FLEX : Introducing FLEXible Execution on CGRA with Spatio-Temporal Vector Dataflow

Thilini Kaushalya Bandara, Dan Wu, Rohan Juneja, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh
School of Computing, *National University of Singapore*
{thilini, danwu20, rohan, dmd, tulika, peh}@comp.nus.edu.sg

***Abstract*—Coarse-Grained Reconfigurable Arrays (CGRAs) are well-suited to resource-constrained edge devices due to their optimal combination of performance, energy efficiency, and adaptability. However, CGRAs typically follow a rigid execution model — either spatio-temporal or spatial — irrespective of the workload, limiting their efficiency. Spatio-temporal execution requires per-cycle reconfiguration, resulting in higher energy consumption. Conversely, spatial execution maintains the same configuration over a longer period; but this fixed mapping constraint can hinder the performance of complex applications and increase data memory accesses, leading to higher energy consumption. We introduce *FLEX*, a CGRA with a novel, flexible *spatio-temporal vector dataflow* execution model. This model processes a vector of data sequentially and chains them spatio-temporally. *FLEX* also supports variable vector lengths determined at compile time, enabling a more flexible execution paradigm. Our execution model reduces the reconfiguration frequency inherent in purely spatio-temporal mapping and mitigates the performance limitations and extra data memory accesses associated with purely spatial mapping. *FLEX* matches the performance of spatio-temporal CGRA but with 45% less energy and a 1.9× power efficiency improvement. Moreover, compared to a baseline spatial CGRA, *FLEX* consumes 35% less energy and delivers a 1.6× improvement in power efficiency at 1.5× higher throughput.**

***Index Terms*—Coarse Grained Reconfigurable Array (CGRA), Edge acceleration, Vector dataflow**

I. Introduction

Edge computing demands high performance within a modest power budget. ASICs are powerful but lack flexibility, driving interest in reconfigurable computing like FPGAs. Yet, FPGAs' bit-level reconfigurability limits power and area efficiency. Coarse-Grained Reconfigurable Arrays (CGRAs), with word-level reconfigurability, offer a balanced solution ideal for edge computing acceleration.

Serveral academic (ADRES [1], HyCUBE [2], SNAFU [3], Amber [4]) and commercial (Samsung SRP [5], Intel CSA [6], Renesas DRP [7], Sambanova RDU [8]) CGRA architectures have been proposed over the years. A CGRA consists of a set of Processing Elements (PEs) interconnected by a network along with on-chip memory. Each PE comprises a simple compute unit, a router for interconnect, and a reconfiguration unit with memory to hold the configurations/instructions generated by the CGRA compiler. CGRAs are ideal for accelerating compute-intensive loop kernels. The compiler takes the loop kernel represented as a Data Flow Graph (DFG), places and routes it onto the CGRA, leveraging inter- and intra-iteration parallelism [51].

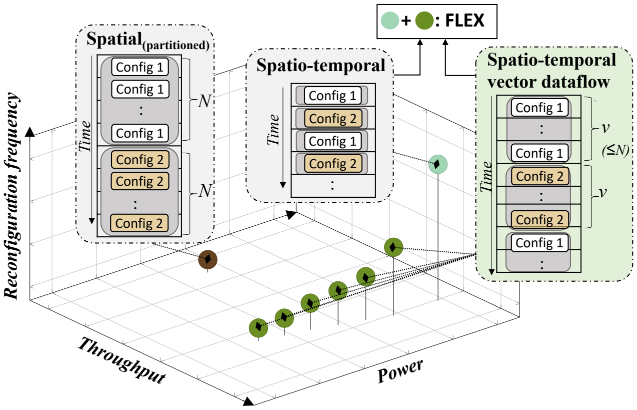


Fig. 1: *FLEX* execution model spans spatio-temporal, spatial[1]and spatio-temporal vector dataflow with variable vector lengths (v).

CGRAs' execution models can be characterized as spatio-temporal [1], [2], [9] or spatial [3], [10]. Spatio-temporal CGRAs deliver higher performance but are at the higher end of the power spectrum (Figure 1). Here a DFG is mapped in both spatial and temporal domains, requiring per-cycle reconfiguration of the PE-array during execution. In spatial execution, the DFG is only mapped spatially among the PEs; so the PE-array completes a full set of loop iterations (N) before switching the configuration, drastically reducing the reconfiguration energy. However, most DFGs do not fit on the PE-array and have to be partitioned into sub-DFGs. The sub-DFGs are executed one at a time, hindering application performance and leading to considerable data transfer overheads through on-chip memory due to dependencies among sub-DFGs (Spatial$_{(partitioned)}$ in Figure 1). Moreover, spatial CGRA proposals manually partition the DFGs [3], which is infeasible for complex applications.

To overcome these limitations, we introduce *FLEX*, a CGRA with a novel, flexible *spatio-temporal vector dataflow* execution model. This model processes a vector of data sequentially for each configuration and chains the vectors spatio-temporally. It supports variable vector lengths v ($v \leq N$), determined at compile time, facilitating a more flexible execution (variable v enables multiple design points as shown in Figure 1). *FLEX* covers a broad spectrum of reconfigurability, from spatio-temporal to spatial execution, achieving the best

[1] *FLEX* follows spatial execution if a kernel fits spatially on the PE-array without the need for partitioning.

of both worlds: high performance of spatio-temporal CGRAs at low power of spatial CGRAs. *FLEX* provides a design space exploration (DSE) framework to choose the best execution mode and vector length (v) for individual application kernels based on the user requirements. With this flexibility, *FLEX* can deliver higher performance/watt across diverse application kernels of varying complexity.

We design a flexible architecture, compiler, and DSE framework to accommodate the proposed spatio-temporal vector dataflow execution of *FLEX*. Our experimental evaluation using a range of applications demonstrates that *FLEX* achieves the same performance as spatio-temporal CGRA with 45% less energy and 1.9× better power efficiency. *FLEX* improves performance over spatial CGRA by 1.5× while saving 35% energy with 1.6× higher power efficiency. Moreover, *FLEX* can be coupled with SIMD execution (Single Instruction, Multiple Data) to enhance the power efficiency of SIMD CGRA by 2.0×.

Our concrete contributions include the following:

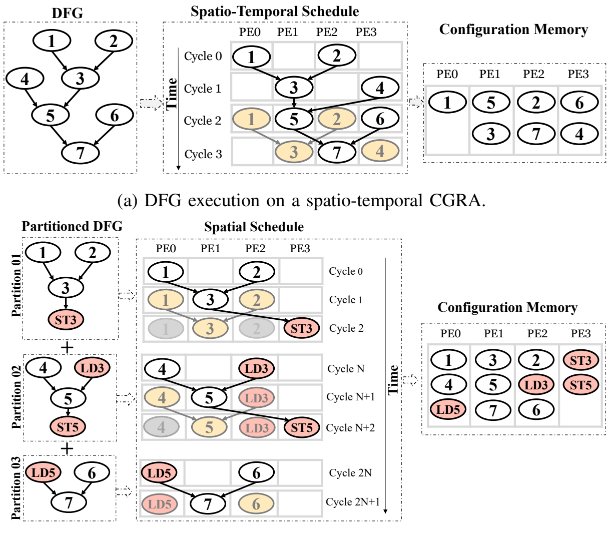
- We introduce a novel CGRA architecture *FLEX* with a flexible execution model spanning spatio-temporal, spatial, and variable vector length (v) spatio-temporal vector dataflow execution.
- We design the *FLEX* compiler to support spatio-temporal vector dataflow for near-optimal mapping.
- We offer design space exploration in *FLEX* to choose the optimal execution model for each application and user requirements, making it a truly general-purpose CGRA.

The framework is open-source and available at https://github.com/ecolab-nus/FLEX.

II. Background and Motivation

State-of-the-art (SOTA) CGRAs follow two prominent execution models: spatio-temporal and spatial. We analyze these two extremes to identify their inefficiencies and motivate the exploration of an alternative execution model that can morph between these extremes.

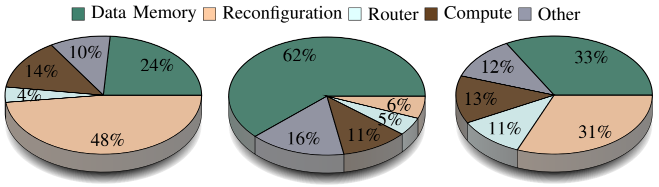
Spatio-temporal CGRA: High reconfiguration energy. Spatio-temporal CGRA execution requires higher energy due to per-cycle reconfiguration. Figure 2a shows the spatio-temporal mapping for the execution of a DFG corresponding to a loop kernel. Each PE executes multiple configurations (II=2, II is the number of cycles between two consecutive loop iterations) for each loop iteration, and the schedule is repeated cyclically for N loop iterations, resulting in $II \times N$ reconfigurations for each PE. Figure 3a shows the average power distribution of a spatio-temporal CGRA for the kernels in Table I (see Sections IV-A, IV-B for experimental setup). Ideally, most of the resources should be utilized in performing actual computation (compute) and dataflow (router, data memory). Instead, nearly half the power is consumed to reconfigure the CGRA. The frequent reconfiguration also limits the clock-gating opportunities as the components, especially the configuration memory, toggle fast between active and idle states. Hence, spatio-temporal CGRAs typically consume higher power than spatial CGRAs.



(a) DFG execution on a spatio-temporal CGRA.

(b) Partitoned DFG execution on a spatial CGRA.

Fig. 2: DFG schedule on 4x1 spatio-temporal and spatial CGRAs. Original DFG in (a) is partitioned into three sub-DFGs (each partition with II=1) in (b). For N iterations of the loop, (a) takes $2N$+2 cycles while (b) takes $3N$+5 cycles.



(a) Spatio-temporal CGRA (b) Spatial CGRA (c) *FLEX*

Fig. 3: Average power distribution in SOTA CGRAs vs *FLEX*

Spatial CGRA: Partitioning degrades performance and increases data memory energy. Spatial CGRAs work well for smaller kernels where the DFG fits spatially. However, the DFGs are too complex for most real-world application kernels to fit on a small spatial CGRA. In such cases, the original DFG is partitioned into sub-DFGs, each mapped spatially and in isolation. Figure 2b shows the execution of the example kernel on a similar-sized spatial CGRA. The DFG is partitioned into three sub-DFGs, and each is mapped spatially. Additional `LOAD/STORE` nodes are added to handle the data dependencies among the partitions. Each partition requires only one reconfiguration for all loop iterations, resulting in low reconfiguration energy (Figure 3b).

However, partitioning reduces parallelism opportunities and introduces additional nodes handling intermediate data, causing longer execution times. In Figure 2b, spatial CGRA takes $3N$+5 cycles while spatio-temporal CGRA takes only $2N$+2 cycles (i.e., ∼50% fewer cycles) for the loop with N iterations.

Moreover, partitioning reduces the data locality by moving the intermediate data (representing dependencies among sub-DFGs) in and out of the PE-array. This results in frequent data memory accesses, increasing power consumption. Figure 3b shows the data memory power dominates with 62% contribution, an artifact of DFG partitioning required by spatial CGRA.

Spatial CGRA: Difficulty with inter-iteration dependencies. Spatial CGRAs mandate the processing of inputs in consecutive cycles. Thus, a kernel with true inter-iteration dependency (the output of one iteration becomes an input to the next iteration) will cause the execution of the next iteration to halt till the output of the current iteration is generated. In the worst-case, the spatial mapping cannot exploit any inter-iteration parallelism leading to poor performance. These dependencies are even more challenging for partitioning as the parent and child nodes must be contained within the same partition.

Spatial CGRA: Complex kernel partitioning challenges. Kernel partitioning differs from traditional graph partitioning with additional objectives: (a) Each sub-DFG should map with II=1; (b) The DFG operations should be distributed to match the compute capabilities of the CGRA and adhere to the data memory layout; (c) Inter-sub-graph dependencies should be acyclic for sequential execution. Current spatial CGRA proposals [3] use manual partitioning, which is infeasible for complex DFGs.

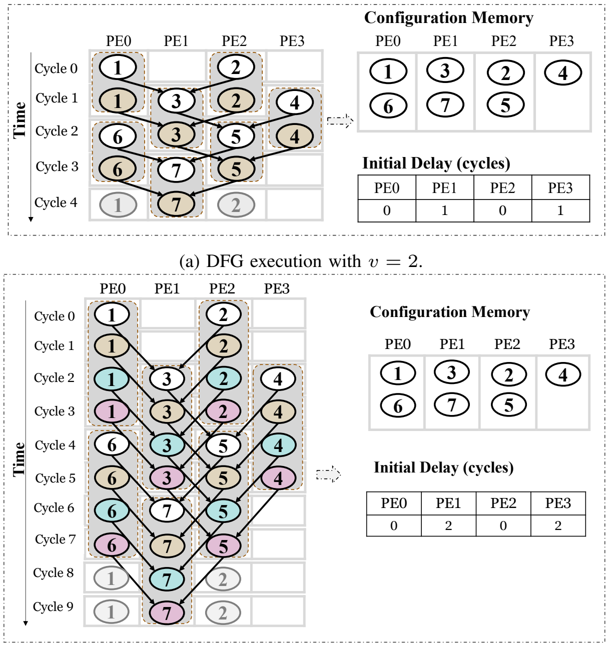
The challenges discussed for both the CGRA execution models are associated with their respective reconfiguration methods. Spatio-temporal and spatial CGRAs lie at the two extreme ends with the highest and lowest reconfiguration frequencies (Figure 1), respectively. Frequent reconfiguration increases the cost of reconfiguration energy while partitioning associated with spatial execution results in performance overheads. This motivated us to explore the intermediate scope with slightly higher reconfiguration frequency than spatial CGRA but lower frequency than spatio-temporal CGRA to minimize the adverse impacts of each approach while maintaining their respective advantages. We propose the concept of spatio-temporal vector dataflow execution (Figure 4), an innovative approach designed to create a more balanced execution model and architecture that covers the entire reconfiguration spectrum. Figure 3c shows that spatio-temporal vector dataflow of *FLEX* reduces the power contribution from the reconfiguration of spatio-temporal CGRA and data memory of spatial CGRA balancing the power distribution among reconfiguration, data memory, compute, and routing. We also provide an automated exploration framework of the reconfiguration frequency spectrum through the choice of the vector length (v). This allows the user to select the optimally tailored execution mode and vector length (v) for each application at compile time based on the specific performance and power needs.

III. Flex

FLEX CGRA architecture enables flexible execution of application kernels with high performance and low power. We elaborate on the complete system comprising the execution model, micro-architecture, compiler, and DSE framework.

A. Execution Model

FLEX supports a reconfigurable execution model spanning the spectrum from spatio-temporal to spatial execution. We introduce the spatio-temporal vector dataflow execution, a novel approach to cover the entire reconfiguration spectrum. On the same silicon substrate, each application kernel can choose the best execution mode at the compile time, depending on the performance and power requirements. Later, we show that spatio-temporal vector dataflow execution is beneficial for most kernels.



PE0	PE1	PE2	PE3
0	1	0	1

(a) DFG execution with v = 2.

PE0	PE1	PE2	PE3
0	2	0	2

(b) DFG execution with v = 4.

Fig. 4: Spatio-temporal vector dataflow on *FLEX*

Spatio-temporal vector dataflow execution: In spatio-temporal vector dataflow execution, each reconfiguration of the PE array processes an entire vector of v inputs sequentially, thus reducing the overheads of configuration. The outputs are then chained spatially and temporally so that the consumer PEs have access to each input as soon as it is produced, minimizing the requirement for intermediate data storage. In *FLEX*, the DSE framework selects the most suitable v for an application. Each compiler-generated configuration runs v times consecutively, with each PE executing the same instruction on successive elements of the vector, sending the vector of v results along identical routes to the same destination PEs.

Figure 4 shows spatio-temporal vector dataflow for the DFG in Figure 2a. Figure 4a shows v=2 scenario, where reconfiguration of the PEs happens every two cycles. PE0 processes two iterations of node 1 in cycle 0 and cycle 1. The produced result is routed immediately to PE1, which executes node 3 in cycle 1 and cycle 2. Similarly, all the intermediate data produced are consumed immediately, which results in a fully chained schedule without using any of the registers on the PE-array. This schedule has a maximum of two configurations per PE, similar to spatio-temporal execution in Figure 2a but with half the reconfiguration frequency. With spatio-temporal vector dataflow, all the PEs may not start simultaneously, as indicated by the initial delay table. *FLEX* with v=2 takes $2N$+1 cycles to execute N iterations of the loop, achieving the same performance as the spatio-temporal CGRA.

Figure 4b is the schedule on *FLEX* with v=4. Every node is executed on four inputs/iterations before reconfiguring. The intermediate outputs produced are stored in registers for one cycle before being consumed by the destination PEs. This schedule requires only two configuration entries per PE and takes $2N$+2 cycles to execute, achieving the same performance as a spatio-temporal CGRA at 75% less reconfiguration.

Spatio-temporal vector dataflow can achieve the same performance as spatio-temporal execution with significantly less reconfiguration. It also avoids the costly memory accesses and performance impact of spatial CGRAs. The vector length v is adaptable, allowing design space exploration to optimize the execution for individual loop kernels.

For V_c: maximum vector length supported by the application kernel and V_h: maximum vector length supported by the hardware, vector length $v \in \{1, 2, 3, .., V\}$ where V=$min\{V_c, V_h\}$. We set V_h=8 in the current architecture empirically based on application analysis. *FLEX* becomes a spatio-temporal CGRA when v=1, II>1 and a spatial CGRA when v=1, II=1.

FLEX can operate in three different execution modes:

- *Spatial execution*: For smaller kernels that can fit spatially onto the CGRA without partitioning.
- *Spatio-temporal vector dataflow execution*: For vectorizable kernels with optimal v determined via DSE.
- *Spatio-temporal execution*: For non-vectorizable kernels with true inter-iteration dependency between consecutive iterations.

B. *FLEX Micro-architecture*

Figure 5 illustrates *FLEX* micro-architecture with 4x4 PE-array connected via single-cycle multi-hop mesh network [2].

1) Processing Elements (PEs): Each PE combines a compute unit, a crossbar for network connectivity, and a configurator for reconfiguration handling. *FLEX* has memory PEs to handle all the data memory accesses and regular PEs to perform all the remaining computations.

2) Compute Unit: The compute unit in a regular PE can perform 16-bit arithmetic operations, logic operations, multiplication, and division. Compute unit is simplified in the memory PEs with arithmetic required for simple memory address generation. Any complex address calculation, like indirect addressing, is translated to multiple DFG nodes and computed using multiple PEs. Local storage within PE holds constants and locally used data.

3) Router/Crossbar: The 7x7 crossbar in each PE forms many-to-many connections among internal and external datapaths. Registers are sprinkled along the datapath to buffer the data to match the latencies. Clock-gated link registers at the outputs, and the internal datapath cater to the buffering required by spatio-temporal vector dataflow. More link registers improve the performance with larger vector lengths; thus, register availability is crucial in determining v for execution.

4) Configurator: The configurator handles reconfiguration for each PE. We split the configuration memory, which holds the configuration words, into two sub-memories, one for reconfiguring the compute unit along with crossbar configurations for its inputs and the other for the crossbar configurations of the external ports. Decoupled memories allow more flexibility for the compiler to perform vectorization with chaining, as they can be toggled individually. Individual PCs are triggered after the specific initial delay determined by the compiler and are incremented after v number of cycles plus idle cycles.

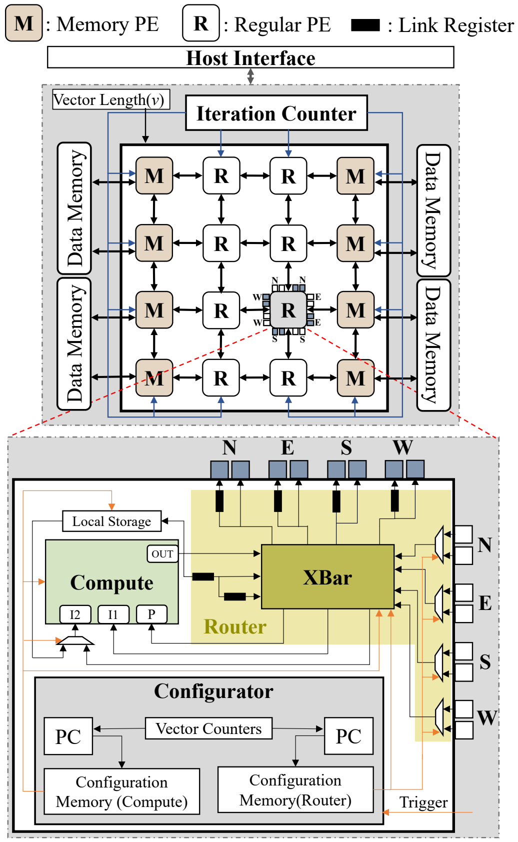


Fig. 5: *FLEX* microarchitecture

5) Top Level: All the memory PEs directly connect to the global data memories. Boundary PEs are connected to the iteration counter unit, which calculates the iteration variable. The global vector length register is loaded with each kernel's compiler-determined optimal v. Based on our application profiling, it is set to 3-bits (V_h=8). This global register is loaded with other configurations and remains the same throughout execution. All the PEs are directly connected to this register. As each PE starts with a different initial delay, each triggered PE needs a local vector counter, which is incremented until the count reaches v and resets.

6) Clock Gating: Clock gating cells are inserted into the configuration memories, program counters, and link registers. If the PE is idle for the entire execution or a particular cycle, PE-level clock gating cells will block the clocking. Micro-level clock gating is applied during vector execution and when the PE is partially active (either the compute unit or the router). During vector execution, the configurator is clock gated for v-1 cycles. Hence, a larger v leads to higher power savings.

Algorithm 1: Blockwise mapping algorithm

```
   Data: DFG,CGRA description, $v$
   Result: vecII,validMapping
1  sortedDFG=$getALAPSortedDFG(DFG)$;
2  timeExtendedDFG = $getTimeExtDFG(sortedDFG, v)$;
3  vecII = $v * getMinII(DFG, CGRA\ description)$;
4  while !validMapping do
5      MRRG = $getMRRG(CGRA\ description, vecII)$;
6      foreach vectorOfNodes in timeExtendedDFG do
7          candidates=$checkAvailability(partiallyMappedMRRG)$;
8          mapping=$place\&Route(candidates, vectorOfNodes[0])$;
9          foreach node in vectorOfNodes do
10             validMapping=$reserveNodeAndRoute(mapping)$;
11     if validMapping then
12         return $\{vecII, validMapping\}$;
13     else
14         vecII = vecII + 1;
```

C. FLEX Compiler

We design a compiler to generate the configuration binary starting from the application loop kernel in C. Initially, we perform vectorization checks with an LLVM auto-vectorization pass to determine V_c (maximum vector length) of each loop kernel. Next, we use LLVM-based Morphor [11] tool to generate the corresponding DFG from the kernel code. Generated DFG and V_c are fed to our customized CGRA mapper.

We design the CGRA mapper to handle the spatio-temporal vector dataflow execution paradigm, which is essentially a blockwise pathfinder mapper [12]. In contrast to SOTA CGRA mappers [11], [42], [45], which place and route individual nodes, blockwise mapper time extends the nodes by v to form blocks which are then placed and routed. Only the first element is placed and routed; all the remaining vector elements will follow the exact mapping. Thus the selected PE and the route are reserved for v cycles.

Mapping Problem Definition: For a given DFG $D = (N_D, E_D)$ and a CGRA, the problem is to derive the minimally time-extended Modulo Routing Resource Graph (MRRG) [13] $M_{vecII} = (N_M, E_M)$ for the time extended instance of D, $D[v]$ which has a valid mapping $\phi = (\phi_N, \phi_E)$ on M_{vecII}. MRRG depicts the routing and the resources for a time-extended CGRA. $D[v] = (N_D[v], E_D[v])$ where $N_D[v]$ corresponds to a vector of node N_D and $E_D[v]$ to vector of edge E_D, v is the vector length.

Algorithm 1 elaborates on the blockwise mapping approach. Original DFG is sorted with as-late-as-possible (ALAP) priority, and each node is time extended by v cycles (Lines 1, 2). The algorithm iterates over all the vectors of nodes, placing and routing the first element of the node (Lines 7, 8) and reserving the exact placement and route for the remaining vector elements (Line 10). Initial $vecII$ (Initiation Interval for vector dataflow) is set to the minimum possible value (Line 3) and incremented by 1 until a valid mapping is reached.

When v=1, blockwise mapper provides spatial or spatio-temporal mappings. If a valid mapping exists for $vecII$=1, *FLEX* will follow spatial execution. Any mapping with $vecII$>1 and v=1 is a spatio-temporal mapping.

D. Design Space Exploration (DSE) with FLEX

FLEX performs DSE to determine the best execution mode for each kernel. Figure 6 summarizes the DSE framework, and algorithm 2 explains the pseudocode. Our DSE framework receives kernel DFG, N: the number of loop iterations, V_c, V_h, *FLEX* architecture description in JSON, and netlist generated from synthesized *FLEX* RTL as inputs. It performs DSE and derives the optimal execution mode. First, the DFG is mapped with v=1 (Line 3). As V=1 denotes inter-iteration dependency that prevents vectorization; the framework selects spatio-temporal execution (Line 4). If initial mapping provides $vecII$=1, spatial execution is selected; else mapping instances are generated for different v. For each v, the blockwise mapper is invoked to obtain the $vecII$ (Line 16) and the number of cycles to complete kernel execution (Line 19). Power estimation is done using the configuration binary corresponding to each mapping ($binary_v$) (Line 20). The design instance that achieves the lowest power with accepted performance is selected as the optimal execution mode.

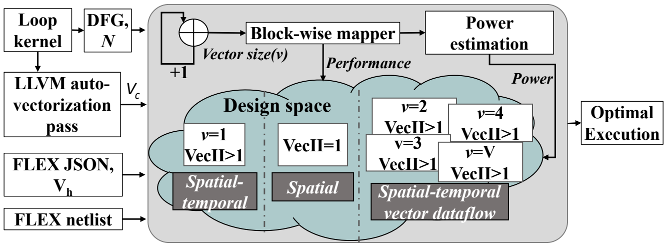


Fig. 6: *FLEX* DSE framework

Algorithm 2: DSE algorithm

```
   Data: DFG, $N$, $V_c$, $V_h$, FLEX JSON, FLEX netlist
   Result: optimum execution mode
1  $V$=$min\{V_h, V_c\}$;
2  $v$=1;
3  $\{vecII_1, validMapping\}$ =
     $blockwiseMapper(DFG, CGRA\ JSON, v)$;
4  if $V$ == 1 then
5      return $SpatiotemporalExecution$
6  else
7      if $vecII_1$ == 1 then
8          return $SpatialExecution$
9      else
10         $binary_1$= $dumpConfigBinary(validMapping, v)$;
11         $performance_1$ = $computePerf(vecII_1, N, v)$;
12         $power_1$= $getPower(binary_1, FLEX\ netlist, v)$;
13         $addToDesignSpace(\{performance_1, power_1\})$;
14         while $v < V$ do
15             $v = v + 1$;
16             $\{vecII_v, validMapping_v\}$ =
                 $blockwiseMapper(DFG, CGRA\ JSON, v)$;
17             if $validMapping_v$ then
18                 $binary_v$=
                     $dumpConfigBinary(validMapping_v, v)$;
19                 $perf_v$ = $computePerf(vecII_v, N, v)$;
20                 $power_v$=
                     $getPower(binary_v, FLEX\ netlist, v)$;
21                 $addToDesignSpace(\{perf_v, power_v\})$;
22         return $selectOptimumExecution()$;
```

IV. Experimental Evaluation

A. Baseline Architectures

We evaluate *FLEX* against spatio-temporal [1], [2] and spatial CGRAs [3], [10]. Our prototypical baseline spatio-temporal and spatial CGRAs are modeled after HyCUBE [2] and SNAFU [3], respectively, with the following parameters for fair comparison:

PE-array: 16 PEs arranged in a 4x4 single cycle multi-hop [2] mesh network with 8 memory PEs on the boundary columns and 8 regular PEs.

On-chip data memory: Spatio-temporal CGRA and *FLEX* have 512B×4 dual-port memories connected to boundary columns. Spatial CGRA has 1KB×4 dual-port memories, slightly larger to accommodate intermediate values due to partitioning if necessary.

Configuration memory: Spatio-temporal CGRA and spatial CGRA have a single configuration memory per PE with 8 and 16 entries, respectively, to fit the selected kernels. *FLEX* has separate routing and compute configurations with 8 entries each (Section III-B4).

Reconfiguration: Spatio-temporal CGRA reconfigures the PEs every cycle. Spatial CGRA reconfigures once a sub-DFG execution completes for all the data elements N. *FLEX* in vector dataflow mode reconfigures every v cycles.

Power management: Clock gating is implemented in all three architectures. Idle PEs are clock gated throughout the execution, and partially active PEs are clock gated during idle cycles. Clock gating for *FLEX* is presented in Section III-B6.

B. Experimental Setup

All three architectures are implemented in System Verilog HDL, synthesized on a commercial 22 nm FDSOI process using Cadence Genus Synthesis Solution at 100 MHz clock. We generate the binary code for the CGRA configurations as described in Section III-C along with the data layout for simulation. We use the same compiler set to v=1 for generating configuration binaries for the baselines. Generated binaries are used to perform RTL simulation using the Cadence Xcelium tool, followed by power analysis with the Cadence Joules tool.

An additional kernel partitioning step is done for spatial CGRA if the kernel is too big to fit with II=1. We partition the DFG into sub-DFGs, ensuring minimal dependencies among the partitions. For the dependencies, intermediate values are stored in the on-chip data memory. `LOAD/STORE` operations are added to handle the flow of intermediate values between the sub-DFGs. The configuration binary is generated for each sub-DFG and combined sequentially for complete execution. For smaller kernels, we use manual partitioning, ensuring a near-optimal solution. For partitioning complex DFGs (e.g., *jpegdct*), we use a modified version of Clauset-Newman-Moore greedy modularity maximization [14].

C. Benchmark Applications

Table I summarizes the benchmark kernels used, the DFG sizes, and any additional memory required for spatial execution to hold the intermediate values. The kernels are selected from popular edge benchmarking suites MachSuite [15], Polybench [16], Wavelib [17], and BEEBS [18], covering multiple domains and constitute commonly used edge applications (e.g., object detection, speech recognition algorithms use *fft*, *conv2d* and *GeMM* kernels). All the kernels except *GeMM* require partitioning to execute on the spatial CGRA. In *aes*, *fir*, and *jpegdct* kernels, more than 50% of the memory operations in the partitioned DFGs are for handling the inter-partition dependencies. Kernel *nw* with inter-iteration dependency does not meet the partitioning criteria and thus cannot execute on the spatial CGRA.

TABLE I: Application kernels used in the evaluation

App. Kernel	Description	No. of nodes		Additional memory
		Original	Partitioned	
fft	256-pt stockham FFT	19	22	20.0%
conv2d	2D convolution , 3x3 filter (64x64 matrix)	29	39	44.4%
aes	Rajndeal with 256 block size and 32-bit key size	37	49	52.2%
stencil3d	7pt stencil calculation on 3D data (16x32x32)	18	20	20.0%
idwt	Sym inverse discrete wavelet transform with db8 wavelet	23	32	42.9%
fir	FIR filter with redundant load elimination	17	25	54.5%
jpegdct	jpeg encoding (8x8 discrete cosine transform)	71	126	77.5%
GeMM	General matrix multiplication, unrolled 4 (64x64 matrix)	16	-	0.0%
nw	Dynamic programming algorithm for optimal sequence alignment (len=128)	30	✗	-

TABLE II: Selected execution modes on *FLEX*. STV: Spatio-temporal vector, ST: Spatio-temporal, S: Spatial

Kernel	**fft**	**Conv2d**	**aes**	**stencil3d**	**idwt**	**fir**	**jpegdct**	**GeMM**	**nw**
Execution Mode	STV	STV	STV	STV	STV	STV	STV	S	ST
Vector length (v)	4	4	4	8	4	4	2	-	-

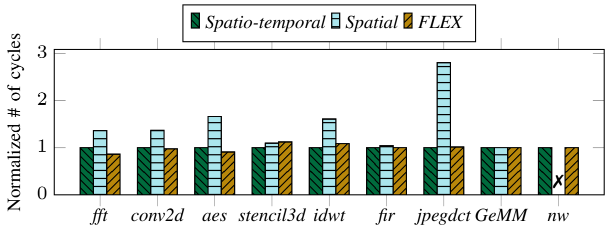


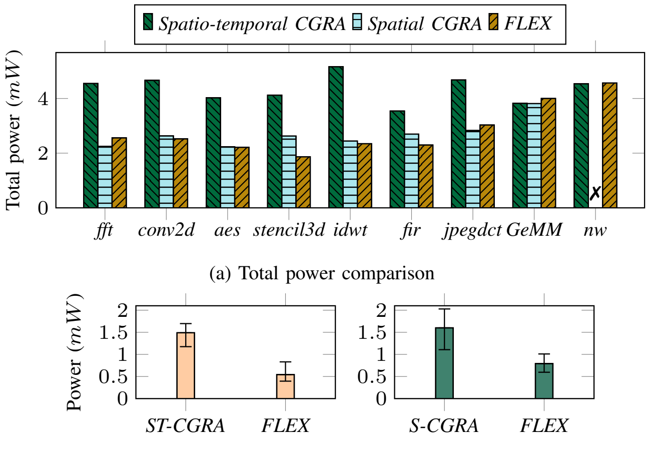
Fig. 7: Kernel runtime normalized w.r.t. spatio-temporal CGRA runtime.

D. Experimental Results

We perform DSE for each kernel and choose the optimum execution mode based on the lowest power under performance constraint (≤10% degradation of optimal performance under spatio-temporal). Table II summarises the derived execution modes. All the kernels that can be vectorized follow spatio-temporal vector dataflow with different v. *GeMM* kernel is small enough to fit spatially on *FLEX*. Kernel *nw* requires spatio-temporal execution due to the inter-iteration dependency that does not allow vectorization.

1) ***Performance*:** We present the performance of each kernel in terms of the number of cycles it takes for a complete execution on each of the architectures. Figure 7 shows the normalized performance of each kernel w.r.t. the performance on spatio-temporal CGRA. *FLEX* vectored execution achieves nearly the same performance as spatio-temporal CGRA for all the kernels. In *fft* and *aes*, *FLEX* achieves slightly better performance as vector chaining improves latency. Spatial CGRA takes longer execution time in all kernels, mainly due to the impact of partitioning; on average, it requires 49% more cycles compared to the spatio-temporal CGRA and *FLEX*. Due to its complexity, kernel *jpegdct* suffers the maximum performance degradation (2.8×) on spatial CGRA.

2) ***Power*:** We compare the average power consumption throughout the execution of each of the kernels. Figure 8a depicts the total power consumption. *FLEX* executing in spatio-



(a) Total power comparison

(b) Configuration memory

(c) Data memory

Fig. 8: Power comparison. ST-CGRA: Spatio-temporal CGRA, S-CGRA:Spatial CGRA

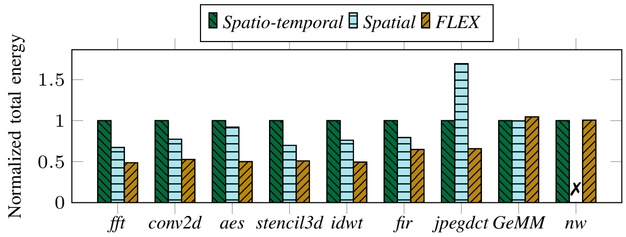


Fig. 9: Total energy comparison normalized w.r.t. spatio-temporal CGRA.

temporal vector dataflow shows 45% average reduction in total power compared to the spatio-temporal CGRA baseline. Moreover, *FLEX* achieves nearly the same low power as the spatial CGRA. For increased vector length (*stencil3d*), *FLEX* consumes even lower power than spatial CGRA. Power savings of *FLEX* is mainly due to the less frequent reconfiguration. Figure 8b depicts how *FLEX* overcomes the significant reconfiguration power in spatio-temporal CGRA by reducing the configuration memory access power by an average of 63% Similarly, *FLEX* avoids unwanted data memory access in spatial CGRA, saving 45% power (Figure 8c).

*3) **Energy**:* We compute the energy consumed by each architecture for the complete execution of a loop kernel. Figure 9 compares the total energy consumption normalized w.r.t. that of spatio-temporal CGRA. *FLEX* delivering higher performance at lower power shows significant improvement in energy consumption compared to the baselines. Spatio-temporal vector dataflow execution in *FLEX* achieves an average energy reduction of 45% and 35% compared to spatio-temporal CGRA and spatial CGRA baselines, respectively.

*4) **Power efficiency**:* Figure 10 shows the effective power efficiency. *FLEX* achieves a maximum of 373 MOPS/mW power efficiency with an average of 327 MOPS/mW across all kernels, which is 1.9× and 1.6× improvement over spatio-temporal and spatial architectures, respectively. *FLEX* executing in either spatio-temporal (*nw*) or spatial (*GeMM*) execution delivers nearly the same power efficiency as the baselines. Moreover, *FLEX* can handle complex kernels like *jpegdct*

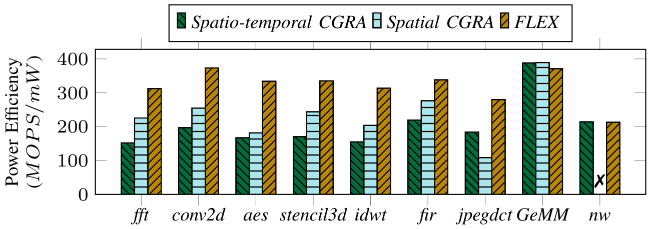


Fig. 10: Power Efficiency

TABLE III: Area

Architecture	Area (mm^2)
Spatio-temporal CGRA	0.091
Spatial CGRA	0.095
FLEX	0.100

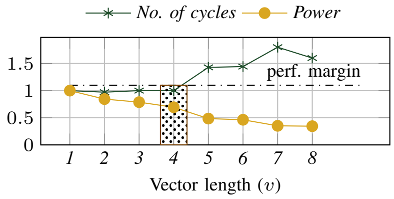


Fig. 11: DSE on *fir* kernel

much better.

We can conclude that *FLEX* provides a better tradeoff between power and performance, achieving the lowest energy consumption and best power efficiency among the compared architectures.

*5) **Area comparison**:* Table III compares the total area of the architectures. *FLEX* has a slight area increase (<10%) mainly due to the partitioned configurations for router and compute. Marginal area increase in spatial CGRA (<5%) is due to the increased size of data and configuration memories.

*6) **Flexibility**:* For any given kernel, *FLEX* DSE allows users to explore spatio-temporal, spatial, or spatio-temporal vector dataflow execution with different vector sizes. Figure 11 depicts DSE on *fir* kernel. Performance and power values are calculated for all $v \in \{1, 2, 3, .., 8\}$ (normalized w.r.t. v=1 in Figure 11) and v=4 is selected, which meets the performance goal with minimum power (achieves power efficiency of 338 MOPS/mw).

E. FLEX on SIMD CGRA

SIMD execution on CGRA [19], [20] can improve performance and reduce reconfiguration. The spatio-temporal vector dataflow execution in *FLEX* is orthogonal to SIMD and hence can be coupled with SIMD to provide even better efficiency. To demonstrate this advantage, we introduce minimal additional resources required by *FLEX* (vector counters, link registers, and additional clock gating cells) on a SIMD CGRA. Figure 12 shows the energy estimation on SIMD spatio-temporal CGRA (SIMD factor of 2) and *FLEX* on SIMD CGRA (SIMD factor of 2) normalized w.r.t. generic spatio-temporal CGRA. *FLEX + SIMD* achieves 54% and 48% energy improvements over spatio-temporal and SIMD spatio-temporal CGRAs, respectively. We note that kernels requiring complete or partial sequential execution are not compatible with SIMD, but they can be executed well on standalone *FLEX* (e.g., *idwt* and *fir* with accumulation).

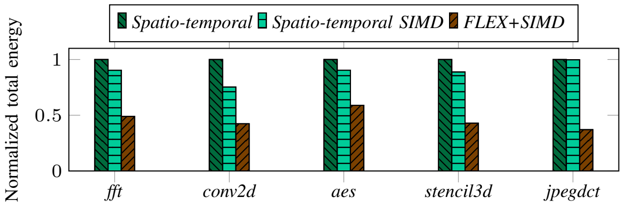


Fig. 12: *FLEX* on SIMD CGRA

TABLE IV: *FLEX* compared to SOTA CGRAs

	UE-CGRA [21]	**HyCUBE** [22]	**SNAFU*** [3], [23]	**TRANSPIRE** [24]	**RIPTIDE** [23]	**FLEX** (This work)	
Execution	S	ST	S	ST	S	ST/S/STV	
Tech node	TSMC 28	40nm CMOS	Intel 22FFL	28 FDSOI	Intel 22FFL	22 FDSOI	
Setup	Post P&R	Post-synthesis simulation					
Frequency (MHz)	750	853	50	50	50	100	
Benchmark	fft			**	fft	fft	**
Power (mW)	14	72	0.54	-	0.24	**2.56**	**2.40**
Throughput (MOPS)	625	6483	71	-	62	**800**	**786**
Power Effi. (MOPS/mW)	45	90	134	166	254	**313**	**327**

* Based on the latest values mentioned in Table III of RipTide [23] paper.
** Average across all the benchmarks reported.

F. *FLEX vs. SOTA CGRAs*

Table IV compares *FLEX* against SOTA CGRAs w.r.t. power, performance, and efficiency (reported numbers are absolute simulation results from respective sources). This is our best effort to make a fair comparison because of the inconsistent details among sources. SOTA CGRAs follow a fixed execution, either spatio-temporal (ST) or spatial (S), while *FLEX*'s execution is flexible. Spatial CGRAs achieve lower power but at the cost of performance. Spatio-temporal CGRAs deliver higher performance but with high power budget. *FLEX* balances power and performance with an improved average power efficiency of 327 MOPS/mw.

V. Related work

A. *Spatial and Spatio-temporal CGRAs*

We categorize SOTA CGRAs into two categories: spatial and spatio-temporal. In spatial architectures, the configuration remains static for complete loop execution. Spatial architectures like Softbrain [10], Tartan [25], Piperench [26], Warp [27], and FPCA [28] follow a static instruction schedule, whereas SNAFU [3], DySer [29], Plasticine [20], RIPTIDE [23], and Q100 [30] trigger instruction firing on data arrival. Spatial execution in SNAFU [3] is mentioned as spatial vector-dataflow execution, and kernels are written in vectorized C code. Spatio-temporal architectures add temporal multiplexing to the spatial scheduling of operations. ADRES [1], HyCUBE [2], Morphosys [9], MATRIX [31], and Remarc [32] adopt static scheduling. TRIPS [33], SGMF [34], Wavescalar [35], and dMT-CGRA [36] allow data-triggered execution.

FLEX covers the entire spectrum from spatial to spatio-temporal execution with its flexible temporal multiplexing at varying frequencies. Most of all, SOTA CGRAs adhere to a fixed execution model while *FLEX* provides a flexible model that can be configured based on application requirements.

B. *Exploiting Parallelism in CGRA Execution*

Several prior CGRAs exploit parallelism beyond SISD for performance improvements. The spatio-temporal vector dataflow in *FLEX* is orthogonal to these approaches and can be combined with minimal changes. SIMD RA [37] supports dynamic reconfiguration of modular regions which operate in SIMD fashion. Smartcell [38] supports both SIMD- and MIMD- parallelism. Plasticine [20] supports SISD- and SIMD-type parallelism and can natively map vector operations. On the other hand, SGMF [34] focuses on the SIMT model. TRANSPIRE [24] supports SIMD-type parallelism with customized binary8 data type. Bio-signal processing CGRAs like BioCare [39], HEAL-WEAR [40], and [41] also employs SIMD execution. Blocks [43] proposes runtime construction of VLIW-SIMD processors on reconfigurable fabric. [19] enables reschedulable dataflow and SIMD execution for improved utilization.

FLEX extended with SIMD achieves 48% energy improvement over a generic SIMD CGRA.

C. *Power Management Techniques*

System-level and circuit-level approaches are proposed to reduce the power consumption in CGRAs. REVAMP [44] proposes a systematic framework for deriving low-power heterogeneous CGRAs from homogeneous CGRAs. On-the-fly CGRA [50] proposes approximate computing for reduced power. UPTPU [46] improves energy efficiency by power-gating the MAC units based on their idleness. Samsung's ULP-SRP [5] adopts fine-grained power gating and dynamically switches between three performance modes for low-power design in biomedical devices. IPA [47] proposes an array of PEs that operate at a near-threshold voltage and clock-gates idle PEs to reduce dynamic power. i-DP CGRA [48] clock-gates idle PEs and interleaves the datapaths to execute two instructions concurrently within each PE. SysCore [49] leverages DVFS for power benefits, whereas Ultra-Elastic CGRAs [21] optimize energy with DVFS for irregular loops.

Traditional power management techniques add significant overhead on low-power CGRAs. With its lightweight power-efficient configuration structures, *FLEX* enables configuring of the hardware to suit each application so that simple clock gating can deliver 45% power savings.

VI. Conclusion

CGRAs provide an excellent balance between performance, power efficiency, and adaptability. However, the current CGRA reconfiguration paradigm fails to reconcile low-power with high-performance for complex application kernels. We propose *FLEX*, a novel CGRA architecture with a flexible execution paradigm that can easily adapt to the application requirements delivering an average of 1.6×-1.9× power efficiency over traditional CGRAs.

Acknowledgment

This research is partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003.

REFERENCES

[1] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *FPL'03*.

[2] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC'17*.

[3] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture," in *ISCA'21*, pp. 1027–1040.

[4] A. Carsello et al., "Amber: A 367 gops, 538 gops/w 16nm soc with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," in *VLSI'22*, pp. 70–71.

[5] C. Kim et al., "Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications," in *FPT'12*.

[6] K. E. Fleming et al., "Processors, methods, and systems with a configurable spatial accelerator," Feb. 11 2020, uS Patent 10,558,575.

[7] T. Fujii et al., "New generation dynamically reconfigurable processor technology for accelerating embedded ai applications," in *VLSI'18*, pp. 41–42.

[8] M. Emani et al., "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science & Engineering'21*, vol. 23, no. 2, pp. 114–119.

[9] H. Singh et al, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," in *IEEE Trans. Comput.'00*.

[10] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA'17*.

[11] D. Wijerathne, Z. Li, M. Karunaratne, L.-S. Peh, and T. Mitra, "Morpher: An open-source integrated compilation and simulation framework for cgra," *WOSET'22*.

[12] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for fpgas," in *FPGA'95*, pp. 111–117.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *DATE'03*, pp. 296–301.

[14] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E'04*, vol. 70.

[15] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *IISWC'14*.

[16] L.-N. Pouchet, "Polybench/c." [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[17] "C Implementation of Discrete Wavelet Transform ." [Online]. Available: https://github.com/rafat/wavelib

[18] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," 2013.

[19] C. Yin, N. Jing, J. Jiang, Q. Wang, and Z. Mao, "A reschedulable dataflow-simd execution for increased utilization in cgra cross-domain acceleration," *TCAD'23*.

[20] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel patterns," in *ISCA'17*, pp. 389–402.

[21] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *HPCA'21*, pp. 412–425.

[22] B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "Hycube: A 0.9v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications," in *A-SSCC'19*, pp. 133–136.

[23] G. Gobieski et al., "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *MICRO'22*, pp. 546–564.

[24] R. Prasad et al., "Transpire: An energy-efficient transprecision floating-point programmable architecture," in *DATE'20*, pp. 1067–1072.

[25] M. Mishra et al., "Tartan: Evaluating spatial computation for whole program execution," in *ASPLOS'06*, p. 163–174.

[26] S. Goldstein et al., "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[27] M. Annaratone et al., "The warp computer: Architecture, implementation, and performance," *IEEE Trans. Comput.'87*, vol. C-36, no. 12, pp. 1523–1538.

[28] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *FCCM'14*, pp. 9–16.

[29] V. Govindaraju et al., "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *MICRO'12*, vol. 32, pp. 38–51, 2012.

[30] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *ASPLOS'14*, p. 255–268.

[31] E. Mirsky and A. DeHon, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *FCCM'96*, 1996, pp. 157–166.

[32] T. Miyamori and K. Olukotun, "Remarc : Reconfigurable multimedia array coprocessor," *Information and Systems'99*, vol. 82, pp. 389–397.

[33] K. Sankaralingam et al., "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA'03*, pp. 422–433.

[34] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," in *ISCA'14*, pp. 205–216.

[35] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO'03*, pp. 291–302.

[36] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *MICRO'18*, 2018, pp. 42–54.

[37] Y. Kim et al., "Exploiting both pipelining and data parallelism with simd reconfigurable architecture," in *ARC'12*, pp. 40–52.

[38] C. Liang and X. Huang, "Smartcell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications," *EURASIP J. Embedded Syst.'09*, no. 1, p. 518659, Jun.

[39] Z. Ebrahimi and A. Kumar, "Biocare: An energy-efficient cgra for bio-signal processing at the edge," in *ISCAS'21*, pp. 1–5.

[40] L. Duch et al., "Heal-wear: An ultra-low power heterogeneous system for bio-signal analysis," *TCAS-I'17*, vol. 64, no. 9, pp. 2448–2461.

[41] B. de Bruin et al., "Multi-level optimization of an ultra-low power brainwave system for non-convulsive seizure detection," *TBCAS'21*, vol. 15, no. 5, pp. 1107–1121.

[42] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," in *DATE'21*, pp. 1192–1197.

[43] M. Wijtvliet, A. Kumar, and H. Corporaal, "Blocks: Challenging simds and vliws with a reconfigurable architecture," *TCAD'22*, vol. 41, no. 9, pp. 2915–2928.

[44] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "Revamp: A systematic framework for heterogeneous cgra realization," in *ASPLOS'22*, p. 918–932.

[45] Z. Li, D. Wu, D. Wijerathne, and T. Mitra, "Lisa: Graph neural network based portable mapping on spatial accelerators," in *HPCA'22*, pp. 444–459.

[46] P. Pandey, N. D. Gundi, K. Chakraborty, and S. Roy, "Uptpu: Improving energy efficiency of a tensor processing unit through underutilization based power-gating," in *DAC'21*, pp. 325–330.

[47] S. Das, K. J. M. Martin, P. Coussy, and D. Rossi, "A heterogeneous cluster with reconfigurable accelerator for energy efficient near-sensor data analytics," in *ISCAS'18*, pp. 1–5.

[48] L. Duch et al., "i-dps cgra: An interleaved-datapaths reconfigurable accelerator for embedded bio-signal processing," *ESL'19*, vol. 11, no. 2, pp. 50–53.

[49] K. Patel, S. McGettrick, and C. J. Bleakley, "Syscore: A coarse grained reconfigurable array architecture for low energy biosignal processing," in *FCCM'11*, pp. 109–112.

[50] M. Brandalero, A. C. S. Beck, L. Carro, and M. Shafique, "Approximate on-the-fly coarse-grained reconfigurable acceleration for general-purpose applications," in *DAC'18*, pp. 1–6.

[51] Z. Li, D. Wijerathne, and T. Mitra,"Coarse Grained Reconfigurable Array CGRA," Book Chapter in Springer Handbook of Computer Architecture 2022.