LASER: A Hardware/Software Approach to Accelerate Complicated Loops on CGRAs

Mahesh Balasubramanian*, Shail Dave*, Aviral Shrivastava*, Reiley Jeyapaul†
*Compiler Microarchitecture Lab, Arizona State University, Tempe, AZ
†ARM, Cambridge, United Kingdom
Email: {mbalasubramanian, shail.dave, aviral.shrivastava}@asu.edu, {reiley.jeyapaul}@arm.com

***Abstract*—Coarse-Grained Reconfigurable Arrays (CGRAs) are popular accelerators predominantly used in streaming, filtering, and decoding applications. Due to their high performance and high power-efficiency, CGRAs can be a promising solution to accelerate the loops of general purpose applications also. However, the loops in general purpose applications are often complicated, like loops with perfect and imperfect nests and loops with nested if-then-else's (conditionals). We argue that the existing hardware-software solutions to execute branches and conditions are inefficient. In order to efficiently execute complicated loops on CGRAs, we present a hardware-software hybrid solution: LASER – a comprehensive technique to accelerate compute-intensive loops of applications. In LASER, compiler transforms complex loops, maps them to the CGRA, and lays them out in the memory in a specific manner, such that the hardware can fetch and execute the instructions from the right path at runtime. LASER achieves a geomean performance improvement of 40.91% and utilization of 43.43% with 46% lower energy consumption.**

I. Introduction

Accelerators have now become an integral part of the modern processor design to accelerate specialized or compute-intensive part of the code. CGRAs are programmable, yet power-efficient accelerators [1]. As shown in Fig 1, a CGRA is an array of processing elements (PEs) connected in a 2-D mesh. Each PE consists of functional unit (FU) for computation, and a register file (RF) to store values. The PEs can get inputs from the neighboring PEs, RF or the data memory. In each cycle, instructions to be executed are issued to every PE. The performance and power-efficiency of CGRA rely on the compiler technology [2]–[4].

The main advantage of CGRAs over custom ASIC (Application Specific Integrated Circuit) and FPGA (Field Programmable Gate Arrays) accelerators is the higher-level of programmability. CGRAs can be programmed at instruction-level, whereas FPGAs are programmed at bit-level [5]. This makes programming much simpler for CGRAs. As opposed to GPUs (Graphics Processing Units), CGRAs can accelerate non-parallel loops also [6].

CGRAs are popular in streaming applications, e.g., set-top boxes, TVs, projectors, for filtering and decoding [1], [7], [8], and over the years, several compiler techniques have been developed to map the innermost loops without conditionals (if-then-else) in the applications on CGRAs [2], [3], [5]. Our vision is to exploit the advantages of CGRAs in general-purpose processors to accelerate compute-intensive loops of general-purpose applications. However, the compute-intensive loops in real general-purpose applications are complex. They often feature several levels of loop nests and nested conditionals, which can be perfect or imperfectly nested (nested loops where the outer loops contains the inner loops along with one or more assignments). Mapping imperfectly nested loops also requires the ability to map loops with conditionals, since they must be (and can be) converted into loops with conditionals.

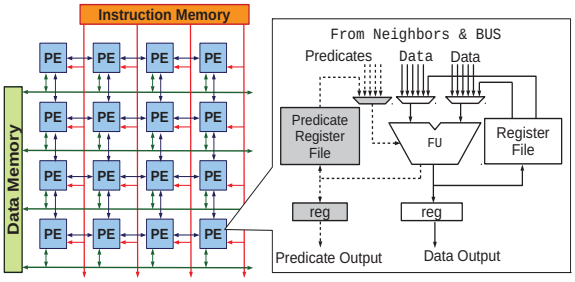


Fig. 1: A 4x4 CGRA with PEs connected in a 2-D mesh. PE consists of an ALU and RF. Additional predicate RF and mux (shaded) are required in each PE by full and partial predication.

The state-of-the-art CGRA compiler techniques cannot map complex loops or give out a mapping that achieves only marginal speedups. The most popular approach to map loops with conditionals is to use partial predication [9]. While this approach can be applied to loops with arbitrary nesting of conditionals, it increases the number of operations to be executed. For our set of compute-intensive loop kernels from MiBench [10], the partial predication approach will increase the number of operation and seriously degrades the ability of CGRA to accelerate the kernel.

To execute complicated loops (with imperfectly nested loops and arbitrary nesting of conditionals), we propose a hardware-software hybrid solution: **LASER** – Loop Acceleration by Selective Execution on CGRA. This technique enhances the abilities of both compiler and hardware for achieving maximum power efficiency and performance. LASER compiler, converts the nested loops into a single loop with conditional statements and fuses the operation of both paths of the conditional (the true-path and the false-path) to the CGRA, so that only one of them is issued and executed. This ensures high-

978-3-9819263-0-9/DATE18/©2018 EDAA

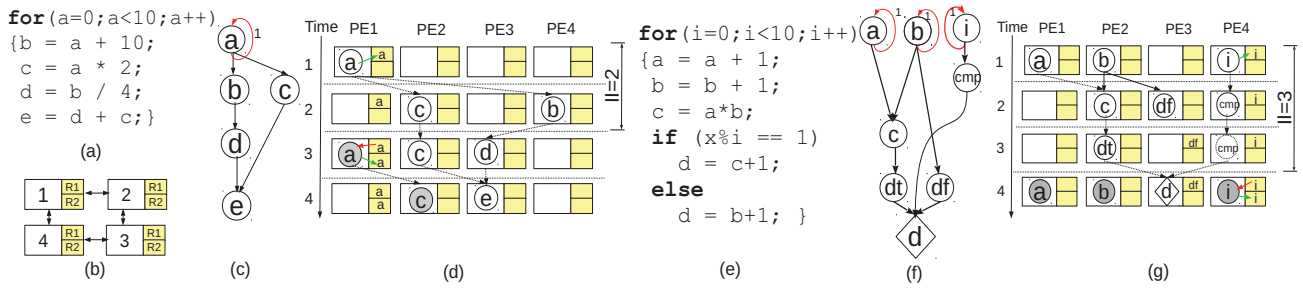


Fig. 2: a) A simple loop to be accelerated on CGRA (b) Flattened 2×2 CGRA where each PE has 2 registers (c) Data-Dependence Graph (DDG) of a simple loop (d) Mapping of the DDG onto the CGRA with II=2. (e) A loop with an if-then-else (f) DDG of the loop with Partial Predication (f) Mapping of DDG on 2×2 CGRA with II=3.

utilization of resources of CGRA. The instruction fetch unit enhancement ensures that only the correct instruction is issued at runtime, based on the branch outcome. LASER outperforms the state-of-the-art partial predication with 43.43% better utilization of PE resources and 40.91% better performance.

II. Background and Terminology

Fig 2(a)-(d) explains how a CGRA executes a compute intensive loop. Fig 2(a) shows a simple loop with 4 operations, to be executed on a 2×2 CGRA (Fig 2(b)), in which each PE has 2 registers. The compiler constructs a *Data Dependence graph* (DDG) of the loop (as shown in Fig 2(c)). DDG is a graph, in which each node represents an operation in the loop and edges represent the dependency between the operations. Fig 2(d) shows the mapping of nodes of the DDG to CGRA at different time. The iterations are software-pipelined [3], [5], [8], and the next iteration of the loop can begin in cycle 4 (denoted by a as shaded node). The interval between the beginning of consecutive iterations is known as the *initiation interval* (II). The II of this mapping is 2. II is the performance metric of CGRA and lower the II, better the performance.

III. Limitations of Related Work

Previous compiler techniques such as [2], [3], [5] accelerate only the innermost loop and fall short in accelerating rest of the loop nest which in turn has to be executed on a core. The communication overhead also multiplies if the trip count of outer-loop is higher. Existing techniques such as [11], [12] are restricted to handle only perfectly nested loops with 2-level. On the other hand, flattening based approach of [13] is promising but restricts the scalability because of its hardware-based solution with modified PE architecture. Major techniques to accelerate loops with conditionals are - (i) Full predication, (ii) Partial Predication, (iii) Dual-Issue and (iv) Path Selection Based Mapping (PSB). Full and partial predication schemes requires predicated register files and muxes (shown in Fig 1 shaded) to communicate the branch outcome. Full predication maps the nodes from both the if- and else- path on the same PE, but at different time, so that correct value is updated at the end of the execution [6]. Partial predication allows execution of nodes from both paths simultaneously but correct outcome needs to be selected through additional select node [9]. Dual issue schemes such as [6] fetches instructions for both paths but executes instructions of only correct path based on the branch condition, but requires additional mux in each PE to select the if-path or else-path instructions and is applicable to single-level only. Path selection based approach [4] selectively issues the instruction based on the branch outcome, but is applicable to only single if-then-else. For nested-conditionals PSB relies on partial predication. In this paper, we evaluate partial predication as it is the only technique that can map loops with nested conditional at lower II.

A. Partial Predication incurs high overhead

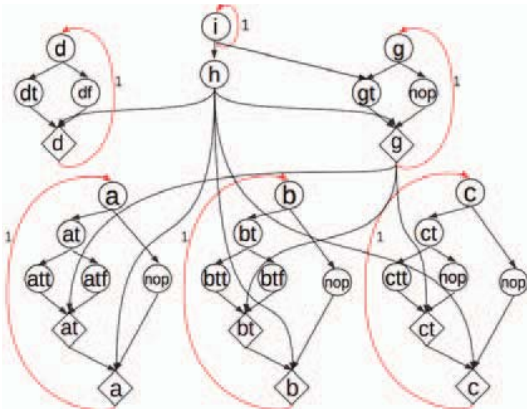
In partial predication, the nodes of DDG from both true and false paths can be mapped on different PEs and a select operation is required to choose the correct outcome based on the condition evaluated. Fig 2(e) shows a simple loop with conditional, while Fig 2(f) shows DDG using partial predication. Node cmp represents condition $x\%i==1$. Nodes d_t and d_f are true and false paths of d and a selection operation is added. Mapping of the DDG is shown in Fig 2(g) with II is 3. Due to the additional nodes required by partial predication, if a variable is computed inside the innermost nest of if-then-else, there is a corresponding node for operation inside each if-path and an else-path and so is a selection. Applying partial predication on a loop with nested conditional in Fig 3(a), we get DDG shown in Fig 3(b). Mapping DDG on 2×2 CGRA yields II of 11. Partial predication method increases the number of nodes in accelerating performance-critical loops with nested conditionals and the nested loops from MiBench benchmark suite. Clearly, there is no technique that can accelerate nested loops and nested conditionals with less overhead.

IV. Our Approach

The compiler transforms arbitrary nested (perfect or imperfect) loops into a single loop with nested conditional by **loop flattening** [13]. Fig 4 shows the transformation of a simple nested loop into a single-level loop with nested conditional. In some special cases, nested loops cannot be converted into

```
1:for(i=0;i<10;i++){
2: if (x%i==1){d+=0;
3:  if (y%i==1){
4:  a+=0;
5:  b+=0;
6:  c+=0;}else{a=a+1;
7:              b=b+1;}}
8:  else d=d+1; }
```

(a)



(b)

Fig. 3: (a) A loop with nested conditional (b) DDG using partial predication results in 31 nodes.Nodes h and g represent conditions *x%i==1* and *y%i==1*.

a single loop[1]. However, in general, loop flattening is needed to convert a nested loop to a loop with conditional statements. Executing branches on CGRA is challenging due to the lack of support from the CGRA's instruction fetch unit (IFU). The existing CGRA IFU issues instructions sequentially from the instruction memory and hence cannot jump memory addresses in case of conditional operations. In LASER, we enhance the CGRA IFU functionality to issue only the instructions of the correct path[2] at runtime. For the correct-path instructions to be issued by the IFU, LASER compiler lays out the program instruction in a specific way such that the IFU jumps to the exact memory location of instruction of the correct-path and issue them at runtime.

With this IFU support to issue correct-path instructions, if a variable c is updated in both true and false path, mapping c_t and c_f on different PEs without a *select* operation will lead to an incorrect execution. This is because the compiler generates instructions statically and since the correct-path executed is unknown at the static time, the PE that will hold the correct value of c at the end of the execution is also unknown. This discrepancy can lead to errors in the value of c at the end of program execution. To overcome this, LASER compiler fuses the true-path operation and false-path operation of the variable

[1]If a loop contains sibling loops, flattening based approach may be impractical, so a loop fission approach [13] should be used. We did not come across any compute-intensive loops that have sibling loops, in our experiments.

[2]Either true-path or false-path based on the branch outcome at runtime.

```
for (;cond1;) {
 /*statements*/
  for (;cond2;) {
   /*statements*/
  }
   /*statements*/
}
```

(a)

```
for (;cond3;) {
  if(cond4) {
   /*outer for-loop statements
       and iterator calculations*/
    }
  else {
   /* inner for-loop statements
       and iterator calculations*/
} }
```

(b)

Fig. 4: (a) An imperfectly nested loop with `cond1` and `cond2` conditions (b) Flattening converts (a) into single-level loop with conditionals with new `cond3` and `cond4`

into a single node, $\langle c_t, c_f \rangle$. This single fused node is mapped to only one PE of the CGRA and only one instruction (either true-path or false-path) is issued at runtime by the IFU. After the execution of the instruction the PE on which the fused node was mapped, holds the correct value of c. Similarly, if a variable d is updated in only one path (only in true-path (d_t) and not updated in the false-path) the compiler creates a no-operation (nop) for the false path and performs the fusing. The fused node will now have $\langle d_t, nop \rangle$, which means that if the branch condition is true d_t is issued by IFU otherwise a nop is issued. LASER compiler transforms complicated loops, maps them on to the CGRA architecture and lays the instructions in the memory in a specific manner, such that the IFU can fetch the instructions from correct-path at runtime.

A. LASER – Compiler

By evaluating the condition of a nest a priori and then mapping the true and false path of the nest on to the same PE, LASER-compiler reduces the total number of nodes created. For example, in the program of Fig 3(a), the assignments to the variable a are inside a nested if-then-else (if-else inside another if-else). So, for a conditional nest of two, four different assignments for variable a are possible. Corresponding four nodes (or operations) are fused as a single node by LASER-compiler. At runtime, correct instruction out of four possible instructions can be provided to the PE to execute the operation from the nested conditional.

Our heuristic targets fusing nodes from different if-else paths pertaining to the conditional nest. Pairing is done with operations from the innermost if-then-else (i.e., one with highest conditional depth d). The unbalanced operations (i.e. one path has more operations than the other) are paired with a no-op. For example, in program of Fig 3(a), operations corresponding to variables a, b and c are fused first. Hence, $\langle a_{tt}, a_{tf} \rangle$ and $\langle b_{tt}, b_{tf} \rangle$ are fused nodes, as shown in Fig 5(a). Such pairing is one-to-one with operations from both the paths. In our example, innermost if-path has 3 operations compared to 2 operations inside respective else path. Hence, the unbalanced operation c_{tt} is fused with a no-op. Note that we do not need any selection among the operations from if-path and else-path so, corresponding select operations are eliminated during this DDG transformation. Once the operations of the

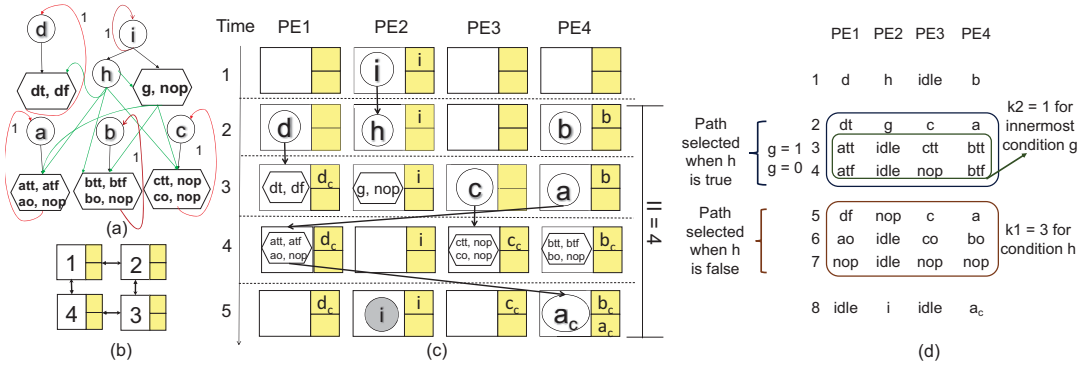


Fig. 5: (a) DDG obtained from LASER-compiler for loop of Fig 3. Nodes from multiple if-paths and else-path to a single node. If such path is absent, balancing no-ops are added and a node such as a_o preserves the old value. (b) 2×2 CGRA where each PE has 2 registers. (c) Mapping with $II = 4$. (d) Instructions are selectively issued during the execution of the kernel.

innermost conditional are fused (i.e. $y\%i == 1$), operations from outer nests can be fused iteratively. So, operations of the conditionals with nest depth of $d-1$ can be fused where d is the highest depth. Thus, we fuse all the operations associated with the condition $x\%i == 1$. The compiler iterates on the entire conditional nest and produces DDG with the fused nodes as shown in Fig 5. Mapping can be then obtained with mapping techniques such as [2], [5]. Mapping the DDG with the fused nodes, obtained from LASER-compiler is like any other mapping with CGRAs. The fused nodes can be also routed to satisfy data-dependency and necessary values are stored in the register file[3].

After obtaining the mapping for CGRA PEs, compiler generates instructions to support the execution of conditional nest. One such layout of instructions for CGRA PEs is shown in Fig 5(d). Instructions are grouped in particular manner so that hardware can easily issue the needed instructions based on the condition evaluated. Compiler associates k value with each of the conditional, which is simply number of the CGRA instructions associated. For example, first condition h ($x\%i == 1$) is evaluated on PE2 which is associated k_1=3 because maximum number of cycles required to execute the if-path or the else-path for h are three. If this condition is true, PEs should be given next three instructions from location 2– 4. In this case, PE2 is issued another conditional g ($y\%i == 1$). g is associated with k_2=1 as all fused nodes related to conditional g are mapped on PEs in a single cycle (time 4 in Fig 5(c)). So, only one instruction for each of CGRA PEs is enough to execute either if or else-path corresponding to g. If g is evaluated as false, k_2=1 instruction will be skipped at run-time. Once instructions from location 2–4 are issued, if-path corresponding to h gets over and next k_1=3 instructions are skipped, which corresponds to else-path of the outer conditional h. Then, instruction at location 8 can be executed allowing independent operations and kernel instructions executes from the location 1 again. Before the architecture can support the execution in such fashion, it is the compiler's job to associate corresponding k values with CGRA instructions and to configure the hardware correctly.

As shown in Algorithm 1, our heuristic first determines conditional with highest nest depth and pairs the nodes from both if and else paths. Pairing can proceed until there is an operation in if-path or else-path (line 5). If no such path exists or if the number of nodes in either of the paths is unbalanced, we need to fuse the nodes with no-ops (lines 8-11). Such assembling results in fused nodes after iterative pairing (lines 3-15). While forming the DDG, compiler preserves the data

Algorithm 1 FuseNodes (Input DDG D, Output DDG P)

1: $d \leftarrow getHighestConditionalDepth()$
2: **for** $i = d$ to 1 **do**
3: $n^i_{if} \leftarrow getLastNode(N^i_{if})$
4: $n^i_{else} \leftarrow getLastNode(N^i_{else})$
5: **while** $n^i_{if} \neq NULL$ or $n^i_{else} \neq NULL$ **do**
6: **if** $n^i_{if} \in N^i_{if}$ and $n^i_{else} \in N^i_{else}$ **then**
7: $fuse(n^i_{if}, n^i_{else})$
8: **else if** $n^i_{if} \in N^i_{if}$ and $n^i_{else} == NULL$ **then**
9: $fuse(n^i_{if}, nop)$
10: **else if** $n^i_{if} == NULL$ and $n^i_{else} \in N^i_{else}$ **then**
11: $fuse(nop, n^i_{else})$
12: **end if**
13: $n^i_{if} \leftarrow getLastRemainingNode(N^i_{if})$
14: $n^i_{else} \leftarrow getLastRemainingNode(N^i_{else})$
15: **end while**
16: **for** n^i_j such that $j = 0$ to $|N|$ **do**
17: **if** n^j_i is an eligible select operation $\in N^j_{other}, \ni input1(n^j_i), input2(n^j_i) = m_{fused} \in M_{fused}$ **then**
18: $Eliminate_{phi}(n^j_i)$
19: **end if**
20: **end for**
21: $RemoveRedundantArcs(E)$
22: $PrunePredicateArcs(E)$
23: **end for**

[3]In Fig 5(c) fused node $\langle\langle a_{tt}, a_{tf}\rangle, \langle a_o, nop\rangle\rangle$ is routed (named as a_c) and the correct value of a is also stored in a register of PE 4 for later usage.

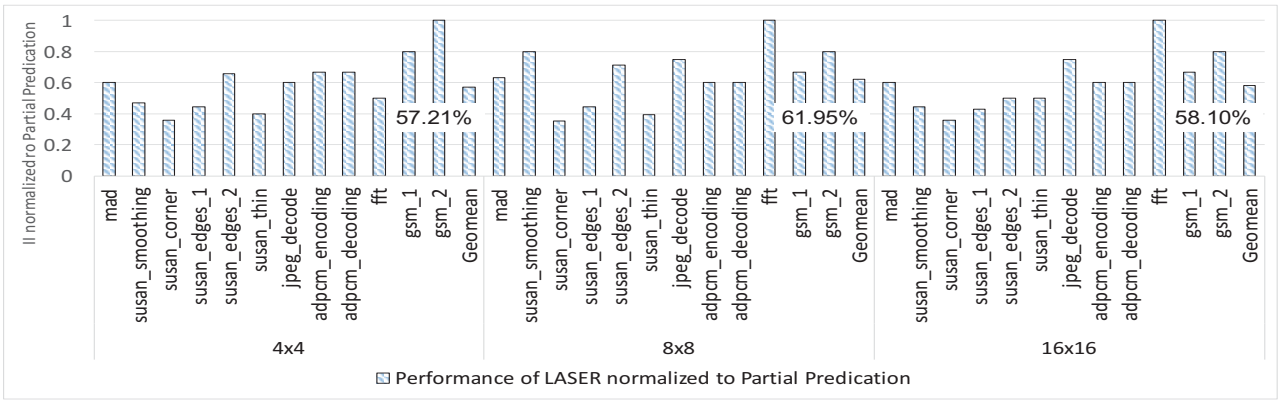


Fig. 6: LASER is a scalable solution with 40.91% cumulative geomean reduction in II compared to partial predication.

dependencies throughout such fusing. After all operations in if-and-else paths are paired for a particular conditional (with any depth), eligible select operations are eliminated via a phi elimination. Then the redundant edges are eliminated and predicate arcs are pruned, which is shown at lines 16-22.

B. LASER – Architecture

LASER-compiler relies on Instruction Fetch Unit (IFU) support to jump to the correct instruction in the instruction memory and issue only those instructions based on the branch outcome. LASER-architecture is shown in Fig 7 which aids in selectively issuing the instructions throughout the loop execution. The IFU keeps track of the all the conditions being evaluated in the loop. Once a PE encounters a conditional

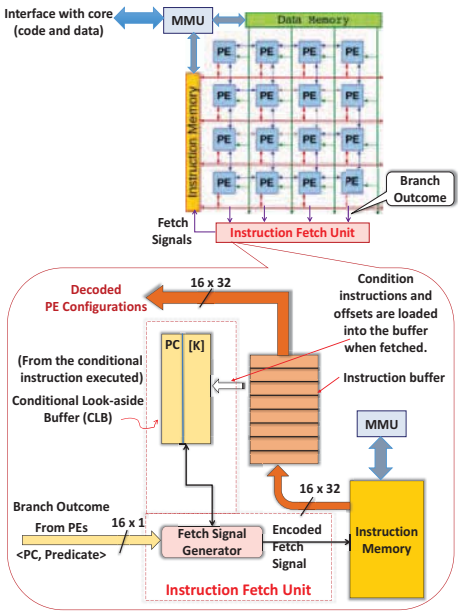


Fig. 7: LASER-Architecture to accelerate complex loops. PEs do not have a predicate network. Branch outcome is communicated to the IFU to issue instructions selectively based on the path taken at runtime.

node and evaluates the outcome, it communicates that to the instruction fetch logic. Based on the information about the latest branch outcome, IFU can lookup in conditional look-aside buffer (CLB) to determine the number of instructions (k) associated with that condition. CLB keeps track of the information about PC of the conditional instruction and corresponding k value. So, if the condition evaluated is false, hardware can look-up for needed k value and IFU skips k instructions. To correctly determine the k_i value, the hardware maintains a state register which gets incremented when a new conditional is evaluated. During execution of the path for a conditional, corresponding cycle counter keeps incrementing by 1. Once the cycle counter reaches the value k_i, it means that all k_i instructions for the path of condition C_i is executed and now it should again execute the instructions from the path of the higher condition nest.

V. Experimental Results

We profiled MiBench and extracted 12 compute-intensive loops which are nested and/or have conditional nest. We implemented LASER-compiler in the DDG construction stage to correctly fuse the nodes of the true and false paths. LASER-compiler can be used with any mapping technique for mapping the nodes onto the CGRA. We compare LASER with partial predication scheme – only viable approach to map loops with nested conditionals. For evaluation, we used REGIMap [5] to map the DDG obtained from LASER and partial predication. PEs perform fixed-point operations with 1-cycle latency and have 4 local registers. The memory bus is shared among PEs in a row. For load and store operations, two instructions are executed, one generates the address and second loads/stores the data.

A. LASER reduces nodes by 43.43%

Partial predication scheme requires three nodes to correctly execute an operation (true and false paths, and a selection) and increases total nodes to be mapped drastically. In Fig 8 the vertical axis denotes the number of nodes normalized to partial predication and the horizontal axis denotes various benchmarks. Due to fusing of nodes and elimination of select

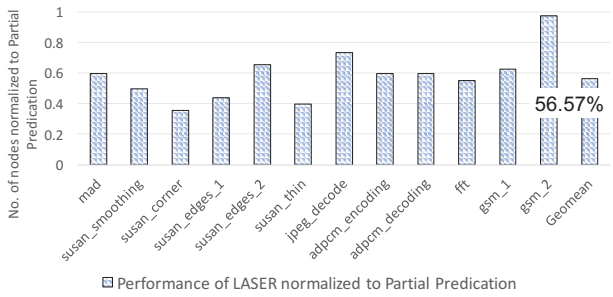


Fig. 8: LASER reduces nodes by 43.43%

operation, LASER reduces the nodes by 43.43%. LASER achieves much better utilization with increase in depth of nested conditionals and with increase in number of operations inside the nests e.g., *susan_corner* has a depth of 24, resulting in the geomean reduction of 64%, but *gsm_2* shows very less reduction, as it has only 2 operations in a conditional.

B. LASER scales better while mapping with 40.91% better geomean performance compared to partial predication.

Fig 6 shows the comparison of II achieved with partial predication and LASER for different CGRA sizes 4×4, 8×8 and 16×16. Compared to partial predication, LASER has a geomean performance improvement of 42.79% on 4×4 CGRA. As the size of CGRA increases to 8×8, the geomean II reduction for LASER was 38.05%, compared to partial predication. For 16×16 CGRA the geomean II reduction is 41.9%. LASER achieves consistent performance improvement with a cumulative geomean reduction of 40.91% across all three configurations of CGRA.

C. LASER reduces energy by 46%

We implemented the RTL model of LASER-architecture shown in Fig 7, and for comparison with partial predication a 4×4 CGRA with predicate network in each PE (Fig 1 including shaded portions) was implemented. Both the models were synthesized in 32nm using RTL compiler. The power is estimated by Cadence RTL power estimation tool. From the power numbers obtained, we estimated the energy consumed (given in [14]) by LASER and partial predication to accelerate

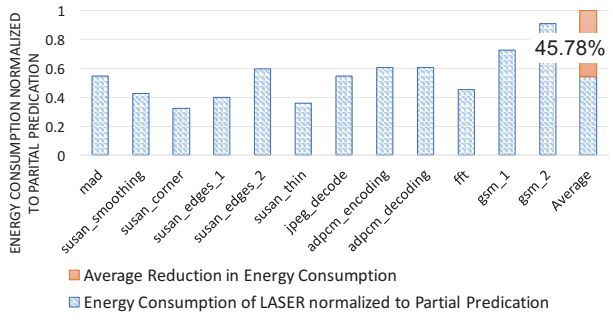


Fig. 9: LASER reduces energy by 46%

the loops of MiBench benchmarks. Energy consumed (nJ) is given by $E = clock\_cycle \times critical\_path\_delay(ns) \times Power(W)$. Fig 9 shows that LASER consumes on an average 45.78% less energy compared to partial predication.

VI. Conclusion

To accelerate general purpose applications with computation bottlenecks as nested loops and nested conditionals, CGRA should behave more like a general purpose modern processor with operationally enhanced IFU, to issue only the correct instruction. State-of-the-art compilers impose a high overhead to accelerate loops with only marginal performance improvement. We have presented LASER, a novel hardware-software approach where, LASER compiler fuses the nodes of various paths of the conditionals, and IFU issues selectively only correct instructions based on the branch outcome. LASER exceeds the state-of-the-art partial predication in accelerating complicated loops efficiently, with 43.43% node reduction and 40.91% better performance.

Acknowledgment

This work was partially supported by funding from National Science Foundation grants CCF 1055094 (CAREER), CNS 1525855 and CCF 1723476.

References

[1] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Designing a coarse-grained reconfigurable architecture for power efficiency. In *DOE NA-22 University Information Technical Interchange Review Meeting*, 2007.

[2] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *PACT*. ACM, 2008.

[3] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: Using epimorphism to map applications on cgras. In *DAC*. IEEE, 2012.

[4] ShriHari RajendranRadhika, Aviral Shrivastava, and Mahdi Hamzeh. Path selection based acceleration of conditionals in cgras. In *DATE*, 2015.

[5] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *DAC*. ACM, 2013.

[6] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Branch-aware loop mapping on cgras. In *DAC*. ACM, 2014.

[7] Navneet Basutkar, Ho Yang, Peng Xue, Kitaek Bae, and Young-Hwan Park. Software-defined dvb-t2 receiver using coarse-grained reconfigurable array processors. In *ICCE*. IEEE, 2013.

[8] B Mei, M Berekovic, and JY Mignolet. Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors. In *Fine-and coarse-grain reconfigurable computing*. Springer, 2007.

[9] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *TACO*, 2013.

[10] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4*. IEEE, 2001.

[11] Yongjoo Kim, Jongeun Lee, Toan X Mai, and Yunheung Paek. Improving performance of nested loops on reconfigurable array processors. *TACO*, 2012.

[12] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Polyhedral model based mapping optimization of loop nests for cgras. In *DAC*. IEEE, 2013.

[13] Jongeun Lee, Seongseok Seo, Hongsik Lee, and Hyeon Uk Sim. Flattening-based mapping of imperfect loop nests for CGRAs. In *CODES+ISSS*. IEEE, 2014.

[14] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *DAC*, page 45. ACM, 2017.