In the Proceedings of the 23rd International Conference
on Compiler Construction, Grenoble, France, April 2014.

Taming Control Divergence in GPUs through Control Flow Linearization

Jayvant Anantpur and Govindarajan R.

Supercomputer Education and Research Centre
Indian Institute of Science
jayvant@hpc.serc.iisc.ernet.in, govind@serc.iisc.ernet.in

Abstract. Branch divergence is a very commonly occurring performance problem in GPGPU in which the execution of diverging branches is serialized to execute only one control flow path at a time. Existing hardware mechanism to reconverge threads using a stack causes duplicate execution of code for unstructured control flow graphs. Also the stack mechanism cannot effectively utilize the available parallelism among diverging branches. Further, the amount of nested divergence allowed is also limited by depth of the branch divergence stack.
In this paper we propose a simple and elegant transformation to handle all of the above mentioned problems. The transformation converts an unstructured CFG to a structured CFG without duplicating user code. It incurs only a linear increase in the number of basic blocks and also the number of instructions. Our solution linearizes the CFG using a predicate variable. This mechanism reconverges the divergent threads as early as possible. It also reduces the depth of the reconvergence stack. The available parallelism in nested branches can be effectively extracted by scheduling the basic blocks to reduce the effect of stalls due to memory accesses. It can also increase execution efficiency of nested loops with different trip counts for different threads.
We implemented the proposed transformation at PTX level using the Ocelot compiler infrastructure. We evaluated the technique using various benchmarks to show that it can be effective in handling the performance problem due to divergence in unstructured CFGs.

Keywords: GPU, Control Divergence, Control Flow Graph

1 Introduction

There has been a tremendous increase in the use of GPUs in general purpose programming, especially to accelerate data parallel code. The emergence of programming models such as CUDA [17], OpenCL [13] etc., has fuelled the use of GPUs.

Programming models such as CUDA, OpenCL, etc., use the Single Instruction Multiple Threads (SIMT) computation model [17]. In this model a large number of threads run in parallel on Single Instruction Multiple Data (SIMD) cores using hardware multithreading to hide the stalls due to long latency instructions. A group of threads, called a warp, is scheduled to execute on the SIMD processors. Each thread in a warp executes the same instruction. The execution of a branch instruction can cause the control flow to diverge. The existing hardware solution to handle branch divergence serializes

execution of the two paths till a reconvergence point [10]. Branch divergence is one of the major sources of performance bottlenecks in GPUs [7] [12] [16] [22]. The diverging threads are reconverged at the immediate post-dominator (IPDOM) of the branch instruction.

IPDOM guarantees earliest reconvergence for structured CFGs but not for unstructured CFGs. Unstructured CFGs (for definition see section 3.2) can result due to the use of programming language constructs such as goto, break statements, short circuiting operations, etc., and also due to compiler optimizations such as function inlining [22]. The work by Wu et al. [22] characterizes the use of unstructured control flow in GPU applications. As per their findings, unstructured CFGs are common in GPU applications and benchmarks (in 40% of the Parboil, Rodinia and Optix benchmarks).

In the case of unstructured CFGs, some basic blocks between the divergent branch and its IPDOM may get executed multiple times due to different paths to reach those blocks from the divergent branch. Diamos et al. [7] proposed a combined hardware and software solution for this problem and found a reasonable reduction in dynamic instruction counts for several real applications. Their proposed solution identifies the thread frontier of each block - set of basic blocks where all other diverged threads may be executing - and then checks for stalled threads waiting in the thread frontier.

Serial execution of different paths of a branch cannot effectively utilize the parallelism among the paths especially in the case of nested branches and nested loops. In the case of nested branches, the different execution paths cannot be interleaved to extract parallelism among them. Rhu et al. [16] proposed a modification to the existing hardware stack to enable interleaved execution of divergent control flow paths. They showed performance improvement by utilizing the available parallelism among the diverging control flow paths. In the case of nested loops, when the inner loops have different trip counts for different threads of the same warp, threads with smaller trip counts have to wait for the threads with larger trip counts. Han et al. [12] proposed a compiler transformation to reduce the effect of divergence due to varying trip-counts. It merges a divergent loop with one or more outer surrounding loops into one loop.

Another limitation of the hardware reconvergence stack is that its depth increases as the nesting level of branches increases.

In this paper we describe a simple compile time transformation to convert unstructured CFGs to structured CFGs. The transformation uses a predicate variable to guard the execution of basic blocks in a CFG. The guard variable acts like a software Program Counter to decide which basic block to execute next. The transformation implements the control flow as a simple ”if-then” structure, linearizing the CFG. The proposed transformation is powerful to convert any unstructured CFG to a structured one. It does not duplicate the user code unlike in the earlier approach of Zhang et al. [23]. Carter et al. [5] proved that any node-splitting technique used to convert an irreducible graph to a reducible graph can increase the code size exponentially. Our algorithm, though does not fall under node-splitting category, will only cause a linear increase in the code size.

To summarize, our contributions are:

- We propose a very simple and elegant transformation to convert an unstructured CFG to a structured CFG, with just a linear increase in the number of basic blocks and instructions.

– We also demonstrate that the proposed transformation is powerful and versatile. It can be used to handle various performance problems due to branch divergence. In particular it (a) ensures reconvergence at IPDOM and hence no duplicate execution of code, (b) enables interleaved execution of blocks from different parts of a divergent branch and (c) enables merging different invocations of inner loops.
– We show the feasibility of implementing the transformation at PTX level and some initial experimental results.

To the best of our knowledge, our work is the first to use the transformation described to convert an unstructured CFG to a structured CFG and further use this idea to reduce negative effects of branch divergence on GPU performance.

2 Motivation

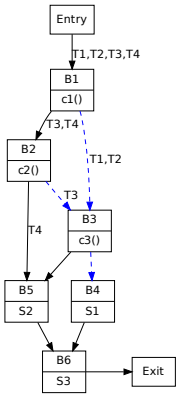
In this section we discuss in detail the problems due to control divergence in both unstructured and structured Control Flow Graphs (CFG), arising because of the existing hardware reconvergence stack and IPDOM mechanism.

2.1 Control Divergence

In GPUs, a group of consecutive threads, called warps, execute together the same instruction in a lock step manner.

```
if ((c1() || c2()) && c3()) {
  S1;
else
  S2;
}
S3;
```

(a) C-Code



(b) CFG

Fig. 1: Short Circuit example

While executing a branch instruction, if the branch condition evaluates to true for some threads and false for the other threads of a warp, then the two parts of the branch statement are executed one after the other, masking out threads based on their branch condition evaluation. This results in smaller groups of threads executing the *then* and *else* parts. These groups are then merged back when control reaches the IPDOM of the branch node. In the case of a structured CFG, IPDOM of the branch node is the earliest reconvergence point, but for an unstructured CFG, there may be other nodes of the graph where some of the diverging groups can potentially reconverge before all the threads reconverge at the IPDOM. So, if all the subgroups are allowed to run without reconverging till the IPDOM, some of the nodes in the CFG may be executed multiple times. This leads to duplicate execution of instructions. We explain this using the example in Figure 1(a). The unstructured CFG is shown in Figure 1(b).

For this example, let us assume a warp with 4 threads T1-T4. Also assume that threads T1-T2 evaluate c1() to true and T3-T4 evaluate it to false. This causes the threads to diverge at the end of block B1. The diverging threads are reconverged at the IPDOM block B6 corresponding to statement S3. Threads T1-T2 take B1→B3 path whereas threads T3-T4 take B1→B2 path. If we assume that thread T3 evaluates c2() to true and T4 evaluates it to false, block B3 will be executed twice, once for threads T1-T2 and then for thread T3. We observe that the problem of branch divergence and repeated execution of basic blocks exacerbates as we go down the unstructured control flow graph. For example, B5 is executed thrice, once for each thread T2, T3 and T4.

2.2 Branch Interleaving

The serial execution of the *then* and *else* parts of a branch also forgoes the potential parallelism that can be achieved by interleaved execution of the two parts.

```
B1 if (c1 == 1) {
B2   S1;
       if (c2 == 1)
B3       S2;
       else
B4       S3;
     } else {
B5   S4;
       if (c3 == 1)
B6       S5;
       else
B7       S6;
     }
```

(a) C-Code Example for Branch Interleaving

```
for (i = 0; i < 2; i++) {
  S1;
  for (j = 0; j < cond; j++)
    S2;
  S3;
}
```

(b) C-Code Example for Loop Merging

Thread	i=1	i=2
T1	cond=10	cond=15
T2	cond=15	cond=10

(c) Trip Counts for Loop Merging Example

Fig. 2: Branch Interleaving and Loop Merging Examples

Consider a branch with multiple basic blocks in each part of the branch as shown in Figure 2(a). The beginning of each block is shown on the left side. In this example, if we assume that the *then* parts execute before *else*, the basic blocks will execute in the order B1, B2, B3, B4, B5, B6, B7. If the execution in the *then* part stalls due to unavailability of operands, e.g. consider a load in B2 that results in an L1/L2 cache miss, with its use in B3, the available ILP in B4 or other basic blocks cannot be utilized to mask latencies in B3. This is because the execution of B4 cannot start until B3 finishes. Thus SIMD cores may remain unused until the operands needed for B3 are ready. One of the main reasons for stalls is the high latency for memory accesses. In the absence of enough threads to hide the high latency for memory accesses, the ability to execute code from other paths of divergent branches can improve the utilization of the hardware and also improve the performance. If the blocks can be ordered differently, e.g. B1, B2, B5, B3, B6, B4, B7, then the stall on use in B3 can potentially be avoided. Since execution stalls when a needed operand is not available, the blocks can be ordered so as to increase the distance between definition and use of an operand. So, if the definition is a memory load in B2 and the use is in B3, then execution of B5 between B2 and B3 can avoid the stall in B3.

2.3 Loop Merging

Consider a kernel with a nested loop in which the inner loop has different trip counts for different threads. Threads of a warp diverging on the inner loop reconverge at the end of that loop. So, threads with smaller trip counts of the inner loop have to wait for all other threads of the warp to finish the inner loop execution, before proceeding further.

Consider the example in Figure 2(b). Figure 2(c) shows a scenario where the two threads have different trip counts for the inner loop. With the existing reconvergence mechanism, both iterations of the outer loop will execute 15 iterations of the inner loop. In the first iteration of the outer loop, thread T1 finishes the inner loop execution after 10 iterations but has to wait for thread T2 to finish its remaining 5 iterations. Instead, if thread T1 is allowed to execute statement S3 and start next iteration of the outer loop, it can join thread T2 in the execution of the inner loop. With this capability of executing different invocations of a loop for different threads of a warp, the hardware can be used more efficiently.

2.4 Hardware Stack Depth

The reconvergence mechanism using hardware stack, handles branch divergence by pushing two entries on to the hardware stack, one each for the two paths of the branch. The entry consists of PC of the path, an active mask representing the set of threads in the warp that follow this path and the reconvergence PC.

```
B1    if (c1) { //branch 1
B2      if (c2) { //branch 2
B4        if (c3) //branch 3
B6          S1;
          else
B7          S2;
B8        S3;
        else
B5        S4;
B9      S5;
      else
B3      S6;
B10   S7;
```

(a) C-Code

PC	Active Mask	RPC
B6	1000	B8
B7	0100	B8
B8	1100	B9
B5	0010	B9
B9	1110	B10
B3	0001	B10
B10	1111	-

(b) Reconvergence Stack during execution of S1

Fig. 3: Stack Depth Example

Each entry is popped when the control flow corresponding to it reaches the reconvergence PC (IPDOM) of the divergent branch node. In this way the stack depth increases for nested divergent branches and hence cost of the hardware needed to support nested branches also increases.

In the example shown in Figure 3(a), let us assume 4 threads and for each branch one thread takes the *else* path and the remaining threads take the *then* path. Also assume that the *else* path is pushed onto the stack first and then the *then* path. Initially there is only one entry on the stack corresponding to all the threads executing block B1. The execution of branch 1 adds two entries to the stack corresponding to blocks B2 and B3. Then the execution of branches 2 and 3 will add two entries each to the stack. So when S1 is executing, there are 7 entries on the stack as shown in Figure 3(b).

All the above mentioned problems with the existing hardware stack based reconvergence mechanism can be solved with our technique of linearizing a control flow graph.

3 Linearization Transformation

In the previous section we saw that the reconvergence mechanism using IPDOM suffers from duplicate execution for an unstructured CFG. Our proposed transformation converts an unstructured CFG to a structured CFG and hence eliminates the problem of duplicate execution.

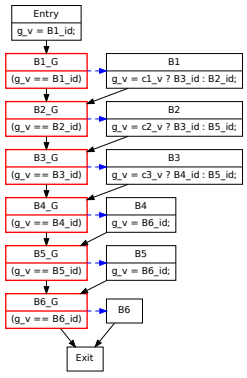
In this section we will discuss the linearization transformation in detail, show that it transforms an unstructured CFG to a structured CFG, prove correctness of the transformation and then analyze the increase in code size.

3.1 Linearization

Linearization algorithm is based on the idea of predicated/guarded execution of the basic blocks of a CFG.

```
int g_v = B1_id;
if (g_v == B1_id) {
  code_for_B1;
  g_v = c1_v ? B3_id : B2_id;
}
if (g_v == B2_id) {
  code_for_B2;
  g_v = c2_v ? B3_id : B5_id;
}
if (g_v == B3_id) {
  code_for_B3;
  g_v = c3_v ? B4_id : B5_id;
}
if (g_v == B4_id) {
  code_for_B4;
  g_v = B6_id;
}
if (g_v == B5_id) {
  code_for_B5;
  g_v = B6_id;
}
if (g_v == B6_id)
  code_for_B6;
```

(a) Transformed Code



(b) Transformed CFG

Fig. 4: Transformed Short Circuit Example

For each basic block of the input CFG, the transformation creates a guard basic block to guard its execution; the guard condition is set by its predecessors. We will explain this with the short circuit example from Figure 1. The transformed code is shown in Figure 4(a) and the corresponding CFG is shown in Figure 4(b).

Execution of the entry block assigns B1_id to the guard variable as block B1 is the only successor of the entry block. As the condition in the first branch statement evaluates to true, code for block B1 is executed and the guard variable is set to index of the successor of B1 based on the branch condition, i.e., value of c1_v, where c1_v contains the result of function call c1(). (Note that in the original code, based on the value of c1_v, control transfers either to block B2 or B3). This way at the end of execution of block B1, the guard variable contains index of the next block to be executed, i.e. B2_id or B3_id. Assuming that block B1 sets the guard variable to the index of block B3, execution of block B2 is skipped and block B3 is executed (i.e. the guard condition for block B3 evaluates to true and code for block B3 is executed). This way the execution continues till block B6. As can be seen from Figures 4(a) and (b), linearization algorithm transforms the input CFG into a sequence of predicated blocks.

3.2 Unstructured CFG to Structured CFG

In this subsection, we explain formally how linearization converts an unstructured CFG to a structured CFG.

Definition 1. *An edge from block B_i to B_j is said to be unstructured if any of the following three conditions is satisfied:*

- *Block B_i has multiple successors, block B_j has multiple predecessors, and neither of B_i or B_j dominates nor postdominates the other,*
- *Block B_j is in a loop, block B_i is not in the same loop and B_j does not dominate all other blocks of the loop,*
- *Block B_i is in a loop, block B_j is not in the same loop and B_i does not postdominate all other blocks of the loop*

For example, in Figure 1(b), edges B2→B3, B2→B5 and B3→B5 are unstructured edges as they satisfy the first condition. Edges B1→B5 and B2→B3 in Figure 6(b) are marked as unstructured edges as they jump into the loop formed by blocks B3, B4 and B5. In Figure 5(b), edge B3→B6 and B2→B5 are unstructured edges as they jump out of the loop formed by blocks B2, B3 and B4.

Zhang et al. [23] showed that repeated applications of their three transformations convert all possible unstructured programs into structured programs. The three transformations proposed by them are (a) Forward Copy - for unstructured edges in an acyclic CFG, (b) Backward Copy - for incoming edge of a loop and (c) Cut - for outgoing edge of a loop. Based on the prior works by Zhang et al. [23], Wu et al. [22], and our extensive study of unstructured CFGs from various benchmark suites, we claim that the 3 conditions specified in the definition of an unstructured edge cover all possible cases of unstructuredness.

We call a CFG containing an *unstructured edge*, an *unstructured CFG (UCFG)*. For the discussion in this subsection we define *unstructured region* to be the region encompassing all blocks of the input CFG except for the Entry and Exit blocks. Also we use the term *unstructured block* to refer to any block from an unstructured region. In the next section we will present an algorithm to find the minimal *unstructured region* of an *unstructured edge*.

Now we will describe the algorithm to transform an UCFG into a structured CFG.

Definition 2. *The common immediate dominator,* CIDOM, D *of a set of blocks* B *in a CFG is a block such that* D *dominates all the blocks in* B *and there does not exist any other block $\hat{D}$ such that $\hat{D}$ dominates all the blocks in* B *and* D *dominates $\hat{D}$.*

Definition 3. *The common immediate postdominator,* CIPDOM, P *of a set of blocks B in a CFG is a block such that* P *postdominates all the blocks in* B *and there does not exist any other block $\hat{P}$ such that $\hat{P}$ postdominates all the blocks in* B *and* P *postdominates $\hat{P}$.*

So, a block that dominates all the blocks in *B*, will dominate the *CIDOM*. Similarly, a block that postdominates all the blocks in *B*, will postdominate the *CIPDOM*.

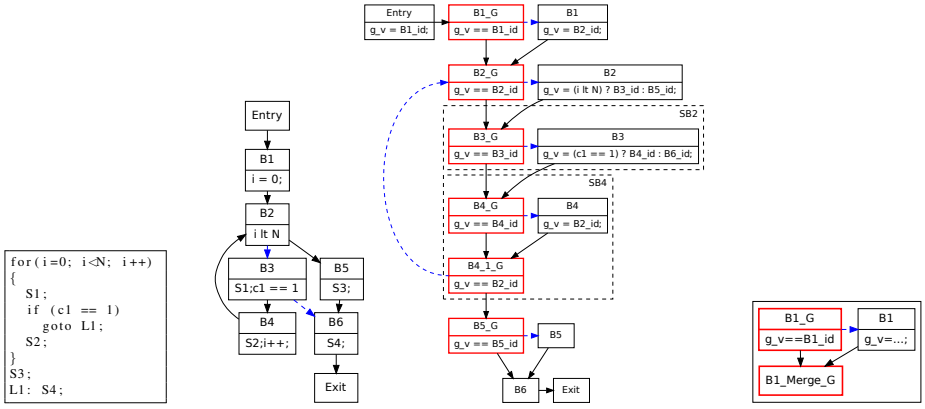
Algorithm 1 Linearization

```
1: procedure LinearizeUnstructuredRegion(Ureg)
2: idom ← immedDom(Ureg)
3: ipdom ← immedPostDom(Ureg)
4: prevGuard ← 0, prevBlock ← 0
5: for all blk ∈ Ureg, inRevPostOrder do
6:     guard = createGuard(blk)
7:     addGuardVarAssign(blk)
8:     addBrEdge(guard, blk)
9:     if isFirstBlock(blk) then
10:        addEdge(idom, guard)
11:    end if
12:    if prevGuard ≠ 0 then
13:        addEdge(prevGuard, guard)
14:    end if
15:    if prevBlock ≠ 0 then
16:        addEdge(prevBlock, guard)
17:    end if
18:    if isLastBlock(blk) then
19:        addEdge(guard, ipdom)
20:        addEdge(blk, ipdom)
21:    end if
22:    prevGuard ← guard
23:    prevBlock ← blk
24:    if isSrcOfRetreatingEdge(blk) then
25:        beGuard ← createBEGuard(blk)
26:        addEdge(guard, beGuard)
27:        addEdge(blk, beGuard)
28:        beDst ← getBackEdgeDst(blk)
29:        beDstGuard ← getGuard(beDst)
30:        addBrEdge(beGuard, beDstGuard)
31:        prevGuard ← beGuard
32:        prevBlock ← 0
33:    end if
34: end for
35: end procedure
```

Our algorithm for transforming unstructured graphs first computes the *CIDOM* and *CIPDOM* of the input CFG. The next step is to generate a reverse post-order (also known as depth first order [1]) for all the blocks in the unstructured region, starting from the *CIDOM* up to the *CIPDOM*. The reverse post-order ensures that before a block is traversed all its predecessors are traversed. This helps to minimize the number of guard checks during execution of the transformed CFG. This is similar to the case of using a reverse post-order traversal in an iterative algorithm for a forward data-flow problem. For example in Figure 4(b), if block B5 was before block B3, then the linearized CFG would have needed a back edge to execute block B5 after block B3. In case of a back edge in a CFG, the reverse post-order contains the destination of the back edge before the source and hence the linearized CFG contains a back edge for it.

Algorithm 1 shows the steps to linearize an unstructured region. The blocks in the input unstructured region are traversed in the reverse post-order and for each block, a guard block is created (line 6). An assignment is added to each unstructured block, to set the guard variable to the block id of one of its successor blocks in the original CFG (line 7). The guard block is populated with an instruction to branch to the unstructured block - block from the original CFG in an unstructured region - when the guard variable value matches the block's index (line 8). The guard block corresponding to the first unstructured block is added as a successor of the *CIDOM* of the unstructured region (lines 9-11). The other successor of the guard block is the guard block for the next unstructured block in the reverse post-order (lines 12-14). This is the successor on the fall through edge of the guard block. Lines 15-17 add a guard block as successor of the previous unstructured block in the reverse post-order. The successor of the last guard block is the *CIPDOM* of the unstructured region (lines 18-20).

Applying this algorithm on the unstructured CFG in Figure 1(b), we get the transformed CFG in Figure 4(b). B1_G to B6_G are the guard blocks (line 6). In blocks B1-B5, an assignment is added to set the guard variable to the id of the appropriate successor block (line 7). Each of B1_G to B6_G contains the guard variable check (line 8). Block B1_G is made the successor of the entry block (lines 9-11). As per lines 12-



(a) Code Snippet (b) CFG (c) Transformed CFG (d) S-Block

Fig. 5: Jump out of a loop

14, blocks B2_G to B6_G are made successors of blocks B1_G to B5_G respectively. Blocks B2_G to B6_G are also added as successors of blocks B1 to B5 respectively (lines 15-17). Finally lines 18-20 add Exit as the successor of blocks B6 and B6_G.

For a block which is the source of a retreating edge (i.e. an edge in which the destination appears before source in the reverse post-order), another guard block is created (lines 24-33). For the retreating edge B4→B2 in Figure 5(b), block B4_1_G is created and a back edge is added to block B2_G as shown in Figure 5(c). None of the blocks in the loop of this back edge, other than the two blocks of the back edge, can be the destination of any other back edge, as that will make the CFG unstructured (jump into a loop). In other words, in the reverse post-order of an unstructured CFG, none of the blocks that lie between the destination and source of a retreating edge, can become the destination of any other back edge unless its source also lies between them. This condition ensures that any two loops in the transformed CFG are either nested or disjoint. If there exists an edge in the input CFG to any of such blocks then destination of the corresponding back edge in the transformed CFG will be moved up, to the closest back edge destination. So, if there was an edge from B5 to B3 in Figure 5(b), then in the linearized CFG the back edge destination would be moved to B2_G.

The transformation converts a CFG into a sequence of if-then statements such that each branch guard block is the IPDOM of its predecessor branch guard block.

3.3 Converting Irreducible Graph to Reducible Graph

The linearization transformation can be applied to any unstructured CFGs including irreducible graphs. Figures 6(c) and (d) show an irreducible graph and its transformed version. In this case the transformed CFG has been obtained by traversing the blocks in the order B1, B2, B3. Even if they were traversed in the order B1, B3, B2, the resultant transformed CFG would still be a reducible graph. Since B1 has two successors viz., B2 and B3, the reverse post-order traversal can select any one of them and so both the orders mentioned above are possible in the reverse post-order.

3.4 Correctness of the transformation

Claim 1: The transformed CFG is structured.
Proof Sketch: To aid in the proof, we will assume that for each block B_i in the unstructured region, in addition to a guard block, a block is created as a merge point as shown in Figure 5(d). The merge block acts as the source of a back edge in case B_i is the source of a retreating edge in the input CFG e.g block B4_1_G in Figure 5(c). Otherwise the merge block is an empty block and is combined with the successor block. We call this combination of the three blocks, an *S-block*, in which the guard block is the entry block and the merge block is the exit block. So for each block B_i in the unstructured region an S-block SB_i is created in the transformed CFG. The transformed CFG can be thought of as a linearized graph of the S-blocks such that (a) if B_i is the predecessor of B_j in the reverse post-order, then SB_i is the predecessor of SB_j in the linearized CFG, and (b) if there is a retreating edge from B_j to B_i in the unstructured CFG, then a back edge from SB_j to SB_i is added in the linearized CFG. As explained before this creates either disjoint or nested loops. The CFG of the blocks in an S-block does not contain an unstructured edge. This is because (a) for any edge in the CFG either the source dominates the destination or the destination postdominates the source, and (b) the CFG has no loops. So, the CFG of the blocks in an S-block is structured. The CFG formed using the S-blocks (i.e. CFG whose nodes are S-blocks) is also a structured CFG because (a) if there are no loops in the unstructured CFG, then each S-block has only one predecessor and one successor, and (b) if there are loops in the unstructured CFG, then they are either nested or disjoint, and each loop is a natural loop [1]. These two conditions ensure that the CFG formed using the S-blocks is also structured. Since the CFG formed using the S-blocks is structured and also the CFG of the blocks in any S-block is structured, the linearized CFG is structured. ■

S-blocks which do not have back edges have empty merge blocks. Each empty merge block has only one successor. Hence it can be eliminated by connecting its predecessors to its successor. It can be seen that the resultant CFG is also structured using the same reasoning as given above. Figure 5(c) shows the S-blocks corresponding to blocks B3 and B4 labelled as SB3 and SB4 respectively. SB3 does not show the empty merge block.
Claim 2: Linearization transformation preserves semantics of the input CFG.
Proof Sketch: We make the following 4 observations regarding the transformation: (1) it does not delete any basic block from the original CFG, (2) it adds guard blocks and they do not modify any user defined variables, (3) it replaces conditional and unconditional branch statements in unstructured blocks by assignments to the guard variable and (4) it adds an assignment to the guard variable in unstructured blocks that do not have branch statements. These observations imply that if the order of execution of blocks in the original CFG is the same as in the transformed CFG then the transformation preserves the semantics of the input CFG.

First we will prove that for every execution order of blocks in the original CFG the execution order of those blocks in the transformed CFG is the same. When the control reaches a basic block, say B_i, in the original CFG there are 3 possibilities:

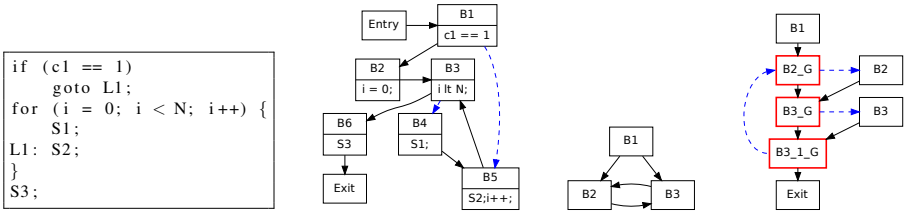
– B_i has no successors. In this case in the original CFG, the execution stops. In the transformed CFG, at the end of the execution of B_i, the guard variable is set to an unused value and hence no other blocks can execute and the execution stops.
– B_i has one successor. In this case in the original CFG, the control will transfer to the successor block. In the transformed CFG, the guard variable will be assigned the index of the successor block and hence only that block can execute next.
– B_i has two successors. In the original CFG, the branch condition at the end of B_i decides the next block to be executed. The transformed CFG sets the guard variable to the index of the next block to be executed.

This shows that if the original CFG executes block B_j after block B_i, the transformed CFG will also execute block B_j after B_i. The transformed CFG may execute one or more guard blocks between B_i and B_j, but since the guard blocks do not modify any of the original program state, their execution will not have any effect on the final output of the program. This proves that the order of execution of blocks in the original CFG remains the same after the transformations.

Next we will prove that for every execution order of the original CFG blocks in the transformed CFG, there is an equivalent order of those blocks in the original CFG. The transformed CFG has two types of blocks viz., guard blocks (GB) and original CFG blocks (OB). So we need to prove that for every execution order of the OBs in the transformed CFG, the original CFG has an equivalent execution order of them.

Consider two OBs, B_i and B_j. If the transformed CFG executes B_j after B_i with no other OB executing between them, then it means B_i sets the guard variable to the index of B_j. This is possible only if B_j is a successor of B_i in the original CFG and the branch condition in B_i evaluates to a value such that the branch to B_j is taken, so the original CFG also executes B_j after B_i.

This proves that the transformation preserves the semantics of user code. ■



(a) Jump into a loop Code Snippet (b) Jump into a loop CFG (c) Irreducible CFG (d) Reducible Transformed CFG

Fig. 6: Jump into a loop and Irreducible Graph Example

3.5 Analysis of Increase in Code Size

Claim 3: Increase in code size due to the transformation is linear in the number of blocks and instructions.

Proof Sketch: One guard block is created for (a) each basic block in the unstructured region and (b) each retreating edge in the unstructured region. Hence the total number

of new blocks added is the sum of number of blocks and number of retreating edges in an unstructured region. The number of retreating edges in a CFG cannot be more than the number of blocks in the CFG. Hence the total number of new blocks added is at most two times the total number of blocks in the original CFG.

Each guard block adds one instruction to compare the guard variable with a block index. For each block in the unstructured region with (a) a fall through edge to its successor, one instruction is added to set the guard variable, and (b) a branch edge to its successor, the branch instruction is replaced with an assignment to the guard variable. This shows that the total number of new instructions added is at most twice the number of blocks in an unstructured region. ■

In contrast to this, the transformations presented by Zhang et al. [23] can have exponential increase in the code size in their Forward Copy transformation. Even the Backward Copy transformation makes a copy of the loop to peel its first iteration. Carter et al. [5] proved that exponential blowup in the size of the CFG is unavoidable when a node-splitting technique is used to convert an irreducible flow graph to a reducible one. They have also stated that their results do not apply to techniques which use predicate variables to guard statements. Since our technique uses predicate variables to guard blocks, the results proved by Carter et al. do not apply to our technique.

3.6 Earliest Reconvergence

In this section we will show that the transformed CFG has the earliest reconvergence point for any divergent branch. The IPDOM of a guard block is the successor on its fall through edge. In Figure 5(c), threads diverging at block B2_G will reconverge at block B3_G and threads diverging at block B4_1_G will reconverge at block B5_G. So each fall through edge successor of a guard block acts as a reconvergence point and since threads can only diverge at a guard block they are immediately reconverged on the fall through edge successor. In a structured CFG, the IPDOM of a divergent branch is its earliest recovergence point. Our proposed transformation converts an unstructured CFG to a structured CFG and hence for any divergent branch, the transformed CFG has the earliest reconvergence point.

4 Minimizing Unstructured Region

In the previous section we assumed the entire CFG to be unstructured. First we propose an algorithm to find the unstructured region in the CFG. The intuition behind finding the unstructured region is to identify blocks for which the linearization transformation is applied. Further, the size of a transformed CFG linearly increases with size of the unstructured region and the unstructured CFG may contain subregions which are structured. To reduce unnecessary code bloat of structured subregions, we propose an algorithm to identify structured regions within the unstructured region.

To be able to apply the transformation only on an unstructured region, it should have a single entry point and a single exit point.

Definition 4. *The* unstructured region *of an unstructured edge is defined as a set of blocks, UR, such that*

- *it is bounded by blocks D and P, where D is the* CIDOM *and P is the* CIPDOM *of all the blocks in the set (D and P are not in the set),*

– *it contains blocks of the unstructured edge,*
– *for any edge $B_k \rightarrow B_l$, where $B_l \in UR$, either $B_k \in UR$ or $B_k = D$,*
– *for any edge $B_k \rightarrow B_l$, where $B_k \in UR$, either $B_l \in UR$ or $B_l = P$.*

Algorithm 2 Unstructured Region

```
1: procedure FindUnstructuredRegion()
2: UE = set_of_all_unstruct_edges
3: for all edge ∈ UE do
4:     UN1 ← φ
5:     UN1.insert(edge.src, edge.dst)
6:     Done ← false
7:     while Done = false do
8:         cidom = findIdom(UN1)
9:         cipdom = findIpdom(UN1)
10:        N1 ← BlksDomBy(cidom)
11:        N2 ← BlksThatCanReach(cipdom)
12:        N3 ← BlksPostDomBy(cipdom)
13:        N4 ← BlksReachableFrom(cidom)
14:        UN2 ← (N1 ∩ N2) ∪ (N3 ∩ N4)
15:        if UN2 = UN1 then
16:            Done ← true
17:        end if
18:        UN1 ← UN2
19:    end while
20: end for
21: end procedure
```

The algorithm to find the minimal unstructured regions is shown in Algorithm 2. It iterates over all unstructured edges in a CFG and finds the unstructured region for each edge. The first step is to mark the source and destination of an unstructured edge as unstructured blocks *UN1* (line 5). Then it finds their *CIDOM* and *CIPDOM* (line 8). Using *CIDOM* and *CIPDOM* as the entry and exit points of the unstructured region, the algorithm adds blocks to the region as per the two criteria: (a) blocks dominated by *CIDOM* and that can reach the *CIPDOM*, and (b) blocks postdominated by *CIPDOM* and that can be reached from *CIDOM*(lines 9-13). Since these steps can add more blocks to the set of unstructured blocks, i.e. *UN1*, they are repeated until no new blocks are added to the set of unstructured blocks.

Algorithm 2 identifies all blocks in an unstructured region. Any non-overlapping structured region is not included in the unstructured region. The unstructured regions of two unstructured edges cannot partly overlap, i.e. they are either disjoint or one will contain the other (including the case where they are the same).

4.1 Structured Region

The unstructured regions found by Algorithm 2 may contain structured sub-regions, i.e. regions with single entry and single exit, and no unstructured edge in them.

Definition 5. *A* structured region *is defined as a set of blocks, SR, such that*

– *it is bounded by blocks D and P where D is the* CIDOM *of all the blocks in SR except for D, and P is the* CIPDOM *of all the blocks in SR except for P and*
– *it contains both D and P but does not contain any unstructured edge.*

Since a structured region has only one entry block and one exit block, if control does not reach the entry block, it cannot reach any of the blocks between the entry and exit, including the exit. Also, the hardware stack based reconvergence mechanism guarantees earliest reconvergence for a structured region. Hence, the cost of linearization is reduced by guarding only the entry block of a structured region and not linearizing the structured region. This helps maintain the original structure of the CFG for the region and hence reduce the side effects of linearization on the other compilation passes. The algorithm to find structured regions is presented in Algorithm 3.

4.2 Optimizations

To further reduce the cost of guard checks, the linearization algorithm optimizes the transformed CFG to remove unnecessary guard checks, nest guard checks, etc. To be able to decide which checks can be eliminated or nested under some other checks, the transformed CFG is analyzed to find the possible values the guard variable can take at each guard block.

Algorithm 3 Structured Region

```
1: procedure FindStructuredRegion()
2: for all blk ∈ set_of_all_unstruct_blks do
3:     structRegion ← false
4:     if isImmedDom(blk) then
5:         ipdom ← immedPostDom(blk)
6:         if blk = immedDom(ipdom) then
7:             idom ← blk
8:             N1 ← visit(idom, ipdom)
9:             N1 ← N1 ∪ visit(ipdom, idom)
10:            structRegion ← true
11:            for all n1 ∈ N1 do
12:                if hasUnstructEdge(blk) then
13:                    structRegion ← false
14:                else if (ipdom ≠ pdom(n1)) then
15:                    structRegion ← false
16:                else if (idom ≠ dom(n1)) then
17:                    structRegion ← false
18:                end if
19:            end for
20:            if structRegion = true then
21:                SN ← N1
22:            end if
23:        end if
24:    end if
25: end for
26: end procedure
```

The first step is to find the guard values that can reach a guard block from its predecessor unstructured blocks. The next step is to propagate the guard values on the two successor edges of a guard block. As mentioned before, there are two types of guard blocks, viz., a guard block created for an unstructured block (GB) and a guard block created for a re-treating edge (RGB). The branch edge of a GB is to an unstructured block and is taken only when its guard check is true, which means for only one of the input guard values the branch edge is taken and all other values flow through the fall through edge. In case of a RGB, both the branch and fall-through edges are to guard blocks. The branch edge can be taken for more than one guard value (e.g., in case of loop merging in Figure 8(b), the branch edge of B6_1_G is taken if the guard value is either B2_id or B4_id), Hence all those values will flow on the branch edge and the remaining values will flow on the fall through edge.

The transformed CFG is iteratively analyzed to propagate the guard values on the input and output edges of each guard block, till no new values are seen at the input of any guard block. The iterative algorithm is guaranteed to terminate as i) the cardinality of the set of guard values at the input of a guard block in each iteration is non-decreasing, ii) once a guard value is added to the input set of a guard block, it is never removed, and iii) the cardinality of the set of all possible guard values is equal to the number of unstructured blocks, At the end of this analysis, values of the guard variable flowing on edges into and out of each guard block are known.

Now we briefly describe some optimizations to reduce the guard checks:

- (O1) If only one incoming edge of a guard block has the matching guard value (i.e. the value being checked by the guard block) then the destinations of all other incoming edges are changed to the next guard block to avoid execution of the guard check on paths containing those edges. Similarly, if only one incoming edge of a

guard block does not have the matching guard value, then the destination of that edge is changed to the next guard block.

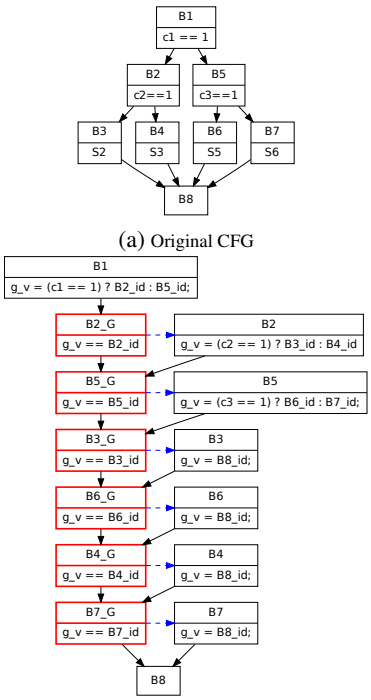
- (O2) If block B_i dominates block B_j in the original CFG, then the guard block for B_j can be nested within the guard block for B_i so that the guard check for B_j is executed only if the guard check for B_i is true.
- (O3) If a guard block has only one predecessor and if that predecessor is a block from an unstructured region, then the guard block is merged with its predecessor.

5 Applications of Linearization

In this section we will discuss some applications of the linearization transformation.

5.1 Branch Interleaving

As discussed in the motivation section, existing hardware reconvergence stack mechanism using IPDOM forgoes the potential parallelism achievable by interleaved execution of other basic blocks. We propose to use the linearization transformation to exploit parallelism among the two paths of a divergent branch.



(a) Original CFG

(b) Transformed CFG

Fig. 7: Branch Interleaving Example

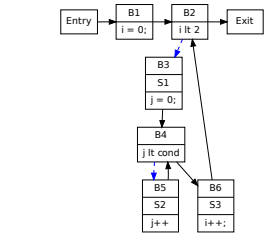
Consider the example in Figure 2(a). The original CFG and the transformed CFG are shown in Figures 7(a) and 7(b) respectively. The transformed CFG has basic blocks from the two arms of the branch statement interleaved. A thread executing the branch statement will execute blocks from either of the two arms. Assuming stall-on-use model, i.e. a core stalls when the value needed is not available in the register, we can identify blocks with potential stalls. For example, if block B2 loads a variable which is used in blocks B3 and B4, then a core can stall while executing an instruction that uses the variable, if the load instruction results in a cache miss. As shown in Figure 7(b), block B5 is inserted between blocks B2 and B3. So, if some threads of a warp take B1→B2 path and others take B1→B5 path, then executing instructions from block B5 after block B2 can help hide the long latency of a load in block B2. In contrast, existing hardware support for branch execution always traverses the blocks in the depth-first order until the IPDOM and hence incur the stalls.

Linearization can also be used to reduce stalls in a block which loads a variable as well as uses it. For example, if block B6

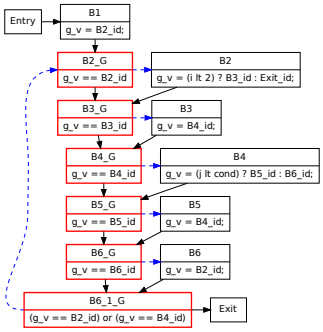
loads a variable and uses it, then it can be split into two sub blocks, B6_1 and B6_2, such that the load instruction is in B6_1 and use of the variable is in B6_2. The execution of these two blocks then can be separated by inserting one or more blocks from the other arm of the branch instruction in B5.

5.2 Loop Merging

In nested loops, if the threads of a warp have different trip counts for the inner loop then threads with smaller trip counts will have to wait for the thread with the largest trip count.



(a) Original CFG



(b) Transformed CFG

Fig. 8: Loop Merging Example

We can use the linearization transformation to let the threads with smaller trip counts proceed further and join the remaining threads in the execution of the inner loop but for a different invocation of the inner loop.

Consider the nested loop example in Figure 2(b). The original CFG and the transformed CFG after linearization are shown, respectively, in Figure 8(a) and (b). The back edge from basic block B6_1_G to basic block B2_G is for both the inner as well as the outer loop. The value of g_v is set to B2_id when a thread finishes the inner loop and is going to start the next iteration of the outer loop, whereas g_v is set to B4_id to continue with the next iteration of the inner loop. So when a thread finishes execution of the inner loop, it will execute basic blocks B2 and B3, and then join the remaining threads to execute the inner loop again. This way, different invocations of the inner loop can be overlapped to reduce the waiting time of threads and improve performance.

5.3 Hardware Stack Depth Reduction

The proposed linearization technique can be used to reduce the nesting depth of branches and hence the depth of the hardware stack used for reconvergence. Assuming the reverse post order traversal of the CFG in Figure 3(a) to be B1, B2, B4, B6, B7, B8, B5, B9, B3 and B10, it can be seen that threads diverging at block B2_G will reconverge at block B4_G and hence the depth of the reconvergence stack at B4_G will be the same as the depth at B2_G. Proceeding further, we see that the stack depth does not increase. Hence, to restrict depth of the reconvergence stack to a certain limit, branches beyond that nesting depth limit can be linearized.

Table 1: Linearization Transformation Statistics, PTX-Number of PTX Instructions, BB-Number of Basic Blocks, Reg-Number of registers used by the compiled code, SASS-Number of instructions in the assembly code, Bf-before transformation, Af-after transformation, AfOpt-after optimizing the transformed code, Incr-Increase(=AfOpt/Bf)

BM	PTX				BB				Reg			SASS		
	Bf	**Af**	**AfOpt**	**Incr**	**Bf**	**Af**	**AfOpt**	**Incr**	**Bf**	**AfOpt**	**Incr**	**Bf**	**AfOpt**	**Incr**
`hotspot` [3],[4]	269	289	275	1.02	19	30	20	1.05	30	30	1.00	383	390	1.02
`hearwall` [3],[4]	1422	1442	1432	1.01	192	206	196	1.02	32	32	1.00	2667	2681	1.01
`mcx` [9]	1358	1447	1408	1.04	138	185	148	1.07	57	63	1.10	1139	1252	1.10
`mum` [2]	232	259	256	1.10	37	51	47	1.27	22	32	1.45	202	226	1.12
`nqueen` [2]	164	175	169	1.03	30	37	31	1.03	16	18	1.12	145	148	1.02
`particlefilter` [3],[4]	52	63	54	1.04	10	17	10	1.0	13	13	1.00	52	51	0.98
`ray` [2]	780	869	805	1.03	84	148	90	1.07	43	50	1.16	933	966	1.03

6 Experimental Evaluation

We evaluated our proposed algorithm by implementing it in the Ocelot [8] compiler framework. The transformation is done at the PTX (version 2.3) IR level. The CFG constructed by Ocelot front end is transformed into a linearized CFG and then the modified PTX code is JIT compiled. We used CUDA toolkit version 4.2 [18] and Tesla C2070 GPU (Fermi) [19]. The CUDA code was compiled with the default optimization level. Each benchmark was run 10 times and the average of the execution time is reported. We used CUDA profiler to measure the runtime and other performance counters.

The proposed transformation avoids duplicate execution of basic blocks and also ensures early reconvergence which are the primary benefits of converting an unstructured CFG to a structured CFG. Further the transformation is expected to reduce the number of global loads and stores. These improvements come, however, at the cost of increased code size. We report these performance metric in our experimental framework.

We compared number of PTX instructions (PTX), number of basic blocks (BB), number of registers used (Reg) and number of assembly instructions (SASS) in the original and transformed code. Table 1 shows the increase in number of PTX instructions per kernel. It is less than 5% for 6 out of 7 benchmarks. The maximum increase in code size is 10% in *mum* benchmark. Table 1 also shows the increase in number of basic blocks per kernel. It is less than 10% for 5 out of the 7 benchmarks and a maximum of 27% in *mum*. The improvements with the optimizations are shown in column AfOpt. Even though the upper bound for increase in code size is linear in terms of the number of basic blocks, the observed increase is less than 7% on an average.

Out of the 7 benchmarks, 4 show an increase of 10% or more in the number of registers. This is one of the major side effects of doing the transformations at the PTX level. In the next subsection we discuss how this transformation can be implemented at a lower level of IR. The increase in the number of assembly instructions is up to 12%.

Table 2 shows the runtime performance numbers measured using CUDA profiler counters. Except for the execution time, other metrics reported are aggregate numbers for all threads on all SMs. For benchmarks *mcx* and *mum* the number of global loads decrease by 4.4% and 48.5% respectively. Also the number of global stores decrease for benchmarks *mcx* (13.3%), *mum* (68.7%) and *heartwall* (2.5%). Benchmark *mum* also shows an improvement of 17.5% in the number of dynamic instructions executed.

Table 2: Runtime profile per kernel, ExecTime - execution time in micro seconds, InstExec - number of assembly instructions executed, GlobalLd - number of global load instructions executed, GlobalSt - number of global store instructions executed, Before - before transformation, AfterOpt - after optimizing the transformed code

BM	ExecTime(us)		InstExec		GlobalLd		GlobalSt	
	Before	AfterOpt	Before	AfterOpt	Before	AfterOpt	Before	AfterOpt
hotspot	342	360	4.32×10^6	4.57×10^6	2.92×10^4	2.92×10^4	1.1×10^4	1.1×10^4
heartwall	4.22×10^4	4.23×10^4	4.68×10^8	4.71×10^8	3.21×10^7	3.21×10^7	7.82×10^6	7.62×10^6
mcx	3.15×10^6	3.97×10^6	2.64×10^{10}	3.05×10^{10}	2.48×10^8	2.37×10^8	8.06×10^7	6.99×10^7
mum	2242	2224	7.80×10^6	6.43×10^6	1.69×10^5	8.76×10^4	5.99×10^4	1.87×10^4
nqueen	112	110	5.94×10^4	5.96×10^4	3	3	256	256
particlefilter	187	205	1.02×10^5	1.49×10^5	1.85×10^4	1.85×10^4	64	64
ray	163	173	1.82×10^6	1.89×10^6	2048	2048	4096	4096

Table 3: Increase in Number of PTX instructions, Orig - Before transformation, Ocelot - After transformations proposed by Wu et al. [22], Linear - After our transformations

BM	Orig	Ocelot	Linear
hotspot	237	242	240
heartwall	1422	1452	1432
particlefilter	54	78	62

Table 4: Reduction in number of instructions executed, Orig - Before transformation, After - After transformations without any optimizations, O1 to O4 are the optimization levels

BM	Orig	After	O1	O2	O3	O4
hotspot	4.32×10^6	4.90×10^6	4.98×10^6	4.88×10^6	4.76×10^6	4.57×10^6
heartwall	4.68×10^8	4.69×10^8	4.69×10^8	4.69×10^8	4.68×10^8	4.71×10^8
mcx	2.64×10^{10}	3.7×10^{10}	3.36×10^{10}	3.26×10^{10}	3.18×10^{10}	3.05×10^{10}
mum	7.80×10^6	6.90×10^6	6.66×10^6	6.66×10^6	6.60×10^6	6.43×10^6
nqueen	5.94×10^4	6.01×10^4	6.05×10^4	6.05×10^4	6.03×10^4	5.96×10^4
particlefilter	1.02×10^5	1.77×10^5	2.14×10^5	1.86×10^5	1.68×10^5	1.49×10^5
ray	1.82×10^6	2.50×10^6	2.16×10^6	2.13×10^6	2.09×10^6	1.89×10^6

We analyzed the slowdown in *mcx* and found it to be due to the increase in number of registers from 57 to 63. Since the register allocator cannot use more than 63 registers on the GPU used in our experiments, code is introduced to spill registers to global memory. This increases the number of cache misses and the load on the memory system. Benchmark *mum* also has an increase in the number of registers from 22 to 32 and hence the occupancy drops from 0.833 to 0.667 reducing the performance improvement in spite of a significant reduction in the numbers of global loads and stores.

Table 3 shows the comparison with the algorithm by Wu et al. [22]. We could get only 3 benchmarks working with their implementation in the Ocelot framework. Since our transformation does not duplicate user code, the increase in code size is less than due to their transformation. For benchmarks *hotspot* and *particlefilter* we had to use CUDA toolkit version 4.0.

Table 4 shows the effect of our proposed optimizations. Optimization level O4 has, in addition to the three optimizations described in section 4.2, some miscellaneous optimizations. As shown in the table, each of these optimizations helps in reducing the number of instructions executed. Higher optimization levels include the optimizations done by the lower optimization levels, e.g. O3 has, in addition to the optimizations done by O1 and O2, the optimization to merge guard blocks with their predecessors.

We used PTX as the IR because of the availability of its documentation and the Ocelot [8] compilation framework. But ideally this transformation should be done as late in the compilation process as possible to avoid its side effects on flow analyses, optimizations, register allocation, etc. Implementing the transformation at a lower level IR can reduce the major side effect of increase in register pressure and hence reduction in the occupancy, e.g. benchmarks *mum* and *mcx* are severely affected because of the increase in number of registers. Unfortunately, there is not enough information in public domain, about assembly level instructions of NVIDIA GPUs and hence we could not implement the linearization algorithm at that level.

The proposed transformation can be implemented at the assembly level with one additional integer register needed to hold the guard variable and one additional predicate register needed to hold result of the guard check (in case of loop merging, an additional predicate register per loop to be merged is needed). To make sure that the transformation will have enough registers left, the register allocator can be restricted to use that many fewer registers. We also believe that the costs and benefits of linearization can be estimated more accurately at an assembly level IR than at PTX IR.

7 Related Work

Wu et al. [22] implemented a transformation pass at the PTX level, to convert unstructured control flow to structured control flow. The transformations are equivalent to the ones used in Zhang's [23] work. These transformations duplicate user code, whereas, our proposed transformations do not. Thread Frontier [7] uses a combined hardware and software solution to handle unstructured control flow. It identifies the thread frontier of each basic block and using extra hardware prioritizes basic blocks and checks for stalled threads waiting in the thread frontiers. Our proposed solution does not need any hardware support and it uses predicated execution to linearize CFGs.

Han et al. [12] have proposed a compiler transformation to merge a divergent loop with one or more outer surrounding loops into a single loop. Even though they also transform the CFG to achieve loop merging, our algorithm uses the idea of linearization to reconverge threads. Stratton et al. [20], Wang et al. [21] and Coutinho et al. [6] discuss various compile time analyses to identify non-divergent branches which can be used to skip linearization of non-divergent branches.

Rhu et al. [16] suggested a dual-path stack to keep the two divergent paths of a branch in parallel. This enables interleaved execution of threads from both the paths. Dynamic warp formation [10] regroups threads dynamically into new warps based on their next program counter values. Dynamic Warp subdivision [15] exploits intra-warp latency hiding, by dynamically subdividing warps and allowing them to run ahead. Thread block compaction [11] uses a common block-wide stack for divergence handling. New warps are formed from threads of a block at divergent branches. They also suggest using likely convergence points to converge threads earlier than IPDOM.

The work on obfuscating C++ programs via control flow flattening [14] converts the high level constructs into if-then-goto constructs and changes the target addresses of goto statements so that they will be determined dynamically.

8 Conclusion

In this paper, we presented a simple and elegant transformation to handle the performance problems arising due to branch divergence in GPUs. We showed that the transformation converts an unstructured CFG to a structured CFG with linear increase in the number of instructions. We also discussed three applications of the transformations viz., branch interleaving, loop merging and reduction in reconvergence stack depth. We described the implementation of this technique at the PTX IR level with only up to 10% increase in code size. As future work, we will use a lower level IR and also develop heuristics for its various applications.

Acknowledgements. We thank the anonymous reviewers for their suggestions and comments. We also thank Vaivaswatha N. and other members of the Lab for HPC for discussions and feedback on improving the paper. The first author acknowledges the funding received under Google India Ph.D. Fellowship.

References

1. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers Principles, Techniques and Tools*. Second Edition, Pearson.
2. A. Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt. *Analyzing CUDA workloads using a detailed GPU simulator*. In ISPASS, 2009.
3. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, K. Skadron. *Rodinia: A Benchmark Suite for Heterogeneous Computing*. In IISWC, 2009.
4. S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, K. Skadron. *A Characterization of the Rodinia Benchmark Suite With Comparison to Contemporary CMP Workloads*. In IISWC, 2010.
5. L. Carter, J. Ferrante, C. Thomborson. *Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger*. In POPL 2003.
6. B. Coutinho, D. Sampaio, F. M. Q. Pereira, W. Meira Jr. *Divergence Analysis and Optimizations*. In PACT 2011.
7. G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, J. Wu, S. Yalamanchili. *SIMD Re-Convergence At Thread Frontiers*. In MICRO 2011.
8. G. Diamos, A. Kerr, S. Yalamanchili, N. Clark. *Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems*. In PACT, 2010.
9. Q. Fang and D. A. Boss. *Monte Carlo Simulation of Photon Migration in 3D Turbid Media Accelerated by Graphics Processing Units*. In Optics Express, vol. 17, issue 22, pp. 20178-20190 (2009).
10. W. W. L. Fung, I. Sham, G. Yuan, T. M. Aamodt. *Dynamic warp formation and scheduling for efficient gpu control flow*. In MICRO, 2007.
11. W. W. L. Fung, T. M. Aamodt. *Thread block compaction for efficient simt control flow*. In HPCA, 2011.
12. T. D. Han, T. S. Abdelrahman. *Reducing Divergence in GPGPU Programs with Loop Merging*. In GPGPU 2013.
13. OpenCL. *www.khronos.org/opencl*.
14. T. László, Á. Kiss *Obfuscating C++ programs via control flow flattening*. Annales Universitatis Scientarum Budapestinensis de Rolando Ëotv̈os Nominatae, Sectio Computatorica, 30:3–19, August 2009
15. J. Meng, D. Tarjan, K. Skadron. *Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance*. In ISCA, 2010.
16. M. Rhu, M. Erez. *The Dual-Path Execution Model for Efficient GPU Control Flow*. In HPCA 2013.
17. Nvidia. *CUDA C Programming Guide*. Oct 2010.
18. Nvidia. *https://developer.nvidia.com/cuda-toolkit-42-archive*.
19. Nvidia. *www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf*.
20. J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, W. W. Hwu. *Efficient Compilation of Fine-Grained SPMD-threaded Programs for Multicore CPUs*. In CGO 2010.
21. S. Wang, M. Hung, Y. Hwang, R. D. Ju, J. Lee. *Pointer Based Divergence Analysis in the SSA Form*. In CPC, 2013.
22. H. Wu, G. Diamos, S. Li, S. Yalamanchili. *Characterization and Transformation of Unstructured Control Flow in GPU Applications*. In The First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems (CACHES), June 2011.
23. F. Zhang and E. H. D'Hollander. *Using hammock graphs to structure programs*. In IEEE Trans. Softw. Eng., pages 231-245, 2004