Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection

Farzad Khorasani Rajiv Gupta Laxmi N. Bhuyan
Computer Science and Engineering Department
University of California Riverside, CA, USA
{fkhor001, gupta, bhuyan}@cs.ucr.edu

ABSTRACT

GPU's SIMD architecture is a double-edged sword confronting parallel tasks with control flow divergence. On the one hand, it provides a high performance yet power-efficient platform to accelerate applications via massive parallelism; however, on the other hand, irregularities induce inefficiencies due to the warp's lockstep traversal of all diverging execution paths. In this work, we present a software (compiler) technique named *Collaborative Context Collection* (*CCC*) that increases the warp execution efficiency when faced with thread divergence incurred either by different intra-warp task assignment or by intra-warp load imbalance. CCC collects the relevant registers of divergent threads in a warp-specific stack allocated in the fast shared memory, and restores them only when the perfect utilization of warp lanes becomes feasible. We propose code transformations to enable applicability of CCC to variety of program segments with thread divergence. We also introduce optimizations to reduce the cost of CCC and to avoid device occupancy limitation or memory divergence. We have developed a framework that automates application of CCC to CUDA generated intermediate PTX code. We evaluated CCC on real-world applications and multiple scenarios using synthetic programs. CCC improves the warp execution efficiency of real-world benchmarks by up to 56% and achieves an average speedup of 1.69x (maximum 3.08x).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – Compilers, Optimization, Code generation

Keywords

GPU, GPGPU, warp, SIMD, SIMT, warp execution, divergence, CCC, context stack

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MICRO-48, December 05-09, 2015, Waikiki, HI, USA
© 2015 ACM. ISBN 978-1-4503-4034-2/15/12$15.00.
DOI: http://dx.doi.org/10.1145/2830772.2830796

1. INTRODUCTION

Graphics Processing Units (GPUs) have become the essential part of high-performance and power-efficient parallel computing. Recent emergence of programming APIs such as CUDA and OpenCL have played an important role in enabling general purpose parallel computing using GPUs. A key to the success of such APIs is the freedom they provide to define different execution paths for the threads inside a SIMD group (warp or wavefront); while the underlying architecture manages the resulting complexities. To emphasize this aspect of GPUs, they are also called SIMT (Single Instruction Multiple Thread) microarchitecture devices. However, the combination of this attribute with GPUs inherent lack of fine-grained task parallelism support may result in a significant performance loss.

All the threads inside a SIMD group (i.e., all the warp lanes) execute one unique instruction at a time. The presence of conditionals—such as due to if-else code blocks—causes *thread divergence* because a conditional may evaluate to true for some warp lanes and false for other lanes. In this situation, the warp takes all the divergent paths, while disabling non-relevant lanes inside every path. That is, the warp scheduler fetches instructions for all the divergent paths while the execution stage is active only for a number of corresponding threads. As a result, a portion of the available processing power goes unutilized for the duration of divergence, diminishing the SIMD execution benefits.

Microarchitectural and compiler solutions have been suggested to remedy thread divergence. While most microarchitectural methods are based on warp formation [1, 2, 3] or warp compaction [4, 5, 6], the compiler solutions rely on the warp lanes majority vote to gain partial warp execution enhancement [7, 8, 9]; however, full warp utilization is usually out-of-reach. The compiler based solutions require program-specific and input-specific information about divergence behavior to *speculate* on scheduling divergent code blocks and lack systematic application procedure. Other software approaches disrupt the GPU kernel autonomy by offloading data/task reordering onto CPU [10], implement global locks with heavy contention for global queues [11], or accept errors in the output by ignoring the task of minorities [12]. The above limitations of software methods make them often inapplicable and unreliable.

In this paper, we propose Collaborative Context Collection (CCC), a software (compiler) technique for GPUs that enables efficient execution of warps in presence of dissimilar intra-warp task assignment or intra-warp load imbalance. Unlike previous solutions, CCC does not rely on heuristics to increase warp utilization; instead, it accumulates tasks to provide maximum warp execution efficiency. CCC utilizes the fast shared memory of the Streaming Multiprocessor (SM) to collect threads' *context* which includes the content of the thread-private registers that are sufficient to describe the thread's task inside the divergent path. *Context collection* provides all the warp lanes with homogeneous tasks and hence allows efficient warp execution. CCC is primarily aimed at removing divergence from repetitive GPU code blocks (e.g., loops). Threads in a warp are provided with a warp-specific stack in shared memory. In each iteration, if there are insufficient contexts of the divergent path in the stack to keep unemployed threads busy, warps collect their context on the stack. Otherwise, each unemployed thread grabs a context from the shared memory and all the warp lanes execute the divergent branch. This eliminates warp underutilization since the warp lanes follow the *all-or-none* principle for taking the divergent path. By collecting tasks at the warp granularity, CCC avoids need for any syncing or fencing operations. CCC exploits fast CUDA intrinsics to implement intra-warp binary reduction and prefix sum necessary for collaborative context storing and restoring.

We further present transformations to extend the applicability of CCC to many common code patterns with intra-warp load imbalance or dissimilar task assignment, such as loops with varying or unknown trip-count and recursive functions. We also provide optimizations for CCC to increase performance in certain situations and to eliminate or reduce the CCC possible side-effects such as memory divergence or theoretical occupancy limitation. Finally, we implemented CCC in a software framework that allows annotating the repetitive patterns and the divergent paths in CUDA C/C++ kernels. Annotations are then replaced and transferred to the Nvidia CUDA Compiler (NVCC) generated intermediate PTX where CCC is applied to PTX code via our source-to-source PTX compiler. Then, the framework feeds the transformed code to the rest of the compilation chain.

This paper makes the following contributions:

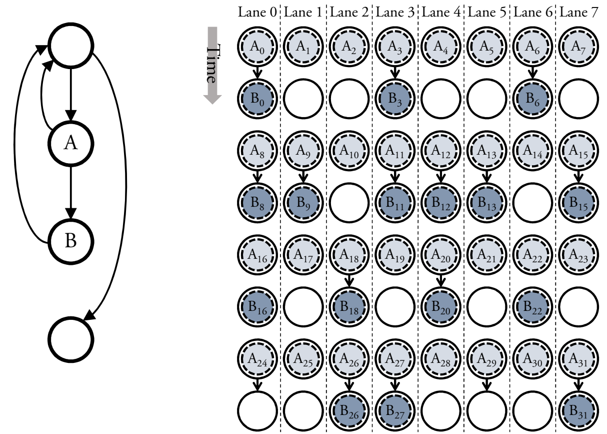
– We propose *CCC*, a compiler technique for CUDA programs that boosts the warp execution efficiency upon divergence. CCC collects tasks at warp granularity and remedies inefficiency due to intra-warp load imbalance or dissimilar task assignment.

– To enhance the applicability of CCC, we present transformations to make common code patterns accessible to CCC and develop optimizations to increase the CCC performance.

– We evaluate CCC for a variety of real-world and synthetic GPU kernels. CCC achieves up to 56% warp execution efficiency enhancement and an average speedup of 1.69x (maximum 3.08x).

```
1  __global__ void CUDA_kernel_BFS(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        bool p = levels[ vIdx ] == curr; // Block A.
9        if( p )
10          process_nbrs( vIdx,
11          curr, levels, v, e, done ); // Block B.
12    } }
```

(a) The CUDA kernel for iterative BFS graph processing.



(b) Kernel CFG. (c) Visualization of warp execution.

Figure 1: BFS graph processing in CUDA [13] - Each warp lane processes a set of vertices. Some lanes may need to execute code block B while code block A is non-divergent. Warp size is 8.

2. Collaborative Context Collection

In this section, we first explain *thread divergence* problem, then discuss our solution.

2.1 Thread Divergence Problem Overview

The SIMT microarchitecture in GPUs provides a parallel processing platform that groups fine-grained threads into warps. A warp owns only one active Program Counter (PC) at a given time, allowing the hardware to schedule one instruction for execution by multiple execution units (SM cores, in terms of CUDA) using only one instruction fetch and decode. This design reduces the die size and power consumption while providing massive parallelism. However, warp lanes must run in lockstep. Therefore, specifying different execution paths for threads of a warp—also called warp lanes—results in the traversal of all the divergent branches by them. For a divergent branch, the processor masks off inactive threads while holding their reconvergence PC in a hardware divergence stack [14]. Until the warp's active PC reaches the reconvergence PC, masked off threads stay inactive; thus some reserved execution units are not utilized, causing warp execution inefficiency.

Figure 1 illustrates the above using BFS adapted from [13]. Each thread is assigned processing of a number of vertices. If a vertex is updated in the previous

CUDA kernel invocation, the thread must update vertex's neighbors. This condition leads to thread divergence as it can evaluate to true for some warp lanes and to false for others. Threads that do not execute divergent branch must wait for other threads in the warp to finish processing `block B` as illustrated in Figure 1(c).

2.2 Boosting Warp Efficiency with CCC

To eliminate thread divergence due to imbalanced load/task assignment to warp lanes, we propose Collaborative Context Collection (CCC). CCC increases the warp execution efficiency for kernels containing repetitive diverging tasks with independent iterations. This pattern is common in GPU thread task assignment. The BFS graph processing CUDA kernel shown in Figure 1(a) matches this model and Figure 1(b) depicts the corresponding control flow graph (CFG). Later in Section 3 we introduce transformations enabling a wide variety of GPU algorithms to be expressed in this form.

In the above program model, threads inside the repetitive code block (the loop) iterate over divergent tasks. The key idea of CCC is keep collecting tasks corresponding to threads of a warp until there are tasks to keep all threads busy. To establish a connection between the divergent task and its required data, we define a **context** as the minimum set of variables that can fully describe the functionality of the task if the task is carried out by another thread. For a GPU thread, these variables are a subset of thread's registers (that are thread-private).

In CCC, at every iteration, if there are insufficient divergent tasks to keep all the warp lanes busy, then lanes that are assigned such tasks collect their context in the **context stack**. Context stack is a warp-specific shared memory region for collecting unprocessed task contexts. After stacking the contexts, the warp moves to the next iteration without entering the divergent branch. Later, if the aggregation of tasks that are stacked and the tasks assigned to the warp lanes in the current iteration exceeds the warp size, lanes without a task can grab a context from the shared memory, and the entire warp executes the divergent branch. Thus, warp lanes execution discipline upon divergence is **all-or-none** which avoids warp underutilization. Figure 2 demonstrates the impact of CCC on the divergent CUDA program in Figure 1 by visualizing the execution of a warp.

2.3 Applying CCC to CUDA kernels

Let us precisely define *context*. We refer to a thread's task context as a set of its designated registers that are:

1. *defined* prior to divergent task path as a function of thread-specific special registers including `%tid`, `%laneid`, and lane masks such as `%lanemask_eq`; and
2. *used* inside the divergent task path.

Note that a context may contain special registers themselves when they are used directly inside the divergent path. This definition enables CCC to distinguish the minimal subset of thread's registers that need to be collected and retrieved at every iteration.

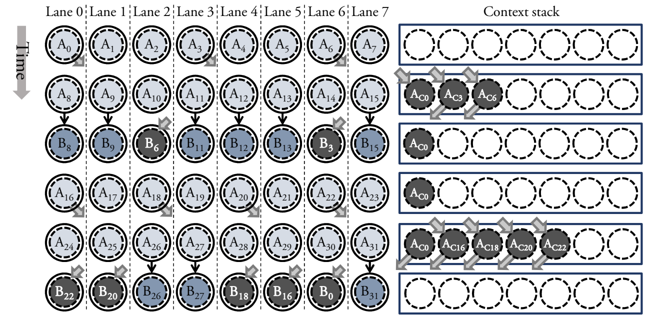


Figure 2: Applying Collaborative Context Collection to the program in Figure 1 eliminates warp execution inefficiency. The figure visualizes the execution of only one warp.

```
1  __global__ void CUDA_kernel_BFS_CCC(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4      volatile __shared__ int cxtStack[ CTA_WIDTH ];
5      int stackTop = 0;
6      int wOffset = threadIdx.x & ( ~31 );
7      int lanemask_le = getLaneMaskLE_PTXWrapper();
8      for(
9      int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
10     vIdx < numV;
11     vIdx += gridDim.x * blockDim.x ) {
12        bool p = levels[ vIdx ] == curr; // Block A.
13        int jIdx = vIdx;
14        int pthBlt = __ballot( !p );
15        int reducedNTaken = __popc( pthBlt );
16        if( stackTop >= redNTaken ) { // All take path.
17           int wScan = __popc( pthBlt & lanemask_le );
18           int pos = wOffset + stackTop - wScan;
19           if( !p ) jIdx = cxtStack[ pos ]; // Pop.
20           stackTop -= reducedNTaken;
21           process_nbrs( jIdx,
22           curr, levels, v, e, done ); // Block B.
23        } else { // None take path.
24           int wScan = __popc( ~pthBlt & lanemask_le );
25           int pos = wOffset + stackTop + wScan - 1;
26           if( p ) cxtStack[ pos ] = jIdx; // Push.
27           stackTop += warpSize - reducedNTaken; } } }
```

Figure 3: Applying CCC on the BFS CUDA kernel in Figure 1(a).

Next we describe CCC's application to the CUDA BFS kernel. Figure 3 shows CUDA BFS kernel in Figure 1(a) after applying CCC. The first highlighted section (lines 4-7 in Figure 3) is initialization of variables and the stack for CCC. The context stack consists of `vIdx` in Original CUDA code (Figure 1(a)) since it is the only variable that depends on the thread index and is used in the divergent region. CCC uses shared memory —the fastest memory after thread-private registers— for collecting contexts. The stack is marked `volatile` to pass data between warp lanes without any need for adding synchronization or fencing primitives. `volatile` qualifier inhibits unsolicited optimization of references and enforces sequential consistency between the threads of a warp accessing the shared memory. Also, all the threads initialize the context stack top to zero indicating no context has been collected yet.

The second highlighted region in Figure 3 (lines 14-15) is executed in every iteration. Threads count the total number of lanes for which the predicate for taking divergent path is false. This is the total number of lanes inside the warp that will be idle during the divergent

branch in the original program. If the result is less than or equal to the number of stacked contexts, it means all unemployed lanes can restore context, i.e., the warp can take divergent path without underutilization (starting from line 17). Otherwise, full warp utilization is not possible by taking the divergent path; hence, threads with task collect their context into the stack (starting from line 24) and the warp moves to the next iteration without executing the divergent branch. In the third highlighted section (lines 17-20 in Figure 3) unemployed threads calculate the stack index from which they pop contexts. After popping, all the threads move down the stack. Note that it is necessary to check the predicate before popping the stack (line 19) since only those warp threads that do not have any task to perform should grab an existing context. Those that have a task simply execute the task context they already hold. The fourth highlighted section (lines 24-27) is the counterpart of the third section for pushing contexts. Finally, there is another section (not shown) for executing leftover task contexts in the stack after finishing all the iterations (such section is present in every program discussed).

For simplicity, the stack size (number of elements) for each warp is made the same as the warp size, although the number of stacked contexts will never exceed warp size minus one because if it does, it means that in a previous iteration all the warp threads could have been utilized but were not. Note that CCC requires iterations of the repetitive pattern to be independent of each other so the reordering of iterations preserves program semantics. Therefore, barriers and memory fences, as long as they do not disrupt this feature, can be used in the iterative code segment. Finally, if a register is written inside the divergent path and the write operation can be expressed in form of a associative reduction function that has an atomic operation counterpart, to apply CCC, the register needs to be transferred to a shared memory buffer. Accesses to the register will be replaced with accesses to the corresponding shared memory buffer; especially the writes inside the divergent path should be made atomically.

The key methods that make CCC feasible yet fast are

- counting the total number of warp lanes with the false predicate–this is a form of intra-warp binary reduction; and
- realizing the stack position to/from which a thread needs to store/restore the context–this is a stream compaction problem that we solve using inclusive intra-warp binary prefix sum (scan).

We employed Harris and Garland's methods [15] for both intra-warp binary reduction and scan. Both methods utilize `__popc()` and `__ballot()` CUDA intrinsics and translate into very few binary operations.

3. CCC TRANSFORMATIONS

A GPU kernel, in its original form, may not readily expose the pattern of repeated divergent code block. Therefore to widen the applicability of CCC, we use enabling transformations for many common forms of GPU kernel patterns.

3.1 Task Repetition with Grid-Stride Loops

```
1  __global__ void CUDA_kernel_BFS(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
5     if( vIdx < numV ) {
6        bool p = levels[ vIdx ] == curr;
7        if( p )
8           process_nbrs( vIdx,
9           curr, levels, v, e, done ); } }

10 int main() { // Host side program.
11    . . . .
12    int gridD = ceil( numV / blockD );
13    gpuKernel <<< gridD, blockD >>> // Kernel launch.
14    ( numV, kernelIter, lev, v, e, done );
15    . . . . }
```

(a) Before transformation.

```
1  __global__ void CUDA_kernel_BFS_with_gridstride_loop(
2  const int numV, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        bool p = levels[ vIdx ] == curr;
9        if( p )
10          process_nbrs( vIdx,
11          curr, levels, v, e, done ); } }

12 int main() { // Host side program.
13    . . . .
14    int gridD = nSMs * maxThreadsPerSM / blockD;
15    gpuKernel <<< gridD, blockD >>> // Kernel launch.
16    ( numV, kernelIter, lev, v, e, done );
17    . . . . }
```

(b) After transformation.

Figure 4: A grid-stride loop applied to the BFS CUDA kernel in to make it accessible by CCC – assumed 100% maximum theoretical occupancy.

It is a well known software technique to launch the GPU kernel with exactly enough threads so that all the Streaming Multiprocessors are occupied. In this technique, threads inside the GPU kernel iterate over assigned tasks using a loop. The BFS CUDA code in Figure 4(a) employs this technique to allow a thread to iterate over multiple vertices. Figure 4 shows an example of such transformation and required changes in the host and the device code. Enabling task repetition with *grid-stride loops* is similar to persistent threads technique in [16] where enough residing GPU threads are invoked inside the kernel. However, in a grid-stride loop, the assignment of tasks to threads is predetermined rather than being controlled by a shared queue. A grid-stride loop can transform a kernel with divergence into a form suitable for CCC.

3.2 Loops with Variable Trip-Count

A common load assignment pattern is to assign a coarse-grained task to each GPU thread via a loop. The loop trip-count determines the amount of fine-grained tasks, i.e. the load volume, assigned to a thread. This approach is specifically prevalent in GPU graph processing (pioneered in [13]) where threads are assigned to graph vertices and process all the vertex's neighbors using a loop. Figure 5(a) exhibits this assignment pat-

tern. Although this load assignment strategy provides code readability, it can introduce heavy load imbalance and warp execution inefficiency. In case of GPU graph processing, since different vertices can have very different number of neighbors, different threads have to iterate different number of times over the loop. In power law graphs the load imbalance can be very high.

```
1  __device__ void process_nbrs(
2  const int vIdx, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     int eIdx = v[ Idx ];
5     int nNbrs = v[ vIdx + 1 ] - eIdx;
6     for( int nbrIdx = 0; nbrIdx < nNbrs; ++nbrIdx )
7        process_nbr( curr, eIdx + nbrIdx,
8        levels, v, e, done); }
```

(a) Before transformation.

```
1  __device__ void process_nbrs(
2  const int vIdx, const int curr, int* levels,
3  const int* v, const int* e, bool* done ) {
4     int eIdx = v[ Idx ];
5     int nNbrs = v[ vIdx + 1 ] - eIdx;
6     int UniCount = intra_warp_reduce_max( nNbrs );
7     for( int nbrIdx = 0; nbrIdx < UniCount; ++nbrIdx )
8        if( nbrIdx < nNbrs )
9           process_nbr( curr, eIdx + nbrIdx,
10          levels, v, e, done); }
```

(b) After transformation.

Figure 5: An example demonstrating the transformation of a CUDA device function (BFS processing of a vertex's neighbors) with variable trip-count to a form accessible by CCC.

Loops with variable trip-count can be expressed in form of a loop with uniform trip-count containing a divergent path, and hence, benefit from CCC. Figure 5(b) depicts this transformation applied to the BFS CUDA device function of Figure 5(a). First, warp lanes reduce the largest trip-count using the butterfly shuffle instruction (as in [17]), and select the resulting value as the uniform trip-count. Then, the code block inside the loop is wrapped by a condition check that verifies if the iteration is less than the thread's original trip-count. Note that although two pieces of code in Figure 5 are functionally equivalent, the transformation is required to keep warp threads active during all the iterations. Without the transformation, the threads that exit early will reconverge after the loop, therefore they will not have a chance to participate in the context collected by other warp lanes in further iterations.

3.3 Recursive Device Functions & Loops with Unknown Trip-Count

CUDA allows recursion on device functions enabling intuitive ways to express algorithms like cuckoo hashing [18] in Figure 6(a). In cuckoo hashing a key-value pair is provided with multiple hash functions. On every insertion attempt, the pair is inserted into the bucket pointed to by one of the hash functions using an atomic exchange. The return value of the atomic operation yields bucket contents before insertion. If content holds another key-value pair, the returned pair has to be rehashed with another hash function. The recursion on insertion attempt stops when returned bucket is empty.

```
1  __device__ bool try_insert(
2  ulonlong* pos, ulonglong& kvp ) {
3     kvp = atomicExch( pos, kvp );
4     return ( uint )( kvp >> 32 ); }

5  __device__ void insert_KVPair( ulonglong kvp,
6  uint loc, const uint tSize, ulonglong* table ) {
7     uint retKey = tryInsert( table + loc, kvp );
8     bool p = retKey != EMPTY_KEY;
9     if( p ) {
10       loc = find_next_location( retKey, loc, tSize );
11       insert_KVPair( kvp, loc, tSize, table ); } }

12 __global__ void generate_hash_table(
13 const int nKVPairs, const uint tableSize,
14 const ulonglong* kvpairs, ulonglong* table) {
15    for(
16    int eIdx = threadIdx.x + blockDim.x * blockIdx.x;
17    eIdx < nKVPairs;
18    eIdx += blockDim.x * gridDim.x ) {
19       ulonglong kvp = kvpairs[ eIdx ];
20       uint key = ( uint )( kvp >> 32 );
21       uint loc = hash_func( key, tableSize, 0 );
22       insert_KVP( kvp, loc, tableSize, table ); } }
```

(a) Before transformation.

```
1  __device__ void insert_KVPair_CCC( ulonglong kvp,
2  uint loc, const uint tSize, ulonglong* table ) {
3     uint retKey = tryInsert( table + loc, kvp );
4     bool p = retKey != EMPTY_KEY;
5     int redNtaken = intra_warp_binary_reduce( !p );
6     if( stackTop >= redNtaken ) {
7        pop( p, retKey, loc, kvp );
8        loc = find_next_location( retKey, loc, tSize );
9        insert_KVPair( kvp, loc, tSize, table );
10    } else {
11       push( p, retKey, loc, kvp ); } }
```

(b) After transformation. Operations and variables related to context stack are shortened for brevity.

Figure 6: An example demonstrating the transformation of a recursive CUDA device function (*cuckoo hashing on GPU* [18]) by CCC.

While some threads may succeed in inserting their key-value pair in the very first try, other threads in the same warp might take the divergent path over and over again causing overall warp underutilization. CCC can be applied to recursive device functions to eliminate the load imbalance. Figure 6(b) shows the resulting code after applying CCC. Before taking the divergent path, which contains the call to the recursive function, warp lanes count the predicates (line 5), and take the divergent path if all of them can be fully utilized. Otherwise, threads that have to call the recursive function stack their contexts, and the warp exits the function to grab fresh key value pairs and repeat the procedure.

Similar to the previous transformation shown, warp lanes discipline for calling the recursive function is *all-or-none* and thus enabling maximum warp utilization. Note that this solution focuses on *single tail recursion* where the recursive function contains a single call to itself. Since single tail recursion can be expressed in form of a loop for which the trip-count is not determined before entering the loop, the transformation for recursive functions also naturally extends to loops with unknown trip-count. Having multiple references to itself (*multiple recursion*) and dynamic parallelism are task generation problems that are beyond the scope of this work.

3.4 Nested and Multi-Path Context Collection

CCC can be applied in a nested manner to a divergent path containing intra-warp divergence. A separate stack collects the context of the parent divergent path and another stack collects the child's. For example, in CUDA BFS kernel, while neighbors of a vertex are visited only when it is updated in the previous GPU kernel invocation (Figure 1(a) line 9), variable trip-count for the inner loop (Figure 5(a) line 6), due to irregularity of the graph, creates load imbalance inside the divergent path. Here, CCC collects the context for the outer and the inner divergent paths independently, and executes each path only when enough contexts of that path exist.

Now consider the example of Iterated Function Systems (IFS) inside CUDA kernels. Figure 7 presents a device function in Fractal Flames GPU program [19] in which each thread executes a random function variation. For such cases, multi-path CCC assign a separate context stack to each task. Threads collect the contexts for each divergent branch separately, and warp lanes execute a path only if enough contexts of a certain task, that can provide full warp efficiency, are available.

```
1  __device__ cuFloatComplex variation_gen(
2  const float x, const float y, const uint var ) {
4     switch( var ) {
5     case 0: // Linear variation.
6        return make_cuFloatComplex( x, y );
7     case 1: // Sinusoidal variation.
8        return make_cuFloatComplex( sinf(x), sinf(y) );
9     . . . .
10    case 7: // Power variation.
11       float theta = atanf( x / y );
12       float len = sqrtf( x * x + y * y );
13       float sinTh = sinf( theta );
14       float mul = powf( len, sinTh );
15       float r = mul * cosf( theta );
16       float i = mul * sinf( theta );
17       return make_cuFloatComplex( r, i ); } }
```

Figure 7: Variation generation CUDA device function in Fractal Flame [19] from Iterated Function System (IFS) class.

4. CCC OPTIMIZATIONS

Next we discuss optimizations to improve CCC performance in certain situations.

4.1 Context Compression: Reducing Context Storage and Saving/Restoring Overhead

If in a context, a register value can be computed from another register's value with a computationally inexpensive operation(s), i.e. one can be expressed as a trivial compute-only function of another, only one of them needs to be collected during storing. For the restoration, one register content is then derived from the other one using the function. This optimization reduces shared memory consumption and lowers storing/restoring overhead.

An example can be found in Figure 6 where the context includes three variables: `kpv`, `loc`, and `retKey`. Since `retKey` can be recomputed from `kvp` using only a shift operation (Figure 6(a) line 4), the context is compressed by saving one less variable. Excluded variable

```
1  __global__ void CUDA_kernel_SSSP(
2  const int numV, int* bitMask, int* costs, int* Ua,
3  const int* v, const int* e, const int* eValues ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        int container = bitMask[ vIdx >> 5 ];
9        bool p = ( container >> ( vIdx & 31 ) ) & 1;
10       if( p ) { \\ Divergent path.
11          int vCost = costs[ vIdx ];
12          visit_nbrs( vIdx, vCost, costs, Ua,
13          v, e, done, eValues ); } } }
```

(a) Without optimization.

```
1  __global__ void CUDA_kernel_SSSP(
2  const int numV, int* bitMask, int* costs, int* Ua,
3  const int* v, const int* e, const int* eValues ) {
4     for(
5     int vIdx = threadIdx.x + blockIdx.x * blockDim.x;
6     vIdx < numV;
7     vIdx += gridDim.x * blockDim.x ) {
8        int container = bitMask[ vIdx >> 5 ];
9        bool p = ( container >> ( vIdx & 31 ) ) & 1;
10       int vCost;
11       if( p ) vCost = costs[ vIdx ];
12       if( p ) { \\ Divergent path.
13          visit_nbrs( vIdx, vCost, costs, Ua,
14          v, e, done, eValues ); } } }
```

(b) With Optimization (before transformation).

Figure 8: SSSP graph processing CUDA kernel from [13] containing a coalesced global memory access to the `costs` buffer in the divergent path. We preserve the coalescence in CCC by excluding the memory access from the divergent path.

(`retKey`) is reconstructed from context variable (`kpv`) using this shift operation during the pop procedure.

4.2 Memory Divergence Avoidance

Applying CCC reorders the iterations of the loop. In the original program, a group of iterations with consecutive indices get assigned to threads with consecutive global indices; but employing CCC may disrupt this assignment. This becomes an issue when there are global/host memory reads and writes in the divergent path that are in a direct relationship with the iteration index. The original coalesced and cache-friendly memory accesses may lose the notion of locality due to context collection and retrieval. The introduction of memory divergence can hurt the performance.

Figure 8(a) gives an example of such scenario: Single-Source Shortest Path (SSSP) in a graph using CUDA [13]. In this example, threads are assigned to process vertices iteratively. A vertex is processed only if its corresponding bit in the `bitMask` buffer is set. In the divergent path, accesses to the `costs` array have locality and are coalesced. However, if we apply CCC to this kernel, as we did in Figure 3, due to reordering of iterations, this memory access will not be necessarily coalesced. That is, nearby memory locations may be accessed at distant iterations wasting memory bandwidth.

CCC can avoid memory divergence by taking the memory access out of the diverging path to the non-divergent block and stacking the memory content alongside the context. In other words, excluding the co-

alesced memory access from the path and executing it with the path predicate, as shown in Figure 8(b). Therefore, CCC can be applied to the new divergent path similar to Figure 3 while preserving coalesced access pattern. The only difference is that now the context includes the memory content (`vCost`) as well.

4.3 Prioritizing the Costliest Branches

When there are multiple divergent paths taken by the warp lanes, context collection for all the branches may limit the theoretical GPU occupancy. Sometimes it might not even be possible to launch the kernel with a context stack for each and every path, due to requested capacity exceeding the limit. For example, applying CCC to all or most of the divergent paths inside the device function in Figure 7 is not possible or limits the occupancy because of limited available shared memory.

To avoid restricting the occupancy due to excessive use of shared memory, and at the same time, to avoid intra-warp divergence as much as possible, we prioritize the costliest branches, i.e. those branches for which context collection provides the most benefits. The cost of taking a divergent branch is proportional to the volume of operations inside the branch plus how infrequently it is visited by the warp lanes; later in Section 6.3 we verify this argument. Therefore, CCC applies to the longest branches with the least probability of traversal. In case of the example in Figure 7, the path belonging to latest variations are longer and more expensive; hence, are more suitable candidates for CCC.

5. CCC IMPLEMENTATION

To implement CCC, we designed a framework based on a combination of user-provided annotations identifying the paths for CCC application and an automatic compilation chain intervention that transforms the code. The user annotates the repetitive pattern and the divergent path inside the CUDA C++ kernel. Then, the framework operates alongside NVCC and applies CCC to the code automatically, as shown in Figure 9. Each transformation described in section 3 is specified by a different annotation. For example, for the code in Figure 1(a) a user needs to insert only `#CCC for const` above the `for` loop and specify the divergent code block by putting `#CCC if` above the `if` condition evaluation line. The first part of the framework, Annotated Source Modifier, marks these specific regions to transfer them to the PTX level. The second part of the compiler, PTX Source-to-source Compiler, applies CCC and the optimizations described in Section 4.

5.1 Annotated Source Modifier

This part of the framework analyzes the code, identifies specific user-specified directives in CUDA C++ source code, and enables recognition of these patterns inside the PTX code. The frameowrk replaces and also inserts assembler statement `asm` with `volatile` keyword to mark the repetitive section and the beginning and the end of the divergent code path region. `asm` statement allows arbitrary code to propagate into and

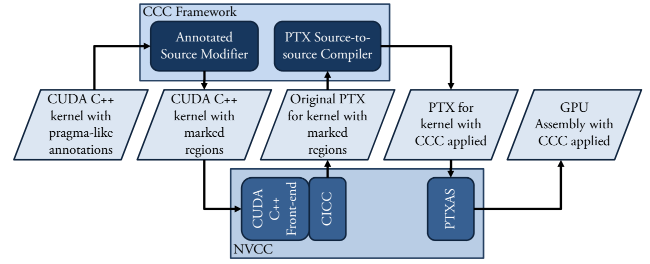
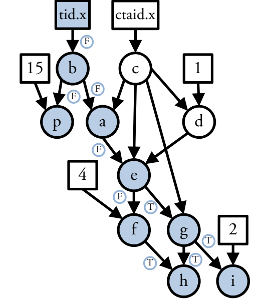


Figure 9: CCC Framework operates alongside NVCC.

```
1  mov.u32      c,  %ctaid.x;
2  mov.u32      b,  %tid.x;
3  min.s32      a,  b,  c;
4  add.s32      d,  c,  1;
5  mad.s32      e,  c,  a,  d;
6  mul.wide.s32 f,  e,  4;
7  setp.le.s32  p,  b,  15;
8  %p  bra      POST_PATH_LBL;
9  #CCC_if_marked_begin
10 add.s32      g,  e,  c;
11 sub.s32      h,  g,  f;
12 add.s32      i,  g,  2;
13 #CCC_if_marked_end
14 POST_PATH_LBL:
```

(a) PTX code.



(b) Resulting graph.

Figure 10: A PTX sample code inside the repetitive section and the resulted graph from connecting definition and usage of virtual registers.

appear inside the PTX code, without even necessarily being a valid PTX statement. In addition, `volatile` keyword prevents optimization on specified assembly statement and preserves the relative order of instructions before and after the statement. These two properties of `asm volatile()` enables marking PTX code regions of interest from inside the CUDA C++ kernel. As shown in Figure 9, the framework feeds the marked CUDA kernel into NVCC front-end— which is tightly bound to CICC (LLVM-based optimizer and PTX generator)— to yield the PTX code.

5.2 PTX Source-to-source Compiler

The second part of the framework receives a PTX source with distinct annotations that mark regions of interest including the beginning and the end of the divergent code block (similar to lines 9 and 13 in Figure 10(a)) and the immediate path after the repetitive pattern. This part outputs the PTX code with CCC and its optimizations applied, and sends the resulting PTX code to the rest of the compilation chain. Applying our technique at the PTX level is advantageous since the PTX code has a closer-to-machine assembly-like form with a limited number of instructions and directives. This facilitates reasoning about the functionality, throughput, inputs, and output of each instruction.

To apply CCC to the PTX code, It is necessary to recognize the context variables corresponding to a divergent path. To identify context registers, we generate a data dependence graph in which every node represents a PTX virtual register and every edge stands for the def-use link that defines the destination node and uses the source node. The BFS traversal starting from thread-specific special registers induces sub-graphs of virtual

registers (directly and indirectly) affected by them. Figure 10(a) shows a sample PTX code and Figure 10(b) presents its corresponding graph, in which the induced sub-graph is highlighted. In addition, the framework assigns every edge a boolean property for which a true value specifies if the def-use link between two virtual registers is established inside the divergent path. The framework then identifies a virtual register as a context if and only if, inside the sub-graph, its corresponding node's incoming edges are assigned false and at least one of its outgoing edges is assigned true. In Figure 10, registers `e` and `f` are recognized as the context.

The framework automates CCC optimizations using this analysis as well. For context compression, the framework examines a context register's parents inside the graph. If all the parent nodes are from the set of literals, function parameters, or other context registers and the instruction corresponding to the connecting edges is high-throughput and compute-only, the context is compressed. In Figure 10, virtual register `f` satisfies the compression condition. To avoid memory divergence, the index of the global memory read is examined to have a reaching definition from `tid.x` and to fulfill the coalescence. To prioritize the costliest branch, the inverse of the instruction throughput inside every branch is aggregated and compared with other branches'.

After identification of context registers, the source-to-source compiler declares appropriate resources such as shared memory buffers and inserts CCC code segments.

6. EXPERIMENTAL EVALUATION

We first briefly describe the benchmarks, then evaluate their performance with and without CCC. Lastly, we analyze the sensitivity of CCC. Experiments were performed on a Nvidia GeForce GTX 780 GPU equipped with 12 Streaming Multiprocessor from Kepler microarchitecture. Up to 2048 threads can reside on each SM while each SM contains 64K 32-bit registers. Up to 32 32-bit registers per thread and up to 24 bytes of shared memory per thread can be requested without affecting the occupancy. All GPU programs are compiled with the highest optimization level flag (-O3) for Compute Capability 3.5 on Ubuntu 14.04 64-bit with CUDA 7.0.

6.1 Benchmarks

We selected 8 real-world benchmarks from various domains. These programs demonstrate substantial amount of intra-warp divergence; hence, they can benefit from CCC. Below we introduce these benchmarks.

BFS. *Breadth-First Search* is an iterative graph traversal algorithm. CUDA implementation of BFS [13] assigns one thread to process a number of vertices and their neighbors. We used LiveJournal [20], a social network graph with approximately 4.8M vertices and 69M edges, as the input graph. We used grid-stride loop, loops with variable trip-count, and nested context collection techniques for BFS.

DQG. *Dynamical Quadrature Grids* [21] program computes the points in a quadrature grid. *Becke* kernel of DQG is used in our experiment. One thread is assigned to a point which then has to iterate over atoms two-by-two. The number of atoms can vary from 2 to 80, which creates a load imbalance between threads. We used a grid-stride loop to enable CCC.

EMIES. *Electromagnetic Integral Equation Solvers* [22] compute the electromagnetic field using Nonuniform Grid Interpolation Method (NGIM). The potential field domain is divided into subdomains of different sizes. The type of parallelization is "one-thread-per-observer". As a result, when comparing the domains, some threads in the warp may satisfy the Near-Field criterion and calculate the volume integral equation while others may not. No transformation were required. Average number of sources per box is set to 64 for the experiments.

FF. *Fractal Flames* [19] belongs to the Iterated Function Systems (IFS) class of algorithms and is based on chaos game. It involves selecting and executing a function randomly from the set of available non-linear functions. Picking and executing different functions for threads inside the warp causes task serialization. We defined 10 function variations and rendered a 2D scene with 10M random points. We used a grid-stride loop to iterative over points and *prioritized 3 costliest branches.* Since the context for all the divergent paths is 8 bytes (4 bytes for x and 4 bytes for y), and also since the original kernel does not consume shared memory, collecting up to 3 branches does not affect the occupancy.

HASH. *GPU Cuckoo hashing* [18] constructs a hash table given a set of key-value pairs. Each thread is assigned to carry out insertion of a set of key-value pairs into the hash table. Threads perform insertions simultaneously via atomic exchange operation. Some threads in the warp may need to retry insertion due to collision creating intra-warp load imbalance. We applied a grid-stride loop to enable CCC and then used the transformation for recursive functions. We also *compressed the initial context* that reduced the context size from 16 to 12 bytes. In the experiments we used 100 M randomly generated 8-byte-long key-value pairs, and the table load factor is set to 0.9.

IEFA. *Inverse Error Function Approximation* [23] involves selection and execution of one out of three possible functions depending on the input. Warp lanes that are assigned to compute the inverse error function usually take different paths. This causes traversal and execution of all three functions by the warp. We prepared the application for CCC with a grid-stride loop. For the experiments, we approximated double-precision inverse error function for 100M values.

RT. *Ray Tracing* [24] is a simple ray tracing CUDA kernel in which threads are assigned to rays and they verify if a ray hits the objects in the scene. A ray that hits an object has to update the closest hit depth and its own color. Threads inside the warp may or may not hit an object. This creates load imbalance and warp underutilization. We defined 8.8M rays (4K resolution) and 80 sphere objects in the scene. We applied a grid-stride loop to iterative over rays.

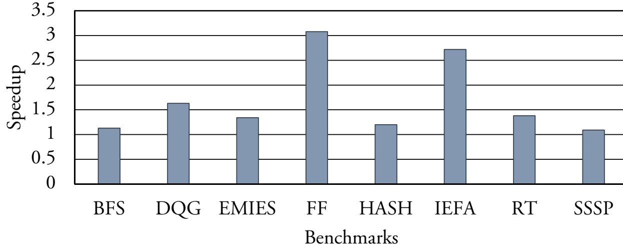


Figure 11: The kernel execution speedup via CCC. For benchmarks with iterative GPU kernel launches (BFS and SSSP) the speedup is measured based on the aggregation of kernels.

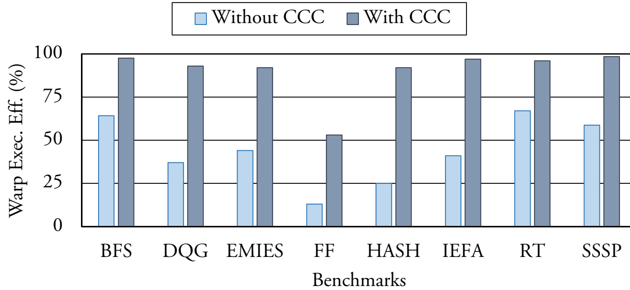


Figure 12: Warp execution efficiency comparison between the kernels with and without CCC. For BFS and SSSP the warp execution efficiency is averaged across all the kernel launches.

SSSP. *Single-Source Shortest Path* finds the shortest path to every vertex reachable from a single source vertex using iterative CUDA kernels. We used Harish et. al. [13] approach for the SSSP. A thread is assigned to process a set of vertices. Similar to BFS, the pattern of SSSP load imbalance is nested. LiveJournal [20] is our input graph. Applied transformations are the same as BFS. We also applied *memory divergence avoidance optimization* for SSSP.

6.2 CCC Performance Improvement

Overview of results. Figure 11 presents the speedups obtained by applying CCC to the eight real-world programs. These speedups are measured exclusively for CUDA kernels. On an arithmetic average, the set of benchmarks experience speedup of 1.69x due to application of CCC. We further profiled the warp execution efficiency (predicated and non-predicated averaged) of these benchmarks with and without CCC and plotted the results in Figure 12. On average, applying CCC increases the warp execution efficiency of benchmarks from 43.7% to 89.8%. We also measured the overhead introduced by CCC in terms of added shared memory and register usage per thread and reported them in Table 1. As mentioned earlier, for our GPU, up to 32 32-bit registers per thread and 24 bytes of shared memory per thread can be requested without affecting the kernel occupancy. Now we examine the results in detail.

Detailed performance analysis of benchmarks. CCC achieves the highest speedup of **3.08x** for the FF benchmark. For FF we collected the longest 3 divergent paths so as not to limit the occupancy by using extra shared memory. Although there are 10 divergent branches in the kernel, collecting the most expensive 3 of them enhanced the warp execution efficiency from 13% without CCC to 53% with CCC. This result demonstrates the importance of *branch prioritization* technique. Also note that the newest Nvidia GPU microarchitecture named Maxwell doubles the maximum shared memory available to the thread-block. More shared memory enables collecting more divergent path contexts without affecting the occupancy. Therefore, this allows higher speedup for benchmarks similar to Fractal Flames where diverging paths are numerous.

Benchmark	32-bit Reg. Usage		Shared Mem. Usage (B)	
	w.o. CCC	with CCC	w.o. CCC	with CCC
BFS	15	26	0	8
DQG	25	31	8	24
EMIES	28	<u>32</u>	0	24
FF	21	29	0	24
HASH	22	28	0	12
IEFA	24	32	0	24
RT	21	27	0	24
SSSP	19	29	0	16

Table 1: The CCC overhead in terms of resource usage (per thread). Underlined entry results from spilling two excessive registers into local memory (L1 cache) via `-maxrregcount` compiler option. The maximum theoretical occupancy is 100% in all cases.

CCC provides the next highest speedup of **2.72x** for IEFA benchmark. All three branches that can be taken by the warp lanes are collected and traversed only when full warp utilization is possible. Collecting all branches boosts the warp execution efficiency of IEFA from 41% to 97%. Similar to FF, IEFA experiences warp divergence due to dissimilar intra-warp task assignment and contains relatively long compute-only divergent task paths. These features make FF and IEFA the most benefiting benchmarks from CCC.

The next benchmark for which CCC shows a relatively high speedup is DQG with speedup of **1.63x**. The different load volume assignments to each GPU thread in the original DQG results in the warp execution efficiency of 37% while CCC enhances to 93%.

The RT benchmark has the highest amount of warp execution efficiency in the original kernel with 67%. This is because warp lanes are assigned to contiguous rays which are more likely to hit an object in the scene. Nevertheless CCC provides speedup of **1.38x** for RT while increasing the warp efficiency to 96%.

Further, CCC increased the warp execution efficiency of EMIES from 44% to 92% resulting in speedup of **1.34x**. EMIES divergent paths are long, but also containing global memory accesses. EMIES is the only benchmark where applying CCC can limit the maximum theoretical occupancy to 75% by requesting 34 registers per thread. However, for this benchmark, we pass the compiler option `-maxrregcount 32` to enforce the compiler to spill two registers into the local memory. Since the kernel asks for 24 bytes of shared memory per

thread, 16 KB of shared memory in the SM is left for L1 cache, which is just enough for spilled registers.

Finally, benchmarks HASH, BFS, and SSSP are primarily memory-bound benchmarks; hence application of CCC results in smaller, yet significant, performance improvements. HASH benchmark relies heavily upon 8-byte-long atomic exchange operation on table entries. Entries accessed by threads inside the warp reside in distant memory segments. These accesses create non-coalesced and cache-unfriendly global memory requests which represent a major performance bottleneck for this kernel. As a result, although the warp execution efficiency increases from 25% to 96%, CCC provides smaller speedup of **1.20x**. The speedup of HASH without *context compression* optimization is lower – nearly **1.17x**.

Benchmarks BFS and SSSP are memory-bound. Both suffer from a great deal of load imbalance; however, the set of non-coalesced memory accesses to the content of neighboring vertices represents a major performance bottleneck. Therefore, applying nested CCC to BFS and SSSP provides smaller speedups of **1.13x** and **1.09x** respectively. The speedup in SSSP is the lowest due to higher amount of memory accesses; SSSP introduces additional memory accesses to the bit mask and edge value buffers. Also without *memory divergence avoidance* optimization, SSSP speedup is 1.06x. Considering the irregularity of the input graph, original BFS and SSSP kernels exhibit 58% and 64% warp execution efficiency on average. Note that this is due to early and late CUDA kernels in which most threads do not take the divergent paths. Kernels for middle graph algorithm iterations carry out most of the computation and exhibit warp execution efficiency as low as 14%. Finally, very regular graphs benefit only slightly from CCC due to lack of imbalance. For example, original BFS and SSSP with roadNet-CA [25] as a very regular 2D graph shows 83% and 85% warp execution efficiency on average. CCC improves the warp efficiency of BFS and SSSP kernels to 94% and 95% and gives 1.02x performance improvement for both.

6.3 Sensitivity Analysis

Next, using synthetic programs, we study the sensitivity of CCC to (a) varying amount of warp divergence over warp threads and (b) varying execution lengths of the divergent path. Original GPU kernel is a grid-stride loop executed 2^{30} times (aggregated over all CUDA kernel launched threads). The context size is 4 bytes.

Varying intra-warp divergence. Figure 13 compares the execution time and the warp execution efficiency of the synthetic GPU kernel when executed normally and when CCC applied. The loop contains a divergent path with 20 FMAD instructions. We repeated the experiments each time with different number of threads inside the warp taking the divergent path (x axis). Increasing this number also increases the amount of load that has to be carried out by the CUDA kernel.

The left plot in Figure 13 shows that the original kernel takes an approximately constant amount of time to finish with different amount of intra-warp divergence

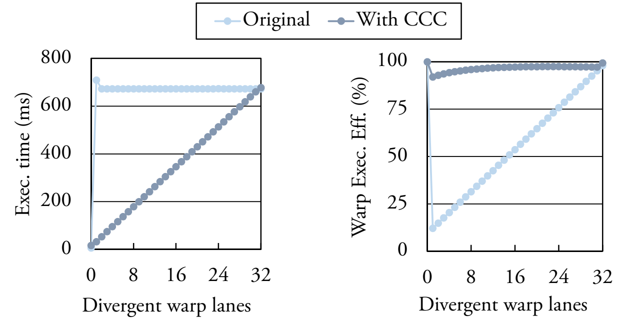


Figure 13: CCC performance enhancement compared to the original divergent kernel over different amount of intra-warp divergence (and hence workload imbalance). The divergent path contains 20 FMAD operations.

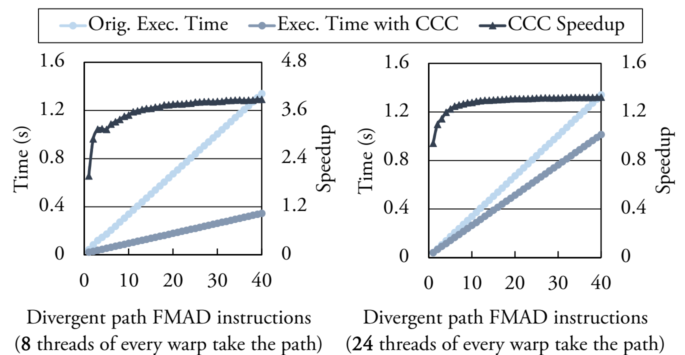


Figure 14: Sensitivity of CCC against different execution path lengths plotted for two different amounts of intra-warp divergence.

and workload imbalance. This is a natural behavior of a SIMD device. But when CCC is applied, we see that by increasing the workload, the execution time grows linearly. It means CCC provides work efficiency. The plot on the right in Figure 13 presents the warp execution efficiencies. We observe that although the warp execution efficiency of the original kernel increases proportionally to the amount of intra-warp divergent tasks, the kernel with CCC always has a high warp efficiency (%96.6 on average) regardless of the amount of intra-warp divergence. *In summary, both plots demonstrate the effectiveness of CCC in form of its resistance against various amounts of load imbalance.*

Varying divergent path length. Figure 14 shows the execution times for the original and CCC kernels for different divergent path lengths. The left and right plots demonstrate it for when $\frac{1}{4}$ and $\frac{3}{4}$ of the warp lanes take the divergent path, respectively. In both plots, it is clear that as the length of the divergent path increases, the speedups approach to inverse of utilized threads in the original codes, i.e. $\frac{4}{1}$ and $\frac{4}{3}$ respectively. It is evident that CCC shows more speedup where the divergent path is longer. In addition, by comparing the two plots, we realize that to cope with the overhead of CCC, either the divergent path has to be long or the divergent ratio should be high. In both plots, the speedup is less than

one only when the divergent path contains only one operation and at the same time 24 threads inside every warp take the path. In all other cases, CCC provides speedup higher than one.

7. RELATED WORK ON DIVERGENCE

Next we discuss microarchitectural and software solutions to address the SIMD thread divergence problem.

Microarchitectural Solutions. Although these techniques cannot be exploited on available hardware, clever solutions can guide future designs. *Dynamic Warp Formation* (DWF) [1] is the basis for many microarchitectural solutions. DWF merges threads from different warps but with the same PC to form new warps with no thread divergence. To enhance DWF performance, Meng et. al. propose *Dynamic Warp Subdivision* (DWS) [2] and Rhu et. al. [3] suggest *SIMD lane permutation* (SLP). Furthermore, Narasiman et. el [26] suggest *Large Warp Microarchitecture* (LWM) in which fewer but wider warps can create sub-warps that match SIMD width size when facing branch divergence.

Compaction techniques have also been proposed to remedy the SIMD divergence problem. Fung et. al. [4] offer *Thread Block Compaction* (TBC) to exploit control flow locality between threads of a block for divergent paths. Unlike our solution CCC, TBC makes the warps synchronize at divergent branches to provide homogeneous tasks for warp lanes. To avoid the overhead of unnecessary compaction on non-diverging branches or workloads, Rhu et. al. [5] propose CAPRI, a *compaction-adequacy predictor* influenced by branch predictors. Moreover, Vaidya et. al [6] attempted to harvest dead execution cycles and position SIMD channels in order to group enabled channels together.

Other microarchitectural techniques focus on efficient scheduling for thread divergence and are complementary to CCC. Kim and Batten [27] propose a fine-grained hardware worklist that acts as a distributed queue to provide load balance in data-driven computations. Doubling stage resources in processing pipelines has also been popular [28]. Rhu and Erez [29] examine a dual-path execution model provided by two PC reconvergence stacks and two register scoreboards in order to expose the warp scheduler to more parallelism when facing divergent execution paths. To extend this solution, [30] replaces the reconvergence stack with two warp split and warp reconvergence tables. Rogers et. al. [31] propose *Divergence-Aware Warp Scheduling* (DAWS) for a cache-conscious warp scheduling upon divergence. Also, [32] and [33] suggest reconvergence methods for GPU kernels with unstructured and recursive control flow.

Software Solutions. Not requiring hardware modifications, software solutions for thread divergence are of great importance; however, existing strategies introduce limitations that restrict their usage. Branch and data herding [12] eliminates branch divergence by guiding all the threads in the SIMD group to take the path with the majority vote. In return herding expects and accepts errors in the output. Similarly, [7], [8], and [9] steer the warp lanes to take one execution path. The problem with such techniques is the lack of systematic reliability and applicability. Unlike CCC, these approaches do not take methodical measures to cope with the divergence problem, do not guarantee utilizing all the warp lanes by relying upon warp lanes majority voting, do not devise task accumulation strategies, and need information from the program and the input to schedule the traversal of divergent paths. These issues prevent wide employment of these solutions. On the other hand, CCC and its transformation and optimization techniques offer methodical approaches to guarantee warp execution enhancement in divergent GPU kernels and can be completely realized and implemented in compile time.

Zhang et. al. [10] try to eliminate thread divergence via thread-data remapping. Unfortunately, this solution not only needs global memory accesses to realize the redirected position of the appropriate data for the warp lanes, it does not preserve GPU kernel autonomy by involving the CPU. Bauer et. al. [34] suggest an intra-CTA producer-consumer model based on which warps can have unique tasks for their threads. As opposed to CCC, this model does not support irregular data-dependent tasks; in other words, the quantity of each task has to be known at compile-time. Tzeng et.al. [11] propose a task management mechanism for irregular parallel workloads based on task donation and stealing. Nonetheless, this technique suffers from GPU underutilization and heavy use of global queues and associated global locks. Merrill et al. [35] enhance the warp execution efficiency of BFS graph traversal via efficient expansion of unequal adjacency lists. Our work is different from [35] in two major ways. First, thread divergence problem in [35] appears as a form of load imbalance on the tasks while it is assumed all the tasks will be carried out. However, CCC allows the existence of conditionals on the tasks in addition to imbalance loads. Second, CCC does not require additional storage to collect the frontiers; instead, it defers processing of tasks via stacking. Our work can also be viewed as a form of in-place stream compaction and consumption.

The load imbalance and consequently thread divergence caused by nested parallelism in GPUs have also been the subject of recent work. Han et. al. [36] introduce loop merging to reorder the code blocks inside a loop with varying trip-count and improve the performance ; however, unlike CCC, the solution does not guarantee full warp execution efficiency. Yang and Zhou created CUDA-NP [37], a source-to-source compiler that transforms GPU codes with parallel sections using the idea of master and slave threads. However, fixed number of slave threads for a master thread can hurt the performance in irregular workloads. In [38] Lee et. al. also propose a framework supporting a number of widely-used parallel patterns for efficient nested parallelism. [39] introduces warp-aware trace scheduling for GPUs based on speculating loads and arithmetic instructions upon divergence in order to exploit ILP. Recently, Schaub et. al. [40] evaluated compiler techniques that aim to mitigate divergence against larger SIMD widths. [41] and [42] offer static code analyzers

helping GPU developers to optimize the code manually. Also, [43] and [44] provide profile-guided approaches to recognize and then optimize code regions exhibiting divergence. These works complement our work.

8. CONCLUSION

We introduced a software technique named Collaborative Context Collection (CCC) that overcomes the SIMD inefficiency of GPU kernels containing thread divergence due to intra-warp load imbalance or dissimilar task assignment. CCC collects the context of divergent threads at the stacks inside the shared memory and retrieves them such that a uniform task is performed by all the warp lanes. CCC increases the warp execution efficiency of real-world applications containing divergent execution paths by up to 56% and provides average speedup of 1.69x (maximum 3.08x).

9. ACKNOWLEDGEMENTS

This work is supported by NSF Grants CCF-0905509, CNS-1157377, CCF-1318103, CCF-1524852, and CCF-1423108 to UC Riverside.

10. REFERENCES

[1] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO-40*, pp. 407–420, 2007.

[2] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA*, pp. 235–246, 2010.

[3] M. Rhu and M. Erez, "Maximizing simd resource utilization in gpgpus with simd lane permutation," in *ISCA*, 2013.

[4] W. Fung and T. Aamodt, "Thread block compaction for efficient simt control flow," in *HPCA*, pp. 25–36, Feb. 2011.

[5] M. Rhu and M. Erez, "Capri: Prediction of compaction-adequacy for handling control-divergence in gpgpu architectures," in *ISCA*, pp. 61–71, 2012.

[6] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi, "Simd divergence optimization through intra-warp compaction," in *ISCA*, pp. 368–379, 2013.

[7] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *GPGPU-4*, 2011.

[8] D. Connors, S. Saleh, T. Joshi, and R. Bueter, "Data-driven techniques to overcome workload disparity," in *Workshop on Irregular Applications: Architectures and Algorithms*, pp. 41–48, 2014.

[9] S. Frey, G. Reina, and T. Ertl, "Simt microscheduling: Reducing thread stalling in divergent iterative algorithms," in *Parallel, Dist. and Network-Based Processing*, 2012.

[10] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping," in *ICS*, 2010.

[11] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the gpu," in *Conference on High Performance Graphics*, pp. 29–37, 2010.

[12] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," in *PACT*, pp. 427–428, 2012.

[13] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *HiPC*, 2007.

[14] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu, *Performance Analysis and Tuning for General Purpose Graphics Processing Units*. Morgan & Claypool Publishers, 1st ed., 2012.

[15] M. Harris and M. Garland, "Chapter 3 - optimizing parallel prefix operations for the fermi architecture," in {*GPU*} *Computing Gems Jade Edition* (W.-m. W. Hwu, ed.), 2012.

[16] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Conf. on High Perf. Graphics*, 2009.

[17] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*, pp. 382–383. Pearson Education, 2013.

[18] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Chapter 4 - building an efficient hash table on the {GPU}," in {*GPU*} *Computing Gems Jade Edition* (W.-m. W. Hwu, ed.), pp. 39 – 53, Morgan Kaufmann, 2012.

[19] C. Schied, J. Hanika, H. Dammertz, and H. Lensch, "Chapter 18 - high-performance iterated function systems," in {*GPU*} *Computing Gems Emerald Edition* (W.-m. W. Hwu, ed.), pp. 263 – 273, Morgan Kaufmann, 2011.

[20] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *ACM KDD*, pp. 44–54, 2006.

[21] N. Luehr, I. Ufimtsev, and T. Martinez, "Chapter 3 - dynamical quadrature grids: Applications in density functional calculations," in {*GPU*} *Computing Gems Emerald Edition* (W.-m. W. Hwu, ed.), pp. 35–42, 2011.

[22] S. Li, R. Chang, A. Boag, and V. Lomakin, "Fast electromagnetic integral-equation solvers on graphics processing units," *Antennas and Propagation Magazine, IEEE*, vol. 54, pp. 71–87, Oct 2012.

[23] M. Giles, "Chapter 10 - approximating the erfinv function," in {*GPU*} *Computing Gems Jade Edition* (W.-m. W. Hwu, ed.), pp. 109 – 116, Morgan Kaufmann, 2012.

[24] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed., 2010.

[25] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *CoRR*, vol. abs/0810.1355, 2008.

[26] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *MICRO-44*, pp. 308–317, 2011.

[27] J. Kim and C. Batten, "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists," in *MICRO-47*, pp. 75–87, 2014.

[28] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained gpu performance," in *ISCA*, pp. 49–60, 2012.

[29] M. Rhu and M. Erez, "The dual-path execution model for efficient gpu control flow," in *HPCA*, pp. 591–602, 2013.

[30] A. ElTantawy, J. Ma, M. O'Connor, and T. Aamodt, "A scalable multi-path microarchitecture for efficient gpu control flow," in *HPCA*, pp. 248–259, Feb 2014.

[31] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *MICRO-46*, 2013.

[32] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "Simd re-convergence at thread frontiers," in *MICRO-44*, pp. 477–488, 2011.

[33] X. Huo, S. Krishnamoorthy, and G. Agrawal, "Efficient scheduling of recursive control flow on gpus," in *ICS*, 2013.

[34] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *SIGPLAN PPoPP* pp. 119–130, 2014.

[35] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *SIGPLAN PPoPP*, pp. 117–128, 2012.

[36] T. D. Han and T. S. Abdelrahman, "Reducing divergence in gpgpu programs with loop merging," in *GPGPU-6*, 2013.

[37] Y. Yang and H. Zhou, "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," in *SIGPLAN PPoPP*, pp. 93–106, 2014.

[38] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware mapping of nested parallel patterns on gpus," in *MICRO-47*, pp. 63–74, 2014.

[39] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy, "Warp-aware trace scheduling for gpus," in *PACT*, 2014.

[40] T. Schaub, S. Moll, R. Karrenberg, and S. Hack, "The impact of the simd width on control-flow and memory divergence," *ACM TACO*, vol. 11, pp. 54:1–54:25, 2015.

[41] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira, "Divergence analysis and optimizations," in *PACT*, 2011.

[42] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira, "Divergence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 35, pp. 13:1–13:36, Jan. 2014.

[43] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira, "Profiling divergences in gpu applications," *Concurrency and Computation: Practice and Experience*, 25(6), 2013.

[44] S. Sarkar and S. Mitra, "A profile guided approach to optimize branch divergence while transforming applications for gpus," in *India Software Engineering Conf.*, 2015.