Application-Level Performance Improvement for Stream Program on CGRA-based systems

Hongsik Lee

Computer Engineering Program
Graduate School of UNIST

Application-Level Performance Improvement for Stream Program on CGRA-based systems

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

Hongsik Lee

12.15.2015
Approved by

Major Advisor
Jongeun Lee

Application-Level Performance Improvement for Stream Program on CGRA-based systems

Hongsik Lee

This certifies that the thesis of Hongsik Lee is approved.

12.15.2015

Thesis Supervisor: Jongeun Lee

Woongki Baek: Thesis Committee Member #1

Seokhyeong Kang: Thesis Committee Member #2

Abstract

Coarse-Grained Reconfigurable Architectures (CGRAs), often used as coprocessors for DSP and multimedia kernels, can deliver highly energy-efficient execution for compute-intensive kernels. Simultaneously, stream applications, which consist of many actors and channels connecting them, can provide natural representations for DSP applications, and therefore be a good match for CGRAs. We present our results of mapping DSP applications written in StreamIt language to CGRAs, along with our mapping flow. One important challenge in mapping is how to manage the multitude of kernels in the application for the limited local memory of a CGRA, for which we present a novel integer linear programming-based solution. Our evaluation results demonstrate that our software and hardware optimizations can help generate highly efficient mapping of stream applications to CGRAs, enabling far more energy-efficient executions (7x worse to 50x better) compared to using state-of-the-art GP-GPUs. Further, we eliminate communication overhead and reduce computation overhead using combination of sychronous/asynchronous processors and DMA. This optimization also improve performance by 17.1% on average comparing to baseline system.

Contents

Contents v

List of Figures vi

I. INTRODUCTION 1

II. RELATED WORK 3

III. CGRA AND NAÏVE MAPPING 5

3.1 CGRA . . . 5

3.2 Naive Mapping . . . 5

IV. SCOPE AND PROBLEMS OF APPLICATION ACCELERATION 7

V. MAPPING STREAM GRAPHS TO CGRA 10

5.1 Supporting Large Data . . . 11

5.1.1 Input and Variables . . . 11

5.1.2 Variables . . . 11

5.1.3 ILP Formulation (S-A case) . . . 12

5.1.4 ILP Formulation (A-A case) . . . 13

5.1.5 Linearization . . . 13

5.2 Increasing Task Granularity . . . 13

5.3 Transparent Configuration Management . . . 14

5.4 Mapping Flow . . . 15

VI. Experiments 16

6.1 Experimental Setup . . . 16

6.2 Comparison with GP-GPUs . . . 17

6.3 Application Speedup . . . 18

6.4 Effect of Limited Memory Size . . . 20

6.5 Analysis on Combination of Synchronous/Asynchronous Processors and DMA System in Limited Memory Size 21

VII. Conclusion **23**

VIII. REFERENCES **24**

List of Figures

4.1 Stream graphs have not only higher kernel portion but also higher kernel count, requiring new strategies. 8

6.1 Application runtimes, normalized to that of Case A. 18
6.2 Performance decrease due to limited local memory. 19
6.3 Framework of Section 5.5 experiment. 20
6.4 Application runtimes, normalized to S-S(A) case 22

CHAPTER I

INTRODUCTION

One of the ideas behind CGRAs (Coarse-Grained Reconfigurable Architectures) is to make programming easier-much easier than FPGAs while retaining custom-hardware-like efficiency. While there have been many advances in architectures and compilers for kernel mapping, scaling the kernel speedup to application level performance has been less than forthcoming. There is very scant evidence in the literature about application level performance scalability of CGRAs, and most of the recent work focuses on kernel speedup only (see Section 6).

Of course there exist applications in which a few kernels account for nearly all of the application runtime. For those applications it is relatively easy to get high application speedup. For others, however, kernels are often scattered and contain difficult-to-map constructs (e.g., function call, complex control flow). In such a case, accelerators like CGRAs have limited effect, and unless programs are rewritten significantly, getting high application speedup seems very difficult. This may suggest that the programming model for CGRAs needs to be redefined, so that it may be easier to use them for larger pieces of computation as well as to allow for a direct evaluation of application speedup as compared to other accelerators such as GP-GPUs.

In this paper we evaluate stream graphs as a potential application representation. Stream graphs, which consist of actors (or filters) and channels connecting them, have been frequently used to capture DSP and multimedia applications, which are also main applications of CGRAs. Stream graphs have also measurably higher kernel portions and kernel counts than other embedded applications (see Section 3), which makes automatic mapping approaches like ours both

plausible and desirable.

Efficient mapping of stream graphs to CGRAs brings new challenges and opportunities. For efficiency reasons CGRAs require kernel data such as input/output arrays to be present on their local memory, which is a scratchpad memory (SPM) with a limited size. But for a large number of kernels as in stream graphs, the SPM capacity may not be sufficient. Optimal selection of kernels is a complicated problem as buffers implementing channels are shared between kernels, which means that the memory requirement of a set of kernels is not necessarily the same as the sum of individual kernels' requirements. Configuration management can also be improved by taking advantage of static schedule of stream graphs. Our solution further includes two optimizations (nested loop optimization, work count optimization) that can help maximize application level performance of stream graphs.

As a final optimization, we improve performance using combinations of synchronous/asynchronous processor operation and DMA while reducing the communication overhead and computation overhead. In conventinal CGRA-system, host-processor and CGRA synchronously operate and communicate between host and CGRA have synchronously done by DMA(Direct Memory Access). We consider this system as a baseline in our last experiment and we design two models comparing the base-line. The first combination is a system in which host-processor and CGRA synchronously operates but DMA asynchronously transmits datas from host to CGRA. The second model has asynchronous processor′s operation and asynchronous DMA. From now, we call two models as S-A and A-A respectively (S-S system is baseline). The performance of A-A case is 17.1% better than S-S case.

The rest of the paper is organized as follows. After describing the CGRA architecture in Section 2, in Section 3 we empirically analyze the advantages and challenges of mapping stream applications to CGRAs. Based on the result we present in Section 4 our architecture-compiler solution for mapping stream graphs to CGRAs, which includes low-cost architectural enhancements and integer linear programming-based optimal kernel selection. In Section 5 we present our experimental results comparing CGRAs with GP-GPUs (and other results), which suggest that for stream graphs, CGRAs can often be more energy-efficient (7x worse to 50x better) than state-of-the-art GP-GPUs. We discuss related work in Section 6 and conclude the paper in Senction 7.

CHAPTER II

RELATED WORK

Stream graphs have been mapped to different architectures, including the Cell processor [20], GP-GPUs [12, 21], FPGA [22, 23], and other reconfigurable architectures [24]. Those techniques often focus on exploiting task-level parallelism (TLP) and balancing workloads among multiple processors. We borrow the workload balancing technique from [20], [28]. But we differenciate managing local memory for ILP constraints in Section 5 while making it easier than [20], [28] using heuristic way. Otherwise, we focus on how to run a given set of actors efficiently on one CGRA. As such, we expect our methodology to be applicable to multi-CGRA platforms, once the workload balancing among CGRAs is done. To the best of authors' knowledge, no prior work has examined efficient mapping of stream graphs to CGRAs.

Application mapping for CGRAs [1–5, 9–11] has been the subject of active research recently. However all of them target kernels only, and do not address application mapping issues, e.g., when there are large data and large number of kernels. In fact, most of the papers cited above report kernel speedups only. Some of them are listed in Table 6.1, where the figure of merit is frequently kernel IPC (Instructions Per Cycle).

One exception is [17], where the kernel portion (before acceleration) is estimated to be about 83% on average for their target application, resulting in the average application speedup of about 2 times compared with their 4-way VLIW main processor execution. While application speedup is likely the key metric to prospective users of CGRAs, it is largely unknown in the literature, partly due to its dependence on the architectural details (which determines control overhead)

Table 2.1: Results of previous work on CGRA mapping

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

and due to the difficulty of mapping embedded applications such as MiBench applications efficiently without manual code rewrite.

In terms of hardware operation with stream program, both target architecture's processors and DMA operates asynchronously. And those multi-processors have same processing elements. But our architecture is divided two different processors, Main processor and CGRA accelerator. So work load balancing used in [20], [28]

Some of the optimizations included in our solution are based on previous work. Many code and data management techniques for SPMs or cache-SPM hybrids [25] have been proposed. While some of them are more complicated than our problem formulation, our primary purpose is kernel selection in a stream graph, which allows for a cleaner formulation of the problem despite its being a data SPM management, which is in general more difficult. Though flattening-based nested loop mapping was proposed before [9], fullscale evaluation of it in conjunction with multiple application-level optimizations is new. Prefetch has been studied extensively for microprocessors [26]. Recently for reconfiguration archtectures a new prefetch scheme was proposed based on piecewise linear prediction [27]. Ours is a much simpler one tailored for stream graphs.

CHAPTER III

CGRA AND NAÏVE MAPPING

3.1 CGRA

CGRAs are composed of ALU-like processing elements (PEs) as opposed to LUTs (look-up tables), which has advantages in terms of reconfiguration time as well as compute efficiency for a certain class of applications. PEs are arranged typically in a 2D array, connected via mesh-like interconnects, and operate in a lock-step manner; thus it is vitally important to avoid stalls in any of the PEs. The instruction set of a PE may vary depending on the particular CGRA instance, but usually arithmetic and logic operations can be performed by any PE while expensive operations and memory operations may be performed by some PEs only. Connected with memory-accessing PEs are a multi-banked scratchpad memory (SPM), which provides guaranteed access time, and serves as a local memory for input/output data. It is the compiler's responsibility to ensure that the data accessed by memory PEs are present in the CGRA's local memory.

3.2 Naive Mapping

Most previous work on application mapping for CGRAs considers single-level loops only, where a loop body is represented by a DFG (Data Flow Graph). If there is a conditional statement in the loop body other than for loop control, it may be converted into data dependency

using if-conversion, assuming predication support is provided by the architecture. Function calls should be eliminated beforehand (e.g., by inlining). Then the problem of mapping a kernel to a CGRA becomes the problem of placing the nodes of, and routing the edges of, the DFG, for which a number of algorithms

typically based on software pipelining have been proposed [1–5]. On the application level, kernels are first identified from a given program. Kernels are those that can be mapped to a CGRA, or typically any innermost loops with a known trip count that contain no function calls inside. After kernels are mapped to the CGRA using one of the algorithms mentioned above, the rest of the program is annotated with code necessary for CGRA execution (such as DMA data transfer and CGRA register initialization), and then compiled for the main processor (MP).

Control Overhead: The minimum overhead in invoking CGRA execution 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) 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.

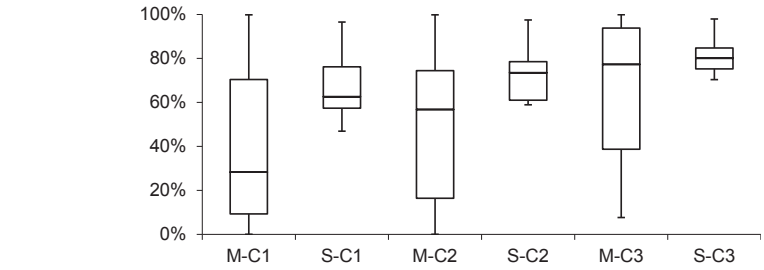
Runtime Reconfiguration: The instruction of a PE is called configuration, which contains information on PE operation and routing. When an executable is loaded, configuration is also loaded in the main memory. To allow quick configuration switch at runtime a CGRA employs a (distributed) configuration cache. Changing to another configuration that is already in the cache causes no runtime overhead, but to bring a new configuration into the cache takes time. If an application has many kernels (as in stream graphs), some configurations may have to be evicted and reloaded later. But for applications with a few kernels, configuration loading may not be a problem because it will happen only at the beginning of a program.

CHAPTER IV

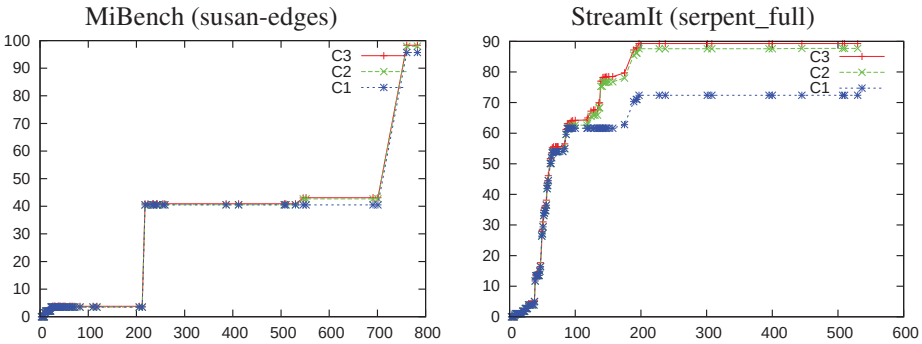
SCOPE AND PROBLEMS OF APPLICATION ACCELERATION

Though accelerators boast impressive kernel speedup, it may not always translate into application level speedup. In this section we examine why it may be hard to achieve good application level speedup with accelerator approaches, and list some of the problems we need address to achieve good application performance.

In the following we use two parameters: the kernel portion before acceleration, p, and the kernel invocation count, K. To find out the amount of speedup achievable by CGRA acceleration, we profile loops in StreamIt applications, which are first converted into C code by StreamIt-to-C translator [6]. For comparison we also profile MiBench applications [7]. We use our inhouse tool based on the SimpleScalar tools [8]. Our tool first constructs interprocedural control flow graph from disassembly code and profile information (profile information is used to resolve indirect branches and find execution frequencies of basic blocks), and then identifies all natural loops in each procedure, checking their nesting depths and whether they contain procedure calls. The size of a loop is measured in the number of instructions, and runtime is approximated to dynamic instruction count. We use all the applications in the benchmark suites except for a few that have problems with our infrastructure. In total we use 19 MiBench applications and 9 StreamIt applications.



(a) Kernel portions under different scenarios C1∼C3
(M for MiBench, S for StreamIt)



(b) Kernel portion (%) accumulated as kernel size is increased

Figure 4.1: Stream graphs have not only higher kernel portion but also higher kernel count, requiring new strategies.

We consider three scenarios different in the definition of a kernel: innermost loops only (C1), outer loops included (C2), and even those containing function calls (C3). In C1 and C2, a loop must not include a function call to qualify for a kernel. Scenario C1 may be considered conservative, whereas C3 is certainly the most aggressive. For C3, the runtime spent in callees is not counted toward kernel portion, but if the callee has loop(s), then those loops may contribute to the kernel portion. We set a limit to the size of a loop so that the outermost while-loop in StreamIt benchmarks may not be included.

The results are summarized in Figure 3.1. The first graph shows the range of kernel portions (p) under different scenarios, which reveals that MiBench applications have much larger variation in terms of kernel portion. We also see the different impact of potential "optimizations" introduced in C2 and C3. For MiBench applications, for instance, aggressive optimizations such as pipelining even loops that contain function calls may be necessary to achieve high application speedup. By contrast StreamIt applications already exhibit in C1 and C2 kernel portions of 67% and 74% on average, respectively, which means in principle we can achieve up to 3x to 4x application level speedup given enough resources.

On the other hand, StreamIt applications are trickier to handle. Figure 3.1(b) shows for

specific applications how kernel portion is distributed among loops as we increase loop size along the x-axis. On the left graph, two very noticeable steps, at loop sizes 200 and 700, indicate the existence of two dominant kernels, indicating a low K. By contrast, the kernel portion of $serpent\_full$ is made up of many small loops each contributing a small amount, indicating a high K. Though not as striking as those two applications, other applications of the benchmark suites exhibit a very similar trend.

There are a few lessons we learn from this study. In the case of general embedded applications represented by MiBench, the absolute portion of kernels may be too low to achieve good application speedup, especially if we consider more realistic scenarios. Even if we consider an aggressive one, some applications still have very low ($< 50\%$) kernel portions, severely limiting the scope for automatic compilation for CGRA-like accelerators. At the same time, the number of significant kernels is generally low, meaning that we need to optimize only a few kernels, which may be done by hand.

On the other hand, stream graphs generally have higher kernel portions, making automatic compilation more attainable. Automatic mapping is desirable as well due to the large number of kernels. The challenges, however, include i) how to efficiently manage large amount of configuration and data (due to the many kernels), and ii) how to minimize kernel invocation overheads. We present our solutions to this problem in the next section.

CHAPTER V

MAPPING STREAM GRAPHS TO CGRA

Our mapping strategy touches all three aspects of computation: processing efficiency, data management, and (host-to-accelerator) communication. First, to handle large data on a limited local memory of a CGRA, we present a static kernel selection approach that can determine the best set of kernels taking into account kernels' memory usage and buffer sharing information. Second, there is an opportunity unique in stream graphs that can help increase the kernel granularity.

The idea is that since stream graphs run indefinitely, the global loop repetitions can be distributed into each filter's trip count. This makes each filter run ω times as long (in terms of workload, not in absolute time) at the cost of increasing the application latency and memory requirement by the same factor. The parameter ω is called work count in this paper. Third, all filters in stream graphs are loops, but many have inner loops nested inside. If we map the entire loop nest as opposed to mapping the innermost loops only, we can not only increase p (kernel portion) but also decrease K (kernel invocation count). Several techniques are recently proposed, but we use one [9] that is based on loop coalescing. Finally, we discuss a simple yet effective prefetch scheme to hide configuration reload latency.

5.1 Supporting Large Data

Here we consider the kernel selection problem without work count optimization (i.e., $\omega = 1$). One simple solution would be to select the most important kernels, measured for instance in runtime contribution, and map them to CGRA. This, however, may not be the optimal solution because of complex data sharing between filters. Our ILP (Integer Linear Programming) formulation uses, among other things, data flow information between filters captured in a data flow graph, and can generate the optimal solution for any given SPM size.

5.1.1 Input and Variables

Data flow graph: $G = (V, E, W, B)$, where $V = \{n_i\}$ is the set of nodes and $E = (i, j)$ is the set of directed edges representing flow dependency from n_i to n_j. Edges must be realized as buffers, whose sizes are determined by W and B, with w_{ij} (or b_{ij}) representing the pop (or peek) size of edge (i, j). Needless to say, $w_{ij} \leq b_{ij}$ for every $(i, j) \in E$. In this paper a node represents one filter in a StreamIt program.

Execution time difference: p_i is the advantage of mapping n_i to CGRA instead of host processor in terms of host processor cycles. One can find it out by mapping a node (which may be a nested loop) both to CGRA and to host processor, and measuring the execution time difference using simulation. In the case of CGRA mapping, we include in the execution time all the control overhead cycles spent for CGRA invocations.

Schedule order of nodes: We require the schedule order of nodes in order to account for the live range of buffers. Without loss of generality we assume that i is the order of n_i in the static schedule.

Constant arrays: Also important to consider is the size of constant arrays, which are often global variables in StreamIt programs and shared by multiple actors. Such an array needs SPM space only if there is at least a node using it that is mapped to CGRA. Therefore we require their usage information in a matrix, whose entry aik is defined to be 1 if n_i uses array k (otherwise it is zero), along with their sizes $\{c_k\}$.

5.1.2 Variables

Decision variables: x_i is a binary variable, which is 1 only if node n_i is mapped to acce-lartor, and 0 otherwise.

Stage difference: y_{ij} is an integer variable, representing the stage difference between two connected nodes n_i and n_j.

DMA request: d_{ij} is binary variable, which is 1 only if the edge connecting n_i and n_j requires DMA, i.e., $x_i \neq x_j$.

5.1.3 ILP Formulation (S-A case)

Constraints:

1. In a subset of a graph with control divergence, all simple paths sharing the start and end points should have the same stage difference. That is, $\sum_{(i,j)\in P} y_{ij} = const$ for every path P connecting split to join.

2. Actors connected by a DMA should have the stage difference of at least 2. That is, $y_{ij} \geq 2d_{ij}$ for every $(i,j) \in E$.

3. Given an edge (i,j), the buffer size requirement for SPM can be determined three constraints of implementing the buffer associated with an edge, depending on where the two connected nodes are mapped as follows.

i) $\mathbf{x_i \neq x_j}$: One is mapped to CGRA, and the other to host processor. In this case DMA is necessary, and the amount of buffer for each actor depends on the stage difference of the actors and the DMA operation, thus requiring us to determine the exact stage of the DMA relative to the actors. However, since in our case one of the two actors must be mapped to CPU, which has a sufficiently large local memory, we can always place the DMA stage right next to the accelerator stage without any disadvantage. For instance, with an edge (i,j), if n_i is mapped to CPU and n_j to accdelrator, and their stages are $s_i = 2$ and $s_j = 5$, then DMA stage could be either 3 or 4, but by choosing 4, accelerator needs only one $(= s_j - 4)$ buffer unit, while CPU uses 2 $(= 4 - s_i)$. Thus regardless of the stage difference of an edge (i,j), we need only one copy of the buffer, which is w_{ij} (the pop size) if n_i is mapped to accelerator, or b_{ij} (the peek size) if it is mapped to CPU. Thus, the SPM size requirement in this case is $R_1(i,j) = x_i(1-x_j)w_{ij} + (1-x_i)x_j b_{ij}$, which needs to be reserved permanently. The DMA size is w_{ij}.

ii) $\mathbf{x_i = x_j = 1}$: Both nodes are mapped to CGRA. DMA is unnecessary, And if the two actors belong to the same stage, we need just temporary buffer space for data flowing between the two actors. However, if the two belong to different stages, we need additional buffer space whose size is proportional to the stage difference, or $R_2(i,j) = x_i x_j y_{ij} w_{ij}$.

The temporary buffer is live from the beginning of n_i until the end of n_j. To model the live range of a buffer, we introduce an integer array mt of size $|V|$, whose purpose is to keep track of SPM size requirement at each node invocation time. To implement the buffer, we need reserve $R_3(i,j) = x_i x_j b_{ij}$ bytes of memory on SPM during $t \in [i,j]$ only.

iii) $\mathbf{x_i = x_j = 0}$: Both are mapped to host processor. Buffer is implemented on global memory, and no DMA is necessary. Putting it all together, the SPM memory size requirement at time t (m_t) is determined as follows.

$$m_t = \sum_{(i,j)\in E} \{R_1(i,j) + R_2(i,j)\} + \sum_{(i,j)\in E \wedge i \le t \le j} R_3(i,j) + C \tag{V.1}$$

Here C is the SPM size for constant arrays, or $C = \sum_k c_k u_k$, where the new binary variable u_k is 1 if array k is used by any node mapped to CGRA. Thus for any k, the following inequalities hold: $u_k \ge a_{ik} x_i$ for $\forall i$, and $u_k \le \sum_i a_{ik} x_i$. Then for the SPM size of M, we requiremt $\le M$ for $\forall t$. Finally the objective of the ILP is to maximize $\sum p_i x_i$. This formulation can be linearized (see Section 4.1.5), but to linearize $R2$ we need to limit y_{ij} to Y_{max}, which is the maximum stage difference across any edge in the graph.

5.1.4 ILP Formulation (A-A case)

In this case, we implement ILP formulation as we simply change the objective function and add one constraint to S-A case. For input, we use execution time of MP (M_i) and CGRA (C_i) instead of execution time difference (p_i). To represent asynchronous operation of processors, we optimally divide the total workload of stream graph to CGRA and MP using M_i and C_i. And we represent divided workload as W, which is an integer variable. Finally, the objective function is minimizing W and additional constraints are $\sum M_i(1 - x_i) \le W$ and $\sum C_i x_i \le W$. Except for above condition, the formulation of A-A case is completely same to S-A case.

5.1.5 Linearization

The product of two binary variables, c = ab, can be linearized as follows: $c \le a$, $c \le b$ and $c \ge a + b - 1$. And the exclusive-OR of two binary variables, $d = a \bigoplus b$, can be linearized as follows: $d \ge a - b$, $d \ge b - a$, $d \le a + b$, $d \le 2 - a - b$. Lastly, The product, $z = ay$, of a binary variable a and an integer variable y, for which $0 \le y \le U$ holds, is a non-negative integer variable and can be linearized as follows: $z \le Ua$, $z \le y$, and $z \ge y - U(1 - a)$.

5.2 Increasing Task Granularity

We perform two optimizations to increase kernel granularity. First, frequently appearing nested loops in stream graphs deserve dedicated optimization. Recently a number of techniques for nested loops are proposed [9–11]. Since most of the loops we find in stream graphs are imperfect loops, we take the approach of [9], which can handle imperfect loops. The idea is

to apply loop coalescing technique to convert a multi-nested loop into single nested one. But to minimize the overhead of newly added operations due to coalescing, they define a small set of special operations such as iterator generator and accumulator that can do an additional operation such as resetting the iterator on a regular basis, the period of which is configurable. While this optimization involves hardware extension at the PE level, it is expected to increase both kernel portion (p) and kernel granularity (thus reduce K). The second optimization targets K. At the steady state, a stream graph is executed on a CGRA by running filters one at a time, sequenced by the host processor. Since stream graphs run indefinitely, we can repeat each filter ω times before executing the next filter without altering the correctness of the output. If there is a feedback loop at the global level, it only sets an upper bound on the value of ω. One critical drawback of the work count optimization is that it requires ω times larger buffers between filters. Due to the limited local memory of CGRA, its effect is in direct conflict with the purpose of kernel selection optimization. Hence one interesting issue here is whether it is possible that combining work count optimization and kernel selection may generate better results for some SPM size. Our preliminary results indicate that they need not be applied simultaneously; that is, kernel selection is only necessary when SPM size is small, and work count optimization only when SPM size is large. This is because mapping a kernel to CGRA generates much larger performance improvement than that of increasing work count. Thus the former has a higher priority in terms of memory allocation, and the latter needs to be considered only if all kernels are mapped to CGRA.

5.3 Transparent Configuration Management

As mentioned earlier, the baseline configuration fetch mechanism uses on-demand fetch, which, on receiving the CGRA call, checks if the requested configuration address exists on the configuration cache, and brings it from the main memory if necessary. For stream graphs with many filters, however, the configuration cache may not be able to retain all the configuration data, and its performance may quickly deteriorates.

Fortunately for stream graphs, the schedule of the filters can be statically determined. One issue is how to encode the information. We extend the CGRA call protocol to include the address for prefetch configuration. The prefetch configuration address would be written to a register (SFR) of the CGRA such that after the CGRA is invoked, prefetch can be started on the DMA. Another issue is how to predict the next kernel. We use a static method, which selects the one appearing next in (the C-converted version of) the stream program. This requires just

one more parameter to be added to the kernel parameters, increasing the control overhead only marginally.

5.4 Mapping Flow

After a stream graph is first converted into a C program, it is first analyzed for its memory requirement taking live ranges into account (using a method from [12]). If the memory requirement is larger than the available SPM size, it proceeds to the kernel selection step, after obtaining detailed statistics such as runtime and buffer access information of each kernel. Otherwise, all the kernels are mapped to CGRA, with the maximum work count possible. Then the next step is to transform the original code into a form that is directly executable on CGRA and Main Processor (MP), which involves partitioning the code, compiling the CGRA part, and creating kernel invocation part in the MP code. The information obtained in the previous step (selected kernels, work count) is also used for the code transformation. Finally, the transformed code is compiled and simulated for evaluation.

CHAPTER VI

Experiments

6.1 Experimental Setup

To evaluate application level performance, we use SimpleScalar cycle-accurate simulator [8]. For accurate bus and memory timing, we integrate DRAMsim [13], and extend the simulator with DMA and CGRA models. Table 1 summarizes key architecture parameters. The main processor (MP) is modeled after ARM Cortex A8 running 720MHz [14]. 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 (64Kbytes in total) via a crossbar switch. We set the timing parameters of caches and processors using Cacti 6.5 [15] and published documents [14, 16, 17]. The load latency of CGRA is 4 CGRA cycles, fully pipelined. The configuration cache has 128 entries, each of which can configure the PE array for one cycle. The stream graphs are from the StreamIt benchmarks [6], which are mapped to the MP-CGRA platform using the flow in Section 4.4. For compiler front-end we use LLVM [18], with a backend implementing modulo scheduling based on [2] 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 extremely large DFG sizes. Stream programs consist of two parts: the initialization part, which is executed only once setting up constant tables, etc., and a global *while* loop, which is executed indefinitely, representing the steady state. Only the latter is measured for the execution time in

Table 6.1: 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 6.2: Power parameters

	G8800	S1070	S2050	MP[19]	MPCGRA[17]
Power(W)	105	120	135	0.70(active) 0.071(idle)	0.38(active) 0.048(idle)
Technology	65nm	55nm	40nm	65nm	90nm

all of our experiments, including the GP-GPU results.

6.2 Comparison with GP-GPUs

Stream graphs have been mapped to many different accelerators, the most recent of which is GP-GPUs. In particular, we use the results of [12], which provides performance of several stream graphs on three different GPUs. For the GPU power numbers (see Table 5.2) we assume that the average power dissipation is 60% of the TDP value published by Nvidia, and divide it by 4 for multi-GPU platforms (S1070 and S2050), since [12] uses only one of the four GPUs present in each platform. The CGRA power includes its SPM and configuration cache power as well as that of the PE array, and is assumed to increase by 13% due to special PEs [9].

Table 5.3 compares the (normalized) runtime and energy consumption for all the stream graphs whose results are reported in [12]. Note that the GPU energy does not include Intel CPU energy whereas CGRA energy does include that of the main processor (ARM Cortex A8).

Table 6.3: Runtime (T) and Energy (E) comparison with GP-GPUs

	Application	G8800	S1070	S2050	CGRA	WC-opt
T	bitonic-sort	1	0.593	0.327	26.29	4.27
	DCT	1	0.733	0.280	2.07	1.55
	DES	1	0.706	0.273	7.88	6.21
	FFT	1	0.689	0.358	12.22	8.92
	filterbank	1	0.801	0.404	98.23	93.97
	FM	1	0.712	0.136	245.52	245.52
E	bitonic-sort	44.05	29.83	18.49	6.83	1
	DCT	140.10	117.34	50.37	1.41	1
	DES	32.54	26.25	11.41	1.30	1
	FFT	20.71	16.31	9.54	1.48	1
	filterbank	2.43	2.22	1.26	1.07	1
	FM	0.88	0.71	0.15	1.00	1

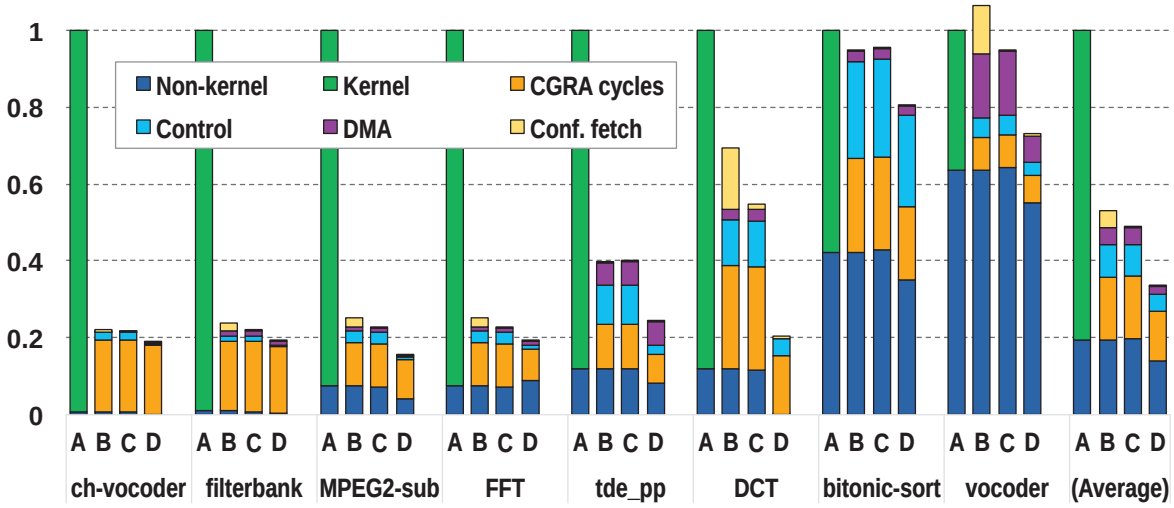


Figure 6.1: Application runtimes, normalized to that of Case A.

Table 6.4: Cases compared

Case	Description	Difference
A	Main processor only	MP only
B	Naïve without configuration prefetch	+CGRA
C	Naïve with configuration prefetch	+Conf. prefetch
D	Nested loop optimization	+Special PEs

For the CGRA results all the optimizations are enabled except for work count optimization, which is enabled in the last column only.

First we see a great variation among the applications. Some applications run much faster on GP-GPUs (e.g., filterbank, FM) while others can run on the MP+CGRA platform with only less than 10x slowdown. But since the MP+CGRA platform dissipates much less power, it beats the GP-GPUs in energy consumption for all but one applications. The highest gain in energy consumption is 50x to 140x reduction (in DCT) whereas the biggest loss is 14% to 6.7x increase (in FM), depending on the GP-GPU compared. Considering the difference in the technology used (see Table 5.2), our results demonstrate that CGRAs can be significantly more energy-efficient than GP-GPUs, as far as stream graphs are concerned.

6.3 Application Speedup

Figure 5.1 shows our simulation results for the four cases listed in Table 5.4. The applications are listed in the order of decreasing kernel portion. The runtime is broken down into multiple parts. The most interesting one is how kernel part is reduced to CGRA cycles. Though this reduction is significant in all applications, it is not as high as might be expected of a 4x4

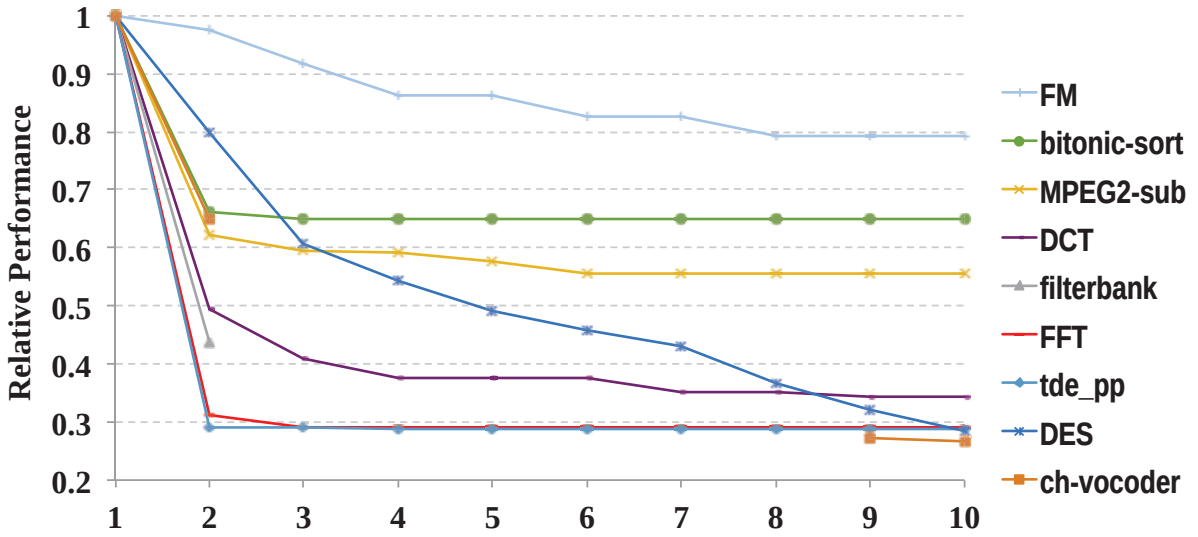


Figure 6.2: Performance decrease due to limited local memory.

CGRA. This is because the kernel time on a CGRA also includes prolog/epilog overhead due to software pipelining.

First we observe from Case A that the kernel portion is about 80.6% though it varies widely across applications. Application vocoder has the smallest, which is due to the math functions used in filters. As expected, kernel portion has a dominant effect on the achievable speedup of a CGRA. Case B represents the conventional approach of mapping only the innermost loops to the CGRA. The impact of configuration prefetch as opposed to on-demand configuration loading is not high for most applications due to the large configuration cache size, but for DCT and vocoder, it can reduce the application runtimes by over 20% and 10%, respectively.

Overall, due to the large kernel portion, mapping innermost loops only already gives the runtime reduction of over 50% (Case C over Case A), or about 2x speedup over Case A. Performance can be further improved by mapping entire loop nests to the CGRA, which can reduce kernel invocation overhead such as control overhead and DMA cycles as seen in the graph. It can also reduce the number of non-kernel cycles, sometimes very dramatically (e.g., DCT). However, additional code inserted when there are sibling inner loops may effectively increase the number of non-kernel cycles, which is the case with FFT. On average, Case D gives an additional runtime reduction of about 31% over Case C, and as compared with Case A, generates about 3x speedup.

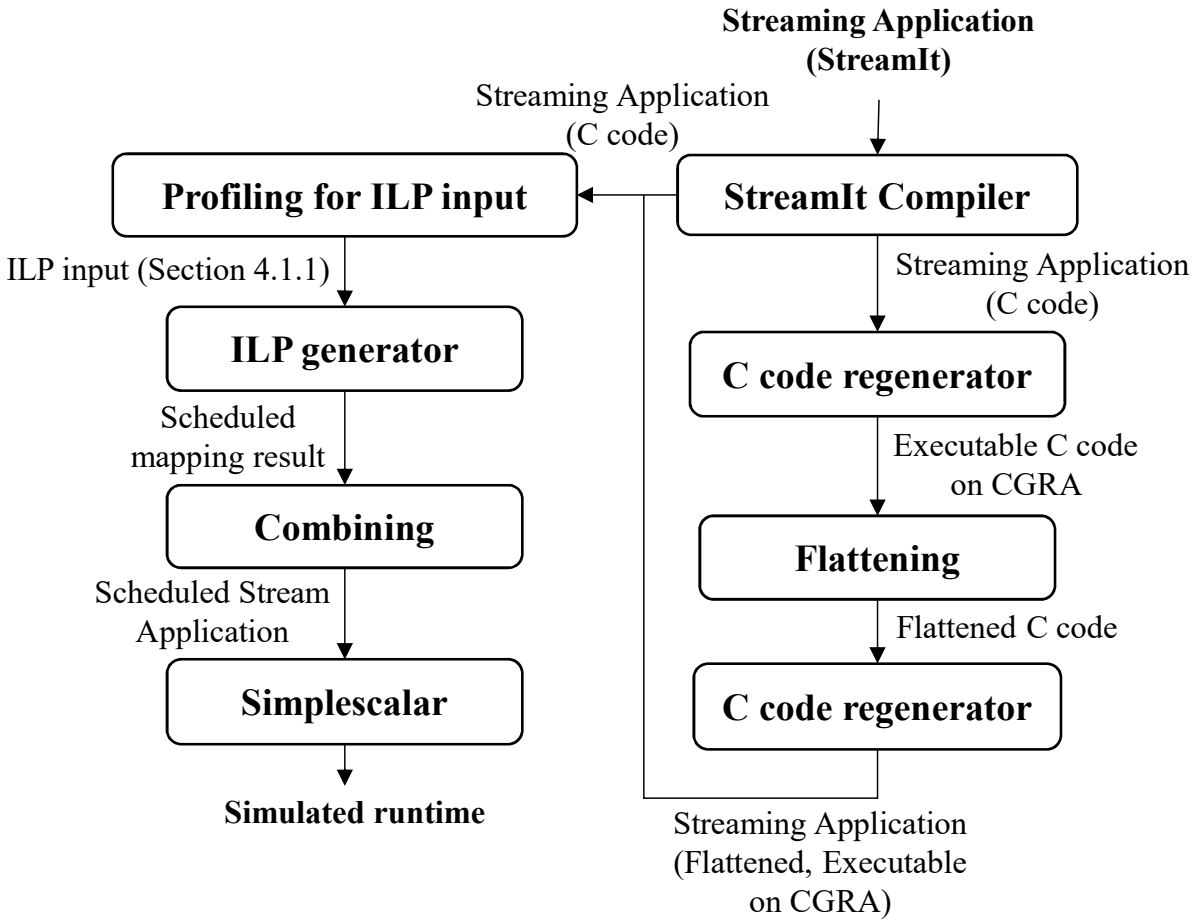


Figure 6.3: Framework of Section 5.5 experiment.

6.4 Effect of Limited Memory Size

Most of the original applications (except for tde_pp) have memory requirements that are small enough to fit in the SPM of our target architecture. In order to see the effect of limited memory size, in this section we use the smallest SPM size in which the application fits and multiply each application′s memory requirement by M times, varying M from 1 to 10. This is effectively the same as reducing the SPM size by the same factor, without making the memory size deviate from a power of 2. Figure 5.2 shows the performance, normalized to that of the $M = 1$ case. As M increases, the performance decreases monotonically. The ILP solver (we use open-source *lp_solve*) generates solutions within several minutes in most cases, but in some cases (the points with empty data on the graph) it took more than one hour, pointing to the need for an efficient heuristic method, which remains for future work.

Table 6.5: Limited SPM size

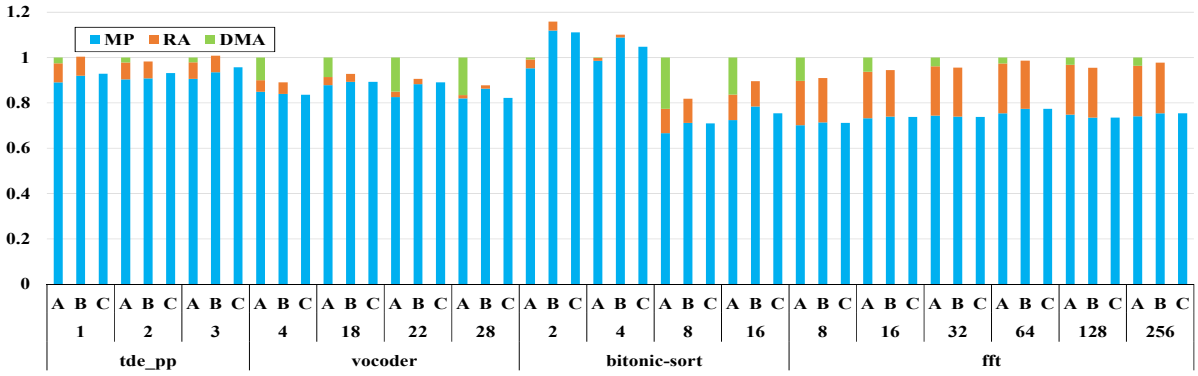
Application(A group)	SPM size	Application(B group)	SPM size
tde_pp	32768	filterbank	32768
vocoder	16384	dct	16384
bitonic-sort	256	ch_vocoder	16384
fft	16384	fm	16384

6.5 Analysis on Combination of Synchronous/Asynchronous Processors and DMA System in Limited Memory Size

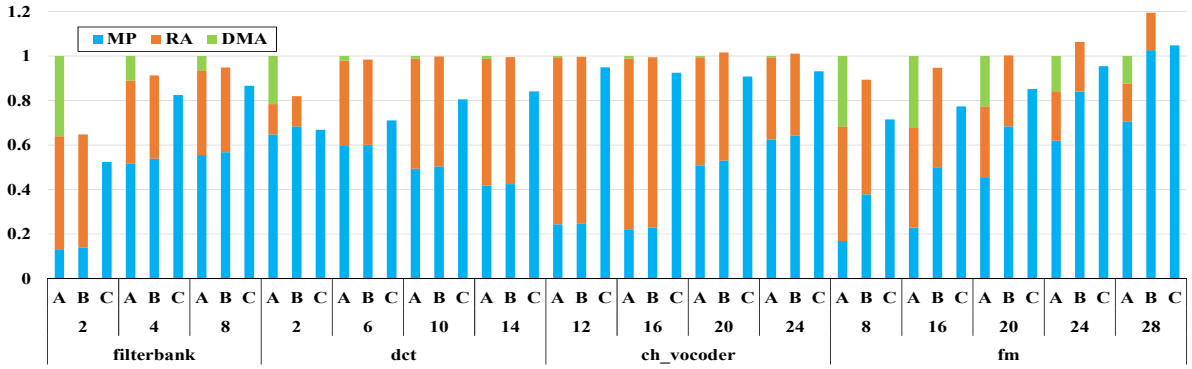
To make SPM size small enough, we use the SPM size in Table 5.5 so that filters in stream program are partially mapped to CGRA. Figure 5.3 shows the experiment framwork of this section. First, we use c code of stream application which generated by StreamIt compiler [6]. Then the C code is automatically flattened by the script which we program using pycparser 2.0 that is a C parser in Python. The code executable on CGRA (in respect of simulator) generated from flattened code by C code regenerator, which make code executable on CGRA by taking parameters such as initial interval, stages etc. (see Section 5.1). From the streaming application which generated in the first step and the flattened and executable C code, we profile two codes for ILP inputs described in Section 4.1.1. And we get schduled mapping results with ILP generator which solve ILP formulation using Gurobi Optimizer 6.5. We generate scheduled stream application by taking each filter's code from the two codes mentioned above. Finally, we simulate the scheduled stream application with Simplescalar [8].

Figure 5.4 shows our simulation results for the cases, S-S, S-A and A-A (the three systems is represented A, B and C respectively in Figure 5.2). The figure is divided into two groups. One is the group A (tde_pp, vocoder, bitonic-sort, fft) and group B (filterbank, dct, ch_vocoder, fm). In group A, see Figure 5.4(a), DMA overhead is completely eliminated by pipelining filters and A-A platform can hide the computation overhead of a processor (host-processor or CGRA) which takes lower amount of cycles during the runtime. the result of *bitonic-sort* has exceptional results when N is 2 or 4, x-axis of Figure 5.4, which means the size of stream graph. Because the size of steam graph is so small ($8-10$ filters) that the portion for improvement in communication overhead of S-A and A-A case is much less than software pipelining overhead; Software pipelining overhead is the additional instructions for double-buffering in which we increase a dimension of buffers. If load/store operation on the buffers increases, software pipelining overhead also increase.

In group B, A-A system generally fail to hide computation overhead completely with parallel execution of processor because the execution cycle portion of CGRA occupies about half or above 50% of total cycles. On the other hand, communication overhead is eliminated in $filterbank$



(a) Group A



(b) Group B

Figure 6.4: Application runtimes, normalized to S-S(A) case

but failed in fm due to software pipelining overhead. As a result, application-level performace of A-A improve by 17.1% on average of 8 applications comparing to S-S system.

CHAPTER VII

Conclusion

For many important target applications of CGRAs, stream graphs can provide very natural and easy-to-use representations. While efficient mapping of stream graphs to CGRA-accelerated platforms can be extremely useful and potentially widen the scope of CGRA applications, it also comes with multi-dimensional challenges. In this paper we present a set of architecture-compiler optimizations tailored for stream graphs considering all three aspects of computation. Our application-level performance show 3x speed-up comparing to baseline. Also we show 17.1% performace improvement on limited local memory using combinations of synchronous/asynchronous hardware operation. Our detailed evaluation results using the StreamIt benchmark suite suggest that CGRAs can be more energy-efficient than GPGPUs, ranging from 0.15x (6.7 times worse) up to 50x in terms of energy consumption, when low-cost optimizations are enabled. Our study also suggests the viability of stream graphs as a program representation for CGRAs.

CHAPTER VIII

REFERENCES

[1] B. Mei et al. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In Proc. FPT, pages 166–173, 2002.

[2] Hyunchul Park et al. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In Proc. PACT, pages 166–176, 2008.

[3] G. Dimitroulakos et al. Resource aware mapping on coarse grained reconfigurable arrays. Microprocessors and Microsystems, 33(2):91–105, 2009.

[4] Liang Chen and T. Mitra. Graph minor approach for application mapping on cgras. In Proc. FPT, pages 285–292, 2012.

[5] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: Using epimorphism to map applications on CGRAs. In Proc. DAC 2012, 2012.

[6] W. Thies et al. Streamit: A language for streaming applications. In R. Horspool,editor, Compiler Construction, volume 2304 of LNCS. Springer, 2002.

[7] M. Guthaus et al. MiBench: a free, commercially representative embedded benchmark suite. WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE.

[8] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. Computer, 35, 2002.

[9] J. Lee et al. Flattening-based mapping of imperfect loop nests for cgras. In Proc. CODES '14, CODES '14, pages 9:1–9:10, New York, NY, USA, 2014. ACM.

[10] D. Liu et al. Polyhedral model based mapping optimization of loop nests for cgras. In Proc. DAC '13. ACM, 2013.

[11] Yongjoo Kim et al. Improving performance of nested loops on reconfigurable array processors. ACM Trans. Archit. Code Optim., 8(4):32:1–32:23, 2012.

[12] A. Hagiescu et al. Automated architecture-aware mapping of streaming applications onto gpus. In Proc. IPDPS 2011, pages 467–478. IEEE, 2011.

[13] David Wang et al. Dramsim: a memory system simulator. SIGARCH Comput. Archit. News, 33:100–107, November 2005.

[14] Texas Instruments. OMAP3530/25 Applications Processor.

[15] Naveen Muralimanohar and Rajeev Balasubramonian. Cacti 6.0: A tool to understand large caches. 2009.

[16] ARM. ARM L220 Cache Controller Technical Reference Manual.

[17] B. Bougard et al. A coarse-grained array accelerator for software-defined radio baseband processing. Micro, IEEE, 28(4):41–50, 2008.

[18] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In Proc. CGO, pages 75–86, 2004.

[19] Texas Instruments. OMAP3530 Power Estimation Spreadsheet.

[20] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. SIGPLAN Not., 43(6):114–124, June 2008.

[21] A. Hormati et al. Sponge: Portable stream programming on graphics engines. SIGARCH Comput. Archit. News, 39(1):381–392, March 2011.

[22] A. Hagiescu, Weng-Fai Wong, D.F. Bacon, and R. Rabbah. A computing origami: Folding streams in fpgas. In Proc. DAC, pages 282–287, 2009.

[23] A. Hormati et al. Optimus: Efficient realization of streaming applications on fpgas. In Proc. CASES, pages 41–50, New York, NY, USA, 2008. ACM.

[24] N. Kapre and A. DeHon. Vliw-score: Beyond c for sequential control of spice fpga acceleration. In Field-Programmable Technology (FPT), pages 1–9, 2011.

[25] R. Banakar et al. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In Proc. CODES 2002. ACM, 2002.

[26] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. ACM Comput. Surv., 32(2):174–199, June 2000.

[27] A. Lifa et al. Dynamic configuration prefetching based on piecewise linear prediction. In Proc. DATE '13. EDA Consortium, 2013

[28] Yoonseo Choi et al. Stream Compilation for Real-Time Embedded Multicore Systems. In Proc. CGO 2009

[29] Hongsik Lee et al. Optimiziing stream program performace on CGRA-based systems. In Proc. DAC 2015