LA-UR- 08-4599

Approved for public release; distribution is unlimited.

Title:	A Complete Implementation of the Conjugate Gradient Algorithm on a Reconfigurable Supercomputer
Author(s):	David DuBois, Andrew DuBois, Thomas Boorman, Carolyn Connor, Steve Poole
Intended for:	Journal: ACM TRETS (Transactions on Reconfigurable Tech and Systems)

Los Alamos
NATIONAL LABORATORY
EST.1943

Form 836 (7/06)

A Complete Implementation of the Conjugate Gradient Algorithm on a Reconfigurable Supercomputer

DAVID DUBOIS, ANDREW DUBOIS, THOMAS BOORMAN, CAROLYN CONNOR
Los Alamos National Laboratory
STEVE POOLE
Oak Ridge National Laboratory

The conjugate gradient is a prominent iterative method for solving systems of sparse linear equations. Large-scale scientific applications often utilize a conjugate gradient solver at their computational core. In this paper we present a field programmable gate array (FPGA) based implementation of a double precision, non-preconditioned, conjugate gradient solver for finite-element or finite-difference methods. Our work utilizes the SRC Computers, Inc. MAPStation hardware platform along with the "Carte" software programming environment to ease the programming workload when working with the hybrid (CPU/FPGA) environment. The implementation is designed to handle large sparse matrices of up to order N x N where N <= 116,394, with up to 7 non-zero, 64-bit elements per sparse row. This implementation utilizes an optimized sparse matrix-vector multiply operation which is critical for obtaining high performance. Direct parallel implementations of loop unrolling and loop fusion are utilized to extract performance from the various vector/matrix operations. Rather than utilize the FPGA devices as function off-load accelerators, our implementation uses the FPGAs to implement the core conjugate gradient algorithm. Measured run-time performance data is presented comparing the FPGA implementation to a software-only version showing that the FPGA can outperform processors running up to 30x the clock rate. In conclusion we take a look at the new SRC-7 system and estimate the performance of this algorithm on that architecture.

Categories and Subject Descriptors: C.1.3 **[Processor Architectures]**:Other Architecture Styles – *Adaptable architectures, Heterogeneous (hybrid) systems Miscellaneous – Hybrid systems*; C.4 **[Computer Systems Organization]**: Performance of systems – *Design Studies, Performance attributes*; G.1.3 **[Numerical Analysis]**: Numerical Linear Algebra - *Linear systems, Sparse, structured, and very large systems*; G.1.5 [Numerical Analysis]: Roots of Nonlinear Equations – *Iterative methods*; G.1.6 **[Numerical Analysis]**: Optimization – *Gradient methods*; G.1.8 **[Numerical Analysis]**: Partial Differential Equations – *Iterative solution techniques*; G.1.0 **[Numerical Analysis]**: General – *Computer arithmetic, Numerical algorithms*; G.4 **[Mathematics of Computing]**: Mathematical Software – *Algorithm design and analysis*;

General Terms: FPGA, Conjugate Gradient, Sparse Matrix-Vector Multiplication, Reconfigurable hardware, Iterative methods

Additional Key Words and Phrases: accelerator, architecture, reconfigurable, hybrid computing

1. INTRODUCTION

The Conjugate Gradient Method (CG) is a member of a family of iterative solvers known as Krylov subspace methods used primarily on large sparse linear systems arising from the discretization of partial differential equations (PDEs). CG is effective for systems of the form:

$$A\vec{x} = \vec{b},$$

where A is a square N x N sparse matrix [5].

CG uses successive approximations to obtain a more accurate solution at each step. It is considered a nonstationary method generating a sequence of conjugate (or orthogonal) vectors. These vectors are the gradients of a quadratic function, when minimized, is equivalent to solving the linear system [1].

To compress the sparse matrix A, the ELLPACK-ITPACK format was chosen [10]. This format is efficient for the matrix-vector multiply operation and performs well on vector style architectures. The ELLPACK-ITPACK format is generated from the original N x N matrix by doing a left shift of all the non-zero elements of the matrix. By compressing the matrix in this fashion, we gain substantial space and computational savings.

Each successive iteration of the CG involves one Sparse Matrix-Vector Multiplication (SMVM), three vector updates, and two inner products. The SMVM is the time dominant computational portion executed per iteration of the CG [14]. For general purpose processors, the SMVM performs poorly for three primary reasons [17]. First, the lack of data locality causes large numbers of misses within the caches of the memory hierarchy. Second, the multiple load/store units on many processors

have a tendency to miss while trying to load the same cache line. Finally, SMVM codes execute a large number of loads compared to the number of floating point operations they perform placing a heavy load on the load/store units, and on integer ALUs that compute the addresses. For most current generation processors, these load/store units are often the bottleneck in SMVM leaving the floating-point units underutilized.

A FPGA is a semiconductor device containing programmable logic elements and programmable interconnects [12]. Most, if not all current generation FPGA devices include higher-level embedded functions (such as adders and multipliers) along with embedded memories. These devices support full or partial in-system reconfiguration allowing them to be retargeted to solve a variety of problems.

FPGAs have rapidly grown in logic density, capability, and performance. Double precision floating-point (DPFP) designs implemented on current generation FPGA devices typically run at clock frequencies 10-30 times slower than typical microprocessor-based systems. Even with this disparity in operational frequency, FPGA-based designs can now offer peak floating point performance equaling or surpassing that offered by microprocessor-based systems [18].

Modern high-density FPGAs allow the user to implement multiple copies of the same computation by unrolling or strip mining loops [6]. With the ability to stream data to/from memory while utilizing fully pipelined functional units, FPGA based systems attain high levels of performance relative to their clock speeds. The ability to configure the FPGA to implement only the logic required for a given functionality, coupled with the much lower operational frequency, yields large power savings when compared to general-purpose processors. Capitalizing on the potential of FPGAs, supercomputing vendors such as SRC Computers [15], Cray [2] and SGI [13] offer high-performance computing systems combining standard CPUs with FPGAs.

Implementation of CG using FPGAs has been documented in recent papers. Morris and Prasanna [11] have presented an FPGA-augmented implementation of the CG on an SRC-6. Their implementation presents a CPU/FPGA accelerated hybrid approach. They offload the time consuming SMVM function to the FPGA while computing the remainder of the CG on a traditional CPU. Problem size is quite constrained due to the dependency on BRAM for storage of several critical data structures. They show a speedup of 1.3X over a 2.8 GHz Xeon processor for one case where the problem data did not fit entirely in the processor's cache. Maslennikow et al. [9] present an FPGA implementation of CG using fractional numbers which is limited to small problem sizes with matrix rank up to 1024. Their implementation is targeted at the Xilinx Virtex2-Pro XC2VP4 and calculate performance of 270 MFlops using a banded matrix of rank 1024 with 5 non-zero elements per row.

In this work we present an optimized FPGA-based implementation of a DPFP, non-preconditioned, Conjugate Gradient algorithm for finite element or finite difference methods. It accepts large sparse matrices represented in ELLPACK-ITPACK format using up to seven sparse elements per row. This work was part of a technology assessment that was performed in the summer of 2005.

The implementation was done using the SRC MAPStation utilizing Intel Xeon Processors, a MAP processor (containing the FPGAs) and the SNAP interconnect [15]. The Carte software tools were used for code development in conjunction with the Intel C++ compiler [8]. Through careful placement of array data in the On-Board Memory (OBM) of the MAP processor, optimal use of the aggregated memory bandwidth was achieved. Functional parallelism was exploited to overlap independent computations and gain substantial speed-ups.

The remainder of this paper is organized as follows: In Sections 2 through 4, we discuss sparse matrices and Krylov methods, introduce the SRC system hardware and software, outline the non-preconditioned CG algorithm, the SMVM, and discuss performance issues with traditional microprocessors. In Section 5, we present our implementations of the CG using the SRC Mapstation and software tools. In Section 6, we discuss our results and place these results in context via comparisons to microprocessor implementations. In Section 7, we present potential future work and we conclude in Section 8.

2. BACKGROUND

Sparse matrices, derived from PDEs, occur in many scientific application areas, especially Physics and Mechanical Engineering where a physical phenomenon needs to be mathematically described. PDEs are used to describe phenomena such as fluid flow, the growth of crystals, gravitation, diffusion, and the behavior of electromagnetic fields.

The solution to a nonsingular linear system $A\vec{x} = \vec{b}$ lies in a Krylov space whose dimension is the degree of the minimal polynomial of A. If this minimal polynomial of A has a low degree, a Krylov method has the opportunity to converge rapidly [7]. Also, iterative methods such as CG scale well to very large problem sizes, parallelize easily, and have a shorter time to solution compared to direct methods (e.g., Gaussian elimination). These are the dominant reasons why Krylov methods are selected for these types of problems and are particularly well suited for use on large-scale scientific simulation codes that in turn are defined by sparse linear systems.

The ELLPACK-ITPACK storage format was chosen as it is the most efficient for the matrix-vector multiply operation whose rows have a similar number of zeros. This format, due to its regular structure, allows a more optimal fit of the problem to the FPGA. The format also allows compilers greater chances for high levels of optimization. Processing a known fixed amount of data per cycle produces a more focused design. Our goal was to produce the best-case results on this problem rather than producing a more general solution. A standard Compressed Row Storage (CRS) format could easily be converted to run on this design.

3. SRC SYSTEM

3.1. MAP Processor

The MAP processor uses reconfigurable components to accomplish control, user-defined computation, data pre-fetch, and data access functions, as shown in Fig 1. Each MAP processor contains both fixed and reconfigurable elements of hardware.

The fixed hardware elements include general control logic and a DMA engine in the Controller FPGA as well as discrete, dual ported SRAM called On-Board Memory (OBM). The reconfigurable hardware elements consist of two Xilinx Virtex II 6000 FPGAs and are referred to as User Logic FPGAs (U_LOGIC).The OBM consists of six banks of dual port 512k x 64-bit SRAM providing a total of 24MB. The Controller FPGA communicates with the OBM via six, unshared ports. The U_LOGIC devices share the remaining six ports. The seventh, discrete bank of dual ported SRAM is 512k x 64-bits. Unlike the OBM, this memory has both ports connected to the bridge port connecting the two U_LOGIC devices. This seventh bank may be configured to look like two independent 2MB memory ports to the U_LOGIC devices.

The entire MAP processor runs at a fixed frequency of 100 MHz. At this frequency, each of the 64-bit OBM memories, and the two ports of the seventh memory, can deliver a sustained maximum bandwidth of 800 MB/s per port. This yields an aggregated memory bandwidth to each U_LOGIC of 6400 MB/s. It should be pointed out that simultaneous access of a particular OBM, or the seventh memory, by both U_LOGIC devices is not permitted since they are shared.

The MAP processor has separate input and output ports with each sustaining a data payload bandwidth of 1400 MB/s. There are also two General Purpose I/O (GPIO) ports providing an additional data payload of 4800 MB/s for direct MAP-to-MAP communications or connections to external devices.

3.2. The SRC MAPStation and SNAP Interface

The SRC MAPStation utilizes a dual Intel Xeon motherboard. Each Xeon processor runs at 2.8 GHz, and has a 512 KB L2 cache. A single SNAP module is inserted into one of the DDR memory slots and is connected to a single MAP processor module.

The SNAP module includes an intelligent DMA controller capable of performing complex DMA prefetch and data access functions. These functions include, data packing, strided access and scatter/gather, to maximize the efficient use of the system interconnect bandwidth.

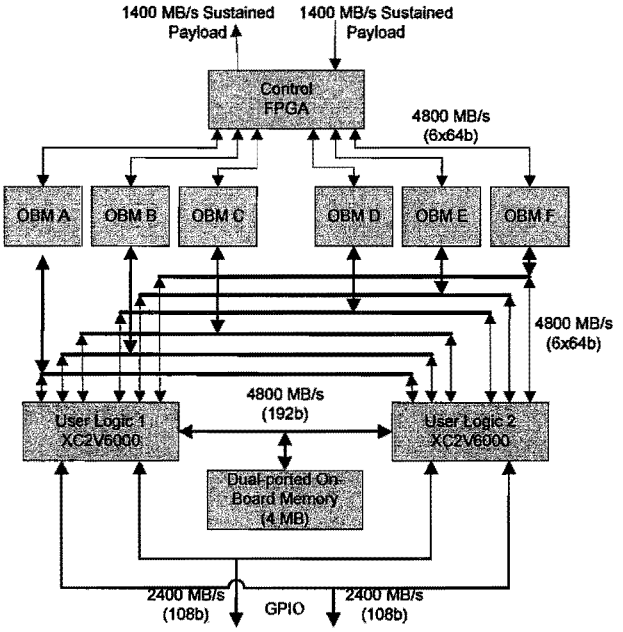

Fig 1. SRC MAP Diagram [15]

3.3. 64-bit Double Precision Floating Point Macros

Our code utilizes a common set of SRC supplied user macros. The MAP compiler translates the source code's various basic operations into macro instantiations [16].

There are two types of double precision floating point macros: Modified (M) and Smaller Area (SA). The M revision macros closely implement the IEEE-754 standard except for handling of special number inputs such as Denormals, NaNs and Infinity. These macros implement the "round to the nearest" mode seemingly the most common rounding mode with Intel Processors. The SA version macros are a subset of the M macros. The SA macros are not as accurate as the M macros but require fewer logic recourses in the FPGA.

TABLE I gives a breakdown of the area and latency of some of the M 64-bit floating-point macros. The latency figure relates to the pipeline depth required when implementing the macro. Our implementation of the CG utilizes the M revision macros exclusively.

Table I. SRC Macros [16]

Area and Latency values for 64-bit floating point SRC macros

FP64:	Slices x*y	Latency
fp_divide_64m	44x96	59
fp_addsub_64m	24x64	13
fp_mult_64m	8x128	11
fp_accum_64m	24x64	101
fp_mac_64m	18x128	112

3.4. MAP Implementation Constraints

The MAP function contains both the computational portion of the code along with the data movement instructions to manage the data movement to and from the MAP's OBM. All array data must be transferred from SCM to OBM using DMA or streaming calls. The compiler automatically moves scalar formal parameters to and from the MAP as needed.

The user logic within the FPGA is clocked at 100 MHz, so achieving good performance relies on parallel execution of functional units. Operations within loops that have no data dependencies can execute concurrently. There are four potential sources of loop slowdown: loop-carried scalar dependencies, loop carried memory dependency, multiple accesses to a memory unit (OBM or BRAM), and periodic input macro calls.

OBM is partitioned into six separately addressable memory banks. When a loop contains multiple references to the same bank, even when there is no dependency among the references, the compiler may need to slow down the loop to allow each reference to get its turn to access the bank.

In this design, we utilize both U_LOGIC devices available on the MAP processor. In order to do this we partitioned the CG function into two separate routines. The two routines have different capabilities and responsibilities as listed below in TABLE II.

Table II. U_LOGIC capabilities & responsibilities

Primary Routine	**Secondary Routine**
Issues all DMAs	Has no access to MAP calling parameters
Initially controls access to OBM	Cannot issue DMAs
Issues ONLY send_perms	Issues ONLY recv_perms

4. THE CONJUGANT GRADIENT

4.1. CG Pseudo code

The non-preconditioned CG implementation we present in this work is shown in the pseudo code of Fig 2. Our approach was to provide an implementation of the CG that fits on the FPGA rather than perhaps the most accurate or fastest converging algorithm. In particular we chose not to implement a preconditioned CG algorithm. Also, we did not attempt to periodically correct for accumulated error on the residual vector. This operation would require an extra SMVM operation along with a vector-vector subtraction. There are several vector-vector/vector-matrix operations executed during the computation:

- DPFP Dot Product (DDOT): 2, 4, 7
- DPFP constant times a vector plus a vector (DAXPY): 5, 6, 8

• DPFP SMVM: 1, 3

Of primary concern are the computations taking place within the "while loop". These operations take the majority of the overall processing time and it is within this loop where the majority of our optimization work is focused.

$$\vec{r} \Leftarrow \vec{b} - A\vec{x} \quad *(1)$$
$$\vec{d} \Leftarrow \vec{r}$$
$$\delta_{new} \Leftarrow \vec{r}^T\vec{r} \quad (2)$$
$$\delta_0 \Leftarrow \delta_{new}$$
$$i \Leftarrow 0$$
$$While\ i < i_{\max}\ and\ \delta_{new} > \varepsilon^2\delta_0\ do$$
$$\vec{q} \Leftarrow A\vec{d} \quad (3)$$
$$\alpha\_accum \Leftarrow d^T\vec{q} \quad (4)$$
$$\alpha \Leftarrow \frac{\delta new}{\alpha\_accum}$$
$$\vec{x} \Leftarrow \vec{x} + \alpha\vec{d} \quad (5)$$
$$\vec{r} \Leftarrow \vec{r} - \alpha\vec{q} \quad (6)$$
$$\delta_{old} \Leftarrow \delta_{new}$$
$$\beta \Leftarrow \frac{\delta new}{\delta old}$$
$$\delta_{new} \Leftarrow \vec{r}^T\vec{r} \quad (7)$$
$$\vec{d} \Leftarrow \vec{r} + \beta\vec{d} \quad (8)$$
$$i \Leftarrow i+1$$
$$end\ while$$

Fig 2. CG Pseudo Code

4.2. C code

Within the C code the following vectors and matrices are used when computing the CG:

- r - Residual vector
- b – Known vector
- d - Search direction vector
- x – starting/current step/result vector
- q - Known vector
- A - Known, symmetric, positive-definite matrix.
- ja – index array

Our implementation utilizes arrays stored in SRAM to hold the various vectors and matrices required for computing the CG. These arrays are N elements in size with the exception of A which is 7N in size.

Of these arrays, r, A, x must be transferred to the FPGA accelerator prior to executing the CG algorithm. The arrays, q and d are computed and modified on the FPGA accelerator itself. With the exception of the index array (ja), all values are 64bit IEEE Floating Point values.

The values of the error tolerance ε, and the maximum iterations i_{max} are scalar values passed to the FPGA. These values are used to control and limit the number of iterations the CG performs.

If we evaluate the inner loop operations (3) – (8) of the CG pseudo code, assuming that they are fully pipelined vector operations, we can estimate the run-time of a single iteration of the CG algorithm. In TABLE III we show the approximate number of clock cycles and floating-point operations (FLOPS) per execution of the inner CG loop.

Using the information from this table we can calculate the estimated best-case sustained performance of this code, if fully pipelined would be ~191 MFLOPs (using a clock cycle of 10ns).

Table III. Clocks and FLOPs

Function	Clock Cycles	DP FLOPs
(3) SMVM	7N	13N
(4) DDOT	N	2N
(5) DAXPY	N	2N
(6) DAXPY	N	2N
(7) DDOT	N	2N
(8) DAXPY	N	2N
Total	**12N**	**23N**

4.3. Sparse Matrix Vector Multiply (SMVM)

When computing the CG, the SMVM computation is the time dominant operation. Current microprocessor-based systems exhibit low computational efficiency when computing the SMVM [17]. The SMVM C-code fragment shown in Fig 3 illustrates why this occurs.

```
for (n=0; n<nrows; n++) {
  *Y = A[0]*X[ja[0]] +
       A[1]*X[ja[1]] + A[2]*X[ja[2]] +
       A[3]*X[ja[3]] + A[4]*X[ja[4]] +
       A[5]*X[ja[5]] + A[6]*X[ja[6]];
  A+=7; ja+=7;
  Y++;
}
```

Fig 3. SMVM C-code

Each Y requires thirteen DPFP operations: seven multiplies and six additions. For a single value of Y we need to read a total of 21 operands and write a single operand. Fifteen of these operands are 64-bit, DPFP values (A, X and Y). Seven of these values are 32-bit, integer values (ja[n]) used as the indirect index into the X array. The retrieval of a large number of operands coupled with a reasonably small amount of computation performed on each operand, places an enormous burden on the memory hierarchy and bandwidth of any computational system.

Compounding the memory bandwidth problem are the indirect memory references of the X vector. While there is some available data reuse, it is in general, too small to yield high-percentages of the available peak floating point rate offered by the CPU.

On general-purpose processors, as the problem size increases, the required data moves out of caches and into main memory. As the data propagates further out in the memory hierarchy, the performance drops off substantially. On the MAP processor, our implementation of SMVM uses OBM exclusively for all of its data, and has constant performance up to the maximum problem size that can fit in the OBM [4].

4.4. Motivation

Our prior work on SMVM on the SRC MAPStation [4] provided insights for the work presented in this paper. The first insight was that there is more than ample logic space available in a single user logic device of the MAP to implement a fully pipelined implementation of the SMVM. The second insight was that unless we can reduce the amount of time consumed transferring data to and from the MAP, there is little to no gain in performance using the FPGA based solution.

Our implementation of the SMVM on the MAP produced a design balancing the amount of logic instantiated to the amount of usable OBM bandwidth. By approaching the problem in this manner, a substantial amount of logic space remained on the user logic device for further use. This space coupled with the entire second user logic FPGA meant that we had plenty of space available for implementing the CG.

The data transfer time to and from the MAP consumed almost 25% of the overall processing time when computing the SMVM. By computing the entire CG on the MAP we eliminate almost this entire transfer penalty. Additionally, we can take advantage of other optimizations to improve the overall performance of the CG on the FPGA system.

5. Conjugate Gradient Implementation

In the CG pseudo code shown in Fig 2, all operations shown are implemented on the MAP with the exception of the initial computation of the residual vector (1). This computation requires both a SMVM operation along with a vector-vector subtraction. Substantial amounts of FPGA logic and routing resources are required to implement the SMVM. Since this operation is only performed once during the computation of the CG, we assigned it to the CPU with the resulting residual vector ("r") being transferred to the MAP during the CG function call.

The vector update functions (DAXPY 5, 6, and 8) will generate OBM conflicts when attempting to pipeline these functions. The conflicts are due to the required read/write operations to the same array. To avoid this conflict we introduce an extra set of arrays (odd, even), placed in separate OBMs, and ping-pong between them during computation.

Since the SMVM (3) is the time dominant kernel executed during each iteration of the CG it is important to maximize its performance. The SMVM code shown in Fig 3 computes the result by forming 7 partial products and accumulating them.

There is sufficient logic within a single U_LOGIC device to compute a single SMVM result per clock. Unfortunately, there is not enough aggregated memory bandwidth available on the MAP to support this.

A naïve implementation would simply place A, d, and ja in three separate OBM banks and then utilize a single SRC 64-bit multiply accumulate macro to generate the result. This approach would require a nested set of loops similar to what is shown in Fig 4. Since the SRC compiler only pipelines inner loops, inner loops with small numbers of iterations lead to poor performance. This performance degradation is due to the pipeline being stopped and started for each iteration of the outer loop.

```
for (n=0; n<nrows; n++) {
   offset = n*7;
   accum = 0;
   for (j=0; j< 7; j++) {
     accum += A[offset+j]*d[ja[offset+j]];
   }
   q[n] = accum;
}
```

Fig 4. MAC Implementation

Our improved SMVM design computes two partial products in parallel, accumulating the result as required. In this design we stripe the A array across two consecutive OBM banks (AL0 and AL1), replicate d in two OBM banks ({dodd0, dodd1} or {deven0, deven1}), store ja array in another OBM bank (jaL), with the result, q, being stored in a separate OBM bank (q). The reason for replicating the d array is to allow for two indirect accesses (based on the jaL values) to the d array without contention. With this array placement we can access two A, d, and ja values per clock. These values are then used to form the partial products that are accumulated to form the result.

The design utilizes a special "user" macro that was specifically designed by SRC for us for this problem. This macro implements a dual Accumulator configuration producing results every 3rd and 6th clock cycle using the higher accuracy M version macros.

Rather than utilize a nested loop structure, which would prevent the SRC compiler from pipelining the SMVM operation, we utilize the SRC cg_count_ceil_32 counter macro to handle the inner loops sequencing. An outer for loop simply ensures that we compute for the proper number of results. The actual sequencing is handled by the SRC cg_count_ceil_32 counter macros. These macros are used to flatten nested loops.

The optimizations described above yield an improvement in run-time from the 7N to 3.5N clock cycles. This solution produces 2 results every 7 clock cycles (70 nsec) for a sustained performance of 371 MFLOPs.

Using our improved SMVM as our starting point we now analyze the operations listed in Fig 2 to identify which operations can be computed concurrently. Once these operations are identified we then group the computations if we can place the required operands and results in separate OBMs so there are no conflicts when pipelining the operations.

The first grouping we implement is the computation of the SMVM (3), and the vector DDOT (4). The computation of the DDOT (4) requires that we have access to the q, and d arrays. Since both arrays must be accessed when computing the SMVM, we combined these operations into a single loop reducing the overall run-time another N clock cycles. Therefore, by optimizing the SMVM (3) and fusing the DDOT (4) within the SMVM loop, we have reduced the run time from 8N to 3.5N clock cycles.

The second grouping we implement is the computation of the vector updates (DAXPYs) (5), (6) and the DDOT in (7). By combining these separate operations into a single loop with parallel functional units, we are able to reduce the run-time for these computations from 3N to N clock cycles.

The final vector update (DAXPY) (8) cannot be grouped with any other operations since it is dependent on the scalar value β and the updated r array. This loop adds another N clock cycles to the overall run-time of the CG.

To ensure that there were no OBM conflicts causing slowdowns within the inner loops, we distributed the arrays as shown in Table IV. The arrays with the odd/even designations are the ping-pong arrays described above.

Table IV. OBM Array Placement

A	B	C	D	E	F	G	H
AL0	AL1	dodd0	deven1	jaL	q	deven0	dodd1
xodd	reven		xeven	rodd			

In this design we stripe the A matrix across two consecutive OBM banks. Since our memory layout requires that a second array of order N reside in the same OBM banks, we can calculate the maximum problem size (based upon N) as follows (SRC reserves 512 locations of OBM for system purposes):

$$MAX\_OBM\_SIZE = 512*(1024) - 512 = 523,776$$
$$\frac{7N}{2} + N = 523,776$$
$$\frac{9N}{2} = 523,776$$
$$N \approx 116,394$$

The U_LOGIC Xilinx XC2V6000 devices contain 144 Block RAM (BRAM) units, with each unit holding 2048 bytes. This yields an internal high-speed memory capacity of 294912 bytes. While in many cases this amount of BRAM would be useful, for our problem space it was not large enough. For example, storing just the vector x in BRAM would require $8*48^3 =$ 884736 bytes. Problems small enough to utilize BRAM for storage also must compete against microprocessor based systems operating entirely out of L1/L2 cache where any benefits from utilizing the FPGA approach are greatly reduced or completely negated.

Striping A has the advantage of placing consecutive values in separate independent memories. By striping A in this fashion we are also able to exploit the full DMA bandwidth of the SRC system (1400MB/s) when transferring the A array from the PC to the MAP OBM thereby reducing our transfer overhead.

There are other ways we could accelerate the computation of the various vector functions (DDOT, DAXPY) such as loop bisection or loop unrolling but these methods would require that the arrays be stored in non-optimal ways for future computation. Our design leverages the accelerated SMVM and then extracts a large amount of functional parallelism. The requirements of the vector updates force us to replicate arrays and eat up storage but this would be a problem even if alternative approaches were undertaken.

In this design we utilize both user logic FPGA devices on the MAP. This is required since the amount of logic required to implement the complete CG exceeds that available within a single device.

The partitioning of the pseudo code in Fig 2 was chosen to enable the concurrent computations and optimizations we identified. The pseudo code for the devices can be found in Fig 5 (Primary Device) and Fig 6 (Secondary Device).

The primary device is responsible for issuing all DMAs, and controls all access to OBM. Prior to executing the CG on the MAP the arrays A, r, ja, and x must be transferred to the MAP OBM. Our design makes use of both the DMA macros and the Streams feature of the SRC system. The A, ja, and x arrays are all transferred using the DMA macros directly into OBM. We utilize the DMA striping feature to distribute A across two consecutive OBM banks.

The streams functionality of the SRC system is used to transfer the r array to the MAP. By utilizing streams we are able to accomplish several operations in parallel (1). Our implementation of the SMVM requires that we have two, independent, copies of the d array. Since d is a copy of the r array we are able to stream in a value of r, and then place it into the three separate OBM banks simultaneously (r and two copies of d). In effect we get two free copies of the array d. The initial vector norm is also easily calculated in the streaming loop since we have access to the r values as they are streamed. This is simply done using an SRC MAC macro.

The secondary device has no DMA capabilities and can only access OBM when granted permission from the primary device. The pseudo code shown in Fig 6 describes the operations performed by the secondary device.

The primary device passes the maximum iteration limit and convergence value to the secondary device prior to entering the main CG loop. The secondary device will then enter its main CG loop and wait on the primary device to signal it to proceed with its calculations. The primary device controls access to the OBM via the send_perms macros. The handshaking between the devices is done using the send_to_bridge/recv_from_bridge macros.

```
DMA stripe A ⇒ AL0, AL1
DMA ja ⇒ jaL
DMA x ⇒ xeven
STREAM DMA r ⇒ { reven, deven0, deven1, δ_new ⇐ r^T r }   (1)
δ_0 ⇐ δ_new
i ⇐ 0
max_iter ⇐ max_iteration
converge_value ⇐ ε^2 δ_0
send_to_bridge(max_iter,
               converge_value,
               nrows)
recv_from_bridge() // handshake
While i < max_iter and δ_new > convergence_value
   { q ⇐ A d_even_odd
     α_accum ⇐ d_even_odd^T q }   (2)
   α ⇐ δnew / α_accum
   send_perms(ALL_OBM)
   send_to_bridge(α, δ_new)
   recv_from_bridge(δ_new)
   send_perms(0)
   i ⇐ i + 1
end while
DMA x ⇐ xeven or xodd
DMA r ⇐ reven or rodd
```

Fig 5. Primary FPGA

```
recv_from_bridge(max_iter, converge_value, nrows)
send_to_bridge() // send ACK
i ⇐ 0
δnew ⇐ 1.0 // initial value
While i < max_iter and δnew > converge_value do
    recv_perms() // memory access
    recv_from_bridge(α, δnew) // memory access
    { xodd_even ⇐ xeven_odd + α deven_odd
      rodd_even ⇐ reven_odd − α q
      raccum ⇐ r̄even_odd^T reven_odd }   (3)
    δold ⇐ δnew
    δnew ⇐ raccum
    β ⇐ δnew / δold
    dodd_even ⇐ reven_odd + β deven_odd   (4)
    send_to_bridge(δnew)
    recv_perms()
    i ⇐ i+1
end while
```

Fig 6. Secondary FPGA

For both devices, operations on arrays defined with the underscore "even_odd" indicate we are working with ping-pong arrays. The inner-loop function being computed will utilize the outer CG iteration count "i" to determine whether to obtain source operands from the "even/odd" array, then write the result to the "odd/even" array.

The results of our optimizations are summarized in Table V. This CG implementation reduces the overall run-time from the original 12N clock cycles to only 5N clock cycles. Through careful array placement in the OBM and the implementation of multiple concurrent operations, this implementation supports a sustained rate of 418MFLOPs for the computation of the CG. This rate includes the required transfers and computations required to execute the entire CG problem entirely on the MAP.

Table V. Optimized CG

Function	**Clock Cycles**	**DP OPs**
(3) SMVM+(4) DDOT	3.5N	15N
(5,6) DAXPYs+(7) DDOT	N	6N
(8) DAXPY	N	2N
Total	**5.5N**	**23N**

Constraints due to locked I/O, limited internal routing, and required operational frequency (100 MHz) come into play as FPGA designs become larger and more complex. These constraints impact the place and route software, making this phase of the design flow slow down considerably. In our case the run-time for the compilation of the two user devices is on the order of hours.

Relaxation of these constraints, such as compiling the devices without locked I/O, allows the place and route software much more flexibility when dealing with macro placement and routing. This can yield better operational frequencies for many operations but is unrealistic for a real, physical design where the OBM, FPGA placement, and I/O connections are fixed and static.

The place and route results for the two user logic FPGA devices are displayed below in Table VI and Table VII. In both cases the user logic devices successfully pass place and route, meeting all constraints. For both devices the clock period constraint was set at 10 ns, and the final place and route result was 9.9995 ns, meeting the requirements. The number of slices utilized in the primary device is 91% which is very high and contributes to long place and route times.

Table VI. Primary FPGA

Device Utilization Summary:		
Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	819 out of 1104	74%
Number of LOCed IOBs	819 out of 819	100%
Number of MULT18X18s	100 out of 144	69%
Number of RAMB16s	9 out of 144	6%
Number of SLICEs	31055 out of 33792	91%

Table VII. Secondary FPGA

Device Utilization Summary:		
Number of BUFGMUXs	1 out of 16	6%
Number of External IOBs	734 out of 1104	66%
Number of LOCed IOBs	734 out of 734	100%
Number of MULT18X18s	64 out of 144	44%
Number of RAMB16s	6 out of 144	4%
Number of SLICEs	19186 out of 33792	56%

6. RESULTS

Every MAP function has a free-running 64-bit counter which increments on each clock tick. This counter begins running as soon as the FPGA has been configured. All timing measurements utilize this counter. All measurements include both the data transfer DMA timings for all required data along with the actual processing times.

For our measurements we have set the error tolerance to an extremely low value keeping the CG MAP function from converging early, forcing it to run the full 5000 maximum iterations. To generate the results we executed 10 runs for each problem size of N, and then averaged the results. The results are shown in Fig 7 below.

For comparative numbers, we generated a version of the non-preconditioned CG code to run on various general purpose machines. The following lists the specifics of the systems:

- Xeon 2.8GHz (SRC). SRC MAPStation with 2GB system memory. Linux Fedora Core 4, Intel C/C++ compiler.
- Opteron 2.6GHz (Asus). ASUS K8N-DL server motherboard with 4GB system memory. Windows XP Pro, Microsoft Visual Studio 2005.
- Opteron 2.6GHz (Tyan). Tyan S2895 server motherboard with 4GB system memory. Linux Fedora Core 4, gcc compiler.
- EMT64. HP xw8200 dual Intel 3.6 GHz EMT-64 Workstation with 2GB system memory. Linux SUSE 9.2, gcc compiler.

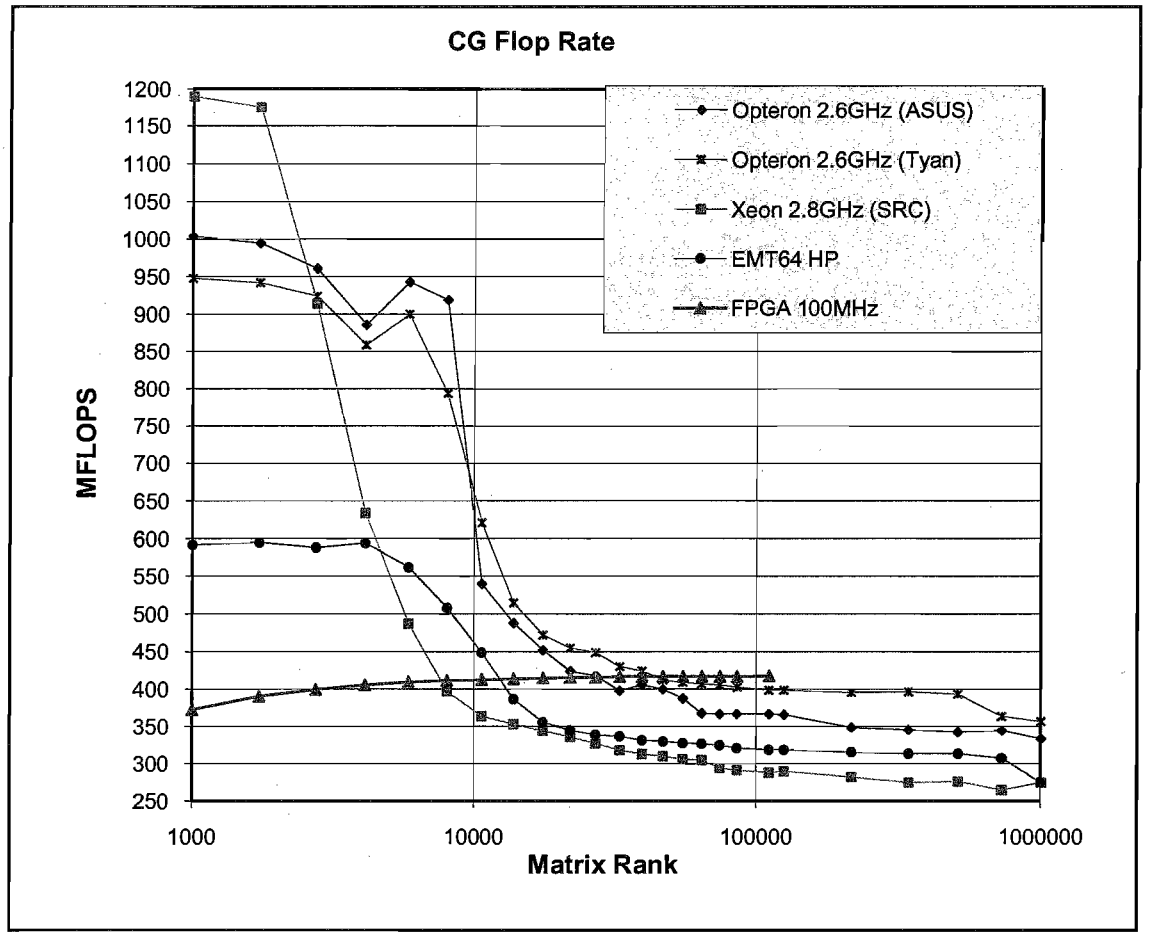

Fig 7. Performance comparison (FPGA vs. general purpose processors) as a function of problem size

For the FPGA based CG implementation we can see that for smaller problem sizes, the effects of the processing pipelines (fill and empty) have an effect on its processing rate. The Transfer Rate numbers are also lower than expected for the smaller problem sizes. This may be due to the overhead associated with setting up the DMA operations vs. the small transfer size.

For problem sizes where $N \approx 8,000$ and greater, we see that the MAP will attain close to its maximum sustained processing rate of 418 MFlops for the CG. All the CPU based systems perform reasonably well for small problems sizes since the requisite data structures can fully reside in cache. The FPGA based implementation outperforms all of the CPU based systems for problem sizes larger than N = 46,656 and offers deterministic performance.

The performance of our FPGA implementation is restricted by the size of the OBM which limits the maximum problem size our implementation can handle (i.e., a maximum matrix rank of 110,592). The other limiting factors are the amount of usable memory bandwidth, the amount of available logic in the U_LOGIC devices, and the clock speed of the MAP processor. We would see nearly a 2X performance increase if enough logic were available to generate an SMVM result each cycle instead of every 3.5 cycles. Unfortunately, it would also require a 3.5X increase in memory bandwidth to achieve this.

7. FUTURE WORK

SRC Computers has recently announced their next generation Reconfigurable Computing System the SRC 7. It introduces the H series MAP which utilizes FPGAs from Altera (EP2S180). The OBM SRAM (QDR) has been redesigned to support up to 16 simultaneous references (reads or writes) and the nominal User Logic speed is expected to increase to 150 MHz. The System Interconnect bandwidth has been increased from 2.8 GB/s to 14.4 GB/s. Two simultaneously accessible banks of DDR2 SDRAM (up to 1GB) have been added which allow for much larger local high speed storage. The maximum rated power consumption for the H series MAP is 80 watts. A diagram of the SRC-7 H series MAP is shown in Fig 8.

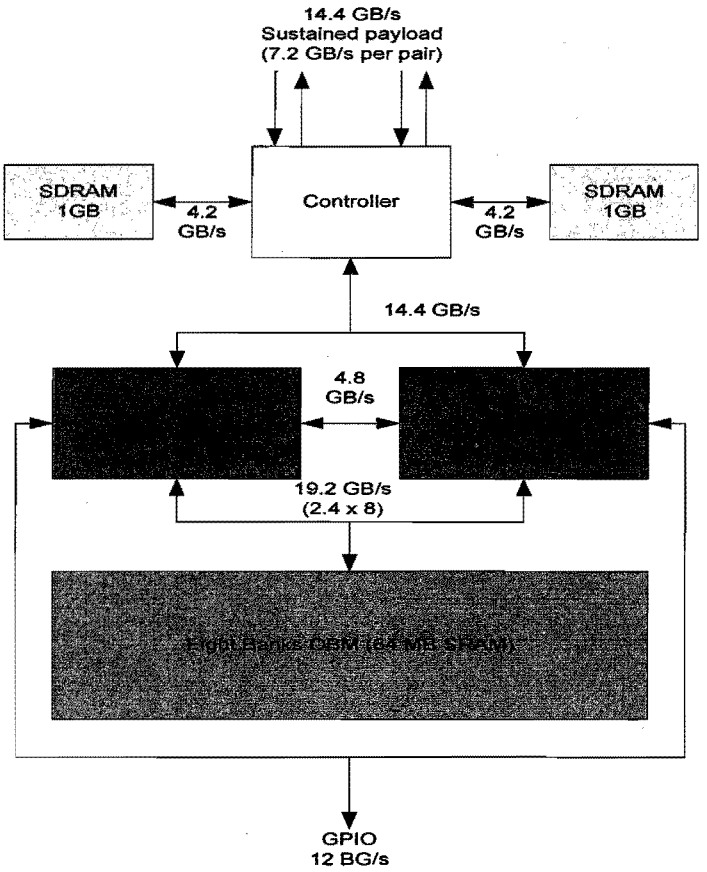

Fig 8. SRC-7 H Series MAP

We are looking at porting our work directly to the SRC-7 where we should see an immediate processing speed up of 50% due to the increase in operating frequency. The ~5X increase in the System Interconnect bandwidth should yield a substantial reduction in the DMA transfer time penalty. In Fig 9 we show projected results for a direct port of the CG code to the SRC-7 based upon the projected improvements in clock rate and bandwidth.

With restructuring of the code we should be able to improve on the performance of the SMVM by computing 4 product terms per clock thereby doubling the performance of the SMVM. This alone would have significant positive impacts when computing the CG. With the additional aggregate OBM bandwidth (SDRAM and SRAM) we anticipate that the CG performance could be accelerated at least 2X compared to the SRC-6 version without much difficulty.

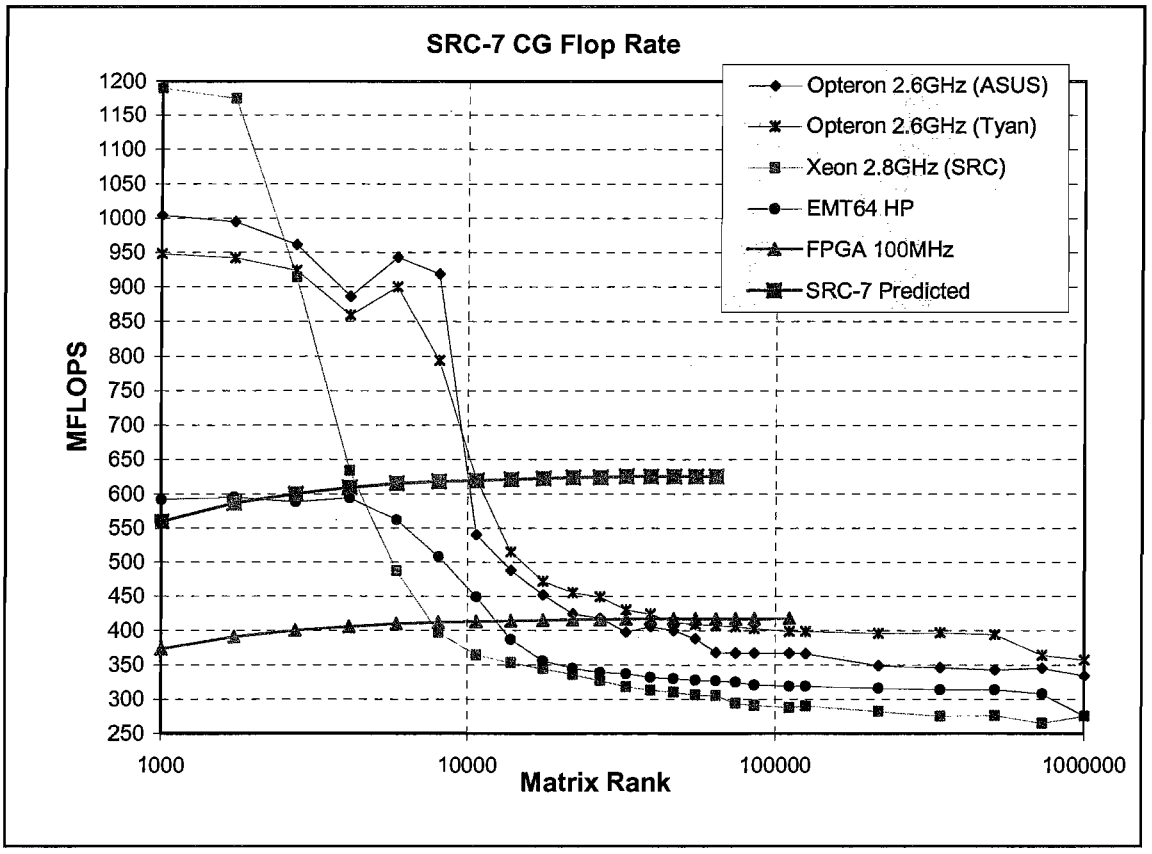

Fig 9. SRC-7 Projected CG Flop Rate with direct porting of code

8. Conclusion

We have presented a design that implements an efficient CG algorithm on the SRC MAPStation. We have demonstrated that an FPGA based system can outperform general purpose processors while running over 30 times slower (i.e. 100 MHz vs. 3.4 GHz). This is possible because an FPGA-based system can be designed to more optimally match the computational units to available memory bandwidth providing a more balanced system. It also provides parallelism not available in any standard microprocessor design and allows unique functionality such as vector replication in a single clock cycle.

FPGAs and CPUs both suffer from the basic physical constraints of limited I/O and limited memory bandwidth for this and other memory bandwidth intensive classes of problems. To efficiently utilize the peak computational capability of FPGA or CPU based systems for this class of problems requires tremendous amounts of memory bandwidth.

Since any useful problem implemented on an FPGA based system will need to transfer data from the SCM to the FPGA's OBM, and since this transfer time impacts overall performance, we have exploited a simple memory copy DMA mechanism SRC provides to minimize this data transfer time.

The results we have presented are deterministic and will scale directly with any improvements in the system user logic frequency, memory bandwidth and/or memory depth. The recently release SRC-7 should directly be able to increase the performance of the CG algorithm presented in this work. With modifications to our code, it is not unreasonable to expect performance increases on this platform to reach 2X those of the SRC-6.

ACKNOWLEDGMENTS

We would like to thank Ken Koch of Los Alamos National Laboratory for supplying SMVM code which included a matrix initialization and test environment.

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness. This paper is published under LA-UR 08-[redacted].

REFERENCES

[1] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Ven der Vorst, H., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994, Philadelphia, PA.

[2] Cray Inc., Cray XD1 Supercomputer Product Page. 2006. < http://www.cray.com/products/xd1/index.html>

[3] D'Azevedo, E.F, Fahey, M.R, Mills, R.T., Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *Proceeding, Part I of Computational Science – ICCS2005 5th International Conference*, Atlanta, GA, USA, May 2005, V.S. Sunderam , G.D. van Albada, P.M.A. Sloot, J.J Dongarra, Eds, Springer, New York, NY, 99-106.

[4] DuBois, D., DuBois, A., Davenport, C., Poole, S., "Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer," LANL, LA-UR-06-5312, August 2006.

[5] Fettig, Kwok, Saied. "Scaling Behavior of Linear Solvers on Large Linux Clusters," National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign, 2002.

[6] Guo, Z., Najjar, W., Vahid, F., and Vissers, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 22 - 24, 2004). FPGA '04. ACM, New York, NY, 162-170. DOI= http://doi.acm.org/10.1145/968280.96830

[7] Ilse, Ipsen and Meyer, "The Idea Behind Krylov Methods," American Mathematical Monthly, volume 105, number 10, pages 889-899, 1998.

[8] Intel Corporation. Intel Compilers for Linux Application Development, 2005. <http://www.intel.com/cd/software/products/asmona/eng/compilers/219760.htm>

[9] Maslennikow, O., Lepekha, V. and Sergyienko, A. 2005. "FPGA Implementation of the Cojugate Gradient Method," Lecture Notes in Computer Science, volume 3911/2006, pages 526-533, 2006.

[10] Mills, R.T., D'Azevedo, E.F., and M.R. Fahey., "Progress Towards Optimizing the PETSc Numerical Toolkit on the Cray X1," Cray Users Group, May, 2005. Available: http://www.ccs.ornl.gov/~rmills/pubs/cug2005.pdf

[11] Morris, G. R., Prasanna, V. K., and Anderson, R. D. 2006. A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 24 - 26, 2006). FCCM. IEEE Computer Society, Washington, DC, 3-12. DOI= http://dx.doi.org/10.1109/FCCM.2006.8

[12] See Wikipedia, *Field-programmable gate array*, June 20, 2006, <http://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=58629320>

[13] SGI, SGI RASC Technology Product Page. 2006. < http://www.sgi.com/products/rasc/>

[14] Shewchuk, J. R. 1994 *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Technical Report. UMI Order Number: CS-94-125., Carnegie Mellon University.

[15] SRC Computers, Inc. Product Page, July, 1999-2008. <http://www.srccomputers.com/products/products.asp>

[16] SRC Computers, Inc. SRC C Programming Environment v2.1 Guide. SRC Computers, Inc. August 31, 2005.

[17] Toledo, S., "Improving Memory-System Performance of Sparse Matrix-Vector Multiplication," *IBM Journal of Research and Development*, 41(6):711-725, 1997.

[18] Underwood, K. 2004. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 22 - 24, 2004). FPGA '04. ACM, New York, NY, 171-180. DOI= http://doi.acm.org/10.1145/968280.968305

[19] Zhuo, L. and Prasanna, V. K. 2005. Sparse Matrix-Vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA, February 20 - 22, 2005). FPGA '05. ACM, New York, NY, 63-74. DOI= http://doi.acm.org/10.1145/1046192.1046202