

Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs *

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

Abstract

The abundant hardware resources on current FPGAs provide new opportunities to improve the performance of hardware implementations of scientific computations. In this paper, we propose two FPGA-based algorithms for floating-point matrix multiplication, a fundamental kernel in a number of scientific applications. We analyze the design tradeoffs in implementing this kernel on FPGAs. Our algorithms employ a linear array architecture with a small control logic. This architecture effectively utilizes the hardware resources on the entire FPGA and reduces the routing complexity. The processing elements (PEs) used in our algorithms are modular so that floating-point units can be easily embedded into them. In our designs, the floating-point units are optimized to maximize the number of PEs integrated on the FPGA as well as the clock speed. Experimental results show that our algorithms achieve high clock speeds and provide good scalability. Our algorithms achieve superior sustained floating-point performance compared with existing FPGA-based implementations and state-of-the-art processors.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have been an attractive option for implementing computationally intensive applications [5, 6]. However, as floating-point implementations require far more hardware resources than earlier FPGAs could provide, very few previous algorithms and implementations for FPGAs employed floating-point arithmetic. Recent improvements in the computing power of FPGAs have motivated researchers to consider floating-point

based applications. Algorithms and implementations for floating-point arithmetic on FPGAs have been developed [14, 2]. Some studies have used FPGAs to accelerate scientific computations. For example, [16] considers molecular dynamics on FPGAs.

Floating-point matrix multiplication is a basic operation in many scientific computing applications. Its implementations on FPGAs can achieve high performance. First, current FPGAs provide a large number of multiplexers and embedded fixed-point multipliers. These hardware primitives can be leveraged to build high-speed floating-point adders/multipliers. Secondly, a single FPGA device now contains enough configurable logic blocks (CLBs) to hold multiple floating-point units that can perform computations concurrently. Thus, algorithms that exploit the inherent parallelism of matrix multiplication can be implemented on the device. On the other hand, FPGAs also pose new challenges for designing parallel algorithms for floating-point matrix multiplication. The resource constraints on FPGAs, such as the size of on-chip memory, result in multiple algorithm design tradeoffs. Also, the routing complexity of the algorithm must be considered. To the best of our knowledge, none of these challenges or the resulting tradeoffs have been addressed in prior works on FPGA-based floating-point matrix multiplication.

In this paper, we analyze the design tradeoffs and propose two algorithms for FPGA-based floating-point matrix multiplication. Our algorithms employ a linear array architecture in which multiple floating-point operations are performed concurrently, effectively utilizing the available resources on FPGA. By allowing communication between neighboring PEs only, our algorithms make use of the short routing wires on FPGA and are able to achieve high clock speed. When implemented on a single FPGA device, our algorithms can achieve performance that compares favorably with that of state-of-the-art processors. Moreover, the linear array architecture with a small

*Supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763 and an equipment grant from Hewlett-Packard Company.

control logic enables our algorithms to scale over multiple FPGAs. Our algorithms consist of modular PEs with embedded floating-point adders/multipliers. Any improvement in the area or speed performance of the floating-point units results in near-linear improvement in the performance of the PEs. We implement single-precision as well as double-precision IEEE-754 compliant floating-point adders and multipliers. Their clock speeds are increased through pipelining. However, excessive pipelining occupies too much area. Hence we select floating-point units that achieve the highest “throughput per unit area” for our algorithms.

We implemented our algorithms using Xilinx ISE 5.2i [17], with Xilinx Virtex-II Pro XC2VP125 as our target device. In both the proposed algorithms, the control logic for the overall architecture consumes less than 5% of the total area. The number of PEs on the device increases linearly as the available resources on the device increase. Typically, as more PEs are configured on the device, the achievable clock speed decreases due to increased routing complexity. However, in our algorithms, the clock speed degradation is less than 15% when the number of PEs increases from 2 to 24. For 32-bit matrix multiplication, our algorithms achieve 26.6 GFLOPS. For 64-bit matrix multiplication, they achieve 8.3 GFLOPS using a single Xilinx Virtex-II Pro XC2VP125.

The following sections are organized as follows. Section 2 discusses related work. Section 3 analyzes the tradeoffs in implementing floating-point matrix multiplication on FPGAs. Section 4 describes the proposed algorithms and Section 5 discusses the implementation of our algorithms on FPGA. Section 6 presents experimental results. Section 7 concludes the paper.

2 Related Work

There has been extensive work on systolic matrix multiplication algorithms. One of the classic matrix multiplication algorithms is Cannon’s algorithm [3]. It is designed based on a two-dimensional processor grid, with each processor holding a large consecutive block of data of the matrices. Amira et. al implemented a similar bit-level algorithm for 8-bit fixed-point numbers on FPGAs [1]. The disadvantage of this architecture is that it requires an I/O bandwidth that is proportional to the problem size. This feature limits the scalability of the algorithm. Li and Pan [13] parallelized a well-known sequential algorithm on a linear array of processors with a reconfigurable-pipelined bus system. Such a bus is not available on FPGAs, hence this algorithm is unsuitable for FPGAs. Jang et. al [12] proposed FPGA-based linear array algorithms for fixed-point matrix multiplication. Their algorithms require fixed I/O bandwidth, and the amount of storage within each PE is the same as the

problem size. In our algorithms, the amount of storage in each PE is independent of the problem size.

Floating-point matrix multiplication on FPGAs has been discussed in several works. In [14], the authors investigated the influence of the floating-point MACs (Multiplier and Accumulators) on the performance of a matrix multiplication algorithm. The work in [15] studied the implementation of floating-point arithmetic and used matrix multiplication as an example application. The main focus of our work is the optimization of the matrix multiplication algorithms. We also optimize the design of floating-point units in the context of these algorithms.

3 Design Tradeoffs in Using FPGAs

Recent FPGAs, such as the Xilinx Virtex-II Pro [17], provide a large number of embedded multipliers as well as fast carry chains. These primitives greatly facilitate the implementation of floating-point multipliers. Similarly, the MUXCY and MUXF multiplexer primitives on FPGAs provides building blocks for the large multiplexers needed in floating-point adders. The abundant registers on the FPGA fabric can be utilized for pipelining to improve the clock speed of the floating-point units. For matrix multiplication algorithms, the on-chip memory of FPGAs can serve as storage for intermediate results. Despite these advantages, FPGAs also pose new challenges for floating-point matrix multiplication. The limited number of the resources, such as the number of slices, the size of on-chip memory and the I/O pin count, impose multiple constraints on algorithm design. These constraints, as well as the inherent characteristics of matrix multiplication, result in multiple design tradeoffs.

3.1 Lower Bound on Latency of Matrix Multiplication

We first discuss lower bound on the latency of matrix multiplication, irrespective of whether the elements are floating-point numbers or fixed-point numbers. Also, the following discussion is device independent. Therefore, we assume unlimited number of PEs can be implemented on a device. Each PE contains one MAC.

The latency L of a matrix multiplication algorithm is determined by its computation time L_1 (cycles for performing computation) and its I/O time L_2 (cycles for I/O). $L \geq \max(L_1, L_2)$. For $n \times n$ matrix multiplication, if p PEs perform computations in parallel in every cycle, $L_1 = \Omega(n^3/p)$. L_2 depends on the available I/O bandwidth. If the I/O bandwidth is fixed, then L_2 equals the I/O complexity of matrix multiplication. I/O complexity is the total number of I/O operations performed by the algorithm. It has been shown that the I/O complexity of any algorithm performing

the “usual” matrix multiplication is $\Omega(n^3/\sqrt{M})$, where M is the size of available on-chip memory [9]. This statement is true for $O(1) \leq M \leq O(n^2)$; when $M > O(n^2)$, the I/O complexity remains $\Omega(n^2)$.

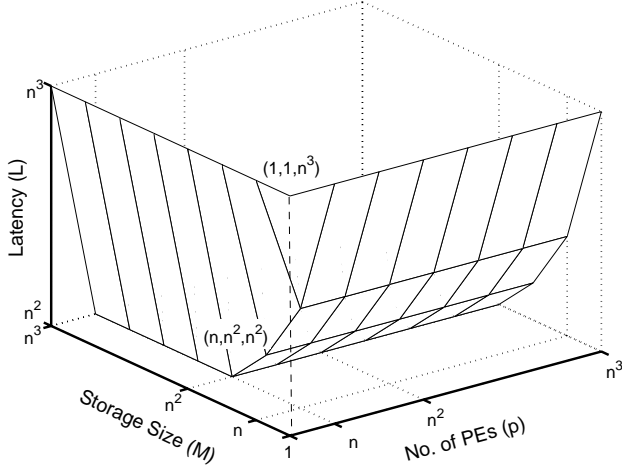


Figure 1. Tradeoffs for matrix multiplication

Figure 1 illustrates the relationship between M , p , and L . If $p = O(1)$, L is $\Omega(n^3)$ regardless of the value of M because the computation time L_1 forms the bottleneck of the algorithm. This is the case for general-purpose processors. For example, consider a processor that has a cache large enough to hold all input matrices and is able to access data from the cache in each cycle. In such a case, $M = O(n^2)$ and L_2 is reduced to $\Omega(n^2)$. However, the processor can only perform a constant number of computations during each cycle. Therefore L_1 and hence L remains $\Omega(n^3)$. On the other hand, if $M = O(1)$, L once again cannot be improved, irrespective of the value of p . In this situation, the I/O bandwidth, rather than the computing power, becomes the bottleneck of the algorithm. Suppose we have a design in which $p = O(n)$ and $M = O(n^2)$. Therefore, both L_1 and L_2 are $\Omega(n^2)$. Thus L is $\Omega(n^2)$, which is the lower bound for any matrix multiplication algorithm given that the I/O bandwidth to external memory is fixed.

3.2 Design Tradeoffs for Floating-Point Matrix Multiplication on FPGAs

Based on the discussion in Section 3.1, we know that a matrix multiplication algorithm can achieve the optimal latency if it has $M = O(n^2)$ and performs $O(n)$ computations during every clock cycle. This leads to the first tradeoff in our analysis, the tradeoff between the number of PEs employed by an algorithm and the storage size within each PE. Suppose there are n PEs. To achieve the optimal latency, the storage size in each PE must be $O(n)$. When

the matrix elements are floating-point numbers, the size of the on-chip memory on the FPGA device can constrain the performance of the algorithm. We solve this problem by making the storage size in each PE a variable, s , which is independent of the problem size. One of our algorithms, introduced in Section 4, employs n^2/s PEs and each PE has a storage of size $O(s)$. Therefore, the algorithm can maintain $M = O(n^2/s \times s) = O(n^2)$ and achieve the optimal latency. In the algorithm, n PEs perform computations during each clock cycle, while the other PEs wait for data.

The second tradeoff of the floating-point matrix multiplication algorithm design is between its I/O bandwidth and latency. We know that the larger is the I/O bandwidth of the algorithm, the shorter is the I/O time L_2 . When L_2 is reduced, latency L can also be reduced if more PEs are employed. On the other hand, an algorithm requires larger I/O bandwidth needs more I/O pins of the FPGA device. When the FPGA device fails to provide enough I/O pins for the algorithm, advanced I/O technology such as Rocket I/O Multi-Gigabit Transceivers [17] can be adopted.

Another design tradeoff exists between the area occupied by an algorithm and its clock speed. The algorithm consists of multiple PEs performing computation concurrently. To maximize the performance of the algorithm, we can maximize the number of PEs implemented on it. Therefore, it is desirable to minimize the area of each PE. On the other hand, the clock speed of the floating-point units, and hence that of the PEs, can be increased through pipelining. However, the pipeline stages need extra slices and increase the area of the PEs and decrease the number of PEs on the FPGA. Because of this design tradeoff, deep pipelining of the floating-point units may be counter productive. Thus, the floating-point units used in our algorithms are designed in the context of matrix multiplication to maximize the “throughput per unit area” metric. This is discussed in Section 5.1.

So far, the analysis on the design of matrix multiplication algorithms has ignored the on-chip communication cost as well as the routing complexity of the algorithms. In order to achieve the optimal latency, the algorithm should ensure that the on-chip communication time does not dominate the computation time L_1 . In addition, the routing of the algorithm should be implementable using available FPGA routing resources. Using long routing wires on FPGA can result in long delays and reduce the achievable clock speed. For floating-point applications on FPGAs, the routing complexity is especially an important design consideration. This is because the floating-point units occupy large numbers of slices and short routing wires on FPGA. We employ the linear array architecture, where only communication between neighboring PEs is allowed. This architecture makes good use of the short connection wires on the FPGA.

4 Matrix Multiplication Algorithms

In this section, we present two algorithms for matrix multiplication. Consider the computation of $C = A \times B$, where A , B and C are all $n \times n$ matrices. Each element of the matrices is a floating-point word. Our algorithms employ the linear array architecture. The first PE in the array is connected to the external memory, which stores the source matrices and the product matrix. Each PE consists of one or more floating-point MACs and storage for s intermediate results. We use M to denote the total amount of storage in all PEs. Both algorithms achieve the optimal latency under the given I/O bandwidth and storage size.

4.1 Algorithm 1

We employ a linear array of $\lceil n * n/s \rceil$ PEs. Each PE contains one MAC and s ($1 \leq s \leq n$) words of storage. Each PE has two input ports for A and B , and one output port for C . The PEs are numbered from left to right. The j^{th} PE receives the input data from the $(j-1)^{\text{th}}$ PE, and passes them to the $(j+1)^{\text{th}}$ PE. The final elements of C are transferred from right to left. The I/O bandwidth is 3 words per clock cycle.

```

For all PEj (j=1, ..., n⌈n/s⌉) do in parallel
  For 1 to n cycles, do
    shifts data in RR.B right to PEj+1
  For n+1 to n2+n do
    PEj shifts data in RR.A, RR.B right
    to PEj+1
    If (RR.B=bk,j mod s), copy it into RR.B1 or
    RR.B2 (alternately)
    If (RR.A=aik), cij' <= cij' + aik × bkj
    (bkj is in either RR.B1 or RR.B2)
    (i = j/s+1, j/s+s+1, ..., j/s+(⌈n/s⌉-1)s +1)

```

Figure 2. Operations of PEs in Algorithm 1

The operations of the PEs are shown in Figure 2. In this algorithm, each PE calculates s elements of C . PE_j computes

$$c_{([j/n]),((j-1) \bmod s+1)}, c_{([j/n]+[n/s]),((j-1) \bmod s+1)}, \dots, c_{(n-([n/s]-[j/n]),((j-1) \bmod s+1))}.$$

Matrices A and B are fed to PE_1 in column-major and row-major order respectively. B starts n cycles earlier than A . We group n cycles into a phase. In phase 0, the first row of matrix B is fed into PE_1 . In phase k , column k of matrix A ($a_{ik}, 1 \leq i \leq n$) and row $k+1$ of matrix B ($b_{(k+1),j}, 1 \leq j \leq n$) enter PE_1 in order. As $b_{k,j}$ traverses the PEs, it is picked up by PEs computing $c_{ij}, 1 \leq i \leq n$. These PEs keep $b_{k,j}$ in their local registers until $a_{ik}, 1 \leq i \leq n$, passes through. When a_{ik} reaches the PEs computing $c_{ij}, 1 \leq j \leq n$, these PEs update $c'_{ij} = c'_{ij} + a_{ik} \times b_{k,j}$. $c'_{ij}, 1 \leq i \leq n, 1 \leq j \leq n$, is the intermediate result of c_{ij} ,

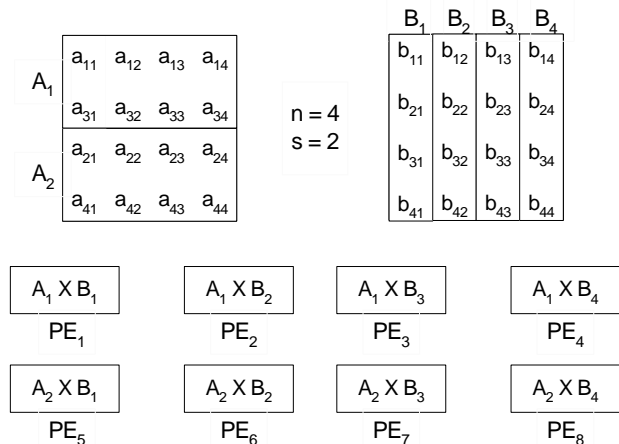


Figure 3. Computation illustrations for Algorithm 1 ($n = 4, s = 2$)

$1 \leq i \leq n, 1 \leq j \leq n$, and is stored in the local storage in the PE computing c_{ij} .

Figure 3 illustrates Algorithm 1 for $n = 4$ and $s = 2$. There are 8 PEs. PE_1 calculates c_{11} and c_{31} , while PE_5 calculates c_{21} and c_{41} . In phase 0, PE_1 stores b_{11} in its register; in phase 1, PE_1 updates $c'_{11} = c'_{11} + a_{11} \times b_{11}$, $c'_{31} = c'_{31} + a_{31} \times b_{11}$. b_{11} reaches PE_5 in phase 1 and is stored in the register. In phase 2, when column 1 of A reaches PE_5 , PE_5 updates $c'_{21} = c'_{21} + a_{21} \times b_{11}$ and $c'_{41} = c'_{41} + a_{41} \times b_{11}$.

The last element, a_{nm} , arrives at the rightmost PE at the $(n + (n-1)n + n + n\lceil n/s \rceil - 1)^{\text{th}}$ cycle. Suppose the computation related to a_{nm} completes in k cycles. k equals the total number of pipeline stages in the floating-point adder and multiplier. The final result of c_{nn} is generated at the $(n^2 + n\lceil n/s \rceil + n - 1 + k)^{\text{th}}$ cycle. It then traverses the PEs and is written to the external memory at $(n^2 + 2n\lceil n/s \rceil + n + k - 1)^{\text{th}}$ cycle. Therefore, the latency of the algorithm $L = O((1 + 2/s)n^2)$. As $1 \leq s \leq n$, we have $1 < 1 + 2/s \leq 3$, and thus $L = O(n^2)$. In Algorithm 1, the size of storage M is $s \times n\lceil n/s \rceil = O(n^2)$ words. According to Section 3, Algorithm 1 achieves the asymptotically optimal latency for the given resources.

4.2 Algorithm 2

We propose an alternate algorithm whose I/O bandwidth is $3r$ words per clock cycle, $1 \leq r \leq n$. Its latency is approximately $1/r$ of that of Algorithm 1. Algorithm 2 achieves the lower bound discussed in Section 3.1. Throughout this section we assume n is a multiple of r .

Algorithm 2 with $r = 2$ is shown in Figure 4. This algorithm consists of a linear array of $\lceil n^2/r^2s \rceil$ PEs. Each

PE contains r^2 MACs and r^2 storage blocks of s words, $1 \leq s \leq n/r$, $1 \leq r \leq n$. Each PE has $2r$ input ports and r output ports. Ports $IOA_1, IOA_2, \dots, IOA_r$ are used to input r elements of A concurrently to the PEs. Ports $IOB_1, IOB_2, \dots, IOB_r$ are used to input r elements of B . Ports $IOC_1, IOC_2, \dots, IOC_r$ output r elements of C . As in Algorithm 1, the j^{th} PE receives the input data from the $(j-1)^{th}$ PE and passes them to the $(j+1)^{th}$ PE.

In Algorithm 2, $A^i(B^i)$ contains columns(rows) $(i-1) \times n/r + 1, \dots, (i-1) \times n/r + n/r$ of $A(B)$, $1 \leq i \leq r-1$. C is partitioned into r^2 submatrices of size $n/r \times n/r$. We use c_{ij}^{xy} , $1 \leq i, j \leq n/r$, to denote the elements of C^{xy} . PE_j computes $c_{([j/n], ((j-1) \bmod n/r + 1))}^{xy}, c_{([j/n] + [n/s], ((j-1) \bmod n/r + 1))}^{xy}, \dots, c_{(n - ([n/s] - [j/n]), ((j-1) \bmod n/r + 1))}^{xy}$, $1 \leq x, y \leq r$. The local storage in PE_j is used to store intermediate results for these elements.

```

/* Matrix B1 contains rows 1, ..., n/2 of B.
Matrix B2 contains rows n/2+1, ..., n of B.
Matrix B1 and B2 enters the third and fourth I/O ports
of PE1 in row major order

Matrix A1 contains columns 1, ..., n/2 of A.
Matrix A2 contains columns n/2+1, ..., n of A
Matrix A1 and A2 enters the first and second I/O ports
of PE1 in column major order n/2 cycles behind B1 and
B2.*/

For all PEj (j=1, ..., n⌈n/s⌉/4) do in parallel
  For t=1 to n/2 do
    PEj shifts words in RR1.B and RR2.B to PEj+1
    If (RR1.B=b1k,j mod s), copy it into RR1.B1
    If (RR2.B=b2k,j mod s), copy it into RR2.B1
  For t=n/2+1 to (n⌈n/s⌉/2)+n/2 do
    PEj shifts words in RR1.A, RR2.A, RR1.B, RR2.B
    to the right (PEj+1)
    If (RR1.B=b1k,j), copy it into RR1.B1 or
    RR1.B2 (alternately)
    If (RR2.B=b2k,j), copy it into RR2.B1 or
    RR2.B2 (alternately)
    If (RR1.A=a1ik), c11ij' <= c11ij' + a1ik × b1kj,
    c12ij' <= c12ij' + a1ik × b2kj
    (b1kj is in either RR1.B1 or RR1.B2)
    (i = j/s+1, j/s+s+1, ..., j/s+(⌈n/s⌉-1)s + 1)
    If (RR2.A=a2ik), c21ij' <= c21ij' + a2ik × b1kj,
    c22ij' <= c22ij' + a2ik × b2kj
    (b2kj is in either RR2.B1 or RR2.B2)
    (i = j/s+1, j/s+s+1, ..., j/s+(⌈n/s⌉-1)s + 1)

```

Figure 4. Operations of PEs in Algorithm 2

As in Algorithm 1, matrix A and B are fed to PE_1 in column-major and row-major order respectively. B starts n/r cycles earlier than A . Each cycle r elements of A and r elements of B are fed to PE_1 . During each cycle the PE in charge of calculating c_{ij}^{xy} performs $c_{ij}^{xy'} = c_{ij}^{xy'} + A_{ik}^x \times B_{kj}^y$, $1 \leq x, y \leq r$. The elements of A_{ik}^x can be shared among the r MACs that compute c_{ij}^{xy} , $1 \leq y \leq r$. Similarly, the elements of B_{kj}^y can be shared among the r MACs that compute c_{ij}^{xy} ($1 \leq x \leq r$). This sharing allows r^2 MACs to perform computation concurrently with $2r$ input ports.

Figure 5 illustrates the data partitioning of Algorithm 2 for $n = 4$, $r = 2$, $s = 1$. In Algorithm 2, the computation of c_{nn} is completed at the $(n + (n-1)n/r + \lceil n^2/r^2 \rceil - 1 + k)^{th}$ cycle. The matrix multiplication is completed in $n^2/r + 2\lceil n^2/r^2 \rceil + n - n/r - k - 1$ cycles. Therefore, $L = O((1/r + 2/r^2s)n^2)$. As $1 \leq r \leq n$ and $1 \leq s \leq n/r$, $L = O(n^2/r)$. The size of storage M is $s \times r^2 \times \lceil n^2/rs \rceil = O(n^2)$ words. According to Section 3, Algorithm 2 achieves the asymptotically optimal latency for the resources used.

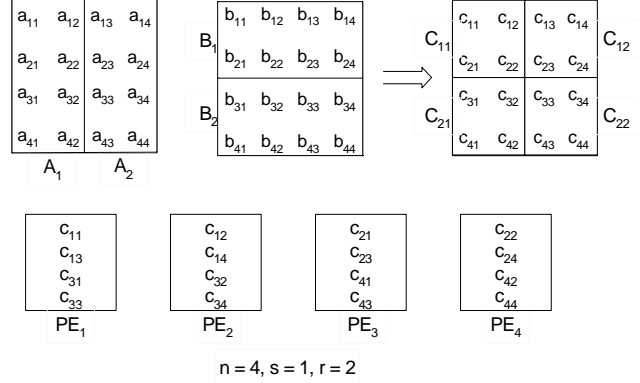


Figure 5. Computation illustrations of Algorithm 2

From the discussion in this section, we know that the latencies of our algorithms depend on the value of variable s . We now investigate how to determine s based on the problem size, the available hardware resources on the FPGA and the number of available FPGAs. First we assume only one FPGA device is available. If Algorithm 1 is used, let l be the largest number of PEs that can be configured on a device. As the floating-point units occupy large area, l is constrained by the total number of slices on the device for floating-point matrix multiplication. For problem size $n \leq l$, we implement n PEs on the device with $s = n$ in each PE. If $n > l$, we use l PEs with $s = l$ in each PE, and perform block matrix multiplication. Next we assume the number of available FPGAs is unlimited and we can connect multiple FPGAs in a linear array. Let m be the total amount of the on-chip memory on a single FPGA device. If l PEs are implemented on the device, each PE can at most have a storage of m/l . If $l < n \leq m/l$, we use n PEs with $s = n$ in each PE. If $n \geq m/l$, we use n PEs with $s = m/l$. In this case, block matrix multiplication is not needed.

5 Implementation on FPGAs

In this section, we discuss the implementation of our proposed algorithms on FPGAs. We first introduce our

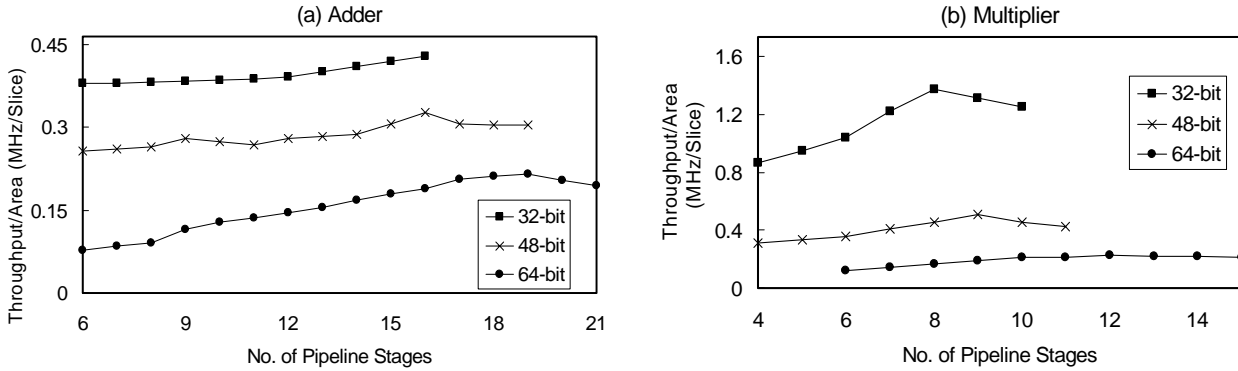


Figure 6. Throughput/Area vs. Number of pipeline stages for floating-point adders/multipliers

Table 1. Performance of 32-bit, 48-bit, 64-bit floating-Point units

Precision	32-bit		48-bit		64-bit	
	adder	multiplier	adder	multiplier	adder	multiplier
No. of Pipeline stages	16	7	16	9	21	12
Area(slices)	510	180	703	423	910	900
Clock Speed(MHz)	220	220	230	215	220	205

floating-point units, and then discuss some implementation considerations for our algorithms.

5.1 Optimization of Floating-Point Units for Matrix Multiplication

We designed deeply pipelined floating-point adders and multipliers on FPGAs, for 32-bit, 48-bit and 64-bit operands. These units comply with the IEEE 754 standard [10]. Moderate pipelining can exploit the unused flip-flops in the slices of the FPGA fabric with a slight increase in area. However, pipelining beyond a certain point results in diminishing returns; that is, the increase of area offsets the gain in clock speed. Hence we use “throughput achieved per unit area” as our metric, which indicates whether the implementation is balanced between clock speed and area. Figure 6 shows the speed-area performance in implementing the floating-point units. These units were implemented on Xilinx Virtex-II Pro XC2VP125.

We also use the “throughput per unit area” metric to select appropriate floating-point MACs for matrix multiplication. For example, for 32-bit matrix multiplication, we do not simply choose the MAC with the highest clock speed. Instead, we compare the “throughput per unit area” of all possible combinations of 32-bit adders and multipliers, and select the MAC that strikes the best balance be-

tween clock speed and area. Table 1 gives the performance of the floating-point units used in our designs. Details of the floating-point unit optimization can be found in [8].

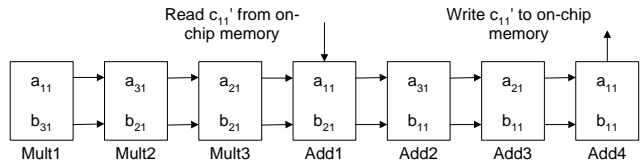


Figure 7. Potential data hazard in Algorithm 1

When pipelined adders/multipliers are employed in our designs, we need to be careful to avoid data hazards. For example, in Algorithm 1, each PE updates c'_{ij} , an intermediate result, every n cycles. Consider a floating-point adder with 4 pipeline stages and a floating-point multiplier with 3 pipeline stages. If the problem size n is 3, c'_{ij} will be read from the PE’s storage before it is written back to the storage by the previous addition. This problem is depicted in Figure 7, where Mult_{*i*}/Add_{*i*} denotes the i^{th} pipeline stage of the floating-point multiplier/adder. At a certain cycle, c'_{11} is being written to and read from the local storage of a PE. In this case, the data read out can be invalid. To avoid this problem, we have to ensure that the problem size $n \geq$ the

number of pipeline stages in the adder. In this example, if $n \geq 4$, there will be no data hazard. For Algorithm 2, the data hazard in the example can be avoided if $n/r \geq$ the number of pipeline stages in the adder.

5.2 Matrix Multiplication Implementation

In our algorithms, the storage inside the PEs is implemented using the on-chip memory of FPGAs. There are two types of on-chip memory on state-of-the-art FPGAs. One is Distributed SRAM, which is distributed over the FPGA fabric and takes up slices. Another type is Block RAM (BRAM). Each BRAM is a dedicated on-chip block of 18Kb true dual-port RAM and occupies very few slices. To save area, we use BRAM as the storage within the PEs.

Another consideration in the implementation of our algorithms on FPGAs is configuration overhead. To use our algorithms for various problem sizes, one solution is to reconfigure the FPGA for each problem size. This solution is impractical for two reasons. First, the total size of the configuration files will be too large. For example, if the problem size ranges from 1 to 80, we need to store 80 different configuration files. Since the configuration file of Virtex-II Pro FPGA is of size 5 MB, we need $80 \times 5 = 400$ MB of configuration memory outside the device. Second, the time needed for configuration may be too long compared to the computation time. A full reconfiguration of the Virtex-II Pro FPGA takes about 100 msec. For problem size 100 and device clock rate of 150 MHz, Algorithm 1 can be completed in around 1 msec.

Our algorithms handle various problem sizes in a flexible way. In our algorithms, the control logic maintain various counters. The values in the counters are used to control the PEs. For example, in the implementation of Algorithm 1, when the value of a certain counter is set to n , the accumulation in each PE occurs every n cycles for a particular entry c'_{ij} . For problem size $p \leq n$, we still use the same configuration file for problem size n . However, by setting the value of the counter to p , the accumulation of c'_{ij} now occurs every p cycles. Full FPGA configuration is only required once for the maximum problem size n that the design will handle. For other problem sizes, only the values of the counters are changed. However, in our approach, if the problem size $p \leq n$, some PEs are not used. It must be noted that if the precision of the source matrices is changed, a new FPGA configuration is necessary.

6 Experimental Results

Our target device is Xilinx Virtex-II Pro XC2VP125, which is one of the largest FPGA devices available. We used the Xilinx ISE 5.2i and Mentor Graphics ModelSim

5.7 development tools. In Section 6.1, we examine the scalability and modularity of our algorithms. In Section 6.2, we present their sustained GFLOPS performance, and compare them with some existing works. We also compare the implementations of our algorithms with that generated by Celoxica Handel-C environment DK1, a system-level design tool. In all the experiments, we used Algorithm 2 with $r = 2$.

6.1 Performance Analysis

Table 2 gives the characteristics of the MAC and PEs for 64-bit matrix multiplication. One MAC consists of one floating-point adder and one floating-point multiplier. The PE in Algorithm 1 contains 1 MAC and storage of s words; the PE in Algorithm 2 contains 4 MACs and storage of $4s$ words. As BRAMs are used for storage within the PEs, the area of a PE increases slightly as s increases. Inside each PE, some slices are used as control and communication overheads. Figure 8(a) shows that when the MAC area is reduced, the overheads inside the PE remains constant. Figure 8(b) shows that as the achievable clock speed of the MAC increases, the clock speed of the PE increases almost linearly. However, as the PE in Algorithm 2 requires more routing, it can achieve at most 180 MHz. Figure 8 shows that any area or speed improvement of the MACs results in performance improvement of the PEs. This modularity simplifies adapting the design for various precisions. For example, to design PEs for 32-bit matrix multiplication, we only need to replace the 64-bit MACs with 32-bit MACs and change the bit-width of the signals inside the PEs.

Table 2. Characteristics of MAC and PEs

	s	Area (slices)	Speed (MHz)	No. of BRAMs
MAC (64-bit)	-	1810	200	-
PE	25	2240	200	4
Alg1	1024	2254	200	8
PE	4	8024	180	16
Alg2	1024	8248	180	32

Table 3 presents the features of our implementations on XC2VP125 for 64-bit matrix multiplication. XC2VP125 can hold up to 24 PEs in Algorithm 1 or 6 PEs in Algorithm 2. Besides the slices occupied by the PEs, additional slices are used for control logic and communication between adjacent PEs in the entire architecture. The area of the control logic is less than 5% of the total area in each of the algorithms. We also investigated the scalability of our algorithms. In Figure 9, the results are shown for Algorithm

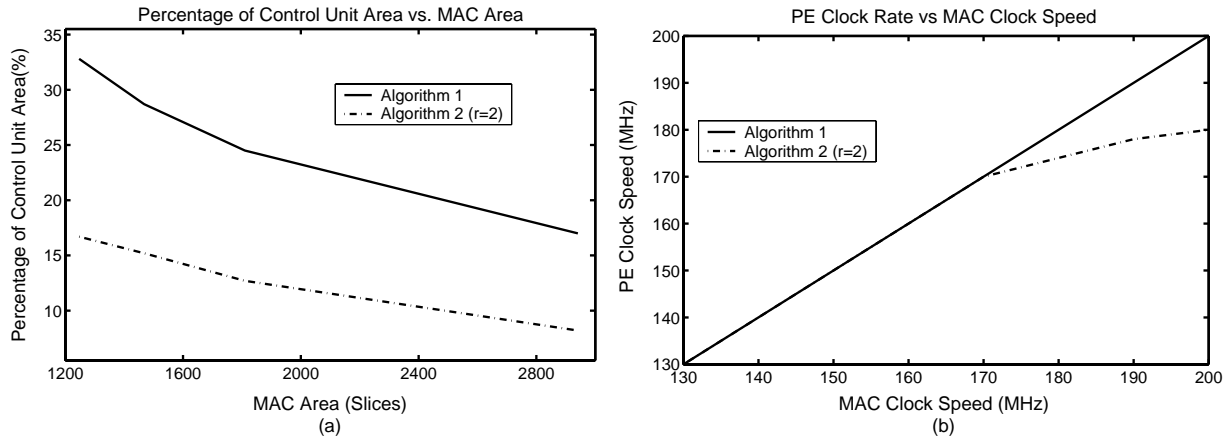


Figure 8. Area and clock speed performance of individual PEs

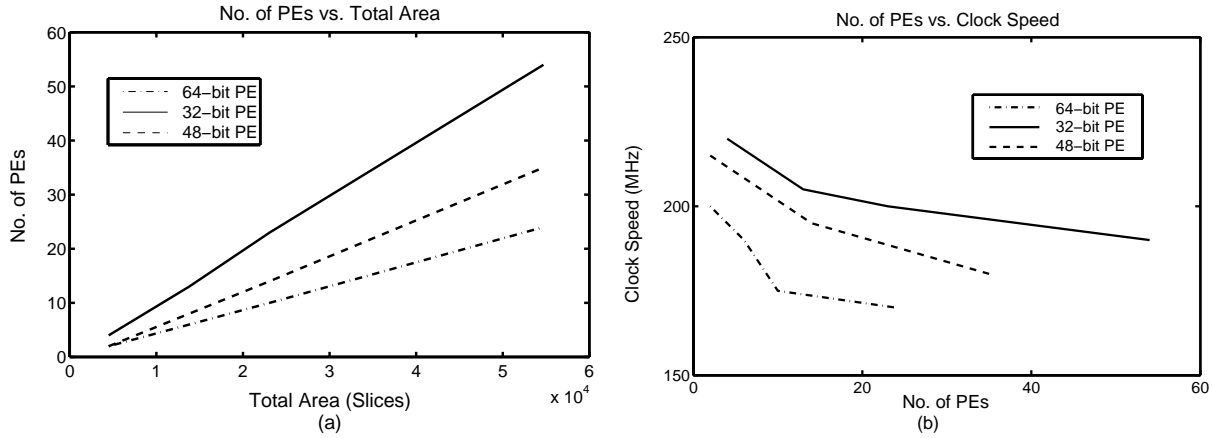


Figure 9. Area and clock speed performance of overall architecture

Table 3. Implementation Features

	No. of PEs	Area (slices)	Speed (MHz)	No. of BRAMs
Alg1	2	4506	200	16
	24	54548	172	224
Alg2	6	50158	125	128

1. Figure 9(a) shows that the number of PEs that can be configured on a device increases linearly as the available resources increase. Figure 9(b) shows that as the number PEs increases from 2 to 24, the degradation in achievable clock speed is less than 15%. The scalability of Algorithm 2 is similar and is omitted due to space limitation. Although the

routing is complex in Algorithm 2, the degradation in clock speed is no more than 30%. Our algorithms can also be implemented on multiple FPGA based designs. Algorithm 1 needs 3 matrix elements during each clock cycle. For 64-bit precision, Algorithm 1 operating at 200 MHz requires an I/O bandwidth of 38.4 Gb/s. If we connect multiple FPGAs in a linear array, the required communication bandwidth between adjacent FPGAs is 38.4 Gb/s. As XC2VP125 is able to support a pipeline throughput of 120 Gb/s at the clock speed of 200 MHz, multiple FPGAs can be connected. Algorithm 2 operating at 120 MHz requires an I/O bandwidth of 46.1 Gb/s. It can also be implemented over multiple FPGAs.

6.2 Performance Comparison

We use GFLOPS (one billion floating-point operations per second) to measure the performance of our algorithms. We first estimate the peak performance of our target device, and compare the sustained performance of the algorithms against it. Ideally, each MAC can perform one floating-point addition and one floating-point multiplication every clock cycle. Hence, the peak performance of an FPGA device can be calculated as

$$2 \times \text{number of the MACs on the FPGA} \times \text{throughput of MAC} = 2 \times \text{area of FPGA} \times (\text{throughput of MAC} / \text{area of MAC}).$$

We notice that the last factor in the above equation is actually the “throughput per unit area” metric that we used to select the MACs in Section 5. According to this equation, the peak performance of XC2VP125 is 12.3 GFLOPS for 64-bit matrix multiplication. It must be noted that this peak performance depends on the performance of the floating-point units and ignores the resource constraints. Our intent is to obtain a simple bound on the performance of the device.

The sustained performance of Algorithm 1 on a single chip (XC2VP125) is calculated as $2n^3/(\text{latency of the algorithm} \times \text{clock period})$, where n is the problem size. In our experiments, one XC2VP125 can hold at most 24 PEs using double-precision arithmetic. If $n > 24$, we partition the input matrices into 24×24 blocks and perform block matrix multiplication. In this case, the overall latency increases; however, the GFLOPS performance of the algorithm remains the same. In Algorithm 1, $n\lceil n/s \rceil + n + k - 1$ cycles are used for the input matrices to fill the pipeline within the architecture, where k is the total number of pipeline stages in the adder and multiplier used. Also, $n\lceil n/s \rceil$ cycles are used for the product matrix to traverse the linear array of PEs. When blocks are fed into the architecture consecutively, these $2n\lceil n/s \rceil + n + k - 1$ cycles of most blocks will overlap with the computations for other blocks. Therefore, the effective latency of Algorithm 1 is $(n/24)^3 \times 24^2 = n^3/24$, and the sustained GFLOPS is $2 \times 24 \times (\text{clock speed})$. Similar analysis applies to Algorithm 2. In our experiments of Algorithm 2 with $r = 2$, we observed that at most 6 PEs can fit in a single chip. Therefore, if $n > 12$, block matrix multiplication is performed. The sustained GFLOPS of Algorithm 2 is $2 \times 12 \times r \times (\text{clock speed})$.

Experiments show that Algorithm 1 and Algorithm 2 on a single XC2VP125 can achieve sustained performance of 8.3 GFLOPS and 6.0 GFLOPS respectively for 64-bit matrix multiplication. Algorithm 1 achieves more than 60% of the peak performance of the device. The performance of our algorithms compares favorably with that of state-of-the-art general purpose processors. For example, for 64-bit

matrix multiplication, a 3.2 GHz Xeon with 1 MB L3 cache achieves 5.5 GFLOPS, and a 3 GHz Pentium 4 with 512 KB L2 cache achieves 5.0 GFLOPS [11]. The performance of the Pentium processors was obtained from the performance summary of Intel Math Kernel Library. This library has been optimized to exploit the architectural features of the Intel processors [11].

Table 4. Performance comparison (32-bit arithmetic)

	Alg1	Alg2	Clemson [15]	NCU [14]
GFLOPS	26.6	21.7	1.74	1.32

Table 4 compares the performance of our algorithms with that of previous works on FPGA-based floating-point matrix multiplication [15, 14]. Their designs were implemented on Xilinx 4000 series. Because of this device, only 2 to 4 PEs could be placed on a single device, with each PE containing one floating-point MAC, and the clock speed of these designs was less than 50 MHz. In our comparison, the performance of these designs is scaled to match modern devices. Based on the performance data reported in [15, 14], we calculate the GFLOPS values of the algorithms if 24 PEs are employed and the designs are run at 200 MHz. Our algorithms achieve superior performance because they effectively reuse the input data, as well as perform more parallel floating-point operations.

Table 5. Comparison with Handel-C-based design, 8×8 64-bit matrix multiplication

	Area (slices)	Clock Speed (MHz)	Latency (us)
Handel-C	22059	12.6	130.56
Alg1	19045	182	0.35
Alg2	33589	135	0.23

We also compare our designs against the implementation compiled by the Celoxica Handel-C development tool [4]. Handel-C is a high-level language based on ANSI-C. For users designing algorithms for FPGAs, Handel-C is more convenient to use than VHDL, since a large part of hardware design and optimization is handled by the tool. We wrote a Handel-C floating-point matrix multiplication program and compared its performance against our algorithms. In the Handel-C code, we used the floating-point library provided by Celoxica [4]. We also used embedded multipliers for implementing floating-point multipliers, and further optimized the program using inner loop parallelization. The Handel-C codes were compiled by Celoxica DK1 Design Suite V1.1.

Table 5 presents the area and speed comparison between our algorithms and Handel-C based design, for 8×8 64-bit matrix multiplication. We can see that the Handel-C based design occupies more slices than our algorithms, while its latency is much longer. Table 5 also shows that although the clock speed of Algorithm 2 is lower than that of Algorithm 1, Algorithm 2 can still achieve shorter latency.

7 Conclusion

We have used floating-point matrix multiplication as an example to illustrate the capability of FPGAs in floating-point applications. We analyzed the design tradeoffs in using FPGAs for floating-point matrix multiplication. Two algorithms were proposed. The I/O bandwidth required by the algorithms is independent of the problem size. The design of the PEs is modular. Floating-point units can be easily embedded into the PEs without degrading the overall PE performance. Any area or speed improvement of these units results in linear performance improvement in our algorithms. With the increase in the available hardware resources on the device, the number of PEs that can be configured increases linearly, while the achievable clock speed decreases moderately. Our algorithms achieve superior floating-point performance over state-of-the-art processors and FPGA-based implementations. Our future work includes designing FPGA-based algorithms for other linear algebra computations, for dense as well as sparse matrices. Our preliminary work on matrix decomposition can be found in [7].

Acknowledgement

The authors thank Seonil Choi, Bo Hong, and Harish Krishnaswamy for valuable discussions and feedback on earlier versions of this manuscript. Thanks are also due to Gokul Govindu for his assistance with the experiments.

References

- [1] A. Amira and F. Bensaali. An FPGA based Parametrisable System for Matrix Product Implementation. In *Proc. of The IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS2002)*, pages 75–79, California, USA, October 2002.
- [2] P. Belanovic and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. In *Proc. of 12th International Conference on Field Programmable Logic and Application*, Montpellier, France, September 2002.
- [3] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.
- [4] Celoxica Company. <http://www.celoxica.com/>.
- [5] S. Choi, R. Scrofano, V. K. Prasanna, and J. W. Jang. Energy-Efficient Signal Processing Using FPGAs. In *Proc. of 11th ACM International Symposium on Field-Programmable Gate Arrays (FPGA 2003)*, California, USA, February 2003.
- [6] O. D. Fidanci, H. Diab, T. El-Ghazawi, K. Gaj, and N. Alexandridis. Implementation trade-offs of Triple DES in the SRC-6e Reconfigurable Computing Environment. In *Proc. of 2002 MAPLD International Conference*, Maryland, USA, September 2002.
- [7] G. Govindu, S. Choi, and V. Prasanna. A high-performance and energy efficient architecture for floating-point based LU decomposition on FPGAs. In *To be published in the Proc. of the 11th Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.
- [8] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *To be published in the Proc. of the 11th Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.
- [9] J. Hong and H. Kung. I/O Complexity: The Red Blue Pebble Game. In *Proc. of ACM Symposium on Theory of Computing*, pages 326–333, Wisconsin, USA, May 1981.
- [10] Institute of Electrical and Electronics Engineers. *IEEE 754 Standard for Binary Floating-Point Arithmetic*. 1984.
- [11] Intel. <http://www.intel.com>.
- [12] J. W. Jang, S. Choi, and V. K. Prasanna. Area and Time Efficient Implementation of Matrix Multiplication on FPGAs. In *Proc. of The First IEEE International Conference on Field Programmable Technology*, California, USA, December 2002.
- [13] K. Q. Li and V. Y. Pan. Parallel Matrix Multiplication on a Linear Array with a Reconfigurable Pipelined Bus System. *IEEE Transactions on Computers*, 50(2):519–525, 2001.
- [14] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A re-evaluation of the practicality of floating point operations on FPGAs. In *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, California, USA, April 1998.
- [15] I. Sahin, C. S. Gloster, and C. Doss. Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems. In *Presented at 3rd Military and Aerospace Programmable Logic Devices (MAPLD) Conference*, Maryland, USA, September 2000.
- [16] C. Wolinski, F. Trouw, and M. Gokhale. A Preliminary Study of Molecular Dynamics on Reconfigurable Computers. In *Proc. of The International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'03)*, Nevada, USA, June 2003.
- [17] Xilinx Incorporated. <http://www.xilinx.com>.