Retargeting and Respecializing GPU Workloads for Performance Portability

Ivan R. Ivanov
Tokyo Institute of Technology
RIKEN R-CCS
Kobe, Japan
ivanov.i.aa@m.titech.ac.jp

Oleksandr Zinenko
Google DeepMind
Paris, France
zinenko@google.com

Jens Domke
RIKEN R-CCS
Kobe, Japan
jens.domke@riken.jp

Toshio Endo
Tokyo Institute of Technology
Tokyo, Japan
endo@is.titech.ac.jp

William S. Moses
University of Illinois Urbana-Champaign
Google DeepMind
Illinois, United States
wsmoses@illinois.edu

***Abstract*—In order to come close to peak performance, accelerators like GPUs require significant architecture-specific tuning that understand the availability of shared memory, parallelism, tensor cores, etc. Unfortunately, the pursuit of higher performance and lower costs have led to a significant diversification of architecture designs, even from the same vendor. This creates the need for performance portability across different GPUs, especially important for programs in a particular programming model with a certain architecture in mind. Even when the program can be seamlessly executed on a different architecture, it may suffer a performance penalty due to it not being sized appropriately to the available hardware resources such as fast memory and registers, let alone not using newer advanced features of the architecture.**

We propose a new approach to improving performance of (legacy) CUDA programs for modern machines by automatically adjusting the amount of work each parallel thread does, and the amount of memory and register resources it requires. By operating within the MLIR compiler infrastructure, we are able to also target AMD GPUs by performing automatic translation from CUDA and simultaneously adjust the program granularity to fit the size of target GPUs.

Combined with autotuning assisted by the platform-specific compiler, our approach demonstrates 27% geomean speedup on the Rodinia benchmark suite over baseline CUDA implementation as well as performance parity between similar NVIDIA and AMD GPUs executing the same CUDA program.

I. Introduction

Accelerators like GPUs remain the hardware target of choice for performance-critical software. Achieving high performance on these accelerators requires programmers to effectively leverage a peculiar programming model, most often exposed as C++ language extensions such as CUDA for NVIDIA GPUs and ROCm for AMD. While the community has developed alternative methods to portably program GPUs, including: a high-level block programming model in Triton [1], automatic mapping of C++ code onto GPUs [2], NumPy-style abstractions with varying degree of automated scheduling in JAX [3], TC [4], and TVM [5]; many of the performance-critical scientific programs, including these very portability frameworks, remain written in CUDA.[1]

While the CUDA programming model and syntax have remained relatively stable over time, the underlying GPU hardware has evolved significantly, adding many new features and instructions. For example, earlier versions of programmable NVIDIA GPUs used "half warps" of 16 threads for scheduling and had a limitation of 1024 threads running concurrently on a hardware unit while modern GPUs use "full warps" of 32 and allow up to 2048 threads per hardware unit. Similar changes can be observed in the amount of available low-latency memory and registers. This trend is even more important when considering GPUs of a different vendor, like AMD, which operate in "wavefronts" of 64 threads and allow up to 4096 threads per hardware unit.

Even when GPU kernels written in CUDA appear to run on newer NVIDIA GPUs, they may often fail to reach similar utilization as the kernels are incorrectly sized for the target architecture. However, this may be avoided through skillful use of the programming model by writing CUDA programs that adapt to different numbers of concurrent threads. But even programs with this flexibility do not permit control of the amount of allocated "shared" memory between several threads in a group or the amount of registers used (which is proportional to the number of threads). Both of these characteristics have a dramatic impact on the overall performance. These sizing problems are often amplified when porting a program to a GPU of a different vendor, let alone the often non-trivial engineering effort of porting itself.

In this paper, we propose a compiler-based mechanism to "resize" GPU programs to a particular architecture. Taking *existing CUDA code*, our compiler can control the *granularity* of the program including the amount of work performed by

[1]In spite of various alternatives, like ROCm and SYCL [6], the CUDA framework, a pioneer of the GPU programming model, is used in significantly more applications due to legacy, maintenance, and network effects.

979-8-3503-9509-9/24 © 2024 IEEE

Accepted for publication by IEEE. © 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

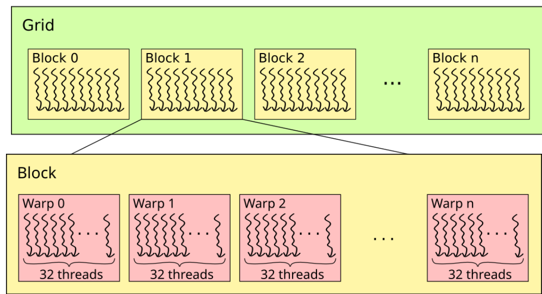

Fig. 1. Execution structure of a GPU kernel (for a modern CUDA GPU with 32 threads per warp).

each GPU thread as well as the amount of memory and registers, without changing the programming model.

Our work builds on and extends the Polygeist compiler [7, 8] to translate CUDA code into a target-agnostic representation based on MLIR [9]. This representation makes multi-level parallelism of the GPU programming model as well as "shared" memory explicit, allowing us to change the amount of work performed per thread and the amount of memory resources used through the generalized "coarsening" transformation. It also allows us to retarget the program originally written in CUDA to work on AMD GPUs automatically.

Our approach is further connected to the backend GPU compiler to extract information from lower levels of the compilation stack, in particular the number of registers used and the amount of spilling. This enables a sort of compile-time autotuning to filter out programs that would have catastrophic performance penalty due to bad resource utilization. Conventional autotuning based on actual runtimes is also available to select the best granularity on the given hardware.

II. Background

A. GPU Programming Model and Architecture

In this work, our inputs are programs written the CUDA programming model and outputs are CUDA and ROCm executables. While not the subject of this study, our system may be adapted for other input programming models like SYCL [6] or OpenCL [10].

1) GPU Execution Structure: Although there has been work on different paradigms like persistent kernels [11, 12], the traditional CUDA programming model forces the programmer to represent computations in a hierarchical structure. Each kernel executes a fixed number of limited threads, which are structured in blocks in a grid, the number of which can be variable (Fig. 1). Both the number of threads per block and blocks per grid must be specified at kernel launch. Threads execute in parallel in groups called warps and execution of blocks themselves may overlap. As the size of the block (i.e. the number of threads) is limited and must be fixed, the most common way to scale the problem size when programming for GPUs is by varying the number of blocks.

The GPU contains multiple *Streaming Multiprocessors* (SMs) which are assigned blocks to execute. Each SM may be assigned multiple blocks, with new blocks assigned and as blocks terminate. SMs execute all threads in warps in lock-step, with each thread executing the same instruction. When a warp executes a high-latency instruction such as a memory load, the SM can switch to executing another warp. This allows CUDA to hide instruction level latencies.

2) Coalesced Memory Access: When threads in a warp execute a load, the load can be *coalesced*, and executed as fewer transactions to memory if it satisfies certain requirements (load size, stride, global offset), reducing minimized bandwidth [13]. Although newer architectures are more lenient on the requirements for good performance, memory locality remains critical to best utilise memory bandwidth [14].

3) Occupancy: When blocks are assigned to SMs they must abide by certain constraints to avoid overwhelming the SM's resources. These constraints include: registers per SM, max resident threads per SM, max threads per block, max shared memory per SM, and max shared memory per block. On the other hand, each block requires a certain amount of shared memory, registers, and threads to execute. Therefore, a given kernel has a maximum number of blocks that can reside on a single SM. The threads of blocks that occupy an SM are called the *active_threads* and the ratio *active_threads* / *max_threads_per_SM* is known as a kernel's occupancy.

While the SM has resources that limit the occupancy, it also has a separate set of resources (known as execution resources) that do not limit the occupancy but instead are shared by the blocks on the SM, including: memory bandwidth, arithmetic logic units (ALUs), and floating point units (FPUs). To achieve the most performance out of a GPU, the active threads must generate enough work so that they can saturate the execution resources of the SMs and hide latency. This is a matter of balancing per-thread workload and occupancy.

4) Performance Portability: The performance of kernels on a different target hardware than the one the programmer intended is hampered by the fact that the capacity of the different occupancy-limiting and execution resources vary between different GPU vendors and even within the same vendor (e.g. pre-Fermi CUDA hardware had 16 threads per warp, current ones have 32, and AMD GPUs have 64). This variability makes it difficult to write kernels that perform well on a variety of target hardware. The fact that dynamic shared memory is more difficult to implement than static sized shared memory in C/C++ CUDA programming exacerbates the problem and kernels tend to be of a fixed granularity.

Our proposed transformations automatically vary the granularity of computation on both the block and thread level to make best use of the available resources for a given target.

B. GPU Compilation in Polygeist/MLIR

1) MLIR: MLIR is a framework for defining and mixing abstractions used internally by a compiler [9]. Unlike other IR's, it has an open set of computational primitives (called *operations*) and types. MLIR provides a number of reusable abstractions organized in *dialects*, which are expected to co-

```
1  scf.parallel (%bx, %by, %bz) = (0, 0, 0)
2                to (grid.x, grid.y, grid.z) {
3    %sh_mem = memref.alloca : memref<32x32xf32>
4    scf.parallel (%tx, %ty, %tz) = (0, 0, 0)
5                 to (block.x, block.y, block.z) {
6      // computation using %b{x,y,z} and %t{x,y,z} ...
7      polygeist.barrier(%tx, %ty, %tz)
8      // ...
9    }
10 }
```

Fig. 2. Representation of a GPU kernel in Polygeist. Lines 1 and 4 represent GPU blocks and threads. Line 3 allocates per-block shared memory. Line 7 features a thread barrier.

exist in a program and model different aspects of it. For example, a simple GPU kernel can be expressed using:

- `arith` for integer and floating point arithmetic;
- `memref` for operations and types for memory access;
- `scf` for structured control flow;
- `gpu` for the common GPU programming model (SIMT).

Compilers built with MLIR often define additional dialects.

This mix-of-abstractions model is particularly useful for targeting accelerators, such as GPUs, in a compiler as it allows the compiler to reason simultaneously about host and device code that are expressible in the same translation unit [15]. It enables a whole range of classical optimizations based on the static single assignment (SSA) form such as common subexpression elimination and loop-invariant code motion to apply across host/device boundary, and supports new accelerator-aware transformations [8, 16].

2) Polygeist: Polygeist is a compiler for C++ and extensions built with MLIR [7]. It introduced an MLIR-based compiler abstraction for GPU barrier synchronization primitives that enables GPU-to-CPU transpilation and barrier optimization [8]. Specifically, it uses the `scf` parallel loop operation to represent GPU blocks and threads, and its custom `polygeist.barrier`[2] operation for thread synchronisation (equivalent to CUDA `__syncthreads`). One iteration of the parallel loop corresponds to one GPU block or thread in the kernel configuration. This does not imply that all iterations are executed concurrently, but that they are subject to block and warp scheduling across SMs. The *parallel* operation itself does not prescribe that and merely indicates independence of individual operations from each other. Therefore, we will differentiate between iterations of the parallel loop and threads.

In accordance with the GPU programming model, the `barrier` operation must be executed by all threads in the synchronization scope, which is typically a block or a warp. This is often referred to as control flow *convergence* since it effectively precludes control flow to vary between threads around `barrier`s. In other words, control flow affecting the `barrier` should not depend on thread IDs. Since Polygeist models multiple levels of parallelism, its `barrier` additionally refers to the parallel loop iterators (interpreted as thread identifiers) for which synchronisation is required, see Fig. 2.

Despite being able to represent the GPU programming model, Polygeist has been unable to generate efficient GPU code

[2]MLIR operation are prefixed with the name of the dialect, e.g. `scf.` or `polygeist.` We omit these prefixes when unambiguous for brevity.

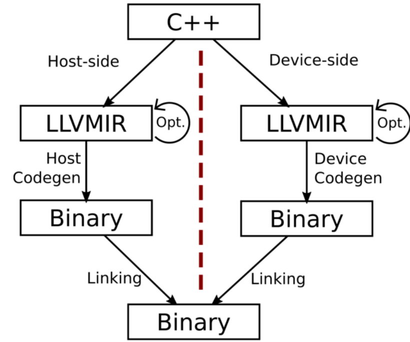

Fig. 3. Conventional GPU compiler treats host (CPU) and device (GPU) code as separate translation units.

from CUDA inputs due to missing optimisations and target information, and only focused on translation to CPUs.

C. Thread Coarsening

A GPU program can be seen as performing similar computations with a large number of threads on an even larger number of work units so that each thread processes multiple units. *Thread coarsening*, i.e. increasing the amount of units items processed by a thread, has been used to improve performance of GPU kernels by hiding the latency of expensive memory access operations, first manually [17], and later automatically in a compiler [18, 19, 20].

However, thread coarsening may have adverse effect on performance, for example, by introducing strided memory loads that hamper coalescing or by increasing the register pressure and thus decreasing GPU occupancy.

III. Polygeist-GPU Pipeline Overview

We extend Polygeist to provide the end-to-end GPU compiler accepting CUDA code, performing optimization and kernel granularity selection, and targeting *both* NVIDIA and AMD GPUs. Contrary to conventional compilers that treat host (CPU) and device (GPU) code as separate translation units (Fig. 3), our approach keeps both parts together thus allowing for simultaneous updates of the kernel configuration and launch and the kernel code itself (Fig. 4).

Specifically, we inline the MLIR representation of the GPU code into that of the CPU code while keeping it wrapped into a region-carrying operation, as shown in Fig. 5. This operation permits code motion across region boundary, except for `parallel`- and `barrier`-related constructs. Contrary to the previous inline representation of GPU kernels in MLIR via `gpu.launch` [15], ours uses explicit parallel loops that are directly amenable to loop analyses and optimizations.

After optimization on this representation, the kernel is outlined and processed by a target-specific pass pipeline available in MLIR to produce a target-specific binary embedded as global data in the IR. Remaining host code is then processed by a target-specific pipeline to replace outlined `gpu_wrappers`

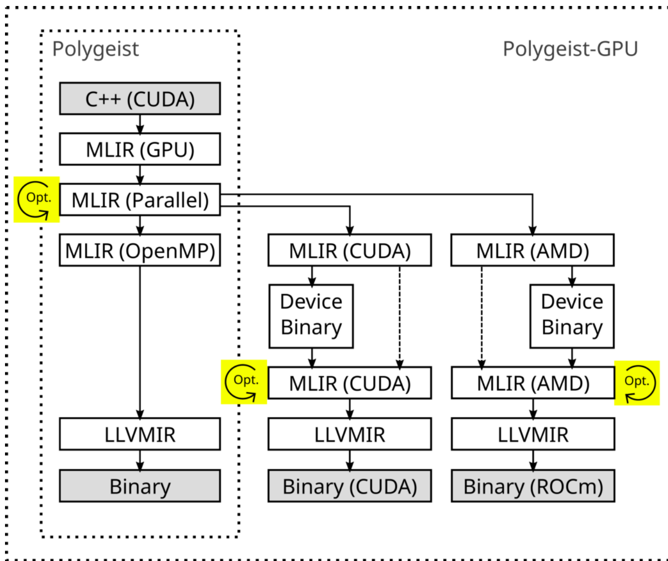

Fig. 4. Polygeist keeps host (CPU) and device (GPU) code in the same translation unit enabling host/device optimizations. Polygeist-GPU (this work) leverages the "parallel" MLIR abstraction to re-optimize device code and potentially re-target it to a different platform without compromising performance.

```
// ... Host-side code ...
%device_mem = gpu.alloc()
gpu.memcpy %device_mem %host_mem
// ... Device-side code ...
polygeist.gpu_wrapper {
  parallel %bi = 0 to grid_size {
    parallel %ti = 0 to block_size {
      // ... use %device_mem ...
    }
  }
}
```

Fig. 5. Kernel code is inlined into host code using a region-carrying MLIR operation (highlighted), allowing for host/device code motion.

with respective GPU runtime calls, and further lowered to a self-contained optimized binary using LLVM.

Other than the optimizations contributed in this work, the parallel GPU representation enables a number of other preexisting optimizations in Polygeist and MLIR, for example elimination of barriers, code motion around barriers, memory-to-register promotion across barriers, and parallel loop-invariant code motion. [8]

IV. Nested Parallel Loop Unroll-and-Interleave

We specialize the classical loop unrolling transformation for nested parallel loops while taking care of GPU-style barrier synchronization. Consider first the simple sequential loop unrolled with a factor of 2 in Fig. 6. All the statements

Original

```
for %i = 0 to 16 {
  A(%i)
  B(%i)
}
```

Unrolled

```
for %i = 0 to 8 {
  A(%i * 2)          //
  B(%i * 2)          //
  A(%i * 2 + 1)
  B(%i * 2 + 1)
}
```

Fig. 6. Unrolling a serial for loop with a factor of 2. Operations from different original iterations are shown in different colors.

Naive unrolling

```
parallel %i = 0 to 8 {
  A(%i * 2)          //
  B(%i * 2)          //
  A(%i * 2 + 1)
  B(%i * 2 + 1)
}
```

Unroll-and-interleave

```
parallel %i = 0 to 8 {
  A(%i * 2)          //
  A(%i * 2 + 1)
  B(%i * 2)          //
  B(%i * 2 + 1)
}
```

Fig. 7. Unrolling a parallel loop with a factor of 2. Operations from different original iterations are shown in different colors.

Original

```
parallel %i = 0 to 16 {
  ...
  for %j = 0 to 32 {
    A(%i, %j)
    B(%i, %j)
  }
  ...

}
```

Unroll-and-interleave

```
parallel %i = 0 to 8 {
  ...
  for %j = 0 to 32 {
    A(%i_0, %j)          //
    A(%i_1, %j)
    B(%i_0, %j)          //
    B(%i_1, %j)
  }
}
```

Fig. 8. Unroll-and-interleave of a parallel loop with nested constant trip-count control flow. Note that we can reuse the `for` statement.

Original

```
parallel %i = 0 to 16 {
  ...
  %n = call @foo(%i)
  ...
  for %j = 0 to %n {
    A(%i, %j)
  }
  ...
}
```

Unroll-and-interleave

```
parallel %i = 0 to 8 {
  ...
  %n_0 = call @foo(%i_0)//
  %n_1 = call @foo(%i_1)
  ...
  for %j = 0 to %n_0 {   //
   A(%i_0, %j)           //
  }                      //
  for %j = 0 to %n_1 {
    A(%i_1, %j)
  }
  ...
}
```

Fig. 9. Unroll-and-interleave of a parallel loop with nested variable trip-count control flow. Note that we cannot reuse the for statement as the trip count differs for different `%i`s.

with side effects from the first unrolled iteration precede their counterparts from the second unrolled iteration to preserve the order of side effect and thus guarantee the validity of the transformation. A parallel loop does not imply any order of side effects between parallel iterations, only within one iteration, which allows us to interleave the statements from different operations arbitrarily as long as their relative order is preserved, as in Fig. 7. In particular, we can group them to produce an effect of unrolling each statement independently, which is conceptually similar to loop vectorization.

A. *Nested Control Flow: Unroll and Jam and Interleave*

Consider now a nested `for` loop with a constant trip count (Fig. 8). Straightforward unrolling of the outer loop (Fig. 9) would replicate the nested loop, which is not always desirable due to code size increase and additional control flow. Another classical loop transformation, unroll-and-jam, fuses the nested loops back together. We can further combine loop unroll-and-jam with statement interleaving to group statements together as before. This process can apply recursively to nested `for` loops, `parallel` loops that have even less side effect ordering constraints and `if/else` conditionals that can be treated as loops with zero or one iterations depending on the condition. When the trip count is not known statically, outside of the `if/else` special case, we consider nested loops as single statement and duplicate it. Note that unroll-jam-interleave can also handle loops with varying trip count by peeling prologue and epilogue, but we have not observed the need for it in GPU-oriented benchmarks.

B. Synchronization

Until now, we assumed *no* synchronization primitives were present in the parallel loops. GPU programming models allow for barrier synchronization across threads (but not blocks). However, the programming model requires barriers to be executed by all threads. That is, barriers *cannot be nested in control flow dependent on value varying across threads in the same block* such as the thread index or a value loaded from memory at a thread index-dependent address. Therefore, barriers can only be nested in control flow with conditions identical across threads, even if the condition cannot be determined at compile time. We leverage this information to perform unroll-jam-interleave for loops containing barriers nested in parallel loops that correspond to GPU threads.

As `polygeist.barrier` explicitly specifies the loop induction variables which scope the synchronization, we can generalize this to handling control flow around barriers when unrolling any outer parallel loop that is synchronized by these barriers. This allows the interleaving to work seamlessly when different thread dimensions (x,y,z) are mapped to nested `parallel for` loops instead of one `multi-parallel`[3].

Consider now the body of the nested loop in the case of interleaving, Fig. 10. The presence of the barriers requires the transformation to preserve the relative order of statements guarded by the barrier. This condition can be satisfied by grouping copies of the statement coming from different iterations together, as in Fig. 10, left. Additionally, several barriers that end up becoming consecutive can be trivially replaced with a single one. Conversely, if a `barrier` is duplicated, e.g. when different blocks run different number of inner loop iterations, the unroll-and-interleave for the outer loop may become illegal, see Fig. 10, right. Therefore we do not apply the transformation to parallel loops that correspond to GPU blocks if they have nested control flow with conditions not known to be constant at compile time.

C. Multi Dimensional loops

The CUDA programming model enables the programmer to specify three dimensions in which the blocks and grid can be partitioned in: *x*, *y*, and *z*. Thus, the parallel loops that we encounter in our parallel kernel representation are `multi-parallels` with 3 (or fewer) dimensions.

This raises the question of *which* dimensions we should perform thread and block coarsening. We provide two ways to specify this in our implementation. The *total* coarsening factor for a `multi-parallel` may be specified, and Polygeist-GPU will attempt to balance the factors across dimensions that are not of a constant size 1.[4] Another option is to explicitly specify the *x*, *y*, and *z* factors.

[3]We refer to a `parallel` loop which has multiple independent iteration values, such as the ones in Fig. 2 as a `multi-parallels`.

[4]For example, for a total factor of 16, we will coarsen the 3 dimensions with 4, 2, and 2 respectively, whereas for 6 we will coarsen with 3, 2, and 1.

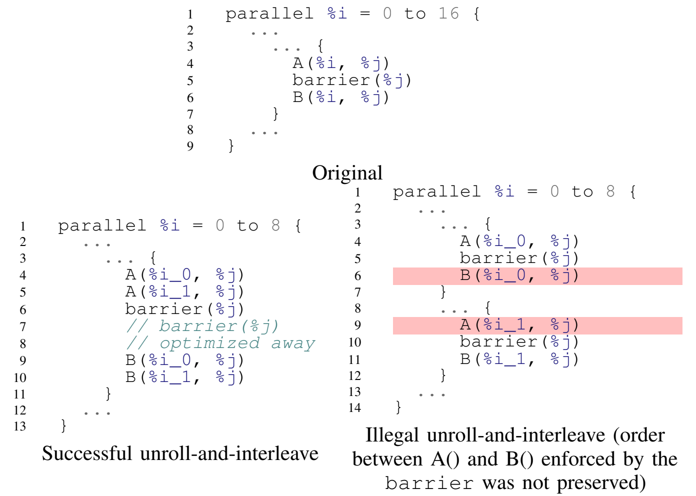

Fig. 10. Legality of unroll-and-interleave of a `parallel` loop with a nested barrier. We unroll w.r.t. `%i`, with a `barrier` with a synchronisation scope `%j` which may be different from `%i`. We preserve the `barrier` semantics (and thus the transformation is legal) only if we interleave all `barriers`.

V. Coarsening as Granularity Variation

The nested parallel unroll-and-interleave transformation lets us implement two specialized kinds of *coarsening* transformations, which increase the amount of work performed by each GPU thread or block. That is, these transformations affect the granularity of the kernel with respect to work units.

A. Thread Coarsening

Performing unroll-and-interleave on the parallel loop representing GPU threads (lines 4-5 in Fig. 2) achieves thread coarsening similar to prior work (subsection II-C). Since the transformation unrolls the loop associated with barriers and given the control flow convergence criteria on GPUs, this transformation is always legal.

Thread coarsening increases the number of work units a thread processes, decreases the size of the block and preserves the number of blocks. In other words, one thread handles the workload of several threads of the same block.

B. Block Coarsening

Performing unroll-and-interleave on the parallel loop representing GPU blocks (lines 1-2 in Fig. 2) produces a novel transformation, *block coarsening*. This transformation may be illegal when it cannot interleave the `barrier` associated with threads and would need to duplicate it instead. That is, if the (thread) barriers are surrounded by control flow that (transitively) depends on the block identifier. In the general case of multiple nested parallel loops, the transformation is illegal if it would duplicate a barrier that synchronizes *another* loop than the one being unrolled.

Block coarsening increases the number of work units processed by a block, preserves the number of threads per block and decreases the number of blocks. Considered differently, each thread now handles the workload of several threads from *different* blocks, hence the legality requirement of non-divergent control flow between blocks.

```
1  parallel %b = 0 to %n {
2    %shared_mem = alloca(32 x f32)
3    parallel %t = 0 to 32 {
4      %v = load %global_mem[%b * 32 + %t]
5      store %v, %shared_mem[%t]
```
Original kernel

```
1  parallel %b = 0 to %n {
2    %shared_mem = alloca(32 x f32)
3    parallel %t = 0 to 16 {
4      %v_0 = load %global_mem[%b * 32 + %t * 2]
5      %v_1 = load %global_mem[%b * 32 + %t * 2 + 1]
6      store %v_0, %shared_mem[%t * 2]
7      store %v_1, %shared_mem[%t * 2 + 1]
```
Thread Coarsening with naive indexing destroys the coalesced access

```
1  parallel %b = 0 to %n {
2    %shared_mem = alloca(32 x f32)
3    parallel %t = 0 to 16 {
4      %v_0 = load %global_mem[%b * 32 + %t]
5      %v_1 = load %global_mem[%b * 32 + %t + 16]
6      store %v_0, %shared_mem[%t]
7      store %v_1, %shared_mem[%t + 16]
```
Thread coarsening with coalescing-friendly indexing [18] preserves coalesced access but executes more load instructions.

```
1  parallel %b = 0 to (%n / 2) {
2    %b_0 = %b * 2
3    %b_1 = %b * 2 + 1
4    %shared_mem_0 = alloca(32 x f32)
5    %shared_mem_1 = alloca(32 x f32)
6    parallel %t = 0 to 32 {
7      %v_0 = load %global_mem[%b_0 * 32 + %t]
8      %v_1 = load %global_mem[%b_1 * 32 + %t]
9      store %v_0, %shared_mem_0[%t]
10     store %v_1, %shared_mem_1[%t]
```
Block Coarsening preserves coalesced access *and* executes the same number of loads

Fig. 11. How different coarsening transformations interact with coalesced memory access.

C. Tradeoffs between Block and Thread Coarsening

Unlike thread coarsening, block coarsening combines shared memory allocations from different blocks, increasing shared memory usage.[5] This may improve performance in kernels that underutilized shared memory capacity or bandwidth, but may degrade it in kernels that used significant portions of shared memory as it becomes the main occupancy limiter. Note that both kinds of coarsening increase register usage per thread, which is another occupancy limiter, but cannot be directly controlled outside of the platform-specific compiler.

Depending on the implementation, thread coarsening may interfere with the coalescing-friendly access pattern originally present in the kernel by introducing strided access patterns.[6] Block coarsening preserves the memory access patterns that existed in independent blocks.

When the coarsening factor is not a divisor of the upper bound of the parallel loop, an additional problem arises with how to execute the remaining iterations. When performing thread coarsening, the remaining iterations must be executed within the same block in order to preserve in-block synchronisation. However, having additional threads that execute the left over work interferes with having workloads balanced across the threads, ensuring full warps, and introduces complexities from convergent branch execution. For these reasons, we limit thread coarsening factors to only divisors of the upper bounds.

On the contrary, when doing block coarsening, we generate an *epilogue* kernel which finishes the work of the remainder of the blocks. Therefore, we extend block coarsening to any coarsening factor, and not only divisors of the upper bound. In section VII we will see how this flexibility in choosing factors allows us to improve performance even further.

[5]This is automatic in our flow as we duplicate the shared memory allocation.

[6]Coalescing still happens for strided accesses, but less efficiently as several instructions may be issued instead of one.

Since thread coarsening reduces the number of threads per block, large coarsening factors or blocks with originally few threads may end up running less than a warp worth of threads. This will result in parallel thread underutilization and degraded performance. Similarly, block coarsening reduces the number of blocks in the grid, which may result in the kernel having less blocks than SMs and a corresponding performance degradation. Our preliminary observations suggests that kernels are usually designed to leverage parallelism at block level when possible, so more parallelism and thus coarsening opportunities can be found at that level.[7]

Finally, block and thread coarsening can be combined to combine the benefits or mitigate the drawbacks.

VI. Alternative Code Paths

As the unroll-and-interleave transformations are performed on a relatively high level of abstraction, we introduce the concept of *alternative regions* in our intermediate representation to provide support for compile-time multi-versioning. We can then apply block and thread coarsening with different factors to different regions so that each region captures the same computation with different granularity. This allows us to postpone selecting the best alternative until the point in the compilation pipeline where a lower-level abstraction provides sufficiently detailed information, e.g., on register usage. Otherwise, we would have had to develop a performance model at the high level of abstraction and commit to a largely imprecise heuristic to chose the best coarsening factors.

Practically, alternatives are represented in a new multi-region MLIR operation (Fig. 12). They are produced in our flow by simply replicating the kernel body region multiple times and applying the coarsening with different factors to different regions. The compilation pipeline can then proceed as normal, with each region optimized and lowered separately from the others, until one of the following decision points is reached.

Early Pruning For Shared Memory Usage: Given that static shared memory must be allocated upfront with sizes known at compile time, we are able to analyze the alternatives immediately after producing them to compute that total amount of shared memory used. The alternatives that require shared memory in excess of what is available on the target hardware can be discarded at this point.

Kernel Statistics: In the parallel loop representation, we can collect information about the number of arithmetic and memory operations using closed-form expressions for loops, symbolic if loop bounds are not known at compile time. In the LLVM representation, we can additionally collect information about the number of branch operations in the GPU control

[7]This is due to the limited parallelism available at the thread level. CUDA programs have up to 1024 threads per block, whereas the number of blocks can go up to $2^{31} - 1$. When one wants to write a non-trivial GPU program that scales, they must build it to scale with the number of blocks, lest have a fixed maximum problem size. As a result, kernels usually scale the problem size using the number of blocks, which is what is recommended by the official CUDA C++ Programming Guide.

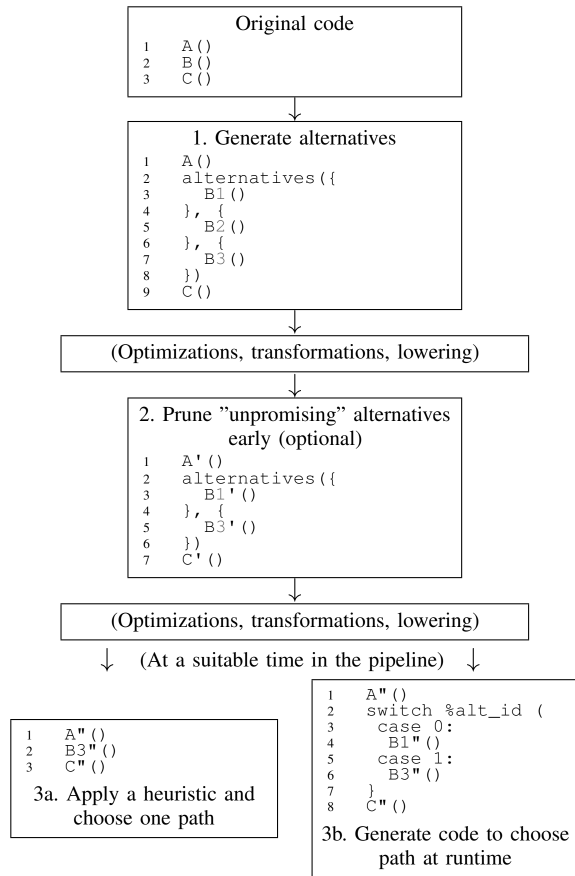

Fig. 12. Generic flow of *alternative code paths* in our compiler's pipeline. We introduce the `alternatives` MLIR operation to specify multiple code paths that have the same effect. The programmer can apply certain transformations or optimizations (1.) and defer choosing the best one until later in the pipeline when a better decision can be made (2. and 3.).

flow, which are known to negatively affect performance as control flow divergence may be realized as executing branches one after the other with irrelevant threads masked out.

Finally, compiling the representation to binary with the platform-specific backend, such as `ptxas`, provides us with information about register usage and spilling, estimated occupancy, etc. We discard alternatives that incur new spilling at this stage since spilling on GPU puts data into local memory that is several orders of magnitude slower than registers.

Timing-Driven Optimization (TDO) or Auto-Tuning: Finally, alternatives that survive preliminary filtering passes are included as independent kernels in the final binary. Our compiler offers a "profiling" mode in which it generates additional logic allowing one of the alternative implementations. Each alternative can be executed one or more times on different data to measure the average execution time, and then select the one that performs best. The compiler can be then called again to remove all the other alternatives and provide a single version without additional dispatch.

TABLE I
GPUS USED FOR EVALUATION AND THEIR SPECIFICATIONS.

	Consumer-grade		HPC	
GPU	NVIDIA A4000	AMD RX6800	NVIDIA A100	AMD MI210
Compute Capability	8.6	gfx1030	8.0	gfx90a
SMs	48	60	108	104
FLOPs (f64)	0.60T	1.01T	9.75T	22.60T
FLOPs (f32)	19.17T	16.17T	19.49	22.60T
Memory Bandwidth	445 GB/s	512 GB/s	1555 GB/s	1638 GB/s
Global Memory	16 GB	16 GB	40 GB	64 GB
L2 Cache	4 MB	4MB	40 MB	16 MB
L1 Cache (Per SM)	128 KB	16 KB	192 KB	16 KB

VII. EVALUATION AND DISCUSSION

We perform three experiments to evaluate our work. First, we demonstrate the performance improvements by *combining* block and thread coarsening compared to either of these transformation separately (Section VII-B). Second, we compare performance of the generated code against the baseline GPU compiler (Section VII-C). Finally, we showcase automatic translation of CUDA code to run on AMD GPUs using compiler representation and compare it with the baseline source-to-source method (Section VII-D).

A. Benchmarking Setup

We use four different GPUs from two vendors in our evaluation as outlined in Table I. Polygeist-GPU[8] was compiled against LLVM version 15 (git commit `00a1258`).

We collect two kinds of time measurements: *kernel* measurements correspond to the time of individual kernel runs, and *composite* measurements correspond to the entire computational part of an application including potentially multiple kernel launches plus the logic between them and host-device communication.

The evaluation is performed on two benchmark suites: Rodinia v3 [21] and HeCBench [22]. Rodinia contains 24 different CUDA benchmarks, of which 9 were excluded.[9] Rodinia's CUDA benchmarks are optimized for an old CUDA architecture and we aim to tune them to run efficiently on modern GPUs. HeCBench contains 400 benchmarks. We were only able to compile 303 of them with clang. Of these, we were able to compile and run 112 with Polygeist-GPU given its incomplete C++/CUDA support. Initial experiments on Rodinia use *composite* measurements to evaluate the overall performance of scientific applications. In-depth analysis of the thread and block coarsening impact relies on *kernel* measurements. We verify correctness of the transformation by comparing the outputs of all Rodinia benchmarks after compiling with Polygeist-GPU in different configurations and with clang.

[8]Polygeist-GPU has been merged into Polygeist and is available at https://github.com/llvm/Polygeist

[9]`hybridsort`, `kmeans`, `leukocyte`, and `mummergpu` use CUDA textures, which have been deprecated and were removed from CUDA v12, and are not supported by Polygeist. `huffman`, and `heartwall` use unsupported features within Polygeist (virtual functions), while `dwt2d`, `b+tree`, and `srad_v2` produce non-deterministic results in baseline and likely are buggy benchmarks.

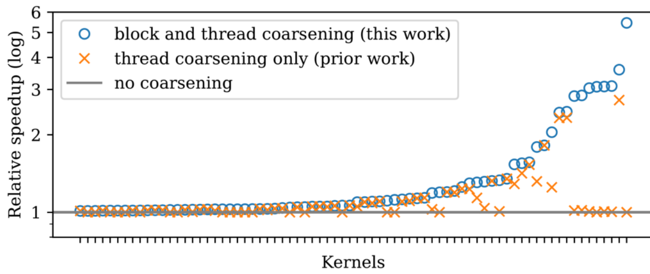

Fig. 13. A combination of block and thread coarsening (this work, blue circles) systematically outperforms thread coarsening alone (orange crosses, reimplementation of the prior work) on the A100 GPU. All measurements are for Polygeist-GPU, with the coarsening strategy changing.

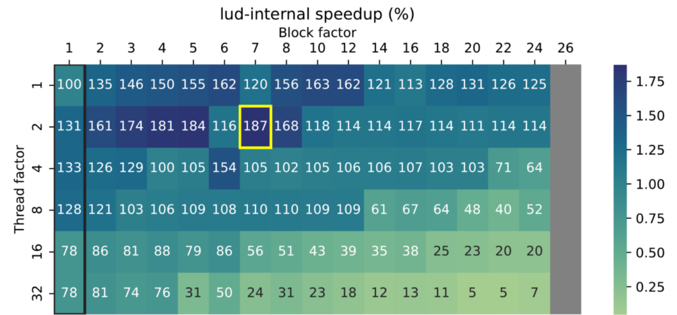

Fig. 14. Performance of different coarsening factor configurations for the main kernel in `Rodinia/lud` relative to the non-coarsened code (higher is better). Block coarsening-only performs better than the respective thread coarsening-only (outlined in black) and a combination of the two is required to reach the peak performance (highlighted). Block coarsening above 26 exceeded the maximum shared memory.

We report a median of 5 runs for composite measurements and a median of 3 runs for kernel measurements. Measurements with runtimes less than 0.0001s are discarded.

Timing-driven optimization (autotuning) is applied unless specified otherwise.

B. Combining Block and Thread Coarsening

To evaluate the combination of block and thread coarsening transformations, we contrast it with thread and block coarsening applied separately. For this experiment, we independently specified *total* factors of 1, 2, 4, 8, 16, and 32 for thread and block coarsening. After discarding kernels with too short runtimes, we gathered results for 181 kernel variations. Overall, the combined coarsening achieves 11.3% speedup, thread coarsening alone 4.4%, and block coarsening alone 8.9%.

106 of the kernels do not achieve a meaningful ($> 1\%$) speedup with either strategy. The remaining 75 which exhibit a speedup in at least one strategy are shown in in Fig. 13. . In many cases, thread coarsening alone cannot achieve the full performance achieved by a both thread and block coarsening.

To get better insight into how these approaches differ, let us focus on a specific benchmark—`lud` from Rodinia—and investigate how kernels are impacted by the different configurations on the A100 GPU. This benchmark had a significant difference of performance between thread-only and the combined coarsening approach, though thread coarsening still achieved a speedup. Fig. 14 illustrates the runtimes of the main `lud` kernel for different total thread and block factors. We find that a combination of both was necessary to achieve the peak performance at (block, thread) factors of (7, 2) for a combined coarsening factor of 14. This is particularly interesting as the best block factor of 7 is a prime! Moreover, for same factors, block-only coarsening achieves better performance than thread-only. The kernel originally had 256 threads. Coarsening it with a factor of 16 or 32 fails to produce a full warp of 32 threads. This explains the performance gap in Fig. 14 between thread coarsening factors that create a full warp (≤ 8) and those that don't (≥ 16).

TABLE II
PROFILING DATA FOR `LUD`.

(block, thread) factors	(1, 1)	(4, 1)	(1, 4)
Runtime	0.184 s	0.122 s	0.139 s
LSU utilization	51%	65%	27%
FMA utilization	24%	27%	26%
L2 $\rightarrow$ L1 Read	583 MB	460 MB	582 MB
L1 $\rightarrow$ L2 Write	267 MB	266 MB	371 MB
L1 $\rightarrow$ SM Read Req.	6.27	4.16 M	6.27 M
SM $\rightarrow$ L1 Write Req.	2.09 M	2.08 M	2.09 M
ShMem $\rightarrow$ SM Read Req.	41.78 M	41.62 M	12.53 M
SM $\rightarrow$ ShMem Write Req.	4.18 M	4.16 M	2.09 M

LSU = load/store unit, FMA = fused multiply/add unit, SM = streaming multiprocessor, ShMem = shared memory, L*N* = cache of level *N*.

The finer granularity of adjustment permitted by block coarsening[10] allowed us to maximize performance.

To analyze how the two approaches affect the computational characteristics of the kernel, we profiled it with NVIDIA Nsight Compute using the configurations with (block, thread) factors of (4, 1) and (1, 4), which have a performance difference of roughly 14% and extracted points of interest in the profiling data in Table II.

The data movement statistics suggest that thread coarsening alone leads to underutilization of the shared memory and requires more transfers between L2 and L1 caches. There was no difference in the total data transferred from global memory. In contrast, our block coarsening transformation was able to make use of the underutilized shared memory and in turn improve performance.

To gain more insight into the finer-grain control we have over block coarsening we experimented with independently varying the coarsening factors along the x and y block dimensions (the `lud_internal` kernel uses two block dimensions). Due to the memory access patterns in the kernel, coarsening in the x direction preserved memory locality better and resulted in

[10]The factor does not need to be a divisor of the grid size (see Section V).

better overall performance and the peak was achieved at a factor of 9 with a speedup of $1.64\times$.

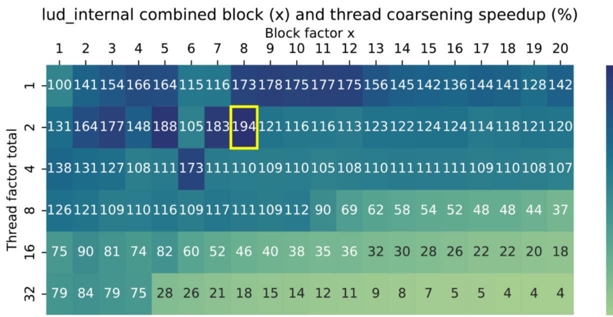

Fig. 15. Peak performance of the `lud` main kernel was achieved by combining both block coarsening in the x dimension and thread coarsening.

Additionally enabling thread coarsening improved performance to reach peak speedup to $1.94\times$ at (block, thread) factors of (2, 8). The performance variation seen in Fig. 15 supports the need for autotuning to get around sudden performance dips (e.g., factors (6, 2)) that are attributable to poor cache utilization.

C. Comparison against a Mainstream CUDA Compiler

Since Polygeist-GPU and clang share the same frontend and backend, we evaluate the impact of our optimizations by comparing composite runtimes produced by the two compilers.

As illustrated in Fig. 16, in absence of optimization the generated code has similar performance on different NVIDIA GPUs regardless of the compiler thanks to the shared front- and back-end. Two exceptions are `lavaMD` on the A100, and `srad_v1 reduce` on the A4000 (and to a smaller degree on the A100). A detailed analysis of the generated code indicates that `lavaMD` speedup is due to Polygeist having better loop invariant code motion with respect to GPU shared memory allowing it to hoist multiple shared memory loads out of the innermost compute loop, which in turn dramatically improved the memory characteristics of the kernel. In turn, the `srad_v1 reduce` performance difference was due different address computation order in the innermost loop which contains an intensive shared memory load-store sequence. This resulted in differences in register allocation and usage for address computation, causing the performance difference.

Parallel optimizations described in this paper improve performance to achieve overall speedups of 17% or 27% depending on the GPU model. For example, `gaussian` contains a kernel with a low arithmetic intensity and significant divergence launched with block size of 16, which fails to saturate available resources and even run on in a full warp. Block coarsening is able to significantly improve performance by making individual threads perform more work and thus use more of the available resources.

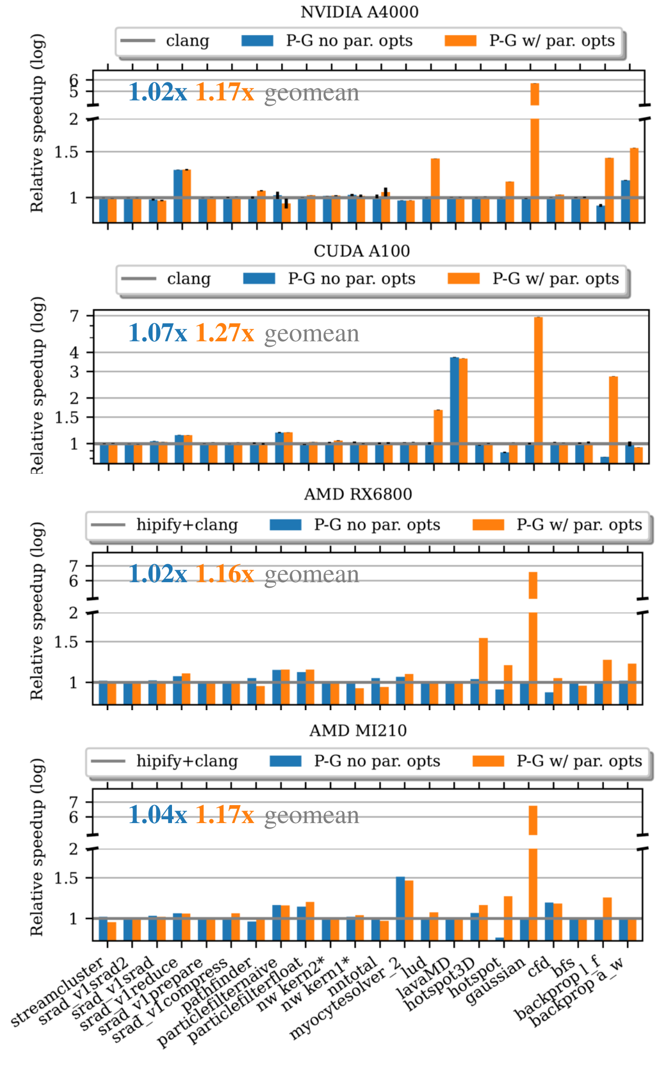

Fig. 16. Polygeist-GPU (P-G) without additional optimizations yields performance similar to baseline clang on NVIDIA GPUs. Optimizations result in 17–27% performance increase. Transpiled to AMD GPUs, our approach achieves a 16–17% improvement over `hipify`'ed CUDA compiled by clang.

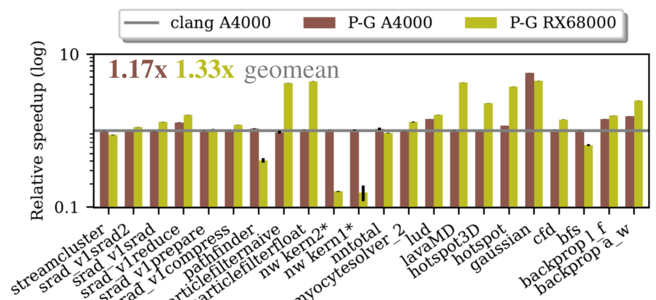

Fig. 17. Rodinia benchmarks compiled by Polygeist for NVIDIA A4000 and AMD RX6800 and clang on A4000.

D. Translation to AMD: hipify+clang vs. Polygeist-GPU

In order to accurately measure the impact of our flow in NVIDIA-to-AMD translation, we need to compare against clang as we share the backend with it. However, clang does not support such cross-compilation. Instead, we use the *hipify* source-to-source translation tool [23] provided by AMD to rewrite CUDA to HIP (corresponding API for AMD GPUs) before invoking clang as a baseline.

1) Ease of use: First, we compare the ease of use of hipify+clang and Polygeist-GPU for compilation.

The first step in the hipify+clang translation flow is to run hipify on the input CUDA source files. In this step, we also had to run hipify on external header dependencies (some Rodinia benchmarks depend on helper headers from the CUDA samples provided by NVIDIA). This step is entirely unnecessary when using Polygeist-GPU.

Hipify required some manual fixes to the source files to enable compilation. We needed to specify missing `#include` statements to the HIP runtime headers and we also had to remove `#ifdef` guards from some headers. These guards function as appropriate when the CUDA runtime headers are used, and thus were handled appropriately by Polygeist-GPU where the frontend compilation happens as if we are compiling for CUDA and only at the IR level do we convert CUDA specific constructs to their HIP counterparts. On the other hand, when using hipify+clang, the CUDA headers must be swapped to their HIP counterparts from the beginning of the pipeline and more intricate usage of C/C++ preprocessor features that depend on the CUDA header structure cannot be converted automatically.

For both approaches, we needed to specify the new appropriate command line flags to the compiler. For both hipify+clang and Polygeist-GPU, these were specifying the target GPU, the new HIP libraries we need to link against, and the installation location of HIP. For Polygeist-GPU, we also add a flag to instruct the compiler to translate to AMD.

Finally, the code can be compiled with the respective compiler and will result in an executable for AMD GPUs.

This shows how our approach of translating target specific code at the IR level can provide a more user-friendly process that can proceed with little manual intervention.

2) Performance: Analogous to the previous section, we compare three ways to compile the Rodinia benchmark suite: (hipify+)clang, Polygeist-GPU with the parallel representation and parallel optimizations disabled and enabled. We use this comparison to check how well Polygeist-GPU can adapt kernels for a different GPU architecture by a different vendor, in this case, the AMD RX6800 and AMD MI210. The results are presented in Fig. 16.

Our flow produces a geomean speedup of 17% (depending on GPU) when optimizations are enabled. However, in this case we can observe different behavior in some benchmarks compared to the CUDA case, demonstrating how different architectures may require different tuning configurations to maximize performance.

To gain insight into how the performance may differ between GPUs from different vendors, we compare the performance of a CUDA GPU and an AMD GPU Fig. 17 with a baseline of the performance of clang on the CUDA GPU. We use the NVIDIA A4000 and AMD RX6800 GPUs (see Table I) which have comparable specifications (A4000 has about 10% more single-precision floating point compute power, while the AMD GPU 11% more memory bandwidth and 60% more double-precision floating point compute power). The RX6800 (Polygeist-GPU) performs 25% (geomean) better than A4000 (clang) and 9% faster than A4000 (Polygeist-GPU).

We investigated the discrepancy in the performance of `nw_*`. Both kernels have a block size of 16 and allocate 2180 bytes per block, for a ratio of 136 bytes of shared memory per thread. This is extremely high as, for example, the next heaviest user of shared memory in Rodinia is `lud`, containing a kernel that uses 12 bytes of shared memory per thread, which is more inline with typical GPU workloads. Even though we generate shared memory in our LLVM backend, when we profile the kernel on AMD GPUs, we observe no usage of shared memory, which indicates that the AMD backend optimizer has offloaded this shared memory to global memory. We hypothesize that this is due to the very small amount of L1 cache available compared to NVIDIA GPUs (Table I), and occupancy would be severely limited if the offloading was not done and indeed, when we disabled it, the resulting kernel performed 15 times worse.

The discrepancies in favor of AMD (`particlefilter*`, `lavaMD`, `hotspot3D`) are due to usage of double precision floating point arithmetic. This can be explained by the greater compute power for this specific case available in the AMD RX6800 compared to the NVIDIA A4000 (see Table I).

VIII. Related Work

A. Thread and Block Coarsening

Thread coarsening as an optimization for GPUs was first discussed by Volkov [17] as a manual optimization to hide that latency. Later, Barua, Shirako, and Sarkar [18] and Magni, Dubach, and O'Boyle [19, 20] implemented this as an automatic transformation for OpenCL and OpenACC.

Block coarsening transformation was first proposed by Unkule, Shaltz, and Qasem [24] and first implemented by Stawinoga and Field [25]. However, prior work did not discuss the legality of the block coarsening in the presence of block-level syncrhonization. Moreover, our work is the first to combine both thread and block coarsening, which demonstrates compounding benefits and is simplified in our parallel representation.

In addition, various heuristics for choosing the optimal factors have been described [18, 20, 25], however, we were not able to readily apply these to our combined coarsening transformation approach and we have left this to future work.

B. Granularity Control

Certain domain specific languages (DSLs) or programming frameworks provide control of the granularity of computations.

Triton [1] is a programming model that provides an abstraction for a *tile* and is able to seamlessly scale its granularity to tune computation to the target hardware. Kokkos [26] and RAJA [27] are programming models that provides a variety of parallel abstractions such as multi-level parallel primitives and are able to generate high-performance code for GPUs. Halide [28] provides a way to specify computation and schedule separately, which in a way allows for granularity control of computation in order to best fit to the target hardware.

However, these all require the programmer to rewrite their software in their new programming model which is often laborious and may not be able to represent arbitrary programs. In contrast, our solution works on existing CUDA code.

C. Translation to AMD

Hipify [23] is a source-to-source tool to translate CUDA code to run on AMD GPUs at either the source code or abstract syntax tree (AST) stage. Both of these representations have drawbacks. Source-based translation fails to handle complex language features such as macros or templates, whereas AST-based translation does not work correctly when the preprocessor options (e.g. `#defines`) are different when compiling for AMD and CUDA. In contrast, we work at an IR level which can solve the problems of both approaches.

OpenCL [10], SYCL [6], and OpenACC [29] all provide a target-agnostic way to write GPU code, however they still require a rewrite if the software is already written in CUDA.

Doerfert et al. [30] propose translating CUDA code to OpenMP, and then using the existing OpenMP offloading infrastructure in LLVM [31] to target different hardware such as NVIDIA or AMD GPUs. Their usage of OpenMP as a common parallel representation is analogous to our parallel representation. Our approach however, makes it easier to preserve GPU specific notions such as for example constant device memory, synchronisation primitives, shared memory and involve them in optimizations.

IX. Conclusion

GPUs are increasingly critical to HPC, especially with recent advances in AI. However, the variety of vendors and accordingly architectures is increasing in recent years with many HPC systems now choosing to use AMD GPUs. This software and hardware ecosystem presents a problem for programmers continuously rewrite applications to keep up with different architectures – a costly and time-consuming process! To efficiently enable existing programs to leverage hardware-specific parameters, we combine fine-grained thread and block coarsening within the compiler. A compiler-based solution can operate at a finer scale than previously possible, and have successfully applied it to retune both CUDA code to a different CUDA architecture and AMD architectures as well. We achieve up to a 27% geomean performance improvement on the Rodinia benchmark suite and were able to automatically translate it to run on an AMD GPU without sacrificing performance. This solution can help prevent vendor lock-in and reduces the cost associated with porting software to different hardware.

Data Availability Statement

The source code for our compiler and benchmarks that support the findings of this work are openly available in Zenodo at https://doi.org/10.5281/zenodo.10465934 [32].

Acknowledgment

This work was supported by JST SPRING, Grant Number JPMJSP2106, and the RIKEN Junior Research Associate Program. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

Artifact Appendix

A. Abstract

This artifact contains the source code for our compiler tool (Polygeist), the benchmarks suite we evaluated our transformation on (Rodinia + HeCBench), and scripts for automatic compilation, execution, and evaluation. It also contains the data set and code dependencies for the benchmark suite. We have implemented a combined thread and block coarsening transformation and timing-driven optimization of GPU kernels in Polygeist/MLIR. We compare how our combined coarsening approach fares against a traditional thread coarsening-only approach and against a mainstream compiler (clang) on Rodinia and HeCBench.

B. Artifact Check-List (Meta-Information)

We used three systems to conduct our experiments:

- **System 1:** Intel(R) Xeon(R) Gold 6252 CPU * 2, 192 GB RAM, NVIDIA A100 PCIe 40GB, running AlmaLinux 8.4.
- **System 2:** Intel(R) Xeon(R) Silver 4215 CPU * 2, 384 GB RAM, AMD MI250, running AlmaLinux 8.4.
- **System 3:** AMD EPYC 7302 16-Core CPU, 256 GB RAM, AMD RX6800, NVIDIA RTX A4000, running Fedora 37.

Other information:

- **Program:** Rodinia and HeCBench.
- **Compiler:** Polygeist (our work), clang (for comparison).
- **Transformations:** Coarsening transformation implemented in MLIR.
- **Data set:** Provided as an artifact.
- **Output:** Benchmark timing information and figures.
- **How much disk space required?:** Approximately 30GB
- **How much time is needed to prepare workflow?:** 1 to 6 hours (highly dependent on the compilation performance)
- **How much time is needed to complete experiments?:** Approximately 2 to 3 days.
- **Publicly available?:** Yes, listed below.
- **Archived (provide DOI)?:** 10.5281/zenodo.10465934

C. Experiment Summary

There are three experiments we conducted in our work:

- **Experiment 1: Fig. 13:** The main experiment. This compares our combined block and thread coarsening approach against thread coarsening, conducted on System 1.
- **Experiment 2: Fig. 16:** Comparison of CUDA code compiled by our compiler against CUDA code compiled by clang conducted on System 1 and System 3.
- **Experiment 3: Fig. 16:** Comparison of CUDA code translated to HIP and compiled by our compiler against hipified HIP code compiled by clang conducted on System 2 and System 3.

D. Dependencies

1) Hardware Dependencies: AMD and/or NVIDIA GPU supported by clang 16.

2) Software Dependencies: A Linux system able to build clang 16. LLD. CUDA 11.4. CUDA 12.1. ROCm 5.3.4 (ROCm not required for CUDA-only experiments).

E. Artifact

The combined artifact archive is available at https://doi.org/10.5281/zenodo.10465934. It consists of the following components:

1) Polygeist: The source code of our tool. It is publicly available at https://github.com/llvm/Polygeist (commit `ba9953a08c9b`)

2) Evaluation Benchmarks: Available at https://github.com/ivanradanov/rodinia (commit `a97759e7`).

3) Benchmark Data Set and Dependencies: They can be found in the combined artifact in the following directories: `data`, `cuda-10.2-samples`, and `cuda-10.2-hip-samples`.

F. Installation

1) Obtaining the Code:

```
$ cd $HOME && git clone https://github.com/llvm/Polygeist
$ cd Polygeist
$ git checkout ba9953a08c9b
$ git submodule update --init --recursive
```

2) Building LLVM: We first need to build the LLVM compiler toolchain which our tool depends on.[11][12]

```
$ cd $HOME/Polygeist
$ mkdir mlir-build && cd mlir-build
```

The cmake configuration step depends on the GPUs available on the system.

For CUDA:

```
$ CUDACXX=<path_to_nvcc-11.4> \
  CUDA_PATH=<path_to_cuda-11.4> \
  cmake ../llvm-project/llvm -GNinja \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_PROJECTS="mlir;clang;openmp" \
  -DLLVM_TARGETS_TO_BUILD="host;NVPTX;AMDGPU" \
  -DMLIR_ENABLE_CUDA_RUNNER=1 -DLLVM_USE_LINKER=lld
```

For AMD:[13]

```
$ ROCM_PATH=<path_to_rocm> \
  cmake ../llvm-project/llvm -GNinja \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_PROJECTS="mlir;clang;openmp" \
  -DLLVM_TARGETS_TO_BUILD="host;NVPTX;AMDGPU" \
  -DHIP_CLANG_INCLUDE_PATH=<hip_clang_include_dir> \
  -DMLIR_ENABLE_ROCM_RUNNER=1 -DLLVM_USE_LINKER=lld
```

For systems with both AMD and CUDA GPUs use both the AMD and CUDA specific configuration arguments.

Finally, compile:

```
$ ninja
$ export MLIR_BUILD_DIR="$(pwd)"
```

3) Building Polygeist:

```
$ cd $HOME/Polygeist
$ mkdir build && cd build
```

The cmake configuration step depends on the GPUs available on the system.

For a system with a CUDA GPU:

```
$ CUDACXX=<path_to_nvcc-11.4> \
  CUDA_PATH=<path_to_cuda-11.4> \
  cmake ../ -GNinja \
  -DMLIR_DIR=$MLIR_BUILD_DIR/lib/cmake/mlir \
  -DLLVM_EXTERNAL_LIT=$MLIR_BUILD_DIR/bin/llvm-lit \
  -DClang_DIR=$MLIR_BUILD_DIR/lib/cmake/clang \
  -DCMAKE_BUILD_TYPE=Release \
  -DPOLYGEIST_ENABLE_CUDA=1 \
  -DCMAKE_CUDA_COMPILER=<path_to_nvcc> \
  -DLLVM_USE_LINKER=lld
```

For a system with an AMD GPU:[13] [14]

```
$ ROCM_PATH=<path_to_rocm> \
  cmake ../ -GNinja \
  -DMLIR_DIR=$MLIR_BUILD_DIR/lib/cmake/mlir \
  -DLLVM_EXTERNAL_LIT=$MLIR_BUILD_DIR/bin/llvm-lit \
  -DClang_DIR=$MLIR_BUILD_DIR/lib/cmake/clang \
  -DCMAKE_BUILD_TYPE=Release \
  -DPOLYGEIST_ENABLE_ROCM=1 \
  -DHIP_CLANG_INCLUDE_PATH=<hip_clang_include_dir> \
  -DCMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES=<cuda_12.1_
                                       include_dir> \
  -DLLVM_USE_LINKER=lld
```

Depending on the system configuration, the required variables specifying CUDA or ROCm paths may vary.

For a system with both AMD and CUDA GPUs, we need to specify both the ROCm and CUDA specific cmake configuration flags and environmental variables.

Finally, we execute the build step:

```
$ ninja
```

G. Experiment Workflow

1) Obtaining the Benchmarks:

```
$ cd $HOME
$ git clone https://github.com/ivanradanov/rodinia
$ cd rodinia
$ git checkout a97759e7
```

This repository includes not only Rodinia but also the HeCBench CUDA benchmarks.

2) Obtaining Data Sets and Dependencies: These can be obtained from the combined artifact. The data set directory `data` should be put under the benchmark root directory `rodinia`. Some of the benchmarks depend on the cuda samples and their hipified version which are at `cuda-10.2-samples`, and `cuda-10.2-hip-samples` in the combined artifact. They can be anywhere and the path to them needs to be provided in a configuration file (explained below).

3) Setting Up the Benchmarks: Our benchmark repository uses configuration files in `rodinia/common/` to specify the Polygeist, Clang/LLVM installations. The config files for the systems we used are {`memkf02`,`memkf01`,`supercomp01a`}`.polygeist.host.make.config` for System 1, 2, and 3 respectively. The structure of the filename must be kept the same, with the first substring in the name representing the machine's hostname.

The variables defined in this file are as follows:

- `POLYGEIST_DIR_RELEASE` The Polygeist build directory (`$HOME/Polygeist/build/` in this case).
- `POLYGEIST_LLVM_DIR_RELEASE` The LLVM build directory (`$HOME/Polygeist/mlir-build/` in this case).

CUDA specific:

- `CUDA_PATH` Path to the CUDA installation
- `CUDA_GPU_ARCH` The CUDA GPU architecture (e.g. `sm_86`).
- `CUDA_SAMPLES_PATH_` Path to the CUDA samples (required by some benchmarks).

AMD specific:

[11] Note that the linking step requires a large amount of memory. Limiting the concurrent processes used by `ninja` by using `-j <num_procs>` can alleviate this problem.

[12] Note we use an older LLVM version with a backported patch. The commit the llvm-project repository needs to be at the exact commit as specified in the git submodule, which will be automatically pulled using the provided command.

[13] The `<hip_clang_include_dir>` refers to the include directory of the clang compiler provided in the ROCm installation, in our case it was the following: `/opt/rocm/llvm/lib/clang/15.0.0/include/`

[14] Note that we must provide a cuda include path which is needed in order to generate the CUDA to HIP translation layer. We only support CUDA 12.1 here.

- `ROCM_PATH` Path to the ROCm installation.
- `AMD_GPU_ARCH` The AMD GPU architecture (e.g. `gfx1030`).
- `HIPIFIED_CUDA_SAMPLES_PATH_` Path to the hipified CUDA samples.

Please see the existing config files above for examples.

4) Running the Experiments: The experiments can be run using the `scripts/run_all_gpu_benches.sh` script in the benchmark repository. Some of the options refer to the Polygeist configurations which are defined in `common/common.polygeist.host.make.config`. Refer to the paper for the meaning of the different timing types. It accepts the following arguments:

- Polygeist-specific
  - `--targets` List of space-separated targets to compile for and run, supported options are `AMDGPU` and `CUDA`
  - `--pgo-configs` The Polygeist configuration(s) to use for TDO kernel timing runs.
  - `--pgo-prof-nruns` Number of TDO kernel timing runs.
  - `--configs` The Polygeist configuration(s) to use for composite timing runs.
- Other
  - `--cuda-benchmarks` The file specifying the CUDA benchmarks to run.
  - `--hip-benchmarks` The file specifying the hipified CUDA benchmarks to run.
  - `--clang` Benchmark using clang.
  - `--hip-clang` Benchmark the hipified CUDA versions using clang.
  - `--nruns` The number of composite timing runs.
  - `--host` The hostname of the machine we are running on - the corresponding config file we defined above will be used.
  - `--dry-run` Only print out what will be done.

The script *must* be run from the root directory of the benchmark repository. We strongly recommend running the script in a `tmux` or `screen` session, especially when on a remote machine as it takes an extremely long time to complete.

Experiment 1 (requires a CUDA GPU, roughly 34 hours):

```
$ cd $HOME/rodinia
$ ./scripts/run_all_gpu_benches.sh \
  --targets CUDA --host <host_name> \
  --cuda-benchmarks ./scripts/cuda_all_apps.sh \
  --pgo-prof-nruns 3 --pgo-configs 15
```

Experiment 2 (requires a CUDA GPU, roughly 3 hours):

```
$ cd $HOME/rodinia
$ ./scripts/run_all_gpu_benches.sh \
  --targets CUDA --host <host_name> \
  --cuda-benchmarks ./scripts/rodinia_cuda_apps.sh \
  --pgo-prof-nruns 3 --nruns 5 \
  --pgo-configs 11 --configs 2 --clang
```

Experiment 3 (requires an AMD GPU, roughly 3 hours):

```
$ cd $HOME/rodinia
$ ./scripts/run_all_gpu_benches.sh \
  --targets AMDGPU --host <host_name> \
  --cuda-benchmarks ./scripts/rodinia_cuda_apps.sh \
  --hip-benchmarks ./scripts/hip_all_apps.sh \
  --pgo-prof-nruns 3 --nruns 5 \
  --pgo-configs 11 --configs 2 --hip-clang
```

All of these will output the results in a timestamped subdirectory in `$HOME/rodinia_results/`. Each subdirectory will have a `results/cmd` file which contains the configuration of the benchmarking run.

H. Evaluation (Generating the Figures)

To generate the figures from the paper, we can use the scripts included in the benchmark repository. They will output the generated figures in `rodinia/plots/figures/`.

For experiment 1:

```
$ cd $HOME/rodinia
$ ./scripts/plots/rodinia-kernel-alt-analysis.py \
  ~/rodinia_results/rodinia_results_<timestamp>/
```

For experiment 2:

```
$ cd $HOME/rodinia
$ ./scripts/plots/rodinia.py \
  ~/rodinia_results/rodinia_results_<timestamp>/
```

For experiment 3:

```
$ cd $HOME/rodinia
$ ./scripts/plots/hip-rodinia.py \
  ~/rodinia_results/rodinia_results_<timestamp>/
```

References

[1] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. “Triton: an intermediate language and compiler for tiled neural network computations”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019, pp. 10–19. URL: https://doi.org/10.1145/3315508.3329973.

[2] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. “Polyhedral parallel code generation for CUDA”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), pp. 1–23. URL: https://doi.org/10.1145/2400682.2400713.

[3] Roy Frostig, Matthew James Johnson, and Chris Leary. “Compiling machine learning programs via high-level tracing”. In: *Systems for Machine Learning* 4.9 (2018). URL: https://mlsys.org/Conferences/doc/2018/146.pdf.

[4] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. “The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically”. In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). ISSN: 1544-3566. DOI: 10.1145/3355606. URL: https://doi.org/10.1145/3355606.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. 2018. arXiv: 1802.04799 [cs.LG].

[6] Ruyman Reyes and Victor Lomüller. “SYCL: Single-source C++ accelerator programming”. In: *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[7] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. “Polygeist: Raising C to Polyhedral MLIR”. In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2021, pp. 45–59. DOI: 10.1109/PACT52795.2021.00011.

[8] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. “High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs”. In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPoPP ’23. Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 119–134. ISBN: 9798400700156. DOI: 10.1145/3572848.3577475. URL: https://doi.org/10.1145/3572848.3577475.

[9] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[10] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming”. In: *Parallel Computing* 38.8 (2012), pp. 391–407. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0167819111001335.

[11] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. “Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks”. In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, pp. 290–298. DOI: 10.1109/ICCD53106.2021.00054.

[12] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. “PERKS: A Locality-Optimized Execution Model for Iterative Memory-Bound GPU Applications”. In: *Proceedings of the 37th International Conference on Supercomputing*. ICS ’23. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 167–179. ISBN: 9798400700569. DOI: 10.1145/3577193.3593705. URL: https://doi.org/10.1145/3577193.3593705.

[13] NVIDIA. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ (visited on 07/31/2023).

[14] Xinyao Yi, David Stokes, Yonghong Yan, and Chunhua Liao. "CUDAMicroBench: Microbenchmarks to Assist CUDA Performance Programming". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 397–406. DOI: 10.1109/IPDPSW52791.2021.00068.

[15] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. "Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation". In: *ACM Trans. Archit. Code Optim.* 18.4 (Sept. 2021). ISSN: 1544-3566. DOI: 10.1145/3469030. URL: https://doi.org/10.1145/3469030.

[16] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. "Structured Operations: Modular Design of Code Generators for Tensor Compilers". In: *Languages and Compilers for Parallel Computing: 35th International Workshop, LCPC 2022, Chicago, IL, USA, October 12–14, 2022, Revised Selected Papers*. Chicago, IL, USA: Springer-Verlag, 2023, pp. 141–156. ISBN: 978-3-031-31444-5. DOI: 10.1007/978-3-031-31445-2_10. URL: https://doi.org/10.1007/978-3-031-31445-2_10.

[17] Vasily Volkov. "Understanding Latency Hiding on GPUs". PhD thesis. EECS Department, University of California, Berkeley, Aug. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html.

[18] Prithayan Barua, Jun Shirako, and Vivek Sarkar. "Cost-Driven Thread Coarsening for GPU Kernels". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243196. URL: https://doi.org/10.1145/3243176.3243196.

[19] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. "A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503268. URL: https://doi.org/10.1145/2503210.2503268.

[20] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic Optimization of Thread-Coarsening for Graphics Processors". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14. Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 455–466. ISBN: 9781450328098. DOI: 10.1145/2628071.2628087. URL: https://doi.org/10.1145/2628071.2628087.

[21] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[22] Zheming Jin and Jeffrey S. Vetter. "A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model". In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2023, pp. 325–327. DOI: 10.1109/ISPASS57527.2023.00041.

[23] AMD. *HIPify*. URL: https://github.com/ROCm-Developer-Tools/HIPIFY (visited on 01/18/2023).

[24] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. "Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality". In: Mar. 2012, pp. 21–40. ISBN: 978-3-642-28651-3. DOI: 10.1007/978-3-642-28652-0_2.

[25] Nicolai Stawinoga and Tony Field. "Predictable Thread Coarsening". In: *ACM Trans. Archit. Code Optim.* 15.2 (June 2018). ISSN: 1544-3566. DOI: 10.1145/3194242. URL: https://doi.org/10.1145/3194242.

[26] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731514001257.

[27] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. "RAJA: Portable Performance for Large-Scale Scientific Applications". In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, pp. 71–81. DOI: 10.1109/P3HPC49587.2019.00012.

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[29] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. "Achieving Portability and Performance through OpenACC". In: *2014 First Workshop on Accelerator Programming using Directives*. 2014, pp. 19–26. DOI: 10.1109/WACCPD.2014.10.

[30] Johannes Doerfert, Marc Jasper, Joseph Huber, Khaled Abdelaal, Giorgis Georgakoudis, Thomas Scogland, and Konstantinos Parasyris. "Breaking the Vendor Lock: Performance Portable Programming through OpenMP as Target Independent Runtime Layer". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. Chicago, Illinois: Association for Computing Machinery, 2023, pp. 494–504. ISBN: 9781450398688. DOI: 10.1145/3559009.3569687. URL: https://doi.org/10.1145/3559009.3569687.

[31] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[32] Ivan R. Ivanov, Oleksandr Zinenko, Jens Domke, Toshio Endo, and William S. Moses. *Retargeting and Respecializing GPU Workloads for Performance Portability - Artifact*. Jan. 2024. DOI: 10.5281/zenodo.10465934. URL: https://doi.org/10.5281/zenodo.10465934.