Reducing Divergence in GPGPU Programs with Loop Merging

Tianyi David Han    Tarek S. Abdelrahman
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{han,tsa}@eecg.toronto.edu

Abstract

Branch divergence can incur a high performance penalty on GPGPU programs. We propose a software optimization, called loop merging, that aims to reduce divergence due to varying trip-count of a loop across warp threads. This optimization merges the divergent loop with one or more outer surrounding loops into one loop. In this way, warp threads do not have to wait for each other in each outer loop iteration, thus improving execution efficiency. We implement loop merging in LLVM. Our evaluation on a Fermi GPU shows that it improves the performance of a synthetic benchmark and five application benchmarks by up to $1.6 \times$ and $4.3 \times$ respectively.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Optimization

General Terms Algorithms, Performance, Experimentation, Measurement

Keywords Branch divergence, Loop optimizations, Performance evaluation

1. Introduction

General-Purpose Graphics Processing Units (GPGPUs) have become an indispensable platform for High Performance Computing (HPC). However, their massive computing power can only be exploited when programs are carefully optimized for the GPU architecture.

GPU memory performance has been the focus of most GPU optimizations and remains the first-order factor to consider. However, the performance penalty of divergent execution, i.e., warp threads taking different execution paths, is an increasingly prominent issue. A divergent branch causes each warp thread to execute both branch paths. A loop with a varying trip-count across warp threads causes all of them to wait for the one that executes the maximum number of iterations. Since such data-dependent control flow is abundant in applications, it is critical to address this inefficiency.

In this paper we propose a software-based optimization, called loop merging. It aims to reduce divergence due to varying trip-count of a loop across warp threads. The optimization merges the divergent loop with one or more outer surrounding loops into a single loop. This allows warp threads not to wait for each other in each outer loop iteration, resulting in a more compact execution schedule.

Loop merging generalizes the standard loop coalescing [17] to allow for non-perfect and while loops, and for control-flow in-between loops. Loop coalescing is traditionally used to improve parallelism and reduce synchronization overhead [17]. It has recently been used to improve memory coalescing and to reduce divergence of a loop nest [8], in the context of irregular sparse matrix applications. Our generalized form of the optimization is more applicable and thus more effective in practice. Indeed, loop coalescing cannot be applied to any of the benchmark applications evaluated in this paper.

We implement loop merging in LLVM, and evaluate its performance impact by using one synthetic benchmark and five GPGPU application benchmarks. The synthetic benchmark is parameterized to model loop characteristics that affect the benefit of the optimization. Experiments on a Fermi GPU show that loop merging is effective in reducing divergence; it improves performance of the synthetic and application benchmarks by up to $1.6 \times$ and $4.3 \times$ respectively. Although this benefit depends on loop characteristics and program inputs, the optimization does not degrade performance in any significant way.

---

1 Warp threads, defined in Section 2.1, are a group of GPU threads that must execute in a SIMD fashion.
The remainder of the paper is organized as follows. Section 2 provides background on NVIDIA GPUs and divergent execution. Section 3 and 4 describe loop merging and its compiler algorithm respectively. Section 5 presents the evaluation methodology and results. Section 6 discusses related work. Section 7 concludes and gives directions for future work.

2. Background

2.1 NVIDIA GPU and CUDA

An NVIDIA GPU consists of multiple Streaming Multiprocessors (SMs). Each SM contains a number of multiply-add units, or cores, that execute in a Single-Instruction Multiple Data (SIMD) fashion [11]. GPUs support a C-extended programming model, called Compute Unified Device Architecture (CUDA). This model allows both host (CPU) code and device (GPU) code, or kernels, to be written in a single source program. A kernel is similar to an entry function in host code, except that it is executed by each GPU thread. Launching a kernel for GPU execution involves calling the kernel function (from the host code) along with a specification of the space of GPU threads that execute it. This space is referred to as a grid and it consists of multiple thread blocks, each of which has multiple threads. Each thread block is scheduled to run (until completion) on a single SM. Threads in a thread block are executed on the SIMD cores of the SM in groups of 32, or warps. Therefore, threads from the same warp, or warp threads, must execute the same instruction at any given time.

2.2 Control Flow Support

Warp threads can take different execution paths, or diverge, despite their SIMD execution model. This scenario is often called branch divergence because it occurs when a data-dependent branch condition evaluates to different values across warp threads.

Through the inspection of GPU assembly code, obtained using the kernel disassembler cuobjdump, and the limited NVIDIA documentation on the GPU Instruction Set Architecture (ISA) [10], we describe three mechanisms of handling branch divergence on a Fermi GPU. They are all based on prediction in hardware: an instruction issued to a warp is executed by a warp thread only if a per-thread condition code, or predicate, is true. Predicates for all warp threads form a predicate mask. Prediction is exposed to the GPU ISA, allowing the compiler to generate predicated instructions.

Consider the if statement in Figure 1a with a data-dependent branch condition P. The first and simplest mechanism is branch predication, shown in pseudo-code in Figure 1b. Instructions in the then and else paths are predicated with condition P and !P respectively. Warp threads go through both branch paths even when the branch is not divergent, lowering execution efficiency.

The second mechanism addresses this inefficiency by extending the instruction sequence with convergence pre-check, as shown in Figure 1c. Warp threads execute a BRA.U instruction [10] before doing predicated execution of the then path. This instruction branches to ELSE_PATH only if the condition !P is true for all warp threads. Effectively it skips the then path if no thread takes that path. Similarly, the BRA.U at line 5 skips the else path if no thread takes it. The two additional BRA.U instructions add overhead and may limit the compiler in scheduling instructions. We observe that the NVIDIA compiler does not generate convergence pre-check when a branch path (either then or else) has less than 3 instructions.

While the above two mechanisms are ideal for simple branches, they cannot efficiently support nested branches and loops, let alone arbitrary control flow. The third mechanism, described by Fung et al. [5], provides generic support by using a hardware predicate stack. Each stack entry specifies the execution path of a subset of warp threads (or a sub-warp), in the form of the next Program Counter (PC) and a predicate mask. The top stack entry is the one under execution. When a warp reaches a divergent branch, it is divided into two sub-warps. The branch target for each sub-warp (along with its predicate mask) is pushed onto the predicate stack separately. To avoid completely serializing their execution, the two sub-warps reconverge at the immediate post-dominator (IPDOM) of the branch block. The

---

2 Basic block Y is a post-dominator of basic block X iff all paths from X to the program exit pass through Y. Y is an immediate post-dominator of X iff there is no other basic block Z such that Z is a post-dominator of X and Y is a post-dominator of Z.
top stack entry is popped when the corresponding sub-warp reaches the reconvergence point. Fung et al. [5] refer to this mechanism as the PDOM reconvergence model.

The Fermi GPU ISA contains instructions that suggest the use of a predicate-stack-like mechanism, although no official NVIDIA documentation confirms it. For example, the SSY instruction is documented to set a future synchronization point before a potentially divergent branch [10]. The synchronization point can be viewed as the reconvergence point in the PDOM model. Figure [13] shows pseudo-instructions generated for the branch in Figure [13]. The SSY instruction at line 1 sets the reconvergence point of the branch to the end of the branch (END_IF), which is the IPDOM of the basic block containing the BRA instruction. A field .S is set in the last instruction of each branch path (lines 5 and 9), suggesting that the instruction also pops off the corresponding entry from the predicate stack before reaching the reconvergence point [21].

1
2
3
4
5
6
7
8
9
10

```
while (1) {
  if (P1) {
    // THEN_PATH
  } else {
    // ELSE_PATH
  }
  if (P2)
    break;
  // EPILOGUE
}
```

(a) Source code.

(b) CFG (using SSY).

(c) CFG (using BRK).

Figure 2. A loop with the exit in a branch in the loop body.

The statically determined reconvergence point may not be the earliest point to reconverge at run-time. Consider the loop in Figure [2a] whose exit lies in one of the branch paths in the loop body (i.e., the break statement). Figure [2c] shows the corresponding Control Flow Graph (CFG), constructed from the output of cuobjdump. The break statement shifts the IPDOM of BB2 from the epilogue (BB5) to the loop exit (BB6), delaying the reconvergence point of @P1 BRA (specified in the SSY instruction). This serializes the execution of warp threads until the end of the loop once they diverge at line 2 in one iteration.

The GPU ISA includes two extra instructions, PBK and BRK [10], to address this performance issue. Figure [2c] shows the CFG that utilizes them, again obtained using cuobjdump. Compared to Figure [2b] the SSY LOOP_EXIT instruction in BB1 is shifted to SSY EPILOGUE in BB2, which sets the reconvergence point for @P1 BRA back to the epilogue as if the loop exit edge is not present. In this way, all warp threads that remain in the loop reconverge at the end of each iteration. The loop exit BRA instruction in BB4 is replaced with a special BRK instruction, whose “break” target is specified by the PBK instruction inserted in BB1.

We observe that the NVIDIA compiler does not always generate PBK and BRK instructions for the loop scenario in Figure [2a] Thus, as described in Section 3.1 we merge loops in such a way that loops with this structure have early reconvergence even when PBK and BRK are not used.

2.3 Impact of Branch Divergence

Branch divergence hurts performance due to lower utilization of cores. If the branch in Figure [1a] is unbiased, only half of the warp threads are active when executing each branch path, resulting in only 50% core utilization. This penalty is present under all three mechanisms of divergent execution.

Further, with a loop whose trip-count varies across warp threads, all threads must wait for the one that executes the maximum number of loop iterations. The resulting performance loss is more significant, depending on the loop body size and the trip-count variance. We refer to this form of divergence loop-induced divergence. Even worse, such “divergent” loops are often nested within one or more outer loops. This exacerbates the divergence penalty, which now occurs in every outer loop iteration.

3. Loop Merging

We propose an optimization, called Loop Merging (LM), that aims to reduce the loop-induced divergence in nested loops, described above. It does so by merging the nested loops into a single loop. Its net effect is that, instead of waiting for each other (at the end of the inner loop) in every outer loop iteration, each warp thread immediately starts the next outer loop iteration once it finishes the current one. Therefore, LM allows warp threads to concurrently execute inner loop iterations from different outer loop iterations, making the SIMD execution schedule more compact.

(a) Code pattern.

(b) Sample execution.

Figure 3. Code pattern targeted by loop merging.

The loop nest in Figure [5a] is used to illustrate the idea and benefit of LM. The outer loop is preceded by outer_prologue and followed by outer_epilogue. It encloses an inner loop around inner_loop_body, along with inner_prologue and inner_epilogue. outer_cond_expr and inner_cond_expr are exit condition expressions for the outer and inner loops respectively.

Consider 3 warp threads executing this kernel segment. Thread 1 (T1) executes 2 outer iterations; Thread 2 and 3 (T2 and T3) execute 3 outer iterations. Figure [5b] gives, for each thread, the number of inner iterations executed in each
outer iteration. The SIMD execution schedule of this loop nest, under the PDOM reconvergence model, is shown in Figure 4a. Since warp threads reconverge before executing inner_epilogue, T2 and T3 must wait for T1 in the first outer iteration, and similarly T1 and T2 must wait for T3 in the second outer iteration. T1 is idle for the entire third outer iteration. Clearly, there are huge “gaps” in this execution schedule, leading to low core utilization.

Figure 4. SIMD execution schedule of the example in Figure 3.

Figure 4b shows the schedule under loop merging. When T2 and T3 finish the inner loop of the first outer iteration, they immediately transition to the inner loop of the next outer iteration, by executing inner_epilogue of the current iteration and inner_prologue of the next. T1 waits for T2 and T3 during this “transition” phase. Subsequently, T2 and T3 re-join T1 to execute the inner loop body, albeit from a different outer iteration. Similarly, when T2 finishes its second outer iteration, T1 and T3 are stalled to allow T2 to immediately transition to the last outer iteration. Overall, this schedule aims for maximal core utilization during the execution of inner loop body, although it may result in divergent execution of the prologue and epilogue. In contrast, the original schedule always favors convergent execution of prologue and epilogue, overlooking divergence in inner loop body execution. Since the inner loop body is often longer and is executed more frequently than the prologue and epilogue, the schedule under loop merging is likely to be more compact, as is the case in this example.

3.1 Code Transformation

Figure 5a shows the transformed code after applying LM to the code pattern in Figure 3a. It has a single merged loop that encloses two branch paths. One path is the original inner_loop_body. The other is the code executed during “transition” phase, i.e., inner_epilogue of the current outer iteration followed by inner_prologue of the next outer iteration (if any). In this way, the notion of outer vs. inner iterations is eliminated altogether.

The exit condition for the merged loop is the same as that for the original outer loop, which gets evaluated at line 9. Therefore, a temporary variable outer_cond is introduced to store the result of outer_cond_expr; it is then checked at line 13. Further, the merged loop is guarded by an initial evaluation of outer_cond_expr. This is to preserve correctness when the original loop nest is never entered.

The temporary variable outer_cond is checked every merged loop iteration (line 13) even though it is only updated every outer loop iteration (line 9). An alternative to this redundant checking is to replace the merged loop with while(1) and add a break statement after line 11 when outer_cond_expr is false. We opt not to adopt this form of the merged loop because it is equivalent to that shown in Figure 2a where the compiler may not generate efficient code using PBK and BRK instructions.

Figure 5. Applying LM to the example in Figure 3.

Figure 5b shows the execution schedule of the optimized code for the example in Figure 3a. It is slightly different from the one in Figure 4b. For example, after T2 and T3 transition from the first outer iteration to the next, T1 executes the inner loop body alone even though T2 and T3 can perfectly join it. This is because, when some of the warp threads go through the “transition” code, others are allowed to execute the inner loop body in the same (merged) iteration.

5 The use of a temporary variable is necessary here because re-evaluation of outer_cond_expr at line 13 may produce a different value (from that at line 9) due to program state changes made by inner_epilogue at line 10, thus breaking program correctness.
tion. In contrast, the expected schedule requires “regular” warp threads to wait for “transitioning” threads. We opt not to enforce this behavior for two reasons. First, we avoid the use of a warp-vote operation, which incurs a high overhead (up to 5 instructions) for each merged iteration. Second, it may not always produce a more compact schedule. In Figure 5a delaying the execution of the inner loop body (in T2 and T3) leads to more convergent execution of future “transition” code (in T1 and T2). This is aligned with observations made in our previous work [4] on the benefit of delaying loop iterations to reduce divergence.

### 3.2 Discussion

The benefit of LM is affected by several factors. The first is the variance of inner loop trip-count across warp threads, which affects the amount of “gaps” in the original execution schedule. A low variance translates into little room for loop-induced divergence reduction to begin with.

LM may not remove all loop-induced divergence because the total number of merged iterations executed by each warp thread may be inherently different, as is the case in Figure 5a. Hence the second factor is the variance of the merged loop’s trip-count across warp threads. A higher variance limits the benefit of the optimization. The trip-counts of both outer and inner loops contribute to this variance.

The third factor is the penalty of divergent execution of the “transition” code. It varies directly with 1) the degree of such divergence, 2) the frequency of executing such “transition” phases, and 3) the length of the “transition” code relative to the inner loop body. The degree of divergence depends on when each warp thread makes the transition. At worst, each transition from each warp thread happens in a separate merged iteration, resulting in a $32 \times$ slow down of “transition” code execution. The execution frequency varies inversely with the average inner loop trip-count.

The fourth factor is the potential degradation of existing memory coalescing when warp threads execute inner loop bodies from different outer loop iterations. The impact of this degradation depends on the extent to which accesses in the original code are coalesced. In many cases, kernels that exhibit loop-induced divergence tend to have a lower number of thread memory requests coalesced into a single memory transaction, simply because fewer threads are active. This makes the potential memory divergence penalty less significant.

The last factor is the dynamic instruction overhead of the code transformation, which is one branch basic block per merged iteration.

### 3.3 Applicability and Legality

It is worthwhile to emphasize the generality of the code pattern targeted by LM, particularly in relation to loop coalescing. First, the loop nest can contain arbitrary natural loops with potential early exits. Thus, the loops may be either counted or have no explicit iterator, allowing the merging of while loops. The loops may or may not carry dependence. Second, the loops do not have to be perfectly nested, allowing the presence of inner_prologue and inner_epilogue. LM can even be applied when the inner loop is inside one or more branches of the outer loop body.

Third, the inner loop body is allowed to contain further control flow (e.g., loops); the same applies to inner_prologue and inner_epilogue. Last, the outer loop may not even be explicitly present, because there is an implicit one around the kernel that assigns thread blocks to SMs. To open the possibility of applying LM, we can reveal this outer loop in the kernel code, by launching just enough thread blocks that achieve the highest parallelism level and distributing work among them explicitly. Note that this process is always legal, even though the grid size of kernel launch is usually not known until runtime. However, applying LM may not be beneficial for a small grid, because the revealed outer loop has a low trip-count.

More formally, LM is applicable to a set of two or more nested natural loops, with only two restrictions: 1) the loop nest cannot contain any explicit or implicit synchronization (e.g., __syncthreads and warp-vote operations), and 2) the exit targets of the inner-most loop must be within the outer-most loop; in other words, it is not allowed to jump directly out of the entire loop nest from the inner-most loop. The rationale for the second restriction will become clear once we describe the compiler algorithm for LM.

LM is always legal because it does not change the order of computation (e.g., loop iterations) done by each thread. It simply “re-aligns” computation across warp threads to make the SIMD execution more convergent.

### 4. Compiler Algorithm for LM

We describe the compiler algorithm for LM in the form of a CFG transformation, although nothing prevents LM from being applied at the source level. We first illustrate the transformation on a base case, using the code pattern in Figure 5a and then describe how it can be extended to support the general case, defined in Section 3.3. Although the algorithm is presented only for two loops, the optimization is applicable to a nest with more than two loops.

Consider the base case when both outer and inner natural loops have a single exit block, and the exit block of the inner loop is its header. In other words, inner_prologue, inner_epilogue, and inner_loop_body in the code pattern in Figure 5a have no early exits. Figure 5b shows the CFG corresponding to this code. BB1 and BB3 represent single basic blocks. Each of BB2, BB4 and BB5 represents a group of basic blocks with control flow in between. For simplicity, we will refer to them as single basic blocks. outer_cond in BB1 and inner_cond in BB3 are compiler-generated temporary variables that store the exit condition for the outer and inner loop (i.e., outer_cond_expr and inner_cond_expr in Figure 5a) respectively. An important observation from
Figure 6. CFG for the code pattern in Figure 3a. The outer_prologue and outer_epilogue are omitted for simplicity.

This figure is that, when excluding the inner loop’s header block (BB3), which is called the header in the rest of the section, the remaining CFG can be partitioned into two sub-CFGs, each of which forms a loop with the header. The first sub-CFG is naturally the inner loop body (BB4), which we name as INNER CFG. It can be reached from the header via edge e_{ii} and exits to the header via edge e_{io}. The second sub-CFG contains the remaining BBs, which we name as TRANSITION CFG because it represents the “transition” code mentioned in Section 3.1. Similarly, it can be reached from the header via edge e_{ti} and exits to the header via edge e_{to}. The essence of LM is to transform the CFG such that BB3 becomes the common header block of the two loops that contain the two sub-CFGs, facilitating the merging of the two loops.

Figure 7 shows the CFG after the transformation, which consists of the following four steps:

1. A copy of the TRANSITION CFG is made, which we name as TRANSITION’.
2. The outgoing edge from the header to the TRANSITION CFG (e_{ti}) is redirected to the TRANSITION’ CFG (e’_{ti}).
3. A new basic block BB6 is created to join the outgoing edges from the INNER and TRANSITION’ CFGs (e’_{io} and e’_{to}) before they reach the header. The new edge e_{m} from BB6 to the header is the back edge of the merged loop.
4. The exit edge of the TRANSITION’ CFG is redirected to BB6, forming edge e_{exit}. BB6 contains a single conditional branch instruction that jumps to the original exit target of the TRANSITION’ CFG if outer_cond is 0. This step is equivalent to eliminating the use of break statement in Figure 3a and is a special case of the transformation in [20] that merges multiple loop exits into one. We refer to this transformation as exit shifting and reuse it later when the base case is generalized.

It is important to keep the original TRANSITION CFG, which preserves correctness when the execution does not even reach the inner loop header. On the other hand, a part of the TRANSITION CFG, which is grayed out in Figure 7, is expected to be eliminated by Dead Code Elimination.

Overall, this transformation process relies heavily on two assumptions about the inner loop: 1) it has a single exit block at its header, and 2) this exit targets the TRANSITION CFG. Only under these conditions can the header serve as a “hub” that brings the two sub-CFGs into a single merged loop.

4.1 The General Case

We extend the basic transformation to support the general case in two steps. We first remove the constraint that the single exit block of the inner loop is its header, by considering the exit block location to be: 1) the tail of the loop’s back edge, and 2) neither the head nor the tail of the back edge. Then we further remove the single-exit constraint on both outer and inner loops.

Figure 8a shows a generic do–while inner loop where the exit block is at the tail of the back edge. In two steps, we can convert it to a while loop that has the exit in its header, as shown in Figure 8b. First, the last branch instruction in the loop’s exit block is peeled off and moved to a new block. Second, this branch block is made the header of the new loop, and a new pre-header block is created to initialize the condition variable so that the branch is always taken the first time. This transformation can be viewed as an inverse of loop inversion [9], which converts a while loop to a do–while loop.

If the exit block of the inner loop is at neither the head nor the tail of its back edge, as shown in Figure 8c, we can shift the exit to the tail of the back edge by applying a variant of the exit shifting transformation mentioned above, thus

---

4 The TRANSITION CFG includes the back edge of the outer loop.
Applying LM, a corresponding edge from $BB2'$ to $BB5'$ appears in the TRANSITION' CFG in Figure 7. Note that this results in a loop nested within the merged loop.

5. Evaluation

We automate LM in a compiler and evaluate its benefit on a synthetic benchmark and five application benchmarks. The parameterizable synthetic benchmark allows us to investigate how the performance impact of LM varies with the factors described in Section 3.2. We describe our compiler implementation, the experimental platform, the benchmarks and finally the performance results.

5.1 Compiler Implementation

We implement LM as a middle-end pass in LLVM [7]. The low-levelness of LLVM IR is a good match to LM, which operates directly on the CFG. We extend the Clang front-end with a compiler directive that allows users to enable LM for specific loop nests. It is specified right before the inner-most loop, and is of the following format:

```
#pragma merge_loop(N)
```

where $N$ specifies the number of loops enclosing the inner-most loop body that are to be merged. The use of directives to apply optimizations is not unusual and is typical in languages such as OpenMP [16] and OpenACC [15].

To incorporate LM in the compilation flow of a CUDA program, we utilize the open-source PTX back-end (NVPTX) NVIDIA recently released. It takes as input LLVM IR with NVIDIA-specific extensions (NVVM IR [13]) and outputs PTX code as a string. An input kernel, annotated with `merge_loop` directives, is first translated to NVVM IR (with `merge_loop` annotations) by the extended Clang front-end. The `merge_loop` annotations in the NVVM IR are handled by the LM pass, before the NVPTX back-end is invoked. The PTX string produced is embedded into an auto-generated kernel wrapper (in C++), which gets compiled along with the host program using a regular C++ compiler. We anticipate the release into public domain of our implementation.

5.2 Experimental Platform

All experiments are performed on a system that has an Intel Core i7-960 CPU with an NVIDIA GeForce GTX 480 GPU and 6GB of main memory. We use the CUDA 5.0 toolkit, running on Ubuntu 10.04. All benchmarks are compiled using the custom flow described above, with and without LM applied. For each benchmark, the performance impact of LM is measured by the speedup of the kernel LM targets, i.e., the ratio of execution time of the kernel without LM to that of the kernel with LM.

5.3 Synthetic Benchmark (SYN−LM)

The kernel code template of SYN−LM, shown in Figure 9, is similar to the generic pattern targeted by LM (Figure 5a). It uses five parameters, listed in Table 1 to model program characteristics that may affect the benefit of LM. OUTER_TC...
is the outer loop trip-count and is constant across warp threads. For each outer loop iteration, the inner loop trip-count is randomly generated. Its average and variance are specified by \texttt{INNER\_TC\_AVG} and \texttt{INNER\_TC\_VAR} (an integer \( \in [0, 10] \)) respectively. The generated trip-count is uniformly distributed between \texttt{INNER\_TC\_AVG} \times (1-\texttt{INNER\_TC\_VAR}/10) and \texttt{INNER\_TC\_AVG} \times (1+\texttt{INNER\_TC\_VAR}/10). With a distinct random number seed per thread, \texttt{INNER\_TC\_VAR} models the variance of inner loop trip-count 1) across warp threads for each outer loop iteration, and 2) across outer loop iterations for each thread.

```c
for (i = 0; i < OUTER\_TC; ++i) {
    n\_inner\_iters = get\_rand(\n        INNER\_TC\_AVG, INNER\_TC\_VAR);\n    for (j = 0; j < n\_inner\_iters; ++j) {
        // inner loop body (INNER\_SIZE instructions)\n    }\n    // epilogue (EP\_SIZE instructions)\n}
```

**Figure 9.** SYN–LM kernel template.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTER_TC</td>
<td>outer loop trip-count</td>
</tr>
<tr>
<td>INNER_TC_AVG</td>
<td>average of inner loop trip-count</td>
</tr>
<tr>
<td>INNER_TC_VAR</td>
<td>variance of inner loop trip-count</td>
</tr>
<tr>
<td>INNER_SIZE</td>
<td>inner loop body size</td>
</tr>
<tr>
<td>EP_SIZE</td>
<td>epilogue size</td>
</tr>
</tbody>
</table>

Both the inner loop body and the epilogue are a sequence of fused-multiply-add (FMA) operations on a single register variable. This variable is initialized to the thread ID at the beginning of the kernel and written out to the global memory at the very end for validation. \texttt{INNER\_SIZE} and \texttt{EP\_SIZE} specify the number of FMA operations in the two code segments respectively.

Compared to the code pattern in Figure 9a, SYN–LM uses the code for random number generation as a fixed-length prologue before the inner loop. It is unnecessary to introduce a parameter for controlling the prologue length, because the “transition” code size can be varied through \texttt{EP\_SIZE} alone.

SYN–LM is run with the highest level of parallelism, i.e., 48 warps/SM on GTX480.

## 5.4 Application Benchmarks

We use five application benchmarks to evaluate the performance impact of LM.

**MCX** [4], or Monte Carlo eXtreme, models the scattering and absorption of photons in a 3-D heterogeneous turbid media. A major source of divergence in the kernel is that each of the millions of photons, simulated by a single thread, requires an unknown and varying number of simulation steps, depending on how it transmits through and reflects from cell boundaries. This translates into an inner loop with a varying trip-count, enclosed by an outer loop that iterates through multiple photons. This loop nest is the target of LM, with a single \texttt{merge\_loop} directive.

**GPU-MCM [1],** or GPU-accelerated Monte Carlo simulation for Multi-Layered media, is similar to MCX, but for simulating photon transport in 2-D media. While suffering from the same divergence problem, it differs from MCX in that it is highly optimized through the use of shared memory. The target of LM is also an inner loop with a varying trip-count that is enclosed by an outer loop.

**MC-GPU** [2] is another Monte Carlo simulation for the transport of X-rays in a 3-D geometry, but uses an algorithm to model photon trajectories that results in several inner loops, each with a varying trip-count. We apply LM to only one inner loop that has the largest trip-count variance, and the greatest ratio of loop body size over the “transition” code size. Thus, we use only one \texttt{merge\_loop} directive.

**MUMmerGPU** [19] is a DNA exact sequence alignment program. Given a reference sequence and multiple query sequences, it finds all sub-strings of each query that maximally match a sub-string of the reference. It builds a suffix tree for the reference sequence on the host. Each query by a single GPU thread traverses this tree. This traversal, in the form of two nested loops with dynamic trip-counts, is the main source of divergence. We select the outer loop of the two to be the target of LM, based on the same criteria as that for MC-GPU. This loop is surrounded by an implicit loop that iterates over multiple queries processed by each thread. Again, we use only one \texttt{merge\_loop} directive.

**MO** [18], or Molecular Orbital, computes molecular orbital amplitudes on a uniformly spaced 3-D grid, originally developed as a part of the molecular visualization program VMD. While the kernel has a three-level loop nest, no inner loop exhibits trip-count variance across warp threads. Without loop-induced divergence, this benchmark is used to assess the penalty of LM in the absence of potential benefit. We use a single \texttt{merge\_loop} directive to merge the outer two loops of the nest.

It is important to note that the standard loop coalescing cannot be applied to any of the five benchmarks, due to the presence of while loops and non-perfect nests, necessitating the generalized form of LM.

MCX and MO include multiple kernel variants. They differ in the enablement of math function acceleration and some memory optimizations (e.g., the use of constant memory). We apply LM to four variants of MCX and two variants of MO.

Table 2 compares characteristics that affect the benefit of LM for the target loop-nest in each of the five benchmarks. These characteristics roughly correspond to the parameters of SYN–LM. Since the parameter values depend on the kernel variant and program inputs, the table shows the numbers for one run that achieves an average LM-induced speedup.
Although MCX, GPU-MCML and MC-GPU are all Monte Carlo applications, they differ not only in the parameters listed in the table, but also in instruction mixture in the inner loop body and prologue/epilogue, and pattern of variation of inner loop trip-count.

Since the benefit of LM also depends on the extent of degradation of existing memory coalescing in the original application, we show in Table 2 the global memory load efficiency of each benchmark: a metric provided by the NVIDIA profiler `nvprof` that varies directly with the degree of memory coalescing. MCX and MC-GPU exhibit a very low degree of coalescing because most of the uncoalesced accesses have been eliminated through buffering in shared memory. This optimization also drastically reduces the number of global memory accesses made in the loop nest LM targets. MUMmerGPU exhibits a very low degree of coalescing because different queries traverse different paths of the suffix tree. The version of MO where input data reside in global memory achieves perfect coalescing.

| Table 2. Loop characteristics of application benchmarks. |
|------------------|------------------|------------------|------------------|------------------|------------------|
| Inner loop body size | MCX | GPU-MCML | MC-GPU | MUMmerGPU | MO |
| `transition` code size | 800 | 300 | 2700 | 120 | 160 |
| Avg. outer loop trip-count | 558 | 74 | 100 | 400 | 60 |
| Avg. inner loop trip-count | 613 | 22 | 2 | 6 |
| Std.dev. of inner loop trip-count | 269 | 975 | 11 | 1.2 | 0 |
| Global memory load efficiency | 0.8% | 93% | 3.8% | 2.3% | 100% |

It is also worth noting that the latest versions of MCX and GPU-MCML include a manually applied optimization that is equivalent to LM. We replace this optimization with a merge loop directive in the evaluation. The performance differences between the compiler-generated and the manually optimized versions of MCX and GPU-MCML are negligible (1.4% and 1.0% respectively on average).

5.5 SYN-LM Experiments

We use three experiments to explore how the benefit of LM varies with INNER_TC_VAR, INNER_TC_AVG, the ratio of INNER_SIZE to EP_SIZE, and OUTER_TC. Overall, they demonstrate that the LM-induced speedup can range from 0.4× to 1.6×.

5.5.1 Impact of INNER_TC_VAR

In the first experiment, we vary INNER_TC_VAR and OUTER_TC while fixing the other parameters. Figure 10 shows how the speedup varies with INNER_TC_VAR (from 0 to 10) for different values of OUTER_TC.

We make three observations from the figure. First, when INNER_TC_VAR is non-zero, the LM-induced speedup increases linearly with INNER_TC_VAR. This can be explained by examining the respective performance penalty due to increasing INNER_TC_VAR on the original and optimized kernels. For the original kernel, an increase in INNER_TC_VAR linearly increases the performance penalty. This is because, in each outer loop iteration, all warp threads must wait for the one that executes the maximum number of inner loop iterations, which increases linearly with INNER_TC_VAR. For the optimized kernel, however, increasing INNER_TC_VAR should not incur any performance penalty, at least when OUTER_TC is very large, because the variation of inner loop trip-count tends to cancel out when loops are merged. The observed speedup increase is a result of the increasing gap between the two penalties.

Our second observation is that the rate of speedup increase varies directly with OUTER_TC. This is because, as OUTER_TC becomes smaller, the variation of inner loop trip-count does induce variation in the number of merged iterations across thread and thus performance penalty on the optimized kernel, resulting in lower speedup improvement. However, when OUTER_TC increases, these variations tend to cancel out (the Law of Large Numbers) and the speedup improves.

Third, LM does not always improve kernel performance. When the inner loop trip-count does not vary across warp threads (i.e., INNER_TC_VAR = 0), LM introduces 9% performance penalty, which comes from two sources. The first source is an overhead of five instructions per inner loop iteration. Three instructions come from the extra branch basic block (BB6 in Figure 7). The rest is due to the fact that Loop Invariant Code Motion (LICM) cannot be applied to the merged loop as aggressively as to the original inner loop. The instruction overhead leads to a 5% increase in dynamic instruction count (INNER_SIZE = 100), as experimentally measured by profiling. The remaining 4% performance penalty must be attributed to the drop of Instructions Per Cycle (IPC). It occurs because the extra branch basic block LM introduces to the merged loop body has only 3 instructions; the shortage of instructions to fill the branch...
delay slots causes the ALU pipeline to stall when executing this block.

As INNER_TC_VAR increases from 0 to 1, the performance penalty of LM actually increases by an additional 5%. This is because, at no variance, the execution of the “transition” code is convergent, so only the above overhead impacts performance. However, when INNER_TC_VAR is 1, there is added overhead due to divergent execution of the “transition” code. With such low variance, there is little benefit to LM, resulting in an overall degraded performance.

### 5.5.2 Impact of INNER_TC_AVG

Figure 11 shows the speedup as a function of INNER_TC_AVG, for different values of INNER_TC_VAR.

![Figure 11. Speedup vs. INNER_TC_AVG and INNER_TC_VAR.](image)

We make two observations from the figure. First, the speedup improves with increasing INNER_TC_AVG (with a diminishing return) when INNER_TC_VAR is non-zero. This is because an increase in the average inner loop trip-count increases the ratio of “regular” (merged) iteration count over “transition” iteration count, downplaying the potential divergence penalty when executing the latter. The same reasoning explains why the speedup does not vary with INNER_TC_AVG when INNER_TC_VAR is zero: the execution of “transition” code is convergent.

Second, as INNER_TC_VAR increases from 1 to 10, the speedup curve shifts up with little change in shape. This suggests that the impact of INNER_TC_VAR and INNER_TC_AVG on the speedup are orthogonal.

### 5.5.3 Impact of INNER_SIZE

Figure 12 shows the speedup as a function of INNER_SIZE, for different values of INNER_TC_VAR.

![Figure 12. Speedup vs. INNER_SIZE and INNER_TC_VAR.](image)

due to loop overhead. Increasing INNER_TC_AVG proportionally amplifies this overhead while increasing INNER_SIZE does not. This is also the reason increasing INNER_SIZE improves speedup but increasing INNER_TC_AVG does not, when INNER_TC_VAR is zero.

### 5.6 Application Benchmark Experiments

Figure 13 shows the speedups of application benchmarks. Since the benefit of LM varies with both program inputs and kernel variants, we report the average speedup (geometric mean) along with maximum and minimum speedups in the form of error bars.

![Figure 13. Speedup of application benchmarks.](image)

LM brings speedup to most benchmarks, particularly MCX and GPU-MCML. These two applications exhibit a very high ratio of INNER_SIZE to EP_SIZE and a very high INNER_TC_VAR (that exceeds those experimented on SYN–LM), thus increasing the benefit of LM. The benefit of LM for MC-GPU (1.0× to 1.4×) is less significant for two reasons. First, although its inner loop body is very long in absolute term, the ratio of INNER_SIZE to EP_SIZE is only moderate. Second, the average inner loop trip-count is low (around 20). This is consistent with the results for SYN–LM in Figure 11.

While MUMmerGPU does not benefit much from LM on average, the speedup can be up to 1.24×. The variation of inner loop trip-count is particularly low for this benchmark, with most trip-counts ranging from 1 to 4. This is because most sub-strings of a query would not match any sub-strings...
of the reference, thus exhibiting very short traversals in the suffix tree.

While MO does not exhibit loop-induced divergence, LM degrades its performance by only 2%.

To estimate the impact of LM on the memory performance of each benchmark, we profile the total number of memory transactions (through `nvprof`) before and after applying LM. We observe that this metric increases by no more than 6.7% for all benchmarks. This is expected because 1) MCX, MC-GPU and MUMmerGPU exhibit a very low degree of memory coalescing to begin with, 2) GPU-MCML has a low percentage of global memory accesses in the loop nest LM targets, and 3) the absence of loop-induced divergence in MO preserves its memory access pattern after applying LM.

Overall, LM can bring a huge performance improvement (up to $4.37 \times$) to some benchmarks. However, its benefit is sensitive to both kernel variants (of MCX) and program inputs (of GPU-MCML, MC-GPU and MUMmerGPU). This behavior demands a strategy of deciding when to apply LM based on compile-time and run-time heuristics, which we leave for future work.

6. Related Work

Loop coalescing [17] is a standard compiler transformation similar to loop merging. It is used to convert perfectly nested parallel DO loops into a single DO loop. It is later extended to handle nested triangular loops, where inner loop bounds are functions of outer loop indices [14]. While originally proposed to simplify loop scheduling and reduce synchronization overhead on parallel machines, it has recently been applied to improve memory coalescing and reduce loop-induced divergence on GPUs [8]. However, the optimization is only considered in the context of irregular sparse matrix applications. Loop merging generalizes loop coalescing in three aspects. First, it does not require the loops to have an explicit notion of index variable. As a result, the loop bounds need not be known before entering the loop. Second, the loops do not have to be perfectly nested. As shown in the previous section, the intervening code between nested loops plays an important role in the benefit of loop merging, which is not considered in loop coalescing. Last, the code transformation is not restricted to be applied at the source level; indeed, it is more desirable to apply it to the CFG. We observe that loop coalescing cannot be applied in any of our application benchmarks due to the presence of while loops and non-perfect nests. This demonstrates the need for our generalization of loop coalescing.

Both software and hardware techniques have been proposed to reduce branch divergence in GPU applications. Han and Abdelrahman [6] propose two software-based optimizations, iteration delaying and branch distribution, that target divergent branches. In contrast, loop merging targets divergence due to varying loop trip-counts. Zhang et al. [22] propose to reorder loop iterations (in software) so that those with more convergent control flow are assigned to the same warp. This work cannot reduce divergence of a loop nest whose inner loop trip-count is not known before executing the loop.

Thread Frontier [3] is a new reconvergence mechanism in hardware that allows earlier reconvergence in the presence of unstructured control flow when compared to the PDOM model. Compared to loop merging, it does not target loop-induced divergence while requiring hardware extension to the NVIDIA GPU. Dynamic Warp Formation (DWF) [5] is a hardware mechanism that reduces branch divergence by re-grouping threads that take the same path into new warps during kernel execution. While it can reduce loop-induced divergence, it requires extension to GPU hardware.

7. Conclusion and Future Work

In this paper we present a software-based optimization for reducing loop-induced divergence: loop merging. Our evaluation of its benefit with a prototype compiler implementation shows that it can improve the performance of both synthetic and application benchmarks by up to $1.6 \times$ and $4.3 \times$ respectively. In each benchmark, the benefit comes from the use of a single merge loop directive, which is not onerous.

Further, while its benefit is sensitive to loop characteristics (e.g., inner loop body size and epilogue size) and program inputs (that affect the average and variance of inner loop trip-counts), LM results in little or no degradation of performance in application benchmarks, indicating its viability. While our evaluation is performed on an NVIDIA GPU, LM can be extended, with little effort, to other SIMD architectures (e.g., those from AMD and Intel).

There are two main directions for future work. The first is the automation of the selection of loop nests targeted by loop merging, through compile-time and run-time heuristics. Second, loop merging can be viewed as a software transformation that statically shifts the reconvergence points when warp threads execute the loop nest. We would like to assess the potential of more generalized mechanisms that shift reconvergence points at run-time.

References


---

3 More precisely, this work applies loop collapsing, a special form of loop coalescing.


