Pacific Northwest
NATIONAL LABORATORY

PNNL-34889

Software Defined Architectures for Portability and Performance

September 2023

Antonino Tumeo
Nicolas Bohm Agostini
Gokcen Kestor
Sutanay Choudhury

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from
the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fox: (865) 576-5728
email: reports@osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov

PNNL-34889

Software Defined Architectures for Portability and Performance

September 2023

Antonino Tumeo
Nicolas Bohm Agostini
Gokcen Kestor
Sutanay Choudhury

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Abstract

The Software Defined Architectures for Portability and Performance (SODAPOP) project developed a co-design framework to partition and map converged applications on specialized heterogeneous architectures. We started from key domain applications that combine scientific simulation with data analytics and machine learning as drivers to integrate our framework. The framework includes high-level compilers that interfaces with high-level programming frameworks, domain-specific optimization passes, and hardware-oriented optimizations. The framework leverages hardware generators to enable specialization and facilitate exploration of custom system designs.

Summary

In this project, we aimed at devising an end-to-end software toolchain enabling rapid design and evaluation of breakthrough heterogeneous hardware designs. Specifically, we started by considering specific application drivers to derive common computational patterns and requirements. Consequently, we extended and enhanced our existing software tools (the COMET Compiler and the SODA Synthesizer) to enable partitioning and mapping of the application drivers on heterogeneous systems-on-chip (SoCs) or systems-in-package (SiPs). Heterogeneous SoCs or SiPs are composed of components of the shelf as well as specialized custom accelerators generated by the toolchain for critical application kernels. These custom accelerators can be mapped on application-specific integrated circuits (ASICs) as well as reconfigurable devices such as Field Programmable Gate Arrays or Coarse Grained Reconfigurable Arrays. The project led to new contributions to our open-source tools, facilitating end-to-end agile hardware design and evaluation. Additionally, the research focused on extending the tools to better support kernels of the application drivers that exhibit low latency inference requirements or irregular behaviors (such as graph traversals).

Acknowledgments

This research was supported by the Data Model Convergence (DMC), under the Laboratory Directed Research and Development (LDRD) Program at Pacific Northwest National Laboratory (PNNL). PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DOE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL01830.

Contents

Figures

Tables

1.0 Introduction

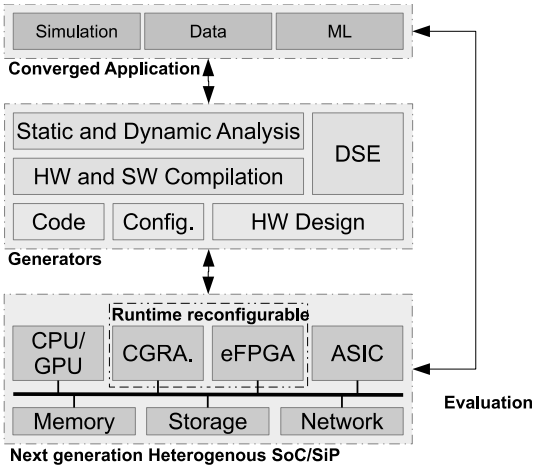

Figure 1: Overview of our Co-Design Approach

In SODAPOP, we developed a co-design framework and demonstrated it on key building blocks of scientific applications that integrate scientific simulation with data analytics and machine learning. SODAPOP integrates and build upon existing open-source solutions (the COMET Compiler and the SODA Synthesizer, composed of SODA-OPT, Bambu, and OpenCGRA), to enable generation of specialized accelerators, and their evaluation down to physical layout. SODAPOP provides mechanisms to map applications on heterogeneous Systems-On-Chips (SoCs) or Systems-in-a-Package (SiPs) composed of off-the-shelf central processing units (CPUs) and Graphic Processing Units (GPUs) and generate specialized custom accelerators for critical tasks. Specialized custom accelerators could be designed as ASICs, leveraging our high-level synthesis backends (Bambu), or as reconfigurable embedded devices, leveraging our OpenCGRA generator as well as extensions to our co-design flow to specialize embedded FPGAs mixing hard macros with fine grained reconfigurable solutions. Figure 1 provides a high-level overview of our co-design approach, which starts by considering novel applications that combine scientific simulation, data analytics, and artificial intelligence/machine learning to drive the enhancements of the toolchain to perform partitioning and generate the specialized architectures.

2.0 Description of the application

Our approach begins with domain applications, considering their requirements and characteristics. In particular, we focus on scientific applications that combine simulation with data analytics and machine learning (ML). Such applications, which include power grid analysis, quantum chemistry, and material science, can benefit from mapping applications onto heterogeneous specialized SoCs/SiPs to meet specific constraints, both at the application level (e.g., real-time decision process) and at the system level (e.g., power efficiency, cooling, etc.).

TODO Description of the applications? Something that motivates LLAMA and OpenCGRA?
Many exploratory queries for scientific research and discovery are characterized by attempts to find optimal combinations of factors that maximize a property of interest. Given a chemical reaction, finding the optimal combination of operating conditions (e.g., temperature), reactants (the reaction's chemical input), and catalysts (substances that increase the reaction rate) is essential to discovering new energy-efficient fuels or lower-energy chemical conversion processes. For highly empirical scientific fields such as chemistry, such combinatorial searches coupled with scientific reasoning is performed by practitioners for hypothesis generation and testing. The emerging capability of large language models (LLMs) to reason in human-comprehensible terms provides an opportunity to accelerate the cycle of scientific discovery via autonomous, machine-driven reasoning.

We have developed ChemReasoner (https://github.com/pnnl/chemreasoner), that given a natural language question such as ("what are the top-3 catalyst for reaction X?") implements a LLM-based heuristic search framework to search over different regions of the chemical space using different variations reaction constraints generated by the LLM itself.

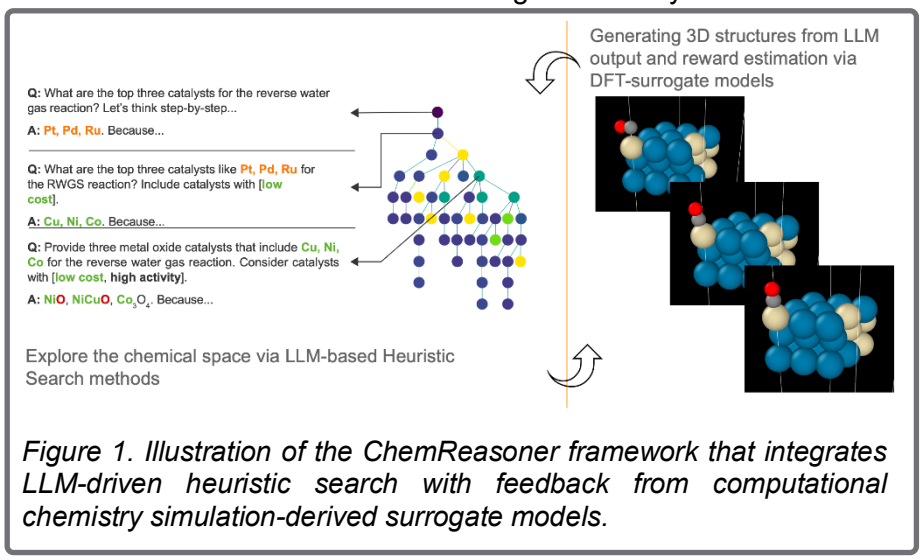

Figure 1. Illustration of the ChemReasoner framework that integrates LLM-driven heuristic search with feedback from computational chemistry simulation-derived surrogate models.

ChemReasoner explores the search space (see Figure 1) of chemical compounds by utilizing prior probabilities that could be derived from a LLM to initiate the exploration, and guides it via reward obtained from the exploration process to prune unpromising actions. We currently use *"adsorption energy"*, a key measure of reactivity as the reward function.

Computational cost: The ChemReasoner framework supports both Monte Carlo Tree Search and Greedy-beam Search (developed by DeepMind in contemporary work). Our preliminary quantitative and qualitative evaluations on a new biofuels-focused reasoning dataset with 100 questions shows that the heuristic search methods significantly outperform ChatGPT. However, both of these methods are computationally intensive, performing 200-300 LLM and GNN inferences for each question. Running a full suite of benchmarks with multiple algorithms costs can be significant, and thus motivates the development of open-source LLMs that can be trained and tailored in-house to meet unique scientific application needs.

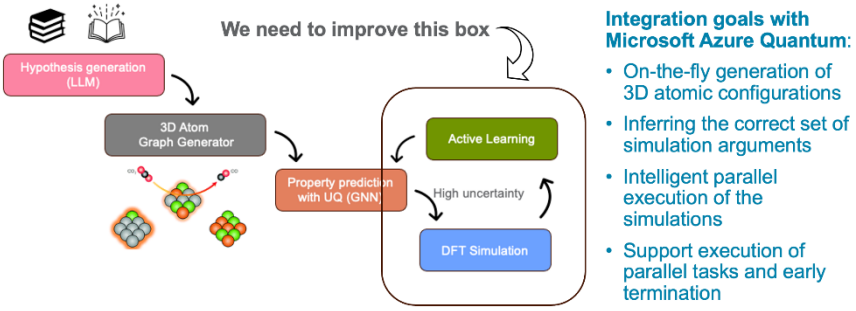

Figure 2. The proposed research seeks to extend ChemReasoner with on-demand simulations.

3.0 Compiler tool extensions

This work proposes a domain-specific compiler framework to develop efficient graph algorithms represented by linear algebra operations. Our work adheres to the GraphBLAS standard, which provides a comprehensive set of graph primitives for sparse matrices and vectors of various types and extends the traditional linear algebra operators with semirings and masking to achieve higher performance. The GraphBLAS-based approach provides a consistent way for graph algorithm implementation through common graph primitives that can be optimized using well-studied techniques, and it avoids the complexity of writing different ad-hoc implementations com- mon with traditional vertex- or edge-centric approaches. In most cases, the algorithmic complexity of the graph algorithms implemented using linear algebra is close to the complexity of the implementation based on vertex- or edge- transverses. Given a graph $G(V, E)$, where V is the set of N vertices (or nodes) in the graph and E is the set of M edges that connect two vertexes, such that $e_{ij} \in E$ iff there are two vertexes $v_i, v_j \in V$ and there exists an edge between the two. While graphs can be represented in various forms, such as a list of edges and the vertices they connect, a graph $G(V, E)$ in algebraic implementations of graph algorithms is represented as an adjacency matrix A of size NxN, where the elements $a_{ij} = 1$ iff there exists an edge $e_{ij} \in E$. Once a graph is represented as an adjacency matrix, the implementation of graph algorithms can leverage well-defined linear algebra operations. For example, visiting all neighbors from a source vertex v_i only requires multiplying the adjacency matrix A by a vector x that has all zero elements except x_i. Similarly, multiplying A by itself k times yields relations between vertices at distance k. For example, A^2x and A^kx return the neighbor lists that are two and k hops away, respectively. Given the nature of graphs, generally $N^2 \gg M$, hence A is very sparse. Storing A in a dense format unnecessarily increases the memory and computing requirements and results in inefficient execution. This framework employs sparse linear algebra operators to reduce both the memory and computing requirements, as only the non-zero elements of A need to be stored in memory and considered during the computation.

The proposed DSL is based on index notation (or Einstein notation), which is concise, expressive, and widely used way to express dense and sparse tensor computations. For example, the multiplication of two matrices A and B can be expressed as $C_{ij} = A_{ik} * B_{kj}$, where i and j are the free indices that appear in the output, whereas, the remaining indices are the summation indices, k in this case. The Einstein notation is adopted and supported in many common programming models to express tensor operations, such as the `numpy.einsum` API in NumPy [13], PyTorch (`torch.einsum`) and TensorFlow (`tf.einsum`). It is also the input language in deep learning frameworks such as Tensor Comprehension [14], and sparse tensor compilers such as TACO [15] and COMET [7], [16]. All these libraries and frameworks implement some variant of the original Ein- stein notation to expand the expressiveness. In this work, we adopt a consistent Einstein notation semantic as used by `numpy.einsum` and state-of-the-art compilers [7], [15]. We refer the reader to the `numpy.einsum` API page for a more comprehensive description of the notation. Note that a summation or contraction index is implied if an index variable appears on the right-hand-side tensors but not on the left-hand- side tensor. In addition, a custom function can be specified to express other types of reduction other than summation, such as $A_i = max(B_{ij})$. It is also possible to express operations such as MTTKRP (Matricized Tensor Times Khatri-Rao Product) as $A_{ir} = B_{ijk} * D_{jr} * C_{kr}$, where the index variable j and k are summed. The sparse formats of each tensor are specified as type annotations, and a compiler will automatically generate code from the expression in the back end. Also, note that for some operation sequences, the order of evaluation can result in different asymptotic time

complexities, such as a chain of matrix multiplications. In this work, we assume such order is already determined and do not attempt to reorder matrix multiplications.
From an implementation point of view, the proposed com- piler is based on the Multi-Level Intermediate Representation (MLIR) framework and built on top of COMET. COMET is a dense and sparse tensor algebra compiler that targets multiple architectures. It has been extensively used to optimize dense tensor contractions within the NWChem quantum chemistry framework. The work introduces specific code optimizations and transformations for sparse linear algebra operators, and DSL support to implement algebraic formulations of graph algorithms. MLIR, which is part of the LLVM ecosystem, provides a solid foundation to build new compiler frameworks and a set of common optimizations and code transformation passes, such as loop unrolling, tiling, and vectorization. New optimizations and architectures added to the MLIR framework will be readily available to the proposed compiler and its users.

3.1 Efficient Code Generation of Graph Computation

The proposed compiler is based on the MLIR framework, which provides a multi-level IR and an infrastructure to perform progress lowering. The key insight in MLIR, hence in this work, is that different optimizations can be performed at each level of the IR stack (or dialects), from high-level, domain-specific optimizations at higher levels to architecture- specific optimizations at low levels and that optimizations and dialects can be composed to efficiently generate executable code for various target architectures.

Index Tree Dialect: This task introduces a new MLIR dialect, the *Index Tree dialect*, between the COMET's existing *tensor algebra dialect* and the MLIR Structured Control Flow (SCF) dialect with the specific objective of performing efficient code transformation and optimizations for sparse computation. The index tree dialect is a representation of an index tree notation, which is widely adopted by various code generation models of tensor computations. While the COMET tensor algebra dialect is designed for traditional tensor algebra operators, the index tree dialect provides a generic and efficient representation of computing expressions (operators and their relations) based on two types of nodes, indexation, and computation nodes. The specific instantiations of the nodes are determined by the computation expressed in the tensor algebra dialect during lowering. However, the index tree nodes are not limited to the traditional tensor algebra operators and can be used to implement extended operators, such as semiring and masking.

Code optimizations and Transformations: To provide efficient code generation for sparse kernels, this work addresses three major challenges during the code generation: 1) high insertion cost into sparse output tensors 2) the unknown size and distribution of nonzero elements in sparse output tensors, and 3) the difficulty of parallelization.

- **Workspace Transformation:** In general, inserting a new element into the sparse data structure of the output tensor has a high time complexity. To eliminate this complexity, most compiler frameworks and libraries store the output of a sparse computation in a dense format. Although this approach greatly reduces the cost of computation, it may lead to "densification" of the data structures and increased memory footprints for the output results, which may result in unnecessary wasted space and an inability to execute. To address these challenges, we developed a novel approach called workspace transformation to store sparse output directly in sparse formats such as CSR. The workspace transformation introduces temporary intermediate dense data structures (called the workspace) to avoid irregular access to the original sparse data structures. This not only simplifies the code generation algorithm, but also improves

the data locality of the generated code by narrowing some irregular access to the dense data structure. Unlike previous work that requires users to determine the index to apply workspace transformation, in this work, the compiler automatically identifies the index involved based on the storage format. The workspace transformation can be applied to two types of indices in a sparse-sparse expression: input indices associated with sparse dimensions in both input tensors and output indices associated with sparse dimensions in the output tensor. The workspace transformation performed depends on the indices to which it is applied.

- **Two-Phase Computation:** One of the challenges of sparse computation comes from the unknown size and distribution of the output tensor. First, the number of nonzero elements in the output tensor is unknown before computation, which makes memory management very difficult. A general method is to allocate a very large chunk of memory to avoid the case where the output size exceeds the allocation, which results in redundant memory usage. Second, it is hard to know beforehand how nonzero elements are distributed among different rows (in case of row-major storage) in the output tensor. To update the output tensor in parallel, it is common to use a lock on the critical data structure, which results in high synchronization overhead. To determine the needed size and real distribution of the output tensor, this work generates the code with two phases for sparse computation. The first phase is called the *symbolic phase*. It follows the same procedure of the given sparse computation (e.g., SpGEMM) in a "symbolic" way that it does not execute the computation, but only records the nonzero distribution of the output. After that, the symbolic phase can also determine the true number of nonzero elements in the output tensor and then allocate the sparse data structure with only the needed memory size. The second phase is called the *numeric phase*. It performs the real "numeric" computation with the prior knowledge from the symbolic phase. For some components of the sparse data structure (e.g., the index array in CSR), the output can be placed directly at the correct location. that is provided by the symbolic phase. Therefore, the two- phase computation can minimize memory usage effectively.
- **Automatic Parallelization:** Another challenge of sparse computation is the inefficient parallel execution due to the un- known distribution of the output nonzero elements. Unlike the dense matrix computation that can be parallelized by simply dividing the regular output, the sparse computation generates irregular sparse output. A na¨ıve parallelization method would require locks on the critical sparse data structure, which may result in a high synchronization overhead. The use of two-phase computation enables efficient parallelization without locks. First, the symbolic phase can be naturally parallelized among one dimension of the tensor. It does not have update conflicts because the symbolic phase does not perform numeric computations but only records the output distribution. Second, the numeric phase can also be parallelized according to the locations provided by the symbolic phase. Different parts of nonzero elements can be updated simultaneously without conflicts. Moreover, the codegen can generate a private workspace for each worker during parallel execution. In this way, multiple workers will not have writing conflicts with each other, and a worker can reuse their private workspace without unnecessary reallocation.

3.2 Performance Evaluation

First, we evaluate the performance of sparse matrix-sparse matrix operation with plus-times semiring (i.e., SpGEMM) using the input matrix as a mask inside both our compiler and SuiteSparse:GraphBLAS when all the optimizations are enabled in the compiler.

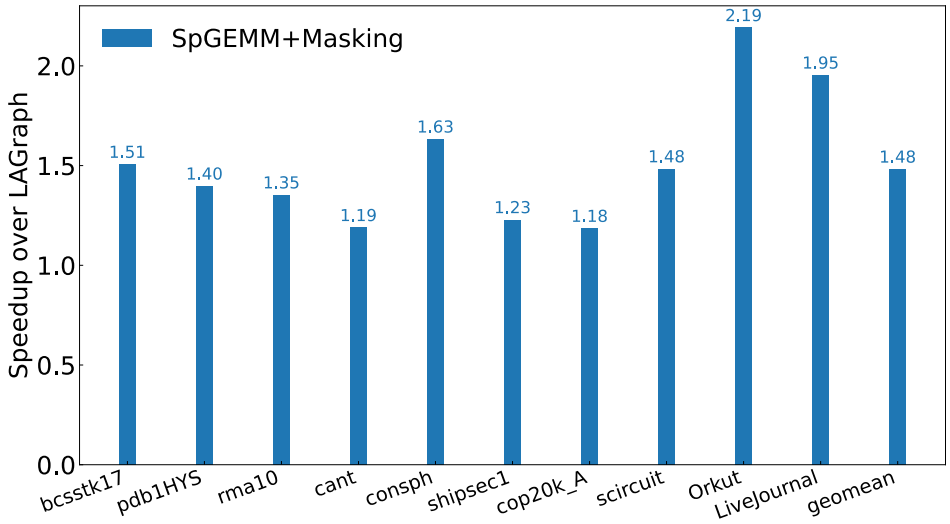

Figure 2: Performance of masked SpGEMM (i.e., B<A> = A*A) as compared to SuiteSparse:GraphBLAS

Figure 2 illustrates the speedup obtained by code generated by the compiler as compared to library-based realization of the same SpGEMM operations. The figure shows that the our performance is better than SuiteSparse:GraphBLAS across various inputs, and the compiler obtains up to 2.19× speedup, with 1.48× geometric mean speedup. Masking optimization avoids unneeded computations based on the requirements of the graph algorithms. Specifically, masking intervenes in the basic sparse vector-sparse matrix multiplication that is performed for each row of the other input matrix. At each iteration, the corresponding sparse row from the mask matrix is converted to an intermediate dense vector to support random $O(1)$ access to the elements in the mask. This accelerates the skipping of computations that do not need to be performed. The workspace transformation also provide some additional speedup as compared to SuiteSparse:GraphBLAS, which is evaluated further in this Section.

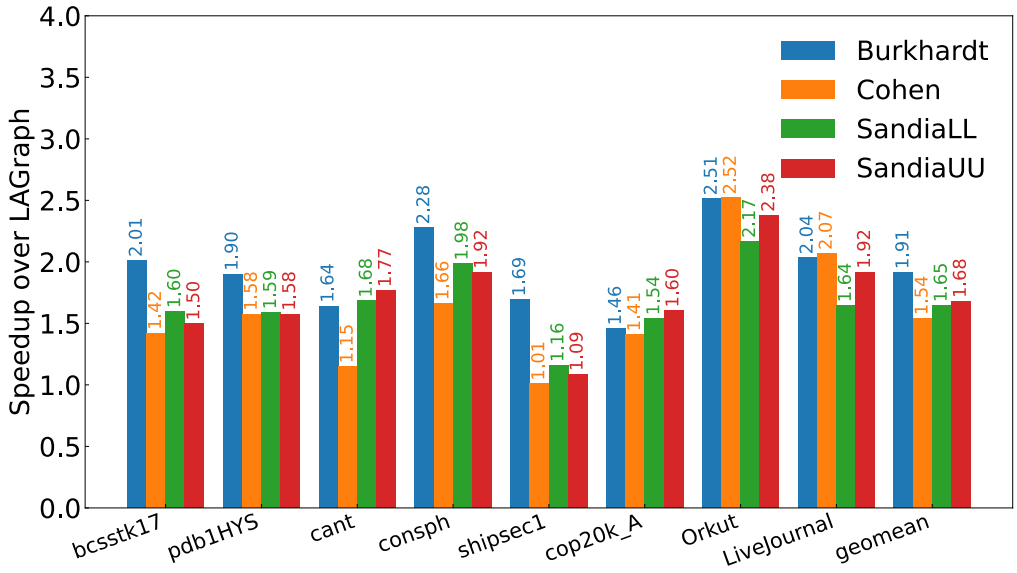

Figure 3: Performance of the four Triangle Counting algorithms with masking as compared to LAGraph

Next, we evaluate the performance of four different *Triangle Counting* algorithms implemented by our compiler. The performance is compared against LAGraph implementations of the same algorithms. In our experiments, we evaluate the implementation of these algorithms with the `plus-pair` semiring instead of SpGEMM operation (i.e., `plus-times` semiring) and with masking instead of the element-wise multiplication operation. These experiments demonstrate the benefit of all optimizations proposed in this paper. The cost to determine the strict lower and upper triangular parts of the input matrix is not included in the performance evaluations. Figure 3 shows the performance comparison of all four Triangle Counting algorithms implemented within our compiler and LAGraph with masking. It shows that our work can achieve up to 2.52× speedup, and 1.91×, 1.54×, 1.65×, and 1.68× geometric mean speedup over LAGraph for Burkhardt, Cohen, SandiaLL, and SandiaUU algorithms across all input matrices, respectively. The performance breakdown of various optimizations proposed inside the compiler is discussed later in this Section. In the results in Figure 3, when the input matrices have a relatively high density (e.g., `bcsstk17`), we do observe diminishing returns for algorithms that use sparser matrices such as lower and upper triangular. We attribute this to our masking implementation that is based on the push method. The push- based masking is more suitable for masks with higher density, whereas, pull-based masking is suitable for sparser masks. We plan to investigate our design choices in future work.

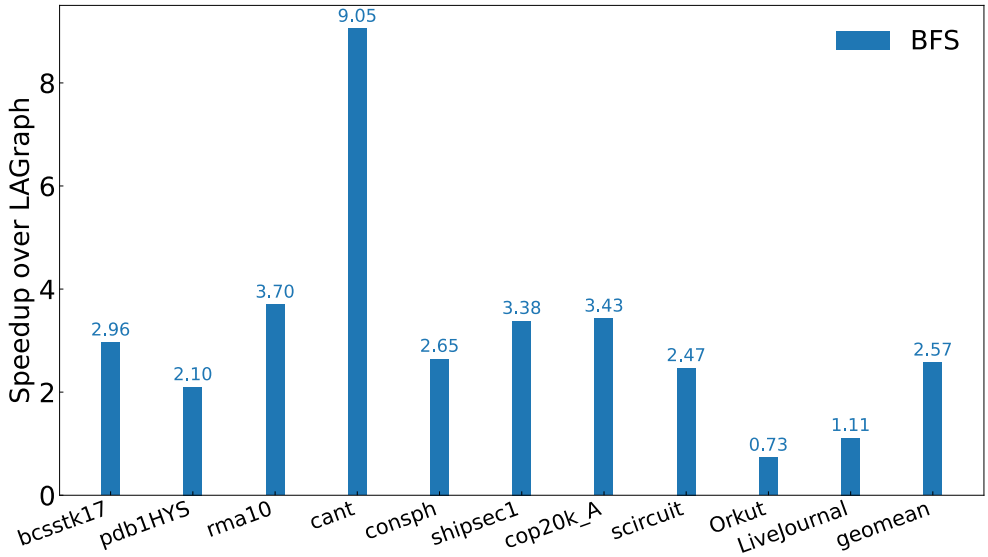

Figure 4: Performance of the BFS algorithm with masking as compared to LAGraph

Next, Figure 4 illustrates the performance comparison of BFS implementation between our work and LAGraph. It shows that our work can achieve up to 9.05× speedup over LAGraph and geometric mean 2.57× speedup for all input matrices. The main speedup comes from our use of the workspace transformation. The major computation in the BFS level algorithm involves finding the next frontier in each iteration. It is achieved by performing a sparse vector-matrix multiplication with masking. Workspace transformation can avoid the expensive insertion into the middle of sparse data structures and performs asymptotically faster.

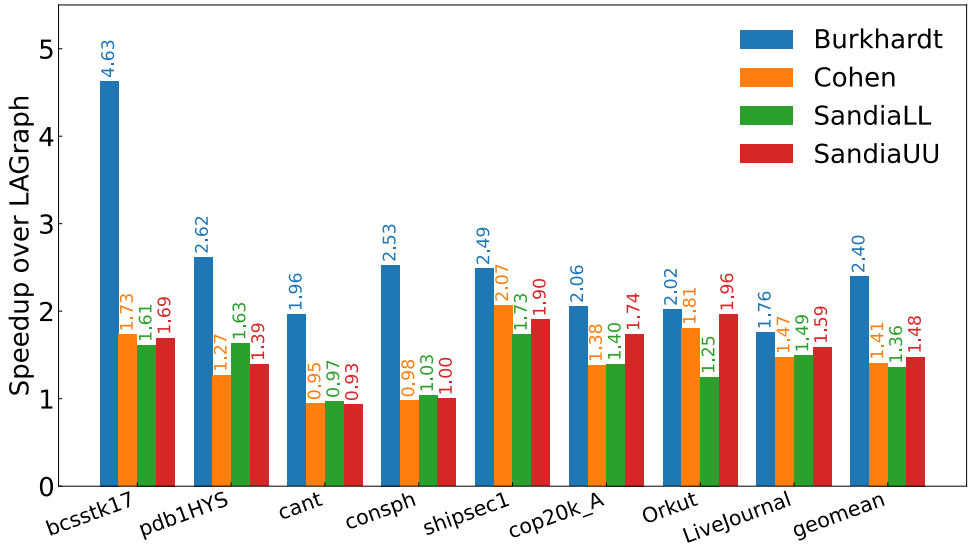

Figure 5:Parallel performance of the four Triangle Counting algorithms with masking as compared to LAGraph.

Parallel Performance: Figure 5 shows our parallel performance of four Triangle Counting algorithm with masking com- pared with LAGraph. All experiments use 24 threads. It shows that our work can achieve up to 4.63× speedup over LAGraph among all used input matrices, besides up to 2.02× speedup among the two large inputs Orkut and LiveJournal. Our work also achieves 2.40×, 1.41×,

1.36×, and 1.48× geometric mean speedup over LAGraph for Burkhardt, Cohen, SandiaLL, and SandiaUU algorithms among all input matrices, respectively. The results demonstrate that the compiler can achieve high-performance parallelization, thanks to the two-phase computation.

4.0 Hardware generator extensions

In this project, we used the SODA Synthesizer toolchain to generate specialized accelerators for key computational kernels used in the application driver. We have focused on the synthesis of a specialized transformer engine for the large language model, and on extending the functionalities of the SODA-OPT frontend with the OpenCGRA backend, providing a new solution (named ML-CGRA) to specialize the generated CGRA for low latency inference.

4.1 Synthesis of LLAMA Models

Large Language Models (LLMs) have emerged as highly capable AI assistants that excel in complex reasoning tasks requiring expert knowledge across a wide range of fields, including specialized domains such as programming and creative writing. They facilitate interaction with humans through intuitive chat interfaces, leading to rapid and widespread adoption. LLM-based autonomous agents have started to be adopted as they excel in diverse tasks by harnessing the human-like capabilities of LLMs when deployed within a framework.

Llama 2 is a family of large language models (LLMs) developed by the AI group at Meta, ranging in scale from 7B to 70B parameters. These LLMs are optimized for dialogue use cases and outperform open-source chat models on most benchmarks. Several implementations exist to perform inference in the Llama2 models. For initial investigation on synthesizing accelerators for LLMs, we leverage the Llama2.c project. Which is an open-source project designed for minimalism and simplicity offering a full-stack (train and inference) solution for Llama 2 models, with the ability to load and execute Meta's pre-trained model weights in fp32 format.

The creation of the accelerator for the llama2 large language model leveraged the generation of a GDS design using the SODA toolchain and the OpenROAD flow while synthesizing the functions that implement inference in the model. The transformer function, synthesized from C, is responsible for generating logits representing the probability of the next token. Weights and inputs are managed through memory mapped configuration struct, allowing for adaptability to represent smaller or actual llama2 models. A primary challenge in implementing monolithic accelerators for large LLMs lies in supporting the LLM’s weights and context, which this accelerator addresses by considering the model configuration and leveraging an external memory.

The accelerator interface expects memory mapped arguments for including configuration, current state, and model weights. The generated accelerator can be configured to support different Llama2 sizes with the following C++ struct in Figure 2.

```
typedef struct {
    int dim; // transformer dimension
    int hidden_dim; // for ffn layers
    int n_layers; // number of layers
    int n_heads; // number of query heads
    int n_kv_heads; // number of key/value heads
    int vocab_size; // vocabulary size
    int seq_len; // max sequence length
} Config;
```

Figure 6. Configuration parameters to support different Lamma2 model sizes. This C struct must be memory mapped in 28 consecutive bytes and its address should be passed to the accelerator.

Our initial synthesis efforts targeted the FreePDK 45nm technology library. We started our design space exploration by changing the target operating frequency. Higher operating frequencies (or smaller cycle latency) yield designs that execute faster, but the accelerator's internal finite state machine may have more states as less operations can be performed in the same cycle with reduced latency. In Figure 3 and Figure 3 we present the final GDS designs for accelerators synthesized with 200MHz, and 500MHz. These accelerators presented finite state machines for the main function with 218 and 298 states. From these values we extrapolate that increasing the frequency from 200MHz to 500MHz (2.5x) can provide a theoretical maximum speedup of 1.8x.

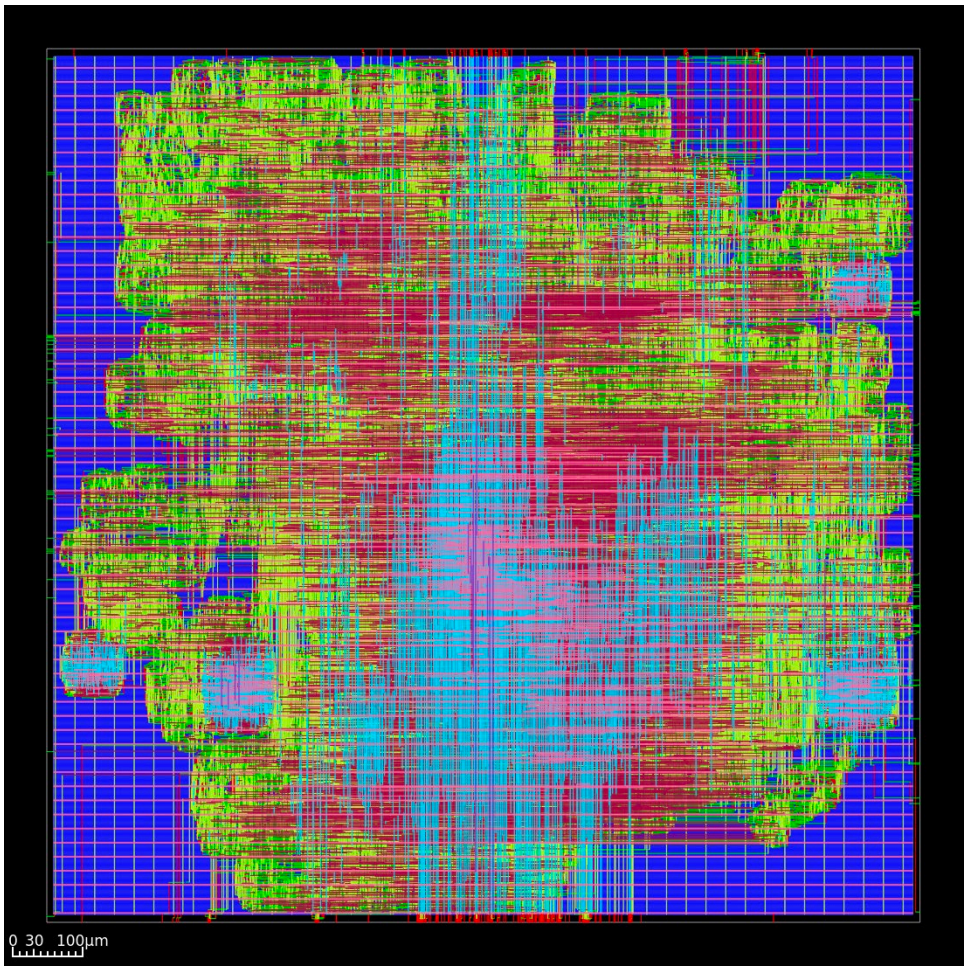

Figure 7. Chip design implementing the inference function of a llama2 model, synthesized with FreePDK 45nm targetting 200MHz, and die area of 1240umX1240um.

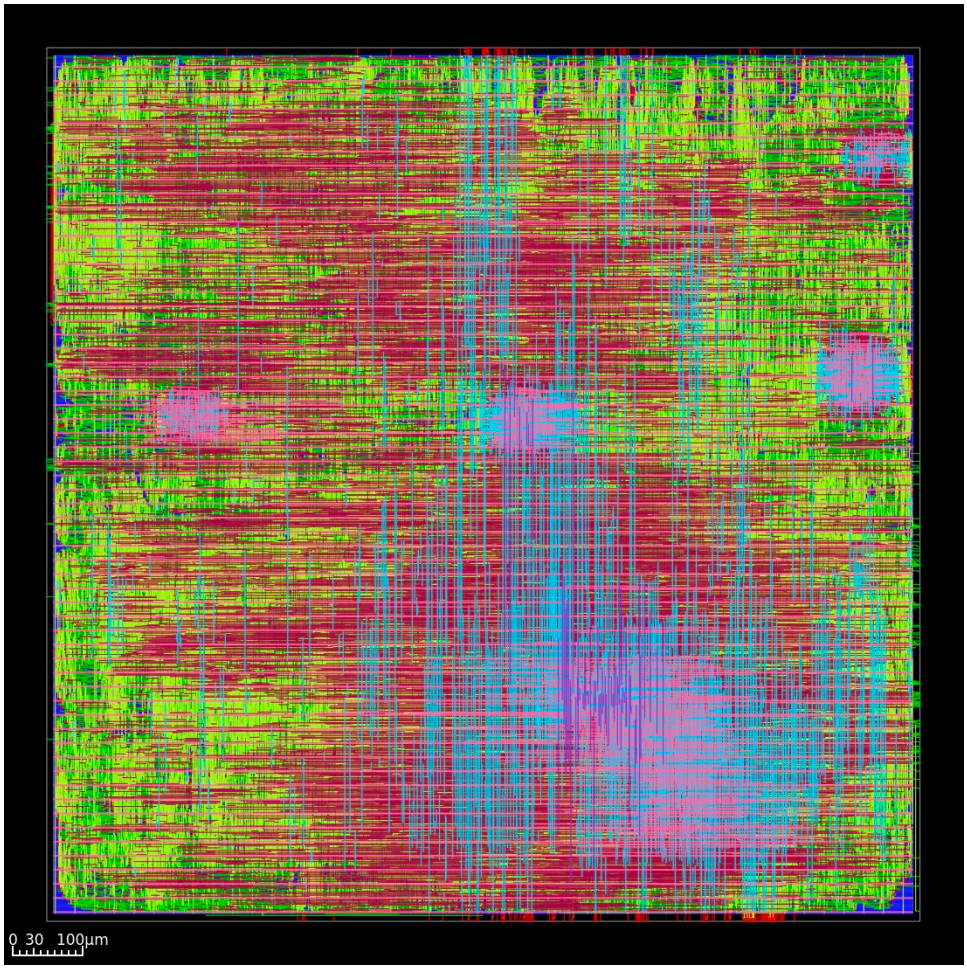

Figure 8. Chip design implementing the inference function of a llama2 model, synthesized with FreePDK 45nm targetting 500MHz execution, and die area of 1240umX1240um.

4.2 ML-CGRA extension

Coarse-Grained Reconfigurable Arrays (CGRAs) can achieve higher energy-efficiency than general-purpose processors and accelerators or fine-grained reconfigurable devices, while maintaining adaptability to different computational patterns. CGRAs have shown some success as a platform to accelerate machine learning (ML) thanks to their flexibility, which allows them to support new models not considered by fixed accelerators. However, current solutions for CGRAs employ low level instruction-based compiler approaches and lack specialized compilation infrastructures from high-level ML frameworks that could leverage semantic information from the models, limiting the ability to efficiently map them on the reconfigurable substrate. This paper proposes ML-CGRA, an integrated compilation framework based on the MLIR infrastructure that enables efficient ML acceleration on CGRAs. ML-CGRA provides an end-to-end solution for mapping ML models on CGRAs that outperforms conventional approaches by 3.15x and 6.02x on 4x4 and 8x8 CGRAs, respectively. The framework is open-source and available from https://github.com/tancheng/mlir-cgra.

In the last decade, a plethora of machine learning (ML) applications have gained prominence across several fields to find fast solutions for challenging problems. As the scope of these problems increases in generality and improved results, the scale of both the underlying data and

model complexity grows exponentially, placing increased pressure on implementations to evaluate these ML models quickly *and* efficiently, leading to the creation of many hardware-acceleration platforms. Due to the sheer scale of modern deployments and the desire to constantly improve model performance, being able to retarget a platform to support new algorithms in a short time is a crucial feature. Simultaneously total power usage provides a counterbalancing concern guiding hardware designs to specialized, more static hardware.

Coarse-Grained Reconfigurable Arrays (CGRAs) provide an interesting compromise of these features, being more flexible than specialized application-specific integrated circuit (ASIC) solutions but outperforming general purpose solutions (such as central processing units and graphic processing units - CPU/GPU) in terms of perf/watt.

While CGRAs offer the potential to achieve better power- and energy efficiency than alternatives, the standard CGRA compilation frameworks focus on inner-most loop kernel acceleration through dataflow graph (DFG) mapping through low level instruction-based compilation approaches. This provides generality but leaves out the potential for a slew of coarser-grain considerations that when properly leveraged, could significantly improve end-to-end performance. For example, maintaining semantic information on ML operators and layers connections simplifies the mapping on pre-defined parametric templates. As a result, in practice, a user wishing to generate a high-performance implementation with conventional compilation flows must work around the compiler's decisions and manually refine the input model, inlining specialized kernels, fusing loops, and tuning parameters by hand. While doable, this approach dramatically reduces the attractiveness of the CGRAs of a substrate. Alternatively, it has been shown that for some applications it is possible to implement design space exploration approaches that also consider compiler optimizations to obtain high-performance implementations. While these solutions allows to find better design points, they still operate at the instruction level and inherit a level of brittleness as the CGRA user cannot easily modify the generated code and needs to repeat the entire design space exploration process.

The goal of this work is to get the best of both CGRA compilation approaches and get the full flexibility of generic compilation passes while also exploiting higher-level structures in the code to improve overall performance. To achieve this, we propose ML-CGRA, an open-source integrated end-to-end compilation framework that enables efficient and flexible ML acceleration on CGRAs.

4.2.1 ML-CGRA

ML-CGRA is an integrated framework built on top of SODA-OPT, the SODA Synthesizer MLIR frontend that compiles machine learning models described in popular open-source ML frameworks, such as Pytorch, ONNX, and Tensorflow, to low-level codes for CGRAs. To achieve globally automatic compilation, we implement CGRA-Opt and CGRA-Runner, the key components of ML-CGRA, which bridge the programmability and mapping gap between the ML models and CGRAs targets. CGRA-Opt contains a set of lowering and optimization passes to enable the efficient mapping of ML models on CGRAs that achieve superior speedup. Conventional CGRA compilation focuses on per-kernel mapping by lowering it into LLVM IR targeting minimized II, which loses the higher-level information of ML operations. In contrast, ML-CGRA keeps such information and enables, when possible, direct offloading of the ML operations at a higher level of abstraction. In addition, ML models always include computational patterns that can be identified by the compiler and mapped onto pre-configured and optimized accelerator templates such as presented in Figure 4. Finally, to help us develop accelerators and unbiasedly evaluate the framework effectiveness, we introduce CGRA-Runner, which

provides the runtime library and the interfaces to facilitate cycle-level and RTL simulation. In the following sections, we introduce the implementation details of the proposed framework.

4.2.1.1 CGRA-Opt

We created a cgra-dialect and transformation passes to build CGRA-Opt. The transformation passes include three important optimizations (i.e., Operation Fusion, Loop Tiling, and Pattern Matching) and Offloading.

Operation Fusion - In traditional loosely coupled accelerators, including CGRAs, data movement is time-consuming and power-hungry. This is an overreaching issue. To overcome this problem and accelerate computation ML-CGRA leverages the `--linalg-fuse-elementwise-ops` pass to fuse element-wise operations on tensors. This improves the utilization of the compute resources of CGRA and minimizes the data movement between the fusible elementwise operations. As an example, Figure 5 stage 5 presents a detailed fusion for three operations including *addf, maxf, and addf*, during which the fusion number can be self-modified.

Loop Tiling - As one of the most precious resources in hardware platforms, on-chip memory/registers are limited in CGRAs, similar to other hardware. To maximize their utilization, we implement a loop tiling pass to tile the input/output tensors based on the size of the input/output buffers. Besides, to further improve framework efficiency, we apply the double-buffering technique on ML-CGRA, as an option, which can minimize the delay in data communication.

Pattern Matching - Mapping a tensor-based ML operation to the CGRA in a specific way can often outperform the conventional mapping. Thus, we implement a pattern-matching pass to recognize the patterns/operations that the pre-configured CGRA templates can directly accelerate without lowering into LLVM-IR. Note that users can easily add any patterns by using this pass. To be specific, the CGRA can be configured as shown in Figure 4 to perform efficient acceleration on the fused pattern *mul-relu-add*. Then, all the *mul-relu-add* operations in the given ML model can be recognized by pattern matching pass, which avoids lowering them into the LLVM-IR to perform the conventional kernel. Similarly, *matmul* operations can be recognized by the pattern matching pass to be efficiently accelerated by configuring the CGRA as a systolic array implementing an efficient matrix multiplication. Note that not all operations/patterns have to be accelerated with a pre-configured CGRA mapping. ML evolves often, and new patterns from an upcoming ML model can always be lowered into LLVM-IR to perform the conventional mapping for CGRA acceleration. As shown in the last code snippet of Figure 5 if the user does not allow the fused operations to be pattern matched, it will be lowered as a function call of `cgra.conv_mapping`, which invokes the conventional DFG mapping.

Offloading - Once the above passes are complete, an offloading pass will lower each operation into a CGRA function call (i.e., Accelerator Cmd as shown in Figure 5 stage 8). The CGRA runner can also use the CGRA function call to evaluate their execution performance.

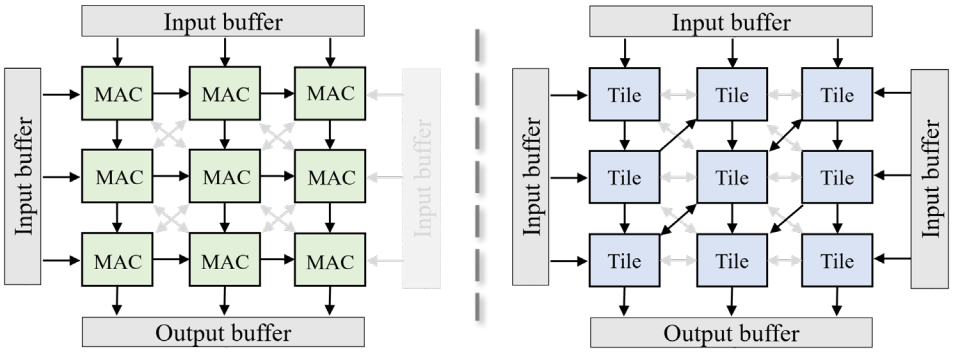

Figure 9: Pre-configuring the CGRA as a optimal spatial accelerator for a specific ML operation -- (a) Configuring the CGRA as a systolic array for *matmul*. (b) Optimized configuration for accelerating fused element-wise operations.

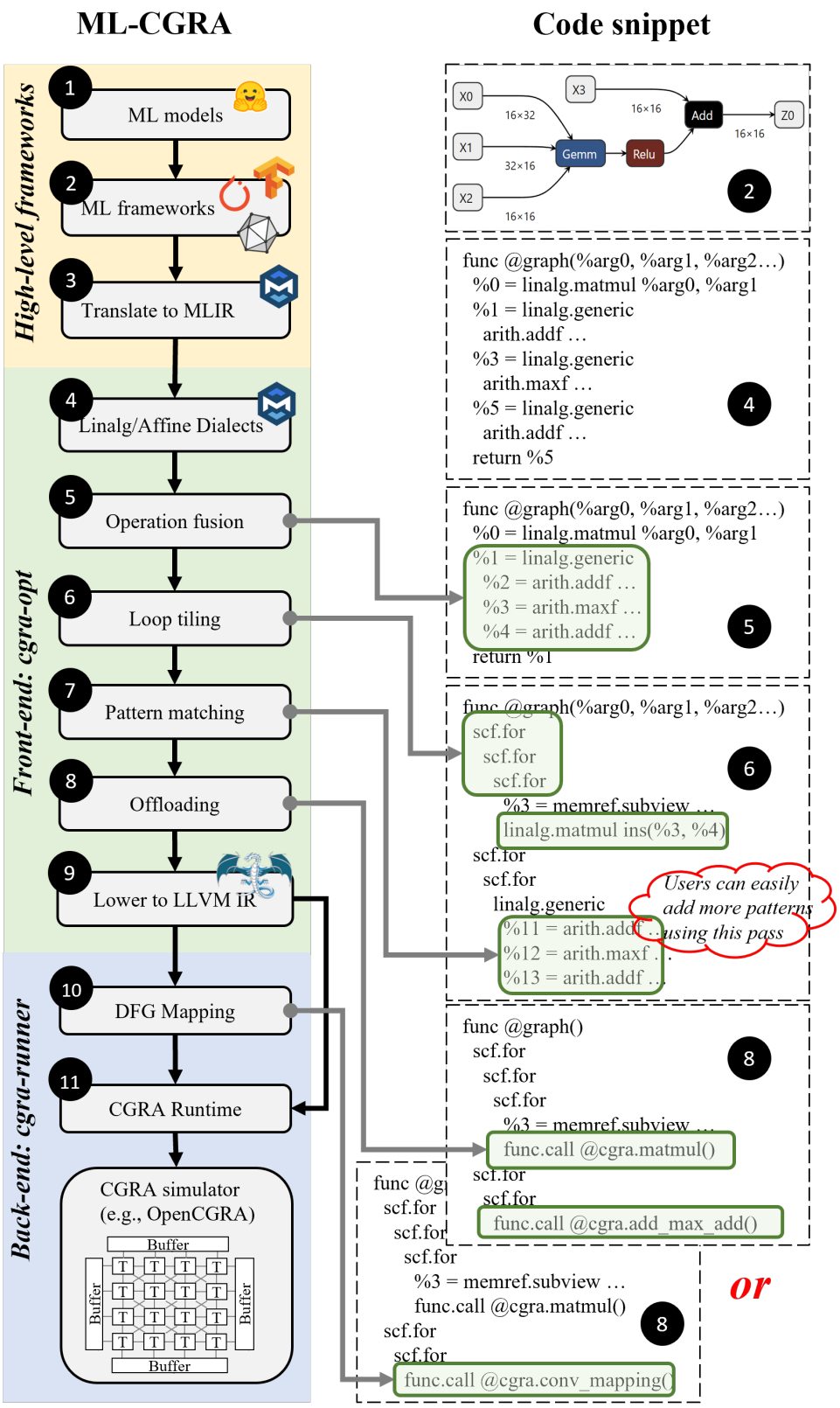

Figure 10: Overview of ML-CGRA Framework with the corresponding code snippets after certain stages -- The fused operations (at stage 8) can be lowered into the function call of

cgra.add_max_add() or *cgra.conv_mapping()* depending on whether a a pre-configured CGRA is provided by the user. *cgra.conv_mapping()* here stands for conventional mapping on CGRA.

4.2.1.2 CGRA-Runner

To support the full end-to-end design flow for specializing and implementing CGRAs, researchers proposed several CGRA-based simulators (e.g. OpenCGRA and CGRA-ME). These simulators model a CGRA at different levels of abstraction, provide compiler support, verification at different granularities, simulation, generation of synthesizable Verilog, and characterization. Thanks to those simulators and their interfaces, we can transform our CGRA function calls generated by the offloading stage to the simulator's runtime library, allowing us to achieve cycle-level and RTL simulation and obtain the exact execution time for performance evaluation.

4.2.2 Evaluation

To illustrate the potential of ML-CGRA, we evaluate a set of state-of-the-art machine learning models from diverse domains accelerated by CGRAs. We compare the performance (i.e., execution time running on the same CGRA) of the code generated by ML-CGRA against the conventional DFG mapping methodology. We also discuss how the proposed framework adapts to CGRAs of different dimensions, to ensure its applicability to larger devices.

4.2.2.1 ML Models

Computer vision and natural language processing are key applications of ML models. A key component of these models is the attention mechanism. The computational patterns and flow of data across computations of attention mechanisms perfectly match the features provided by a CGRA. Table 1 presents detailed information about the attention-based models used in our evaluation.

Table 1. Detailed information of the target machine learning models -- the models are from computer vision and natural language domains.

Model Name	Domain	# Attention Heads	# Layers	Parameters
MiniLM	Text Classification	12	6	66M
Sentence-BERT	Sentence Similarity	12	12	120M
CamemBERT	Token Classification	12	12	110M
Vision Transformer	Image Classification	12	12	86M

4.2.2.2 Experimental Results

Given an ML model provided by hugging face, we can compile it to the *Linalg* MLIR dialect through Torch-MLIR. The *Linalg* representation of the model serves as the input to ML-CGRA. ML-CGRA can apply several optimization passes to generate the optimal code for a target CGRA configuration. We implement a cycle-accurate CGRA simulator to obtain the execution times in terms of cycles. Note that the simulator exploits the CGRA runtime to implement the CGRA-Runner.

Baseline - We consider the conventional low-level mapping methodology of ML models on a 4x4 CGRA as the baseline. We compute data communication overheads considering a typical direct memory access (DMA) controller running at 200 MHz. We employ input buffers of 1 KB for 4x4 CGRAs and of 4 KB for 8x8 CGRAs.

Speedup - Figure 7 shows the normalized speedups of ML-CGRAs over baseline implementation across 4 different models. To further break down the benefits from different optimizations applied in ML-CGRA, we compare baseline with 4 implementations including (1) Baseline + Fusion, (2) Baseline + Fusion & Double Buffering, (3) ML-CGRAs equipped with all optimizations (including Fusion and pre-configured CGRAs) but double buffering, and (4) ML-CGRAs, i.e. implementation (3), with double buffering. The results demonstrate that the full-version ML-CGRA (implementation (4)) is able to fuse operations automatically and accelerates specific kernel with predefined optimized patterns to obtain superior performance, leading to 3.15x over the baseline and better realizing the potentials of CGRA architecture in ML tasks. Furthermore, the results also show that the double buffering technique brings comparable speedups for both baseline (1.5x from implementation (1) to (2)) and ML-CGRA (1.65x from implementation (3) to (4)). For implementation (1) where the baseline is augmented with operation fusion, although operation fusion significantly reduces the data movement, most of the models are still bounded by the computation loads of matrix multiplications, which leads to on average only 3% improvement in terms of speedup across all the models.

Scalability - We also evaluate the scalability of ML-CGRA by deploying the models onto different sizes of CGRAs (e.g. 4x4 and 8x8). As Figure 7 shows, ML-CGRA framework can effectively accelerate machine learning algorithms on 8x8 CGRAs and obtain about 90.17 % (123.61% with double-buffering enabled) improvement on speedup compared with 4x4 CGRAs, which is reasonable as the system quickly changes to communication-bound from compute-bound when the hardware resources increase by 4 times. Existing tiling pass in ML-CGRA targets output-stationary. However, ML-CGRA is built on top of MLIR, which makes it easier to plug in with open-source tiling/data-movement passes to maximize the data reuse and potentially improve the scalability further.

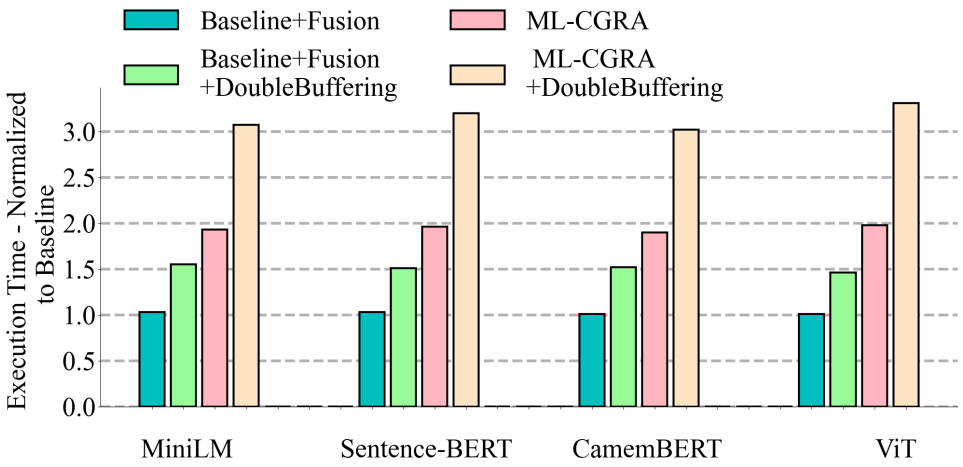

Figure 11. Normalized acceleration speedup of different optimizations across models with respect to the baseline -- The baseline leverages the conventional kernel mapping for models on a 4x4 CGRA.

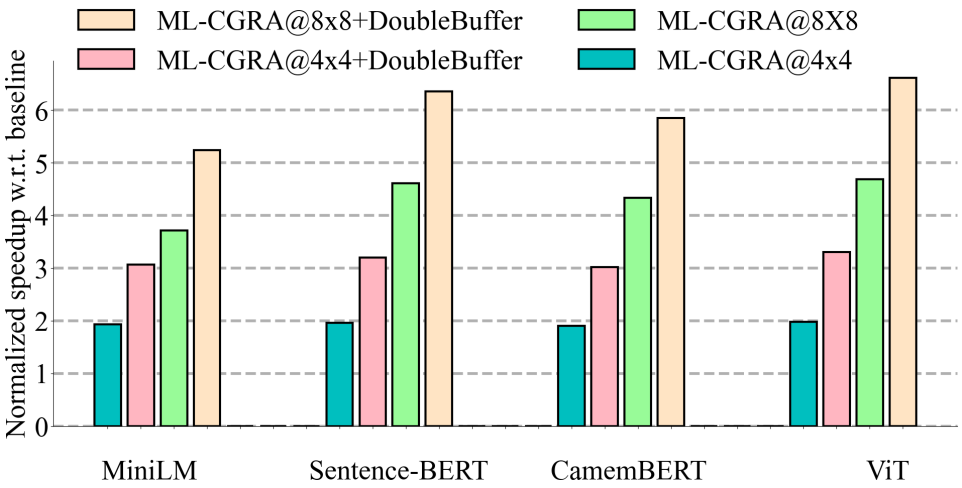

Figure 12. Normalized speedup for models running on different CGRAs -- The baseline leverages the conventional kernel mapping for models on a 4x4 CGRA. Each input/output buffer is 1KB and 4KB in the 4x4 and 8x8 CGRAs, respectively.

4.2.3 Final considerations

This section introduced ML-CGRA, an open-source integrated compilation framework to enable efficient machine learning implementation and acceleration on CGRAs. Under ML-CGRA, users can effectively compile machine learning models from high-level frameworks to optimized low-level codes for CGRAs. Compared with a conventional CGRA implementation for machine learning kernels, ML-CGRA provides an end-to-end solution to specifically accelerate machine learning models with pre-configured CGRAs and to improve on average the execution performance for a diverse set of models, respectively.

5.0 References

[1] N. B. Agostini, S. Curzel, V. Amatya, C. Tan, M. Minutoli, V. G. Castellana, J. Manzano, D. Kaeli, and A. Tumeo. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In IEEE/ACM International Conference on Computer-Aided Design, ICCAD'22, San Diego, CA, 2022. IEEE.

[2] S. Ahmad, S. Subramanian, V. Boppana, S. Lakka, F. Ho, T. Knopp, J. Noguera, G. Singh, and R. Wittig. Xilinx First 7nm Device: Versal AI Core (VC1902). In 2019 IEEE Hot Chips 31 Symposium (HCS), pages 1–28, Los Alamitos, CA, USA, aug 2019. IEEE Computer Society.

[3] K.Ando, S.Takamaeda-Yamazaki, M.Ikebe, T.Asai, and M.Motomura. A multithreaded CGRA for convolutional neural network processing. Circuits and Systems, 8(6):149–170, 2017.

[4] I. Bae, B. Harris, H. Min, and B. Egger. Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(11):2301–2310, 2018.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association.

[6] 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), pages 184–189. IEEE, 2017.

[7] W. C. Developers. Wave Computing. https://wavecomp.ai.

[8] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An Image is Worth 16x16 Words: Trans- formers for Image Recognition at Scale. In International Conference onLearning Representations, 2021.

[9] 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 Ar- chitecture. Computing in Science & Engineering, 23(2):114–119, 2021.

[10] X. Fan, D. Wu, W. Cao, W. Luk, and L. Wang. Stream processing dual- track CGRA for object inference. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 26(6):1098–1111, 2018.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In CGO, pages 2–14, Seoul, South Korea, 2021. IEEE.

[12] J. Lee and J. Lee. NP-CGRA: Extending CGRAs for Efficient Processing of Light-weight Deep Neural Networks. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1408–1413. IEEE, 2021.

[13] M. Liang, M. Chen, Z. Wang, and J. Sun. A CGRA based neural network inference engine for deep reinforcement learning. In 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), pages 540–543. IEEE, 2018.

[14] L. Martin, B. Muller, P. J. O. Suarez, Y. Dupont, L. Romary, E. V. de la Clergerie, D. Seddah, and B. Sagot. CamemBERT: a Tasty French Language Model. CoRR, abs/1911.03894, 2019.

[15] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 304–315, 2019.

[16] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In MICRO'94, 1994.[17] N. Reimers and I. Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. CoRR, abs/1908.10084, 2019.

[18] C. Tan, N. B. Agostini, T. Geng, C. Xie, J. Li, A. Li, K. J. Barker, and A. Tumeo. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pages 304–316. IEEE, 2022.

[19] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In 2020 IEEE 38th International Conference on Computer Design (ICCD), pages 381–388. IEEE, 2020.

[20] 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), pages 1388–1393. IEEE, 2021.

[21] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima. A cgra-based approach for accelerating convolutional neural networks. In 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, pages 73–80. IEEE, 2015.

[22] TVM Developers. VTA: Deep learning accelerator stack, 2020.

[23] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou. MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. CoRR, abs/2002.10957, 2020.

[24] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA),pages 268–281. IEEE, 2020.

[25] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang. HASCO:Towards Agile hardware and software co-Design for Tensor Computa- tion. In Proceedings of the 48th Annual International Symposium on Computer Architecture, ISCA '21, page 1055–1068. IEEE Press, 2021.

[26] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In

IEEE International Symposium on High-Performance Computer Architecture, HPCA'22, pages 741–755, Seoul, South Korea, 2022. IEEE.

[27] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen. HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation. In 57th ACM/IEEE Design Automation Conference, DAC'20, pages 1–6, San Francisco, CA, USA, 2020. IEEE.

[28] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen. DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator. In IEEE/ACM International Conference on Computer-Aided Design, ICCAD'20, pages 1–9, San Diego, CA, USA, 2020. IEEE.

[29] L. Zulberti, M. Monopoli, P. Nannipieri, and L. Fanucci. Architectural Implications for Inference of Graph Neural Networks on CGRA-based Accelerators. In 2022 17th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), pages 373–376. IEEE, 2022.

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354

1-888-375-PNNL (7665)

www.pnnl.gov