CONFIGURATION APPROACHES TO IMPROVE COMPUTING EFFICIENCY OF COARSE-GRAINED RECONFIGURABLE MULTIMEDIA PROCESSOR

Chen Yang, Leibo Liu*, Yansheng Wang, Shouyi Yin, Peng Cao and Shaojun Wei

1. Institute of Microelectronics, Tsinghua University, Beijing China
2. IC School of Southeast University, Nanjing Jiangsu, China
Email: chyang00@mails.tsinghua.edu.cn, liulb@tsinghua.edu.cn, wangys07@mails.tsinghua.edu.cn, yinsy@tsinghua.edu.cn, caopeng@seu.edu.cn, wsj@tsinghua.edu.cn

ABSTRACT

This paper proposes three configuration approaches to improve computing efficiency of a coarse-grained reconfigurable array, including input data relocation, line-based context switching, and loop interval minimization. These proposed approaches fully exploit the parallelism and pipelining of the reconfigurable array, which reduce interval latency when switching the configuration contexts, and therefore greatly enhance computing efficiency. These proposed techniques are used in a coarse-grained reconfigurable multimedia system (REMUS). Measured results show that, owing to the proposed approaches, REMUS can achieve 1080p@30fps performance for H.264 high profile video decoding under 200MHz working frequency. When normalized to the same technology, REMUS outperforms XPP-III 6.98x in energy efficiency.

1. INTRODUCTION

A coarse-grained reconfigurable array (CGRA) has become a potential solution for emerging multi-standards in the multimedia processing market. For instance, PipeRench [1], ADRES [2], DREAM [3], and XPP-III [4] have all been proven to be effective solutions for H.264 decoder. The CGRA has a similar mode of operation to that of the general purpose processor, where context is the control bit of the reconfigurable array. One context generally consists of a few hundred to a few thousand bytes, which induces large delay for context delivering. On the other hand, frequent context switching on the CGRA is required to perform different kernels of a complex application, which induces large delay for context switching. Consequently, the major bottleneck of a CGRA is the excessive configuration time. Many technologies have focused on removing this bottleneck by reducing context delivering time. In compression method [5], [6], contexts are compressed to reduce the number of reconfiguration cycles. Moreover, a double buffering memory and a mapping methodology [7] are used to reduce repeatable reading of the same data from the external memory.

The configuration technology is essential to reduce the context switching time, thereby enhancing the computing efficiency of the reconfigurable architecture. Hence, this paper proposes three configuration approaches, including input data relocation (IDR), line-based context switching (LCS), and loop interval minimization (LIM). These configuration techniques fully exploit the parallelism and pipelining of the reconfigurable array. In this way, interval latency is reduced when switching contexts, and therefore computing efficiency of the CGRA is enhanced greatly.

2. RECONFIGURABLE ARRAY

Before proposing the configuration techniques, the architecture of reconfigurable cell array (RCA) and reconfigurable processor unit (RPU) both designed by our research group is introduced in Fig. 1. RCA consists of an 8×8 processing elements (PEs) array with hierarchical context memory architecture. The entire contexts are loaded into the context memory at the boot phase. At run time, the hardware configuration can be changed on a cycle-by-cycle basis by broadcasting the pointer of a context to the context memory. RPU, consisting of up to 256 PEs, is a matrix of four RCAs. These four RCAs can be dynamically combined together subject to the requirement of calculation so as to achieve algorithm-level parallelism.

PE is the basic computing element of the RCA, as shown in Fig. 2. (a). It consists of three parts: input data MUX including MUX_A, MUX_B and MUX_T, a 16-bit temp register used to store intermediate result, and an ALU served as the basic calculation unit. All the input and output of the ALU are buffered by 16-bit pipeline registers. Each PE can be programmed to execute word or sub-word level operations, as illustrated in Fig. 2. (b). An interlayer mesh bus is employed as route to transmit data between adjacent rows. Discrete input FIFO and output FIFO are designed to prepare input data and output results for the RCA respectively. They are connected to each PE by a 256-bit
width data interface. In order to save the interconnection area, a line-switched mesh I/O structure is proposed as shown in Fig. 2. (c). In this structure, only one line of PEs and a corresponding line of FIFO data ports are selected and connected to the I/O bus at one configuration. After that, a full mesh network is utilized to transmit data between the I/O FIFO and PEs in the I/O bus.

3. INPUT DATA RELOCATION

To ensure the correctness of input data, the context cannot be switched until all the input data are consumed. If input data are not routed to the PEs in the top row of the RCA (also the first stage of the pipeline), the input data cannot be used at the first cycle and must be maintained until all the input data are consumed. IDR is an approach to relocate the input data to the temp registers within the PEs in the top row, thereby consuming all the input data at the first cycle. As a result, extra delay time before starting input data of the next context can be removed.

A 4×4 PEs array is taken as an example to explain how IDR works. The direct mapping of a data flow graph (DFG) on the 4×4 PEs array is shown in Fig. 3. (a). All the input data (IN0–IN9) are stored in the same entry of input FIFO and read out at the first cycle. The calculation process of four rows of PEs consumes 4 clock cycles; as a result, the output results (OUT0–OUT1) can be obtained at the sixth clock cycle. Overall, it takes a total of 6 clock cycles to perform the DFG. IN8 and IN9 are connected to the bottom-left and bottom-right PEs respectively in Fig. 3. (a). Therefore, IN8 and IN9 should be maintained until the fourth cycle of the pipeline, which induces extra 3 idle clock cycles before starting input data of the next context.

IDR approach relocates IN8 and IN9 to temp registers within the PEs in the top row. In order to clearly illustrate the data paths of buffering IN8 and IN9, we draw the temp register, which is a part of PE, separately as a striped box in Fig. 3. (b). Six temp registers in different rows are employed to make IN8 and IN9 delay 3 clock cycles. As a result, all of the input data are consumed at the first cycle and can be changed for the next context at the second cycle.

4. LINE-BASED CONTEXT SWITCHING

A commonly used method for context switching is by diagram. In this method, the reconfigurable array cannot start next context until current context is completed. When the input data of next context are independent of the output data of current context, LCS approach can switch contexts by line. The next context can be executed before current context is completed, which improves the utilization of the array. As a result, the reconfigurable array could execute multiple contexts simultaneously, thereby improving the performance of the RCA.

Taking a 4×4 PEs array as an example, two contexts of DFGs, i.e., context1 and context2, are shown in Fig. 4. Utilizing IDR approach, all the input data can be relocated to the top row of the DFG so that they are consumed at the first cycle. It needs 4 clock cycles to calculate the context1 and then obtains the output results (i.e., OUT0 and OUT1). As for context2, it takes 3 clock cycles to complete the calculation before the results (i.e., OUT2 and OUT3) are produced.

Assuming that context1 and context2 are sequentially executed on the 4×4 PEs array. The conventional approach for context scheduling is to switch contexts by DFGs, as illustrated in Fig. 5. (a). It takes a total of 11 clock cycles to perform context1 and context2. In order to obtain the output
results of context1 before those of context2, context2 can start at the third cycle of context1 pipeline by using LCS method. As shown in Fig. 5, (b), the top row of the 4×4 PEs array is switched to execute addition operations of context2 at the fourth cycle; at the same time, the third row of the 4×4 PEs array is processing subtraction and bypass operations of context1. As a result, the pipeline length of context1 and context2 can be reduced to 7 clock cycles.

5. LOOP INTERVAL MINIMIZATION

LIM is an approach of getting the optimal timing parameters to minimize interval of iterations, which is introduced by the example in Fig. 6. Symbols in, cl, hd, and ot represent executions at corresponding clock cycle. Symbol in and cl represents inputting data and calculation respectively. Symbol ot is outputting data, while symbol hd represents that current input data need to be maintained. There are four inputs (marked as 1~4) and one output (marked as 5) in Fig. 6. For each path (i.e., from input to output), the cycle numbers of input, calculation, and output are set to $i_j$, $c_j$, and $o_j$ respectively, where $j$ is the path number. As for the RCA, the total cycles required for inputting data are set to $I$. After that, the RCA needs $W$ cycles to compute the result. Finally, RCA requires total $O$ cycles to output results. Symbol $G$ is the interval between adjacent iterations.

Assuming that the DFG is executed for $N$ iterations on the RCA, the total time $T$ consumed is given by (1), where $I = \max(i_j)$ and $O = \max(o_j)$ respectively. $I + W + O$ are the execution time of the first iteration. As for each of the following $N-1$ iterations, only $G+I$ clock cycles are required. $N$ is determined by an algorithm and cannot be changed. Limited by the bandwidth of input FIFO and output FIFO, $I$ and $O$ are constant.

$$T = I + W + O + (G + I) \times (N - 1) \quad (1)$$

Therefore, $W$ and $G$ are two key timing parameters that affect the pipeline performance. In order to improve the performance, $W$ and $G$ should be as small as possible. However, $W$ and $G$ should be large enough to ensure the correctness of output results. Similarly to setup time in the sequential circuit, the computing time $W$ should be large enough to ensure that the calculation of current iteration is completed before starting output. Similarly to hold time in the sequential circuit, the interval time $G$ should be large enough to ensure that the output results of current iteration cannot be contaminated by the next iteration. Therefore, $W$ and $G$ should satisfy (2) and (3) respectively, where $s_j = i_j + c_j - o_j$.

$$I + W \geq \max(s_j) + 1 \quad (2)$$

$$I + G \geq \max(s_j) + 1 - \min(s_j) \quad (3)$$

Based on (2) and (3), the minimal computing time $W$ and the minimal interval time $G$ are classified referring to Fig. 7. $S_{max}$ and $S_{min}$ are the maximum and minimum value of $s_j$, respectively.

6. IMPLEMENTATION RESULTS

In order to confirm the efficiency of the aforementioned configuration approaches, they are used on the RPU. The performances of the sub-algorithms of H.264 decoding on the RPU with and without the proposed configuration approaches are compared in Table 1. This includes motion compensation (MC), deblocking filter and inverse discrete cosine transform (IDCT). Owing to the proposed configuration approaches, clock cycles per macro block (MB) are improved by 26.3%~49.9%.

Relying on the proposed RPU, VLSI architecture of a coarse-grained reconfigurable multimedia processor named...
REMUS is implemented by our research group [8], as shown in Fig. 8. (a). An ARM7 is responsible for application scheduling and other peripherals control. An EnD supports three standards of entropy decoder: H.264, AVS, and MPEG2. An upA is an array of ARM7 cores, which controls the configuration and data I/O of RPU. An EMI is a specifically designed 64-bit external memory interface. Two RPUs are the main computing engines in REMUS. Taking H.264 decoding algorithm as an example, RPU0 performs IDCT, intra-prediction, and MC, while RPU1 is dedicated to deblocking filter. The die photo and VLSI implement result of REMUS are shown in Fig. 8. (b) and Fig. 8. (c), respectively. Measured results show that 1080p@30fps of H.264 (High Profile) decoding can be achieved on REMUS when utilizing a 200MHz working frequency. As for AVS (Jizhun Profile) and MPEG2 (Main Profile), much higher performance, i.e., 1080p@39fps and 1080p@41fps, can be achieved on REMUS under 200 MHz, respectively.

The comparison of different architectures that realize H.264 decoding is given in Table 2. REMUS achieves a 1.81x faster decoding speed at the same frequency and a 14.3x energy efficiency improvement over XPP-III. When normalized to the same technology (65nm technology), REMUS still outperforms XPP-III 6.98x in energy efficiency. Compared with the state-of-the-art many-core processor and DSP, as expected, REMUS shows considerable advantages in many aspects.

---

### Table 1. Performance of Compute-intensive Sub-algorithms of H.264 Decoding on RPU.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Conventional approaches (cycles/MB)</th>
<th>Proposed IDR, LCS, and LIM (cycles/MB)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>628 (typical)</td>
<td>322 (typical)</td>
<td>1.95</td>
</tr>
<tr>
<td>Deblocking</td>
<td>739 (typical)</td>
<td>387 (typical)</td>
<td>1.91</td>
</tr>
<tr>
<td>IDCT 4x4</td>
<td>144 (typical/worst)</td>
<td>106 (typical/worst)</td>
<td>1.35</td>
</tr>
<tr>
<td>IDCT 8x8</td>
<td>322 (typical/worst)</td>
<td>192 (typical/worst)</td>
<td>1.69</td>
</tr>
</tbody>
</table>

### Table 2. H.264 Decoding Performance Comparison of Different Architectures.

<table>
<thead>
<tr>
<th>Technology (mm)</th>
<th>Area (mm²)</th>
<th>Frequency (MHz)</th>
<th>Resolution</th>
<th>Performance (fps)</th>
<th>Power (mW)</th>
<th>Normalized Performance¹</th>
<th>Energy Efficiency²</th>
<th>(MBs/s/MHz)</th>
<th>(MBs/s/mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMUS</td>
<td>64</td>
<td>90</td>
<td>140</td>
<td>40</td>
<td>130</td>
<td>0.466</td>
<td>490.1</td>
<td>30</td>
<td>529</td>
</tr>
<tr>
<td>XPP-III¹[4]</td>
<td>64</td>
<td>90</td>
<td>140</td>
<td>40</td>
<td>130</td>
<td>0.466</td>
<td>490.1</td>
<td>30</td>
<td>529</td>
</tr>
<tr>
<td>Many-core[9]</td>
<td>64</td>
<td>90</td>
<td>140</td>
<td>40</td>
<td>130</td>
<td>0.466</td>
<td>490.1</td>
<td>30</td>
<td>529</td>
</tr>
<tr>
<td>DSP[10]</td>
<td>64</td>
<td>90</td>
<td>140</td>
<td>40</td>
<td>130</td>
<td>0.466</td>
<td>490.1</td>
<td>30</td>
<td>529</td>
</tr>
</tbody>
</table>

¹Totally 8160 MBs per 1080p frame; ²Number of MBs per second per MHz; ³Number of MBs per second per mW.

### 7. REFERENCES


