Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA

Dhananjaya Wijerathne[1], Zhaoying Li[1], Manupa Karunaratne[2], Li-Shiuan Peh[1], Tulika Mitra[1]
[1]*School of Computing, National University of Singapore,* [2]*Advanced Micro Devices, Inc*
[1]{dmd, zhaoying, peh, tulika}@comp.nus.edu.sg, [2]Manupa.Karunaratne@amd.com

***Abstract*—This paper presents Morpher, an open-source end-to-end compilation and simulation framework, to assist design space exploration and application-level developments of CGRA-based systems. Morpher can take an application with a compute-intensive kernel as input, compile the kernel onto a user-provided CGRA architecture, and automatically validate the compiled kernels through cycle-accurate simulation using test data extracted from the application. Morpher can handle real-world application kernels without being limited to simple toy kernels through its feature-rich compiler. Morpher architecture description language lets users easily specify architectural features such as complex interconnects, multi-hop routing, and memory organizations. In the experimental study, we evaluate Morpher against the state-of-the-art and demonstrate Morpher's ability to provide end-to-end compilation, simulation, and validation. Morpher is available online at https://github.com/ecolab-nus/morpher.**

I. Introduction

Coarse-Grained Reconfigurable Arrays (CGRAs) have emerged as promising reconfigurable accelerators, offering superior power efficiency compared to the FPGAs by maintaining reconfigurability at the word/instruction level. CGRAs appear commercially in Samsung Exynos 7420 SoC [1], Intel Configurable Spatial Accelerator [2], Sambanova RDU [3], Renesas Configurable Processor [4], and academic ones like HyCUBE [5], [6] among others. The simplest CGRA architecture has a set of processing elements (PE) interconnected in a grid [7]. Each PE comprises a register file, ALU, and a control memory to store instructions that are executed in a time-multiplexed nature. CGRAs are statically scheduled; thus, they do not need hardware structures for conflict resolution and synchronization, which makes them lightweight. In recent years, various CGRA architectures [5], [8]–[10] have been proposed with performance-enhancing features.

The recent success of the FPGAs can be primarily attributed to the quality design tools such as Vivado for high-level synthesis and open-source VTR [11] for FPGA architecture exploration and CAD research. In contrast, open-source CGRA design and exploration tools are in their infancy. Table I lists differentiating features of existing open-source CGRA design frameworks [12]–[15]. CGRA-ME [12] is a compilation and RTL generation framework for classic CGRA architectures. However, it only supports simple kernels without control divergence (if-then-else constructs) within the loop body and has limited support for loops with recurrence edges (inter-iteration dependencies). Most loop kernels on real-world applications that could be accelerated on a CGRA have some form of conditional inside the loop body [16], [17]. Additionally, conditional instructions are inserted into the loop body by loop flattening; an important optimization used to reduce the invocation overhead of nested loops [18]. Therefore, the CGRA-ME framework's applicability in practical applications is constrained by its limited support for control divergence and recurrence edges. Additionally, CGRA-ME avoids memory modeling and covers only the basic architectural features of the CGRA PE array. Furthermore, no open-source tool is available for simulating the compilation result of CGRA-ME.

TABLE I
Comparison with open-source CGRA design frameworks

Features		CGRA-ME	Pillars	OpenCGRA	CCF	Morpher
DFG Generation	Models control divergence	✗	✗	✓	✓	✓
	Recurrence edges	✗	✗	✓	✓	✓
Architecture Modeling	Adapt user defined architectures	✓	✓	✓	✗	✓
	Multi-hop connections	✗	✗	✗	✗	✓
	Different memory organizations	✗	✗	✓	✗	✓
P&R Mapper	Architecture adaptive mapping	✓	✓	✗	✗	✓
	Data layout aware mapping	✗	✗	✗	✗	✓
	Recurrence aware mapping	✗	✗	✓	✓	✓
Simulation & validation	Cycle accurate simulation	✗	✓	✓	✓	✓
	Test data generation	✗	✗	✗	✗	✓
	Validation against test data	✗	✗	✗	✗	✓

A design space exploration framework for CGRA called Pillars [13] supports automatic RTL code generation and cycle-accurate simulation based on scala-based architectural description language. However, as Pillars employs the CGRA-ME framework for front-end compilation, it also suffers from most of the drawbacks of CGRA-ME. The two other prominent CGRA frameworks, OpenCGRA [14] and CCF [15] support kernels with control divergence and recurrence edges. However, their mappers are not architecture adaptive, necessitating code-level modifications to target new CGRA architectures. Except for CGRA-ME, all other tools support cycle-accurate simulation, but only with the user's assistance in creating separate test benches with test data. None supports automatic test data generation or validation of CGRA execution against test data extracted from the application.

We propose Morpher, an open-source, fully automated end-to-end CGRA compilation and simulation framework, to realize fully functional CGRA designs. Morpher supports kernels with control divergence and recurrence edges allowing the user

M. Karunaratne was with National University of Singapore when this research was conducted. D. Wijerathne, Zhaoying Li, and M. Karunaratne have contributed equally to this work.

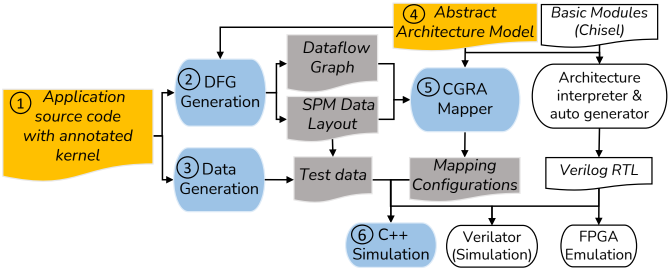
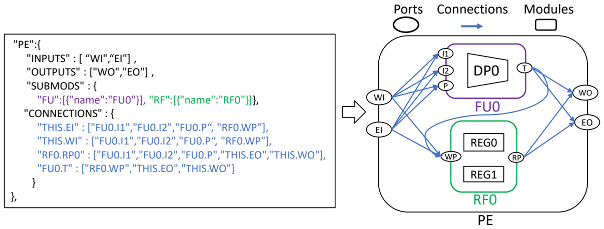


Fig. 1. Overview of Morpher Framework



```
"PE":{
    "INPUTS" : [ “WI",“EI"] ,
    "OUTPUTS" : ["WO","EO"] ,
    "SUBMODS" : {
      "FU":[{"name":"FU0"}], "RF":[{"name":"RF0"}]},
   "CONNECTIONS" : {
      "THIS.EI" : ["FU0.I1","FU0.I2","FU0.P“, "RF0.WP”],
      "THIS.WI" : ["FU0.I1","FU0.I2","FU0.P“, "RF0.WP"],
      "RF0.RP0" : ["FU0.I1","FU0.I2","FU0.P","THIS.EO","THIS.WO"],
      "FU0.T" : ["RF0.WP","THIS.EO","THIS.WO"]
     }
},
```

Fig. 2. An example of two input two output Processing Element model written in Morpher ADL. The definition of internal connections of primitive (RF, FU) modules are not shown.

to compile real application kernels without being restricted to simple kernels. It allows users to define complex CGRA architectures with various memory organizations and interconnects through a flexible architecture specification language. Morpher can compile complex kernels more quickly thanks to efficient mapping algorithms. Furthermore, Morpher validates the compilation result by running simulations with test data automatically extracted from the target application.

II. Morpher

A. Overview

Fig. 1 illustrates the overall Morpher framework, including its current status and future development plan. The shaded portions represent implemented functionality, whereas unshaded portions represent upcoming planned integration. The pieces of the framework are numbered for easy reference. The framework has two inputs: application source code with annotated kernel ①, and the abstract architecture model ④. The main components of the framework are Data-Flow Graph (DFG) generation ②, test data generation ③, CGRA Mapper ⑤, and C++ Simulator ⑥.

CGRAs target loop kernels where the application spends a significant fraction of the execution time. The DFG generator is an LLVM-based pass that extracts the DFG of the target loop annotated in the application source code. Additionally, it constructs the multi-bank data layout by allocating kernel variables onto the memories of the target CGRA. The test data generator creates the data required for simulation and verification.

The CGRA mapper maps the extracted DFG onto the CGRA fabric to maximize parallelism by exploiting intra- and inter-iteration parallelism with software pipelining (i.e., modulo scheduling) [19]. Morpher supports a rich set of primitive constructs that not only model functional units and register files but also complex software-defined routers and multi-banked memories accessible via shared bus interfaces. The architecture is modeled as a time-extended resource graph called Modulo Routing Resource Graph (MRRG) [20], [21]where the nodes of the DFG are mapped to the time-space resource instances to maximize throughput. The resultant mapping configuration file describes the configuration for each resource cycle-by-cycle. Finally, the simulator uses the mapping configurations and the test data to simulate the execution of the application on the specified architecture.

Unshaded portions in Fig. 1 are from Pillars framework [13], which we intend to integrate into Morpher. We believe this integration will work well because while Pillars offers a full hardware-centric feature set such as automatic RTL generation, Verilator-based simulation, and FPGA emulation, Morpher offers a comprehensive front-end compiler toolchain.

B. DFG and Data Layout Generation

CGRA designs employ a variety of execution paradigms (control flow handling techniques, address generation models, memory models), and the compiler should generate a DFG to fit into the execution paradigm of the target CGRA. Morpher DFG generator currently supports DFG generation for multiple execution paradigms, including different control flow handling techniques (partial predication, full predication, dual-issue [16]), load-store address generation models (on array address generation, decoupled access execute/stream dataflow [22]), and memory models (memory-mapped slave).

Morpher creates the multi-bank data layout by allocating the live-in/live-out variables (scalar and arrays) in the memory banks. It currently supports simple data placement policies to distribute data uniformly on multiple banks based on data size or the number of variables. The base addresses of array variables and addresses of scalar variables are recorded in the data layout file. The address information is also embedded in the corresponding DFG nodes as constants.

C. Test Data Generation

Morpher automatically instruments, i.e., inserts data recording functions, to the application C source code. The recording functions can capture live-in and live-out variables of the target loop kernel. These recording functions are automatically inserted into the basic blocks that enter and exit the target loop kernel. The instrumented C program is then executed in a general purpose processor to record the live-in/live-out variables as test data. These test data are passed to the simulator, which uses live-in variables as initial data and the live-out variables as the expected result for verification.

D. Abstract Architecture Specifications

Morpher abstract architecture description language (ADL) is designed for flexibility to cover diverse CGRA architectures. It comprises three main components: Modules, Ports, and Connections. Modules model hardware blocks such as PEs, RFs, ALUs, LSUs, and Memories. Ports establish connections between modules that carry data between producers and consumers. Connections describe the connectivity among ports. Fig. 2 shows an example of a processing element with a FU, RF with two registers, two inputs, and two outputs modeled in Morpher ADL.

Generally, CGRAs have a hierarchical module structure; thus, a module could contain a list of sub-modules. There are three primitive modules in Morpher ADL, Functional Units (FU), Register Files (RFs), and Memory Units (MU) which could collectively model any number of custom modules as required. Multiplexers are inferred through the connections rather than being explicitly modeled.

FU can model hardware ALUs (Arithmetic logic unit) and LSUs (Load store unit). FU is composed of an internal structure called Data Path (DP), a list of supported operations, and operation latency. DP eventually gets mapped with a DFG node. RF models register files with a configurable number of registers and read/write ports. RF is composed of an internal structure called REGs that models registers. REGs could also be used to model scattered registers inside any module. MU models memories in detail: bank sizes (bit width and depth), number of read/write ports and allocated variables.

Morpher ADL provides a special syntactic sugar to automatically connect the PEs and MUs according to a given interconnection pattern eliminating the need for the user to specify all connections between PEs/MUs to populate CGRA.

E. CGRA Mapper

The CGRA Mapper takes the DFG and the abstract architecture description file and generates mapping configurations. Initiation Interval (II), the cycle difference between the initiation of two consecutive loop iterations, determines the throughput of the kernel. Therefore the goal of the mapper is to generate a mapping with the lowest possible II. The lower bound of II, i.e., Minimum II (MII), is derived based on the CGRA resource availability and the recurrence dependencies in the kernel [19]. The mapper attempts to map the loop starting with II set to MII and iteratively increments II by one until a feasible schedule is obtained.

Initially, Morpher analyses the connectivity between MUs and FUs. MUs are already allocated with the variables. Therefore some variables could only be accessed by some FUs. All FUs are then annotated with accessible variable names (as possible candidates for memory operation placements). Thereafter it sorts the nodes of the DFG in a topological ordering to create a scheduling list. The subsets of nodes that belong to recurrence cycles are prioritized according to the size of the cycle [23]. Each node of the scheduling list is mapped to a space-time instance of the supported FU node of MRRG such that it utilizes the ports that result in the least accumulated cost when routing data from the parent nodes of the current mapping node. We employ Dijkstra's shortest path algorithm in establishing such routes and allow the ports to be over-subscribed if necessary.

Morpher iteratively resolves over-subscriptions after obtaining the initial mapping. Morpher currently supports three approaches to resolve over subscriptions: adaptive-heuristic-based approach inspired by SPR [24], Simulated Annealing (SA) based approach [25], and learning-induced approach (LISA [26]). In the heuristic-based approach, the cost of the over-subscribed ports is increased for the next mapping iterations. When the mapping converges, the resources with the most demand are more likely to be used for mapping the dependencies with fewer options for routing compared to the competitors. In SA based approach, the node placement is changed based on an SA-based cooling schedule. With LISA, the node placement is guided by labels inferred from a trained Graph Neural Network (GNN) model. We deem the mapping a success where none of the resources are over-subscribed. Note that our modular code base makes it easy for researchers to add their mapping methods to the Morpher toolchain. In the future, we plan to incorporate hierarchical mapping approaches for better scalability [27]–[29] and include automated design space exploration of heterogeneous CGRA architectures [30].

F. Simulation and Verification

Morpher simulator is a model of the CGRA composed of functional units, registers, multiplexers, and memories. Currently, the Morpher simulator only models the variations of HyCUBE CGRA architecture [5], [6]. The CGRA model acts as a memory-mapped slave device to a host processor. First, the live-in variables recorded from the test data generator are loaded for each memory unit. Then the simulator executes operations mapped on the FUs, multiplexes the data, and writes the data to registers on a cycle basis following the mapping configurations. The post-simulation memory content is validated against the expected results. Once integrated with the Pillars framework, Morpher would be able to support RTL generation and FPGA emulation in addition to architecture adaptive simulation.

G. Open-source Artifact

Morpher open-source artifact contains the entire framework built in C++. Morpher is available online at https://github.com/ecolab-nus/morpher. A single python script invokes the tool, and multiple verified test cases are provided for reference. Furthermore, the open-source repository contains Continuous Integration (CI) workflows for automatically running functional tests in each code update to assure error-free code.

III. Experimental Study

This study aims to demonstrate key features of the Morpher framework: 1) Quality and faster compilation, 2) the ability to map real applications kernels with automated verification, and 3) the ability to model diverse CGRA architectures.

A. Mapping Quality and Compilation time

We compare Morpher's mapping results (II and compilation time) with those of CGRA-ME, the only open-source framework available for architecture-adaptive compilation. We map the kernels from polybench benchmark suite [32] to a 4x4 CGRA with 2-D mesh interconnection topology. Each PE has RFs of 4 registers, and all PEs have access to the memory. For a fair comparison, both tools take similar DFGs as input. Table III shows the results. All three mapping methods (heuristic, SA, and LISA) in Morpher achieve the

TABLE II
A SUMMARY OF 4X4 CGRA ARCHITECTURE EXPLORATION CASE STUDIES CONDUCTED USING MORPHER

		adpcm		aes		gemm_nt		disparity		dct		texture_syn		fft		nw		gsm	
MII		7		10		6		3		9		3		4		15		6	
Architecture Casestudies		**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**	**II**	**Compile Time (m)**
Generic CGRA [31]		17	323	24	410	14	87	26	380	23	367	12	135	11	467	19	37	10	409
ALU-Independent Routing [8], [10]		9	14	19	79	11	9	15	43	18	22	5	1	5	16	19	3	8	88
Multi-Hop Routing [5]	2-Hops	9	8	15	17	9	5	12	1	14	12	5	2	5	2	15	2	8	96
	3-Hops	9	4	13	5	8	3	10	2	13	8	5	2	5	10	15	3	7	39
	4-Hops	8	1	13	3	7	2	11	0.3	13	4	6	12	5	10	15	2	10	175
Multi-Hop Scattered Registers	4-Hops	8	1	12	2	7	2	11	0.3	13	3	6	10	5	7	15	2	7	18

TABLE III
MAPPING QUALITY AND COMPILATION TIME COMPARISON

Benchmark	DFG Nodes	MII	Achieved II		Compilation time (s)	
			CGRA-ME	Morpher	CGRA-ME	Morpher
2mm	21	2	2	2	96	13
atax	18	2	2	2	79	19
bicg	28	2	5	2	1540	125
cholesky	14	1	1	1	14	13
doitgen	26	2	3	2	264	14
gemm	23	2	3	2	280	10
gemver	26	2	3	2	404	22
gesummv	30	2	5	2	1115	54
mvt	18	2	2	2	81	10
symm	22	2	2	2	95	10
syrk	18	2	2	2	109	11
trmm	22	2	2	2	109	11
Average	22.17	1.92	2.67	1.92	348.83	26

minimum possible II for all the DFGs, while CGRA-ME failed to do so for five kernels. Furthermore, the Morpher LISA mapping approach [26] is the fastest, compiling 13.3x faster than CGRA-ME on average.

B. Accelerating real applications on CGRA

This study demonstrates how Morpher can be used to accelerate real applications on CGRA-based systems. It also highlights the importance of supporting control divergence and recurrence edges. The target application, Microspeech (wake-word detection), selected from the tinyML benchmark suite [33], uses deep neural network (DNN). The target CGRA is 4x4 HyCUBE with a partial predication-based execution model [5]. Microspeech application consists of a neural network with a CONV and fully connected layers. The CONV layer is accelerated on HyCUBE by lowering it to the GEMM kernel [34]. Two variations of the GEMM kernel, unrolled (GEMM-U) and unrolled-flattened (GEMM-U-F), are shown in Listing 1.

TABLE IV
EXECUTION TIME COMPARISON OF MICROSPEECH CONV LAYER ON 4X4 HYCUBE CGRA

Kernel	Nodes	II (MII)	Compute time (ms)	Data transfer time (ms)	Total execution time (ms)*
GEMM	26	4 (4)	2.70	3.39	6.09
GEMM-U	58	6 (4)	1.17	3.39	4.56
GEMM-U-F	79	8 (8)	1.31	1.79	3.10

* Execution times are calculated at 488 MHz CGRA frequency and 50 MBps host to CGRA data transfer rate. The execution time is the elapsed time between the host CPU invoking the CONV layer and the results being written back into the host memory.

```
for (i=0;i<R1; i++)
 for (j=0;j<C2; j++)
  for (k=0;k<C1; k=k+4): //map this (GEMM-U)
   O[i][j] += W[i][k]* I[k][j]+ W[i][k+1]* I[k+1][j]
   + W[i][k+2]* I[k+2][j]+W[i][k+3]* I[k+3][j];

for (n=0;n<R1*C2*C1; n++){: //map this (GEMM-U-F)
   O[i][j] += W[i][k]* I[k][j]+ W[i][k+1]* I[k+1][j]
   + W[i][k+2]* I[k+2][j]+W[i][k+3]* I[k+3][j];k+=4;
   if(k+1>=C1) {k=0; ++j;} if(j==C2) {j=0; ++i;}}
```

Listing 1. Two versions of GEMM kernel

Table IV shows the total execution time of the CONV layer with those GEMM versions. GEMM-U kernel reduces the compute time by nearly half compared to the original GEMM kernel due to high MAC utilization in unrolled mapping. The flattened kernel does not have invocation overheads present in the other two kernels [18]. Therefore, GEMM-U-F offers the best total execution time compared to other kernels. Note that loop flattening removes the inner loops by introducing conditional statements inside the loop body. It also adds a longer recurrence edge to the loop body [18]. The Morpher successfully compiles the kernels because it handles control divergence and recurrence edges. Furthermore, it automatically verifies the compiled kernels by running simulations using pre-processed audio data extracted from the application.

C. Modeling complex CGRAs

We evaluated the various features that are modeled by Morpher using a 4x4 CGRA instance, as summarized in Table II. We describe a set of architectures in the first column that introduce richer interconnection resources to the baseline (e.g., introducing switches, multi-hop links, etc). The trend shows that Morpher utilizes such resources to improve II towards MII while reducing compilation time. Note that MII is independent of interconnection resources and is only dependent on the kernel and PE array size. ALU-independent routing architectures [8], [10] use configurable switches to perform data routing as opposed to generic CGRA, which routes data through ALU [31]. It can bring about 66% throughput improvement. Furthermore, single-cycle multi-hop connections [5] allows 71% throughput improvement (at 4-hop point). Using scattered registers in the wires (also known as pipeline registers) instead of register files can give a clear area and power advantage. Compilation results show they provide nearly identical performance (with a similar number of registers), highlighting the benefit of employing scattered registers.

IV. CONCLUSION

Morpher presents a unified compiler and simulation framework. Morpher has the ability to model modern CGRA architectures, map complex workloads with a higher mapping quality at a shorter compilation time, and automatically verify the mapping results through cycle-accurate simulation.

V. ACKNOWLEDGMENT

This work was partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003 and Singapore Ministry of Education Academic Research Fund T1 251RES1905.

References

[1] 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.

[2] K. E. Fleming, K. D. Glossop, S. C. Steely Jr, J. Tang, A. G. Gara *et al.*, "Processors, methods, and systems with a configurable spatial accelerator," Feb. 11 2020, uS Patent 10,558,575.

[3] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021.

[4] "Renesas Configurable Processor." [Online]. Available: https://www.renesas.com/sg/en

[5] 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.

[6] 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.

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

[8] N. Farahini, S. Li, M. A. Tajammul, M. A. Shami, G. Chen, A. Hemani, and W. Ye, "39.9 gops/watt multi-mode CGRA accelerator for a multi-standard basestation," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2013, pp. 1448–1451.

[9] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[10] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[11] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: architecture and CAD for FPGAs from verilog to routing," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012, pp. 77–86.

[12] 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.

[13] Y. Guo and G. Luo, "Pillars: An Integrated CGRA Design Framework," in *Third Workshop on Open-Source EDA Technology (WOSET)*, 2020.

[14] C. Tan, N. B. Agostini, J. Zhang, M. Minutoli, V. G. Castellana, C. Xie, T. Geng, A. Li, K. Barker, and A. Tumeo, "OpenCGRA: Democratizing Coarse-Grained Reconfigurable Arrays," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2021, pp. 149–155.

[15] S. Dave and A. Shrivastava, "Ccf: A CGRA compilation framework," 2017.

[16] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-aware loop mapping on CGRAs," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[17] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[18] J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for CGRAs," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, 2014, pp. 1–10.

[19] B. R. Rau, "Iterative modulo scheduling," *International Journal of Parallel Programming*, vol. 24, no. 1, pp. 3–64, 1996.

[20] 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.

[21] 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.

[22] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "CASCADE: High throughput data streaming via decoupled access-execute CGRA," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–26, 2019.

[23] T. Oh, B. Egger, H. Park, and S. Mahlke, "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2009, pp. 21–30.

[24] 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.

[25] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[26] Z. Li, D. Wu, D. Wijerathne, and T. Mitra, "LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 444–459.

[27] 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.

[28] D. Wijerathne, Z. Li, T. K. Bandara, and T. Mitra, "PANORAMA: Divide-and-Conquer Approach for Mapping Complex Loop Kernels on CGRA," 2022.

[29] 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.

[30] K. B. Thilini, D. Wijerathne, T. Mitra, and L.-S. Peh, "REVAMP: A Systematic Framework for Heterogeneous CGRA Realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[31] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[32] J. Karimov, T. Rabl, and V. Markl, "Polybench: The first benchmark for polystores," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2018, pp. 24–41.

[33] P. Warden and D. Situnayake, *TinyML*. O'Reilly Media, Incorporated, 2019.

[34] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-performance low-memory lowering: GEMM-based algorithms for DNN convolution," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 99–106.