Title

A Fully Pipelined and Dynamically Composable Architecture of CGRA (Coarse Grained Reconfigurable Architecture)

Permalink

https://escholarship.org/uc/item/9446s3nx

Author

Zhou, Peipei

Publication Date

2014

Peer reviewed|Thesis/dissertation

A Fully Pipelined and Dynamically Composable Architecture of CGRA (Coarse Grained Reconfigurable Architecture)

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Electrical Engineering

by

Peipei Zhou

2014

Abstract of the Thesis

A Fully Pipelined and Dynamically Composable Architecture of CGRA (Coarse Grained Reconfigurable Architecture)

by

Peipei Zhou

Master of Science in Electrical Engineering

University of California, Los Angeles, 2014

Professor JINGSHENG JASON CONG, Chair

Future processor will not be limited by the transistor resources, but will be mainly constrained by energy efficiency. Reconfigurable architecture offers higher energy efficiency than CPUs through customized hardware and more flexibility than ASICs. FPGAs allow configurability at bit level to keep both efficiency and flexibility. However, in many computation-intensive applications, only word level customizations are necessary, which inspires coarse-grained reconfigurable arrays(CGRAs) to raise configurability to word level and to reduce configuration information, and to enable on-the-fly customization. Traditional CGRAs are designed in the era when transistor resources are scarce. Previous work in CGRAs share hardware resources among different operations via modulo scheduling and time multiplexing processing elements. In the emerging scenario where transistor resources are rich, we develop a novel CGRA architecture that features full pipelining and dynamic composition to improve energy efficiency and implement the prototype on Xilinx Virtex-6 FPGA board. Experiments show that fully pipelined and dynamically composable architecture(FPCA) can exploit the energy benefits of customization for user applications when the transistor resources are rich.

The thesis of Peipei Zhou is approved.

DEJAN MARKOVIC

MILOS ERCEGOVAC

JINGSHENG JASON CONG, Committee Chair

University of California, Los Angeles

2014

Table of Contents

List of Figures

List of Tables

Acknowledgments

CHAPTER 1

Introduction

The classic von Neumann architecture allows efficient sharing of the executions of different instructions on a common pipeline, providing an elegant solution when the transistor resources are scarce. As the number of transistors in a chip is increasing exponentially, future processor chips are no longer limited by the transistor resources, but are mainly constrained by energy efficiency. Therefore, a fundamental departure from the von Neumann architecture is likely required to achieve much higher energy efficiency.

One architectural trend is to implement the massive transistors available on a chip into a sea of heterogeneous accelerators [FXB10, LHW12, CGG12a, CCG13]. At design time, each accelerator is customized in ASIC for a single application which is frequently used in the application domain of the target users. By offloading most computation tasks from general-purpose CPU cores to accelerators, an accelerator-rich architecture can bring 10-100x energy efficiency [CSR11, CCG13]. However the low flexibility of ASICs leads to the limited workload coverage of the processor chip and the high nonrecurring engineering cost of application algorithm updates.

A more promising development trend is reconfigurable architectures to keep both flexibility and energy savings. Field programmable gate arrays (FPGAs) have been widely used as hardware accelerators for different applications by customization before task executions [Xild]. However their bit-level reconfigurability is often unnecessary in most computation-intensive applications. Only word-level customizations are necessary if the target application needs to keep the full precision. Coarse-grained reconfigurable architectures (CGRAs)

is introduced to raise the reconfigurability to word level, and thus to reduce configuration information and to enable on-the-fly customization. They are composed of a sea of word-level processing elements that include ALUs, SRAMs and DMAs [Har01]. These components are connected together by word-level data interconnects and controlled by synchronization modules and resource managers.

CGRAs can be categorized in two classes. The first class is tightly coupled CGRAs, e.g., Chess [MSK99], Matrix [MD96] and DySER [GHN12]. They are designed as a part of a CPU's pipeline and act as enhanced execution units to allow custom instructions. They experience limited benefits of a full customization. The second class is loosely coupled CGRAs, e.g., PipeRench [GSB00], MorphoSys [SLL00] and CHARM [CGG12b]. They act as co-processors as peers to CPUs. They can be customized in a larger design space than tightly-coupled CGRAs and achieve higher energy efficiency. Our work belongs to the second class.

The fact is that CGRAs were originally proposed at a time when transistor resources were much more constrained than energy consumption. Existing CGRAs follow the design style developed in the past, and time-multiplex the PEs among multiple operations. Modulo scheduling is performed to map the operations in a user application to the limited number of PEs in a CGRA. Each PE contains a configuration RAM to store the assigned multiple instructions and switches during execution. With the exponential increase of the transistor count on a chip following Moore's law, however, it is now possible to include a massive number of PEs in a CGRA. In many cases, it is possible to map all the operations in a user application kernel loop to different PEs for the maximum parallelism with no resource conflict. The primary design target of CGRAs will no longer be hardware sharing given the limited area constraint, but rather achieving the highest energy efficiency under the rich transistor resources. In this paper we propose a novel CGRA that pursues the new design target with the following two features:

- *Full Pipelining.* Given the rich transistor resources, we can assign each operation

to a separate hardware module and accommodate all the operations from a user application in our CGRA without suffering the extra cost of hardware sharing. The maximum throughput under this mapping is one innermost loop iteration per clock cycle. To achieve this throughput for energy savings, new design challenges emerge, e.g., contentions for on-chip memory ports and data interconnects. In addition, the conventional design of homogeneous PEs leads to low utilization in the scenario with no hardware sharing. All of these issues are solved in our new CGRA design.

- *Dynamic Composition.* When the rich transistor resources are implemented into a large CGRA, a single user application will occupy only a small part of a CGRA. There can be many idle resources left. In this work, we dynamically compose, as in [CGG12b], as many copies of accelerators as possible from the idle resources to further improve the system throughput. The challenge is that the placement of the PEs to compose these copies of accelerators does not necessarily guarantee routability in a conventional mesh-based architecture and may need many trials during the runtime composition. In this work we propose a scalable architecture with high connectivity to significantly reduce the effort on placement and route.

We name our new architecture *FPCA* (Fully Pipelined Composable Architecture). We implement a working prototype of our FPCA on top of a commodity FPGA board with the goal of providing early experience for future ASIC implementation. We also develop both a design automation flow and a compiler for FPCA, and use these tools to perform the design space exploration. Experiments show that our FPCA can fully exploit the energy benefits due to customization for user applications in the scenario of rich transistor resources.

CHAPTER 2

Background

2.1 Conventional CGRA

Conventional CGRA features word-size ALUs, multipliers and bus interconnect. In [GSB00], the PipeRench architecture features reconfigurable fabric that can be divided into physical pipeline stages. PipeRench's compiler can compile the design into virtual stages that can be mapped onto any physical pipeline stage. Fig. 2.1 shows how virtual stages are mapped to
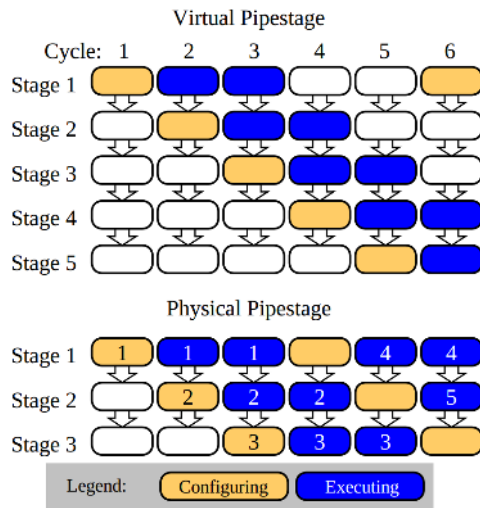
Figure 2.1: Pipeline reconfiguration showing how 5 virtual stages are mapped to 3 physical stages: (a) the virtual pipeline stages, (b) the physical pipeline stages

physical stages. The top portion shows a five-stage application and the state of each stage of the pipeline in the consecutive five cycles. The bottom portion shows the state of the physical stages in the fabric that is executing the application. Once the pipeline is full, two

results will be generated from the pipeline every five cycles. A row of PEs is used to create a physical stage of the pipeline, also called a physical stripe, as shown in Fig, 2.2.
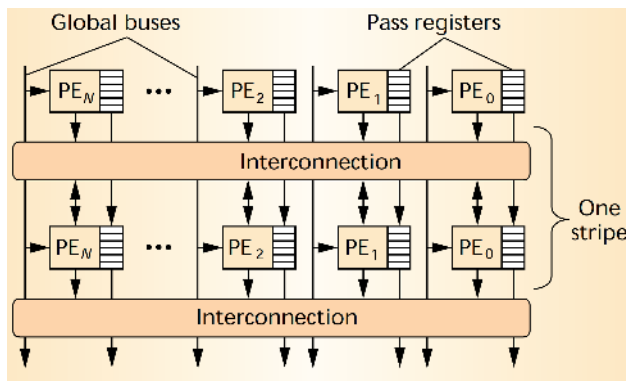
Figure 2.2: PipeRench architecture: PEs and interconnect

The limitation of PipeRench is that the state of any pipeline stage must be a function of the current state of only that stage and the previous stage, so that the applications that PipeRench supports are constrained.

MorphoSys [SLL00] is a reconfigurable computing system which includes a reconfigurable processing unit (RC array), a general-purpose (core) processor (TinyRISC), and a high bandwidth memory interface. The overall chip architecture is shown in Fig. 2.3.
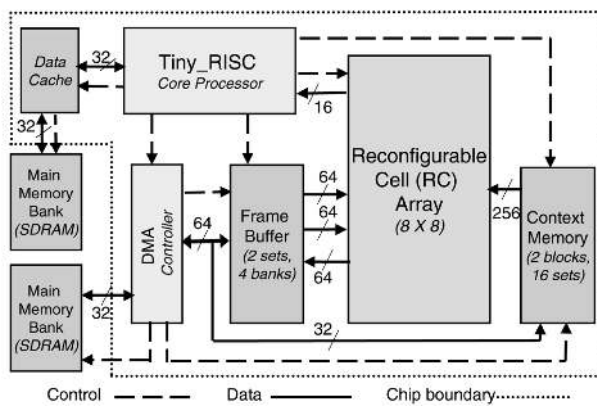
Figure 2.3: MorphoSys chip

Each RC incorporates a 28-bit ALU, a 16 * 12-bit multiplier, a shift unit, 16-input and

5

8-input multiplexers and a 16-bit register file. A 32-bit context register stores the current context word and provides configuration signals for the RC functional unit. RC cells are connected through 2D mesh network, as shown in Fig. 2.4. The RISC processor executes the unaccelerated part of the application and offloads some portion of the application onto the RC array.
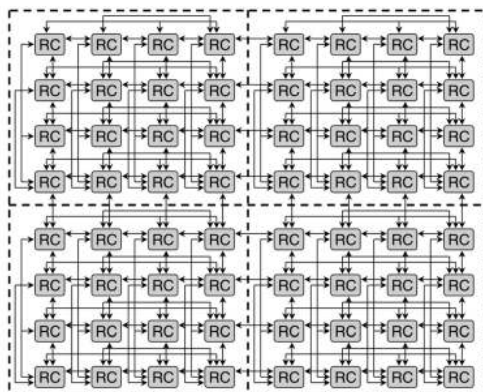
Figure 2.4: MorphoSys 8*8 array with 2D mesh and complete row/column connectivity per quadrant

Morphosys supports dynamic composition. However, configuring the RC array on the mesh network has an routing time overhead that is not negligible.

ADRES [MVV03] features a tightly coupled VLIW/CGRA processor, as shown in Fig. 2.5. The ADRES consists a matrix of basic components that includes functional units (FUs) and register files(RFs). The FUs can execute word-level operations, and RFs can store intermediate data. An example of a reconfigurable cell is shown in Fig. 2.6.

The ADRES matrix has two functional modes, VLIW processor modes and reconfigurable function modes. For the VLIW processor, FUs in the first row are configured and connected together through multi-port register files. For the reconfigurable function modes, the function units will load the configuration information that is stored locally on a cycle-by-cycle basis. When applications are mapped onto ADRES, outer loop and acyclic code are executed on the VLIW processor, and the innermost loop is executed on the matrix of reconfigurable
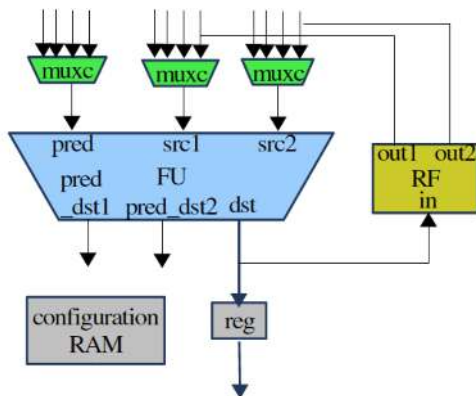
Figure 2.5: ADRES core

functional units (FUs). ADRES uses modulo scheduling to compile application code onto the hardware architecture and compute II based on a data flow graph and existing hardware resources. However, the RAM that stores the cycle-by-cycle configuration information is a big overhead. As reported in [BBK07], the configuration memories have a fixed depth of 128 words and consume 36% of the total area.

The PPA (Polymorphic Processor Array) [PPM09] exploits coarse-grain pipeline parallelism found in streaming applications and fine-grain parallelism through modulo scheduling of innermost loops. The architecture of the PPA loop accelerator is shown in Fig. 2.7. PPA supports dynamic reconfiguration by pre-storing different execution configuration information. Thus, PPA can execute an innermost loop with a varing number of function unit resources during run-time. However, the run-time modes are limited by the number of pre-stored execution modes.

[BDV08] proposes CGA(coarse-grained array accelerator) for software-defined radio baseband processing. The CGA accelerator is based on the ADRES [MVV03] architecture template, and its top-level architecture is shown in Fig. 2.8.

The CGA accelerator is tightly coupled with the main CPU by sharing a larger register. The compiler partitions the loop code into *preloop code*, *loop invocation code* and *postloop code*. *preloop code* and *postloop code* are compiled into binary code for the main CPU, while

7
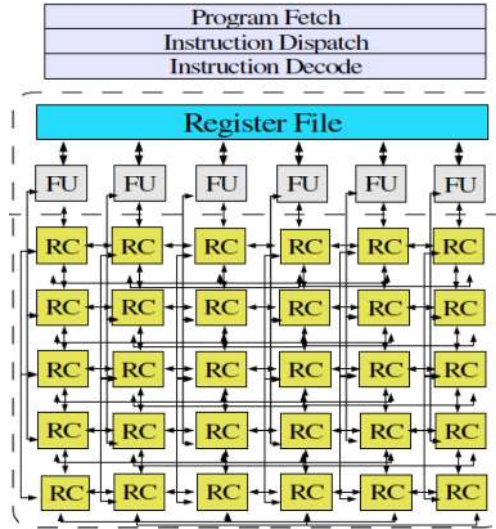
Figure 2.6: Reconfigurable Cell of ADRES

loop invocation code is compiled for the CGA. However, as with ADRES and PPA, CGA uses 2D mesh network. Thus, the placement of the functional units to compose the accelerators does not necessarily guarantee routability in this conventional mesh-based architecture, and it may need many trials during the runtime reconfiguration.
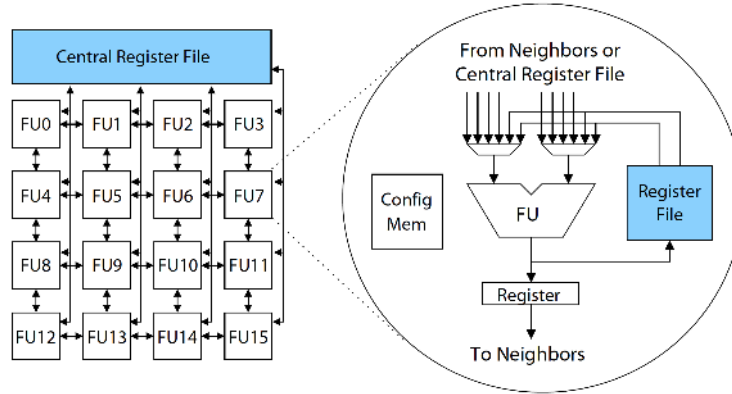
Figure 2.7: PPA loop accelerator
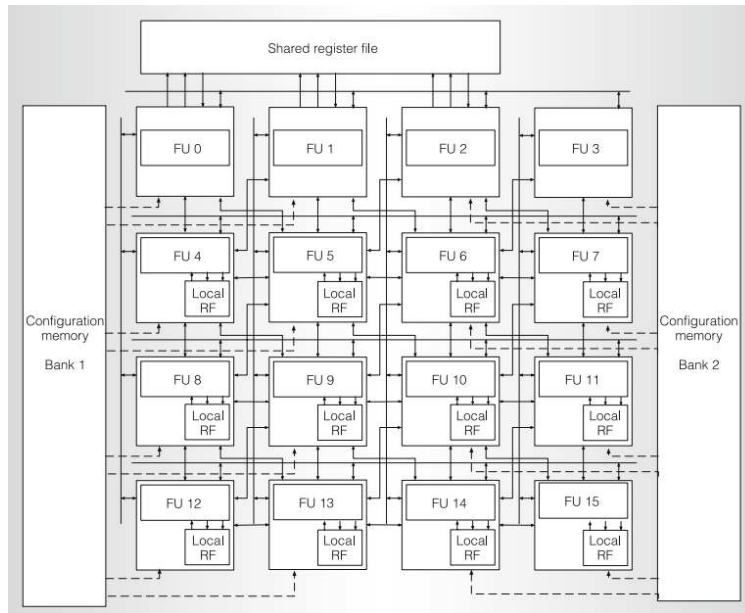
Figure 2.8: 16 functional units (FUs) are connected to one shared and 12 local register files (RFs) through a reconfigurable interconnect, and configurations are fetched from the configuration memory banks.

9

CHAPTER 3

The FPCA Architecture

3.1 Programming Model

It is common that a reconfigurable architecture focuses on the workloads with the potential of hardware acceleration. These kinds of workloads have regular computation patterns and data access patterns so that a reconfigurable architecture can be customized for these patterns before execution.

In this initial study, we use the programming model of a two-level data flow graph (DFG).
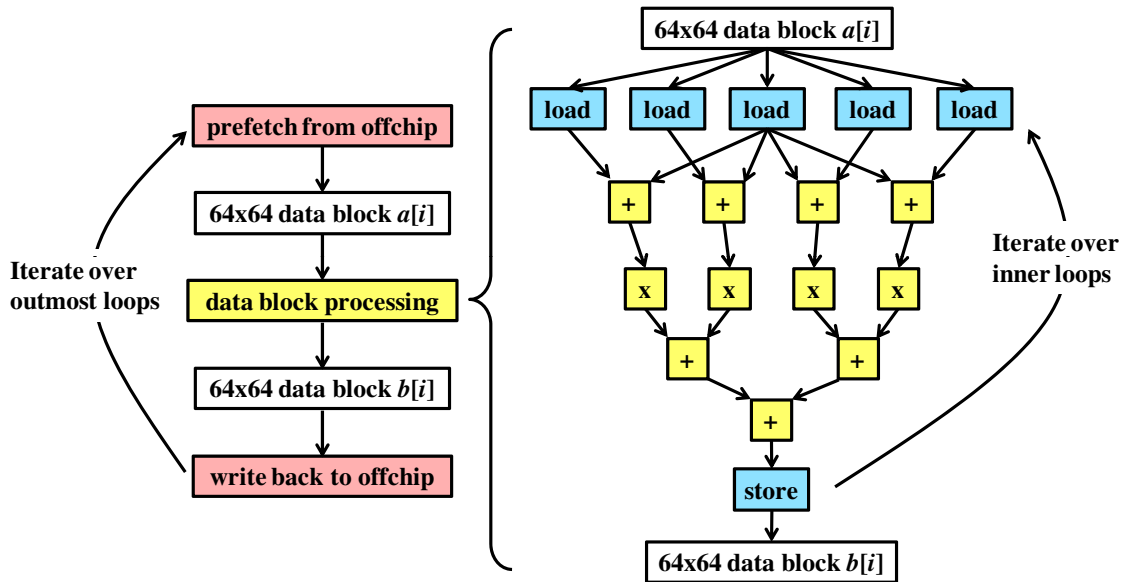
Figure 3.1: An example of the programming model based on a two-level data flow graph (DFG) supported by our FPCA.

```
#define SQR(x) ((x)*(x))
for( int i = 0; i < 64; i++ )
    for( int j = 1; j < 63; j++ )
        for( int k = 1; k < 63; k++ )
            b[i][j][k] =
                SQR(a[i][j][k] - a[i][j][k-1]) +
                SQR(a[i][j][k] - a[i][j][k+1]) +
                SQR(a[i][j][k] - a[i][j-1][k]) +
                SQR(a[i][j][k] - a[i][j+1][k]);
```

Figure 3.2: The original user code of a workload from which the DFG in Fig. 3.1 is transformed.

Fig. 3.1 is an example of our model which is transformed from the original user code in Fig. 3.2. It uses the 'gradient' benchmark in the application domain of medical imaging [CSR11]. The top level of the DFG, i.e., the leftmost part of Fig. 3.1, starts from the prefetch of data blocks from the main memory to on-chip memories. The on-chip memories here act as scratch-pad memories (SPMs) that can be managed by software to avoid the overhead of cache misses in CPU execution. The data stored in on-chip memories can be reused by multiple load operations to save off-chip communication. For example, there are five load operations on array a in the innermost loop. But each element in array a will need to be fetched from off-chip only once if we allocate a 64×64 SPM storage for data block $a[i]$. This kind of SPM insertion for data reuse can be automated by analysis of data access patterns in user applications [CHL11]. After data block prefetching, the DFG in the leftmost part of Fig. 3.1 begins processing the data block. This processing can be represented by the bottom-level DFG, as shown in the rightmost part of Fig. 3.1. It starts from load operations on the input data block, goes through a number of arithmetic operations, and ends with store operations on the output data block. This DFG will be iterated over the innermost loops in the original user code to process the whole data block prefetched from off-chip and to generate the output data block. In the example code in Fig. 3.2, the bottom-level DFG is iterated over j and k in the range from 1 to 63. Then the generated data block will be written back from on-chip memories to the main memory, as shown in the leftmost part of Fig. 3.1. The top-level DFG will be further iterated over the outermost loops of the original user code,

e.g., i in the range from 0 to 64 for the example code in Fig. 3.2. We can see that the key to our programming model is to model loops, since loops are the source of huge workloads where hardware acceleration and energy savings become meaningful. By customizing hardware for the operations in the loop only once, we can execute the hardware for thousands or even millions of times and enjoy the energy savings brought by customization.

We do not require users to directly code using our programming model. Instead, the compiler of our FPCA will check whether original user codes, e.g., the ANSI C codes in Fig. 3.2, can be transformed to our programming model, and perform such transformation if possible. Details about our compiler can be found in Section 3.7.

3.2 Design Principle

Here, We specify the design principles for FPCA architecture.

3.2.1 Full Pipelining

During the design of a hardware accelerator, there is a trade-off between hardware throughput and resource usage. To implement the operations of the innermost loop in Fig. 3.2, we can allocate a separate processing element (PE) for each operation and get an architecture that completes a new loop iteration every clock cycle. We can also reduce the throughput to one loop iteration every two clock cycles and use time multiplexing to reduce the hardware usage to only half of the adders and multipliers. This reduction is controlled by the difference in the starting clock cycles of two adjacent loop iterations, i.e., loop pipeline initial interval (II). Fig. 3.3 shows the hardware resource reduction versus the increase of pipeline II. The hardware resources are measured by the number of FPGA LUTs (and also FFs and DSPs) reported by the Xilinx Vivado high-level synthesis toolkit [CLN11, Xilc].

As shown in Fig. 3.3, when the II is increased from 1 to 2, though the throughput is reduced by half, the resource usage is reduced by only 20-40%. This synthesis result show
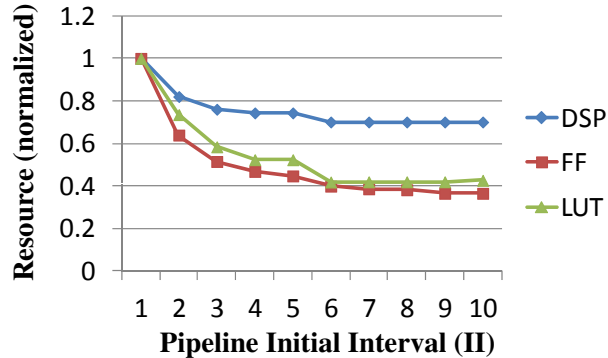
Figure 3.3: Resource usage versus loop pipeline initial interval (II).

that, when an accelerator is designed with a pipeline II equal to 1, it achieves the highest ratio of performance over resource usage. It also means achieving the lowest energy consumption if the power is assumed to scale with the number of active logic gates.

In addition, for denoise kernel in medical imaging applications, we sweep the target clock period from 4ns to 10ns, II from 1 to 4, measure the achieved clock periods and different resource usages. We plot processed words/slice/second, words/LUT/second, words/FF/second and words/DSP/second in Fig. 3.4. The results show that, under different target clock periods, when II = 1, it always achieves the best performance/area.

The insight behind this is that the time multiplexing in the design with a large pipeline II is not perfect. The number of operations of the same type is not always divisible by the pipeline II. When the pipeline II is sufficiently large, we still need at least one PE for each type of operation that appears in the user application. In addition, the time multiplexing costs extra logics to store more pipeline states in a design with a larger pipeline II. Time multiplexing will also cost more energy since the data path of each PE has to switch among the multiple operations assigned to it every clock cycle. In the past decades when transistor resources were scarce, time multiplexing was a reasonable design choice. Because our FPCA faces an emerging era with rich transistor resources but demanding energy efficiency, the full pipelining (i.e., II= 1) of the architecture becomes the optimal choice.

13
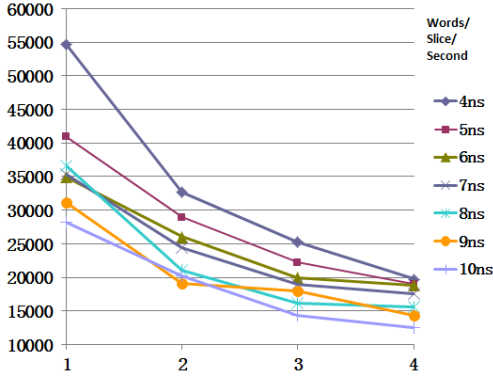
(a) Words/Slice/Second under different IIs
and clock periods
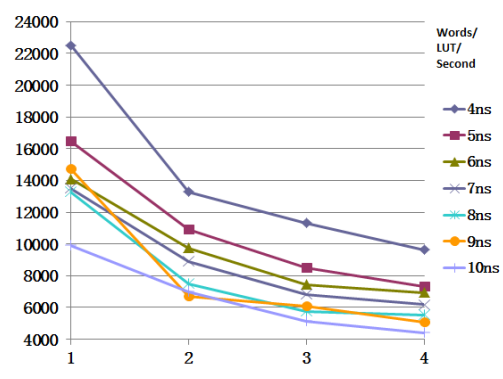
(b) Words/LUT/Second under different IIs
and clock periods
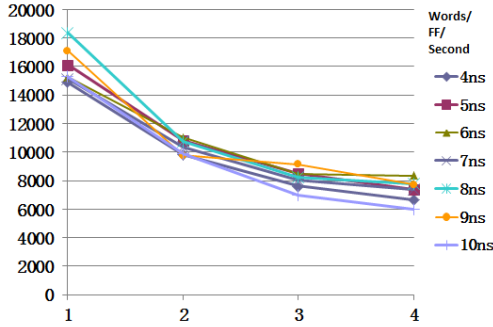
(c) Words/FF/Second under different IIs
and clock periods
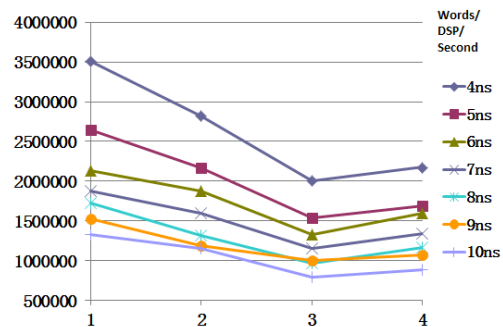
(d) Words/DSP/Second under different IIs
and clock periods

Figure 3.4: performance/area for denoise kernel under different IIs and clock periods

Last but not least, a formal proof of best performance/area of full pipelining is provided
in Appendix A.

3.2.2 Dynamic Composition

When the transistor resources are rich, a single application may occupy only a small part of
the reconfigurable array.

(a) Compose accelerators for two applications from the reconfigurable array. (b) Dupli-
cate multiple copies of accelerators for a single application from the reconfigurable array.

14
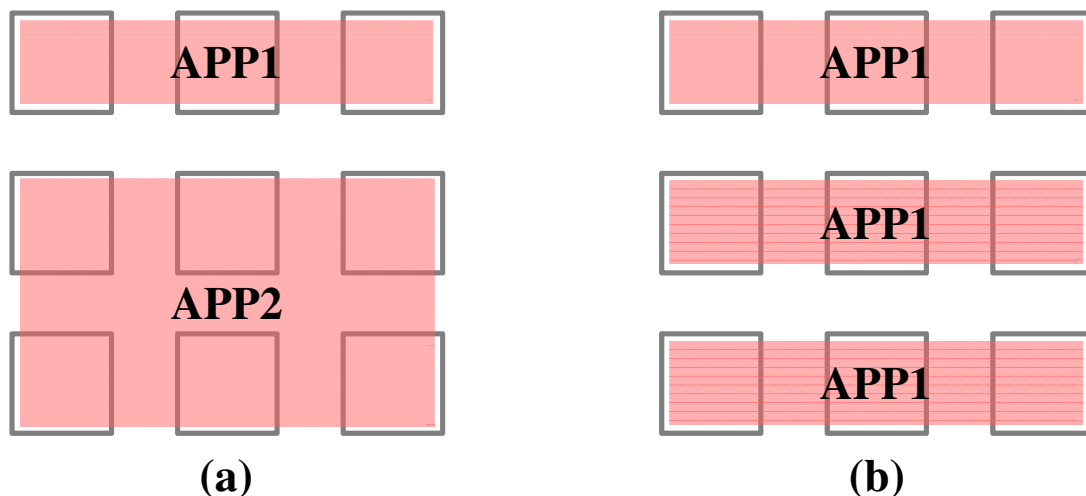
Figure 3.5: (a) Compose accelerators for two applications from the reconfigurable array. (b) Duplicate multiple copies of accelerators for a single application from the reconfigurable array.

This allows us to share the reconfigurable array among multiple applications called by the user during runtime, as shown in Fig. 3.5(a). Even if there is only one application executed by the user, we can use the idle resources to compose multiple copies of the accelerator for the application from the reconfigurable array to further improve the overall throughput, as shown in Fig. 3.5(b). We can implement a runtime scheduler that maintains the status table of all the PEs (ALUs, on-chip memories, and DMACs) in the array, and map the incoming application to idle resources according to the table, as proposed in [CGG12b]. This kind of dynamic composition can significantly improve the utilization of hardware resources. It makes a reconfigurable architecture even more efficient. The main challenge of dynamic composition lies in the routing of the logic resources. The logic resources which are idle in the reconfigurable array will be different at each time of application mapping. The placement of the mapping result will change, even for the same application. The routing for a new placement has to be performed again, which is a time-consuming negotiation-based process and is not guaranteed to succeed in a conventional mesh-based architecture. In addition,

the programmable interconnects are global resources. The interconnects in the local region under routing may have already been occupied by other applications in adjacent regions. This makes the routing in the scenario of dynamic composition even harder. The programmable interconnects in our FPCA are specially redesigned for the purpose of dynamic composition. Details can be found in Section 3.5.4.

3.3 Overview of Architecture

The architecture overview of our FPCA is shown in Fig. 3.6.
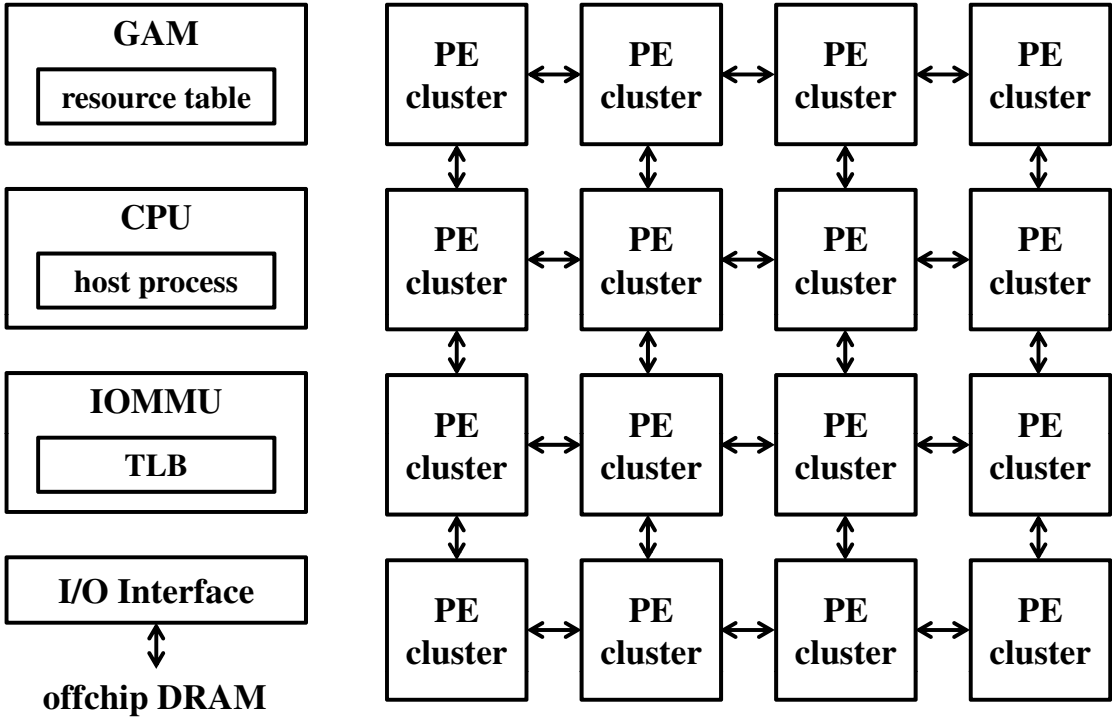
Figure 3.6: FPCA architecture overview

Our FPCA is a complete system-on-chip except for the main memory. It contains one or several general-purpose CPU(s) that run the host process of user applications. The host process will execute the general-purpose operations that are more friendly to CPUs than hardware accelerators. It will also send the computation tasks that have been compiled for

16

our FPCA acceleration to the global accelerator manager (GAM). The GAM, first introduced in [CGG12a], maintains a status table of all the resources in our FPCA and maps the incoming tasks to idle resources in the reconfigurable array.

The main part of the architecture is an array of processing element (PE) clusters.
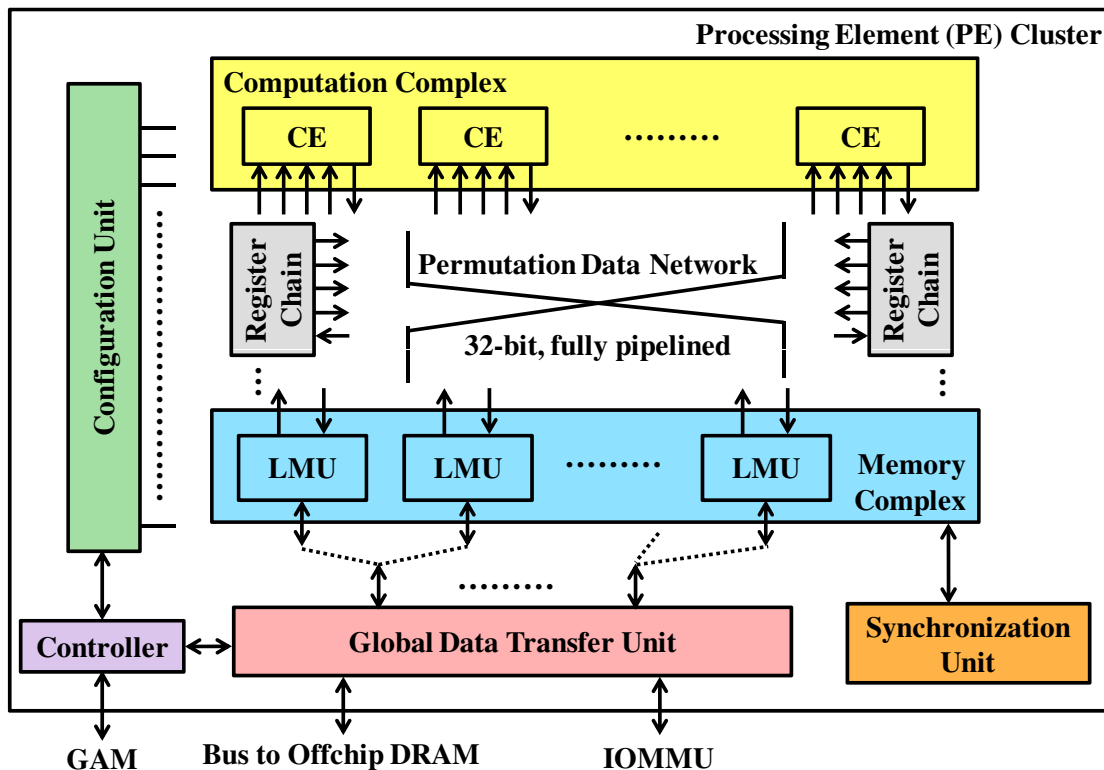
Figure 3.7: Internal structure of a processing element(PE) cluster

The internal structure of a PE cluster is shown in Fig. 3.7. Each cluster contains a set of 32-bit heterogeneous PEs including computation elements (CEs), local memory units (LMUs) and register chains to act as ALUs, on-chip buffers and registers respectively. They are connected by a permutation network which can be customized for the arbitrary topology of the application DFG. There is also a global data transfer unit (GDTU) in each PE cluster to transfer data between LMUs and the main memory. It contains several channels, each connected to all the LMUs to allow broadcasts. The synchronization unit enforces the time sequence of loop pipelining. The controller initiates a computation task, monitors its

17

execution, and reports back to the GAM. The configuration unit provides constant configuration bits to all the modules after dynamic composition of accelerators. Details about these modules can be found in Section 3.5. All the PE clusters are organized in a mesh with neighbor-to-neighbor (N2N) connections, as shown in Fig. 3.6. We can see that our FPCA is a two-level architecture where PEs are first connected by a permutation network with a high connectivity within a cluster, and then by a global N2N network for more scalable connectivity. A single application can usually fit into a PE cluster with the guaranteed routability, and the challenge of dynamic composition in respect to the routing is much alleviated. The global mesh keeps the scalability of our architecture.

We allow user applications to run under an operating system (OS) with a virtualized memory space. As shown in Fig. 3.6, our FPCA includes an input/output memory management unit (IOMMU) connected to all the PE clusters. It processes data transfer requests with both the starting virtual addresses and sizes of data blocks sent from a GDTU in a PE cluster. The IOMMU contains a translation lookaside buffer (TLB) for page translation from virtual addresses to physical addresses, and may consult with the OS in case of a TLB miss. It also cuts a multi-dimensional data block on its page boundaries and returns the GDTU with a set of direct memory accesses with continuous addresses. The GDTUs in all the PE clusters are also connected via a system bus to the I/O interface coupled with the off-chip DRAM to execute the direct memory accesses.

3.4 Example Execution Flow

This section gives an example of how the application in Fig. 3.2 is executed in our FPCA.

3.4.1 Composition

In the first step, the GAM will try to map all the nodes in Fig. 3.1 to separate idle modules in PE clusters to compose a copy of a hardware accelerator, e.g., APP1, as shown in Fig. 3.8.
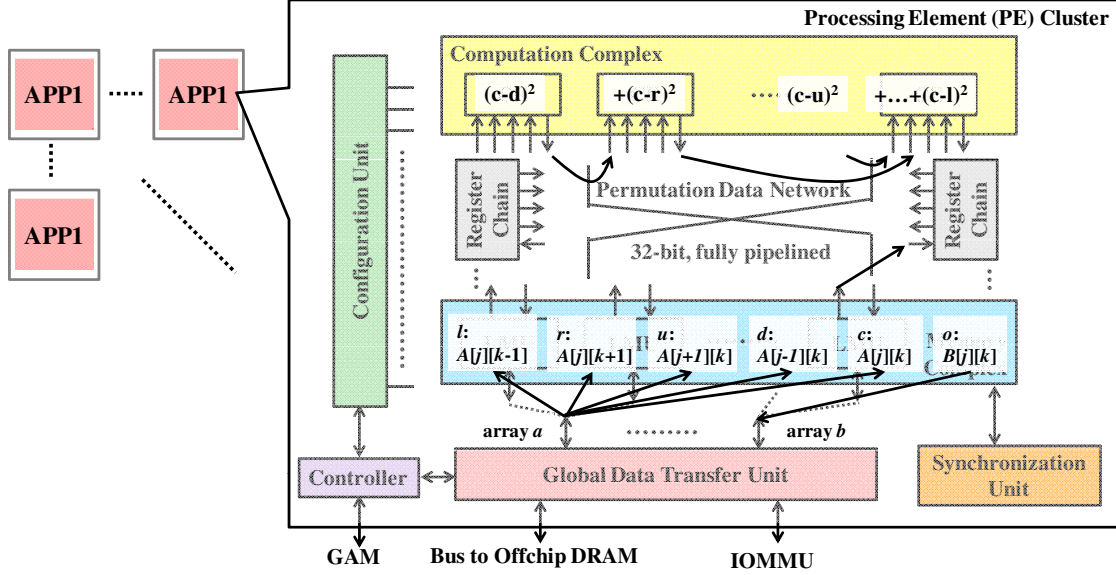
Figure 3.8: Mapping result of the example in Fig. 3.1

The prefetch of the input data block $a[i]$ and the write-back of the output data block $b[i]$ are mapped to the two channels in the GDTU. The five load operations ($A[j-1][k]$, $A[j+1][k]$, $A[j][k+1]$, $A[j][k+1]$ and $A[j][k]$), and the store operation ($B[j][k]$) are mapped to the six LMUs (data blocks $a[i]$ and $b[i]$ are denoted as A and B respectively). The arithmetic operations are mapped to the four CEs. Since there is only one application called by the host CPU, the GAM keeps duplicating APP1 until all the resources are occupied. All the copies of APP1 will run in parallel.

3.4.2 Parallel Execution of Subtasks

We exploit the coarse-grained parallelism among the iterations of the outermost loops for the top-level DFG in the leftmost part of Fig. 3.1.

This exploitation is illustrated in Fig. 3.9; it denotes the steps of executing parallel subtasks in the application of Fig. 3.2. A descriptions follows:

1. *Subtask distribution.* After accelerator composition, the runtime scheduler in the GAM

subtask 0: | page translation for $a[0]$ and $b[0]$ | prefetch $a[0]$ from offchip | compute $a[0]$ for $b[0]$ | write back $b[0]$ to offchip |

subtask 1: | page translation for $a[1]$ and $b[1]$ | prefetch $a[1]$ from offchip | compute $a[1]$ for $b[1]$ | write back $b[1]$ to offchip |

subtask 2: | page translation for $a[2]$ and $b[2]$ | prefetch $a[2]$ from offchip | .... |

subtask 3: | page translation for $a[3]$ and $b[3]$ | .... |

....

time
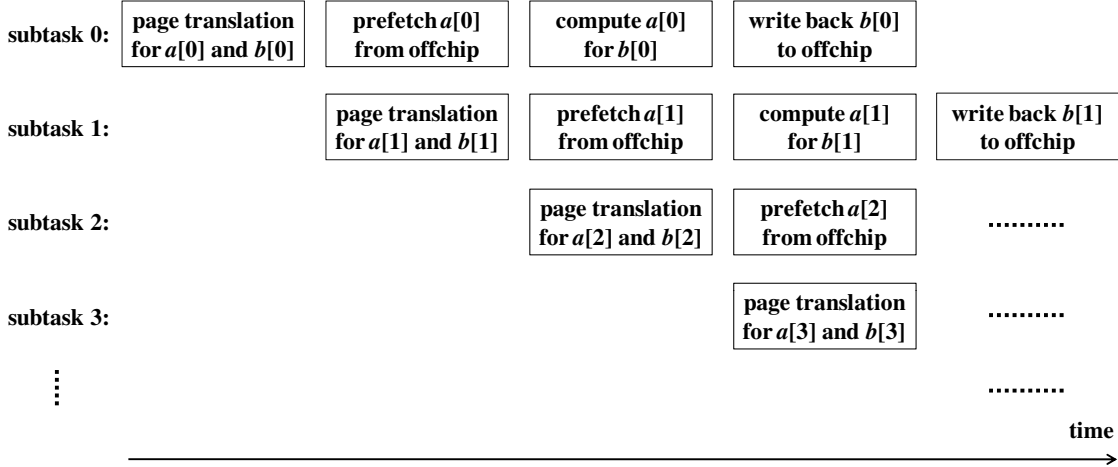
Figure 3.9: Parallel execution of subtasks.

will decompose the task of the target application into independent subtasks, e.g., subtask $0, 1, \cdots, 63$ for each enumeration of iterator i in Fig. 3.2. Then the GAM distributes the subtasks to all the copies of APP1; let's say the first copy gets the list of subtask $0, 1, \cdots, 15$. The GAM also informs each copy of APP1 of the starting virtual addresses of the data arrays in user applications.

2. *Page translation.* The GDTU in the copy of APP1 will initiate data transfer requests based on the subtask IDs i and the starting virtual addresses of the data arrays a and b. It will send a set of requests of data blocks $a[0], a[1], \cdots$ and $b[0], b[1], \cdots$ to the external module IOMMU for page translation, and will wait for the response.

3. *Prefetch the first data block.* Once the request of $a[0]$ returns, the GDTU channel mapped with $a[i]$ will transfer the 64×64 data block $a[0]$ from the main memory to the five LMUs.

4. *Process the first data block, and concurrently prefetch the second data block.* After the whole data block $a[0]$ is transferred, the five LMUs will load the data elements in $a[0]$ according to their associated access patterns and push them to the network of CEs. The LMU mapped with $B[j][k]$ will receive the results to fill up the data block $b[0]$. In

20

our FPCA, LMUs are equipped with double buffering to allow overlap of computation and off-chip communication. In parallel with CE execution, the five LMUs will flip the two memory spaces in them so that $a[1]$ can also be transferred by the GDTU channel (if the request of $a[1]$ has returned from the IOMMU).

5. *Process the second data block, and concurrently prefetch the third data block and write back the first data block.* After the data block $b[0]$ is fully computed, the GDTU channel mapped with array b will transfer the data block from the LMU to the main memory. At the same time, the computation of $a[1]$ for $b[1]$ and the transfer of $a[2]$ from off-chip will be performed as well, if the transfer of $a[1]$ finishes. The parallel executions of
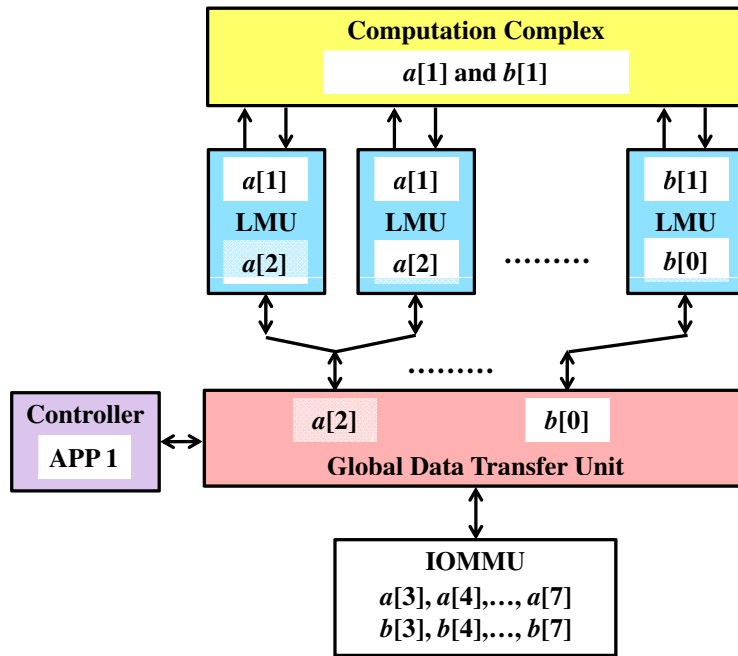
Figure 3.10: The independent data blocks simultaneously processed by different modules at a point in time.

page translation, computation and off-chip read and write at this stage is shown in Fig. 3.10.

6. *Continue the parallel execution until completion.* Fig. 3.10 will be repeated on the

21

following subtasks until the copy of ACC1 finishes all the subtasks $0, 1, \cdots, 15$. Sometimes the off-chip communication can stall since the main memory is a shared resource and may be kept busy by other devices. In this case, the computation will automatically stall to keep the synchronization with the off-chip communication, as specified in Fig. 3.9. The double buffering of our LMU is implemented as a FIFO with depth equal to 2 and the granularity equal to a data block. For example, if a data block is not consumed by the write back of the GDTU, it will keep the FIFO full to prevent any further computation until the GDTU transfers the data block and commits to the FIFO.

3.4.3 Parallel Processing of Data Elements

The bottom-level DFG in the rightmost part of Fig. 3.1 is executed by LMUs, CEs, register chains and the permutation network together. All the nodes in the DFG are mapped to separate modules, and the parallelism among the operations within an innermost loop iteration is automatically exploited. For example, the four multiplications are performed in parallel by four CEs. The parallelism among the operations across different innermost loop iterations is also exploited. All the modules are fully pipelined, i.e., taking a new data at each input every clock cycle and sending a new result at each output every clock cycle. It means that while a module is processing the data element for an iteration $j = 1$ and $k = 2$, its precedent module is performing the load operation on the data element for the next iteration $j = 1$ and $k = 3$. With this pipeline mechanism applied to every module, even if the bottom-level DFG is large, we could achieve the throughput of one loop iteration every clock cycle. A large DFG will only occupy more hardware resources.

In conventional CGRAs [GSB00,SLL00,GHN12], the communication among PEs is a big overhead. Each data element sent out or coming in has to go through the flow control to check whether there is space at its destination. We solve this problem by taking advantage of the fact that, after mapping the DFG to all the modules, the delay from the input LMUs

of the DFG to the output LMUs is determined. Therefore we do not need to implement the flow control with any data address or valid bit in CEs, register chains or the permutation network. These modules just sense their inputs every cycle and send out the corresponding results after their intrinsic delays. To guarantee that the correct results are stored in the output LMU, we only need to simultaneously start all the LMUs used by the DFG. At the beginning, each input LMU iterates over the data domain according to its mapped access pattern and puts a new data on its output every cycle, while each output LMU performs countdown first. After the delay of one execution of the DFG, each output LMU finishes countdown, and starts sensing its input and storing data in its memory space with the address calculated from the mapped access pattern. The duration of the countdown, i.e., the delay of the DFG, is calculated at compile time and included in the configuration bits of each LMU.

3.5 Detailed Module Design

3.5.1 Computation Element

A computation element collects a set of ALUs with the computation patterns that frequently appear in user applications. In conventional CGRAs, a computation element is usually implemented as a set of homogeneous ALU nodes with certain specific topology.

For example, the CGRA in [CGG12b] uses the tree structure in Fig. 3.11. It contains 15 homogeneous nodes, each equipped with an adder and a multiplier. The problem is that in a fully pipelined architecture, each node will be assigned only one operation. This means that at least one ALU between the adder and the multiplier will be wasted, which limits the hardware utilization to below 50%. When we map the arithmetic operations in the DFG of Fig. 3.1, we can only use 11 ALUs out of the total 30 ALUs, as shown in Fig. 3.11. The utilization is only 36%. Note that the homogeneity of computation nodes is a reasonable design choice in a conventional architecture with time multiplexing, since each node will be
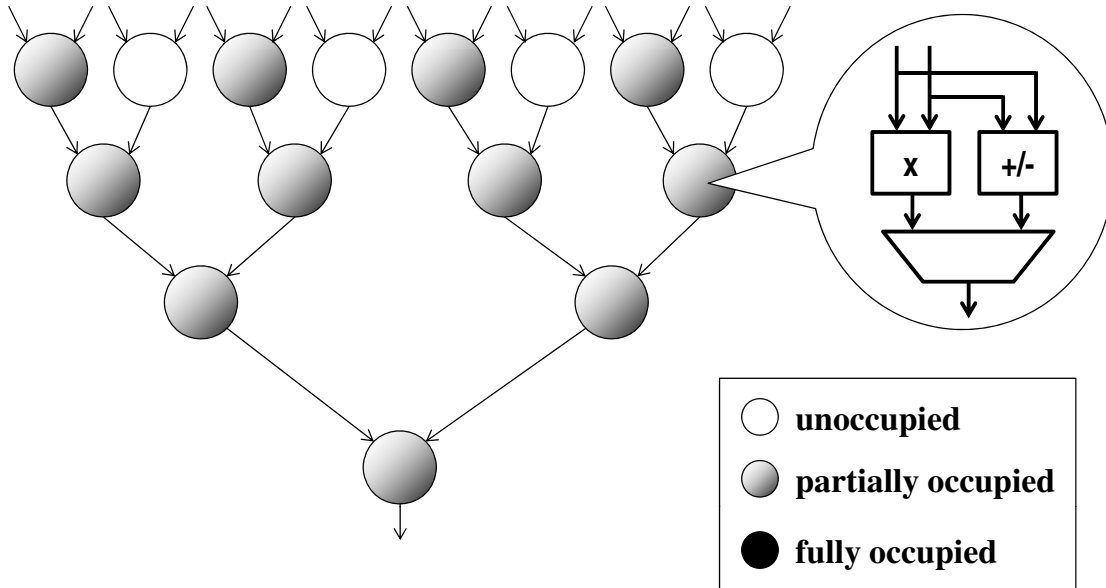
Figure 3.11: The tree structure of homogeneous nodes used in the computation element in [CGG12b] and mapped with the arithmetic operations in the DFG in Fig. 3.1.

assigned multiple operations. These operations could cover both addition and multiplication, and then both the adder and the multiplier in a node need to be used.

To improve the utilization problem that emerges in the fully pipelined architecture, we adapt the popular computation pattern of the Xilinx DSP block [Xila] for the design of the computation element, as shown in Fig. 3.12. Each computation element contains three heterogeneous nodes: a two-input adder, a two-input multiplier, and a three-input adder. It can evaluate any subset of the expression in the bottom left part of Fig. 3.12. There is also a dedicated connection between two adjacent CEs to reduce the burden on the permutation network. When we map the arithmetic operations in the DFG of Fig. 3.1, two out of the three nodes in the first and the third CEs are occupied, and all three nodes in the other two CEs are occupied. The utilization is as high as 83%.

More detailed implementation of fully pipelined CE is shown in Fig. 3.13.
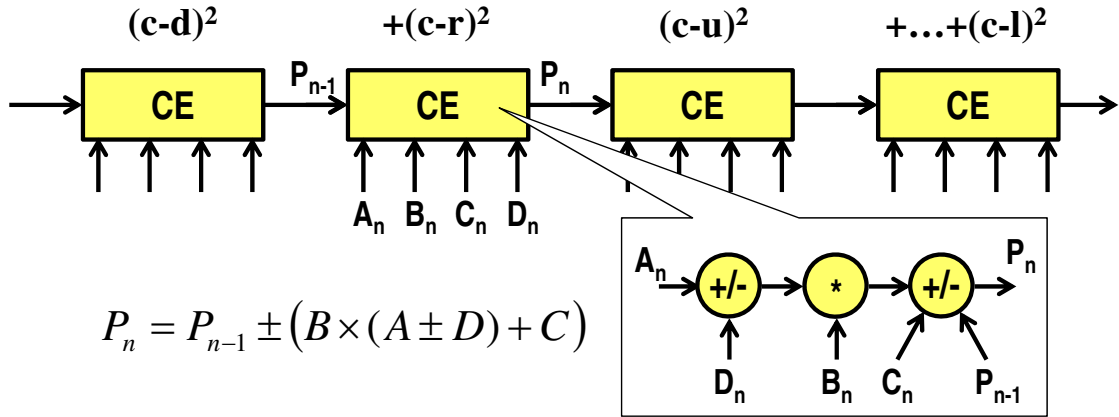
Figure 3.12: Our computation element with heterogeneous nodes to improve utilization in a fully pipelined architecture.

$$P_n = P_{n-1} \pm \left(B \times (A \pm D) + C \right)$$
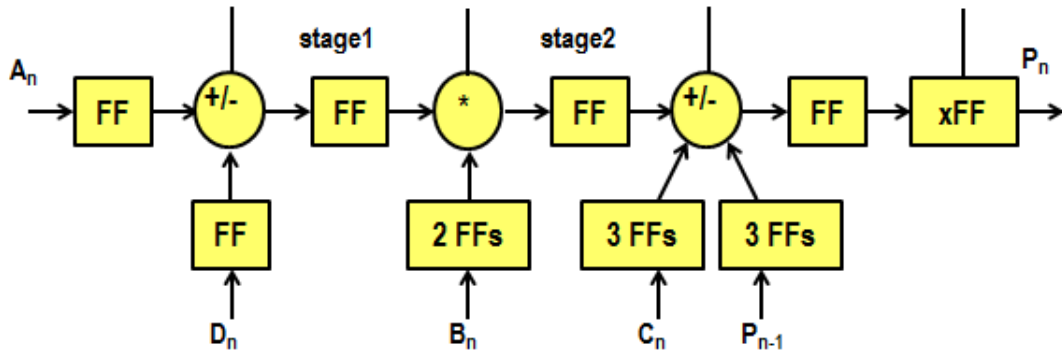
Figure 3.13: Detailed implementation of fully pipelined CE

3.5.2 Local Memory Unit

A local memory unit contains an on-chip memory bank, an address generator and a FIFO controller, as shown in Fig. refLMU.

The memory bank is dual-port so that the data access for computation and the data transfer with the main memory can be performed in parallel.

Since each load operation will be executed every clock cycle in a fully pipelined architecture, a large shared memory, e.g., the design in [CGG12b] will suffer severe memory port
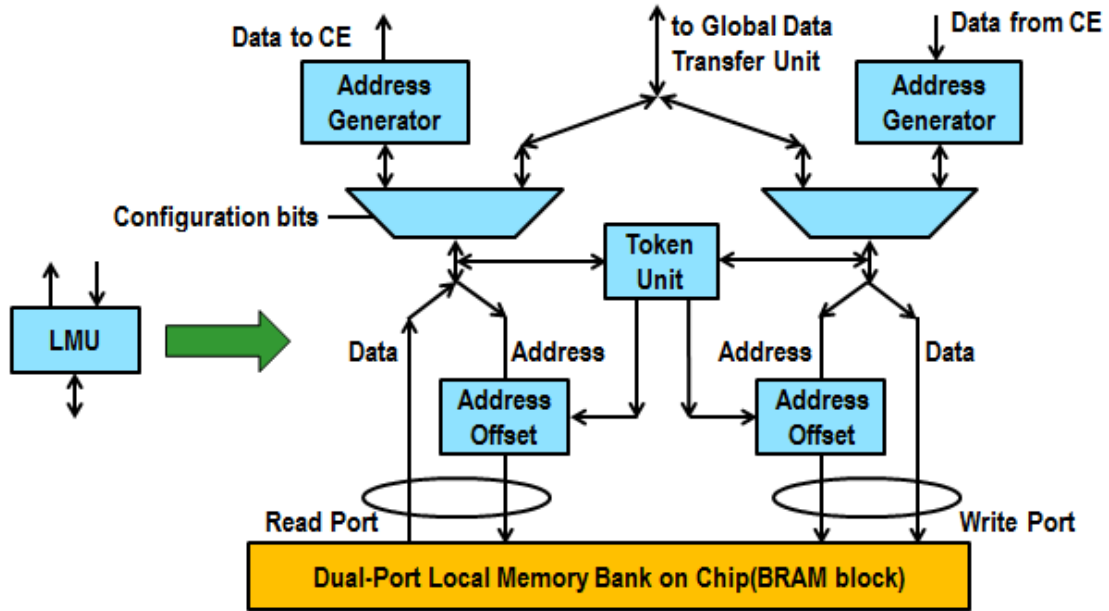
Figure 3.14: Detailed implementation of the local memory unit

contention and will fail to achieve the designed throughput. In our FPCA, all the load operations are mapped to separate LMUs, each with an address generator configured to the access pattern of the corresponding load operation. For example, the LMU mapped to $r : A[j-1][k]$ will iterate its address in the domain $1, 2, \cdots, 62; 65, 66, \cdots, 126, \cdots$. This is a 2D rectangular domain with the incremental factor of 1 in the first dimension and 64 in the second dimension. Currently our address generator supports the iteration domain of a parallelogram with up to three dimensions. This design meets the needs of almost all the applications we found.

The FIFO controller supports the double buffering of LMU. It provides the empty/full signals of a FIFO with the depth equal to 2 and the granularity equal to a data block.
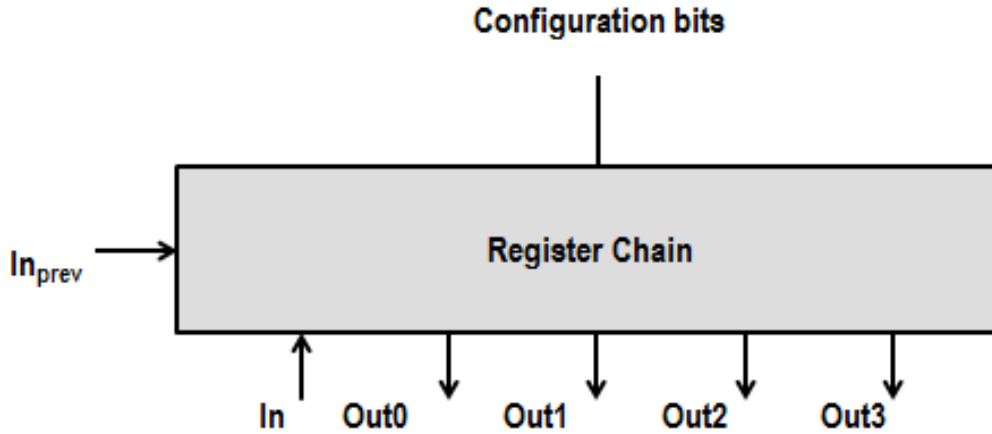
Figure 3.15: Register chain with 1 input and 4 fanout outputs

3.5.3 Register Chain

In some applications, some intermediate results in the DFGs will be used by multiple CEs. However these CEs are not aligned to work on the same data element from the same loop iteration. For example, all four CEs in Fig. 3.8 need to use the data element from the LMU mapped to $c : A[j][k]$. However, the first CE starts processing data elements earlier than the second CE since the first CE sends its result to the second CE. The same applies to the third and the fourth CEs. Therefore, we need a temporary storage for the result of the LMU mapped to $c : A[j][k]$ so that it can be used by different CEs at different clock cycles. This temporary storage will also be used for register-level data reuse to save load operations on LMUs.

The register chain in our FPCA, as shown in Fig. 3.15 is such a temporary storage with one input and multiple outputs which can be connected to different CEs via the permutation network. To illustrate how it works, suppose the delays of the second CE, the third CE and the fourth CE compared to the first CE are 1, 3 and 6 cycles respectively. A conventional CGRA with time multiplexing can have a single flip-flop (FF) to store a data element for 6 cycles. The problem is that in our fully pipelined architecture, there is a new data element

27

coming in every clock cycle. The single FF does not have sufficient room to store the new element until the old element is released 6 cycles later. Therefore, we implement a chain of FFs in the register chain to allow the data element to move along the chain every cycle without storage conflicts. The outputs of every several FFs are connected not only to the successive FFs, but also to the outputs of the register chains. To get the target delay for each output of the register chain, some FFs will be bypassed by configuration bits. To get the example delays of 1, 3, and 6 cycles, only 1, 2 and 3 FFs before the last three outputs need to be kept active. Since a later output can reuse the delay of an earlier output, many FFs are saved.

3.5.4 Permutation network

The permutation network connects LMUs, CEs, and register chains together. Its connection is kept constant during accelerator execution and is free of arbitration upon data transmission which is a big overhead in conventional designs based on bus or network-on-chip [CGG12b]. The connection of the permutation network can be configured only during accelerator composition. Once configured, each data path in the network provides the throughput of one element every clock cycle. We use the design in [Wak68], and the outputs of the network can be configured to any permutation of the inputs of the network. Fig. 3.16 shows the permutation network with 8 inputs and 8 outputs. The complexity of the network is $O(n \log n)$ where n is the number of the network IOs. It means that the permutation network is scalable in common cases. The only constraint of the permutation network is that one input can be connected to only one output in one configuration. It does not support the connection with a fanout > 1. However this problem can be solved by our register chain which can duplicate the incoming data element to its multiple outputs.

With the full connectivity of the permutation network, we only need to care about the logic resources during the dynamic composition. As long as all the logic resources are mapped to the DFG nodes, any connection among these nodes can always be provided by the network.
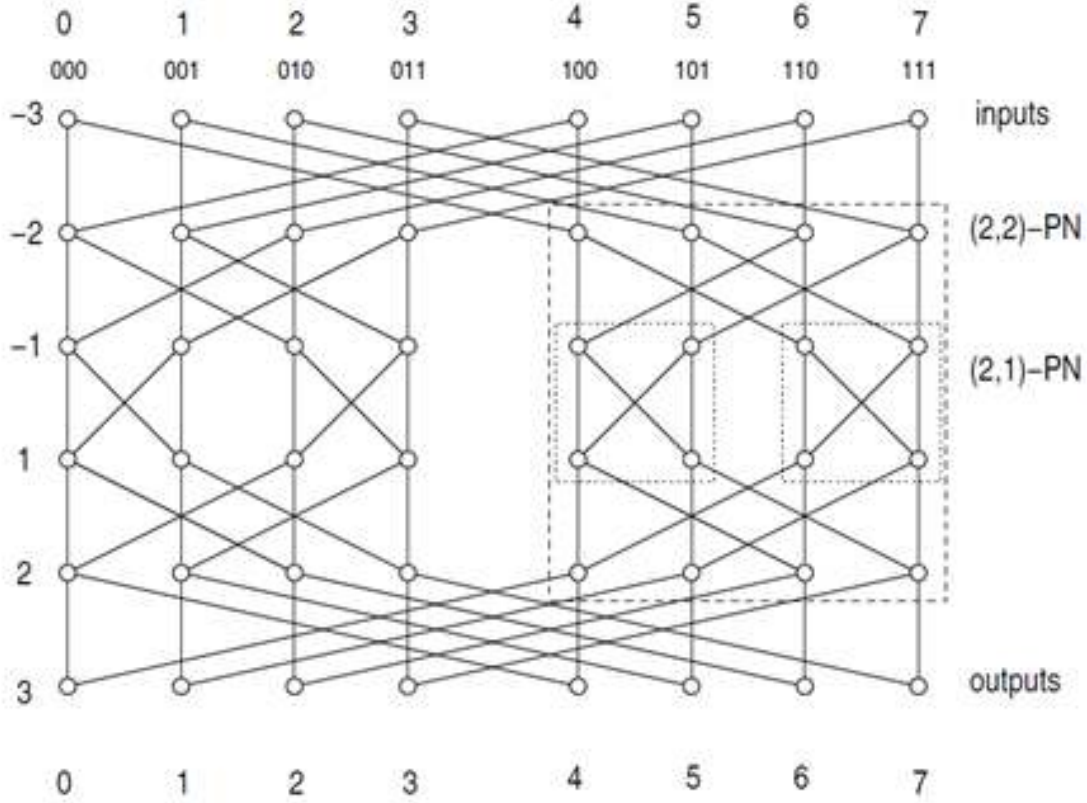
Figure 3.16: Permutation network with 8 inputs and 8 outputs implemented in Benes network

The guaranteed routability consolidates the benefit of the dynamic composition. One PE cluster can even accommodate two or more applications if each application occupies only a small part of logic resources. The connections from two applications can still be merged into a certain permutation from the network inputs to the outputs and can be provided by the network.

3.5.5 Synchronization Unit

The synchronization unit enforces the simultaneous starts of all the LMUs used by the same DFG. It checks the emptiness of data blocks in input LMUs and the fullness of output LMUs. If no LMU is empty/full, the synchronization unit will issue a start signal to all the involved

29

LMUs. Different DFGs form different threads in the synchronization unit and are processed separately.

3.5.6 Global Data Transfer Unit
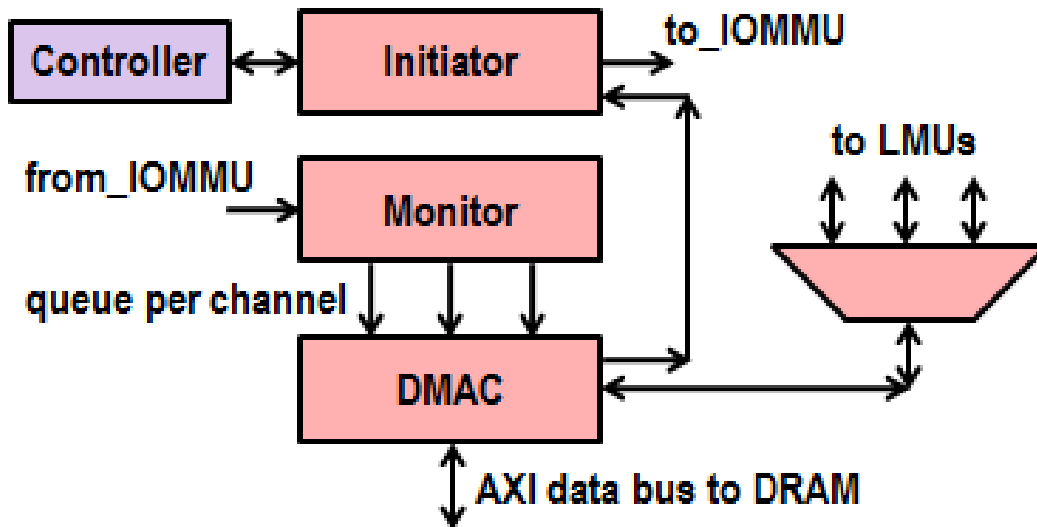
Figure 3.17: Detailed implementation of the global data transfer unit

The GDTU contains a request initiator and several direct memory access controllers, one for each channel. The request initator generates data transfer requests from the assigned subtasks and the starting virtual addresses of the data arrays in user applications. At the beginning, it will send a number of the generated requests to the IOMMU for page translation. Once any request has been processed by the IOMMU and returned to the GDTU, the GDTU will immediately pick up a new request from the generated requests and send it out. This mechanism ensures that there are always a certain number of requests being processed by the external IOMMU. It is based on the consideration of the nondeterministic latency of page translation. The multiple requests without TLB misses can hide the long latency of a request with a TLB miss which needs to wait for the OS to response.

3.6 Design Automation

We use the platform-based design methodology and develop a design automation flow for our FPCA. It accepts a configuration file with the FPCA's architectural parameters, e.g., the number of PE clusters in the array, the number of CEs and LMUs per cluster, etc. These configurations will be combined with hardware templates to generate all the RTL codes of our FPCA. The RTL codes can be used for RTL simulations. They can also be pushed to an FPGA synthesis flow (e.g., Xilinx Vivado [Xilb]) to generate the bitstream of a working prototype to run on an FPGA board.

3.7 Compiler Support

Considering that the possible number of FPCA mapping solutions grows exponentially with the application size, manual mapping is not scalable in terms of both feasibility and solution quality. To solve this problem, we build an automatic compilation flow using an LLVM compiler infrastructure [htt], which maps kernel programs to the proposed FPCA platform. This automated mapping flow contributes in two ways: First, it enables high-quality mapping of input applications in a timely manner; also, coming from the architecture point of view, it allows for extensive design space exploration for different FPCA configurations. The overall compilation flow is shown in Fig. 3.18. The flow takes our FPCA architecture parameters as inputs, including the computation pattern supported by a CE, on-chip SPM size, and configuration for the on-chip data network. The flow consists of three steps which are described as follows:

DFG Generation. The LLVM-based compiler front-end transforms the input kernel program to an in-memory data flow graph, in which each node represents either an operation (arithmetic, logic and control) or a memory reference (load and store). Data reuse techniques in [CHL11] are adopted in this flow to activate data reuse among memory references. For instance, the reuse distance between array reference $a[i][j][k]$ and $a[i][j][k+1]$ in Fig. 3.2 is
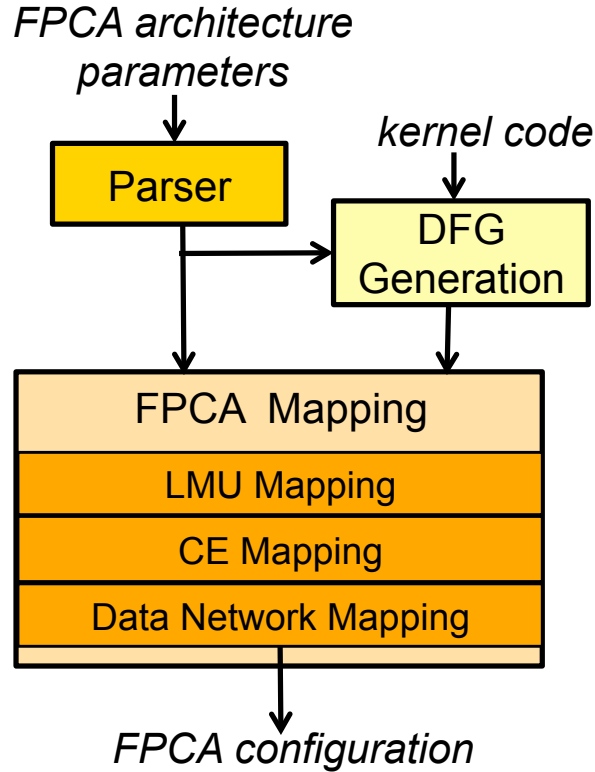
Figure 3.18: Overall FPCA compilation flow.

one iteration (here we assume row-major order), and is 64 iterations between $a[i][j][k]$ and $a[i][j+1][k]$. Under the data block size constraint, the FPCA compiler selects the entire or subset of data reuse existing in the original DFG to minimize the amount of off-chip data transfers. The inserted on-chip memories for data reuse form the data blocks that separate the top-level DFG and the bottom-level DFG (see Fig. 3.1 for an example).

LMU Mapping. Each memory reference in the input data flow graph is mapped to one SPM bank, which guarantees conflict-free SPM access at every cycle. For example, the six memory references in Fig. 3.1 are mapped to six SPM banks (five for read, one for write).

CE Mapping. Given the CE computation pattern, the compiler decomposes the kernel data flow graph into a series of FPCA-executable CEs, which are called *CE candidates*. In our flow, the subgraph identification and isomorphism checking techniques proposed

in [CJ08] are employed to generate CE candidates efficiently. A filtering scheme based on *characteristic vector* [CJ08] is applied to reduce the number of expensive graph isomorphism checking operations. Note that instead of generating all the CE candidates in an input data flow graph, we only target those that can be mapped to the CE computation pattern. Therefore, the micro-architectural constraints in the given CE design, such as depth, size, number of inputs/outputs, can be applied to prune the identification space. Efficient pruning techniques in [CHM06] are used to remove unnecessary evaluations (including using a pre-generated greedy solution as the initial pruning bar and pre-sorting all CE candidates to estimate the final mapping size from the current partial one), which significantly speeds up the algorithm runtime. If the mapping algorithm cannot terminate within a given amount of time (currently set to 600 seconds), the optimal solution obtained so far will be selected to generate all the configuration files for FPCA execution.

Data Network Mapping. This step deals with the topology mapping of the bottom-level DFG in the rightmost part of Fig. 3.1. It is coupled with CE scheduling and register chain insertion to guarantee that data at the input ports of each CE are synchronized. Here, we call those mapping solutions with balanced path delay at each CE's input ports *valid* mappings. For example, Fig. 3.19 shows a valid mapping of Fig. 3.1. One register chain has been inserted to create additional delay at unbalanced paths $LMU_3 \rightarrow CE_2$, $LMU_3 \rightarrow CE_3$ and $LMU_3 \rightarrow CE_4$. Since the register chain creation scheme directly depends on the CE mapping result, our compiler flow employs an area-centric metric to optimize these two types of on-chip resources simultaneously. To maximize resource utilization, a branch-and-bound algorithm is built to select the valid mapping solution with optimal resource usage. At each step, one CE is included in the mapping solution. When a valid mapping solution is found, it will be compared to the optimal solution obtained so far. Every valid mapping solution has been evaluated in this process to ensure mapping optimality.
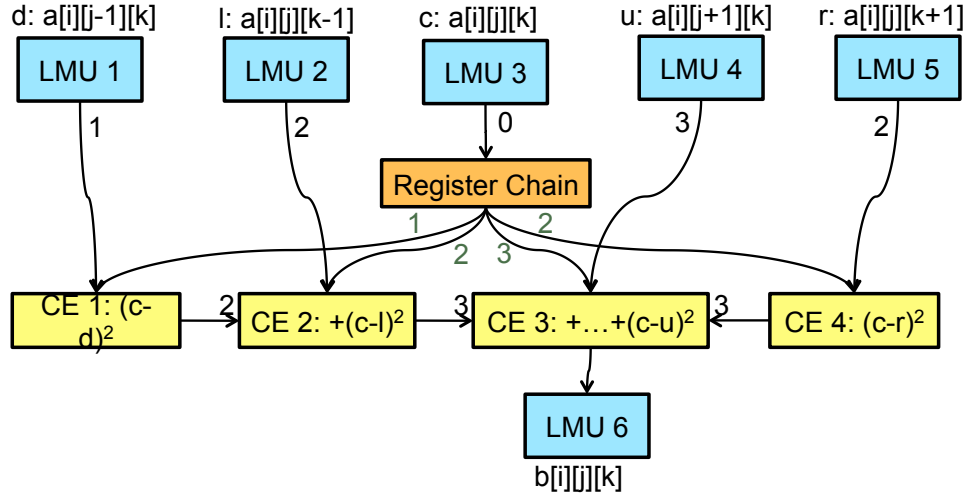
Figure 3.19: A valid mapping for Fig. 3.1.

3.8 Design Space Exploration

The FPCA introduces design flexibility in three levels: intra-CE, intra-cluster and inter-cluster. The first level of flexibility comes from the fact that we can easily change our computation patterns to accommodate new application domains. In this paper we set the CE computation pattern to the popular one used by the Xilinx DSP block [Xila]. However, the selection of computation patterns for different domains is an interesting research problem within the FPCA context which offers opportunities for possible further improvement. Intra-cluster level flexibility is the flexibility to set different numbers for different modules within one cluster. The metrics and different design choices are discussed in Section 3.8.1. For inter-cluster level flexibility discussed in Section 3.8.2, we will see the difference between cluster-based design and flat design.

3.8.1 Module Number Allocation Within Cluster

Let's first think about the system constraints that the fully pipelined design has put forward. Full pipelining means that modules within one cluster can access any others without the

possibility of being blocked. Every output data port of a module can send data to one input port of the interconnect and every input data port of a module can receive data from one output port of the interconnect.

There are three kinds of modules, REG, CE and LMU, that need to transfer data through the permutation network. CE has 4 inputs and 1 output. REG has is 1 input and 6 outputs. LMU constantly has 1 input and 1 output. For the permutation network, the number of ports is in the form of 2^N where N is a positive integer. The smallest permutation network that can meet the interconnect requirement of a GRADIENT kernel is 32. In this way, we have the following constraints:

$$6 * REG + 1 * CE + 1 * LMU \leq 32 \tag{3.1}$$

$$1 * REG + 4 * CE + 1 * LMU \leq 32 \tag{3.2}$$

Then we define our metrics, $\overline{\mu}$, the average module utilization ratio for a set of applications here: $\overline{\mu} = avg(\mu)$. And μ stands for the average of module utilization ratios for one application. Now, our exploration converges to a linear programming problem: maximize $\overline{\mu}$, subject to Equations (3.1) and (3.2).

We developed a solver to find all the settings of $\{REG, CE, LMU\}$ that meet the requirements, sort their average module utilization ratios $\overline{\mu}$, and find the largest one. The results are shown in Table 3.1. According to Table 3.1, {2,6,6} is the module allocation for the number of {REG, CE, LMU} within one cluster, which achieves the largest utilization ratio when we only compose one copy of the accelerator at one time. Similar analysis can be applied to composing multiple copies of the accelerator in clusters.

3.8.2 Cluster-Based Design vs. Flat Design

Since we have settled on the design for one set of module allocations which has 2 REG, 6 CE and 6 LMU within one PE cluster, let's name it as one *unit* in this paper. One *unit*

Table 3.1: Module requirement for applications

$\{REG, CE, LMU\}$	$grad$	$conv$	$edge\_sobel$	$\overline{\mu}$
$\{3, 5, 9\}$	0.60	0.32	0.60	0.51
$\{3, 4, 10\}$	0.64	0.33	0.66	0.54
$\{2, 5, 10\}$	0.63	0.30	0.70	0.54
$\{2, 4, 14\}$	0.64	0.29	0.74	0.56
$\{2, 6, 6\}$	0.72	0.39	0.72	0.61

now is a scale of chip size. If one chip has four *units*, then it has 8 REG, 24 CE and 24 LMU in all. Now the question for one chip with 4 *units* size: shall we build it as flat as one PE cluster which has only one huge interconnect with enough ports connecting all the modules (approach_1), or shall we build it in four clusters where each is 1 *unit* size and has one small–scale interconnect (approach_2)? Considering the energy efficiency, we would like to choose the design with fewer active resources. And the cluster-based design proves to have more scalability than the flat design because the interconnect part in the flat design scales up very fast. The number of switches in the data network with N ports (N inputs and N outputs) is $0.5N(2 \log_2 N - 1)$.

For approach_1, the interconnect port number is 32*4 = 128 and there are 832 switches. For approach_2, there are four PE clusters. Each cluster has one 32–port interconnect and 144 switches. In total there are 576 switches. And same thing applies as N becomes larger: the flat design approach consumes many more registers than the cluster-based design because the interconnect for the former one scales up $N \log_2 N$ times as port numbers scale up N times. However, for the cluster-based design, once the size of one PE cluster is fixed, the total area scales up N times, which is the same as how the port number scales up.

From Fig. 3.20, we can see that to design a chip that is as big as 256 *units*, when we change the single cluster size from 1 *unit* to 256 *units*, the average slice registers and slice
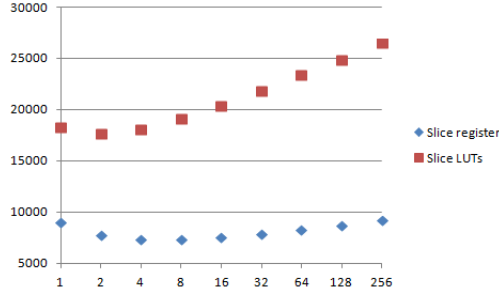
Figure 3.20: Slice registers and slice LUTs as cluster scales up. Chip size is 256 *units*.

LUTs for one *unit* have first decreased and then increased after the lowest point. The reason is because at first, area is saved because of the decreasing number of PE clusters. Then the area consumption reaches the lowest level and increases because the interconnect in one cluster increases rapidly and out-weighs the saved area by fewer PE clusters. We found that when one cluster size is set to 2 or 4 units, its area consumption achieves the lowest level. We could further extend the same method to determine the size of one cluster when we have larger chips. And this also could be proved using a mathematical deduction that the area consumption has the same pattern, no matter how large the chip is.

One note here is that cluster-based design is more scalable than the flat design in terms of area consumption. As long as the application mapping results are correct, we tend to prefer the cluster-based design over the flat design because we probably don't need the full connectivity for all the modules on the chip.

CHAPTER 4

Experimental Results

4.1 Settings

We implemented a working prototype of FPCA in Xilinx Virtex-6 FPGA XC6VLX240T. We measured the performance and power of our FPCA via the Xilinx Watchdog IP and the P4400 Kill-A-Watt Power Meter. We used the Xilinx soft processor IP Microblaze implemented in the FPGA, and the low-power hard processor ARM Corte-A9 in Zynq SoC [Xild] for comparison.

We pick up three typical benchmarks from three application domains. GRADIENT is from medical imaging [CSR11]. CONVOLUTION is from digital processing. SOBEL is from the Sobel edge detection algorithm [VNS07]. All the benchmarks use the input data of a $256 \times 256 \times 256$ 3D array.

4.2 Performance Gain and Energy Efficiency

The performance gain and energy efficiency improvement of our FPCA are shown in Table 4.1.

We first suppress the global accelerator manager (GAM) in our FPCA to compose only one copy of a hardware accelerator for each application. The runtime of our FPCA is around 0.235 seconds, which corresponds with a 1.4 clock cycle per loop iteration. It is close to the design target of loop pipeline initial interval (II) equal to 1. The gap with the theoretic

Table 4.1: Performance gain and significant energy efficiency improvement of our FPCA.

		GRADIENT	CONVOLUTION	SOBEL
Microblaze Processor in the Xilinx FPGA @ 100MHz	runtime (s)	45.2	30.9	45.1
	energy (J)	94.9	64.9	94.7
Dual-Core ARM Cortex-A9 MPCore @ 800MHz	runtime (s)	0.346 (1x)	0.576 (1x)	0.787 (1x)
	energy (J)	0.381 (1x)	0.634 (1x)	0.866 (1x)
FPCA prototype in FPGA @ 100MHz	runtime (s)	0.235 (1.5x)	0.253 (2.3x)	0.234 (3.4x)
	energy (J)	0.729 (0.52x)	0.784 (0.81x)	0.725 (1.19x)
FPCA projected on 45nm ASIC with power gating	runtime (s)	0.059 (5.8x)	0.063 (9.1x)	0.059 (13.3x)
	energy (J)	0.015 (25x)	0.016 (39x)	0.015 (57x)

value mainly comes from the extra overhead of page translation and bank switching in the main memory upon discontinuous data accesses. This inspires future researches on memory controllers for high-throughput reconfigurable architectures.

As shown in Table 4.1, since the Microblaze is implemented in an FPGA and is much less efficient than the ASIC processor ARM, we mainly use ARM as the baseline, and report the speedup and energy savings in brackets. We observed a 1.5-3.4x speedup of our FPCA prototype implemented in the FPGA compared to the Dual-Core ARM, However our FPCA prototype consumes more energy since its underlying hardware is an FPGA. The data paths in FPGAs have to go through LUTs and routing switches, and therefore are much longer than their ASIC counterpart. We can project the evaluation results of our prototype to ASICs. As reported in [KR07], the ratio of critical path delay, from FPGA to ASIC, is around 4, and the dynamic power consumption ratio is around 12. If our FPCA is further implemented in ASIC and power gating is applied to idle resources, a >50x energy savings can be achieved.

Next, we show the benefits of accelerator duplication based on dynamic composition. The hardware resources available in the Xilinx FPGA allow us to implement a PE array which can compose up to 4 copies of a hardware accelerator for the application GRADIENT. We gradually release the suppressing on the GAM to compose 1, 2, and 4 copies. The

Table 4.2: Performance improvement with dynamic composition.

duplicating accelerator for GRADIENT from idle resources				
# of copies	1	2	4	4 (dummy DDR)
runtime (s)	0.235	0.118 (2x)	0.118 (2x)	0.059 (4x)
cycle/iteration	1.4	0.7	0.7	0.35

performance improvement is shown in Table 4.2. When we duplicate the accelerator into two copies, the performance is doubled. However when we further duplicate the accelerator into four copies, the performance improvement stops. We found that in this case, the off-chip bandwidth is saturated in the FPGA chip which we used for our FPCA prototype. This is due to the limited number of FPGA pins connected to the DRAM chip. In an ASIC implementation in the future, the bandwidth will not be an issue since we will have freedom to increase the bandwidth to a sufficient value. In the experiments described in this paper, we implemented a dummy DDR on the FPGA chip to take over the off-chip accesses, and found that the performance can keep scaling with the number of accelerator copies.

4.3   Composition Time

In CGRA, only word-level or sub-word level computation is enabled. Compared to FPGA, it saves much of the reconfiguration time to compose an application. To process the input data of $256 \times 256 \times 256$ 3D array, the runtime and composition time for different benchmarks are shown in Table 4.3.

Table 4.3: Composition time for FPCA.

	GRADIENT	CONVOLUTION	SOBEL
runtime (ms)	235	253	234
composition time (ms)	0.357	0.355	0.356

For FPGA, the time to download the bitstream usually takes more than 10s. Compared to FPGA, our FPCA saves time in the reconfiguration process.

4.4 Area Breakdown

We also measured the resource usage for each module in our FPCA architecture and the area breakdown is shown in Table 4.4. A major fration, 42 percent of LUTs amd 25.8 percent of registers, goes to the interconnect subsystem, which includes multiplexers, pipeline registers. The computation element, local memory unit, register chain, global data tranfer unit, synchronization unit, controller comsume 12 percent, 15 percent, 3 percent, 23 percent, 0.2, 4 percent of LUTs and 9.7 percent, 10.3 percent, 9.7 percent, 38 percent, 0.02 percent, 6.6 percent of registers.

Table 4.4: Area breakdown for one PE cluster in FPCA.

	Slice register	Slice LUTs	# in current design	Percentage register	Percentage LUTs
Computation element	192	511	6	9.7%	12%
Local memory unit	136	432	6	10.3%	15%
Register chain	768	512	2	9.7%	3%
Global data transfer unit	3018	3897	-	38%	23%
Permutation network	2048	7136	-	25.8%	42.3%
Synchronization unit	2	32	-	0.02%	0.2%
Controller	523	659	-	6.6%	4%
total	7943	16872	-	-	-

CHAPTER 5

Conclusion

We developed a novel CGRA architecture that enables full pipelining and dynamic composition to improve energy efficiency by taking full advantage of abundant transistors in the upcoming era. Several new design challenges are solved. We implemented a prototype of the proposed architecture in a commodity FPGA chip for verification. Experiments show that our architecture can achieve a >50x energy savings due to customization for user applications with rich transistor resources.

APPENDIX A

Formal Proof of Best Performance/Area for Full Pipelining

A loop in the accelerator can have different implementations with different resource usage and performance. The performance and area model under different pipeline IIs are formulated, and we have following assumptions:

1. Based on our experiments, when setting the same target clock frequency for different pipeline IIs, the achieved clock frequencies varied 4.36% on average. Thus, we assume the clock periods for different IIs are the same.

2. Some resources can be shared among different arithmetic units when II is larger than 1, while some other resources can not. The arithmetic unit is a unit that does one arithmetic operation. For example, for Vertex-7 FPGA, one double-precision adder needs 3 DSPs, 731 LUTs, 754 FFs, and one double-precision multiplication needs 11 DSPs, 256 LUTs, 456 FFs.

3. When II is larger than 1, there are multiplexers for the sharing computation among arithmetic units. These are the overheads for the time-multiplexing on arithmetic units.

A.1 Performance and Area Models

In this section we describe the performance and area models.

A.1.1 Performance Model

While using loop pipelining, parallelism across different loop iterations can be exploited by initiating the next iteration of the loop before the completion of the current iteration. The total execution cycle of a pipelined loop can be estimated using Equation (A.1).

$$Cycle_k = II_k * (TC_k - 1) + Dth_k \tag{A.1}$$

With given loop code, the trip count TC_k of each loop is fixed. The minimal initiation interval II_k is constrained by loop-carried dependence or memory port conflict. Dth_k is loop depth, which is the number of cycles to fill the pipeline and to get the result of the first iteration of a loop. When TC_k is large, performance measured in new iterations per cycle can be estimated using Equation (A.2).

$$Performance_k = \frac{1}{II_k} \tag{A.2}$$

When the clock period is taken into account, performance measured in new iterations per second can be estimated using Equation (A.3).

$$Performance_k = \frac{1}{II_k * CP} \tag{A.3}$$

Here, CP stands for clock period that can be viewed as a constant for different IIs.

A.1.2 Area Model

The area consumption of a loop can be estimated by summing up the area of non-shareable components $Area_{NS}$, sharable components $Area_S$, multiplexers $Area_{MUX}$ and registers $Area_{reg}$.

$$Area = Area_{NS} + Area_S + Area_{MUX} + Area_{reg} \tag{A.4}$$

$Area_{NS}$: Non-shareable components contain logic for iterators and memory partitioning. Thus, we have

$$Area_{NS} = Area_{it} + Area_{mp} \tag{A.5}$$

$Area_{it}$ is the area of the logic for incrementing the iterators and monitoring the loop exit condition, which can be viewed as a constant. $Area_{mp}$ is the logic for memory partitioning. Fig. A.1 shows the block diagram of a partitioned memory system [LWZ12]. It consists of
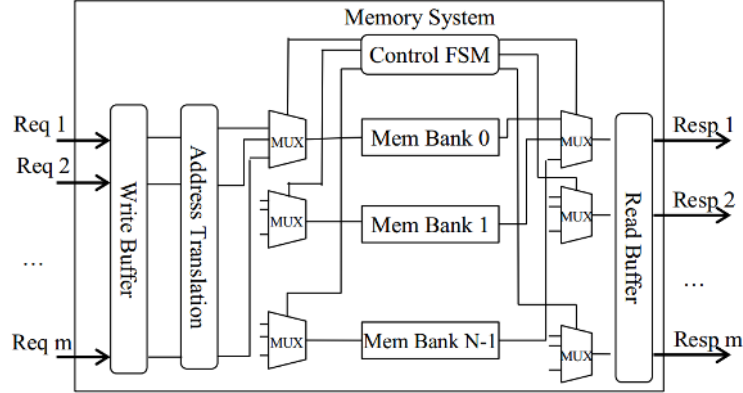
Figure A.1: Block diagram of a partitioned memory system

memory banks, address translation unit, control FSM, N input MUXs and m output MUXs.

In the system, if there are m memory requests, and N memory banks, then we have

$$N = \lceil \frac{m}{II_k} \rceil \tag{A.6}$$

The cost of an address translation unit is proportional to m, say $\lambda_1 * m$. The cost of the control FSM unit is proportional to N, say $\lambda_2 * N$. And cost of MUX consisting of N M:1 MUX between the write buffers and memory banks and M N:1 MUX between the memory banks and read buffers is proportional to $m * N$, say $\lambda_3 * m * N$. Thus, the area model for the memory partitioning part is shown in Equation (A.7). ($\lambda_1, \lambda_2, \lambda_3$ are platform-dependent parameters)

$$Area_{mp} = \lambda_1 * m + \lambda_2 * N + \lambda_3 * m * N \tag{A.7}$$

$Area_S$: The computation logic is used to do arithmetic operations and can be viewed as sharable components. When II_k increases, computation logic drops and can be expressed

in Equation (A.8). Because there is no perfect sharing for all the arithmetic operations, the ceiling function is used here.

$$Area_S = \sum (\lceil \frac{N_i}{II_k} \rceil * Area_i) \tag{A.8}$$

N_i is the number of i_{th} type arithmetic operation in the DFG. $Area_i$ is the area of i_{th} type arithmetic operation in the DFG. Under the same clock period, the area of the same type arithmetic operation for different IIs is the same.

$Area_{MUX}$: When II increases, the number of i_{th} type arithmetic unit, i.e., $\lceil \frac{N_i}{II_k} \rceil$, decreases. Therefore, the number of MUX connecting these units decreases. The size of each MUX, i.e., $Area_{k:1MUX}$, depends on II. For FPGA, the area of k:1 MUX is invariant when $k \leq 6$ because the MUX is implemented by LUT with 6 inputs and 1 output. For ASIC, the area of k:1 MUX is proportional to k.

Therefore, the total area for MUX is shown in Equation (A.9). ($Area_{k:1MUX}$ in FPGA is a constant when $k \leq 6$. $Area_{k:1MUX} \propto k$ in ASIC)

$$Area_{MUX} = \sum (\lceil \frac{N_i}{II_k} \rceil * Area_{k:1MUX}) \tag{A.9}$$

$Area_{reg}$: When II increases, the number of i_{th} type arithmetic unit, i.e., $\lceil \frac{N_i}{II_k} \rceil$ decreases. However, the number of registers to store the intermediate results at the output of each arithmetic unit is equal to II. Therefore, the product, i.e., the total area of registers is shown in Equation (A.10). ($area_{reg}$ is the area for one register)

$$Area_{reg} = \sum (\lceil \frac{N_i}{II_k} \rceil * II_k * area_{reg}) \tag{A.10}$$

This model is also correct when there is operation fusion. The operation fusion is related to target clock period. When a target clock period is given, operation fusion is determined and we can get a new DFG after the fusion. Then the number of registers to store the intermediate results for one iteration depends on the new DFG. And when II increases, the number of operations of the same type (including the fused operations) is not always divisible by the pipeline II.

Thus, performance/area measured in a new iteration/second/unit area can be expressed in Equation (A.11).

$$\frac{Performance}{Area} = \frac{1}{II_k * CP}$$

$$* \frac{1}{Area_{NS} + \sum(\lceil \frac{N_i}{II_k} \rceil * Area_i) + \sum(\lceil \frac{N_i}{II_k} \rceil * Area_{k:1MUX}) + \sum(\lceil \frac{N_i}{II_k} \rceil * II_k * area_{reg})}$$

$$(A.11)$$

By multiplying II_k into a denominator that describes the area, we can get:

$$\frac{Performance}{Area} = \frac{1}{CP}$$

$$* \frac{1}{Area_{NS} * II_k + \sum(N_i + R_{ik}) * Area_i + \sum(N_i + R_{ik}) * Area_{k:1MUX} + \sum(N_i + R_{ik}) * II_k * area_{reg}}$$

$$(A.12)$$

Here, R_{ik} is the number of i_{th} type arithmetic unit introduced by division and ceiling function followed by multiplication. $R_{ik} = N_i$ - $(N_i \bmod II_k)$.

A.2   Best Performance/Area for Full Pipelining

Based on the performance and area models shown in Section A.1, here we prove that $\frac{Performance}{Area}$ is the highest when $II_k = 1$. This is because every term in the denominator in Equation (A.12) when $II_k = 1$ is smaller than that when $II_k \geq 2$.

For the first term, we have:

$$Area_{NS} * II_k = Area_{it} * II_k + (\lambda_1 * m + \lambda_2 * N + \lambda_3 * m * N) * II_k$$

$$= Area_{it} * II_k + \lambda_1 * m * II_k + \lambda_2 * (m + R_{mk}) + \lambda_3 * m * (m + R_{mk}) \quad (A.13)$$

Here, $N * II_k = \lceil \frac{m}{II_k} \rceil * II_k = m + R_{mk}$ and $R_{mk} = m$ - (m mod II_k).

Thus, it is clear to see that

$$Area_{NS} * II_1 < Area_{NS} * II_k, k \geq 2 \quad (A.14)$$

47

For the second term, we have

$$R_{i1} \le R_{ik}, k \ge 2 \tag{A.15}$$

This is true because R_{i1} is constant 0. Therefore,

$$\sum (N_i + R_{i1}) * Area_i \le \sum (N_i + R_{ik}) * Area_i, k \ge 2 \tag{A.16}$$

For the third term, when $II_k = 1$, $Area_{k:1MUX} = 0$. Therefore,

$$\sum (N_i + R_{i1}) * Area_{1:1MUX} < \sum (N_i + R_{ik}) * Area_{k:1MUX}, k \ge 2 \tag{A.17}$$

For the fourth term, it is trivial to see that

$$\sum (N_i + R_{i1}) * II_1 * area_{reg} < \sum (N_i + R_{ik}) * II_k * area_{reg}, k \ge 2 \tag{A.18}$$

As shown before, each term in the denominator in Equation (A.12), when $II_k = 1$, is smaller than when $II_k \ge 2$.

Therefore, we conclude that full pipelining achieves the best performance/area.

References

[BBK07]   Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. "Architectural exploration of the ADRES coarse-grained reconfigurable array." In *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 1–13. Springer, 2007.

[BDV08]   Bruno Bougard, Bjorn De Sutter, Diederik Verkest, Liesbet Van der Perre, and Rudy Lauwereins. "A Coarse-Grained Array Accelerator for Software-Defined Radio Baseband Processing." *IEEE micro*, **28**(4):41–50, 2008.

[CCG13]   Yu-ting Chen, Jason Cong, Mohammad Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao, and Yi Zou. "Accelerator-Rich CMPs: From Concept to Real Hardware." In *International Conference on Computer Design*, 2013.

[CGG12a]   Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. "Architecture support for accelerator-rich cmps." In *Proceedings of the 49th Annual Design Automation Conference*, pp. 843–849. ACM, 2012.

[CGG12b]   Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. "CHARM: A composable heterogeneous accelerator-rich microprocessor." In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pp. 379–384. ACM, 2012.

[CHL11]   Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. "A reuse-aware prefetching scheme for scratchpad memory." In *Proceedings of the 48th Design Automation Conference*, pp. 960–965. ACM, 2011.

[CHM06]   N. Clark, A. Hormati, S. Mahlke, and S. Yehia. "Scalable Subgraph Mapping for Acyclic Computation Accelerators." In *Proc. CASES*, pp. 147–157, 2006.

[CJ08]   J. Cong and W. Jiang. "Pattern-Based Behavior Synthesis for FPGA Resource Reduction." In *International Symposium on FPGAs*, pp. 107–116, 2008.

[CLN11]   Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **30**(4):473–491, April 2011.

[CSR11]   Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. "Customizable domain-specific computing." *IEEE Design and Test of Computers*, **28**(2):6–15, 2011.

[FXB10]   H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. "Introduction to the wire-speed processor and architecture." *IBM Journal of Research and Development*, **54**(1):3:1–3:11, January 2010.

[GHN12]   Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing." *IEEE Micro*, **32**(5):0038–51, 2012.

[GSB00]   Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. "PipeRench: A reconfigurable architecture and compiler." *Computer*, **33**(4):70–77, 2000.

[Har01]   R. Hartenstein. "Coarse grain reconfigurable architectures." In *Asia and South Pacific Design Automation Conference*, pp. 564–569, 2001.

[htt]   http://llvm.cs.uiuc.edu. *The LLVM Compiler Infrastructure*.

[KR07]   Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(2):203–215, February 2007.

[LHW12]   Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. "The Accelerator Store: A Shared Memory Framework For Accelerator-Based Systems." *ACM Transactions on Architecture and Code Optimization*, **8**(4):1–22, January 2012.

[LWZ12]   Peng Li, Yuxin Wang, Peng Zhang, Guojie Luo, Tao Wang, and Jason Cong. "Memory partitioning and scheduling co-optimization in behavioral synthesis." In *Proceedings of the International Conference on Computer-Aided Design*, pp. 488–495. ACM, 2012.

[MD96]   Ethan Mirsky and André Dehon. "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources." In *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157–166, 1996.

[MSK99]   Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. "A reconfigurable arithmetic array for multimedia applications." In *International Symposium on FPGAs*, pp. 135–143, 1999.

[MVV03]   Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix." In *Field Programmable Logic and Application*, pp. 61–70. Springer, 2003.

[PPM09]   Hyunchul Park, Yongjun Park, and Scott Mahlke. "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications." In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 370–380. ACM, 2009.

[SLL00]   Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications." *Computers, IEEE Transactions on*, **49**(5):465–481, 2000.

[VNS07]   Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. "pn: A Tool for Improved Derivation of Process Networks." *EURASIP Journal on Embedded Systems*, **2007**:1–13, 2007.

[Wak68]   Abraham Waksman. "A permutation network." *Journal of the ACM (JACM)*, **15**(1):159–163, 1968.

[Xila]   Xilinx. "Virtex-6 FPGA DSP48E1 Slice.".

[Xilb]   Xilinx. "Vivado Design Suite.".

[Xilc]   Xilinx. "Vivado High-Level Synthesis.".

[Xild]   Xilinx. "Zynq-7000 All Programmable SoC.".