A New Reconfigurable Architecture with Applications to IoT and Mobile Computing

Amir Masoud Gharehbaghi(✉), Tomohiro Maruoka, and Masahiro Fujita

The University of Tokyo, Tokyo, Japan
{amir,maruoka}@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract. Traditional reconfigurable devices known as FPGAs utilize a complicated programmable routing network to provide flexibility in connecting different logic elements across the FPGA chip. As such, the routing procedure may become very complicated, especially in the presence of tight timing constraints. Moreover, the routing network itself occupies a large portion of chip area as well as consumes a lot of power. Therefore, limiting their usage in mobile applications or IoT devices with higher performance and lower energy demands. In this paper, we introduce a new reconfigurable architecture which only allows communication between neighboring logic elements. This way, the routing structure and the routing resources become much simpler than traditional FPGAs. Moreover, we present two different method for scheduling and routing in our new proposed architecture. The first method deals with general circuits or irregular computations and is based on integer linear programming. The second method is for regular computations such as convolutional neural networks or matrix operations. We have shown the mapping results on ISCAS benchmark circuits as general irregular computations as well as heuristics to improve the efficiency of mapping for larger benchmarks. Moreover, we have shown results on regular computations including matrix multiplication and convolution operations of neural networks.

Keywords: Reconfigurable architecture · Placement and routing · FPGA · Mobile computing · IoT · Convolutional neural network

1 Introduction

Recently, with the advances in internet of things (IoT) technology as well as the increase in the computing power of mobile devices, there is a growing interest in performing different computational tasks on IoT or mobile platforms, especially for edge computing solutions.

In the edge computing paradigm, it is essential to reduce the amount of data that is transferred between IoT nodes and the servers. Therefore, the IoT nodes or mobile devices require to perform more computation tasks. This way, the communication bottleneck due to data transfer bandwidth or communication latency is avoided.

Programmable hardware devices, such as Field Programmable Gate Arrays (FPGAs), have shown to be superior in term of performance as well as energy efficiency for different computation tasks compared to using CPUs or GPUs [1–5]. Consequently, very promising to be used in IoT or mobile platforms. Another category of programmable devices is Coarse Grain Reconfigurable Architecture (CGRA). CGRAs usually provide a less complicated and more regular communication among the processing elements or the processor cores compared to FPGAs to enable acceleration of applications [6–9].

In this work, we have proposed a new reconfigurable hardware architecture suitable for IoT or mobile platforms with the following characteristics.

1. Traditional FPGA devices have a complicated routing architecture to provide flexibility in connecting internal logic blocks together. Consequently, mapping an application to an FPGA device may become very difficult when the utilization of internal blocks is high, or the timing constraints are very tight and hard to achieve. Therefore, we have proposed an architecture which provides connectivity only to neighbors employing a mesh topology, similar to CGRAs; consequently, simplifying the routing architecture as well as the routing method greatly.
2. In the traditional FPGA devices, the routing network, because of its complexity, occupies a large portion of the chip area; consequently, increasing the power consumption. However, in our proposed architecture, because of connectivity to only neighbors, the routing network is greatly simplified, reducing its size as well as the power consumption.
3. Traditional FPGA devices are very fine grained, meaning that the basic functional blocks can only implement basic logic functions. However, in our architecture, the basic functional blocks may be more complicated, for example ALUs or more complicated processing elements. However, unlike CGRAs the functional blocks are not very coarse grained.

In addition to the new proposed architecture, we have provided a scheduling and routing algorithm using integer linear programming (ILP). Starting from a data-flow graph model of the application, ILP formulation results in efficient mapping of the application to our proposed architecture, as well as the optimum latency for executing the application. Moreover, we have introduced heuristics to improve the efficiency of the routing method for larger circuits.

The proposed mapping method is general and may be used for random logic or irregular computations as well as regular computations such as operations on a matrix or convolution operations in neural networks. However, for regular computations there is a more efficient way that is scalable to even very large problems that employs the inherent regularity of the operations. Our proposed mapping method for large regular computations is based on automatic mapping for the small instances of the problem, and induction-based generalization for larger problem.

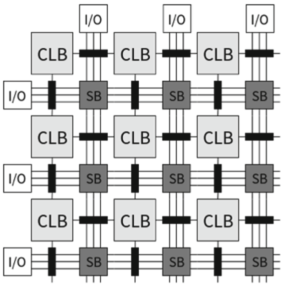

Fig. 1. Traditional FPGA architecture

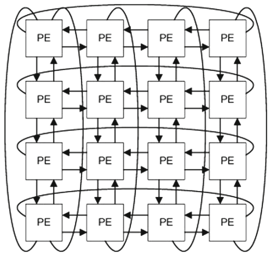

Fig. 2. Traditional CGRA architecture

Please note that because of our simpler and more regular routing architecture compared to traditional FPGA devices, it is possible to achieve very deep pipelining with higher clock frequency, consequently improving the performance of the computation on our proposed architecture.

The rest of the paper is organized as follows. Section 2 gives a background on reconfigurable devices and ILP. Section 3 presents our proposed hardware architecture. Section 4 presents our scheduling and routing (mapping) method for general circuits or irregular computations. Section 5 presents our method for regular computations. Section 6 shows the experimental results. Section 7 concludes the paper and gives some future directions.

2 Background

2.1 Reconfigurable Hardware

Field-programmable gate array (FPGA) is a semiconductor device with a matrix of programmable logic blocks and a programmable routing network. Figure 1 shows a simple architecture of a FPGA. Configurable logic blocks (CLBs) contain 1 or more lookup-tables (LUT) to implement small logic functions. LUTs may have 4 or more inputs and can implement any logic function of 4 or more inputs. Each CLB may contain 1 or more LUTs. The routing network consists of programmable switch blocks (SB) and a number of links connecting SBs together. By programming SBs, CLBs can be connected together to implement larger logic functions. Moreover, some SBs are connected to input/output (I/O) pins to be able to transfer data between FPGA and the outside world.

Coarse grain reconfigurable architecture (CGRA) is another kind of reconfigurable device, shown in Fig. 2. One main difference between CGRA and FPGA is the communication network. CGRAs usually use 2-D mesh or torus network. Moreover, the processing elements of CGRA are processor cores or some complicated processing units, providing coarse grained parallelization of operations or tasks.

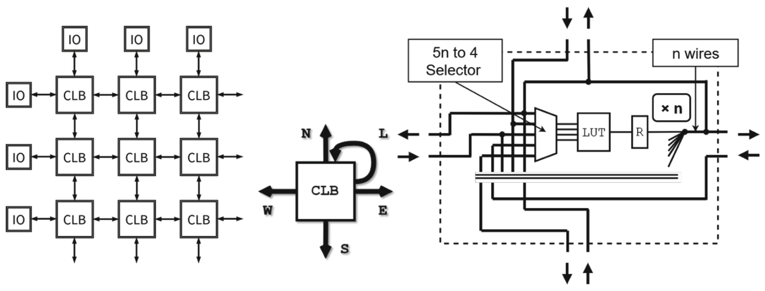

Fig. 3. Proposed architecture

2.2 ILP

Integer linear programming (ILP) is an optimization method that tries to find the values for a set of integer variables given a set of linear constraints, such that a linear function becomes minimized or maximized.

For example: minimize $x + 6 * y$ subject to:

$$2 * x + y \leq 7,\ \ 3 * x + 5 * y \leq 15,\ \ x, y \geq 0$$

There are numerous problems in the field of electronic design automation (EDA) that can be mapped to an ILP problem, including scheduling and routing, which is the target of this paper.

3 Hardware Architecture

In this section, we present our proposed hardware architecture and comparison to the traditional FPGA architecture.

The general architecture is shown in Fig. 3 (left). The basic building blocks are configurable logic block (CLB), that are connected in a 2-D mesh architecture. Each CLB is connected to its four neighboring CLBs (North, South, East, West), as shown in Fig. 3 (center). Moreover, each CLB's data can be used in the next cycle with the Local link. CLBs at the edges are connected to I/O pins to transfer data to/from the device.

Inside each CLB is shown in Fig. 3 (right). Each CLB contains a number of 4-input lookup tables (LUTs). LUTs can be used to implement any 4-input logic function. The 4 inputs of the LUT are selected from the 5 inputs (N, S, E, W, L). Output of each LUT is latched and is send to all the four neighbors.

Each CLB may have a number of LUTs (n). Therefore, n different logic functions may be implemented in a CLB. Also, because of the connectivity to all neighbors, n data line exists between every two neighboring CLBs.

Although in the architecture shown in Fig. 3, LUTs are used inside CLBs, LUTs may be replaced with more complex processing elements such as ALUs. This way, higher-level applications, instead of logic functions, can be mapped to the hardware easily.

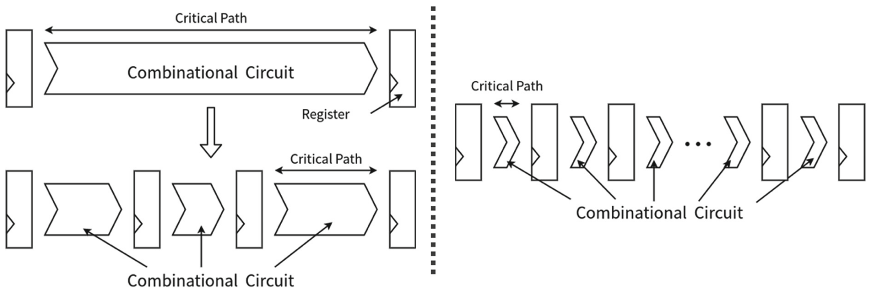

Fig. 4. Comparison of pipelining in a FGPA (left) and the proposed architecture (right)

Following are the main differences between the proposed architecture and a traditional FPGA.

In the proposed architecture, data can be transferred only from a CLB to a neighboring CLB. Therefore, to transfer data from a CLB to other non-neighboring CLBs, some CLBs in between should be used as transfer points. However, in a FGPA data may be transferred from any CLB to any other CLB as long as there is an available routing resource. Consequently, data transfers in FPGA may take a long time, increasing the critical path delay, and reducing the clock frequency. However, in our proposed architecture, all data transfers are between two neighboring CLBs, and take exactly one clock cycle. Moreover, the critical path delay is fixed, and so the clock frequency.

This means that a given application, when mapped to our architecture, may result in very deep pipeline. In general, having a very deep pipeline helps to improve the performance, although the latency may be increased. Another important feature of our proposed architecture is that the critical path delays are fixed; hence, the timing closure problem may not happen. Figure 4 illustrates the differences in the pipelining of a FPGA and our proposed architecture.

4 General Mapping Method

In this section, we present our mapping method for the proposed hardware architecture. The input of the mapping method is the data-flow graph (DFG) of the application, and the general information about the hardware such as the architecture mesh size, and the number of LUTs in a CLB. The output of the method specifies that each node of the DFG is executed on which LUT of which CLB, and at which cycle. For example, the simple DFG shown in Fig. 5 (left) will be mapped to the proposed architecture as shown in Fig. 5 (right).

4.1 ILP Variables

We have formulated the problem of mapping as an integer linear programming (ILP) problem that can be solved with any ILP solver, as follows. We have defined the binary variable D to represent the data transfers.

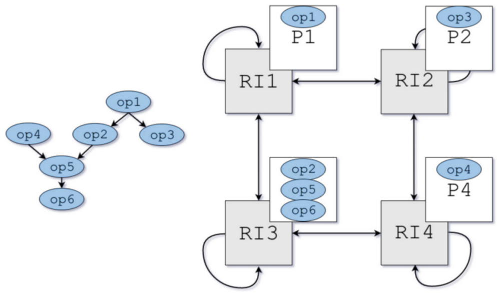

Fig. 5. Example of mapping a DFG in to the proposed architecture

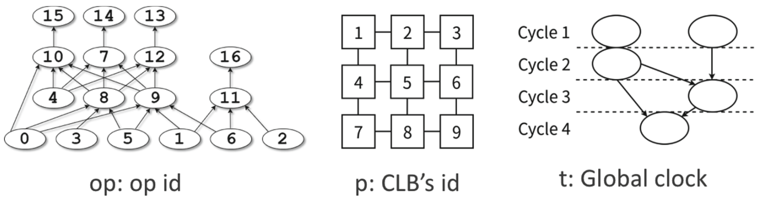

Fig. 6. Parameters in mapping DFG nodes

$D(op, t, p)$: op data is mapped to be used at cycle t on CLB number p
op, t and p parameters are shown in Fig. 6.

An example of binding is shown in Fig. 7. Assuming, each gate in the figure is going to be executed on one LUT, one possible way is as follows. Gate1 is executed in CLB5 at time 2, and data is transferred at time 3: D(1, 3, 5) = 1. Gate2 is assigned to CLB8, which is a neighbor of CLB5, and executes at time 3, and its output is transferred at time 4: D(2, 4, 8) = 1.

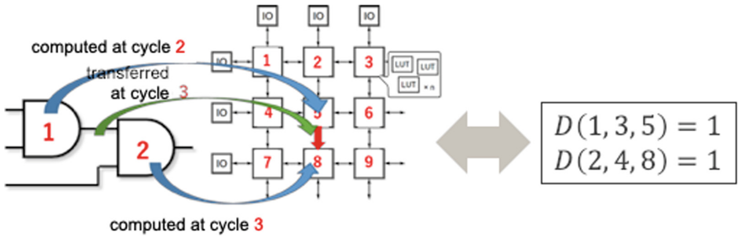

Fig. 7. Mapping example

4.2 ILP Constraints

There is a constraint that all the nodes' data should appear at least once, meaning that all the operations are performed, as follows:

$$\forall op, \sum_t \sum_p D(op, t, p) \geq 1$$

Note that in the definition of variables, we have assumed that any op can be mapped to any CLB, meaning that all CLBs are capable of executing any DFG node function. However, in general, we may have different DFG node types and different CLB types, and each type of DFG node may be mapped to different CLB types. In that case, we need additional constraints to define not allowed resource bindings, by adding constraints that which data transfers are not feasible. It means that some $D(op, t, p)$ variables are always 0.

Data transfer constraints make sure that all the edges of DFG are assigned at least once. Moreover, because of the proposed architecture, data transfer constraints make sure that correct sequence of data transfer is performed, meaning that data is only transferred to the neighboring nodes in one cycle. The constraints are as follows:

$$\begin{aligned}&\forall op, t, p, op_{par},\\ &-D(op,t,p) + \sum\nolimits_{p' \in p_s} D(op, t-1, p') + \sum\nolimits_{p' \in p_s} D(op_{par}, t-1, p') \geq 0\\ &\forall op, t, p,\\ &-D(op,t,p) + \sum\nolimits_{p' \in p_s} D(op, t+1, p') + \sum\nolimits_{op_{chi}} \sum\nolimits_{p' \in p_s} D(op_{chi}, t+1, p') \geq 0\end{aligned}$$

The other constraint is regarding the number of registers in a block. Assuming the number of registers is n, maximum n operations may be performed in a block:

$$\forall p, \sum\nolimits_{op} \sum\nolimits_{t} D(op, t, p) \leq n$$

4.3 Objective Function

In our problem, the goal is to map in such a way that the latency is minimum, meaning that the last cycle that any DFG node is executed (T) be minimum. The constraint for the objective function is as follows:

$$\forall op, t, p, (t * D(op, t, p) \leq T)$$

However, practically we can avoid adding the above constraint, as the additional constraints slow down the ILP solver. In our solution, when we are generating the problem for the ILP solver, we only add $D(op, t, p)$ variables for $t \leq T$. This way, if the problem has a solution with time limit of T, it can be solved. Otherwise, the problem has no solution, meaning that the execution of the given DFG in the given architecture cannot be finished in less than T cycles.

5 Regular Computations Mapping Method

The general mapping method presented in the previous section works for any kind of circuit. However, it is required to employ heuristics to become scalable to larger circuits, and, heuristics may largely affect the quality of results. Therefore, we are proposing another mapping method that is both scalable and optimal

$$\begin{pmatrix} I_{stim}1 \\ I_{stim}2 \\ I_{stim}3 \\ I_{stim}4 \end{pmatrix} = \begin{pmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \\ w41 & w42 & w43 & w44 \end{pmatrix} \cdot \begin{pmatrix} I_s1 \\ I_s2 \\ I_s3 \\ I_s4 \end{pmatrix}$$

Fig. 8. Matrix-vector multiplication

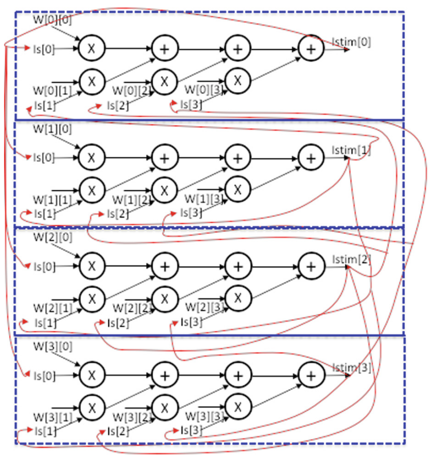

Fig. 9. Multiplication solution with global communication

for large regular computations. One of the important characteristics of regular computations, like matrix operations or convolutions on neural networks, is that the general flow of the operations is very similar or the same for different sizes of the problem. Therefore, if the optimal solution for the small size problem is known, its generalization to larger size problems is possible.

The general flow of the proposed mapping method for regular computations is as follows:

1. Map the smaller size problem optimally by an automatic method, like the proposed method in the previous section.
2. Use induction methods to generalize the solution of small problem for large one, semi automatically by human guidance.
3. Map the large problem optimally, automatically, utilizing the induction.

The generalization phase results in a set of constraints that basically limits the search space of the solver; consequently, larger problems can be solved very efficiently by an automated method. Following, the method is shown on two different examples.

5.1 Matrix-Vector Multiplication

Matrix-vector multiplication is one of the basic operations on matrices that have many applications in different domains including neural networks. Figure 8 shows the multiplication of a 4×4 matrix, and Fig. 9 shows its corresponding data flow graph that is mapped to 4 blocks. The flow of data among the blocks is like a ring connection, and the mapping is followed by the "natural" order of operations by a human. As shown in the figure, the mapping in this case is not optimal as it requires a lot of global data transfers among the blocks.

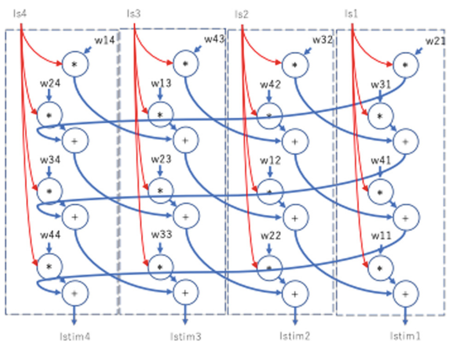

Fig. 10. Multiplication solution 1: local transfer of partial products

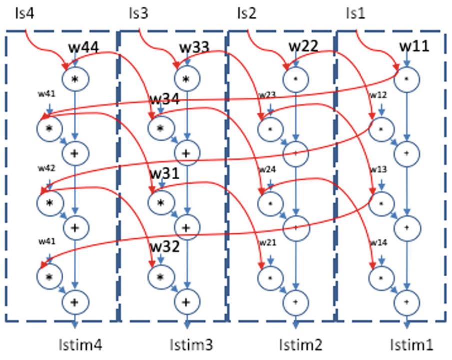

Fig. 11. Multiplication solution 2: local transfer of input vector

Following the proposed method, we have tried to find the optimal solution for this matrix-vector multiplication. Two different optimal results are obtained that are shown in Figs. 10 and 11. In the first solution, the partial products are propagated from one block to its neighbor block, while in the second solution, the input vectors are propagated.

The above observations regarding the flow of data in the solutions can be done by a human, and that knowledge can be used in a program to automatically generate and verify the solution for larger size matrices, as shown in the experimental results section.

5.2 Convolutional Neural Networks

Nowadays, convolutional neural networks (CNNs) are used in many applications; hence, acceleration of the convolution operations are very important. In this section, we present an optimal mapping of convolution operations in one convolution layer of CNN for image classification as an example.

Figure 12 shows a typical neural network for image classification. At first, a moving window of different sizes performs feature extraction on the input image, using convolution operations. At the end a fully connected neural networks performs image classification based on the extracted features from the image.

The convolution operation on a moving window usually requires a lot of processing power; hence, a target of acceleration. We have proposed a sequence of operations for a window, following ring connection-like order of operations as well as moving the window on the input image to maximize the utilization of processing element in each block as well as making data transfers locally. Figure 13 shows the order of operations for 4×4 window, though it can be extended to other window sizes. Figure 14 shows the data flow graphs for image of 4×4 and window size of 2×2. Similarly, it can be extended to any image size and any window size.

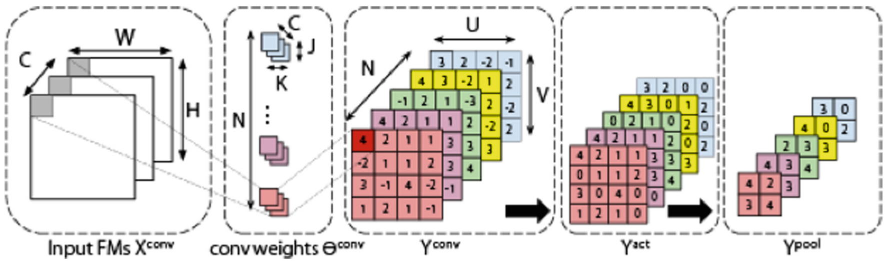

Fig. 12. An example CNN for image classification

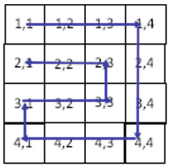

Fig. 13. Order of convolution operations: ring connection in mesh architecture

6 Experimental Results

In this section, we present our experimental results on mapping different DFGs into our proposed architecture. In our experiments, we have used DFGs obtained from ISCAS benchmark circuits and mapped them to our proposed hardware assuming LUTs in our blocks. Similarly, the DFGs may be obtained from other applications, and the block contents may be ALUs or other more complex processing elements, without affecting the mapping process or the mapping results. The process of obtaining DFGs is as follows:

1. The original circuit is synthesized and mapped into to netlist of LUTs with ABC tool [11]
2. The netlist of LUTs is converted to DFG as follows: each LUT in the netlist becomes a node in DFG, and for each connection in the netlist we have the corresponding edge in the DFG.

We convert DFGs to ILP formulas with our mapping program written in Python. For the experiments, we have used Gurobi optimizer [12]. All the experiments are performed on a server with Xeon E5 2.2 GHz processor and 512 GB memory running Linux kernel 4.16.

In the first experiment, we generated DFG for the combinational ISCAS benchmark circuits, as explained above. The DFG are then mapped to our hardware structure, as shown in Table 1. Columns 1 and 2 show the circuit name, and its number of inputs/outputs, respectively. Columns 3, 4, and 5 show the number of nodes, edges, and levels in the DFG, respectively. Columns 6 shows the size of the mesh structure. Columns 7 and 8 show the size of the ILP problem in terms of number of variables and number of constraints of ILP formula.

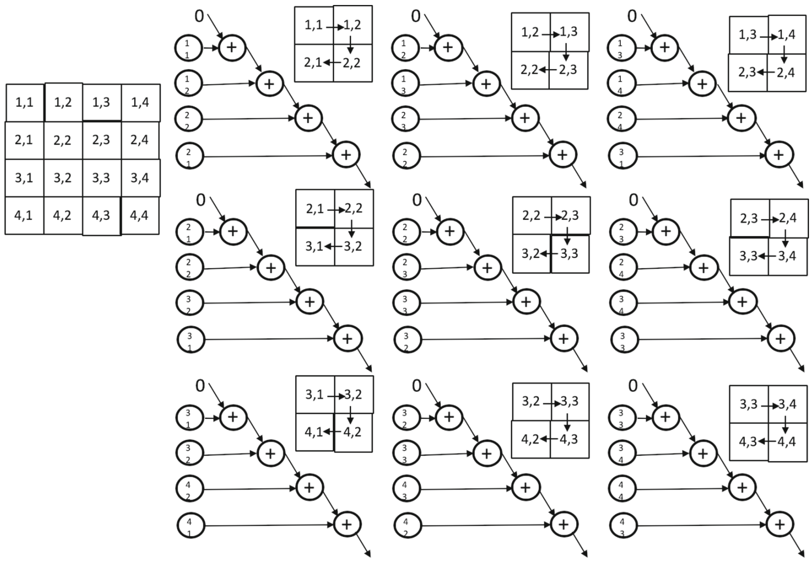

Fig. 14. Data flow graphs for 4×4 image and window size of 2×2

Table 1. Experiment 1: mapping of DFGs from ISCAS benchmark circuits

Circuit	DFG				Mesh size	ILP		Latency	Mapping time
	I/O	Nodes	Edges	Level		Var.	Const.		
s298	17/20	42	148	6	6×6	4,528	12,693	6	7.0 s
s344	24/26	44	147	6	6×6	5,484	13,615	6	17.4 s
s382	24/27	56	201	6	6×6	5,368	15,214	6	19.5 s
s400	24/27	55	203	6	6×6	5,336	15,409	6	19.0 s
s420	34/17	59	216	7	8×8	9,364	27,019	N/A	>1 day
s444	24/27	55	216	6	6×6	5,284	15,085	6	17.6 s
s510	25/13	98	353	7	8×8	11,692	41,840	7	32,664 s
s526	24/27	83	299	6	8×8	9,773	30,406	6	836.5 s
s641	35/24	79	234	9	8×8	31,356	75,939	N/A	>1 day

Column 9 shows the latency of the mapped circuit that is the number of clock cycles to finish executing the DFG on the target hardware. Column 10 shows the execution timr of the ILP solver to find the optimum solution.

As shown in the results, the complexity of finding optimum solution increases by increasing the number of DFG nodes as well as the number of I/O of the circuit. Therefore, for some circuits the optimum results could not be found even in 1 day.

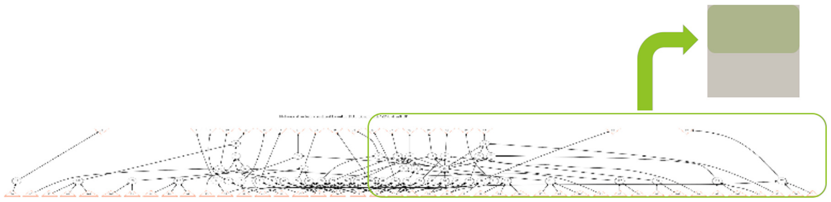

Fig. 15. Partitioning of the architecture into 2

In the next experiment, we tried to improve ILP solving time by partitioning the DFG into 2 sections, such that each section will be implemented in one of the 2 partitions of the hardware as shown in Fig. 15.

We have used hMETIS graph partitioning tool [13]. hMETIS tries to partition the graph into 2 subgraphs in a way that each subgraph has almost the same number of nodes, and the number of edges crossing a subgraph into another subgraph is minimum. In other words, the number of data transfers among the subgraphs is minimum, while almost uniformly distributing the DFG nodes. The results are shown in Table 2 for the four hardest cases. Comparing the results without graph partitioning and with graph partitioning, the latency is not increased while the execution time of ILP solver is reduced by an average more than one order of magnitude. Note that hMETIS is very fast and for all the DFGs finishes in less than 100ms. Moreover, for one of the cases that could not be solved because of timeout, the optimum solution is obtained. However, for one of the circuits we still cannot find optimum solution, and we need other heuristics to reduce the runtime further, and it is part of our future work.

In the next experiment, we used the proposed mapping method for matrix-vector multiplication. First, we tried the mapping for 4×4 matrix, as explained before, as the guidance for generalization to larger matrix operations. Then, we added constraints to the mapping program to limit the flow of operations for arbitrary size matrices similar to the case of 4×4. The results of mapping larger matrices are shown in Table 3. Without the generalization of the solution for 4×4 matrices, the problem for matrices larger than 8×8 could not be finished in one day. However, with the proposed generalization methods, matrix size of 32×32 can be mapped into 4 cores in less than 25 min.

Table 2. Experiment 2: mapping after partitioning original DFG to 2

Circuit	Mesh size	Sub-ILP		Whole-ILP		Latency	Mapping time (no partition)	Mapping time (w/partition)
		Var.	Const.	Var.	Const.			
s420	8×8	4,374	13,745	5,224	14,625	8	>1 day	18,936 s
s510	8×8	2,602	8,413	4,241	15,231	7	32,664s	1,415 s
s526	8×8	2,284	7,532	3,795	10,411	6	836.5s	40.0 s
s641	8×8	7,030	17,934	N/A	N/A	N/A	>1 day	>1 day

Table 3. Experiment 3: mapping of matrix-vector multiplication

Matrix dimension (N)	# cores (M)	Mapping time (s)
15	3	14
16	4	15
8	2	1.2
12	4	1.6
32	4	1437

7 Conclusions

In this work, we have presented a new reconfigurable architecture for IoT and mobile platforms. Our hardware consists of several logic blocks connected through a mesh network. The proposed hardware architecture provides a general platform for both fine- and coarse-grained computation. Compared to traditional FPGA devices, our proposed hardware can achieve easily very deep pipelining; hence, providing high performance computation.

In addition, we have proposed the general mapping method based on ILP formulation and heuristics to improve its efficiency. Our mapping method starts from the data-flow graph (DFG) of an application and results in optimal latency. The general mapping method works for any kind of computation. However, for regular computations such as matrix operations or convolutions on CNN, we have proposed a method scalable to large designs. The method is based on solving the problem automatically for small problem, generalize it by human, and solve the problem for larger problems automatically.

Our future work is improving the efficiency of the general mapping as well as experimenting with mapping high-level applications to coarse-grained blocks. Moreover, we are working on automation of the generalization process.

References

1. Rafique, A., Kapre, N., Constantinides, G.A.: Enhancing performance of tall-skinny QR factorization using FPGAs. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 443–450 (2012)
2. Aluru, S., Jammula, N.: A review of hardware acceleration for computational genomics. IEEE Des. Test **31**(1), 19–30 (2014)
3. Morcel, R., et al.: Minimalist design for accelerating convolutional neural networks for low-end FPGA platforms. In: International Symposium on Field-Programmable Custom Computing Machines (FCCM), p. 196 (2017)
4. Zhang, X., et al.: Machine learning on FPGAs to face the IoT revolution. In: International Conference on Computer-Aided Design (ICCAD), pp. 894–901 (2017)
5. Song, M., et al.: In-situ AI: towards autonomous and incremental deep learning for IoT systems. In: International Symposium on High Performance Computer Architecture (HPCA), pp. 92–103 (2018)

6. Farmahini-Farahani, A., Ho Ahn, J., Morrow, K., Sung Kim, N.: DRAMA: an architecture for accelerated processing near memory. IEEE Comput. Arch. Lett. **14**(1), 26–29 (2015)
7. Liang, S., Yin, S., Liu, L., Guo, Y., Wei, S.: A coarse-grained reconfigurable architecture for compute-intensive MapReduce acceleration. IEEE Comput. Arch. Lett. **15**(2), 69–72 (2016)
8. Tanomoto, M., Takamaeda-Yamazaki, S., Yao, J., Nakashima, Y.: A CGRA-based approach for accelerating convolutional neural networks. In: International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC), pp. 73–80 (2015)
9. Bae, I., Harris, B., Min, H., Egger, B.: Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **37**(11), 2301–2310 (2018)
10. ISCAS'89 benchmarks. http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/
11. Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/
12. Gurobi optimization. www.gurobi.com
13. hMETIS - Hypergraph & Circuit Partitioning. http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview