Data Locality Exploitation for Coarse-grained Reconfigurable Architecture in a Reconfigurable Network-on-Chips

Hung K. NGUYEN[†] [‡], Quang-Vinh TRAN[†], and Xuan-Tu TRAN[†]

[†]SIS Laboratory, VNU University of Engineering and Technology, 144 Xuan Thuy road, Cau Giay, Hanoi, Vietnam

E-mail: [‡]kiemhung@vnu.edu.vn

Abstract This paper proposes a Coarse-grained Reconfigurable Architecture (CGRA) applied to the multimedia processing and communications processing. To solve the huge bandwidth requirement of parallel processing arrays, the proposed CGRA architecture focuses on the exploitation of data locality to reduce data access bandwidth and increase efficiency of pipelined execution of the kernel loops. The proposed architecture has been modeled using both C and VHDL language aiming at simulating and analyzing various parameters of the target architecture, as well as supporting hardware/software co-verification when mapping applications onto the target system. Some benchmark applications have been mapped onto the models of the CGRA in order to prove the high flexibility and performance of the architecture that is suitable for a wide range of multimedia and communications processing applications. The proposed CGRA can be applied as computing resources in reconfigurable Network-on-Chips.

Keyword CGRA, Data locality, Reconfigurable computing, parallel processing, Network-on-Chip.

1. Introduction

Implementation of modern hand-held mobile devices always gives designers some challenges such as reducing chip area and power consumption, increasing application performance, shortening time-to-market, and simplifying the updating process. Besides, these systems are often designed not only for a specific application but also for multiple applications. Such sharing of resources by several applications makes the system cheaper and more versatile. Application Specific Integrated Circuits (ASICs), Digital Signal Processors (DSPs), and Application-Specific Instruction Set Processors (ASIPs), have been used for implementing the mobile multimedia systems. However, none of them meets all of the above challenges [1]. Recently, a very promising solution was the reconfigurable computing systems that are integrated many heterogeneous processing resources such as software programable processors, hardwired IP (Intellectual Property) core, reconfigurable hardware architectures, etc. based on a flexible communication infrastructure in the form of Network-on-Chip (NoC) [3] as shown in Figure 1. To program such a system, an application is first represented intermediately as a series of tasks that depends on each other by Control and Data Flow Graph (CDFG) [2], and then partitioned and mapped onto the heterogeneous computational and routing resource of the system. Especially, computation-intensive kernel functions of the application are mapped onto the reconfigurable hardwares so that they can achieve high performance approximately equivalent to that of ASIC while maintaining a degree of flexibility close to that of DSP processors. By dynamically reconfiguring hardware, reconfigurable computing systems allow many hardware tasks to be mapped onto the same hardware platform, thus reducing the area and power consumption of the design [4].

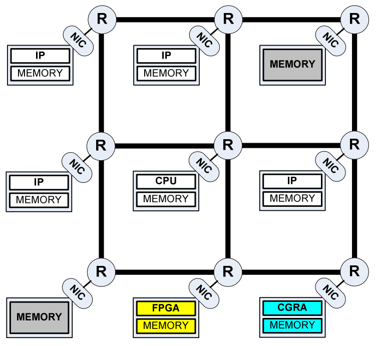

Figure 1. System-level application model of CGRA

The reconfigurable hardware architecture is generally classified into the Field Programmable Gate Array (FPGA) and coarse-grained dynamically reconfigurable architecture (CGRA). FPGA supports the fine-grained reconfigurable fabric that can operate and be configurated at bit-level. FPGAs are extremly flexible due to their higher reconfigurable capability. However, the FPGAs consume more power and have more delay and area overhead due to greater quantity of routing required per configuration [5]. This limits the capability to apply FPGA to mobile devices. To overcome the limitation of conventional microprocessors and fine-grained

Copyright ©2014 by IEICE

reconfigurable devices in the field of multimedia and communication baseband processing, we developed and modeled a CGRA architecture. In contrast with FPGAs, the CGRA aims at reconfiguring and manipulating on data at word-level. The CGRA is proposed to exploit high data-level parallelism (DLP), instruction-level parallelism (ILP) and TLP (Task Level Parallelism) of the computation-intensive loops of an application. The CGRA also supports the capability of dynamic reconfiguration by enabling the hardware fabrics to reconfigure into different functions even if the system is working. By dynamically reconfiguring the hardware, many different functions are mapped to the same hardware structure, thus leading to a reduction in size, cost and power consumption of the system. Consequently, high flexibility and performance, and low power consumption of the CGRA make itself ideal to satisfy the design requirements of multimedia processing applications.

The rest of the paper is organized as follows. Architecture and application model of the proposed CGRA at Network-on-Chip level are presented in Section 2. In Section 3, mapping of some benchmark examples and evaluation of the CGRA in terms of flexibility and performance is introduced. Finally, some conclusions are given in Section 4.

2. Architecture

2.1. Coarse-grained Reconfigurable Architecture

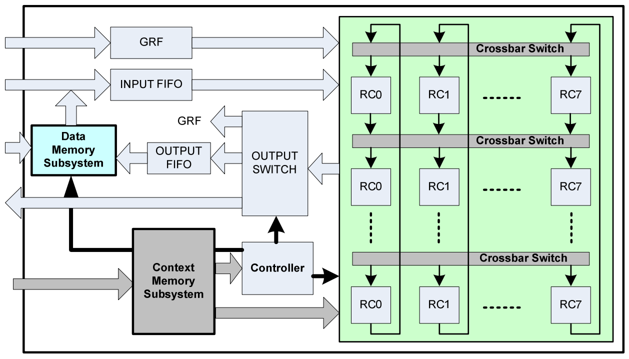

Figure 2. CGRA architecture

The CGRA consists of an Reconfigurable Computing Array (RCAs), an Input FIFO, an Output FIFO, Global Register File (GRF), Data/Context memory subsystem, and a controller, etc. (Figure 2). In turn, the RCA is an array of 8x8 RCs (Reconfigurable Cells), and can configure partially to implement computation-intensive tasks. The input and output FIFO is the I/O buffers between external data flow and RCA. Each RC can get data from the input FIFO or/and GRFs, and store data back to the output FIFO. These FIFOs are all 512-bit in width and 8-row in depth, and can load/store sixty-four bytes or thirty-two 16-bit words per cycle. Especially, the input FIFO can broadcast data to every RC that have been configured to receive the data from the input FIFO. This mechanism aims at exploiting the reusable data between iterations. The interconnection between two neighboring rows of RCs is implemented by the crossbar switches. Through the crossbar switch, an RC can get results that come from an arbitrary RC in the immediately above row of it. The Controller generates the control signals that maintain execution of RCA accurately and automatically according to configuration information in the Context Registers. The architecture of RCA core is basically loop-oriented one. Executing model of the RCA core is pipelined multi-instruction-multi-data (MIMD) model. In this model, each RC can be configured separately to process its own instructions, and each row of RCs corresponds to a stage of pipeline. Multiple iterations of a loop are possible to execute simultaneously in the pipeline.

RC is the basic processing unit of RCA. Each RC includes the data-path that can execute signed/unsigned fixed-point 8/16-bit operations with two/three source operands, such as arithmetic and logical operations, multiplier, and multimedia application-specific operations (e.g. barrel shift, shift and round, absolute differences, etc.). Each RC also includes a local register called LCR. This register can be used either to adjust operating cycles of the pipeline when a loop is mapped onto the RCA, or to store coefficients during executing an RCA core loop.

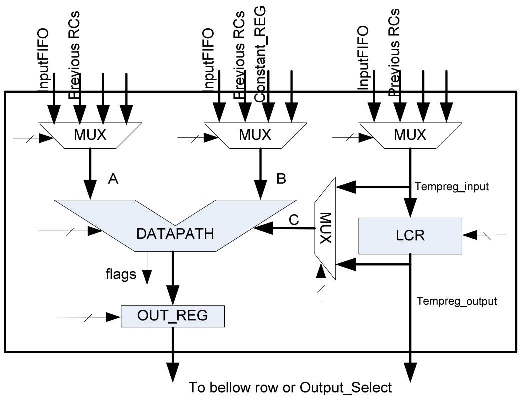

Figure 3. RC architecture

The data processed by RCA are clasified into two types: variables are streamed into RCA through the input FIFO, meanwhile constants are fed into the RCA via either GRF for scalar constants or LCR array for array constants. The Constant type is again classified into global constant and local constant. Global constant is determined at compile-time, therefore it is initialized in context memory of the CGRA at compile-time and loaded into GRF/LCRs while configuring the CGRA. Local constant (or immediate value) is not determined at compile-time, but is the result generated by other tasks at run-time, therefore it need to be loaded dynamically into GRF/LCRs by configuration words.

2.2. Configuration Model

The configuration information for CGRA is organized into the packet called context. The context specifies particular operation of the RCA core (i.e. operation of each RC, interconnection between RCs, input source, output location, etc.) as well as the control parameters that control operation of the RCA core. The total length of a context is 128 32-bit words. An application is composed of one or more contexts that are stored into the context memory of CGRA.

The function of CGRA is reconfigured dynamically in run-time according to the required hardware tasks. To deal with the huge configuration overhead in CGRAs, hardware design of the proposed CGRA supports a mechanism to pre-load and pre-decode the configuration context from external memory to the context memory of CGRA. By this method, the configuration of the CGRA can take place behind the calculation of RCA. As a result, once the RCA finish calculating with the current context, it can be immediately changed into the next context.

2.3. Execution Model

It is a well-know rule of thumb that 90% of the execution time of a program is spent by 10% of the code of LOOP constructs [4]. These LOOP constructs are generally identified as kernel loops. Most of them have computation-intensive and data-parallel characteristics with high regularity, so they can be accelerated by hardware circuits. The CGRA architecture is basically the such-loop-oriented one. By mapping the body of the kernel loop onto the RCA, the RCA just needs configuring one time for executing multiple times, therefor it can improve the efficiency of the application execution. Executing model of the RCA is the pipelined multi-instruction-multi-data (MIMD) model. In this model, each RC can be configured separately to a certain operation, and each row of RCs corresponds to a stage of a pipeline. Multiple iterations of a loop are possible to execute simultaneously in the pipeline.

For purpose of mapping, a kernel loop is first analyzed and loop transformed (e.g. loop unrolling, loop pipelining, loop blocking, etc.) in order to expose inherent parallelism and data locality that is then exploited to maximize the computation performance on the target architecture. Next, the body of the loop is represented by data-flow graphs (DFGs) as shown in Figure 4. Thereafter, DFGs are mapped onto RCA by generating configuration information, which relate to binding nodes to RCs and edges to interconnections. Finally, these DFGs are scheduled in order to execute automatically on RCA by generating the corresponding control parameters for the CGRA's controller. Once configured for a certain loop, RCA operates as the hardware dedicated for this loop. When all iterations of loop have completed, this loop is removed from the RCA, and the other loops are mapped onto the RCA.

In architecture model of RCA core, five parameters are used for controlling the execution of a loop on RCA as follows:

- *Input count* (N_I) is defined as the number of cycles spent to take data from the input FIFO.
- *Output count* (N_O) is defined as the number of cycles spent to write the entire results of one loop iteration to Output FIFO.
- *Iteration count* (N_L) is the number of the loop iterations.
- *Iteration Interval* (II) is defined as the number of cycles calculated from the 1st input of the ith iteration to the 1st input of the (i+1)th iteration.
- *Output Time* (N_W) is defined as the number of clock cycles between the first output and the first input in the same iteration.

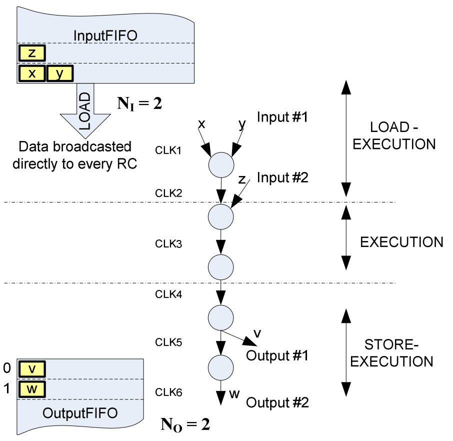

Figure 4. DFG representation of a simple loop body

The execution of a loop is scheduled so that different phases of successive iterations are overlapped each other as much as possible. As defined, *II* indicates when the next iteration is possible to start. The smaller *II* is, the sooner next iteration starts, thereby the more iterations of the loop are possible to execute simultaneously in the pipeline. Consequently, achieving the minimum *II* value is the object of optimization to improve execution performance of a loop on RCA. Scheduling also needs to ensure that there are not any conflicts between resources as multiple phases take place simultaneously.

Parallel processing increases computation performance but also increase the pressure on data bandwidth. The system's bandwidth is necessary to ensure that data is always available for all resources running concurrently without IDLE state. A way for increase data availability is to exploit *Data locality* that refers to capability of data reuse within a short period of time [6]. Exploiting the data locality has the potential to increase the processing efficiency of the system because the data can be cached in the internal memory for reuse later, thus reducing stalled times due to waiting for external memory accesses. Moreover, the data reuse also has the potential to minimize the number of access to external memory, thus achieves a significant reduction in the power consumption [7]. Compared with the execution model in [8] and [9], our execution model exploits the overlapping data between two successive iterations, so it can enhance performance and reduce input data bandwidth. In this model, RCA core can start computing as soon as the data of the first input appears on the input of the RCA, so LOAD phase and EXECUTION phase of the same iteration can happen simultaneously. In other words, the modified execution model allows overlapping three phases LOAD, EXECUTION, STORE of the same iteration as much as possible. As shown in Figure 4, an iteration of RCA core in our model is started by LOAD-EXECUTION phase, and then is EXECUTION phase, finally finished by STORE-EXECUTION phase. On the other hand, our model also allows the data of the next iteration be LOADed simultaneously with the data of the current iteration, so it not only maximize the degree of overlapping between the consecutive iterations but also maximize the data reuse.

3. Verification and Evaluation

An environment for developing and verifying application is built at the different abstract levels. Firstly, a C model is used for hardware/software co-verifying an algorithm on the CGRA. Secondly, a cycle-accurate RTL (Register Transfer Level) model, which is written in VHDL language, is used for evaluating performance of the algorithm.

To evaluate performance and flexibility of the CGRA architecture, the section presents mapping some benchmark examples in the field of the digital signal and multimedia processing, including motion estimation (ME) algorithm, convolution, and matrix-vector multiplication, onto the CGRA. Although the actual size of the RCA array is 8×8, but for simplifying presentation in this section some figures will only illustrate the RCA array with size of 4×4.

3.1. Mapping of benchmark examples

A) Matrix – Vector Multiplication

The multiplication of an M×N matrix by a length-N vector is represented by the following equations:

$$\begin{bmatrix} Z_1 \\ \vdots \\ Z_2 \\ Z_m \end{bmatrix} = \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ X_{m1} & X_{m2} & \dots & X_{mn} \end{bmatrix} \times \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} \qquad (1)$$

Figure 5(a) shows a DFG for multiplying a 4×4-matrix with a 4×1-vector. Mapping of DFG onto the RCA array and its pipelined execution is shown in Figure 5(b-c). Since the vector Y is used repeatedly for computing every value Z_i, therefore its value will be loaded into registers of GRF before starting the calculating process. As a result, the amount of memory access will be significantly reduced.

B) Convolution

Convolution is a mathematical operation on two functions and producing a third function that is typically viewed as a modified version of one of the original functions. It is applied popularly in field of the image and signal processing. A well-known application of convolution is FIR filter as shown in Figure 6. Direct form realization of 4-tap FIR filter is based on the direct implementation of (2):

$$y[n] = \sum_{k=0}^{N-1} h[k] * x[n-k] \qquad (2)$$

where, x[k] are input samples (x[k]=0 ∀k<0); h[k] are the coefficients of frequency response; and y[n] are output samples.

As shown in Figure 6(a), the DFG for 4-tap FIR filter is similar to the DFG for matrix–vector multiplication. However, note that there are three values of the sequence x[n] that are repeatly used to calculate the two consecutive values of the sequence y[n]. Therefor, in order to exploit these reused data between successive iterations, the DFG as shown in Figure 6(b) will be used. In this DFG, the samples x[n] will be multiplied by the coefficients h[n] and then accumulated together in the result z[n] according to the method of 4-stage pipelined execution. The samples x[n] will be streamed by the input FIFO and broadcasted to all the chosen RCs. As a result, only one sample x[n] is outputed by the input FIFO at the moment. Moreover, because each sample x[n] is accessed only once, the memory access bandwidth will be reduced significantly.

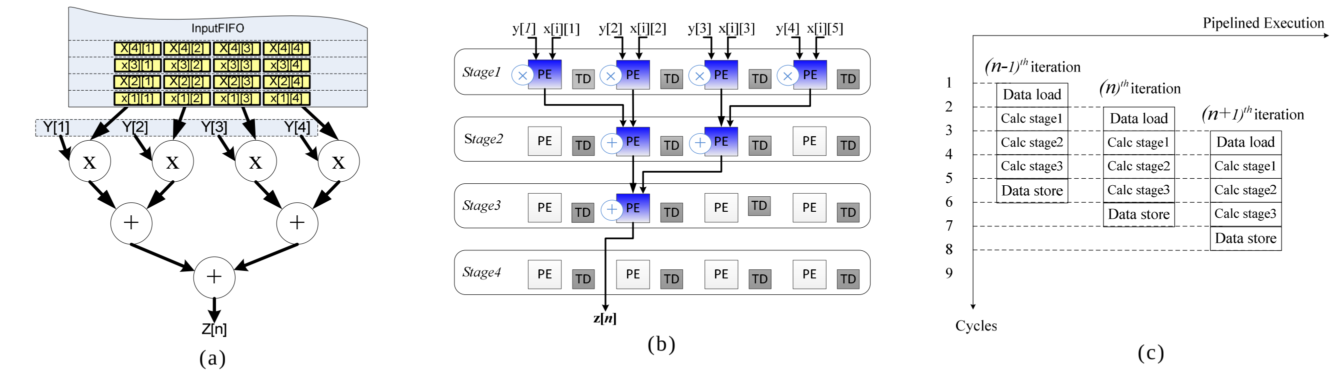

Figure 5. DFG DFG (a), mapping of DFG on CGRA (b), and pipelined execution (c) of matrix–vector multiplication

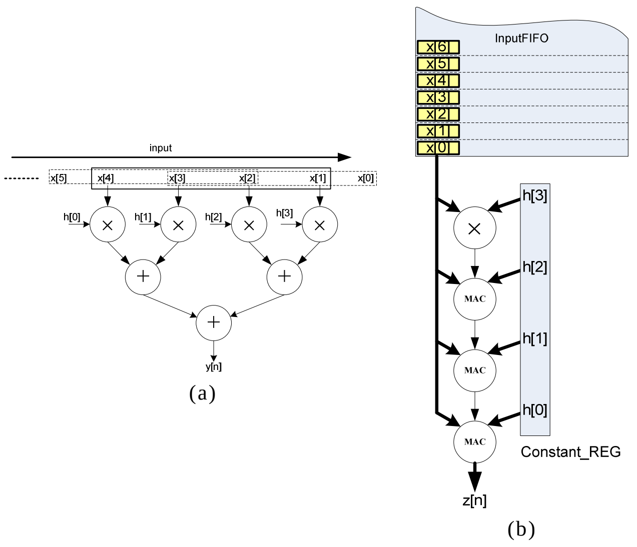

Figure 6. DFG for implementing a 4-tap FIR filter

C) *Motion Estimation*

Motion Estimation (ME) is the powerful tool used popularly in the latest video coding standard such as H.264/AVC [10]. ME exploits temporal redundancy of a video sequence by finding the best matching candidate block of each current 16×16-pixel macroblock from a search window in reference frames. The two operations that have the highest complexity of ME are the sum of absolute differences (SAD) and Sum of absolute Hadamard transformed values of residues (SATD). Because the 4×4 block is the smallest block size supported by almost all of standards, the SA(T)D of the 4×4-blocks are first calculated, and then these results are used to calculate SA(T)D for the larger block sizes.

SAD Computation: Figure 7(a) presents a DFG for computing SAD of a 4×4-block (SAD4×4). In this diagram, the absolute difference of each row of one 4×4-block is performed independently on the consecutive stages of the pipeline. Scheduling of the data and computation sequence is shown in Figure 7(b-c). 4×4 current pixels are used to initialize registers in GRF, whereas one row of the 4 reference pixels (i.e. P_i) is fed to RCA via the input FIFO every cycle.

SATD Computation: The optimized DFG for computing SATD of a 4×4-block, which can be mapped completely onto one 8×8-RCA, is shown in Figure 8. A 4×4-block is first divided into two halves of 8 pixels to input to the DFG in sequential order. Eight RCs in the 1st row generate eight residues in parallel and then transmit them to 2-D Hadamard transform unit. The transformed residues of the 1st half are stored in LCRs (denoted as T in Figure 8) waiting for transformed residues of the 2nd half. Once residues of the 2nd half have transformed, they are

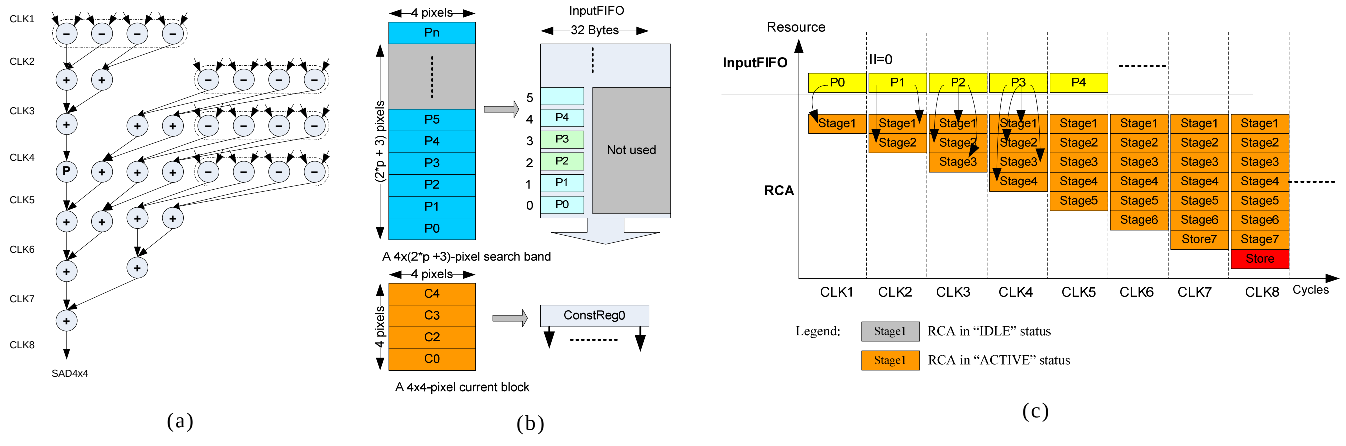

Figure 7. DFG (a), data organization (b) and scheduling (c) of SAD computation on CGRA

compared with the transformed residues of the 1st half to find maximum values. The maximum values then are transferred to the adder-tree in order compute SATD value, finishing computing SATD of a 4×4-block. The process is fully pipelined with the latency of eight cycles. No intermediate data is buffered when computing SATD of a 4×4-block, therefore, no additional internal memory is required.

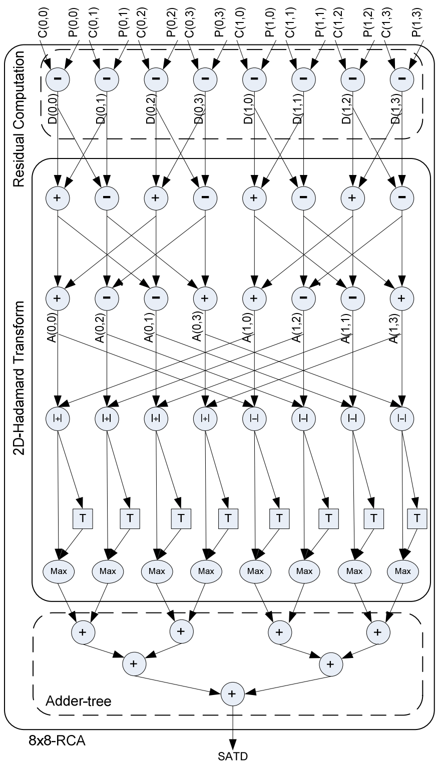

Figure 8. The DFG for SATD computation

3.2. Evaluation

Table 1 shows the parameters for evaluating the performance of four benchmark loops. Here, N_L is the number of loop iterations. ILP (Instruction Level Parallelism), LLP (Loop Level Parallelism) and TLP (Task Level Parallelism) denote the number of instructions, the number of iterations, and the number of task can be performed in parallel on RCA.

A) Matrix–vector multiplication and Convolution

Assuming that both matrix - vector multiplication and convolution use the same DFG as shown in Figure 5, it requires (N_L+3) cycles to complete the tasks with the amount of data had to be read from memory is N_L×4. If convolution is performed according to the DFG in Figure 6(b), the number of cycles to complete the computation is (N_L+4) cycles. However, by reusing the overlapping data between two consecutive iterations of the FIR filter, the proposed model allows the number of memory access to be reduced from N_L×4 to (N_L+3). This reduction is significant for large value of N_L, and has a potential to impact significantly on the memory bandwidth. Moreover, because only one byte instead of four bytes is read per cycle, it also has a potential to impact significantly on the data bus width for fetching the data to the RCA array while many 4-tap FIR filters are mapped concurrently onto the RCA. Input/output data stream is continuous, therefore it is possible to get a higher performance by utilizing 100% pipeline for computing. Compared to the execution model in [8], the proposed execution model exploits the overlapping data between two successive iterations, thereby it can reduce input data bandwidth, while maintaining computation performance approximately equivalent to that of [8].

B) SAD computation

The DFG in Figure 7 is an arrangement of RCs in order to achieve a higher data reuse ratio. Although this is not the optimized solution in terms of the length of pipeline, but the data structure for this solution is much simpler, while maintaining the high performance because pipeline utilization is 100% of the total operating time. Compared to the architecture in [8], Table 1 shows that the performance of the proposed architecture can be increased approximately twice, while the number of memory read access can be reduced approximately 4 times.

C) SATD computation

The DFG for computing SATD in Figure 8 illustrates the ability to support dual-loop calculation of the proposed CGRA architecture compared to the architecture in [8]. Instead of having to divide the loop body into two contexts as the architecture in [8], one iteration of the loop is mapped into two RCA's iterations without context switching. The executing process is continuous and does not require the memory for buffering intermediate results, therefore pipeline utilization is very high.

4. Conclusions

In this paper, we proposed the architecture, and then developed high-level models of a coarse-grained reconfigurable architecture (CGRA). We also demonstrated the executing model of the CGRA through mapping some examples onto the developed architecture. Simulation results show that the proposed CGRA can be reconfigured for a wide range of applications in the fields of multimedia processing and communications. The CGRA has the ability to exploit the parallel mechanism and data locality in algorithm to increase processing performance, as well as to reduce the memory access bandwidth. In the future, we will continue to optimize the proposed CGRA, and validate the CGRA at NoC level on a FPGA platform.

Table 1 Mapping result of kernel loops

Kernel Loop	**Matrix – vector**		**Convolution**		**SAD**		**SATD**	
	our	[8]	our	[8]	our	[8]	our	[8]
II	0	0	0	3	0	2	1	0
Pipeline Utilization (%)	100%	100%	100%	100%	100%	50%	100%	100%
ILP	7	7	4	7	31	31	47	64/15$^{(*)}$
LLP	5	5	6	5	8	4	5	4/4$^{(*)}$
TLP	6	6	16	6	1	1	1	2/1$^{(*)}$
Execution Time (cycles)	$N_L + 3$	$N_L + 3$	$N_L + 4$	$N_L + 3$	$N_L + 7$	$2\times N_L + 6$	$2\times(N_L + 4)$	$N_L + 5$
Memory access (times × bytes/time)	$N_L\times4$	$N_L\times4$	$N_L +3$	$N_L\times4$	$(N_L+3)\times4$	$N_L\times16$	$N_L\times16$	$\geq(N_L\times16)$

$^{(*)}$: context1/context2

Acknowledgement

This work has been supported by VNU, University of Engineering and Technology under Project No. CN.14.04 and Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.01-2013.17.

References

[1] Christophe Bobda, "Introduction to Reconfigurable Computing – Architectures, Algorithms, and Applications". Springer, 2007.

[2] João M. P. Cardoso Pedro C. Diniz: Compilation Techniques for Reconfigurable Architectures, 2009, Springer.

[3] Natalie Enright Jerger, Li-Shiuan Peh: On-Chip Networks, 2009 by Morgan & Claypool.

[4] A. Shoa and S. Shirani, "Run-Time Reconfigurable Systems for Digital Signal Processing Applications: A Survey", Journal of VLSI Signal Processing, Vol. 39, pp.213–235, 2005, Springer Science.

[5] G. Theodoridis, D. Soudris and S. Vassiliadis: "A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools Basic Definitions, Critical Design Issues and Existing Coarse-grain Reconfigurable Systems", Springer 2008, p89-149.

[6] Kathryn S. McKinley, Steve Carr, Chau-Wen Tseng: "Improving Data Locality with Loop Transformations", ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 18, Issue 4, July 1996, Pages 424 - 453.

[7] S. Sohoni, and R. Min, et al. "A study of memory system performance of multimedia applications". SIGMETRICS Performance 2001, pages 206–215.

[8] M. Zhu, L. Liu, S. Yin, et al.: "A Cycle-Accurate Simulator for a Reconfigurable Multi-Media System," IEICE Transactions on Information and Systems, vol. 93, pp. 3202-3210, 2010.

[9] Hung K. Nguyen, Peng Cao, Xuexiang Wang, Jun Yang, Longxing Shi, Min Zhu, Leibo Liu, Shaojun Wei: Hardware Software Co-design of H.264 Baseline Encoder on Coarse-Grained Dynamically Reconfigurable Computing System-on-Chip, IEICE TRANSACTIONS on Information and Systems (SCI index), Vol.E96-D, No.3, pp.601-615, 2013.

[10] Iain E. Richardson: "The H.264 advanced video compression standard", second edition, 2010, John Wiley & Sons, Ltd.