

Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems

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 reconfigurable computing systems provide new opportunities for high-performance parallel implementations of scientific computations. In this paper, we study designs for floating-point matrix multiplication, a fundamental kernel in a number of scientific applications, on reconfigurable computing systems. We first analyze design tradeoffs in implementing this kernel. These tradeoffs are caused by the inherent parallelism of matrix multiplication and the resource constraints, including the number of configurable slices, the size of on-chip memory, and the available memory bandwidth. We propose three parameterized algorithms which can be tuned according to the problem size and the available hardware resources. Our algorithms employ a linear array architecture with simple control logic. This architecture effectively utilizes the available resources, and reduces routing complexity. The Processing Elements (PEs) used in our algorithms are modular so that it is easy to embed floating-point units into them. Experimental results on a Xilinx Virtex-II Pro XC2VP100 show that our algorithms achieve good scalability and high sustained GFLOPS performance. We also implement our algorithms on Cray XD1. XD1 is a high-end reconfigurable computing system that employs both general-purpose processors and reconfigurable devices. Our algorithms achieve a sustained performance of 2.06 GFLOPS on a single node of XD1.

Keywords: Scientific Computing, Field-Programmable Gate Arrays

C.3.e Reconfigurable hardware **F.2.1.c Computations on matrices** **G.1.0.g Parallel algorithms**

I. INTRODUCTION

Reconfigurable devices, in particular, Field-Programmable Gate Arrays (FPGAs), offer the design flexibility of software, but with time performance closer to Application Specific Integrated Circuits (ASICs). Due to their low computing density, early reconfigurable devices were mainly used for fixed-point applications. However, with rapid advances in technology, current reconfigurable devices contain significantly more hardware resources than their predecessors. Hence, they are now feasible for a broader range of applications, including those requiring floating-point operations. Some researchers have even suggested that current reconfigurable devices can perform better than general-purpose processors in terms of peak and sustained floating-point performance [1].

Dramatic increases in the computing power of reconfigurable hardware have also made it suitable for high performance computing and supercomputing [2]. Several high-end reconfigurable computing systems have been built. In these systems, general-purpose computing systems work together with reconfigurable devices, and share their memory space. Example systems include SRC MAPstation [3], Cray XD1 [4] and Starbridge Hypercomputer [5], among others. In order to use such systems, high-performance designs on reconfigurable computing systems for scientific computing are necessary.

Floating-point matrix multiplication is a basic operation in many scientific computing applications. Its implementations on reconfigurable computing systems are able to achieve high performance for several reasons. First, a single reconfigurable device contains enough configurable slices to hold multiple floating-point units that can perform computations concurrently. Secondly, current reconfigurable fabrics provide large amounts of on-chip memory as well as abundant number of I/O pins; this can alleviate performance bottleneck due to processor memory bandwidth. These two features enable designs on reconfigurable computing systems to exploit the inherent parallelism of matrix multiplication. Furthermore, current reconfigurable devices contain a large number of multiplexers and embedded fixed-point multipliers. These primitives can be leveraged to build high-speed floating-point adders and multipliers.

However, reconfigurable hardware also poses new challenges for designing parallel algorithms for floating-point matrix multiplication. The resource constraints, such as the number of configurable slices, the size of on-chip memory and the available memory bandwidth, impose multiple constraints on architecture design. These constraints, as well as the inherent characteristics of the matrix multiplication, result in various design tradeoffs. In addition, the routing complexity of the architecture

must also be considered. Using long routing wires on the reconfigurable device will result in low clock speed and reduce the overall performance of the algorithm.

Although there have been some studies on FPGA-based fixed-point matrix multiplication [6], they cannot be applied directly to the floating-point counterpart. Due to the complexity of floating-point adders/multipliers, the design space of floating-point matrix multiplication is much larger. Recently some work has been done on FPGA-based floating-point matrix multiplication [1], [7]. However, to the best of our knowledge, none of the previous work has discussed the design tradeoffs among the area, the total storage size, the latency and the required memory bandwidth.

We first analyze the inherent attributes of matrix multiplication, and derive a lower bound on the performance of any matrix multiplication algorithm on reconfigurable hardware. We then identify the design parameters, including the number of PEs, the total storage size required by the algorithm, and the number of I/O operations performed per clock cycle. By exploring the design space formed by these design parameters, we identify the design tradeoffs for floating-point matrix multiplication. Throughout this paper, our analysis is for a generic reconfigurable fabric, and not for any specific device. Based on the design tradeoffs, we propose three algorithms denoted as Algorithm 1, Algorithm 2 and Algorithm 3. All of these algorithms employ a linear array architecture in which multiple floating-point operations are performed concurrently, effectively utilizing the available resources. By allowing communication between neighboring PEs only, our algorithms make use of the short connection wires on the device and are able to achieve high clock speed. Also, the linear array architecture with a small and regular control logic results in little degradation in performance when the design is scaled over multiple devices. Our algorithms consist of modular PEs with floating-point adders/multipliers embedded in them. Any improvement in the area or speed performance of the floating-point units results in a near-linear improvement in the performance of the PEs. Algorithm 1 and Algorithm 2 achieve the lower bound on the latency of matrix multiplication algorithms. However, the storage size required by these algorithms increases with the problem size. Algorithm 3 trades off the parallel processing capability for smaller storage size and lower memory bandwidth requirement. For given on-chip memory, Algorithm 3 minimizes the requirement on the memory bandwidth.

We implemented our algorithms using Xilinx ISE 7.1i [8], with Xilinx Virtex-II Pro XC2VP100 FPGA as our target device. Although our work is not dependent on the data representation, we used IEEE double-precision (64-bit) numbers in our experiments, as they are used in most scientific

computations. We employed our own floating-point adders and multipliers that follow the IEEE 754 standard [9]. Three implementations are available for each floating-point unit. According to their compliance to the IEEE 754 standard, the implementations are called “full-compliant”, “moderate-compliant”, and “least-compliant”, respectively.

In all of our algorithms, the area for the control logic and the routing constitutes less than 5% of the total area. The number of PEs on the FPGA increases linearly as the available resources increase. As more PEs are configured, typically the achievable clock speed decreases due to the increase in the routing complexity. In our algorithms, the clock speed degradation is less than 35% when the number of PEs increases from 2 to 20. For 64-bit matrix multiplication, the sustained performance of our design is up to 4 GFLOPS, which is comparable with that of the state-of-the-art general-purpose processors. We also implemented our design on one compute node of Cray XD1, using one Xilinx Virtex-II Pro XC2VP50 FPGA (a smaller device). The sustained performance of our design is 2.06 GFLOPS. Note that our designs are not optimized. We also predict the performance as improved floating-point units become available.

The paper is organized as follows. Section II discusses the background and the related work. Section III analyzes the tradeoffs in implementing floating-point matrix multiplication on FPGAs. Section IV describes the proposed algorithms and Section V discusses the implementation of our algorithms. Section VI presents our experimental results. Section VII concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Reconfigurable Computing Systems

Field Programmable Gate Arrays (FPGAs) provide a hardware fabric upon which applications can be programmed. FPGAs are based on look-up tables, flip-flops, and multiplexers. An FPGA device consists of tens of thousands of logic blocks (a cluster 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.

Recently, the computing power of FPGAs has increased rapidly. Besides more configurable slices, current FPGA fabrics now contain large numbers of hardware primitives, such as fixed-point multipliers, Block RAMs (BRAMs), etc [8]. Therefore, FPGAs have become an attractive option for

implementations of floating-point applications because multiple floating-point units can be configured on one device. Several high-end reconfigurable computing systems have also been developed. In these systems, general-purpose computing systems are combined with FPGAs which serve as hardware application accelerators. These systems consist of multiple FPGAs and general-purpose processors that share a memory system. Representative systems include SRC MAPstation [3], Cray XD1 [4] and Starbridge Hypercomputer [5].

We now discuss architectural details of Cray XD1. The basic architectural unit is a compute blade, which contains two AMD Opteron processors and one Xilinx Virtex-II Pro XC2VP50 FPGA. Each FPGA has 32 MB of SRAM, organized into four banks, with a bandwidth of 12.8 GB/s. The FPGA can also access the DRAM of the microprocessors with a bandwidth of 2.8 GB/s. A chassis has six compute blades whose FPGAs are connected in a circular array.

B. Matrix Multiplication on Parallel Systems

There has been extensive work on parallel algorithms for matrix multiplication. Two classical algorithms are Cannon's algorithm [10] and Fox's algorithm [11]. They are designed based on a square processor grid with a block data distribution in which each processor holds consecutive blocks of data. During each iteration, the blocks on one processor are either transferred to its neighboring processors or are broadcast to the other processors in the same row of the grid. Choi et al. developed PUMMA [12] to implement Fox's algorithm on general 2D grids. In [13], Van de Gejin and Watts proposed matrix multiplication algorithms for distributed memory concurrent computers. In their algorithms, the blocks on one processor are broadcast within the row as well as within the column. Also, the computation and communication are efficiently overlapped. Due to the size and routing complexity of FPGA-based floating-point units, a 2D grid is infeasible for implementations on FPGAs. Moreover, broadcasting data within an FPGA-based design causes large overheads. Li and Pan [14] parallelized a well-known sequential algorithm on a linear array of processors. However, their work needs a reconfigurable pipeline optical bus which support massive volume of data transfer. Such a bus is not available on FPGAs, and implementing it on FPGAs causes large overheads.

In contrast to the above previous work, the designs proposed in this paper do not use broadcasting or special hardware to transfer data. In our designs, PEs are connected in a linear array, and only communication among neighboring PEs is allowed. Such communications are realized using the short connection wires on FPGAs. Also, the communications are efficiently overlapped with the computations.

C. Matrix Multiplication on Reconfigurable Computing Systems

Amira et al. implemented a bit-level algorithm for 8-bit fixed-point numbers on FPGAs, which employs a serial-parallel matrix-matrix multiplier based on the Baugh-Wooley algorithm [15]. The I/O bandwidth required by the design is proportional to the problem size. Jang et al. [6] proposed FPGA-based linear array algorithms for fixed-point matrix multiplication. Their algorithms achieve optimal latency, $\Theta(n^2)$, for $n \times n$ matrix multiplication. The algorithms need n multipliers, n adders and total storage of size n^2 words. In [6], no hardware resource constraints are considered. This is because [6] targets fixed-point matrix multiplication, which usually requires much fewer hardware resources than what is provided by the FPGA device. However, due to the design complexity of floating-point units, various resource constraints, such as the number of slices, the size of on-chip memory and the available memory bandwidth, become important factors that may affect the performance of a design. In particular, it is usually impractical to configure n floating-point adders and n floating-point multipliers on one FPGA device, when n is larger than 50.

Floating-point matrix multiplication on FPGAs has been discussed in several papers. In [16], the authors investigated the influence of the floating-point MACs (Multiplier and Accumulators) on the performance of matrix multiplication. The work of [17] studied the implementation of floating-point arithmetic and used matrix multiplication as an example application. The focus of the above mentioned work is FPGA-based designs for arithmetic, and not for matrix multiplication. In [1], Underwood et al. discussed matrix multiplication as part of the BLAS (Basic Linear Algebra Subprograms). Their architecture performs a collection of matrix-vector multiplications in parallel, and blocks the matrix to reduce total storage requirements. In their work, the authors focused on examining the potential capacity of FPGAs in performing floating-point BLAS operations, and comparing the computing capacity of FPGAs with that of general-purpose processors.

In [18], the authors proposed a parallel algorithm for matrix multiplication, which consists of a master processor and multiple slave processors. The master distributes the data from the source matrices to the slaves, so that the slaves can compute different blocks in the result matrix in parallel. This algorithm has a latency of $\Theta(n^3)$, and is only suitable for large matrices. In [7], we extended the architecture in [6] for floating-point matrix multiplication. We analyzed the tradeoffs among area, latency, bandwidth. Based on our analysis, we proposed two algorithms in which the number of floating-point units and the storage size are independent of the problem size. However, in the design of [7], the

total required storage size is $\Theta(n^2)$ words, which may be difficult to satisfy for floating-point matrix multiplication on FPGAs. Moreover, the memory bandwidth requirement in [7] is not optimized.

In this paper, we provide parameterized and optimized designs for floating-point matrix multiplication on reconfigurable computing systems. Our designs can be tuned to adapt to various problem sizes and various hardware resource constraints. With the given hardware resources, our designs are able to achieve optimal latency and minimize the memory bandwidth requirement. We use the most updated floating-point units which are more IEEE 754 compliant than those used in [7]. To illustrate our ideas, we also implement our design on Cray XD1, a high-end reconfigurable computing system.

III. DESIGN ISSUES

In this section, we discuss all design issues encountered by floating-point matrix multiplication. We first present the computational model of our work. Then we analyze the inherent attributes of matrix multiplication. Finally we discuss the design tradeoffs for FPGA-based floating-point matrix multiplication.

A. Computational Model

Our work is based on the computational model shown in Figure 1: the FPGA device contains certain amount of on-chip memory and has access to external (off-chip) memory through I/O pins. Initially, all the source data

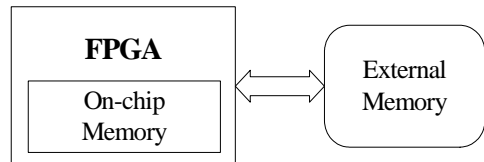


Fig. 1. Computational model used in this work

are stored in the external memory and need to be read into the FPGA. During the computation, intermediate results are stored in the on-chip memory. Finally, the final results are written back into the external memory. The number of configurable slices on the FPGA, the size of the on-chip memory and the available memory bandwidth between the FPGA and the external memory are all limited.

B. Analysis of Matrix Multiplication

We now discuss the lower bound on the latency of any matrix multiplication algorithm, regardless of the precision or the implementation platform. The latency L of any algorithm is determined by its computation time L_1 (cycles for performing computation) and its I/O time L_2 (cycles for I/O). For $n \times n$ matrix multiplication, if p PEs perform computations during every cycle, $L_1 = \Omega(\frac{n^3}{p})$. The I/O time L_2 depends on the available I/O bandwidth, which is assumed to be fixed in the algorithm and independent of the problem size. Hence L_2 is on the same order as the I/O complexity, which refers to the total number of I/O operations performed by the algorithm. It has been proved that the

I/O complexity of any matrix multiplication algorithm is equal to $\Omega(\frac{n^3}{\sqrt{M}})$, where M is the available storage size for the algorithm [19]. This equation holds for $\Theta(1) \leq M \leq \Theta(n^2)$. When $M > \Theta(n^2)$, the I/O complexity remains $\Omega(n^2)$.

If we use B to denote the number of I/O operations performed in each clock cycle, the lower bound on the latency of matrix multiplication is:

$$L \geq \max(L_1, L_2) \geq \max\left(\frac{n^3}{p}, \frac{n^3/\sqrt{M}}{B}\right) \quad (M \leq n^2) \quad (1)$$

Figure 2 illustrates the relationship between M , p , and L . When $p = O(1)$, L is $\Omega(n^3)$ regardless of the value of M as the computation time L_1 forms the bottleneck of the algorithm. This is the case for general-purpose processors. Consider a processor that has a cache large enough to hold all the source matrices and is able to access data from the cache in every cycle. In this case, $M = \Theta(n^2)$ and hence L_2

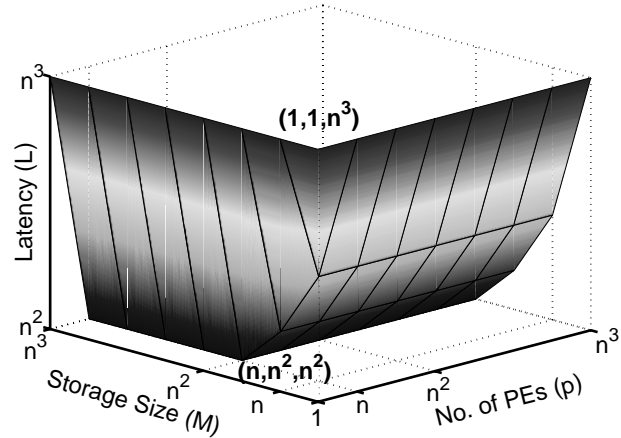


Fig. 2. Tradeoffs for matrix multiplication

is reduced to $\Omega(n^2)$. However, the processor can only perform a constant number of computations during each clock cycle. Hence L_1 , and therefore L remains $\Omega(n^3)$. On the other hand, if $M = \Theta(1)$, L once again cannot be improved, irrespective of the number of PEs that perform computations during each clock cycle. This is the situation where I/O bandwidth, rather than the computing units, forms the bottleneck of the algorithm. The optimal latency can be achieved when $p = \Theta(n)$ and $M = \Theta(n^2)$. In this case, both L_1 and L_2 are $\Omega(n^2)$, and L reaches the lower bound, $\Omega(n^2)$.

Note that for FPGA-based implementation of floating-point matrix multiplication, sometimes $p = \Theta(n)$ and $M = \Theta(n^2)$ cannot be satisfied. Due to the complexity of the floating-point units, only tens of floating-point units can be configured on a current FPGA device, while n could be several magnitudes higher. Moreover, n^2 can be much larger than the size of the on-chip memory of the FPGA. Thus, we also have to consider cases where $p < n$ and $M < n^2$.

Based on Equation 1, the following cases arise for various values of p and M :

- 1) $p \geq n$, $M = n^2$: the optimal latency, $\Theta(n^2)$, can be achieved for matrix multiplication.
- 2) $p \geq n$, $M < n^2$: the latency of matrix multiplication is bounded by the I/O time.

- 3) $p < n$, $M = n^2$: the latency of matrix multiplication is bounded by the computation time.
- 4) $p < n$, $M < n^2$: the latency of matrix multiplication depends on both p and M .

As case 2 and case 3 are not balanced, we do not consider them. In our work, we propose designs for cases 1 and 4.

C. Design Tradeoffs for Floating-Point Matrix Multiplication on FPGAs

From the above analysis, the latency of a matrix multiplication algorithm is determined by several parameters, including the number of PEs p , the storage size M and the memory bandwidth B . These parameters are determined by the hardware resource constraints and result in various design tradeoffs. Such tradeoffs have been rarely discussed in previous works on fixed-point matrix multiplication, because fixed-point arithmetic requires few hardware resources. However, the design tradeoffs are important for FPGA-based floating-point matrix multiplication because floating-point units require a sizeable percentage of the FPGA resources. Also, the large bit-width of floating-point numbers (32-bit or 64-bit) implies higher requirements on storage and memory bandwidth for floating-point matrix multiplication implementations.

The first tradeoff exists between the total storage size used by a design and the required memory bandwidth. Suppose the number of PEs is fixed and L_1 is fixed. From Equation 1, we know that L_2 depends on both M and B . When the total storage size of the matrix multiplication algorithm decreases, B has to be increased to prevent L_2 from increasing. On the other hand, when the available memory bandwidth decreases, M has to be increased.

The second tradeoff lies between the memory bandwidth and the hardware resources required by a design. Suppose M is fixed and the I/O complexity of the matrix multiplication algorithm is fixed. In this case, the I/O time L_2 decreases as the number of I/O operations performed in one clock cycle increases. However, to really reduce the latency, more PEs are needed to reduce computation time L_1 . Thus, to utilize higher memory bandwidth, more hardware resources are needed. On the other hand, when more PEs are implemented to reduce L_1 , higher memory bandwidth is required to reduce L_2 .

The third design tradeoff is between the number of PEs in a design and the size of local storage in each PE. Since the size of on-chip memory of one FPGA device is fixed, with more PEs configured on the device, the local storage of the PEs decreases. On the other hand, if the memory constraint of the FPGA device requires the designer to decrease the size of the local storage, multiple devices may be needed to provide more PEs. For example, in case 1 in Section III.B, a design with n PEs

must have a total storage of size $\Theta(n^2)$ to achieve optimal latency. Thus, the local storage in each PE must be of size $\Theta(n)$. Suppose the FPGA device can at most hold x PEs. For some problem size y , $\Theta(y \times x)$ may exceed the size of the on-chip memory of the device, and optimal latency $\Theta(n^2)$ cannot be achieved. We solve this problem by setting the storage size in each PE as a variable, which is independent of the problem size. For example, if $s \times x$ equals the size of the on-chip memory, the PEs in the algorithm could be designed to store s intermediate results. The design now has to employ $\Theta(\frac{n^2}{s})$ PEs so that $M = \Theta(n^2)$. Thus, $\Theta(\frac{n^2}{s \times x})$ FPGAs are required to achieve the optimal latency.

D. Design Issues

Besides the design tradeoffs caused by the hardware resources, we need to consider several other design issues. The first concerns the floating-point units used in the design. A tradeoff exists between the area occupied by the floating-point matrix multiplication algorithm and its clock speed. To exploit the parallelism of matrix multiplication, we hope to maximize the number of PEs that are performing computation concurrently. Hence, it is desirable that the area of each PE is minimized. On the other hand, the clock speed of the floating-point units, thus the PEs, can be improved through pipelining. The pipeline stages take up extra slices and increase the area of the PEs, further decreasing the number of PEs on the FPGA. This design tradeoff rarely exists in fixed-point matrix multiplication algorithms because the implementation of fixed-point arithmetic is much simpler and seldom needs pipelining. Because of this tradeoff, when we design the floating-point units for the matrix multiplication algorithm, we cannot simply increase their clock speed or reduce their area. Instead, we need to optimize these units in the context of the algorithm so that they are speed-area balanced, as discussed in [20].

Secondly, note that the memory bandwidth available to an FPGA-based design may be constrained by the number of I/O pins of the FPGA device. This is more prone to happen to floating-point based applications, because floating-point numbers have a much larger bit-width than fixed-point numbers. Therefore, it is impractical to design the architecture whose I/O bandwidth increases with the problem size. We thus prefer a linear array architecture whose memory bandwidth is independent of the problem size. Another advantage of the linear array architecture is that it can easily scale over multiple FPGAs.

In our analysis so far we have ignored the on-chip communication costs as well as the routing complexity of the algorithm. These two factors are crucial for the efficiency of the algorithm. First,

to achieve optimal latency, we should ensure that the on-chip communication overheads do not dominate the computation time L_1 . Secondly, the routing of the algorithm should be implementable using available FPGA routing resources. For floating-point applications on FPGAs, the floating-point units usually occupy large numbers of slices and short connection wires on the FPGA. If the long connection wires are used to connect the PEs, the clock speed of the algorithm would be greatly reduced. Therefore, to minimize the communication overheads and the routing complexity, we use the linear array architecture, where only communication between neighboring PEs is allowed. This architecture can make good use of the short connection wires on the FPGA, and reduce the communication overheads among the PEs.

IV. MATRIX MULTIPLICATION ALGORITHMS

In this section, we present our matrix multiplication algorithms. Consider $C = A \times B$, where A , B and C are all $n \times n$ matrices. Each element of the matrices is a floating-point word. Matrices A and B are initially stored in the external memory, and are read into the FPGA during the algorithms. The final results of C are written back to the external memory.

We propose three algorithms, and they all employ a linear array of PEs. The first two algorithms need $M = \Theta(n^2)$ floating-point words, and achieve the optimal latency $\Theta(n^2)$. In the third algorithm, M can be less than n^2 , and the required memory bandwidth is minimized. With p PEs, the latency of the third algorithm is $\Theta(\frac{n^3}{p})$. At the end of this section, we summarize the features of these three algorithms and discuss the scenarios where they are most suitable.

A. Algorithm 1

In Algorithm 1, there is a linear array of $n \times \frac{n}{s}$ PEs. Each PE contains one floating-point multiplier, one floating-point adder, two registers and s ($1 \leq s \leq n$) words of storage. For simplicity of analysis, we assume n is a multiple of s . Each PE has one input port for A , one input port for B , and one output port for C . The l th PE receives data from the $(l-1)$ th PE on its left, and passes them to the $(l+1)$ th PE on its right. The final elements of C are transferred from right to left. The first PE, PE_0 , is connected to the external memory. The I/O bandwidth of this algorithm is 3 words per clock cycle.

Algorithm 1 is shown in Figure 3, and its architecture is shown in Figure 4. In this algorithm, each PE calculates s elements of C . PE_0 computes $c_{00}, c_{\frac{n}{s},0}, \dots, c_{(n-\frac{n}{s}),0}$; PE_1 computes $c_{(\lceil \frac{1}{n} \rceil),1}, c_{(\lfloor \frac{1}{n} \rfloor + \frac{n}{s}),1}, \dots, c_{(n-(\frac{n}{s}-\lfloor \frac{1}{n} \rfloor)),1}, \dots$; PE_l computes $c_{(\lfloor \frac{l}{n} \rfloor), (l \bmod s)}, c_{(\lfloor \frac{l}{n} \rfloor + \frac{n}{s}), (l \bmod s)}, \dots$,

```

for  $PE\ j = 0$  to  $\frac{n^2}{s} - 1$  (in parallel) do
  for 1 to  $n$  cycles do
    shifts data in RR.B right to  $PE_{j+1}$ 
  end for
  for  $n + 1$  to  $n^2 + n$  cycles do
    shifts data in RR.A, RR.B right to  $PE_{j+1}$ 
    if  $RR.B = b_{k,j \bmod s}$  then
      copy it into RR.B1 or RR.B2 alternatively
    end if
     $\{i = \lfloor \frac{j}{s} \rfloor, \lfloor \frac{j}{s} \rfloor + \frac{n}{s}, \dots, n - (\frac{n}{s} - \lfloor \frac{j}{n} \rfloor)\}$ 
    if  $RR.A = a_{ik}$  then
       $\{b_{kj}$  is either in RR.B1 or in RR.B2 $\}$ 
       $c'_{ij} = c_{ij} + a_{ik} \times b_{kj}$ 
    end if
  end for
end for

```

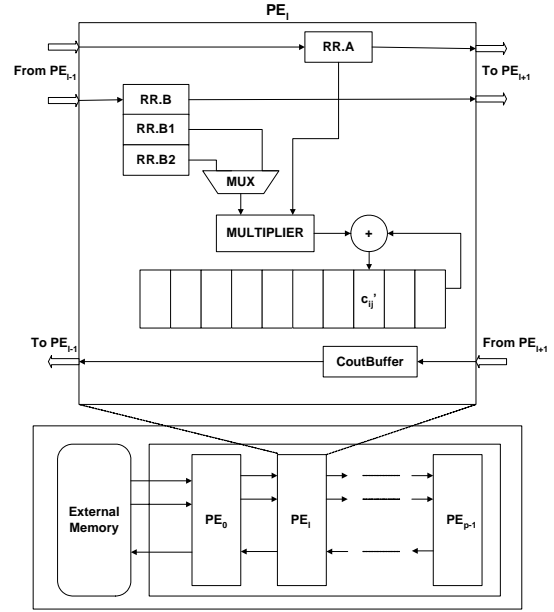


Fig. 3. Algorithm 1: Describes the cycle-specific behavior of each PE

Fig. 4. Architecture for Algorithm 1

$c_{(n - (\frac{n}{s} - \lfloor \frac{j}{n} \rfloor)), (l \bmod s)}$. Matrices A and B are fed into the PEs in column-major and row-major patterns respectively. B starts n cycles earlier than A , and is fed into the lower I/O port in Figure 4. A is fed into the upper I/O port. We group n cycles into a phase. In phase 0, the first row of matrix B is fed into PE_0 . In phase k , column k of matrix A ($a_{ik}, 0 \leq i \leq n - 1$) and row $k + 1$ of matrix B ($b_{(k+1),j}, 0 \leq j \leq n - 1$) enter PE_0 in order. For example, a_{11} enters the upper I/O port of PE_0 during the same cycle as b_{21} enters the lower I/O port of PE_0 . As b_{kj} traverses the PEs, it is picked up by the PEs that are computing c_{ij} . These PEs keep b_{kj} in the registers until a_{ik} passes through. When a_{ik} reaches the PEs computing c_{ij} , the PEs update $c'_{ij} = c_{ij} + a_{ik} \times b_{kj}$. c'_{ij} is the intermediate result of c_{ij} ($0 \leq i \leq n - 1, 0 \leq j \leq n - 1$) and is stored in the storage within the PE that is computing c_{ij} .

The final result of C traverses the linear array in the reversal direction, that is, from PE_{n-1} to PE_0 . Suppose during a clock cycle, PE_l generates a final element in C , c_{ij} . PE_l then transfers c_{ij} to PE_{l-1} . If PE_l receives $c_{i'j'}$ from PE_{l+1} at the same clock cycle, it stores $c_{i'j'}$ in its *CoutBuffer*. The data in *CoutBuffer* are transferred to PE_{l-1} in clock cycles in which PE_l does not generate C elements. The *CoutBuffer* is of the same size as the storage which stores the intermediate result of matrix C .

For the algorithm to perform correctly, we observe that two requirements should be satisfied. We then state these requirements, and show how they are satisfied by the algorithm with the number of registers in the PEs.

1. Since a_{ik} stays at each PE for one clock cycle only, b_{kj} should arrive no later than a_{ik} .

The number of cycles needed for b_{kj} to arrive at PE_l is $(k-1)n+j+l$. a_{ik} needs $n+(k-1)n+i+l$ cycles to arrive at PE_l . The requirement is satisfied since $(k-1)n+j+l \leq n+(k-1)n+i+l$ for all i, j . Suppose $s = \frac{n}{2}$, there are $2n$ PEs in the linear array. $c_{1,n-1}$ is computed by PE_{n-1} . We show how $b_{2,n-1}$ arrives at PE_{n-1} no later than a_{12} for $c'_{1,n-1} = c'_{1,n-1} + a_{12} \times b_{2,n-1}$. $b_{2,n-1}$ needs $(2-1)n+n$ cycles to arrive at PE_0 and needs $(n-1)$ more cycles to move to PE_{n-1} , requiring a total of $3n-1$ cycles. a_{12} needs $n+(2-1)n$ cycles to arrive at PE_0 and $(n-1)$ more cycles to move to PE_{n-1} . Therefore, a_{12} arrives PE_{n-1} at $3n$ -th cycle, later than $b_{2,n-1}$.

2. Once b_{kj} reaches PEs that compute c_{ij} , a copy of b_{kj} should reside in these PEs until a_{ik} arrives.

A PE can hold b_{kj} in its registers. We now show that using two registers in each PE can satisfy this requirement. The registers are denoted as $RR.B1$ and $RR.B2$ in Figure 4, and they always store two consecutive elements (b_{kj} and $b_{k+1,j}$). For example, when b_{22} arrives at PE_2 , b_{02} is in $RR.B1$ and b_{12} is in $RR.B2$. If we can prove that $a_{n-1,0}$ has passed PE_2 by then, b_{22} can replace b_{02} in $RR.B1$ because b_{02} is no longer needed by PE_2 . Regardless of the value of s , b_{kj} is no longer needed by a PE after a_{nk} has passed the PE. For PE_l , a_{nk} arrives at the $(n+(k-1)n+n+l)$ th cycle, and $b_{k+2,j}$ arrives at the $((k+1)n+j+l)$ th cycle. Since $j < n$, $n+(k-1)n+n+l > (k+1)n+j+l$ for all k and l . Thus b_{kj} can be replaced when $b_{k+2,j}$ arrives at PE_l . This proves that each PE only needs two registers to hold b_{kj} .

We illustrate Algorithm 1 using an example in Figure 5. When $n = 4$ and $s = 2$, there are totally 8 PEs. PE_0 calculates c_{00} and c_{20}, \dots , PE_4 calculates c_{10} and c_{30}, \dots , and PE_7 calculates c_{13} and c_{33} . In phase 0, PE_0 stores b_{00} in its register; in phase 1, PE_0 updates $c'_{00} = c'_{00} + a_{00} \times b_{00}$, $c'_{20} = c'_{20} + a_{20} \times b_{00}$. b_{00} reaches PE_4 in phase 1 and is stored in the register. In phase 2, when column 0 of A reaches PE_4 , PE_4 updates $c'_{10} = c'_{10} + a_{10} \times b_{00}$ and $c'_{30} = c'_{30} + a_{30} \times b_{00}$. Figure 6 shows the data flow for Algorithm 1, assuming the pipeline delays of the adder and the multiplier are both 2. Due to space limitation, we only show the data flow from clock cycle 5 to cycle 13.

The last element, $a_{n-1,n-1}$, arrives at the last PE at the $(n+(n-1)n+n+\frac{n}{s}-1)$ th cycle. Suppose the computation related to $a_{n-1,n-1}$ completes in q cycles. For fixed-point numbers, $q = 1$. However, floating-point arithmetic usually takes multiple clock cycles, hence $q > 1$. The final $c_{n-1,n-1}$ is generated at $(n^2 + \frac{n^2}{s} + n - 1 + q)$ th cycle. It then traverses the PEs and is written back to the external memory at the $(n^2 + 2\frac{n^2}{s} + n + q - 1)$ th cycle. Therefore, the latency of the algorithm $L = \Theta((1 + \frac{2}{s})n^2)$. As

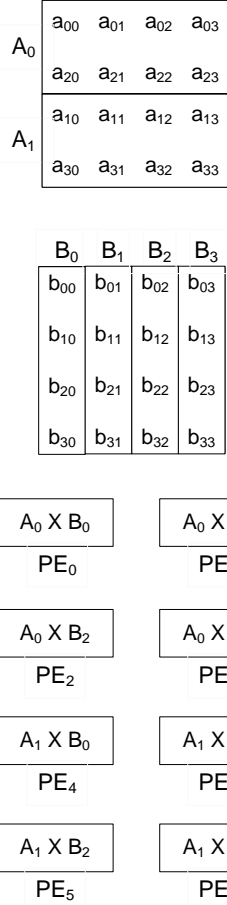


Fig. 5. Computation illustrations for Algorithm 1 ($n = 4, s = 2$)

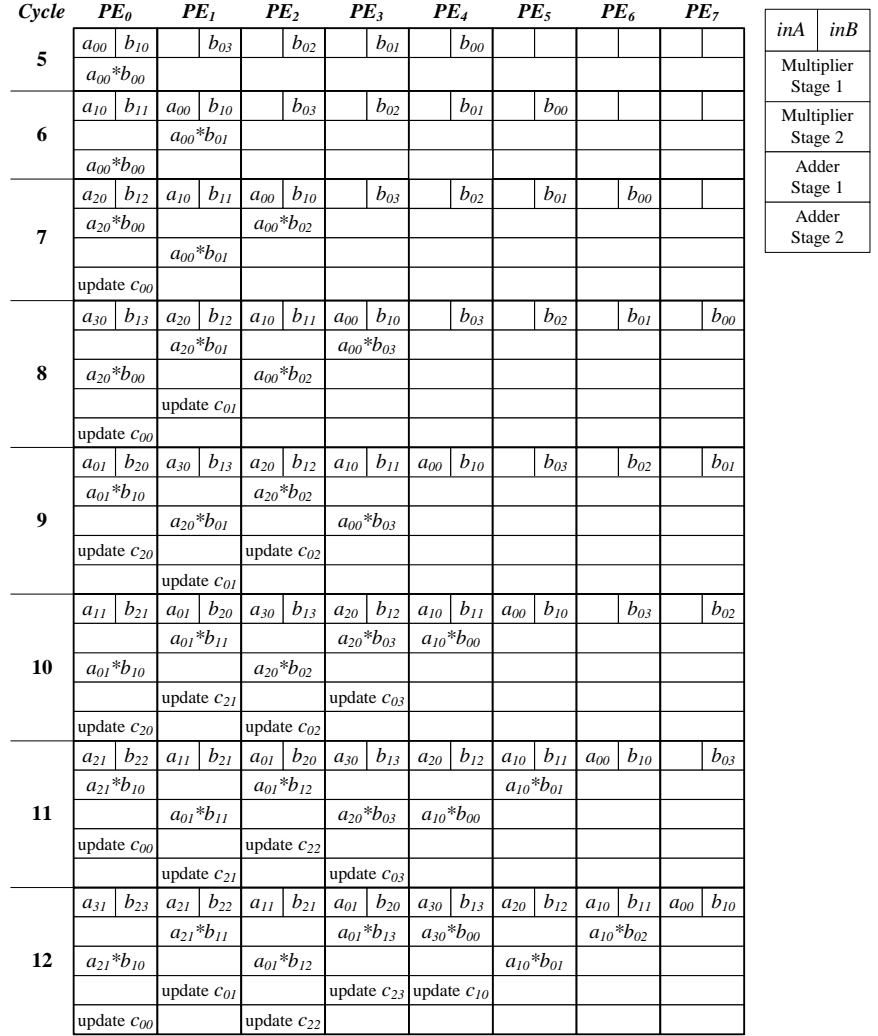


Fig. 6. Data flow for Algorithm 1 ($n = 4, s = 2$ and the pipeline delays of both the adder and the multiplier are 2)

$1 \leq s \leq n$, we have $1 < 1 + \frac{2}{s} \leq 3$, and thus $L = \Theta(n^2)$. In Algorithm 1, the size of storage M is $s \times \frac{n^2}{s} = \Theta(n^2)$ words. According to Section III, Algorithm 1 achieves the asymptotically optimal latency.

B. Algorithm 2

From the discussion in Section III, we know that a matrix multiplication algorithm that requires higher I/O bandwidth can achieve shorter latency. Therefore, we propose the second algorithm whose I/O bandwidth is $3r$ ($1 \leq r \leq n$) words per clock cycle, and whose latency is approximately $1/r$ of that of Algorithm 1.

In Algorithm 2, there is a linear array of $\frac{n^2}{r^2 s}$ PEs. Each PE contains r^2 registers, r^2 floating-point multipliers, r^2 floating-point adders, and r^2 storage blocks of s words ($1 \leq s \leq \frac{n}{r}$, $1 \leq r \leq n$). For simplicity, we assume n is a multiple of both s and r . 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 l th PE receives the input data from the $(l-1)$ th PE and passes them to the $(l+1)$ th PE. Figure 7 and Figure 8 show Algorithm 2 and its architecture, respectively. In both of the figures, $r = 2$; each PE contains two *CoutBuffers*, whose size is the same as the storage in the PE which stores the intermediate results of C .

In Algorithm 2, C are partitioned into r^2 submatrices of size $\frac{n}{r} \times \frac{n}{r}$. We use c_{ij}^{xy} ($0 \leq x, y \leq r-1$, $0 \leq i, j \leq \frac{n}{r} - 1$) to denote elements in C_{xy} . PE_0 computes $c_{00}^{xy}, c_{\frac{n}{s},0}^{xy}, \dots, c_{(n-\frac{n}{s}),0}^{xy}$; PE_1 computes $c_{(\lfloor \frac{1}{n} \rfloor),1}^{xy}, c_{(\lfloor \frac{1}{n} \rfloor + \frac{n}{s}),1}^{xy}, \dots, c_{(n-(\frac{n}{s}-\lfloor \frac{1}{n} \rfloor)),1}^{xy}$; \dots ; PE_l computes $c_{(\lfloor \frac{1}{n} \rfloor), (l \bmod \frac{n}{r})}^{xy}, c_{(\lfloor \frac{1}{n} \rfloor + \frac{n}{s}), (l \bmod \frac{n}{r})}^{xy}, \dots, c_{(n-(\frac{n}{s}-\lfloor \frac{1}{n} \rfloor)), (l \bmod \frac{n}{r})}^{xy}$. The local storage in PE_l is used to store intermediate results for these elements.

The basic idea of Algorithm 2 is to perform the calculations of C_{xy} ($0 \leq x, y \leq r-1$) in parallel, using r^2 MACs per PE. $A_{xk} \times B_{ky}$ for each x, y could be performed using Algorithm 1, with one MAC, 2 registers and 2 I/O ports. However, in this way, each PE would need $2r^2$ registers and $2r^2$ I/O ports, which is impractical for implementations on FPGAs. On the other hand, Algorithm 2 is able to reduce the requirements on registers and I/O ports.

In Algorithm 2, we partition A, B into r submatrices. A^0 contains rows $0, \dots, \frac{n}{r} - 1$ of A ; A^1 contains rows $\frac{n}{r}, \dots, \frac{2n}{r} - 1$ of A ; and so on. B^0 contains columns $0, \dots, \frac{n}{r} - 1$ of B ; B^1 contains columns $\frac{n}{r}, \dots, \frac{2n}{r} - 1$ of B ; and so on. The elements of the submatrices of A and B are denoted as a_{ik}^x and b_{kj}^y respectively, $0 \leq x, y \leq r-1$, $0 \leq i, j \leq \frac{n}{r} - 1$, $0 \leq k \leq n-1$.

As in Algorithm 1, matrix A and B are fed to the PEs in column-major and row-major pattern respectively. B starts $\frac{n}{r}$ cycles earlier than A . During each clock cycle, one element from each of the r submatrices of A is fed into the PEs; similarly, r elements of B are fed into the PEs. We group $\frac{n}{r}$ clock cycles into a phase. In phase 0, the first row of matrix B is fed into PE_0 . In phase k , column k of matrix A and row $k+1$ of matrix B enter PE_0 in order. As b_{kj}^y traverses the linear array, they are stored in registers by the PEs that compute c_{ij}^{xy} until a_{ik}^x passes through. When the PE which is in charge of c_{ij}^{xy} is not idle, the r^2 MACs in the PE are performing $(c_{ij}^{xy})' = (c_{ij}^{xy}) + a_{ik}^x \times b_{kj}^y$, $0 \leq x, y \leq r-1$. Thus, each element is shared among r MACs. This sharing allows r^2 PEs to perform computation concurrently with $2r$ registers and $2r$ input ports.

Figure 9 illustrates how Algorithm 2 is applied to our example in Section IV, when $r = 2$, $s = 1$. There are totally 4 PEs. PE_0 calculates $c_{00}, c_{02}, c_{20}, c_{22}; \dots$; PE_3 calculates $c_{11}, c_{13}, c_{31}, c_{33}$. In phase 0, PE_0 stores b_{00} and b_{02} in registers; in phase 1, PE_0 updates $c'_{00} = c_{00} + a_{00} \times b_{00}$,

for $PE\ j = 0$ to $\frac{n^2}{4s} - 1$ (in parallel) **do**
 for 1 to $\frac{n}{2}$ cycles **do**
 shifts data in RR1.B and RR2.B right to PE_{j+1}
 if RR1.B = $b_{k,j}^0 \bmod s$ **then**
 copy it into RR1.B1
 end if
 if RR2.B = $b_{k,j}^1 \bmod s$ **then**
 copy it into RR2.B1
 end if
 end for
 for $\frac{n}{2} + 1$ to $\frac{n}{2} + \frac{n}{2}$ cycles **do**
 shifts data in RR1.A, RR2.A, RR1.B, RR2.B right to PE_{j+1}
 if RR1.B = b_{kj}^0 **then**
 copy it into RR1.B1 or RR1.B2 alternatively
 end if
 if RR2.B = b_{kj}^1 **then**
 copy it into RR2.B1 or RR2.B2 alternatively
 end if
 $\{i = \lfloor \frac{j}{s} \rfloor, \lfloor \frac{j}{s} \rfloor + s, \dots, n - (\frac{n}{s} - \lfloor \frac{j}{s} \rfloor)\}$
 if RR1.A = a_{ik}^0 **then**
 $\{b_{jk}^0$ is in either RR1.B1 or RR1.B2
 $c_{ij}^{00'} = c_{ij}^{00'} + a_{ik}^0 \times b_{kj}^0$
 $c_{ij}^{01'} = c_{ij}^{01'} + a_{ik}^0 \times b_{kj}^1$
 end if
 if RR2.A = a_{ik}^1 **then**
 $\{b_{jk}^1$ is in either RR2.B1 or RR2.B2
 $c_{ij}^{10'} = c_{ij}^{10'} + a_{ik}^1 \times b_{kj}^0$
 $c_{ij}^{11'} = c_{ij}^{11'} + a_{ik}^1 \times b_{kj}^1$
 end if
 end for
end for

Fig. 7. Algorithm 2, $r = 2$: Describes the cycle-specific behavior of each PE

$c'_{02} = c'_{02} + a_{00} \times b_{02}$, $c'_{20} = c'_{20} + a_{20} \times b_{00}$, $c'_{22} = c'_{22} + a_{20} \times b_{02}$. In phase 2, the first column of A reaches PE_3 , and PE_3 starts updating elements of C .

Similar as for Algorithm 1, we can prove Algorithm 2 performs correctly. The proof is not shown here due to space limitation. In Algorithm 2, the final result of $c_{n-1,n-1}$ is available at $(n + (n-1)\frac{n}{r} + \frac{n^2}{r^2s} - 1 + q)$ th cycle. As $c_{n-1,n-1}$ takes $\frac{n^2}{r^2s}$ clock cycles to traverse back to the external memory, the matrix multiplication completes in $\frac{n^2}{r} + \frac{2n^2}{r^2s} + n - \frac{n}{r} + q - 1$ cycles. Therefore, $L = O((\frac{1}{r} + \frac{2}{r^2s})n^2)$. As $1 \leq r \leq n$ and $1 \leq s \leq \frac{n}{r}$, $0 \leq \frac{1}{r} + \frac{2}{r^2s} \leq 3$, and thus $L = \Theta(n^2)$. The size of storage M is $s \times r^2 \times \lceil \frac{n^2}{rs} \rceil =$

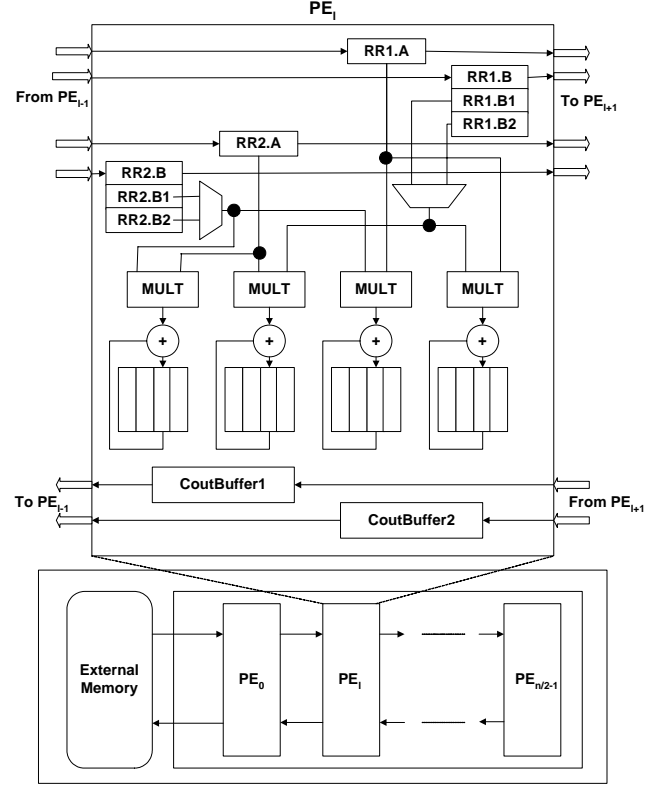


Fig. 8. Architecture for Algorithm 2, $r = 2$

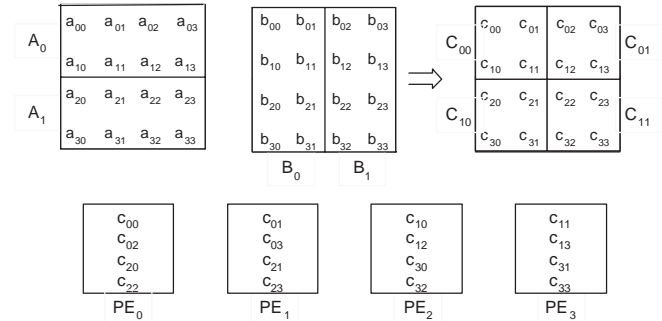


Fig. 9. Computation illustrations for Algorithm 2 ($n = 4$, $s = 1$, $r = 2$)

$\Theta(n^2)$ words. According to Section III, Algorithm 2 achieves the asymptotically optimal latency.

C. Algorithm 3

In Algorithms 1 and 2, the total storage size, M , has to be equal to n^2 . In the third algorithm, M is determined by the available storage size on the device, and thus can be smaller than n^2 . For the given p and M , Algorithm 3 minimizes the required memory bandwidth.

The architecture of Algorithm 3 is similar to that of Algorithm 1, and hence is not shown here. There are p PEs connected in a linear array, and the first PE, PE_0 , is connected to the external memory. Each PE consists of one floating-point multiplier and one floating-point adder. However, different from Algorithm 1, each PE in Algorithm 3 has a local storage of size $\frac{M}{p}$, and $\frac{2\sqrt{M}}{p}$ registers.

In Algorithm 3, we are actually performing block matrix multiply with the block size of $\sqrt{M} \times \sqrt{M}$. When $\sqrt{M} = n$, Algorithm 3 is the same as Algorithm 1. When $\sqrt{M} < n$, the blocks are denoted as A_{gz} and B_{zh} , where $g, z, h = 0, 1, \dots, \frac{n}{\sqrt{M}} - 1$. Without loss of generality, we assume n is a multiple of \sqrt{M} , and \sqrt{M} is a multiple of p .

```

{for every block matrix multiply}
for PE  $k = 0$  to  $p - 1$  (in parallel) do
  if  $k = j \pmod p$  then
    copy  $b_{qj}$  to a register
  end if
  if  $q = 1$  (the first row of the  $B$  block) then
    shift  $b_{qj}$  right to  $PE_{k+1}$ 
  else
    shift  $b_{qj}, a_{iq}$  right to  $PE_{k+1}$ 
    for every  $b_{qj'}$  in the registers do
       $c'_{ij'} \leq c'_{ij'} + a_{iq} \times b_{qj'}$ 
    end for
  end if
end for

```

Fig. 10. Algorithm 3 for matrix multiplication

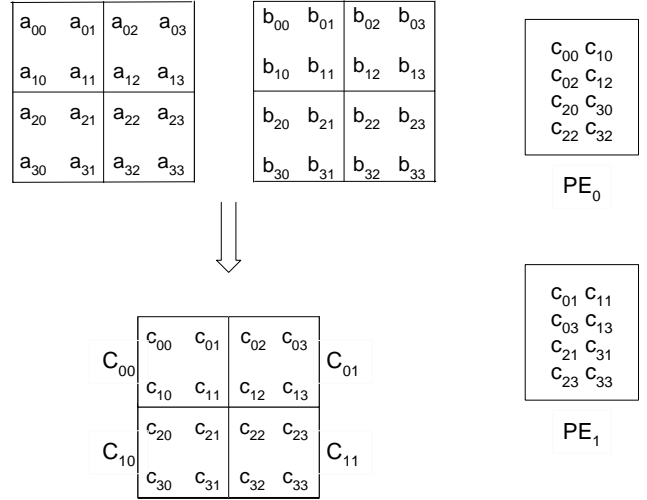


Fig. 11. Computation illustrations for Algorithm 3 ($n = 4, M = 4, p = 2$)

In Algorithm 3, the blocks of B are read in the column-major order and the blocks of A are read in the row-major order. For each block matrix multiply $A_{gz} \times B_{zh}$, A_{gz} is read in the column-major pattern, while B_{zh} is read in the row-major pattern. PE_k is in charge of computing the k th, $(p+k)$ th, \dots , $((\frac{\sqrt{M}}{p} - 1)p + k)$ th columns of C_{gh} . Before the computation starts, the first row of B_{zh} is read into the architecture. As these \sqrt{M} numbers traverse the linear array, PE_k ($k = 0, \dots, p - 1$) stores the k th, $(p+k)$ th, \dots , $((\frac{\sqrt{M}}{p} - 1)p + k)$ th numbers into its registers. Afterwards, every $\frac{\sqrt{M}}{p}$ clock cycles, one element of A_{gz} and one element of B_{zh} are read into the architecture. As a_{iq} passes

through one PE, it is multiplied with every stored B element whose row index is q . The intermediate results for C_{gh} are stored in the local storage of the PEs.

Figure 11 illustrates Algorithm 3 applied to our example in Section IV, for $M = 4$, $p = 2$. PE_0 computes column 0 and PE_1 computes column 1 of $C_{00}, C_{01}, C_{10}, C_{11}$. Note that since the storage size in each PE is $\sqrt{M} = 2$, at any time each PE at most stores two intermediate results of C .

C_{gh} ($g, h = 0, 1, \dots, \frac{n}{\sqrt{M}} - 1$) need to be accumulated. According to the analysis of Algorithm 1, the effective latency for each block matrix multiply, T' , equals $(\frac{\sqrt{M}}{p})^3 = \frac{M\sqrt{M}}{p}$. Thus, every T' clock cycles, an intermediate result of C is generated. If T' is larger than the number of pipeline stages in the adder, we can simply use a floating-point adder for the accumulation without incurring any data hazards. Otherwise, we can perform the accumulation on a host processor. This accumulation is not considered in this paper.

In Algorithm 3, we perform $(\frac{n}{\sqrt{M}})^3$ block matrix multiplies whose effective latency is $\frac{M\sqrt{M}}{p}$. Since \sqrt{M} clock cycles are needed to read the first row of the first B block, the effective latency of Algorithm 3 equals $\sqrt{M} + (\frac{n}{\sqrt{M}})^3 \times \frac{M\sqrt{M}}{p} = \sqrt{M} + \frac{n^3}{p} = \Theta(\frac{n^3}{p})$. In the algorithm, 3 words are exchanged with the external memory every $\frac{\sqrt{M}}{p}$ cycles. Thus, the required memory bandwidth is $\Theta(\frac{p}{\sqrt{M}})$. According to Equation 1, the memory bandwidth required by Algorithm 3 achieves the theoretical lower bound, under the given storage size and the number of PEs.

D. Analysis of the Proposed Algorithms

Table I summarizes the performance of the proposed matrix multiplication algorithms. In the table, p is the number of PEs in the architecture; M is the storage size; B is the number of I/O operations performed in each clock cycle.

The latency of both Algorithm 1 and Algorithm 2 is on the order of n^2 , and is inversely proportional to s . These two algorithms are suitable for cases where the size of on-chip memory is small, while multiple FPGAs are available. The value of s depends on n , M and the number of available FPGAs; the value of r in Algorithm 2 is dependent on B .

Unlike in the other two algorithms, the latency of Algorithm 3 is on the order of n^3 . Under the given

TABLE I

COMPARISON OF DIFFERENT MATRIX MULTIPLICATION ALGORITHMS

	p	M	B	Latency
Algorithm 1	$\Theta(\frac{n^2}{s})$	n^2	$\Theta(1)$	$\Theta((1 + \frac{2}{s})n^2)$
Algorithm 2	$\Theta(\frac{n^2}{r^2s})$	n^2	$\Theta(r)$	$\Theta((\frac{1}{r} + \frac{2}{r^2s})n^2)$
Algorithm 3	p	M	$\Theta(\frac{p}{\sqrt{M}})$	$\Theta(\frac{n^3}{p})$

storage size M , its required memory bandwidth, B , is minimized. In Algorithm 3, p is determined by M and B . Hence, this algorithm is more flexible and can be adapted to various scenarios with different hardware requirements. Note that when $M = n^2$ and $p = n$, Algorithm 3 is identical to Algorithm 1 when $s = n$.

V. IMPLEMENTATION ON FPGAS

In this section, we first introduce our floating-point units, and then discuss some implementation issues for our algorithms. These issues include the avoidance of data hazards, implementation of local storage and FPGA configuration.

A. Floating-Point Units

In our experiments, we used our own double-precision floating-point adders and multipliers. For each floating-point unit, there are three implementations. The first implementation is “full-compliant” with IEEE standard 754 [9]; it supports NaN diagnostics, all four IEEE rounding modes, exception generation, and denormal numbers. The “moderate-compliant” implementation implements the round-to-nearest rounding mode and support for exceptions, but does not implement NaN diagnostics; denormal numbers are treated as zero. In the “least-compliant” implementation, the only rounding mode implemented is round toward zero; denormal numbers are treated as zero, exceptions are not generated, and NaN diagnostics are not supported. Table II gives the characteristics of the floating-point units we used in our experiments. The details of the implementations of the floating-point adders/multipliers can be found in [20].

Floating-point units are usually pipelined to achieve high clock speed. Moderate pipelining can exploit the unused flip-flops in the slices of the FPGA fabric with a slight increase in area. However, too much pipelining results in diminishing returns; that is, the cost of area offsets the gain in clock speed [20]. Therefore, our floating-point units are only moderately pipelined and run at 170 MHz.

TABLE II
ANALYSIS OF 64-BIT FLOATING-POINT UNITS

Compliance	Least		Moderate		Full	
	adder	multiplier	adder	multiplier	adder	multiplier
No. of Pipeline stages	11	8	17	13	16	17
Area(slices)	892	835	1334	1187	1503	2085
Clock Speed(MHz)	170	170	170	170	170	170

The floating-point units in our algorithms are independent modules; that is, the algorithms and

architectures are independent of the floating-point units used. When we replace the current floating-point units with new ones, we do not need to modify the algorithms or the architectures. In particular, when smaller and faster floating-point units are available, they can be easily plugged into our designs without effort. The modularity of our designs also simplifies the changing of the precision of the designs. For example, to design PEs for 32-bit matrix multiplication, we only need to replace the 64-bit floating-point units with 32-bit floating-point units and change the bit-width of the data bus inside the PEs. In Section V, we will investigate the performance improvement of our designs when the performance of the floating-point units is improved.

B. Data Hazards

When our algorithms are used with floating point adders/multipliers, we need to be careful to avoid data hazards. For example, in Algorithm 1, each PE updates c'_{xy} , 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 12: at a certain cycle, c'_{11} is being written to and read from the storage concurrently. In this case, the data read out is probably invalid and the final result will be erroneous.

To avoid this problem, we have to ensure that the problem size n is larger than the number of pipeline stages in the floating-point adder. In our example, when $n > 4$, there will be no data hazard. This problem also arises in Algorithm 2. In this case, it can be avoided if $\frac{n}{r}$ is larger than the number of pipeline stages in the adder. In Algorithm 3, the value of c'_{ij} is updated every $\frac{M}{p}$ cycles. If $\frac{M}{p}$ is larger than the number of pipeline stages in the adder, no data hazards are caused.

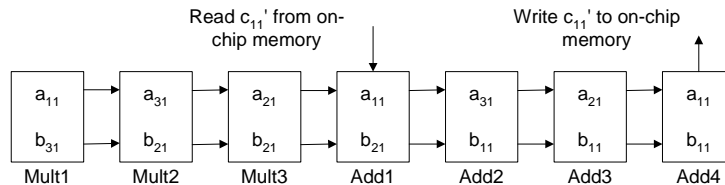


Fig. 12. Potential data hazard in Algorithm 1 (Mult_{*i*} denotes the *i*th pipeline stage of the floating-point multiplier; and Add_{*j*} denotes the *j*th pipeline stage of the floating-point adder)

C. Matrix Multiplication Implementation

In our algorithms, each PE needs some amount of local storage, which is implemented using the on-chip memory of FPGAs. There are two types of on-chip memory on current 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 few slices. To conserve area and reduce routing complexity, we use BRAMs as the local storage within the PEs. The *CountBuffers* in the PEs are also implemented using BRAMs.

Another consideration in the implementation of our algorithms on FPGAs is FPGA configuration. 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 5 MB in size, we need $80 \times 5 = 400$ MB of configuration memory outside the device. Second, the time needed for configuration may be too long. A full reconfiguration of the Virtex-II Pro FPGA takes about 100 msec. When the problem size is 100 and the FPGA device is clocked at 150 MHz, the matrix multiplication can be completed in around 1 msec using our algorithm. In this case, the configuration time is too long compared with the computation time.

Our algorithms handle various problem sizes in a flexible way. In our algorithms, the control logic keeps multiple counters. These counters control the PEs to perform different operations at different clock cycles. For example, in the implementation of Algorithm 1, when one counter has the upper limit set as n , the accumulation in each PE happens every n cycles for a particular entry c'_{ij} in C . For all problem sizes p ($p \leq n$), we still use the same configuration file for problem size n . However, the upper limit of the counter is set to p . Consequently, the accumulation of c'_{ij} occurs every p cycles. Therefore, using Algorithm 1, the full reconfiguration is only performed once for the maximum problem size n that we will handle, before any computation starts. For different problem sizes, only the upper limits of the counters are changed.

VI. EXPERIMENTAL RESULTS

Our target device is Xilinx Virtex-II Pro XC2VP100, which is one of the largest FPGA devices available. It contains 44,096 slices, and about 1 MB on-chip memory. We used the Xilinx ISE 7.1i and Mentor Graphics ModelSim 5.7 development tools [8], [21]. We first examine the scalability and modularity of our designs, using Algorithm 1 and Algorithm 2 as examples. We then discuss the memory bandwidth requirement of Algorithm 3. Next, we present the sustained MFLOPS performance of our designs, and compare it with previous work and several general-purpose processors. To

study the performance gap between our designs and the design generated by a system-level design tool, we compare our designs with the implementation generated by Celoxica Handel-C environment DK1. In all the experiments, $r = 2$ for Algorithm 2.

A. Performance Analysis

Table III shows the characteristics of the MACs and PEs for 64-bit matrix multiplication. A MAC consists of one floating-point adder and one floating-point multiplier. The “Full”, “Moderate” and “Least” MACs contain full-compliant, moderate-compliant and least-compliant floating-point adders and multipliers, respectively. Each PE in Algorithm 1 contains 1 MAC and s storage entries; each PE in Algorithm 2 contains 4 MACs and $4s$ storage entries. As BRAMs are used for the storage within the PEs, the value of s rarely affects the area of a PE, as shown in Table III. As s increases, more BRAMs are used.

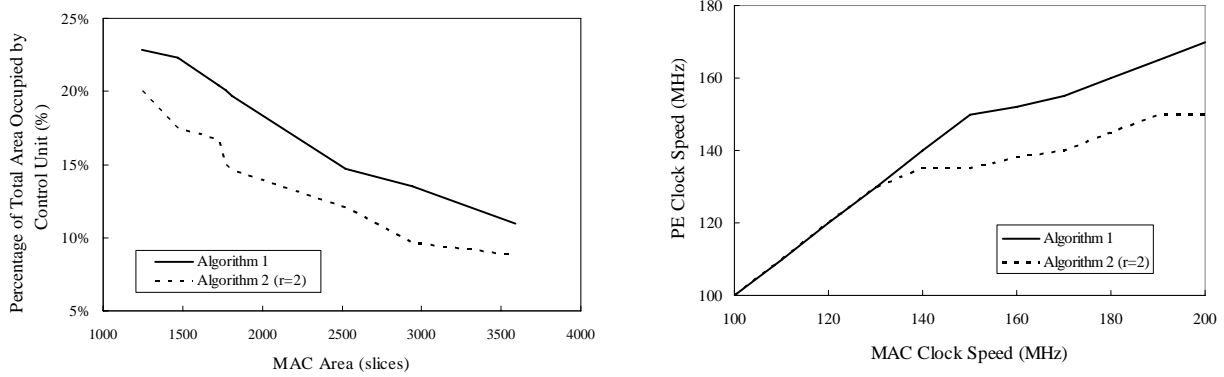
In each PE, some slices are used for the control logic and the routing. These slices form what we call “control unit” of the PE. Figure 13 shows the area and clock speed performance of the PEs when various floating-point units are used. In the experiments, we vary the number of pipeline stages in the floating-point adders/multipliers to vary their area and clock speed. Figure 13(a) shows that when the area of the MAC increases, the percentage of the total area occupied by the control unit decreases. This is because the area of the control units in both algorithms remain constant, regardless of the floating-point units used. Figure 13(b) shows that as the achievable clock speed of the MAC increases, the clock speed of the PE increases accordingly. However, the latter increases slower than the former, due to the routing of the control unit. Furthermore, as Algorithm 2 requires more complex routing and control logic, it can at most achieve 150 MHz. Figure 13 indicates that the implementation of the PEs in our algorithms is independent of the actual implementation of the MACs. Any area or speed improvement of the MACs results in performance improvement of the PEs.

TABLE III
CHARACTERISTICS OF MAC AND PEs FOR 64-BIT MATRIX MULTIPLICATION

	s	No. of BRAMs	Area (slices)			Speed (MHz)		
			Least	Moderate	Full	Least	Moderate	Full
MAC	-	-	1727	2521	3588	170	170	170
PE (Algorithm 1)	4 1024	4 8	2158 2169	2955 2963	4015 4030	155 155	155 155	150 150
PE (Algorithm 2)	4 1024	16 32	8298 8315	11470 11491	15740 15759	140 140	140 140	138 138

TABLE IV
ALGORITHM CHARACTERISTICS

	Algorithm 1				Algorithm 2			
	No. of PEs	No. of BRAMs	Area (slices)	Speed (MHz)	No. of PEs	No. of BRAMs	Area (slices)	Speed (MHz)
Least	20	80	44001	100	5	40	41230	103
Moderate	14	56	42649	105	3	24	34697	110
Full	10	40	41157	105	2	16	31681	110



(a) Percentage of MAC Area Occupied by Control Unit vs. MAC Area

(b) PE Clock Speed vs. MAC Clock Speed

Fig. 13. Area and clock speed performance of PEs

Table IV presents the characteristics of our algorithms on XC2VP100 for 64-bit matrix multiplication. It shows the maximum number of PEs that can be configured on the FPGA and the maximum number of BRAMs required by the algorithms. Not surprisingly, the number of PEs that can be configured on the device decreases when the area of the floating-point units increases. For least-compliant floating-point units, the XC2VP100 can at most hold 20 PEs in Algorithm 1, or 5 PEs in Algorithm 2 for $r = 2$. The algorithms use no more than 80 BRAMs, whose total size is 144 KB. Besides the slices occupied by the PEs, the algorithms also take up additional slices that are used for the control logic and routing of the entire linear array. We call these slices “control logic area” of the algorithms. This area is less than 5% of the total area in both algorithms.

The scalability of our algorithms are investigated in Figure 14 and Figure 15. Figure 14(a) and Figure 15(a) show that the total area of both algorithms increases linearly as the number of PEs increases, regardless of the floating-point units used. However, when more PEs are configured on the FPGA, the achievable clock speed of the algorithms decreases due to the increased routing complexity. For least-compliant floating-point units, as the number of PEs increases from 2 to 20, the degradation in the achievable clock speed of Algorithm 1 is less than 35%, as shown in Figure

14(b). When the algorithm employs the other two types of floating-point units, the clock speed degrades faster because these units occupy larger area and more routing resources. As the routing is more complex in Algorithm 2, there is more degradation in clock speed with the increase in the number of PEs. With the least-compliant floating-point units, when the number of PEs increases from 1 to 5, the degradation in clock speed of Algorithm 2 is about 25%, as shown in Figure 15(b).

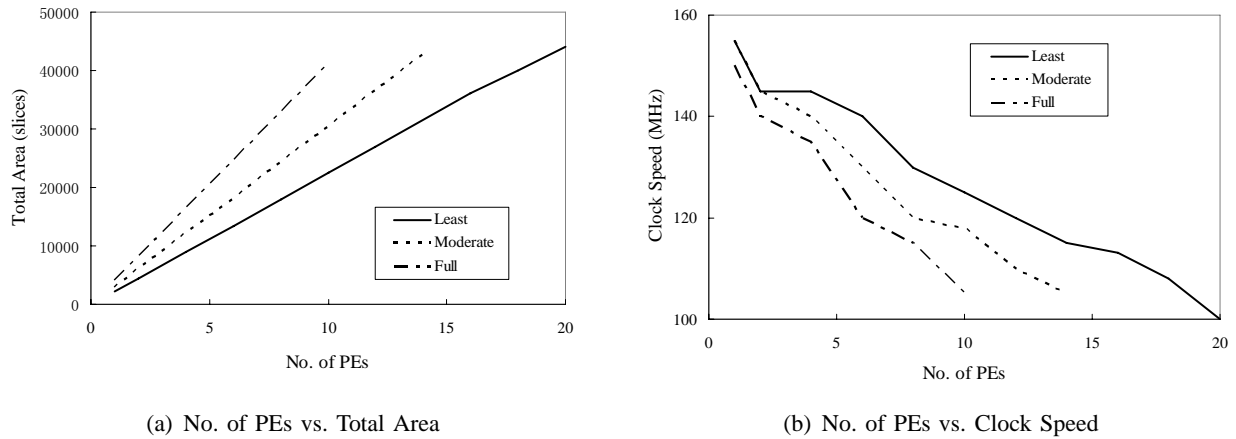


Fig. 14. Area and clock speed performance of Algorithm 1

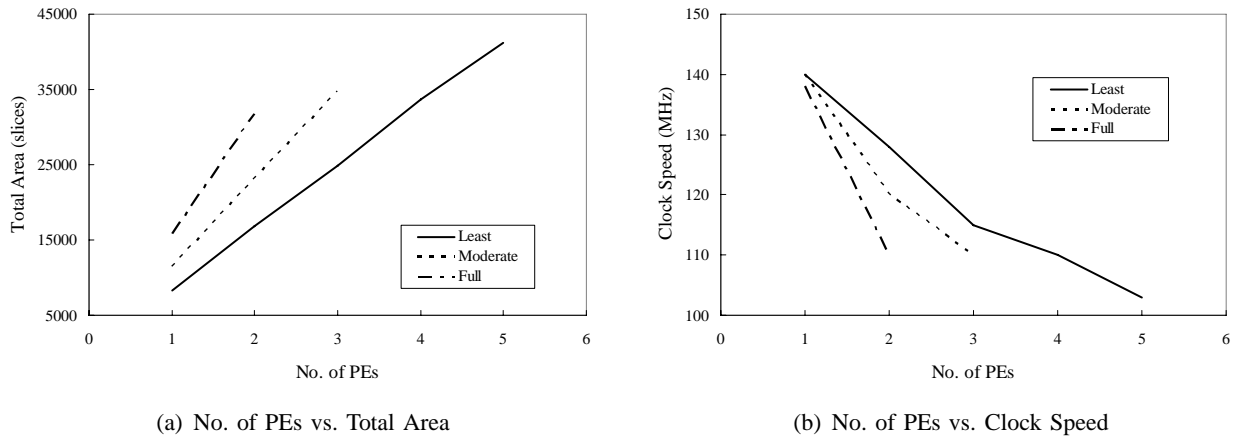


Fig. 15. Area and clock speed performance of Algorithm 2

Our designs can also be implemented using multiple FPGAs. Algorithm 1 needs 3 matrix elements during each clock cycle. Suppose the designs run at a clock speed of 170 MHz. For 64-bit matrix multiplication, the upper bound on the memory bandwidth required by Algorithm 1 is $64 \times 3 \times 170$ Mb/s = 32.6 Gb/s. If we connect multiple FPGAs in a linear array, the required communication bandwidth between the FPGAs is also 32.6 Gb/s. Since XC2VP100 has 1164 user I/O pins, it is able to support a pipeline throughput of 1164×170 Mb/s = 197.8 Gb/s. Thus, multiple FPGAs configured using Algorithm 1 can be connected through the user I/O pins. When $r = 2$, the upper bound on

the communication bandwidth required by Algorithm 2 is $2 \times 64 \times 3 \times 170 \text{ Mb/s} = 65.2 \text{ Gb/s}$ at the clock speed of 170 MHz. Therefore, it can also be implemented over multiple FPGAs using the user I/O pins. For larger values of r , the use of Rocket I/O Multi-Gigabit Transceivers or more advanced I/O technology may be necessary [8].

Although the above discussion is based on Algorithms 1 and 2, the conclusions also apply to Algorithm 3. In particular, the architecture of Algorithm 3 is also modular as well as scalable; the hardware resources required by the architecture increases linearly as the number of PEs increases.

We next examine the required memory bandwidth of Algorithm 3 when the total storage size M and the number of PEs p vary. In practice, M is usually determined by the size of available on-chip memory, while p is determined by the total area of the FPGA device. For simplicity, in our experiments, we vary the number of PEs p from 1 to 8, and vary the total storage size M from 16^2 words to 128^2 words. Note that

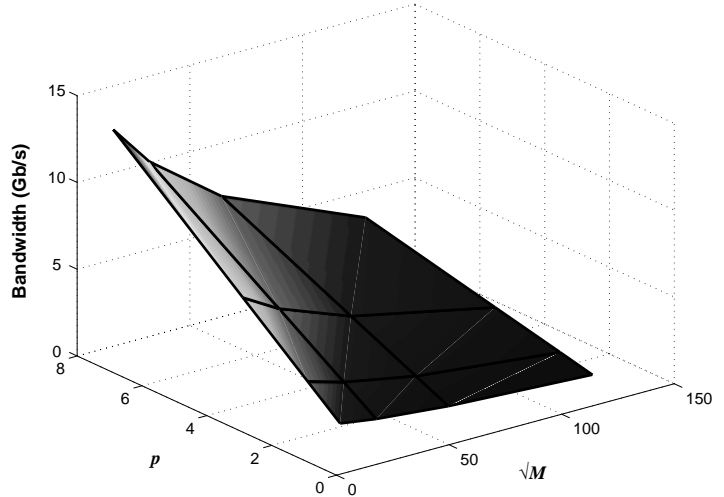


Fig. 16. Memory bandwidth required by Algorithm 3 vs. p , \sqrt{M}

the trend shown by the experimental results can also be applied to larger p and M . Algorithm 3 at most reads two and writes one matrix element every $\frac{p}{\sqrt{M}}$ clock cycles. The bandwidth for each point in Figure 16 is thus calculated as: $B = 3 \times 64 \times \frac{p}{\sqrt{M}} \times \text{clock speed}$.

Figure 16 shows that when the storage size is fixed, a larger p leads to a higher memory bandwidth requirement because more floating-point values are needed in each clock cycle. On the other hand, when the number of PEs is fixed, the larger is M , the less memory bandwidth is required.

B. Performance Comparison

We use MFLOPS and GFLOPS to measure the floating-point performance of our algorithms, as they are popular metrics for comparing floating-point performance. We first calculate the peak performance of our target device, and compare the sustained performance of the algorithms against it. In the following discussion, we use the least-compliant floating-point units unless stated otherwise.

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 $MFLOP_{peak} =$

$2 \times \text{the number of the MACs on the FPGA} \times \text{speed of the MAC} = 2 \times \text{area of FPGA} \times \frac{\text{speed of the MAC}}{\text{area of the MAC}}$.

Thus, the peak performance of XC2VP100 is 8.7 GFLOPS for 64-bit matrix multiplication.

We first look at the MFLOPS performance of Algorithm 1. Suppose the problem size is n , and $s = n$. Since matrices are fed into the FPGA consecutively, the cycles for filling the pipelines are not needed for most of the matrices. Therefore, the effective latency of Algorithm 1 is n^2 , and the sustained MFLOPS performance is calculated as: $MFLOP_1 = \frac{2n^3}{(\text{Effective latency of the algorithm}) \times (\text{clock period})} = 2n \times \text{clock speed}$. Note that, this equation only holds when $n \leq p_{max1}$, where p_{max1} is the maximum number of PEs that can be configured on the device. The effective latency of Algorithm 2 is $\frac{n^2}{r}$; the sustained MFLOPS performance of Algorithm 2 is thus calculated as: $MFLOP_2 = 2nr \times \text{clock speed}$. This equation holds when $\frac{n}{r^2} \leq p_{max2}$, where p_{max2} is the maximum number of PEs of Algorithm 2 that can be configured on the device. The XC2VP100 FPGA holds at most 20 PEs in Algorithm 1 or 5 PEs in Algorithm 2 for $r = 2$. Therefore, in our experiments, we set $n = 20$ for Algorithm 1 and Algorithm 2.

When $n > p_{max1}$, we divide the source matrices into blocks, and use Algorithm 3 to perform block matrix multiplication. In this case, the effective latency is increased to $\Theta(\frac{n^3}{p})$, where p is the number of PEs and $p \leq p_{max1}$. In this case, the sustained performance of Algorithm 3 is: $MFLOP_3 = 2p \times \text{clock speed}$. Thus, the MFLOPS performance of Algorithm 3 is not dependent on the problem size n , but on the number of PEs configured on the FPGA and the clock speed of the design. In our experiments, we set $n = 1024$ for Algorithm 3 due to the size constraint of the external SRAM memory. However, the algorithm achieves the same performance for other problem sizes.

Experiments show that Algorithm 1 and Algorithm 2 achieve sustained performance of about 4 GFLOPS when $n = 20$, a little less than 50% of the peak performance of the FPGA device. Note that we did not utilize manual placement or any other optimizations; otherwise, our designs could have achieved even higher GFLOPS performance. The current performance of our FPGA-based designs compares favorably with general-purpose processors. For example, a Xeon processor-based platform at 3.2 GHz with 1 MB L3 cache only achieves 5.5 GFLOPS performing 64-bit matrix multiplication, while a 3-GHz Pentium 4 processor with 512 KB L2 cache reaches 5.0 GFLOPS [22]. These numbers are obtained when the processors were executing Intel Math Kernel Library, which is optimized to exploit the best performance of the Intel processors [22].

We next compare our algorithms against other works on floating-point matrix multiplication on FPGAs. Our algorithms achieve higher performance as they support more parallel operations during one clock cycle. This is illustrated when we compare against the designs in [17] and [16]. For a clock speed of 170 MHz and with 20 MACs, the sustained performance of [17] and [16] is 1.23 GFLOPS and 0.94 GFLOPS, respectively. Our design also achieves higher performance under the given storage size and memory bandwidth. The architecture in [1] achieves 4 GFLOPS of peak floating-point capability with 512 MB/s of memory bandwidth and 192 KB of on-chip memory in the FPGA. In contrast, our Algorithm 3 achieves 6.7 GFLOPS with the same storage size and the same memory bandwidth requirement.

We also compare our designs against the implementation compiled by the Celoxica Handel-C development tool [23]. Handel-C is a high-level language based on ANSI-C that can generate FPGA hardware designs. For researchers designing high-level algorithms for FPGA-based computers, Handel-C is more convenient to use than VHDL, since most of the hardware design and optimization is

handled by the tool. To investigate the performance gap between the design generated by the tool and our design, we wrote a Handel-C floating-point matrix multiplication program and compare its performance with our algorithms. In the Handel-C code, we used the floating-point library provided by Celoxica [23] to perform the floating-point computations. We used embedded multipliers for implementing floating-point multipliers, and further optimized the program using inner loop parallelization. The Handel-C code was compiled by Celoxica DK1 Design Suite V1.1. Table V shows the area and speed comparison between our algorithms and Handel-C based design, for 8×8 64-bit matrix multiplication. We see that the Handel-C based design takes more slices than our algorithms, while its latency is much longer.

C. Experiments on Cray XD1

To illustrate our ideas, we implemented Algorithm 3 on one compute blade in Cray XD1 [24] (The architectural description for XD1 is in Section II). As XC2VP50 only contains 23616 slices, $p = 8$.

TABLE V
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
Algorithm 1	18029	130	0.5
Algorithm 2	16797	128	0.25

M is determined by the size of SRAM, and is 1024^2 . Our design achieved a sustained performance of 2.06 GFLOPS when $n = 1024$. In XD1, six compute blades fit into one chassis. We project that when all the six FPGAs in a chassis are used, the sustained performance of our design would be 12.4 GFLOPS. The required SRAM bandwidth by the design is 2.1 GB/s, and the required DRAM bandwidth equals 73.1 MB/s. These bandwidth requirements are much smaller than the available bandwidth in XD1.

As we have discussed above, our designs are modular so that improved floating-point MACs can be easily plugged in. Moreover, if the performance of the MAC is improved, the performance of the designs will improve accordingly. The PE used in our experiments occupies more than 2000 slices and runs at 155 MHz. Therefore, we project the performance of Algorithm 3 when the area of PE ranges from 1600 to 2000 slices, and the clock speed increases from 160 MHz to 200 MHz.

Figure 17(a) shows the projected sustained performance of one chassis, as a function of the area and the clock speed of the PE. We calculate the MFLOPS performance using equation: $2 \times \text{number of PEs on the device} \times \text{clock speed of the PE} \times 6$. 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 up to 27 GFLOPS.

The performance of our design also depends on the device used. With a larger device, more PEs can be configured and higher performance can be achieved. Figure 17(b) shows the projected sustained performance of our design using a chassis if XC2VP100 were to be used in XD1. As XC2VP100 contains about twice as many slices as XC2VP50, its performance is also about twice as that of XC2VP50.

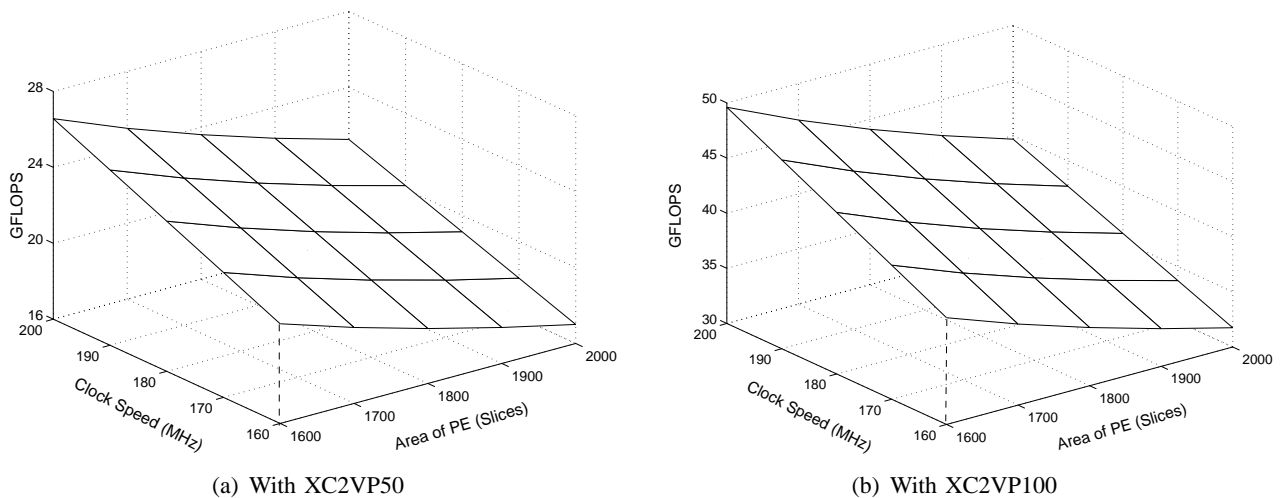


Fig. 17. Projected Sustained Performance of Matrix Multiply Design Using a Chassis

VII. CONCLUSION

We have used floating-point matrix multiplication as an example to illustrate the capability of reconfigurable computing systems in floating-point applications. We analyzed the advantages and design tradeoffs in FPGA-based floating-point matrix multiplication. Based on the analysis, we identified four different cases for various values of the number of PEs p and the required storage size M . We have proposed three algorithms for cases 1 and 4. Algorithm 1 and Algorithm 2 achieve optimal latency of $\Theta(n^2)$ for $n \times n$ matrix multiplication. However, in these two algorithms, the number of PEs and the required storage size increase with the problem size n . These algorithms are suitable for cases where $p \geq n$ and $M = n^2$. In Algorithm 3, the number of PEs is determined by the available resources and the storage size is determined by the size of on-chip memory on the FPGA. Therefore, p can be less than n and M can be less than n^2 . With the given resources, Algorithm 3 achieves optimal latency and minimizes the required memory bandwidth. The design of the PEs in our algorithms are oblivious to the actual implementation of the floating-point units. Any area or speed improvement of these units results in linear performance improvement in our algorithms. In all the algorithms, with the increase of the available hardware resources, the number of PEs that can be configured increases linearly, while the achievable clock speed decreases moderately. Our algorithms are suitable for matrix multiplication of various sizes and achieve floating-point performance comparable with general-purpose processors.

REFERENCES

- [1] K. D. Underwood and K. S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," in *Proc. of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.
- [2] T. A. El-Ghazawi, K. Gaj, N. A. Alexandridis, A. Michalski, O. D. Fidanci, M. Taher, E. El-Araby, E. Chitalwala, and P. Saha, "Reconfigurable computers: an empirical analysis," in *Proc. of the 13th ACM International Symposium on Field-Programmable Gate Arrays*, February 2005.
- [3] SRC Computers, Inc., "<http://www.srccomp.com/>."
- [4] Cray Inc., "<http://www.cray.com/>."
- [5] Starbridge Hypercomputers, "<http://www.starbridgesystems.com/products/hardware.html>."
- [6] 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*, December 2002.

- [7] L. Zhuo and V. K. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs," in *Proc. of the 18th International Parallel & Distributed Processing Symposium*, April 2004.
- [8] Xilinx Incorporated, "<http://www.xilinx.com/>."
- [9] Institute of Electrical and Electronics Engineers, *IEEE 754 Standard for Binary Floating-Point Arithmetic*. IEEE, 1984.
- [10] L. E. Cannon, "A Cellular Computer to Implement the Kalman Filter Algorithm," Ph.D. dissertation, Montana State University, 1969.
- [11] G. C. Fox and S. W. Otto, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing*, vol. 4, pp. 17–31, 1987.
- [12] J. Choi, J. J. Dongarra, and D. W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, October 1994.
- [13] R. A. V. D. Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.
- [14] K. Q. Li and V. Y. Pan, "Parallel Matrix Multiplication on a Linear Array with a Reconfigurable Pipelined Bus System," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 519–525, 2001.
- [15] A. Amira and F. Bensaali, "An FPGA based Parameterisable System for Matrix Product Implementation," in *Proc. of The IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS2002)*, October 2002.
- [16] 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*, April 1998.
- [17] 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*, September 2000.
- [18] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev, "64-bit Floating-Point FPGA Matrix Multiplication," in *Proc. of the 13th International Symposium on Field Programmable Gate Arrays*, February 2005.
- [19] J. Hong and H. Kung, "I/O Complexity: The Red Blue Pebble Game," in *Proc. of ACM Symposium on Theory of Computing*, May 1981.
- [20] G. Govindu, R. Scrofano, and V. K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," in *Proc. of International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2005.
- [21] Mentor Graphics Corp., "<http://www.mentor.com/>."
- [22] Intel, <http://www.intel.com>.
- [23] Celoxica Company, "<http://www.celoxica.com/>."
- [24] L. Zhuo and V. K. Prasanna, "High Performance Linear Algebra Operations on Reconfigurable Systems," in *Proc. of SuperComputing 2005*, November 2005.