FIFO Initialization: Efficient Support for Serial Loops on Spatial Elastic CGRAs

Anonymous Author(s)

Abstract

We propose FIFO Initialization (FI), a novel hardware mechanism for spatial elastic Coarse-Grained Reconfigurable Arrays (CGRAs). FI pre-loads values directly into the FIFOs of a CGRA's Processing Elements (PEs) during CGRA configuration. FI eliminates the need to dedicate PEs for holding initial and/or loop carried values in serial loops (i.e., loops that carry dependence). FI enables CGRAs to benefit from two optimizations: (1) look-ahead expansion (LE) to efficiently pipeline serial loops, such as reductions and first-order linear recurrences, and (2) sliding-window optimization (SWO) for resource-efficient data reuse in stencils. To the best of our knowledge this work is the first to utilize these optimizations—previously used for multicores and VLSI circuit design—on CGRAs, enabled by FI.

We extend an 8×8 target CGRA with support for FI and build compiler support that exploits FI to apply LE and SWO. We use simulation to assess benefits to pipelining performance and resource utilization for benchmarks with serial loops and stencils. Evaluation shows that FI combined with LE achieves 9.5× throughput increase on average over state-of-the-art approaches to handling serial loops. Further, FI with SWO reduces PE usage by up to 3.9× in stencils while enabling efficient data reuse. Synthesis of the target CGRA with 15nm technology reveals that FI introduces minimal area/power overheard of 1.24%/4.59%.

Keywords: Spatial elastic CGRAs, reductions, recurrences, look-ahead expansion, sliding-window data reuse

1 Introduction

Coarse-Grained Reconfigurable Arrays (CGRAs) provide a compromise between the energy efficiency of ASICs and the fine-grained re-configurability of FPGAs [17, 33, 48]. They have received increased interest in recent past, both as high performance accelerators [47] and for edge embedded computing [6]. Example of commercially available CGRAs include Monza [6], SambaNova [51] and ULP-SRP [26].

A CGRA is typically a two-dimensional grid of Processing Elements (PEs) interconnected by a reconfigurable network. The operation performed by a PE is controlled by software, Similarly, the inter-connectivity of the PEs is also controlled by software, allowing the output of a PE to be connected to the input of another, possibly via other PEs.

A CGRA is used by *mapping* a data flow graph (DFG)—a directed acyclic graph with nodes representing operations and edges representing the flow of data between operations—to the PE grid. Each DFG node is assigned a PE and the network is configured to reflect inter-node connectivity. Multiple instances of the DFG are executed in a pipelined fashion for high *throughput*.

This work focuses on the class of *spatial elastic* CGRAs (Section 2 describes this class and justifies this focus), which for brevity we refer to as CGRAs. This class is typically used to accelerate *parallel* inner-most loops in kernels. The body of such a loop is abstracted into a DFG, which is mapped to a target CGRA for a higher rate of loop iterations executed per cycle.

Nonetheless, there are many kernels that can benefit from CGRA acceleration but have inner-most loops that are *not* parallel, i.e., *serial*, because they have *flow* or *true loop carried dependences* [41]. Examples include loops that perform reductions and first-order linear recurrences, common in scientific computing, signal processing [57], time-series analysis [57], and ML kernels [19].

Loop carried dependences introduce ϕ nodes (Section 3.1) into a DFG. These nodes serve to provide initial values from outside the corresponding loop for the first iteration, then pass through loop carried values for subsequent ones. Loop carried dependences may also introduce cycles in DFGs, which limits their efficient pipelining. Existing approaches for handling loop carried dependences [6, 17, 20, 24, 58, 62, 67] are limited by either: (1) allocating PEs for ϕ nodes, which reduces CGRA utilization and lowers throughput, or (2) requiring extensive hardware support [67].

We introduce a novel mechanism called *FIFO Initialization* (FI) that alleviates the aforementioned limitations. FI pre-loads values directly into the FIFOs of PEs when the CGRA is configured to eliminate the need to allocate PEs toϕ nodes. FI facilitates the efficient pipelining of loops that perform reductions, recurrences through look-ahead expansion (LE)—an optimization previously used for multicores and VLSI circuit design to efficiently pipeline serial loops. Further, it leads to resource-efficient data reuse with the sliding-window optimization (SWO) used for multicores. To the best of our knowledge ours is the first work to utilize these two optimizations on CGRAs, both enabled by FI.

We extend an 8×8 target CGRA to support FI and build compiler support to apply LE and SWO, utilizing FI. We use cycle-accurate simulation to assess throughput and PE usage for several benchmarks with serial loops that commonly appear in scientific computing, machine learning, and signal processing. Evaluation shows that FI combined with LE achieves 9.5× throughput increase on average over state-of-the-art approaches to handling reduction and recurrence

loops. For stencils, FI with SWO realizes data reuse with up to 3.9× reduction in PE usage. Synthesis of the target CGRA in 15nm technology shows that FI introduces only 1.24% area overhead and 4.59% power overhead, demonstrating its efficiency and practicality.

Thus, the contributions of this work are:

- A novel low-cost hardware mechanism to efficiently support serial loops on spatial elastic CGRAs.
- Enabling LE and SWO optimizations for CGRAs.
- A compiler toolchain that leverages FI.

Section 2 introduces background material on CGRAs. Section 3 describes our target serial loops. Section 4 surveys existing approaches that handle serial loops. Section 5 details FI and how it enables LE and SWO. Section 6 describes our compiler support. Section 7 presents the evaluation. Section 8 reviews related work and Section 9 concludes.

2 CGRAs

The design space of CGRAs has been explored in terms of the complexity and interconnection of PEs [18, 28, 37], the configuration approach [31, 43, 49], the execution model [18, 37, 43, 49, 53], and how control flow is handled [3, 13, 21, 25]. Most relevant to this work are a CGRA's configuration approach and execution model.

Spatially-configured, or *spatial*, CGRAs fix the functionality of each PE and PE connectivity during the pipelined execution of a DFG. In contrast, *temporally-configured* (or *temporal*) CGRAs allow the operation performed by a PE and the network configuration to change during execution, possibly on every cycle.

Synchronous CGRAs statically schedule PEs operations and PEs execute in lock step. In contrast, *elastic* CGRAs use FIFO buffers in each PE to implement data-driven execution in which a PE operates only when its input data is available, allowing PEs to execute asynchronously.

In general, PEs receive their data either from a scratchpad memory (e.g., [64]) or from a host's system memory (e.g., [1]). While the latency of access is different for these memory sources, *streaming* is used to deliver data such that at *steady state*, a PE receives data potentially on every cycle, to ensure full utilization of PEs.

We focus on spatial elastic CGRAs because they have lower resource overhead and power consumption compared to synchronous temporal ones [32, 33]. This allows them to be clocked at higher frequencies [32]. Further, elastic CGRAs are better able to tolerate variable memory access latencies and irregular control flow. Finally, they simplify CGRA toolchains by eliminating the need to schedule operations in time [24, 32]. It is no surprise that recent commercialization of CGRAs has adopted spatial elastic execution [6, 51], further motivating us to target this class of CGRAs.

We target the spatial elastic CGRA shown in Fig. 1. It consists of an $n \times m$ array of PEs connected by programmable

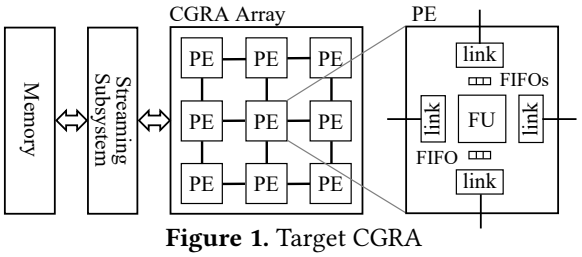


Figure 1. Target CGRA

interconnection links. Each PE contains a Function Unit (FU) that can perform integer, floating point, and logical operations. These operations have different latencies depending on their complexity. The FU is surrounded by PE-local FIFOs for data-driven execution.

PEs along the two vertical boarders perform no computation and are dedicated to IO. They interface to a memory streaming subsystem, which streams data from/to memory at a rate of w words per cycle. There is an initial latency to streaming, denoted by M_s, which is incurred at the start of streaming, and when streaming is interrupted and re-started.

3 Target Loops

Spatial elastic CGRAs typically target parallel inner-most loops that carry no data dependence. However, many applications that can benefit from CGRAs have serial inner-most loops, i.e., ones that *carry dependence*. We focus on loops with *flow* or *true* loop carried dependences, where an iteration writes the same memory location that is read by a subsequent iteration, preventing parallel execution. The earlier access is the *source* of the dependence, while the latter is its *destination*. The dependence *distance* is the number of iterations between the source and destination.

We describe three computational patterns with serial loops that carry flow dependences: *reductions*, *first-order linear recurrences* and *stencils*. In reductions, a loop iteration accumulates a value computed to that computed in the previous iteration. Reductions are common in HPC and AI/ML kernels where dot products, norms and aggregations are dominant. A simple reduction is shown in Fig. 2; elements of `a` are summed into `acc`. This gives rise to a loop carried dependence that serializes the loop.

A linear recurrence (hereafter a recurrence) computes the value of a variable in an iteration as a linear combination of the variable's values computed in prior iterations. Recurrences are common in signal processing, time-series analysis, and large-language models [19, 57]. *We only focus on first-order recurrences*, an example of which is shown in Fig. 3. `res[i]` is a function of `res[i-1]`, giving rise to a loop carried dependence and serializing the loop.

Stencils, including convolutions, are ubiquitous in HPC and ML kernels. Common to them is that the same input is used in multiple consecutive iterations. Fig. 4(a) shows the key loop of a 1-d Jacobi stencil. An element of `a` is read across three consecutive iterations of the loop.

```
float y=0.0f;
for(int i=0;i<n;i++)
  y+=x[i];
```

Figure 2. A reduction

```
y[0]=0.0f;
for(int i=1;i<n;i++)
   y[i]+=c*y[i-1]+x[i];
```

Figure 3. A recurrence

```
for(int i=1;i<n-1;i++)
  b[i]=0.3333f*(a[i-1]
       +a[i]+a[i+1]);
```

(a) Stencil code

```
float reuse1 = a[0];
float reuse2 = a[1];
for(int i=1;i<n-1;i++){
  float reg = a[i+1];
  b[i]=0.3333f*(reuse1
       +reuse2+reg);
  reuse1 = reuse2;
  reuse2 = reg;
}
```

(b) Optimized code

Figure 4. 1-d Jacobi code

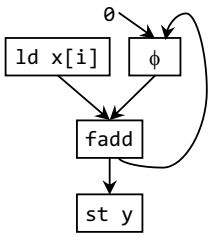


Figure 5. Reduction DFG

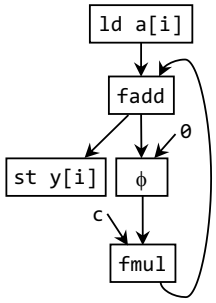


Figure 6. Recurrence DFG

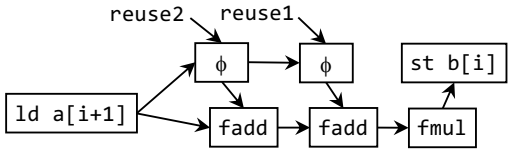


Figure 7. 1-d Jacobi DFG

Reads in a stencil do not give rise to loop carried dependences. However, compilers apply a *sliding-window* (SWO) optimization [9, 10] to eliminate redundant loads. It introduces a *reuse register chain* to store and forward reused values across iterations. In the optimized code for 1-d Jacobi shown in Fig. 4(b), `reuse1` and `reuse2` form the chain: `reuse2`'s value is forwarded to `reuse1` and `a[i]`'s value is forwarded to `reuse2`. Two loop carried dependences arise since register values written in an iteration are read in the next.

3.1 DFG ϕ Nodes and Feedback Paths

Loop carried dependences introduce ϕ *nodes* into a DFG. These nodes serve to (1) provide initial values (from outside the corresponding loop) of the variables that give rise to the dependences for the first iteration, and (2) pass through the loop carried values for subsequent iterations. Thus, ϕ nodes play the same role in a DFG as ϕ instructions in SSA-based intermediate representations [41][1].

Loop carried dependences also introduce *cycles* with associated *feedback paths* in the DFGs of reductions and recurrences. This is because the source of the dependence is a node that is topologically later in the DFG compared to the node of its destination.

Fig. 5 shows the DFG for the reduction of Fig. 2. The ϕ node holds the initial value of `acc` (i.e., 0) for the first iteration and passes through the value computed by the `fadd` for subsequent ones. The loop carried dependence results in a cycle with a feedback path as shown. Without the ϕ node holding the initial value, execution deadlocks because of the circular dependence. Fig. 6 shows the same for the recurrence of Fig. 3.

The DFG of Fig. 4(b)'s stencil is shown in Fig. 7. The register chain is captured by the two ϕ nodes that provide the initial values of `reuse1` and `reuse2` for the first iteration then `a[i+1]` in subsequent ones. No cycles exist in the DFG; the sources of the two dependences (the ϕ nodes) are topologically earlier nodes than their destinations (the `fadd`s).

3.2 Throughput with Feedback Paths

We define the latency of a feedback path L as the total latency of the operations in the cycle formed by this path. The ideal throughput in the presence of a feedback path, T_{ideal}, is

$$T_{ideal} = \min(1\ ,\ \frac{d}{L}) \qquad iterations/cycle, \qquad (1)$$

where d is the distance of the dependence that gives rise to the feedback path.

When $d < L$, the result of the operations on the feedback path are needed d cycles later, which is sooner than they are produced in L cycles. This leads to stalls and thus lower T_{ideal}. For example, in the DFG of Fig. 5, $d = 1$ and L is the sum of the latencies of the ϕ and `fadd` nodes, assumed to be 1 and 3 respectively. Each newly computed value is immediately required by the next cycle, yet, its generated 4 cycles later. This limits the `fadd` to produce an output every 4 cycles, lowering T_{ideal} to 1/4.

When (i.e., $d \geq L$), data is present when it is needed and the feedback path no longer influences T_{ideal}. In the above example, had d been 4, then after an initial latency, the output of the `fadd` is ready at its input on every cycle and $T_{ideal} = 1$.

This notion of feedback path latency resembles the recurrence-constrained minimum initiation interval (*RecMII*) of modulo scheduling for temporal CGRAs [24, 54].

4 Handling Serial Loops

Existing approaches that handle serial loops on spatial elastic CGRAs include: *explicit mapping of ϕ nodes*, *spilling to memory*, *parallel loop unrolling*, and *reduction PEs*.

Explicit mapping of ϕ nodes assigns each ϕ node to its own PE, effectively storing initial and/or loop carried values in the PE. We refer to this approach as *mapped-ϕ* and it is used by several CGRAs [17, 24, 62].

Spilling to memory writes loop carried values to memory and serializes dependent memory operations. It requires the compiler to identify dependent memory operations and

[1] We focus on ϕ nodes resulting from loops as opposed to ones resulting from selection.

to add corresponding edges to the DFG [17]. Hardware support is required for dependent PEs to produce and consume tokens [17]. We refer to this approach as *memory-spill*.

Parallel loop unrolling increases the DFG size linearly to improve throughput by the unroll factor [61, 68]. A reduction or recurrence loop is unrolled multiple times, but only one instance of the reduction or recurrence computation is kept. Results from the unrolled computations are aggregated by a reduction tree [5, 55, 66]. For recurrences, an additional downwards sweep is required after the reduction tree to capture intermediate results [46, 55].

Reduction PEs augment FUs in PEs to support reductions at a throughput of 1 result per cycle. Such support may be added to all PEs, keeping the CGRA homogeneous, or it can be added only to a subset of PEs, making the CGRA heterogeneous. No similar support exits for recurrences.

The above approaches suffer from two key limitations: *inefficient use of resources* and *lower throughput*.

The mapped-ϕ approach dedicates PEs to only store loop carried values in a PE's configuration registers. This wastes PEs and reduces CGRA utilization. Further, memory-spill incurs additional memory writes and reads, making it inferior to mapped-ϕ for the small-distance loop carried dependences of reductions and recurrences. Finally, reduction PEs increase area and power consumption [23, 63]; a single reduction FU can take up to 40% more area than a floating-point ALU [23]. Mitigating this penalty with heterogeneity complicates mapping [4, 11].

More importantly, mapped-ϕ and memory-spill do not address the potentially long feedback-path latencies, which is detrimental to throughput since T_{ideal} linearly drops with longer feedback path latencies (Equation 1 in Section 3.2). Further, with memory streaming, long feedback path latencies results in consuming data at a slower rate than the streaming rate. This saturates PE FIFOs, stalling PEs. Eventually, memory streaming is stalled and the penalty of restarting it is incurred, further lowering throughput.

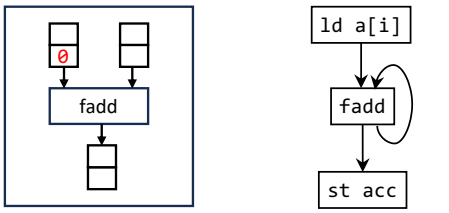
5 FIFO Initialization

FI eliminates ϕ PEs, allows efficient pipelining of reductions and recurrences, and enables resource-efficient data reuse.

5.1 Eliminating ϕ PEs

The ϕ nodes in a DFG serve to provide initial values and pass through loop carried values for variables that give rise to loop carried dependences. FI: (1) stores initial values directly into the FIFOs of PEs to which nodes that read these initial values are mapped, and (2) re-directs the output of nodes that write loop carried values to these FIFOs. This alleviates the need to dedicate PEs to ϕ nodes.

Fig. 8 illustrates FI for the reduction example DFG of Fig. 5. The ϕ node provides the initial value of `acc` coming from outside the loop (i.e., 0) and passes the loop carried value



(a) `fadd` PE FIFOs (b) Optimized DFG

Figure 8. Example of FI

computed by the `fadd`. FI pre-stores the initial value in the FIFO of the PE to which the `fadd` is mapped, as shown in Fig. 8(a). In the the first iteration of the loop, the value in the FIFO is immediately used. Subsequent iterations use the output of the `fadd`, which is now fed back to the same FIFO. This allows the DFG to be optimized as shown in Fig. 8(b), eliminating the ϕ node and thus, the need for a ϕ PE.

FI requires hardware extensions to a PE in order to support the initialization of FIFO entries during PE configuration. This support is described in Section 5.4.

5.2 Efficient Pipelining

The pipelined execution of the optimized DFG in Fig. 8(b) remains limited by the latency of the `fadd` operation. If this latency is 3, $T_{ideal} = 1/3$. In general, T_{ideal} drops for a reduction or a recurrence when the latency of the feedback path L is larger than the dependence distance of the loop d (Equation 1).

This drop can be mitigated by increasing the dependence distance using a technique known as *look-ahead expansion* (LE) [27, 45]. The technique has been used in the context of ASIC circuit design and parallel computing. To the best of our knowledge, it has not been used for CGRAs. Its direct use for CGRAs limits T_{ideal}. We show that FI enables LE to be used effectively, resulting in $T_{ideal} = 1$.

5.2.1 Reduction Look-Ahead Expansion.

A reduction can be expressed as: $y_i = y_{i-1} + x_i$, reflecting a dependence distance of 1. Substituting y_{i-1} with its own reduction yields $y_i = y_{i-2} + x_{i-1} + x_i$, which has a dependence distance of 2, This process can be repeated $L-1$ times to result in the following *expanded* equation:

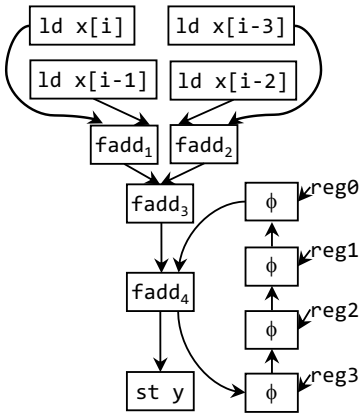
$$y_i = y_{i-L} + x_{i-L+1} + x_{i-L+2} + \cdots + x_{i-1} + x_i, \qquad (2)$$

which reflects a loop with a dependence distance of L. Given a feedback path latency of L, full pipelining is possible and $T_{ideal} = 1$. The use of the expanded reduction requires the corresponding loop to be modified to pre-compute the first $L-1$ terms of the reduction, which is accomplished by peeling the first $L-1$ iterations of the loop.

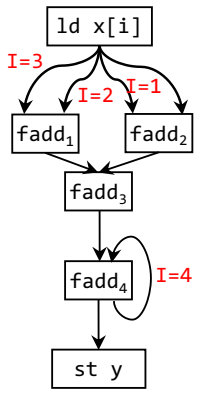
This process of expansion and loop peeling is illustrated for the reduction loop of Fig. 2 and its corresponding DFG of Fig. 5. If the latency of the `fadd` is 3 and that of the ϕ node is 1, $L = 4$ and $T_{ideal} = 1/4$. Fig. 9(a) shows the loop with 3 peeled iterations and the 4 initial values (initial value of the

```
float y;
float reg0=0.0f;
float reg1=reg0+x[0];
float reg2=reg1+x[1];
float reg3=reg2+x[2];
for (int i=3;i<N;++i){
  y=reg0+x[i-3]+
    x[i-2]+x[i-1]+x[i];
  reg0=reg1;
  reg1=reg2;
  reg2=reg3;
  reg3=y;
}
```

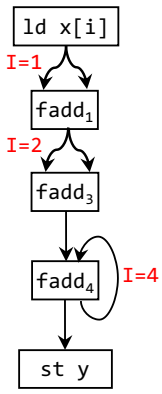
(a) Modified reduction loop



(b) With LE



(c) With FIFO Initialization



(d) With RD

Figure 9. Example of reduction LE

reduction and 3 of the peeled iterations) are pre-computed in `reg0` to `reg3` and used inside the loop.

Fig. 9(b) shows the DFG of the body of the modified loop. The DFG shows the use of four ϕ nodes on the feedback path due to the use of the `reg` variables. It also reflects the need to compute the *partial sums* of $x(i-3)+x(i-2)+x(i-1)+x(i)$, which did not exist in the original loop. This is done with 3 additional `fadds`: $\texttt{fadd}_1$, $\texttt{fadd}_2$ and $\texttt{fadd}_3$, as shown. These `fadds` require additional memory loads for `x[i − 3]`, `x[i − 2]`, `x[i − 1]` and `x[i]`.

In general, partial sums are computed using additional arithmetic nodes and corresponding loads, increasing DFG size. This increase can be significant, limiting the practicality of the direct use of LE on CGRAs. More seriously, although LE increases the dependence distance, it also proportionally increases latency with the addition of ϕ nodes on the feedback path. Assuming the latency of a ϕ node is 1,

$$T_{ideal} = \frac{L}{2 \times L - 1}, \tag{3}$$

when LE increases the dependence distance to L, making $T_{ideal} \approx 1/2$.

FI alleviates the above limitations to make the use of LE practical on CGRAs. This is illustrated in Fig. 9(c), which shows the modified DFG with FI applied. A label $I = a$ next to an edge reflects a initialized entries in the FIFO associated with the edge. The ϕ nodes eliminated by initializing 4 entries in FIFO of $\texttt{fadd}_4$ to initial `reg0` to `reg3` values (hence $I = 4$ label on the edge). The latency on the feedback path is now $L - 1$, or 3, resulting in $T_{ideal} = 1$.

Further, a single load node is now used to read the reduction input $x(n)$, starting with $n = 3$, fanning out to 4 consumers. The FIFO associated with the edge labeled $I = 1$ is initialized to $x[2]$; with label $I = 2$ is initialized to $x[2]$ and $x[1]$; and with $I = 3$ to $x[2]$, $x[1]$ and $x[0]$.

Nonetheless, the arithmetic nodes that compute the partial sums of the x terms remain. The number of such nodes is in general $O(L)$, where L is the number of LE steps. Such increase may exhaust available PEs. Further, the high fanout of the top load node complicates DFG mapping. This can be addressed by a technique known as *recursive doubling* (RD) [27, 45]. It exploits the fact that partial sums evaluated in one iteration are re-computed in subsequent ones. For example, in Fig. 9(c), when $i = 3$, $\texttt{fadd}_1$ computes $x[0]+x[1]$ and $\texttt{fadd}_2$ computes $x[2]+x[3]$. Two iterations later, when $i = 5$, $\texttt{fadd}_1$ computes $x[2]+x[3]$ and $x[4]+x[5]$; i.e., $\texttt{fadd}_2$ re-computes what $\texttt{fadd}_1$ computed 2 iterations earlier.

RD pre-computes the first two partial sums to initialize the FIFO of $\texttt{fadd}_3$ with $x[0]+x[1]$ and $x[1]+x[2]$ in the example. $\texttt{fadd}_1$ and $\texttt{fadd}_2$ now compute the same partial sum $x[i-1]+x[i]$ in any iteration i, starting with $i = 5$. Only one `fadd` node is needed along with the initialization of one of its FIFOs. The $x[2]+x[3]$ and $x[3]+x[4]$ sums are pre-computed before the loop, used to initialize the FIFO of $\texttt{fadd}_3$, and the loop start value is $i = 5$. The resulting DFG after RD is shown in Fig. 9(d).

In general, RD reduces the number of partial sum compute nodes from $O(L)$ to $O(logL)$. Thus, henceforth, when we refer to using LE, we imply that RD is also used.

5.2.2 Recurrence Look-Ahead Expansion.

Recurrences are expanded similar to reductions. For example, the linear recurrence in Fig. 3 can be expressed as $y_i = \alpha y_{i-1} + x_i$. An L-step LE rewrites the recurrence as:

$$y_i = \alpha^{L+1} y_{i-L-1} + \sum_{k=0}^{L} \alpha^k x_{i-k}, \tag{4}$$

with coefficients α^k (α raised to the power k) and $d = L$. The first $L - 1$ iterations of the loop must be peeled. However, without FI it suffers from the same throughput and memory access limitations as a reduction.

In the example of Fig. 6, the feedback path has $d = 1$ and assuming $L = 8$, $T_{ideal} = 1/8$. After LE, the number of load nodes increases from 1 to 8 and an additional 8 `fmul` and 7 `fadd` nodes are needed to compute the recurrence's

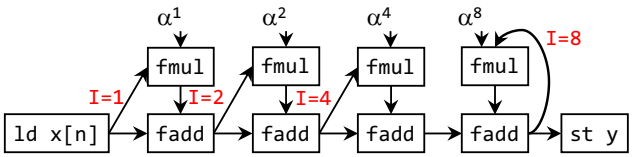


Figure 10. Recurrence example with FI and RD

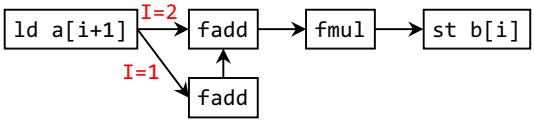


Figure 11. 1-d Jacobi DFG with FI

partial sums $\sum_{k=0}^{L} \alpha^k x_{i-k}$ introduced by peeling. FI and RD eliminate these additional nodes, as shown in Fig. 10. The DFG has no ϕ nodes with $d = 8$, resulting in $T_{ideal} = 1$. A single load is used with the FIFOs of its successors initialized as shown in the figure. The number of additional arithmetic nodes is reduced from 15 to 6.

5.2.3 Impact of FI. FI eliminates ϕ nodes, freeing PEs for arithmetic operations and reducing feedback path latencies to account only for arithmetic operations. ***This allows the combined use of LE and FI to achieve $T_{ideal} = 1$***, up from d/L. The node overhead to compute partial sums is reduced from $O(L)$ to $O(\log L)$, which is consistent with prior work [27, 45], ***However, this only achievable on CGRAs with FI eliminating ϕ nodes***.

5.3 Resource-Efficient Data Reuse

SWO results in loop carried dependences that do not introduce cycles but introduce ϕ nodes, as shown in Fig. 7. These nodes reduce CGRA utilization but can be eliminated by FI, as shown in Fig. 11. The initial values of `reuse2` and `reuse1` are stored directly into the `fadd` nodes' FIFOs, alleviating the need for the two ϕ nodes/PEs. The FIFO associated with the edge labeled $I = 1$ is initialized with `reuse2`; the one labeled $I = 2$ with `reuse2` and `reuse1`.

For 2-d stencils, reuse exists not only within rows but also across columns. For example, the bottom-right corner of a 3×3 stencil window of an input array is reused 8 times across subsequent window positions as the window slides horizontally and vertically over the array. Exploiting reuse across columns can result in long reuse chains, depending on the size of the stencil window and the size of the array. The number of initialized values with FI may exceed the capacity of PE FIFOs and require the dedication of PEs just for their FIFO capacity (Section 6). This overhead can be alleviated by tiling and flattening to reduce reuse distance, as described in the Supplementary Material.

5.4 PE Extensions

Fig. 12 shows the parts of the PE relevant to FI. The external inputs to a PE are from neighboring ones to the east (E). west (W), north (N) and south (S). These inputs are routed to the

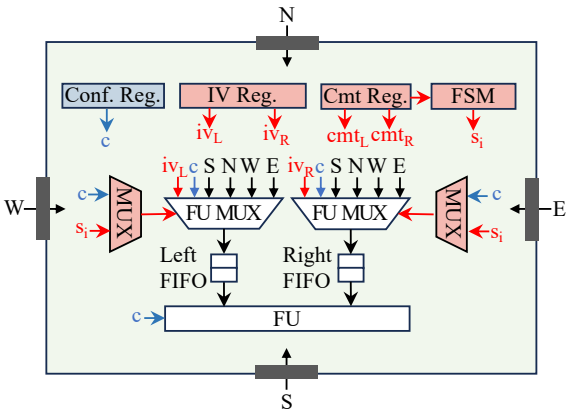


Figure 12. PE Extensions

two FU multiplexers (FU MUXs) that feed the FIFOs of the FU inputs (actual connections are not shown to simplify the figure). The Conf. register holds the configuration bitstream for the PE, including a constant input (c). The bitstream determines which FU MUX input is fed to the FIFOs and what functionality the FU implements.

A PE is extended by adding: a double-word Initial Value (IV) register, a commit value (Cmt) register, two MUXs and a finite state machine (FSM), as shown in red in Fig. 12. IV holds initial values for the left and right FIFOs, iv_L and iv_R, each used as an additional FU MUX input. Cmt is a two bit register, where cmt_L and cmt_R indicates whether respectively iv_L and iv_R are to be pushed onto the corresponding FU FIFO. Two additional MUXs select iv_L and iv_R during initialization.

The FSM orchestrates initialization, starting after the PE's configuration bits are loaded into Conf. For each FIFO entry to be initialized, the initial values are loaded into IV and Cmt by shifting them through the existing inter-PE communication links. Based on the Cmt bits, the FSM selects the iv_L and iv_R inputs to the FU MUXs and pushes the respective values onto the FIFOs. This process is repeated for as many times as the maximum number of initialized values in any FIFO.

FI adds minimal hardware to a PE, in the form of registers, MUXs and additional inputs to existing MUXs. It does increase configuration time since FIFO initial values must repeatedly be shifted into each PE.

6 Compiler Support

We build a compilation flow around MLIR [30] and LLVM [29] to support FI, shown in Fig. 13. The flow takes as input: (1) C/C++ code with CGRA target loops identified by pragmas and (2) target CGRA parameters, including latencies of operations, FIFO depths, etc. Polygeist [40] is used to compile the code into MLIR's affine dialect [38] IR.

Preprocessing. We use MLIR's scalarization to convert reductions from a memory representation to register accumulations, specified as MLIR iteration arguments [39]. We implement a dependence analysis to identify reductions and recurrences, representing them in a unified form: VarA =

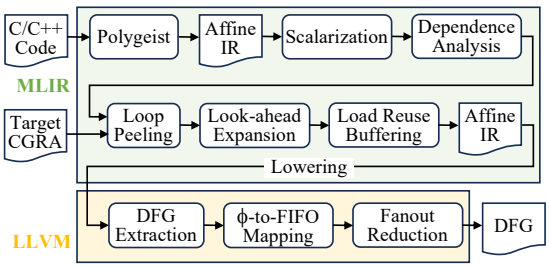


Figure 13. Overview of the compilation flow

$c \cdot \text{VarA} \oplus f(i)$, where c is a constant, $\oplus$ is a binary operator and $f(i)$ is a function of the loop induction variable i. This form allows uniform handling of both reductions and recurrences, with $c = 1$ for reductions.

Loop peeling. CGRA parameters are used to determine L_{max}, the maximum feedback latency in each loop, which gives the number of LE steps $S = 2^k - 1$ where $k = \lceil log_2(L_{max}) \rceil$. This ensures that S is a power of 2, which is necessary for RD [27, 45]. The first S iterations of each loop are peeled, and the initial values needed for FI are computed and passed as iteration arguments.

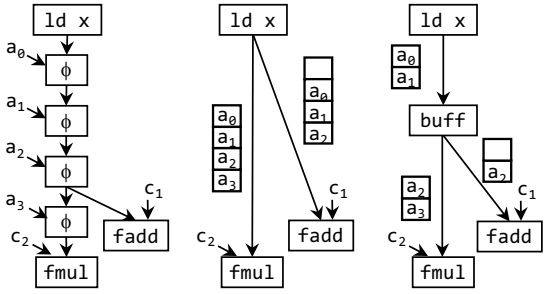
LE application. Each loop body is then modified to implement LE by adding the necessary multiply-add chain of LE, replacing the original recurrence operation with the resulting look-ahead value, and rewriting the loop yield so the partial sums and recurrence values are shifted forward. MLIR canonicalization is used to simplify the expanded operations, particularly to remove redundant multiplications by 1 in reductions. Initial values are stored as explicit iteration arguments to allow their easy identification during DFG generation since these arguments are lowered to ϕ instructions.

SWO application. We perform data reuse analysis to identify groups of memory accesses that give rise to reuse. The reuse distances are calculated for each group. We replace all accesses in a group by a register chain fed from the access with the longest reuse distance in the group. The chains are also represented as iteration arguments of loops in MLIR.

DFG extraction. The affine IR is lowered to LLVM IR using the standard MLIR lowering passes [40]. The LLVM IR is passed to several custom LLVM passes that extract DFGs of bodies of target loops, following common approaches [16, 52, 68]. However, we initially map ϕ operations in the IR to ϕ nodes in the DFG, possibly with feedback paths.

ϕ-node processing. The extracted DFGs are then post processed to identify ϕ nodes and ϕ-node chains and fuse them with their user nodes as initialized FIFO values, obtained from the lowered MLIR IR. Initial values are duplicated for each user of a ϕ-node chain when there are multiple users of the chain. This process is illustrated in Fig. 14(a)s, which shows an example DFG with its ϕ nodes. The fused DFG is shown in Fig. 14(b). The initialized values are shown as FIFO entries next to associated edges.

Buffer nodes. When the number of initialized values exceeds a FIFO's capacity, buffer nodes are added to the DFG to store initial values, as shown in Fig. 14(c). Such nodes need not be duplicated for multiple users since they can be shared.



(a) DFG (b) Optimized DFG (c) Use of buffer node

Figure 14. Mapping ϕ nodes to FIFOs

Table 1. PE operation latencies (in cycles)

Operation	Latency	Operation	Latency
fadd	3	fdiv	6
fsub	3	fcomp	2
fmul	4	exp	6
integer/logical	1	other (inc. phi)	1

Fanout reduction. Fanout reduction is performed on nodes with high fanout by inserting fanout-reduction nodes [68] to promote successful DFG mapping. This is important for SWO since one load can fanout to a large number uses.

7 Evaluation

Our evaluation of FI aims to answer the following research questions (RQs):

- **RQ1**. How does FI with LE improve throughput of reductions and recurrences compared to two state-of-the-art approaches: mapped-ϕ and parallel loop unrolling?
- **RQ2**. To what extent does FI reduce PE usage in stencils with SWO?
- **RQ3**. What is the impact of key CGRA architectural parameters on FI's benefit?
- **RQ4**. What are area/energy overheads of FI support?

Target CGRA Parameters. FI is evaluated on an 8×8 target CGRA (Section 2). Data is streamed data to/from memory at the rate of $w = 16$ words/cycle. The default latency to access memory is $M_s = 400$ cycles. FIFO depth is $L = 8$ and the latency of inter-PE communication is 1 cycle. The latencies of PE operations are shown in Table 1. These CGRA parameters are typical of those used in prior work [1, 14, 24, 52, 56, 65]. The impact of key ones is explored Section 7.3.

Simulator. We utilize a cycle-accurate simulator of the target CGRA, extended with FI support. The simulator models cycle-by-cycle execution of pre-mapped DFGs. In each cycle, it dequeues from the FIFOs if a PE's FU operands are present and the FU is not stalled. It issues the operation and puts the operation into a pipeline queue to model the operation's latency. When an input FIFO is filled up, the PE that feeds into this FIFO is also stalled. Should the stalling back-propagates to the memory streaming subsystem, the streaming is stalled, incurring M_s when restarted.

Table 2. Evaluation benchmarks

Benchmark	Category	Description
Recurrences		
ema [35]	Sig. Proc.	Exponential moving average
decay [44]	Sig. Proc.	Exponential decay filter
hp_filter [35]	Sig. Proc.	High-pass filter
m_sgd [59]	ML	SGD with momentum update
w_avg [44]	Sig. Proc.	Weighted sum recurrence
Reductions		
gemm [50]	Lin. Alg.	Matrix multiplication
mvt [50]	Lin. Alg.	Matrix-vector product
g_cdf [2]	Stat.	Gaussian CDF computation
sep [34]	Phys.	Spring energy profile
fd [15]	Phys.	Fluid dynamics simulation
Stencils		
conv2d [50]	Img. Proc.	3×3 2D convolution
conv5x5 [22]	Img. Proc.	5×5 2D convolution
jacobi2d [50]	Num.	Jacobi 2D stencil

The simulator accounts for cycles spent on peeled iterations, configuration, and FI value initialization. These represent critical time spent on non-CGRA execution and are important for a realistic evaluation of FI. This approach of cycle-accurate simulation with pre-defined architectural parameters is well-suited for CGRA evaluation due to the largely deterministic nature of spatial dataflow execution, and is commonly adopted by prior work [42, 62].

Metrics. We assess benefit with three metrics: (1) *throughput* (T) in iterations per cycle, which reflects raw performance; (2) the number of PEs used N_{PE} where a smaller N_{PE}, leads to lower energy consumption since unused PEs can be power-gated [6, 17]; and (3) throughput per PE (T_{PE}), defined as T/N_{PE}, and it captures performance per utilized PE.

Benchmarks. We use the benchmarks shown in Table 2. They are drawn from Polybench [50], prior work [35] and common kernels in signal processing and scientific computing. The default trip count of inner most loops is 10000. The impact of this trip count is explored in Section 7.3.

7.1 Reductions and Linear Recurrences

Fig. 15 depicts the throughput T for the reduction and recurrence benchmarks. It compares T with the use of mapped-ϕ approach, mapped-ϕ with LE, FI without LE and FI with LE. Several conclusions can be drawn from the figure.

First, the use of mapped-ϕ alone delivers minimal throughput, averaging 0.1 iterations per cycle. ***FI with LE delivers significant throughput improvements*** over mapped-ϕ. Fig. 15(a) shows that FI with LE improves T by up to 12.7× over mapped-ϕ and by 9.5× on average. Similarly, Fig. 15(c) shows that FI with LE also achieves 6.2× better T_{PE} on average (up to 9.8×) compared to mapped-ϕ.

Second, ***FI is the key enabler for LE***. FI with LE achieves 3.9× higher throughput than mapped-ϕ with LE. Although mapped-ϕ with LE does provide improvement over mapped-ϕ (1.3–4.3×), it requires 2.4× more PEs on average compared to FIFO initialization (41 PEs vs. 17 PEs), as shown in Fig. 15(b), revealing a critical limitation of mapped-ϕ. Consequently, FI with LE achieves 8.8× higher T_{PE} on average, as Fig. 15(c) shows. This is because LE without FI results in ϕ node chains, as described in Section 5, increasing PE usage and limiting throughput.

Finally, FI alone provides throughput improvements, although more modest, ranging from 1.1× to 1.4× over mapped-ϕ. This improvement comes from eliminating ϕ nodes and reducing the feedback-cycle latency.

Fig. 16 compares the throughput of FI with LE to that of mapped-ϕ with and without parallel loop unrolling. The unroll factor used is the maximum that keeps the DFG size less the target CGRA size. The `fd` benchmark is too large to unroll for the target CGRA.

Fig. 16(a) shows that parallel loop unrolling improves the throughput of mapped-ϕ by up 8.2× and by 6.2× on average. Nonetheless, for most benchmarks, this improved throughput remains below that of FI with LE, even with FI with LE using fewer PEs as Fig. 16(b) shows. FI with LE uses only 14 PEs on average compared to 49 PEs for mapped-ϕ with unrolling (3.5× fewer).

Most importantly, combining unrolling with FI and LE achieves the highest T across all benchmarks; 4.04× on average speedup over mapped-ϕ with unrolling and 2.85× on average over FI and LE without unrolling, substantially outperforming either in isolation. This reflects that ***FI with LE and parallel loop unrolling are synergetic***, combining them together achieves the best T.

Nonetheless, FI with LE remains the most efficient approach when taking PE usage into account. Fig. 16(c) reveals that ***FI with LE achieves the highest*** T_{PE}, with an average improvement of 5.74× over mapped-ϕ with unrolling alone and 1.58× over unrolled FI with LE. This is because LE achieves high throughput with a logarithmic increase in PE count, while parallel unrolling increases PE usage linearly.

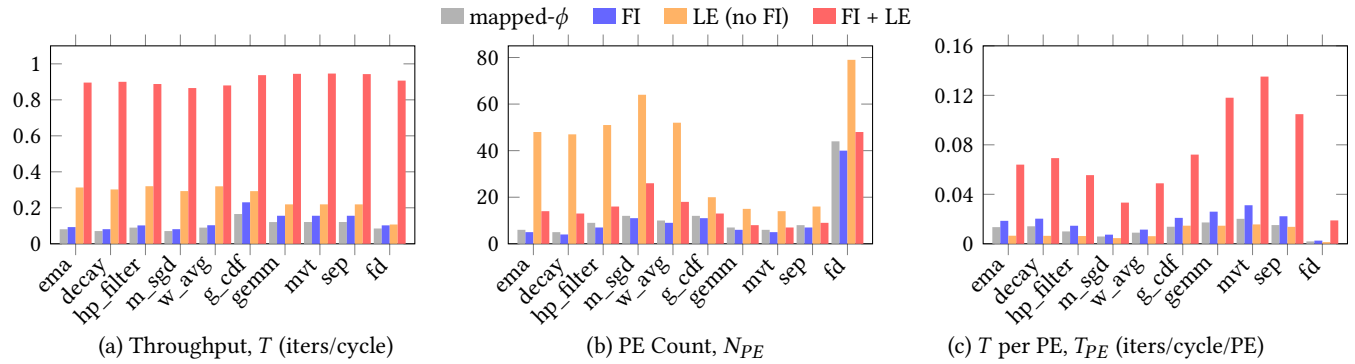
Table 3. Result of applying SWO optimization

Benchmark	Loads	mapped-ϕ		FIFO-8		FIFO-16	
		Size	Reuse nodes	Size	Reuse nodes	Size	Reuse nodes
jacobi2d	5⟶1	39	32	10	3	8	1
conv2d	9⟶1	34	18	20	4	19	3
conv5x5	25⟶1	120	68	68	16	64	12

7.2 Data Reuse for Stencils

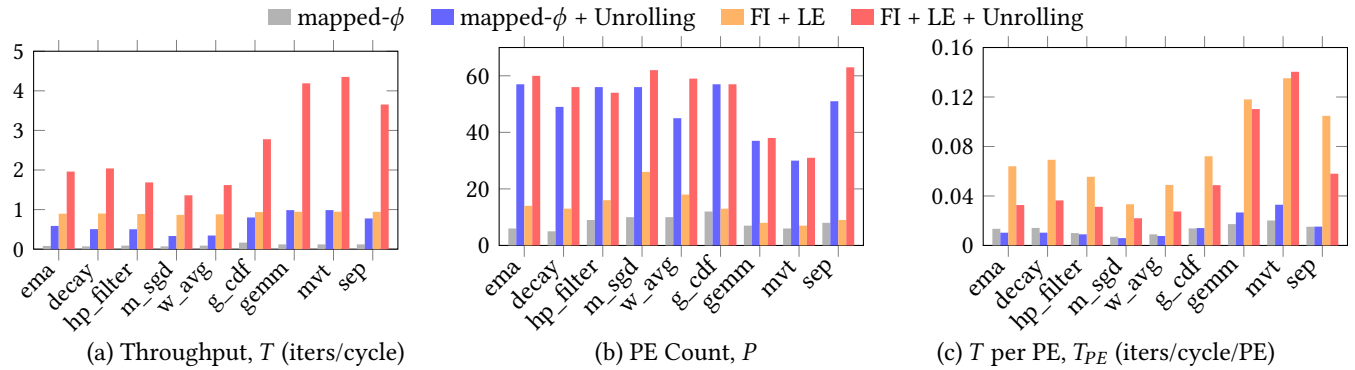
Table 3 shows the reduction in the number of loads, the resulting DFG sizes and the number of reuse nodes for the stencil benchmarks after applying SWO for both mapped-ϕ and FI (for FIFO depths of 8 and 16).

The table shows that SWO reduces the number of load to 1 for both approaches, but FI dramatically reduces the number of nodes, and thus, PEs. For `jacobi2d`, the DFG size is reduced from 39 nodes to 10 nodes compared to mapped-ϕ—a 3.9× reduction—while achieving the same 5⟶1 load reduction. For `conv5x5`, the reduction is from 120 to 68 nodes



(a) Throughput, T (iters/cycle) (b) PE Count, N_{PE} (c) T per PE, T_{PE} (iters/cycle/PE)

Figure 15. Performance comparison for reductions and linear recurrences



(a) Throughput, T (iters/cycle) (b) PE Count, P (c) T per PE, T_{PE} (iters/cycle/PE)

Figure 16. Comparison with parallel loop unrolling

(1.8×). Larger FIFO sizes (16 vs. 8) provide additional modest reductions by eliminating buffer PEs (Section 6).

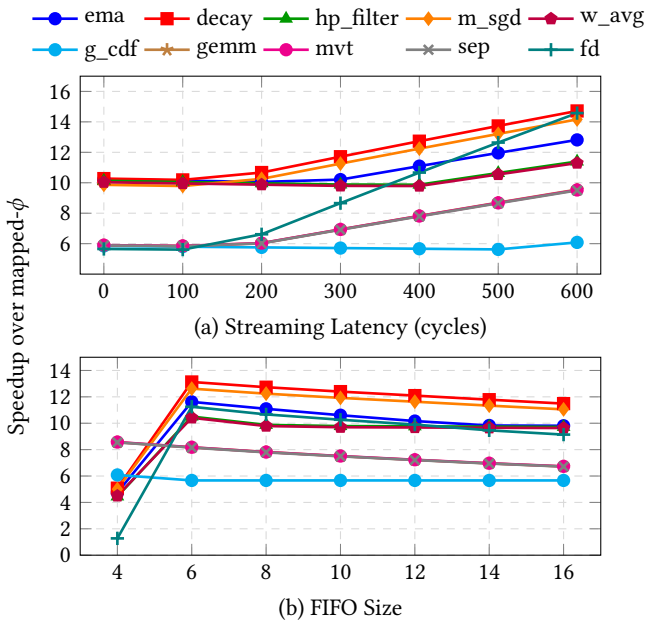
These results demonstrate that ***FI makes the SWO optimization more resource efficient***. Furthermore, when the use of mapped-ϕ results in nodes that exceed a target CGRA capacity, ***FI makes the optimization feasible***.

7.3 Impact of Parameters

Impact of streaming latency. Fig. 17(a) shows the improvement in throughput by using FI with LE over using mapped-ϕ, for streaming latencies M_s varying from 0 to 600 cycles, while keeping other parameters fixed. The figure shows that ***FI delivers throughput improvement over mapped-ϕ across the entire range of M_s***. Indeed, as M_s increases, the improvement delivered by FI with LE increases. This due to FI eliminating the stalls that arise from the ϕ nodes on feedback paths. The stalls force restarts of streaming, incurring a penalty of M_s for each restart (Section 4).

When $M_s = 0$, there are two groups of benchmarks with speedups of 10× and 6× respectively. These speedups reflect the inherent benefit of LE over mapped-ϕ, which is to compensate for the feedback path latencies without the influence of streaming stalls.

Impact of FIFO depth. Fig. 17(b) shows throughput improvement of FI with LE over mapped-ϕ, for FIFO depth ranging from 4 to 16 (keeping other parameters fixed). When



(a) Streaming Latency (cycles)

(b) FIFO Size

Figure 17. Impact of M_s and FIFO depth on throughput

the depth increases from 4 to 6, FI exhibits better improvement because FIFO depth of 4 is often insufficient to fully buffer the data required for LE, leading to multiple buffer PEs and thus higher feedback path latency. However, this improvement drops or flattens beyond 8 entries. This is because

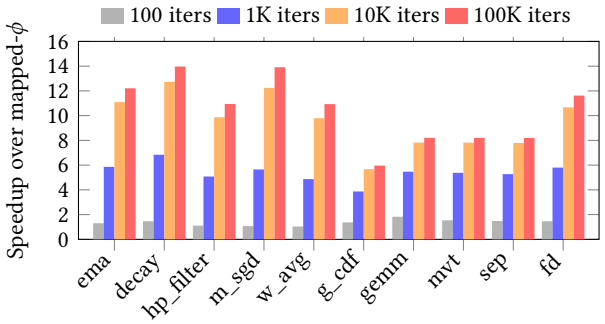


Figure 18. Impact of loop trip count on throughput

Table 4. Area/Power without and with FI

Metric	Without	With	Overhead
Total PE Area (μm^2)	499,537	505,748	1.24%
Combinational Area (μm^2)	270,497	273,510	1.11%
Sequential Area (μm^2)	229,040	232,238	1.40%
Total Power (mW)	122.97	128.62	4.59%
Dynamic Power (mW)	106.25	111.69	5.13%
Leakage Power (mW)	16.73	16.95	1.32%

larger FIFO sizes take longer to fill up, which reduces the number of stalls propagated back to the memory streaming. Therefore, mapped-ϕ performance improves, reducing the relative improvement of FI with LE.

Impact of trip count. Fig. 18 shows the improvement in T of FI with LE over mapped-ϕ for: 100 (mini), 1,000 (small), 10,000 (medium), and 100,000 (large) trip counts. FI with LE is effective regardless of the trip count: on average 1.4×, 5.4×, 9.5×, and 10.4× for mini, small, medium and large trip counts respectively. Its effectiveness increases with higher trip counts since configuration overhead is better amortized with longer running loops.

7.4 Area and Power Overhead of FI

We use the Synopsys Design Compiler [60] and the NanGate 15nm OCL library [36] to synthesize our target CGRA with and without FI support. Both instances are clocked at 1 GHz. Table 4 shows the area and power of each instance and the overhead of FI. The table shows a minimal increase in area of 1.24%, stemming mainly from the additional registers, control logic and multiplexers. The power overhead of 4.59% is similarly modest. These small overheads are justified by the significant performance improvements enabled by FI.

8 Related Work

CGRAs and their associated software have been extensively studied in prior work; Liu el al. [33] and Li el al. [32] provide extensive surveys. The bulk of this work, particularly that on spatial elastic CGRAs, targeted parallel inner-most loops with only a handful of attempts to address serial loops [6, 17, 24, 58]. They share the use of mapped-ϕ or memory-spill [58], with the goal of mainly improving programmability.

Nonetheless, there is work to address the performance of serial loops. UE-CGRA [62] accelerates feedback paths of serial loops by mapping nodes on these paths to PEs with higher frequency. It results in prolonged feedback paths with high latencies, yielding modest performance gains. DR-CGRA [20] supports serial loops by executing loop iterations as parallel threads and mapping dependences to direct inter-thread communication utilizing dedicated hardware support. While this approach avoids spilling back to memory, it requires substantial hardware overhead [20] and does not address feedback path latencies. In contrast, FI combined with LE effectively reduces feedback path latency with minimal hardware support. It retains programmability with significant performance improvements.

The acceleration of reductions and recurrences has been explored for parallel computing, GPUs and VLSI circuit design [7, 8, 27, 45, 46]. Kogge and Stone [27] are first to propose LE. Parhi and Messerschmitt [45, 46] build upon this work and combine it with parallel loop unrolling for VLSI circuit design. Blelloch [7, 8] proposes work-efficient parallel scans for recurrences on multicores and GPUs.

Exploiting data reuse for stencils on CGRAs and FPGAs has been studied. Liao el al. [32] utilize inter-PE connection registers for data reuse but only for small reuse distances. SODA [12] automates exploiting reuse on FPGAs using on-chip memory buffers. We exploit data reuse at potentially large reuse distances and using existing PE FIFOs, alleviating the need for area and power hungry memory buffers.

To the best of our knowledge, ***ours is the first use of LE and SWO on CGRAs***. Direct use of prior work on these optimizations on CGRAs has inefficiencies, which FI alleviates.

9 Concluding Remarks

FI is a novel mechanism that pre-loads initial values into PE FIFOs during the configuration of spatial elastic CGRAs. It eliminates the need to dedicate PEs for ϕ nodes in DFGs of serial loops. It reaps the full benefits of two optimizations: LE for efficient pipelining of reductions and recurrences, and SWO for resource-efficient data reuse in stencils.

Evaluation with a target CGRA and associated compiler support demonstrates that FI achieves 9.5× average throughput improvement over state-of-the-art mapped-ϕ. FI complements parallel loop unrolling, and combining them yields 4.04× average speedup over unrolling alone and 2.85× average speedup over LE alone. FI reduces DFG sizes by up to 3.9× in stencils. These benefits come at area and power overheads of only 1.24% and 4.59%, respectively, demonstrating that FI is both effective and practical.

This work can be extended in several directions. FI and its compiler support can be improved to handle higher-order linear recurrences, non-linear recurrences, and more complex forms of loop carried dependences. FI can be further leveraged by exploring loop transformations such as tiling, fusion and loop interchange to increase data reuse opportunities. Finally, FI can be explored for temporal CGRAs.

References

[1] Mahdi Abbaszadeh, Tarek S Abdelrahman, Reza Azimi, Tomasz S Czajkowski, and Maziar Goudarzi. 2023. Efficient Data Streaming for a Tightly-Coupled Coarse-Grained Reconfigurable Array. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 435–443.

[2] Milton Abramowitz and Irene A Stegun. 1965. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*. Vol. 55. Courier Corporation.

[3] Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava, and Reiley Jeyapaul. 2018. LASER: A hardware/software approach to accelerate complicated loops on CGRAs. In *Design, Automation and Test in Europe Conf.* 1069–1074.

[4] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: a systematic framework for heterogeneous CGRA realization. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. 918–932.

[5] Leslie Barron and Tarek S Abdelrahman. 2018. User-transparent translation of machine instructions to programmable hardware. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 7–14.

[6] Nathan Beckmann, Brandon Lucia, Graham Gobieski, Tony Nowatzki, Thomas Jackson, Guénolé Lallement, Keyi Zhang, Amolak Nagi, Atharv Sathe, and Harsh Desai. 2024. Monza: An Energy-Minimal, General-Purpose Dataflow SoC for the Internet of Things. *IEEE Micro* (2024), 1–9. https://doi.org/10.1109/MM.2024.3426611

[7] Guy E Blelloch. 1990. Prefix sums and their applications. (1990).

[8] Guy E Blelloch, Siddhartha Chatterjee, and Marco Zagha. 1995. Solving linear recurrences with loop raking. *J. Parallel and Distrib. Comput.* 25, 1 (1995), 91–97.

[9] David Callahan, Steve Carr, and Ken Kennedy. 1990. Improving register allocation for subscripted variables. *ACM Sigplan Notices* 25, 6 (1990), 53–65.

[10] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality. *ACM SIGPLAN Notices* 29, 11 (1994), 252–262.

[11] Sichao Chen, Chang Cai, Su Zheng, Jiangnan Li, Guowei Zhu, Jingyuan Li, Yazhou Yan, Yuan Dai, Wenbo Yin, and Lingli Wang. 2024. Hier-CGRA: A Novel Framework for Large-scale CGRA with Hierarchical Modeling and Automated Design Space Exploration. *ACM Trans. Reconf. Technol. Syst.* 17, 2 (2024).

[12] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the International Conference on Computer-Aided Design*. 1–8.

[13] S. Alexander Chin and Jason H. Anderson. 2018. An Architecture-Agnostic Integer Linear Programming Approach to CGRA Mapping. In *Design Automation Conference*. 1–6.

[14] Florent De Dinechin, Bogdan Pasca, and E Normale. 2011. Custom arithmetic datapath design for FPGAs using the FloPoCo core generator. *Design & Test of Computers, IEEE* 28, 4 (2011), 18–27.

[15] Joel H Ferziger, Milovan Perić, and Robert L Street. 2019. *Computational methods for fluid dynamics*. springer.

[16] Xuchen Gao, Yunhui Qiu, Yuan Dai, Wenbo Yin, and Lingli Wang. 2024. A CGRA Front-end Compiler Enabling Extraction of General Control and Dedicated Operators. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 799–804.

[17] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–564. https://doi.org/10.1109/MICRO56248.2022.00046

[18] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (2012), 38–51.

[19] Albert Gu and Tri Dao. 2024. Mamba: Linear-time sequence modeling with selective state spaces. In *First conference on language modeling*.

[20] Elad Hadar and Yoav Etsion. 2024. DR-CGRA: Supporting Loop-Carried Dependencies in CGRAs Without Spilling Intermediate Values. *arXiv preprint arXiv:2405.17365* (2024).

[21] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. 2013. Power-Efficient Predication Techniques for Acceleration of Control Flow Execution on CGRA. *ACM Trans. Archit. Code Optim.* 10, 2 (2013).

[22] Bernd Jähne. 2005. *Digital image processing*. Springer.

[23] Edin Kadric, Paul Gurniak, and André DeHon. 2016. Accurate parallel floating-point accumulation. *IEEE Trans. Comput.* 65, 11 (2016), 3224–3238.

[24] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[25] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2019. 4D-CGRA: Introducing Branch Dimension to Spatio-Temporal Application Mapping on CGRAs. In *Int'l Conf. on Computer-Aided Design*. 1–8.

[26] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. 2014. Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7, 3 (2014), 1–15.

[27] Peter M Kogge and Harold S Stone. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers* 100, 8 (1973), 786–793.

[28] Philipp Käsgen, Mohamed Messelka, and Markus Weinhardt. 2021. HiPReP: High-Performance Reconfigurable Processor - Architecture and Compiler. In *Int'l Conf. on Field-Programmable Logic and Applications*. 380–381.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[30] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[31] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. 2009. FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In *Int'l Conf. on Field-Programmable Technology*. 376–379.

[32] Zhaoying Li, Dhananjaya Wijerathne, and Tulika Mitra. 2024. Coarse-grained reconfigurable array (CGRA). *Handbook of Computer Architecture* (2024), 465–505.

[33] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei. 2019. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.* 52, 6 (2019).

[34] Tiantian Liu, Adam W Bargteil, James F O'Brien, and Ladislav Kavan. 2013. Fast simulation of mass-spring systems. *ACM Transactions on Graphics (TOG)* 32, 6 (2013), 1–7.

[35] Sepideh Maleki and Martin Burtscher. 2018. Automatic Hierarchical Parallelization of Linear Recurrences. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 128–138.

[36] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open Cell

Library in 15nm FreePDK Technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. 171–178.

[37] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application*. 61–70.

[38] MLIR Developers. 2020. MLIR Affine dialect. https://mlir.llvm.org/docs/Dialects/Affine/

[39] MLIR Developers. 2024. MLIR SCF dialect. https://mlir.llvm.org/docs/Dialects/SCFDialect/

[40] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12.

[41] Steven Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[42] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. 2016. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 27–39.

[43] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Int'l Symp. on Computer Architecture*. 416–429.

[44] Alan V Oppenheim. 1999. *Discrete-time signal processing*. Pearson Education India.

[45] Keshab K Parhi and David G Messerschmitt. 1989. Pipeline interleaving and parallelism in recursive digital filters. I. Pipelining using scattered look-ahead and decomposition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37, 7 (1989), 1099–1117.

[46] Keshab K Parhi and David G Messerschmitt. 1989. Pipeline interleaving and parallelism in recursive digital filters. II. Pipelined incremental block filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37, 7 (1989), 1118–1134.

[47] Artur Podobas, Kentaro Sano, and Jason Anderson. 2022. First Int'l Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC). In *IEEE Int'l Parallel and Distributed Processing Symp. Workshops*. 625–626.

[48] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective. *IEEE Access* 8 (2020), 146719–146743.

[49] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A Template-based Framework for Exploring Coarse-Grained Reconfigurable Architectures. In *Int'l Conf. on Application-specific Systems, Architectures and Processors*. 1–8.

[50] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.

[51] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Xiaoyan Li, et al. 2024. Sambanova sn40l: Scaling the ai memory wall with dataflow and composition of experts. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1353–1366.

[52] Omar Ragheb, Stephen Wicklund, Matthew Walker, Rami Beidas, Adham Ragab, Tianyi Yu, and Jason Anderson. 2024. CGRA-ME 2.0: A research framework for next-generation CGRA architectures and CAD. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 642–649.

[53] Omar Ragheb, Tianyi Yu, Rami Beidas, and Jason Anderson. 2022. Elastic Multi-Context CGRAs. In *Int'l Parallel and Distributed Processing Symp. Workshops*. 655–662.

[54] B Ramakrishna Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*. 63–74.

[55] Michael Schlansker and Vinod Kathail. 1994. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–429.

[56] Yongming Shen, Tianchu Ji, Michael Ferdman, and Peter Milder. 2019. Argus: An end-to-end framework for accelerating CNNs on FPGAs. *IEEE Micro* 39, 5 (2019), 17–25.

[57] Steven Smith. 2003. *Digital signal processing: a practical guide for engineers and scientists*. Newnes.

[58] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. 2014. Architectural specialization for inter-iteration loop dependence patterns. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 583–595.

[59] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*. pmlr, 1139–1147.

[60] Synopsys. 2024. Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[61] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2021. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1388–1393.

[62] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-elastic cgras for irregular loop specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 412–425.

[63] Artem Vasilyev. 2019. *Evaluating spatially programmable architecture for imaging and vision applications*. Stanford University.

[64] Daniel Vazquez, Jose Miranda, Alfonso Rodriguez, Andres Otero, Pascuale Davide Schiavone, and David Atienza. 2024. Strela: Streaming elastic cgra accelerator for embedded systems. *arXiv preprint arXiv:2404.12503* (2024).

[65] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. University of California, Berkeley.

[66] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. 2022. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro* 42, 5 (2022), 59–69.

[67] Mark Wijtvliet, Henk Corporaal, and Akash Kumar. 2021. CGRA-EAM—rapid energy and area estimation for coarse-grained reconfigurable architectures. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14, 4 (2021), 1–28.

[68] Tianyi Yu, Omar Ragheb, Stephen Wicklund, and Jason Anderson. 2024. MLIR-to-CGRA: A Versatile MLIR-Based Compiler Framework for CGRAs. In *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 184–192.