Evaluation of a High-Level-Language Methodology for High-Performance Reconfigurable Computers

Jahyun J. Koo, David Fernández, Ashraf Haddad and Warren J. Gross
Department of Electrical and Computer Engineering
McGill University, Montreal, Quebec, Canada
{jeff.koo,david.fernandezbecerra,ashraf.haddad}@mail.mcgill.ca, warren.gross@mcgill.ca

Abstract

High-performance reconfigurable computers (HPRCs) consisting of CPUs with application-specific FPGA accelerators traditionally use a low-level hardware-description language such as VHDL or Verilog to program the FPGAs. The complexity of hardware design methodologies for FPGAs requires specialist engineering knowledge and presents a significant barrier to entry for scientific users with only a software background. Recently, a number of High-Level Languages (HLLs) for programming FPGAs have emerged that aim to lower this barrier and abstract away hardware-dependent details. This paper presents the results of a study on implementing hardware accelerators using the Mitrion-C HLL. The implementation of two floating-point scientific kernels: dense matrix-vector multiplication (DMVM) and the computation of spherical boundary conditions in molecular dynamics (SB) are described. We describe optimizations that are essential for taking advantage of both the features of the HLL and the underlying HPRC hardware and libraries. Scaling of the algorithms to multiple FPGAs is also investigated. With four FPGAs, 80 times speedup over an Itanium 2 CPU was achieved for the DMVM, while a 26 times speedup was achieved for SB.

1. Introduction

High-Performance Reconfigurable Computers (HPRCs) are an emerging architecture for the acceleration of floating-point scientific applications because of their capability to take advantage of both the coarse-grained parallelism offered by traditional CPU-based multiprocessors and fine-grained parallelism offered by reprogrammable devices such as FPGAs. Generally, developers who are well-versed in hardware design have been required to program FPGAs using low-level hardware description languages (HDLs) such as VHDL or Verilog. This constitutes a significant barrier to entry for scientists with a software-only background to program FPGAs. Several commercial high-level languages (HLLs) for programming FPGAs have been proposed to overcome some of the hardware-dependent limitations associated with FPGA-based implementations such as Handel-C [2], Impulse C [3] and Mitrion-C [4].

This paper presents the results from a study on implementing scientific applications on an SGI Altix 350 with RASC RC100 FPGA accelerators. The FPGAs are programmed using the Mitrion-C HLL. The features provided by Mitrion-C and the RASC RC100 hardware are described in Section 2 and 3, respectively. Two kernels are implemented in Mitrion-C. In Section 4, the DMVM and SB kernels are implemented on FPGAs using the features described in Sections 2 and 3. We also describe various optimization techniques to improve the performance of these algorithms and investigate their execution in parallel using multiple FPGAs. The performance comparison between the FPGA implementations and the software implementation is provided in Section 5. Section 6 presents some advantages and limitations of the features described in this study and Section 7 offers conclusions.

2. Mitrion-C High-Level Language

Mitrion-C is a single-assignment, dataflow oriented, parallel HLL with C-like syntax. Hardware considerations such as timing are not exposed to the programmer who focuses on expressing the dataflow of the algorithm. Parallelism is expressed using explicit language constructs and can be exploited in the form of vectorization and/or pipelining. Two array data types are offered: *lists* of elements accessible sequentially, and *vectors* in which all elements can be addressed at once, or in any order. The resulting architecture is determined by the selection of data type (list or vector) and the parallel operator applied to it. A parallel *foreach* construct indicates a data-parallel loop. Combining the vector data type with the foreach loop creates ex-

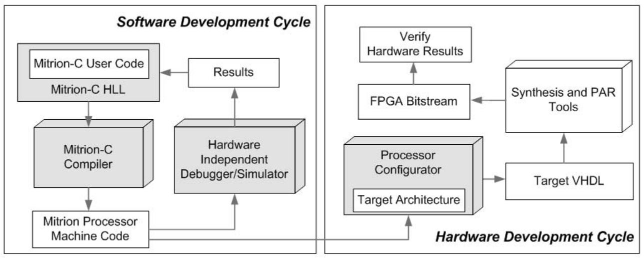


Figure 1. Mitrion-C design flow.

plicit parallel structures with multiple processing elements (PEs). However, this design style rapidly consumes FPGA resources. In contrast, applying a foreach loop to a list implements a pipeline. Applications can use a combination of both lists and vectors. Multi-dimensional collections are also allowed (for example, lists of vectors, or vectors of vectors of lists). This allows a tradeoff to be reached between circuit area and execution time.

The design flow is described in Figure 1. The Mitrion compiler generates Mitrion machine code which can be used for simulation in the debugger/simulator. The simulator provides a graphical representation of the hardware data flow graph. This graphical tool is key in identifying program errors and finding design bottlenecks. The debugger/simulator also provides breakpoint and watch functions similar to the ones used in software debuggers.

The Mitrion processor configurator generates VHDL from the Mitrion machine code and the pre-defined target FPGA architecture. The generated processor, that runs at a fixed frequency of 100 MHz, is wrapped in the SGI core services which provide memory and communications interfaces to the RC100 and CPUs in the Altix 350. Synthesis and place-and-route are performed by Synplify Pro and Xilinx ISE tools.

3. RASC RC100 FPGA System

Our SGI Altix 350 contains eight 1.5 GHz Intel Itanium 2 CPUs with 16GB of shared memory connected by a NUMAlink interconnection network. Two RASC RC100 accelerators, each containing two Xilinx Virtex-4 LX200 FPGAs are connected to the Altix system through the NUMAlink and the TIO ASIC. Each of the four FPGAs has five 8MB QDR SRAM DIMM memory banks, although the core services currently provide access to 32MB per FPGA. Each bank can be accessed by the FPGA at a rate of 128 bits per cycle at the Mitrion processor clock frequency of 100MHz, which results in a bandwidth of 1.6 GB/s. A loader FPGA enables fast bitstream loading into the computational FPGAs. The overall view of a single RASC RC100 is given in Figure 2.

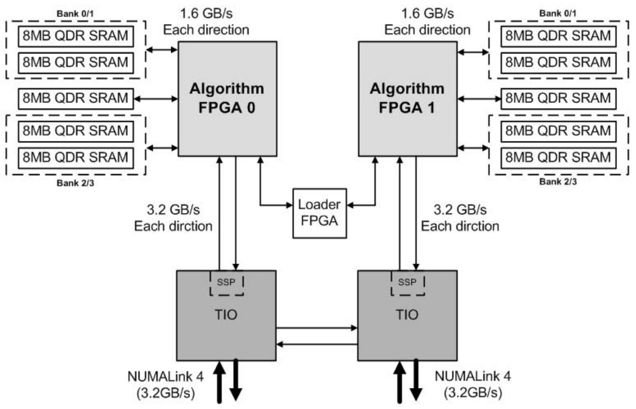


Figure 2. RASC RC100 hardware.

The FPGA external SRAM banks have limited memory as shown in Figure 2. Therefore, if operating on large data sets, the RASC RC100 provides a feature called *streaming* [7]. Streaming reduces the overhead of data transfer by overlapping data loading and unloading with algorithm execution. As shown in Figure 3, the streaming feature requires the designer to split the 32MB SRAM into two memory blocks; one for input memory (Bank A) and the other for output memory (Bank B). These memory banks will be segmented into two subblocks so that the implemented algorithm can process data in Bank A1 and write the results in Bank B0, while the next input data is being loaded into Bank A0 and the computed results unloaded from Bank B1 to the host buffer. When all data in Bank A1 has been processed, the FPGA will start executing on the loaded segment (Bank A0), while the freed segment (Bank A1) will begin loading the next input data set.

Another technique provided is an automatic scaling over multiple FPGAs called *wide-scaling*. A single data set can be automatically partitioned and sent to multiple FPGA running identical bitstreams. The results are re-assembled into a single results data set automatically. In the cases where different algorithm bitstreams are required for each FPGA, or a non-uniform data partitioning is required, manual scaling of the application by the programmer is needed. An example of both techniques is given in Section 4.

4. Implemented Applications

In this paper, two computationally intensive algorithms are implemented. First is the Dense Matrix-Vector Multiplication algorithm (DMVM), which is the dominant kernel of many scientific applications. DMVM is a short, simple and very computationally intensive algorithm making it ideal for hardware acceleration. The second algorithm is a component of molecular dynamics simulations, specifi-

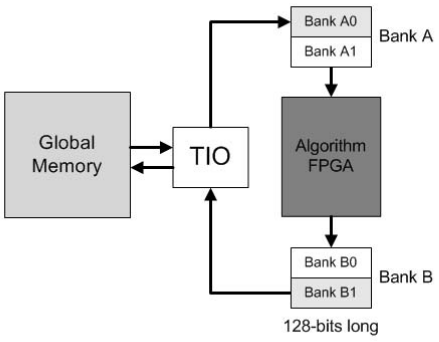


Figure 3. Configuration for streaming large data sets.

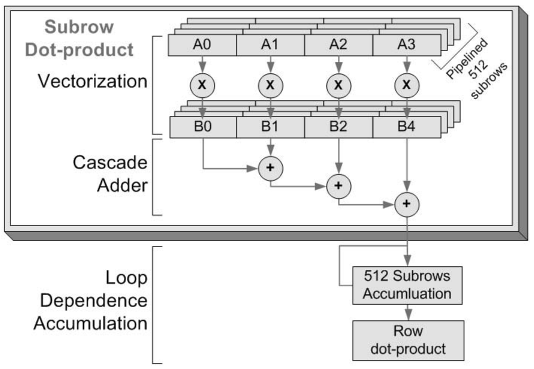


Figure 4. One row dot-product PE.

cally, the computation of the spherical harmonic boundary conditions (SB). The SB algorithm was chosen because of its inherent fine-grained parallelism. The following subsections describe the implementation of both algorithms using Mitrion-C. Single-precision floating point is used instead of fixed-point or custom-precision arithmetic to ensure that the design methodology and results for hardware and software are comparable.

4.1. Floating-Point Dense Matrix-Vector Multiplication(DMVM)

The DMVM algorithm has a pair of nested for-loops. The outer loop iterates over the rows of the matrix, while the inner loop iterates over the columns of the selected row, computing a dot-product between the current row and column-vector for each iteration of the outer loop.

We implemented 2048 $\times$ 2048 matrix-matrix multiplication consisting of 2048 DMVM operations. The first DMVM hardware implementation calculates multiple scalar-scalar products within a dot-product. To fully exploit parallelism, the vector data type is used and then a cascade addition is applied to the scalar-scalar products. Due to the size of the test matrix (2048 $\times$ 2048), the limited number of floating point multipliers and the limited memory bandwidth, a whole row cannot be computed at once, therefore the matrix rows and column-vector are split into subrows. The subrows of each row in the matrix and the column-vector are stored as a length-512 lists of four-element-vectors in two different memory banks. This subrow processing is similar to the techniques used in [9] and [5]. Each subrow is chosen to contain four elements since only four 32-bit single-precision floating point values can be stored in each memory address of the external FPGA SRAMs.

The foreach loop is used to do parallel multiplication of the four elements in the matrix and column-vector subrows and to compute the sum of their result as shown in Figure 4. After every subrow dot-product is computed, the accumulation over 512 subrow dot-products is done using a for loop to obtain one completed vector dot-product. To execute the whole DMVM, a second foreach loop is used. This loop reads the rows from the matrix and the column-vector and computes the dot-products of the matrix rows and the given column-vector as described previously. It finally writes the resulting values to memory. We store the matrix in one 16 MB FPGA memory bank and the column-vectors in the other 16 MB bank. Figure 5 shows the memory layout for one row of the matrix stored in RAM0 and the column-vector in RAM1.

A second design was implemented to enhance the performance using fine-grained optimization of the Mitrion-C code. First, four subrow dot-product PEs are used in parallel instead of one PE as shown in Figure 5. Another optimization is done by replacing the cascaded addition with a two level reduction tree in the subrow dot-product PE([9] [5]). An additional two level reduction tree is implemented to accumulate the outputs from the four PEs. These optimizations increased the number of scalar-scalar products performed in parallel from 4 to 16, and decreased the number of iterations required in the sequential loop-dependent accumulation from 512 (Figure 4) to 128 (Figure 5).

The DMVM design was first implemented on one FPGA and then scaled to two and four FPGAs. The data is manually partitioned into row blocks and processed by FPGAs, each of which used identical bitstreams. Multiple CPUs are used to simultaneously start the FPGA processing by using Pthreads.

4.2. Spherical Boundary Conditions in Molecular Dynamics (SB)

The spherical harmonic boundary condition simulates solvating protein in water (used to study proteins in cellular environment). The purpose of the algorithm is to calculate

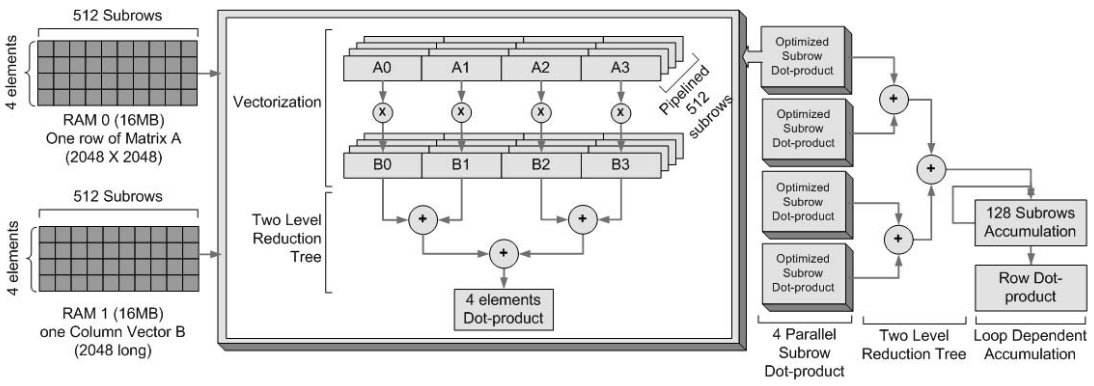


Figure 5. Optimized row dot-product PE.

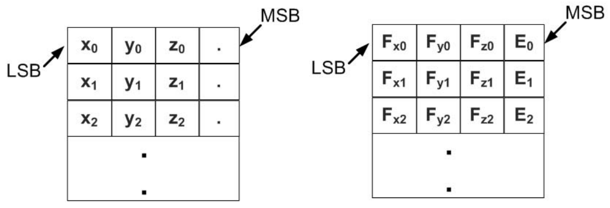
the energy and force on each of the protein's atoms solvated in water. The force and energy are calculated in regards to reference potentials, which correspond to boundary conditions. The potentials are represented by two concentric spheres. The energy for each atom i due to the boundary conditions is [1]:

$$E_s = k_s \left(\left| \vec{r_i} - \vec{r}_c \right| - r_s \right)^{\exp}, \qquad (1)$$

where k_s is the force constant, measured in kcal/mol/A^2, associated with each sphere, and *exp* is the exponent defined for both spheres. A default value of 2 was used for *exp*. $\vec{r_i}$ is the current position of atom i. $\vec{r}_c$ is the center position. The values r_i and r_c are measured in Angstroms. Moreover, the force applied by the same potential is:

$$\vec{F}_s = \left(\exp_s k_s \left(\left| \vec{r_i} - \vec{r_c} \right| - r_s \right)^{\exp - 1} \right) \hat{r}_{i,c}, \qquad (2)$$

where $\hat{r}_{i,c}$ is a unit vector in the direction from atom i, to the center of the sphere. The force and energy of an atom located between the two spheres is calculated using the inner sphere's potential. If, however, the atom is outside the outer sphere, its force and energy are calculated with reference to that sphere's potential. The software version, derived from the source code presented in [8], uses one dimensional arrays, as an input, to store the positions ($\vec{r_i}$) of the individual atoms in the protein structure. Each array element in Figure 6(a) contains the x, y and z coordinates of an atom. The algorithm iterates sequentially through the set of atom. It calculates the distance from the atom to the center point and uses this information to determine which sphere to use in the energy and force calculations. Next, the energy and force equations are applied. The program finally writes the output to a one dimensional array. The first three 32-bits of each row in the array refer to the force's x, y and z components, while the last 32-bit stores the energy as shown in Figure 6(b). The source of parallelism in this algorithm comes from the fact that the force and energy calculations of each atom are completely independent from other atoms. Therefore, a pipelined architecture is selected and implemented using a list data type that is processed by a Mitrion-C foreach loop. To process a single atom, the value of $\left(\left| \vec{r_i} - \vec{r}_s \right| - r_s \right)$ is first evaluated and then the energy and force calculation (Equations 1, 2) are performed in parallel.



(a) Input memory (b) Output memory

Figure 6. SB memory structure.

To increase the parallelism, our Mitrion-C design employs a mixture of pipelining and vectorization over several atoms. Due to limited FPGA resources, only four atom pipelines can be implemented in parallel as shown in Figure 7. This limitation is a direct result of the intensive force/energy computations which involve floating-point square roots, division and a number of multiplications. Streaming is used to accommodate data sets that are larger than the SRAM banks. In this application, the RASC wide-scaling is used to scale to two and four FPGAs using the same bitstream on each and allowing the RASC libraries to automatically perform the data partitioning.

5. Results

The applications were developed using gcc 4.1.0, RASC Library 2.0 and Mitrion-C 1.1. Software versions of the applications were developed in ANSI-C and run on a sin-

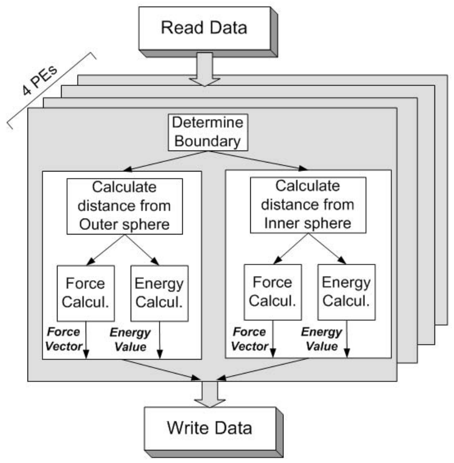


Figure 7. SB design with four PEs.

Table 1. Summary of hardware resources.

Resource (total)	DMVM	SB
Flip Flops (178,176)	23%	37%
LUTs (200,448)	22%	38%
Multipliers (96 18×18)	66%	33%
18kb Block RAMs (336)	9%	7%

Table 2. Speedup over 1.5 GHz Itanium 2 using a single FPGA.

	DMVM	SB
Software	0.900 s	1.607 s
Non-optimized Hardware	0.105 s (8× Speedup)	-
Optimized Hardware	0.042 s (21× Speedup)	0.158 s (10× Speedup)

Table 3. Speedup over a 1.5 GHz Itanium 2 for multiple FPGAs.

Number of FPGAs	DMVM	SB
1	0.042 s (21× speedup)	0.158 s (10× speedup)
2	0.022 s (41× speedup)	0.101 s (16× speedup)
4	0.011 s (80× speedup)	0.061 s (26× speedup)

gle 1.5 GHz Itanium 2 in the SGI Altix 350. The resources consumed by the optimized hardware-implemented DMVM and SB applications are shown in Table 1. The average time spent for the synthesis and place-and-route of the DMVM and SB was 3 and 4 hours respectively. To test the DMVM application, random single-precision floating point 2048 × 2048 matrices were used. This is the maximum matrix size that can fit in the 16MB memory banks of a single FPGA. The host program times the end-to-end execution time of both the software and the hardware versions. The hardware time also includes sending and receiving data in addition to running the bitstream, to ensure a fair SW/HW comparison. To determine the average speedup, 2048 matrix-vector multiplications were measured. Table 2 shows the speedup results for the non-optimized and optimized versions of the hardware implementations with respect to the corresponding software implementation.

For the SB application, a single precision floating point data-set of 32 MB was used, which amounts to 2,097,152 atom computations. This data was obtained from a Protein Data Bank (PDB) file [6]. The speedup was calculated by running the host program 30 times and taking the average of the time readings. Table 2 shows the speedup results obtained.

The two algorithms were scaled to two and then four FPGAs. Manual partitioning was performed for the DMVM application (1024 rows and 512 row blocks for two and four FPGAs respectively). Pthreads were used to send the data, execute the hardware implementation and receive the data. The RASC wide-scaling was used to run the SB on multiple FPGAs. The speedup results achieved with multiple FPGAs are shown in Table 3.

6. Discussion

We note that even though Mitrion-C abstracts away most hardware details, the designer needs to interact with the hardware at certain points in the design process. The designer must explicitly define the memory layout and pack variables into the external SRAMs. An iterative design process determined by the hardware constraints was often necessary. After the place-and-route, it was often discovered that the generated circuit did not fit on the FPGA, and therefore the designer needed to change the Mitrion-C code to make the circuit fit (such as changing vectors to lists). For example, a new DMVM design with eight subrow dot-product PEs only used approximately 35% of Flip-Flop and 41% of LUT resources. However, this design could not be placed-and-routed to meet timing constraints after 9 hours.

Based on the results presented in Section 5, it was determined that different types of applications can be accelerated

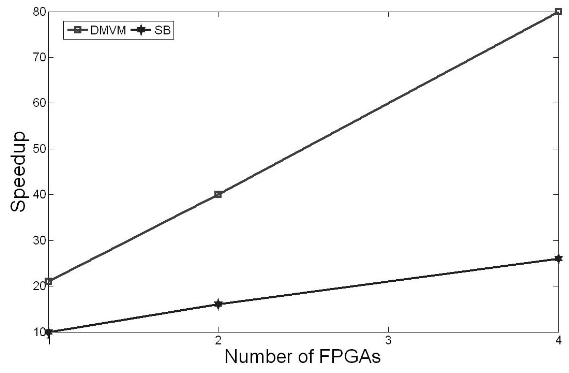


Figure 8. Speedup achieved when scaling from one to four FPGAs.

significantly with HPRC by employing a mixture of pipelining and vectorization techniques. In this study, we determined that a higher order of vectorization provides explicit parallelism at the cost of dramatically increasing the silicon space consumed. Therefore, we recommend that applications should be initially implemented using only pipelining and then have designers check the amount of hardware resources used. If more resources are available, PEs can be duplicated until available resources are used. This can easily be done by replacing list data types with vectors or by reshaping the original data to a combination of vectors and lists.

RASC streaming and wide-scaling features were also evaluated. it was determined that streaming is a suitable solution for applications with input buffer bigger than the available FPGA external memory. The two implemented algorithms were also successfully scaled to multiple FPGAs using both automatic and manual partitioning. Manual partitioning was implemented for the DMVM algorithm because the RASC wide-scaling does not generate the correct row boundaries for arbitrary matrix sizes. On the other hand, in the SB algorithm, wide-scaling was used simply by indicating the number of desired FPGAs in the host program.

Figure 8 plots the speedup versus the number of FPGAs for one, two and four FPGAs. The speedup of the DMVM was doubled when doubling the number of FPGAs used. On the other hand, the speedup of the SB increased by a factor of only 1.6$\times$ when doubling the number of FPGAs. This is due to the fact that SB is a streaming algorithm which reads new data every clock cycle and is bound by the NUMAlink bandwidth from the shared memory to the FPGA SRAMs. However, the DMVM does a single block transfer to the SRAMs and is compute-bound. We expect further upgrades of the RASC software and Mitrion processor to increase the available memory bandwidth and improve the performance of streaming applications.

To enhance the bandwidth of the DMVM implementation, we tried storing the matrix in the two FPGA external memory RAM banks while storing the column-vector in FPGA internal block RAMs. This approach was unsuccessful, as the system ran out of block RAM resources when attempting to synthesize the design, even though our initial estimation showed that there were enough internal block RAMs to store the vector.

7. Conclusion

This paper showed how the high-level design flow using Mitrion-C and the SGI RASC platform effectively enables the scientific community to overcome the hardware barrier imposed by traditional low-level methodologies. Two computationally intensive algorithms were implemented in the SGI RC100 using Mitrion-C. The development was for the most part free of hardware design considerations and focused on the data flow of the algorithms. Both fine and coarse grained optimization techniques, provided by Mitrion-C and RASC respectively, were used. The DMVM algorithm achieved a 21 times speedup with one FPGA and 80 times with four FPGAs. Moreover, the SB algorithm achieved 10 times speedup with one FPGA and 26 times with four FPGAs.

References

[1] R. Brunner, A. Dalke, A. Gursoy, W. Humphery, and M. Nelson. *NAMD Programming Guide*. University of Illinois and Beckman Institute, v1.5 edition, Sept. 1998.

[2] Celoxica. *Handel-C Language Reference Manual for DK4 version*. Celoxica Limited.

[3] Impulse Accelerated Technologies, Inc. C Programming Tools for FPGA Platforms, 2007. http://www.impulsec.com/.

[4] S. Mohl. *The Mitrion-C Programming Language*. Mitrionics Inc., 2005. http://www.mitrionics.com/forum/.

[5] G. R. Morris, V. K. Prasanna, and R. D. Anderson. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Proc. 14th Annual IEEE Symp. on Field-Prog. Custom Comp. Machines (FCCM'06)*, pages 3–12. IEEE Computer Society, 2006.

[6] Research Collaboratory for Structural Bioinformatics(RCSB). RCSB Protein Data Bank (PDB), Feb 2007. http://www.pdb.org/pdb/home/home.do.

[7] Silicon graphics Inc. *Reconfigurable Application-specific Computing User's Guide*, 007-4718-004 edition, 2006.

[8] Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign. NAMD-Scalable Molecular Dynamics, Mar 2006. http://www.ks.uiuc.edu/Research/namd/.

[9] 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(FPGA'05)*, pages 63–74. ACM Press, 2005.