An Agile Deploying Approach for Large-Scale Workloads on CGRA-CPU Architecture

Jiahang Lou, Xuchen Gao, Yiqing Mao, Yunhui Qiu, Yihan Hu, Wenbo Yin and Lingli Wang*
State Key Laboratory of Integrated Chips and Systems, Fudan University, Shanghai, China
*Email: llwang@fudan.edu.cn

***Abstract*—Adopting specialized accelerators such as Coarse-Grained Reconfigurable Architectures (CGRAs) alongside CPUs to enhance performance within specific domains is an astute choice. However, the integration of heterogeneous architectures introduces complex challenges for compiler design. Simultaneously, the ever-expanding scale of workloads imposes substantial burdens on deployment. To address above challenges, this paper introduces CGRV-OPT, a user-friendly multi-level compiler designed to deploy large-scale workloads to CGRA and RISC-V CPU architecture. Built upon the MLIR framework, CGRV-OPT serves as a pivotal bridge, facilitating the seamless conversion of high-level workload descriptions into low-level intermediate representations (IRs) for different architectures. A salient feature of our approach is the automation of a comprehensive suite of optimizations and transformations, which speed up each kernel computing within the intricate SoC. Additionally, we have seamlessly integrated an automated software-hardware partitioning mechanism, guided by our multi-level optimizations, resulting in a remarkable 2.14× speed up over large-scale workloads. The CGRV-OPT framework significantly alleviates the challenges faced by software developers, including those with limited expertise in hardware architectures.**

***Index Terms*—CGRA, MLIR, compiler, heterogeneous architecture, software-hardware partition**

I. Introduction

Real-world applications often entail large-scale computational workloads, such as those in Deep Learning (DL) and High-Performance Computation (HPC), which continuously grow in size and diversify in types. This escalating demand for computational power, flexibility, and parallelism aligns perfectly with the capabilities offered by Coarse-Grained Reconfigurable Architectures (CGRAs), diverging from the conventional von Neumann Architecture. Moreover, the CGRA can alter its configuration at the cycle-level cost due to the fewer configuration data compared to the field-programmable gate array (FPGA), enabling it to continuously adapt to compute different kernels within a complex task.

Nevertheless, as the CGRA consistently operates as an accelerator alongside a CPU, this heterogeneous nature brings about challenges in compiler design. When deploying large-scale workloads on a heterogeneous platform like CGRA-CPU, two primary challenges become evident: the presence of diverse kernel types and the substantial size of individual kernels. To address the former challenge, an automated approach for recognizing and offloading hot-spot code regions is essential. To tackle the latter challenge effectively, a comprehensive suite of code optimizations aimed at improving data locality and parallelism is imperative. These two aspects constitute the core focus of this paper.

Some existing CGRA frameworks [1]–[3] are specifically designed to enhance architectural configurations by tailoring them to the specific applications. However, they often lack code-level transformations, which misses the opportunity to leverage high-level optimization strategies. There also exist studies centered around source code transformation. However, CGRAOMP [4] constrains the programming model to OpenMP, limiting its universality, while optCGRA [5] is confined to specific application domains.

The popularity of hardware accelerators significantly depends on compiler effectiveness. It is with this in mind that our motivation arises. We aim to create a compilation framework that combines adept programming usability with an agile optimization methodology, harmonizing with the intrinsic complexities of CGRA-CPU heterogeneous architectures. This motivation propels the development of CGRV-OPT.

CGRV-OPT is an MLIR-Based [6] end-to-end optimizing and deploying flow for large-scale data-flow applications such as AI inference, targeting CGRA and RISC-V heterogeneous architecture. The oriented hardware in this paper is FDRA SoC [7] which contains a RISC-V CPU and a spatial PE array which accelerates data-flow computation. The primary goal of CGRV-OPT is to fully leverage the performance potential of the SoC in an automated manner. The contributions of this paper are:

- An open-source[1] end-to-end compilation framework to agilely optimize and deploy applications towards CGRA-CPU SoC, which supports multiple types of high-level programming models.
- A series of IR transformations and optimizations to achieve higher efficiency for computing and memory access, incorporating hardware-specific information.
- An automated hardware/software partitioning explorer to map different kernels in large-scale workloads to the CPU or CGRA, on the basis of performance models.
- Experiments show CGRV-OPT achieves a 2.14× speedup on average for heterogeneous architectures over exclusive CPU execution.

II. Background

A. MLIR

The MLIR [6] is an open-source compiler infrastructure. Different dialects in MLIR provide a range of abstraction levels,

[1]https://github.com/MIONkb/cgrv-opt

This work is supported by the National Key R&D Program of China (2021ZD0114701).

979-8-3503-4859-0/DATE24/© 2024 EDAA

spanning from high-level tensors and graph computing to low-level machine instructions. The following features of MLIR have greatly facilitated the development of our workloads deployment framework for heterogeneous platforms:

Firstly, the incorporation of multi-level abstractions enables us to optimize applications at various desired level. Secondly, MLIR offers convenient APIs for manipulating source code and boasts a wealth of built-in passes, which significantly streamlines our development. Thirdly, the support for modularized development has enabled distinct development teams to seamlessly integrate their individual contributions into MLIR. This facilitates the utilization of PyTorch [8] and TensorFlow [9] as application inputs .

There are several MLIR-based compilers targeting heterogeneous platforms. For instance, SODA [10] introduces a code partitioning technique tailored for High-Level Synthesis applications, which inspired the kernel extraction method in our framework. ML-CGRA [11] extends SODA to the CGRA domain. It identifies kernels in the `linalg` dialect based on predefined computation patterns, which could limit its versatility. Additionally, ML-CGRA employs a basic software simulation for CGRA execution without considering hardware details, whereas our framework provides an end-to-end solution, covering user programming to hardware verification. This distinction may result in ML-CGRA producing overly idealized simulation results. Xilinx also utilizes MLIR to facilitate static management of vectorized computing, data transfer, and synchronization within its AI engines [12].

B. FDRA Heterogeneous SoC

The targeted heterogeneous architecture, FDRA SoC [7], is depicted in Fig. 1. The complete SoC includes a CGRA [13], a RISC-V Rocket core [14], memory hierarchy, buses, and various other peripherals that are not shown for simplicity.

The CGRA is a highly parameterized array composed of Processing Elements (PEs), interconnect blocks, and I/O blocks. The Rocket core is a RISC-V CPU with a five-stage, in-order pipeline. Its primary role includes configuring and executing the CGRA. Data transfers to and from scratchpad memory are efficiently handled by a DMA controller under CPU control. Data stored within the scratchpad is seamlessly relayed to the CGRA for high-parallelism computing operations. This hierarchical structure is instrumental in ensuring the system's overall efficiency and optimal performance.

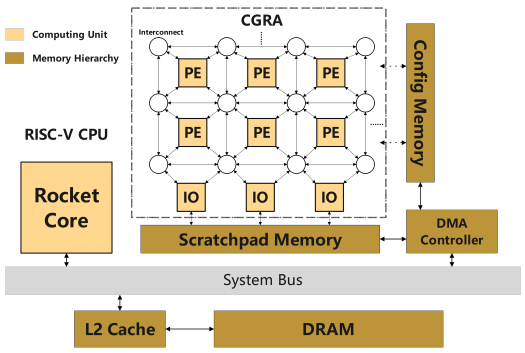

Fig. 1. The FDRA SoC, a CGRA and RISC-V CPU heterogeneous platform

The front-end compiler of FDRA SoC takes in LLVM IR as inputs and produces data-flow graphs (DFGs) as outputs, which helps bridge MLIR to FDRA. FDRA's DFG-to-CGRA mapper reads the DFGs and the Architecture Description Graph (ADG) of the CGRA. Then It generates configuration data for the CGRA. The integration with CGRV-OPT streamlines the agile deployment of workloads and reduces the need for manual intervention.

III. Framework: CGRV-OPT

A. Overview

The framework of CGRV-OPT is presented in Fig. 2. An AI model described using PyTorch or TensorFlow, as well as a program described using C source code, are all accepted as inputs by CGRV-OPT. This is made possible through the integration of three open-source projects: torch-mlir [8], tf-mlir [9], and polygeist [15]. These three projects translating applications to MLIR files that CGRV-OPT can manipulate.

Then CGRV-OPT carries out a sequence of optimizations and transforms: Initially, it identifies and extracts all kernels. After this, the optimization process focuses on enhancing the performance of these kernels, eventually leading to their transformation into LLVM IRs. These LLVM IRs can be forwarded to the FDRA toolchain to generate CGRA configuration data. On a divergent trajectory, the host code, designed to operate on the Rocket CPU, undergoes conversion to LLVM IR format. The conclusive step involves the compiling and linking, thereby yielding the ultimate executable program.

In addition, we have developed an automated software-hardware partitioning process for CGRV-OPT, which is an alternative for users. This process models the performance of kernels running on both the CPU and CGRA, applies all the optimizations and transformations available in CGRV-OPT, and determines the suitable hardware for each kernel.

The remaining portion of this section will delve into the process of kernel recognition, explore kernel optimizations at the loop-level and their transformations until reaching LLVM IR, and investigate the integration into FDRA toolchain. The partitioning process will be comprehensively discussed in Section IV.

Table I summarizes the developed passes in CGRV-OPT. Fig. 3 shows an example of code conversion process.

B. Kernel Recognition

CGRV-OPT automates the procedure of identifying key computing code regions referred to as kernels which have the potential to yield performance profit on the accelerator. A kernel in this context denotes a computational operation, such as `conv2d` or `matmul` expressed in the `linalg` dialect, or a nested `for` loop in the `affine` dialect. The process of kernel recognition is visually depicted in Fig.3(b)-(c). This recognition process is achieved by employing the passes denoted as ① and ② in Table I.

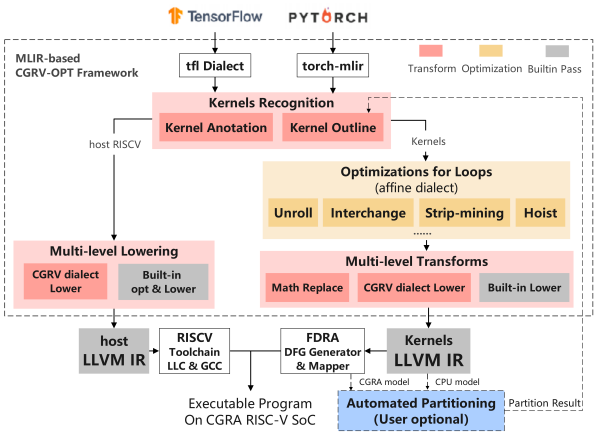

Fig. 2. Framework of CGRV-OPT. The blue section represents the optional automated software-hardware process.

It's important to note that the decision of whether a particular kernel should be mapped to the accelerator is determined by either manual intervention or the automated software-hardware partitioning workflow as elaborated in Section IV.

C. Optimizations for Nested-Loops

Our CGRA is designed to accelerate nested-loop kernels. Therefore, after identifying the kernels, the entire MLIR module is converted to the `affine` dialect, which enables us to perform following loop-level optimization and transformation. Following four loop-level optimization processes are all illustrated in Fig.3(c)-(d).

1) Loop Unrolling: CGRV-OPT applies loop unrolling to the innermost loop of a kernel, aiming to attain enhanced instruction-level parallelism (ILP) and hardware resources utilization. To determine the most suitable unroll factor (UF) for the kernel, Pass ③ in Table I generates a data-flow graph (DFG) for each available UF. It calculates the utilization of these DFGs using (1) and then select the UF that leads to the highest utilization.

$$Util = \min\left(\frac{L/SNodes\#}{I/OUnits\#}, \frac{ComputingNodes\#}{ALUs\#}\right) \quad (1)$$

2) Loop Interchange: Loop interchange changs the order of nested loops to enhance cache performance and data locality. In Pass ④, an inner loop pertaining to a higher-dimensional array will be reordered to the outer loop level after verifying dependence legality.

3) Loop Strip-mining: Loop strip-mining is employed to adjust a kernel to fit within memory size limitations. In Pass ⑤, a single loop is transformed into two nested loops. The outer loop iterates through segments or "strips" of the iteration space, while the inner loop, tagged with the `CGRV.kernel{}` annotation, carries out computations within each strip ("mining" it). This approach reduces the memory space accessed by the kernel, and the resulting kernel is called iteratively to process the data, as showed in Fig.3(d).

4) Load/Store Hoist: Pass ⑥ "hoists" memory operations to a higher loop level. This means that if a load or store operation has a memory index that is unrelated to the current loop level, it can be relocated outside of that loop level. Furthermore, if there is a pair of load and store operations existing within the same loop iteration, both accessing the same memory index and not relying on the specific value of the iteration, this load-store pair can be repositioned outside of the loop. This relocation results in the creation of a "loop-carried value", where the value from one loop iteration is carried over to the next iteration. The relocation of load/store operations reduces the frequency of memory accesses, while the resulting loop-carried value enables the utilization of accumulation computing patterns in CGRA, such as summation and product accumulation.

D. Multilevel Transform Techniques

In addition to loop-level optimizations, CGRV-OPT also integrates transformations spanning various other levels, ranging from the kernel level to the arithmetic operation level.

1) Kernel Outlining and Lowering: A recognized and optimized kernel stays within the original application module and needs to be outlined into an individual function. Pass ⑧ extracts each `CGRV.Kernel` into an outlined `func`. As a result of this transformation, a `CGRV.KernelCall` operation pointing to the outlined `func` is introduced to replace the kernel code region.

When loop strip-mining is applied, a kernel is moved to the inner loop level. For this type of kernel, the outlined kernel function, which executes on CGRA, is called repeatedly during each loop iteration. In each iteration, the repeated CGRA execution follows the same computational pattern but processes different sets of data blocks. In other words, in each loop iteration, the CPU configures the CGRA with the same settings but needs to update the address of the data blocks transferred into the scratchpad memory to handle different data sets.

As a result, CGRV-OPT introduces `LoadOffset` and `StoreOffset` operations, belonging to the self-defined `CGRV` dialect, around the CGRA call function, as depicted in Fig. 3(e). Simultaneously, the loop indices in the outline function body related to loop levels outside of the function are changed to constant values, as depicted in Fig. 3(f). This modification allows the CPU to transfer only the head address of the data blocks needed for the current iteration when calling the kernel function executed on CGRA.

2) Arithmetic Operation Replace: Nonlinear operators within AI models, such as square root ($x^{\frac{1}{2}}$) and inverse square root ($x^{-\frac{1}{2}}$), are typically represented using operations in the math dialect. In Pass ⑦, we aim to reduce the computational complexity of these operations through approximation techniques. As illustrated in Fig. 3(c)-(d), a `math.rsqrt` is converted to a sequence of simple arithmetic operations using the fast inverse square root algorithm.

3) Lower to LLVM: Both the outlined kernels and the host code with the `CGRV.KernelCall` operations will be lowered to the `llvm` dialect by pass ⑨ and subsequently translated into LLVM IR for the subsequent compilation flow.

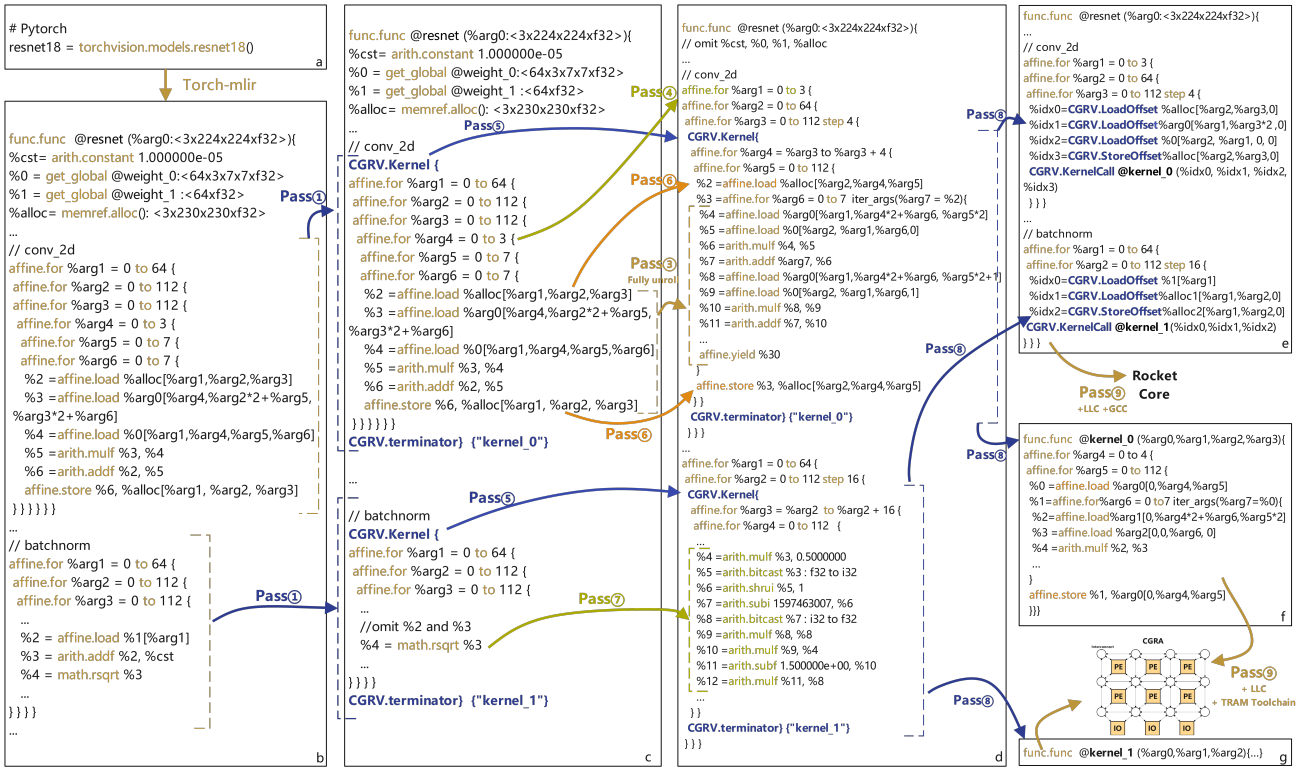

Fig. 3. Code conversion process for deploying a ResNet Model from PyTorch to executable. Passes ①-⑨ correspond to the passes listed in Table I.

TABLE I
PASSES IN CGRV-OPT

Target	Pass Name
Recognize kernels from `affine` and `linalg` dialect	① -affine-for-to-kernel
	② -linalg-op-to-kernel
Loop-level optimizations to increase parallelism and locality	③ -loop-unroll
	④ -loop-interchange
	⑤ -loop-stripmine
	⑥ -hoist-loadstore
Support uncustomized operations	⑦ -math-replace
Realize soft-hardware partitioning	⑧ -extract-kernel-to-func
Abstractions Lowering	⑨ -convert-cgrv-call-to-llvm
Automated software/hardware partition	⑩ -cgrv-auto-partition

In more detail, the `CGRV.KernelCall` operations will be transformed into general LLVM function calls for configuring and executing the CGRA, while the `LoadOffset` and `StoreOffset` operations will be converted into offset calculations for memory addresses.

Throughout the entire transformation flow, additional MLIR built-in passes are also employed such as loop-fusion and Common Subexpression Elimination (CSE) passes.

E. FDRA Toolchain Integration

After the LLVM IR for each outlined kernel is obtained, this IR is then sent to the FDRA toolchain for DFG generation, CGRA mapping, and configuration data construction. All instances of the CGRA configuration is wrapped as a CGRA execution function, including instructions on CGRA configuration, data block movement and CGRA execution.

The host portion of the MLIR code is also converted to LLVM IR, and the `CGRV.KernelCall` operations are transformed into function calls to the CGRA execution functions.

Following steps includes compile all source files with LLVM infrastructure and `gcc` compiler specific to RISC-V. Function verification and evaluating cycle count can be get through Verilator simulation or FPGA prototyping.

IV. Automated Software-Hardware Partitioning

Users can choose to manually map a kernel to the CGRA based on their expertise. Alternatively, an automated and reliable software-hardware partitioning process is available. CGRV-OPT utilizes analytical models to estimate the performance of running on both the CPU and CGRA. Leveraging these performance models, CGRV-OPT seamlessly integrates all the optimizations and transformations outlined in Section III to automate the partitioning process.

A. Estimation Models

CGRV-OPT constructs three models to estimate the cycle counts for kernel execution: CGRA computation, data transfer, and CPU execution.

The cycle count of a kernel computed by CGRA is depicted in (2). The Initial Iteration (II) and the number of iterations are determined during the kernel definition. II will be greater than 1 in the presence of access conflicts. The cycles of the critical path denote the cost of draining the CGRA pipeline.

$$Cycles_{cgra} = II \cdot Iter\# + Cycles_{cp} \quad (2)$$

The cycles consumed during data transfer consists of two parts: Direct Memory Access (DMA) transfer and CGRA configuration. Equation (3) is employed as a rough representation, assuming a linear relationship between cycles and data size. Here, the parameter α serves as a scaling factor to account for the additional time incurred due to configuration overhead.

$$Cycles_{trans} = \alpha \cdot data\# \tag{3}$$

A rough model to estimate the execution time of one kernel on CPU is shown in (4), computed based on cycles from FP and ALU units and processor memory units.

$$Cycles_{cpu} = Iter\# \cdot \sum Cycle(op) \tag{4}$$

All three models have been verified with less than 16% error when tested on the PolyBench benchmarks, which is sufficient for CGRA and CPU partitioning.

B. Automated Partitioning Process

The array of methodologies discussed in Section III is systematically harnessed to achieve process automation. Commencing with the identification and extraction of all kernels within the application module, they are subsequently treated as distinct sub-functions. In a sequential progression, the application of loop unrolling is meticulously orchestrated to maximize the hardware utility of each kernel, while loop interchange and strip-mining techniques are strategically employed to enhance reference locality and align with the constraints imposed by memory size limitations. The strategic integration of load/store hoisting and math approximation techniques further augments the optimization landscape.

Following these comprehensive optimizations, CGRV-OPT employs predictive models to determine the most suitable hardware for mapping each kernel. Firstly, CGRV-OPT checks whether a kernel is suitable for CGRA acceleration by examining whether unsupported operations on CGRA still exist, even after applying a math operation replacement pass. Then the cycle cost is calculated. The cycle cost of CGRA execution is the sum of (2) and (3). In the scenario where a kernel is designated for CPU execution, the respective sub-function is seamlessly inlined once again, exempting it from being forwarded to the FDRA SoC toolchain. The ultimate flow of lowering and compilation follows the same steps described in Section III.

V. Evaluation

A. Experiment Setup

To support the execution of complex dataflow applications like neural network inference, we have extended the hardware support to include floating-point computations in FDRA SoC. In our experiment, we set our CGRA with dimensions of 8x8, featuring 16 IO blocks and 48 PEs. The memory specifications include 8KB scratchpad banks, an L2 cache of 1MB, and a DRAM of 512MB. The SoC runs at 100 MHz on Verilator or Xilinx VCU118 FPGA board.

Our experiment comprises two main components. The first part is dedicated to evaluating individual kernels, spanning from small to large, with the aim of demonstrating the efficacy of our multi-level optimization framework. We translate the C benchmarks from MachSuite [16] and PolyBench [17] into MLIR IR using Polygeist and subsequently pass them through CGRV-OPT for optimization.

The second part of our study focuses on deploying large-scale workloads on the CGRA RISC-V platform. We conduct a performance comparison to underscore the advantages of our framework, evaluating three scenarios: (1) executing all kernels on the CPU, (2) offloading all feasible kernels to CGRA, (3) employing our automated software-hardware partitioning approach to offload kernels. All kernels running on CGRA are optimized using all available methods in CGRV-OPT. Additionally, our evaluation includes ML-CGRA with the same 8x8 scale.

B. Optimization on Individual Kernels

Table II presents the cycle counts and the number of nodes for each selected benchmark, both before and after the optimizations performed by CGRV-OPT. The first three benchmarks demonstrate an average increase of 2.1× in cycles and also higher PE utility on CGRA after the optimizations within CGRV-OPT. It's noteworthy that the data array size in `gemm64` and `conv2d` 7x7 exceeds the scratchpad limitation we set. Consequently, these two benchmarks cannot be executed on CGRA until loop strip-mining is employed to partition the data blocks.

In Table III, we present the speedup variations observed when different optimization steps are applied to the Conv2d 7x7 kernel, demonstrating the impact of optimization passes. It's important to note that the kernel only becomes compatible with the memory limitations of CGRA after loop strip-mining. Table III also indicates that performance doesn't consistently improve as different optimizations are incrementally applied; however, it's the cumulative effect of all optimizations that yields the highest performance improvement.

TABLE II
Optimization results on individual kernels

Benchmark	Before CGRV-OPT		After CGRV-OPT		
	Cycle#	**Node#**	**Cycle#**	**Node#**	**Speedup**
md	2173	29	1670	52	1.3×
stencil	10793	5	5373	13	2×
gemm32	35918	5	11681	17	3×
gemm64*	–	5	77209	17	–
conv2d 7×7*	–	6	1314096	31	–

*Unable to execute on CGRA before optimization due to memory size limitation.

TABLE III
Speedup of a Convolution Kernel on CGRA with Progressive Optimizations

Passes	CGRA Utility	Normalized Speedup
No pass applied*	×	×
Loop stripmine	0.09	1
Loop interchange	0.09	0.99×
Load-store hoist	0.09	1.72×
Loop unroll	0.48	6.9×

*Unable to execute on CGRA before strip-mining due to memory size limitation.

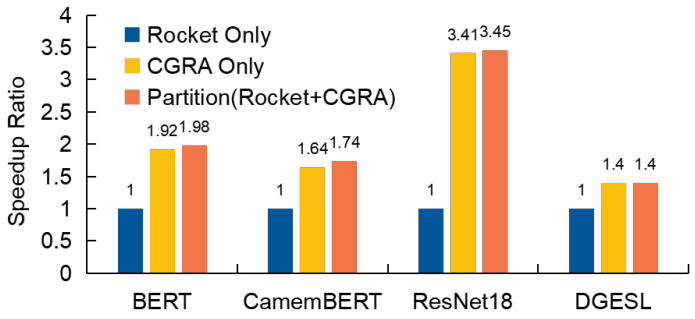

Fig. 4. Speedup ratio for large-scale workloads.

C. Deploy Large-Scale Workloads

In demonstrating the scalability of our framework, we employ three DL and one HPC workloads. The speedup ratio, computed based on the cycles of the Rocket core, is illustrated in Fig.4. Offloading all possible kernels to CGRA with extreme optimization yields an average speedup of 2.09×. Yet, leveraging the automated offloading and optimization flow of CGRV-OPT enables the platform to achieve an enhanced average speedup of 2.14×. This outcome highlights two key observations: CGRAs exhibit efficient acceleration for large-scale workloads with multiple data-flow kernels, and our auto-partitioning process possesses a noticeable ability to identify acceleration potential. `ResNet18` enjoys the highest speed-up because most of the kernels in the model (mainly 3D-convolution) are well-suited for execution on our CGRA after optimization. As for `DGESL` [18], all kernels will benefit from offloading, a fact accurately acknowledged by CGRV-OPT's prediction models.

We also compared the cycle costs on CGRA between our framework and ML-CGRA. The cycle ratios between CGRV-OPT and ML-CGRA, both excluding CPU execution time, are 1.34× for BERT and 2.19× for CamemBERT. Our FPGA execution results surpass the ideal results expected from ML-CGRA because ML-CGRA's simulator doesn't consider CGRA configuration time and other additional time costs during data transfer. Meanwhile, our framework can offload a broader range of kernel types to CGRA due to our automated software-hardware partitioning flow. This might result in better performance improvements for the entire set of large-scale workloads as it fully utilizes parallelism. In contrast, ML-CGRA can only offload a kernel if it matches a predefined pattern. Consequently, it is unable to map any kernels to the CGRA if a model lacks this pattern, such as `ResNet18`, highlighting the limitations of ML-CGRA.

We found that kernels with a high volume of memory access or numerous iterations benefit from CGRA offloading, enhancing performance. On the contrary, CPUs efficiently handle kernels with fewer iterations, reducing CGRA configuration overhead, aligning with our automated partitioning principles. Additionally, experimental results reveal that although many computing kernels move to the CGRA, a substantial portion of execution time involves CPU tasks like memory allocation and data copying. This indicates a potential area for further optimization.

VI. Conclusion

This paper introduces a comprehensive end-to-end agile workflow for deploying large-scale workloads on CGRA-CPU heterogeneous architectures. The approach is distinguished by its automated optimization techniques and software-hardware partitioning, making CGRA utilization accessible even to individuals with limited hardware expertise. Furthermore, our methodology, initially designed for CGRA and RISC-V CPU platforms, underscores its adaptability to various accelerator-CPU heterogeneous architectures, capitalizing on the intrinsic scalability of the MLIR infrastructure.

References

[1] J. Weng et al., "DSAGEN: Synthesizing Programmable Spatial Accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 268–281, 2020.

[2] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "AURORA: Automated Refinement of Coarse-Grained Reconfigurable Accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1388–1393, 2021.

[3] Li, J., Qiu, Y., Zhu, G., Zhu, Q., Yin, W., Wang, L., "THRAM: A Template-based Heterogeneous CGRA Modeling Framework Supporting Fast DSE," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2023.

[4] Kojima, T., Adhi, B., Cortes, C., Tan, Y., Sano, K., "An architecture-independent cgra compiler enabling openmp applications," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 631–638, 2022.

[5] Bae I, Harris B, Min H, et al., "Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2301–2310, 2018.

[6] C. Lattner et al., "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, p. 2–14, 2020.

[7] Y. Qiu, Y. Mao, X. Gao, S. Chen, J. Li, W. Yin, and L. Wang, "Fdra: A framework for dynamically reconfigurable accelerator supporting multi-level parallelism," *ACM Transactions on Reconfigurable Technology and Systems*.

[8] Y. Z. Sean Silva and S. Laurenzo, "Torch-mlir project." https://github.com/llvm/torch-mlir.

[9] P. J., "Mlir in tensorflow ecosystem," 2020.

[10] Agostini, Nicolas Bohm, et al., "Bridging Python to Silicon: The SODA Toolchain," in *IEEE Micro 42.5 (2022)*, pp. 78–88, 2022.

[11] Luo Y, Tan C, Agostini N B, et al., "ML-CGRA: An Integrated Compilation Framework to Enable Efficient Machine Learning Acceleration on CGRAs," in *2023 60th ACM/IEEE Design Automation Conference (DAC).*, pp. 1–6, 2023.

[12] Xilinx, "mlir-aie project." https://github.com/Xilinx/mlir-aie.

[13] Y. Qiu, Y. Cao, Y. Dai, W. Yin, and L. Wang, "Tram: An open-source template-based reconfigurable architecture modeling framework," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, p. 61–69, 2022.

[14] Asanovic, Krste, et al., "The rocket chip generator," in *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 4 (2016)*, pp. 6–2, 2016.

[15] Moses W S, Chelini L, Zhao R, et al., "Polygeist: Raising C to polyhedral MLIR," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 45–59, 2021.

[16] Reagen, Brandon, et al., "MachSuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)* , pp. 110–119, 2014.

[17] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite." http://www.cs.ucla.edu/pouchet/software/polybench.

[18] Dongarra J J, Luszczek P, Petitet A., "The LINPACK benchmark: past, present and future," in *Concurrency and Computation: practice and experience,2003,15(9)*, pp. 803–820.