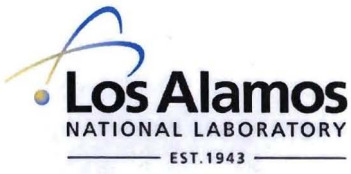
LA-UR- 09-01455

Approved for public release; distribution is unlimited.

Title:	Non-Preconditioned Conjugate Gradient on Cell and FPGA Based Hybrid Supercomputer Nodes
Author(s):	David DuBois Andrew DuBois Thomas Boorman Carolyn Connor
Intended for:	FCCM 2009 (April 2009, Napa, CA)



Form 836 (7/06)

Non-Preconditioned Conjugate Gradient on Cell and FPGA based Hybrid Supercomputer Nodes

David DuBois, Andrew DuBois, Thomas Boorman, Carolyn Connor
Los Alamos National Laboratory
dhd@lanl.gov, ajd@lanl.gov, tmb@lanl.gov, connor@lanl.gov

Abstract

This work presents a detailed implementation of a double precision, Non-Preconditioned, Conjugate Gradient algorithm on a Roadrunner heterogeneous supercomputer node. These nodes utilize the Cell Broadband Engine Architecture™ in conjunction with x86 Opteron™ processors from AMD. We implement a common Conjugate Gradient algorithm, on a variety of systems, to compare and contrast performance. Implementation results are presented for the Roadrunner hybrid supercomputer, SRC Computers, Inc. MAPStation SRC-6 FPGA enhanced hybrid supercomputer, and AMD Opteron only. In all hybrid implementations wall clock time is measured, including all transfer overhead and compute timings.

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 [7].

CG uses successive approximations to obtain a more accurate solution at each step. It is considered a non-stationary 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 [2].

Each 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 kernel executed per iteration of the CG [3] [6] [21].

For general purpose processors, the SMVM performs poorly for three primary reasons [24]. 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.

In general the vector-vector (DOT, DAXPY) and vector-matrix (SMVM) operations utilized during the computation of the CG exhibit poor floating point utilization. This is due to the high application Bytes/Flop requirements when compared to the processor supplied Bytes/Flop [6][25].

Sparse Linear Algebra (e.g. CG/SMVM) has been identified as a key computational focus area or "Dwarf" (algorithmic method that captures a pattern of computation and communication), for evaluating parallel programming models and architectures. Asanovic et al. [1] recommend the use of a set of identified "Dwarfs" instead of traditional benchmarks.

Implementation of CG using FPGAs has been documented in recent papers. Morris and Prasanna [20] have presented an FPGA-augmented implementation of the CG on an SRC-6. Their implementation presents a CPU/FPGA accelerated hybrid approach. Maslennikow et al. [18] present an FPGA implementation of CG using fractional numbers which is limited to small problem sizes with matrix rank up to 1024.

Williams et al. [26] evaluated the Cell Processors for use in Scientific Computing. They introduce a performance model for the Cell and apply it to several key scientific computing kernels, including sparse matrix multiply and stencil computations. Li et al. [16]

present a implementation of the NAS CG benchmark on the Cell Processor.

In this work we present a comparison of various architectures on a common, non-preconditioned, CG algorithm. We acknowledge that preconditioning is of paramount importance for efficient implementations of iterative methods such as the CG; however, our focus is to compare the per-iteration performance of the CG algorithm on these platforms, not the efficacy of the preconditioner.

The platforms we investigate include two hybrid supercomputers nodes: IBM/LANL Roadrunner TriBlade [15], SRC-6 MAPStation [22], along with traditional AMD Opteron nodes.

The TriBlade consists of an AMD Opteron blade and two Cell QS22 blades[11]. The Opteron blade contains two dual-core processors, while the Cell blades each contain two Cell eDP (enhanced Double Precision) processors. Each Opteron core is connected to an individual Cell chip via a dedicated PCIe connection. For this work we utilize a single Opteron/Cell eDP pair, or 1/4 of the available raw compute performance supplied by a single TriBlade.

The MAPStation utilizes Intel Xeon Processors along with a SRC MAP processor which contains two user logic Xilinx FPGAs [22]. Carte [22][23], SRC's Programming Environment is used giving the programmer access to the user programmable logic of the MAP processor and the microprocessor through a single C or Fortran program.

The Opteron only implementation utilized a Hewlett Packard HP xw9400 workstation [8]. The system utilized Microsoft Vista as the base Operating System along with Microsoft Visual C++ 2008 for development.

All implementations utilize a banded sparse matrix with up to 7, double precision, elements per row. If any row contains fewer than 7 nonzero elements it must be padded with zeros to the full 7 elements in length. Due to alignment restrictions with the Cell processor an extra element of zero padding is required.

Due to the small, fixed number of elements per row the sparse matrix ELLPACK-ITPACK [19] format was chosen. This format is efficient for the matrix-vector multiply operation and performs well on vector style architectures.

Each of the implementations presented in this paper make use of loop unrolling, and loop fusion whenever possible. This helps with cache reuse in the processor only case and allows for more efficient data layout, management, and vectorization in the other cases.

Our FPGA based implementation makes heavy use of architectural features provided by both the SRC-6 hardware architecture and the Carte Software development environment [22][23]. Concepts presented here can carry over to other FPGA based systems but the actual implementation presented here is specific to the SRC-6 MAPStation. Our previous work on SMVM and CG for the SRC-6 MAPStation provides details of the FPGA implementation and results [4] [5] [6].

2. Background

2.1. CG and ELLPACK-ITPACK

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 [14]. 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.

$$A\_matrix = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 \\ 6 & 7 & 0 & 0 \\ 8 & 0 & 0 & 0 \end{bmatrix} A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 5 \\ 6 & 7 & 0 \\ 8 & 0 & 0 \end{bmatrix} ja = \begin{bmatrix} 1 & 3 & 0 \\ 1 & 2 & 4 \\ 1 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Figure 1. ELLPACK-ITPACK Format

In Figure 1 we illustrate how a simple 4x4 matrix is represented in ELLPACK-ITPACK format. The non-zero elements of A_matrix are packed starting from left to right to generate the coefficient array A. The original column indices are then stored in the column index array ja. With ELLPACK-ITPACK, the row or rows with the maximum number of non-zero elements determines how many elements must be stored per row in the compressed format. In this example all rows with fewer than 3 non-zero entries are zero-filled to the full 3 elements per row.

2.2. IBM/LANL Roadrunner Hardware

On June 10, 2008 Roadrunner became the first general purpose system to reach the PetaFlop (PF) milestone becoming the world's fastest supercomputer [15]. This petascale computer is unique in that it leverages high-performance commodity processors (Cell) to achieve extremely high levels of performance and excellent power efficiency [17].

Roadrunner is a heterogeneous cluster of clusters, each of which is Cell accelerated. Each compute node is composed of node-attached Cells, rather than a simple cluster of Cells. The fundamental building block is a Connected Unit (CU). Each CU is composed of 180 compute nodes and 12 I/O nodes all connected via a high speed switch fabric. The full Roadrunner system is composed of 18 CUs.

The TriBlade is the fundamental building block for each CU. Each TriBlade consists of an AMD Opteron blade along with two Cell QS22 blades.

In all, the Roadrunner system is made up of 6,500 AMD dual core Opteron processors, 12,240 Cell processors with a total peak (theoretical) performance in excess of 1.3 PFs. A total of 98 TeraBytes of memory is equally distributed between the Opteron and Cell nodes of the system.

2.2.1. TriBlade

A TriBlade is composed of an IBM LS21 Opteron Blade, two IBM QS22 Cell Blades, and a forth blade which provides the communications fabric for the computer node. The forth blade connects each QS22 blade through four PCI Express x8 links to the Opteron blade and provides the node with an Infiniband 4x DDR cluster interconnect.

2.2.2. IBM BladeCenter QS22

The IBM BladeCenter QS22 utilizes the IBM PowerCell^{TM} 8i processor. The following summarizes the capacities of the QS22:

- Two 3.2 GHz IBM PowerXCell 8i processors
- Up to 32 GigaBytes (GB) of PC2-6400 800 MHz DDR2 Memory
- 460 (peak) single-precision gigaflops per blade
- 217 (peak) double-precision gigaflops per blade
- IBM Enhanced I/O Bridge chip

2.2.3. Cell Broadband Engine Architecture (PowerXCell 8i)

The Cell Broadband Engine Architecture (CBEA) is a single-chip multiprocessor [20]. Nine processing elements operate on a shared, coherent memory as shown in Figure 2. Unlike current homogeneous multi-core solutions, the CBEA utilizes a heterogeneous configuration consisting of two types of computing elements: the PowerPC Processing Elements (PPE) and the Synergistic Processor Element (SPE, Figure 3). A single CBEA processor contains one PPE and eight SPEs.

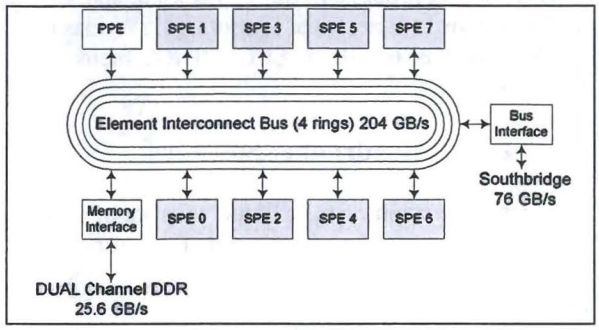


Figure 2. Cell Broadband Engine Architecture

The PPE is a 64-bit PowerPC architecture core and can run both 32-bit and 64-bit Operations Systems (OS) and applications. SPEs are optimized for running SIMD applications, and operate as independent processor elements, each running an individual application program or threads. In this configuration, the PPE provides OS support and top-level thread control for an application while the SPEs provide the accelerated application performance.

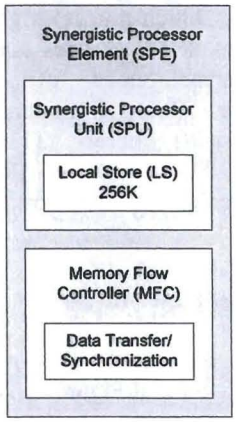


Figure 3. Synergistic Processor Element (SPE)

The SPEs access memory via Direct Memory Access (DMA) commands moving data and instructions between main storage and a private local memory called Local Storage (LS). All SPE instruction and data load/store requests access this private LS rather than shared main storage. This memory hierarchy of storage (register file, LS, main storage), coupled with asynchronous DMA transfers between LS

and main storage, explicitly parallelizes computation with the transfers of data and instructions.

2.2.4. IBM BladeCenter LS21

The IBM BladeCenter LS21 supplies up to two dual-core 2200 series AMD Opteron processors in a single width card [10]. For the Roadrunner system the AMD Opteron HE processors run at 1.8 GHz and are standard low-power Opteron processors (68 W max). Each LS21 contains 16 GB of ECC, DDR-2 memory and no hard disk.

2.3. IBM/LANL Roadrunner Software

The compute portion of a TriBlade consists of two QS22 boards and a single LS21 board. Each runs its own operating system image and "shares" a common user application.

Applications written and executed on the Roadrunner system are designed and written in a different manner than previous parallel processing applications.

The majority of a user application runs on the AMD Opteron processors of the LS21. Message Passing Interface (MPI) is used to communicate with other processors in a typical Single Program, Multiple Data (SPMD) fashion. Computationally-complex logic is offloaded to a "subordinate" Cell processor when needed.

Key to obtaining performance on the Roadrunner system is determining which processes get off loaded to the Cell processors. IBM provides two techniques for performing asynchronous offloads. These techniques are the Data Communication and Synchronization (DaCS) library [13] and the Application Library Format (ALF) [9]. The CG implementation presented in this work uses DaCS exclusively.

3. Implementation Detail

All implementations utilize the CG implementation outlined in the pseudo-code of Figure 4. The following vectors and matrices are used:

- r – residual vector
- b- known vector
- d –search direction vector
- x – initial guess/current step/result vector
- q – temporary vector
- A – known, sparse, symmetric, positive-definite matrix
- ja- column index array (not explicitly shown in Figure 4)

The value ε is an error tolerance, where $\varepsilon < 1$ and should be set so that the algorithm terminates when $\|r_{(i)}\| \le \varepsilon \|r_{(0)}\|$.

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

Figure 4. CG Pseudo-Code

A synthetic sparse system indicative of a 3D regular mesh using a 7 point stencil was used for testing. This test system contains a reasonable amount of spatial and temporal locality so that it doesn't unfairly bias the Cell, FPGA, or Opteron implementation.

The sparse matrix has a fixed structure and all implementations take advantage of loop unrolling when computing the SMVM. The C-code in Figure 5 illustrates the unrolling.

```
for (n=0; n<nrows; n++) {
*q=     A[0]*d[ja[0]] + A[1]*d[ja[1]] +
        A[2]*d[ja[2]] + A[3]*d[ja[3]] +
        A[4]*d[ja[4]] + A[5]*d[ja[5]] +
        A[6]*d[ja[6]];

        A+=8; ja+=8; // For Cell
        A+=7; ja+=7; // Others
        Y++;
}
```

Figure 5. SMVM C-code fragment illustrating loop unrolling and the associated memory access burden

All implementations take advantage of fused loops whenever possible. The first sets of fused loops are the computation of the SMVM (3) and the dot product

calculation (4). We acknowledge that in the general case this optimization would not normally be possible, but for the known structure used in this problem we chose to exploit it and did so across all implementations reported and compared herein. The second sets of fused loops are the calculation of the dot product for the δ_{new} value (7) and the update of the direction vector (8).

For the FPGA implementation substantial logic resources are required to implement the SMVM. As such, it was decided to compute the initial SMVM operation (1) on the Xeon processor of the MAPStation. For this reason, all implementations report the wall clock runtime after this operation.

For testing purposes the error tolerance value (ε) was set so that the problem would not converge to a solution allowing us to run for the full number of iterations requested. All implementations take the value of the system rank and i_{max} as input parameters allowing control over the synthetic system size and the number of iterations to execute.

To compute the performance of the CG for an input n x n matrix size we utilized Table I. In this table, we break down the number of Double Precision Floating Point Operations required per vector/matrix operation listed in Figure 4.

Table I

Function	DP FLOPs
(3) SMVM	13n
(4) DDOT	2n
(5) DAXPY	2n
(6) DAXPY	2n
(7) DDOT	2n
(8) DAXPY	2n
Total	**23n**

3.1. Cell Implementation

The Cell implementation focused on moving as much of the vector-vector and vector-matrix processing down to the SPE as possible. The SPEs operate as function accelerators for the PPE. All functions, SMVM, DOT, NORM, and DAXPY are fully implemented by the SPEs. The problem is evenly divided among the requested number of SPE's with each SPE processing a contiguous block, where the block size is the system rank divided by the number of SPEs. The PPE handles the execution flow and sequencing.

The SPE's, once started, enter an event loop waiting for function requests from the PPE. These requests, along with the required parameters, are all passed via the mailbox communication mechanism. To reduce the function call overhead the SPE vector functions are executed inline. The SPEs return results via the mailbox communication mechanism.

Careful attention was paid to the general SPE programming tips IBM has published [9]. The CG implementation specifically utilized the following recommendations:

- Local Store: Design for the local store (LS) size. The LS holds up to 256 KB for program, stack, local data structures, and DMA buffers.
- DMA Transfers:
  - Use SPE-initiated DMA transfers.
  - Overlap DMA with computation by double buffering.
  - Use double buffering to hide memory latency.
- Loops: Unroll loops to reduce dependencies and increase dual-issue rates. This exploits the large SPU register file.
- SIMD Strategy
- Load/Store:
  - Scalar loads and stores are slow, with long latency.
  - SPUs only support quadword loads and store.
  - Load or store scalar arrays as quadwords, and perform your own extraction and insertion to eliminate load and store instructions.
- Branches: Eliminate nonpredicted branches.
- Multiplies: Keep array elements sized to a power-of-2 to avoid multiplies when indexing.
- Dual-Issue:
  - Choose intrinsic carefully to maximize dual-issue rates or reduce latencies.
  - Use software pipeline loops to improve dual-issue rates.

A primary concern with implementing the CG on the CBEA is how to effectively compute the SMVM. The limited size of each SPE Local Store (LS) makes it impossible to store the source vector locally. Since we impose no limitation on the structure of the sparse matrix, the indirect addressing of the source vector must be dealt with if reasonable performance is to be achieved.

Several approaches were tried with limited success. The first was a direct implementation of a gather on the elements of the source vector using the Memory Flow Controller (MFC) DMA lists. While this implementation has the benefit of being direct and easily realized, the performance was poor. The overhead of setting up DMAs for individual double precision elements is extremely high. This method also

suffers from not allowing for reuse of previously gathered items.

The preferred implementation utilized a software-managed cache. Two different cache implementations were tested. The first was our purpose-designed software cache with the second being an implementation supplied by IBM in the Cell Broadband Engine SDK Libraries starting with Version 2.1.

Both software-managed cache solutions were useful in boosting performance of the SMVM operation. A software-managed cache allows the user to control various aspects of the cache design. Parameters such as set associativity, number of lines, and line size allow the user to tune the performance of the cache for a given problem.

While a software-managed cache has many benefits, it does come with certain costs. Of particular importance is the space utilized by the cache. Since the SPE Local Store holds both the code and data, one must be careful to balance the impacts of a large software-managed cache. Another difficulty with the software-managed cache is the computational overhead (additional branches) required by more complex cache implementations.

In order to maximize performance of the SMVM, a large software-managed cache was employed. This cache allows the SMVM to make use of locality (spatial/temporal) exhibited by the structure of the coefficient matrix A. A direct map cache implementation was selected for this problem because it requires minimal overhead for detecting if an element is resident and updating is simple.

The sparse matrix column mapping array (ja) defines the actual column mapping of the non-zero sparse elements within the original matrix. These indices are used to indirectly access elements from the source vector during the SMVM operation. The software cache maps these indices to a series of lines of data. Each line contains a linear sequence of double precision elements from the source array.

The cache is structured as 8 lines of 16Kbyte elements (or 2K doubles). This large line size provided the best performance on the test cases we used. It is possible that for highly unstructured data, this large cache line size could provide less than optimal performance. In these cases, the various parameters of the cache can be easily modified to suit the problem. In other cases, a completely different implementation of the software cache can be employed. If thrashing becomes an issue an n-way set associative cache could be useful. The cache tag management makes use of vector intrinsic and vector storage to improve performance.

The cache is split up into two related arrays; the tag (tag0) and data arrays (X_cache0). By defining the tag array as an array of vector elements, we can speed up operations on this array using SPE vector intrinsic operations. This implementation offered approximately a 20-30% improvement in overall performance versus a scalar array implementation.

A structural diagram of the software cache is presented in Figure 6 below. The diagram shows the mapping of the index values to the various components of the cache.

IBM supplies alternative versions of SPE software-managed caches. These were evaluated, and it was found that these versions did not provide the same level of performance as our purpose-designed, direct map version. Another drawback of the IBM cache design for this problem is that the memory required for the cache is not directly accessible to the user code. Since large amounts of memory must be dedicated to the cache, this becomes a problem when memory space is at a premium as is the case with the SPE LS. Since our direct map cache memory is global to the SPE, it can be reused and we make use of this to reduce our data storage memory footprint, thus enabling more code space.

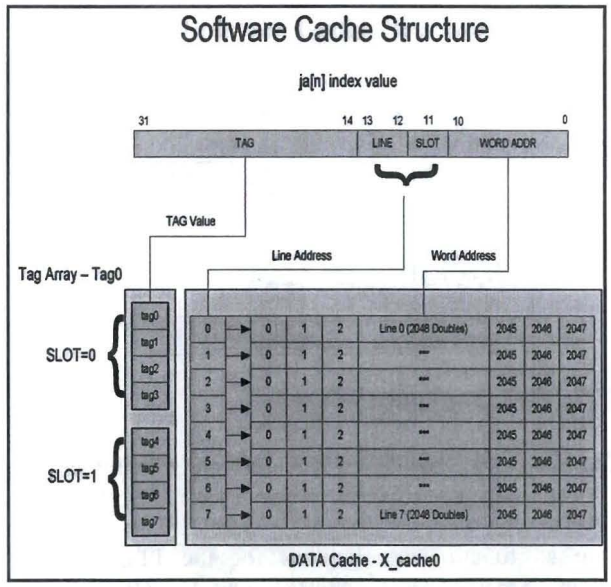


Figure 6. Structural Mapping of the SMVM to the Software Cache in the CG on Cell Implementation.

All operations required for computing the SMVM utilize vector intrinsic to enhance performance. It was found that better performance was gained by restructuring the way data was accessed from local store. By accessing the 14 operands required to

compute two consecutive SMVM results and then shuffling the data to fully utilize the dual issue capability, performance improved by approximately 30%. Computation and communications are fully double buffered and overlapped.

Similarly to the SMVM, all other vector-vector operations utilize vector intrinsic to enhance performance. They all utilize double buffering and fully overlap computation and communication.

3.2. FPGA Implementation

Full details of the FPGA implementation of both the SMVM and CG can be found in our previous work [4] [5][6]. Both User Logic Devices of a single MAP processor were used to implement the CG on the MAPStation.

Through careful placement of array data in the On-Board Memory (OBM) of the MAP processor, near-optimal use of the aggregated memory bandwidth is achieved. True functional parallelism was exploited (via replicated logic) to fully overlap independent computations and gain substantial speed-ups.

3.3. Opteron Implementation

The Opteron implementation makes use of the loop fusion and unrolling optimizations discussed earlier. The choice of the HP wx9400 was due to the improved AMD Opteron performance over the Opterons used on the Roadrunner TriBlade.

The HP xw9400 system is a dual-socket, dual-core AMD 2.2 GHz Opteron (2214) based system. The system utilizes 8 GB of DDR2-667 memory running Windows Vista x64.

Best performance was obtained by mapping the CG code to both cores of a single socket for this implementation. The code did not explicitly utilize threading.

4. Results

In Figure 7 we present the results of our CG implemented on the hybrid Cell and FGPA platforms along with an Opteron only system. We have also included a "projected" result for the SRC-7, the latest machine from SRC. The SRC-7 projected results were calculated using only the 50% system clock rate increase over the SRC-6 (i.e., 150 MHz vs. 100 MHz) and with the assumption that the current FPGA circuit configuration would place and route at this new frequency in the new Altera Stratix II devices used on the SRC-7.

For the hybrid nodes, the effects of data transfers to/from the accelerator (FPGA/Cell) subsystem are apparent for small system sizes. Both Cell and FPGA based systems must transfer much of the system down to the accelerator (x,A,ja,b) and return the solution vector once completed (x). For the Cell we utilized Opteron initiated DaCS RDMA transfers with pinned buffers on the PPE since this mode of operation has the best sustained performance for large data transfers.

While both accelerators transfer data at roughly the same raw data rate (1.2 GB/s) from the host processor, the Cell based system must deal with endian issues. The Opteron uses a little-endian representation while the Cell uses big-endian. For this implementation we chose to utilize the PPE for doing the endian conversion. In this problem where we transfer a large chunk of data and then process for long periods the performance gain is negligible compared to the increased complexity of implementing on the SPE.

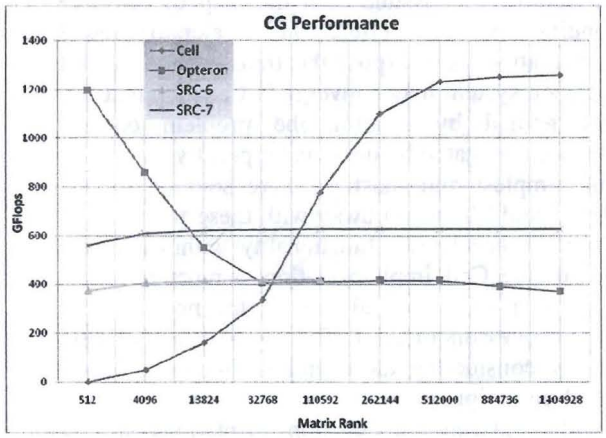


Figure 7. CG Performance Comparison (Cell vs. SRC-6 vs. Opteron), including projected SRC-7 results

The Cell processor obtains a significant performance advantage (up to 3X) over all the other processors (except the SRC-7 "projected" results) for larger problem sizes with matrix ranks of 110,592 or greater. This performance advantage comes from Cell's superior sustained memory bandwidth that we have determined separate from this work to be ~18 GB/s for large matrix ranks. This sustained memory bandwidth was exploited via the use of double buffered DMAs to overlap data movement and computation, and via the software data cache that was tailored to the data requirements of the CG. The performance increase of Cell over the Opteron only system demonstrates the advantages of programmer controlled explicit data movement vs. the fixed caching hierarchies of commodity processors for this type of problem.

The SRC-7 projected results show what we consider to be the minimum performance of the CG on this new FPGA platform. It is possible that the SRC-7 could obtain 2X or more of the performance of the SRC-6 results because of its increased system bandwidths. These results are speculative at the moment, but do show that FPGA based systems have the potential to provide better performance compared to current commodity processors.

5. Discussion

Both of the accelerator based systems (FPGA and Cell) require that the linear system be transferred from the host and results returned for the CG. This extra transfer penalty has a real impact on the performance of this class of problems (i.e., memory bandwidth bound problems).

Real iterative solvers attempt to converge with as few iterations as possible with the help of effective preconditioners. The effect on accelerator based implementations is to expose this transfer time. A well conditioned system may converge fast enough that any benefit gained by running the problem on the accelerator is negated by the transfer penalty.

The simplest and most effective solution to this problem would be to do away with these transfers by merging the accelerator functionality within the host CPU. For the Cell implementation, a more powerful PPE implementation could obviate the need for the Opteron processors of the TriBlade, and manufacturers are likely considering these options in future hybrid architecture designs.

For the Opteron and Cell implementations we rely on caching to gain performance though exploitation of data locality. Our banded test case offers both processors good spatial locality. For more irregular test systems the effectiveness of the caching, in both cases will decrease and lead to poorer performance of the CG. The Cell software cache implementation should degrade in a more gradual fashion due to its much large line size. It also has the benefit that it can be reconfigured in various ways to exploit any available data locality.

6. Conclusion

In this paper we have presented an implementation of a non-preconditioned Conjugate Gradient algorithm on a hybrid Cell processor system. We have shown that the Cell processor is capable of significant sustained memory bandwidth which we exploited to obtain up to 3X the performance compared to a commodity Opteron processor and an older FPGA-based system. The Cell processor requires the programmer to handle all data movements and placements explicitly which adds programming complexity but directly allowed for this performance increase.

Acknowledgements

We would like to thank John Turner of Oak Ridge National Laboratory (formerly Los Alamos National Laboratory) and John Wohlbier of Los Alamos National Laboratory for support and guidance.

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- 09-00248.

References

[1] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., Yelick, K. A., The Landscape of Parallel Computing Research: A View from Berkeley. EECS Department, University of California, Berkeley, (December 18, 2006). Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] 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.

[3] deLorimier, M. and DeHon, A., "Floating-point Sparse Matrix-vector Multiply for FPGAs," In FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, pages 75–85, New York, NY, USA, 2005. ACM Press.

[4] DuBois, D., DuBois, A., Boorman, T., Connor, C., Poole, S., An Implementation of the Conjugate Gradient Algorithm on FPGAs. In *Proceedings of the 2008 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)* (Stanford, Palo Alto, California, USA. April 14-15, 2008)

[5] DuBois, D., DuBois, A., Boorman, T., Connor, C., Poole, S., Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer with Application. ACM Transactions on Reconfigurable Technology and Systems (TRETS). SFR (2008).

[6] DuBois, D., DuBois, A., Connor, C., Poole, S., Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer . In *Proceedings of the 2008 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)* (Stanford, Palo Alto, California, USA. April 14-15, 2008)

[7] 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.

[8] Hewlett-Packard Development Company, L.P., HP xw9400 Workstation Product information datasheet, (October 2008). Available: http://h10010.www1.hp.com/wwpc/pscmisc/vac/us/product_pdfs/xw9400_datasheet_Oct08.pdf

[9] IBM Corporation, ALF for Cell BE Programmer's Guide and API Reference, (10/19/2007). Available: http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465/$file/ALF_Prog_Guide_API_v3.0.pdf

[10] IBM Corporation, BladeCenter LS21 Product Datasheet, (2008). Available: http://www-03.ibm.com/systems/bladecenter/hardware/servers/ls21/specs.html

[11] IBM Corporation, BladeCenter QS22 Product Datasheet, (2008). Available: (ftp://ftp.software.ibm.com/common/ssi/pm/sp/n/bld03019usen/BLD03019USEN.PDF

[12] IBM Corporation, Cell Broadband Engine Programming Handbook, Including the PowerXCell 8i Processor, (May 12, 2008). Available: http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D/$file/CellBE_PXCell_Handbook_v1.11_12May08_pub.pdf

[13] IBM Corporation, DaCS Hybrid-x86 Prog Guide API v3.0, (10/19/2007). Available: http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/ADFBD392E0ED2D4C00257353006B2744/$file/DaCS_Hybrid-x86_Prog_Guide_API_v3.0.pdf

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

[15] Komornicki, A., Mullen-Schulz, G., Roadrunner: Hardware and Software Overview, IBM Redbook Form Number:REDP-4477-00, (January 23, 2009). Available: http://www.redbooks.ibm.com/abstracts/redp4477.html

[16] Li, D., Huang, S., Cameron, K., CG-Cell: An NPB Benchmark Implementation on Cell Broadband Engine. ICDCN 2008, LNCS 4904, pp. 263-273, 2008. Springer-Verlag Berlin Heidelberg 2008.

[17] Los Alamos National Laboratory, Roadrunner-Science at the Petascale, (October 2008). Available: http://www.lanl.gov/asc/docs/rr_factsheet.pdf

[18] 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.

[19] *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

[20] 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

[21] 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.

[22] SRC Computers, Inc, Product Page, July, 1999-2008. Available: http://www.srccomp.com/products/products.asp

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

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

[25] Wellein, G., Hager, G., Zeiser, T., "Basic principles of modern processors: Memory Hierarchy Optimization of Data Access." (April, 2005). Avalable: http://www.rrze.unierlangen.de/ausbildung/vorlesungen/04-25_2005_ptfs.pdf

[26] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. 2006. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers* (Ischia, Italy, May 03 - 05, 2006). CF '06. ACM, New York, NY, 9-20. DOI= http://doi.acm.org/10.1145/1128022.1128027