Instruction Scheduling for the GPU on the GPU

Ghassan Shobaki[1], Pınar Muyan-Özçelik[1], Josh Hutton[1], Bruce Linck[1],
Vladislav Malyshenko[1], Austin Kerbow[2], Ronaldo Ramirez-Ortega[1], and Vahl Scott Gordon[1]

[1]Department of Computer Science, *California State University, Sacramento*, United States
[2]GPU-Compute Compiler Team, *Advanced Micro Devices*, United States
ghassan.shobaki@csus.edu, pmuyan@csus.edu, gordonvs@csus.edu

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444869

***Abstract*—In this paper, we show how to use the GPU to parallelize a precise instruction scheduling algorithm that is based on Ant Colony Optimization (ACO). ACO is a nature-inspired intelligent-search technique that has been used to compute precise solutions to NP-hard problems in operations research (OR). Such intelligent-search techniques were not used in the past to solve NP-hard compiler optimization problems, because they require substantially more computation than the heuristic techniques used in production compilers. In this work, we show that parallelizing such a compute-intensive technique on the GPU makes using it in compilation reasonably practical. The register-pressure-aware instruction scheduling problem addressed in this work is a multi-objective optimization problem that is significantly more complex than the problems that were previously solved using parallel ACO on the GPU. We describe a number of techniques that we have developed to efficiently parallelize an ACO algorithm for solving this multi-objective optimization problem on the GPU. The target processor is also a GPU. Our experimental evaluation shows that parallel ACO-based scheduling on the GPU runs up to 27 times faster than sequential ACO-based scheduling on the CPU, and this leads to reducing the total compile time of the rocPRIM benchmarks by 21%. ACO-based scheduling improves the execution-speed of the compiled benchmarks by up to 74% relative to AMD's production scheduler. To the best of our knowledge, our work is the first successful attempt to parallelize a compiler optimization algorithm on the GPU.**

***Index Terms*—parallel compiler optimization, instruction scheduling, Ant Colony Optimization (ACO), GPU computing, multi-objective optimization**

I. Introduction

Some important compiler optimization problems, such as instruction scheduling and register allocation, are NP-hard problems [1]. An NP-hard problem has no known polynomial-time algorithm that computes the exact solution to every instance of the problem. Production compilers solve these NP-hard optimization problems using heuristic approaches, as it has been widely believed that implementing precise algorithms for NP-hard problems within a compiler is impractical.

In Operations Research (OR), researchers have successfully computed precise, and often exact, solutions to NP-hard problems using Artificial Intelligence (AI) techniques, including Branch-and-Bound (B&B), Constraint Programming (CPR) and Ant Colony Optimization (ACO). Such techniques are based on intelligent searches that require substantially more computation than heuristic approaches. Despite the success of these intelligent-search techniques in OR, they have not been applied to NP-hard compiler optimization problems, because that was not feasible in the past. However, today's powerful computing platforms have motivated some researchers to explore applying AI techniques to NP-hard problems in code optimization [2]–[11]. The results of this recent research show that applying AI techniques may produce code with significantly better performance. For example, Shobaki et al. show that solving instruction scheduling using B&B gives significantly better performance than a well-tuned heuristic on an AMD Graphics Processing Unit (GPU) target [10].

Recent research on applying AI techniques to compiler optimizations shows that the increase in compile time caused by using such expensive techniques can be controlled by applying them selectively to the hot code and setting reasonable time limits [8], [9]. Although such measures can limit the increase in compile time, using AI techniques still leads to a significant increase in compile time relative to using heuristics.

In this work, we explore using the GPU to parallelize an AI technique for solving the register-pressure-aware (RP-aware) instruction scheduling problem. The AI technique that we parallelize is ACO [12]. We perform ACO-based scheduling on the GPU, while the rest of the compilation stages are performed on the CPU. The GPU is also the target processor that we generate code for. So, we show how to schedule for the GPU using an AI technique that is parallelized on the GPU.

In RP-aware instruction scheduling, the objective is finding an instruction order that achieves the best possible balance between the two conflicting objectives of minimizing the schedule length and minimizing register pressure. Minimizing the schedule length can be also viewed as maximizing instruction-level parallelism (ILP). Maximizing ILP is important on a GPU target, because a GPU does not reorder instructions within a thread. Register pressure (RP) is the number of virtual registers that have overlapping live ranges, and thus must be assigned to different physical registers. Minimizing RP is particularly important on a GPU target, because the number of registers used in each thread determines *occupancy*, which is the number of thread groups that may run concurrently, as detailed in the body of the paper.

Balancing ILP and RP is a challenging problem, because executing more independent instructions in parallel (maximizing ILP) tends to increase the demand for registers. Even optimizing one of these two objectives (ILP or RP) is NP-hard [1]. Current production compilers solve this problem using heuristics (usually greedy heuristics). However, recent research

979-8-3503-9509-9/24/$31.00 © 2024 IEEE

on both CPUs [7], [9] and GPUs [10], [13] has shown that these heuristics may, in some cases, produce poor schedules that significantly degrade performance. Previous work shows that instruction scheduling has a higher impact on the performance of GPU programs than on the performance of CPU programs.

ACO is a population-based optimization technique inspired from nature. Ants in nature find short paths between a food source and their nest by depositing pheromones as they carry food. In an ACO algorithm, a pheromone table is used to simulate the deposition and dissipation of pheromones. Dorigo et al. [14] introduced ACO and applied it to the Traveling Salesman Problem (TSP). In later research, ACO was applied to other combinatorial optimization problems.

In previous work, an ACO algorithm was proposed for solving the RP-aware instruction scheduling problem [11] based on the ACO algorithm described by Gambardella and Dorigo [15] for solving the Sequential Ordering Problem (SOP), which is a generalization of the TSP [16]. The ACO algorithm for the RP-aware scheduling problem capitalizes on the similarities between scheduling and the SOP.

Despite these similarities, the RP-aware instruction scheduling problem is much more complex than the SOP. First, the SOP is a single-objective optimization problem, while the RP-aware scheduling problem is a multi-objective optimization problem (MOOP) [17], [18]. The first objective is minimizing RP and the second objective is minimizing the schedule length. Second, the RP-aware scheduling problem involves latency constraints, and thus the solution to a given instance of the problem is a schedule (an assignment of a cycle to each instruction) not just an order as in the SOP.

These complexities make parallelizing the ACO-based scheduling algorithm much more challenging than parallelizing the ACO algorithm for the SOP. Not all the techniques that have been proposed to parallelize the ACO algorithm for the SOP can be applied to RP-aware scheduling.

Parallelizing ACO-based scheduling is particularly challenging on the GPU, because different schedules may have different lengths, which increases *thread divergence*, a GPU-specific factor that affects performance. A main contribution of this paper is proposing a number of techniques for minimizing this divergence in ACO-based scheduling (Section V-B)

Parallel compilation has been tackled by researchers since the 1970s [19]. After the advent of GPU computing, GPUs have been used to accelerate some compiler algorithms. Most of these algorithms are analyses, such as the points-to analysis [20]–[22], inter-procedural data-flow analysis [23], higher-order control-flow analysis [24] and inter-procedural static analysis of large-scale system code [25].

To the best of our knowledge, our work is the first work on parallelizing a compiler optimization on the GPU. Some previous research tackled compiler-related algorithms, such as graph coloring [26], [27]. However, that work was applied to general graphs, not to conflict graphs in a compiler. Our work directly addresses a compiler optimization problem.

The results of our experimental evaluation show that parallel ACO-based scheduling on the GPU runs up to 27 times faster than sequential ACO-based scheduling on the CPU, and this leads to reducing the total compile time of the rocPRIM benchmarks [28] by 21% relative to using sequential ACO-based scheduling on the CPU. ACO-based scheduling improves the execution speed of the compiled benchmarks by up to 74% relative to AMD's production scheduler.

II. Background

A. Problem Definition

Instruction scheduling is a compiler optimization in which instructions are reordered to achieve better performance. Better performance is achieved by hiding latencies and minimizing register pressure. This paper focuses on the instruction scheduling pass that is invoked before register allocation (pre-allocation instruction scheduling).

In many compilers, including the LLVM compiler used in this work, scheduling is done within a basic block [1]. The input to the instruction scheduler is an instruction sequence with dependencies represented by a data dependence graph (DDG). In a DDG, a node represents an instruction, an edge represents a dependency and an edge label represents a latency. An example DDG is shown in Figure 1. The output is a schedule, which is an assignment of a machine cycle to each instruction. The objective is finding a schedule that achieves the best possible balance between minimizing the schedule length and minimizing RP. The schedule length is the number of cycles used in the schedule, and RP is modeled using the cost function described below.

The number of cycles in the schedule depends on the machine model. Our implementation of the proposed algorithm supports a general machine model. The experimental results, however, were produced using a simple machine model, in which the processor can issue one instruction of any type in each cycle. This simple model still captures instruction latencies, and this appears to be the most important consideration.

RP computation is based on the *Def* and *Use* sets of the scheduled instructions. The Def set of an instruction is the set of registers that are defined by that instruction, and the Use set is the set of registers that the instruction uses. Given an instruction schedule, the RP for a given data type at a given point in the schedule is the number of registers of that type that are live at that point.

Multiple cost functions have been used for representing RP during scheduling [8]–[10]. In this work, we use the adjusted peak register pressure (APRP) cost function that was introduced by Shobaki et al. [10] specifically for a GPU target.

The peak register pressure (PRP) of a given data type in a given schedule is the maximum value of that type's RP at any point in the schedule. On a GPU, multiple PRP values may give the same occupancy. The APRP of a given PRP value x is defined as the maximum PRP that gives the same occupancy as x. For example, on the AMD GPU used in this work, a PRP of 24 vector general-purpose registers (VGPRs) or less gives the maximum occupancy of 10, while PRP values in the range [25–28] give an occupancy of 9. Therefore, PRP values in the range [1–24] are mapped to an APRP of 24 and PRP values in the range [25–28] are mapped to an APRP of 28.

In previous work, two different approaches have been explored for solving the RP-aware scheduling problem, which is a two-objective optimization problem. The first approach is minimizing a weighted sum of the schedule length and the RP cost [8], [9]. The second approach is a two-pass approach in which RP is treated as a primary objective that is minimized in the first pass (the RP pass), while schedule length is treated as a secondary objective that is minimized in the second pass (the ILP pass) [10]. Since the two-pass approach was found to work better on the GPU [10], we use it in this work.

B. GPU Computing

In GPU computing, the GPU is utilized to accelerate compute-intensive applications with data parallelism in many different areas, including computer vision, machine learning and many more [29]. An ACO algorithm is a compute-intensive algorithm that involves data parallelism, since all ants perform the same computation but on different data.

HIP [30] is a GPU programming environment introduced by AMD, which allows writing portable code that can run on AMD or NVIDIA GPUs. A HIP application consists of a sequential part that is run on the CPU and a data-parallel part that is launched on the GPU as a *kernel*. A kernel is launched on the GPU as a grid of blocks with each block consisting of the same number of threads. Each thread executes the kernel code using a different data element, and all threads are executed on the Compute Units (CUs) in parallel. The GPU thread scheduler assigns different blocks to different CUs.

A block consists of *wavefront*s. A wavefront is a group of threads that are executed in lockstep. The corresponding term on NVIDIA GPUs is a *warp*. On the GPU used in this work, a wavefront consists of 64 threads. Each wavefront is executed by a single SIMD unit. In order to efficiently utilize the multiple SIMD units within a CU, each CU should have multiple wavefronts, preferably from different blocks. The SIMD *occupancy* of a kernel running on an AMD GPU is the number of wavefronts that can be resident on each SIMD unit. Hence, occupancy determines the number of wavefronts that can run concurrently on the GPU.

III. Previous Work

ACO is a population-based technique that was introduced by Dorigo et al. [14] for solving hard combinatorial optimization problems. Since an ACO algorithm has an inherent parallel nature, different parallelization strategies have been proposed. Pedemonte et al. [31] provide a comprehensive survey and a taxonomy of ACO parallelization strategies.

Many of the previously proposed parallel ACO algorithms target the TSP [14], [32]–[47]. However, parallel ACO algorithms have also been proposed for solving other optimization problems, including the Maximum-Weight Clique Problem [48], the Quadratic Assignment Problem [49], designing multiproduct batch plants [50] and resource-constrained job scheduling [51]. These approaches use different parallel architectures, including supercomputers [32], multi-core CPUs [36], [48], [49], [51], [52] and GPUs [34], [37]–[47], [50], [52], [53].

Because ACO is a computationally expensive technique, only a limited number of attempts have been made to apply it to a compiler optimization problem. Lintzmayer et al. apply ACO to register allocation [54], and Shobaki et al. apply ACO to RP-aware instruction scheduling [11]. In the current paper, we describe how to parallelize the ACO instruction scheduling algorithm of Shobaki et al. on the GPU.

The RP-aware scheduling problem addressed in this work is a multi-objective optimization problem with precedence constraints. These two characteristics make this problem more complex than most of the other problems that have been solved using parallel ACO. As indicated by Falcón-Cardona et al. [55], the parallelization of ACO has not been yet properly exploited in a multi-objective optimization context. Mora et al. [56] present a study on colony-level parallelization schemes for multi-colony, multi-objective ACO algorithms. In our work, we use ant-level rather than colony-level parallelization.

Cano et al. [57] propose a GPU parallelization of the multi-objective grammar-based Ant Programming algorithm for classification introduced by Olmo et al. [58]. However, the problem that they target (classification), is a machine-learning problem not an optimization problem.

Parallel ACO approaches for precedence-constrained problems are also limited. Among the above-mentioned approaches, only Thiruvady et al. [51] tackle a precedence-constrained problem, but they use multi-core CPUs not GPUs. The instruction scheduling problem is similar to the SOP since it has precedence constraints. Although sequential ACO approaches have been proposed for solving the SOP [15], [59], [60], we are not aware of any parallel ACO approach for solving the SOP.

Because the RP-aware scheduling problem is both a multi-objective problem and a precedence-constrained problem, none of the previously proposed parallel ACO approaches directly applies to it. For instance, the work of Cecilia et al., who propose a fine-grained parallel ACO approach for solving the TSP [37], [38] would not work efficiently for a precedence-constrained problem. In the TSP, which does not have precedence constraints, the number of candidate cities is always equal to the number of unvisited cities, which provides a high degree of data parallelism in the next-city-selection step. In the instruction scheduling problem, precedence constraints limit the number of candidate instructions, and thus limit the degree of data parallelism in the next-instruction-selection step. Therefore, applying the algorithm of Cecilia et al. to a precedence-constrained problem would result in a substantial amount of unnecessary computation.

Menezes et al. [43] perform a comparison between a coarse-grained parallel ACO approach (mapping an ant to a thread, as we do in this paper) and a fine-grained parallel ACO approach (mapping an ant to a block or a wavefront, as proposed by Cecilia et al.) for solving the TSP. Their results show that there is no clear best strategy and that the performance depends on the problem size and the number of ants.

IV. Algorithm Description

A. Sequential Algorithm

On a high level, the sequential ACO scheduling algorithm proposed by Shobaki et al. [11] consists of two passes. In the first pass (the RP pass), ILP is ignored and the algorithm searches for a schedule that minimizes RP. In the second pass (the ILP pass), latencies are taken into account, and the algorithm searches for the shortest schedule that maintains the best RP found in the first pass. Thus, in the second pass, the best RP found in the first pass is treated as a constraint.

The ACO algorithm is an iterative algorithm. In each iteration, it simulates a certain number of ants, each of which constructs a candidate schedule. Each schedule is dependent on the pheromone values stored in the pheromone table, the guiding heuristic and a random factor. With this randomization, each ant is likely to produce a different schedule. At the end of each iteration, the best schedule produced by any ant in that iteration (*the iteration winner*) is used to update the pheromone table. The algorithm terminates when the cost of the global best schedule is equal to a pre-computed lower bound (LB) or when a certain number of iterations, called the *termination condition*, have been performed without finding an improvement.

The pheromone table guides the construction of candidate schedules. It is a two-dimensional table with n rows and n columns, where n is the number of instructions. For any two instructions i and j, the value τ_{ij} in the table is the amount of pheromone placed on the link between instructions i and j.

At the end of each iteration, the pheromone table entry of each link (i, j) in the iteration winner is incremented according to a certain formula [11] to increase the chances of constructing similar schedules in the next iterations.

A candidate schedule is constructed by selecting one instruction at a time from the *ready list*. The ready list is a list containing the unscheduled instructions whose dependencies have been satisfied. Selecting the next instruction is done randomly, but with a bias that takes into account the pheromone-table entries and the guiding heuristic. The formula for performing this may be found in the original paper [11].

That selection formula is designed to balance *exploitation* and *exploration*. Exploitation refers to using the pheromone table to bias the search towards selecting schedules that are similar to the best schedules found so far. Exploration refers to using randomization to bias the search towards discovering new schedules. In the second pass, the iteration winner is selected from the schedules that satisfy the RP constraint.

At the end of each iteration, another formula is used to reduce the amount of pheromone in each table entry to simulate the decay of pheromones [11]. This formula involves a parameter, called the *decay factor*, that controls the rate at which pheromones are dissipated. Experimentally, we used a decay factor of 0.8.

Before the ACO search starts, an initial schedule is constructed using a heuristic, such as the Critical-Path (CP) heuristic [1], the Last-Use-Count heuristic [61] or AMD's heuristic. Initially, this schedule is the global best schedule.

B. Parallelization on the GPU

In the above-described ACO algorithm, it is noted that the schedule construction performed by each ant is independent of other ants, and thus ants can construct their schedules in parallel. To exploit this, we parallelize the ACO algorithm by mapping each ant to a GPU thread. Whenever ACO is invoked on a scheduling region, we launch a GPU kernel that performs the schedule construction using multiple ants in parallel and then updates the pheromone table at the end of each iteration.

Our parallel algorithm is implemented in HIP for an AMD GPU. The GPU is both the target processor and the processor that is used to perform scheduling during compilation. An AMD GPU is used because AMD's GPU compiler is an open-source LLVM-based compiler. The parameters of our scheduling kernel are selected to achieve the following:

- Avoiding block-level synchronization by ensuring that all the threads in a block execute in lockstep. This is achieved by setting the number of threads per block to the wavefront size, which is 64 threads on the GPU that we use.
- Distributing the work across multiple CUs to better utilize the CU resources. This is achieved by selecting the number of blocks (which is equal to number of wavefronts in our case) to be greater than the number of CUs. Since the GPU that we use has 60 CUs, we launch 180 blocks.

With the above two settings, each launch of the scheduling kernel has a total of 11,520 threads, which corresponds to 11,520 ants constructing schedules in parallel.

After completing memory allocation and data transfer from the CPU to the GPU, we launch a cooperative GPU kernel. The kernel has a main loop that is repeated until a schedule of cost equal to the LB is found or the termination condition is satisfied. In each iteration, all threads construct their schedules in parallel. A different random seed is input to each thread to maximize the chances of constructing a different schedule from other threads.

After the schedule construction stage is complete, all threads are synchronized. At that point, all threads cooperate to find the best schedule constructed in that iteration using a parallel computational pattern called a *reduction* [62]. Then all threads are synchronized again to work in parallel on updating the pheromone table based on the iteration's best schedule.

After completing the pheromone table update, the first thread compares the iteration's best schedule with the global best schedule found so far. If the iteration's best schedule is better, the first thread updates the global best schedule. If the cost of this schedule is equal to the pre-computed LB, the first thread sets a flag for terminating the whole kernel with an optimal solution. Otherwise, the first thread increments the variable that counts the number of iterations without improvement (the termination condition). At the end of the iteration, all threads are synchronized, and each thread resets its schedule to start constructing a new schedule. After the kernel completes executing, the global best schedule is copied from the GPU to the CPU and memory is freed.

C. Example

The proposed algorithm is illustrated by the example shown in Figure 1 [10]. The Def and Use sets of each instruction are shown in the DDG, where r1, r2, …, r7 are virtual registers. For simplicity, we assume that scheduling is done for a single-issue machine and that the ACO algorithm performs only one iteration in each pass with two ants per iteration.

In the first pass, the algorithm ignores latencies and focuses on searching for a minimum-RP schedule. The schedule found by each ant is shown in Figure 1.b. The details of constructing these schedules are omitted to focus on the high-level ideas. The schedule of Ant 1 has a PRP of 4, because each of Instructions A, B, C, and D opens a new live range. The schedule of Ant 2 has a PRP of 3, because Instruction F in Cycle 3 closes the live ranges of C and D. Therefore, the best PRP at the end of the first pass is 3.

In the second pass, latencies are considered. Stalls are added to the best-RP schedule found in the first pass (the schedule of Ant 2) to satisfy latency constraints. This produces the leftmost schedule in Figure 1.c. This schedule is the initial schedule in the second pass, and its PRP of 3 is set as the target PRP.

Next, each ant constructs a schedule that meets the PRP target, and the shorter schedule is selected as the iteration's best schedule. If at any point in the construction of an ant's schedule, the PRP exceeds 3, that ant is terminated. In this example, both ants manage to find schedules that meet the PRP target of 3. These schedules are shown in Figure 1.c. Since the schedule of Ant 2 is shorter (it uses only 10 cycles, while the schedule of Ant 1 uses 12 cycles), that schedule becomes the iteration's winner. Furthermore, since that schedule is better than the global best schedule at that point, the global best schedule is updated to the schedule of Ant 2.

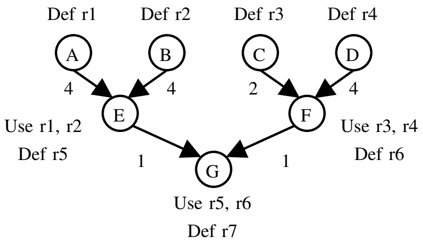
Since in this example, the number of iterations per pass is only one, the algorithm will terminate with the final schedule being the schedule of Ant 2, which is optimal in this case.

We note that in the second pass, different ants may construct schedules of different lengths (different numbers of stalls). Some stalls are necessary to satisfy the latency constraints, while other stalls are optionally added to minimize RP. In the schedule of Ant 2 in Figure 1.c, the stall in Cycle 8 is a necessary stall to satisfy the latency constraint between Instructions C and F. After scheduling Instruction C, the ready list is empty and the only valid option at that point is scheduling a stall in Cycle 8.

In contrast, the stall in Cycle 4 of the same schedule is an *optional stall* that has been added to reduce the chances of violating the RP constraint. After scheduling Instruction D, the ready list contains Instruction C. So there are two options at that point: scheduling Instruction C and increasing PRP to 4, or scheduling a stall to wait until Instruction E becomes ready. Taking the latter option reduced the PRP to 2 at Cycle 6.

It is not always clear whether scheduling an optional stall is beneficial. Scheduling too many optional stalls may result in an excessively long schedule. Our ACO algorithm includes a heuristic for deciding whether scheduling an optional stall is likely to be beneficial. The heuristic takes into account how the PRP will be impacted by the ready instructions and the instructions that will become ready after scheduling optional stalls (*semi-ready instructions*). The heuristic also tracks the number of optional stalls that have been added so far and reduces the probability of scheduling an optional stall when many optional stalls have been inserted already.

(a) DDG



(b) Pass 1

Ant 1	Ant 2
1: A	1: C
2: B	2: D
3: C	3: F
4: D	4: A
5: F	5: B
6: E	6: E
7: G	7: G
PRP = 4	PRP = 3

(c) Pass 2

ILP-pass initial	Ant 1	Ant 2
1: C	1: D	1: A
2: D	2: C	2: B
3: stall	3: stall	3: D
4: stall	4: stall	4: stall
5: stall	5: F	5: stall
6: F	6: A	6: E
7: A	7: B	7: C
8: B	8: stall	8: stall
9: stall	9: stall	9: F
10: stall	10: stall	10: G
11: stall	11: E	PRP = 3
12: E	12: G	
13: G	PRP = 3	
PRP = 3		

Fig. 1. Parallel ACO scheduling example

V. Optimizing the ACO Algorithm on the GPU

This section describes the techniques that we have developed to optimize the performance of our parallel ACO algorithm on the GPU. As mentioned earlier, there is no previous work on parallelizing an ACO algorithm for a multi-objective optimization problem on the GPU. The two main challenges involved in parallelizing an algorithm on the GPU are memory optimizations, especially memory coalescing, and thread divergence. Our approach to handling these challenges is described in the next two subsections.

A. Memory Optimizations

Memory coalescing is a GPU-specific feature, which allows for a better utilization of the GPU memory bandwidth. In memory coalescing, multiple memory accesses are combined into a single transaction. This is possible when multiple threads within the same wavefront access consecutive locations in memory. To maximize memory coalescing, we have changed the memory layout of several data structures in the parallel code, such that threads in a wavefront access consecutive locations in the GPU memory during their execution. More specifically, for the frequently accessed data members of C++ classes, we replaced each data member with an array of members in which each array element corresponds to a thread.

Dynamic memory allocation on the GPU is known to be very slow [63]. Therefore, we avoided it by allocating and initializing objects on the CPU and then copying them to the GPU. In addition, we replaced linked lists with two-dimensional arrays, in which each row corresponds to an element in the list and each column corresponds to a thread.

This has multiple performance advantages. First, only one dynamic allocation of a contiguous memory block is needed, which significantly reduces the allocation time. Second, the resulting access pattern tends to maximize memory coalescing, because when multiple threads traverse their lists in parallel, they are likely to access adjacent memory locations.

When the arrays are allocated on the CPU, they cannot be resized dynamically on the GPU. Therefore, when a list is replaced with an array, the size of that array must be an upper bound (UB) on the expected list size. Clearly, a tighter UB leads to more efficient memory usage and to faster allocation. For example, a trivial but loose UB on the size of the ready list is the total number of instructions.

A tighter UB on the ready-list size is the maximum number of independent instructions in the DDG, because the instructions in the ready list must be independent. We can compute an estimate of the maximum number of independent instructions using the transitive closure of the DDG, which must be computed anyway for other purposes. The transitive closure indicates for every pair of instructions x and y, whether x depends on y, y depends on x or the two instructions are independent. Given the transitive closure of the DDG, a tighter UB on the ready list size is one plus the maximum number of independent instructions that any instruction in the DDG has.

For example, in the DDG of Figure 1.a, the total number of instructions, which is 7, is a loose UB on the ready list size, as the ready list can never have all 7 instructions in it. The transitive closure shows that the maximum number of independent instructions that any instruction in the DDG has is 4. For example, Instruction A is independent of Instructions B, C, D and F. This leads to a tighter UB of 5 on the ready list size.

To minimize memory allocation and transfer time, we batch the allocation and the transfer by consolidating individual variables into large arrays. With this consolidation, a single memory allocation call and a single memory transfer call are needed for each large array instead of making thousands of calls to allocate and transfer individual variables.

B. *Thread-Divergence Optimizations*

Thread divergence is another important GPU-specific factor that affects performance. Thread divergence occurs when threads in the same wavefront follow different execution paths. For example, in executing an *if* statement, some threads may execute the *if* path while others execute the *else* path. Since the threads in a wavefront are executed in lockstep, control paths taken by the threads in a wavefront are traversed one at a time. When one path is executed, threads that are not taking that path stay idle but still occupy resources. Therefore, thread divergence degrades performance and should be minimized.

As described in Section IV-B, the scheduling kernel consists of three main stages: schedule construction, best schedule selection, and the pheromone-table update. As explained below, significant thread divergence is likely to happen only in the schedule-construction stage.

Selecting the best schedule in each iteration is done using parallel reduction, which may cause thread divergence, but this divergence can be reduced by using an efficient reduction kernel [64, Chapter 5.3]. Updating the pheromone table does not cause much divergence, because each thread is assigned a different column to update.

Schedule construction is the stage that is expected to have significant thread divergence. As described in Section IV-A, schedule construction of a single ant is performed by repeatedly selecting the next instruction from the ant's ready list.

An important cause of divergence in schedule construction is that some ants select the next instruction based on exploitation, while others select it based on exploration, and each of these two selection schemes is implemented using a different formula. To minimize the divergence caused by this, we perform the randomized selection between exploration and exploitation at the wavefront level rather than the thread level, that is, all the threads within each wavefront select the next instruction based on the same method. Of course, this does not mean that all the threads within a wavefront will select the same instruction. Experimentally, this technique gave a significant reduction in divergence and consequently in scheduling time in the first pass.

Another important cause of divergence in our ACO scheduling algorithm is that the schedules explored in the second pass generally have different numbers of stalls. For example, the schedule constructed by Ant 1 in Figure 1.c has 5 stalls and a total length of 12, while the schedule constructed by Ant 2 has only 3 stalls with a total length of 10. In Cycle 3, for example, Ant 1 schedules a stall, while Ant 2 schedules Instruction D, and thus each ant performs different operations in that cycle. More specifically, Ant 2, which schedules an actual instruction must update the ready list by traversing the successor list of Instruction D. This is a particularly challenging cause of divergence that is specific to our latency-constrained scheduling problem and is not an issue in other sequencing problems, such as the TSP and the SOP.

The approach that we take to reducing the divergence caused by the difference in schedule lengths among threads is unifying the optional-stall insertion strategy within each wavefront. We allow the insertion of optional stalls only in a fraction of the wavefronts and disallow optional stalls in the rest of the wavefronts. Experimentally, allowing only a quarter of the wavefronts to insert optional stalls gave the best results.

Another optimization that we use to minimize the number of cycles in which different threads perform different kinds of operations is terminating all the threads within a wavefront as soon as one thread completes constructing its schedule. This technique takes advantage of the fact that in our ACO algorithm, only the iteration winner updates the pheromone table. If at least one thread completes constructing a schedule, other threads won't have a chance to produce the iteration winner, because they will necessarily be using more cycles and thus constructing longer schedules.

A third cause of divergence is that each ant has a different ready list with a possibly different size. As a result, the threads that have shorter ready lists to scan must wait for the threads that have longer ready lists to scan. Furthermore, updating the ready list after selecting the next instruction may also cause divergence, as this update involves iterating through the selected instruction's list of successors to check if they have become ready. Since different instructions may have different numbers of successors, this is another cause of divergence.

Differences in ready-list and successor-list sizes make solving a problem with precedence constraints on the GPU harder. In the absence of precedence constraints, ready lists in all threads will have exactly the same size, and no successor lists need to be traversed.

Minimizing the differences in the ready list and the successor list is particularly challenging, because the very purpose of launching many ants in parallel is constructing different schedules. When each ant makes different selections and constructs a different schedule, more sub-spaces will be explored, thus increasing the chances of finding a better final solution. Naturally, these differences will cause the ready lists and the successor lists to be different.

We have experimented with different techniques for minimizing the divergence that results from the differences in ready-list sizes. We tried to unify the ready list sizes among the threads within each wavefront by limiting the ready list size to the minimum list size in any thread in that wavefront or to the mid point between the minimum size and the maximum size, but these attempts did not give better overall results. When the ready list size in some threads is limited, the options to be considered are limited. This won't affect correctness, because all options will eventually be considered by the end of the schedule construction, but it may affect the quality of the constructed schedule, because it may defer some good options that must be considered early. Our experimentation was based on the hope that although some good options may be considered too late in some wavefronts, they will be considered early in other wavefronts. However, the experimental results showed that limiting the ready-list size does not give better overall results.

Finally, in an attempt to minimize the differences in schedule length among the threads within the same wavefront, we experimented with using different guiding heuristics in different groups of wavefronts. As explained earlier, the ACO search is guided by common heuristics. Some heuristics minimize the schedule length more aggressively than others [61]. More aggressive ILP heuristics, such as the Critical-Path heuristic, tend to produce shorter schedules. Using the same heuristic within each wavefront and different heuristics in different wavefronts results in a better exploration of the solution space with smaller differences in behavior within the same wavefront.

VI. Experimental Results

A. Experimental Setup

The sequential ACO algorithm and the proposed parallel ACO algorithm were implemented in LLVM 15 and evaluated using the rocPRIM benchmarks [28]. The tests were run on a system with an AMD Ryzen Threadripper 1950X 16-Core processor running at 3.4 GHz and an AMD Radeon VII GPU running at 1.8 GHz. The ROCm version is 5.4.1, and the operating system is Ubuntu 20.04.5 LTS.

In this evaluation, we use 180 blocks with 64 threads per block, which gives a total of 11,520 threads. Each thread simulates an ant. When the scheduling region is larger, more iterations are needed to find a high-quality solution. Therefore, we divided the scheduling regions into three categories based on their sizes (the number of instructions in the region) and used larger termination conditions for categories with larger sizes. The categories used are [1-49], [50-99], and ≥ 100, and the termination conditions are 1, 2, and 3, respectively.

TABLE 1
Benchmark Statistics

Stat	Value
Number of benchmarks	341
Number of kernels	269
Number of scheduling regions	181,883
Number of scheduling regions processed by ACO in pass 1	1,734
Number of scheduling regions processed by ACO in pass 2	12,192
Avg. processed region size in pass 1	68.3
Avg. processed region size in pass 2	40.2
Max. processed region size in pass 1	1,176
Max. processed region size in pass 2	2,223

The proposed algorithm is evaluated by comparing its performance to that of the AMD production scheduling algorithm [65], which is a greedy algorithm that is built on the LLVM generic scheduling algorithm. It extends the LLVM scheduler to specifically model the AMD GPU, and it is well tuned for ROCm. Therefore, it is a state-of-the-art scheduler for the AMD GPU.

The rocPRIM benchmarks [28] used in our experiments utilize the reusable rocPRIM library kernels to test their performance. The rocPRIM kernels are used in the implementation of many important higher-level libraries, frameworks and applications. So, significantly improving the performance of some rocPRIM benchmarks indicates that our algorithm can significantly improve the performance of a wide range of applications.

Table 1 shows some statistics about the rocPRIM benchmarks. We included only the benchmarks that are sensitive to scheduling. Sensitivity was determined by applying the base LLVM scheduling algorithm, the proposed parallel ACO algorithm and the CP heuristic to each benchmark. If the coefficient of variation for the three execution times was within 3%, the benchmark was considered insensitive to scheduling.

As shown in the table, 341 scheduling-sensitive benchmarks are included. These benchmarks use 269 kernels. Some kernels are invoked by multiple benchmarks, but different benchmarks may invoke the same kernel with different parameters. A total of 181,883 scheduling regions were scheduled. In LLVM, a scheduling region is a basic block or part of a basic block.

Each scheduling region is first scheduled using AMD's heuristic scheduler. The cost of the heuristic schedule is then compared with the lower bound (LB). If the cost is equal to the LB, the heuristic schedule is optimal and ACO is not needed. If the cost of the heuristic schedule is greater than the LB, ACO is invoked to search for a better schedule. The table shows that ACO scheduling was applied to 1,734 regions in the first pass and 12,192 regions in the second pass. The average region size processed by ACO is 68.3 in the first pass and 40.2 in the second pass. The largest region size processed is of size 1,176 in pass 1 and 2,223 in pass 2.

Every test in this evaluation was run five times and the median was taken for each scheduling region (Tables 2, 3.a, 3.b, 4.a, 4.b, 6 and Figures 2, 3) or benchmark (Tables 5, 7 and Figure 4). All the results reported below are based on the medians.

B. Effect of ACO

In this subsection, we show the gain that is achieved by using the ACO scheduling algorithm relative to AMD's scheduling algorithm (base LLVM). The two metrics used in the evaluation are the occupancy and the schedule length. Occupancy is evaluated at the kernel level, while the schedule length is evaluated at the scheduling-region level. The percentage improvements reported here are based on computing the sum of occupancies across all kernels or the sum of scheduling lengths across all regions.

Ideally, if the termination conditions are the same for the sequential and the parallel algorithms, both algorithms should find the same solutions. Practically, however, the solutions will not be exactly the same due to the randomness of the ACO algorithm. On a sufficiently large data set, the differences in the overall occupancy and schedule length between the sequential and the parallel algorithms should be negligible.

The results in Table 2 show that the ACO algorithm improves the overall occupancy by 0.66% and the overall schedule length by 5.52% relative to the AMD algorithm. The maximum percentage improvement on any region is 300% in occupancy and 78.5% in schedule length. Overall, these numbers show that the ACO-based algorithm gives significantly better schedules than AMD's heuristic scheduler. Note that although the aggregate improvements are not high, large improvements on individual hot regions can have a high impact on the execution time as shown in Section VI-E.

TABLE 2
IMPROVEMENT OF ACO RELATIVE TO AMD SCHEDULER

Stat	Value
Regions processed by ACO in pass 1	1,734
Regions processed by ACO in pass 2	12,192
Overall occupancy increase	0.66%
Max. occupancy increase in any kernel	300.00%
Overall schedule length reduction	5.52%
Max. schedule length reduction	78.52%

C. Effect of Parallelization

In this subsection, we show the speedup delivered by the proposed parallel ACO algorithm relative to the sequential ACO algorithm in the first pass (Table 3.a) and in the second pass (Table 3.b). The speedup ratio is the ratio between the scheduling time of the sequential algorithm run on the CPU and the scheduling time of the parallel algorithm run on the GPU. This ratio was computed only for *comparable regions*, which are the regions that both algorithms took the same number of iterations to schedule. Due to the randomness in ACO, each algorithm may take a different number of iterations to solve the same instance. The third row in each table shows the geometric-mean speedup for each size range.

As expected, the parallel ACO algorithm runs significantly faster than the sequential ACO algorithm. The speedup is greater on larger regions. For example, in the first pass, the geometric-mean speedup is 2.07 across the regions in the size range [1-49], while it is 12.48 across the regions of size 100 or greater. This is also expected, because on smaller regions, the benefit from parallelization may be over-weighed by the overhead of launching a GPU kernel and copying data between the CPU and the GPU.

TABLE 3.a
PARALLEL SPEEDUP IN THE FIRST PASS

Inst. count range	1-49	50-99	≥ 100
Regions processed by ACO	728	643	363
Comparable regions	716	600	327
Geometric mean speedup	2.07	7.44	12.48
Max. Speedup	5.69	12.69	27.19
Min. Speedup	0.63	3.30	5.66

TABLE 3.b
PARALLEL SPEEDUP IN THE SECOND PASS

Inst. count range	1-49	50-99	≥ 100
Regions processed by ACO	9,808	1,574	810
Comparable regions	9,613	1,354	436
Geometric mean speedup	1.99	4.80	7.55
Max. Speedup	8.25	13.03	17.37
Min. Speedup	0.45	1.08	4.10

The last two rows in each table show the maximum and the minimum speedup on any region in each size range. The

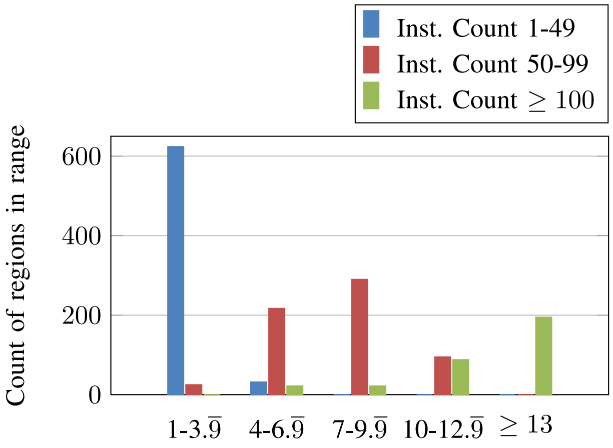


Fig. 2. Speedup distribution in the first pass

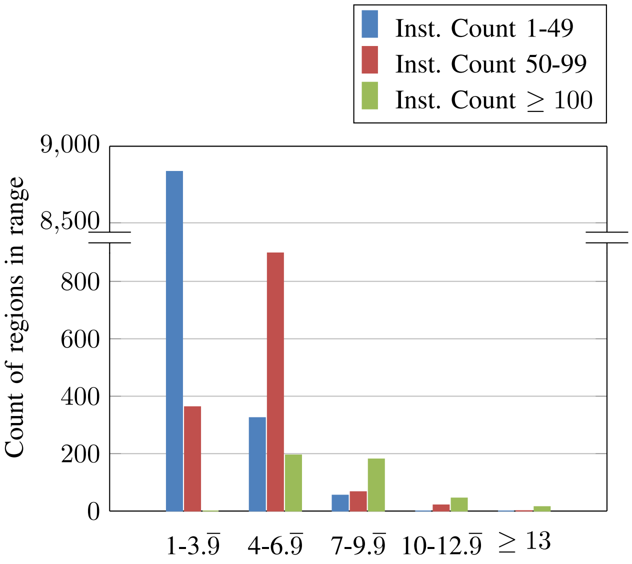


Fig. 3. Speedup distribution in the second pass

maximum speedup on any region is 27.19. The minimum speedup shows that for regions of size 50 or greater, every region benefits from parallelization. The regions that do not benefit from parallelization are the easier regions for which the sequential algorithm finds quality solutions very quickly.

It is noted that the speedup ratios in the second pass are significantly lower than the ratios in the first pass. This is attributed to the fact that there is more thread divergence in the second pass, due to the latency constraints that cause different ants to construct schedules of different lengths as shown in Figure 1.c. The techniques described in Section V-B reduce the effect of thread divergence, but they don't eliminate it.

Figures 2 and 3 show the distribution of the speedup ratios for each region size range.

Next, we study the effect of the optimizations described in Section V. Table 4.a shows the effect of the memory optimizations described in Section V-A, and Table 4.b shows the effect of the thread-divergence optimizations described in Section V-B. The entries in these tables are the percentage improvements in the parallel ACO scheduling time. Both the overall improvements across all scheduling regions and the maximum improvement on any region are shown for each pass.

The improvement from memory optimizations is very high in both passes. The improvements from thread-divergence optimizations are not as high as the improvements from memory optimizations but are still quite significant, especially on larger regions in the second pass. For the regions of size 100 instructions or greater, the overall improvement from thread-divergence optimizations is 15.42%, and the maximum improvement on any region is 101.40%. As explained in Section V-B, there is more thread divergence in the second pass. The improvement on larger regions is greater than the improvement on smaller regions, because as the region size increases, a larger percentage of the ACO scheduling time is actual computation time on the GPU, as opposed to copying time.

TABLE 4.a
IMPROVEMENTS IN ACO TIME FROM MEMORY OPTIMIZATIONS

Inst. count range	1-49	50-99	$\geq$ 100
Pass 1 overall improvement	645%	1055%	897%
Pass 1 max. improvement	1163%	1592%	1929%
Pass 2 overall improvement	593%	994%	709%
Pass 2 max. improvement	2647%	1629%	3052%

TABLE 4.b
IMPROVEMENTS IN ACO TIME FROM DIVERGENCE OPTIMIZATIONS

Inst. count range	1-49	50-99	$\geq$ 100
Pass 1 overall improvement	0.68%	3.81%	7.00%
Pass 1 max. improvement	17.14%	15.84%	65.96%
Pass 2 overall improvement	3.78%	12.06%	15.42%
Pass 2 max. improvement	55.56%	71.53%	101.40%

D. Compile Times

Table 5 shows the total compile times for the rocPRIM benchmarks using the base AMD scheduler, the sequential ACO scheduler, and the parallel ACO scheduler. In practice, ACO is a computationally expensive algorithmic technique that should be applied selectively only when a significant benefit is expected.

TABLE 5
TOTAL COMPILE TIMES

Scheduler	Total Compile Time (seconds)
Base AMD	840
Sequential ACO	1225 (45.8%)
Parallel ACO	967 (15.1%)

As described earlier, a heuristic is applied first in each pass, and the ACO scheduler is invoked only if the heuristic RP (in the first pass) or schedule length (in the second pass) is not at the LB. In addition to this natural filtering, we found experimentally that the benefit from ACO in the second pass is likely to be significant only when the gap between the heuristic schedule length and the LB is greater than a certain threshold. A small improvement in schedule length is unlikely to translate into an improvement in execution time. As detailed in Section VI-F, a threshold of 21 cycles gave the best execution-time results. This filter limits the compile-time cost and minimizes execution-time regressions that may result from negative side effects on un-modeled factors.

Additionally, we found that in some cases, the ACO scheduler may find a schedule with a better occupancy in the first pass, but due to the stronger occupancy constraint in the second pass, it finds a significantly long schedule. In such cases, which don't occur very often, reverting to the heuristic schedule gives better execution-time results. To handle such cases, we added a post-scheduling filter that compares the occupancy and the schedule length of the final ACO schedule with the heuristic occupancy and schedule length and selects the schedule that achieves a better balance between occupancy and ILP. This filter is parameterized and can be tuned to achieve the best performance. Experimentally, we found that if the ACO algorithm increases occupancy by 3 but degrades the schedule length by more than 63 cycles, reverting to the heuristic schedule gives the best execution-time results.

Table 5 shows that the total compile time of the rocPRIM benchmarks using the base LLVM compiler with the AMD scheduler is 840 seconds. When the sequential ACO scheduler is used, the compile time increases by 45.8%. With the parallel ACO scheduler, the increase in compile time relative to the base LLVM compiler is 15.1%. So, scheduling on the GPU using the parallel ACO algorithm reduces the total compile time by 21% relative to scheduling on the CPU using the sequential ACO algorithm. This result shows that parallelization on the GPU makes using a computationally expensive scheduling algorithm much more practical. In future work, we plan to work on further reducing the compile time by exploring additional techniques, such as scheduling multiple regions in parallel.

E. Execution-Time Results

To evaluate the execution-time performance, the rocPRIM benchmarks were compiled with the proposed algorithm used for scheduling (the modified build). The filtering described in Section VI-D was applied. Each benchmark was run five times using the modified build and five times using the base LLVM compiler with the AMD scheduler, and the median throughput was taken for each benchmark.

Figure 4 shows the execution-time results for the benchmarks on which a *significant* difference was measured between the median throughput of the proposed algorithm and that of the AMD algorithm. A difference is significant if it is 1% or greater. All significant differences were improvements. The maximum regression was 0.7%. Regressions are caused by negative side effects on un-modeled factors. An instruction scheduler models register pressure and schedule length, but it does not model other factors that affect performance, such as caching. Since some factors are impossible to model accurately at compile time, regressions are unavoidable but can be minimized with a reasonable cost-benefit analysis.

The x-axis represents the benchmarks and the y-axis represents the percentage improvement relative to the AMD scheduler. The maximum improvement is 74%. The last bar shows that the geometric-mean improvement is 13.2%. The graph also shows that 20 benchmarks were improved by 5% or greater and 11 benchmarks were improved by 10% or greater.

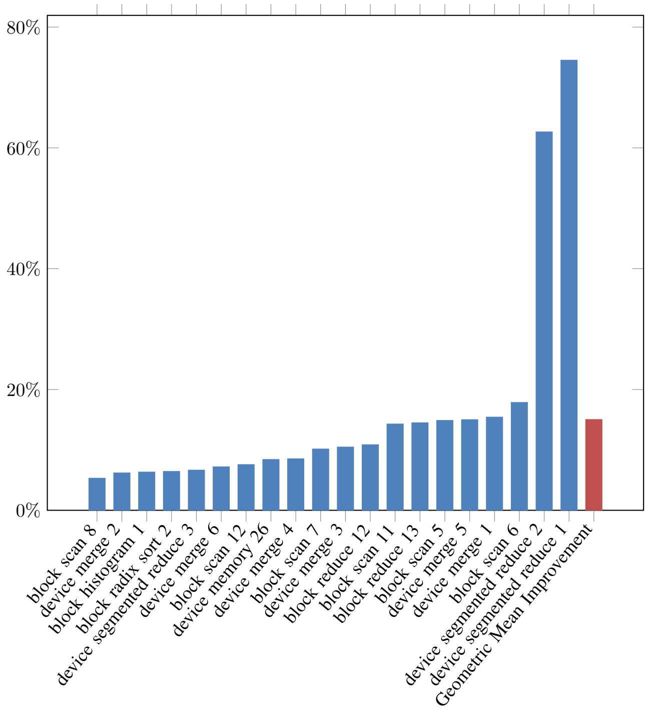


Fig. 4. Execution-time speedup of rocPRIM benchmarks

F. Experimenting with Design Parameters

The parameters used to generate the results reported in the previous subsections were chosen based on experimentation. In this subsection, we show the results of our experimentation with two design parameters, namely the percentage of wavefronts inserting optional stalls (Table 6) and the cycle threshold used to filter out unpromising scheduling regions (Table 7).

Table 6 shows the percentage increase in ACO scheduling time when the percentage of wavefronts inserting optional stalls is increased for the regions of size 100 or greater. The baseline is zero wavefronts inserting optional stalls. The table also shows the improvement in schedule length that is achieved with each percentage of wavefronts relative to 0%. When the percentage of wavefronts inserting optional stalls is increased, a better schedule length may be achieved, because considering optional stalls increases the chance of meeting the target occupancy. If the target occupancy is not met, the scheduler falls back to the second pass's input schedule, which can be quite long.

The table shows that the improvement in schedule length comes at the cost of increasing the scheduling time. We chose 25% because it gives the best balance between the scheduling time and the quality of the schedule. Note that when zero wavefronts consider inserting optional stalls, it will be impossible to find an optimal schedule for some regions.

Table 7 shows the execution-time statistics using different settings of the cycle threshold described in Section VI-D. Recall that if the difference between the length of the input schedule and the LB is less than the cycle threshold, the region is filtered

TABLE 6
EXPERIMENTATION WITH OPTIONAL STALLS

% Blocks inserting optional stalls	25%	50%	75%
% Increase in ACO Time	8.65%	12.30%	20.28%
% Improvement in schedule length	0.27%	0.30%	0.95%
Max. % improvement in schedule length	15.75%	15.75%	23.58%

out. The results in the table show that increasing the cycle threshold eliminates significant regressions, and that a cycle threshold of 21 gives the best results.

TABLE 7
EXPERIMENTATION WITH CYCLE-BASED FILTER

Cycles	5	10	15	20	21	25
Imps. $\geq$ 3%	18	20	20	21	20	20
Imps. $\geq$ 5%	17	20	20	24	24	24
Imps. $\geq$ 10%	9	10	11	9	11	11
Regs. $\geq$ 3%	4	3	1	1	0	0
Regs. $\geq$ 5%	4	3	1	1	0	0
Regs. $\geq$ 10%	3	3	1	1	0	0
Max. Reg.	14.5%	14.5%	10.5%	10.5%	0.7%	1.3%

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we show that the GPU can be used to parallelize a compute-intensive compiler optimization, which is ACO-based instruction scheduling. Efficient parallelization is achieved using a number of techniques for improving memory performance and reducing thread divergence. Our work is the first successful attempt to parallelize a compiler optimization on the GPU and achieve significant improvements in execution speed with a reasonable increase in compile time.

Our experimental evaluation shows that ACO-based scheduling can be performed up to 27 times faster on the GPU, and this leads to reducing the total compile time of rocPRIM by 21% relative to sequential ACO scheduling on the CPU.

In future work, we will continue to work on optimizing the performance of the proposed algorithm, and we will work on maximizing the utilization of the GPU by scheduling multiple regions in parallel.

ACKNOWLEDGEMENTS

This work was supported in part by the US National Science Foundation (NSF) through Award 1911235. The first and second authors were supported by a Progress-to-Promotion award from the Provost office at California State University, Sacramento. The authors thank the GPU-compute compiler team at Advanced Micro Devices (AMD) for donating the machine that was used to generate the experimental results. We also thank Jeffrey Byrnes for the help he provided on setting up the benchmarks and thank Patrick Brannan and Lynn Koropp for the technical support that they provided. Finally, we thank the anonymous reviewers for their constructive comments and suggestions that led to improving the final paper.

APPENDIX

A. Abstract

The artifact contains the source code for an instruction scheduling compiler pass built in the AMDGPU backend of LLVM. It also contains a Docker image which has LLVM with the instruction scheduling pass built on Ubuntu 20.04. Users can replicate the compile-time and execution-time experiments described in the paper with the Docker image. We tried to make running the experiments as convenient as possible. New benchmarks can be added and the settings can be adjusted if the user wishes to perform further experiments.

B. Artifact Checklist (Meta-Information)

- **Algorithm:** Parallel Ant Colony Optimization
- **Run-time environment:** ROCm kernel-mode drivers and Docker.
- **Hardware:** An AMD GPU supported by the ROCm software stack.
- **Output:** Spreadsheets of run-time benchmark throughputs and compile-time metrics.
- **How much disk space required (approximately)?:** 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 3 hours.

C. Description

1) How Delivered: The artifact, containing the source code, is available on GitHub at
https://github.com/CSUS-LLVM/OptSched/tree/CGO24.
However, it is recommended to download the ”docker” directory separately from
https://github.com/CSUS-LLVM/OptSched/tree/CGO24/docker.
The Docker image will download and build the compiler pass from the source automatically.

2) Hardware Dependencies: The results presented in this paper were produced using an AMD Radeon VII GPU, which is a Vega 20 GPU. Since instruction scheduling is a hardware-dependent compiler optimization, some experimental results may vary significantly when the experiments are run on different hardware.

D. Installation

Building the included Docker image will collect all the required user-space dependencies, compile LLVM with the Parallel ACO compiler pass, and download the rocPRIM benchmarks used in the paper.

To build the Docker image, download the ”docker” directory from the GitHub repository and use the ”docker build” command. The build command must be supplied with an argument indicating which AMD GPU targets to build for. Assuming the ”docker” directory is in the current working directory, and the user wants to build for the ”gfx906” architecture, the build command would be

```
docker build --build-arg="amdgpu_arch='gfx906'" docker
```

E. Experiment Workflow

A user wishing to conduct new experiments may wish to adjust the configuration files located in `~/.optsched-cfg`. In the included Docker image, all the default settings match those used to generate the results in Tables 1-3.b and Figures 2-4.

A user seeking to replicate the results presented in the paper will also need to gather baseline results using AMD’s stock compiler. Set ”ACO_BEFORE_ENUM” in sched.ini to ”NO” for these baseline experiments. With this setting, the Parallel ACO scheduler will not do any scheduling, but the compile-time metrics will still be reported in a format that the provided scripts can parse.

1) Compile-Time Experiments: Compile-time experiments are performed by building the rocPRIM library with the parallel ACO instruction scheduling pass and collecting the printed output, which contains data on various compile-time metrics.

The rocPRIM source is already downloaded in the included Docker image. It can be found at
`~/aco/benchmarks/rocPRIM`.
To build rocPRIM with parallel ACO, make a new directory and run the following commands from that directory

```
CXX=hipcc cmake -DBUILD_BENCHMARK=ON
-DAMDGPU_TARGETS="$AMDGPU_ARCH"
~/aco/benchmarks/rocPRIM/
make |& tee build.log
```

This will build rocPRIM and place the parallel ACO logs in build.log. The following command will convert the output compile-time metrics into an Excel spreadsheet named ”out.xlsx”.

```
extractOptSchedData.py build.log out
```

2) Execution-Time Experiments: Execution-time experiments are performed by running the rocPRIM benchmarks produced from building rocPRIM with the parallel ACO scheduler and measuring their throughput. A script has been provided to run them automatically. The following command will replicate the execution-time experiment that produced the results in Figure 4:

```
runPrimTests.py --benchpath
</path/to/benchmarks> --testpath
~/aco/util/testSets/primSensitiveTests.txt
--repetitions 5 --output results.csv
```

Each benchmark listed in ”primSensitiveTests.txt” will be run 5 times, and the median throughput in GB/s for each benchmark will be reported in results.csv.

F. Evaluation and Expected Result

After collecting the compile-time and execution-time results for both the baseline AMD and the parallel ACO builds, the results spreadsheets can be used to compare their performance.

The parallel-ACO throughput is compared with the base-AMD throughput for each benchmark. The performance improvements of parallel ACO relative to base AMD should be as shown in Figure 4.

References

[1] K. Cooper and L. Torczon, *Engineering a compiler*, 2nd ed. Morgan Kaufmann, 2011.

[2] C. Kessler, “Scheduling expression DAGs for minimal register need,” *Computer Languages*, vol. 24, no. 1, pp. 33–53, 1998.

[3] A. Malik, “Constraint programming techniques for optimal instruction scheduling,” Ph.D. dissertation, University of Waterloo, 2008.

[4] G. Barany and A. Krall, “Optimal and heuristic global code motion for minimal spilling,” in *International Conference on Compiler Construction*, 2013, pp. 21–40.

[5] L. Domagała, D. van Amstel, F. Rastello, and P. Sadayappan, “Register allocation and promotion through combined instruction scheduling and loop unrolling,” in *International Conference on Compiler Construction*, 2016, p. 143–151.

[6] R. Lozano, “Constraint-based register allocation and instruction scheduling,” Ph.D. dissertation, KTH Royal Institute of Technology, 2018.

[7] R. C. Lozano, M. Carlsson, G. Blindell, and C. Schulte, “Combinatorial register allocation and instruction scheduling,” *ACM Trans. on Programming Languages and Systems*, vol. 41, no. 3, 2019.

[8] G. Shobaki, M. Shawabkeh, and N. Rmaileh, “Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach,” *ACM Trans. Archit. Code Optim.*, vol. 10, no. 3, 2013.

[9] G. Shobaki, A. Kerbow, C. Pulido, and W. Dobson, “Exploring an alternative cost function for combinatorial register-pressure-aware instruction scheduling,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, 2019.

[10] G. Shobaki, A. Kerbow, and S. Mekhanoshin, “Optimizing occupancy and ILP on the GPU using a combinatorial approach,” in *ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, p. 133–144.

[11] G. Shobaki, V. Gordon, P. McHugh, T. Dubois, and A. Kerbow, “Register-pressure-aware instruction scheduling using Ant-Colony Optimization,” *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, 2022.

[12] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.

[13] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, “Register optimizations for stencils on GPUs,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, p. 168–182.

[14] M. Dorigo, V. Maniezzo, and A. Colorni, “Ant System: Optimization by a colony of cooperating agents,” *IEEE Trans. Syst., Man, and Cybern. - Part B*, vol. 26, no. 1, pp. 29–41, 1996.

[15] L. Gambardella and M. Dorigo, “An Ant Colony System hybridized with a new local search for the Sequential Ordering Problem,” *INFORMS J. Comput.*, vol. 12, no. 3, pp. 237–255, 2000.

[16] L. Escudero, “An inexact algorithm for the Sequential Ordering Problem,” *European Journal of Operational Research*, vol. 37, no. 2, pp. 236–249, 1988.

[17] J. Ning, C. Zhang, P. Sun, and Y. Feng, “Comparative study of Ant Colony Algorithms for multi-objective optimization,” *Information*, vol. 10, no. 1, 2019.

[18] L. Gambardella, E. Taillard, and G. Agazzi, “MACS-VRPTW: A multiple Ant Colony System for vehicle routing problems with time windows,” in *New Ideas in Optimization*, 1999, p. 63–76.

[19] D. Skillicorn and D. Barnard, “Compiling in parallel,” *Journal of Parallel and Distributed Computing*, vol. 17, no. 4, pp. 337–352, 1993.

[20] M. Mendez-Lojo, M. Burtscher, and K. Pingali, “A GPU implementation of inclusion-based points-to analysis,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, p. 107–116.

[21] Y. Su, D. Ye, and J. Xue, “Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems,” in *20th Annual International Conference on High Performance Computing*, 2013, pp. 149–158.

[22] R. Nasre, “Time- and space-efficient flow-sensitive points-to analysis,” *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, 2013.

[23] T. Blaß and M. Philippsen, “GPU-accelerated fixpoint algorithms for faster compiler analyses,” in *Proceedings of the 28th International Conference on Compiler Construction*, 2019, p. 122–134.

[24] T. Prabhu, S. Ramalingam, M. Might, and M. Hall, “EigenCFA: Accelerating flow analysis with GPUs,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011, p. 511–522.

[25] Z. Zuo, K. Wang, A. Hussain, A. Sani, Y. Zhang, S. Lu, W. Dou, L. Wang, X. Li, C. Wang, and G. Xu, “Systemizing interprocedural static analysis of large-scale systems code with Graspan,” *ACM Trans. Comput. Syst.*, vol. 38, no. 1–2, 2021.

[26] Z. Zheng, X. Shi, L. He, H. Jin, S. Wei, H. Dai, and X. Peng, “Feluca: A two-stage graph coloring algorithm with color-centric paradigm on GPU,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 160–173, 2021.

[27] M. Osama, M. Truong, C. Yang, A. Buluç, and J. Owens, “Graph coloring on the GPU,” in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 231–240.

[28] AMD, “rocPRIM documentation,” . [Online]. Available: https://rocmdocs.amd.com/projects/rocPRIM/en/latest/index.html

[29] NVIDIA, “GPU-accelerated applications,” . [Online]. Available: https://www.nvidia.com/en-us/gpu-accelerated-applications

[30] AMD, “HIP programming guide v5.4,” . [Online]. Available: https://docs.amd.com/bundle/HIP-Programming-Guide-v5.4/page/Introduction{_}to{_}HIP{_}Programming{_}Guide.html

[31] M. Pedemonte, S. Nesmachnow, and H. Cancela, “A survey on parallel Ant Colony Optimization,” *Applied Soft Computing*, vol. 11, no. 8, pp. 5181–5197, 2011.

[32] M. Randall and A. Lewis, “A parallel implementation of Ant Colony Optimization,” *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1421–1432, 2002.

[33] S. Janson, D. Merkle, and M. Middendorf, “Parallel Ant Colony algorithms,” in *Parallel Metaheuristics*, 2019, pp. 171–201.

[34] R. Skinderowicz, “The GPU-based parallel Ant Colony System,” *J. Parallel Distributed Comput.*, vol. 98, pp. 48–60, 2016.

[35] J. Peake, M. Amos, P. Yiapanis, and H. Lloyd, “Vectorized candidate set selection for parallel Ant Colony Optimization,” in *Genetic and Evolutionary Computation Conference (GECCO)*, 2018, pp. 1300–1306.

[36] J. Cecilia and J. García, “Re-engineering the Ant Colony Optimization for CMP architectures,” *J. Supercomput.*, vol. 76, p. 4581–4602, 2020.

[37] J. Cecilia, J. García, A. Nisbet, M. Amos, and M. Ujaldón, “Enhancing data parallelism for Ant Colony Optimization on GPUs,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 42–51, 2013.

[38] J. Cecilia, A. Llanes, J. Abellán, J. Gómez-Luna, L. Chang, and W. Hwu, “High-throughput Ant Colony Optimization on Graphics Processing Units,” *Journal of Parallel and Distributed Computing*, vol. 113, pp. 261–274, 2018.

[39] G. Guerrero, J. Cecilia, A. Llanes, J. García, M. Amos, and M. Ujaldón, “Comparative evaluation of platforms for parallel Ant Colony Optimization,” *J. Supercomput.*, vol. 69, p. 318–329, 2014.

[40] L. Dawson and I. Stewart, “Improving Ant Colony Optimization performance on the GPU using CUDA,” in *IEEE Congress on Evolutionary Computation*, 2013, pp. 1901–1908.

[41] A. Uchida, Y. Ito, and K. Nakano, “An efficient GPU implementation of Ant Colony Optimization for the Traveling Salesman Problem,” in *International Conference on Networking and Computing (ICNC)*, 2012, pp. 94–102.

[42] R. Skinderowicz, “Implementing a GPU-based parallel MAX–MIN Ant System,” *Future Generation Computer Systems*, vol. 106, pp. 277–295, 2020.

[43] B. Menezes, H. Kuchen, H. A. Neto, and F. de Lima Neto, “Parallelization strategies for GPU-based Ant Colony Optimization solving the Traveling Salesman Problem,” in *IEEE Congress on Evolutionary Computation (CEC)*, 2019, pp. 3094–3101.

[44] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, “Parallel Ant Colony Optimization on Graphics Processing Units,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 52–61, 2013.

[45] Y. Zhou, F. He, and Y. Qiu, “Dynamic strategy based parallel ant colony optimization on GPUs for TSPs.” *Science China Information Sciences*, vol. 60, no. 068102, 2017.

[46] H. Zhi-bin, F. Guang-Tao, F. Tian-Hao, D. Dan-Yang, B. Peng, and X. Chen, “High performance ant colony system based on GPU warp specialization with a static–dynamic balanced candidate set strategy,” *Future Generation Computer Systems*, vol. 125, pp. 136–150, 2021.

[47] J. Fu, L. Lei, and G. Zhou, “A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection,” in *Third International Workshop on Advanced Computational Intelligence*, 2010, pp. 260–264.

[48] D. E. Baz, M. Hifi, L. Wu, and X. Shi, “A parallel Ant Colony Optimization for the Maximum-weight Clique Problem,” in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 796–800.

[49] S. Tsutsui and N. Fujimoto, “A comparative study of synchronization of parallel ACO on multi-core processor,” in *Companion Publication of Genetic and Evolutionary Computation Conference (GECCO Companion)*, 2015, pp. 777–778.

[50] A. Borisenko and S. Gorlatch, “Optimizing a GPU-parallelized Ant Colony Metaheuristic by parameter tuning,” in *Parallel Computing Technologies (PaCT)*, ser. Lecture Notes in Computer Science, 2019, vol. 11657, pp. 151–165.

[51] D. Thiruvady, A. Ernst, and G. Singh, “Parallel Ant Colony Optimization for resource constrained job scheduling,” *Annals of Operations Research*, vol. 242, no. 2, pp. 355–372, 2016.

[52] I. Dzalbs and T. Kalganova, “Accelerating supply chains with Ant Colony Optimization across a range of hardware solutions,” *Comput. Ind. Eng.*, vol. 147, 2020.

[53] S. Mane, P. Lokare, and H. Gaikwad, “Overview and applications of GPGPU based parallel ant colony optimization,” in *International Conference on Advances in Computing and Information Technology (ICACIT)*, 2014.

[54] C. Lintzmayer, M. Mulati, and A. da Silva, “Register allocation with graph coloring by Ant Colony Optimization,” in *International Conference of the Chilean Computer Science Society*, 2011, pp. 247–255.

[55] J. Falcón-Cardona, G. Leguizamón, C. Coello, and M. Tapia, “Multi-objective Ant Colony Optimization: An updated review of approaches and applications,” in *Intelligent Systems Reference Library (ISRL) book series*, 2022, vol. 218.

[56] A. Mora, P. García-Sánchez, J. Merelo, and P. Castillo, “Pareto-based multi-colony multi-objective Ant Colony Optimization algorithms: an island model proposal,” in *Soft Comput*, 2013, vol. 17, p. 1175–1207.

[57] A. Cano, J. Olmo, and S. Ventura, “Parallel multi-objective Ant Programming for classification using GPUs,” *Parallel and Distributed Computing*, vol. 73, no. 6, pp. 713–728, 2013.

[58] J. Olmo, J. Romero, and S. Ventura, “Classification rule mining using Ant Programming guided by grammar with multiple Pareto fronts,” in *Soft Comput*, 2012, vol. 16, pp. 2143–2163.

[59] L. Gambardella, R. Montemanni, and D. Weyland, “An enhanced Ant Colony System for the Sequential Ordering Problem,” in *Operations Research Proceedings 2011*, 2012, pp. 355–360.

[60] R. Skinderowicz, “An improved Ant Colony System for the Sequential Ordering Problem,” *Computers & Operations Research*, vol. 86, pp. 1–17, 2017.

[61] G. Shobaki, L. Sakka, N. A. Rmaileh, and H. Al-Hamash, “Experimental evaluation of various register-pressure-reduction heuristics,” *Softw. Pract. Exper.*, vol. 45, no. 11, p. 1497–1517, 2015.

[62] M. Harris, “Optimizing parallel reduction in CUDA,” . [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[63] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, “ScatterAlloc: Massively parallel dynamic memory allocation for the GPU,” in *Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[64] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed. Morgan Kaufmann, 2017.

[65] AMD, “Amd. 2019. gcn max occupancy scheduler,” . [Online]. Available: http://llvm.org/doxygen/classllvm_1_1GCNMaxOccupancySchedStrategy.html