Efficient Control Flow Restructuring for GPUs

Nico Reissmann, Thomas L. Falch
and Benjamin A. Bjørnseth
Dept. of Computer and Information Science
NTNU, Norway
{reissman, thomafal, benjambj}@idi.ntnu.no

Helge Bahmann
Google Zürich
hcb@chaoticmind.net

Jan Christian Meyer
and Magnus Jahre
Dept. of Computer and Information Science
NTNU, Norway
{janchris, jahre}@idi.ntnu.no

***Abstract*—The CUDA and OpenCL programming models have facilitated the widespread adoption of general-purpose GPU programming for data-parallel applications. GPUs accelerate these applications by assigning groups of threads to SIMD units, which execute the same instruction for all threads in a group. Individual group threads might diverge and follow different paths of execution. Divergent branches cause performance degradation by under-utilizing the execution pipeline, resulting in a major performance bottleneck. The presence of unstructured control flow in addition to divergent branches causes further degradation, since it results in repeated execution of instructions.**

In this paper, we propose a transformation which converts unstructured to structured control flow. It only creates tail-controlled loops, and properly nests all control flow splits and joins by inserting predicates. We implement an additional pass to NVIDIA's CUDA compiler to experimentally evaluate our transformation using synthetic unstructured control flow graphs, as well as kernels in the Rodinia benchmark suite. Our approach effectively eliminates redundant execution and potentially improves execution time for the synthetic unstructured control flow graphs. For the kernels in the benchmark suite, it only adds a minor, average overhead of 2.1% to the execution time of already structured kernels, and reduces execution time for the only unstructured kernel by a factor of five. The representational overhead at compile-time is linear in terms of instructions.

***Keywords*-GPGPU, Unstructured Control Flow, Control Flow Graph, Control Flow Restructuring, Branch Divergence**

I. Introduction

Programming models such as CUDA [1] and OpenCL [2] allow developers to port applications to Graphic Processing Units (GPUs) and use their computing power for general purpose processing (GPGPU). GPUs accelerate data-parallel applications by mapping groups of threads to parallel execution units. These thread groups run in lock-step, executing the same instruction in Single Instruction Multiple Data (SIMD) mode. Individual threads in a group can *diverge* by following different paths of execution. Current GPUs handle these divergent branches by executing all paths sequentially, and masking out threads that do not take a path. Divergent threads reconverge at the immediate post-dominator (IPDOM)[1] of the branch instruction [3].

Branch divergence causes performance degradation by under-utilizing the execution pipeline. Moreover, IPDOM is the earliest point of reconvergence for structured control flow graphs (CFGs), but can result in redundant basic block execution with unstructured control flow. Branch divergence is therefore a major performance bottleneck [4], [5], [6], [7], exacerbated by unstructured control flow. The causes of unstructured control flow are programming language constructs such as goto, switch, and break statements, short circuiting operations, and compiler optimizations such as function inlining. Transforming unstructured control flow eliminates the redundant execution caused by divergence, mitigating its performance penalty. Moreover, compilers targeting AMD GPUs represent programs in the AMD IL [8] intermediate representation (IR). In contrast to NVIDIA's PTX [9], it only supports structured control flow, making transformations necessary.

In this paper, we propose a transformation to convert unstructured to structured control flow. It is based on the work from Bahmann *et. al.* [10] and consists of two phases: *loop restructuring* and *branch restructuring*. Loop restructuring converts all cyclic structures to tail-controlled loops, while branch restructuring ensures proper nesting of control flow splits and joins. This transformation works by adding predicates and branches to CFGs. We modify the algorithm to admit head-controlled loops, and separate the implementation of the loop and branch restructuring phases. This seperation is possible because the insertion order of additional predicates and branches is irrelevant for GPUs. In contrast to previous solutions [11], [12], [6], [13], the use of predication instead of node splitting for restructuring CFGs avoids the risk of exponential code inflation [14].

We implement control flow restructuring as an additional pass to NVIDIA's CUDA compiler, and evaluate it experimentally using synthetic unstructured control flow graphs (CFGs), as well as all kernels in the Rodinia benchmark suite [15]. The synthetic unstructured CFGs demonstrate that our approach effectively eliminates redundant execution and potentially improves execution time for unstructured graphs with branch divergence. We use the Rodinia benchmark suite to evaluate transformation cost in terms of execution time and representational overhead at compile-time. Control flow restructuring adds a minor average overhead of 2.1% to the execution time of already structured kernels, and reduces execution time for the only unstructured kernel by a factor of five. While the overhead for already structured kernels is notable, it is significantly lower than previously reported results [11], [12]. The representational overhead at compile-

[1]See Section III for a definition of IPDOM

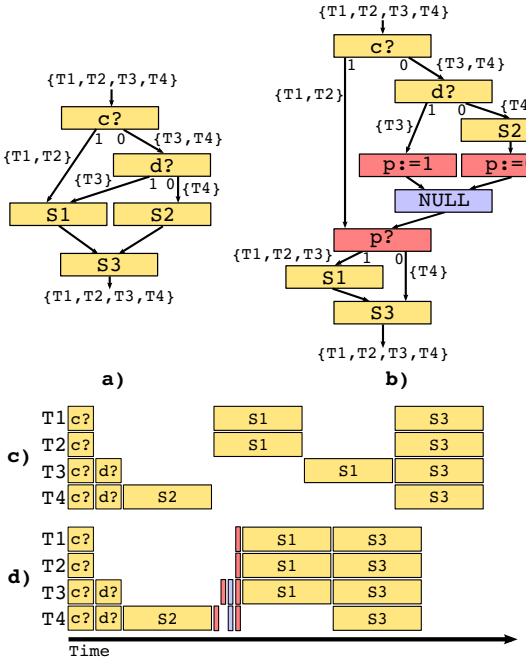

Fig. 1. Example illustrating the negative effect of unstructured control flow in the presence of branch divergence. **a)**: CFG for the pseudocode in Section II. **b)**: Control flow restructured CFG from a). **c)**: Possible execution schedule for the CFG in a). **d)**: Possible execution schedule for the CFG in b).

time is linear in terms of instructions.

The paper is organized as follows: Section II describes the problem of branch divergence for unstructured control flow. Section III introduces terminology and definitions, while Section IV describes our algorithm. We empirically evaluate it using synthetic unstructured CFGs and the Rodinia benchmark suite [15] in Section V. Section VI discusses related work, and Section VII concludes and suggests further directions for research.

II. Motivation

The IPDOM of a branch is the earliest possible point of reconvergence in structured CFGs, causing no redundant execution. In unstructured graphs, however, it is possible to introduce earlier points of reconvergence in order to avoid multiple executions of basic blocks. The following pseudocode shows a simple if-then-else statement with a short circuited condition:

```
if (c || d) {
  S1;
} else {
  S2;
}
S3;
```

Figure 1a depicts the corresponding CFG. The CFG is unstructured due to splits and joins not being properly nested.

Consider a warp of four threads executing this code segment, with threads $T1$ and $T2$ taking execution path $(c?, S1, S3)$, thread T3 execution path $(c?, d?, S1, S3)$, and thread $T4$ execution path $(c?, d?, S2, S3)$ (see Figure 1a). The threads only reconverge before executing basic block $S3$. Thus, the basic block $S1$ would be executed twice, once for threads $T1$ and $T2$, and once for thread $T3$ as shown in the example schedule in Figure 1c.

Figure 1b depicts the CFG after control flow restructuring. The basic idea is to insert predicate assignments ($p := 0$ and $p := 1$) and branches ($p?$) such that all splits and joins are properly nested, and the resulting CFG is structured. This results in threads $T3$ and $T4$ reconverging at $NULL$ and threads $T1$, $T2$, $T3$, and $T4$ at $p?$, avoiding the duplicated execution of $S1$ as shown in the schedule of Figure 1d. The problem of repeated basic block execution compounds in bigger subgraphs, possibly resulting in more than two executions of individual nodes.

Structured graphs do not result in redundant code execution on GPUs, because nested divergent branches always reconverge in the inverse order of their execution, *i.e.* the inner branch reconverges before the outer branch. Our transformation converts kernels to structured graphs which consist only of tail-controlled loops and properly nested control flow splits and joins. For tail-controlled loops, divergent branches reconverge at the loop's epilogue, while divergent splits reconverge at the corresponding join. Thus, our transformation always produces graphs which preclude redundant code execution.

Developers are aware of the potential disadvantages of unstructured control flow for GPUs, and therefore try to avoid it. A compiler supporting control flow restructuring in combination with divergence analysis [16] would allow a greater class of programs to be automatically translated into efficient GPU code.

III. Terms and Definitions

A *control flow graph* is a directed graph consisting of nodes containing statements and edges representing transitions between statements. Outgoing edges are numbered with unique consecutive indices starting from zero (although we will omit writing out the index if a node has only one outgoing edge). Statements take the following form:

- $v := expr$ designates an assignment statement. The right hand side expression is evaluated and the result is assigned to the variable named on the left.
- $v?$ designates a branch statement. The variable is evaluated and execution resumes at the node reached through the correspondingly numbered edge.
- Other kinds of statements corresponding to original program behavior (observable side-effects) are allowed as well. We omit their discussion, as they are irrelevant for the control flow behavior discussed in this paper.

Only branch statements may have more than one outgoing edge. Furthermore, we require that each CFG has two designated nodes: The *entry node* without a predecessor, and the *exit node* without a successor. CFGs represent programs in imperative form: Starting at the entry node, successively evaluate each statement, until reaching the exit node.

Nodes are generally denoted by n with sub- and superscripts. Edges denoted by e with sub- and superscripts. An edge from a node n_1 to a node n_2 is written as $n_1 \to n_2$. We call n_1 the edge's *source* and n_2 its *target*.

Definition 1: A CFG is called *single-entry/single-exit (SESE)* if its shape can be contracted into a single node by repeatedly applying the following steps:

1) If n' is unique successor of n, and n is unique predecessor of n', then remove edge $n \to n'$ and merge n and n'.
2) If n has only successors $n_0, n_1, \ldots$ and possibly n', n' has only predecessors $n_0, n_1, \ldots$ and possibly n, and each of $n_0, n_1, \ldots$ has only a single predecessor and successor, then remove all arcs $n \to n_i$, $n_i \to n'$, $n \to n'$ and merge all vertices.
3) If n has an edge pointing to itself and only one other successor, remove the edge $n \to n$.
4) If n has an outgoing edge targeting n' and n' has only one outgoing edge targeting n, then remove $n \to n'$, n', and $n' \to n$.

Single-entry/single-exit CFGs are a subset of reducible CFGs and can be characterized by allowing only the following constructs:

- Straight line code.
- Properly nested conditionals ("`if/then/else`" or "`switch/case`" statements without fall-throughs).
- Tail-controlled loops ("`do/while`" loops without "`break`" or "`continue`" statements).
- Head-controlled loops ("`for`" or "`while`" loops without "`break`" or "`continue`" statements).

Definition 2: A CFG is called *tail-structured* if its shape can be contracted into a single node by repeatedly applying rule 1, 2, and 3 from Definition 1.

Tail-structured CFGs are a subset of SESE CFGs and correspond to programs with only straight line code, properly nested conditionals, and tail-controlled loops.

Definition 3: A CFG is called *linear* if every vertex has exactly one incoming and outgoing edge.

Linear CFGs are a subset of tail-structured CFGs and correspond to programs with only straight line code.

Definition 4: A CFG is called *minimal* if it does not contain any nodes n and n' such that n' is the unique successor of n, and n the unique predecessor of n'. Thus, a minimal CFG contains no linear subgraphs.

Definition 5: An edge $n_1 \to n_2$ *dominates* node n if

1) $n_2 \neq n$ and both n_1 and n_2 dominate n, or
2) $n_2 = n$ and n_1 dominates n.

The dominator graph of edge e is the subgraph of all nodes dominated by e.

Intuitively, the dominator graph of an edge e is the subgraph where every path from the entry node to every node in this subgraph must pass through edge e.

Definition 6: A node n' is said to be the immediate post-dominator (IPDOM) of another node n iff:

- n' post-dominates each immediate successor $n_0, n_1, \ldots$ of n, and
- for each other node n'' which also post-dominates each of $n_0, n_1, \ldots$ it holds that either $n'' = n'$ or that n'' post-dominates n'.

Intuitively, the immediate post-dominator is the earliest point in a CFG where all paths starting at some node n necessarily reconverge.

IV. Control Flow Restructuring

Regularization of control flow can be facilitated by node cloning, predication or a combination of both techniques. Here we follow the approach laid out by Bahmann *et al.* [10] using predication only. Assume a CFG with a single entry and exit node. We restructure it using the procedures described below. The approach consists of two phases:

- Loops are detected and (possibly) transformed into a tail-controlled loop.
- Branches are restructured such that branch and join points are symmetric.

A. Loop Restructuring

We start by identifying all strongly connected components (SCCs) and process each of them according to the procedure below. By necessity, neither entry nor exit node are part of any SCC. First, identify the following nodes and edges:

- *Entry edges* $e_0^E, e_1^E, \ldots$: All edges from a node outside the SCC into the SCC
- *Entry nodes* $n_0^E, n_1^E, \ldots, n_{k-1}^E$: All nodes that are target of at least one entry edge
- *Exit edges* $e_0^X, e_1^X, \ldots$: All edges from a node inside the SCC out of the SCC
- *Exit nodes* $n_0^X, n_1^X, \ldots, n_{l-1}^X$: All nodes that are target of at least one exit edge
- *Repetition edges* $e_0^R, e_1^R, \ldots$: All edges inside the SCC that have one entry node as target

See Figure 2 for illustration. We denote the set of nodes belonging to SCC by L. Initially, L induces the SCC subgraph. The following modifies the original graph, and we updates L as well such that it eventually induces a suitable structured loop subgraph. When we say "create a node within L" (as opposed to just "create a node") in the following, it means: Create the node in the CFG and update L such that it also has this node as member.

1) Pick two unused variables q and r to identify continuation location and loop repetition state, respectively.
2) If there are multiple entry nodes:
   a) Create a branch node b^E within L that evaluates q and continues at e_m^E iff $q = m$.
   b) Replace each entry edge: If the edge originally pointed to e_m^E, create an assignment statement $q := m$, divert the original entry edge to it, and continue control flow to b^E from there.
   c) Replace each repetition edge: If the edge originally pointed to e_m^E, create an assignment node $q := m$

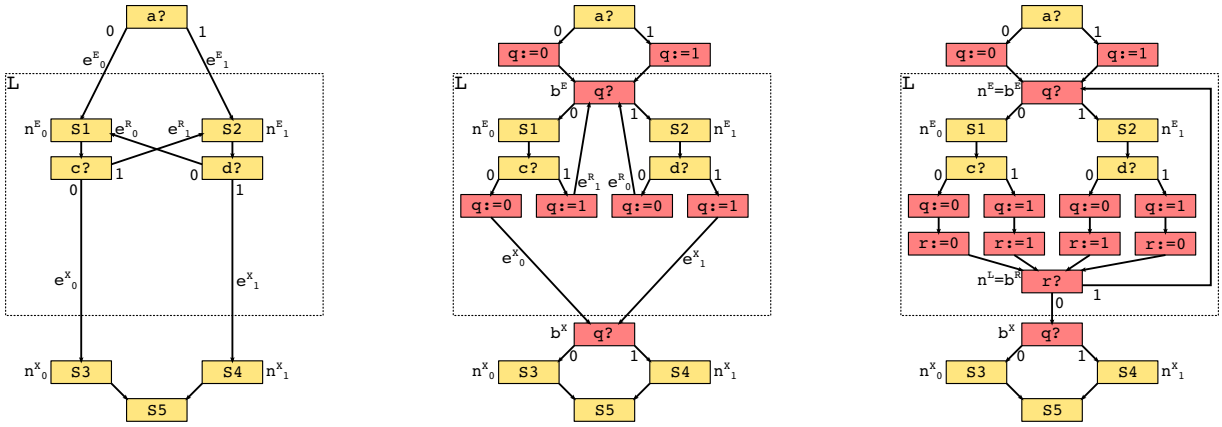

Fig. 2. Loop restructuring. **Left**: A CFG with an unstructured loop. Nodes and subgraphs identified by restructuring algorithm are marked as per algorithm description. **Center**: Intermediate state of loop restructuring after remodeling entry/exit control flow. **Right**: Final state after converting the loop to a single repetition and exit edge. Inserted nodes remodeling the original control flow are marked in red.

within L, divert the original repetition edge to it, and continue control flow to b^E from here. Record the newly recorded edges as repetition edges in lieu of the replaced ones.

After this step there is only one entry node: Either the newly created node b^E or the single original entry node. Denote it by n^E.

3) If there are multiple exit nodes:
   a) Insert a branch node b^X that evaluates q and continues at e^X_m iff $q = m$.
   b) Replace each exit edge: If the edge originally pointed to e^X_m, create an assignment node $q := m$ within L, divert the original repetition edge to it, and continue control flow to b^X.

   After this step there is only one exit node: Either the newly created node b^X or the single original exit node. Denote it by n^X.
4) If there are any two distinct nodes that are origin of either repetition and/or exit edges[2]:
   a) Create a branch node b^R within L evaluating r that continues at n^X if $r = 0$ and at n^E otherwise.
   b) Create an assignment node $r := 0$ within L and an edge from it to b^R. Divert all exit edges to it.
   c) Create an assignment node $r := 1$ within L, create an edge from it to b^R. Divert all repetition edges to it.

Note that the algorithm above does not actually modify the graph if it is already tail-structured. After this processing is complete, L contains two marked nodes:

- n^E is the unique entry node; all edges from outside L into L will have this node as target
- n^L is the unique last node; there is only one edge leaving L, it originates in n^L

n^E has only one predecessor node within L, the node n^L. The edge $n^L \to n^E$ is the unique repetition edge of this loop.

[2]Note that this includes the cases of two or more repetition or exit edges

Temporarily remove this repetition edge, keeping track of the two nodes it used to connect. We repeatedly apply this whole loop transformation algorithm for any other SCC in the graph.

After all SCCs have been transformed as above, the resulting graph is acyclic. We process this acyclic graph according to the algorithm in the next section, and then re-insert all repetition edges that were set aside.

B. Branch Restructuring

First, construct the "head" subgraph H as follows: Add the entry node to H. If the last node added has exactly one outgoing edge, add it as well as its target node to H. There are now two cases to consider:

- H covers the entire original graph.
- H covers only a portion of the original graph. There is a node b that was added to H last that has at least two outgoing edges.

In the first case the algorithm terminates: The original CFG is linear.

In the second case, record the outgoing edges of b as $f_0, f_1, \ldots, f_{m-1}$. Compute the dominator graphs of each f_k as B_k: This is the set of nodes and their connecting edges reachable from the entry node only through f_k. We call these the "branch" subgraphs. Record the remaining nodes and their connecting edges as the "tail" subgraph T. Some B_k may be empty, the corresponding edge f_k would in this case go directly to some node in T; in this case, create a "dummy" node in B_k and route the path through it. (See left of Figure 3 for illustration.)

We denote by $c_0, c_1, \ldots, c_{n-1}$ the *continuation points* in the tail subgraph: These are the nodes with T with at least one edge from either branch subgraph. There must be at least one such continuation point, and if there is exactly one then this branching construct has already a suitable structure. Otherwise, restructure T and B_k as follows:

- Choose an unused auxiliary variable, denote it by p.
- Turn T into a graph with a single entry point e: Set up branches such that control resumes at c_k if p evaluates to

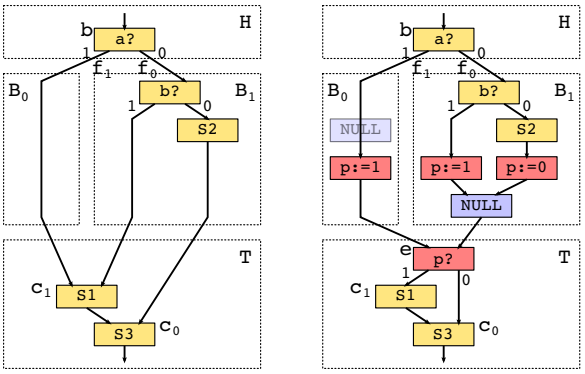

Fig. 3. Branch restructuring. **Left**: A CFG with unstructured branches. Nodes and subgraphs identified by restructuring algorithm are marked as per algorithm description. **Right**: The result of restructuring the CFG on the left hand side. Inserted nodes remodeling the original control flow through predication are marked in red. Dummy nodes inserted for structural purposes are marked in blue.

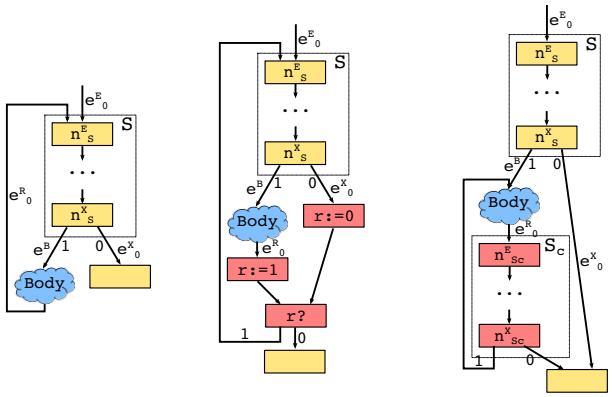

Fig. 4. Loop Restructuring with Copying. **Left**: A head-controlled loop. Nodes, edges, and subgraphs are labeled as identified by the algorithm. **Center**: The result of loop restructuring on a head-controlled loop. **Right**: The result of loop restructuring with copying on a head-controlled loop.

k on entry.

- Turn each B_j into a graph with a single exit point: Divert edges pointing to c_k into statements that assign k to p, rejoin control flow for this in a single node that then proceeds to e.

Now, all edges leaving any B_j point to e. Finally, if some B_j has multiple exit paths, join all these paths into a single "dummy" node within B_j, and create a single exit edge from this node to e.

Recursively apply the same algorithm to each B_j as well as T. The control flow of the resulting graph is then tail-structured.

In the specification above, we utilized "n-way" branches: a single variable is used to identify one of n possible branch destination points. A subsequent pass can introduce additional auxiliary variables to reduce these constructs to 2-way branches. Additionally, superfluous dummy nodes in straight line code can be eliminated.

C. Loop Restructuring with Copying

The loop restructuring algorithm in Section IV-A transforms all loops into tail-controlled loops by inserting additional branches and assignments. Only loops that are already tail-controlled, *i.e.* do-while loops, are not altered. However, programmers express loops commonly as head-controlled loops, *i.e.* for and while loops. Loop restructuring would restructure these loops and introduce additional overhead. Figure 4 shows a simple head-controlled loop on the left and its equivalent after loop restructuring in the middle. The algorithm transforms an unconditional branch to a conditional one, and inserts two assignments, one of them being executed every loop iteration. This could potentially lead to overhead in execution time (see Section V).

In order to mitigate the effect of loop restructuring on head-controlled loops, we employ loop inversion [17]. Basically, loop inversion transforms a head-controlled loop to an if-statement that surrounds a tail-controlled loop. Compilers perform this optimization in order to reduce the impact of branches at the expense of code duplication: a head-controlled loop features two branches, one conditional and one unconditional, while a tail-controlled loop features only one conditional branch.

In order to identify head-controlled loops, we inspect the entry, repetition, and exit nodes/edges of an SCC. We consider an SCC head-controlled, if it fulfills the following criteria:

- a single entry edge e_0^E, repetition edge e_0^R, and exit edge e_0^X
- the target of e_0^E, namely n_S^E, is the first node of a linear subgraph S
- the source of e_0^X, namely n_S^X, has two outgoing edges, e_0^X and e^B, and is the last node of subgraph S

The left image in Figure 4 illustrates the used notation. The linear subgraph S represents the condition of the loop, with edge e^B leading to the loop's body, and edge e_0^X exiting it.

We restructure such a loop as follows:

- copy linear subgraph S. We further denote to this copy as S_c with its first node n_{Sc}^E and last node n_{Sc}^X
- divert edge e_0^R to n_{Sc}^E
- insert a new repetition edge from n_{Sc}^X to the target of e^B
- insert a new exit edge from n_{Sc}^X to the target of e_0^X

Basically, the condition of the head-controlled loop is copied, and represents together with its body the new tail-controlled loop. The final result is shown in the right image of Figure 4. Note, even though we facilitate copying throughout this approach, it cannot lead to exponential code bloat [14].

V. Experimental Evaluation

The transformation of unstructured control flow can eliminate redundant execution caused by branch divergence and therefore improve performance. This section describes the results of applying control flow restructuring to synthetic unstructured CFGs and kernels from the Rodinia benchmark suite [15]. The synthetic unstructured CFGs are used to demonstrate that our approach effectively eliminates redundant execution for unstructured graphs with branch divergence. We evaluate the dynamic overhead of branch restructuring and

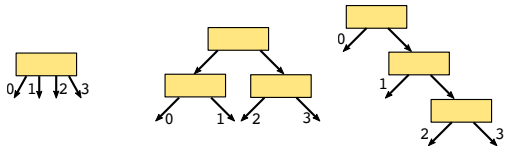

Fig. 5. **Left**: Basic block with 4 outgoing edges. **Center**: 4-way branch resolved in a breadth-first manner. **Right**: 4-way branch resolved in a depth-first manner.

its potential impact on execution time. The benchmark suite consists mostly of SESE graphs, and we use it to evaluate the overhead of our transformations on these graphs in terms of execution time and representational overhead at compile-time.

A. Compiler Implementation

We evaluated control flow restructuring by implementing it as an additional pass to NVIDIA's CUDA compiler. The pass takes PTX as input, restructures all CFGs, and produces PTX for further processing as output. We extracted the grammar for parsing PTX from the Ocelot compiler framework [18] and create an AST with it. This AST is converted to a CFG, restructured with our algorithms from Section IV, and converted back to PTX.

A necessary constraint of control flow restructuring on a CFG is the support of n-way branches. These need to be resolved to cascades of 2-way branches with the help of additional auxiliary variables in order to make a conversion to PTX possible. Different cascades, such as breadth-first or depth-first as shown in Figure 5, or a mix of both, are possible. For our experiments, we resolve n-way branches with depth-first cascades of 2-way branches.

B. Experimental Platform and Setup

The evaluation is performed on a system with an Intel Core i7-3770K CPU @3.5 GHz, an NVIDIA Tesla K20, and NVIDIA's driver version 346.46. We use the CUDA 7.0 toolkit, running on Ubuntu 12.04. We perform our experiments on a NVIDIA platform, since it allows us to experiment with structured and unstructured control flow. An AMD platform would have given us only the possibility to execute structured control flow, and would have made it impossible to quantify the difference between structured and unstructured control flow.

All programs were compiled with `-Xcicc=-O0` and `-Xptxas=-O0` to ensure no interference from other compilation stages. Ideally, control flow restructuring should be carried out as late in the compilation pipeline as possible in order to avoid side effects from other compilation stages.

Each benchmark in Section V-D is run 10 times, and we report the average kernel execution time of all runs. We measured execution times using the CUDA profiler. In case benchmarks consists of multiple kernels, we add the execution time of all kernels in each run before computing the average. Benchmark results were verified to equal their results when restructuring is disabled.

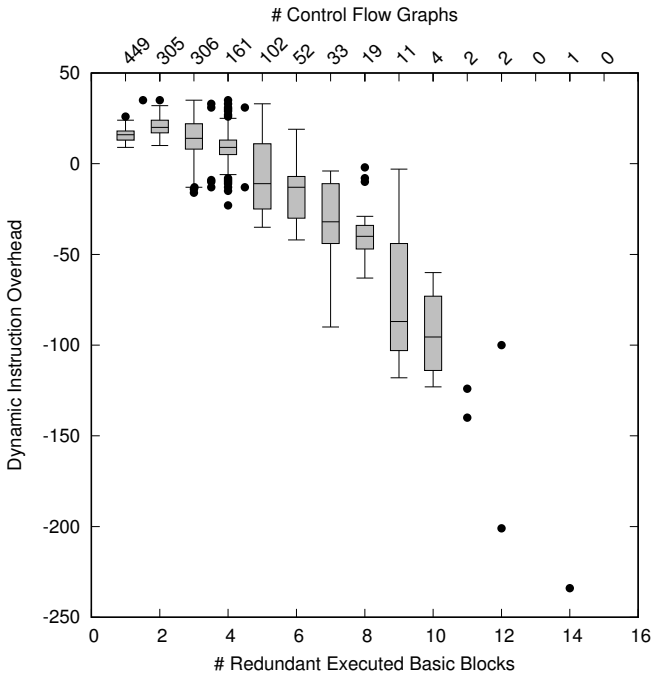

Fig. 6. Dynamic Instruction Overhead for all unstructured CFGs up to size 7.

C. Synthetic Control Flow Graphs

This section demonstrates that our approach effectively eliminates redundant basic block executions for unstructured graphs with branch divergence. We evaluate the dynamic overhead of branch restructuring and its potential impact on execution time.

1) Experimental Setup: We evaluate the dynamic overhead of control flow restructuring by generating the incidence matrices for all acyclic CFGs with binary branches for a given dimension. We filter out all minimal unstructured CFGs and convert these matrices to CUDA code. All branches were made divergent to ensure redundant execution of basic blocks. The other basic blocks contained no computation, in order to ensure accurate dynamic overhead measurements.

We compile the CFGs with and without branch restructuring, and count the redundant executions of basic blocks for the unstructured case as well as the number of executed instructions for both cases using the CUDA profiler. We compute the *dynamic instruction overhead* for each graph by subtracting the number of executed instructions of the restructured CFG from the corresponding unstructured CFG.

2) Key Observations: We produce all synthetic CFGs up to 7 nodes, resulting in 1447 CFGs after filtering. We restrict our experiments to synthetic CFGs of this size, because it produces a sufficient number of unstructured graphs to demonstrate the effect of branch restructuring. Figure 6 shows the dynamic instruction overhead for these graphs. We group the CFGs by their number of redundantly executed basic blocks and count the number of CFGs for each group. For example, as shown in Figure 6, we count 449 CFGs which execute one basic block

redundantly, and only 2 CFGs which execute 12 basic blocks redundantly. We plot a box and whisker plot for each group. The bottom and top of the boxes represent the first and third quartile, and the line inside the box the median. The ends of the whiskers indicate 1.5 times the interquartile range, and all points not within that range are outliers plotted as small dots.

Unstructured control flow in combination with branch divergence leads to redundant execution of basic blocks. Figure 6 shows that 73% of the synthetic CFGs have up to 3 redundant executions, and that the maximum number of redundantly executed basic blocks is 14. The maximum dynamic instruction overhead is 35, indicating that the added dynamic overhead of branch restructuring in the presence of branch divergence is small. Thus, in our experiments branch restructuring is desirable as long as the combined instruction count of the redundant executions exceeds 35. Moreover, Figure 6 clearly shows that the dynamic overhead of branch restructuring becomes smaller, the more redundant executions a graph contains. The dynamic instruction overhead is always negative for graphs with more than 7 redundant executions, indicating that fewer instructions are executed in the restructured than the corresponding unstructured graph. For these graphs, the number of redundantly executed instructions in the unstructured graphs *always* exceeds the overhead inserted by branch restructuring. Thus, branch restructuring is always desirable for these graphs even without any computation contained in the basic blocks.

D. Benchmarks

This section describes the results of applying control flow restructuring to kernels from the Rodinia benchmark suite [15]. We evaluate the overhead of our transformations on these kernels in terms of execution time and representational overhead at compile-time.

1) Structural Analysis: In order to obtain an overview of the structural complexity of the benchmarks, we classified CFGs as *Linear* per Definition 3, *Tail-structured* per Definition 2, *single-entry/single-exit (SESE)* per Definition 1, *reducible* or *irreducible*. Tail-structured and SESE were identified by structural analysis [19], and irreducibility was determined by T1/T2 analysis [20]. A graph's cyclicity was identified by determining the presence of SCCs [21].

Table I shows the distribution of each class. The majority of the CFGs are single-entry/single-exit, and most acyclic SESE graphs are also tail-structured. Thus, the majority of programs in the Rodinia benchmark suite are expressed using simple if-then-else statements and head-controlled loops. Control flow restructuring introduces no overhead for the acyclic graphs, but transforms head-controlled loops to tail-controlled ones. We expect therefore an overhead associated with loop restructuring. Only a minority of the CFGs are in the reducible class. We inspected the source code for these graphs and found that the acyclic one is due to a switch statement with return statements in its cases. It is part of mummergpu. The cyclic graphs are due to loops with multiple exits and are part of hotspot, hybridsort, mummergpu, myocyte, particlefilter, and pathfinder.

	acyclic	cyclic
Linear	28	-
Tail-structured	56	3
SESE	3	139
Reducible	1	10
Irreducible	-	-
Total	240	

TABLE I
PROGRAM CLASSIFICATION FOR THE RODINIA BENCHMARK SUITE. CLASSES ARE RELATED AS FOLLOWS: LINEAR $\subset$ TAIL-STRUCTURED $\subset$ SESE $\subset$ REDUCIBLE

Overall, the Rodinia benchmark suite consists mostly of SESE graphs, which can always be executed efficiently on GPUs. It offers little opportunity for improvements through control flow restructuring, considering that the presence of branch divergence is also required. This is rather unsurprising, since developers are aware of the potential disadvantages of unstructured control flow for GPUs and therefore try to avoid it. A compiler supporting control flow restructuring would be able to remove unstructured control flow altogether. This would allow programmers to delegate this task to the compiler and spend their time on tuning other aspects of a program.

2) Execution Times: Figure 7 shows the measured execution times for the Rodinia benchmark suite. We use six different restructurer configurations:

- *nvcc*: The benchmarks were compiled with the unmodified nvcc compilation pipeline.
- *no restructuring*: The PTX files are parsed, converted to CFGs, and immediately reconverted. No restructuring is performed.
- *loop restructuring*: Loop restructuring as described in Section IV-A.
- *loop copy restructuring*: Loop restructuring with copying as described in Section IV-C.
- *loop + branch restructuring*: Loop and branch restructuring as described in Section IV-B.
- *loop copy + branch restructuring*: Loop restructuring with copying and branch restructuring.

The *no restructuring* configuration serves as baseline, and all other configurations are normalized to it. The reason for using the *no restructuring* and not the *nvcc* configuration as baseline is due to the conversion passes. The CFG to AST conversion lays out basic blocks differently than they are in the input PTX file. This results in a different basic block order and therefore a different number of fall-through branches in the output PTX. The effect alters execution time by no more than 8%, except in the case of mummergpu, where we observe a 5 fold increase. We found that the difference is due to an additional basic block in the layout of nvcc. The basic block contains no instructions and has one incoming and outgoing edge and could therefore be safely removed without effecting the computation. However, ptxas produces pbk and brk in-

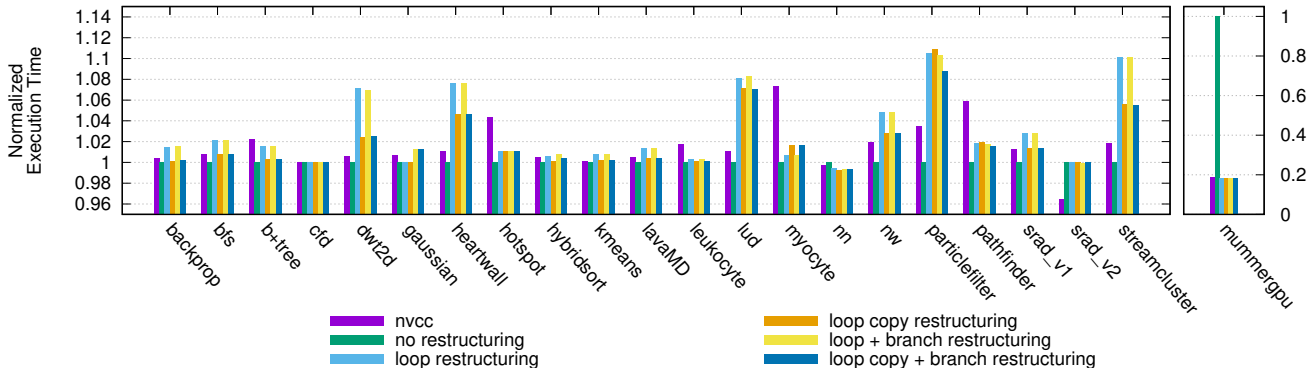

Fig. 7. Execution Times for different Control Flow Restructuring Configurations

structions for the inner kernel loop when it is present. These instructions allow an early reconvergence of divergent threads in the loop, making it possible to avoid redundant executions for loops with multiple exits. Although these instructions allow to reduce execution time when divergence is present, nvcc seems not be able to reliably generate them.

Loop restructuring transforms all loops into tail-controlled loops by inserting additional branches and assignments. Only loops that are already tail-controlled are not altered. However, the majority of the loops in the Rodinia benchmark suite are head-controlled. These loops are converted to tail-controlled loops by converting one unconditional to a conditional branch, and inserting two assignments, with one of them being executed every loop iteration. This results in a noticeable execution time overhead for most benchmarks. The overhead is particularly pronounced with over 5% for dwt2d, heartwall, lud, nw, particlefilter, and streamcluster.

The benchmarks dwt2d, lud, nw, particlefilter, and streamcluster consist of very small kernels with an average execution time of less than 1.5ms per invocation. Loop restructuring adds additional instructions to the kernels of these benchmarks, and therefore creates an overhead that is a noticeable fraction of the execution time. For example, the average execution time of streamcluster's kernel is only 750μs, but it is invoked 1611 times. In case of heartwall, the average kernel execution time is with 195ms significantly longer, but it consists of 48 head-controlled loops which are responsible for the overhead in execution time.

For most benchmarks, overhead is reduced using loop restructuring with copying. It transforms head-controlled to tail-controlled loops by employing loop inversion [17] instead of inserting assignments and branches. Thus, no additional assignment is executed every loop iteration. Another positive effect on execution time can be observed for mummergpu. Loop restructuring improves its performance 5 fold, rendering it equivalent to the code produced by nvcc. It allows divergent threads to reconvergence early and therefore reduces redundant execution as the pbk and brk instructions.

Branch restructuring is employed after all loops have been restructured and ensures proper nesting of splits and joins. Figure 7 shows that it has no significant effect on the execution times, and performs similarly to the corresponding loop restructuring. Most acyclic graphs are already tail-structured and we suspect a proper nesting of splits and joins in the cyclic ones as well. It is therefore no surprise that the execution times show no significant change compared to the corresponding loop restructuring configurations.

Overall, the experiments with the Rodinia benchmark suite indicate that control flow restructuring adds minor and varying overhead to the execution times of programs. It varies between not measurable and 12%, with an average of 2.1% among all benchmarks. The reason for this is that the Rodinia benchmark suite consists mainly of SESE graphs, and control flow restructuring only inflicts no overhead to the subset of tail-structured graphs. While this overhead is not insignificant, it is much lower than the 100 - 150% reported by Domínguez *et al.* [11], [12]. On the other hand, when unstructured control flow and branch divergence is present, control flow restructuring can help to reduce execution time significantly as demonstrated for mummergpu. This suggests that it should be applied more selectively, *e.g.* in combination with structural analyses [19] to discover unstructured subgraphs, and divergence analysis [16] for detecting divergent branches. In contrast to other restructuring methods [11], [12], [6], [13], it also does not lead to exponential code inflation [14].

3) Compile-Time Overhead: Control flow restructuring can add constructs to a CFG, causing representational overhead at compile-time. This is quantified in Figure 8, which relates the number of instructions before restructuring to the number of instructions after restructuring for the *loop copy + branch restructuring* configuration. The grey line marks the identity function, representing points with no overhead.

There is a clear linear relationship for all cases, suggesting that control flow restructuring is practically feasible in terms of space requirements. All linear and tail-structured graphs lie exactly on the line, confirming that no representational overhead is introduced. SESE and reducible graphs lie slightly above the line, indicating the insertion of additional instructions. The average representational overhead for these graphs in terms of instructions is 5.2%. Figure 8 is representative for

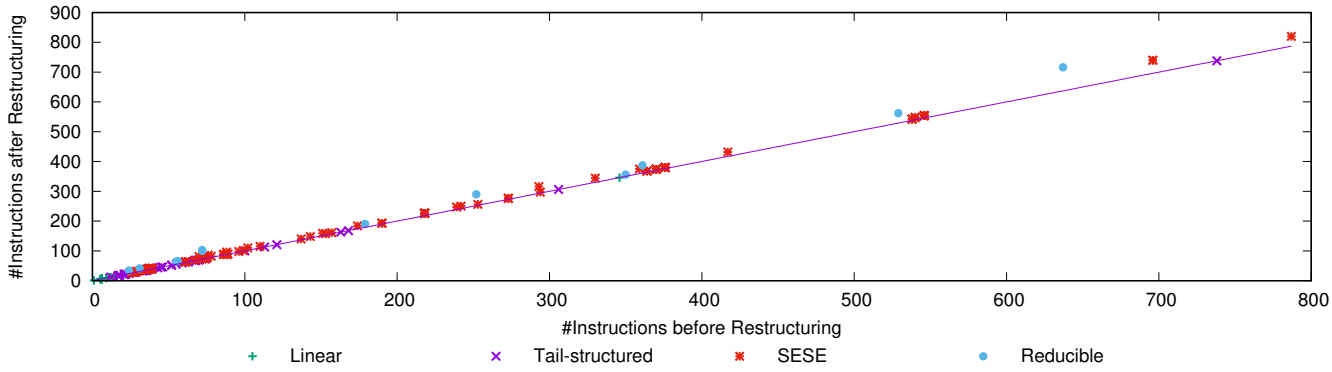

Fig. 8. Representational Overhead for Loop Copy + Branch Restructuring Configuration

all the other configurations, which exhibited similar behavior for their representational overhead.

VI. Related Work

Reducing the performance impact of thread divergence is a topic of extensive and ongoing research. Several works proposed changes to GPU hardware to ameliorate the problem. ElTantawy *et al.* [22] replaced the traditional stack based thread reconvergence mechanism with a set of tables, potentially allowing warps to reconverge before the branch's IPDOM. They evaluated their approach on a set of benchmarks with unstructured control flow and achieved a harmonic mean speedup of 32% compared to traditional execution. Branch herding was proposed by Sartori *et al.* [23]. It forces all threads of a warp to take the path of the majority. This led to incorrect results, but was acceptable for error tolerant applications such as visual computing applications. Their hardware implementation improved performance for a set of benchmarks by 30% on average. Brunie *et al.* [24] proposed to add additional hardware to co-issue different instructions to disjoint sets of the same warp, or to a subset of a different warp. Diamos *et al.* [4] proposed thread frontiers, a combined hardware and software approach. In this approach, the compiler finds potential early reconvergence points, while additional hardware checks whether a warp can reconverge at these points.

Two software based approaches were proposed by Han *et al.* [5]. They reduce branch divergence through iteration delaying and branch distribution. Iteration delaying reorders loop iterations with branches so that branches taking the same direction are executed together. Branch distribution factors out similar code from branches. Both techniques require manual code rewriting. Zhang *et al.* [25] removed divergence through data reordering and job swapping, *i.e.* changing the mapping between threads, data, and work. This must be done asynchronously by the CPU at runtime, and therefore requires to launch a kernel multiple times in a loop. Lee *et al.* [26] proposed algorithms that remove all control flow by predicating and linearizing different execution paths. They implemented their algorithms in the CUDA LLVM compiler and showed that a predication-only architecture based on their algorithms is competitive in performance to one with hardware support for tracking divergence.

Finally, like our method, several approaches transform unstructured to structured control flow to reduce the impact of branch divergence. Anantpur *et al.* [7] proposed a technique for transforming unstructured to structured CFGs by linearizing them with the help of guard variables. They implemented it as PTX transformations and evaluated it on a set of benchmarks. It increased code size by up to 10% and execution time by up to 73%. Wu *et al.* [6], [13] use adaptions of the transformations of Zhang *et al.* [27]. They show that several Rodinia, Parboil, and Optix benchmarks, as well as CUDA SDK samples contain unstructured control flow. Applying their transformations increased static instruction count, and decreased performance by up to 1% due to code expansion. Dominguez *et al.* [12], [11] developed a tool for translating PTX to AMD IL in order to understand the performance differences between structured and unstructured control flow on GPUs. They also used the transformations of Zhang *et al.* [27] to handle unstructured control flow. Their tool produced code that performed 2.1 times worse on average than a straightforward manual CUDA to OpenCL translation.

VII. Conclusion and Future Work

In this paper, we presented a transformation for converting unstructured to structured control flow. Our evaluation shows that our approach effectively eliminates redundant basic block execution and improves execution time for unstructured graphs with branch divergence. It adds a minor average overhead of 2.1% to execution time of already structured kernels. While this overhead is notable, it is significantly lower than the 100-150% reported by Domínguez *et al.* [11], [12]. This suggests that our transformations should be applied more selectively, *e.g.* in combination with structural analysis [19] to discover unstructured subgraphs, and divergence analysis [16] for detecting divergent branches. The representational overhead at compile-time is linear in terms of instructions. In contrast to other restructuring methods [11], [12], [6], [13], exponential code inflation is impossible [14].

We also showed that the main increase in execution time in structured kernels is due to restructuring of head-controlled loops. Our main direction for future work is therefore to extend our algorithm to SESE graphs in order to avoid the added overhead and therefore the need for structural analysis. Another direction for future work would be to combine loop restructuring with loop merging [28]. This optimization merges a divergent loop with one or more of its surrounding loops in order to overlap the iteration spaces of the inner loop for threads of different warps.

REFERENCES

[1] "CUDA C Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide, accessed: 2015-11-02.

[2] "OpenCL - The open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl, accessed: 2015-11-02.

[3] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420.

[4] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "Simd re-convergence at thread frontiers," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 477–488.

[5] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 3:1–3:8.

[6] H. Wu, G. Diamos, S. Li, and S. Yalamanchili, "Characterization and transformation of unstructured control flow in GPU applications," in *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.

[7] J. Anantpur and G. R., "Taming control divergence in GPUs through control flow linearization," in *Compiler Construction*, ser. Lecture Notes in Computer Science, A. Cohen, Ed. Springer Berlin Heidelberg, 2014, vol. 8409, pp. 133–153.

[8] Advanced Micro Devices, "ATI Intermediate Language (IL) Specification v2.4," 20011.

[9] "Parallel Thread Execution ISA Version 4.3," http://docs.nvidia.com/cuda/parallel-thread-execution, accessed: 2015-11-02.

[10] H. Bahmann, N. Reissmann, M. Jahre, and J. C. Meyer, "Perfect reconstructability of control flow from demand dependence graphs," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 66:1–66:25, Jan. 2015.

[11] R. Domínguez, D. Schaa, and D. Kaeli, "Caracal: Dynamic translation of runtime environments for GPUs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 5:1–5:7.

[12] R. Dominguez and D. Kaeli, "Unstructured control flow in gpgpu," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1194–1202.

[13] H. Wu, G. Diamos, J. Wang, S. Li, and S. Yalamanchili, "Characterization and transformation of unstructured control flow in bulk synchronous GPU applications," *International Journal of High Performance Computing Applications*, 2012.

[14] L. Carter, J. Ferrante, and C. D. Thomborson, "Folklore confirmed: reducible flow graphs are exponentially larger." in *POPL*. ACM, 2003, pp. 106–114, aCM SIGPLAN Notices 38(1), January 2003.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[16] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr., "Divergence analysis and optimizations," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 320–329.

[17] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[18] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 353–364.

[19] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers," *Comput. Lang.*, vol. 5, no. 3-4, pp. 141–153, Jan. 1980.

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

[21] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[22] A. ElTantawy, J. Ma, M. O'Connor, and T. Aamodt, "A scalable multi-path microarchitecture for efficient GPU control flow," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 248–259.

[23] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, pp. 279–290, Feb 2013.

[24] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 49–60, Jun. 2012.

[25] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," *SIGPLAN Not.*, vol. 46, no. 3, pp. 369–380, Mar. 2011.

[26] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović, "Exploring the design space of spmd divergence management on data-parallel architectures," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 101–113.

[27] F. Zhang and E. D'Hollander, "Using hammock graphs to structure programs," *Software Engineering, IEEE Transactions on*, vol. 30, no. 4, pp. 231–245, April 2004.

[28] T. D. Han and T. S. Abdelrahman, "Reducing divergence in gpgpu programs with loop merging," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 12–23.