# 64-bit Floating-Point FPGA Matrix Multiplication

Yong Dou
National Laboratory for Parallel and Distributed Processing
Changsha, P.R.China, 410073
yongdou@163.net

S. Vassiliadis, G. K. Kuzmanov, G. N. Gaydadjiev
Computer Engineering,
EEMCS, TU Delft,
P.O. Box 5031, 2600 GA Delft, The Netherlands
http://ce.et.tudelft.nl
{S.Vassiliadis,G.Kuzmanov,G.N.Gaydadjiev}@EWI.TUDelft.NL

## ABSTRACT

We introduce a 64-bit ANSI/IEEE Std 754-1985 floating point design of a hardware matrix multiplier optimized for FPGA implementations. A general block matrix multiplication algorithm, applicable for an arbitrary matrix size is proposed. The algorithm potentially enables optimum performance by exploiting the data locality and reusability incurred by the general matrix multiplication scheme and considering the limitations of the I/O bandwidth and the local storage volume. We implement a scalable linear array of processing elements (PE) supporting the proposed algorithm in the Xilinx Virtex II Pro technology. Synthesis results confirm a superior performance-area ratio compared to related recent works. Assuming the same FPGA chip, the same amount of local memory, and the same I/O bandwidth, our design outperforms related proposals by at least 1.7X and up to 18X consuming the least reconfigurable resources. A total of 39 PEs can be integrated into the xc2vp125-7 FPGA, reaching performance of, e.g., 15.6 GFLOPS with 1600 KB local memory and 400 MB/s external memory bandwidth.

## Categories and Subject Descriptors

B.2.4 [**Arithmetic and Logic Structures**]: High-Speed Arithmetic; C.1.3 [**Other Architecture Style**]: Adaptable architectures; F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices

## General Terms

Algorithms, Design, Performance

## Keywords

Matrix multiplication, Floating-point, FPGA

## 1. INTRODUCTION

A broad range of complex scientific applications strongly depend on the performance of the floating-point matrix multiplication kernel. The LINPACK Benchmark has been used for over 20 years to evaluate high performance computers designed to run such complex applications. LINPACK includes Basic Linear Algebra Subprograms [3] (BLAS) which are high quality "building block" routines performing basic vector and matrix operations. The so-called Level 3 BLAS target matrix-matrix operations of order $O(n^3)$. It has been shown that high-performance Level-3 BLAS could be made portable by representing these operations by matrix multiplications.

Various methods for implementing the matrix multiplication algorithm, exploiting the specific features of different computer systems, e.g., distributed memory or hierarchical shared memories, have been considered [2, 5, 7, 20]. In this paper, our primary goal is to propose a general hardware solution of the floating-point matrix multiplication problem. More specifically, the contributions of our work are:

- We propose a general block matrix multiplication algorithm, applicable for arbitrary matrix sizes. The matrices are scheduled in streams and the results are generated in blocks so that the data are reused and localized. The algorithm considers practical hardware limitations in terms of local storage, I/O bandwidth, and computational logic.
- We introduce a scalable linear array of processing elements implementing the proposed multiplication algorithm and map this organization into Xilinx Virtex II Pro FPGAs. The design is pipelined and the number of the pipeline stages is minimized after a careful analysis of the design trade-offs in the context of the considered FPGA technology. Synthesis results indicate that a linear array implementation requires less local storage than recent related proposals. Furthermore, assuming the same amount of local memory and the same I/O bandwidth, our design outperforms the related art considered by at least 1.7X and up to 18X.
- Synthesis results suggest that a single processing element requires an area of 1401 Virtex II Pro slices and can run at 200MHz. A total of 39 PEs can be integrated into the xc2vp125-7 FPGA, which can reach performance of, e.g., 15.6 GFLOPS with 1600 KB local memory and 400 MB/s external memory bandwidth.

The remainder of this paper is organized as follows. In Section 2, some brief background on floating point matrix multiplication is presented. The design, proposed in this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*FPGA'05,* February 20–22, 2005, Monterey, California, USA.
Copyright 2005 ACM 1-59593-029-9/05/0002 ...$5.00.

paper, is extensively described and analyzed in Section 3. Section 4 introduces an FPGA implementation of our proposal, including some synthesis and performance data. Our FPGA prototype implementation is compared to other related works in Section 5 and the discussion is finally concluded in Section 6.

## 2. BACKGROUND

In general, the standard matrix multiplication[1] $C = A \times B$ is defined as follows:

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}, (0 \leq i < M, 0 \leq j < R),$$

Where A, B, and C are $M \times N$, $N \times R$, and $M \times R$ matrices, respectively. The computations from the above definition can be described by a straightforward algorithm with the following pseudocode:

```
for (i = 0; i < M; i = i + 1)
    for (j = 0; j < R; j = j + 1){
        C[i, j] = 0;
        for (k = 0; k < N; k = k + 1)
            C[i, j] = C[i, j] + A[i, k] * B[k, j];}
```

The positions of the three nested loops in the algorithm can be exchanged. With respect to the position of loop k, the matrix multiplication is referred to as *internal* product, *middle* product and *external* product. The computational complexity[2] of the above algorithm is $2 \times M \times R \times N$, i.e., $O(n^3)$, and requires $M \times N + R \times N + 2 \times M \times R$ memory accesses, i.e., $O(n^2)$. As a rule, parallel processing reduces the computational complexity but increases the demands to the data memory bandwidth. The result is that data communication complexity increases dramatically at a given limited bandwidth. Another practical limitation to the efficient multiplication of matrices is the limited amount of local memory utilized for storing intermediate results. The last issue is becoming extremely severe for large matrices. An efficient general approach to overcome both the memory communication bottleneck (due to the limited bandwidth) and the local storage limitation (both for sequential and parallel machines) is to exploit *data reusability.* This can be done, for instance, by partitioning the matrix data into smaller sub-matrices and processing each of these partitions. Such an approach, where the matrix data in the resulting matrix is generated into portions of dense rectangular sub-blocks, is referred to as *block matrix multiplication.* The pseudocode in Figure 1 describes a sequential algorithm for block matrix multiplication. This algorithm assumes that the resulting $M \times R$ matrix data are obtained sequentially in consecutive $S_i \times S_j$ sub-blocks.

In our work, we consider floating-point matrix multiplication, more specifically the ANSI/IEEE Std 754-1985 floating-point standard. This standard specifies formats for 32-bit and 64-bit representations. More specifically, we consider the design issues of 64-bit floating-point numbers, which are more time-consuming and area-demanding than that of 32-bit floating-point numbers. The standard defines 4 types of numbers: normal, infinity, NaN (not a number), and denormal. In our design, we do not support denormal numbers.

[1]Depending on the operands, we refer to the × operator either as to a matrix multiplication, or as to a scalar multiplication.

[2]Here, by complexity we mean time complexity.

**Figure 1: Sequential block matrix multiplication.**

```
for (i = 0; i < M/S_i; i = i + 1)
    for (j = 0; j < R/S_j; j = j + 1){
        for (L_i = 0; L_i < S_i; L_i = L_i + 1)
            for (L_j = 0; L_j < S_j; L_j = L_j + 1)
                C[i · S_i + Li, j · S_j + L_j] = 0;
        for (k = 0; k < N; k = k + 1){
            for (L_i = 0; L_i < S_i; L_i = L_i + 1)
                for (L_j = 0; L_j < S_j; L_j = L_j + 1)
                    C[i · S_i + Li, j · S_j + L_j] =
                    = C[i · S_i + Li, j · S_j + L_j]+
                    +A[i · S_i + L_i, k] × B[k, j · S_j + L_j];}}
```

## 3. THE PROPOSED DESIGN

In this section, we introduce a parallel block-scheduling computational scheme, the supporting parallel algorithm, and present a linear array architecture to implement the matrix multiplication.

### 3.1 Parallel Block-Scheduling

We begin by noting that the proposed algorithms to follow can be applied to any matrix size and sub-block dimensions. For simplicity of discussion and without loss of generality, to describe the approach, we consider square matrix multiplications and the matrices dimensions to be multiples of the sub-block dimensions.

**The parallel block (PB) algorithm:** We separate the sequential block algorithm (see Figure 1) into two parallel algorithms, called Master and Slave as illustrated in Figure 2. The Master algorithm is executed on a single processor, and the Slave - on multiple processors. In the discussion to follow, we refer to the Slave processors as to Processing Elements (PE). The Master sends the data from matrices A and B within messages and loads the results into matrix C ordered as $S_i$ by $S_j$ blocks, according to the scheduling algorithm. The data in the messages are correctly ordered by the Master and are delivered in a preserved order to the PE chain. The algorithms in Figure 2 are described in a Single Program Multiple Data (SPMD) computational model, augmented with synchronous interprocess message passing primitives and parallel tasking primitives. The primitive Fork, spawns parallel tasks to be executed in the designated processor. It is defined as:

Fork (F, P )- spawn a process in processor P to execute the program or function F.

Synchronous message passing primitives Send and Receive are defined as:

Send (P, Q, X )- send value of X to message Q in PE P;

Rcv (P, Q, X )- receive a message from message Q in PE P and place in the data variable X.

Memory access primitives Store and Load are defined as:

Store (Y, X)- write values of X to the address Y;

Load (Y, X)- read values of X from the address Y.

TA and TB are local variables for the Slave program. The variable TC is a two dimensional array of $S_i/P$ rows and $S_j$ columns. TC can be accessed by the Master program through the Store or Load primitives.

**Figure 2: Proposed parallel block matrix multiplication.**

**Master algorithm:**

```
Master (Mpid=0){
   for (pid=1; pid≤ P; pid=pid+1) Fork (Slave(pid),pid);
   for (i = 0; i < N/S_i; i = i + 1)
      for (j = 0; j < N/S_j; j = j + 1){
         Store(TC[0 : S_i − 1, 0 : S_j − 1],0);
            for (k = 0; k < N; k = k + 1){
            Send (Mpid+1,FIFOA,A[i : i + S_i − 1, k]);
            for (L_j = 0; L_j < S_j; L_j = L_j + 1)
               Send (Mpid+1,FIFOB,B[k, j ∗ S_j + L_j]);
            }
            Load (C[i ∗ S_i : i ∗ S_i + S_i − 1, j ∗ S_j : j ∗ S_j + S_j − 1], TC[0 : S_i − 1, 0 : S_j − 1]);
      }
}
```

**Slave algorithm:**

```
Slave(pid){
   Rcv (pid,FIFOA,A[0 : S_i − 1]);
   Send (pid+1,FIFOA,A[0 : S_i − 1]);
   TA[0 : S_i/P − 1]=A[(pid-1)∗S_i:pid∗S_i − 1];
   for (L_j = 0; L_j < S_j; L_j = L_j + 1){
      Rcv (pid,FIFOB,TB);
      Send (pid+1,FIFOB,TB);
      for (L = 0; L < S_i/P; L = L + 1)
         TC[L_j, L]=TC[L_j, L]+TA[L]*TB;
   }
}
```

In each PE, the scheduling algorithm is performed in the following steps:

**Step 1:** The Master processor sends $S_i$ elements of one column of array A so that each PE receives $Si/P$ elements.

**Step 2:** The Master processor sends $S_j$ elements of one row of array B to all PEs. The elements of array A and B are multiplied in each PE and added to the corresponding temporary elements of array C. Results are accumulated into the local PE memory.

**Step 3:** Repeat N times steps 1 and 2. Finally, the PE local memories will contain $S_i \times S_j$ elements of C.

**Step 4:** The Master processor transfers the $S_i \times S_j$ block of C from the PE local memories to the main memory. If there are unprocessed blocks, go to step 1.

**Computational scheme example:** In the following, we explain through an example, how the computational scheme of the PB algorithm operates. The left-hand-side of Figure 3 illustrates the general block matrix multiplication (recall the algorithm from Figure 1). The upper-left $S_i \times S_j$ block of C is a product of the upper $S_i$ rows of A and the left-most $S_j$ columns of B. An illustrative example of how the proposed parallel block matrix multiplication works is depicted in the right-hand-side of Figure 3. The example assumes $4 \times 4$ matrices ($N = 4$), result matrix partitioning into $2 \times 2$ sub-blocks ($S_i = S_j = 2$), and 2 processing elements. To generate a $2 \times 2$ block of C, the data from two entire rows of A and two entire columns of B are required. The bottom-right scheme illustrates the computations for one sub-block (the upper-left block) of matrix C. The sub-block is initialized with data $c^0_{xx}$ first. Then the first (or last)[3] pair of data from matrix A (a14, a24) and the first (respectively last) pair of the two B columns (b41, b42) are loaded into PE0 and PE1 and added to the contents of $c^0_{xx}$. This is essentially a MAC operation, which produces an intermediate result, denoted as the $c^1_{xx}$ sub-matrix in Figure 3. Similarly, the remaining data of the two A rows and the two B columns are streamed to the two processing elements in pairs, producing a series of intermediate results ($c^2_{xx}$ and $c^3_{xx}$). Thus, after N=4 steps, the correct final result for the c11, c12, c21, c22 is calculated by the PEs and is ready to be loaded by the master processor. PE0 and PE1 continue computing the next $2 \times 2$ blocks until the entire matrix C is calculated.

## 3.2 The Linear Array Organization

We propose a linear array of PEs to implement the PB matrix multiplication algorithm. If the bandwidth is large enough, multiple linear arrays can work in parallel to compute different blocks of the matrix. In most cases, however, the bandwidth is a limiting factor, therefore, we focus on these cases with bandwidth limitations.

**Topology and operation:** Generally speaking, the linear array is a computing engine controlled by a host processor. The interface between the host and the array can be implemented by a bus or by a Direct Memory Access (DMA) controller, as illustrated in Figure 4. The array has a very simple interconnect topology. Within the bus interface or the DMA, there are 3 FIFOs to interface with the first PE of the array. Strictly identical interface interconnects all

[3]The order data are loaded into the PE may vary.

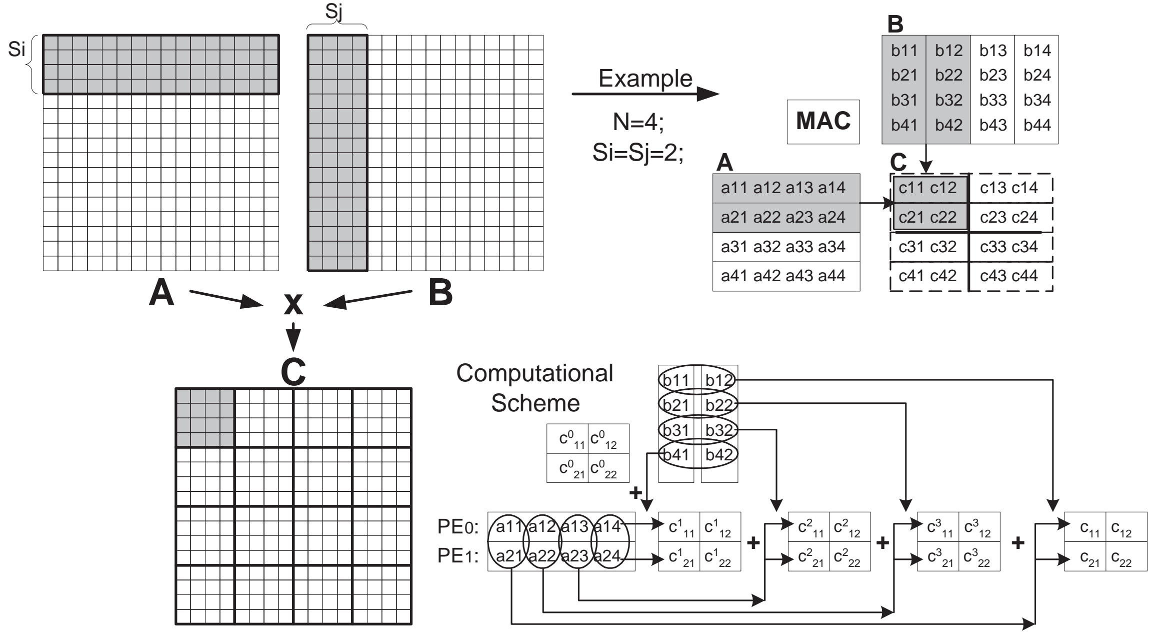
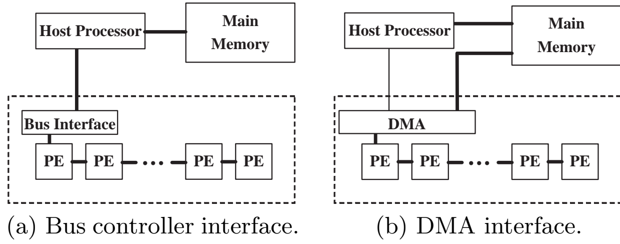
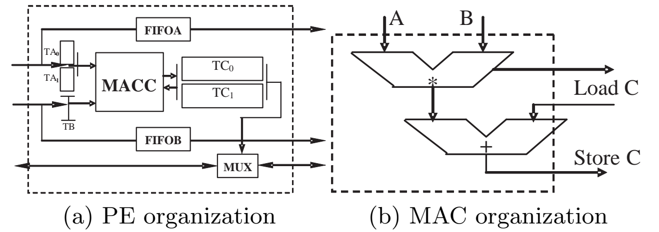



**Figure 3: The PB computational scheme - an example for $N = 4$ and $S_i = S_j = 2$.**




(a) Bus controller interface. (b) DMA interface.

**Figure 4: PE array interface and general topology.**




(a) PE organization (b) MAC organization

**Figure 5: The structure of the PE and the MAC.**

PEs in the array, as well. In Figure 4(a), the host processor alone performs the data transfers between the array and the main memory through a standard bus. In Figure 4(b), a DMA controller connects the array with the main memory. The DMA controls the execution of multiple level loops, generates the data matrix addresses, and performs the data transfers while the host only monitors these operations. The host sends matrix A and B into dedicated FIFOs (according to the aforeproposed scheduling algorithms), initializes, and loads the local PE memories through dedicated memory channels. Whenever there are data in the FIFOs, the array processes them and stores the results in the local memories. The results of the matrix multiplication are loaded from the PE local memories to the host, while the following computations are running in a pipeline manner.

**PE organization:** Each PE is designed as a pipeline and comprises two sets of data registers, two FIFOs, one MAC unit, S words of local storage, and control logic, as depicted in Figure 5(a). The first stage of the pipeline is built of two data register files ($TA_0$, $TA_1$) which fetch the elements of one matrix (e.g., A) from the preceding PE. Each element of A is reused $S_j$ times and each of the register files is designed to store $\frac{S_i}{P}$ elements of A. In order to keep the pipeline busy, the two register files are switched alternatively, i.e., at any moment one is involved in the computations while the other is receiving new data. The elements of B are fetched into a single register and since they are not reused, we do not need to store them locally after each computation. The fetched elements of both A and B are also queued in local FIFOs (FIFOA and FIFOB) to be forwarded to the next PE. During the second pipeline stage, the fetched elements of A and B, and one temporary element of C from the local memory are sent to the MAC unit pipeline (details follow) to perform a multiply and accumulate operation. Finally, the results of the MAC operation are stored in the local memory, which is organized in two banks ($TC_0$, $TC_1$) and operates similarly to the alternatively switching TA register files.

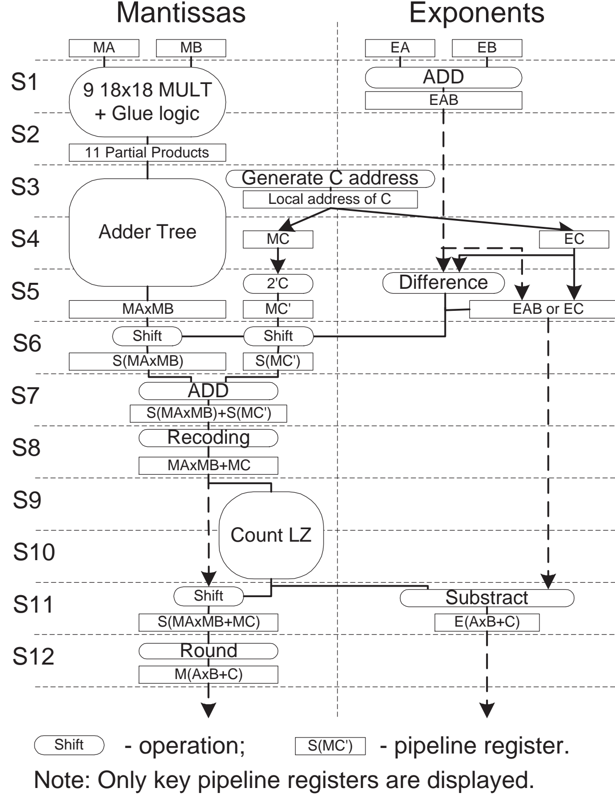



**Figure 6: The MAC unit pipeline.**

**Local storage requirements:** As stated in the descriptions of the PB algorithm and of the PE organization, $2 \times S_i$ memory elements are required for storing the A columns, and $2 \times S_i \times S_j$ - for intermediate results of an $S_i \times S_j$ block of C. Therefore, the minimal requirements to the local storage are for $2 \times S_i \times S_j + 2 \times S_i$ data elements.

**MAC unit organization:** We propose a MAC unit structure that couples the multiplication and the accumulation closely, sketched in Figure 5(b). The multiplier receives the elements of A and B in a data-driven manner. That means whenever both of the data are available, they will enter the pipeline. During the multiplication phase, a load address is generated to prefetch the proper C element from the local storage. Before the partial result of the $A \times B$ multiplication is ready, the C element is available and some pre-processing computations have been done (to be explained later). The results from the addition phase are stored (accumulated) in the local memory. Figure 6 illustrates the MAC unit pipeline stage partitioning. In more details, the stages are:

**S(1,2):** *Mantissas:* Decompose each of the two 53-bit mantissas (MA and MB), so that both 51 LS (Least Significant) bit partitions are sent to nine $18 \times 18$ multipliers and the rest 2 MS (Most Significant) bits of each mantissa generate 4 partial products trough a glue logic (or LUT). For a better pipeline balance and throughput, we assume double staged multipliers. As a result, 11 partial products are propagated to the next pipeline stage.
*Exponents:* The two exponents (EA and EB) are added in S1 and the result (EAB) is propagated trough S2.

**S(3,4,5):**[4] *Mantissas:* The 11 partial products are summed through a three-level adder tree, assuming three pipeline stages. In S3, a local memory address is generated and the corresponding C element is available in S4. In S5, the mantissa of C (MC) is coded into two's complement code ($MC'$).
*Exponents:* EAB is propagated trough S3 and S4. In S5, the difference between EA and the exponent of C (EC) is calculated. Based on the difference result, the greater of EAB or EC is propagated further through the pipeline.

**S6:** *Mantissas:* The mantissa with the minor exponent is shifted with respect to the exponent difference from S5, resulting in S(MAxMB) or S($MC'$).
*Exponents:* Propagated.

**S7:** *Mantissas:* The shifted mantissas from S6 (S(MAxMB) and S($MC'$)) are summed in two's complement notation.
*Exponents:* Propagated.

**S8:** *Mantissas:* Recoded from two's complement notation to sign-magnitude.
*Exponents:* Propagated.

**S(9,10):** *Mantissas:* Propagated. Counting leading zeros in the 53 mantissa bits takes two stages. In S9, the 53 mantissa bits are decomposed into 4 groups to obtain 4 numbers of zeros. In S10, the 4 numbers are added to generate the total number of leading zeros.
*Exponents:* Propagated.

**S11:** *Mantissas:* Shifted to the left according to the number of the leading zeros (result denoted as S(MAxMB+MC)).

[4]In an ASIC design, these three stages may be reduced to two, but for an FPGA design three stages are optimal due to the specific adder tree implementation.

*Exponents:* The new exponent of C (E(AxB+C)) is calculated by subtracting the number of leading zeros from the exponent of the previous stage, and is propagated to the final stage S12.
**S12:** *Mantissas:* Rounded to the nearest even. The new mantissa of C (M(AxB+C)) is available.
*Exponents:* Propagated to the output.

### 3.3 Performance Analysis

To analyze the performance potentials of the proposed design, assume square matrices and the following notations: N - dimensions of the square matrices; BW - bandwidth between the main memory and the linear array [words]/[s]; PEAK - overall performance of the array [FLOPS].

Considering limited memory bandwidth and large matrices, the overall processing time of our pipelined design is determined by the data transfers between the main memory and the array. The amount of data that must be moved to and from the main memory to compute one $S_i \times S_j$ block of C is $(S_i \times N + S_j \times N + 2 \times S_i \times S_j)$, comprising: $S_i \times N$ elements of A; $S_j \times N$ elements of B; and $2 \times S_i \times S_j$ elements of C. For the entire matrix C, the amount of data in number of [words] transferred is:

$$(S_i \times N + S_j \times N + 2 \times S_i \times S_j) \times \frac{N}{S_i} \times \frac{N}{S_j} = \frac{N^3}{S_j} + \frac{N^3}{S_i} + 2 \times N^2.$$

The time for transferring these data is: $\frac{\frac{N^3}{S_j} + \frac{N^3}{S_i} + 2 \times N^2}{BW}$, [s]. Recall that the computational complexity of the matrix multiplication is $2 \times M \times R \times N$ (see Section 2). Considering square matrices, $2 \times N^3$ floating point (FP) operations are required to multiply two $N \times N$ matrices. Therefore, regarding the overall performance, we consider the following equation:

$$PERF = \frac{\#FP_operations}{processing_time} = \frac{2 \times BW \times N^3}{\frac{N^3}{S_i} + \frac{N^3}{S_j} + 2 \times N^2} =$$

$$= \frac{2 \times BW}{\frac{1}{S_i} + \frac{1}{S_j} + \frac{2}{N}}, \quad [FLOPS]. \qquad (1)$$

**Maximum performance:** We note, that (1) is general and may support the performance estimation of other pipelined matrix multipliers. Therefore, we introduce some additional notations, to meet the specific organization of our proposal, namely: L - local memory size in (64-bit) [words]; P - number of PEs in the linear array; and F - clock frequency of the array in [Hz]. The following theorem holds:

THEOREM 1. *Consider the multiplication of two $N \times N$ matrices employing the PB matrix multiplication algorithm and the proposed design. Further consider that the result is produced in portions of $Si \times Sj$ blocks by a linear array of P processors with a limited main memory bandwidth. Given the fixed size of the local memory is L [words], the condition for the array to achieve the maximal performance is:*

$$S_i = \frac{L}{2 + \sqrt{2L}} \quad and \quad S_j = \sqrt{\frac{L}{2}}.$$

PROOF. According to (1), the performance is a function of the variables $S_i$ and $S_j$. Consider the following function:

$$f(S_i, S_j) = \frac{1}{S_i} + \frac{1}{S_j} + \frac{2}{N}; \qquad S_i, S_j \in \mathbb{N}. \qquad (2)$$

Assuming the proposed design and a fixed local memory size of L [words] (L$\in \mathbb{N}$), the following equations hold:

$$L = 2 \times S_i \times S_j + 2 \times S_i \quad \Leftrightarrow$$

$$S_j = \frac{L - 2 \times S_i}{2 \times S_i} \qquad (3)$$

Thus, function $f(S_i, S_j)$ is equivalent to the following function $F(S_i)$:

$$F(S_i) = \frac{2 \times Si}{L - 2 \times S_i} + \frac{1}{S_i} + \frac{1}{N}; \qquad L, S_i \in \mathbb{N}. \qquad (4)$$

The condition for the maximal performance is the same as that of the minimal $f(S_i, S_j)$. Therefore, we equalize the derivative of $F(S_i)$ to zero:

$$F'(S_i) = \frac{(2 \times L - 4) \times S_i^2 + 4 \times L \times S_i - L^2}{(-2 \times S_i^2 + L \times S_i)^2} = 0 \qquad (5)$$

Given $F'(S_i) = 0$, the following holds:

$$\left\| \begin{array}{l} (2 \times L - 4) \times S_i^2 + 4 \times L \times S_i - L^2 = 0 \\ -2 \times S_i^2 + L \times S_i \neq 0 \end{array} \right. \qquad (6)$$

Solving (6) for $L, S_i \in \mathbb{N}$, gives a single value of $S_i$ for the maximum performance and according to (3), the value of $S_j$ is calculated. The derived equations are:

$$S_i = \frac{L}{2 + \sqrt{2L}} \quad and \quad S_j = \sqrt{\frac{L}{2}}.$$

Indeed, the theorem statement holds. □

Consider the maximum performance conditions from Theorem 1 and substitute in (1), then:

$$PERF_{MAX} = \frac{BW}{\frac{1}{L} + \sqrt{\frac{2}{L}} + \frac{1}{N}}, \quad [FLOPS]. \qquad (7)$$

In practice, the fixed local memory size is in the order of kilobytes and the matrices are large, i.e., $L >> 1$ and $N >> L$. Considering these boundary conditions, we can conclude:

$$PERF'_{MAX} = \lim_{L,N>>1} \frac{BW}{\frac{1}{L} + \sqrt{\frac{2}{L}} + \frac{1}{N}} \quad \Leftrightarrow$$

$$PERF'_{MAX} = BW \times \sqrt{\frac{L}{2}} \quad , \quad [FLOPS]. \qquad (8)$$

**Peak performance bandwidth:** Equation (8) gives the maximum performance for a limited bandwidth BW, fixed memory size L, and large matrices. In the realistic case of limited number of processing elements, however, too large bandwidth will not contribute to any performance improvement. *The saturation point, beyond which the bandwidth does not affect the performance, we denote as the point of **peak performance***, which is:

$$PERF_{PEAK} = 2 \times P \times F, \qquad [FLOPS]. \qquad (9)$$

Equation (9) is quite intuitive assuming $P$ processing elements, working at frequency $F$, each of them performing two FP operations per cycle. Equalizing (8) and (9), we derive the ***peak main memory bandwidth*** *required for the*

**Table 1: Synthesis Data for a Single PE.**

| Xilinx V2P125 -6/-7 | MUL | ADD | Control | Total |
|---|---|---|---|---|
| Operand Width | 64 | 64 | 64 | 64 |
| Pipeline Stages | 5 | 8 | 1 | 13 |
| Area in Slices | 585 | 738 | 96 | 1419 |
| Area in % of design | 41.2 | 52 | 6.8 | 100 |
| Flip-Flops | 1003 | 687 | 225 | 1915 |
| LUT | 665 | 1321 | 198 | 2184 |
| Embed. multipliers | 9 | 0 | 0 | 9 |
| Clock rate -6 [MHz] | 178 | 177 | N.A. | 177 |
| Clock rate -7 [MHz] | 203 | 200 | N.A. | 200 |

*maximum performance of P processing elements, operating at frequency F, with a fixed amount of local memory L*:

$$BW_{PEAK} = \frac{2 \times P \times F}{\sqrt{\frac{L}{2}}} \quad , \quad [words]/[s]. \tag{10}$$

**Peak performance local memory size:** From (10) we can easily derive the amount of local memory required for the best array performance at a given limited bandwidth BW and P processing elements operating at frequency F:

$$L_{PEAK} = 8 \times (\frac{P \times F}{BW})^2 \quad , \quad [words]. \tag{11}$$

**Data hazards condition:** Since the floating point MAC is pipelined, it is possible that data hazards arise during the computations. According to the proposed algorithm, P processing elements calculate an $S_i \times S_j$ block of C. Each PE calculates $(S_i \times S_j)/P$ matrix elements. Data hazards occur when data, required for a MAC operation, are delayed in the addition pipeline, which is 8 stages deep. Therefore, the condition to avoid data hazards is:

$$\frac{S_i \times S_j}{P} > 8 \tag{12}$$

# 4. FPGA IMPLEMENTATION

We consider reconfigurable technology to implement the proposed matrix multiplier. Initially, we employed a "top-down" methodology to design the unit and then tuned this design through a "bottom-up" performance optimization following a "iterative refinement" approach. For our prototype designs, we consider the Xilinx Virtex II Pro FPGA and the Xilinx ISE6.0 design environment. Table 1 presents detailed synthesis data on the implementation of a single PE. As indicated in line 4, the MAC unit (comprising MUL and ADD) occupies nearly 94% of the entire area required by the processing element. Regarding the operating frequency, the MAC and ADD units are well balanced as clock rates are virtually the same. The control logic is only 6.8% of the entire PE, including two 64-bit registers for storing A and B elements and an interface for loading C. We focus on two key parts of the FPGA-specific design: the implementation of the *critical MAC computations* and the implementation of the *operation control*.

**Critical MAC computations:** In Table 2, the FPGA resources utilization and performance figures (synthesis data) are reported for the key computationally critical units of the MAC pipeline. In the last column of the table, the frequencies for speed grade -6 and -7 of the considered V2P125 FPGA chip are reported, respectively.

**Table 2: Critical MAC Computations-Performance.**

| Computation (Xilinx V2P125 -6/-7) | Area [LUT] | Clock [MHz] |
|---|---|---|
| Registered multiplier MUL18X18S | 0 | 280/332 |
| 89-bit addition, 49th carry-out select | 169 | 184/210 |
| Code 55 bits into 2's complement | 56 | 192/219 |
| Shift 56 bits up to 53 positions | 310 | 177/200 |
| Count the leading zeros for 15 bits | 19 | 377/421 |

Multiplication: The Virtex II Pro FPGA supports two types of 18 bit two's complement multipliers, depending on the operating frequency. The lower frequency multipliers are single staged and operate at 165/186 MHz (for V2P125 -6/-7). We utilize the double staged high frequency multipliers MUL18X18S (Table 2), to achieve better pipeline balance and improved throughput. Since the pipeline registers are implemented in the hardwired multiplier itself, no additional resources are required.

Addition: The result of multiplying two 53-bit mantissas is 106 bits wide, but it essentially requires only 89-bit actual additions as the LS 17 bits are added to zero. Yet, this operation requires the widest operands in the MAC design. With the Virtex II Pro fast carry chain, an addition performs well in serial mode but the total delay is proportional to the length of the carry chain. An 89-bit serial addition operates at 161/184 MHz for -6/-7 speed grades, respectively. To reduce the delays, we employed an 89-bit carry select addition. We have implemented two 40-bit adders to calculate the 40 MS bits of the result in parallel as the first adder is implemented with a carry in bit of "1", while the other has no carry in. The 40 MS bits of the final result are determined based on the carry-out bit of the addition of the 49 LS bits by selecting the appropriate of the two results calculated. This method yields approximately 14% increase in the clock rate at the low price of an extra 40-bit adder (see Table 2).

Shifting: The shift operations include shift right, shift left, and two's complement arithmetic shift. They present the most critical computations in the MAC unit both in terms of time and hardware costs. The implemented shifting structure is not area efficient and its performance depends on the data width and shift amount. As indicated in Table 2, a 53-bit positions shifter requires 310 LUTs and its clock rate is 177/200MHz for FPGA speed grades -6/-7, respectively. In [4], Xilinx propose a barrel shifter reference design based on embedded multipliers. Their solution, however, suffers from some key shortcomings, we point next. First, the design contains three stages: encoding the shift amount to "one-hot" format, fine shifting by embedded multipliers, and bulk shifting by multiplexors. This multilevel logic incurs more time delays than a single level logic. Second, the fast registered multiplier comprises 2 stages of pipeline, therefore the shift operation has to be divided into 2 stages at least. Third, the embedded multipliers turn to be a limited resource, especially for floating-point computations. E.g., a single 53-bit shifter requires seven 18-bit embedded multipliers. Fourth, shifting through multipliers saves design real estate, but some placement flexibility is lost due to the locking of the barrel shifters to specific multiplier locations. Therefore, we can safely conclude that both shifting implementations (i.e., the traditional and the multipliers based

structures) are not quite efficient in the considered FPGA technology and a 200 MHz floating-point unit is hard to be implemented within a reasonable area budget. We believe, however, that this technological shortcoming can be solved in some of the future FPGA products.

Counting leading zeros: The complexity of a single stage counting logic prevents the design from fast clock rate and incurs large area consumption. The area of a single 54-bit leading zero counter is over two times larger than the area of four 15-bit counters. Therefore, we distribute the counting into two stages employing some techniques from [15]. In the first stage, the 54-bit data is decomposed into four groups, which are processed separately. The generated four numbers are summed in the second stage.

**Operation control:** The proposed matrix multiplier has a modular structure composed of a liner array of interconnected processing elements. There are unified communication channels between the adjacent PEs with a FIFO-based interface to tolerate the varying delay of the incoming data. The store and load operations from the host to the PEs are pipelined so that there is no central broadcast of data along the array. Within a PE, the MAC operates in a data driven manner. Each operand entering the MAC has an associated validation bit and only when both operands are available, the MAC unit produces valid results. A data number is compared with the internal processor identifier (pid) to select a particular operand TA from the incoming data of matrix A. Three counters control the matrix A and matrix C accesses. One counter is used to calculate the operating number of matrix A, used in the multiplication. After $S_j$ multiplications, the operand from matrix A will be changed by a new one, which requires a read operation from FIFOA to register TA (see Figure 5(a)). The second and the third counter are employed to calculate the local load and store addresses of C. Thus, besides the MAC unit, the structure of the PE only employs several counters and a comparator, and can be fully parameterized for an arbitrary matrix size by a few numbers: pid, $S_i$, $S_j$ and P.

# 5. COMPARISONS TO RELATED WORK

We compare our proposal to several recent and closely related works reported in the literature. The performance evaluations are based on synthesis for Xilinx Virtex II Pro FPGAs with speed grades -6 and -7.

**Closest works:** Regarding 64-bit floating point FPGA matrix multiplication, there are few closely related works reported. In [19], authors investigate the achievable 64-bit floating-point performance by a single FPGA chip. Their research targets applications in scientific computing, more precisely some Basic Linear Algebra Subroutines (BLAS), e.g., dot production, vector matrix multiplication, and matrix multiplication. The paper discusses the relationship between the memory bandwidth and the sustained computation performance in details. For matrix multiplication, an $S \times S$ block algorithm is adopted. The memory requirements are for $6 \times S^2$ 64-bit words of local storage. In contrast, our improved block algorithm has the least memory requirements of $2 \times S^2 + 2 \times S$ and no central broadcasting. In [22], an algorithm has been proposed employing $(N^2/S)$ PE with storage size of S words per PE. A tradeoff between the storage size and the number of PEs has been provided. To achieve optimal performance of the design in [22], the block size has to be equal to the number of PEs. If the MAC unit we propose is employed, the authors of [22] suggest[5] that their design can potentially achieve peak performance similar to ours for a single FPGA (15.60 GFLOPS). The algorithm proposed in [22] incurs higher requirements to the I/O bandwidth than ours. More specifically, we estimated that a bandwidth in the order of at least 3200 MB/s is required for the 8.30 GFLOPS performance reported in [22] (448KB local memory). This bandwidth is clearly higher than the requirements of our design for nearly the same performance (i.e., 400 MB/s for 9.05 GFLOPS for 512 KB local memory, suggested in Table 4). The work in [14] considers implementations of floating-point units on Xilinx Virtex II FPGAs for 41-bit multiplication and addition. The proposed FP format is self-defined so that the 32 mantissa bits conveniently match the multipliers widths. Although such a format is different than ours, we still compare the results from [14] to ours, assuming similar conditions. Table 3 assists a detailed comparison between our proposal and the related works in [14, 19]. Our design has the least reconfigurable area and runs at the highest clock frequency compared to the related works considered. The number of the pipeline stages is reduced to 13, which additionally saves area and reduces the pipeline latency. As a result, a larger number of PEs can be implemented on a chip, thus improving the computations density.

Regarding performance, its true evaluation should be done measuring the processing speed in GFLOPS. In Table 4 we compare the performance of our design to the three considered related proposals. The considered performance figures actually represent the potentially maximum performance, assuming large matrices and local memory sizes in the order of KBytes and more. The potential maximum performance (denoted as $PERF'_{MAX}$ in (8)) is greatly influenced by the sustained I/O bandwidth (BW) and the embedded local memory size (L). Authors of [19] calculate their po-

**Table 3: Related Work Comparisons - Single PE.**

| Xilinx FPGA | xc2vp100 -6 | | |
|---|---|---|---|
| | Ours | [19] | [14] |
| Operand Width | 64 | 64 | 41 |
| Pipeline stages | 13 | 34 | 25 |
| Area in Slices | 1419 | 2875 | 1641 |
| Clock [MHz] | 177 | 140 | 171 |
| #PEs on a chip | 31 | 24 | 26 |

**Table 4: Performance Comparisons.**

| Bwidth [MB/s] | local mem [KB] | Performance [GFLOPS] | | |
|---|---|---|---|---|
| | | Ours | [19] | [14] |
| 100 | 256 | 1.60 | 0.92 | N.A. |
| 200 | 256 | 3.20 | 1.85 | N.A. |
| 200 | 512 | 4.52 | 2.61 | N.A. |
| 400 | 512 | 9.05 | 5.23 | N.A. |
| 400 | 1024 | 12.80 | 7.39 | N.A. |
| 400 | 1600 | 15.60 | 9.18 | N.A. |
| 800 | 256 | 12.80 | 7.39 | N.A. |
| 800 | 512 | 18.10 | 10.45 | N.A. |
| 800 | 1024 | 25.60 | 14.78 | N.A. |
| 2 736 | 108 | 39.46 | 22.78 | 2.200 |

[5] personal communication

tential maximum performance to be $BW \times \sqrt{L/6}$, which is definitely lower than ours, which is $BW \times \sqrt{L/2}$, recall (8). Therefore, our design outperforms [19] in the entire range of bandwidths and local memory sizes, considered in Table 4, by a factor of 1.7 X. The authors of [14] do not give an explicit formula for the potential maximum performance of their designs. Therefore, we use the performance data they report for certain BW and L to calculate the performance of our design and of [19]. Clearly, our proposal outperforms the designs from [14] with a factor of 18X (see the last row of Table 4).

**Other related works:** The scheduling methods for FPGA matrix multiplication proposed in [8, 9, 16] still suffer from small block matrices and do not consider the factor of limited bandwidth. Unlike other works on 64-bit FP units, e.g., [6,18], our work gives detail performance evaluation and discusses the trade-offs in building long word FPU pipelines. In [14], the authors propose a "Propagate-Kill Chain" for encoding the position of the leading "1" using embedded multipliers for the shift operation. Their method still needs one stage for leading zero counting and two stages for the embedded multipliers. Their performance is similar to ours, but at the expense of additional embedded multipliers. We also give credit to other related works regarding FPGA based FP matrix multiplications, which have been beneficial for our proposal. Works in [13] discussed the implementation of an FP unit in the earlier versions of the FPGA chips, which did not support block multipliers and embedded little on-chip memory. Authors in [11, 12] presented works on building FP units of word width shorter than 64 bits, optimized for FPGAs. The authors of [1, 17] propose customizable designs of floating-point units used in the development of automatic design tools. In [10], a memory organization for efficient access of rectangular data blocks is proposed. This memory organization can be considered in our design for performance efficient implementations. We also envision that the proposed design can be implemented as a reconfigurable extension into tightly coupled custom computing machines like the processor reported in [21].

# 6. CONCLUSIONS

In this paper, we introduced a floating point design of a hardware matrix multiplier optimized for FPGA implementations. The core of the proposal is a general block matrix multiplication algorithm, applicable for arbitrary matrix sizes. The algorithm exploits data locality and reusability and considers the limitations of the I/O bandwidth and the local storage volume. We introduced a scalable linear array of processing elements supporting the proposed algorithm, where each processing element is finely pipelined. We employed several methods to improve the throughput of the pipeline, including carry-select addition, distributed leading zero counters, pre-computation of local elements, and fusion of multiplication with addition, while keeping the number of pipeline stages minimal. The 64-bit ANSI/IEEE Std 754-1985 floating point standard was considered for a practical Xilinx Virtex II Pro implementation. Synthesis results confirmed a superior performance-area ratio compared to related recent works. Assuming the same FPGA chip, the same amount of local memory, and the same I/O bandwidth, our design outperformed the related proposals by at least 1.7X and up to 18X consuming the least reconfigurable resources. For instance, a single processing element requires an area of 1401 Virtex II Pro slices and can run at 200MHz. A total of 39 PEs can be integrated into the xc2vp125-7 FPGA, reaching performance of, e.g., 15.6 GFLOPS with 1600 KB local memory and 400 MB/s external memory bandwidth.


# Acknowledgments

Yong Dou's work is supported by the National Science Foundation of China under contract # 90307001. This research has been also supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, and the Technology Foundation STW (project reference code AES.5021). We would also like to acknowledge Ling Zhuo and Keith D. Underwood for the design discussions with them.


# 7. REFERENCES


[1] A. Abdul Gaar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang. Automating Customisation of Floating-Point Designs. In *Proceedings of the 12th International Workshop on Field Programmable Logic and Application (FPL 2002)*, pages 523–533. LNCS 2438, August 2002.

[2] J. Choi. A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. In *11th IEEE International Parallel Processing Symposium (IPPS '97)* , pages 310–314, April 1997.

[3] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, pages 1–17, March 1990.

[4] P. Gigliotti. Implementing Barrel Shifters Using Multipliers. In *Xilinx Applicatioin Notes, http://direct.xilinx.com/bvdocs/appnotes/xapp195.pdf.*

[5] K. Goto and R. A. van de Geijn. On Reducing TLB Misses in Matrix Multiplication. In *FLAME Working Notes #9, Technical Report TR-2002-55.* The University of Texas at Austin, Department of Computer Sciences, November 2002.

[6] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of High-performance Floating-point Arithmetic on FPGAs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 149–156, April 2004.

[7] J. Gunnels, G. Henry, and R. van de Geijn. High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories . In *FLAME Working Notes #4, Technical Report TR-2001-22.* The University of Texas at Austin, Department of Computer Sciences, June 2001.

[8] J. Jang, S. Choi, and V. K. Prasanna. Area and Time Efficient Implementations of Matrix Multiplication on FPGAs. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2002)*, pages 93–100, December 2002.

[9] J. Jang, S. Choi, and V. K. Prasanna. Energy-Efficient Matrix Multiplication on FPGAs. In *Proceedings of the 12th International Workshop on*

*Field Programmable Logic and Application (FPL 2002)*, pages 534–544. LNCS 2438, August 2002.

[10] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. Visual data rectangular memory. In *Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004)*, pages 760–767, September 2004.

[11] J. Liang, R. Tessier, and O. Mencer. Floating Point Unit Generation and Evaluation for FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing machines (FCCM 2003)*, pages 185–194, April 2003.

[12] G. Lienhart, A. Kugel, and R. Manner. Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing machines (FCCM 2002)*, pages 182–191, April 2002.

[13] 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 *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing machines (FCCM 1998)*, pages 206–215, April 1998.

[14] E. Roesler and B. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. In *Proceedings of the 12th International Workshop on Field Programmable Logic and Application (FPL 2002)*, pages 637–646. LNCS 2438, August 2002.

[15] M. S. Schmookler and K. J. Nowka. Leading Zero Anticipation and Detection – A Comparison of Methods. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 7–16, June 2001.

[16] R. Scrofano, S. Choi, and V. K. Prasanna. Energy Efficiency of FPGAs and Programmable Processors for Matrix Multiplication. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2002)*, pages 422–425, December 2002.

[17] N. Shirazi, P. Y. Cheung, A. A. Gaffar, and W. Luk. Customising Floating- Point Designs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing machines (FCCM 2002)*, pages 315–317, April 2002.

[18] K. D. Underwood. FPGA vs. CPUs: Trends in Peak Floating-Point Performance. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA 2004)*, pages 171–180, February 2004.

[19] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA Trends in sustainable floating-point BLAS performance. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing machines (FCCM 2004)*, April 2004.

[20] R. A. van de Geijnv and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. In *LAPACK Working Note 99, technical report, University of Tennessee* , 1995.

[21] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, November 2004.

[22] L. Zhuo and V. K. Prasanna. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 94–103, April 2004.