NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs

Oreste Villa
NVIDIA
ovilla@nvidia.com

Mark Stephenson
NVIDIA
mstephenson@nvidia.com

David Nellans
NVIDIA
dnellans@nvidia.com

Stephen W. Keckler
NVIDIA
skeckler@nvidia.com

ABSTRACT

Binary instrumentation frameworks are widely used to implement profilers, performance evaluation, error checking, and bug detection tools. While dynamic binary instrumentation tools such as PIN and DynamoRio are supported on CPUs, GPU architectures currently only have limited support for similar capabilities through static compile-time tools, which prohibits instrumentation of dynamically loaded libraries that are foundations for modern high-performance applications. This work presents NVBit, a fast, dynamic, and portable, binary instrumentation framework, that allows users to write instrumentation tools in CUDA/C/C++ and selectively apply that functionality to pre-compiled binaries and libraries executing on NVIDIA GPUs. Using dynamic recompilation at the SASS level, NVBit analyzes GPU kernel register requirements to generate efficient ABI compliant instrumented code without requiring the tool developer to have detailed knowledge of the underlying GPU architecture. NVBit allows basic-block instrumentation, multiple function injections to the same location, inspection of all ISA visible state, dynamic selection of instrumented or uninstrumented code, permanent modification of register state, source code correlation, and instruction removal. NVBit supports all recent NVIDIA GPU architecture families including Kepler, Maxwell, Pascal and Volta and works on any pre-compiled CUDA, OpenACC, OpenCL, or CUDA-Fortran application.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Dynamic compilers**; **Just-in-time compilers**.

KEYWORDS

Dynamic binary instrumentation, CUDA, GPGPU, GPU computing

ACM Reference Format:
Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358307

In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3352460.3358307

1 INTRODUCTION

Binary instrumentation is a powerful technique that allows transformation of application binaries for a variety of purposes, including application profiling, performance modeling, error checking, and fault injection. Binary instrumentation is also one of the backbones of modern computer architecture research and education [33]. Binary instrumentation *frameworks* are used to develop custom *instrumentation tools* which apply specific transformations to application binaries that are then used for unique studies. Notable examples of binary instrumentation frameworks that target CPU architectures include ATOM [35], Intel Pin [12, 19], DynamoRIO [2], and HP Caliper [13]. All these frameworks employ some form of dynamic binary rewriting without needing application source code.

With the advent of GPU computing, equivalent frameworks are desirable to target GPU architectures. Compiler-based instrumentation tools, such as SASSI [36] [32] and Ocelot [8], have helped fill this void for NVIDIA GPUs and the CUDA programming model [5]. For example, SASSI exposes a rich API for ease of use, allowing end users to write custom *instrumentation tools* that can be used to characterize applications and explore the GPU's architectural design space [9]. However compiler-based instrumentation tools can have significant practical limitations:

- They require source code recompilation which is problematic if applications are large or the source code is not available.
- They are tied to a specific compiler (or compiler version) precluding the use of alternative toolchains for the target application.
- They cannot target code generated by the GPU driver, via JIT-compilation of PTX [30] code.
- They cannot target proprietary accelerated libraries such as cuBLAS, cuFFT, cuSOLVER, and cuDNN [29] for which the source code is not publicly available. Popular machine learning frameworks such as Caffe [14] and Torch7 [4] make heavy use of cuBLAS and cuDNN.
- They require compile time selection of the instrumentation sites, functions, and parameter types; and therefore they cannot adapt the instrumentation based on dynamic application behavior.

Compiler-based approaches provide a stopgap solution for GPU instrumentation but they do not provide the comparable power of dynamic instrumentation that is available on CPUs. To the best of

our knowledge, no general purpose binary instrumentation tool for GPUs is publicly available today.

This paper presents NVBit a dynamic binary instrumentation framework targeting NVIDIA GPUs. NVBit provides a rich set of high level APIs that allows instruction inspection, callbacks to CUDA driver APIs, and injection of arbitrary CUDA functions into any application before kernel launch. Operating at the SASS [23] level and using dynamic recompilation, NVBit analyzes GPU kernel register requirements to generate ABI compliant instrumented code without requiring tool developers to have architectural-level knowledge of the underlying GPU implementation. NVBit enables basic-block instrumentation, multi-function injection to the same location, inspection of ISA visible state, dynamic selection of instrumented or uninstrumented code, permanent modification of register state, correlation with source code, and instruction removal. NVBit supports the NVIDIA GPU architecture families of Kepler, Maxwell, Pascal, and Volta and works on pre-compiled CUDA [5], OpenCL [37], OpenACC [39] and CUDA-Fortran [26] application binaries that can make use of embedded PTX and/or GPU accelerated libraries. This work makes the following primary contributions:

- We develop and present the NVBit user-level API, which can be used to inspect/instrument instructions and to intercept CUDA driver API calls.
- We describe NVBit's underlying working principles, mechanisms and implementation details.
- We provide several examples of instrumentation tools that highlight the unique features of NVBit, including the ability to support proprietary libraries, sampling to reduce instrumentation overhead, and instruction emulation that allows architects to systematically reason about ISA extensions.

NVBit enables the development of GPU instrumentation tools that are now on par with its CPU counterparts. We believe NVBit will be immediately applicable to numerous use cases in the architecture and high performance computing communities, ranging from GPU architectural simulators (similar to CMP$im [1] and FM-SIM [20]) and error checkers such as Valgrind [22], to debuggers [40] and fault injection frameworks [9, 17].

2 BACKGROUND

Before describing NVBit's design we briefly review NVIDIA's GPU architecture and software stack design, along with terminology that relates to the capabilities provided by NVBit.

2.1 GPU Architecture

GPUs are highly multi-threaded accelerators that execute parallel sections of the target applications. To improve efficiency, GPU threads are grouped in warps. A warp is a set of 32 GPU threads sharing the same program counter which execute in a single instruction, multiple thread (SIMT) fashion. Each GPU thread owns a set of general purpose registers (up to 255) and can perform accesses to global, local and constant memory via load/store semantics. Divergence of control flow is handled via predication or a per-warp active mask that enables/disables threads from following a specific control flow path.

Many warps are then assigned to execute concurrently on a single GPU core called a streaming multiprocessor (SM) in a unit called a thread block or cooperative thread array (CTA). A GPU then consists of multiple 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 these units. NVBit allows passing numerous parameters to instrumentation functions that will affect program execution including register values, predicates, active masks, and constant bank values.

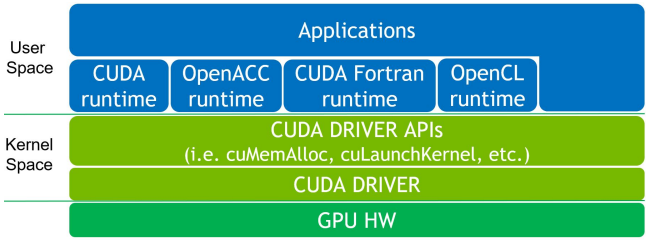

Figure 1: CUDA driver software stack.

2.2 GPU Software Stack

NVBit targets the CUDA compute stack. A user can write parallel programs using CUDA [5] and use a front-end compiler, such as NVIDIA's NVCC [24], to generate intermediate code in a virtual ISA called parallel thread execution (PTX [30]). 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 SASS machine code [23] that runs on a specific device. This backend compiler can be invoked in two ways: (1) ahead-of-time via a *ptxas* PTX assembler [30] or (2) at run-time with a just in time (JIT) compiler embedded in the GPU driver. NVBit interacts directly with the CUDA driver and handles machine code after it has already been compiled into SASS.

GPU compute programs adhere to a well-defined application binary interface or ABI, which defines the interface properties between caller and callee. Examples include what registers are caller-saved versus callee-saved, which are used to pass parameters, and how many registers can be used before resorting to passing parameters on the stack. NVBit uses a dynamic assembler to generate ABI compliant code in order to inject generic CUDA device functions into any application.

As shown in Figure 1, compute programs running on NVIDIA GPUs interface with the GPU driver using a well-defined set of CUDA APIs [25]. The CUDA driver API can be accessed by runtimes (such as those for CUDA [5], OpenCL [37], OpenACC [39] and CUDA-Fortran [26]) or directly by the user. NVBit interfaces directly with the CUDA driver, thus seamlessly interposing between software stacks that exist above it.

When an application starts, contexts (*CUcontexts*) are created by the CUDA driver to maintain the state of the used GPU devices. Modules (*CUmodules*) containing functions (*CUfunctions*) are loaded on demand before a kernel and its dependent functions are launched. NVBit provides callbacks for all CUDA driver APIs, plus specific callbacks when the application is started or terminated. The additional driver level callbacks provided by NVBit are similar to

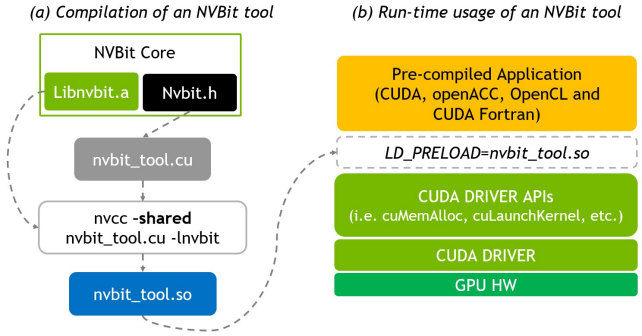

Figure 2: NVBit framework overview. An NVBit tool is first compiled as a shared library and then injected (preloaded) at run-time using *LD_PRELOAD*.

those provided by CUPTI [28], with regard to interfaces, function enumeration, and parameters.

3 NVBIT: HIGH LEVEL VIEW

Viewed from above, the NVBit framework (i.e. NVBit core) is composed of a single static library (*libnvbit.a*) and header file (*nvbit.h*) that provides all the APIs required to implement an NVBit tool. An NVBit tool is the pseudonym for any *instrumentation tool* developed with NVBit. An NVBit tool is created by: (1) developing a .cu file using the NVBit API; (2) compiling it with NVIDIA NVCC; (3) linking it with the static library *libnvbit.a*. This process generates a shared library, which in Linux is typically a file with the .so extension. Many shared libraries can be created, corresponding to many NVBit tools, but only a single library can be injected on a target application at run-time.

To use an NVBit tool (i.e. shared library) developers inject the library's functionality at run-time into applications that use the CUDA driver. The injection mechanism is based on the standard *LD_PRELOAD* [18], an optional environment variable under Linux that contains one or more paths to shared libraries, indicating that the loader should load this shared object before any other library. Figure 2 shows the flow for NVBit tool compilation and run-time *preloading*. From the user's perspective, the typical NVBit tool will implement one or more GPU device functions which are injected in an application's GPU kernels according to user defined injection points. The dynamic instrumentation of a binary is typically done when the kernel is launched for the first time, although it can be done at other times within the CUDA driver callbacks.

Listing 1 shows a simple example of an NVBit tool that counts every thread level instruction executed, then prints the counter at the end of the application. Line 11 of Listing 1 shows the implementation of the injection function which atomically increments a CUDA managed variable (`counter`) each time it is executed. The macro `NVBIT_EXPORT_DEV_FUNCTION` (defined inside *nvbit.h*) is necessary to prevent the compiler from performing dead code elimination, since the injection function appears unused to the compiler in most cases. At Line 18 a callback is triggered every time a target application executes a CUDA driver call. In this callback:

- We return immediately if we are not at the entry of a kernel launch CUDA driver call (Line 21).

```
1  /* NVBit include, any tool must have it */
2  #include "nvbit.h"
3
4  /* Counter variable used to count instructions */
5  __managed__ long counter = 0;
6
7  /* Used to keep track of kernels already instrumented */
8  std::set<CUfunction> instrumented_kernels;
9
10 /* Implementation of instrumentation function */
11 extern "C" __device__ __noinline__ void incr_counter() {
12   atomicAdd(&counter, 1);
13 } NVBIT_EXPORT_DEV_FUNC(incr_counter);
14
15 /* Callback triggered on CUDA driver call */
16 void nvbit_at_cuda_driver_call(CUcontext ctx,
17           int is_exit, cbid_t cbid, const char *name,
18           void *params, CUresult *pStatus) {
19
20   /* Return if not at the entry of a kernel launch */
21   if (cbid != API_CUDA_cuLaunchKernel || is_exit)
22     return;
23
24   /* Get parameters of the kernel launch */
25   cuLaunchKernel_params *p = (cuLaunchKernel_params *) params;
26
27   /* Return if kernel is already instrumented */
28   if(!instrumented_kernels.insert(p->func).second)
29     return;
30
31   /* Instrument all instructions in the kernel */
32   for (auto &i: nvbit_get_instrs(ctx, p->func)) {
33     nvbit_insert_call(i, "incr_counter", IPOINT_BEFORE);
34   }
35 }
36
37 /* Callback triggered on application termination */
38 void nvbit_at_term() {
39   cout << "Total thread instructions " << counter << "\n";
40 }
```

Listing 1: Simple example of an NVBit tool that counts every thread level instruction and prints the total when the application terminates.

- We cast the callback parameters into the specific parameters of the kernel launch (Line 25).
- We check if we have already encountered this kernel; if yes, we return as we have already instrumented it (Line 28).
- We iterate over all instructions composing the kernel, retrieving them with *nvbit_get_instrs* (Line 32).
- We insert a call to the instrumentation function before each instruction using *nvbit_insert_call* (Line 33).

Finally, when the application terminates we print the counter variable used by the injected device function (Line 39).

This is a pedagogical, though fully functional example. NVBit allows users to write arbitrarily complicated tools. A skilled CUDA programmer could optimize this example using a variety of techniques, including instrumenting basic blocks, reducing a radix-tree of counters (perhaps one per thread block) or using warp-level reductions to improve the overhead of the instrumented binary.

4 NVBIT: USER LEVEL APIS

This section provides an overview of the main NVBit user level APIs, which are divided into five categories: Callback, Inspection, Instrumentation, Control, and Device.

Callback API: The Callback APIs are triggered by the NVBit core when a particular event in the target application is encountered. These events are application start or termination, and entry/exit of any CUDA driver API call. Listing 2 shows the callback

```
/* Triggered when the application starts/ends */
void nvbit_at_init();
void nvbit_at_term();
/* Triggered at entry (is_exit=0) or exit (is_exit=1) of a CUDA
   driver call named "name" (i.e. cuMemAlloc). The "cbid"
   identifies the CUDA driver call (following the same
   enumeration used by CUPTI). "Params" is a pointer to the
   structure of parameters used by the driver call, which needs
   to be casted to the correct struct for the specific "cbid".
   "pStatus" points to the return status value of the CUDA
   driver call (valid only at exit) */
void nvbit_at_cuda_driver_call(CUcontext ctx,
        int is_exit, cbid_t cbid, const char* name,
        void* params, CUresult* pStatus);
```

Listing 2: Callback API: triggered at particular events.

```
/* Gets plain instructions of a CUfunction */
const std::vector<Instr*>&
    nvbit_get_instrs(CUcontext c, CUfunction f);
/* Gets basic blocks of a CUfunction */
const std::vector<std::vector<Instr*>>&
    nvbit_get_basic_blocks(CUcontext c, CUfunction f);
/* Gets related CUfunctions of a CUfunction */
std::vector<CUfunction>
    nvbit_get_related_funcs(CUcontext c, CUfunction f);
```

Listing 3: Inspection API: functions used to retrieve instructions and related CUfunctions.

APIs in detail. The user can inspect and instrument the *CUfunctions* of the current running application using the CUDA driver callbacks. For instance, when a kernel is launched, a callback to *nvbit_at_cuda_driver_call* occurs with one of the parameters being the *CUfuction* launched (i.e. kernel). NVBit's callback interface uses similar event enumerations as CUPTI [28] which should make NVBit easy to use for current GPU programmers.

Inspection API: Listing 3 shows the main Inspection API which allows users to retrieve and inspect the instructions composing a *CUfunction*. The inspection API provides two different views of *CUfunction* bodies. One view presents the body as a flat vector of the *CUfunction*'s instructions, in program order. In the absence of indirect control flow (ICF) instructions, this API can alternatively return the body as a vector of vectors of instructions, where each sub-vector represents a basic block. A basic block is an uninterrupted sequence of instructions, including predicated instructions, that each thread will execute without change in control flow. Basic blocks are generated from the static instructions of a kernel by grouping consecutive (program counters) PCs up to (a) the PC before a control flow instruction (b) the PC which is target of a control flow instruction. ICF instructions are the exception where this is not possible because they use register values to compute the final target address (which can only be discovered when the instruction is executed). In this case the basic block will also return the simpler flat view of the CUfunction, but we find the use of ICF instructions is uncommon and this condition is rarely encountered. Since a *CUfunction* can also call other *CUfunctions*, the Inspection API provides a call to retrieve them (*nvbit_get_related_funcs*). Once the function instructions have been retrieved, the instrumentation function can inspect them to understand their properties. NVBit provides a class *Instr* that abstracts the actual machine level SASS instruction (which can vary across GPU families) by disassembling and transforming the instructions using a higher level user-friendly intermediate representation. Listing 4 shows the main methods

```
class Instr {
  public:
    /* memory operation type */
    enum memOpType { NONE, LOCAL, GENERIC, GLOBAL,
                     SHARED, TEXTURE, CONSTANT};
    /* operand type */
    enum operandType {
      IMM,    // val[0] = immediate
      REG,    // val[0] = register number
      PRED,   // val[0] = predicate register number
      CBANK,  // val[0] = constant bank id
              // val[1] = constant bank offset
      SREG,   // val[0] = special register number
      MREF,   // val[0] = ra (register base)
              // val[1] = imm (immediate)
              // Addr [Rval[ra]|(Rval[ra+1]<<32)] + imm
    };
    /* operand structure */
    typedef struct {
      operandType type; /* operand type */
      bool is_neg;      /* is negative */
      bool is_abs;      /* is absolute */
      long val[2];      /* value */
    } operand_t;

    /* returns the "string" containing the SASS
       (i.e. IMAD.WIDE R8, R8, R9) */
    const char* getSass();
    /* returns the offset in bytes of this instruction
       within the function */
    uint32_t getOffset();
    /* returns the id of this instruction within the function */
    uint32_t getId();
    /* returns true if instruction uses a predicate */
    bool hasPred();
    /* returns the predicate number,
       only valid if hasPred() == true */
    int getPredNum();
    /* returns true if predicate is negated (i.e. @!P0),
       only valid if hasPred() == true */
    bool isPredNeg();
    /* returns the full opcode (i.e. IMAD.WIDE) */
    const char* getOpcode();
    /* returns type of memory operation memOpType_t */
    memOpType getMemOpType();
    /* returns true if memory operation is a load */
    bool isLoad();
    /* returns true if memory operation is a store */
    bool isStore();
    /* returns the #bytes of the memory operation */
    int getMemOpBytes();
    /* returns the number of operands */
    int getNumOperands();
    /* gets a specific operand */
    const operand_t* getOperand(int num_operand);
    /* gets line info, binary must be compiled with
       option (--generate-line-info/-lineinfo) */
    void getLineInfo(char** file, uint32_t* line);
}
```

Listing 4: Inspection API: Class Instr used to abstract machine level SASS instruction.

(but not all) of the *Instrs* class. NVBit ensures a one-to-one mapping between SASS instructions and high level *Instrs*. An *Instr* can be correlated to the application source code using the method *Instr::getLineInfo*, which provides file name and line number, provided this information has not been stripped from the application's binary.

Instrumentation API: An instrumentation tool can use the Instrumentation API (Listing 5) to inject multiple device functions before or after, any or all of a *CUfunction*'s instructions. To inject a function, we use the *nvbit_insert_call*, and we specify the location (i.e. before or after an *Instr*) and the name of the function to inject. Instrumentation functions are injected by name because their code location (i.e. program counter) is unknown until run-time. We can add arguments to the just-injected function such as register values, predicate values, and immediate values using *nvbit_add_call_arg*. Argument passing is positional and it must match the signature of

```
/* Enumeration used by nvbit_insert_call to specify where
   we want to insert the device function for a given Instr,
   if before or after */
typedef enum { IPOINT_BEFORE, IPOINT_AFTER } ipoint_t;

/* This function inserts a device function call named
   "dev_func_name", before or after an instruction "Instr".
   Device functions are identified by name (as opposed to
   function pointers) and need to be exported with the
   macro NVBIT_EXPORT_DEV_FUNC() */
void nvbit_insert_call(const Instr* instr,
      const char * dev_func_name, ipoint_t point);

/* Parameters passable to the last injected call */
typedef enum  {
  PRED_VAL,     // predicate value of the instruction
  PRED_REG,     // predicate register of the thread
  IMM32,        // immediate value 32-bit (val0=imm32)
  IMM64,        // immediate value 64-bit (val0=imm64)
  REG_VAL,      // register value         (val0=reg_num)
  CBANK_VAL     // value of constant bank
                // (val0=bank_id val1=bank_offset)
} arg_t;

/* Add parameters to the last injected call */
void nvbit_add_call_arg(arg_t arg, long val0, long val1);

/* Remove the original instruction */
void nvbit_remove_orig(const Instr* instr);
```

Listing 5: Instrumentation API: function injection and arguments passing.

```
/* Run instrumented or original code for this function
   based on flag value */
void nvbit_enable_instrumented(CUcontext ctx,
                        CUfunction func, bool flag);
/* Reset instrumentation of a function, allowing to
   reapply instrumentation */
void nvbit_reset_instrumentation(CUcontext ctx, CUfunction func);
```

Listing 6: Control API: enable/disable running of the instrumented function and reset instrumentation.

```
/* device function used to read/write register values.
   Writes are permanent into application state */
__device__ int  nvbit_read_reg32(int reg);
__device__ long nvbit_read_reg64(int reg);
__device__ void nvbit_write_reg32(int reg, int val);
__device__ void nvbit_write_reg64(int reg, long val);
```

Listing 7: Device API: read or write any register from injected functions.

the injected function. For instance, if the injected function requires two arguments, there must be two calls to *nvbit_add_call_arg* after the function is injected. It is also possible to remove or replace an original instruction using *nvbit_remove_orig*, for instance when replacing it with a functionally equivalent injected function (shown later in Section 6.3). When replacing instructions, NVBit cannot guarantee the application will continue to work as expected, unless the injected functionality is identical to the replaced instructions.

Control API: NVBit provides the ability to control instrumentation at application run-time, for example, to dynamically select the execution of an instrumented or non-instrumented version of a function. At run-time the NVBit framework will select one version or the other, based on the last value set by the user with the call *nvbit_enable_instrumented*, shown in Listing 6. The user can reset applied instrumentation at any time, so a new instrumentation selection can be applied. This operation may be performed at any point inside the callback *nvbit_at_cuda_driver_call*, even while the application is executing. This allows instrumented functions to be rewritten on the fly depending on the dynamic conditions of the application's execution, a significant advancement over static compiler-based GPU instrumentation.

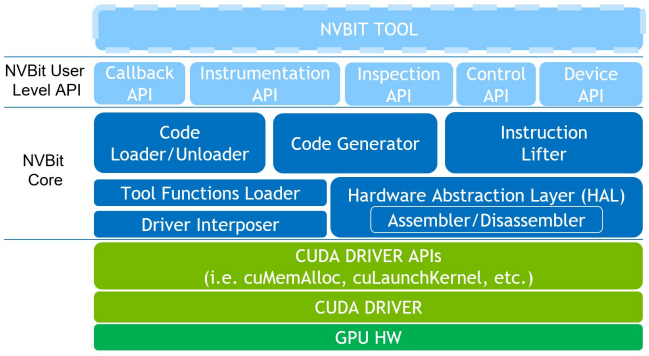

Figure 3: NVBit core components and user-level API.

Device API: Finally, NVBit provides a device level API that can be used within the instrumentation (i.e. injected) functions. Most importantly, it is possible to read and write any register used by the application kernels or device functions using this API. While arbitrary writes of register values may cause catastrophic application level errors, the ability to modify GPU state is a requirement for use cases such as fault injection or instruction emulation. Listing 7 describes NVBit's Device API.

5 NVBIT: IMPLEMENTATION

The NVBit core is responsible for interacting with the CUDA driver and providing the user facing functionality described in Section 4. This section describes the internal implementation details of the NVBit core and discusses the overheads related to the JIT-compilation.

5.1 Software Components Details

Figure 3 shows the high level components of the NVBit core that we now describe in more detail.

Driver Interposer: A *Driver Interposer*, located at the bottom of the NVBit core layer in Figure 3, intercepts the CUDA driver APIs using the function overloading mechanisms provided by *LD_PRELOAD* [18]. When the CUDA driver loads an application function (*CUfunction*), the Driver Interposer records its properties. This includes, maximum register usage, maximum stack usage, dependent functions (i.e. functions that can be called by the current function) and the memory location where the instructions have been loaded. These required properties are consumed by other components within the NVBit core library. For instance, the maximum register consumption is used when computing the correct amount of registers to save before jumping into an instrumentation function. The Driver Interposer is also responsible for propagating the CUDA driver callback API to the NVBit user level callback API.

Tool Functions Loader: The *Tool Functions Loader* is responsible for loading all the device functions within the dynamic library of the NVBit tool itself. This process does not happen automatically when the application starts because the CUDA driver is unaware of device and global functions contained in the NVBit tool library.

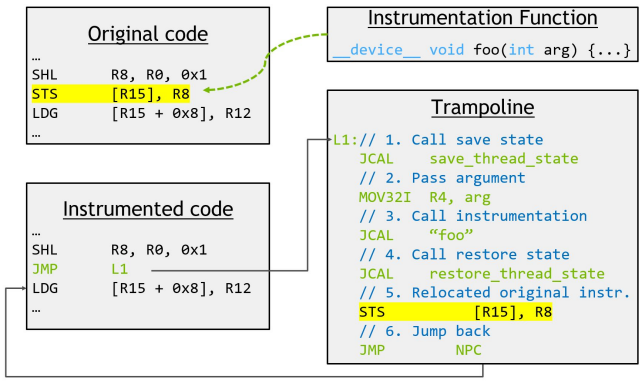

Figure 4: Instrumented code generation process.

Some of the loaded device functions (those exported with the macro `NVBIT_EXPORT_DEV_FUNCTION`) are recorded in a map associating the function name with a structure containing the function attributes such as the number of registers used, the requested stack size, and the location where the code is loaded in GPU memory. This information is used by the *Code Generator* when creating code necessary to jump to the instrumentation functions. The *Tool Functions Loader* is also responsible for loading other pre-built device functions (embedded in *libnvbit.a*) such as those used to save and restore registers before jumping into the user injected functions. For efficiency NVBit implements a fixed set of save and restore functions, each targeting a specific number of general purpose registers.

Hardware Abstraction Layer (HAL): The *Hardware Abstraction Layer (HAL)* is initialized when a *CUcontext* is started on a specific device. During HAL's initialization, device specific information is recorded, such as the size of each instruction in bytes, alignment requirements, number of registers available per thread, and ABI version. Within a GPU family, the instruction size is unique and fixed. Kepler, Maxwell, and Pascal have 64-bit-wide encodings, while Volta has 128-bit-wide encodings. The ABI version specifies which registers and special registers (such as those holding the convergence barrier state in Volta) must be saved and restored before entering and exiting an instrumentation function. The HAL also initializes device specific assembly/disassembly functions. These functions are used to assemble code in the *Code Generator* or to disassemble code in the *Instruction Lifter*. While this component is not strictly required, using a HAL improves the portability of NVBit across GPU generations, where the SASS ISA is not guaranteed to remain constant.

Instruction Lifter: The *Instruction Lifter* is responsible for retrieving the "raw" buffer of SASS instructions for each application level *CUfunction*. When the user requests to inspect the instructions of a *CUfunction* (using *nvbit_get_instrs* or *nvbit_get_basic_blocks*) the *Instruction Lifter* converts each one to an object of class *Instr* as shown in Listing 4. Explained previously, the *Instr* class is machine independent and represents a single SASS instructions. Disassembled instructions (i.e. *Instr*) can be arranged in a vector or subdivided in vectors (representing basic blocks) depending on the user's API usage.

Code Generator: At the exit of the CUDA driver callback, if instrumentation was applied, the *Code Generator* begins functioning. Figure 4 provides an example of how NVBit's instrumentation code generation occurs. The top left of Figure 4 is an example of *original code* for a *CUfunction* inspected by the user. The top right is an *instrumentation* function (named `foo`) that the user wants to inject before the highlighted instruction (green arrow). The *Code Generator* executes the following steps:

- Makes a copy of the *original code* in system memory (we refer to this copy as *instrumented code*).
- Generates a new region of code allocated in GPU memory, named *trampoline*.
- Modifies the *instrumented code*, substituting the highlighted instruction (`STS [R15], R8`) with a jump to the *trampoline* (`JMP L1`). Inserting trampolines elegantly preserves instruction layout, while in-place expansion would be significantly more complicated possibly requiring additional runtime data structures.

The generated *trampoline* typically contains the following instructions:

1. A call to a routine that saves the state of the thread before executing the instrumentation function. NVBit saves only the minimum amount of general purpose registers, and the appropriate save routine is selected by analyzing the register requirements of both the original code and injected function. General purpose registers, conditional codes, and predicates are saved by this routine on the stack.

2. A sequence of instructions (only one MOV in this example) to pass the arguments specified by the user. The argument passing convention (i.e. which registers to use and when to use the stack) is defined by the specific ABI of the target device, which is initialized and handled by the HAL.

3. A jump to the actual program counter of the instrumentation function `foo` which is retrieved by accessing the injection function map populated by the *Tool Functions Loader*.

4. A call to a routine that restores the state of the thread from the stack (i.e. inverse function of the save routine).

5. Execution of the "relocated" original instruction (`STS [R15], R8`). Critically, if this relocated instruction is a relative control flow instruction, the offset must be adjusted to account for the new position and the original target location.

6. Finally, a return to the *instrumented code* at the next program counter after the relocated instruction (`JMP NPC`).

There is one trampoline per instrumented instruction, however for efficiency purposes the allocation of space for these trampolines is handled in bulk using a custom memory allocator. The content of these trampolines can vary depending on how many injection functions are inserted before or after the same GPU instructions and if the injection happens before, after, or in both locations. If the function *nvbit_remove_orig* is used (Listing 5), the "relocated" original instruction must also be converted into a `NOP`.

Code Loader/Unloader: At run-time, the user can decide to enable or disable instrumentation for a particular *CUfunction*. The *Code Loader/Unloader* swaps *original code* with *instrumented code* on demand, based on the values passed to the control API *nvbit_enable_instrumented*. The cost of this operation is identical to that of a *cudaMemcpy* from host to device with the number of

bytes equal to the size of the *original code*. To allow swapping, both *original code* and *instrumented code* must have the exact number of bytes and occupy the exact same location in GPU memory. Only in this way can NVBit guarantee that absolute jumps in the program targeting the *CUfunction* will continue to work regardless of which version (instrumented or not) is running. The trampolines, since they are only created in device memory, are always GPU resident and do not need to be removed unless the control API *nvbit_reset_instrumented* is used or the *CUmodule* for this particular *CUfunction* is unloaded. The *Code Loader/Unloader* also computes the stack and register requirements for the kernel launch, based on which version of the code will be executing.

5.2 JIT-Compilation Overhead

Binary instrumentation intrinsically introduces overheads, resulting in execution slowdown. NVBit overheads stem from two aspects (a) the *cost of generating the instrumentation code* (i.e. JIT-compilation overhead), and (b) the *cost of running the instrumentation code*. JIT-compilation overhead is one drawback to binary instrumentation versus compiler-based instrumentation where code generation happens offline. However, as we later demonstrate NVBit allows us to perform tasks that cannot be achieved with compiler based instrumentation. This section focuses on quantifying the JIT-compilation overhead, but later in Section 6.2 we discuss the overhead of running a specific NVBit tool and exemplify the use of *sampling* as one technique for reducing it. The overhead of running the instrumentation code depends on what the user is trying to accomplish with it (i.e. the body of the instrumentation function) and for this reason are extremely difficult to broadly quantify.

Within the NVBit's core, the JIT-compilation overhead can be divided into six components: (1) retrieving the original GPU code, (2) disassembling the GPU program, (3) converting the binary into the format presented to the developer via the NVBit API, (4) executing the C/C++ user code to inject instrumentation functions and arguments, (5) running the *Code Generator* to produce the final instrumented code and (6) swapping the original code with the instrumented code. While the components (1), (2), (3) and (6) depend on the characteristics of the application, the components (4) and (5) depend on how much of the application is being instrumented. For this evaluation we assume each kernel is instrumented once, with each instruction also instrumented once, which approximates a rough upper bound on the JIT-compilation overhead an NVBit's user might observe. This upper bound is not theoretical, but an extrapolation from our most demanding use cases.

Figure 5 shows the breakdown of the JIT-compilation overhead in the six components previously listed when applying the NVBit tool described in Listing 1. The selection of this NVBit tool is arbitrary, since JIT-compilation overhead depends on the amount of newly generated code (due to trampolines, jumps and arguments passing instructions) rather than the body of the instrumentation function (which is compiled offline). Using another instrumentation function, would result in a similar JIT-compilation overhead.

We benchmark using the OpenACC SpecAccel suite [34] selecting medium problem sizes, where each benchmark can complete in under one minute on a NVIDIA TITAN V GPU [31]. Selecting relatively short executing benchmarks ensures that the JIT-compilation

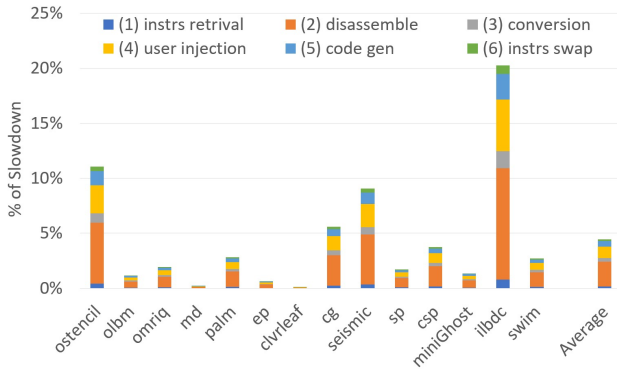

Figure 5: JIT-compilation overhead with respect to a native execution for the OpenACC SpecAccel benchmarks.

overhead is not dwarfed by long execution time of the application, though this will obviously vary per application. Also, while the SpecAccel benchmarks have a CUDA native implementation we opted to use the OpenACC implementation to highlight that NVBit is agnostic to the high-level language used since it instruments binaries.

We see that, on average, the JIT-compilation overhead is below 5%, but in some cases can reach 20%. Because in this evaluation we instrumented each kernel once, the JIT-compilation overhead is higher when the application is composed of many unique kernels (which is the case for the `ilbdc` application). This is further exacerbated when these kernels execute only once and are short, for instance when launched with a small number of thread blocks. For all of these applications, the biggest contributor to the JIT-compilation overhead is the disassembly phase, which converts the GPU binary code into the intermediate representation used internally by NVBit.

6 NVBIT TOOLS: USE CASE EXAMPLES

This section presents three use-cases of NVBit, emphasizing its unique capabilities such as support of pre-compiled libraries, sampling and instruction emulation.

6.1 Memory Access Address Divergence

Understanding memory access patterns is very important when optimizing applications or designing memory subsystems. NVBit allows one to easily extract this information by instrumenting every memory operation to collect reference addresses, which then can be analyzed directly on the GPU or sent to the CPU for further processing. Entire cache simulators can be built around these mechanisms. Listing 8 shows an NVBit tool that computes the number of unique cache lines requested for each warp-level global memory instruction. An application that requires many cache lines per warp-level memory instruction will perform less efficiently than one which only requires one line.

In Line 7, the instrumentation function `ifunc` takes four arguments: a predicate value, two register values, and one immediate. The values for these arguments will be set at run-time on each instrumented memory instruction. A false predicate value means that the instrumented instruction is not actually executing so the

```
1  /* counters for unique cache lines accessed and for
2     the number of memory instructions executed */
3  __managed__ float uniq_lines = 0;
4  __managed__ long mem_instrs = 0;
5
6  extern "C" __device__ __noinline__
7    void ifunc(int pred, int r1, int r2, int imm) {
8    /* Return if predicate is false */
9    if(!pred) return;
10
11   /* Construct address */
12   long addr = (((long)r1) | ((long)r2 << 32)) + imm;
13
14   /* Compute active mask of the warp */
15   int mask = __ballot(1);
16
17   /* Only the first active thread in the warp
18      increments the memory instruction counter */
19   if (get_lane_id() == __ffs(mask) - 1)
20     atomicAdd(&mem_instrs, 1);
21
22   /* Count how many threads in the warp access
23      the same cache line */
24   long cache_addr = addr >> LOG2_CACHE_LINE_SIZE;
25   int cnt = __popc(__match_any_sync(mask, cache_addr));
26
27   /* Each thread contributes proportionally to the
28      cache line counter */
29   atomicAdd(&uniq_lines, 1.0f / cnt);
30 } NVBIT_EXPORT_DEV_FUNC(ifunc);
31
32 void nvbit_at_cuda_driver_call(CUcontext ctx,
33         int is_exit, cbid_t cbid, const char *name,
34         void *params, CUresult *pStatus) {
35
36   /* OMITTING CODE: same as Listing 1, Lines 20-30 */
37
38   /* Iterate over kernel's instructions */
39   for (auto &i: nvbit_get_instrs(ctx, p->func)) {
40     /* If instr isn't global memory operation skip */
41     if(i->getMemOpType() != Instr::GLOBAL) continue;
42     /* Iterate on operands of instruction */
43     for(int n = 0; n < i->getNumOperands(); n++) {
44       operand_t * op = i->getOperand(n);
45       /* If operand isn't a memory reference skip */
46       if(op->type != Instr::MREF) continue;
47       /* Inject instrumentation and its arguments */
48       nvbit_insert_call(i, "ifunc", IPOINT_BEFORE);
49       nvbit_add_call_arg(PRED_VAL);
50       nvbit_add_call_arg(REG_VAL, op->val[0]);
51       nvbit_add_call_arg(REG_VAL, op->val[0] + 1);
52       nvbit_add_call_arg(IMM32, op->val[1]);
53     }}}
54
55 void nvbit_at_term() {
56   printf("Average cache lines requests per memory "
57          "instruction %f\n", uniq_lines/mem_instrs);
58 }
```

Listing 8: Memory access address divergence. It computes the average number of cache lines requested per warp-level memory instruction.

`ifunc` returns immediately (Line 9). In Line 12, the memory reference address is constructed by combining the two register values and the immediate. A mask of all the active threads in the warp is computed (Line 15), and is then used to select a single leader thread that atomically increments the global memory reference counter (line 20). Each thread computes the cache line address it accesses (Line 24) and it "reduces" it to a single value using special warp-level CUDA intrinsics (Line 25). This value indicates how many other threads are accessing the same cache line within the warp. In Line 29, all threads then atomically increment the cache line counter proportionally to the reduced value (i.e. if two threads access the same cache line, each one of them will increment the cache line counter by 1/2). At inspection time (Line 41) we instrument only the instructions for which the memory operation

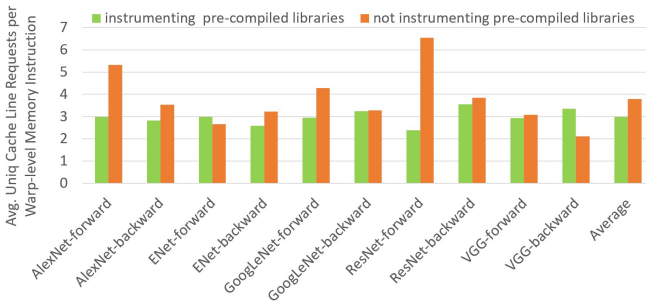

Figure 6: Address divergence analysis on ML workloads, with and without instrumenting pre-compiled libraries.

type is global (`Instr::GLOBAL`). For those instructions and for each memory reference operand (`Instr::MREF`) we inject the function `ifunc` (Line 48) followed by the passing of the four arguments (Lines 49 to 52). Finally, at program termination we print the ratio between unique cache lines requested and total warp-level memory instructions executed (Line 57). Listing 8 could have been implemented more efficiently by injecting, at most, one trampoline for each instruction and passing multiple references as arguments to the function. However, for ease of implementation and discussion, we opted for the simpler approach.

We run the instrumentation tool in Listing 8 on a variety of Machine Learning (ML) workloads implemented in Torch7 [4] including AlexNet, ENet, GoogLeNet, ResNet and VGG on the ImageNet dataset[3]. These ML workloads use pre-compiled libraries developed by NVIDIA such as cuBLAS and cuDNN [29]. A compiler-based approach would not be able to capture the memory references emitted within those libraries, resulting in an incomplete analysis.

Figure 6 shows the results of our analysis in which we have first enabled and then disabled instrumentation of all the pre-compiled libraries. In our tool, by disabling instrumentation of the pre-compiled libraries we are reproducing the behavior of a possible compiler-based approach which does not have access to the libraries' source code. Excluding those libraries distorts reality and considerably overestimates the memory divergence of the applications.

To understand the extent of this problem we have also applied an optimized version of the instruction count tool in Listing 1 and measured the percentage of instructions executed by these workloads inside pre-compiled libraries. This percentage ranges from 74% to 96%, and averages 88% of the total executed instructions across the various ML workloads. These accelerated libraries contain several hundreds of distinct kernels. For instance, cuBLAS has dozens of similar kernels with different precision levels and transpositions. Even having access to the libraries' source code it would require a significant amount of time to compile them. NVBit instead does not require source code and it works on any application binary that makes use of these libraries, applying instrumentation at run-time only on the kernels that are actually invoked.

6.2 Kernel Sampling: Instruction Histogram

While Section 5.2 showed the JIT-compilation overhead, additional overheads incur due to the execution of instrumented code. These overheads are not due to the NVBit framework itself, but instead

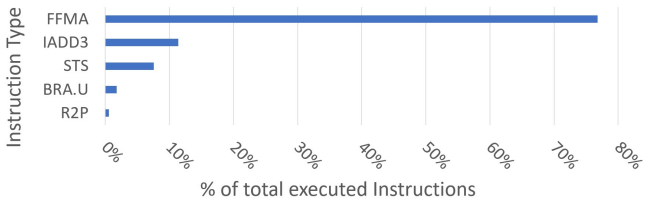

Figure 7: Top-5 instruction type executed on the SpecAccel:clvrleaf application, extracted with an NVBit tool.

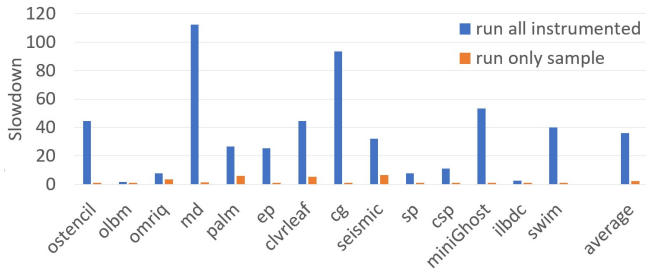

Figure 8: Slowdown of full instrumentation and sampling approach with respect to native execution.

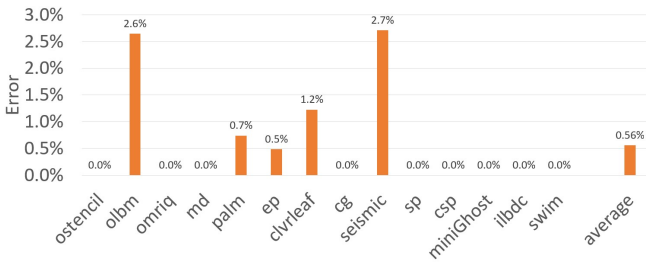

Figure 9: Error for kernel sampling with respect to full instrumentation, reported as a single number for each benchmark averaged across instruction categories.

are determined solely by how often the instrumented code is executed and by how complex each instrumented call is. Naturally, instrumentation that is more invasive or computationally intensive at each instrumentation site will slow down the application more. One way to reduce the execution overhead, is to write highly optimized instrumentation functions by following the same optimization rules that apply to any CUDA program. NVBit can also mitigate this overhead by using standard techniques such as *sampling* to reduce the frequency of instrumentation callbacks. This section shows how NVBit can implement sampling by allowing a user to select (at run-time) whether a kernel runs its instrumented or uninstrumented version.

In this example we use sampling as follows. We instrument all the kernels, but we launch the instrumented version only once for each set of unique grid dimension values. For instance, if a kernel *foo* is launched 100 times with grid dimensions {128,128,1} and 50 times with grid dimensions {64,64,64} we run the instrumented version of *foo* only twice, once for each unique grid dimension. For the remaining 148 times we run *foo* uninstrumented. We perform this selection by using the NVBit API *nvbit_enable_instrumented* within our instrumentation tool before a kernel is launched. We then use the information collected during the instrumented kernel execution to approximate the information not collected during the uninstrumented execution.

To highlight this approach, we used an instrumentation tool that performs an analysis of all the instructions executed to construct a histogram of the Top-5 instructions executed like the one in Figure 7. We used the OpenACC SpeccAccel benchmarks [34] selecting large problem sizes, requiring several minutes of execution time for each benchmark on a NVIDIA TITAN V GPU [31]. Running full instrumentation for the entire application could result in very long execution time, yielding up to 112× slowdown and approaching 24 hours for the longest benchmark in this test.

Figure 8 shows the slowdown of both the full instrumentation approach and the sampling approach with respect to native execution. On average the full instrumentation approach is 36.4× slower than native execution, while the sampling approach incurs in only 2.3× slowdown. Although targeting the same applications, Figure 8 and Figure 5 cannot be directly compared because Figure 5 represents only the JIT-compilation overhead (cost of generating the instrumented code). As explained in Section 5.2, JIT-compilation overhead is independent of the content of the injected functions, and depend primarily on how many jumps and trampolines are generated.

The speedup of the sampling approach comes with a possible decrease of accuracy. Figure 9 shows the error for the kernel sampling approach, reported as a single number for each benchmark averaged across instruction categories. The error is computed comparing with the results collected without sampling which are by definition exact. Using this sampling technique results in an average error less than 0.6%. For this particular combination of sampling approach and instrumentation tool the sampling error depends on the control flow characteristics of the kernels executed. If the control flow of a kernel does not change based on the values computed, but it is only a function of the grid dimensions, the sampling error is 0%. Many applications have this property because they operate on grids or meshes in which the computed values do not alter the control flow properties of the algorithms.

Depending on the target applications and instrumentation tool, more advanced sampling techniques such as adaptive statistical profiling [10, 41] can be implemented using the underlying mechanisms presented in this section.

6.3 Instruction Emulation: Warp-wide FFT

While NVBit's instrumentation is semantic-preserving by default, we now demonstrate how to use NVBit's Device API (Listing 7) to modify ISA-visible state. Prior art has used similar functionality to study fault injection [9] and automatic type conversion [16]. This section discusses instruction emulation, typically employed for architectural exploration and pre-silicon compiler testing.

This section demonstrates a hypothetical warp-wide (32-point) FFT instruction named *WFFT32*. Listing 9 shows an NVBit tool that

```
1  /* Compute a 32-point warp-wide FFT (across lanes) */
2  extern "C" __device__ __noinline__
3    void wfft32_emu(int reg_dst_num, int reg_src_num) {
4    /* Read input register */
5    long in = nvbit_read_reg64(reg_src_num),
6      /* Implementation of the warp-wide FFT function */
7    shuffle_fft_warp(in, out);
8    /* Write value in destination registers */
9    nvbit_write_reg64(reg_dst_num, out);
10 } NVBIT_EXPORT_DEV_FUNC(wfft32_emu);
11
12 void nvbit_at_cuda_driver_call(CUcontext ctx,
13         int is_exit, cbid_t cbid, const char *name,
14         void *params, CUresult *pStatus) {
15
16   /* OMITTING CODE: same as Listing 1, Lines 20-30 */
17
18   /* Iterate over kernel's instructions */
19   for (auto &i: nvbit_get_instrs(ctx, p->func)) {
20     operand_t * ops = i->get_operands();
21
22     /* Identify "proxy" instruction
23       asm("or.b32 %0, %1, 0xfefefefe;") */
24     if (i->getOpcode() == "LOP32I.OR" &&
25         ops[2].val[0] == "0xfefefefe") {
26       nvbit_insert_call(i,"wfft32_emu",IPOINT_BEFORE);
27       nvbit_add_call_arg(IMM32, ops[0]->val[0]);
28       nvbit_add_call_arg(IMM32, ops[1]->val[0]);
29       /* remove the "proxy" instruction */
30       nvbit_remove_orig(i);
31     }}}
```

Listing 9: NVBit tool used to emulate a warp wide 32-point FFT instruction (WFFT32) with a functionally equivalent function named wfft32_emu().

replaces the hypothetical *WFFT32* instruction with the device function *wfft32_emu* which emulates its functionality. Since the current NVCC compiler cannot emit a real encoding for the nonexistent *WFFT32* instruction, we use a proxy instruction instead. We select "`LOP32I.OR R1, R2, 0xfefefefe`" as the proxy instruction. The `LOP32I.OR` is the logical OR instruction that can take one input register and one output register (matching the signature of *WFFT32*) along with an immediate operand. We use the immediate operand as a "magic" number to differentiate the proxy instruction from all the other instances of `LOP32I.OR` instructions. Because we are hijacking an existing instruction, it is possible that some applications contain instances of our proxy instruction; we can mitigate that possibility in several ways, including writing an NVBit tool that searches for native instances of our proxy instruction in applications, or choosing graphics-related proxy instructions that are not likely to be used by CUDA applications. The proxy instruction can be generated using an inline assembly PTX instruction as shown in Listing 10.

Listing 9, line 3, shows the implementation of the *wfft32_emu* function. For clarity we omit the implementation of the FFT[1] and focus on the Device APIs `nvbit_read_reg64` and `nvbit_write_reg64` to read and modify registers. On line 24, we use NVBit to identify the proxy instruction used to represent the *WFFT32* instruction. On line 26, we inject a call to the emulated function and pass as arguments the source and destination register numbers used by the the proxy instruction (Line 27 and Line 28). Finally, on line 30 we remove the proxy instruction as its only purpose was to allow the injection of the emulation function.

[1]See [6, 38] for implementations of warp-wide FFTs using warp shuffle operations.

```
__global__ void fft32_kernel(float2 *in, float2 *out) {
  /* Get the thread identifier */
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  /* Inserts the proxy instruction representing the WFFT32.
     We use or.b64 which translates to "LOP32I.OR R1, R2,
     0xfefefefe". NVBit will replace it with fft32_emu */
  asm("or.b64 %0, %1, 0xfefefefe;":"=l"(in[tid]):"l"(out[tid]));
}
```

Listing 10: Example of FFT kernel using the "proxy" PTX instruction representing the hypothetical WFFT32.

When instrumenting the kernel in Listing 10 with the NVBit tool in Listing 9, the inline assembly PTX instruction will be replaced with the *wfft32_emu* function. We can use NVBit to combine instruction emulation and instruction tracing to trace instruction sets that do not exist, potentially enabling future trace-based GPU simulators. For illustrative purposes, we combined the FFT instruction emulation tool with the instruction count tool in Listing 1. Executing the kernel in Listing 10, which computes one 32-point FFT per warp using *WFFT32*, we find that each warp executes 21 instructions. If however, we replace the *WFFT32* instruction with CUDA code that performs the warp-wide FFT, we find instead that each warp executes 150 instructions. This simple example shows how NVBit can help us gauge the impact of ISA changes.

7 LIMITATIONS AND DISCUSSION

While NVBit is designed for generality, allowing arbitrary injection of any CUDA device function, it does have some limitations.

Shared and constant memory usage: Injected functions may not use *shared* and *constant* memory because that memory can be used by the application itself. Using it in an instrumentation tool could cause the instrumented programs to fail. In practice, programs commonly use all of the shared memory capacity, leaving nothing for the instrumentation library itself regardless.

Use of libraries in instrumentation functions: Instrumentation functions cannot use accelerated libraries used by the targeted application. Doing so could create "recursion" of instrumentation in which an instrumented function becomes itself instrumented.

Non-deterministic applications: While NVBit is architected to be minimally invasive, additional instructions, register pressure, and cache effects from user level instrumentation and the NVBit framework itself can alter the behavior of applications that contain race-conditions or rely on specific scheduling or timing assumptions for correct behavior. In other words, if an application is already susceptible to non-deterministic behavior, instrumenting with NVBit will likely exacerbate this non-determinism. For instance, collecting memory addresses (with the NVBit tool in Section 6.1) on an application that uses in-memory synchronization via spin-loops, can result in a very different number of memory instructions being executed if the application is run multiple times. While worth noting, this limitation is common to all the instrumentation approaches (static or dynamic) and not specific to NVBit.

More on execution overheads: In NVBit every active thread enters, then leaves, the instrumentation function, thus the overhead must be paid by all of them. Within the instrumentation function, however, it is possible to implement thread specialization based on thread identifier, for instance having a subset of threads return immediately. We are planning to explore some form of predicate

matching before jumping to the instrumentation function allowing a finer gain selection of the threads. Also, as explained, before jumping to an instrumentation function, specific registers are saved in memory for every thread. Although fully parallelized, it takes many cycles and can destroy cache locality. We examined carving out instrumentation registers, to limit the state that needs to be saved, however this poses new challenges, including decreased occupancy and requiring a new ad-hoc (unsupported) ABI.

Information accessible with NVBit: The information that can be obtained via the NVBit *Inspection APIs* is comparable to what can be observed with NVIDIA's tools such as *nvdisasm*[23] and *cuda-gdb*[27]. For instance, developers can use *nvdisasm* to observe the SASS code of any GPU binary (if the SASS is present), and use *cuda-gdb* to observe the translation and mapping of any embedded PTX code to SASS. Furthermore, one can use *cuda-gdb* to read values from memory and registers, allowing manual inspection of the entire ISA visible machine state (as with NVBit). The primary advantage of NVBit over these specific tools is that this information can be analyzed at run-time with high performance C/C++ code, many orders of magnitude faster that what the user can do interactively with *nvdisasm* and *cuda-gdb*. Additionally, the dynamic instrumentation aspect of NVBit, where the user can inject generic CUDA functions into a running application, is not possible with any other tool today.

On the fly dynamic instrumentation: As explained in Section 4, NVBit allows instrumentation before a kernel is launched. However, once the kernel is executing, the code cannot be further changed (until the next launch). This is in contrast with the existing approaches on CPU that have the ability to breakpoint at any time and modify the code on the fly. While worth mentioning, this is not a limitation of NVBit but rather of the GPUs that cannot self-compile their code but must rely on the CPU to drive execution.

8 RELATED WORK

This paper is the first to introduce dynamic binary instrumentation on NVIDIA GPUs. On CPUs, this capability is available in many frameworks such as ATOM [35], HP Caliper [13], DynamoRIO [2] and Intel Pin [12, 19]. ATOM [35] was introduced more than 25 years ago targeting the Alpha processor [21] and it is perhaps the first dynamic binary instrumentation framework proposed. It was followed by HP Caliper [13] whose initial implementation targeted the IA-64 Itanium processor and later by DynamoRIO [2] first released on Intel x86 and later on ARM and IBM Power architectures. Finally, Intel PIN [19], targeting the entire Intel x86 family, is arguably the most widely used dynamic binary instrumentation framework. With Intel Pin the developer writes instrumentation tools, or "Pintools" that specify where and how the program will be instrumented. Pin and Pintools work together to instrument the target program. While Pin performs the JIT-compilation and instrumentation, the Pintool directs how/where to instrument. Inside a Pintool, the developer can inspect basic blocks or plain instructions of any application's function and decide to inject arbitrary functions before or after each one of them. In NVBit the separation of concerns between "NVBit core" and "NVBit tools" is inspired by Intel Pin (i.e. Pin and Pintools) but NVBit goes beyond Pin in functionality because it must also handle runtime API interposition, the complications of translating and mapping C/C++, PTXAS and SASS, as well as unique ISA features of modern GPUs through a hardware abstraction layer.

To the best of our knowledge, all prior instrumentation approaches for GPUs have focused on compile-time instrumentation. Ocelot [7] 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 [30], but has also been used to perform instrumentation of GPU programs [8] and to implement GPU simulators [15]. While Ocelot is a useful tool, it is limited in functionality because it operates at the PTX level which does not exactly correspond to the binary code executed by the GPU. Consequently, Ocelot interferes with the back-end compiler optimizations and is more invasive and less precise in its ability to instrument a program than NVBit. SASSI [32] solved this problem by enabling compile-time instrumentation directly on the low level SASS, leveraging the NVIDIA production back-end compiler, but still has the same compile-time restrictions that can limit its usefulness. Hayes et al. [11] demonstrated a similar approach to our decoding/encoding strategy used by Hardware Abstraction Layer (HAL) to support different NVIDIA GPU families.

9 CONCLUSION

This work introduces NVBit, a new dynamic binary instrumentation framework targeting NVIDIA GPUs. NVBit improves upon prior compiler-based instrumentation tools by enabling fast and efficient instrumentation of GPU code regardless of application source code availability. By working directly at the SASS level, NVBit can instrument applications that have been produced by users with NVCC, via JIT compilation of PTX, or through the inclusion of precompiled shared libraries such as cuDNN, cuBLAS, and cuSolver. As the number of GPU accelerated libraries proliferates the likelihood of applications relying on these highly optimized libraries increases, evidenced by most machine learning frameworks providing direct support for GPU acceleration via cuDNN. NVBit overcomes prior portability limitations by operating directly on SASS, but utilizes a hardware abstraction layer that enables it to seamlessly work across recent generations of NVIDIA GPUs without being tied to specific compilers. This same HAL enables NVBit users to obtain information from the GPU hardware without having to reverse engineer SASS assembly that is not guaranteed to remain constant from GPU generation to generation. The development of NVBit will enable a new class of performance analysis, optimization, and architectural exploration for GPUs that has not been possible using existing tools which only contain subsets of NVBit's broad functionality.

ACKNOWLEDGMENTS

This research was, in part, funded by the U.S. Government under the DoE PathForward program. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Jaleel Aamer, S. Cohn Robert, Luk Chi-Keung, and Jacob Bruce. 2008. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In *Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*.

[2] Derek Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[3] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. 2016. An Analysis of Deep Neural Network Models for Practical Applications. *CoRR* abs/1605.07678 (2016). arXiv:1605.07678

[4] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[5] Shane Cook. 2012. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, Waltham, MA.

[6] Carlo del Mundo and Wu-chun Feng. 2014. Towards a Performance-portable FFT Library for Heterogeneous Computing. In *Proceedings of the 11th ACM Conference on Computing Frontiers*.

[7] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. 2009. *Translating GPU Binaries to Tiered SIMD Architectures with Ocelot*. Technical Report 09-01. Georgia Institute of Technology. http://www.cercs.gatech.edu/tech-reports/tr2009/abstracts/01.html

[8] Naila Farooqui, Andrew Kerr, Gregory Diamos, Sudhakar Yalamanchili, and Karsten Schwan. 2011. A Framework for Dynamically Instrumenting GPU Compute Applications Within GPU Ocelot. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*.

[9] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. 2017. SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 249–258.

[10] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 156–164.

[11] Ari B. Hayes, Fei Hua, Jin Huang, Yan-Hao Chen, and Eddy Z. Zhang. 2019. Decoding CUDA Binary. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 229–241.

[12] Kim Hazelwood and Artur Klauser. 2006. A Dynamic Binary Instrumentation Engine for the ARM Architecture. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 261–270.

[13] Robert Hundt. 2000. HP Caliper: a Framework for Performance Analysis Tools. *IEEE Concurrency* 8, 4 (October 2000), 64–71.

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the International Conference on Multimedia*. 675–678.

[15] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU Workloads and Systems. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*.

[16] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. 2013. Automatically Adapting Programs for Mixed-precision Floating-point Computation. In *Proceedings of the International Conference on Supercomputing (ICS)*. 369–378.

[17] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. 2012. Classifying Soft Error Vulnerabilities in Extreme-scale Scientific Applications Using a Binary Instrumentation Tool. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*.

[18] Linux Programmer's Manual. http://man7.org/linux/man-pages/man8/ld.so.8.html. Accessed: 2019-02-11.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.

[20] Rano Mal and Yul Chu. 2017. A Flexible Multi-core Functional Cache Simulator (FM-SIM). In *Proceedings of the Summer Simulation Multi-Conference*.

[21] Edward McLellan. 1993. The Alpha AXP Architecture and 21064 Processor. *IEEE Micro* 13, 3 (1993), 36–47.

[22] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 89–100.

[23] NVIDIA CUDA Binary Utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html. Accessed: 2019-02-11.

[24] NVIDIA CUDA Compiler Driver NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Accessed: 2019-02-11.

[25] NVIDIA CUDA Driver APIs. https://docs.nvidia.com/cuda/cuda-driver-api/index.html. Accessed: 2019-02-11.

[26] NVIDIA CUDA Fortran. https://developer.nvidia.com/cuda-fortran. Accessed: 2019-02-11.

[27] NVIDIA CUDA GDB. https://docs.nvidia.com/cuda/cuda-gdb/index.html. Accessed: 2019-02-11.

[28] NVIDIA CUPTI Callback APIs. https://docs.nvidia.com/cuda/cupti/group__CUPTI__CALLBACK__API.html. Accessed: 2019-02-11.

[29] NVIDIA GPU Accelerated Libraries for Computing. https://developer.nvidia.com/gpu-accelerated-libraries. Accessed: 2019-02-11.

[30] NVIDIA Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2019-02-11.

[31] NVIDIA TITAN V. https://www.nvidia.com/en-us/titan/titan-v/. Accessed: 2019-02-11.

[32] SASSI Instrumentation Tool for NVIDIA GPUs. https://github.com/NVlabs/SASSI. Accessed: 2019-02-11.

[33] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the Workshop on Computer Architecture Education*.

[34] Standard Performance Evaluation Corporation (SPEC): ACCEL. https://www.spec.org/accel/. Accessed: 2019-02-11.

[35] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 196–205.

[36] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 185–197.

[37] John. E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering* 12, 3 (May-June 2010), 66–73.

[38] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast Convolutional Nets with fbfft: A GPU Performance Evaluation. *CoRR* abs/1412.7580 (2014).

[39] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the International Conference on Parallel Processing*. 859–870.

[40] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. 2008. How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation. In *International Conference on Compiler Construction*. 147–162.

[41] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, Efficient, and Adaptive Calling Context Profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 263–271.