COALDA - Compiler Optimization for CUDA Memory Coalescing

Yuxiang Chen, Hanlin Bi, Ziang Li, Qifa Wang, Zihao Ye

***Abstract*—In the realm of CUDA (Computer Unified Device Architecture) programming, efficient memory access is crucial for achieving high performance. However, developers commonly face the challenge of uncoalesced memory access, where data is not efficiently organized for parallel retrieval, leading to significant underutilization of memory bandwidth. This issue not only slows down data transfer between host and the GPU but also hampers the overall computational efficiency. Recognizing this challenge, COALDA (Compiler Optimization for CUDA Memory Coalescing) is introduced as an innovative solution. It aims to optimize memory access patterns by reorganizing uncoalesced accesses into coalesced forms. This transformation is crucial for enhancing the performance of CUDA applications, making them more efficient and effective in handling large-scale data processing taks. Through the application of advanced compiler techniques and leveraging the LLVM framework, COALDA presents a promising approach to tackle this pervasive issue in CUDA programming.**

I. Introduction

THE evolution of NVIDIA GPUs and the CUDA(Computer Unified Device Architecture) programming model has been pivotal in advancing the fields of parallel computing and deep learning. NVIDIA GPUs, with their highly parallel architecture, are especially designed to handle the demanding computational tasks efficiently. CUDA, as a parallel computing platform and application programming interface model, leverages this architecture, enabling dramatic increase in computing performance by harnessing the power of GPU.

A. CUDA Programming Model

As illustrated by figure 1, the CUDA programming model is a finely orchestrated system designed to leverage the parallel processing capabilities of NVIDIA GPUs. At the top level, the Grid represents the entire computation, a collection of thread blocks that execute a given kernel. Each Block is a group of threads that can be scheduled and executed concurrently. Threads within a block can share data effectively via the shared memory, a limited but fast memory pool.

This memory hierarchy is designed to provide a balance between the large data storage capacity and the high-speed data access requirements of complex parallel computations. Optimizing the use of this hierarchy, such as maximizing the use of registers and shared memory while minimizing reliance on global memory, is critical for achieving high performance in CUDA applications. The CUDA model also supports synchronization primitives to coordinate the execution of threads, ensuring correct program execution.

Discord moderators and members of the LLVM and NVIDIA community

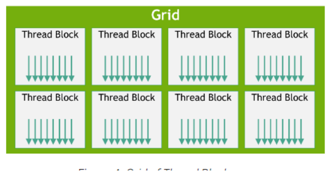

Fig. 1. CUDA Programming Model

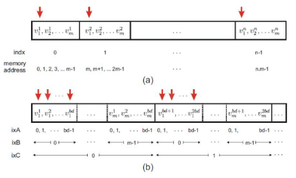

Fig. 2. Memory Coalescing

B. SIMT and Memory Coalescing

The GPU functions like a SIMT (Single Instruction Multiple Thread) machine, where a warp (usually 32) of threads execute the same code using. When a thread in a warp access consecutive memory addresses, the hardware can combine these accesses into a single memory transaction, greatly increasing badnwith utilization. The hierarchical arrangement of threads into warps, blocks, and grid is thus closely tied to memory coalesing. Properly aligning data structures and access patterns with this hierarchy ensures that memory transactions are coalesced, elading to optimized performance on the GPU. Non-coalesced accesses, where threads access scattered memory locations, results in multiple transactions, reducing performance. Therefore, undersatnding the designing with the CUDA memory hierarchy in mind is crucial for achieving memory coalescing and maximizing the efficiency of CUDA applications.

A good example is illustrated in figure 2. Suppose n vectors of length m are stored in a linear fashion. Element i of vector j denoted as V_j^i Each thread in the GPU kernel is assigned to a vector of length m Threads in CUDA are grouped in an array of blocks and every thread in GPU has a unique id which can be identified as $global\_idx = bd * bx + tx$, where bd is block dimension, bx is block index, and tx is thread index within each block.

In the case where we want to access the first components of each vector, i.e address $0, m, 2m, ...$ of the memory, a lot of store and load would need to be performed by the GPU kernel because the size of each load only guarantees one such address. Assuming that memory is accessed in chunks of m

addresses, to load address 0, the first m addresses needs to be loaded but only the one is needed. To load address m, the second m addresses need to be loaded but only the first one is needed, etc.

The solution is to store data in the following fashion: store the first element of the first bd vectors in consecutive order, followed by the first element of the second bd vectors and so on. The rest of the elements are stored in a similar fashion. If n is not a factor of bd, we need to pad the remaining data in the last block with some trivial values like 0.

In the linear data storage described above, component $i(0 <= i < m)$ of vector $indx(0 <= indx < n)$ is addressed by $m * indx + i$; the same component in the coalesced storage pattern is addressed as $(m * bd) * ixC + bd * ixB + ixA$, where $ixC = \lfloor (m.indx + i)/(m.bd) \rfloor = bx$, $ixB = i$, $ixA = \mod (indx, bd) = tx$. The original linear indexing is mapped to coalesced indexing according to

$$m.indx + i \rightarrow m.bd.bx + i.bd + tx$$

We want to identify such patterns and use compiler pass to reorganize them to a strided pattern. Currently we only consider the case that all threads filling a consecutive region in memory so that we can neglect which thread holds which data.

II. Related Work

A. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs

GPUDrano [1] is a scalable static analysis tool designed to detect uncoalesced global memory accesses in CUDA programs. Uncoalesced memory accesses, which occur when a GPU program accesses DRAM in a non-optimal way, can significantly increase latency and energy consumption. The authors demonstrate GPUDrano's effectiveness by comparing it against dynamic analysis and showcasing its ability to identify and reduce uncoalesced accesses in the Rodinia GPU benchmark suite. Their findings indicate that fixing these detected inefficiencies can lead to performance improvements of up to 25%. This study's insights into GPU memory access patterns and the development of a practical tool for optimizing such patterns are particularly relevant to our work, as they provided a foundational understanding of memory access inefficiencies and practical solution to address them.

B. Static detection of uncoalesced accesses in GPU programs

This study [2] presents a significant advancement in the field of GPU programming optimization. The author introduced a novel static analysis method for detecting uncoalesced global memory accesses in GPU programs, which are known to hinder performance significantly. They offer a comprehensive formalization of the problem of uncoalesced memory accesses and propose an abstraction-based approach for their detection. Their methodology contrasts dynamic analysis techniques, demonstrating its efficiency and effectiveness in the evaluation using the Rodinia benchmark suite. This work is particularly relevant to our research as it provides a new perspective on improving GPU program performance through static analysis, highlighting both the challenges and potential solutions in this domain.

III. Compilation Pipeline

Our compilation pipeline integrates Clang and NVCC to efficiently compile and optimize CUDA code for GPU execution. The process is delineated into several meticulous steps, each crucial for achieving optimal performance.

In the initial compilation phase with Clang, two distinct processes occur. First, the CUDA device code is compiled. This is achieved by including the –cuda-device-only flag with Clang, ensuring that only device-specific code is targeted, resulting in a .bc (LLVM bitcode) file for the device code. Separately, the host code is also compiled using Clang, producing another .bc file. This separation of device and host code compilation is a key aspect of our pipeline.

The next step involves optimization with the Opt tool. Our custom optimization pass is applied to the device code's .bc file. Utilizing the opt tool allows us to load, modify, and enhance the initial bitcode, leading to an optimized .bc file. This step is pivotal in refining the code for better performance.

Following the optimization, the LLVM code generation phase begins. The llc tool is used to convert the optimized device bitcode into PTX (Parallel Thread Execution) assembly code. This PTX code is further compiled into a .cubin file, representing the binary device code. This conversion is critical for preparing the code for execution on the GPU.

Device linking constitutes the next critical phase. Here, the device code, contained within the .cubin file, is linked with the host code. This step integrates the separately compiled components into a single object file for the target kernel, ensuring cohesive operation of the device and host segments.

The final assembly is conducted with NVCC, the Nvidia CUDA Compiler. In this step, the device code and the main driver function (typically main) are assembled together. This assembly culminates in the creation of an executable file, such as rgb.out, ready for deployment and execution on the target GPU environment.

This process exemplifies our dedication to optimizing CUDA code for maximum efficiency and performance on GPU platforms. The pipeline is meticulously designed to leverage the strengths of both Clang and NVCC, ensuring a robust and streamlined compilation workflow.

IV. Design

It is important to note that COALDA primarily focuses on optimizing bandwidth for data transfers between the host and the GPU, which is the major field for uncoalesced memory accesses

There are overall two steps:

A. Recognize uncoalesced access regions

Since we're focusing on data copy between host memory and device memory, our approach targets store operations as starting point. We noticed that stores that relates to large data copies are frequently coupled with GEP operations both in their value and pointer operands. Moreover, since the operation is meant for data copy, the value operand for the store shall originates from a load instruction. Consider the source code in CUDA:

$$arr_{dst}[3 * tid + 1] = arr_{src}[3 * tid + 1]$$

It needs firstly load the data from arr_{src} and create a temporal pointer to manage that data. Then it creates another pointer for store's pointer operands that has the correct offset. Another important observation is that since we're consider data copy for aggregated data types like arrays, pointers should come from LLVM GetElementPtr operations.

Then, we set off to analyse the addresses accessed by GEPs operations to determine whether they're related to SIMD instructions. We achieve this by walking backwards the dependence tree of the addresses being calculated. We created our heuristic to identify all possible uncoalesced addresses accessed by SIMD operations: a finite automata that is used to traverse and parse the dependence tree. A node in our finite automata can be denoted as

$$N = (State, Children, AST).$$

$State$ is the current instruction we're looking at. $Children$ is the new states that will be generated by the current state and AST is the symbolic expression for pointer address offset. . Each node in the finite automata represents a possible symbol or transformation, which can be binary operators representing calculation, propagate operation that transfer to next node, symbols that related to SIMD operations such as thread id, block index, etc. We demonstrate the sudo code for our backward data flow analysis in Algorithm 1.

To further identify uncoalesced memory addresses, we defined the canonical expression for an uncoalesced memory address, built on upon COALDA's AST syntax:

$$stride * (globalTID/localTID) + offset$$

'stride' is usually the size of unit data structure being copied or transmitted, which must be greater than 1 (otherwise the access is coalesced). $globalTID/localTID$ is the index of thread in each thread blocks. If there are more than one thread blocks, then the thread index will be global. 'offset' is the byte index of the element in the data structure that is being transferred by the current thread. By integrating our own heuristic to backward dataflow analysis, COALDA is able to filter out all false positive uncoalesced memory accesses, which serves a great promise to correctness after optimization. Note that only addresses that match COALDA's canonical expression will be considered for address coalescing fix up. There are a lot more uncoalesced memory address expressions, for example, non-linear expression such as $arr_{dst}[tid * tid + 1] = arr_{src}[tid*tid+1]$. However, generalizing our coalescing technique to all non-linear expression is a much harder task and non-linear expression is not as common as linear ones in CUDA programming.

Based on our own AST in parsing the addresses from backward dataflow analysis into this canonical address form, COALDA can further classify store operations into groups that has the similar canonical expression discussed in next section.

Algorithm 1 BT-automata

```
Data: $Node \leftarrow (State, Children, AST)$
if $Node.Children$ not empty then
    Return
end
if $Node.State = Loadptr$ then
    Add $ptr$ to $Node.Children$
    $Node.AST = Node.Children.first.AST$
end
if $Node.State = Allocaptr$ then
    Find oldest $Store$ that writes to $ptr$ to $Node.Children$
    $Node.AST = Node.Children.first.AST$
end
if $Node.State = Storeval, ptr$ then
    Add $val$ to $Node.Children$
    $Node.AST = Node.Children.first.AST$
end
if $Node.State = Add\ x1, x2$ then
    Add $x1$, $x2$ to $Node.Children$
    $Node.AST = Node.Children.x1.AST + Node.Children.x2.AST$
end
if $Node.State = Mul\ x1, x2$ then
    Add $x1$, $x2$ to $Node.Children$
    $Node.AST = Node.Children.x1.AST \times Node.Children.x2.AST$
end
if $Node.State = SignExtend\ I$ then
    Add $I$ to $Node.Children$
    $Node.AST = Node.Children.first.AST$
end
if $Node.State = CallThreadIdx$ then
    $Node.AST = ThreadIdx$
    Return
end
if $Node.State = CallBlockDim$ then
    $Node.AST = BlockDim$
    Return
end
if $Node.State = CallBlockIdx$ then
    $Node.AST = BlockIdx$
    Return
end
for $C$ in $Node.Children$ do
    $Node \leftarrow C$ Run BT-automata on $Node$
end
```

B. Grouping uncoalesced loads/stores

We can "coalesce" a group of store operations if the addresses they operates on can be merged into a contiguous area by assigning them to a same group. First, we need to make sure the pointer source operands are the same for those loads and store, i.e, the base pointer address, such as `int* pixel_src`. Then, we will mostly focus on the canonical expression for pointer offset. In general, if the *stride* and $globalTID/localTID$ field in the canonical expression match, plus the base pointer two stores operates on being the same, then two stores can be "coalesced". That is because they're transferring different parts of the same data structure, whose size *stride* number of bytes. However, because of 'stride' is a coefficient for *tid*, memory access within one thread wrap will acting as striding access, which leads to low DRAM bursts utilization since not all bytes in that DRAM row access is utilized. The main difficultly is how to determine if two AST of canonical expressions has the same stride and TID type, as the tree structure can be much different. Instead of getting too deep into finding formal equivalence, we apply a heuristic because we are dealing with linear access pattern. We can apply the distribution rule until the expression cannot be further expand and compare the subtrees containing only multiplies. This means we're effectively pushing down multiply operators to be child node of add operator, which is shown in figure(3). After the transformation, all multiply operator will be children node of add and comparing what're being multiplied in each subtree is sufficient to determine if two canonical expressions has the same TID type and stride, shown in figure(4).

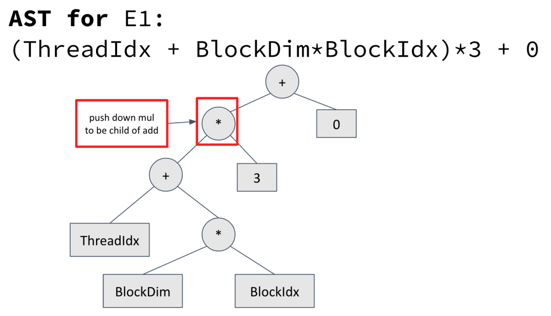
AST for E1:
(ThreadIdx + BlockDim*BlockIdx)*3 + 0
push down mul to be child of add
+
*
0
3
+
ThreadIdx
*
BlockDim
BlockIdx

Fig. 3. AST before distribution transform

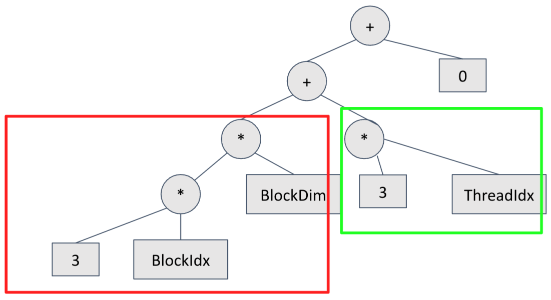

Fig. 4. AST after transform. Boxed area is for subtree compare

C. *Restructure to unstrided patterns*

COALDA is able to restructure uncoalesced memory accesses back to coalesced ones according to the group classification done in the recognization step.

The coalescing step is to reduce the *stride* factor back to one, which is done by reassigning addresses for stores in the same group. COALDA only considers data copies that will cover the whole contiguous memory region, for example, transferring the whole RGB pixel rather than R and G field excluding B. By address recalculations, threads in SIMD will transfer the whole data structure in one go rather than each different byte in the data structure, fully utilizating DRAM burst because memory being accessed is contiguous. COALDA achieves this by transforming original GEP operations in stores' operands to access memory in coalesced way. After that it performs dead code elimination to erase address calculations for original uncoalesced loads and stores. For example, `DEST = pixel_src[3 * global_tid + 0]` is clearly uncoalesced. If it can be grouped with other loads that work on the same pointer base `pixel_src`, then COALDA will apply a linear transformation on its original pointer offsets shown in listing(1). In this example, we're keeping the original pointer base, %19 unchanged and only assigned a different offset, %LoadValLoadNewOffsetZext1, to the GEP on line 14. Backtracking the calculation source for %LoadValLoadNewOffsetZext1, we found:

$$
\begin{aligned}
&\%LoadValLoadNewOffsetZext1 \\
&= 3 \times BlockDim \times BlockIdx + threadIdx + 0 \times BlockDim.
\end{aligned}
$$

which is the desired coalesced offset.

```
1 // original load
2 // DEST = pixel_src[3 * global_tid + 0];
3   %BlockIndex = call i32
      ↪ @llvm.nvvm.read.ptx.sreg.ctaid.x()
4   %BlockDim = call i32
      ↪ @llvm.nvvm.read.ptx.sreg.ntid.x()
5   %16 = mul i32 %BlockIndex, %BlockDim
6   %LocalTid = call i32
      ↪ @llvm.nvvm.read.ptx.sreg.tid.x()
7   %multDimIndex = mul i32 %BlockDim, %BlockIndex
8   %GlobalTID = add i32 %multDimIndex, %LocalTid
9   %"multDimIndexScaled#3" = mul i32 %multDimIndex, 3
10  %"GlobalTIDScaled#3" = add i32
      ↪ %"multDimIndexScaled#3", %LocalTid
11  %"offset$0TimesBlockDim" = mul nsw i32 0,
      ↪ %BlockDim
12  %valLoadNewOffsetWithOffset1 = add i32
      ↪ %"GlobalTIDScaled#3", %"offset$0TimesBlockDim"
13  %LoadValLoadNewOffsetZext1 = zext i32
      ↪ %valLoadNewOffsetWithOffset1 to i64
14  %24 = getelementptr inbounds i32, ptr %19, i64
      ↪ %LoadValLoadNewOffsetZext1
15  %25 = load i32, ptr %24, align 4
```

Listing 1: Resulted coalesced load by applying COALDA

V. Evaluation

The compiler pass focuses on and should be able to recognize and restructure any kernel code that reads/writes a consecutive region of memory to achieve memory coalescing. Especially, it is practically helpful for shared memory based gather-and-scatter pattern, where the kernel first loads a consecutive region of global memory into shared memory (gather), performs some computation inside its shared memory, and writes back the results from shared memory to global memory (scatter). The compute stage could be arbitrary. Such pattern is widely seen in GPGPU programming and can be effectively addressed by our pass.

However it is worth mentioning that fixing memory coalescing does not guarantee a better whole program performance. Restructuring memory access pattern also leads to changes in other performance metrics, notably cache hit rate, as the coalesced pattern has worse spatial locality and leads to more compulsory misses.

Another observation is that the bandwidth benefit for changing to coalesced pattern is offset by modern hardware design.

Usually, modern hardware has internal mechanism for merging and forwarding data loads and writes, which addresses the inefficiency issue at the hardware level. As of our profiling results, we found that sometimes uncoalesced kernels automatically achieve a non-worst-case bandwitdh or even an optimal one.

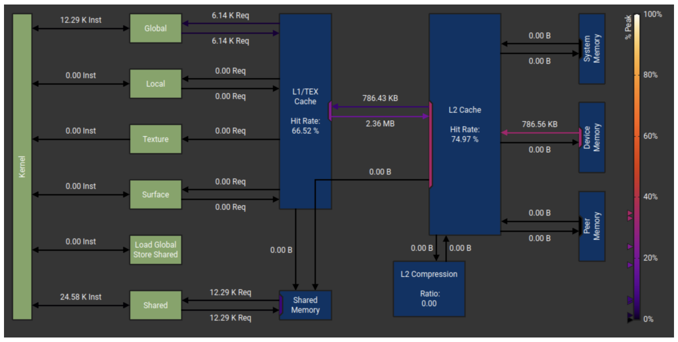

Fig. 5. Simple Kernel BEFORE Coalescing

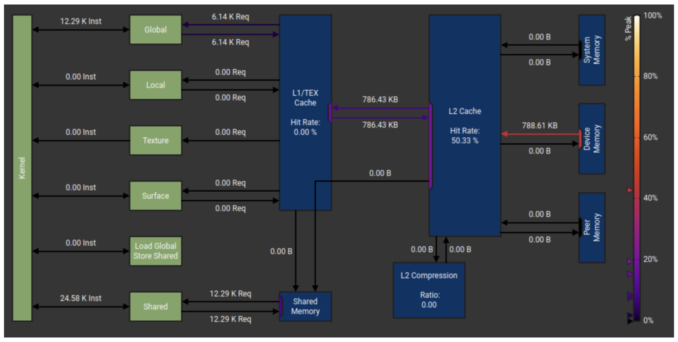

Fig. 6. Simple Kernel AFTER Coalescing

VI. Results and Analysis

We tested the compiler pass on some kernel functions with gather-and-scatter pattern, with both the global memory reads and writes being intuitive but uncoalesced. For a simple kernel, one thread intuitively corresponds to one RGB pixel, which is three integers. To further test on larger strides and data volume, we have another unrolled kernel with one thread corresponding to three RGB pixels or nine integers. On the LLVM IR level, by applying the pass the regions are successfully recognized and both memory operations are restructured to a coalesced pattern. The correctness is verified by applying the modified IR to further compilation pipeline, producing correct program behaviors.

We applied NVIDIA Nsight Compute for profiling and verification, with the MemoryWorkloadAnalysis Chart especially useful. As for the GPU we are using caches all traffic between the processing units and global memory through L2 cache, we want to focus on L2 cache traffic. For the L2 write back bandwidth, the simple kernel is reduced by three times and the unrolled kernel is reduced by nine times, matching the theoretical value precisely. For the L2 read bandwidth, the simple kernel is already the optimal and there is no improvement. The unrolled kernel does only achieves an improvement by around 6.4x, as the uncoalesced bandwidth is not the worst case one. Also, the traffic between L2 cache and device memory does not change at all. This could be explained by hardware merging and forwarding mentioned above. For cache hit rate, both kernels drop after restructuring to the coalesced pattern, as of worse spatial locality.

small

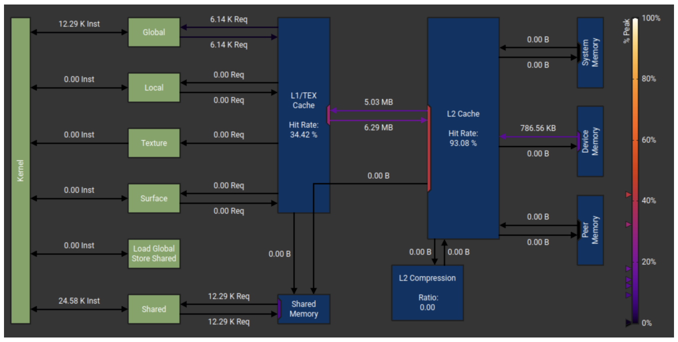

Fig. 7. Unrolled Kernel BEFORE Coalescing

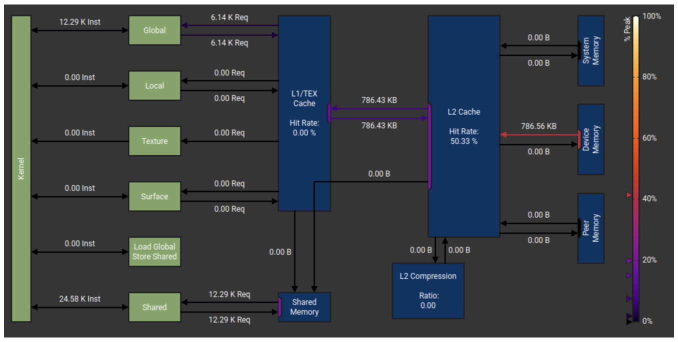

Fig. 8. Unrolled Kernel AFTER Coalescing

```
#define TILE_WIDTH 512

/*
 The intuitive kernel implementation
*/

__global__ void
rgb_increase_brightness_pass_ready(
  int *pixel_dst,
 int *pixel_src, int size,
 float factor) {
 /*
  Declaration
 */
 int global_tid = blockIdx.x *
          blockDim.x + threadIdx.x;
 int local_tid = threadIdx.x;
 __shared__ int pixel_smem_src[3 * TILE_WIDTH];
 __shared__ int pixel_smem_dst[3 * TILE_WIDTH];

 /*
  "Gather": Read input pixels to shared memory
 */
 pixel_smem_src[3 * local_tid + 0] =
  pixel_src[3 * global_tid + 0];  // r
 __syncthreads();
 pixel_smem_src[3 * local_tid + 1] =
  pixel_src[3 * global_tid + 1];  // g
 __syncthreads();
 pixel_smem_src[3 * local_tid + 2] =
  pixel_src[3 * global_tid + 2];  // b
 __syncthreads();

 /*
  Computation, very simple here
  but can be more complicated
 */
 pixel_smem_dst[3 * local_tid + 0] =
  min(255, (int)(factor *
    (pixel_smem_src[3 * local_tid + 0])));
 pixel_smem_dst[3 * local_tid + 1] =
  min(255, (int)(factor *
    (pixel_smem_src[3 * local_tid + 1])));
 pixel_smem_dst[3 * local_tid + 2] =
  min(255, (int)(factor *
    (pixel_smem_src[3 * local_tid + 2])));
 __syncthreads();

 /*
  "Scatter": Write result to destination
 */
 pixel_dst[3 * global_tid + 0] =
  pixel_smem_dst[3 * local_tid + 0];
 __syncthreads();
  pixel_dst[3 * global_tid + 1] =
 pixel_smem_dst[3 * local_tid + 1];
 __syncthreads();
  pixel_dst[3 * global_tid + 2] =
 pixel_smem_dst[3 * local_tid + 2];
};
```

VII. Conclusion and Future Work

Currently, COALDA only addresses a subset of uncoalesced memory access patterns, but we see potential for the framework to address more. For example, when operating on tensors it is common to see column major accesses on row-major data, which can be recognized similarly by altering the pass and solved by inserting code to transpose the tensor in advance.

Also, in our brief implementation the pass is applied at an very early stage in the compilation pipeline. We want the access pattern to be preserved in the end. Considering that in real-world usage, the pipeline is rather deep and involves other code restructuring techniques, it is worth doing to move the pass to a rather late stage in the whole pipeline, such as right before PTX generation or even one that directly operates on PTX.

The complete source code and additional resources for this project are available at our GitHub repository.

References

[1] A. F. N. Alur and O. Authors], "Gpudrano: Detecting uncoalesced accesses in gpu programs," *Journal Name*, vol. Volume Number, no. Issue Number, p. Page Range, 2017.

[2] R. Alur and other authors, "Static detection of uncoalesced accesses in gpu programs," *Formal Methods in System Design*, vol. 60, no. 2, pp. 146–173, 2022.