SASSI User Guide

Mark Stephenson (mstephenson@nvidia.com)

August 13, 2015

Contents

List of Figures

List of Tables

1 Introduction

This document describes SASSI, a selective instrumentation framework for NVIDIA GPUs. SASSI stands for SASS Instrumenter, where SASS is NVIDIA's name for its native ISA. SASSI is a pass in NVIDIA's backend compiler, `ptxas`, that selectively inserts instrumentation code. The purpose of SASSI is to allow users to measure or modify `ptxas`-generated SASS by injecting instrumentation code *during* code generation.

NVIDIA has many excellent development tools. Why the need for another tool? NVIDIA's tools such as `cuda-memcheck` and `nvvp` provide excellent, but *fixed-function* inspection of programs. Tools like `cuda-memcheck` perform binary patching to insert instrumentation code. While they are great at what they are designed for, the user has to choose from a fixed menu of program characteristics to measure. If you want to measure some aspect of program execution outside the purview of those tools you are out of luck. One example that we provide in this guide — in Section 9 — is a perfect example of something that SASSI is good at, that NVIDIA's production tools currently cannot handle.

SASSI is invoked extremely late in the compiler's flow to minimize any disruption the instrumentation code has on the code the user wants to measure. Because SASSI is part of the compiler, it is important for us to clearly state up front that SASSI is not a *binary instrumenter*. SASSI instrumentation of programs requires the source code (either CUDA, OpenCL, or PTX). As a consequence, any device libraries (e.g., `cuFFT`) that your application links in will not be instrumented unless the user has explicitly recompiled the library with SASSI. We also note here that SASSI is a fork of the production backend compiler, and therefore, the code it instruments may deviate from that of the official `ptxas`.

As a small research effort however, working within the backend compiler affords us stability and portability. NVIDIA's architectures change dramatically from generation to generation, yet because SASSI is part of the compiler flow, it can target Fermi, Kepler, and Maxwell devices, and it runs on Windows and Linux. NVIDIA Research is releasing SASSI "as-is", and users should realize that it is a *research prototype* with no promise of support or continuity.

2 Prerequisites, Getting, and Installing SASSI

SASSI is, in a nutshell, a specialized version of `ptxas` that has a few extra options for instrumenting applications. We distribute this specialized `ptxas` as a (closed-source) binary blob using GitHub's "releases" functionality. However, to make SASSI more usable, we also distribute several sample instrumentation libraries and utilities that we have found to be useful.

Throughout the rest of the guide, we will refer to the following environment variables:

- `$SASSI_REPO`: The base directory of the SASSI project, which you will clone from GitHub.
- `$CUDA_HOME`: The location of your CUDA 7 installation (which is a prerequisite for installing SASSI).
- `$SASSI_HOME`: The location of your SASSI installation, which is essentially a deep copy of your CUDA 7 installation, with a few added files and a different version of `ptxas`.

2.1 Prerequisites

First, let's walk through SASSI's prerequisites.

1. **Platform requirements**: SASSI requires an X86 64-bit host; a Fermi-, Kepler-, or Maxwell-based GPU; and at the time of this writing we have generated SASSI for Ubuntu (12 and 15), Debian 7, and CentOS 6.[1]
2. **Python requirement**: The SASSI installation script requires Python 2.7 or newer.
3. **Install CUDA 7**: At the time of this writing, CUDA 7 can be fetched from:

   `https://developer.nvidia.com/cuda-downloads`

[1] Please let us know if your platform is not listed, and we'll see what we can do.

```
$SASSI_REPO
  | EULA
  | FAQ
  | README
  | doc
    | sassi-user-guide.pdf
    | sassi-isca-2015.pdf
  | example
    | Makefile
    | matrixMul.cu
  | instlibs
    | env.mk
    | LICENSE
    | Makefile
    | src
       | branch.cu
       | memdiverge.cu
       | ...
    | utils
       | sassi-dictionary.hpp
       | ...
```

Figure 1: The directory structure of the SASSI project.

4. **Make sure you have a 346.41 driver or newer:** The CUDA 7 installation script can install a new driver for you that meets this requirement. If you already have a newer driver, that should be fine. You can test your driver version with the `nvidia-smi` command.

5. **Get the SASSI project**: You can get the SASSI project from GitHub via the following command:

   ```
   git clone https://github.com/NVlabs/SASSI
   ```

   The project that you get will be your `$SASSI_REPO`. The project contains several sample instrumentation libraries, documentation, and useful utilities.

6. **Compilers [Optional]**: To use the instrumentation libraries we provide, you will need a C++-11 compiler. We have been using GCC 4.8.2. For details about the host compiler that you can use on your platform for CUDA 7, consult the following guide:

   ```
   http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/
   ```

2.2 SASSI Project Structure

The SASSI project that is hosted on GitHub has the directory structure show in Figure 1. The `doc` directory contains this document and an ISCA paper that describes SASSI's inner workings.

The `example` directory contains a `Makefile` and a single application from the CUDA SDK, which this guide uses for the examples in Sections 7-10.

The `instlibs` directory contains sample SASSI instrumentation libraries that range in complexity from very simple to fairly complex. There are examples that profile branch behavior, memory behavior, and operand value behavior. In addition, there are a couple of pedagogical examples that demonstrate SASSI's API. We will continue to add instrumentation libraries to this directory over time. The `instlibs/utils` directory contains a templatized device-side hash map to facilitate bookkeeping.

2.3 Installation

We are using GitHub's "releases" functionality to release the closed-source portion of SASSI. Specifically, we are distributing SASSI as a self-extracting installer that installs SASSI's specialized version of `ptxas` and header files that describe SASSI's API. To get a binary installer for your architecture, go to:

`https://github.com/NVlabs/SASSI/releases`

Choose the installer for your architecture, download it, and execute it. The installer is simply a shell script. For example to install SASSI on an Ubuntu 15 x86-64 host, get the installer, `SASSI_x86_64_ubuntu_vivid.run`, then execute it:

```
sh SASSI_x86_64_ubuntu_vivid.run
```

The installation script performs the following very simple tasks:

1. **Has the end user agree to the End User License Agreement:** The license agreement is exactly the same as the EULA for the CUDA 7 toolkit.
2. **Asks the user for the location of the CUDA 7 toolkit:** This guide refers to this location as `$CUDA_HOME`.
3. **Asks the user where it should install SASSI**: Note, this directory should be different from your CUDA 7 location. Also note, you may need to run the script as root if you're planning on installing SASSI in the default location of `/usr/local/sassi`. This guide refers to this location as `$SASSI_HOME`.
4. **Copies the whole CUDA 7 directory to the SASSI install location:** This step can take a while depending on the details of your host and filesystem.
5. **Copies the SASSI header files to the SASSI install location.** The header files are the best documentation for SASSI's interface, so you should know that they are installed in `$SASSI_HOME/include/sassi/`
6. **Copies SASSI's `ptxas` to the SASSI install location.** More specifically, SASSI's `ptxas` is moved to `$SASSI_HOME/bin/`

Before going any further, you should make sure that the installation script succesfully installed SASSI. If there were no warnings, errors, or abort messages, it probably worked. Nevertheless, let's check. First, we'll make sure that the header files were installed in the right place:

```
ls -l $SASSI_HOME/include/sassi
total 64
-rw-r--r-- 1 mstephenson mstephenson 4759 Jul 17 08:05 sassi-branch.hpp
-rw-r--r-- 1 mstephenson mstephenson 11402 Jul 17 08:05 sassi-core.hpp
-rw-r--r-- 1 mstephenson mstephenson 2818 Jul 17 08:05 sassi.hpp
-rw-r--r-- 1 mstephenson mstephenson 3396 Jul 17 08:05 sassi-ins-properties.h
-rw-r--r-- 1 mstephenson mstephenson 3994 Jul 17 08:05 sassi-kernel.hpp
-rw-r--r-- 1 mstephenson mstephenson 4090 Jul 17 08:05 sassi-memory.hpp
-rw-r--r-- 1 mstephenson mstephenson 6499 Jul 17 08:05 sassi-opcodes.h
-rw-r--r-- 1 mstephenson mstephenson 10999 Jul 17 08:05 sassi-regs.hpp
-rw-r--r-- 1 mstephenson mstephenson 3636 Jul 17 08:05 sassi-types.h
```

Now we can check to see if the SASSI-enabled `ptxas` was installed. All SASSI instrumentation is turned off by default, so without providing any SASSI-specific options, SASSI's `ptxas` will behave like `ptxas` from your CUDA 7 installation. Issuing `ptxas -h` will print out all options available, including the optional arguments for instrumenting with SASSI. Let's see if the SASSI options are available:

```
$SASSI_HOME/bin/ptxas -h | grep sassi
--sassi-after-args <comma separated list of additional information> (-sassi-after-args)
--sassi-before-args <comma separated list of additional information> (-sassi-before-args)
```

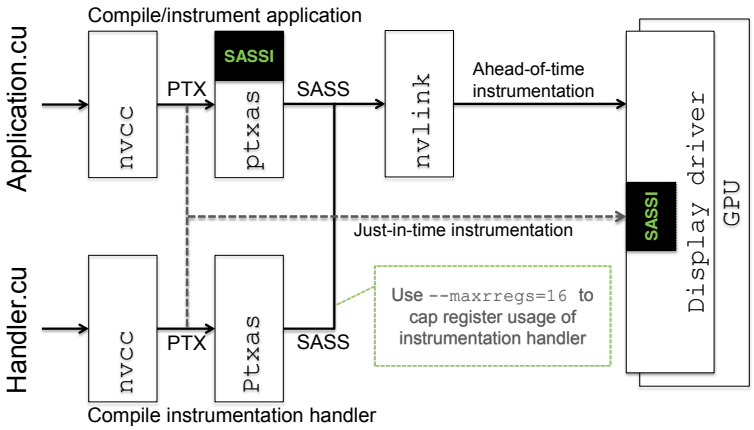

Figure 2: SASSI's instrumentation flow.

```
--sassi-debug (-sassi-debug)
...
```

Note: If at any point you upgrade or patch your version of CUDA 7, you should re-run the installation script. The SASSI installation script simply performs a deep copy of the CUDA 7 installation.

3 NVIDIA's architectures and compiler flow

This section provides background on basic GPU architecture terminology and the NVIDIA GPU compilation flow. Some of the examples later in this guide require a baseline level of knowledge about GPU architectures.

3.1 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). 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 of SMs.

3.2 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). SASSI currently only handles compute shaders. For compute shaders, a *front-end* compiler can be used to simplify the task of writing a shader. For example, a user can write parallel programs using high-level programming languages such as CUDA or OpenCL, 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. SASSI instrumentation relies on ahead-of-time compilation because it is embedded in `ptxas`.

Figure 2 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. When more than one module is compiled, the modules have to be linked together using `nvlink` before running on the GPU.

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 at some point in the future also be embedded in the driver to JIT compile PTX inputs, as shown by dotted lines in Figure 2.

4 Instrumenting an application

This section describes how to instrument an application. The high-level instrumentation flow is the same as the standard compilation flow shown in Figure 2. We use a SASSI-modified version of `ptxas` to instrument all of the modules in the GPGPU application. In addition, we define *what* instrumentation we are going to perform in a separate CUDA library, compile it separately, and link it in with the other modules in the program. There are some important requirements we impose on both the instrumented code and instrumentation libraries. We describe these requirements in Section 5.

A SASSI-modified version of `ptxas` accepts a few optional arguments not present in the default `ptxas`. All instrumentation intent is conveyed to SASSI via command-line arguments to `ptxas`. Section 11 lists all of the available command-line options for SASSI. Note that SASSI's user interface is currently simple, can be limiting, and is subject to change. In general, SASSI requires two pieces of information to instrument an application. First, we need to tell SASSI *where* it should insert instrumentation; and second, we need to tell SASSI *what* information it should extract for each instrumentation site.

4.1 Instrumentation sites (the *where*)

We will first describe the knobs that SASSI exposes to convey *where* it should insert instrumentation. SASSI has a few flags for guiding instrumentation sites, but the two most common are:

`--sassi-inst-before=`"⟨ list of instruction *classes* from Table 1⟩"
`--sassi-inst-after=`"⟨ list of instruction *classes* from Table 2⟩"

As the name implies `--sassi-inst-before` inserts instrumentation code before certain instructions, and `--sassi-inst-after` inserts code after instructions. The two flags take a comma-separated list of instrumentation sites.

Flag	Description
all	Inserts instrumentation before *all* instructions.
calls	Inserts instrumentation before function calls.
cond-branches	Inserts instrumentation before conditional branches.
memory	Inserts instrumentation before memory operations. SASSI does **not** consider "load constant"' to be mem
reg-reads	Inserts instrumentation before instructions that read registers.
reg-writes	Inserts instrumentation before instructions that write registers.

Table 1: Possible instrumentation sites for `--sassi-inst-before`.

In addition to the `--sassi-inst-before/after` flags, we can have SASSI insert instrumentation at kernel entry points with the flag,`--sassi-kernel-entry`. There are a couple of other instrumentation points that we will expose shortly in a future release.

Flag	Description
calls	Inserts instrumentation after function calls.
memory	Inserts instrumentation after memory operations. SASSI does **not** consider "load constant"' to be memory op
reg-writes	Inserts instrumentation after instructions that write registers.

Table 2: Possible instrumentation sites for `--sassi-inst-after`.

As an example, consider the case where we want to insert instrumentation before instructions that touch memory *or* instructions that write any registers, we would provide `ptxas` the argument,

`--sassi-inst-before="memory,reg-writes"`.

If you pass `--sassi-inst-before` or `--sassi-inst-after` invalid arguments, SASSI will abort the compilation with an error. The `--sassi-inst-after` option does not allow for instrumentation after any class of instructions that includes branches. This simplifies code generation and user expectations. Note, as Section 8 shows, it is still possible to determine where a branch *will* go using SASSI.

Let's compile an application and instruct SASSI to inject instrumentation before all memory operations:

```
nvcc gaussian.cu -gencode arch=compute_50,code=sm_50 \
  -Xptxas --sassi-inst-before=''memory'' -O3 -c -rdc=true -o gaussian.o -std=c++11
```

With this command, SASSI will spit out the following message:

```
********************************************************************************
*
*                       SASSI Instrumentation Details
*
*  For the settings you passed in, you'll need to make sure that you have
*  an instrumentation library with the following properties:
*  - It MUST BE compiled using only 16 registers!! To accomplish this
*    simply compile your library with the nvcc flag, --maxrregcount=16
*  - It must define the following functions:
*      __device__ void sassi_before_handler(SASSIBeforeParams*)
*
********************************************************************************
```

This message is a good segue to the next topic— *what* instrumentation gets inserted.

4.2 Instrumentation arguments (the *what*)

The instrumentation that SASSI injects is essentially a function call to a CUDA function that the user separately defines. So the SASSI commands we used last will insert a call to the following function before every memory operation:

```
__device__ void sassi_before_handler(SASSIBeforeParams*);
```

You can, with a couple of minor limitations that we discuss in Section 12, put whatever CUDA computation you want in `sassi_before_handler`, and it will execute before every memory operation executes.

SASSI assumes the user wants access to *baseline* information about each instrumented instruction, so it bundles up basic information about the instrumented instruction, places it in a C++ object of type `SASSIBeforeParams`, and passes along a pointer to the object as an argument to the handler. The argument type is shown in Figure 3. It allows us to test whether the instruction will execute (which it might not because NVIDIA's architectures feature predicated execution), get its opcode, get its virtual PC, etc.

```
class SASSIBeforeParams : public SASSICoreParams {
public:
  ////////////////////////////////////////////////////////////////////////////
  // Returns true if the instruction will execute.
  ////////////////////////////////////////////////////////////////////////////
  __device__ bool GetInstrWillExecute() const;

 ////////////////////////////////////////////////////////////////////////////
  // Gets the opcode of the instruction.
  ////////////////////////////////////////////////////////////////////////////
  __device__ SASSIInstrOpcodes GetOpcode() const;

  ////////////////////////////////////////////////////////////////////////////
  // Returns the offset of the instruction from the function's entry.
  ////////////////////////////////////////////////////////////////////////////
  __device__ uint32_t GetInsOffset() const;

  ////////////////////////////////////////////////////////////////////////////
  // Returns the name of the kernel/function that contains this instruction.
  ////////////////////////////////////////////////////////////////////////////
  __device__ const char *GetFnName() const;

  ////////////////////////////////////////////////////////////////////////////
  // Gets a probably unique virtual PC, which is repeatable between runs.
  ////////////////////////////////////////////////////////////////////////////
  __device__ uint64_t GetPUPC() const;

  ////////////////////////////////////////////////////////////////////////////
  // Returns the SHA1 hash of the function name.
  ////////////////////////////////////////////////////////////////////////////
  __device__ const uint32_t *GetFnNameSHA1() const;

  ////////////////////////////////////////////////////////////////////////////
  // Various ways to query whether the instruction is a memory operation.
  ////////////////////////////////////////////////////////////////////////////
  __device__ bool IsMem() const;
  __device__ bool IsMemRead() const;
  __device__ bool IsMemWrite() const;
  __device__ bool IsSpillOrFill() const;
  __device__ bool IsSurfaceMemory() const;

  ////////////////////////////////////////////////////////////////////////////
  // Returns true if the instruction type is a control transfer operation.
  ////////////////////////////////////////////////////////////////////////////
  __device__ bool IsControlXfer() const;
  __device__ bool IsConditionalControlXfer() const;

  // Other methods follow...
};
```

Figure 3: The baseline parameter that's passed in to an instrumentation handler allows the instrumentation handler to query the above methods.

Flag	Description
cond-branch-info	Extracts information about conditional branches and passes it to the instrumentation handler. *For instrumentation sites that do not correspond to a conditional branch instruction, the instrumentation handler will be passed a NULL pointer.* See `$SASSI_HOME/include/sassi/sassi-branch.hpp` for a description of the object type passed to the handler.
mem-info	Extracts information about memory operations and passes it to the instrumentation handler. *For instrumentation sites that do not correspond to memory instructions, the instrumentation handler will be passed a NULL pointer.* See `$SASSI_HOME/include/sassi/sassi-memory.hpp` for a description of the object type passed to the handler.
reg-info	Extracts information about the instruction's register usage. The pointer passed to the instrumentation handler should *always* point to a valid structure. See `$SASSI_HOME/include/sassi/sassi-regs.hpp` for a description of the object type passed to the handler.

Table 3: If the `--sassi-inst-before` flag is used, the user may also consider specifying additional information, using the `--sassi-before-args` flag, about the instrumentation site that will be passed to the handler.

While we can query basic characteristics with `SASSIBeforeParams` objects, this information is clearly insufficient for many studies. We can optionally specify additional information to extract for each instruction. If the user has specified that instrumentation should be inserted before some set of instructions (using the `--sassi-inst-before` flag), then we can optionally specify additional arguments to send to the handler by using the `--sassi-before-args` flag. This flag also takes a comma-separated list of values. Currently, allowed values for `--sassi-before-args` are 'cond-branch-info', 'mem-info', and 'reg-info'. Table 3 summarizes the functions of these values. Similarly, Table 4 summarizes the allowable options for "after" instrumentation.

When you specify additional arguments to be passed, SASSI will tell you the order it in which it will pass the arguments to the handler. It is important that you link in a handler that matches the type signature SASSI demands, or your link step will fail.

5 IMPORTANT: Required compiler flags

This section discusses some flags and constraints that that SASSI requires for proper operation. While the complete examples that we show in the next several sections will discuss these flags and constraints, it is important to highlight them here, because failure to follow this recipe will cause your instrumented programs to fail:

- You must compile your instrumentation handlers using only 16 registers: GPGPU programs often using a staggering number of registers. The SASSI approach of injecting function calls implies that live registers must be saved and restored across call sites. By forcing the instrumentation handlers to use only 16 registers, SASSI can drastically limit the number of saves and restores required across handler call boundaries. SASSI instrumentation will only save and restore the first 16 live registers, so if your handler uses registers outside this set, there's a good chance your instrumented program will break. This restriction only applies to the instrumentation handlers, not the programs you instrument– they can still use as many registers as the user deems necessary. We can trivially set the number of registers to 16 with the following `nvcc` flag: `--maxrregcount=16`.

Flag	Description
mem-info	Extracts information about memory operations and passes it to the instrumentation handler. *For instrumentation sites that do not correspond to memory instructions, the instrumentation handler will be passed a NULL pointer.* See `$SASSI_HOME/include/sassi/sassi-memory.hpp` for a description of the object type passed to the handler.
reg-info	Extracts information about the instruction's register usage. The pointer passed to the instrumentation handler should *always* point to a valid structure. See `$SASSI_HOME/include/sassi/sassi-regs.hpp` for a description of the object type passed to the handler.

Table 4: If the `--sassi-inst-after` flag is used, the user may also consider specifying additional information, using the `--sassi-after-args` flag, about the instrumentation site that will be passed to the handler.

- You must explicitly specify what *code* you want to generate: The CUDA toolchain allows users to generate two forms of code: virtual and real ISA code. In the case that you generate virtual ISA code (e.g., `compute_35`, `compute_50`), the display driver will just-in-time compile your code when it is loaded. This will not work for SASSI because SASSI is not in the driver's compiler. Instead, you must specify a real ISA, which always starts with an "sm" (e.g., `sm_35`, `sm_50`). This will cause the toolchain to precompile your code with `ptxas`. For example, if we are targeting an `sm_50`, I would probably supply the following option to `nvcc` to generate the correct code: `-gencode arch=compute_50,code=sm_50`. Please run `nvcc -h` for details about this option.
- You must generate position independent code with `-dc` (or equivalently, `-rcd=true`): SASSI instrumentation handlers are linked in with an instrumented application. To make "cross-module" calls with the CUDA toolchain, you must use this flag. While position independent code adds modularity to GPU programs and brings CUDA more in line with general purpose computing, it can affect the performance of the generated code, sometimes significantly. Future work will try to remove this requirement.

6 Setting up your environment for the examples

The next four sections walk users through complete examples. First, let's compile all the instrumentation handlers that come with SASSI, including the ones the next few sections cover. First, navigate to the instrumentation library directory:

```
1 cd $SASSI_REPO/instlibs
```

Next, we need to edit the file `env.mk`. There are four environment variables we need to set:

- `CUDA_HOME`: This is the location of your CUDA 7 root.
- `SASSI_HOME`: This is the location of your SASSI installation root (not the repository root).
- `CCBIN`: The location of the C++-11 compiler you want to use.
- `GENCODE`: What GPUs you want to compile your instrumentation libraries for. Remember you must specify a *real* architecture, not a virtual one.

After editing the file, you *should* be able to simply invoke `make` (in `$SASSI_REPO/instlibs`), which will compile the instrumentation libraries and install them in `$SASSI_REPO/instlibs/lib`. For users that are targetting Fermi cards (sm_20, sm_21), it is very important to note that not all of the instrumentation libraries will compile. Most of the instrumentation libraries use features that are only available in 3.0+

architectures. There is a special make target, `make fermi-only`, that will only make the instrumentation libraries that will run on Fermi.

The examples in the next several sections all rely heavily on NVIDIA's CUPTI library, so let's add it to our `LD_LIBRARY_PATH`:

```
export LD_LIBRARY_PATH=$SASSI_HOME/extras/CUPTI/lib64:$LD_LIBRARY_PATH
```

7 Example 1: Creating a histogram of opcodes

This simple example creates a histogram of the opcodes that are encountered during the execution of a program. As we have already mentioned, the compiler flow for SASSI instrumentation involves 1) compiling the application to enable callbacks to instrumentation handlers to be inserted, and 2) defining the instrumentation handler, compiling it, and linking it in with the application.

Before looking at these two steps in detail, let's just build and run the example. If you have not done so, please setup your environment per the instructions in Section 6. Now, do:

```
# Go to the example directory.
cd $SASSI_REPO/example
make clean
# Instrument the application and link with the instrumentation library.
make ophist
# Run the example. (Alternatively, type 'make run')
./matrixMul
```

This can take several seconds to minutes depending on the horsepower of your GPU. If all went well, you should now have a file in that directory, called `sassi-ophist.txt`, that contains a histogram of the encountered opcodes.

7.1 How to compile your application

Now let's take a look at how the makefile builds the `matrixMul` application. For this example, we want to track all instructions that execute, so we'll need to tell SASSI to add instrumentation callbacks before *all* instructions:

```
/usr/local/sassi7/bin/nvcc -I./inc -c -O3 \
  -gencode arch=compute_50,code=sm_50 \
  -Xptxas --sassi-inst-before=''all'' \
  -dc \
  -o matrixMul.o matrixMul.cu
```

Lines (1) and (5) are the options that you would ordinarily compiler your CUDA application with. Line (2) is required because we need the compiler to generate *actual* not virtual code. In this case, we are targetting a first-generation Maxwell card, with `sm_50`. Line (3) instructs SASSI to inject instrumentation before all SASS instructions, and line (4) is required because we are going to link in the instrumentation handler momentarily, and cross-module calls require "relocatable device code." In fact, let's look at the message that SASSI spits out:

```
*************************************************************************
*
*                       SASSI Instrumentation Details
*
*  For the settings you passed in, you'll need to make sure that you have
*  an instrumentation library with the following properties:
*  - It MUST BE compiled using only 16 registers!! To accomplish this
*    simply compile your library with the nvcc flag, --maxrregcount=16
```

```
*  - It must define the following functions:
*      __device__ void sassi_before_handler(SASSIBeforeParams*)
*
*********************************************************************************
```

This tells us that we have to define and link in a function with the signature, `__device__ void sassi_before_handler(SASSIBeforeParams*)`, which we have defined in a library called `libophist.a`. Don't worry, we'll eventually show you what's in the instrumentation library, and how we build it, but for now, let's just link it in:

```
1 /usr/local/sassi7/bin/nvcc -o matrixMul matrixMul.o \
2   -gencode arch=compute_50,code=sm_50 \
3   -lcudadevrt \
4   -L/usr/local/sassi7/extras/CUPTI/lib64 -lcupti \
5   -L../instlibs/lib -lophist
```

This example uses `nvcc` to link the instrumented application with the instrumentation handler and CUPTI. The instrumentation library, as we'll see soon, uses CUPTI to initialize on-device counters and copy them off the device before it is reset. Line (2) again specifies the NVIDIA architecture that we are targetting. Lines (3-5) link in the various required libraries for this application. Of note, line (4) links in CUPTI, and line (5) links in the instrumentation library, `ophist`.

It is a good idea to use `cuobjdump` to dump the code that generated, and make sure that your application was instrumented:

```
1 /usr/local/sassi/bin/cuobjdump -sass matrixMul
```

You should see *a lot* of code bloat! Each original instruction should be supplanted with a branch to an instrumentation trampoline.

7.2 Writing the instrumentation library

The instrumentation library that corresponds to this example is fully implemented in `$SASSI_REPO/instlibs/src/ophist.cu`. The library code is shown in Figure 4.

Notice, the handler, `sassi_before_handler` is a `__device__` function, which means it will run on the GPU. As you've seen, with the right instrumentation flags, we can get SASSI to inject calls to this function before every instruction.

This example is extremely simple, and only requires the basic information SASSI extracts about each instruction, which is passed in as an argument of type `SASSIBeforeParams*`. The handler first checks whether the instruction will execute with the `GetInstrWillExecute()` method. Note this is because an instruction might have a guarding predicate that is `false` for this invocation of the handler.

If the instruction will execute, the handler simply gets the instruction's opcode with `GetOpcode()` and then uses CUDA's `atomicAdd` function to increment the opcode's associated counter.

Even though there is a performance penalty (which our instrumentation overheads dwarf) for using Unified Virtual Memory, we find that UVM is quite convenient for writing instrumentation handlers. For this example, `dynamic_instr_counts[]` is declared as a `__managed__` type, which means that we can access it on the host *and* the device.

In `$SASSI_REPO/instlibs/utils` there is a convenience class, `sassi::lazy_allocator`, that allows us to initialize variables before the first kernel launch, and be notified before the program exits or the device is reset (so we can dump counters to a file, for instance). This class relies on CUPTI to respond to these events.

7.2.1 Building the handler as a library

The makefile in `$SASSI_REPO/instlibs/src` shows the standard way to build a library with the NVIDIA CUDA toolchain. The only way to instrument with SASSI involves creating a standalone library, or even an object file, and later linking it in with your SASSI-compiled application.

Let's take a look at what the makfile does for the `ophist` library:

```
#include <cupti.h>
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include "sassi_intrinsics.h"
#include "sassi_lazyallocator.hpp"
#include <sassi/sassi-core.hpp>

// Keep track of all the opcodes that were executed.
__managed__ unsigned long long dynamic_instr_counts[SASSI_NUM_OPCODES];

////////////////////////////////////////////////////////////////////////////
/// This is a SASSI handler that handles only basic information about each
/// instrumented instruction. The calls to this handler are placed by
/// convention *before* each instrumented instruction.
////////////////////////////////////////////////////////////////////////////
__device__ void sassi_before_handler(SASSIBeforeParams* bp)
{
   if (bp->GetInstrWillExecute())
   {
     SASSIInstrOpcode op = bp->GetOpcode();
     atomicAdd(&(dynamic_instr_counts[op]), 1ULL);
   }
}

////////////////////////////////////////////////////////////////////////////
/// Write out the statistics we've gathered.
////////////////////////////////////////////////////////////////////////////
static void sassi_finalize()
{
  cudaDeviceSynchronize();

  FILE *resultFile = fopen("sassi-ophist.txt", "w");
  for (unsigned i = 0; i < SASSI_NUM_OPCODES; i++) {
    if (dynamic_instr_counts[i] > 0) {
      fprintf(resultFile, "%-10.10s:␣%llu\n", SASSIInstrOpcodeStrings[i], dynamic_instr_counts[i]);
    }
  }
  fclose(resultFile);
}

////////////////////////////////////////////////////////////////////////////
/// Lazily initialize the counters before the first kernel launch.
////////////////////////////////////////////////////////////////////////////
static sassi::lazy_allocator counterInitializer(
  /* Initialize the counters. */
  []() {
    bzero(dynamic_instr_counts, sizeof(dynamic_instr_counts));
  }, sassi_finalize);
```

Figure 4: Instrumentation library for creating a histogram of opcodes.

```
/usr/local/cuda-7.0/bin/nvcc \
  -ccbin /usr/local/gcc-4.8.4/bin/ -std=c++11 \
  -O3 \
  -gencode arch=compute_50,code=sm_50 \
```

```
5   -c -rdc=true -o ophist.o \
6   ophist.cu \
7   --maxrregcount=16 \
8   -I/usr/local/sassi7/include \
9   -I/usr/local/sassi7/extras/CUPTI/include/ \
10  -I/home/mstephenson/Projects/sassi/instlibs/utils/ \
11  -I/home/mstephenson/Projects/sassi/instlibs/include
12 ar r /home/mstephenson/Projects/sassi/instlibs/lib/libophist.a ophist.o
13 ranlib /home/mstephenson/Projects/sassi/instlibs/lib/libophist.a
```

The instrumentation libraries in the instlibs rely on C++-11 support, hence line (2). We also build the instrumentation library to support the architecture we plan to run on in (4). Change this to match your architecture. Finally, of supreme importance, we instruction NVCC to use only 16 registers for the handler in (7). We include *include* paths in (8)-(11). Finally, lines (12) and (13) bundle the single object file into a library.

8 Example 2: Conditional Branch

This example examines the branch behavior of an application. We introduce a couple of features not required for the last example. First, we need to instruct SASSI to pass an additional argument to our instrumentation handler, one that conveys information about the behavior of conditional branches; and second, we use a hash map to record per-branch statistics.

Before walking through the specifics of the example, let's just build and run it. If you have not done so, please setup your environment per the instructions in Section 6. Now, do:

```
# Go to the example directory.
cd $SASSI_REPO/example
make clean
# Instrument the application and link with the instrumentation library.
make branch
# Run the example. (Alternatively, type 'make run')
./matrixMul
```

This can take some time to finish running. Assuming everything went well, you should now have a file in that directory called `sassi-branch.txt`. Here are the contents of the file for `sm_50`:

```
Address          Total/32   Dvrge/32   Active     Taken      NTaken
c23752810000000d0 1926400              61644800              61644800
c237528100000668 19264000             616448000  554803200  61644800
```

For `sm_50`, the simple application that we are testing this instrumentation library with has only one kernel, which only has two *conditional* branches (i.e., the branch direction depends on some statically unknown condition). The file separates out the branch statistics per conditional branch. The columns shown show 1) the number of branches encountered per warp; 2) the number of those branches that *diverge* (a condition where within a warp, some threads take the branch, and others fall through); 3) the total number of active threads that executed the branch; 4) the number of active threads that take the branch; 5) the number of active threads that fall through; 6) and the type of the branch. For complete coverage of this example, please see "Case Study I" in the accompanying ISCA paper in the `doc` directory.

8.1 How to compile your application

Now let's take a look at how the makefile builds the `matrixMul` application for this this example. For this example, we are only interested in conditional branches, so while we could still add instrumentation before all instructions, it will be more efficient to limit our instrumentation to only conditional branches:

```
/usr/local/sassi7/bin/nvcc -I./inc -c -O3 \
  -gencode arch=compute_50,code=sm_50 \
  -Xptxas --sassi-inst-before=''cond-branches'' \
  -Xptxas --sassi-before-args=''cond-branch-info'' \
  -dc \
  -o matrixMul.o matrixMul.cu
```

Lines (1) and (6) are the options that you would ordinarily compiler your CUDA application with. Line (2) is required because we need the compiler to generate *actual* not virtual code. In this case, we are targetting a first-generation Maxwell card, with `sm_50`. Line (3) instructs SASSI to inject instrumentation before conditional branch SASS instructions only. The instrumentation library for this example needs to know the direction that the conditional branch will take (e.g., TAKEN or NOT TAKEN), and this information is available in objects of `SASSICondBranchParams`. The `cond-branch-info` flag will direct SASSI to pass such objects to the instrumentation handler. This is done in line (4). Line (5) is required because we are going to link in the instrumentation handler momentarily, and cross-module calls require "relocatable device code." Here is the message that SASSI emits for this example:

```
********************************************************************************
*
*                        SASSI Instrumentation Details
*
*  For the settings you passed in, you'll need to make sure that you have
*  an instrumentation library with the following properties:
*  - It MUST BE compiled using only 16 registers!! To accomplish this
*    simply compile your library with the nvcc flag, --maxrregcount=16
*  - It must define the following functions:
*      __device__ void sassi_before_handler(SASSIBeforeParams*,SASSICondBranchParams*)
*
********************************************************************************
```

Notice that SASSI tells us that we have to make sure to link in a handler with this signature:

```
__device__ void sassi_before_handler(SASSIBeforeParams*,SASSICondBranchParams*);
```

which we have defined in a library called `libbranch.a`, which we'll describe shortly. We link this in exactly like we did with the last example:

```
1 /usr/local/sassi7/bin/nvcc -o matrixMul matrixMul.o \
2   -gencode arch=compute_50,code=sm_50 \
3   -lcudadevrt \
4   -L/usr/local/sassi7/extras/CUPTI/lib64 -lcupti \
5   -L../instlibs/lib -lbranch
```

The only difference is linking in `libbranch.a` versus `libophist.a`.

8.2 Writing the instrumentation library

The instrumentation library that corresponds to this example is fully implemented in `$SASSI_REPO/instlibs/src/branch.cu`. The instrumentation handler portion of the library is shown in Figure 5. In the last example, we first checked to see if the instruction would execute using `bp->GetInstrWillExecute()`. We don't do that here, because the branch instruction's guarding predicate is often used to determine the branch direction.

The handler first determines the thread index within the warp (line 9), the bitmask of active threads (line 14), and the direction in which the thread is going to branch (line 15). CUDA provides several warp-wide broadcast and reduction operations that NVIDIA's architectures efficiently support. For example, many of the handlers we write 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."[CUDA Programming Guide].

The handler also uses `__ballot` on lines 16 and 17 to set masks corresponding to threads that are going to take the branch (`taken`), and the threads that are going to fall through (`ntaken`). With these masks, the handler uses the *population count* instruction (`__popc`) to efficiently determine the number of threads in each respective category (`numActive`, `numTaken`, `numNotTaken`).

On line 24 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 30 finds the hash table entry for the instrumented branch (using the templatized hash table in `$SASSI_REPO/instlibs/utils`). Lines 42-46 update the counters.

Under the covers, this library relies on the CUPTI library to register callbacks for kernel *launch* and *exit* events. 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. Please see the source code of the instrumentation handler for full details of the example.

8.2.1 Building the handler as a library

We build the library exactly the way we did for the `ophist` library.

```
////////////////////////////////////////////////////////////////////////////////
//
/// This function will be inserted before every conditional branch instruction.
//
////////////////////////////////////////////////////////////////////////////////
__device__ void sassi_before_handler(SASSIBeforeParams *bp, SASSICondBranchParams *brp)
{
  // Find out thread index within the warp.
  int threadIdxInWarp = get_laneid();

  // Get masks and counts of 1) active threads in this warp,
  // 2) threads that take the branch, and
  // 3) threads that do not take the branch.
  int active = __ballot(1);
  bool dir = brp->GetDirection();
  int taken = __ballot(dir == true);
  int ntaken = __ballot(dir == false);
  int numActive = __popc(active);
  int numTaken = __popc(taken);
  int numNotTaken = __popc(ntaken);
  bool divergent = (numTaken != numActive && numNotTaken != numActive);

  // The first active thread in each warp gets to write results.
  if ((__ffs(active)-1) == threadIdxInWarp) {
    // Get the address, we'll use it for hashing.
    uint64_t inst_addr = bp->GetPUPC();

    // Looks up the counters associated with 'inst_addr', but if no such entry
    // exits, initialize the counters in the lambda.
    BranchCounter *stats = (*sassi_stats).getOrInit(inst_addr, [inst_addr,brp](BranchCounter* v) {
        v->address = inst_addr;
        v->branchType = brp->GetType();
        v->taggedUnanimous = brp->IsUnanimous();
      });

    // Why not sanity check the hash map?
    assert(stats->address == inst_addr);
    assert(numTaken + numNotTaken == numActive);

    // Increment the various counters that are associated
    // with this instruction appropriately.
    atomicAdd(&(stats->totalBranches), 1ULL);
    atomicAdd(&(stats->activeThreads), numActive);
    atomicAdd(&(stats->takenThreads), numTaken);
    atomicAdd(&(stats->takenNotThreads), numNotTaken);
    atomicAdd(&(stats->divergentBranches), divergent);
  }
}
```

Figure 5: Instrumentation handler portion of the conditional branch behavior profiling library. See the library's source code for the full example.

```
/usr/local/cuda-7.0/bin/nvcc \
  -ccbin /usr/local/gcc-4.8.4/bin/ -std=c++11 \
  -O3 \
  -gencode arch=compute_50,code=sm_50 \
```

```
5   -c -rdc=true -o branch.o \
6   branch.cu \
7   --maxrregcount=16 \
8   -I/usr/local/sassi7/include \
9   -I/usr/local/sassi7/extras/CUPTI/include/ \
10  -I/home/mstephenson/Projects/sassi/instlibs/utils/ \
11  -I/home/mstephenson/Projects/sassi/instlibs/include
12 ar r /home/mstephenson/Projects/sassi/instlibs/lib/libbranch.a branch.o
13 ranlib /home/mstephenson/Projects/sassi/instlibs/lib/libbranch.a
```

The instrumentation libraries in the `instlibs` directory rely on C++-11 support, hence line (2). We also build the instrumentation library to support the architecture we plan to run on in (4). Change this to match your architecture. Finally, of supreme importance, we instruction NVCC to use only 16 registers for the handler in (7). We include *include* paths in (8)-(11). Finally, lines (12) and (13) bundle the single object file into a library.

9 Example 3: Memory Divergence

This example, as with the previous example, performs *before* instrumentation, and requests that SASSI pass an extra parameter to the instrumentation handler. However, in this case, we inject instrumentation before operations that touch (read or write) memory, and we pass along information about instruction memory usage.

Before walking through the specifics of the example, let's just build and run it. If you have not done so, please setup your environment per the instructions in Section 6. Now, do:

```
# Go to the example directory.
cd $SASSI_REPO/example
make clean
# Instrument the application and link with the instrumentation library.
make memdiverge
# Run the example. (Alternatively, type 'make run')
./matrixMul
```

This can take some time to finish running. Assuming everything went well, you should now have a file in that directory called `sassi-memdiverge.txt`. The memory behavior of the sample application is so regular that these results are fairly boring. For `sm_50`, these results show that for every load and store to *global* memory, all 32 threads in a warp were active, and within each warp, four cache lines were touched. The rationale for printing out a matrix will become more clear later in this section. For complete coverage of this example, please see "Case Study II" in the accompanying ISCA paper in the `doc` directory.

9.1 How to compile your application

Now let's take a look at how the makefile builds the `matrixMul` application for this this example. For this example, we are only interested in memory operations, so while we could still add instrumentation before all instructions, it will be more efficient to limit our instrumentation to only memory operations:

```
/usr/local/sassi7/bin/nvcc -I./inc -c -O3 \
  -gencode arch=compute_50,code=sm_50 \
  -Xptxas --sassi-inst-before=''memory'' \
  -Xptxas --sassi-before-args=''mem-info'' \
  -dc \
  -o matrixMul.o matrixMul.cu
```

Lines (1) and (6) are the options that you would ordinarily compiler your CUDA application with. Line (2) is required because we need the compiler to generate *actual* not virtual code. In this case, we are targetting a first-generation Maxwell card, with `sm_50`. Line (3) instructs SASSI to inject instrumentation before SASS instructions that touch memory only. It is worth noting that SASSI does not consider loads from the constant memory window to be memory operations. The instrumentation library for this example needs to know the address that the memory operation targets, and this information is available in objects of `SASSIMemoryParams`. The `mem-info` flag will direct SASSI to pass such objects to the instrumentation handler. This is done in line (4). Line (5) is required because we are going to link in the instrumentation handler momentarily, and cross-module calls require "relocatable device code." Here is the message that SASSI emits for this example:

```
********************************************************************************
*
*                       SASSI Instrumentation Details
*
*  For the settings you passed in, you'll need to make sure that you have
*  an instrumentation library with the following properties:
*  - It MUST BE compiled using only 16 registers!! To accomplish this
*    simply compile your library with the nvcc flag, --maxrregcount=16
```

```
*  - It must define the following functions:
*      __device__ void sassi_before_handler(SASSIBeforeParams*,SASSIMemoryParams*)
*
*********************************************************************************
```

Notice that SASSI tells us that we have to make sure to link in a handler with this signature:

```
__device__ void sassi_before_handler(SASSIBeforeParams*,SASSIMemoryParams*);
```

which we have defined in a library called `libmemdiverge.a`, which we'll describe shortly. We link this in exactly like we did with the last example:

```
/usr/local/sassi7/bin/nvcc -o matrixMul matrixMul.o \
  -gencode arch=compute_50,code=sm_50 \
  -lcudadevrt \
  -L/usr/local/sassi7/extras/CUPTI/lib64 -lcupti \
  -L../instlibs/lib -lmemdiverge
```

9.2 Writing the instrumentation library

The instrumentation library that corresponds to this example is fully implemented in `$SASSI_REPO/instlibs/src/memdiverge.cu`. The instrumentation handler portion of the library is shown in Figure 6. This example aims to create a 32×32 matrix, where the rows of the matrix tally the number of threads that were active for each warp's invocation of a memory operation, and the columns tally the number of cache lines touched by the memory operation. The code then, simply determines for each memory operation, 1) how many threads in the warp are active (`numActive`), and 2) how many unique cache lines are touched by the active threads (`unique`). At the end of the handler on line (44), we simply increment the counter associated with `numActive` and `unique`.

On line (13), we get the address of the memory operation using the `GetAddress()` method of the `SASSIMemoryParams` object passed in. On line (15), we filter out all memory accesses except global memory requests (i.e., we don't consider *shared* or *local* requests), though it would be trivial to consider other memory windows instead. On line (20), we find out the cache line to which the address maps.

Line (22) uses `__ballot()`, a commonly used warp-collective CUDA function, to determine how many threads in the warp are actively participating. Line (24) uses `__popc()`, another built-in CUDA function, to count the number of set bits and report the number of active threads. The while loop in (25)-(26) uses `__ffs()`, `__ballot()`, and `__broadcast()` to determine the number of unique values stored in the `lineAddr` variable across the warp. Every thread in the warp computes the same value for `unique`, but we only let the first active theard commit the results on line (43).

```
/// The counters that we will use to record our statistics.
__managed__ unsigned long long sassi_counters[WARP_SIZE + 1][WARP_SIZE + 1];

////////////////////////////////////////////////////////////////////////////
///
/// This is the function that will be inserted before every memory operation.
///
////////////////////////////////////////////////////////////////////////////
__device__ void sassi_before_handler(SASSIBeforeParams *bp, SASSIMemoryParams *mp)
{
  if (bp->GetInstrWillExecute())
  {
    intptr_t addrAsInt = mp->GetAddress();
    // Don't look at shared or local memory.
    if (__isGlobal((void*)addrAsInt)) {
      // The number of unique addresses across the warp
      unsigned unique = 0; // for the instrumented instruction.

      // Shift off the offset bits into the cache line.
      intptr_t lineAddr = addrAsInt >> LINE_BITS;

      int workset = __ballot(1);
      int firstActive = __ffs(workset) - 1;
      int numActive = __popc(workset);
      while (workset) {
        // Elect a leader, get its line, see who all matches it.
        int leader = __ffs(workset) - 1;
        intptr_t leadersAddr = __broadcast<intptr_t>(lineAddr, leader);
        int notMatchesLeader = __ballot(leadersAddr != lineAddr);

        // We have accounted for all values that match the leader's.
        // Let's remove them all from the workset.
        workset = workset & notMatchesLeader;
        unique++;
        assert(unique <= 32);
      }

      assert(unique > 0 && unique <= 32);

      // Each thread independently computed 'numActive' and 'unique'.
      // Let's let the first active thread actually tally the result.
      int threadsLaneId = get_laneid();
      if (threadsLaneId == firstActive) {
        atomicAdd(&(sassi_counters[numActive][unique]), 1LL);
      }
    }
  }
}
```

Figure 6: Instrumentation handler portion of the memory divergence library. See the library's source code for the full example.

9.2.1 Building the handler as a library

We build the library exactly the way we did for the `ophist` library.

```
/usr/local/cuda-7.0/bin/nvcc \
```

```
2   -ccbin /usr/local/gcc-4.8.4/bin/ -std=c++11 \
3   -O3 \
4   -gencode arch=compute_50,code=sm_50 \
5   -c -rdc=true -o memdiverge.o \
6   memdiverge.cu \
7   --maxrregcount=16 \
8   -I/usr/local/sassi7/include \
9   -I/usr/local/sassi7/extras/CUPTI/include/ \
10  -I/home/mstephenson/Projects/sassi/instlibs/utils/ \
11  -I/home/mstephenson/Projects/sassi/instlibs/include
12 ar r /home/mstephenson/Projects/sassi/instlibs/lib/libmemdiverge.a memdiverge.o
13 ranlib /home/mstephenson/Projects/sassi/instlibs/lib/libmemdiverge.a
```

The instrumentation libraries in the `instlibs` directory rely on C++-11 support, hence line (2). We also build the instrumentation library to support the architecture we plan to run on in (4). Change this to match your architecture. Finally, of supreme importance, we instruction NVCC to use only 16 registers for the handler in (7). We include *include* paths in (8)-(11). Finally, lines (12) and (13) bundle the single object file into a library.

10 Example 4: Value Profiling

This example is the most involved example, and performs fairly obtrusive instrumentation to profile the values assigned to the general-purpose registers during the execution of a program. This example uses *after* instrumentation to inject instrumentation after every instruction that writes an ISA-visible register. We then instruct SASSI to extract and pass along information about each SASS instruction's register usage, which our handler uses to inspect register values.

As with our other example, before walking through the specifics of the example, let's just build and run it. If you have not done so, please setup your environment per the instructions in Section 6.

```
# Go to the example directory.
cd $SASSI_REPO/example
make clean
# Instrument the application and link with the instrumentation library.
make valueprof
# Run the example. (Alternatively, type 'make run')
./matrixMul
```

This can take a really long time to finish running because the slowdowns of instrumentation for this example are fairly severe (1000×). Assuming everything went well, you should now have a file in that directory called `sassi-valueprof.txt`. Here are the abridged contents of the file for `sm_50`:

```
Value profiling results
ADDRESS | WEIGHT | [regnum, type, scalarness, bitstring]*
-----------------------------------------------------------
[c237528100000008, 61644800, [[1, ``uint'', SCALAR , [00000000111111111111101110100000]]]]
[c237528100000010, 61644800, [[16, ``uint'', SCALAR , [00000000000000000000000000000TTTT]]]]
[c237528100000018, 61644800, [[2, ``uint'', SCALAR , [00000000000000000000000000000TTTTT]]]]
...
[c237528100000208, 616448000, [[5, ``int'', SCALAR , [00000000000000000000000000000101]]]]
[c237528100000210, 616448000, [[4, ``float'', SCALAR , [00111100001000111101011100001010]]]]
[c237528100000238, 616448000, [[26, ``float'', SCALAR , [00111100001000111101011100001010]]]]
...
```

The results dump the general-purpose register usage of each SASS instruction. For the first instruction above, we see that it writes to register `1`, which is of type `uint`, and it always writes the same constant `[00000000111111111111101110100000]`. Because it writes a constant, it is also marked as *scalar*, which means every participating thread is writing the same value. The next instruction writes to register `16`. It's writes are always *scalar* too, but slightly more interestingly it does not write a constant. Instead, as the `T` represents, the bottom four bits assume `0` and `1`. The top 28 bits are still always `0`, however. Looking at the CUDA source, we see that the matrices in this example are artificially initialized to constant values, leading to the very high number of scalar and constant bits. For complete coverage of this example, please see "Case Study III" in the accompanying ISCA paper in the `doc` directory.

10.1 How to compile your application

Now let's take a look at how the makefile builds the `matrixMul` application for this this example. For this example, we instrument after all instructions that write registers.

```
/usr/local/sassi7/bin/nvcc -I./inc -c -O3 \
  -gencode arch=compute_50,code=sm_50 \
  -Xptxas --sassi-inst-after=''reg-writes'' \
  -Xptxas --sassi-after-args=''reg-info'' \
  -Xptxas --sassi-iff-true-predicate-handler-call \
  -dc \
  -o matrixMul.o matrixMul.cu
```

Lines (1) and (7) are the options that you would ordinarily compiler your CUDA application with. Line (2) is required because we need the compiler to generate *actual* not virtual code. In this case, we are targetting a first-generation Maxwell card, with `sm_50`. Line (3) instructs SASSI to inject instrumentation *after* SASS instructions that write registers. The instrumentation library for this example needs to know what registers are used along with the values in the registers, which it can obtain in objects of type `SASSIRegisterParams`. On line (4), the `reg-info` flag will direct SASSI to pass such objects to the instrumentation handler. Line (5) directs SASSI to only call the instrumentation handler if the instruction is going to execute (i.e., the instruction is unpredicated, or its guarding predicate is *true*). For *before* instrumentation, we can easily use the `GetInstrWillExecute()` method to determine whether the instruction will execute. This is trickier for *after* instrumentation, where the instruction's guarding predicate could have been clobbered by the instruction itself. For this example, we only care about instructions that executed, so this is the right option to use.

Line (6) is required because we are going to link in the instrumentation handler momentarily, and cross-module calls require "relocatable device code." Please note, that the SASSI flags on lines (3) and (4) both contain the word *after*, unlike in the previous examples. Here is the message that SASSI emits for this example:

```
********************************************************************************
*
*                       SASSI Instrumentation Details
*
*  For the settings you passed in, you'll need to make sure that you have
*  an instrumentation library with the following properties:
*  - It MUST BE compiled using only 16 registers!! To accomplish this
*    simply compile your library with the nvcc flag, --maxrregcount=16
*  - It must define the following functions:
*      __device__ void sassi_after_handler(SASSIAfterParams*,SASSIRegisterParams*)
*
********************************************************************************
```

Notice that SASSI tells us that we have to make sure to link in a handler with this signature:

```
__device__ void sassi_after_handler(SASSIAfterParams*,SASSIRegisterParams*);
```

which we have defined in a library called `libvalueprof.a`, which we'll describe shortly. We link this in exactly like we did with the last example:

```
1 /usr/local/sassi7/bin/nvcc -o matrixMul matrixMul.o \
2   -gencode arch=compute_50,code=sm_50 \
3   -lcudadevrt \
4   -L/usr/local/sassi7/extras/CUPTI/lib64 -lcupti \
5   -L../instlibs/lib -lvalueprof
```

10.2 Writing the instrumentation library

The instrumentation library that corresponds to this example is fully implemented in `$SASSI_REPO/instlibs/src/valueprof.cu`. The instrumentation handler portion of the library is shown in Figure 7. Note, that because we specified the `--sassi-iff-true-predicate-handler-call` flag, this handler will only be called if the instruction was actually executed.

Line (9) gets the lane in the warp that the current thread is running on. Line (10) finds the first active thread– this thread is going to be the one that all other threads compare themselves against when looking for *scalar* values. Line (13) gets the probably unique virtual PC of the SASS instruction. We use this PC to index into a hashmap on line (18). If the PC isn't in the hashmap already, it is added and initialized on line (19). Thus, on line (22), we will have an initialized statistics counter for the instruction. Line (23) simply increments a counter we maintain for this PC to track how many total times it has executed.

Line (24) iterates over all the destination registers in the SASS instruction. For each destination:

- Line 26 retrieves information about the register, including type information and register number.
- Line 27 gets the 32-bit value in the register.
- Line 30 uses CUDA's `atomicAnd` operation to keep track of the bits that are 1. If a bit in the operand ever becomes 0 at any point during the program, the `atomicAnd` is going to sticky set it to 0.
- Line 31 uses CUDA's `atomicAnd` operation to keep track of the bits that are 0. Note the bitwise negation of bits, "∼". If a bit in the operand ever becomes 1 at any point during the program, the `atomicAnd` is going to sticky set it to 0.
- Lines (33)-(38) check and record the scalarness of the value.

```
1  ////////////////////////////////////////////////////////////////////////////
2  //
3  // This example uses the atomic bitwise operations to keep track of the constant
4  // bits produced by each instruction.
5  //
6  ////////////////////////////////////////////////////////////////////////////
7  __device__ void sassi_after_handler(SASSIAfterParams* ap, SASSIRegisterParams *rp)
8  {
9    int threadIdxInWarp = get_laneid();
10   int firstActiveThread = (__ffs(__ballot(1))-1); /*leader*/
11
12   // Get the "probably unique" PC.
13   uint64_t pupc = ap->GetPUPC();
14
15   // The dictionary will return the SASSOp associated with this PC, or insert
16   // it if it does not exist. If it does not exist, the lambda passed as
17   // the second argument to getOrInit is used to initialize the SASSOp.
18   SASSOp *stats = sassi_stats->getOrInit(pupc, [&rp](SASSOp *v) {
19       SASSOp::init(v, rp);
20     });
21
22   // Record the number of times the instruction executes.
23   atomicAdd(&(stats->weight), 1);
24   for (int d = 0; d < rp->GetNumGPRDsts(); d++) {
25     // Get the value in each destination register.
26     SASSIRegisterParams::GPRRegInfo regInfo = rp->GetGPRDst(d);
27     SASSIRegisterParams::GPRRegValue regVal = rp->GetRegValue(ap, regInfo);
28
29     // Use atomic AND operations to track constant bits.
30     atomicAnd(&(stats->operands[d].constantOnes), regVal.asInt);
31     atomicAnd(&(stats->operands[d].constantZeros), ~regVal.asInt);
32
33     int leaderValue = __shfl(regVal.asInt, firstActiveThread);
34     int allSame = (__all(regVal.asInt == leaderValue) != 0);
35     // The warp leader gets to write results.
36     if (threadIdxInWarp == firstActiveThread) {
37       atomicAnd(&(stats->operands[d].isScalar), allSame);
38     }
39   }
40 }
```

Figure 7: Instrumentation handler portion of the value profiling library. See the library's source code for the full example.

10.2.1 Building the handler as a library

We build the library exactly the way we did for the `ophist` library.

```
1  /usr/local/cuda-7.0/bin/nvcc \
2    -ccbin /usr/local/gcc-4.8.4/bin/ -std=c++11 \
3    -O3 \
4    -gencode arch=compute_50,code=sm_50 \
5    -c -rdc=true -o valueprof.o \
6    valueprof.cu \
7    --maxrregcount=16 \
8    -I/usr/local/sassi7/include \
9    -I/usr/local/sassi7/extras/CUPTI/include/ \
10   -I/home/mstephenson/Projects/sassi/instlibs/utils/ \
11   -I/home/mstephenson/Projects/sassi/instlibs/include
12 ar r /home/mstephenson/Projects/sassi/instlibs/lib/libvalueprof.a valueprof.o
13 ranlib /home/mstephenson/Projects/sassi/instlibs/lib/libvalueprof.a
```

The instrumentation libraries in the `instlibs` directory rely on C++-11 support, hence line (2). We also build the instrumentation library to support the architecture we plan to run on in (4). Change this to match your architecture. Finally, of supreme importance, we instruction NVCC to use only 16 registers for the handler in (7). We include *include* paths in (8)-(11). Finally, lines (12) and (13) bundle the single object file into a library.

11 Other command-line arguments for instrumentation

This section describes a few additional flags that guide how instrumentation is performed.

11.1 –sassi-debug

This flag performs two functions. First, it inserts canary values on the stack before calling a handler function, then upon return, it performs a check to make sure the canary values are intact. If the canary values differ from expectation, the check traps. This option is meant as a sanity check to help guard against unknown stack corruption. (Because SASSI manipulates the stack.)

This flag can also be used to selectively disable some instrumentation points. When you assemble a PTX file with this option enabled, SASSI will dump the SASS that was generated to file named `sassi-instrumented-<PID>.txt`, and it will show you where the instrumentation was inserted. For example, look at this snippet from the example code, with the branch example (Section 8):

```
      .
      .
      .
# BAR
# ==== BASIC BLOCK HEADER ====
# IADD
# IADD
# ISETP
+_Z13matrixMulCUDAILi16EEvPfS0_S0_ii, 111, _Z20sassi_before_handlerP17SASSIBeforeParams...
# BRA
# ==== BASIC BLOCK HEADER ====
# SYNC
      .
      .
      .
```

The dump shows the instructions that were processed by SASSI, including the ones around which SASSI added instrumentation. The lines that begin with "+", show where instrumentation was inserted. The tuple on each "+" line shows the kernel, the unique function ID of the instrumented instruction, and the handler that was called for each instrumentation site. So, the first "+" line shows that a call to `sassi_before_handler` was inserted before the `BRA` instruction.

If you take this output and put it in a file called `sassi-skips.txt`, you can selectively disable instrumentation for certain points by changing the "+" to a "-". This selective skipping of instrumentation points is only enabled in `--sassi-debug` mode. This simple interface has allowed us to binary search for problems with SASSI's code generation. But users may find it useful, though tedious, for finer-grained control over instrumentation sites.

11.2 –sassi-inst-inline

By default SASSI adds instrumentation via a code trampoline, which better preserves the original code layout, i.e., the offset of each instruction from the beginning of the function/kernel. More specifically, SASSI replaces the instrumented instruction with jump to an instrumentation trampoline, which performs the instrumentation, executes the instruction, and if necessary, branches back to the next instruction to be executed. By setting this option to true, SASSI will instead forgo a code trampoline, and inject the instrumentation directly inline.

11.3 –sassi-iff-true-predicate-handler-call

NVIDIA GPUs support predication. Most instructions can have a boolean source argument that dictates whether or not that instruction will actually execute. If this predicate operand is false, should the instrumentation handler be called? By default, the handler is always called, regardless of the predicate's value.

For "before" instrumentation, the handler can determine whether or not the instruction *will execute*; in the case of "after" instrumentation, the instrumentation handler can inspect the value of the guarding predicate.

With this flag enabled, the handler will be called if and only if the instruction is actually executed.

12 What can go in an instrumentation handler?

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 stack, 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.

We should also point out that designing instrumentation handlers for SASSI requires the user to carefully consider synchronization and data sharing. Writing parallel programs is tricky business, and SASSI instrumentation code is highly parallel by design.

13 Discussion

SASSI has some clear shortcomings. First, as we mentioned in the introduction, it is not a binary instrumenter, so source code is required. Furthermore, the additional complexity of the build process has proven to be fairly difficult for some users. A true binary instrumentation approach like that of `cuda-memcheck` would make the process seamless. On the other hand, there are some advantages: we are investigating porting the tool to the driver, which would allow us to handle graphics applications (which are JITted)[2]; in addition, SASSI can gather potentially useful properties that the compiler knows about such as register liveness, operand datatypes, basic block boundaries, etc. Related to this, the instrumentation code should be much less intrusive than that of binary patching.

14 Bug Reports

We plan to track issues using GitHub's issue tracking features.

[2]For this case, it does not make sense to have the driver compile the application, then disassemble it, patch it, then re-apply.