# NVBit: Nvidia's Binary Instrumentation Tool

Mainak Chaudhuri
Dept. of Computer Science and Engineering, IIT Kanpur



## NVBit

- What is instrumentation?
  - A technique for inserting extra code into an application to observe/study/change its behavior
  - Primary purpose is program analysis and performance profiling
  - There are many other use cases such as architectural studies by intercepting function calls, or specific type of instructions, or all instructions
    - Warp divergence analysis, cache analysis, etc.
    - Any microarchitecture feature can be analyzed once the right set of instructions or API calls is captured



## How to do instrumentation

- Where to insert code
  - For cache study, insert new code for every load and store instruction
  - For warp divergence study, insert new code for every branch or memory instruction
- What code to insert
  - For cache study, new code should model a cache or a cache hierarchy
    - Input to this code is the intercepted load/store
  - For warp divergence study, new code should capture degree of divergence in a warp
    - Input to this code is the intercepted branch/memory instruction



## How to do instrumentation

- How to insert new code
  - Source code instrumentation
    - Directly changes the source code
    - Drawbacks: need access to source code, cannot instrument external libraries, need recompilation
  - Static binary instrumentation
    - Directly changes the binary after it is compiled
    - Positive: does not need access to source code
    - Drawbacks: branch targets need careful manipulation, cannot instrument dynamically loaded libraries and objects
  - Dynamic binary instrumentation
    - Inject code in binary at run-time (a la JIT comp.)
    - Positives: source code not needed, can instrument anything that runs as part of the binary's AS

## NVBit

- Does dynamic binary instrumentation
  - Takes two inputs: the application binary and an instrumentation program written in C/C++
    - The instrumentation program describes where in the binary to insert new code and what code to insert (writing this program is the main task)
    - The instrumentation program is known as an nvbit tool and is compiled as a shared object
    - As the application runs, the nvbit tool is used to dynamically and incrementally instrument part of the binary that is about to execute
      - A function is instrumented only once by maintaining state per instrumented function
    - Concepts similar to just-in-time compilation are used to affect dynamic instrumentation
      - Instrumented code is just-in-time compiled to generate the instrumented binary



## NVBit tool

- Entry into nvbit tool can take place at several points of a binary
  - Referred to as callbacks
  - All callback functions execute on the CPU
    - These are traditional C functions
  - When the binary starts, nvbit_at_init () gets called
    - Use this to initialize any states and structures that the instrumentation procedure would need
  - When the binary is about to exit, nvbit_at_term () gets called
    - Use this to write out any collected stats or close files or clean up structures



## NVBit tool

- Entry into nvbit tool can take place at several points of a binary
  - When a GPU context is created, nvbit_at_ctx_init (CUcontext) is called
    - Use it to collect information about CUcontext at start
  - When a GPU context terminates, nvbit_at_ctx_term (CUcontext) is called
    - Use it to collect information about CUcontext at end
  - Before the first kernel is launched, nvbit_tool_init (CUcontext) is called
    - Use it to initialize any CUcontext-dependent state needed for instrumentation



## NVBit tool

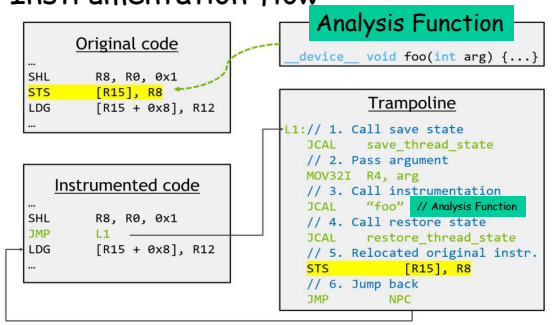
- Entry into nvbit tool can take place at several points of a binary
  - At the beginning and at the end of every CUDA driver API call, nvbit_at_cuda_event (CUcontext ctx, int is_exit, nvbit_api_cuda_t cbid, const char* name, void *params, CUresult* status) is invoked
    - is_exit is 0 if it is beginning and 1 if it is end of API
    - cbid is the unique identifier of driver API
    - name is the API name
    - params is a pointer to a structure containing API parameters
    - *status is the error code (if any) of the API call

# NVBit tool

- Entry into nvbit tool can take place at several points of a binary
  - The *instrumentation code* is included in the nvbit_at_cuda_event function
    - For example, CUDA APIs for launching a kernel can be identified from cbid and the kernel can be instrumented if not already instrumented
    - Inserts calls to the to be injected function (may call it *analysis function*) at appropriate locations and compiles the new binary that will be executed
    - Adds two sources of overhead to the overall execution time of a running application: extra time to compile the new binary and extra time to execute the instrumentation procedure and the analysis function

# NVBit tool

- Instrumentation flow




Dynamic selection between original and instrumented code

Source: Villa et al. MICRO 2019.

10

# Injecting instrumentation code

- Instrumentation APIs of NVBit are used to inject calls to analysis function
  - The extra code that injects such a call is executed every time nvbit_at_cuda_event function is called
    - This overhead is proportional to the number of CUDA API driver calls
  - The injected analysis function executes once per instrumentation grain
    - For now, we will assume that this is once per dynamic instruction of the instrumented kernel
    - This introduces the lion's share of the total overhead due to execution of extra code

11

# Injecting instrumentation code

- Instrumentation APIs of NVBit are used to inject calls to analysis function
  - The analysis function to be injected must be a legitimate function that can run on the GPU
  - The function nvbit_insert_call (Instr* i, const char* fun_name, ipoint_t point) is used to insert a call to the function with name "fun_name" before or after instruction i
    - point is IPOINT_BEFORE or IPOINT_AFTER
  - Each argument to the analysis function can be passed using the appropriate flavor of nvbit_add_call_arg_flavor (Instr* i, …)
    - Argument order should be maintained

12

## Injecting instrumentation code

- Most commonly used flavors of nvbit_add_call_arg_*
  - nvbit_add_call_arg_guard_pred_val (Instr*)
    - Pass the predicate of the instruction
  - nvbit_add_call_arg_const_val32 (Instr*, uint32_t)
    - Pass a 32-bit value as argument
  - nvbit_add_call_arg_const_val64 (Instr*, uint64_t)
    - Pass a 64-bit value as argument
  - nvbit_add_call_arg_reg_val (Instr*, int rn)
    - Pass the contents of register number rn as argument



## Injecting instrumentation code

- An instruction i can be removed from the original binary using the nvbit_remove_orig (Instr* i) call
  - Typical use case is to replace an instruction by an injected analysis function
- To select between instrumented or original kernel at run-time, nvbit_enable_instrumented (CUcontext, CUfunction, flag) call is used
  - Must be called at the beginning of the kernel launch API to decide whether the original kernel or the instrumented kernel will run



## Injecting instrumentation code

- Injected analysis function can examine and update register values
  - int32_t nvbit_read_reg (uint64_t rn)
    - Returns the value in register rn
  - void nvbit_write_reg (uint64_t rn, int32_t v)
    - Writes v into register rn
  - int32_t nvbit_read_pred_reg ()
    - Returns the calling warp's predicate register value
  - void nvbit_write_pred_reg (int32_t v)
    - Writes v into the calling warp's predicate register



## Inspection APIs

- Instrumentation code usually needs to inspect the binary of the kernel for inserting calls to the analysis function
  - To get a vector of all functions called by the kernel
    - nvbit_get_related_functions (CUcontext ctx, CUfunction f) where f is the target kernel
  - To get a vector of all instructions in program order for a function f
    - nvbit_get_instrs (CUcontext ctx, CUfunction f)
  - To get the static CFG of function f
    - nvbit_get_CFG (CUcontext ctx, CUfunction f)
    - A CFG has a vector of basic blocks (member bbs)

## Instrumentation granularity

- Instrumentation can be done at different grains
  - For each instruction
  - For each basic block
    - A basic block is a sequence of instructions terminating at a control flow changing instruction
    - A basic block has a unique single entry point and a unique single exit point

17

## Instrumentation granularity

- Instrumenting at a coarser grain can speed up the instrumented code
  - Less instrumentation overhead
  - For example, the instructions can be counted per basic block and then added to the total instruction count on each visit to a basic block

18

## Code reference

- Read the header files to know the available NVBit APIs
  - core/nvbit.h lists all NVBit APIs and structure definitions
  - core/cuda.h and core/generated_cuda_meta.h list all CUDA structure definitions and constants
  - core/tools_cuda_api_meta.h lists CUDA API to callback id mapping
  - core/instr_types.h lists all instruction-related structures and constants

19