Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems *

Ling Zhuo and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
{lzhuo, prasanna}@usc.edu

Abstract

Recently, reconfigurable computing systems have been built which employ Field-Programmable Gate Arrays (FPGAs) as hardware accelerators for general-purpose processors. These systems provide new opportunities for high-performance computing. In this paper, we investigate hybrid designs that effectively utilize both the FPGAs and processors in the reconfigurable computing systems. Based on a high-level computational model, we propose designs for floating-point matrix multiplication and block LU decomposition. In our designs, the workload of an application is partitioned between the FPGAs and processors in a balanced way; the FPGAs and processors work cooperatively without data hazards or memory access conflicts. Experimental results on Cray XD1 show that with one Xilinx XC2VP50 FPGA (a relatively small device available in XD1) and an AMD 2.2 GHz processor, our designs achieve up to 1.4X/2X speedup over the design that employs AMD processors/FPGAs only. The performance of our designs scales with the number of nodes. Moreover, our designs achieve higher performance when improved floating-point units or larger devices are used.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are a form of reconfigurable hardware. They offer the design flexibility of software, but with time performance closer to Application Specific Integrated Circuits (ASICs). With rapid advances in technology, current FPGA devices contain much more resources than their predecessors. FPGAs are now able to provide high computational parallelism as well as I/O parallelism. They have become an attractive option to accelerate scientific applications [18, 20].

*Supported by the United States National Science Foundation under grant No. CCR-0311823 and in part by No. ACI-0305763.

With the increasing computing power of FPGAs, high-end reconfigurable computing systems have been built, which employ FPGAs as application-specific accelerators for general-purpose processors. Such systems include SRC MAPstation [17] and Cray XD1 [5], among others. These systems are similar to existing distributed systems, with multiple compute nodes connected through an interconnect network. However, the compute nodes of these systems contain both FPGAs and general-purpose processors.

These reconfigurable computing systems provide a new platform for parallel and distributed computing. Coarse-grain parallelism can be performed on multiple computing nodes, while FPGAs are suitable for fine-grain parallelism. With FPGAs and processors, these systems can also achieve higher performance than systems with processors only. However, existing works on these systems mainly focus on the usage of FPGAs, and do not exploit the computing power of the general-purpose processors.

In this paper, we investigate hybrid designs that effectively utilize both the FPGAs and the processors in the reconfigurable computing systems. We address various design challenges, including workload partition for load balance, communication and cooperation between FPGAs and processors, parallelism within FPGAs and scalability over multiple compute nodes.

We select floating-point matrix multiplication and block LU decomposition as our example applications. Both of them are basic kernels in scientific computing, and their high performance is crucial to the performance improvement of many applications. Our proposed hybrid designs utilize both the FPGAs and the processors, and can be implemented on multiple compute nodes in the system. Within each compute node, the workload is partitioned between the processors and the FPGAs using a performance modeling method. The computations on the FPGAs and processors are coordinated so that data hazards and memory access conflicts are avoided. On the FPGAs, we employ existing FPGA-based designs to achieve high performance.

To illustrate our ideas, we implemented our designs on

Cray XD1. Each node of XD1 consists of two 2.2 GHz AMD Opteron processors and a Xilinx XC2VP50 FPGA. The nodes in XD1 are connected through MPI (Message Passing Interface) [14], and six nodes fit in a chassis. In the experiments, we used our own FPGA-based double-precision floating-point units and matrix multiplier. Our designs achieve up to 1.4X speedup over designs that use the processors only, and 2X speedup over designs that use the FPGAs only. In particular, with 6 processors and 6 FPGAs, our designs achieve 24.6 GFLOPS for matrix multiplication. Experimental results show that our designs are scalable over nodes in one chassis. Moreover, with smaller and faster floating-point units as well as available larger FPGA devices, we show that the performance of our designs will increase accordingly.

The rest of the paper is organized as follows. Section II introduces background of FPGAs and related work. Section III presents the computational model of reconfigurable computing systems. Section IV discusses the design challenges. Section V presents our hybrid designs for matrix multiplication and block LU decomposition. Section VI shows the implementation on one chassis of Cray XD1 and analyzes the performance. Section VIII concludes the paper.

2 Background & Related Work

2.1 Linear Algebra on Parallel and Distributed Systems

There has been extensive research work for dense linear algebra on distributed systems [1, 12]. High performance linear algebra libraries have also been designed and implemented. One of the most widely used libraries on distributed-memory MIMD systems is ScaLAPACK (Scalable LAPACK) [4]. ScaLAPACK implements a subset of LAPACK (Linear Algebra PACKage) [3] routines, including routines for solving systems of linear equations, least squares problems, and eigenvalue problems. Another library is HPL, a Portable Implementation of the High-Performance Linpack [7] Benchmark for Distributed-Memory Computers [15].

In both of these libraries, the block-cyclic data layout has been selected for the dense matrices. Such layout can minimize the frequency of data movement between different levels of the memory hierarchy. In both ScaLAPACK and HPL, the computational model consists of a one- or two-dimensional processor grid, where each process stores pieces of the matrices and vectors. In ScaLAPACK, a message-passing library designed for linear algebra, BLACS (Basic Linear Algebra Communication Subprograms), is used for inter-process communication. HPL uses MPI explicitly in the programs.

These existing libraries have achieved high performance on traditional distributed systems. However, they cannot fully exploit the computing power of the reconfigurable computing systems which contain both processors and FPGAs. Still, their computational model is very useful to our work when there is a large number of compute nodes.

2.2 Linear Algebra on FPGAs

FPGAs provide a hardware fabric upon which applications can be programmed. An FPGA device consists of tens of thousands of logic blocks (clusters of *slices*) whose functionality is determined by programmable configuration bits. These logic blocks are connected using a set of routing resources that are also programmable. Thus, mapping a design to an FPGA consists of determining the functions to be computed by the logic blocks, and using the configurable routing resources to connect the blocks. The configurations of logic blocks and the routing resources can be modified by loading a stream of bits onto the FPGA.

Many researchers have studied the impact of increasing computing power of current FPGAs. In [20], we proposed a design for floating-point dense matrix multiplication. For problem size n, the effective latency of the design is $\Theta(n^2)$, using storage size of $\Theta(n^2)$. In [8], a block matrix multiplication algorithm is discussed for large n, and a floating-point MAC (Multiplier and ACcumulator) is implemented. In [6, 22], FPGA-based designs for floating-point sparse matrix-vector multiplication are proposed and achieve high speedup over general-purpose processors. In [18], FPGA-based implementations of BLAS (Basic Linear Algebra Subprograms) operations are discussed. The main focus of that work was to examine the potential capacity of FPGAs in performing BLAS operations. The only work that has implemented linear algebra applications on the reconfigurable computing systems is [21]. However, it only employs the FPGAs in the systems. In this paper, we aim to exploit the computing power of both the FPGAs and the processors in the systems.

3 Reconfigurable Computing Systems

3.1 Hardware Architecture

Several high-end reconfigurable computing systems have been developed. Two representative systems are discussed below.

SRC MAPstation In SRC MAPstation, there is one Intel microprocessor and one reconfigurable logic resource which is referred to as a MAP processor. Each MAP processor consists of two Xilinx FPGAs and one FPGA-based controller. Each FPGA has access to six banks of on-board

(SRAM) memory, which provides a total memory bandwidth of 4.8 GB/s. The FPGA controller facilitates communication and memory sharing between the microprocessors and the FPGAs. Multiple MAPstations can be connected by Ethernet to form a cluster.

Cray XD1 Cray XD1 [5] also combines microprocessors with FPGAs. The basic architectural unit is a compute blade, which contains two AMD Opteron processors and one Xilinx Virtex-II Pro FPGA. Each FPGA has access to four banks of QDR II SRAM, and the total SRAM bandwidth available to the FPGA is 12.8 GB/s. Through Cray's RapidArray processors, the FPGAs can access the DRAM of the microprocessors, with a DRAM bandwidth of 2.8 GB/s. Compute blades communicate through RapidArray internal switches, and six compute blades fit into one chassis. Multiple chassis can be connected through RapidArray external switches.

3.2 Architectural Model

The reconfigurable computing system can be modeled as a distributed systems with multiple compute nodes. In this system, each node consists of one or more general-purpose processors and one FPGA device. The processors and the FPGAs are closely connected within each node so that they can share their memory. As no global memory space exists among multiple nodes, a node cannot access the local memory of another node without the assistance of the remote node.

An FPGA has access to multiple levels of memory. The first level is the on-chip memory of the FPGA, usually Block RAMs (BRAMs). In the large state-of-the-art devices, the aggregate memory bandwidth of BRAMs is over 100 GB/s, while the total size is usually less than 10 Mb. The second level of memory is off-chip but on-board memory, which is usually SRAM. The storage capacity of SRAM memory is much larger than that of the on-chip memory, however its bandwidth to FPGA is usually less than 20 GB/s. The FPGA also has access to the main memory of the general purpose processor. Such memory is usually DRAM and in the range of gigabytes, but the bandwidth is even less than the SRAM bandwidth.

The architectural model of the systems is shown in Figure 1. In the figure, P_0, ..., P_{t-1} refer to the compute nodes. M_i is the local memory of P_i. It consists of DRAM and SRAM.

Each reconfigurable computing system provides its own interconnect scheme among the compute nodes. However, the overall system architecture is similar. In many of these systems, a network hierarchy exists: multiple nodes are connected closely to form a group. Multiple groups are further connected to form a larger system. We thus call the network inside a group the "lower hierarchy", and the network among the groups the "upper hierarchy". The network topology is shown in Figure 2. Our work in this paper mainly focus on the designs for the lower hierarchy.

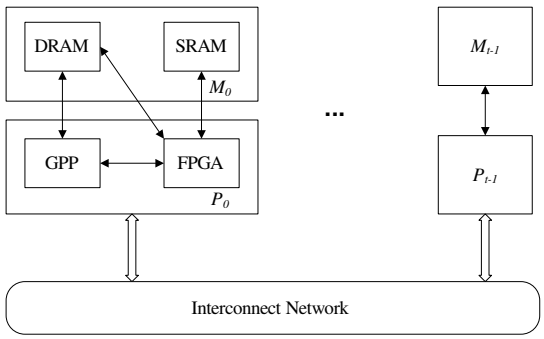

Figure 1. Architectural model for reconfigurable computing systems

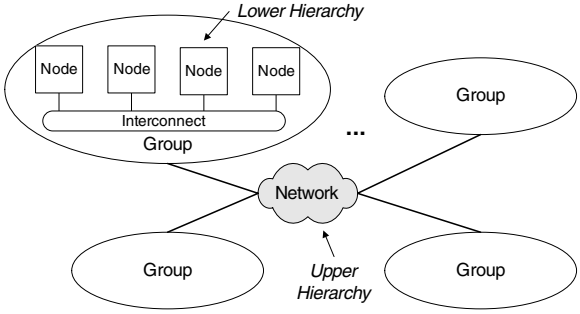

Figure 2. Overall architecture of reconfigurable computing systems

4 Design Challenges

Since the reconfigurable computing systems contain additional computing resources, it is reasonable to expect them to achieve higher performance than systems with processors only. To achieve this goal, hybrid designs are needed to exploit the computing power of both the FPGAs and processors. Also, the designs need to be scalable so that their performance increases with the number of nodes. However, to propose such designs, certain design challenges have to be overcome. These challenges are discussed in this section.

1. Concurrency Between Processor & FPGA

Workload Partition: Within a compute node in the reconfigurable computing systems, an application needs to be partitioned between the processor and the FPGA. In general, the computationally intensive section of the application is executed on hardware, and the processor is used for control intensive section. Performance profilers, such as

VTune [11] and gprof [9], can be used to identify the most time-consuming section of an application. If this section is suitable for hardware acceleration, it will be implemented on the FPGA, while the remaining part of the application is executed by the processors.

For computationally intensive applications, one critical goal of workload partition is load balancing. If the workload between the processor and the FPGA is not balanced, one of them has to wait for the other to complete and hence wastes its own computing power. Thus, we employ a performance modeling method to determine the workload partition. Suppose N_p floating-point operations are assigned to the processor and N_f operations are assigned to the FPGA. Assume that the processor performs O_p operations during each clock cycle and the FPGA performs O_f operations in one clock cycle. Also, the clock speeds of the processor and the FPGA are F_p and F_f, respectively. Thus, the computational time of the processor $T_p = \frac{N_p}{O_p \times F_p}$, and the computational time of the FPGA $T_f = \frac{N_f}{O_f \times F_f}$. We should choose N_p and N_f so that $T_p = T_f$.

Note that the accuracy of this modeling method is dependent on the accuracy of estimating O_f, O_p, F_f and F_p. As we have full control of the FPGA-based designs, O_f and F_f can be accurately determined. However, $O_p \times F_p$ reflects the sustained performance of the processor. Thus, we also need to execute part of the computations on the processor to verify the results of the performance modeling method.

Communication Between Processor & FPGA: Besides workload partition between the FPGA and the processor, the communication and coordination between them are also very important.

First, the processor needs to notify the FPGA-based design to start and to be notified when the computations on the FPGA are complete. The status registers on the FPGA can be used for this purpose.

The second issue is the coordination of memory access. Since both the processor and the FPGA have access to the DRAM memory, memory accesses, especially memory writes, must be coordinated to avoid conflicts. Thus, during workload partition, we need to store the results generated by the processor and the FPGA in separate memory locations. In some cases, the FPGA needs the results of the computations on the processors and hence read-after-write hazards may occur. To avoid such hazards, the FPGA-based design cannot initiate read access to the DRAM memory of the processor. Also, the processor will not start writing to the SRAM memory of the FPGA until its computations are complete. Similarly, the processor cannot initiate read access to the SRAM memory.

2. Parallelism within FPGA

Although the typical clock speed of FPGA-based designs is much lower than that of the general-purpose processors, they can still achieve high performance for certain applications through spatial parallelism. By configuring multiple operators on-chip, an FPGA-based design could perform tens of, even hundreds of operations during each clock cycle. High I/O parallelism can also be achieved due to the large number of embedded BRAMs and I/O pins on the FPGA. However, the parallelism provided by an FPGA is also constrained by the available hardware resources on the device, as discussed in [20]. In this paper, we employ an existing FPGA-based design for matrix multiplication [20] to achieve parallelism on FPGAs.

3. Scalability Over Multiple Compute Nodes

In this paper, our algorithms are based on the overall system architecture in Figure 2. Designs must be scalable with regard to both the lower and higher hierarchies. Our proposed designs are scalable with regard to the number of compute nodes in the lower network hierarchy.

Prior research in distributed systems can be employed to enable the designs to scale in the upper level of network hierarchy. For example, we can can use our matrix multiplication design in the lower hierarchy and distributed algorithms in ScaLAPACK [4] in the upper hierarchy. However, this is beyond the scope of this paper.

5 Hybrid Designs for Example Applications

We have selected floating-point matrix multiplication and block LU decomposition as example applications. They are two widely used kernels in scientific computing. In this section, we propose hybrid designs for them on the reconfigurable computing systems.

5.1 Design for Matrix Multiplication

5.1.1 Architecture and Algorithm

Consider $C = A \times B$, where A, B and C are all $n \times n$ matrices. Each element of the matrices is a floating-point word. The architecture of the design for one group of compute nodes is shown in Figure 3. Suppose there are l compute nodes in one group. The nodes are denoted as $P_0, P_1, \ldots, P_{l-1}$; $FPGA_i$ and GPP_i denote the FPGA and the processor within node i, respectively. The columns of matrix A are grouped in multiple column stripes. Each stripe consists of $\frac{n}{k}$ submatrices of size $k \times k$, where k is the number of Processing Elements (PEs) on one FPGA. Similarly, the rows of B are grouped in multiple row stripes. The $k \times k$ submatrices in matrices A and B are denoted as A_{gh} and B_{gh}, $0 \leq g, h \leq \frac{n}{k} - 1$.

In our design, matrices A and B are initially stored in the DRAM memory of P_0, and are transferred to the other nodes in the group. During the computation, matrix A is read in column-major order, and matrix B is read in row-major order. One column stripe of matrix A and one row

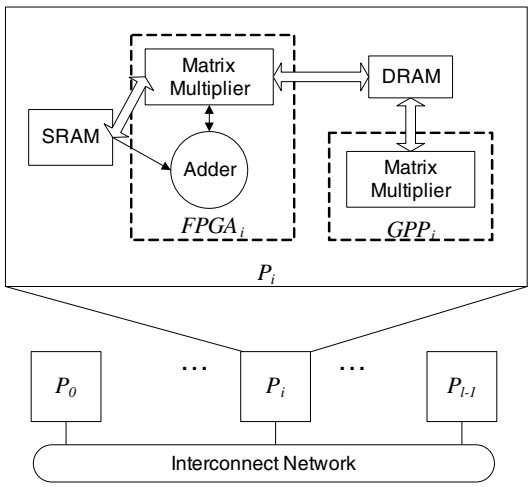

Figure 3. Design for matrix multiplication on multiple nodes

for node P_i $i = 0$ to $l - 1$ (in parallel) **do**
  get a column stripe of A, A_{ph} $(0 \leq p \leq \frac{n}{k} - 1)$
  store A_{ph} into memory
  get a row stripe of B, B_{hq} $(0 \leq q \leq \frac{n}{k} - 1)$
  if $i = q \mod l$ **then**
    store B_{hq} into memory
  end if
  for $p = 0$ to $(\frac{n}{k} - 1)$ **do**
    for every $B_{gq'}$ in the memory **do**
      read $C'_{pq'}$ from memory
      $C'_{pq'} <= C'_{pq'} + A_{ph} \times B_{hq'}$
      write $C'_{pq'}$ into memory
    end for
  end for
end for

Figure 4. Matrix multiplication algorithm on l nodes

stripe of matrix B are read and transferred at the same time. For each row stripe of B, P_i stores submatrices i, $(l+i)$, ..., $((\frac{n}{kl} - 1)l + i)$ into its DRAM memory. When A_{gq} is transferred to P_i, it is multiplied with $\frac{n}{kl}$ submatrices of B stored in the node. The row indices of these submatrices B are all the same as the column index of A_{gq}. The algorithm is shown in Figure 4. In the figure, C'_{gh} refers to the intermediate result of submatrices in matrix C, $0 \leq g, h \leq \frac{n}{k} - 1$. The final results of C are transferred back to P_0, and are written back to the DRAM memory.

In our design, P_i is in charge of computing $\frac{n}{l}$ columns of C, During the computations, each node needs all the elements of A and only $\frac{n}{l}$ columns of B.

5.1.2 Workload Partition

The workload of one compute node is further partitioned between the FPGA and the processor. Matrix multiplication is a well-known computationally intensive application, hence no performance profiling is needed for workload partition. Each node needs to perform $\frac{n^3}{l}$ floating-point multiplications and $\frac{n^3}{l}$ floating-point additions. One simple partitioning assigns n_f rows of A to FPGA and n_p rows to the processor, where $n_f + n_p = n$ and $\frac{n_f}{n_p} = \frac{O_p F_p}{O_f F_f}$. In this way, each FPGA multiplies an $n_f \times n$ and an $n \times \frac{n}{l}$ matrix. Each processor performs a $(n_p \times n) \times (n \times \frac{n}{l})$ matrix multiplication.

In such partition, there is no overlapping between the outputs of the FPGA and the processor. Thus no memory write conflicts occur. Within each node, the processor sends $n_f n$ elements of matrix A and $\frac{n^2}{l}$ elements of matrix B to the FPGA. When the computation is complete, the FPGA writes $\frac{n_f n}{l}$ elements of matrix C into the DRAM memory.

5.1.3 FPGA-based Matrix Multiplier

Each FPGA employs our FPGA-based design for matrix multiplication proposed in [20]. This design has k PEs connected in a linear array and is used to perform $k \times k$ matrix multiplication. The architecture of the design is shown in Figure 5. The PEs are labeled from left to right, as $PE_0, PE_1, \ldots, PE_{k-1}$. Each PE consists of one floating-point multiplier and one floating-point adder. The BRAM memory on the FPGA device serves as the internal storage of the PEs. We have proved that when the internal storage is of size $\Theta(k^2)$ words, the design achieves the optimal latency of $\Theta(k^2)$.

PE_0 reads submatrices of A and B from the SRAM memory, and transfers them along the linear array. The resulting submatrix is transferred in the opposite direction and are summed up with the intermediate results in the SRAM memory. As one word is read from and written to the SRAM memory in each clock cycle, the required bandwidth is $2F_f$ words/second.

Since each FPGA computes $n_f \times \frac{n}{l}$ elements of C, we need a storage of $\frac{n_f n}{l}$ words for the intermediate results of C. When the matrix size is large, block matrix multiplication is performed. Matrices A and B are partitioned into blocks of size $b \times b$. In this case, b_f rows of A are assigned to the FPGA, while b_p rows are assigned to the processor. We have $\frac{b_f b}{l} = M$, where M is the size of the SRAM memory of the FPGA.

5.1.4 Algorithm Analysis

We now derive the latency of our design. Assume we have l nodes. Suppose the DRAM memory bandwidth is B_{DRAM} and the interconnect bandwidth between any two nodes is

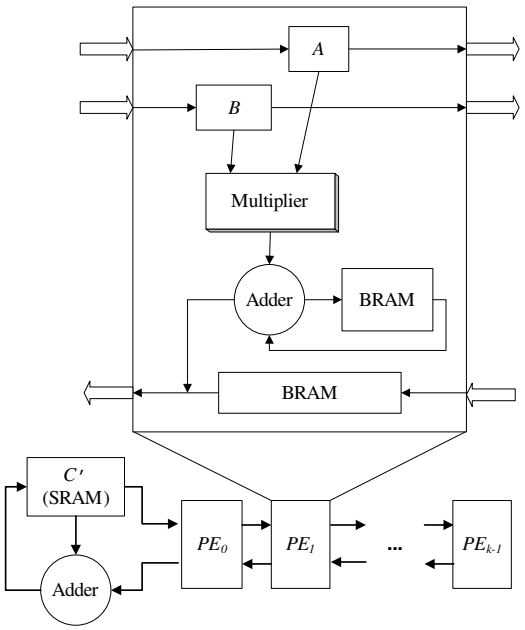

Figure 5. Architecture of FPGA-based matrix multiplier

B_N. In our design, $P_1, \ldots, P_{l-1}$ each receives all the elements of A and $\frac{n}{l}$ columns of B from P_0, and sends $\frac{n}{l}$ columns of C back to P_0. Thus, the latency for network communications is $T_N = (n^2(l-1) + \frac{2n^2(l-1)}{l})/B_N$. Within each node, the FPGA reads $n_f n$ elements of matrix A and $\frac{n^2}{l}$ elements of matrix B from the DRAM memory, and writes $\frac{n_f n}{l}$ elements of matrix C back. Thus, the latency for DRAM memory access is $T_{DRAM} = (n_f n + \frac{n^2}{l} + \frac{n_f n}{l})/B_{DRAM}$.

During the computation, each processor performs a $(n_p \times n) \times (n \times \frac{n}{l})$ matrix multiplication. Thus, the latency for computations on the processor is $T_p = \frac{n_p n^3}{l(O_p \times F_P)}$. Each FPGA multiplies an $n_f \times n$ and an $n \times \frac{n}{l}$ matrix, that is, performs $\frac{n_f}{k} \times \frac{n}{k} \times \frac{n}{kl}$ submatrix multiplies. As the effective latency for each submatrix multiply is k^2 FPGA clock cycles, the latency for computations on FPGA is $T_f = \frac{n_f n^2}{klF_f}$.

Based on the above analysis, the total latency of our design for $n \times n$ matrix multiplication is:

$$T = T_N + T_{DRAM} + \max(T_p, T_f)$$

This latency can be further reduced by overlapping the communications and computations. As we stated in Section 5.1.1, the columns in matrices A and B are grouped into stripes. Except for the first stripe of matrices A and B, the transferring of the stripes among the nodes are overlapped with the computations on the FPGA. Additionally, the FPGA writes its result back to the DRAM memory when the processor is performing computations.

5.2 Design for Block LU Decomposition

Matrix factorization is important in many scientific applications, including solving linear equations. One of the most commonly used methods for matrix factorization is LU decomposition. It factors an $n \times n$ matrix A into a $n \times n$ lower triangular matrix L and a $n \times n$ upper triangular matrix U. In this section, we propose a hybrid design for block LU decomposition for large n. Note that we assume that A is a nonsingular matrix and no pivoting has been implemented. Also, the diagonal entries of the resulting L matrix are all 1s. We follow the algorithm given in [12]. Due to space limitation, we do not give the algorithm description here. We only note that when the block size is $b \times b$, there are totally $\frac{n}{b}$ iterations; iteration t contains $(\frac{n}{b} - t)^2$ block multiplications of size $b \times b$.

There are l nodes in our design, and matrix A is initially stored in the DRAM memory of P_0. P_0 generates the operands for the block multiplications which are performed cooperatively by $P_1, \ldots P_{l-1}$. The workload of block multiplications are partitioned among the FPGAs and the processors in the same way as discussed in Section 5.1.2. Such workload partition does not cause memory access conflict. However, as the operands of $P_1, \ldots, P_{l-1}$ are generated by P_0, data hazards may occur. Thus, in our design, the processors are not allowed to initiate read access to the SRAM memory and the FPGAs are not allowed to initiate read access to the DRAM memory.

6 Experimental Results

6.1 Implementation on Reconfigurable Computing Systems

To illustrate our ideas, we implemented our designs on a Cray XD1 system. The lower hierarchy in XD1 is a chassis, which contains 6 nodes. Each node contains two AMD 2.2 GHz processor and one Xilinx Virtex-II Pro XC2VP50. The maximum SRAM memory bandwidth is 12.8 GB/s, and the DRAM memory bandwidth is 2.8 GB/s. The nodes in one chassis are connected through a non-blocking crossbar switching fabric which provides two 2 GB/s links to each node.

In XD1 and some other systems, to utilize the FPGA accelerator, two programs are needed. One is a VHDL program that describes the FPGA-based design. The target device, XC2VP50, contains 23616 slices and about 4 Mb of on-chip memory. For the FPGA-based design, We used Xilinx ISE 7.1i and Mentor Graphics ModelSim 5.7 development tools in our experiments [13, 19].

In our design, the implementations of floating-point adder and multiplier have no effect on the architecture or the algorithm. Therefore, these floating-point units can be

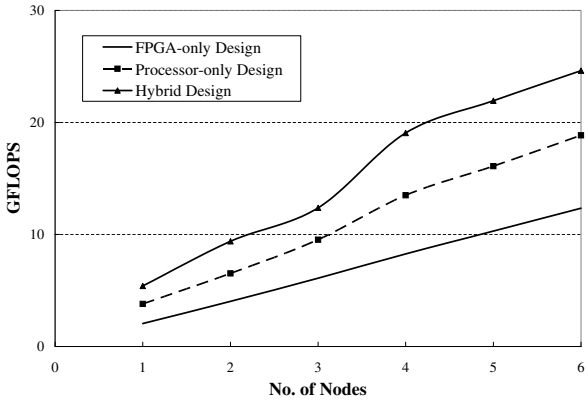

Figure 6. GFLOPS performance of our design for matrix multiplication

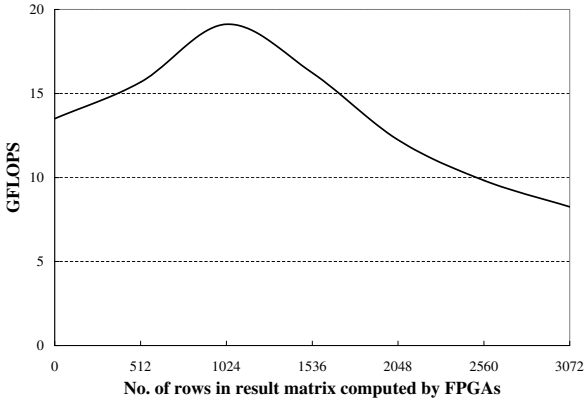

Figure 7. GFLOPS performance of our design for matrix multiplication vs. b_f ($b = 3072$, $l = 4$)

plugged into the design as modules, with no or few modifications to the interface between they and other parts of the architecture. In our experiments, we used our own 64-bit floating-point adders and multipliers that comply with IEEE-754 format. Their implementation details can be found in [10]. Table 1 gives the characteristics of floating-point units used by us.

Table 1. 64-bit Floating-Point Units

	Adder	Multiplier
No. of Pipeline Stages	11	8
No. of Slices	892	835
Clock Speed(MHz)	170	170

The other program is a C program which runs on the processor. It is in charge of file operations and memory accesses. The SRAM and BRAM memory on the FPGA are mapped into the memory space of the processor so that the processor can write to them directly. The results generated by the FPGA-based design is written to the DRAM memory directly. Besides the shared memory space, the processors and the FPGAs can also communicate through user-defined registers. These registers are located on the FPGAs, and can be read/written by the processors. The C program also handles the network communication using MPI standard. When the C program needs to perform part of the computations in the applications, we employ the AMD Core Math Library (ACML) [2]. Note that the C program only employs one AMD processor in each compute node.

6.2 Performance of Proposed Designs

We first implemented our FPGA-based matrix multiplier on one FPGA in XD1. At most 8 PEs can be configured on it, hence $k = 8$. Our design occupies about 89% of the total area of the device.

We then determined the workload partition between the FPGA and the processor within each node. As each PE performs two floating-point operations in each clock cycle, in our design, $O_f = 16$. The maximum achievable clock speed of the design is 130 MHz, hence $F_f = 130 \times 10^6$. The AMD processor runs at 2.2 GHz and yields at most two floating-point results in each clock cycle. Thus, $O_p = 2$ and $F_p = 2.2 \times 10^9$. According to the discussion in Section 4, $\frac{n_p}{n_f} = \frac{O_p \times F_p}{O_f \times F_f} = 2.1$. Thus, in our design, we set $n_p = 2n_f$. Such partition is also verified by our experiments. For 2048×2048 floating-point matrix multiplication, our FPGA-based architecture achieves a sustained performance of 2.0 GFLOPS. On the other hand, one AMD processor achieves 3.9 GFLOPS using *dgemm* routine.

We next implemented our design on multiple nodes in one chassis of XD1. Suppose we use 8 MB of the SRAM memory attached to the FPGA to store intermediate results of C. Therefore, $\frac{b_f \times (b_f + 2b_f)}{l} = 2^{20}$, and $b \approx 512\sqrt{l}$. Note that b needs to be a multiple of both l and k. Therefore in our experiments, when $l = 1, 2, 3$, $b_f = 512$, $b_p = 1024$ and $b = 1536$. When $l = 4, 6$, $b_f = 1024$, $b_p = 2048$ and $b = 3072$. When $l = 5$, $b_f = 1000$, $b_p = 2000$ and $b = 3000$.

Figure 6 shows the performance of our design when l increases from 1 to 6. For the purpose of comparison, we implemented two other designs. The "FPGA-only design" employs the FPGAs in the nodes only, while the "Processor-only design" employs only the processors. Note that the FPGA-only design still employs the processors in the compute nodes for data communication. The total amount of data transferred in these designs is the same as that in the hybrid design. From Figure 6, we see that all three designs are scalable, while our design achieves better performance than the other two. In particular, our designs achieves up to 1.4X speedup over the processor-only design, and about 2X speedup over the FPGA-only design.

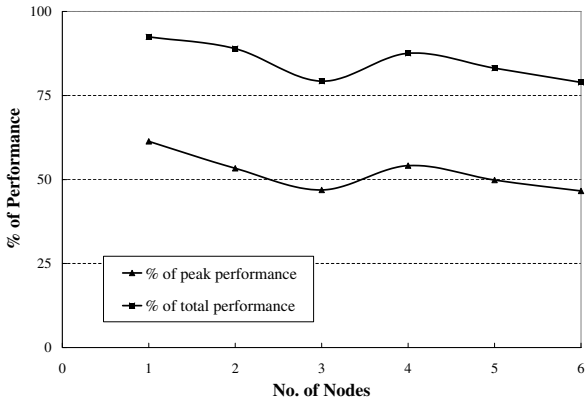

Figure 8. Percentage of peak performance achieved by our design for matrix multiplication

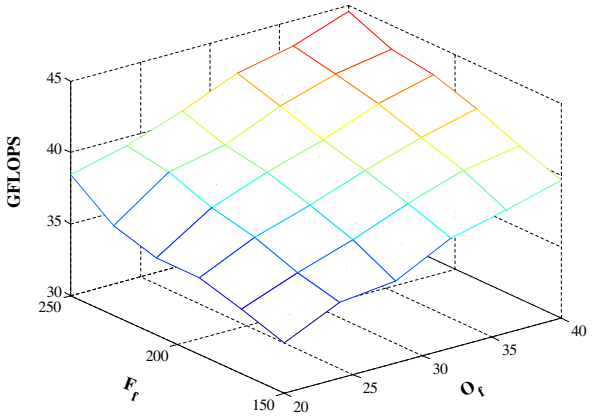

Figure 9. Projected performance of our design for matrix multiplication ($b_f = 2048$, $l = 6$)

We also examine the effect of the workload partition on the performance of our design. In the experiment, we set $b = 3072, l = 4$; b_f varies from 0 to 3072. Note that the size of required SRAM memory increases with b_f. The results of the experiment are shown in Figure 7. When b_f first becomes larger than 0, the processors in the system are utilized and the performance increases. The maximum performance of the design is achieved when $b_f = 1024$. After that, the FPGAs become overloaded and the processors are underloaded. Hence the performance of our design begins to decrease. This again verifies our workload partition $\frac{b_p}{b_f} = 2$.

For block LU decomposition, our design achieves about 4.5 GFLOPS with one FPGA and one AMD processor ($b_f = 512$, $b_p = 1024$ and $b = 1536$). One 1.8 GHz AMD processor achieves less than 3 GFLOPS [16] for LU decomposition; the upper bound on the performance of the FPGA for LU decomposition is its performance for matrix multiplication, and is 2.06 GFLOPS. Thus, our design achieves about 90% of the total performance of the FPGA and the processor.

6.3 Performance Comparison & Projection

We compare the sustained performance of our design for matrix multiplication to the peak performance. Suppose the peak performance of l nodes is denoted as $PEAK_l$. It is calculated as $PEAK_l = l \times (PEAK_f + PEAK_p)$, where $PEAK_f$ and $PEAK_p$ are peak performance of one FPGA and one processor, respectively. For the AMD processor in XD1, $PEAK_p$ can be computed as $O_p \times F_p$, which is 4.4 GFLOPS. $PEAK_f$ can be calculated assuming the FPGA is configured with only floating-point units without any control logic or routing overheads. With the floating-point units used in our experiments, $PEAK_f$ of XC2VP50 is 4.4 GFLOPS.

Figure 8 shows the percentage of peak performance that can be achieved by our hybrid design. Due to the MPI communication cost, the percentage of $PEAK_l$ that can be achieved by the design decreases as l increases. However, the curve rises slightly at $l = 4$. This is because b_p increases from 1024 to 2048, and the GFLOPS performance of *dgemm* routine increases with the problem size.

We also compare the performance of our design with the total performance of the FPGA-only design and the processor-only design. As shown in Figure 8, our design achieves more than 75% of the total performance. When $l = 4$, our achieves about 87% of the total performance. This shows that the cooperation and communication between the FPGA and the processor within a node cause few overheads. Also, our design effectively utilizes both the FPGA and the processor.

In our FPGA-based matrix multiplier, the implementations of floating-point units have no effect on the architecture or the control logic [20]. Therefore, when improved floating-point units are available, they can be plugged in easily. In this case, $O_f \times F_f$ will change accordingly, and hence the performance of our design will increase. Another way of improving performance is using a more powerful device. XC2VP50 is a relatively small device in the Xilinx Virtex-II Pro family. With a larger device, more PEs can be configured. For example, Xilinx Virtex-II Pro XC2VP100 contains 44096 slices, and can contain twice as many PEs as XC2VP50.

In our current implementation, $O_f = 16$ and $F_f = 130 \times 10^6$. We thus project the performance of our design for matrix multiplication when O_f ranges from 20 to 40 and F_f increases from 150 MHz to 250 MHz. We fix b_f to 2048, while b and b_p vary. From Figure 9, when $O_f = 40$ and F_f = 250 MHz, with 6 nodes, our design can achieve

44.6 GFLOPS. Note that the projected performance is computed assuming the DRAM memory bandwidth and the interconnect bandwidth available in XD1. Also, even with the highest F_f, the required SRAM memory bandwidth of the design is lower than the available bandwidth in XD1.

7 Conclusion

We have proposed hybrid designs for high-end reconfigurable computing systems. These systems contain multiple compute nodes that are connected through interconnect network. Each node contains both general-purpose processors and FPGAs. We modeled these systems as a two level hierarchical distributed system. Based on the computational model, we identified the design challenges, including the workload partition between the processors and FPGAs, coordination and communication between the processors and FPGAS, and scalability over multiple compute nodes. We proposed designs for floating-point matrix multiplication and block LU decomposition. We have shown that in our designs, the communication between the FPGA and the processor within a node cause little overhead. When implemented on Cray XD1, our designs effectively utilize both the processors and the FPGAs in the system. Our designs achieve up to 1.4X speedup over designs that use the processors only, and 2X speedup over designs that use the FPGAs only. Our designs are scalable over multiple nodes.

References

[1] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt and R. van de Geijn. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. *Concurrency: Practice and Experience*, 9(9):837–857, 1997.

[2] AMD Core Math Library. `http://developer.amd.com/acml.aspx`.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK user's guide third edition. `http://www.netlib.org/lapack/lug/lapack_lug.html`, August 1999.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[5] Cray Inc. `http://www.cray.com/`.

[6] M. deLorimier and A. DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *Proc. of the 13th ACM International Symposium on Field-Programmable Gate Arrays*, California, USA, February 2005.

[7] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.

[8] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev. 64-bit Floating-Point FPGA Matrix Multiplication. In *Proc. of the 13th International Symposium on Field Programmable Gate Arrays*, California, USA, February 2005.

[9] GNU gprof. `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html`.

[10] G. Govindu, R. Scrofano, and V. K. Prasanna. A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing. In *Proc. of International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2005.

[11] Intel. `http://www.intel.com`.

[12] Jaeyoung Choi and J. J. Dongarra and L. S. Ostrouchov and Petitet and A. P. and D. W. Walker and R. C. Whaley. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5(3):173–184, Fall 1996.

[13] Mentor Graphics Corp. `http://www.mentor.com/`.

[14] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, 1994.

[15] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. `http://www.netlib.org/benchmark/hpl/index.html`.

[16] M. Pont. *High-Performance Math Libraries - Who says you can't get performance and accuracy for free?* Numerical Algorithms Group, March 2005. `http://www.nag.co.uk/IndustryArticles/HighPerformanceMathLibraries.pdf`.

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

[18] K. D. Underwood and K. S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Proc. of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2004.

[19] Xilinx Incorporated. `http://www.xilinx.com`.

[20] L. Zhuo and V. K. Prasanna. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. In *Proc. of the 18th International Parallel & Distributed Processing Symposium*, New Mexico, USA, April 2004.

[21] L. Zhuo and V. K. Prasanna. High Performance Linear Algebra Operations on Reconfigurable Systems. In *Proc. of SuperComputing 2005*, Washington, USA, November 2005.

[22] L. Zhuo and V. K. Prasanna. Sparse Matrix-Vector Multiplication on FPGAs. In *Proc. of the 13th ACM International Symposium on Field-Programmable Gate Arrays*, California, USA, February 2005.