# Branch and Data Herding: Reducing Control and Memory Divergence for Error-tolerant GPU Applications

John Sartori and Rakesh Kumar

***Abstract*—Control and memory divergence between threads within the same execution bundle, or warp, have been shown to cause significant performance bottlenecks for GPU applications. In this paper, we exploit the observation that many GPU applications exhibit error tolerance to propose branch and data herding. Branch herding eliminates control divergence by forcing all threads in a warp to take the same control path. Data herding eliminates memory divergence by forcing each thread in a warp to load from the same memory block. To safely and efficiently support branch and data herding, we propose a static analysis and compiler framework to prevent exceptions when control and data errors are introduced, a profiling framework that aims to maximize performance while maintaining acceptable output quality, and hardware optimizations to improve the performance benefits of exploiting error tolerance through branch and data herding. Our software implementation of branch herding on NVIDIA GeForce GTX 480 improves performance by up to 34% (13%, on average) for a suite of NVIDIA CUDA SDK and Parboil [16] benchmarks. Our hardware implementation of branch herding improves performance by up to 55% (30%, on average). Data herding improves performance by up to 32% (25%, on average). Observed output quality degradation is minimal for several applications that exhibit error tolerance, especially for visual computing applications.**

EDICS: Parallel Architectures and Design Techniques

## I. Introduction

GPUs and similar SIMD architectures are becoming increasingly popular in the high performance desktop, server, and scientific computing domains, especially as single-thread performance languishes. With the emergence of high-level programming models such as NVIDIA CUDA [14], ATI Stream, OpenCL [7], and Microsoft DirectCompute [10], and the corresponding general purpose GPUs (GPGPUs), focus has shifted from exclusively graphics processing applications to also supporting myriad data-parallel applications. Single instruction multiple data (SIMD) architectures are area and energy efficient for data-parallel applications, as instruction sequencing logic is shared by multiple execution units, leaving more area and power for the execution units themselves. However, the performance delivered by these architectures continues to lag the performance demands of emerging applications, as performance is often limited by the number of execution units that can fit within the area and power budget of the chip. As such, performance optimizations for GPUs and other SIMD architectures are an active area of research.

The nature of SIMD execution requires that groups of parallel threads that execute together (warps) must execute the same instruction in lockstep. While the SIMD nature of execution allows the processor design to be relatively simple, application performance may suffer significantly whenever threads in the same warp behave differently due to control or memory divergence [14], [12]. Control divergence results in serialized execution of divergent control paths, leaving execution resources idle and throttling parallelism. Similarly, memory divergence causes a warp to stall until the longest memory request for a vector load completes before executing any dependent instructions. Recent work has shown that control and memory divergence between threads within a warp cause significant performance bottlenecks for many GPU applications [9], [6].

In this paper, we attempt to reduce the amount of control and memory divergence in GPU applications to improve their performance. We draw on the observation that many GPU applications produce acceptable outputs even if a small number of threads in a SIMD execution unit are forced to go down the wrong control path or are forced to load from an incorrect (albeit spatially local) address. This is not surprising, considering that many GPU applications are data-intensive – different threads in a warp are often operating on similar, often spatially correlated, data. Similarly, the fraction of branches that diverge tends to be small (even though the corresponding performance degradation is large). We exploit these observations to propose two novel optimizations – *branch herding* and *data herding*. Branch herding eliminates control divergence by forcing all threads in a warp to take the same control path. This prevents serialization of branch paths that causes execution resources to remain idle for threads on the inactive control path. Data herding eliminates memory divergence by forcing each thread in a warp to load from the same memory block. This reduces the number of memory stalls. This also reduces bandwidth pressure, as fewer blocks need to be loaded from memory. With the aid of static and profiling-based analyses, branch and data herding are applied discriminately to safely increase performance while maintaining acceptable output quality.

This paper makes the following contributions:

- We demonstrate the potential for significant performance benefits without a significant degradation in output quality from carefully reducing control and memory divergence for several GPU applications that exhibit error tolerance. We confirm that an indiscriminate elimination of divergence can cause significant degradation in output quality. Similarly, a naïve implementation of divergence reduction can actually *degrade* performance in some scenarios.
- We propose two optimizations – branch herding and data herding – that eliminate control and memory divergence, respectively. Our software implementation of branch herding involves using CUDA intrinsics to force diverging threads to take the same direction at a branch as the majority of the threads. A hardware implementation

J. Sartori and R. Kumar are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: {sartori2,rakeshk}@illinois.edu.

of branch herding uses majority logic to identify the branch direction all threads should take. Data herding is implemented in hardware by identifying the most popular memory block (that the majority of loads map to) and mapping all loads from different threads in the warp to that block.

- While it is known that several data-parallel application can tolerate errors [2], [18], [22], what is really needed is a way to exploit available error tolerance safely and efficiently. To support our branch and data herding implementations, we also propose a static analysis and compiler framework that guarantees that control and memory errors introduced by herding will not cause exceptions, a profiling framework that aims to improve performance while maintaining acceptable output quality, and hardware optimizations to improve the performance benefits of herding.
- We quantify the potential performance benefits from different implementations of branch and data herding. Our software implementation of branch herding on NVIDIA GeForce GTX 480 improves performance by up to 34% (13%, on average) for a suite of NVIDIA CUDA SDK and Parboil [16] benchmarks. Our hardware implementation of branch herding improves performance by up to 55% (30%, on average). Data herding improves performance by up to 32% (25%, on average).
- We also evaluate output quality degradation for different GPU kernels and full applications utilizing our implementations of branch and data herding. We provide quantitative evaluations for all applications and visual evaluations when available. Our framework aims to maintain acceptable output quality degradation for applications that can tolerate errors.

Note that our evaluations in this paper assume a GPU architecture that matches current-generation NVIDIA CUDA devices [13], [14], [12], though we expect the ideas to be applicable to other GPU / SIMD architectures as well.

The rest of the paper is organized as follows. Section II provides background on control and memory divergence and motivates data and branch herding. Section III describes branch herding and its various implementations. Section IV describes data herding and its implementation. Section V describes a safety, performance, and output quality assurance framework for branch and data herding. Section VI discusses the methodology of our study. Section VII presents results and analysis. Section VIII discusses related work. Section IX summarizes and concludes.

## II. Background and Motivation

Below we describe the control and the memory divergence problem and discuss how carefully eliminating divergence may lead to significant performance benefits.

### A. Control Divergence

SIMD architectures bolster throughput by sacrificing per-thread control flow logic in order to increase the number of execution units on a chip. Since multiple threads (a warp) execute the same instruction in lockstep on a SIMD multiprocessor (SM), only one block of instruction fetch, decode, and issue logic is needed per SM, allowing a greater fraction of the GPU's power and area budget to be spent on execution units. While such an architectural organization is beneficial for most data-parallel applications, the requirement that all threads in a warp must execute in lockstep can lead to inefficiencies when different threads take different control paths at a branch (control divergence).

```
while (--i && (xx + yy < T(4.0))) {
    y = x * y * T(2.0) + yC;
    x = xx - yy + xC;
    yy = y * y;
    xx = x * x;
} return i;
```

Fig. 1. The main computation loop for Mandelbrot. The loop is unrolled 20 times in the actual application kernel.

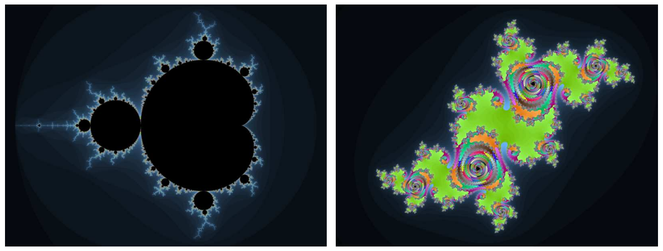


Fig. 2. Original Mandelbrot (left) and Julia (right) images. The color of each pixel corresponds to the number of main loop iterations executed by a thread to determine whether the point is in the Mandelbrot (or Julia) set.

Because instruction sequencing logic is shared by all execution lanes in a SM, the common mechanism for resolving control divergence in a GPU is to execute instructions from one control path for a subset of threads until reaching a point where control reconverges, then beginning execution on the other control path for the remaining threads until revisiting the reconvergence point [6], [14], [21], [4]. Since divergent branches necessarily throttle the parallelism and throughput of a SM, they can cause significant performance degradation for GPU applications [6], [9]. For a warp size of 32 (common in NVIDIA CUDA GPUs [14]), execution could be slowed down by a factor of 32 if all threads take divergent control paths through a section of code.

To understand this better, consider Mandelbrot [20] – an application from the NVIDIA CUDA SDK that exhibits control divergence. Mandelbrot generates the Mandelbrot and Julia sets – complex fractal patterns that are characterized by simple equations. Figure 1 shows the main loop of the kernel used to compute the Mandelbrot and Julia sets. In the actual kernel code, the loop is unrolled 20 times. Each thread in the program computes whether a particular point in the complex plane is in the Mandelbrot (or Julia) set. The program outputs images depicting the Mandelbrot and Julia sets (Figure 2). The color of a pixel corresponds to the number of main loop iterations ($i$) a thread executes to determine whether the point is in the Mandelbrot (or Julia) set.

Control divergence arises in Mandelbrot because the number of iterations required to determine whether a point is in the Mandelbrot (or Julia) set varies based on the point's location, especially in image regions near the set boundary, where some threads execute many iterations while others finish quickly. Divergence results in reduced parallelism, as some lanes in the SMs go unused while threads that have finished their computations wait until all threads in the same warp reach a reconvergence point.

The effect of control divergence on performance can be

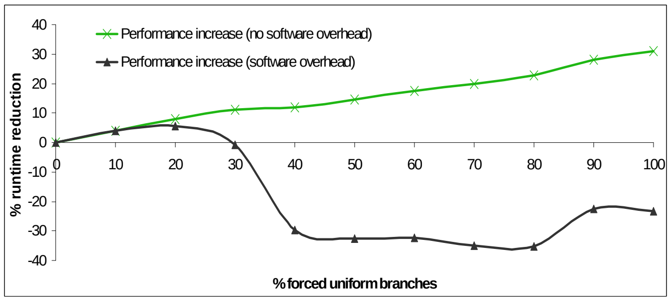


Fig. 3. The performance of Mandelbrot can be increased by forcing uniformity for more branches. However, if software overhead is added to ensure branch uniformity, increasing the number of affected branches increases overhead and can even result in degraded performance.

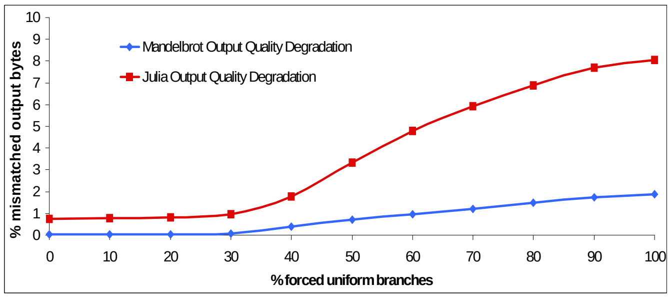


Fig. 4. While eliminating control divergence can increase performance, blindly forcing branch uniformity can result in degraded output quality.

significant. Figure 3 shows the potential performance increase (runtime reduction) if control divergence can be eliminated for a fraction of the static branches in Mandelbrot (from 0% to 100% of branches). The branches are chosen uniformly randomly when the fraction is less than 100%. Control divergence is preempted by changing the source code to vote within a warp on the condition of a branch and forcing all threads in the warp to take the same (majority) direction at the branch (details in Section III). Experiments were run natively on a NVIDIA GeForce GTX 480 GPU (details in Section VI).

While only 10% of dynamic instructions in Mandelbrot are branches, and less than 1% of branches diverge, performance can potentially be increased by 31% by eliminating control divergence. As the *no software overhead* performance series in Figure 3 demonstrates, performance increases for Mandelbrot as control divergence is eliminated for more branches. Figure 4 shows that the quality of the Mandelbrot output set degrades by less than 2%, even when divergence has been eliminated for all static branches. This shows that for certain error-tolerant applications, it may be possible to get significant performance benefits from eliminating control divergence for minimal output quality degradation. A quick look at the Julia output set, however, also suggests that an indiscriminate selection of branches for herding may result in significant output quality degradation for several applications. Therefore, any implementation of branch herding needs to carefully select the branches to target. Figure 5 shows visual representations of the Mandelbrot and Julia output sets as the percentage of forced uniform branches increases from 20% to 100% in increments of 40%.

The *software overhead* performance series of Figure 3 demonstrates another important consideration for any technique that eliminates control divergence. Since the fraction of

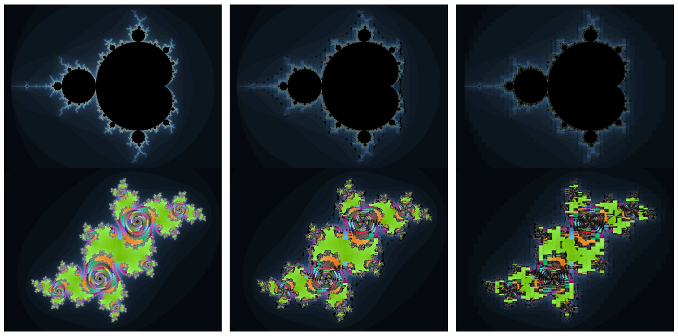


Fig. 5. Progression of Mandelbrot (top) and Julia (bottom) images from 20% to 100% forced branch uniformity in 40% intervals.

divergent branches in a program may be small (in this case, less than 1%), an indiscriminate application of a technique to all branches may result in significant overhead that diminishes or even eliminates performance gains that result from reduced divergence. This result reinforces the conclusion that care should be exercised in selecting the branches to target for elimination of control divergence. Also, a low-overhead mechanism for eliminating control divergence may enable significantly more benefits. The result also confirms that naïve implementations of techniques to eliminate control divergence may actually *decrease* performance in some scenarios.

## B. Memory Divergence

Like control divergence, memory divergence occurs when threads in the same warp exhibit different behavior. In the GPU, a load operation for a warp is implemented as a collection of scalar loads, where each thread potentially loads from a different address. When a load is issued, the SM sets up destination registers and corresponding scoreboard entries for each thread in the warp. The load then exits the pipeline, potentially before any of the individual thread loads have finished. When all the memory requests corresponding to the warp load have finished, the destination vector register is marked as ready. Instructions that depend on the load must stall if any lanes of the destination vector register are not ready.

Memory divergence occurs when the memory requests for some threads finish before those of other threads in the same warp [9]. Individual threads that delay in finishing their loads prevent the SM from issuing any dependent instructions from that warp, even though other threads are ready to execute. Memory divergence may occur for two reasons. (1) The time to complete each memory request depends on several factors, including which DRAM bank the target memory resides in, contention in the interconnect network, and availability of resources (such as MSHRs) in the memory controller. (2) Since the target data for a collection of memory requests made by a warp may reside in different levels of the memory hierarchy, the individual memory operations may complete in different lengths of time.

Most GPU architectures do not implement out-of-order execution due to its relative complexity and hardware cost. Rather, GPUs cover long latency stalls by multithreading instructions from a pool of ready warps. Providing each SM with plenty of ready warps ensures that long latency stalls will not be exposed. Memory divergence delays the time when a warp may execute the next dependent instruction, cutting into the pool of ready warps and potentially exposing stalls

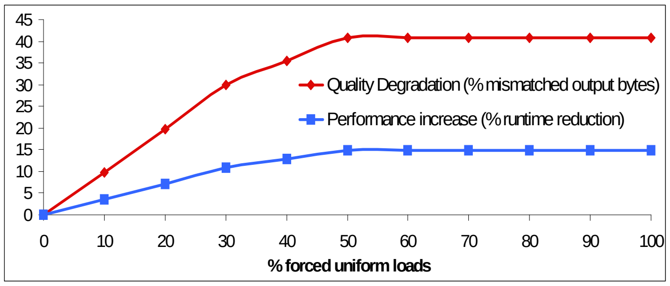


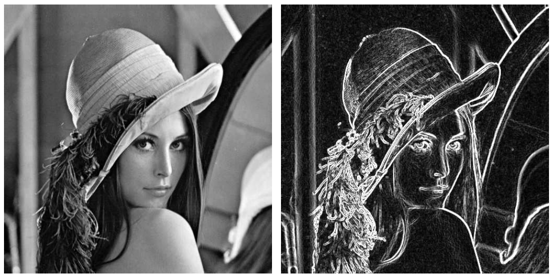
Fig. 6. Eliminating memory divergence (forcing more uniform loads) increases performance but also degrades output quality.

that throttle performance. Divergent memory accesses may also throttle performance by consuming additional resources, such as MSHRs and memory bandwidth. Therefore, eliminating memory divergence can potentially increase performance, especially for data-intensive GPU applications.

Another rarely discussed impact of memory divergence is on memory utilization. If different loads fetch from different memory blocks, more memory blocks need to be brought into the chip. (A memory block is the unit of memory pulled in from the memory system by a memory request.) More requests increase the bandwidth pressure on the GPU, which is often already bandwidth-limited. So, if memory divergence is eliminated (for example, when all loads fetch from the same memory block), bandwidth pressure reduces.

To gauge the potential benefit of eliminating memory divergence, we look at the SobelFilter application from the NVIDIA CUDA SDK. SobelFilter applies an edge detection filter kernel to an input image and produces an output image. Each thread in SobelFilter loads a block of pixels from the input image and processes them in different arrangements with the edge detection kernel. We eliminate load divergence for the three kernels of SobelFilter by modifying the application so that for each load, all threads in a warp load data from the same address (that of the first active thread in the warp). Thus, the individual thread loads can be coalesced into a single memory request, making divergence impossible.

While the actual loads for individual threads in a warp may indeed diverge, the threads all load data from a localized region of the input image. Since the image data exhibits spatial correlation, eliminating divergence by loading from an address that corresponds to a neighboring pixel may often return a similar or even identical value. Figure 6 shows the impact on performance and output quality of increasing the fraction of warp loads that are forced to load from the same address. The figure reveals that eliminating memory divergence (forcing load uniformity) increases performance by up to 15%. However, output quality is also degraded, resulting in up to 40% mismatching bytes in the output image. Thus, some intelligence may be required to determine how and for which loads to eliminate memory divergence such that acceptable output quality is maintained. Figure 7 shows the Lena input image along with the pristine filter output (0% forces load uniformity), while Figure 8 shows a progression of output images produced by filtering the Lena input image with an increasing fraction of forced uniform loads (from 20% to 100% load uniformity).



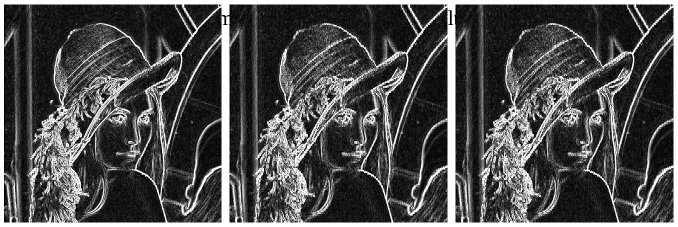


Fig. 8. Lena images processed by the Sobel edge detection kernel – progression from 20% to 100% forced load uniformity in 40% intervals.

## III. Branch Herding

The previous section demonstrated that for an application with divergent branches, eliminating control divergence has the potential to increase performance, possibly at the expense of output quality. Due to the unique handling of divergent control flow instructions in GPUs and the forgiving nature of many data-intensive GPU applications, we propose a SIMD-specific technique for eliminating control divergence. We call our technique *branch herding*. Branch herding eliminates control divergence by herding all the threads in a warp onto the control path taken by the majority of threads. Thus, when the threads in a warp each evaluate the boolean condition for a branch, the majority outcome is decided and all threads follow the majority decision, precluding the possibility of control divergence. Because control divergence is eliminated, branch herding has the potential to increase performance for applications with divergent branches. Also, for GPU applications that can tolerate errors, acceptable output quality can be maintained when branch herding is used (see Sections V and VII), even though some minority of threads are allowed to perform inexact computations.

The implementation of branching in GPUs leads to benefits for branch herding in addition to the elimination of branch path serialization. The normal implementation of branching in the GPU uses a reconvergence stack and a special reconvergence instruction that is inserted before a potentially divergent branch [6], [4], [21]. The reconvergence instruction passes to the hardware the location (PC) of the reconvergence point of the branch (the next instruction that will be executed by threads on both control paths). The instruction at the reconvergence point is also flagged using a special field in the instruction encoding [21], [4]. Whenever a branch is reached, a 32-bit thread mask is computed for the warp, indicating which active threads take the branch. If the branch diverges, the mask is pushed onto the reconvergence stack, along with the PC indicating the alternate branch target and the reconvergence PC. A subset of the threads (indicated by the mask) execute the taken branch path [21], [4], while the other lanes in the SM are idle. When execution reaches the reconvergence point, the stack is popped, and the remaining threads (indicated by the mask) begin executing from the stored PC. The next time

```
__device__ inline bool BRH(int condition){
return (__popc(__ballot(condition)) > BRH_THRESH);}

if( BRH(condition) )
while( BRH(condition) )
for(initial; BRH(condition); update)
```

Fig. 9. Software branch herding implementation and example uses.

the reconvergence point is reached, all threads that originally reached the branch begin executing together again. Note that this mechanism can also handle nested divergence.

Since branch herding eliminates control divergence, the reconvergence stack is not needed for herded branches. In addition, by ensuring that all branches will be uniform branches [12], branch herding obviates the need for the special reconvergence instruction. Thus, the compiler does not insert the reconvergence instruction when the branch herding compiler flag is set or when a kernel call or particular branch instruction is marked for branch herding. It may also be possible to eliminate the reconvergence instruction by identifying the reconvergence point using a field of the branch instruction.

### A. Software Branch Herding

Branch herding can be implemented relatively efficiently in software, using the CUDA intrinsics ballot ($\_\_ballot$) and population count ($\_\_popc$) [14]. The ballot intrinsic is a warp vote function that combines predicates computed by each thread in a warp and sets the $N^{th}$ bit in a 32-bit integer if the predicate evaluates to non-zero for the $N^{th}$ thread in the warp. In the context of branch herding, the result is a 32-bit integer that specifies the branch condition outcome for each thread in a warp. The ballot result is broadcasted to a destination register for each thread in the warp. We use the population count intrinsic to count the number of set bits in the ballot result. In context, this means that each thread knows how many threads in the warp should take the branch. The branch herding function compares the population count to 16 (half warp size) and returns *true* if the majority of threads take the branch and *false* otherwise. Figure 9 shows the software implementation of branch herding, and provides examples of how software branch herding can be used in programs, simply by passing the condition of a control statement (e.g., if, while, for) to the branch herding procedure.

### B. Hardware Branch Herding

Though our implementation of software branch herding only adds 3 extra instructions per branch, even this overhead may be intolerable in several scenarios, especially in tight loops or for programs that have a large fraction of branches that diverge only infrequently. Profiling information for benchmarks from the NVIDIA CUDA SDK and Parboil [16] suites that exhibit control divergence (Figure 10) reveals that the fraction of dynamic branches that diverge is indeed often very low. This is primarily because GPU programmers usually take pains to reduce potential control divergence. Nevertheless, as demonstrated in Section II, even a small fraction of divergent branches can significantly reduce performance. Ideally, branch herding should be implemented as a lightweight hardware mechanism to maximize potential benefits.

For the normal implementation of branching in the SM, each active thread evaluates the branch condition to identify whether it should fetch the next instruction from the branch target or fall through. After the branch condition has been evaluated for each thread, the SM combines the condition bits from the threads to form the 32-bit branch mask, then checks for uniformity of the mask (all 0s or all 1s). If the branch is not uniform, the SM updates the reconvergence stack, as explained above.

Hardware branch herding works the same way as the normal branching implementation, but instead of evaluating the uniformity of the mask and potentially updating the reconvergence stack, the SM evaluates the majority value for the mask. The majority condition determines the next instruction for all threads in the warp. Evaluation of the majority logic can take place in the timing slack apportioned for the uniformity logic and updating the reconvergence stack (since divergence is impossible with branch herding). Thus, hardware branch herding should not affect cycle time and should not incur additional cycles of overhead. Overhead will be in terms of area, since one block of majority logic is needed per SM. However, the area of one majority block for a 32-bit word is insignificant compared with the area of the SM. Branch herding logic can be activated at a coarse granularity by setting an enable bit in the hardware when the GPU is initialized for a kernel call or at a fine granularity by using a special field in the branch instruction to denote that the branch should be herded. The branch instruction contains an optional field (*.uni*) to identify a uniform branch (i.e., a branch for which it is possible to statically determine that the branch will not diverge). For branch herding, we override the field with a different code (*.hrd*) to indicate that the branch should be herded.

## IV. Data Herding

As discussed in Section II-B, memory divergence can occur when a load instruction for a warp generates multiple memory requests that access different regions of memory or different levels of the memory hierarchy. The number of memory requests generated by a load instruction is determined by coalescing hardware in the SM [14]. Memory coalescing is performed to determine the minimum number of unique memory requests that can satisfy the individual scalar loads that make up a vector load instruction. Each scalar load address maps to a block of memory (32, 64, or 128 bytes depending on the data type), and each memory request fetches one block from memory. If multiple scalar loads map to the same block of memory, they are coalesced into a single memory request. The GPU hardware is designed such that if all scalar loads in the same warp access consecutive addresses, they can be coalesced into a single request. Besides generating memory divergence, non-coalesced loads are inefficient because they generate multiple memory requests and fetch data that is not used, wasting precious memory bandwidth and consuming additional memory controller resources such as MSHRs.

We propose a data herding implementation based on a modified coalescing policy. The modified coalescing hardware generates only one memory request for a collection of scalar loads. Rather than forming a queue of unique memory requests required to satisfy the scalar loads, the modified hardware identifies the most popular memory block (that the majority of loads map to) and maps all loads to that block – some naturally and some forcefully. This is done by comparing the number of loads that coalesce into each potential memory request and

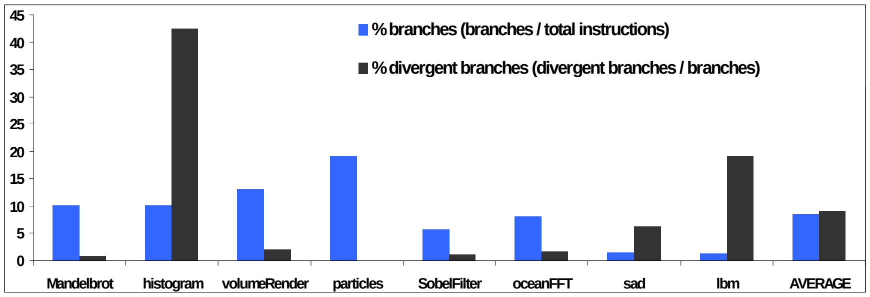


Fig. 10. Branch statistics for applications that exhibit control divergence.

discarding requests for all but the most popular block. The upper $N - log_2(line\_size)$ bits of an N-bit address identify the memory block that an address maps to. For any address that does not already map to the most popular memory block, the most significant $N - log_2(line\_size)$ bits of the address are overwritten with the bits that identify the most popular block. We propose data herding only for loads to ensure that all expected locations are initialized in the case of a store and to avoid conflicts that might result if stores were forcefully mapped to the same memory block.

Since our implementation of data herding ensures a single memory request for each load, and a single request is satisfied at only one level of the memory hierarchy, we prevent both types of memory divergence and also reduce memory traffic. Thus, bandwidth-limited applications may benefit substantially from data herding. Also, it is interesting to note that data herding, in itself, will never generate a memory exception, due to the nature of GPU memory design and allocation properties. In short, the threads involved all belong to the same process, and the entire memory block they will map to also belongs to the same process. An exception could, however, be generated, depending on how herded data are used later in the program. We address safety concerns associated with herding in Section V.

## V. Safety, Performance, and Output Quality Assurance for Branch and Data Herding

It is well-known that several data-parallel applications exhibit error tolerance [2], [18], [22]. To efficiently exploit this error tolerance through branch or data herding, the challenges lie in (1) guaranteeing that loading the wrong data or taking the wrong branch path will not cause an exception, and (2) maximizing performance improvement while maintaining acceptable output quality. In this section, we describe a *static analysis and compiler framework* that guarantees (1) by identifying branches and data that are safe for herding, and a *profiling framework* that targets (2) by identifying the subset of safe branches and data for which herding increases performance while maintaining acceptable output quality.

The first step in identifying safe branches and data for herding is to identify *vulnerable operations* that, if affected by an error, *might* cause exceptions. These are pointer dereference and array reference (vulnerable to Segfault), integer division (vulnerable to INT divide by zero), and branch condition check (vulnerable to stack overflow if an error causes infinite looping or recursion). We have written a clang [17] plugin that performs safety analysis by first parsing a program into its abstract syntax tree (AST) and searching through the AST to identify vulnerable operations.

After identifying vulnerable operations, our tool generates the control and data dependence graphs from the AST and traces through them to identify the branches and data that the vulnerable operations depend on. Then, to guarantee freedom from exceptions, the tool does not allow the compiler to insert herding directives for the branches and data identified as unsafe during static analysis. Note that control dependence-based static safety analysis for branch herding is conservative. Even if vulnerable operations are control-dependent on the outcome of a branch, the branch may be herded safely if it can be determined from the code that herding the branch will only result in skipping the dependent vulnerable operations. However, in such cases, it can be determined statically that the branch will diverge, and the divergence could be eliminated in software (provided that the resulting output quality degradation is acceptable). Preventing herding of "unsafe" branches and data ensures that errors induced by herding will only impact output quality.

After identifying which branches and data can be safely herded, the next step is to identify which can be profitably herded. As noted in Section II, one challenge of branch and data herding is determining which branches and data to herd so as to improve performance while maintaining acceptable output quality. While this can be done by the programmer, often with little effort (the programmer is often aware of which branches may diverge and whether or not it would be acceptable for some threads to perform inexact computations based on the associated branches or data), we also present an automated profiling-based framework for determining which safe branches and data may be most profitable for herding.

We use the CUDA Compute Profiler [14] to determine which safe branches/loads to safe data exhibit divergence. These are candidates for branch/data herding. Our profiling framework starts with no herded branches/loads, progressively marks a larger fraction of the candidate branches/loads for herding, and at each step profiles the program for a set of test inputs to characterize the space of output quality degradation and performance vs. number of herded branches/loads. From this sampling we can determine an approximate upper bound on output quality degradation corresponding to a given amount of herding by selecting the worst case degradation observed for a given amount of herded branches/loads. During runtime, performance counters [14] track the number of herded branches/loads and disable branch/data herding before the specified approximate threshold has been exceeded. To enable profiling and quality monitoring, the programmer should mark

```
// Static Safety Analysis
Generate Abstract Syntax Tree $AST(A)$ for Application $A$
Search $AST(A)$ to identify Vulnerable Operations $VO$
foreach(Vulnerable Operation $v \in VO$)
    Trace Control and Data Dependencies of $v$ to classify Safe/Unsafe Branches/Data
// Quality and Performance-targeted Profiling
Profile $A$ and identify Divergent, Safe Branches/Loads as Herding Candidates $C$
Herded Branches/Loads $H = \emptyset$
Baseline Output Quality Degradation,Performance $= Q(\emptyset), P(\emptyset)$
foreach(Candidate $c \in C$)
    foreach(Test Input $t$)
        Profile Output Quality Degradation $Q(H+c,t)$, Performance $P(H+c,t)$
    if($\exists$ $t$ such that $P(H+c,t) > P(H,t)$)
        Approximate Quality Degradation Bound $B(H+c) = max[Q(H+c,t)|_t$
        $H += c$
// Runtime Quality Monitoring
User specifies desired maximum Output Quality Degradation $Q_{max}$
Use Profiling Data to find maximum Herding Threshold $T_h$ such that $B(T_h) \leq Q_{max}$
Count Herding Instances $I_h$ and disable herding when $I_h == T_h$
```

Fig. 11. Pseudocode describing safety, performance, and output quality assurance framework for branch and data herding.

TABLE I. BENCHMARKS

| Benchmark | Description (†CUDA SDK, ‡ Parboil) |
|---|---|
| Mandelbrot | Compute Mandelbrot and Julia sets† |
| histogram | 64 and 256-bin Histograms† |
| volumeRender | Volume Rendering of 3D Textures† |
| particles | Particle Interaction Simulation† |
| SobelFilter | Sobel Edge Detection Filter† |
| oceanFFT | Ocean Heightfield Simulation† |
| binomialOptions | Binomial Option Pricing† |
| nbody | Gravitational n-body Simulation† |
| dxtc | DirectX Texture Compression† |
| recursiveGaussian | Recursive Gaussian Blur Filter† |
| lbm | Lattice-Boltzmann Method Fluid Dynamics‡ |
| sad | Sum of Absolute Differences‡ |

the variable in the code that represents output quality and specify the desired approximate bound on output quality degradation. Figure 11 presents pseudocode describing the control flow of our safety, performance, and output quality assurance framework for branch and data herding.

It should be noted that our profiling framework can only provide output quality guarantees for profiled inputs (or inputs similar to the profiled inputs). For all other inputs, we only provide an approximate upper bound on output quality degradation. However, we observed that the approximate bound is often effective in practice. Creating more rigorous techniques for performance and output quality assurance is a subject of ongoing work.

Note that while hardware-based herding implementations can improve the performance benefit of herding (Section VII), software-based herding can be implemented for off-the-shelf GPUs and applications, and thus has the potential to demonstrate real, immediate benefits of exposing control and data errors in applications. In fact, our software herding results (Section VII) show speedups for applications running natively on NVIDIA GeForce GTX 480. Typically, we use data herding for all loads to the largest data structure of the application identified as safe for herding. Section VII provides information on which branches and data were identified as safe and profitable for herding by our framework. Where possible, we aim for conservative results by using input data not characterized during profiling when capturing performance and output quality results.

## VI. METHODOLOGY

We perform experiments using two different execution environments. We run branch herding experiments natively on a CUDA system comprised of a NVIDIA GeForce GTX 480 GPU and a 2.27 GHz Intel Xeon E5520 CPU with 24 GB of memory. The NVIDIA CUDA v3.2 Toolkit and SDK are installed on the system.

Software branch herding performance and output quality are measured directly at runtime. Thus, reported benefits are for native execution on a state-of-art GPU architecture. To measure the number of cycles taken to execute a kernel that uses hardware branch herding ($total\_cycles_{HW\_br\_herd\_kernel}$), we start with the number of cycles taken to execute the same kernel when software branch herding is used ($total\_cycles_{SW\_br\_herd\_kernel}$) and use CUDA Compute Profiler [14] profile counters to measure the number of instructions added by software branch herding function calls ($instruction\_count_{SW\_br\_herd}$). We scale these instruction counts by the CPI for the corresponding kernels ($CPI_{SW\_br\_herd\_kernel}$) and discount the total cycle count by this amount.

$$total\_cycles_{HW\_br\_herd\_kernel} = total\_cycles_{SW\_br\_herd\_kernel} - instruction\_count_{SW\_br\_herd} * CPI_{SW\_br\_herd\_kernel}$$

Since evaluating data herding requires changing the behavior of coalescing hardware and cannot be easily emulated in software, we use the GPGPU-Sim [1] simulator for our experiments. The simulator models the behavior of a NVIDIA Quadro FX 5800 GPU and can run natively-compiled CUDA v2.1 binaries.

Potentially error-tolerant benchmarks are selected from the NVIDIA CUDA SDK and Parboil [16] benchmark suites. For evaluation of branch herding, we use all benchmarks for which more than 0.5% of the dynamic branches diverge. For data herding, we select benchmarks only from the NVIDIA CUDA SDK (v2.1) that are compatible with GPGPU-Sim v2.x, which is designed around CUDA v2.1. In addition to computation kernels, we evaluate full, end-to-end benchmarks (e.g., volumeRender, particles, oceanFFT, lbm, etc.), that contain multiple kernel calls, as a partial means of demonstrating that outputs from kernels that use herding are still acceptable in the context of the greater application. Table I provides short descriptions of the benchmarks used in our evaluations.

Although we do not expect any performance overhead for hardware branch herding (Section III), we collect results assuming different cycle overheads to provide both conservative and expected performance results. While we also expect that data herding based on modified coalescing can be performed in the same timing slack used for normal coalescing, we assume a cycle overhead for a more conservative estimate of the performance benefits.

## VII. RESULTS

### A. Branch Herding

Branch herding increases the performance of GPU applications that normally exhibit control divergence by preventing the serialization of branch paths and eliminating overheads associated with divergent branch handling. Figure 12 shows potential performance gains for branch herding for applications that normally exhibit control divergence. Hardware branch herding increases performance by 30% on average and up to 55% for individual applications. While we do not expect any performance overhead for hardware branch herding (see Section III), we also show conservative results that assume a 1 cycle overhead for hardware branch herding. Our software branch herding implementation, which runs natively on commercial GPU products, achieves 13% performance benefits, on average. Recall that the software branch herding implementation targets only safe branches that exhibit divergence AND show benefits from software branch herding. Therefore, performance improvements are significantly higher than any naïve software branch herding implementation that targets all static branches (Figure 3).

Since branch herding exploits error tolerance to eliminate divergence, it may result in output quality degradation. Table II compares output quality degradation for the benchmarks with and without branch herding. Quantifying output quality degradation is difficult, because really, the consumer of the data determines whether or not it is acceptable, and acceptability is often application-dependent. We provide output quality measurements in terms of the quality metrics incorporated by the original benchmark writers, however, our framework is modular and can easily use any other metrics (e.g., SNR) of interest to the programmer or end user. Output quality degradation is reported in terms of the fraction of mismatching bytes in the program output, except where otherwise noted. Overall, branch herding does not result in much additional output quality degradation (and degradation can be approximately bounded by our framework). Branch and data herding may be especially applicable for visual computing applications (e.g., video rendering or gaming), where performance and energy-efficiency may be more critical than perfect output quality. We provide image outputs for several visual computing applications to demonstrate that post-herding output quality may often be acceptable for such applications.

***Mandelbrot***: In Mandelbrot, which is described in detail in Section II, typically only a small fraction of dynamic branches diverge, but divergence is spread over all of the static branches in the program. Analysis identifies all branches as safe for herding. While herding more divergent branches improves performance, the amount of branch herding that can be allowed depends on the desired output quality and the region of interest in the image, since the amount of divergence depends on the region of the Mandelbrot set being viewed. Regions with intricate detail can result in substantial divergence, while monochrome regions generate no divergence. Although the overall fraction of divergent branches is often small, they can significantly impact performance. Hardware branch herding achieves about 3.5x better performance improvement than the software version, since software branch herding adds overhead to many non-divergent branches in a relatively tight loop.

Output images resemble those in Section II. Note that because branch herding may estimate whether a point is in the Mandelbrot set before completely finishing the calculation for that point, even though some output pixels are not colored correctly by the application, the determination of the Mandelbrot set may be correct for those points. Thus, whether or not branch herding produces acceptable results may depend on whether the output data will be used, e.g., for a visualization or as a mathematical set.

***SobelFilter***: Divergence is targeted in the SobelFilter kernel (described in Section II) in corner cases where the computed output pixel value for one or more threads in a warp does not lie in the valid output range. Ignoring these cases with branch herding causes the affected pixel values to roll over on the

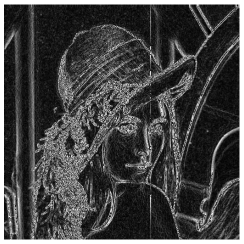


Fig. 13. Lena image processed by Sobel edge detection kernel with branch herding. Compare to original result in Figure 7.

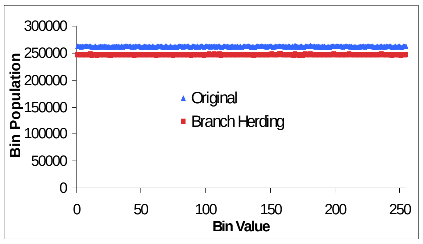


Fig. 14. Comparison of histogram output with and without branch herding.

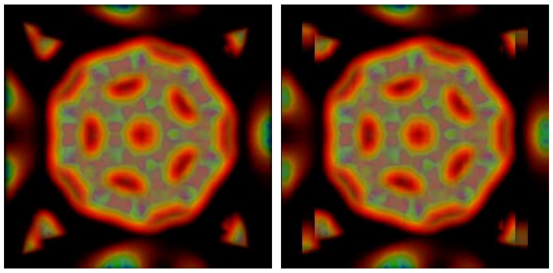


Fig. 15. Output comparison – original volume rendering (left) and branch herding result (right).

opposite side of the output range, adding some noise to the output image, which can be seen in Figure 13. Our framework confirms the safety of herding in this case, as it only affects pixel values. Herding is not profitable for all branches, since herding branches in tight loops that rarely diverge does not improve performance. Despite noise added by herding, edges are still detected.

***histogram***: Histogram has the highest fraction of divergent branches of all the applications we tested and sees considerable speedups for both software and hardware branch herding. All the divergence is caused by one static branch in a frequently-called function that adds data to the sub-histogram generated by a warp. (Sub-histograms are later merged together to create the final output.) This branch is safe for herding, as herding only affects histogram data. Branch herding may cause a few values not to be added to the bins, resulting in slightly undercounting the bin values. On average, bin values are undercounted by 6%, as seen in Figure 14. Output quality is reported as the average absolute difference between the bin values in the computed and reference outputs. It should be noted that quality degradation, and thus acceptability, depends on the characteristics of the input data.

***volumeRender***: VolumeRender renders a 3D texture. Although we can safely use branch herding for all the branches, most divergence is due to two static branches that cause threads to finish their computations either when the object at that pixel is opaque or too far away to be seen. Branch herding can result in some threads exiting early when the majority of threads in the same warp have finished their computations. Eliminating divergence improves performance significantly, and only increases output quality degradation by 1%. Figure 15 compares the original image produced by volumeRender to the image produced with branch herding.

***particles***: The particles application performs a simulation of physical interactions between a system of particles in an enclosed volume. The output describes the positions and

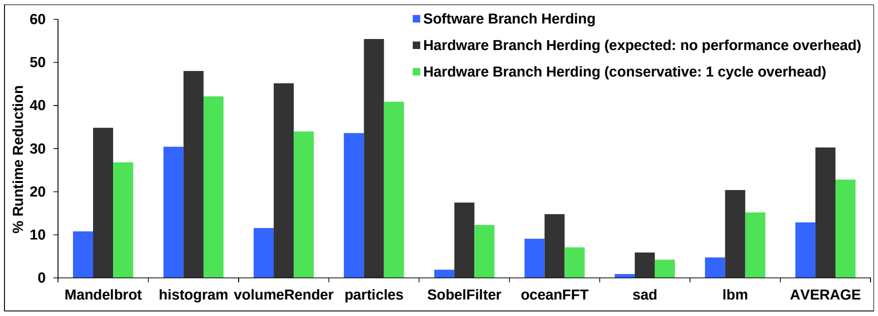


Fig. 12. Potential performance improvement for software and hardware branch herding. Although we don't expect any additional performance overhead for our implementation of hardware branch herding, we also show a conservative performance measurement assuming a 1 cycle overhead. Overhead is at most 1 cycle, since the additional logic (majority) is simpler than population count logic, which evaluates within a single cycle.

TABLE II. OUTPUT QUALITY DEGRADATION (%) FOR BRANCH HERDING COMPARED TO ORIGINAL

| % Mismatch | Mandelbrot | histogram | volumeRender | particles | SobelFilter | oceanFFT | sad | lbm |
|---|---|---|---|---|---|---|---|---|
| Original | 0.03 | 0.00 | 6.72 | 18.24 | 0.00 | 0.03 | 0.00 | 6.7E-7 |
| Branch Herding | 1.87 | 5.82 | 7.61 | 18.24 | 6.00 | 0.03 | 0.42 | 5.6E-5 |

velocities of the particles after a certain number of time steps. Herding branches identified by the framework only impacts these positions and velocities. A large fraction of the instructions in *particles* are branches that are part of collision checks between particles and with the surface of the enclosure. Even though the fraction of divergent branches is less than 1%, the number of divergent branches and the effect of divergence on performance is significant. Eliminating divergence with branch herding does not affect the output much because even if a collision is missed in one time step, it will likely be observed in a subsequent time step. This will result in a slightly different collision, but a similar or identical net effect. Both software and hardware branch herding improve performance significantly without producing any noticeable degradation in the output. Whether or not results are acceptable may depend on whether the simulation is for a visualization or a scientific experiment. For example, degraded output quality may be more acceptable in a physics simulation performed for a video game.

***oceanFFT***: The oceanFFT benchmark computes a heightfield for a region of ocean using spectral methods. Divergence in oceanFFT arises due to boundary checks at the edge of the simulated region. Ignoring divergence with branch herding results in some slight deviations in the output around the edges of the simulated region, but does not cause the reported output quality to change by a noticeable amount. In cases where the application would be used for a graphic visualization of the ocean, the deviations caused by branch herding would most likely be unnoticeable to the human eye.

***sad***: The sad benchmark performs sum of absolute differences-based motion estimation as part of the H.264 video encoder. Previous works have observed error tolerance for SAD-based motion estimation [18] due to the approximate nature of the block matching that it performs. We use branch herding for all safe branches in the sad kernel, which results in less than 0.5% output quality degradation. For most branches identified as unsafe, disallowing herding does not hurt much, since the alternate branch path is empty. In most cases, inexactness imposed by branch herding does not impact sad values enough to hinder block matching in the greater application. Thus, herding is often acceptable.

***lbm***: The lbm benchmark performs a lid-driven cavity fluid dynamics simulation involving a fluid that interacts with obstacles in a simulated volume. We use branch herding to eliminate divergence in the condition that tests for collisions between the fluid and an obstacle in a particular cell of the volume. Since the branch paths following the collision-detection branch contain many instructions, throughput can be affected substantially if the branch diverges. Though most cells in the volume remain error-free, branch herding causes some perturbations in the fluid simulation results. Thus, if the goal of the simulation is to simulate the fluid dynamics as accurately as possible (which may very well be the case in a scientific simulation), branch herding may be inappropriate for lbm.

## *B. Data Herding*

Figure 16 shows potential for performance improvements for various benchmarks with data herding. Benefits can be substantial or nonexistent, depending on the benchmark. For the three benchmarks that do not see benefits for data herding, less than 0.2% of dynamic instructions are loads. Output quality degradation associated with data herding is compared against original output quality degradation in Table III.

Data herding achieves performance benefits for two reasons. First, all non-coalesced loads to the herded data will be coalesced into a single memory request. This reduces memory bandwidth usage and contention for resources. Reduced bandwidth and contention can also reduce the latency of memory requests. Second, since only one memory request is made for a load, memory divergence is eliminated, and warps do not spend cycles waiting for additional requests to finish after the first request returns. Figure 17 shows results for data herding quantifying the reduction in bandwidth usage and cycles that ready warps spend stalled and waiting for outstanding memory requests.

Below we explain results for individual benchmarks.

***histogram***: In histogram, we target loads to the initial data set to be binned in the histogram, as well as the data in the sub-histograms computed by the warps. Static analysis identifies these data as safe for herding. The benchmark consists of two kernels – one that adds values to sub-histograms and one that

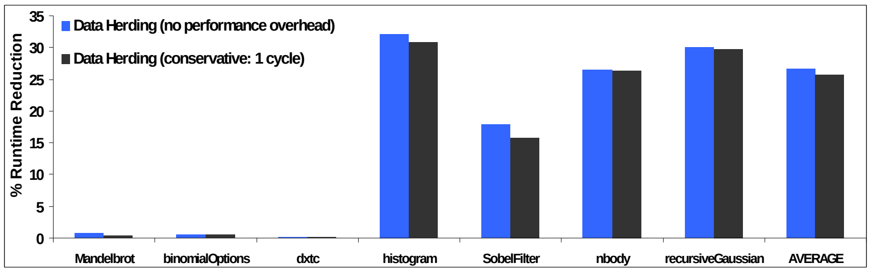


Fig. 16. Data herding improves performance for error-tolerant benchmarks, except when the fraction of loads is very small, leaving little opportunity for improvement.

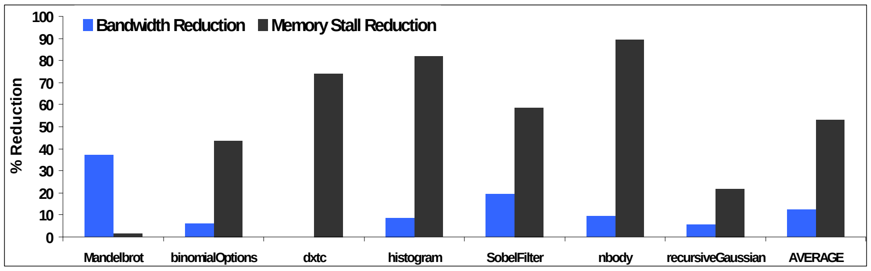


Fig. 17. Data herding improves performance by reducing memory stalls and bandwidth usage due to divergent memory requests.

TABLE III. OUTPUT QUALITY DEGRADATION (%) FOR DATA HERDING COMPARED TO ORIGINAL

| % Mismatch | Mandelbrot | histogram | nbody | binomialOptions | SobelFilter | dxtc | recursiveGaussian |
|---|---|---|---|---|---|---|---|
| Original | 0.02 | 0.00 | 0.00 | 3.8E-5 | 0.00 | 0.019 | 0.00 |
| Data Herding | 0.99 | 0.6 | 0.95 | 3.8E-5 | 1.81 | 0.019 | 0.00 |

merges sub-histograms. Most of the speedup from data herding comes from the kernel that performs merging, since it can generate many non-coalesced loads. While we observed that data herding often has only a small effect on output quality, output quality degradation depends on the characteristics of the input data. For example, uniformly distributed random data can be herded without affecting output quality substantially. On the other hand, if individual sub-histograms contain very distinct bin counts, data herding may be inappropriate for this benchmark. This brings up an important point to remember about profiling-directed herding. Output quality could potentially change undesirably for a pathological input data set. Thus, while our results do not guarantee acceptable output quality for the benchmarks over all possible data sets, they do demonstrate the potential for benefits for error tolerant applications, especially if the target data set can be accurately characterized.

***nbody***: Nbody performs an all-pairs N-body simulation for a collection of bodies. The application is considerably bandwidth-limited, especially as the number of bodies increases, since the data requirement scales approximately as $O(N^2)$, stemming from the $O(N^2)$ forces that exist between N bodies. The output of the N-body simulation describes the positions of all the bodies after a specified number of timesteps. We use data herding for the body data and observe less than 1% output quality degradation, measured in terms of the average absolute difference in body positions between the computed output and a reference data set. While the deviations in the output set are visually imperceptible, they do exist. Thus, herding may be appropriate for a visualization, but may be inappropriate for a high-precision scientific simulation.

***SobelFilter***: As in Section II, we herd image data for SobelFilter. While the performance results are similar to the maximum benefits achieved in the motivational experiment, the output quality degradation is significantly less, since loads that map to the most popular memory block receive their actual data with our proposed implementation of data herding (Section IV). Output quality is also better than in the branch herding case, since data herding takes advantage of spatial correlation in the image data, which contributes to the error resilience of SobelFilter. Since the output image after herding is visually indistinguishable from the original filtered image, we omit the image here to save space and refer the reader to the images in Section II.

***recursiveGaussian***: RecursiveGaussian performs Gaussian blur filtering on an input image. As in the case of SobelFilter, we herd the input image data. Error tolerance stems from the spatially correlated image data and the nature of the Gaussian filtering operation. Since the output value for a pixel is a weighted sum of the neighboring pixels, based on a Gaussian function, mixing in a few incorrect values is usually imperceptible, especially if the incorrect pixel values are close to the intended values due to spatial correlation. Because of the shape of the Gaussian function, the farther a neighboring pixel is from the pixel being computed, the less it affects the output. Thus, ignoring memory divergence due to non-contiguous data that cannot be coalesced usually has little effect on the output, since the data tend to be further apart in the image. We often did not observe any difference in output quality when data herding was used. Of course, output quality degradation may be greater for highly uncorrelated inputs. Figure 18 compares the original filter result to the post-herding result for a sample input image.

***Mandelbrot, binomialOptions, and dxtc***: For these three

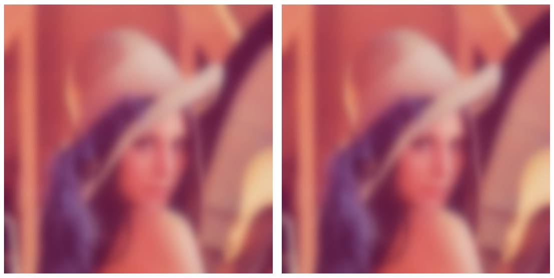


Fig. 18. Output comparison – Original gaussian blur filtering (left) and data herding result (right).

applications that do not see benefits from data herding, loads make up only 0.2% of the instruction mix. Thus, there is almost no potential for benefits with these applications to begin with.

## VIII. Related Work

***Dynamic Warp Subdivision***: The basic unit of SIMD execution is the warp. However, all threads in a warp must be ready in order to issue the next instruction. When SIMD restrictions stall execution, some threads in the warp may be ready while others are stalled. Normally, GPUs use warp-level multi-threading to hide latency, but this strategy requires a large, costly register file. Instead of deep warp-level multi-threading, dynamic warp subdivision [9] advocates using intra-warp latency hiding to increase throughput, by allowing a divergent warp to occupy multiple scheduler slots without increasing its register usage. This allows threads on divergent branch paths to subdivide their warp and execute independently. Similar to a previous work advocating "diverge on miss" [15], this also allows a subset of threads in a warp to continue execution when the remaining threads are still waiting on memory. The main drawback to dynamic warp subdivision is that it at least doubles the complexity and hardware cost of scheduling logic for each SM [9].

***Dynamic Warp Formation***: The goal of dynamic warp formation [6] is to increase hardware utilization by dynamically combining threads from multiple divergent warps. When multiple warps diverge, threads that take the same branch direction in one warp can be grouped with threads that take the same branch direction in other warps. Thus, fuller warps are formed dynamically, increasing throughput and partially mitigating the inefficiency caused by control divergence. The scheduler forms new warps out of ready threads by grouping threads that have the same next PC. Thread block compaction [5] applies dynamic warp formation whenever a divergent branch is encountered by synchronizing warps and compacting them into new warps, in which all threads take the same control path. A large warp microarchitecture [11] performs a similar optimization by exposing a larger warp of threads to the scheduler, which is able to select SIMD width-sized sub-warps that have the same control behavior.

While dynamic warp formation has the potential to increase throughput for some applications, it is not always possible to find enough divergent threads that take the same branch direction to fill a warp within the scheduling window of available warps. Thread block compaction may help in this regard, but in some cases, warps must remain partially empty anyway, even with the additional hardware overhead required for dynamic warp formation. Nested divergence complicates the problem, making it harder to find a full warp of threads with the same next PC.

Dynamic warp formation also adds complexity in the register file, which is typically heavily banked, such that each lane of a SM can access one bank of the register file. Dynamically grouping multiple threads from the same home lane into the same warp requires adding a crossbar network so that each thread can access its registers when mapped to a different lane than its home lane. This also results in bank conflicts when multiple threads from the same home lane are grouped into the same warp, such that register file accesses are serialized over multiple cycles. One possible solution to this problem involves passing along the home lane that a thread belongs to and using lane information during dynamic warp formation so that threads are only grouped together if they belong to different home lanes. This method reduces bank conflicts, but it adds complexity to the dynamic warp formation hardware and also makes it somewhat harder to find threads that can be grouped into efficient, full warps, potentially diminishing the effectiveness of dynamic warp formation. Furthermore, for some divergence patterns, it is impossible to group threads in this manner [6].

***Divergence Avoidance Through Software Transformation***: Besides hardware-based techniques such as those discussed above, software-based techniques for avoiding divergence have also been proposed [3], [23]. These techniques aim to avoid divergence by re-mapping memory or transforming memory references to reorganize the layout of data, improve memory coalescing, and reduce control and memory divergence. Like software-based herding, these software-based techniques have the benefit of being immediately deployable on real GPUs.

***Best-effort Computing for Parallel Applications***: Related works on best effort computing for a GPU version of semantic document search [2] and parallel implementations of recognition and mining applications [8] also recognize and exploit the forgiving nature of certain parallel algorithms to increase performance by relaxing correctness. The authors observe acceptable results for target applications after relaxing data dependencies and dropping computations. They relax data dependencies between iterations of a function call to give the parallel processor or GPU more work to do in parallel. They also monitor the usefulness of iteratively computed data during runtime and drop computations between iterations when the observed usefulness of the computed data falls below a threshold. The idea of exploiting the forgiving nature of parallel applications to improve performance is common to our work. We, however, propose a different set of optimizations that target GPU and SIMD-specific inefficiencies.

***Reliability - Performance Tradeoffs for Data-parallel Workloads***: A similar work demonstrates that reliability can be traded for increased efficiency in certain data-parallel workloads [22]. The authors argue that data-parallel physics animations require perceptibility, rather than strict numerical correctness. As such, they propose reducing floating point precision to improve energy efficiency. Exploiting error tolerance enables higher performance for the same cost, as they can afford to put more, reduced-precision FPUs on a chip, as opposed to fewer, high-precision FPUs.

***Outcome Tolerant Branches***: A work on Y-branches [19] showed that taking the wrong direction for some branches may still bring the processor to a correct architectural state. By toggling the outcome of random branches in a program, the authors observed that for 40% of dynamic branches, taking either branch direction leads to a valid architectural state. The percentage was higher (around 50%) when allowing a mispredicted branch to continue executing on the wrong path. The authors note that outcome tolerance (the property of a branch indicating that the program output does not depend on the chosen branch direction) is a result of redundancies inserted by the programmer or compiler, as well as partially dead code.

Branch herding may benefit from outcome-tolerance in branches, but does not require it. In general, we rely on the error resilient nature of applications to tolerate inexactness in some thread computations. We also evaluate the effect on program outputs of allowing some branches to take incorrect control paths, observing acceptable outputs for many applications. In our experiments, we never observed a program crash as a result of herding branches onto the same branch path.

## IX. Conclusion

In this paper, we demonstrate that significant potential performance benefits are possible from safely and efficiently reducing control and memory divergence for GPU applications that can tolerate errors. We propose two optimizations – branch herding and data herding – that eliminate control and memory divergence, respectively. To ensure safety when introducing control and memory errors, while targeting performance benefits and acceptable output quality, we propose a static analysis and compiler framework, a profiling framework, and hardware support for branch and data herding. Our software implementation of branch herding uses CUDA intrinsics and forces diverging threads to take the same direction at a branch as the majority of the threads. Our hardware implementation of branch herding uses majority logic to identify the branch direction all threads should take. Data herding is implemented in coalescing hardware by identifying the most popular memory block (that the majority of loads map to) and mapping all loads from different threads in the warp to that block. Our software implementation of branch herding on NVIDIA GeForce GTX 480 improves performance by up to 34% (13%, on average) for a suite of NVIDIA CUDA SDK and Parboil [16] benchmarks. Our hardware implementation of branch herding improves performance by up to 55% (30%, on average). Data herding improves performance by up to 32% (25%, on average). For this level of performance benefits, observed output quality degradation is minimal for several applications that exhibit error tolerance.

## References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163 –174, 2009.

[2] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi. Best-effort semantic document search on GPUs. In *GPGPU*, pages 86–93, 2010.

[3] S. Che, J. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *SC*, pages 13:1–13:11, 2011.

[4] B. Coon. *United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture*. NVIDIA, 2008.

[5] W. Fung and T. Aamodt. Thread block compaction for efficient SIMT control flow. In *HPCA*, pages 25–36, 2011.

[6] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, pages 407–420, 2007.

[7] Khronos Group. *OpenCL*, 2010.

[8] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS*, pages 1–12, 2009.

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

[10] Microsoft. *GPGPU Computing Horizons*, 2010.

[11] V. Narasiman, M. Shebanow, C. Lee, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO*, pages 308–317, 2011.

[12] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*, 2009.

[13] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[14] NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.0*, 2010.

[15] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for SIMD cores. In *SC*, pages 22:1–22:11, 2009.

[16] The IMPACT Research Group. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[17] University of Illinois. clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[18] G. Varatkar and N. Shanbhag. Energy-efficient motion estimation using error-tolerance. In *ISLPED*, pages 113–118, 2006.

[19] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *PACT*, pages 56–, 2003.

[20] Wikipedia. Mandelbrot set, 2011. http://en.wikipedia.org/wiki/Mandelbrot_set.

[21] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235 –246, 2010.

[22] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *MICRO*, pages 394 –406, 2007.

[23] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380, 2011.

PLACE PHOTO HERE

**Rakesh Kumar (M'07)** is an Assistant Professor in the Electrical and Computer Engineering Department at the University of Illinois at Urbana Champaign. He received a B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology (IIT), Kharagpur in 2001 and a Ph.D. degree in Computer Engineering from the University of California, San Diego in September 2006. Prior to moving to Champaign in 2007, he was a visiting researcher with Microsoft Research at Redmond. His research interests include reliable and low power computing. His past research on heterogeneous multi-core architecture and conjoined-core architectures has directly influenced processor products and roadmaps from several companies. His current research interests are in error resilient computer systems and low power computer architectures for emerging workloads. His research has been recognized by several awards, including Best Paper Awards (CASES 2011, SRC TECHCON 2011), Best Paper Award Nominations (HPCA 2012), ARO Young Investigator Award, Arnold O Beckman Research Award, FAA Creative Research Award, UCSD CSE Best Dissertation Award, and an IBM PhD Fellowship. Other recognitions include Keynote Invitations (WRA 2011, WDSN 2011, LPonTR 2011, etc.), Invited/Plenary lectures at conferences and workshops (CASES 2011, ISLPED 2010, IOLTS, 2010, etc.), and Invited Guest Editorships (IEEE Transactions on Multimedia, IEEE Embedded Systems Letters, etc.). He has served as a Chair of two Workshops in the area of robust computing and multi-core computing (SELSE 2011 and dasCMP 2005-2008). When not doing computing research, he enjoys studying interactions between technology, policy, and society.

PLACE PHOTO HERE

**John Sartori (S'03)** is a Ph.D. candidate in Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He received a B.S. degree in Electrical Engineering, Computer Science, and Mathematics from the University of North Dakota and a M.S. degree in ECE from UIUC. His thesis research explores design, architecture, and compiler techniques for stochastic processors. His research has been recognized by several awards, including a Best Paper Award (CASES 2011), a Best Paper Award Nomination (HPCA 2012), and an Intel Computer Engineering Fellowship.