Hardware/Software Co-Design for Matrix Computations 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 scientific computations. However, the co-existence of the processors and the FPGAs in such systems also poses new challenges to application developers. In this paper, we investigate a design model for hybrid designs, that is, designs that utilize both the processors and the FPGAs. The model characterizes a reconfigurable computing system using various system parameters, including the floating-point computing power of the processor and the FPGA, the number of nodes, the memory bandwidth and the network bandwidth. Using the model, we investigate hardware/software co-design for two computationally intensive applications: matrix factorization and all-pairs shortest-paths problem. Our designs balance the load between the processor and the FPGA, as well as overlap the computation time with memory transfer time and network communication time. The proposed designs are implemented on 6 nodes in a Cray XD1 chassis. Our implementations achieve 20 GFLOPS and 6.6 GFLOPS for these two applications, respectively.

1 Introduction

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). Due to their low computing density, early FPGAs were mainly used for applications that were not computationally demanding. However, with rapid advances in technology, current FPGA devices contain much more resources than their predecessors [20]. Thus, FPGAs have been employed to accelerate various scientific applications and have achieved superior performance compared with general-purpose processors [19, 21].

With the increasing computing power of FPGAs, high-end systems that employ FPGAs as application-specific accelerators have been built. Such systems include SRC 6 and 7 [16], Cray XD1 and XT3 [4], SGI RASC [15], among others. These systems contain multiple compute nodes (also called nodes) that are connected through an interconnect network. Each node contains both FPGAs and general-purpose processors. These reconfigurable computing systems provide multiple levels of parallelism. Coarse-grain parallelism can be employed on multiple computing nodes, while fine-grain parallelism can be explored on the FPGAs. Moreover, within each node, the processors and the FPGAs can collaborate to compute the tasks assigned to the node. Thus, such systems can achieve higher performance than systems with processors only.

However, to effectively utilize a reconfigurable computing system, certain design challenges have to be addressed. One of them is the workload partition between hardware (FPGA accelerators) and software (executed on the processors) within each node. An efficient partitioning technique should consider not only the computing power of the processor and the FPGA, but also the data transfer time between them. Another design challenge is the workload distribution among multiple nodes. Moreover, when data are exchanged among the nodes, the hardware/software co-design within each node needs to take network communication costs into consideration. To the best of our knowledge, these issues have not been addressed in previous works.

In this paper, we propose a **design model** for hybrid designs on reconfigurable computing systems for a class of applications. The designs are hybrid in that they utilize **both the processors and the FPGAs in the system**. The model provides various design parameters to analyze a system, including the floating-point computing power of the processor and the FPGA, the number of nodes, the memory bandwidth

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

as well as the network bandwidth. Using the model, we propose designs for two example applications, LU decomposition for matrix factorization and Floyd-Warshall algorithm [3] for all-pairs shortest-paths problem. Both of these applications are used extensively in scientific computing.

For each example application, we identify various tasks. Based on the inherent attributes of the tasks and the system parameters, hardware/software partitioning is performed. For LU decomposition, we partition the computations of one type of tasks (block matrix multiplication) between the processor and the FPGA, and perform all the other types of tasks on the processor only. For the Floyd-Warshall algorithm, the tasks are not partitioned because they contain a large number of data dependencies. Instead, a task is assigned entirely either to the processor or to the FPGA. The partition is further optimized by overlapping the computations with the data transfer and network communications.

To illustrate our ideas, we implemented our designs on Cray XD1. In our implementations, 6 nodes are used. Within each node, our designs employ a 2.2 GHz AMD Opteron processor and a Xilinx XC2VP50 FPGA. The nodes communicate using MPI (Message Passing Interface) [13]. Two baseline designs are used for performance comparison. "FPGA-only design" employs the FPGAs in the nodes only, while "Processor-only design" employs only the processors. Using 6 nodes in one chassis of XD1, our hybrid design achieves 20 GFLOPS for LU decomposition. It achieves 1.3X speedup and 2X speedup over the Processor-only design and the FPGA-only design, respectively. For the Floyd-Warshall algorithm, the hybrid design achieves 6.6 GFLOPS. It achieves 5.8X speedup and 1.15X speedup over the Processor-only design and the FPGA-only design, respectively. Experimental results also show that our designs achieve more than 80% of the sum of the performance of the two baseline designs. The proposed design model can also be used for performance prediction of a given application. Our designs achieve more than 85% of the performance predicted by the design model.

The rest of the paper is organized as follows. Section II introduces related work. Section III discusses representative systems and architectural model of the reconfigurable computing systems. Section IV presents the design model for hybrid designs. Section V presents our hybrid designs for the example applications. Section VI presents the experimental results on XD1 and analyzes the performance. Section VII concludes the paper.

2 Related Work

In [9], an FPGA-based design for Conjugate Gradient iterative method is developed using a HLL (high-level language)-to-HDL (hardware description language) compiler on an SRC reconfigurable computer. In [17], a design for all-pairs shortest-paths is implemented on one node of Cray XD1. In these designs, only the computing power of the FPGA is utilized, while the computing power of the processor is mostly unused.

In [14], the authors partition the tasks in a molecular dynamics simulation system between the processor and the FPGA. They then model the performance of several alternatives for the tasks mapped to the FPGA. We also proposed a simple method for partitioning matrix multiplication and block LU decomposition in [22]. In contrast to these results, in this paper, we propose a generic design model that can be applied to a class of applications. In addition, our model not only considers the computing power of the processor and the FPGA, but also the data transfer time and the network communication costs.

Various programming models have also been proposed for reconfigurable computing systems. The work in [2] starts from high-level representation, and provides methods to explore hardware/software trade-offs based on profiling of the source code. In [6], a unified programming model is presented for specifying application threads from a single application program. The threads are either compiled to run on the processor or synthesized to run on the FPGA. These efforts focus on facilitating the programming and compilation on a single node in a reconfigurable computing system. On the other hand, our work focuses on the effective utilization of both the processor and the FPGA within multiple nodes.

3 Reconfigurable Computing Systems

Many reconfigurable computing systems have become available. One representative system is Cray XD1 [4]. The basic architectural unit of XD1 is a compute blade, which contains two AMD 2.2 GHz processors and one Xilinx Virtex-II Pro XC2VP50. Each FPGA has access to four banks of QDR II SRAM. The maximum FPGA-SRAM memory bandwidth is 12.8 GB/s. Through Cray's RapidArray processors, the FPGA can access the DRAM of the processors at a bandwidth of 2.8 GB/s. Six compute blades fit into one chassis, connected through a non-blocking crossbar switching fabric which provides two 2 GB/s links to each node. The recently introduced supercomputer of Cray, XT3, also supports reconfigurable computing by incorporating FPGA modules from DRC [5]. Each DRC module contains a Virtex-4 FPGA and SRAM memory of up to 64 MB. The module has access to adjacent Opteron processors and DRAM memory at a bandwidth of up to 6.4 GB/s.

Other systems include SRC reconfigurable computers [16] and SGI RASC [15]. SRC Computers offers the MAP processor which includes two Xilinx Virtex-II Pro FPGAs, in single-MAP workstations or multiple-MAP clusters [16]. SGI builds the RC100 Blade which has two Xilinx Virtex-

4 FPGAs. Each blade is directly connected to the shared global memory.

A reconfigurable computing system can be seen as a distributed system with multiple nodes. The nodes are connected through an interconnect network. Each node can be based on either general-purpose processors, FPGAs, or both. Each FPGA contains certain amount of on-chip memory, usually Block RAMs (BRAMs). The FPGAs also has access to off-chip but on-board memory, which is SRAM. Through the connection between the processors and the FPGAs, an FPGA can also access the the main memory of the general-purpose processor (Cray systems and SRC computers) or the shared global memory (SGI RASC), which is DRAM.

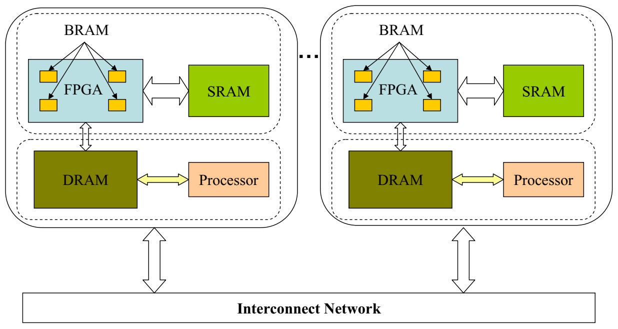

Figure 1. Architectural Model for Reconfigurable Computing Systems

In this paper, we only consider systems where each node consists of both FPGAs and processors. Currently, in such systems, only the processors of the nodes are connected to the interconnect network. The architectural model of these systems is shown in Figure 1.

4 Design Model and Partitioning Strategy

We propose a design model for hybrid designs on reconfigurable computing systems. Our model targets computationally intensive applications which are suitable for hardware acceleration, such as matrix computations. The design methodology using the model takes the following steps:

1. Identify various tasks within an application. The computational complexity of the tasks as well as the dependencies among them are then analyzed.

2. Characterize the hardware resources and the computing capacity of a system using various parameters.

3. Perform hardware/software partitioning based on the system parameters and the attributes (computational complexity, data dependencies, etc) of the tasks.

4. Improve the partition by overlapping the computations with data transfer and network communications (if multiple nodes are employed).

In the rest of this section, we give more details of our design model. As task identification is totally dependent on the application, it is discussed in Section 5 for each example application. Note that our model is unsuitable for performing hardware/software co-design for control-intensive applications or applications that contain few computationally intensive tasks.

4.1 System Parameters

We assume that a reconfigurable computing system consists of p nodes interconnected by a high bandwidth, low latency network. Without loss of generality, we assume each node has one processor and one FPGA. We characterize the system using the following parameters:

O_f: number of floating-point operations performed per clock cycle by the FPGA in a node;

O_p: number of floating-point operations performed per clock cycle by the processor in a node;

F_f: clock speed (number of clock cycles per second) of the FPGA-based design;

F_p: clock speed of the processor;

B_d: DRAM memory bandwidth available to the FPGA-based design, measured as the number of bytes transferred between the FPGA and DRAM per second;

B_n: network bandwidth between any two nodes, measured as the number of bytes transferred per second;

b_w: word width. In this paper, we consider double-precision floating-point numbers. Thus, $b_w = 8$ bytes.

The computing power is defined as the number of floating-point operations performed in a second. For both the processor and the FPGA, the computing power is dependent on the application. For an FPGA-based design, we can get exact values of O_f and F_f, and hence the computing power of the FPGA is calculated as $O_f \times F_f$. For the processor, we use $O_p \times F_p$ to refer to the sustained performance for a given application. It is obtained by executing a sample program of the application.

Note that our model does not consider the memory access latency. This is because for the applications considered, data are streaming into and out of the FPGA. Therefore, the memory access latency is only incurred only once.

4.2 Workload Partition

A simple way of partitioning a task within a node is to execute the computationally intensive part on the FPGA, and use the processor for the control intensive part. However, the computing power of the processor is mostly wasted. Therefore, we partition the workload of an application between the hardware and software so that both the processor and the FPGA are fully utilized. Suppose N_p floating-point operations are assigned to the processor and N_f operations are assigned to the FPGA. Thus, the computation time (time for execution of the task) of the processor $T_p = \frac{N_p}{O_p \times F_p}$, and the computation time of the FPGA $T_f = \frac{N_f}{O_f \times F_f}$. We can choose N_p and N_f so that $T_p \approx T_f$.

However, the above partition is not accurate because it does not consider the data transfer time inside a node. As the input data are stored in the DRAM memory, they have to be transferred to the FPGA. The data are streamed into the FPGA so that the computations on the FPGA can be overlapped with the data transfer. However, the computations on the processor cannot begin until the transfer is completed. Thus, data transfer time has to be included in the workload partition. Suppose there are D_f bytes of input data to be transferred to the FPGA. We have

$$T_p + \frac{D_f}{B_d} = T_f \qquad (1)$$

At the end of the application, the result data generated by the FPGA also need to be transferred back to the DRAM. As such transfer is initiated by the FPGA, it can be overlapped with the computations on the processor. Moreover, because of spatial parallelism, the transfer can also be overlapped with the computations on the FPGA. Thus, as an approximation, we do not consider this data transfer time.

Some tasks contain a large number of data dependencies and are not suitable for partitioning. For these tasks, the coordination and communication between the processor and the FPGA may become the performance bottleneck. Therefore, such tasks are assigned entirely to either the processor or the FPGA. In this case, $T_p \approx T_f$ is achieved by tuning the numbers of tasks assigned to the processor and the FPGA.

4.3 Multiple Nodes

An application can also employ multiple nodes in the reconfigurable computing systems. Since the nodes communicate through the processors, the computations on the processors cannot overlap with the network communications. On the other hand, the computations on the FPGAs are not affected. Thus, the partition needs to be adjusted so that the communication costs are included. Suppose D_p bytes of data need to be transferred between two nodes, Equation 1 should be modified to:

$$T_p + \frac{D_f}{B_d \times F_f} + \frac{D_p}{B_n} = T_f \qquad (2)$$

When the tasks are assigned to multiple nodes, load balancing among the nodes is important. If a node is overloaded, it will become the performance bottleneck in the system. Thus, in our model, we need to adjust the number of tasks assigned to each node so that the execution time of each node is approximately equal.

4.4 Hardware/Software Coordination

Besides workload partition, the coordination between the processor and the FPGA is also important in the design model. First, the processor needs to notify the FPGA-based design to start. It also needs to be notified when the computations on the FPGA are completed. The status registers in the FPGA can be used for these purposes. As the latency for the processor to check the registers is negligible compared with the computation time of a task, we do not consider it in our model. Nonetheless, the frequency of coordination is given for each example application.

The second issue is the coordination of memory accesses. As both the processor and the FPGA have access to the DRAM memory, memory accesses, especially memory writes, must be coordinated to avoid conflicts. Thus, we need to make sure that the processor and the FPGA write to separate memory locations. Another coordination issue is data dependency between the processor and the FPGA. For example, if the FPGA reads the input from the DRAM memory before the computations on the processor are completed, read-after-write hazards may occur. Therefore, in our model, the FPGA cannot read the DRAM memory before getting permission from the processor. Similarly, the processor has to get permission from the FPGA before reading the SRAM memory.

4.5 Performance Prediction

Our design model can also be used for performance prediction. After the values of the system parameters are determined, the workload for a given application is partitioned following the model. Then we can calculate the total execution time for the application on both the processor (T_{tp}) and the FPGA (T_{tf}) based on the data dependencies among the tasks. Moreover, for simplicity, we assume all the data transfer and network communications are overlapped with the computations on the FPGA. Thus, the predicted total latency of the design is $\max\{T_{tp}, T_{tf}\}$. In Section 6, we compare the predicted performance with the experimental results.

5 Example Applications

5.1 Design for LU Decomposition

LU decomposition factors an $n \times n$ matrix A into an $n \times n$ lower triangular matrix L and an $n \times n$ upper triangular matrix U. The diagonal entries of the resulting L matrix are all 1s. As customary in hardware implementation of matrix factorization, we assume that A is a nonsingular matrix and no pivoting is needed. For small matrices, we can use the LU decomposition algorithm described in [3].

5.1.1 Algorithm Description

In our work, we consider LU decomposition of large matrices. Therefore, we follow the block algorithm given in [10], which is described as follows. At the beginning of the algorithm, there are four matrices: A^0_{00}, A^0_{01}, A^0_{10}, and A^0_{11}. A^0_{00} is a $b \times b$ matrix, A^0_{01} is a $b \times (n-b)$ matrix, A^0_{10} is an $(n-b) \times b$ matrix, and A^0_{00} is an $(n-b) \times (n-b)$ matrix. The goal of the algorithm is to decompose A into two matrices, L and U, such that

$$\begin{pmatrix} A^0_{00} & A^0_{01} \\ A^0_{10} & A^0_{11} \end{pmatrix} = \begin{pmatrix} L^1_{00} & 0 \\ L^1_{10} & L^1_{11} \end{pmatrix} \begin{pmatrix} U^1_{00} & U^1_{01} \\ 0 & U^1_{11} \end{pmatrix} \quad (3)$$

The steps of the algorithm are as follows:

1. Perform a sequence of Gaussian eliminations on the $n \times b$ matrix formed by A^0_{00} and A^0_{10} in order to calculate the entries of L^1_{00}, L^1_{10}, and U^1_{00};
2. Calculate U^1_{01} as the product of $(L^1_{00})^{-1}$ and A^0_{01};
3. Evaluate $A^1_{11} \leftarrow A^0_{11} - L^1_{10}U^1_{01}$;
4. Apply steps 1 to 3 recursively to matrix A^1_{11}. During the tth ($0 \le t \le \frac{n}{b}-1$) iteration, the initial matrices are A^t_{00}, A^t_{01}, A^t_{10}, A^t_{11}; the resulting matrices L^t_{00}, U^t_{00}, L^t_{10}, U^t_{01} and A^{t+1}_{11} are obtained. An *iteration* denotes an execution of steps 1 to 3.

5.1.2 Task Identification

Five tasks have been identified in block LU decomposition: *opLU*, *opL*, *opU*, *opMM* and *opMS*. In the tth iteration, *opLU* is performed in step 1 to obtain L^t_{00} and U^t_{00}. $(\frac{n}{b} - t)$ *opL* operations are performed in step 1 to obtain L^t_{10} using A^t_{10} and $(U^t_{00})^{-1}$. In step 2, $(\frac{n}{b} - t)$ *opU* operations generate U^t_{01} using A^t_{01} and $(L^t_{00})^{-1}$. In step 3, block matrix multiplications (*opMM* operations) and matrix subtractions (*opMS* operations) are performed for $(\frac{n}{b} - t)^2$ times. *opL* and *opU* operations need the outputs of *opLU*; *opMM* operations need the outputs of *opL* and *opU* operations; *opMS* operations need the outputs of *opMM* operations.

Among the five operations, *opMS* is the least computationally intensive ($\Theta(n^2)$) and does not need hardware acceleration. The other operations are all computationally intensive, with a complexity of $\Theta(n^3)$. However, it is impractical to partition *opLU*, *opL* and *opU* between the processor and the FPGA because they contain a lot of data dependencies. Therefore, in our design, only *opMM* is performed on both the processor and the FPGA. All the other operations are performed on the processor.

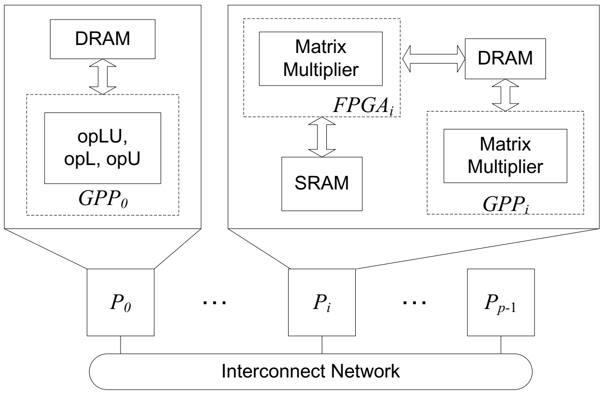

Figure 2. Architecture of our design for LU decomposition in 0th iteration

5.1.3 Proposed Design

We now present our hybrid design for LU decomposition. Let p nodes be denoted as P_0, P_1, $\ldots$, P_{p-1}. Matrix A is partitioned into $b \times b$ blocks which are denoted as A_{uv} $(0 \le u, v \le \frac{n}{b} - 1)$. Initially, P_i $(0 \le i \le p-1)$ stores A_{iv} and A_{ui} $(i \le u, v \le \frac{n}{b} - 1)$, $A_{(i+p),v}$ and $A_{u,(i+p)}$, $(i+p \le u, v \le \frac{n}{b} - 1)$, $\ldots$, $A_{(i+p(\frac{n}{bp}-1)),v}$ and $A_{u,(i+p(\frac{n}{bp}-1))}$ $(i + p(\frac{n}{bp} - 1) \le u, v \le \frac{n}{b} - 1)$.

In the tth iteration, $P_{t'}$ performs *opLU*, *opL*, *opU* operations on its processor, where $t' = t \mod p$. After one *opL* and one *opU* are completed, their outputs are transferred to the other nodes where *opMM* operations are performed on both the processors and the FPGAs. The outputs of the *opMM* operations, A'_{uv} $(t+1 \le u, v \le \frac{n}{b} - 1)$, are sent to $P_{t''}$, where $t'' = \max\{u, v\}$. $P_{t''}$ performs $A_{uv} = A_{uv} - A'_{uv}$, $t+1 \le u, v \le \frac{n}{b} - 1$. The architecture of our design in the 0th iteration is shown in Figure 2. *FPGA$_i$* and *GPP$_i$* in the figure refer to the FPGA and the processor of node i, respectively.

In our design, $b \times b$ block matrix multiplications are performed by $p-1$ nodes together. For $E = C \times D$, the columns of matrix C are grouped in column stripes. Each stripe consists of $\frac{b}{k}$ submatrices of size $k \times k$, where k is the number of Processing Elements (PEs) on one FPGA [21]. Similarly, the rows of D are grouped in row stripes. Suppose $P_{t'}$ contains matrix C and D. It transfers the column stripes of matrix C and the row stripes of matrix D to the other nodes. P_i $(0 \le i \le p-1, i \ne t')$ stores $\frac{1}{p-1}$ of D stripe to its memory and generates $\frac{n}{p-1}$ columns of E. In particular, P_i needs to perform $\frac{b^3}{p-1}$ floating-point multiplications and $\frac{b^3}{p-1}$ floating-point additions. We need to partition this workload between the processor and the FPGA.

Workload Partition We proposed a partitioning scheme on a node for matrix multiplication in [22]. The partition as-

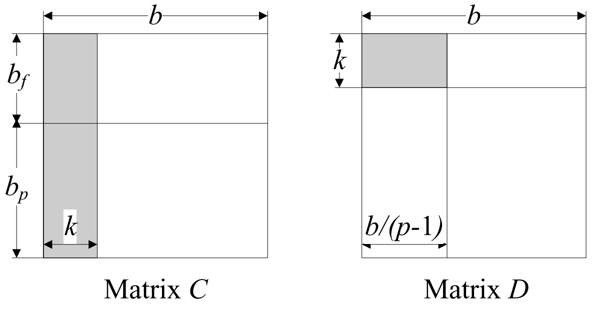

Figure 3. Workload partition for multiplying one column stripe of C and one row stripe of D

signs b_f rows of matrix C to FPGA and b_p rows to the processor, where $b_f + b_p = b$ and $\frac{b_f}{b_p} = \frac{O_p F_p}{O_f F_f}$. When p nodes are employed, each FPGA multiplies a $b_f \times b$ and a $b \times \frac{b}{p-1}$ matrix. Each processor performs a $(b_p \times b) \times (b \times \frac{b}{p-1})$ matrix multiplication. On each node, SRAM memory of size $\frac{b_f b}{p-1}$ is needed to store the intermediate results. The partition is shown in Figure 3. However, in such partition, the data transfer time and the network communication costs are not considered. Thus, we improve the partition in this paper as follows.

For performing $E = C \times D$, it takes $T_{comm} = \frac{2bkb_w}{B_n}$ to transfer one column stripe of C and one row stripe of D from $P_{t'}$ to P_i $(i \neq t')$. Within P_i, the processor then moves $b_f k$ elements of matrix C and $\frac{bk}{p-1}$ elements of matrix D from its DRAM memory to the FPGA. T_{mem} thus equals $\frac{(b_f k + bk/p-1)b_w}{B_d}$. After transferring the data, the processor of P_i performs a $(b_p \times k) \times (k \times \frac{b}{p-1})$ matrix multiplication. Thus, the computation time of the processor is $T_p = \frac{2b_p bk}{(p-1)(O_p \times F_P)}$. The FPGA of P_i multiplies a $b_f \times k$ and a $k \times \frac{b}{p-1}$ matrix. That is, it performs $\frac{b_f}{k} \times 1 \times \frac{b}{k(p-1)}$ $(k \times k)$ submatrix multiplies. As the effective latency for each submatrix multiply is k^2 FPGA clock cycles [21], the computation time of the FPGA is $T_f = \frac{b_f b}{(p-1)F_f}$.

Except for the first stripes of C and D, T_{comm} and T_{mem} can be overlapped with T_f to reduce latency. Thus, the values of b_p and b_f should be determined so that

$$T_f = T_{comm} + T_{mem} + T_p \qquad (4)$$

Note that T_{mem} does not include the time for sending results back from the FPGA to the DRAM memory. This is because such sending can be done without interrupting the computations on the processor and the FPGA.

Hardware/Software Coordination In the above partitioning, the processor generates b_p rows of E and the FPGA generates b_f rows of E. As they write to separate memory locations, there will not be access conflict. For the multiplication of one column stripe of C and one row strip of D, the processor needs to signal the FPGA to start computations as well as be notified when the FPGA is done. Thus, the frequency of coordination is $\frac{2}{T_f} = \frac{2(p-1)F_f}{b_f b}$ times per second.

Load Balancing While the other nodes are performing *opMM* operations, $P_{t'}$ performs operations *opLU*, *opL* and *opU*. According to the discussion above, the total latency of one *opMM* equals $\frac{b}{k} \times \frac{b_f b}{(p-1)F_f} = \frac{b_f b^2}{(p-1)kF_f}$. Suppose when $P_{t'}$ performs one *opLU*/*opL*/*opU* , the other nodes perform l *opMM* operations. If the latency of performing one *opLU*, *opL* and *opU* operation is T_{lu}, T_{opl} and T_{opu}, for load balancing, l should be determined so that

$$\max\{T_{lu}, T_{opl}, T_{opu}\} + \frac{lb}{k} \times T_{comm} = \frac{lb_f b^2}{(p-1)kF_f} \qquad (5)$$

We include the communication costs because $P_{t'}$ also needs to send the block matrices for *opMM* operations to the other nodes.

5.2 Design for Floyd-Warshall Algorithm

Consider a weighted and directed graph G with n vertices. The all-pairs shortest-paths problem is the problem to find a shortest (least-weight) path between every pair of vertices in the graph. The Floyd-Warshall algorithm [3] is an efficient technique to solve this problem.

5.2.1 Algorithm Description

In our work, we focus on all-pairs shortest-paths problems with large problem sizes. Therefore, we follow a blocked version of the Floyd-Warshall algorithm [7]. Suppose matrix D^0 is partitioned to blocks of size $b \times b$. The blocks are denoted as D^0_{uv} $(0 \leq u, v \leq \frac{n}{b} - 1)$. There are totally $\frac{n}{b}$ iterations in the algorithm. D^{t+1} is used to denote the matrix generated after iteration t $(0 \leq t \leq \frac{n}{b} - 1)$. In iteration t, there are three steps:

1. Perform the Floyd-Warshall algorithm in [3] on block D^t_{tt}. Such operation is denoted as *op*1;
2. Perform the Floyd-Warshall algorithm in [3] on block D^t_{tq} $(0 \leq q \leq \frac{n}{b} - 1$, $q \neq t)$ using the columns of D^t_{tq} and the rows of D^t_{tt}. Similarly, perform the regular Floyd-Warshall algorithm on block D^t_{qt} using the rows of D^t_{qt} and the columns of D^t_{tt}. We denote these operations as *op*21 and *op*22, respectively;
3. Perform the Floyd-Warshall algorithm in [3] on the remaining blocks. For block D^t_{uv} $(0 \leq u, v \leq \frac{n}{b} - 1$, $u \neq t, v \neq t)$, the rows of D^t_{tv} and the columns of D^t_{ut} are needed. Such operation is called *op*3.

5.2.2 Task Identification

The operations in the steps are identified as the tasks in the block Floyd-Warshall algorithm. In each iteration, one $op1$ operation, $\frac{n}{b} - 1$ $op21$ operations, $\frac{n}{b} - 1$ $op22$ operations and $(\frac{n}{b} - 1)^2$ $op3$ operations are performed. $op21$ and $op22$ operations need the outputs of $op1$; $op3$ operations need the outputs of $op21$ and $op22$.

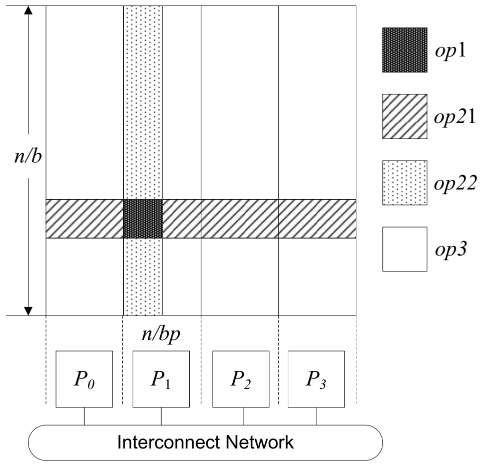

Figure 4. Illustration of operations of the nodes for Floyd-Warshall algorithm ($\frac{n}{b} = 8$, $p = 4$, $t = 2$).

All the tasks in block Floyd-Warshall algorithm are computationally intensive and have a complexity of $\Theta(n^3)$. Except the inputs, these tasks are all the same. Therefore, we can use the same processor-based algorithm and FPGA-based design for these operations. As the tasks contain a large number of data dependencies, they are not suitable for partitioning between the processor and the FPGA.

5.2.3 Proposed Design

Suppose there are p nodes in the system, denoted as P_0, P_1, ..., P_{p-1}. Initially, each node stores $\frac{n}{bp}$ columns of the $b \times b$ blocks in D^0. In particular, P_i stores columns $\frac{in}{bp}, \frac{in}{bp} + 1, \ldots, \frac{(i+1)n}{bp} - 1$ $(0 \le i \le p - 1)$.

In iteration t $(0 \le t \le \frac{n}{b} - 1)$, there are $\frac{n}{b}$ phases. In phase 0, $op1$ operation is performed on block D^t_{tt} by $P_{t'}$, where $t' = \lceil \frac{t}{n/bp} \rceil$. The resulting D^t_{tt} is then transferred to all the other nodes. When P_i $(0 \le i \le p - 1, i \ne t')$ performs $\frac{n}{bp}$ $op21$ operations, $P_{t'}$ performs $\frac{n}{bp} - 1$ $op21$ operations and one $op22$ operation. In the following phase, the result of the $op22$ operation is transferred to the other nodes for $op3$ operations. When P_i $(0 \le i \le p - 1, i \ne t')$ performs $\frac{n}{bp}$ $op3$ operations, $P_{t'}$ performs $\frac{n}{bp} - 1$ $op3$ operations and one $op22$. Again, the result of the $op22$ operation is transferred to the other nodes for $op3$ operations. This continues until all the $op22$ operations and $op3$ operations are completed.

Note that except phase 0, each node performs the same number of operations in each phase. Thus, load balancing is maintained in the system. Figure 4 illustrates the operations of the nodes when $\frac{n}{b} = 8$, $p = 4$ and $t = 2$. Different operations of the application are shown in different patterns.

Workload Partition Within each node, entire operations are assigned to the processor and the FPGA. Each of the operations contains b^3 floating-point additions and b^3 floating-point comparisons. Thus, the computation time of the processor $T_p = \frac{2b^3}{O_p \times F_p}$. On the FPGA, We employ the design proposed in [18]. In that design, with k floating-point adders and k floating-point comparators, the latency of performing the regular Floyd-Warshall algorithmon on a $b \times b$ matrix takes $\frac{2b^3}{k}$ clock cycles. Therefore, $T_f = \frac{2b^3}{kF_f}$. The on-chip memory required by the design is of size $2k^2$ words, and the size of required on-board memory is $2b^2$ words.

We aim to share the workload between the processor and the FPGA. In each phase discussed above, each node performs $\frac{n}{bp}$ operations and sends/receives one block. When an operation is implemented on the FPGA, two blocks need to be accessed from the DRAM memory. Thus, we have $T_{comm} = \frac{b^2}{B_n}$ and $T_{mem} = \frac{2b^2}{B_d}$. Suppose the processor and the FPGA each performs l_1 and l_2 operations, respectively. Besides $l_1 + l_2 = \frac{n}{bp}$, we have

$$l_1 \times T_p + T_{comm} + l_2 \times T_{mem} = l_2 \times T_f \qquad (6)$$

Hardware/Software Coordination As the FPGAs and the processors execute separate tasks in our design, there is no memory access conflict. For l_2 operations performed by the FPGA, the processor needs to give the start signal and be notified when the FPGA is done. Thus, the frequency of coordination is $\frac{2}{l_2 \times T_f} = \frac{2kF_p}{2l_2b^3}$ times per second.

6 Experimental Results

To illustrate our ideas, we implemented our designs on XD1 which is briefly discussed in Section 3. The FPGA-based designs are described using VHDL. We used Xilinx ISE 7.1i and Mentor Graphics ModelSim 5.7 development tools [12, 20]. In our experiments, we used our own 64-bit floating-point adders and multipliers that comply with IEEE-754 standard [8]. On the processor, a C program is executed. It is in charge of file operations, data transfer and MPI communications. Note that in our implementation, the C program only employs one AMD processor in each node.

6.1 Implementation Details

We first determine the values of the system parameters. When our FPGA-based matrix multiplier in [21] is implemented on one FPGA in XD1, at most 8 PEs can be configured. Hence $k = 8$. As each PE performs two floating-point operations in each clock cycle, $O_f = 16$. The clock speed of the design $F_f = 130 \times 10^6$. To obtain the sustained performance of the processor for matrix multiplication, *dgemm* subroutine in AMD Core Math Library (ACML) [1] is executed. When the matrix size is 2048, one AMD processor achieves 3.9 GFLOPS using *dgemm* routine so that $O_p \times F_p \approx 3.9 \times 10^9$. As the system we used for implementation only contains one chassis, we have $p = 6$. In XD1, $B_n = 2$ GB/s. As the FPGA-based design gets one word from the DRAM memory in each clock cycle, $B_d = 1.04$ GB/s. On each node, 8 MB of SRAM memory is allocated to store the intermediate results so that we have $\frac{b_p b}{p-1} \leq \frac{8\text{ MB}}{b_w}$. As b needs to be a multiple of both k and $p-1$, we set $b = 3000$.

Next we need to determine the workload partition. According to Equation 4, b_p = 1720 and b_f = 1280. To get the value of l, we first obtain the latencies of *opLU*, *opL* and *opU* operations on the processor. When $b = 3000$, the routine used and the latency of each operation are shown in Table 1. According to Equation 5, we set $l = 3$.

Table 1. Routines and Latencies for Various Operations in LU Decomposition

Operation	*opLU*	*opL*	*opU*
Routine	*dgetrf*	*dtrsm*	*dtrsm*
Latency (s)	4.9	7.1	7.1

We have justified our selection of b_p, b_f and l through experiments. Figure 5 shows the latency of a $b \times b$ block matrix multiplication when b_f increases from 0 to b. The computations are performed on $P_1, \ldots, P_{p-1}$, while P_0 sends data to all the other nodes. We see that when b_f increases from 0 to 1280, the latency keeps decreasing because the processor shares its workload with the FPGA. However, when b_f further increases, the FPGA becomes overloaded and the latency begins to increase. Figure 6 shows the latency of our design for performing the 0th iteration of LU decomposition when l increases from 0 to 5. In the experiment, b_f = 1280 and b_p = 2720. Operations *opLU*, *opL* and *opU* are performed on the processor of P_0, while the block matrix multiplications are performed on the other nodes. We see that latency decreases when l increases from 0 to 3, because the computing power of $P_1, \ldots, P_{p-1}$ is better utilized. The latency begins to increase when l is further increased as P_0 is under-utilized. However, the increase is not noticeable until $l = 5$.

In the design for Floyd-Warshall algorithm, p and B_n are the same as in the design for LU decomposition. We implemented the design proposed in [18] on the FPGA in XD1. At most $k = 8$ PEs can be configured and each PE performs two floating-point operations in each clock cycle. Thus, $O_f = 16$. Our implementation achieved 120 MHz, hence $F_f = 120 \times 10^6$. As the FPGA-based design gets one word from the DRAM memory in each clock cycle, $B_d = 960$ MB/s. On each node, 8 MB of SRAM memory is allocated so that we have $2b^2 \leq \frac{8\text{ MB}}{b_w}$. As b needs to be a multiple of k, we set $b = 256$. On the processor, for each $b \times b$ block, we implemented the regular Floyd-Warshall algorithm as described in [3]. When $b = 256$, the sustained performance of the algorithm is 190 MFLOPS. Thus, $O_p \times F_p \approx 190 \times 10^6$. According to Equation 6, we have $\frac{l_1}{l_2} = \frac{1}{5}$ and $l_1 + l_2 = \frac{n}{bp}$. Therefore, we set n as 18432. In this case, l_1 = 2 and l_2 = 10.

Figure 7 shows the latency of one iteration in our design for the Floyd-Warshall algorithm when the workload partition varies. We see that when l_1 decreases from 12 to 2, the latency keeps on decreasing because the FPGA keeps on taking more workload from the processor. However, when l_1 is 1, the FPGA becomes overloaded and the latency begins to increase. Unlike the LU decomposition, there is a large difference between the computing power of the FPGA and the processor. Therefore, when only the FPGA is employed, the latency is even smaller than some cases where the processor and the FPGA work together.

6.2 Performance Analysis

To evaluate our hybrid designs, we compare their performance against two baseline designs. "FPGA-only design" employs the FPGAs in the nodes only, while "Processor-only design" employs only the processors. Both of these designs employ $p = 6$ nodes in XD1. For fair comparison, the baseline designs follow the same steps as that in the hybrid design. We use GFLOPS to measure the sustained floating-point performance of the designs.

For LU decomposition, the performance of the hybrid design increases with the number of blocks, $\frac{n}{b}$, as shown in Figure 8. This is because block matrix multiplication *opMM* is the the only operation which exploits the computing power of both the FPGA and the processor. On the other hand, the performance of the design for the Floyd-Warshall algorithm almost remains the same when n increases. The reason is that the proportion between the computational loads of the FPGA and the processor is independent of the problem size. In the following discussion, for LU decomposition, $n = 30000$ and $b = 3000$; for Floyd-Warshall algorithm, $n = 92160$ and $b = 256$.

Figure 9 shows that our designs achieve 20 GFLOPS and 6.6 GFLOPS for the two applications, respectively. The performance of the hybrid design for Floyd-Warshall algorithm is low because both the designs on the processor and the FPGA are not optimized. Some optimizations exist which can achieve higher performance [11]. However, implementations of such optimizations are beyond the scope of this

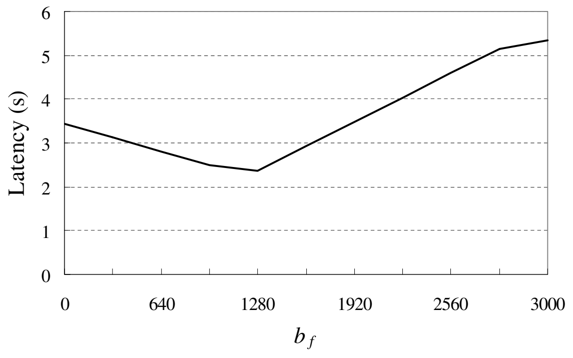

Figure 5. Latency of one $b \times b$ block matrix multiplication vs. b_f ($b = 3000$, $p = 6$)

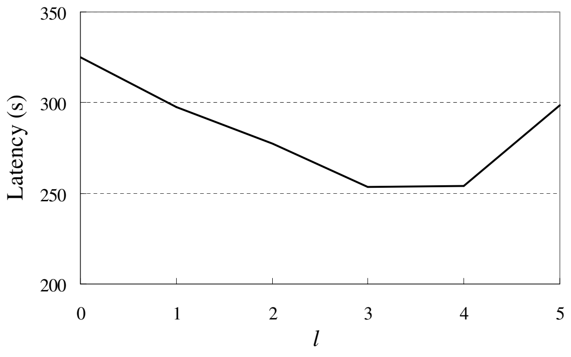

Figure 6. Latency of the 0th iteration in LU decomposition using our design vs. l ($n = 30000$, $p = 6$)

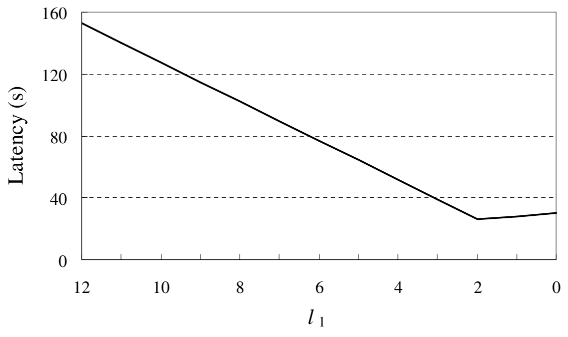

Figure 7. Latency of one iteration in Floyd-Warshall algorithm using our design vs. l_1 ($b = 256$, $n = 18432$, $p = 6$)

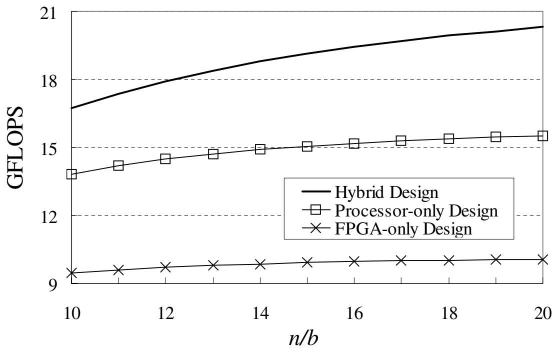

Figure 8. GFLOPS of LU decomposition vs. $\frac{n}{b}$ ($b = 3000$)

paper.

Figure 9 also compares the hybrid designs with the baseline designs. Our design for LU decomposition achieves 1.3X speedup and 2X speedup over the Processor-only design and the FPGA-only design, respectively. In addition, it achieves about 80% of the sum of the performance of the two baseline designs. For Floyd-Warshall algorithm, our design achieves 5.8X speedup and 1.15X speedup over the Processor-only design and the FPGA-only design, respectively. As the communication costs are smaller compared to the computational load in the Floyd-Warshall algorithm, our design achieves more than 95% of the sum of the performance of the baseline designs.

We also compare the performance of our designs with the performance predicted by the design model. In the prediction, we use the same system parameters and the same workload partition as in the experiments. However, we assume all the communication costs and memory transfer time are overlapped with the computations on the FPGA. Our design for LU decomposition achieves about 86% of the predicted performance. This is because in our implementation, we used the atomic ACML routines for *opLU*, *opL* and *opU*. In this case, a large part of the network communications has to be performed before the routines start. On the other hand, in our implementations for Floyd-Warshall, T_{comm} and T_{mem} are overlapped with T_f very well. Thus, our design achieves about 96% of the predicted performance. For both applications, the design model has provided a fairly accurate prediction.

7 Conclusion

In this paper, we proposed a design model for realizing hybrid designs on reconfigurable computing systems. The designs are hybrid in that they utilize both the processors and the FPGAs in these systems. In our model, a system is characterized by various parameters, including the floating-point computing power of the FPGA and the processor, the

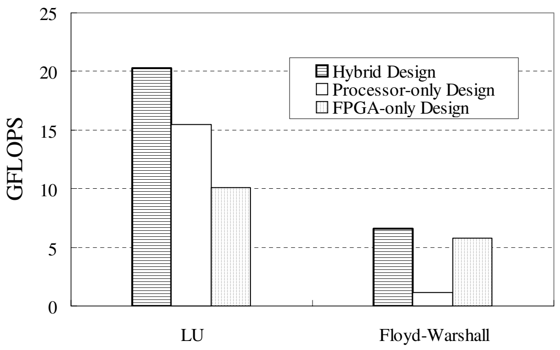

Figure 9. Performance comparison with baseline designs.

number of nodes, the memory bandwidth and the network bandwidth. We used LU decomposition and Floyd-Warshall algorithm as our example applications. Experimental results show that our designs utilize the computing power of both the processors and the FPGAs efficiently. The performance of our designs is more than 85% of the performance predicted using the design model. In the future, we plan to extend the proposed model so that it can be used for a broader range of applications.

References

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

[2] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. Brayton, and A. Sangiovanni-Vincentelli. Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proc. of the 10th International Symposium on Hardware/Software Codesign (CODES)*, Colorado, USA, May 2002.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

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

[5] DRC, The Coprocessor Company. `http://www.drccomputer.com/`.

[6] E. Anderson and J. Agron and W. Peck and J. Stevens and F. Baijot and E. Komp and R. Sass and D. Andrews. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In *Proc. of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2006.

[7] G. Venkataraman and S. Sahni and S. Mukhopadhyaya. A Blocked All-Pairs Shortest-Paths Algorithm. *Journal of Experimental Algorithmics*, 8, 2003.

[8] 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.

[9] G.R. Morris and R.D. Anderson and V.K. Prasanna. A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer. In *Proc. of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2006.

[10] 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.

[11] M. Penner and V. Prasanna. Cache-Friendly Implementations of Transitive Closure. In *Proc. of International Conference on Parallel Architectures and Compiler Techniques*, Barcelona, Spain, September 2001.

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

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

[14] R. Scrofano and M. Gokhale and F. Trouw and V.K. Prasanna. A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers. In *Proc. of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2006.

[15] Silicon Graphics, Inc. `http://www.sgi.com/`.

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

[17] U. Bondhugula and A. Devulapalli and J. Dinan and J. Fernando and P. Wyckoff and E. Stahlberg and P. Sadayappan. Hardware/Software Integration for All-Pairs Shortest-Paths on a Reconfigurable Supercomputer. In *Proc. of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2006.

[18] U. Bondhugula and A. Devulapalli and J. Fernando and Pete Wyckoff and P. Sadayappan. Parallel FPGA-based All-Pairs Shortest-Paths in a Directed Graph. In *Proc. of the 20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes, Greece, April 2006.

[19] 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.

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

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

[22] L. Zhuo and V. K. Prasanna. Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems. In *Proc. of the 12th International Conference on Parallel and Distributed Systems (ICPADS)*, Minnesota, USA, July 2006.