Decoding CUDA Binary

Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, Eddy Z. Zhang
Department of Computer Science, Rutgers University, NJ, USA
arihayes@cs.rutgers.edu, fh214@scarletmail.rutgers.edu, andy.jinhuang@rutgers.edu,
yc827@cs.rutgers.edu, zz124@scarletmail.rutgers.edu

***Abstract*—NVIDIA's software does not offer translation of assembly code to binary for their GPUs, since the specifications are closed-source. This work fills that gap. We develop a systematic method of decoding the Instruction Set Architectures (ISAs) of NVIDIA's GPUs, and generating assemblers for different generations of GPUs. Our framework enables cross-architecture binary analysis and transformation. Making the ISA accessible in this manner opens up a world of opportunities for developers and researchers, enabling numerous optimizations and explorations that are unachievable at the source-code level. Our infrastructure has already benefited and been adopted in important applications including performance tuning, binary instrumentation, resource allocation, and memory protection.**

***Index Terms*—CUDA, GPU, Code Generation, Code Translation and Transformation, Instruction Set Architecture (ISA)**

I. Introduction

The CUDA parallel computing platform developed by NVIDIA has benefited greatly from the popularity of their powerful GPU hardware, seeing widespread use. But the proprietary, closed-source nature of this platform makes research more difficult and limited. Although NVIDIA provides documentation of the intermediate language, they provide very little information about the hardware-specific instruction sets.

Open-sourcing of software and hardware has two-fold benefits. The level of documentation that open-sourcing permits allows for better tuning of specific software. It also allows the community to spot errors and security flaws that can be corrected before they become more widespread.

Openness of an instruction set architecture (ISA), in particular, allows the development of more effective compilation techniques. The proprietary compiler is not guaranteed to be optimal when generating executable code. Indeed, it is usually the case that code can be further tuned to improve its efficiency. For example, by using known instruction encoding to tune allocation and scheduling, several works [1] [2] [3] [4] are able to achieve performance beyond that of what NVIDIA's compiler, nvcc, can produce on its own. Furthermore, several other works [5] [6] [7] [8] [9] [10] demonstrate that modifications to the binary code can allow for useful techniques that are impossible for a programmer to fully implement in source or intermediate language.

An instruction set architecture (ISA) acts as the interface between software and hardware. The ISA allows independent development of software and hardware. The details the ISA describes, such as binary instruction encoding and data types, are vital to the development of compilers. An open ISA helps research in GPU micro-architecture. There is a large number of GPU works that rely on GPUGPU-Sim [11]. However, the cycle-level GPU simulator is unable to run the binary code used by actual NVIDIA devices, thus reducing its accuracy - as well as the accuracy of any works which make use of it. Further, an open ISA may help enhance GPU security: the work by Hayes and others [10] showed that understanding the ISA details is a necessity when tracking and protecting sensitive data across the whole system.

We develop a framework of techniques that enables easier interaction with the ISA of various NVIDIA architectures. Our work has found use in many different settings for program analysis and performance tuning of GPUs [6] [7] [10] [12].

In particular, we make the following contributions:

- We design a method which largely automates the creation of a GPU assembler. [1] The method first analyzes binary code and then generates a translator for converting assembly code to binary code. This automation allows researchers to quickly add support for new ISAs.
- We provide the binary encoding for multiple existing GPU architectures, including how operands and opcodes are mapped to 0s and 1s, and how instruction dependence is handled. We have made our findings freely available on Zenodo, with opcodes at [13], and operands at [14].
- We provide the GPU *ELF* file format in Zenodo [15], which is necessary for complete editing of executable.
- Our techniques and framework have demonstrated compatibility with at least four consecutive major versions of NVIDIA devices: Compute Capability 2.x (Fermi), Compute Capability 3.x (Kepler), Compute Capability 5.x (Maxwell) and Compute Capability 6.x (Pascal).
- We develop an extensible framework that supports analysis and transformation of assembly code for use with the GPU assemblers, which has been used in multiple works [6] [7] [10] [12]. Our framework is available on GitHub at https://github.com/decodecudabinary/Decoding-CUDA-Binary.

Prior studies have been trying to understand the hidden workings of NVIDIA's devices. The work by Wong *et al.* [16] focuses on micro-architecture details such as cache design and memory latency. There have also been efforts at decoding binary instructions: Hou *et al.* [17] created an assembler for Compute Capability 2.x, Zhang *et al.* [4] decoded Compute Capability 3.x instructions used in an SGEMM implementation, Gray [3] developed an assembler for Compute Capability

[1] Artifact available at: https://doi.org/10.5281/zenodo.2337060

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

5.x, and Jia *et al*. [18] decoded Compute Capability 7.x instructions. Each of these existing works is limited to a specific architectural generation, and in some cases a specific application. Our work is a comprehensive and cross-architecture framework that is useful for general-purpose and future-proof applications.

The rest of the paper is organized as follows. In Section II, we provide an overview of our framework. We compare CPU ISA with GPU ISA in Section II-B. Section III describes our automatic decoding method. Section IV summarizes our findings for different generations of NVIDIA GPUs. Section V outlines applications of our framework. We discuss related works in Section VI.

II. Overview and Background

The assembly code used for NVIDIA devices is known as "SASS", which corresponds exactly to the actual binary code. Most details of SASS are kept secret. In addition to the closed-source compiler, **nvcc**, NVIDIA provides a closed-source disassembler called **cuobjdump**, which can translate the binary into SASS assembly code. However, NVIDIA offers no means of translating the SASS to binary, leaving researchers unable to make use of the assembly code. Our work fills this gap.

A. Assembler Generating Framework

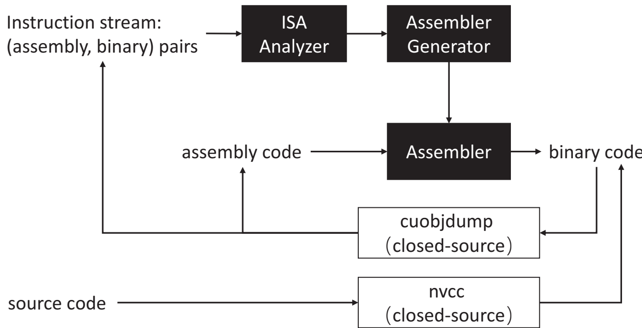


Fig. 1. Our assembler generation framework. Our work is highlighted with black background, and the existing tools are in white background.

We show our assembler generating framework in Fig. 1. We first use *nvcc* to generate binary code from source programs, and then we use *cuobjdump* to retrieve the SASS assembly, as well as the mapping between the assembly and binary. Our ISA Analyzer will automatically generate variants of given {*a*ssembly, binary} pairs to enrich the data set for analysis.

Next, our ISA Analyzer starts analyzing the binary encoding of different components of each type of instructions, for instance, which bits correspond to opcode, operands, modifiers, etc. The ISA Analyzer yields a list of recorded operations, that is, the set of decoded assembly/binary instructions. An example of the recorded IADD instruction for computing capability 3.5 is shown in Fig. 2.

Our ISA analyzer exploits the fact that there is one-to-one mapping between each SASS assembly instruction and each binary instruction in the listing generated by the disassembler *cuobjdump*. An example of cuobjdump's output is shown in Fig. 3: every assembly instruction corresponds to one 64-bit binary instruction (represented as a hex number). By retrieving enough {*a*ssembly, binary} pairs (Section III-B), we can decode the instruction set architecture.

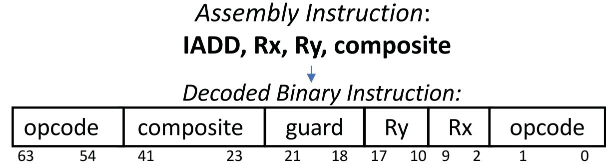


Fig. 2. Decoded IADD instruction for computing capability 3.5. We show which bits correspond to which component in the IADD instruction.

Our Assembler Generator takes as input the binary encoding information generated by our ISA Analyzer, and outputs an Assembler. The generated assembler can convert assembly code to binary code with respect to the underlying GPU architecture. This corresponds to the conversion from assembly to binary in the compilation process shown in Fig. 1.

Performing the decoding analysis manually is a slow and arduous process, especially considering the frequency with which the GPU ISA changes - usually once per GPU generation. The framework we developed is important as it can automatically decode the ISA and generate assemblers.

Furthermore, we incorporated our assemblers into a larger framework that permits analysis and transformation of assembly code for multiple GPU architectures, enabling several applications which we will discuss in Section V.

B. Unique CUDA Binary Instructions & Behavior

In this section we describe the unique features of GPU ISA compared with CPU ISA. We summarize the differences from the following aspects: memory instruction, control flow, dependence handling, register shuffling, barrier setup, and compile-time instruction scheduling.

Memory Instructions: There are different types of memory on the GPU, each associated with a type of assembly/binary instructions, as shown in Table I. Global memory can be accessed by all threads. Local memory is a range of thread-private memory in GDDR memory. Shared memory is an on-chip memory that is private to each thread block.

```
IMAD R3, R0, c[0x0][0x28], R3;            0x51080c00051c000e
MOV R10, RZ;                              0xe4c03c007f9c002a
ISCADD R0.CC, R3, c[0x0][0x148], 0x2;     0x60c40800291c0c02
I2F.F32.S32 R2, c[0x0] [0x150];           0x65c000002a1ca80a
IMAD.HI.X R3, R3, R4, c[0x0][0x14c];      0x93181000299c0c0e
S2R R4, SR_CLOCKLO;                       0x86400000281c0012
IADD32I R10, R10, 0x1;                    0x40000000009c2829
```

Fig. 3. An example of output from cuobjdump, compiled for NVIDIA's Compute Capability 3.5 architecture. We have omitted certain details such as instruction address for illustration purpose.

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

TABLE I
COMMON MEMORY INSTRUCTIONS ON GPU. WE REFER TO EACH GENERAL REGISTER AS RX OR RY, AND EACH LITERAL AS 0XA FOR ILLUSTRATION PURPOSE.

Assembly	Description
LDG Ry, [Rx+0xa]	Load from global memory
STG [Rx+0xa], Ry	Store to global memory
LDL Ry, [Rx+0xa]	Load from local memory
STL [Rx+0xa], Ry	Store to local memory
LDS Ry, [Rx+0xa]	Load from shared memory
STS [Rx+0xa], Ry	Store to shared memory
LDC Ry, c[0xa][Rx+0xa]	Load from constant memory
TEX Ry, Rx, 0xa, 0xa, 0xa	Texture fetch

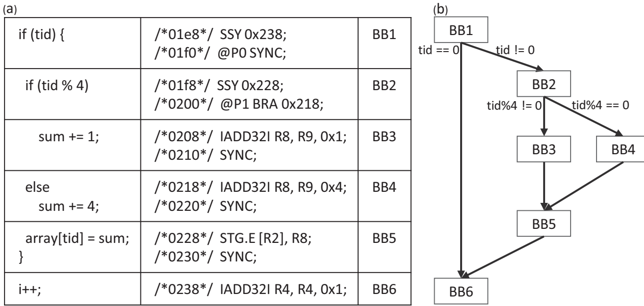


Fig. 4. A simple example of thread-warp divergence. (a) is the source and assembly code, (b) is the control flow graph.

Constant memory is a type of read-only memory that is split into logical banks. Texture memory is another type of read-only memory, but is only accessed through texture instructions.

Divergence and Re-convergence: GPU threads are organized into thread-warps, threads within a warp run the same instruction at one time. When threads inside a warp need to diverge on the execution path, the program uses the SSY instruction to signal the hardware to prepare and specify the address for re-convergence. Threads that take different execution paths will be serialized instead of running in parallel. When every thread in the warp has reached a re-convergence command - either a .S modifier or a SYNC instruction, depending on the architecture - it will wait until the thread warp reaches the instruction whose address is specified by the SSY instruction, and then return to running in lock-step.

We show an example of divergence and re-convergence handling in Fig. 4. The source code and assembly code is on the left, organized into basic blocks. The control flow graph is on the right, showing the paths the thread-warp may be split across. The SSY instruction in basic block BB1 indicates that threads may take different paths (diverge), and the instruction with 0x238 (BB6) is where they will re-converge. The conditional SYNC instruction causes some threads to immediately prepare for re-convergence, but others in the same thread-warp execute basic block BB2 instead. In BB2, the already divergent threads hit the next SSY, preparing for another potential divergence; some take the conditional branch to BB4, whereas others execute BB3 instead. At the end of BB3 and BB4, these doubly-divergent threads re-converge in BB5. At the end of BB5, they finally re-converge in BB6 with the threads that only executed the first SYNC from BB1.

Warp-Register Shuffling: Thread-warps are allocated consecutive physical registers, aligned to hardware-specified boundaries. Compute Capability 3.0 introduced the SHFL instruction, which can read registers from another thread in the same thread-warp. This permits a thread-warp to perform internal communication far faster than would otherwise be possible, thanks to the low latency of the register file as compared to other memory types.

Barrier Instructions: The BAR instruction is be used for synchronization, halting the thread until every other live thread in the same thread-block has reached its position. In Compute Capability 3.0 the TEXDEPBAR instruction was introduced, which halts the thread until specified texture operations are completely resolved; texture dependences are thus handled by the compiler, rather than determined by the device.

Compile-Time Scheduling: As of Compute Capability 3.0, instruction scheduling is handled by the compiler rather than by the hardware. On this architecture every $8-$th instruction, rather than being a real instruction, is a set of scheduling codes inserted by the compiler. These scheduling codes dictate the minimum number of cycles that the thread must wait between every two consecutive instructions in the following seven instructions in order to satisfy dependence constraints.

Starting with Compute Capability 5.0, NVIDIA moved even more control logic away from the hardware, saving power and space. Thus instruction-level barrier has been added to the scheduling codes generated by the compiler. The scheduling codes on Compute Capabilities 5.x and 6.x occur in place of every fourth instruction. As of Compute Capability 7.0, they are embedded into each individual instruction, rather than controlling larger blocks of instructions.

III. ASSEMBLER GENERATION

The ISA for NVIDIA GPUs changes in almost every major version. Prior works [17] [4] [3] [18] have analyzed the ISA, but only for a particular architecture. To ensure forward compatibility as the ISA continues to change across generations, we develop a generic method for decoding the instruction set and create a system which can generate GPU assemblers for newer architectures in a partially automated manner. As a result, we can quickly add compatibility for additional architectures to support open-source research work.

Our Assembler Generator makes use of our ISA Analyzer, shown in black background in Fig. 1. The ISA Analyzer takes in a list of binary and SASS assembly instruction pairs from NVIDIA's cuobjdump tool, and determines which bits in the binary instruction correspond to which components in the assembly instruction. To ensure we have enough input, we use bit flipping to prepare additional {*assembly, binary*} pairs for further analysis, as described below in Section III-B. Finally, the Assembler Generator component outputs C++ code capable of converting SASS assembly to binary.

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

A. Structure of an Instruction

When decoding an instruction, it is vital to gather and maintain information about every component of the instruction that affects the binary encoding. The first component is the *opcode*. We treat bits which indicate the expected operand types as part of the opcode. For example, if two instructions are both named IADD, but one of them adds two registers whereas the other adds a register to an integer literal, then we treat them as two distinct operations due to the different encoding.

The second component of the instruction is *operands*. CUDA operands can be one of the following types such as literal value, register, predicate register, special register, bit-field, memory, constant memory, texture shape, texture channel, and etc. Most of these are encoded with a single value, but memory operands may be represented by up to two values (register and offset), and constant memory by up to three (bank, register, and offset). Many instructions allow for one or more optional unary operations to be attached to their operands: arithmetic-negation, bitwise-complement, absolute value, and logical-negation, each of which is typically encoded as a single bit.

Hexadecimal (integer literal) operands need to be handled specially during analysis, because they are encoded differently depending on the type of instruction. For control flow instructions the assembly will typically use absolute addresses whereas the binary encoding will use relative offsets. Furthermore, a negative value might be handled by directly encoding a negative value, or by simply enabling a unary arithmetic-negation bit.

Another important component of the instruction is its optional modifiers. Most modifiers are attached to the *opcode* in the assembly, though there are exceptions, in which one or more of them are instead attached to operands. Some modifiers act as simple boolean values that flip a single bit, but many instructions have more complicated modifiers with a range of possible values. For example, several instructions make use of a two-bit logic modifier, which can be ".AND" if the operation will perform a logical **and**, ".OR" if the operation will perform a logical **or**, or ".XOR" if the operation will perform a logical **exclusive-or**. The GPU does not have separate instructions dedicated to each of these logical operations, so an understanding of these modifiers is valuable.

Certain instructions expect multiple modifiers of the same type, in which case their order has important meaning. For example, the PSETP (predicate set-predicate) instruction uses two logic steps to reduce three predicate registers into a single value. PSETP.AND.OR will apply **and** and then **or**, whereas PSETP.OR.AND will do the opposite and has a different encoding. This is why it is necessary to know the type of these modifiers. Another example is the format modifiers which can appear twice in cast instructions, such as the F2F instruction, which casts a numeric value from one floating-point format to another floating-point format. Using our knowledge of the types of each modifier, our Assembler Generator is able to determine whether or not a modifier follows another of the same type, and stores information about the first instance separately from the second instance.

The fact is that every part of nearly every type of instructions uses predictable rules to enforce values on particular bits with respect to the assembly instruction.

B. Analyzing the ISA

The first task before analyzing the ISA is to gather enough instruction $\{$*assembly, binary*$\}$ pairs as input. This is vital, since with little data, it is impossible to determine precisely which bits correspond to which parts of the instruction and which encoding rules are used. We start by gathering multiple executable from real source programs, extracting the assembly and binary mapping with cuobjdump, and giving them to our ISA Analyzer's parser. We gathered executable from the CUDA SDK [19] and Rodinia benchmark suite [20]. After processing all of the input samples, we expand our data set via our bit flipper module.

Our bit flipper takes the binary instruction of every known operation as input, and outputs variants of each one, which we can inject into an executable in order to extract more assembly code. This is similar to a technique used in [4]. Each variant generated by the bit flipper is identical to the instruction it is based on, except that a single distinct bit has been flipped. These flipped instructions will permit the analyzer to infer how different parts of the instruction are affected by each precise bit. Depending on which bits are changed, a new operation might be generated instead; in this case, we resume bit flipping, repeating the process until the results converge.

An important consideration for the bit flipper is that the disassembler may crash without producing output upon encountering unexpected instructions. Our solution is to narrow the range of bits that are flipped - skipping over most of the opcode bits. An alternate solution would be to disassemble flipped instructions one at a time, so that the disassembler's crashes do not prevent retrieval of any valid instructions. This alternate solution may lead to more complete data set, but in practice we find that this is unnecessary; our faster implementation is sufficient to prepare an assembler which can reproduce every program we have tried.

The first time the analyzer encounters an operation, it makes the broadest possible assumptions. For the opcode, each modifier, and each unary operation, it assumes that every bit in the binary instruction is important to the encoding (setting every value to True inside a boolean array), and records the current values. For each operand, it searches for every subset bits of the instruction which matches each possible interpretation of each of the operand's values, recording the maximum possible matching bit-sequence(s). Upon encountering another instance of a known operation, it is compared to the recorded data, narrowing down which bits are actually important to each part of the instruction.

For example, suppose we encounter the two instances of FFMA in Fig. 5, and are trying to decode the first operand. During the first instruction the operand is R9 (register 9), so

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

FFMA **R9**, R4, R5, RZ; **R9: 1001**

1[**1001**]100000000111111110000000000000000001[0**1001**]11000001[000000**1001**]10

FFMA **R5**, R5, R6, R5; **R5: 0101**

110011[000000000000**0101**]000000000000000000011000111[0000**0101**][0000**0101**]10

Bit 63 Bit 2

Fig. 5. Looking for the bits controlled by the first operand.

we look for the value 9 (1001 in binary format) in the binary. We identify three matching bit-sequences of size 10, 5, and 4, respectively starting from bit 2, bit 19, and bit 59. Therefore we set the maximum size at these locations to 10, 5, and 4, and set the maximum size for other locations to 0. During the next FFMA, the same operand is R5, so we look for value 5 (101 in binary format) in the opcode. The position (bit 2) which we previously marked as having the size 10 (for the bit-sequence length) only has size 8 now, thus we reduce its value to 8. The other two non-zero positions no longer contain the operand's value, so we reduce them each to 0. At this point, we have correctly identified that the first register in FFMA assembly instruction controls exactly the eight bits which start at bit 2.

Algorithm 1 Binary Instruction Decoding

Global: knownOps is list of known encodings
Input: a is a parsed assembly instruction
Input: binary is binary code corresponding to a
Result: knownOps is updated

```
 1: procedure ANALYZEINST(ASSEM a, bool[] binary)
 2:     OPERATION op ⇐ knownOps.lookup(a.opcode)
 3:     if op == null then
 4:         op ⇐ knownOps.insert(a.opcode)
 5:         op.opcodeBinary ⇐ binary
 6:         op.opcodeBits ⇐ {true, true, ..., true}
 7:         for each OPERAND 'oprd' in op.operands do
 8:             oprd.comp[0:2].size ⇐ {64, 63, ..., 1}
 9:     for each bit 'b' in binary do
10:         if binary[b] != op.opcodeBinary[b] then
11:             op.opcodeBits[b] ⇐ false
12:     for each string 'm' in a.mods do
13:         if m not in op.mods then
14:             op.mods.insertModifier(m)
15:             op.mods[m].binary ⇐ binary
16:             op.mods[m].bits ⇐ {true, true, ..., true}
17:         for each bit 'b' in binary do
18:             if binary[b] != op.mods[m].binary[b] then
19:                 op.mods[m].bits[b] ⇐ false
20:     for each OPERAND 'oprd' in op.operands do
21:         ASMOPERAND asmOprd ⇐ a.operands[oprd]
22:         analyzeOperand(oprd, asmOprd, binary)
23: end procedure
```

Fig. 6 shows the structures used in this section's pseudocode. The key structs here are ASSEM, which is used to hold a parsed assembly instruction, and OPERATION, which is used to

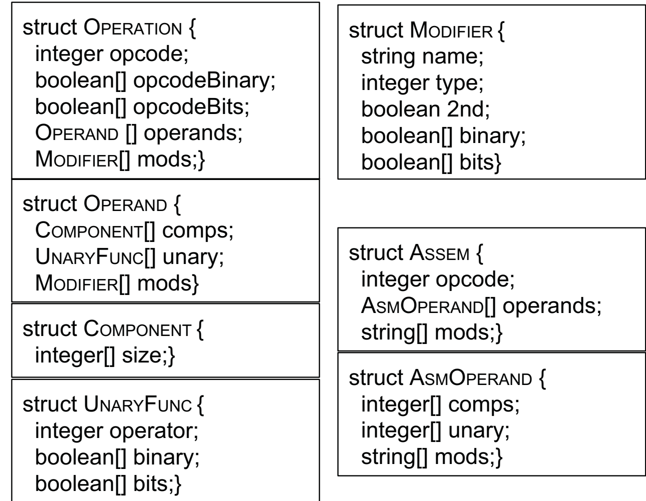


Fig. 6. Structures used in our Algorithms' pseudocode.

Algorithm 2 Binary Operand Decoding

Input: $oprd$ is an operation's operand
Input: $asmOprd$ is an operand's assembly
Input: $binary$ is an instruction's binary
Result: oprd is updated

```
 1: procedure ANALYZEOPERAND(OPERAND oprd, ASMOPERAND asmOprd, bool[] binary)
 2:     for each COMPONENT 'comp' in oprd.comps do
 3:         for each bit 'b' in comp do
 4:             for size = comp.size[b] to 0 do
 5:                 if binary[b:b+size] == comp.value
 6:                     comp.size[b] ⇐ size
 7:                     break
 8:     for each unary 'un' in asmOprd.unary do
 9:         if un not in oprd.unary
10:             oprd.unary.insert(m)
11:             oprd.unary[un].binary ⇐ binary
12:             oprd.unary[un].bits ⇐ {true, true, ..., true}
13:         for each bit 'b' in binary do
14:             if binary[b] != oprd.unary[un].binary[b]
15:                 oprd.unary[un].bits[b] ⇐ false
16: end procedure
```

maintain analysis of an instruction. The ASSEM struct contains an identifier for the opcode, a list of operands, and a list of modifier strings. The ASMOPERAND struct, which represents the assembly for a single operand, contains the components of the operand's value (since some memory operands have up to three discrete values) a list of unary operators (such as negation, absolute value), and a list of modifier strings. The OPERATION struct, in addition to lists analogous to those of the ASSEM struct, contains the 'opcodeBinary' array which holds the binary code from one instance of the associated instruction, and 'opcodeBits' which indicates which parts of opcodeBinary have remained consistent across different

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

instances of the instruction. The Operand struct used by Operation is analogous to the ASMOPERAND struct. The COMPONENT struct used by OPERAND contains the 'size' array, which holds the maximum valid size for its operand component at each possible bit position. The UNARYFUNC struct, representing a unary operator, holds the operator type, the binary code from a single instance of the instruction where the operator was present, and the 'bits' array which indicates which parts of the binary have been consistent across instances. Finally, the MODIFIER struct used by OPERATION and OPERAND represents a modifier; it includes the modifier's name, its type, whether it's the second instance of its type, the binary code associated with a single instance of the instruction in which this modifier was present, and the 'bits' array which indicates which parts of the binary have been consistent across instances.

Algorithms 1 and 2 describe the ISA analyzer. At a high level, the goal of these functions is to analyze every individual component of the instruction's assembly, by updating and comparing the instruction's binary value when these individual component are present.

Algorithm 1 takes an instruction in its assembly format and its binary format. It identifies the corresponding operation in the list of known encodings (by looking up the opcode), or generates a new one with default values if the given assembly instruction is not yet a known operation. It then performs the analysis: examining the instruction's opcode bits, the predicate guard, and its modifiers, updating the operation's encoding record accordingly. The helper function in Algorithm 2 is used to analyze each of the instruction's operands, identifying the bits associated with their modifiers, unary operators, and the components of their values (up to three). An example of the search for the value-components was shown in Fig. 5, with register operands that have only a single component (the register ID).

By applying the above algorithm to inputs which include the extra instruction variants created by the bit flipper, we obtain the necessary information for the Assembler Generator to understand all of the instructions and values encountered.

C. Assembler

Once we have provided the assembler generator with sufficient input, we can use it to output an actual assembler in C++ language. Algorithm 3 contains pseudo-code for generating the assembler and making use of the list of encodings yielded by Algorithm 1. It loops through the complete list of decoded operations and creates separate conditional blocks for each operation. The operation list includes, for each component of the instruction, a list of its associated bits and their expected values. For each component of an assembly instruction, Algorithm 3 generates code to set its binary bits appropriately: the opcode bits, each modifier, and so on.

Fig. 7 shows an example of generated assembler code. The first conditional block executes if the instruction is an IADD (integer addition) operation: it sets the opcode bits based on recorded values for IADD and each of the three operands' bits correspondingly. Additional code blocks are generated for each type of operations. In the end it sets the conditional guard's bits, and returns the final binary format.

Our actual implementation is more optimized and complicated than the example in Fig. 7. We omit those details for illustration purpose. For example, to make the assembler more efficient, we distinguish between operands that are encoded the same for every operand-list and those which differ; the assembler generator only needs to print the shared operands once for the entire opcode, whereas the others need to be printed once per operand-list. Additionally, upon encountering anything unexpected, such as a modifier that the assembler generator does not recognize, our generated assemblers are designed to print an error message to the standard error stream.

Algorithm 3 Assembler Generator

Input: *operations* is the list of binary ISA encodings
Result: output the source code of an assembler

```
1:  procedure genAssembler(Operation[] operations)
2:      for each Operation 'op' in operations do
3:          print "if opcode == " op.opcode
4:          for each Operand 'oprd' in op.operands do
5:              for each Component 'c' in oprd.comps do
6:                  for each bit 'b' do
7:                      integer s ⇐ c.size[b]
8:                      if s > 0 then
9:                          print "binary[b:b+"s "] = c.value"
10:         print "for each Modifier 'm'"
11:         for each Modifier 'm' do
12:             print "if m.name == " m.name
13:             print "if seenModType[m.type] == " m.2nd
14:             for each bit 'b' in m.binary do
15:                 if m.bits[b] then
16:                     print "binary[" b "] = " m.binary[b]
17:             print "end if"
18:             print "seenModType[m.type] ⇐ true"
19:             print "end if"
20:         print "end for"
21:         print "else"
22:     print "throw error"
23:     print "end if"
24: end procedure
```

IV. DECODING SUMMARY

In this section we describe the findings we have gathered with the help of our Assembler Generator. Although instructions are of fixed length, NVIDIA's instruction sets lack the relative simplicity of a RISC architecture. It includes complicated instructions such as multiplication-and-addition, multi-function operation that performs trigonometric functions including sine and cosine, and so on. Although we can make generalizations about which bits are used for which components of the instruction, there are few consistent rules across different instructions.

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

```
procedure assembler (instruction inst)
    if inst.opcode == "IADD" then
        binary = opcode_Table3x[IADD];
        binary[2:9] = inst.operands[0].value;
        binary[10:17] = inst.operands[1].value;
        binary[23:41] = inst.operands[2].value;
    else if inst.opcode == "MOV" then
    ...
    end if
    //auto-generated code ends here
    if inst.guard then
        binary[18:21] = inst.guard;
    end if
    return binary;
```

Fig. 7. A simplified example of an assembler that has been automatically generated.

TABLE II
COMMON INSTRUCTIONS FOR COMPUTE CAPABILITY 3.X, WITH OPERANDS DEFINED IN TERMS OF FIG. 8. MEMORY OPERANDS ARE DENOTED WITH SQUARE BRACKETS. PC REFERS TO THE PROGRAM COUNTER, LOP REFERS TO AN ARBITRARY LOGIC OPERATION, AND COMP IS SHORT FOR COMPOSITE.

Instruction	Effect
MOV reg1, comp	reg1 ⇐ comp
S2R reg1, special_reg	reg1 ⇐ special_reg
IADD reg1, reg2, comp	reg1 ⇐ reg2+comp
IMUL reg1, reg2, comp	reg1 ⇐ reg2×comp
IMAD reg1, reg2, comp, reg4 IMAD reg1, reg2, reg4, comp	reg1 ⇐ reg2×comp+reg4 reg1 ⇐ reg2×reg4+comp
PSETP p2, p1, p3, p4, p5	p2 ⇐ p3 LOP p4 LOP p5; p1 ⇐ !p2
BRA const/lit comp	PC ⇐ const/lit comp
CAL const/lit comp	callstack.push(PC); PC ⇐ const/lit comp
RET	PC ⇐ callstack.pop()
LD reg1, [reg2 + 32-bit lit]	reg1 ⇐ [reg2 + 32-bit lit]
ST [reg2 + 32-bit lit], reg1	[reg2 + 32-bit lit] ⇐ reg1

Table II shows a set of common instructions used in GPU, and their effects. We use 'regX' to refer to 32-bit general registers, 'pX' to refer to 1-bit predicate registers, and 'composite' to refer to operands with multiple possible types. The mapping between the bit-sequence and the operand/opcode/modifier in the assembly instruction is shown in Fig. 8, for instance, reg1 bits are 2 to 9 in computing capability 3.x. We have provided a complete table of opcodes for all decoded instructions in Zenodo [13].

A. Operands

There are several operand types we observe in the assembly code. First, 32-bit general **registers**, which we usually just call **registers**, are used in most instructions. Depending on the ISA, these registers' index is encoded as either six bits or eight bits. The register with maximum ID (either 63 or 255) is the read-only zero register, which always holds a value of zero, and is written as RZ in assembly code. In operations which use 64-bit or larger values, the GPU will use a range of consecutive registers, starting with the register specified in the instruction.

TABLE III
THE MOST COMMON SPECIAL REGISTERS USED ON GPU.

Special Register	Encoding	Meaning
SR_TID.X	33	Thread ID (x-dimension)
SR_TID.Y	34	Thread ID (y-dimension)
SR_TID.Z	35	Thread ID (z-dimension)
SR_CTAID.X	37	Thread-Block ID (x)
SR_CTAID.Y	38	Thread-Block ID (y)
SR_CTAID.Z	39	Thread-Block ID (z)
SR_CLOCK_LO	80	Cycle Counter (32 bits)

Predicate registers hold boolean values, and are encoded as three bits. Additionally, predicates can be used as conditional **guards** for most instructions, allowing individual threads to selectively skip execution of instructions. The conditional guard's binary is in a different location for each distinct ISA, but is always four bits: the lower three bits are the predicate register ID, and the highest bit is used for logical negation. The predicate register with ID 7 is the null predicate, which always holds a value of true, and is written as PT in the assembly code.

Special registers are only used in the S2R instruction, and are encoded with eight bits. In the assembly code they can be written as SRx for some value x, but are more commonly used with the English name of the associated value. For example, the special register "SR_CLOCKLO" can be used to retrieve the lowest 32-bits of the clock value, and the special register "SR_TID_Y" can be used to retrieve the current thread's ID along the thread block's y-dimension. Table III shows common special registers and their encoded values. The encodings for most special registers remain the same across GPU generations, though additional special registers are added over time. We have provided a more complete list of special registers in Zenodo [21].

Memory operands have a register and a literal offset, regardless of whether they are used for global memory, local memory, or shared memory. The literal offset is usually either 24 bits or 32 bits, depending on the instruction.

Constant memory operands point to a type of read-only memory. There is a dedicated load-constant-memory (LDC) instruction, but constant memory operands are also usable in other instructions. They have up to three components: a memory bank, a register, and a literal offset. The bank and offset use a combined 19, 20, or 21 bits for their encoding, depending on the ISA and the instruction. When using 19 or 21 bits, the highest 5 are the bank, and the rest are the offset. When it uses 20 bits, the lowest 16 are the offset, and the highest four and lowest one are the bank. The register component, which is added to the offset, is only used in the LDC instruction and some control-flow instructions.

Texture operations have a texture operand, whose value is a shape such as 1D, 2D-Array, CUBE, etc, and is encoded as three bits. On some architectures, texture operations also have a channel operand with some combination of "R", "G", "B", and "A", encoded with up to four bits.

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

Barrier operations may use an SB operand (written as SBx for some integer x) and a bitfield operand (written as a set of bit indices).

Finally, instructions may use literal, numeric values of various sizes. In assembly code these are usually written as hexadecimal values, though in floating-point operations they may be written in decimal instead. Notably, the encoding for floating-point literals does not quite match the IEEE Standard, due to lack of enough bits - for example, in some instructions 19 bits may be used to try to hold a 64-bit double-precision floating point value. These floating point values are handled by discarding the lowest bits until the remaining portion fits in the available space.

We use the term 'composite' to refer to a single operand that has multiple possible types. On Compute Capabilities 2.x and 3.0, the most common composite operand is 20 bits; it can either hold a 20-bit literal, a 6-bit register ID, or 20-bit constant memory location without a register component. We have also seen 16-bit composites that can either hold a literal or a register.

On Compute Capabilities 3.x through 6.x, the most common composite is 19 bits; it can either hold a 19-bit literal, an 8-bit register, or a 19-bit constant memory location without a register component. We have also seen 24-bit and 32-bit composites which can hold either a literal of that size or a 21-bit constant memory location without the register component.

In Fig. 8, we show common locations and sizes for different operands in each of the architectures that we have studied. In Zenodo [14] we have provided the operands for specific instructions that have been decoded. In Table II, we show a subset of this data, with several common instructions for version 3.x of the ISA. We refer to general registers as regX and predicate registers as pX with integer values x, and texture shapes as tex.

B. ISA Change Over Time

Compute Capability 2.x: The earliest hardware we consider is NVIDIA's "Fermi" generation, which includes devices with Compute Capabilities of 2.0 and 2.1. Instructions can be either 4 bytes or 8 bytes, but we have only seen the compiler generate 8 byte instructions. We find that the executable uses little-endian format for instructions, but CUDA's disassembler instead displays the hexadecimal values of each instruction starting with the most significant byte. This architecture handles instruction-level scheduling via the hardware.

Compute Capability 3.0: The earliest of NVIDIA's "Kepler" generation has Compute Capability 3.0. For this architecture, the ISA is very similar to the previous one, with every pre-existing instruction having exactly the same binary encoding as before, though some additional instructions have been added.

The major difference between the ISA of this architecture and the previous is that some instruction scheduling has been offloaded from the hardware to the compiler. For Kepler devices, every eighth instruction (including the first one) is not a real instruction, but instead contains the instruction scheduling information for GPU. [17] developed an assembler for Compute Capability 2.x, but also explored this architecture due to its similar ISA. The aforementioned latency instructions have no official name of which we are aware, but Asfermi used the name **SCHI** to represent them, which we also use in this work.

We refer to the least significant bit as bit 0, and the most significant bit as bit 63. Bits 0-3 of the SCHI instruction contain the value 7. Bits 4-11 control the minimum number of cycles that the GPU must wait between dispatching the first following instruction and the second. Bits 12-19 control the minimum cycles between the second following instruction and the third. This continues for the next five groups of eight bits. Finally, bits 60-63 hold the value 2. The seven dispatch values can hold a value of 0x4 when the associated instruction is able to be dispatched in the same cycle as the next, or a value between 0x20 and 0x3f to mean the minimum number of cycles between them is the value minus 15.

Compute Capability 3.x: The remainder of the Kepler generation comprises Compute Capabilities 3.2, 3.5, and 3.7. Unlike the earlier Kepler devices, these GPUs allow threads to use up to 256 registers instead of 64, so the six bits which the previous ISA's instructions used to indicate the register are insufficient. Therefore, although the assembly code looks much like that of the previous generation, every instruction has a new encoding. A SCHI instruction now holds the value 0 in its two least significant bits and the value 2 in six most significant bits - the seven dispatch intervals have been shifted two bits, but are otherwise unchanged.

The manner in which the disassembler presents SCHI instructions on Kepler is not conducive to analysis or editing. It provides the 64-bit binary value of the entire SCHI, but offers no indication of its meaning. Furthermore, since the SCHI describes a group of consecutive instructions, addition, deletion, or reordering of instructions becomes more complicated. To solve these issues, we use our framework to split up each SCHI, and place their values in-line with the assembly instructions, as described below.

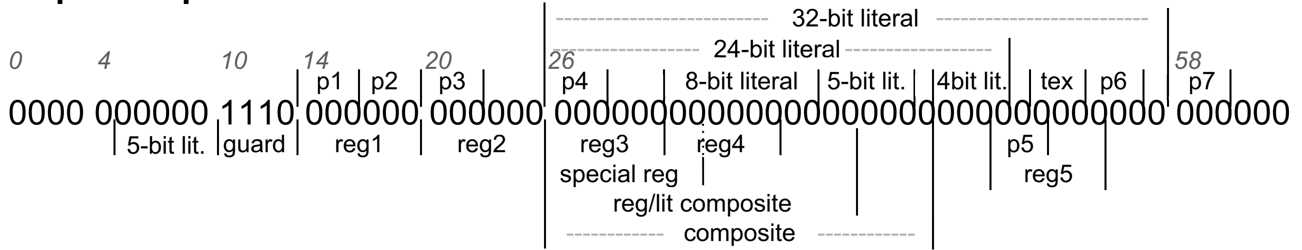
Fig. 9 shows how we extract the separate SCHI values on Compute Capability 3.x, associating them with individual instructions. In this example, the scheduling codes indicate that the thread waits at least 0x2f minus 0x1f cycles (16) after dispatching the first instruction, 0 cycles after dispatching the second instruction, 0x23 minus 0x1f cycles (4) after each of the third and fourth instructions, 1 cycle after the fifth instruction, 3 cycles after dispatching the sixth instruction, and 9 cycles after dispatching the seventh instruction. The compiler has determined that these stalls are necessary according to the latency of the instructions involved.

Compute Capabilities 5.x and 6.x: The "Maxwell" and "Pascal" generations, which include devices with Compute Capabilities 5.0, 5.2, 5.3, 6.0, 6.1, and 6.2, use a different ISA than the previous generations of NVIDIA devices, with the opcode contained in bits 52-63.

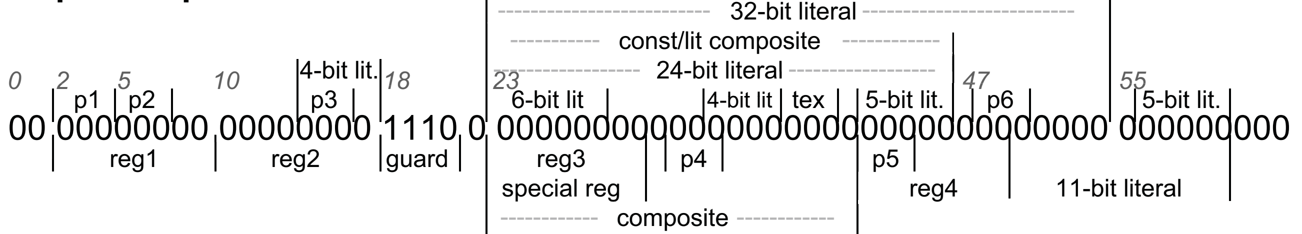
In the previous generation, the SCHI values contained only dispatch intervals and one or two rarely-used flags. But in

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

Compute Capabilities 2.x and 3.0



Compute Capabilities 3.x



Compute Capabilities 5.x and 6.x

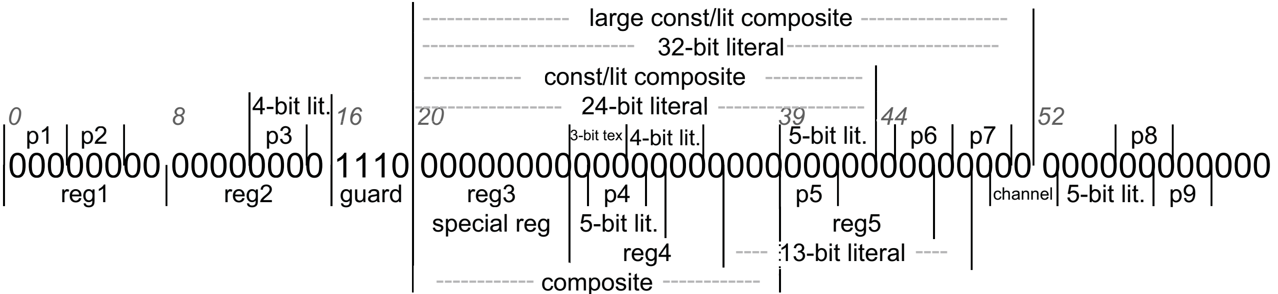


Fig. 8. Common operands on different CUDA architectures.

(a)

0x08a088808c8c10bc	
MOV R1, c[0x0][0x44];	0x2f
S2R R0, SR_CTAID.X;	0x04
ISUB R1, R1, 0x30;	0x23
S2R R3, SR_CTAID.Y;	0x23
S2R R4, SR_TID.Y;	0x20
IMUL R2, R0, c[0x0][0x28];	0x22
S2R R5, SR_TID.X;	0x28

(b) 0x08a088808c8c10bc

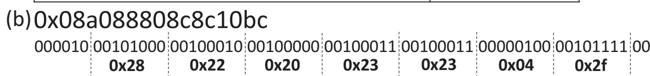


Fig. 9. An example of how we extract the scheduling information for each group of seven instructions on Kepler GPUs. These 8-bit values indicate dispatch behavior.

(a)

0x007f9800e7a007f3	
SSY 0x6c8;	1, 0x3, 7, 7, 000000
S2R R0, SR_TID.X;	1, 0xd, 1, 7, 000000
XMAD R3, R2, 0xa0, R0;	0, 0x6, 7, 7, 110000

(b) 0x007f9800e7a007f3

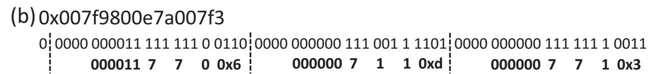


Fig. 10. An example of how we extract the scheduling information for each group of three instructions on Maxwell and Pascal GPUs. The first and second values indicates dispatch behavior. The third and fourth indicate which barriers to set. The fifth indicates which barriers to wait for.

Maxwell and Pascal, they have been extended to control barriers and the register cache. Every fourth instruction on these architectures is a SCHI, so they each affect the following three real instructions. Different from previous ISAs, the binary for a SCHI instruction no longer has any bits dedicated to its opcode, and is therefore identified purely by its position in the code.

As with the previous generation, NVIDIA's dissembler outputs each instruction group's SCHI data only as a single,

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

combined hexadecimal value. As such, we again break up the SCHI data, in-lining values with individual instructions so that the code can be easily understood and edited. Fig. 10 shows how we extract the separate values, and how they line up with the following three instructions. The 21 SCHI bits associated with each instruction are distributed as follows. Bits 0 through 3 are the minimum number of cycles to wait before dispatching the next instruction. According to [3], bit 4 is a "yield hint flag", which encourages the GPU to switch to another thread. Higher dispatch values in the lowest bits will fail without the yield hint. Bits 5 through 7 and 8 through 11 each indicate a barrier to set after the instruction, or have value 7 when no barrier needs to be set. Bits 12 through 17 indicate which combination of the six barriers the thread must wait for before dispatching the associated instruction. We skip the remaining four bits, which the disassembler already in-lines as cache "reuse" modifiers attached to register operands.

Although this architecture allows instructions to set two barriers, trying to set an unexpected barrier can cause erroneous behavior. The first barrier slot is used to handle true dependencies, for variable-latency instructions that have a destination register (e.g. loads); the second barrier slot is used to handle anti-dependencies, for variable-latency instructions that have source registers (e.g. stores). If an instruction depends on a barrier, the thread will wait for every instruction that set that barrier.

In the case of Fig. 10, the thread must wait at least 3 cycles after the first instruction. Upon dispatching the second instruction, it sets write barrier #1, and then must wait at least 13 cycles. Since the lowest two bits are set for the barrier wait bit-field, the thread waits for both barrier #0 and barrier #1 to be released before dispatching the third instruction, and after dispatch it waits at least 6 cycles.

Compute Capability 7.x: The "Volta" generation, which includes Compute Capabilities 7.0 and 7.2, has another new ISA. In this generation, instructions have been expanded to 16 bytes in size, with scheduling information embedded into each individual instruction. We have not completely decoded this ISA yet, but it is in progress, and can be decoded with similar methods to the previous architectures.

V. Applications

There have been several works which interact with the binary-level GPU code, but due to the closed-source nature of NVIDIA's ISAs, they have been forced to make use of third-party projects. Our work is designed to support as many of these ISAs as possible, and has already been used by authors of [7], [10], and [12].

Our framework uses Flex [22] and Bison [23] to parse the assembly code, generating an intermediate representation (IR) in which the precise version of the ISA is largely irrelevant. Applications can directly make use of our framework as a front-end and back-end, operating on this IR. For example, if an application needs to convert local memory operations to shared memory operations, it can simply scan the IR for local memory instructions, change each one's memory type,

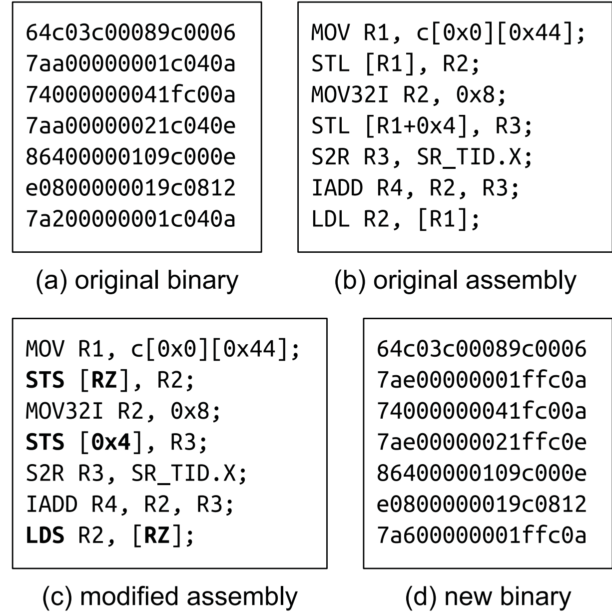


Fig. 11. Example of converting local memory instructions to shared memory instructions.

and change their memory addresses as necessary, as shown in Fig. 11.

In Fig. 11, (a) shows Compute Capability 3.x binary code in hexadecimal format, (b) shows the corresponding assembly code extracted with our front-end, (c) shows the result of modifying the memory instructions, and finally (d) shows the new binary generated by our assembler. Our framework already performs each of these steps except the transformation from (b) to (c).

Our framework can serve as an architecture-independent infrastructure for various applications described below. Unlike existing assemblers, it can provide them compatibility with multiple generations of GPUs. Our framework is available at https://github.com/decodecudabinary/Decoding-CUDA-Binary.

Compilation CUDA developers are currently reliant on NVIDIA's closed-source software for compilation. With access to the instruction encoding, however, the development of an open-source GPU compiler is possible. This would allow researchers to easily explore various compiler optimizations, such as new algorithms for GPU register allocation.

IR Generation Our framework enables versatile control flow analysis by leveraging our intermediate representation. When we parse the assembly into its IR, we organize the instructions into basic blocks. We convert branch targets from literal offsets to pointers, and break up instruction-scheduling values into their associated instructions. This organization of the code results in human-readable assembly, allowing developers to better understand a program, and facilitates techniques such as binary instrumentation.

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

```
CAL 0xa8;
DSETP.LT.AND P0, PT, R10, R12, PT;
@P0 BRA 0x80;
EXIT;
ISETP.LE.U32.AND P0, PT, R14, R0, PT;
@!P0 BRA 0xe8;
```

(a) original assembly

```
label2:
      CAL label5;
label3:
      DSETP.LT.AND P0, PT, R10, R12, PT;
  @P0 BRA label2;
label4:
      EXIT;
label5:
      ISETP.LE.U32.AND P0, PT, R14, R0, PT;
 @!P0 BRA label7;
```

(b) human-readable assembly

```
label2:
      CAL label5;
label3:
      DSETP.LT.AND P0, PT, R10, R12, PT;
  @P0 BRA label2;
label4:
      MOV R10, RZ;
      MOV R12, RZ;
      EXIT;
label5:
      ISETP.LE.U32.AND P0, PT, R14, R0, PT;
 @!P0 BRA label7;
```

(c) instrumented assembly

Fig. 12. Example of instrumenting the code to clear some registers before exit.

Binary Instrumentation Once we organize the assembly into basic blocks, code can easily be inserted or deleted, with scheduling data placed automatically. In [10] our framework was used to perform GPU binary instrumentation, in order to protect otherwise vulnerable regions of memory. In Fig. 12, we show a simple example: (a) is a snippet of raw assembly code, (b) is human-readable assembly generated by our framework, and (c) is modified assembly that has been instrumented to clear some registers before exiting the kernel.

Binary instrumentation can be done even without access to a program's source code. Furthermore, since our IR is not tied to a single version of the ISA, changes to the code can be compatible with many architectures, using our generated assemblers to target different devices as needed.

VI. Related Work

There are several works which interact with the binary code, but are limited to specific architectures, and so could benefit from our framework in order to support additional devices. In GPES [5], the authors transform the binary to enable workload partitioning to be performed in a manner totally transparent to applications - though they state that more sophisticated binary analysis is still needed to support complex kernels. [8] proposes a synchronization scheme which uses binary-level instructions to achieve greater efficiency than other lock-based approaches. In [9], the KernelGen framework employs several instructions that exist only in the binary ISA, modifying the code and other parts of the ELF.

In [17], the Fermi generation (Compute Capability 2.x) was examined. In [4], the late Kepler generation (Compute Capability 3.x) was explored to optimize the SGEMM application. [3] reverse-engineered Maxas (Compute Capability 5.x), and [18] looked at Volta (Compute Capability 7.x). But to the best of our knowledge, ours is the first work that strives to provide assembler generating framework in order to support as many NVIDIA architectures as possible.

VII. Conclusion

The closed-source nature of NVIDIA's ISA restricts the capabilities of developers and researchers. Sufficient knowledge of the ISA enables better optimization and security, as well as allowing various applications including register allocation and binary instrumentation. The assembler generation technique we developed permits rapid decoding of the machine code for various GPU generations. Furthermore, the framework we built makes these assemblers easy to use for architecture-independent applications. Authors of several works [6] [7] [10] [12] have utilized our framework.

Appendix A
Artifact Appendix

A. Abstract

Our artifact includes a copy of our Assembler Generator, which takes CUDA benchmarks (executables) as input, and creates CUDA SASS assemblers (written in C++) as output. Our software and inputs expect a Linux environment, and for compilation they require GNU C/C++, Flex and Bison, and NVIDIA's CUDA Toolkit. There are no explicit hardware requirements.

This appendix contains detailed information on our artifact and its usage, including links for obtaining additional inputs beyond the selection of benchmarks included in our artifact. Our artifact itself is freely available at Zenodo [24].

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

B. Artifact Check-List (Meta-Information)

- **Algorithm:** CUDA SASS Assembler Generation
- **Compilation:** Requirements: GNU C/C++; Flex and Bison; NVIDIA CUDA Toolkit
- **Data set:** Rodinia Benchmark Suite and CUDA SDK Code Samples
- **Run-time environment:** Linux; we used openSUSE and Ubuntu in our experimentation.
- **Output:** C++ code for assembling CUDA SASS assembly into binary
- **How much disk space required (approximately)?:** 25 Megabytes, or more with additional benchmarks/data.
- **How much time is needed to complete experiments (approximately)?:** Seconds or minutes, depending on the benchmarks included in the data set.
- **Publicly available?:** Yes; complete code will be made available at a later date.

C. Description

1) How Delivered: A functional copy of our Assembler Generator for this artifact is freely available at Zenodo [24].

2) Hardware Dependencies: There are no hardware dependencies.

3) Software Dependencies: For full functionality, our tools require a Linux operating system; we have tested them on openSUSE and Ubuntu.

Software requirements include C/C++, Flex and Bison, and NVIDIA's CUDA Toolkit (https://developer.nvidia.com/cuda-downloads). We used version 6.5 of the CUDA Toolkit when preparing this artifact, however, we believe any version between v5.0 and the latest release (v10.0) should work.

4) Data Sets: Our data sets consist primarily of the Rodinia Benchmark Suite - http://lava.cs.virginia.edu/Rodinia/download_links.htm - and the CUDA SDK Code Samples - which can be installed as part of the CUDA Toolkit (https://developer.nvidia.com/cuda-downloads). Our artifact includes a small subset of these benchmarks, and a Bash script to compile them.

In our own experiments, we used all of the following benchmarks from Rodinia and the CUDA SDK: backprop, bfs, bicubicTexture, b+tree, cfd, dct8x8, dxtc, FDTD3d, gaussian, heartwall, hotspot, imageDenoising, interval, kmeans, lavaMD, leukocyte, lud, matrixMul, MC_SingleAsianOptionP, mummergpu, myocyte, nbody, nn, nw, particlefilter, particles, pathfinder, RAY, recursiveGaussian, srad, streamcluster.

The artifact copy of our Assembler Generator expects benchmarks to target Compute Capability 3.x, 5.x, and/or 6.x devices only.

D. Installation

Our Assembler Generator and each of the benchmarks is accompanied by a Makefile. To build each of them, run make inside each of their directories. For the provided subset of benchmarks, we include a Bash script that will compile all of them and place their executable into a single directory.

E. Experiment Workflow

GPU kernel functions are extracted from compiled benchmarks and fed into the Assembler Generator one-at-a-time for analysis; persistent data from analysis is passed through standard in and standard out. After processing each of the kernel functions, the Assembler Generator is used to perform one or more rounds of bit flipping in order to improve its results: bit-flipped binary code is written into an executable, and then re-extracted with its assembly and analyzed. Finally, the Assembler Generator is invoked once more to prepare CUDA SASS assemblers (written in C++). These new assemblers can be tested on one or more of the benchmarks to confirm it produces the same code as the original.

In other words, the basic experiment workflow steps are as follows: prepare benchmarks, extract kernel functions, analyze kernel functions, generate bit-flipped code, inject bit-flipped code into executable, extract bit-flipped kernel function, analyze bit-flipped kernel function, generate assembler code, assemble code into benchmarks, verify that benchmarks have not changed.

F. Evaluation and Expected Result

We provide a Bash script, with filename procExes.sh, which performs all of the above Experimental Workflow steps except for preparing the benchmarks (the CUDA executables used as input). Before running the script, all input executables should be placed in the **exes** subdirectory.

The script places generated assemblers' code into files named generatedAssemblerXX.txt, where XX is the version number of the target architecture. If a Compute Capability 3.5 assembler is generated, it is placed into the source code for our included **asm2bin** tool, and tested on each Compute Capability 3.5 benchmark to confirm its correctness.

G. Experiment Customization

Additional benchmarks can be used; simply place the desired CUDA executable inside the **exes** sub-directory before running our procExes.sh script.

The target architecture for the provided subset of benchmarks can be changed by modifying their Makefile. For example, to use Compute Capability 5.0 code instead of 3.5, change the -arch=sm_35 flag to -arch=sm_50.

H. Notes

When injecting assembled or bit-flipped binary code into an executable, our program modifies the GPU ELF according to the specifications we have made available on Zenodo [15].

When generating bit-flipped code, our Assembler Generator outputs it as a list of BINCODE instructions. BINCODE is not a real opcode used by NVIDIA, but rather a phony opcode we use in our own tools to indicate that the instruction contains only binary code.

A more complete and well-documented copy of our Assembler Generator and related tools is available at https://github.com/decodecudabinary/Decoding-CUDA-Binary.

Acknowledgment

The authors would like to thank the anonymous reviewers who provided constructive criticism for this work. This work

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.

is supported by grants NSF-CCF-1421505 and NSF-CCF-1628401. Any opinions, findings, conclusions, or recommendations in this material are those of the authors, and do not necessarily reflect the views of this work's sponsors.

References

[1] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–10.

[2] A. Lavin, "maxdnn: an efficient convolution kernel for deep learning with maxwell gpus," *arXiv preprint arXiv:1501.06633*, 2015.

[3] S. Gray, "Maxas: Assembler for nvidia maxwell architecture," 2014.

[4] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," in *ACM SIGPLAN Notices*, vol. 52, no. 8. ACM, 2017, pp. 31–43.

[5] H. Zhou, G. Tong, and C. Liu, "Gpes: A preemptive execution system for gpgpu computing," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*. IEEE, 2015, pp. 87–97.

[6] A. B. Hayes and E. Z. Zhang, "Unified on-chip memory allocation for simt architecture," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 293–302.

[7] A. B. Hayes, L. Li, D. Chavarría-Miranda, S. L. Song, and E. Z. Zhang, "Orion: A framework for gpu occupancy tuning," in *Proceedings of the 17th International Middleware Conference*. ACM, 2016, p. 18.

[8] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 109–118.

[9] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström, "Kernelgen–the design and implementation of a next generation compiler platform for accelerating numerical models on gpus," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1011–1020.

[10] A. B. Hayes, L. Li, M. Hedayati, J. He, E. Z. Zhang, and K. Shen, "Gpu taint tracking," in *USENIX ATC*, 2017, pp. 209–220.

[11] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[12] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. Chavarría-Miranda, and H. Corporaal, "Critical points based register-concurrency autotuning for gpus," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1273–1278.

[13] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, "Decoding cuda binary - opcodes," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2339154

[14] ——, "Decoding cuda binary - decoded instructions," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2339116

[15] ——, "Decoding cuda binary - file format," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2339027

[16] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[17] Y. Hou, J. Lai, and D. Mikushin, "Asfermi: An assembler for the nvidia fermi instruction set," *URL: http://code.google.com/p/asfermi/(accessed: 03.12. 2012)*, 2011.

[18] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[19] NVIDIA, "GPU computing sdk." [Online]. Available: https://developer.nvidia.com/gpu-computing-sdk

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[21] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, "Decoding cuda binary - special registers," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2339176

[22] V. Paxson *et al.*, "Flex–fast lexical analyzer generator," *Lawrence Berkeley Laboratory*, 1995.

[23] C. Donnelly and R. Stallman, "Bison. the yacc-compatible parser generator," 2000.

[24] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, "Decoding cuda binary artifact," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2337060

Authorized licensed use limited to: Rutgers University Libraries. Downloaded on December 11,2023 at 21:51:31 UTC from IEEE Xplore. Restrictions apply.