4D-CGRA: Introducing Branch Dimension to Spatio-Temporal Application Mapping on CGRAs

Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, Li-Shiuan Peh
School of Computing, National University of Singapore
{manupa,dmd,tulika,peh}@comp.nus.edu.sg

ABSTRACT

Coarse-Grained Reconfigurable Arrays (CGRA) are a promising class of accelerators that provide good balance between flexibility, performance, and power. As the CGRAs are designed to support dataflow, the acceleration is limited to loops with simple control flows. The compiler generates static schedules of loop kernels on the CGRA and completely eliminates the burden of resource conflict resolution from the hardware. In the presence of complex control flows, the static scheduling on CGRA requires independent resource reservations for mutually-exclusive dataflows along control-divergent paths. Such reservations are not only wasteful but also limit performance by increasing the schedule length. We introduce a novel architecture, 4D-CGRA, that encourages mutually-exclusive dataflows to map to the same set of resources but allows execution of the appropriate dataflows at runtime based on the branch outcomes. We achieve this by introducing an architecture-enabled new branch dimension corresponding to the branching decisions. We design a novel compiler to model integrated placement and routing in four dimensions (two spatial, one temporal, one branch). 4D-CGRA achieves upto 2.33x (average 1.44x) performance gain compared to a generic CGRA, with the same area, power budget.

1. INTRODUCTION

Internet-of-Things (IoT) devices demand high performance at low power. ASICs can accelerate workloads on such platforms but suffer from lack of flexibility, while FPGAs provide complete reconfigurability down to the bit level, leading to high area, power overheads. CGRAs (Coarse-Grained Reconfigurable Arrays) retain configurability but with a coarser operation-level granularity.

A CGRA, shown in Figure 1, is essentially a set of processing elements (PE) arranged in a grid with each PE connected to its neighbors. Each PE typically consists of a simple ALU, a register file and a control memory to store configuration information (instructions). An on-chip multi-bank scratchpad memory (SPM) feeds data to the entire array during execution. CGRAs are statically scheduled [1] with the compiler targeting frequently executed loops for acceleration, mapping the operations onto individual PEs, and configuring the ALUs and inter-PE interconnects appropriately to handle dependencies among the operations. The schedule covers all the PEs and interconnect for a fixed number of cycles per loop iteration and the schedule is repeated for all iterations. Thus the schedule is defined along both the spatial (PEs) and the temporal (cycles) dimensions. Therefore, each PE is functioning in a carefully engineered lock-step execution model that does not incur stalls in compute or communications. This is the key

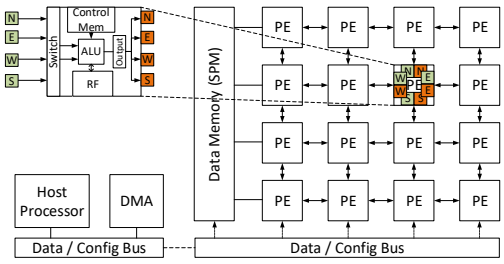

Figure 1: A high-level view of CGRA architecture

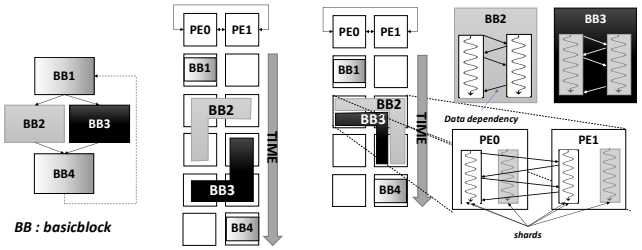

Figure 2: (a) Control-flow graph of a loop body; Mapping onto (b) generic CGRA and (c) 4D-CGRA with shards.

feature of CGRAs. Recently, CGRAs have been deployed commercially, for example, in Samsung Exynos 7420 SoC[2] of Samsung Galaxy S6, Configurable Spatial Accelerator from Intel[3], Movidius Myraid 2[4] (Intel), Wave DPU[5] (Wave Computing) among others.

The state-of-the-art statically-scheduled CGRAs are excellent at supporting dataflow, but limited in their ability to accelerate loops with complex control flow. They rely on predication [6] to handle control divergence, where the compiler maps both paths of each conditional branch onto the CGRA, but only the instructions from the taken path (with true predicate value) are permitted to execute at runtime. The predication effectively replaces the control flow with dataflow of predicate values. Figure 2a shows the control flow graph of a loop with control divergence, consisting of mutually exclusive basic blocks BB2 and BB3. The resources allocated for the different control paths cannot overlap in either the spatial or the temporal dimensions as shown in Figure 2b. This leads to static allocation of duplicate resources that are left unused at runtime, resulting in increased schedule length and limiting the performance. The goal of our proposed *4D-CGRA* architecture is to reduce this wastage of resources to support complex control flows and thereby shorten the schedule length

and substantially improve power, performance.

As the paths at control-divergence point are mutually exclusive, an obvious choice to improve resource utilization is to overlap their mapping in the schedule as shown in Figure 2c. Each loop iteration can now complete in four cycles instead of five, improving performance. This simple solution, however, turns out to be exceedingly difficult to support with the current execution model, where in each cycle, a PE simply executes the only operation scheduled on it (provided the predicate is true). If multiple operations, albeit from mutually exclusive paths, are mapped onto a PE in the same cycle, the PE needs to identify and execute the correct operation at runtime depending on the branch outcome. With only one conditional branch in the loop body, a simple hardware mechanism can pick between two potential candidates per PE per cycle. However, the loops with complex control flows (e.g., multiple conditionals, nested conditionals, switch statements etc.) lead to multiple candidate instructions to choose from, resulting in high complexity. The hardware overhead now involves associative search of all the candidate instructions to select the correct one. Moreover, CGRAs heavily rely on software pipelining [7] to improve throughput by concurrent execution of multiple loop iterations. A PE needs to distinguish not only the operations from different paths within a loop but also the operations from different in-flight loop iterations (with possibly different branch outcomes).

We propose a novel execution model to address the challenges with juxtaposed scheduling of control-divergent paths in the presence of arbitrarily complex control flows within the loops on CGRAs and runtime selection of instructions at the PEs. The distinguishing feature of our approach is that we restrict the dynamic selection to only a fraction of the instructions and leverage the static schedule as much as possible. Our design exploits the fact that all the instructions within a basic block (straight line code sequence with a single entry, single exit) have the same decision in terms of the control flow. Thus once a PE selects one instruction within a basic block, it can execute the rest of the instructions from that basic block without further considerations. Note that the instructions from a basic block are scheduled (scattered) across multiple PEs to take advantage of the instruction level parallelism (ILP). We define a *shard* as the subset of instructions from a basic block mapped onto a single PE and the first scheduled instruction within the shard as *header*. A basic block is split into multiple shards, one per participating PE with possible data dependencies among them. Multiple mutually exclusive shards are mapped onto the same PE as shown in Figure 2c. Based on the runtime control flow outcomes, a PE selects a shard by its header and then executes the rest of the instructions from that shard without further evaluation of the control flow conditions. Therefore, the perception of multiple instructions being scheduled in the same cycle on the same resource (compute or route) can be viewed as scheduling along a new *branch dimension*. The planes in the branch dimension create multiple configurations for the same spatio-temporal resource, while each plane corresponds to a specific branch outcome — hence the name 4D-CGRA.

We then design an enhanced CGRA architecture and corresponding compiler specifically for 4D-CGRA execution model. The PE supports limited runtime selection of the shards, and

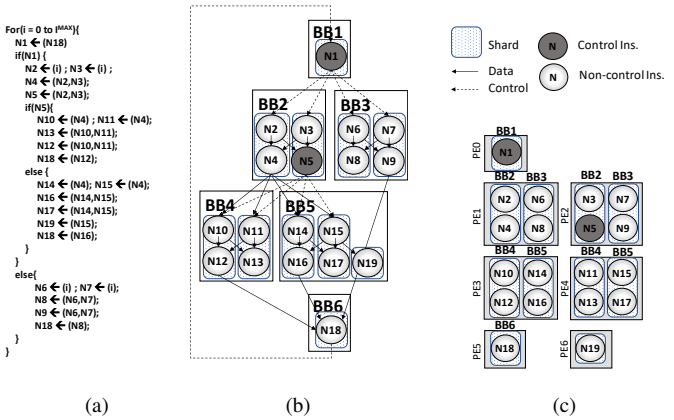

Figure 3: (a) Loop with nested conditionals, (b) Breakdown of basic blocks into shards, (c) Mapping.

allows the compiler to juxtapose the schedules belonging to the shards of control-divergent paths onto a PE without increasing the schedule length. Similarly, the interconnect enables mutually-exclusive shards to share the same links, scheduled by the compiler.

4D-CGRA is truly a hardware-software co-designed architecture that achieves upto 2.33x performance-per-watt improvement (average 1.44x) compared to a generic CGRA with the same area/power budget. 4D-CGRA has an area of 0.44mm^2 at 40nm technology node and consumes 17-66mW@714MHz, effectively providing configurable, ultra-low-power acceleration for IoT devices.

2. 4D-CGRA EXECUTION MODEL

We introduce a novel *4D execution paradigm* to support complex control flows within loop body. The model allows sharing of CGRA resources among mutually-exclusive execution paths and dynamic selection of the appropriate path.

Shard. The central concept here is the shard, a pre-defined sequential execution order of a subset of instructions within a basic block. A basic block is split into multiple shards and each shard is exclusively mapped to a single PE. The size of the shard could vary from a single instruction to the size of basicblock, depending on the amount of parallelism and dependencies in a basic block. The size of a shard is not predetermined, and instead is a consequence of performing the instruction mapping to CGRA fabric in a 4-Dimensional space (See Section 4).

The first scheduled instruction of each shard is called the *header*. For example, in Figure 3, we have two shards for basic block BB2: ⟨N2, N4⟩ in PE1 with N2 as the header and ⟨N3, N5⟩ in PE2 with N3 as the header. There can be data dependencies among the shards that lead to inter-PE communication. Multiple shards from mutually-exclusive execution paths may map to the same PE, for example, ⟨N2, N4⟩ from BB2 and ⟨N6, N8⟩ from BB3 on PE1. Therefore, we need a mechanism to identify and execute the correct shard at runtime at each PE depending on the latest branch outcome. This mechanism is provided by the *expected* and *transitive* tags.

Expected tag. The control flows from a set of shards belonging to one basic block to another set of shards in a subsequent basic block along a specific execution path, determined

by the outcome of the conditional branches. Each shard is associated with a static expected tag based on the outcome of the latest conditional branch before reaching this shard. For example, the shards in BB4 and BB5 have expected tags with the outcome of N5. With software pipelining of multiple loop iterations, there is a possibility that multiple outcomes of the same conditional branch belonging to consecutive loop iterations can be in-flight within the CGRA fabric simultaneously. To distinguish these branch outcomes from different iterations, the expected tag is augmented with a modulo iteration counter that is incremented per iteration and wrapped around when it reaches the maximum number of pipelined iterations. Thus, the expected tag consists of (1) outcome of the latest conditional branch, and (2) modulo iteration counter. Moreover, the downstream shards in a join basicblock (e.g., BB6) may have more than one expected tag because they may be executed along multiple paths. Therefore, the arrival of any such tag from predecessor shards should trigger the downstream shard (e.g., the shards of BB6 may get executed due to the arrival of tags from BB4 or BB5).

Transitive tag. The flow of control is modeled through the transitive tags. The execution starts with a transitive tag at BB1. This transitive tag is inherited from one shard to another and is updated when it encounters a conditional branch at the end of a shard. A shard can be executed only when the transitive tag it receives matches its expected tag. For example, in Figure 3, the control instruction N5 in BB2 may update the transitive tag to execute all the shards in BB4 or all the shards in BB5, depending on the outcome. Moreover, not all the headers of the subsequent shards may have data dependencies with the control instruction of the preceding shard, thus predicate dependencies are added between such headers and the control instructions, to deliver the tag.

Empirically, the number of conditional instructions that defines branches rarely exceeds eight. Therefore, the tag (both expected and transitive) should contain 4 bits to differentiate such branches (3 bits) and their outcomes (1 bit). Similarly, the number of pipelined iterations is less than four for most scenarios. Thus, 2 bits are sufficient for the modulo iteration counter that is used to distinguish between pipelined iterations. So, the tag is 6 bits (4+2) and it is appended to 32-bit data transmitted throughout the fabric.

3. 4D-CGRA ARCHITECTURE

4D-CGRA architecture realizes the 4D execution paradigm through innovative design of the PE micro-architecture shown in Figure 4. In conventional CGRA, the instructions are fetched and executed sequentially based on the static schedule generated by the compiler. However, in 4D-CGRA execution paradigm, there are multiple shards mapped to a PE and one of them needs to be selected for execution. The *Control Memory* holds the instructions corresponding to all the shards with the instructions corresponding to a shard typically placed sequentially. Each instruction contains control information for (1) ALU, (2) Interconnect, and (3) Address of the next instruction in the shard (NCA). Thus, one could view the contents of the control memory as a composition of multiple linked lists, each corresponding to a shard.

To support the 4D-execution paradigm, each PE contains 16 control registers with (1) *Header Pointers* (HP) and (2) *Interim Pointers* (IP). First, to enable tag matching for activation of a shard, each PE maintains pointers to the headers of the shards in the (HP). In addition HP contains the source channel (see Section 3.1) of the transitive tag, the temporal cycle (see Section 3.1), and the expected tag itself.

Once the PE receives a transitive tag that matches with the expected tag of a shard header, the corresponding shard is selected for execution. The HP fetches the first instruction of the shard (the header instruction) from the control memory for execution. The retrieved instruction includes the address of the next instruction (next NCA) in the shard. A new (IP) register is allocated for the next instruction with the same tag (that resulted in the activation of the header) and the next NCA. Moreover, an identical shard belonging to the next iteration may execute prior to the end of executing the shard from the current iteration because of loop pipelining. Therefore, the modulo iteration counter of the HP is incremented to differentiate the tags among pipelined iterations. The NCA field of the newly allocated IP gets updated as the execution continues with the remaining instructions in the shard. Finally, there may be a control-flow join at the end of the shard. The first instruction of the linked shard (e.g., N18 in Figure 3b) is brought into the same IP and the execution continues for the linked shard. At the end of the linked shard, the IP is freed.

3.1 Detailed architecture

Tag matching. The 4D-CGRA execution paradigm allows the architecture to reduce tag matching overhead in two ways. First, tag matching is only required for the headers of the shards; the remaining instructions in the shards simply follow the execution. Second, the compiler determines the cycle at which each shard can potentially be selected. So when a transitive tag arrives at a PE at cycle t, it is compared against only the HPs waiting for the trigger in cycle t. The PE maintains a modulo cycle counter (*currentCC*) that increments and wraps around at the end of the schedule. Each HP has a cycle counter field (CC), also known as the *temporal cycle*, that indicates the cycle number in which it could potentially be activated. The compiler guarantees that at most one of the shard headers will match in both cycle counter and tag value per cycle. In 4D-CGRA, the transitive tag arrives to a PE along with the data through the same channel. These channels are the north, south,

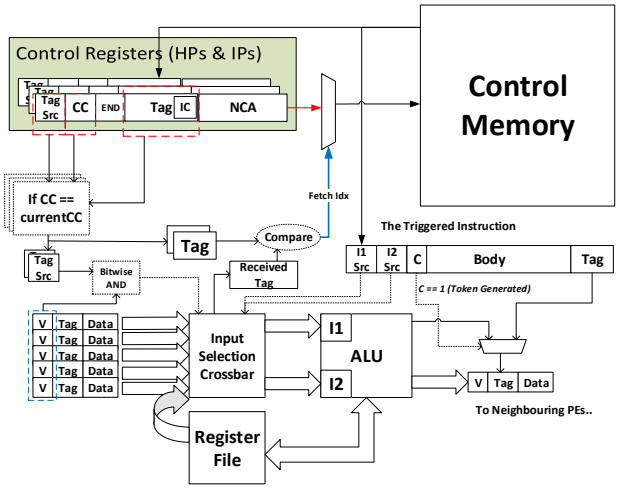

Figure 4: Architecture of 4D-CGRA processing element

east, and west input ports of the PE. Thus a HP waits for the activation trigger only on a specific input channel as specified in the static schedule, also known as the *source channel*.

Configuration word. Once the tag match is done, the NCA of the matched shard header is used to fetch the full configuration word (instruction) from the control memory. The configuration word contains all the information required to configure the PE and the interconnect to execute the current instruction (input sources for the ALU operands, the opcode for the ALU, and the output channels for transmitting the result) and an address to the next instruction (new NCA) in the shard. The output of an ALU consists of (a) valid bit, (b) tag, and (c) data. The valid bit is set to true after the tag-match and the data is the result of the ALU computation.

Thereafter, the output of the ALU is tranported to the destination PE. However at the compile time, the destination PE might be scheduled with multiple shards belonging mutually-exclusive execution paths. Thus, the source data for such shards might be expected from different source channels. However at runtime, only a single data would be arriving belonging the taken path. Thus, the corresponding source channel is used to identify the taken path. The tag is updated only for control instructions; other instructions simply inherit the existing tag in the control register for the corresponding live shard. The control instruction updates the tag by composing together branch ID (retrieved directly from the configuration word), branch outcome from the ALU output, and the modulo iteration counter (inherited from the iteration field of the existing tag in the control register of the current live shard).

3.2 Routing in 4D-CGRA

In fully statically-scheduled CGRAs, the compiler guarantees congestion free communication when multiple pairs of PEs exchange data in the same cycle to handle the data dependencies. In case of data dependency between two distant PEs, the data transfer takes place through one or more intermediate PEs in multiple cycles. The routing between distant PEs ties up multiple intermediate PEs and prevents them from performing useful computation. Recently, software-scheduled NoCs have been proposed as an alternative for CGRAs, to facilitate concurrent computation and data transfer in the intermediate PEs [8, 9]. A software-defined NoC ensures that the scheduler-generated static routes are completely congestion-free and thereby eliminates the routing and buffer management overheads of a dynamically routed NoC. To support software-defined NoC in the context of 4D-CGRA architecture, the routing control information has to be appended to every instruction. In 4D-CGRA, we allow data dependencies from mutually exclusive execution paths to share physical routing resources.

4. 4D-CGRA COMPILER

The compiler maps the loop kernels to 4D-CGRA architecture and is crucial in defining the performance. Figure 5 shows two compiled schedules of two mutually exclusive paths, that could be realized by mapping the operations from mutually exclusive basic blocks BB2 and BB3 to the extended spatio-temporal resources that belong to the branch planes N4-true and N4-false, as shown in the last sub-figure. The key constraints used in engineering the 4D-schedule is that the common operations (e.g., N9, N4) among the different

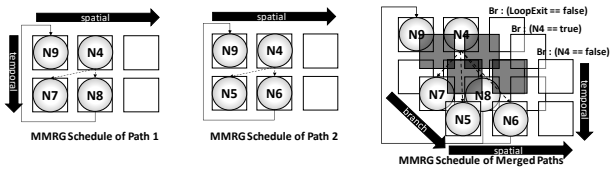

Figure 5: Merger of MRRG schedules from two mutually exclusive control paths.

paths are placed in the same spatio-temporal locations and extended bins for resources occupied by common operations are blocked in the new branch planes (N4-true & N4-false). An additional benefit of the compact merged schedule is that it allows the opportunity to power gate the unused PEs.

Figure 6 shows the 4D-compilation framework. It accepts as input the application source code in C with clearly annotated loops to be mapped to the CGRA. We utilize the Clang C compiler to generate the LLVM bitcode[10] for each annotated loop. The extracted loop body is analyzed for dependencies and then we generate the DFG.

4.1 DFG Generation

In the context of CGRAs, the control flows within the loop are handled via predication [6] that can be broadly classified as (a) partial predication and (b) full predication.

In partial predication, all the computations in the different control paths are allowed to execute but the results from only one of the control paths are committed based on the outcome of the control instructions. Thus partial predication favors generic CGRAs by reducing unnecessary serialization in the presence of control dependencies as generic CGRAs allocates scheduling slots for all the control paths anyway. Full predication, on the other hand, only allows execution of the instructions along the correct path based on the control flow outcomes and hence serializes execution in the presence of control dependencies. For 4D-CGRA, full predication is required to guarantee the selection of the correct path.

4.2 Place and Route Mapper

Given a DFG and a CGRA, the loop mapping is performed through modulo scheduling. We first determine the lower bound on Initiation Interval (II), denoted by *Minimum II (MII)*, as the maximum of the resource minimum II (ResMII) and recurrence minimum II (RecMII). The compiler attempts to map the loop starting with II set to MII and iteratively increments II by one until a feasible schedule is obtained. For each II value, we create a time-extended (II cycles) resource graph of the CGRA, known as Modulo Routing Resource Graph

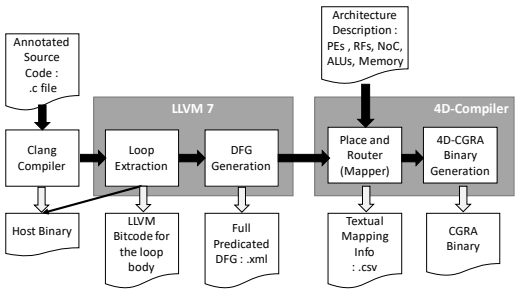

Figure 6: The 4D-CGRA compilation framework.

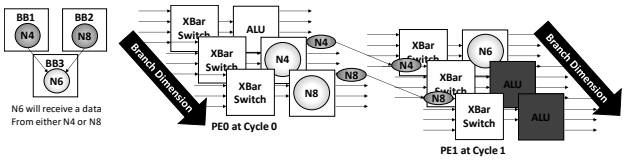

Figure 7: A mapping example with detailed MRRG.

(MRRG[11]). Note that the CGRA resources include the compute elements as well as the configurable data-paths inside (e.g., ALU to register file) and outside (PE-PE links) of the PEs.

Figure 7 illustrates the mapping of the fragment of a DFG in the upper left corner. Basic blocks BB1 and BB2 are mutually exclusive while BB3 is the join node. N4 and N8 are both mapped in PE0 in cycle 0, but in two different branch planes. Only one of them will be executed and send the data to N6 mapped on PE1. During execution, there are two possibilities in cycle 1 for PE1; either the output of N4 or N8 will appear in its west input port from PE0, sharing the routing links between N4 and N8.

Problem Definition. Given a fully-predicated DFG $D = (V_D, E_D)$ and a CGRA, the problem is to construct a minimally time-extended MRRG of the CGRA $H_{II} = (V_H, E_H)$ consisting of two type of nodes: PEs (V_H^F) and ports (V_H^P), for which there exists a mapping $\phi = (\phi_V, \phi_E)$ from D to $G_{II} = (V_G, E_G)$, where G_{II} is branch extension of H_{II} for which each MRRG node will contain contain multiple bins ($V^F_{G,b_i} \subseteq V_G$ and $V^P_{G,b_i} \subseteq V_G$), one for each branch outcome b_i. However, the usage of such bins is restricted to the nodes of the same iteration. For the next steps, we define a function $\lambda : V_G \rightarrow V_H$, that maps each of the branch-extended bins back to the spatio-temporal resource.

The modulo scheduling enables wraparound edges allowing a resource in the MRRG to be used in $\lceil L/II \rceil$ different timesteps where L is the schedule length. Therefore, any dependent node that include wraparound edges from the parent nodes, execute concurently with the nodes of the next iteration. Similiarly, two mutually exclusive instructions, with and without wraparound edges from the parent nodes, belong to two different loop iterations. Thus, the mutual exclusion no longer holds. Therefore, the bins of a branch plane b_i, corresponding to resources that are already mapped with instructions of another iteration in the branch plane b_j where $i \neq j$, are to be blocked.

In the mapping function, a DFG node $v \in V_D$ mapped to a bin of a MRRG node $\phi(v) \in V_{G,b_p} \subseteq V_G$ is associated with a timestep $R(v) = T(\phi(v)) * \tau$ where $\tau \in [0, \lceil L/II \rceil]$ is the modulo loop iteration and $T(\phi(v)) \in [0, II-1]$ is the cycle count (scheduling cycle) of the MRRG node. Two mutually exclusive DFG nodes u and v can be mapped to the same MRRG node, i.e., $\lambda(\phi(u)) = \lambda(\phi(v))$ only if $R(u) = R(v)$. The same criterion applies to ports that are used to route the data produced by the DFG nodes.

Mapping Algorithm with Branch Dimension. First, we extract the sub-DFGs, $D_{p_i} = (V_{D^{p_i}}, E_{D^{p_i}})$, for every possible execution path p_i in the loop body. We sort the set of sub-DFGs based on the number of nodes $|V_{D^{p_i}}|$ in decreasing order. We map each sub-DFG independently starting with the largest sub-DFG D_{p_M}. This is because the largest sub-DFG will likely require the longest II value for successful mapping. After mapping the first, D_{p_M}, for the rest of the sub-DFGs D_{p_i}, we only map the unmapped nodes – $\Delta D_{p_i} = (V_{D^{p_i}}, E_{D^{p_i}})$. Therefore, each of the nodes of ΔD_{p_i} will be utilizing new branch-planes that are mutually exclusive to previously mapped nodes.

Algorithm 1: 4D-CGRA Compiler Mapping Algorithm

```
1  DFG : D = (V_D, E_D) , CGRA : C = (V_C, E_C)
2  (P, D^P) = getSubDFGs(D) ; Sort((P, D^P)) ; {P_M, D^P_M} = Largest({P, D^P});
3  II = max( RecII(D),ResII(D^P_M, C) )
4  V_A ← mapped nodes ; E_A ← mapped edges ; B = {(b, o_b)} = getBranch(D);
5  while !success do
6      G = 4D_MMRG(C,II,B)
7      foreach (P_i, D^P_i) = (V_{D_i}, E_{D_i}) in (P, D^P) do
8          ΔD^P_i = (V_{D_i} − V_A, E_{D_i} − E_A)
9          if 4D_MAP(ΔD^P_i, G) then
10             V_A ← V_A ∪ V_{D_i} ; E_A ← E_A ∪ E_{D_i} ; success = 1 ;
11         else
12             success = 0 ; break;
13     II++
14 Function 4D_MAP(ΔD^P_i, G)
15     let ΔD^P_i = (V_Δ, E_Δ) ; TopologicalSort(V_Δ) ;
16     while !success OR !MaxIterations do
17         foreach v_Δ in V_Δ do
18             (b_{v_Δ}, o_{b_{v_Δ}}) = get_IDOM(v_Δ)
19             G_{(b_{v_Δ}, o_{b_{v_Δ}})} = (V_{G_b}, E_{G_b}) = getBranchPlane((b_{v_Δ}, o_{b_{v_Δ}}))
20             foreach v_g in V_{G_b} do
21                 if !checkR(v_a, v_g, λ) then
22                     continue;
23                 foreach v_p in Parents(v_Δ) do
24                     L[v_g] = L[v_g] ∪ LeastCostPath(φ(v_p), v_g)
25             (v_l, p_l) = min(L) ; assign(v_l, p_l);
26         if C_l = oversubscribe(G) is empty then
27             success = 1
28         else
29             IncreaseCosts(C_l) ; success = 0;
30     return success ;
```

We follow an iterative approach for mapping. Each node of the sub-DFG $u \in V_{\Delta D^{p_i}}$ is mapped to a bin of the PE node of MRRG $v^F \in V_G^F$ such that it utilizes the ports $v^P \in V_G^P$, that results in the least accumulated cost, when routing data from the parent nodes of u. The bin is selected based on the branch outcome that defines the execution of the node u. We employ Dijkstra's shortest path algorithm in establishing such routes and allow the ports to be over-subscribed if necessary. At the end of one iteration of mapping, the cost of the over-subscribed ports $v^P \in V_G^P$ are increased for future iterations (inspired by SPR [12]). The main intuition behind increasing the cost is to encourage the data to be routed through alternative routing resources; when the mapping converges, the resources with most demand are likely to be used for mapping the dependencies with fewer options for routing compared to the competitors. In subsequent iterations, the placement of node $u \in V_{\Delta D^{p_i}}$ may change to avoid over-subscribed resources from the previous iteration. We deem the mapping of the sub-DFG D_{p_i} a success where none of the resources are over-subscribed.

Furthermore, the nodes of $V_{\Delta D^{p_i}}$ are the new set of nodes that are executed because of the path p_i. They must hence be mutually exclusive to already mapped nodes of other paths : V_A. The branch outcome that determines the execution of a given node is determined by finding the terminating conditional instruction of the immediate dominator basicblock ($get\_IDOM()$ in Algo. 1). Therefore, that instruction (along with the outcome) is used to obtain the subset of MRRG resources corresponding to the branch plane ($getBranchPlane()$ in Algo. 1).

For each branch outcome b_p that is mutually exclusive to branch outcome b_q of V_A, the bins $v_{b_p} \in V_{G,b_p}$ that has

$\lambda(\phi(v_{b_p})) = \lambda(\phi(v_{b_q}))$ for $v_{b_q} \in V_{G,b_q}$ could be used to map nodes in $V_{\Delta D^{p_i}}$ if $R(v_{b_p}) = R(v_{b_q})$ (*checkR*() in Algo. 1). This allows overlapping of the MRRG resources for mutually exclusive nodes of the DFG. If a constituent execution path cannot be mapped after a fixed number of attempts, the MMRG is time-extended by one more cycle and we start mapping again from the largest sub-DFG.

4.3 Shard and instruction generation

After the place and route of the DFG on the MRRG, we have a spatio-temporal schedule with mutually exclusive instructions possibly mapped to the bins of the same spatio-temporal resources. For each PE (space dimension), the compiler then collects all the nodes in the schedule along each branch plane and creates a shard corresponding to each basic block. The shards from the different basic blocks mapped to a PE are placed in the control memory one after another, while the instructions within a shard are placed sequentially in the same temporal order as in the schedule. Morever, NOPs are only inserted if routing is required in the absense of any compute instruction. In the control memory, each instruction includes a pointer (NCA) to the next instruction from that shard; the terminating instruction of a shard points to the first instruction of the linked shard in case of control flow join or simply releases the control register. The first instruction of each shard is denoted as the header and its address is placed in *HPs*.

5. RELATED WORK

The compilers for CGRAs [11] exploit loop-level parallelism through modulo scheduling, with modern graph-based scheduling algorithms [1, 13, 14, 15] modeling the CGRA as a resource graph that captures both the spatial and the temporal dimensions. However, statically scheduled CGRAs handle the control-flow through *partial predication* (Figure 8) by executing all branch paths concurrently and performing a selection at the end using a select instruction. This leads to shorter runtime but resource and power wastage[6]. There are only a few efforts in the literature to support complex control flows in CGRAs. BrMap[16] in Figure 8 uses a dual-issue PE architecture allowing the PE to issue two instructions belonging to two mutually exclusive execution paths together. However, in the presence of complex conditionals (e.g., nested if-else), there is a need to merge multiple execution paths. TrMap[17] presents a mapping approach for statically scheduled CGRAs where each PE can issue instructions from the control memory based on received tokens. Performing such token matching in hardware with a fully-associative search for all instructions incurs huge power overhead. TrMap combines the instructions that are parents of the select instruction to be merged as one during compilation, prior to P&R. Moreover, the instructions in the divergent paths might require data inputs from instructions that are in common (i.e., common to all paths) basic blocks (blue arrows in Figure 8). Hence, if the merged instructions are to be placed in the spatio-temporal schedule, all dependencies need to be routed to that PE resource in the MMRG. This creates congestion which affects the schedule adversely, because the merging is done without considering the routability. In constrast to prior approaches, the 4D-CGRA compiler utilizes the architecture-enabled branch dimension to perform routing-aware merging of mutually exclusive instructions.

6. EVALUATION

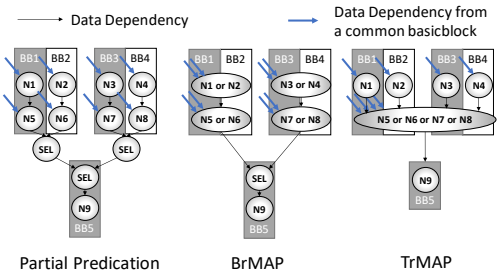

Figure 8: The state-of-the-art approaches of handling control-flow divergence in statically-scheduled CGRAs

We compare 4D-CGRA against several baselines: (1) A generic CGRA with completely static scheduling, with a PC (program counter) that points to the next instruction to be executed based on the compiled schedule and wraps around after II cycles, (2) BrMap [16], a CGRA with dual-issue based partial predication support, and (3) TrMap[17] that provides high-overhead complex control flows at the granularity of instructions (see Section 5).

We implemented all the above architectures in Verilog HDL and synthesized onto a commercial 40nm process using Synopsys Toolchain. We implement the PEs of each architecture to have similar configuration except for the control registers. Each PE contains a register file with four registers, two read ports, and two write ports, a second register file to hold up to eight constants with a single read/write port[18], a control memory that holds up to 32 instructions, a typical ALU, and a crossbar switch to provide the ALU with data. The PEs in the left-most column of the CGRA are connected to an on-chip 4KB scratchpad memory.

Benchmarks. We use two benchmark suites, Mediabench[19] and Cortexsuite[20] that contain computationally heavy loops suitable for CGRA acceleration. We observe that 55% and 73% of the most compute intensive loops consist of at least one if-else structure within the loop body, in Mediabench and Cortexsuite, respectively. This motivates the need for CGRAs to handle loops with complex control divergence. Not all the loops can be mapped onto CGRAs due to factors such as dynamic memory allocation, which are beyond the scope of this work. Therefore we include two other kernels from MachSuite[21] and WaveLib[22] to cover more domains.

Table 1 shows the kernels used in our evaluation. The main criteria for the selection of the kernels is the presence of

Benchmark (Suite)	Domain	Instructions including SELECT	Max Ins. executed per iteration	Conditional Structure
dwt (Wavelib)	Signal Processing	113	58 (51.33%)	1) if-elseif-elseif-elseif{if-else}-elseif-elseif{if-else}
taylor (CortexSuite)	Linear Algebra	75	59 (78.67%)	1) if-elseif-else 2) if-elseif-else
g721 (Mediabench)	Speech Processing	57	27 (47.37%)	1) if-else{if {if-else} - else {if-else}}
gsm (Mediabench)	Communications	87	36 (41.38%)	1) if-elseif-elseif-else
nw (Machsuite)	Dynamic Programming	68	39 (57.35%)	1) if-elseif-else
mpeg2enc (Mediabench)	Video Compresion	157	54 (35.06%)	1) if {if-else} - elseif {if - else} - elseif {if - else} - else {if - else}
mpeg2dec (Mediabench)	Video Compression	149	48 (32.21%)	1) if {if-else} - elseif {if - else}-elseif {if - else} - else {if - else}
word2vec (CortexSuite)	Machine Learning	147	65 (44.22%)	1) if { if - else } - else 2) if { if - else } - else
superres (CortexSuite)	Computer Vision	74	59 (79.73%)	1) if-elseif-else 2) if-elseif-else

Table 1: Benchmark characteristics

Kernel	Rec. II	Mapped II (Res. II)				Compilation Time			
		Generic	BrMap	TrMap	4D	Generic	BrMap	TrMap	4D
dwt	5	10 (8)	8 (6)	11 (5)	5 (4)	35m	32m	342m	13m
taylor	4	7 (5)	5 (4)	8 (5)	5 (4)	2m	9m	30m	41m
g721	2	5 (4)	5 (3)	5 (3)	3 (2)	0.3m	96m	17m	11m
mpeg2enc	9	14 (10)	14 (10)	14 (10)	12 (4)	30m	30m	30m	36m
superres	4	5 (5)	5 (4)	5 (4)	5 (4)	0.3m	5m	2m	44m
nw	11	16 (5)	15 (4)	16 (3)	12 (3)	2m	14m	12m	3m
mpeg2dec	9	13 (10)	13 (10)	13 (10)	12 (3)	31m	31m	31m	12m
word2vec	13	17 (10)	16 (6)	13 (5)	17 (5)	9m	14m	175m	21m
gsm	1	7 (6)	7 (5)	8 (5)	3 (3)	0.3m	63m	156m	35m

Table 2: Initiation Intervals for 4x4 CGRAs

control divergence (if-then-else, multiple conditionals, nested conditionals, switch statements, etc.) in the loop body. The third column shows the number of instructions that would be scheduled in a generic CGRA (including SELECT instructions to handle divergent paths). In 4D-CGRA, the number of instructions to be scheduled is reduced as select operations are no longer needed. The fourth column shows the number of instructions in the longest execution path that should be scheduled (and most likely determine the II) in the case of 4D-CGRA. We observe that this can vary between 32% to 79% (Average 52%) compared to the number of scheduled instructions of generic CGRA, because of control divergence. This motivates 4D-CGRA to utilize the branch planes to share the same spatio-temporal resources among mutually exclusive instructions. The last column indicates the type of control divergence present within the loop.

RTL power and area estimates. Figure 9 shows the area and power breakdown of the four architectures (same area, power for generic and BrMap). The generic CGRA consumes lowest area/power and TrMap consumes highest area/power, while 4D-CGRA falls in-between. It should be noted that BrMap[16] takes up the same area and power as Generic CGRA, because it employs dual-issue of two mutually exclusive instructions, and thus can be implemented with the same amount of configuration memory, but at double-width and half-height. For 4D-CGRA, the addition of control registers (CR) leads to 26% and 29% overhead in area and power, respectively, w.r.t. a generic CGRA. TrMap has much higher area and power overheads: 60% and 78% over that of a generic CGRA, respectively, due to the complex token matching hardware required. By leveraging static scheduling within shards, 4D-CGRA can achieve much lower power and area costs with its reduced scope of triggering, down from the number of instructions to the number of shards.

Throughput comparison. We first target 4x4 instances of the four architectures. Table 2 shows the recurrence II, achieved II through mapping the kernels to four different architectures and the compilation time. The resource II changes for the four architectures because of the variation in the degree of sharing among execution paths. The BrMap merges instruction pairs from various execution paths and the TrMap merges parents of SELECT operations. 4D-CGRA needs to schedule at least the largest execution path in the best case (assuming the rest of the paths are able to share resources with the largest execution path). 4D-CGRA has the least II value for all the benchmarks, resulting in highest throughput.

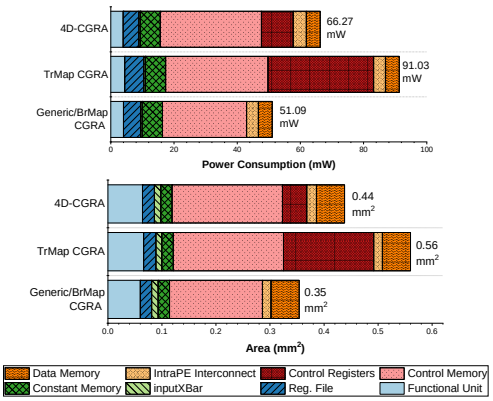

Figure 9: Area and power breakdown of 4x4 CGRA architectures

Figure 10 shows the throughput of each kernel, normalized to that of 4x4 Generic CGRAs. On average, 4D-CGRA, BrMap, and TrMap achieve 54%, 5%, to 4% performance gain compared to a generic CGRA. BrMap's performance suffers because it can only merge two instructions, and the merging ignores routing resources. TrMap's lower performance is because it indiscriminately merges parents of SELECT operations prior to P&R. 4D-CGRA presents a new paradigm to CGRA mapping by proposing the new branch dimension to be used in both computation and routing of mutually exclusive nodes, leading to resource-aware merging of multiple mutually exclusive instructions. The benchmarks $dwt, taylor, g721, gsm, nw$ perform much better for 4D-CGRA compared to the other kernels.

The poor performance of *word2vec* and *superres* is due to 4D-CGRA using fully predicated DFGs whereas the others use partially predicated DFGs. When creating fully predicated DFGs, the calculation required to perform the branching decision is carried out prior to the execution of the instructions belonging to the branch-dependent basic blocks. Therefore calculation of the branching decision and instruction in the subsequent basic blocks are serialized. However, in partially predicated DFGs, the instructions of the subsequent basic blocks are allowed to execute unless they encounter a SELECT operation that require the output of the branching decision. Thus, at that point, the branch outcome will select one of the paths and disregard the rest. Therefore, given sufficient resources in the CGRA, partially predicated DFGs may perform better compared to the fully predicated DFGs, but at the cost of wastage of power/resources executing false paths. *word2vec* and *superres* are two kernels that exhibit the serialization penalty over the gains of sharing of resources. As for *mpeg2enc* and *mpeg2dec*, they perform only marginally better on 4D-CGRA than the baselines because they are already affected by the higher recurrence II present in the kernels, thus hitting a ceiling in the performance in 4D-CGRA.

Iso-Area comparison At 4x4 CGRA size, the three architectures have different area footprints. Therefore, we use

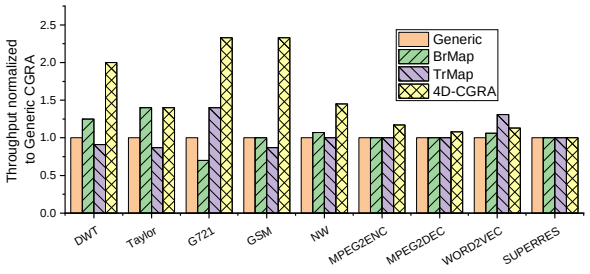

Figure 10: Throughput analysis (normalized to 4x4 Generic CGRA instance) of selected control-intensive kernels

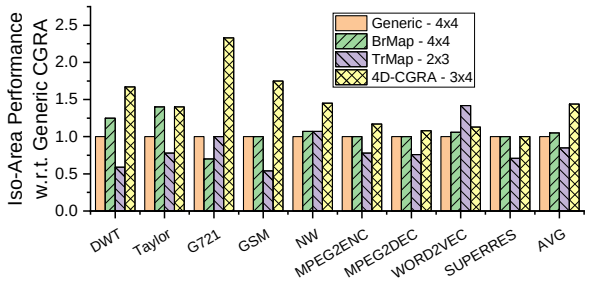

Figure 11: Iso-area performance w.r.t. generic 4x4 CGRA.

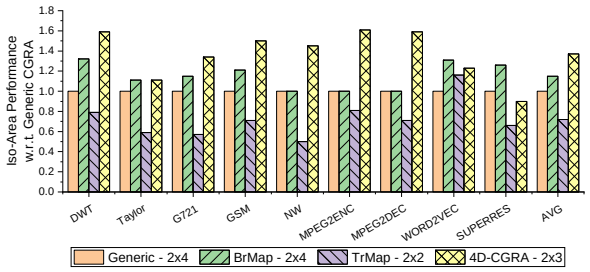

Figure 12: Iso-area performance w.r.t. generic 2x4 CGRA.

different array sizes across the different architectures to equalize their area overhead. We evaluate two iso-area configurations equivalent to 16-PE (4x4) generic CGRA and 8-PE (2x4) generic CGRA. The approximately area-equivalent array sizes for TrMap and 4D-CGRA are shown in Table 3. We observe that the area-equivalent configurations also consume approximately the same power.

Figures 11 and 12 show iso-area performance comparison w.r.t. 4x4 and 2x4 generic CGRA. On average, the 4D-CGRA performs 1.44x better, whereas BrMap and TrMap could only achieve 1.05x and 0.85x performance, respectively, compared to the 4x4 Generic CGRA, that consumes similar power and area. Moreover, performance/watt comparisons are identical to the performance comparisons because all the configurations are consuming about the same amount of power (1.44x, 1.05x and 0.85x normalized performance/watt gains for 4D-CGRA, BrMap and TrMap, respectively).

We next compare 4D-CGRA's performance against that of an iso-area 2x4 Generic CGRA in Figure 12. On average, 4D-CGRA performs 37% better with respect to generic CGRA, while BrMap and TrMap could only achieve 1.15x and 0.83x performance of the 2x4 generic CGRA. The highest performing $g721$ in Figure 11 is now affected by the reduction of resources. $g721$ was performing better due to higher degree of sharing of resources among mutually exclusive paths. However, instructions in a given mutually exclusive path, might have dependencies from instruction that are not unique to the path. Thus, such dependencies are unable to share routing resources. The reduction of fabric size also results in a reduction of possible routing paths which affects the routability of such dependencies. Therefore, $g721$'s degree of sharing reduces

	Generic	BrMAP	TrMAP	4D-CGRA
Config. 1	4x4(1.00)	4x4(1.00)	3x3(0.95)	3x4(0.96)
Config. 2	2x4(1.00)	2x4(1.00)	2x3(1.20)	2x3(0.98)

Table 3: Iso-area architectures corresponding to two baseline generic CGRA configurations 4x4 and 2x4. The parenthesized values indicate area normalized to generic CGRA.

in lower fabric size resulting in low normalized performance. The benchmarks $dwt, taylor, gsm, nw$ exhibit approximately the same normalized performance gain compared to generic CGRA at 2x4 iso-area configuration. However, *mpeg* kernels experience better normalized performance in 2x4 iso-area configuration because their 4D-CGRA performance are not affected by the constrained resources, unlike the generic CGRA. The *word2vec* and *superres* kernels are impacted more by constrained resources, because they were already suffering from the penalties of using full predication.

7. CONCLUSION

The novel 4D-CGRA brings a new architecture-enabled branch dimension that creates multiple planes for the same spatio-temporal resources to be used by the instructions from mutually-exclusive control paths. The architecture handles complex control divergence at very low-overhead by introducing the notion of shards, a sequence of instructions belonging to a basic block on a PE, and by lifting activation due to control flows at the granularity of shards rather than individual instructions. 4D-CGRA achieves up to 2.33X performance improvement (average 1.44X) compared to a generic CGRA with the same area/power budget.

8. ACKNOWLEDGMENTS

This work was supported by the National Research Foundation, Prime Ministers Office, Singapore under Grant NRF2015-IIP003 and by the NRF-RSS2016-005.

9. REFERENCES

[1] L. Chen *et al.*, "Graph minor approach for application mapping on CGRAs," *ACM TRETS '14*.
[2] C. Kim *et al.*, "ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications," in *FPT '12*.
[3] K. E. Fleming *et al.*, "Processors, methods, and systems with a configurable spatial accelerator," July 5 2018. US Patent App. 15/396,402.
[4] B. Barry *et al.*, "Always-on vision processing unit for mobile applications," *MICRO '15*.
[5] C. Nicol, "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing," *Wave Computing White Paper*, 2017.
[6] K. Han *et al.*, "Power-efficient predication techniques for acceleration of control flow execution on CGRA," *ACM TACO '13*.
[7] B. Mei *et al.*, "ADRES: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *FPL '03*.
[8] M. Karunaratne *et al.*, "HyCUBE: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC '17*.
[9] M. Gao *et al.*, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *HPCA '16*.
[10] C. Lattner *et al.*, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04*.
[11] B. Mei *et al.*, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEE Proceedings-Computers and Digital Techniques '03*.
[12] S. Friedman *et al.*, "SPR: an architecture-adaptive cgra mapping tool," in *FPGA '09*.
[13] M. Hamzeh *et al.*, "REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *DAC '13*.
[14] S. Dave *et al.*, "RAMP:resource-aware mapping for cgras," in *DAC'18*.
[15] M. Karunaratne *et al.*, "DNestMap: mapping deeply-nested loops on ultra-low power cgras," in *DAC '18*.
[16] M. Hamzeh *et al.*, "Branch-aware loop mapping on cgras," in *DAC '14*.
[17] S. Yin *et al.*, "Trigger-centric loop mapping on cgras," *TVLSI '16*.
[18] P. Theocharis *et al.*, "A bimodal scheduler for coarse-grained reconfigurable arrays," *ACM TACO '16*.
[19] C. Lee *et al.*, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO '97*.
[20] S. Thomas *et al.*, "Cortexsuite: A synthetic brain benchmark suite.," in *IISWC '14*.
[21] B. Reagen *et al.*, "Machsuite: Benchmarks for accelerator design and customized architectures," in *IISWC '14*.
[22] R. Hussain, "Wavelib : C Implementation of Discrete Wavelet Transform)." `https://github.com/rafat/wavelib/`.