Hardware Based Loop Optimization for CGRA Architectures

Chilankamol Sunny, Satyajit Das, Kevin Martin, Philippe Coussy

Hardware Based Loop Optimization for CGRA Architectures

Chilankamol Sunny[1], Satyajit Das[1][0000−0002−7550−2641], Kevin J. M. Martin[2][0000−0002−8122−1192], and Philippe Coussy[2][0000−0002−7222−5271]

[1] IIT Palakkad, Kerala, India
112004004@smail.iitpkd.ac.in, satyajitdas@iitpkd.ac.in
[2] Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France
{kevin.martin,philippe.coussy}@univ-ubs.fr

Abstract. With the increasing demand for high performance computing in application domains with stringent power budgets, coarse-grained reconfigurable array (CGRA) architectures have become a popular choice among researchers and manufacturers. Loops are the hot-spots of kernels running on CGRAs and hence several techniques have been devised to optimize the loop execution. However, works in this direction are predominantly software-based solutions. This paper addresses the optimization opportunities at a deeper level and introduces a hardware based loop control mechanism that can support arbitrarily nested loops up to four levels. Major contributions of this work are, a lightweight Hardware Loop Block (HLB) for CGRAs that eliminates control instruction overhead of loops and an acyclic graph transformation that removes loop branches from the application CDFG. When tested on a set of kernels chosen from various application domains, the design could achieve a maximum of 1.9× and an average of 1.5× speed-up against the conventional approach. The total number of instructions executed is reduced to half for almost all the kernels with an area and power consumption overhead of 2.6% and 0.8% respectively.

Keywords: Coarse grained reconfigurable array (CGRA) · Loop optimization · Hardware loop · CDFG transformation.

1 Introduction

Coarse Grained Reconfigurable Array (CGRA) architectures have proven to be good targets as specialized hardware in low power embedded applications, such as Wireless Sensor Networks (WSN), Internet of Things (IoT) and Cyber Physical Systems (CPS). Due to the regular structure with several highly optimized processing elements (PEs) and simple interconnects, CGRAs provide high performance and energy efficiency. Analyzing the energy consumption of the PEs, it is inferred that the scope of improvement lies in the control path rather than on individual units [16]. Since CGRAs are mostly used to accelerate the compute-intensive portions of an application, specifically the loops, optimizing the control path for loops will result in significant improvement in energy efficiency.

Widely reported works in CGRA implementations [5] [11] [14] rely on software based loop optimizations to reduce the control flow bottleneck. Popular

techniques like loop unrolling [16] [6], modulo scheduling [13] [8] and polyhedral model based optimizations [10] are broadly used in CGRAs. All these approaches exploit the parallelism available in the innermost loop body and reduce the number of branch execution by transforming the loop to a data flow graph (DFG). However, with the growing complexity of the applications, the loop bounds are becoming larger and number of loop nests as well as total number of loops in an application are increasing. Reducing only the branch instruction in the innermost loop is not sufficient anymore to get high energy efficiency for a whole application. To deal with outer loops, additional instructions to perform iteration counter update, terminating condition checking and branching are required. These additional instructions diminish the overall energy efficiency. Therefore, any effort in the direction of eliminating these non-contributing loop control instructions will aid in optimizing the control path of the whole application and achieving better performance and energy gain.

In this paper, we introduce a novel hardware based loop optimization technique for CGRAs which eliminates the control flow bottleneck in loops. As the target CGRA, we choose the state-of-the-art Integrated Programmable Array (IPA) architecture [3]. In this CGRA, to optimize the loop control bottleneck, loop unrolling is used. On top of this we propose the hardware based loop optimization to enhance the overall performance and energy efficiency. As of now, the optimization is limited to loops with statically known bounds. Experimental results show that the proposed solution reduces the total number of instructions executed to half of that in the baseline software based implementation and achieves a maximum of 1.9× performance gain for a wide range of image and signal processing applications. The first major contribution of this work is a novel lightweight Hardware Loop Block (HLB) for CGRAs which eliminates branch instruction overhead for loop increments or decrements. The second major contribution is in the compilation flow. We propose an acyclic graph transformation in the compilation to transform the software loops in the application CDFG eliminating loop branches. Experiments show that the total number of branch instruction execution is eliminated up to an average of 30× compared to that of the software based approach. Synthesis results show that the added hardware in the baseline architecture brings an area and a power consumption overhead of 2.6% and 0.8% respectively.

The rest of the paper is organized as follows. Section 2 presents the related works on loop optimizations with hardware support, from both general purpose processor and CGRA architectural domains. Section 3 introduces the baseline architecture and loop model and explains the motivation behind this work. Section 4 is dedicated to detail the proposed approach and present the implementation. In Section 5 experimental results are discussed and section 6 concludes the paper.

2 Related Works

Hardware based loop optimizations primarily are of two types, zero overhead looping extensions and instruction memory hierarchy optimizations [16]. Zero overhead looping refers to automatically updating the iteration count and taking branch decisions by using dedicated hardware units. One major optimization

on instruction memory hierarchy favoring loops is to include loop buffers. Loop buffers cache the instructions that make up the loop body to reduce the instruction fetch cost. We have seen hardware loop support on processors from the early x86 processor, the ISA of which included *loop* instruction and *rep* instruction prefix. The *rep* prefix triggered repeated execution of a single instruction. Many early ISAs included *rep*-like prefixes to support single cycle loops. Program address based and instruction count based zero overhead loop accelerations were proposed for DSP [15] and RISC [9] architectures. They could even provide multi-level loop support by using stack or scratch pad memory. In VLIW architectures [12], a special hardware unit is attached with every issue slot to automatically generate the instruction address. By employing loop buffers, DSP [1] and x86 processors could significantly reduce their energy consumption. The parallel ultra-low power (PULP) cluster architecture [7] is a multi-core platform with hardware support for loops. PULP implemented hardware loop by including an additional controller and a set of registers to store the loop information to its RISC-V cores. By having two sets of dedicated registers, the design could support two level nesting of loops. The instruction pre-fetch-buffer of the architecture acts as a loop cache if the loop body would fit into it, amplifying the effect of the hardware loop.

State-of-the-art CGRA architectures are adopting to implement hardware based loop solutions. Vadivel et al. [16] proposed a zero-overhead loop accelerator (ZOLA) for CGRA architectures. It supports nesting of loops up to four levels. ZOLA is enabled using a custom instruction after configuring all the loop parameters in the configuration register. The configuration of inner loops are kept intact until the outermost loop finishes execution. This prevents the reuse of configuration registers within a nested loop. Hence the total number of loops in a nested structure is limited by the number of configuration registers. For instance, in a ZOLA implementation employing four configuration registers, a nested loop of depth (number of levels) four can have only one loop at each level. In this paper, we propose a hardware based loop implementation which can support up to four levels of nesting and arbitrary number of sibling loops. The term sibling loops refers to loops at the same level or having the same loop as immediate parent. Furthermore, loop configuration is done inline with the execution eliminating the need for an extra configuration phase. ZOLA tries to reduce energy consumption by freeing the extra hardware units its reference architecture employs to implement loop control flow. However, the replacement of units is possible only when they are not used for any other computation. The solution we propose aims to minimize the number of loop control instructions executed by the baseline model, rather than trying to reduce the PE usage.

LASER [2] is another attempt to take hardware support for optimizing the loop execution. It is a hardware-software hybrid solution designed particularly for loops with imperfect nesting and random nesting of conditionals. Nesting of loops is called perfect if all the assignment instructions are inside the innermost loop; otherwise it is called imperfect nesting. In LASER, the compiler flattens nested loops into a single loop with conditional statements and fuses the true-path and false-path operation nodes. The instruction fetch unit (IFU) is modified to select and execute either the true-path or the false-path instruction based on the branch outcome. By enhancing IFU, LASER tries to better support loops

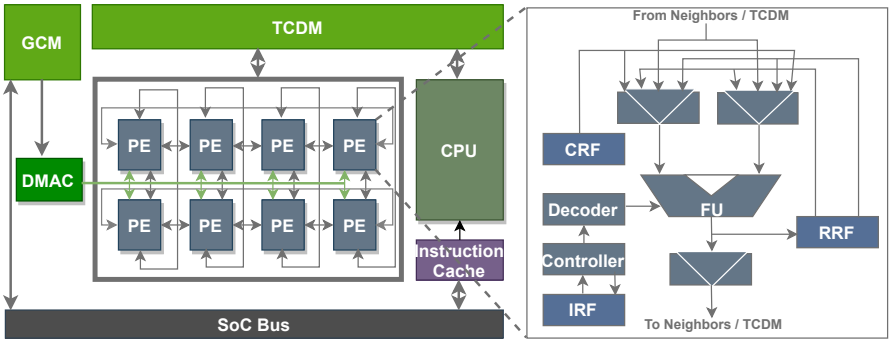

Fig. 1: IPA Integrated System

with conditional blocks rather than optimizing the general execution of loops. The basic implementation of looping mechanism remains to be software based. It does not eliminate the condition checking or branch instructions. The solution we propose eliminates those loop control instructions.

3 Background and Motivation

This section gives an introduction to IPA [5], the baseline architecture used to implement our design. The software based loop model of IPA as well as the problem statement is discussed in this section.

3.1 Architecture

The IPA integrated system consists of an array of inter-connected homogeneous processing elements (PEs), a global configuration memory (GCM), and a DMA controller. It is loosely coupled with a host CPU. A multi-banked tightly coupled data memory (TCDM) facilitates data communication between IPA and the CPU. Fig. 1 gives the IPA integrated system architecture. Each PE in the PE array houses a functional unit (FU), a controller, a decoder, an instruction register file (IRF), a regular register file (RRF), a constant register file (CRF) and router connections at the input and output. Global configuration memory stores the configurations (instructions and constants) for all the PEs in the PE array, which will be loaded to IRFs and CRFs of individual PEs, prior to execution. At each clock cycle, the PE controller fetches an instruction from IRF and FU executes it to completion.

3.2 Baseline Loop Model

IPA manages loop control flow by supporting branch instructions. The innermost loop is partially unrolled to exploit instruction and data level parallelism. However, once the loop body completes an iteration, the iteration counter is updated, an unconditional jump to the instruction which checks the terminating condition is performed, the condition check is done and then a conditional jump is performed either to the loop start or to the subsequent instruction in the

kernel. Evidently, the counter update, condition checking and branching instructions which contribute only to the loop control mechanism are executed as many times as does the loop body. Performance bottleneck imposed by this overhead can be promptly inferred by comparing the number of loop control instructions against the number of kernel instructions executed. For an imperfectly-nested loop structure of depth d, total number of kernel instructions executed (N_{ke}) can be represented as:

$$N_{ke} = \sum_{i=1}^{d} \sum_{j=1}^{n} p * \prod_{k=1}^{i} q' \tag{1}$$

Total number of loop control instructions executed (N_{lc}) can be computed as:

$$N_{lc} = q * \sum_{i=1}^{d} \sum_{j=1}^{n} \prod_{k=1}^{i} q' \tag{2}$$

where n is the number of loops at level i, p is the number of kernel instructions in j^{th} loop at level i, q' is the loop count of loop at level k and q is the number of loop control instructions executed per iteration. Loop count is the number of times the loop is to be executed. In the proposed design, we target to eliminate the loop control instruction to improve performance and energy efficiency.

4 Proposed Architecture and Compilation Flow

We propose a novel hardware based approach to implement loop control structure on CGRAs. Features which distinguish this design from other solutions are:

i Support for arbitrarily nested loops: the proposed solution can handle loop nesting up to four levels, which suffices for the studied benchmarks. The loops can be perfectly or imperfectly nested. It can handle arbitrary number of sibling loops.
ii Synchronous termination of loops at multiple levels: if an instruction happens to be the last instruction of multiple loops in the nested structure, the iteration count of each of those loops are updated in a single cycle in which the innermost loop terminates. If it was the last iteration of those loops, then they all will be terminated instantly.
iii On the go loop configuration: loop configuration is done inline with the execution. A separate pass through the loop structure is not required to set the configuration registers.

The baseline implementation is extended to provide hardware loop support by introducing a Hardware Loop Block (HLB) in the architecture and a premapping CDFG transformation in the compilation flow. Two dedicated instructions, $LOOP\_INIT$ and $LOOP\_CNT$ are also provided to initialize and configure the loops. $LOOP\_INIT$ carries the start and end addresses and $LOOP\_CNT$ holds the loop iteration count.

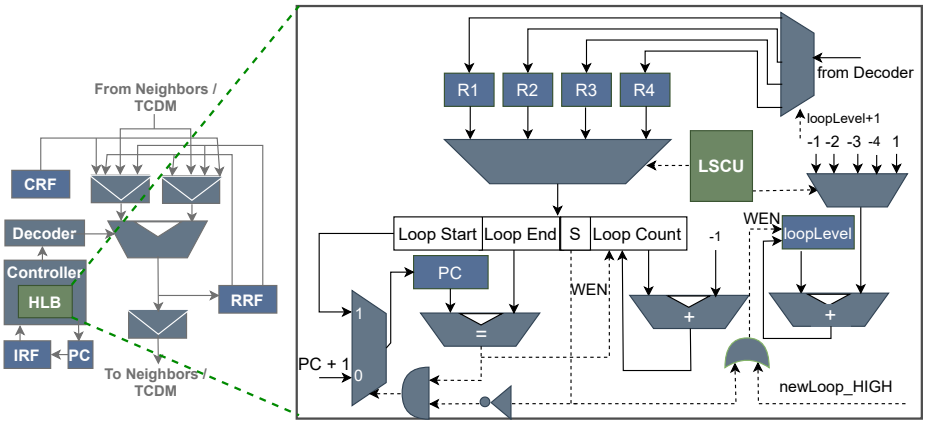

Fig. 2: Baseline PE Architecture extended with HLB

4.1 Hardware Loop Block (HLB)

Hardware loop units are incorporated into each PE in the PE array, modifying its architecture as shown in Fig. 2. In the individual PEs, HLB is integrated with the Controller, redefining its branch unit. The branch unit is responsible for updating the program counter (PC), which points to the next instruction to be fetched from the IRF. Loop characteristics such as start address, end address and iteration count are used to configure the loop and need to be stored in registers. To support four levels of nested loop, HLB reserves four configuration registers, one for each level and an extra register for loopLevel which holds the current level number. Each time a new loop is encountered, loopLevel is incremented by one, signifying a move one level deeper and when a loop runs to completion, loopLevel is decremented by one indicating a move up to the parent loop. At each level, the corresponding loop configuration register gets active. This register selection is controlled by the Loop Select Control Unit (LSCU). For instance, when a loop at level 2 is in execution, LSCU chooses the configuration register, R2 (Fig. 2). For every loop that appears at a particular level, the same register is reused to hold the configuration data facilitating the support for arbitrary number of sibling loops at each level.

At each clock cycle, the design compares the current PC value to the configured loop end instruction address to detect the branch-point. Until PC reaches the loop end, execution happens sequentially incrementing PC by one. When the PC and end address values evaluate to be equal, the loop count value is decremented by one and the PC is set to loop start address as long as the loop count is a non-zero value. Loop count becoming zero implies that the current loop has run to its completion and hence it is set as a trigger to decrement loopLevel. Consequently, the parent loop parameters become the current loop configuration, if there is any. PC is updated as PC + 1 which takes the control to its parent loop or to the rest of the kernel as the case may be. If the loop at hand was the outermost loop, then loopLevel will be zero indicating that no loop is in execution.

When a child loop finishes its execution, the design compares PC with the end address of parent loop as well. This is to support synchronous configuration

update and/or termination of loops sharing same end address. If the PC value and the parent end address are found to be the same, the loop count of the parent loop is decremented. If the parent loop was in its last iteration, then the loopLevel decrement will be by two. If not, loopLevel will be decremented by one and PC will be updated with the loop start address of the parent loop. This checking repeats for the subsequent levels as well causing loopLevel to be decremented by one, two, three or four depending on the scenario. LSCU facilitates this conditional update on PC and loopLevel. This feature allows the required termination of loops and/or loop count update at multiple levels in a single cycle, benefit of which is particularly evident in the case of perfectly nested loops. It is evident that current PC value and the loop end/start addresses are involved in choosing the next instruction. Since these addresses may vary across PEs, a completely centralized implementation of HLB is not possible. The baseline compilation flow and the required modifications are discussed in the next section.

4.2 Compilation flow

IPA supports block oriented execution of applications. It implies that the kernel to be accelerated is partitioned into single-entry-single-exit blocks of instructions called basic blocks (BBs) and only one basic block is executed at a time. Therefore the configuration as well as the assembly code are prepared in a block by block fashion. The IPA compiler takes as input the CGRA model and the high level application expressed as a control and data flow graph (CDFG) in which nodes represent BBs and edges represent the control flow between them. Each BB itself is represented as a data flow graph (DFG), BB = (D, O, A) where D is the set of data nodes, O is the set of operation/activity nodes and A is the set of arcs representing data dependencies. PE array (PEA) of the IPA is modelled by a bipartite directed graph of operator nodes and register nodes. CGRA model is the time extended model of PEA in which the timing is represented by connections between registers and operators. It is homomorphic to the basic block DFG and hence the mapping of a DFG onto the PEA is equivalent to finding a DFG in the PEA graph [4].

The baseline compilation flow (*Fig.* 3) starts with generating a CDFG from the application expressed in C language. This is done using GCC 4.8. Next step is to select the BB to be mapped, for which depth first search (DFS) strategy is used. The first BB is mapped without any constraints. However, to map the subsequent BBs, the constraints imposed by the maps of previously mapped BBs are to be considered. For this reason, the first map out of the many mappings generated for the previous BB is chosen as the start point. Constraints from this mapping are inferred and consolidated and this happens in the update constraints phase. To schedule the DFG of each BB, a backward traversal list scheduling algorithm is used. Operation nodes are listed by a priority order based on mobility and fanout. Once the highest priority node is identified, the compiler finds a placement for it in the PEA model and schedules it. If a placement solution could not be found, then the DFG is transformed to improve the placement possibilities. Graph transformations increase the physical or temporal distance between source and sink to keep data dependencies or reduce the node

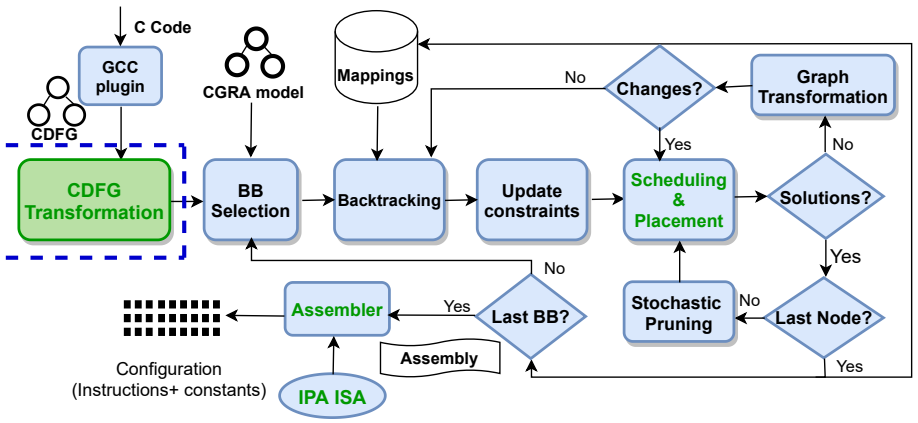

Fig. 3: Baseline Compilation Flow extended with CDFG Transformation

fan outs by operation splitting. The modified DFG then goes into the scheduling and placement phase. If the transformation does not make any change in the DFG, then that calls for a backtracking. Another map from the mapping list is chosen, constraints are updated and the process starts all over again. If the mapping is successful, a stochastic pruning is applied on the partial mapping set to prevent it from growing exponentially. Once the assembly file containing the map for all the BBs is prepared, it is fed to the assembler together with the ISA. Assembler then generates the configuration for the PE array.

In the proposed model, a CDFG transformation phase is introduced as the first step in the compilation flow. Since the transformation is done prior to mapping, it is termed pre-mapping CDFG transformation. Additionally, the priority order in which operations are listed is redefined so that loop activity node would be scheduled to fire as the first activity in the concerned BB. The IPA ISA is extended with two dedicated instructions and the assembler is also modified accordingly. The newly introduced step, CDFG transformation and the modified blocks like Scheduling & Placement, Assembler and IPA ISA are highlighted in Fig. 3. The pre-mapping CDFG transformation is detailed next.

We introduce two new terms, *Hardware Loop Header (HLH)* and *Hardware Loop Terminal (HLT)* to title the entry and exit BBs of the loop. HLH is the BB which dominates all other blocks which constitute the loop body and HLT is last block in that set. Algorithm 1 lists down the steps in transforming the input CDFG. The transformation is applied to loops of depth up to four as the hardware unit is so designed. Iteration count of the loop is computed from the initial and final values of loop control variable and the step by which the variable gets updated in each iteration of the loop. The loop step can be greater than or equal to one and the update can be increment or decrement. The algorithm identifies the first BB in the set of BBs that forms the loop body, which is essentially the true path successor of loop condition checking BB. This BB is designated as the HLH. The last BB in the loop body referred as loop latch in the baseline model is set as the HLT. A loop activity node and a data node holding the loop count value are inserted into the HLH. Address of the last instruction in HLT is the loop end address. As discussed earlier, loop characteristics such

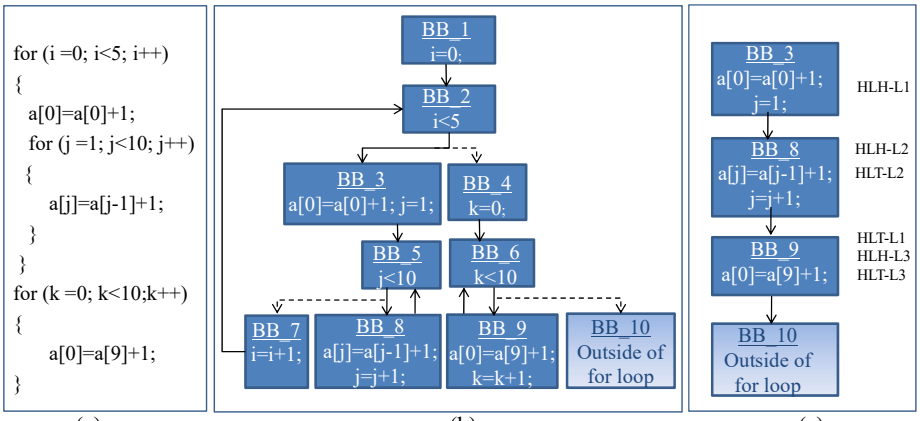

Fig. 4: (a) Sample program; (b) Corresponding CDFG; (c) Transformed CDFG

as start address, end address and loop count are used to configure the hardware loop and that explains the significance of HLH and HLT. As the next step, the BB that does the condition checking which was the loop header in the baseline model and the back jump from HLT to this BB are eliminated from the CDFG. This essentially does a cyclic to acyclic graph transformation on that part of the CDFG which represents the loop. If the loop control variable is not used as an operand in any of the BBs, then the nodes that correspond to its initialization and increment/decrement operations are removed. If the BB housing those nodes does not include any other node, then that BB as a whole will be pulled out of the CDFG. If the loop control variable is used inside the loop body, then those operation nodes will be preserved and the variable gets updated by the the step value, each time the loop iterates.

Once this transformation is done for each loop in the CDFG, the algorithm does another pass through the list of loops to merge the HLTs of outer and inner loops if possible. The merging is performed recursively down the levels as long as it is feasible to do so. By doing this merge, these loops will have the same end address which would trigger the hardware unit to update the loop count of all those loops together as the inner one finishes its last iteration. As the algorithm runs to completion transforming the CDFG, we are left with a very simple acyclic graph representing a significantly reduced set of instructions. Fig. 4 shows a sample program, the corresponding CDFG and the transformed version of it.

5 Experiments and Results

5.1 Experimental Setup

In this section, we analyse the implementation results providing performance and area comparison for a set of commonly used DSP applications. To ensure an unbiased evaluation and test the range of applicability, we have chosen a set of loop intensive kernels from various application domains with varying structure

Algorithm 1: Pre-Mapping CDFG Transformation

```
Result: CDFG'
loopLevel =0; lp=getLoop(CDFG, ++loopLevel); CDFG'=CDFG;
while lp not NULL do
    if lp.height > 4 then
        continue;
    else
        loopStartBB=lp.header.truepathSuccessor;
        loopCount=computeLoopCount(lp);
        loopNode=createLoopNodeIn(loopStartBB);
        loopNode.sourceDataNode=createDataNode(loopCount);
        lp.HLH=loopStartBB; lp.HLT=lp.latch;
        lp.HLT.successor=lp.header.falsepathSuccessor;
        lp.header.falsepathSuccessor.predecessor=lp.HLT;
        CDFG'=removeBB(lp.header, CDFG');
        if isUsedElsewhere(lp.controlVar, CDFG) then
            continue;
        else
            loopInitBB=loopStartBB.predecessor;
            if initOnlyBB(loopInitBB, lp.controlVar) then
                loopInitBB.predecessors.successor=loopStartBB;
                loopStartBB.predecessors=loopInitBB.predecessors;
                CDFG'=removeBB(loopInitBB, CDFG');
            else
                removeConstFlag=isConstDataNodeUsedElsewhere (loopInitBB,
                  lp.controlVar);
                removeInitNodes(loopInitBB, lp.controlVar, removeConstFlag);
            end
            removeConstFlag=isConstDataNodeUsedElsewhere(lp.HLT, lp.controlVar);
            removeUpdateNodes(lp.HLT, lp.controlVar, removeConstFlag);
        end
    end
    lp=getLoop(CDFG', ++loopLevel);
end
loopLevel=0; lp=getLoop(CDFG', ++loopLevel);
lpNext=getLoop(CDFG', loopLevel+1);
while lp not NULL && lpNext not NULL do
    if (lp.height < 4) && isEmpty(lp.HLT) && (lp.HLT.predecessorList.size==1) &&
      (lp.HLT.predecessor==lpNext.HLT) then
        lpNext.HLT.successor=lp.HLT.successor;
        CDFG'=removeBB(lp.HLT, CDFG'); lp.HLT=lpNext.HLT;
    end
    lp=getLoop(CDFG', ++loopLevel); lpNext=getLoop(CDFG', loopLevel+1);
end
```

of loop nesting. Finite Impulse Response (FIR) filter is included to represent the family of digital filters. By virtue of the four-level nesting of loops, the image processing kernels, 2D convolution, erosion and dilation will aid in a corner case testing of the proposed approach. The kernels, Jacobi-1D and Seidel-2D performing stencil computations are used in several different application domains including data mining and machine learning. Similar is the case with Floyd Warshal kernel which is used for solving all pair shortest path problems. Jacobi-1D performs computations with 3-point stencil pattern over one dimensional data whereas Seidel-2D is a computation over 2 dimensional data with 9-point stencil pattern providing a deeper loop to test with. The generic kernels, matrix multiplication and addition are also included. Altogether the set has a good mix of perfectly and imperfectly nested loops. A detailed characteristics of these kernels is listed in Table 1.

Performance of the hardware loop implementation is compared against that of the baseline IPA architecture that employs only software based loop opti-

Table 1: Maximum depth and iteration counts of loops for the listed kernels

Kernel	Depth of Nesting	Max. # of Loop Iterations
Floyd Warshal	3	60×60×60 = 216,000
Jacobi-1D	2	20×(28+28) = 1,120
Seidel-2D	3	20×38×38 = 28,880
FIR filter	2	190×10 = 1,900
2D Convolution	4	80×60×3×3 = 43,200
Erosion	4	78×58×3×3 = 40,716
Dilation	4	78×58×3×3 = 40,716
Matrix Multiplication	3	32×32×32 = 32,768
Matrix Addition	2	32×32 = 1,024

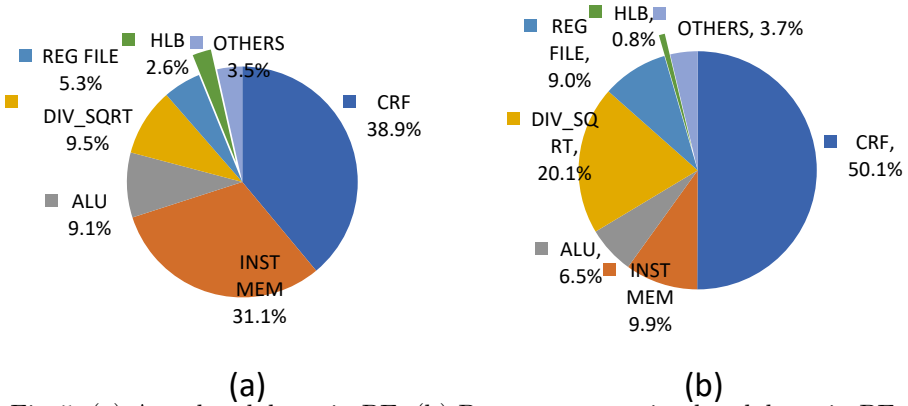

Fig. 5: (a) Area breakdown in PE; (b) Power consumption breakdown in PE

mization. The IPA achieves an energy efficiency improvement up to 18×, with an average of 9.23× (as well as a speed-up up to 20.3×, with an average of 9.7×) [5] compared to a RISC-V core [7] specialized for ultra-low power near-sensor processing.

5.2 Implementation results

This section describes the implementation results for the baseline IPA architecture and the proposed design. Both the designs were synthesized with Cadence Genus Synthesis Solution 17.22-s017_1 using 90nm CMOS technology libraries. The IPA implementation on which the experiments are conducted is configured to have a 4×2 PE array with each PE housing a 21×64-bits IRF, a 20×32-bits CRF and a 32×8-bits RRF. On top of this baseline IPA, Hardware Loop Block (HLB) module was integrated to support the proposed optimization. Fig. 5 (a) and (b) reports the area and power consumption overhead of the proposed HLB of 2.6% and 0.8% respectively over the baseline architecture.

Table 2: Comparison of the total number of instructions executed between the baseline and the proposed models

Kernel	**Baseline Model**	**Proposed Model**	**Gain Achieved**
Floyd Warshal	3,514,585	2,204,373	1.59×
Jacobi-1D	36,525	13,895	2.63×
Seidel-2D	1,627,047	827,389	1.97×
FIR filter	25,858	11,795	2.19×
2D Convolution	1,066,898	1,009,438	1.06×
Erosion	67,379	48,722	1.38×
Dilation	92,379	37,154	2.49×
Matrix Multiplication	410,130	176,399	2.33×
Matrix Addition	12,818	5,391	2.38×
Average			2.00×

5.3 Performance analysis

Table 2 presents the comparison of executed instructions in the software based baseline and the proposed hardware based optimization. It is observed that the number of instructions executed is reduced to half for almost all the kernels; achieving an average gain of 2×. The obtained results clearly emphasize that by eliminating loop control instructions, the total number of instructions executed can be greatly reduced. The same is illustrated in Fig. 6 which gives a comparison of the number of instructions executed by accelerating a simple loop on the baseline and proposed models for a 2×1 CGRA. The loop control instructions that would be eliminated by the proposed technique are highlighted in red (Fig. 6(b)). Since the loop variable i is used as an operand in the loop body, its initialization and update operations are preserved. By eliminating the highlighted instructions, total number of instructions executed is reduced from 46 to 18 achieving a gain of 2.56× (Fig. 6(e)). It is interesting to note that the gain on reducing the number of branches is 12× ((Fig. 6(d)) which is multiple times of what obtained for the total number of instructions. At any given instant, every PE in the baseline PE array will be executing instructions from the same basic block (BB). To achieve this block-level synchronicity, branch instructions (JMP/CJMP) that govern the control flow between BBs are copied to every PE. Therefore the effect of eliminating one branch instruction from the code will be reflected in the execution profile by many folds. On top of that, more than 50% of the loop control instructions that get eliminated would be branch instructions as can be seen from Fig. 6(b). This explains the huge difference on the gain obtained for branches and total number of instructions. Speed-up or the performance gain achieved by the enhancement over the baseline model is computed as the ratio of execution cycles. As depicted in Table 3, an average gain of 1.5× is recorded for the set of kernels we considered.

Table 4 gives the comparison of the number of branches executed. As expected, the proposed solution outperforms the baseline by a huge margin. An average gain of 30× and a maximum of 78× is recorded. To have the instruction execution on PEs synchronized to the BB boarders, each PE in the PEA would

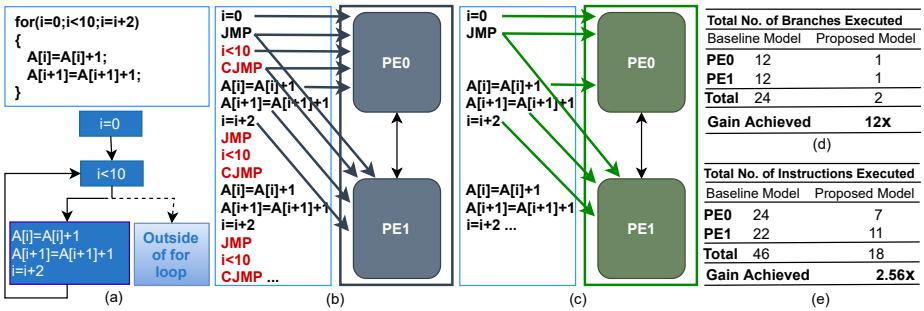

Fig. 6: (a) Partially unrolled kernel and corresponding CDFG; (b) Instructions executed on the baseline model, and (c) proposed model for a 2×1 CGRA; (d) Total number of branches executed on baseline vs proposed models; (e) Total number of instructions executed on baseline vs proposed models

Table 3: Comparison of latency in terms of execution cycles between the baseline and the proposed models

Kernel	**Baseline Model**	**Proposed Model**	**Gain Achieved**
Floyd Warshal	2,203,210	1,548,434	1.42×
Jacobi-1D	15,197	11,268	1.35×
Seidel-2D	670,945	582,652	1.97×
FIR filter	15,327	9,358	1.64×
2D Convolution	538,623	399,280	1.35×
Erosion	41,726	28,059	1.49×
Dilation	41,750	28,013	1.49×
Matrix Multiplication	271,930	140,713	1.93×
Matrix Addition	9,777	6,569	1.49×
Average			1.50×

be executing the branch instruction simultaneously. Therefore, eliminating one branch instruction will reduce the number of cycles by one but the reduction on the number of branches will be by the number of PEs. This is the reason why we do not see a direct correlation between the reduction in branches and execution cycles. The total number of basic blocks is also found to be reduced (Table 5) in the proposed solution for all the kernels, effect of which is already seen in the number of branches. It is by the pre-mapping CDFG transformation that the BBs and the branch operations got eliminated. Therefore, the gain in terms of number of BBs and branches acts as an efficiency measure of the CDFG transformation as well. In the next phase, we will explore energy efficiency of the proposed approach.

6 Conclusion

The performance bottleneck imposed by the overhead of having loop control instructions in kernel execution is often substantial. It is impossible to get rid of all the control bottlenecks with software only loop optimizations. In this paper,

Table 4: Comparison of the number of branches executed between the baseline and the proposed models

Kernel	**Baseline Model**	**Proposed Model**	**Gain Achieved**
Floyd Warshal	1,325,288	446,770	2.97×
Jacobi-1D	11,815	164	72.04×
Seidel-2D	487,064	6,248	77.96×
FIR filter	9,126	764	11.95×
2D Convolution	432,646	274,084	1.58×
Erosion	41,661	14,200	2.93×
Dilation	41,661	9,370	4.45×
Matrix Multiplication	139,526	4,164	33.51×
Matrix Addition	4,358	69	63.16×
Average			30.10×

Table 5: Comparison of the number of basic blocks mapped between the baseline and the proposed compilation flow

Kernel	**Baseline Model**	**Proposed Model**	**Gain Achieved**
Floyd Warshal	16	13	1.23×
Jacobi-1D	14	10	1.40×
Seidel-2D	14	10	1.40×
FIR filter	11	8	1.34×
2D Convolution	24	20	1.20×
Erosion	22	18	1.22×
Dilation	24	20	1.20×
Matrix Multiplication	19	14	1.56×
Matrix Addition	11	7	1.57×
Average			1.40×

we introduced a hardware based loop control mechanism for CGRAs targeted to reduce the number of non-contributing control operations. By moving the loop control to hardware, the proposed model could achieve an average speed-up of 1.5× and reduce the number instructions executed to half, against the baseline CGRA architecture running software based loops. Up to 78× and an average of 30× gain is observed on reducing the number of executed branch instructions with an area and power consumption overhead of 2.6% and 0.8% respectively. The significant improvement noted on a set of kernels with varying loop structure suggests that the discussed design can be presented as a generic loop optimization technique to a wide range of applications.

References

1. Bajwa, R.S., Hiraki, M., Kojima, H., Gorny, D.J., Nitta, K.i., Shridhar, A., Seki, K., Sasaki, K.: Instruction buffering to reduce power in processors for signal processing.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems **5**(4), 417–424 (1997)
2. Balasubramanian, M., Dave, S., Shrivastava, A., Jeyapaul, R.: Laser: A hardware/software approach to accelerate complicated loops on cgras. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1069–1074. IEEE (2018)
3. Das, S., Martin, K.J., Coussy, P., Rossi, D.: A heterogeneous cluster with reconfigurable accelerator for energy efficient near-sensor data analytics. In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1–5. IEEE (2018)
4. Das, S., Martin, K.J., Coussy, P., Rossi, D., Benini, L.: Efficient mapping of cdfg onto coarse-grained reconfigurable array architectures. In: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 127–132. IEEE (2017)
5. Das, S., Martin, K.J., Rossi, D., Coussy, P., Benini, L.: An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**(6), 1095–1108 (2018)
6. Dragomir, O.S., Bertels, K.: Extending loop unrolling and shifting for reconfigurable architectures. Architectures and Compilers for Embedded Systems (ACES) pp. 61–64 (2010)
7. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10), 2700–2713 (2017)
8. Hamzeh, M., Shrivastava, A., Vrudhula, S.: Epimap: Using epimorphism to map applications on cgras. In: Proceedings of the 49th Annual Design Automation Conference. pp. 1284–1291 (2012)
9. Kavvadias, N., Nikolaidis, S.: Elimination of overhead operations in complex loop structures for embedded microprocessors. IEEE Transactions on Computers **57**(2), 200–214 (2008)
10. Liu, D., Yin, S., Liu, L., Wei, S.: Polyhedral model based mapping optimization of loop nests for cgras. In: Proceedings of the 50th Annual Design Automation Conference. pp. 1–8 (2013)
11. Masuyama, K., Fujita, Y., Okuhara, H., Amano, H.: A 297mops/0.4 mw ultra low power coarse-grained reconfigurable accelerator cma-sotb-2. In: 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 1–6. IEEE (2015)
12. Mathew, B., Davis, A.: A loop accelerator for low power embedded vliw processors. In: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. pp. 6–11 (2004)
13. Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H., Kim, H.s.: Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 166–176 (2008)
14. Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., Olukotun, K.: Plasticine: A reconfigurable architecture for parallel patterns. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). pp. 389–402. IEEE (2017)
15. Tsao, Y.L., Chen, W.H., Cheng, W.S., Lin, M.C., Jou, S.J.: Hardware nested looping of parameterized and embedded dsp core. In: IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings. pp. 49–52. IEEE (2003)
16. Vadivel, K., Wijtvliet, M., Jordans, R., Corporaal, H.: Loop overhead reduction techniques for coarse grained reconfigurable architectures. In: 2017 Euromicro Conference on Digital System Design (DSD). pp. 14–21. IEEE (2017)