ScaleCUDA: A New GPU Programming Framework on Multi-Level Intermediate Representation through ScaleHLS

Abhi Kamboj*
akamboj2@illinois.edu
University of Illinois, at Urbana-Champaign

Armandeep Singh*
as46@illinois.edu
University of Illinois, at Urbana-Champaign

Abstract

Graphics Processing Units (GPUs) and manycore processors in general are some of the most important and powerful tools in modern computing as their ability to massively parallelize computations is an excellent way to accelerate computationally intensive programs. Due to the massive presence of GPUs in heterogeneous systems, the large design space of parallel programming, and the substantial variance in performance between low and high end implementations of GPU kernels, effectively developing this software is a critical challenge. The development and optimization process can be automated. Successful projects in this area take advantage of the affine properties of the input functionality as identified by polyhedral models in order to output optimized kernels for manycore processors.

This paper proposes ScaleCUDA, a tool for automated GPU code (CUDA) generation and optimization. ScaleCUDA looks to take advantage of two essential components of high end optimization: identification of affine properties through polyhedral modelling and analysis at multiple levels of abstraction. It will do so by way of ScaleHLS[7], a high-level synthesis (HLS) design space exploration tool. ScaleHLS uses MLIR, a multi-level compiler infrastructure, in order to effectively explore the design space at various levels of abstraction and create an optimized HLS design. To translate C++ or HLS C++ input into the desired optimal CUDA output, ScaleCUDA bridges the gap between the polyhedral and CUDA representations in MLIR while taking advantage of the HLS optimization and design space exploration tools of ScaleHLS.

We test the pipeline with GEMM C++ code and show similar performance to code directly compiled and optimized through NVIDIA cuda frameworks.

Our implementation is opensourced and can be accessed on the ScaleCUDA branch here: [1]. Our experimentation code is also provided here: [2].

Keywords: GPU, manycore, optimization, MLIR, ScaleHLS

ACM Reference Format:
Abhi Kamboj and Armandeep Singh. 2025. ScaleCUDA: A New GPU Programming Framework on Multi-Level Intermediate Representation through ScaleHLS. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation

The goal of ScaleCUDA is to provide a framework for quickly developing powerful GPU software that approaches the achievable GPU performance ceiling. This is possible based on the success of the previous project, Polyhedral Parallel Code Generation (PPCG)[6], which was able to use polyhedral representations of C++ code to generate CUDA kernels which compete with high end and manually optimized libraries as seen in Figure 1.

The effectiveness of MLIR as an optimizer gives reason to believe that an MLIR-based implementation of the polyhedral conversion concept can be even more effective than the original version. Frameworks such as Pytorch, a python library for machine learning, interfaces directly with NVVM to develop CUDA binaries. However, this direct link bypasses many dialects in the MLIR framework that could be leveraged to optimize the output which ScaleCUDA plans to utilize.

A few previous works have attempted to leverage MLIR to optimize code on a GPU, however, none have integrated an HLS flow and focused on HLS based performance optimizations (e.g. loop perfectization, loop reordering, etc.) as ScaleCUDA intends to do. IREE [5] focuses on end to end optimizations for embedded computational systems. Furthermore, [2] focuses on low level MLIR matrix multiplication acceleration using the warped matrix multiplication operation (wmma).

*Both authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://github.com/akamboj2/scalehls/tree/scalecuda
[2] https://github.com/akamboj2/CuBLAS_ScaleCUDATesting

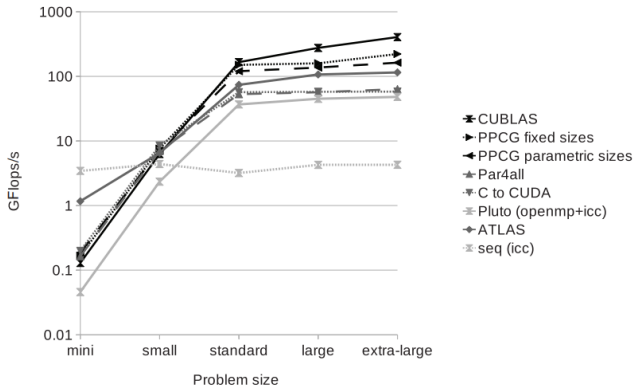

Figure 1. PPCG Performance:[6] This graph shows a comparison of optimized code on GPUs for exsiting works. CUBLAS, a framework for linear algebra made by nvidia compiles code directly onto an NVIDIA GPU, thus indisputably sets the performance ceiling for GPUs. Nonetheless, PPCG is able to reach very similar performance. ScaleCUDA aims to perform similarily well through the MLIR framework.

Furthermore, using MLIR will allow ScaleCUDA to take advantage of the existing optimization and design space exploration functionalities developed in ScaleHLS. Even if ScaleCUDA is not able to top highly refined implementations in terms of performance, the much faster development cycle would make ScaleCUDA a useful tool for two reasons. Firstly, the fast turn around time coupled with near-optimal performance would be very helpful for design space exploration as a very accurate heuristic/evaluation of the GPU for a given task. Secondly, developers who are not knowledgeable on the niche and minute optimizations of CUDA would be able to access their benefits without the required time and learning.

1.2 Objective

As a GPU programming framework, ScaleCUDA is aiming to generate CUDA implementations which compete with the premier CUDA libraries such as cuBLAS or cuDNN. ScaleCUDA should also top the preceding project, PPCG. This paper also looks to demonstrate the effectiveness of the ScaleHLS optimization and design space exploration features.

1.3 Contributions

We extend ScaleFlow by adding a CUDA emitter. Specifically, we interface the Affine and NVVM dialects through the MLIR framework. Our contribution is the MLIR translation between these dialects. The important contributions ScaleCUDA will reference are ScaleHLS, PPCG, MLIR as a whole, the Affine dialect, the NVVM and NVGPU dialects, and Polygeist. It is the interface between these contributions that ScaleCUDA focuses on.

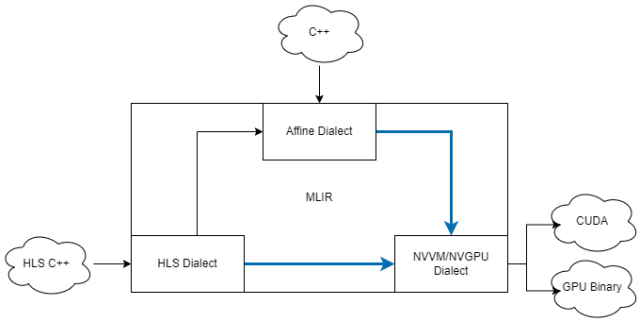

Figure 2. Basic Block Diagram of ScaleCUDA within MLIR: The black lines and boxes represent preexisting components in the MLIR framework. The blue arrows represent ScaleCUDA proposed contributions. Polygiest translates from C++ to the Affine dialect. ScaleHLS translates from HLS C++ to the HLS dialect and the Affine dialect. ScaleCUDA's current implementation lowers the Affine dialect to the NVVM GPU Dialect and future extensions may convert from HLS to NVVM.

2 ScaleCUDA Overview

ScaleCUDA takes advantage of many existing utilities surrounding MLIR and revolves around a new link between different components. MLIR[3] is a framework designed to build scaleable and modular compiler systems using dialects to communicate between intermediate representations (IRs). Figure 2 shows how ScaleCUDA fits into the existing framework.

We leverage the existing utilities ScaleHLS and Polygeist[4] to complete the ScaleCUDA framework. Polygeist determines a polyhedral representation of input C++ and outputs it into the Affine dialect. On the other hand, ScaleHLS provides an HLS IR which can be translated into the Affine dialect within MLIR. MLIR also contains two GPU-oriented dialects: NVVM and NVGPU[1]. These two dialects can be compiled into deployable CUDA kernels.

ScaleCUDA implements a translation between the Affine dialect and the GPU NVVM dialects, completing the pathways for both HLS C++ and C++ to be optimized and translated into GPU compatible code. The Affine dialect specifically is very useful because it uses a polyhedral representation which is a critical tool for exposing parallelisms.

3 Methodology

The translation between dialects in MLIR consists of a series of "passes" and "pass pipelines". Passes traverse the input code at a desired depth and mutate in some way. Pass pipelines are collections of passes applied sequentially [3].

The translation between Affine and GPU is also performed as a pass. The existing pass which performs this translation is called convert-affine-for-to-gpu. This pass converts affine for loops into dimensions for a GPU kernel, and converts the

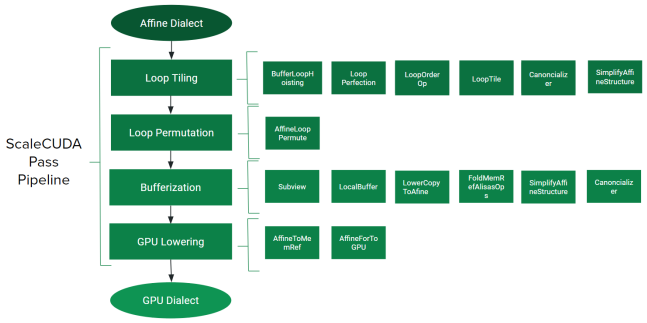

Figure 3. ScaleCUDA Pipeline Diagram: This block diagram shows the entire ScaleCUDA pipeline. ScaleCUDA lowers the Affine dialect to the GPU dialect leveraging existing ScaleHLS passes and custom passes.

contents of the loops into the kernel code. While this is an important idea for this translation, this pass alone is rather naive. It requires manual input of the dimensionality of the GPU kernel and also applies no optimizations.

ScaleCUDA looks to utilize the convert-affine-for-to-gpu pass and build a pass pipeline around it, adding passes to increase functionality and further optimization. Implementing this pass will complete the chain of translations within MLIR to take some HLS C++ or C++ input and convert it into optimized GPU software.

The first step is to use ScaleHLS or Polygeist to generate Affine dialect code. Next, the ScaleCUDA pipeline is applied to produce GPU dialect output. From the GPU dialect there are multiple routes the user can take to produce binaries for GPUs. We test and report the results of one of these routes as described in the experiments section. The overall ScaleCUDA pipeline is shown in Figure 3.

3.1 AffineLoopPermute

The first custom pass created for the ScaleCUDA pipeline is the AffineLoopPermute pass. One of the biggest insufficiencies of the convert-affine-for-to-gpu pass is its naievete. The pass does not consider memory depedencies or true parallelizabilities of affine loops.

For example in a matrix multiplication, as shown in the Affine IR code[3], the innermost loop is not parallelizable because of the sequential addition to the same output, however it is categorically and technically affine. So, the convert-affine-for-to-gpu pass will distribute this inner loop among different threads in the GPU without second thought, resulting in a large set of race cases and inaccuracies.

The AffineLoopPermute pass we created addresses this issue in 3 main steps:

1. It first iterates over all instructions looking for nested for loops and creating a list of affine for loops.
2. Next, it iterates over this list of affine for loops, checking for memory dependencies and parallelizability of each loop, marking each loop as either parallel or not and keeping a tally of the number of parallelizable loops.
3. Lastly, it generates a loop permutation map such that the truly parallel loops are the outermost.

Having the parallelizable loops on the outside of the nest ensures that when the conversion to a GPU grid and kernel takes place, the dependent loops are preserved within the kernel while the parallel loops are the ones split up and parallelized. The tally of parallel loops can be fed into the convert-affine-for-to-gpu pass as arguments to ensure the sequential loops are left untouched.

3.2 ScaleCUDA Pipeline

The ScaleCUDA Pipeline as a whole is framed around the AffineLoopPermute and convert-affine-for-to-gpu passes. Between the two, it is possible to create a functional GPU output given an Affine dialect input, but not a high performing output. ScaleCUDA utilizes optimizations provided by ScaleHLS to enhance the generated software. Specifically, the ScaleHLS pipelines for Loop Tiling and Local Buffers are employed.

3.2.1 Loop Tiling. Loop tiling is an important feature because it allows for its user to take advantage of temporal and spacial locality. Temporal and spacial locality are especially critical for GPUs because of the multi-layer cache system found in GPUs. Specifically, the shared memory between cores on the same SM can be fully utilized with tiling. In fact, loop tiling is considered one of the primary tools in the GPU/CUDA developers toolbox in terms of optimization.

The loop tiling ScaleHLS pipeline is applied before the AffineLoopPermute pass. This way, the parallelizable tiles can be distributed as thread blocks in the GPU's thread grid. In other words, each SM will represent a tile and the cores in the SM will each compute one entry within the tile. This design of matching tiles to thread blocks is commonly utilized by GPU Programmers, and ScaleCUDA is able to generate this automatically.

3.2.2 Local Buffers. In order to maximize the loop tiling paradigm's performance, the kernel must load the relevant information of the tile into shared memory, so all cores in the SM (entries in the tile) can reuse data loaded in from global memory. High memory reuse is highly sought after in GPU kernel development.

The ScaleCUDA pipeline takes advantage of ScaleHLS' local buffer passes. The local buffer pass pipeline is applied to the permuted and tiled affine loops. However, ScaleCUDA employs a mechanism to ensure the local buffers are only created for the non-parallelizable loops. This means that the local buffers are only present within the kernel deployed on the GPU - creating buffers in the outer loops which are

[3] https://github.com/akamboj2/scalehls/blob/scalecuda/samples/polybench/gemm/test_gemm_mult_out.mlir

distributed across the GPU would not make sense. These local buffers are then converted into instantiations of shared memory within the GPU, maximizing data reuse.

3.3 Experiments

We conduct experiments on two sets of GEMM C++ code on an NVIDIA 3080 RTX GPU. First simple matrix element-wise addition and second matrix multiplication. Although these are very basic operations, they are fundamental to deep networks and modern day computing application, so being able to efficiently perform them is vital.

The experiments are conducted on each matrix operation code as follows:

1. Compile the GEMM C++ code to an Affine representaiton in MLIR using Polygeist
2. Pass the Affine representation through the ScaleCUDA pipeline includinhg:
   a. Existing scaleHLS loop optimizations (perfectization, ordering, tiling, etc.)
   b. Our custom passes (i.e. AffineLoopPermute pass)
   c. Local buffer allocation passses
   d. Existing GPU passes (e.g. AffineForToGPU)
3. Exit the GPU MLIR code through an existing GPU to executable pipeline from the buddy-mlir repo [4]
4. Time the previous step to deduce latency and performance and compare to CuBLAS and existing cuda optimizations.

The intermediate results of can be viewed on our offical Github Repo. For example, the ScaleCUDA optimized addition can be seen at https://github.com/akamboj2/scalehls/blob/scalecuda/samples/polybench/gemm/test_gemm_add_out.mlir and multiplication at https://github.com/akamboj2/scalehls/blob/scalecuda/samples/polybench/gemm/test_gemm_mult_out.mlir

4 Results

4.1 Matrix Addition

As described in 3.3 we perform 5 trials of element-wise matrix addition on 3 different optimization frameworks: ScaleCUDA (ours) and CuBLAS.

CuBLAS [5] is an implementation of Basic Linear Algebra Subprograms (BLAS) on top of the NVIDIA CUDA runtime. It is a handwritten library used to optimize C code directly on a GPU. We write a simple matrix addition code using this library to compare against ScaleCUDA.

The results shown in Figure 4 demonstrate that ScaleCUDA performs slightly better than the existing baselines, however it has a slightly larger variance. This implies that the performance is not very consistent. We are skeptical of these results as the handwritten CuBLAS library should perform better than our MLIR lowering of matrix addition. We expound on this further on this in Section 5.

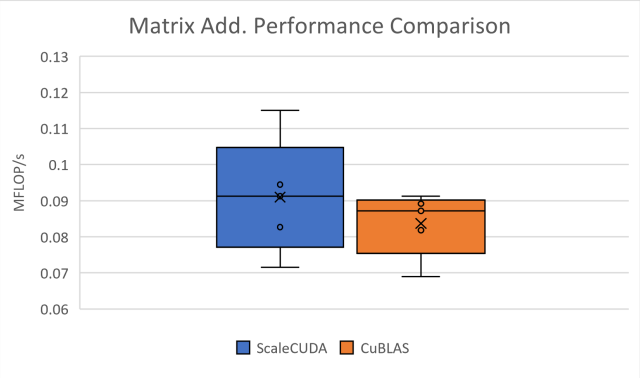

Figure 4. ScaleCUDA Matrix Addition Performance: The box plot above shows a comparison of performance of ScaleCUDA and CuBLAS. ScaleCUDA performs marginally better, however, with a higher variance.

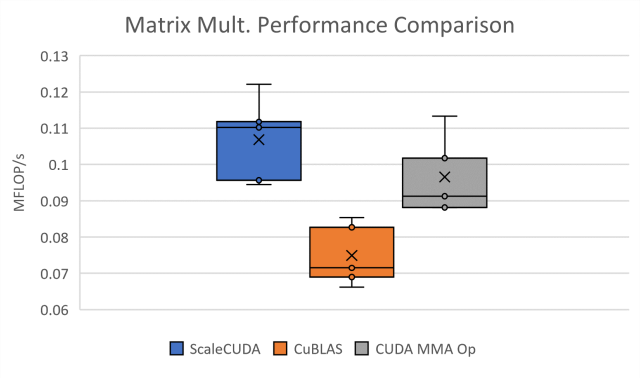

Figure 5. ScaleCUDA Matrix Multiplication Performance: The box plot above shows a comparison of performance of ScaleCUDA, CuBLAS, and the CUDA MMA Op across 5 matrix multiplication trials for each method. ScaleCUDA shows a slightly higher performance in this simple metric.

4.2 Matrix Multiplication

As described in 3.3 we perform 5 trials of matrix multiplication on 3 different optimization frameworks: ScaleCUDA (ours), CuBLAS, and using the CUDA MMA operation in MLIR.

Similar to the previous section using CuBLAS we write a simple matrix multiplication code using this library to compare against ScaleCUDA. However, for matrix multiplication the NVVM dialect has a an inbuilt function to perform

[4] https://github.com/buddy-compiler/buddy-mlir
[5]

mma (matrix-multiply accumulate) and wmma (warped matrix multiply and accumulate) [6]. We use the buddy-mlir's gpu-mma.mlir [7] which utilizes the nvvm mma operation as another baseline to compare our performance against.

The results shown in Figure 5 are also similar to the previous section ScaleCUDA performs slightly better than the existing baselines, however it has a slightly larger variance. This implies that the performance is not very consistent. We are skeptical of these results as the hand designed CuBLAS library should perform better than our MLIR lowering of matrix multiplication. We expound on this further on this in the next Section 5.

5 Limitations

Due to time constraints we were not able to get the ScaleCUDA pipeline to generate and run the code on the GPU. Thus, after generating the GPU dialect output, the code was manually modified to function with the buddy-mlir's GPU execution pipeline. When attempting to integrate GPU execution directly into our pipeline we received various MLIR errors and despite engaging with MLIR community experts, we were unable to resolve the errors within the project's time frame.

Although the main computational enhancements (loop permutation, tiling, bufferization) was not altered, the modifications to be able to run our optimized code on the GPU could have unintentionally changed the performance results. Nonetheless, we believe the reported results show a relatively accurate (but weak) representation of how our pipeline compares with optimal CUDA matrix multiplication. More experiments and development must be done before strong conclusions should be drawn from this data.

Furthermore, this was tested on medium to small matrices (16x16, 256x256, 8192x8192). This is why the results are on the order of MFLOPS in Figures 5,4 as opposed to GFLOPS as shown in previous works (Figure 1). Results may vary for larger matrices, as the GPU may have more parallelization power and this should be further investigated. This is also potentially the reason why our framework performed better than the CuBLAS library which theoretically is likely to be an upperbound on our performance.

Despite the lack of experiments, visual inspection shows the potential of the ScaleCUDA framework.

6 Future Work

There are many potential extensions that can be made to the ScaleCUDA framework, aside for more rigorous experimentation and completion of the current proposed framework as described in Section 5.

One direct extension of ScaleCUDA is being able to support more operations, e.g. convolutions. Theoretically, convolutions are a series of for loops performing dot products, so the current framework should be able to optimize them fairly well, however, at first attempt ScaleCUDA was running into errors when optimizing a convolution, thus more effort needs to be focused into this extension.

Another, potential extension of ScaleCUDA is optimizing shared memory resources amongst threads on a GPU. If the same data needs to be used on multiple GPU threads that data should be stored in a shared memory buffer that multiple threads can access. A ScaleCUDA extension should be able to recognize which data should be shared and create shared buffers accordingly.

7 Conclusions

Overall, state-of-the-art deep learning performance on a GPU is often driven by hand tuned highly optimized libraries. This hinders scalability and limits the modularity and reusability of that optimized code. MLIR is an intermediate representation infrastructure built on the LLVM compiler infrastructure to promote code portability and optimality across different languages and devices.

We use MLIR to develop ScaleCUDA, a pipeline aimed at leveraging the MLIR infrastructure to optimize code on a GPU. We use the existing ScaleHLS and Polygiest frameworks to create and test ScaleCUDA. Our rough preliminary results on element-wise matrix addition and matrix multiplication demonstrate ScaleCUDA's potential as a powerful design space exploration and optimization tool. We believe that these results may motivate future research and development on automatic code generation for GPU code optimization using an IR infrastructure.

References

[1] Vinod Grover and Yuan Lin. 2012. Compiling CUDA and other languages for GPUs. In *GPU Technology Conference (GTC)*. 1–59.

[2] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-based code generation for GPU tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 117–128.

[3] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[4] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.

[5] The IREE Authors. 2019. *IREE*. https://github.com/iree-org/iree

[6] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[6] https://mlir.llvm.org/docs/Dialects/NVVMDialect/#nvvmwmmamma-mlirnvvmwmmammaop

[7] https://github.com/buddy-compiler/buddy-mlir/blob/main/examples/MLIRGPU/gpu-mma.mlir

[7] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009