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

Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, Lothar Thiele

***Abstract*—Coarse-Grained Reconfigurable Array (CGRA) has emerged as a promising hardware accelerator due to the excellent balance between reconfigurability, performance, and energy efficiency. The performance of a CGRA strongly depends on the existence of a high-quality compiler to map the application kernels on the architecture. Unfortunately, the state-of-the-art compiler technology falls short in generating high-performance 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 regular computational kernels prevalent in existing and emerging application domains. The key strategy behind *HiMap*'s efficiency and scalability is to exploit the regularity in the computation by employing a virtual systolic array as an intermediate abstraction layer in a hierarchical mapping. *HiMap* first maps the loop iterations of the kernel onto a virtual systolic array and then distills out the unique patterns in the mapping. These unique patterns are subsequently mapped onto sub-spaces of the physical CGRA. They are arranged together according to the systolic array mapping to create a complete mapping of the kernel. 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. Even though the Application Specific Integrated Circuit (ASIC) accelerators offer the best performance and power efficiency, the rigid implementation makes it a less attractive accelerator choice. Reconfigurable accelerators are getting popular because they offer reasonable performance and power 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.

This work was supported by the Singapore Ministry of Education Academic Research Fund TI 251RES1905 and Huawei International Pte. Ltd. D. Wijerathne, Z. Li, and T. Mitra are with the Department of Computer Science, School of Computing, National University of Singapore, SG. E-mail: ((dmd, zhaoying, tulika)@comp.nus.edu.sg). A.Pathania is with the University of Amsterdam. E-mail: (a.pathania@uva.nl). A. Pathania was with National University of Singapore when this research was conducted. Lothar Thiele is with the Computer Engineering and Networks Laboratory, ETH Zürich, 8092 Zürich, Switzerland. E-mail: (thiele@tik.ee.ethz.ch). (Corresponding author. Tulika Mitra)

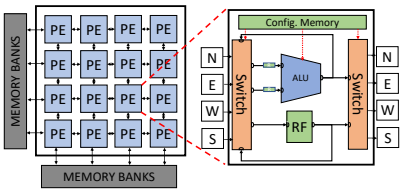


Fig. 1: An abstract block diagram for a 4x4 CGRA.

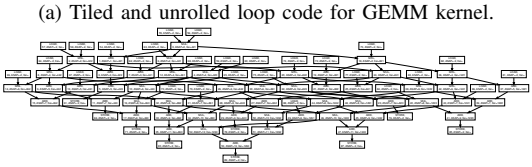
CGRAs offer word-level reconfigurability and therefore are more power-efficient than the FPGAs [1]. Another major difference between the two is that the FPGAs primarily rely on spatial mapping, while most CGRAs support spatio-temporal mapping due to the per-cycle reconfigurability. There have been many recent works on CGRAs in the industry [1]–[3] and academia [4]–[11].

Figure 1 shows a CGRA consisting of an array of Processing Elements (PE) connected in a 2D mesh network. Each PE typically comprises an Arithmetic Logic Unit (ALU), crossbar switches, 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, register files, or outside memory banks in each cycle. It either keeps the results in the RF or sends them to the neighboring PEs or both. The configuration memory instructions control 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).

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 [12]. Software pipelining allows independent operations (operations without data dependency) in different iterations to run parallelly. 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 [13]. 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

```
for (i0 = 0; i0 < N; i0 += b)  :L1
  for (j0 = 0; j0 < M; j0 += b)  :L2
    for (k0 = 0; k0 < K; k0 += b)  :L3
        C[i0+0][j0+0] = C[i0+0][j0+0] + A[i0+0][k0+0]*B[k0+0][j0+0];
        .....
        C[i0+0][j0+0] = C[i0+0][j0+0] + A[i0+0][k0+b-1]*B[k0+b-1][j0+0];
        .....
        .....
        C[i0+b-1][j0+b-1] = C[i0+b-1][j0+b-1] + A[i0+b-1][k0+b-1]*B[k0+b-1][j0+b-1];
```

(a) Tiled and unrolled loop code for GEMM kernel.



(b) DFG of GEMM kernel for block size $b = 3$.

Fig. 2: Example of a multidimensional kernel.

using small loop kernels on CGRAs with a relatively small number of PEs [4], [6], [9]–[11].

Multi-dimensional loop kernels, i.e., loops with multiple nested 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 (ILP). Hence, these kernels are well suited to be accelerated on CGRAs with a large number of PEs. 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 formed from the multi-dimensional iterators.

Figure 2a shows the tiled and fully unrolled General Matrix Multiply (GEMM) loop kernel as an example of a multi-dimensional kernel. Loop tiling breaks down the kernel's iteration space into smaller-sized blocks [14], [15]. The block size (tile size) b determines the size of the fully unrolled data flow graph of the loop body. Kernel DFGs with smaller block sizes have few parallel instructions compared to bigger block sizes. For example, GEMM kernel DFG with a block size of 3 (Figure 2b) consists of 54 compute instructions while DFG with a block size of 8 consists of 1024 compute instructions. Smaller DFG kernels underutilize the CGRA resources, especially in bigger CGRAs. State-of-the-art CGRA mapping algorithms can handle modest-sized kernels but cannot find a valid mapping when the DFG size grows (beyond 500 nodes) because of the large number of nodes and complex dependencies between them [4], [11], [16], [17]. Therefore, the state-of-the-art compilers can compile only small block sizes, which results in an order of magnitude lower performance than the maximum achievable performance.

We propose a hierarchical mapping approach, *HiMap*, that can achieve the ideal performance envelope for multi-dimensional kernels. *HiMap* solves the mapping problem hierarchically by doing abstraction at both kernel and architecture levels. Kernel DFG is abstracted into a graph of iterations, where each iteration constitutes multiple DFG nodes. This graph of iterations is called Iteration Space Dependency Graph (ISDG). The target CGRA is also abstracted into a virtual array where each virtual processing element constitutes multiple CGRA PEs. This virtual array is called Virtual Systolic Array (VSA). Using those two abstractions, *HiMap* solves the mapping problem in two levels; mapping the ISDG onto the VSA and mapping operations in each iteration to CGRA PEs. *HiMap* maps ISDG onto VSA using a space-time transformation incorporated from systolic design methodology. After ISDG to VSA mapping, operations in each iteration are mapped onto actual CGRA PEs using a heuristic-based mapping approach.

We have first explored and presented a preliminary version of the hierarchical mapping idea in [18]. The main focus of [18] is generating scalable mapping inside the PE array with little attention to the impact of external communication (data feeding and unloading) for the mapping. Systolic data flow, which is the basis of *HiMap* mapping, requires a specific communication mechanism for feeding and unloading data. This communication mechanism entails the data layout in the local memory and connectivity between local memory and PE array. The data layout refers to the placement of array variables and the element order within the data memory. The data layout depends on the corresponding systolic mapping. Therefore, different computation mappings require data layouts with different reordering of original array variables. The connectivity between local memory and PE array largely influences the feasibility of the resultant mapping. The limited memory connectivity can result in low performance or even make it unfeasible to support some kernels.

In this paper, we explore the impact of the external communication of the target architecture on kernel performance. We have enhanced the hierarchical mapping algorithm to consider local memory resource constraints and automatically generate the corresponding data layout. We evaluate the performance under different memory resource configurations. We have detailed the modifications required on local memory organization of existing CGRA architectures to enable systolic data flow.

II. Background and related work

HiMap utilize systolic design methodology in the hierarchical mapping approach. It uses systolic mapping as an intermediate overlay to guide the CGRA mapping. This section provides background details on systolic design methodology and CGRA mapping.

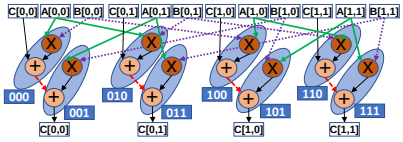
A. Systolic Arrays

Systolic arrays are hardware structures that are used for the fast and efficient execution of regular algorithms. An array of processing elements are connected in algorithm-specific ways to perform computations on different input data repeatedly. Researchers have proposed many systolic architectures to accelerate various multi-dimensional kernels [19]–[22]. In systolic arrays, data flows regularly and rhythmically, passing through processing elements before it returns to memory. Systolic arrays achieve full utilization of the compute and memory resources when executing the kernels, hence offering high power efficiency.

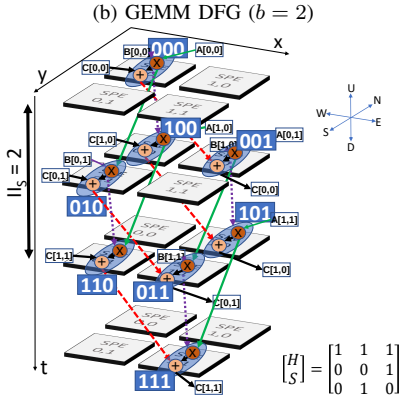
Initial systolic architectures were designed manually in special-purpose systems for specific tasks [21], [22]. Many spatial dataflow accelerators also use systolic arrays [23]–[25]. They specifically target kernels in the deep learning domain.

```
for (i0 = 0; i0 < N; i0 += 2) :L1
 for (j0 = 0; j0 < M; j0 += 2) :L2
  for (k0 = 0; k0 < K; k0 += 2) :L3
      C[i0+0][j0+0] = C[i0+0][j0+0] + A[i0+0][k0+0]*B[k0+0][j0+0]
      C[i0+0][j0+0] = C[i0+0][j0+0] + A[i0+0][k0+1]*B[k0+1][j0+0]
      C[i0+0][j0+1] = C[i0+0][j0+1] + A[i0+0][k0+0]*B[k0+0][j0+1]
      .....
      C[i0+1][j0+1] = C[i0+1][j0+1] + A[i0+1][k0+1]*B[k0+1][j0+1]
```

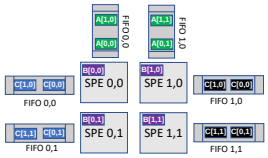
(a) GEMM Kernel Source Code ($b = 2$)



(b) GEMM DFG ($b = 2$)



(c) Systolic Mapping



(d) Data Layout

Fig. 3: Systolic Data Flow for GEMM kernel.

These dataflow accelerators also consist of an array of PEs connected in a mesh-like network, which is similar to PE organization in conventional CGRAs. Unlike conventional CGRAs, the network in these accelerators is not fully reconfigurable. Moreover, PEs only support a limited number of operations, such as Multiply-And-Accumulate (MAC).

B. Systolic Mapping

Figure 3c shows the systolic data flow for GEMM kernel DFG (Figure 3a and 3b) on a 2x2 systolic array. Iterations in the kernel are mapped and executed on Systolic Processing Elements (SPE). One main feature in systolic mapping is that dependent iterations are mapped to neighboring SPEs in either space or time dimension. This feature allows for multiple data reuse as the data flows through the array. Systolic mapping allows reused data to flow within the array using internal registers. Thus it is more efficient than repeatedly accessing the same data from the memories where data is stored initially. In Fig. 3c, input-reused data flow through neighbor SPEs in the y-dimension (green solid edges) and t-dimension (purple dot edges) while output-reused data flow in the x-direction (red dash edges).

In the quest of automating systolic array generation, authors of [26] propose a systolic mapping algorithm to assign each point in the iteration space of a multi-dimensional kernel to a space-time position (time slot and coordinates of the processing element) on a systolic array. Each point in iteration space corresponds to an iteration C_i, indexed by respective iteration variables. Iteration vector $I(C_i)$ is the vector of iteration variables. Systolic mapping algorithm in [26] obtains a direct mapping function between the iteration vector $I(C_i)$ of a multi-dimensional kernel and the space-time position $P(C_i)$ of the iteration C_i on a systolic array. Space-time position $P(C_i)$ is denoted by a tuple (t, x, y) where t is the time slot (time position) and x, y are the coordinates of the systolic processing element (space position). The general form of the systolic mapping function ($\phi^{'}$) for transforming l-nested loop to a two-dimensional systolic array is

$$\phi^{'} : P(C_i) = \begin{bmatrix} H \\ S \end{bmatrix} \times I(C_i) \tag{1}$$

$$I(C_i) = \begin{bmatrix} i_1 \\ : \\ i_l \end{bmatrix} \qquad P(C_i) = \begin{bmatrix} t \\ x \\ y \end{bmatrix} \qquad \begin{bmatrix} H \\ S \end{bmatrix} = \begin{bmatrix} h_1 & \dots & h_l \\ s_{11} & \dots & s_{1l} \\ s_{21} & \dots & s_{2l} \end{bmatrix}$$

(H, S) is the space-time mapping matrix. H refers to time hyperplane where $H \times I(C_i)$ gives the time position of the iteration C_i. S refers to space hyperplane where $S \times I(C_i)$ gives space position of the iteration C_i on the systolic array. (H, S) is calculated using a heuristic search algorithm [26] that satisfies the necessary conditions to assure correct transformation. The conditions are:

1) H has to preserve data dependency relations between iterations. If C_i has a data dependency from $C_j \Rightarrow H \times (I(C_i) - I(C_j)) > 0$.
2) Two iterations must not map to the same SPE at the same time. $H \times I(C_i) = H \times I(C_j)$ and $S \times I(C_i) = S \times I(C_j)$ cannot be both true at the same time.
3) Condition for the number of registers between two SPEs.
4) Condition for avoiding collisions in data links.

Valid (H, S) for the GEMM kernel is shown in Figure 3c. According to function $\phi^{'}$, iteration $I(C_i) = (011)^T$ maps onto space-time position $P(C_i) = (211)^T$. The dependent iterations are placed nearby in a way that dependent data flow through neighboring SPEs.

Recent works on FPGA compilation exploit systolic-based execution for compute-bound kernels [27]. *PolySA* [28] uses systolic array mapping techniques to implement CNN and MM kernels on FPGA. Authors of [29] implement CNN on FPGA using systolic architecture.

C. External Communication of Systolic Arrays

In systolic arrays, each iteration of the kernel is executed on an SPE. If an iteration requires outside data, I/O memory should supply the data to the corresponding SPE. The way data should be placed and ordered in I/O memory depends on the iteration to systolic array mapping (space-time mapping matrix) and each iteration's input/output requirements. The authors of [19] studied systolic memory communication related to Warp systolic array. External communication of systolic array, i.e.,

input/output data transfer between local memory and PE array, occurs in a mapping-dependent deterministic fashion.

Figure 3c shows the I/O dataflow of the GEMM kernel. The data elements of matrix A should arrive from boundary PEs in the north (N) direction, while initial data of matrix C come from the boundary PEs in the west (W) direction. The values of matrix B are preloaded through the local memory of each PE and stay stationary. The output values of matrix C leave from the boundary PEs in the east (E) direction. Boundary PEs should be connected with auto sequencing first-in-first-out (FIFO) memories to maintain data streams entering and leaving the PE array. The data layout of the above mapping is shown in Figure 3d. Elements in matrix A, C, and B should be placed in top FIFOs, left FIFOs, and local PE memory. The output elements of matrix C will be stored in the right FIFOs.

Systolic mapping entails constraints on the memory resource type and the connectivity between local memory and PE array. Consequently, the above mapping is infeasible on a CGRA architecture with simple dual-port memories without FIFO functionality. Furthermore, the above mapping is also infeasible on a CGRA where only one side of boundary PEs are connected to local memory [5].

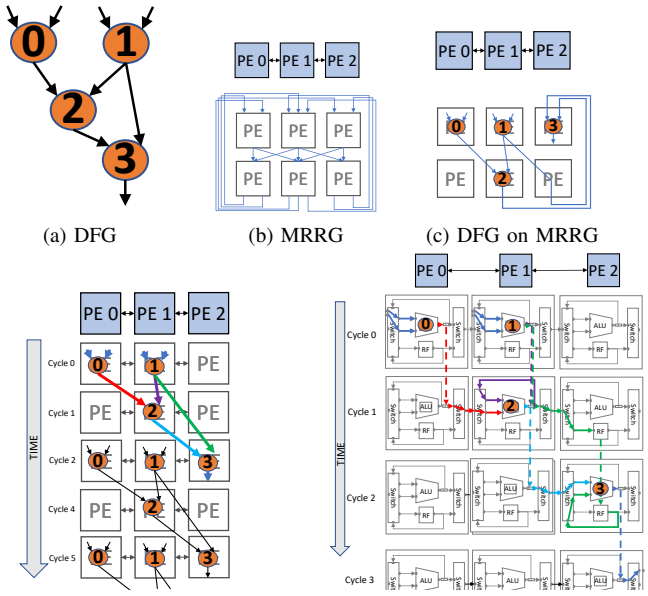
D. CGRA Mapping

Given an application with a compute-intensive loop kernel and a CGRA architecture, the CGRA compiler needs to generate a kernel schedule such that application throughput is maximized. The loop kernel is represented as a DFG where the nodes represent the operations and edges represent the data dependencies among operations. CGRA mapping process consists of:

1) Placement - Assigning DFG operations to functional units in space and time so that dependency constraints are met
2) Routing - mapping data signals between producer and consumer DFG operations using wires and registers.

The CGRA compilers exploit ILP in compute-intensive loop kernels through a software pipelining technique called modulo scheduling [12]. Software pipelining extracts iteration-level parallelism since ILP in a single basic block is not adequate. It allows independent operations in successive iterations to run simultaneously by overlapping operations in consecutive iterations. The objective of the modulo scheduling is to generate a repeating schedule such that multiple iterations are initiated in a pipelined fashion. A new loop iteration can initiate execution in every Initiation Interval (II). Thus, the modulo scheduling aims to minimize the II value. Figure 4a shows a simple DFG of a loop kernel. Initially, CGRA resources and DFG characteristics determine the lower bound of II, i.e., minimal II (MII). MII is the smallest possible value in which a modulo schedule exists. It is computed as the maximum value of both resource-constrained II and recurrence-constrained II [12].

The compiler represents the CGRA architecture using Modulo Routing Resource Graph (MRRG) [30]. MRRG models the basic CGRA components (ALUs, RFs, and switches) and their connections in a time-space view. In other words, MRRG is a time-extended resource graph of the CGRA. The time axis



(a) DFG (b) MRRG (c) DFG on MRRG

(d) Execution of DFG (e) CGRA data path configurations

Fig. 4: Illustration of CGRA Mapping Process.

is restricted to II cycles since the goal is to find the repeating schedule. This schedule is also called a modulo schedule. Figure 4b shows an MRRG with II = 2 of a linear CGRA with three PEs. The compiler mapping algorithm aims to find a valid mapping (modulo schedule) of the DFG on the MRRG. Valid mapping is defined as placement and routing with no resource usage conflict. The compiler starts mapping DFG on an MRRG with MII, and II is increased until the valid mapping is obtained. Figure 4c shows a valid modulo schedule for II=2. The execution of the CGRA schedule is shown in Figure 4d.

Routing is the most time-consuming part of the mapping process. The CGRA mapper needs to establish valid routes between producer and consumer operations. The data signals are routed between functional units through switches and RFs. Therefore, the mapper needs to decide the switch and RF port configurations at each cycle. Figure 4e shows how data routes are established in a CGRA architecture with two switches and one RF. The figure shows how the data dependency between operation 1 and operation 3 is routed (green edges).

The existing CGRA mapping algorithms can be categorized into three main categories based on their approach. They are either heuristic-based [4], [6]–[8], [13], [17], [31], graph-based [9], [10], or ILP-based [11]. These approaches work well for accelerating loop kernels with irregular data dependencies. However, the authors of these compilers only evaluate them for loop kernels with relatively small DFGs and small CGRA sizes (4x4, 8x8) [4], [6], [9]–[11].

III. Target CGRA Architecture

Figure 5 shows the target CGRA architecture along with the system-level setup. The CGRA accelerates the compute-intensive kernel in an application while the host processor runs the rest of the application. The host processor also manages the data transfer between off-chip memory and CGRA memories. DMA engine and memory controller facilitates the transfer

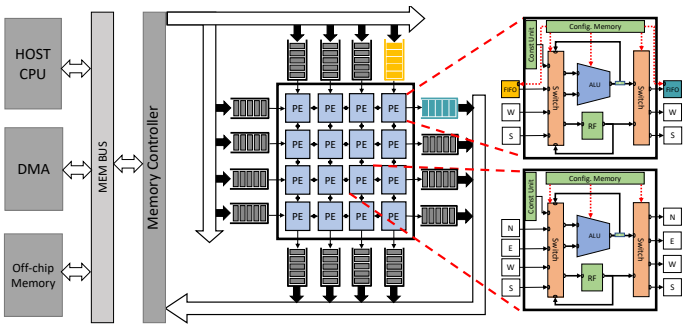


Fig. 5: Target CGRA architecture.

of data between the off-chip main memory and on-chip memories.

As explained in the previous section, systolic data flow entails a few constraints on local memory connectivity and memory type. Accordingly, we designed the local memory organization of the CGRA to support systolic data flow. The target CGRA accesses input/output (I/O) data from two types of on-chip memories, boundary FIFO memories, and constant units. In the following, the term I/O memory resources refer to those two types of on-chip memories. Systolic mapping of kernels requires simple streaming data access in FIFO order. Therefore, boundary PEs are connected with FIFO memories. The top row and leftmost column of boundary PEs are connected to load FIFOs that hold input data. FIFOs connected to the bottom row and rightmost column of the boundary PEs store the result data (store FIFOs). Therefore, switches in boundary PEs have connections to load and store FIFOs. The configuration memory produces the read/write signals to FIFOs to sequence the data flow. Each PE consists of a constant unit that holds the pre-loaded input data in a single register or register file with multiple registers. Constant units improve the memory capability of the array, and it is useful to achieve higher utilization when the kernels are memory-bound. If the total number of operations is larger than the total number of input and output elements in a kernel, then the kernel is compute-bound; otherwise, it is memory-bound.

CGRA consists of an array of processing elements connected in a 2D mesh network. Each PE consists of an ALU, RF, crossbar switches, and configuration memory. The execution model of the CGRA is similar to what is explained in the introduction.

IV. Motivating Example

HiMap is a hierarchical mapping approach that uses abstractions at both kernel and architecture levels. Kernel DFG is abstracted into Iteration Space Dependency Graph (ISDG), and the CGRA is abstracted into Virtual Systolic Array (VSA). It first maps ISDG onto the VSA. Then, it identifies the unique iterations in terms of computation and data routing. It does detailed CGRA mapping only for the unique iterations, which it then replicates to create the final CGRA mapping.

We first present example mapping schedules of a two-dimensional kernel on linear (one-dimensional) CGRA arrays to show the effectiveness of our approach. BiCG is a two-dimensional kernel of BiCGStab Linear Solver [32]. Figure 6a shows the source code of the tiled BiCG loop kernel where

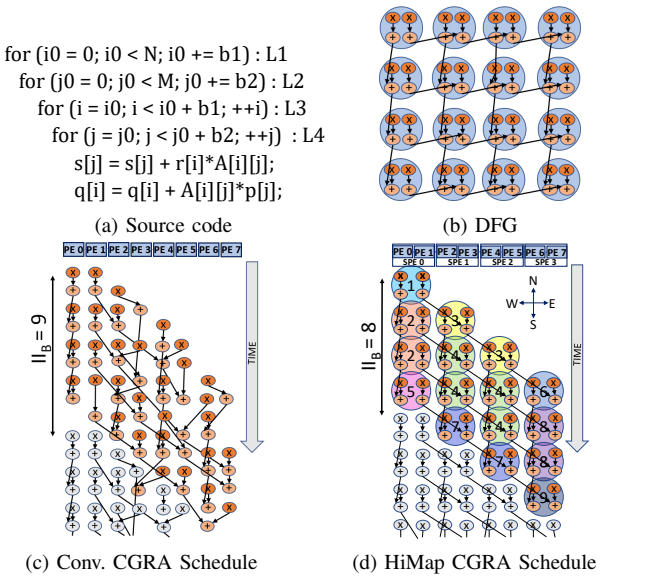


Fig. 6: Illustration of *HiMap* CGRA schedule of BiCG kernel.

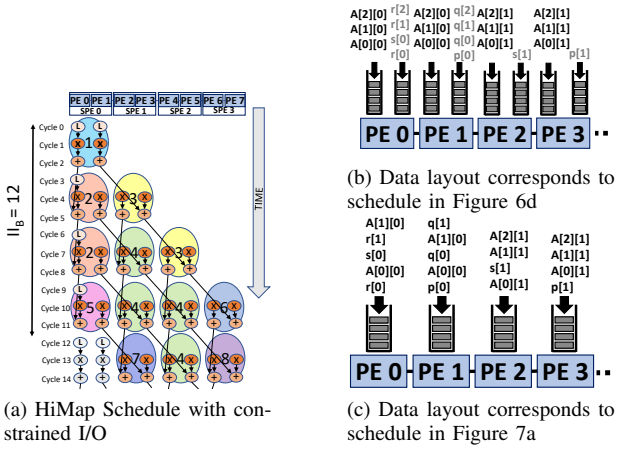


Fig. 7: Impact of memory resources on *HiMap* CGRA schedule of BiCG kernel.

$(b1, b2)$ 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 6b 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. Thus, ellipses and connections between them form the ISDG. Figures 6c and 6dshow conventional and *HiMap* mapping schedules on the target 8x1 linear CGRA with each PE is connected with two FIFOs for data loading. The colored nodes belong to the current computation block, 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.

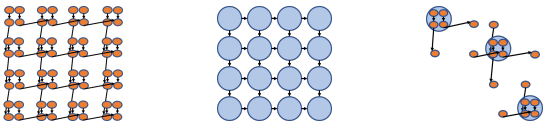
The conventional mapping schedule is irregular as the mapper does not see the regularity of the dependencies among operation clusters [4], [6]. Therefore, the conventional approach takes a long time to find an optimal placement and routing. *HiMap* exploits the regularity of iteration level dependencies to generate a regular schedule. It creates 4x1 Virtual Systolic Array (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 the *HiMap* schedule. The number inside each ellipse, along with unique colors, 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 northwest iteration and produce data to the south and southeast iterations. Legend NSWE is used to depict north, south, west, and east directions. *HiMap* identifies and performs detailed mapping for these unique iterations. Then the unique iteration mappings are replicated to generate the final CGRA mapping schedule. 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.

Figure 7a demonstrates the impact of memory resources on kernel performance. It shows *HiMap* mapping schedule of BiCG kernel on a CGRA with limited I/O memory as shown in 7c (each PE connected with one FIFO instead of two in the previous case). According to the mapping schedule, PE 0 does the computation of $s[j] + r[i] * A[i][j]$ which is related to one iteration of BiCG kernel. When there are two FIFOs connected to one PE, elements $r[i]$ and $A[i][j]$ can be placed in different FIFOs allowing single-cycle access for both data elements. In that case, only two cycles are needed to complete multiplication and addition operations. However, in the schedule on CGRA with limited I/O memory, PE 0 needs two cycles to access $r[i]$ and $A[i][j]$ as both data elements are placed in a single FIFO. Thus, it increases the number of cycles to 3 to complete one iteration. Consequently, II_B of the schedule is increased to 12, decreasing the kernel performance. Figures 7b and 7c show the input data layout corresponding to the HiMap schedules in 6dand 7a, respectively. Some array variables (A, s, p) are distributed among different FIFOs, while some FIFOs contain data from multiple array variables. The data layout depends on the corresponding CGRA schedule. The host processor should move the data to FIFOs according to this data layout for correct execution. Therefore, the mapping algorithm should generate the data layout at compile time to achieve correct system-level execution.

V. Problem Formulation

This section presents the definitions and problem formulation for the hierarchical application mapping approach on CGRA.

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 dependencies between operations of the tiled and unrolled multi-dimensional kernel. $(b_1, b_2, .., b_l)$ represents block size where l is the tiled loop level. Figure 8a shows the fully unrolled DFG of BiCG kernel ($b1 = b2 = 4$).



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

Fig. 8: Illustration of graph definitions for BiCG kernel.

Iteration Space Dependency Graph (ISDG): ISDG is a directed acyclic graph $D^{'} = (\mathcal{C}, \mathcal{E})$ where the 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) [33]. Two clusters are connected if and only if there is a node in one that is connected to a node in the other. Figure 8b shows the ISDG of the BiCG kernel. Each iteration in $\mathcal{C}$ is indexed with i. $I(C_i)$ represents the iteration vector of C_i. ISDG is a multi-dimensional block where block size $(b_1, b_2, .., b_l)$ determines the dimensions.

Intra-Iteration Data-Flow Graph (IDFG): We define an IDFG to 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 that 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 the BiCG kernel are shown in Figure 8c.

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 $H_{II_B} = (V_H, E_H)$ is a resource graph of the CGRA that is time extended to II_B cycles [13]. V_H consists of two types of nodes: ALUs (V^F_H) in each PE and ports (V^P_H) in interconnects and RFs [13]. 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. $U = |V_D|/|V^F_H|$.

Data Domain: Let $R = R_0, R_1, .., R_{m-1}$ be the memory references in the DFG D. The data domain M is defined as a set of all memory elements accessed by all the memory references through outer loop iterations.

Data Placement: Let d_r be a data element accessed by memory reference $r \in R$. Data placement of a memory reference r is described as a function $P(r) = (x_r, y_r, t_r)$ $\forall r \in R$, where (x_r, y_r) is the index of the I/O memory that

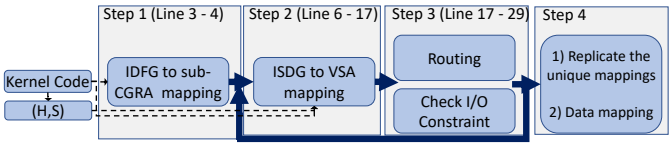


Fig. 9: Overview of the *HiMap* algorithm.

d_r is mapped to and t_r is the cycle that d_r is accessed. I/O memory index is the same as the index of the PE that I/O memory is directly connected to.

I/O Memory Constraint: The validation of the data placement is interpreted as the number of parallel data elements accessed by a PE should be less than or equal to the available number of I/O memory resources connected to that PE.

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 exists;

1) a mapping $\phi = (\phi_V, \phi_E)$ from $D = (V_D, E_D)$ to H_{II} which maximizes the utilization U
2) a valid data placement $P(r)$ for all $r \in R$.

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.

VI. 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 9 shows the four main parts of the algorithm:

1) Intra-Iteration Data-Flow Graph to sub-CGRA mapping,
2) Iteration Space Dependency Graph to Virtual Systolic Array mapping,
3) Inter-iteration routing and I/O memory constraint check,
4) Unique iteration replication and data placement.

Algorithm 1 presents the pseudo-code of *HiMap* algorithm while Algorithm 2 shows the corresponding subroutines. Step 1 (Line 4-5) determines node placement within sub-CGRA, and step 2 (Line 7-15) determines the placement of the iterations on VSA. Using these two placements, *HiMap* can determine the CGRA schedule with all the nodes placed on the target CGRA. Step 3 (Line 15-26) checks the validity of that CGRA schedule in terms of routing and I/O memory resource constraints. Step 3 identifies unique iterations, and inter-iteration routing is done only for those unique iterations. Step 1 generates multiple sub-CGRA mappings sorted based on their utilization. Steps 2 and 3 iterate through the sub-CGRA mappings till it finds a valid CGRA mapping of unique iterations. Step 4 replicates unique iteration mappings to generate the final valid mapping. Step 4 also generates the corresponding data layout. We use a running example, BiCG kernel mapping on linear (8,1) CGRA (Figure 6), to explain the *HiMap* algorithm.

Algorithm 1: HiMap Algorithm

1 Inputs: Kernel : K, CGRA : $G = (V_G, E_G)$, Space-time Mapping Matrix : (H, S), Block Sizes $(b_3, b_4, .., b_l)$
2 Outputs: CGRA Mapping : ϕ, Data Placement : γ
3 $D^{''} = (V^F_{D''}, E^F_{D''})$= getIDFG($K$)
4 IDFGMappings = IDFG_MAP($D^{''}$) ▷ Get mappings of the IDFG on multiple sub-CGRAs
5 BoxDims = Sort(IDFGMappings) ▷ Sort mappings based on utiization
6 **foreach** $(s_1, s_2, t) \in$ *BoxDims* **do**
7 $b_1 = c/s_1, b_2 = c/s_2$, $II_B = t.(b_3.b_4..b_l)$
8 $D = (V_D, E_D)$ = GenerateDFG($K, b_1, b_2, b_3, .., b_l$)
9 $D^{'} = (\mathcal{C}, \mathcal{E})$ = ClusterDFG(D)
10 H_{II} = CreateMRRG(G, II_B)
11 **foreach** $C_i \in \mathcal{C}$ **do**
12 $P(C_i) = \begin{bmatrix} H \\ S \end{bmatrix} \times I(C_i)$ ▷ Systolic Mapping of each iteration
13 **foreach** $n_i \in C_i$ **do**
14 $P(n_i) = (P(C_i) \times (t\ s_1\ s_2)^T + P^{'}(n))$ mod $(II_B\ -\ -)^T$
15 .
16 **foreach** $C_i \in \mathcal{C}$ **do**
17 **if** $D^{''}_i \notin D^{''}_v$ **then**
18 $D^{''}_v = D^{''}_v \cup D^{''}_i$,
19 $C_v = C_v \cup C_i$ ▷ Determine unique IDFGs
20 $D_v = (V_{D_v}, E_{D_v}) \leftarrow \emptyset$ ▷ Create minimal DFG
21 **foreach** $(n_i, n_j) \in E_D$ **do**
22 **if** $n_i \in C_v$ & $n_j \in C_v$ **then**
23 $E_{D_v} = E_{D_v} \cup (n_i, n_j), V_{D_v} = V_{D_v} \cup n_i \cup n_j$
24 **if** $n_i \notin C_v$ & $n_j \in C_v$ **then**
25 $E_{D_v} = E_{D_v} \cup (\gamma(n_i), n_j)$
26 success = ROUTE(D_v) & IO_CONSTRAINT_CHECK(D_v)
27 **if** *success* **then**
28 ϕ = REPLICATE(); ▷ Replicate the mapping
29 γ = DATA_PLACEMENT();

A. IDFG to sub-CGRA mapping

This step aims to determine the possible placement of nodes in one iteration on time extended sub-CGRA. Only intra-iteration edges, i.e., edges between nodes within an iteration, are routed when determining the correct placement. This step ignores inter-iteration edge routing. Inter-iteration edges are routed in step 3. *HiMap* obtains a mapping ($\phi^{''}$) of IDFG computation nodes on a sub-CGRA in a way that sub-CGRA resource utilization is maximized. However, step 3 could fail to find valid mapping due to routing or I/O memory resource constraints for the highest utilized sub-CGRA mapping. Therefore, multiple mappings are generated for different sub-CGRA sizes with different utilization. Figure 10 shows two possible IDFG to sub-CGRA mappings of BiCG kernel.

Function `getIDFG()` generates the IDFG $D^{''}$ by analyzing LLVM IR of the loop kernel [34]. Function `IDFG_MAP()` performs the IDFG to sub-CGRA mapping, which includes operation placement and intra-iteration dependency routing. First, it performs topological sort of the operations in IDFG based on their dependencies. Different possible rectangular sizes (s_1, s_2) and time depths are considered for sub-CGRA size ($s_1, s_2 \leq |V^F_{D''}|, t \leq |V^F_{D''}|/(s_1.s_2) + t^{Max}$). The sub-CGRA is unrolled in the time dimension to create time extended sub-CGRA ($H^{''}$). The time depth (t) of $H^{''}$ is initially set to its resource minimum value $|V^F_{D''}|/(s_1.s_2)$ (Line 4).

HiMap uses a heuristic-based iterative algorithm for IDFG to sub-CGRA mapping (Line 4-16). The goal is to assign operations to PE ALU ($V^F_{H''}$) and route the data signals between operations using the switch and register file 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

Algorithm 2: Main Subroutines of HiMap Algorithm

1 **Function** IDFG_MAP($D^{''}$)
2 TopologicalSort($D^{''}$), $S \leftarrow (1, 2, .., |V^F_{D''}|)$, $T \leftarrow (0, 1, ..., t^{Max})$
3 **foreach** $(s_1, s_2, t_0) \in S^2 \times T$ **do**
4 $G^{''}$ = CGRAcluster(G, s_1, s_2), $t = |V^F_{D''}|/(s_1.s_2) + t_0$
5 $H^{''}$ = TimeExtend($G^{''}, t$) ▷ Time extend the sub CGRA
6 **while** *!success OR !MaxIterations* **do**
7 **foreach** $v \in V^F_{D''}$ **do**
8 **foreach** $h \in H^{''}$ **do**
9 **foreach** p *in Parents(v)* **do**
10 $L[h] = L[h] \cup LeastCostPath(\phi^{''}(p), h)$
11 $(h_l, path_l) = min(L)$;
12 assign($h_l, path_l$); ▷ Assign a placement and routing
13 **if** $C_l = oversubscribe(H)$ *is empty* **then**
14 success = 1; IDFGMapps = IDFGMapps $\cup \phi^{''}_{s_1,s_2,t}$;
15 **else**
16 IncreaseCosts(C_l) ; success = 0;
17 return IDFGMapps
18 **Function** ROUTE(D_v)
19 H = CreateReducedMRRG(G)
20 **while** *!success OR !MaxIterations* **do**
21 **foreach** $v \in V^F_D$ **do**
22 **foreach** $h \in H$ **do**
23 **foreach** p *in Parents(v)* **do**
24 $L[h] = L[h] \cup LeastCostPath(\phi(p), h)$
25 $(h_l, path_l) = min(L)$;
26 assign($h_l, path_l$); ▷ Assign a placement and routing
27 **if** $C_l = oversubscribe(H)$ *is empty* **then**
28 success = 1;
29 **else**
30 IncreaseCosts(C_l) ; success = 0;
31 return success;
32 **Function** IO_CONSTRAINT_CHECK(D_v)
33 **foreach** $R_j \in R_0, R_1, .., R_{m-1}$ **do**
34 $(x_d, y_d, t_d) = P(R_j) = P(C_i) \times (t\ s_1\ s_2)^T + P^{'}(n_i)$; $parallel\_access[x_d][y_d][t_d] + +$;
35 **foreach** $x, y \in c \times c$ **do**
36 **foreach** $t \in (0, .., lat)$ **do**
37 **if** $maxParAcc < parallel\_access[x][y][t]$ **then**
38 $maxParAcc = parallel\_access[x][y][t]$;
39 **if** $g(x, y) < maxParAcc$ **then**
40 return fail;
41 return success;

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 [6]). The higher cost in oversubscribed ports encourages the mapper to avoid oversubscribed ports when possible. We deem the mapping a success if none of the ports is oversubscribed. Mapping also terminates after user-defined maximum iterations. Then it starts with increased time depth or a different sub-CGRA size. Time depth is increased until user defined t^{Max} value. Finally, function IDFG_MAP() returns all valid mappings candidates for different sub-CGRA sizes (s_1, s_2) and time depths (t). For example, GEMM kernel has only two IDFG nodes ($|V^F_{D''}| = 2$). Therefore only two rectangular sizes are considered for GEMM sub-CGRA $(s_1, s_2) = (1, 1), (1, 2)$. The maximum time depth (t) allowed is configurable through parameter t^{Max} ($t \leq |V^F_{D''}|/(s_1.s_2) + t^{Max}$). For example, if t^{Max} value is 3, the sub-CGRA mapping candidates would be (s1,s2,t) = (1,1,2), (1,1,3), (1,1,4), (2,1,2), (2,1,3), (2,1,4).

(a) $(s_1, s_2, t = 1, 1, 4)$. (b) $(s_1, s_2, t = 2, 1, 2)$.

Fig. 10: *IDFG to Sub-CGRA mappings of BiCG IDFG.*

B. ISDG to VSA Mapping

For each valid sub-CGRA mapping (s_1, s_2, t), *HiMap* creates Virtual Systolic Array (VSA) on $c \times c$ CGRA (Line 7-9). Each systolic PE (SPE) in VSA contains a sub-CGRA. Therefore, VSA size is equal to $(c/s_1, c/s_2)$. For example, for the BiCG mapping with the sub-CGRA size of (2,1), VSA size is (4,1) on an (8,1) linear CGRA.

In the systolic mapping algorithm, multi-dimensional kernel ISDG is mapped onto two-dimensional VSA using a transformation function $\phi^{'}$ (Equation 1). For a correct transformation, in the case of two-dimensional VSA, the first two dimensions (b_1, b_2) in the kernel ISDG should be similar to the two dimensions in VSA [26]. Therefore, block size (b_1, b_2) is chosen similar to the VSA size to fit ISDG on the VSA (Line 7-8). In the case of one-dimensional VSA, as in our example, only the first dimension b_1 should be similar to the VSA dimension. Figure 11 shows the linear (4,1) VSA on an (8,1) linear CGRA. $b1$ needs to be equal to 4 to fit the ISDG onto VSA. We use array padding to avoid partial tiles when the block sizes do not evenly divide the trip counts of the target application kernel [15]. Then, *HiMap* constructs the DFG D and ISDG $D^{'}$ according to the determined block sizes [33]. ISDG is then placed on the VSA using a systolic mapping approach. *HiMap* utilize the function $\phi^{'}$ (Equation 1) to determine the space-time position of the iterations on the VSA (Line 12).

Figure 11 shows the ISDG to VSA mapping for BiCG kernel on a linear (4,1) VSA. Figure 3c shows the ISDG to VSA mapping for the GEMM kernel on a two-dimensional VSA. Valid (H, S) is also shown in Figure 11. In Figure 11, iteration $I(C_i) = (01)^T$ of BiCG kernel maps on to space-time position $P(C_i) = (11)^T$ according to function $\phi^{'}$. Notice that both mappings can start a new computation to utilize all the SPEs in every time unit. A new computation is executed after 4-time units in BiCG mapping and 2-time units in GEMM mapping. Therefore, those mappings have 100% SPE utilization. We call the duration between the initiation of two computations as the initiation interval of the ISDG mapping on VSA (II_S). As mentioned before, ISDG is transformed onto two-dimensional VSA where (b_1, b_2) was determined according to the spatial dimensions of VSA. The remaining block sizes $(b_3, b_4, .., b_l)$ of the ISDG determines the time dimension of the transformation. Therefore, II_S depends on block sizes $(b_3, b_4, .., b_l)$ and is equal to $(b_3 \times b_4 \times .. \times b_l)$. The block sizes can affect the final performance, especially in systems with hardware-controlled memories such as caches [14], [35]. [35] shows that optimal tile sizes can achieve 10% average performance improvement over cubic tiling (equal tile size in all dimensions). However, with software-controlled memories in CGRA, the effect of tile size is limited compared to

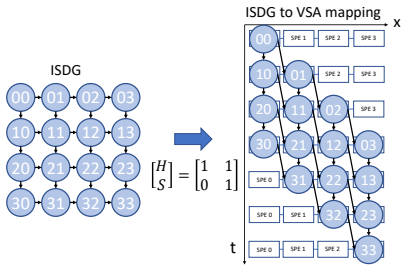


Fig. 11: ISDG to VSA mapping for BiCG kernel (b1=b2=4).

the hardware caches. In experiments, we choose the trivial configuration for the remaining block sizes $b_2 = b_3 = b_4$.

There could be multiple valid systolic mappings for a given kernel with different SPE utilizations ($|\mathcal{C}|/|V^F_{H'}|$). Also, there could be valid systolic mappings with the same SPE utilization but with different dataflows. For example, the GEMM kernel dataflow shown in Figure 3c is widely known as weight-stationary dataflow (considering matrix B as weight matrix). The GEMM kernel can have output-stationary dataflow with different (H, S). *HiMap* can support both dataflows given the corresponding (H, S) as input. In experiments, *HiMap* use the space-time mappings that achieve the highest SPE utilization.

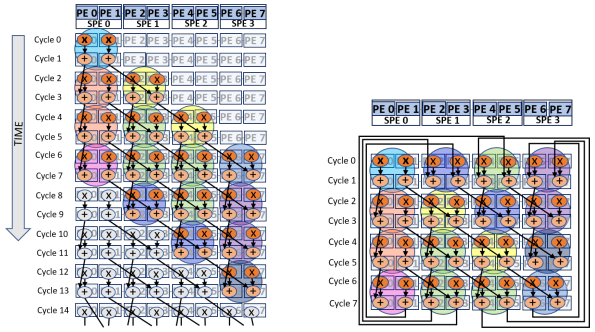
C. Routing and I/O Memory Constraint Check

Step 1 determines node placement within sub-CGRA, and step 2 determines the placement of the ISDG on VSA. Using these two placements, *HiMap* can determine the schedule with all nodes placed on the target CGRA. For example, Figure 12a shows the possible schedule of BiCG kernel corresponding to the sub-CGRA mapping shown in Figure 10b and the ISDG to VSA mapping shown in Figure 11. The space-time placement of each DFG node $n_i \in V_D$ on the CGRA is given by the function $P(n_i)$ where $P^{'}(n_i)$ is the relative placement of the node n_i within sub-CGRA, $P(C_i)$ is the position of the corresponding iteration on VSA, t is the time depth and $(s1, s2)$ is the sub-CGRA size.

$$P(n_i) = P(C_i) \times (t\ s_1\ s_2)^T + P^{'}(n_i) \tag{2}$$

This schedule could be invalid due to two types of resource constraints in the target CGRA: 1) Failure to route the inter-iteration dependency edges due to routing resource constraints. 2) Failure to access the input/output data due to I/O memory resource constraints. *HiMap* checks these constraints parallelly because both constraints should be satisfied for a valid mapping.

Inter-iteration Dependency Routing: For the dependency routing, function `createMRRG()` 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 II_S is the initiation interval of ISDG to VSA mapping, and t is the depth of sub-CGRA mapping (Line 7). The modulo schedule, i.e, the placement $(P^{''}(n_i))$ of each DFG node $n_i \in V_D$ on the MRRG is given by $P^{''}(n_i) = P(n_i) \mod (II_B\ -\ -)^T$ (Line 14). $\mod (II_B\ -\ -)^T$ performs the modulo operation only on the time dimension. Figure 12b shows the BiCG kernel modulo schedule, i.e., DFG placement on MRRG of the linear CGRA.



(a) DFG execution schedule (b) DFG on MRRG

Fig. 12: Inter-iteration dependency routing for BiCG kernel.

HiMap only determines the detailed routing of edges between the nodes in unique iterations. Unique iterations capture all the different routing possibilities between all iterations in the final mapping. Thus, *HiMap* can replicate unique iteration mappings to create the final mapping. *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 needs to segregate the interaction between unique iterations to determine the detailed routing. For that *HiMap* creates a minimal DFG (D_v), which captures the interaction between unique iterations. Minimal DFG is created by removing all the nodes in duplicated iterations. Figure 13 shows the minimal DFG corresponding to BiCG DFG. To create the minimal DFG, first, all the nodes in unique iterations are added to the node-set of initially empty minimal DFG. If both source and destination nodes of an edge belong to unique iterations, those edges are added to the edge set of minimal DFG. It forms an intermediate minimal DFG as shown in Figure 13. However, it omits edges between some unique iterations. Edges are omitted when a destination node of an edge does not belong to the unique iteration set. For example, in Figure 13, the edge between nodes belonging to yellow and blue color iterations is omitted in intermediate minimal DFG. To add omitted edges, for each edge in original DFG $(u, child(u))$, when a destination node do not belong to an unique iteration ($child(u) \notin D_v$), it recursively visits child destination nodes until it finds a node which belongs to unique iterations ($v \in D_v$). Then the omitted edge (u, v) is added to the edge set of minimal DFG (Line 20-25).

Minimal DFG is then mapped onto a reduced MRRG. The reduced MRRG dimensions depend on the number of unique iterations in each ISDG dimension and the sub-CGRA size. For example, BiCG minimal DFG is mapped onto a reduced MRRG as shown in Figure 14a. Since each dimension has 3 unique iterations and sub-CGRA size is (2,2), the size of the reduced MRRG is ($x = 2 \times 3, II_B = 2 \times 3$). The function `ROUTE()` attempts to do detailed routing of the minimal DFG on the reduced MRRG (Line 17). It also employs cost-based Dijkstra's shortest path algorithm for routing. The approach is similar to how routing is done in the `IDFG_MAP()` function,

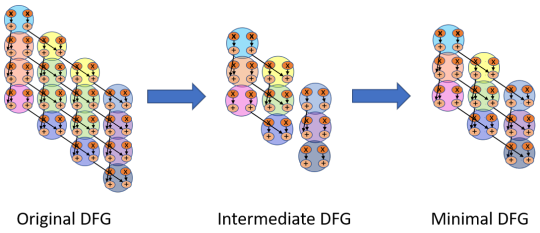
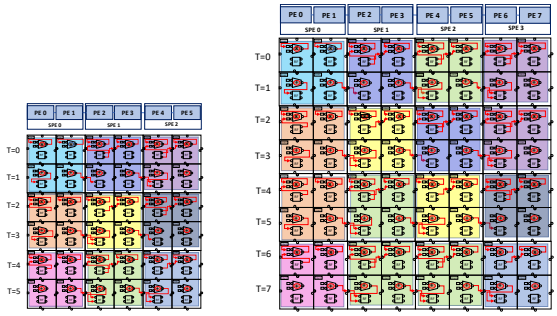


Fig. 13: Minimal DFG of BiCG kernel.



(a) Mapping of minimal DFG on the reduced MRRG .

(b) Replicated MRRG mapping.

Fig. 14: Data path configurations of BiCG kernel.

as explained in step 1. If the routing fails due to routing resource congestions after user-defined maximum iterations, it repeats steps 2 and 3 with a new sub-CGRA mapping obtained in step 1. Lower utilized sub-CGRA mappings make more routing resources available (in the time dimension) and resolve routing resource congestions. A successful outcome of function `ROUTE()` contains the port configurations of all interconnects and RFs. Figure 14a shows the detailed routing of the minimal DFG of the BiCG kernel on the reduced MRRG.

I/O Memory Resource Constraint: The target CGRA access input/output data from two types of I/O memory resources, FIFOs connected to boundary PEs and constant units. *HiMap* checks whether the I/O memory requirement of the schedule matches the available I/O memories of the target CGRA. This constraint check is done on the schedule generated from step 2. If the available I/O memory resources are not sufficient, *HiMap* rejects the schedule and repeats step 2 with the next sub-CGRA size.

Each memory reference $r \in R_0, R_1, .., R_{m-1}$ in the unrolled kernel is directly connected with a DFG node in the kernel. The space-time placement of memory reference r is similar to the space-time placement of the directly connected DFG node. Therefore, the placement of memory reference r $(P(r))$ is equal to the directly connected DFG node placement. If n_i is the directly connected DFG node to memory reference r,

$$P(r) = (x_r, y_r, t_r) = P(C_i) \times (t \ s_1 \ s_2)^T + P^{'}(n_i) \quad (3)$$

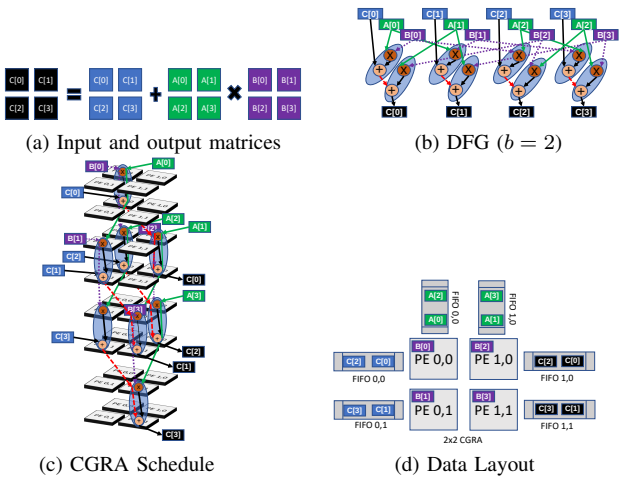
Let d_r is a data element that memory reference r is accessed. (x_r, y_r) is the index of the I/O memory that d_r is mapped to and t_r is the cycle that d_r is accessed. The validation condition of the data placement function $P(r)$ is checked by the function `IO_CONSTRAINT_CHECK()` (Line 31). It checks whether the number of I/O memories connected to each PE is less than or equal to the maximum number of parallel data accesses in that PE. The maximum number of parallel data accesses is equal to the number of data elements mapped to the same IO memory index (x_r, y_r) in the same cycle. Function $g(x, y)$ return the number of I/O memories connected to each PE indexed by coordinates (x, y). If the schedule is invalid due to limited I/O memory resources, *HiMap* repeats steps 2 and 3 with a new lower utilized sub-CGRA mapping obtained in step 1. Those sub-CGRA mappings are time extended, so parallel accesses are serialized to reuse the limited I/O resources in the time dimension.

D. Unique Mapping Replication and Data Placement

Replicating Unique Iteration Mappings: Once the schedule is validated w.r.t both routing and I/O memory resources, the resource configurations of all PEs can be obtained by replicating unique iteration mappings. The function `ROUTE()` obtains both ALU and routing configurations for unique iterations. These unique iteration mappings are arranged together to create the final valid mapping (ϕ) of D on CGRA. *HiMap* replicate each unique iteration mapping one or more times in the final mapping. The replication position, i.e., the space-time position where each unique iteration mapping is replicated, can be determined by the ISDG/DFG placement on MRRG $(P^{''}(n_i))$. Figure 14b shows the replicated mapping of the DFG of BiCG kernel on the MRRG. Notice how ALU and port configurations that correspond to each unique iteration (shown in unique colors) is replicated in the final mapping. According to the generated mapping, each PE has a repeating instruction stream with a length equal to II_B.

Data Placement: Notice that *HiMap* schedule has no address generation instructions. Data is either streamed in and out of the CGRA through boundary FIFOs or loaded through the constant units. CGRA PEs access the stream of data in I/O memories by enabling read/write signals. The host processor manages the data movement between the CGRA I/O memories and main memory using DMA. Hence the host processor requires a mapping between each memory reference in the unrolled kernel and the corresponding CGRA I/O memory. Therefore, *HiMap* needs to generate a data layout that consists of the I/O memory ID of each memory reference and the order of the referenced data elements arranged inside the I/O memory. The data layout is derived using the data placement function $P(r) = (x_r, y_r, t_r)$. (x_r, y_r) gives the index of I/O memory where each memory reference r is mapped. To determine the data elements order inside the I/O memory, function `DATA_PLACEMENT()` arrange the memory references in ascending order according to the value t_r.

Figure 15 shows the *HiMap* data mapping of GEMM kernel $(b = 2)$. Figure 15a shows the input and output variables corresponding to the GEMM kernel. Figure 15b shows the DFG of the GEMM kernel while Figure 15c shows the corresponding CGRA schedule on 2x2 CGRA. Figure 15d shows the data layout generated from *HiMap*. Arrays A and C are mapped onto boundary FIFOs, and B is mapped onto the constant



(a) Input and output matrices (b) DFG ($b = 2$)

(c) CGRA Schedule (d) Data Layout

Fig. 15: *HiMap* Data Mapping for GEMM.

unit in each PE. Figure shows the data elements accessed by the memory references at outer loops iteration vector $(i0, j0, k0) = (0, 0, 0)$. The data elements C[1] and C[3] are mapped onto FIFO (0,1) because the directly connected DFG nodes are mapped onto PE (0,1). C[1] is placed before C[3] in the FIFO (0,1) because the directly connected DFG nodes are mapped at t=3 and t=5 in the CGRA schedule.

VII. Experimental Evaluation

We now evaluate *HiMap* for multi-dimensional kernels on different CGRA sizes. Each PE in the CGRA contains RF with four registers, an ALU, a constant unit, a configuration memory that holds up to 32 instructions, and a crossbar switch. Left and top boundary PEs are connected to load FIFOs, whereas right and bottom boundary PEs are connected to store FIFOs, as shown in Figure 5. 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 [33]. We perform functional validation of the resultant mappings through cycle-accurate software simulation of the executions on CGRA architecture.

Benchmarks: We choose nine multi-dimensional ($Dim > 1$) compute-intensive loop kernels from MachSuite [36], MiBench [37] and Polybench [32] benchmark suites, as listed in Table I. 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 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 [12]. Therefore, for evaluation, we choose multi-dimensional ($Dim > 1$) kernels with inter-iteration dependencies.

Baselines: We evaluate *HiMap* against two state-of-the-art CGRA compilation algorithms, *HyCUBE* [4] and *CGRA-ME* [16]. *HyCUBE* compiler uses a heuristic-based mapping algorithm, which is an augmented version of *SPR* (Schedule, Place, and Route) [6]. *CGRA-ME* uses simulated annealing and ILP-based mapping approach [11]. 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 from the two frameworks and call it the "Best of HyCUBE & CGRA-ME (*BHC*)" mapping result.

TABLE I: 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
CONV2D	2	2-D Convolution with 2x2 filter	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

Utilization Comparison: Figure 16 shows the CGRA compute 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% compute utilization, i.e., performance envelope of CGRA for five kernels. Compute resource utilization for kernel ADI is 83%, while for kernel CONV2D it is 70%. For kernels BiCG, and FW the utilzation is 66%. Kernels ADI, CONV2D, BiCG, and FW consist of five, seven, four, and two compute operations in one iteration. For these kernels final valid mapping is obtained with sub-CGRA mappings with $(s_1, s_2, t) = (2, 1, 3)$, $(2, 1, 5)$, $(2, 1, 3)$, and $(1, 1, 3)$, respectively. The utilization of these sub-CGRA mappings is 5/6 (83%), 7/10 (70%), 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. Register file (RF) utilization (average number of RF registers in use per cycle) of valid *HiMap* mappings are 5%, 50%, 59%, 65%, 78%, 32%, 32%, 58%, 70% for ADI, ATAX, BICG, MVT, CONV2D, GEMM, SYRK, FW, TTM respectively. Kernels with high utilized final mappings and relatively complex data dependencies result in higher RF utilization. In future works, we will explore the possibility of merging sub-CGRA mappings in the time dimension to further improve the utilization of hierarchical mapping.

Performance and Power Efficiency Comparison: Figure 16 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 17.4x higher than the performance of *BHC*. The power-efficiency of *HiMap* increases with the CGRA size and achieves 5x better power efficiency than *BHC*. The power efficiency of

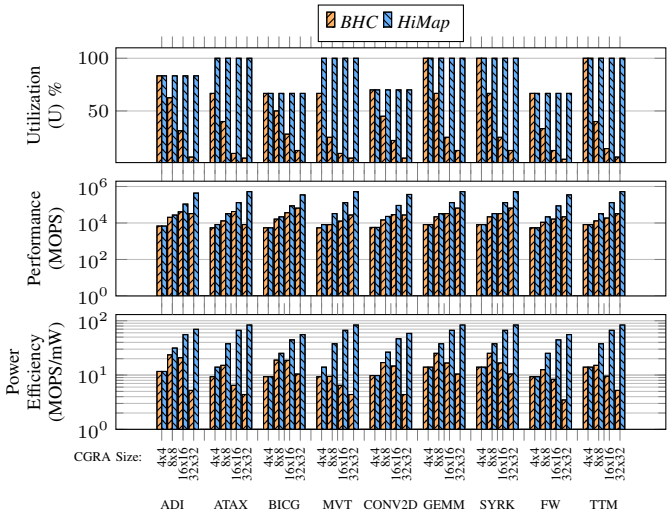


Fig. 16: Comparison of *BHC* and *HiMap* for different CGRA sizes along X-axis.

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.

Impact of I/O Memory Resources: As *HiMap* is I/O memory aware mapping technique, we explore the impact of I/O memory resources on kernel utilization and performance. We compare kernel performance on CGRAs with two different I/O memory configurations illustrative of different memory capabilities. I/O-1 is similar to the previous configuration where each boundary PE is connected to load/store FIFOs, and each PE contains a constant unit as shown in Figure 5. Therefore, in I/O-1, each boundary PE has two I/O memory resources (FIFO and constant unit), and every other PE has one I/O memory resource (constant unit). In I/O-2, boundary PEs do not have constant units. Therefore, I/O-2 has one I/O memory resource per PE. Thus, I/O-2 has low I/O memory resources compared to I/O-1.

Figure 17 shows the PE utilization and performance of *HiMap* schedule on the two different I/O memory configurations. The results show that I/O memory resources are critical to PE utilization and performance. The I/O-resources limit the utilization of kernels ADI, ATAX, MVT, GEMM, SYRK, and TTM. For the I/O-2 configuration, valid mapping is obtained with lower utilized sub-CGRA mappings than the I/O-1 configuration. Those sub-CGRA mappings are time extended, so parallel accesses are serialized to reuse the I/O memory resources in the time dimension. For I/O-2 configuration, valid sub-CGRA sizes are $(s1, s2, t) = (2, 1, 4)$ for ADI and $(s1, s2, t) = (1, 1, 3)$ for the other kernels. Since BICG, CONV2D, and FW kernel utilization is already limited by routing resources, there is no impact from the limitation of I/O memory resources to those two kernels.

Compilation Time: Figure 18 provides the compilation time for the main steps of the *HiMap* algorithm on 8x8 CGRAs with different I/O constraints (I/O-1 and I/O-2). IDFG to sub-CGRA (step 1) generates multiple valid sub-CGRA mappings. Therefore, the mapping time depends on the complexity of the

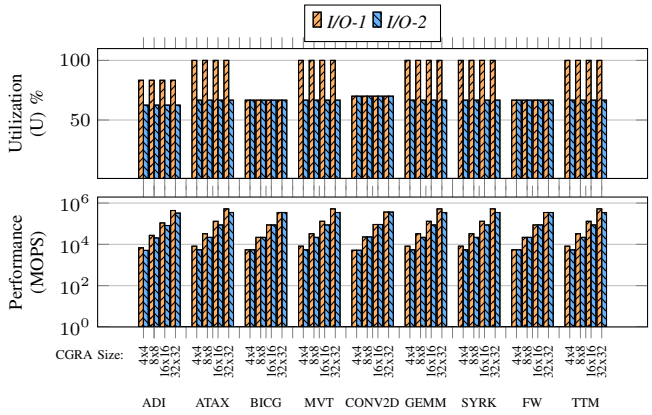


Fig. 17: Utilization and performance comparison for different I/O memory resources.

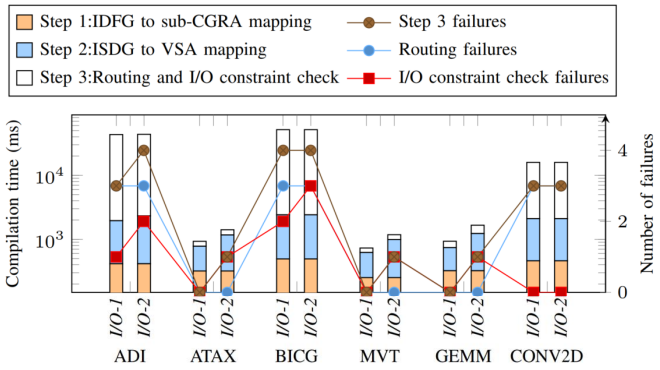


Fig. 18: Compilation time analysis on 8x8 CGRA (t^{Max} = 3, $MaxIterations$ = 10).

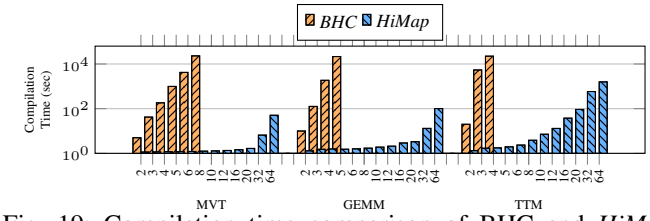


Fig. 19: Compilation time comparison of BHC and *HiMap* algorithms for different block sizes ($c = b_1 = b_2 = b_3 = b_4$) along X-axis.

IDFG. ADI, BICG, and CONV2D have more IDFG nodes compared to other benchmarks. Therefore, the compilation time for step 1 of those benchmarks is higher than the other benchmarks. ISDG to VSA (step 2) and routing and I/O constraint check (step 3) iterate over sub-CGRA mappings generated in step 1. Step 3 terminates (fails) when the available routing or I/O resources are insufficient to establish a valid mapping. When the number of failures increases, more time is spent on steps 2 and 3. Figure 18 also shows the number of failures in step 3 along with the number of routing failures and I/O constraint check failures. ADI, BICG, and CONV2D fail more times than the other benchmarks. Therefore more time is spent on steps 2 and 3 for those benchmarks. In step 3, the routing function dominates the compilation time over the I/O constraint check function because of the iterative nature of the routing algorithm. The number of failures in I/O-2 is higher

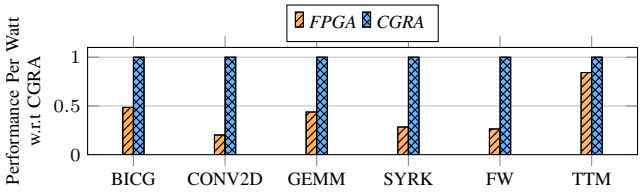


Fig. 20: Performance-per-watt comparison of CGRA and FPGA (for same block sizes).

than I/O-1 for all benchmarks except BICG and CONV2D. Therefore, compilation time for I/O-2 is higher than for I/O-1 on those benchmarks.

Figure 19 shows the compilation time comparison between *HiMap* and *BHC* on CGRA with I/O-1 memory configuration. 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 three 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, 5, and 8 for TTM, GEMM, and MVT, respectively. 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 I.

Comparison with FPGA: We also compare the CGRA performance-per-watt with *HiMap* against *Xilinx Zynq ZC702* FPGA. The *Vivado* High-Level Synthesis (HLS) tool estimates the performance and power values for the FPGA. The FPGA runs at a maximum clock rate of 200 MHz, and power consumption is around 120 mW$\sim$130 mW for the different kernels. We fully unroll the kernel with the maximum possible block sizes until we exceed the FPGA resources. The HLS compiler map all the operations in the unrolled DFG onto the compute resources (DSP, LUT), and the resource utilization increases with the DFG size. For a fair comparison, we also use the same block size for the CGRA. Figure 20 shows the performance-per-watt of FPGA and CGRA (with I/O-1 memory configuration) normalized w.r.t. the CGRA. On average, the CGRA achieves 1.9x better performance-per-watt compared to the FPGA. The FPGA falls behind the CGRA because the loop initiation interval of the mappings generated from HLS is higher than the *HiMap* for all the benchmarks. In addition, the overhead of fine-grained reconfigurability results in a slower clock frequency and higher power in FPGA.

Comparison with Specialized Architectures: We compare the *HiMap* mappings with the dataflow of manually designed specialized architectures. Due to reconfiguration overhead in CGRA, the performance and power efficiency are always lower in CGRAs compared to specialized architectures. Therefore, for a fair comparison, we compare the compute utilization of the dataflow mappings. Google Tensor Processing Unit (TPU) [25], and Microsoft Catapult [38] are prominent neural network accelerators. Both accelerators utilize systolic array microarchitecture to accelerate the GEMM kernel. The TPU systolic array employs weight-stationary dataflow, where filter weights are kept stationary in each PE. This dataflow is similar to the dataflow shown in Figure 3c when array B is considered a weight matrix. The Catapult systolic array employs output-stationary dataflow, where the partial output sums are accumulated in each PE. Both dataflows achieve 100% MAC utilization. We can represent both dataflows using a space-time mapping matrix (H, S). We use those (H, S) values to compile the GEMM kernel using *HiMap*. The resultant mappings achieve 100% utilization on I/O-1 CGRA. This proves the dataflow similarity of HiMap and manually designed systolic arrays. We also compare the *HiMap* CONV2D mapping with a DNN accelerator [24] which accelerates convolutional 2D kernel. The dataflow of the DNN accelerator is called output-oriented mapping (OOM). The space-mapping (S) of OOM is similar to the mapping obtained from HiMap. However, this specialized architecture can achieve 100% compute utilization while HiMap can achieve 70% utilization on I/O-1 CGRA. The DNN accelerator has multi-level data sharing network between PEs. This dedicated multi-level routing network allows complex dependency routing and enables high utilized mapping. In contrast, the limited network resources in a 2D mesh network in target CGRA causes routing failures and end up with low utilized mappings.

VIII. Conclusion

We introduced *HiMap*, a fast and scalable approach for mapping multi-dimensional kernels on bigger CGRAs. *HiMap* exploits the regular nature of the inter-iteration dependencies through hierarchical mapping. It first maps the iterations in a kernel onto a VSA to identify unique iterations. It then maps the unique iterations onto the CGRA individually, which then replicates and stitches together to generate the final complete mapping. *HiMap* achieves 17.3x performance and 5x performance-per-watt improvement with minimal compilation time than the state-of-the-art.

References

[1] C. Nicol, "A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing," *Wave Computing White Paper*, 2017.

[2] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–15, 2014.

[3] F. et al., "Intel's Exascale Dataflow Engine."

[4] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[5] B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "HyCUBE: A 0.9 V 26.4 MOPS/mW, 290 pJ/op, Power Efficient Accelerator for IoT Applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2019, pp. 133–136.

[6] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an Architecture-adaptive CGRA Mapping Tool," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 191–200.

[7] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid Optimization/heuristic Instruction Scheduling for Programmable Accelerator Codesign," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–15.

[8] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ureca: Unified register file for cgras," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1081–1086.

[9] L. Chen and T. Mitra, "Graph Minor Approach for Application Mapping on CGRAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–25, 2014.

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

[11] S. A. Chin and J. H. Anderson, "An Architecture-agnostic Integer Linear Programming Approach to CGRA Mapping," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[12] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 63–74.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.* IEEE, 2002, pp. 166–173.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.

[15] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE, 2000, pp. 32–32.

[16] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A Unified Framework for CGRA Modelling and Exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017, pp. 184–189.

[17] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP: Resource-aware Mapping for CGRAs," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[18] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "HiMap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1192–1197.

[19] S. Borkar, R. Cohn, G. Cox, T. Gross, H. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson *et al.*, "Supporting Systolic and Memory Communication in iWarp," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 70–81, 1990.

[20] R. Hughey and D. P. Lopresti, "Architecture of a Programmable Systolic Array," in *[1988] Proceedings. International Conference on Systolic Arrays*. IEEE, 1988, pp. 41–49.

[21] H.-T. Kung, "Why Systolic Architectures?" *Computer*, no. 1, pp. 37–46, 1982.

[22] S.-Y. Kung, S.-C. Lo *et al.*, "Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 603–614, 1987.

[23] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[24] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[26] P. Lee and Z. M. Kedem, "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays," *IEEE transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 64–76, 1990.

[27] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "Automatic Mapping of Nested Loops to FPGAs," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007, pp. 101–111.

[28] J. Cong and J. Wang, "PolySA: Polyhedral-based Systolic Array Auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[29] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[30] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEE Proceedings-Computers and Digital Techniques*, vol. 150, no. 5, pp. 255–261, 2003.

[31] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[32] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, vol. 437, 2012.

[33] Y. Yu and E. H. D'HOLLANDER, "Loop Parallelization Using the 3D Iteration Space Visualizer," *Journal of Visual Languages & Computing*, vol. 12, no. 2, pp. 163–181, 2001.

[34] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[35] S. Mehta, G. Beeraka, and P.-C. Yew, "Tile size selection revisited," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 1–27, 2013.

[36] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 110–119.

[37] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[38] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Toward accelerating deep learning at scale using specialized hardware in the datacenter," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–38.

Dhananjaya Wijerathne received the B.Sc.Eng. degree in electronics and telecommunication from University of Moratuwa, Moratuwa, Sri Lanka, in 2016. He is currently working toward a Ph.D. degree from the National University of Singapore, Singapore. He was a Senior Electronic Engineer with ParaQum Technologies, Colombo, Sri Lanka, from 2016 to 2018. His research interests include programmable accelerators and compiler optimizations.

Zhaoying Li received the B.S. degree in software engineering from Shandong University, Jinan, China, in 2018. He is currently working toward a Ph.D. degree from the National University of Singapore, Singapore. His current research interests include reconfigurable architectures and compiler optimizations.

Anuj Pathania received his Ph.D. degree from Karlsruhe Institute of Technology (KIT), Germany in 2018. He is currently an Assistant Professor at the University of Amsterdam. His research focuses on the high-performance low-power design of embedded systems.

Tulika Mitra received a BE degree in computer science from Jadavpur University, Kolkata, India, in 1995, an ME degree in computer science from the Indian Institute of Science, Bengaluru, India, in 1997, and a Ph.D. degree from the State University of New York, Stony Brook, NY, USA, in 2000. She is currently Provost's Chair Professor of Computer Science at the School of Computing, National University of Singapore, Singapore. Her research interests include the design automation of embedded real-time systems with particular emphasis on software timing analysis/optimizations, application-specific processors, energy-efficient computing, and heterogeneous computing.

Lothar Thiele received the Diplom-Ingenieur and Dr.Ing. degrees in electrical engineering from the Technical University of Munich, Munich, Germany, in 1981 and 1985, respectively. His current research interests include models, methods, and software tools for the design of embedded systems, embedded software, and bioinspired optimization techniques. Mr. Thiele was a recipient of the Dissertation Award of the Technical University of Munich, in 1986, the Outstanding Young Author Award of the IEEE Circuits and Systems Society in 1987, the Browder J. Thompson Memorial Award of the IEEE in 1988, the IBM Faculty Partnership Award in 2000 and 2001, the Honorary Blaise Pascal Chair of University of Leiden, in 2005, and the "EDAA Lifetime Achievement Award" in 2015.