# Energy Efficient Hardware Loop Based Optimization for CGRAs

Chilankamol Sunny, Satyajit Das, Kevin J M Martin, Philippe Coussy

**Journal of Signal Processing Systems**
https://doi.org/10.1007/s11265-022-01760-9

# Energy Efficient Hardware Loop Based Optimization for CGRAs

**Chilankamol Sunny · Satyajit Das · Kevin J. M. Martin · Philippe Coussy**

Received: 3 October 2021 / Revised: 21 January 2022 / Accepted: 6 April 2022



**Abstract** Research interest and industry investment in edge computing solutions have increased dramatically in recent years. Consequent quest for bal-

C. Sunny
IIT Palakkad, Palakkad, Kerala, India
E-mail: 112004004@smail.iitpkd.ac.in

S. Das
IIT Palakkad, Palakkad, Kerala, India
E-mail: satyajitdas@iitpkd.ac.in

K. J. M. Martin
Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France
E-mail: kevin.martin@univ-ubs.fr

P. Coussy
Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France
E-mail: philippe.coussy@univ-ubs.fr

anced performance, energy efficiency and flexibility bestowed surging popularity on Coarse Grained Reconfigurable Array (CGRA) architectures. To further improve the performance and energy efficiency, several hardware and software-based loop optimizations are adopted for CGRAs. In this paper, we propose a centralized hardware-based loop optimization technique to achieve better area and energy results compared to the previously implemented distributed version. Without incurring any performance degradation, area overhead against the reference architecture is reduced down to 1.5% for a 4×2 CGRA configuration. A maximum of 47.3% and an arithmetic mean of 27.2% reduction in energy consumption is attained by the centralized version of hardware loop compared to the baseline model employing software loop. Furthermore, the paper explores the co-existence of CGRA-specific hardware and software optimizations and their impact on loop efficiencies. Enhanced results are obtained by coupling loop unrolling with centralized hardware loop support. The combination allows achieving up to 68.7% reduction in energy consumption and 5.46× speed-up against the baseline model with no optimizations applied.

**Keywords** Coarse grained reconfigurable array (CGRA) · Loop optimization · Hardware loop · Loop unrolling

## 1 Introduction

The wave of widespread specialization is observed in industry from a long time earlier. This is on account of the momentous improvement in energy efficiency and performance the specialized accelerators could bring in. Varying nature and growing size of applications and rapidly evolving software force flexibility an equally relevant criteria for the acceptability and sustainability of computing fabrics [16]. Nevertheless, achieving satisfactory balance between performance, energy efficiency and flexibility is a hard-to-meet challenge. Application-specific integrated circuits (ASICs) with highest performance and energy efficiency results are the least flexible architectures, whereas highly flexible general-purpose processors (GPPs) and graphics processing units (GPUs) report extremely low energy efficiency [14, 19]. The field-programmable gate arrays (FPGAs) as well as the flexible digital signal processors (DSPs) are less energy-efficient than ASICs. Further, it is much more challenging to program FPGAs compared to CPUs. Coarse-grained reconfigurable array (CGRA) architectures possess near-ASIC energy efficiency and performance and software-like programmability [14, 29], and gain increasing attention from industry and academia. To cope with the growing demand for ultra-low power computing imposed by edge computing paradigms like internet of things (IoT) and cyber physical systems (CPS), studies on CGRAs are now headed towards further improving the energy efficiency.

Most of the applications that run on CGRA spend a major portion of the computation time and energy on loops. This narrows down the scope of improving overall energy efficiency to optimizing the loop execution. Numerous loop transformation techniques that convert loops into semantic equivalents of

lesser computation complexity or exhibiting better parallelism have been designed and deployed in commercial compilers to optimize the loop execution. Majority of these techniques target to optimize the innermost loop execution on CGRAs entrusting outer loops to the host processor. By devising such software-based loop optimizations [8,11,13,18], CGRAs could achieve significant improvement in performance and energy efficiency. However, synchronization overhead with the host processor for the execution of outer loops became a bottleneck leaving room for further improvement. CGRA implementations with processing elements (PEs) supporting globally synchronized branch instructions to execute nested loops and conditionals [6] gained popularity as they could handle the outer loops as well. This approach minimizes the synchronization overhead with the host yet necessitated the execution of extra loop control instructions. Case studies on processor architectures [1,12,15,25] reveal that improved performance and energy efficiency can be attained when loop-specific hardware optimizations are applied. Few recent CGRA architectures have come up with such architectural modifications to better support loop execution and reported good results [2,22,24,26].

In this paper, we introduce a centralized hardware-based loop optimization for CGRAs where a central PE synchronizes the loop execution between all the PEs with minimal communication overhead. State-of-the-art Integrated Programmable Array (IPA) architecture [5] is chosen as the reference CGRA to work on. In our previous work [22], we extended the IPA model with a hardware-based loop control mechanism to eliminate the non-contributing loop control instructions from the execution stream. With this enhancement, IPA renders hardware support to arbitrarily nested loops. Features like synchronous termination of loops at multiple levels, support for any number of sibling loops (loops at the same level of nesting) and on-the-fly loop configuration distinguish the model from other hardware solutions like ZOLA [26]. The proposed centralized design does not impose any alteration in the compilation flow set for the distributed version in which each PE houses a hardware loop unit. Instead it offers the flexibility to work with the same configuration (instructions and constants for each PE) generated for the basic implementation. Without incurring any performance degradation, area overhead against the baseline IPA architecture is reduced down to 1.5% in the centralized version. Compared to the baseline, the proposed model achieves a maximum of 47.3% and an arithmetic mean of 27.2% reduction in energy consumption when the basic distributed hardware loop model marks 44.5% and 23.3% respectively.

As the second contribution, we explore the impact of combining hardware and software-based loop optimizations on CGRAs. IPA supports loop unrolling as an optimization technique to reduce the overhead and better expose instruction-level parallelism in innermost loop execution. This paper investigates the combined effect of hardware loop with loop unrolling. By coupling loop unrolling with centralized hardware loop implementation, up to 68.7% reduction in energy consumption is attained in comparison to the baseline IPA model with no optimizations applied.

The rest of the paper is organized as follows. Section 2 presents related works on hardware and software-based loop optimizations. Section 3 describes the baseline CGRA architecture and the hardware-based looping technique we introduced in [22]. Section 4 introduces the centralized approach and details the implementation. An overview of the compilation flow is given in section 5. In section 6, performance, area and energy results for both distributed and centralized versions of the hardware loop are discussed. Section 7 concludes the paper.

## 2 Related Work

A larger portion of the works that optimize loop execution on CGRAs are focused on the placement and routing of the loop bodies. Another popular division is the transformation of loops prior to placing. In the context of CGRAs, loop unrolling [8,26] is the most widely used transformation technique. The unrolled loop kernel is mapped onto the CGRA leveraging its massively parallel architectural features. Other optimization techniques are modulo scheduling [11,18] that facilitates overlapped execution of loop iterations and loop affine transformation [13] that optimizes the PE utilization rate and the cost of communication between PE array and controller. A vast majority of the designs based on the above approaches pivot on optimizing only the innermost loop neglecting the communication overhead of outer loops. Authors of [13] formulated loop mapping as a polyhedral based nonlinear optimization problem and introduced affine transformation on the loop body to map up to two innermost levels of the loop nesting. It is a promising still a minimal effort in optimizing the mapping of deep nested loop structures. Purely software-based solutions like the discussed techniques often prove to be insufficient to meet the growing demand for energy efficient execution imposed by low power budget applications.

Studies show that, since their early beginnings, processor architectures have availed hardware support for repeated execution of instructions. Some examples are loop buffers, *loop* and *rep* instructions of x86 processors and the zero overhead loop accelerators on DSP [25], RISC [12] and VLIW architectures [15]. The parallel ultra-low power (PULP) cluster architecture [9] is a multi-core platform with hardware support for loops. It could support two level nesting of loops by employing two sets of registers to hold the loop configuration data. In stream processors, many optimizations like loop distribution, loop fusion and re-ordering were introduced to improve the locality of stream register file (SRF) [23]. Primary objective of many of such hardware-software hybrid solutions is to enhance reusing of data streams across different iterations of loop so as to minimize the memory access time [27]. Coming to CGRAs, hardware support was not a necessity until scope of acceleration spanned beyond the innermost loop. Hence, CGRA solutions that wield architectural features to support loop execution are very few in number. The approach presented in [26] reports good results, thanks to the extension of the

reference CGRA architecture with zero-overhead loop accelerator (ZOLA), single cycle loop support and loop buffers. An average energy reduction of 18.3% was achieved by the design combining these three hardware techniques. ZOLA alone could save energy up to 25% for applications for which it could replace the functional unit (FU) required to calculate the loop index. For those applications where the FU is used for other computations, it cannot be removed and hence ZOLA is an advisable choice only when its energy overhead is lower than what it saves through the reduction of register accesses and control flow computations. Usage of loop buffers could reduce the energy consumption as it eliminated significant number of instruction memory accesses. However, buffer sizes should be carefully chosen profiling the application, otherwise having them only aggravates the overhead.

Another notable work with architectural support for optimizing loop execution is the ultra-elastic CGRAs (UE-CGRAs) [24] that can efficiently execute loops with irregular control flow and memory accesses and inter-iteration loop dependencies. The solution co-designed across compiler, architecture, and VLSI accelerates true-dependency bottlenecks and reduces energy consumption by supporting fine-grain dynamic voltage and frequency scaling (DVFS) on individual PEs. It could achieve reasonable improvement in performance and energy efficiency compared to traditional inelastic and elastic CGRAs on executing irregular loops. LASER [2] is yet another hardware solution taking compiler support to accelerate nested loops and loops with nested conditionals. Compiler transforms nested loops into single-level loops with conditionals and the hardware fetches and executes the instructions from the right path at runtime. Both LASER and UE-CGRAs are specialized solutions focusing on loops that would suffer from performance degradation on conventional CGRAs. In both of these approaches, basic implementation of loops remains to be software-based and hence there exists the discussed issue of control flow bottleneck. On the other hand, the technique we introduced in [22] is a generic loop optimization solution that eliminates loop control instructions by implementing the looping mechanism entirely in hardware. However, each PE in the PE array houses a dedicated unit to manage the loop control flow.

In this paper, we propose a centralized implementation of the model to minimize the area overhead imposed by this design. None of the mentioned works discuss the impact of applying the techniques they propose in combination with existing software-based optimizations like loop unrolling or modulo-scheduling. This paper explores the co-existence of hardware and software-based optimizations by applying loop unrolled kernels on the hardware loop model.

## 3 Background and Motivation

This section gives an introduction to the baseline CGRA architecture and the hardware-based looping technique we introduced in [22]. It also details the motivation behind implementing the hardware loop in centralized fashion as

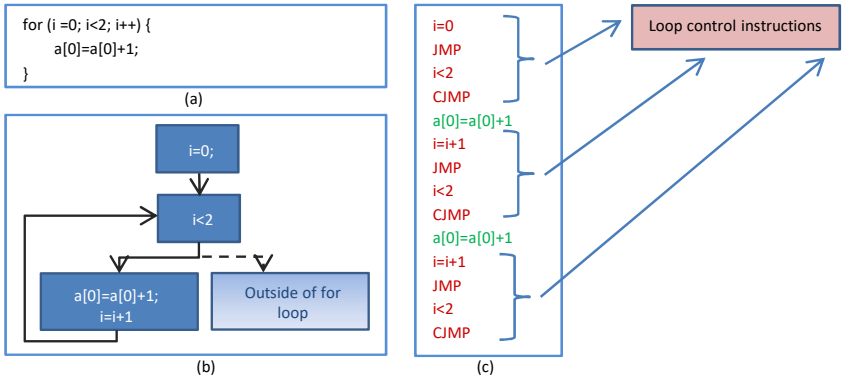


**Fig. 1** (a) Sample program; (b) Corresponding CDFG; (c) Representative assembly code

well as the need for coupling hardware and software-based optimizations. In this discussion, the first and last instructions in the loop body are referred as start and end of loop respectively. The number of times the loop is to be iterated is termed as loop count.

In CGRAs, loop control flow is often implemented by supporting repeated branching to the loop start. Additional instructions are injected into the kernel code to keep track of the iterations and perform the branching. Using a simple example, Figure 1 demonstrates the overhead imposed by this software-based loop implementation.

**Listing 1** Matrix Multiplication Pseudo-code

```
for(int i = 0; i < r1; ++i){
    for(int j = 0; j < c2; ++j){
        for(int k = 0; k < c1; ++k)
            mult[i][j] += first[i][k] * second[k][j];
        }
    }
}
```

The burden is higher in the case of nested loops. For example, consider the matrix multiplication pseudo-code shown in Listing 1 with a nested loop of depth three. It is assumed that the product matrix is initialized with zero. Total number of loop control instructions executed, $N_{lc}$ and total number of kernel instructions executed, $N_{ke}$ can be computed as shown in (1) and (2).

$$N_{lc} = q * (r_1 + (r_1 * c_2) + (r_1 * c_2 * c_1)) \tag{1}$$

$$N_{ke} = p * (r_1 * c_2 * c_1) \tag{2}$$

where $q$ is the number of loop control instructions executed per iteration and $p$ is the number of instructions that constitute the loop body.

**Table 1** Comparison between different architectural approaches

| | **CMA** [17] **(8×10)** | **ZOLA** [26] | **LASER** [2] | **Plasticine** [20] **(16×8)** | **Plasticine with SARA [28]** | **SNAFU** [10] **CGRA (6×6)** | **IPA** [5] **(4×2)** |
|---|---|---|---|---|---|---|---|
| **Memory ops** | CPU | CGRA | CGRA | CGRA | CGRA | CGRA | CGRA |
| **Innermost loop** | CGRA | CGRA | CGRA | CGRA | CGRA | CGRA | CGRA |
| **Outer loop** | CPU | CGRA | CGRA | CGRA | CGRA | CPU | CGRA |
| **Offload+Sync** | CPU | CPU | CPU | CGRA | CGRA | CPU | CPU |
| **Maps** | DFG | NA | DDG | CDFG | CDFG | DFG | CDFG |
| **Source** | Customized | ZOLA Assembly | ANSI C | Scala-based | Scala-based | Vector Assembly | ANSI C |
| **Power** | 11mW | NA | NA | 22W | NA | $< 1mW$ | 5mW |
| **Main performance claim** | Energy efficiency of 243 MOPS/mW | 18.3% energy gain over ref. CGRA | 46% energy gain over CGRA with predicate network | 76.9× energy efficiency over FPGA | 1.9× speed-up over GPU | 9.9× speed-up over CPU | 20×speed-up over CPU |

The number of additional instructions executed per iteration of the loop ($q$) is more or less fixed for a particular CGRA implementation but the impact is more on loops containing lesser number of kernel instructions. Therefore the overhead of loop execution is proportional to the ratio of $N_{lc}$ to $N_{ke}$. Minimizing $N_{lc}$, the total number of loop control instructions will result in considerable improvement in performance and energy efficiency. Loop unrolling is one of the techniques that CGRA architectures employ to reduce this overhead. By unrolling a loop, number of times the loop gets executed is reduced by the unrolling factor whereas the number of kernel instructions in that loop is increased by the same factor. In the above example, if the innermost loop is unrolled by a factor of 4, $N_{lc}$ is reduced while $N_{ke}$ remains the same as can be seen in (3) and (4).

number of instructions that constitute the innermost loop, $p' = 4 * p$;
loop count of innermost loop, $c_1' = c_1/4$

$$N'_{lc} = q * (r_1 + (r_1 * c_2) + (r_1 * c_2 * c_1/4)) \tag{3}$$

$$N'_{ke} = 4 * p * (r_1 * c_2 * c_1/4) = N_{ke} \tag{4}$$

Although enhanced performance is attained by applying software solutions like loop unrolling, modulo scheduling and affine transformation, the scope of optimization is limited to innermost loop only. This is not sufficient to get high energy efficiency for kernels with complex nesting of loops. In view of this, we introduced a hardware-based loop optimization technique [22] where the looping mechanism is entirely implemented in hardware. Focus was on minimizing or completely eliminating the parameter $q$, the number of loop control instructions executed per iteration.

State-of-the-art Integrated Programmable Array (IPA) architecture [5] is extended to implement the hardware loop design. A comparative study of different CGRA architectures including the recent works is given in Table 1. Cool Mega-Array (CMA) [17], LASER [2], ZOLA [26], Plasticine [20] with vanilla compiler, Plasticine with SARA [28] compiler and the CGRA generated by SNAFU framework [10] are the discussed approaches. Some information could not be extracted from a few papers for which we have entered "NA" in the corresponding cell. It is evident from the table that IPA gives the best trade-off between performance, energy efficiency and ease of programming. It also provides support for conditionals and nested loop execution and hence chosen as the baseline architecture.

Figure 2 represents the SoC design in which the IPA accelerator is loosely coupled with a host CPU. IPA and CPU are connected to a shared multi-banked tightly coupled data memory (TCDM) through a low-latency logarithmic interconnect. It is through TCDM that data are fed and results are taken to and from the IPA. Global configuration memory (GCM) stores the instructions and constant values that are to be sent to each PE. An instruction cache is provided for the host CPU for better performance. IPA is an interconnected array of processing elements (PEs). The figure shows a 4×2 PE array configuration of IPA, augmented with hardware loop support. Architectural details of individual PEs is also given. A parametric number of PEs are configured to include Load-Store Units (LSUs) as well with which the TCDM is interfaced. Further details of data communication and compiler support for the same are featured in [4] and [21]. Each PE is incorporated with a hardware loop block (HLB) in its Controller unit to optimize the loop execution.

Figure 2(b) gives a simplified view of HLB with which the loop control flow is entirely stitched into hardware. This unit is responsible for updating the PC. Inside the loop, control flows sequentially by incrementing PC by one. As the loop completes one iteration, PC points to the loop start. This is repeated for as many times as specified by the loop count. To facilitate this functionality, HLB keeps track of loop configuration parameters like loop count and address of the first and last instructions in the loop body. To check if an iteration is over, PC value is compared against loop end address in each cycle. If they are equal, loop count is decremented by one and PC is loaded with loop start address as long as the new loop count is greater than zero. In every other case, PC is incremented by one.

The dedicated loop unit, HLB is integrated into every PE in the PE array. Each PE maintains a copy of loop configuration parameters for each level of loop nesting and processes the loops independently. This is not an ideal solution in terms of area and energy consumption. In this paper, we present a centralized implementation of the discussed model to overcome this overhead. The block-oriented, globally synchronized execution model of IPA ensures that each PE reaches the start and end of loop synchronously. This feature is exploited to bring on a central control on loop execution for the entire CGRA.

## 4 Proposed Architecture

We propose a conductor-performer model (Figure 3(a)) in which one of the PEs in the entire array is designated as the conductor-PE, controlling the loop execution in every other PE. Figure 4 gives the architecture of HLB at conductor-PE, configured to support up to four levels of loop nesting. The design includes a set of configuration registers (one for each level of loop nesting), a register (*loopLevel*) pointing to the current loop and the necessary logic to implement the functionality. Each configuration register stores the loop count and start and end instruction addresses. Corresponding registers are populated on encountering each loop in the kernel code. At any point of time, only

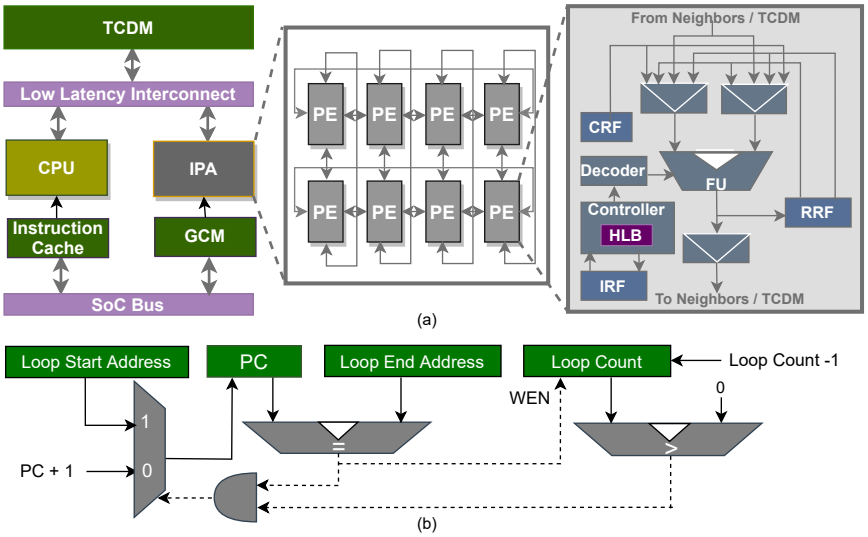


**Fig. 2** (a) IPA SoC; (b) Simplified HLB - FU: Functional Unit; CRF: Constant Register File; IRF: Instruction Register File; RRF: Regular Register File

one loop will be in execution from each level of nesting. Therefore loops at the same level can reuse the configuration register facilitating the design to support any number of sibling loops.

At each clock cycle, conductor-PE compares its PC to the loop end address from the register chosen by *loopLevel*. Based on the outcome and loop count value, decision on how to update the PC is taken as discussed in section 3. This decision is communicated to the performer-PEs through a three-bit wire carrying two signals. The signals are one-bit *loopStartFlag* and two-bit *loopLevel*. If PC is to be loaded with loop start address, *loopStartFlag* will be set to one and zero otherwise. The Loop Select Control Unit (LSCU) updates *loopLevel* to specify which loop is in action. Performer-PEs listen to the conductor signals and update their PCs as directed.

HLB architecture for performer-PEs is given in Figure 3(b). Even though all PEs hit the loop borders synchronously over time, the instruction addresses may vary from PE to PE as can be seen in Figure 5(a). Therefore even the performer-PEs should maintain their own copy of PC and register the loop start address. Furthermore, such a design helps to limit the width of conductor-performer communication channel. Rather than an instruction address, a one-bit flag and a two-bit loop selector suffice to govern the loop control on performers. In each clock cycle, performer-PEs check the *loopStartFlag*. If it is one, PC is updated with the loop start address stored in the register chosen by *loopLevel*. If the flag value is zero, the performers proceed to the very next instruction in their instruction register file (IRF) by incrementing PC by one.

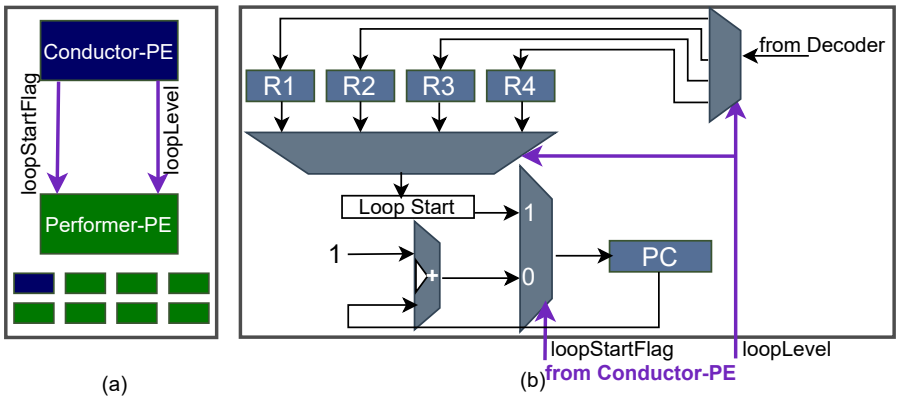


**Fig. 3** (a) Conductor-Performer Model; (b) Hardware Loop Block in the Performer-PEs

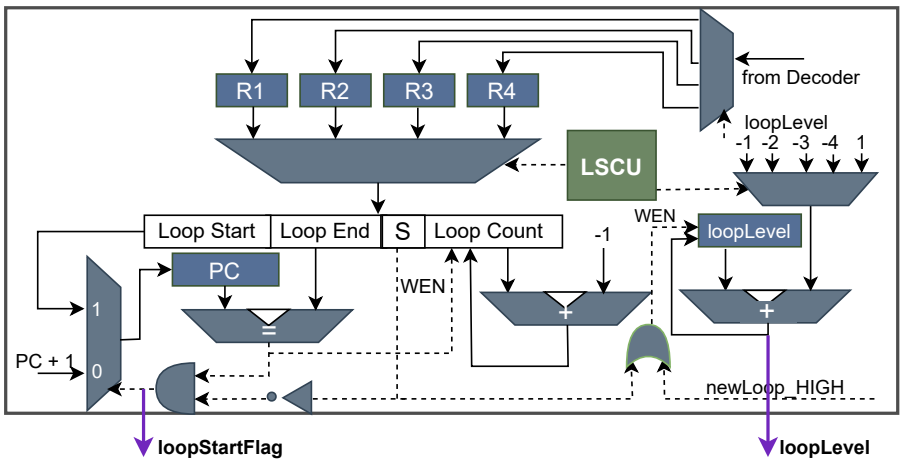


**Fig. 4** Hardware Loop Block in the Conductor-PE

Figure 5(b) and (c) together illustrate the loop control flow in conductor-performer model. A representative snapshot of conductor-PE IRF and the log depicting how *loopStartFlag*, *loopLevel* and PC are getting updated in each cycle are given. PC shown in the log is that of the conductor-PE. Two dedicated instructions, *LOOP_INIT* and *LOOP_CNT* are used to initialize the hardware loop. Loop start address is computed as two plus the *LOOP_INIT* instruction address. Loop end address and loop count appear as operands in *LOOP_INIT* and *LOOP_CNT* instructions respectively. These values are fed to the corresponding registers to configure the loop. Loop count field in the configuration register is initialized with one less than the loop count value to facilitate computing branch outcome by sign bit comparison.

Instructions executed in cycle one and two initiate the outermost loop. *loopLevel* becomes one, and the level one loop configuration register R1 is

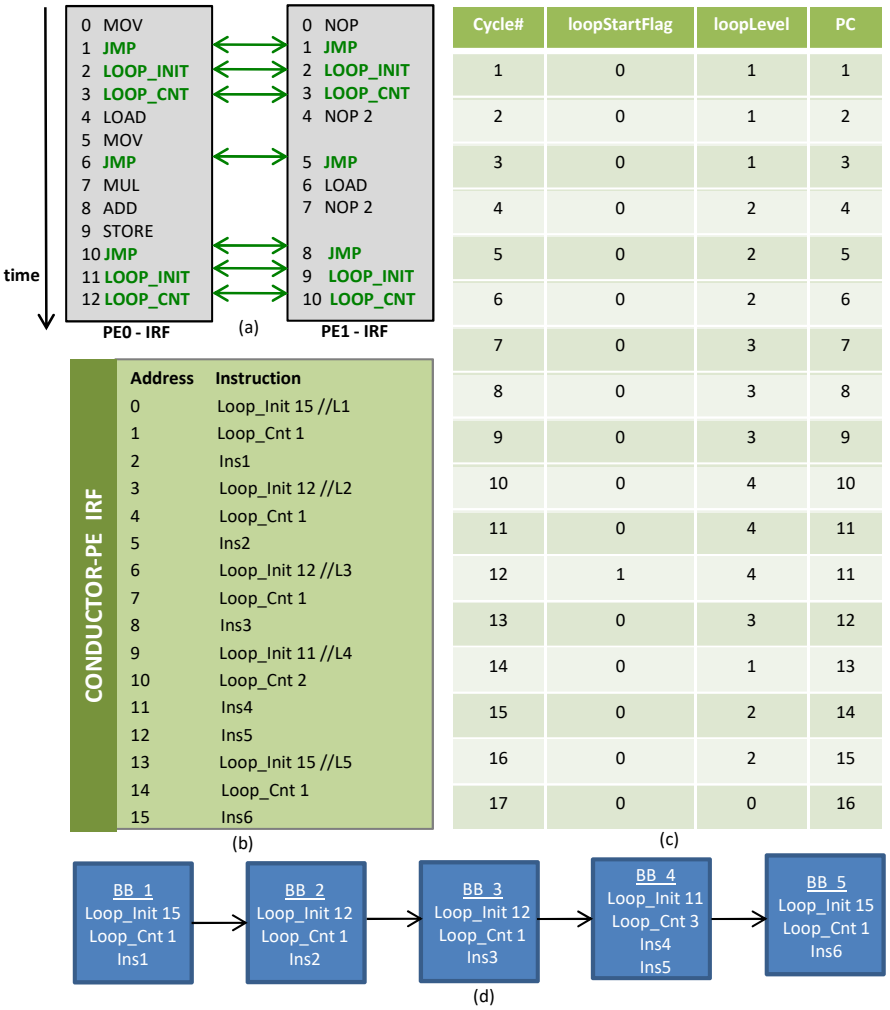


**Fig. 5** (a) Snapshot of PE IRFs holding temporally synchronized instructions; (b) Conductor-PE IRF with a sample pseudo-code; (c) Log of register updates according to the pseudo-code in (b)

updated with the loop parameters. Conductor-PE stores all the three parameters where the performer-PEs register the loop start address only. In cycle three, conductor-PE compares its PC against the loop end address stored in R1 which is found to be different. Hence *loopStartFlag* is set to zero and PC is incremented by one in conductor and performer-PEs. *LOOP_INIT* and *LOOP_CNT* instructions executed in cycle four and five together mark a new loop and this triggers incrementing *loopLevel* by one making its value to two. Loop parameters are loaded onto the second level register R2 in both conductor and performer-PEs. PC is also incremented by one since *loopStartFlag* remains at zero. In cycle six, PC is compared against the end address of L2.

Since they are different, *loopStartFlag* is kept at zero. Consequently, PC is incremented by one in all the PEs.

L3, the loop at level three is initialized by the instructions executed in cycle seven and eight. *loopStartFlag* is still zero and *loopLevel* and PC are incremented by one. By having *loopLevel* as three, loop parameters are fed to register R3 in all the PEs. The next cycle resembles cycle six and PC is incremented by one in all the PEs. Loop L4 is initialized by the instructions executed in cycle ten and eleven. *loopLevel* is updated as four choosing register R4 as the active register. PC and the loop end address in R4 are found to be the same in cycle twelve and the loop count sign bit is zero. Therefore conductor-PE sets *loopStartFlag* to one and decrements loop count in R4 by one. Performer-PEs check the flag and PC is updated with loop start address in R4. In cycle thirteen as well, PC and loop end address of L4 are the same but this time, the loop count sign bit is non-zero indicating that the loop L4 has finished its execution.

Before setting the *loopLevel* to point to the parent loop, L3, conductor-PE compares its PC value against loop end address stored in R3 to check whether that was the last instruction of L3 as well. But it is not and hence L3 is activated by making *loopLevel* to 3. In this case as well, *loopStartFlag* is zero and PC is incremented by one to point to the next instruction in loop L3. In the next cycle, it is found that loop L3 is completed and hence a check on the parent loop L2 is also done. From the values of PC, loop end address and loop count in R2, it is evident that L2 also finishes its execution. Then a loop end check on loop L1 is done. This fails and hence loop L1 becomes active, setting *loopLevel* to one and *loopStartFlag* to zero. The instruction executed in cycle fifteen is again a loop instruction pushing *loopLevel* value to two choosing the second level register R2 to hold the L5 configuration data from decoder. In cycle seventeen, PC value is found to be the end address of both L5 and L1 and the loop count values of both the loops indicate that they are in their last iteration. Consequently *loopLevel* is decremented by two bringing its value to zero. This indicates that the entire loop execution will be over by the end of this cycle. As *loopStartFlag* is zero, PC is incremented by one which takes the control out of the loop structure in every PE.

## 5 Compilation Flow

The IPA compilation flow is augmented with a pre-mapping CDFG transformation phase for hardware loop support. Both distributed and centralized versions of the hardware loop model employ the same compilation flow which is given in Figure 6(a). The kernel to be accelerated is converted into a control and data flow graph (CDFG) using GCC 4.8, with nodes representing basic blocks (BBs) and edges representing the control flow between them. The CDFG is so constructed that it aids in the software-based implementation of loops. A cyclic-to-acyclic graph transformation is done on this CDFG to eliminate injecting extra instructions into the kernel for loop control. Transformed

CDFG is then fed to the mapping module that performs scheduling and placement of nodes in a block by block fashion. This is essentially a mapping of the CDFG on to the time extended model of PE array comprising of operator and register nodes. The best of the several mappings identified is chosen and corresponding assembly code is generated. Using the IPA ISA extended with two dedicated loop configuration instructions, assembler converts it into a series of configurations comprising of the instructions and constants to be loaded onto the PE array. Figure 6(b) lists down the major steps in CDFG transformation. The same is explained with an example in the following section.

### 5.1 CDFG Transformation

Figure 7 shows a sample program, the corresponding CDFG and the transformed version of it. In this figure, rectangles represent basic blocks (BBs) and the arrows depict the flow of control from one BB to another. True and false paths in the case of conditional jumps are represented by solid and dotted arrows respectively. The execution flow of the CDFG in Figure 7(b) can be presented as: $BB\_1 \rightarrow BB\_2 \rightarrow (either\ BB\_3\ or\ BB\_4)\ if\ BB\_3 \rightarrow BB\_5 \rightarrow (either\ BB\_7\ or\ BB\_8)\ if BB\_8 \rightarrow BB\_5$... and so forth. Let the three loops in the sample program with loop control variables i, j and k be called L1, L2 and L3 respectively. Loop L2 forms a cycle in the CDFG, ($BB\_5 \rightarrow BB\_8 \rightarrow BB\_5$) and its outer loop L1 forms another, given by ($BB\_2 \rightarrow BB\_3 \rightarrow L2 \rightarrow BB\_7 \rightarrow BB\_2$). Similarly, L3 as well. The software-based IPA loop model identifies the first BB in the loop body as the loop header and the last one as the loop latch as it has a back jump to the header BB. For instance, $BB\_2$ is the loop header of loop L1 and $BB\_7$, the loop latch. Condition checking on the loop control variable is done in loop header and its update is done in the latch BB.

As the first step, a pre-checking on the loop at hand is done to confirm whether it can be run as a hardware loop or not since the design supports only up to a given number of (typically four) levels of loop nesting. Once the loop is selected, the loop count 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 first BB in the set of basic blocks that forms the loop body is identified and its DFG is modified to insert a node holding the computed loop count value. As the next step, the loop condition checking BB, which is the present loop header is eliminated from the CDFG and the first BB of the loop is designated as the Hardware Loop Header (HLH). As shown in Figure 7(c), loop headers, $BB\_2$, $BB\_5$ and $BB\_6$ are eliminated and $BB\_3$, $BB\_8$ and $BB\_9$ are marked as the HLH of loops L1, L2 and L3 respectively. The last BB in the loop body referred as loop latch is now set as the Hardware Loop Terminal (HLT). The only predecessor of the eliminated condition checking BB was the one in which the loop control variable is initialized. The BB designated as HLH becomes its successor. Jump from the last BB in the loop (HLT) to the eliminated BB is removed next.

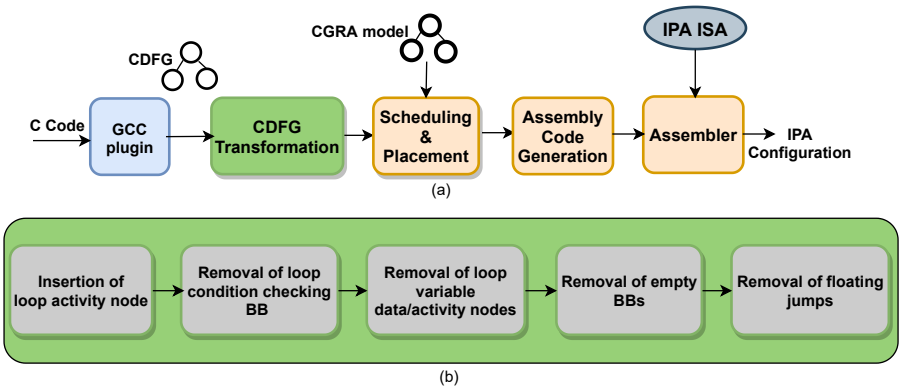


**Fig. 6** (a) IPA Compilation Flow with CDFG Transformation for Hardware Loop Support; (b) Major Steps in CDFG Transformation

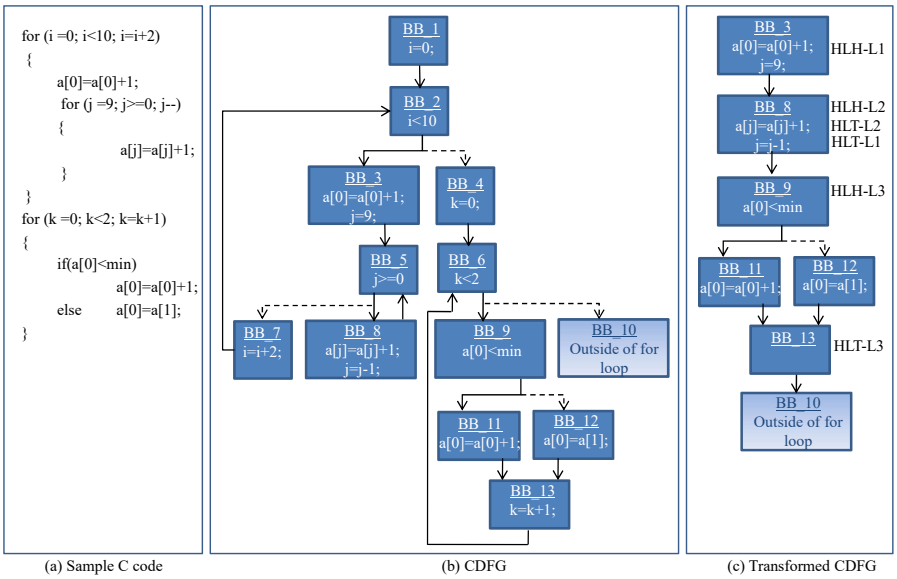


**Fig. 7** (a) Sample program; (b) Corresponding CDFG; (c) Transformed CDFG

HLT is then connected to the first BB that appears in the false path of the removed condition checking BB as its predecessor. This eliminates every cycle in the CDFG transforming it into an acyclic graph.

Next step in the transformation is to perform a conditional elimination of the control variable and its associated activities from the CDFG. The decision is made after checking the entire CDFG to see whether the loop control variable is used other than to control the looping. If found to be used elsewhere, then its initialization and increment/decrement operations are preserved. Otherwise, nodes that manipulate the control variable are removed from the respective

BBs. Once the loop variable is eliminated, we proceed to determine whether the BB which does the variable initialization can be removed or not. If the BB is totally dedicated for initializing the control variable then that BB as a whole will be pulled out of the CDFG. In the given example, control variables i and k of loops L1 and L3 are used only for the looping mechanism where as j of loop L2 is used as an operand in the loop body. Hence every operation on variable j except the condition checking is preserved. In the case of loop L1, $BB\_1$ which was dedicated for initializing variable i is removed. From loop L3, $BB\_4$ which initializes its control variable k is eliminated. Similarly update of k is removed from $BB\_12$ and that of i is removed from $BB\_7$ making them empty. Both $BB\_12$ and $BB\_7$ were terminal BBs. The case of removing an empty BB is handled carefully if it happens to be the HLT of the loop. Such a BB is removed only if it is possible to correctly determine which BB can become the new HLT. There may be cases like in loop L3 where there exists multiple predecessors to HLT as the new HLT candidate. In such cases the HLT BB will be preserved even if it is empty to mark the end of the loop body. This explains why the empty BB, $BB\_12$ is in the transformed CDFG.

As the last step, merging of the HLTs of outer and inner loops is done if feasible. The conditions to be satisfied by the HLT of outer loop for the merging are: i) it should be an empty block (as a result of eliminating loop variable update) ii) the HLT of inner loop should be its only immediate predecessor. Since the requirements are met, the empty BB, $BB\_7$ is removed from the CDFG and its only predecessor, $BB\_8$ is set as the HLT of both L1 and L2. This merging is performed to facilitate synchronous loop count update/termination of inner and outer loops with same loop end address. Resultant CDFG is a very simple acyclic graph representing a significantly reduced set of instructions as can be seen in Figure 7(c).

## 6 Experiments and Results

This section analyses the efficiency of the proposed approach in different metrics such as performance, area and energy consumption. Area and energy results of the basic distributed hardware loop model are also discussed. Experimental results are compared against those of the baseline IPA architecture that operates on software-based loop implementation. IPA reports an energy efficiency improvement up to 18×, with an average of 9.23× and a maximum speed-up of 20.3×, with 9.7× on average [7] compared to a RISC-V core [9] specialized for ultra-low power near-sensor processing. A set of loop intensive kernels with varying structure of loop nesting from different application domains are chosen for the experiments. The list includes kernels used in other works like Binarization and Erosion from [26], Seidel-2D, Jacobi-1D and Floyd Warshall from [3] and FIR Filter, Matrix Multiplication and Non-Separable Filter from [7]. Table 2 gives the computational features of the kernels such as number of levels of loop nesting with the number of loops at each level, number of conditional jumps present and arithmetic intensity. Based whether condi-

**Table 2** Listed kernels and their computational features

| Kernel | No. of Levels of Nesting | Total No. of Ins | No. of Ins in Loop | No. of Load/Store Ins | No. of Arithmetic Ins | No. of Cond Jumps | Arithmetic Intensity | Arithmetic Intensity (Innermost Loop) |
|---|---|---|---|---|---|---|---|---|
| Control-Oriented | | | | | | | | |
| 2D Convolution | 4(1-1-1-1) | 59 | 46 | 3 | 33 | 5 | 11.00 | 8.00 |
| Dilation | 4(1-1-1-1) | 50 | 34 | 4 | 21 | 5 | 5.25 | 4.00 |
| Erosion | 4(1-1-1-1) | 46 | 32 | 4 | 20 | 3 | 5.00 | 3.00 |
| Projection | 2(2-2) | 41 | 25 | 6 | 13 | 3 | 2.17 | 1.25 |
| Floyd Warshall | 3(1-1-1) | 33 | 22 | 6 | 12 | 1 | 2.00 | 0.67 |
| Binarization | 2(1-1) | 23 | 13 | 3 | 6 | 1 | 2.00 | 0.67 |
| Data-Oriented | | | | | | | | |
| Matrix Addition | 2(1-1) | 20 | 10 | 3 | 6 | 0 | 2.00 | 0.67 |
| Seidel-2D | 3(1-1-1) | 39 | 30 | 10 | 18 | 0 | 1.80 | 1.00 |
| Jacobi-1D | 2(1-2) | 38 | 29 | 8 | 19 | 0 | 2.38 | 1.50 |
| FIR filter | 2(1-1) | 22 | 12 | 3 | 8 | 0 | 2.67 | 1.50 |
| Histogram Equalization | 2(1-1) | 20 | 10 | 2 | 7 | 0 | 3.50 | 1.50 |
| Matrix Multiplication | 3(1-1-1) | 28 | 17 | 3 | 12 | 0 | 4.00 | 1.50 |
| Non-Separable Filter | 4(1-1-1-1) | 36 | 24 | 3 | 18 | 0 | 6.00 | 1.50 |

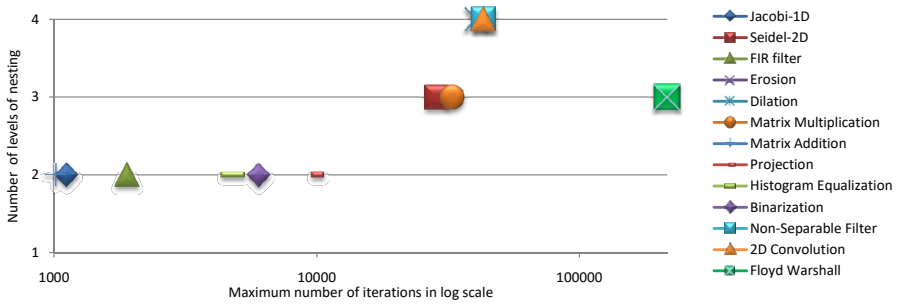


**Fig. 8** Spectrum covered by the depth of nesting and iteration counts of the considered kernels

tional jumps are present or not, kernels are categorized into control-oriented and data-oriented. Figure 8 depicts the spectrum covered by their number of levels of nesting and the maximum number of iterations.

## 6.1 Implementation results

In this section, implementation results for baseline and proposed architectures are discussed. The baseline IPA is configured as a 4×2 PE array with each PE comprising of a 21×64-bits IRF, a 20×32-bits CRF and a 32×8-bits RRF. IPA is modified by integrating HLB unit in each of the 8 PEs for the basic distributed version of the hardware loop model. In the centralized implementation, PE-0-0 in the IPA PE array is set as the conductor-PE and the remaining seven PEs as the performer-PEs. Both distributed and centralized models are configured to support up to four levels of loop nesting, which suffices for the studied benchmarks. The model can be extended to support any number of nested levels at the cost of including one configuration register per level of nesting. The designs were synthesized with Cadence Genus Synthesis Solution $17.22 - s017\_1$ using 90nm CMOS technology libraries. Placement and Routing was performed using Cadence Innovus $17.14 - s077\_1$. Cadence Voltus Power Analysis - Power Calculator $17.21 - s032\_1$ was used for power analysis

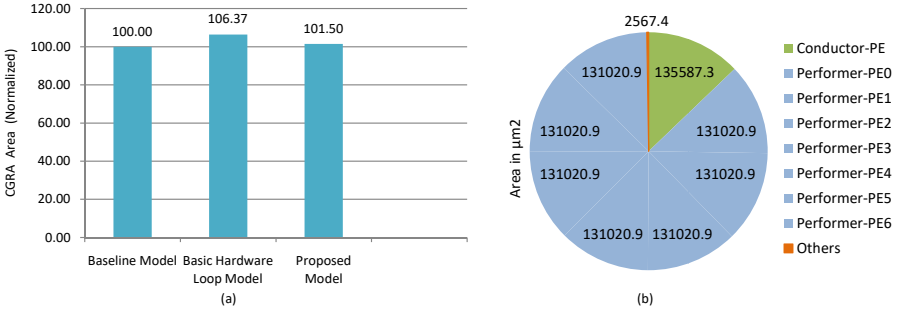


**Fig. 9** (a) Synthesized area of IPA PE array in different implementations; (b) Area breakdown of IPA (4X2) in the proposed model

at the supply of 0.9 V, in typical process conditions. Simulation results for performance analysis were collected using Mentor Questasim-64 10.7b-1.

Figure 9(a) gives a comparison of synthesized area of the PE array for baseline and the two versions of hardware-based loop model. It is evident from the chart that the area overhead for the hardware loop implementation can be significantly reduced by adopting the proposed conductor-performer model. The 6.4% increase in CGRA size in the distributed version is reduced to 1.5% in the centralized model. The area reduction achieved by the centralized over the distributed version will be more in the case of CGRAs of bigger dimensions ($8 \times 8$ for example). The area breakdown of IPA with centralized hardware loop model is depicted in Figure 9(b).

## 6.2 Performance analysis

Compared to the distributed version, centralized implementation does not suffer any performance degradation in terms of latency or the number of instructions executed. Both implementations give the same performance results and hence only the proposed model is discussed against the baseline.

### *6.2.1 Effect of Hardware-based Optimization*

Table 3 gives the average (arithmetic) and maximum gain achieved in reducing various parameters like latency and total number of instructions. With the proposed model, total number of instructions executed is reduced by 2.63× compared to the baseline for Floyd Warshall and Jacobi-1D kernels. The average gain obtained is 1.93×. Along with achieving better performance and energy efficiency, fewer number of instructions can result in reduced configuration size/instruction memory occupancy and lesser configuration load time. Up to 77.76× and an average of 26.20× gain is observed in reducing the number of branch instructions. Latency in terms of execution cycles is noted for all the kernels. This corresponds to the total execution time including the configuration (instructions and constants) load time. Latency is found to be reduced

**Table 3** Gain achieved by the proposed over the baseline model

| Parameter | Average Gain | Maximum Gain |
|---|---|---|
| No. of Instructions Executed | 1.93× | 2.63× |
| No. of Branch Instructions Executed | 26.20× | 77.76× |
| No. of Execution Cycles | 1.49× | 1.97× |
| No. of Basic Blocks Mapped | 1.32× | 1.57× |

up to 1.97× (for Matrix Multiplication) and 1.49× on average by the proposed model. The table also presents the gain in reducing the number of basic blocks that would correspond to a reduction in compilation time and the number of branch instructions executed. Thanks to the efficient CDFG transformation phase in the compilation flow, the number is reduced up to 1.57× and 1.32× on average compared to the baseline.

### *6.2.2 Combined Effect of Hardware and Software-based Optimizations*

IPA supports loop unrolling as an optimization technique to reduce loop overhead and also to leverage spatial parallelism. The innermost loop is fully or partially unrolled prior to mapping. Unrolling factor is chosen depending on loop count and number of instructions in the loop as well as the PE array size. By employing loop unrolling, IPA achieves better latencies at the cost of increased number of instructions. Hardware loop alone reduces latency as well as the number of instructions executed. Combining these two, better results are obtained on both parameters. For instance, while running the Histogram Equalization kernel, IPA attains 3.72× speed-up by unrolling the innermost loop by a factor of four when the number of instructions is increased by 15%. On IPA augmented with hardware loop support, a speed-up of 1.49× and a reduction of 49% in the number of instructions are achieved for the same kernel without unroll. Executing the unrolled version on the IPA hardware loop model, speed-up is increased to 5.46× when the number of instructions is reduced by 42%. Figure 10 presents the gain achieved in latency by executing kernels with loops unrolled on HLB-integrated-IPA against the baseline IPA loop model without any optimizations applied. For almost all the kernels gain improves with the loop unrolling factor. Innermost loops that iterate for fewer number of times are fully unrolled. By fully unrolling, the loop is entirely taken off and hence the gain on innermost loop execution is solely from unrolling with no contribution from the hardware-based optimization. This explains why there is no notable improvement in gain for kernels like Erosion and Dilation.

To indicate how well the proposed solution would perform over the CPU, a comparison of latency in terms of execution cycles between RISC-V processor and IPA integrated with hardware loop is given in Table 4. Results on the proposed model are gathered by executing loop unrolled kernels; unrolling factor is given in the table. It can be seen that the proposed approach outperforms the CPU by a big margin achieving an average (arithmetic) gain of 23.13×.

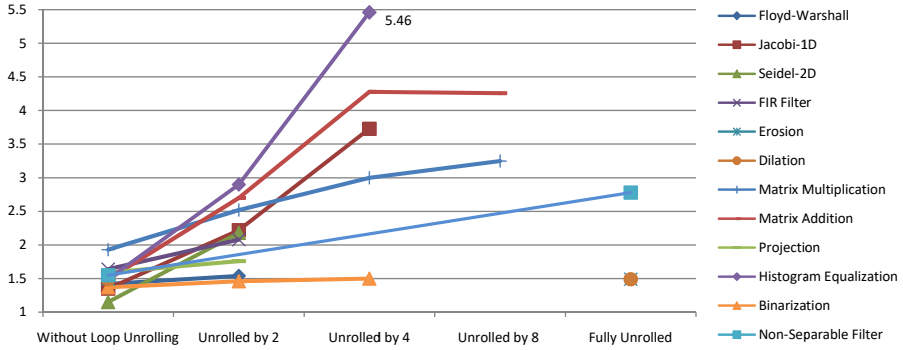


**Fig. 10** Gain in latency achieved against the baseline model by applying loop unrolled kernels on the proposed model

**Table 4** Comparison of latency in terms of execution cycles between CPU and the proposed model

| Kernel | Unroll Factor | CPU | Proposed Model | Gain |
|---|---|---|---|---|
| Floyd Warshall | 2 | 31,752,000 | 1,426,779 | 22.25× |
| Jacobi-1D | 4 | 277,201 | 4,075 | 68.02× |
| Seidel-2D | 2 | 5,487,200 | 308,385 | 17.79× |
| FIR filter | 2 | 62,091 | 7,370 | 8.42× |
| Erosion | Full (3) | 272,147 | 27,996 | 9.72× |
| Dilation | Full (3) | 443,879 | 27,996 | 15.86× |
| Matrix Multiplication | 8 | 1,311,354 | 83,618 | 15.68× |
| Matrix Addition | 8 | 49,819 | 2,296 | 21.70× |
| Projection | 2 | 1,478,400 | 44,708 | 33.07× |
| Histogram Equalization | 4 | 361,518 | 8,086 | 44.71× |
| Binarization | 4 | 260,300 | 44,459 | 5.85× |
| Non-Separable Filter | Full (3) | 2,459,761 | 170,357 | 14.44× |

Subsequent tables give a comparison of performance efficiency between baseline and proposed models when loop unrolling is employed on both the models. Table 5 compares the total number of instructions executed on baseline and proposed models for various loop unrolling factors. The effect of unrolling on achievable gains can be explained in terms of $(N_{ke})$ and $(N_{lc})$ formulated in section 3. Loop overhead is proportional to the ratio of $(N_{lc})$ to $(N_{ke})$. By partially unrolling a loop, loop count is reduced and the number of kernel instructions in the loop body is increased by the unrolling factor. As a result, $(N_{lc})$ which is proportional to the loop count decreases while $(N_{ke})$ remains the same. Thus the overhead of executing loops on the baseline model reduces with loop unrolling factor. Lesser the overhead, lower will be the gain achieved by the proposed solution over the baseline model. This fall with loop unrolling factor is evident in the gain on total number of instructions executed. However, the trend is not reflected in the results of all kernels since other parameters like PE utilization can affect the figures. The number of independent kernel instructions is often increased by unrolling the loop. This will create

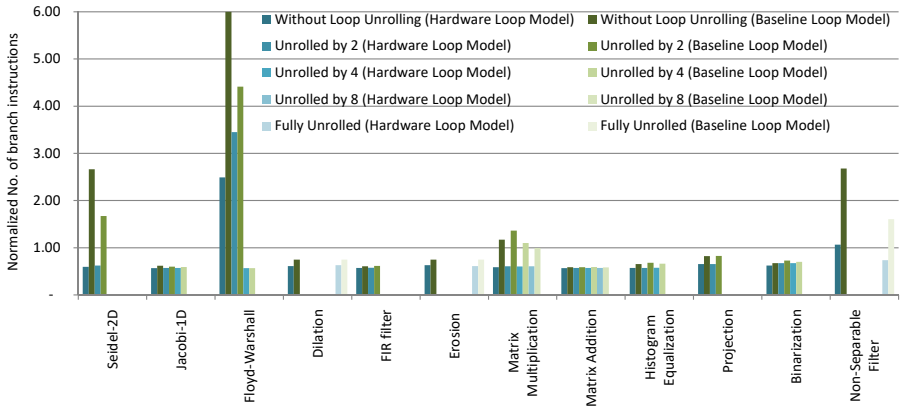


**Fig. 11** Effect of loop unrolling on the number of branches executed on baseline and proposed models

more flexible mapping opportunities, leading to better PE utilization. With more number of PEs participating, the number of branch instructions will be increased. This is because control flow altering instructions like jump and conditional jump instructions are executed by each and every PE following the globally synchronized execution model of IPA. This increases the overhead of executing loops on the baseline model and consequently, the gain will be more.

Figure 11 gives the number of branch instructions executed in normalized form for different unrolling factors for both baseline and proposed models. In any case, lesser number of branches are executed by the proposed model compared to the baseline. For both the models, the number grows with the unrolling factor if better PE utilization is achieved. However, the increase in the number of branch instructions with loop unrolling factor does not increase the number of execution cycles in the baseline implementation. This is because the corresponding branch instructions are executed simultaneously by all the PEs. Therefore the gain on latency achieved by the proposed over the baseline model depends solely on the measure of overhead eliminated. As discussed above, the overhead (given by the ($N_{lc}$) to ($N_{ke}$) ratio) falls with loop unrolling factor and the gain on latency decreases accordingly. This is evident in Table 6. Table 7 gives a comparison on the number of basic blocks between baseline and proposed models in the presence of loop unrolling. It is observed that the number of basic blocks is not affected by unrolling unless the unrolled loop contains conditional statements. Number of basic blocks increases with the unrolling factor if conditionals are present; Binarization kernel is an example.

The performance of the proposed model expressed in Million Operations Per Second (MOPS) for the studied benchmarks is given in Table 8. It also lists the CGRA occupation which is the average fraction of the PE array active per execution cycle.

**Table 5** Comparison of total number of instructions executed between baseline and proposed models in the presence of loop unrolling

| Kernel | Unroll Factor | Baseline Model | Proposed Model | Gain Achieved |
|---|---|---|---|---|
| Floyd Warshall | 2 | 2,650,585 | 2,549,917 | 1.04× |
| Jacobi-1D | 2 | 26,414 | 16,763 | 1.58× |
| | 4 | 22,292 | 15,257 | 1.46× |
| Seidel-2D | 2 | 1,205,192 | 976,097 | 1.23× |
| FIR filter | 2 | 30,816 | 17,897 | 1.72× |
| Erosion | Full (3) | 92,379 | 37,154 | 2.49× |
| Dilation | Full (3) | 92,379 | 51,205 | 1.80× |
| Matrix Multiplication | 2 | 566,573 | 254,757 | 2.22× |
| | 4 | 419,382 | 188,986 | 2.22× |
| | 8 | 383,048 | 172,899 | 2.22× |
| Matrix Addition | 2 | 12,324 | 7,150 | 1.72× |
| | 4 | 13,896 | 6,201 | 2.24× |
| | 8 | 10,568 | 6,729 | 1.57× |
| Projection | 2 | 138,279 | 64,351 | 2.15× |
| Histogram Equalization | 2 | 72,300 | 41,335 | 1.75× |
| | 4 | 67,592 | 33,913 | 1.99× |
| Binarization | 2 | 79,618 | 61,217 | 1.30× |
| | 4 | 67,618 | 58,218 | 1.16× |
| Non-Separable Filter | Full (3) | 717,645 | 380,688 | 1.89× |

### 6.3 Energy Results

Energy estimates are prepared using the switching activity obtained by simulating the placement-and-routed net list design at a clock frequency of 17.5MHz. Results include the energy spent on the entire PE array including the LSUs interfacing with TCDM and the interconnects used for data transfer between PEs. As expected, the energy consumption is considerably reduced by the hardware-based loop model in comparison with the baseline that uses the conventional looping technique. Table 9 lists down the energy results for the compared designs over various kernels with and without loop unrolling. A reduction up to 44.5% and an arithmetic mean of 23.3% is achieved by the basic distributed model compared to the baseline model employing software loop. Furthermore, a maximum of 47.3% and an arithmetic mean of 27.2% reduction in energy consumption is attained by the centralized approach over the baseline model. Compared to the basic distributed model of hardware loop, the centralized version reduces the energy consumption by 5.02% on average. In the presence of loop unrolling, energy consumption is reduced on all the three designs. These results confirm the inter-operability of the proposed hardware-based optimization with any of the existing software-based optimization techniques. For instance, the energy consumption of $77.81\mu J$ recorded for Matrix Multiplication kernel on the baseline IPA architecture is reduced to $24.36\mu J$ by unrolling the innermost loop by a factor of 8 and executing it on the centralized hardware loop model. This marks a reduction of 68.7% in energy consumption.

**Table 6** Comparison of latency in terms of execution cycles between baseline and proposed models in the presence of loop unrolling

| Kernel | Unroll Factor | Baseline Model | Proposed Model | Gain Achieved |
|---|---|---|---|---|
| Floyd Warshall | 2 | 1,771,147 | 1,426,779 | 1.24× |
| Jacobi-1D | 2 | 9,377 | 6,856 | 1.37× |
| | 4 | 4,973 | 4,075 | 1.22× |
| Seidel-2D | 2 | 353,299 | 308,385 | 1.15× |
| FIR filter | 2 | 10,602 | 7,370 | 1.44× |
| Erosion | Full (3) | 41,724 | 27,996 | 1.49× |
| Dilation | Full (3) | 41,733 | 27,996 | 1.49× |
| Matrix Multiplication | 2 | 190,068 | 108,075 | 1.76× |
| | 4 | 149,144 | 90,622 | 1.65× |
| | 8 | 128,738 | 83,618 | 1.54× |
| Matrix Addition | 2 | 5,227 | 3,619 | 1.44× |
| | 4 | 2,976 | 22,85 | 1.30× |
| | 8 | 2,752 | 2,296 | 1.20× |
| Projection | 2 | 59,454 | 44,708 | 1.33× |
| Histogram Equalization | 2 | 22,620 | 15,243 | 1.48× |
| | 4 | 11,854 | 8,086 | 1.47× |
| Binarization | 2 | 55,138 | 45,938 | 1.20× |
| | 4 | 49,163 | 44,459 | 1.11× |
| Non-Separable Filter | Full (3) | 214,773 | 170,357 | 1.26× |

### 6.4 Comparison to other approaches

The reference CGRA architecture used by the ZOLA model [26] is a configurable array of Functional Unit(FU)s. To run applications involving loops, the reference CGRA will be customized to include an extra FU as an ALU to compute the loop control flow decisions. It also requires another FU as an Immediate Unit (IMM) and a custom instruction in loop body to support branching. By having the zero-overhead loop accelerator integrated with the Accumulate Branch Unit(ABU), the use of extra ALU and IMM units and the associated issue slots could be avoided resulting in an average energy gain of 6.8% against the reference CGRA when tested on a set of image processing kernels. ZOLA tries to reduce energy consumption by removing the extra FUs its reference architecture employs to implement loop control flow. The solution we introduce aims to minimize the number of loop control instructions executed by the baseline IPA architecture, rather than trying to reduce the PE usage. In ZOLA model, the FU and issue slot replacement is not possible if the units are used for other computations as well. Hence ZOLA is an advisable choice only for those applications for which its energy overhead is lower than what it saves. Our approach could cut down the number of instructions executed to half for almost all the kernels we tested with. The centralized hardware loop model could achieve as much as 68.7% and 47.3% reduction in energy consumption with and without loop unrolling against the baseline.

The LASER model [2] outperforms the state-of-the-art partial predication techniques in accelerating complicated loops with nested conditionals.

**Table 7** Comparison of number of basic blocks between baseline and proposed models in the presence of loop unrolling

| Kernel | Unroll Factor | Baseline Model | Proposed Model | Gain Achieved |
|---|---|---|---|---|
| Floyd Warshall | 2 | 18 | 15 | 1.20× |
| Jacobi-1D | 2 | 14 | 10 | 1.40× |
| | 4 | 14 | 10 | 1.40× |
| Seidel-2D | 2 | 14 | 10 | 1.40× |
| FIR filter | 2 | 11 | 9 | 1.22× |
| Erosion | Full (3) | 23 | 20 | 1.15× |
| Dilation | Full (3) | 23 | 20 | 1.15× |
| Matrix Multiplication | 2 | 14 | 11 | 1.27× |
| | 4 | 14 | 9 | 1.56× |
| | 8 | 14 | 11 | 1.27× |
| Matrix Addition | 2 | 11 | 9 | 1.22× |
| | 4 | 11 | 7 | 1.57× |
| | 8 | 11 | 9 | 1.22× |
| Projection | 2 | 25 | 21 | 1.19× |
| Histogram Equalization | 2 | 11 | 9 | 1.22× |
| | 4 | 11 | 9 | 1.22× |
| Binarization | 2 | 17 | 15 | 1.13× |
| | 4 | 23 | 21 | 1.10× |
| Non-Separable Filter | Full (3) | 14 | 11 | 1.27× |

**Table 8** MOPS and CGRA occupation of the proposed model for the studied benchmarks

| Kernel | MOPS | CGRA Occupation % |
|---|---|---|
| Floyd Warshall | 41.41 | 29.5 |
| Jacobi-1D | 122.53 | 87.3 |
| Seidel-2D | 109.84 | 78.3 |
| FIR filter | 62.20 | 44.3 |
| Erosion | 53.88 | 38.4 |
| Dilation | 46.64 | 33.2 |
| Matrix Multiplication | 66.58 | 47.4 |
| Matrix Addition | 106.51 | 75.9 |
| Projection | 31.18 | 22.2 |
| Histogram Equalization | 110.74 | 78.9 |
| Binarization | 30.22 | 21.5 |
| Non-Separable Filter | 62.19 | 44.3 |

For MiBench benchmarks, LASER consumed on an average 45.78% less energy compared to reference architecture employing partial predication scheme. However, the hardware based optimization LASER showcases is entirely dedicated for the efficient execution of conditional statements. The support is extended to loops by flattening nested loops into single-level loops with nested conditionals. Loop control remains to be software-based necessitating the execution of control instructions for each iteration of the loop. The hardware based optimization we introduced completely shifts the loop control to hardware eliminating the need for executing loop control instructions.

**Table 9** Comparison of Energy($\mu J$) consumed in different IPA implementations for various kernels

| Kernel | Unroll Factor | Baseline Model | Basic Hardware Loop Model | Proposed Model |
|---|---|---|---|---|
| Floyd Warshall | - | 630.43 | 474.84 | 451.15 |
| | 2 | 506.80 | 437.54 | 415.58 |
| Jacobi-1D | - | 4.35 | 3.46 | 3.28 |
| | 2 | 2.68 | 2.10 | 2.00 |
| | 4 | 1.42 | 1.25 | 1.19 |
| Seidel-2D | - | 191.98 | 178.68 | 169.71 |
| | 2 | 101.09 | 94.57 | 89.82 |
| FIR Filter | - | 4.39 | 2.87 | 2.73 |
| | 2 | 3.03 | 2.26 | 2.15 |
| 2D Convolution | - | 154.12 | 122.44 | 116.30 |
| Erosion | - | 11.94 | 8.60 | 8.17 |
| | Full (3) | 11.94 | 8.59 | 8.15 |
| Dilation | - | 11.95 | 8.59 | 8.16 |
| | Full (3) | 11.94 | 8.59 | 8.15 |
| Matrix Multiplication | - | 77.81 | 43.15 | 40.99 |
| | 2 | 54.39 | 33.14 | 31.48 |
| | 4 | 42.68 | 27.79 | 26.40 |
| | 8 | 36.84 | 25.64 | 24.36 |
| Matrix Addition | - | 2.80 | 2.01 | 1.91 |
| | 2 | 1.50 | 1.11 | 1.05 |
| | 4 | 0.85 | 0.70 | 0.67 |
| | 8 | 0.79 | 0.70 | 0.67 |
| Projection | - | 22.49 | 15.17 | 14.41 |
| | 2 | 17.01 | 13.71 | 13.02 |
| Histogram Equalization | - | 12.63 | 9.07 | 8.62 |
| | 2 | 6.47 | 4.67 | 4.44 |
| | 4 | 3.39 | 2.48 | 2.36 |
| Binarization | - | 19.20 | 15.00 | 14.24 |
| | 2 | 15.78 | 14.09 | 13.38 |
| | 4 | 14.07 | 13.638 | 12.95 |
| Non-Separable Filter | - | 135.62 | 94.06 | 89.34 |
| | Full (3) | 61.46 | 52.24 | 49.62 |
| **Avg. Reduction wrt Baseline Model** | | | **23.3**% | **27.2**% |
| **Max. Reduction wrt Baseline Model** | | | **44.5**% | **47.3**% |
| **Avg. Reduction wrt Basic Hw. Loop Model** | | | | **5.02**% |

## 7 Conclusion

Many proven hardware and software loop optimizations on processor architectures have been adopted to CGRAs for enhanced performance and energy efficiency. In this paper, we proposed a centralized implementation of the hardware loop that we introduced previously. Furthermore, we analysed and confirmed combining hardware loop support with software optimization techniques like loop unrolling as a means to improve the gains. Compared to

the basic hardware loop model, the proposed centralized solution could reduce area overhead against the reference architecture to 1.5% for a 4×2 CGRA configuration. By executing loop unrolled kernels on the centralized hardware loop model, up to 5.46× speed-up and 68.7% reduction in energy consumption are attained compared to the baseline with no optimizations applied. The proposed model can be further enhanced by having multiple hardware loop units in the CGRA, each controlling a cluster of PEs in centralized fashion. Such a design can aid in exploiting loop level and thread level parallelism in the kernel execution.

## Declarations

**Funding**
This work was funded by the Science and Engineering Research Board (SERB), Government of India under grant file No. SRG/2020/001005.

**Conflicts of interests**
The authors have no conflicts of interest to declare that are relevant to the content of this article.

**Acknowledgement**
First published in Journal of Signal Processing Systems, 2022 by Springer Nature

## 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. Bielecki, W., Skotnicki, P.: Insight into tiles generated by means of a correction technique. The Journal of Supercomputing **75**(5), 2665–2690 (2019)
4. Das, S.: Architecture and programming model support for reconfigurable accelerators in multi-core embedded systems. Ph.D. thesis, Lorient (2018)
5. 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)
6. 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)
7. 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)
8. Dragomir, O.S., Bertels, K.: Extending loop unrolling and shifting for reconfigurable architectures. Architectures and Compilers for Embedded Systems (ACES) pp. 61–64 (2010)

9. 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)
10. Gobieski, G., Atli, A.O., Mai, K., Lucia, B., Beckmann, N.: Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 1027–1040. IEEE (2021)
11. 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)
12. Kavvadias, N., Nikolaidis, S.: Elimination of overhead operations in complex loop structures for embedded microprocessors. IEEE Transactions on Computers **57**(2), 200–214 (2008)
13. 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)
14. Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., Wei, S.: A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. ACM Computing Surveys (CSUR) **52**(6), 1–39 (2019)
15. 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)
16. Nowatzki, T., Gangadhan, V., Sankaralingam, K., Wright, G.: Pushing the limits of accelerator efficiency while retaining programmability. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 27–39. IEEE (2016)
17. Ozaki, N., Yoshihiro, Y., Saito, Y., Ikebuchi, D., Kimura, M., Amano, H., Nakamura, H., Usami, K., Namiki, M., Kondo, M.: Cool mega-array: A highly energy efficient reconfigurable accelerator. In: 2011 International Conference on Field-Programmable Technology, pp. 1–8. IEEE (2011)
18. 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)
19. Podobas, A., Sano, K., Matsuoka, S.: A survey on coarse-grained reconfigurable architectures from a performance perspective. IEEE Access **8**, 146719–146743 (2020)
20. 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)
21. Prasad, R., Das, S., Martin, K.J., Tagliavini, G., Coussy, P., Benini, L., Rossi, D.: Transpire: An energy-efficient transprecision floating-point programmable architecture. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1067–1072. IEEE (2020)
22. Sunny, C., Das, S., Martin, K.J., Coussy, P.: Hardware based loop optimization for cgra architectures. In: International Symposium on Applied Reconfigurable Computing, pp. 65–80. Springer (2021)
23. Tian, W., Xue, C.J., Li, M., Chen, E.: Loop fusion and reordering for register file optimization on stream processors. Journal of Systems and Software **85**(7), 1673–1681 (2012)
24. Torng, C., Pan, P., Ou, Y., Tan, C., Batten, C.: Ultra-elastic cgras for irregular loop specialization. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 412–425. IEEE (2021)
25. 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)

26. 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)
27. Zhang, Y., Li, G., Yang, X.: Recognition and optimization of loop-carried stream reusing of scientific computing applications on the stream processor. In: International Conference on Computational Science, pp. 474–481. Springer (2007)
28. Zhang, Y., Zhang, N., Zhao, T., Vilim, M., Shahbaz, M., Olukotun, K.: Sara: Scaling a reconfigurable dataflow accelerator. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 1041–1054. IEEE (2021)
29. Zheng, S., Zhang, K., Tian, Y., Yin, W., Wang, L., Zhou, X.: Fastcgra: A modeling, evaluation, and exploration platform for large-scale coarse-grained reconfigurable arrays. In: 2021 International Conference on Field-Programmable Technology (ICFPT), pp. 1–10. IEEE (2021)