CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA

DHANANJAYA WIJERATHNE, ZHAOYING LI, MANUPA KARUNARATHNE, ANUJ PATHANIA, and TULIKA MITRA, National University of Singapore, Singapore

A Coarse-Grained Reconfigurable Array (CGRA) is a promising high-performance low-power accelerator for compute-intensive loop kernels. While the mapping of the computations on the CGRA is a well-studied problem, bringing the data into the array at a high throughput remains a challenge. A conventional CGRA design involves on-array computations to generate memory addresses for data access undermining the attainable throughput. A decoupled access-execute architecture, on the other hand, isolates the memory access from the actual computations resulting in a significantly higher throughput.

We propose a novel decoupled access-execute CGRA design called *CASCADE* with full architecture and compiler support for high-throughput data streaming from an on-chip multi-bank memory. *CASCADE* offloads the address computations for the multi-bank data memory access to a custom designed programmable hardware. An end-to-end fully-automated compiler synchronizes the conflict-free movement of data between the memory banks and the CGRA. Experimental evaluations show on average 3x performance benefit and 2.2x performance per watt improvement for *CASCADE* compared to an iso-area conventional CGRA with a bigger processing array in lieu of a dedicated hardware memory address generation logic.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; *Heterogeneous (hybrid) systems.*

Additional Key Words and Phrases: Coarse Grained Reconfigurable Arrays, Multi-Bank Memory partitioning, Decoupled access-execute architectures,

ACM Reference Format:
Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 20XX. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Trans. Embedd. Comput. Syst.* XX, X, Article XXX (October 20XX), 25 pages. https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

1 INTRODUCTION

A modern computing system increasingly relies upon hardware accelerator(s) working in tandem with a Central Processing Unit (CPU) to feed the incessant increase in demand for power-efficient computations. A Coarse-Grained Reconfigurable Array (CGRA) has emerged as a prominent accelerator because it offers a good blend of computational throughput, power-efficiency, and reconfigurability [19, 28, 32, 35]. The CGRA by design is an array of Processing Elements (PEs)

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2019.
Authors' address: Dhananjaya Wijerathne, dmd@comp.nus.edu.sg; Zhaoying Li, zhaoying@comp.nus.edu.sg; Manupa Karunarathne, manupa@comp.nus.edu.sg; Anuj Pathania, pathania@comp.nus.edu.sg; Tulika Mitra, tulika@comp.nus.edu.sg, National University of Singapore, Department of computer science, School of Computing, Computing 1, 13 Computing Drive, Singapore, 117417.

1539-9087/20XX/10-ARTXXX $15.00
https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

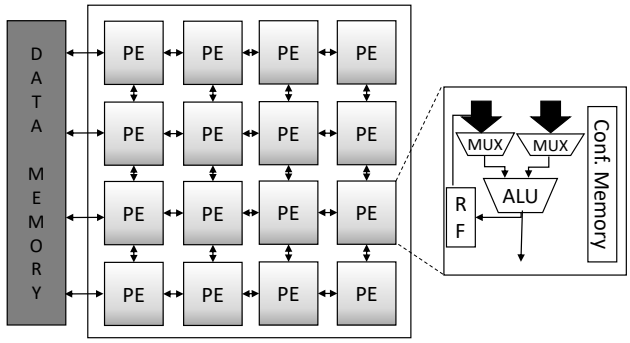


Fig. 1. Conventional 4x4 CGRA architecture.

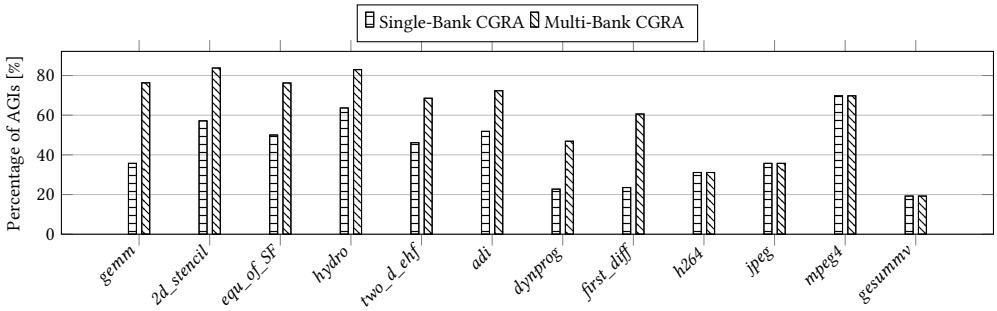


Fig. 2. Percentage of AGIs in kernels for single and multi-bank CGRA.

connected using a 2D mesh network, as shown in Figure 1. A PE typically constitutes an Arithmetic Logic Unit (ALU), register file, and configuration memory.

The CGRA is best suited to accelerate the frequently executed compute-intensive loop kernels. A CGRA compiler statically schedules the loop kernel on to the PEs and places the necessary instructions in the configuration memory before the commencement of execution. A PE executes the instructions stored in its configuration memory in every cycle with operands obtained from either its register file or from its neighboring PEs. The PE then writes the result into its register file or transports the results to one or more neighboring PEs or both. Some PEs can access the on-chip data memory to fetch the operands and store the results.

A conventional CGRA design involves on-array computation to generate the data memory addresses corresponding to the load/store operations. We observe that a substantial percentage of PEs are allocated to execute Address Generation Instructions (AGIs) in a kernel. Moreover, an on-chip multi-bank memory is often deployed with the CGRA to offer high memory bandwidth as it allows multiple data memory accesses in the same cycle [35]. CGRA requires higher bandwidth to support the high throughput data streaming loop kernels. Unfortunately, the multi-bank memory also requires additional computations to determine the bank indices (using a banking function) and intra-bank offsets (using an intra-bank offset function) to provide conflict-free memory accesses further inflating the involved AGIs [45]. Figure 2 shows the rise in the percentage of AGIs in the kernels when we move from a single-bank to multi-bank CGRA. The AGI percentage is quite high ranging from 20% to 80%. It is possible to provide conflict-free access in the multi-bank CGRA even without the use of the banking or intra-bank offset function by taking care of the bank conflicts in the CGRA mapping schedule [22]. However, this approach generally results in a higher Initiation

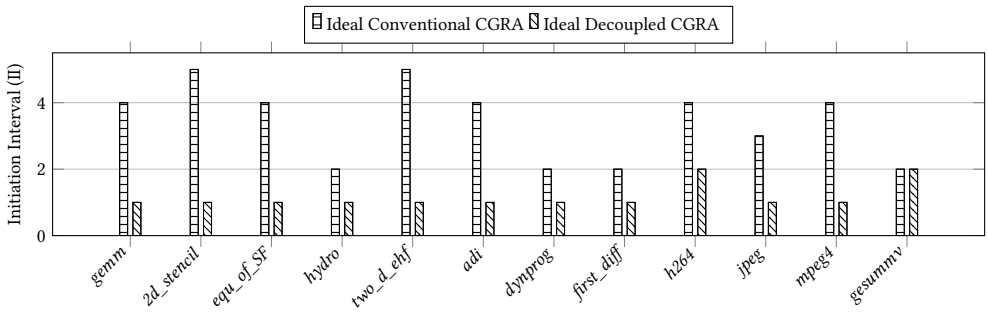


Fig. 3. II for kernels corresponding to an ideal 4x4 conventional and a decoupled CGRA. Both CGRAs have perfect memory system. A lower II value results in higher throughput.

Interval (II). The number of cycles between consecutive loop iterations defines the II. A higher II results in lower resource utilization and throughput.

The on-array AGI execution overhead motivates the use of a decoupled access-execute CGRA design [18, 31] that advocates the separation of execution of AGIs from the remaining actual computational instructions to improve the loop kernel performance. Figure 3 shows the performance potential of an ideal decoupled access-execute CGRA compared to an ideal conventional CGRA. We assume that there are no memory conflicts in both the CGRAs, i.e., the memory subsystem can provide infinite bandwidth and one-cycle latency for load/store accesses. The main difference is that in the case of the conventional CGRA, the memory address generation computations are scheduled on the PE array consuming precious resources. But in the ideal decoupled access-execute CGRA, the memory address generation computations are moved out of the CGRA and can be executed with zero overhead. In other words, the CGRA focuses only on pure computation with an ideal memory. This ideal decoupled CGRA can reduce the II of the loop kernels by 80% on an average compared to an ideal conventional CGRA resulting in 5x increase in the throughput. Unfortunately, the few existing decoupled access-execute CGRAs [18, 31] cannot achieve this ideal performance because (a) they do not support multi-bank memory essential for high bandwidth and (b) they lack the compiler support necessary for conflict-free memory access.

CASCADE fills this gap by designing a holistic decoupled access-execute CGRA. *CASCADE* includes a novel customized hardware for multi-bank memory address generation. The address generation typically corresponds to regular computations on iteration variables that determine the load/store addresses, and most loop kernels exhibit strided address generation patterns. The banking and intra-bank offset functions also have standard templates that are applicable across loop kernels. Therefore, the address generation computations are more suited to be offloaded to custom programmable hardware than to be imposed onto the CGRA. *CASCADE* supplies the data to the PE array of the CGRA at the same rate as the array consumes the data to prevent any stalling of the on-array computations. The architecture is adroitly supported by an automated compiler that performs conflict-free data placement in the multi-bank memory, programs the hardware for address generation, and maps the computation onto the CGRA in tandem for perfect synergy between the memory access and computation components. *CASCADE* accomplishes close to the ideal CGRA performance — an average 3x improvement compared to conventional CGRA with a similar overall area but bigger array size to substitute for the additional memory address generation logic in *CASCADE*. Finally, an additional benefit of our approach is the substantially reduced compilation time. It is far easier for the compiler to generate the schedule when the address generation operations no longer need to be scheduled on the CGRA.

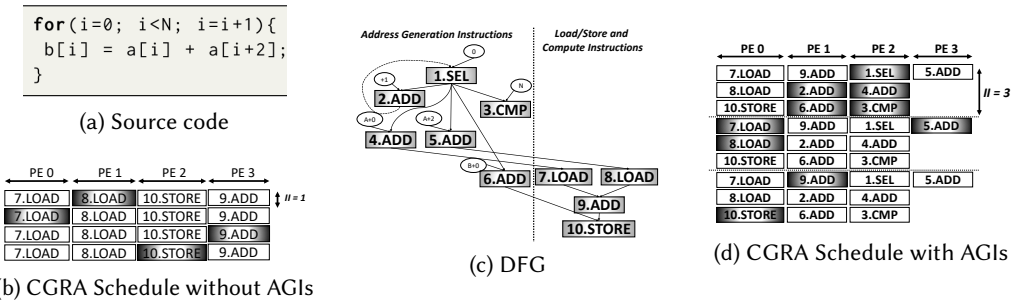


Fig. 4. Illustration of II reduction in a CGRA schedule from three to one by offloading the address generation.

We note that both the decoupled access-execute architecture and the compiler support for multi-bank memory have been explored before *independently* in the context of CGRAs [45] [31]. *CASCADE* is the first CGRA design that investigates both the opportunities and challenges associated with the decoupled access-execute architecture in conjunction with the multi-bank memory. In particular, we observe high-performance overhead of on-array memory address generation, especially with multi-bank memory. In the case of Field Programmable Gate Arrays (FPGAs), the high-level synthesis tools embed arbitrarily complex address generation logic with the actual computation [13]. But, the overhead is far higher in case of CGRAs with a limited number of PEs meant to perform computation. Thus, decoupling address generation from computation provides substantial performance benefit in CGRAs. Moreover, unlike previously proposed decoupled architectures, software-based address generation with one or more CPUs is incapable of supporting high-throughput data streaming to the on-chip array [21]. This issue has been largely ignored in the literature and motivated us to design custom programmable hardware responsible for fast address generation in the presence of a multi-bank memory. Finally, compiler support for decoupled access-execute CGRAs is missing [31]. Our compiler configures address generation hardware as well as orchestrates data movement between the access and execute components in tandem with the computation mapping. In summary, the novelty of *CASCADE* lies in providing an end-to-end solution with architecture and compiler support for high-throughput data streaming on CGRAs.

2 MOTIVATING EXAMPLE

We present a motivating example to illustrate the offload of AGIs from the PE array in Figure 4. Figure 4c shows the Data Flow Graph (DFG) corresponding to a simple loop kernel shown in Figure 4a. The DFG nodes represent the operations, whereas the edges represent the data dependencies between the operations. Six operations – Operations 1 to 6 – correspond to the address generation while only Operation 9 corresponds to the actual computation. Operations 1, 2, and 3 are related to the iteration variable initialization, increment, and bound checking, respectively. Operations 4, 5, and 6 add the base addresses of the arrays to the iteration variable to generate the effective addresses used in the load (Operations 7 and 8) and store (Operation 10) operations.

CGRA mapping algorithms use the iterative modulo scheduling [34] to place the DFG operations on the PEs in a software pipelined fashion and also route the data dependencies among the operations. Figure 4d shows the optimal CGRA schedule of the DFG shown in Figure 4c on a 4x1 CGRA. The shaded nodes belong to the current iteration, while the white nodes represent the previous or next iterations. The schedule repeats every three cycles, i.e., II=3.

Figure 4b shows the CGRA schedule of the DFG without AGIs, wherein II=1 leads to a 3x performance improvement. *CASCADE* can achieve II=1 because it offloads address generation and

memory accesses to a specialized hardware unit called Stream Engine (SE) as shown in Figure 5. SE fetches data from the multi-bank memory and feeds the CGRA through Stream Registers (SR).

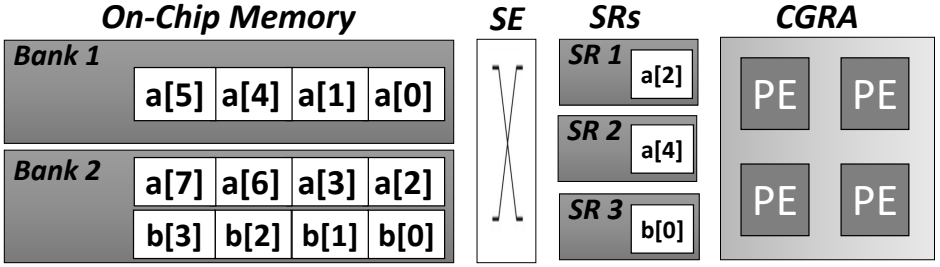


Fig. 5. Abstract diagram showing data transfer from multi-bank memory via SE to CGRA.

Each load and store operation in the loop kernel is considered a different *stream*. In our example, $a[i]$ and $a[i+2]$ are two separate load streams while $b[i]$ is a store stream. The memory addresses accessed by stream $a[i]$ and $a[i+2]$ are $[0, 1, 2, \ldots, N-3]$ and $[2, 3, 4, \ldots, N-1]$, respectively. Our compiler guarantees conflict-free memory access by the different streams by making sure that the data elements accessed in parallel are not placed in the same bank. The compiler generates conflict-free banking function to determine data placement in memory banks when two parallel streams access the same data array. The original addresses are no longer valid after data placement with the banking function. Each data element has a new memory location that can be expressed using a bank ID and intra-bank offset.

Let x be the original address in our example. The banking and intra-bank offset function of array a is then $\lfloor \frac{x}{2} \rfloor \% 2$ and $\lfloor \frac{x}{4} \rfloor \times 2 + x\%2$, respectively. So stream $a[i]$ and $a[i+2]$ maps to an access sequence with (bank ID, intra-bank offset) pairs $[(0,0), (0,1), (1,0), (1,1), \ldots]$ and $[(1,0), (1,1), (0,2), (0,3), \ldots]$, respectively. The banking function ensures that the bank IDs of the data accessed in parallel are conflict-free. Host processor configures SE with the banking function and stream access parameters. SE generates bank ID and intra-bank offset of each stream at run-time to fetch data and place them in SRs. CGRA needs to load/store data from the corresponding SRs.

3 RELATED WORK

A CGRA achieves high performance and power-efficiency through its considerably streamlined architecture. The simplicity of the architecture requires a complex compiler where the compiler must explicitly map computations in a target kernel on to the CGRA's PEs considering both space and time dimensions. It also has to determine the routing (communication) between PEs and generate the necessary configurations (for the muxes) to route the dependencies [28]. The CGRA mapping problem has been extensively studied [6, 7, 11, 14–16, 20, 39], but only a few works take into consideration the interdependence between memory management and CGRA mapping. Authors of [23, 43–45] propose conflict-free loop kernel mapping algorithms considering both memory management and CGRA mapping. Authors in [44, 45] propose an array element-level data partitioning mechanism using a banking function. However, they do not consider the hardware/software overhead of the function's execution.

In a multi-bank memory, memory conflict arises if two data elements accessed in the same cycle reside in the same memory bank with a single read port. A CGRA compiler needs to ensure that the number of data accesses per bank per cycle is less than the number of available ports in one memory bank. A naive approach will be to maintain multiple copies of the data arrays accessed in parallel in different banks, but limited on-chip memory size makes this approach unsustainable. A

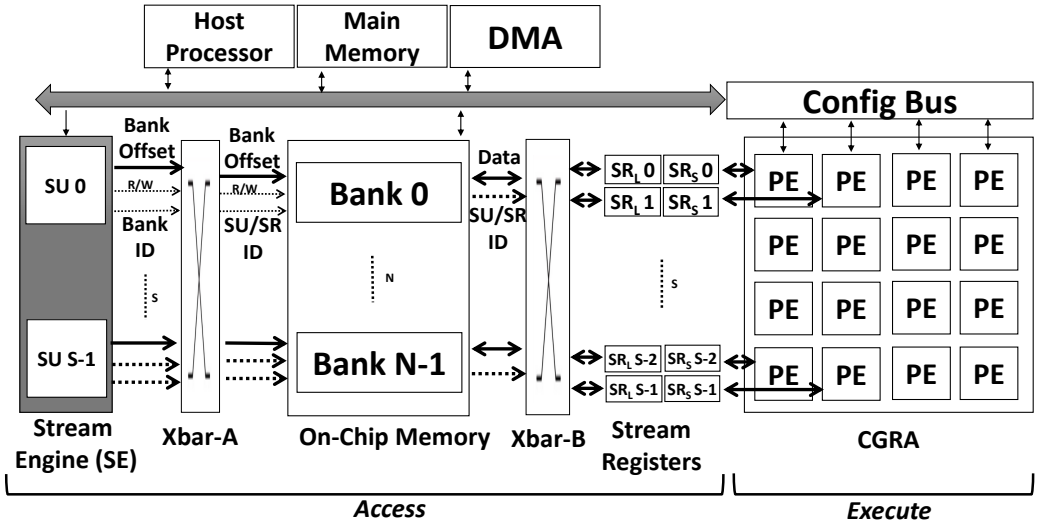


Fig. 6. *CASCADE* architecture.

better solution would be to let the compiler partition the data into memory banks to avoid access conflicts without any data replication.

Several recent works have studied the multi-bank data placement problem [30, 33, 37, 38]. Most of the previous works on this problem target FPGAs, and they design arbitrarily complex mapping functions without a fixed format. Even though it is possible to synthesize those functions into FPGA hardware, it is not possible to implement them as configurable hardware units. Section 5.4 provides additional background details about the multi-bank data placement problem.

The authors of [36] were the first to propose the generic decoupled access-execute architecture model. Many hardware accelerators [5, 8] adapt this model and use stream-based memory access to improve performance. Recent works [12, 18, 31] introduced the decoupled access-execute model to CGRAs, where they provide detailed programming interface and architecture abstraction for their architectural design but lack compiler support. As a result, the programmer has to rewrite applications in the architecture-specific intrinsic. Moreover, none of the existing works on decoupled access-execute CGRAs provide a solution for multi-bank data placement problem. *CASCADE design proposed in this work is the first decoupled access-execute CGRA to support multi-bank memories with dedicated programmable address generation hardware units and automated compiler support for both computation and memory access.*

4 CASCADE ARCHITECTURE

Figure 6 shows a high-level view of *CASCADE* architecture motivated by decoupled access-execute architectural model [36]. It consists of *Access* and *Execute* components responsible for memory access and compute operations, respectively. The memory access operations handle multi-bank memory address generation, while we define the remaining operations as compute operations because they operate on the data.

The *Execute* component is identical to a conventional CGRA where a set of PEs is arranged in a grid with each PE connected to its neighbors. Each PE consists of ALU, register file, and control memory to store configuration information. The *Access* component consists of a stream engine (SE), multi-bank Scratch Pad Memory (SPM), and set of stream registers (SR) connected using two crossbars as shown in Figure 6. The data is brought into on-chip multi-bank memory from off-chip

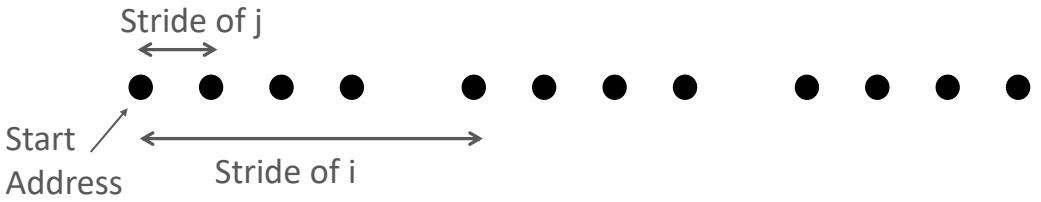


Fig. 7. Two-stride stream address sequence.

memory via Direct Memory Access (DMA) operations, similar to conventional CGRAs. On-chip memory guarantees service to all memory accesses from the *Access* component. Each memory access operation can be modeled as a stream because it is executed repetitively in the loop. There are two types of streams, load and store streams. The SE is composed of multiple Stream Units (SUs) wherein each SU is assigned a stream. These streams interface to the PEs through two types of SRs: load stream register (SR_L) and store stream register (SR_S). Each PE connects to at most one pair of $\{SR_L, SR_S\}$ registers; the PEs that does not connect to any SR need to access memory data through other PEs.

Each SU is responsible for address generation and bank selection to support delivery/reception of data to/from $\{SR_L, SR_S\}$. Each SU produces (bank ID, intra-bank offset) sequence for the assigned stream, corresponding to the execution of memory instruction in the loop. The crossbar Xbar-A connects the SUs to the banks, and crossbar Xbar-B connects the banks to the SRs. During execution, each SU selects a non-conflicting bank (guaranteed by the compiler) by setting Xbar-A and transmits the intra-bank offset. Once data comes out of the bank, the SU configures Xbar-B to transmit the data to the matching $\{SR_L, SR_S\}$ pair.

4.1 Stream Tuple Definition

The addresses of a load/store stream can have arbitrarily complex pattern based on the number of iteration variables used to generate them. This pattern also depends upon the start address, loop bounds of iteration variables, and strides (multiple of iteration variables). We define a generic, fundamental stream pattern with two strides using a stream tuple. This tuple can be used as a basic building block to support arbitrarily complex patterns in *CASCADE*. Figure 7 shows the addresses defined by stream tuple <SA, AS_j, AS_i, max_j, max_i> where SA is the start address, AS_j is the address stride of iteration variable j, AS_i is the address stride of iteration variable i, max_j is the loop bound of j, and max_i is the loop bound of i. The address generation computation defined by the stream tuple is functionally equivalent to Listing 1.

Listing 1. Memory address computation by stream tuple.

```
for( i=0; i<max_i; i=i+1){
    for( j=0; j<max_j; j=j+1){
       addr = AS_i*i + AS_j*j + SA;
    }
}
```

Address generation on the host processor is not a viable option because it takes on an average of 18 *ARM64* instructions to generate a single address for a stream tuple. So we design a configurable hardware unit (SE) that can generate an address per cycle for a stream tuple. Host CPU initially configures the SE using the stream tuple, and subsequently, the SE generates the strided address stream. *CASCADE* is not limited to two-stride stream patterns. Section 5.3 explains the generation of

strided address stream for streams with more than two strides through an iterative reconfiguration of the SE by the host CPU.

4.2 Stream Address Generation in Hardware

Address generation for a stream tuple can be performed more efficiently by adding a linear shift to the previous address in the sequence. Affine access $\phi(i, j)$ can be expressed as follows.

$$\phi(i,j) = \begin{bmatrix} AS_i & AS_j \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + SA \tag{1}$$

$$\Delta\phi = \begin{cases} S2, & \text{if } j_n = max_j - 1 \\ S1, & \text{otherwise} \end{cases} \tag{2}$$

$$S1 = AS_j \qquad S2 = AS_i - max_j.AS_j$$

$$j_{n+1} = \begin{cases} 0, & \text{if } j_n = max_j - 1 \\ j_n + 1, & \text{otherwise} \end{cases} \qquad i_{n+1} = \begin{cases} 0, & \text{if } i_n = max_i - 1 \\ i_n + 1, & \text{if } j_n = max_j - 1 \\ i_n, & \text{otherwise} \end{cases}$$

$$\phi_{n+1} = \phi_n + \Delta\phi \tag{3}$$

The $(n+1)^{th}$ access address is equal to the summation of $\Delta\phi$ and n^{th} access address. $\Delta\phi$ is a function of the strides and iteration variables as given by Equation (2). Section 4.4 explains how this formulation leads to an efficient address generator implementation.

4.3 Bank and Offset Functions

The bank function $\beta(x)$ and intra-bank offset function $O(x)$ map address x to bank ID and intra-bank offset, respectively.

$$\beta(x) = \lfloor \frac{x}{B} \rfloor \% N \tag{4}$$

$$O(x) = \lfloor \frac{x}{N.B} \rfloor .B + x\%B \tag{5}$$

where N is the number of banks, and B is the block size. Values of B and N depend on the access pattern of the streams, and Section 5.4 explains the method to obtain them.

4.4 Stream Engine (SE) Architecture

Figure 8 shows the detailed architecture of the Stream Engine (SE) that generates bank ID and intra-bank offset of S streams per cycle. It consists of one SU per stream and a shared Iteration Variable Generation (IVG) unit. SU consists of a Stream Address Generation Unit (SAGU), Bank Address Generation Unit (BAGU), and configuration memory to hold the stream tuple and bank/offset parameters.

SAGU and IVG generate stream addresses based on Equation (2). SAGU has an address accumulator (Acc_{addr}) initialized to SA. In IVG, two accumulators increment iteration variables i and j conforming to loop bounds. Acc_j holds j, and increments it in each clock cycle. When Acc_j reaches max_j, Acc_j resets to 0, and Acc_i increments i. IVG generates a signal inside SAGU to select between $S1$ and $S2$. If Acc_j is equal to $max_j - 1$, S2 will be selected else S1 will be selected. The selected output is added to the current address to generate the next address.

BAGU generates bank ID and intra-bank offset according to Equations (4) and (5), respectively. Floor division and modulo operations can be implemented using shifters if B and N are both

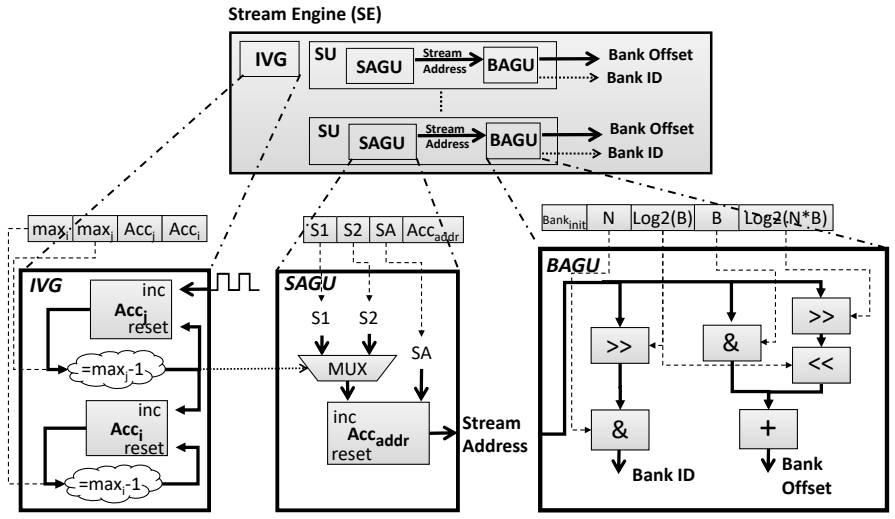


Fig. 8. Detailed architecture of Stream Engine (SE).

power-of-two. So we restrict B and N to power-of-two values. Input for BAGU is stream address generated by SAGU and output is the bank ID and intra-bank offset of each stream.

SUs are capable of generating addresses in a time-multiplexed manner to support more than S streams by holding multiple stream tuples and bank/offset parameters. We sequence through the configuration memory of the SU according to the II value of the CGRA schedule.

4.5 Data Transfer from Off-Chip Memory

The CGRA operates on multi-bank on-chip SPM data. Host processor configures DMA engine to move data between off-chip memory and banked SPM. DMA engine calculates SPM bank location (bank ID and intra-bank offset) of each data element using bank/offset functions when it accesses the SPM. Dynamic SPM management technique is used to overlap data filling and consumption [10, 40]. Data transfer between off-chip and on-chip memory takes place at the granularity of blocks. The host issues DMA operations to move data blocks to/from SPM while CGRA operates on a different data block. The block size and number of blocks present in on-chip memory at any point in time are calculated based on the SPM size and memory requirement of all arrays.

5 CASCADE COMPILER

Figure 9 shows the *CASCADE* compiler framework. The compiler generates the CGRA mapping, SE configurations, and conflict-free banking function parameters. The compiler first extracts the AGI-free DFG by removing all AGIs. It then gives the DFG to the CGRA mapper to generate the CGRA schedule and routes. Memory access patterns are extracted and subsequently used to generate the stream tuples and conflict-free banking function parameters.

5.1 Definitions

The compiler converts all loops to normalized loops in which the loop variable starts at 0 or a constant and gets incremented by one at each iteration until the exit condition is satisfied. We define affine array access [4] for a d-dimensional array in a normalized l-level loopnest using iteration vector $\vec{i}$ by $< A, C, LB >$ tuples where A is d x l matrix of strides, C is d-dimensional vector

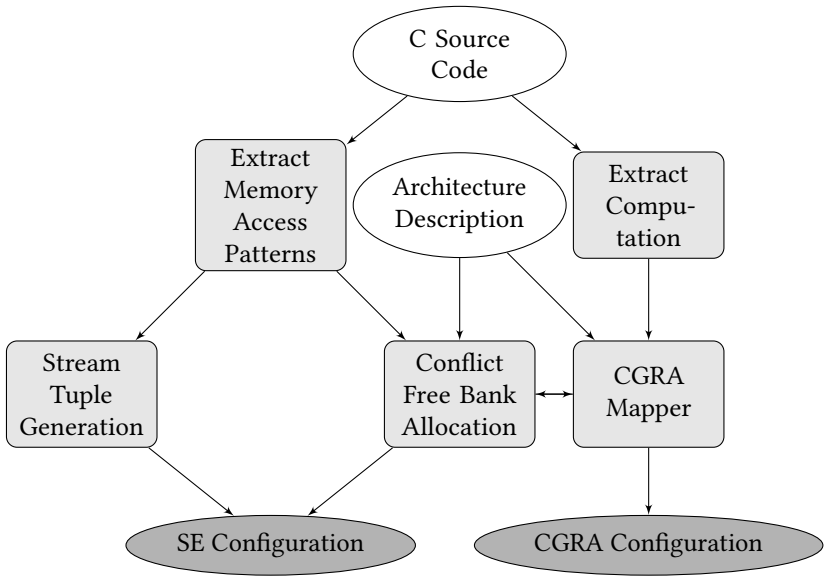


Fig. 9. *CASCADE* compiler framework.

of constant start offsets, and LB is l-dimensional vector of loop bounds. It maps the iteration vector with loop bounds $0 \leq \vec{i} \leq LB$ to an array element location $\phi(\vec{i}) = A\vec{i} + C$ where

$$\vec{i} = \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{l-1} \end{bmatrix} \qquad LB = \begin{bmatrix} lb_0 \\ lb_1 \\ \vdots \\ lb_{l-1} \end{bmatrix} \qquad C = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{d-1} \end{bmatrix}$$

$$A = \begin{bmatrix} a_{0,0} & \dots & a_{0,l-1} \\ a_{1,0} & \dots & a_{1,l-1} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \dots & a_{d-1,l-1} \end{bmatrix}$$

Iteration domain M is formed by $\vec{i}$ within the loop bounds. We define access pattern P^R as the set of m affine accesses in the loop kernel which access the data array R.

$$P^R = \{\phi_1(\vec{i}), \dots, \phi_m(\vec{i})\}$$

AGI-free DFG $D = (V_D, E_D)$ is obtained by removing all AGIs of affine accesses in the loop kernel. V_D and E_D are vertex and edge set of the AGI-free DFG D, respectively. Modulo Routing Resource Graph (MRRG) [29] is the time extended (II cycles) resource graph of the CGRA. The CGRA compiler establishes a mapping from the AGI-free DFG to the MRRG. We define the control step to identify the set of parallel instructions executed at each cycle in MRRG, i.e., MRRG has II number of control steps.

We define control step access pattern P^R_S as a subset of affine accesses in the loop kernel access pattern (P^R) that access data array R at control step S.

$$P^R_S = \{\phi_1(\vec{i}), \dots, \phi_q(\vec{i})\}$$

5.2 Extraction of Access, Execute Components

The input to *CASCADE* compiler is *C* source code where the loop kernels targeted for acceleration are annotated by pragmas and can be easily identified and extracted. We use *Clang C* compiler to obtain *LLVM IR* (bitcode) of each target loop kernel. *CASCADE* compiler analyzes *LLVM IR* for data and control dependencies. Memory access patterns and pure computation components are then extracted.

Extraction of computation (execute) component: The *LLVM IR* is analyzed to generate AGI-free DFG that consists of computation and load-store operations but does not include address generation logic for load/store instructions. In *LLVM IR*, Get Element Pointer (GEP) instructions precede all load/store operations. We disregard all GEP instructions and GEP dependent parent instructions to generate AGI-free DFG.

Extraction of Memory Access Patterns: The compiler extracts memory access patterns corresponding to all load/store operations in the loop kernel in the form of affine array access (Section 5.1). As GEP instructions precede load/store operations, we analyze the parent nodes of GEP instructions to find the access patterns corresponding to load/store operations. With GEP as a leaf node, we recursively traverse all the previous data-dependent instructions until we reach the iteration variables. *LLVM* inbuilt function (*getCanonicalInductionVariable*) is used to identify all the iteration variables. The traversal captures the mathematical equations involved in the address calculation. Equations are later converted to the affine format to obtain matrix A, vector C, and LB.

5.3 Nested Loop Mapping

When the compiler maps nested loops on to CGRAs, it places the innermost loop body operations on to the CGRA PEs. Outer loop iteration variables maintain fixed values for one invocation of the innermost loop. Typically the host processor handles loop control, i.e., calculates the values of the outer loop iteration variables and sends them to the array when invoking a new innermost loop instance [25, 26]. Some optimizations like loop flattening may reduce the number of invocations at the cost of overhead because the compiler maps computations to evaluate outer loop iteration variables onto the array [25].

There are many ways to handle imperfect nested loops, where the outer loop body also contains statements [25]. These statements can be fissioned into a new loop to be executed by the host processor or guarded by predicates and moved into the innermost loop. We transform imperfect nested loops to perfect nested loops by guarding outer loop statements through predicates [17].

In *CASCADE*, nested loop control is handled through SUs, as we offload AGIs including iteration variables evaluations to SUs. The datapath of a SU can generate strided access pattern corresponding to two iteration variables in one invocation. Host processor needs to configure the SUs to handle access patterns with more than two iteration variables. For example, to handle the three-level loopnest shown in Listing 2, the host processor has to reconfigure each SU N times where N is the trip count of the outermost loop. Our compiler generates host processor code and corresponding SU configuration (stream tuple) as shown in Listing 3.

Listing 2. Three-level nested loop.

```
for( k=0; k<N; k=k+1){
    for( i=0; i<M; i=i+1){
       for( j=0; j<L; j=j+1){
         .. = a[4*k+4*i+j];
       }
    }
 }
```

Listing 3. Stream tuple generation for loop nest in Listing 2.

```
for( k=0; k<N; k=k+1){
    Stream<SA=4*k, AS_j=1, AS_i=4, max_j=L, max_i=M>
}
```

We now explain how a strided access pattern of arbitrary depth in a loop can be expressed using the hardware supported strided pattern, for example from the access pattern in Listing 2 to the sequence of stream tuples in Listing 3. The objective is to express extracted affine memory access in terms of stream defined by the tuple $< SA, AS_j, AS_i, max_j, max_i >$. We first flatten multi-dimensional array accesses into a single dimension.

$$A = \begin{bmatrix} a_{0,0} & \dots & a_{0,l-1} \\ a_{1,0} & \dots & a_{1,l-1} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \dots & a_{d-1,l-1} \end{bmatrix} \Rightarrow A^{flat} \quad , C = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{d-1} \end{bmatrix} \Rightarrow C^{flat}$$

Here A^{flat} and C^{flat} vectors are obtained by flattening matrix A and vector C (Section 5.1) to one dimension, as shown below.

$$A^{flat} = \begin{bmatrix} a'_0 & \dots & a'_{l-1} \end{bmatrix} \quad , C^{flat} = \begin{bmatrix} c' \end{bmatrix}$$

$$a'_n = a_{0,n} + a_{1,n} * w_0 + a_{2,n} * w_0 * w_1 + \dots + a_{d-1,n} * \prod_{k=0}^{d-2} w_k$$

$$c' = c_0 + c_1 * w_0 + c_2 * w_0 * w_1 + \dots + c_{d-1} * \prod_{k=0}^{d-2} w_k$$

where $W = [w_0, w_1, \dots, w_{d-1}]$ is the vector of widths of each dimension of the accessed array. If there are more iteration variables than what hardware can support (in our case two), the start address absorbs the iteration variables except for the ones in the two innermost loops. Thus, we can express the start address as an affined function of new iteration vector with $\vec{i'} = \begin{bmatrix} i_0 & i_1 & \dots & i_{l-3} \end{bmatrix}$.

$$SA = c' + a'_0 * i_0 + \dots + a'_{l-3} * i_{l-3};$$

AS_i, AS_j, max_i, and max_j are equal to the values correspond to two innermost iteration variables of A^{flat} and LB.

$$AS_i = a'_{l-2}; \quad AS_j = a'_{l-1}; \quad max_i = lb_{l-2}; \quad max_j = lb_{l-1};$$

5.4 Conflict-Free Memory Bank Allocation

Previous memory partitioning algorithms focus on supporting a single access pattern [30, 37, 38]. In CGRA, it is possible to access the same data array in different access patterns in a mutually exclusive manner. We define these access patterns as control step access patterns ($P^R_1, .., P^R_S$). Control step access patterns occur when memory accesses for the same data array happens in different control steps. Memory partitioning algorithm should place data array on memory banks considering all control step access patterns. Authors in [41, 42] analyze this problem, but both works use linear transformation based cyclic partitioning scheme that leads to the complex bank and intra-bank offset functions. These functions can be synthesized in FPGAs (though they occupy area) but far more challenging to compute inside CGRAs because it leads to increased schedule length and II.

Background: There are two basic memory partitioning schemes, cyclic and block partitioning. In cyclic partitioning, we cyclically partition data into different banks. Cyclic partitioning memory bank function is $x\%N$, where N is the number of banks and x is the address of the data element, and the intra-bank offset function is $\lfloor \frac{x}{N} \rfloor$. Block partitioning divides the data into a set of blocks and then stores them in different memory banks. The block partitioning memory bank function is $\lfloor \frac{x}{B} \rfloor$, and intra-bank offset function is $x\%B$. We use block-cyclic partitioning which is the combination of both these partitioning schemes. The banking function of block-cyclic partitioning has the form of $\lfloor \frac{x}{B} \rfloor \%N$.

Recently, authors in [37] introduced a linear transformation vector $\vec{\alpha}$ to block-cyclic partitioning and named it Generalize Memory Partitioning (GMP). GMP defines the bank mapping function using the following equation.

$$\beta(\vec{x}) = \left\lfloor \frac{\vec{\alpha}.\vec{x}}{B} \right\rfloor \%N$$

For a given access pattern, memory partitioning algorithm finds α, N, and B such that partitioned memory does not yield any access conflicts. The GMP algorithm enumerates through α, N and B. For each α, N, and B it takes two access streams at a time and checks whether they are conflict-free after applying the banking function to them. The condition that needs to be satisfied for valid banking function β for two access streams $\phi_0(\vec{i}) = A_0.\vec{i} + C_0$ and $\phi_1(\vec{i}) = A_1.\vec{i} + C_1$ is stated below.

$$\forall \vec{i} \in M, \beta(\phi_0(\vec{i})) \neq \beta(\phi_1(\vec{i}))$$

We check the above condition through a polyhedral based method called conflict polytope testing.

Definition: (Conflict Polytope) A conflict polytope of two simultaneous access streams $\phi_0(\vec{i}) = A_0.\vec{i} + C_0$ and $\phi_1(\vec{i}) = A_1.\vec{i} + C_1$ is a parametric polytope restricted to the iteration domain M as stated below.

$$P_{conf}(\phi_0(\vec{i}), \phi_1(\vec{i})) = \{\vec{i} |\ \ \forall \vec{i} \in M,\ \ \beta(\phi_0(\vec{i})) = \beta(\phi_1(\vec{i}))\}.$$

The polytope is essentially a set of inequalities obtained from the above conflicting condition. If the conflict polytope $P_{conf}(\phi_0(\vec{i}), \phi_1(\vec{i}))$ is empty (number of integer points in the conflict polytope is 0), it implies $\forall \vec{i} \in M, \beta(\phi_0(\vec{i})) \neq \beta(\phi_1(\vec{i}))$. Counting the number of integer points in a polytope is a well-formulated problem in linear algebra and can be solved using *Ehrhart* algorithm [9]. The GMP algorithm checks the conflict-free condition across all combinations of the access streams in the access pattern. If all combinations satisfy the conflict-free condition, the respective α, N, and B would be the parameters for conflict-free banking function.

Our Approach: We extend the GMP algorithm to find multi-access pattern memory partitioning scheme. We use block-cyclic version (without parameter α) of the banking function in our approach since the version with parameter α result in complex intra-bank offset functions without having a fixed format. Block-cyclic version results in a slightly increased number of banks compared to the version with parameter α. However, it simplifies the mapping functions to a fixed format so that we can implement them as configurable hardware functional units, namely *BAGU*. So, our banking function has the following format.

$$\beta(x) = \left\lfloor \frac{x}{B} \right\rfloor \%N$$

To obtain non-complex intra-bank offset function for multidimensional arrays, w_{d-1} (width of the $d-1$ dimension) should be an integral multiple of NxB. If not, $d-1^{th}$ dimension is padded with bits such that new dimension width is an integral multiple of NxB. Once padded, the intra-bank offset function becomes

$$O(x) = \lfloor \frac{x}{N.B} \rfloor .B + x\%B$$

Algorithm 1: Algorithm for calculating a conflict-free banking function.

Input: Access Patterns(P_1^R, ..., P_S^R)
Result: Valid Bank Function Parameters
for *B=1; B<B_{max}; B=2*B* **do**
  for *N=1; N<N_b; N=2*N* **do**
    j<-0
    for *s:1* $\rightarrow$ *S* **do**
      c = All stream combinations in P_s^R
      for *i: 0* $\rightarrow$ *c* **do**
        $P_{conf} = CreateConflictPolytope(N, B, \phi_x(), \phi_y())$
        $points = CountPoints(P_{conf})$
        if *points>0* **then**
          break;
        if $i = c - 1$ **then**
          j<-j+1
    if $j = n$ **then**
      N,B are valid bank parameters for all control step access patterns , return;

Our algorithm searches for the number of banks N and block size B considering multiple access patterns. Let us consider access patterns in two cycles accessing the same data array. Cycle 1 access pattern consists of two access streams $\phi_0(\vec{i})$ and $\phi_1(\vec{i})$. Cycle 2 access pattern consists of two access streams $\delta_0(\vec{i})$ and $\delta_1(\vec{i})$. Valid banking function $\beta(\vec{x})$, which works for both mutually exclusive access patterns should satisfy the following condition.

$$\forall \vec{i} \in M,\ \beta(\phi_0(\vec{i})) \neq \beta(\phi_1(\vec{i})) \text{ and } \beta(\delta_0(\vec{i})) \neq \beta(\delta_1(\vec{i}))$$

Algorithm 1 shows how conflict polytope-based test is used to check the above conditions. We extend this approach when there are multiple control step access patterns and access streams within one control step access pattern. Input for the algorithm is a set of mutually exclusive control step access patterns on an array, and the output is a set of valid bank function parameters.

5.5 CGRA Mapper

The CGRA Mapper maps AGI-Free DFG onto *CASCADE* PE array through modulo scheduling. Initially, we sort all the nodes in the DFG in topological order. We define minimum II as the maximum of resource minimum II and recurrence minimum II. In *CASCADE*, resource minimum II depends on the number of SUs and the number of load-store streams in the AGI-free DFG. CGRA Mapper attempts to find a valid mapping starting from the minimum II and increases II iteratively until it obtains a valid mapping. For each increased II value, we create time extended (II cycles) resource graph of the CGRA, which is known as MRRG. Note that MRRG includes both compute resources (PE ALUs) and configurable data paths, inside (ALU-register file links) and outside (PE-PE links) PEs. The goal of the mapping algorithm is to map all the computations and dependencies in the DFG to an MRRG with minimum II.

Problem Definition: Given an AGI-Free DFG $D = (V_D, E_D)$ where V_D consists of five types of nodes – load operations (V_D^L), child operations of load (V_D^{CL}), store operations (V_D^S), parent operations of store (V_D^{PS}) and compute operations (V_D^C) – the problem is to construct a minimally time-extended MRRG $H_{II} = (V_H, E_H)$ where V_H consists of four type of nodes: SUs (V_H^{SU}), SR-connected PEs (V_H^{SR}), normal PEs (V_H^F), and links (V_H^L – the configurable data paths inside and outside PEs that is time

extended), for which there exists a mapping $\phi = (\phi_V, \phi_E)$ from D to $H_{II} = (V_H, E_H)$ under set of constraints γ that restricts the mapping of V_D^L, V_D^S nodes to V_H^{SU} nodes, V_D^{CL}, V_D^{PS} nodes to V_H^{SR} nodes and V_D^C nodes to either V_H^{SR} or V_H^F nodes.

Place and Route Algorithm: We follow an iterative approach for mapping. The algorithm maps each node of the AGI-free DFG $u \in V_D$ to a node of MRRG $v^F \in \{V_H^F, V_H^{SU}, V_H^{SR}\}$ under constraint γ such that it utilizes the links $v^L \in V_H^L$, that results in the least accumulated cost, when routing data from the parent nodes of u. We employ *Dijkstra's* shortest path algorithm in establishing such routes and allow the links to be over-subscribed if necessary. Initially, all links are assigned a similar cost. At the end of one iteration of mapping, the algorithm increases the cost of the over-subscribed links $v^L \in V_H^L$ for future iterations (inspired by SPR [14]). The main intuition behind increasing the cost is to encourage the routing of data through alternative routing resources. When the mapping converges, the resources with most demand are likely to be used for mapping the dependencies with fewer options for routing compared to the competitors. In subsequent iterations, the placement of node $u \in V_D$ may change to avoid over-subscribed resources from the previous iteration. We deem the mapping of the DFG a success when none of the resources are over-subscribed.

5.6 Synergistic Banking and CGRA Mapping

Conflict-free memory allocation and CGRA mapping are heavily inter-dependent tasks as fixing one create constraints on the other and vice versa [23, 45]. Bank mapping algorithm attempts to create a conflict-free bank layout according to control step access patterns. If it fails to find a conflict-free bank layout, CGRA mapping has to be changed, and the bank mapping algorithm has to create a new conflict-free bank layout. It is an iterative process that continues till we find both conflict-free bank layout and valid mapping. For example, let us assume an access pattern with five load streams and a CGRA mapping with II=1 (all five streams access memory in one cycle). Further, assume that we cannot find a conflict-free memory layout with the given number of banks. In this case, we try to obtain new CGRA schedule and conflict-free bank layout by increasing the II to two and distributing five load accesses in two cycles. We continue increasing II and distributing memory accesses until we obtain conflict-free memory bank layout and valid CGRA schedule. This process is guaranteed to terminate at II=5 as there are no load conflicts when II=5 (because only one load access happens in one cycle).

Algorithm 2 presents synergistic memory banking and computation mapping on PE array. Initial resource minimum II of the CGRA mapping depends on the number of SUs available in *CASCADE* and number of load/store streams in the loop kernel in addition to CGRA fabric resources. This initial II is an input to the algorithm along with the AGI-free DFG and architectural parameters. Function $find\_bank\_parameters$ divides the number of accesses in the loop kernel access pattern (P^R) into II number of sets to form a new set of control step access patterns. Then it searches for a banking solution with these control step access patterns to find bank parameters with a minimum number of banks (N_{min}^a). This process is applied to each array in the loop kernel to obtain valid banking parameters. If the obtained control step access patterns are different from the control step access pattern in the existing CGRA mapping, the mapping algorithm finds new CGRA mapping by constraining the mapping with resultant control step access patterns. We place different arrays in different banks to avoid inter-array conflicts. Therefore, the algorithm adds N_{min}^a of all arrays to find total required number of banks (N_{req}). If the algorithm fails to find a valid mapping and valid bank parameters for all arrays with the available number of banks, it increases the II of the CGRA mapping. It is an iterative process that continues until it finds the conflict-free bank layout and valid mapping.

Algorithm 2: Synergistic Banking and CGRA Mapping.

Input: AGI-free DFG ($D = (V_D, E_D)$), CGRA architecture description ($C = (V_C, E_C)$), Data Arrays($A_1, ..A_n$), Loop Kernel Access Patterns($P^{R_1}, ..., P^{R_n}$), Number of Banks(N_b)
Result: Valid Bank and Computation Mapping $H_{II} = (V_H, E_H)$

$II = max(RecII(D), ResII(D, C))$
while *true* **do**
  N_{req} <- 0
  H_{II}, II = CGRA_Mapper(II, D, C)
  foreach $a \in (A_1, ..A_n)$ **do**
    for *all II control step combinations formed by* P^a **do**
      $N^a = find\_bank\_parameters(P^a_1, .., P^a_{II})$
      if $N^a < N^a_{min}$ **then**
        N^a_{min}<-N^a
        $P^{'}$ <- $(P^a_1, .., P^a_{II})$
    $Constrained\_Mapping(H_{II}, P^{'})$
    $N_{req} = N_{req} + N^a_{min}$
  if $N_b \geq N_{req}$ **then**
    return;
  $II = II + 1$

CASCADE does not support indirect addressing where array addresses depend on the data (e.g., A[B[i]]). To handle indirect addressing, it needs a separate datapath to route read data to address generation units. We leave this as future work because the majority of the loops amenable to CGRA acceleration do not include indirect addressing.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Comparative Baselines: We evaluate *CASCADE* against two different baselines *Generic*s [22] and *JLM-B* [45] that represent two contrasting approaches to memory bank conflict handling by the compiler. Both baselines are prototypical of conventional CGRA architectures and embed the address generation computations in the PE array. In contrast, *CASCADE* decouples the address generation operations into a separate *Access* component.

Generic: Data arrays are divided into bank-sized blocks and are placed consecutively in the memory banks. The CGRA mapping algorithm creates schedules where the memory access operations are carefully placed along the timeline to avoid memory bank conflicts [22]. This may result in longer schedules and higher II value. The address generation computations are performed in the CGRA.

JLM-B: The compiler places data in the memory banks in a conflict-free manner according to the banking functions proposed in Joint Loop Mapping (JLM) [45]. Authors of JLM do not specify where they compute banking functions. Therefore, we assume the PE array evaluates them.

Compiler Implementation: We implement *CASCADE* compiler as a pass in *LLVM 8.0* [24] that extracts the AGI-free DFG and memory access pattern of a target loop kernel. We use pragmas to annotate the target loop within a benchmark for acceleration. Extracted memory access pattern and AGI-free DFG are the input for our synergistic banking and CGRA mapping algorithm, and we implement them in *C++*. We utilize *Ehrhart* algorithm implementation provided in *Polylib*

Table 1. Benchmark loop kernels characteristics.

Benchmark	Benchmark Suite	Node with AGI	Nodes w/o AGI (and w/o LD/ST)	Loop depth	LD/ST streams	Conflicting array accesses	Memory footprint (KB)
gemm	Polybench	42	27 (16)	3	8,1	Y	12
2d_stencil	Polybench	42	18 (9)	2	5,1	Y	8
gesummv	Polybench	26	21 (17)	2	3,1	N	8
adi	Polybench	27	13 (7)	3	5,1	Y	12
dynprog	Polybench	22	17 (14)	3	2,1	Y	32
first_diff	Livermore	17	13 (10)	1	2,1	N	8
equa_of_SF	Livermore	48	24 (14)	1	9,1	Y	16
two_d_ehf	Livermore	52	28 (14)	2	12,2	Y	16
hydro	Livermore	22	8 (4)	1	3,1	Y	12
h264	Mediabench	45	31 (28)	2	2,1	N	0.8
jpeg	Mediabench	28	18 (16)	2	1,1	N	0.8
mpeg4	Mediabench	43	13 (11)	1	1,1	N	8

5.22.5 [2] library to count the number of integer points in conflict polytopes for conflict-free bank allocation. Synergistic banking and CGRA mapping algorithm generate CASCADE CGRA and SE configurations.

RTL Implementation: *CASCADE* introduces architectural modifications for the decoupled access-execute design, while baseline *Generic* and *JLM-B* use the conventional design. We implement both *CASCADE* (with 4x4 PE array and 8 SUs) and conventional CGRA (with 4x4 PE array) architectures in *Verilog HDL*. We then synthesize them onto a commercial 40 nm process using *Synopsys* Toolchain. Both architectures consist of 4 KB SPM with eight banks. Conventional CGRA architecture consists of 8 memory accessible PEs. We obtain the clock frequency, power, and area estimates from the RTL implementation. The maximum clock frequency achievable for both the baseline and *CASCADE* is 510 MHz at 40 nm process.

Performance Estimation: We use IIs of valid CGRA mappings as the main performance metric to compare the performance of conventional CGRAs and *CASCADE* as both designs have the same clock frequency. As the compiler guarantees conflict-free banking and valid mapping, the simulated execution of the CGRAs should produce the same performance results as compiler-generated II value. Nevertheless, we still performed functional and timing validation of *CASCADE* architecture through cycle-accurate software simulation to confirm the equivalence.

Benchmark Loop Kernels: We select representative loop kernels (Table 1) from *Livermore* [27], *Polybench* [3], and *Mediabench* [1] benchmark suites. Selected loop kernels have different memory access patterns with different stride depths. Loop kernels *gemm, 2d_stencil, adi, equation of sf, two_d_ehf, hydro*, and *fft* have conflicting array accesses. Table 1 lists the number of nodes in the DFG with AGIs and without AGIs for each loop kernel. The values inside brackets of the fourth column denote the number of nodes without AGI and memory instructions. The number of load/store streams, loop depths in the nested loops, and the memory footprint (the total size of the data arrays) are also listed. Note that entire data arrays do not need to reside in on-chip SPM at any point in time as the DMA engine transfers active data blocks between SPM and off-chip memory. AGIs account for more than 50% of the DFG nodes in four loop kernels (*2d_stencil, equation of sf, hydro,* and *mpeg4*). The rest have around 40% AGIs.

6.2 Experimental Results

***CASCADE* Memory Configurations:** We first explore the impact of the memory configuration, namely the number of banks and SUs on the performance of *CASCADE*. Figure 10 shows the achieved II for *two_d_ehf* benchmark running on *CASCADE* with 4x4 PE array but a different number of SUs and memory banks. The results show that both the number of memory banks and SUs are critical, and adding either will result in a lower II value.

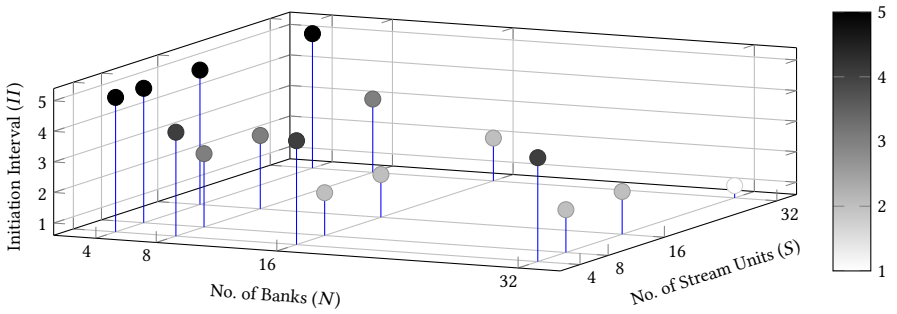


Fig. 10. Achieved II for different *CASCADE* configurations (w.r.t. number of banks and number of stream units) for *two_d_ehf* benchmark.

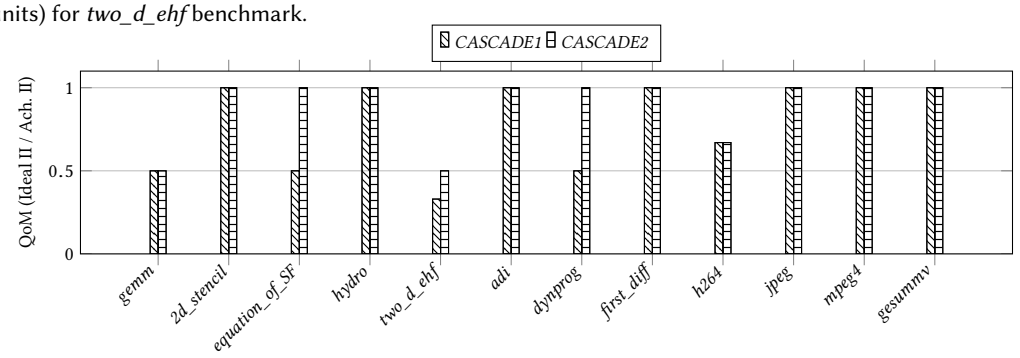


Fig. 11. QoM (Ideal II / Achieved II) of *CASCADE* versus Ideal decoupled CGRA with perfect memory.

Given the importance of memory parameters, we conduct the rest of the experiments with two different *CASCADE* configurations illustrative of different memory capabilities. *CASCADE1* has 4x4 PE array, 8 SUs, 8 load/store SR pairs, and 8 memory banks. In *CASCADE1*, two left-most CGRA PEs in each row connect to two load/store SR pairs. The remaining PEs access memory data through the PEs in the two leftmost columns. *CASCADE2* has 4x4 PE array with 16 SUs, 16 load/store SR pairs, and 16 memory banks. Thus, *CASCADE2* has a matching number of PEs and load/store SRs. Therefore, each CGRA PE in *CASCADE2* has its own load/store SR pair. Clearly, *CASCADE2* has far more powerful memory subsystem than *CASCADE1* even though the PE array design is the same in both configurations.

Comparison with Ideal CGRA: We define a metric, Quality of Mapping (QoM), as the ratio of the II of an ideal 4x4 decoupled CGRA that has infinite memory bandwidth and no memory conflicts, and the achieved II of *CASCADE1* or *CASCADE2*. The QoM being one implies that *CASCADE1* (or *CASCADE2*) unlocks the maximum performance. Figure 11 shows the QoM of *CASCADE1* and *CASCADE2*.

CASCADE2 achieves ideal II for all but three benchmarks. Benchmarks *gemm* and *h264* require very high utilization of the fabric to achieve the ideal II because they have 16 nodes (ideal II=1 with 100% utilization) and 28 nodes (ideal II=2 with 86% utilization), respectively. Moreover, the inability to route the dependencies in a restricted environment of high utilization makes these kernels perform poorly. For *two_d_ehf*, the compiler cannot find a non-conflicting solution with 16 banks for its 12 load access streams and fails to achieve the ideal II. Additionally, *CASCADE1* is unable to achieve the ideal II for *equation_of_SF* and *dynprog* because of insufficient SUs.

Comparison with 4x4 Baseline CGRA: Figures 12 and 13 show the area and power breakdown of *CASCADE* and baseline generic CGRA. The additional architectural components (SUs, crossbar

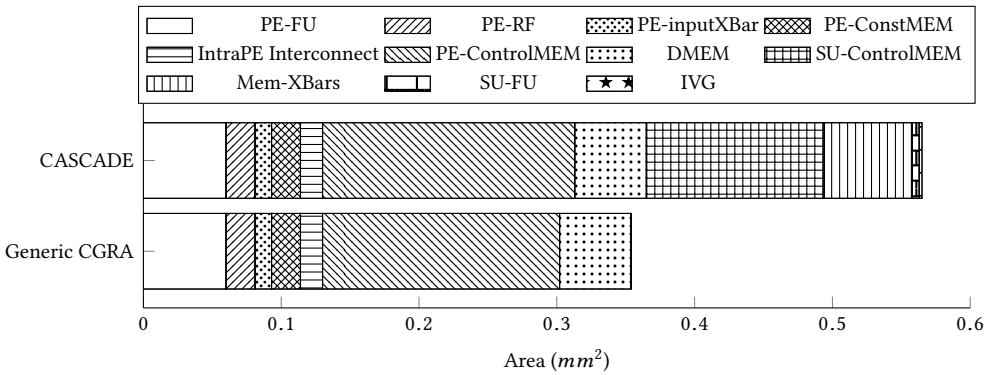


Fig. 12. Area breakdown of 4x4 *CASCADE* with 8 SUs and 4x4 baseline generic CGRA.

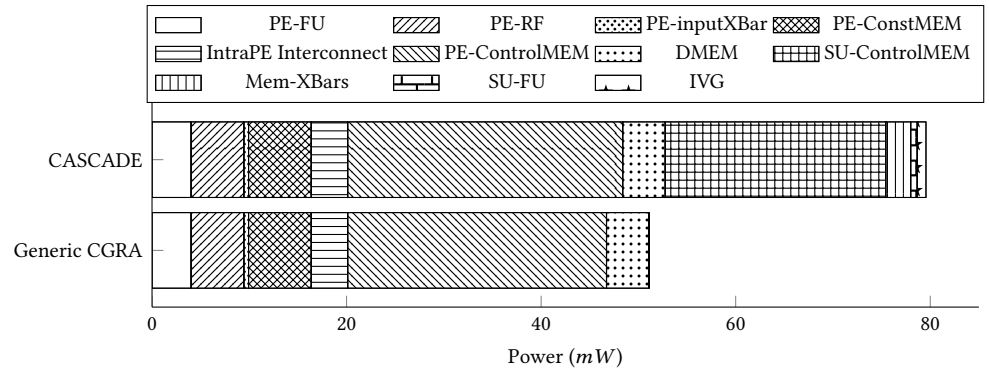


Fig. 13. Power breakdown of 4x4 *CASCADE* with 8 SUs and 4x4 baseline generic CGRA.

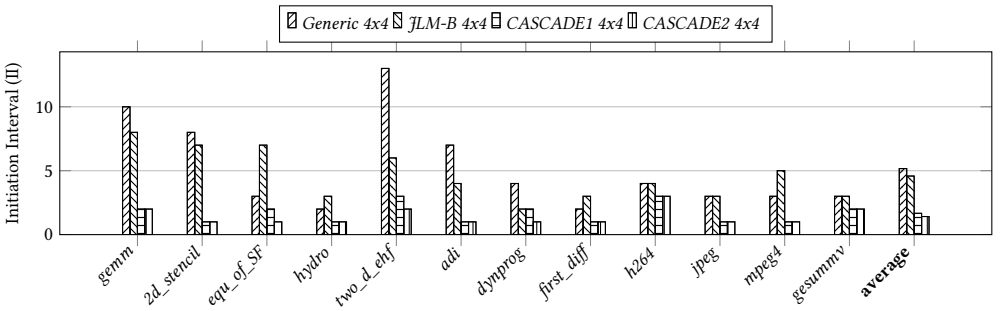
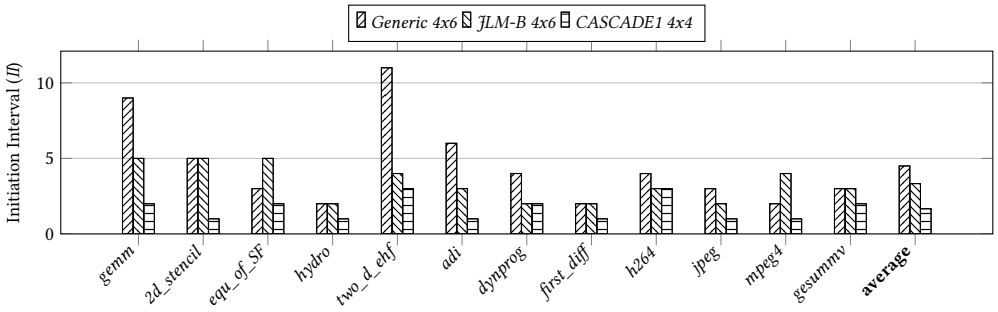


Fig. 14. *CASCADE* versus baseline 4x4 CGRA.

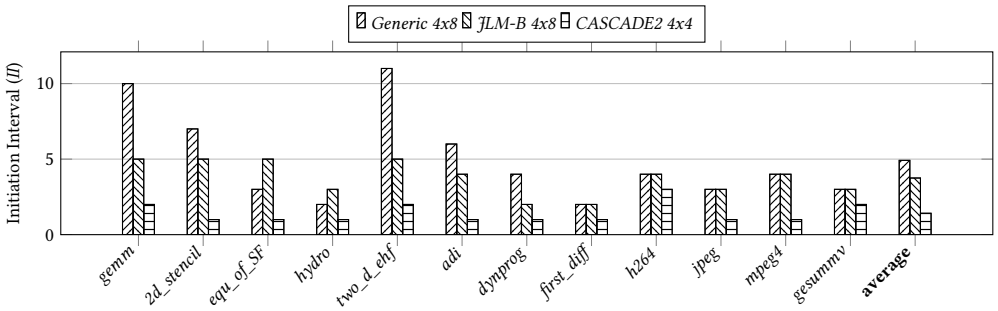
switches, and CGRA control memories) consumes 59% more area compared to the conventional CGRA. However, we note that AGIs corresponds to an average of 65% of all computation in a loop kernel. Figure 14 shows by offloading this AGI execution to the stream engine, *CASCADE1* achieves 3.4x and 3.2x better performance, whereas *CASCADE2* achieves 3.9x and 3.7x better performance compared to *Generic 4x4* and *JLM-B 4x4* approaches, respectively. *In other words, we pay a 0.6x area overhead to obtain more than 3x performance gain. We demonstrate later that* CASCADE1 *achieves 3x and 2.3x better performance whereas* CASCADE2 *achieves 3.7x and 3x better performance compared to* Generic *and* JLM-B *approaches on baseline CGRAs with the same area, i.e., bigger PE array.* The

Table 2. Iso-area baseline CGRAs corresponding to *CASCADE* configurations. The parenthesized values indicate area normalized to *CASCADE*.

CASCADE	*Generic*
CASCADE-1 PE:4x4 SU:8 (1.00)	4x6 (1.06)
CASCADE-2 PE:4x4 SU:16 (1.00)	4x8 (0.99)



(a) 8 Memory Accesses per Cycle



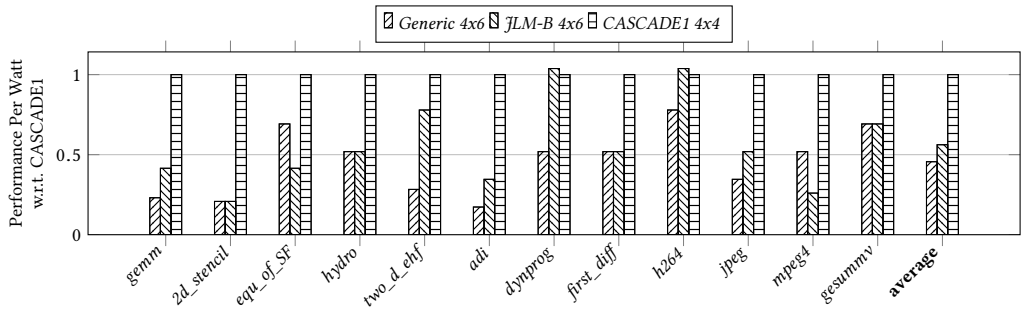
(b) 16 Memory Accesses per Cycle

Fig. 15. Performance of 4x4 *CASCADE* versus iso-area baseline CGRA with bigger PE arrays (4x6 and 4x8).

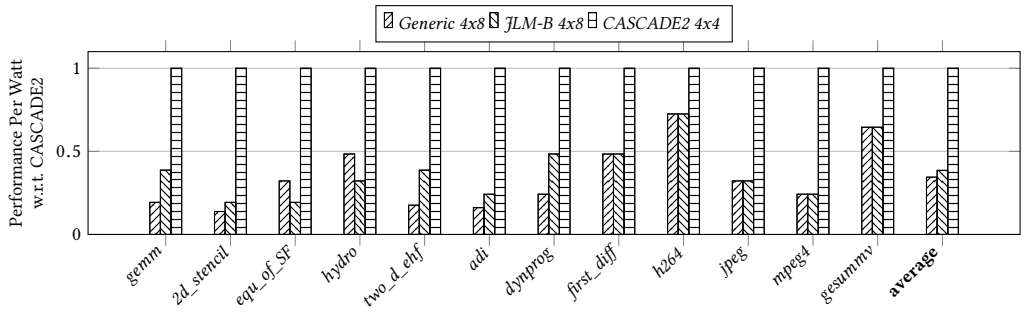
main contributors for the difference in silicon area of *CASCADE* concerning baseline CGRA are the SU control memories (36%), crossbar switches (18%), CGRA control memories (3%), and SU functional units (2%). The control memory of *CASCADE* is slightly larger than the baseline CGRA because each PE has two additional channels – the stream registers (SR_L, SR_S) – that require control information.

Comparison with Iso-Area Baselines: As *CASCADE* requires more area due to SE that is not present in conventional CGRAs, we create baseline CGRA architectures that have roughly the same area as each *CASCADE* configuration (Table 2) by increasing the PE array size. *CASCADE1* and *CASCADE2* are iso-area equivalent to 4x6 and 8x4 PE array in conventional CGRA, respectively. Moreover, to match the memory system, we pair 4x6 PE array and 8x4 PE array with 8 and 16 memory accessible PEs, respectively. Note that our baselines, namely *Generic* and *JLM-B*, have identical baseline CGRA architecture but differ in compiler approach to resolve memory access conflict.

Figures 15a and 15b show a performance comparison of *CASCADE1* and *CASCADE2* with corresponding iso-area baselines. *CASCADE* delivers better performance compared to both the baselines



(a) 8 Memory Accesses per Cycle



(b) 16 Memory Accesses per Cycle

Fig. 16. Performance per watt of 4x4 *CASCADE* versus iso-area baseline CGRA with bigger PE arrays (4x6 and 4x8).

for all the loop kernels. *CASCADE1* achieves 3x and 2.3x better performance whereas *CASCADE2* achieves 3.7x and 3x better performance compared to *Generic* and *JLM-B*, respectively, even though the latter have larger PE arrays.

It is important to note that some of the kernels do not require banking function for conflict-free memory accesses (i.e., the number of banks are sufficient to hold the arrays separately, and each array has single access per loop iteration). We identified the set of kernels (*first_diff*, *h264*, *jpeg*, *mpeg4*, and *gesummv*) that do not require banking function. In *CASCADE1*, this set of kernels perform 1.9x and 2.1x better compared to iso-area *Generic* and *JLM-B*, respectively. In *CASCADE2*, these perform 2.4x better compared to both *Generic* and *JLM-B*, respectively. Such performance improvement is primarily because of the offloading of AGI nodes to SUs.

In contrast, the rest of the kernels, that require banking function for conflict-free memory accesses are observed to provide superior performance because of the offloading of both AGI and banking functions to SUs. In *CASCADE1*, this set of kernels perform 3.5x and 2.5x better compared to iso-area *Generic* and *JLM-B*, respectively. In *CASCADE2*, they perform 4.6x and 3.4x better compared to iso-area *Generic* and *JLM-B*, respectively. Additionally, it is interesting to note that *hydro* and *equation_of_SF* that have conflicting array accesses, perform better in *Generic* over *JLM-B* because the overhead of banking computation nullifies the benefits of conflict-free data placement.

Figures 16a and 16b show the performance per watt comparison of *CASCADE1* and *CASCADE2* with the two baselines running on iso-area baseline CGRAs. Average power consumptions of *CASCADE1*, *CASCADE2*, 4x6, and 4x8 iso-area baseline CGRAs are 76 mW, 107 mW, 79 mW, and

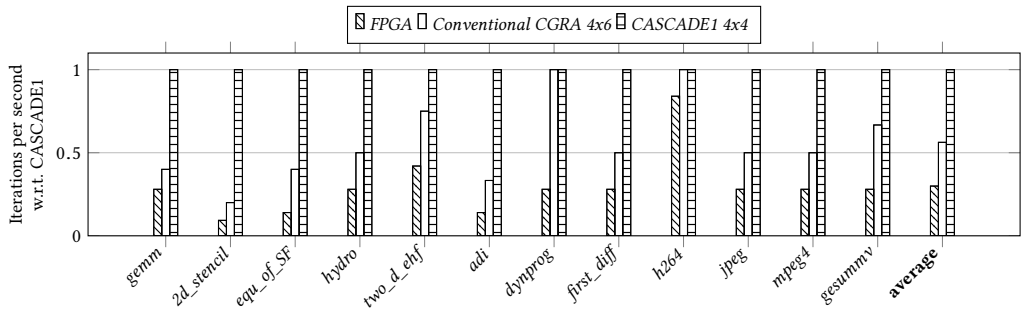


Fig. 17. Performance comparison of *CASCADE1* with FPGA and iso-area baseline CGRA.

110 mW, respectively. *CASCADE1* achieves 2.2X and 1.8X better performance per watt whereas *CASCADE2* achieves 2.9X and 2.6X better performance per watt compared to iso-area *Generic* and *JLM-B*, respectively.

Comparison with FPGA: We evaluate *CASCADE1* and iso-area baseline CGRA (4x6 PE array) against *Xilinx Zynq-7010* FPGA. *Vivado* High-Level Synthesis (HLS) tool estimates performance and power values for the FPGA. We use HLS array partitioning and choose cyclic/block partitioning factors to obtain the best performance for each benchmark. The FPGA runs at a maximum clock rate of 200 MHz, and power consumption is around 120 mW∼128 mW for all the benchmarks. Process technology used for *Xilinx Zynq-7010* FPGA and *CASCADE* is 28 nm and 40 nm, respectively. Thus, we scale *CASCADE* frequency by a factor of 1.42 for a fair comparison. So *CASCADE* and baseline CGRA scaled frequency is 714 MHz (scaled from 510 MHz at 40 nm to 28 nm).

We compare the performance (loop iteration per second) and performance per watt characteristics of FPGA, iso-area baseline CGRA, and *CASCADE1*. Figures 17 and 18 show the performance and performance per watt characteristics, respectively. On average, *CASCADE1* and iso-area baseline CGRA achieve 3.8x, 1.9x better performance and 7.5x, 3.5x better performance per watt compared to FPGA. We observe that in general, *CASCADE1* and FPGA achieve similar II values for most benchmarks, while baseline CGRA has higher II values. In FPGA, address generation logic and computation logic are synthesized onto fine-grained hardware resources as opposed to baseline CGRA where limited processing elements are utilized for address calculations wasting precious resources. The similarity of achieved II for FPGA and *CASCADE* shows the advantage of designing specialized data path for address generation. On the other hand, even with similar II value, the FPGA falls behind *CASCADE* and baseline CGRA in terms of both performance and performance per watt due to the overhead of fine-grained reconfigurability resulting in slower clock and higher power. These results show the advantage of coarse-grained reconfigurability of CGRAs compared to FPGAs.

Compilation Time: CGRA Mapping problem is NP-complete, and the complexity of the mapping increases with the number of nodes in the DFG. In baseline CGRAs, the compiler maps the DFG consisting of both address generation and computation. In contrast for *CASCADE*, the compiler maps AGI-free DFG consisting of only computation instructions. Therefore, the mapping for *CASCADE* is relatively simpler compared to baseline CGRA mapping. *CASCADE* achieves three orders of magnitude faster compilation time (Figure 19) compared to the two baselines.

7 CONCLUSION

This paper proposes a novel decoupled access-execute CGRA design called *CASCADE* with full architecture and compiler support for high-throughput data streaming from multi-bank memory.

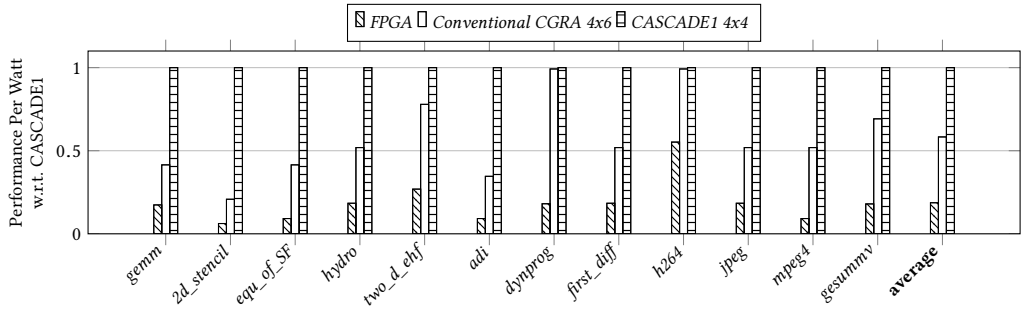


Fig. 18. Performance-per-watt comparison of *CASCADE1* FPGA and iso-area baseline CGRA.

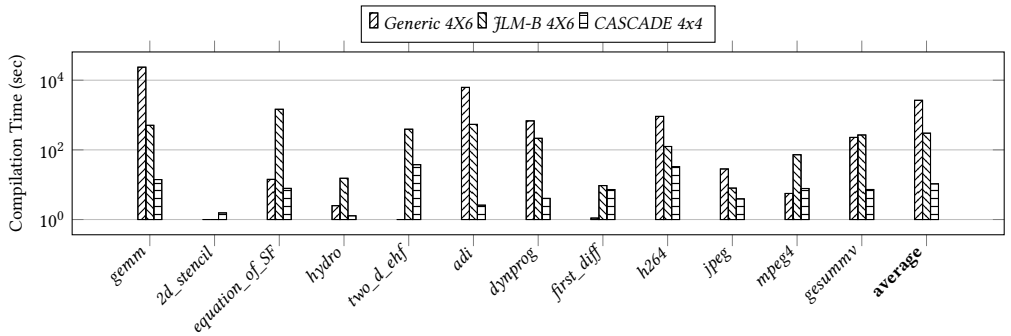


Fig. 19. Compilation time of *CASCADE* versus baselines.

CASCADE compiler offloads address generation computations to novel programmable dedicated hardware unit called stream engine. The compiler maps remaining actual computations to the PE array allowing the CGRA to perform at its full potential. Our evaluations show CASCADE accomplishes close to the ideal performance of a CGRA with perfect memory subsystem. It achieves on average 3x performance benefit and 2.2x performance per watt improvement w.r.t. iso-area conventional CGRA design while reducing the CGRA compilation time by three orders of magnitude as an additional benefit.

8 ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation, Prime Minister's Office, Singapore under Grant NRF2015-IIP003 and Huawei International Pte. Ltd.

REFERENCES

[1] 2019. MediaBench 2 Benchmark. http://mathstat.slu.edu/~fritts/mediabench/.
[2] 2019. PolyLib - A Library of Polyhedral Functions. http://icps.u-strasbg.fr/polylib/.
[3] 2019. The Polyhedral Benchmark Suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.
[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. Compilers: Principles, Techniques, and Tools Second Edition.
[5] George Charitopoulos, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios N Pnevmatikatos. 2018. A Decoupled Access-Execute Architecture for Reconfigurable Accelerators. In *Proceedings of the 15th International Conference on Computing Frontiers*. ACM, 244–247.
[6] Samit Chaudhuri and Asmus Hetzel. 2017. SAT-Based Compilation to a Non-VonNeumann Processor. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 675–682.

[7] Liang Chen and Tulika Mitra. 2014. Graph Minor Approach for Application Mapping on CGRAs. *Transactions on Reconfigurable Technology and Systems (TRETS)* 7, 3 (2014), 21.

[8] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The Reconfigurable Streaming Vector Processor (RSVPTM). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 141.

[9] Philippe Clauss and Vincent Loechner. 1998. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI signal processing systems for signal, image and video technology* 19, 2 (1998), 179–194.

[10] Emilio G Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[11] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. RAMP: Resource-Aware Mapping for CGRAs. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[12] Nasim Farahini, Ahmed Hemani, Hassan Sohofi, Syed MAH Jafri, Muhammad Adeel Tajammul, and Kolin Paul. 2014. Parallel Distributed Scalable Runtime Address Generation Scheme for a Coarse Grain Reconfigurable Computation and Storage Fabric. *Microprocessors and Microsystems* 38, 8 (2014), 788–802.

[13] Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen, Mathew Hall, Bain Syrowik, Tomasz Czajkowski, et al. 2014. Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis. In *12th International Conference on Embedded and Ubiquitous Computing*. IEEE, 120–129.

[14] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. SPR: An Architecture-Adaptive CGRA Mapping Tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 191–200.

[15] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2012. EPIMap: Using Epimorphism to Map Applications on CGRAs. In *DAC Design Automation Conference*. IEEE, 1280–1287.

[16] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2013. REGIMap: Register-Aware Application Mapping on Coarse-Grained Reconfigurable Architectures (CGRAs). In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 18.

[17] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. 2013. Power-Efficient Predication Techniques for Acceleration of Control Flow Execution on CGRA. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 2 (2013), 8.

[18] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient Execution of Memory Access Phases Using Dataflow Specialization. In *SIGARCH Computer Architecture News*, Vol. 43. ACM, 118–130.

[19] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A cgra with reconfigurable single-cycle multi-hop interconnect. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[20] Manupa Karunaratne, Cheng Tan, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2018. Dnestmap: mapping deeply-nested loops on ultra-low power CGRAs. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[21] Heba Khdr, Santiago Pagani, Ericles Sousa, Vahid Lari, Anuj Pathania, Frank Hannig, Muhammad Shafique, Jürgen Teich, and Jörg Henkel. 2016. Power Density-Aware Resource Management for Heterogeneous Tiled Multicores. *Transactions on Computers (TC)* 66, 3 (2016), 488–501.

[22] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. 2010. Operation and Data Mapping for CGRAs with Multi-bank Memory. In *ACM Sigplan Notices*, Vol. 45. ACM, 17–26.

[23] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. 2011. Memory Access Optimization in Compilation for Coarse-Grained Reconfigurable Architectures. *Transactions on Design Automation of Electronic Systems (TODAES)* 16, 4 (2011), 42.

[24] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[25] Jongeun Lee, Seongseok Seo, Hongsik Lee, and Hyeon Uk Sim. 2014. Flattening-Based Mapping of Imperfect Loop Nests for CGRAs. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 9.

[26] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. 2013. Polyhedral Model Based Mapping Optimization of Loop Nests for CGRAs. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 19.

[27] Frank H McMahon. 1986. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*. Technical Report. Lawrence Livermore National Lab., CA (USA).

[28] Bingfeng Mei, M Berekovic, and JY Mignolet. 2007. ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors. In *Fine-and coarse-grain reconfigurable computing*. Springer, 255–297.

[29] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2002. DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures. In *International Conference on Field-Programmable Technology, 2002 (FPT). Proceedings*. IEEE, 166–173.

[30] Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. 2015. Efficient Memory Partitioning for Parallel Data Access in Multidimensional Arrays. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 160.

[31] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–429.

[32] Sai Manoj PD, Jie Lin, Shikai Zhu, Yingying Yin, Xu Liu, Xiwei Huang, Chongshen Song, Wenqi Zhang, Mei Yan, Zhiyi Yu, et al. 2017. A Scalable Network-on-Chip Microprocessor with 2.5 D Integrated Memory and Accelerator. *Transactions on Circuits and Systems I: Regular Papers* 64, 6 (2017), 1432–1443.

[33] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. 2016. System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 3 (2016), 435–448.

[34] B Ramakrishna Rau. 1994. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of MICRO-27. The 27th Annual International Symposium on Microarchitecture*. IEEE, 63–74.

[35] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. 2000. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *Transactions on computers* 49, 5 (2000), 465–481.

[36] James E Smith. 1982. Decoupled Access/Execute Computer Architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 10. IEEE Computer Society Press, 112–119.

[37] Yuxin Wang, Peng Li, and Jason Cong. 2014. Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis. In *Proceedings of the international symposium on Field-programmable gate arrays*. ACM, 199–208.

[38] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 12.

[39] Dongjun Xu, Ningmei Yu, PD Sai Manoj, Kanwen Wang, Hao Yu, and Mingbin Yu. 2015. A 2.5-D Memory-Logic Integration with Data-Pattern-Aware Memory Controller. *Design & Test* 32, 4 (2015), 1–10.

[40] Yanqin Yang, Meng Wang, Haijin Yan, Zili Shao, and Minyi Guo. 2010. Dynamic Scratch-Pad Memory Management with Data Pipelining for Embedded Systems. *Concurrency and Computation: Practice and Experience* 22, 13 (2010), 1874–1892.

[41] Shouyi Yin, Zhicong Xie, Chenyue Meng, Leibo Liu, and Shaojun Wei. 2016. Multibank Memory Optimization for Parallel Data Access in Multiple Data Arrays. In *International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[42] Shouyi Yin, Zhicong Xie, Chenyue Meng, Peng Ouyang, Leibo Liu, and Shaojun Wei. 2017. Memory Partitioning for Parallel Multipattern Data Access in Multiple Data Arrays. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 2 (2017), 431–444.

[43] Shouyi Yin, Xianqing Yao, Dajiang Liu, Leibo Liu, and Shaojun Wei. 2015. Memory-Aware Loop Mapping on Coarse-Grained Reconfigurable Architectures. *Transactions on Very Large Scale Integration (VLSI) Systems* 24, 5 (2015), 1895–1908.

[44] Shouyi Yin, Xianqing Yao, Tianyi Lu, Dajiang Liu, Jiangyuan Gu, Leibo Liu, and Shaojun Wei. 2017. Conflict-Free Loop Mapping for Coarse-Grained Reconfigurable Architecture with Multi-Bank Memory. *Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2471–2485.

[45] Shouyi Yin, Xianqing Yao, Tianyi Lu, Leibo Liu, and Shaojun Wei. 2016. Joint Loop Mapping and Data Placement for Coarse-Grained Reconfigurable Architecture with Multi-Bank Memory. In *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 127.