Optimizing the data placement and transformation for multi-bank CGRA computing system

Zhongyuan Zhao*, Yantao Liu*, Weiguang Sheng *, Tushar Krishna†, Qin Wang* and Zhigang Mao*
*Department of Micro/NaNo Electronics, Shanghai Jiao Tong University, Shanghai, China
†School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia
*Email: wgshenghit@sjtu.edu.cn

***Abstract*—This paper provides a data placement optimization approach for Coarse-Grained Reconfigurable Architecture (CGRA) based computing platform in order to simultaneously optimize the performance of CGRA execution and data transformation between main memory and multi-bank memory. To achieve this goal, we have developed a performance model to evaluate the efficiency of data transformation and CGRA execution. This model is used for comparing the performances difference when using different data placement strategies. We search for the optimal data placement method by firstly choosing the method which generates the best CGRA execution efficiency from the candidates who can generate the optimal data transformation efficiency. Then we choose the best data placement strategy by comparing the performance of the selected strategy with the one generated through existing multi-bank optimization algorithm. Evaluation shows our approach is capable of optimizing the performance to 2.76x of state-of-the-art method when considering both data-transformation and CGRA execution efficiency.**

***Index Terms*—CGRA, data placement optimization, multi-bank memory**

I. Introduction

Coarse-Grained Reconfigurable architecture (CGRA) based computing system is one of the promising platforms for computation-intensive applications considering performance, flexibility, area efficiency and energy efficiency. Recent research works on CGRA have gained much success in applications for machine learning [8], cloud computing [9], high resolution displays and high quality audio [10]–[13].

Generally, CGRA computing system consists of CGRA accelerating unit and host processor. The computation-intensive part (e.g. loops) of the program is often mapped onto CGRA and the host processor executes the control-intensive part of the program. To efficiently map the loops onto CGRA, most CGRA compilers use the software pipelining [14]–[17] based algorithm to exploit the loop and instruction level parallelism. However, the limitation of memory bandwidth makes the parallel memory access a bottleneck of the efficiency. Multi-bank CGRA provides the basis for solving the parallel memory access problem, whereas the compiler that can fully takes use of the Multi-bank memory resource is also necessary. The Data placement techniques [1]–[3] try to place the simultaneously accessed data into different memory banks to minimize the memory access conflicts brought by the software pipelining algorithm. However, it is exigent to overcome several problems as follows:

- *The Multi-bank memory that feeds the data to CGRA has limited number of banks.* The restricted bank number of the multi-bank memory has become a critical problem for solving the parallelism memory access issue. Too many banks will dramatically increase the hardware overhead. Thus, it is important for software to help scheduling the operations to fully utilize the multi-bank hardware resources.
- *The limited size of the multi-bank CGRA memory.* Generally, the multi-bank memory that directly feeds the data to CGRA accelerator is on-chip scratchpad memory or global buffer. It is inevitable that the data size of the application exceeds the CGRA memory. The overhead that data transformed from main memory to on-chip memory should be effectively controlled.
- *The contradiction between data communication and data placement.* The data placement algorithm can't guarantee the data is continuously in the multi-bank memory, whereas the data is continuously stored in the main memory. This brings the additional overhead for transforming the data between main memory and multi-bank memory.

Improving the efficiency of the stand alone CGRA accelerator is critical, whereas the data transformation efficiency from main memory to multi-bank CGRA memory must also be guaranteed. Based on this motivation, we build an accurate performance model that reflects the efficiency of the whole CGRA computing system. The performance model presents the efficiency of data transformation between main memory and multi-bank memory, and the efficiency of the stand alone CGRA accelerator with multi-bank memory. Then,we propose an efficient heuristic based algorithm managing the data placement and data communication to optimize the system performance based on the model we build.

To prove the efficiency of our approach, we implement our algorithm and test the executable code on a chip prototype of CGRA computing system (Fig .1). Comparing with the existing multi-bank CGRA optimization, our approach increases

This work was supported by National Science Foundation of China (61201059), National High Technology Research and Development Program of China (2012AA012702), Cross Research Fund of Biomedical Engineering of Shanghai Jiaotong University (YG2014MS70), and Industry University Research Collaboration Program of Shanghai Minhang District (2016MH301).

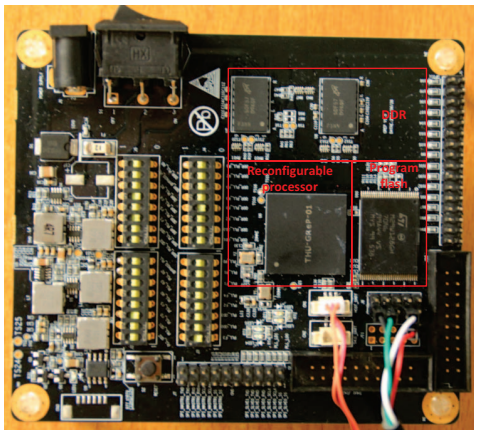


Fig. 1. The test chip of CGRA computing system including one CGRA array, one host processor, 4096kB and 16 banks on chip multi-bank memory and main memory.

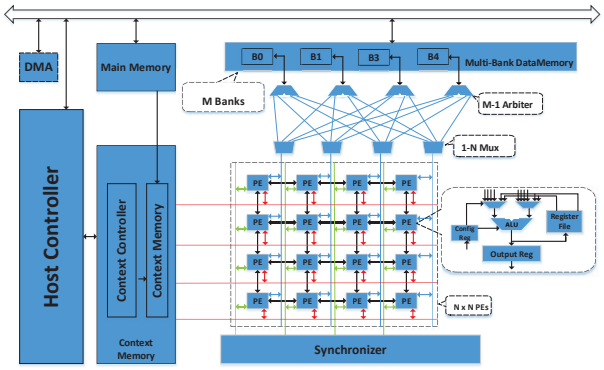


Fig. 2. The high-level system architecture of the CGRA based computing platform. The computing system is heterogeneous, including a host processor and a 4x4 CGRA accelerator. Data is transformed between main memory and multi-bank though DMA.

the whole system performance by 2.76x of state-of-the-art approach.

II. Background and Related Work

A. The CGRA Computing system

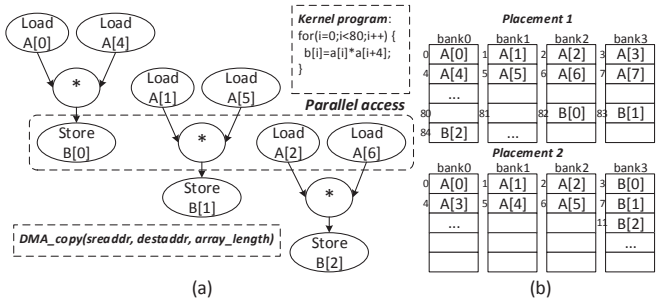
The system level architecture of CGRA computing system is shown in Fig.2. In this system, there are two kinds of computing unit (the host processor and the CGRA accelerator) and three kinds of memory (main memory, multi-bank data memory for CGRA accelerator and the CGRA configuration memory). The CGRA accelerator consists of 4x4 processing elements connected through torus topology. Each processing element (PE) can access data from its local register file, multi-bank memory and other PE's output registers. There is a synchronizer which synchronizes all PEs at the end of every control step. All PEs in the same column are connected to one column bus and and any column bus is connected to any bank through a full crossbar fabric [5], which allows different PEs simultaneously access different banks. There will be multiple cycles in one control step if memory conflict happens between PEs (different PEs simultaneously access the same bank) or the operation latency is different among PEs. The address of the multi-bank memory is interleaved among the banks of the memory.

The execution flow of the CGRA computing system is as follows: at first host processor initiates all the CGRA configuration information and input data into main memory. Before CGRA accelerates a loop, the input data of this loop should be transformed from main memory to multi-bank memory and the configuration information should be loaded onto CGRA's configuration memory. When CGRA finishes computing, the output data is transformed from multi-bank memory to main memory. The parts that consume time includes the task of data transformation from main memory to multi-bank memory and the task of CGRA computation. Existing multi-bank optimization approach brings additional overhead to data transformation, which only optimizes the CGRA computation part. While our method tries to find an optimal solution considering both data transformation and CGRA computation.

B. Multi-bank optimization algorithms

Many Multi-bank optimization algorithms are provided to fully utilize the multi-bank CGRA memory resources. The conflict-free mapping algorithm [4], [5] uses the force-directed method scheduling the operations to reduce the conflict of the parallel access under the software pipelining [14]–[16] execution mode. Then, the data placement algorithm is used to separate the data that will be simultaneously accessed to different memory banks. Conflict-free mapping algorithm for CGRA can significantly reduce the conflicts and help shrink the number of cycles of the control step. But this algorithm is regardless of the data transformation overhead from main memory to the multi-bank memory, which may lead to further performance reduction. There are previous optimization work like [18] which also optimize the pure CGRA execution time.

Figure 3 shows the example of the data placement problem. Iterations of the loop are executed in a pipelined manner. There are two sources of conflicts, one is inside single array A and another is between array A and B. If we use data placement 1 to place the array A and array B, conflicts will happen in every control step because for every iteration i, A[i] and A[i+4] are located in the same bank. What's more, due to the software pipelining based scheduling algorithm, B[i] will be accessed at the same time with A[i+2] and A[i+6]. As they are in the same bank, one more cycle will be added in the control step. However, these two arrays are organized continuously in the multi-bank memory (we assume 4 banks). The data transformation task is efficient in this placement strategy because the DMA is only initiated twice. Conflict-free mapping algorithm [5] will generate the "placement 2" strategy. The conflict is avoided in each control step, but data is transformed multiple times due to the non-continuous placement of array A and B in the multi-bank memory. Thus the goal of this paper is to find out a data placement method



(a) (b)

Placement method	Data transform	Multi-bank conflict
Placement 1	***DMA_copy***(A, 0, total_length), ***DMA_copy***(B, 80, total_length)	1. for array A, A[i] and A[i+4] conflict in every control step 2. for array A and B, B[i], A[i+2] and A[i+6] conflict in every control step
Placement 2	***DMA_copy***(A, 0, 3), ***DMA_copy***(A+3,4,3), ... ***DMA_copy***(B, 3, 1), ***DMA_copy***(B+1,7,1), ...	No conflict inside array A, and between array A and B during any control step

(c)

Fig. 3. (a) the software pipelining execution example, (b) the data organization in multi-bank memory under two data placement strategies. The first placement strategy continuously transforms the data to multi-bank memory. The second strategy separates the data in multi-bank memory in order to reduce the memory access conflict, (c) The data transformation and pipelined execution summarization of two strategies.

which can optimize both CGRA computation efficiency and data transformation efficiency.

C. Data placement

Given a finite n-dimension array A, the address of any data in array A can be represented as $A(\vec{x}) = (x_0^A, x_1^A, ..., x_n^A)$. If we use the linear transformation vector $\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_{n-1})$ to transform the multi dimensional array to single dimensional array, the address of data $A(\vec{x})$ on multi-bank memory can be calculated as:

$$Adr(A(\vec{x})) = I_A + \vec{\alpha} \cdot \vec{x} \tag{1}$$

where I_A is the offset address in multi-bank memory of array A. This equation shows: for any two accessed data $A(\vec{x_1})$ and $A(\vec{x_2})$, which are in the same array, their address difference is not influenced by the offset address of array A, whereas for any two accessed data $A(\vec{x_1})$ and $B(\vec{x_2})$, which are in the different arrays, their address difference in multi-bank will be influenced by the offset of array A and B.

III. The system performance model

The accurate performance model will effectively assist the algorithm in generating the optimal solution. In this performance mode, the execution time can be partitioned into three parts. As shown in equation 2,

$$T_{total} = T_{cgra} + T_{invoke} + T_{trans} \tag{2}$$

where T_{cgra} is the stand alone CGRA accelerating time, which hinges upon the effective scheduling and mapping algorithm. T_{invoke} is the time that CGRA is invoked and reconfigured. The invocation and reconfiguration overhead includes preparing the relative parameters for CGRA and initializing the configuration information of the CGRA. Although CGRA is dynamically reconfigurable during the execution of the program, the CGRA invocation overhead can't be ignored. T_{trans} critically depends on the efficiency of the DMA controller. Too many DMA invocation times will dramatically decrease the efficiency of the data transformation. The T_{cgra} and T_{trans} are two paradoxical metrics. The avoidance of the memory conflicts when accessing multiple arrays or multiple addresses of single array forces the data placement algorithm to place conflicting arrays into different banks. Thus, DMA will have to non-continuously transform single array into multi-bank memory, which will inevitably bring the additional overhead to data transformation. In this way, when T_{cgra} is optimized, T_{trans} is worse. Balancing the T_{cgra} and T_{trans} is the primary problem formulation of this paper.

A. The CGRA execution time model

The CGRA execution time is determined by three parameters:

1) Control steps. The control step reflects the cycle when there is no conflict and all PE operations are within the same latency. When multiple PEs are executing in parallel, they are in the same control step. However, when there exist conflicts or different operation latencies, there maybe multiple cycles inside one control step.
2) The cycle number inside each control step. The ideal situation is that there is only one cycle in each control step, which means there are no conflicts or stalls at all.
3) The scheduling of the data flow graph (DFG) and the layout of data placed onto the multi-bank memory.

Given the data placement strategy D of multi-bank memory, and the schedule and map strategy S, the CGRA execution time can be calculated. Firstly, under the strategy S, the initiation interval (II) of the software pipelining algorithm and the total control step can be computed. We use $T_{cs}(S)$ to denote the total control step under schedule and map strategy S and it can be computed as equation 3:

$$T_{cs}(S) = L + (N-1) \times II \tag{3}$$

where L is the control step latency of the data flow graph (DFG), N is the iteration time of the loop. The CGRA execution time under data placement strategy D and schedule and map strategy S is as equation 4:

$$T_{cgra}(D,S) = \sum_{i=1}^{T_{cs}(S)} C_i(D,S) \tag{4}$$

where $C_i(D,S)$ is the cycle of control step i.

B. Data transformation time model

Given the data layout D of the multi-bank memory, we assume there are m accessed arrays and N_{max} number of banks. For each array A_i we assume they use N(i) banks. We use $T_i(D)$ to denote the data transformation time of array A_i under the data placement strategy D.

$$T_i(D) = \begin{cases} 1, & N_i(D) = N_{max} \\ \frac{L_{A_i}}{N_i(D)}, & \text{else} \end{cases} \tag{5}$$

where L_{A_i} is the total data length of array A_i, $N_i(D)$ is the number of banks allocated for array A_i under data placement strategy D. Thus, given a multi-bank memory data placement strategy for all the arrays, the total data transformation can be calculated as equation 6:

$$T_{trans}(D) = H \times \sum_{i=1}^{m} T_i(D) + \sum_{i=1}^{m} T_{trans}(L_{A_i}) + \delta \quad (6)$$

where H is the DMA invocation overhead, δ is the additional time considering cache miss influence. In our performance model, we assume the DMA invocation overhead H as a constant value. For the cache influence issue, if two different data placement strategies access main memory in the same pattern, we also assume δ doesn't change.

C. Performance difference

Based on equation 1 to 5, the performance model is capable of generating the total execution time under a data placement strategy D, and scheduling and mapping strategy S.

$$\begin{aligned} T_{total}(D,S) &= \sum_{i=1}^{T_{cs}(S)} C_i(D,S) + T_{invoke}(t) \\ &+ H \times \sum_{j=1}^{m} T_i(D) + \sum_{j=1}^{m} T_{trans}(L_{A_i}) + \delta \end{aligned} \quad (7)$$

As this performance model is used to analyze how performance is influenced under different data placement strategies. In order to compare the difference, we don't need to calculate the exact performance under each strategy. At first, to guarantee the accuracy of this model there should be an accurate model simulating the cache action and CGRA invocation overhead, which is beyond the purpose of our goal. Second, different data placement strategies only change the data organization on multi-bank memory. Data remains the same in main memory and is accessed continuously under different placement strategies. It is reasonable to assume cache influence δ remains the same. At last, data placement doesn't influence the kernels organized inside the application, the number of kernels and the CGRA invocation time remains the same.

Thus, we can calculate the difference between different data placement strategies by eliminating the same factors, and leaving the variable part (equation 8):

$$\begin{aligned} Diff(D_p, D_q, S) &= \sum_{i=1}^{T_{cs}(S)} (C_i(D_p,S) - C_i(D_q,S)) \\ &+ \sum_{j=1}^{m} H \times (T_j(D_p) - T_j(D_q)) \end{aligned} \quad (8)$$

if $Diff(D_p, D_q, S) > 0$, the performance under date placement strategy D_q is better than D_p, otherwise, the performance of D_p is better than D_q.

IV. The optimization algorithm

The optimization algorithm starts with an initial data placement strategy D_{ini}, then it keeps searching for another data placement strategy D_{new} and comparing it with the previous one using a different model in the last section until we find the optimal solution.

When the address space of the multi-bank memory is large, the searching space of this algorithm is also large. To reduce the searching space, we give a heuristic method to choose the data placement strategy from several candidates instead of searching. We choose the candidate placement strategies by forcing all arrays to be continuously placed in the multi-bank memory. In this way, we can search the optimal solution by changing the offset of each array and compare them according to the performance difference model we build. The goal of our method is to firstly choose a best data placement strategy from the candidate strategies whose efficiency of data transformation is the best. The number of candidate placement strategy is N^{m-1} where N is the total bank number of the multi-bank memory and m is the number of accessed arrays in the kernel. Because, for any integer value p and q and two accessed data $A(\vec{x_1})$ and $B(\vec{x_2})$, the bank difference BD can be calculated as:

$$\begin{aligned} BD &= [(I_B + pN + \vec{\alpha_B} \cdot \vec{x_2}) - (I_A + qN + \vec{\alpha_A} \cdot \vec{x_1})]\%N \\ &= [(I_B + \vec{\alpha_B} \cdot \vec{x_2}) - (I_A + \vec{\alpha_A} \cdot \vec{x_1}) + (p-q)N]\%N \\ &= [(I_B + \vec{\alpha_B} \cdot \vec{x_2}) - (I_A + \vec{\alpha_A} \cdot \vec{x_1})]\%N \end{aligned} \quad (9)$$

This means bank difference is not relative to the specific offset address of two arrays, but relative to the specific bank they are located in.

After we find the optimal data placement strategy from all the candidates, we then compare it with the data placement strategy using the conflict free mapping algorithm according to the performance difference model and select the final placement method. The algorithm is shown as follows:

Algorithm 1 The Data placement algorithm

```
1:  function DATAPLACE((A_1, A_2, ..., A_m))
2:      D_ini ← Place(A_1, 0), (A_2, 0), ..., (A_m, 0);
3:      D_old ← D_ini
4:      for 0 <= i_1 < N do
5:          ...
6:          for 0 <= i_m < N do
7:              D_new ← Place(A_1, i_1), (A_2, i_2), ..., (A_m, i_m);
8:              if Diff(D_old, D_new) > 0 then
9:                  D_old ← D_new
10:             else
11:                 continue;
12:             end if
13:         end for
14:     end for
15:     D_new ← PlaceConflictFreeMap()
16:     if Diff(D_old, D_new) > 0 then
17:         returnD_new
18:     else
19:         returnD_old
20:     end if
21: end function
```

The reason we first select the candidates from the solutions which have the best data transformation efficiency is that the data transformation time always occupies most of the program execution time of CGRA computing system. We consider the data transformation efficiency, then consider the CGRA

TABLE I
DATA TRANSFORMATION TIME AND CGRA EXECUTION TIME COMPARISON (CYCLES)

kernels	Data transformation time	CGRA execution time
conven	1946	1268
lcs	10932	16929
dotp	5711	1408
spmv	8700	3080
mcf	1111	144
auctor	15056	88854
aes	16998	445030
fil	8763	338758
karatsuba	9987	121385

execution efficiency. Table 1 shows the data transformation time and CGRA execution time from 9 kernels we select. We use the placement strategy which focuses on optimizing the data transformation time. Except for the data transformation time of kernels *aes*, *fil* and *auctor*, the data transformation time of all the other kernels are occupy more than 40% of the total execution time. Secondly, this selection strategy will guarantee the best efficiency of data transformation. In addition, the CGRA execution time will be further improved. If the offset difference between arrays can be correctly set, the conflicts between arrays can also be avoided. At last, we also compare the performance to the strategy which is stress on optimizing the CGRA execution time. For kernels like *aes*, *fil* and *auctor*, stress on optimizing the CGRA execution time will be more effective.

V. EVALUATION

To show the efficiency of our data placement algorithm, we first set the data placement using the strategy generated from conflict-free [5] mapping algorithm, we use the CGRA computing test chip (figure 1) to test the efficiency of the method. Then, we use the method of our data placement algorithm and run on the chip. The tested kernels are selected from Mibench [6] which covers part of the berkeley 13 algorithms [7].

A. Data transformation influence

We make a comparison on the performance of data transformation between our method, conflict-free mapping, and the best strategy chosen from the candidate strategies which generate the optimal data transformation efficiency (best trans). Compare to the strategy of optimizing the data transformation, the conflict-free algorithm brings average 5.6 times of the transformation overhead (table 2), whereas our method brings 2.8 data transformation overhead on average.

Table 3 shows the DMA initiation times comparison between strategies mentioned above. For kernels like *dotp*, *auctor* and *mcf*, the conflict-free memory mapping algorithm is inefficiency because they incurs too many times of DMA initiation. This shows the non-continuously transformation will significantly increase the DMA initiation times and increase the data transformation time.

TABLE II
DATA TRANSFORMATION COMPARISON (CYCLES)

kernels	*ours*	*conflict-free*	*best trans*
conven	1946	11931	1946
lcs	10932	29985	10932
dotp	5711	19672	5711
spmv	8700	28864	8700
mcf	1111	7600	1111
auctor	15056	104430	15056
aes	118854	118854	16998
fil	42786	42786	8763
karatsuba	73627	73627	9987
average	**2.8**	**5.4**	**1**

TABLE III
DMA INVOCATION TIMES COMPARISON

kernels	*ours*	*conflict-free*	*best trans*
conven	3	92	3
lcs	257	455	257
dotp	4	179	4
spmv	58	369	58
mcf	4	64	4
auctor	32	705	32
aes	96	96	17
fil	296	296	176
karatsuba	36	36	21
average	**1.7**	**14.5**	**1**

B. CGRA execution influence

We then compare the performance of stand alone CGRA execution time between three methods mentioned above (table 4). The conflict-free mapping algorithm is capable of generating 2.92 times performance of the data placement method which only optimizes the data transformation. For kernels like aes, fil and karatsuba, the CGRA execution time is far more longer than their data transformation time (table 1 shows the CGRA execution time of these kernels consume at least 10 times of their data transformation time). If we compare the average efficiency of data transformation and CGRA execution based on table 2 and table 4, we can find that, although the conflict-free mapping algorithm optimizes the CGRA execution time by 2.92 times, it generates the overhead of data transformation by 5.4 times. The data prove that our motivation of optimizing both data transformation and and CGRA execution is meaningful.

It should be noticed that the conflict-free mapping algorithm may not generate the optimal solution. For example, in *lcs*, the performance of the strategy which focuses on optimizing the data transformation is better than using the conflict-free mapping strategy. This is because the number of banks required by the conflict-free mapping algorithm exceeds the number of banks in multi-bank memory. In this way, conflict is inevitable even using the conflict-free mapping algorithm. However, the efficiency of data transformation can be guaranteed by the method who focus on optimizing the data transformation.

C. Total influence

Table 5 shows the performance comparison of three methods when adding the time of data transformation and CGRA

TABLE IV
THE STAND ALONE CGRA EXECUTION TIME COMPARISON (CYCLES)

kernels	*ours*	*conflict-free*	*best trans*
conven	1268	1032	1268
lcs	16929	18992	16929
dotp	1408	459	1408
spmv	3080	568	3080
mcf	144	96	144
auctor	88854	32432	88854
aes	16998	16998	118854
fil	211320	211320	338758
karatsuba	43256	43256	121385
average	**1.98**	**1**	**2.92**

TABLE V
THE TOTAL PERFORMANCE COMPARISON (CYCLES)

kernels	*ours*	*conflict-free*	*best trans*
conven	3214	12963	3214
lcs	27861	68950	27861
dotp	7119	20131	7119
spmv	11780	59432	11780
mcf	1255	7696	1255
auctor	103910	136862	103910
aes	295664	295664	462028
fil	254106	254106	347521
karatsuba	131372	116883	131372
average	**1**	**2.74**	**1.1**

execution. For all of the tested kernels, our method generates the best performance, either compared with strategy using the data transformation optimization method or conflict-free mapping method. This means the performance model we design is reliable when we use it to compare the performance difference between different placement strategies.

Conflict-free mapping algorithm is competitive in executing kernels whose CGRA execution time is far more longer than data transformation. Whereas for most applications whose data transformation time is longer, focusing on optimizing the data transformation time will be more effective than conflict-free mapping algorithm.

VI. CONCLUSION AND FUTURE WORK

Optimizing the efficiency of both data transformation and CGRA accelerator computation is meaningful to nowadays multi-bank CGRA computing system. The research work of this paper provides a pre-exploration work on simultaneously optimizing the data transformation and the CGRA accelerator computation.

The data placement algorithm is hard to be totally automatic, because it is extremely difficult to optimize the data communication between main memory and multi-bank buffer without the guidance of the programmer. As long as multi-bank memory size is limited and the data scalings of modern application keep increasing, the total automatically optimization on simultaneously optimizing the data transformation and the accelerators execution will always be a great challenge for compiler researchers. However, our work can be developed as a memory profiler which can feedback to the programmers and help the programmers manage the data with optimal efficiency.

REFERENCES

[1] Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu and Shaojun Wei, "Efficient memory partitioning for parallel data access in multidimensional arrays," in 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, 2015, pp. 1-6.

[2] Shouyi Yin, Zhicong Xie, Chenyue Meng, Leibo Liu and Shaojun Wei, "Multibank memory optimization for parallel data access in multiple data arrays," in 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2016, pp. 1-8.

[3] L. Song, M. Feng, N. Ravi, Y. Yang and S. Chakradhar, "COMP: Compiler Optimizations for Manycore Processors," in 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, 2014, pp. 659-671.

[4] S. Yin, Xianqing Yao, Tianyi Lu, L. Liu and S. Wei, "Joint loop mapping and data placement for coarse-grained reconfigurable architecture with multi-bank memory," in 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2016, pp.1-8.

[5] S. Yin *et al.*, "Conflict-Free Loop Mapping for Coarse-Grained Reconfigurable Architecture with Multi-Bank Memory," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 9, pp. 2471-2485, Sept. 1 2017.

[6] Krste Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," in Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[7] M. R. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), 2001, pp. 3-14.

[8] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efcient Reconfgurable Accelerator for Deep Convolutional Neural Networks," IEEE Journal of Solid-State Circuits 52, 1 (Jan 2017), pp. 127-138.

[9] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis, "TETRIS: Scalable and Efcient Neural Network Acceleration with 3D Memory," in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 17). ACM, New York, NY, USA, pp. 751-764

[10] B. Egger *et al.*, "A space- and energy-efcient code compression/decompression technique for coarse-grained reconfgurable architectures," in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 197-209.

[11] C. Kim *et al.*, "ULP-SRP: Ultra low power Samsung Reconfgurable Processor for biomedical applications," in 2012 International Conference on Field-Programmable Technology, pp. 329-334.

[12] J. Lee, Y. Shin, W. J. Lee, S. Ryu, and J. Kim, "Real-time ray tracing on coarse-grained reconfgurable processor," in 2013 International Conference on Field-Programmable Technology (FPT), pp. 192-197.

[13] Yongjun Park, Hyunchul Park, and Scott Mahlke, "CGRA Express: Accelerating Execution Using Dynamic Operation Fusion," in Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 09). ACM, New York, NY, USA, pp. 271-280.

[14] B Ramakrishna Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in Proceedings of the 27th annual international symposium on Microarchitecture. ACM, pp. 63-74.

[15] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfgurable architectures using modulo scheduling," in Computers and Digital Techniques, IEEE Proceedings, Vol. 150. IET, pp. 255-261.

[16] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula, "EPIMap: using epimorphism to map applications on CGRAs," in 2012 49th ACM/IEEE Design Automation Conference (DAC), pp. 1280-1287.

[17] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "REGIMap: Register-aware application mapping on CoarseGrained Reconfgurable Architectures (CGRAs)," in 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp.1-10.

[18] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1-8.