Flexible Software Profiling of GPU Architectures

Mark Stephenson[†] Siva Kumar Sastry Hari[†] Yunsup Lee[‡] Eiman Ebrahimi[†]
Daniel R. Johnson[†] David Nellans[†] Mike O'Connor[†*] Stephen W. Keckler[†*]
[†]*NVIDIA,* [‡]*University of California, Berkeley, and* [*]*The University of Texas at Austin*
{mstephenson,shari,eebrahimi,djohnson,dnellans,moconnor,skeckler}@nvidia.com, yunsup@cs.berkeley.edu

Abstract

To aid application characterization and architecture design space exploration, researchers and engineers have developed a wide range of tools for CPUs, including simulators, profilers, and binary instrumentation tools. With the advent of GPU computing, GPU manufacturers have developed similar tools leveraging hardware profiling and debugging hooks. To date, these tools are largely limited by the fixed menu of options provided by the tool developer and do not offer the user the flexibility to observe or act on events not in the menu. This paper presents SASSI (NVIDIA assembly code "SASS" Instrumentor), a low-level assembly-language instrumentation tool for GPUs. Like CPU binary instrumentation tools, SASSI allows a user to specify instructions at which to inject user-provided instrumentation code. These facilities allow strategic placement of counters and code into GPU assembly code to collect user-directed, fine-grained statistics at hardware speeds. SASSI instrumentation is inherently parallel, leveraging the concurrency of the underlying hardware. In addition to the details of SASSI, this paper provides four case studies that show how SASSI can be used to characterize applications and explore the architecture design space along the dimensions of instruction control flow, memory systems, value similarity, and resilience.

1. Introduction

Computer architects have developed and employed a wide range of tools for investigating new concepts and design alternatives. In the CPU world, these tools have included simulators, profilers, binary instrumentation tools, and instruction sampling tools. These tools provide different features and capabilities to the architect and incur different design-time and runtime costs. For example, simulators provide the most control over architecture under investigation and are necessary for many types of detailed studies. On the other hand, simulators are time-consuming to develop, are slow to run, and are often difficult to connect to the latest software toolchains and applications. Binary rewriting tools like Pin [21] allow a user to instrument a program to be run on existing hardware, enabling an architect insight into an application's behavior and how it uses the architecture. Such tools have been used in a wide variety of architecture investigations including, deterministic replay architectures [26], memory access scheduling [25], on-chip network architectures [24], and cache architecture evaluation [18]. They have even been used as the foundation for multicore architecture simulators [23]. In addition, these types of tools have been used in a wide range of application characterization and software analysis research.

With the advent of GPU computing, GPU manufacturers have developed profiling and debugging tools, similar in nature to their CPU counterparts. Tools such as NVIDIA's NSight or Visual Profiler use performance counters and lightweight, targeted binary instrumentation to profile various aspects of program execution [34, 35]. These tools have the advantage that they are easy to use, and run on hardware, at hardware speeds. Unfortunately for computer architects and compiler writers, the production-quality profilers are not flexible enough to perform novel studies: one must choose what aspects of program execution to measure from a menu of pre-selected metrics. They also cannot be used as a base on which to build other architecture evaluation tools.

As one solution to the shortcoming described above, the architecture community has turned toward simulators, such as GPGPU-Sim to analyze programs and guide architecture development [1]. Simulators are immensely flexible, and allow architects to measure fine-grained details of execution. The major disadvantage of simulators is their relatively slow simulation rates. This forces researchers and architects to use trimmed-down input data sets so that their experiments finish in a reasonable amount of time. Of course, application profiles can be highly input-dependent, and there is no guarantee that simulation-sized inputs are representative of real workloads.

This paper presents a new GPU tool called SASSI for use in application characterization and architecture studies. Unlike the current suite of GPU profiling tools, which are tailor-made for a particular task or small set of tasks, SASSI is a versatile instrumentation framework that enables software-based, selective instrumentation of GPU applications. SASSI allows a user to select specific instructions or instruction types at which to inject user-level instrumentation code. Because SASSI is built into NVIDIA's production machine-code-generating

ISCA'15, June 13-17, 2015, Portland, OR USA

DOI: http://dx.doi.org/10.1145/2749469.2750375

compiler, and runs as its last pass, injection of instrumentation code does not affect compile-time code generation optimizations. Further SASSI is highly portable and gracefully handles the latest versions of CUDA, and can be extended to handle OpenGL and DirectX 11 shaders. The tool allows us to collect results across multiple generations of NVIDIA architectures including Fermi, Kepler, and Maxwell.

As with the production-quality profiling tools, selective instrumentation allows hardware-rate analysis, yet, as the case studies we demonstrate confirm, our approach is flexible enough to measure many interesting and novel aspects of execution. Because selective instrumentation is far faster than simulation, users can easily collect data based on real-world execution environments and application data sets.

Because GPU architectures follow a different programming paradigm than traditional CPU architectures, the instrumentation code that SASSI injects contains constructs and constraints that may be unfamiliar even to expert CPU programmers. Also unlike their CPU counterparts, GPU instrumentation tools must cope with staggering register requirements (combined with modest spill memory), and they must operate in a truly heterogeneous environment (*i.e.*, the instrumented device code must work in tandem with the host system).

In the remainder of this paper, we first discuss background information on GPU architecture and software stacks. Then we describe the details of SASSI, focusing on the key challenges inherent with GPU instrumentation, along with their solutions. Finally, we demonstrate the tool's usefulness for application profiling and architectural design space exploration by presenting four varied case studies (in Sections 5–8) that investigate control flow, memory systems, value similarity, and resilience.

2. Background

This section provides background on basic GPU architecture terminology and the NVIDIA GPU compilation flow. While SASSI is prototyped using NVIDIA's technology, the ideas presented in this paper are not specific to NVIDIA's architectures, tools, or flows, and can be similarly applied to other compiler backends and GPUs.

2.1. GPU Architecture Terminology

GPU programming models allow the creation of thousands of threads that each execute the same code. Threads are grouped into 32-element vectors called *warps* to improve efficiency. The threads in each warp execute in a SIMT (*single instruction, multiple thread*) fashion, all fetching from a single Program Counter (PC) in the absence of control flow. Many warps are then assigned to execute concurrently on a single GPU core, or *streaming multiprocessor* (SM) in NVIDIA's terminology. A GPU consists of multiple such SM building blocks along with a memory hierarchy including SM-local scratchpad memories and L1 caches, a shared L2 cache, and multiple memory controllers. Different GPUs deploy differing numbers

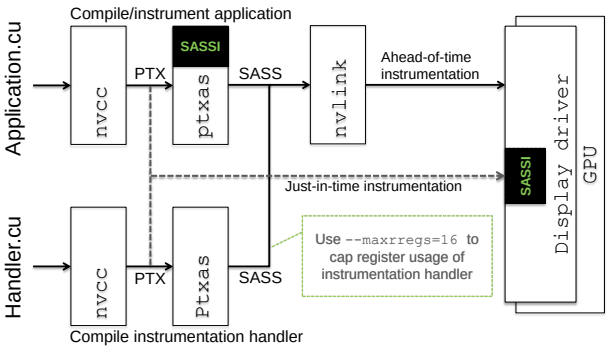


Figure 1: SASSI's instrumentation flow.

of SMs. Further details of GPU application execution, core, and memory architecture are explained in the case studies of Sections 5–8.

2.2. GPU Software Stack

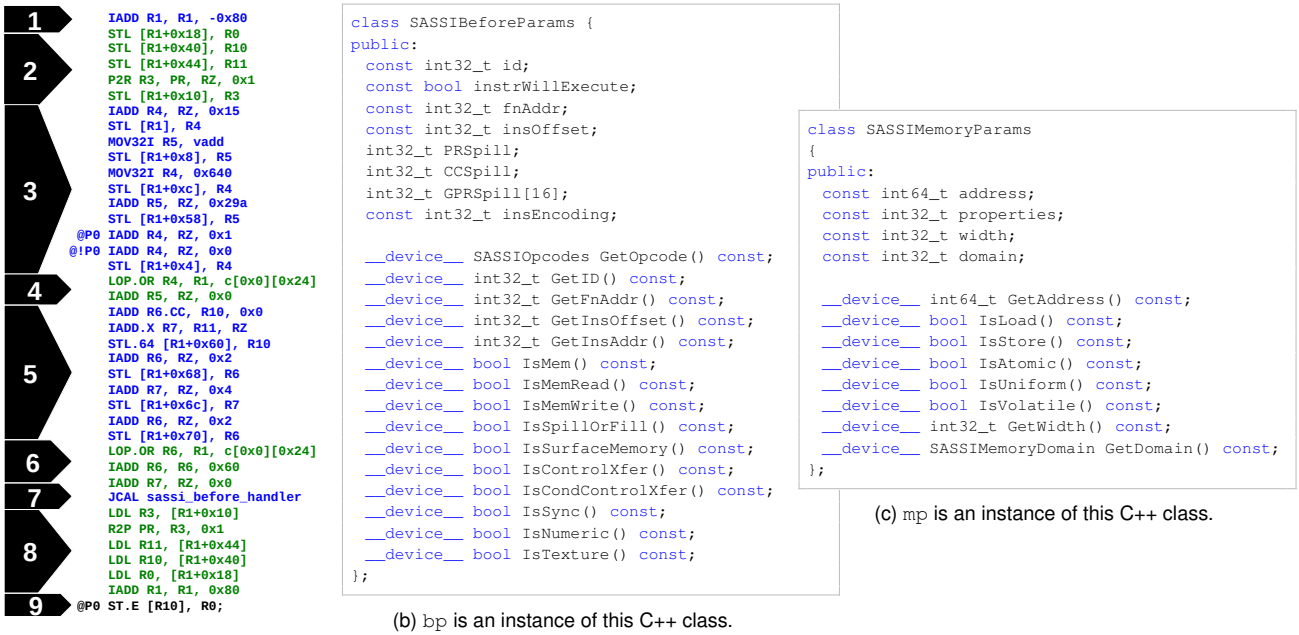
Historically, NVIDIA has referred to units of code that run on the GPU as *shaders*. There are several broad categories of shaders, including DirectX shaders, OpenGL shaders, and compute shaders (*e.g.*, CUDA kernels). A *front-end* compiler can be used to simplify the task of writing a shader. For example, for compute shaders, a user can write parallel programs using high-level programming languages such as CUDA [32] or OpenCL [39], and use a front-end compiler, such as NVIDIA's NVVM, to generate intermediate code in a virtual ISA called parallel thread execution (PTX).

PTX exposes the GPU as a data-parallel computing device by providing a stable programming model and instruction set for general purpose parallel programming, but PTX does not run directly on the GPU. A *backend* compiler optimizes and translates PTX instructions into machine code that can run on the device. For compute shaders, the backend compiler can be invoked in two ways: (1) NVIDIA supports ahead-of-time compilation of compute kernels via a PTX assembler (`ptxas`), and (2) a JIT-time compiler in the *display driver* can compile a PTX representation of the kernel if it is available in the binary.

Compute shaders adhere to a well-defined *Application Binary Interface* or ABI, which defines different properties of the interface between a caller and a callee. Examples include what registers are caller-saved vs. callee-saved, what registers are used to pass parameters, and how many can be passed in registers before resorting to passing parameters on the stack. In particular, this paper focuses on the ABI between on-device (GPU) callers and callees.

3. SASSI

This section describes SASSI, our backend compiler-based instrumentation tool. SASSI stands for SASS Instrumentor, where SASS is NVIDIA's name for its native ISA. We explain where SASSI sits in the compiler flow, describe SASSI injection and instrumentation code, and discuss how the instrumentation interoperates with host (CPU) code.



(a) Instrumented code.

(b) `bp` is an instance of this C++ class.

(c) `mp` is an instance of this C++ class.

Figure 2: SASSI instrumentation. (a) The instruction at ❾ is the original store instruction. The other instructions are the code that SASSI has inserted to construct an ABI-compliant function call. The sequence does the following: ❶ Stack allocates two objects, `bp` and `mp`, instances of `SASSIBeforeParams` and `SASSIMemoryParams`. The class definitions of `SASSIBeforeParams` and `SASSIMemoryParams` are shown in (b) and (c), respectively. ❷ Saves live registers `R0`, `R10`, and `R11` to the `bp.GPRSpill` array, and saves the live predicate registers to `bp.PRSpill`. ❸ Initializes member variables of `bp`, including `instrWillExecute` (which is true iff the instruction will execute), `fnAddress` and `insOffset` (which can be used to compute the instruction's PC), and `insEncoding` (which includes the instruction's opcode and other static properties). ❹ Passes a generic 64-bit pointer to `bp` as an argument to `sassi_before_handler` in registers `R4` and `R5` per NVIDIA's compute ABI. ❺ Initializes member variables of `mp`, including `address` (which contains the memory operation's effective address), `width` (which is the width of the data in bytes), `properties` (which contains static properties of the operation, *e.g.*, whether it reads memory, writes memory, is atomic, etc.). ❻ Passes a generic 64-bit pointer to `mp` in `R6` and `R7` per NVIDIA's compute ABI. ❼ Performs the call to `sassi_before_handler`. ❽ Restores live registers, and reclaims the allocated stack space. ❾ Executes the original store instruction.

3.1. SASSI Tool Flow

Figure 1 shows the compiler tool flow that includes the SASSI instrumentation process. Shaders are first compiled to an intermediate representation by a *front-end* compiler. Before they can run on the GPU, however, the *backend* compiler must read the intermediate representation and generate SASS. For compute shaders, the backend compiler is in two places: in the PTX assembler `ptxas`, and in the driver.

SASSI is implemented as the final compiler pass in `ptxas`, and as such it does not disrupt the perceived final instruction schedule or register usage. Furthermore as part of `ptxas`, SASSI is capable of instrumenting programs written in languages that target PTX, which includes CUDA and OpenCL. Apart from the injected instrumentation code, the original SASS code ordering remains unaffected. With the SASSI prototype we use `nvlink` to link the instrumented applications with user-level instrumentation handlers. SASSI could also be embedded in the driver to JIT compile PTX inputs, as shown by dotted lines in Figure 1.

SASSI must be instructed *where* to insert instrumentation, and *what* instrumentation to insert. Currently SASSI supports inserting instrumentation *before* any and all SASS instructions. Certain classes of instructions can be targeted for instrumentation: control transfer instructions, memory operations, call instructions, instructions that read registers, and instructions that write registers. SASSI also supports inserting instrumentation *after* all instructions other than branches and jumps. Though not used in any of the examples in this paper, SASSI supports instrumenting basic block headers as well as kernel entries and exits. As a practical consideration, the *where* and the *what* to instrument are specified via `ptxas` command-line arguments.

3.2. SASSI Instrumentation

For each instrumentation site, SASSI will insert a CUDA ABI-compliant function call to a user-defined instrumentation handler. However, SASSI must be told *what* information to pass to the instrumentation handler(s). We can currently extract and pass to an instrumentation handler, the following information for each site: memory addresses touched, registers written and read (including their values), conditional branch information, and register liveness information.

```
/// [memory, extended memory, controlxfer, sync, ...
/// numeric, texture, total executed]
__device__ unsigned long long dynamic_instr_counts[7];

/// SASSI can be instructed to insert calls to this handler
/// before every SASS instruction.
__device__ void sassi_before_handler(SASSIBeforeParams *bp,
                                     SASSIMemoryParams *mp) {
  if (bp->IsMem()) {
    atomicAdd(dynamic_instr_counts + 0, 1LL);
    if (mp->GetWidth() > 4 /*bytes*/)
      atomicAdd(dynamic_instr_counts + 1, 1LL);
  }
  if (bp->IsControlXfer()) atomicAdd(dynamic_instr_counts + 2, 1LL);
  if (bp->IsSync()) atomicAdd(dynamic_instr_counts + 3, 1LL);
  if (bp->IsNumeric()) atomicAdd(dynamic_instr_counts + 4, 1LL);
  if (bp->IsTexture()) atomicAdd(dynamic_instr_counts + 5, 1LL);
  atomicAdd(dynamic_instr_counts + 6, 1LL);
}
```

Figure 3: A trivial example instrumentation handler. SASSI can be instructed to insert a function call to this handler before all instructions.

Figure 2(a) shows the result of one memory operation being instrumented by SASSI. The instruction at (9) is the original memory instruction, and all prior instructions are SASSI-inserted instrumentation code. In this example, the user has directed SASSI to insert instrumentation before all memory operations (the *where*), and for each instrumentation site to extract memory-specific details (such as the address touched) about the memory operation (the *what*). SASSI creates a sequence of instructions that is an ABI-compliant call to a user-defined instrumentation handler. The caption of Figure 2 provides a detailed explanation of the sequence. SASSI creates extra space on a thread's stack in this example to store parameters that will be passed to the instrumentation handler.

While generating an ABI-compliant function call incurs more instruction overhead than directly in-lining the instrumentation code, maintaining the ABI has two main benefits. First, the user is able to write handlers in straight CUDA code. They do not need to understand SASS, PTX, or even details of the target architecture. Second, our approach is portable; the same handler can be used for Fermi, Kepler, and Maxwell devices, which are significantly different architectures.

Figure 3 shows a pedagogical instrumentation handler, `sassi_before_handler`, from the setup shown in Figure 2. The handler takes two parameters, pointers to instances of `SASSIBeforeParams` (`bp`) and `SASSIMemoryParams` (`mp`), respectively, and uses them to categorize instructions into six overlapping categories. Note that a single SASS instruction can simultaneously belong to more than one of these categories. As Figure 2(b) shows, the C++ object of class `SASSIBeforeParams` contains methods that allow a handler to query for basic properties of the instruction, including the instruction's address, whether it uses texture memory, and whether it alters control flow. This example uses several of `bp`'s methods to categorize the instruction, and it uses CUDA's `atomicAdd` function to record instances of each category. Additionally, the handler uses the `mp` object to determine the width in bytes of a memory operation's data. We can easily instruct SASSI to inject calls to this function before all SASS instructions, the mechanics of which we describe later.

One challenging aspect of GPU instrumentation is the sheer number of registers that may have to be spilled and filled in the worst case to create an ABI-compliant function call. Even though the compiler knows exactly which registers to spill, there are many instrumentation sites in typical GPU programs that require spilling 32 or more registers *per thread* if done naively. NVIDIA's Kepler and Maxwell architectures require spilling $\sim$128 registers per thread in the worst case. Compounding this problem, threads are executed in SIMT fashion; thus all the threads try to spill their live state to memory at the same time, creating serious bandwidth bottlenecks. To alleviate this issue, we impose a simple constraint on SASSI instrumentation handlers; handlers must use *at most* 16 registers, the minimum number of registers required per-thread by the CUDA ABI. This limit can trivially be enforced by using the well-known `-maxrregcount` flag of `nvcc` to cap the maximum number of registers used when compiling the instrumentation handler.

It is important to note that SASSI does not change the original SASS instructions in any way during instrumentation. Furthermore, the register limit of 16 that we impose on the instrumentation handler may increase the runtime overhead of instrumentation, but it will not reduce an instrumentation handler's functional utility. With handlers for which the compiler does not find an allocation that uses 16 or fewer registers, the compiler will simply insert register spill code.

3.3. Initialization and Finalization

Unlike CPU instrumentation, GPU instrumentation must coordinate with the host machine (CPU) to both initialize instrumentation counters, and to gather their values (and perhaps log them to a file). For CUDA, SASSI leverages the CUPTI library, which allows host-side code to register for callbacks when certain important CUDA events occur, such as kernel launches and exits [33]. In all of the case studies in this paper, we use CUPTI to initialize counters on kernel launch and copy counters off the device on kernel exits. On kernel launch, our CUPTI "kernel launch" callback function uses `cudaMemcpy` to initialize the on-device counters appropriately. On kernel exits, our CUPTI "kernel exit" callback function uses `cudaMemcpy` to collect (and possibly aggregate) the counters on the host side. Furthermore, `cudaMemcpy` serializes kernel invocations, preventing race conditions that might occur on the counters. This approach is excessive for the cases where we do not need per-kernel statistics. If instead we only wanted to measure whole-program statistics, we could simply initialize counters after the CUDA runtime is initialized, and copy counters off the device before the program exits (taking care to register callbacks for CUDA calls that reset device memory, which would clear the device-side counters). The appropriate initialization and finalization mechanisms can be chosen by the user depending on the specific use case.

4. Methodology

The case studies we present in this paper are meant to demonstrate how SASSI's capabilities can be used for different types of architecture experiments. Section 5 explores SASSI's ability to inspect application control flow behavior, which can be a critical performance limiter on modern GPUs. Section 6 leverages SASSI to perform a detailed memory analysis, which specifically characterizes an application's *memory divergence*. Section 7 shows how SASSI allows access to an instrumented application's register contents to enable value profiling.

While SASSI is capable of instrumenting applications that target Fermi, Kepler, and Maxwell devices, the results we present in this paper were gathered on Kepler-based architectures. Specifically, the experiments presented in the aforementioned case studies target an NVIDIA Tesla K10 G2 with 8GB memory and display driver version 340.21. In addition, all experiments use the CUDA 6 toolkit, and we simply replace the standard `ptxas` with our SASSI-enabled version.

The final case study in Section 8, characterizes an application's sensitivity to transient errors by injecting faults into the architecturally visible state of a GPU. The experiments demonstrate how SASSI can be used to *change* a kernel's behavior (*e.g.*, by altering register values and memory locations). The experimental flow targets a Tesla K20 with 5GB memory, display driver version 340.29, and uses the CUDA 6.5 toolkit.

We choose benchmarks to present in each of the case study sections that reveal interesting behavior. With the exception of NERSC's `miniFE` application [17, 27], all of the benchmarks come from Parboil *v*2.5 [40] and Rodinia *v*2.3 [7].

5. Case Study I: Conditional Control Flow

Our first case study discusses a tool based on SASSI for analyzing SIMT control flow behavior. Explicitly parallel Single Program Multiple Data (SPMD) languages such as CUDA and OpenCL allow programmers to encode unconstrained control flow, including *gotos*, nested loops, if-then-else statements, function calls, etc. For GPUs when all of the threads in a warp execute the same control flow (*i.e.*, the threads in the warp share the same PC), they are said to be *converged*, and each thread in the warp is therefore *active*. Conditional control flow however, can cause a subset of threads to *diverge*. Such divergence has serious performance ramifications for GPU architectures. For NVIDIA's architectures, the hardware chooses one path to continue executing, and defers the execution of threads on the alternate paths by pushing the deferred thread IDs and their program counters onto a *divergence stack* [19]. At this point, only a subset of the threads actually execute, which causes warp efficiency to drop. At well-defined reconvergence points (which are automatically determined by the compiler), the hardware pops the deferred threads off the stack and begins executing the deferred threads.

This case study uses SASSI to collect per-branch control flow statistics. Specifically, we will show an instrumentation handler that uses counters to record for each branch 1) the total number of times the branch was executed, 2) how many threads were active, 3) how many threads took the branch, 4) how many threads "fell through", 5) and how often it caused a warp to split (*i.e.*, divergent branch).

```
1  __device__ void sassi_before_handler(SASSIBeforeParams *bp,
2                                       SASSICondBranchParams *brp)
3  {
4    // Find out thread index within the warp.
5    int threadIdxInWarp = threadIdx.x & (warpSize-1);
6
7    // Find out which way this thread is going to branch.
8    bool dir = brp->GetDirection();
9
10   // Get masks and counts of 1) active threads in this warp,
11   // 2) threads that take the branch, and
12   // 3) threads that do not take the branch.
13   int active = __ballot(1);
14   int taken = __ballot(dir == true);
15   int ntaken = __ballot(dir == false);
16   int numActive = __popc(active);
17   int numTaken = __popc(taken), numNotTaken = __popc(ntaken);
18
19   // The first active thread in each warp gets to write results.
20   if ((__ffs(active)-1) == threadIdxInWarp) {
21     // Find the instruction's counters in a hash table based on
22     // its address. Create a new entry if one does not exist.
23     struct BranchStats *stats = find(bp->GetInsAddr());
24
25     // Increment the various counters that are associated
26     // with this instruction appropriately.
27     atomicAdd(&(stats->totalBranches), 1ULL);
28     atomicAdd(&(stats->activeThreads), numActive);
29     atomicAdd(&(stats->takenThreads), numTaken);
30     atomicAdd(&(stats->takenNotThreads), numNotTaken);
31     if (numTaken != numActive && numNotTaken != numActive) {
32       // If threads go different ways, note it.
33       atomicAdd(&(stats->divergentBranches), 1ULL);
34     }}}
```

Figure 4: Handler for conditional branch analysis.

5.1. SASSI Instrumentation

Instrumentation where and what: This analysis targets program control flow. We instruct SASSI to instrument before all conditional control flow instructions, and at each instrumentation site, we direct SASSI to collect and pass information about conditional control flow to the instrumentation handler.
Instrumentation handler: Figure 4 shows the instrumentation handler we use for this case study. SASSI will insert calls to this handler before every conditional branch operation.

The handler first determines the thread index within the warp (line 5) and the direction in which the thread is going to branch (line 8). CUDA provides several warp-wide broadcast and reduction operations that NVIDIA's architectures efficiently support. For example, all of the handlers we present in this paper use the `__ballot(predicate)` instruction, which "evaluates `predicate` for all active threads of the warp and returns an integer whose N^{th} bit is set if and only if `predicate` evaluates to non-zero for the N^{th} thread of the warp *and* the N^{th} thread is active" [32].

The handler uses `__ballot` on lines 13-15 to set masks corresponding to the active threads (`active`), the threads that are going to take the branch (`taken`), and the threads that are going to fall through (`ntaken`). With these masks, the handler

Table 1: Average branch divergence statistics.

	Benchmark (Dataset)	Static Total Branches	Static Divergent Branches	Static Divergent %	Dynamic Total Branches	Dynamic Divergent Branches	Dynamic Divergent %
Parboil	bfs (1M)	41	19	46	3.66 M	149.68 K	4.1
	bfs (NY)	41	22	54	933.88 K	119.45 K	12.8
	bfs (SF)	51	26	51	3.75 M	184.63 K	4.9
	bfs (UT)	41	20	49	697.28 K	104.08 K	14.9
	sgemm (small)	2	0	0	1.04 K	0	0.0
	sgemm (medium)	2	0	0	528.00 K	0	0.0
	tpacf (small)	25	5	20	14.85 M	3.75 M	25.2
Rodinia	bfs	7	2	29	3.71 M	525.54 K	14.2
	gaussian	10	4	40	492.38 M	1.18 M	0.2
	heartwall	161	50	31	226.85 M	95.44 M	42.1
	srad_v1	28	7	25	9.44 M	46.03 K	0.5
	srad_v2	19	12	63	11.20 M	2.38 M	21.3
	streamcluster	7	0	0	442.11 M	0	0.0

uses the *population count* instruction (__popc) to efficiently determine the number of threads in each respective category (numActive, numTaken, numNotTaken).

On line 20 the handler elects the first active thread in the warp (using the *find first set* CUDA intrinsic, __ffs) to record the results. Because this handler records per-branch statistics, it uses a hash table in which to store counters. Line 23 finds the hash table entry for the instrumented branch (using a function not shown here). Lines 27-33 update the counters.

As we described in Section 3, we rely on the CUPTI library to register callbacks for kernel *launch* and *exit* events [33]. Using these callbacks, which run on the host, we can appropriately marshal data to initialize and record the values in the device-side hash table.

5.2. Results

Table 1 summarizes the average per-branch divergence statistics for a selected set of Parboil and Rodinia benchmarks with different input datasets. For each benchmark, we calculate the fraction of branches in the code that were divergent ("Static" column), and how often branches diverged throughout execution ("Dynamic" column) thereby reducing warp efficiency.

Some benchmarks are completely convergent, such as sgemm and streamcluster, and do not diverge at all. Other benchmarks diverge minimally, such as gaussian and srad_v1, while, benchmarks such as tpacf and heartwall experience abundant divergence. An application's branch behavior can change with different datasets. For example, Parboil's bfs shows a spread of 4.1–14.9% dynamic branch divergence across four different input datasets. In addition, branch behavior can vary across different implementations of the same application (srad_v1 vs. srad_v2, and Parboil bfs vs. Rodinia bfs).

Figure 5 plots the detailed per-branch divergence statistics we can get from SASSI. For Parboil bfs with the 1M dataset, two branches are the major source of divergence, while with the UT dataset, there are six branches in total (including the previous two) that contribute to a 10% increase in dynamic branch divergence. SASSI simplifies the task of collecting per-branch statistics with its easy-to-customize instrumentation handler, and also makes it tractable to run all input datasets with its low runtime overhead.

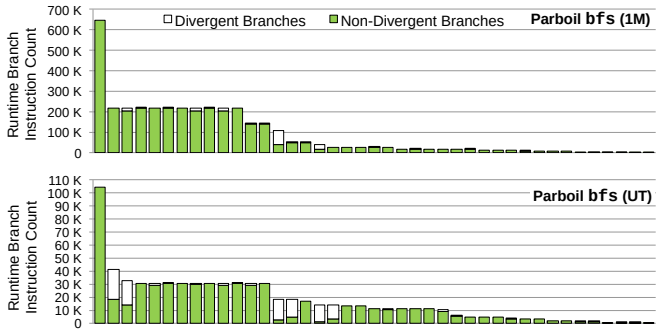


Figure 5: Per-branch divergence statistics of the Parboil bfs benchmark with different input datasets. Each bar represents an unique branch in the code. The branches are sorted in a descending order of runtime branch instruction count.

6. Case Study II: Memory Divergence

Memory access patterns can impact performance, caching effectiveness, and DRAM bandwidth efficiency. In the SIMT execution model, warps can issue loads with up to 32 unique addresses, one per thread. Warp-wide memory access patterns determine the number of memory transactions required. To reduce total requests sent to memory, accesses to the same cacheline are combined into a single request in a process known as *coalescing*. Structured access patterns that touch a small number of unique cachelines are more efficiently coalesced and consume less bandwidth than irregular access patterns that touch many unique cachelines.

Warp instructions that generate inefficient access patterns are said to be *memory address diverged*. Because warp instructions execute in lock-step in the SIMT model, all memory transactions for a given warp must complete before the warp can proceed. Requests may experience wide variance in latency due to many factors, including cache misses, memory scheduling, and variable DRAM access latency.

Architects have studied the impact of memory divergence and ways to mitigate it in simulation [6, 22, 36, 37, 42]. For this case study, we demonstrate instrumentation to provide in-depth analysis of memory address divergence. While production analysis tools provide the ability to understand broad behavior, SASSI can enable much more detailed inspection of memory access behavior, including: the frequency of address divergence; the distribution of unique cachelines touched per instruction; correlation of control divergence with address divergence; and detailed accounting of unique references generated per program counter.

6.1. SASSI Instrumentation

Instrumentation where and what: We instruct SASSI to instrument before all memory operations, and at each instrumentation site, we direct SASSI to collect and pass memory-specific information to the instrumentation handler.

```
1  __device__ void sassi_before_handler(SASSIBeforeParams *bp,
2                                       SASSIMemoryParams *mp)
3  {
4    if (bp->GetInstrWillExecute()) {
5      intptr_t addrAsInt = mp->GetAddress();
6      // Only look at global memory requests. Filter others out.
7      if (__isGlobal((void*)addrAsInt)) {
8        unsigned unique = 0; // Num unique lines per warp.
9
10       // Shift off the offset bits into the cache line.
11       intptr_t lineAddr = addrAsInt >> OFFSET_BITS;
12
13       int workset = __ballot(1);
14       int firstActive = __ffs(workset)-1;
15       int numActive = __popc(workset);
16       while (workset) {
17         // Elect a leader, get its cache line, see who matches it.
18         int leader = __ffs(workset) - 1;
19         intptr_t leadersAddr = bcast(lineAddr, leader);
20         int notMatchesLeader = __ballot(leadersAddr != lineAddr);
21
22         // We have accounted for all values that match the leader's.
23         // Let's remove them all from the workset.
24         workset = workset & notMatchesLeader;
25         unique++;
26       }
27
28       // Each thread independently computes 'numActive', 'unique'.
29       // Let the first active thread actually tally the result
30       // in a 32x32 matrix of counters.
31       int threadIdxInWarp = threadIdx.x & (warpSize-1);
32       if (firstActive == threadIdxInWarp) {
33         atomicAdd(&(sassi_counters[numActive-1][unique-1]), 1LL);
34       }}}}
```

Figure 6: Handler for memory divergence profiling.

Instrumentation handler: Figure 6 shows the instrumentation handler for this case study. SASSI inserts a call to this handler before every operation that touches memory. Because NVIDIA's instruction set is predicated, this handler first filters out threads whose guarding predicate is false (line 4). This handler then selects only addresses to *global* memory (line 7).[1] Next, the handler computes each thread's requested cache line address (line 11). For this work, we use a 32B line size. Each active thread in the warp will have a *thread-local* value for the computed address in `lineAddr`. Lines 16-26 use reduction and broadcast operations to iteratively find the number of unique values of `lineAddr` across a warp.

This handler elects a leader thread to record the statistics (line 32). The leader populates a 32×32 (lower triangular) matrix of values in `sassi_counters[][]`, where the rows of the matrix record the number of active threads in the warp (`numActive`) and the columns correspond to the number of unique line addresses found. This handler also uses CUPTI to initialize and dump the statistics kept in `sassi_counters`.

6.2. Results

Figure 7 shows the the distribution (PMF) of unique cache lines (32B granularity) requested per warp instruction for a selection of address divergent applications. The distributions show the percentage of thread-level memory accesses issued from warps requesting N unique cache lines, where N ranges from 1 to 32.

[1] NVIDIA GPUs feature several memory spaces, including local memory, global memory, and shared memory.

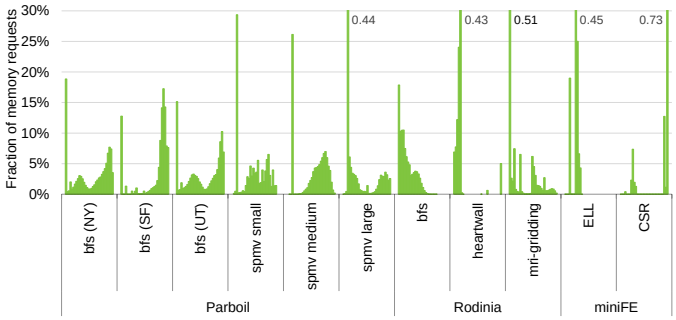


Figure 7: Distribution (PMF) of unique cachelines requested per warp memory instruction for a selection of memory address divergent applications.

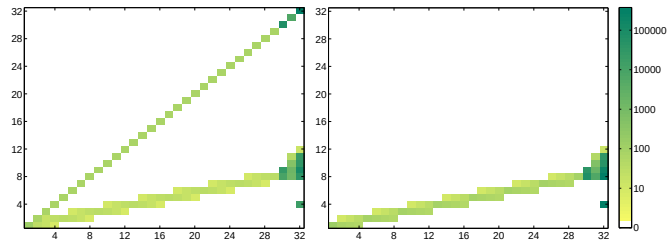


Figure 8: Memory access behavior for `miniFE` variants using different data formats. Warp occupancy is along the x-axis, address divergence is along the y-axis.

The applications shown exhibit complex data access patterns to irregularly structured data. Graph traversal operations such as `bfs` (breadth-first search) have unpredictable, data-dependent accesses that are frequently irregular. We can see how data-dependent behavior impacts memory divergence for three different datasets for `bfs` from the Parboil suite; each exhibits similar overall behavior, but the extent of the problem varies across datasets. Other applications such as `mri-gridding`, `spmv`, and `miniFE` use sparse data representations or indirection that limits dense, regular access.

Advanced developers structure their code to use access patterns or data formats that improve memory access regularity [2, 30]. For instance, Figure 7 shows two variants of `miniFE` that use different matrix formats (ELL vs. CSR). We can see that `miniFE`-ELL makes most of its memory requests from warp instructions with low address divergence. On the other hand, `miniFE`-CSR makes the majority of its accesses from warp instructions with high address divergence – with 73% of memory accesses being fully diverged (from warp instructions requesting the maximum 32 unique lines). Figure 8 provides an in-depth look at the two variants of `miniFE`. We see a two-dimensional plot that accounts for both warp occupancy (number of active threads) as well as address divergence (number of unique cachelines requested). In the case of CSR, an irregular access pattern results. In this implementation, many instructions are maximally address divergent, generating as many unique requests as active threads (the diagonal). For ELL, we can see that the distribution of unique requests,

```
__device__ void sassi_after_handler(SASSIAfterParams* ap,
                                    SASSIRegisterParams *rp)
{
  int threadIdxInWarp = threadIdx.x & (warpSize-1);
  int firstActiveThread = (__ffs(__ballot(1))-1); /*leader*/

  // Get the address of this instruction, use it as a hash into a
  // global array of counters.
  struct handlerOperands *stats = find(ap->GetInsAddr());

  // Record the number of times the instruction executes.
  atomicAdd(&(stats->weight), 1);
  stats->numDsts = rp->GetNumGPRDsts();
  for (int d = 0; d < rp->GetNumGPRDsts(); d++) {
    // Get the value in each destination register.
    SASSIGPRRegInfo regInfo = rp->GetGPRDst(d);
    int valueInReg = (int)rp->GetRegValue(ap, regInfo);
    stats->regNum[d] = rp->GetRegNum(regInfo);

    // Use atomic AND operations to track constant bits.
    atomicAnd(&(stats->constantOnes[d]), valueInReg);
    atomicAnd(&(stats->constantZeros[d]), ~valueInReg);

    // Get the leader's 'valueInReg', see if all threads agree.
    int leaderValue = __shfl(valueInReg, firstActiveThread);
    int allSame = (__all(valueInReg == leaderValue) != 0);

    // The warp leader gets to write results.
    if (threadIdxInWarp == firstActiveThread) {
      atomicAnd(&(stats->isScalar[d]), allSame);
    }}}
```

Figure 9: A simplified handler for value profiling.

while still correlated to the number of active threads, is shifted lower. In this case, threads are making more aligned requests that are better coalesced.

7. Case Study III: Value Profiling and Analysis

This section presents a simple profiling handler that tracks all instructions that produce register values and determines the following properties: (1) which bits of the generated values are always constant across all warps, and (2) which instructions are *scalar*, *i.e.*, the instruction produces the same values for all threads within a warp. Scalar analysis has been proposed to reduce register file capacity by allowing a significant amount of sharing across threads in a warp [20]. Similarly, there have been proposals to pack multiple, narrow-bitwidth operands into a single register [41], and hardware mechanisms exist for exploiting narrow-bitwidth computations (*e.g.*, by aggressively clock-gating unused portions of the datapath) [5]. This section's analysis provides insight into how many register file bits (a premium in GPUs) current applications are wasting. Such insight is a valuable guide to the opportunities that architecture mechanisms can provide.

7.1. SASSI Instrumentation

Instrumentation where and what: To track the values a shader's instructions produce, we use SASSI to instrument after all instructions that write to one or more registers. We instruct SASSI to collect and pass to the instrumentation handler the register information for each instrumented instruction.

Instrumentation handler: Figure 9 shows a simplified version of the value profiling instrumentation handler for this case study. The handler performs the following five steps: (1) it elects a leader from the warp's active threads to write back results about whether a write is scalar; (2) it gets a pointer to the current instruction's statistics by hashing into a device-global hash table; (3) it iterates through the destination registers, extracting the value to be written to each destination; (4) it uses the `atomicAnd` function to keep track of constant one- and zero-bits; and (5) it uses `__shfl` and `__all` to communicate and decide whether a value is scalar. The leader records the outcome in the hash table.

The instrumentation library uses the CUPTI library to register callbacks for kernel launch and exit events [33]. Our launch callback initializes the hash table, and our exit callback copies the hash table off the device. This approach allows us to track statistics per kernel invocation.

Table 2: Results for value profiling.

		Dynamic %		Static %	
	Benchmark	const bits	scalar	const bits	scalar
Parboil	bfs	72	46	79	52
	cutcp	16	25	45	42
	histo	70	20	65	27
	lbm	25	4	28	7
	mri-gridding	66	66	60	35
	mri-q	19	40	52	51
	sad	51	5	58	35
	sgemm	17	47	27	44
	spmv	54	43	60	48
	stencil	49	35	58	42
	tpacf	70	26	72	33
Rodinia	b+tree	73	76	74	80
	backprop	73	37	72	33
	bfs	72	44	68	38
	gaussian	71	54	57	50
	heartwall	60	11	75	54
	hotspot	65	43	67	43
	kmeans	38	33	59	51
	lavaMD	46	30	54	40
	lud	33	19	42	22
	mummergpu	57	12	62	18
	nn	40	31	40	31
	nw	23	16	27	18
	pathfinder	66	19	65	37
	srad_v1	47	26	53	35
	srad_v2	48	28	60	35
	streamcluster	38	54	54	42

7.2. Results

In our library's thread exit callback function, the library dumps the value profile for the associated kernel invocation, recording the profile for each instruction that writes one or more registers. For example, the output for a *texture* load from Parboil's `bfs` that loads a 64-bit quantity into two adjacent registers is:

```
TLD.LZ.P    R12, R16, RZ, 0x0000, 1D, 0x3;
  R12  <- [00000000000000TTTTTTTTTTTTTTTT]
  R13* <- [00000000000000000000000000000001]
```

This output shows that this instruction *always* loaded the value 1 into `R13`, across all threads in the kernel. At the same time, only the lower 18 bits of `R12` varied (as is indicated by the `T` values) during the kernel's execution; the upper 14 bits were always 0. The analysis identifies `R13` as scalar, as noted by the asterisk.

Our instrumentation library generates a coarse summary of the scalar and bit invariance properties of the instrumented program. Table 2 summarizes the results for Parboil and Rodinia. We use the largest input data set available for Parboil, and the default inputs for Rodinia. For each benchmark we show the dynamic and static percentage of register bit assignments that are constant and scalar. The static metrics weigh each instruction equally, while the dynamic metrics use instruction frequency to approximate the true dynamic statistics.

These results show that for these suites of benchmarks, the architecture is making poor use of the register file. Most benchmarks have a significant percentage of dynamic scalar operations, ranging up to 76% for `b+tree`. In addition, for these profile runs, most of the operands in these benchmarks require only a fraction of the 32-bit register allotted to them. With a remarkably concise amount of code, SASSI exposes interesting insights that can drive architecture studies that aim to improve register file efficiency.

8. Case Study IV: Error Injection

This section demonstrates how SASSI can be employed to evaluate and analyze GPU application sensitivity to transient hardware errors, by injecting errors into the architecture state of a running GPU. To the best of our knowledge, the only prior research that examines GPU error vulnerability used CUDA-GDB [14, 31]. That work also performed instruction-level statistical error injection to study application vulnerability, but lacked the ability to modify predicate registers and condition codes. Furthermore, because breaking after every instruction using CUDA-GDB and transferring control to the host system is prohibitively expensive, that work required a complex and heuristic-based profiling step to select error injection sites.

A SASSI-based error injection approach overcomes these two challenges as it can modify any ISA visible state, including predicate registers and condition codes. Further, the instrumented code executes on the GPU, which makes the profiling step much faster and more accurate. Performing error injections using SASSI requires three main steps: (1) profiling and identifying the error injection space; (2) statistically selecting error injection sites; and (3) injecting errors into executing applications and monitoring error behavior. Steps (1) and (3) occur on different executions using two different SASSI instrumentation handlers.

In this study, we define an architecture-level error as a single-bit flip in one of the destination registers of an executing instruction. If the destination register is a general purpose register (32-bit value) or a condition code (4-bit value), one bit is randomly selected for flipping. For predicate registers, we only flip a destination predicate bit that is being written by the instruction.

8.1. SASSI Instrumentation

Instrumentation where and what: For the profiling step, we instrument after all instructions that either access memory or write to a register and exclude instructions that are predicated out. We collect and pass the register and memory information for each instrumented instruction to the handler, which records the state modifications so that an off-line tool can stochastically select the error injection site. For error injections, we instrument the same set of instructions and use the handler to inject the error into the location selected by the stochastic process.

Instrumentation handler: In the profiling step, we collect the following information to identify the error injection space: (1) static kernel names, (2) the number of times each kernel executes, (3) the number of threads per kernel invocation, and (4) the number of dynamic instructions per thread that are not predicated out and either write to a register or a memory location. We use CUPTI to collect (1) and (2) and instrument the instructions using SASSI to collect (3) and (4).

Using this information we randomly select 1,000 error injection sites per application, which is a tuple consisting of the static kernel name, dynamic kernel invocation ID, thread ID, dynamic instruction count, seed to select a destination register, and seed to select the bit for injection. This step is performed on the host CPU.

In the last and the most important step, we inject one error per application run and monitor for crashes, hangs, and output corruption. In each injection run, we check if the selected kernel and its dynamic invocation count has been reached using CUPTI. If so, we copy the remaining error site tuple into the device memory. During kernel execution, we check if the current thread is the selected thread for injection in the instrumentation handler. For the selected thread, we maintain a counter and check if the dynamic instruction that just executed is the selected instruction for injection. If it is the selected instruction, we inject the error into the bit and the register specified by the seeds in the tuple.

After the handler injects the error the application continues unhindered (unless our experimental framework detects a crash or a hang). We categorize the injection outcome based on the exit status of the application, hang detection, error messages thrown during execution, and output differences from that of an error-free reference run.

8.2. Results

Figure 10 shows how different applications respond to architecture-level errors. As mentioned earlier, we performed 1,000 error injection runs per application. This figure shows that approximately 79% of injected errors on average (using our error model) did not have any impact on the program output. Only 10% resulted in crashes or hangs. Approximately 4% of injections showed symptoms of failures (unsuccessful kernel execution or explicit error messages in stdout/stderr), which can be categorized as potential crashes with appropriate error monitors. The remaining injections corrupt some application output (stdout or stderr or a program defined output file). We categorize such cases as potential silent data corrup-

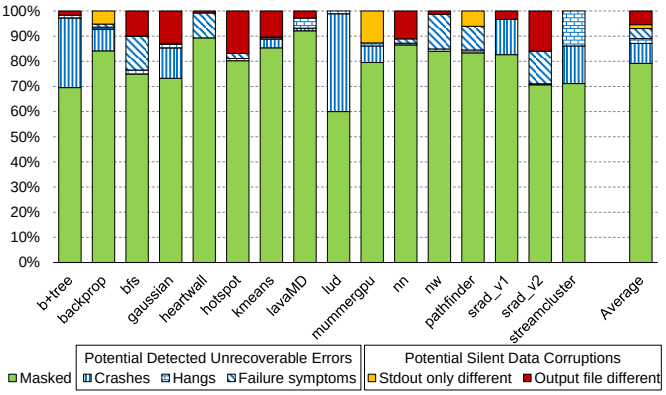


Figure 10: Error injection outcomes of different applications. Each bar shows the results from 1,000 error injection runs.

tions (SDCs). We observed that 1.5% of injections showed differences only in stdout/stderr when compared to error-free executions without corrupting program output files, which may be acceptable for some applications. Lastly, only 5.4% showed differences in program output files. Additional details can be found in [16].

SASSI provides the ability to explore the architecture vulnerability of GPUs. We further expect that SASSI will be a valuable tool in exploring hardware and software error mitigation schemes.

9. Discussion

This section discusses additional SASSI usage considerations, including runtime overheads, instrumentation side-effects, limitations, and possible extensions.

9.1. Runtime Overhead

The overhead of SASSI instrumentation depends on where we insert instrumentation and what instrumentation we insert. Table 3 shows the overheads of instrumentation for each of the case studies. For each benchmark, the three columns under the "Baseline" heading show the wall clock runtime t (in seconds), the total time spent executing kernels k (in milliseconds), and the total number of kernel launches. The benchmarks in the table are sorted by the fraction of total runtime spent in the GPU kernel, from smallest to largest.

We use `nvprof` [35] to collect device-side statistics and the Linux `time` command to measure whole-program runtimes. All results were collected on a 3GHz Intel® Xeon® E5-2690 v2 with an NVIDIA Tesla K40m and driver version 346.41. We use the median of five runs for each experiment, compile benchmarks at the "-O3" optimization level, and invoke the applications with the largest supplied input data set.

For each case study we measure the total wall-clock runtime T and device-side runtime K of instrumentation relative to the baselines t and k, respectively. As the two instances where instrumentation leads to performance improvements indicate, these measurements contain some jitter.

As expected, the fewer SASS instructions SASSI instruments, the lower the overhead of instrumentation. As Case Study I only instruments conditional branches, it sees relatively modest slowdowns. However, as Case Studies III and IV add instrumentation after every SASS instruction that writes a register, the slowdowns can be significant. The maximum slowdown we see is over 160× for whole-program execution.

For applications with a considerable fraction of CPU and data transfer time, the whole-program overhead of instrumentation is typically negligible, whereas for GPU-bound applications, the overhead can be large. Even heavy instrumentation does not significantly disrupt many of these applications, simply because they are CPU-bound. Also of note, `nvprof` treats the marshalling of data between the device and host as kernel time, which reduces the apparent kernel-level impact of instrumentation on memory-bound applications such as `mummergpu`.

We removed the body of the instrumentation handlers to measure the overhead of spilling and setting up ABI-compliant calls. Surprisingly the runtime overhead does not decrease dramatically when we stub out the code in the instrumentation handlers. For all case studies, the overhead of ABI-compliance and spilling live registers dominates, consuming roughly 80% of the total overhead.

Future work will consider optimizations to reduce the baseline overhead of instrumentation. One approach involves tracking which live variables are statically guaranteed to have been previously spilled but not yet overwritten, which will allow us to forgo re-spilling registers. In addition, while passing references to C++ objects is convenient for programmers, stack-allocating the objects is expensive; We may consider mechanisms to more efficiently pass parameters to the handlers.

Because SASSI-instrumented programs run at native hardware rates, it enables users to quickly refine their experiments, a feature not possible in the context of GPU simulation frameworks. Our worst kernel-level slowdown of 722× is much faster than simulators such as GPGPU-Sim, which are 1–10 million times slower than native execution.

9.2. Instrumentation Side-effects

While SASSI is intended to be minimally invasive, additional instructions, register pressure, and cache effects of SASSI instrumentation can alter the behavior of applications that contain race-conditions or rely on specific scheduling or timing assumptions. Even *without* instrumentation, `tpacf` in the Parboil suite produces inconsistent results across different architectures, particularly on the medium and large input sets. While we did not observe applications that exhibited non-deterministic behavior with the addition of SASSI, if an application is already susceptible to non-deterministic behavior, SASSI instrumentation will likely exacerbate this non-determinism.

Table 3: Instrumentation overheads. The "Baseline" column shows the wall clock time t, and the time spent executing kernels k, for each of the benchmarks. The "T" column for each case study shows the total runtime of the instrumented application with respect to t, and the "K" column shows the device-side runtime with respect to k.

		Baseline			Case Study I Cond. Branches		Case Study II Memory Divergence		Case Study III Value Profiling		Case Study IV Error Injection	
	Benchmark	t=Total time (s)	k=Kernel time (ms)	Kernel launches	T	K	T	K	T	K	T	K
Parboil	sgemm	2.0	7.8	4	1.0t	1.9k	1.5t	111.8k	2.1t	293.3k	2.2t	286.4k
	spmv	2.2	24.3	58	1.0t	3.5k	1.3t	19.9k	1.8t	72.8k	1.8t	73.1k
	bfs	2.3	54.3	37	1.1t	3.7k	1.2t	11.7k	1.6t	25.5k	1.4t	20.8k
	mri-q	0.3	9.2	15	1.5t	16.1k	1.1t	1.2k	22.3t	722.1k	21.1t	678.3k
	mri-gridding	9.6	374.2	81	1.5t	17.3k	1.4t	13.9k	6.3t	139.8k	4.7t	98.9k
	cutcp	3.0	176.1	31	5.1t	81.3k	3.8t	60.3k	42.6t	714.4k	40.3t	676.2k
	histo	40.4	4466.1	71042	4.4t	29.8k	5.9t	46.0k	30.8t	270.4k	29.1t	257.0k
	stencil	1.6	188.2	104	4.3t	27.1k	9.4t	69.9k	32.3t	255.5k	32.3t	258.6k
	sad	3.1	498.9	7	1.1t	1.1k	1.1t	1.6k	3.8t	17.8k	3.5t	16.2k
	lbm	7.2	5611.4	3003	2.0t	2.2k	18.3t	23.1k	103.0t	129.6k	98.2t	125.0k
	tpacf	5.4	4280.6	4	18.9t	23.0k	10.9t	13.6k	160.6t	205.0k	148.9t	187.0k
Rodinia	nn	0.3	0.1	3	1.0t	2.0k	1.0t	2.2k	0.9t	8.7k	1.0t	8.2k
	hotspot	0.7	0.4	4	1.1t	8.6k	1.0t	16.3k	1.0t	121.6k	1.1t	120.2k
	lud	0.4	1.7	48	1.0t	7.1k	1.0t	22.4k	1.3t	80.8k	1.3t	67.1k
	b+tree	1.8	12.5	20	1.0t	3.5k	1.0t	10.0k	1.3t	39.3k	1.2t	38.4k
	bfs	2.0	16.4	55	1.0t	4.7k	1.1t	14.0k	1.3t	34.4k	1.4t	34.7k
	pathfinder	1.3	12.1	8	1.1t	2.6k	1.1t	7.1k	1.2t	20.3k	1.2t	20.7k
	srad_v2	2.3	23.0	8	1.0t	5.1k	1.1t	12.3k	1.7t	69.1k	1.7t	69.9k
	mummergpu	7.7	90.1	13	1.1t	1.3k	1.1t	1.1k	1.2t	4.0k	1.1t	3.5k
	backprop	0.3	4.8	10	1.0t	1.5k	1.1t	5.4k	1.3t	17.8k	1.3t	18.5k
	kmeans	1.6	32.3	10	1.0t	2.1k	0.9t	2.4k	1.5t	26.7k	1.5t	25.5k
	lavaMD	0.6	21.5	6	1.4t	13.8k	2.1t	30.8k	17.7t	452.5k	16.2t	422.4k
	srad_v1	0.4	21.2	708	1.4t	8.5k	4.6t	62.0k	14.5t	227.8k	14.5t	232.5k
	nw	0.3	25.5	258	1.0t	1.0k	1.3t	5.3k	2.0t	13.9k	1.9t	13.5k
	gaussian	1.5	254.9	2052	4.4t	18.7k	2.3t	8.4k	12.7t	69.4k	6.3t	32.9k
	streamcluster	7.1	2431.5	11278	2.0t	3.8k	8.7t	22.8k	34.7t	99.9k	33.0t	95.6k
	heartwall	0.5	227.4	40	9.9t	22.1k	30.0t	70.6k	103.3t	229.6k	93.7t	220.7k
	Minimum	0.3	0.1	3	1.0t	1.0k	0.9t	1.1k	0.9t	4.0k	1.0t	3.5k
	Maximum	40.4	5611.4	71042	18.9t	81.3k	30.0t	111.8k	160.6t	722.1k	148.9t	678.3k
	Harmonic mean	0.9	1.3	12.2	1.4t	3.4k	1.6t	5.7k	2.4t	31.8k	2.4t	29.0k

9.3. Concurrency Issues and Limitations

Because SASSI instrumentation code is written in CUDA, it is parallel by construction. Designing instrumentation handlers for SASSI requires the user to carefully consider synchronization and data sharing. SASSI handlers can exploit most of the explicit parallel features of CUDA, including operations for voting, atomics, shuffle, and broadcast. However, not all CUDA is legal within SASSI instrumentation handlers. For example, thread barriers (`syncthreads`) cannot be used because the instrumentation function may be called when the threads in a warp are diverged; `syncthreads` executed by diverged warps precludes all threads from reaching the common barrier. Finally, SASSI instrumentation libraries that use shared resources, such as shared and constant memory, not only risk affecting occupancy, but they could also cause instrumented programs to fail. For instance, it is not uncommon for programs to use *all* of shared memory, leaving nothing for the instrumentation library. In practice, we have not been limited by these restrictions.

9.4. SASSI Extensions

Exploiting compile-time information: As part of the backend compiler, SASSI has structural and type information that cannot be easily reconstructed dynamically. For instance, unlike SASSI, binary instrumentation frameworks generally cannot identify *static* basic block headers [12]. Operand datatype information can also be passed to SASSI handlers, information that is not explicitly encoded in a program's binary code.

Instrumenting heterogeneous applications: SASSI can be used in conjunction with host-side instrumentation tools like Pin to enable whole-program analysis of applications. This approach requires some degree of coordination between the host- and device-side instrumentation code, particularly when used to form a unified stream of events for analysis. We have already built a prototype to examine the sharing and CPU-GPU page migration behavior in a Unified Virtual Memory system [29] by tracing the addresses touched by the CPU and GPU. A CPU-side handler processes and correlates the traces.

Driving other simulators: SASSI can collect low-level traces of device-side events, which can then be processed by separate tools. For instance, a memory trace collected by SASSI can be used to drive a memory hierarchy simulator.

9.5. Graphics Shaders

Instrumentation of OpenGL and DirectX shaders is feasible with SASSI. Graphics shaders require SASSI to be part of the driver because they are always JIT compiled. Graphics shaders do not adhere to the CUDA ABI nor do they maintain a stack, and therefore SASSI must allocate and manage a stack from which the handler code can operate. Aside from stack management, the mechanics of setting up a CUDA ABI-compliant call from a graphics shader remain unchanged.

10. Related Work

To our knowledge, this paper is the first to introduce an accurate and flexible selective instrumentation framework for GPU applications. The major contribution of this work is demonstrating a middle ground for measuring, characterizing, and analyzing GPU application performance that provides accurate hardware-rate analysis while being flexible enough to measure many interesting aspects of execution.

Many profilers rely on specialized hardware support, such as NSight [34], Visual Profiler [35], and ProfileMe [11]. SASSI on the other hand, like the remainder of the related work in this section, is purely software-based. We qualitatively compare SASSI to alternative approaches, including binary instrumentation and compiler-based frameworks.

10.1. Binary Instrumentation

Tools such as Pin [21], DynamoRIO [12], Valgrind [28], and Atom [38] allow for flexible binary instrumentation of programs. Binary instrumentation offers a major advantage over compiler-based instrumentation approaches such as SASSI employs: users do not need to recompile their applications to apply instrumentation. Not only is recompilation onerous, but there are cases where vendors may not be willing to relinquish their source code, making recompilation impossible.

On the other hand, compiler-based instrumentation approaches have some tangible benefits. First, the compiler has information that is difficult, if not impossible, to reconstruct at runtime, including control-flow graph information, register liveness, and operand data-types. Second, in the context of just-in-time compiled systems (as is the case with graphics shaders and appropriately compiled compute shaders), programs are *always recompiled* before executing anyway. Finally, compiler-based instrumentation is more efficient than binary instrumentation because the compiler has the needed information to spill and refill the minimal number of registers.

10.2. Direct-execution Simulation

Another approach related to compiler-based instrumentation is *direct execution* to accelerate functional simulators. Tools such as RPPT [9], Tango [10], Proteus [4], Shade [8], and Mambo [3] all translate some of the simulated program's instructions into the native ISA of the host machine where they execute at hardware speeds. The advantage of these approaches for architecture studies is that they are built into simulators designed to explore the design space and they naturally co-exist with simulator performance models. The disadvantage is that one has to implement the simulator and enough of the software stack to run any code at all. By running directly on native hardware, SASSI inherits the software stack and allows a user to explore only those parts of the program they care about. While we have not yet done so, one can use SASSI as a basis for an architecture performance simulator.

10.3. Compiler-based Instrumentation

Ocelot is a compiler framework that operates on PTX code, ingesting PTX emitted by a front-end compiler, modifying it in its own compilation passes, and then emitting PTX for GPUs or assembly code for CPUs. Ocelot was originally designed to allow architectures other than NVIDIA GPUs to leverage the parallelism in PTX programs [13], but has also been used to perform instrumentation of GPU programs [15]. While Ocelot is a useful tool, it suffers from several significant problems when used as a GPU instrumentation framework. First, because Ocelot operates at the virtual ISA (PTX) level, it is far divorced from the actual binary code emitted by the backend compiler. Consequently, Ocelot interferes with the backend compiler optimizations and is far more invasive and less precise in its ability to instrument a program. SASSI's approach to instrumentation, which allows users to write handlers in CUDA, is also more user-friendly than the C++ "builder" class approach employed in [15].

11. Conclusion

This paper introduced SASSI, a new assembly-language instrumentation tool for GPUs. Built into the NVIDIA production-level backend compiler, SASSI enables a user to specify specific instructions or instruction types at which to inject a call to a user-provided instrumentation function. SASSI instrumentation code is written in CUDA and is inherently parallel, enabling users to explore the parallel behavior of applications and architectures. We have demonstrated that SASSI can be used for a range of architecture studies, including instruction control flow, memory systems, value similarity, and resilience. Similar to CPU binary instrumentation tools, SASSI can be used to perform a wide range of studies on GPU applications and architectures. The runtime overhead of SASSI depends in part on the frequency of instrumented instructions and the complexity of the instrumentation code. Our studies show a range of runtime slowdowns from 1–160×, depending on the experiment. While we have chosen to implement SASSI in the compiler, nothing precludes the technology from being integrated into a binary rewriting tool for GPUs. Further, we expect that the SASSI technology can be extended in the future to include graphics shaders.

12. Acknowledgments

We would like to thank the numerous people at NVIDIA who provided valuable feedback and training during SASSI's development, particularly Vyas Venkataraman. We thank Jason Clemons who helped us generate figures, and Neha Agarwal who provided an interesting early use case.

References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[2] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA, Tech. Rep. NVR-2008-004, December 2008.

[3] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang, "Mambo: A Full System Simulator for the PowerPC Architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 8–12, 2004.

[4] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl, "PROTEUS: A High-performance Parallel-architecture Simulator," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 1992, pp. 247–248.

[5] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, January 1999, pp. 13–22.

[6] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, November 2012, pp. 141–151.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[8] B. Cmelik and D. Keppel, "Shade: A Fast Instruction-set Simulator for Execution Profiling," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1994, pp. 128–137.

[9] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1988, pp. 4–11.

[10] H. Davis, S. R. Goldschmidt, and J. Hennessy, "Multiprocessor Tracing and Simulation Using Tango," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 1991.

[11] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 1997, pp. 292–302.

[12] Derek Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[13] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU Binaries to Tiered Many-Core Architectures with Ocelot," Georgia Institute of Technology Center for Experimental Research in Computer Systems (CERCS), Tech. Rep. 0901, January 2009.

[14] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 221–230.

[15] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, March 2011.

[16] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: Evaluating Resilience of GPU Applications," in *Proceedings of the Workshop on Silicon Errors in Logic - System Effects (SELSE)*, April 2015.

[17] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Labs, Tech. Rep. SAND2009-5574, September 2009.

[18] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2010, pp. 60–71.

[19] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović, "Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2014, pp. 101–113.

[20] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic, "Convergence and Scalarization for Data-parallel Architectures," in *International Symposium on Code Generation and Optimization (CGO)*, February 2013, pp. 1–11.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2005, pp. 190–200.

[22] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2010, pp. 235–246.

[23] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, January 2010, pp. 1–12.

[24] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-chip Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 196–207.

[25] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2007, pp. 146–160.

[26] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 2005, pp. 284–295.

[27] National Energy Research Scientific Computing Center, "MiniFE," https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/minife, 2014.

[28] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2007, pp. 89–100.

[29] NVIDIA. (2013, November) Unified Memory in CUDA 6. Available: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[30] NVIDIA. (2014, August) CUDA C Best Practices Guides. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[31] NVIDIA. (2014, August) CUDA-GDB :: CUDA Toolkit Documentation. Available: http://docs.nvidia.com/cuda/cuda-gdb/index.html

[32] NVIDIA. (2014, November) CUDA Programming Guide :: CUDA Toolkit Documentation. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[33] NVIDIA. (2014, November) CUPTI :: CUDA Toolkit Documentation. Available: http://docs.nvidia.com/cuda/cupti/index.html

[34] NVIDIA. (2014) NVIDIA NSIGHT User Guide. Available: http://docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight_visual_studio_edition_user_guide.htm

[35] NVIDIA. (2014, August) Visual Profiler Users's Guide. Available: http://docs.nvidia.com/cuda/profiler-users-guide

[36] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2013, pp. 99–110.

[37] J. Sartori and R. Kumar, "Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications," *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 279–290, February 2013.

[38] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 1994, pp. 196–205.

[39] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, May/June 2010.

[40] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing, Tech. Rep. IMPACT-12-01, March 2012.

[41] S. Tallam and R. Gupta, "Bitwidth Aware Global Register Allocation," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 2003, pp. 85–96.

[42] P. Xiang, Y. Yang, and H. Zhou, "Warp-level Divergence in GPUs: Characterization, Impact, and Mitigation," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 284–295.