Accelerating Edge AI with Morpher: An Integrated Design, Compilation and Simulation Framework for CGRAs

***Abstract*—Coarse-Grained Reconfigurable Arrays (CGRAs) hold great promise as power-efficient edge accelerator, offering versatility beyond AI applications. Morpher, an open-source, architecture-adaptive CGRA design framework, is specifically designed to explore the vast design space of CGRAs. The comprehensive ecosystem of Morpher includes a tailored compiler, simulator, accelerator synthesis, and validation framework. This study provides an overview of Morpher, highlighting its capabilities in automatically compiling AI application kernels onto user-defined CGRA architectures and verifying their functionality. Through the Morpher framework, the versatility of CGRAs is harnessed to facilitate efficient compilation and verification of edge AI applications, covering important kernels representative of a wide range of embedded AI workloads. Morpher is available online at https://github.com/ecolab-nus/morpher-v2.**

I. Introduction

In the ever-evolving era of artificial intelligence, the growing need for edge devices to adeptly manage advanced machine learning (ML) workloads is becoming increasingly important. These devices, operating under severe power and computational performance constraints, must not only efficiently execute a wide range of ML algorithms, but also cater to an array of diverse workloads such as signal and image processing. Even within the ML sphere, the advent of new kernel types presents a continuous challenge, outpacing the capabilities of current ML accelerators. Despite their efficiency in traditional ML workloads, these accelerators fall short in adaptability for non-ML tasks, highlighting the urgent need for more flexible solutions.

Emerging as a solution to this, Coarse-Grained Reconfigurable Arrays (CGRAs) represent an innovative class of hardware accelerators, combining the flexibility of FPGAs with a higher energy efficiency comparable to ASIC-based ML accelerators. Their inherent word-level reconfigurability and high energy efficiency make them ideal for power and area-constrained edge devices. Additionally, the use of dataflow computing in CGRAs naturally aligns with the computational patterns of many AI workloads. Their unique blend of adaptability and efficiency has led to adoption of CGRAs commercially in Samsung Exynos 7420 SoC [9], Intel Configurable Spatial Accelerator [10], Sambanova RDU [17], Renesas Configurable Processor [11], and academic prototypes like HyCUBE [2], [3] among others [20], [23], [26], [27].

A CGRA architecture, as depicted in Figure 1, is characterized by a grid of interconnected Processing Elements (PEs) and multi-banked memories accessible to a subset of PEs,

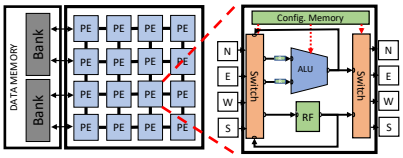

Fig. 1: A 4x4 CGRA Architecture

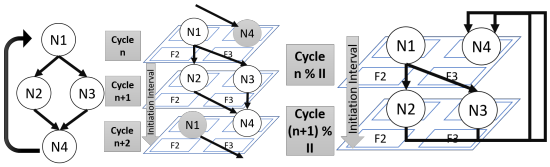

Fig. 2: (a) a loop DFG (b) a loop schedule (c) a loop scheduled on MRRG (Modulo Resource Routing Graph).

rendering it simple yet robust. The PEs consist of configurable switches, a register file, ALU, and control memory, enabling the time-multiplexed execution of instructions. The use of static scheduling negates the need for hardware structures for conflict resolution and synchronization, leading to a lightweight footprint for CGRAs. However, the effectiveness of CGRAs relies heavily on high-quality compiler mapping of application kernels onto the architecture, marking the CGRA compilation problem a substantial research area [5]–[7], [12]–[15], [22].

Application kernels are typically statically scheduled on CGRAs, a process that exposes all architectural features to the compiler for spatio-temporal mapping of dataflow graph (DFG) nodes, as illustrated in Figure 2. This mapping includes assigning operations to CGRA PEs and routing data dependencies through configurable switches and registers. A common strategy, loop pipelining, allows concurrent scheduling of operations from different iterations, enhancing the kernel's throughput as shown in Figure 2b. This process, also known as modulo scheduling, can be achieved by mapping DFG onto a modulo spatio-temporal resource graph, known as the Modulo Routing Resource Graph (MRRG) (Figure 2c) [8].

In this study, we direct our attention to the optimization of ML workloads on user-defined CGRAs, employing the Morpher tool chain—a comprehensive design tool developed for CGRA modeling and compilation. Morpher is an open-source framework that provides comprehensive support for modeling diverse CGRA architectures, supporting complex

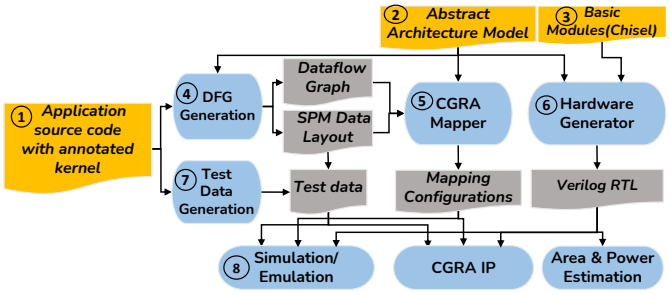

Fig. 3: Overview of Morpher Framework

kernels, and verifying functionality. It enables users to design architecture characteristics through its architecture description language (ADL) and proficiently maps complex compute kernels. Morpher also auto-generates Verilog RTL for custom CGRAs, and validates functionality through Verilator-based simulations [13], [28]. Hosted on GitHub, it integrates workflows for compilation, RTL generation, simulation, and verification, while incorporating Continuous Integration (CI) workflows to ensure error-free code and tested use cases.

The organization of this paper is as follows: Section II provides an overview of Morpher framework. In Section III, we detail how the target CGRA is modeled. Section IV not only outlines our approach to accelerate ML kernels on CGRAs, but also presents evaluations of the applied optimizations and verification process of their execution.

II. Morpher Framework Overview

Fig. 3 illustrates the overall Morpher framework. The pieces of the framework are numbered for easy reference. Yellow pieces represent user-provided inputs, blue pieces represent the functional components, and grey ones represent intermediate results generated by the functional components. The framework has three inputs: application source code with annotated kernel ①, the abstract architecture model ②, and a library of hardware description of basic CGRA modules ③. The main components of the framework are Data-Flow Graph (DFG), and data layout generation ④, CGRA Mapper ⑤, hardware (RTL) generation ⑥, test data generation ⑦, simulation and emulation ⑧.

CGRAs target loop kernels where the application spends a significant fraction of the execution time. The DFG generator ④ 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 the variables in the loop kernel to the on-chip memories of the target CGRA.

The CGRA mapper ⑤ 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) [8]. Morpher ADL supports a rich set of primitive constructs that model functional units, register files, complex software-defined routers, and multi-banked memories accessible via shared bus interfaces. The mapper models the CGRA as a time-extended resource graph called MRRG [5] where the nodes of the DFG are mapped to the time-space resource instances to maximize throughput and minimize data routing cost. The resultant mapping configuration file describes the configuration for each resource cycle-by-cycle.

The architecture generator ⑥ generates the Verilog RTL of the target CGRA design based on the user-provided abstract architecture model and the library of basic CGRA modules written in Chisel. The test data generator ⑦ for an application creates the data required for simulation and verification of the application execution. Finally, the simulator and emulator ⑧ use the mapping configurations, the test data, and Verilog RTL to simulate and emulate the execution of the application on the specified architecture.

III. Modeling the Target CGRA Architecture

For this study, the target CGRA is designed with an 8×8 PE array, comprising 8 data memories connected to the boundary PEs located on the left and right sides. The CGRA is logically structured into four clusters, with each cluster accommodating a 4x4 PE array and two 8kB memory banks, in line with a 16-bit data path, as shown in Figure 1. This arrangement is described using the abstract ADL of Morpher, a flexible tool in .json format adept at capturing a range of CGRA architectures.

Morpher's ADL balances high abstraction for user-friendliness with the need to handle intricate architectural specifics vital for verilog RTL generation. It does this by incorporating a library of crucial CGRA hardware modules like ALUs, LSUs, register files, multiplexers, and memory units, all developed in the Chisel language. This lets users tailor optimized architectures. Consequently, Morpher streamlines the design process, translating the ADL into an scala based Intermediate Representation (IR) that forms the Chisel top design and verilog RTL.

IV. Accelerating ML kernels on CGRA

In this section, we illustrate our strategy for accelerating diverse ML workloads, focusing primarily on the General Matrix Multiply (GEMM) and Convolution (CONV) kernels, using the Morpher toolchain. These kernels, despite being merely two examples among many ML kernels, act as crucial components in a plethora of ML models, contributing significantly to layers such as Fully Connected (FC), Convolution, Transformer models, LSTM, GRU, Bilinear, Self-Attention, and Graph Neural Networks (GNN) layers. While our methodology is demonstrated using GEMM and CONV kernels, it maintains broad applicability to numerous ML kernels on user defined CGRAs. Our attention is centered on diverse optimization strategies, including loop tiling, unrolling, and loop coalescing, which when combined, facilitate improved utilization of the CGRA resources and substantially boost performance.

A. Tiling Strategy for ML Kernels

The GEMM and CONV (single-input multiple-output channels) kernels are implemented on CGRA in a tiled manner

using an output stationary dataflow, as shown in Listing 1 and 2. This is just one instances of many tiling techniques widely explored for spatial accelerators and effectively applicable to CGRAs [24], [25].

At the single CGRA level (lines 9-12 in Listing 1 and lines 9-16 in Listing 2), a specific-sized kernel, called "TILE," is mapped to an individual CGRA cluster. For GEMM, a matrix multiplication of size $TI \times TK \times TJ$ is mapped to a CGRA cluster. For CONV, a convolution of a tile of size $TO1 \times TO2 \times TCo$ with a filter of size $K \times K$ is carried out. Each cluster computes an output matrix (O) with weights (W) and an input matrix (I) as inputs, the sizes of which are governed by the capacity of on-chip memory banks within each CGRA cluster. This mapping process is facilitated by the Morpher tool chain, detailed further in the next section.

The sequential loops manage data transfer from off-chip to on-chip memory, while computation is handled by parallel loops mapped onto CGRA clusters (lines 2-7 in Listing 1 and Listing 2). At the CGRA cluster level, data parallelism among different output tiles is leveraged. This allows multiple tiles to be spatially mapped on the CGRA cluster array, with each CGRA computing a single output tile. At the off-chip to on-chip level, any data exceeding the capacity of the on-chip memory banks is stored in off-chip memory, ensuring efficient data management throughout the system.

```
1  // Sequential loop: from off-chip to on-chip
2  for m in range(M/(TI*X)):
3  for n in range(N/(TJ*Y)):
4  for k in range(K/(TK)):
5  // Parallel loop: CGRA clusters
6    for x in range(X):
7    for y in range(Y):
8  // Single CGRA level
9      for i in range(TI):
10     for j in range(TJ):
11     for k in range(TK)://map this
12       O[][] += W[][]* I[][];
```

Listing 1: GEMM loop tiling and dataflow

```
1  //Sequential loop: from off-chip to on-chip
2  for i.0 in range (O1/ X*TO1):
3  for j.0 in range (O2/ Y*TO2):
4  for c.0 in range(Co/ TCo):
5  // Parallel loop: CGRA clusters
6    for x in range(X):
7    for y in range(Y):
8  // Single CGRA level
9      for i in range(TO1):
10     for j in range(TO2):
11     for c in range(TCo):
12         temp = 0;
13         for k1 in range(K):
14         for k2 in range(K):// map this:
15             temp += I[] * W[];
16     O[] = temp;
```

Listing 2: CONV loop tiling and dataflow

B. Micro Kernel Mapping on CGRA

In this section, we explore the process of optimizing and mapping GEMM and CONV kernels, onto a single CGRA cluster. We further analyze the performance implications of these optimizations. The evaluated GEMM and CONV kernels have dimensions of 64^3 and $64^3 \times 3^2$, respectively. Their corresponding tile sizes fitting into onchip memory banks of single CGRA cluster are $64 \times 16 \times 64$ for GEMM and $64^2 \times 1 \times 3^2$ for CONV.

Both GEMM and CONV kernels consists of nested loops. The user only needs to provide the application C source code to the Morpher toolchain and annotate the innermost loop that should be mapped onto the CGRA, here annotated as "map this" (Line 11 in Listing 1 and line 14 in Listing 2). The toolchain then generates a dataflow graph, representing the innermost loop body, and maps it onto the CGRA cluster. This mapping generates the necessary configurations to exploit the parallel computational capacity of the CGRA for executing the kernel. The toolchain also manages data layout in memory banks, mapping data arrays onto them to synchronize computation and data mapping.

Table I provides a performance evaluation. The Initiation Interval (II) represents the cycle count between start of two consecutive iterations, while the Minimum II (MII) is the smallest possible II dictated by the CGRA resource and loop's recurrence constraints [21]. For both the base GEMM kernel (26 DFG nodes) and the base CONV kernel (27 DFG nodes), total execution times are 2.69 ms and 314.70 ms. In both cases, Morpher succeeds in achieving the theoretical MII of 4. Notably, the lower performance of the CONV kernel arises due to an increased kernel invocation overhead, which includes transferring outer loop iteration variables from the host processor to the CGRA, as well as extended pipeline draining time. The latter refers to the period during which the pipeline completes executing instructions after the final loop iteration has commenced. These factors are amplified due to the CONV kernel's higher number of nested loop levels (5 compared to GEMM's 3). This overhead, combined with less than optimal resource utilization (40% for GEMM and 42.19% for CONV), spotlights the opportunities in optimizing CGRA performance for complex ML kernels.

Incorporating loop unrolling optimization into the GEMM kernel, as shown in Listing 3, significantly elevates performance. This optimization inflates the number of operations within the loop body, hence increasing the number of DFG nodes and amplifying parallelism, which optimizes the utilization of the CGRA PEs. As a result, the unrolled version (GEMM-U) demonstrates an increased DFG nodes count from 26 to 58 and an enhancement in utilization from 40% to 60%. This culminates in a decrease in computation time from 0.56ms to 0.25ms. These reductions confirm that loop unrolling efficiently enhances the compute utilization, resulting in an an improved performance, which is 1.13× better than the base kernel.

```
1  for (i=0;i<TI; i++)
2  for (j=0;j<TJ; j++)
3  for (k=0;k<TK; k=k+4): //map this
4    O[i][j] += W[i][k]* I[k][j]+ W[i][k+1]* I[k+1][j]
5      W[i][k+2]* I[k+2][j]+W[i][k+3]* I[k+3][j];
```

Listing 3: Unrolled GEMM kernel (GEMM-U)

TABLE I: Performance comparison of different kernels on target CGRA with speedup compared to base kernels

Kernel	Nodes	II (MII)	Utilization	Compute time (ms)*	Data transfer time (ms)*	Total execution time (ms)*	Speedup
GEMM	26	4 (4)	40.63%	0.56	2.13	2.69	1×
GEMM-U	58	6 (4)	60.42%	0.25	2.13	2.38	1.1×
GEMM-U-C	79	8 (8)	61.72%	0.27	0.49	0.76	3.5×
CONV	27	4 (4)	42.19%	8.32	306.38	314.70	1×
CONV-U-C-1	100	12 (7)	52.08%	1.53	12.75	14.28	22×
CONV-U-C-2	153	11 (10)	86.93%	1.26	11.19	12.45	25.2×

* The evaluation is conducted at a 100 MHz CGRA frequency and a 50 MBps host-to-CGRA data transfer rate. The primary focus of this study is to evaluate performance enhancement through compilation, not striving for the maximum performance achievable through efficient RTL silicon implementation, which is currently in the development phase.

Loop coalescing significantly enhances the efficiency of the CGRA implementation, by reducing invocation overheads and pipeline draining time. This is clearly demonstrated in the results for the GEMM-U-C (Listing 4) and the CONV-U-C (Listing 5) kernels. The GEMM-U-C kernel coalesces all three loops, resulting in a DFG with 79 nodes and an II of 8. This kernel requires only a single loop invocation per CGRA cluster to complete 64^3 kernel size. The data transfer time is reduced to 0.49 ms, culminating in a total execution time of 0.76 ms. This significantly enhances the overall performance, as evidenced by a performance boost of 3.54 times compared to base kernel. Similarly, the CONV kernel also shows marked improvement when optimized. The CONV-U-C-1 kernel, which coalesces the innermost two loops and fully unrolls them when K=3, results in a DFG with 100 nodes and an II of 12. The compute time is significantly reduced to 1.53 ms, as is the data transfer time to 12.75 ms, yielding a total execution time of 14.28 ms. This optimization leads to an impressive performance increase to 22.03× compared to the base kernel.

Finally, the CONV-U-C-2 kernel, which coalesces all five loop levels, demonstrates a further improvement. This kernel necessitates 16 invocations per CGRA cluster to complete $64^3 \times 3^2$ kernel size. It results in a DFG with 153 nodes and an II of 11 with 86% utilization. This optimization results in a performance boost of 25.28× compared to the base CONV implementation. These findings underscore the vital role and efficacy of loop coalescing in achieving significant performance gains in CGRA implementations.

```
for (n=0;i=0;j=0;k=0;n<TI*TJ*TK; n++){: //map this
  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 = k + 4;
  if(k+1 >= TK) {k=0; ++j;}
  if(j == TJ) {j=0; ++i;}}
```

Listing 4: Unrolled & coalesced GEMM kernel (GEMM-U-C)

```
for (int ijk=0;ijk<TCo*TO1*TO2; ijk++){: //map this
  O[] = I[] * W[] + I[] * W[] + I[] * W[]
  + I[] * W[] + I[] * W[] + I[] * W[]
  + I[] * W[] + I[] * W[] + I[] * W[]; j = j + 1;
  if(j+1 > O2){j=0;++i;}
  if(i == O1){i=0;++c;}}
```

Listing 5: Unrolled & coalesced CONV kernel (CONV-U-C-2)

C. Functional Verification

Morpher simplifies the task of generating application test data for the simulation of loop kernels, an indispensable component of CGRA functional verification. It instruments the application source code by inserting data recording functions to capture the live-in (I, W, O arrays, iteration variables transferred from outermost loops) and live-out (output O array) variables of the target loop kernel. This instrumented program is then run on a general-purpose processor, and the variables are recorded as test data for use by the simulator. In the ensuing simulation and verification phase, the Chisel top design of the target CGRA is simulated using Verilator and Chisel I/O testers, with the CGRA model functioning as a memory-mapped slave device to a host processor. The live-in variables from the recorded test data are loaded into each memory unit, and the mapping configurations from the mapper are uploaded into the automatically generated control modules. The simulator then carries out the operations, routing data through multiplexers, operating on the functional units, and recording the results to registers and memories, all as per the mapping configurations. The post-simulation memory content is finally compared with the expected results, validating the CGRA functionality with Morpher generated configurations.

V. Conclusion and Future Works

CGRAs backed by the efficient kernel mapping of the Morpher toolchain, offer a promising route for ML application acceleration. In our future work, we aim to merge the Morpher toolchain with MLIR's high-level compilation front-end. This integration will automate optimization techniques, further exploring the CGRA design space, and enhancing performance. This effort continues to strive towards unlocking the full potential of CGRA technology.

VI. 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] S. Friedman et al., "SPR: an architecture-adaptive CGRA mapping tool," in FPGA'09

[2] M. Karunaratne et al., "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in DAC'17

[3] Wang, Bo et al., "HyCUBE: A 0.9 V 26.4 MOPS/mW, 290 pJ/cycle, power efficient accelerator for IoT applications," in A-SSCC'19

[4] Dhananjaya Wijerathne, et al., "Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA", in WOSET, 2022.

[5] B. Mei et al., "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in FPT'02

[6] Z. Li et al., "LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators," in HPCA'22

[7] D. Wijerathne et al., "PANORAMA: Divide-and-Conquer Approach for Mapping Complex Loop Kernels on CGRA," in DAC'22

[8] B Ramakrishna Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in MICRO'94

[9] C. Kim et al., "Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications," in TRETS'14

[10] Fleming et al., "Intel's Exascale Dataflow Engine,"

[11] "Renesas Configurable Processor" , https://www.renesas.com/sg/en

[12] S. Alexander et al., "CGRA-ME: A unified framework for CGRA modelling and exploration," in ASAP'17

[13] Y. Guo et al., "Pillars: An Integrated CGRA Design Framework," in WOSET'20

[14] S. Dave et al., "CCF: A CGRA compilation framework"

[15] T. Cheng et al., "OpenCGRA: Democratizing Coarse-Grained Reconfigurable Arrays" in ASAP'21

[16] J. Weng et al., "DSAGEN: Synthesizing programmable spatial accelerators" in ISCA'20

[17] R. Prabhakar et al.," Plasticine: A reconfigurable architecture for parallel patterns" in ISCA'17

[18] J. Lee et al., "Flattening-based mapping of imperfect loop nests for CGRAs" in CODES'14

[19] M. Hamzeh et al., "Branch-aware loop mapping on CGRAs" in DAC'14

[20] D. Wijerathne et al., "CASCADE: High throughput data streaming via decoupled access-execute CGRA" in TECS'19

[21] T. Oh et al., "Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures" in LCTES'09

[22] D. Wijerathne et al., "Himap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction," in DATE'21

[23] N. Farahini et al., "39.9 gops/watt multi-mode CGRA accelerator for a multi-standard basestation" in ISCAS'13

[24] S. Dave et al., "Dmazerunner: Executing perfectly nested loops on dataflow accelerators" in ACM TECS'2019.

[25] G. E. Moon et al., "Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication" in IEEE TPDS'2021.

[26] V. Govindaraju et al., "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing" in Micro'12

[27] B. Mei, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix" in FPL'03

[28] W. Snyder. "Verilator and systemperl." in DAC 2004