HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction

Dhananjaya Wijerathne*, Zhaoying Li*, Anuj Pathania†, Tulika Mitra*, and Lothar Thiele‡
*National University of Singapore**, *University of Amsterdam*†, *Swiss Federal Institute of Technology Zurich*‡
dmd@comp.nus.edu.sg, zhaoying@comp.nus.edu.sg, a.pathania@uva.nl, tulika@comp.nus.edu.sg, thiele@ethz.ch

***Abstract*—Coarse-Grained Reconfigurable Array (CGRA) has emerged as a promising hardware accelerator due to the excellent balance among reconfigurability, performance, and energy efficiency. The CGRA performance strongly depends on a high-quality compiler to map the application kernels on the architecture. Unfortunately, the state-of-the-art compilers fall short in generating high quality mapping within an acceptable compilation time, especially with increasing CGRA size. We propose *HiMap* – a fast and scalable CGRA mapping approach – that is also adept at producing close-to-optimal solutions for multi-dimensional kernels prevalent in existing and emerging application domains. The key strategy behind *HiMap*'s efficiency and scalability is to exploit the regularity in the loop iteration dependencies by employing a virtual systolic array as an intermediate abstraction layer in a hierarchical mapping. Experimental results confirm that *HiMap* can generate application mappings that hit the performance envelope of the CGRA. *HiMap* offers 17.3x and 5x improvement in performance and energy efficiency of the mappings compared to the state-of-the-art. The compilation time of *HiMap* for near-optimal mappings is less than 15 minutes for 64x64 CGRA while existing approaches take days to generate inferior mappings.**

I. Introduction

The demand for hardware accelerators is rising with the increasing performance and low power requirements in modern application domains such as machine learning, signal processing, and multimedia. Reconfigurable accelerators are getting popular because they offer reasonable performance and energy efficiency while being flexible enough to support different applications. Field Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Arrays (CGRAs) have emerged as prominent reconfigurable accelerators. The bit-level reconfigurability in FPGAs offers more flexibility but comes at a higher cost of area and power. CGRAs offer word-level reconfigurability and therefore are more power-efficient than the FPGAs [1]. There have been many recent works on CGRAs in industry [1]–[3] and academia [4]–[8].

Figure 1 shows a CGRA consisting of an array of Processing Elements (PE) connected in a 2D mesh network. Each PE typically comprises of an Arithmetic Logic Unit (ALU), an internal Register File (RF), and a configuration memory. On-chip memory banks feed the data in and out of the array during execution. A PE executes an operation on the data it receives from the neighboring PEs or outside memory banks in each cycle. It either keeps the results in the RF or sends them to the neighboring PEs through output registers or both. The instructions in the configuration memory control both the operation execution (ALU configurations) and data routing between PEs (switch and RF port configurations). The CGRA compiler statically determines which operation should execute in which PE at which cycle (placement) and the data routes between the PEs according to the data dependencies (routing).

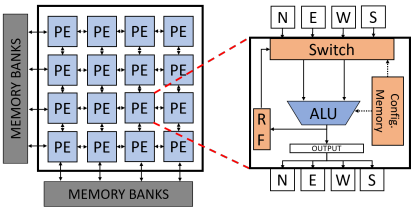


Fig. 1. An abstract block diagram for a 4x4 CGRA.

CGRAs are widely used to accelerate compute-intensive loop kernels. CGRA compilers exploit the inter- and intra-loop parallelism in the loop kernels via software pipelining [9]. Typically loop body of the kernel consists of many operations with irregular dependencies among them (intra-iteration dependencies). Existing compiler algorithms focused on mapping such irregular loop kernels on CGRA resources. They model this as a graph mapping problem between the Data Flow Graph (DFG) representing the loop body and the Modulo Routing-Resource Graph (MRRG) representing the hardware resources and their connections [10]. Due to the inherent complexity of the graph mapping problem, the compilation time increases substantially with the size (in terms of the number of nodes and edges) of the graphs (DFG and MRRG). Therefore, existing CGRA mapping techniques were evaluated using small loop kernels on CGRAs with a relatively small number of PEs.

Multi-dimensional loop kernels, i.e., loops with multiple nested loop levels, are ubiquitous in many popular applications. These kernels often have few operations in the loop body but unrolled kernels exhibit abundant instruction-level parallelism. Hence, these kernels are well suited to be accelerated on CGRAs with a large number of PEs. However, due to the inadequacy of compilers, the performance of the mapping is way below the ideal performance achievable. Moreover, dependencies between operations belonging to different iterations (inter-iteration dependencies) limit the exploitable parallelism in multi-dimensional kernels. Interestingly, unlike intra-iteration dependencies, inter-iteration dependencies have regular patterns as they are formed from the multi-dimensional iterators.

We propose a mapping approach, *HiMap*, that can achieve the ideal performance envelope for multi-dimensional kernels. Unlike conventional approaches, *HiMap* exploits the regularity of inter-iteration dependencies by first solving the mapping problem at the iteration level. *HiMap* creates a Virtual Systolic

Array (VSA) on top of the target CGRA. Then *HiMap* maps the iterations onto the VSA such that dependent iterations are placed nearby in space or time using a systolic mapping algorithm. Even though all the iterations have the same computations, the data dependencies among iterations make the routing different. However, there is a limited number of unique iterations in terms of both computation and routing. This paves the way to scalability as *HiMap* only has to generate detailed CGRA mappings of the few unique iterations. It can then replicate the unique iteration mappings to generate the complete CGRA mapping. *HiMap* achieves an average 2.8x better resource utilization compared to the conventional mapping approach on eight multi-dimensional kernels while achieving the optimal solutions for five of them. *HiMap* achieves 17.3x performance and 5x performance-per-watt improvement compared to the state-of-the-art. It takes only a few minutes to generate quality mappings, while the conventional mapping approaches take days to produce inferior mappings.

II. Motivating Example

We first present an example mapping schedule of a two-dimensional kernel on a linear (one-dimensional) CGRA array to show the effectiveness of our approach. BiCG is a two-dimensional kernel of BiCGStab Linear Solver [11]. Figure 2a shows the source code of tiled BiCG loop kernel where $(b1, b2)$ is the block size and (N, M) is the problem size. The loop blocking (tiling) breaks down the loop into smaller sized blocks when the problem size is bigger. We fully unroll the inner loops L3 and L4 to generate the DFG. The outer loops L1 and L2 invoke the inner two loops to complete the execution.

Figure 2b shows the simplified DFG of unrolled L3 and L4 loops for $b1 = b2 = 4$. Ellipses highlight the cluster of operations that correspond to each iteration in the two-dimensional iteration space formed by loops L3 and L4. Figures 2c and 2d show conventional and *HiMap* mapping schedules of the DFG on a 8x1 linear CGRA, respectively. The colored nodes belong to the current block of computation, while the white nodes represent the next block of computation. The Block Initiation Interval (II_B) is the time interval between the initiation of two successive blocks of computations. The lower the II_B value, the higher the CGRA PE utilization and thus higher throughput.

The conventional mapping schedule is irregular as the mapper does not see the regularity of the dependencies among operation clusters. Therefore, the conventional approach takes a long time to find an optimal placement and routing. *HiMap* exploits the regularity of iteration level dependencies to generate a regular schedule. It creates 4x1 VSA by clustering 8x1 CGRA. Each systolic PE (SPE) in VSA contains a sub-CGRA of size 2x1. Afterward, *HiMap* places iterations on SPEs such that dependent iterations are nearby either in time or space. *HiMap* yields higher throughput as the II_B value is lower than the conventional mapping. Even though there are 16 iterations, 9 of them are unique based on both routing and computation in *HiMap* schedule. The number inside each ellipse identifies the unique iterations. For example, iterations with ID 2 consume data from the north iteration and produce data to the south and south-east iterations. Iterations with ID 3 consume data from the north-west iteration and produce data to the south and south-east iterations. *HiMap* identifies and performs detailed mapping for these unique iterations. Then the unique iteration mappings are replicated to generate the final CGRA mapping schedule.

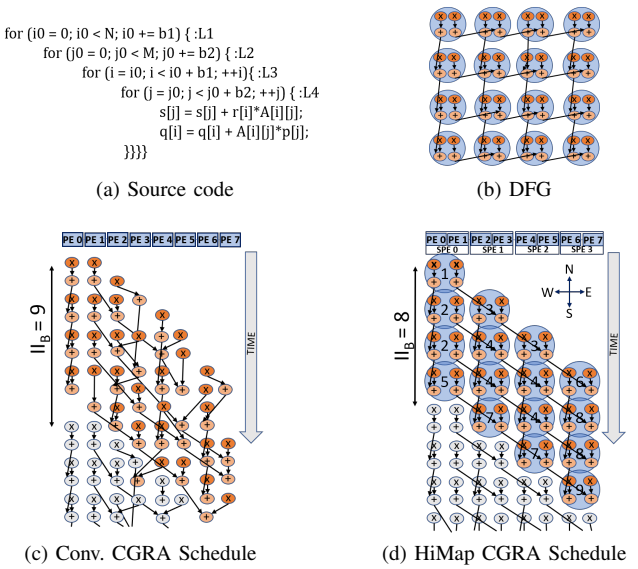
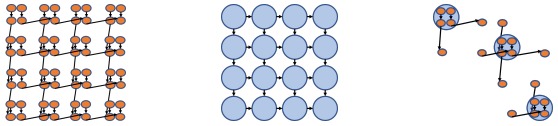


Fig. 2. Illustration of HiMap CGRA schedule of BiCG Kernel.

As the block size increases beyond 4 ($b1 > 4$ and $b2 > 4$), the number of unique iterations remains 9 because the unique iterations except those in corners (2, 3, 4, 7, and 8) will repeat in the mapping schedule. Since increasing block size does not necessarily increase the number of unique iterations, *HiMap* compilation time is not increased with the block size. Therefore, *HiMap* is capable of mapping bigger block sizes to fit bigger CGRAs. Bigger block sizes offer more parallelism that can be exploited in bigger CGRAs and also reduce the number of on-chip memory accesses because expanded DFG allows better reuse of data within the PE array.

III. Related Works

We can classify the existing CGRA mapping algorithms into three main categories: heuristic-based [6], [10], [12], [13], graph-based [5], [14], [15], or ILP-based [16]. Graph-based approaches formulate DFG to MRRG mapping using existing problems in graph theory. ILP-based methods define the mapping problem using a set of ILP constraints and solve the formulation using ILP solvers. However, these approaches are generally evaluated for loop kernels with relatively small DFGs and small CGRA sizes (4x4, 8x8). *HiMap* is motivated by systolic architectures [17]–[20] that maximize data reuse through regular systolic data flow [19] and thus minimize memory accesses [21]. It also results in conflict-free on-chip memory access and eliminates the need for conflict removal mechanisms [22]. However, CGRAs are far more flexible than systolic arrays. We can consider a systolic array as one specific instance of various possible CGRA mappings. *HiMap* generates this instance on CGRA and achieves the best of both worlds. For example, the dataflow of the systolic array in Google Tensor Processing Unit (TPU) [23] is the same as the dataflow of CGRA configured with the GEMM [24] kernel using *HiMap*.



(a) DFG (b) ISDG (c) IDFGs

Fig. 3. Illustration of graph definitions for BiCG kernel.

IV. Problem Formulation

Data-Flow Graph (DFG): We define DFG $D = (V_D, E_D)$ as a directed acyclic graph with V_D representing operations and E_D representing the dependencies between operations. $(b_1, b_2, .., b_l)$ represents the block size where l is the tiled loop level. Figure 3a shows the DFG of BiCG kernel ($b1 = b2 = 4$).

Iteration Space Dependency Graph (ISDG): ISDG is a directed acyclic graph $D^{'} = (\mathcal{C}, \mathcal{E})$ where vertices $\mathcal{C}$ represent the cluster of nodes belonging to the same iteration and the edges $\mathcal{E}$ represent the dependencies between clusters ($\mathcal{C} \subseteq \mathcal{P}(V_D)$ where $\mathcal{P}()$ represents the power set) [25]. Two clusters are connected if and only if there is a node in one that is connected to a node in the other. Figure 3b shows the ISDG of BiCG kernel. Each iteration in $\mathcal{C}$ is indexed with i and $\mathcal{C}_i^I$ represents the iteration vector of $\mathcal{C}_i$.

Intra-iteration Data-Flow Graph (IDFG): We define an IDFG to succinctly capture the interaction of operations in a single iteration. IDFG is a directed acyclic graph $D_i^{''} = (V_{D_i^{''}}, E_{D_i^{''}})$ with $V_{D_i^{''}}$ representing two types of nodes: computation nodes ($V^F_{D_i^{''}}$), and input/output nodes ($V^I_{D_i^{''}}$), and with edges $E_{D_i^{''}} \subseteq E_D$ representing the dependencies. $V^F_{D_i^{''}}$ are the nodes within the iteration $C_i \in \mathcal{C}$ and the $V^I_{D_i^{''}}$ are the nodes outside C_i which directly connects to the nodes inside C_i. Each iteration cluster $C_i \in \mathcal{C}$ is associated with an IDFG $D_i^{''}$. Three IDFGs of BiCG kernel are shown in Figure 3c.

CGRA Graph: CGRA is defined as a graph $G = (V_G, E_G)$. The size of the CGRA PE array is $c \times c$. VSA $G^{'}$ is formed by clustering CGRA PEs into sub-CGRAs ($G^{''}$) of size $s_1 \times s_2$.

Modulo Routing Resource Graph (MRRG): MRRG $H_{II} = (V_H, E_H)$ is a resource graph of the CGRA that is time extended to II_B cycles [10]. V_H consists of two types of nodes: ALUs (V_H^F) in each PE and ports (V_H^P) in interconnects and RFs [10]. As the CGRA schedule repeats after II_B cycles, the resources at cycle $II_B - 1$ have connectivity with the resources at cycle 0 in the MRRG. $H^{'} = (V_{H'}, E_{H'})$ is defined as the time extended resource graph of the VSA $G^{'}$ ($V_{H'} \subseteq \mathcal{P}(V_H)$). $H^{''} = (V_{H''}, E_{H''})$ is defined as the time extended resource graph of the sub-CGRA $G^{''}$.

Utilization: We define CGRA resource utilization U as the ratio between the number of operations in DFG D and the number of PEs in MRRG. $U = |V_D|/|V_H^F|$.

Problem Definition: Given a kernel and a CGRA, the problem is to construct a time extended MRRG $H_{II} = (V_H, E_H)$ of the CGRA for which there exist a mapping $\phi = (\phi_V, \phi_E)$ from $D = (V_D, E_D)$ to H_{II} which maximizes the utilization U. *HiMap* decomposes the problem into two hierarchical problems: find a mapping $\phi^{''}$ from IDFG ($D^{''}$) to time extended sub-CGRA ($H^{''}$) and find a mapping $\phi^{'}$ from ISDG (D') to time extended VSA ($H^{'}$) in a way that they collectively solve the original problem.

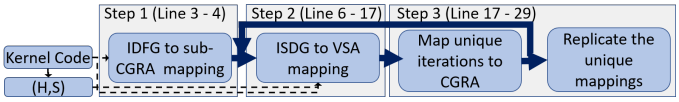


Fig. 4. Overview of the *HiMap* algorithm.

Algorithm 1: HiMap Algorithm

1 Kernel : K_l, CGRA : $G = (V_G, E_G)$, Space-time Mapping Matrix : $(H, S), (b_3, b_4, .., b_l)$
2 $D^{''} = (V^F_{D^{''}}, E^F_{D^{''}})$= getIDFG($K$)
3 IDFGMappings = MAP($D^{''}$) ▷ Get mappings of the IDFG on multiple sub-CGRAs
4 BoxDims = Sort(IDFGMappings) ▷ Sort mappings based on utiization
5 **foreach** $(s_1, s_2, t) \in$ *BoxDims* **do**
6 $b_1 = c/s_1, b_2 = c/s_2, II_B = t.(b_3.b_4..b_l)$
7 $D = (V_D, E_D)$ = GenerateDFG($K, b_1, b_2, b_3, .., b_l$)
8 $D^{'} = (\mathcal{C}, \mathcal{E})$ = ClusterDFG(D)
9 H_{II} = CreateMRRG(G, II_B)
10 **foreach** $C_i \in \mathcal{C}$ **do**
11 $C_i^P = \begin{bmatrix} H \\ S \end{bmatrix} \times C_i^I$ ▷ Systolic Mapping of each iteration
12 **foreach** $n_i \in C_i$ **do**
13 $n_i^P = (C_i^P \times (t\ s_1\ s_2)^T + n^{P^{'}}) \mod (II_B\ 0\ 0)^T$
14 **foreach** $(C_i, C_j) \in \mathcal{E}$ **do**
15 $(t^r, x^r, y^r) = C_j^P - C_i^P$
16 **if** !(($t^r == 1$) && ($|x^r| + |y^r| \leq 1$)) **then**
17 AddForwardingPath() ▷ Break multi-cycle multi-hop paths
18 **foreach** $C_i \in \mathcal{C}$ **do**
19 **if** $D_i^{''} \notin D_v^{''}$ **then**
20 $D_v^{''} = D_v^{''} \cup D_i^{''}, C_v = C_v \cup C_i$ ▷ Determine unique IDFGs
21 $D_v = (V_{D_v}, E_{D_v}) \leftarrow \emptyset$ ▷ Create minimal DFG
22 **foreach** $(n_i, n_j) \in E_D$ **do**
23 **if** $n_i \in C_v$ & $n_j \in C_v$ **then**
24 $E_{D_v} = E_{D_v} \cup (n_i, n_j), V_{D_v} = V_{D_v} \cup n_i \cup n_j$
25 **if** $n_i \notin C_v$ & $n_j \in C_v$ **then**
26 $E_{D_v} = E_{D_v} \cup (\gamma(n_i), n_j)$
27 success = ROUTE(D_v) ▷ Route minimal DFG on MRRG
28 **if** *success* **then**
29 REPLICATE(); ▷ Replicate the mapping
30 **Function** MAP($D^{''}$)
31 TopologicalSort($D^{''}$), $S \leftarrow (1, 2, .., |V^F_{D^{''}}|), T \leftarrow (0, 1, \ldots, M)$
32 **foreach** $(s_1, s_2, t_0) \in S^2 \times T$ **do**
33 $G^{''}$ = CGRAcluster(G, s_1, s_2), $t = |V^F_{D^{''}}|/(s_1.s_2) + t_0$
34 $H^{''}$ = TimeExtend($G^{''}, t$) ▷ Time extend the sub CGRA
35 **while** *!success OR !MaxIterations* **do**
36 **foreach** $v \in V^F_{D^{''}}$ **do**
37 **foreach** $h \in H^{''}$ **do**
38 **foreach** *p in Parents*(v) **do**
39 $L[h] = L[h] \cup LeastCostPath(\phi^{''}(p), h)$
40 $(h_l, path_l) = min(L)$;
41 assign($h_l, path_l$); ▷ Assign a placement and routing
42 **if** $C_l = oversubscribe(H)$ *is empty* **then**
43 success = 1; IDFGMapps = IDFGMapps $\cup \phi^{''}_{s_1, s_2, t}$;
44 **else**
45 IncreaseCosts(C_l) ; success = 0;
46 return IDFGMapps

V. HiMap Mapping Algorithm

Algorithm Overview: *HiMap* is an iterative algorithm that terminates when a valid mapping is found. The goal of the algorithm is to maximize the resource utilization of the target CGRA. Figure 4 shows the three main parts of the algorithm: 1) IDFG to sub-CGRA mapping, 2) ISDG to VSA mapping, and 3) identifying unique iterations, routing, and replication. Algorithm 1 presents the pseudo-code of *HiMap* algorithm. Step 1 (Line 3-4) maps operations in single iteration to sub-CGRA and generates multiple sub-CGRA mappings sorted based on their utilization. *HiMap* maps IDFG to sub-CGRA before mapping the ISDG to VSA because ISDG and VSA sizes depend on the sub-CGRA size. Step 2 (Line 6-17) and 3 (Line 17-29) iterate through the sub-CGRA mappings till it finds a valid CGRA mapping.

IDFG to sub-CGRA mapping: *HiMap* obtains a mapping ($\phi^{''}$) of IDFG computation nodes on a sub-CGRA in a way that sub-CGRA resource utilization is maximized. However, multiple mappings are generated for different sub-CGRA sizes with different utilization since step 3 could fail to find valid CGRA mapping for the highest utilized sub-CGRA mapping. Function `MAP()` performs topological sort of the operations in IDFG based on their dependencies. Different possible rectangular sizes (s_1, s_2) are considered for sub-CGRA ($s_1, s_2 \leq |V^F_{D^{''}}|$). The sub-CGRA is unrolled in the time dimension to create time extended sub-CGRA ($H^{''}$). The time depth of $H^{''}$ is initially set to its resource minimum value (Line 33).

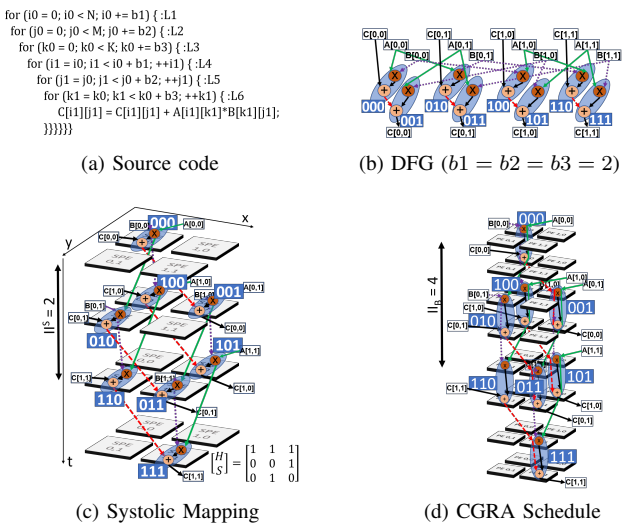
HiMap uses a heuristic based iterative algorithm for IDFG to sub-CGRA mapping (Line 33-45). The goal is to assign compute operations to PE ALU ($V^F_{H^{''}}$), load/store operations to memory ports and route the data signals between operations using ports ($V^P_{H^{''}}$). Each node $v \in V^F_{D^{''}}$ is assigned to node $h^F \in V^F_{H^{''}}$, that results in least accumulated cost when routing data from parent nodes of v. Routing is done utilizing ports in $h^P \in V^P_{H^{''}}$. *HiMap* uses Dijkstra's shortest path algorithm for establishing routes while allowing ports to be oversubscribed if necessary. All ports are initially assigned the same cost. At the end of each iteration, the costs of oversubscribed ports are increased for future iterations (inspired by SPR [13]). This encourages routing to avoid oversubscribed ports. We deem the mapping a success if none of the ports are oversubscribed. Mapping also terminates after user-defined maximum iterations. Then it starts with increased time depth or a different sub-CGRA size. Finally function `MAP()` returns all valid mappings for different sub-CGRA sizes (s_1, s_2) and time depths (t).

ISDG to VSA Mapping: For each valid sub-CGRA size (s_1, s_2), *HiMap* creates Virtual Systolic Array (VSA) on $c \times c$ CGRA (Line 6-8). Each systolic PE (SPE) in VSA contains a sub-CGRA. Therefore VSA size is equal to $(c/s_1, c/s_2)$. The block size (b_1, b_2) is chosen similar to the VSA size to fit ISDG on the VSA (Line 6-7). Then *HiMap* construct the DFG D and ISDG $D^{'}$ according to the determined block sizes [25]. ISDG is then placed on the VSA using a systolic mapping approach. In the quest of automating systolic array generation, authors of [17] propose an algorithm to obtain a direct mapping function between the iteration vector $\mathcal{C}^I_i$ and the space-time position $\mathcal{C}^P_i$ of the iteration $\mathcal{C}_i$ on a systolic array. The general form of the systolic mapping function ($\phi^{'}$) for transforming l-nested loop to a two-dimensional systolic array is

$$\phi^{'}: \mathcal{C}^P_i = \begin{bmatrix} H \\ S \end{bmatrix} \times \mathcal{C}^I_i \tag{1}$$

$$\mathcal{C}^I_i = \begin{bmatrix} i_1 \\ \vdots \\ i_l \end{bmatrix} \qquad \mathcal{C}^P_i = \begin{bmatrix} t \\ x \\ y \end{bmatrix} \qquad \begin{bmatrix} H \\ S \end{bmatrix} = \begin{bmatrix} h_1 & \dots & h_l \\ s_{11} & \dots & s_{1l} \\ s_{21} & \dots & s_{2l} \end{bmatrix}$$

(H, S) is the space-time mapping matrix. *HiMap* utilize the function $\phi^{'}$ to determine the space-time position of iterations on VSA (Line 11). (H, S) is input to *HiMap* algorithm, and it is pre-calculated using a heuristic search algorithm [17] that satisfies the necessary conditions to assure correct transformation. There could be multiple valid systolic mappings for a given kernel with different SPE utilizations ($|\mathcal{C}|/|V^F_{H^{'}}|$). *HiMap*



(a) Source code (b) DFG ($b1 = b2 = b3 = 2$)

(c) Systolic Mapping (d) CGRA Schedule

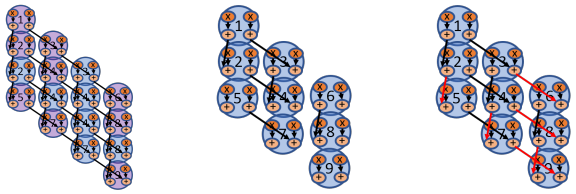
Fig. 5. *HiMap* CGRA schedule for GEMM kernel.

chooses the mappings that achieve the highest SPE utilization. Figure 5c shows the systolic mapping for GEMM kernel DFG (Figure 5b and 5a) on 2x2 CGRA. The sub-CGRA size is 1x1. Thus, VSA size is 2x2. Valid (H, S) for GEMM kernel is also shown in Figure 5c. According to function $\phi^{'}$, iteration $\mathcal{C}^I_i = (011)^T$ maps on to space-time position $\mathcal{C}^P_i = (211)^T$. Notice how dependent iterations are placed nearby in a way that dependent data flow through neighbor SPEs. II^S is the initiation interval of the systolic mapping on VSA. II^S depends on block sizes $(b_3, b_4, .., b_l)$ and is equal to $(b_3 \times b_4 \times .. \times b_l)$. In case of two-dimensional kernels $II^S = 1$. Therefore, II_S of three or higher dimensional kernels depends on the block sizes $(b_3, b_4, .., b_l)$, which is a user input to the *HiMap* algorithm.

The approach of replicating iteration mappings in step 3 demands all dependent iterations are placed in nearby SPEs, i.e., single-cycle single-hop placement. However, for some kernels with complex inter-iteration dependencies, it is impossible to find such systolic mapping (e.g., Floyd-Warshall kernel [18]). For such mappings, *HiMap* modify the IDFGs by adding pseudo input-output nodes to intermediate iterations that multi-cycle multi-hop paths need to go through (Line 17). It imitates multi-cycle multi-hop paths as a combination of single-cycle single-hop paths between iterations.

Identifying unique iterations, routing and replication: *HiMap* constructs the MRRG by extending the CGRA resource graph to II_B cycles. The value of II_B is equal to $II^S \times t$ where t is the depth of sub-CGRA mapping (Line 6). Then, it determines the absolute placement (n^P) of each DFG node $v \in V_D$ on the MRRG based on the relative placement of IDFG nodes within sub-CGRA ($n^{P^{'}}$) and the position of the iterations (C^P) on VSA (Line 13). Then *HiMap* determines the set of unique IDFGs distinguished by the dependency edges in and out of the iteration cluster. Input and output nodes ($V^I_{D^{''}}$) in IDFG represent the source and destination of the dependencies. Two IDFGs are the same if the relative placements (relative to corresponding destination and source node) of all input and output nodes ($V^I_{D^{''}}$) of the IDFGs are the same.

HiMap only determines the detailed routing of nodes be-



(a) (b) (c)

Fig. 6. (a) DFG (b) Intermediate DFG (c) Minimal DFG.

tween unique iterations because it can be replicated to create the final mapping. For that *HiMap* creates a minimal DFG (D_v) that captures the interaction between unique iterations by removing nodes in duplicated iterations. Figure 6c shows the minimal DFG corresponding to DFG in Figure 6a. First, all the nodes and edges in unique iterations are added to the initially empty minimal DFG. It forms an intermediate minimal DFG excluding edges between unique iterations (Figure 6b). Then, missing edges are added to the minimal DFG (Line 21-26). Afterwards, the function `ROUTE()` attempts to do detailed routing of the minimal DFG on the CGRA (Line 27). The approach is similar to the way routing is done in the `MAP()` function. A successful outcome of function `ROUTE()` contains the port configurations of all interconnects and RFs corresponding to unique iterations. Then both ALU and routing configurations are replicated according to the ISDG placement to create the mapping (ϕ) of D on CGRA. According to the generated mapping, each PE has a repeating instruction stream with a length equal to II_B. However, *HiMap* keeps unique instructions in the configuration memory of each CGRA PE to avoid configuration memory bloat. PE program counters generate the instruction stream according to the mapping schedule.

VI. Experimental Evaluation

We now evaluate *HiMap* for multi-dimensional kernels on different CGRA sizes. Each PE in the CGRA contains a RF with four registers (two r/w ports), an ALU, a configuration memory that holds up to 32 instructions, and a crossbar switch. To eliminate memory access bottlenecks in some kernels, each PE also contains a data memory that holds up to 64 data elements. We implement the CGRA architecture in Verilog HDL and synthesize it on a 40 nm process using *Synopsys* toolchain to estimate the power consumption. The maximum clock frequency is 510 MHz. We implement *HiMap* in C++ to accept the C source code of the target kernel as input and generate IDFG by analyzing the LLVM bitcode of the kernel. *HiMap* constructs the unrolled DFG and ISDG using the method proposed in [25]. We perform functional validation of the resultant mappings through cycle-accurate software simulation of the executions on CGRA architecture.

Benchmarks: Table I shows categorization of compute-intensive loop kernels from MachSuite [26], MiBench [27] and Polybench [11] benchmark suites based on the loop dimensionality (Dim) and the existence of inter-iteration dependency. For kernels without inter-iteration dependencies, we can potentially execute all the iterations in parallel. For kernels with single dimension ($Dim = 1$) iteration dependencies, the only option is

TABLE I
Loop kernel categorization.

No inter-iteration dependency	With inter-iteration dependency			
Dim = 1/ 2/ 3	Dim = 1	Dim = 2	Dim = 3	Dim = 4
MachSuite: aes mix col, add row, bd softmax, relu, add bias, take diff, get delta matrix weight, knn md, update weights, viterbi comp prob. MiBench: jpeg fdct islow. PolyBench: huffman encode, correlation, covariance, trisolv, fd2d.	MachSuite: aes expand key, spmv, viterbi MiBench: basic math usqrt, susan Polybench: stencil jacobi1d, cholesky, symm, gesummv, durbin, dynprog, gramschmidt, reg detect.	Polybench: adi, atax, bicg, mvt, fd2d, gemmver, jacobi 2d. Machsuite: mvt, nw, stencil 2d. Conv2D	Polybench: gemm, syrk, mm, floyd-warshall. Machsuite: fft. Conv3D	Polybench: ttm, doitgen.

TABLE II
Characteristics for multi-dimensional kernels.

Benchmark	Loop levels (Dim)	Description	Max unique iterations
ADI	2	Alternating Direction Implicit solve	3
ATAX	2	Matrix Transpose and Vector Multiplication	9
BICG	2	BiCG Sub Kernel of BiCGStab Linear Solver	9
MVT	2	Matrix Vector Product and Transpose	9
GEMM	3	General Matrix Multiply	27
SYRK	3	Symmetric rank-k operation	27
Floyd–Warshall	3	Shortest path and transitive closure	34
TTM	4	Tucker Decomposition	45

to place the iterations along the time dimension. Consequently, for these two types of kernels, we cannot get any benefit through virtual systolic mapping of the iterations, and we can apply existing software pipelining techniques to extract instruction-level parallelism [9]. Therefore, for evaluation, we choose eight multi-dimensional ($Dim > 1$) kernels with inter-iteration dependencies, as listed in Table II.

Baselines: We evaluate *HiMap* against two state-of-the-art CGRA compilation algorithms, *HyCUBE* [6] and *CGRA-ME* [28]. *HyCUBE* compiler uses a heuristic-based mapping algorithm, which is an augmented version of *SPR* (Schedule, Place, and Route) [13]. *CGRA-ME* uses simulated annealing and ILP-based mapping approach [16]. It suffers from a much longer compilation time than *HyCUBE*. Consequently, it fails to generate valid mappings for most kernels on CGRA size bigger than 8x8. Therefore, we report the best utilization results obtained from the two frameworks and call it "Best of HyCUBE & CGRA-ME (*BHC*)" mapping result.

Utilization Comparison: Figure 7 shows the CGRA resource utilization (U) comparison (quality of mapping) between *BHC* and *HiMap* mapping approaches on different CGRA sizes. The utilization of *HiMap* is higher or equal (average 2.8x better) compared to *BHC* for all benchmarks in all CGRA sizes. *HiMap* achieves 100% utilization, i.e., performance envelope of CGRA for five kernels. Resource utilization for kernel ADI is 83%, while for kernels BiCG, and FW it is 66%. Kernels ADI, BiCG, and FW consist of five, four, and two compute operations in one iteration, respectively. For these kernels final valid mapping is obtained with sub-CGRA mappings with $(s_1, s_2, t) = (2, 1, 3)$, $(2, 1, 3)$, and $(1, 1, 3)$, respectively. The utilization of these sub-CGRA mappings is 5/6 (83%), 4/6 (66%), and 2/3 (66%). The utilization of the final valid mapping is similar to the utilization of corresponding sub-CGRA mapping because *HiMap* replicates the sub-CGRA mappings to obtain the final valid mapping. *HiMap* fails to find valid routing with 100% utilized sub-CGRA mappings due to high routing resource congestions in these three kernels. Lower utilized sub-CGRA mappings make more routing resources available (in the time dimension) and resolve routing resource congestions. In future works, we will explore the possibility of merging sub-CGRA mappings in the time dimension to further improve the

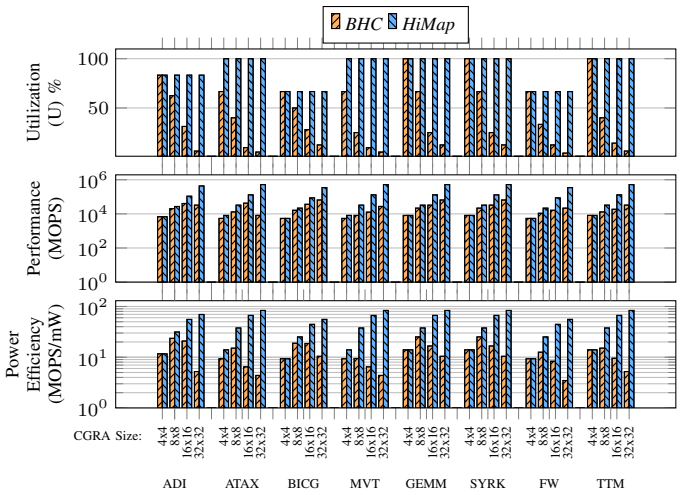


Fig. 7. Comparison of *BHC* and *HiMap* for different CGRA sizes along X-axis.

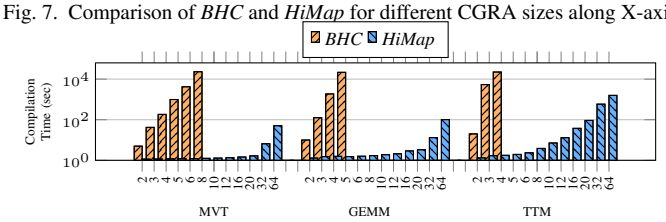


Fig. 8. Compilation time comparison of BHC and *HiMap* algorithms for different block sizes ($c = b_1 = b_2 = b_3 = b_4$) along X-axis.

utilization of hierarchical mapping.

When we increase the CGRA size, *HiMap* increases the size of the DFG by increasing the block size, while *BHC* maps the small DFG keeping the block size small. *BHC* fails to find a solution when the number of DFG nodes is higher than 400 due to scalability issues. As the number of PEs in bigger CGRAs are higher than the number of nodes in the smaller DFGs, the resource utilization remains low for *BHC* even when the mapping for each block is optimal.

Performance and Power Efficiency Comparison: Figure 7 shows the performance and power efficiency of *BHC* and *HiMap* across different CGRA sizes. The performance of both *HiMap* and *BHC* increases with the CGRA size. However, on average, the performance of *HiMap* is 17.3x higher than the performance of *BHC*. The power-efficiency of *HiMap* increases with the CGRA size and achieve 5x better power efficiency compared to *BHC*. The power-efficiency of the *BHC* approach decreases with an increase in the CGRA size as the power increase in bigger CGRAs is more than the corresponding performance benefit of under-utilized *BHC* mappings.

Compilation Time: Figure 8 shows the compilation time comparison between *HiMap* and *BHC*. We keep the block size (b=b_1=b_2=b_3=b_4) equal to the CGRA size ($c \times c$) for both approaches ($b = c$). *BHC* fails to find a valid mapping beyond the block size of 8, 5, and 4 for MVT, GEMM, and TTM, respectively (timeout after 3 days). The maximum compilation time for *HiMap* is less than 15 minutes, even for a block size of 64 on 64x64 CGRA. *BHC* takes nearly one day to find a mapping with a block size of 4 for TTM and block size of 5, 8 for GEMM, MVT. The compilation time for *HiMap* is not much affected by the block size as the number of unique iterations remains constant as block size increases. The maximum number of unique iterations identified by *HiMap* is shown in Table II.

VII. Conclusion

We introduced, *HiMap*, a fast and scalable approach for mapping multi-dimensional kernels on to bigger CGRAs. *HiMap* exploits the regular nature of the inter-iteration dependencies through hierarchical mapping and achieves 17.3x performance and 5x performance-per-watt improvement with minimal compilation time compared to the state-of-the-art.

VIII. Acknowledgement

This work was supported by Singapore Ministry of Education Academic Research Fund TI 251RES1905 and Huawei International Pte. Ltd.

References

[1] C. Nicol, "A Coarse Grain Reconfigurable Array (CGRA) for statically scheduled data flow computing," Wave computing white paper, 2017.
[2] C. Kim et al., "ULP-SRP: Ultra low-power samsung reconfigurable processor for biomedical applications," TRETS'14.
[3] K. Fleming et al., U.S. Patent No. 10,416,999. Washington, DC: U.S. Patent and trademark office, 2019.
[4] A. Nayak et al., "A framework for adding low-overhead, fine-grained power domains to CGRAs," in DATE'20.
[5] S. Dave et al., "URECA: Unified register file for CGRAs." DATE'18.
[6] M. Karunaratne et al., "HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect," DAC'17.
[7] Wang, Bo, et al., "HyCUBE: A 0.9 V 26.4 MOPS/mW, 290 pJ/cycle, power efficient accelerator for IoT applications." A-SSCC'19.
[8] T. Kojima et al., "Body bias optimization for variable pipelined CGRA." FPL'17.
[9] B. Rau,"Iterative Modulo Scheduling: An algorithm for software pipelining loops," International symposium on microarchitecture, 1994.
[10] B. Mei et al., "DRESC: A retargetable compiler for CGRAs," FPT'02.
[11] L. Pouchet, and G. Scott. "Polybench: The polyhedral benchmark suite," Polybench/C 4.1.
[12] M. Karunaratne et al.,"4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs,", ICCAD'19.
[13] S. Friedman et al., "SPR: an architecture-adaptive CGRA mapping tool," FPGA'09.
[14] M. Hamzeh et al.,"REGIMap: Register-aware application mapping on CGRAs," DAC'13.
[15] L. Chen, T. Mitra, "Graph minor approach for application mapping on CGRAs," ACM TRETS'14.
[16] S. Chin, J. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," DAC'18.
[17] P. Lee, Z. Kedem. "Mapping nested loop algorithms into multidimensional systolic arrays," IEEE TPDS'90.
[18] S. Kung, S. Lo, "Optimal systolic design for the transitive closure and the shortest path problems," IEEE Transactions On Computers, 1987.
[19] H. Kung, "Why systolic architectures?," IEEE Computer, 1982.
[20] J. Cong, J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," ICCAD'18.
[21] S. Dave S et al., "DMazerunner: Executing perfectly nested loops on dataflow accelerators," ACM TECS'19.
[22] D. Wijerathne et al., "CASCADE: High throughput data streaming via decoupled access-execute CGRA," ACM TECS'19.
[23] Jouppi, N. P. et al., "In-datacenter performance analysis of a tensor processing unit." ISCA'17.
[24] Kågström, B. et al., "GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark," TOMS'98.
[25] Y. Yu et al., "Loop parallelization using the 3D iteration space visualizer," Journal of visual languages & computing, 2001.
[26] B. Reagen et al., "Machsuite: Benchmarks for accelerator design and customized architectures," in IISWC'14.
[27] M. Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite," WWC-4'01.
[28] S. Chin et al., "CGRA-ME: A unified framework for CGRA modelling and exploration," ASAP'17.