Optimizing Occupancy and ILP on the GPU using a Combinatorial Approach

Ghassan Shobaki
California State University
Sacramento, California, USA
ghassan.shobaki@csus.edu

Austin Kerbow
Advanced Micro Devices
Santa Clara, California, USA
Austin.Kerbow@amd.com

Stanislav Mekhanoshin
Advanced Micro Devices
Santa Clara, California, USA
Stanislav.Mekhanoshin@amd.com

Abstract

This paper presents the first general solution to the problem of optimizing both occupancy and Instruction-Level Parallelism (ILP) when compiling for a Graphics Processing Unit (GPU). Exploiting ILP (minimizing schedule length) requires using more registers, but using more registers decreases occupancy (the number of thread groups that can be run in parallel). The problem of balancing these two conflicting objectives to achieve the best overall performance is a challenging open problem in code optimization. In this paper, we present a two-pass Branch-and-Bound (B&B) algorithm for solving this problem by treating occupancy as a primary objective and ILP as a secondary objective. In the first pass, the algorithm searches for a maximum-occupancy schedule, while in the second pass it iteratively searches for the shortest schedule that gives the maximum occupancy found in the first pass. The proposed scheduling algorithm was implemented in the LLVM compiler and applied to an AMD GPU. The algorithm's performance was evaluated using benchmarks from the PlaidML machine learning framework relative to LLVM's scheduling algorithm, AMD's production scheduling algorithm and an existing B&B scheduling algorithm that uses a different approach. The results show that the proposed B&B scheduling algorithm speeds up almost every benchmark by up to 35% relative to LLVM's scheduler, up to 31% relative to AMD's scheduler and up to 18% relative to the existing B&B scheduler. The geometric-mean improvements are 16.3% relative to LLVM's scheduler, 5.5% relative to AMD's production scheduler and 6.2% relative to the existing B&B scheduler. If more compile time can be tolerated, a geometric-mean improvement of 6.3% relative to AMD's scheduler can be achieved.

CGO '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7047-9/20/02...$15.00
https://doi.org/10.1145/3368826.3377918

CCS Concepts • **Software and its engineering → Compilers**.

Keywords Compiler Optimizations, Instruction Scheduling, Register Pressure Reduction, Instruction-Level Parallelism (ILP), Graphics Processing Unit (GPU)

ACM Reference Format:
Ghassan Shobaki, Austin Kerbow, and Stanislav Mekhanoshin. 2020. Optimizing Occupancy and ILP on the GPU using a Combinatorial Approach. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368826.3377918

1 Introduction

The performance of a kernel running on the Graphics Processing Unit (GPU) is highly dependent on *occupancy*, which is the number of thread groups (*wavefronts*) that can be run in parallel. Occupancy depends on the amount of resources used by a thread group. Since registers are scarce resources, register usage in each thread is an important factor that determines occupancy. If a single thread uses fewer registers, the hardware scheduler can run more threads in parallel. The number of registers used by a thread is determined by the compiler's register allocation algorithm. Register allocation is highly dependent on the instruction order produced by the pre-allocation instruction scheduler. The instruction order determines *register pressure* (RP), which is the number of virtual registers that are simultaneously live and must be assigned to different physical registers. Therefore, pre-allocation instruction scheduling must minimize RP to maximize occupancy.

However, minimizing RP conflicts with another objective of instruction scheduling, which is minimizing the schedule length or exploiting Instruction-Level Parallelism (ILP). Exploiting ILP tends to increase RP, because scheduling more independent instructions in parallel requires more registers to hold the results of these parallel instructions. The instruction scheduling algorithm must then balance the two conflicting objectives of exploiting ILP and minimizing RP.

On the GPU, minimizing RP to maximize occupancy is critically important, because occupancy has a high impact on GPU performance [9]. As detailed in Section 2, reducing RP by one register can in some cases double the occupancy, and that can potentially double the program's speed if it does not have any negative side effects.

On the other hand, compiler scheduling for ILP is also important on the GPU, because although the hardware can exploit thread-level parallelism (TLP), it does not reorder instructions within a single thread. Therefore, compiler scheduling for ILP can still give a significant performance gain.

In this paper, we present a combinatorial scheduling algorithm that optimizes both occupancy and ILP (schedule length) on the GPU by first maximizing occupancy and then searching for the shortest schedule that gives that maximum occupancy. The algorithm proposed in this paper uses a Branch-and-Bound (B&B) enumeration technique similar to that proposed in our previous work [26, 28]. However, unlike the previous algorithm, which minimizes a weighted sum of RP and schedule length in a single pass, the proposed algorithm is a two-pass algorithm that optimizes RP as a primary objective in the first pass and then optimizes ILP as a secondary objective in the second pass.

As explained in the paper, the proposed algorithm has great advantages relative to the previous algorithm in the GPU environment, including:

1. It separates the two complex problems of minimizing RP and minimizing schedule length, thus making it possible to solve one problem at a time more efficiently.
2. It guarantees that a certain amount of time is spent optimizing occupancy, which is the primary objective.
3. As detailed in the paper, it makes it possible to use kernel-level occupancy information collected in the occupancy pass to relax RP constraints in the ILP pass.

To ensure reasonable compile time when a combinatorial scheduling algorithm is used, a limit must be set on the scheduling time. In the proposed two-pass algorithm, each pass has its own time limit to ensure that enough time is spent optimizing each objective.

The proposed algorithm was implemented in the LLVM compiler and its performance was evaluated experimentally using 13 benchmarks from the PlaidML machine learning framework. The evaluation was done relative to LLVM's generic scheduling algorithm [17], AMD's well-tuned production scheduling algorithm [2] and the previous B&B algorithm [26, 28]. The results show that the proposed algorithm speeds up every PlaidML benchmark (except for one negligibly small regression) by up to 35% relative to LLVM's scheduler, 31% relative to AMD's scheduler and 18% relative to the previous B&B scheduler. The geometric-mean gains are 16.3% relative to LLVM's scheduler, 5.5% relative to AMD's scheduler and 6.2% relative to the previous B&B scheduler. These performance gains are achieved with limited increase in compile time. If more compile time can be tolerated, a geometric-mean gain of 6.3% relative to AMD's scheduler can be achieved.

To the best of our knowledge, the proposed algorithm is the first combinatorial scheduling algorithm that gives substantial performance gains relative to a state-of-the-art scheduling algorithm with a practically acceptable increase in compile time. Furthermore, the proposed algorithm is the first general algorithm for the register-pressure-aware scheduling problem on the GPU.

The rest of this paper is organized as follows. Section 2 defines the problem. Section 3 summarizes previous work. Section 4 discusses the background and the motivation. Section 5 describes the proposed algorithm. Section 6 presents the experimental results, and Section 7 summarizes the conclusions and outlines future work.

2 Problem Definition

The problem addressed in this paper is pre-allocation instruction scheduling with the primary objective of maximizing occupancy and the secondary objective of minimizing schedule length. Thus, the objective is finding the shortest schedule among all maximum-occupancy schedules.

Occupancy is the number of *wavefronts* that are run in parallel. A wavefront is a group of threads that must be executed in lockstep. Occupancy depends on multiple factors, including the number of registers used in each thread, which is highly dependent on instruction scheduling. Register usage is one of the most important factors that determine occupancy, because registers are scarce resources, and thus more threads can be run in parallel if each thread uses fewer registers. In this paper, register usage is used to model occupancy in the instruction scheduling pass.

Register usage in each thread for a given register type is modeled in instruction scheduling by the *peak register pressure* (PRP) for that register type. PRP is the maximum register pressure at any point in the scheduling region. A scheduling region is assumed to be a straight-line piece of code (a basic block or part of a basic block), and this assumption is true for the scheduling regions in LLVM.

It should be noted that the PRP calculated in instruction scheduling is not necessarily equal to the actual number of registers used by the register allocator. Since register allocation is an NP-hard problem [5], the register allocation algorithm may produce a sub-optimal solution in which register usage exceeds the PRP. However, with accurate modeling of the register file during pre-allocation scheduling and a precise register allocation algorithm, we may assume that the PRP is a good approximation of the actual register usage in most cases. This assumption has been validated for the GPU used in the experimental evaluation.

On a GPU, a range of PRP values may produce the same occupancy value. To account for this, we introduce the *adjusted peak register pressure* (APRP) step function to model occupancy during instruction scheduling. The APRP of a given PRP value x is the maximum PRP value that gives the same occupancy as x.

For example, on the AMD GPU used in the experimental evaluation, there are vector general-purpose registers (VGPRs) and scaler general-purpose registers (SGPRs). VGPR usage is usually the bottleneck that determines occupancy, because the demand for VGPRs is much higher in most cases. A total of 256 VGPRs are available for each thread. Using 24 or fewer VGPRs gives the maximum occupancy of 10 wavefronts. Using more than 24 VGPRs gives occupancy values less than 10, as given by the equation:

$$Occupancy = \min(10, floor(64/ceiling(PRP/4)) \quad (1)$$

where PRP is the per-thread PRP of VGPR. Table 1 shows the occupancy values computed by Equation (1). For example, the table shows that PRP values of 25, 26, 27 and 28 give the same occupancy of 9 wavefronts. Therefore, all of these PRP values map to an APRP of 28. As explained in Section 5, this mapping can greatly improve the performance of the proposed B&B algorithm. If a schedule with a PRP of 28 is found at some point in the search, any schedule with a PRP value between 25 and 28 will not give better occupancy and can thus be pruned early to speed up the search.

Table 1. Register usage and occupancy on an AMD GPU

VGPR PRP	Occupancy
≤ 24	10
≤ 28	9
≤ 32	8
≤ 36	7
≤ 40	6
≤ 48	5
≤ 64	4
≤ 84	3
≤ 128	2
≤ 256	1
> 256	1 with spills

3 Previous Work

Compiler researchers have been studying instruction scheduling for many decades. However, most published work on scheduling, especially earlier work, focused on exploiting ILP. Published work on RP-aware scheduling or balancing RP and ILP is limited. Work on RP-aware scheduling for the GPU is even more limited. In this section, we summarize previous work on RP-aware scheduling, as well as some related work on GPU performance.

Instruction scheduling for minimum RP, or the Minimum-Register Instruction Sequence (MRIS) problem, has been studied since 1970 by Sethi and Ullman [24] who proposed an algorithm that solves the problem optimally for an expression tree. Unfortunately, expressions in current production compilers are not necessarily trees. Therefore, that algorithm cannot be used in a production compiler.

A more practical algorithm for avoiding high RP while scheduling for ILP was proposed by Goodman and Hsu [7] using a heuristic approach. The idea is to track RP during scheduling, and if RP approaches the physical limit (number of physical registers available), favor an instruction that minimizes RP. Other heuristic approaches were later proposed by Govindarajan et al. [8], Touati [29], and Barany and Krall [3]. Production compilers use similar heuristic approaches [19]. However, our experimentation shows that the scheduling algorithm in LLVM may produce poor schedules, because it does not achieve a good balance between RP and ILP.

Recently, some researchers proposed combinatorial approaches. Kessler [15] proposed a dynamic programming approach to the MRIS problem and the problem of balancing RP and ILP. Govindarajan et al. [8] and Barany and Krall [3] proposed Integer Linear Programming solutions to the MRIS problem. Malik [20] proposed a Constraint Programming (CP) solution for the MRIS problem, and Domagala et al. [6] used CP to integrate RP-aware scheduling and loop unrolling. Lozano et al. [18] also used CP to solve the integrated instruction scheduling and register allocation problem.

In previous work, we proposed a B&B algorithm for solving the RP-aware scheduling problem [26, 28]. Using that algorithm, we achieved significant performance gains relative to LLVM's scheduling algorithm on Intel and ARM processors. As detailed in Section 4, that algorithm is not suitable for the GPU. The B&B algorithm proposed in the current paper uses a two-pass approach and a GPU-specific cost function to satisfy the scheduling needs of the GPU.

The only published work that directly addresses the problem of RP-aware scheduling on the GPU is the work of Rawat et al., who describe a re-ordering algorithm that minimizes RP for stencil computation on the GPU [21]. They model stencil computation as a DAG of trees. For each tree, they use the Sethi-Ullman algorithm [24] to find the optimal order. At the DAG level, they use a greedy heuristic to schedule the DAG of trees. They report speedups in the range of 1.22x to 2.43x for the NVCC compiler and 1.15x to 2.08x for the LLVM compiler. Rawat et al. later generalize their work and develop a source-to-source instruction reordering strategy for reducing RP [22]. They apply the generalized algorithm to various kinds of benchmarks on two different multi-core Intel processors and report significant performance gains.

Volkov studied the interaction between occupancy and ILP and its effect on GPU performance [30]. He showed that in many cases, it may be possible to achieve better performance by lowering occupancy and exploiting more ILP. It is not clear what compiler scheduling algorithm was used in Volkov's experiments. As shown in the experimental evaluation, the greedy algorithms that most production compilers currently use are unable to balance occupancy and ILP; they often maximize occupancy at the cost of degrading ILP. Our proposed algorithm is capable of optimizing both occupancy and ILP, as it intelligently searches for ILP-optimal schedules at higher occupancy levels.

In addition to scheduling, some researchers studied alternative approaches to improving the performance of GPU applications. Hong et al. describe an approach to optimizing

GPU kernels by identifying bottleneck resources and then searching for a combination of optimizations to alleviate the bottleneck [13]. Their GPU performance model is based on abstract kernel emulation. Hayes and Zhang propose a unified memory allocation framework to alleviate single-thread RP [10] and a framework for GPU occupancy tuning [9].

4 Background and Motivation

The proposed algorithm uses a B&B enumeration technique similar to that proposed in previous work [28], but it is based on a two-pass approach rather than a single-pass weighted-sum approach and has additional features that were designed specifically for the GPU. In this section, we give a brief description of the previous algorithm and explain why it does not perform well on the GPU.

4.1 Previous Algorithm

Our previous algorithm for solving the RP-aware scheduling problem optimizes a weighted sum of schedule length and RP. The weight of RP relative to schedule length must be set to a very high value on the GPU, because reducing RP is critically important. The algorithm iteratively invokes a B&B enumerator that searches for a schedule with minimum RP at a given target schedule length.

First, the algorithm uses a heuristic to find an initial schedule. Then, the enumerator is invoked iteratively at different target lengths starting at a schedule-length lower bound and ending at the maximum schedule length that can possibly improve the weighted sum. The schedule-length lower bound is computed using the algorithm of Langevin and Cerny [16]. The algorithm terminates when it determines that finding a lower RP schedule at the next target length cannot improve the current best weighted sum.

We described multiple cost functions for modeling RP during scheduling, including the Sum of Live Interval Lengths (SLIL) [26] and the Peak Excess Register Pressure (PERP) [28]. Excess RP is the difference between RP and the physical limit, and the PERP is the maximum excess RP at any point in the schedule.

In each iteration, the B&B enumerator searches exhaustively for a schedule that minimizes the RP cost function at a given target length. To efficiently conduct an exhaustive search, certain pruning techniques are used.

4.2 GPU-Specific Scheduling Needs

Although our previous algorithm produced good results on the CPU, it did not work well on the GPU, because of the following differences between the GPU and the CPU:

1. In compiling for the CPU, the purpose of minimizing RP is to avoid spilling. As long as RP does not exceed the number of physical registers, no spills will be generated, and performance won't be affected. Our previous results show that spills occur in only a subset of scientific applications [26]. In compiling for the GPU, spilling is extremely costly [4] and rare, and the main purpose of reducing RP is increasing occupancy, which highly impacts the performance of a wider range of GPU applications. Reducing RP by one register may increase occupancy from 1 to 2, thus potentially doubling the speed of a GPU application.
2. Although the GPU hardware has a wavefront-level scheduler, it does not have an instruction-level scheduler within a thread. The GPU scheduler can hide latencies by switching between ready-to-execute wavefronts [1], but it does not do any reordering within a thread. Therefore, exploiting ILP within a thread is the responsibility of the compiler, and compiler scheduling for ILP can give a significant performance gain. Our experimental results show that schedulers that deemphasize ILP, like the LLVM scheduler, generate poor code for the GPU. Therefore, a good scheduling algorithm for the GPU must spend adequate time optimizing *each* of the two objectives (RP and ILP).
3. As explained in Section 2, multiple PRP values may give the same occupancy. Therefore, an efficient scheduling algorithm for the GPU should not waste time searching for a schedule with a lower PRP unless it increases occupancy.
4. A GPU kernel generally consists of multiple scheduling regions. The occupancy of a kernel is a function of the highest register usage in any region in the kernel. For example, if a kernel running on an AMD GPU consists of ten scheduling regions, and one region uses 32 VGPRs, while the other nine regions use 24 VGPRs each, the kernel occupancy will be 8 (see Table 1), because occupancy is determined by the bottleneck region that uses 32 registers. Therefore, limiting register usage in the nine non-bottleneck regions to 24 when optimizing ILP will be unnecessarily restrictive. Allowing all ten regions to use up to 32 registers will give a much higher degree of freedom in optimizing ILP in the non-bottleneck regions. This, however, requires making kernel-level register usage information available to all scheduling regions in the kernel, which cannot be done in a single-pass scheduler.
5. Another factor that may limit occupancy in a kernel is local data share (LDS) usage. Since LDS usage is known before scheduling, an effective GPU scheduling algorithm can take advantage of this to avoid searching for lower RP values that will not give better occupancy. For example, if it is known that occupancy is limited to 8 because of LDS, the scheduler will not need to find schedules that use fewer than 32 VGPRs. Taking advantage of this will both tighten the RP solution space and relax the RP constraint when optimizing ILP.

4.3 Disadvantages of the Previous Algorithm

The previous algorithm does not perform well on the GPU, because it does not satisfy any of the GPU-specific needs described in the previous subsection. The details are:

1. It first searches for a maximum-occupancy (minimum RP) at the schedule-length lower bound. Such a schedule is unlikely to exist, because the shortest possible schedule is unlikely to use the fewest registers. Usually, a minimum-RP schedule tends to use more cycles. Therefore, the algorithm may waste substantial time searching for a schedule that does

not exist. Thus, it may timeout without finding a maximum-occupancy schedule. This is unacceptable on the GPU, as maximizing occupancy is very important.
2. It does not take advantage of the fact that multiple PRP values map to the same occupancy, and that may cause it to waste time searching for a lower RP schedule that does not actually give higher occupancy.
3. It does not take advantage of the fact that the kernel occupancy is determined by register usage in the bottleneck region, and thus over-constrains the search in the ILP pass.
4. It does not take advantage of LDS information to limit the search and better optimize ILP.

5 Algorithm Description

In this section, we describe the new algorithm that we have designed to address the GPU-specific scheduling needs described in the previous section. The pseudo-code is listed in Alg. 1. The proposed algorithm is a two-pass algorithm that first finds a possibly-long maximum-occupancy schedule, and then searches for the shortest schedule that maintains that maximum occupancy. In the occupancy pass, instruction latencies are ignored (set to unity) and the B&B enumerator is invoked to search for a minimum-APRP (maximum-occupancy) schedule. In the ILP pass, the B&B enumerator is invoked iteratively to search for the shortest possible schedule that gives the best kernel-level occupancy found in the occupancy pass. The best kernel-level occupancy is the best occupancy that could be found in the bottleneck scheduling region in that kernel.

The B&B enumerator used in this work is based on that proposed in previous work [26, 28] but is modified to work efficiently in each of the two passes of the current algorithm. The details are described in the following subsections.

5.1 The B&B Enumerator

The enumerator takes as input a scheduling region with a data dependence graph (DDG), a target schedule length and a target APRP. It exhaustively searches for a *feasible schedule*. A feasible schedule is a schedule that satisfies the latency constraints in the DDG, its length is equal to the target length and its APRP is less than or equal to the target APRP. The target APRP is the APRP that corresponds to the target occupancy. In Table 1, for example, a target occupancy of 8 corresponds to a target APRP of 32 VGPRs.

The APRP passed to the enumerator for each scheduling region is the kernel-level target APRP. In the occupancy pass, the kernel-level target occupancy is normally the GPU's maximum occupancy, but a lower target occupancy may be used because of LDS usage constraints or manual occupancy target lowering via a command-line option. In the ILP pass, the target occupancy for each scheduling region is the best occupancy that was actually achieved in the occupancy pass for the bottleneck scheduling region in the kernel.

The B&B enumerator searches for a schedule that meets both the length and the APRP targets by constructing a schedule incrementally. At each step, it adds an instruction or a stall. The search can be represented by a decision tree in which the root represents an empty schedule, leaf nodes represent complete schedules and internal nodes represent partial schedules. After adding one instruction to the schedule, the enumerator steps forward from the current tree node to one of its child nodes.

The enumerator stores the current best schedule and the current best cost found so far. Initially, the current best schedule is the initial schedule input to the enumerator. In the occupancy pass, the initial schedule is produced using some heuristic, such as LLVM's generic heuristic. In the ILP pass, the initial schedule is the best schedule found in the occupancy pass, as explained below. Whenever the enumerator finds a lower cost schedule, it updates the current best schedule and the current best cost.

The enumeration algorithm applies at each tree node certain pruning techniques to eliminate infeasible or non-promising solutions as early as possible. A pruning technique is either a feasibility test that checks if a feasible schedule may be found below the current tree node or a cost test that checks if there is a feasible schedule with a lower cost than the current best cost. If any feasibility or cost test fails, the enumerator undoes the last decision and backtracks to the previous tree node to examine another option.

In our B&B enumerator, we use the pruning techniques proposed in previous work [26, 28] but modify them to take advantage of the special properties of each pass as follows.
1. Range Tightening (Lines 1 and 2 in `CheckNode()`). The decisions made by the enumerator can be used to tighten instructions' *scheduling ranges*. A scheduling range is the range of cycles in which an instruction may appear in a feasible schedule [25]. For example, if an instruction initially has a range of [4-7], and the enumerator fills cycle 4 with other instructions, cycle 4 will no longer be feasible for this instruction, and the scheduling range will be tightened to [5-7]. Range tightening is not needed in the occupancy pass, because, with all latencies set to unity, a schedule length that is equal to the number of instructions is always feasible.
2. Dynamic Lower Bound (Lines 3 and 4 in `CheckNode()`). Rim and Jain's algorithm [23] is used to compute a dynamic lower bound (DLB) on the schedule length at the current node. This DLB is then compared with the number of cycles (used and remaining) to determine if the target length is still feasible. The DLB is not needed in the occupancy pass, because the target length is always feasible.
3. Register Pressure Cost (Lines 6, 7 and 8 in `CheckNode()`). The enumerator computes the RP cost (APRP) at each tree node. Since APRP is a monotonically increasing function during the search (its value at a child node will always be greater than or equal to its value at its parent), the APRP at the current node is a valid lower bound on the APRP at any node in the sub-tree below it. Therefore, if the APRP value at the current node is not less than the best cost found so far,

```
Function Schedule(kernel)
    targetAPRP = GetTargetAPRP()
    foreach region in the kernel do
        OccupancyPass(region)
        targetAPRP = UpdateTargetAPRP(region.bestSched)
    foreach region in the kernel do
        ILPPass(region)
Function OccupancyPass(region)
    region.bestSched = FindHeuristicSched()
    if region.bestSched.APRP > targetAPRP then
        SetAllLatenciesToOne(region.DDG)
        Enumerate(region, region.instCnt, targetAPRP)
Function ILPPass(region)
    UB = SatisfyLatencies(region.DDG, region.bestSched)
    LB = ComputeLB(region.DDG)
    if region.bestSched.length == LB then return
    for targetLength = LB to UB-1 do
        targetsMet = Enumerate(region, targetLength, targetAPRP)
        if targetsMet then return
    return
Function Enumerate(region, targetLength, targetAPRP)
    targetsMet = FALSE
    enumBestAPRP = region.bestSched.APRP
    currentNode = rootNode
    while ! allNodesExplored && ! targetsMet do
        node = GetNextFeasibleNode()
        if CheckNode(node, targetLength, targetAPRP) == TRUE then
            currentNode=StepForward(node)
        else
            if currentNode == rootNode then
                allNodesExplored = TRUE
            else
                currentNode=BackTrack()
    if targetsMet then region.bestSched = enumBestSched
Function CheckNode(node, targetLength, targetAPRP)
    feasible = TightenSchedRanges(node)
    if !feasible then return FALSE
    DLB = ComputeDLB(node)
    if DLB > targetLength then return FALSE
    if CheckHistory(node) == FALSE then return FALSE
    APRP = ComputeAPRP(node)
    if pass==occupancy then return APRP < enumBestAPRP
    else return APRP <= targetAPRP
```

Algorithm 1: The proposed two-pass B&B algorithm

the entire sub-tree below the current node may be pruned. The APRP is computed incrementally by updating the set of live registers after each decision.

4. History-Based Domination (Line 5 in `CheckNode()`). In this technique, information about previously enumerated tree nodes is stored in a history table and then used to prove, under certain conditions, that the current node cannot have below it a better schedule than the best schedule below the history node. The current node is compared with previously visited *similar* nodes. Two nodes are similar if their partial schedules are permutations of the same set of instructions.

5.2 Occupancy Pass

In this pass, ILP is ignored by setting all latencies in the DDG to one and assuming a single-issue target machine. With these settings, the schedule length will be equal to the number of instructions for any instruction order, thus effectively ignoring ILP and scheduling for the sole objective of minimizing the APRP (maximizing occupancy). This ensures that the algorithm spends sufficient time maximizing occupancy. Note that setting latencies to one and focusing on minimizing APRP simplifies the problem and makes it possible to do a faster search. Longer latencies make the scheduling problem harder and thus slow down the search.

A certain time limit is set on the search time in this pass. The best schedule found within this limit is input to the ILP pass. Note that the schedule found in this pass may not be optimal if the search does not complete within the time limit.

5.3 ILP Pass

In this pass, actual latency values and machine modeling information are used, and the B&B enumerator is invoked iteratively to search for the shortest schedule that maintains the best kernel-level occupancy found in the occupancy pass. In each iteration, the B&B enumerator is invoked with a different target length. The first target length is the schedule-length lower bound that is computed using the algorithm of Langevin and Cerny [16]. The last target length is an upper bound that is computed as follows.

The best-occupancy schedule found in the occupancy pass will, most likely, be an invalid schedule, because it does not satisfy latency constraints. At the beginning of the ILP pass, this schedule is converted into a valid schedule by adding stalls to satisfy latency and resource constraints. Adding such stalls produces a valid schedule with the best APRP found in the occupancy pass (the target APRP). The length of this schedule is an upper bound on the target length that needs to be considered.

The algorithm searches for the shortest schedule that meets the target APRP by iteratively invoking the B&B enumerator with target lengths starting at the lower bound and ending at the upper bound. If a feasible schedule that meets both the target APRP and the target length is found, that schedule is optimal and the search terminates.

It should be noted that in the proposed two-pass algorithm, each pass has its own time limit to ensure that the algorithm spends enough time optimizing each objective.

5.4 Example

In this subsection, the proposed algorithm is illustrated by applying it to the seven-instruction DDG in Figure 1. The Def and Use sets of each instruction are shown on the DDG, where r1, r2, . . . , r7 are virtual registers. For simplicity, we assume that scheduling is done for a single-issue machine, and that each PRP value gives a unique occupancy value (APRP is always equal to PRP).

First, we apply the previous B&B algorithm that optimizes a weighted sum of schedule length and PRP [26, 28]. To express the importance of PRP relative to schedule length on the GPU, the weight of PRP must be set to a sufficiently large value. Let's assume that this value is 10 (increasing PRP by 1 is as costly as increasing schedule length by 10). The algorithm starts the search at the schedule-length lower bound. An effective lower-bound algorithm, such as the algorithm of Rim and Jain [23], will compute a tight lower bound of 8 cycles for this DDG.

Thus, the previous algorithm will start the search at target length 8 and will find the 8-cycle schedule shown on the left in Figure 1.b. This schedule has a PRP of 4, because at Cycle 4, the four registers defined by instructions A, B, C and D are simultaneously live. The algorithm will then iteratively search for a lower PRP schedule at larger target lengths.

First, the algorithm will search for a lower RP schedule at target length 9 but will not find such a schedule. Then it will search at length 10 and will successfully find the 10-cycle schedule with a PRP of 3 shown on the right in Figure 1.b. For this simple DDG (which happened to be a tree), a PRP of 3 is clearly optimal, but there is no known polynomial-time algorithm for computing a tight lower bound on the PRP for a general DDG. Therefore, the algorithm will not terminate at this point and will continue to search for a schedule with a lower PRP at larger schedule lengths.

Since the weight of PRP relative to schedule length is 10, reducing the PRP by 1 will decrease the weighted sum by 10 points, while increasing the schedule length by 9 will increase the weighted sum by only 9 points. Accordingly, the algorithm will search for lower PRP schedules at target lengths between 11 and 19. This search will clearly waste compile time without finding any better schedule.

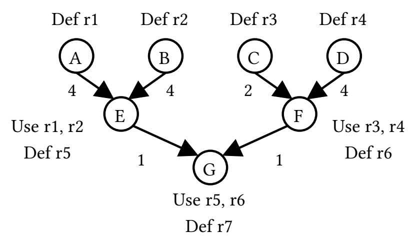
Next, we show how the proposed algorithm schedules this DDG. In the occupancy pass, the proposed algorithm will set all latencies to unity and search for a minimum-APRP schedule. It will find the seven-cycle schedule shown on the left in Figure 1.c. Note that by ignoring ILP, the search for a minimum PRP schedule can be done much faster. Specifically, it is already known that any schedule in the occupancy pass will have exactly seven cycles (the number of instructions). Therefore, there will be no need to compute a schedule-length lower bound at each node to check for feasibility.

In the ILP pass, latencies are accounted for again. At the beginning of the ILP pass, the proposed algorithm will add stalls to the schedule found in the occupancy pass to satisfy latency constraints. The resulting schedule is the 13-cycle schedule shown in the middle of Figure 1.c. This schedule has the minimum PRP of 3 but its length is not optimal. The length of this schedule is an upper bound on the target length that needs to be considered in the ILP pass (compare this with an upper bound of 19 in the previous algorithm).

In the ILP pass, the proposed algorithm will search for a shorter schedule with a PRP of 3. In the first iteration, it will search at the schedule-length lower bound, which is 8. In this iteration, the B&B algorithm will search for a schedule that meets both a target length of 8 and a target PRP of 3. It will backtrack as soon as it determines that with the current partial schedule, one of these two targets cannot be met.

Having to meet two targets greatly tightens the solution space and thus speeds up the search. In this iteration, the algorithm will quickly determine that meeting both targets is impossible. More specifically, whenever the algorithm

(a) DDG



(b) Previous algorithm

At length 8	At length 10
1: A	1: A
2: B	2: B
3: D	3: D
4: C	4: stall
5: stall	5: stall
6: E	6: E
7: F	7: C
8: G	8: stall
PRP = 4	9: F
	10: G
	PRP = 3

(c) Proposed algorithm

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

Figure 1. Example

constructs a partial schedule that starts with A, B, C and D (in any order), it will determine that the PRP is already 4 (greater than the target) and will thus backtrack without exploring any deeper nodes. Similarly, whenever it constructs a partial schedule that starts with A and B (in any order) and adds two stalls to schedule E before C or D, it will determine that, with 4 cycles used and 5 cycles remaining, the target length of 8 cannot be met. Therefore, the algorithm will determine within the top 4 levels in the tree hat meeting both targets is impossible.

Next, the proposed algorithm will explore a target length of 9 and will determine fairly quickly that it is impossible to achieve a PRP of 3 at this length. The search at this length could be slightly slower than the search at length 8 (because it is less constrained), but using the Rim-and-Jain lower bound will allow the B&B algorithm to determine without exploring deeper nodes that a target length of 9 cannot be met with any partial schedule that maintains a PRP of 3.

In the next iteration, the algorithm will search for a schedule with a PRP of 3 at target length 10 and will find the right-most schedule in Figure 1.c. Unlike the previous algorithm, the proposed algorithm will now terminate with a provably optimal schedule. Since a PRP of 3 is known to be optimal, there will be no point in considering longer schedules. This early termination with a provably optimal schedule is a great advantage of the proposed algorithm relative to the previous algorithm. This advantage is more pronounced on architectures having instructions with very long latencies, such as memory instructions on a GPU.

Another advantage of the proposed algorithm can be seen by considering a short time limit that causes both algorithms to timeout (output the best schedules found so far before completing the search). In this case, the previous algorithm may output the leftmost schedule in Figure 1.b, which gives low occupancy, while the proposed algorithm may output the leftmost schedule in Figure 1.c, which gives higher occupancy. With maximizing occupancy being the primary objective, the latter schedule is better.

6 Experimental Results

6.1 Experimental Setup

The proposed B&B algorithm was implemented in the LLVM compiler for the AMD GPU target. The LLVM revision used in our evaluation was frozen in April 2019 (LLVM 9.0). The tests were run on an AMD Radeon RX Vega 64 GPU running at 1.63 GHz. This GPU has 64 compute units, 40 waves per compute unit and 64 threads per wave. Compilation for the GPU was done on an Intel® Xeon® E3-1245 v5 processor running at 3.5 GHz. The ROCm 2.3 software stack with OpenCL was used. The benchmarks are 13 machine learning programs from the PlaidML framework [14].

The GPU used in this evaluation can issue one instruction every four cycles. An instruction takes at least four cycles to complete, but some instructions take more cycles. If the current instruction depends on a previous instruction and the results are not available, the current instruction must wait until the results become available. The hardware sequencer will switch to a different wavefront if it finds a wavefront with a ready instruction. If it does not, the compute unit must stall. Therefore, the compiler scheduler can minimize the stall time by hiding long latencies within a thread.

The performance of the proposed B&B algorithm was evaluated relative to each of the following algorithms:
1. The generic scheduling algorithm in LLVM [17], which is a greedy algorithm based on list scheduling [5].
2. AMD's production scheduler [2], which is a greedy algorithm that is built on the LLVM scheduler. It extends the LLVM scheduler to model GPU-specific factors, including the APRP and using kernel-level occupancy to relax RP constraints and achieve better ILP. This scheduler is well tuned for the ROCm architecture. Therefore, it is the state-of-the-art scheduler for the AMD GPU used in this evaluation.
3. The previous B&B scheduling algorithm [26, 28].

We note that the AMD GPU backend includes a load-store clustering pass that is invoked before the scheduling algorithm (whether it is the AMD algorithm or the proposed B&B algorithm). In this pass, constraints are added to the DDG based on certain heuristics to force loads and stores to be scheduled in clusters to improve memory performance.

6.2 Benchmark Information

The benchmarks used in this evaluation are machine learning programs using the PlaidML framework [14]. Table 2 shows some information about these benchmarks. There are 13 benchmarks, in which there are 3813 GPU kernels containing 16642 scheduling regions, which is a large enough dataset to give statistical significance. The average scheduling region size is 48.8 instructions, which is significantly larger than the sizes reported on the CPU [26]. The largest region has 921 instructions, which is challenging to schedule optimally.

Table 2. Benchmark Information

Stat	Value
Total number of benchmarks	13
Total number of kernels	3813
Total number of scheduling regions	16,642
Total number of instructions	811,864
Avg. region size	48.8
Max. region size	921

6.3 Combinatorial Scheduling Performance

Table 3 shows statistical information about the performance of the proposed B&B algorithm in each pass. The initial heuristic schedule in the occupancy pass was generated using the LUC heuristic described in previous work [27]. Each pass was given a time limit of 1ms/instr. So, a 100-instruction region was given 100ms.

First, we comment on the results for the occupancy pass. The total number of scheduling regions in the benchmark set is 16642 regions. Only 541 regions (3.3%) were passed to the B&B scheduler. The rest of the scheduling regions (96.7%) were not passed to the B&B scheduler, because their heuristic schedules were already giving the maximum occupancy. Although the B&B scheduler was applied to only 3.3% of the regions, these regions are larger regions with higher RP, and are thus expected to have a higher impact on performance. Row 7 shows that the average size of a region passed to the B&B scheduler is 209 instructions, which is about four times larger than the average region size across the entire set.

Rows 3 and 4 together show that, with the given time limit, the B&B scheduler scheduled 13.0% of the regions to optimality, and that 11.7% were improved relative to the heuristic. For 1.3% of the regions, the B&B scheduler completed within the time limit without finding a better schedule.

Table 3. B&B Performance

	STAT	Occupancy Pass	ILP Pass
1	Total regions processed	16642	16642
2	Regions passed to B&B	541 (3.3%)	10537 (63.3%)
3	Regions opt. and imp.	63 (11.7%)	9513 (90.3%)
4	Regions opt. and not imp.	7 (1.3%)	0 (0.0%)
5	Regions timed out and imp.	226 (41.8%)	152 (1.4%)
6	Regions timed out and not imp.	245 (45.3%)	872 (8.3%)
7	Avg. region size passed to B&B	209	70
8	Largest opt. region	287	877
9	Largest imp. region	921	921

Row 5 shows that 41.8% of the regions were improved relative to the heuristic although the exhaustive search did not complete within the time limit. Row 6 shows that 45.3% of the regions timed out with no improved schedule.

More advanced algorithmic techniques are needed to optimally schedule the harder scheduling regions. In future work, we plan on applying parallelization and graph transformations [11, 12]. Interestingly, the execution-time results in the next subsection show that, in spite of the timeouts, the proposed algorithm gives significant performance gains relative to the other three algorithms.

Next, we comment on the results of the ILP pass. Row 2 shows that 63% of the scheduling regions were passed to the B&B scheduler. This relatively high percentage (compared to 3.3% for the occupancy pass) is attributed to the fact that the schedule input to the ILP pass is the minimum-RP schedule found in the occupancy pass. Due to the inherent conflict between RP and ILP, the minimum-RP schedules will be relatively long (see the middle schedule in Figure 1.c).

Row 3 shows that 90% of the schedules passed to the B&B scheduler were scheduled to optimality and improved relative to the minimum-RP schedules. Row 4 shows that all optimally scheduled regions were improved. Row 5 shows that 1.4% of the instances timed out but with improved schedules relative to the minimum-RP schedules. Row 6 shows that 872 instances timed out without improvement.

Table 4 shows the overall effect of the proposed algorithm on both occupancy and schedule length relative to the other algorithms, namely LLVM, AMD and the previous B&B (Prev B&B in the table). The weight of RP relative to schedule length was set to 100000 in the previous B&B scheduler to reflect the importance of RP on the GPU. For a fair comparison, a time limit of 1ms/instr was used in each pass of the proposed B&B scheduler, and a time limit of 2ms/instr was used for the previous B&B scheduler. So, each algorithm was given a total limit of 2ms/instr. The table shows the percentage improvement achieved by the proposed algorithm in occupancy and schedule length relative to each of the other three algorithms. The aggregate occupancy and schedule length for each scheduler were computed by taking the total sum of occupancies across all kernels and the total sum of schedule lengths across all scheduling regions.

The numbers in Table 4 show that the proposed algorithm increases occupancy by 4.13% relative to the LLVM scheduler,

Table 4. Improvements in occupancy and sched. length

	Occupancy	Sched Length
%Imp relative to LLVM	4.13%	35.61%
%Imp relative to AMD	4.14%	19.20%
%Imp relative to Prev B&B	4.09%	7.35%

by 4.14% relative to the AMD scheduler and by 4.09% relative to the previous B&B scheduler. The proposed B&B scheduler produces these improvements in spite of the significant number of timeouts in the occupancy pass. This suggests that potentially higher occupancy values may be achieved in the future with further algorithmic enhancements.

Table 4 shows that, on average, the previous B&B scheduler produces lower occupancies than the proposed B&B scheduler. This is attributed to the fact that the previous B&B scheduler starts its search at the schedule-length lower bound and a maximum-occupancy schedule is unlikely to exist at that length (compare the leftmost schedule in Figure 1.b with the leftmost schedule in Figure 1.c).

In terms of schedule length, the proposed B&B scheduler produces significant improvements relative to other schedulers. It improves schedule length by 35.61% relative to LLVM, by 19.20% relative to AMD and by 7.35% relative to previous B&B. The large improvements in schedule length relative to LLVM and AMD are expected, because both schedulers are heuristic schedulers that are heavily biased towards minimizing RP (maximizing occupancy), and minimizing RP tends to increase schedule length. This result shows that it is extremely hard to balance two conflicting objectives using a heuristic approach. The proposed combinatorial approach produces better results for both occupancy and ILP, because it first searches for a maximum-occupancy schedule in the occupancy pass and then it searches for a minimum-length schedule among all maximum-occupancy schedules in the ILP pass. These results confirm that an intelligent search technique can give significantly better results than a heuristic technique for solving the RP-aware scheduling problem.

The results in Table 4 show that the proposed B&B algorithm gives a significant improvement in schedule length relative to the previous B&B algorithm. This is attributed to the fact that the proposed algorithm uses the kernel-level occupancy rather than the region-level occupancy as a target occupancy in the ILP pass as explained in Sections 4 and 5.

6.4 Execution Times

In this subsection, we present the actual execution-time improvements achieved using the proposed algorithm relative to the other algorithms. Execution times for the machine learning benchmarks used are measured in examples processed per second. The time limit was set to 1ms/instr per pass for the proposed B&B algorithm and 2ms/instr for the previous B&B algorithm.

Table 5 shows the percentage increase in speed (examples per second) produced by the proposed algorithm relative to each of the other algorithms. Each benchmark was run three

Table 5. Execution-time improvements

Benchmark	%Imp relative to LLVM	%Imp relative to AMD	%Imp relative to PREV B&B
Densenet121	16.95%	3.29%	8.90%
Densenet169	19.81%	2.53%	6.91%
Densenet201	25.82%	2.11%	10.54%
Imdb_lstm	2.75%	1.65%	2.47%
Inception_resnet_v2	18.23%	10.85%	7.50%
Inception_v3	34.95%	31.49%	17.65%
Mobilenet	12.20%	9.76%	0.61%
Nasnet_large	9.39%	1.16%	3.45%
Nasnet_mobile	13.50%	1.10%	3.58%
Resnet50	17.70%	1.91%	5.02%
Vgg16	15.29%	4.41%	2.75%
Vgg19	16.65%	4.79%	6.47%
Xception	12.02%	-0.34%	5.51%
Geo-mean	16.32%	5.47%	6.18%

times using each algorithm, and the median speed was used in computing the percentage differences in the table. The proposed algorithm speeds up every benchmark relative to each of the other algorithms (except for a negligibly small regression on Xception relative to AMD). The geometric-mean speedup is 16.32% relative to LLVM's scheduler, 5.47% relative to AMD's scheduler and 6.18% relative to the previous B&B scheduler. The maximum speedup is seen on Inception_v3, where the proposed algorithm gives a speedup of 34.95% relative to LLVM, 31.49% relative to AMD and 17.65% relative to the previous B&B algorithm.

The 16.32% geometric-mean speedup of the proposed algorithm relative to LLVM's algorithm shows the importance of compiler instruction scheduling on the GPU. Unlike the AMD scheduler, LLVM's generic scheduler is not tuned for the AMD GPU. The proposed scheduler gives a substantial geometric-mean improvement of 5.47% relative to AMD's well-tuned production scheduler, which shows that combinatorial scheduling can make a significant difference.

The fact that the previous B&B algorithm under-performs AMD's algorithm shows that a combinatorial algorithm may under-perform a well-tuned heuristic algorithm if it does not efficiently use the time limit. As explained in Sections 4 and 5, a major difference between the proposed B&B algorithm and the previous B&B algorithm is the search order. The previous algorithm is not guaranteed to spend enough time optimizing occupancy.

6.5 Compile Times

Table 6 shows the total compile time for all benchmarks when each scheduling algorithm is used. The time limit was 1ms/instr per pass for the proposed B&B (New B&B) and 2ms/instr for the previous B&B (Prev B&B) algorithm.

Clearly, using combinatorial scheduling substantially increases compile time, which is an expected result. Using combinatorial optimization techniques like B&B or Constraint Programming (CP) in compilers is still in the research phase, but our work is closer to practicality than any published combinatorial scheduling technique.

Table 6. Compile Times

Scheduler	Total Compile Time (s)
LLVM	256
AMD	259
Prev B&B	1287
New B&B	566

Table 7. Compile time details

	Time (s)
Total compile time	566
Scheduling time	283
Scheduling time in the occupancy pass	49 (17%)
Scheduling time in the ILP pass	234 (83%)

The results in Table 6 show that on the GPU, the compile time using the proposed B&B is less than half the compile time using the previous B&B algorithm, and Table 5 shows that the execution time is 6.18% faster with the proposed algorithm. Therefore, the proposed algorithm is a strong candidate for deployment in a production compiler. For performance-critical GPU applications, the increase in compile time may be tolerated if it produces a significant performance gain. For the reasons explained in the previous sections, the proposed algorithm has the potential to produce even more significant performance gains in the future.

Table 7 shows more details about the compile time of the proposed algorithm. The time spent in the scheduling algorithm is 283s (50% of the total compile time). Most of the scheduling time (83%) is spent in the ILP pass and only 17% of it is spent in the occupancy pass. As shown in Table 3, the number of scheduling regions processed by the B&B algorithm in the ILP pass is significantly greater than the number of regions processed in the occupancy pass.

It is important to note that the compile times reported in the current paper may be significantly reduced in the future for the following reasons:
1. Our current implementation is a research prototype that involves substantial overhead.
2. In our current work, we apply our B&B algorithm to all scheduling regions. Applying the algorithm to only the hot regions will significantly reduce the compile time.
3. In future work, we plan on exploring multiple ideas for speeding up the algorithm, including graph transformations [11, 12] and parallelization.

6.6 Time Limit

In this subsection, we explore the effect of the time limit on the compile time and the resulting performance of the proposed algorithm. Three different time limits between 0.25ms/instr and 5ms/instr per pass are explored. Recall that the base time limit used in previous subsections is 1ms/instr.

Table 8 shows that increasing the time limit significantly increases compile time from 49% relative to the base LLVM compiler with 0.25 ms/instr to 491% relative to the base compiler with 5ms/instr. This increase in compile time reduces the number of timeouts in both passes, but even with the highest time limit, many instances time out. Clearly, the algorithm's convergence is slow, and further increases of the time limit are unlikely to solve the remaining instances. More advanced algorithmic techniques are needed.

Table 8. Effect of time limit in ms/instr on compile time and timeouts

	0.25	1.0	5.0
Compile Time (s)	381 (49%)	566 (121%)	1513 (491%)
Occupancy Timeouts	483	471	453
ILP Timeouts	1049	1024	998

Table 9 shows the execution-time speedup produced by the proposed scheduler relative to AMD's scheduler for each of the time limits in Table 8. If the time limit is reduced from the base limit of 1ms/instr to 0.25ms/instr, performance drops by only about 1% (from 5.47% to 4.46%) in geometric-mean, but the savings in compile time are substantial. As shown in Table 8, compile time drops from 566 s to 381 s when the time limit is reduced from 1ms/instr to 0.25ms/instr. With a time limit of 0.25ms/instr, the geometric-mean performance gain relative to AMD's well-tuned scheduler is 4.46%, which is quite significant. This shows that the proposed algorithm can give substantial performance gains relative to a state-of-the-art heuristic with a reasonable increase in compile time.

It is noted that in Table 9, increasing the time limit does not always produce better run-time performance. For example, increasing the limit from 1ms/instr to 5ms/instr results in a slightly worse run-time performance for Inception_v3. This is attributed to the fact that the proposed algorithm models only two factors that affect performance, namely APRP and schedule length. It does not model other factors, such as memory-system performance and caching.

It should be emphasized that the proposed algorithm is a deterministic search algorithm not a stochastic algorithm. It keeps track of the best solution found so far and updates that best solution only if a better solution is found. So, if given more time, the proposed algorithm will either find a better solution or keep the same solution; it will never find a worse solution. However, the improvements found by the algorithm are relative to the cost function that it optimizes, and this cost function models only APRP and schedule length (ILP). So, even though the proposed algorithm will never find a schedule with worse APRP or worse schedule length, it may find a worse schedule relative to some un-modeled factor, thus degrading run-time performance.

Table 9. Effect of time limit in ms/instr on execution time

Benchmark	0.25	1.0	5.0
Densenet121	3.29%	3.29%	3.46%
Densenet169	2.53%	2.53%	2.66%
Densenet201	2.00%	2.11%	2.43%
Imdb_lstm	1.65%	1.65%	1.65%
Inception_resnet_v2	7.97%	10.85%	10.99%
Inception_v3	28.52%	31.49%	30.58%
Mobilenet	2.86%	9.76%	19.21%
Nasnet_large	1.52%	1.16%	2.05%
Nasnet_mobile	0.00%	1.10%	1.10%
Resnet50	1.91%	1.91%	1.91%
Vgg16	4.26%	4.41%	4.26%
Vgg19	4.17%	4.79%	4.17%
Xception	0.11%	-0.34%	1.95%
Geo-mean	4.46%	5.47%	6.34%

For a specific example, we have examined the effect of increasing the time limit on Inception_v3. We found that increasing the time limit from 0.25ms/instr to 1ms/instr improves the total APRP by 6% and the total schedule length by 0.5% across all scheduling regions. Increasing the time limit to 5ms/instr, results in improvements of 12% and 1% in APRP and schedule length, respectively relative to a time limit of 0.25ms/instr.

These consistent improvements in APRP and schedule length, however, do not result in consistent improvements in execution speed, because APRP and schedule length are not the only factors that determine the execution speed. Further analysis of this benchmark showed that it has many memory-bound kernels. For one particular big kernel, the proposed algorithm increases occupancy from 4 to 7. Increasing occupancy in a memory-bound kernel may degrade performance, because it increases contention for a bottleneck resource. In future work, we plan on extending the algorithm to model memory performance.

7 Conclusions and Future Work

This paper presents a two-pass B&B compiler scheduling algorithm that optimizes both occupancy and ILP on the GPU. Unlike the previous B&B algorithm, which simultaneously optimizes a weighted sum of RP and schedule length, the proposed algorithm optimizes each objective in a separate pass. This allows the algorithm to first focus on the primary objective of maximizing occupancy and then find the shortest schedule that achieves that maximum occupancy. Another important advantage of the proposed two-pass B&B algorithm is that searching for the shortest schedule in the ILP pass is constrained by the kernel-level rather than the region-level best occupancy, which allows the algorithm to find much shorter schedules for many regions.

In future work, we plan on exploring the parallelization of our algorithm and using graph transformations [11, 12] to solve the harder instances that timeout and reduce compile time. We also plan on modeling memory performance.

Acknowledgments

The authors thank Vang Thao for his work on preparing this paper for publication. We also thank the anonymous reviewers for their constructive feedback that greatly improved the final paper. This work was supported in part by the US National Science Foundation (NSF) through Award 1911235.

References

[1] AMD. 2012. *AMD Graphics Cores Next (GCN) Architecture*. White Paper.

[2] AMD. 2019. GCN Max Occupancy Scheduler. http://llvm.org/doxygen/classllvm_1_1GCNMaxOccupancySchedStrategy.html

[3] Gergö Barany and Andreas Krall. 2013. Optimal and Heuristic Global Code Motion for Minimal Spilling. In *Proc. Intl. Conf. on Compiler Construction.* Italy.

[4] Wei-Yu Chen, Guei-Yuan Lueh, Pratik Ashar, Kaiyu Chen, and Buqi Cheng. 2018. Register Allocation for Intel Processor Graphics. In *Proc. International Symposium on Code Generation and Optimization (CGO '18)*.

[5] Keith D. Cooper and Linda Torczon. 2012. *Engineering a compiler* (2nd ed.). Morgan Kaufmann.

[6] Łukasz Domagala, Duco van Amstel, Fabrice Rastello, and P. Sadayappan. 2016. Register Allocation and Promotion through Combined Instruction Scheduling and Loop Unrolling. In *Proc. International Conference on Compiler Construction.*

[7] James R. Goodman and Wei-Chung Hsu. 1988. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proc. Int'l Conf. Supercomputing*.

[8] Ramaswamy Govindarajan, Hongbo Yang, José N. Amaral, Chihong Zhang, and Guang R. Gao. 2003. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Trans. Comput.* 52, 1 (Jan 2003), 4–20. https://doi.org/10.1109/TC.2003.1159750

[9] Ari B. Hayes, Lingda Li, Daniel Chavarría-Miranda, Shuaiwen Leon Song, and Eddy Z. Zhang. 2016. Orion: A Framework for GPU Occupancy Tuning. In *Proc. 17th International Middleware Conference.*

[10] Ari B. Hayes and Eddy Z. Zhang. 2014. Unified On-Chip Memory Allocation for SIMT Architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. 293–302.

[11] Mark Heffernan and Kent Wilken. 2005. Data-Dependency Graph Transformations for Instruction Scheduling. *Journal of Scheduling* 8, 5 (01 Oct 2005), 427–451. https://doi.org/10.1007/s10951-005-2862-8

[12] Mark Heffernan, Kent Wilken, and Ghassan Shobaki. 2006. Data-Dependency Graph Transformations for Superblock Scheduling. In *Proc. 39th International Symposium on Micro-architecture (MICRO 39).*

[13] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2018. GPU Code Optimization Using Abstract Kernel Emulation and Sensitivity Analysis. In *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. 736–751.

[14] Intel. 2017. PlaidML machine learning benchmarks. https://github.com/plaidml/plaidbench#intel-corporation-machine-learning-benchmarks

[15] Christoph W. Kessler. 1998. Scheduling expression DAGs for minimal register need. *Computer Languages* (1998).

[16] Michel Langevin and Eduard Cerny. 1996. A Recursive Technique for Computing Lower-Bound Performance of Schedules. *ACM Trans. Des. Autom. Electron. Syst.* 1, 4 (Oct. 1996), 443–455. https://doi.org/10.1145/238997.239002

[17] LLVM. 2019. LLVM Generic Scheduler. https://llvm.org/doxygen/classllvm_1_1GenericScheduler.html

[18] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2018. Combinatorial Register Allocation and Instruction Scheduling. *arXiv preprint arXiv:1804.02452* (2018).

[19] Vladimir Makarov. 2013. Mechanism for Performing Instruction Scheduling based on Register Pressure Sensitivity. U.S. Patent No. 8,549,508.

[20] Abid Malik. 2008. *Constraint Programming Techniques for Optimal Instruction Scheduling.* Ph.D. Dissertation. University of Waterloo.

[21] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register Optimizations for Stencils on GPUs. In *Proc. 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18).*

[22] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *Proc. International Conference for High Performance Computing, Networking, Storage, and Analysis.* IEEE Press.

[23] Min Rim and Rajiv Jain. 1994. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 4 (April 1994), 451–458. https://doi.org/10.1109/43.275355

[24] Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. of the ACM* (1970), 715–728.

[25] Ghassan Shobaki. 2006. *Optimal Global Instruction Scheduling Using Enumeration.* Ph.D. Dissertation. Department of Computer Science, UC Davis.

[26] Ghassan Shobaki, Austin Kerbow, Christopher Pulido, and William Dobson. 2019. Exploring an Alternative Cost Function for Combinatorial Register-Pressure-Aware Instruction Scheduling. *ACM Trans. Archit. Code Optim.* 16, 1, Article 1 (Feb. 2019), 30 pages. https://doi.org/10.1145/3301489

[27] Ghassan Shobaki, Laith Sakka, Najm Eldeen Abu Rmaileh, and Hasan Al-Hamash. 2015. Experimental Evaluation of Various Register-Pressure-Reduction Heuristics. *Softw. Pract. Exper.* 45, 11 (Nov. 2015), 1497–1517. https://doi.org/10.1002/spe.2297

[28] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach. *ACM Trans. Archit. Code Optim.* 10, 3, Article 14 (Sept. 2013), 31 pages. https://doi.org/10.1145/2512432

[29] Sid-Ahmed-Ali Touati. 2005. Register Saturation in Instruction Level Parallelism. *International Journal of Parallel Programming* 33, 4 (01 Aug 2005), 393–449. https://doi.org/10.1007/s10766-005-6466-x

[30] Vasily Volkov. 2010. Better Performance at Lower Occupancy. In *Proc. the GPU technology conference (GTC).*