Loop overhead reduction techniques for coarse grained reconfigurable architectures

Kanishkan Vadivel*, Mark Wijtvliet*, Roel Jordans*, Henk Corporaal*

*Department of Electrical Engineering
Eindhoven University of Technology
Eindhoven, The Netherlands
Email: k.vadivel@student.tue.nl

Radboud RadioLab, Department of Astrophysics/IMAPP
Radboud University
P.O. Box 9010
6500 GL Nijmegen, The Netherlands

Abstract—Due to their flexibility and high performance, Coarse Grained Reconfigurable Array (CGRA) are a topic of increasing research interest. However, 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. This paper investigates three hardware based loop optimization techniques that can significantly improve the energy efficiency of CGRAs. The three techniques are evaluated on processing kernels from the image processing domain as well as an industrial computer vision application. Energy consumption and area estimates are obtained using a CGRA synthesized with a commercial 40nm library. For the three applied techniques (zero-overhead loop accelerator, single-cycle loop support, and loop buffers) the simulation results show overall energy gains of 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.

Keywords—loop support, energy efficiency, reconfigurable architectures, CGRA

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 opti-
mized 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.

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 evaluation on the impact of the energy-efficiency 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 introduces the hardware loop optimizations, and how they can efficiently be applied to CGRAs. Section V then describes how these optimizations will be evaluated and Section VI 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 mainly 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 [6] and RISC [7] 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 [8]. 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 [9]. Others [10] 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%.

III. BACKGROUND

This section introduces the reference CGRA architecture that will be used to obtain our energy and area results. Additionally, a programming example will illustrate how this architecture can be used to compute signal processing kernels.

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 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 Fileless (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.

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.

Listing 1: Binarization kernel source code

```c
void binarization(char *IN, char *OUT)
{
    for(int i=0; i<10*10; i++)
        OUT[i] = (IN(i) > 127)?1:0;
}
```

B. CGRA Programming Example

The binarization algorithm is used to illustrate how an application is mapped on to the CGRA. Binarization performs a simple image thresholding kernel with a single loop at its core as shown in listing-1. The custom CGRA configuration for executing this kernel is shown in fig. 3. In order to reduce the loop body size, the configuration uses two ALUs, one for updating the loop count $ALU_{loop}$ and another for thresholding the pixel values of the input image (ALU). It is also possible to execute the application on 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 and, results in increased application run-time and instruction memory accesses. The schedule for the loop body of the binarization algorithm on two ALU configuration is given in fig. 4. For simplicity, the remaining part of the schedule, the schedule of ALU, LSU and IMMs are not shown here. The given schedule iterates the loop instructions(indicated by red-lines) for 100 times. The empty cells in the schedule are interpreted as NOP instructions. The CGRAs uses bypass paths as a source and destination operands compared to register names in regular architectures. Hence for better interpretation of schedule, blue arrows are used in table to mark the data-paths in the schedule. The CGRA has two cycles branch latency. In order to avoid pipeline-hazards, the two cycles following relative branch instruction (brci) are scheduled with two NOP instructions. By applying loop unrolling, the branch delay slots can be filled with computations, but for clarity of this example this optimization is omitted.

It can be observed from the schedule that the flexibility (reconfiguration) of CGRA allows the application to be executed in energy efficient manner through its direct mapping onto hardware without the overhead of register file accesses. Further energy saving can be achieved by analysing and optimizing the energy consumption of individual units in the platform in order to obtain high FU utilizations.
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. 4. 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. 5. 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 overhead. This allows ZOLA to support data-dependent loops (e.g. while loops) without any additional overhead.

### 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. 4. 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. 5. 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 overhead. This allows ZOLA to support data-dependent loops (e.g. while loops) without any additional overhead.
to execute loops independent of each other. For instance, “loop L0, L1” and “loop 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. 6 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. 7. 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. 7: Loop Buffer Organization. (BCL-Buffer control logic, LB-Loop Buffer)](image)

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. EXPERIMENTAL SETUP

A set of three image processing kernels namely Binarization, Erosion, and FFoS are used in the baseline setup to identify the possibilities for improving the energy efficiency of the CGRA. The kernels are chosen in such a way that it expresses most common cases of signal processing application [11]. FFoS is an image processing application developed for an industrial setting where the centres 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 analyse the energy consumption of individual functional units in the CGRA, the design is synthesised 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.

The geometric mean of the benchmarks is shown in fig. 9.

![Normalized Energy breakdown for benchmarks](image)

The fig. 8 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 arbitrator (multiple load and/or store requests to global memory are handled by an arbitrator) cost is 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.

The geometric mean of the benchmarks is shown in fig. 9. 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 [12], 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.

VI. Evaluation

In this section the evaluation and results for the three evaluated hardware accelerators 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 25% 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 25% 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 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 fig. 10a, 10b and 10c. The binarization application has static loop as its kernel and hence having an buffer will not cache any loop instructions and degrades the performance as shown in 10a. The buffer size effect on erosion kernel is given in fig. 10b. 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. 10b. Any increase in buffer size thereafter has 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.

C. Results

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Configuration</th>
<th>IMem Accesses</th>
<th>Normalized Energy</th>
<th>Normalized Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binarization</td>
<td>Original</td>
<td>8212</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>With ZOLA</td>
<td>8210</td>
<td>0.86</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>With SCL</td>
<td>14</td>
<td>0.57</td>
<td>0.91</td>
</tr>
<tr>
<td>Erosion</td>
<td>Original</td>
<td>8990</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>With ZOLA</td>
<td>8992</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>With LB(8-line)</td>
<td>513</td>
<td>0.89</td>
<td>0.98</td>
</tr>
<tr>
<td>FFoS</td>
<td>Original</td>
<td>14232</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>With ZOLA</td>
<td>14248</td>
<td>0.97</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>With SCL</td>
<td>13230</td>
<td>0.96</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>With SCL+LB (8-line)</td>
<td>296</td>
<td>0.88</td>
<td>1.06</td>
</tr>
</tbody>
</table>

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) is are used.

The fig. 11 shows the comparison of energy savings achieved by the proposed optimizations on the individual benchmarks. Table I 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 fig. 11, the energy saving on IF/ID and FU is higher for the binarization and erosion kernels compared to FFoS. This is because the issue-slot that previously performed loop condition calculation cannot be removed because 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 reduction of energy consumption in the IF and ID. Adding single cycle loop support does not provide any gains for the erosion kernel since it does not have any steady state loop. However the overhead of steady state loop support is very minimal and therefore energy stays constant. 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.

The binarization kernel does not benefit from a loop buffer since its only loop is a single cycle loop. Over 97% of the instructions in erosion and FFoS are the loop instructions. Therefore with optimal buffer sizes, the overall energy is reduced by 6.6% and 8.6% for erosion and FFoS respectively compared to ZOLA + single cycle support. The overhead added by the loop optimization hardware is the highest for in FFoS since it uses 16-line buffer for its 7-issue processor configuration compared to the erosion which uses 8-line buffer for the 5-issue processor configuration.

VII. 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 VI 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 cancelled out (for 2 out of 3 kernels) by the removal of hardware that is no longer required, such as an extra ALU issue slot. For the third kernel the area increase is between 1% and 6%. This paper shows that CGRAs that are optimized for energy efficiency can be a key player in the search for energy efficient mobile compute devices. The hardware optimizations discussed in this paper will be integrated in the energy efficient CGRA architecture that our group is developing.