Energy efficient loop mapping techniques for coarse-grained reconfigurable architecture

Vadivel, K.

Award date:
2017

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Energy Efficient Loop Mapping Techniques for Coarse-Grained Reconfigurable Architecture

Master's Thesis

Kanishkan Vadivel

Supervisors:
Prof. dr. ir. Henk Corporaal
Dr. ir. Roel Jordans
Prof. dr. ir. Kees van Barkel

Eindhoven, 31st August 2017

Energy Efficient Loop Mapping Techniques for Coarse-Grained Reconfigurable Architecture

Kanishkan Vadivel
Department of Mathematics and Computer Science, Eindhoven University of Technology
Eindhoven, The Netherlands
Email: k.vadivel@student.tue.nl

***Abstract*—Due to their flexibility and high performance, Coarse Grained Reconfigurable Array (CGRA) are a topic of increasing research interest. CGRAs also have the potential to achieve very high energy efficiency in comparison to other reconfigurable architectures when hardware optimizations are applied. Some of these optimizations are common for more traditional processors but can also lead to large efficiency gains for reconfigurable architectures. On the other hand, good compiler support is essential for CGRA since the applications need to be mapped on architectures for each configuration individually. The execution time of the code generated by the compiler depends more on optimizations it has. This paper investigates three hardware based loop optimization techniques that can significantly improve the energy efficiency of CGRAs and introduces the new optimizations that are added to the compiler backend. The implemented techniques are evaluated on processing kernels from the image processing domain as well as an industrial computer vision application. Energy consumption and area estimates of the hardware extensions are obtained using a CGRA synthesized with a commercial 40nm library. For the three applied hardware techniques (zero-overhead loop accelerator, single-cycle loop support, and loop buffers) the simulation results show gain in geometric mean of the energy reduction is 6.8% for zero-overhead loop support, 13.2% for ZOLA combined with single-cycle loop support and 18.3% for a combination of all optimizations. In parallel to that, added compiler extensions achieve up to 62.8% reduction in application run-time compared to the code generated by the compiler without the extensions.**

***Keywords*-loop support, energy efficiency, reconfigurable architectures, CGRA, codegenration**

I. Introduction

Embedded systems are increasingly becoming mobile and therefore battery powered. The energy in these batteries is often a significant constraint on the device's performance, operating lifetime and functionality. A typical power budget for the processor in a mobile phone is around 1 Watt [1]. This means that the energy efficiency of the processor dictates the functionality that can be made available to the user while still achieving a decent operating period. In order to improve the energy efficiency, computation systems in mobile devices often consist of Heterogeneous System on Chip (HSoC) or a similar system on the board level. These HSoCs typically consist of one or more processors that are connected together via an on-chip network as well as to hardware accelerators. The hardware accelerators perform tasks such as managing the 3G communication and decoding video, and are usually implemented as an application specific piece of hardware.

HSoCs have been the norm for many types of mobile devices since they provide a decent energy efficiency due to the hardware accelerators and flexibility due to the available processors. However, with new communication standards and applications following each other at an increasing rate of inclusion, the fixed function hardware accelerators are being replaced by reconfigurable fabric. An example of a mobile device using reconfigurable hardware as an accelerator is the Google Glass [2]. The trend of including reconfigurable hardware as an accelerator can also be observed in the increased popularity of devices such as the Xilinx Zynq and the Altera SoC. These devices integrate two general purpose processors with an Field Programmable Gate Array (FPGA). The FPGA can be configured to form almost any digital electronic circuit and can therefore be used as a reconfigurable hardware accelerator. This allows designers to perform post manufacturing bug fixes and system upgrades.

General purpose processors provide a high level of flexibility and programmability but lack the required compute performance, energy efficiency, or both. In general purpose processors a large percentage of the energy overhead can be attributed to instruction fetching and decoding, but even more to data movement between memories, caches and register files [3]. FPGAs can avoid much of this type of overhead by allowing spatial mapping of the applications, but their fine grained reconfigurability leads to a high configuration cost. An FPGA is typically reconfigured on the gate level, which gives these devices a high degree of flexibility but also requires a large configurable interconnect. Both size and flexibility of this interconnect lead to long wires, increasing power and lowering the maximum attainable clock frequency, and a high number of configuration bits. A significant contribution to the static and dynamic power dissipation of the device can be attributed to the configuration memory and the interconnect network in an FPGA. Coarse Grained Reconfigurable Arrays (CGRAs) require fewer configuration bits, due to their coarser grained units, which results in a lower energy consumption while still allowing spatial application mapping [4].

Most CGRAs can be seen as reconfigurable processors,with a configurable network that determines the structure of the processor as well as an instruction memory. By supporting spatial layout of applications, CGRAs can often reduce loop-bodies to only a few instructions or even a single instruction.

Despite this, CGRA designers in the past have not optimized the instruction memory hierarchy to the extent that application specific processors such as Very Long Instruction Word (VLIW) or Single Instruction Multiple Data (SIMD) processors are using [5]. Energy reduction techniques used in SIMD and VLIW processors can also be applied on CGRAs. In addition to that, a good compiler is essential to gain maximum performance out of CGRAs when programmability is desired.

The main contributions of this paper are:

- An implementation of three instruction memory hierarchy optimizations on a reference CGRA architecture. Namely: zero-overhead loop support, single-cycle loop support and loop buffers.
- An extension of current CGRA compiler to support newly added hardware feature and for improving schedule quality.
- An evaluation on the impact of these techniques with respect to their use in CGRAs

The paper is organized as follows. Section III introduces the architecture of the reference CGRA and uses an example to illustrate how the loop optimizations can be beneficial for CGRAs. Section IV and Section V introduces the hardware and compiler based loop optimizations, and how they can efficiently be applied to CGRAs platform and its compiler. Section VI then describes how these optimizations will be evaluated and Section VII discusses the energy and area results.

II. Related work

Efficient execution of loops in applications has had a significant amount in research in the past since digital signal processing algorithms typically spend a large fraction of their execution time in loops. The current and past research work in this area can be classified as either Hardware or Software [6] [7] based technique. The hardware based techniques [8] can be further categorized under two groups, namely: zero overhead looping extension and instruction memory hierarchy optimizations. In zero overhead looping, dedicated hardware units are used to automatically update the loop count and to take branch decisions in parallel to normal program execution. Loop buffer based techniques are used to reduce costly instruction memory access.

Support for single loop levels already goes back to very early processor designs. The early x86 processors already included a *loop* instruction which automatically decremented the CX register and branched back to the beginning of the loop if CX still was nonzero. Multi-level loop support has also been added in the past. Such zero overhead loop accelerators were proposed for DSP, RISC, and VLIW processor architectures. The extensions proposed for DSP [9] and RISC [10] are mainly based on two methods: 1) program address based and, 2) instruction count based. In program address based methods, the address of the last and first instructions of the loop body is used as the branch point and branch target address respectively. On the other hand, the number of instructions in the loop body is used to identify branch and target locations in the instruction count based methods. In both methods, nested loop support is provided with the help of stack or scratch pad memory space in the processor. Methods based on distributed address generators were also proposed for VLIW architectures [11]. In a distributed address generation scheme, every issue slot is equipped with a special hardware unit which automatically generates the instruction address, allowing program flow in each slot to be controlled independently. In CGRAs this is not required since processor configurations are made on design time.

Similarly to the loop accelerator approach, single cycle loop support has also been available in processor architectures since their early beginnings. Many architectures include an operation prefix which allows for repeated execution of the tagged operation (similar to the x86 *rep* instruction prefix). That these extensions are useful for signal processing operations is not disputed but information is lacking on how useful they are. Furthermore, combining single cycle loop support with a loop accelerator design can help improve the performance (and especially energy consumption) even more. In this paper, we combine both commonly used techniques in our CGRA architecture and quantify their impact on both the hardware cost and performance.

A third technique frequently used in digital signal processors (DSP), which are often used in a similar context as CGRAs, is to incorporate loop buffers. In DSP processors energy reductions between 25% and 30% are reported for applications such as speech processing prediction algorithms and image compression [12]. Others [13] use knowledge of the application structure to directly control the loop buffers cache controller. Doing so eliminates the need for any runtime prediction of branching behaviour, this can be a significant amount of overhead in architectures like the x86. The authors show a reduction of external instruction memory accesses of almost 38%.

Several loop scheduling techniques were proposed in the past to reduce register pressure for explicit data path architectures, some of which are based on LLVM framework [14]. A target independent Software pipelining is also proposed for LLVM framework in [15] [16], which can be reused in CGRA platform.

III. Background

This section introduces the CGRA architecture and its current compiler framework. Additionally, a programming example will illustrate how the compiler maps a signal processing kernel to this architecture.

A. Architecture

The architecture of the CGRA consists of a host processor and reconfigurable logic [4], as shown in fig. 1. The host processor is responsible for configuring CGRAs and moving application data to and from it using global memory interface. The CGRA configuration data is a bitfile, similar to bitfiles of FPGAs, that configures data paths, control paths, and

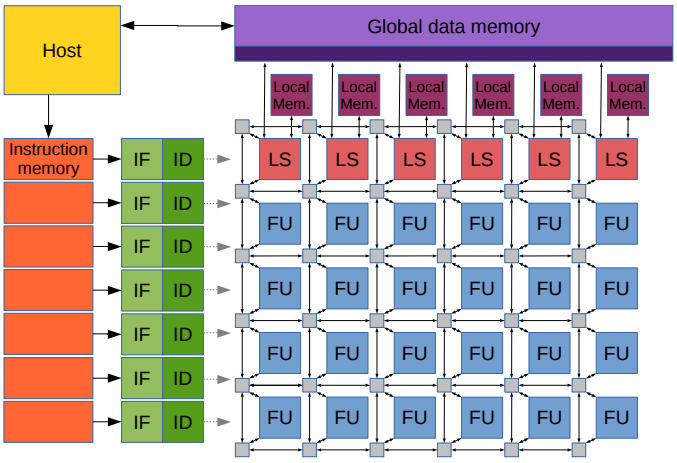


Fig. 1: Architecture overview [4]. *Gray-boxes in the FU array representes the switch-box routings.*

functional unit behavior. The Functional Units (FU) are the heart of the CGRA and can perform computations or memory operations. Examples of such functional units are: Arithmetic Logic Units (ALUs), Register Filess (RFs), Load Store Units (LSUs), Accumulate Branch Units (ABUs) and, Immediate Units (IMMs). The inputs and outputs of FUs are connected to switchbox networks to form a reconfigurable data-paths in CGRAs. These explicit data-paths allows the FUs to directly pass a data between FUs or to the same FU for creating a spatial mapping of an application to achieve high energy efficiency. In this paper we will use the reconfigurable fabric of the architecture as a stand-alone CGRA.

The FUs are controlled by Instruction Fetch and Instruction Decode units (IF/IDs). Each IF/ID has a dedicated instruction memory (IM) from which it reads the instruction during every cycle. The Instruction memory together with IF/ID forms an issue-slot to the processor instance. The IF/IDs control the operations of the FUs, Multiple IF/IDs, each drive a group of FUs forms a VLIW-like processor configuration with extensive bypassing between FUs. For each application the CGRA can be reconfigured in order to form an optimized processor for the application.

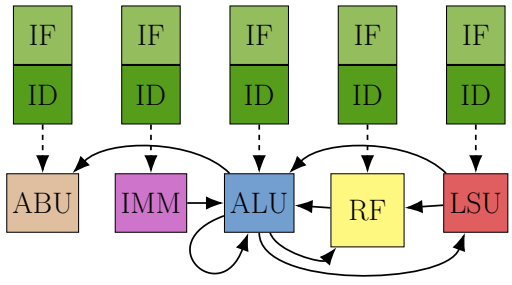


Fig. 2: Sample architecture configuration. *The control and data paths are represented with dotted and solid lines respectively.*

An example CGRA configuration for a minimalistic processor is shown in fig. 2. It contains an IMM unit for generating constant values, an ALU for computation, a LSU for global or local-memory load/store operations, a RF for storing intermediate live variables and, an ABU for computing the program counter (PC) during each cycle and to handling control flow in the program. It can be observed that the outputs are bypassed for some FUs. For example, the output of the LSU is bypassed to the ALU, allowing the result of a load to be used directly (in the next clock cycle) by the ALU.

B. CGRA Programming Example

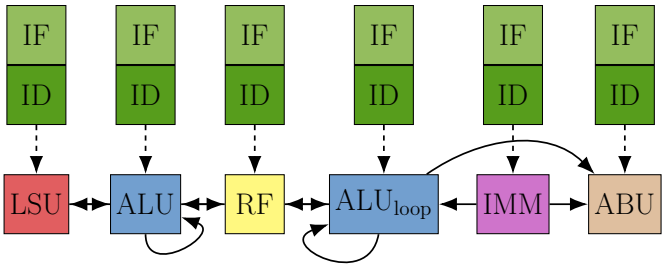
Algorithm 1 Accumulate algorithm

Input: Array A with signal
Output: Accumulated result at B

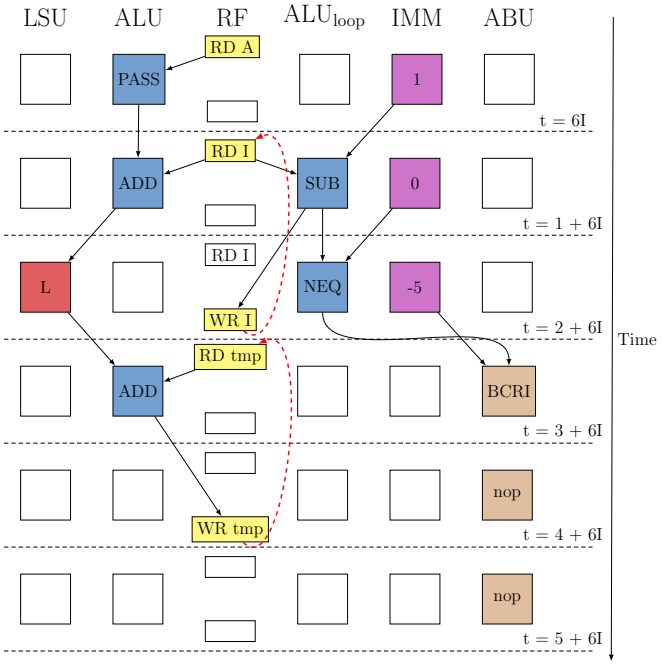
```
1: procedure ACCUMULATE(A[ ], B, Count)
2:     int tmp ← 0
3:     for int I = Count; I != Count; I = I - 1 do
4:         tmp ← (tmp + A[I])
5:     end for
6:     (∗B) ← tmp
7: end procedure
```

The initial version of the CGRA compiler is implemented using the LLVM framework [17]. The compiler uses a Basic block level, operation based, heuristic scheduling algorithm to generate code for the CGRA from high-level language (C-code). The accumulate algorithm is used as an example to illustrate how the CGRA compiler maps an application on to the CGRA. Accumulate is a simple reduction kernel with a single loop at its core, as shown in algorithm 1. The custom CGRA configuration for executing this kernel is shown in fig. 3a. In order to reduce the loop body size, the configuration uses two ALUs, one for updating the loop count (ALU_{loop}) and another for accumulating the data values of the input signal (ALU). It is also possible to execute the application on a single ALU configuration, similar to one provided in fig. 2. Although it saves an issue-slot to the processor configuration, it leads to more instructions in the loop body and, results in increased application run-time and instruction memory accesses.

The schedule for the loop body of the accumulate algorithm on two ALU configuration is given in fig. 3b. For simplicity, the remaining part of the schedule, the schedule of previous and next basic blocks are not shown. The given schedule iterates the loop instructions for N times, where N is the *Count* value from algorithm 1. The labels of the IMM and RF units in the schedule indicate the variables produced by that unit and, the labels on other units represents the operation that is scheduled on those units. The left most two units in the schedule (LSU and ALU) are used for processing the data. It calculates the memory address, loads the data, adds it with temporary variable *tmp* and, store the result back in *tmp* to use it back in next iteration. The right most three units execute the loop control flow computations. It updates the loop index variable (I) and compares it with the loop-bound (*Count*) to



(a) Custom architecture configuration



(b) Scheduled loop kernel instructions. *Solid arrow line indicates (explicit) bypassing and dotted arrow lines indicates a data dependency between successive iterations of the loop. The two cycles following the relative branch instruction (bcri) are scheduled with two 'nop' instructions to avoid pipeline-hazards.*

Fig. 3: Accumulate kernel on the CGRA

generate input for the conditional branch of the loop. In the rest of this paper, the actual data processing computations are referred as core-computation and control flow computations are refered as loop-overhead computations of the loop.

It can be observed from the schedule that the compiler uses an explicit bypassing feature of the platform to improve the energy efficiency of the schedule. However, the current scheduling scope limits its scheduling and bypassing capabilities to Basic-block level. Extending its scheduling scope improves the schedule quality for loop kernels in terms of FU utilization, register pressure and execution time for reducing energy usage. In addition to that, by analyzing and optimizing the energy consumption of individual units in the platform for loop kernels, the energy efficiency of the CGRA platform can be improved further for signal processing applications.

IV. Proposed Architecture Extensions

In this section, three hardware extensions are added to the reference architecture in order to improve the energy efficiency of the CGRA platform, these are:

1) Zero-overhead-loop accelerator
2) Single Cycle Loop Support
3) Loop Buffers

In the *zero-overhead-loop accelerator*, the ALU required to perform loop index calculation, and therefore the branch condition, is replaced by a dedicated custom circuit inside the branch unit (ABU). This saves an issue-slot (including the associated instruction memory) and reduces switching activity caused by the ABU instructions during loop execution. *Single cycle loop support* is implemented in order to avoid repeated instruction fetch and decoding of the same instruction in single instruction loop body. Due to the CGRA's reconfigurability, it is often possible to reduce (parts of) the application to a single instruction that is repeated several iterations. And finally, *loop buffers* are used as an optimization to the instruction memory hierarchy with the aim to reduce instruction fetch cost for repeatedly executing small group instructions that dominates execution time in the application, such as loops.

A. Zero-overhead loop accelerator

As it can be inferred from the example schedule presented in section III-B, an extra functional unit (ALU_{loop}) with access to the register file is required for computing the control flow decisions of the loop. In addition to that, the ABU requires a dedicated instruction in the loop body and uses an IMM unit to trigger branching during every iteration of the loop. Furthermore, the two branch slots in the CGRA cause toggling of the instruction lines controlling the ABU's operation, as shown in fig. 3. The architecture extension, a zero-overhead loop accelerator (ZOLA) is added to allow the removal of the extra ALU (ALU_{loop}) and it's issue slot and to alleviate instruction switching in the loop body. The design for this accelerator is shown in fig. 4. The extensions are added to ABU since it is responsible for control flow operations. The ZOLA allows configuration of loop characteristics such as loop starting/ending instruction addresses and loop iteration counts to be configured. These configuration values are stored in the internal registers of the ABU and allow it to automatically generate the address for loop execution without requiring an external condition or instructions in the loop body.

The ZOLA is enabled using a custom instruction after configuring all loop parameters in the ABU register. Once ZOLA is enabled, it compares the current Program counter (PC) value to the configured loop exit instruction address during each cycle to detect the branch-point. To simplify branch-condition computation to a single bit comparison, the loop-count parameter is initialized with the iteration count of the loop and it is decremented at end of each iteration. The most significant bit, the sign bit, can now be used to detect a branch condition. Once the program reaches the last instruction of the loop (the branch-point), ZOLA replaces the PC with the loop's start address if the sign-bit of the loop-count is zero. Otherwise, it disables the ZOLA and allows normal PC update.

The explicit bypassing feature of the CGRA also allows loop condition to be passed from other FUs with almost zero

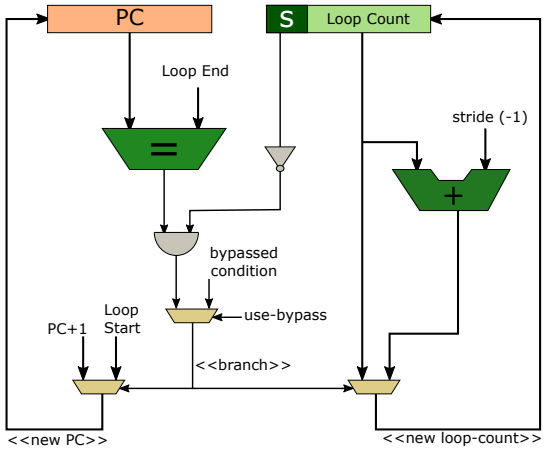


Fig. 4: Zero Overhead Loop Accelerator Architecture. *Program counter is indicated as PC and, sign bit of the loop-count value is indicated with 'S' symbol.*

overhead. This allows ZOLA to support data-dependent loops (e.g. while loops) without any additional overhead.

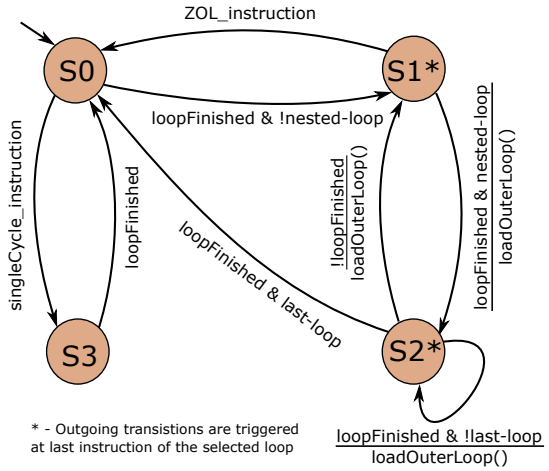


Fig. 5: State diagram of Nested loop support implementation in ZOLA

The accelerator allows arbitrary nested-loop support for up to four levels deep, this is provided by using a state machine to select the relevant loop parameters from the configuration registers. The state diagram of the nested loop support is shown in fig. 5. The states S0, S1, and S2 correspond to nested-loop support. The S3 corresponds to Single cycle loop support which is explained in section IV-B. The custom ZOLA instruction uses start-loop and end-loop IDs as its parameters to execute loops independent of each other. For instance, *"loopr L0, L1"* and *"loopr L2, L3"* will run 2x 2-level nested loops L0–L1 and L2–L3 without affecting each other. The loop-count of the inner loops of a nested loop section would normally be required to be re-initialized during each iteration of its outer loop, which leads to configuration instructions inside the loop body. To eliminate such configuration for the inner-most loop and to keep the hardware cost as low as possible, the updated loop-count of the inner loop is written to a temporary register. The values of these temporary registers are discarded at the end of the loop. Since the original configuration in the ABU configuration register file is unmodified, this preserves the original loop-count of the innermost loop in the register-file for the next outer loop iteration and saves the need for extra ZOLA configuration instructions inside the outer loop.

B. Single cycle loop support

Another large energy saving can be achieved in CGRA by keeping the loop body as static instruction. This is possible when the loop body can be reduced to a single instruction using software pipelining, which is often possible for (parts of) the application due to the reconfigurability of the CGRA. Doing so will repeatedly execute the same instruction over multiple cycles and avoids the need for continuous IF/ID and therefore, accesses to the instruction memory. Additional hardware extensions are required to stall the CGRA instruction fetch and decode pipeline in order to obtain the maximum energy reduction. In the proposed design, the pipeline stall support is extended from ZOLA to reduce instruction-fetch cost and at the same time provide support for efficient execution of static instructions. The state S0 and S3 in fig. 5 correspond to single cycle loop support in the platform. Encoding of the ZOLA instruction (single-cycle or not) differentiates single-cycle loops from multi-instruction loops that are controlled by the ZOLA. This is to minimize configuration overhead for single cycle loops as single-cycle loops require only the loop iteration count as a parameter compared to the multi-cycle loop which requires three parameters.

C. Loop buffers

Caching the repeatedly executed loop instructions in a relatively small buffer, compared to the much larger instruction memory, reduces the IF cost for loop execution. The proposed distributed loop buffer (DLB) organization is illustrated in fig. 6. The DLB uses the ZOLA state variables to automatically buffer the loop instruction, this is possible since the ZOLA parameters effectively specify which part of the application will be repeatedly executed and thus can be stored in the loop buffers. The DLB consists of two units, namely: the loop buffer (LB) and the buffer control logic (BCL). The LB is a simple storage unit with dedicated address lines, data lines and write enable. The BCL is the control unit which generates the control signals for LBs in order to enable or disable buffering of certain (loop) instructions. The LB is instantiated for each IF unit in the configuration and placed in-between IF and ID units. The BCL is placed inside the ABU and its buffer control signals (buffer-enable, write-enable, and buffer-hit/read-enable) are connected to all LB and IFs. This controls where the instruction is read from and consequently passed on to the ID for decoding when there is a request from the application. Integrating the BCL into the ABU allows buffer control signals to be generated at the same time as the PC is updated which allows accessing (read or write) LBs without any stalls in the pipeline.

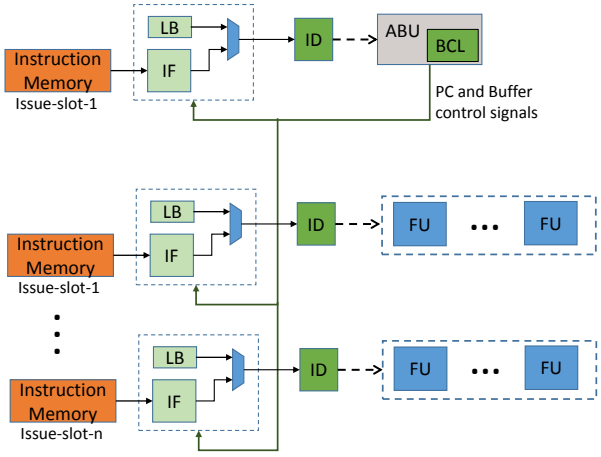


Fig. 6: Loop Buffer Organization. *(BCL-Buffer control logic, LB-Loop Buffer)*

The buffer control logic is designed in such a way that it buffers the most frequently executed loop instructions first (e.g. instructions for the innermost loop) to gain maximum possible benefit from the LB. BCL achieves this by identifying the innermost loop during the first iteration and buffering on next iteration. From the third iteration, the buffered values are used instead of reading operations from the instruction memory when there is a buffer-hit. Once the inner loop is finished, its next level loop instructions are copied to the buffer if there is a free space left without overwriting the instructions for the innermost loop.

V. Compiler Support

In this section, the extensions added to the CGRA compiler are discussed in detail. Furthermore, the application of the proposed methods in the context of loop optimization is demonstrated with example loop kernels.

A. Overview

Four major optimizations are added to the CGRA compiler for improving the loop instruction schedule quality. They are,

1) Register access reduction
2) Hardware loop support
3) Implicit addressing
4) Software pipelining

In phase-1 (Register-access reduction stage), the explicit bypassing capability of the compiler is extended from single basic-block scope to basic-blocks with loop control flow dependencies. This allows explicit bypassing of data between loop iterations and improves resource utilization because of register pressure reduction(as can be observed from fig-3b). The support for hardware extensions proposed in section-IV are implemented at phase-2 (Hardware loop support pass). In phase-3 (Implicit addressing pass), the support for automatic address generation on the LSU for linear memory accesses are added to the compiler. Finally, in the last phase (Software pipelining), the loop instructions of the inner-loops are software pipelined to improve the throughput of the loop iterations.

The deployment view of the CGRA compiler is shown in fig-7. The register access reduction, also referred as the bypass optimization, is added to the scheduling stage. The remaining phases are added as a separate passes in the LLVM framework. The software pipelining pass requires loop control/data flow analysis and loop transformations, which are same as the implementation of hardware loop pass. Hence, it is used in combination with hardware loop pass. The machine code optimization stage is a combination of LLVM's built-in passes such as Dead Code Elimination (DCE), Loop Invariant Code Movement(LICM), Common Subexpression Elimination (CSE) and Machine Sink (MS) passes for removing computation(dead instructions) that are no loger required because of hardware-loop and implicit-addressing instructions. The dead Phi-Elimination pass is a custom pass to handle dead phi-nodes since the inbuild DCE pass will not handle phi-nodes with dead instructions as its operands. In addition to the custom passes, the existing stages in the backend are also modified to support custom and inbuilt optimization stages.

B. Phase-1: Register access reduction

The main goal of Register access reduction is to provide scheduler independent explicit bypassing support for basic-blocks with loop control-flow dependencies to address the problem discussed in section III-B. The targeted loop control flow pattern, a loop with a single entry and exit block, is given in fig-8b. During loop execution, the latest result produced by the FUs is available on their output ports at the beginning of the next iteration, or at the entry point of the next basic block. The process used to guide the scheduler for explicit bypassing of this data, without affecting data and control flow dependencies of the scheduling pass, is shown in fig-8a. Initially, the loop kernel is scheduled with the normal scheduling process implemeted in [17] and then the results that are available for bypassing are identified using bottom-up scanning of the original schedule.

In the CGRA architecture, bypassing is explicitly controlled by software, which means each operand should come from an unique location (RF or from an other bypassing source). Hence, additional configuration instructions are inserted at the loop entry block for pre-loading the data from the RF to the corresponding buffer port. To avoid schedule conflicts (read-before-write and invalid buffering path) that may arise while scheduling the loop with buffer instructions, a schedulability test is carried out for the buffer instructions prior to actual insertion. In addition to that, a barrier chain is added in the dependency graphs to force buffering after the actual instruction scheduling.

By moving these buffered operand's source location from the RF to a buffering port, and repeating the scheduling process for the basic-block will automatically allow the scheduler to use bypassing between basic blocks. However, relocating the input source operands results in a new schedule which may not buffer the same results as the schedulability test. In addition to that, the bypassed results need not be stored in the RF during each iteration. Therefore, additional

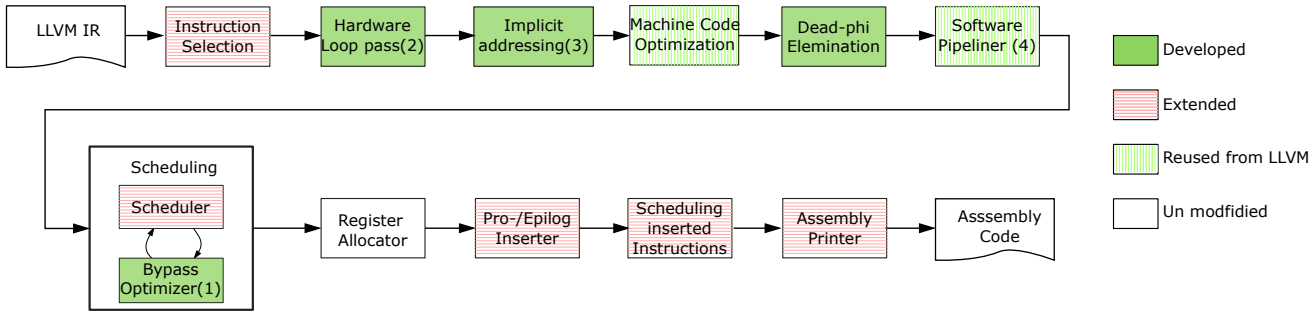


Fig. 7: Deployment view of the CGRA compiler backend. *The phase in which the optimizations are implimented are indicated with the numbers in the brackets.*

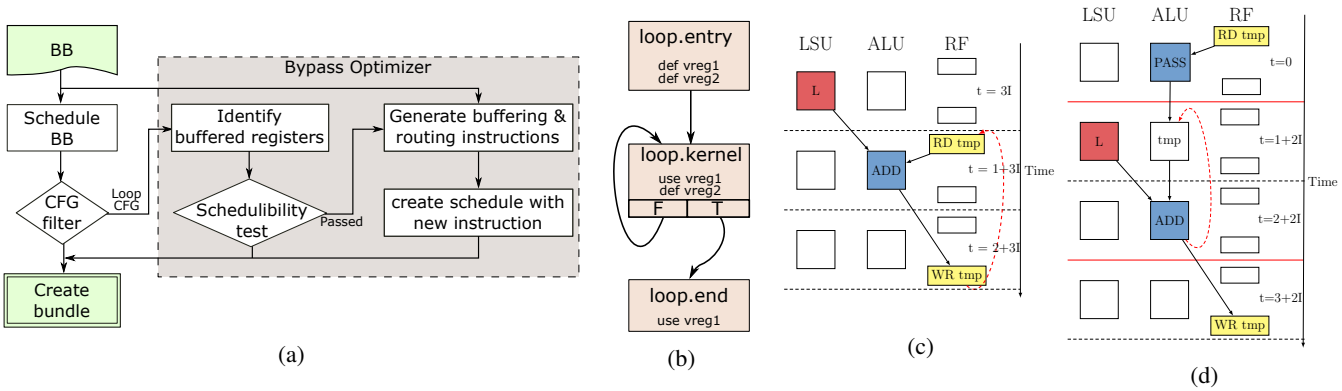


Fig. 8: Overview of the register access reduction technique. *The process view and the loop pattern supported by the current bypass optimizer is given in* (a) *and* (b) *respectively. The intial schedule for the example loop kernel is given in* (c). (d) *shows the schedule generated by the optimizer. The resultant loop kernel is highlighted with solid red lines and the bypassed register is indicated with a dotted arrow line.*

instructions are inserted in the current basic block before and after the loop branching point for moveing the results to the proper buffering port and storing the buffered results to the RF once the loop is terminated.

The original schedule, and the schedule generated by register access reduction pass for the example loop kernel is shown in fig-8c and 8d respectively(assuming that the CGRA configuration is not having any restriction on bypass connectivity). The original schedule takes 3-cycles/iteration. Usage of the register access reduction pass bypasses the operand *tmp* and results in two cycle loop kernel as indicated with solid red lines in the figure. The proposed optimization pass can also be easily extended to bypassing data between basic-blocks with other control dependencies and function calls.

C. Phase-2: Hardware loop support

Hardware loop support enables the compiler to make use of the hardware extensions discussed in section-IV. The hardware-loop pass is implemented in a two-step process. The first step identifies the loops that are suitable for hardware looping, and it extracts the loop parameters such as loop entry block, exit block and loop-count using the custom developed target specific analysis functions. In this stage, initially, the loops with analyzable control flow dependencies are taken

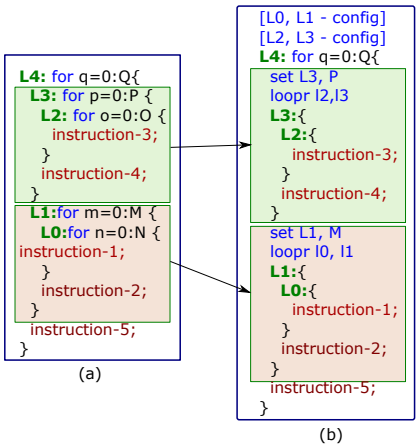


Fig. 9: Illustrative example for hardware loop pass. *The pseudo code of the original loop and the output of hardware loop pass are shown in* (a) *and* (b) *respectively. Hardware loop instructions are highlighted with bule color.*

as eligible candidates for hardware looping, and they are converted to a standard loop pattern: a loop with single entry and exit blocks as shown in fig-8b. Then, by taking the loop branching instruction as a reference point, the loop index variable and its update computations are identified by the instruction analysis on the data flow graph (DGF). The

analysis starts with a loop branch condition and ends with phi nodes, which select data based on their predecessor block. Since the phi node is ensured to have cyclic control flow, the loop index and its starting value (immediate/register) can be directly identified. However, to avoid loop index computation during each cycle, ZOLA requires the loop trip-count to be configured in the ABU registers before entering the loop. In order to calculate the loop trip-count from the initial loop index variable and its DFG, the loop overhead computations are simulated inside the pass if the DFG is composed of analyzable instructions (instructions without any external symbols as operands). The ZOLA is configured to use the identified trip-count if the simulation succesful. Otherwise, a normal branch instruction is used.

In step-2, the loop configuration instructions (loop start address, loop end address and loop-count configurations) and hardware-loop trigger instructions (loopr) are inserted at proper locations before the loop. The ZOLA requires the loop address locations to be configured before the hardware loop starting block and, the loop-count to be reloaded for nested loops on its pre-header during each iteration of its outer loop. The inner-most loop in the nested hardware looping is the exception to this, because of the reason described in section-IV-A. Hence, the inner-most loop-count is configured at the same block as the loop address is configured. The fig-9 shows the configuration instructions generated by the hardware loop pass for an example nested loop section.

The subsequent passes (software pipelining and implicit addressing) in the CGRA compiler require information about the hardware looping for branch analysis. In addition to that, the software pipelining pass relies on the loop-count information generated by the hardware loop pass in a specific format (loop-count and branching point in single instruction). Furthermore, the ZOLA requires the address of the last instruction of the loop to be pre-configured. However, the LLVM framework does not allow custom labels inside the basic blocks (because of the strict basic-block property). Hence, pseudo loop-trigger instructions are inserted in the hardware loop pass, and they are replaced with actual hardware loop instructions before printing it as assembly code. To make sure that the most frequently executed loops are buffered, the loops in the complex nested loop section are selected based on their loop depth, a number of instructions (smaller loops have higher priority) in the loop body and the branch probability details obtained from loop analysis pass.

D. Phase-3: Implicit addressing

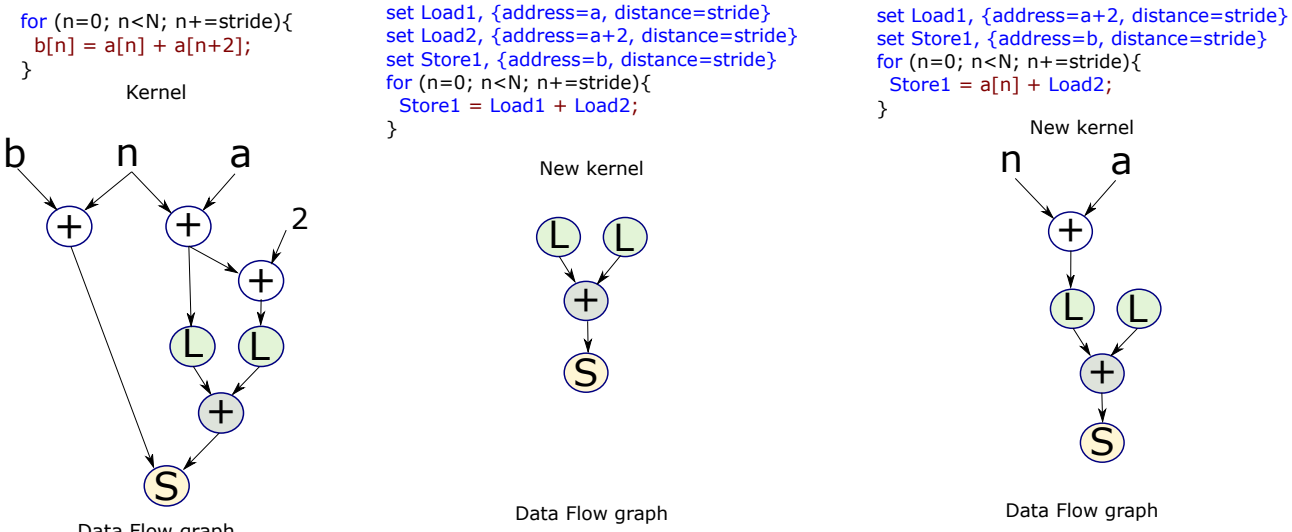
The linear memory access address generation computations are optimized to effectively use the CGRA hardware support in the Implicit addressing. The sample loop kernel with a linear memory access pattern and its corresponding address calculation (data flow) generated by the current compiler is shown in fig-10a. During each iteration, the new addresses are calculated by adding an initial memory address to the stride which requires extra computations. The CGRA platform supports implicit hardware addressing, where the stride and access distance can be pre-configured for linear memory accesses to avoid explicit address generation. Extending the compiler to support this feature requires an analysis pass for starting memory address, and access-distance identification. With some modifications, the algorithm described in Section V-C can be reused to achieve this goal.

The number of linear load/store operations that can be converted to use an implicit feature of the platform depends on the number of LSU available in the configuration. In addition to that, the configurations (starting address and access distance) generated for the implicit instructions should be mapped on the same LSU where the implicit instructions are scheduled. Thus, custom LSU resource map and instruction bindings are implemented to guide the scheduling process for the implicit instructions. Fig-10b shows the code generated for two LSU configuration by the implicit addressing pass for the kernel. The pass identifies the linear memory access and converts all of them to implicit instructions and generates configuration instructions on the loop pre-header since there are sufficient resources. Under resource constraint, compute intensive address calculations are converted to the implicit instructions. Mapping the computation shown in fig-10a on a single LSU configuration results in the code shown in fig-10c. The pass selects the instruction based on the tree height of DFG to replace the compute intensive address calculations.

E. Phase-4: Software Pipelining

Software pipelining is a loop scheduling technique which overlaps the instructions from successive iterations of the loop in the schedule to increase instruction level parallelism. Modulo scheduling is the most common technique for implementing software pipelining, a swing-modulo-scheduler (SMS) is integrated into the CGRA compiler because of two main reasons: 1. SMS can produces an optimal software pipelined schedule in terms of schedule time and register pressure [18]. 2. The SMS is already implemented as a target independent compiler backend pass in LLVM for the Hexagon architecture [16] and it can be reused for the CGRA compiler with some modifications.

In SMS, the operations in the loop body are overlapped such that there is a fixed initiation interval (II) between the start of the consecutive loop iterations, which is smaller than the total length of the original loop. The minimum II of the loop is constrained by the loop carried dependencies and available resources. The reader is referred to [7] for implementation details of SMS. To use LLVMs software pipelining pass in the CGRA compiler, the resource and instruction model of the backend is re-designed using the TargetItineary class of LLVM. In addition to that, several machine instruction analysis functionalities such as branch analysis and hardware loop analysis are also added to the backend to support the SMS pass. The software pipelining pass generates an approximate schedule and actual instruction scheduling is carried out in scheduling stage as shown in fig-7. Hence, the TargetItineary is defined in such a way that it expresses the approximate resource model of the CGRA instance(a resource model

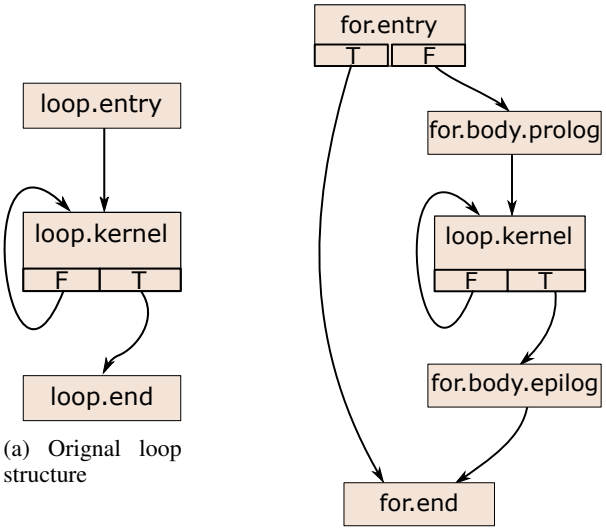


(a) Orignal address computation

(b) Implicit addressing pass output on CGRA configuration with two LSU resources

(c) Implicit addressing pass output on CGRA configuration with one LSU resource

Fig. 10: Illustrative example for Implicit addressing pass. *In* (b)*, the pass converts all load and store instructions to implicit mode since there are sufficient resources. Under resource constraint, the pass replaces the compute intensive address calculations with implicit operations as shown in* (c)*.*



(a) Orignal loop structure

(b) Software pipelined loop

Fig. 11: Code-generation with software pipelining

without bypass configuration details) for avoiding unnecessary restrictions on SMS schedule. The loop patern generated by the SMS pass is shown in fig-11b. Even though the SMS pass is software pipelining the code, the basic-block scheduling scope increases the schedule length of the kernel due to increased register pressure between the prolog and epilog sections. Hence, a bypass optimization pass is necessay to alleviate this effect, especially for smaller loop kernels.

VI. Experimental Setup

Experimental setup used for evaluating the CGRA hardware and compiler extensions are discussed in this section.

A. CGRA Platform

A set of six image processing kernels namely Binarization, Erosion, FFoS, FFT, IIR and Projection are used in the baseline setup to identify the possibilities for improving the energy efficiency of the CGRA at the architecture level. The kernels are chosen in such a way that it expresses most common cases of signal processing application [19]. FFoS is an image processing application developed for an industrial setting where the centers of OLED pixels have to be detected. This application uses the binarization and erosion kernels as well as performing vertical and horizontal projection. The optimal CGRA configuration (issue slots, vector units, and bypasses) for each kernel is identified manually from the source code and then the applications are mapped to it. The assembly code for the CGRA is hand-written in order to ensure the best possible performance out of the platform.

To analyze the energy consumption of individual functional units in the CGRA, the design is synthesized for each application and simulated for a commercially available 40nm ASIC library. The energy and area values of the CGRA logic (everything except memory modules) presented in the rest of the paper are based on post-synthesis simulation results of the kernels. The energy spent on memory modules such as the instruction memory, global (data) memory and local (data) memory are calculated from the datasheet of the 40nm commercial low-power memory module with the following configurations,

- **Instruction Memory (per issue slot)** - 256 rows, with one read and one write port of width 12-bit.
- **Global Memory** - 32KB memory, with one read and one write port of width 32-bit

- **Local Memory** - 1KB memory with one read-write port of width 32-bit.

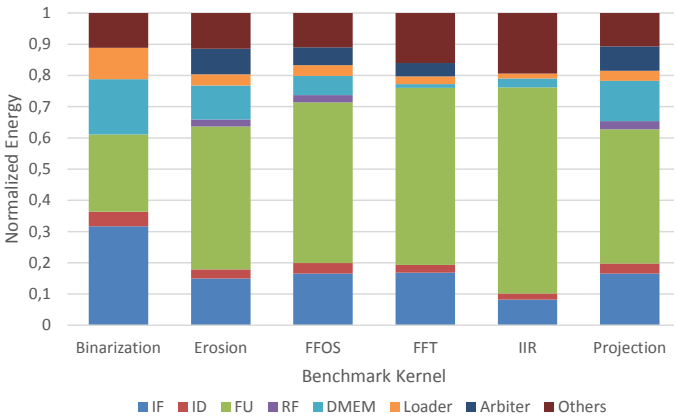


Fig. 12: Normalized Energy breakdown for benchmarks(base-line)

The fig. 12 shows the normalized energy breakdown for the individual benchmark kernels. The energy spent in the functional units (e.g. ALU, MUL and RF) represents the amount of energy used for performing computation and register-file usage for executing the application. It can be observed that the RF is not used for some kernels. This is because of the explicit-bypassing feature of the platform which can handle live variables in the bypass network without the need for a register file. The control path energy is specified under the 'IF' and 'ID' sections. The data memory access cost and its arbiter (multiple load and/or store requests to global memory are handled by an arbiter) costs are listed as 'DMEM' and 'Arbiter'. The 'Loader' and 'Other' groups account for the energy spent on (re-)configuring the CGRA platform and, data-bus, bypass, and control signals of the configuration.

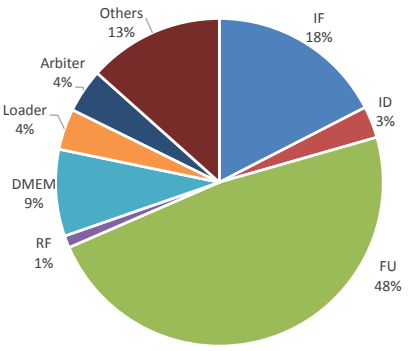


Fig. 13: Geometric mean of the base-line energy breakdown for the benchmark applications

The geometric mean of the benchmarks is shown in fig. 13. It can be seen that the instruction fetching, and to a lesser extend, instruction decoding account for a significant portion of the total energy. On the other hand, the application is mapped spatially on CGRA in a highly optimized way in terms of processor configuration (vector units, issue-slots and explicit bypasses) and instruction scheduling [17], which leads to less room for improvement in the 'FU', 'DMEM' and 'RF' categories. In addition to that, the 'Loader' and 'Misc' groups are fixed components of the CGRA architecture and cannot easily be altered. Hence, one of the most interesting places for energy improvement is the control path which is composed of the IF and ID units. In general, loops are the hot-spots in most signal processing applications, which narrows down our scope further to optimizing the control path for the loops in order to improve overall energy efficiency of the application running on CGRA.

B. CGRA compiler

The benchmarks used to test the compiler are given in table III. The benchmark set is limited because of the absence of compiler support for memory functionalities (memory transfer instructions), function calls, register size and, variable amount shifts and select operations. A common configuration, a 10-issue VLIW configuration with three ALUs, two LSUs, and two Multiplier Unit (MUL) with ABU, RF and IMM unit is used to evaluate the result of each extension added to the backend. The results of the compiler with new extensions are verified by running the generated assembly code on the synthesized CGRA platform and comparing the result with original compiler generated results. The extensions to the compiler are evaluated in terms of overall execution time, register pressure, and compilation time.

TABLE I: CGRA compiler benchmark set

Benchmark	Basic blocks	Number of loops	Loop depths
Binarization	3	1	{1}
Histogram	4	2	{1, 1}
FIR	5	1	{2}
Projection	5	1	{2}
2D Convolution	9	1	{4}

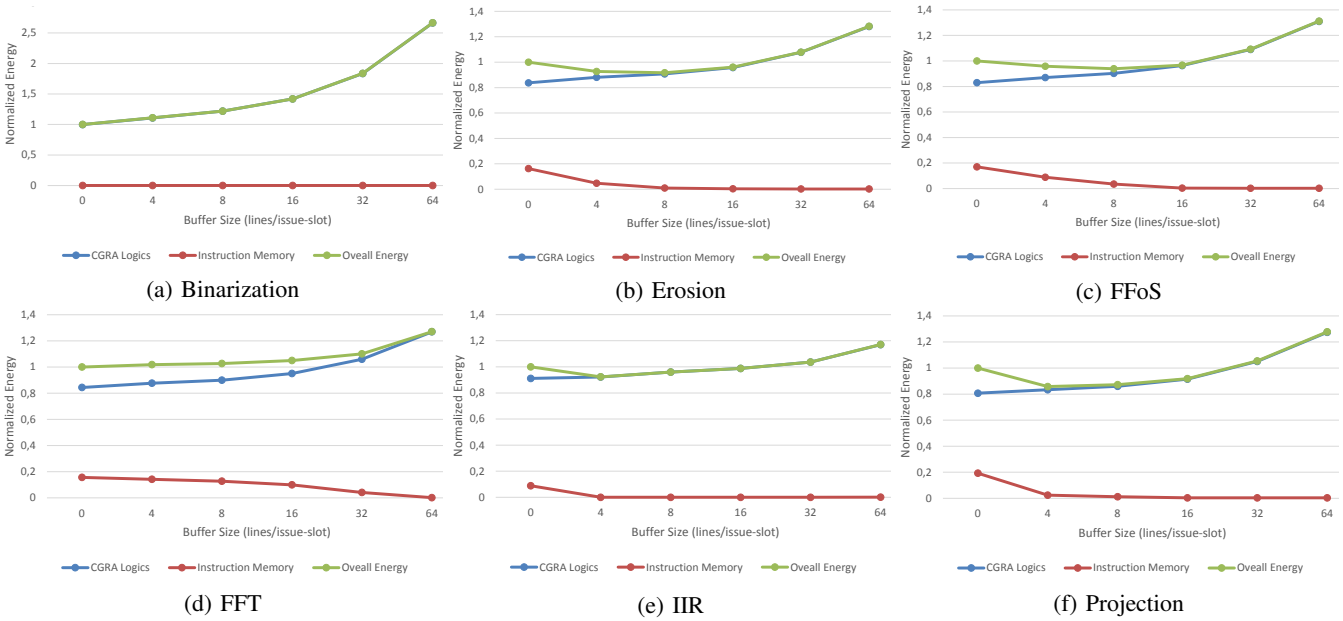
VII. Evaluation

In this section, the evaluation and results for the three evaluated hardware accelerators and the compiler extensions will be discussed. Since some optimizations depend on each other they will be discussed in their required combination.

A. Zero Overhead Loop Accelerator and Single Cycle Loop

The results of the baseline setup show that around 21% of energy is spent on instruction fetch and decode. Using the zero-overhead-loop accelerator discussed in section IV-A and replacing an issue slot that handles loop computations with a dedicated circuit could save up to 21% for applications that can be reduced to a single-cycle loop. However, for some applications, it is not possible to remove an issue slot since the FU which handles loop control flow computations might also be used for other computations.

For ZOLA to provide a gain in efficiency, the energy overhead from ZOLA should be lower than the energy savings for the application. In some applications, the loop calculating



(a) Binarization (b) Erosion (c) FFoS

(d) FFT (e) IIR (f) Projection

Fig. 14: Effect of buffer size on individual benchmarks

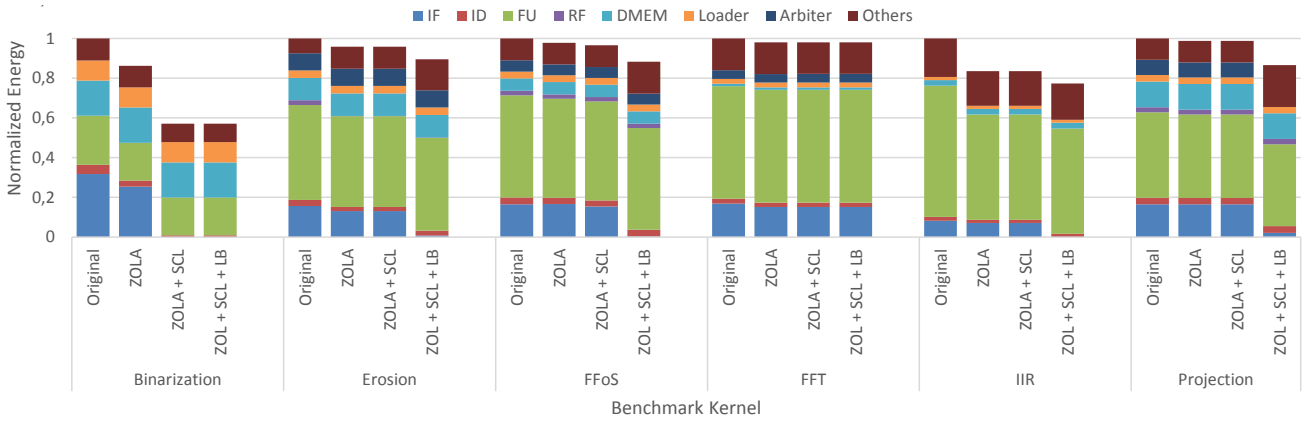


Fig. 15: Normalized Energy comparison of proposed methods, SCL means: single cycle loop support. *Note: Loop buffers are not used for the FFT and Binarization kernel*

ALU might not be removed from configuration (e.g. where it is used for other computations as well). In such cases, the gain from ZOLA is only through the reduction of control flow computations on FU and register accesses. And therefore, the overhead of the ZOLA should be lower than the ALU operation cost. By combining ZOLA with single cycle loop support, maximum energy saving can be achieved for static loops in the application since in that case the IF/ID can be disabled completely. In addition to that, having a static instruction eliminates the instruction switching on FUs and leads to energy efficient execution. The (re-)configuration feature of CGRA allows hardware to be configured in such a way that it enables single cycle loops in an application. However, for larger kernels this might not be feasible because of hardware cost. Hence it is important to keep the overhead of single-cycle loop control low, as it can not always be used.

B. Loop Buffer Tuning

Since the size of loop buffers is application dependent the application should be profiled to extract the ideal buffer size. The effect of buffer size on the energy for individual kernels is shown in section VII. The binarization application has a static loop as its kernel and hence having a buffer will not cache any loop instructions and degrades the energy efficiency as shown in 14a. The buffer size effect on erosion kernel is given in fig. 14b. The erosion kernel has an inner loop of 6 instructions long, hence moving from 4-line to 8-line buffer caches almost all loop instructions in the application as can be observed from the instruction memory energy usage in fig. 14b. Any increase in buffer size thereafter only adds overhead. Hence it can be concluded that the 8-lines are the optimal buffer size for this kernel. The FFoS kernel has one single-cycle loop and several (nested) loops. The hot-spot of the FFoS kernel is a 2-level

nested loop with a loop body of 10 instructions long. Hence, the overhead when moving from 8-line to 16-line is higher compared to the gain. So the optimal buffer size is 8lines. The FFT kernel has a single loop with 43-instructions in it. Since the loop iterations are low (248 iterations in total) for this kernel, buffering the loop instructions will not improve the performance because of insufficient temporal locality as can be observed from the instruction memory access cost in 14d. Hence, the loop buffer is not used for this kernel. The IIR and Projection kernels have same loop pattern as of Erosion and same can be observed in fig-14e and 14f.

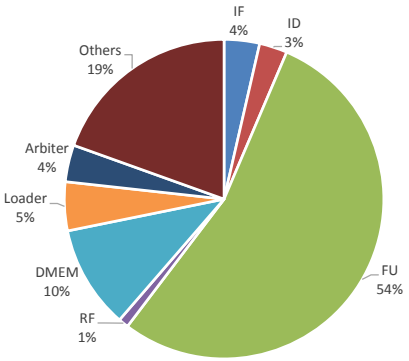


Fig. 16: Geometric mean of energy breakdown for the benchmark applications with proposed hardware extensions

TABLE II: Energy-Area trade-off of proposed methods. *Normalized energy and area values corresponds to the CGRA logics execulding global and local data memory.*

Kernel	Configuration	IMem Accesses	Normalized Energy	Normalized Area
Binarization	Original	8212	1	1
	With ZOLA	8210	0.86	0.91
	With SCL	14	0.57	0.91
Erosion	Original	8982	1	1
	With ZOLA	8990	0.95	0.95
	With LB(8-line)	513	0.89	0.98
FFoS	Original	14232	1	1
	With ZOLA	14248	0.97	0.99
	With SCL	13230	0.96	0.99
	With SCL+LB −(8-line)	296	0.88	1.06
FFT	Original	10639	1	1
	With ZOLA	10642	0.98	0.99
IIR	Original	6630	1	1
	With ZOLA	6632	0.83	0.99
	With LB(4-line)	38	0.77	1.02
Projection	Original	4655	1	1
	With ZOLA	4667	0.98	1.01
	With LB(4-line)	595	0.86	1.05

C. Results

The design-time configuration of the zero overhead loop accelerator and single cycle loop support do not depend on the application properties, whereas the ideal size of the loop buffer depends on the application. In the comparison that follows, the ideal loop buffer sizes (rounded to a power of two) are used.

The figure 15 shows the comparison of energy savings achieved by the proposed optimizations on the individual benchmarks. The fig-16 and table II summarizes the energy gain and area overhead for the individual methods. The ZOLA improves the overall energy efficiency for all the benchmark set that were considered with an area overhead of 1%. As can be observed from table II, the energy saving on FUs is lower for FFoS and Projection kernel compared to others. This is because the issue-slot that previously performed loop condition calculation cannot be removed as it is also used for other computations. However, the energy dissipated in the FUs and RF are reduced because the control flow is now performed by a dedicated unit and results in an overall energy saving.

The highest achieved energy saving of almost 43% is achieved for binarization when using steady state loop support combined with ZOLA. The main energy saving comes from the reduction of energy consumption in the IF and ID. The FFoS kernel has one steady state loop where 7% of the execution time is spent, which explains the small reduction in IF energy over ZOLA in the comparison graph when single cycle loop support is enabled. Adding single cycle loop support does not provide any gains for remaining kernel since it does not have any steady state loops. However, the overhead of steady state loop support is very minimal and therefore energy stays constant.

The binarization kernel does not benefit from a loop buffer since its only loop is a single cycle loop. Insufficient temporal locality limits FFT kernel to benefit from loop buffers. Over 97% of the instructions in erosion, FFoS, IIR and Projection are the loop instructions. Therefore with optimal buffer sizes, the IF energy is reduced by 6.6%, 8.6%, 7.4% and 12.3% for erosion, FFoS, IIR, and Projection respectively compared to ZOLA + single cycle support. The overhead added by the loop optimization hardware is the highest for in FFoS and projection since it uses 16-line and 8-line buffers respectively for its 7-issue processor configuration compared to the erosion and IIR which uses 16-line and 4-line buffer for the 6-issue processor configuration. Even though projection uses 4-line buffer, the area cost is comparable to erosion which uses 16-line buffer for the same issue-width. This is because of difference in SIMD lane width. The erosion uses more FUs than projection, which leads to the smaller difference in relative percentage increase.

D. Compiler Extensions

The hardware loop support(HLS) reduces the loop overhead computations. However, some loop index variables are also used for linear address generations in the benchmarks. Hence, using hardware and implicit addressing passes as a standalone pass will not improve the schedule quality (execution time and register pressure) to the same level as that of their combined use. Software pipelining the original code produces a schedule with high register pressure and operation count. Therefore, the software pipelining pass is used in combination with hardware loop support and implicit addressing passes. Even though the software pipelining breaks the dependency between the

operations in the innermost loop, it increases register pressure significantly for smaller loops. The effect can be reduced with the bypass optimization pass as a post-processing stage (after the actual scheduling).

The inner-most loop is the most frequently executed code section. Hence, the comparison of the register pressure in the inner-most loop for the implemented optimizations is shown in fig-17. Using HLS and IA passes provides higher gain compared to their separate use as expected. However, the binarization and projection kernels achieve higher gain with implicit addressing compared to hardware loop support. This is because of the two linear address calculations in binarization and, register accesses in the memory initialization loop is completely eliminated in projection kernel. The histogram kernel does not have any loop-carried data dependencies when implicit addressing and hardware loop support are used. Therefore, bypass-optimization will not have any effect on its schedule. In addition to that, the current memory dependency analysis used by the software pipelining does not support complex address analysis which limits the histogram kernel to be pipelined and results in the same register pressure as of HLS+IA.

The overall loop execution time comparison of different optimization stages are shown in fig-18. The hardware loop support, implicit addressing, and their combined use has the same behavior as expected and explained before. The bypass optimization does not improve the execution time for binarization and histogram since there is no loop-carried data dependencies. On the other hand, more gain is achieved for projection since most of the variables in its inner-most loop can be bypassed. The software pipelining pass increases the execution time for all the benchmarks because of the newly introduced register pressure in the inner-loop. However, with bypass optimization, the execution time reduces significantly since there are no intra-iteration dependencies. The overall execution time for 2D convolution and FIR remain high compared to the optimization without software pipelining. This is because the generated prolog and epilog sections schedule length (execution time) is higher compared to that of the reduction by software pipelining. The tradeoff of the implemented optimizations is shown in Table III. With a trade of doubling the compile time, the implemented optimization eleminates 88% of register accesses in the inner-most loop and improves overall loop execution time by 62.8%.

TABLE III: CGRA compiler optimization tradeoff in terms of Geomentric Mean. *Note: Register access values corresponds to inner-most loop of the kernel. Execution and compile time corresponds to the entire application.*

Benchmark	Register accesses	Exeuction time	Compile time
Original	1	1	1
Hardware loop support(HLS)	0.86	0.845	1.14
Implicit address-ing(IA)	0.74	0.779	1.12
HLS + IA	0.21	0.417	1.20
HLS + IA + by-pass optimization	0.12	0.377	1.36
Software Pipelin-ing(SP)	0.62	0.712	1.88
SP + bypass opti-mization	0.12	0.372	2.01

VIII. Conclusions

Instruction fetching and decoding represents a significant part of the energy consumption in CGRAs. In order to reduce this type of overhead this paper discusses and evaluates three hardware optimizations that aim to reduce the cost of the IF and ID stages. These three methods are: zero-overhead loop support, single cycle loop support, and loop buffers. Results are shown for three benchmark applications and a variety of CGRA configurations. As can be observed in section section VII these optimizations, or combinations thereof can have a significant impact on the energy efficiency of the architecture. This paper shows that the geometric mean of the energy reduction is 6.8% for zero-overhead loop support, 13.2% for ZOLA combined with single-cycle loop support and 18.3% for a combination of all optimizations. Of course, such hardware additions come at a cost in area. The area increase for the three optimizations are canceled out (for 3 out of 5 kernels) by the removal of hardware that is no longer required, such as an extra ALU issue slot. For the remaining kernels, the area increase is between 1% and 6%. A paper about the work has been submitted for DSD2017 [20].

The basic block scheduling scope and lacking loop optimizations of result in inefficient assembly code schedule in the current compiler. Adding compiler support for hardware extensions and implicit address feature replaces the expensive computations in the loop with low-cost domain specific instructions for reducing execution time and register pressure. As can be observed in section section VII-D these optimization reduces the execution time and register pressure by 58.3% and 79% respectively. Integrating loop scheduling(software pipelining) and loop code specific optimizations(bypass optimization) on top of that results in overall reduction of 62.8% and 88% in execution time and register pressures respectively.

This paper shows that CGRAs that are optimized for energy efficiency with good compiler support for high-level programming can be a key player in the search for energy efficient mobile compute devices. The hardware and compiler optimizations discussed in this paper will be integrated in the energy efficient CGRA architecture that our group is developing.

References

[1] C. Van Berkel, "Multi-core for mobile phones," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009.

[2] TechInsights Inc. (2014) Google glass teardown. [Online]. Available: http://www.techinsights.com/about-techinsights/overview/blog/google-glass-teardown

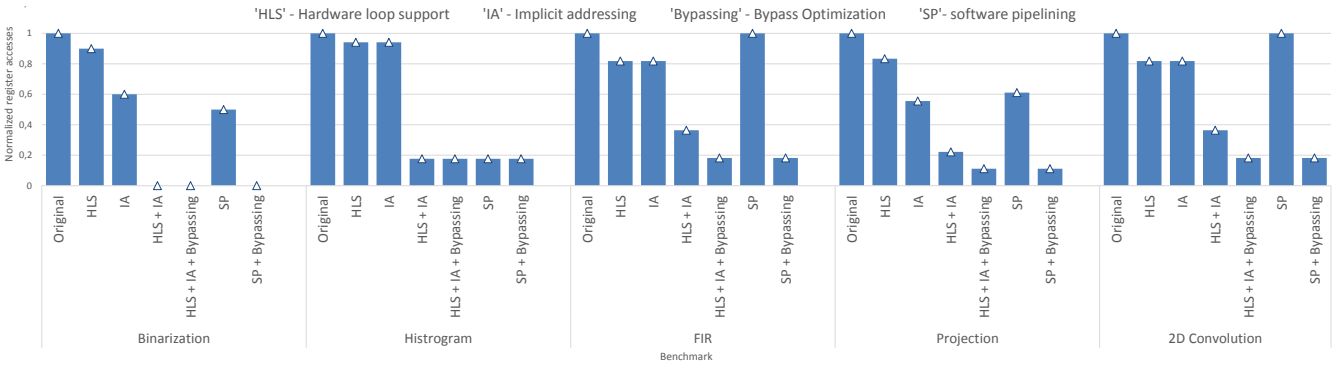


Fig. 17: Inner-most loop's normalized register access comparison of compiler optimizations.

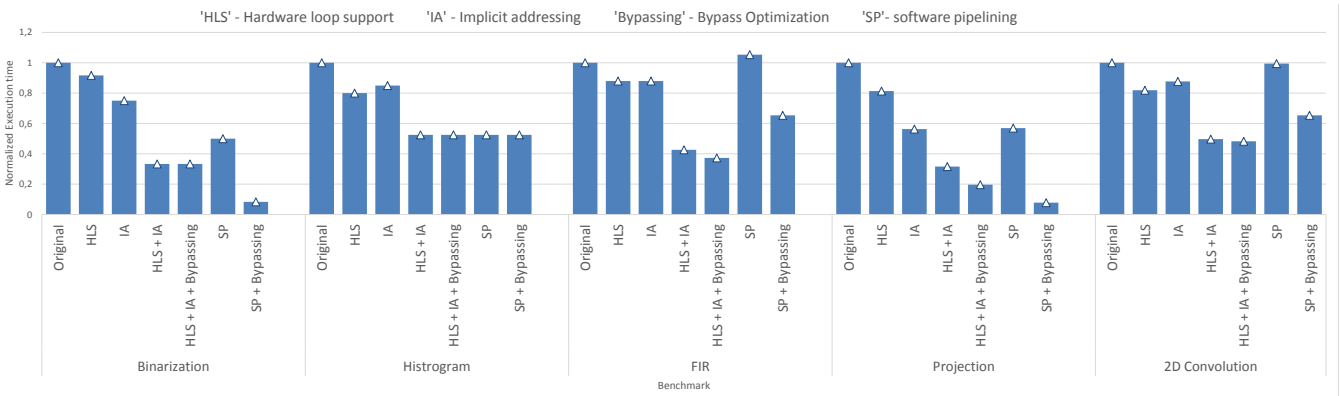


Fig. 18: Benchmark execution time comparison of compiler optmizations

[3] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, Jun. 2010.

[4] M. Wijtvliet, L. Waeijen, M. Adriaansen, and H. Corporaal, "Position paper: Reaching intrinsic compute efficiency requires adaptable micro-architectures," in *Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2016)*, 2016, pp. 25–31.

[5] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 235–244.

[6] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, Oct 1991.

[7] J. Llosa, A. González, E. Ayguadé, and M. Valero, "Swing module scheduling: a lifetime-sensitive approach," in *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*. IEEE, 1996, pp. 80–86.

[8] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *Execution Control*. Wiley-IEEE Press, 1997, pp. 91–98. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5266236

[9] Y.-L. Tsao, W.-H. Chen, W.-S. Cheng, M.-C. Lin, and S.-J. Jou, "Hardware nested looping of parameterized and embedded dsp core," in *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.*, Sept 2003, pp. 49–52.

[10] N. Kavvadias and S. Nikolaidis, "Elimination of overhead operations in complex loop structures for embedded microprocessors," *IEEE Transactions on Computers*, vol. 57, no. 2, pp. 200–214, Feb 2008.

[11] B. Mathew and A. Davis, "A loop accelerator for low power embedded vliw processors," in *International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004.*, Sept 2004, pp. 6–11.

[12] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 4, pp. 417–424, Dec 1997.

[13] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ser. ISLPED '99. New York, NY, USA: ACM, 1999, pp. 267–269. [Online]. Available: http://doi.acm.org.dianus.libr.tue.nl/10.1145/313817.313944

[14] D. She, Y. He, B. Mesman, and H. Corporaal, "Scheduling for register file energy minimization in explicit datapath architectures," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 388–393.

[15] R. Jordans and H. Corporaal, "High-level software-pipelining in LLVM," in *SCOPES '15 - 18th International Workshop on Software and Compilers for Embedded Systems*, Sankt Goar, Germany, June 2015, pp. 97–100.

[16] B. Cahoon, "An implementation of swing modulo scheduling in a production compiler," in *LLVM developers meeting*, October 2015, patch under review: http://reviews.llvm.org/D16829.

[17] M. Adriaansen, M. Wijtvliet, R. Jordans, L. Waeijen, and H. Corporaal, "Code generation for reconfigurable explicit datapath architectures with llvm," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug 2016, pp. 30–37.

[18] J. M. Codina, J. Llosa, and A. González, "A comparative study of modulo scheduling techniques," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/514191.514208

[19] *The BDTImark2000: A Summary Measure of DSP Speed*. Berkeley Design Technology, Inc, 2004.

[20] K. Vadivel, M. Wijtvliet, R. Jordans, and H. Corporaal, "Loop overhead reduction techniques for coarse grained reconfigurable architectures," in *2017 Euromicro Conference on Digital System Design (DSD)*, Accepted.