



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

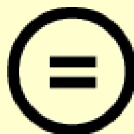
다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#) 

**SCALING KERNEL SPEEDUP TO
APPLICATION-LEVEL PERFORMANCE
WITH CGRAS: STREAM PROGRAM
APPROACH**

Seongseok Seo

**Department of Electrical Engineering
Graduate School of UNIST**

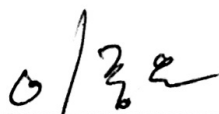
Scaling Kernel Speedup to Application-level Performance with CGRAs: Stream Program Approach

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Seongseok Seo

01.27.2014

Approved by



Major Advisor

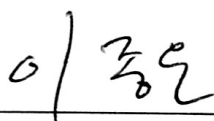
Jongeun Lee

Scaling Kernel Speedup to Application-level Performance with CGRAs: Stream Program Approach

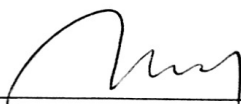
Seongseok Seo

This certifies that the thesis of Seongseok Seo is approved.

01.27.2014



Thesis Supervisor: Jongeun Lee



Wonki Jeong: Thesis Committee Member #1



Giljin Jang: Thesis Committee Member #2

Abstract

While accelerators often generate impressive speedup at the kernel level, the speedup often do not scale to the application-level performance improvement due to several reasons.

In this paper we identify key factors impacting the application-level performance of CGRA (Coarse-Grained Reconfigurable Architecture) accelerators using stream programs as the target application.

As a practical remedy, we also propose a low-cost architecture extension focusing on the nested loops appearing very frequently in stream programs.

We also present detailed application-level performance evaluation for the full StreamIt benchmark applications, which suggests that CGRAs can realistically accelerate stream applications by 3.6~4.0 times on average, compared to software-only execution on a typical mobile processor.

Contents

Contents	vi
List of Figures	vii
I. Introduction	1
II. Related Work	4
III. Target Architecture	6
IV. Application Mapping Flow	8
4.1 Stream Program	8
4.2 Kernel Mapping	9
4.3 Application Level Mapping	9
4.4 Configuration Prefetch	10
V. Optimizations	11
5.1 Pipelining of Nested Loop	11
5.2 Special Operators	12
5.3 Architecture Extension	14
5.4 Mapping Flow for Nested Loop	15
VI. Experiments	18
6.1 Experimental Setup	18
6.2 Mapping Nested Loops	19
6.3 Full Application Performance	21
VII. Conclusion	23
References	24

List of Figures

3.1	Our compilation flow for stream programs.	7
5.1	Skeleton C code for accumulators.	14
5.2	Extended PE, It has only one instance of α _REG though it seems as if there are two.	14
5.3	Mapping of 4 new instructions.	15
5.4	Decision flow for transforming imperfect nested loops.	16
6.1	Kernel runtimes, normalized to that of Case C (asterisk: kernel for which fission is required).	20
6.2	Application runtimes, normalized to that of Case A.	21

Introduction

To continue on the exponential performance growth without dissipating too much power, researchers are actively considering even disruptive innovations. Reconfigurable architectures with word-level granularity, called coarse-grained reconfigurable architectures, are one of them, and hold a promise as they have both extremely low power consumption and high performance for certain application domains [2]. With recent developments in compiler technology for CGRAs, they are a step closer to realizing their potential.

However, previous work on compilation [3, 5, 8, 16, 17] for CGRAs addresses only the problem of mapping loops to the architecture, and the problem of how to scale kernel speedup to the application level received little attention. As summarized in Table I.1, most previous works report only the kernel IPC, without any consideration of application speedup. One exception is [2], where the kernel portion (before acceleration) is calculated to be about 83% on average, giving the average application speedup of about 2 times compared to their 4-way VLIW main processor execution. While application-level performance improvement is likely very important to prospective users, it is not only largely unknown in the literature but also heavily dependent on many factors including the number and sizes of kernels and the details of the compilation method such as how the kernels and the rest of the application communicate.

On some level, this emphasis on kernel speedup in CGRA research is understandable. CGRAs

Table I.1: Results of previous work on CGRA mapping

Ref. (CGRA size)	Target	Kernel IPC
[16] (8x8)	DSP kernels (idct, fft, etc.)	12 – 28.7
[17] (4x4)	214 loops from H.264, AAC, MP3	9.6
[2] (4x4, 4-way)	Software Defined Radio	8.76 – 11.05
[5] (4x4 or 8x8)	DSP kernels	11 – 29
[3] (4x4)	Kernels from media and embedded	10 – 15
[8] (4x4)	Kernels from DSP and multimedia	N.A.

are originally proposed as malleable hardware that could replace hard-wired ASIC accelerators. As such, only a few kernels, typically compute-intensive ones, are mapped to run on the low-power accelerator. Also, in embedded systems, where CGRAs are primarily used, application programs are often aggressively tweaked for more performance or lower power, which may jeopardize any realistic and objective application-level performance evaluation of CGRAs. Finally, almost all applications spend most of their execution time on loops anyway.

The last point is particularly interesting and persuasive, since Amdahl’s law says that 90% of kernel portion and 10 times kernel speedup, for instance, may generate up to 5X ($= 1/(0.1 + 0.9/10)$) performance improvement at the application level, an expectation that is hardly met on larger benchmarks. There are several factors contributing to this gap between expected vs. achieved performance levels. The Amdahl’s law assumes infinitely parallel work during parallel sections, which is far from true in many embedded applications. Also the law only gives an upper bound, ignoring all the control, communication, and other overheads on implemented architectures. Further, though most application runtime is spent on loops, not many of the loops may be mappable to CGRAs—to be mappable, a loop must be both recognized by the compiler as a loop and supported by the architectural features. Our preliminary study reveals that while over 99% of the runtime¹ is typically due to any repeating sequences of instructions, the loops that could be mapped to a CGRA account for far less—less than 50% on average for MiBench applications.

As a practical way to drastically improve *achieved* application-level performance, we explore using explicitly parallel programs, such as stream programs. Stream programs provide natural representations for important target application domains of CGRAs, such as multimedia and DSP, and also exhibit ample parallelism explicitly stated in the programs. Yet, they are devoid of problems plaguing C-based compilation such as global pointers, unbounded arrays, dynamic memory allocation, and recursive functions. As a result, stream applications have much higher

¹In terms of dynamic instruction count.

portions of exploitable loops (90% on average for StreamIt applications), which provides a strong basis for expecting higher application-level performance.

In this paper we first show that stream applications, due to their higher kernel portions, can be realistically accelerated using CGRA by 3.6~4.0 times on average, over software-only execution. This is significant, since our evaluation uses entire stream applications from the StreamIt benchmark suite [19], whose sizes are on a par with those of MiBench applications, and there is no manual code change necessary to realize the performance. Our evaluation results also shows that application performance is strongly impacted not only by kernel portion but also by various overheads due to control, DMA, and configuration fetch.

To increase kernel portion and decrease overhead, we propose mapping entire loop nests directly to CGRA using *loop flattening* [6]. While loop flattening itself does not always improve performance due to extra operations it introduces, our low-cost architecture extension, which supports arbitrary-depth nested loops with rectangular iteration space, can effectively eliminate the extra operations. Our experimental results demonstrate that our optimizations can very effectively increase performance of kernels (by 56%, on average) and stream applications containing nested loops (by 35%, on average), compared with mapping inner-most loops only. Over software-only execution, on average 3.6~4.0 times speedup at the application level is obtained.

Our results can be interpreted in two ways: (1) as a predictor of application level performance of using CGRA after manual code rewrite, (2) demonstration of the efficacy of using stream programs as a front-end for programming CGRA architectures.

Related Work

As mentioned earlier, previous work on CGRA mapping [3, 5, 8, 16, 17] has mainly focused on kernel performance only, and the inner-most loop in particular.

Currently there are only a few options of mapping a nested loop on a CGRA. First, only the innermost loop can be selected as a kernel executing on the CGRA while outer loops execute on the main processor. While straightforward, this option incurs repeated overheads due to CGRA invocation, pipeline filling/drainage, and variable initialization. This overhead is magnified if the trip count of the innermost loop is low while that of an outer loop is high. Second, one may fully unroll the innermost loop, which then becomes a straight-line code. This loop transformation can be useful if the innermost loop's trip count is very low; otherwise, it may not be practical due to the increased code size. Third, software pipelining can be performed directly along an outer loop using Single-dimension Software Pipelining (SSP) [18]. SSP generalizes and improves single-loop software pipelining by allowing to pipeline a loop nest at any loop level. However, choosing an outer loop means that inner loops must be executed without software pipelining (such as using branch instructions), which may be difficult or impossible on a CGRA. Lastly, a CGRA may be architecturally modified to support multiple levels of loops [13], which however requires not-so-small hardware extension and that the number of loop levels must be fixed *a priori* at design time.

Stream applications have been researched extensively, including efficient realizations on the Cell processor [14], GPU [10], FPGA [7, 9], and other reconfigurable architectures [11]. Those techniques often focus on exploiting task-level parallelism (TLP) and balancing workloads among multiple processors. Our target architecture, CGRA, does not lend itself easily to such TLP due to its lock-step execution model among PEs. Instead we focus on how to run each actor efficiently. Also to the best of authors' knowledge, no prior work has examined application-level performance of stream programs on CGRA this thoroughly.

Target Architecture

CGRA: The basic idea of CGRA is to use ALU-like processing elements (PEs), as opposed to LUTs (look-up tables), as a basic unit of configuration. The PEs are arranged in a 2D array, connected via mesh-like interconnects, and operate in lock steps, making it important to avoid stalls in any PE. Arithmetic and logic operations are usually performed by any PE, but expensive operations such as multiplication/division/memory operations may be performed by some PEs only. For memory-accessing PEs, a multi-banked scratchpad memory serves as the local memory, which provides guaranteed access time with no memory stall. Thus it is the compiler's responsibility to ensure that the data accessed by memory PEs are present in the CGRA's local memory.

The interconnect architecture connects the output port of one PE to its neighboring PEs' input ports. Each PE's output is registered, and the PEs all operate in lock-step. Hence, pipelined execution is a natural way to exploit the parallelism of a CGRA. The instruction of a PE is called *configuration*, which can be quickly changed to another thanks to the distributed configuration cache. Section 5.3 contains more detailed description of our target architecture.

Overhead: The minimum overhead in invoking CGRA execution typically includes the following: (1) the MP performing a series of write instructions to set key parameters of CGRA execution, such as configuration address, initiation interval, and prolog/epilog size, and (2) in-

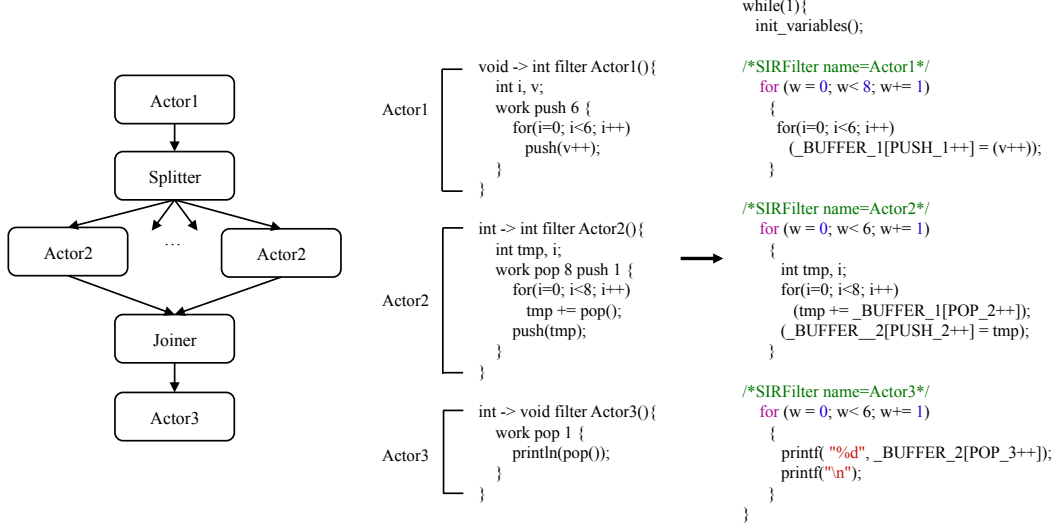


Figure 3.1: Our compilation flow for stream programs.

interrupt or polling latency for the MP to resume execution at the end of a CGRA execution. Those overheads are referred to as *control overhead*.

CGRAs are equipped with configuration caches to enable quick configuration switch, typically at zero-cycle overhead. However, if the total size of the configurations needed to execute an application is greater the capacity of the configuration cache, dynamic reloading of configuration from the main memory is necessary. As StreamIt applications contain many actors, dynamic reloading of configuration is necessary in almost all the applications of the StreamIt benchmark suite.

First the main processor (MP) issues instructions to perform data transfers for scalar/array values between main memory (or a cache) and the CGRA’s local memory, either directly or via DMA. Second, MP performs a series of write instructions to set key parameters of CGRA execution, such as configuration address, initiation interval, and prolog/epilog size. Third at the end of a CGRA execution, MP is given back the control, which can be done via interrupt or polling.

Application Mapping Flow

4.1 Stream Program

Many applications that run on infinite streams of data can be naturally represented as stream programs. A stream program consists of *actors* communicating with each other via explicit FIFO channels. Explicit representation of data flow and dependence between actors makes it easy to extract the parallelism in the program, including task parallelism (among actors) and loop parallelism (inside an actor). Stream programs are a good target for CGRAs because each actor is a loop, since it should be able to run indefinitely, which helps make the kernel ratio very high.

While many stream programming languages exist, in this paper we use StreamIt [19], which facilitates backend optimization by supporting hierarchical structure, based on three operators, *pipeline*, *splitjoin*, and *feedbackloop*. The leaf nodes of the hierarchy are called *filters*. Filters do the actual work, which is represented by one or more phases (i.e., loops) that are cycled through during the steady state.

4.2 Kernel Mapping

Since we focus on loop parallelism, we can use the conventional kernel-based mapping flow. Using the StreamIt-to-C translator [19], we first convert a StreamIt program into a C program, from which suitable¹ innermost loops, or *kernels*, are identified and mapped to the CGRA, while the rest of the C program is mapped to the main processor.

4.3 Application Level Mapping

Going from kernel-level to application-level involves making sure that the CGRA has all the configuration, data, and control information necessary to execute each kernel.

Configuration: The configuration cache is a distributed hardware cache, whose block size is equal to the size of the configuration bits to program all PEs for one cycle. Thus to execute a loop scheduled with the initiation interval of n , n blocks of configuration are required. The simplest scheme to bring configuration from the main memory is *on-demand loading*, which works as follows. The MP first invokes the CGRA by providing the start address of the configuration and the initiation interval. The CGRA first checks if the requested address range is already present on the configuration cache. If it is a hit, then the CGRA can proceed to execute the loop using the existing configuration; otherwise, it requests DMA (Direct Memory Access) to load the configuration from the main memory, which delays the CGRA execution until configuration loading is completed.

Data: There are two issues related to data: (i) transfer of input/output data (i.e., arrays) for a kernel and (ii) the management of data if they are too large to fit in the local memory of the CGRA. The first can be done by performing DMA before/after a kernel execution. The second issue arises from the limited capacity of the CGRA's local memory, which is a scratch-pad. A typical solution is *loop tiling*; however, arrays with irregular access patterns (e.g., array A in $A[B[i]]$) must still be loaded in their entirety, which may disqualify some loops from being CGRA-mappable.

Control: Control has two components. First, scalar variables may need to be transferred. Start addresses of arrays on the CGRA's local memory are a typical example. Second, to execute a kernel on the CGRA, the CGRA must know the kernel parameters such as the configuration

¹Typically, any inner-most loop with a known trip count that does not include any function call.

address (serving as the kernel ID), the initiation interval, the prolog (epilog) length, and the trip count. These variables and parameters can be passed by the main processor using a sequence of store instructions.¹ We assume that once a kernel is finished, it takes only a few cycles to synchronize between the main processor and the CGRA.

4.4 Configuration Prefetch

For applications with many actors, the configuration cache may not be able to retain all the configuration data necessary. In such a case the performance of on-demand loading quickly deteriorates, but this configuration loading latency can be easily hidden for stream applications by a simple prefetch. When the parameters for the current kernel are passed to the CGRA, the next kernel, as determined by compiler to be the one appearing next in the stream program, can be scheduled to be prefetched. This requires only two more parameters to be added to the kernel parameters, increasing the control overhead marginally.

¹To shorten the control overhead, static information (such as II and prolog length) could be retrieved directly from the memory or cache. However its effect is not expected to be high.

Optimizations

5.1 Pipelining of Nested Loop

Stream programs consist of actors which are mostly nested loops. The loops are formed from at least two sources: scheduling between actors for rate matching, and accessing input/output channels. Consequently, most actors in stream programs are at least 2- or 3-deep nested loops. Further, due to the indefinite stream-based nature of the applications, there is one large loop at the outermost level, which could be tiled—for more efficient implementation, for instance—creating another level of loop.

Often those loop nests have low trip counts at the innermost loop, which, if mapped to CGRA directly, can result in high overhead in terms of control and main processor cycles, in addition to increased prolog/epilog overhead in CGRA execution. To reduce these overheads we apply loop flattening (or loop collapsing) [6] transforming a nested loop into a single level loop, which can be very efficiently executed in loop accelerators such as CGRA. The original iterators, if necessary, can be computed from the new iterator using modulo and division operations. The outer loop computation (including iterator update and loop control) is now done on the CGRA itself instead of the main processor, and there being only one¹ instance of software

¹Our flow requires loop fission in some cases, which results in multiple flattened loops.

pipelining for the entire loop nest, the loop control and prolog/epilog overhead can be minimized.

While straightforward in concept, the critical challenge is in maximizing performance when the loop is imperfect and/or recurrent. Transforming imperfect loops into perfect ones using guarded statements [4] can sometimes be detrimental to performance, since it increases the number of operations in the innermost loop, due to the outer loop operations moving into the innermost.

From the performance standpoint, the idea of regenerating the original iterators, in the first place, already introduces new operations into the innermost loop, which can negatively impact performance. Recurrence, which is frequently manifested in accumulation and index calculation, only complicates the problem, as it limits certain loop transformations such as loop fission.

Lastly for a general trip count that is not a power of two, regenerating the original iterators requires expensive operators such as modulo and division. In summary, loop flattening increases the total number of operations on the CGRA though it can eliminate the repeated overhead of calling the CGRA.

5.2 Special Operators

For the above-mentioned reasons, loop flattening alone is not always beneficial in mapping nested loops, as demonstrated in our experiments. But since nested loops appear frequently in stream applications, there is an incentive to apply application-specific architecture optimizations to mitigate the impact of new operations added by flattening. The newly added operations have very similar patterns, which makes it easy to define and implement specialization via *special operators*. Moreover, we find that the special operators we introduce can also be used for other common computations (e.g., accumulation) that are not related to iterator update or loop control.

The family of special operators that we propose is referred to as Universal Regular Iterator and Accumulator (URIA), to which these computations can be efficiently mapped: iterator update, index update, and reduction operations (such as summation). URIA operator has five parameters in addition to the operator type (see Table V.1). We classify the URIA operators

Table V.1: URIA operator parameters (values are for the examples)

Parameter	Nested Iterator	Resetting Accum.	Leaping Index
α	0 (initial value)	0 (reset value)	N (leap value)
δ	1 (increment)	increment	increment
ρ	Nk	Nj	Nj
ω	Nj - 1	— (<i>unused</i>)	—
op	ADD	ADD	ADD

into two groups, namely, iterator and accumulator.

Iterator: With URIA operators, the original iterators are directly computed rather than regenerated using expensive operators. URIA can support any iterator of a rectangular iteration space, such as iterator j in the following code in C.

```

for (i=0; i<Ni; i++)
    for (j=0; j<Nj; j++)
        for (k=0; k<Nk; k++)
            Statements ...

```

In general an iterator of a rectangular iteration space repeats the following sequence (j 's example, parentheses added for readability):

$((0, 0, \dots, 0), (1, 1, \dots, 1), \dots, (N_j - 1, N_j - 1, \dots, N_j - 1))^+$ where each value is repeated N_k times before incremented. Implementing this through regeneration requires both modulo and division operations while it can be generated directly using one URIA operator, with parameters listed in Table V.1.

Accumulator: Index update, which is a special case of accumulation, happens frequently in stream applications when actors access channels. In other applications index update can also be used as pointer update. Accumulation is another very frequent operation in DSP applications.

Fig. 5.3 shows the skeleton code for an accumulator operation. Note that the RHS of the accumulation statement may be either a constant or an expression. Again such an accumulator can be implemented using one URIA operator (including the resetting or leaping behavior), with parameters listed in Table V.1.

```

for (i=0; i<Ni; i++){
    sum = 0;
    for (j=0; j<Nj; j++)
        sum += ...
}

```

(a) Resetting accumulator

```

for (i=0; i<Ni; i++){
    index += N;
    for (j=0; j<Nj; j++)
        index += ...
}

```

(b) Leaping index

Figure 5.1: Skeleton C code for accumulators.

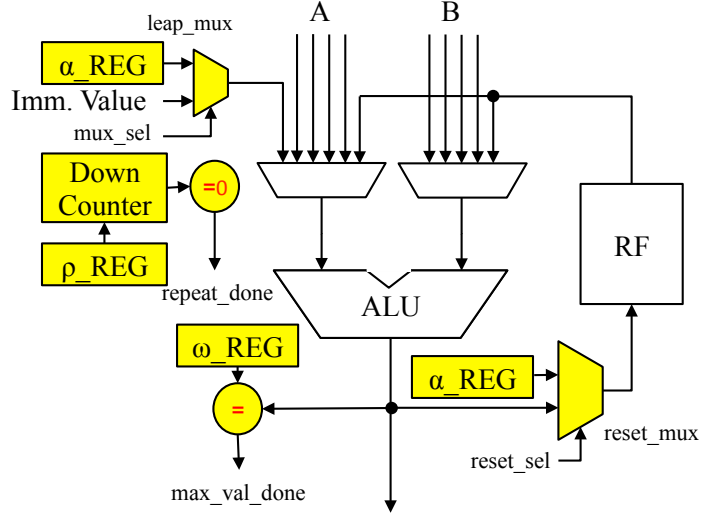


Figure 5.2: Extended PE, It has only one instance of α_REG though it seems as if there are two.

5.3 Architecture Extension

It is straightforward to define the semantics of the special operators. For instance, following is the definition of the resetting accumulator, where x is the output of the operator, kept in a register:

$$x \leftarrow \begin{cases} \alpha, & \text{if the current iteration is } \rho\text{'s multiple} \\ x \text{ op } \delta, & \text{in other iterations.} \end{cases}$$

This definition presumes that the operator is evaluated once every iteration, which can be done in a way similar to updating a rotating register file every II (Initiation Interval) cycles [12]. Finding out if the current iteration is ρ 's multiple can be implemented using a simple counter. The **op** symbol above represents any arithmetic operation supported by the underlying PE, which means that the iterators/accumulators can be based on not only addition, but also subtraction, multiplication, or any Boolean operation.

The URJA operators can be easily implemented on top of any PE architecture that supports

a local register file and has an ALU and input muxes (i.e., parts shown in white in Fig. 5.2). Though our new operators have several parameters, only some of them are encoded in the configuration, with α , ρ , and ω being written directly into the corresponding registers. This way, the new operators can be encoded with little impact on the configuration bit-width. The area overhead is very small, as it adds only 1 down-counter and 3 registers plus small combinational circuitry as highlighted in the figure. The controller of a PE must be extended but the change is minimal, since all the control signals for our extension can be statically determined by the operator type, and the controller extension is stateless except for the nested iterator that requires 1-bit state only. The critical path is affected only by the mux (*reset_mux*), which adds little delay to the cycle time of a PE.¹

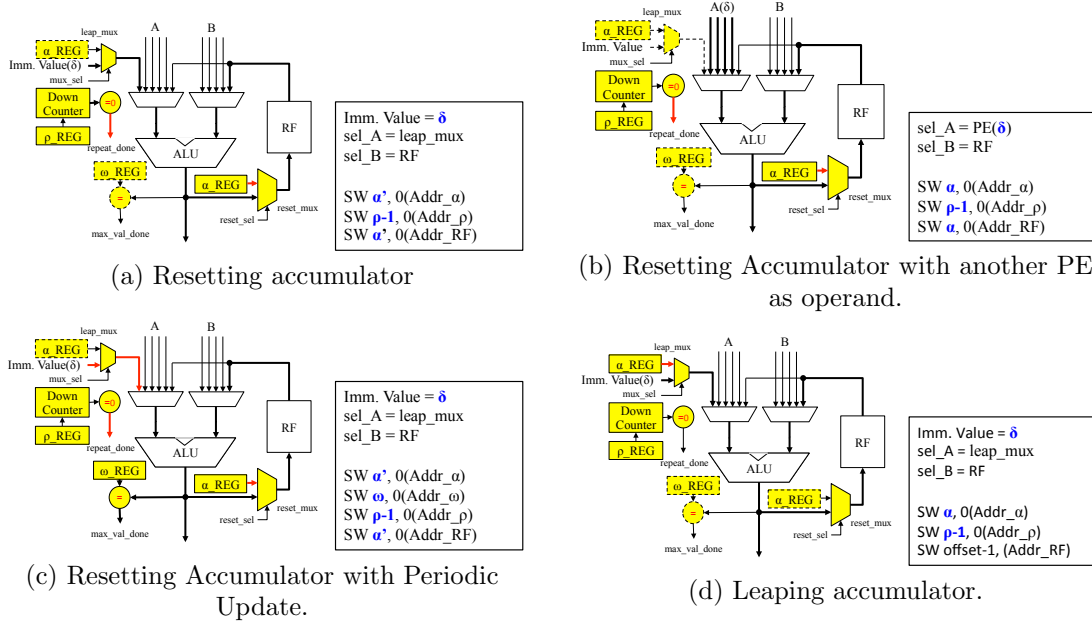


Figure 5.3: Mapping of 4 new instructions.

5.4 Mapping Flow for Nested Loop

While a perfectly nested loop can be easily flattened into a single-level loop, most loops are imperfect. Hence, regardless of whether URJA operators are used, we need a mapping flow to transform an imperfect loop into one or more perfect loops. Imperfect statements can be either guarded using predicates or fissioned into a new loop.² The former can degrade performance

¹Even when the PE works as a nested iterator (enabling α to be selected when the ALU output reaches ω), the mux is used in a later iteration than the comparator (*max_val_done*).

²While full unrolling of a loop can be another way to handle imperfect loops, it is impractical unless the trip count is very small; thus, we regard loop unrolling as an independent optimization that can be enabled at the designer's discretion.

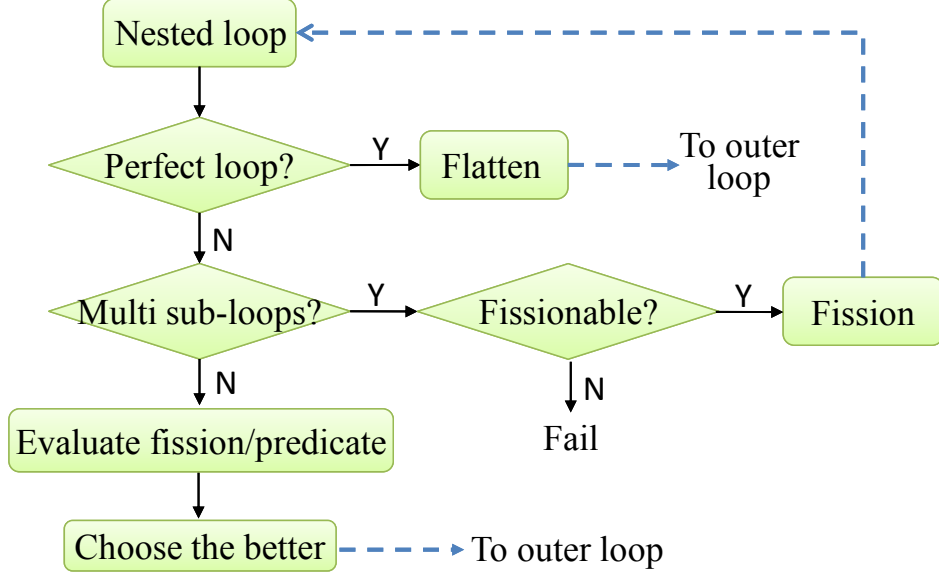


Figure 5.4: Decision flow for transforming imperfect nested loops.

by increasing the initiation interval of the pipeline, but the latter incurs some control overhead and may not be applicable for recurrent loops. Since which option will generate higher performance is not obvious, we generate estimates for both cases and choose the better. To manage complexity, we do this decision-making recursively, starting from the second-innermost loop to outer loops.

Fig. 5.3 illustrates our decision flow for a 2-deep nested loop. If a loop is imperfect, it should be one of these two types: (i) one that has only one inner loop, and (ii) one that has multiple inner loops in a sequence. For the second type, loop fission is the only way to map the loop nest to a CGRA, unless some of the inner loops are completely unrolled. Therefore after checking inter-iteration dependence, we either fission each inner loop into a new loop nest, or (if it is impossible due to dependence cycle(s)) conclude that the loop nest, in its entirety, cannot be mapped to a CGRA. The fissioned loops must be either perfect or of the first type.

For the first type, we only need to take care of the imperfect statements before/after the only inner loop, which can be again fissioned out (dependence permitting) or moved into the inner loop with proper guards. Depending on which ones are fissioned out as separate loop(s), there are four possible cases, which are evaluated to choose the best one. Evaluating each case involves transforming the loop body at the data flow graph (DFG) level and scheduling the modified DFG on the target CGRA to find out the initiation interval and the overall execution

time of a loop.

Once all the loop levels are successfully examined, finally the original loop nest can be transformed into one or more perfect loop nests, which are individually flattened and mapped to the CGRA.

Experiments

6.1 Experimental Setup

To evaluate application-level performance, we use SimpleScalar cycle-accurate simulator [1], which supports multiple levels of caches and virtual memory. For accurate bus and memory timing, we integrate DRAMsim [20], and extend the simulator with DMA and CGRA models. Table VI.1 summarizes key architecture parameters. In addition, our CGRA has homogeneous PEs, but memory operations can be performed by four specific PEs only, which are connected to the CGRA’s 4-bank local memory via a crossbar switch. The load latency is 4 CGRA cycles, fully pipelined. Configuration cache has 64 blocks, each of which can configure the PE array for one cycle. The increased cycle time of our special PE is assumed to be 5% slower than the base CGRA.

We use StreamIt programs, which are first converted into C using StreamIt-to-C translator [19], and fed to LLVM [15] integrating our CGRA scheduler. After CGRA mapping is

Table VI.1: Architecture parameters

Component	Parameters
Main Proc	720 MHz, in-order, 2-instruction issue
Cache	L1 (16+16KB), L2 (128KB)
CGRA	520 MHz, 4x4 PE array, mesh+diagonal
Bus+DRAM	DDR-333 (32-bit), pipelined bus (64-entry)

Table VI.2: Cases compared

Case	Description	Difference
A	Software only	Use MP (Main Proc.)
B	Mapping inner-most loop only (IML)	+CGRA
C	IML, with configuration prefetch	+Conf. prefetch
D	Mapping nested loop (NL)	+Nested loop mapping
E	NL, with special PEs	+Special PEs

Table VI.3: Kernel experiment results (see text for explanation of heading)

ID	Application	Loops N/K	Case C		Case D		Case E		
			#n	#e	#n	#e	#n	#e	#x
1	channelvocoder	34/72	9	9	19	17	13	13	4
2	filterbank	40/90	9	9	17	14	11	11	3
3	mpeg2-subset	21/32	12	11	23	20	15	14	4
4	tde-pp	47/48	46	59	61	73	55	67	6
5	fft	25/29	17	17	23	21	18	17	2
6	dct	16/34	11	11	24	22	16	16	4
7	bitonic-sort (2x)	19/44	20	20	24	23	20	20	2
8	vocoder	16/59	7	17	32	29	22	19	6

finished, the mapping results as well as the corresponding control/prefetch/DMA code are back-annotated into the C source code, which is recompiled using SimpleScalar-GCC to generate executable for main processor (MP). The CGRA compiler implements modulo scheduling based on [17] without rotating register file support, but with predicated execution support for conditionals within loops. One application, *serpent_full*, is not used in our experiments because our compiler back-end fails to schedule some loops due to their large size.

Table VI.2 lists the five cases we consider. We perform two experiments. In the first experiment, we select one kernel from each application, and evaluate our two nested loop mapping approaches (without or with URJA operators, i.e., Case D or E) compared to the conventional mapping (inner-most loops only; Case C). In the second experiment, we evaluate the entire benchmark applications.

6.2 Mapping Nested Loops

Nested loops are frequently found in stream applications, as evidenced in Column 2 of Table VI.3, which lists the number of all kernels (K) and that of kernels in a nested loop (N). However, two benchmarks, *fm* and *des*, consist of single-level loops only, enclosed only by a global loop. In another benchmark, *beamformer*, though there are many nested loops, most of them have multiple inner loops and recurrence, which cannot be handled by our mapping flow of Fig. 5.3. These applications are used for the second experiment only. For the other applications, we choose the most important loop as the kernel, as ranked by the product of initiation interval

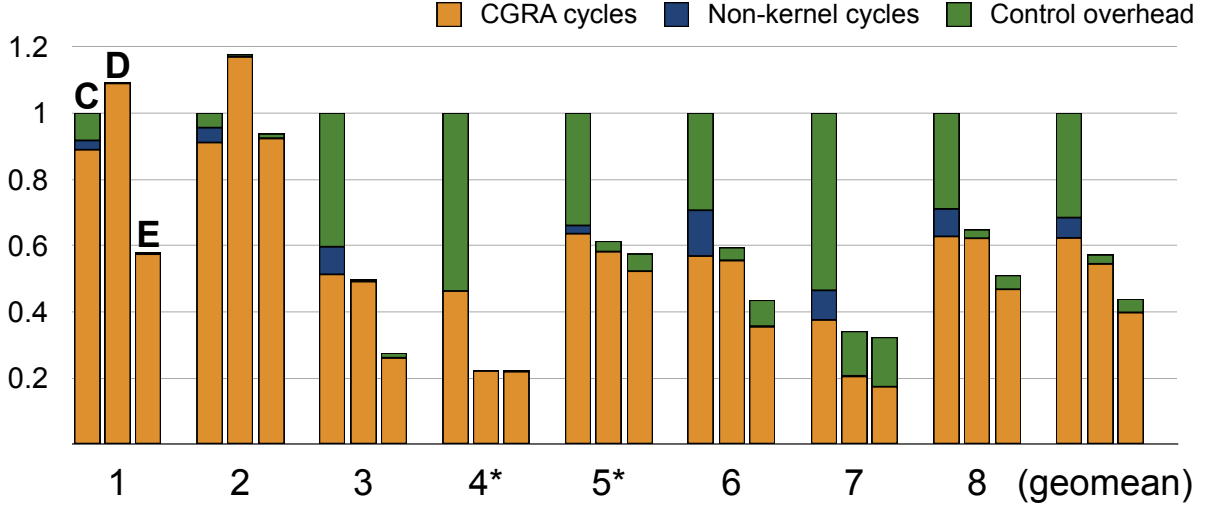


Figure 6.1: Kernel runtimes, normalized to that of Case C (asterisk: kernel for which fission is required).

(II) of the loop and its execution count. Exceptions are made when an identical kernel would be selected from different applications, as is the case with *channelvocoder* and *filterbank*, and *tde-pp* and *fft*; in those cases, the second-in-rank is chosen in one of the applications to avoid duplication. The kernel of *bitonic-sort* has the inner-most loop with a very small body (10 operations) and just 2 iterations, which is unrolled manually.

The right-hand side of Table VI.3 shows how the loop body (or its control-data flow graph) is changed by our nested loop mapping. Here #n and #e represent the number of operation nodes and that of data-dependent edges, respectively (in Case C, of the *inner-most* loop only). As all the kernels are imperfect loops, we observe that the number of nodes increases often significantly by loop flattening (Case D). With our special operators, however, the extra nodes are nearly eliminated. The number of special PEs required is listed in the last column. It turns out that imperfect statements are best handled by predication rather than fission, which is due in large part to the large load latency on a PE. In Case E, all kernels are better off with predication and in Case D, only *vocoder* is better off with fission. Kernels for *tde-pp* and *fft* can only be fissioned due to multiple sub-loops.

Fig. 6.1 shows the kernel runtimes, as normalized to that of Case C. The numbers on the x-axis indicate the application ID, defined in Table VI.3. Overall, kernel runtimes are reduced 42.8% on average by loop flattening alone, though they sometimes increase. The reduction comes mostly from control overhead reduction, which is significant in many kernels.¹ Loop flattening also reduces prolog/epilog overhead of pipelining (included in CGRA cycles), which is significant

¹Control overhead is sensitive to MP’s memory write speed. In our experimental setting, device write transactions from MP bypass all caches and reach the CGRA in 5~6 processor cycles per word.

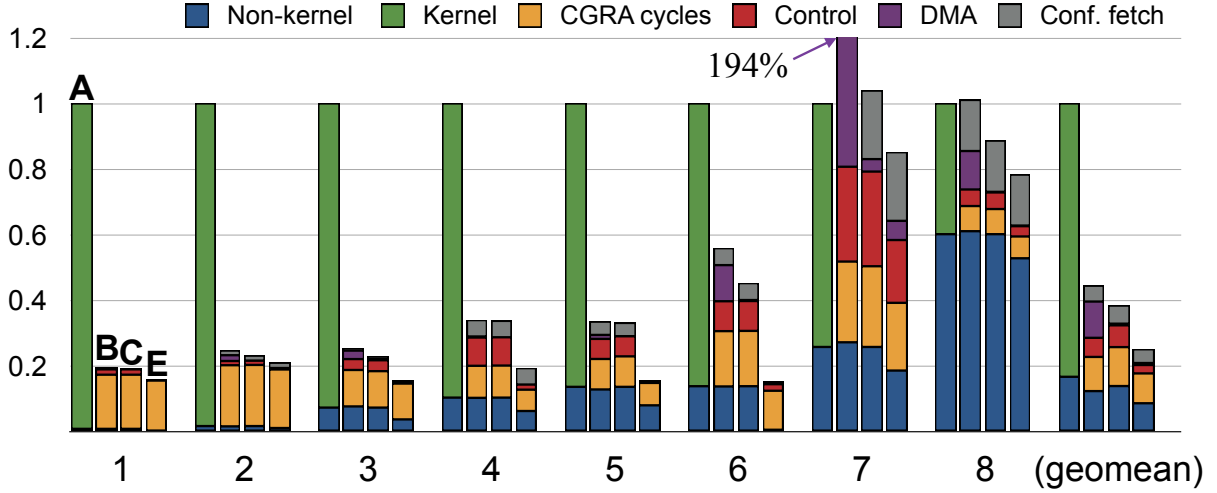


Figure 6.2: Application runtimes, normalized to that of Case A.

only when the trip count is small, which explains why we see a negative correlation between higher CGRA cycle reduction and the amount of control overhead—both are related to the trip count of the inner-most loop.

While reduced pipelining overhead is often canceled out in Case D by increased node count of the loop body, we observe in Case E that the CGRA cycles are also reduced. Our special PEs require initialization of some registers, which adds to the control overhead. But in all cases we see that the extended PE architecture consistently provides speedup over Case D, reducing the kernel runtime by 23.3% on average.

6.3 Full Application Performance

Fig. 6.2 shows our cycle-accurate simulation results. First we observe from Case A that the kernels take about 83.4% of the application runtime. The C programs generated from stream applications consist of two parts, initialization and a global loop. The initialization is executed only once, setting up constant tables and preparing other data structures whereas the global loop is executed indefinitely, representing the steady-state. The execution time in this experiment include only the latter. First we observe that the kernel portion—which, as reported in this graph, is accurate—varies considerably depending on the application. *vocoder* has the smallest, which is due to math functions used in actors. As expected, kernel portion has a dominant effect on the achievable speedup of CGRA.¹

¹The application IDs are given in the decreasing order of kernel portion.

Cases B and C represent the conventional approach mapping inner-most loops to CGRAs. The impact of on-demand configuration loading (as opposed to configuration prefetch) is not high for most applications because due to the large size of configuration cache, all configuration accesses soon become hit. And while the impact of repeated configuration misses is very high, a simple prefetch scheme can eliminate it.

Due to the large kernel portion, mapping inner-most loops only already gives an impressive runtime reduction of 61.7% (Case C over Case A), or about 2.5X speedup over Case A (software execution). However, this level of performance is far from ideal, due to non-kernel cycles, control, and DMA overhead, as seen in the graph. Our technique to map entire nested loops can reduce the overheads significantly, generating about 35% runtime reduction (Case E over Case C), or about 4X speedup from Case A.

In some applications DMA cycles become quite significant. This is because in Applications 7 and 8, the kernel portion is relatively small, which increases the need to do DMA between MP and CGRA. Though not shown in the graph, the application-level runtime reduction is quite high in the other 3 applications as well, generating the average application speedup of 3.6X from Case A to Case E, while the conventional inner-most loop only mapping (Case C) generates only 2.6X speedup over Case A.

Conclusion

Stream programs are not only widely used in multimedia and DSP domains, but also have measurably higher kernel portions, making it an ideal front-end for CGRAs. At the same time, while the heterogeneous model of computing employed by CGRAs makes objective and realistic evaluation of application-level performance very difficult, using stream programs, as we have demonstrated in this paper, seems to somewhat avoid the difficulty. Focusing on the nested loops appearing very frequently in stream programs, we also propose a low-cost architecture extension. Our detailed performance evaluation using the full StreamIt benchmark applications suggests that CGRAs can realistically accelerate stream applications by 3.6~4.0 times on average, compared to software-only execution on a typical mobile processor.

References

- [1] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35, 2002. [18](#)
- [2] B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins. A coarse-grained array accelerator for software-defined radio baseband processing. *Micro, IEEE*, 28(4):41–50, 2008. [1](#), [2](#)
- [3] Liang Chen and T. Mitra. Graph minor approach for application mapping on cgras. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 285–292, 2012. [1](#), [2](#), [4](#)
- [4] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel® itanium® processor. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 182–191, Washington, DC, USA, 2001. IEEE Computer Society. [12](#)
- [5] Grigorios Dimitroulakos, Stavros Georgiopoulos, Michalis D. Galanis, and Costas E. Goutis. Resource aware mapping on coarse grained reconfigurable arrays. *Microprocessors and Microsystems*, 33(2):91 – 105, 2009. [1](#), [2](#), [4](#)
- [6] A. Ghuloum et al. Flattening and parallelizing irregular, recurrent loop nests. In *Proc. PPOPP 95*, 1995. [3](#), [11](#)

REFERENCES

- [7] A. Hagiescu, Weng-Fai Wong, D.F. Bacon, and R. Rabbah. A computing origami: Folding streams in fpgas. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 282–287, 2009. [5](#)
- [8] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: Using epimorphism to map applications on CGRAs. In *Proceedings of the 49th Design Automation Conference (DAC)*, 2012. [1](#), [2](#), [4](#)
- [9] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '08*, pages 41–50, New York, NY, USA, 2008. ACM. [5](#)
- [10] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. *SIGARCH Comput. Archit. News*, 39(1):381–392, March 2011. [5](#)
- [11] N. Kapre and A. DeHon. Vliw-score: Beyond c for sequential control of spice fpga acceleration. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–9, 2011. [5](#)
- [12] Suhyun Kim and Soo-Mook Moon. Rotating register allocation for enhanced pipeline scheduling. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, PACT '07*, pages 60–72, Washington, DC, USA, 2007. IEEE Computer Society. [14](#)
- [13] Yongjoo Kim, Jongeun Lee, Toan X. Mai, and Yunheung Paek. Improving performance of nested loops on reconfigurable array processors. *ACM Trans. Archit. Code Optim.*, 8(4):32:1–32:23, January 2012. [4](#)
- [14] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008. [5](#)
- [15] C. Lattner and V. Adve. Llm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004. [18](#)
- [16] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 166–173, 2002. [1](#), [2](#), [4](#)

REFERENCES

- [17] Hyunchul Park, Kevin Fan, and Scott Mahlke. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *In Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, 2008. [1](#), [2](#), [4](#), [19](#)
- [18] H. Rong, Zhizhong Tang, R. Govindarajan, A. Douillet, and G.R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 163–174, 2004. [4](#)
- [19] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R.Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin Heidelberg, 2002. [3](#), [8](#), [9](#), [18](#)
- [20] David Wang et al. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33:100–107, November 2005. [18](#)

