Reducing Branch Divergence in GPU Programs

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 has a significant impact on the performance of GPU programs. We propose two novel software-based optimizations, called *iteration delaying* and *branch distribution* that aim to reduce branch divergence. Iteration delaying targets a divergent branch enclosed by a loop within a kernel. It improves performance by executing loop iterations that take the same branch direction and delaying those that take the other direction until later iterations. Branch distribution reduces the length of divergent code by factoring out structurally similar code from the branch paths. We conduct a preliminary evaluation of the two optimizations using both synthetic benchmarks and a highly-optimized real-world application. Our evaluation shows that they improve the performance of the synthetic benchmarks by as much as 30% and 80% respectively, and that of the real-world application by 12% and 16% respectively.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

Algorithms, Performance, Experimentation, Measurement

Keywords

Branch divergence, GPGPU, Data parallel programming

1. INTRODUCTION

General-Purpose Graphics Processing Units (GPGPUs) have become increasingly popular in High-Performance Computing (HPC). However, programmers must carefully tune their applications for the GPU architecture in order to best utilize their massive computing power. Not surprisingly, there has been a large number of optimizations, both manual and automatic, that aim to improve the performance of GPU programs. The majority of these optimizations target the GPU memory hierarchy, utilizing the on-chip software-managed cache and adjusting the pattern of accesses to the off-chip device memory [1, 5–7, 14–18].

In contrast, there has been less work on optimizations that target another fundamental aspect of GPU performance, namely its Single Instruction Multiple Data (SIMD) execution model. While this model enables simpler control hardware, it imposes heavy performance penalties on kernels with control flow. In such kernels, SIMD threads *diverge*, i.e., follow different paths of execution. The hardware makes all these paths execute sequentially, even though each thread executes only one of the paths. We envision this *branch divergence* issue to become more important as the GPGPU community continues to push the boundary of "GPU-friendly" applications. Present techniques for handling branch divergence either demand hardware support or require host-GPU interaction, which incurs overhead.

We present two novel software-based optimizations for reducing branch divergence in GPU programs: *iteration delaying* and *branch distribution*. Iteration delaying improves the utilization of execution units in the presence of a divergent branch within a loop, by executing only one branch path in each iteration and delaying the threads that follow the other path until later iterations. Branch distribution aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths.

We conduct a preliminary evaluation of the benefit of our proposed optimizations, before embarking on their implementation in a compiler. We use two synthetic benchmarks (one for each optimization) and one highly-optimized real-world application called Monte Carlo simulation for Multi-Layered media (MCML) [9]. We parameterize the synthetic benchmarks to explore the impact of various kernel characteristics on the benefit of these optimizations. Results on a Fermi GPU show that iteration delaying and branch distribution improve the performance of the synthetic benchmarks by up to 30% and 80%, respectively. Further, the optimizations improve the overall performance of the highly-optimized MCML by 12% and 16%, respectively.

The remainder of the paper is organized as follows. Section 2 provides background on NVIDIA Fermi GPUs and on branch divergence. Section 3 describes the two optimizations we propose. Section 4 presents evaluation methodology and results. Section 5 discusses related work, and Section 6 gives directions for future work.

2. BACKGROUND

This section briefly describes the CUDA programming model and the architecture of NVIDIA GPUs [13]. In particular, we describe the SIMD execution model and how divergent branches are executed.

GPGPU-4 Mar 05-05 2011, Newport Beach, CA USA

2.1 CUDA

The Compute Unified Device Architecture (CUDA) provides a C-extended programming model that allows the programmer to writes the host (CPU) code and the device (GPU) code in a single source program [12]. The device code, or *a kernel*, is similar to a C function, except that it is executed many times, once by each GPU thread. Launching a kernel for GPU execution involves calling the kernel function in the host code, along with a specification of the space of GPU threads that execute it, called a *grid*. A grid contains multiple *thread blocks*, organized in a two-dimensional space. Each thread block contains multiple threads, organized in a three-dimensional space. To allow different threads to access different data and follow different control-flow paths, each thread is given a unique identifier, accessible within the kernel function through the built-in vector variables `blockIdx` and `threadIdx`. Threads within a thread block can barrier-synchronize using the `__syncthreads` primitive. Synchronization across different thread blocks is generally *not* supported. A kernel can access multiple GPU memories during execution, including 1) off-chip *global*, *constant* and *texture memory* that are shared by all GPU threads and the host, and 2) on-chip *shared memory* that is shared among threads within a thread block. While the read-only constant and texture memory are cached on-chip by hardware, the shared memory is a software-managed cache for the global memory.

2.2 GPU architecture

An NVIDIA GPU consists of a number of *streaming multiprocessors* (SMs). Each SM contains a number of *CUDA cores*, which receive instructions from a single issue unit[1], and execute them in a SIMD fashion. Each core is essentially a fused multiply-add (FMA) floating-point arithmetic unit. Each SM also has a number of special functional units (SFUs) for executing transcendental functions.

The GPU executes a kernel by scheduling thread blocks onto the SMs. Once a thread block is assigned to a SM, it must be executed in its entirety by the SM. Each active thread block is split into groups of 32 threads called *warps*, each of which is executed on the SM in a SIMD fashion. This means that all threads within a warp must execute the same instruction at any given time. In the presence of a data-dependent branch that causes different threads in the same warp to follow different paths (also known as ***branch divergence***), the warp serially executes each branch path taken, disabling threads that are not on that path. The threads reconverge after all divergent paths are completed [8]. To allow such execution, each CUDA core supports *predication*, by conditionally executing an instruction based on a per-thread *predicate*. Further, the hardware exposes predicated instructions to the ISA, and the CUDA compiler can transform a small branch directly into predicated instructions to avoid the overhead of handling divergence and reconvergence in hardware. However, this transformation is disabled for large branches because predicated instructions are *always* executed, even when the branch is not divergent. It is important to note that the above discussion about divergence only applies to the threads *within* a warp; different warps can be scheduled independently.

[1]In the case of Fermi, 32 CUDA cores in a SM are divided into two groups, each having a dedicated instruction issue unit.

2.3 Impact of branch divergence

Branch divergence can hurt performance due to lower utilization of the execution units, which cannot be compensated for through increased levels of parallelism. To illustrate its impact, Figure 1 shows three common scenarios of kernel code that exhibit such divergence.

```
tid = threadIdx.x;
if (a[tid] > 0) {
  ++x;
}
```

(a) if statement.

```
tid = threadIdx.x;
if (a[tid] > 0) {
  ++x;
} else {
  --x;
}
```

(b) if-then-else statement.

```
n = a[threadIdx.x];
for (i = 0; i < n; ++i) {
    // work
}
```

(c) Loop with variable trip-count

Figure 1: Common scenarios of branch divergence.

In the first scenario (Figure 1a), if any thread executes `++x`, *all* threads in the same warp must go through `++x`, regardless of whether they actually execute it. In the average case, where half of the warp threads evaluate the branch condition to true, the utilization of the execution units is only 50%. The second scenario (Figure 1b) can be viewed as a sequential composition of two `if` statements of the first scenario: each thread in a warp must go through *both* branch paths sequentially, even though it just executes one of them. Assuming both branch paths are of equal size, this leads to only 50% utilization of the execution units as well. In the last scenario (Figure 1c), the number of iterations of loop `i` each thread goes through is the max iteration count for all threads within the warp. The performance impact depends on the size of the loop body and the variance of the loop trip counts, i.e., the `n`'s.

3. THE OPTIMIZATIONS

We present two optimizations that aim to reduce the performance penalty caused by branch divergence. We refer to them as *iteration delaying* and *branch distribution*.

3.1 Iteration delaying

Consider a kernel where each GPU thread in a warp executes a loop that contains a potentially divergent `if-then-else` branch similar to that in Figure 1b. The branch condition is often data dependent and the branch direction cannot be determined at compile-time. Iteration delaying is a run-time optimization technique that targets this scenario. The main idea is that, in each loop iteration, instead of all warp threads going through both paths of the branch, they all take one of the paths. Those that should take the other path simply do nothing, delaying their computations to a subsequent iteration, where potentially more (or even all) threads are taking their path, resulting in higher utilization of execution units.

To illustrate the benefit of iteration delaying, consider three threads in a warp, each executing three iterations of a loop that contains a divergent branch. Assume that each of the two paths of the branch has 100 FMA instructions. Figure 2a and 2b show the execution of these threads before

and after iteration delaying respectively. Each loop iteration is represented as a tuple, where the first component is the iteration number and the second component is the branch direction (T for taken and N for not-taken). Comparing the two executions, we see that iteration delaying allows iteration 2 and 4 to be executed concurrently (same for iteration 7, 5 and 6), which saves 200 FMA instructions. In general, iteration delaying results in more loop iterations, each having less dynamic instructions.

Thread 1	Thread 2	Thread 3	Instr. Count
(1,T)	-	(3,T)	100
-	(2,N)	-	100
-	(5,T)	(6,T)	100
(4,N)	-	-	100
(7,T)	-	-	100
-	(8,N)	(9,N)	100
Total			600

(a) Original execution.

Thread 1	Thread 2	Thread 3	Chosen Br. Dir.	Instr. Count
(1,T)	-	(3,T)	T	100
(4,N)	(2,N)	-	N	100
(7,T)	(5,T)	(6,T)	T	100
-	(8,N)	(9,N)	N	100
Total				400

(b) Iteration delaying with majority-vote strategy.

Thread 1	Thread 2	Thread 3	Chosen Br. Dir.	Instr. Count
-	(2,N)	-	N	100
(1,T)	(5,T)	(3,T)	T	100
(4,N)	(8,N)	-	N	100
(7,T)	-	(6,T)	T	100
-	-	(9,N)	N	100
Total				500

(c) Iteration delaying with minority-vote in the first iteration.

Figure 2: Example of the benefit of iteration delaying.

A critical aspect of iteration delaying is the decision on which branch path to take in each iteration. In the example above, if the threads take the not-taken (N) path in the first iteration, the execution is at best something like that shown in Figure 2c and takes 100 more instructions.

We propose two strategies to make this decision. The first is *majority-vote*. In each iteration, all threads in a warp communicate with each other to determine the number of threads that take each path, and then choose the direction that at least half of the threads (i.e., 16) take. The rationale behind this strategy is to utilize at least half of the execution units. However, the majority-vote strategy does have a drawback: it may starve threads that follow a "cold" path, i.e., one taken less frequently. In order to allow these threads to execute, iteration delaying can be disabled altogether at some point, e.g., when at least one thread has finished all iterations, and all threads continue as normal after that. Although correctness is preserved this way, performance may still suffer because there are very few threads (that lag behind) executing toward the end of the loop. Figure 3 shows such a case, where the not-taken branch (N) is only taken

Thread 1	Thread 2	Thread 3	Instr. Count
(1,T)	(2,N)	(3,T)	100 + 100
(4,T)	(5,T)	(6,T)	100
(7,T)	(8,T)	(9,T)	100
Total			400

Figure 3: An extreme case where iteration delaying with majority-vote results in more dynamic FMA instructions.

by Thread 2 in its first iteration. The majority-vote strategy delays Thread 2 until both Thread 1 and Thread 3 finish, resulting in 600 FMA instructions, and thus, lost benefit.

The second decision strategy is *round-robin*, in which the branching decision for the i'th iteration is the opposite of that for the $(i-1)$'th iteration. Since the decision does not depend on the actual path(s) the threads want to take in an iteration, it can result in an idle iteration if no thread actually takes this direction. Thus, the direction is reverted in this special case. Although this strategy may not be as aggressive in saving instructions as majority-vote at the beginning, it does not starve threads and therefore can be applied for the entire loop with higher utilization rate toward the end of the loop. In the example above (Figure 2a), both strategies give identical execution.

3.1.1 Implementation

The main challenge of implementing iteration delaying is reaching a consensus among the warp threads on which path of the branch to take in each loop iteration. The new instructions provided by Fermi GPUs allow the two decision strategies described above to be implemented very efficiently. Consider a general kernel template and the code after applying iteration delaying with majority-vote, shown in Figure 4 and Figure 5a respectively. The `__ballot` instruction collects branch conditions for the 32 warp threads into a 32-bit integer. The `__popc` instruction counts the number of bit 1's in a 32-bit integer. The variable `thresh` in line 10 is a threshold that determines what "majority" is, and it is a parameter of the majority-vote strategy. Due to space limitation, the code that prevents starvation is omitted.

```
1 for (int i = 0; i < N_ITERATIONS; ++i) {
2   int cond = ... // compute branch cond.
3   if (cond) {
4     // code segment 1
5   } else {
6     // code segment 2
7   }
8   // non-branch code
9 }
```

Figure 4: Code pattern that iteration delaying targets.

The round-robin strategy is even simpler to implement, as `cond_for_all` just needs to be inverted. To make the strategy more flexible, we invert `cond_for_all` periodically, as shown in Figure 5b. After `num_zeros` iterations that take the `else` path, the variable is inverted for `num_ones` iterations that take the `if` path. The ratio of `num_zeros` to the sum of `num_zeros` and `num_ones` (i.e., `period`) defines the *duty cycle*– or simply the *cycle*–of the round-robin strategy. The `__any` instruction returns true iff at least one thread's condition is true. The `__all` instruction returns true iff all threads' conditions are true. Both warp-vote instructions have been supported since the second-generation

GPUs. The conditional selection between `__any` and `__all` is always convergent, so only one warp-vote instruction is executed per thread. It is worthwhile to note that idle iteration removal may not improve performance, because it saves the loop-housekeeping instructions from idle iterations but adds instruction overhead to *all* iterations.

```
1  int not_delayed = 1;
2  int cond;
3  for (int i = 0; i < N_ITERATIONS; ) {
4    if (not_delayed) {
5      cond = ...  // compute branch cond.
6    }
7
8    // Make a convergent branch decision.
9    int cond_for_all =
10     (__popc(__ballot(cond)) >= thresh);
11   // Should I do work in this iteration?
12   not_delayed = (cond_for_all == cond);
13
14   if (not_delayed) {
15     if (cond_for_all) {
16       // code segment 1
17     } else {
18       // code segment 2
19     }
20     // non-branch code
21
22     ++i;
23   }
24 }
```

(a) Majority-vote strategy.

```
// branch direction period: 0,0,0,1
// 1 means the IF branch path
// 0 means the ELSE branch path
int num_zeros = 3, num_ones = 1;
int period = num_zeros + num_ones;
int counter = -1;
...
// update the direction
if (++counter == period) counter = 0;
cond_for_all = (counter >= num_zeros);
// remove idle iteration
cond_for_all = cond_for_all ?
  __any(cond) : __all(cond);
```

(b) Round-robin strategy.

Figure 5: Code after iteration delaying.

3.1.2 Discussion

In general, iteration delaying targets a top-level branch within a kernel loop that does not have barriers. The branch is not limited to the `if-then-else` we described; it can be a `switch` or an `if` statement. Further, there is no additional restrictions on the parent loop except the barrier-free requirement. In particular, the loop does not have to be parallelizable since iteration delaying preserves the order of iterations in each thread.

There are four aspects of a kernel that affect the benefit of iteration delaying. The first is the size of the branch code (in both paths) relative to the fixed instruction overhead introduced by iteration delaying. The second aspect is the size of the branch code relative to the other code in the loop body (e.g., line 8 in Figure 4). Delaying the execution by one iteration foregoes the entire loop body, reducing the number of threads executing the non-branch part of the loop. Therefore, as the relative size of the branch code decreases, iteration delaying is expected to be less beneficial. The third aspect is the branching pattern, i.e., how often one path is taken over the other, and how often the branch direction changes per thread. The parameters and effectiveness of the strategy to decide the branch direction in each iteration highly depends on this aspect. For example, the threshold for majority-vote does not have to be half of the warp size; it may be made to depend on the "hotness" of the paths. Further, iteration delaying favors fast-changing branch directions which requires fewer number of iteration delays before reaching a convergent iteration. In fact, one of the best branching patterns for iteration delaying is that the branch direction (per thread) alternates over iterations. The last aspect that determines the benefit of iteration delaying is the memory access behavior of the instructions in the branch paths. Iteration delaying may destroy coalesced memory accesses in the original code. However, it is unlikely to achieve memory coalescing in divergent code anyways.

3.2 Branch distribution

Iteration delaying relies on a per-thread loop that surrounds the target branch. To reduce the divergence of a multi-path branch on its own, we propose *branch distribution*, which "factors out" code from the branch paths that are structurally the same, so that the total number of dynamic instructions is reduced. Branch distribution is similar to code hoisting [11], but is more aggressive.

Consider the code fragment shown in Figure 6a. Code hoisting leaves it untransformed since there is no common sub-expression in both branch paths. However, the structures of the two branches are almost identical, and we can produce the less divergent code shown in Figure 6b. Thus, this optimization "distributes" the branch condition evaluation over the two branch bodies, which results in one or more smaller branch blocks interleaved with blocks of straight-line code, reducing the impact of divergence.

```
if (c > 0) {
  x = x * a1 + b1;
  y = y * a1 + b1;
} else {
  x = x * a2 + b2;
  y = y * a2 + b2;
}
```

(a) Original code.

```
if (c > 0) {
  a = a1;
  b = b1;
} else {
  a = a2;
  b = b2;
}
x = x * a + b;
y = y * a + b;
```

(b) Optimized code.

Figure 6: An example that illustrates branch distribution.

However, branch distribution does not always improve performance, for three reasons. The first is that, it introduces instruction overhead, including extra branch instructions and those in the prologue of the example above that produce `a` and b. Therefore, branch distribution is only beneficial if the code factored out is large enough that the resulting benefit of convergence compensates for the overhead. In fact, the benefit of this optimization increases with the size of the code factored out relative to that of the code left to be divergent. The second reason is that, branch distribution may increase register usage, particularly when the code factored-out uses many inputs and produces many outputs, and may therefore reduce the level of parallelism. Consider the following extreme example: the kernel (for each thread) computes $\sum_{i=0}^{10} (x+i)^2$ in one branch path

and $\sum_{i=0}^{10}(x-i)^2$ in the other, both in loop-unrolled forms. Branch distribution would introduce a prologue that computes 10 intermediate variables $y_i (i = 0..9)$ that are equal to $x+i$ or $x-i$ based on the branch condition. These variables are then squared and summed in a common piece of code. Clearly, this code requires at least 10 registers to hold the y_i's, but the register usage for the original code can be as few as 3. The third and final reason is that, branch distribution may reduce Instruction-Level Parallelism (ILP) in each thread by breaking large basic blocks into smaller ones.

4. EXPERIMENTAL EVALUATION

We conduct a preliminary evaluation of the two optimizations we propose. Our goals are two-folds. First, we would like to determine the extent of the performance benefit the optimizations can bring. Second, we would like to assess their impact on a real-world application (MCML). In this section, we first describe the benchmarks and the hardware platform we use and then discuss the results.

4.1 Benchmarks

4.1.1 SYN-ITDELAY

SYN-ITDELAY is a synthetic benchmark for evaluating iteration delaying. It has the same code structure as shown in Figure 4. The benchmark models an unpredictable branching pattern by randomly generating the branch condition based on the algorithm used in MCML. This random number generator compiles into 19 PTX instructions using CUDA 3.0 targeting GTX480. Each thread uses a unique seed.

Each of the two branch paths consists of N pairs of FMA instructions, where N is a parameter. All FMA instructions involve only one floating-point variable, and are dependent back-to-back. Each instruction pair has the following form:

```
val = val * MULT1 + ADD1;
val = val * MULT2 + ADD2;
```

where `MULT1`, `ADD1`, `MULT2` and `ADD2` are constants such that the two instructions as a whole do not change `val`. We use this code for two reasons. First, it allows us to vary the size of the branch paths without introducing extra variables. Second, it allows us to correlate performance result with instruction count accurately. This is because 1) it prevents the NVCC compiler (with the `-O3` flag) from doing any optimization in the branch paths, and 2) it ensures that the execution never produces any denormalized numbers, which may affect instruction latency depending on how the ALU hardware is optimized for these corner cases.

This benchmark is run with enough warps (32 on GTX 480) so that the ALU pipelines are always filled. This filters out any performance impact due to changes in ILP.

4.1.2 SYN-BRDIS

SYN-BRDIS is a synthetic benchmark for evaluating branch distribution. The code before and after the optimization is shown in Figure 7. `code_seg_1` and `code_seg_2` form the prologue of the loop that remains divergent, while `code_seg_3` and `code_seg_4` form the divergent epilogue. The branch condition is constructed in a way that guarantees divergence in every iteration. `code_factored_out` represents the code factored out by branch distribution. Each of `code_seg_1` and `code_seg_2` is a sequence of 20 FMA instructions that are dependent back-to-back. They start with one input `val` and produce M outputs (`vin`'s), where M is a parameter in the range $[1, 20]$. This is achieved by having $20-M$ instructions that self-update `val` like SYN-ITDELAY and M instructions that produce `vin`'s in order, such that `vin_`$(i+1)$ = `vin_`i * `vin_`i + `vin_`i. Similarly, each of `code_seg_3` and `code_seg_4` is also a sequence of 20 FMA instructions that are dependent back-to-back. They take M inputs (`vout`'s) and produce a single output `val`. `code_factored_out` takes M inputs and produces M outputs by simply self-updating each `vin_`i $2N$ times like SYN-ITDELAY and storing the result in `vout_`i.

```
1  for (int i = 0; i < N_ITERATIONS; ++i) {
2    val = ... // random number generation
3    if ((threadIdx.x + i) & 1) {
4      code_segment_1(val,
5          vin_1, vin_2, ..., vin_M);
6      code_factored_out(
7          vin_1, vin_2, ..., vin_M,
8          vout_1, vout_2, ..., vout_M);
9      code_segment_3(val,
10         vout_1, vout_2, ..., vout_M);
11   } else {
12     code_segment_2(val,
13         vin_1, vin_2, ..., vin_M);
14     code_factored_out(
15         vin_1, vin_2, ..., vin_M,
16         vout_1, vout_2, ..., vout_M);
17     code_segment_4(val,
18         vout_1, vout_2, ..., vout_M);
19   }
20 }
21 result[threadIdx.x] = val;
```

(a) Original code.

```
1  for (int i = 0; i < N_ITERATIONS; ++i) {
2    val = ... // random number generation
3    if ((threadIdx.x + i) & 1)
4      code_segment_1(val,
5          vin_1, vin_2, ..., vin_M);
6    else
7      code_segment_2(val,
8          vin_1, vin_2, ..., vin_M);
9
10   code_factored_out(
11       vin_1, vin_2, ..., vin_M,
12       vout_1, vout_2, ..., vout_M);
13
14   if ((threadIdx.x + i) & 1)
15     code_segment_3(val,
16         vout_1, vout_2, ..., vout_M);
17   else
18     code_segment_4(val,
19         vout_1, vout_2, ..., vout_M);
20 }
21 result[threadIdx.x] = val;
```

(b) Optimized code.

Figure 7: Synthetic benchmark for branch distribution.

The two parameters, M and N, allow us to control the number of inputs/outputs of the code factored out (M), and the size of the code factored out relative to that of the divergent code (one path), i.e., $2*N*M/2*20 = N*M/20$.

4.1.3 MCML

The Monte Carlo simulation for Multi-Layered media (MCML) is a real-world medical application that models the scattering and absorption of photons in the tissue. It has been accelerated on a GTX280 [9]. We use a highly-

optimized version of this code as our base to evaluate the two optimizations we propose. This highly-optimized version includes aggressive memory optimizations that bring the GPU code speedup to two orders of magnitude over the (unoptimized) CPU version.

MCML has one kernel, shown Figure 8, where each thread is assigned a number of photons to simulate. Iteration delaying is applied to the main branch at line 4. Its `if` and `else` paths have around 200 and 170 PTX instructions respectively. In contrast, branch distribution targets a branch within the `if` path (line 5 and 6), which is shown in Figure 9. The code is split into two cases based on the moving direction of the photon in the z-direction. The two paths, each having about 80 PTX instructions, are almost identical except line 5 vs. 22, line 7 vs. 23, line 11 vs. 27, and line 12 vs. 28. Branch distribution merges the two paths into one, and leave the following code divergent: 1) a new layer definition that is `photon_layer+1` or `photon_layer-1` depending on the branch direction, and 2) a new variable passed as the last argument to `RFresnel` that is `photon_uz` or `-photon_uz`. Lines 11 and 27 become convergent once the new layer variable is introduced. The conditions at lines 12 and 28 are combined using disjunction.

```
1  Initialize the photon
2  while (true) {
3    Move the photon in its current direction
4    if (it hits a tissue layer boundary) {
5      Decide if photon transmits through
6          or reflects
7      Update the photon direction
8    } else {
9      Drop photon weight to
10         the current tissue location
11   }
12   if (photon weight is close to zero) {
13     // this photon is considered dead
14     if (more photons to process)
15       Init the next photon to be processed
16     else
17       exit
18   }
19 }
```

Figure 8: Pseudo-code for the MCML kernel.

```
1  // photon_uz: photon's direction in Z-axis
2  if (photon_uz > 0) {
3    // IOR: index of refraction
4    ni = IOR of current layer (photon_layer)
5    nt = IOR of next layer (photon_layer+1)
6    // Fresnel computation
7    r = RFresnel(ni, nt, photon_uz)
8
9    if (rand() > r) {
10     // transmit
11     ++photon_layer;
12     if (photon_layer > max_layer)
13       Kill the photon
14     else
15       Update direction (photon_[ux,uy,uz])
16   } else {
17     // reflect
18     photon_uz = -photon_uz;
19   }
20 } else {
21   ni = IOR of current layer (photon_layer)
22   nt = IOR of prev. layer (photon_layer-1)
23   r = RFresnel(ni, nt, -photon_uz)
24
25   if (rand() > r) {
26     // transmit
27     --photon_layer;
28     if (photon_layer < 0)
29       Kill the photon
30     else
31       Update direction (photon_[ux,uy,uz])
32   } else {
33     // reflect
34     photon_uz = -photon_uz;
35   }
36 }
```

Figure 9: The branch in the MCML kernel to which branch distribution is applied.

4.2 Results and discussion

The hardware platform we used is an Intel Core 2 Quad 9440 CPU with an NVIDIA Geforce GTX 480 GPU and 4GB of main memory. We used the CUDA 3.0 toolkit, running on Ubuntu 8.04.

4.2.1 Iteration delaying

We first evaluate the performance benefit of iteration delaying as a function of the size of the branch code relative to the non-branch code. We evaluate each of the three strategies that determines the convergent branch direction in each iteration: majority-vote, round-robin and round-robin with idle iteration removal. The speedup results are shown in Figure 10 for randomized branch directions, as described above. For majority-vote, `thresh` is set to 16 and for round-robin, the cycle (as defined in Section 3.1.1) is set to 50%. The horizontal axis is the *branch ratio* r, defined as the ratio of the size of each branch path to that of the non-branch code. We vary r from 1 to 50. For all strategies, the speedup starts below 1.0 (due to the instruction overhead) and increases as r increases. The best speedup achieved by majority-vote is around 1.18x, noticeably lower than 1.30x achieved by round-robin and its variant. We attribute this to the scheme that stops iteration delaying at some point to prevent majority-vote from starving threads (Section 3.1). A detailed look at the execution shows that only 67% of real iterations have completed when the stopping condition occurs. Therefore, around 1/3 of the iteration space is still executed divergently. In contrast, the round-robin strategies do not starve threads and can be applied to the end of the loop iterations. Further, idle iteration removal results in lower performance for round-robin when the branch ratio is small. This is expected because the extra warp-vote instruction for *every* loop iteration, which compiles into at least 5 PTX instructions, only removes idle iterations that occur occasionally.

We also evaluate the performance benefit of iteration delaying as a function of branch behavior. The results is shown in Figure 11 for a branch ratio r of 8, which is representative of our MCML application discussed below. For majority-vote, `thresh` is 16; for both round-robin strategies, the cycle is 50%. The horizontal axis is the *branch frequency* f, defined as the fraction of time the branch is taken, and it is varied between 0% and 100%. The figure shows that while majority-vote is not sensitive to the branch frequency, round-robin favors frequent branch direction changes, as expected. Also, the end-points ($f = 0\%, 100\%$) of the curve for majority-vote strategy reveal the impact of the instruc-

tion overhead introduced by the optimization, because the branch is convergent in these cases. The 10% slowdown is expected since each branch path has 160 FMA instructions and the overhead is about 15 instructions.

Finally, we evaluate the performance impact of the parameters of the decision strategies: `thresh` for majority-vote and the cycle for round-robin. We vary `thresh` between 1 and 32 and we vary the cycle between 10% and 90%, as shown in Table 1. The branch ratio is 8 and the branch frequency is 10%. The results show that for all strategies, variance exists and thus, the choice of these parameters is important.

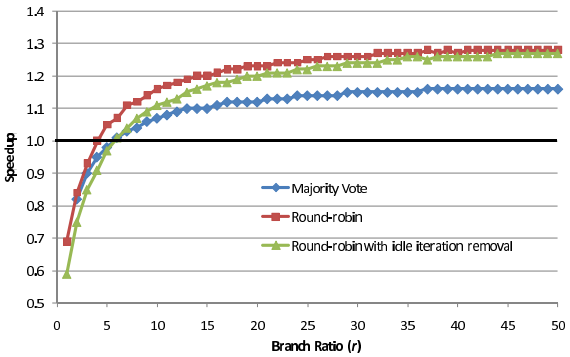

Figure 10: Performance impact of iteration delaying as a function of the branch ratio r.

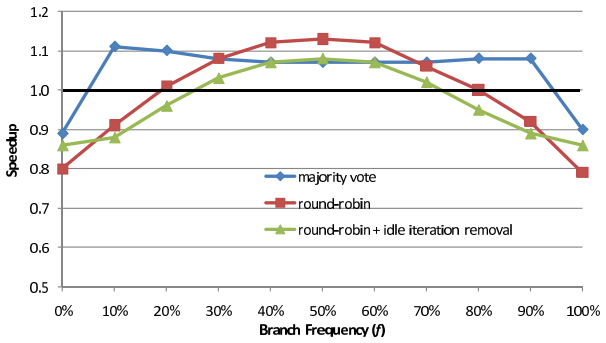

Figure 11: Performance impact of iteration delaying as a function of the branch frequency f.

Majority Vote		Round-robin	
Threshold	Speedup	Cycle	Speedup
1	0.89	90%	1.05
8	1.15	70%	1.04
16	1.11	50%	0.91
24	0.95	30%	0.92
32	0.73	10%	0.55

Table 1: Performance impact of decision parameters.

Iteration delaying improves the performance of MCML by up to 1.12x. This is achieved with a round-robin strategy with a cycle of 75%. The branch ratio for MCML is 8. Thus, the achieved speedup of the application is consistent with those achieved by the synthetic benchmark, as shown in Figure 11.

4.2.2 *Branch distribution*

We study the performance impact of branch distribution using SYN-BRDIS, varying: the size of the code factored out relative to the size of one path of the divergent code (i.e., $R = N * M/20$) and the number of inputs (and outputs) needed by the factored-out code (i.e. M). The result is shown in Figure 12. Out experiments indicate that the performance is not sensitive to M, thus the plot is not shown. In addition to the measured speedup, the figure also plots the theoretical speedup with respect to R, which is a ratio of (approximate) instruction counts before and after applying branch distribution, i.e., $2(1 + R)/(2 + R)$. We attribute the 10% difference between the measured and theoretical speedup to the non-branch code per iteration: the 19 PTX instructions that initializes `val` using the MWC random number generator.

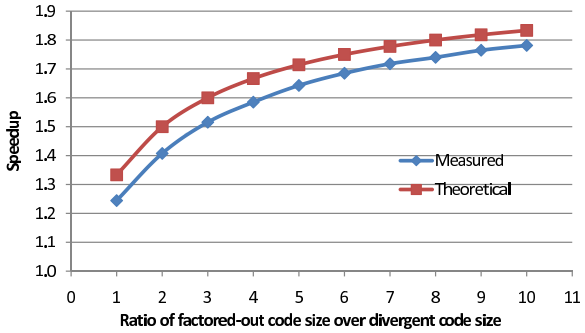

Figure 12: Performance impact of branch distribution as a function of the relative size of the code factored out.

We evaluate the performance impact of branch divergence on MCML. We find that it greatly depends on the frequency of the first-level branch path at line 5 and 6 of Figure 8 is executed. We construct three program inputs, i.e. a 7-layer skin model with different per-layer properties, so that this branch path is executed 4%, 30% and 70% of the time respectively[2]. The speedup brought by branch distribution in these three cases are 5.6%, 2.9% and 16.1% respectively. The benefit varies non-linearly with the "hotness" of the parent branch. This is expected because the target branch (i.e., line 2 of Figure 9) contains further branches inside, and the input change may cause drastic change of the photon movement behavior (e.g., more likely to reflect than to transmit) and the dynamic instructions executed in the target branch paths.

Finally, we isolate the impact of branch distribution on the target branch in MCML. We extract the code in Figure 9 into a separate kernel, and perform a fixed number of simulation steps. We obtain a speedup of 1.32x. By inspecting the PTX code, we find that the ratio R is around 2. The result of SYN-BRDIS tells us that the speedup is roughly around 1.4x, which is aligned with the actual result.

5. RELATED WORK

Loop collapsing is a standard compiler transformation that can reduce divergence in the case shown in Figure 1c. It is incorporated into a GPU compiler framework by Lee et al. [6]. However, they apply loop collapsing only to a very specific pattern of irregular loop nests that is commonly used in sparse matrix-vector multiplication. In contrast, the opti-

[2]These numbers do not represent the amount of execution time spent in this path.

mizations we proposed are more generic and target other types of branch divergence.

Code factoring techniques, such as hoisting, sinking and procedural abstraction, have been developed to reduce code size in embedded systems [3]. Equivalent instruction sequences are identified through isomorphism in the corresponding control/data flow graphs, and factored out into functions through register renaming. Branch distribution is similar to these techniques, but with a different goal: reducing branch divergence. This requires us to explore different trade-offs when applying this optimization.

Dynamic Warp Formation (DWF) [4] is a hardware mechanism to improve the efficiency of SIMD branch execution on GPUs. Every cycle the thread scheduler reforms warps from the active threads by grouping those that are executing the same path (e.g., with the same next PC value) into the same warp. Apart from the additional hardware that does thread regrouping, DWF requires the register file to be augmented to allow each thread to access other threads' registers, in order to reduce the overhead of context migration. In contrast, our optimizations require no hardware support beyond what is available on GPUs today.

Zhang et al. [19] perform runtime data re-mapping across multiple warps, which can be inefficient and requires host-GPU communication. They also place restrictions on the pattern in which threads may read their inputs. In contrast, our optimizations target threads within a warp, need no data re-mapping, place no restrictions on data access patters and are amenable to compiler implementation.

There is also work that aims to improve performance in the presence of branch divergence by increasing parallelism during divergent execution. Meng et al. [10] propose a hardware mechanism, called Dynamic Warp Subdivision (DWS), that splits a warp into sub-warps at divergent branches and that can be scheduled independently and executed in an interleaved fashion. Carrillo et al. [2] propose a code transformation, called branch splitting, in which a parallelizable loop that encloses a multi-path branch is split into multiple loops, each containing one branch path. In this way, the single kernel that executes the original loop is split into smaller ones, each of which has potentially lower register usage and may be executed with higher level of parallelism. This work is orthogonal to the optimizations we propose.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented two novel optimizations that aim to reduce branch divergence: iteration delaying and branch distribution. Both optimizations can be applied directly in software targeting NVIDIA Fermi-based GPUs. Our preliminary evaluation shows that the two optimizations can improve the performance of both synthetic benchmarks and a real-world application. Thus, we are encouraged to pursue compiler automation of the optimizations.

There are several directions for future work. One direction is to explore the selection of the parameters of the optimizations, for example to adapt to the branching patterns of an application. A second direction is to assess the combined effect of our optimizations with dynamic warp formation.

7. REFERENCES

[1] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Compiler Construction*, pages 244–263, 2010.

[2] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for GPGPU. In *Proc. of Computing frontiers*, pages 147–150, 2009.

[3] S. K. Debray and et al. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22:378–415, March 2000.

[4] W. Fung and et al. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Trans.Archit.Code Optim.*, 6(2):1–37, 2009.

[5] B. Jang and et al. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. on Parallel and Distributed Systems*, 22(1):105 –118, Jan. 2011.

[6] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proc. of PPoPP*, pages 101–110, 2009.

[7] A. Leung and et al. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proc. of GPGPU*, pages 51–61, 2010.

[8] E. Lindholm and et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[9] W. C. Y. Lo and et al. GPU-accelerated Monte Carlo simulation for photodynamic therapy treatment planning. In *Proc. of ECBO*, 2009.

[10] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proc. of ISCA*, pages 235–246, 2010.

[11] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[12] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide v3.0, Mar. 2010.

[13] NVIDIA. NVIDIA GF100: World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism, Aug. 2010.

[14] S. Ryoo and et al. Program optimization carving for GPU computing. *J.Parallel Distrib.Comput.*, 68(10):1389–1401, 2008.

[15] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proc. of PACT*, pages 513–522, 2010.

[16] S.-Z. Ueng and et al. CUDA-Lite: Reducing GPU programming complexity. In *Proc. of LCPC*, pages 1–15, 2008.

[17] M. Wolfe. Implementing the PGI Accelerator model. In *Proc. of GPGPU*, pages 43–50, 2010.

[18] Y. Yang and et al. A GPGPU compiler for memory optimization and parallelism management. In *Proc. of PLDI*, pages 86–97, 2010.

[19] E. Zhang and et al. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proc. of Supercomputing*, pages 115–126, 2010.