High Performance Linear Algebra Operations on Reconfigurable Systems [*]

Ling Zhuo and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, California, 90089-2560 USA
{lzhuo, prasanna}@usc.edu

ABSTRACT

Field-Programmable Gate Arrays (FPGAs) have become an attractive option for scientific computing. Several vendors have developed high performance reconfigurable systems which employ FPGAs for application acceleration. In this paper, we propose a BLAS (Basic Linear Algebra Subprograms) library for state-of-the-art reconfigurable systems. We study three data-intensive operations: dot product, matrix-vector multiply and dense matrix multiply. The first two operations are I/O bound, and our designs efficiently utilize the available memory bandwidth in the systems. As these operations require accumulation of sequentially delivered floating-point values, we develop a high performance reduction circuit. This circuit uses only one floating-point adder and buffers of moderate size. For matrix multiply operation, we propose a design which employs a linear array of FPGAs. This design exploits the memory hierarchy in the reconfigurable systems, and has very low memory bandwidth requirements. To illustrate our ideas, we have implemented our designs for Level 2 and Level 3 BLAS on Cray XD1.

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). 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. For example, a Xilinx Virtex-II Pro FPGA contains millions of gates, several megabytes of on-chip memory, as well as a large number of hardware primitives such as fixed-point multipliers [27]. Thus, FPGAs are now attractive for a much broader range of applications, including those requiring floating-point arithmetic. Recently, FPGAs have been employed to accelerate scientific applications, and have achieved superior performance compared with general-purpose processors [23, 26, 30].

Dramatic increases in the computing power of FPGAs have aroused strong interests in the supercomputing industry. Several vendors have developed or are developing high performance reconfigurable computing systems. Such systems include SRC MAPstation [24], Cray XD1 [5] and Starbridge Hypercomputer [25], among others. In these systems, general-purpose computing systems are combined with FPGAs which serve as a hardware application accelerator. These systems consist of multiple FPGAs and general-purpose processors that share a memory system. The given application is partitioned; the control-intensive part is executed on the processors while the computation-intensive part runs on the FPGAs. Such systems are referred to as "reconfigurable systems" in this paper.

An interesting feature of these systems is that the FPGAs have access to a large amount of memory, either through direct connection or through the switch provided in these systems. In addition, the FPGAs usually have access to multiple memory banks; the memory bandwidth available to FPGA-based computations is greatly increased. The off-chip memory provided by the systems and the on-chip memory of the FPGA fabrics together form a memory hierarchy. To develop high-performance designs for these systems, it is crucial to effectively utilize this memory hierarchy.

Despite the evolution of the device technology and rapid emergence of reconfigurable systems, corresponding FPGA-based libraries for scientific computing have been rarely studied. One of the most essential libraries for scientific computing is that for Basic Linear Algebra Subprograms (BLAS) [16]. It is known that certain operations, such as dot product, matrix-vector multiply and matrix multiply, are fundamental to most computations in scientific applications. These operations serve as basic building blocks for many numerical linear algebra applications, including the solution of linear systems of equations, linear least square problems, eigenvalue problems and singular value problems [2, 22]. Building a high performance library for these basic linear algebra operations is crucial for improving the performance of many scientific applications.

FPGA-based designs for BLAS operations have been discussed [26, 31]. However, these prior works all target a single FPGA device and a simple memory model, while reconfigurable systems consisting of more than one FPGAs have a much more complex memory model. Therefore, in this work, we propose a BLAS library which effectively exploits the memory hierarchy and the multiple FPGAs in these systems. As far as we know, this work is the

[*]This work is supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763. Equipment grants from Xilinx and HP are gratefully acknowledged.

first one to investigate a complete set of linear algebra operations on a realistic model of reconfigurable systems.

Our designs for Level 1 and Level 2 BLAS are able to achieve more than 90% of the peak performance of the FPGA device under the given memory bandwidth. A reduction circuit is proposed to accumulate sequentially delivered floating-point values. In contrast to state of the art [19, 28], this circuit uses only one floating-point adder. The buffer size needed by the circuit is $\Theta(\alpha^2)$, where α is the pipeline delay of the adder. Our design for Level 3 BLAS employs a linear array of FPGAs. It utilizes the DRAM of one general-purpose processor, the SRAM attached to the FPGAs, and the on-chip memory of the FPGAs. The performance of this design scales with the available hardware resources.

To illustrate the performance of our designs, we implemented the designs on Cray XD1. In XD1, Xilinx Virtex-II Pro XC2VP50 FPGA is used, which contains 23616 slices and about 4 Mb on-chip memory. For 64-bit floating-point matrix-vector multiply, if the data are stored in DRAM initially, the performance of our design is constrained by the bandwidth between DRAM and FPGA. Using a DRAM memory bandwidth of 1.3 GB/s, our design achieves 262 MFLOPS on one FPGA. If the data are stored in SRAM initially, our design achieves 1.05 GFLOPS. Our design for matrix multiply achieves 2.06 GFLOPS on one FPGA, using SRAM memory bandwidth of 2.1 GB/s and DRAM memory bandwidth of 24.3 MB/s. We predict that our design can achieve 12.4 GFLOPS using all the FPGAs in one chassis of XD1, and achieve 148.3 GFLOPS using 12 chassis in XD1. The required memory bandwidth increases with the number of FPGAs used. However, the bandwidth requirements in these cases are all met by the available bandwidth in XD1.

Note that XC2VP50 is not the largest available device; indeed it belongs to the previous generation of the Xilinx FPGA family. Moreover, in our experiments, we used our own IEEE-754 format double precision floating-point units. These designs are not engineered for area or speed performance. As the performance of these units improves, the performance of our design will increase accordingly. We show that if the area and speed of the floating-point units are improved and a large FPGA device (Xilinx Virtex-II Pro XC2VP100) is used, a chassis in XD1 can achieve about 50 GFLOPS. Again, the bandwidth requirements are met by XD1.

The rest of the paper is organized as follows. Section 2 introduces the structure of FPGAs and related work on BLAS operations. Section 3 presents the hardware architecture and the memory model of representative reconfigurable systems. Section 4 proposes our designs for Level 1 BLAS and Level 2 BLAS. Section 5 discusses our design for matrix multiply. Section 6 discusses implementation on Cray XD1 and the achieved and projected performance. Section 7 concludes the paper.

2. BACKGROUND & RELATED WORK

2.1 FPGAs

Field Programmable Gate Arrays (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. Recently, the number of logic blocks in an FPGA device has increased rapidly. At the same time, more hardware primitives are embedded into FPGA fabrics, including fixed-point multipliers, embedded memory blocks, etc.

Many researchers have studied the impact of increasing computing power of current FPGAs. Floating-point cores with various precision as well as various number of pipeline stages have been designed [10, 20]. In [32], floating-point sparse matrix-vector multiply has been implemented on an FPGA device. More complex applications, for example, molecular dynamics, have also been implemented on FPGAs and have been shown to achieve high performance [23].

2.2 BLAS Operations

The set of Basic Linear Algebra Subprograms, which is commonly referred to as BLAS, is used in a wide range of software, including LINPACK [6]. BLAS are building block routines for performing basic vector and matrix operations, and are divided into three levels: Level 1 BLAS perform vector-vector operations, Level 2 BLAS perform matrix-vector operations, and Level 3 BLAS perform matrix-matrix operations. Optimizations for BLAS library on general-purpose processors have been widely studied [7, 16]. These include loop unrolling to reduce loop overhead, register blocking to reduce the number of memory accesses and cache blocking to maximize cache reuse and also to reduce memory accesses.

The early work on FPGA-based linear algebra focused on fixed-point arithmetic [3, 14]. Recently, there have been some efforts in implementing floating-point linear algebra applications. In [30], 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 [26], FPGA-based implementations of operations from all levels of BLAS library are considered. The main focus of that work was to examine the potential capacity of FPGAs in performing BLAS operations. In [31], we analyzed the design tradeoffs for BLAS operations under various hardware resource constraints.

These prior works are all based on a simple computational model which consists of a single FPGA and dedicated memory, as shown in Figure 1. However, as a reconfigurable system contains multiple levels of memory and multiple FPGAs, achieving high performance on it is much more challenging. In this paper, we propose BLAS designs based on the computational model of the reconfigurable systems that is discussed in Section 3.2.

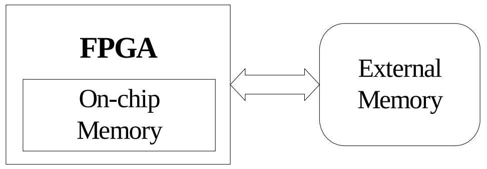

Figure 1: Simple FPGA-based Memory Model

2.3 Reduction Circuit

In many linear algebra operations, such as dot product and matrix-vector multiply, accumulation of sets of floating-point values is required. Thus, a reduction circuit is needed in designs for these op-

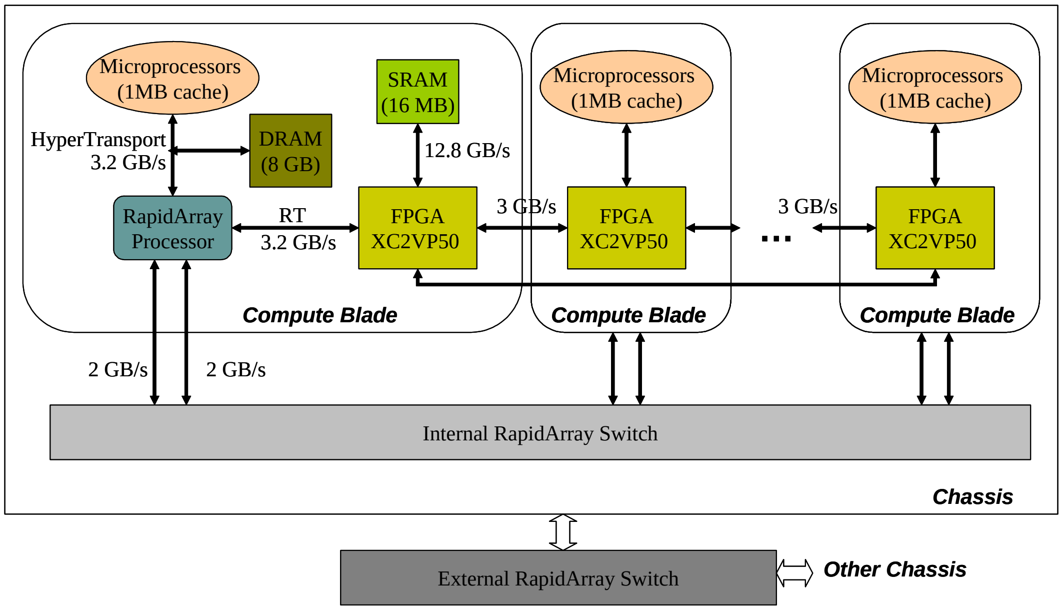

Figure 2: Hardware Architecture of Cray XD1

erations. Actually, besides linear-algebra applications, many other scientific applications need reduction circuits. For example, such accumulation is needed at the end of the Lennard-Jones computations in molecular dynamics [23].

When the adder is pipelined, sequential additions can lead to read-after-write data hazards. Simple solutions exist for this problem, such as using a single-stage but slow adder or stalling the pipeline. However, these solutions are ineffective and may greatly hurt the performance. Thus, reduction circuits for pipelined architectures have to be specially designed. Research in this area started several decades ago, when pipelined computers and vector computers first became available. Kogge proposes a method which uses $\lg(s)$ adders to reduce s inputs [15]. In [21], the authors propose a vector reduction method which uses one adder and a fixed number of buffers. The method in [21] is well suited for reducing one input vector. However, for multiple input vectors, the method has to interleave the sets; otherwise, the buffer in their design will overflow.

In order to achieve high clock speed, floating-point adders implemented on FPGAs are usually deeply pipelined. We have proposed several reduction circuits suitable for implementation on FPGAs using pipelined adders. The design in [28] uses one adder and a buffer of size $\Theta(\lg(s))$ for reducing multiple input sets, where s is the size of the largest input set. However, in this design, the size of each set must be a power of 2. For s not a power of 2, we have proposed two designs that employ two adders [19]. One design needs a buffer of size $\Theta(\lg(s))$, while the other design needs a buffer of size $\alpha(\lg(\alpha))$, where α is the number of pipeline stages in the floating-point adder. All of our designs complete the reduction of s inputs in $\Theta(s)$ time. In this paper, we propose a reduction circuit which uses *one floating-point adder only*, and can reduce multiple input sets of *arbitrary* size without stalling.

3. RECONFIGURABLE HIGH-END COMPUTING SYSTEMS

3.1 Hardware Architecture

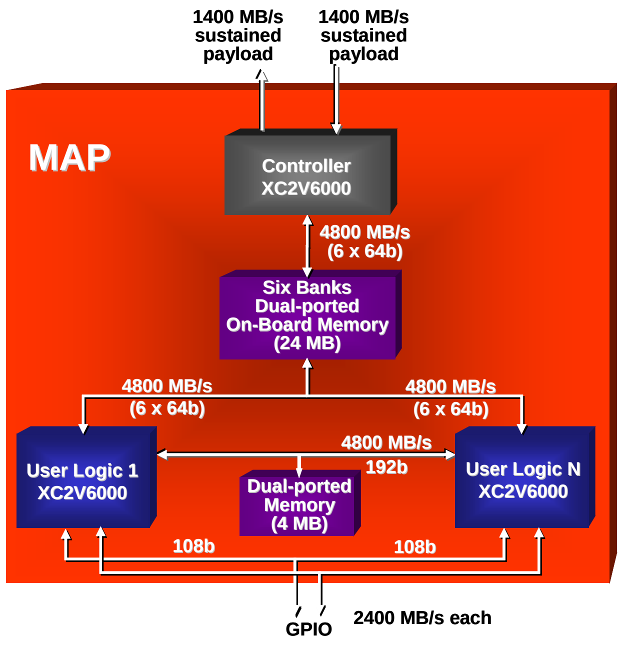

Figure 3: Hardware Architecture of SRC MAP Processor

3.1.1 *SRC MAPstation*

In SRC MAPstation, one Intel microprocessor runs on a Linux operating system, and the reconfigurable logic resource 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. Through the FPGA controller, the microprocessors and the FPGAs communicate and share their memory. The hardware architecture of a MAP processor is shown in Figure 3 [24]. SRC uses its own compilation system, Carte, to provide a unified application development environment [24].

3.1.2 *Cray XD1*

Cray XD1 [5] also uses FPGAs as hardware accelerator. 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. Through the RapidArray Processors, the FPGAs can access the DRAM of the microprocessors. Compute blades communicate through RapidArray internal switches, and six compute blades fit into one chassis. Inside a chassis, the FPGAs are connected in a circular array through the Xilinx RocketI/O Multi-Gigabit Transceivers [27]. Multiple chassis are connected using RapidArray external switches. A typical installation of XD1 contains 12 chassis. The hardware architecture of XD1 is shown in Figure 2. The "RT" between RapidArray Processor and the FPGA refers to the RapidArray Transport links designed by Cray.

3.2 Computational Model

3.2.1 *System Model*

In this section, we discuss a computational model of reconfigurable high-end computing systems by abstracting their features. In these systems, there are multiple compute nodes. Each node consists of one or more general-purpose processors and one FPGA device. The processors are attached to DRAM memory, while the FPGA is attached to SRAM memory. The processors and the FPGA share the memory within each node; the FPGA can directly access the DRAM without the cooperation of the processor.

All the nodes are connected through an interconnect network. The system can be modeled as shown in Figure 4. In the figure, P_0, . . . , P_{n-1} refer to the compute nodes. M_i is the local memory of P_i and consists of DRAM and SRAM. Compared with Figure 2, Figure 4 provides a high-level abstraction.

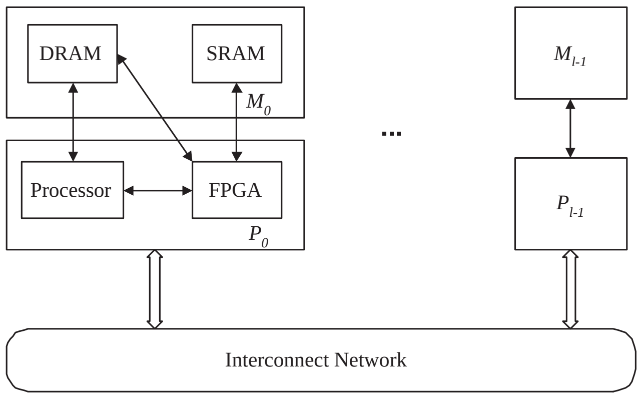

Figure 4: Computational Model

In our work, we focus on designs on FPGAs. Thus, the computations are performed on the FPGA using the local memory in each node, but not the processors.

3.2.2 *Memory Model*

Inside each node, the FPGA has access to multiple levels of memory that have various storage capacity and bandwidth. The first level is the on-chip memory of FPGAs, usually Block RAMs (BRAMs). BRAMs are embedded hardware primitives on FPGA fabrics. In a large state-of-the-art device, the aggregate memory bandwidth of BRAMs is over 100 GB/s, while the total size is usually less than 10 Mb.

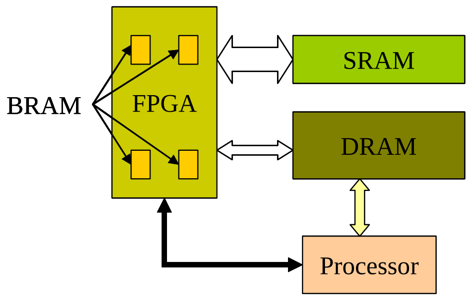

Figure 5: Memory Model for a Single Node in Reconfigurable Systems

Table 1: Characteristics of memory for a single FPGA in reconfigurable systems

	SRC		Cray	
Level	Size	Bandwidth	Size	Bandwidth
A	648 KB	260 GB/s	522 KB	209 GB/s
B	24 MB	4.8 GB/s	16 MB	12.8 GB/s
C	8 GB	1.4 GB/s	8 GB	3.2 GB/s

The second level of memory is off-chip but on-board memory, which is usually SRAM. To access this memory, the FPGA-based designs have to incorporate certain memory controllers. The storage capacity of SRAM memory is much larger than that of the on-chip memory, however its bandwidth to FPGA is much lower. An FPGA also has access to the main memory of the general purpose processor, which is usually DRAM and in the range of gigabytes. The bandwidth between DRAM and the FPGA is much less and usually is less than 5 GB/s.

These various levels of memory together form a memory hierarchy, as shown in Figure 5. The characteristics of memory at each level are summarized in Table 1. Here we refer to the on-chip memory as Level A, SRAM as Level B, and DRAM as Level C. The memory hierarchy in reconfigurable systems is quite similar to that in the general-purpose processor systems. However, the bandwidth of the first memory level in the reconfigurable systems is much higher. This permits high computational parallelism as well as high I/O parallelism in FPGA-based systems. Secondly, an application executed on a general-purpose processor usually has no control over the content in the cache. In contrast, FPGA-based designs in the reconfigurable systems initiate read and write operations to the various levels of memory, and can utilize the memory more efficiently. The third difference is that in the reconfigurable systems, the FPGAs can directly access Level C memory without going through Level B memory.

4. LEVEL 1 & 2 BLAS

We consider dot product from Level 1 BLAS and matrix-vector multiply from Level 2 BLAS. Although our designs can be applied to non-square matrices, in the following discussion, only the square matrices are considered. The operations use floating-point arithmetic.

4.1 Dot Product

In this operation, two vectors, u and v, each consisting of n elements, are given. The operation can be formulated as:

$$u \times v = \sum_{i=0}^{n-1} u_i v_i$$

In this operation, for each pair of floating-point numbers, one floating-point multiplication and one floating-point addition are performed. Since there is no data reuse in this operation, it is I/O bound. In particular, its performance is determined by the rate at which the vectors are input.

A *tree-based architecture* can be used to compute dot product. In this architecture, in each clock cycle, k multipliers accept elements from each of the vectors and initiate k multiplications. An adder tree with $k - 1$ adders is employed to sum up the outputs of the multipliers. When $k < n$, we need a reduction circuit to accumulate the outputs of the root adder in the adder tree. Design of reduction circuits is discussed in Section 4.3.

By varying the value of k, the architecture can match with the available I/O bandwidth. Moreover, the floating-point adders and multipliers are pipelined so that additions, multiplications and I/O operations can be overlapped.

4.2 Matrix-Vector Multiply

This operation multiplies an $n \times n$ matrix with a column vector, and yields another column vector. It can be formulated as:

$$y = Ax, y_i = \sum_{j=0}^{n-1} a_{ij} x_j \quad (i = 0, 1, \ldots, n-1)$$

Each element of A is used in only one floating-point multiplication. Therefore, this operation is also I/O bound. However, since each element in x is used n times, there is certain data reuse. According to the format in which matrix A is stored, we propose two architectures for this operation.

If A is stored in row-major order, matrix-vector multiply actually consists of n dot products. Thus, we use a tree-based architecture similar to that for dot product. Each multiplier is attached to a local storage, which stores part of vector x. In particular, the pth multiplier ($p = 0, 1, \ldots, k-1$) stores elements $p, k+p, \ldots, (\frac{n}{k} - 1)k+p$ of x. In each clock cycle, each multiplier reads one element of A, finds the corresponding element of x in the local storage, and multiplies the two numbers. An adder tree is used to accumulate the outputs of the multipliers. When $k < n$, a reduction circuit is needed outside the adder tree. One advantage of the tree-based architecture is that it can be easily extended for sparse matrix-vector multiply [32].

On the other hand, if A is stored in column-major order, another architecture can be used. In this architecture, there are k adders, each is connected to a multiplier. The adder is attached to a local storage, which stores the intermediate results of some elements of vector y. In particular, the pth adder ($p = 0, 1, \ldots, k-1$) stores the intermediate results of elements $p, k+p, \ldots, (\frac{n}{k} - 1)k + p$ of vector y. During each clock cycle, the k multipliers multiply k distinct elements of A with one element of x. The adders then accumulate the results of the multiplications with the intermediate results of y stored in the local storage. In this architecture, the intermediate result of y_i ($i = 0, 1, \ldots, n-1$) is only used every n clock cycles. Therefore, as long as n is larger than the number of pipeline stages in the floating-point adder, data hazards do not occur.

In both of the architectures, the size of required on-chip memory is n words. When vector x is too large to be stored on the FPGA, block matrix-vector multiply is employed. Suppose the on-chip memory of the FPGA can store at most b words. In the first architecture, matrix A is partitioned into blocks of size $b \times n$, and x is partitioned into blocks of size $1 \times b$. Each A block is read in row-major order and multiplied with the corresponding x block. In the second architecture, matrix A is partitioned into $n \times b$ blocks, and y is partitioned into $1 \times b$ blocks. Each A block is read in column-major order and multiplied with vector x to generate the corresponding y block.

Note that both dot product and matrix-vector multiply are I/O bound operations. Therefore, even if the computing power is unlimited, the performance of these operations is bounded by the available memory bandwidth. If the problem size is so small that all the source data can be stored in SRAM, the performance of the operations is constrained by the bandwidth between the FPGA and SRAM. Otherwise, if the source data are stored in DRAM, the performance is constrained by the bandwidth between the FPGA and DRAM.

4.3 Reduction Circuit

As discussed above, a reduction circuit is needed in the architectures for both dot product and matrix-vector multiply. In this paper, we propose a reduction circuit which uses only one floating-point adder to reduce multiple input sets of arbitrary size. Suppose there are p input sets. We use s_i to denote the size of the ith input set ($0 \leq i \leq p-1$), where s_i is an arbitrary positive integer. The inputs in the sets are delivered sequentially, and the reduction circuit reduces all the inputs in a set into a single value. We use α to denote the number of pipeline stages in the floating-point adder.

4.3.1 Intuitive Idea

When $s_i > \alpha$, we can reduce s_i inputs to α items using one adder, without causing read-after-write data hazards. To achieve this, we write the first α inputs into a buffer; in each of the subsequent clock cycles, one item in the buffer and the new input to the circuit are given to the adder as operands; the output of the adder is written back to the buffer. After $s_i + \alpha$ clock cycles, s_i inputs are reduced to α items.

However, due to data dependencies, reducing the resulting α items requires $\Theta(\alpha \lg(\alpha))$ clock cycles using a tree traversal. During this period, a buffer is needed to store the incoming inputs, and the buffer size increases with the number of input sets.

To reduce the buffer size, we need to utilize the adder more efficiently. Suppose α distinct sets are stored in a buffer and each set has α items. Then we can interleave the additions from the α sets so that the adder is fully utilized and no data hazard occurs. In this way, reducing α^2 items from α distinct sets at most takes α^2 clock cycles. At the same time, another buffer of size α^2 is needed to store the new inputs. Thus, we can reduce multiple inputs sets with one adder and two buffers of size α^2.

4.3.2 Architecture

Based on the idea discussed above, we propose a reduction circuit whose architecture is shown in Figure 6. It contains one floating-point adder and two buffers of size α^2. The input can enter one

Table 2: Characteristics of 64-bit Floating-Point Units and Reduction Circuit

	Adder	Multiplier	Reduction Circuit
Number of Pipeline Stages	14	11	-
Area (Slices)	892	835	1658
Clock Speed (MHz)	170	170	170

of the buffers, or the adder directly. The adder selects its operands from the input and the buffers. If the output of the adder is the final result of a set, it is written into external memory; otherwise, it is written back into one of the buffers.

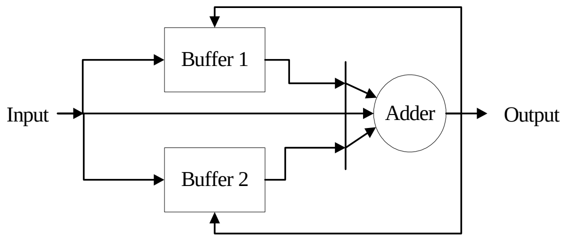

Figure 6: Architecture of reduction circuit

The buffer which accepts new inputs is denoted as *Buf*$_{in}$. For set i, if $s_i \le \alpha$, s_i inputs are written into *Buf*$_{in}$ directly; otherwise, α inputs are written into *Buf*$_{in}$ and are then added with the remaining new inputs of the set. In the s_i (if $s_i > \alpha$) or α (if $s_i \le \alpha$) cycles in which *Buf*$_{in}$ accepts new inputs, the adder is used by another buffer. This buffer is called *Buf*$_{red}$, which stores no more than α items for each set. *Buf*$_{red}$ interleaves α additions from α distinct sets. The results of these additions are written back to *Buf*$_{red}$. Buffer 1 and Buffer 2 function as *Buf*$_{in}$ and *Buf*$_{red}$ alternately. When *Buf*$_{in}$ is full, the two buffers are swapped. That is, *Buf*$_{in}$ becomes *Buf*$_{red}$ and *Buf*$_{red}$ becomes *Buf*$_{in}$.

The adder reads from *Buf*$_{red}$ only when *Buf*$_{in}$ is accepting new inputs. In such clock cycles, the adder does not read from *Buf*$_{in}$. Hence the use of the adder is collision-free. When *Buf*$_{in}$ is full and becomes *Buf*$_{red}$, each of its column contains items from α distinct sets. Therefore, when the adder reads from *Buf*$_{red}$ column by column, the adder interleaves additions from α distinct sets. Thus the circuit avoids any read-after-write hazard. Moreover, it can be shown that the buffers in our design do not overflow. This is because α^2 clock cycles are needed to fill *Buf*$_{in}$ and α^2 clock cycles are needed to empty *Buf*$_{red}$. Detailed proofs can be found in [29].

In the reduction circuit, the latency of reducing one input set depends on the sizes of subsequent sets. Therefore, we calculate the total latency of reducing p sets. We have proved that the reduction circuit reduces p sets in less than $(\sum_{i=0}^{p-1} s_i + 2\alpha^2)$ cycles [29].

4.4 Performance

In this section, we examine the performance our design for double precision floating-point dot product and matrix-vector multiply on a single FPGA. Our target device is Xilinx Virtex-II Pro XC2VP50, which is used in XD1. This device contains 23616 slices, about 4 Mb of on-chip memory and 852 I/O pins. In our experiments, we used Xilinx ISE 6.2i [27] and Mentor Graphics ModelSim 5.7 [17] development tools.

We used our own floating-point adder and multiplier. The implementation details of these units can be found in [9]. These floating-point units comply with the IEEE-754 double-precision format [12]. Their characteristics are shown in Table 2. The characteristics of the reduction circuit are also shown in Table 2. Although the reduction circuit contains only one floating-point adder, its area increases due to the control logic. Its clock speed is determined by that of the floating-point adder, and is 170 MHz.

We have implemented the tree-based architectures on the target device. If we ignore the memory bandwidth constraint, the maximum value of k is determined by the number of I/O pins for dot product; for matrix-vector multiply, k is determined by the number of slices on the device. However, in these two cases, the required memory bandwidth exceeds the available bandwidth between the FPGA and SRAM in XD1. Since the maximum bandwidth from SRAM to the FPGA is 6.4 GB/s, in our experiments, we set $k = 2$ for dot product, and $k = 4$ for matrix-vector multiply. The characteristics of the designs are shown in Table 3, which are obtained after place & route. For dot product, our design uses about 16% of the total area of the FPGA device; for matrix-vector multiply, our designs uses less than 31% of the total area.

Table 3: Characteristics of Designs for Level 1 BLAS and Level 2 BLAS

BLAS	Level 1	Level 2
No. of Multipliers, k	2	4
Area (Slices)	5210	9669
% of Total Area	22%	41%
Clock Speed (MHz)	170	170
Memory Bandwidth (GB/s)	5.5	5.6
Sustained MFLOPS	557	1355
% of Peak MFLOPS	80%	97%

We next measure the sustained MFLOPS performance of the designs. n is chosen to be 2048 so that vector x is stored in the on-chip memory and matrix A is stored in SRAM. We compare the sustained MFLOPS performance of the designs to the peak MFLOPS performance. Since both Level 1 and Level 2 BLAS are I/O bound, their peak MFLOPS is determined by the memory bandwidth between FPGA and SRAM. Suppose the memory bandwidth is bw words per second and the computing power is unlimited. For dot product, the minimum latency is $\frac{2n}{bw}$ second. Thus, the peak performance is $\frac{2n}{2n/bw} = bw$ FLOPS. For matrix-vector multiply, the minimum latency $\approx \frac{n^2}{bw}$ second. Thus, the peak performance of matrix-vector multiply is $\frac{2n^2}{n^2/bw} = 2bw$ FLOPS. Due to the latency of the reduction circuit, our design for dot product achieves 80% of the peak performance using a memory bandwidth of 5.5 GB/s, as shown in Table 3. However, the effect of reduction circuit becomes negligible for matrix-vector multiply, and our design achieves more than 95% of the peak performance.

5. LEVEL 3 BLAS

The last BLAS operation we consider in this paper is dense matrix multiply from level 3 BLAS. Suppose we have two $n \times n$ matrices, A and B. This operation computes $C = AB$, given by:

$$c_{ij} = \sum_{q=0}^{n-1} a_{iq} b_{qj} \quad (i, j = 0, 1, \ldots, n-1)$$

In contrast to the other two operations, matrix multiply has lots of data reuse. Each element of A and B is used n times. Thus, this operation is not I/O bound if appropriate architecture and algorithm are used.

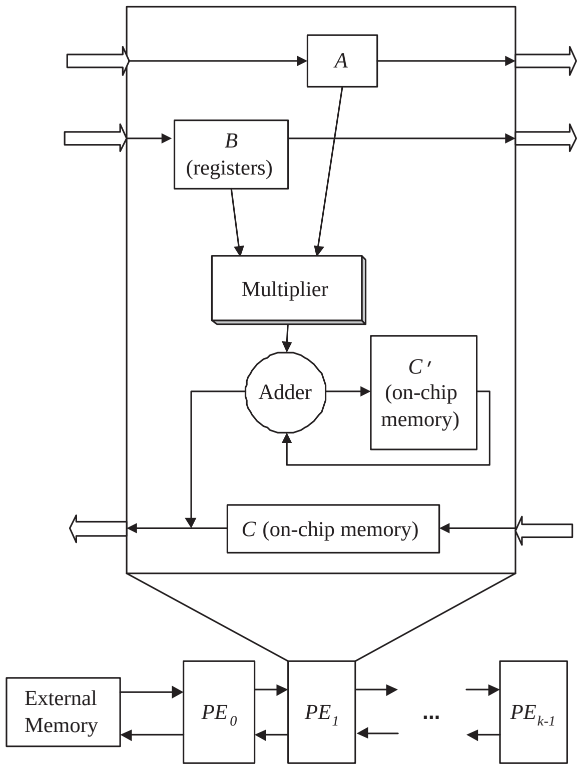

Figure 7: Architecture for Matrix Multiply on a Single Node

The key to an effective design of matrix multiply is to fully utilize the internal memory. Suppose the size of the internal memory is M. It has been proved [11] that the I/O complexity of any implementations of the "usual" matrix multiply algorithm is $\Omega(\frac{n^3}{\sqrt{M}})$, when $\Theta(1) \le M \le \Theta(n^2)$. Here the *I/O complexity* refers to the total number of words that are read from and written to the external memory by the algorithm.

In [31], we proposed an FPGA-based design for matrix multiply. This architecture achieves asymptotically optimal latency and uses minimum memory bandwidth with the given on-chip memory. However, in [31], the output of the result matrix is not fully addressed. In this paper, we first complete the architecture in [31] by enabling streaming output. In Section 5.2, we further extend our design to the memory hierarchy and multiple nodes in reconfigurable systems.

5.1 Matrix Multiply

In our design, there are k Processing Elements (PEs) connected in a linear array. The PEs are labeled from left to right, as $PE_0, PE_1, \ldots, PE_{k-1}$. PE_0 reads matrices A and B from the external memory, and transfers them along the linear array. The final elements of C are transferred in the opposite direction, and are written to the external memory by PE_0. Each PE consists of three I/O ports, one floating-point multiplier and one floating-point adder.

In this design, the internal memory is the on-chip memory of the FPGA. Suppose its size is M, and $m = \sqrt{\frac{M}{2}}$. Each PE also contains $\frac{2m}{k}$ registers, and two local storage of size $\frac{m^2}{k}$. One local storage stores intermediate results of C and is denoted as C' storage; another storage stores the final elements of C and is called C storage.

The design is shown in Figure 7. It performs block matrix multiply where the block size is $m \times m$. The blocks are denoted as A^{gz} and B^{zh}, where $g, z, h = 0, 1, \ldots, \frac{n}{m} - 1$. Without loss of generality, we assume n is a multiple of m, and m is a multiple of k.

For each block matrix multiply $A^{gz} \times B^{zh}$, A^{gz} is read in column-major order, while B^{zh} is read in row-major order. PE_p, $p = 0, \ldots, k-1$, is in charge of computing columns p, $(k+p)$, $\ldots$, $((\frac{m}{k}-1)k+p)$ of C^{gh}. The computation starts by reading the first row of B^{zh} into the architecture. As these m numbers traverse the linear array, $PE_p, p = 0, \ldots, k-1$, stores the pth, $(k+p)$th, $\ldots$, $((\frac{m}{k}-1)k+p)$th numbers into its registers. Afterwards, every $\frac{m}{k}$ clock cycles, one element of A^{gz} and one element of B^{zh} are read into the architecture. As an element of A^{gz} passes through a PE, it is multiplied with every element of B^{zh} stored in the PE whose row index matches its column index. The intermediate results for C^{gh} are stored in C' storage.

In this design, each intermediate result of C^{gh} is updated every $\frac{m^2}{k}$ clock cycles. Therefore, as long as $\frac{m^2}{k}$ is larger than α (the number of pipeline stages in the floating-point adder), no data hazard will occur.

When a final element of C is generated, it is transferred to C storage of PE_{p-1} if $p > 0$; otherwise, it is written to the external memory. Each PE generates $\frac{m^2}{k}$ final elements of C in consecutive clock cycles. When the PE is transferring these elements ($p > 0$) or writing them to the memory ($p = 0$), the elements transferred to the PE by its right neighbor are stored in C storage. Afterward, the PE starts to transfer or write the elements stored in C storage. Therefore, the size of C storage is also $\frac{m^2}{k}$.

Our design performs $A^{gz} \times B^{zh}$ in three stages. The first stage initializes the registers of all the PEs using the first row of B^{zh}. It takes $m \times \frac{m}{k} + (k-1) = \frac{m^2}{k} + (k-1)$ clock cycles. The second stage performs arithmetic computations, and the last element of C is generated by PE_{k-1} after $m^2 \times \frac{m}{k} + (k-1) + \frac{m^2}{k} + \alpha = \frac{m^3}{k} + \frac{m^2}{k} + (k-1) + \alpha$ clock cycles. In the third stage, the last element of C traverses the linear array from PE_{k-1} to PE_0. This takes at most $\frac{m^2}{k} \times (k-1)$ clock cycles. When a series of block matrix multiplies are performed consecutively, the first stage of one block multiply can be overlapped with the other two stages of the previous block multiply. Therefore, the effective latency for computing $A^{gz} \times B^{zh}$ is $\frac{m^3}{k}$.

For $n \times n$ matrix multiply, this architecture performs $(\frac{n}{m})^3$ block matrix multiplies. Thus, the effective latency of our design is $(\frac{n}{m})^3 \times$

$\frac{m^3}{k} = \frac{n^3}{k}$. The total storage size used by the design is $2m^2$. Two words from the external memory are input every $\frac{m}{k}$ cycles. The design outputs m^2 words to the external memory every $\frac{m^3}{k}$ cycles. Thus, the required memory bandwidth is $\frac{3k}{m}$ words per clock cycle. The I/O complexity of our design is $\Theta(\frac{n^3}{m})$, which is the lower bound on I/O complexity with internal memory of size $2m^2$.

5.2 Matrix Multiply on Multiple FPGAs

To perform matrix multiply in the reconfigurable systems, we can directly use the architecture in Section 5.1. Since the architecture consists of a linear array of PEs, it can be easily implemented on multiple FPGAs. However, such an implementation does not utilize the SRAM attached to the FPGAs. In this section, we present a matrix multiply design which efficiently utilizes the memory hierarchy of the reconfigurable systems discussed in Section 3.

The design in Section 5.1 employs three types of storage. The first type is the register file, which stores selected elements of B; the second is the on-chip memory of FPGA, which stores the intermediate or final results of C^{gh}; and the third type of storage is the external memory, which stores matrices A and B and also accepts the final elements of C.

Based on this storage hierarchy, we propose a design for matrix multiply which uses multiple FPGAs. In this design, the internal memory is SRAM and the on-chip memory. Suppose the total size of the SRAM available to the FPGAs is $2b^2$. We first partition the matrices A and B into $b \times b$ blocks. These blocks are denoted as A^{iq} and B^{qj}, $i, j, q = 0, 1, \ldots, \frac{n}{b} - 1$. Each A^{iq} (B^{qj}) is further partitioned into smaller blocks of size $m \times m$, which are denoted as A^{iq}_{gz} and B^{qj}_{zh}, $g, h, z = 0, 1, \ldots, \frac{b}{m} - 1$. Without loss of generality, we assume n is a multiple of b, and b is a multiple of m.

The design is shown in Figure 8. In this design, there are l FPGAs connected in a linear array. Each FPGA is attached to its own SRAM. The FPGAs are labeled from left to right, as *FPGA*$_0$, *FPGA*$_1$, $\ldots$, *FPGA*$_{l-1}$. *FPGA*$_0$ reads elements of A and B from the DRAM of the processor which is connected to *FPGA*$_0$. When the final results of C traverse the linear array and reach *FPGA*$_0$, they are written back to the same DRAM.

The organization of each FPGA in the linear array is similar to that of the PE in Figure 7. It uses one storage of size $\frac{2b}{l}$ to store elements in B^{qj}. This storage is implemented using the on-chip memory. The FPGA also contains two storage of size $\frac{b^2}{l}$ to store the intermediate results and the final results of C^{ij}, respectively. These storage are implemented using SRAM.

Each FPGA is configured to perform the matrix multiply design described in Section 5.1, which is labeled as "MM" in Figure 8. MM performs $A^{iq}_{gz} \times B^{qj}_{zh}$, and generates part of C^{ij}_{gh}. MM contains k PEs, and the local storage of the PEs in MM is implemented using the on-chip memory of the FPGA device. The total size of the on-chip memory needed by MM is $2m^2$. The result of MM is given to a floating-point adder and is combined with the intermediate result of C^{ij}_{gh}.

The algorithm is similar to that in Section 5.1, except that each element is substituted by a $m \times m$ block. For $A^{iq} \times B^{qj}$, A^{iq} is read in column-major order, and B^{qj} is read in row-major order. The first row of $m \times m$ blocks of B^{qj} is first read into the design. As

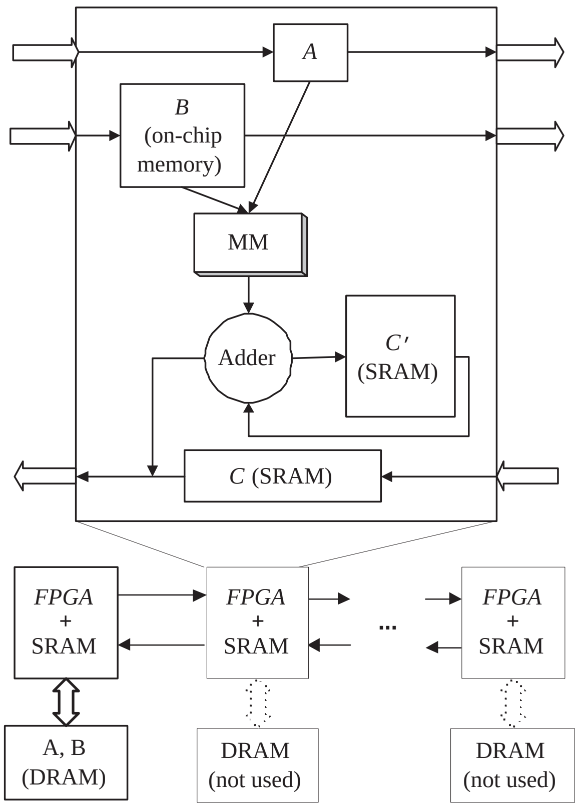

Figure 8: Matrix Multiply on Multiple Nodes

these $\frac{b}{m}$ blocks traverse the linear array of FPGAs, *FPGA*$_f$ stores blocks f, $(l + f)$, $\ldots$, $((\frac{b}{ml} - 1)l + f)$ into the on-chip memory. As A^{iq}_{gz} passes through an FPGA, it is multiplied by MM with $\frac{b}{l}$ blocks of B^{qj} stored in the FPGA.

Recall that MM employs k PEs, and reads one element of A and one element of B every $\frac{m}{k}$ cycles. Therefore, for this hierarchical design, one $m \times m$ block of A^{iq} and one block of B^{qj} are read from DRAM every $\frac{b}{ml} \times m^2 \times \frac{m}{k} = \frac{m^2 b}{kl}$ cycles.

Following the analysis in Section 5.1, the effective latency of computing $A^{gz} \times B^{zh}$ is $\frac{b^3}{kl}$. Thus, for $n \times n$ matrix multiply, the effective latency of this design is $\frac{n^3}{kl}$ cycles. The total size of SRAM used is $2b^2$. The I/O complexity of our design (the total number of words that are read from and written to DRAM) is $\Theta(\frac{n^3}{b})$. This is the lower bound on I/O complexity of $n \times n$ matrix multiply with internal memory of size $2b^2$.

5.3 Performance

We implemented our design in Section 5.1 on a Xilinx Virtex-II Pro XC2VP50 FPGA. The experimental setup is the same as in Section 4.4. Due to the constraint of on-chip memory, we set $m = 128$. Using our 64-bit floating-point adder and multiplier, a single PE occupies 2158 slices, and runs at 155 MHz after place & route. As shown in Figure 9, the area of the design increases linearly with k, the number of PEs in the design. On the other hand, as more PEs are used, more routing resources are needed and the routing becomes more complex. Thus, the achievable clock speed of the design degrades as k increases. We can configure at most 10 PEs on the above device. In this case, the maximum achievable clock speed is 125 MHz in this case. Therefore, the maximum sustained GFLOPS performance of our design on the device is $2n^3/\frac{n^3}{10} \times \frac{125}{1000}$ = 2.5 GFLOPS for 64-bit matrix multiply.

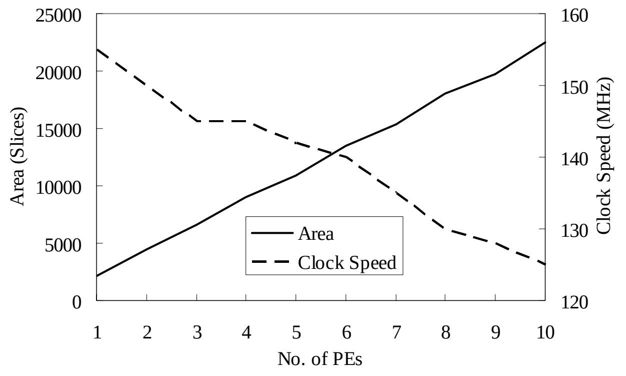

Figure 9: Area & Clock Speed of Matrix Multiply Design on a Single FPGA

6. EXPERIMENTAL RESULTS

6.1 Design Flow

To illustrate our ideas, we implemented our designs on a Cray XD1. In XD1, to utilize the FPGA accelerator, two programs are needed. One is a C program which runs on the processors, and the other is a VHDL program that describes an FPGA-based design.

Before loaded onto XD1, an FPGA-based design needs to be modified as follows: 1. SRAM memory controllers (SRAM cores) need to be inserted if the design needs to access the SRAM banks; 2. An RT (RapidArray Transport) core needs to be inserted for the design to communicate with the processor; 3. An application-specific component Rt_Client is needed to control the communications between the FPGA, the processor and the SRAM. The resulting structure of an FPGA-based design on XD1 is shown in Figure 10.

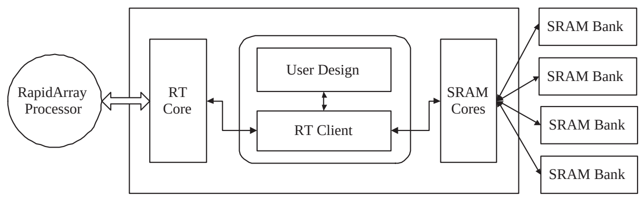

Figure 10: Block Diagram of an FPGA-based Design on Cray XD1

After modifying the design, the following steps are required to load the design onto XD1 [4]:

1. Write the C program and build the executable for it.
2. Synthesize, place & route the FPGA-based design using Xilinx ISE [27]. At this step, the design can be debugged using ModelSim of Mentor Graphics [17], a simulator for VHDL.
3. Generate the binary file for the FPGA-based design. The file is then converted to Cray-specific FPGA logic file using command-line tools on XD1.
4. Load the logic file onto the FPGA. Write a job script for the design, and submit the job to the system.

6.2 Matrix-Vector Multiply

We implemented the tree-based architecture for Level 2 BLAS in Section 4 on Cray XD1. In our experiments, one node was used. Before the computation starts, matrix A is read from the DRAM of the processor and distributed to the four SRAM banks attached to the FPGA. Next, the processor initializes the local storage of the PEs using vector x. During the computation, the design on the FPGA reads one word from each SRAM bank in one clock cycle. The processor and the FPGA communicate through several status registers about the problem size n and completion of initialization and computation.

Since the size of SRAM memory attached to an FPGA is 16 MB, n can at most be $\sqrt{2} \times 1024$. In our experiments, we set $n = 1024$. As discussed in Section 4.4, based on the available bandwidth between FPGA and SRAM, we set $k = 4$.

The additional components including the RT core, memory controllers and control logic for status registers occupy approximately 3000 slices. They also affect the clock speed of the design. The characteristics of the design are shown in Table 4. As the design reads one 64-bit word and 8-bit parity code from each SRAM bank during every clock cycle, the achieved SRAM memory bandwidth is 5.9 GB/s. The total latency for performing matrix-vector multiply is 8.0×10^{-3} seconds. The computation time is 1.6×10^{-3} seconds, and the remaining time is spent in moving the data between DRAM and SRAM. The achieved DRAM memory bandwidth is 1.3 GB/s.

Under the used DRAM memory bandwidth, the peak performance of any design for matrix-vector multiply is 325 MFLOPS (calculated as discussed in Section 4.4). The sustained performance of our design is 262 MFLOPS, which is about 80% of the peak performance. If matrix A is initially stored in SRAM, our design achieves 1.05 GFLOPS.

Table 4: Performance of Level 2 and Level 3 BLAS on a single FPGA in XD1

BLAS	Level 2	Level 3
k	4	8
Area (Slices)	13772	21029
% of Total Area	58%	89%
Clock Speed	164 MHz	130 MHz
SRAM Bandwidth	5.9 GB/s	2.1 GB/s
DRAM Bandwidth	1.3 GB/s	24.3 MB/s
Sustained Performance	262 MFLOPS	2.06 GFLOPS
% of Peak Performance	80.6%	46.6%

6.3 Matrix Multiply

We implemented one node in the design shown in Figure 8 on XD1, that is, $l = 1$. Thus, our design uses one FPGA and the processor connected to it. The elements of matrices A and B are read from DRAM of the processor, written into the on-chip memory and then read by the "MM" component in Figure 8. Two of the four SRAM banks attached to the FPGA are used for storing intermediate results of C, and is called C' storage (Section 5.1). The other two banks store the final results of C and serve as C storage. As in the matrix-vector multiply design, several status registers are employed for communication between the processor and the FPGA.

Our design on one FPGA consists of k PEs and one floating-point adder. Due to the RT core and the memory controllers, at most 8 PEs can be configured on the device, that is, $k = 8$. To simplify the implementation, we set $m = k = 8$. Under the constraint on the size of SRAM, b can be at most 1024. In our experiment we set $b = 512$.

The characteristics of the design is shown in Table 4. For $n = 512$, the total latency of matrix multiply is 131 $\times$ 10^{-3} seconds. Unlike matrix-vector multiply, only 0.7% of the total latency is for I/O operations. During most of the time, the floating-point operations are performed concurrently with the I/O operations.

Since three $m \times m$ blocks are read from or written into DRAM memory every $\frac{m^2 b}{k}$ clock cycles, the DRAM bandwidth used by the design is 48.8 MB/s. As for SRAM, one word is read from and written into C' storage during every clock cycle, and the bandwidth is 2.1 GB/s. Note that when only one FPGA is used, C storage is not needed. When multiple FPGAs are employed, one $m \times m$ block is read from and written into C storage every $\frac{m^2 b}{k}$ clock cycles. Thus, the total SRAM bandwidth required by the design is 2.1 GB/s + 32.5 MB/s = 2.1 GB/s.

For $n = 512$, the sustained performance is 2.06 GFLOPS. For $n > 512$, we set $b = 512$; that is, matrices A and B are partitioned into blocks of size 512×512. These blocks are read by the design consecutively. If the results of block multiplies are accumulated by the general-purpose processors, the sustained performance of the FPGA will not be affected.

For our design of matrix multiply, the peak performance is not determined by the available memory bandwidth, but by available hardware resources on the FPGA device. In the ideal scenario, only floating-point units are configured on the device without incurring any overheads, and each unit performs one floating-point operation during every clock cycle. Hence, the peak performance of the device is calculated as 2 $\times$ *maximum number of floating-point units that can be configured on the device* $\times$ *maximum clock speed of the units*. Using our floating-point units, the peak performance of XC2VP50 is thus 4.42 GFLOPS. Our design achieves a little less than 50% of the peak performance due to the clock speed degradation caused by the routing and the control logic. Moreover, the components for XD1, such as the RT core and the memory controllers, occupy more than 10% of the slices on the FPGA.

Note that a 2.6 GHz AMD Opteron processor with a L1 cache of 64 KB and a L2 cache of 1 MB achieves 4.1 GFLOPS. This was obtained using *dgemm* function of 64-bit AMD Core Math Library [1]. A 3.2 GHz Xeon processor-based platform with 1 MB L3 cache achieves 5.5 GFLOPS performing 64-bit matrix multiply, while a 3 GHz Pentium 4 processor with 512 KB L2 cache achieves 5.0 GFLOPS. These numbers were obtained by executing Intel Math Kernel Library [13]. Note that the math libraries for the processors employ common software optimizations as well as several optimizations specific to the processors. In contrast, our design is not optimized with respect to area or clock speed. Manual placement or other optimizations can be employed to further improve the performance of our design.

6.4 Projected Performance

6.4.1 Performance of A Single Chassis

We now examine the performance of our design of matrix multiply in Figure 8 using one chassis of XD1. In this case, l, the number of FPGAs, is 6 because one chassis of XD1 contains 6 FPGAs. Since $2b^2$ cannot be larger than the total size of SRAM (96 MB), we set $b = 2048$. Every $\frac{m^2 b}{kl}$ cycles, *FPGA*$_f$ exchanges at most three $m \times m$ blocks with DRAM if $f = 0$, or with *FPGA*$_{f-1}$ if $f > 0$. For $k = m = 8$, the required DRAM bandwidth and the interconnect bandwidth between two adjacent FPGAs equals 73.1 MB/s. This bandwidth is much smaller than the available DRAM bandwidth and the interconnection bandwidth among FPGAs in XD1. On the other hand, employing multiple FPGAs does increase the latency because each element needs to traverse more PEs. The increase in the latency is $k \times l = 48$ clock cycles. It is negligible compared with the total latency. The sustained performance of a chassis $\approx$ 2.06 $\times$ 6 = 12.4 GFLOPS.

Most of the PE area is occupied by the floating-point adder and floating-point multiplier. Thus, the performance of our design depends largely on that of the employed floating-point units. The implementation of these units has no effect on the architecture or the control logic of the design. Therefore, when improved floating-point units are available, they can be plugged into our design easily. Moreover, if the performance of these units is improved, the performance of the design will improve accordingly [30]. The PE used in our experiments occupies more than 2000 slices and runs at 155 MHz. Therefore, we project the performance of our design when the area of PE ranges from 1600 to 2000 slices, and the clock speed increases from 160 MHz to 200 MHz.

Figure 11 shows the projected sustained performance of one chassis, as a function of the area and the clock speed of the PE. We calculate the GFLOPS performance using equation: 2 $\times$ *number of PEs on the device* $\times$ *clock speed of the PE* $\times$ 6. Also, 25% of the performance is deducted to account for the degradation of the clock speed caused by the routing. When the PE occupies 1600 slices and runs at 200 MHz, one chassis can achieve more than 27 GFLOPS. As more PEs are configured and the clock speed of the PEs increases, the memory bandwidth requirement also increases. If we set $b = 2048$, and $k = m$, *with the smallest and fastest PE, the required SRAM bandwidth of our design is 2.5 GB/s, and the required DRAM bandwidth is 147.7 MB/s.* These requirements are met by the available memory bandwidth in XD1.

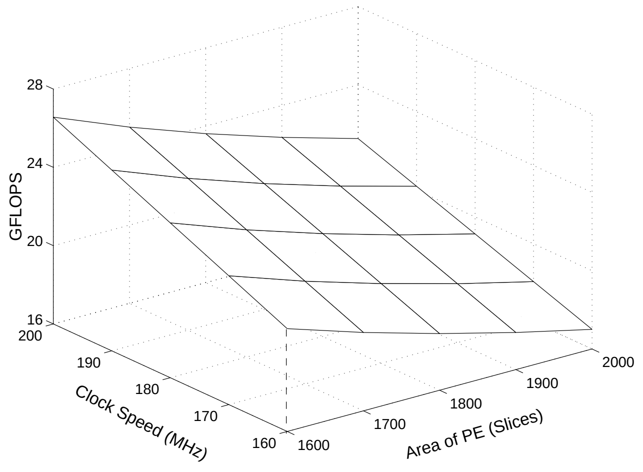

Figure 11: Projected Sustained Performance of Matrix Multiply Design Using a Chassis in XD1

The performance of our design also depends on the device used.

XC2VP50 is a relatively small device in the Xilinx Virtex-II Pro family. With a larger device, more PEs can be configured and higher performance can be achieved. Figure 12 shows the projected sustained performance of our design using a chassis if Xilinx Virtex-II Pro XC2VP100 were to be used in XD1. XC2VP100 contains 44096 slices, about 8 Mb of on-chip memory and 1164 I/O pins. As this device contains about twice as many slices as XC2VP50, its performance is also about twice as that of XC2VP50. With the smallest and fastest PE, the required SRAM bandwidth and DRAM bandwidth are 2.7 GB/s and 284.8 MB/s respectively. These requirements are met by the available memory bandwidth in XD1.

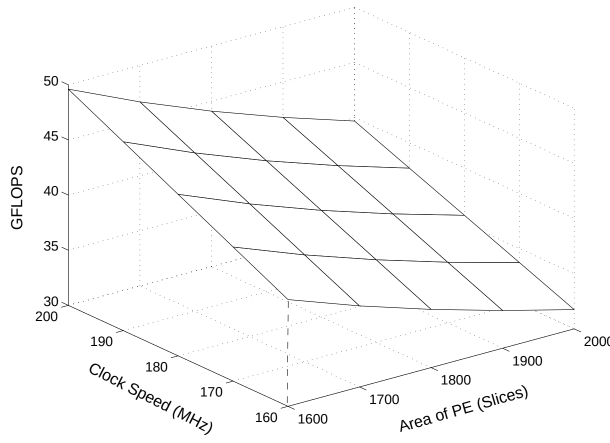

Figure 12: Projected Sustained Performance of Matrix Multiply Design Using a Chassis, With Xilinx Virtex-II Pro XC2VP100

6.4.2 *Performance of Multiple Chassis*

We also predict the performance of our design using all the chassis in Cray XD1. In a typical installation, there are 12 chassis in XD1. They communicate through a crossbar-switch fabric that connects the RapidArray processors. The link between two chassis has a bandwidth of 4 GB/s.

When 12 chassis are employed, the total number of FPGAs $l = 6 \times 12 = 72$. With the current FPGA device in XD1 and the PE in our implementation, $k = 8$. As each element of the input matrices has to traverse all the PEs, the latency is increased by $k \times l = 576$ clock cycles. For large n, this increased latency is negligible compared with the total latency.

The SRAM bandwidth required by the design is 3.0 GB/s, and the required DRAM bandwidth is 877.5 MB/s. These requirements are met by the available bandwidth in XD1. The required interconnection bandwidth between two chassis is the same as the required DRAM bandwidth, and is also met by the interconnection bandwidth in XD1. Thus, using 12 chassis, our design for matrix multiply can achieve $2.06 \times 6 \times 12 = 148.3$ GFLOPS.

7. CONCLUDING REMARKS

In this paper, we analyzed the memory model of reconfigurable high-end computing systems, which employ FPGAs as hardware accelerator. We proposed an FPGA-based BLAS library for such systems. Our designs can employ multiple FPGAs and effectively utilize the memory hierarchy in these systems. Three operations, dot product, matrix-vector multiply and matrix multiply, were discussed. To illustrate our ideas, the designs were implemented on Cray XD1. For 64-bit matrix multiply, our design achieved a sustained performance of 2.06 GFLOPS. We projected that on a 12-chassis installation of XD1, 148.3 GFLOPS can be achieved by our design. In our experiments, we used our own floating-point units. If the performance of these units is improved, the performance of our designs will improve accordingly. We also proposed a compact but efficient reduction circuit. This circuit is able to accumulate multiple sets of floating-point values without stalling the pipeline in the floating-point adder. The circuit uses one adder and two buffers of size α^2, where α is the number of pipeline stages in the adder.

Besides the work presented in this paper, we have proposed FPGA-based designs for other linear algebra applications. For example, we have proposed a design for Sparse Matrix-Vector Multiply (SpMXV)[32], which employs the tree-based architecture similar to the one discussed in Section 4. This design makes no assumption on the sparsity of the matrix, and accepts matrices in Compressed Row Storage format. As the design exploits both computational parallelism and I/O parallelism of the FPGA, it achieves high performance for floating-point SpMXV.

Based on the SpMXV design, we have proposed an FPGA-based design for floating-point Jacobi iterative solver [18]. Jacobi is a basic iterative method, and is usually used as preconditioners for the more efficient methods like conjugate gradient (CG) [22]. Our design is parameterized, deeply pipelined and highly parallelized. For matrices having irregular structure, our design achieves a large speedup over highly optimized software implementations on general-purpose processor systems.

In the future, we plan to extend our existing designs and investigate more linear algebra applications for implementations on reconfigurable systems. Besides effectively utilizing the memory hierarchy and multiple FPGAs in these systems, we plan to exploit the computing power of the general-purpose processors that are connected to the FPGAs.

Acknowledgement

The authors thank Dr. Geert C. Wenes from Cray Inc. for providing technical documentations and assistance with understanding the XD1 architecture. Thanks are also due to Aditya Kothadiya for useful feedback on earlier versions of this manuscript.

8. REFERENCES

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

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Ed.*. SIAM, 1994.

[3] D. Benyamin, W. Luk, and J. Villasenor. Optimizing FPGA-based Vector Product Designs. In *Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM'99)*, pages 188–197, April 1999.

[4] Cray Inc. *Cray XD1 FPGA Development*.

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

[6] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Soc. for Industrial and Applied Math., 1979.

[7] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16(1):1–17, Mar. 1990.

[8] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev. 64-bit Floating-Point FPGA Matrix Multiplication. In *Proc. 13th Int'l Symp. Field Programmable Gate Arrays (FPGA'05)*, February 2005.

[9] 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. Int'l Conf. Eng. Reconfigurable Systems and Algorithms (ERSA'05)*, June 2005.

[10] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna. Analysis of High-Performance Floating-Point Arithmetic on FPGAs. In *Proc. 11th Reconfigurable Architectures Workshop*, April 2004.

[11] J. Hong and H. Kung. I/O Complexity: The Red Blue Pebble Game. In *Proc. ACM Symp. Theory of Computing*, pages 326–333, May 1981.

[12] Inst. of Electrical and Electronics Engineers. *IEEE 754 Standard for Binary Floating-Point Arithmetic*. IEEE, 1984.

[13] Intel Corp. `http://www.intel.com`.

[14] J. W. Jang, S. Choi, and V. K. Prasanna. Area and Time Efficient Implementation of Matrix Multiplication on FPGAs. In *Proc. 1st IEEE Int'l Conf. Field Programmable Technology (FPT'02)*, December 2002.

[15] P. M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Pub. Corp., 1981.

[16] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979.

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

[18] G. R. Morris and V. K. Prasanna. An FPGA-Based Floating-Point Jacobi Iterative Solver. submitted to 8th Int'l Symp. Parallel Architectures, Algorithms, and Networks, 2005.

[19] G. R. Morris, L. Zhuo, and V. K. Prasanna. High-Performance FPGA-Based General Reduction Methods. In *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM'05)*, April 2005.

[20] Nallatech. `http://www.nallatech.com`.

[21] L. M. Ni and K. Hwang. Vector Reduction Methods for Arithmetic Pipelines. In *Proc. 6th Int'l Symp. Computer Arithmetic*, pages 144–150, June 1983.

[22] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge Univ. Press, 1992.

[23] R. Scrofano and V. K. Prasanna. Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware. In *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms (ERSA'04)*, pages 284–290, June 2004.

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

[25] Starbridge Hypercomputers. `http://www.starbridgesystems.com/products/hardware.html`.

[26] K. D. Underwood and K. S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM'04)*, April 2004.

[27] Xilinx Inc. `http://www.xilinx.com`.

[28] L. Zhuo, G. R. Morris, and V. K. Prasanna. Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores. In *Proc. 12th Reconfigurable Architectures Workshop*, April 2005.

[29] L. Zhuo and V. K. Prasanna. High-Performance and Area-Efficient Reduction Circuits on FPGAs. unpublished report.

[30] L. Zhuo and V. K. Prasanna. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. In *Proc. 18th Int'l Parallel & Distributed Processing Symp. (IPDPS'04)*, New Mexico, USA, April 2004.

[31] L. Zhuo and V. K. Prasanna. Design Tradeoffs for BLAS Operations on Reconfigurable Hardware. In *Proc. 34th Int'l Conf. Parallel Processing (ICPP'05)*, Oslo, Norway, June 2005.

[32] L. Zhuo and V. K. Prasanna. Sparse Matrix-Vector Multiplication on FPGAs. In *Proc. 13th ACM Int'l Symp. Field-Programmable Gate Arrays (FPGA'05)*, California, USA, February 2005.