ELectronics EXpress

LETTER

IEICE Electronics Express, Vol.15, No.10, 1–12

MALMM: A multi-array architecture for large-scale matrix multiplication on FPGA

You Huang[1,2a)], **Junzhong Shen**[1,2], **Yuran Qiao**[1,2], **Mei Wen**[1,2], **and Chunyuan Zhang**[1,2]

[1] *College of Computer, National University of Defense Technology, Changsha 410073, China*

[2] *National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China*

a) *hy690212@163.com*

Abstract: Large-scale floating-point matrix multiplication is widely used in many scientific and engineering applications. Most existing works focus on designing a linear array architecture for accelerating matrix multiplication on FPGAs. This paper towards the extension of this architecture by proposing a scalable and highly configurable multi-array architecture. In addition, we present a work-stealing scheme to ensure the equality in the workload partition among multiple linear arrays. Furthermore, an analytical model is developed to determine the optimal parameters for matrix multiplication acceleration. Experiments on real-life convolutional neural networks (CNNs) show that we can obtain the optimal extension of the linear array architecture.

Keywords: matrix multiplication, field-programmable gate arrays (FPGAs), work-stealing

Classification: Integrated circuits

References

[1] J. Shen, *et al.*: "Towards a multi-array architecture for accelerating large-scale matrix multiplication on FPGAs," arXiv preprint arXiv:1803.03790 (2018).

[2] Y. Dou, *et al.*: "64-bit floating-point FPGA matrix multiplication," Proc. ACM International Symposium on Field-programmable Gate Arrays (FPGA'05) (2005) 86 (DOI: 10.1145/1046192.1046204).

[3] Z. Jovanovic and V. Milutinovic: "FPGA accelerator for floating-point matrix multiplication," IET Comput. Digit. Tech. **6** (2012) 249 (DOI: 10.1049/iet-cdt.2011.0132).

[4] L. Zhuo and V. K. Prasanna: "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," IEEE Trans. Parallel Distrib. Syst. **18** (2007) 433 (DOI: 10.1109/TPDS.2007.1001).

[5] V. B. Y. Kumar, *et al.*: "FPGA based high performance double-precision matrix multiplication," Proc. International Conference on VLSI Design (VLSI'09) (2009) 341 (DOI: 10.1109/VLSI.Design.2009.13).

[6] Y. Qiao, *et al.*: "Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency," Concurrency Computat.: Pract. Exper. **29**

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

(2016) e3850 (DOI: 10.1002/cpe.3850).
[7] R. D. Blumofe and C. E. Leiserson: "Scheduling multithreaded computations by work stealing," J. ACM (JACM) **46** (1999) 720 (DOI: 10.1145/324133.324234).
[8] A. Krizhevsky, *et al.*: "Imagenet classification with deep convolutional neural networks," Advances in Neural Information Processing Systems (2012) 1097.
[9] J. Cong and B. Xiao: "Minimizing computation in convolutional neural networks," Proc. International Conference on Artificial Neural Networks (2014) 281 (DOI: 10.1007/978-3-319-11179-7_36).
[10] K. Simonyan and A. Zisserman: "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556 (2014).
[11] C. Szegedy, *et al.*: "Going deeper with convolutions," Cvpr (2015) (DOI: 10.1109/CVPR.2015.7298594).
[12] D. Yi, *et al.*: "Learning face representation from scratch," arXiv preprint arXiv:1411.7923 (2014).
[13] K. He, *et al.*: "Deep residual learning for image recognition," Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (2016) 770 (DOI: 10.1109/CVPR.2016.90).

1 Introduction

Large-scale floating-point matrix multiplication is widely used in many complicated computation tasks such as scientific computing and deep learning. Recently, modern field-programmable gate arrays (FPGAs) have been becoming a prevailing option for accelerating large-scale matrix multiplication due to their reconfigurability and abundant logic resources. Previous studies [2, 3, 4, 5] have primarily focused on accelerating matrix multiplication on FPGAs by using an efficient architecture, i.e. the one-dimensional systolic array. This architecture was demonstrated successfully in matrix multiplication acceleration, which contributes to low bandwidth requirement and good scalability of the accelerator designs. As the increase of the massive parallel resources of FPGAs, extending this linear architecture can be an attractive option for accelerating large-scale floating-point matrix multiplication.

In this paper, we focus on the extension of the linear array architecture. According to our studies, there exist two approaches to extend the linear array architecture: 1. increasing the length of the linear array; 2. adopting multiple parallel linear arrays. However, all of these approaches can hardly ensure the computation efficiency of the accelerator, if we adopt a fixed architecture for various problem sizes. This paper address this challenge by proposing a configurable multi-array architecture. Different from previous designs which only adopt fixed structures for matrix multiplication acceleration, our design allows adjusting the structure dynamically, by changing the number of PEs in used as well as the PE arrays work in parallel.

This paper is extended from our previous work [1]. Compared to [1], the following improvements are made: 1. the extension of linear architecture is further discussed in this paper; 2. optimization of the work-stealing scheme is presented, which effectively improves the performance of the accelerator; 3. we apply our

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

structure to a wide range of CNN benchmarks, which demonstrates the high efficiency of our multi-array architecture.

The main contributions of this paper can be summarized as follows:

1. We propose an FPGA-based matrix multiplication accelerator with a configurable multi-array structure, with support for a work-stealing scheme to optimize workload partition among PE arrays.
2. Based on the design, we formulate a performance model to estimate the execution time of the proposed accelerator by evaluating the realistic memory bandwidth as well as quantifying data traffic volumes. With the help of the proposed model, the optimal configuration of the multi-array architecture can be obtained.
3. As a case study, we evaluate the accelerator using several real-life CNN models. Besides, we obtain the optimal extension of the linear array architecture for each CNN model and achieve good performance.

2 Background and related work

2.1 Background

In this paper, we focus on the dense matrix multiplication problem $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where matrix $\mathbf{A} \in \mathbb{R}^{M \times K}$, $\mathbf{B} \in \mathbb{R}^{K \times N}$ and $\mathbf{C} \in \mathbb{R}^{M \times N}$, respectively. The calculation of matrix $\mathbf{C}$ is given by

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}. \tag{1}$$

Due to on-chip resource limitation, blocking large-scale matrix multiplication into fine-grained sub-block computational tasks is required. Here, we introduce the block matrix multiplication algorithm in Dou [2], which has been proved to be successful in matrix multiplication acceleration. As shown in Fig. 1, matrix $\mathbf{A}$ is split into $\lceil M/S_i \rceil$ sub-blocks (namely SA) of size $S_i \times K$ and matrix $\mathbf{B}$ is partitioned into $\lceil N/S_j \rceil$ sub-blocks (namely SB) of size $K \times S_j$. In this way, the result matrix $\mathbf{C}$ can be calculated by performing sub-block matrix multiplications on the SA_i and SB_j, where $i \in [1, \lceil M/S_i \rceil]$ and $j \in [1, \lceil N/S_j \rceil]$. The basic idea of this algorithm is to split the multiplication of SA_i and SB_j into multiple inner-product operations of two vectors, i.e U_k and V_k, where U_k is the k^{th} column of SA_i and V_k is the k^{th} row of SB_j ($k \in [1, K]$). Here we define $C_{i,j}$ as the product of SA_i and SB_j, then $C_{i,j}$ can be calculated by accumulating C_1, C_2, …, and C_K iteratively.

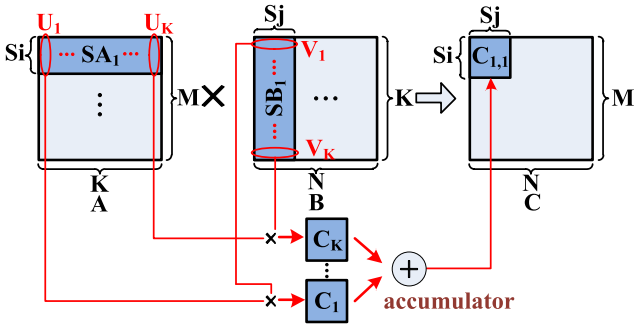

Fig. 1. block matrix multiplication.

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

2.2 Related work

Attempts to implement floating-point matrix multiplication on FPGAs have been widely presented in literature [2, 3, 4, 5, 6]. However, the majority of implementations share a very similar structure — a linear array of PEs.

Dou *et al.* [2] proposed a general parallel block matrix multiplication algorithm as well as a scalable linear array of PEs to implement the algorithm. Zhuo *et al.* [4] proposed three parameterized algorithms that are suitable for matrix multiplication of various size. However, their realistic evaluations were mainly based on small problem sizes due to the resource constraints on FPGAs. Kumar *et al.* [5] presented two designs which used build-in floating-point IP cores, thereby achieved high frequency. However, their designs used broadcast operations which introduced complicated routing. [3] proposed an architecture which returned the results block as soon as they were computed, and left final accumulation to the host. However, its performance depends on communication link speed. Different architectures do exist. An FPGA-based matrix multiplier proposed by Qiao *et al.* [6] contained several parallel chains. However, evaluations on the intensified memory access conflicts among PE arrays were not provided in their work. To the best of our knowledge, there have not yet been many efforts to adopt multi-array architecture for matrix multiplication acceleration on FPGAs. In this paper, we extend the linear architecture to a configurable multi-array structure.

3 Proposed architecture

3.1 Linear architecture extension

Majority of previous works mainly focus on single-array architecture, which contributes to low bandwidth requirement and good scalability of the accelerator designs. Since the massive parallel resources of FPGAs continue to increase, extending this linear architecture becomes an attractive option for accelerating large-scale floating-point matrix multiplication. Two extension options can be summarized from previous work [4, 5]. Fig. 2(a) shows the method of adding more PEs in the array to extend the architecture. This method can remain the advantage of low bandwidth requirement of the original linear architecture, and the computational throughput of the accelerator can be effectively improved (which is proportional to number of PEs). However, with the increase of the number of PEs, the transfer delay between PEs is also getting worse [5]. This is because the input data are only delivered between adjacent PEs, and large delay are required for the PEs that located at the end the PE arrays (i.e. PE6, PE7 in Fig. 2a), especially when the number of PEs are large. As a result, the computational throughput of the

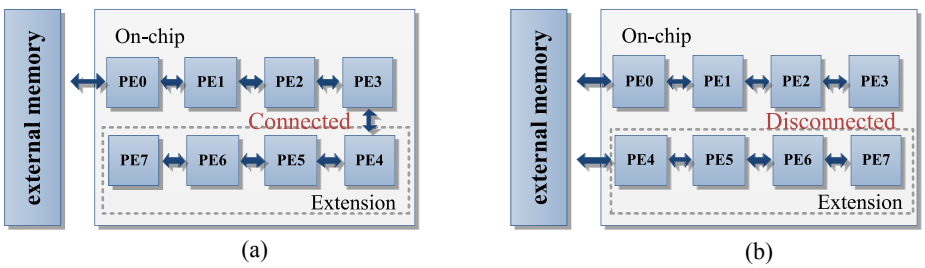

Fig. 2. (a) Linear extension; (b) Parallel extension.

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

accelerator may not increase linearly with the number of PEs. Another method is to expand the single PE array to multiple parallel PE arrays, which is shown in Fig. 2(b). As we can see from Fig. 2(b), there exist external memory access conflicts between two parallel PE arrays. As a result, this method significantly increases the memory bandwidth requirement [2]. The advantage of this method is that the transfer delay between PEs does not deteriorate due to the changeless length of the linear array. Since the actual bandwidth is a limiting factor, the number of PE arrays should be taken into consideration.

3.2 Multi-array architecture

Combining the above methods, we propose a configurable multi-array architecture. Note that this multi-array architecture is mainly extended from the work [2]. However, we make the following improvements which differ from theirs. As shown in Fig. 3, firstly, we extend the single-array architecture into multi-array architecture. Secondly, to support dynamic changing the number of PEs in an array as well as the number of PE arrays, we place a configurable multiplexer between two adjacent PE arrays (detailed follows). In this way, we can adjust the architecture dynamically according to the memory bandwidth budget of targeted FPGA platform. Thirdly, we implement additional control units to support arbitrary block size, and a phase synchronization unit is introduced to guarantee the correctness when the block sizes of **A** and **B** are different. It also can be seen from Fig. 3, the proposed architecture is composed of several modules, including Memory Access Controller (MAC), Workload Queue Management (WQM), and Matrices Processing Engine (MPE). Due to limited amount of on-chip memory on FPGAs, source data and results are stored in the external memory (i.e. the DDR chip). MAC is responsible for data transfer management between DDR and MPE. MPE contains several linear PE arrays which can work in parallel. WQM manages workload queues for all PE arrays and employs dynamic load balancing on the workloads.

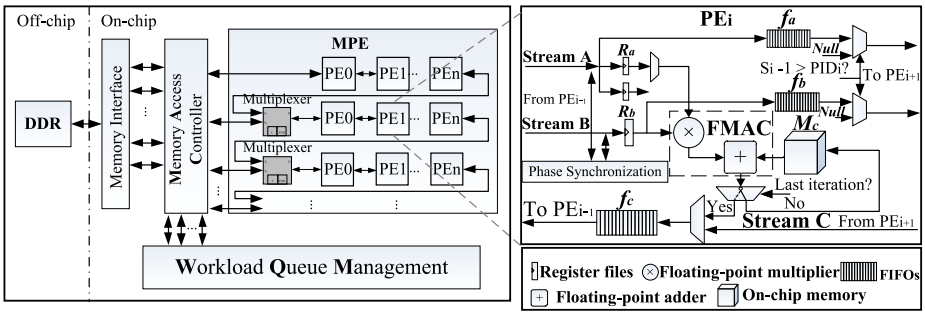

Fig. 3. Block diagram of our proposed architecture.

3.3 MPE design

As shown in Fig. 3, the *MPE* module consists of several linear arrays of PEs, in addition with some multiplexers placed between adjacent PE arrays.

We apply two operation modes in the two adjacent PE arrays, namely the *Independent* mode and the *Cooperation* mode. In the *Independent* mode, the multiplexer between the PE arrays is disabled, meaning that the PE arrays can

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

execute computation tasks independently without any data communication. While in the *Cooperation* mode, the multiplexer between the PE arrays is enabled. As a result, the data paths of the PE arrays are connected by the multiplexer. As shown in Fig. 3, the PE array that placed behind a multiplier can fetch data from the proceeding PE array in this mode. In *Cooperation* mode, the required memory bandwidth of the PE arrays is lower since the PE arrays share the same memory interface when they are connected. The memory access conflicts can be reduced since the connected PE arrays share the same memory interface. In addition, larger block sizes can be supported in the *Cooperation* mode since the number of PEs in the connected array has increased. Note that the multipliers are initialized by the host CPU, and our architecture preserves the scalability of the linear array architecture.

The fully pipelined structure of PE is presented in the right part of Fig. 3. The PE consists of two sets of data registers for input data buffering, three First-In-First-Outs (FIFOs) for delivering data between PEs, local memory for temperate data storing, and floating-point multiply-and-accumulate unit (*FMAC*). Different from previous studies, we implement additional control units to support arbitrary block size. In addition, we implement a phase synchronization unit (*PSU*) to guarantee the correctness of the final results when the block sizes for matrices **A** and **B** are different. By conditionally inserting stalls into the computation pipeline of the PE, the *PSU* ensures that the k^{th} column of *SA* and k^{th} row of *SB* are fetched into each PE simultaneously. The dataflow in each PE consists of three stages:

Prefetch. In this stage, the PE picks up the corresponding element in V_1 (i.e. the first column of *SA*) based on the PE identifier (PID), then stores the data into the local register R_a. For instance, PE_1 with $PID = 1$ will picks up the second element in V_1.

Compute. In this stage, the k^{th} row of *SB* (i.e. U_k) and the $(k+1)^{th}$ column of *SA* (i.e. V_{k+1}) are fetched into the PE simultaneously, where $1 < k \leq K$. The buffered element in R_a is multiplied with all the elements of SB_k in order. Therefore, the data buffered in R_a is reused S_j times. In the meantime, the PE also buffers the corresponding data in V_{k+1} into R_a. Note that we apply double buffering in R_a to overlap buffering data of the next iteration and computation of the current iteration. The products of the multipliers in *FMAC* are then added with the intermediate results generated in the previous iteration, which are stored in the local memory M_c. Finally, the newly sums are written back into the M_c. Note that in the last iteration (i.e. $k = K$), the final results are written into FIFO f_c instead of M_c.

Write back. In this stage, the PE (except PE_0) sends its local results to the proceeding PE from the f_c. As a results, the result data are delivered from the end of each independent PE arrays to the *MAC* module.

3.4 WQM design

The *WQM* module is responsible for workloads assignment for the PE arrays. It manages multiple workload queues to buffer the computation tasks for the PE arrays. Note that one workload queue corresponds to one PE array. For the proposed multi-array architecture, the steadiness of an even partition of workloads among PE arrays is the key to achieve better performance. Since the workloads are

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

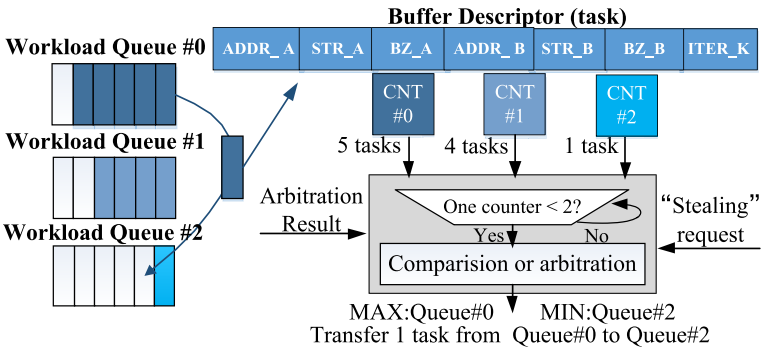

Fig. 4. Illustration of our proposed work-stealing scheme.

not always equally partitioned, the system performance would be bottlenecked by the PE array with the most workloads. To address this issue, we adopt the work-stealing scheme [7] in the design of the *WQM* module.

The basic idea of the work-stealing scheme is to enable an idle PE array to acquire computation tasks from the overloaded PE array(s). Fig. 4 depicts the working procedure of the work-stealing scheme, a controller (omitted in Fig. 4) is designed to manage the workload delivery among the workload queues. It can be seen that a counter is implemented to record the number of tasks for each workload queue. Note that in our previous work [1] only when a workload queue becomes empty will the controller steal a task from a nonempty queue. However, during the stealing procedure, the PE array corresponding to the empty workload queue has to wait for the arrival of the workload. This leads to additional latency to task scheduling, thus performance deterioration. In this work, we overcome the above defect in [1] by proposing a better solution. The key idea of our optimization is to set a threshold for the counters. As shown in the bottom right part of the Fig. 4, once the number of tasks the counter record is no more than one, the controller will transfer a task to the workload queue with least tasks from the workload queue with the most workloads. In this way, the least task workload queue remains nonempty, which makes the related PE array keep working. The above task scheduling is executed in parallel with PE arrays, which effectively overlaps transmission and computation time. Moreover, we implement a round-robin arbiter in the controller to arbitrate multiple concurrent work-stealing requests. The controller repeats the detection and arbitration during the entire computation procedure of the PE arrays. It's important to note that the work-stealing scheme mainly benefits the structure with more PE arrays. It is because the more the number of parallel PE arrays, the harder the workloads is to be equally partitioned.

3.5 MAC design

The *MAC* module is responsible for managing data transfer between the external memory and the accelerator. As shown in Fig. 4, the workloads executed by the *MAC* module are organized by a self-defined data structure named *buffer descriptor*. A *buffer descriptor* contains the following parameters: *ADDR* specifies the memory locations that store the sub-matrices; *STR* specifies the stride of each memory transfer; *BZ* specifies the block sizes and *ITER_K* specifies the iteration (*K*).

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

As mentioned in the above context, elements of matrix **A** are fetched into the PE arrays in column-major order. However, the matrix **A** is stored in row-major order. Therefore, the access of matrix **A** may cause inefficient memory bandwidth utilization. To improve the effective memory bandwidth, we transpose matrix **A** to allow its data to be fetched in row-major order. In this way, the burst transfer mode that favored by the external memory can be used to access both matrices **A** and **B**. As a result, the memory bandwidth for the accelerator is significantly improved, which contributes to performance improvement of the overall system.

4 Performance modeling

In this section, we will illustrate how to determine the optimal solution of mapping the block matrix multiplication algorithm onto the multi-array architecture.

Let the bandwidth of the off-chip memory be BW (B/s, i.e. Bytes per second), the number of PE in a single PE array be P (when all the multiplexers are disabled), the number of PE arrays work in parallel be N_p, block size of the matrix **A** (on rows) be S_i and block size of the matrix **B** (on columns) be S_j. For **A** of size $M \times K$ and **B** of size $K \times N$, the average number of sub-block matrix multiplications performed by each PE array can be expressed as:

$$N_{work} = \left\lceil \frac{1}{N_p} \times \left\lceil \frac{M}{S_i} \right\rceil \times \left\lceil \frac{N}{S_j} \right\rceil \right\rceil. \tag{2}$$

Note that we pad matrices **A** and **B** with zeros if M and N are not integer multiples of S_i and S_j. In addition, the time (in seconds) taken to load a workload (i.e. SA_i and SB_j) and write back the corresponding $C_{i,j}$ can be calculated by:

$$T_{work} = 4 \times (S_i \times K + S_j \times K + S_i \times S_j)/BW, \tag{3}$$

where term $4 \times (S_i \times K + S_j \times K)$ represents the traffic volume of input data (in bytes), and term $4 \times (S_i \times S_j)$ is the amount of corresponding results. To simplify the model, we assume that all the workloads are equally partitioned. Therefore, the time taken to transfer data between the external memory and the PE arrays can be expressed as $T_{trans} = N_{work} \times T_{work}$. According to the data path described in section III, the computation time $T_{compute}$ (in seconds) of a single PE array can be determined as follows:

$$T_{compute} = N_{work} \times (S_i + max\{S_i, S_j\} \times K + Stage_{fmac})/F_{acc}, \tag{4}$$

where $Stage_{fmac}$ denotes the stages of the computation pipeline in each PE, and F_{acc} is the working frequency of the accelerator. Since the memory access and computation process are overlapped in our architecture, it is difficult to directly estimate the execution time of the accelerator. However, the lower bound and upper bound of the execution time T_{total} can be determined by:

$$T_{compute} < T_{total} < T_{trans} + T_{compute}. \tag{5}$$

To simplify the discussion on the parameters that affect T_{total}, we assume $S_i = S_j$ for the rest of this paper. It can be inferred that the attainable memory bandwidth BW for each PE array is mainly affected by N_p and S_i, which can be expressed by $BW = f(N_p, S_i)$.

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

This is because S_i determines the burst length of memory access, and N_p affects the conflicts of memory accesses of the PE arrays. From the above equations, it can be seen that N_p and S_i are the key factors that affect the performance of our accelerator. To reduce the size of design space, the constraints on N_p and S_i are considered. We observe that there exists a relationship between S_i and N_p. Assuming P_m represents the maximum number of the independent PE arrays (when all the multiplexers are disabled), the relationship between N_p and S_i can be determined as:

$$\begin{cases} N_p \in \{1, 2, 3, \ldots, P_m\}, & if\ 1 \le S_i \le P \\ N_p \in \left\{1, 2, \ldots, \left\lfloor \frac{P_m}{2} \right\rfloor\right\}, & if\ P < S_i \le 2P \\ \ldots & \\ N_p \in \{1, 2, \ldots, n\}, & if\ \left\lfloor \frac{\lfloor P_m/n \rfloor}{2} \right\rfloor \times P < S_i \le \left\lfloor \frac{P_m}{n} \right\rfloor \times P \\ \ldots & \\ N_p = 1, & if\ \left\lfloor \frac{P_m}{2} \right\rfloor \times P < S_i \le P_m \times P \end{cases}. \quad (6)$$

To this end, the size of design space can be constrained by equation 6. Given the fixed problem size and $P_m * P$ (i.e. the total number of PEs), the proposed analytical model can be used to determine the optimal $\langle S_i, N_p \rangle$ that minimizes the range of T_{total}. For better understanding, we give a example of $P_m = 4$. Without constraints, there exist $P_m \times P \times P_m = 16P$ possibilities for the space design. However, by using the equation 6, the range of N_p and S_i can be restricted. As a result, the size of design space can be reduced to $4P + 2P + 2P = 8P$.

5 Experimental results

In this section, we evaluate the effectiveness of our proposed performance model. In addition, we present our on-board test based on the guidance of the models. The FPGA platform used in our experiment is the Xilinx VC709 board, which contains a XC7VX690T FPGA and two DDR3 DRAMs. All synthesized results are obtained from Xilinx Vivado 2016.4.

In order to quantify function f, we evaluate the average effective memory bandwidth of a PE array in terms of block sizes and number of PE arrays. As shown in Fig. 5, two observations can be found. First, the effective memory bandwidth goes up with the increase of block size. Second, the effective bandwidth declines when we increase the number of PE arrays.

As a case study, we use real-life CNN models (AlexNet [8], VGG-16 [10], C3D, GoogleNet [11], FaceNet [12], ResNet-18 [13]) to illustrate the validity of our performance model. The main operations in these CNN algorithms (e.g. Convolutional layers, Fully Connected layers) can be represented by matrix multiplication [9]. Note that we focus on determining the optimal design parameters under fixed P_m and P, therefore we do not make full use of the resource of the FPGA to pursuit maximum performance. In our experiment, we set $P_m = 4$ and $P = 64$. After post-synthesis, a maximum frequency of 200 MHz (F_{acc}) is achieved. Table I summarized the resource utilization of the overall system. It can be seen that the overall

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

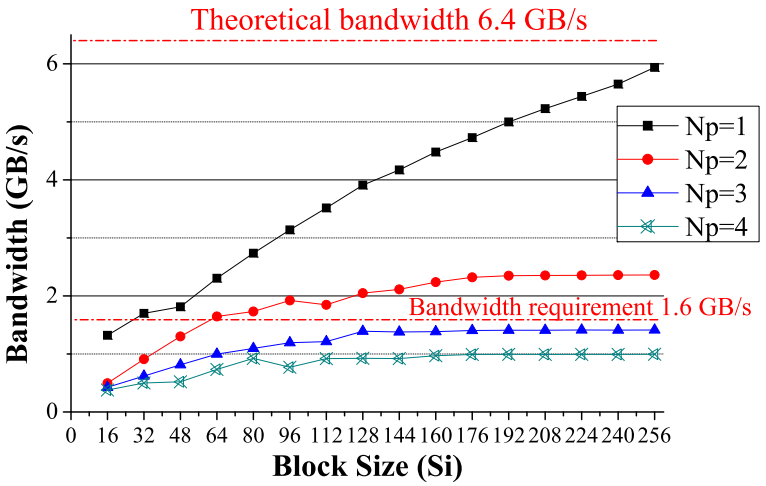

Fig. 5. Effective memory bandwidth with varying block size and number of PE arrays.

Table I. FPGA resource utilization

Resource	DSP	Bram	LUT	FF
Available	3600	2940	433200	866400
Utilization	1032 (28%)	560 (19%)	193765 (45%)	292644 (34%)

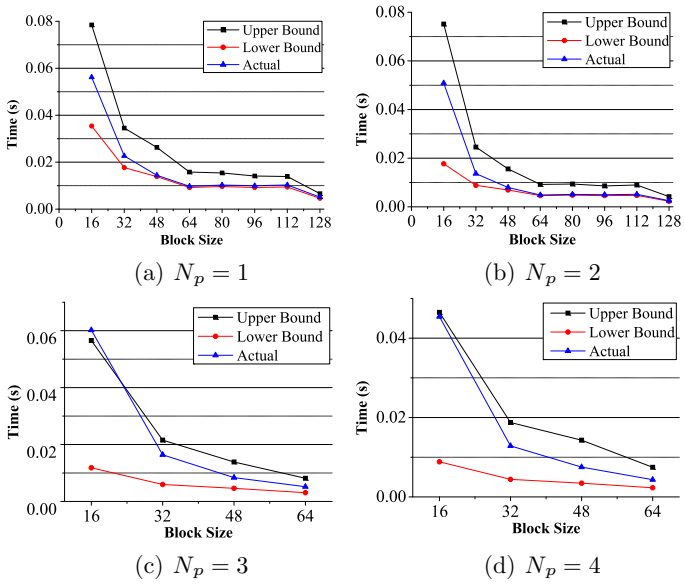

Fig. 6. Comparing actual measurement of execution time with its predictions for conv-2 of AlexNet.

resource utilization is below 50%, which contributes to the high frequency of our accelerator.

We give a detailed comparison between the predicted and actual execution time for a convolution layer of AlexNet in Fig. 6. It can be seen that the predicted lower bound of execution time closely follows the actual measurement when the memory requirement of each PE array is satisfied. However, when the memory bandwidth requirement is unsatisfied, the actual time of each PE array becomes more close to

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

Table II. Optimal (N_p, S_i) of all layers in AlexNet.

Layers	$M * K * N$	Optimal (N_p, S_i)	Performance (GFLOPS)		
			Optimal	$N_p = 4$	$N_p = 1$
Conv-1	$96 * 363 * 3025$	$(2, 128)$	59.7	58.5*	49.2
Conv-2	$128 * 1200 * 729$	$(2, 128)$	87.8	72.1*	61.4
Conv-3	$384 * 2304 * 169$	$(4, 64)$*	77.3*	77.3*	57.4
Conv-4	$192 * 1728 * 169$	$(4, 64)$*	64.2*	64.2*	51.2
Conv-5	$128 * 1728 * 169$	$(2, 128)$	62.9	62.8*	43.9
fc-6	$128 * 9216 * 4096$	$(2, 128)$	100.9	82.1*	70.7
fc-7	$128 * 4096 * 4096$	$(2, 128)$	99.3	81.5*	69.5
fc-8	$128 * 4096 * 1000$	$(2, 128)$	96.9	85.9*	67.8

*means updated performance of our accelerator using optimized work-stealing scheme.

the upper bound of predicted execution time. In addition, it can be found that using multiple PE arrays does not always bring benefits. For example, the case of $(N_p, S_i) = (1, 32)$ is better than $(N_p, S_i) = (2, 16)$. The main reason is that both of the cases are memory-bound (<1.6 GB/s) and the case of $(N_p, S_i) = (1, 32)$ can reach higher memory bandwidth (it can be confirmed by Fig. 5), which contributes to its higher performance.

The optimal $\langle N_p, S_i \rangle$ of all the layers in AlexNet is given in Table II. It can be seen that when compared to other extension approaches, i.e extending the number of PEs only $(N_p = 1, P = 256)$ and extending the number of PE arrays only $(N_p = 4, P = 64)$, our accelerator that implemented with the optimal $\langle N_p, S_i \rangle$ reaches the highest performance for all layers. Note that because of our further optimization for work-stealing scheme, the results of all the layers in AlexNet are updated. Since our work-stealing scheme mainly benefits the structure with more parallel PE arrays, only the case of $(N_p, S_i) = (4, 64)$ obviously improves the performance. As a result, the optimal $\langle N_p, S_i \rangle$ of Conv-3 and Conv-4 are also updated to $(4, 64)$. To demonstrate the effectiveness of our optimization, we compare our original work [1] with the optimized design on the case of $(N_p, S_i) = (4, 64)$. As shown in Table III, we achieve better performance for all layers of AlexNet compared to our previous work (up to 22% increase), which demonstrates that the additional workloads scheduling time is effectively reduced. Table IV shows the peak performance and the overall performance of each CNN model. The second column (i.e. $M * N * K$) in Table IV represents the scale of the transformed matrices of the computational layer in each CNN model. (N_p, S_i) are the optimal parameters for peak performance layer. Note that the scale of the transformed matrices also affects the performance of the accelerator. As shown in Table IV, the overall performance of each CNN model varies widely. Since theoretical peak performance of the proposed architecture is denoted by $2 \times F_{acc} \times P_m \times P$ [2], peak efficiency is given in Table IV (where peak efficiency = actual peak performance/theoretical peak performance). The results of all CNN models show that our multi-array architecture can reach high computation efficiency.

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018

Table III. Comparison with our previous work.

$N_p = 4$	Performance (GFLOPS)								
	*C*1	*C*2	*C*3	*C*4	*C*5	*f*6	*f*7	*f*8	*Overall*
[1]	57.1	70.3	62.9	54.8	44.9	79.3	78.1	83.6	66.3
This work	58.5	72.1	77.3	64.2	62.8	82.1	81.5	85.9	73.1

Table IV. Optimal (N_p, S_i) of peak performance in each CNN model (batch size is 128).

CNN Models	$M * N * K$	(N_p, S_i)	GFLOPS		**Peak Efficiency**
			Peak	**Overall**	
AlexNet	$128 * 9216 * 4096$	(2, 128)	100.9	79.56	98.5%
VGG-16	$128 * 4096 * 4096$	(2, 128)	99.3	83.4	96.9%
C3D	$128 * 4096 * 3025$	(2, 128)	99.2	80.9	96.8%
GoogleNet	$128 * 1024 * 1024$	(2, 128)	90.9	40.7	88.7%
FaceNet	$128 * 320 * 10575$	(2, 128)	81.7	54.9	79.7%
ResNet-18	$128 * 4068 * 1000$	(2, 128)	96.9	64.9	94.6%

6 Conclusion

In this paper, we present *MALMM*, a multi-array architecture for large-scale matrix multiplication, which is highly configurable and scalable. To improve the performance of *MALMM*, we propose a optimized work-stealing scheme to achieve better workload balancing among PE arrays. Since our focus is to obtain the optimal design options for the architecture extension, an efficient analytical model is also developed. Moreover, we evaluate our design using several CNN models on a VC709 board. As a result, our accelerator with optimal extension can reach the high performance and computation efficiency for all CNN models.

Acknowledgments

This work was supported by National Program on Key Basic Research Project 2016YFB1000401 and 2016YFB1000403.

DOI: 10.1587/elex.15.20180286
Received March 20, 2018
Accepted April 16, 2018
Publicized April 27, 2018
Copyedited May 25, 2018