17] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1985.
- [18] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, February 2005.