Warp-Aware Trace Scheduling for GPUs

James A. Jablin
Brown University
Dept. of Computer Science
jjablin@cs.brown.edu

Thomas B. Jablin
University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering
jablin@illinois.edu

Onur Mutlu
Carnegie Mellon University
Dept. of Electrical and Computer Engineering
onur@cmu.edu

Maurice Herlihy
Brown University
Dept. of Computer Science
herlihy@cs.brown.edu

ABSTRACT

GPU performance depends not only on thread/warp level parallelism (TLP) but also on instruction-level parallelism (ILP). It is not enough to schedule instructions within basic blocks, it is also necessary to exploit opportunities for ILP optimization beyond branch boundaries. Unfortunately, modern GPUs cannot dynamically carry out such optimizations because they lack hardware branch prediction and cannot speculatively execute instructions beyond a branch.

We propose to circumvent these limitations by adapting Trace Scheduling, a technique originally developed for microcode optimization. Trace Scheduling divides code into traces (or paths), and optimizes each trace in a context-independent way. Adapting Trace Scheduling to GPU code requires revisiting and revising each step of microcode Trace Scheduling to attend to branch and warp behavior, identifying instructions on the critical path, avoiding warp divergence, and reducing divergence time.

Here, we propose "Warp-Aware Trace Scheduling" for GPUs. As evaluated on the Rodinia Benchmark Suite using dynamic profiling, our fully-automatic optimization achieves a geometric mean speedup of 1.10× on a real system by increasing instructions executed per cycle (IPC) by a harmonic mean of 1.12× and reducing instruction serialization and total instructions executed.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors - Code Generation, Compilers, Optimization

General Terms

Languages, Performance

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628101.

Keywords

GPU; compiler optimization; instruction-level parallelism; global instruction scheduling; trace scheduling

1. INTRODUCTION

GPU architectures use a Single-Instruction, Multiple-Thread (SIMT) model marshaling thousands of threads across hundreds of cores, each thread having its own register state but all threads sharing one program counter. Threads are managed in groups called *warps* in CUDA, or alternatively wavefronts in OpenCL. Warp size varies, but for CUDA, warps contain 32 threads. GPUs schedule an entire warp of threads at once and warp threads cannot be scheduled separately. GPUs interleave warp execution to increase resource utilization and instruction throughput.

Each new generation of GPUs provides increasing levels of resources such as more registers, shared memory, functional units, and arithmetic cores. Most efforts to manage GPU resource utilization [17, 20, 25, 31, 32] have focused on thread or warp-level parallelism (TLP). Our contribution in this paper is to focus attention on the complementary use of instruction-level parallelism (ILP) to improve resource utilization. While this subject is well-understood for CPU architectures, it has received little attention for GPUs.

Modern GPU architectures [30] have neglected ILP for two major reasons. First, they typically lack branch prediction and hardware support for speculation. Second, the conventional GPU application has abundant TLP. However, as GPUs become more general purpose, it will become increasingly necessary to improve GPU performance by exploiting ILP within a single thread. By contrast many CPUs, recognizing the relative scarcity of ILP within basic blocks, implement hardware branch prediction, enabling faster execution by successfully guessing branch outcomes and fetching and executing instructions.

Trace Scheduling [9], first developed for microcode optimization, attempts to decrease execution time by exposing ILP across branch boundaries through static global instruction scheduling. Generally, Trace Scheduling organizes sequences of *basic blocks*, a series of instructions free of control flow besides the last instruction, into regions called *traces*. Optimization efforts then focus on scheduling instructions within traces. Trace Scheduling proceeds in three steps. *Trace selection* partitions basic blocks into traces. *Trace*

formation extends traces to expose additional opportunities for the final step, *local scheduling*. *Local scheduling* schedules instructions within each trace to reduce execution time.

Trace scheduling, as used previously for microcode, is not immediately applicable to GPUs. The principal challenge is avoiding warp divergence, which occurs when a warp's threads take different execution paths. GPUs serialize the different executions, executing first one path, then the other, at substantial cost to performance. It is well-established [6, 10–12, 18, 19, 23, 24] that avoiding divergence is performance critical.

This paper describes *Warp-Aware Trace Scheduling*, a new technique for GPU global instruction scheduling. Our optimization consists of three major steps, customized for the GPU execution model. The first step selects traces that start at basic blocks that are likely to be divergence-free. These traces define the earliest non-divergent entry for scheduling instructions for speculative execution. The second step harnesses the GPU's native compiler infrastructure for predication. In the final step, the instruction scheduler schedules long-latency memory operations (and their dependent instructions) as early as possible in a trace.

Because GPUs have little or no exception handling, any such speculation must be conservative; a failed speculation may reduce performance, but will not require rolling back speculative computations. For this reason, we only speculate on global memory `load`'s and arithmetic instructions. When appropriate, the scheduler moves instructions from divergent to non-divergent basic blocks, reducing divergence time.

We make the following contributions:

1. Comprehensive rethinking and redesign of CPU-oriented trace scheduling techniques to accommodate GPU requirements such as avoiding divergence when possible, and reducing divergence time otherwise.

2. Development and evaluation of Warp-Aware Trace Scheduling, a fully-automatic ILP optimization for reducing execution time.

Across the entire Rodinia Benchmark Suite [5], run on a real system (NVIDIA GTX 670), Warp-Aware Trace Scheduling achieves a geometric mean speedup of 1.10× by improving instructions executed per cycle (IPC) by a harmonic mean of 1.12× and reducing instruction serialization and total instructions executed. Our results suggest that Warp-Aware Trace Scheduling can be a promising way of improving single-thread performance on GPUs, thereby enabling the architecture to execute a wider variety of programs. More aggressive speculation, made possible by adding new *speculative* `load` and *exception* `check` instructions (see Section 4.3 for further details), and complementary optimization will likely further increase performance.

The organization for the rest of the paper follows. Section 2 describes PTX and our profiling infrastructure. Section 3 motivates the importance of optimizing GPU ILP. Section 4 first provides a brief overview of Trace Scheduling, and then in subsequent subsections details our approach to adapting Trace Scheduling to the GPU. Afterward, Section 5 describes our experimental methodology, and Section 6 reports Warp-Aware Trace Scheduling's results on the Rodinia Benchmark Suite. Section 7 interprets and discusses our results, and Section 8 describes related work. Section 9 posits future research opportunities to complement and improve Warp-Aware Trace Scheduling, and Section 10 concludes.

2. BACKGROUND

To evaluate Warp-Aware Trace Scheduling, we develop a PTX-to-PTX optimizer for fully automating profiling, analyzing, and optimizing PTX [29], NVIDIA's pseudo-assembly language. Subsequently, the graphics driver compiles PTX into executable binary code. Our PTX toolchain records dynamic program profiles, information to guide Warp-Aware Trace Scheduling. While this section describes PTX, our technology and ideas easily extend to HSA-based systems. HSAIL [13], a low-level intermediate language, functions similarly to PTX but supports a variety of processor architectures from HSA Foundation members AMD, ARM, Imagination, MediaTek, Qualcomm, Samsung, and Texas Instruments.

For profiling purposes, we implement our own instrumentation system rather than use the CUDA Profiling Tools Interface (CUPTI) [28]. While CUPTI is capable of collecting an extensive array of metrics, it can not collect each thread's behavior at every branch. Hence the need for our own instrumentation.

Our tool takes as input a GPU program, and as output, it produces an instrumented version of the program. Code destined for GPU execution is called the GPU *kernel*. In the instrumented version of the kernel, code is inserted before every conditional branch to collect each threads' branch behavior. The inserted code uses each thread's ID to update a unique counter for each branch direction. Consequently, there is one counter per branch per thread. Library calls to our tool are inserted before kernel invocations to allocate and initialize GPU memory for branch counters and after kernel invocations to compile the counters' results. Each branch's statistics are reduced to a single percentage, the probability of taking the branch. The probability is the number of times the branch was taken divided by the total number of times the branch was visited times 100.

PTX contains the following controlflow instructions: `@`, `call`, `ret`, `exit`, and `bra`. We avoid interprocedural analysis because after optimization for the Rodinia benchmarks, full inlining occurs and no `call` instructions exist. Regardless, our current implementation of Warp-Aware Trace Scheduling is intraprocedural. Since our profiler only instruments top-level functions, the instructions `ret` and `exit` behave identically, ending execution. All PTX instructions may conditionally execute based on a guard predicate, `@{!}p`. The `!` symbol is optional and performs logical negation. The `bra` instruction defines conditional and unconditional branching. In PTX, indirect branches are supported but unimplemented. However, indirect function calls are implemented. Conditional branches are formed by applying a guard predicate to an unconditional branch. For the purposes of this paper, a branch refers to a two-way conditional branch. The rest of the paper reflects this definition. Our optimizer effectively profiles the Rodinia benchmarks and reflects PTX ISA version 3.2 [29].

For efficient GPU global instruction scheduling, it is necessary to be aware of the GPU execution model and warp divergence (see Section 1). Warp-Aware Trace Scheduling minimizes execution time by exposing ILP while avoiding divergence and reducing divergence time. Generally, GPU execution proceeds as warps execute one common instruction at a time. However, when threads of a warp fail to identically evaluate a branch, lockstep warp execution halts. The warp's threads *diverge*; the warp splits and its threads

take different execution paths. Execution proceeds in this fashion until all the warp's threads *converge* again, and all the warp's threads continue in lockstep.

In addition to its tracing capabilities, our tool also performs analysis and optimization, representing GPU programs as a controlflow graph (CFG) and dataflow graph (DFG). A CFG node represents a basic block (BB), a single-entry single-exit code region. CFG edges are directed and represent potential paths of control flow. In a DFG, nodes represent instructions, and edges represent variable definitions (outgoing edges), and variable uses (incoming edges) through registers. The DFG does not contain edges for data flow through memory. Instrumenting, analyzing, and predicting occur automatically, only employing information available in the GPU program's source code.

3. MOTIVATION

Warp-Aware Trace Scheduling improves GPU resource utilization by increasing ILP [30] to decrease overall execution time. GPU utilization is measured by thread-level parallelism (TLP), the number of resident warps, and additionally, instruction-level parallelism (ILP). Table 1 summarizes the GPUs' historical technical specifications *per thread block* and *per streaming multiprocessor (SM)*. New GPU architectures contain more local hardware resources (Special Function Units, Instruction Issue, and Arithmetic Cores) exceeding the nominal growth in number of threads per SM. Beyond NVIDIA GPUs, this progression is also depicted in other companies' GPU hardware [1, 15].

While adjusting *occupancy*, the ratio of the number of resident warps to the maximum number of resident warps, to expose TLP is routine, no suitably obvious and automatic solution exists for increasing ILP. If these trends persist, ILP will have expanding performance implications. We propose to automatically and efficiently extract ILP using Warp-Aware Trace Scheduling.

ILP describes the potential overlap between executing instructions. On the GPU, SMs schedule instructions from ready warps of a block. At every instruction issue, an SM's warp scheduler selects a ready warp, if any, and issues the warp's next instruction to the warp's threads. The duration between instruction issue and completion is called *latency*. Achieving full utilization requires "hiding" all latency; at every clock cycle, all warp schedulers should have an instruction ready to issue.

4. GLOBAL INSTRUCTION SCHEDULING

Warp-Aware Trace Scheduling is a GPU-specific type of global instruction scheduling. Global instruction scheduling exposes ILP by grouping code, commonly basic blocks, into regions and scheduling instructions within each region, independently of other regions, to reduce execution time. Generally, global scheduling performs the following steps:

Step	Section	Description
1. Trace Selection	4.1	assign basic blocks to regions
2. Trace Formation	4.2	facilitate local scheduling, potentially adding nodes and edges
3. Local Scheduling	4.3	schedule instructions within each region

Specification	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
per Block								
Max Number of Threads	512				1024			
per SM								
Max Resident Blocks	8						16	
Max Resident Warps	24		32		48		64	
Max Shared Memory	16KB				48KB			
Number of 32-bit Registers	8K		16K		32K		64K	
Special Function Units	2				4	8	32	
Instruction Issue (per Clock Cycle)	0.25				1	2	8	
Arithmetic Cores	8				32		192	

Table 1: Historical streaming multiprocessor (SM) and thread block characteristics based on CUDA Compute Capability

Faraboschi *et al.* [8] provide a comprehensive survey of global instruction scheduling.

The following sections address our modifications to Trace Scheduling's basic steps to create Warp-Aware Trace Scheduling for GPUs. Global scheduling algorithms may use different definitions of region. Our scheduling algorithm defines region as a trace, pioneered by Fisher [9], and more extensively described by Ellis [7]. A trace is a set of basic blocks where each basic block is a predecessor of the next basic block, and the set is cycle free. The rest of the paper refers to regions as traces.

4.1 Trace Selection

Trace selection, the first step in global scheduling, groups sequences of frequently executed basic blocks into traces. Generally, given a weighted controlflow graph (CFG), trace selection chooses a *seed* node, a start node for the trace, and *grows* a trace, adding an adjacent node to the trace. All nodes are only visited once and the process repeats until all nodes are visited. The goal is to recognize paths of frequently executed basic blocks, identifying the program's critical path.

In contrast to prior work, trace selection for GPUs must adapt to the GPU-specific challenge of avoiding warp divergence. The challenge is creating traces defining the earliest non-divergent entry for scheduling instructions for speculative execution. Algorithms 1 and 2 present our GPU Trace Selection and Best Successor Of algorithms. Compared with prior work [8], our algorithms differ in two important respects: seed selection and trace growth to promote efficient local scheduling and divergence avoidance for GPUs.

Prior work selects traces by traversing CFG nodes or edges weighted by profile-generated execution frequency of basic blocks or branch paths respectively. Then, in prior work, trace selection *grows* traces *forward*, traversing the seed's child nodes, and *backward*, traversing the seed's parent nodes.

If divergence exists among frequently executed basic blocks, prior approaches to seeding and growing traces could make it worse, scheduling instructions from non-divergent

Algorithm 1 GPU Trace Selection Algorithm

```
1  function Trace Selection(Set nodes)
   /* nodes sorted by CFG tree-height and divergence. */
2      while there are unvisited nodes do
3          seed = next unvisited node    /* Select a seed */
4          Mark seed visited
5          current = seed
6          loop                          /* Grow trace forward */
7              s = Best Successor Of(current)
8              if s == 0 exit loop
9              Add s to the trace
10             Mark s visited
11         end loop
12     end while
13 end function
```

Algorithm 2 GPU Best Successor Of Algorithm

```
1  function Best Successor Of(Node src)
2      Edge e = the highest probability edge
3           among all src's outgoing edges
4      if Probability(e) ≤ MIN_PROB return 0
5      n = e's target
6      if n is visited return 0
7      return n
8  end function
9
10 function Probability(Edge e)
11     return e's weight
12 end function
```

basic blocks to divergent basic blocks. Instead, our algorithm begins by generating a list of seeds, nodes likely to be non-divergent. From these seeds the algorithm only grows forward, never backward. Therefore all trace roots are likely non-divergent. This assists local scheduling to avert divergence and reduce divergence time (see Section 4.3). Finally, we substitute dynamic profiling of program branch behavior instead of execution frequency because branch behavior provides additional insight differentiating basic block's warp behavior.

As mentioned previously (see Section 2), we developed a PTX optimizer for automatically profiling, analyzing, and optimizing PTX [29], CUDA's virtual instruction set architecture (ISA). Our PTX toolchain records dynamic program profiles to guide Warp-Aware Trace Scheduling.

Our GPU Trace Selection algorithm starts with a set of seed nodes, gathered using a breadth-first traversal on the CFG and sorted by the following properties:

- Control-equivalent to program entry node
- Loop head
- Contains PTX synchronization or communication instructions (*e.g.* `bar` and `membar`)†
- Control-equivalent to a basic block containing PTX synchronization or communication instructions

Ties are broken by most likely predicted successor. The result is a list sorted by breadth-first tree-height of likely non-divergent nodes. For reference, the definitions for dominance, post-dominance, and control-equivalence [2] are included below:

Definition (Dominance)**.** A node $\boldsymbol{A}$ dominates a node $\boldsymbol{B}$ if and only if all paths from the start node to $\boldsymbol{B}$ go through $\boldsymbol{A}$.

Definition (Post-Dominance)**.** A node $\boldsymbol{A}$ post-dominates a node $\boldsymbol{B}$ if and only if all paths from $\boldsymbol{B}$ to the exit node go through $\boldsymbol{A}$.

Definition (Control-Equivalence)**.** A node $\boldsymbol{A}$ is control-equivalent to a node $\boldsymbol{B}$ if $\boldsymbol{A}$ dominates $\boldsymbol{B}$ and $\boldsymbol{B}$ post-dominates $\boldsymbol{A}$.

After choosing a seed, GPU Trace Selection only grows traces forward. Warp-Aware Trace Scheduling's objective is to schedule instructions as early as possible and in non-divergent basic blocks. Traces with non-divergent roots represent this intent. We do not allow traces to grow backward because of the need to avoid divergence.

In prior work, Smith [33] also only grew traces forward, but in his algorithm the next unscheduled basic block is selected as the new seed, contrary to the general approach of identifying the next node by highest execution count. Our Trace Selection algorithm prioritizes non-divergent basic blocks, necessarily skipping basic blocks.

During trace growth, the GPU Best Successor Of algorithm (Algorithm 2) determines the most likely successor based on branch prediction. The MIN_PROB variable (line 4) is a cutoff criterion to stop trace growth when a likely successor cannot be determined since the branch prediction does not distinguish divergent and non-divergent branches, an additional reason for our deliberate approach to seed choice. Our results (see Section 6) evaluate the applicability of branch prediction without regard for warp behavior. Searching for a MIN_PROB value maximizing performance, MIN_PROB was experimentally determined to be 65.

The example CFG in Figure 1 contains three traces ($A \rightarrow B \rightarrow C$, $D \rightarrow E \rightarrow F$, and $G \rightarrow I$), four `if`-statements (A,B,C; D,E,F; J,K,L; and G,H,I), and one loop (D,E,F). The example contains one branch, J, affected by the MIN_PROB criterion. Nodes A, C, D, J, F, and G include conditional branches. Nodes B, E, K, and H end with unconditional branches. Nodes A, C, G, and I form a control-equivalent set. D is control-equivalent to F and J is control-equivalent to L. Note too, Trace #1 stops at C because D is a loop head and Trace #2 stops at F because traces cannot traverse a backedge (see definition of trace, Section 4). Node J does not start a trace because its branch fails the MIN_PROB criterion. While the example demonstrates the MIN_PROB criterion's effects on a branch off the critical path, a branch like J could just as likely lie on the critical path.

A reversal in a single branch's behavior can drastically affect trace selection. Suppose C's edge probabilities were reversed, $C \rightarrow D$ is predicted 5 and $C \rightarrow J$ is predicted 95. Three traces would be selected, $A \rightarrow B \rightarrow C \rightarrow J$, $D \rightarrow E \rightarrow F$, and $G \rightarrow I$. Longer traces tend to increase local scheduling opportunities.

4.2 Trace Formation

Compared to prior work, Warp-Aware Trace Scheduling forms traces most similar to Fisher's original proposal for

†instructions that assume uniform, non divergent warp behavior

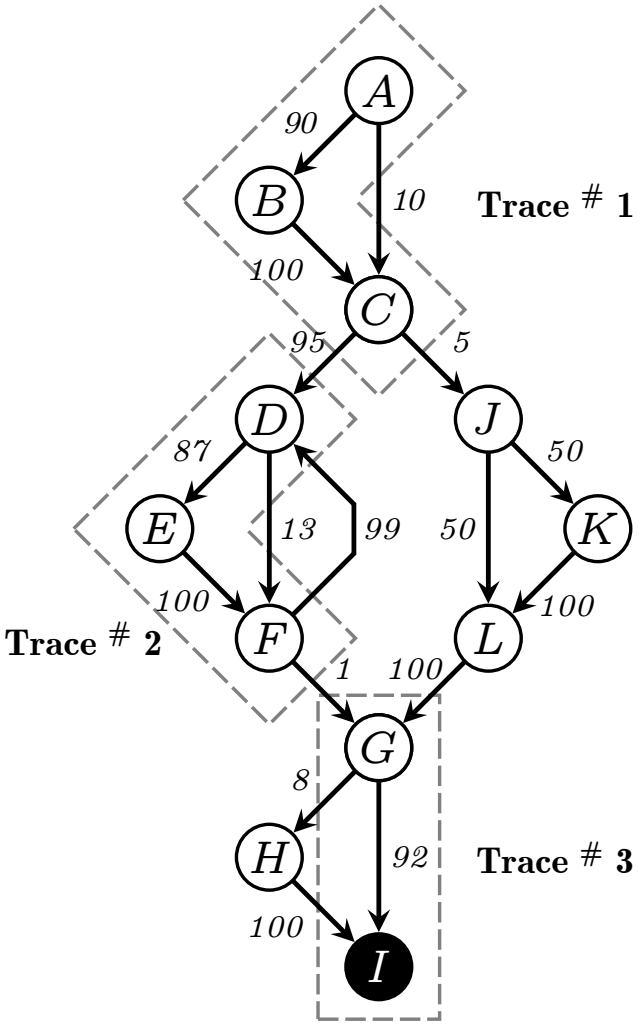

Figure 1: An example controlflow graph (CFG) with three traces: Trace #1 ($A \rightarrow B \rightarrow C$), Trace #2 ($D \rightarrow E \rightarrow F$), and Trace #3 ($G \rightarrow I$). Nominally, all single nodes, not already in a trace represent a trace (*e.g.* H).

CPUs [9]. However, unlike prior work, GPU trace formation respects program source code block structure, leaving it intact. All explicit attempts to form longer traces are conspicuously absent. However, contrary to Fisher, GPU traces are implicitly enlarged. After Warp-Aware Trace Scheduling, during the code's subsequent passage through the GPU compiler toolchain, predication is automatically performed. For PTX synchronization instructions, GPU trace formation adds extra edges to the trace to create scheduling constraints to preserve correct program behavior.

Many prior works, for example Superblock [14] and Hyperblock [22] scheduling, advanced structure-transforming innovations to decrease implementation effort of bookkeeping, expand traces, and enable optimization. Superblock scheduling forms longer traces by removing side entrances via tail duplication, and Hyperblock scheduling explicitly applies predication.

PTX code sequence *before*...

```
⋮                                ...
25          setp.*.*   %p0,   ...;
⋮                                ...
28   @%p0   bra        BB0;
29                               ...
⋮                                ...
39                               ...
40          BB0:
⋮                                ...
```

...and *after* `if`-conversion

```
⋮                                ...
25          setp.*.*   %p0,   ...;
29   @!%p0                       ...
⋮    @!%p0                       ...
39   @!%p0                       ...
⋮                                ...
```

Table 2: Example PTX code sequence showing the application of `if`-conversion using predication

After Warp-Aware Trace Scheduling, `ptxas` (or the CUDA device driver) compiles PTX to SASS [26]. SASS supports predication which `ptxas` employs for optimization, like `if`-conversion. To complement NVIDIA's toolchain, Warp-Aware Trace Scheduling ignores opportunities to form longer traces, relying on `ptxas` and later predicate-modifying optimizations (see Section 9 for how to potentially weaken dependence on NVIDIA's GPU toolchain to explicitly lengthen GPU traces).

GPU trace formation expects PTX code sequences, matching the pattern in the top half of Table 2, will be optimized to straightline code. The instruction on line 25 sets the branch's condition and on line 28, the branch to label (basic block) `BB0` is executed if predicate register `%p0` is `true`, skipping lines 29 through 39. The code sequence in the bottom half of Table 2 represents the above sequence after applying `if`-conversion, using predication. After `if`-conversion, line 28 is removed and code between lines 29 and 39 will conditionally execute based on `!%p0`. The `!` symbol performs logical negation.

For PTX synchronization instructions, GPU trace formation adds extra edges to the trace to create scheduling constraints to preserve correct program behavior.

4.3 Local Instruction Scheduling

Local scheduling, historically referred to as local compaction, moves instructions within but not between traces to improve ILP. Consider two instructions, ***A*** and ***B***, and ***B*** is a conditional branch. Moving ***A*** *above* ***B*** is called *upward code motion* and the opposite transformation, moving ***A*** *below* ***B***, is called *downward code motion*. Dependences define execution-order constraints between instructions. During upward code motion, an instruction breaks its control dependence on the preceding branch. If the instruction's new basic block has multiple successors, this motion is

	Restricted [4]	General [4]	Boosting [34]	**Deviant (ours)**
Scheduling Restrictions	*Legal* and *Safe*	*Legal*	none	excludes texture, shared, and constant memory operations and all `store` instructions
Hardware Support	none	non-trapping instructions	shadow register file, shadow store buffer, and support for re-executing instructions	none
Exception Handling for Speculative Instructions	prohibited	ignored	supported	absent

Table 3: Characterizing global scheduling models from prior work and our new GPU model (found on the far right). Most GPUs contain limited exception handling capabilities (shown in more detail in Table 4); The GPU provides no mechanism to detect exceptions.

Exception	**Support**
Arithmetic	
Integer Floating-Point	No mechanism to detect occurrence[‡]
Memory	
Texture Constant Shared Global	CUDA runtime error "unknown error" at conclusion of kernel execution, available on host device (CPU)

Table 4: GPU exception handling by instruction type (for NVIDIA GPUs)

speculative because it is uncertain whether the instruction's result will be needed. The following restrictions govern code motion:

Legal – Preserve sequential correctness
Safe – Prevent any exception caused by speculative execution from terminating execution

Prior work enforced exception handling to varying degrees with hardware and software support. Table 3 describes the general scheduling models' policies and enforcement strategies to obey the aforementioned restrictions. The rightmost column in Table 3 summarizes our new GPU scheduling model. In general, the GPU is *unsafe* (see CUDA C Programming Guide [27], Section F.2). On the GPU, arithmetic exceptions, like integer and floating-point division by zero, underflow, and overflow, are not signaled, no mechanism exists to detect the exception, and execution continues normally. Additionally, as Table 3 and Table 4 indicate, all GPU memory operations are *unsafe*, speculatively executing a memory operation can cause an error and may be reported only after kernel execution concludes. Warp-Aware Trace Scheduling conforms to the CUDA Programming Model's exception handling policy. We say the GPU global scheduling model is **deviant** because GPU execution continues even if errors occur, failing to conform to (*deviating* from) prior scheduling models.

Because GPUs have such weak exception handling mechanisms, our scheduler only speculates on global `load`'s and arithmetic instructions and prohibits speculation of all `store` instructions and texture, shared, and constant memory operations. Looking forward, we recommend adding new instructions, a *speculative* `load` and an *exception* `check`, providing a *safe* and practical solution to govern data speculation, similar to instructions (`PREFETCH`h) used by current Intel 64 architectures [16]. A *speculative* `load` always succeeds, never causes a page fault, and sets a *valid* flag. *exception* `check` tests the *valid* flag.

Warp-Aware Trace Scheduling's local trace scheduler performs *no scheduling on instructions within basic blocks*. It *only* performs *scheduling on instructions between basic blocks*. During PTX-to-SASS translation, `ptxas` performs register allocation and scheduling within basic blocks. When scheduling instructions between basic blocks, Warp-Aware Trace Scheduling's local trace scheduler only applies *upward code motion*, removing instructions from the top of one basic block and appending them to the bottom of their new basic block. The local trace scheduler never moves an instruction down, not even an instruction in its native basic block. Remember the details of GPU trace selection in Section 4.1. By construction, trace roots are the earliest, likely non-divergent basic block to schedule into. The scheduler greedily schedules instructions to minimize execution time on the critical path, so it is possible that at scheduling's conclusion some basic block schedules have been lengthened. The main objective is to increase ILP and additionally, separate dependent global memory `load`'s and `store`'s, scheduling `load`'s as early as possible, because generally, `load`'s and `store`'s respectively begin and end chains of dependent instructions.

4.3.1 Bookkeeping

To maintain program correctness (*legality*) during scheduling, it may be necessary to insert copies of moved instructions, a process called *bookkeeping*, also known as *compensation*. The instruction copies are referred to as compensation code. Due to GPU-specific warp behavior, Warp-Aware Trace Scheduling's bookkeeping is tightly controlled

[‡]CUDA C Programming Guide [27], Section G.2. Floating-Point Standard

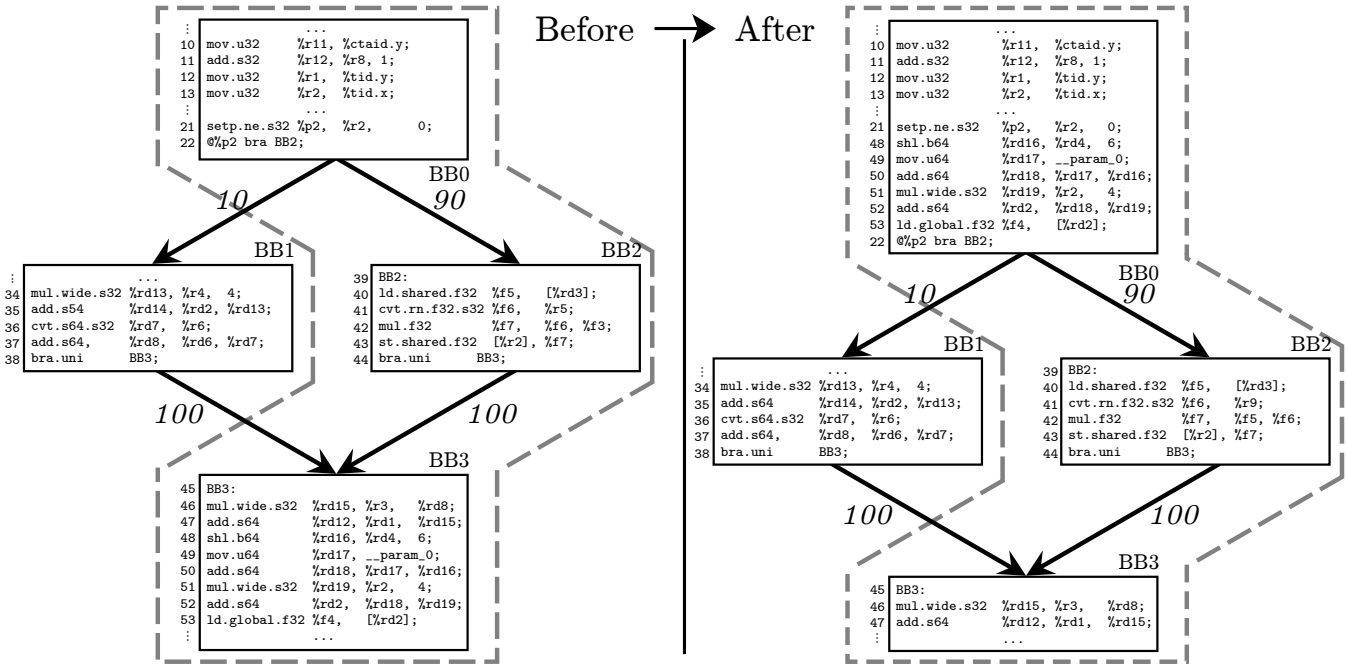

Figure 2: Example local scheduling of an `if-else`-statement. The `if` is predicted likely *not taken*. Consequently, during local scheduling, instructions from BB3 (lines 48 – 53) are *moved upward* into BB1. After local scheduling note line 53, a global memory `load`, will be speculatively executed.

to avoid inserting instruction copies into divergent basic blocks, thereby reducing divergence time.

Typically, compensation is required to correct misspeculation. However, Warp-Aware Trace Scheduling only speculates global `load`'s, never `store`'s, and maintains original basic block boundaries and order of conditional branches. Consequently, Warp-Aware Trace Scheduling is *recovery-free* (*i.e.* it requires no support for misspeculation to preserve program correctness).

Starting from the trace root's successors, the local trace scheduler, respecting data dependences, attempts upward code motion to schedule instructions on the trace from root to leaf, preferring to schedule in the trace's root to potentially achieve earliest possible execution. If an instruction cannot be scheduled in the root and another convergent basic block cannot be found, the scheduler begins to schedule the next instruction. Loops are scheduled independently. The scheduler operates without a model of instruction latencies. Consequently, our approach greedily schedules based on branch prediction, and as previously mentioned can produce unbalanced basic blocks, preferring to schedule instructions to minimize critical path execution. The result is a scheduler that attempts to minimize overall execution time and divergence time, without inducing additional warp divergence or increasing divergence time.

Figure 2 demonstrates warp-aware local scheduling on an `if-else`-statement. The example contains one trace: BB0 → BB2 → BB3. The figure's left side depicts the original code sequence, *before* local scheduling, and the right side depicts the optimized code sequence, *after* local scheduling. No instructions are moved from BB2 because these instructions (lines 41–43) depend on the shared `load` instruction on line 40. The scheduler only speculates on global `load`'s. Most of BB3's instructions (lines 48–53) are *moved upward* into basic block BB1. However, instructions on line 46 and 47 remain because line 47 depends on line 46, which in turn depends on line 37 (register `%rd8`). While it is possible to move the instructions on lines 46 and 47 to BB1, local scheduling is not optimized to avoid increasing execution time. The scheduler considers CFG structure and branch profiling to identify divergence and avoid it as well as increasing divergence time. In this instance, considering BB0 → BB1's edge weight (*10*), the scheduler determines that BB1 is a divergent basic block and prevents scheduling extra instructions into it. If the scheduler determined BB1 non-divergent, the scheduler would schedule instructions on lines 46 and 47 to both BB1 and BB2.

5. EXPERIMENTAL METHODOLOGY

We evaluated Warp-Aware Trace Scheduling using the Rodinia Benchmark Suite [5]. The benchmarks were compiled using CUDA 5.5 with full optimization (O3), targeted for a compute capability 3.0 device (sm_30) using the NVIDIA driver version 319.37, the driver version accompanying CUDA 5.5. Programs were profiled, analyzed, and optimized using the optimizer we developed and previously discussed (see Section 4.1). All results are from execution on an NVIDIA GTX 670 (a compute capability 3.0 device), not simulation.

Results for each benchmark were measured as an average of at least 10 runs and reflect kernel, not whole program, performance. Different input sets were used for training and testing. Besides performance, the results before and after our optimization are identical. While current GPUs contain weak exception handling capabilities, Warp-Aware Trace Scheduling maintains sequential correctness (*i.e.* legal-

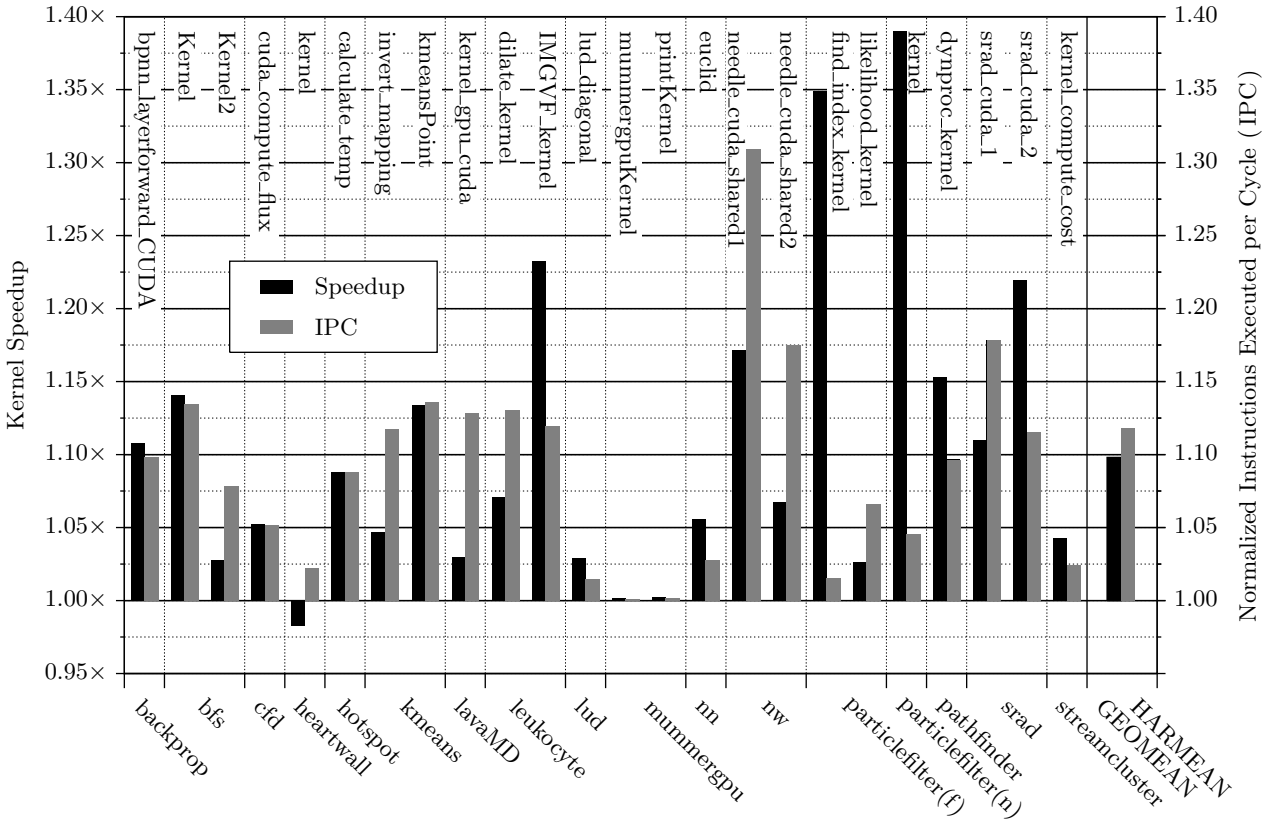

Figure 3: Kernel speedup and normalized instructions executed per cycle (IPC) with Warp-Aware Trace Scheduling on the Rodinia benchmarks. Geometric mean speedup and harmonic mean IPC are found on the right.

ity, see Section 4.3). Our technique only speculates global `loads`, never `stores` (see Section 4.3).

6. RESULTS

Figure 3 demonstrates the performance benefit of Warp-Aware Trace Scheduling applied to the Rodinia Benchmark Suite on a real system. On Figure 3's dual y-axis, kernel speedup is on the left and normalized instructions executed per cycle (IPC)[§] is on the right. IPC results were recorded using NVIDIA's profiler and represent relative IPC after Warp-Aware Trace Scheduling over the baseline (original) IPC. The bottom x-axis shows benchmark names and on the top x-axis, the corresponding names of the benchmark's kernels.

Shown on the far right of Figure 3, across the Rodinia benchmarks, the geometric mean speedup is 1.10×. In terms of speedup, notable standouts include **particlefilter(n)**'s *kernel*, 1.34×; **particlefilter(f)**'s *find_index_kernel*, 1.35×; **leukocyte**'s *IMGVF_kernel*, 1.23×; and **srad**'s *srad_cuda_2*, 1.22×. **heartwall** produces the only negative speedup performance mark at 0.98×.

Comparing speedup to IPC, the results are markedly correlated. Intuitively, identifying and improving IPC on critical program paths can reduce overall execution time. On the far right, across the Rodinia benchmarks, the IPC's harmonic mean is 1.12×.

6.1 Performance Analysis

If Warp-Aware Trace Scheduling automatically identifies no opportunities, it leaves the kernel unchanged. Specifically, Warp-Aware Trace Scheduling dismisses kernels if:

- there are too few (or no) branches to speculate
- trace selection fails; unpredictable critical path
- the program contains insufficient ILP

Intuitively, if no branches exist in a program, there are no basic block boundaries to schedule across, and global scheduling will have marginal performance benefit. In our evaluation, the branches in Rodinia's kernels are mostly predictable, containing a majority of branches with lopsided ratios (*e.g.* 90 – 10). Based on our trace selection's MIN_PROB criterion (see Section 4.1), too many 50 – 50 branches, or other suitably balanced branches, can thwart identification of program critical path and trace generation. Finally, if the program contains insufficient ILP to expose, again, global instruction scheduling will be ineffective.

We inspected SASS to affirm our technique's results and identify and understand related `ptxas` optimizations. Scrutinizing the differences in SASS before and after applying our optimization we concluded our optimization had successfully identified program's critical path and divergent basic

[§]the number of executed instructions divided by the number of SMs divided by elapsed clocks

blocks and hoisted instructions above branches. Since SASS is not executable binary microcode but rather just another (virtual) low-level assembly language, more thorough analysis was done using NVIDIA's profiler, collecting statistics from hardware counters. Analyzing complementary hardware counters reported by NVIDIA's profiler suggests modest increases to cache misses and instruction replay overhead in order to expose additional ILP and increase IPC is a desirable tradeoff to achieve higher overall performance.

Warp-Aware Trace Scheduling is applied before `ptxas` optimization and register allocation. On average the total number of used registers increased by 4. In particular, kernels in **lud**, **nw**, and **srad** showed the greatest increases in number of used registers by 17, 16, and 14, respectively but these increases did not affect occupancy (see Section 3).

6.2 IPC Analysis

Note that increasing IPC (instruction throughput) does not always result in a corresponding improvement in execution time. For instance, notice the difference between IPC and speedup for **nw**'s *needle_cuda_shared1* (see Figure 3). Increasing IPC may also increase the percentage of replayed instructions, a negative performance metric. The percentage of replayed instructions is calculated as:

$$\left(\frac{\text{instructions issued} - \text{instructions executed}}{\text{instructions issued}}\right) \times 100,$$

the difference between the number of instructions issued by the number of instructions executed, and:

$$\text{instructions issued} \geq \text{instructions executed}.$$

Replay is a hardware optimization to hide instruction latency. When a variable, long-latency instruction, such as a `ld/st` instruction stalls the pipeline, the stalling warp is removed from the pipeline and the pipeline starts executing ready instructions from a previously waiting warp. Later, the original stalling instructions will be reissued and hopefully, execute faster. A replay could occur due to a local cache miss, memory bank conflict, and non-uniform memory access.

To further investigate and identify the source of performance improvements, for each benchmark we used NVIDIA's profiler to gather global memory replay overhead, shared memory replay overhead, and instruction replay overhead. For benchmarks that showed positive performance improvements, no correlation between either replay overhead and IPC, or replay overhead and speedup was found. These results reinforce our hypothesis that IPC can substitute for TLP (*e.g.* occupancy), to improve GPU resource utilization and attain higher performance, even at the expense of increases to negative performance metrics.

6.3 Summary

Warp-Aware Trace Scheduling, like prior global scheduling techniques, primarily focuses on reducing execution time by increasing ILP (IPC) on the critical path. However, its unique features to avoid divergence and reduce divergence time contribute to improved performance. Targeting and reducing divergence time achieves reductions in negative performance metrics, such as instruction serialization and total instructions executed. The next section describes Figure 3's results, focusing on increasing ILP, avoiding divergence, and reducing divergence time for faster program execution.

7. DISCUSSION

This section studiously interprets some benchmarks' performance results reported in the previous section. First, Table 5 reflects on benchmarks' kernels not amenable to Warp-Aware Trace Scheduling. To begin, Warp-Aware Trace Scheduling disregards kernels that contain limited or no branches, no control flow. **backprop**'s *bpnn_adjust_weights_cuda* only contains one branch. Dynamic profiling indicates it is taken and immediately encounters an `exit` instruction. Consequently, global scheduling would be ineffective.

In **leukocyte**'s *GICOV_kernel*, few branches exist and each basic block's arithmetic intensity, the ratio of arithmetic instructions to off-chip memory instructions, is high. Consequently, increasing ILP will have only marginal performance benefit because sufficient ILP exists within each basic block. **lud**'s *lud_internal* contains no branches. **lud**'s second kernel, *lud_perimeter*, contains few lopsided (*e.g.* $90-10$) branches and many even (*e.g.* $50-50$) branches. Attempting to select expansive traces in *lud_perimeter* fails.

In **particlefilter(f)**, *normalize_weights_kernel* has a sufficient number of branches and its critical path is clearly defined. However, its critical path skips an extreme amount of the program; most often, the kernel exits without performing much computation.

For **particlefilter(f)**'s other kernel, *sum_kernel*, intense computation is only found within its single loop. Warp-Aware Trace Scheduling restricts upward code motion to within traces, never between traces. It is likely *sum_kernel* would benefit from loop unrolling, a complementary optimization to trace scheduling (for further details, see Section 9).

Returning to Figure 3, these kernels generally contain predictable branches, enable selection and formation of long traces, and contain sufficient ILP to effectively employ global scheduling.

Among Rodinia, **cfd**'s *cuda_compute_flux* is unique in its use of the `unroll` pragma. This pragma tells the compiler to replace a loop with a repeated sequence of the loop's instructions, with fewer branch instructions between. For *cuda_compute_flux* after unrolling, no loops, no backedges, remain, allowing Warp-Aware Trace Scheduling to select a single long trace from the kernel.

Only **heartwall**'s *kernel* produces negative speedup. Note the fractional increase in IPC. Here, while Warp-Aware Trace Scheduling was effective uncovering ILP, slowdown was produced because our heuristics to avoid divergence failed, scheduling extra instructions to divergent basic blocks. Considering the ILP exposed and our conservative approach to speculation, reversing *kernel*'s negative performance may still not achieve distinct performance benefits. For **heartwall**, the overriding performance problem is not limited ILP but rather low global memory bandwidth due to inefficient memory access patterns.

A marked contrast exists between IPC and speedup for **lavaMD**'s *kernel_gpu_cuda*. For *kernel_gpu_cuda*, Warp-Aware Trace Scheduling fails to appreciably improve execution time because loops are scheduled independently and *kernel_gpu_cuda*'s global memory instructions are isolated in two tight loops.

Both **mummergpu**'s kernels are problematic for global scheduling. *printKernel*'s execution time is dominated by one large loop containing numerous divergent branches with

Benchmark	Kernel	Number of branches	Critical Path (MIN_PROB)	Arithmetic Intensity
backprop	bpnn_adjust_weights_cuda	1	✗	✗
cfd	cuda_initialize_variables	∅	✓	✓
	cuda_time_step	∅	✓	✓
	cuda_compute_step_factor	∅	✓	✓
gaussian	Fan1	1	✓	✓
	Fan2	✓	✗	✗
leukocyte	GICOV_kernel	✗	✓	✓
lud	lud_perimeter	✓	✗	✓
	lud_internal	∅	✓	✓
particlefilter(f)	normalize_weights_kernel	✓	✓	✗
	sum_kernel	✗	✓	✗

Table 5: Rodinia benchmarks' kernels for which Warp-Aware Trace Scheduling identified insufficient ILP opportunities to justify application of global scheduling techniques

few global `load`'s to speculate. With safer data speculation support, like the solution proposed in Section 4.1, more aggressive speculation could enable higher performance. *mummergpuKernel*'s problems only differ in that it contains nested loops. A loop optimization, or the combination of global scheduling and loop optimization, may produce better results.

Warp-Aware Trace Scheduling achieves impressive performance results for **particlefilter(n)**'s *kernel* and **particlefilter(f)**'s *find_index_kernel*. But the increase in IPC, an indicator of the degree of ILP, is minimal. In these kernels, Warp-Aware Trace Scheduling's results are attributable to reduced divergence time and scheduling instructions from divergent basic blocks to non-divergent basic blocks. Comparing the original and optimized kernels using NVIDIA's profiler shows a marked reduction in instruction serialization and total instructions executed.

8. RELATED WORK

Trace Scheduling, initially proposed by Fisher [9], now dominates current approaches to global instruction scheduling. Works related to Fisher's Trace Scheduling mostly vary regarding trace shape and schedule construction. For further information, Faraboschi *et al.* [8] survey global instruction scheduling and region shapes.

Specifically, much related work pertains to enhancing ILP on VLIW/EPIC (IA-64) architectures. These architectures employ dynamic hardware branch prediction as well as hardware parallelism detection and extraction techniques. GPUs are simple in-order SIMT (Single-Instruction Multiple-Thread) architectures without hardware branch prediction and analogous hardware parallelism detection and extraction. Without enhancement, Trace Scheduling could not successfully navigate between the GPU architecture's pitfalls and challenges. Warp-Aware Trace Scheduling emits performance by adapting global scheduling to the GPU execution model and avoiding divergence.

Regarding static GPU controlflow analyses to enable optimization, Coutinho *et al.* [6] propose a static branch divergence analysis. Their analysis could complement but not substitute for dynamic profiling because Warp-Aware Trace Scheduling requires accurate determination of program critical path, not just differentiation of divergent and non-divergent instructions.

Various software-only GPU-specific optimizations generally target divergence. Branch fusion, proposed by Coutinho *et al.* [6], merges common code from divergent program paths using their aforementioned static branch divergence analysis. Han *et al.* [12] describe two optimizations: iteration delaying and branch distribution. Iteration delaying improves performance by targeting a divergent branch within a loop, executing iterations following the branch and delaying execution of iterations that do not follow the branch until later. Hopefully, the delayed iterations execute with more threads, achieving higher resource utilization. Branch distribution functions similarly to branch fusion.

9. FUTURE WORK

We plan to investigate the following strategies to expose additional ILP for higher performance: trace expansion, local scheduling, and misspeculation support. Warp-Aware Trace Scheduling could benefit from trace-enlarging techniques, like Superblock scheduling's tail duplication [14] and Hyperblock scheduling's predication [22]. If loops dominate runtime, potential loop optimizations include loop unrolling and Software Pipelining [21].

Regarding local scheduling, our current approach could be augmented with accurate modeling of instruction latencies. Further, alternative list scheduling heuristics could produce more efficient schedules. Our local scheduler only applies *upward* code motion, but precedent suggests *downward* code motion could provide additional performance benefits.

Finally, with precise exceptions and efficient misspeculation support, more aggressive speculation is possible. Section 4.3 describes adding new *speculative* `load` and *exception* `check` instructions. Currently, only a fraction of potential, long latency, memory instructions are speculated and no `store` instructions since current GPUs provide no hardware support for handling exceptions.

Using ILP rather than TLP to support effective latency hiding could lead to additional optimizations, trading kernel characteristics, like occupancy, for higher performance. For instance, register pressure can have negative performance impact. Substituting ILP for TLP could reduce register

pressure without exceeding the warp scheduler's ability to hide long-latency memory instructions.

Potentially, with divergence analysis like the analysis proposed by Coutinho *et al.* [6], dependence on trace selection's MIN_PROB criterion could be reduced or eliminated. Local scheduling could also benefit from more accurate identification of divergence.

Discussed in more detail in Section 4.1, traces are identified using dynamic profiling. Alternatively, trace selection could use static branch prediction heuristics [3]. However, no study exists demonstrating static GPU branch prediction to be suitable, sufficient in branch coverage and accuracy, for enabling optimization. Additionally, our current profiling approach could be enhanced to allow application of Warp-Aware Trace Scheduling during just-in-time (JIT) compilation. Currently, our profiling approach's overhead makes this impractical.

10. CONCLUSION

This paper presented a detailed implementation and evaluation of Warp-Aware Trace Scheduling, the first fully-automatic speculative global scheduling optimization for GPUs. We revisited and revising Trace Scheduling's three major steps for the GPU architecture. First, during trace selection, the kernel is partitioned into traces with likely *non-divergent* roots. Next, in trace formation, our implementation adds edges to synchronization instructions to constrain scheduling and preserve correct program behavior. Finally, instructions within traces are scheduled to improve ILP and hide global memory latency, if possible, breaking control dependences and speculating on instruction execution. Furthermore, our local scheduling effectively avoids divergence and reduces divergence time.

Our evaluation of Warp-Aware Trace Scheduling on a real GPU system demonstrates significant performance improvements in execution time and IPC for 17 benchmarks and 24 kernels. Overall, Warp-Aware Trace Scheduling automatically achieves a geometric mean 1.10× speedup using dynamic profiling, improving IPC by 1.12× (harmonic mean), and reducing instruction serialization and total instructions executed. These current results reinforce our implementation approach and basic concepts for our global instruction scheduling technique. Executing our plans for future work, composing additional optimizations with Warp-Aware Trace Scheduling, we anticipate more higher performance improvements.

11. ACKNOWLEDGMENTS

James A. Jablin is supported by the Department of Energy Office of Science Graduate Fellowship Program (DOE SCGF), made possible in part by the American Recovery and Reinvestment Act of 2009, administered by ORISE-ORAU under contract no. DE-AC05-06OR23100.

We acknowledge the support of Intel Science and Technology Center for Cloud Computing.

12. REFERENCES

[1] Advanced Micro Devices, Inc. *AMD Graphics Cores Next (GCN) Architecture*, 2012.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] T. Ball and J. R. Larus. Branch Prediction for Free. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 300–313, New York, NY, USA, 1993. ACM.

[4] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-m. W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In *Proceedings of the 18th annual international symposium on Computer architecture*, ISCA '91, pages 266–275, New York, NY, USA, 1991. ACM.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pages 44–54, October 2009.

[6] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr. Divergence Analysis and Optimizations. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 320–329, Washington, DC, USA, 2011. IEEE Computer Society.

[7] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures (Parallel Computing, Reduced-Instruction-Set, Trace Scheduling, Scientific)*. PhD thesis, New Haven, CT, USA, 1985.

[8] P. Faraboschi, J. Fisher, and C. Young. Instruction Scheduling for Instruction Level Parallel Processors. *Proceedings of the IEEE*, 89(11):1638–1659, 2001.

[9] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981.

[10] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.

[11] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[12] T. D. Han and T. S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[13] HSA Foundation. *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)*, May 2013.

[14] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar

Compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993.

[15] Intel Corporation. *Performance Interactions of OpenCL Code and Intel® - Quick Sync Video on Intel HD Graphics 4000*, 2012.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2013.

[17] A. Jog, O. Kayıran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406, New York, NY, USA, 2013. ACM.

[18] A. Jog, O. Kayıran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[19] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 157–166, Piscataway, NJ, USA, 2013. IEEE Press.

[20] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[21] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM.

[22] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[23] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM.

[24] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[26] NVIDIA Corporation. *CUDA Binary Utilities*, July 2013.

[27] NVIDIA Corporation. *CUDA C Programming Guide*, July 2013.

[28] NVIDIA Corporation. *CUPTI*, July 2013.

[29] NVIDIA Corporation. *Parallel Thread Execution ISA v3.2*, July 2013.

[30] NVIDIA Corporation. *Tuning CUDA Applications For Kepler*, July 2013.

[31] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[32] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[33] M. D. Smith. Support for Speculative Execution in High-Performance Processors. Technical report, Stanford, CA, USA, 1992.

[34] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 344–354, New York, NY, USA, 1990. ACM.