SplitMS: Split Modulo-Scheduling for Accelerating Loops Onto CGRAs

Christie Sajitha Sajan, Kevin J M Martin, Satyajit Das, Philippe Coussy

SplitMS: Split Modulo-Scheduling for Accelerating Loops onto CGRAs

Christie Sajitha Sajan*†, Kevin J. M. Martin*, Satyajit Das†, Philippe Coussy*
*Univ. Bretagne-Sud, CNRS UMR 6285, Lab-STICC, Lorient, France
†IIT Palakkad, India
christie.sajan@univ-ubs.fr, kevin.martin@univ-ubs.fr, satyajitdas@iitpkd.ac.in, philippe.coussy@univ-ubs.fr

***Abstract*—Coarse-Grained Reconfigurable Array (CGRA) architectures are popular for accelerating loop kernels due to a good balance between energy efficiency and flexibility. Modulo scheduling (MS) is the preferred solution for efficiently mapping loops onto CGRAs. Existing CGRA MS algorithms suffer from low resource utilization if the number of operation nodes in the Data Flow Graph (DFG) is less than the number of Processing Elements (PEs) in the CGRA. To improve instruction level parallelism (ILP), the common approaches unroll the loop before applying MS. However, finding valid MS solutions for larger DFGs becomes difficult for CGRAs with resource constraints. This paper proposes a novel Split Modulo-Scheduling (SplitMS) technique to improve the ILP by segmenting the target CGRA into clusters and mapping loop chunks. We also present a lightweight hardware approach to support the cluster execution. Experiments show that SplitMS for a 4×[2×2] CGRA cluster achieves an average speedup of 2.8× over MS for a 4×4 target CGRA with 8 Load-Store Units (LSUs). SplitMS increases an average of 2.9× the PE utilization and 3× the energy efficiency over the conventional MS approach.**

***Index Terms*—CGRA, Loop Splitting, Loop Unrolling, Modulo Scheduling**

I. Introduction

CGRAs serve as accelerators that offer high performance, flexibility, and energy efficiency [1], [2]. CGRAs comprise a set of interconnected processing elements (PEs). Fig. 1 shows the PEs interconnected following a mesh torus topology, integrated into a System on Chip (SoC) that includes also a host CPU, with data sharing facilitated through Tightly Coupled Data Memory (TCDM). Each PE consists of an

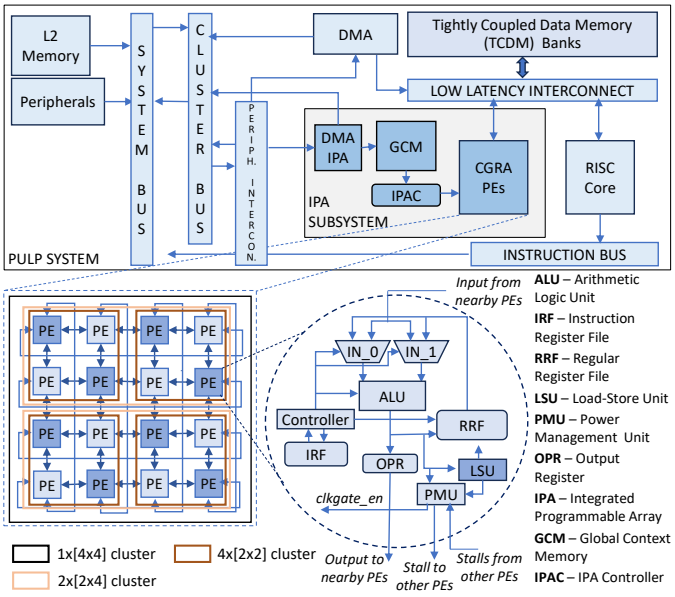

Fig. 1. IPA Cluster configuration in PULP SoC

Arithmetic Logic Unit (ALU), an Instruction Register File (IRF), a Regular Register File (RRF), and an optional Load Store Unit (LSU). The regular structure of the CGRA offers a very high Instruction Level Parallelism (ILP), particularly interesting in accelerating the loop kernels.

Over the last two decades, several loop optimization techniques have been used to improve Instruction Level Parallelism (ILP) for CGRAs. Loop unrolling is one of the common choices that involves expanding the loop fostering iteration overlaps while maintaining a scheduling barrier at the loop's end. Increasing the unrolling factor improves ILP with a cost of increased code size, and mapping complexity. Modulo Scheduling (MS) is a widely recognized technique to improve ILP over different iterations of the innermost loop of kernels for CGRAs. Modulo Scheduling (MS) creates a schedule for operations from multiple innermost loop iterations that repeat at fixed intervals while respecting dependencies within and between iterations. The interval between the initiation of successive iterations is called the Initiation Interval (II). Unlike unrolling, the increase in code size is minimal with modulo scheduling. A minimum initiation interval (MII) serves as a lower bound for the II for which a modulo scheduling exits for

the target architecture. The MII is determined by either one of the two key factors: available resources and loop recurrence, *MII* = *max*(*ResMII*, *RecMII*) [3], where ResMII represents the resource-constrained MII, determined by the ratio of operation nodes in the loop's Data Flow Graph (DFG) to the number of resources (usually in terms of number of Processing Elements (PEs)) in the CGRA, and RecMII, the recurrence-constrained MII, relies on the loop's existing recurrence dependencies. A recurrence in a loop occurs if an operation in one iteration directly or indirectly depends on the same operation from the previous iteration. For a loop with embarrassingly parallel workloads, the MII solely depends on ResMII as RecMII becomes unity.

The ResMII, defined as the ratio of operations to processing elements (PEs), denoted as Equation (1) reaches unity when the number of resources matches the number of operations in the Data Flow Graph (DFG).

$$ResMII = \lceil \#operations / \#PEs \rceil \tag{1}$$

This serves as a simplified illustration highlighting the necessity of split modulo scheduling. In scenarios where the operations are fewer than the available PEs in the CGRA, ResMII does not impact the Minimum Initiation Interval (MII), leading to suboptimal resource utilization and room for performance enhancement. To enhance resource utilization, one approach involves loop unrolling before applying modulo scheduling [4]–[7]. In the rest of the paper, this is termed as *"unroll then modulo schedule (UMS)"* approach.

Increasing the number of operations in the formula for ResMII ideally boosts resource utilization, resulting in an MII of 1 in Equation (1). However, loop unrolling complicates the mapping problem by exposing complex data dependencies across the iterations, potentially leading to workload imbalance in the PEs. In addition, the presence of resource constraints (e.g. heterogeneous PEs in CGRA, limited load/store units) further degrades the mapping performance. Moreover, the UMS approach suffers from unintended pipeline stalls due to the increased code size and mapping constraints. The PEs in a CGRA are directly connected allowing data to flow between them, forming a pipeline of concurrent computations. If one of the instructions stalls, the entire pipeline gets stalled due to synchronized execution. In this paper, we propose an alternative to optimize ResMII without loop unrolling that entails reducing the denominator of Equation (1) by decreasing the resource count. This concept underpins *split modulo scheduling (SplitMS)*, where the target architecture's resources are reduced through architectural segmentation or clustering, allowing modulo scheduling with fewer resources while replicating the schedule for similar architecture segments. The loop splits or chunks get thus executed in parallel. This achieves higher PE utilization like UMS without suffering from unintended pipeline stalls. If one or more splits get stalled, the other chunks can continue the execution.

Experiments demonstrate that for a 4×4 CGRA, SplitMS achieves 75% PE utilization, compared to 28% with state-of-the-art Conventional Modulo Scheduling (CMS). Additionally, SplitMS provides an average speedup of 2.8× over CMS, where UMS often fails to find valid solutions.

The contributions of the paper are:

1) a novel scheduling algorithm referred to as split modulo scheduling for better utilization of CGRA resources with improved performance; the approach iteratively explores theoretical speed-up by splitting the target CGRA into clusters and finds the best split;
2) a compilation flow and lightweight architecture support for the split modulo scheduling;
3) performance evaluation of the proposed method with comparison to conventional modulo scheduling (CMS) and unroll then modulo scheduling (UMS) techniques for a wide set of edge computing kernels.

The rest of the paper is organized as follows. Section II outlines the state-of-the-art works in the MS for CGRA targeted to improve ILP. The proposed approach is detailed in section III with a motivating example and execution model. The experiments are presented in section IV with a detailed discussion. Finally, the paper concludes in section V.

II. Related Work

Modulo scheduling is an early technique used to improve loop execution times on CGRA, by exposing more ILP. As the number of computing resources increases in CGRAs, unrolling (partially or completely) the loops further exposes more ILP. However, at the same time, it increases the mapping complexity. Some approaches have been proposed to deal with this complexity, like the work presented in [5], able to manage up to several hundreds of nodes, by clustering the large DFG obtained after unrolling, and then mapping smaller DFGs. In this paper, the technique proposed is the opposite: we first find the mapping for the DFG of the loop, and then we replicate this mapping.

Another group of related works concerns kernel partitioning techniques. The first subgroup performs kernel partitioning from the kernel point of view, where the issue is that the instruction memory is not large enough to accommodate the program of the full kernel [8], [9]. These techniques are orthogonal to the proposed approach in this paper. The second subgroup performs kernel partitioning from the data point view, where the issue is that the data does not fit the local (scratchpad) data memory [10]. The authors propose to first map the data in the memory, and then find the subtask mapping that uses the data. In this paper, we assume that the data fits in the local data memory similarly to [4]–[7], [11].

Finally, other loop transformation techniques were studied in [12]. The technique applies to perfectly nested loops, where loop interchange and loop skewing are the considered transformations in the frame of the polyhedral model. Surprisingly, the simple loop-splitting technique, as used in OpenMP, has not been studied for CGRAs. In this paper, we explore this transformation to make use of data level parallelism (DLP) present in embarrassingly parallel loops.

III. Split modulo-scheduling

A. Motivation

Modulo-scheduling is a widely used software pipelining technique that overlaps different iterations of a kernel loop to find a repeating schedule. The DFG formed by this repeating schedule is referred to as *Modulo Data Flow Graph (MDFG)*. The operations executed before and after the MDFG are called *prologue* and *epilogue* respectively. In signal processing kernels, the degree of parallelism achieved through MS is often low compared to the available Processing Elements (PEs) in target CGRA, resulting in low PE utilization. *PE Utilization* is the number of CGRA PEs used for executing the entire kernel. An established method to enhance parallelism is (partial or complete) loop unrolling before modulo-scheduling named hereafter Unroll then Modulo Schedule or UMS. However, this approach introduces an additional overhead to the compilation flow due to increased operations while mapping the MDFG to the CGRA. Higher unrolling factors further increase this challenge. Fig. 2(a) illustrates a sample loop modulo-scheduled on a 2×2 CGRA using conventional modulo-scheduling, where the upper-right PE is unused in both the cycles i and i+1, therefore the number of used PEs is 3 and unused PE is 1, reaching thus 75% PE utilization. The same loop is then unrolled by a factor of 2 and modulo-scheduled for the same CGRA, as shown in Fig. 2(b), resulting in a loop execution cycle improvement from 25 cycles in conventional modulo-scheduling to 20 cycles in UMS, and achieving 100% PE utilization. However, the number of operations in UMS increases from 4 to 9. We present an alternative method to effectively boost PE utilization by dividing loops into chunks and distributing these chunks onto smaller, similar segments/clusters of the CGRA. As depicted in Fig. 2(c), the loop example from the previous scenario is split into two chunks and modulo-scheduled across two segments of the CGRA. Our approach achieves 100% utilization, akin to UMS while reducing the execution cycle down to 14 cycles.

B. Proposed Approach

Split Modulo Scheduling enhances PE utilization by replicating modulo schedules across multiple clusters of PEs in a target CGRA. In this paper, we consider clusters as equivalent segments. As illustrated in Fig. 1, a 4×4 CGRA can be configured as 1×[4×4], 2×[2×4], or 4×[2×2] clusters. SplitMS greedily searches over the list of configurations for the best split. It initially maps the loop onto the largest cluster of the target CGRA and determines the II. In subsequent iterations, it maps the loop onto progressively smaller clusters, calculating the theoretical speedup compared to the initial configuration. This process identifies the optimal split for maximum performance. Each loop chunk is executed on a CGRA cluster. Given the similar configurations of all CGRA clusters, the mapping solution for one cluster can be replicated across all clusters. The detailed methodology and the algorithm are discussed below.

The proposed methodology is depicted in Fig. 3. The compiler takes the application code and the CGRA model as inputs. The CGRA model includes information about the hardware (number of PEs, Load/Store units), and the clusters, called splitting parameter. CGRA Clustering is performed where the CGRA is divided into different clusters using the splitting parameter. Initially, the whole CGRA is considered a single CGRA cluster and MS is performed to find a mapping solution. The CGRA is then split into equal clusters using the splitting parameter. The number of clusters is computed and a mapping solution is found for one PE cluster. Theoretical speedup is calculated between the clustered and unclustered CGRA to determine the optimal speedup. The split factor obtained to get the optimal speedup is used for splitting the loops into loop chunks and the mapping solution of one cluster is replicated to other PE clusters. The assembler converts the assembly code (mapping solution) into instructions for each PE and generates the context (binary code). The hardware simulator validates the execution on the target CGRA and various execution parameters are monitored, including memory stalls. In the simulator, the data is loaded into the data memory, and the context is loaded into the context memory of the CGRA. From the context memory, the instructions are loaded into the instruction memory of the CGRA and are executed.

The algorithm that determines the optimum split factor and generates the mapping for a single cluster is presented in Algorithm 1. It takes $p \times q$ CGRA model (the largest cluster) and the loop DFG as inputs. Additionally, a parameter s_{update} (user sets the value based on the configurations available) is provided to compute the splitting factor in each iteration. SplitMS first computes II_{cms} by applying any conventional modulo scheduling (CMS) technique on the $CGRA_{p\times q}$. Vari-

Algorithm 1 Split Modulo Scheduling (SplitMS) Algorithm

Require: CGRA dimensions $p \times q$, DFG (Data Flow Graph) represented as $D = (N, E)$ where N is the set of operation nodes and E is the set of edges connecting the nodes, and *s_update*
Ensure: Split modulo scheduling solution

$II_{cms} =$ CMS$(CGRA_{p\times q}, D)$
$T_{speedup} = T_{speedup\_max} = 1$
$Csize =$ max_Csize $= p \times q$
$s =$ s_best $= 1$
while $Csize > 1$ **do**
  if !initial_iteration **then**
    Csize = Csize/s_update
    s = max_Csize/Csize
  end if
  $CGRA_{m\times n} =$ clusterCGRA$(CGRA_{p\times q}, s)$
  if $CGRA_{m\times n} =$ NULL **then**
    continue
  end if
  $II_{SplitMS} =$ CMS$(CGRA^{1}_{m\times n}, D)$
  if $II_{SplitMS} = -1$ **then**
    continue
  end if
  $T_{speedup} = \frac{(II_{cms}+loop\_overhead_{cms})\times s}{(II_{SplitMS}+loop\_overhead_{SplitMS})}$
  if $T_{speedup} \geq T_{speedup\_max}$ **then**
    $T_{speedup\_max} = T_{speedup}$
    s_best $= s$
  end if
end while
if $T_{speedup\_max} = 1$ **then**
  $II_{SplitMS} = II_{cms}$
  $CGRA^{1}_{m\times n} = CGRA_{p\times q}$
end if
$M =$ genMapping$(CGRA^{1}_{m\times n}, D, II_{SplitMS})$
return M, s_best

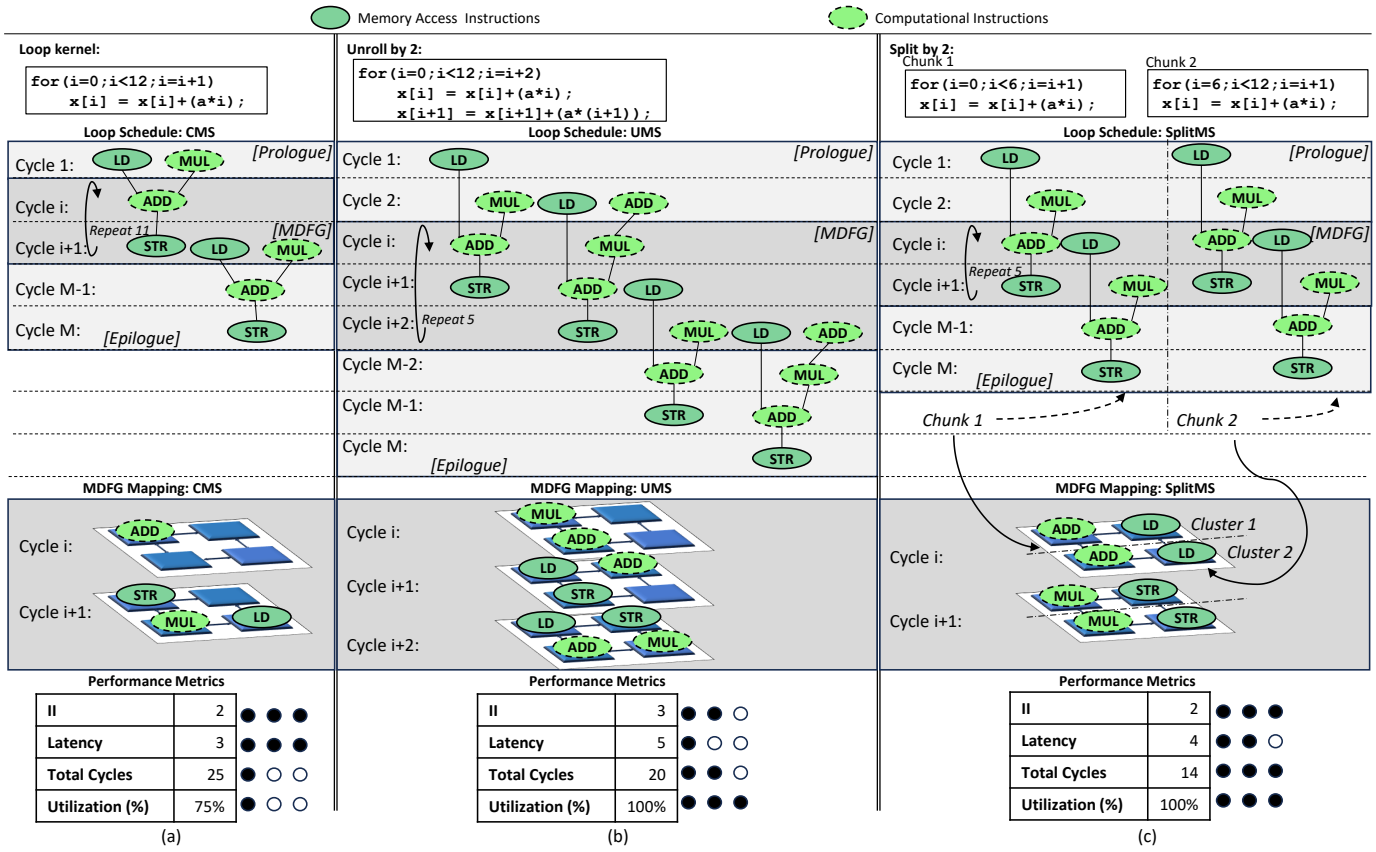

Fig. 2. Comparison between (a) Conventional Modulo-Scheduling (CMS), (b) Unroll then Modulo Schedule (UMS) and (c) Split and Modulo-Schedule (SplitMS)

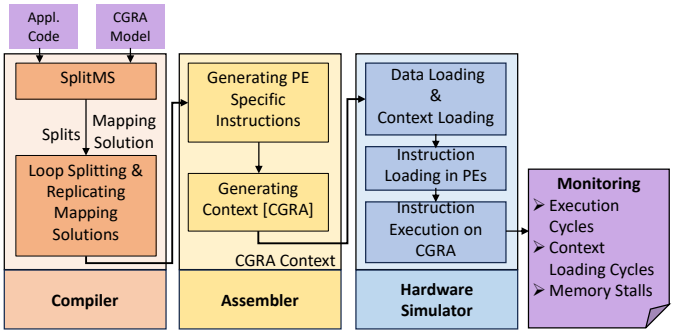

Fig. 3. Methodology of Cluster-Based CGRA with SplitMS

ables $T_{speedup}$ and $T_{speedup\_max}$ are initialized to 1. The letter T here represents theoretical values. The splitting factor s and the best splitting factor s_{best} are initialized to 1. The cluster size $Csize$ and the maximum cluster size max_Csize are initialized to $p \times q$. The algorithm iterates while the cluster size $Csize$ is greater than 1. s is updated based on whether it is the initial iteration or not. The CGRA is clustered into s similar segments of dimension $m \times n$, denoted by $CGRA_{m \times n}$ (progressively smaller clusters returned by *clusterCGRA* function). The *clusterCGRA* function is statically initialized with all the different cluster configurations available for the target CGRA. Initiation interval $II_{SplitMS}$ is calculated by CMS on the first cluster $(CGRA^1_{m \times n})$ among the list of clusters returned by the *clusterCGRA* function. If CMS on the clustered CGRA is unsuccessful, the algorithm proceeds to the next iteration to find a valid solution for the next split. Speedup is computed by comparing initiation intervals between the original and clustered CGRA configurations. If the $T_{speedup}$ exceeds $T_{speedup\_max}$, $T_{speedup\_max}$ and s_{best} are updated.

If the maximum speedup ($T_{speedup\_max}$) remains 1, indicating no improvement, $II_{SplitMS}$ and $CGRA^1_{m \times n}$ are set to the values obtained from the CMS on the original CGRA. The mapping M is generated using $CGRA^1_{m \times n}$, DFG, and $II_{SplitMS}$. The algorithm returns the mapping M and the best splitting factor s_{best}. This iterative approach optimizes CGRA utilization and performance in mapping operations onto the architecture. The algorithm systematically explores configurations to achieve the best splitting factor and mapping for the best split.

Fig. 4 illustrates the proposed SplitMS technique with an example. Consider a loop kernel of eight iterations with two computation operations (MUL, ADD) and two memory access operations (LD, STR). The loop body has an array (x) that accesses eight memory locations in memory. The memory has four memory banks where the data elements of the array are arranged in row-major order across the memory banks.

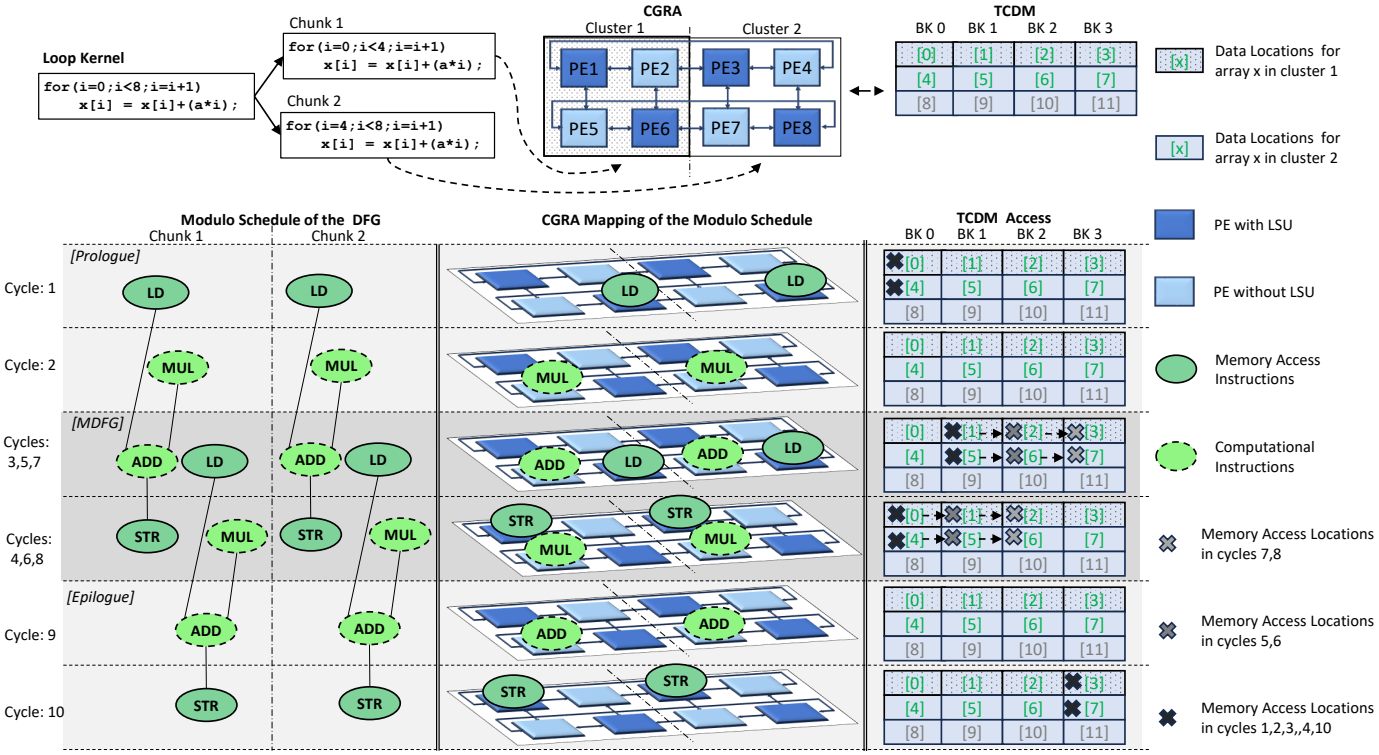

Fig. 4. Illustration of SplitMS

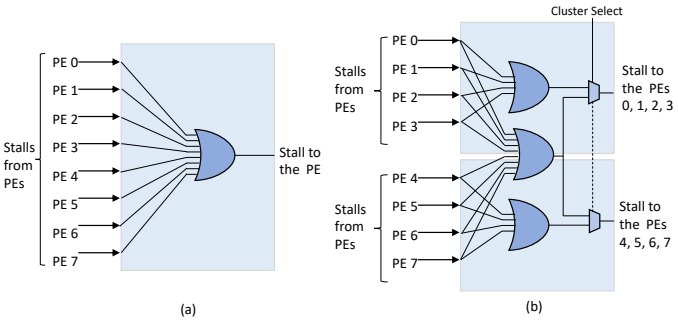

Fig. 5. (a) GFM for 2×4 CGRA (b) CFM for 2×[2×2] CGRA

Consider a CGRA of size 2×4 with four LSUs distributed on the CGRA alternately as shown in the Fig. 4. Let the split factor be 2, therefore the CGRA is divided into two clusters of size [2×2]. The loop is split into two chunks (chunk 1 & chunk 2) with four iterations each. This idea of modulo scheduling with minimum DFG nodes and minimum CGRA PEs simplifies the mapping process while improving performance and PE utilization.

C. Cluster-based CGRA

The PEs of the target CGRA presented in Fig. 1 are organized into clusters to facilitate parallel execution of loop chunks. The CGRA ensures the synchronized execution of the instructions. When one PE is stalled due to events such as memory access contention, all PEs halt execution. PEs resume execution once the stall is resolved. As illustrated in Fig. 1, the Power Management Unit (PMU) employs a Global Freeze Mechanism (GFM) to manage stalls, clock-gating all PEs during these events. However, in a clustered execution model, stalls in one cluster should not impact the execution of other clusters. Hence, we introduce the Cluster-Level Freeze Mechanism (CFM), allowing clusters to operate independently. Fig. 5 depicts the system-level diagram of both GFM and CFM. A PE typically generates a stall due to memory bank conflicts, which adds extra cycles for instruction execution. When a PE generates a stall signal, it informs all other PEs to prevent them from starting the next scheduled instruction. The GFM aggregates stall signals from all PEs using an OR operation to determine whether to proceed with execution. If any PE generates a stall, all other PEs suspend execution.

In a clustered system, a stall from one PE should only affect the PEs within its cluster rather than the entire CGRA. Thus, the GFM must be decoupled between clusters. The OR operation is performed only on stall signals from PEs within the same cluster. Consequently, each cluster's PEs make execution decisions based solely on their cluster's stall signals, not those of the entire CGRA.

Consider a 2×4 CGRA, configurable as 1×[2×4] or 2×[2×2]. In Fig. 5(a), the GFM is illustrated, where the OR operation integrates the stalls from all the PEs, with the output distributed back to all PEs for execution halt. Conversely, Fig. 5(b) depicts the CFM, where the OR operation is limited to stalls within each cluster, affecting only the PEs within that cluster. The *cluster select* determines the number of clusters in the CGRA, decoupling the GFM. In a 2×4 CGRA, the

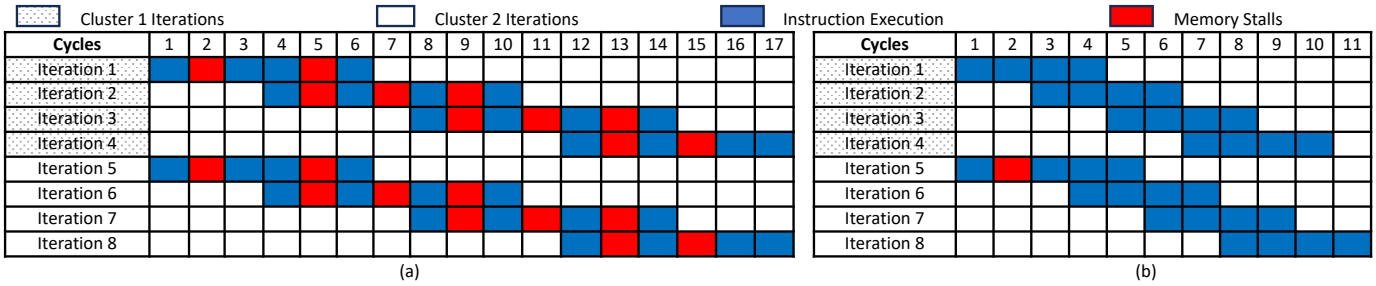

Fig. 6. Illustration of loop execution by SplitMS for the example in Fig. 4 with (a) GFM (a) CFM

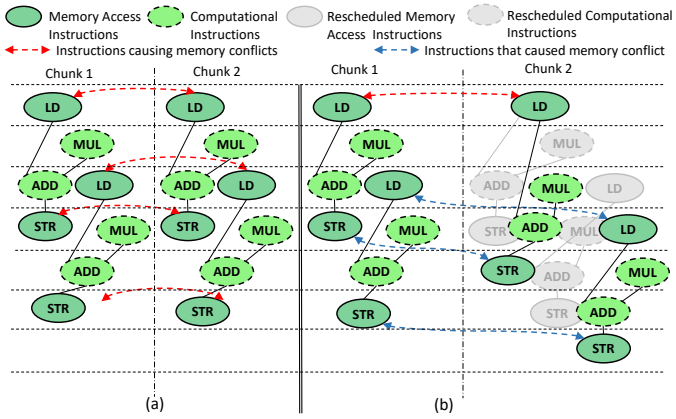

Fig. 7. Schedule for the example in Fig. 4 that (a) represents the memory conflicts (b) represents automatic rescheduling after a memory conflict

maximum number of clusters is 2. Thus, the *cluster select* requires 1 configuration bit for the freezing mechanism.

D. Cluster-based Execution

Several parallel memory accesses to the same memory bank cause delay/stall in the execution of instructions. This reduces the overall performance. Thanks to CFM, the clusters can reschedule among themselves avoiding future memory bank conflict when memory conflict occurs among clusters. This also eliminates the need for complex static data mapping techniques between clusters to avoid memory bank conflicts across clusters. For example, consider the same example from Fig. 4 which executes eight iterations, and the data is stored across four memory banks. We can see that the concurrent accesses across clusters compete for the same memory bank leading to conflicts. Parallel memory access techniques to reduce the conflicts are already explored in the literature [13]–[15]. These static techniques are complementary to our solution. The example from Fig. 4 is one of the worst cases considering the memory access pattern for this kernel. This worst case is handled at run-time by the CFM. Fig. 7(a) shows the MS for the two clusters without the CFM where the red double-headed dotted arrows indicate the memory bank conflicting instructions between the clusters. By decoupling the global-freeze system, the MS becomes rescheduled as shown in Fig. 7(b) which avoids the memory stalls after the initial memory banking conflict. The red double-headed arrows show the presence of memory banking conflict between the corresponding memory access instructions. The blue arrows indicate the memory access that had created conflicts that are rescheduled by moving the instructions to the successive cycles thereby avoiding the memory banking conflict in future iterations. Fig. 6 shows the execution of a loop with 8 iterations by having GFM and CFM in clustered execution of CGRA.

IV. Experiments

A. Experimental setup

The Integrated Programmable Array (IPA) [11] CGRA is the target CGRA architecture used to perform experiments for the proposed approach. The IPA CGRA is integrated into the PULP [16] cluster as shown in Fig. 1. The IPA compiler written in Java provides the mapping solution in IPA assembly code and the IPA assembler written in C++ generates the context code for simulation. Energy results are gathered based on the switching activity in the placed-and-routed netlist design. The SystemVerilog description of the CGRA is synthesized using the Cadence Genus with 90 nm CMOS technology libraries. Placement and routing is executed using Cadence Innovus, and power analysis is conducted using Cadence Voltus at a supply voltage of 0.9 V under typical process conditions.

We chose several compute-intensive loop kernels from the polybench benchmark suite. The loops possess no inter-iteration dependency (if non-nested loop) or with no inter-iteration dependency on the outermost loop (if nested loop). For the experiments, the loops were split into 2 and 4 chunks. CRIMSON [4] MS is used to map the loops onto CGRA. We considered a 4×4 CGRA with 8 LSUs connected to the shared TCDM with 16 memory banks for the baseline. Fig. 1 also shows the clustering of the baseline CGRA to run the proposed SplitMS. The PEs are clustered as 1×[4×4] PEs, 2×[2×4] PEs, and 4×[2×2] PEs. The placement of the LSUs is shown in dark blue color.

Experiments were performed for the following scenarios:

- Conventional Modulo-Scheduling (CMS): CRIMSON MS is applied on a loop kernel for a 1×[4×4] CGRA;
- Unroll by a factor of 2 then MS (UMS2): CRIMSON is applied on a loop unroll by 2 on a 1×[4×4] CGRA;
- Unroll by a factor of 4 then MS (UMS4): CRIMSON is applied on a loop unroll by 4 on a 1×[4×4] CGRA;

- Split by 2 MS (SplitMS2): CRIMSON is applied on a loop kernel for a 2×[2×4] CGRA;
- Split by 4 MS (SplitMS4): CRIMSON is applied on a loop kernel for a 4×[2×2] CGRA.

B. Results

This section evaluates the proposed approach using the following metrics: execution time, PE utilization, energy consumption, area overhead, and compilation time.

1) Execution Time: Table I displays the mapping II and the execution cycles achieved by CMS, UMS, and SplitMS. The UMS and SplitMS results are categorized into UMS2, UMS4, and SplitMS2, SplitMS4, corresponding to unroll and split factors of 2 and 4, respectively. Additionally, the theoretical speedup ($T_{speedup}$) discussed in Algorithm 1 is presented for various split factors. It is evident that during kernel execution, the actual speedup closely approaches $T_{speedup}$. In SplitMS4, the actual speedup is more for kernels like gemver, gemsummv, fir, backprop, thanks to the automatic rescheduling in clustered execution using CFM. In these cases, memory stalls are less as presented in Fig. 8. UMS2 and UMS4 often failed to find solutions due to architectural constraints, shown by the blank spaces in Table I.

Fig. 8 illustrates the execution latency breakdown, including context-loading, kernel execution, and memory stalls. In UMS, loop control instructions occupy less space in the context memory than in SplitMS, as every chunk in SplitMS contains loop control instructions. Thus, UMS has fewer context-loading cycles compared to SplitMS. However, SplitMS experiences fewer memory stalls due to the scheduling of a DFG mapped to a small-sized cluster and automatic rescheduling phenomena. SplitMS4 achieves the best performance, with an average speedup of 2.8× compared to CMS.

2) PE Utilization: Fig. 9 depicts the PE utilization for various cases, with the highest utilization of 75% achieved by SplitMS4. PE utilization is calculated as the ratio of the number of PEs used for mapping to the total number of PEs available in the CGRA. In CMS, many PEs remain unused, representing an opportunity for performance improvement by leveraging these idle PEs. High utilization is attainable through small cluster sizes (as in SplitMS) and by increasing the number of operations (as in UMS). Both loop splitting and loop unrolling enhance utilization and performance. Specifically, SplitMS4 shows 2.9× and SplitMS2 shows 1.9× better PE utilization compared to CMS.

3) Energy: Table II shows the energy consumption for different methods. Energy is the time taken to execute the loop on CGRA times the power. The average energy gain for SplitMS4 is 3× compared to CMS. SplitMS2 provides energy efficiency similar to UMS2 with an average gain of 1.8× compared to CMS (for 50% kernels with the mapping solution).

4) Area: Table III shows the area occupied by the different modules of the PE for the baseline IPA and IPA with clusters. With only 0.12% area overhead, the proposed SplitMS achieves almost 3× better utilization.

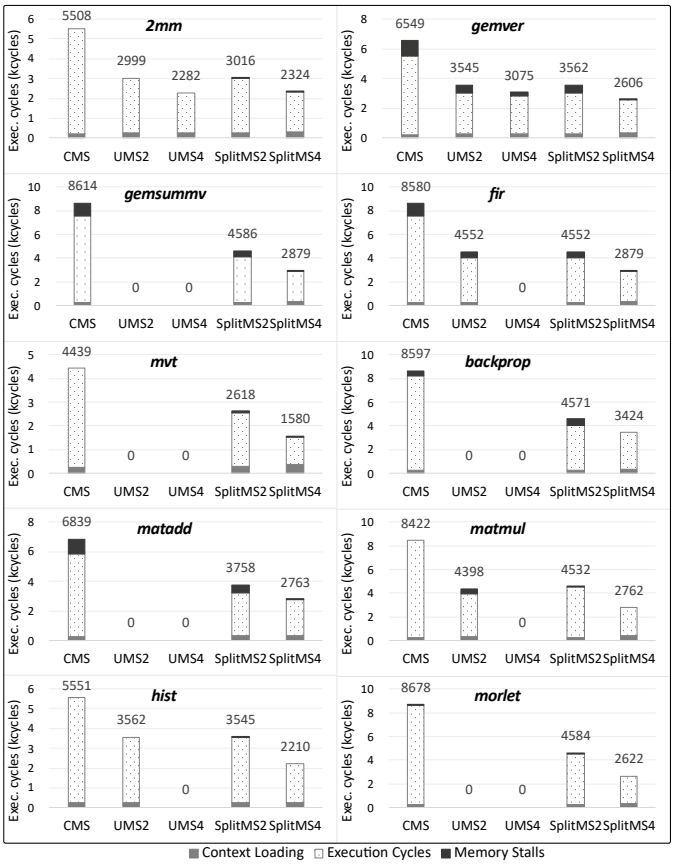

Fig. 8. Execution time breakup with configuration, execution, and memory stalls for the methods

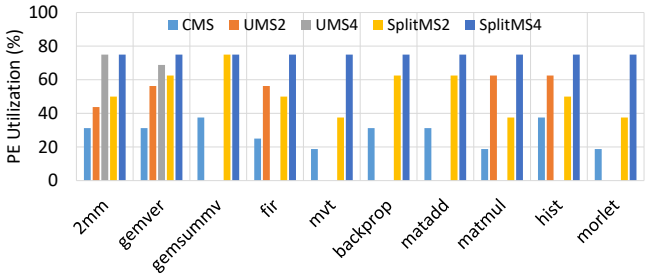

Fig. 9. PE Utilization Comparison

5) Compilation Time: Fig. 10 illustrates the compilation time (mapping process) in seconds for different methods. Increasing the number of operations and resources for modulo scheduling raises the mapping complexity. Unrolling increases the number of operations, while clustering reduces the resources needed for modulo scheduling. The figure clearly shows that SplitMS has lower mapping complexity compared to UMS. Additionally, mapping solutions could not be found for 50% of the kernels in UMS2, increasing to 80% in UMS4. In contrast, SplitMS successfully found mapping solutions for all kernels.

TABLE I
EXECUTION TIME COMPARISON BETWEEN DIFFERENT METHODS (CYCLES)

Kernel	CMS		UMS				SplitMS				Speedup over CMS					
			UMS2		UMS4		SplitMS2		SplitMS4		UMS		SplitMS2		SplitMS4	
	II	Exec	II	Exec	II	Exec	II	Exec	II	Exec	UMS2	UMS4	Theo.	Actual	Theo.	Actual
2mm	1	5508	1	2999	2	2282	1	3016	2	2324	1.84×	2.41×	2×	1.83×	3×	2.37×
gemver	1	6549	1	3545	3	3075	1	3562	3	2606	1.85×	2.13×	2×	1.84×	2.4×	2.51×
gemsummv	2	8614	-	-	-	-	2	4586	4	2879	-	-	2×	1.88×	2.67×	2.99×
fir	2	8580	2	4552	-	-	2	4552	4	2879	1.88×	-	2×	1.88×	2.67×	2.98×
mvt	2	4439	-	-	-	-	2	2618	2	1580	-	-	2×	1.7×	4×	2.81×
backprop	2	8597	-	-	-	-	2	4571	5	3424	-	-	2×	1.88×	2.29×	2.51×
matadd	1	6839	-	-	-	-	1	3758	3	2763	-	-	2×	1.82×	2.4×	2.48×
matmul	2	8422	2	4398	-	-	2	4532	2	2762	1.91×	-	2×	1.86×	4×	3.05×
hist	1	5551	2	3562	-	-	2	3545	3	2210	1.56×	-	1.5×	1.57×	2.4×	2.51×
morlet	3	8678	-	-	-	-	3	4584	3	2622	-	-	2×	1.89×	4×	3.31×

TABLE II
ENERGY CONSUMPTION (NJ) COMPARISON BETWEEN DIFFERENT METHODS

Kernel	CMS	UMS		SplitMS		Gain over CMS			
		UMS2	UMS4	SplitMS2	SplitMS4	UMS2	UMS4	SplitMS2	SplitMS4
2mm	50.6	26.3	19.2	26.4	19.4	1.9×	2.6×	1.9×	2.6×
gemver	60.5	31.4	26.7	31.4	21.9	1.9×	2.3×	1.9×	2.8×
gemsummv	80.2	-	-	41.2	24.4	-	-	1.9×	3.3×
fir	80.2	41.1	-	41.1	24.4	2×	-	2×	3.3×
mvt	40.2	-	-	22.5	11.6	-	-	1.8×	3.5×
backprop	80.1	-	-	41.1	29.6	-	-	1.9×	2.7×
matadd	63.2	-	-	33	23	-	-	1.9×	2.7×
matmul	78.5	39.1	-	40.7	22.4	2×	-	1.9×	3.5×
hist	50.7	31.4	-	31.4	18.9	1.6×	-	1.6×	2.7×
morlet	81	-	-	41.5	22.1	-	-	2×	3.7×

TABLE III
AREA (μm^2) BREAKDOWN OF THE COMPONENTS OF A PE

PE Modules		Baseline IPA	Cluster-based IPA
LSU		55 469.42	55 474.72
IRF		47 394.05	47 305.49
ALU		14 044.28	14 003.41
RRF		8 463.66	8 465.17
Controller		5 130.27	5 241.53
PMU	GFM	29.52	–
	CFM	–	126.4
	Counter	18.92	18.92
Total PE Area		130 550.11	130 635.64

V. CONCLUSION

In this paper, we introduced SplitMS to enhance PE utilization, instruction-level parallelism, and a lightweight hardware approach to support SplitMS. While previous techniques relied on the UMS approach to achieve similar goals, we demonstrated that this method lacks scalability. With SplitMS, we achieved an average of 75% utilization for a resource-constrained CGRA, where UMS often fails to find valid solutions. Compared to CMS, our approach delivers an average of 2.8× better performance and a 3× better energy profile, with only a 0.12% area overhead. In this paper, we focused on kernel loops without inter-iteration dependencies. Loops with inter-iteration dependencies will be addressed in our future work.

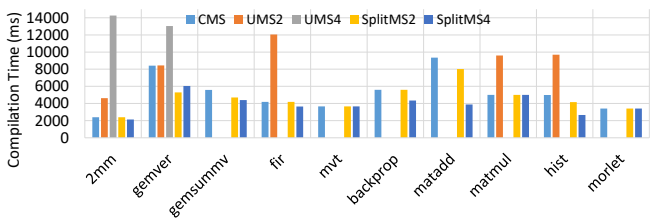

Fig. 10. Compilation Time (ms) comparison

REFERENCES

[1] A. Podobas, K. Sano, and S. Matsuoka, "A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective," *IEEE Access*, vol. 8, 2020.

[2] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[3] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *MICRO 27*, San Jose, California, United States, 1994.

[4] M. Balasubramanian and A. Shrivastava, "CRIMSON: Compute-Intensive Loop Acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping on CGRAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, Nov. 2020.

[5] D. Wijerathne, Z. Li, T. K. Bandara, and T. Mitra, "Panorama: divide-and-conquer approach for mapping complex loop kernels on cgra," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 127–132. [Online]. Available: https://doi.org/10.1145/3489517.3530429

[6] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–10.

[7] J. Pager, R. Jeyapaul, and A. Shrivastava, "A software scheme for multithreading on cgras," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 1, pp. 1–26, 2015.

[8] G. Ansaloni, K. Tanimura, L. Pozzi, and N. Dutt, "Integrated kernel partitioning and scheduling for coarse-grained reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 12, pp. 1803–1816, 2012.

[9] T. Kojima, A. Ohwada, and H. Amano, "Mapping-aware kernel partitioning method for cgras assisted by deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1213–1230, 2022.

[10] C. Li, J. Gu, S. Yin, L. Liu, and S. Wei, "Combining memory partitioning and subtask generation for parallel data access on cgras," in *2021 26th Asia and South Pacific Design Automation*

Conference (ASP-DAC), 2021, pp. 204–209. [Online]. Available: https://doi.org/10.1145/3394885.3431414

[11] S. Das, K. J. Martin, D. Rossi, P. Coussy, and L. Benini, "An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1095–1108, 2018.

[12] S. Yin, D. Liu, L. Liu, S. Wei, and Y. Guo, "Joint affine transformation and loop pipelining for mapping nested loop on CGRAs," in *DATE*, Mar. 2015, pp. 115–120.

[13] D. T. Harper III, "Increased memory performance during vector accesses through the use of linear address transformations," *IEEE transactions on computers*, vol. 41, no. 02, pp. 227–230, 1992.

[14] J. Takala and T. Järvinen, *Stride Permutation Access in Interleaved Memory Systems*. United States: Marcel Dekker Inc., 2003, pp. 63–84.

[15] C. J. Colbourn and K. Heinrich, "Conflict-free access to parallel memories," *Journal of Parallel and Distributed Computing*, vol. 14, no. 2, pp. 193–200, 1992.

[16] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.