2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures

Yunsup Lee*, Vinod Grover†, Ronny Krashinsky†, Mark Stephenson†, Stephen W. Keckler†‡, Krste Asanović*
*University of California, Berkeley, †NVIDIA, ‡The University of Texas at Austin
{yunsup, krste}@eecs.berkeley.edu, {vgrover, rkrashinsky, mstephenson, skeckler}@nvidia.com

***Abstract*—Data-parallel architectures must provide efficient support for complex control-flow constructs to support sophisticated applications coded in modern single-program multiple-data languages. As these architectures have wide datapaths that process a single instruction across parallel threads, a mechanism is needed to track and sequence threads as they traverse potentially divergent control paths through the program. The design space for divergence management ranges from software-only approaches where divergence is explicitly managed by the compiler, to hardware solutions where divergence is managed implicitly by the microarchitecture. In this paper, we explore this space and propose a new predication-based approach for handling control-flow structures in data-parallel architectures. Unlike prior predication algorithms, our new compiler analyses and hardware instructions consider the commonality of predication conditions across threads to improve efficiency. We prototype our algorithms in a production compiler and evaluate the tradeoffs between software and hardware divergence management on current GPU silicon. We show that our compiler algorithms make a predication-only architecture competitive in performance to one with hardware support for tracking divergence.**

I. Introduction

Data-parallel architectures such as Cray-1-like vector machines [14, 29], Intel packed-SIMD units [17, 28], NVIDIA and AMD graphics processing units [2, 16, 23, 24], and Intel MIC accelerators [30] have long been known to provide greater performance and energy efficiency than general-purpose architectures for applications with abundant data-level parallelism. Programming these systems is inherently challenging, and over decades of research and development only a few models have attained broad success. Implicitly parallel autovectorization approaches were popular with early vector machines, while explicitly parallel *Single-Program Multiple-Data* (SPMD) accelerator languages like CUDA [21] and OpenCL [26] have proven to be accessible and productive for newer GPUs and SIMD extensions. By nature, SPMD programs tend to have substantial and complex per-thread control flow, extending beyond simple if-then-else clauses to nested loops and function calls.

To achieve an efficient mapping, threads are processed together in SIMD vectors, but orchestrating the execution of SPMD threads on wide SIMD datapaths is challenging. As each thread executes control flow independently, execution may *diverge* at control conditions such as if-then-else statements, loops, and function calls. The architecture must therefore track and sequence through the various control paths taken through the program by the different elements in a vector. This is generally done by selectively enabling a subset of threads in a vector while each control path is traversed. Because divergence leads to a loss of efficiency, *reconvergence* is another important component of divergence management on data-parallel architectures.

As described in Section II, NVIDIA GPUs support the SPMD model directly in hardware with a thread-level hardware ISA that includes thread-level branch instructions [9, 16]. This approach allows the compiler to use a fairly conventional thread compilation model, while pushing most of the divergence management burden onto hardware. Threads are grouped into 32-element vectors called *warps*, and when threads in a warp branch in different directions, the hardware chooses one path to continue executing while deferring the other path by pushing its program counter and thread mask onto a specialized *divergence stack*. Reconvergence is also managed through tokens pushed and popped on this stack by the compiler.

Vector-style architectures with a scalar+vector ISA employ a compiler-driven approach to divergence management [1, 12, 13, 27, 32, 33]. In this model, the vector unit cannot execute branch instructions. Instead, the compiler must explicitly use scalar branches to sequence through the various control paths for the elements or threads in a vector, while using vector predication to selectively enable or disable vector elements.

Although a wide range of software and hardware SPMD divergence management schemes are implemented in the field, software divergence management in particular has received relatively little attention from the academic research community. At first glance the topic may seem like a recasting of classic vectorization and predication issues, but the challenges are unique in the context of modern architecture for several reasons: (1) Unlike traditional vectorization, the parallelization of arbitrary thread programs is a functional requirement rather than an optional performance optimization. (2) The divergence management architecture must not only partially sequence all execution paths for correctness, but also reconverge threads from different execution paths for efficiency. (3) Traditional compiler algorithms for predication in serial processors are *thread-agnostic*, as they only need to consider optimizing the control flow for a single thread of execution. A data-parallel architecture on the other hand requires different *thread-aware* performance considerations. (4) GPUs and other multithreaded processors with a shared register pool are particularly sensitive to register pressure, as register count determines the number of threads that can execute concurrently. This constraint results in unique tradeoffs and optimization opportunities for the divergence management architecture. Finally (5), in SPMD programs, uniform control and data operations can be *scalarized* to improve efficiency, a challenge that is related to but different than vectorization [5, 12, 15].

In the remainder of this paper, we further describe and analyze the design-space, tradeoffs, and unique challenges of SPMD divergence management architectures. In Section III we detail our thread-aware predication-based compiler algorithm for SPMD divergence management. We have developed optimizations including a *static branch-uniformity optimization* and a compiler-instigated *runtime branch-uniformity optimization* that eliminates unnecessary fetch and issue of predicate-false instructions. As described in Section IV, we use these algorithms to modify an NVIDIA production compiler to only use predication and uniform branches, eliminating all use of the hardware divergence stack.

In Section V, we first characterize the control flow of a wide range of data-parallel applications. We then compare and analyze in detail the performance characteristics of software-based and hardware-based divergence management architectures on production GPU silicon. We describe conditions where software predication performs better, and other conditions where the hardware divergence stack performs better. Finally, we discuss the tradeoffs and suggest promising areas of future work for further optimization of our software divergence management implementation in Section VI.

II. Background

This section provides an overview of how different data-parallel architectures support the control flow present in the SPMD programming model, where data-parallel computation is expressed in the form of multithreaded *kernels*. The programmer writes code both for a single thread and for an explicit kernel invocation that directs a group of threads to execute the kernel code in parallel. We explain next how different data-parallel architectures manage divergence with the if-then-else statement and the loop example shown in Figure 1. Due to the SIMD execution nature of these architectures, the hardware must provide a mechanism to execute an instruction only for selected elements within a vector. Some architectures expose predication to the compiler, while others hide it from the compiler and manage divergence implicitly in hardware.

A. Vector Machines

Compilers for vector machines manage divergence explicitly. Figure 2 illustrates how a vector machine would typically handle the control flow shown in Figure 1. The example shows a mixture of scalar and vector instructions, along with scalar and vector registers. To execute both sides of the if-then-else statement conditionally, the vector machine first writes the result of the `vslt` compare instruction into a vector predicate register `vf0`, then conditionally executes `vop2` under `vf0`. Similarly, `vop3` is executed under the negated condition `!vf0`. Vector register `vc` is partially updated from both sides of the execution paths. The diverged execution paths merge at the immediate post-dominator, where `vop4` is executed. The compiler statically encodes this information by emitting `vop4` under no predicate condition.

```
void kernel() {
  a = op0;
  b = op1;
  if (a < b) {
    c = op2;
  } else {
    c = op3;
  }
  d = op4;
}
```

(a) If-then-else statement

```
void kernel() {
  bool done = false;
  while (!done) {
    a = op0;
    b = op1;
    done = a < b;
  }
  c = op2;
}
```

(b) Loop

Figure 1. Control flow examples (kernel invocations omitted).

Several optimizations such as density-time execution and compress-expand transformations have been proposed [33] and evaluated [14] to save execution time of sparsely activated vector instructions. However, these optimizations cannot prevent vector instructions with an all-false predicate mask from being fetched and decoded. The compiler can optionally insert a check to test whether the predicate condition is null, meaning that instructions under that predicate condition are unnecessary. In Figure 2a, both conditionally executed paths are guarded with a dynamic check. A `vpopcnt` instruction, which writes a count of all `true` conditions in a vector predicate register to a scalar register, is used to count active elements. A scalar branch (`branch.eqz` instruction) checks whether the count is zero to jump around unnecessary work. These checks may not always turn out to be profitable, as the condition could truly be unbiased and it would be better to schedule both sides of the execution paths simultaneously.

Figure 2b shows how loops in SPMD programs are mapped to vector machines. Loop mask `vf0`, which keeps track of active elements executing the loop, is initialized to `true`. A `vpopcnt` instruction is combined with a branch-if-equals-zero instruction to test whether all elements have exited the loop. All instructions in the loop body

```
      va = vop0
      vb = vop1
      vf0 = vslt va,vb
      s0 = vpopcnt vf0
      branch.eqz s0,else
 @vf0 vc = vop2
else:
      s0 = vpopcnt !vf0
      branch.eqz s0,ipdom
!@vf0 vc = vop3
ipdom:
      vd = vop4
```

(a) If-then-else statement

```
      vf0 = true
loop:
      s0 = vpopcnt vf0
      branch.eqz s0,exit
@vf0  va = vop0
@vf0  vb = vop1
@vf0  vf1 = vslt va,vb
      vf0 = vand vf0,!vf1
      j loop
exit:
      vc = vop2
```

(b) Loop

Figure 2. Divergence management on vector architectures.

```
    a = op0
    b = op1
    p = slt a,b
    push.stack reconverge
    tbranch.eqz p,else
    c = op2
    pop.stack
else:
    c = op3
    pop.stack
reconverge:
    d = op4
```

(a) If-then-else statement

```
    done = false
    push.stack reconverge
loop:
    tbranch.neqz done,exit
    a = op0
    b = op1
    done = slt a,b
    j loop
exit:
    pop.stack
reconverge:
    c = op2
```

(b) Loop

Figure 3. Hardware divergence management on NVIDIA GPUs.

(`vop0`, `vop1`, `vslt`) are predicated under the loop mask `vf0`, except the `vand` instruction, which updates the loop mask. The loop backwards branch is implemented with an unconditional jump instruction (`j`).

Intel MIC accelerators, Cray-1 vector processors, and AMD GPUs execute in a similar manner as shown in Figure 2. The Intel MIC has 8 explicit vector predicate registers [11] while the Cray-1 has one vector predicate register on which vector instructions are implicitly predicated [29, 33]. AMD GPUs use special `vcc` and `exec` predicate registers to hold vector comparison results and to implicitly mask instruction execution [2]. AMD GPUs also provide an escape hatch of sorts for complex, irreducible control flow. Fork and join instructions are provided for managing divergence in these cases, and a stack of deferred paths is physically stored in the scalar register file [3].

Some vector machines lack vector predicate registers and instead have an instruction to conditionally move a word from a source register to a destination register. Packed-SIMD extensions in Intel desktop processors are a common example of this pattern. However, these approaches have limitations, including the exclusion of instructions with side-effects from poly-path execution [18]. Karrenberg and Hack [12, 13] propose compiler algorithms to map OpenCL kernels down to packed-SIMD units with explicit vector blend instructions.

B. NVIDIA Graphics Processing Units

Hardware Divergence Management. NVIDIA GPUs provide implicit divergence management in hardware. As shown in Figure 3, control flow is expressed with *thread branches* (`tbranch.eqz` and `tbranch.neqz` instructions) in the code. When threads in a warp branch in different directions, the warp diverges and is split into two subsets: one for branch taken and the other for branch not taken. Execution of one subset is deferred until the other subset has completed execution. A *divergence stack* is used to manage execution of deferred warp subsets. The compiler pushes a *reconvergence* point with the current active mask onto the divergence stack before the thread branch. The reconvergence point indicates where diverged threads are supposed to join.

In Figure 3a, the reconvergence point is the immediate post-dominator of the if-then-else statement. When the warp splits, the hardware picks one subset (`then` clause), and pushes the other (`else` clause) onto the stack. Then the hardware first executes `op2` under an updated active mask, which is now set to the active threads of the `then` clause. When the hardware executes the `pop.stack` operation, it discards the currently executing warp subset and picks the next warp subset on the top of the stack (`else` clause). When execution reaches the next `pop.stack` operation, it pops the PC for the reconvergence point. If all threads follow the same `then` or `else` path at the branch, the warp hasn't diverged, and so no thread subset is pushed on the stack, and the only `pop.stack` operation pops the PC for the reconvergence point. Divergence and reconvergence nest hierarchically through the divergence stack.

The reconvergence point of a loop is the immediate post-dominator of all loop exits. Since there is only one exit in the example shown in Figure 3b, the exit block is the reconvergence point. The compiler similarly pushes the reconvergence point onto the stack and sequences the loop until all threads have exited the loop. All exited threads will pop a token off the stack, until eventually the reconverged PC with the original mask at loop entry is recovered.

```
     a = op0
     b = op1
     f0 = slt a,b
     cbranch.ifnone f0,else
 @f0 c = op2
else:
     cbranch.ifnone !f0,ipdom
@!f0 c = op3
ipdom:
     d = op4
```

(a) If-then-else statement

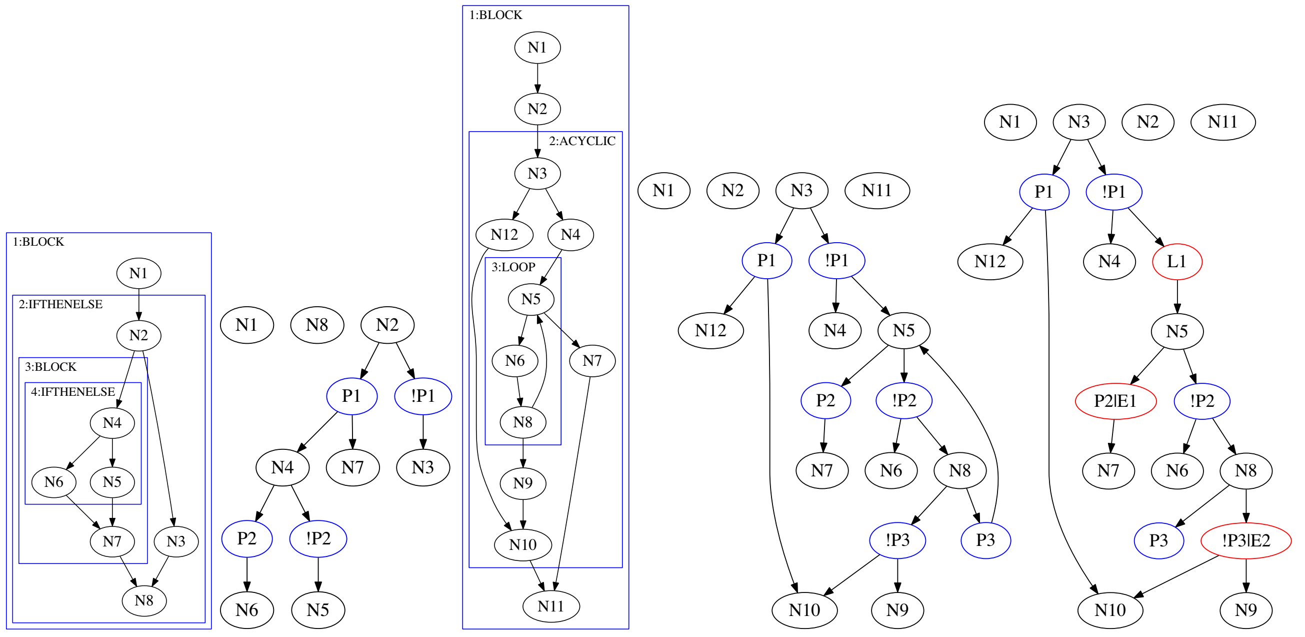
```
     f0 = true
loop:
     cbranch.ifnone f0,exit
 @f0 a = op0
 @f0 b = op1
 @f0 f1 = slt a,b
     f0 = and f0,!f1
     j loop
exit:
     c = op2
```

(b) Loop

Figure 4. Software divergence management on NVIDIA GPUs.

Software Divergence Management. NVIDIA GPUs also provide support to manage divergence in software. The hardware provides predicate registers, native instructions with guard predicates, and *consensual branches* [22], where the branch is only taken when all threads in a warp have the same predicate value. Figure 4 shows how the compiler could manage divergence on NVIDIA hardware. Conditional execution is expressed with predicates, and consensual branches (`cbranch.ifnone` instruction) can be added to jump around unnecessary work (see Figure 4a), and also sequence a loop until all threads have exited (see Figure 4b).

Although consensual branches are implemented in current NVIDIA GPUs, they have not been previously publicly described except for in an issued US patent [22]. NVIDIA's thread-level predication and consensual branches are isomorphic to vector predication and the scalar branches on popcount values used in vector processors. This scheme is only used in a very limited fashion by the current NVIDIA backend compiler. The compiler selects candidate if-then-else regions using pattern matching, and employs a simple heuristic to determine if predication will be advantageous compared to using the divergence stack.



(a) Ctrl flow graph (b) Ctrl dependence graph (c) Ctrl flow graph (d) Ctrl dependence graph (e) Thread-aware ctrl dependence graph

Figure 5. Thread-aware predication code examples. (a) control flow graph, and (b) control dependence graph of an example with nested if-then-else statements, (c) control flow graph, (d) control dependence graph, (e) modified control dependence graph with loop masks and exit masks of a loop example with two exits.

III. Thread-Aware Predication Algorithms

This section describes algorithms that allow a compiler to systematically map divergence present in SPMD programming models down to data-parallel hardware, unlike the heuristic algorithm used to opportunistically predicate control flow in the backend NVIDIA compiler. The proposed compiler algorithms remove all control flow by predicating and *linearizing* different execution paths. These algorithms are applicable to data-parallel architectures that can partially execute instructions and support thread-aware conditional branches. For the discussion, without any loss of generality, we assume predication and consensual branches as the underlying mechanisms. Our general approach to predication is similar to if-conversion described by Mahlke et al. [19], which is a variant of the RK algorithm described in Park and Schlansker [27]. Because predicating compilers are well-studied and have been used effectively on a variety of platforms, this section focuses only on aspects of compilation most relevant to thread-aware predication. We also discuss optimizations to make predication more efficient.

A. Thread-Aware Control Dependence Graphs

The control dependence graph (CDG) is the foundation upon which our predication algorithms rely [8, 20, 37]. A CDG relates nodes in a control flow graph (CFG) by their *control dependencies*. For example, Figure 5b shows a *labeled* CDG for the CFG in Figure 5a [37]. In our labeled CDG, we insert *predicate nodes* (sometimes called *region nodes*) between dependent basic blocks. The predicate nodes provide an explicit relationship between basic blocks and the predicates that guard their execution. For example, consider the if-then-else region $N4$, $N5$, and $N6$. The corresponding CDG has a predicate node $P2$ (and its negation $!P2$) controlling $N6$ and $N5$. As a practical matter, the compiler sets a predicate's value using the same test a branch would use. Using the CDG, the compiler can simply trace a path from the entry node of a region to determine how to *guard* the execution of a basic block with predication. For instance, in 5b we see that $N5$'s instructions should only execute when $P1$ is true and $P2$ is false.

Figure 5d shows that a CDG's utility extends well beyond simple if-then-else regions, and can handle a variety of complex control flow such as the loop and unstructured control flow shown in the CFG in Figure 5c.

While the CDG as presented thus far applies to scalar processors, we can extend it to match the semantics of data-parallel architectures. With predication, when a group of threads execute the same loop simultaneously, the group must be sequenced until all the threads have exited the loop. We augment the CDG's basic structure with *loop masks* to track which threads are still actively executing the loop, *continue masks* to track which threads are in the current iteration of the loop, and *exit masks* to track which threads exited via specific loop exit blocks.

Figure 5e, which extends Figure 5d with these masks, serves as an example in the following description. In the example, $N4$ is the loop's *landing pad*, $N5$ is the *loop header*, and the exit paths are through predicates $P2$ and $!P3$. A *loop mask* represents the threads that are active in the loop body. The compiler initializes a loop mask in the loop's landing pad ($N4$) with a runtime value that represents the active threads at the loop's entrance. For our example, the compiler introduces a loop mask, $L1$, which guards the execution of the loop body. Exit masks, on the other hand, represent threads exiting the loop (through predicate nodes $P2$ and $!P3$ in our example). Exit masks are initialized to `false` in the loop's landing pad.

The compiler inserts instructions at the loop exits to keep the loop masks and exit masks up-to-date. For loop masks, the compiler inserts predicate `and` instructions to mask off threads that exit the loop. A consensual branch is added to all loop exits to check whether the loop mask is null (i.e., all threads are done executing the loop body). For exit masks, the compiler inserts predicate `or` instructions to aggregate the threads that have exited (*e.g.*, $P2|E1$ and $!P3|E2$).

While a *continue mask* is unnecessary for our example, we use them to optimize for the case where all threads in the current iteration execute a continue statement to move to the next iteration. The continue mask is added to the loop header block ($N5$), is initialized to the loop mask of the current iteration, and is iteratively `and`-ed at every continue and exit block with the negated mask of threads that leave the current loop iteration. A consensual branch is added to every continue block to jump to the loop header when the continue mask turns out to be null.

Because the resultant CDG is still valid, downstream predication algorithms can obliviously handle cyclic regions. Karrenberg and Hack [12] use similar loop masks and exit masks to vectorize loops. However, they do not use a CDG formulation to systematically generate predicates, and do not optimize for loop continues.

B. Static Branch-Uniformity Optimization

If the compiler can prove that a branch condition is *thread invariant*, meaning that all active threads simultaneously have the same value, the compiler can replace the branch with a consensual branch and forgo predicating the region. Consensual branches do not affect the hardware's divergence management as all active threads are guaranteed to branch in the same direction. This compile-time *static branch-uniformity optimization* can avoid unnecessary work and also reduce register pressure. For example, in Figure 4a, if the compiler proves that `f0` is thread invariant, the compiler can safely remove the `f0` and `!f0` predicate guards for `op2` and `op3`. In more complex regions, removing the predicate guards can shorten the live range of the associated predicate, thereby reducing register pressure.

We use a modified version of *variance analysis* described in [15,35] to select thread-invariant predicates. A basic block is *convergent* if it is only control dependent on thread-invariant predicates. For convergent basic blocks, all threads that execute simultaneously will either enter with a full mask or not enter at all. Therefore, rather than predicating and linearizing convergent basic blocks, we omit the guard predicate and *preserve* the original control flow. We ignore the thread-invariant predicates in the CDG when generating guard predicates for other basic blocks. If the loop header is convergent, all threads will enter, execute, and exit the loop convergently; hence the CDG does not need to be transformed with loop, continue, and exit masks. Certain control edges must be preserved, otherwise the linearized basic block might execute incorrectly, as convergent basic blocks omit their guard predicates. The rule is to preserve outgoing edges of a convergent basic block if there is only one outgoing edge or when the branch condition of the convergent basic block is proven to be thread invariant.

We modified the variance analysis algorithm to work with the labeled CDG so that we could add an additional rule: if a basic block is controlled by a thread-variant predicate (i.e., proven to be non-convergent), mark all predicates controlling that basic block as thread variant. This constraint is added so that non-convergent basic blocks will have no preserved control edges coming in. For example, in Figure 5d, if predicate $P3$ is thread variant, then mark $P1$ as thread variant, as $N10$ is control dependent on both $P1$ and $!P3$. Otherwise, the compiler will insert a consensual branch at $N3$ since predicate $P1$ is invariant, and assuming the execution went down $N12$ and $N10$, there will be an uninitialized guard predicate representing threads from the $N9$–$N10$ control edge.

To see how this analysis works in practice, assume in Figures 5a/5b that the compiler can prove that $P1$ is thread invariant but cannot do the same for $P2$. As basic block $N2$ is convergent, the predicate generation algorithm can ignore predicate nodes $P1$ and $!P1$, and only consider predicate nodes $P2$ and $!P2$. As a result, basic blocks $N4$, $N7$, and $N3$ do not have guard predicates, while $N6$ and $N5$ are guarded by $P2$ and $!P2$ respectively. Control flow edges $N2$–$N4$ and $N2$–$N3$ are preserved as $N2$ is convergent and $P1$ is thread invariant. $N7$–$N8$ are also preserved as $N7$ is convergent and only has one outgoing edge. Similarly $N3$–$N8$ is preserved as $N3$ is convergent and only has one outgoing edge. As a result of this static branch-uniformity optimization, only $N4$–$N6$–$N5$–$N7$ are predicated. If all the predicates turn out to be thread invariant, all control flow will be preserved. On the other hand, if they are all thread variant, then all control flow will be predicated.

C. Runtime Branch-Uniformity Optimization

For branch conditions that the compiler cannot prove as thread invariant, the compiler can still optimize the control flow by inserting dynamic predicate uniformity tests that consensually branch around whole regions when the active threads all agree. We refer to this as compiler-instigated *runtime branch-uniformity optimization*.

This optimization is guided by *structural analysis* to uncover control-flow structures of interest [31]. Structural analysis allows us to reconstruct control flow structure from a standard CFG. For example, Figure 5a overlays a structural analysis of a CFG with nested if-then-else structures. The structural analysis recursively discovers that blocks $N4$, $N5$, and $N6$ form an if-then-else region. This region is then compressed into an IFTHENELSE block and the algorithm repeats. Likewise, in Figure 5c structural analysis identifies a LOOP ($N5$, $N6$, and $N8$) and an ACYCLIC structure ($N3$, $N4$, $N7$, $N9$, $N10$, $N12$, and the loop).

We consider adding runtime checks to single-entry single-exit (SESE) substructures of IFTHENELSE and IFTHEN flow structures. A simple heuristic algorithm decides to put a runtime uniformity check around the SESE region of the substructure when there are more than three instructions or more than two basic blocks in the SESE region. If the optimization is selected, the compiler adds a header block and a tail block around the SESE region. A consensual branch is added to the header block that branches to the tail block when the predicate guarding the region is uniform. One interesting ramification of this approach is that the inserted branches form scheduling barriers that constrain instruction scheduling.

As an example, assume in Figures 5a/5b that a runtime branch-uniformity check is added to substructures of 4:IFTHENELSE. A header block and a tail block are added around both $N6$ and $N5$. Consensual branches are added to both header blocks checking whether $P2$ and $!P2$ are false, respectively. If the respective predicates are false, instructions in $N6$ and $N5$ are neither fetched nor executed. As Figure 5c has no IFTHENELSE structures, no runtime uniformity checks are inserted.

Shin [32] describes a similar dynamic branch-uniformity optimization called BOSCC (branches-on-superword-condition-codes), but relies on predicate hierarchy graphs to nest regions that are covered by BOSCC to reduce runtime overhead of checking the uniformity of branch conditions. In contrast, we utilize the structural analysis to pick candidate SESE regions for runtime checks.

D. Linearizing the Control Flow

For predicated execution, basic block ordering is clearly important. Consider an if-then-else region for which the compiler has inserted runtime branch-uniformity checks. For cases in which the branches are not uniform, execution will fall through in a predicated fashion from the header, through *both* paths of control flow, and finally exit.

A simple approach to basic block ordering that topologically sorts the loop tree of the CFG as in [12] only works when *all* control flow will be predicated. As we have seen, this assumption is not valid for our solution, because our optimizations intentionally preserve some parts of the original control flow graph. To avoid inserting extraneous branches, which would defeat the purpose of our work, the basic blocks from a predicated region have to be placed contiguously. We refer to this placement problem as linearizing the control flow.

We again turn to structural analysis to achieve a linearized schedule. In addition to the IFTHEN and IFTHENELSE structures, our analysis also discovers BLOCK, LOOP, ACYCLIC, and OTHER structures [31]. The OTHER structure is a catch-all category that represents regions we do not attempt to predicate and schedule. For these, we fall back to a low-performance escape hatch where we put sequencing code at the entry block and the exit block to sequence active threads through the structure one-by-one. Note that irreducible loops will be part of a single-entry single-exit OTHER structure. For the purposes of this paper, the detailed form of the structures is not important. What is important is that the structures discovered in the analysis hierarchically group together basic blocks that need to be contiguous in a predicated execution model.

Once structural analysis reports the control-flow structures, we reverse the direction of all edges in the CFG, and then perform a depth-first post-order traversal from the exit node to generate a valid schedule. We reference the result of the structural analysis to pick which children to visit first to obtain a contiguous schedule of basic blocks from the same control-flow structure. The rule is to first pick children that are from the same innermost structure. To correctly schedule all structures in a loop, we remove backedges that connect loop tails to loop headers and make all edges from the outside point to the loop tail.

Karrenberg and Hack [13] describe a similar static branch-uniformity optimization to reduce register pressure while vectorizing OpenCL kernels for packed-SIMD units in x86 processors. Reducing register pressure is especially important on an x86 processor, as it only has a limited number of scalar and vector registers. While their motivation is similar to ours, they use a different formulation. Uniform branches are identified with a dataflow lattice approach, while we use variance analysis that uses control dependence information to do so. To linearize basic blocks, they utilize a region analysis based on a depth-first search with post-dominator information to identify region exits, while we use structural analysis, which can identify irreducible regions more easily.

IV. Thread-Aware Predication in CUDA Compiler

As NVIDIA GPUs support both hardware-managed and software-managed divergence, we can compare these schemes by implementing our compiler algorithms in the production NVIDIA CUDA toolchain and running real workloads on existing hardware.

A. Implementation

The CUDA compiler takes a CUDA program and translates it to native SASS instructions [25] through a two-step process. The CUDA LLVM compiler first takes a CUDA program and generates PTX instructions with virtual registers and simple branches to represent data and control dependencies. The `ptxas` backend compiler then generates native SASS instructions from the PTX code by allocating hardware registers, inserting instructions that sequence the divergence stack, and performing a very limited version of if-conversion with simple heuristics discussed in Section II-B. When using our predication algorithms, our modified compiler disables passes that insert divergence stack instructions and perform if-conversion.

We implement our compiler algorithms described in Section III in the CUDA LLVM compiler. The LLVM compiler uses a static single assignment (SSA) based internal representation. SSA form is inherently incompatible with the conditional update semantics of predication. The production compiler toolchain in which we prototype these techniques is sufficiently rigid to preclude implementation of the predicate-aware techniques such as [6, 34]. Instead, to interoperate with LLVM's internal representation and built-in passes, during our predication pass we embed a throw-away instruction in each basic block to hold its guard predicate, ordering, and linearization information. The metadata held by the throw-away instruction survives various LLVM passes including the instruction DAG selection pass. We modify `ptxas` to accept the intermediate form with the throw-away instruction mapped to a pseudo-PTX instruction to deliver metadata needed for predication. The throw-away instruction withstands another set of optimization passes in `ptxas`, and is discarded in a late phase once the compiler predicates all instructions with the respective guard predicate and rewires the basic blocks with consensual branches to adhere to the ordering generated by our LLVM predication pass. The resulting binary can be executed on both Kepler and Fermi GPUs without modifications to the CUDA driver.

B. Limitations

While most SASS instructions accept a guard predicate, the shared memory atomic operations, which are implemented with a load-lock and store-unlock instruction sequence, do not. To support programs with conditional execution of shared atomic instructions, we modify `ptxas` to guard the lock/unlock sequence with a divergent branch and handle reconvergence through the divergence stack. Only 4 out of 28 benchmarks are programmed with shared memory atomic operations (SA column of Table I).

Integer division and remainder instructions are expanded into a loop with conditional branches by `ptxas`, invalidating the ordering information generated by our LLVM predication pass. To avoid this problem, we call the "Expand Integer Division" pass to legalize integer division and remainder operations in the LLVM compiler before we call our predication pass. We also add this pass to the baseline compiler, which only uses the divergence stack to handle control flow, for a clearer comparison against thread-aware predication. Only 4 out of 28 benchmarks use an integer division or a remainder operation (DR column of Table I).

V. Evaluation

Our study uses 11 benchmarks from Parboil [36] and 11 benchmarks from Rodinia [4], which cover compute-intensive domains including linear algebra, image processing, medical imaging, biomolecular simulation, physics simulation, fluid dynamics, data mining, and astronomy. We also added 6 benchmarks we wrote to characterize our thread-aware predication CUDA compiler, including a control-flow heavy N-queens benchmark and several FFT benchmarks with different radices. We characterize our thread-aware predication approach on an NVIDIA Tesla K20c GPU (Kepler GK110) and compare performance results with baseline runs that only use the divergence stack to handle control flow. To draw a clearer comparison between hardware and software divergence management schemes, we disable the limited if-conversion heuristic in the baseline compiler so that the baseline solely uses the divergence stack to handle the control flow, and therefore clearly delineate contributions of predication.

A. Benchmark Characterization

Table I reports compile-time and runtime statistics of the different kernels of the 28 benchmarks. The BB, R, and L columns of the table, which count the number of basic blocks, regions, and loops of each kernel, show that the benchmarks are composed of a non-trivial number of control-flow structures. The Br column of the table counts the total number of conditional branches in each kernel. The push and pop of stack instructions columns count the number of baseline compiler generated instructions that sequence the divergence stack. The CBranch instruction columns count the number of consensual branches inserted by the thread-aware predication compiler. The SBU column counts the number of branches that are proved to be non-divergent by the static branch-uniformity optimization. The RBU column reports the number of consensual branches that were added by the runtime branch-uniformity optimization. Unlike the if-then-else statements, consensual branches are required to implement loop constructs correctly. The LC column counts these branches.

We have developed a SASS instrumentation tool, which injects instrumentation code before all conditional branches at the final pass of `ptxas`; we use this tool to collect the runtime uniformity statistics and record them in the Br Uniformity columns. Figure 6a visualizes the classification of compile-time and runtime branches statistics. On average, the compiler proved 11% of branches to be non-divergent (SBU), added dynamic checks for 53% of the branches (RBU), turned 22% of the branches into consensual branches for loops (Loop), and removed the remaining with predication (Predicated). At runtime, 50% of the branches turned out to be uniform, and 48% to be divergent. The remaining 2% of the branches were not executed. In general, the predication compiler is doing a good job optimizing for

TABLE I. BENCHMARK STATISTICS COMPILED FOR KEPLER AND RUN ON TESLA K20C (GK110)

		Compile-time Statistics																			Runtime Statistics						
Application	Kernel	Structures			Inst.			Stack Inst.		CBranch Inst.			Registers				Pred Regs.				Br Uni.		Occup.		Runtime (ms)		Speedup
		BB	R	L	Br	SA	DR	Push	Pop	SBU	RBU	LC	S	P	PS	PSR	S	P	PS	PSR	U	D	S	PSR	S	PSR	
p-bfs	BFS	17	12	2	10	1	0	5	10	0	5	2	20	20	20	20	2	4	4	4	0.18	0.82	0.75	0.75	0.44	0.46	0.96×
	BFS_in_GPU	28	18	4	16	1	0	7	13	0	4	5	31	43	42	42	2	7	5	5	0.38	0.62	0.75	0.50	2.71	3.10	0.90×
	BFS_multi_blk	46	32	7	28	1	0	13	24	0	11	7	39	62	62	62	4	7	6	6	0.16	0.66	0.75	0.50	5.31	5.54	0.93×
p-cutcp	lattice6overlap	30	24	4	17	0	0	9	17	2	8	6	28	43	33	30	1	7	4	3	0.47	0.53	0.69	0.69	4.94	4.48	1.10×
p-histo	main	31	30	4	21	2	0	11	23	1	11	4	23	34	34	30	7	7	7	7	1.00	0.00	0.75	0.75	0.34	0.40	0.86×
	final	10	6	3	6	0	0	2	2	0	0	3	38	42	42	42	1	2	2	2	1.00	0.00	0.75	0.50	0.06	0.06	1.00×
	prescan	46	25	6	25	0	1	11	20	4	11	6	14	16	14	14	2	5	5	5	0.52	0.09	1.00	1.00	0.03	0.03	0.95×
	intermediates	353	208	64	208	0	0	143	174	0	80	64	22	24	24	24	3	5	5	5	0.46	0.54	1.00	1.00	0.21	0.23	0.94×
p-lbm	StreamCollide	3	1	0	1	0	0	1	2	0	1	0	34	41	41	42	1	1	1	1	0.00	1.00	0.75	0.63	2.13	2.12	1.01×
p-mri-gridding	uniformAdd	7	3	0	3	0	0	2	4	0	3	0	6	8	8	6	1	2	2	1	0.64	0.36	1.00	1.00	0.12	0.12	1.00×
	gridding	38	33	7	23	0	3	13	20	11	5	7	62	70	61	58	3	7	4	3	0.00	1.00	0.50	0.50	149.58	155.79	0.96×
	binning	6	3	0	3	0	0	1	3	0	2	0	8	11	11	8	1	4	4	3	0.00	1.00	1.00	1.00	1.94	1.94	1.00×
	reorder	3	1	0	1	0	0	0	0	0	1	0	14	14	14	14	1	1	1	1	0.33	0.67	1.00	1.00	2.55	2.55	1.00×
	scan_L1	20	11	2	11	0	0	6	12	2	3	2	17	16	16	16	2	6	4	4	0.55	0.45	1.00	1.00	0.81	0.81	0.99×
	splitRearrange	10	4	1	5	0	0	3	6	0	3	2	22	25	25	21	2	3	3	3	0.67	0.33	1.00	1.00	1.49	1.65	0.91×
	scan_inter1	5	3	1	3	0	0	1	2	1	1	1	16	15	15	15	1	3	1	1	0.67	0.33	0.61	0.61	0.01	0.01	1.05×
	scan_inter2	5	3	1	3	0	0	1	2	1	1	1	15	16	15	15	1	3	1	1	0.00	1.00	0.61	0.61	0.01	0.01	1.01×
	splitSort	21	12	3	12	4	0	10	16	2	5	3	43	48	41	41	3	7	4	4	0.31	0.69	0.63	0.63	4.35	4.28	1.02×
p-mri-q	ComputeQ	5	3	1	3	0	0	0	0	2	0	1	21	22	22	22	1	3	1	1	1.00	0.00	1.00	1.00	1.60	1.56	1.03×
	ComputePhiMag	3	1	0	1	0	0	0	0	0	1	0	10	10	10	10	1	1	1	1	1.00	0.00	1.00	1.00	0.00	0.00	1.04×
p-sad	mb_calc	27	18	5	17	0	0	9	12	0	8	5	49	62	62	62	3	6	6	6	0.67	0.33	0.50	0.50	8.90	8.31	1.05×
	larger_calc_8	6	3	1	3	0	0	2	3	0	1	1	26	24	24	24	1	2	2	2	0.50	0.50	1.00	1.00	2.92	2.85	1.03×
	larger_calc_16	4	2	1	2	0	0	1	1	0	0	1	26	24	24	24	1	2	2	2	0.63	0.38	0.25	0.25	0.57	0.61	0.96×
p-sgemm	mysgemmNT	6	3	2	3	0	0	0	0	1	0	2	45	38	49	49	1	4	1	1	1.00	0.00	0.63	0.56	2.01	1.87	1.04×
p-spmv	spmv_jds	7	4	1	4	0	0	3	4	0	2	1	19	22	22	20	1	5	5	6	0.75	0.25	0.48	0.48	0.11	0.11	0.98×
p-stencil	block2D_hybrid	34	14	1	14	0	0	12	24	1	12	1	31	42	40	37	7	7	7	7	0.71	0.29	1.00	0.75	0.66	0.68	0.98×
p-tpacf	gen_hists	56	37	5	30	1	0	22	40	4	21	5	29	36	31	31	4	7	4	4	0.59	0.41	0.38	0.38	2040.31	2143.03	0.95×
r-b+tree	findK	12	8	1	7	0	0	2	6	1	3	1	20	24	23	23	1	4	2	2	0.18	0.64	1.00	1.00	1.50	1.52	0.99×
	findRangeK	18	13	1	11	0	0	4	12	1	7	1	27	38	32	32	1	4	2	2	0.29	0.57	1.00	0.75	1.35	1.74	0.77×
r-backprop	layerforward	20	14	3	11	0	1	4	7	1	5	3	21	33	31	31	2	7	6	6	0.40	0.60	1.00	1.00	0.38	0.39	0.96×
	adjust_weights	3	1	0	1	0	0	0	0	0	1	0	19	19	19	19	1	2	2	2	0.00	1.00	1.00	1.00	0.34	0.34	1.00×
r-bfs	Kernel	8	6	1	5	0	0	1	2	0	3	1	16	18	18	18	1	2	2	2	0.40	0.60	1.00	1.00	0.43	0.44	0.95×
	Kernel2	4	2	0	2	0	0	0	0	0	2	0	11	11	11	11	1	2	2	2	0.50	0.50	1.00	1.00	0.05	0.05	0.94×
r-gaussian	Fan1	3	1	0	1	0	0	0	0	0	1	0	11	11	11	11	1	1	1	1	0.00	1.00	1.00	1.00	0.01	0.01	1.01×
	Fan2	5	3	0	3	0	0	0	0	0	3	0	12	15	15	12	1	2	2	2	0.00	1.00	0.25	0.25	0.25	0.26	0.90×
r-hotspot	calculate_temp	14	7	1	8	0	0	3	7	1	4	2	33	35	35	34	3	6	4	3	0.38	0.63	0.75	0.75	0.11	0.11	0.97×
r-lud	lud_diagonal	28	21	7	18	0	0	2	3	0	0	7	44	74	73	73	2	7	7	7	0.83	0.17	0.25	0.25	0.06	0.07	0.90×
	lud_internal	1	0	0	0	0	0	0	0	0	0	0	17	17	17	17	0	0	0	0	–	–	1.00	1.00	0.01	0.01	1.01×
	lud_perimeter	35	27	8	21	0	0	2	3	0	4	8	42	49	49	42	3	7	7	7	0.80	0.20	0.25	0.25	0.08	0.10	0.85×
r-nn	euclid	3	1	0	1	0	0	0	0	0	1	0	8	8	8	8	1	1	1	1	0.00	1.00	1.00	1.00	0.01	0.01	1.01×
r-pathfinder	dynproc	13	6	1	7	0	0	3	6	1	3	2	17	18	17	18	3	4	3	3	0.43	0.57	1.00	1.00	0.15	0.13	1.16×
r-srad-v1	srad	15	8	2	8	0	2	6	13	0	3	2	22	20	20	26	2	5	5	5	0.62	0.15	1.00	1.00	0.09	0.10	0.87×
	srad2	13	7	2	7	0	2	4	4	0	3	2	20	20	20	32	2	5	5	5	0.67	0.33	1.00	1.00	0.07	0.09	0.86×
	reduce	66	43	4	35	0	1	25	46	3	25	4	26	38	36	36	2	7	5	5	0.17	0.43	1.00	0.75	0.08	0.09	0.88×
	extract	3	1	0	1	0	0	0	0	0	1	0	4	4	4	4	1	1	1	1	0.00	1.00	1.00	1.00	0.01	0.01	1.00×
	prepare	3	1	0	1	0	0	0	0	0	1	0	10	10	10	10	1	1	1	1	0.00	1.00	1.00	1.00	0.02	0.02	1.01×
	compress	3	1	0	1	0	0	0	0	0	1	0	4	4	4	4	1	1	1	1	0.50	0.50	1.00	1.00	0.01	0.01	1.01×
r-srad-v2	srad_cuda_1	28	18	0	14	0	0	5	24	4	13	0	23	26	26	26	5	7	7	7	0.36	0.64	1.00	1.00	0.98	1.04	0.95×
	srad_cuda_2	11	5	0	5	0	0	3	8	2	4	0	20	22	22	20	3	5	5	4	0.40	0.60	1.00	1.00	1.01	1.02	1.00×
r-streamcluster	compute_cost	7	4	1	4	0	0	0	0	0	3	1	18	15	15	15	1	4	4	4	0.75	0.25	1.00	1.00	0.52	0.59	0.86×
nqueens	nqueens	47	44	13	32	0	0	11	16	4	7	14	27	54	54	57	2	7	7	7	0.58	0.42	1.00	0.50	55.06	56.05	0.98×
radix2fft	mp_radix2	6	3	1	3	0	0	0	0	2	0	1	35	37	35	35	1	3	1	1	1.00	0.00	0.50	0.50	0.01	0.01	1.01×
	sp_radix2	3	2	1	2	0	0	0	0	1	0	1	22	26	22	22	1	2	1	1	1.00	0.00	0.50	0.50	0.01	0.01	1.04×
radix3fft	radix3fftd	4	2	1	2	0	0	0	0	1	0	1	34	43	33	33	1	2	1	1	1.00	0.00	0.75	0.75	0.01	0.01	0.99×
radix4fft	radix4fftd	3	2	1	2	0	0	0	0	1	0	1	36	40	36	36	1	2	1	1	1.00	0.00	0.25	0.25	0.01	0.01	0.98×
radix5fft	radix5fftd	4	2	1	2	0	0	0	0	1	0	1	44	48	44	44	1	2	1	1	1.00	0.00	0.25	0.25	0.01	0.01	0.99×
radix6fft	radix6fftd	4	2	1	2	0	0	0	0	1	0	1	44	53	45	45	1	2	1	1	1.00	0.00	0.25	0.25	0.01	0.01	1.00×

Note: p=Parboil, r=Rodinia for application names. Kernel names are abbreviated. BB=Basic Blocks, R=Regions, L=Loops, Br=Branches, SA=Shared Atomics, DR=Integer Division/Remainder, SBU=Static Branch-Uniformity optimization, RBU=Runtime Branch-Uniformity optimization, LC=Consensual Branches for Loops, S=Compiled with divergence stack, P=Compiled with thread-aware predication, PS=P+SBU, PSR=P+SBU+RBU, Br Uni.=Branch Uniformity, U=Uniform, D=Divergent, Occup=Occupancy. FFT benchmarks are grouped as one benchmark in future sections.

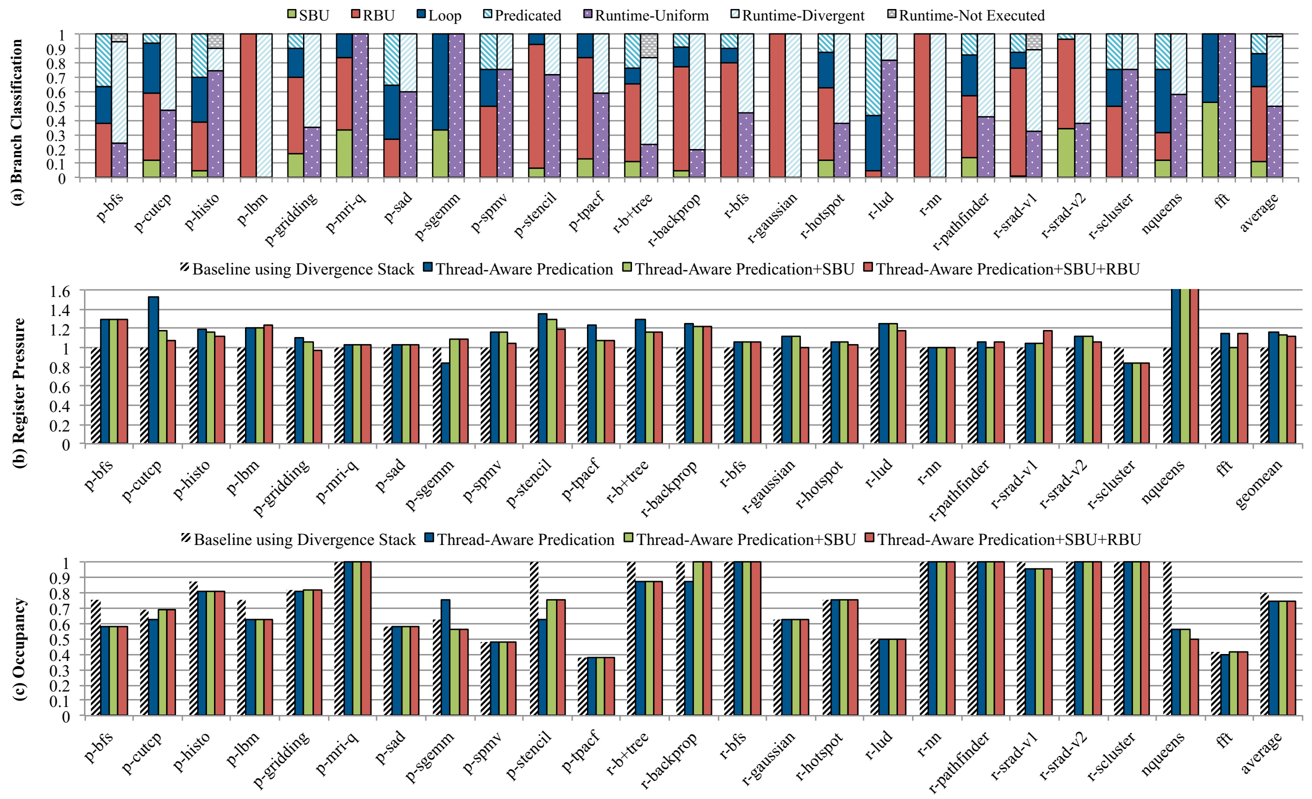


Figure 6. Benchmark characterization with thread-aware (TA) predication: (a) branch classification, (b) register pressure, (c) occupancy. SBU=Static Branch-Uniformity optimization. RBU=Runtime Branch-Uniformity optimization. In (b), register usage for nqueens (outside figure) is 2× for TA-predication and +SBU data points and 2.1× for the TA+SBU+RBU.

branch uniformity. However, there are certain benchmarks such as `r-lud` where the compiler can improve, as the runtime-uniform bar is 17× taller the SBU+RBU bar.

The S column of both register sections counts the number of data registers and predicate registers used by the baseline compiler, which only uses the divergence stack to handle control flow. The P column captures the number of registers used by the thread-aware predication compiler. The PS and PSR columns count the number of registers used by the compiler when the static and runtime branch-uniformity optimizations are enabled respectively. Figure 6b shows the register usage normalized to the baseline register count. The general trend is that predication increases register pressure, since a conservative register allocator cannot reuse registers for both sides of a branch. Normally with branches, a register allocator can easily reuse the same register on different sides of the branch. With predication, the allocator would have to prove that certain predicate conditions are disjoint to do so. For some branches, the static branch-uniformity optimization can prove that a branch is non-divergent so that the compiler can safely remove the predicates from both sides of the branches, make register allocation easier, and alleviate register pressure. Runtime branch-uniformity tends to further reduce register pressure by preventing the compiler from blending instructions from both sides of the branches, hence reducing the live ranges of values.

Register pressure affects occupancy (the number of threads that can execute simultaneously) as the threads share a common pool of physical registers. Figure 6c shows the average occupancy of kernels reported in the occupancy column of the benchmark statistics table, normalized to the maximum occupancy of the processor. Occupancy decreases for those applications that experience increased register pressure: `p-bfs`, `p-histo`, `p-lbm`, `p-sgemm`, `p-stencil`, `r-b+tree`, `r-srad-v1`, and `nqueens`. The static branch-uniformity optimization recoups occupancy lost by the baseline predication algorithm for `p-cutcp`, `p-stencil`, `r-backprop`, and `fft`. As discussed in the next section, occupancy has a strong influence on performance and is a critical metric for compiler optimizations in throughput processors.

B. *Performance Analysis*

Figure 7 shows the performance of all benchmarks normalized to the performance using the divergence stack. Table I has runtime and speedup numbers of all kernels that comprise these benchmarks for the baseline compiler and the thread-aware predication compiler with both branch-uniformity optimizations enabled.

The performance of the thread-aware predication compiler targeting the Kepler GPU only with predication and consensual branches is competitive with the baseline compiler using the divergence stack. The thread-aware predication

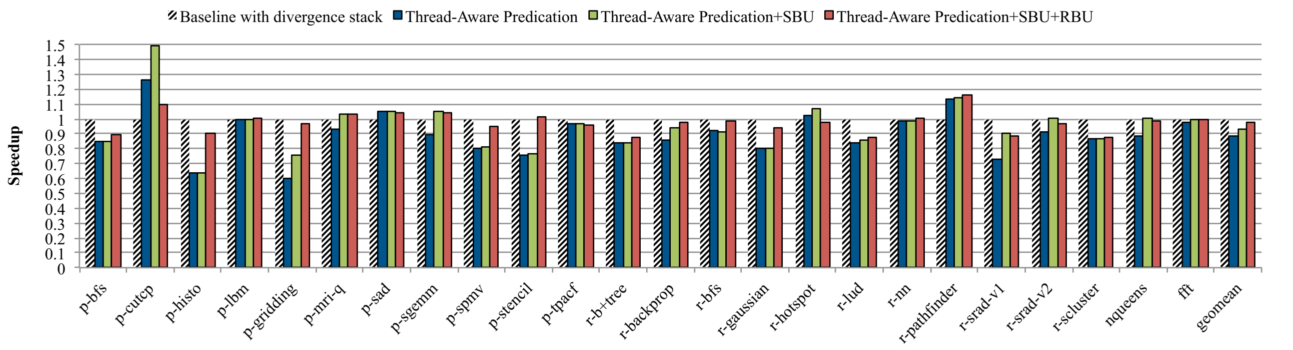


Figure 7. Speedup of thread-aware predication against divergence stack on NVIDIA Tesla K20c. SBU=Static Branch-Uniformity optimization. RBU=Runtime Branch-Uniformity optimization.

compiler with both branch-uniformity optimizations generates code that is only 2.7% slower on average than the baseline compiler. Without any optimizations, the thread-aware predication compiler is 11.3% slower than the baseline. Static branch-uniformity optimization boosts performance by 4.7%, and the runtime branch-uniformity optimization adds an additional 3.9%. Runtime branch-uniformity optimizations harm performance in some cases, especially when the branch conditions are truly unbiased. In such cases, consensual branches added for runtime uniformity checks are pure overhead. Adaptive optimization could use online profiles and recompilation to elide unnecessary uniformity checks. By manually picking the best-performing cases, the thread-aware predication compiler is only 0.6% slower.

The static branch-uniformity optimization tends to increase performance, as this optimization reduces register pressure. The only exception is `p-tpacf`, where performance decreases monotonically with each additional branch-uniformity optimization. Upon inspection the compiler generates a better instruction schedule when both optimizations are disabled by intermixing more parallel execution paths. When both optimizations are disabled, the compiler uses 3 more predicate registers (Table I), and has more freedom with instruction scheduling because more than 80% of the branches are predicated (Figure 6a).

The runtime branch-uniformity optimization increases performance for 11 out of 24 benchmarks by skipping regions with a null guard predicate. Among those 11 benchmarks, `p-histo`, `p-gridding`, `p-spmv`, and `p-stencil` benefit from these runtime checks by 26%, 21%, 14%, and 25% respectively. However, runtime checks can also reduce performance. The worst of the nine benchmarks that performed worse with this runtime optimization is `p-cutcp`, which dropped by 39%. For this benchmark, we found that the compiler is able to schedule multiple fused-multiply-add instructions from multiple execution paths simultaneously when the dynamic checks are not included.

Nine benchmarks performed better with the thread-aware predication compiler than the baseline compiler and 17(19) benchmarks performed within 0.95×(0.90×) of the performance of the baseline compiler respectively. Of the five benchmarks that see a performance loss of more than 10%, `p-bfs`, `r-b+tree`, and `r-srad-v1` exhibit reduced occupancy (Figure 6c). Table I shows that the two kernels with reduced occupancy in `p-bfs` use 11 and 23 additional registers respectively. Likewise, offending kernels `r-b+tree` and `r-srad-v1` use 5 and 10 more registers, respectively. For each of these cases, the additional registers per thread are enough to limit the parallelism that the processor can exploit. As mitigating register pressure is critical to obtaining performance in throughput processors, Section VI-C discusses options for improving register allocation for predicted code. For the remaining two benchmarks `r-lud` and `r-scluster`, the compiler is not able to optimize for all branch uniformity exhibited during runtime (the runtime-uniform bar is much larger than the SBU+RBU bar in Figure 6a). We discuss options for improving branch-uniformity checks in Section VI-C.

Although not shown in the graph, the limited if-conversion heuristic implemented in the NVIDIA production compiler only makes 6 out of 24 benchmarks run faster when compared to the baseline compiler. Eleven benchmarks run slower with the if-conversion heuristic, while the remaining 7 benchmarks run at them same speed. The average performance does not change with the if-conversion heuristic. Out of those 6 benchmarks that run faster with the limited if-conversion heuristic, only 3 benchmarks are 1%, 2%, and 3% faster than code generated from our thread-aware predication compiler respectively.

C. Discussion on Area, Power, and Energy

Quantifying the impact of software divergence management on area, power, and energy is challenging, since we ran CUDA workloads on GPU silicon to obtain performance numbers and benchmark statistics. The primary motivation to remove the divergence stack is to reduce hardware design complexity and associated verification costs. We estimate that area and power savings from eliminating the divergence stack are not significant, so performance would serve as a good proxy for power and energy consumption. The performance of code generated by the thread-aware predication compiler is competitive to the one generated by the baseline compiler using the divergence stack. For that reason, we believe that the power and energy consumption of the software divergence management scheme is on par with the hardware scheme. With improvements proposed in Section VI-C, software divergence management has potential to outperform hardware divergence management schemes in terms of performance, and therefore power and energy efficiency as well.

VI. Discussion

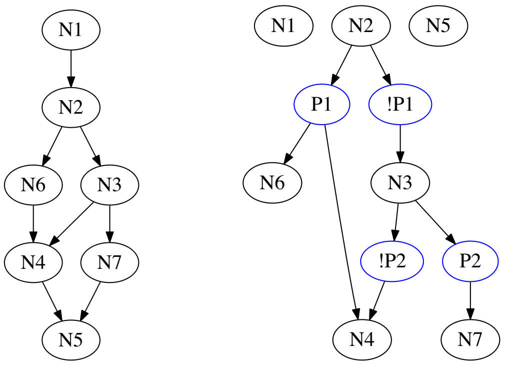
The risks and benefits of predication on latency-oriented architectures such as traditional CPUs are well understood. In that context, branch predictability and instruction path length play a major role [18, 19]. The tradeoffs associated with predication on throughput-oriented architectures with a divergence stack are less studied and less intuitive. This section discusses some of the reasons that extremely aggressive predication is effective on such architectures. We also discuss ways in which we can potentially co-design future throughput-oriented architectures to make predication even more effective.

A. Advantages of Software Divergence Management

In theory, the predicated code should perform as well as code that uses the divergence stack, since the underlying mechanism to handle divergent execution is fundamentally the same. With predication, the compiler is explicitly scheduling the operations in the same way a divergence stack would do implicitly. The reconvergence point when using a divergence stack is known to be the immediate post-dominator in a CFG for if-then-else statements [9]. For loops, the reconvergence point is the immediate post-dominator of all exit blocks, which is the post-tail block of a loop. The algorithms in Section III will find the same reconvergence points and put a predicate with reconverged threads rather than setting up a reconvergence point with a `push.stack` instruction. Whereas the hardware stack has to spill to DRAM if there is too much divergence, the predicating compiler correspondingly manages excessive divergence through explicit predicate register spills. Hence, the main benefit of software divergence management is reducing hardware complexity by eliminating hardware structures used for divergence management without altering the programmer's view of the machine.

There are additional advantages of managing divergence explicitly in software. Figure 8a shows a CFG for a simple short-circuit code segment. Following the reconvergence rule discussed above, when threads diverge at basic blocks $N2$ and $N3$, they will reconverge at $N5$ where parallel execution will resume. Basic block $N4$ can be a partial reconvergence point; however, with a divergence stack, threads will execute $N4$ serially. Using predication with the CDG shown in Figure 8b, diverged threads from $N2$ and $N3$ will join at $N4$ to execute in parallel.

Managing divergence explicitly by the compiler provides more control over irregularly structured code than the divergence stack mechanism can. The order of execution, under divergence stack control, is nondeterministic; the hardware can choose to execute either side of the branch first. This nondeterminism puts a limit on what the compiler can guarantee when analyzing the control flow. For example, the variance analysis used for the static branch-uniformity optimization can make stronger guarantees when divergence is managed by the compiler as it can analyze hazards between different execution paths with a known execution order. Better variance analysis not only results in faster performance with fewer registers, but also opens up more opportunities for scalarizing SPMD code on data-parallel architectures [15].



(a) CFG (b) Control dependence graph

Figure 8. Short-circuit example showing limitations of divergence stack.

```
# function pointer PC stored in r3
# predicate register p2 holds the active threads
@p2 jalr r3
```
(a) With a divergence stack

```
loop:
    p0, r4 = find_unique p2, r3
@p0 jalr r4 # known to be unique
    p2 = p2 and !p0
    cbranch.ifany p2, loop
```
(b) With predication

Figure 9. Supporting virtual function calls with predication.

B. Function Calls

Predicated function calls can be supported by a straightforward calling convention. The convention designates one predicate register (e.g., `p0` register) as the *entrance mask* to hold a mask of threads that are active at function entrance. The compiler then guards via predication all instructions in the function with the entrance mask. If the entrance mask is live across a function call, the compiler should move it to a callee-saved register or spill it to the stack before calling the function.

Predicated virtual function calls can be supported with a simple instruction added to the hardware. Figure 9b shows the predicated version of Figure 9a. The new `find_unique` instruction will return a unique value of a vector register (namely an address for the function) and a predicate mask of active threads that holds the unique value.

Following the calling convention, we save the resulting predicate mask in `p0` and then jump to the unique program counter. When control from the function returns, we mask off the threads that executed the function, and check whether any active threads still remain. If so, we loop back and repeat with a new program counter until all active threads are sequenced.

C. Improving Software Divergence Management

Although our experimental thread-aware predication compiler is competitive in performance to a well-tuned production compiler that uses the divergence stack, we expect that we can make software divergence management even more effective with the following software and hardware improvements.

Tuning our compiler. Our heuristics for deciding when to enable the runtime branch-uniformity optimization have not been extensively tuned. More importantly, many of the downstream compiler passes have not been tuned with our optimizations in mind. In fact, some downstream compiler optimizations are not predicate-aware, rendering them ineffective. Some effort globally tuning the compiler and implementing common predicate-aware analyses and transformations could provide performance boosts.

Predication-aware register allocation. Register count affects occupancy, which has a strong influence on performance. Several studies [7, 10] look into techniques to reduce register pressure under predication for superscalar and VLIW architectures. We can apply similar techniques to reduce register count, and hence increase occupancy.

Better branch-uniformity optimizations. As shown in Figure 7, the compiler is only able to capture a small fraction of the runtime branch uniformity. The current variance analysis used in static branch-uniformity optimization only considers convergent basic blocks, where all threads will enter with a full mask or not. The compiler does not analyze the case where a subset of the warp has the same branch condition. With the execution order of divergent regions understood by the compiler, we can extend the analysis to report scalar branch conditions across a subset of the warp. We spotted some cases where the structural analysis was not reporting all regions of interest to add runtime branch-uniformity checks. Other algorithms should also be considered to determine where and when to add these runtime checks.

Branch if any instruction. The current hardware only supports `cbranch.ifnone` or `cbranch.ifall` instructions. To emulate a `cbranch.ifany` instruction, we need two branches in a row, a `cbranch.ifnone` followed by an unconditional jump. A new instruction would improve performance as it would decrease instruction count and make unrolling easier by eliminating a branch instruction from the middle of an unrolled region.

Adaptive optimization. As shown in our performance results, static branch-uniformity optimization and runtime branch-uniformity optimization can sometimes reduce performance. Adaptive optimization in a just-in-time compilation scheme (such as implemented in the NVIDIA CUDA driver) could profile branch behaviors and generate code that selects the best optimizations on a per-branch basis. Feedback-directed optimization could similarly improve our results.

VII. CONCLUSION

Trading complexity back and forth between software and hardware is a classic debate in computer architecture. Divergence management is a prime target for these tradeoffs, with a wide range of software, hardware, and hybrid schemes implemented in the field. However, while hardware divergence management schemes have received a lot of attention from the academic research community, the benefits and drawbacks of software divergence management on data-parallel architectures have been less explored.

Hardware divergence management has its advantages. It enables a fairly conventional thread compilation model, makes register allocation easier, and simplifies the task of supporting complex irreducible control flow. However, in doing so the hardware takes on the burden of implementing fairly complex divergence management structures. By trading the complexity with the compiler to manage some or all divergence explicitly in software, we can potentially simplify the hardware without sacrificing programmability.

In this paper, we have presented novel compiler algorithms to systematically map arbitrarily nested control flow present in SPMD programs down to data-parallel architectures with predicates and consensual branches. We have implemented these compiler algorithms in a production CUDA compiler, and used it to run real workloads and gather runtime statistics on existing hardware. Our detailed performance analysis on an NVIDIA Tesla K20c show that software divergence management architectures can be competitive to hardware divergence management architectures. We anticipate that with our suggested software and hardware improvements, software divergence management schemes can be even more effective.

ACKNOWLEDGMENTS

This work was funded by DARPA award HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, an NVIDIA graduate fellowship, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, Oracle, and Samsung. It was also funded by DOE contract B599861. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 177–189, January 1983.

[2] "AMD Graphic Core Next Architecture," AMD Fusion Developer Summit 11, 2011.

[3] "Southern Islands Series Instruction Set Architecture," AMD White Paper, 2012.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," *International Symposium on Workload Characterization (IISWC)*, pp. 44 –54, October 2009.

[5] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira, "Divergence Analysis and Optimizations," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 320–329, October 2011.

[6] B. D. de Dinechin, "Using the SSA-Form in a Code Generator," *Compiler Construction*, pp. 1–17, 2014.

[7] A. E. Eichenberger and E. S. Davidson, "Register Allocation for Predicated Code," *International Symposium on Microarchitecture (MICRO)*, pp. 180–191, November 1995.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, pp. 319–349, July 1987.

[9] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no. 2, pp. 1–35, June 2009.

[10] D. M. Gillies, D.-c. R. Ju, R. Johnson, and M. Schlansker, "Global Predicate Analysis and Its Application to Register Allocation," *International Symposium on Microarchitecture (MICRO)*, pp. 114–125, December 1996.

[11] "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual," Intel White Paper, 2012.

[12] R. Karrenberg and S. Hack, "Whole-function Vectorization," *International Symposium on Code Generation and Optimization (CGO)*, pp. 141–150, April 2011.

[13] R. Karrenberg and S. Hack, "Improving Performance of OpenCL on CPUs," *International Conference on Compiler Construction (CC)*, pp. 1–20, March 2012.

[14] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs Between Programmability and Efficiency in Data-Parallel Accelerators," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 6:1–6:38, August 2013.

[15] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanović, "Convergence and Scalarization for Data-Parallel Architectures," *International Symposium on Code Generation and Optimization (CGO)*, February 2013.

[16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computer Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar/Apr 2008.

[17] C. Lomont, "Introduction to Intel Advanced Vector Extensions," Intel White Paper, 2011.

[18] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," *International Symposium on Computer Architecture (ISCA)*, pp. 138–149, June 1995.

[19] 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," *International Symposium on Microarchitecture (MICRO)*, pp. 45–54, December 1992.

[20] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, March/April 2008.

[22] J. R. Nickolls, R. C. Johnson, R. S. Glanville, and G. J. Rozas, "Unanimous branch instructions in a parallel thread processor," US Patent 8,677,106, March 2014.

[23] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," NVIDIA White Paper, 2009.

[24] "NVIDIA's Next Gen CUDA Compute Architecture: Kepler GK110," NVIDIA White Paper, 2012.

[25] "CUDA Binary Utilities," NVIDIA Application Note, 2014.

[26] "The OpenCL Specification Version 1.2," Khronos OpenCL Working Group, 2011.

[27] J. C. H. Park and M. Schlansker, "On Predicated Execution," Hewlett Packard Laboratories, Tech. Rep. HPL-91-58, May 1991.

[28] S. Raman, V. Pentkovski, and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium-III Processor," *IEEE Micro*, vol. 20, no. 4, pp. 47 –57, Jul/Aug 2000.

[29] R. M. Russell, "The Cray-1 Computer System," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, January 1978.

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *IEEE Micro*, vol. 29, no. 1, pp. 10–21, Jan/Feb 2009.

[31] M. Sharir, "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers," *Computer Languages.*, vol. 5, no. 3-4, pp. 141–153, January 1980.

[32] J. Shin, "Introducing Control Flow into Vectorized Code," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 280–291, September 2007.

[33] J. E. Smith, G. Faanes, and R. Sugumar, "Vector Instruction Set Support for Conditional Operations," *International Symposium on Computer Architecture (ISCA)*, pp. 260–269, June 2000.

[34] A. Stoutchinin and F. De Ferriere, "Efficient Static Single Assignment Form for Predication," *International Symposium on Microarchitecture (MICRO)*, pp. 172–181, December 2001.

[35] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. mei W. Hwu, "Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs," *International Symposium on Code Generation and Optimization (CGO)*, pp. 111–119, April 2010.

[36] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois, Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[37] M. Weiss, "The Transitive Closure of Control Dependence: the Iterated Join," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, pp. 178–190, June 1992.