

GPUCheck: Detecting CUDA Thread Divergence with Static Analysis

Taylor Lloyd
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
tjlloyd@ualberta.ca

Karim Ali
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
karim.ali@ualberta.ca

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
Department of Computing Science
jamaral@ualberta.ca

January 25, 2019

Abstract

Graphics Processing Units (GPUs) have been widely used to accelerate the performance of programs. However, such performance gains can be significantly degraded by irregular data accesses and by control-flow divergence. Both of these performance issues arise only in the presence of thread-divergent expressions—expressions that evaluate to different values for different threads. The effect on performance depends on the number of threads that are left idle by control divergence, or on the number of distinct memory accesses that are required to satisfy all the memory references from a single warp of threads. Even experienced programmers may fail to identify control divergence or non-coalesceable memory accesses. This paper introduces GPUCheck: a static analysis tool that detects branch divergence and non-coalesceable memory accesses in GPU programs. GPUCheck relies on a static dataflow analysis to find thread-dependent expressions and on a novel symbolic analysis to determine when such expressions could lead to performance issues. GPUCheck supports programmers by informing them, at compile time without executing any GPU code, of potential performance problems.

```

1 den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr)) ;
2 c = 1.0 / (1.0+den) ;
3 if (c < 0){temp_result[ty][tx] = 0;}
4 else if (c > 1) {temp_result[ty][tx] = 1;}
5 else {temp_result[ty][tx] = c;}

```

Figure 1: Original diffusion coefficient calculation in `srad`.

```

6 den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr)) ;
7 c = min(max(1.0f / (1.0f+den),0.0f),1.0f) ;
8 temp_result[ty][tx] = c;

```

Figure 2: Modified diffusion coefficient calculation in `srad`.

0.1 Introduction

Modern supercomputers rely extensively on GPUs to deliver the computing power that propels them to the Top500 list [1]. For example, Titan [24], the current top supercomputer in the USA, utilizes nearly 20,000 GPUs. The successful use of GPU acceleration in some areas of computation justifies this shift toward GPU-based high-performance computing. GPUs improve, by orders of magnitude, the performance of neural networks [5], fluid dynamics [20], and molecular dynamics [14]. Power-envelope restrictions for the next generation of supercomputers will increase reliance on GPUs even further, due to their superior performance-per-watt ratio. The upcoming Summit supercomputer [23] aims to deliver over 200 petaflops in a power envelope of 10 MW, a five-fold increase over the performance of Titan while consuming only 10% more energy.

Despite the increased reliance on GPUs, the structured style of parallel processing that GPUs require makes their performance sensitive to two problems: (1) *branch divergence* [13], in which adjacent threads exhibit different control-flow behaviour causing hardware stalls, and (2) *non-coalesced memory accesses* [7], in which adjacent threads access disparate memory addresses, overloading the underlying memory system with requests. Experienced GPU programmers have a number of options to produce code that avoids both problems. For example, rephrasing control-flow operations as arithmetic operations can reduce branch divergence. Similarly, for non-coalesced memory accesses, programmers can either change which thread performs which memory access, restructure the underlying data structure, or load sections of data into a shared-memory buffer with different memory access characteristics. Nevertheless, even experienced programmers may unwittingly produce code that suffers from poor performance. Such issues often arise when an algorithm designed for a CPU is translated into a GPU equivalent without substantial restructuring. GPU algorithms typically must be restructured around the constraints of the underlying platform, separating work that can be parallelized across threads, and often recomputing intermediate products.

GPU threads are grouped into *warps* of 32, which execute in lock-step. When thread behaviour within a warp diverges, branch divergence and non-coalescable accesses become possible. We define instructions where threads within a warp may evaluate an instruction differently to be *thread-dependent* instructions.

The example in Figure 1, taken from the `srad` benchmark in the Rodinia suite [8], illustrates branch divergence. To ensure that the coefficient `c` stays within the range 0 to 1, the code tests the value of `c` and makes the appropriate correction. However, the value `qsqr` is derived from data calculated from each thread’s location in a 2D grid. Therefore, the value of `c` is different for each thread, causing potential control-flow divergence: a different group of threads may execute each of the statements in Lines 3–4, leading to three execution cycles, each requires a memory access. Figure 2 shows an alternative implementation, where testing the value of `c` is a computation of `min` and `max` operations that are available as instructions in NVidia GPUs. In the transformed code, all threads execute the memory-access statement in Line 7 simultaneously. By modifying the example to avoid the divergence, performance can be substantially improved.

Static analysis can detect divergence performance issues in many common cases. Branch divergence and non-coalescable memory accesses can both only occur when the behaviour of threads within a warp differ due to thread-dependent behaviour. And thread-dependent behaviour can only occur when an instruction depends, either directly or indirectly, on the thread index or a value produced by an atomic operation. Therefore, thread-dependence detection can be defined as a taint-analysis problem [2] that identifies whether variables and memory locations *may* be *tainted* by an atomic operation or by the thread index. A branch whose conditional expression is tainted may exhibit divergent behaviour. However, reporting all tainted branches would lead to many false positives. Thus, the taint analysis must be combined with a pruning algorithm to provide helpful feedback to programmers.

This paper addresses the following research problem: *given a program containing portions that are designated for execution on Nvidia GPUs, detect and report branch divergences and non-coalescable memory accesses*. To achieve that, we present GPUCheck, a static analysis tool, built on top of a novel static analysis framework, that identifies and reports the locations in a given program source code that are likely to exhibit poor GPU performance. The paper also describes a prototype implementation of GPUCheck built on top of the Clang/LLVM compiler [18]. The input to this prototype is an intermediate representation of the program with mappings to the original source code. GPUCheck uses a novel inter-procedural context-sensitive Arithmetic Control Form (ACF), a representation for static address range analysis and conditional expression analysis also described in this paper. The prototype identified performance issues in 17 well-known Rodinia benchmarks [8], including the example in Figure 1.

Unlike GPUCheck, previous attempts to analyze GPU workloads observe dynamic behaviour on either a simulator [3] or on physical GPUs [7]. These analyses produce precise results, but come at a cost. To benefit from dynamic analysis, the application developer must have access to GPUs with similar characteristics to the target system, or experience substantial overheads to simulate GPU execution. In addition, a developer must have access to representative data sets to examine relevant code paths. In contrast, GPUCheck analyzes a given program statically, and reasons about all possible execution paths through the program. GPUCheck tends to be substantially faster than dynamic techniques, because it does not need to execute a program to perform its analysis. On the Rodinia benchmark suite, the median completion time for GPUCheck’s analysis is 0.21 second, in comparison with 36.36 seconds required by Nvidia’s own dynamic profiler `nvprof` [22].

In summary, this paper makes the following contributions:

1. Arithmetic Control Form (ACF), a representation for statically computing differences between expressions computed by various threads.
2. GPUCheck, a static analysis tool that identifies common sources of performance degradation in GPU programs.

0.2 CUDA Programming, Divergence, and Coalescing

GPUCheck applies traditional static analysis techniques to the context of GPU execution. In this section, we provide background on the GPU execution model, and common causes of performance degradation.

0.2.1 CUDA Programming Model

Parallelism in the CUDA programming model is of a SIMT (Single Instruction Multiple Thread) form. In CUDA, a program is divided into host code and a series of *kernels*. The code for each kernel describes the execution of a single thread, but the programming model assumes that many threads will execute that same kernel code in parallel. Threads are grouped into blocks, and a number of blocks is executed simultaneously. The number of threads per block and the number of blocks are collectively referred to as a grid, and must be specified each time a kernel is executed. CUDA kernels have access to device variables such as `threadIdx` and `blockIdx` to specify thread-specific behaviour. Divergent behaviour often occurs when a conditional statement or the calculation of a memory-access address depends on the value of these variables.

```

9 kernel_compute_cost(int num, int dim, long x
    , Point *p, float *coord_d, ...) {
10
11     const int tid;
12     tid = blockDim.x * bid + threadIdx.x;
13     ...
14     float x_cost = d_dist(tid, x, num, dim,
        coord_d) * p[tid].weight;

```

(a) Noncoalescable memory accesses

```

15 kernel_compute_cost(int num, int dim, long x
    , float *p_x, float *p_y, float *p_z,
        float *p_weight, float *coord_d, ...) {
16     const int tid;
17     tid = blockDim.x * bid + threadIdx.x;
18     ...
19     float x_cost = d_dist(tid, x, num, dim,
        coord_d) * p_weight[tid];

```

(b) Perfectly coalesced accesses

Figure 3: Extract from the Rodinia benchmark `streamcluster`.

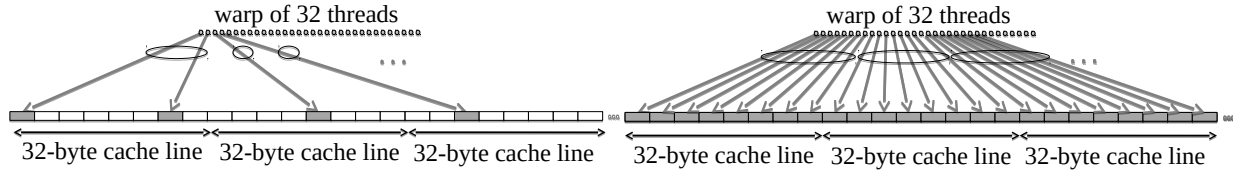


Figure 4: The access pattern in the Rodinia benchmark `streamcluster` before (left) and after the code transformation (right).

0.2.2 Branch Divergence Leads to Resource Underutilization

Modern NVidia GPUs issue each instruction to a *warp* of 32 threads simultaneously: All threads in a warp must execute the same instruction each cycle. *Branch divergence* occurs when a conditional branch instruction is issued to a warp and the condition evaluates to a different value for some threads within the warp. In this case, the hardware must first execute the code in the taken path—leaving idle the threads that evaluate the condition to false—and subsequently it must execute the not-taken path while leaving idle the threads that evaluate the condition to true. All threads may continue executing when the control flow re-converges at the join point in the flow graph. Such underutilization of processing resources reduces the performance of GPUs.

Branch divergence is problematic because stalled threads are still assigned registers and execution slots, preventing other threads from being started to perform useful work.

0.2.3 Global Memory Coalescing is Key for Performance

A warp of threads issues instructions simultaneously, causing as many as 32 simultaneous memory access requests to be issued in one execution cycle. However, the bandwidth available to access a GPU global memory subsystem is limited. Thus, the hardware in a GPU is able to *coalesce* (merge) adjacent or overlapping requests that originated on the same cycle by threads within a warp into fewer requests, each accessing more data. Once a memory access request is issued, no threads in a warp can continue executing until all of the threads have been serviced. Therefore, coalescing multiple memory accesses into fewer requests dramatically improves throughput. However, the hardware is only able to coalesce requests if the threads in a warp are accessing adjacent or overlapping locations in memory, and if the range of accessed addresses is aligned to a cache line boundary. If the execution of a statement leads threads to access memory locations that do not fit within an aligned range of memory addresses, then multiple memory requests are necessary. Therefore, the execution time is longer than in the case where all the accesses for the warp are coalesced into a single request. Each global memory request requires hundreds of cycles to be completed. Thus, GPU programs should be structured to avoid non-coalescable memory accesses.

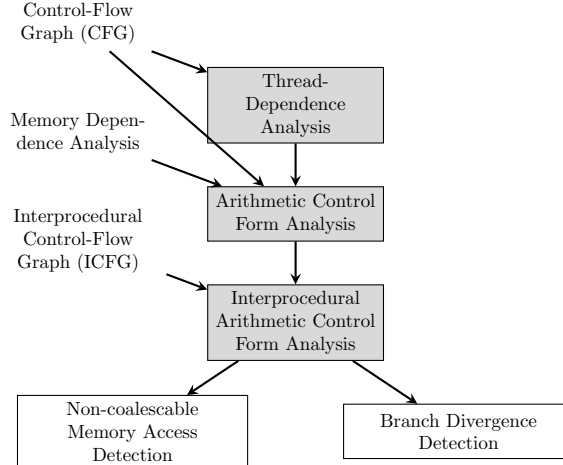


Figure 5: GPUCheck Analysis Workflow

Figure 3a shows an extract from the Rodinia benchmark `streamcluster`, which illustrates a non-coalescable memory access, as detected by GPUCheck. The code computes a weighted Euclidean distance between points, using the array of structures `p[tid].weight` in the computation. The data structure `Point` occupies six 32-bit words as shown on the left of Figure 4, and there is a gap between the `weight` field, shown in grey, of subsequent structures. As a consequence, the execution of the code in Line 14 of Figure 3a leads to the coalescing pattern shown on the left of Figure 4 where each memory transaction, represented by the ellipses, fetches few `weight` fields. To fix this performance issue, the code should use separate arrays for each member in the struct as shown in Figure 3b. The right of Figure 4 shows the placement in memory of the array `p_weight`. In this pattern, the accesses are coalesced into fewer memory transactions, each fetching the maximum number of `weight` fields allowed by the memory bandwidth.

0.3 A New Static Analysis Engine

The core of GPUCheck is a series of static analyses, which combine to determine how execution behaviour differs between threads, and the effects that those differences may have on performance. In this section, we provide an overview of the two core analyses in GPUCheck: thread-dependence analysis and Arithmetic Control Form. Figure 5 shows the analyses in GPUCheck and their dependences. Figure 5 also includes the dependent performance detection analyses detailed in Section 0.4.

0.3.1 IFDS-based Taint Analysis over SSA to Detect Thread Dependence

An expression is *thread dependent* if it generates different values depending on the thread executing it at runtime. To detect potential thread-dependent expressions statically, GPUCheck identifies sources of thread dependence and propagates them through their uses in the program. As a motivating example, consider the pseudocode in Figure 6 where `a` is thread dependent, because it contains the value of the thread identifier `threadIdx`. Similarly, `c` is thread dependent, because it derives from `a`. However, is `b` thread dependent? The answer depends on the analyzed program point. For instance, at the point immediately after Line 23, the value of `b` is a constant 1, and at the point immediately after Line 25, `b` is a constant 0. However, `b` as used at Line 28 is thread dependent, because the value of `b` depends on the `if` statement whose condition contains the thread-dependent value `c`. Given this intuition, we define thread dependence as follows:

Definition 0.3.1. An expression is *thread dependent* at a program point p if it may evaluate to different values by different threads at p .

```

20 a = threadIdx;
21 c = a % 2;
22 if(c) {
23   b = 1;
24 } else {
25   b = 0;
26 }
27 Arr[a] = 0;
28 Arr[b] = 0;

```

Figure 6: An example illustrating thread dependence.

If a given expression is not a source of thread dependence, then it can only be thread dependent through data-flow dependencies or control-flow dependencies. Detecting thread dependence through data-flow dependencies is straightforward, because an operand of the expression must also be thread dependent. However, calculating thread dependence caused by control-flow dependencies requires additional analysis. Control-flow thread dependence occurs when a conditional branch evaluates a thread-dependent expression that may evaluate to different values depending on the execution path.

Divergence—both for branches and for memory access addresses—originates from source statements and is propagated to other expressions in the program through control-flow and data-flow dependencies. For each expression in the program, the thread-dependence analysis in GPUCheck computes a boolean value that determines if the expression is thread dependent. GPUCheck regards source statements as *tainting* statements and employs a taint analysis to determine which expressions are tainted by thread-dependent calculations. In a GPU program, a statement may be a source of thread dependence due to a variety of operations. For instance, a statement that reads hardware values, or performs an atomic operation such as compare-and-swap, may compute values that may be unique to each thread. GPUCheck is concerned primarily about branch divergence and memory coalescing, which are behaviours between threads within a warp. Therefore, it only considers thread identifiers and atomic memory operations as sources of thread dependence. Although other operations can result in thread dependence (SM identifier, block identifier, warp identifier), these operations all return the same value across a warp, and therefore can never cause intra-warp thread dependence.

GPUCheck uses an Interprocedural Finite Distributive Subset (IFDS) [25] taint analysis over a static single assignment (SSA) intermediate representation to propagate thread-dependence information through the IFDS supergraph representation of the program. Given the set D of all SSA expressions in the program, the dataflow problem consists in determining which subset of D is thread dependent at each point in the program. The thread-dependence property is propagated through distributive dataflow functions that can be decomposed into micro-functions. Each micro-function $f_x(y)$ expresses the propagation of the expression $x \in D$ through the statement that defines the expression $y \in D$. The special function *gen* creates and propagates the thread-dependence property regardless of the prior state, the function *prop* propagates the input thread-dependence state to output, and the function *kill* propagates no thread-dependence property, regardless of input. The use of an expression x in a program statement does not change the thread-dependence property of x . Therefore, for each statement that defines an expression $d \in D$, the thread-dependence property of each expression x where $x \neq d$ remains unchanged: $f_x(d) = \text{prop}$. If the statement that defines d reads the thread identifier, then d is a source of thread dependence and the micro-function $f_d(d) = \text{gen}$. Otherwise, $f_d(d)$ depends on the thread-dependence property of the operands used in the statement that defines d and on the control-flow dependencies of the statement that defines d . To compute $f_d(d)$, let $O(y)$ be the set of expressions that are used in the statement that defines expression y (the operands). Let $CDG(y)$ be the set of basic blocks that the statement that defines y is control dependent on, as determined by a control-dependence graph (CDG) [12]. Then $CDG(y)$ is the set of basic blocks that determines whether or not y executes. Let $cd(y)$ be the set of expressions used as conditions for branches exiting each basic block

```

29 int readBounded(int* a) {
30     int tx = threadIdx.x;
31     if(tx > 256)
32         tx = 256;
33     int *addr = a + tx;
34     return *addr;
35 }

```

```

36     tx0 = threadIdx.x
37     p1 = tx > 256
38 p1? tx1 = 256
39     tx =  $\psi$ (tx0, p1?tx1)
40     tmp = 4 * tx
41     addr = a + tmp
42     return addr

```

(a) A bounded array access.

(b) If-converted ψ -SSA form for the code in (a).Figure 7: An example illustrating if-conversion in ψ -SSA, which serves as inspiration for our ACF analysis.

in $CDG(y)$. The value of $f_d(d)$ is then given by:

$$f_d(d) = \bigcup_{o \in O(d)} (f_o(d) \cup \bigcup_{c \in cd(o) \setminus cd(d)} f_c(d)) \quad (1)$$

Intuitively, the first half of the equation, $\bigcup_{o \in O(d)} f_o(d)$, captures thread dependence over data dependencies by combining the thread dependence of all operands of d . The control dependences for an operand $o \in O(d)$ of the definition d (denoted $cd(o)$) are the decisions that lead to the execution of the expression that defines o . To capture relevant control dependences, the analysis computes the conditions that are required to reach an operand of d , but not d itself, producing the set difference $cd(o) \setminus cd(d)$. Such conditions are control dependences, and are combined to produce $f_d(d)$.

0.3.2 Arithmetic Control Form (ACF)

The thread-dependence analysis determines which expressions in the program are thread-dependent. The intuition is that a conditional expression that is thread dependent is a potential source of control-flow divergence, and a memory-access expression that is thread dependent is a potential source of non-coalescable memory accesses. However, this intuition alone may lead to numerous false positives, i.e., GPUCheck would be signalling potential divergences that are not actual divergences. For memory accesses, we are interested in determining if the range of addresses accessed by all threads in a warp falls within a single cache line. To achieve that, we have designed the Arithmetic Control Form (ACF) analysis. Given a thread-dependent expression, ACF determines the difference between the value of this expression as computed by each thread.

The value computed by an expression depends not only on the flow of values through expressions, but also on the conditional statements in the code. Existing work in support of if-conversion in SSA form [19, 27] serves as an inspiration for ACF. [27] introduced ψ -nodes to represent the flow of SSA values through a segment of straight-line code in the presence of predicated execution. Intuitively, a ψ -node combines the results of multiple predicated instructions, unifying values in straight-line code in the same way ϕ -nodes unify values from differing basic blocks in traditional SSA. Figure 7a shows a simple bounded array indexing operation. The ψ -SSA form for this code is shown in Figure 7b after if-conversion. The transformed code is in single-assignment form and the if-statement conditional expression is stored in predicate register `p1`. The ψ -node thus uses predicates to select between multiple possible values.

In the ψ -SSA form, ψ -nodes are equivalent to the sum of all incoming values multiplied by the associated incoming predicate. ACF extends this notion by computing complex predicates through as much of the program’s control-flow as necessary to obtain an expression that precisely captures all possible execution traces. In other words, the ACF value for an expression is equal to a sum, where each summed element corresponds to a control-flow path and has a value equal to the expression as computed along that path, multiplied by the conditions required to execute that path.

For each execution of the code, a single predicate combination evaluates to 1 and all the other combinations evaluate to 0. Through this transformation, ACF produces a symbolic equation for each expression


```

43 void memcpy(char* tgt, char* src, size_t sz) {
44     int tx = threadIdx.x;
45     int dim = blockDim.x;
46     for(int i=0; i+tx<sz; i+=dim) {
47         char *tgtaddr = tgt + i + tx;
48         char *srcaddr = src + i + tx;
49         *tgtaddr = *srcaddr;
50     }
51 }

```

Figure 8: An example illustrating how ACF handles loops.

of interest. In essence, ACF is an alternative program representation, suited for analysis rather than actual execution. ACF represents the value generated by each expression as a tree of arithmetic operations, constants, and unknown values. For each expression of interest in a given CUDA kernel, ACF computes the differences between the expression as it is evaluated by each thread. Threads are computed by substituting constant thread identifiers, and simplification is performed by merging common predicates and eliminating subexpressions that are not thread dependent. In practice, most differences statically evaluate to a constant after simplification and thus can be used to determine if a tainted expression leads to either thread divergence or non-coalesced accesses. Consider the example code in Figure 7a, which implements a bounded array access. To compute the address accessed by each thread in Line 33, ACF analyzes all possible paths through the function. In this case, there are two paths, corresponding to the `if` case or the `else` case. Let v be a variable in the program. In ACF, the notation $[v]$ indicates that v is represented symbolically, and $ACF(v)$ is the ACF value for v . A subscript indicates that a reference is thread-dependent, and specifies the thread. The ACF value for `addr` on Line 33 is then defined as:

$$\begin{aligned}
 ACF_t(addr) = & \\
 & ([threadIdx.x_t] > 256) * ([a] + 4 * 256) + \\
 & ([threadIdx.x_t] \leq 256) * ([a] + 4 * [threadIdx.x_t])
 \end{aligned}$$

Whenever possible, ACF replaces variable references with their definitions. For example, $[threadIdx.x_t]$ is used instead of `tx` in the ACF representation of the code in Figure 7a. An unknown value, such as $[a]$ in Figure 7a, is represented symbolically. Given the ACF representation for the code in Figure 7a, a consuming analysis may query for the difference between the `addr` returned by threads 0 and 1: $ACF_1(addr) - ACF_0(addr)$. The thread-dependence analysis determines which symbolic references, such as $[a]$, are not thread dependent. Such references cancel out in the computation of differences as follows:

$$\begin{aligned}
 & ACF_1(addr) - ACF_0(addr) \\
 & = ((1 > 256) * ([a] + 4 * 256) + (1 \leq 256) * ([a] + 4 * 1)) \\
 & - ((0 > 256) * ([a] + 4 * 256) + (1 \leq 256) * ([a] + 4 * 0)) \\
 & = ([a] + 4) - [a] \\
 & = 4
 \end{aligned}$$

To reason about loops, ACF has to handle loop induction variables. Similar to unknown variables, ACF handles a loop induction variable `iv` symbolically, using the following notation: $ACF(iv) = [iv]$. For instance, Figure 8 shows a simple parallel implementation for `memcpy`. The query $ACF(srcaddr)$ is used to determine the value of `srcaddr`. To solve this query, ACF captures the common behaviour across all loop iterations, under the assumption that if values common within the loop are not thread dependent, they will often cancel when a difference is calculated. $ACF(srcaddr)$ is therefore calculated as follows: $ACF_t(srcaddr) = [src] + [i] + [threadIdx.x_t]$.

```

52  int main() {
53      int x = b(4);      58      int a(int i) {          62      int b(int j) {
54      int y = a(x);      59          int ar = i - 16;    63          int br = j + 8;
55      int z = b(y);      60          return ar;        64          return br;
56      return y;          61      }                    65      }
57  }

```

Figure 9: An example illustrating the need for Inter-procedural Arithmetic Control Form (IACF).

ACF treats the index variable $[i]$ symbolically. If both the initialization expression and the reinitialization expression for $[i]$ are thread independent, then the value of $[i]$ is also thread independent. Thus, when computing the difference between expressions involving $[i]$ for any loop iteration, the symbolic value $[i]$ disappears, resulting in either a constant distance between threads or a distance that depends either on the thread identifier or on other symbolic variables.

To calculate the ACF representation for the function call $c = f(\langle args \rangle)$ with the return expression ret , GPUCheck first calculates $ACF(ret)$, then replaces any arguments in $ACF(ret)$ with the actuals in $\langle args \rangle$. For the code example in Figure 9, $ACF(y)$ is calculated as follows:

$$\begin{aligned}
 ACF(y) &= ACF(a(x)) \\
 &= [i] - 16 \\
 &= ACF(x) - 16 \\
 ACF(x) &= ACF(b(4)) \\
 &= [j] + 8 \\
 &= 12 \\
 ACF(y) &= 12 - 16 = -4
 \end{aligned}$$

In this example, calls are resolved *down* the call stack. However, in many cases, branch divergence and memory coalescing analyses require upward call resolution, because the analysis is performed inside a nested function. For instance, in the example in Figure 9, what is $ACF(br)$? In ACF, the answer can only be $[j] + 8$. To provide more precise results, an inter-procedural ACF is needed.

0.3.3 Inter-procedural Arithmetic Control Form (IACF)

To operate inter-procedurally, GPUCheck maps the actual arguments from a function call to the formal parameters in the function definition. This mapping may lead to multiple ACF representations for a given program expression—potentially one for each calling context.

Producing IACF requires an inter-procedural control-flow graph (ICFG). GPUCheck constructs a set of IACF representations iteratively by first calculating the intra-procedural ACF representation of each function. GPUCheck then inspects this representation for references to the function’s arguments. For each reference to a function argument, GPUCheck identifies all non-recursive call sites to the function, and substitutes the actual arguments for the formal parameters for each call site. This process continues until all arguments corresponding to non-recursive function calls have been replaced. Similar to loop induction variables, arguments to recursive calls remain symbolic references. Therefore, IACF generalizes over recursive paths similar to looping paths, sacrificing some precision for performance.

For the example in Figure 9, the best approximation that the intra-procedural analysis produces for the possible values of br in Line 63 is $ACF(br) = [j] + 8$. IACF produces a more precise result. IACF discovers that there is a set of two possible values for br , because there are two calls to the function $b()$, $b(4)$ in

Line 53 and $\mathbf{b}(y)$ in Line 55.

$$\begin{aligned} IACF(\mathbf{br}) &= \{[j] + 8\} \\ &= \{ACF(4) + 8, ACF(y) + 8\} \\ &= \{12, 4\} \end{aligned}$$

While IACF sets may grow arbitrarily large, GPU kernels tend to have small call graphs. In our experimental evaluation with the Rodinia benchmark suite, which is representative of typical GPU applications, we found that any computed IACF expression set has at most 6 expressions. Therefore, we believe that IACF performs adequately for GPU code. In applications where memory space or computational time is limited, ending IACF expansion when a set reaches a specified maximum size (similar to k-limiting [16]) allows for sacrificing precision to improve performance.

0.3.4 ACF for SSA Form

We have implemented a prototype for GPUCheck on top of the LLVM compiler infrastructure. Since the release of `gpucc` [31], Clang is able to compile CUDA programs to LLVM IR. Using `gpucc`, GPUCheck can operate on CUDA programs just like any other LLVM GPU language.

GPUCheck generates ACF on demand from LLVM’s SSA representation, which requires a memory-dependence analysis and a control-dependence graph [10]. Given an expression e , the computation of $ACF(e)$ requires the computation of the ACF for each of the operands of e . GPUCheck memoizes the ACF for these operands and makes them available when needed for future ACF computations. Once the ACF for all operands of e is determined, arithmetic operations can be trivially converted to ACF to compute $ACF(e)$.

Most SSA values deterministically compute a value from their operands, and so the equivalent ACF expression is simply the same operation computed on the ACF of their operands. However, ϕ -nodes merge values over multiple incoming control-flow paths by specifying a mapping of each predecessor basic block to a value. To calculate the ACF expression for a ϕ -node, GPUCheck first calculates a predicate for each predecessor basic block b that evaluates true iff the ϕ -node is immediately preceded at runtime by b . The ACF value for the ϕ -node is then simply the sum of each predicate multiplied by the associated definition.

Let $CDG(e)$ be the set of control dependences for an expression e , as determined by the program control-dependence graph. Let $cond(e)$ refer to the set of conditional expressions corresponding to each control dependence in $CDG(e)$ that results in the execution of e . Finally, let $operand(x)$ denote the expression operands to a ϕ -node x . To determine the value of a ϕ -node, the ACF representation for each operand expression is multiplied against the ACF representation of each conditional expression $cond(e)$ required to reach that operand (Equation 2).

$$ACF(\phi) = \sum_{i \in operand(\phi)} \left(\prod_{c \in cond(i)} ACF(c) \right) * ACF(i) \quad (2)$$

If a ϕ -node contains a cyclic reference (e.g., loop induction variables), its ACF representation remains symbolic (i.e., $ACF(\phi_{loop}) = [\phi_{loop}]$). Otherwise, for each operand, all conditions required to select that value are converted to ACF representation, and multiplied against the value.

To resolve dependences between operations with memory, the generation of ACF requires a pointer-aware memory-dependence analysis as presented by [15], that provides a set of dominating stores for each memory load where possible. The prototype implementation uses LLVM’s *MemDepAnalysis*. If a dominating store exists, the ACF representation uses the stored value to be used in place of the load instruction. Otherwise, the load becomes a symbolic reference.

The memory-dependence analysis propagates the effects of the statements of a procedure on memory — through an inter-procedural memory dependence analysis, on the local variables of the caller, and on global variables.

0.4 Detecting Divergent Behaviour

We have implemented the detection algorithms in GPUCheck using IACF expression sets. These algorithms calculate differences between IACF expressions evaluated by different threads in a warp to detect thread-dependent behaviour. IACF expressions typically contain many run-time references that the thread-divergence analysis have determined to be thread independent. In the calculation of the difference between two IACF expressions evaluated by different threads, thread-independent run-time references cancel out. In many cases, the result of the difference is a constant offset between the accesses in two threads that can be used for performance analysis.

0.4.1 Divergent-Branch Analysis

Divergent-branch analysis takes as input the set of thread-dependent conditional branches in the program under analysis, as discovered by the thread-dependence analysis. To improve the precision of GPUCheck, the analysis assumes a constant grid of 256 threads per block and 1 block per grid. Ideally, the actual runtime grid would be used, but the grid is defined in host code and therefore not available during device-code analysis. The use of a constant grid lead the analysis to report divergences that do not occur (false positives) and also to miss actual divergences (false negatives), because some CUDA code may assume the use of a particular grid geometry. Typical CUDA code queries hardware registers (`gridDim`, `blockDim`) and adapts to arbitrary geometry, but if the source code expects a particular grid geometry then GPUCheck may produce incorrect results. Given that the results produced by GPUCheck are used to raise warnings to developers, risking producing incorrect results in some cases to improve precision or the common cases is a reasonable tradeoff.

GPUCheck constructs IACF expression sets for each thread-dependent conditional branch. For each IACF expression $IACF_t(e)$, and for each thread t_x in a warp of threads $t_0 \dots t_{31}$, GPUCheck calculates the difference $IACF_{t_x}(e) - IACF_{t_0}(e)$. If a difference is non-zero, the warp of threads is branch divergent. If a difference is a non-constant ACF expression, then GPUCheck is unable to determine the divergence of a branch for that warp. GPUCheck calculates the divergence ratio for each branch over all warps as $\frac{\text{warps divergent}}{\text{warps total}}$. The result of this analysis is, for each thread-dependent branch instruction, a range $[d_{min}, d_{max}]$ where d_{min} is the minimum divergence ratio for the branch and d_{max} is the maximum divergence ratio. Thus, the range $[0,0]$ indicates that no branch divergence can ever occur, a $[1,1]$ indicates that all warps diverge, and a range $[0,1]$ indicates that no information could be determined statically.

0.4.2 Non-coalescable Memory Access Analysis

GPUCheck uses IACF to bound the number of requests required to fulfill a memory operation that accesses a thread-dependent address. The analysis in this paper only models accesses to global memory. Shared-memory accesses require different access patterns for coalescing to occur. Moreover, penalties for non-coalescable accesses are substantially lower in shared memory. Therefore, GPUCheck does not analyze shared-memory accesses.

GPUCheck uses an address-space analysis to identify memory operations on pointers that may point to global memory. Non-coalescable memory access analysis takes the set of thread-dependent addresses from loads and stores that may point to global memory as input. Similar to divergent-branch detection, the analysis assumes a constant grid to improve precision at the expense of possible unsound results. For each IACF expression, and for each thread x , the analysis calculates the difference $IACF_{t_x}(addr) - IACF_{t_0}(addr)$. The calculated differences are either constant, in which case they are collected in a set C , or non-constant, in which case they iterate a counter *nonconst*. There are two possible reasons for an IACF difference to not be constant: (1) the IACF analysis is imprecise, leaving constant but unknown values in the expression; or (2) the expression is dependent on run-time data that cannot be statically determined. For constant address differences, all accesses within a 256-byte range can be coalesced into a common request. Figure 10 presents the coalescing algorithm that GPUCheck uses to calculate the number of memory requests required for a given memory access, given a set of constant offsets C and a number of non-constant accesses *nonconst*. The

```

Function coalescedRequests(C, nonconst)
  requests={ };
  for c ∈ C do
    fit = false;
    for r ∈ requests do
      if c ≥ r.low && c ≤ r.high then
        | fit = true;
      else if c ≥ r.high - 256 && c ≤ r.high then
        | r.low = c;
        | fit = true;
      else if c ≤ r.low + 256 && c ≥ r.low then
        | r.high = c + 8;
        | fit = true;
      end
    if fit ≠ true then
      | requests.append( (low: c, high: c+8) );
    end
  end
  return (requests.size, requests.size + nonconst);

```

Figure 10: Coalescing algorithm in GPUCheck.

algorithm computes the number of required requests by greedily calculating the minimum number of 256-byte spans (from $r.low$ to $r.high$) that can serve all of the constant offsets in C . This calculation is based on the fact that complete coalescing only occurs when all the accesses are to the same 256-byte cache line. This coalescing algorithm considers only the size of requests, and cannot identify when additional requests are required because of unaligned addresses. This imprecision results from the canceling of symbolic values using ACF thread differences. GPUCheck cannot determine the offset of base pointers shared between threads, and therefore is unaware of access alignment. The result of this analysis, for each thread-dependent memory access, is a $[r_{min}, r_{max}]$ range where r_{min} is the minimum number of memory requests — at least 1, and r_{max} is the maximum number of memory requests — at most 32, required per warp. As full range $[1,32]$ indicates that no information is known statically. A range $[1,4]$ indicates that all accesses will be coalesced, and the range $[32,32]$ indicates that no coalescing is ever possible.

0.5 Fast Static Detection of Previously Unknown Divergence Issues Leads to Better Performance

The main goal of GPUCheck is to assist developers by detecting potential performance-limiting issues in GPU programs at compile-time so that they can be eliminated to improve performance. Ideally, GPUCheck would run every time code is compiled. Therefore, the analysis must be sufficiently fast to avoid interrupting development.

This section reports on a performance evaluation of the LLVM GPUCheck prototype. This evaluation uses the CUDA implementation of the Rodinia heterogeneous computing benchmarks [8], a benchmark suite that captures a representative sample of GPU computing tasks.

Prior to GPUCheck, developers could only detect GPU performance issues through dynamic profiling: executing programs against testing data, and recording characteristics of the execution. To facilitate profiling, NVIDIA provides `nvprof`, a dynamic profiler. Both `nvprof` and GPUCheck identify non-coalesced memory accesses and divergent branches. Additionally, `nvprof` detects a wide variety of other issues such as determining overall occupancy [22].

Table 1: Classification of thread-dependent branches in Rodinia benchmarks. Non-divergent indicates that no warps ever diverge, partial divergence indicates that some warps definitely diverge, and Total divergence indicates that all warps always diverge.

Benchmark	Non-divergent	Partial Divergence	Total Divergence	Unknown
backprop	1	1	0	2
bfs	0	0	0	6
b+tree	0	2	0	12
gaussian	1	0	0	0
heartwall	16	5	16	53
hotspot3D	2	0	0	2
hotspot	9	2	0	5
huffman	3	3	0	7
lavaMD	0	3	0	3
leukocyte	2	2	1	10
lud	0	5	0	0
myocyte	0	2	0	25
nn	0	0	0	1
nw	0	5	0	1
pathfinder	2	2	0	6
srad	8	8	0	2
streamcluster	0	0	0	2
Total	44	40	17	137

To provide a comparison, we profiled each Rodinia benchmark using `nvprof`, collecting all `--analysis-metrics`, which includes branch divergence counters for each branch in the source code, dynamic memory coalescing counters for each memory access in source code, as well as occupancy information and the dynamic instruction mix. Through debug information, assembly-level performance counters are associated with lines in the original source. If `nvprof` reports a line as divergent, or a memory operation requiring at least four requests, then that line is deemed either divergent or non-coalescable. An execution of GPUCheck converts the software source into LLVM IR and links all device modules. GPUCheck analyses all branch instructions and memory access operations. GPUCheck uses debug source information to report offending source lines. The benchmarks are compiled using NVCC from CUDA 8 at optimization level `-O2` with `lineinfo` included. The benchmark generated code is executed and profiled on an NVidia Pascal Titan X, using a host system running on an Intel i7-4770 with 32GB of RAM running CentOS 6. Seventeen benchmarks from the Rodinia suite are analyzed — at the time of evaluation, the CUDA implementations of `cfid`, `hybridsort`, `kmeans`, `mummergpu`, and `dwt2d` could not be compiled with Clang/LLVM because of its incomplete CUDA support, and thus these benchmarks were not included in the evaluation. This evaluation addresses the following questions:

- **Q1:** Does GPUCheck identify problems in common benchmarks?
- **Q2:** Does GPUCheck provide similar results to dynamic profiling?
- **Q3:** How many problems identified by GPUCheck reflect real performance opportunities?
- **Q4:** Is GPUCheck performant enough to be used during active development?

0.5.1 Q1: Does GPUCheck identify problems in common benchmarks?

We divide thread-dependent branches into four categories, based on the static range of possible divergence: (1) Non-divergent branches never diverge within a warp; (2) Partial-divergence branches diverge in at least

Table 2: Classification of thread-dependent memory accesses in Rodinia benchmarks. Coalesced indicates 4 or fewer memory requests required. Non-coalesced indicates that more than 4 requests are always required, and unknown indicates that between 1 and 32 requests are required per warp.

Benchmark	Coalesced	Non-coalesced	Unknown
backprop	1	0	17
bfs	10	0	4
b+tree	24	0	1
gaussian	1	0	6
heartwall	63	0	18
hotspot3D	10	0	11
hotspot	3	0	0
huffman	15	0	13
lavaMD	9	0	0
leukocyte	0	0	3
lud	11	0	0
myocyte	0	3	16
nn	5	0	0
nw	2	0	8
pathfinder	3	0	0
srad	17	0	8
streamcluster	6	0	3
Total	180	3	108

1 warp; (3) Total-divergent branches diverge in every warp; (4) Unknown represents branches with static ranges spanning multiple categories, i.e. the result of the analysis is the range [0-0.5] indicating that the branch may or may not be partially divergent.

This categorization is pragmatic. Total divergence is a separate category because it can cause dramatic performance degradation. Table 1 shows that GPUCheck identifies 17 total branch divergences and 40 partial branch divergences across 14 of the 17 analyzed benchmarks. When thread-dependence comes solely from thread identifiers and there are 256 threads per block the results produced by GPUCheck are sound. Therefore, these 17 branches always diverge and result in a reduction of GPU performance. In addition to these known divergences, GPUCheck also identifies thread-dependent branches with statically unknown divergence behaviour. These branches are likely thread-dependent as well because data-dependent behaviour is the primary reason for statically unknown branch behaviour. GPUCheck identified between 57 and 194 divergent branches, depending on the runtime behaviour of the unknown branches, and was able to determine that 44 thread-dependent branches are definitely not divergent.

Each thread-dependent memory access is categorized as coalesced ($r_{max} \leq 4$), non-coalesced ($r_{min} > 4$), or unknown if GPUCheck’s computed range contains 4 requests/warp ($r_{min} \leq 4 \leq r_{max}$). In the Nvidia Titan X Pascal, a memory operation for a full warp requires 4 cycles to be dispatched. Thus, if a memory access requires up to 4 requests/warp, that access does not increase the memory access latency. Table 2 shows the memory coalescing behaviour as determined by GPUCheck. Across all the analyzed Rodinia benchmarks, only three memory accesses were provably non-coalescable, all due to an extremely common pattern. The code in Figure 11 to demonstrates this pattern. On Line 72, the array access is multiplied by a parameter, *Size*, to create a 2-dimensional array access pattern. Unfortunately, the value of *Size* is not known at compile time. On one hand, the occurrence of run-time values in benchmarks leads to GPUCheck reporting nearly all non-coalescable accesses as unknown — only 3 of the 111 possible non-coalescable accesses are reported as non-coalescable. On the other hand, GPUCheck is able to prove that 180 thread-dependent memory accesses as definitely coalescable.

Although static analysis cannot precisely determine most non-coalescable accesses, the presence of un-

known access patterns indicates a data-dependent address expression that is likely to be a non-coalescable access. In the remainder of the evaluation, unknown divergence and unknown access patterns are reported as problems identified by GPUCheck.

GPUCheck identifies between 59 and 304 divergent branches and non-coalescable accesses in the popular Rodinia benchmark suite.

0.5.2 Q2: Does GPUCheck provide similar results to dynamic profiling?

Table 3 shows divergent branches and non-coalescable memory accesses as found by both the NVidia dynamic profiler and by GPUCheck. Differences in methodology cause GPUCheck and `nvprof` to report different but overlapping sets of divergency issues: GPUCheck cannot detect uncoalesced accesses caused by cache misalignment, and reports potential branch divergence as a ratio of divergent warps per branch. By contrast, `nvprof` measures branch divergence dynamically, so branches only a portion of threads in a warp are executing may report different values. Due to these methodological differences, GPUCheck results are not a strict superset of `nvprof` results. Further, `nvprof` reports divergence and coalescing issues using line numbers. Thus this evaluation compares the line numbers where GPUCheck reports at least one issue with the ones reported by `nvprof`. However, lines may contain multiple memory accesses, and thus the numbers reported may not match those in Tables 1 and 2.

Ideally, both GPUCheck and `nvprof` would identify all possible issues in all benchmarks, however both GPUCheck and `nvprof` have limitations. For a given instruction, `nvprof` aggregates across all executions. For instance, a memory operation that generates 32 requests per warp (i.e., fully non-coalescable) 10% of the time would be reported by `nvprof` as requiring 3 requests per access, while GPUCheck would correctly identify the non-coalescable access. A similar strategy is used by `nvprof` for divergent branches. When inspecting branches, GPUCheck and `nvprof` use different thresholds to identify divergence. GPUCheck statically analyzes the branch condition per warp, reporting the branch as divergent when more than 40% of warps are divergent, or if the thread-dependent branch condition is also based on a value that is not known at compile time. By contrast, `nvprof` uses an unpublished threshold of all runtime executions of a warp, causing occasional disagreement on whether a particular branch is divergent. For example, `nvprof` does not report any non-coalescable accesses for the *gaussian* benchmark, where an 8.8% performance gain can be achieved after fixing three non-coalescable accesses that GPUCheck reports (see Q3). Conversely, in this evaluation non-coalescable accesses detected with `nvprof`, but not by GPUCheck, never in exceeds five requests/warp, which leads to only marginal performance degradation.

































As a static analysis tool, GPUCheck analyzes kernels and code paths that may never be executed at runtime. This feature allows GPUCheck to identify performance issues throughout the code, while `nvprof` is limited to code paths actually exercised by the provided workloads. Given that it is based in actual execution of the program, `nvprof` can provide additional insight into the impact of various performance issues by determining the percentage of runtime affected by each issue.

GPUCheck and `nvprof` are complementary: GPUCheck identifies additional performance opportunities ignored by `nvprof`, and can be used during development

0.5.3 Q3: How many problems identified by GPUCheck reflect real performance opportunities?

For an analysis like GPUCheck that may miss issues and may report issues that are not real, a measurement of precision and recall would be desirable. However, computing precision and recall requires a ground truth that is not available for the issues addressed by GPUCheck. As the case studies reported in this section will evidence, the decision on whether an issue is real or not is highly correlated with the skills of the CUDA programmer that examines the issue reported. The best that we can do to address the precision/recall question is to compare the issues reported by GPUCheck to those reported by `nvprof`,

Table 3: Divergency issues found in the Rodinia Benchmark Suite. Black indicates an issue found only by GPUCheck. White indicates an issue found only by nvprof. Grey indicates an issue found by both. The adjacent fractions are the number of issues found by GPUCheck, over the total issues found.

Benchmark	Divergent Branches	Non-coalescable Accesses
backprop	 $\frac{3}{3}$	 $\frac{7}{8}$
bfs	 $\frac{4}{4}$	 $\frac{4}{4}$
b+tree	 $\frac{11}{11}$	 $\frac{1}{1}$
gaussian	 $\frac{3}{3}$	 $\frac{3}{3}$
heartwall	 $\frac{74}{76}$	 $\frac{17}{22}$
hotspot	 $\frac{4}{4}$	 $\frac{0}{2}$
hotspot3D	 $\frac{2}{2}$	 $\frac{5}{5}$
huffman	 $\frac{9}{16}$	 $\frac{12}{12}$
lavaMD	 $\frac{3}{3}$	 $\frac{0}{6}$
leukocyte	 $\frac{13}{14}$	 $\frac{3}{4}$
lud	 $\frac{5}{5}$	$\frac{0}{0}$
myocyte	 $\frac{14}{14}$	 $\frac{14}{14}$
nn	 $\frac{1}{1}$	 $\frac{0}{1}$
nw	 $\frac{6}{6}$	 $\frac{2}{2}$
pathfinder	 $\frac{5}{5}$	$\frac{0}{0}$
srad	 $\frac{12}{12}$	 $\frac{8}{8}$
streamcluster	 $\frac{2}{2}$	 $\frac{2}{4}$

```

66 __global__ void Fan1 ( ... ) {
67     int xidx =
68         blockIdx.x * blockDim.x + threadIdx.x;
69     if(xidx >= Size-1-t) return;
70     int off = Size*(t+1)+t;
71     m_cuda[Size*xidx+off] =
72         a_cuda[Size*xidx+off] / a_cuda[Size*t+t];
73 }
74 }

75 __global__ void Fan2( ... ) {
76     ...
77     if(yidx >= Size-1-t) return;
78     if(xidx >= Size-t) return;
79     ...
80     a_cuda[Size*xidx+yidx+off] -=
81         m_cuda[Size*xidx+off] *
82         a_cuda[Size*t+yidx+t];
83     if(yidx == 0) {
84         b_cuda[xidx+1+t] -=
85             m_cuda[Size*xidx+yidx+off] *
86             b_cuda[t];
87 }

```

Figure 11: Original `gaussian` kernel functions (edited for clarity).

the Nvidia dynamic profiler. Using case studies, we establish that issues raised by GPUCheck represent real performance issues. We investigate and repair four benchmarks (`gaussian`, `lavaMD`, `nw`, and `srad`) to demonstrate the performance gains that may be obtained by acting on GPUCheck reports. For each of these benchmarks, we fixed any issues detected by GPUCheck, and executed both our modified and the original code three times each on our experimental machine, interleaving executions. We measure speedups over mean kernel execution time, ignoring the negligible variance.

Gaussian Elimination (`gaussian`)

The `gaussian` benchmark in the Rodinia benchmark suite uses two kernels, `Fan1` and `Fan2`, to solve for variables in a linear system of arbitrary size. Figure 11 shows simplified excerpts of both kernels. GPUCheck identifies non-coalesced memory accesses in `Fan1` at Line 72 and in `Fan2` at Line 81 and Line 85, both missed by `nvprof`. `nvprof` identifies divergent branches at boundary checks in `Fan1` and `Fan2`, because the grid geometry of threads and blocks is not an exact match for the problem size. Better tuning of the grid to match the problem size may improve performance but we deliberately exclude such branch divergences as performance impact is likely to be limited. Instead, we concentrate on the 6 non-coalescable (unknown) accesses picked up only by GPUCheck.

In `Fan1`, the first element in each row of the matrix is initialized by a thread. The access stride by adjacent threads is the width of a row, because the matrix is stored in row-major format. Changing the indexing of the matrix to column-major format allows these accesses to be coalesced. With this change, `Fan1` initializes adjacent elements in each thread. When changing storage schemas, it is often necessary to consider how other accesses will be affected. `Fan1` and `Fan2` operate on the same matrix, thus the access pattern in `Fan2` is also changed by this transformation.

Fortunately, `Fan2` is a two-dimensional CUDA kernel. Reversing the matrix storage schema in `Fan2` is equivalent to exchanging the x and y thread dimensions in the kernel. We do not present the modified code here, because the modifications are straightforward changes to the array indexing operations by swapping `threadIdx.x` and `threadIdx.y`.

After applying the modifications based on the output of GPUCheck, `Fan1` runs 11.5% faster and `Fan2` runs 5.9% faster than the original code. Overall, the `gaussian` kernels complete 8.8% faster. The 6 non-coalescable memory accesses reported by GPUCheck are not detected by `nvprof`, and were previously undetectable by automated means. All 6 represent real non-coalescable accesses.

```

88 int wtx = threadIdx.x;
89 ...
90 while(wtx<NUMBER_PAR_PER_BOX){
91     rA_shared[wtx] = rA[wtx];
92     wtx = wtx + NUMBER_THREADS;
93 }

```

Figure 12: Extract from `lavaMD` demonstrating buffering in shared memory.

LavaMD (`lavaMD`)

LavaMD is an N-body computation for simulating molecular dynamics interactions. It was originally produced by the Lawrence Livermore National Laboratory, and is derived from the `ddcMD` application, which performs the same computation sequentially. The `lavaMD` benchmark’s kernel makes heavy use of shared memory buffers, copying memory in tiles for efficient memory access patterns as shown in Figure 12. However, the buffer size `NUMBER_PAR_PER_BOX` is carried over from previous CPU implementations, and set to 100 in the original benchmark. By contrast, this kernel is launched with 128 threads, leaving nearly a quarter of threads idle through these loops.

Both GPUCheck and `nvprof` identify the loop at Line 90 in Figure 12 as a source of branch divergence, repeating in 4 locations. We fixed this issue by modifying the shared memory buffers and number of threads per block to 96 elements. By using a power of two for the shared memory sizes, the allocations divide evenly into the device’s shared memory space, improving utilization. Additionally, reducing the number of idle threads allows more throughput per thread. With only these values changed, the `lavaMD` kernel executes 25.6% faster. We are unsure why `nvprof` missed 1 of There are 6 additional non-coalescable memory accesses detected by `nvprof`, caused by poor memory alignment while copying data to shared memory. GPUCheck missed these accesses, because it currently cannot identify alignment issues.

Needleman-Wunsch (`nw`)

Needleman-Wunsch is an algorithm from the field of bioinformatics used to align proteins and nucleotides. The implementation in the Rodinia benchmark suite executes as a tiled matrix computation with a halo. Figure 13 shows the halo initialization from the original kernels. GPUCheck identifies the northwest corner setup on Line 94 as a point of divergence, and both GPUCheck and `nvprof` identify the west column setup on Line 98 as a non-coalesced access.

The divergence can be resolved by allowing all threads within the first warp to setup the corner, providing a small improvement by avoiding divergence. In this case, the non-coalesced access cannot be fixed, because threads must read across both rows and columns of the matrix. Performance can still be improved, because this non-coalesced access is tightly synchronized. The synchronization points above and below this access hurt performance, because all threads in the block must wait while the access completes. A cursory inspection shows that all threads write to different elements of `temp`, and thus all of the halo setup can be synchronized together, eliminating extra synchronization in Lines 97 and 99. By providing other warps with work to perform while resolving each non-coalesced access, performance can still be improved.

GPUCheck finds an additional 5 partial divergences due to a triangular loop, which cannot be easily removed, but are nonetheless true divergent branches.

Applying both transformations makes the `nw` kernels execute 5.5% faster.

Speckle Reducing Anisotropic Diffusion (`srad`)

The `srad` algorithm attempts to remove correlated noise, or *speckles* from imagery without destroying underlying information. This algorithm has applications in various imaging technologies such as ultrasound or radar.

```

94 if (tx == 0)
95     temp[tx][0] = matrix_cuda[index_nw];
96 ...
97 __syncthreads();
98 temp[tx + 1][0] = matrix_cuda[index_w + cols * tx];
99 __syncthreads();
100 temp[0][tx + 1] = matrix_cuda[index_n];
101 __syncthreads();

```

Figure 13: Original halo computation in `nw` kernels.

Table 4: Execution time for GPUCheck vs dynamic profiling. Branches and accesses show the number of instructions requiring ACF analysis over all instructions analyzed.

Benchmark	Branches	Accesses	GPUCheck Time (s)	Profiling Time (s)
backprop	4/13	18/145	0.14	2.38
bfs	6/12	14/67	0.12	24.38
b+tree	14/33	25/214	0.30	4.73
gaussian	4/9	7/69	0.09	5.02
heartwall	90/258	81/1364	5.53	281.23
hotspot	16/48	3/194	0.26	1.86
hotspot3D	4/16	21/195	0.43	105.55
huffman	13/41	28/277	0.28	41.26
lavaMD	6/29	9/162	0.15	21.16
leukocyte	15/66	3/332	0.21	57.79
lud	5/77	11/272	0.29	36.36
myocyte	27/4216	19/7499	1.25	1880.55
nn	1/2	5/32	0.09	1.37
nw	6/50	10/280	0.19	201.21
pathfinder	10/37	3/111	0.14	6.73
srad	18/58	25/540	0.53	8.04
streamcluster	2/9	9/82	0.13	2080.28

One step of the `srad` computation calculates a coefficient of diffusion c , a value between 0 and 1, based on a stencil of nearby values. GPUCheck identifies branch divergence in this code, pointing to repeated conditional memory operations as shown in Figure 1 on Lines 3, 4, and 5. Figure 2 presents our modified code that uses arithmetic `min` and `max`, which have single-instruction implementations on the GPU, to remove the conditional behaviour entirely. Our modified `srad` kernel executes 30.8% faster. Dynamic profiling through `nvprof` fails at detecting this issue, which is representative of the performance gains from repairing branch divergence problems using GPUCheck. The benchmark `srad` contains a second kernel, in which GPUCheck found no issues, thus the overall `srad` kernel execution time is improved by 15.7%.

Fixing the performance issues as reported by GPUCheck in 4 sample Rodinia benchmarks led to improving GPU kernel performance by 5.5–25.6% in terms of execution time. No reported issues were false positives, though not all could be repaired without major algorithmic reworking.

0.5.4 Q4: Is GPUCheck performant enough to be used during active development?

GPUCheck is intended to be used actively during the development of GPU algorithms and applications. Therefore, the performance of the analysis is important. The analysis time reported for each benchmark consists of the time for the branch divergence, memory coalescing, and supporting analyses. These times are representative if the application is typically compiled using Clang/LLVM, allowing GPUCheck to raise warnings during compilation.

Table 4 shows the execution time required for each benchmark. GPUCheck completes its analysis in 90 milliseconds to 5.5 seconds for each benchmark, with an arithmetic mean analysis time of 596 milliseconds, typically within Nielsen’s recommended threshold for interactive user interfaces [21]. Therefore, GPUCheck can be integrated seamlessly into existing development environments, without adding much overhead to the normal workflow of developers. By contrast, `nvprof` requires an arithmetic mean of 4.7 minutes to collect the required profiling information (min: 1.37 seconds, max: 34.67 minutes, arithmetic mean: 4.667 minutes), with a total of 1.4 hours for all benchmarks. The notable outlier is `heartwall`, where GPUCheck executes for over 5s — but for which `nvprof` needs 4 minutes. The `heartwall` benchmark program contains a large number of thread-dependent code paths, leading to ACF expressions being generated for 171 expressions. By comparison, only 46 expressions are inspected in `myocyte`. Most interestingly, GPUCheck analysis time and `nvprof` profiling time are uncorrelated. GPUCheck scales with thread-dependent code size, while `nvprof` scales with execution time.

GPUCheck is substantially faster than `nvprof`, with mean benchmark analysis time under one second.

0.6 The ACF Analysis Distinguishes GPUCheck from Alternative Approaches to Divergency Detection

Automated tools exist for most programming languages to detect common programming mistakes [17, 6, 9]. Such tools make use of static analysis techniques to verify correct program behaviour. This section intently narrows the discussion to highlight prior work on detecting branch divergence and non-coalescable memory accesses for GPU kernels.

[26] propose a conservative divergent-branch analysis over affine expressions on thread identifiers, now implemented within LLVM. An extension to their work generalizes to divergent values throughout a GPU kernel. In contrast, GPUCheck does not require affine expressions, and can solve for non-linear conditions. The ACF framework used by GPUCheck allows for non-linear relationships and conditional, inter-procedural data flow to be accurately and precisely modelled.

Transforming non-coalescable memory accesses into coalesced accesses has been an active subject of research. [30] show that given perfect knowledge of memory access layouts, minimizing non-coalesced accesses is an NP-hard problem. However, the authors do not consider what analysis might be used to generate such layouts. GPUCheck is well-suited to perform such a task. Affine polyhedral models have also been used to transform memory accesses [4, 29]. [28] have recently extended the polyhedral model to include limited non-affine expressions. The ACF framework, and therefore GPUCheck, can compute arbitrary non-affine expressions, and maintain precision through arbitrary function calls. ACF is also being implemented in an industrial-strength compiler to guide GPU loop transformations for memory coalescing [Citation omitted for blind review].

[11] present an approach that requires dynamic analysis to identify non-coalesced accesses using memory traces. Their work generates high-accuracy results, but requires the analysed kernels to be executed, suffering from the same issues as any dynamic profiler. On the other hand, GPUCheck runs its analysis without even a GPU present, by calculating inter-thread behaviour. Integrating GPUCheck in a development framework and reporting potential performance issues at compilation time, allows programmers to fix problems while

their implementation is fresh in their minds, and there is no need to wait until the actual execution of a GPU program.

0.7 GPUCheck Efficiently Uncovers Performance Issues Early

GPUs enable power-efficient parallel processing, and are becoming increasingly popular to accelerate scientific applications. However, GPUs are subject to two well-known performance problems: branch divergence and non-coalescable accesses. In this paper, we introduce GPUCheck, a static analysis tool that reasons about GPU program behaviour. Compared to dynamic profiling, GPUCheck has the following advantages:

1. ACF is a framework for reasoning about parallel behaviour in the presence of unknown values and arbitrary control-flow. GPUCheck leverages these abilities to uncover GPU performance issues at compile-time.
2. GPUCheck’s analysis time scales with the code size, while profiling time scales with actual program execution time. The difference in times can be very significant, because GPU programs tend to be highly parallel.
3. Since GPUCheck uses static analysis, it needs no test data and takes into consideration all possible executions through the code. In contrast, profiling can detect only issues that actually occur in the test data.
4. GPUCheck does not require a physical GPU to identify problems, because it does not execute GPU code. When there is competition for the use of GPU computation, GPUCheck frees up more GPU time for useful work.

GPUCheck detects branch divergences and non-coalescable memory accesses on 17 programs from the Rodinia benchmark suite. Fixing those issues improves performance, in terms of execution time, by 5.5–25.6%. An LLVM-based prototype demonstrates that GPUCheck is complementary to dynamic profiling, and represents a strong foundation on which future analysis of parallel systems can be built.

Bibliography

- [1] Top500 supercomputers. <https://www.top500.org/>, 2017. Accessed: 2017-06-01.
- [2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.
- [3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, 2009.
- [4] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *International conference on Supercomputing (ICSC)*, pages 225–234, 2008.
- [5] Michael Beyeler, Nicolas Oros, Nikil Dutt, and Jeffrey L Krichmar. A GPU-accelerated cortical neural network model for visually guided robot navigation. volume 72, pages 75–87, 2015.
- [6] Oliver Burn. Checkstyle. <https://checkstyle.sourceforge.net>, 2003. Accessed: 2017-04-27.
- [7] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.
- [8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.
- [9] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 673–674. ACM, 2006.
- [10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
- [11] Naznin Fauzia, Louis-Noël Pouchet, and P Sadayappan. Characterizing and enhancing global memory data coalescing on GPUs. In *Code Generation and Optimization (CGO)*, pages 12–22. IEEE Computer Society, 2015.
- [12] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.
- [13] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2007.

- [14] Andreas W Gotz, Mark J Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. generalized born. *Journal of chemical theory and computation*, 8(5):1542–1555, 2012.
- [15] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. volume 24, pages 28–40. ACM, 1989.
- [16] Neil D Jones. Program flow analysis: Theory and applications. 1981.
- [17] Ted Kremenek. Finding software bugs with the Clang static analyzer. *California: Apple Inc*, 2008.
- [18] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
- [19] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. In *ACM SIGMICRO*, volume 23, pages 45–54, 1992.
- [20] L Mangani, E Casartelli, Giulio Romanelli, Magnus Fischer, A Gadda, and P Mantegazza. A GPU-accelerated compressible RANS solver for fluid-structure interaction simulations in turbomachinery. In *ASME Turbo Expo 2016*, pages V07BT34A022–V07BT34A022. American Society of Mechanical Engineers, 2016.
- [21] Jakob Nielsen. *Usability Engineering*. Elsevier, 1994.
- [22] NVidia. Profiler user’s guide. <https://docs.nvidia.com/cuda/profiler-users-guide/>, 2016. Accessed: 2016-11-29.
- [23] Oak Ridge National Laboratory. Summit. <https://www.olcf.ornl.gov/summit/>, 2016. Accessed: 2016-11-27.
- [24] Oak Ridge National Laboratory. Titan Cray XK7. <https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/>, 2016. Accessed: 2016-11-27.
- [25] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of programming languages (PoPL)*, pages 49–61, 1995.
- [26] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence analysis. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 35, page 13, 2013.
- [27] Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 172–181, 2001.
- [28] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *International Symposium on Code Generation and Optimization (CGO)*, page 185, 2014.
- [29] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.
- [30] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Principles and practice of parallel programming (PPoPP)*, volume 48, pages 57–68, 2013.
- [31] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. gpucc: An open-source GPGPU compiler. In *International Symposium on Code Generation and Optimization (CGO)*, pages 105–116, 2016.