Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Towards Calculating HPC CUDA Kernel Performance on Nvidia GPUs

Master Thesis

Dumeni Manatschal

July 25, 2025

Advisors: Philipp Schaad, Yakup Budanaz, Prof. Dr. Torsten Hoefler

Department of Computer Science, ETH Zürich

Abstract

This thesis aims at providing the ground work to facilitate a performance estimation model for CUDA kernels using a cycle counting model. After a short overview of past GPU performance modeling techniques, it conducts an exhaustive, in-depth analysis of Nvidia's SASS instruction set and CUDA ELF formats for architectures Maxwell up to and including Blackwell, facilitating deep insight into Nvidia's SASS instruction format, enabling precise microbenchmarking based on SASS instructions only, while utilizing *Python* as a tool. Finally, in addition to a VSCode extension featuring a precise, in-depth visualization to a precise, custom CUDA kernel disassembler, it provides insights into Nvidia's SASS instruction scheduling and barrier mechanisms and a series of tutorials, jumpstarting understanding of SASS and a concrete proposal for a Cycle Counting Model using data that can be provided by the techniques presented in this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

Optimizing Cuda kernels for specific GPU models can be labour intensive. GPU hardware configurations aren't as standardized as is common on CPUs. Here are a few examples [5]:

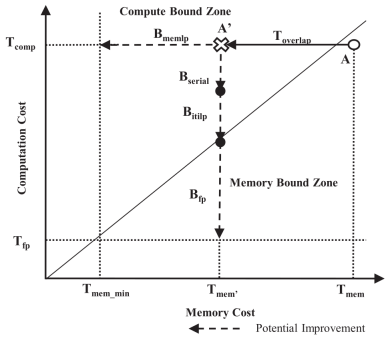
- Until Pascal architecture, shared memory and L1 cache are separated. From Volta onwards, they share the same memory and it is even possible to use more memory as shared memory or viceversa more memory as L1 cache, depending on the programmers need.
- Volta features one L2 cache around 6 graphics processing clusters with 14 streaming multiprocessors each, Ampere contains two separate L2 caches, linked by a bus where each L2 cache directly services 4 graphics processing clusters, featuring 16 streaming multiprocessors each. Next to the obviously improved processing capacity, the issue arises, how to deal with the split cache during SM to SM communication.
- On Pascal, Int32 and Float32 instructions could not be concurrently executed on the same Cuda core while on Volta these two types of instructions run fully concurrent.
- On Ampere it is possible to asynchronously load data into the cache from global memory while on previous architectures, this wasn't possible.

The question arises, if it is possible to build a tool that can both approximate how well a kernel will run on a real GPU while simultaneously providing hints to the programmer where potential bottlenecks will occur.

1.2 Modeling a GPU and Related Work

This task has been performed by many people in the past. Inspiration for this work is drawn from [1], [2], [3], [4] and [5].

In [1] the authors build a fully analytical profiler, based on their own theoretical model, described in [6]. They even construct a fully analytical model that predicts power consumption for a kernel [7]. [6] introduced the concept of *memory* and *compute parallelism* and showed that in the ideal case, all compute operations on a GPU are hidden by the memory accesses. Commonly it is referred to as the *MWP-CWP model* (memory-warp-parallelism-compute-warp-parallelism). All modeling is based on a very involved set of equations, a few heuristic parameters and simply counting PTX instructions and dividing them in *memory accessing* and *compute only* categories. The kernel itself is split into *compute* and *memory warps* that are fed into an idealized model of a pipeline. The analytical GPU model in [6] was published in 2009, the power consumption predictor in 2010 [7] and the actual profiling tool in 2012 [1].



Name	Description	Unit
T_{exec}	Final predicted execution time	cost
T_{comp}	Computation cost	cost
T_{mem}	Memory cost	cost
T_{overlap}	Overlapped cost due to multi-threading	cost
T'_{mem}	$T_{\text{mem}} - T_{\text{overlap}}$	cost
T_{fp}	Ideal T_{comp}	ideal cost
$T_{\text{mem\_min}}$	Ideal T_{mem}	ideal cost
B_{serial}	Benefits of removing serialization effects	benefit
B_{itilp}	Benefits of increasing inter-thread ILP	benefit
B_{memlp}	Benefits of increasing MLP	benefit
B_{fp}	Benefits of improving computing efficiency	benefit

Figure 1.1: The MWP-CWP [1] model introduces memory and compute parallelism. Every kernel has a portion between T_{mem_min} and T_{mem} memory access time and another portion between T_{fp} and T_{comp} compute time. These two times are mapped on the shown 2D canvas in the left figure. A not optimized kernel may start at point **A** on the top right edge. Memory access optimizations move the point to the left and compute optimizations move the point downwards. The optimal performance point for **A** to reach is (T_{fp}, T_{mem_min}). The canvas is split into two triangles by the diagonal. If **A** ends up in the upper left triangle, the kernel is *compute bound*, if it ends up in the lower right triangle, it is *memory bound*. The right hand figure names the variables.

The most relevant part of [1] is shown in Figure 1.1. The graph's upper left triangle in Figure 1.1 is the *compute bound zone* and the lower right triangle the *memory bound zone*. T_{comp} represents the total compute work and T_{mem} the total memory access work. At point **A** on the top right of Figure 1.1, is the starting point for an optimization. By applying different optimization techniques, the kernel performance can be moved along the diagonal of the plot to the (T_{fp},T_{mem_min}) location in the lower left that represents the optimum performance balance. If the point cannot be exactly reached, the

kernel will end up in either the *compute* or *memory bound* regions.

Another model is introduced in 2009 [4] where the authors propose converting a CUDA kernel's PTX [8] instructions into a *work-flow-graph*, taking the shape of a DAG. The nodes represent instructions and the graph features two sets of edges: control and dependencies. The edge weights represent the required cycles to transition from one node to the next using said edge. After constructing the graph, the authors apply a set of very complex graph reduction transformations based on the underlying assumption that the graph runs on an idealized pipeline. The reduction operations are repeated until the graph is reduced to one single *source* and one *sink* node connected by one single *edge*. The weight of the resulting edge then corresponds to the total estimated cycle count for the kernel. Figures 1.2 and **??** show two graphs used in [4] including their original descriptions. Both show the *starting point* at the top and the final state of the transformations at the bottom. Note that the graph transformations are highly non-trivial and based only on equations. The graph representation intrinsically includes *instruction level parallelism* while [1] explicitly calculates this aspect using an equation. The model highlighted in Figure 1.2 is commonly referred to as *WFG* [4] model.

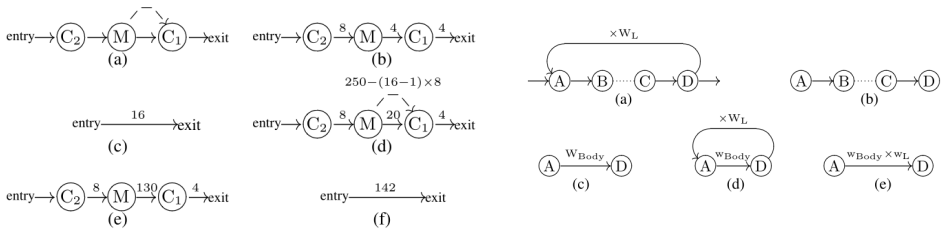


Figure 1.2: The WFG [4] model converts PTX [8] into a *work-flow-graph* that takes the form of a directed, acyclic graph. The nodes represent instructions and the edges are control flow and data dependencies. The edge weights represent the require cycles to transition from one instruction to the next. The graph is reduced using a complex set of reduction transformations until only two nodes connected by one edge are left: the source and the sink and the weight of the connecting edge is the total number of cycles that the kernel requires.

A much newer model, published in 2023 is proposed in [2]. Their first *contribution* is a *summary* of five older, analytical models, including the two models (*MWP-CWP* [1], *WFG* [4]) mentioned beforehand. In fact, it shows that all five models can (with a bit of tweaking) be mapped to a *pipeline model* with some assumptions of ideality. For example, while [1] does assume that the pipeline is partially idle during filling time and [4] additionally offers a compelling view on instruction level parallelism, both assume that there is always a ready instruction to be scheduled and especially, that an *ideal* scheduler exists. With that, both omit pipeline stalls. [2] attempts to address this shortcoming by using the *work-flow-graph* representation used in [4] and directly running it on a pipeline model. With that approach [9] shows that

stall cycles can indeed play a role as Figure 1.3 from [2] shows.

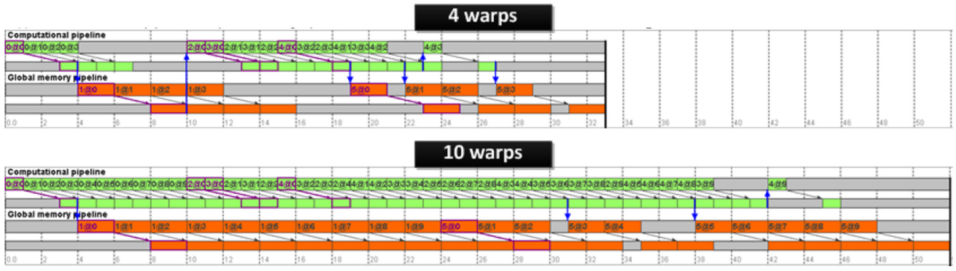


Figure 1.3: The Pipeline Model [2] uses similar graphs to the WFG [4] model and runs them using a pipeline. As this figure show, *pipeline stalls* can play a role in the final simulation result.

The authors also conclude that the *scheduler* can play a defining role in the performance of a kernel. Figure 1.4 from [2] shows that simpler models are close to a *roofline* plot shape. *MWP-CWP* [1] and *WFG* [4] do show non-ideal characteristics with a monotonously increasing curve. The pipeline models [2], featuring two versions with two different schedulers *Pipeline Model (RR)* (*Round Robin*) and *Pipeline Model (GTO)* (*Gready Then Oldest first*), show non-monotonic non-idealities that the authors contribute specifically to their explicit modeling of a pipeline.

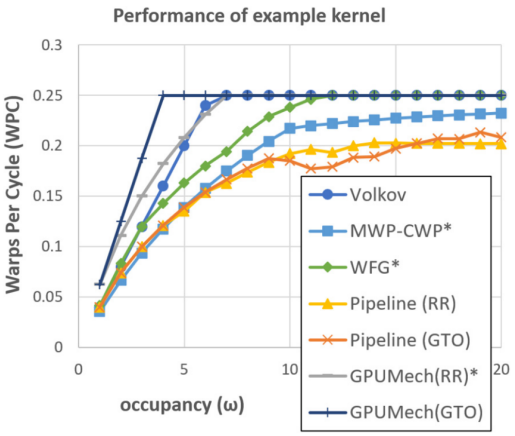


Figure 1.4: The Pipeline Model [2] shows that simpler models produce outputs similar to a *roofline plot*. It also shows that simulating kernels on a more realistic pipeline simulation produces more nuanced outputs that by using equations only, by also simulating for example stall cycles during the simulation, as evident by the non monotonic progress of the yellow and orange outputs of the Pipeline Model [2].

[2] utilizes two different schedulers (RR and GTO). [3] showcases a machine learning method based on a support vector machine to predict the perfor-

mance of a kernel. While the performance model itself is not too interesting, the authors also showcase a set of compelling benchmarkings to characterize the scheduler of a GPU by microbenchmarking kernels that contain only *compute instructions* and kernels that include *memory instructions*. Their analysis shows that

1. while no warp is blocking, the GPU utilizes Round Robin
2. if there is resource contention or memory wait, finish warps from the current thread blocks with lower ID first and then schedule warps from available thread blocks with the lowest ID first (this is GTO scheduling strategy).

As time progressed from 2009 onwards, performance models have become increasingly precise, attempting cycle-correct predictions. GCoM [9] features an elaborate memory model. For example it explains sectored L1 caches. However, *simulations* require hours-long measurements of the GPU hardware before it can start computations to predict a cycle count for a kernel. Computing predictions themselves using GCoM also can take hours up to days. Models such as GCoM are aimed at to be tools for GPU architecture-space exploration rather than to just to predict CUDA kernel performance. Thus, the compute time is justified.

While older performance models rely on counting and categorizing PTX instructions for their predictions, models that attempt cycle-correct predictions, such as GCoM [9] move one level down to SASS [10].

While PTX [8] is platform independent, SASS [10] is the assembly instruction set running directly on an Nvidia GPU's hardware and is platform dependent but almost completely undocumented. The Nvidia compiler NVCC produces SASS instructions out of PTX. There is no direct match between instruction namings, though, since NVCC utilizes hardware specific operantions available only on some architectures but not on others. If looking into this topic, one can observe many statements in forums where users ask how they can produce specific SASS instructions out of PTX.

There exist different SASS dialects, matching different compute capabilities and architectures [8]. Maxwell and Pascal both utilize SM 50 to 52 while Volta supports SM 70 and 72. Turing uses SM 75, Ampere and Ada both utilize SM 80 and 86. The new Hopper architecture uses SM 90 and is superseded by Blackwell that introduces SM 100 and 120. [11] provides insight into SM 50 to SM 62 outlining various mechanisms using *bit-flipping* techniques. [12] provides various insights into the inner workings of several GPU architectures. [13] provides a way to extract the full SASS definitions for SM 50 up to 120 as text files from Nvidia's *cuobjdump* tool.

1.3 Contributions

This thesis aims at providing the ground work to facilitate a performance estimation model for CUDA kernels that is based on evaluating SASS instructions directly, extracting as much information as possible from Nvidia's instruction set, such that it becomes possible to gain architecture specific insights.

- In Chapter 1 we provide a short comprehensive overview of past modeling attempts, inspiring our proposed model in Chapter 7.
- In Chapter 2 we formalize SASS by fully evaluating [13], producing a full *Python* abstraction for SASS instructions.
- In Chapters 2, 3 and 5 we provide various *Python* modules to effectively decode and encode SASS instructions utilizing NVCC produced, and custom CUDA kernels.
- With Chapter 4 we provide a way to efficiently interact with the various SASS instruction sets by providing a visual tool as VSCode extension to show maximal information about each decoded SASS instruction.
- Chapter 6 offers a comprehensive series of tutorials aimed at jumpstarting familiarization with SASS, including providing insight into Nvidia's SASS software scheduler bits as well the instruction barrier mechanisms.
- With Chapter 7 we introduce microbenchmarking utilizing SASS instructions directly in a convenient way using *Python* as a tool as well as multiple strategies to do so effectively, starting with simple templates to strategies that calculate and bulk-generate benchmarking CUDA kernels numbering into the thousands utilizing a SASS instruction generator.
- Finally Chapter 7 proposes a cycle counting model befitting the outputs that can be achieved utilizing our *Python* SASS framework.

Chapter 2

SASS

This chapter introduces Nvidia's **SASS** hardware level instruction set and outlines the format of SASS instructions. It includes a full overview as well as details about the predicate, opcode, operands, extensions, and the cache bits.

2.1 Nvidia SASS Demystified

This section introduces **SASS** and contains background information and highlights a couple of corner points with respect to the various Nvidia architectures.

SASS [10] is Nvidia's lowest level instruction language. According to StackOverflow [14] it stands for **StreamingASSembly**. Every new architecture comes with it is own flavor of SASS. **CUDA** is compiled to **PTX** [8] and **PTX** is transformed into **SASS**. CUDA and PTX are well and thoroughly documented, while SASS only features a list of the instructions that exist and a rough outline of their format [10]. The listings are not complete, and the outline of the format does not even come close to being useful. It is meant to be used in conjunction with `cuobjdump -sass`. Thus, [10] is also more about tools on how to disassemble a CUDA binary than SASS instructions.

Instruction classes are described in detail in Section 2.4. At this point, the reader should understand the term **instruction class** as **one specific variant of an instruction**. For example, the instruction DFMA in SM 86 has 9 distinct instruction classes, while on SM 120 there are only 5.

Generally, there is a high degree of similarity between similar architectures' SASS flavors. For example, the two Ampere architectures SM 80 and SM 86 are similar, as well as the Volta architectures SM 70 and SM 72. Until the introduction of Blackwell architecture, instruction classes just seem to be

added while at least some version of older features are still included in the newer architecture.

Major architecture upgrades can also introduce rather extensive updates in the extents of their SASS flavor. All SASS flavors for SM 50 up to SM 62 cover the Maxwell and Pascal architectures and tend to be rather compact, featuring around 300 distinct instruction classes. Volta architecture with SM 70 and 72 is still around that level of complexity with around 320 distinct instruction classes, but introduces a completely new encoding mechanism.

Up to SM 62, all instructions are encoded in quadruples of 64 bit words where the first three words belong to three distinct instructions and the last word is shared, where each instruction gets 21 bits each with varying offsets. The total instruction length is 88 bits where the top 3 bits are unused.

Starting with Volta and SM 70, each instruction is encoded in two 64 bit words, totaling 128 bits each.

Turing architecture, which is SM 75 introduces major new concepts, for example *UniformRegisters* and a notable increase up to 580 instruction classes. Ampere (SM 80 and 86) increase that number to 616 and 643 respectively and also introduce memory descriptors for data transfer.

Hopper architecture with SM 90 is the uncomfortable max with 728 instruction classes. After that, Blackwell (SM 100 and SM 120) can boast only up to 578 distinct instruction classes. It looks like some serious cleaning house action by Nvidia.

Architectures from SM 70 upwards, with the exception of SM 75 and 90 come in pairs. The lower SM number refers to the HPC variant and the higher SM number refers to the consumer GPU for graphics rendering. The HPC variant is always less complex than the consumer-grade one. The consumer GPUs contain more instructions related to graphics rendering. The difference is most notable with Blackwell, where SM 120 seems to be much more complex than SM 100. This may be related to Nvidia introducing neural rendering techinques with *Blackwell*. The additional complexity is related not only to the number of instructions, but also to all other calculations that will be introduced in this chapter.

2.2 Format Introduction

This section provides a short introduction of the different components of a SASS instruction using the following four sample instructions as examples:

- **DFMA**, double precision fused multiply and add, Figure 2.2
- **ISETP**, integer compare and set predicate, Figure 2.3

- **MOV**, move value to register, Figure 2.4
- **STG**, store to global, Figure 2.5, 2.6

The next Section 2.5 expands on this section by providing **in depth** details on each topic.

Figure 2.1 shows the basic fields a *SASS* instruction contains. They are:

- Predicate: this is a prefix that the vast majority of instructions have. If the predicate is *true* (or PT as we shall see later), the instruction is executed, otherwise it will be skipped.
- **Opcode**: this is the **operation code**. It is what stands behind the instruction codes **DFMA**, **ISETP**, etc. Every instruction has one, but it is not necessarily unique.
- Dst Operand: this is the location where the instruction stores the result. Most instructions have a destination slot, albeit not all of them.
- Src Operands: these are the inputs to the instruction. As we will see later, they can take various forms.
- Extensions: extensions provide additional space for *configurations*. For example, an instruction may have a 32 bit and a 64 bit variant.
- $Cache: these are all the fields starting with $. They are related to how an instruction is executed by the instruction dispatcher and if they set or wait for barriers.
- Const: these contain bits that are *encoded* in the instruction with constant values that cannot be changed by any configuration.

All are decoded using the **Cubinext VSCode extension** that will be explained in detail in Chapter 4. This section serves as an introduction only and all concepts will be explained in *great detail* in the next Section 2.5.

Figure 2.1 shows an extension of the format that Nvidia usually outlines on their SASS website [10]. Figures 2.2, 2.3, 2.4, 2.5 and 2.6 show actual examples. Examining these examples, we note that most of the examples do not seem to conform to the template. However, as we shall see at the end of Section 2.5, the template is correct.

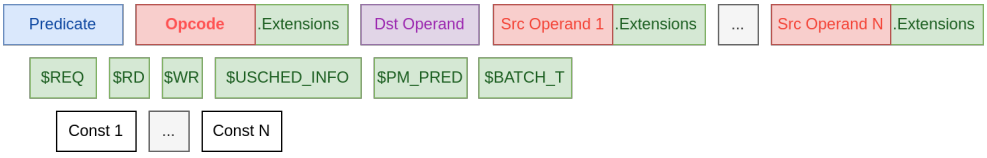


Figure 2.1: General format for a SASS instruction: Row 1: predicate, **instruction**, dst- and src-operands, configuration extensions, Row 2: $cache bits, Row 3: constant encoding values.

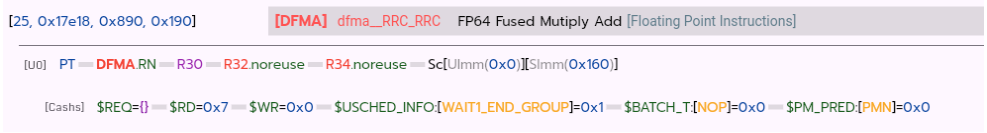


Figure 2.2: Example of one **DFMA** variant on SM 86, decoded using Cubinext VSCode extension from Chapter 4. Note the labels **[U0]** for row 1, **[Cashs]** for row 2.

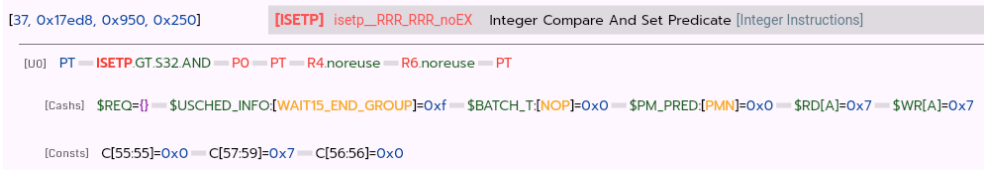


Figure 2.3: Example of one **ISETP** variant on SM 86, decoded using Cubinext VSCode extension. Note the labels **[U0]** for row 1, **[Cashs]** for row 2, **[Consts]** for row 3.

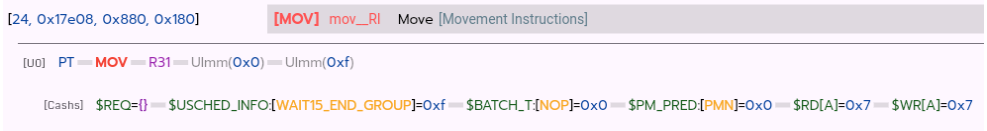


Figure 2.4: Example of one **MOV** variant on SM 86, decoded using Cubinext VSCode extension. Note the labels **[U0]** for row 1, **[Cashs]** for row 2.

ISETP in Figure 2.3 does not seem to have a purple destination register showcased in Figure 2.1. Neither does **STG** in Figures 2.5 and 2.6. **ISETP** uses a Predicate register as destination. Predicates are encoded with 3 bits. Sometimes the first operand is PT and seems to be nothing more than a space filler. Thus it is not possible to always figure out which operator is clearly the destination register. Nevertheless it is usually clear from context.

STG uses a *list* of registers and *immediate values* shown as SImm(0x0). This represents an address location.

In Figure 2.3, operand P0 is actually the destination. Instructions that use predicate registers as destination have this issue because sometimes a predicate PT is in the first position after **Opcode** as a 3 Bit placeholder.

The general division into three categories *instruction bits*, *cache bits* and *constants* applies.

Instruction Bits

The instruction bits are all parts related to what the instruction does. They include src, dst, **opcode** and configuration extensions.

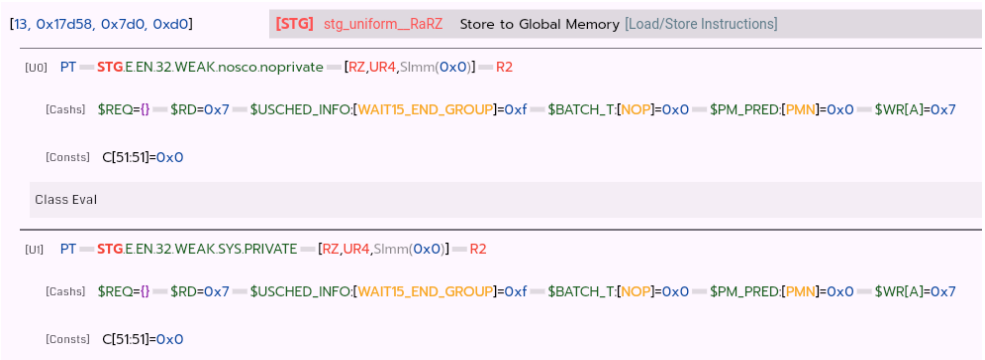


Figure 2.5: Example of one **STG** (store to global) variant on SM 86, decoded using Cubinext VSCode extension. Note the labels **[U0/1]** for row 1, **[Cashs]** for row 2, **[Consts]** for row 3. This **STG** variant is also available on SM 70 to SM 75,

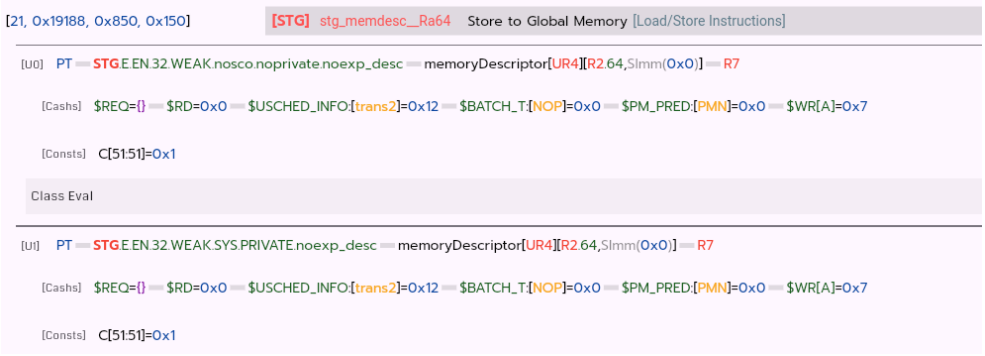


Figure 2.6: Example of another **STG** (store to global) variant on SM 86, decoded using Cubinext VSCode extension. Note the labels **[U0/1]** for row 1, **[Cashs]** for row 2, **[Consts]** for row 3. This variant of **STG** is only available on SM 80 and newer. Figures 3.3 and 3.4 show a detailed view for how *universes* [U0] and [U1] are decoded.

Cache Bits

The cache bits are represented in the second row of a decoded instruction as displayed in the sample Figures 2.2, 2.3, 2.4, 2.5 and 2.6.

The reason why these bits are called $cache is that they are all defined with a *dollar sign* $, linking the term to *cash*. They include software scheduler $USCHED_INFO bits, barriers $WR (write barrier) and $RD (read barrier), wait-for-barrier requests $REQ and two other fields, probably related to graphics rendering, called $BATCH_T and $PM_PRED. In the context of CUDA programs, the last two are always 0.

The examples in Figures 2.2, 2.3, 2.4, 2.5 and 2.6 also have parts in yellow and hex numbers in blue. These are decoded values. If they are blue, they are nameless, if they are yellow, they can be assigned to a register name. For example, the very common $USCHED_INFO[WAIT1_END_GROUP]=0x1.

In this case $USCHED_INFO is the name of the instruction field, 0x1 is the value and WAIT1_END_GROUP the mapped name.

Constants

Constant values are represented in the third row of a decoded instruction as illustrated in Figures 2.2, 2.3, 2.4, 2.5, and 2.6. They are optional and represent values that are encoded into the instruction with fixed values. Fixed values cannot be changed by any defined configuration or encoding scheme. Their key function lies in distinguishing instruction classes. How SASS instructions can be decoded is explained in great detail in Section 3.3. To a lesser extent, but also key, is that not all instructions can set barriers. For those instructions, the bits representing the $WR and $RD barriers are encoded to 0x7 using the fixed value scheme. A third reason for constant encoding values are *alternates* for instruction classes. Alternates are described in Section 2.4.2. They represent *specializations* of regular instruction classes. For example, **IMAD** (integer multiply and accumulate) is a multiplication combined with an addition. **IMUL** (integer multiply) is just a multiplication. **IMUL** is an *alternate* of **IMAD** where the addition operand is set to zero.

2.3 Resources Introduction

At this stage, the reader has seen the general structure of SASS instructions. Before presenting **detailed** semantics in the next Section 2.5, this section explains the meanings of the most common dst and src operand values, as well as other components of the instructions. This explanation provides the necessary background for understanding how SASS instructions are interpreted and executed by the hardware.

Everything in a SASS instruction is either a *register*, a *function*, an *opcode* or a *value*. The *registers* can be subdivided into operand registers, extension registers and $cache registers. The next few sections outline each of these in more detail.

2.3.1 Registers

There are mainly four types of registers.

- **regular registers** R1, ..., R255, RZ: each one of these represents 32 bits. Consecutive registers, for example R2 and R3 represent 64 bits that an instruction can access using R2, three consecutive registers represent 96 bits, etc.
- **uniform registers** UR1, ..., UR63, URZ: they are introduced with SM 75 and should be seen as *alternate data path* to the regular registers. Notably, in most cases they can be specified with a bit length. For

example, UR2.32 or UR2.64, though, it depends on the instruction they are used with. It can be tricky to combine *uniform* registers with *regular* registers. According to [12], uniform registers are part of a distinct data path, designed to still allow simple arithmetic operations while the regular registers are busy with more complex ones. Thus, simple instructions have separate versions for both types of registers, for example **MOV** and **UMOV** to move values into a *regular* and *uniform* register respectively as well as **IADD3** and **UIADD3** or **ISETP** and **UISETP** or **IMAD** and **UIMAD**.

- **predicate registers** P0, ..., P7, PT: these are used to store boolean values. PT contains *constant True*. Predicates can be set, for example, using **ISETP** (integer set and test predicate) to create *if* statements.
- **uniform predicate registers** UP0, ..., UP7, UPT: the same way that there exist *uniform registers* there also exist *uniform predicates* since SM 75. Instructions that use *uniform registers* usually also require *uniform predicates*. This serves to highlight the separate paths for *regular* and *uniform* registers again and there are two versions for instructions using *regular predicates* and *uniform predicates*, for example **ISETP** and **UISETP**.
- **extension registers** such as .GT.S32.AND in Figure 2.3: this is the largest *category* of registers. Almost every instruction can be configured using a large number of settings. For example, some instructions such as **STG** and **LDG** (store to global and load from global) can run as 32 bit and as 64 bit versions. These are akin to *PTX* [8] `.b8, .b16, .b32, .b64, .b128` used as `ld.global.b32 r2, array[r1];` to access a 32 bit value from global memory.

Specific values for registers are given using an `@` notation:

RegisterCategory@RegisterValue.

For example, the register R0 is fully specified as *RegisterCategory@R0*, the predicate P3 is fully specified as *Predicate@P0*.

All registers denoted as *RegisterCategory@RegisterValue* are associated with a value that is used to encode them in an instruction. Usually, the number of entries in the *RegisterValue* set determines how many bits are used for one particular *RegisterCategory*.

For example, the *RegisterCategory* Register (containing R0, R1, ...) contains 255 entries. Thus *Register@R2* is encoded with 8 bits, specifically with the 8 bit value 0x2. The category Predicate (with P0, P1, ...) contains 8 entries. Thus *Predicate@P3* is encoded with 3 bits.

Chapter 3 explains in detail how an instruction is encoded and decoded at the bit level. There we will see that often, the bit length associated with a

register or something else doesn't match the bit length it will be encoded in. Especially extension registers may be encoded in bit spaces that are longer than strictly required by the register.

2.3.2 Functions

The *Functions* are used to represent values in SASS instructions. Since the instructions need to run on hardware, they must define bit lengths and formats for everything. For example, a value 0x42 can be used as UImm(16) or SImm(16) where 16 denotes the bit length. There are also various formats for floating point numbers: F16Imm, F32Imm, F64Imm as well as a format BITSET used for bitmasks.

Generally, their definitions are not strictly documented, though, what they mean is clear from context. For example, an instruction using a 32 bit floating point *immediate value* would define it as `F32Imm(32)`. An instruction that uses a 16 bit offset to a base address would use `UImm(16)`.

2.3.3 Values

This type is used for values that cannot be mapped to a register value. For example $RD and $WR use values between 0x0 and0x7. In this case, the values enumerate *barriers* (see Section 2.5.16), they are displayed as hex values $WR=0x0 or $RD=0x7. Note that the values are encoded using functions with a bit length as well but do not intuitively align with the concept of *immediate values*.

2.3.4 Opcode

Opcode is short for operation code. For example **MOV** or **ISETP**. See Section 2.5.2 for a thorough explanation.

2.3.5 Operand Registers

This section outlines the registers that can be used in an instruction in the same way one would use the concept of registers on a CPU. This section expands on Section 2.3.1.

For example, we can use *Register@R34* and **MOV**(e) a value into it.

The **operand registers** include

- Register: R0, ..., R255
- Predicate: P0, ..., P7
- UniformRegister (SM 75 upwards): UR0, ..., UR63
- UniformPredicate (SM 75 upwards): UP0, ..., UP7

- SpecialRegister, SR0, ..., SR255
- BarrierRegister, B0, ..., B63
- Various special registers used for graphics, for example PARAMA_ARRAY_2D_ARRAY_1D_2D_1D. Note that these registers are normally used in positions of operands but work like configuration extensions. An example is given in Figure 2.7.

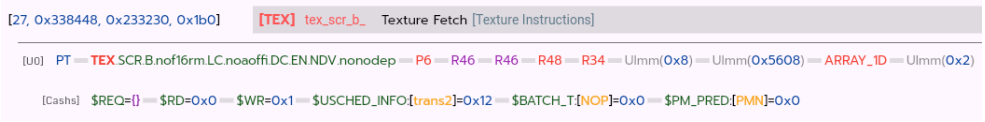


Figure 2.7: Example of graphics rendering **TEX** instruction including operand register PARAMA_ARRAY_2D_ARRAY_1D_2D_1D@ARRAY_1D special operand register.

In SASS, all of the these operand registers feature designations for subsets. Note that the notation follows *RegisterCategory@RegisterValue* where applicable.

1. Register@R255: RZ
2. NonZeroRegister: R0, ..., R244
3. UniformRegister@UR63: URZ
4. NonZeroUniformRegister: UR0, ..., UR62
5. Predicate@P7: PT
6. UniformPredicate@UP7: UPT
7. SpecialRegister@SR_CLOCK: C++ **clock64()** equivalent, many more special names are in this category
8. B3B0: B0, ..., B3

2.3.6 Extensions Registers

These are registers that are used to further configure an instruction. For example the instruction **STG** in Figure 2.5 has six extensions for the opcode **STG**.

STG.E.EN.32.WEAK.nosco.noprivate

and **1** extension for one of its operands, the register R2.64.

Note the similarity to the PTX instruction and configuration

st.weak.ss.cop.level::cache_hint.vec.type

and that documentation for a lot of the extensions can be found in *PTX* equivalents.

The easiest way to explain how the extensions work, is to introduce the first portion of the *format definition* in Listing 2.1, specifically for the instruction class stg_memdesc_Ra64 featured in Figure 2.5. The *format definition* will be explained in detail in Section **??**.

Listing 2.1: FORMAT definition for instruction class stg_uniform_RaRZ

```
CLASS "stg_uniform__RaRZ"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
      /E("noe"):e
      /COP("EN"):cop
      /SZ_U8_S8_U16_S16_32_64_128("32"):sz
      /SEM("WEAK"):sem
      /SCO("nosco"):sco
      /PRIVATE("noprivate"):private
   [ ZeroRegister("RZ"):Ra + UniformRegister:Ra_URc + SImm(24/0)*:Ra_offset ]
  ','Register:Rb
```

The notation starting with a "/" defines extensions to the instruction definition. Compare Listing 2.1 with Figure 2.5 and note how the format definition matches the real-world instruction. For example /SZ_U8_S8_U16_S16_32_64_128("32"):sz matches .32. All extensions for the **opcode** are listed in Listing 2.2.

Listing 2.2: All extensions for the opcode of the STG variant stg_uniform_RaRZ.

```
/E("noe"):e
/COP("EN"):cop
/SZ_U8_S8_U16_S16_32_64_128("32"):sz
/SEM("WEAK"):sem
/SCO("nosco"):sco
/PRIVATE("noprivate"):private
```

We take the the shortest one as explanatory example: */COP("EN"):cop*. *COP* is the name of the register while *EN* is one possible value. In the encoding Section 3.2 the **cop** will be defined as *alias* for the extension. At this stage, this is not important and is a preview only.

Now that we know how extensions are defined, we take the longest and most interesting one and show its full definition.

Listing 2.3: Full definition for the extension register SZ_U8_S8_U16_S16_32_64_128.

```
SZ_U8_S8_U16_S16_32_64_128 = {U8=0, S8=1, U16=2, S16=3, 32=4, 64=5, 128=6}
```

Since extension registers are now demystified, we can take a look at the **STG** instruction in Figure 2.5 again: the instruction is configured as follows:

STG.E.EN.32.WEAK.nosco.noprivate

Given the possibilities with SZ_U8_S8_U16_S16_32_64_128 we could also try the following configurations:

- STG.E.EN.**U8**.WEAK.nosco.noprivate
- STG.E.EN.**S8**.WEAK.nosco.noprivate
- STG.E.EN.**U16**.WEAK.nosco.noprivate
- STG.E.EN.**S16**.WEAK.nosco.noprivate
- STG.E.EN.**128**.WEAK.nosco.noprivate

Note that since instructions such as **STG** interact with a memory system, getting the sizes wrong is an excellent source for CUDA errors and registers that seem to contain garbage values.

2.3.7 Cache bits Registers

Recall the tree parts of an SASS instruction: the **opcode** and its operands, the $cache bits and the constants. Review for example Figure 2.4 for context. The names of the cache bit fields are also defined as registers.

While $REQ, $RD and $WR don't feature names for their values, $USCHED_INFO, $BATCH_T and $PM_PRED do.

$BATCH_T and $PM_PRED are always their equivalents of 0x0, which is NOP and PMN, $USCHED_INFO plays a significant role since it directly influences how instructions are dispatched. See Section 2.5.15 for details.

Listing 2.4: Definition for USCHED_INFO.

```
USCHED_INFO = {DRAIN=0,
               WAIT1_END_GROUP=1, ..., WAIT15_END_GROUP=15,
               trans1=17, ..., trans11=27}
```

2.4 Instruction Classes Introduction

Before more details are presented, this section explains the difference between the operation code (**Opcode**, for example, **DFMA**) and the concept of instruction classes. Note that a full example for an instruction class and all its components will be given in Section 2.4.3. The most expressive parts of two distinct instruction classes, both for the instruction **DFMA** are given in Listings 2.5 and 2.6.

While **Opcodes** are shared between up to 10 or more distinct instruction classes, the **name** of an instruction class is **always unique**.

Listing 2.5: Instruction class dfma__RRsI_RRI

```
CLASS "dfma__RRsI_RRI"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /Round1("RN"):rnd
      Register:Rd
      ',' [-] [||] Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ',' [-] [||] Register:Rb {/REUSE("noreuse"):reuse_src_b}
      ','F64Imm(64):Sc
```

Listing 2.6: Instruction class dfma__RUR_RUR

```
CLASS "dfma__RUR_RUR"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /Round1("RN"):rnd
      Register:Rd
      ',' [-] [||] Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ',' [-] [||] UniformRegister:URb
      ',' [-] [||] Register:Rc {/REUSE("noreuse"):reuse_src_c}
```

The reader should note the distinction in the format of the **operands**. As a rule on newer architectures, mainly SM 70 and above, instruction classes are organized in this fashion: there is usually a prefix that matches the **Opcode** of the instruction (in this case **DFMA**) and a suffix that indicates what shape the operands take. *RUR* would be *Register, UniformRegister, Register*. *RRI* corresponds to *Register, Register, Immediate value*. The reader should not assume, though, that there is any kind of enforced syntax in the names.

The following two sections explain the concepts of **main-** and **alternate instruction classes**. It is important to point out, that the text documents defining all of these classes are not too well structured, albeit they do follow a basic syntax with included variance. The concept of main and alternate classes is one of the things that is true in 99% of the cases. The direction of development is usually recognizable in the instruction class definitions as architectures progress. For example, SM 50's definition is a mess while starting with SM 70 one can recognize a distinct cleaning-up effort. But only starting with Blackwell architecture a newer architecture is less expansive than its predecessor (in this case Hopper).

2.4.1 Main Instruction Classes

As a rule, about half of all instruction classes are defined with the single keyword *"CLASS"*. These are **main** instruction classes and every one of them is uniquely distinguishable at the bits level (as will be outlined in the decoder Section 3.3) from all other main instruction classes.

2.4.2 Alternate Instruction Classes

The second half of the instruction classes are defined as *"ALTERNATE CLASS"*. These are all (mostly) specializations of a main instruction class. For example, the instruction **IMUL** ($a * b$) is an *alternate* of the instruction **IMAD** ($a * b + c$). **IMUL** multiplies its two operands while **IMAD** adds an additional value to the result of the multiplication. **IMUL** is **IMAD** with the third operand (c) set to RZ (zero) (see Figure 2.8).

Figure 2.8: This is a decoded **IMUL**, multiplying R6 with 0x8 and storing the result into R18 with the third operand set to zero (RZ).

If one just examines the bit encoding of alternate classes, they are indistinguishable from their **main** class parent.

Finally, the Listings 2.7 and 2.8 show their respective format definitions. **IMUL** looks the same as **IMAD**, just missing the last source operand Rc.

Listing 2.7: Main instruction class imad__RsIR_RIR, Note that the class has three src operands Ra, Sb and Rc.

```
CLASS "imad__RsIR_RIR"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
         /LOOnly("LO"):wide
         /FMT("S32"):fmt
      Register:Rd
      ','Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ','SImm(32)*:Sb
      ',' [-] Register:Rc {/REUSE("noreuse"):reuse_src_c}
```

Listing 2.8: Alternate instruction class imul__RsIR_RIR. Note that there is no src operand Rc while **IMAD** in Listing 2.7 has one.

```
ALTERNATE CLASS "imul__RsIR_RIR"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
         /LOOnly("LO"):wide
         /FMT("S32"):fmt
      Register:Rd
      ','Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ','SImm(32)*:Sb
```

To explain the *alternate* relationship, for the first time, we check out some of the *encoding definition* for these two classes in Listing 2.9. For **IMUL**, the last

operand is encoded as constant value 255, that corresponds to *Register@RZ*, meaning zero. This is the first explicit example of a **Constant**, which would be in the third row in the decoded format template in Figure 2.1.

Listing 2.9: Encodings for the operator Rc for instruction classes imad__RsIR_RIR (TOP) and imul__RsIR_RIR (BOTTOM)

```
IMAD: BITS_8_71_64_Rc=Rc;
IMUL: BITS_8_71_64_Rc=*255;
```

This specialization mechanism can be applied to any part of an instruction definition. Another example is a general **IADD** instruction and a specialization **IADD32I** that fixes the data type to a 32 bit integer, rather than 8, 16, 32, 64 and 128 bits.

2.4.3 Full Instruction Class Example

This section illustrates a full *instruction class* definition based on the instruction **DMUL**. The reader should note that all relevant parts will be explained in detail and put into a relevant context later in their own dedicated sections. However, it is useful for the imaginative context of the reader to show a full *instruction class* definition at this point.

The **FORMAT** portion in Listing 2.10 defines the shape of the instruction as well as the cache bits. Cache bits refer to the lines encompassed in

$$\$ \; (\; (\; \ldots \;) \;) \; \$$$

because of the similarity between the term *cache* used in GPUs and *cash*, referring to *dollar signs* $.

Listing 2.10: Format definition for instruction class dmul__RCR_RC which is a variant for **DMUL**.

```
CLASS "dmul__RCR_RC"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /Round1("RN"):rnd
      Register:Rd
      ',' [-] [|!|] Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ',' [-] [|!|] C:Sb[UImm(5/0*):Sb_bank]*   [SImm(17)*:Sb_addr]
   $( { '&' REQ:req '=' BITSET(6/0x0000):req_bit_set } )$
   $( { '&' RD:rd '=' UImm(3/0x7):src_rel_sb } )$
   $( { '&' WR:wr '=' UImm(3/0x7):dst_wr_sb } )$
   $( { '?' USCHED_INFO("DRAIN"):usched_info } )$
   $( { '?' BATCH_T("NOP"):batch_t } )$
   $( { '?' PM_PRED("PMN"):pm_pred } )$ ;
```

The **CONDITIONS** depicted in Listing 2.11 define for all relevant portions of the **FORMAT**, which values constitute a valid instruction. For example, using the condition

$$(((Rd)+((Rd)==`Register@RZ)) \ \%\ 2) == 0:$$

- if destination register *Rd* is R2 (meaning *Rd* = 2), then the condition is *True*
- if destination register *Rd* is R3 (meaning *Rd* = 3), then the condition is *False* and the instruction invalid
- if destination register *Rd* is R255 = RZ (meaning *Rd* = 255), then the condition is *(255 + (255 == 255)) %2) = (255 + 1) %2 = 0 = True*, the instruction is valid as well

We can refer to these conditions as *validation conditions* for a given instruction. All of them must be *True* for an instruction to be valid.

Listing 2.11: Validation conditions definition for instruction class dmul__RCR_RC which is a variant for **DMUL**.

```
CONDITIONS
OOR_REG_ERROR
 (((Rd)==`Register@RZ)||(((Rd)<=(%MAX_REG_COUNT-2))&&((Rd)!=`Register@R254))) :
 "Register Rd is out of range"
MISALIGNED_REG_ERROR
 (((Rd)+((Rd)==`Register@RZ)) % 2) == 0 :
 "Register Rd is misaligned"
OOR_REG_ERROR
 (((Ra)==`Register@RZ)||(((Ra)<=(%MAX_REG_COUNT-2))&&((Ra)!=`Register@R254))) :
 "Register Ra is out of range"
MISALIGNED_REG_ERROR
 (((Ra)+((Ra)==`Register@RZ)) % 2) == 0 :
 "Register Ra is misaligned"
INVALID_CONST_ADDR_SASS_ONLY_ERROR
 ((Sb_bank <= 17) || (Sb_bank >= 24 && Sb_bank <= 31)) :
 "Invalid constant bank error"
MISALIGNED_ADDR_ERROR
 (Sb_addr & 0x7) == 0 :
 "Constant offsets must be aligned on a 8B boundary"
INVALID_CONST_ADDR_ERROR
 (Sb_bank >= 24 && Sb_bank <= 31) -> (Sb_addr <= 255) :
 "RTV banks may not use an offset greater than 255"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 (%SHADER_TYPE == $ST_CS) -> !(Sb_bank >= 8 && Sb_bank <= 31) :
 "CS may not use RTV banks"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 DEFINED TABLES_opex_2(batch_t,usched_info,reuse_src_a) :
 "Invalid combination of batch_t, usched_info, reuse_src_a"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 ((reuse_src_a==1)) -> ((usched_info==17)||(usched_info==18)||(usched_info==19)
          ||(usched_info==20)||(usched_info==21)||(usched_info==22)
          ||(usched_info==23)||(usched_info==24)||(usched_info==25)
          ||(usched_info==26)||(usched_info==27)) :
 "?DRAIN and ?WAITn_END_GROUP tokens are not allowed with .reuse"
```

As a side note: some validation conditions are more informative than others. For example, the condition *INVALID_CONST_ADDR_SASS_ONLY_ERROR* in Listing 2.12 informs about the expected range and layout of the constant memory banks.

Listing 2.12: Memory bank related condition for class dmul__RCR_RC which is a variant for DMUL.

```
INVALID_CONST_ADDR_SASS_ONLY_ERROR
   (( Sb_bank <= 17) || ( Sb_bank >= 24 && Sb_bank <= 31)) :
   "Invalid constant bank error"
```

The **PROPERTIES** portion of the definition in Listing 2.13 is not very informative apart from containing information about the source and destination operands. For example *VALID_IN_SHADERS* can be used to determine if an instruction is a valid CUDA instruction or if it can only be used in graphics shaders. *MIN_WAIT_NEEDED* from Section 2.5.14 is also defined here.

Listing 2.13: Properties definition for instruction class dmul__RCR_RC which is a variant for DMUL.

```
PROPERTIES
 INSTRUCTION_TYPE = INST_TYPE_COUPLED_EMULATABLE;
 IERRORS = (1<<IERROR_ILLEGAL_INSTR_DECODING)+(1<<IERROR_INVALID_CONST_ADDR_LDC)
         +(1<<IERROR_MISALIGNED_ADDR)+(1<<IERROR_MISALIGNED_REG)
         +(1<<IERROR_OOR_REG)+(1<<IERROR_PC_WRAP)+0;
 MIN_WAIT_NEEDED = 0 ;
 SIDL_NAME = `SIDL_NAMES@DMUL_C ;
 VALID_IN_SHADERS = ISHADER_ALL ;
 IDEST_OPERAND_MAP = (1<<INDEX(Rd));
 IDEST_OPERAND_TYPE = (1<<IOPERAND_TYPE_DOUBLE);
 IDEST2_OPERAND_MAP = (1<<IOPERAND_MAP_NON_EXISTENT_OPERAND);
 IDEST2_OPERAND_TYPE = (1<<IOPERAND_TYPE_NON_EXISTENT_OPERAND);
 ISRC_B_OPERAND_MAP = (1<<INDEX(Sb_bank))+(1<<INDEX(Sb_addr));
 ISRC_B_OPERAND_TYPE = (1<<IOPERAND_TYPE_DOUBLE);
 ISRC_C_OPERAND_MAP = (1<<IOPERAND_MAP_NON_EXISTENT_OPERAND);
 ISRC_C_OPERAND_TYPE = (1<<IOPERAND_TYPE_NON_EXISTENT_OPERAND);
 ISRC_E_OPERAND_MAP = (1<<IOPERAND_MAP_NON_EXISTENT_OPERAND);
 ISRC_E_OPERAND_TYPE = (1<<IOPERAND_TYPE_NON_EXISTENT_OPERAND);
 ISRC_A_OPERAND_MAP = (1<<INDEX(Ra));
 ISRC_A_OPERAND_TYPE = (1<<IOPERAND_TYPE_DOUBLE);
```

The **PREDICATES** section in Listing 2.14 offers insight into the sizes of the values that the instruction operands carry. This part is the least homogeneous and least reliable out of all, though and also exhibits the most change over multiple SM architectures. It is useful if the information can be extracted, though.

Listing 2.14: Predicates definition for instruction class dmul__RCR_RC which is a variant for DMUL.

```
PREDICATES
 IDEST_SIZE = 64;
 IDEST2_SIZE = 0;
 ISRC_B_SIZE = 64;
 ISRC_C_SIZE = 0;
 ISRC_E_SIZE = 0;
 ISRC_A_SIZE = 64;
 VIRTUAL_QUEUE = $VQ_REDIRECTABLE ;
 ILABEL_Ra_SIZE = 64;
```

The **OPCODES** section in Listing 2.15 is the simplest one. It defines the instruction opcode DMUL as well as its binary code that is used to encode the instruction. The *pipe* field is used in conjunction with the latencies Section 2.4.4.

Listing 2.15: Opdoces definition for instruction class dmul__RCR_RC which is a variant for DMUL.

```
OPCODES
        DMULfma64lite_pipe =  0b101000101000;
        DMUL =  0b101000101000;
```

The final section is **ENCODING**. It defines exactly how each part of an instruction is encoded to bits. This section of an *instruction class* is explained in great detail in the encoding and decoding Sections 3.2 and 3.2.

Listing 2.16: Encoding definition for instruction class dmul__RCR_RC which is a variant for DMUL.

```
ENCODING
!dmul__RCR_RC_unused;
BITS_3_14_12_Pg = Pg;
BITS_1_15_15_Pg_not = Pg@not;
BITS_13_91_91_11_0_opcode=Opcode;
BITS_2_79_78_stride=rnd;
BITS_8_23_16_Rd=Rd;
BITS_8_31_24_Ra=Ra;
BITS_1_73_73_sz=Ra@absolute;
BITS_1_72_72_e=Ra@negate;
BITS_5_58_54_Sc_bank,BITS_14_53_40_Sc_addr = ConstBankAddress2(Sb_bank,Sb_addr);
BITS_1_62_62_Sc_absolute=Sb@absolute;
BITS_1_63_63_Sc_negate=Sb@negate;
BITS_6_121_116_req_bit_set=req_bit_set;
BITS_3_115_113_src_rel_sb=VarLatOperandEnc(src_rel_sb);
BITS_3_112_110_dst_wr_sb=VarLatOperandEnc(dst_wr_sb);
BITS_2_103_102_pm_pred=pm_pred;
BITS_8_124_122_109_105_opex=TABLES_opex_2(batch_t,usched_info,reuse_src_a);
```

2.4.4 Latencies

Since Nvidia GPUs employ a software scheduler, it is necessary for the compiler to know exactly how long each instruction will take. The text files *latencies.txt*, that are also extracted from *cuobjdump* using DocumentSASS [13], contain numbers that look suspiciously like required cycles.

The **initial** assumption was: if it is possible to extract the latencies for all instructions from this text file, it would all but replace the need for microbenchmarking.

As it turns out, this assumption is wrong. There are multiple categories of instructions that need to be treated differently (see Figure 2.14). The fixed latency instructions category is the only one that would have benefited from this kind of decoding.

In this work, we were able to show that the Nvidia compiler inserts a latency value into specific bits in each instruction's encoding that tell the instruction dispatcher exactly how many cycles to wait after each instruction and also that this number varies depending on where each instruction is located in a kernel relative to other instructions that are related with data dependencies.

Meaning, if there are enough unrelated instructions to complete, the CUDA compiler seems to space dependent instructions as far away from one another as possible, having to wait as few cycles as possible using $USCHED_INFO as a result. See Section 2.5.15 for more information on **USCHED_INFO**.

Tutorial 6.9 shows that the fixed latency instructions require a minimal number of cycles to complete but can be made to use between 2 and 15 cycles using the $USCHED_INFO value. Since for a decoded CUDA kernel we can simply read this number out of all decoded SASS instructions and sum them up for all fixed latency instructions, fully mapping the latencies is useless, unless we intend to prove that the Nvidia CUDA compiler is not ideal.

Nevertheless, some effort was invested in decoding the tables in the latencies.txt file. Figure 2.9 depicts one decoded table for an **ISETP** variant. The prospective latency cycles are the right-most column. Note that the remaining part of fully mapping the instructions entails finding out how exactly how the values have to be summed up and also if the mapping in the individual rows is even correct. Some inspiration for this may be offered by the author of DocumentSASS [13].

2.4.5 SASS Format Expressions

In almost all components of an instruction class definition as outlined in Listings 2.11, 2.14, 2.13 and 2.16, there are expressions that look like equations. Especially the validation conditions (see Section 2.4.3) boast some rather

Latencies

TypeName	Input	TableName	Row	Col	Cross	Val
[T]	GPR	TABLE1	FXU_OPS{Rd, Rd2}	FXU_OPS{Rb, Ra, Rc}	0[x]{Rb, Ra}	6
[T]	SCOREBOARD	TABLE8	ALL_OPS_WITH_BMOV{sBoard}	ALL_OPS_WITH_BMOV{sBoard}	0[x]0	0
[T]	PRED	TABLE9	FXU_WITH_IMMA{Pd, Pv, Pu, nPd}	MATH_PRED_NO_FP16_FP64_OPS{Pb, Ps, Plg, Pq, Pa, Pp, Pc, Pr}	{Pv, Pu}[x]{Pp}	5
[T]	PRED	TABLE9	FXU_WITH_IMMA{Pd, Pv, Pu, nPd}	MATH_WITH_MMA{Pg}	{Pv, Pu}[x]{Pg}	13
[F]	GPR	TABLE2	FXU_OPS{Rd, Rd2}	FXU_OPS{Rd, Rd2}	0[x]0	1
[F]	PRED	TABLE10	FXU_WITH_IMMA{Pd, Pv, Pu, nPd}	FXU_WITH_IMMA{Pd, Pv, Pu, nPd}	{Pv, Pu}[x]{Pv, Pu}	1
[A]	GPR	TABLE3	FXU_OPS{Rb, Ra, Rc}	FXU_OPS{Rd, Rd2}	{Rb, Ra}[x]0	1
[A]	PRED	TABLE11	MATH_PRED_OPS{Pb, Ps, Plg, Pq, Pa, Pp, Pc, Pr}	FXU_WITH_IMMA{Pd, Pv, Pu, nPd}	{Pp}[x]{Pv, Pu}	1
[A]	PRED	TABLE11	MATH_WITH_MMA{Pg}	FXU_WITH_IMMA{Pd, Pv, Pu, nPd}	{Pg}[x]{Pv, Pu}	1

Figure 2.9: Depiction of the latencies table for one variant of ISETP. The rightmost column *Val* contains prospective cycle numbers for the instruction. The hard part would be to find out how exactly to sum these values up or find out if they have to be summed up or if there is some other mechanism involved.

involved expressions. Listing 2.17 showcases an example. Note that the pattern that most frequently occurs is the **implication** $\rightarrow$.

Listing 2.17: SASS format expression example: the most frequent expression pattern is the logical implication →

```
((reuse_src_a==1)) -> ((usched_info==17)||(usched_info==18)||(usched_info==19)
          ||(usched_info==20)||(usched_info==21)||(usched_info==22)
          ||(usched_info==23)||(usched_info==24)||(usched_info==25)
          ||(usched_info==26)||(usched_info==27))
```

The reason for introducing the sass format expressions now even if they will not be used before the assembler Chapter 3 is that their Python abstraction **SASS_Expr** is used everywhere, even for the value 0 in some places. Most things defined in the instruction classed definitions can be expressed as equation-like expression and conveniently evaluated by passing a set of variable values, that will later be introduced as *enc_vals* (Section 3.2.7).

As an interesting **piece of trivia**: the consumer version of Blackwell architecture (SM 120) contains one format expression that is 70kB long.

Components

Next to having all components that equations usually have (braces, equal signs, all the usual operators, ...), they also contain **constant values**.

- fields starting with `%` are PARAMETERS. The two most common ones are *MAX_REG_COUNT* and *SHADER_TYPE*
- fields starting with `$` are CONSTANTS. They are defined for each SM.
- fields starting with `` ` `` are used for register values. For example, a validation condition may impose that register `Rd == `Register@RZ`

Operator precedence

The assumption was that operator precedence follows the programming language C. This assumption was correct and operator precedence is resolved using an adaptation of Dijkstra's Shunting Yard [15] algorithm.

SASS_Expr

All format expressions are fully captured using the **SASS_Expr** Python class. In fact this Python class is so central to the developed software framework that it is the one that boasts two of the most important methods in the entire framework.

1. *assemble(..)*: takes a set of encoding values, some location indices and a vector of target bits and assembles the expression as actual CUDA binary bits. This process is explained in detail in Section 3.2.

2. *inv(..)*: takes SM properties, an instruction bit vector, some location indices and encoding values and disassembles the bits designated by the location indices into their respective format component. Inversion does not necessarily produce a unique result, which gives rise to the concept of *Universes*, for example in Figure 2.6. Disassembling instructions is covered in detail in Section 3.3.

SASS_Bits and SASS_Range

SASS_Bits and SASS_Range are the smallest units with which SASS_Expr and related concepts operate. Both feature C++ implementations connected to Python using Nanobind [16] for memory efficiency. In some calculations there may be millions of SASS_Bits and SASS_Range instantiated.

SASS_Bits is most commonly used with its *.from_int* and *.from_uint* initializers. They are characterized by three parameters and can be tightly packed using *Python*'s Pickle module or expressed as strings for convenience. Especially, they feature special *encoding modifiers* as methods that can be used during encoding and decoding of SASS instructions (see Chapter 3).

- **value**: the value is passed as signed `int64` type. *Python* uses the same bit range for its `int` representation
- **bit_len**: inside of SASS_Bits, the value is converted to an array of bits of this length. If the value that is passed requires more bits, an exception is thrown. This ensures effective control.
- **sign**: interpret the bits as signed or unsigned value influences how internal operations are performed, for example using two's complement.

SASS_Bits is used as *value* for all register types as well as function type values. For example in UImm(16) with a value of 0x0815, the value is `SASS_Bits.from_int(value=0x0815, bit_len=16, signed=False)`.

Since all computations related to format expressions are use a lot of *set* operations, SASS_Bits is often used inside of *Python* sets to express a range of value possibilities for example for a register operand.

Representing a sufficiently large range of function operands such as F32Imm or F32Imm using plain *Python* sets and SASS_Bits is not feasible. Thus SASS_Range is the equivalent of a set of SASS_Bits characterized by

- **minimal value**: this is an `int64` sized value
- **maximal value**: this is an `uint64` sized value
- **sign**: signed ranges will cover negative and positive values while unsigned ones start at 0

- **bit mask**: this is also an `int64` sized value. Bits set to 1 in the bit mask are ignored. This allows representing conditions such as "`(Sb_addr & 0x7) == 0`" from Listing 2.11 with one single **SASS_Range** using the bit mask `0x7`. If the bit mask is not zero, iterating over all values of a **SASS_Range** using its built-in iterator will omit the values covered by the bit mask. For example, if the bit mask is `0x7`, all values ending with binary `0b...111` will be omitted. Note that even with a non-zero bit mask, sampling a **SASS_Range** with its build-in `pick` mechanism is still uniformly at random.

SASS_Range can be iterated and sampled uniformly at random, the same way that a set of **SASS_Bits** can. The output of the **SASS_Range** `pick` mechanism is a **SASS_Bits** object. **SASS_Range** also supports all needed set operations.

With this knowledge, the reader may inspect Listings 2.11, 2.14, 2.13 and 2.16 again and discover more details.

2.5 Details

Sections 2.2 and 2.3 provide an introduction to how a SASS instruction is set up and what the individual components roughly do, this section expands on them by adding **in depth** details. Some of the concepts introduced in this section are complemented by a working example in the Tutorials Chapter 6.

2.5.1 Predicate

The predicate of an instruction refers to the blue part of the instruction template represented in Figure 2.1. It determines if the instruction it belongs to will be executed or not and exists for virtually all instructions. There are only very few exceptions on older architectures, older than SM 70 and not worth mentioning. In conjunction with **ISETP**, the predicate can be used to construct *if* statements.

The predicate format definition **always** looks the same apart from having a Predicate and a UniformPredicate version.

Listing 2.18: The format definition for the instruction predicate and its equivalent using UniformPredicate.

```
PREDICATE @[!]Predicate(PT):Pg
PREDICATE @[!]UniformPredicate(UPT):UPg
```

The predicate of an instruction always takes either Predicate or UniformPredicate registers, depending on its instruction dependent definition as given in Listing 2.18. Note that in the given examples, Figures 2.2, 2.3, 2.4, 2.5 and 2.6, the predicate is always PT.

PT (or UPT) represents the constant value *True*. An instruction would never be executed if the predicate was !PT (or !UPT). The remaining (uniform)predicate registers (U)P0 to (U)P6 can be set using *compare and set predicate* instructions, for example **ISETP**. In Figure 2.3, **ISETP** sets the predicate register P0 that could later on be used as instruction predicate !P0 or P0.

Both of the predicate register categories (uniform and regular) are encoded using 3 bits with one additional bit for the inversion operation [!].

2.5.2 Opcode

The **Opcode** is the placeholder for the instruction code bits and is what is displayed in a typical disassembly of a CUDA binary and is loosely listed on the Nvidia SASS website [10]. Each Opcode has a direct translation to bits. For example, **DFMA** is mapped to 0b101000101011.

There are much fewer actual opcodes compared to instruction classes. For example **DFMA** on SM 86 relates to no less than 9 different instruction classes. Each instruction class is one different variant for **DFMA**. Some feature different operand types (direct memory access or registers) while others may be specializations (see Section 2.4.2).

Note that the opcode is only one part of how instruction bits are mapped to their actual instruction and the mapping between an opcode name (for example **DFMA**), a binary code and an instruction class is arbitrary. For example, on SM 86, there is one **DFMA** with binary code 0b101000101011 and another one with binary code 0b1111000101011. On older architectures, mainly up until SM 62, the length of the operation code encodings varies between 3 bits and 13 bits and can even have interruptions. SM 70 and newer usually encode their instructions in 13 bits, though the registered binary code may be shorter and must be padded with zeros. In rare cases, the binary instruction code is given as decimal value and has to be translated to binary first, respecting the used encoding pattern.

As already pointed out, **Opcodes**, their binary codes and their respective instruction classes can be defined in all possible combinations. Only the names of the instruction classes are always unique. For example, all following combinations are possible and the enumeration may not be complete:

- Opcodes **Op1**, **Op2** both with binary code B1 in instruction classes C1 and C2
- Opcodes **Op1** once with binary code B1 and another time with B2 in instruction classes C1 and C2
- Opcodes **Op1** with binary code B1 in instruction classes C1, C2, C3, C4

The reader should remember that the **Opcode** is not uniquely responsible for identifying an instruction. On some older architectures there are even

multiple distinct, seemingly non related operations with the same binary code but different **Opcode** codes.

How instructions can be uniquely decoded is explained in detail in the decoding Section 3.3.3 about the instruction class fingerprint.

2.5.3 Register Operands

Recall Figure 2.1: every instruction may include *register operands* in the positions of destination and source operands. This is one of the two most basic forms that operands can take. The other form is functions that will be explained in the next Section 2.5.7.

The **IMAD** instruction is an example that has only *register* operands.

Register operands can be either destination or source registers and mostly belong to the following register categories:

- Register, UniformRegister (constant *zero*: RZ/URZ)
- Predicate, UniformPredicate (constant *True*: PT/UPT)
- SpecialRegister (many special names available, for example SR_CLOCL
- BarrierRegister (various subsets are available, for example B3B0)

Which category of registers to use is always clear from the instruction format definition (see Section 2.4)

Destination register

Listings 2.7 and 2.8 both contain one specific line for the destination: *Register:Rd* where *d* stands for *destination*. There are several variants of the definition for destinations.

- *Register:Rd*, this is the most common form
- *Register:Rd2*, some instruction have two destination registers, for example the **TEX** instruction in Figure 2.7. Graphics instructions are the most complex ones in the system.
- *UniformRegister:URd*, often there are instructions that have a *Register* and a *UniformRegister* version. For example **MOV** (Figure 2.4) has an equivalent **UMOV** instruction.
- *Predicate:Pu*, predicate setting instructions such as in **ISETP** featured in Figure 2.3 usually use *Predicate:Pu* as the destination definition

Note that the *naming* patterns used in the format definitions should not be relied upon without limits. They evolve over multiple SM architectures. If one specific SM is targeted, it is possible to verify that some naming pattern always hold that can then be used to perform analyses.

Source register

Inspecting Listings 2.7 and 2.8 we detect two additional specifications: *Register:Ra* and *SImm(32)*:Sb*. The latter one is a function and will be treated in Section 2.5.7. *Register:Ra* specifies a source register. Source registers are almost always enumerated with *a,b,c*, then *d* is left out for the *destination* and the more advanced architectures continue with *e*. Some examples for source register specifications are

- *Register:Ra/b/c/e*, this is the most common form
- *UniformRegister:URa/b/c/e*, often there are instructions that have a *Register* and a *UniformRegister* version. For example **MOV** (Figure 2.4) has an equivalent **UMOV** instruction.
- *[!]Predicate:Pr*: instructions such as **ISETP** featured in Figure 2.3 have versions that support *chaining*. For this they have one chained predicate source operand that inserts the result of a past instruction in the chain, featuring inversion [!] capability (See Tutorial 6.10).

2.5.4 Register Size

As indicated in multiple places, there are various bit sizes to work with. The usual suspects are 16, 32, 64 and 128 bits while newer architectures also feature 8 and 256 bits.

It is very important to understand how these sizes interact with the register specifications to not accidentally use, load or write *garbage* values.

The smallest *register unit* are the *Register* category (R0, ..., R255). Each one covers **at most** 32 bits. For larger values, multiple, consecutive registers are used. For example, a 64 bit value may use R2 and R3. This is **especially** important for registers that represent **addresses** since they are all 64 bit wide!

To load 64 bits using regular *Register* registers, load the lower bits in the even numbered register, for example R2 and the upper bits in the subsequent odd register R3. This is a common pattern for example using the **MOV** or **IMAD** instructions to load *kernel arguments*. See Tutorials 6.6 and 6.8 for examples.

Load and store instructions **LDG** and **STG** can be given a size as extension.

UniformRegister registers meaning all UR0, ..., UR63 can be initialized using the **ULDC** instruction, that can take a size as extension as well.

2.5.5 Register Sets

The *register values* are defined with *set semantics*. For example, *unsigned integers* may be defined as in Listing 2.19 and afterwards unified to a larger set as in Listing 2.20.

Figure 2.10: Loading from constant memory using UniformRegisters. Note the extension .64, denoting that the target register UR6 has to load a 64 bit value, in this case the address located at constant memory bank 0x0 with offset 0x170, which in this particular example, is the third argument in a kernel.

Listing 2.19: Define register values for unsigned integers

```
UInteger8 U8 = 0;
UInteger16 U16 = 2;
UInteger32 U32 = 4;
UInteger64 U64 = 6;
```

Listing 2.20: Unify multiple uint lengths into one larger set.

```
UInteger = UInteger8 + UInteger16 + UInteger32 + UInteger64;
```

This mechanism is used extensively in instruction format definitions. For example, an instruction may limit the available registers for an operand to only *UInteger*. Note that *NonZeroRegister* also is exactly this mechanism at work.

2.5.6 INVALID Register Values

Some *register categories* are padded with *INVALID* fields. This mechanism seems to be used for two purposes

- fill register definitions up to a power-of-two length (see Listing 2.21)
- invalidate a previously valid register value by prefixing it with *INVALID* (see Listing 2.22)

Listing 2.21: Use invalid to pad register value definitions up to a power-of-two length.

```
TEXWmsk34C RGB=0, RGA, RBA, GBA, RGBA, INVALID5=5, INVALID6=6, INVALID7=7;
```

Listing 2.22: Use invalid to invalidate previously valid register values by simply prefixing the register value name with INVALID.

```
XXHI noX, INVALIDSHRXMODE1, X, XHI;
```

It is unclear if the *INVALID* registers are considered *valid* or not to use in instructions. Sometimes they are explicitly excluded by *validation conditions* but most of the time this is not the case.

2.5.7 Function Operands

Functions are the second basic way (next to registers) to specify operands for an instruction. Let's examine the MOV instruction illustrated in Figure 2.4 again. It moves the immediate value 0x0 into the destination register R31, bitmasked with the immediate value 0xf, meaning, not doing anything.

PT MOV R31 UImm(0x0) UImm(0xf)

We call UImm(0x0) a *Function*. Not because it exhibits functional behavior but because it looks like one. In fact, this naming is rooted in the parser, where all of these constructs were given this name to distinguish them from registers. Functions have a name and an argument. For example in the format for the instruction MOV in Listing 2.23.

Listing 2.23: Instruction format defintion for one of the MOV instruction variants

```
CLASS "mov__RI"
  FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
    Register:Rd
    ','UImm(32)*:Sb
    ','UImm(4/0xf)*:PixMaskU04
```

Examining Listing 2.23, we see the definitions *UImm(32)*:Sb* and *UImm(4/0)*:PixMaskU04*. Both of them serve to inject *immediate values* directly into a program. The *arguments* are two integers divided by a "/". The first number denotes *number of bits for the value* and the second one the *default value*. Default values are all ignored by the framework implemented for this report.

Namings for Function fields follow a couple of different schemes, related to their implied functionality:

- *Sa/b/c/e*: these are used to frame arbitrary values induced into an instruction as *immediate values* and follow a similar scheme as with registers: *Sa/b/c* then *d* is left out and more modern architectures may feature an *Se*
- *Sa/b/c_addr/offset*: these are used to frame addresses or address-offsets
- this enumeration is not exhaustive. For example Listing 2.23 features a function *PixMaskU04*.

The reader should note that these namings don't have any syntax implication at all, apart from being the same in multiple places to connect things in the definition.

Sidenote: to this day, it is unknown what the little stars in the format definitions "*" mean.

The Functions perform the duty of framing loose values with a bit length. Here is an exhaustive list of all function names that occur. Note that *lists* and *attribute* style operands are explained in detail in the next three Sections 2.5.8, 2.5.9 and 2.5.10.

- *UImm*: unsigned, immediate integer value, used most often to frame immediate integer values moved into registers or added to other values or within list or single/double attribute operands to denote address offsets
- *RSImm*: signed, immediate integer value, usually used for address offsets that are not packed into a list or attribute style operand
- *SImm*: the signed equivalend for UImm, used for signed immediate integer values or address offsets within list or attribute style operands
- *SSImm*: this is a relict used only in SM 62 and older in a similar way to RSImm
- *F16Imm*: regular 16 bit, signed floating point immediate value
- *F32Imm*: regular 32 bit, signed floating point immediate value
- *F64Imm*: regular 64 bit, signed floating point immediate value
- *BITSET*: always used for a 6 bit bitmask field in the cache bits. Their usage is explained in detail in Section 2.5.17.
- *E8M7Imm*: special 16 bit signed floating point number with 8 bit exponent and 7 bit mantissa, used exclusively as format templates for F16Imm floats in the *convertFloatType* encoding modifier (explained in the encoding Section 3.2.5)
- *E6M9Imm*: special 16 bit signed floating point number with 6 bit exponent and 9 bit mantissa, used exclusively as format templates for F16Imm floats in the *convertFloatType* encoding modifier (explained in the encoding Section 3.2.5)

2.5.8 List Operands

List operands are used to calculate memory access addresses, using various sources, including potentially multiple registers of various kinds and immediate values representing relative or absolute offsets. As such, they are very common.

Figure 2.5 is an example for this kind of operand. Figure 2.11 is an example for an attribute operand that will be explained in the next section. Note that single/double attribute operands are prefixed with a register (in the case of Figure 2.11, this register is called *A*).

All the operands' values in a list operand are summed up. In the case of Figure 2.5, the operands of

[RZ, UR4, SImm(0x0)]

produce a memory access at value(RZ)+value(UR4)+0x0.

This is a good place to highlight another mechanism that is frequently used in instruction class definitions. Close inspection of the format definition for this variant of **STG** in Listing 2.24 shows the first entry in the list operant as *ZeroRegister("RZ"):Ra*, yielding RZ as the only possibility in the first spot of the list operand. This mechanism is explained further in the instruction fingerprint Section 3.3.3.

Listing 2.24: Instruction format defintion for one of the **STG** instruction used in Figure 2.5. Note the first entry in the list operand's definition: ZeroRegister("RZ"):Ra. This is one mechanism on how to narrow the available bandwith of registers to use in instruction operands. *ZeroRegister* is technically a subset of *Register*, the other one being *NonZeroRegister*.

```
CLASS "stg_uniform__RaRZ"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
         /E("noe"):e
         /COP("EN"):cop
         /SZ_U8_S8_U16_S16_32_64_128("32"):sz
         /SEM("WEAK"):sem /SCO("nosco"):sco
         /PRIVATE("noprivate"):private
      [ ZeroRegister("RZ"):Ra + UniformRegister:Ra_URc + SImm(24/0)*:Ra_offset ]
      ','Register:Rb
```

Listing 2.24 also illustrates the loose naming convention for entries in list operands. Note that all entries contain the term *Ra*. Apart from fairly consistently enumerating operands with *a/b/c/e* in sequence of their place in the instruction definition, there is no deeper meaning or syntactic rule to be extracted.

In the *encoding* Section 3.2, the parts of the operands as outlined so far will be explained and used in more detail. As already pointed out in Section 3.2.1, we call the designations after the colon *alias*.

2.5.9 Single Attribute Operands

Single attribute operands are similar to register operands (Section 2.5.3) with one list attached as attribute. The reason for naming these operands *single* attribute operands is explained in Section 2.6.4.

Listing 2.25 shows the format for the **LDTRAM** instruction depicted in Figure 2.11. At this point it is worth pointing out, that it is not possible

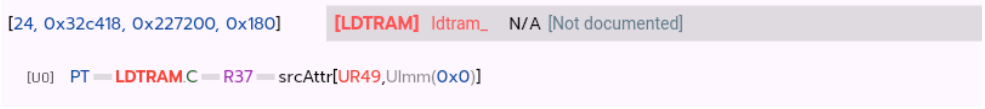


Figure 2.11: Example for the simplest kind of attribute operand: one prefix combined with one list operand.

to associate every instruction example used in this report with a specific function, because the instructions are not fully documented, not even with one sentence. Especially for rare formats, sometimes the first best example instruction is used without also explaining what the instruction does because the format **definition** is the important part.

Listing 2.25: LDTRAM instruction is one of the few ones featuring an attribute operand.

```
CLASS "ldtram_"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /MODE_ldtram:mode
      Register:Rd
      ',' A:srcAttr[ UniformRegister("URZ"):URa + UImm(10/0)*:URa_offset ]
```

A:srcAttr is the same kind of notation as *Register:Rd*: *A* is the category of registers and *srcAttr* is the name alias. To keep in line with the usual notation guidelines, there could also exist a *B:srcAttr*, if the operand was in second position of the source operands.

An exhaustive list of all attribute alias names up until Blackwell (SM 120) is given here:

- desc, idesc
- gdesc, gdescA, gdescB
- srcAttr, ttuAddr
- indexURd, indexURc, indexURb
- tmemA, tmemB, tmemC, tmemE, tmemI

While these designations are somewhat indicative of what they stand for (for example *tmemA* is used in instructions that seem to be related Hopper's new distributed memory system and the *t* may be a placeholder for *tensor*), their semantic meaning is the same as for every register *alias* and the naming process may be similar to how regular programmers come up with variable names. Thus finding a deeper meaning in these designations is not necessarily productive.

2.5.10 Double attribute operands

Double attribute operands extend the single attribute operands from the previous Section 2.5.9 with a second list attribute. This notation is very

common, and several already introduced instruction examples illustrate this notation: Figure 2.2, 2.6 and 2.10. The reason for why the attached lists are named 'attributes' is explained in Section 2.6.4.

We start with **DFMA**.

Listing 2.26: The double attribute operand is very common. **DFMA** instruction, depicted in Figure 2.2 is just one instruction using this notation to facilitate constant memory access.

```
CLASS "dfma__RRC_RRC"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /Round1("RN"):rnd
   Register:Rd
      ',' [-] [||] Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ',' [-] [||] Register:Rb {/REUSE("noreuse"):reuse_src_b}
      ',' [-] [||] C:Sc[UImm(5/0*):Sc_bank]*   [SImm(17)*:Sc_addr]
```

Examining the third source operand in Listing 2.26 reveals the naming aliases *Sc_bank* and *Sc_addr*. This is a hint that this instruction accesses the GPU constant memory at memory bank *Sc_bank* at absolute offset *Sc_addr*. There are a range of *restrictions* imposed on this notation for which values are allowed for these two locations. This restriction mechanism will be explained in detail in the instruction generator Section 3.4 where the validation conditions are evaluated in detail.

Since this notation is very common, there are plenty of instructions where the memory access is either the first, second or third source operand and the *alias* naming *Sa/b/c* also changes with the position:

- *C:Sa[UImm(5/0*):Sa_bank]* [SImm(17)*:Sa_addr]*
- *C:Sb[UImm(5/0*):Sb_bank]* [SImm(17)*:Sb_addr]*
- *C:Sc[UImm(5/0*):Sc_bank]* [SImm(17)*:Sc_addr]*

The second example is instruction **STG** in Figure 2.6.

This example is important as it highlights the introduction of *memory descriptors* that came with SM 80 (Ampere). The reader is encouraged to examine the two examples depicted in Figures 2.5 and 2.6 again. Figures 2.5 shows the older version of **STG**, also available on GPUs with compute model SM 70 to SM 75, using only a *list operand*, while Figure 2.6 depicts an **STG** only available on GPUs with compute model newer or equal to SM 80, utilizing a *memory descriptor*.

Listing 2.27: SM 80 introduces memory descriptors. Thus from SM 80 onwards there are **STG** instructions with *memoryDescriptor* as name alias using the register *DESC*.

```
CLASS "stg_memdesc__Ra64"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
         /EONLY:e
         /COP("EN"):cop
```

```
        /SZ_U8_S8_U16_S16_32_64_128("32"):sz
        /SEM("WEAK"):sem /SCO("nosco"):sco
        /PRIVATE("noprivate"):private
        /EXP_DESC("noexp_desc"):e_desc
    DESC:memoryDescriptor[UniformRegister:Ra_URc]
                         [Register:Ra
                            /ONLY64:input_reg_sz_64_dist + SImm(24/0)*:Ra_offset]
    ','Register:Rb
```

An exhaustive list up until Blackwell (SM 120) of all double attribute alias names is given here:

- srcConst
- Sa, Sb, Sc
- memoryDescriptor

srcConst is always used in instructions that access constant memory. The same is true for *Sa/b/c*, though, *srcConst* is used in all older architectures too while *Sa/b/c* is introduced only in later architectures. *memoryDescriptor* is introduced in SM 80. In fact, most memory accessing instructions that use pure *list* operands in SM 75 and earlier, use *memoryDescriptor* starting with SM 80.

2.5.11 Prefix operations

At this point, the reader may have noticed artifacts in the format notation looking such as *[-]* or *[||]* appearing at the beginning of operand definitions. Listing 2.26 contains an. These depict operations performed on the value of the operand before it is *inserted* into its place in the instruction.

An exhaustive list of all available operations is

- *[-]*: negate the value
- *[∼]*: bit-wise inversion
- *[!]*: logical inversion
- *[||]*: absolute value

Prefix operands are common for register, single, and double attribute operands.

Figure 2.12: DMUL operation with negated register operand and absoluted constant memory access.

2.5.12 Extensions

Finally, this is the last piece of the instruction class format that needs to be addressed. Extensions have been mentioned often up to this point. This section will provide more details and more examples as well as introduce some of the useful and more common extensions.

All examples up to this point have extension registers. The reader may for example check out Figure 2.6 or 2.2. Lists can also have extensions. One example is shown in Figure 2.13.

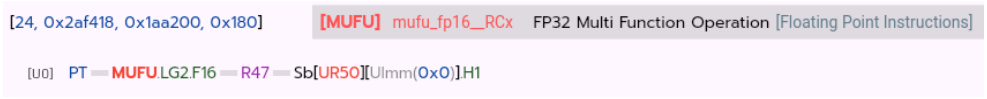


Figure 2.13: MUFU operation with an extension added to the double attributes.

It is fair to say, that extension registers are one of the SASS features that have become more extensive towards more current models and that at this point every SASS component apart from predicates (see Section 2.5.1) and functions (see Section 2.5.7) can have extension registers to further configure the component.

Input/Output Size

One very common extension sets input and output sizes.

The classic example is LDG (load from global memory) and its twin reversed operation STG (store to global memory). Both use the extension register *SZ_U8_S8_U16_S16_32_64_128*. Luckily some registers contain in their name what they define: {U8=0, S8=1, U16=2, S16=3, 32=4, 64=5, 128=6}

Reexamining Figure 2.5 and 2.6 we see that the STG instruction features .32 and in 2.6, the list operand register R2 is extended by .64.

Another good example are conversion instructions F2I (float to integer conversion) and its inverse I2F (integer to float conversion). With these instructions, the *Float*-side is usually restricted to one or at most two formats, for example 32 or 64 bit floats while the *Integer* side offers a wider range of options, often signed or unsigned as well as 8 bit up to 128 bit.

A third example are float-to-float conversion instructions F2F instructions. There are downconversion and upconversion variants. Their size extensions combine differnet floating point formats, for example F16 and E8M7 formats.

The last is HMMA (matrix multiply and accumulate) and IMMA (integer matrix multiply and accumulate) where the matrix formats are passed to the instruction using extension registers.

There are various more extension registers denoting sizes. A non-exhaustive list related to the presented 4 instructions and aiming at relating intuition to the reader, is given here:

- **SZ_32_64_128**
- SZ_U8_S8_U16_S16_32_64
- SZ_U8_S8_U16_S16_32_64_128, **used** for example in **STG**, **LDG**, **ULDC**
- **DSTFMT_U64_S64**
- DSTFMT_U8_S8
- DSTFMT_U8_S8_U16_S16_U32_S32, **used** in **F2I** as destination size
- **SRCFMTA_U8_S8_U4_S4**
- SRCFMT_U16_S16, **used** in **I2F** as destination size
- **DSTFMT_SRCFMT_F16F32_E8M7F32_E6M9F32_BF16F32**
- DSTFMT_SRCFMT_F16F64_F32F64
- DSTFMT_SRCFMT_F32F16_F32E6M9, **used** in **F2F** as source and destination formats
- **SIZE_1688_16816_1684**
- SIZE_8816_8832_16816_16832_16864
- SIZE_8832_8864_16832_16864_168128, **used** in **HMMA** and **HMMA** as matrix formats,

Reuse/Noreuse

Many instructions feature .reuse/.noreuse extensions. For example instruction **DFMA** in Figure 2.2 offers this setting for the first and second source operand.

The extension is only ever available for registers, never for immediate values or memory accesses.

According to "Decoding CUDA Binary" [11], these bits encourage the GPU to keep values in the cache. This could not be verified in this work. It is however telling, that the setting .reuse is only available with the cache bits configuration transX. As explained in the next Section 2.5.13, this setting encourages the GPU to switch threads. Thus another meaning is postulated: reuse across different threads rather than encourage caching.

Since all experiments in this work are single threaded, this could not be verified. For instructions that feature the *reuse/noreuse* functionality, casual

inspection of benchmarking result yielded no difference in the cycles in both modes.

The *.reuse/.noreuse* extension is encoded in the $cache bits. It merits further investigation.

2.5.13 Instruction Categories and SASS Scheduler

Nvidia employs a **software scheduler**. This means, that every single instruction must know either exactly how many cycles it takes or use another mechanism to determine when the result is available.

In fact, all SASS instructions can be subdivided into categories as depicted in Figure 2.14:

- **control or sync instruction**: these instructions include things like **BRA** and **EXIT**. They cannot be benchmarked since the first one branches to a different part of the program and the second one terminates it. These instructions utilize a *min_wait_needed* value explained in Section 2.5.14.
- **data instruction, fixed latency**: these are the most well behaved instructions. They always take exactly the same number of cycles. This category includes **FFMA** (fused multiply and add, 32bit), **FADD** (float addition, 32bit). They don't need to be benchmarked since their *latencies* can simply be extracted from the decoded instruction's $USCHED_INFO value (see Section 2.5.15).
- **read only data instruction, variable latency, using a barrier mechanism**: this category includes for example **DADD** (double precision float addition) and **DMUL** (double precision float multiplication). All of these instructions feature at least one barrier (read or write). Any memory accesses are read only to constant memory. To find the latency of these instructions, the cycle count has to be measured with a benchmarking program (see Section 7.3.1).
- **read/write data instruction, variable latency, using a barrier mechanism**: this category includes all memory read and write instructions, for example **LDG** and **STG**. To find the cycle count of these instructions, they have to be measured using a benchmarking program (see Section 7.3.2).

The following $cache bits sections deal with this categorization as well as explain in detail how the fixed-latency and the barrier mechanisms work. The next Chapter 3 will then explain in great detail how instructions are encoded and decoded.

First things first: the attentive reader has probably realized that apart from the the full format definition in the full instruction class section, depicted

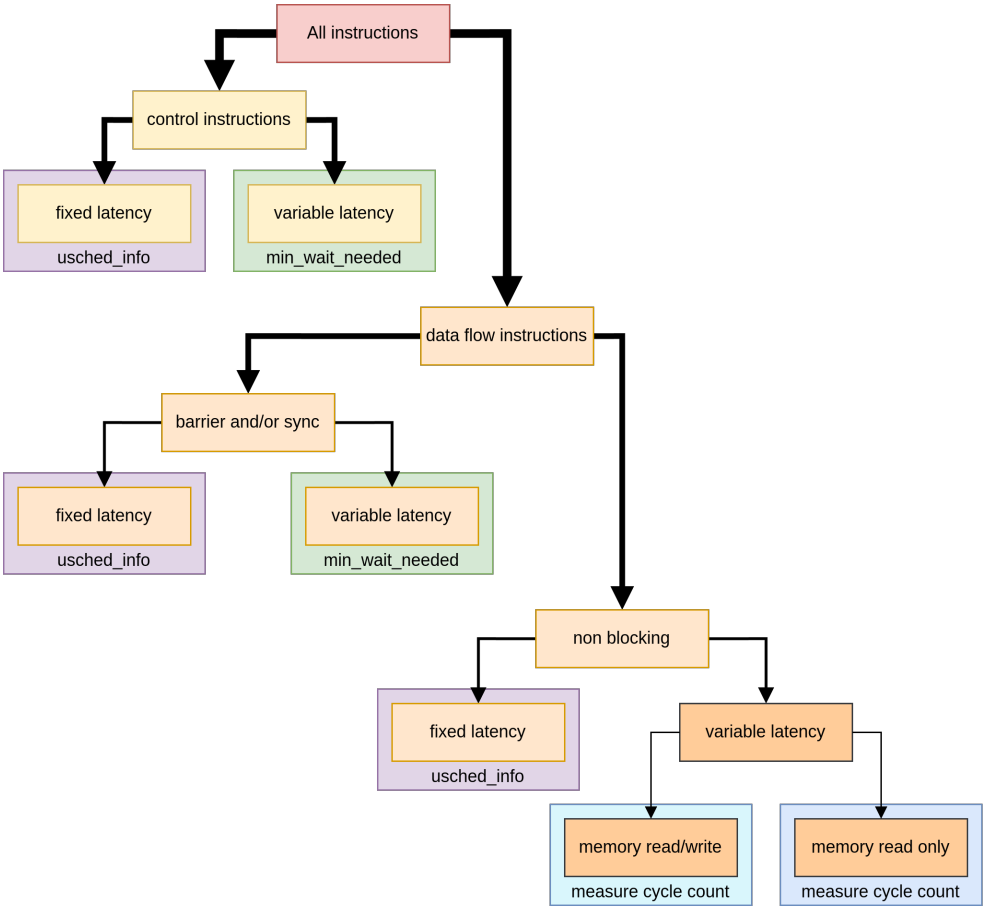


Figure 2.14: Subdivision of all SASS instructions into categories based on scheduler properties.

in Listing 2.10, the *cache* bits have always been missing from any format definition.

This section and the following two Sections 2.5.16 and 2.5.17 make up for it by first completing the incomplete definition of DFMA in Listing 2.26 with its cache bits in Listing 2.28 and also explaining what most of the entries mean and how they work. The depicted variant of DFMA is shown in Figure 2.2.

Listing 2.28: The full format definition for instruction DFMA, including its cache bits.

```
CLASS "dfma__RRC_RRC"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /Round1("RN"):rnd
      Register:Rd
      ',' [-] [|!] Register:Ra {/REUSE("noreuse"):reuse_src_a}
      ',' [-] [|!] Register:Rb {/REUSE("noreuse"):reuse_src_b}
      ',' [-] [|!] C:Sc[UImm(5/0*):Sc_bank]*   [SImm(17)*:Sc_addr]
   $( { '&' REQ:req '=' BITSET(6/0x0000):req_bit_set } )$
   $( { '&' RD:rd '=' UImm(3/0x7):src_rel_sb } )$
   $( { '&' WR:wr '=' UImm(3/0x7):dst_wr_sb } )$
   $( { '?' USCHED_INFO("DRAIN"):usched_info } )$
   $( { '?' BATCH_T("NOP"):batch_t } )$
   $( { '?' PM_PRED("PMN"):pm_pred } )$ ;
```

2.5.14 Min Wait Needed

This section explains the *variable latency* category in Figure 2.14 dubbed *min_wait_needed*. The value is extracted from the instruction class definition properties section. It exists for **all** instruction classes in all supported SMs (SM 50 to SM 120).

Since there exist instructions, for example BRA and EXIT, that either change the control flow or indicate waiting for some barrier that is set somewhere else, potentially in another thread or exit the kernel completely and are thus difficult to benchmark, we have to assume a baseline cycle count for those.

Instead of just inventing some number, we utilize *min_wait_needed* from the instructions class definition's PROPERTIES section (see Section 2.4). This number is greater than 0 for all instruction classes in this category. Incidentally it is also exactly 1 for all instructions where it is not 0.

The instruction dispatcher is assumed to be able to dispatch one to two instruction per cycle, this is reasonable since the minimal wait time in the $USCHED_INFO register category is WAIT1 and trans1, even though, in Tutorial 6.9 we see that on SM 86, the minimal cycle count for any instruction is 2. The next Section 2.5.15 explains $USCHED_INFO in depth.

For a coherent cycle counting performance model, the instructions in this category may require special treatment:

- branching instructions, for example BRA, spawn multiple edges in a potential control flow graph along which cycles can be accumulated.
- synchronization instructions, accessing some externally set barrier, for example BAR, need to assume that the linked barrier is always in a signaled state and point out some potential synchronization issues in a different way.

2.5.15 Cache bits: Scheduler

This section deals with the $USCHED_INFO portion of the $cache bits. An example showcasing how $USCHED_INFO works is in Tutorial 6.9.

Examining Figures 2.2, 2.3 and 2.5 the reader can examine three different settings for the *USCHED_INFO* field.

- $USCHED_INFO:[WAIT1_END_GROUP]=0x1 (Figure 2.2)
- $USCHED_INFO:[WAIT15_END_GROUP]=0xf (Figure 2.3)
- $USCHED_INFO:[trans2]=0x12 (Figure 2.5)

The implied values (15 for WAIT15_END_GROUP and 2 for trans2) are **exactly** the **number of cylces** the instruction dispatcher in the GPU waits after dispatching an instruction with that particular *USCHED_INFO* configuration.

WAIT1_END_GROUP is different: the GPU instruction dispatcher always seems to wait at least 2 cycles. This may be specific to SM 86, though.

15 cycles is the maximal amount the instruction dispatcher can be instructed to wait after dispatching an instruction. It is possible, though, to stuff a kernel with **NOP** instructions with wait cycles up to 15 as is done in one of the benchmarking techniques used to measure instructions that take more cycles than just 15. See Tutorial 6.12 for an example.

As indicated in "Decoding CUDA Binary" [11], the difference between WAITx and transX is that the latter encourages a switch between threads. That is why instructions accessing memory or performing expensive operations usually use transX rather than WAITx.

2.5.16 Cache bits: Barriers

Focusing on Figures 2.2, 2.3 and 2.5, the reader can detect yet another component in more detail: barriers. All instructions decoded by the Cubinext VSCode extension from Chapter 4 feature this component in their representation, but not all of them actually have them. The decoder *augments* all instructions to the same components. The *augmentations* are highlighted with a suffix [A] with the value that makes the respective field do nothing. For example

- $WR[A]=0x7
- $RD[A]=0x7

Listing 2.29 borrows the relevant portion in the format definition from Listing 2.28. All instructions classified as *variable latency* in Figure 2.14 all have at least one of the definitions. All instructions classified as as *fixed latency* in Figure 2.14 have none.

Listing 2.29: Read/Write barrier cache bits in a format definition that features them.

```
$( { '&' RD:rd '=' UImm(3/0x7):src_rel_sb } )$
$( { '&' WR:wr '=' UImm(3/0x7):dst_wr_sb } )$
```

For read/write barriers ($RD, $WR), the value 0x7 means, **wait for no barrier**:

- 0x7: no barrier
- 0x0 to 0x5: wait for enumerated barrier
- 0x6: does not exist

What Is "Read" and What Is "Write"?

The instruction **STG** *stores* to global memory while **LDG** *loads* from it. Figures 2.5 and 2.6 show that the **STG** instruction only has a *read* barrier $RD.

This makes sense if one takes the point of view of *resources*, for example the *memory system* or a specific pair of *32 bit floating point multipliers*. Using the instruction **STG**, the memory system **reads** data and signals a $RD barrier when finished.

A Bit of Chaos

The author has encountered instances where instructions featuring both $RD and $WR barriers only react to one of them. Generally, when in doubt, use the $RD barrier. Better is, if an instruction features two barriers, test it. The tutorials Chapter 6 outlines general techniques for how to do this easily.

Another example is instruction **SETCTAID**. Note that this instruction can only be used in a graphics context and is not a valid CUDA instruction. It can still serve as example, since the author can't completely rule out that it is the only example.

Its format definition contains the full definition for a $WR barrier.

```
$( { '&' WR:wr '=' UImm(3/0x7):dst_wr_sb } )$
```

Its validation conditions also contain a *condition*

```
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
   (dst_wr_sb == 0x7) :
      "SETCTAID cannot specify a write scoreboard"
```

making it illegal to set a $WR barrier. As outlined in the instruction classes Section 2.4, all *validation conditions* must evaluate to *True* for an encoding to be valid. Thus, this instruction's format definition allows setting bits that then make the encoding *invalid*.

Augmented Barrier Bits

Absolutely **all** instructions **encode** the barrier bits. Instructions that do not contain the barrier fields in their format definitions will encode those bits with the fixed values 0x7. The decoder from Chapter 4 shows barrier bits that are not defined in the format definition of an instruction with an [A] suffix. It *augments* instruction formats at the decoding stage with all missing $cache definitions. For example, if a $WR definition is missing, one is added.

This was done so to test if it is possible to *manually* set barriers for instructions that do not contain them in their format definitions. It turned out to not work: **STG** instruction in Figure 2.24 has one *read barrier* $RD. The write barrier is represented with the suffix [A] $WR[A]=0x7. As as showcased in Figures 2.15 and 2.16, an attempt was made to use different values for the write barrier. The result is, if instructions that do not explicitly have barriers in their format definition are encoded with anything else than 0x7 in the barrier bits locations, the kernel will not work.

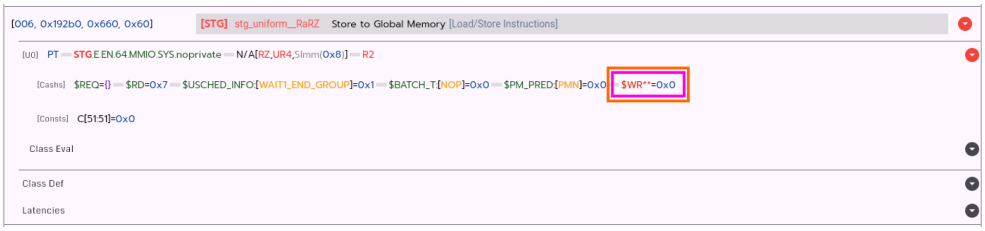


Figure 2.15: Comparing to Figure 2.5, this version of **STG** was changed to include a $WR barrier, as indicated in red with two following stars setting write barrier 0x0, even though, the **STG** instruction does not have a write barrier: $WR**=0x0

```
.sandwitch.template_86.test_0x0 0
Run mod.experiment.sandwitch.template_86.test_0x0 with input = 0
 + Before kernel mod.experiment.sandwitch.template_86.test_0x0
       finished: output[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] = 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 101
1, 1012, 1013, 1014, 1015
CUDA Error: unspecified launch failure
 + After kernel mod.experiment.sandwitch.template_86.test_0x0
       finished: output[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] = <+>1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010,
1011, 1012, 1013, 1014, 1015<->
```

Figure 2.16: Trying to run a CUDA kernel with this modified instruction produces a CUDA error unspecified launch failure.

Thus, very **importantly**, the instructions that do not feature barriers in their format definitions, cannot use barriers!

For the ones that do, as explained in detail in the benchmarking Section 7.3.4, using barriers actually introduces an overhead. For example, on SM 86

(Ampere), DFMA as illustrated in Figure 2.2 requires 55 cycles if it waits for its barrier to be signaled but the correct result is already available after 48 cycles.

This gap in number of cycles is consistently present in all non-memory related instructions featuring barriers (DFMA, DMUL, DADD, I2F, F2I, ...): using barriers on SM 86 implies a base cost of 7 cycles.

2.5.17 Cache bits: Wait Requirement

None of the examples used so far (Figures 2.2, 2.3, 2.5 and 2.6) would actually wait for a barrier.

The mechanism to wait for a barrier is set using the $REQ bits. Listing 2.30 borrows the relevant format portion from Listing 2.28. $REQ is defined as BITMASK(6/0x000000). Recalling the Functions Sections 2.3.2 and 2.5.7 this is a function format using 6 bits with an implied default value 0x0. Figure 2.17 shows an example that waits for all barriers.

Listing 2.30: The $REQ bits depict a set illustrating a **bitmask** with a bit set to 1 to wait for a specific barrier.

```
$( { '&' REQ:req '=' BITSET(6/0x0000):req_bit_set } )$
```

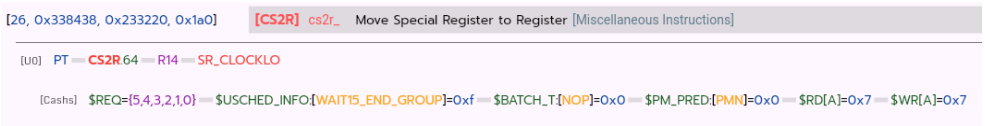


Figure 2.17: CS2R instruction waiting for all existing barriers using $REQ={5,4,3,2,1,0}.

The bit limitation of this field implies that each instruction can at most wait for 6 barriers. Note that each of the $RD/$WR fields can represent up to 6 barriers individually with a total of 12 split evenly between reading and writing

Unexpected blocking

It does seem that instructions that have barriers can block for more than indicated in $USCHED_INFO. For example FADD, DADD and DMUL where found to **block** for 34 to 35 cycles if $USCHED_INFO=WAITx was used. This phenomenon has not been investigated further yet.

2.6 Parser and the py_sass Module

This is the last section in this chapter. Up to this point, the reader has seen plenty of evidence for a formal *Python* representation of SASS instructions. This formalization is achieved by parsing the output produced by DocumentSASS[13]. It offers a way to extract text files dubbed *instructions.txt* and another set dubbed *latencies.txt* out of the Nvidia tool *cuobjdump* [10].

This section covers the parser that turns the text output into *Python* to the extent that is necessary to facilitate a good entry point into utilizing the *Python* framework developed during this thesis and presented in this report. The next Chapter 3 details how formal instructions as outlined in this chapter are encoded and decoded to and from their bit representations that run on Nvidia GPUs.

All concepts presented in this chapter are covered by the *py_sass* module.

One portion of the extracted data defines all available registers, constants and lookup tables. The vast majority, though, defines the instruction classes previously described in the instruction classes Section 2.4.

The *instructions.txt* files are roughly split into the sections

- introductory properties and constants
- register definitions
- lookup table definitions
- instruction class definitions

2.6.1 Resources and Registers

Introductory properties include

- number of bits per word (luckily always 64)
- number of bits per instruction (88 in SM 62 and earlier, 128 in all newer SMs)
- various names and constants used in all kinds of definitions

All of these definitions are used to verify and test the parsed text for consistency and coherence. That is, if some constant term is used somewhere, it should be defined in the constants section of the text. At the final point of the parsing process, the vast majority of the contents of the instructions.txt files is used to verify and double-check parsing results.

The defined terms and lists of terms, constants, properties, registers, etc. are all roughly treated as **enumerations**. The values are partly defined in the files and partly guessed by the author, turning out to be correct at the end.

A concise description on how this part of the parser works is not conducive to enhancing understanding. The reader is invited to check out the code.

All of these *instructions.txt* portions are parsed into *Python dicts* and/or *lists* and available as abstraction in the *Python* class *SM_Cu_Details*.

2.6.2 Property Classes

During this work, a lot of details defined by the data in instructions.txt and verified and/or unified over all architectures. The reader should use the two classes **SASS_Class_Props** and **SM_Cu_Props** for properties of single instruction classes and one SM architecture as a whole.

2.6.3 TT Python Classes

The TT classes cover the *format* portion of the instruction classes. An example is available in Listing 2.10.

The entire formalization of the instruction classes formats, including nomenclature of the SASS instructions formats presented so far for example, the split in the operands definitions

- dst/src registers
- list, single and double attribute operands
- $cache bits and constants

is the result of a top-down, multilevel parsing approach of the *instruction class definitions* portion of the *instructions.txt* files. One full example of such a definition is given in the instruction classes Section 2.4.3.

First the instruction class definitions in the text files are parsed into a data structure using one single *token* class **TT_Term** (the *TT* prefix stems from *token*). After this step, all parts of the definitions are contained in a *Python* abstraction that can faithfully reproduce every nuance of the input text while omitting white spaces and indentations.

Parsing into **TT_Term**s was a highly exploratory process since at this point the full extent of the syntax for the instructions was not yet known and consequently *TT_Term* was extended over time to represent all different variations that were encountered.

The result of the first parsing step is one *Python* class that contains and expresses the full syntax variations of **all** instruction classes for Nvidia SASS, starting with SM 50 (Maxwell) up to SM 120 (Blackwell).

The **second** step is splitting up the **TT_Term** abstraction into a more meaningful set of token classes. The following enumeration provides a concise

outline of the most important ones. Note that this outline is in line with the instruction class template provided in Figure 2.1 at the very beginning of this report.

The short descriptions of the *TT* token classes are designed to illustrate the **nodes** of the underlying data structure, not as full documentation. The reader is invited to check out the code for that.

1. **TT_Op**: This token class encompasses all prefix operations as outlined in the prefix operations Section 2.5.11.
2. **TT_Ext**: This token class encompasses all extensions for all operands that have them as outlined in the extensions Section 2.5.12.
3. **TT_Reg**: This token class represents everything that is a register. This includes things such as R11 or UR2, but also the attribute prefixes in operands such as Sc[UImm(0x0)][SImm(0x160)] where Sc is actually defined as a register *register*. Registers are detailed in Section 2.5.3.
4. **TT_Func**: These token classes cover everything that has a value and a bit length, such as SImm(..), UImm(..), F64Imm(..), etc... and also bitmasks such as BITSET(..). Section 2.5.7 details the formal aspects.
5. **TT_Pred**: This is the simplest token class since all predicates look and are defined exactly in the same way. Technically, though a predicate is the same as a register operand since *Predicate* and *UniformPredicate* are register categories. See Section 2.5.1 for more details.
6. **TT_Param**: **TT_Param** is perhaps the most potent token class. It represents one entire operand, no matter if its a register, function or *attribute* operand. It binds all prefix operations registers/functions and extension configurations into one object.
7. **TT_List**: This token class is a simple list abstraction where **all** contained elements are **TT_Param** objects. Note that lists can also have extension registers as showcased in Figure 2.13. List operands (Figure 2.5) are abstracted as **TT_List**. *Single attribute* and *double attribute* operands are abstracted as **TT_Param** containing one register and one or two **TT_List** in its *attrib* property. See Sections 2.5.8, 2.5.9 and 2.5.10 for information on list, single- and double-attribute operands.
8. **TT_Cash**: This token class is distinct from the others and captures the syntax of all $ ((...)) $ cache bit lines.
9. **TT_Instruction**: Every tree-like data structure requires a root node. **TT_Instruction** is it for the syntax abstraction token classes. The reader should note that this is not exactly a tree and also, that there are more TT token classes than the ones presented here. Figure 2.18 depicts which TT token classes can contain which others in a *syntax abstraction diagram*.

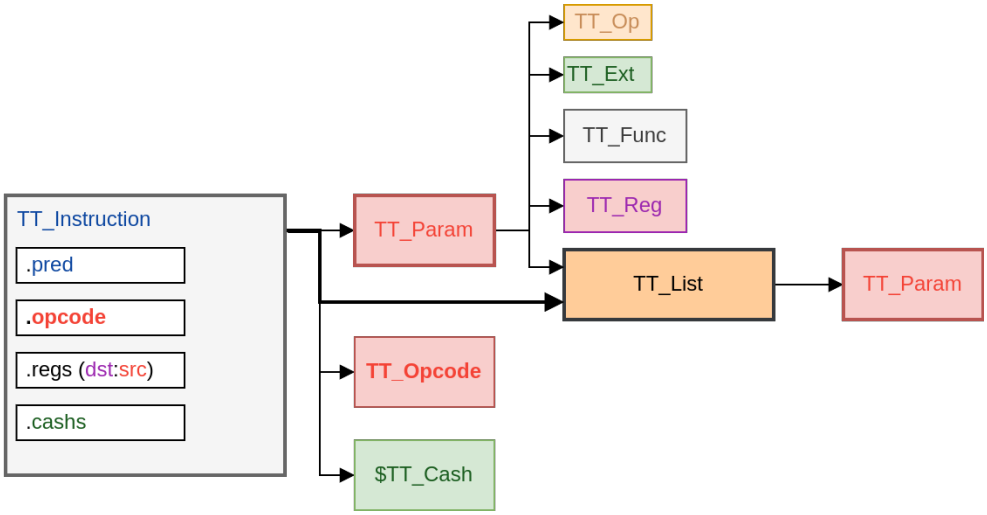


Figure 2.18: Which TT token classes can contain which other ones? The cadence goes roughly from left to right, with the exception of **TT_List** that is contained in **TT_Param** (see Figure 2.5) but also contains **TT_Param** itself (see Figure 2.6 for a **TT_Param** containing one Sc containing two **TT_List**). **TT_Instruction** is the main interface for the user to the parsed instruction class format as presented in Listings 2.28 and 2.10.

TT_Instruction has **four** main access fields:

- .pred: access **TT_Param** representing the predicate
- .**opcode**: access TT_Opcode representing the **Opcode**
- .regs (dst:src): access a list containing **TT_Param** or **TT_List** representing the operands
- .$cashs: access a list containing **TT_Cash** containaing the $cache fields

2.6.4 Fully Capturing SASS Syntax

This section finalizes the part about the data structure that the parser creates by providing some insights on development history context and finally fully captures the full extend of SASS syntax in two images and a few words.

Revisiting our favorite two **STG** variants: Figures 2.5 and 2.6 and augmenting the collection with the **MOV** instruction in Figure 2.4 we examine all the components again. For simplicity, a simplified representation is provided here:

1. PT **STG** [RZ, UR4, SImm(0x0)] R2
2. PT **STG** memoryDescriptor[UR4][R2, SImm(0x0)] R7
3. PT **MOV** R31 SImm(0x0) UImm(0xf)

PT is the predicate in all three instructions. The definition is exactly the same as for register operands. The predicate always has exactly [logical inversion] as prefix operation [!] and **never** any extensions. The only irregularity is that it is sometimes missing. It makes sense to have a separate type for the predicate to simplify testing for integrity. All of these reasons are why the predicate is not a register operand even though, their definitions are the same.

R2, R7 and R31 are register operands while SImm(0x0) and UImm(0xf) are function operands. These two represent mutually distinct concepts.

Originally the token class **TT_Param** held constructs such as

memoryDescriptor[UR4][R2, SImm(0x0)]

using a class called **TT_Accessor**. The textitAccessor part begin the "memoryDescriptor". **TT_Accessor** with the registers and values in the [..] begin the **attributes** of the accessor, describing where and what is accessed. Listing 2.31 shows selective parts of the format definition for the **STG** with the *memoryDescriptor*.

Listing 2.31: Selected parts of the format definition for a memoryDescriptor.

```
CLASS "stg_memdesc__Ra64"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode ...
      ...
      DESC:memoryDescriptor[UniformRegister:Ra_URc]
                           [Register:Ra + SImm(24/0)*:Ra_offset]
      ...
```

Examining Listing 2.31 we see that *memoryDescriptor* accessor is defined as "`DESC:memoryDescriptor`", **DESC** begin a **register** and *memoryDescriptor* its name **alias**. This is exactly the same as a regular register operand definition ("`Register:Ra`" or "`Register:Rc`").

Therefore a *single* and *double* attribute operand is nothing more than a register operand with one or two **TT_List**s added as attributes.

Initially it seems confusing why everything, even lists, can have extensions. For example Figure 2.13 shows a double attribute operator where an extension seems to be added to the **TT_List**. This makes sense, if we look at double attribute operators as register operators with added attributes and always put the extensions at the end.

After this insight, **TT_Accessor**s were removed and replaced in the parsed data structure with a register and one or two lists, accessible as *.attr*.

It now makes sense to re-examine *list* operators too. In fact, a *list* operand is the same as a *single attribute operand* with no register at the beginning. Thus,

list operands can have extensions too. For example **LDGSTS** has *list* operator that looks much the same as the one illustrated for the **STG** instruction in Figure 2.5 with an extension .noexp_desc

The most basic building blocks of SASS instructions are thus:

- **Opcode** with the instruction code
- Registers with well defined names and values
- Functions with well defined sign, numerical format and bit length

Opcode contains a binary code for the instruction and can be configured using extension registers. The operand registers can be augmented with prefixed operations and extended with modifiers, also themselves defined as registers. The functions don't have either prefixed operations or extensions. They represent immediate values, can be signed or unsigned, positive or negative, floats, doubles or integers, depending on which function name is used.

We can now return to Figure 2.1 and by filling in some details realize that all instructions actually do conform to the outlined pattern on the Nvidia website [10].

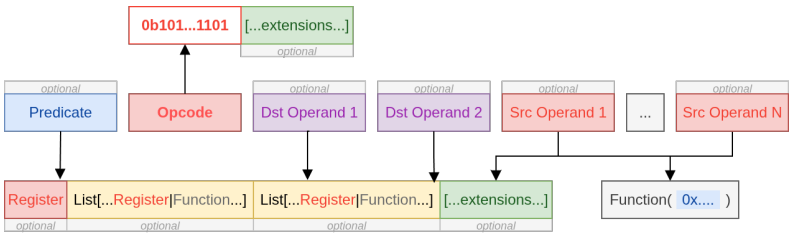


Figure 2.19:

What we have to do for that, is extend the definition for an *instruction operand* to the full **double attribute operand** and make everything but the **Opcode** optional.

2.6.5 SM_SASS and SASS_Class

This section finally highlights the two entry points to access parsed SASS instructions into the developed *Python* framework, how the individual components are organized, and most importantly, how they can be accessed by showcasing some very common programming patterns.

The parsed results for all classes are available in the *Python* class **SM_SASS**. An outline of the structure is in Figure 2.20. **SM_SASS** combines all information related to one SM architecture, for example SM 86. Following are some common Python access patterns.

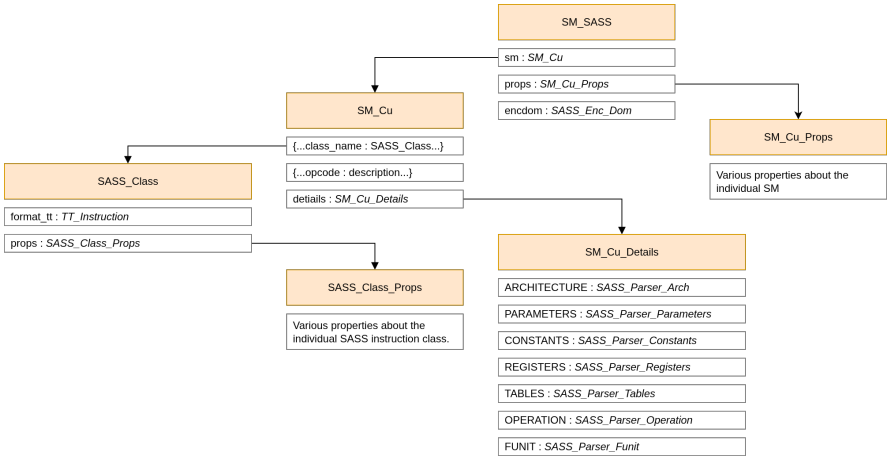


Figure 2.20: Outline of the Parser's architecture. A very common Python pattern to access an individual instruction class, for example the parsed format of instruction **DFMA** in Figure 2.2 is `SASS.sm.classes_dict['dfma__RRC_RRC'].FORMAT`. `RRC` is an indication to which operand types this **DFMA** variant has: Register, Register, Constant (also sometimes I for Immediate).

```
# Load SM 86
sass = SM_SASS(86)

instr_class:SASS_Class = sass.sm.classes_dict['dfma__RRC_RRC']
format_tt:TT_Instruction = instr_class.FORMAT
# print(..) is supported by virtually all subclasses
print(format_tt)
print(format_tt.pred)

# opc will contain the opcode of dfma__RRC_RRC: "DFMA"
opc:str = instr_class.props.opcode
# Access sets containing the naming aliases for the
# operands
dst_names:Set[str] = instr_class.props.dst_names
src_names:Set[str] = instr_class.props.src_names

# sass.sm.details contains the registers in a dictionary as well
# as stored in objects using setattr(..)
all_register_names:List[str] = [i for i in sass.sm.details.REGISTERS_DICT]
all_RXX_registers:List[str] = [i for i in sass.sm.details.REGISTERS.Register]
# The register values are Dict[str: Set[int]] dictionaries
reg_val_pairs = [(register_name, register_value)
                 for register_name, register_value
                    in sass.sm.details.REGISTERS.Register.items()]
```

Chapter 3

SASS Assembler/Disassembler

While the last Chapter 2 was all about the formal definition of SASS instructions, this chapter first details how SASS instruction format information can be evaluated, then how they are encoded, how they can be disassembled from a binary bit vector and finally how it is possible to randomly sample the entire *configuration space* of SASS instructions and even fully enumerate portions of it while keeping other parts fixed.

3.1 Why?

In the previous Chapter 2 we established that scheduling information can be extracted from individual instructions. Note that the Nvidia *cuobjdump -sass* and *nvdisasm* tools also turn Kernels into their SASS instructions but omit a lot of information while doing it:

- *cuobjdump* is tied to an architecture. Only the version for *Blackwell* (SM 100, SM 120) shows information related to barriers and scheduling bits and not even all of it. For all other architectures (including Hopper, at the time of writing), the tool shows the decoded instruction bits only.
- *nvdisasm* can extract the control flow of a kernel, which is also utilized in this work.

The goal is to decode a kernel and draw as much information as possible out of each individual instruction. Since every **Opcode** has many different instruction classes, the output of *cuobjdump -sass* is not enough, since that one completely omits the $cache bits and only decodes down to **Opcode** level, even ignoring many extension registers.

If we are able to decode to individual instruction classes, we can connect the **smallest instruction class unit** to real CUDA kernels and have the possibility to extract the largest amount of information from it with the future goal to

fully *link* microbenchmarked SASS instructions to a decoded CUDA kernel and *literally* count how many cycles the CUDA kernel requires. Of course, a *real* performance model must include things such as *memory latency hiding* and *thread switching* and ideally also cover *coalesced memory accesses*.

The ability to precisely encode and decode SASS instructions and with that entire CUDA kernels (see Section 5.2) as well as *randomly generate* valid SASS instructions (see Section 3.4) opens various ways to precisely microbenchmark and even bulk-generate variable latency SASS instructions by enumerating all their extensions, create one benchmarking kernel per configured variant and measure their cycle count exactly.

In fact, it is even possible to benchmark the barrier mechanism itself (see Section 7.3.4).

3.2 Encoding

This section explains how the operand fields, as defined in the *instruction class format* definition as shown in Listing 2.10 for **DMUL** and explained in abundance in the previous Chapter 2 can be turned into bits using the encoding section of an instruction class definition, for example the one in Listing 2.16 for **DMUL**.

3.2.1 Revisiting the Instruction Class Format

Before we dive into the depths of how instructions are encoded, there is one more distinction that has to be introduced. We do this using the simple sample format definition `Register:Ra`

- *Register category*: the register category in `Register:Ra` is simply *Register*. Every SM defines all their usable registers in the REGISTERS section. Registers are used for register operand definitions as well as extensions. One register category contains multiple values. For example, the category *Register* contains all register designations R1 to R255.
- *Alias*: in `Register:Ra` this refers to *Ra*. In all operand format definitions, the *alias* is the designation after the colon. Perhaps the reader recalls the discussion in the *Fully Capturing SASS Syntax* Section 2.6.4 where we realized that `Register:Ra` and `DESC:memoryDescriptor` are the same thing. In this case, *DESC* is the *register category* and *memoryDescriptor* the *alias*.

Why is the distinction between *register category* and *alias* important? The *alias* names are used to relate the **bit positions** explained in the next Section 3.2.2 with the format definitions of the register and function operands. The extensions sometimes use the *alias* and other times the *register category*.

3.2.2 Bit Positions

The bit positions for all encodings are stored in a section called FUNIT (functional unit). All possible bit ranges that can be encoded are spelled out with a string made of dots and Xes. For example for the operand format definition `[-][||]C:Sb[UImm(5/0*):Sb_bank]*[SImm(17)*:Sb_addr]` there exists one such string named *BITS_14_53_40_Sc_addr* for the *alias* name *Sb_addr*.

```
BITS_14_53_40_Sc_addr '.....dotdotdot.....XXXXXXXXXXXXXX.....dotdotdot.....'
```

The dot-X-strings are 88 characters long for SM 50 up to 62 and 128 characters long for newer SMs and encode the absolute position of every bit.

The keen eye will now realize, that there are only 14 'X'es in the string whereas *Sb_addr* is a 17-bit SImm function operand. Where do the bits get lost? In short, there are a variety of modifications that can be made to the bits before they are encoded into their final resting place. These modifications are explained in the next Sections 3.2.3, 3.2.4 and 3.2.5.

3.2.3 Lookup tables

All lookup tables are defined in the TABLES section of an SM definitions file *instructions.txt*. Listing 3.1 illustrates the concept: the numbers on the left of the → are inputs while the output is located on the right.

Listing 3.1: Example for a lookup table: TABLES_mem_0. Note how for example both triples (2,2,1) and (1,2,1) map to the output 4.

```
TABLES_mem_0
2 2 1 -> 4      1 0 0 -> 0      3 4 0 -> 8      0 2 0 -> 9
1 2 1 -> 4      1 5 1 -> 0      3 1 0 -> 8      0 2 1 -> 11
2 1 1 -> 4      2 2 0 -> 5      3 2 0 -> 8      0 3 0 -> 13
1 1 1 -> 4      2 1 0 -> 5      3 3 0 -> 8      0 3 1 -> 14
2 4 1 -> 6      2 4 0 -> 7      3 5 0 -> 12     0 4 0 -> 15
1 4 1 -> 6      2 3 0 -> 7      0 0 1 -> 1
2 3 1 -> 6      2 5 0 -> 10     0 1 0 -> 2
1 3 1 -> 6      2 5 1 -> 10     0 1 1 -> 3
```

In fact, the lookup table illustrated in Listing 3.1 is used quite often, also in one of our favorite STG instructions in Figure 2.5. The relevant portion of it's format definition is replicated in Listing 3.2

Listing 3.2: Relevant portion of the format definition for STG instruction in Figure 2.5 to go full circle with encoding.

```
CLASS "stg_uniform__RaRZ"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
        ...
```

```
/SEM("WEAK"):sem
/SCO("nosco"):sco
/PRIVATE("noprivate"):private
  ...
```

Depending on which register values are used from the SEM, SCO and PRIVATE register categories, a different value is encoded using the syntax in Listing 3.3.

Listing 3.3: Encode extensions sem, sco and private using lookup table TABLES_mem_0.

```
BITS_4_80_77_mem=TABLES_mem_0(sem,sco,private);
```

Notice that in Listing 3.1, not every mapping is reversible. For example, the triples $(2,2,1)$ and $(1,2,1)$ both map to 4. And so it is with many others. This gives rise to the *decoding universes* further explained in the decoding Section 3.3.

3.2.4 Address Encoding Modifiers

Addresses get special treatment. Usually the process is to use a single or double attribute operand or a list operand to facilitate memory accesses. Since Nvidia GPUs' memory is byte-aligned, there is technically no need for the lowest three bits in an address: 0b100101111 does not exist as a byte-aligned address. We have to use 0b100101000 instead.

There are multiple modifiers used for preprocessing addresses before they are encoded.

ConstBankAddress0

`ConstBankAddress0(bank, addr)` encodes *addr* into one less bit than the length of *addr*.

A typical definition is `C:Sa[UImm(5/0*):bank]*[SImm(17/0)*:addr]` using a signed SImm function with 17 bits. Since addresses are usually not signed, the first bit can be discarded and the value encoded in **16** bits using an encoding definition as showcased in Listing 3.5.

Listing 3.4: Encode constant memory by removing the signed bit with zero right shift before encoding. Note the first nr 16 in the bits description `BITS_16_53_38_Ra_offset`.

```
BITS_5_58_54_Sc_bank, BITS_16_53_38_Ra_offset = ConstBankAddress0(bank, addr);
```

The **suffix 0** for ConstBankAddress**0** indicates that *addr* is shifted to the right by 0 positions before encoding. *ConstBankAddress2* uses the same principle with 2 right-shift positions.

ConstBankAddress2

`ConstBankAddress2(bank, addr)` encodes *addr* into three less bits than the length of *addr*. Assuming *addr* is represented by SImm(17/0) as in the previous section with *ConstBankAddress0*, *ConstBankAddress2* will omit the sign bit and discard the two least significant bits and finally encode only 14 bits.

Listing 3.5: Encode constant memory by removing the signed bit right-shifting the bits by two positions before encoding. Note the first nr 14 in the bits description `BITS_14_53_40_Sc_addr`.

```
BITS_5_58_54_Sc_bank,BITS_14_53_40_Sc_addr =  ConstBankAddress2(bank, addr);
```

Usually fields encoded with *ConstBankAddress2* enforce the two least significant bits to be 0 with a validation condition using a bit mask 0x3 shown in Listing 3.6.

Listing 3.6: Validation condition enforces the two least significant bits of Sb_addr to be 0

```
MISALIGNED_ADDR_ERROR
   (Sb_addr & 0x3) == 0:
      "Constant offsets must be aligned on a 4B boundary"
```

MULTIPLY/SCALE

Not all memory accesses are traditional constant memory accesses. Sometimes there is a pointer. Instruction **SURED** is technically a graphics instruction but serves to highlight the mechanism with its format definition in Listing 3.7. **SURED** (as do all instructions that start with **SU...** work on surfaces. **SURED** performs a "Reduction Op on Surface Memory" [10].

Listing 3.7: Instruction **SURED** contains a memory access using a pointer construct. Note that it is not the * that indicates the pointer but *bank5* and *tsPtrIdx*.

```
CLASS "sured_imm_"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
        ...
      ',' UImm(5)*:bank5
      ',' UImm(16)*:tsPtrIdx
```

tsPtrIdx is encoded using
`BITS_14_53_40_Sc_addr = tsPtrIdx MULTIPLY 4 SCALE 4;` Since *tsPtrIdx* is a UImm(16) function, it has 16 bits. *MULTIPLY 4* first shaves off the **two most significant bits** by left-shifting the value by 2, then *SCALE 4* **right-shifts the resulting value back into place**. The result is a 14 bit value with the two most significant bits set to 0.

MULTIPLY is used in texture instructions only and zeroing the two most significant bits is a peculiar operation. *SCALE*, on the other hand is used in a great many instructions as an alternative to *ConstBankAddress2*.

At this point, we remind the reader that the focus of this work are not the consumer grade GPUs (SM 86, SM 120, etc) but HPC devices (SM 80, SM 90, SM 100, etc). However, due to ease of access of consumer grade GPUs all tests and benchmarks were performed on SM 86 which is the consumer grade Ampere variant. Note, however, that all of this work is directly portable to all SMs supported by the decoder, which are SM 50 to SM 120.

3.2.5 ConvertFloatType

ConvertFloatType is not as much a title as it is a modifier that is able to convert a float data type at the encoding stage based on the configuration of the instruction. The following example utilizes **HFMA2** (matrix fused multiply and add) instruction on SM 86 shown in Listing 3.8 as an example.

Listing 3.8: Format definition for **HFMA2** instruction.

```
CLASS "hfma2__RIR"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode
         /OFMT("F16_V2"):ofmt
         /FMZ("nofmz"):fmz
         /SAT("nosat"):satrelu
      Register:Rd
      ',' [-] [|!] Register:Ra
      ',' F64Imm(64):Sb
      ',' F64Imm(64):Sb1
      ',' [-] [|!] Register:Rc
```

convertFloatType has a function syntax:
`convertFloatType(expr1, val1, .. , exprN, valN, fallbackValue)`.

In **HFMA2**, the function value F64Imm(64):Sb is encoded into 16 bits according to Listing 3.9. `ofmt` encodes the target type as a format expression while `F64Imm(64):Sb` contains the source bits.

Listing 3.9: convertFloatType encoding example.

```
BITS_16_63_48_Sc=Sb convertFloatType(ofmt==`OFMT@E8M7_V2 || ofmt==`OFMT@BF16_V2,
                                     E8M7Imm,
                                     ofmt==`OFMT@E6M9_V2,
                                     E6M9Imm,
                                     F16Imm);
```

If `ofmt==`OFMT@E8M7_V2 || ofmt==`OFMT@BF16_V2` evaluates to *True*, then the format `E8M7Imm` is used. *Elseif* `ofmt==`OFMT@E6M9_V2` is *True* then `E6M9Imm`, *else* it is `F16Imm`.

Note that the actual conversion of the bits was not implemented in this work! Completing the *convertFloatType* encoding modifier with specific conversion implementations is a trivial task and does not require refactoring.

Peculiarity of convertFloatType and MUFU

There are plenty of format specialties also in the encoding stage. For example, the MUFU (multi-functional unit) instruction, that can be configured to compute, among others, high- and low- precision 64 bit reciprocal values, uses **convertFloatType** to select which bits of the result will be encoded. This seems to be one specific case for which the encoding process was adapted. It is documented in the PTX documentation [17]. A similar documentation is available for the RSQ operation [18]. Listing 3.10 showcases one encoding of this.

Listing 3.10: Use convertFloatType to select which bits, resulting from a MUFU operation will be encoded in the instruction. The reader should Note that the result is encoded into a **32** bit location!. Depending on the result of the first argument (True/False), the input *Sb* is kept as 64 bit floating point or converted to a 32 bit variant in the other case. This case can actually be demystified using PTX documentation about reciprocal value and square root calculation [17][18].

```
BITS_32_63_32_Ra_offset=
    Sb convertFloatType(mufuop==`MUFU_OP@RCP64H || mufuop==`MUFU_OP@RSQ64H,
                        F64Imm, F32Imm)
```

3.2.6 Encoding Constant Values

Not all encoded values are part of the instruction format. For example bits related to $RD and $WR barriers for fixed latency instructions are encoded with the constant value 0x7 (see Section 2.5.16). Listings 3.11, 3.12 and 3.13 showcase this concept. Recall, that manually encoding barrier value unequal to 0x7 for instructions that do not feature barriers in their format definitions, will crash the kernel.

Listing 3.11: Encoding for instruction with $RD and $WR barriers

```
BITS_3_115_113_src_rel_sb=VarLatOperandEnc(src_rel_sb);
BITS_3_112_110_dst_wr_sb=VarLatOperandEnc(dst_wr_sb);
```

Listing 3.12: Encoding for instruction with only $RD barrier while $WR is encoded with 0x7

```
BITS_3_115_113_src_rel_sb=VarLatOperandEnc(src_rel_sb);
BITS_3_112_110_dst_wr_sb=7;
```

Listing 3.13: Encoding for a fixed latency instruction with both $RD and $WR barriers are encoded with 0x7. The reader is invited to ignore the '*' characters.

```
BITS_3_115_113_src_rel_sb=*7;
BITS_3_112_110_dst_wr_sb=*7;
```

Newer models use integer values only as constant encoding values. Earlier models use register values as well.

Constant encoding values are visualized by the VSCode extension (see Chapter 4) in *the third row*. The reader can revisit our favorite decoded instruction examples in Figures 2.2, 2.3, 2.5 and 2.6. The values are represented with a leading *C* for Const and two square brackets containing the indices of the bit range where the constant value is encoded. For example

C[57:59]=0x7 or C[51:51]=0x1

Constant encoding values can be used for *decoding* instruction bits. More about that is explained in the decoding Section 3.3.

3.2.7 Full Encoding Definition and The enc_vals Concept

The preceding sections cover the special encoding cases with *modifiers* (Section 3.2.4), *constant values* (Section 3.2.6) and *lookup tables* (Section 3.2.3. This section explains the concept of *enc_vals* and how to use it to fully encode custom SASS instructions.

enc_vals is a dictionary containing **SASS_Bits** representations. To create custom instructions, selected fields in the enc_vals can be changed, for example to change an extension register, invert the predicate or set different destination or source registers.

We use one **DFMA** variant (*dfma__RsIR_RIR*) as an example instruction. The full *Python* code is available in Appendix A.2.2.

First we check out the full *format definition* for this instruction class in Listing 3.14, including the cache bits.

Listing 3.14: ENCODING section of the DFMA variant *dfma__RsIR_RIR*.

```
FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /Round1("RN"):rnd
Register:Rd
',' [-] [||] Register:Ra {/REUSE("noreuse"):reuse_src_a}
','F64Imm(64):Sb
',' [-] [||] Register:Rc {/REUSE("noreuse"):reuse_src_c}
$( { '&' REQ:req '=' BITSET(6/0x0000):req_bit_set } )$
$( { '&' RD:rd '=' UImm(3/0x7):src_rel_sb } )$
$( { '&' WR:wr '=' UImm(3/0x7):dst_wr_sb } )$
$( { '?' USCHED_INFO("DRAIN"):usched_info } )$
$( { '?' BATCH_T("NOP"):batch_t } )$
$( { '?' PM_PRED("PMN"):pm_pred } )$ ;
```

In addition to the frequently cited *format definition*, this time we require the *encoding definition*. The one for the instruction class *dfma__RsIR_RIR* is in Listing 3.15. Provided that an encoding line is not a *constant value* (for example, as shown in Listing 3.13 where the constant value 7 is encoded) or a *register value*, the right side of the equals sign is an operand **alias**. For example, the operand definition `Register:Ra` would be represented in the encoding definition by `Ra` which is the *alias* of `Register:Ra`.

Listing 3.15: ENCODING section of the DFMA variant *dfma__RsIR_RIR*.

```
ENCODING
!dfma__RsIR_RIR_unused;
BITS_3_14_12_Pg = Pg;
BITS_1_15_15_Pg_not = Pg@not;
BITS_13_91_91_11_0_opcode=Opcode;
BITS_2_79_78_stride=rnd;
BITS_8_23_16_Rd=Rd;
BITS_8_31_24_Ra=Ra;
BITS_1_73_73_sz=Ra@absolute;
BITS_1_72_72_e=Ra@negate;
BITS_32_63_32_Ra_offset=Sb;
BITS_8_71_64_Rc=Rc;
BITS_1_74_74_Sc_absolute=Rc@absolute;
BITS_1_75_75_Sc_negate=Rc@negate;
BITS_6_121_116_req_bit_set=req_bit_set;
BITS_3_115_113_src_rel_sb=VarLatOperandEnc(src_rel_sb);
BITS_3_112_110_dst_wr_sb=VarLatOperandEnc(dst_wr_sb);
BITS_2_103_102_pm_pred=pm_pred;
BITS_8_124_122_109_105_opex=
                        TABLES_opex_1(batch_t,usched_info,reuse_src_a,reuse_src_c);
```

Comparing Listings 3.14 and 3.15, we realize that every *alias* in encoding Listing 3.15 is also present in format Listing 3.14. This is why designations like *Pg* or *Pg@not* are called *alias*: they represent some part of an instruction format as placeholders or *aliases*.

The *alias* naming is from an early stage of decoding the instructions sets and is used everywhere and is not a bad designation. A bit more precise is the

following:

- Sometimes an *alias* is needed: there is a *Register:Ra* and *Register:Rc*. Thus we need *Ra* and *Rc* to distinguish these positions because *Register* (the *register category*) is not unique.
- For the prefix operators [-] and [|||] we need an *alias* because we cannot really use "[-]" in the encoding section. All prefix operators use their *parent register alias* (for example *Ra*) and add the performed operation designation after an @ sign. For example, [-] for *Ra* becomes *Ra@negate*. [|||] for *Rc* turns into *Rc@absolute*. In addition to these two operations, there are *@invert* and *@not*. One may recognize the last one from the frequently used *Pg@not* in the instruction predicate.
- The $cache definitions partly feature two *aliases*. The encoding definition **always** uses the last one in the line. For example, in Listing 3.14, the$WR barrier definition has *aliases wr* and *dst_wr_sb*. All encoding definitions would use *dst_wr_sb*.

Naturally there are exceptions to the rule: since extension registers, especially for the **Opcode** are only given once, they may be encoded in the encodings section using their *register category* instead of using their *alias*. for example, the Round1:rnd that is encoded as BITS_2_79_78_stride=rnd; could also be encoded as BITS_2_79_78_stride=Round1;. In the *validation conditions* section (see Listing 2.11), the *alias* is always used, though. This gives rise to a remarkably annoying **register-category/alias duality** that makes it a bit more tricky to evaluate the SASS instruction sets than would be necessary. The Section 3.4, detailing how to build an instruction generator goes into more detail.

For now, we are content to use encoding values as simple *Python dictionary* using a convenience method *sass_bits_from_str* that is also defined in Appendix A.2.2. A sample definition is shown in Code 1. Note that the *enc_vals* dictionary is actually a type [string: SASS_Bits]. `'7U:3b'` is the *string* representation of a **SASS_Bits** object representing a the value *7*, *unsigned* using *3 bits*.

```
enc_vals = {
    # Predicate configuration
    'Pg@not': sass_bits_from_str('0U:1b'),
    'Pg': sass_bits_from_str('7U:3b'),
    # Extensions
    'fmz': sass_bits_from_str('0U:2b'), # keep 0=.nofmz
    'rnd': sass_bits_from_str('0U:2b'), # keep 0=.RN
    'sat': sass_bits_from_str('0U:1b'), # keep 0=.nosat
    # Destination register
    'Rd': sass_bits_from_str('33U:8b'),
    # Source register 1
    'Ra': sass_bits_from_str('36U:8b'),
    'Ra@absolute': sass_bits_from_str('1U:1b'),
    'Ra@negate': sass_bits_from_str('0U:1b'),
    'reuse_src_a': sass_bits_from_str('0U:1b'),
    # Source immediate value
    'Sb': sass_bits_from_str('-1425045178S:32b'),
    # Source register 3
    'Rc': sass_bits_from_str('30U:8b'),
    'Rc@absolute': sass_bits_from_str('1U:1b'),
    'Rc@negate': sass_bits_from_str('1U:1b'),
    'reuse_src_c': sass_bits_from_str('0U:1b'),
    # Cache and barriers
    'req_bit_set': sass_bits_from_str('5U:6b'),
    'dst_wr_sb': sass_bits_from_str('0U:3b'),
    'src_rel_sb': sass_bits_from_str('0U:3b'),
    'usched_info': sass_bits_from_str('1U:5b'),
    # Keep 0
    'pm_pred': sass_bits_from_str('0U:2b'),
    'batch_t': sass_bits_from_str('0U:3b')
}
```

Code 1: Define the encoding values as a *Python dictionary* named *enc_vals*.

Assuming we have suitable *register* values defined, for example `R2`, `PT` or `WAIT15` (see for example Appendix A.5 lines 15 to 44) we can use the enc_vals dictionary defined in Code 1 to selectively overwrite values to create a customized SASS instruction, using another convenience method *overwrite_helper*, also defined in the Appendix A.2.2. A sample overrwrite is shown in Code 2.

```
# Overwrite predicate configuration
enc_vals = overwrite_helper(False, 'Pg@not', enc_vals)
enc_vals = overwrite_helper(PT, 'Pg', enc_vals)
# Overwrite destination register
enc_vals = overwrite_helper(R2, 'Rd', enc_vals)
# Overwrite source register 1
enc_vals = overwrite_helper(R4, 'Ra', enc_vals)
enc_vals = overwrite_helper(False, 'reuse_src_a', enc_vals)
enc_vals = overwrite_helper(True, 'Ra@absolute', enc_vals)
enc_vals = overwrite_helper(False, 'Ra@negate', enc_vals)
# Overwrite source immediate value
enc_vals = overwrite_helper(0x17, 'Sb', enc_vals)
# Overwrite source register 3
enc_vals = overwrite_helper(R20, 'Rc', enc_vals)
enc_vals = overwrite_helper(True, 'Rc@absolute', enc_vals)
enc_vals = overwrite_helper(False, 'Rc@negate', enc_vals)
enc_vals = overwrite_helper(False, 'reuse_src_c', enc_vals)
# Overwrite cache bits and barriers
enc_vals = overwrite_helper(0b000010, 'req_bit_set', enc_vals)
enc_vals = overwrite_helper(0x1, 'dst_wr_sb', enc_vals)
enc_vals = overwrite_helper(0x7, 'src_rel_sb', enc_vals)
enc_vals = overwrite_helper(WAIT15, 'usched_info', enc_vals)
```

Code 2: Override individual fields in the *enc_vals* dictionary defined in Code 1 with new values using the helper method *overwrite_helper* defined in Appendix A.2.2.

After creating our custom instruction encoding, we need to check that the encoding is indeed valid. For example, some instructions only accept registers that are divisible by 2 or even by 4 given certain conditions such as input size that may be 32, 64 or 128 bits. Another common restriction is, that *reuse_src_a/b/c* cannot be set to *True* if we do not use a transX setting for $USCHED_INFO (Note that in the example above, we use WAIT15).

For this we can call *check_expr_conditions* presented in Code 3.

```python
# Check if the new configuration is valid
Instr_CuBin.check_expr_conditions(
    kk_sm.sass.sm.classes_dict[class_name],
    enc_vals, throw=True)
```

Code 3: Python code to check all relevant *validation conditions* using the encoding values in *enc_vals* for the instruction class given by *class_name*.

The next Section 3.2.8 explains the *validation conditions* in detail. The *validation conditions* are used by the convenience method in Code 3 to check the validity of a new encoding.

3.2.8 Instruction Validation Conditions

In this section we examine the conditions an instruction class encoding needs to fulfill to work on a real GPU. For the instruction class *dfma__RsIR_RIR* that we also used in Section 3.2.7, the validation conditions are listed in Listing 3.16. The general concept of *validation conditions* was introduced in Section 2.4.3.

Listing 3.16: Validation CONDITIONS section of the DFMA variant *dfma__RsIR_RIR*.

```
CONDITIONS
OOR_REG_ERROR
 (((Rd)==`Register@RZ)||(((Rd)<=(%MAX_REG_COUNT-2))&&((Rd)!=`Register@R254))):
    "Register Rd is out of range"
MISALIGNED_REG_ERROR
 (((Rd)+((Rd)==`Register@RZ)) % 2) == 0:
    "Register Rd is misaligned"
OOR_REG_ERROR
 (((Ra)==`Register@RZ)||(((Ra)<=(%MAX_REG_COUNT-2))&&((Ra)!=`Register@R254))):
    "Register Ra is out of range"
MISALIGNED_REG_ERROR
 (((Ra)+((Ra)==`Register@RZ)) % 2) == 0:
    "Register Ra is misaligned"
OOR_REG_ERROR
 (((Rc)==`Register@RZ)||(((Rc)<=(%MAX_REG_COUNT-2))&&((Rc)!=`Register@R254))):
    "Register Rc is out of range"
MISALIGNED_REG_ERROR
 (((Rc)+((Rc)==`Register@RZ)) % 2) == 0:
    "Register Rc is misaligned"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 DEFINED TABLES_opex_1(batch_t,usched_info,reuse_src_a,reuse_src_c):
    "Invalid combination of batch_t, usched_info, reuse_src_a, reuse_src_c"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 ((reuse_src_a==1)||(reuse_src_c==1)) ->
    ((usched_info==17)||(usched_info==18)||(usched_info==19)||
     (usched_info==20)||(usched_info==21)||(usched_info==22)||
```

```
      (usched_info==23)||(usched_info==24)||(usched_info==25)||
      (usched_info==26)||(usched_info==27)):
    "?DRAIN and ?WAITn_END_GROUP tokens are not allowed with .reuse"
```

Every condition **must evaluate** to **true**. Recalling the statement about transX and .reuse, the last condition *ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR* expresses exactly that.

```
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 DEFINED TABLES_opex_1(batch_t,usched_info,reuse_src_a,reuse_src_c):
    "Invalid combination of batch_t, usched_info, reuse_src_a, reuse_src_c"
```

Since all the expressions for each condition are parsed into **SASS_Expr** (see Section 2.4.5) and based on Section 3.2.7 we have a full set of encoding values, we can simply evaluate all the conditions expressions using **SASS_Expr**'s evaluation mechanism. If all of them evaluate to *True*, the encoding is valid.

Since *exceptions form the rule*, there is one caveat here as well: not all *aliases* that need *inserting* in all the validation conditions may be contained in an *enc_vals*-set.

There is an issue regarding the encoding values from Section 3.2.7 that wasn't mentioned there: sometimes in a *format definition*, a field uses an *alias* but the field itself is a **dummy** that is later encoded using a **constant value** (see Section 3.2.6) instead of what is defined in the *format definition*. Constant values (as well as constant register values) are not included in the *enc_vals* set. The good thing about this is that *constant value* and *constant register value* encodings are never wrong. But we must ignore the validation conditions that include *aliases* that are not contained in the *enc_vals* dictionary.

All of this is performed by *check_expr_conditions* shown in sample Code 3 in the previous section. In Section 3.4 we will reverse the process and show how to calculate a *randomized sampler* using the *validation conditions* and the *encoding definitions* used in Section 3.2.7.

In this and the previous couple of sections, a *randomized sampler* was assumed to exist. In fact, that is where we got our valid sample of the **DFMA** instruction class *dfma__RsIR_RIR* that we have been using in Code 1. It wouldn't be a pleasant experience to manually evaluate all the validation conditions every time a new instruction encoding is required for a custom CUDA kernel. Section 3.4 explains in detail how the *randomized sampler* is calculated and works.

3.2.9 Instruction to Bits and Bits to Words

Before we head into the decoding Section 3.3, a word about actual bit assembly: this section covers the gap between *enc_vals*, presented in Section 3.2.7 and the final *bit representation* of an instruction that can be injected into a kernel template.

The first step is **assembling** the encoding values presented in Section 3.2.7 into a **bit vector**. The second step is **encoding** the bit vector correctly into words. Since word encoding has changed between SM 62 and SM 70, the SM ranges 50 to 62 and 70 to 120, the second step will be treated separately for these two groups.

enc_vals to Bits Assembly

In Section 2.4.5 we mentioned that **SASS_Expr** has two magic methods *assemble* and *inv*. This section sheds light on the *assemble* method while the decoding Section 3.3 will do the same for the *inv* method.

To showcase the encoding procedure, Code 4 shows very selectively that that **SASS_Expr** only evaluates *itself* on line 10 using the encoding values we pass to it and in fact, it is **SASS_Bits** (Section 2.4.5) that contains the magic bit. Evaluating the **SASS_Expr** using the encoding values hides all complications outlined in Sections 3.2.4, 3.2.3 and 3.2.5.

```
1  class SASS_Expr:
2   ...
3   def assemble(self,
4              enc_vals:typ.Dict,
5              enc_ind:typ.List[typ.Tuple],
6              target_bits:BitVector, sm_nr:int) -> BitVector:
7        ...
8      if self.startswith_opcode():
9          if len(enc_ind) != 1: raise Exception(sp.CONST__ERROR_UNEXPECTED)
10         bits:SASS_Bits = self(enc_vals)
11         target_bits = bits.assemble(target_bits, enc_ind[0], sm_nr)
12       ...
13  ...
```

Code 4: Assembling the Opcode bits using **SASS_Expr**.assemble.

Since the *assemble* method is called on **SASS_Bits** we check out the *assemble* method in Code 5 too where line 23 writes the bits from **SASS_Bits** into the

BitVector. This is done in **reverse**. Meaning, the **highest** *BitVector* index maps to the **least significant bit** of the instruction bits.

A *sanity counter* that throws an exception at 200 is included. Note that up until SM 62, each *BitVector* has a length of **88** and starting with SM 70, the length is **128**. Thus, using 200 as safety net always works.

```
/**
 * Fill instruction bits back-to-front (or maybe Nvidia's front-to-back??)
 */
static BitVector assemble(
        SASS_Bits b,
        BitVector instr_bits,
        const IntVector& enc_inds,
        const int& sm_nr)
{
    if(enc_inds.size() == 0)
        throw std::runtime_error("Illegal: len(enc_inds) > 0 required but \
                                                    [len(enc_inds) == 0]");
    BitVector::const_iterator ib_begin = b._bits.begin();
    BitVector::const_iterator ib = b._bits.end()-1;
    IntVector::const_iterator ie_begin = enc_inds.begin();
    IntVector::const_iterator ie = enc_inds.end()-1;
    int sanity_counter = 0;
    while(true) {
        if(sanity_counter > 200)
            throw std::runtime_error("Unexpected: sanity_counter \
                                                        out of range");
        if(*ie < 0 || *ie >= instr_bits.size())
            throw std::runtime_error("Unexpected: \
                                            *ie < 0 || *ie >= nn.size()");
        instr_bits[*ie] = *ib;
        if(ie == ie_begin || ib == ib_begin) break;
        ie--;
        ib--;
        sanity_counter++;
    };
    return instr_bits;
}
```

Code 5: Assembling a **SASS_Bits** value into a *BitVector*, back-to-front.

To fully assemble an instruction defined with *enc_vals*, we loop over all

encoding expressions and assemble each one into an initially all-zeros *BitVector*. At the end, we can compare the resulting pattern to a bitmask defined for each instruction class. Looking at Listing 3.15, Note the **first** line `!dfma__RsIR_RIR_unused`. It represents a bitmask defined in the *FUNIT* section, depicted approximately in Listing 3.17. It is prefixed with an inverter *"!"*. After a valid assembly of the instruction class *dfma__RsIR_RIR*, every spot marked by an 'X' in the bit mask must still be 0. This opens up a great way to test the encoder, which we will be able to do after we introduce the instruction generator in Section 3.4 using Monte Carlo random sampling.

Listing 3.17: The bit mask dfma__RsIR_RIR_unused can be used to validate an assembled *dfma__RsIR_RIR* instruction. 'X' marks a logical 1, with the '!' inversion, all spots marked by 'X' must be 0 after a valid assembly.

```
'XXX...dotdotdot...X..XXXXXXXXXX.XXXXXXXXXXX..XX...dotdotdot...'
```

SM 50 to 62: Bits to 64-bit Words

Encoding bits to words for SMs 50 to 62 is more complicated than their more modern counterparts starting with SM 70. This section was inspired by[11]. Figure 3.1 illustrates how *cuobjdump -sass* groups SASS instructions in its output. AS it turns out, this is not by accident.

```
code for sm_62
        Function : _Z6kernelPKfPfmf
  .headerflags    @"EF_CUDA_TEXMODE_UNIFIED EF_CUDA_64BIT_ADDRESS EF_CUDA_SM62 EF_CUDA_VIRTUAL_SM(EF_CUDA_SM62)"
                                                        /* 0x003fd800e3e007f6 */
 /*0008*/                   MOV R1, c[0x0][0x20] ;     /* 0x4c98078000870001 */
 /*0010*/                   S2R R2, SR_TID.X ;         /* 0xf0c8000002170002 */
 /*0018*/                   IADD R2.CC, R2, c[0x0][0x8] ;  /* 0x4c10800000270202 */
                                                        /* 0x003fc400064007f6 */
 /*0028*/                   IADD.X R3, RZ, RZ ;        /* 0x5c1008000ff7ff03 */
 /*0030*/                   I2F.F32.U64 R0, R2 ;       /* 0x5cb8000000270e00 */
```

Figure 3.1: Depiction of instructions **MOV**, **S2R** and **IADD** with leading $cache bits word.

Instructions up until SM 62 are encoded in triples. One of those triples is encompassed with a green box in Figure 3.1. However, **one line** only represents 64 bits. One full instruction requires 88 bits. The remaining 24 bits are included inside the orange box that encompasses the top line inside the green box. These bits contain the $cache (see Section 2.28) bits for the following three instructions.

The reader may notice, that $3 \cdot 24 = 72$ but we only have 64 bits available. Each instruction **MOV**, **S2R** and **IADD** get 21 bits, which gives them a sum total of 85 bits where the top 3 bits for **every** instruction in the instruction sets up and including SM 62 are not used.

The 21 bits are split with the following offsets, expressed with *Python* list index rules, meaning, a negative index starts from the back:

- MOV: $cache_bits[-21:]
- S2R: $cache_bits[-42:-21]
- IADD: $cache_bits[-63:-42]

SM 70 to 120: Bits to 64-bit Words

Compared to SMs 62 and older, the encoding scheme on the newer SMs is easy: it is simply all 128 bits split down the middle into two 64 bit words. The only thing that has to be respected is the sequence. As can be seen in Figure 3.2, the instruction bits come first and the $cache bits (the non-decoded portion in *cuobjdump -sass*) are second.

```
code for sm_70
        Function : _Z6kernelPKfPfmf
.headerflags    @"EF_CUDA_TEXMODE_UNIFIED EF_CUDA_64BIT_ADDRESS EF_CUDA_SM70 EF_CUDA_VIRTUAL_SM(EF_CUDA_SM70)"
/*0000*/                   IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28] ;  /* 0x00000a00ff017624 */
                                                                    /* 0x000fe400078e00ff */
/*0010*/              @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ ;           /* 0x000000ffffff389 */
                                                                    /* 0x000fe200000e00ff */
/*0020*/                   S2R R4, SR_TID.X ;                      /* 0x0000000000047919 */
                                                                    /* 0x000e220000002100 */
/*0030*/                   MOV R2, c[0x0][0x168] ;                 /* 0x00005a0000027a02 */
                                                                    /* 0x000fe40000000f00 */
```

Figure 3.2: Depiction of instructions IMAD, SHFL, S2R and MOV. Note that the $cache bits are **after** the bytes defining the actual instruction.

However, in all the *instruction definitions* for SM 70 and upwards, that at this point of this work have been cited to absurdity, the $cache bits are inside the first 64 bits of the instruction. Listing 3.18 shows some of the encoding bit locations for IMAD.

Listing 3.18: Some $cache patterns for the instruction IMAD depicted in Figure 3.2 in the top spot /*0000*/ that is actually the instruction class *imad__RRC_RRC* on SM 70 and at the bottom, the bit pattern for the Opcode location. Note how all bit locations for the $cache bits are in the front and the Opcode in the back of the definitions.

```
BITS_6_121_116_req_bit_set
'......XXXXXX.....................dotdotdot.........................'
BITS_3_115_113_src_rel_sb
'...........XXX...................dotdotdot.........................'
BITS_3_112_110_dst_wr_sb
'..............XXX...............dotdotdot.........................'
BITS_8_124_122_109_105_opex
'...XXX...........XXXXX...........dotdotdot.........................'
BITS_12_11_0_opcode
'................................dotdotdot.............XXXXXXXXXXXX'
```

64-bit Words to Bytes

This transformation only exists for SM 70 and upwards. There is no need to test modified CUDA kernels on SM 62 and below. In fact, it may be hard

to find a GPU featuring one of those architectures. The newest version of CUDA (12.8 at the time of writing) even prints a warning that SM 75 will be deprecated soon. Thus, the transformation from kernel template to binary executable is limited to SM 70 and upwards, since all of them use the same encoding process.

Examining Figure 3.2 see that the $cache bits are located **after** the instructions bits. This is important since we have to reverse the byte order correctly to encode it into a binary. In **every** instruction definition, the $cache bits are **in front** of the remaining instruction bits. For example, in Listing 3.18, the $cache bit locations for (from the top down) $REQ, $RD, $WR and $USCHED_INFO are depicted. The bottom one depicts the **Opcode** that, as the 'X'-spots indicate, is located on the far right side of the encoding pattern.

To showcase precisely how to encode a 128 bit instruction, as it is encoded by the SASS definition, we pick a specific variant of **IMAD** and do some *Python* conversions with it.

```
# Full BitVector, split in two 64 bit sections exactly in the middle
# into cache bits (c_bits) and instruction bits (i_bits)
c_bits = [0000000000001111111001000000000000001111000111000000000011111111]
i_bits = [0000000000000000000010100000000011111111000000010111011000100100]
```

Code 6: `c_bits` and `i_bits` are BitVectors that encode the $cache and *instruction* portions of an **IMAD** instruction respectively.

Examining the `c_bits` precisely and comparing it to Listing 3.18 reveals

- leading 12 zeros: *000000000000*, meaning `BITS_6_121_116_req_bit_set` is all zeros corresponding to $REQ={0x0}
- 6 consecutive ones: *111111*, meaning `BITS_3_115_113_src_rel_sb` and `BITS_3_112_110_dst_wr_sb` are *111* = 0x7 respectively, which means, $WR=0x7 and $RD=0x7

Doing the same for `i_bits` reveals that the **Opcode** binary code is *011000100100*. The *defined* opcode for the specific **IMAD** instruction we used is *0b11000100100*. This is a match up to a leading zero. Recall that Section 2.5.2 mentioned that the binary codes for instructions may be padded with zeros in actual encoded instructions. This is such a case.

Continuing the encoding process, both `c_bits` and `i_bits` are packed into 8 bit packages and then converted to hex in reversed order. Conversion to *hex* is for convenience while reversing the order is not! Code 7 shows this process with an example.

```
# Input clarified
c_bits = [00000000 00001111 11100100 00000000
          00000111 10001110 00000000 11111111]
i_bits = [00000000 00000000 00001010 00000000
          11111111 00000001 01110110 00100100]

# Using these list conversions...
c_bytes = ['0b' + "".join(str(b) for b in c_bits[i:(i+8)])
                                        for i in range(0,len(c_bits),8)]
i_bytes = ['0b' + "".join(str(b) for b in i_bits[i:(i+8)])
                                        for i in range(0,len(i_bits),8)]

# ... we get the following for c_bytes and i_bytes
c_bytes = ['0b00000000', '0b00001111', '0b11100100', '0b00000000',
           '0b00000111', '0b10001110', '0b00000000', '0b11111111'],

i_bytes = ['0b00000000', '0b00000000', '0b00001010', '0b00000000',
           '0b11111111', '0b00000001', '0b01110110', '0b00100100']

# Then converting to hex in **reversed** order...
c_hex = [hex(int(i,2))[2:].zfill(2) for i in reversed(c_bytes)]
i_hex = [hex(int(i,2))[2:].zfill(2) for i in reversed(i_bytes)]

# ... results in two hex values lists that we can finally use
c_hex = ['ff', '00', '8e', '07', '00', 'e4', '0f', '00']
i_hex = ['24', '76', '01', 'ff', '00', '0a', '00', '00'],
```

Code 7: Using Code 6 as an input, we transform the BitVectors.

Joining `c_hex` and `i_hex` from Code 7 into a *string* by also reversing their byte order and prefix them with *'0x'* yields the same numbers that we can see for the **IMAD** instruction in Figure 3.2:

- /* 0x000fe400078e00ff */
- /* 0x00000a00ff017624 */

Note that reversing the byte order is only necessary to compare to the output of *cuobjtump -sass*. We can check with a tool named **smd** that is part of the *py_cubin* module of Section 5.2, that at the offsets **0x800** and **0x808**, the CUDA binary shown in Figure 3.2 contains the following bytes:

```
0x800: 24 76 01 FF 00 0A 00 00
0x808: FF 00 8E 07 00 E4 0F 00
```

Therefore, the only thing left to do, is to convert the hex lists of Code 7 into real bytes and store them at the correct offsets in the CUDA binary file: 0x800 contains `i_hex` and 0x808 contains `c_hex`.

```
1 # Using Python's bytearray.fromhex to get a binary string...
2 c_cubin = bytearray.fromhex("".join(c_hex))
3 i_cubin = bytearray.fromhex("".join(i_hex))
4
5 # ... yields the final product
6 c_cubin = bytearray(b'$v\x01\xff\x00\n\x00\x00')          # to 0x808
7 i_cubin = bytearray(b'\xff\x00\x8e\x07\x00\xe4\x0f\x00') # to 0x800
```

While this section takes an instruction class from its formal definition to finished bytes, ready to go into a CUDA binary, the Cubin Chapter 5 will explain how to find the CUDA binary inside a C++ binary and how to find the individual CUDA binary components and modify them.

3.2.10 One More Thing

It's not over before it's over! Nvidia knows many ways of invalidating things without removing them. For example by encoding an operand named *Sb* with zero bits: `BITS_0_Sb=Sb;`

3.3 Decoding

While Section 3.2 shows how to encode a formal instruction class into finished bytes, this section shows how to *decode* bytes into an *instruction class*. Especially, this section shows how **SASS_Expr**.*inv(...)* mentioned in Section 2.4.5 works and how an individual *instruction class* can be identified from two 64 bit words taken from a real CUDA binary using its *finterprint*. This section assumes that the instruction bytes are available. The Cubin Chapter 5 will explain how to find them.

3.3.1 Inverse Universes

While all instruction classes (that are not ALTERNATES, see Section 2.4.2) can be uniquely isolated by the decoding process (as we will see in Section 3.3.3) , some encoded instructions can't be uniquely reversed to one unique configuration. Thus, reality is split into multiple universes that are indistinguishable

by their bits but feature different configurations. For example, multiple different extension register configurations may map to the same bit encoding. Thus, to decode the process and be thorough, because we do not know ahead of time which information can be discarded and which one cannot, we must distinguish all configurations.

In the same style as in Section 3.2.9 in Code 4, where we showcased the least difficult part of the *assemble* functionality of **SASS_Expr** (see Section 2.4.5), in this section we show the most cumbersome portion of **SASS_Expr**'s *inv*ersion functionality.

First, though, recall STG in Figure 2.6 where we can see two versions of the same instruction decoded, dubbed as `[U0]` and `[U1]`. Note that Figure 2.6 features a link back to this place for convenience. Figures 3.3 and 3.4 show how the STG in Figure 2.6 is decoded and **exactly** how the individual bits are inverted. The reader may recall Section 3.2.3 where we showed that not all table lookups are invertible. This is one instance of this. We can see that table **TABLES_mem_0** using sco, sem and private as arguments maps to the **same** value using both triples .SYS.WEAK.PRIVATE and .nosco.WEAK.noprivate. What the individual extensions do is not documented directly for SASS, though a lot have *similar*, documented configurations for PTX [8]. In this case, the precise meaning of the extensions is not relevant, only that both configurations map to the same behavior on the GPU.

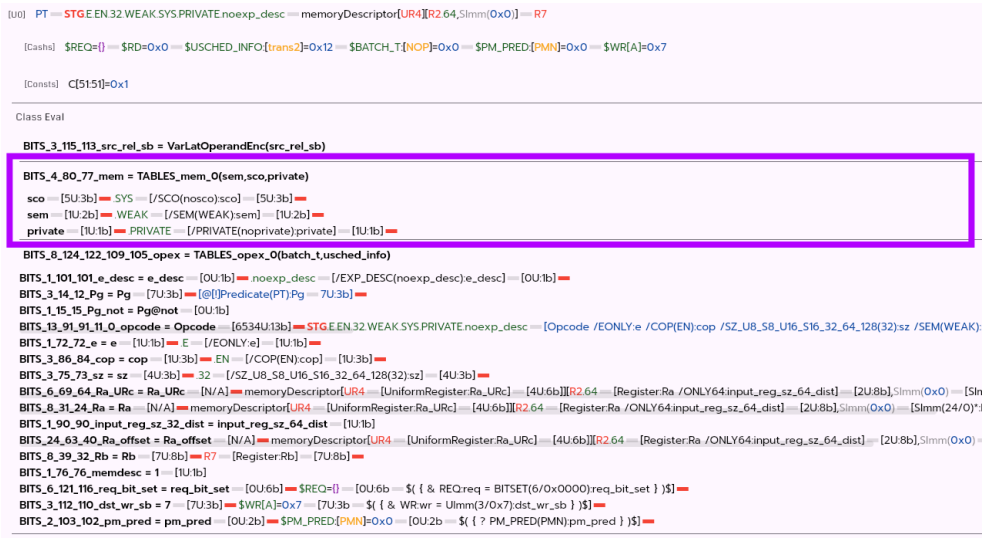


Figure 3.3: Universe **[U0]** for STG instruction in Figure 2.6. The **purple** framed evaluated field is responsible for the multiple evaluation possibilities: the lookup to **TABLES_mem_0** using sco, sem and private is not uniquely invertible.

The **first** step in inverting an encoding is to reverse the steps outlined in the

Figure 3.4: Universe **[U1]** for STG instruction in Figure 2.6. The purple framed evaluated field is responsible for the multiple evaluation possibilities: the lookup to **TABLES_mem_0** using sco, sem and private is not uniquely invertible.

previous Section 3.2. Since this is essentially just do the same in reverse, we omit showing this step here again. The reader may imagine a 128 bit long *BitVector* as shown in Listing 3.19 as input.

SM 50 to SM 62

For these older models, we must decode triples of instructions encoded as four 64 bit words. Thus, for these SMs, we get three 88 bit long *BitVectors* to decode at a given time.

SM 70 and newer

This, more interesting, category of SMs decodes one instruction at the time, consisting of two 64 bit words each, turning into one 128 bit long *BitVector* (see Listing 3.19)

Listing 3.19: One 128 bit long BitVector to decode for SM70: this is the IMAD instruction used in Section 3.2.9.

```
bit_vector =
  [00000000000011111111001000000000000000011110001110000000001111111
   00000000000000000000101000000000111111110000000101110110001001 00]
```

Using the *BitVector* representation, the **second** step is explained in detail in the next Section 3.3.3.

```
def inv(self,
        details:SM_Cu_Details,
        instr_bits:BitVector,
        enc_ind:typ.List[typ.Tuple],
        enc_vals:typ.Dict,
        code_name:str) -> SASS_Expr_Dec|typ.List[SASS_Expr_Dec]:
    ...
        elif self.startswith_table():
            if len(enc_ind) != 1: raise Exception(sp.CONST__ERROR_UNEXPECTED)
            val = SASS_Expr_Dec.get_bdh(instr_bits, enc_ind[0])
            inv_table = details.TABLES_INV_DICT[str(self.get_first_op())]
            # Get table values with bit precision
            # For example, there is one entry in table 'VarLatOperandEnc' ==
            #   {
            #     0: [(0,)], 1: [(1,)], 2: [(2,)], 3: [(3,)], 4: [(4,)],
            #     5: [(5,)], 6: [(6,)], 7: [(7,), (65535,)]
            #   }
            # where the invers entry for 7 maps to 7 and 65535. With 3
            # encoding bits, 7 and 65535 are the same, though (0b111)
            # => make such entries be the same in the result
            # => use sets and stuff
            t_val_variant = {tuple(bin(j)[2:][-len(enc_ind[0]):]
                                   for j in i) for i in inv_table[val['d']]}

            # Map values back to params of table call
            t_args = [(i['i'], i['a'], i['v'])
                                   for i in self.get_table_args()['tt']]
            res = []
            for t_vals in t_val_variant:
                sub_res = []
                for arg in t_args:
                    val_b = t_vals[arg[0]]
                    # Case distinction into the different operand types
                    if isinstance(arg[2], TT_Func):
                        sub_res.append(SASS_Expr_Dec.func_from_val(
                                       arg[2], val_b, enc_ind[0], code_name))
                    elif isinstance(arg[2], TT_Reg):
                        sub_res.append(SASS_Expr_Dec.reg_from_val(
                                       arg[2], val_b, enc_ind[0], code_name))
                    elif any(isinstance(arg[2], t) for t in AT_OP.values()):
                        # AT_OP is for example Pg@not (AT == @)
                        sub_res.append(SASS_Expr_Dec.op_from_val(
                                       arg[2], val_b, enc_ind[0], code_name))
                    else: raise Exception(sp.CONST__ERROR_UNEXPECTED)
                res.append(sub_res)
            return res
    ...
```


Code 8: Inverting a table lookup from encoded bits to formal values using **SASS_Expr**.*inv(...)*.

In this section we showcase how the mechanism to invert bits to their formal definition using the example of table lookup (Section 3.2.3).

Code 8 shows the relevant portion of the invert method. We can see, that apart from not being trivial, there is a **case distinction** using SASS_Expr_Dec classes. SASS_Expr_Dec is in many ways the *decoding equivalent* for the TT_Term class introduced in Section 2.6.3. Recall that TT_Term was used in the initial *formalization* step to capture the entire syntax-space for all SASS instruction set dialects before begin split up into more specialized TT-classes (for example TT_Pred, TT_Param and TT_Cash).

SASS_Expr_Dec captures the entire range of parsed *possibilities* just adding new things as they show up. This indirect approach is very useful at this stage as well, because it allows to capture the entire space of values that needs to be reversed into formal instruction definitions before the requirement to come up with an actual, more precise class structure.

Note that SASS_Expr_Dec as well as the instruction class fingerprint, introduced in the next Section 3.3.3 are part of the *py_sass* module outlined in Section 2.6 while the *encoding* and *decoding* actual bytes procedures are part of the *py_cubin* module that will be outlined in Chapter 5.

After this step, SASS_Expr_Dec will represent the *value* of the group of classes that a potential user will interact with directly, similarly to the token *TT* classes outlined in Section 2.6.3. Thus it is very useful to understand namings and approximate intention. The Instr_Cubin_Repr classes will be outlined in the next Section 3.3.2.

3.3.2 Instr_Cubin_Repr and Attachments

This section introduces a group of classes that a potential user of the provided analysis framework will directly interact with to deal with decoded bits and bytes. One usecase is: decode one instruction and depending on the operands of the decoded instruction, create different kind of custom content. This usecase is demonstrated in the benchmarking Section 7.3.5.

Incidentally, these classes are also exactly the ones represented in the evaluation figures for STG, Figures 3.3 and 3.4 and every other figure showing decoded instructions using the VSCode extension from Chapter 4. Meaning, one of these classes is the STG, another one represents .32 and yet another one represents the *double attribute operand* (see Section 2.5.7), etc.

- Instr_Cubin_Repr: this is the representation of one single decoded SASS instruction. As such, this is the equivalent *decoding* class to the parser class TT_Instruction, introduced in Section 2.4. The most notable distinction between the two concepts are, that one Instr_Cubin_Repr contains *multiple universes* decoded from one set of bits as presented in

Listing 3.19 while the **TT_Instruction** only contains one single, parsed structure formalization.

- **Instr_CuBin_Ext**: represents one decoded extension register. The equivalent to **TT_Ext**.
- **Instr_CuBin_Op**: represents one decoded [prefix operation]. The equivalent to **TT_Op**.
- **Instr_CuBin_Opcode**: represents one decoded **Opcode**. The equivalent to **TT_Opcode**.
- **Instr_CuBin_Pred**: represents one decoded predicate. The equivalent to **TT_Pred**.
- **Instr_CuBin_Param_RF**: represents one decoded destination or source register or function operand. The equivalents are **TT_Param** containing either **TT_Reg** or **TT_Func**.
- **Instr_CuBin_Param_Attr**: represents one decoded destination or source single or double attribute operand. Corresponds to **TT_Param** containing one or two attributes.
- **Instr_CuBin_Param_L**: represents one **TT_List** operand.
- **Instr_CuBin_Const**: represents one decoded constant value, the equivalent to **TT_Const**.
- **Instr_CuBin_Cash**: contains one **TT_Cash** field.
- **Instr_CuBin_Misc**: everything that doesn't go anywhere else. This one does not have an equivalent *TT* class. In every decoded instruction, for example the one in Figure 2.4, there is a field containing offset numbers: [13, 0x17d58, 0x7d0, 0xd0]. These numbers are the instruction number, the offset from the start of the C++ binary, the offset from the start of the Cubin binary and finally the offset from the start of the instructions part of the Cubin binary. These offsets will be highlighted with an illustration in the Cubin Chapter 5, including the distinction between *C++ binary* and *CUDA binary*..
- **Instr_CuBin_Eval**: this one contains everything that is shown in the *Eval* section of the decoder. Figures 3.3 and 3.4 are two examples. This is by far the most complex one of these classes.

3.3.3 Instruction Class Fingerprint

Finally, this section explains in detail how a **fingerprint** can be calculated for every **instruction class** such that each one can individually be identified from a BitVector as presented in Listing 3.19.

There are several mechanisms at play. Two of them will be demonstrated with two groups of example instructions on SM 50, resulting into an algorithm that generalizes the style of mechanisms into a lookup table generator that can then be used by the decoder for all SMs. In fact, the older SMs are more suitable for this kind of development because they seem much less organized than their newer counterparts. For example SM 50 features one group of 30 instruction classes that share the same binary operation code but not even all have the same actual instruction **Opcode**.

Remembering the ALTERNATE classes concept explained in Section 2.4.2, recall that their encoded versions are indistinguishable from their main *instruction class parent*. Thus the decoder will always default to the main *instruction class*. This gives rise to funny moments when after encoding a custom **IMUL** instruction, the VSCode visualization from Chapter 4 shows an **IMAD** instruction.

F2F: float-to-float conversion

Since there are a great many ways to convert floats into other floats because they feature an increasing number of formats as the architectures evolve, there are a great many instruction classes for **F2F** too. A selection of seven **F2F** classes is shown in Figure 3.20. Irrelevant portions of the class definitions are omitted for clarity.

Listing 3.20: A selection of seven **F2F** classes that all share the **same binary Opcode**. Irrelevant portions of the class definitions are omitted for clarity.

```
CLASS "Imm_F2F_2_16"
   Opcode ... /F2Ffmts2_16:fmts /F2FRound2(RN):rnd ...
CLASS "Imm_F2F_2_64"
   Opcode ... /F2Ffmts2_64:fmts /F2FRound2(RN):rnd ...
CLASS "Imm_F2F_2_64_32"
   Opcode ... /F2Ffmts2_64_32:fmts /F2FRound2(RN):rnd ...
CLASS "Imm_F2F_2_64_16"
   Opcode ... /F2Ffmts2_64_16:fmts /F2FRound2(RN):rnd ...
CLASS "Imm_F2F_1_16"
   Opcode ... /F2Ffmts1_16:fmts /F2FRound1(PASS):rnd ...
CLASS "Imm_F2F_1"
   Opcode ... /F2Ffmts1(F32.F32/PRINT):fmts /F2FRound1(PASS):rnd ...
CLASS "Imm_F2F_1_64"
   Opcode ... /F2Ffmts1_64:fmts /F2FRound1(PASS):rnd ...
```

We notice that one extension register .fmts, denoting the resulting format of the input, is different for every instruction class. Listing 3.21 verifies this insight. In fact, using extension registers not only configures a given **Opcode** instruction, it also contributes to its unique fingerprint. Intuitively this makes sense, since every configuration should take a unique data path trough the GPU. In practice, **Opcode** and extensions are not sufficient.

Listing 3.21: Isolating the .fmt register values from Listing 3.20 that within the presented group of F2F instructions, all are uniquely identified by their combination of .fmts.

```
F2Ffmts2_16 {'F32.F16': {6}}
F2Ffmts2_64 {'F16.F64': {13}, 'F32.F64': {14}}
F2Ffmts2_64_32 {'F64.F32': {11}}
F2Ffmts2_64_16 {'F64.F16': {7}}
F2Ffmts1_16 {'F16.F16': {5}}
F2Ffmts1 {'F32.F32': {10}}
F2Ffmts1_64 {'F64.F64': {15}}
```

BAR: SM 50's barrier instructions

Listing 3.22 shows four different portions of encoding sections of four different *instruction classes* for the BAR instruction. Note the *constant value* encodings `AFix_BAR` and `BFix_BAR` that enumerate 0 to 3 in t bit binary code. In this case, these encoding bits can clearly be used to distinguish the four *instruction classes*.

Listing 3.22: Four BAR instructions on SM 50 that are distinguishable using *constant* encoding values, represented in a simplified maner to highlight the distinctions: the different incarnations of BAR are *enumerated* from 0 to 3 using constant encoding values.

```
class "BAR_Arv"
  FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /BarArv:barmode ...
ENCODING
  AFix_BAR = 0
  BFix_BAR = 0
    ...
class "BAR_Arv_a"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /BarArv:barmode ...
ENCODING
  AFix_BAR = 1
  BFix_BAR = 0
    ...
class "BAR_Arv_imm"
  FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /BarArv:barmode ...
ENCODING
  AFix_BAR = 0
  BFix_BAR = 1
    ...
class "BAR_Arv_imm_a"
  FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode /BarArv:barmode ...
ENCODING
  AFix_BAR = 1
  BFix_BAR = 1
    ...
```

Expanding to a Full Algorithm

In total we can use the following four categories.:

- Extension registers: some instructions have the same **Opcode** for multiple specialized instruction classes for different bit sizes or formats.
- **Opcode** binary code: this one distinguishes between families of instruction classes that do roughly the same but differently. One example may be various **F2F**, **I2F** or **F2I** instructions.
- Constant encoding values: some instructions feature fixed, constant values in their encoding that do not show up in the format specifications and are distinct between different instruction classes of the same family, for example the previously highlighted **BAR** instructions on SM 50. Especially on older SMs, *register values* may also be used directly in the encoding section. For example `Bop = `Bop@AND`. This practice seems to be discontinued with the newest SMs. Constant encoding values belonging to $cache bits are ignored.
- Zero/NonZeroRegister operands: some instruction classes feature two variants where one variant specifically uses `ZeroRegister={255}` and the other `NonZeroRegister={0,1,...,254}` as operand register types. These are two non-overlapping sets. If the encoding field for this operand is 255 it is one variant, otherwise the other one. This includes `ZeroUniformRegister` and `NonZeroUniformRegister`, `NonZeroRegisterFAU` and `Predicate_vimnmx` too.

These conditions are encoded as *lookup table* as shown in Figure 3.3.3. Note that each entry for a set of bit indices maps to a **set of tuples**. In *Python*, tuples can be hashed and used in sets.

```
lookup_table[class_name] = {
   encoding_indices_1 : {
      tuple(possible set of bits 1), ..., tuple(possible set of bits N)
   },
    ...
   encoding_indices_N : {
      tuple(possible set of bits 1), ..., tuple(possible set of bits N)
   }
}
```

We do the following two things:

1. For every instruction class check all four required categories.
2. For every matching category store the indices of the bits for the encoding field and the expected value.

To illustrate the algorithm we use a third example that utilizes the ZeroRegister/NonZeroRegister mechanism (in addition to constant values and extensions shown previously with **BAR** and **F2F** instructions). We choose this one because it is probably the most interesting idea on how to distinguish instructions. Listings 3.24 and 3.23 show the relevant portions of their *format definitions* (omitting extensions, $cache bits, etc.).

Listing 3.23: **LD** instruction using UImm(32) operand.

```
CLASS "ld__uImmOffset"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode ...
   Register:Rd
   ',' [ ZeroRegister("RZ"):Ra + UImm(32/0)*:Ra_offset ]
   ',' [!]Predicate("PT"):Pnz
```

Listing 3.24: **LD** instruction using SImm(32) operand.

```
CLASS "ld__sImmOffset"
   FORMAT PREDICATE @[!]Predicate(PT):Pg Opcode ...
      Register:Rd
      ',' [ NonZeroRegister:Ra + SImm(32/0)*:Ra_offset ]
      ',' [!]Predicate("PT"):Pnz
```

In Listing 3.23, the memory access utilizes an UImm(32) offset while the other one in Listing 3.24 utilizes an SImm(32). The Operand Ra for both instructions is encoded in 8 bits. Assuming, these bits are indexed (31, 32, ..., 38), the lookup table will get the following entries:

```
1 # Instruction class ld__uImmOffset gets 1 additional entry
2 lookup_table["ld__uImmOffset"][(31, 32, ..., 38)].add( (1,1,1,1,1,1,1,1) )
3 # Instruction class ld__sImmOffset gets 254 additional entries
4 lookup_table["ld__sImmOffset"][(31, 32, ..., 38)].add( (0,0,0,0,0,0,0,0) )
5  ...
6 lookup_table["ld__sImmOffset"][(31, 32, ..., 38)].add( (1,1,1,1,1,1,1,0) )
```

The lookup table can be calculated offline and is stored in **SM_SASS**. Chapter 4 shows how the decoder used in this section is visualized using a VSCode extension based on *Flutter* [19].

3.4 Instruction Generator

3.4.1 Introduction

After thoroughly exploring the encoding and decoding mechanisms in the previous Sections of this chapter as well as the precise semantics of SASS

instructions in Chapter 2, let's first introduce the term **configuration space** of an instruction class. The **configuration space** is the set of all possible, **valid** configurations using all extensions, destination and source operands.

We envision a generator that can be queried in the following ways:

- Uniformly at random sample the entire *configuration space* for an instruction given its instruction class name.
- Given an instruction class name and a subset of encoding values as *anker points*, where each encoding value has a set of possibilities, allow iterating all **valid** possibilities for the given instruction and the given anker encoding values by *filling in* the blanks *uniformly at random*.

Such a complete SASS instruction set representation will enable exhaustive testing of the decoder and the encoder in a closed-loop setting: generate randomized instructions using the generator, encoding then decoding them and check if the instruction classes match.

Sidenote: without this configuration space representation, building the encoder, decoder and the method to create customized CUDA kernels for benchmarking would not have been possible.

3.4.2 Validation Conditions, Evaluation and Statistics

First, for convenience, let's revisit the validation conditions section for the DMUL variant used in Section 2.4's Listing 2.11 in Listing 3.25.

Listing 3.25: Conditions definition for instruction class dmul_RCR_RC which is a variant for DMUL.

```
CONDITIONS
OOR_REG_ERROR
 (((Rd)==`Register@RZ)||(((Rd)<=(%MAX_REG_COUNT-2))&&((Rd)!=`Register@R254))) :
 "Register Rd is out of range"
MISALIGNED_REG_ERROR
 (((Rd)+((Rd)==`Register@RZ)) % 2) == 0 :
 "Register Rd is misaligned"
OOR_REG_ERROR
 (((Ra)==`Register@RZ)||(((Ra)<=(%MAX_REG_COUNT-2))&&((Ra)!=`Register@R254))) :
 "Register Ra is out of range"
MISALIGNED_REG_ERROR
 (((Ra)+((Ra)==`Register@RZ)) % 2) == 0 :
 "Register Ra is misaligned"
INVALID_CONST_ADDR_SASS_ONLY_ERROR
 ((Sb_bank <= 17) || (Sb_bank >= 24 && Sb_bank <= 31)) :
 "Invalid constant bank error"
MISALIGNED_ADDR_ERROR
 (Sb_addr & 0x7) == 0 :
 "Constant offsets must be aligned on a 8B boundary"
INVALID_CONST_ADDR_ERROR
 (Sb_bank >= 24 && Sb_bank <= 31) -> (Sb_addr <= 255) :
```

```
  "RTV banks may not use an offset greater than 255"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 (%SHADER_TYPE == $ST_CS) -> !(Sb_bank >= 8 && Sb_bank <= 31) :
 "CS may not use RTV banks"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 DEFINED TABLES_opex_2(batch_t,usched_info,reuse_src_a) :
 "Invalid combination of batch_t, usched_info, reuse_src_a"
ILLEGAL_INSTR_ENCODING_SASS_ONLY_ERROR
 ((reuse_src_a==1)) -> ((usched_info==17)||(usched_info==18)||(usched_info==19)
          ||(usched_info==20)||(usched_info==21)||(usched_info==22)
          ||(usched_info==23)||(usched_info==24)||(usched_info==25)
          ||(usched_info==26)||(usched_info==27)) :
 "?DRAIN and ?WAITn_END_GROUP tokens are not allowed with .reuse"
```

Reducing the Input Size

What we should do to create the full *configuration space representation* is solve all of the validation conditions for all possible inputs. At first glance, this seems to be feasible. However:

- There are instruction classes that possess 300 to 400 individual validation condition expressions.
- Some validation conditions take multiple registers (for example Ra, Rb) of the *Register* category as input. Recalling Section 2.3, the *Register* category has 255 different values. Having two of them as inputs would yield $255 \cdot 255$ combinations to test, not even mentioning all other inputs or instructions with three register operands.

Thus, calculating the entire configuration space is not feasible. Also, it is not necessary: the validation conditions don't do magic, unexpected things. For example `(((Rd)+((Rd)==`Register@RZ)) %2) == 0` simply ensures that `Rd` is even numbered or RZ (see Section 2.4.3 for a more precise examination of this expression). For all *Register*, *UniformRegister* and more like them, it is sufficient to use a sufficiently large subset, always including the *zero* representation (for example RZ or URZ) and enough values to, for example only use the ones divisible by 4 and not get a result set that is too small for the remaining calculations.

Accelerating Expression Evaluation

At this point, we could try to evaluate all validation condition expression for a reduced input size. However, there still are 10'000ds of expressions to validate, some of which take multiple seconds to fully evaluate, due to the recursive, non performance focused way the **SASS_Expr** works.

Inspecting Listing 3.25 and many more validation condition sections for many more instruction classes, reveals that there seems to be some measure

of uniformity for the conditions. For example `(Sb_addr & 0x7) == 0` occurs often, so do all the expressions limiting Ra, Rb, Rc, etc. to even numbered registers.

Exhaustive statistics show that there are indeed validation condition expressions that occur many more times than others. Accelerating solving those will go a long way in making the calculations feasible. For SM 50 to SM 120, the total number of condition expressions (that do not contain functions like `(Sb_addr & 0x7) == 0` where *Sb_addr* is an SImm(17)) to evaluate is 118′416. Listing 3.26 shows the four most common expressions, that together occur 43′786 times, which is almost 50%. Also, the validation condition `(((Rd)+((Rd)==`Register@RZ)) %2) == 0` used as example before is the 4th most common validation condition.

Listing 3.26: The four most common validation condition expressions.

```
Nr    | Expression
------|-------------------------------------------------------------
17110 | (((Rd) == `Register@RZ) || (((Rd) <= (%MAX_REG_COUNT - 1))
      |                                && ((Rd) != `Register@R254)))
------|-------------------------------------------------------------
12196 |  (((Rd) == `Register@RZ) || ((Rd) <= (%MAX_REG_COUNT - 1)))
------|-------------------------------------------------------------
10322 | DEFINED TABLES_opex_0(batch_t,usched_info)
------|-------------------------------------------------------------
4158  | (((Rb) + ((Rb) == `Register@RZ)) % 2) == 0
```

The statistics in Listing 3.26 do not just contain the expressions shown there, but all expressions that fit the respective pattern. For example,
`(((Rb) + ((Rb) == `Register@RZ)) %2) == 0` may also occur as
`(((Rb) + ((Rb) == `Register@RZ)) %4) == 0` or
`(((Ra) + ((Ra) == `Register@RZ)) %2) == 0`. Still, solving it is the same for all of them. Luckily **SASS_Expr** also delivers the **patterns** for the expressions.

Listing 3.27: Every **SASS_Expr** expression has a stored, generic pattern that can be used to assign an efficient, custom solver. For example, Note the last 5 tokens are the types (Op_Mod, Op_Int, Op_RBrace, Op_Equal, Op_Int), ignoring the value of the modulo integer in the expression.

```
(((Rb) + ((Rb) == `Register@RZ)) % 2) == 0

pattern = (
  Op_LBrace, Op_LBrace, Op_LBrace, Op_Alias_0, Op_RBrace, Op_Plus,
  Op_LBrace, Op_LBrace, Op_Alias_0, Op_RBrace, Op_Equal, Op_Register,
  Op_RBrace, Op_RBrace, Op_Mod, Op_Int, Op_RBrace, Op_Equal, Op_Int
```

Following is the *solver* for the expression presented in Listing 3.27. In this case, it is a simple *Python* list comprehension.

```
@staticmethod
def __expr_type_8(sass:SM_SASS, expr:list, alias_nt:list, domain_vals:dict):
    alias = alias_nt[0]
    vals = domain_vals[alias]
    reg_val_1 = expr[11].value()
    mod_val = expr[15].value()

    old = domain_vals[alias]
    domain_vals[alias] =
          set(v for v in vals if ((v + int(v == reg_val_1)) % mod_val) == 0)
    # SASS_Expr_Domain_Common.final_domain_check(domain_vals)
    return domain_vals
```

Basic Acceleration Types

For expressions such as `DEFINED TABLES_opex_0(batch_t,usched_info)` and `(((Rd)+((Rd)==`Register@RZ)) %2) == 0` that do not contain *large value* representations such as UImm(32), the custom solver simply performs the table lookup, filters register values and for implication patterns, resolves the implication by calculating both sides of the implication and selecting and merging the three sets for which the implication is *True*.

```
@staticmethod
def implication(
    left_side_T:list, # left side of impl., evaluates to True
    left_side_F:list, # left side of impl., evaluates to False
    right_side_T:list,# right side of impl., evaluates to True
    right_side_F:list # right side of impl., evaluates to False
):
    # A | B |A->B
    # ------------
    # T | T | T
    # T | F | F
    # F | T | T
    # F | F | T

    return f(merge all sets for which A->B == T)
```

Large Function Expressions

All expressions that contain any kind of function (for example, SImm, UImm or F64Imm) that use more than 5 bits (meaning, have more than 32 values), are considered **large function expressions**. They are also solved with custom solvers, utilizing the **SASS_Range** class. It can perform the required set operations, correctly apply bit masks, and also be sampled uniformly at random.

There are exactly 46 expression patterns that require **SASS_Range**. Thus, they can **all** be solved with a custom solver. **Conveniently** there is no instruction class where the sets of variables for the large-function group of expressions overlaps with the remaining expressions. They are completely distinct. For example:

- There is no validation condition that contains a Ra **and** Sb_addr or Sb_offset.
- Registers and large functions are together in the list of operands for instructions, but they are never validated together in one single validation condition.

This is great, because we can calculate domains for the two groups of validation conditions separate from each other.

With these simplifications and accelerations, it is possible to indeed solve all the validation conditions and get a sufficiently large configuration space for every instruction class.

Set Contraction

The last problem that needs solving is *hard drive space requirement*. Millions of encoding value dictionaries (see Section 1 for an explanation of the term *encoding values*), mostly consisting of {*Python string* : **SASS_Bits**} pairs (the *string* represents the *alias*) need to be reduced to {*Python string* : set(...**SASS_Bits**...)} dictionaries to save space. Listing 3.28 depicts this shape and illustrates the problem.

Listing 3.28: Depiction of the output shape of the validation condition solutions as list of dictionaries: 'a','b','c' represent **alias**, for example Ra in *Register:Ra*, and the **SASS_Bits** are replaced with 0/1/2/3 integers for clarity.

```
enc_vals = [
    {'a':0, 'b':0, 'c':0},
    {'a':0, 'b':0, 'c':1},
    {'a':0, 'b':0, 'c':2},
    {'a':0, 'b':0, 'c':3},
    {'a':0, 'b':1, 'c':0},
    {'a':0, 'b':1, 'c':1},
    {'a':0, 'b':2, 'c':0},
    {'a':0, 'b':2, 'c':1},
    {'a':1, 'b':0, 'c':0},
    {'a':1, 'b':0, 'c':1},
    {'a':1, 'b':1, 'c':0},
    {'a':1, 'b':1, 'c':1},
    {'a':1, 'b':2, 'c':0},
    {'a':1, 'b':2, 'c':1},
]
```

By applying the algorithm

1. sort all variables by size of their sets in ascending order: the *sets* are set(a) = {0,1}, set(b)={0,1,2}, set(c)={0,1,2,3}.
2. recursively forwards group to the last variable: in this case, group by *a*, then by *b* and lastly by *c*
3. backwards group after recursion returns

Grouping forwards and then backwards groups all variables except the one with the smallest domain into larger sets, if possible. The *Python* implementation for this algorithm is available in Appendix A.1. Listing 3.29 shows the result if the algorithm is applied to Listing 3.28. The result is a shorter list of {*Python string* : set(...**SASS_Bits**...)} dictionaries with fewer total values than the input list of {*Python string* : **SASS_Bits**} *enc_vals* dictionaries.

Listing 3.29: Depiction of the same enc_vals as in Listing 3.28 after applying the contraction algorithm. Based on the ascending sorting order with respect to the domain sizes, the only non-contracted key is always the one with the smallest domain. In this case that is 'a'. The domain s are dom(a)={0,1}, b={0,1,2}, c={0,1,2,3}.

```
enc_vals = [
    {'a': {0}, 'b': {0}, 'c': {0, 1, 2, 3}}
    {'a': {0}, 'b': {1, 2}, 'c': {0, 1}}
    {'a': {1}, 'b': {0, 1, 2}, 'c': {0, 1}}
```

Finally, the format shown in Listing 3.29 is sufficiently small in size to be stored in one file for each SM.

The next Section 3.4.3 puts this into a larger perspective, outlining the entire process for how to calculate the *configuration space* for all instructions.

3.4.3 Valid Sets Calculations and Consolidation

This section integrates the previous Section 3.4.2 into the algorithm that generates the entire valid sets domain of the configuration space for one SM. The process is split into two steps: *valid sets calculation* and *valid sets to enc_vals consolidation*.

Valid Sets Calculation

For all instruction classes separately, do the following:

1. get the input value sets for all validation conditions:
   - for the large function expressions, the algorithm ends here
   - the other expressions get a list with value sets for every variable. For example, the set for a *Register* variable Ra = {0,1,...,10,100,...110,240,..,255} (see Section 2.3 for an overview of the available register resources and their values). Note that the input for Ra is **limited** as outlined in one of the previous sections.
2. compute the *cross product* for all sets of all variables. The result is a list of dictionaries as in Listing 3.28 where the input looks like Listing 3.29. This step effectively reverses the set contraction algorithm.
3. iterate all validation condition expressions using all *cross product* dictionaries and discard all dictionaries that evaluate to false for at least one validation condition
4. apply the set contraction algorithm of Section 3.4.2 to the remaining dictionaries to shape them back into the original form of *variable* = set(...values...), depicted in Listing 3.29
5. if we encounter an expression's pattern (see Listing 3.27) for the first time, test the calculated valid sets list for correctness

Store all valid sets in one *Python* pickle file per instruction class.

A side note on testing valid sets: it is infeasible to test every single expression. The calculations are based on the *patterns* of the expressions (see Listing 3.27). It is sufficient to test the calculations for every *pattern* once since every expression with the same pattern is calculated in exactly the same way.

Consolidation

Not all parts of an instruction class are included in the validation conditions. Thus, the encoding stage (see Section 2.16) contains variables that do not yet

have a set of possible values assigned.

In the *consolidation stage*, all valid sets that are computed in the previous section are **completed** such that each one represents one full encoding stage.

The result are a list of dictionaries as shown in Listing 3.29. Sampling one instruction's valid sets will result in **one enc_vals** dictionary (see Section 3.2.7). The two ways of sampling the valid sets are covered in the next Section 3.4.4.

Large and Small Calculations

The full calculations for the valid sets produce rather large files that require up to 8GB in memory, if loaded. Memory requirements for the Hopper architecture (SM 90) are the greatest. As it stands, the large domain calculations were only necessary to figure out the decoding algorithm in Section 3.3.

Thus, for practicality, there is an addition to the calculations to only use a targeted, reduced set of inputs.

- All $cache bits can be set to fixed values $WR=0x7, $RD=0x7, $REQ={}, $USCHED_INFO = {trans1, trans2, WAIT1}
- register operands (Ra, Rb, ..., Rd) are set to a range between 30 and 50
- instruction predicates are set to PT and their [inversion operator [!]] is set to 0

These values strongly reduce memory requirements and loading times for the files containing the valid sets and make them very useful to generate valid instructions either for later modification or just for fun.

3.4.4 Monte Carlo Sampling and EncDom

Since originally the valid sets were called *encoding domains*, the name for them in the code base for this work is still *EncDom*.

This section shows how the sets, calculated in the previous Section 3.4.3 can be used to generate random, but valid instructions that can then be used as is or modified.

Section 5.2.3 explains how to load a Cubin binary and turn *enc_vals* into a real instruction, either by creating a new one or replacing an existing one. For now, in the following *Python* snippets, we only get encoding values (see Section 3.2.7).

```
# Load the desired SM_SASS and add the encdom to it. Use the small one
sass = SM_SASS(86)
sass.load_encdom_small()

# Generate a valid set of encoding values for class_name
enc_vals = sass.encdom.pick(class_name)
```

`sass.encdom.pick(class_name)` can be called repeatedly and every time, *end_vals* will be a different, valid encoding for the instruction class *class_name*, looking like the *Python* dictionary in Code 1.

The following *Python* code showcases how the *configuration space* can be iterated using `sass.encdom.fixed_iter2`. Note that collecting all possible extension combinations as is done in the *Python* example can indeed produce 100'000 or more instruction variants for certain instructions.

```
# Get some class named class_name
class_:SASS_Class = sass.sm.classes_dict[class_name]
# ... and its properties
props:SASS_Class_Props = class_.props

# Get all values of all extensions for the class_name using
# the very useful SASS_Class_Props
ankers = {
    e:tt.value.get_domain({}, filter_invalid=True)
                                    for e,tt in props.tt_exts.items()}
result_enc_vals = []
for i in sass.encdom.fixed_iter2(instr_class=class_name, ankers=ankers):
    enc_vals:dict = i

    # Selectively modify some encoding values. For example
    enc_vals = overwrite_helper(False, 'Pg@not', enc_vals)
    enc_vals = overwrite_helper(PT, 'Pg', enc_vals)
    # ...

    result_enc_vals.append(enc_vals)
```

Chapter 4

VSCode Decoding Visualization

After seeing the entire encoding and decoding mechanisms, this section is a refreshing change of pace. It outlines some corner points of the VSCode extension that is used to visualize the results of the decoder as well as how it bridges *Flutter* and *Python*.

4.1 About

The VSCode extension is built with *Flutter* [19] and *SQLite* [20].

After decoding a CUDA binary, the *Python* decoder uses a straight forward database scheme to store all components into an SQLite database. This approach scales well and is portable and SQL can be used to query the instructions from it. For example, decoding test-binaries with about 2500 instructions creates a database file of about 20MB in size.

The SQLite database is then *serialized* by the *py_cubin* server and sent to the VSCode extension where it is *deserialized* and visualized using plain *SQL* statements to construct the visualizations for each SASS instruction.

Every decoded CUDA binary and the C++ surroundings that encompass it are fully contained in the SQLite database created based on it. The database can be downloaded and saved and then uploaded again to the VSCode visualizer, if needed.

The extension requires the py_cubin server to run.

4.2 Visualized Parts

The visualizer contains all SASS components that were mentioned up to this point, including some that were not. Figure 4.1 shows the top three instructions in the *folded* state as well as the *steering components*:

- The blue framed icon allows loading a binary containing CUDA kernel. It is not necessary to use *cuobjdump* first.
- The red framed icons allow *downloading* a decoded binary as SQL file as well as *uploading* a downloaded SQL file.
- The purple dropdown menus allow selecting a *decoded binary* (left) and within a binary, a *decoded kernel*. The decoder supports loading multiple C++ binaries at once and CUDA binaries with multiple kernels.
- The green framed *radio button* named `Single` and `All` allows decoding only the selected kernel at the time or all at once. This allows scaling the tool to binaries containing **thousands** of kernels. Since the decoding process is not too fast, decoding thousands of kernels at once can take a long time. Since that is necessary to inspect binaries created in the benchmarking Section 7.3.5, this *lazy loading* mechanism was added. Selected kernels are only decoded once and the results kept in storage. The *radio button* allows changing between the *lazy loading* mode (`Single`) and loading everything at once (`All`). The default setting is `Single`.
- The icons in the orange box contain a sliding tab widget. The left most icon is the decoder. The other two are space left for future work.

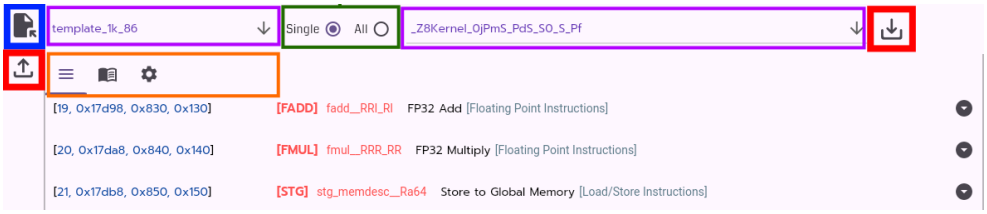


Figure 4.1: The first *stage* of the VSCode extension: show four different kinds of instruction offsets, the **Opcode**, the instruction class name and some documentation for every instruction. See Section 5.1.4 for detailed explanation of the meaning of each displayed offset.

Unfolding the **STG** instruction as shown in Figure 4.2 reveals the first level. It contains the decoded *instruction* that is also shown in *cuobjdump -sass* including all extension registers and the type of all function operands. In addition, it contains in further, unfold-able elements the full *formal class definition* that contains all information that is available for any instruction class, as well as the table with the *Latencies*.

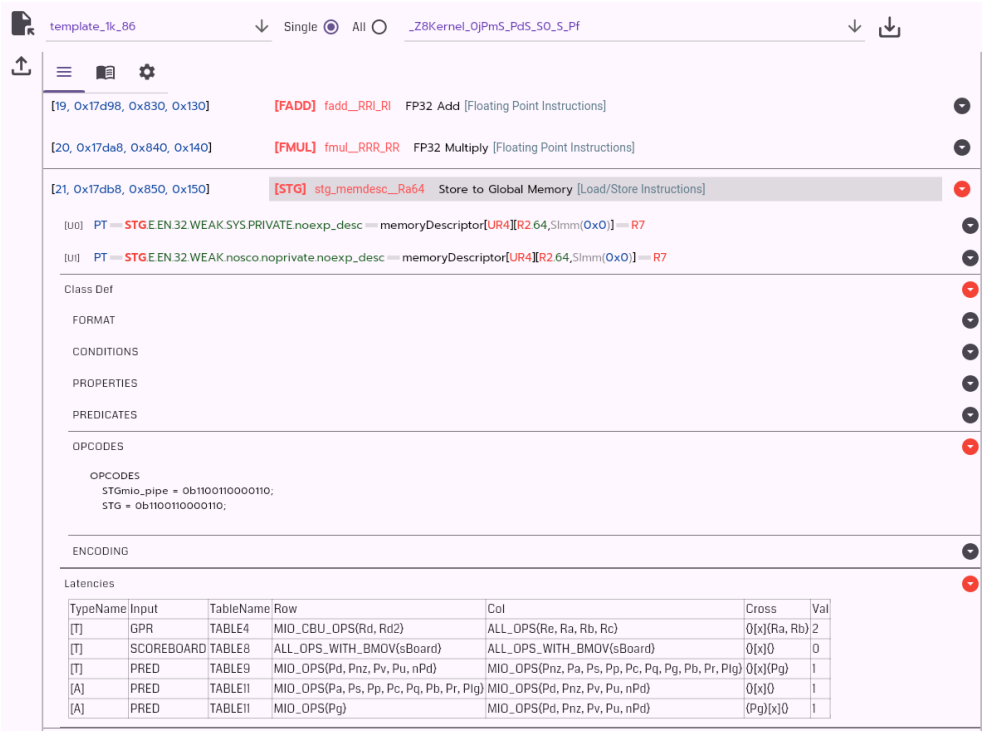


Figure 4.2: Unfolding one **Opcode** by clicking reveals the decoded *universes* as well as the options to unfold the *Class Definition* and the *Latencies* table.

Unfolding one selected *universe* in Figure 4.3 reveals the decoded $cache bits as well as constant values, if any are present, as is the case with the selected universe for **STG**. Furthermore, every universe contains a complete *trace* of how it was decoded, including all tables, which fields were decoded from table lookup operations and the full **SASS_Bits** *string* representation for every field in the encoding definition for the instruction class.

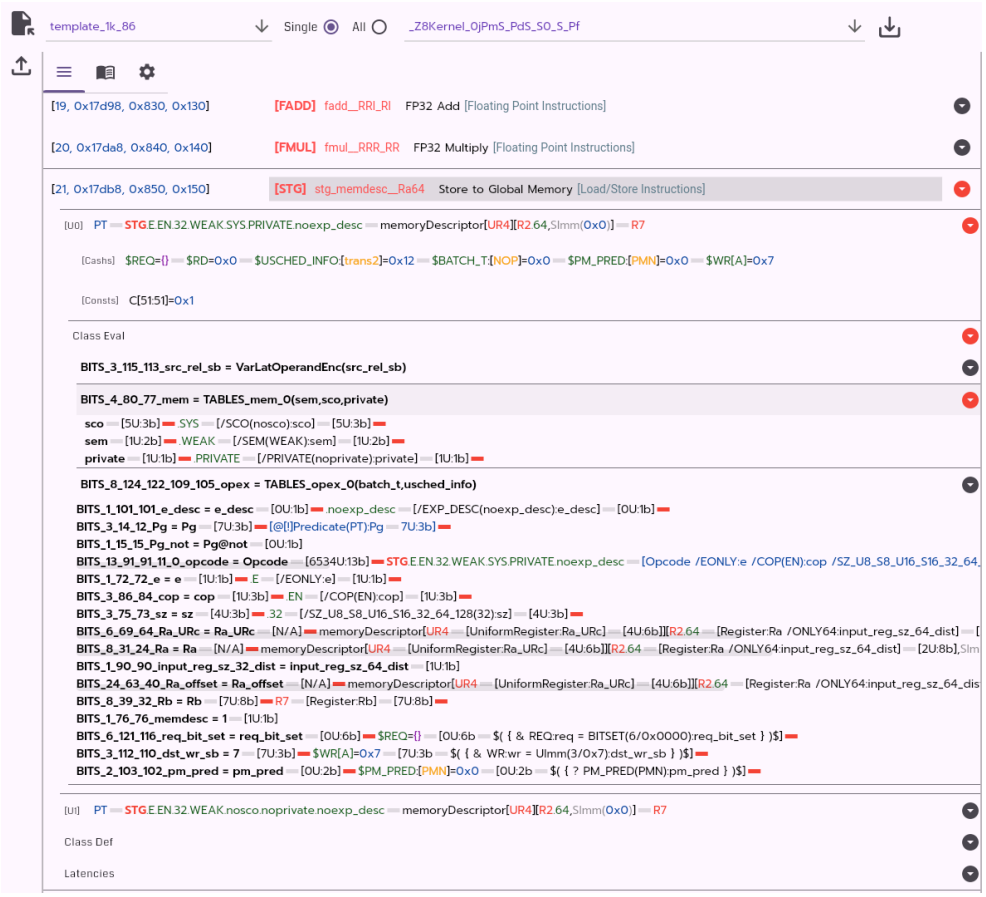


Figure 4.3: Clicking on one decoded *universe*, for example **[U0]**, reveals the decoded $cache bits and one additional unfordable element *Class Eval*. The *Class Eval* section contains all elements of the instruction in conjunction with where they are decoded and their formal definition.

See Section 5.1.4 for a detailed explanation of the offset values that look like [000, 0x18ff8, 0x800, 0x0].

Chapter 5

Cubin Binaries

This chapter is all about how to decode an actual binary file, containing one or more CUDA kernels. Check out Chapters 2 for a full formal introduction to SASS and Chapter 3 for how SASS instructions are encoded and decoded. This chapter *takes* and instruction as encoded by Section 3.2 and *injects* it into an existing binary or *extracts* encoded instructions from an existing binary to *input* them into the decoder of Section 3.3.

5.1 ELF Structure

This section outlines the **Cubin ELF** format. It is what the SM_Cubin_File decodes into *Python* structures and includes details of which bits do what, how Cubins can be found in larger C++ binaries and how specific kernels can be found inside of one Cubin section, in what sequence, at which offsets, etc.

5.1.1 A Word on Nomenclature

In this section, and also in other parts of this report, we distinguish between C++ and CUDA binaries.

- **C++ binary**: includes the entire binary file as produced by the *NVCC* compiler. It contains the C++ and the CUDA instructions and follows the standard Unix ELF format
- **CUDA binary**: includes **only** the CUDA part of the binary file. This is identical with the output of *cuobjdump [c++ binary] -xelf all*

5.1.2 Finding CUDA ELF

Any C++ binary file can be loaded in binary mode. To find a CUDA ELF we search for the bit pattern `\x7f\x45\x4c\x46\x02\x01\x01`. Note that these

are 7 bytes. Byte nr 8 can be either \x33 for SMs 50 to SM 90 or \x41 for SMs 100 and 120.

To find the *end* of the CUDA portion, we check out the 8 bytes at offset (0x20). Listing 5.1 shows the first 5 64 bit words for an SM 86 CUDA binary. *Note* the value *0x33* at the end of the first word and *0x12* in second place of the last one. Since the MSB is in the *right* side for this notation, the word represents an *offset* of *0x1200* = 4608 bytes. The offset is from the place where we found the beginning of the CUDA binary using the 7 byte code and 0x33 or 0x41 and leads to a location close to the end of the CUDA binary, skipping all the instructions and most of the configuration bytes. From there, we can find the end of the CUDA binary searching for the 4 byte code \x01\x1b\x03\x3b. Listing 5.2 illustrates the process with an example.

Listing 5.1: 8 bytes at offset 0x20 contain an offset that leads *close* to the end of the CUDA binary. the MSB is on the right side. Thus, the last word represents the value **0x1200** = 4608

```
0x000: 7F 45 4C 46 02 01 01 33
0x008: 07 00 00 00 00 00 00 00
0x010: 02 00 BE 00 80 00 00 00
0x018: 00 00 00 00 00 00 00 00
0x020: 00 12 00 00 00 00 00 00
```

A further distinction is in finding the architecture number. Up to and including Hopper (SM 90), the architecture is stored in location index 48 while for Blackwell (SM 100 and 120) it is at location 49. The *architecture number* is the same as the SM number. Meaning, Ampere has 80 and 86, Hopper has 90, etc. The following *Python* snipped shows how to extract the architecture number.

```
1 def read_arch(is_blackwell:bool, bits:bytes) -> int:
2     if is_blackwell: arch = int(bits[48+1])
3     else: arch = int(bits[48])
4     return arch
```

Before we can use the previous *Python* sippet, we have to find out if the current architecture is *Blackwell* or not. We can do this by examining the byte at index location 8. All architectures up to *Blackwell* will have the value **7** in this location while *Blackwell* has the value **8**.

```python
def read_arch_b(bits:bytes) -> int:
    is_blackwell = int(bits[8]) == 8
    # We know more now, go read the correct byte for
    # the architecture
    return SM_CuBin_Lib.read_arch(is_blackwell, bits)
```

Listing 5.2 shows some selected sections of a C++ binary with a contained Cubin binary, layed out with some descriptive comments.

Listing 5.2: This Listing shows all relevant offsets to safely find the CUDA binary in a C++ binary. It also shows a concrete example for offsets.

```
arch: 86
k_start: 0x187f8                 <== The result of the search
k_end: 0x19aa0

0x00000: 7F 45 4C 46 02 01 01 03 <== start C++ binary
0x00008: 00 00 00 00 00 00 00 00
0x00010: 03 00 3E 00 01 00 00 00
0x00018: 70 40 00 00 00 00 00 00
 ... ...
0x187f8: 7F 45 4C 46 02 01 01 33 <== start CUDA, 0x33: SM 50 to 90
0x18800: 07 00 00 00 00 00 00 00 <== 0x07, not Blackwell
0x18808: 02 00 BE 00 80 00 00 00
0x18810: 00 00 00 00 00 00 00 00
0x18818: 00 12 00 00 00 00 00 00 <== 0x1200 offset to almost
0x18820: 00 0F 00 00 00 00 00 00     the end
0x18828: 56 05 56 00 40 00 38 00
 ... ...
0x19a00: 00 12 00 00 00 00 00 00 <== we find 0x1200 here again
0x19a08: 00 00 00 00 00 00 00 00 ... this last section contains
0x19a10: 00 00 00 00 00 00 00 00     the offsets to all kernels
0x19a18: A8 00 00 00 00 00 00 00     in the CUDA binary. There
0x19a20: A8 00 00 00 00 00 00 00     are no huge values in this
0x19a28: 08 00 00 00 00 00 00 00     section which is why we
0x19a30: 01 00 00 00 05 00 00 00     can safely use the 4 bytes
0x19a38: 10 06 00 00 00 00 00 00     \x01\x1b\x03\x3b to find
0x19a40: 00 00 00 00 00 00 00 00     the end.
0x19a48: 00 00 00 00 00 00 00 00
0x19a50: F0 08 00 00 00 00 00 00
0x19a58: F0 08 00 00 00 00 00 00
0x19a60: 08 00 00 00 00 00 00 00
0x19a68: 01 00 00 00 05 00 00 00
0x19a70: 00 12 00 00 00 00 00 00
0x19a78: 00 00 00 00 00 00 00 00
0x19a80: 00 00 00 00 00 00 00 00
0x19a88: A8 00 00 00 00 00 00 00
0x19a90: A8 00 00 00 00 00 00 00
0x19a98: 08 00 00 00 00 00 00 00
0x19aa0: 01 1B 03 3B 34 02 00 00 <== this is the end
```

Why do this manually instead of using Nvidia's *cuobdump* tool? The vast majority of work is decoding the CUDA binary itself, not finding it inside its C++ parent. Thus, adding a little work to remove one system call to an external tool reduces the number of created garbage files in the process and overall reduces complexity of the tool chain. It is also easier to *keep* the non-SASS-instruction portions of the file stored to easily use them when a decoded binary is assembled back into runnable program.

In the next Section 5.1.3 we use *Python*'s *elftools* module to decode the Cubin ELF format and check out the relevant sections in more detail.

5.1.3 Decoding CUDA ELF

While CUDA follows an ELF [21] structure and it is possible to find all sectors using the *Python* module *elftools*, the names of the sectors are Nvidia specific and one must check them out first to find out what is in them.

Assuming we have a CUDA binary that we got from Section 5.1.1, we typically find some sections that generally seem to belong to ELF formats to facilitate navigation.

- **.shstrtab**, **.strtab**, **.symtab**: contain lists of used symbols
- **.debug_frame**, **.rel.debug_frame**, **.rela.debug_frame**: contain some bytes that seem to be related to marking a certain spot in the CUDA binary. This is speculation, though, and there is no need to find out in more detail.

Then there are a range of sections with *nv.*-prefixed names.

- **.nv.callgraph**, **.nv.prototype**, **.nv.rel.action**: contain some information but may be related to the execution of a kernel too
- **.nv.info**: this is one of the two interesting sections. Assuming, we have three kernels, named *kernelA*, *kernelB* and *kernelC*, we will find **.nv.info._Z7kernelAjPm**, **.nv.info._Z7kernelBjPm** and **.nv.info._Z7kernelCjPm**. These are the *shibbolethed* names of the kernels that can also be selected in the *visualization tool* in Chapter 4.
- **.nv.constant0**: it just so happens that at least one of the kernels uses a constant value. Constants get their own sections.
- **.text**: This is the second, very interesting section. For **every** kernel, there exists one **.text** section: **.text._Z7kernelAjPm**, **.text_Z7kernelBjPm** and **.text._Z7kernelCjPm**

The **.text.[kernel name]** sections are the only ones modified if instructions are changed.

Sidenote: all of these offsets for every decoded instruction are available in **Instr_CuBin_Misc** from decoding Section 3.3.2.

Now we have all kernels contained in a CUDA binary, contained in a C++ binary as bytes, combined with all offsets we require for debugging and a pleasant experience. The next Section 5.1.4 shows how this information is visualized in the visualization tool from Chapter 4.

5.1.4 Linking with SASS Visualization

In this section, we examine how the different offsets introduced in the previous Sections 5.1.1 and 5.1.3 are shown in the SASS visualizer from Chapter 4. Figures 5.1 and 5.2 show the first and the last instructions in the same kernel used in Listing 5.2 while Listing 5.3 shows the binary code for the same two instructions.

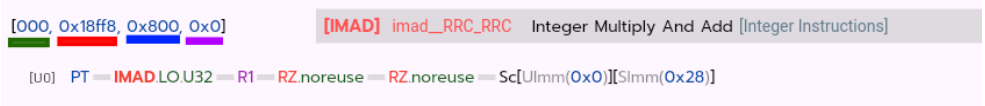


Figure 5.1: First **IMAD** instruction in an SM 86 CUDA binary. Note the indicated offsets [000, 0x18ff8, 0x800, 0x0]. From left to right, these are: [instruction number, offset starting at 0x0 C++ binary, offset starting at 0x0 CUDA binary, offset starting at 0x0 instructions section].

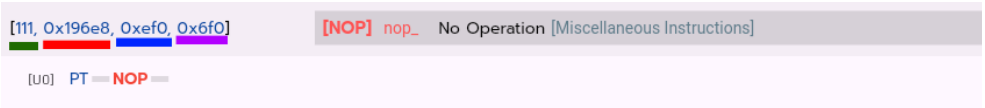


Figure 5.2: Last **NOP** instruction in an SM 86 CUDA binary. Note the indicated offsets [111, 0x196e8, 0xef0, 0x6f0]. Compared to Figure 5.1, the last entry 0x6f0 = 0xef0 - 0x800. Listings 5.2 and 5.3 show selected parts of the same binary file. For example, in Listing 5.2 we see the first bytes of the Cubin binary, starting at 0x187f8 and Listing 5.3 shows the **first** and **last** instructions with absolute offsets relative to the C++ binary (left) and with offsets relative to the beginning of the CUDA binary (right). These are the same instructions as shown in Figure 5.1 and this Figure: **IMAD** and **NOP**.

Listing 5.3: The **first** IMAD and the **last** NOP instructions in an SM 86 kernel. Left: offsets relative to the beginning of the C++ binary, right: offsets relative to the beginning of the CUDA binary. Note that the two 8 byte words before and after are all zero.

```
   Absolute offsets in C++ binary      CUDA binary offsets
   ==============================      ===================
                          24 76... => first IMAD
   0x18fe8: 00 00 00 00 00 00 00 00 | 0x7f0: 00 00 00 00 00 00 00 00
   0x18ff0: 00 00 00 00 00 00 00 00 | 0x7f8: 00 00 00 00 00 00 00 00
XX 0x18ff8: 24 76 01 FF 00 0A 00 00 | 0x800: 24 76 01 FF 00 0A 00 00 XX
XX 0x19000: FF 00 8E 07 00 E4 0F 00 | 0x808: FF 00 8E 07 00 E4 0F 00 XX
             .....                                .....
                          18 79... => last NOP
XX 0x196e8: 18 79 00 00 00 00 00 00 | 0xef0: 18 79 00 00 00 00 00 00 XX
XX 0x196f0: 00 00 00 00 00 C0 0F 00 | 0xef8: 00 00 00 00 00 C0 0F 00 XX
   0x196f8: 00 00 00 00 00 00 00 00 | 0xf00: 00 00 00 00 00 00 00 00
   0x19700: 00 00 00 00 00 00 00 00 | 0xf08: 00 00 00 00 00 00 00 00
```

The next Section 5.2 introduces the *Python* module *py_cubin*, implementing the scheme outlined in this and the previous Sections 5.1.3 and 5.1.1 as well as a way to convert modified binaries back to executables.

5.2 SM_CuBin_File, SM_CuBin_Kernel and the py_cubin Module

The concepts introduced in Sections 5.1.1 and 5.1.3 are available in the *Python py_cubin* module in the classes SM_CuBin_File and SM_CuBin_Kernel. This section outlines some good entry points using *Python* samples.

5.2.1 Opening a CUDA Binary

The following *Python* code demonstrates how to open a binary containing CUDA kernels and how to get an overview of which *index* maps to which *kernel*.

```
# Get the absolute path to the binary file
ff = '/.../.../binary_file'
# Load the **correct** SM_SASS version, for example for SM 86
sass = SM_SASS(86)
# Load and decode file
cb1 = SM_CuBin_File(sass, ff)
# Load and wipe file, no decoding
cb2 = SM_CuBin_File(sass, ff, whipe=True)

# Assuming we load a file with three
# kernel: kernelC, kernelB and kernelA
mm = ff.get_kernel_index_map()
# mm = {
#   '_Z7kernelCjPm': 0,
#   '_Z7kernelAjPm': 1,
#   '_Z7kernelBjPm': 2
#}
```

5.2.2 Increasing the Available Register Count

Usually, before modifying a CUDA kernel, it is prudent to increase the available register count to something such as 100. Failing to do so and running into the register limit will produce an *invalid instruction encountered* CUDA error.

```
sass = SM_SASS(86)
# Open a CUDA binary 'template'
target_cubin = SM_CuBin_File(sass, template)
# Get the current register count for every kernel in
# the CUDA binary
```

```
nrc = target_cubin.reg_count()
# Override the register count for all kernels in the
# CUDA binary
nnrc = {n:100 for n,k in nrc.items()}
target_cubin.overwrite_reg_count(nnrc)
```

5.2.3 Adding SASS Instructions to CUDA Templates

SM_CuBin_File features three convenient methods: `get_instr`, `replace_instr` and `create_instr`.

```
# Returns a representation of the indicated instruction
def get_instr(kernel_index:int, instruction_index:int) -> Instr_CuBin_Repr:
    ...

# Replace instruction in the target kernel at the indicated
# location with the new one, returns the old one
def replace_instr(
        target_kernel_index:int,
        target_instruction_index:int,
        source_instruction:Instr_CuBin_Repr) -> Instr_CuBin_Repr:
    ...

# Comon pattern using the above two instructions
# and two open SM_CuBin_Files cb1 and cb2 and
# some index n
cb2.replace_instr(0, n, cb1.get_instr(0,n))

# Create instruction in the target kernel at the indicated
# location using the passed enc_vals and class_name,
# returns the old one
# NOTE: the enc_vals and class_name must fit together
def create_instr(
        kernel_index:int,
        instruction_index:int,
        class_name:str
        enc_vals:dict) -> Instr_CuBin_Repr:
    ...
```

Check out the tutorial in Section 6.3 to see how to create the inputs for `create_instr` and use it to create custom CUDA kernels.

5.2.4 Assembling Modified CUDA Template to C++ Binary

In the previous two Sections 5.2.1 and 5.2.3 we open an existing CUDA *template* and modify it a bit using `create_instr` or `replace_instr`. The following sample code turns the modified, decoded binary, represented by a SM_CuBin_File back into a runnable binary.

Sidenote: autocorrection of CUDA errors is not included in this process.

Check out Sections 3.3 and 3.2 for information about how individual instructions are decoded and can be assembled back into a runnable state. All the other parts, required for a runnable binary are stored by SM_CuBin_File and appended/prefixed to the reassembled SASS instructions.

```
# Assuming cb1 is an open SM_CuBin_File,
# assemble it to a runnable binary at
# full path location 'file_name'
cb1.to_exec(file_name)
# Change the permissions of the new file so that the operating
# system is allowed to run it
os.system('chmod +x {0}'.format(file_name))
```

Chapter 6

SASS Tutorials

In this chapter, we present a series of tutorials aimed at providing understanding for the final custom kernel SASS template that will be used in the benchmarking Section 7.3 and is presented in the last Tutorial 6.13 in this chapter.

6.1 Introduction

This section introduces how to create kernels to test things with SASS using some examples, starting with the simplest possible kernel, the "empty" kernel, containing only 3 real instructions and a lot of **NOP**s and ending with the rather complex kernel used as template usable for various benchmarking scripts. The tutorials chapter explains the principles and SASS instructions, while the full example is available in Appendix A

Since CUDA kernels are embedded in C++ binaries, there are two stages of embedding:

1. the CUDA instructions inside the C++ binary
2. individual CUDA kernels inside the CUDA instructions section inside the C++ binary

The py_cubin module described in the previous Chapter 5 is able to decode a binary and exposes every CUDA kernel and it's instructions using an index and within each CUDA kernel, every SASS instruction, also using an index.

Because we don't want to deal with the regular C++ binary's structure and how to embed *n* CUDA kernels inside of a C++ binary, we use C++ *templates* to crate binaries with the desired number of Kernels in it, each one with a sufficient number of SASS instruction *slots* that we can overwrite with our own instructions. Note that this is also the place where we can configure the *interface* between C++ and SASS. It is not trivial to correctly embed a

CUDA kernel with it's arguments and calling CUDA kernels with made up arguments doesn't necessarily work.

Thus, before we jump to SASS, it is prudent to present a good way to create the C++ side of a template to reuse it as much as possible. We like a template that features at least two things:

- a good variety of Kernel parameters we can choose from and pass data back and forth with various data types
- at least one parameter that can be passed from a bash terminal to configure integrated loops

The following kernel produces about 60 SASS instructions and a good variety of input and output parameters. We can pass at most **8** parameters to a CUDA kernel and use them without too much trouble.

```
__global__ void
kernelT(unsigned int a,      // Easy to pass as terminal arg, use for loops
        uint64_t *control,   // Good idea to write progress "checks"
        uint64_t *ui_output, // Using "unsigned" ints makes the compiler
                             // not use "x < 0" checks and produces
                             // fewer instructions
        double* d_output,    // We like some correctly formated doubles.
                             // Use pointers. They allow for two way
                             // communication.
        uint64_t* ui_input,  // Some transfer space for integers (32/64 bit
                             // doesn't matter with ints
        double* d_input,
        uint64_t *clk_out_1, // A fixed output for measured clocks
        float* f_output      // Floats and doubles are not int32 and int64
) {
    // Using 'pragma unroll X' is a good way to influence the number
    // of instructions the CUDA compiler spawns:
    // less unrolling => fewer instructions
    #pragma unroll 2
    for(unsigned int i=0; i<a; ++i){
        // The compiler doesn't change the relative position of clock64(),
        // provided the the result is used. Good for "sandwitching" code
        int64_t t1 = clock64();
        /* something we like to easily find inside of SASS */
        int64_t t2 = clock64();
        // static_cast with some calculations as argument are SASS
        // instruction bombs
        f_output[i] = static_cast<float>(a *
                                  (static_cast<float>(t2-t1) + 1.256f));
    }
    return;
}
```

Code 9: This is a simple kernel with a good range of arguments that will produce about 60 instruction slots if compiled with NVCC.

The reader is invited to check out Appendix A.3 for a full C++ example with two embedded CUDA kernels and some more useful information.

6.2 Encountering Illegal Instructions

This tutorial is about the most likely CUDA error one may encounter and its likely causes. The most annoying one may be "an illegal instruction was encountered" because it comes without a line number.

The most likely situation one may encounter this error is during stitching together a kernel that may use more than a trivial amount of registers. Usually with adding an instruction that one has used 100 times before.

The likely response is to go check the old kernel, where the instruction was used successfully last time and comparing the bits. Then one notices that they are exactly the same and starts doubting the world.

The reason for this CUDA error is most likely running out of usable registers because the register count is too low. Since we use template kernels, if not modified, the register limit of the template kernel applies. That is why it's imperative to change the register limit to a high enough count in the beginning.

```
nrc = target_cubin.reg_count()
# Use a fixed value like '100' here
nnrc = {n:100 for n,k in nrc.items()}
target_cubin.overwrite_reg_count(nnrc)
```

Note that everything that can be used in destination or source register slots counts towards the total used register count.

6.3 Create a Custom SASS Instruction

This tutorial shows the process to select and create a custom SASS instruction to be integrated inside a custom kernel. This process is the prequel for all tutorials that load a template kernel, wipe all instruction slots with **NOP** and replace them with new ones, starting with the next Tutorial 6.4. The full code sample for this Tutorial is in Appendix A.2.1.

First we have to load the fitting *SASS* abstraction. Since we need to get some valid, default values for the instruction from somewhere, we need the EncDom instruction generator introduced in Section 3.4 too. Note that the **small** version is always sufficient for this kind of work!

```
# Load the desired SM_SASS and add the encdom to it.
# Use the small one!
sass = SM_SASS(86)
sass.load_encdom_small()
```

After loading the *SASS* representation for the required SM version (in this case, it is 86), we pass it to the *creator* class as the first argument. The second step is to create the frame for a *Python* class that will contain the encoding values for the instruction.

Make sure to name the *arguments* properly. It should be intuitive which kind of values should be passed. Keep in mind, that we are working with SASS instructions directly that do not feature any other kind of debugging mechanism than CUDA errors and oftentimes, the only viable debugging strategy is commenting all instructions but one, run, uncommenting the next instruction, run, repeat.... which is a very time consuming process. Precision is key! Prefix the names with *u* for *UniformRegisters*, use *type annotations*, denote which arguments are source and which one is the destination.

The following code snippet contains two *TODO*s that need to be filled out.

```
class SASS_KK__umov__UR:
    def __init__(self, sass:SM_SASS,
                         negate_upred:bool,
                         upred:tuple,
                         target_ureg:tuple,
                         source_ureg:tuple,
                         usched_info_reg:tuple
    ):
        class_name = 'umov__UR'

        # Generate a valid set of encoding values for class_name
        enc_vals = sass.encdom.pick(class_name)
        # Print the encoded values as strings to be copy-pasted into
        # the enc_vals dictionary below
        print(SASS_Create_Utils.enc_vals_dict_to_init(enc_vals))

        # TODO: fill enc_vals in
        # => Set a debug point here! <=
        enc_vals = {
        }

        # TODO: add enc_vals overwrites using
```

```
23          # overwrite_helper(...)
24
25          # The reader may recognize this handy helper method call from earlier
26          # chapters
27          Instr_CuBin.check_expr_conditions(
28                  kk_sm.sass.sm.classes_dict[class_name],
29                  enc_vals,
30                  throw=True)
31
32          # Use simple names to access both enc_vals and class_name
33          # from the 'outside'. Following tutorials will use these
34          # two to create real instructions and insert them into
35          # kernel
36          self.enc_vals = enc_vals
37          self.class_name = class_name
```

Run the code snipped above with some valid, but non-important input. The registers are abstracted using the **SM_CuBin_Regs**. It turns the register designations into *Python* tuples with some useful names.

```
1  # Abstract of registers as tuples
2  regs = SM_CuBin_Regs(sass.sm.details.REGISTERS_DICT)
3
4  # Call the creator with some 'valid' arguments. It is not
5  # important if the instruction would actually work like this
6  # but to check if the creator works and the
7  # Instr_CuBin.check_expr_conditions throws an exception or not
8  SASS_KK__umov__UR(sass,
9      negate_upred=False, upred=regs.UniformPredicate__UPT__7,
10     target_ureg=regs.UniformRegister__UR2__2,
11     source_ureg=regs.UniformRegister__UR4__4,
12     usched_info_reg=regs.USCHED_INFO__WAIT1_END_GROUP__1)
```

Before we run the initial *creator* sniped at the top of this tutorial, we set a **break point** where it is indicated in the code snipped to set a break point at **line 19**. Then we run the code and at the break point location, the `print(..)` on line 15 will have printed the encoding values in a *copy-paste* format. Copy and paste the encoding values into the `enc_vals` dictionary and remove the *pick* and *print* code lines 15 and 12 with a comment.

The `enc_vals` dictionary on line 19 should now look approximately as follows:

```
        enc_vals = {
            # predicate, set by param
            'UPg': SASS_Create_Utils.sass_bits_from_str('2U:3b'),
            # negate predicate, set py param
            'UPg@not': SASS_Create_Utils.sass_bits_from_str('0U:1b'),
            # source ureg, set by param
            'URb': SASS_Create_Utils.sass_bits_from_str('31U:6b'),
            # destination ureg, set py param
            'URd': SASS_Create_Utils.sass_bits_from_str('45U:6b'),
            # keep 0
            'batch_t': SASS_Create_Utils.sass_bits_from_str('0U:3b'),
            # keep 0
            'pm_pred': SASS_Create_Utils.sass_bits_from_str('0U:2b'),
            # wait for nothing
            'req_bit_set': SASS_Create_Utils.sass_bits_from_str('0U:6b'),
            # set py param
            'usched_info': SASS_Create_Utils.sass_bits_from_str('17U:5b')
        }
```

Note that one usually has to go about finding out what the individual encoding values do manually by browsing the *instructions.txt* file for the fitting SM. This shortcoming is addressed in the Future Work Section 8.4 that proposes an *instruction browser* that will streamline this process.

In the second *TODO* location on line 22 in the initial *creator frame* we can now use the `overwrite_helper` to replace values in the `enc_vals` dictionary with the ones we passed as parameters to the class.

```
        enc_vals = overwrite_helper(negate_upred, 'UPg@not', enc_vals)
        enc_vals = overwrite_helper(upred, 'UPg', enc_vals)
        enc_vals = overwrite_helper(target_ureg, 'URd', enc_vals)
        enc_vals = overwrite_helper(source_ureg, 'URb', enc_vals)
        enc_vals = overwrite_helper(usched_info_reg, 'usched_info', enc_vals)
```

We can now use the class `SASS_KK__umov__UR` with custom parameters to create customized instruction encodings that we can use in conjunction with the *create* and *replace* methods introduced in Section 5.2.3.

All Tutorials featuring a full code sample in the Appendix use this kind of *creator* class and all used *creator* classes are implemented in the *PySASSCreate Python* project included in the code base for this work.

Also, note that the two *helper* methods `overwrite_helper` and `enc_vals_dict_to_init` are given in their full form in Appendix A.2.2 for reference.

6.4 Empty Kernel

This tutorial shows how to load, clean, modify and store a template using the least amount of SASS instructions possible. The full code is available in Appendix A.4.

Figure 6.1: Simple kernel only containing the initial **MOV** and the return **EXIT** and **BRA** instructions. Compiling a kernel with no content will yield this as a result.

Note that the **enumeration indices** match the **instruction indices** in Figure 6.1!

[00] **[MOV]**, mov__RC: the first instruction is always the **MOV** in Figure 6.1. No matter which architecture, it always uses R1 and address 0x0x0x28. An experiment using **printf** and **CALL** suggests that the first 5 bytes are used to store stack and frame pointers in a similar way as it's being done in a CPU.

[01] **[EXIT]**, exit_: **EXIT** is always the second to last instruction

[02] **[BRA]**, bra_: the last instruction is **BRA**. Note that it branches onto itself, using an offset of -0x10, that in a world where each instruction is 128 bits jumps back to the beginning of the **BRA** instruction.

We can create this kernel by first loading the template using *SM_CuBin_File* and wipe the template using **NOP** instructions. We only need the template instructions to create space in the template. The alternative would be to correctly embed N kernels in a C++ binary with all offsets and sizes set

manually. It is much easier to create a template. The traditional way to remove all template instructions is with a loop.

The loop is not very efficient, though, and doesn't scale to 1000ds of instructions, let alone, 1000ds of kernels. Using the parameter `wipe=True` will short-cut loading the template by only loading the binary structure and immediately replacing all instructions with the SM-appropriate **NOP**.

```
# Fast way: use whipe=True
==========================
target_cubin = SM_CuBin_File(kk_sm.sass, template, whipe=True)

# Traditional: load template then whipe it with a NOP loop
==========================================================
target_cubin = SM_CuBin_File(kk_sm.sass, template)

# Create a generic NOP instruction and replace the instructions
# in the target cubin with it
nop = sc.SASS_KK__NOP(kk_sm)
for i in range(0, len(target_cubin[0])):
    # replace instruction in
    #   kernel index: 0
    #   instruction index: i
    target_cubin.create_instr(0, i, nop.class_name, nop.enc_vals)
```

After loading a template, we may want to increase the number of available registers slots we can use. The CUDA compiler extracts the number of required registers from the CUDA code such that register usage can be optimized. If we don't want to spend time constructing kernels where the compiler can't optimize register usage, using a lot of carefully constructed data dependencies, we have to overwrite the respective bits with a higher value. Note that we generally assume that we have N kernels in a binary. We have to overwrite the register count for all the kernels we want to use.

```
nrc = target_cubin.reg_count()
nnrc = {n:k+10 for n,k in nrc.items()}
target_cubin.overwrite_reg_count(nnrc)
```

We then use `target_cubin.create_instr(kernel_index, instr_index...)` to replace three instruction slots with **MOV**, **EXIT** and **BRA** and turn the finished product back into a binary.

```
# Assemble a new binary with the modified kernel
target_cubin.to_exec("some.useful.binary.name")
# Make the kernel executable
os.system('chmod +x {0}'.format("some.useful.binary.name"))
# Run it with subprocess, capture the output
result = subprocess.run(['{0}'.format(exe_name), exe_arg],
                        capture_output=True, text=True)
# parse result.stdout...
```

6.5 Kernel Arguments Addresses and Alignment

This example consists of two kernels, compiled with the CUDA compiler. It showcases address alighment for kernel arguments as well as the difference between using pointers and passing values by value.

First we look at this very simple example. We use *printf* to validate that we indeed use the correct values for a and b and check what the compiler creates.

```
// Pass two 32 bit unsigned integers by value.
__global__ void kernelT1(unsigned int a, unsigned int b) {
    printf("param a=%u, param b=%u\n", a, b);
    return;
}
```

Passing 32 bit values by copying them allows using the simplest mechanism imaginable to get them into the standard 32 bit registers.

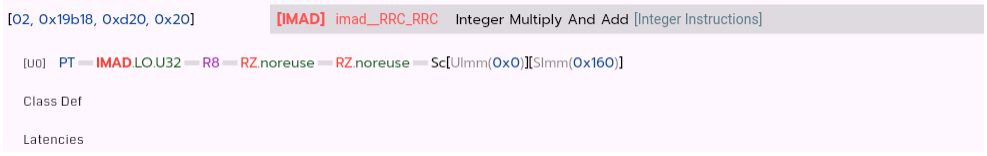


Figure 6.2: Load the first argument using IMAD and a constant memory access at 0x0x0x160.

Figures 6.2 and 6.3 show how to use an IMAD version where one operand is a constant memory access and the other two operands are both RZ. **Especially**, the two arguments can be loaded at absolute memory bank offsets 0x160 and 0x164. **In fact**, the first argument is always at memory bank 0x0 with offset 0x160. As we see in the next kernel, that uses a pointer, the subsequent

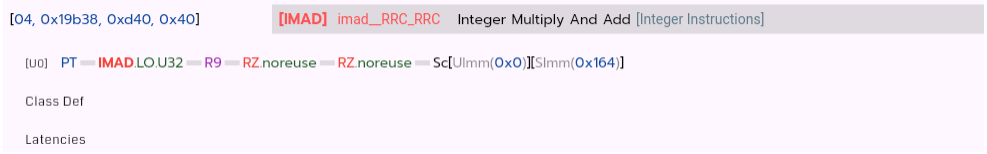


Figure 6.3: Load the second argument using **IMAD** and a constant memory access at 0x0x0x164.

arguments are subject to memory alignment, much the same way as C++ aligns **struct** entries.

kernelT2 uses a pointer to pass the second argument while the first one remains *by value*.

```
1 // Pass two 32 bit unsigned integers, the first one one by value and
2 // the second one using aa pointer
3 __global__ void kernelT2(unsigned int a, unsigned int* b) {
4     printf("param a=%u, param b=%u\n", a, *b);
5     return;
6 }
```

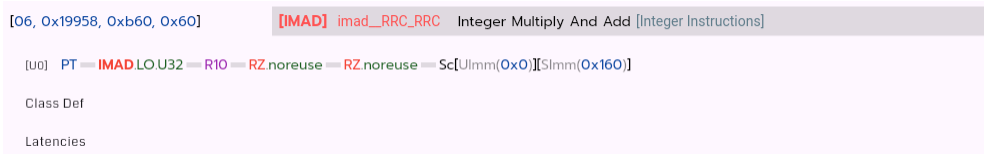


Figure 6.4: Load the first argument using **IMAD** and a constant memory access at 0x0x0x160 the same way as in Figure 6.2.

Figure 6.4 loads the first argument in the same way as we observer in Figure 6.2. However, in Figure 6.5, the pointer argument requires a bit more doing. **Very importantly**, we need **two IMAD** instructions. The first one loads constant memory offset 0x168 into register R2 and the second one loads constant memory offset 0x16c into register R3.

At this point, we interject that the registers in Nvidia GPUs are sorted in 4 byte chunks where the RXX registers cover 32 bits each. To cover 64 bits, we can either use a uinform register URXX with added extension .64 or **two adjacent** RXX registers, for example R2 and R3, covering 32 bits each. We **must** explicitly load R3 as well, even if we never use it.

This gives rise to the pattern, we observe in Figure 6.5. First we load the **address** value of argument two into 64 bits of memory, covered by R2 and R3, then we only use R2 in the **LDG** instruction featuring the *memoryDescriptor*. Note that **LDG** uses UR4 as base register. UR4 is loaded using **ULDC** and

a *weird* offset 0x118. This offset seems to be quite randomly chosen but the **pattern** is the same on all architectures newer than SM 80. **Recall** that *memoryDescriptors* were introduced in SM 80. Section 6.6 will showcase a generic and more convenient way to load kernel arguments, supporting also the SM 70ies. For now, we are more interested in the memory alignment of the arguments.

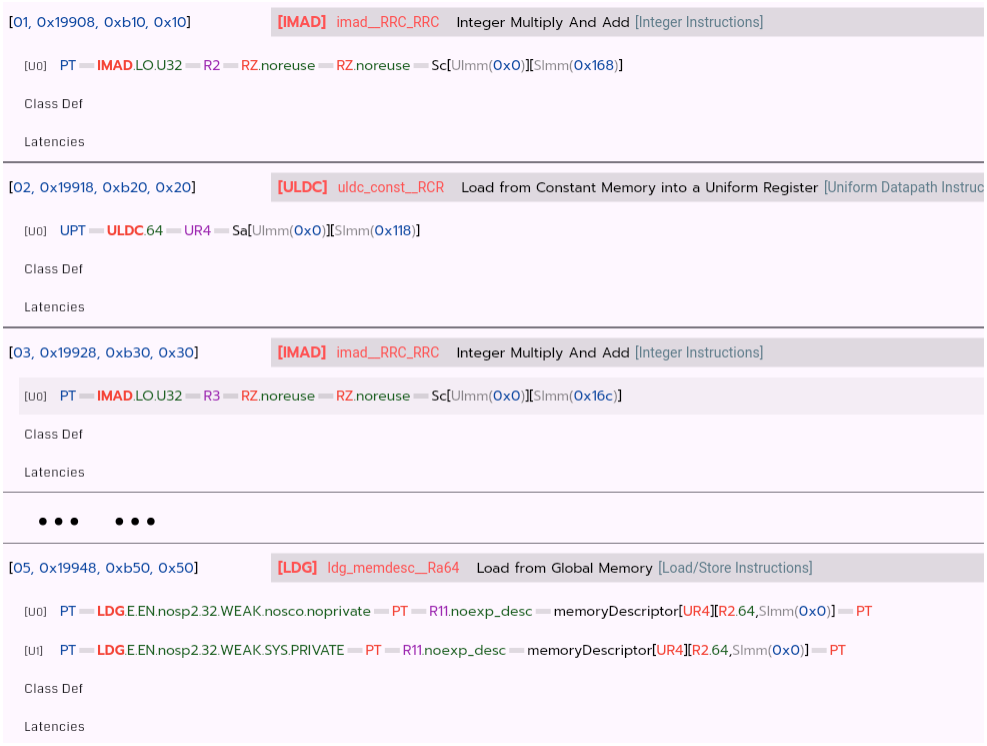


Figure 6.5: This argument is quite different from the one shown in Figure 6.3. Two **IMAD**, one **ULDC** and one **LDG** instructions are necessary. Note that **IMAD** loads address offsets 0x168 as well as 4 bytes later 0x16c.

For *kernelT3* we use three arguments: the first one is an 8 byte pointer and the other two *unsigned int* value types

```
1 // Pass three 32 bit unsigned integers, the first one using a pointer and
2 // the second and third ones by value
3 __global__ void kernelT3(unsigned int* a, unsigned int b, unsigned int c) {
4     printf("param a=%u, param b=%u, param c=%u\n", *a, b, c);
5     return;
6 }
```

Figure 6.6 shows the two value type *unsigned int* arguments. Argument two

is at offset 0x168 and argument 3 at offset 0x16c which is 4 bytes later. Thus, we can deduce that alignment follows the same rule set as we are used to from *C structs*: they are aligned with respect to their own size.

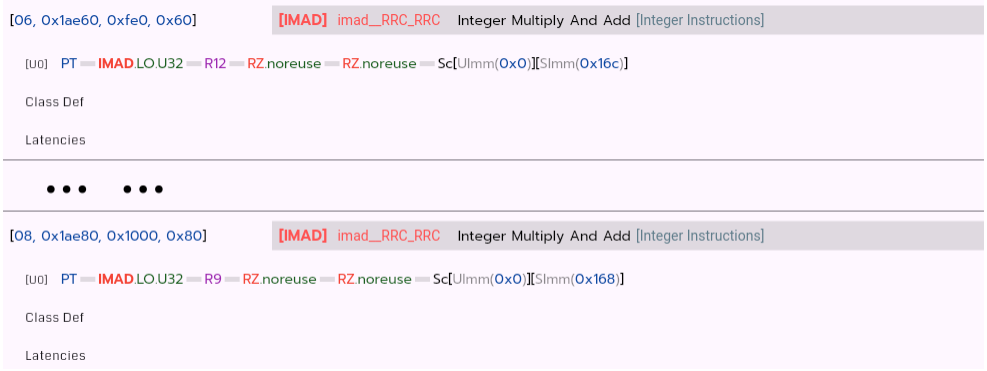


Figure 6.6: The first argument of *kernelT3* is a pointer and loaded in the same way as shown in Figure 6.5, the other two arguments are *4 byte value types*, aligned to 4 byte borders 0x168 and 0x16c.

Since we have to create a *template kernel* to modify anyways, we can always add code that uses the arguments and manually check at which addresses they can be loaded. This method is safe but tends to produce platform specific solutions.

One may ask: how can SASS not be platform specific? Up to SM 100 (Blackwell) instruction sets tend to be augmented with new things in newer SMs rather than have things removed. For example, **STG** and **LDG** variants that work on SM 75 also work on SM 80 and SM 86 and the same instruction patterns can be used. The extensions configurations may change, though, meaning that on SM 75 .64 may need different bits to be set.

For example: instruction class stg_uniform__RaRZ exists on both SM 75 and SM 86. On SM 75, the size extension configuration uses *register category* SIZE3 while on SM 86, it's SZ_U8_S8_U16_S16_32_64_128. In the following bit of register definition, we see that SIZE3 offers one more definition-slot .U.128 that is invalid on SM 86.

```
# SM 86
SZ_U8_S8_U16_S16_32_64_128 = {
   'U8':0, 'S8':1, 'U16':2, 'S16':3, '32':4, '64':5, '128':6, 'INVALID7':7
}
# SM 75
SIZE3 = {
   'U8':0, 'S8':1, 'U16':2, 'S16':3, '32':4, '64':5, '128':6, 'U.128':7
}
```

6.6 Generic Kernel Arguments

In this section we show a good technique how to pass values to kernels using the same sequence of arguments as presented in the introductory Section 6.1: *uint32, uint64*, uint64*, double*, uint64*, double* uint64*, float**. The full script showcased in this section is available with comments in Appendix A.5.

These instructions are in addition to the empty kernel from Section 6.4.

First we load the *uin32* type again. There are a multiple ways to load something from a constant memory offset. We check two ways using [ULDC] and [IMAD]

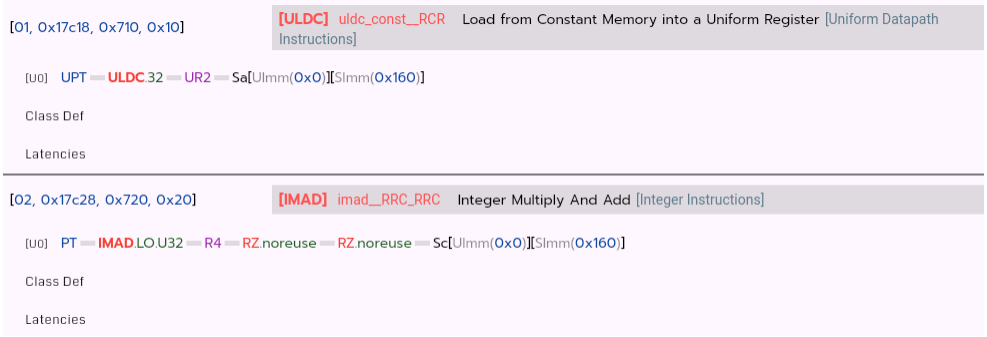


Figure 6.7:

[01] [ULDC], uldc_const__RCR in Figures 6.7 loads const memory location 0x0x0x160 into uniform register UR2. Note the size indicator .32. While this works and we get the value into UR2, uniform registers are usually refered to as 'additional data path' and used for addresses.

[02] [IMAD], imad__RRC_RRC in Figures 6.7 loads const memory location 0x0x0x160 into register R4. [IMAD] performs $a * b + c$. In this case we set a and b to **zero** using RZ constant register. This way is generally favored by the Nvidia compiler to load value type arguments. **It is important** to keep the size in mind: while **IMAD** has a .U32, it can only distinguish between .U32 and .S32. There is no 64 bit version. For larger types we have to load the subsequent register R7 as well. See Section 6.5 and Figure 6.4 for an example. Failing to do so will probably produce an **an illegal memory access was encountered**-CUDA error.

For new instructions, it is a good idea to always first use registers with multiple of 4 numbers, for example R4, R8, R12, etc. and should always check if an encoding is valid (see Section 3.2 and 6.3).

Next we load all pointer type arguments. They all contain 64 bit addresses so we use **ULDC**. Figures 6.8 and 6.9 load the base addresses of the remaining 7 arguments into all even numbered uniform registers UR4 to UR16.

Figure 6.8: Map the base addresses of the pointer arguments *uint64**, *uint64**, *double**, *uint64** types to uniform registers UR4, UR6, UR8, UR10.

Figure 6.9: Map the base addresses of the remaining pointer arguments *double* uint64**, *float** to UR12, UR14, UR16.

We can now load and store values into the arrays pointed to by the argument pointers. Note that the arrays have to be allocated with the proper CUDA procedures in C++.

For example, we can store the value in register R4 that was loaded in Figure 6.7 in instruction number [02] into the **second** position of the array with base address in UR4 using the **STG** variant in Figure 6.10 in instruction number [10] using the **relative offset** 0x8.

Figure 6.10: Store .32 bit value in register R4 at location address UR4 + 0x8.

In Figure 6.11 we move an *immediate value* into register R2 using the very convenient **MOV** instruction and then store that value into first position of the array with base address in UR4 at relative address offset 0x0. Incidentally, the array mapped to uniform register UR4 is the *control* argument. See the full script in Appendix A.5, line 30. It is a good idea to use one array argument for *control entries* where distinct values can be written at defined offsets to check if the customized kernel runs without errors or crashes with a generic CUDA error. It is also a good idea to use *adequately named variables* for registers since even small and customized kernels can have a lot of lines of code and one tends to loose sight of which registers do what. It is the same principle as using *adequate variables names* for other programs.

Figure 6.11: Move an immediate value into R2 using the **MOV** instruction and then store to global memory using **STG** with relative offset 0x0. Note that **UMOV** also exists and can move values into uniform registers.

After loading a value argument into a register and seeing how to load pointer addresses and store values into specific locations of an attached array, Figure 6.12 shows how to load a value from a mapped base address and store the value into another location, also using a mapped base address and a relative offset. Consulting Section A.5 around line 176 reveals that we are transferring a value from the argument *ui_input* to *ui_output*.

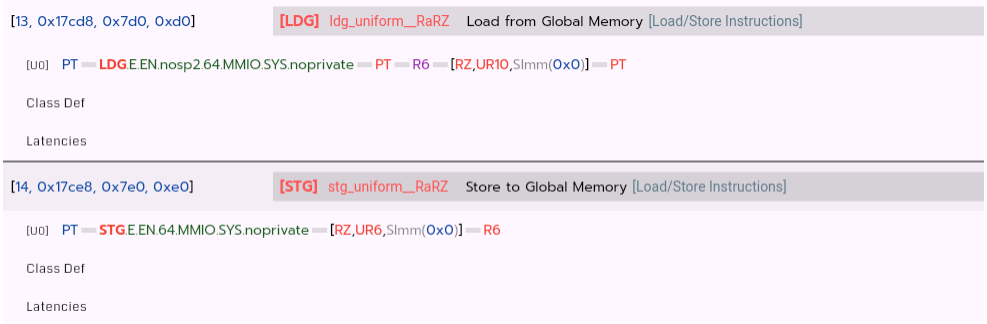


Figure 6.12: Load a value using the mapped base address of an array passed with a pointer argument and relative offset using **LDG** and storing it into a different array using **STG**.

This example uses the kernel template shown in Appending A.3. If we run it, for example on SM 86 using the input value *1*

Listing 6.1: Run the kernel outlined in this section with input value **1**.

```
$ ./tutorial_1_load_kernel_args_86 1
```

it produces the following output:

Listing 6.2: Print output after running the example in Listing 6.1. **Especially**, note the two labels **BeforeKernel** and **AfterKernel** that can be used to parse the output.

```
[KernelAddress]0x58d177ac0690[/KernelAddress]
   [LoopCount_0]
      [BeforeKernel]
         [Control]0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
         [UiOutput]0[/UiOutput]
         [UiInput]10001[/UiInput]
          ...
      [/BeforeKernel]
      [AfterKernel]
         [Control]815, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
         [UiOutput]10001[/UiOutput]
         [UiInput]10001[/UiInput]
          ...
      [/AfterKernel]
   [/LoopCount_0]
[/Kernel_0]
```

Examining the print output shown in Listing 6.2, Figures 6.10 and 6.11 are responsible for `[Control]0, 0, ... [/Control]` to `[Control]815, 1, ... [/Control]` using UR4 while Figure 6.12 changes `[UiOutput]0[/UiOutput]` to `[UiOutput]10001[/UiOutput]` using `[UiInput]10001[/UiInput]` using the base registers UR10 and UR6. All output produced by kernel templates as introduced in Appendix A.3 produce this kind of output that is then used to create benchmarking loops and

facilitate convenience. The main kernel template will be introduced in Tutorial 6.13.

6.7 Introducing Barriers WD, RD and REQ

This section explains how barriers are set and waited for in SASS using the same script as in the previous Section 6.6 available in Appendix A.5. The first thing we do is revisit Figure 6.12 and admit that it omits a very important detail: the **LDG** instruction has to set a **barrier** to $WR=0x0 to ensure that the subsequent **STG** waits until the load operation is finished. We achieve that by setting $REQ={0} in **STG** as illustrated in Figure 6.13.

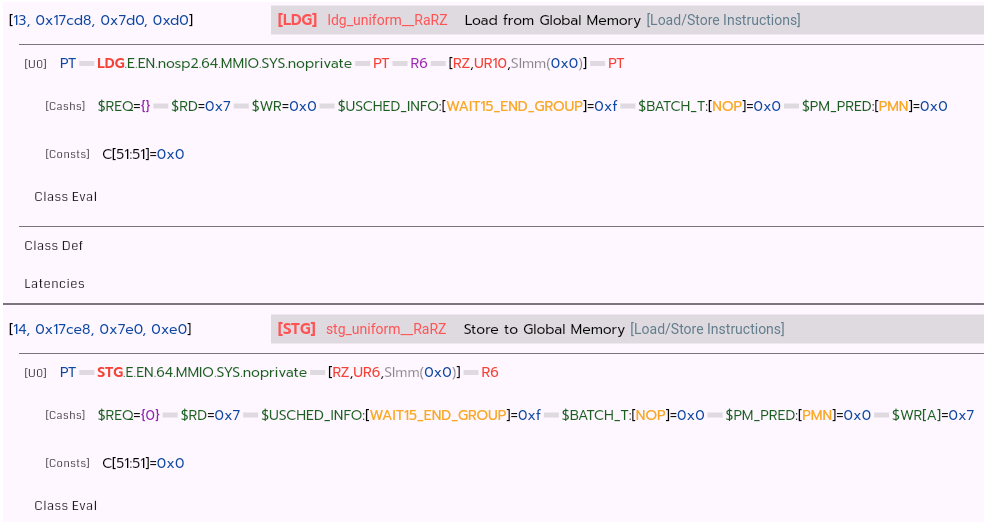


Figure 6.13: The same as Figure 6.12 with added cache lines. Note in instruction [13] in the [Cashs] line we set $WR=0x0 and in instruction [14] we set $REQ={0}.

If we don't set the **wait condition** to something that includes $REQ={0}, in Listing 6.2, the line `[UiOutput]10001[/UiOutput]` in the `[AfterKernel]` section will be the same as in the `[BeforeKernel]` section, meaning, we **don't wait for LDG to finish**!

Perhaps the reader recalls that earlier in Section 2.5.16 we mentioned that we can **wait** for up to 6 barriers

- $WR=0x0 to $WR=0x5
- $RD=0x0 to $RD=0x5

in one instruction using the $REQ bits and that we can set **at most** one $WR and one $RD barrier, preferably with two distinct values, if and only if the format definition (for example in Section 2.5.16, Listing 2.29) supports a barrier type.

It is **helpful** to set the $REQ bits using **binary numbers**. For example, if we like to wait for $WR=0x0 set by some random previous instruction, we can set $REQ = 0b000001. $WR=0x1 would be $REQ = 0b000010 and waiting for both needs $REQ = 0b000011. $REQ uses the BITMASK(6/0X0000) function. Every bit denotes one barrier to wait.

The notation in the VSCode decoder translates $REQ = 0b000001 to $REQ={0} and $REQ = 0b000011 to $REQ={1,0} and one full set of wait conditions will be $REQ={5,4,3,2,1,0}.

Barriers **need to be set** to have an influence in the program flow. By **default** they are **reset**. We could set $REQ={5,4,3,2,1,0} for every single instruction. If we didn't set any $WR or $RD it would not make a difference. For example, $WR=0x0 **sets** barrier enumerated with 0x0. The **next** $REQ={0} will **reset** the barrier after it has been **signaled** by the instruction that **set** it.

One more thing: cudaDeviceSynchronize() stalls until **all** transfers are completed. Including those that don't use a barrier but should, for correctness.

6.8 Dealing with Floats, Doubles and MUFU

This section introduces an example using floats, doubles and the *multifunctional unit* **MUFU**, performing a division. It builds on the kernel shown in Section 6.6 to map the kernel arguments to uniform registers and store some *control* values in the control array. We also store the intermediate results of the calculation in the float and double output arrays at consecutive positions such that we can build the calculation in steps and check that the respective newest instruction produces the expected result but only show the instructions that matter. The script for this example is available in Appendix A.6.

First we need so see how we can produce float and double values in a kernel. Figure 6.11 uses an **MOV** instruction to get one *integer* immediate value into the register R2 such that it can then be used by the following **STG** instruction. If the only goal is benchmarking variable latency instructions that have a barrier, we can use the same **MOV** also for floating point instructions as well, since the only thing we care about is checking when the *barrier* gets signaled.

Of course, we can always pass a float value as argument to the kernel. For this example, though, we can only pass integers, perhaps because we exhausted the 8 available, uncomplicated argument slots for the kernel template or because we want to benchmark *fixed latency instructions* or *variable latency instructions* without using the barrier mechanism, for which we must know the result of a calculation in advance (see Section 7.3.1) but we do not want to change the kernel template.

Following we compute **a/b** and return the result as 32 and 64 bit floats.

First we load *a* that we pass as *unsigned int* as first argument to the kernel into register R4.

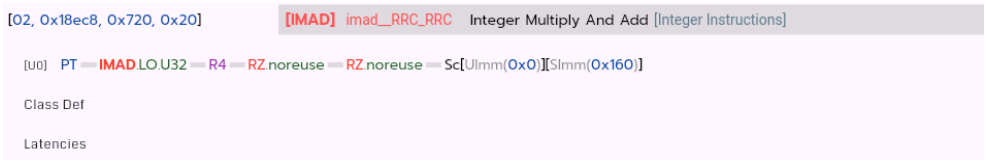


Figure 6.14:

Now we use **I2F** to convert the *unsigned int* value in R4 into one 32 and one 64 bit float into R6 and R8 respectively.

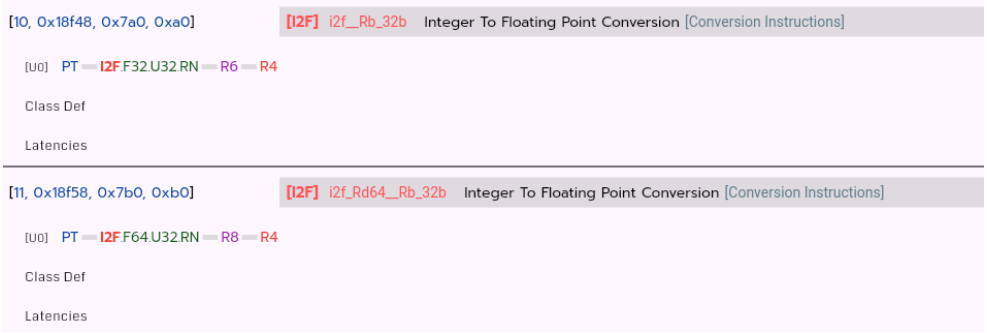


Figure 6.15:

Next, we load *b* that we pass in first position of the *uint64* array mapped to UR10 into R14 and converti it to both a 32 and a 64 bit float into R10 and R12. We don't actually use R12 in the **MUFU** unit but we store it back to global memory to check the conversion.

Figure 6.16:

The **MUFU** unit can calculate the reciprocal value of 16 and 32 bit floats. We use it to invert the 32 bit float in R10 and store it in R16.

Figure 6.17:

Since we only have the inverse as 32 bit float but we like to use **DMUL** to multiply with the 64 bit version of *a* stored in R8, we *upscale* it to a 64 bit float and store the result in register R18

Figure 6.18:

Now we can finally use **FMUL** and **DMUL** to calculate the product of *a* and *1/b* for the final division. The output is shown in Listing 6.3.

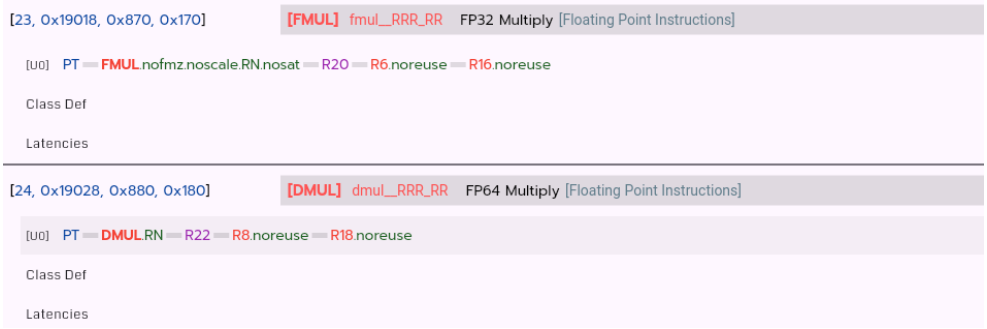


Figure 6.19:

Listing 6.3: The output of the example shown in Figures 6.14 to 6.19 using a=10 and b=3, calculating 10/3=3.33. FOutput contains the 32 bit float result and DOutput the 64 bit result. In third position is the reciprocal value calculated using MUFU.

```
[KernelAddress]0x612b13f34b50[/KernelAddress]
   [LoopCount_0]
      [BeforeKernel]
         [Control]0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
         [UiOutput]0, 0, 0, 0[/UiOutput]
         [DOutput]0, 0, 0, 0[/DOutput]
         [FOutput]10001, 10002, 10003, 10004[/FOutput]
         [UiInput]3, 10002, 10003, 10004[/UiInput]
         [DInput]10001, 10002, 10003, 10004[/DInput]
         [ClkOut1]999999998, 999999998, 999999998, 999999998[/ClkOut1]
      [/BeforeKernel]
      [AfterKernel]
         [Control]815, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
         [UiOutput]0, 0, 0, 0[/UiOutput]
         [DOutput]10, 3, 0.333333, 3.33333[/DOutput]
         [FOutput]10, 3, 0.333333, 3.33333[/FOutput]
         [UiInput]3, 10002, 10003, 10004[/UiInput]
         [DInput]10001, 10002, 10003, 10004[/DInput]
         [ClkOut1]999999998, 999999998, 999999998, 999999998[/ClkOut1]
      [/AfterKernel]
   [/LoopCount_0]
[/Kernel_0]
```

6.9 Measure a Clock to Examine USCHED_INFO

This tutorial loads the arguments of a kernel template as shown in Section 6.6 and then calculates a few clock cycles, showing how to add and subtract as well as impressively showing the uselessness of measuring clock cycles for fixed latency instructions by measuring how many *clock cycles* it takes to measure *clock cycles* using $USCHED_INFO register values WAIT1 to WAIT15. We see that after waiting *enough* cycles, we get the correct result while the

clock cycle measurement always shows exactly the number of wait cycles we use for $USCHED_INFO.

The full script generating the customized kernel is available in Appendix A.7. Note that it uses a template not featured in this report but that is available in the code base.

Listing 6.4: Assuming the generated name for the tutorial binary is tutorial_3_clocks_86, use this bash line to run the tutorial binary. The frist argument is the *input size* of 15 because we use WAIT1 to WAIT15. The second argument is not used, thus we use 1. Then follow 15 zeros. These zeros initialize the `ui_input` array with 15 zeros.

```
..$ ./tutorial_3_clocks_86 15 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

To be certain that we produce well defined output, we must use **MOV** with an immediate value 0x0 to initialize all used registers (even **and** odd numbered ones). Figure 6.20 shows five **MOV** instructions with the first one unfolded that moves 0x0 into R6. The following five **MOV** do the same with R7 to R11.

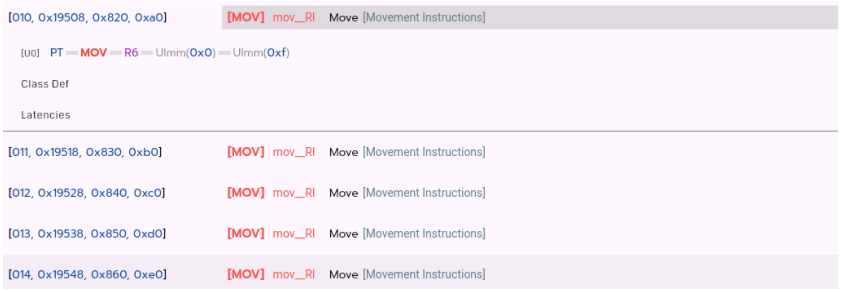


Figure 6.20: Ensure that all used registers (even and odd numbered) contain *zero* initially.

In Figure 6.21 we measure two clocks using special register SR_CLOCKLO. We repeat Figures 6.21 and 6.22 **15** times in sequence.

Instruction **CS2R** first appearing at [015] uses $USCHED_INFO = WAIT1_END_GROUP. The next incarnation will use $USCHED_INFO = WAIT2_END_GROUP, etc ... up to $USCHED_INFO = WAIT15_END_GROUP for the last incarnation, depicted in Figure 6.22. As we will see when we discuss the output of this kernel, this has some surprising influences on the behavior of the kernel.

[015] **[CS2R]**, cs2r_: measure the first clock into R6. Note that this instruction uses $USCHED_INFO = WAIT1_END_GROUP.

[016] **[IADD3]**, iadd3_noimm__RRR_RRR: perform an addition operation using the first measured clock. $USCHED_INFO = WAIT1_END_GROUP in instruction [015] means, that the **instruction dispatcher** should wait

1 cycle before dispatching the next instruction. Since CS2R may not be finished after 1 cycle, the resulting value in R6 may be invalid. Since we want to capture this effect to find out exactly how many cycles we must wait for CS2R to conclude, we put an IADD3 at exactly this place using the output register of CS2R. We **cannot** put IADD3 in any other place. The more we move IADD3 away from CS2R, the more *cycles will pass* between those instructions. This is **latency hiding** at work for **fixed latency instructions**. If we put IADD3 right after the CS2R there is no latency hiding. We must wait the minimum cycle count that CS2R requires to complete.

[017] [CS2R], cs2r_: measure the second clock into R8

[018] [IADD3], iadd3_noimm__RRR_RRR: compute the difference R8 - R6 - 15. Since we are interested in the cycle count for CS2R, we can subtract a fixed value of 15 which is exactly the $USCHED_INFO value of instruction IADD3 at location [016].

Figure 6.21: This is the **first** of 15 instruction *bundles* that measures two clocks around one IADD3 instruction and finally computes the difference. Note that instruction [015] uses $USCHED_INFO = WAIT1_END_GROUP. Figure 6.22 shows the last usage.

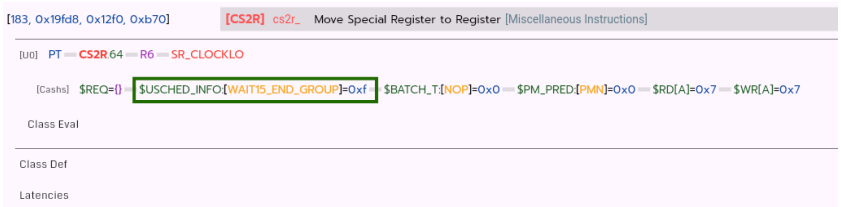


Figure 6.22: This is the **last** of 15 instruction *bundles* that measures two clocks around one IADD3 instruction and finally computes the difference. Note that instruction [015] uses $USCHED_INFO = WAIT15_END_GROUP. Figure 6.21 shows the first usage.

Listing 6.5: The output is divided into two sections `BeforeKernel` and `AfterKernel` to compare between input and output values. Listing 6.4 shows the bash call for this output. We note *several* things, starting with `ClkOut1`. It is initialized with 999999998. In the output section we see the progression (2, 2, 3, ..., 15). It seems that both WAIT1 and WAIT2 produce a measured cycle count of 2. After that the numbers always match the used WAITx. The two fields `UiOutput` and `UiInput` are described in Figure 6.23. They both should contain the same value since they both contain the *output* of the first **CS2R** instruction in line 015. We note that the first 5 entries of `UiOutput` are all **zero**! From position 6, `UiInput` and `UiOutput` match again with value 1170912518. This means that for **CS2R** to conclude, it requires at least WAIT6. In `ClkOut1` we measured (2,2,3,4,5) before measuring 6, all of which are effectively *wrong* measurements. The game outlined in this Tutorial can effectively be repeated with any SASS instruction.

```
[BeforeKernel]
  [UiOutput]0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/UiOutput]
   [UiInput]0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/UiInput]
   [ClkOut1]999999998, 999999998, 999999998, 999999998, 999999998,
            999999998, 999999998, 999999998, 999999998, 999999998,
            999999998, 999999998, 999999998, 999999998, 999999998[/ClkOut1]
[/BeforeKernel]
[AfterKernel]
  [UiOutput]0, 0, 0, 0, 0, 1170912518, 1170912689, 1170912861,
            1170913034, 1170913208, 1170913383, 1170913559, 1170913736,
            1170913914, 1170914093[/UiOutput]
   [UiInput]1170911677, 1170911844, 1170912011, 1170912179, 1170912348,
            1170912518, 1170912689, 1170912861, 1170913034, 1170913208,
            1170913383, 1170913559, 1170913736, 1170913914, 1170914093[/UiInput]
   [ClkOut1]2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15[/ClkOut1]
[/AfterKernel]
```

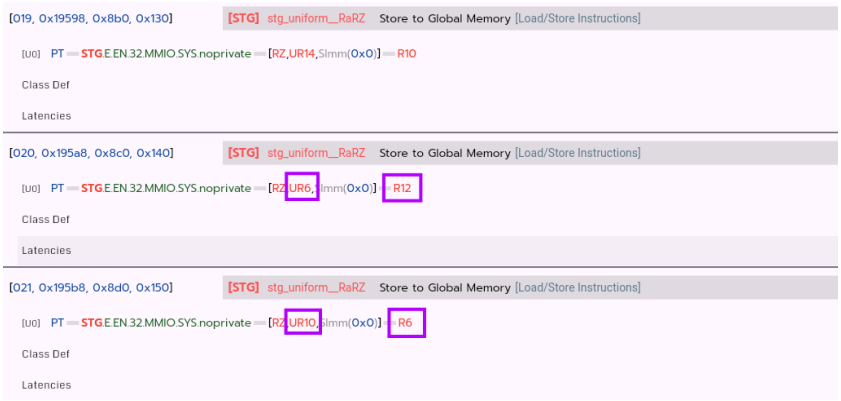


Figure 6.23: The **STG** instructions that store the measured clock values from Figure 6.21. Note that R12 and R6 are stored into UR6 and UR10 respectively and recall that in Figure 6.21 R12 is the destination register for **IADD3**, storing the output of the first **CS2R** before the instruction necessarily concludes. UR6 is `UiOutput` and UR10 is `UiInput`.

Listing 6.5 explains in detail what the output of this kernel means. From that we must conclude that using simple `clock()` in any CUDA kernel (that maps

directly to **CS2R** reading SR_CLOCKLO will yield wrong measurements because the measured cycle count does not correspond to the latency of the instruction but to the value in $USCHED_INFO. Checking out any NVCC compiled CUDA kernel, for example in Figure 6.24 will inform the reader that NVCC rarely uses large WAITx values. In fact, it seems to aim at spreading out instructions that are linked by data dependencies as much as possible.

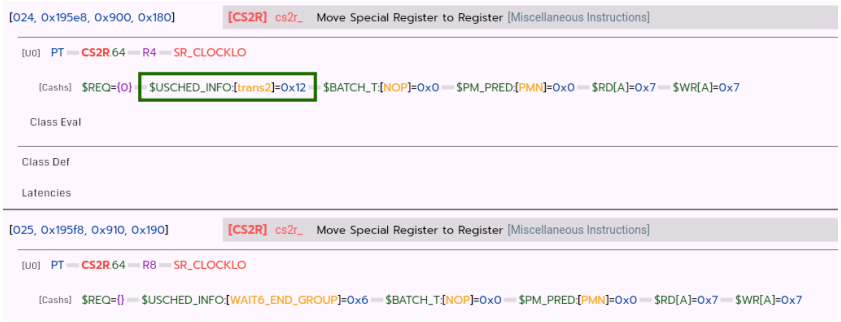


Figure 6.24: Sample of **CS2R** out of an NVCC compiled kernel.

The insight in this Tutorial has one upside though: we can now for all fixed-latency instructions as shown in Figure 2.14, simply use the value in $USCHED_INFO instead of having to benchmark the instructions. As we will see in Section 7.3 it is possible to bulk-benchmark variable-latency instructions as well, further reducing the microbenchmarking burden to the reader.

6.10 Instruction Chaining

This tutorial introduces *chaining*. This is a mechanism with which *predicate* instructions can be chained with one logical operation. For example, the first **ISETP** calculates condition 1 into P0, then the second **ISETP** instruction calculates the second condition into P0 using the previous P0 as additional input, connected with a logical operation. This is helpful if we like to calculate `condition 1 AND condition 2` and use only two **ISETP** instructions instead of two **ISETP** and one **PSETP** (Combine Predicates and Set Predicate) instructions.

The full script for this tutorial is in Appendix A.8.

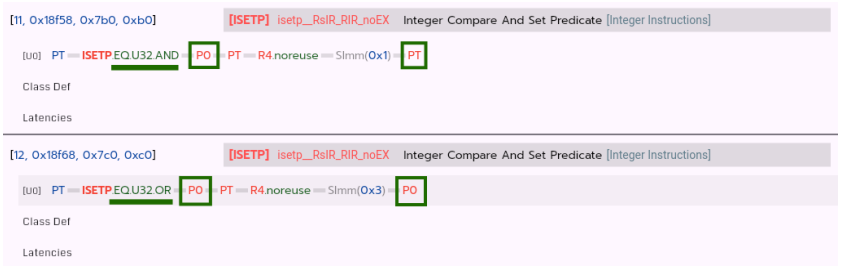


Figure 6.25: Calculate $a == 1 || a == 3$ and store the result in P0. [11] calculates $a == 1$ and stores the result in P0. Note that the instruction uses PT as *chaining* input combined with the *chaining* operation .AND. Then [12] calculates $a == 3$ and *chains* the result with the current P0 using an .OR extension configuration.

Figure 6.26: Calculate $a == 0 || a == 2$ and store the result into P1. Chaining works exactly the same as in Figure 6.25.

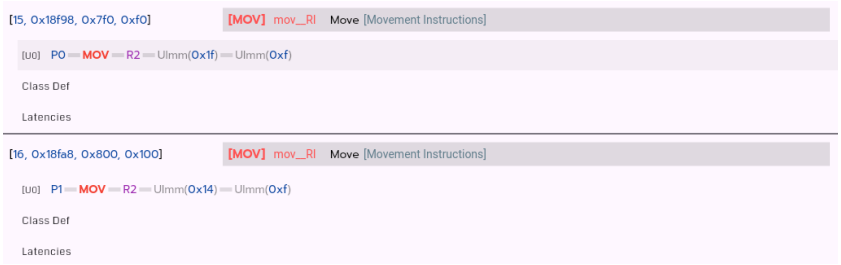


Figure 6.27: Use predicates P0 and P1 to selectively **MOV** either the value 0x1f = 31 or 0x14 = 20 into R2.

After the instruction in Figure 6.27, we write R2 into the second position of the control array we pass as an argument, using **STG**. We can now examine the output of the customized kernel using the inputs 0, 1, 2 and 3. The expected outcome is 21 for inputs 1 and 2, and 31 for the inputs 3 and 1 in Listing 6.6.

Listing 6.6: Examine the output of the customized chaining kernel using inputs 0, 1, 2 and 3. For clarity we only show the bash command and the `Control` output. If we run the kernel with another input, for example 42, we get 99 as output. The knowledgeable reader will recognize this number as a Star Wars Easter Egg.

```
..$ ./tutorial_4_isetp_chaining_86 1 0
   [Control]815, 20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
..$ ./tutorial_4_isetp_chaining_86 1 1
   [Control]815, 31, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
..$ ./tutorial_4_isetp_chaining_86 1 2
   [Control]815, 20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
..$ ./tutorial_4_isetp_chaining_86 1 3
   [Control]815, 31, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]

We secretly initialized R2 with the value 99...
..$ ./tutorial_4_isetp_chaining_86 1 42
   [Control]815, 99, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[/Control]
```

6.11 Creating a Loop

Since it is not practical to just increase the number of instructions in the kernel template indefinitely, we require a loop construct, for example, to perform multiple tests in one customized kernel. This tutorial shows a good way to do this. The full script is in Appendix A.9.

Since we have seen plenty of examples of custom CUDA instructions, this tutorial will focus on the *Python* code. Note that we initialize the input arguments as usual, as explained in Tutorial 6.6.

The first step is to initialize an index variable. It is practical to use **IADDR** for this since it already provides space for modification, for example if the reader needs to calculate the starting value of the index using additional sources.

```
1 # Init a_iter_R6 to Zero
2 ii = sc.SASS_KK__IADD3_NOIMM_RRR_RRR(kk_sm,
3                                      a_iter_R6,
4                                      negate_Ra=False, src_Ra=RZ,
5                                      negate_Rb=False, src_Rb=RZ,
6                                      negate_Rc=False, src_Rc=RZ,
7                                      usched_info_reg=wait15)
8 target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
9 i+=1
```

This is **One Of Two** of the most important locations in this tutorial: how do we calculate the address offset to jump back to the beginning of the loop

without having to modify the entire code if we change some details? We can use the instruction index `i_start = i`. If we store this index and make the jump back to the beginning of the loop dependent of this stored index, we can add an arbitrary amount of instructions inside the loop without breaking it.

```
# This is the starting point of the loop. Use
# the instruction index i to calculate the jump size
i_start = i
```

We can use **IADD3** to increment the index. Note that we use the *immediate value* version of **IADD3** this time. After incrementing the index variable we use **ISETP** into the predicate register P0 to check if it is still smaller than the maximal iteration index. In this case, we use the kernel input variable a that is in first place of the kernel arguments and can be passed from the command line.

```
# Increment the loop variable by 1: use Ra = Ra + 1
ii = sc.SASS_KK__IADD3_IMM_RsIR_RIR(kk_sm,
                                    a_iter_R6,
                                    negate_Ra=False, Ra=a_iter_R6,
                                    src_imm=1,
                                    negate_Rc=False, Rc=RZ,
                                    usched_info_reg=wait15)
target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
i+=1

branch_pred = kk_sm.regs.Predicate__P0__0
ii = sc.SASS_KK__isetp__RRR_RRR_noEX(kk_sm,
                                pred_invert=False, pred=PT,
                                Pu=branch_pred,
                                Ra=a_R4,
                                icmp=kk_sm.regs.ICmpAll__GT__4,
                                Rb=a_iter_R6,
                                fmt=kk_sm.regs.FMT__S32__1,
                                usched_info_reg=wait15)
target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
i+=1
```

This is **Two Of Two** of the two most important locations in this tutorial: store the final instruction index before the loop ends in `i_end` and use it in the **BRA** instruction later to calculate the size of the jump back. The

knowledgeable reader will have detected two Star Trek Easter Eggs at this point of the tutorial. Congratulations!.

```
# Use the current instruction index to calculate the jump
# size!
i_end = i
```

Use the same predicate registerP0, this time as instruction predicate. The **BRA** instruction will be executed as long as P0 contains the equivalent of *True*. The jump index in *imm_val* is calculated as `int(-16*(i_end - i_start + 1))`. Each instruction requires 16 bytes and we must jump the difference between `i_end` and `i_start` **and** add 1 instruction to include the **BRA** instruction.

```
ii = sc.SASS_KK__BRA(kk_sm,
                     pred_invert=False, pred=branch_pred,
                     Pp_invert=False,
                     Pp=PT,
                     imm_val=int(-16*(i_end - i_start + 1)),
                     usched_info_reg=wait15)
target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
i+=1

# Write the iteration variable into second place of the
# control register
ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
                          offset=0x8,
                          source_reg=a_iter_R6,
                          usched_info_reg=wait15,
                          req=0x0, rd=0x7,
                          size=32)
target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
i+=1
```

6.12 NOP fillers

If we want to benchmark SASS instructions that may take more than 15 cycles, we need a practical mechanism to construct custom kernels to *fill in* additional cycles because $USCHED_INFO will only take us up to 15 cycles. For this, we can use **NOP** instructions.

Note that this Tutorial merely outlines the technique and there is no full example available. The technique is however used in the benchmarking

Section 7.3. In the following *Python* code, the variable `wait_cycles` is passed as parameter, indicating how many *wait cycles* need to be added.

```
1  # Add NOP instructions that wait for a given number of cycles.
2  # NOTE that with WAIT1, the actual wait is 2 cycles.
3  # This is consistent accross all Nvidia GPUs.
4  if wait_cycles > 15:
5      ii = sc.SASS_KK__NOP(kk_sm,
6          usched_info_reg=
7              getattr(kk_sm.regs, 'USCHED_INFO__WAIT{0}_END_GROUP__{0}'
8                  .format(min(15, wait_cycles-15))))
9      target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
10     i+=1
11 if wait_cycles > 30:
12     ii = sc.SASS_KK__NOP(kk_sm,
13         usched_info_reg=
14             getattr(kk_sm.regs, 'USCHED_INFO__WAIT{0}_END_GROUP__{0}'
15                 .format(min(15, wait_cycles-30))))
16     target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
17     i+=1
18 if wait_cycles > 45:
19     ii = sc.SASS_KK__NOP(kk_sm,
20         usched_info_reg=
21             getattr(kk_sm.regs, 'USCHED_INFO__WAIT{0}_END_GROUP__{0}'
22                 .format(min(15, wait_cycles-45))))
23     target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
24     i+=1
25 if wait_cycles > 60:
26     ii = sc.SASS_KK__NOP(kk_sm,
27         usched_info_reg=
28             getattr(kk_sm.regs, 'USCHED_INFO__WAIT{0}_END_GROUP__{0}'
29                 .format(min(15, wait_cycles-60))))
30     target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
31     i+=1
```

Note that `kk_sm.regs` points to an **SM_CuBin_Regs** class. Thus, we can use *Python*'s `getattr` to get the correct WAITx in code lines 7, 14, 21 and 28.

This technique can be used to produce **multiple** kernels, each one with a longer wait time at the required location. With the code base for this report it is not possible to change the value in $USCHED_INFO for an instruction at runtime. Thus the *programming pattern* is to create multiple CUDA binaries and run them in sequence, testing each ones result for correctness.

6.13 Kernel Template for Flexible Benchmarking

Finally, we can use all knowledge collected in this tutorial chapter to construct a massively flexible, custom CUDA kernel with which we can construct single benchmarks to measure clock cycles of single instructions with fixed and variable latency (using barriers and without), and even bulk-generate benchmarks with 1000 kernels in one single CUDA binary, spanning hundreds if not thousands of binaries.

This template is used in all benchmarks in the benchmarking Section 7.3.

Before we load all kernel arguments as we did in in almost all the tutorial up to this point, this time, we do something special: we put an **CS2R** instruction at the very beginning of the kernel in Figure 6.28. This opens up the possibility for more precise clock cycle measurements.

```
[00, 0x17c88, 0x700, 0x0]   [CS2R] cs2r_  Move Special Register to Register [Miscellaneous Instructions]
[01, 0x17c98, 0x710, 0x10]  [MOV] mov_RC  Move [Movement Instructions]
  [U0]  PT = MOV = R1 = Sb[Ulmm(0x0)][Slmm(0x28)] = Ulmm(0xf)
```

Figure 6.28: At the very top of the kernel, we put an **CS2R** instruction, getting the first cycle count. Usually for SM 86, a **MOV** instruction as shown here in position [01] is at the beginning.

After the two initial instructions in Figure 6.28 we load all kernel arguments as outlined in Tutorial 6.6.

```
[02, 0x17ca8, 0x720, 0x20]  [ULDC] uldc_const_RCR  Load from Constant Memory into a Uniform Register [Uniform Datapath Instructions]
  [U0]  UPT = ULDC.64 = UR2 = Sa[Ulmm(0x0)][Slmm(0x160)]
  Class Def
  Latencies
  . . .
[10, 0x17d28, 0x7a0, 0xa0]  [ULDC] uldc_const_RCR  Load from Constant Memory into a Uniform Register [Uniform Datapath Instructions]
  [U0]  UPT = ULDC.32 = UR16 = Sa[Ulmm(0x0)][Slmm(0x198)]
  Class Def
  Latencies
[11, 0x17d38, 0x7b0, 0xb0]  [ULDC] uldc_const_RCR  Load from Constant Memory into a Uniform Register [Uniform Datapath Instructions]
```

Figure 6.29: Load all kernel arguments. See Appendix A.3 for a layout of the arguments.

Next follows the interesting part of this kernel template in Figure 6.30: we use **CS2R** instructions as *boundaries* for **slots** where we can insert any instructions we may need. Figure 6.30 shows an instantiation of the kernel template, ready to *benchmark* a **DFMA** instruction (blue box).

Figure 6.30: The red boxes mark the **CS2R** boundary instructions. The instructions in the purple boxes numbered 1 trough 4 serve as **slots** to insert instructions used in different **stages** of the kernel.

Next to the **main slot** that contains the main instruction (in this case **DFMA**) the kernel features 4 slots where instructions can be inserted. Note that slots 2, 3 and 5 are within the *benchmarking loop*:

1 `pre_clock_prequels`: The first two **CS2R** instructions in Figure 6.30 frame the *prequels* frame. It contains all instructions we want to run *outside* of the benchmarking loop and outside of any clock measurement that impacts our benchmarking results. We call this slot `pre_clock_prequels` and use it to initialize any registers or other resources we may require inside of the benchmarking loop. In Figure 6.30 we use two **I2F** to initialize floating point values.

2/3, **5** `post_clock_prequels/sequels` and `main_class`: The next two **CS2R** instructions frame the *benchmarking section* of the kernel. It includes the main instruction as well as two groups of instructions before and after: the slot nr 2 contains the *post clock prequel instructions* that run after the first **CS2R** but before the main instruction. Recall that this slot is within the *benchmarking loop*. Thus, only fixed-latency instructions should be used such that we can deal with the increased number of cycles by removing them. In Figure 6.30 we use two **MOV** instructions to write zero into the destination register of the main instruction **DFMA** in each loop iteration. The slot nr 3 contains the *post clock sequel instructions* that run after the main instruction but before the second **CS2R** that collects the second clock cycle measurement for the main instruction

benchmark. In Figure 6.30 this slot is empty. But it can be filled, for example, if we benchmark instructions that may take longer than 15 cycles, with an array of **NOP** instructions as we did in Tutorial 6.12.

4 `post_clock_sequels`: The last slot is outside of the *benchmarking loop*. The instructions within it run after the second **CS2R** collects the second clock cycle measurement we require to complete one benchmarking loop. This slot is useful for writing the *result* of the main instruction global memory and, if we used any fixed-latency instructions in the `post_clock_prequels` slot, subtract their latencies from the total measured cycle count. In Figure 6.30 we use one **STG** instruction to write the result of the benchmarked **DFMA** to global memory and one **IADD3** to subtract 30 cycles from the second **CS2R** clock cycle count such that the two **MOV** instructions we used inside the `post_clock_prequels` section are negated. Note that the kernel features a special target array and target registers for measured clocks and it is not necessary to handle this part manually if one sticks to the laid out pattern. How the **IADD3** is able to change the measured clock cycles of the second **CS2R** will become clear in the benchmarking Section 7.3.

Figure 6.31 facilitates the *benchmarking loop* with an **ISETP** and a **BRA** instruction that together compose a loop as shown in Tutorial 6.11. As shown in that tutorial, the kernel template demonstrated in this tutorial uses the *flexible* offset calculation for the loop, too.

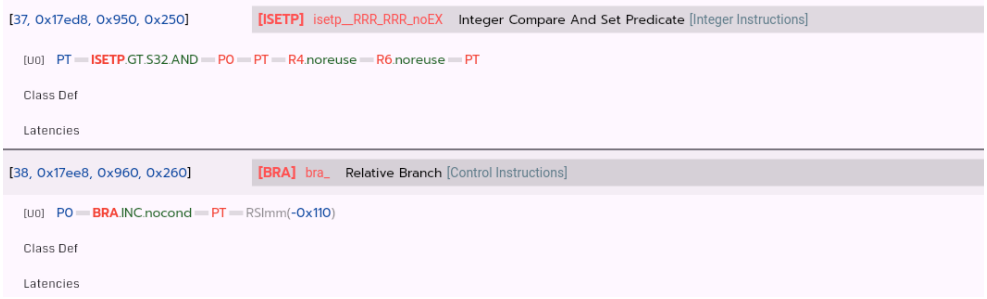


Figure 6.31: The last two relevant instructions for the kernel template with slots, introduced in this tutorial.

Recall that in Figure 6.30 we mentioned, that there is *no need* to handle the results of the **CS2R** instructions manually. Since all the **CS2R** instructions are *part of the template*, their results are written into prespecified registers. After the instructions in Figure 6.31, the required differences are calculated and stored in the clock measurement output array. For example, Listing 6.5 shows a fairly complete output such as is produced by this section's kernel

template. Note the field `ClkOut1`. This template uses the same mechanism to return results.

Figure 6.32 shows the final relevant instructions for this kernel template: they increment the destination registers for all operations for convenience. The simplest way to do this is in fact to move the mapped address in the *uniform registers* by the respective bit size.

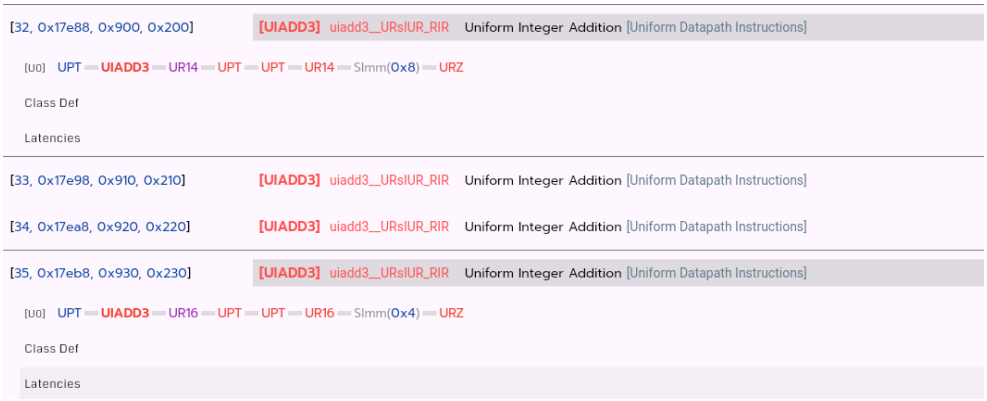


Figure 6.32: These instructions *increment* the destination *uniform registers* used to map the input and output kernel arguments. Note how instruction [32] increments the offset by 0x8 while instruction [35] only increments by 0x4. The reason is that UR14 represents a 64 bit data type and UR16 a 32 bit data type.

This tutorial concludes this chapter. In the next Chapter 7 we will use this kernel template extensively to introduce three different benchmarking techniques in Section 7.3 where we will also show some necessary *Python* code to use it effectively.

Chapter 7

Results and Discussion

7.1 Formalizing SASS

In Chapter 2, we were able to fully formalize Nvidia's SASS instruction sets for SM 50 (Maxwell) up to and including SM 120 (Blackwell) using the text files provided by DocumentSASS [13] extracted from Nvidia's *cuobjdump* tool. We verify that all documented and several undocumented SASS instructions are included and provide a *Python* module *py_sass* that implements all concepts laid out in Chapter 2.

As such, we formalize the concept of *instruction class* and with the module *py_sass* provide a parser that abstracts all SASS instructions into a *Python* data structure, allowing unprecedented access to all formal parts of the SASS instruction set.

Using this abstract *Python* data structure, it is possible to extract patterns from the entire set of instructions that are useful for further investigations and benchmarking production. For example, we are able to conclusively split all SASS instructions into variable- and fixed latency instructions as well as distinguish between data-, control- and memory-accessing instructions and provide mechanisms to benchmark all of them.

7.2 SASS Instructions Decoding, Encoding, Generation and Visualization

Using the *Python py_sass* module, in Chapter 3, Section 3.3 we discover a *fingerprint* with which we are able to decode every SASS instruction down to the level of their individual *instruction class* and their *alternates* (see Section 2.4.2). This is the most precise decoding that is possible, since there is no formally lower level of definition.

In addition, in Section 3.2 we discover the global *encoding* mechanism utilized to encode all SASS instructions and provide a template to easily do so using *Python*.

To finalize this section of the results, in Section 3.4, we create an *instruction generator* that represents the *entire* space of valid SASS instructions for SMs 50 to 120 and allows sampling this space uniformly at random using Monte Carlo technique.

With this *instruction generator* we are able to show that our encoding and decoding algorithms indeed do decode and encode all existing SASS instructions beyond reasonable doubt.

In Chapter 5 we implement a decoding scheme for CUDA binaries that can extract the CUDA instructions from a C++ binary the same way that *cuobjdump -sass* does but is also able to reassemble the extracted instructions into a working binary.

In addition to being able to connect every SASS instruction in arbitrary CUDA kernels with their most basic, formal definition, which allows maximal information extraction, this also enables a process where one can use a CUDA kernel template file and replace all instructions with own, custom SASS instructions, created first with the *instruction generator* and modified to specification, reassembling the modified kernel into a runnable binary file and in this way provide a large amount of freedom to create custom CUDA kernels that could not be created using NVCC. We provide a substantial selection of tutorials in Chapter 6, introducing the reader to this process.

All mechanisms described in this section are included in the provided *Python* module *py_cubin*.

Further, we provide a VSCode extension based on the framework *Flutter* that can open and, using a server provided by *py_cubin*, visualize all decoded SASS instructions of CUDA kernels in the greatest detail possible, visualizing all components of a SASS instruction as opposed to *cuobjdump -sass* and *nvdisasm* that only decode the instruction bits down to the level of the operation code (for example **MOV** or **DFMA**), omitting large portions of the SASS instruction bits, especially the bits related to the scheduler and the barriers.

Only starting with CUDA 12.8, Nvidia provides insight into the barriers and the scheduler bits via *cuobjdump -sass* for Blackwell architecture only.

7.3 Benchmarking

This section describes three benchmarking approaches to benchmark

- fixed latency instructions
- variable latency instructions using their barrier mechanism
- variable latency instructions not using their barrier mechanisms
- bulk-generating benchmarking kernels to cover a large range of the SASS instruction set at once

All featured benchmarks and tests were executed on a PC with

- 32 GB main memory
- AMD Ryzen 7 5800X 8-Core Processor
- Nvidia RTX 3080 (GA102, SM 86)
- CUDA 12.8
- Ubuntu 24.04.2 LTS

Note that the results presented are instances of the techniques developed based on this work. As such, any presented example is applicable to many other instructions with very little effort and presenting results for all of them would require showing hundreds of thousands of plots and tables. Thus in the following sections, the techniques are outlined with sample results and it is **stressed** that all the results are reproducible using calculated CUDA binaries available with this work that can at any time be run again with variable loop count, provided a GPU supporting SM 86 is available. All results for the bulk-generation technique outlined in Section 7.3.5 as well as all the calculated CUDA kernels are available in full with the source code of this report contained in the *PySASSCreate* module in the *benchmark_results* subfolder.

All tests were performed with single threaded kernels and with no load to speak of otherwise present on the GPU. Thus effects such as *resource contention* were not captured.

7.3.1 Fixed and Variable Latency Instructions Using a Loop

Fixed latency instructions encompass for example all single precision floating point and integer instructions (**FMUL**, **FADD**, **FFMA**, ..., **IMUL**, **IADD3**, **IMAD**, etc. Overall, this is the largest group of instructions, spanning roughly 2/3 of the instruction set.

Variable latency instructions include all double precision floating point instructions, for example **DMUL**, **DFMA**, **DADD**, etc. as well as as memory accessing instructions such as **STG** and **LDG**.

Note that the *Python* module *py_sass* provides a full categorization of all instructions for a given SM number.

Since fixed latency instructions cannot be augmented with barriers, as established in Chapter 2, the only way to find out how many cycles such an instruction requires is to repeatedly run it with an ever increasing $USCHED_INFO value and check if the result is correct. See Tutorial 6.9 for an effective demonstration of the $USCHED_INFO.

Variable latency instructions can be checked using their barrier mechanism, though it is also possible to benchmark them using the **NOP** filling technique outlined in Tutorial 6.12.

In this section, we outline a *Python* approach for both of these groups and present some results.

We outline one benchmark for **FADD** using portions of its *Python* script, based on the kernel template introduced in Tutorial 6.13 and at the end provide some results. Especially, the reader is encouraged to notice the four instruction slot locations for instructions as outlined in Tutorial **??**6.13:

- `self.add_pre_clock_prequels`
- `self.add_post_clock_prequels`
- `self.add_main`
- `self.add_pre_clock_sequels`
- `self.add_post_clock_sequels`

Note that all of the benchmarking scripts are available in the code base with this work and providing all of them in full in this report is not productive. The technique outlined with the following script is *transferrable* to all fixed latency instructions in any of the SASS instruction sets for SM 70 and above, provided one finds a GPU featuring that instruction set.

First it is necessary to initialize all operands of the instruction with well defined values. For the **FADD** we use **I2F** twice to convert two integers into 32 bit floats.

```
# Preparations: create some 32 bit floats in R32, 34, 36
f32_32 = sc.SASS_KK__i2f__IU_32b(kk_sm, Pg_negate=False,
                Pg=PT, Rd=R32, Sb=10, dst_format=32,
                usched_info_reg=wait15, wr=0x0, rd=0x7)
f32_34 = sc.SASS_KK__i2f__IU_32b(kk_sm, Pg_negate=False,
                Pg=PT, Rd=R34, Sb=15, dst_format=32,
                usched_info_reg=wait15, wr=0x0, rd=0x7)

# Read the values into the operand registers
self.add_pre_clock_prequels(f32_32.class_name, f32_32.enc_vals)
self.add_pre_clock_prequels(f32_34.class_name, f32_34.enc_vals)
```

Inside of the *benchmarking loop* we have to reinitialize the destination register with zero in every iteration.

```
# Put a well defined number in the FADD target register
# that is not the result, for example 0
move1 = sc.SASS_KK__MOVImm(kk_sm,
                           exec_pred_inv=False, exec_pred=PT,
                           target_reg=R30, imm_val=0,
                           usched_info_reg=wait15)
self.add_post_clock_prequels(move1.class_name, move1.enc_vals)
```

Then we run the main instruction, in this case **FADD**, using a configurable number for $USCHED_INFO. With this, we can instantiate this benchmarking process multiple times in a loop with ever increasing values for $USCHED_INFO until the result is correct and based on the insights in Tutorial 6.9, we can assume that the value we used for $USCHED_INFO in that iteration is the number of clock cycles for the main instruction. This insight seems to have been applied by [12] as well, albeit it is not clear to which extent they formalized the approach. Note that $USCHED_INFO can delay for at most 15 cycles. Thus, we have to split the total wait cycle count onto the main instructino and multiple following **NOP** instructions if the required cycle count is larger than 15.

```
# This is the instruction that is benchmarked
# Get the usched_info appropriate for the currently required
# wait_cycles
main_usched_info =
    getattr(kk_sm.regs, 'USCHED_INFO__WAIT{0}_END_GROUP__{0}'
        .format(min(15, wait_cycles)))
main = sc.SASS_KK__fadd__RRR_RR(kk_sm,
       Pg_negate=False, Pg=PT, Rd=R30,
       Ra_reuse=False, Ra_absolute=False, Ra_negate=False, Ra=R32,
       Rc_reuse=False, Rc_absolute=False, Rc_negate=False, Rc=R34,
       usched_info_reg=main_usched_info,
       req=0x0
)
self.add_main(main.class_name, main.enc_vals)
```

Right after the main instruction we add **NOP** instructions to facilitate waiting for more than 15 cycles after the main instruction. This technique is outlined in Tutorial 6.12.

```
# Add NOP instructions that wait for a given number of cycles.
# NOTE that with WAIT1, the actual wait is 2 cycles.
# This is consistent accross all Nvidia GPUs.
# NOTE: this particular example takes about 4 cycles, so all of
# the following are for nothing. They serve as reference.
# NOTE: in the i2f_f2i benchmark, they are not there for nothing.
if wait_cycles > 15:
    nop = sc.SASS_KK__NOP(kk_sm, usched_info_reg=
        getattr(kk_sm.regs, 'USCHED_INFO__WAIT{0}_END_GROUP__{0}'
            .format(min(15, wait_cycles-15))))
    self.add_pre_clock_sequels(nop.class_name, nop.enc_vals)

 ... add more NOP as required ...
```

After **NOP** filling, move the result of the main instruction to a different register using a fixed latency instruction such as **MOV**.

```
# This one writes the value in R30 into f_output_base_ureg.
# NOTE: f_output_base_ureg is incremented by 1 unit in every loop cycle
# NOTE: MUST be done before the next clock measurement, otherwise,
# we add 15 cycles to the execution time of the NOP operations above.
# That is bad because for this one, we care about the result being
```

```
# correct rather than the exact cycle measure. We can always subtract
# 15 from the final result, but if we like to determine if an
# instruction concludes, moving the result to global memory exactly after
# a given number of cycles, is not optional!!
result = sc.SASS_KK__mov__RR(kk_sm,
                             Pg_negate=False, Pg=PT,
                             Rd=R38,
                             Rb_reuse=False, Rb=R30,
                             usched_info_reg=wait15)
self.add_pre_clock_sequels(result.class_name, result.enc_vals)
```

Correct for any additional cycles used by fixed latency instructions inside of the *benchmarking loop*. In this case we used **MOV** twice and need to subtract 15 cycles.

```
# Subtract 30 cycles from the measurement:
# props.cs2r_clk_reg_2 is the register that contains the
# last clock measurement. We do that after the last clock
# measurement but before everything else. This removes
# the clocks needed for the sc.SASS_KK__STG_RaRZ operation
# right above this one and the sc.SASS_KK__MOVImm operation at
# the beginning of the measurement needed for resetting
# the register R34 before each conversion. Both need 15 fixed
# cycles with a sum total of 30.
sub = sc.SASS_KK__IADD3_IMM_RsIR_RIR(kk_sm,
                       target_reg=props.cs2r_clk_reg_main_2,
                       negate_Ra=False, Ra=props.cs2r_clk_reg_main_2,
                       src_imm=-30, # subtract 30
                       negate_Rc=False, Rc=RZ,
                       usched_info_reg=wait15)
self.add_post_clock_sequels(sub.class_name, sub.enc_vals)

move = sc.SASS_KK__STG_RaRZ(kk_sm,
                    uniform_reg=props.f_output_base_ureg, offset=0x0,
                    source_reg=R38,
                    usched_info_reg=wait15,
                    rd=0x7,
                    size=32)
self.add_post_clock_sequels(move.class_name, move.enc_vals)
```

We show some results for variable and fixed latency instructions combined in Section 7.3.3.

7.3.2 Variable Latency Instructions

This technique is tailored to variable latency instructions featuring at least one barrier. As opposed to the technique laid out in the previous Section 7.3.1, this technique does not require a loop since we can simply set a barrier and wait for it to be signaled.

We benchmark DADD, DFMA and DMUL with this approach. Note that it can easily be adapted for all variable latency instructions.

Note that the only notable difference to the previous benchmarking technique (using a loop) is that $USCHED_INFO gets a constant value `usched_info_reg=usched_info_wb` and that we do not need a loop which results in one CUDA binary to run for each benchmark.

```
# Convert R30 from float to integer (F64 to SU64) into R32
# Main instruction goes between the two CS2R clock measurements
main = sc.SASS_KK__dfma__RRR_RRR(kk_sm,
         Pg_negate=False, Pg=PT,
         Rd=R30,
         Ra_reuse=False, Ra_negate=False, Ra_absolute=False, Ra=R32,
         Rb_reuse=False, Rb_negate=False, Rb_absolute=False, Rb=R34,
         Rc_reuse=False, Rc_negate=False, Rc_absolute=False, Rc=R36,
         usched_info_reg=usched_info_wb, req=0, wr=0x0, rd=0x7
         )
self.add_main(main.class_name, main.enc_vals)
```

We show some results for variable and fixed latency instructions combined in Section 7.3.3.

7.3.3 Variable and Fixed Latency Results

We list the results of three variable and their three fixed latency equivalents. Listing **??** shows the results for the double precision instructions. Note that **all** single precision instructions FADD, FFMA and FMUL take **5 cycles**.

Listing 7.1: DADD, DFMA, DMUL and F2I measured with procedures from Sections 7.3.1 and 7.3.2.

	w barrier	w/o barrier
DADD	49	42
DFMA	55	48
DMUL	49	42
F2I (64B)	45	37

7.3.4 Barrier Mechanism

Sections 7.3.1 and 7.3.2 and their respective results in Listing 7.1 revealed a discrepancy for the variable latency instructions DADD, DFMA, DMUL and F2I if measured using their barrier mechanism and without. We note that the discrepancy in required cycles is 7 cycles. Random measurements in a Monte Carlo sense have yielded that this seems to be the case with all instructions utilizing the barrier mechanism.

Thus we conclude that the barrier mechanism itself requires 7 cycles to be queried. We also note that this is more than the 5 cycles all single precision floating point instructions require to complete.

7.3.5 Bulk Instruction Generation and Benchmarking

This is the most involved technique developed during this work.

The goal is to provide full test coverage for all instructions and their *entire* configuration space (all valid combinations of extension registers). We assume that there exist instructions that have different cycle counts depending on their configuration and may require a different number of cycles to complete depending on if they run for the first or n-th time in a loop.

In Section 7.3.5 we show that this is achievable for all non-memory accessing instructions and also verify the two assumptions formulated in the previous paragraph. In Section 7.3.5 we outline an approach that has the potential to work for all instructions, including memory accessing ones.

Figure 7.1 shows the base approach that is required: all input operands must be defined for a tested instruction. For all memory accessing instructions, especially the ones that *write* to the memory system, the destination must point to valid memory and *especially* the *correct* type of memory. For example, some instructions write to *shared* memory while others write to *global* memory.

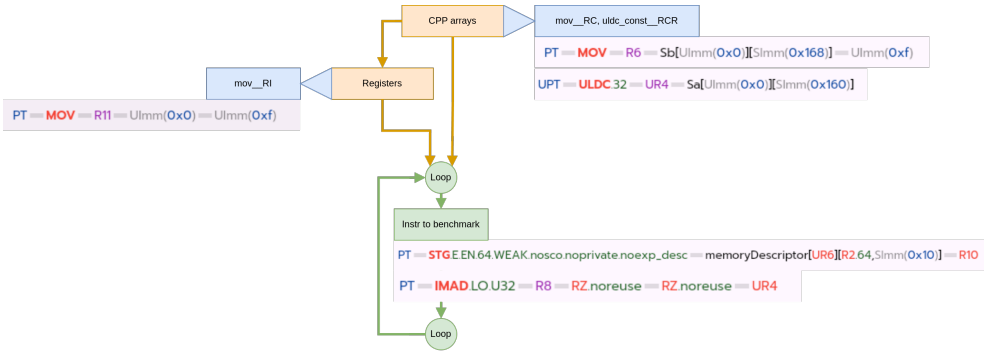


Figure 7.1: Resolving the input operands for a tested instruction.

Non-Memory Accessing Instructions

For this technique, we provide a *resolver* to instantiate additional instructions based on the requirements for any instruction to be measured. We can do this using the *instruction generator* in Section 3.4 by first *generating* a randomized encoding for any instruction in this category, fixing all extension encodings as *ankers* and use the *enc_dom* instruction space iterator to create one valid instruction for every valid combination of the extension registers.

The following *Python* class shows the full kernel instructions injected into the kernel template explained in detail in Tutorial 6.13. Note the `ResolveUtils.resolve_operands` method that does most of the work in this case.

```
class BInstrPrequelResolve(BInstrBase):
  def __init__(self, kk_sm:KK_SM,
            props:KernelWLoopControlProps,
            main_class_name:str,
            main_enc_vals:dict
  ):
     super().__init__(kk_sm, props,
            class_name=BInstrBase.CONST__EARLY_BIRD,
            enc_vals={},
            resolve_operands=False)

     # Get the pre-clock prequels by resolving the operands
     # of the current instruction
     pre_clock_prequels = ResolveUtils.resolve_operands(
                 kk_sm, main_class_name, main_enc_vals, props)

     # Add the pre-clock prequels
     for class_name, enc_vals in pre_clock_prequels:
         self.add_pre_clock_prequels(class_name, enc_vals)
     # Add the main instruction
     self.add_main(main_class_name, main_enc_vals)
     # No sequels
```

Since this piece of code is too extensive to be provided in the Appendix, the reader is invided to check out the *PySASSCreate* module provided with this work. The relevant code is *bulk_resolve_utils.py*.

All tests run in a 10 by 10 loops mode: we run the benchmarked instruction inside the calculated CUDA kernel 10 times in an `Inner Loop` and run the calculated CUDA kernel itself also 10 times in an `Outer Loop`. All numbers

shown in the lists in the following Listings denote *cycle counts* captured between two **CS2R** instructions using SR_CLOCKLO.

First we validate the bulk generation technique by comparing it to simple benchmarks in Section 7.3.1 for double precision floating point operations. We note that Listings 7.2, 7.3 and 7.4 show the same values as their counterpart, single benchmarks in Section 7.3.1.

Listing 7.2: **DMUL** measured with a bulk-generated CUDA kernel. The cycle count is the same as in Section 7.3.1.

```
Outer Loop [1/1] - Kernel_[0-real[0]-/999]: Kernel_0|dmul__RCR_RC
  Inner Loop [1/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [2/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [3/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [4/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [5/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [6/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [7/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [8/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [9/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [10/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
```

Listing 7.3: **DFMA** measured with a bulk-generated CUDA kernel. The cycle count is the same as in Section 7.3.1.

```
Outer Loop [1/1] - Kernel_[269-real[269]-/351]: Kernel_269|dfma__RRR_RRR
  Inner Loop [1/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [2/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [3/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [4/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [5/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [6/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [7/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [8/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [9/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
  Inner Loop [10/10]          [49, 49, 49, 49, 49, 49, 49, 49, 49, 49]
```

Listing 7.4: **DADD** measured with a bulk-generated CUDA kernel. The cycle count is the same as in Section 7.3.1.

```
Outer Loop [1/1] - Kernel_[560-real[560]-/999]: Kernel_560|dadd__RRU_RU
  Inner Loop [1/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [2/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [3/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [4/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [5/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [6/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [7/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [8/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [9/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
  Inner Loop [10/10]          [45, 45, 45, 45, 45, 45, 45, 45, 45, 45]
```

Listing 7.5 showcases the results for one F2F variant, taking 16 cycles. We note that visual inspection of all test results for non-memory accessing, variable latency instructions does indeed seem to support the assumption that they all require 16 cycles ore more, which is more than $USCHED_INFO (see Tutorial 6.9) can provide. Though it is telling, that the barrier mechanism itself (see Section 7.3.4) seems to require 7 cycles across the board. This is a good place to point out that variable latency instructions, for example double precision floating point operations seem to be limited by available resources and require a barrier mechanism to *wait effectively* if all required resources are busy, rather than requiring different cycle counts based on their configuration. [3] shows with impressive benchmarks that double precision floating point multiplication seem to suffer from resource constraints because they link two single precision floating point multipliers.

Listing 7.5: F2F measured with a bulk-generated CUDA kernel. The cycle count is the same as in Section 7.3.1. We note that a visual check over all measurement result seems to indeed verify the assumption that all variable latency instructions require at least 16 cycles.

```
Outer Loop [1/1] - Kernel_[837-real[837]-/991]:
                                        Kernel_837|f2f_f32_downconvert__RCXR_RCXR
  Inner Loop [1/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [2/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [3/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [4/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [5/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [6/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [7/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [8/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [9/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
  Inner Loop [10/10]          [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
```

Even though, the non-memory accessing bulk benchmarks seem to overall work well, there seems to be some trouble with instructions that may require more complex initialization data structures. It shall be noted, though, that to our best visual inspection, HMMA in Listing 7.6 seems to be the only instruction affected.

Listing 7.6: HMMA measured with a bulk-generated CUDA kernel. Note the cycle count of 2. This seems improbable. It may be related to the required *sparse* input format.

```
Outer Loop [1/1] - Kernel_[843-real[843]-/999]: Kernel_843|hmma_sparse_indexedRF_
   Inner Loop [1/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [2/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [3/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [4/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [5/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [6/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [7/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [8/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [9/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
   Inner Loop [10/10]          [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

In Listing 7.7 we find one instance of an instruction that takes one cycle more in the first execution in more than half the times the CUDA kernel was executed. FCHK (Single Precision FP Divide Range Check) is one of the few instructions that feature this behavior that is not related to the graphics pipeline. Listing7.8 shows a much greater difference for the TMML (Texture MipMap Level) as well as TXD (Texture Fetch With Derivatives) instruction in Listing 7.9.

Listing 7.7: FCHK (Single Precision FP Divide Range Check) measured with a bulk-generated CUDA kernel. Note that the first execution sometimes utilizes one cycle more than the rest.

```
Outer Loop [1/1] - Kernel_[6-real[6]-/999]: Kernel_6|fchk__RUR_RU
   Inner Loop [1/10]          [26, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [2/10]          [26, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [3/10]          [26, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [4/10]          [26, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [5/10]          [26, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [6/10]          [27, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [7/10]          [27, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [8/10]          [27, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [9/10]          [27, 26, 26, 26, 26, 26, 26, 26, 26, 26]
   Inner Loop [10/10]          [27, 26, 26, 26, 26, 26, 26, 26, 26, 26]
```

Listing 7.8: TMML (Texture MipMap Level) measured with a bulk-generated CUDA kernel. Note the remarkable overhead for the first execution everytime the CUDA kernel runs the first iteration of the inner loop.

```
Outer Loop [1/1] - Kernel_[668-real[668]-/736]: Kernel_668|tmml_b_
   Inner Loop [1/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [2/10]          [411, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [3/10]          [411, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [4/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [5/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [6/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [7/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [8/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [9/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [10/10]          [412, 111, 111, 111, 111, 111, 111, 111, 111, 111]
```

Listing 7.9: TXD (Texture Fetch With Derivatives) measured with a bulk-generated CUDA kernel. Note the remarkable overhead for the first execution everytime the CUDA kernel runs the first iteration of the inner loop.

```
Outer Loop [1/1] - Kernel_[0-real[0]-/999]: Kernel_0|txd_urc_
   Inner Loop [1/10]          [2320, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [2/10]          [586, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [3/10]          [586, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [4/10]          [334, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [5/10]          [335, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [6/10]          [335, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [7/10]          [334, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [8/10]          [333, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [9/10]          [335, 122, 122, 122, 122, 122, 122, 122, 122, 122]
   Inner Loop [10/10]          [334, 122, 122, 122, 122, 122, 122, 122, 122, 122]
```

The SULD (Surface Load) is one of the instructions exhibiting different cycle counts for different configuration variants in Listings 7.10, 7.11 and 7.12.

Listing 7.10: SULD (Surface Load, version 1) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.11 and 7.12 we note different cycle counts in subsequent loop interations within the kernel.

```
Outer Loop [1/1] - Kernel_[207-real[207]-/999]: Kernel_207|suld_d_imm_
   Inner Loop [1/10]          [689, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [2/10]          [401, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [3/10]          [400, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [4/10]          [400, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [5/10]          [401, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [6/10]          [400, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [7/10]          [401, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [8/10]          [400, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [9/10]          [400, 111, 111, 111, 111, 111, 111, 111, 111, 111]
   Inner Loop [10/10]          [401, 111, 111, 111, 111, 111, 111, 111, 111, 111]
```

Listing 7.11: SULD (Surface Load, version 2) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.10 and 7.12 we note different cycle counts in subsequent loop interations within the kernel.

```
Outer Loop [1/1] - Kernel_[208-real[208]-/999]: Kernel_208|suld_d_imm_
   Inner Loop [1/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [2/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [3/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [4/10]          [680, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [5/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [6/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [7/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [8/10]          [677, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [9/10]          [408, 118, 118, 118, 118, 118, 118, 118, 118, 118]
   Inner Loop [10/10]          [407, 118, 118, 118, 118, 118, 118, 118, 118, 118]
```

Listing 7.12: SULD (Surface Load, version 3) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.10 and 7.11 we note different cycle counts in subsequent loop interations within the kernel.

```
Outer Loop [1/1] - Kernel_[209-real[209]-/999]: Kernel_209|suld_d_imm_
   Inner Loop [1/10]          [398, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [2/10]          [398, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [3/10]          [399, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [4/10]          [397, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [5/10]          [397, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [6/10]          [399, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [7/10]          [399, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [8/10]          [397, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [9/10]          [398, 108, 108, 108, 108, 108, 108, 108, 108, 108]
   Inner Loop [10/10]          [398, 108, 108, 108, 108, 108, 108, 108, 108, 108]
```

As *conclusion* of this section we note that classical CUDA instructions (for example **DMUL**, **DADD**, **DFMA**, **F2F**, **FCHK**, etc) seem to exhibit fairly constant cycle counts independently of their configurations, albeit rarely, the first time the instruction is called inside of a CUDA kernel, it may take one cycle more. Note that all thests were performed in single threaded kernels with otherwise no load to speak of on the GPU.

Memory Accessing Instructions

The `ResolveUtils.resolve_operands` fails for almost all memory accessing instructions. In this section we provide a better approach on how to tackle these kinds of instructions generically. We do this by utilizing the capabilities of the *py_sass* module to perform analysis on the instruction set as a whole and isolate all *memory accessing patterns* used by instructions and generate a *pattern matching* scheme as already done for the *instruction generator* in Section 3.4.3.

This is one of many cases where exhaustive analysis of the SASS instruction set using the provided *py_sass* module provides valuable insight.

Listing 7.13: The four most common memory access patterns in the SM 86 instruction set. For example, the most common pattern reflects the classical memory bank access and occurs 163 times.

```
All Access Patterns (Verbose Descrption)
========================================
 163:[UImm]-[SImm]
      bmov_dst64__C
      attr_1.0:[UImm(5/0*):Sb_bank]-attr_1.1:[SImm(17)*:Sb_addr]
 159:[UniformRegister]-[UImm]
      bmov_dst64__CX
      attr_1.0:[UniformRegister:URb]-attr_1.1:[UImm(16)*:Sb_offset]
  34:[NonZeroRegister]
      ald_PHYS_
      attr_1.0:[NonZeroRegister:Ra]
  26:[ZeroRegister,UImm]
      ald__LOGICAL_RaRZ_default
      attr_1.0:[ZeroRegister(RZ):Ra+UImm(10)*:Ra_offset]
```

The *verbose* access patterns in Listing **??** can be contracted to a set based notation as shown in Listing

Listing 7.14: All memory access patterns illustrated in Listing 7.13, contracted to a set notation to be used for pattern matching.

```
All Memory Access Patterns
==========================
{'NonZeroRegister', 'SImm'}
{'NonZeroRegister', 'UniformRegister', 'SImm'}
{'NonZeroRegister', 'UniformRegister'}
{'NonZeroRegister'}
{'SImm', 'NonZeroUniformRegister'}
{'Register', 'SImm'}
{'Register', 'SImm'},{'Register', 'UniformRegister', 'SImm'}
{'Register', 'SImm'},{'UniformRegister'},{'Register', 'SImm'}
{'Register', 'SImm'},{'ZeroRegister', 'UniformRegister', 'SImm'}
{'Register', 'UImm'}
{'Register', 'UniformRegister', 'SImm'}
{'Register', 'UniformRegister', 'SImm'},{'Register', 'SImm'}
{'Register', 'UniformRegister', 'SImm'},{'UniformRegister'},{'Register', 'SImm'}
{'Register', 'UniformRegister', 'SImm'},{'ZeroRegister', 'SImm'}
{'UImm'}
{'UImm'},{'NonZeroRegister', 'SImm'}
{'UImm'},{'SImm'}
{'UImm'},{'UImm'},{'SImm'}
{'UImm'},{'UniformRegister', 'SImm'}
{'UImm'},{'ZeroRegister', 'SImm'}
{'UniformRegister', 'SImm'}
{'UImm', 'UniformRegister'}
{'UniformRegister'}
{'UniformRegister'},{'NonZeroRegister', 'SImm'}
```

```
{'UniformRegister'},{'Register', 'SImm'}
{'UniformRegister'},{'UImm'}
{'UniformRegister'},{'UImm'},{'SImm'}
{'UniformRegister'},{'UniformRegister'}
{'UniformRegister'},{'UImm', 'ZeroRegister'}
{'UImm', 'ZeroRegister'}
{'ZeroRegister', 'UniformRegister', 'SImm'}
{'ZeroRegister', 'UniformRegister'}
{'UImm', 'ZeroUniformRegister'}
```

Why is this useful? With these access patterns, we can create `prequel` instructions to fit all different memory access types and modify the main instruction to use the correct operands that we initialize with the `prequel` instructions. The following *Python* code shows a portion of the required case distinctions. The full code is available in the code base with this work in *bulk_resolve_utils2.py*.

```
...
if mem_access_pattern == {'NonZeroRegister', 'SImm'}:
    if is_first_attrib:
        raise Exception(sp.CONST__ERROR_NOT_IMPLEMENTED)
    elif is_second_attrib:
        # Addresses are 64 bits: we really really need both of these.
        # Otherwise we get a memory access error
        pc = sc.SASS_KK__MOVImm(kk_sm,exec_pred_inv=False,
                    exec_pred=PT, target_reg=R52, imm_val=0x0,
                    usched_info_reg=wait15, req=0x0)
        # add the instruction as prequel
        prequel_instr.append((pc.class_name, pc.enc_vals))
        pc = sc.SASS_KK__MOVImm(kk_sm,exec_pred_inv=False,
                    exec_pred=PT, target_reg=R53, imm_val=0x0,
                    usched_info_reg=wait15, req=0x0)
        # add the instruction as prequel
        prequel_instr.append((pc.class_name, pc.enc_vals))
        # modify the main instruction to use the prequel instruction
        main_enc_vals = SM_CuBin_Utils.overwrite_helper(
                R52, type_map['NonZeroRegister'], main_enc_vals)
        main_enc_vals = SM_CuBin_Utils.overwrite_helper(
                props.ui_input_base_offset, type_map['SImm'], main_enc_vals)
    elif is_list:
        pc = sc.SASS_KK__MOVImm(kk_sm,exec_pred_inv=False,
                    exec_pred=PT, target_reg=R52, imm_val=0x0,
                    usched_info_reg=wait15, req=0x0)
        # add the instruction as prequel
        prequel_instr.append((pc.class_name, pc.enc_vals))
        pc = sc.SASS_KK__MOVImm(kk_sm,exec_pred_inv=False,
```

```
                    exec_pred=PT, target_reg=R53, imm_val=0x0,
                    usched_info_reg=wait15, req=0x0)
        # add the instruction as prequel
        prequel_instr.append((pc.class_name, pc.enc_vals))
        # modify the main instruction to use the prequel instruction
        main_enc_vals = SM_CuBin_Utils.overwrite_helper(
                R52, type_map['NonZeroRegister'], main_enc_vals)
        main_enc_vals = SM_CuBin_Utils.overwrite_helper(
                props.ui_input_base_offset, type_map['SImm'], main_enc_vals)
    Instr_CuBin.check_expr_conditions(class_, main_enc_vals, throw=True)
...
```

In the same spirit as with the *non memory accessing* instruction in the previous Section 7.3.5, we provide a selection of results. All results are available with the software for this work. It has to be noted, though, that also with this approach, most calculated benchmarking CUDA binaries fail with *invalid memory access* errors.

Listings 7.15, 7.16 and 7.17 show three different versions of LD (Load from generic Memory) featuring three different cycle counts. This is one instruction where the assumption stated earlier, that the extension configuration has an influence on the cycle count is true for an instruction that is not mainly related to texture rendering. We note that this is the case often for the tests that do run with the memory accessing instructions category.

Listing 7.15: LD (Load from generic Memory) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.16 and 7.17 we note different cycle counts for instruction variants with different extension configurations.

```
Outer Loop [1/1] - Kernel_[210-real[210]-/999]: Kernel_210|ld__uImmOffset
  Inner Loop [1/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [2/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [3/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [4/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [5/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [6/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [7/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [8/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [9/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
  Inner Loop [10/10]          [28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
```

Listing 7.16: LD (Load from generic Memory) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.15 and 7.17 we note different cycle counts for instruction variants with different extension configurations.

```
Outer Loop [1/1] - Kernel_[212-real[212]-/999]: Kernel_212|ld__uImmOffset
  Inner Loop [1/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
```

```
  Inner Loop [2/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [3/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [4/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [5/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [6/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [7/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [8/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [9/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
  Inner Loop [10/10]          [34, 34, 34, 34, 34, 34, 34, 34, 34, 34]
```

Listing 7.17: LD (Load from generic Memory) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.15 and 7.16 we note different cycle counts for instruction variants with different extension configurations.

```
Outer Loop [1/1] - Kernel_[218-real[218]-/999]: Kernel_218|ld__uImmOffset
  Inner Loop [1/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [2/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [3/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [4/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [5/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [6/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [7/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [8/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [9/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
  Inner Loop [10/10]          [30, 30, 30, 30, 30, 30, 30, 30, 30, 30]
```

As the reader may have realized while inspecting Listings 7.15, 7.16 and 7.17, the cycle counts are all very constant. The suspicion is that all the samples access the $cache rather than actual global memory which may be the reason why the tests worked. We do manage to find tests for ATOM (Atomic Operation on Generic Memory) exhibiting more irregular cycle counts in Listing 7.18.

Listing 7.18: ATOM (Atomic Operation on Generic Memory) measured with a bulk-generated CUDA kernel. Comparing to Listings 7.15 we note the larger and irregular cycle count exhibited by this instruction.

```
Outer Loop [1/1] - Kernel_[156-real[156]-/999]: Kernel_156|atomg_uniform__RaRZ
  Inner Loop [1/10]          [263, 262, 262, 262, 262, 262, 262, 263, 262, 262]
  Inner Loop [2/10]          [279, 279, 280, 280, 279, 281, 279, 280, 279, 281]
  Inner Loop [3/10]          [261, 263, 261, 262, 262, 262, 261, 263, 262, 262]
  Inner Loop [4/10]          [278, 281, 279, 280, 278, 280, 280, 281, 278, 281]
  Inner Loop [5/10]          [262, 262, 262, 262, 262, 261, 263, 261, 263, 261]
  Inner Loop [6/10]          [280, 280, 279, 280, 279, 280, 280, 280, 281, 280]
  Inner Loop [7/10]          [262, 263, 263, 263, 262, 262, 262, 262, 263, 262]
  Inner Loop [8/10]          [279, 280, 278, 281, 279, 280, 281, 279, 281, 282]
  Inner Loop [9/10]          [261, 262, 262, 263, 261, 263, 262, 262, 262, 262]
  Inner Loop [10/10]          [280, 279, 281, 279, 281, 279, 281, 279, 280, 281]
```

To put the numbers in Listing 7.18 into perspective, we provide the test results of an STG instruction using the pattern in Section 7.3.3 in Listing 7.19.

Listing 7.19: STG (Store to Global) measured with a CUDA kernel generated as in Section 7.3.3. Comparing to Listings 7.18 shows the same irregular cycle count pattern.

```
Outer Loop [1/1] - Kernel_[0-real[0]-/0]: Kernel_0|stg_uniform__RaRZ
  Inner Loop [1/10]          [275, 275, 274, 277, 274, 274, 276, 273, 275, 275]
  Inner Loop [2/10]          [274, 277, 276, 274, 275, 273, 275, 275, 273, 275]
  Inner Loop [3/10]          [274, 274, 274, 275, 274, 276, 275, 275, 275, 275]
  Inner Loop [4/10]          [274, 276, 275, 276, 275, 276, 275, 275, 275, 275]
  Inner Loop [5/10]          [275, 275, 274, 274, 275, 275, 276, 274, 275, 274]
  Inner Loop [6/10]          [274, 274, 275, 275, 274, 276, 275, 276, 274, 275]
  Inner Loop [7/10]          [273, 276, 275, 275, 274, 275, 275, 273, 276, 274]
  Inner Loop [8/10]          [273, 276, 275, 274, 275, 274, 276, 274, 276, 274]
  Inner Loop [9/10]          [273, 275, 274, 275, 275, 274, 275, 275, 277, 275]
  Inner Loop [10/10]          [275, 275, 275, 275, 275, 274, 275, 276, 275, 275]
```

We *conclude* that while it seems to be possible to bulk-generate CUDA kernels testing memory accessing instructions and provide an approach to facilitate such a thing, it is still far from completed and does not provide the desired full picture of the memory system. We also note that the memory accessing instructions exhibit more irregular behavior than non-memory accessing instructions and it is thus much more interesting to benchmark this part of the SASS instruction set with the presented, calculated approach.

7.3.6 Limitations

There **is** a time and a place for optimizations! Even in research.

The main limitation of the current state of the software developed in this work is *memory usage*. *Python* is not memory efficient. Especially the *py_cubin* module will benefit greatly from some optimizations. It is the opinion of the author that optimizing *Python* code is in normal circumstances close to hopeless, and portions of the *py_cubin* module have to be ported to C++ using *Nanobind* [16]. This technique was applied successfully for the *py_sass* module to represent the two data types **SASS_Bits** and **SASS_Range** which is what ultimately enabled the calculations for the *instruction generator* in Section 3.4.

The second limitation is *speed*. Dealing with millions of CUDA kernels does require some time to calculate. However, constructing bulk CUDA kernel generators also requires a lot of iterations. Next to reducing memory requirements, shortening the duration of one development cycle is paramount and can also be achieved by porting selected portions of the *py_cubin* module to C++.

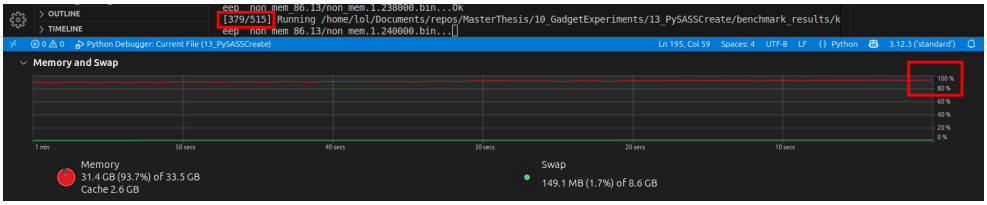


Figure 7.2: Note that the tests are at 379/515 and the memory is almost full.

7.4 Cycle Counting Model

7.4.1 Introduction

Inspired by [1] we present a model that can use information to be gained using out *py_sass* module as well as the microbenchmarking techniques outlined in Section 7.3 as inputs and place a kernel into the 2D canvas in Figure 7.3, indicative of its expected performance.

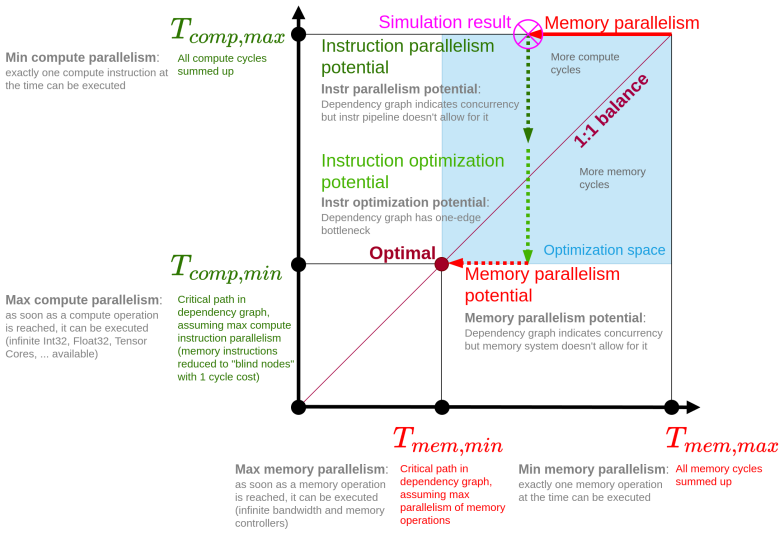


Figure 7.3: 2D performance expectation canvas, inspired by [1].

T_{mem} and T_{comp} are counted in cycles and indicate the amount of work to be done in each of the two categories *memory* and *compute*. Higher T_{comp} or T_{mem} indicates *more* work in the respective category while lower T_{comp} or T_{mem} indicates less work and is good for better performance.

All non-memory accessing instructions are in the T_{comp} category and require an abstract C_i number of cycles. All memory instructions are in the T_{mem} category and require $1 + \#\text{mem\_access\_cycles} = 1 + M$ cycles. Since each instruction has to be dispatched by the dispatcher, requiring compute, we

utilize a 1 cycle offset for each instruction in the T_{mem} category.

Figure 7.3 depicts the case where $T_{comp} = T_{mem}$ while Figures 7.4 and 7.5 show a more practical case where $T_{mem} \neq T_{comp}$.

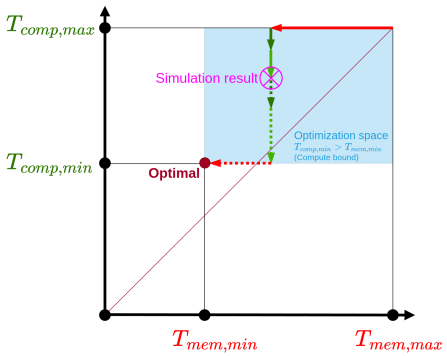


Figure 7.4: Compute bound: the **Optimal** point is located on the lower left corner of the *optimization space* rectangle while the corner itself is located in the *compute bound* upper left triangle. Thus the depicted kernel is *compute bound*.

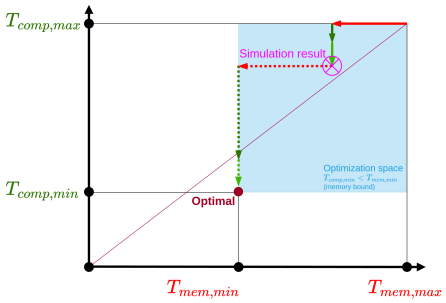


Figure 7.5: Memory bound: the **Optimal** point is located on the lower left corner of the *optimization space* rectangle while the corner itself is located in the *memory bound* lower right triangle. Thus the depicted kernel is *memory bound*.

Note that *py_sass* module from Chapter 2 is provides all instructions in both T_{comp} and T_{mem} via its **SASS_Class_Props** and **SM_Cu_Props** classes. See Figure 2.14 for the full SASS instruction categorization graph. In addition, it provides the category T_{ctrl}, including all data-flow changing *control* instructions. For categories T_{mem} we propose using 1 as compute cycle count. Since the instructions in T_{ctrl} are either difficult or impossible to benchmark, we propose a compute cycle count of 1 for those as well. The value 1 is rooted in the evaluation of the SASS specifications in Chapter 2. See Section 2.5.14 for details regarding the *MIN_WAIT_NEEDED* value.

7.4.2 Kernel DAG

Figure 7.6 depicts the simplest graph with only one instruction while Figure 7.7 uses a more complex DAG featuring memory and compute instructions. In an *ideal setting* the assumption would be that all instructions can execute in parallel during the first C cycles of the program. Data dependencies make that impossible. Figure 7.7 introduces such data dependencies.

7.4.3 Compute T_{mem}

$T_{mem,max}$ and $T_{mem,min}$ in Figure 7.3 are calculated as follows: Figure 7.8 shows how all **compute** instructions are replaced with placeholder **blind-nodes** with 0 cost. The purpose of the blind-nodes is to keep the control flow of the

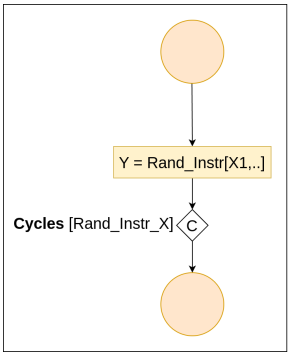


Figure 7.6: Depiction of one instruction: $Y = \text{Rand\_Instr[X1,...]}$ where $X1, ...$ indicates the instruction's dependencies and C its cost in cycles. Note that memory instructions would contain $1 + M$ cycles. The two orange nodes are *source* and *sink*.

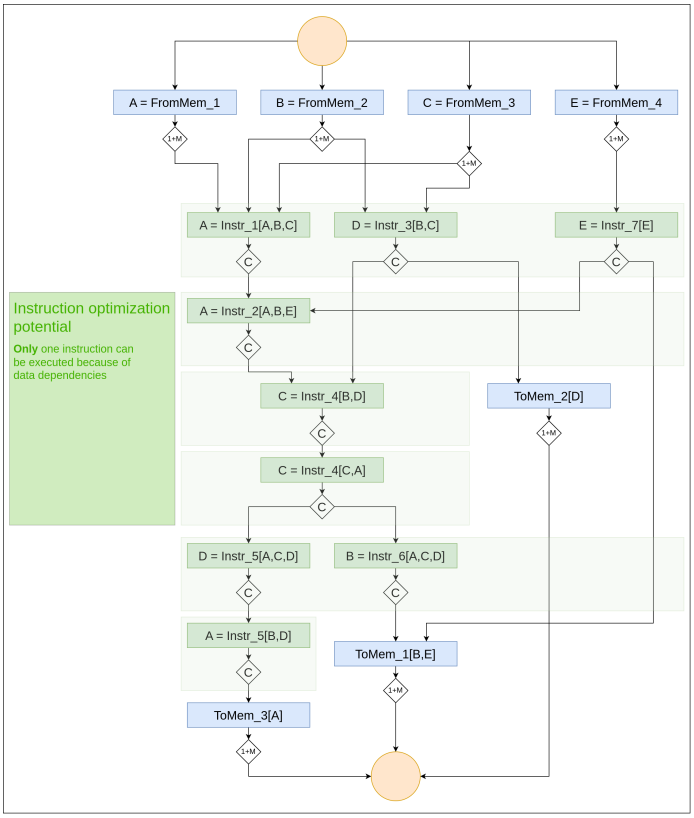


Figure 7.7: The soft green, horizontal boxes indicate *parallel instruction execution*. Note the green side box indicating optimization potential: the three instructions in the adjacent soft green boxes are connected trough data dependencies and inhibit parallelism.

graph coherent such that the **memory** instructions still have to respect the control flow of the program, even if the *cost* is 0. Subsequently, in Figure 7.9 to get $T_{mem,max}$ we sum up *all* memory operations cycle counts. This covers the case where every single memory operation has to be executed in sequence, without any memory parallelism. For $T_{mem,min}$, we compute the kernel DAG's *critical path*. Then we sum up all memory operation cycles for the memory instructions along the *critical path*. This is the case where we assume that we always have enough memory bandwidth and controllers available to immediately start any memory operations.

Each memory accessing instruction has cost $1 + M$ where 1 is the cost for dispatching the instruction and M is the memory access latency.

Note that the *Python* module *py_sass* provides in depth access to parses SASS instructions that can be utilized to find data dependencies. Since Nvidia employs a software scheduler controlled by $USCHED_INFO and the barrier mechanism, these two mechanisms must be employed for the dependencies

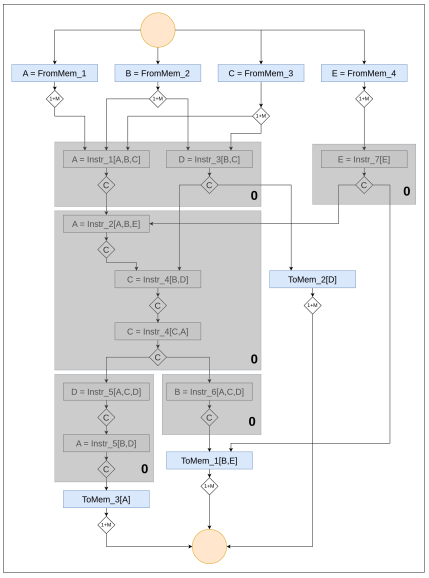


Figure 7.8: Replace all *compute* instructions with **blind-nodes** with cost 0 to compute the memory cost of the kernel.

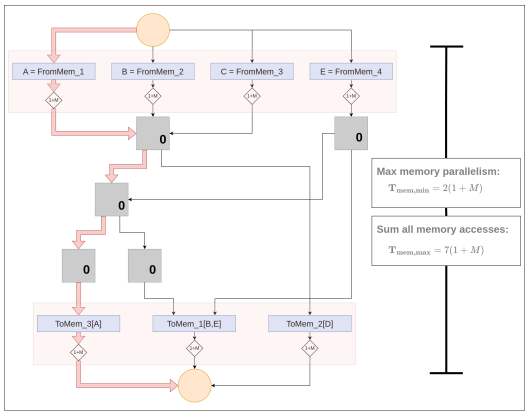


Figure 7.9: Compute the *critical path* of the kernel DAG to compute $T_{mem,min}$. $T_{mem,max}$ is simply the sum of the latencies of all memory operations.

in the kernel DAG as well. The goal is that the graph depicts what the Nvidia NVCC compiler does, not what the ideal world would look like.

7.4.4 Compute T_{comp}

For $T_{comp,min}$ and $T_{comp,max}$, in Figure 7.10 we replace all *memory* instructions with **blind-nodes** with the cost 1. This reflects that the instruction dispatcher still requires one cycle to dispatch a memory instruction. Then, in Figure 7.11 we compute the *critical path* for the modified kernel DAG. $T_{comp,max}$ is the sum of all latencies for the *compute instructions* **and** the dispatch cost for all memory instructions, while $T_{comp,min}$ only sums up the latencies of the instructions and memory instructions dispatch cost that lie on the critical path.

Recall that the compute instructions in T_{comp} are subdivided into fixed- and variable latency instructions. For the kernel DAG, the fixed latency instructions can simply employ the value of their respective $USCHED_INFO field while the variable latency instructions require benchmarking, as do the instructions in the T_{mem} category.

7.4.5 Accounting for GPU Resources

The only thing left is to take into account the *available resources*. These can be for example *memory controllers* for memory instructions or *multipliers* for

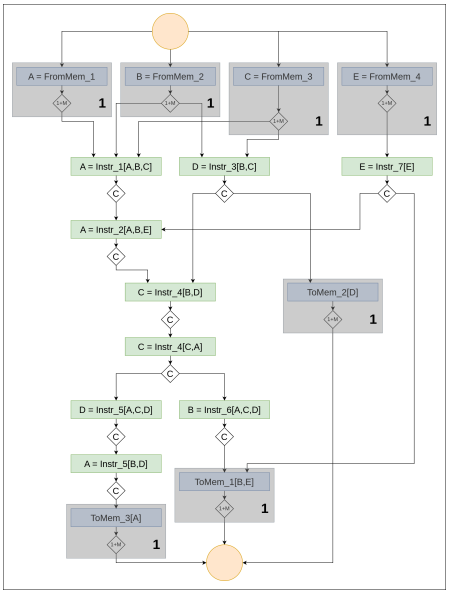


Figure 7.10: Replace all *memory* instructions with **blind-nodes** with the cost 1.

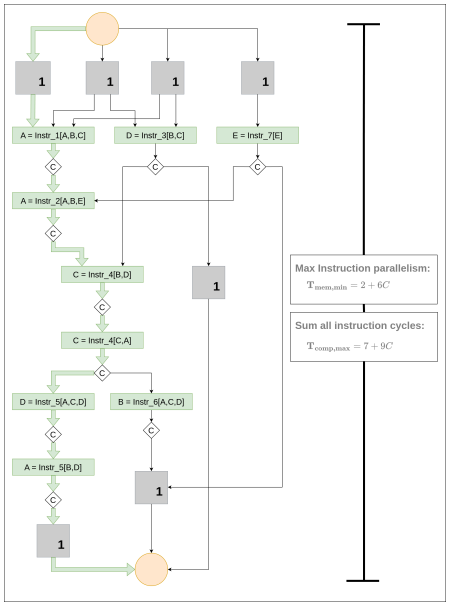


Figure 7.11: Compute the critical path in the modified kernel DAG. $T_{comp,max}$ is the sum of all latencies, *compute instructions* and *dispatch latencies* for the memory instructions (gray boxes). $T_{mem,min}$ only includes the cost of the *compute instructions* and memory instruction *dispatch cost* that lie on the critical path.

compute instructions. To conclude this section, Figure 7.12 depicts a situation in which the kernel DAG has to be **squeezed** through the available resources of a GPU's CUDA pipeline.

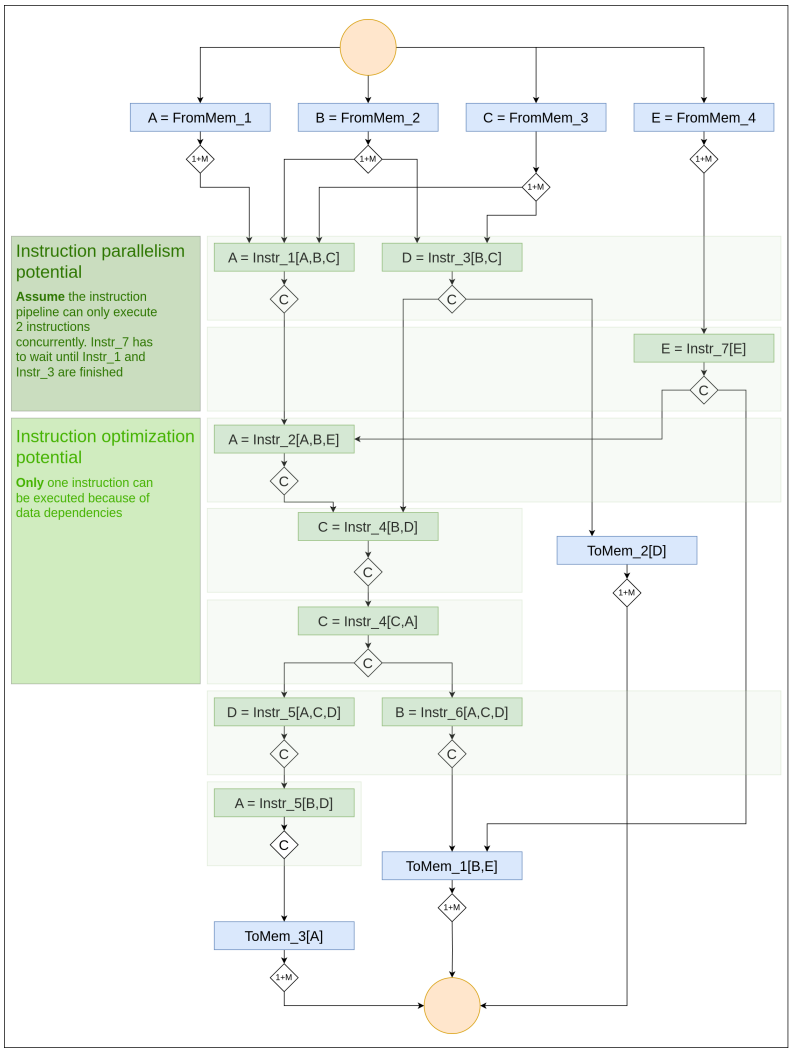


Figure 7.12: The kernel DAG is **squeezed** trough the resources of a GPU. Note how the dark green side box depicts parallelization potential due to resource contention for the *compute* instructions.

Chapter 8

Future Work

8.1 Bulk-Benchmark SASS Memory Instructions

In Section 7.3.5 we show initial computation approaches to facilitate *bulk-generation* of benchmarking CUDA kernels for memory accessing SASS instructions. While the approach is promising, it requires more work.

8.2 Optimize the Framework

Some components in the provided *Python* framework suffer from excessive memory usage. Optimizing these components in a targeted way utilizing *Nanobind* will go a long way in enabling more research utilizing the provided *Python* modules.

8.3 Control Flow Visualization

While the *Python* module *py_cubin* already contains a prototype for a control flow extraction, it is far from finished. Currently, the control flow is extracted using the Nvidia tool *nvdisasm* and the result is mapped onto the SASS instructions that are decoded using *py_cubin*. It would be better to extract the control flow utilizing *py_cubin* only. Further analysis of the instruction set as demonstrated countless times in this work may facilitate this.

Further, the control flow should be visualized in the provided VSCode extension from Chapter 4.

8.4 SASS Instruction Browser

Currently, as outlined in Tutorial 6.3, it can be a bit labor intensive to map a not yet used SASS instruction into the *Python* framework. Providing an

instruction browser, included in the VSCode extension from Chapter 4 has the potential to optimize the process.

Providing a simple search window with limited information about each instruction such that a user can search for key worlds such as *add* or *mult* and then cumulately show all information regarding a selected instruction's operands and a configurable template to produde the *Python* abstraction may be helpful.

8.5 Implement the Cycle Counter Model

In Section 7.4 we provide a theoretical model for a cycle counter. The best case scenario for this thesis and its author is, that someone will use the design in this thesis to implement one.

Chapter 9

Conclusion

This thesis has successfully achieved its goal of laying foundational groundwork needed to facilitate a performance estimation model for CUDA kernels based on a cycle counting approach by directly and in-depth analyzing Nvidia's SASS instructions in Chapter 2. We began by presenting a comprehensive overview of past GPU performance modeling efforts in Chapter 1, followed by a detailed formalization of the SASS instruction set in Chapter 2, culminating in a full *Python* abstraction enabling precise decoding and encoding of SASS instructions from both NVCC-generated and custom CUDA kernels in Chapters 3 and 5. Finally we showcase multiple benchmarking techniques based on the developed *Python* framework, directly utilizing SASS instructions in Chapter 7.

Moreover, we developed multiple *Python* modules that serve as practical tools for interacting with SASS instructions, complemented by a VSCode extension in Chapter 4 that provides an intuitive and detailed visualization of decoded instructions. To support learning and deeper understanding, we delivered a thorough tutorial series in Chapter 6 jumpstarting the readers understanding of direct SASS instruction utilization.

Central to our contributions is the introduction of microbenchmarking strategies directly utilizing SASS instructions utilizing *Python* as a tool including showcasing a technique capable of calculating thousands of benchmarking CUDA kernels via an instruction generator. Finally, leveraging the insights gained through these analyses and tools, we proposed a cycle counting model in Chapter 7 tailored to the capabilities of our *Python* SASS framework, proposing a concrete method for predicting CUDA kernel performance with architectural specificity.

Altogether, these contributions provide a robust and practical foundation for future work in architecture-aware GPU performance modeling and open new avenues for precise, instruction-level analysis and optimization of CUDA.

Bibliography

[1] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 11–22.

[2] J. Lemeire, J. G. Cornelis, and E. Konstantinidis, "Analysis of the analytical performance models for gpus and extracting the underlying pipeline model," *Journal of Parallel and Distributed Computing*, vol. 173, pp. 32–47, 2023.

[3] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for gpus with caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1800–1813, 2014.

[4] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010, pp. 105–114.

[5] P. Schaad, T. Ben-Nun, and T. Hoefler, "Boosting performance optimization with interactive data movement visualization," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2022, pp. 1–16.

[6] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 152–163.

[7] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 280–289.

[8] Nvidia. "Parallel thread execution isa version 8.8." (2025), [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[9] J. Lee *et al.*, "Gcom: A detailed gpu core model for accurate analytical modeling of modern gpus," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 424–436.

[10] Nvidia. "Cuda binary utilities." (2025), [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/.

[11] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, "Decoding cuda binary," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2019, pp. 229–241.

[12] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[13] 0xD0GF00D, *Documentsass*, https://github.com/0xD0GF00D/DocumentSASS, 2025.

[14] B. Cabral. "What is sass short for." (2025), [Online]. Available: https://stackoverflow.com/questions/9798258/what-is-sass-short-for.

[15] E. Dijkstra, "Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60," Jan. 1961.

[16] W. Jakob, *Nanobind: Tiny and efficient c++/python bindings*, https://github.com/wjakob/nanobind 2022.

[17] Nvidia. "Convertfloattype for rcp64." (2025), [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/#floating-point-instructions-rcp-approx-ftz-f64.

[18] Nvidia. "Convertfloattype for rsq64." (2025), [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/#floating-point-instructions-sqrt.

[19] Google, *Flutter apps framework based on dart*, https://flutter.dev/, 2025.

[20] D. R. Hipp, *Sqlite home page*, https://www.sqlite.org/, 2025.

[21] U. S. Laboratories, *Elf - osdev wiki*, https://wiki.osdev.org/ELF, 2025.

Appendix A

Appendix

A.1 Enc_Vals Reduction Algorithm

This algorithm is used in Section 3.4.2 to reduce a list of dictionaries of [str:SASS_Bits] to a list of dictionaries containing strings and sets of SASS_Bits: [str: set(...SASS_Bits...)].

```
@staticmethod
def __group_rec(k_sorted, s_okok):
  k = k_sorted[0]
  if len(k_sorted) > 1:
    gg = []
    for i,g in itt.groupby(s_okok, key=operator.itemgetter(k)):
      rec,ok = SASS_Expr_Domain_Contract.__group_rec(
        k_sorted[1:],
        [dict(j for j in i.items() if j[0] != k) for i in list(g)]
      )
      gg.append((i, rec))
  else:
    return [{k: set(itt.chain.from_iterable(
                                        [i.values() for i in s_okok]))}],k

  # group by the result of the recursion (item 1), then take only the group
  # element m[0] from each group
  pp = [(i,set(m[0] for m in g))
                      for i,g in itt.groupby(gg, key=operator.itemgetter(1))]
  # collect all elements with current key k into a set for the same group
  res = list(itt.chain.from_iterable([{k:n} | s for s in r] for r,n in pp))
  return res,k

@staticmethod
```

```
25  def group(okok:typing.List[dict]):
26    if not okok: return []
27
28    res_k = {}
29    for k in okok[0].keys():
30      res_k[k] = set()
31    # collect all values for each alias in the result in the set,
32    # so that each value is unique
33    for i in okok:
34      for k in res_k.keys(): res_k[k].add(i[k])
35    # res_k contains a set for each alias with all values in all
36    # surviving possibilities out of itt.product. Sort them by
37    # their size and get the aliases (keys) in ascending order
38    # into k_sorted
39    asc_keys = sorted([(k, len(v)) for k,v in res_k.items()],
40                                          key=operator.itemgetter(1))
41    k_sorted = [k[0] for k in asc_keys]
42    # we must apply the groupby to a sorted list of
43    # possibilities (okok). Sort the okok list of dictionaries
44    # according to the alias values
45    s_okok = sorted(okok, key=operator.itemgetter(*k_sorted))
46
47    res,k = SASS_Expr_Domain_Contract.__group_rec(k_sorted, s_okok)
48
49    # add
50    gg = [(i[k], ([(kk,i[kk]) for kk in i.keys() if not kk==k])) for i in res]
51    pp = []
52    for i in gg:
53      # print(i[0], [(kk, vv) for kk,vv in i[1]])
54      ff = False
55      for p in pp:
56        if all([kk in p and vv == p[kk] for kk,vv in i[1]]):
57          if k in p: p[k] = p[k].union(i[0])
58          else: raise Exception(sp.CONST__ERROR_UNEXPECTED)
59            ff = True
60            break
61      if ff: continue
62      pp.append(dict(i[1]))
63      pp[-1][k] = i[0]
64    return pp
```

A.2 PySASSCreate

This Python project contains all benchmarking scripts as well as the various *sass_create* classes to create custom SASS instructions. This section features two full examples for such *creator* classes. Section A.2.1 shows a simplified version while Section A.2.2 shows a full version, also introducing the two *helper methods* commonly used.

A.2.1 Simple Create Instruction Creator Example

```
class SASS_KK__umov__UR:
    def __init__(self, sass:SM_SASS,
                         negate_upred:bool,
                         upred:tuple,
                         target_ureg:tuple,
                         source_ureg:tuple,
                         usched_info_reg:tuple
    ):
        class_name = 'umov__UR'

        # NOTE: we remove these lines by commenting them out as soon
        # as the creator is finished.
        # Generate a valid set of encoding values for class_name
        # enc_vals = sass.encdom.pick(class_name)
        # Print the encoded values as strings to be copy-pasted into
        # the enc_vals dictionary below
        # print(SASS_Create_Utils.enc_vals_dict_to_init(enc_vals))

        enc_vals = {
            # predicate, set by param
            'UPg': SASS_Create_Utils.sass_bits_from_str('2U:3b'),
            # negate predicate, set py param
            'UPg@not': SASS_Create_Utils.sass_bits_from_str('0U:1b'),
            # source ureg, set by param
            'URb': SASS_Create_Utils.sass_bits_from_str('31U:6b'),
            # destination ureg, set py param
            'URd': SASS_Create_Utils.sass_bits_from_str('45U:6b'),
            # keep 0
            'batch_t': SASS_Create_Utils.sass_bits_from_str('0U:3b'),
            # keep 0
            'pm_pred': SASS_Create_Utils.sass_bits_from_str('0U:2b'),
            # wait for nothing
            'req_bit_set': SASS_Create_Utils.sass_bits_from_str('0U:6b'),
            # set py param
```

```
            'usched_info': SASS_Create_Utils.sass_bits_from_str('17U:5b')
        }

        enc_vals = overwrite_helper(negate_upred, 'UPg@not', enc_vals)
        enc_vals = overwrite_helper(upred, 'UPg', enc_vals)
        enc_vals = overwrite_helper(target_ureg, 'URd', enc_vals)
        enc_vals = overwrite_helper(source_ureg, 'URb', enc_vals)
        enc_vals = overwrite_helper(usched_info_reg, 'usched_info', enc_vals)

        # The reader may recognize this handy helper method call from earlier
        # chapters
        Instr_CuBin.check_expr_conditions(
                kk_sm.sass.sm.classes_dict[class_name],
                enc_vals,
                throw=True)

        # Use simple names to access both enc_vals and class_name
        # from the 'outside'. Following tutorials will use these
        # two to create real instructions and insert them into
        # kernel
        self.enc_vals = enc_vals
        self.class_name = class_name
```

A.2.2 Full Create Class for a DFMA instruction variant

This section shows one full *Python class* example for how to use the enc_vals of one DMA instruction and modify them, including the *overwrite_helper*, a convenience method to save a lot of lines of code.

```
# Convenience method, converting a Python string to SASS_Bits
def sass_bits_from_str(s:str):
    lead, bit_len_str = s.split(':')
    bit_len = int(bit_len_str[:-1])
    if lead.endswith('S'): signed = 1
    else: signed = 0
    val = int(lead[:-1])
    return SASS_Bits.from_int(val=val, bit_len=bit_len, signed=signed)

# Convenience method to change the value of one given enc_vals field
def overwrite_helper(val:tuple|int|bool, val_str:str, enc_vals:dict):
    if isinstance(val, tuple): v = val[-1]
    elif isinstance(val, int): v = val
    elif isinstance(val, bool): v = int(val)
```

```
15      else: raise Exception(sp.CONST__ERROR_ILLEGAL)
16      v_old:SASS_Bits = enc_vals[val_str]
17      v_new:SASS_Bits = SASS_Bits.from_int(v, bit_len=v_old.bit_len,
18                                           signed=v_old.signed)
19      enc_vals[val_str] = v_new
20      return enc_vals
21
22  class SASS_KK__dfma__RsIR_RIR:
23      def __init__(self, kk_sm:KK_SM,
24                   Pg_negate:bool, Pg:tuple,
25                   Rd:tuple,
26                   Ra_reuse:bool, Ra_absolute:bool, Ra_negate:bool, Ra:tuple,
27                   Sb:int,
28                   Rc_reuse:bool, Rc_absolute:bool, Rc_negate:bool, Rc:tuple,
29                   usched_info_reg:tuple, req:int=0, wr:int=0x7, rd:int=0x7):
30          class_name = "dfma__RsIR_RIR"
31
32          # Uncomment these to print the sample of enc_vals produced
33          # by kk_sm.get_enc_vals(..)
34          # ww, enc_vals = kk_sm.get_enc_vals(class_name, {}, {})
35          # print(SASS_Create_Utils.enc_vals_dict_to_init(enc_vals))
36
37          enc_vals = {
38              # Predicate configuration
39              'Pg@not': sass_bits_from_str('0U:1b'),
40              'Pg': sass_bits_from_str('7U:3b'),
41              # Extensions
42              'fmz': sass_bits_from_str('0U:2b'), # keep 0=.nofmz
43              'rnd': sass_bits_from_str('0U:2b'), # keep 0=.RN
44              'sat': sass_bits_from_str('0U:1b'), # keep 0=.nosat
45              # Destination register
46              'Rd': sass_bits_from_str('33U:8b'),
47              # Source register 1
48              'Ra': sass_bits_from_str('36U:8b'),
49              'Ra@absolute': sass_bits_from_str('1U:1b'),
50              'Ra@negate': sass_bits_from_str('0U:1b'),
51              'reuse_src_a': sass_bits_from_str('0U:1b'),
52              # Source immediate value
53              'Sb': sass_bits_from_str('-1425045178S:32b'),
54              # Source register 3
55              'Rc': sass_bits_from_str('30U:8b'),
56              'Rc@absolute': sass_bits_from_str('1U:1b'),
57              'Rc@negate': sass_bits_from_str('1U:1b'),
58              'reuse_src_c': sass_bits_from_str('0U:1b'),
```

```
59              # Cache and barriers
60              'req_bit_set': sass_bits_from_str('5U:6b'),
61              'dst_wr_sb': sass_bits_from_str('0U:3b'),
62              'src_rel_sb': sass_bits_from_str('0U:3b'),
63              'usched_info': sass_bits_from_str('1U:5b'),
64              # Keep 0
65              'pm_pred': sass_bits_from_str('0U:2b'),
66              'batch_t': sass_bits_from_str('0U:3b')
67          }
68
69          # Overwrite predicate configuration
70          enc_vals = overwrite_helper(Pg_negate, 'Pg@not', enc_vals)
71          enc_vals = overwrite_helper(Pg, 'Pg', enc_vals)
72          # Overwrite destination register
73          enc_vals = overwrite_helper(Rd, 'Rd', enc_vals)
74          # Overwrite source register 1
75          enc_vals = overwrite_helper(Ra, 'Ra', enc_vals)
76          enc_vals = overwrite_helper(Ra_reuse, 'reuse_src_a', enc_vals)
77          enc_vals = overwrite_helper(Ra_absolute, 'Ra@absolute', enc_vals)
78          enc_vals = overwrite_helper(Ra_negate, 'Ra@negate', enc_vals)
79          # Overwrite source immediate value
80          enc_vals = overwrite_helper(Sb, 'Sb', enc_vals)
81          # Overwrite source register 3
82          enc_vals = overwrite_helper(Rc, 'Rc', enc_vals)
83          enc_vals = overwrite_helper(Rc_absolute, 'Rc@absolute', enc_vals)
84          enc_vals = overwrite_helper(Rc_negate, 'Rc@negate', enc_vals)
85          enc_vals = overwrite_helper(Rc_reuse, 'reuse_src_c', enc_vals)
86          # Overwrite cache bits and barriers
87          enc_vals = overwrite_helper(req, 'req_bit_set', enc_vals)
88          enc_vals = overwrite_helper(wr, 'dst_wr_sb', enc_vals)
89          enc_vals = overwrite_helper(rd, 'src_rel_sb', enc_vals)
90          enc_vals = overwrite_helper(usched_info_reg, 'usched_info', enc_vals)
91
92          # Check if the new configuration is valid
93          Instr_CuBin.check_expr_conditions(kk_sm.sass.sm.classes_dict[class_name],
94                                            enc_vals, throw=True)
95
96          self.class_name = class_name
97          self.enc_vals = enc_vals
```

A.3 Full simple C++ Template

This full example showcases a good way to use a kernel template with a good range of input and output arguments as well as a good way to call one specific Cuda kernel using a map with names and addresses. One cannot assume that the Nvidia Cuda compiler will leave the names of the kernels ”untouched” enough to deterministically change them with a Python script and then call the correct one with the desired inputs.

We use a map with the kernel name and its address, using the C++ name of the kernel for this purpose. Then we can use the name used in the template, for example *Kernel_0* to call the kernel we want with a well defined name. The name *Kernel_0* can be passed from the terminal to the program, combined with a set of parameter values.

Recall that in *Python* we can only really access a kernel using an index and only have the compiled kernel name available that may not completely contain the kernel name in C++. Usually the compiler only adds prefixes and suffixes. For example *Kernel_0* may be transformed into *_Z8Kernel_0jPmS_PdS_SO_S_Pf* and sorting by name looses some meaning. Creating templates with multiple different kinds of Kernels in it that should be used differently based on which kernel it is, may thus not be as trivial as it seems.

```
1  /**
2   *
3   * ============================================================================
4   * IMPORTANT: This file is autogenerated. Do not change!!
5   * ============================================================================
6   * [KernelCount]1[/KernelCount]
7   */
8
9  #include "cuda_runtime.h"
10 #include "device_launch_parameters.h"
11 #include <iostream>
12 #include <fstream>
13 #include <algorithm>
14 #include <thread>
15 #include <format>
16 #include <vector>
17 #include <chrono>
18 #include <map>
19 #include <string>
20 #include <unordered_map>
21 #include <stdint.h>
22
```

```
23  // Kernel template 0
24  // =================
25  __global__ void
26  Kernel_0(unsigned int a, uint64_t *control, uint64_t *ui_output,
27         double* d_output, uint64_t* ui_input, double* d_input,
28         uint64_t *clk_out_1, float* f_output) {
29      #pragma unroll 2
30      for(unsigned int i=0; i<a; ++i){
31          int64_t t1 = clock64();
32          int64_t t2 = clock64();
33          f_output[i] = static_cast<float>(a
34                                * (static_cast<float>(t2-t1) + 1.256f));
35      }
36      return;
37  }
38
39  // Kernel template 1
40  // =================
41  __global__ void
42  Kernel_1(unsigned int a, uint64_t *control, uint64_t *ui_output,
43         double* d_output, uint64_t* ui_input, double* d_input,
44         uint64_t *clk_out_1, float* f_output) {
45      /* A second kernel, doing things */
46      return;
47  }
48
49  /**
50   * We like to print result vectors with a "label"
51   * [Label]...[/Label]
52   * Then we can 'snip' the printed output out of a
53   * large string in Python in a well defined way
54   * and process the kernel output.
55   */
56  template<class T>
57  void print_out(const std::string& prefix,
58                 const std::string& label,
59                 const std::vector<T>& data) {
60      size_t ss = data.size();
61      std::cout << prefix << "[" << label << "]";
62      for(int i=0; i<ss; ++i){
63          std::cout << data[i];
64          if(i<(ss-1)) std::cout << ", ";
65      }
66      std::cout << "[/" << label << "]" << std::endl;
```

```
67  }
68
69  void print_out_single(const std::string& prefix,
70                        const std::string& label,
71                        const char* data) {
72      std::cout << prefix << "[" << label << "]"
73                << data
74                << "[/" << label << "]" << std::endl;
75  }
76
77  /**
78   * One method to run them all => save some code lines!
79   * The Cuda compiler needs more RAM for 100'000 than
80   * 1000 lines of code and we prefer stuffing as many
81   * kernels as we can in one binary rather than be as
82   * verbose as possible.
83   */
84  int run_test(void* kernel_ptr, int input, int loop_count,
85               const std::string& filename, int kernel_index,
86               const std::string& kernel_name, const std::string& enc_vals,
87               const std::vector<uint64_t>& ui_input_params){
88      std::cout << "[Kernel_" << kernel_index << "]" << std::endl;
89
90      // print enc vals
91      std::cout << "[EncVals]" << std::endl;
92      std::cout << kernel_name << "|" << enc_vals << std::endl;
93      std::cout << "[/EncVals]" << std::endl;
94
95      uint64_t addr = reinterpret_cast<uint64_t>(kernel_ptr);
96      std::cout << "[KernelAddress]0x"
97                << std::format("{:x}", addr)
98                << "[/KernelAddress]" << std::endl;
99
100     for(int lp=0; lp<loop_count; ++lp){
101         std::cout << "   [LoopCount_" << lp << "]" << std::endl;
102         // ================================================================
103         // Define input data
104         // Initializing the input data with some meaningful, non-zero values
105         // makes it easier to distinguish good from bad values. Usually,
106         // bad values are either 0 or a very large number.
107         // ================================================================
108         int control_size = 15;
109         int printout = loop_count;
110         std::vector<uint64_t> control(control_size);
```

```
111         std::vector<uint64_t> ui_output(printout);
112         std::vector<double> d_output(printout);
113         std::vector<float> f_output(printout);
114         std::vector<uint64_t> ui_input(printout);
115         std::vector<double> d_input(printout);
116         std::vector<uint64_t> clk_out_1(printout);
117         for(int i=0; i<control.size(); ++i) control[i] = 0;
118         for(int i=0; i<ui_output.size(); ++i) ui_output[i] = 0;
119         for(int i=0; i<d_output.size(); ++i) d_output[i] = 0.0;
120         for(int i=0; i<f_output.size(); ++i) {
121             f_output[i] = static_cast<float>(i+10001);
122         }
123         // Use some of user defined input, if it exists
124         for(size_t i=0; i<ui_input_params.size(); ++i) {
125             ui_input[i] = ui_input_params[i];
126         }
127         for(size_t i=ui_input_params.size(); i<ui_input.size(); ++i) {
128             ui_input[i] = i+10001;
129         }
130         for(int i=0; i<d_input.size(); ++i) {
131             d_input[i] = static_cast<double>(i+10001);
132         }
133         for(int i=0; i<clk_out_1.size(); ++i) clk_out_1[i] = 999999998;
134 
135         // ================================================================
136         // Print input data
137         // ================================================================
138         std::cout << "      [BeforeKernel]" << std::endl;
139         print_out("         ", "Control", control);
140         print_out("         ", "UiOutput", ui_output);
141         print_out("         ", "DOutput", d_output);
142         print_out("         ", "FOutput", f_output);
143         print_out("         ", "UiInput", ui_input);
144         print_out("         ", "DInput", d_input);
145         print_out("         ", "ClkOut1", clk_out_1);
146 
147         // ================================================================
148         // Copy input data to GPU
149         // ================================================================
150         uint64_t* device_control;
151         cudaMalloc(&device_control, control.size()*sizeof(uint64_t));
152         cudaMemcpy(device_control, control.data(),
153                    control.size()*sizeof(uint64_t), cudaMemcpyHostToDevice);
154 
```

```
155         uint64_t* device_ui_output;
156         cudaMalloc(&device_ui_output, ui_output.size()*sizeof(uint64_t));
157         cudaMemcpy(device_ui_output, ui_output.data(),
158                    ui_output.size()*sizeof(uint64_t), cudaMemcpyHostToDevice);
159 
160         double* device_d_output;
161         cudaMalloc(&device_d_output, d_output.size()*sizeof(double));
162         cudaMemcpy(device_d_output, d_output.data(),
163                    d_output.size()*sizeof(double), cudaMemcpyHostToDevice);
164 
165         float* device_f_output;
166         cudaMalloc(&device_f_output, f_output.size()*sizeof(float));
167         cudaMemcpy(device_f_output, f_output.data(),
168                    f_output.size()*sizeof(float), cudaMemcpyHostToDevice);
169 
170         uint64_t* device_ui_input;
171         cudaMalloc(&device_ui_input, ui_input.size()*sizeof(uint64_t));
172         cudaMemcpy(device_ui_input, ui_input.data(),
173                    ui_input.size()*sizeof(uint64_t), cudaMemcpyHostToDevice);
174 
175         double* device_d_input;
176         cudaMalloc(&device_d_input, d_input.size()*sizeof(double));
177         cudaMemcpy(device_d_input, d_input.data(),
178                    d_input.size()*sizeof(double), cudaMemcpyHostToDevice);
179 
180         uint64_t* device_clk_out_1;
181         cudaMalloc(&device_clk_out_1, clk_out_1.size()*sizeof(uint64_t));
182         cudaMemcpy(device_clk_out_1, clk_out_1.data(),
183                    clk_out_1.size()*sizeof(uint64_t), cudaMemcpyHostToDevice);
184 
185         std::cout << "      [/BeforeKernel]" << std::endl;
186         // ================================================================
187         // Run Kernel
188         // ================================================================
189         void* args[] = {&input, &device_control, &device_ui_output,
190                         &device_d_output, &device_ui_input,
191                         &device_d_input, &device_clk_out_1,
192                         &device_f_output};
193         cudaError_t err;
194         err = cudaLaunchKernel(kernel_ptr, 1, 1, args, 0, nullptr);
195         // Maybe there is some trouble launching the kernel?
196         if ( err != cudaSuccess ) {
197             print_out_single("         ", "CUDAError", cudaGetErrorString(err));
198         }
```

```
199
200         // DO NOT forget this one!
201         err = cudaDeviceSynchronize();
202         if ( err != cudaSuccess ) {
203             print_out_single("         ", "CUDAError", cudaGetErrorString(err));
204         }
205
206         // Maybe one too much? :-D
207         err = cudaGetLastError();
208         if ( err != cudaSuccess ) {
209             print_out_single("         ", "CUDAError", cudaGetErrorString(err));
210         }
211
212         std::cout << "      [AfterKernel]" << std::endl;
213
214         // ================================================================
215         // Copy output data to CPU
216         // ================================================================
217         cudaMemcpy(control.data(), device_control,
218                    control.size()*sizeof(uint64_t), cudaMemcpyDeviceToHost);
219         cudaMemcpy(ui_output.data(), device_ui_output,
220                  ui_output.size()*sizeof(uint64_t), cudaMemcpyDeviceToHost);
221         cudaMemcpy(d_output.data(), device_d_output,
222                    d_output.size()*sizeof(double), cudaMemcpyDeviceToHost);
223         cudaMemcpy(f_output.data(), device_f_output,
224                    f_output.size()*sizeof(float), cudaMemcpyDeviceToHost);
225         cudaMemcpy(ui_input.data(), device_ui_input,
226                   ui_input.size()*sizeof(uint64_t), cudaMemcpyDeviceToHost);
227         cudaMemcpy(d_input.data(), device_d_input,
228                    d_input.size()*sizeof(double), cudaMemcpyDeviceToHost);
229         cudaMemcpy(clk_out_1.data(), device_clk_out_1,
230                  clk_out_1.size()*sizeof(uint64_t), cudaMemcpyDeviceToHost);
231         cudaFree(device_control);
232         cudaFree(device_ui_output);
233         cudaFree(device_d_output);
234         cudaFree(device_f_output);
235         cudaFree(device_ui_input);
236         cudaFree(device_d_input);
237         cudaFree(device_clk_out_1);
238
239         // ================================================================
240         // Print output data
241         // ================================================================
242         print_out("         ", "Control", control);
```

```
243         print_out("          ", "UiOutput", ui_output);
244         print_out("          ", "DOutput", d_output);
245         print_out("          ", "FOutput", f_output);
246         print_out("          ", "UiInput", ui_input);
247         print_out("          ", "DInput", d_input);
248         print_out("          ", "ClkOut1", clk_out_1);
249 
250         std::cout << "      [/AfterKernel]" << std::endl;
251         std::cout << "   [/LoopCount_" << lp << "]" << std::endl;
252     }
253     std::cout << "[/Kernel_" << kernel_index << "]" << std::endl;
254 
255     return 0;
256 }
257 
258 int main(int argc, char** argv){
259     // The first arg is the path to the binary name
260     std::string fn = std::string(argv[0]);
261     int split_ind = fn.find_last_of("/") + 1;
262     std::string path = fn.substr(0,split_ind);
263     std::string filename = fn.substr(split_ind);
264 
265     if(argc != 3){
266         std::cout << std::vformat("{0} [input] [bin_location]",
267                                             std::make_format_args(fn))
268                   << std::endl;
269         std::cout << " * [input]: an integer denoting how many inner
270                                                 loops all kernels do"
271                   << std::endl;
272         return 0;
273     }
274     // .. then maybe an integer?
275     unsigned int input = static_cast<unsigned int>(std::stoi(argv[1]));
276     int loop_count = (input == 0) ? input++ : input;
277 
278     // Now which kernel
279     std::string k_name = std::string(argv[2]);
280 
281     std::cout << "[Filename]"
282               << filename
283               << "[/Filename] [Input]" << input << "[/Input] ";
284 
285     // We construct this map using a Python script where we can type
286     // the number of kernels we like, maybe also have some differnt
```

```
287     // kernel templates available. Then we know both the name of the
288     // kernel and its address at compile time.
289     std::map<std::string, void*> kernel_ptr = {
290                     {"Kernel_0", reinterpret_cast<void*>(&Kernel_0)},
291                     {"Kernel_1", reinterpret_cast<void*>(&Kernel_1)}
292     };
293 
294     if(kernel_ptr.find(k_name) == kernel_ptr.end()) {
295        std::cout << "Kernel "
296                  << k_name << " doesn't exist"
297                  << std::endl;
298     }
299 
300     std::vector<uint64_t> ui_input_params;
301     for(int i=2; i<argc; ++i) {
302         ui_input_params.push_back(std::stoi(argv[i]));
303     }
304 
305     run_test(kernel_ptr["Kernel_0"], input, loop_count,
306                                      filename, 0, "Kernel_0", "",
307                                      ui_input_params);
308 
309     return 0;
310 }
```

A.4 Simple Kernel

This script is used in the tutorial example in Section 6.4.

```
1  import os
2  from py_cubin import SM_CuBin_File
3  from kk_sm import KK_SM
4  import sass_create_86 as sc
5  # import sass_create_75 as sc
6  
7  class Kernel:
8      def create_whipe(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
9          # whipe=True replaces all instructions with NOP implicitly
10         target_cubin = SM_CuBin_File(kk_sm.sass, template, whipe=True)
11 
12         nrc = target_cubin.reg_count()
13         nnrc = {n:k+10 for n,k in nrc.items()}
```

```
14          target_cubin.overwrite_reg_count(nnrc)
15
16          WAIT15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
17
18          i=0
19          empty = sc.SASS_KK__Empty(kk_sm, WAIT15)
20          # ================================================================
21          # Add initial mov instruction
22          target_cubin.create_instr(0, i,
23                                    empty.class_name__mov, empty.enc_vals__mov)
24          i+=1
25          # ================================================================
26          # Add final exit and bra instructions
27          target_cubin.create_instr(0, i,
28                                    empty.class_name__exit, empty.enc_vals__exit)
29          i+=1
30          target_cubin.create_instr(0, i,
31                                    empty.class_name__bra, empty.enc_vals__bra)
32          i+=1
33
34          return target_cubin
35
36      def create(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
37          target_cubin = SM_CuBin_File(kk_sm.sass, template)
38
39          nrc = target_cubin.reg_count()
40          nnrc = {n:k+10 for n,k in nrc.items()}
41          target_cubin.overwrite_reg_count(nnrc)
42
43          # Create one NOP instruction that can be multiplied to everywhere
44          nop = sc.SASS_KK__NOP(kk_sm)
45          for i in range(0, len(target_cubin[0])):
46              target_cubin.create_instr(0, i, nop.class_name, nop.enc_vals)
47
48          WAIT15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
49
50          i=0
51          empty = sc.SASS_KK__Empty(kk_sm, WAIT15)
52          # ================================================================
53          # Add initial mov instruction
54          target_cubin.create_instr(0, i,
55                                    empty.class_name__mov, empty.enc_vals__mov)
56          i+=1
57          # ================================================================
```

```
58          # Add final exit and bra instructions
59          target_cubin.create_instr(0, i,
60                                    empty.class_name__exit, empty.enc_vals__exit)
61          i+=1
62          target_cubin.create_instr(0, i,
63                                    empty.class_name__bra, empty.enc_vals__bra)
64          i+=1
65
66          return target_cubin
67
68
69      def __init__(self, sm_nr:int):
70          kk_sm = KK_SM(sm_nr, ip="127.0.0.1", port=8180, webload=True)
71          t_location = os.path.dirname(os.path.realpath(__file__))
72          template_location =
73             "{0}/template_projects/template_1k/benchmark_binaries".format(t_location)
74          template = "{0}/template_1k_{1}".format(template_location, sm_nr)
75          modified_location = "{0}/tutorials_binaries".format(t_location)
76          exe_name =
77             "{0}/tutorial_0_simple_kernel_{1}".format(modified_location, sm_nr)
78
79          modified_file:SM_CuBin_File = self.create(kk_sm, template)
80          modified_file.to_exec(exe_name)
81          os.system("chmod +x {0}".format(exe_name))
82
83
84  if __name__ == "__main__":
85      b = Kernel(86)
86      print("Finished")
87      pass
```

A.5 Kernel with 1 Value- and 7 Pointer Arguments

This script is used in the tutorial example in Section 6.6.

```
1   import os
2   from py_cubin import SM_CuBin_File
3   from kk_sm import KK_SM
4   import sass_create_86 as sc
5
6   class Kernel:
7       def create(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
```

```
8         target_cubin = SM_CuBin_File(kk_sm.sass, template, whipe=True)
9
10        nrc = target_cubin.reg_count()
11        nnrc = {n:k+10 for n,k in nrc.items()}
12        target_cubin.overwrite_reg_count(nnrc)
13
14        # Zero and True: we need them all the time
15        RZ = kk_sm.regs.Register__RZ__255
16        PT = kk_sm.regs.Predicate__PT__7
17        # Some temp blabla
18        staging_R2 = kk_sm.regs.Register__R2__2
19
20        # ====================================================
21        # It is a good idea to initialize registers in this
22        # way to make things more readable. The kk_sm.regs
23        # are tuples with 3 entries:
24        # UniformRegister__UR2__2 = ('UniformRegister', 'UR2', 2)
25        #  - register category
26        #  - register name
27        #  - register number (can be different than register name!)
28        wait15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
29        a_UR2 = kk_sm.regs.UniformRegister__UR2__2
30        control_UR4 = kk_sm.regs.UniformRegister__UR4__4
31        ui_output_UR6 = kk_sm.regs.UniformRegister__UR6__6
32        d_output_UR8 = kk_sm.regs.UniformRegister__UR8__8
33        ui_input_UR10 = kk_sm.regs.UniformRegister__UR10__10
34        d_input_UR12 = kk_sm.regs.UniformRegister__UR12__12
35        clk_UR14 = kk_sm.regs.UniformRegister__UR14__14
36        f_output_UR16 = kk_sm.regs.UniformRegister__UR16__16
37
38        # 1. load all ptr registers
39        # 2. load first value
40        # 3. deal with floats
41        # => create a two int kernel thing...
42
43        a_R4 = kk_sm.regs.Register__R4__4
44        uii0_R6 = kk_sm.regs.Register__R6__6
45
46        # Kernel index
47        ki = 0
48        i=0
49        empty = sc.SASS_KK__Empty(kk_sm, wait15)
50        # ====================================================
51        # Add initial mov instruction
```

```
52          target_cubin.create_instr(ki, i,
53                                    empty.class_name__mov,
54                                    empty.enc_vals__mov)
55          i+=1
56
57          # ====================================================
58          # Load unsigned int once using ULDG and another time
59          # using IMAD (equivalent)
60          # With ULDC we can pass the size, which makes it
61          # easier to deal with larger data types, it is more
62          # cumbersome if we have value types. In this instance
63          # the value of a is stored into U2, and there is a lack
64          # of nice instructions to do things with it. Thus, IMAD
65          # is the better choice.
66          # Generally, the UniformRegister category is considered
67          # a 'distinct data path' and is most often used to deal
68          # with addresses.
69          ii = sc.SASS_KK__ULDC(kk_sm,
70                                uniform_reg=a_UR2,
71                                m_offset=0x160,
72                                usched_info_reg=wait15, size=32)
73          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
74          i+=1
75          ii = sc.SASS_KK__IMAD_RRC_RRC(kk_sm,
76                                        Rd=a_R4, Ra=RZ, Rb=RZ,
77                                        m_bank=0x0, m_bank_offset=0x160,
78                                        usched_info_reg=wait15)
79          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
80          i+=1
81
82          # ====================================================
83          # For pointer types, using ULDC is the best option. It
84          # exists on all architectures starting with SM 75 and
85          # creates an uniform register with the base address.
86          # There are a lot of useful instructions to do things
87          # with this construct.
88          ii = sc.SASS_KK__ULDC(kk_sm,
89                                uniform_reg=control_UR4,
90                                m_offset=0x168,
91                                usched_info_reg=wait15, size=64)
92          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
93          i+=1
94          ii = sc.SASS_KK__ULDC(kk_sm,
95                                uniform_reg=ui_output_UR6,
```

```
96                              m_offset=0x170,
97                              usched_info_reg=wait15, size=64)
98          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
99          i+=1
100         ii = sc.SASS_KK__ULDC(kk_sm,
101                             uniform_reg=d_output_UR8,
102                             m_offset=0x178,
103                             usched_info_reg=wait15, size=64)
104         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
105         i+=1
106         ii = sc.SASS_KK__ULDC(kk_sm,
107                             uniform_reg=ui_input_UR10,
108                             m_offset=0x180,
109                             usched_info_reg=wait15, size=64)
110         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
111         i+=1
112         ii = sc.SASS_KK__ULDC(kk_sm,
113                             uniform_reg=d_input_UR12,
114                             m_offset=0x188,
115                             usched_info_reg=wait15, size=64)
116         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
117         i+=1
118         ii = sc.SASS_KK__ULDC(kk_sm,
119                             uniform_reg=clk_UR14,
120                             m_offset=0x190,
121                             usched_info_reg=wait15, size=64)
122         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
123         i+=1
124         ii = sc.SASS_KK__ULDC(kk_sm, uniform_reg=f_output_UR16,
125                             m_offset=0x198,
126                             usched_info_reg=wait15, size=64)
127         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
128         i+=1
129
130         # ================================================================
131         # Store the uint32 value a to the control register in second place
132         # with offset 0x8.
133         # NOTE that we initialize the control array with all 0, thus just
134         # writing 32 bits into a 64 bit location is fine. If we didn't
135         # initialize the control register, this could cause trouble and
136         # produce garbage for the 32 LSB of the target location.
137         ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
138                                   offset=0x8,
139                                   source_reg=a_R4,
```

```
                                usched_info_reg=wait15,
                                req=0x0, rd=0x7,
                                size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================================
        # Write 815 in control array location index 0.
        ii = sc.SASS_KK__MOVImm(kk_sm,
                                exec_pred_inv=False,
                                exec_pred=PT,
                                target_reg=staging_R2,
                                imm_val=815,
                                req=0x0,
                                usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        # Use the size modifier size=32, since R2 is a 32 bit spot. Also
        # NOTE that we initialize the control array with all 0, thus just
        # writing 32 bits into a 64 bit location is fine. If we didn't
        # initialize the control register, this could cause trouble and
        # produce garbage for the 32 LSB of the target location.
        ii = sc.SASS_KK__STG_RaRZ(kk_sm,
                                uniform_reg=control_UR4,
                                offset=0x0,
                                source_reg=staging_R2,
                                usched_info_reg=wait15,
                                req=0x0, rd=0x0,
                                size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================================
        # Use the base uniform register for uint64* ui_input to load the
        # first value into R6 using LDG and transfer it to the ui_output
        # array using STG. NOTE the barrier WR in LDG and req in STG!
        ii = sc.SASS_KK__ldg_uniform_RaRZ(kk_sm,
                                  Rd=uii0_R6, Ra=RZ,
                                  Ra_URb=ui_input_UR10, ra_offset=0x0,
                                  RD=0x7, WR=0x0, REQ=0x0,
                                  USCHED_INFO=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__STG_RaRZ(kk_sm,
```

```
                                    uniform_reg=ui_output_UR6,
                                    offset=0x0,
                                    source_reg=uii0_R6,
                                    usched_info_reg=wait15,
                                    req=0b000001, rd=0x7,
                                    size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================================
        # Add final exit and bra instructions
        target_cubin.create_instr(ki, i,
                                  empty.class_name__exit,
                                  empty.enc_vals__exit)
        i+=1
        target_cubin.create_instr(ki, i,
                                  empty.class_name__bra,
                                  empty.enc_vals__bra)
        i+=1

        return target_cubin


    def __init__(self, sm_nr:int):
        kk_sm = KK_SM(sm_nr, ip='127.0.0.1', port=8180, webload=True)
        t_location = os.path.dirname(os.path.realpath(__file__))
        temp_loc = '{0}/template_projects/template_1k'.format(t_location)
        template = '{0}/benchmark_binaries/template_1k_{1}'.format(temp_loc, sm_nr)
        mod_loc = '{0}/tutorials_binaries'.format(t_location)
        exe_name = "{0}/tutorial_1_load_kernel_args_{1}".format(mod_loc, sm_nr)

        modified_file:SM_CuBin_File = self.create(kk_sm, template)
        modified_file.to_exec(exe_name)
        os.system('chmod +x {0}'.format(exe_name))


if __name__ == '__main__':
    b = Kernel(86)
    print("Finished")
    pass
```

A.6 Kernel with float-, double-, and MUFU operations

This script is used in the tutorial example in Section 6.8. Note that this example uses a different template than the one showcased in Appendix A.3!

```
import os
from py_cubin import SM_CuBin_File
from kk_sm import KK_SM
import sass_create_86 as sc

class Kernel:
    def create(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
        target_cubin = SM_CuBin_File(kk_sm.sass, template, whipe=True)

        nrc = target_cubin.reg_count()
        nnrc = {n:100 for n,k in nrc.items()}
        target_cubin.overwrite_reg_count(nnrc)

        # Zero and True: we need them all the time
        RZ = kk_sm.regs.Register__RZ__255
        PT = kk_sm.regs.Predicate__PT__7
        # Some temp blabla
        staging_R2 = kk_sm.regs.Register__R2__2

        # ======================================================
        # It is a good idea to initialize registers in this
        # way to make things more readable. The kk_sm.regs
        # are tuples with 3 entries:
        # UniformRegister__UR2__2 = ('UniformRegister', 'UR2', 2)
        #  - register category
        #  - register name
        #  - register number (can be different than register name!)
        wait15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
        wait1 = kk_sm.regs.USCHED_INFO__WAIT1_END_GROUP__1
        a_UR2 = kk_sm.regs.UniformRegister__UR2__2
        control_UR4 = kk_sm.regs.UniformRegister__UR4__4
        ui_output_UR6 = kk_sm.regs.UniformRegister__UR6__6
        d_output_UR8 = kk_sm.regs.UniformRegister__UR8__8
        ui_input_UR10 = kk_sm.regs.UniformRegister__UR10__10
        d_input_UR12 = kk_sm.regs.UniformRegister__UR12__12
        clk_UR14 = kk_sm.regs.UniformRegister__UR14__14
        f_output_UR16 = kk_sm.regs.UniformRegister__UR16__16

        # 1. load all ptr registers
        # 2. load first value
```

```
41          # 3. deal with floats
42          # => create a two int kernel thing...
43
44          a_R4 = kk_sm.regs.Register__R4__4
45          af32_R6 = kk_sm.regs.Register__R6__6
46          af64_R8 = kk_sm.regs.Register__R8__8
47          ui0_R14_i32 = kk_sm.regs.Register__R14__14
48          ui0_R10_f32 = kk_sm.regs.Register__R10__10
49          ui0_R12_f64 = kk_sm.regs.Register__R12__12
50          ui0_rcp32_R16 = kk_sm.regs.Register__R16__16
51          ui0_rcp64_R18 = kk_sm.regs.Register__R18__18
52          a32_d_ui032_R20 = kk_sm.regs.Register__R20__20
53          a64_d_ui064_R22 = kk_sm.regs.Register__R22__22
54
55          # Kernel index
56          ki = 0
57          i=0
58          empty = sc.SASS_KK__Empty(kk_sm, wait15)
59          # =====================================================
60          # Add initial mov instruction
61          target_cubin.create_instr(ki, i,
62                                    empty.class_name__mov,
63                                    empty.enc_vals__mov)
64          i+=1
65
66          # =====================================================
67          # Load unsigned int once using ULDG and another time
68          # using IMAD (equivalent)
69          # With ULDC we can pass the size, which makes it
70          # easier to deal with larger data types, it is more
71          # cumbersome if we have value types. In this instance
72          # the value of a is stored into U2, and there is a lack
73          # of nice instructions to do things with it. Thus, IMAD
74          # is the better choice.
75          # Generally, the UniformRegister category is considered
76          # a 'distinct data path' and is most often used to deal
77          # with addresses.
78          ii = sc.SASS_KK__ULDC(kk_sm,
79                                uniform_reg=a_UR2,
80                                m_offset=0x160,
81                                usched_info_reg=wait15, size=32)
82          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
83          i+=1
84          ii = sc.SASS_KK__IMAD_RRC_RRC(kk_sm,
```

```
                                    Rd=a_R4, Ra=RZ, Rb=RZ,
                                    m_bank=0x0, m_bank_offset=0x160,
                                    usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================
        # For pointer types, using ULDC is the best option. It
        # exists on all architectures starting with SM 75 and
        # creates an uniform register with the base address.
        # There are a lot of useful instructions to do things
        # with this construct.
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=control_UR4,
                              m_offset=0x168,
                              usched_info_reg=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=ui_output_UR6,
                              m_offset=0x170,
                              usched_info_reg=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=d_output_UR8,
                              m_offset=0x178,
                              usched_info_reg=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=ui_input_UR10,
                              m_offset=0x180,
                              usched_info_reg=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=d_input_UR12,
                              m_offset=0x188,
                              usched_info_reg=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=clk_UR14,
```

```
129                             m_offset=0x190,
130                             usched_info_reg=wait15, size=64)
131         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
132         i+=1
133         ii = sc.SASS_KK__ULDC(kk_sm, uniform_reg=f_output_UR16,
134                             m_offset=0x198,
135                             usched_info_reg=wait15, size=64)
136         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
137         i+=1
138
139         # ======================================================================
140         # ======================================================================
141         # == Some sample computations with floats ==============================
142         # ======================================================================
143         # ======================================================================
144         # Do some float computations and write into the d_output and f_output
145         # registers
146
147         # First, we convert the unsigned int a to a 32 and a 64 bit float
148         ii = sc.SASS_KK__i2f__Rb_32b(kk_sm, Pg_negate=False, Pg=PT,
149                                 Rd=af32_R6, dst_fsize=32,
150                                 Rb_signed=False, Rb=a_R4,
151                                 usched_info_reg=wait15, rd=0x0)
152         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
153         i+=1
154         ii = sc.SASS_KK__i2f_Rd64__Rb_32b(kk_sm, Pg_negate=False, Pg=PT,
155                                 Rd=af64_R8,
156                                 Rb_signed=False, Rb=a_R4,
157                                 usched_info_reg=wait15, rd=0x1)
158         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
159         i+=1
160
161         # Then we use the first ui_input entry and do the same with it
162         ii = sc.SASS_KK__ldg_uniform_RaRZ(kk_sm, Rd=ui0_R14_i32,
163                                     Ra=RZ, Ra_URb=ui_input_UR10, ra_offset=0x0,
164                                     USCHED_INFO=wait15,
165                                     WR=0x2, RD=0x7, REQ=0x0, size=32)
166         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
167         i+=1
168
169         ii = sc.SASS_KK__i2f__Rb_32b(kk_sm, Pg_negate=False, Pg=PT,
170                                 Rd=ui0_R10_f32, dst_fsize=32,
171                                 Rb_signed=False, Rb=ui0_R14_i32,
172                                 usched_info_reg=wait15, rd=0x2, req=0b000100)
```

```
173         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
174         i+=1
175         ii = sc.SASS_KK__i2f_Rd64__Rb_32b(kk_sm, Pg_negate=False, Pg=PT,
176                                 Rd=ui0_R12_f64,
177                                 Rb_signed=False, Rb=ui0_R14_i32,
178                                 usched_info_reg=wait15, rd=0x3, req=0b000100)
179         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
180         i+=1
181
182         # Write the results into the output register for floats and doubles
183         # respectively in first and second positions
184         # NOTE: the size param is really important here. The wrong one
185         # definitely produces garbage
186         # NOTE: STG uses the RD barrier
187         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
188                                   uniform_reg=f_output_UR16, offset=0x0,
189                                   source_reg=af32_R6,
190                                   usched_info_reg=wait15,
191                                   req=0b111111,
192                                   rd=0x0,
193                                   size=32)
194         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
195         i+=1
196         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
197                                   uniform_reg=d_output_UR8, offset=0x0,
198                                   source_reg=af64_R8,
199                                   usched_info_reg=wait15,
200                                   req=0b111111,
201                                   rd=0x0,
202                                   size=64)
203         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
204         i+=1
205         # NOTE: second position for the floats is 0x4, not 0x8
206         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
207                                   uniform_reg=f_output_UR16, offset=0x4,
208                                   source_reg=ui0_R10_f32,
209                                   usched_info_reg=wait15,
210                                   req=0b111111,
211                                   rd=0x0,
212                                   size=32)
213         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
214         i+=1
215         # NOTE: second position for the doubles is 0x8
216         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
```

```
                                uniform_reg=d_output_UR8, offset=0x8,
                                source_reg=ui0_R12_f64,
                                usched_info_reg=wait15,
                                req=0b111111,
                                rd=0x0,
                                size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # After checking that we indeed got the correct outputs in the
        # first and second positions of the float and double output
        # arrays we do some division with it.
        # First we calculate the reciprocal value of the second output.
        # NOTE: this functionality only seems to exist for 32 or 16
        # bit floats (at least on SM 86) and we have to upscale the
        # result to use it
        # as 64 bit float
        ii = sc.SASS_KK__mufu__RRR_RR(kk_sm,
                    Pg_negate=False, Pg=PT,
                    mufuop=kk_sm.regs.MUFU_OP__RCP__4,
                    Rd=ui0_rcp32_R16,
                    Rb_absolute=False, Rb_negate=False, Rb=ui0_R10_f32,
                    usched_info_reg=wait15, req=0x0, rd=0x7, wr=0x0)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__STG_RaRZ(kk_sm,
                    uniform_reg=f_output_UR16, offset=0x8,
                    source_reg=ui0_rcp32_R16,
                    usched_info_reg=wait15,
                    req=0b000001,
                    rd=0x0,
                    size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # Upscale the MUFU result to 64 bits and write it in third
        # location in the doubles output array
        ii = sc.SASS_KK__f2f_f64_upconvert__R_R32_R_RRR(kk_sm,
                    Pg_negate=False, Pg=PT,
                    Rd=ui0_rcp64_R18,
                    Rb_absolute=False, Rb_negate=False, Rb=ui0_rcp32_R16,
                    usched_info_reg=wait15, req=0x0, rd=0x7, wr=0x0)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
```

```
261         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
262                                   uniform_reg=d_output_UR8, offset=0x10,
263                                   source_reg=ui0_rcp64_R18,
264                                   usched_info_reg=wait15,
265                                   req=0b000001,
266                                   rd=0x0,
267                                   size=64)
268         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
269         i+=1
270 
271         # # Now we can actually perform a division between inputs a and b and
272         # # and write the result in 4th position in the d and f output arrays.
273         # # We use DMUL and FMUL for that.
274         ii = sc.SASS_KK__fmul__RRR_RR(kk_sm,
275             Pg_negate=False, Pg=PT,
276             Rd=a32_d_ui032_R20,
277             Ra_reuse=False, Ra_absolute=False, Ra_negate=False, Ra=af32_R6,
278             Rb_reuse=False, Rb_absolute=False, Rb_negate=False, Rb=ui0_rcp32_R16,
279             usched_info_reg=wait15, req=0x0)
280         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
281         i+=1
282         ii = sc.SASS_KK__dmul__RRR_RR(kk_sm,
283             Pg_negate=False, Pg=PT,
284             Rd=a64_d_ui064_R22,
285             Ra_reuse=False, Ra_absolute=False, Ra_negate=False, Ra=af64_R8,
286             Rb_reuse=False, Rb_absolute=False, Rb_negate=False, Rb=ui0_rcp64_R18,
287             usched_info_reg=wait15, req=0x0, rd=0x7, wr=0x0)
288         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
289         i+=1
290 
291         # Store the result of the f32 multiplication into the fourth location
292         # of the float output array.
293         # NOTE fmul doesn't have a barrier. It takes 5 cycles.
294         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
295                                   uniform_reg=f_output_UR16, offset=0xc,
296                                   source_reg=a32_d_ui032_R20,
297                                   usched_info_reg=wait15,
298                                   req=0x0,
299                                   rd=0x0,
300                                   size=32)
301         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
302         i+=1
303         # Store the result of the 64 bit multiplication in the fourth
304         # location of the d output array.
```

```
305         # NOTE: dmul has a barrier. Wait for it!
306         ii = sc.SASS_KK__STG_RaRZ(kk_sm,
307                                   uniform_reg=d_output_UR8, offset=0x18,
308                                   source_reg=a64_d_ui064_R22,
309                                   usched_info_reg=wait15,
310                                   req=0b000001,
311                                   rd=0x0,
312                                   size=64)
313         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
314         i+=1
315 
316         # ======================================================================
317         # ======================================================================
318         # ======================================================================
319         # ======================================================================
320         # ======================================================================
321 
322         # Store the uint32 value a to the control register in second place
323         # with offset 0x8.
324         # NOTE that we initialize the control array with all 0, thus just
325         # writing 32 bits into a 64 bit location is fine. If we didn't
326         # initialize the control register, this could cause trouble and
327         # produce garbage for the 32 LSB of the target location.
328         ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
329                                   offset=0x8,
330                                   source_reg=a_R4,
331                                   usched_info_reg=wait15,
332                                   req=0x0, rd=0x7,
333                                   size=32)
334         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
335         i+=1
336 
337         # ======================================================================
338         # Write 815 in control array location index 0.
339         ii = sc.SASS_KK__MOVImm(kk_sm,
340                                 exec_pred_inv=False,
341                                 exec_pred=PT,
342                                 target_reg=staging_R2,
343                                 imm_val=815,
344                                 req=0x0,
345                                 usched_info_reg=wait15)
346         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
347         i+=1
348         ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
```

```
                                    offset=0x0,
                                    source_reg=staging_R2,
                                    usched_info_reg=wait15,
                                    req=0x0, rd=0x7,
                                    size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================================
        # Add final exit and bra instructions
        target_cubin.create_instr(ki, i,
                                  empty.class_name__exit,
                                  empty.enc_vals__exit)
        i+=1
        target_cubin.create_instr(ki, i,
                                  empty.class_name__bra,
                                  empty.enc_vals__bra)
        i+=1

        return target_cubin


    def __init__(self, sm_nr:int):
        kk_sm = KK_SM(sm_nr, ip='127.0.0.1', port=8180, webload=True)
        t_location = os.path.dirname(os.path.realpath(__file__))
        temp_loc = '{0}/template_projects/template_1k_no_loop'
                                                .format(t_location)
        template = '{0}/benchmark_binaries/template_1k_no_loop_{1}'
                                                .format(temp_loc, sm_nr)
        mod_loc = '{0}/tutorials_binaries'.format(t_location)
        exe_name = "{0}/tutorial_2_floats_{1}".format(mod_loc, sm_nr)

        modified_file:SM_CuBin_File = self.create(kk_sm, template)
        modified_file.to_exec(exe_name)
        os.system('chmod +x {0}'.format(exe_name))


if __name__ == '__main__':
    b = Kernel(86)
    print("Finished")
    pass

```

A.7 Measuring Clock Cycles

This script is used in the tutorial example in Section 6.9. Note that this example uses a different template from the one presented in the Appendix A.3! Especially, it requires a template with more than 200 instructions instead of the usual 60.

```
import os
from py_cubin import SM_CuBin_File
from kk_sm import KK_SM
import sass_create_86 as sc

class Kernel:
    def add_clk_measure(self, kk_sm:KK_SM,
                        i:int,                          # instruction index
                        target_cubin:SM_CuBin_File, # target binary
                        wait_reg:tuple,
                        set_req:bool,                   # do we wait?
                        clk_output_ureg:tuple,      # output ureg for clk
                        ui_output_ureg:tuple,       # alternate ureg output
                        ui_input_ureg:tuple,        # alternate ureg output
                        stg_offset:int              # offset into output ureg
        ) -> int:
        wait15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
        wait2 = kk_sm.regs.USCHED_INFO__WAIT2_END_GROUP__2
        RZ = kk_sm.regs.Register__RZ__255
        PT = kk_sm.regs.Predicate__PT__7

        clk_diff_R10 = kk_sm.regs.Register__R10__10
        clk_diffo_R11 = kk_sm.regs.Register__R11__11
        clk1_R6 = kk_sm.regs.Register__R6__6
        clk1o_R7 = kk_sm.regs.Register__R7__7
        clk2_R8 = kk_sm.regs.Register__R8__8
        clk2o_R9 = kk_sm.regs.Register__R9__9
        clk_diff_R10 = kk_sm.regs.Register__R10__10
        staging_R12 = kk_sm.regs.Register__R12__12

        # Make sure we build on zeros
        # Conditionally set a barrier wait requirement if a flag
        # is set.
        ii = sc.SASS_KK__MOVImm(kk_sm,
                            exec_pred_inv=False, exec_pred=PT,
                            target_reg=clk1_R6, imm_val=0x0,
                            usched_info_reg=wait15,
                            req=0b000011 if set_req else 0x0)
```

```
39          target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
40          i+=1
41          ii = sc.SASS_KK__MOVImm(kk_sm,
42                                  exec_pred_inv=False, exec_pred=PT,
43                                  target_reg=clk1o_R7, imm_val=0x0,
44                                  usched_info_reg=wait15,
45                                  req=0x0)
46          target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
47          i+=1
48          ii = sc.SASS_KK__MOVImm(kk_sm,
49                                  exec_pred_inv=False, exec_pred=PT,
50                                  target_reg=clk2_R8, imm_val=0x0,
51                                  usched_info_reg=wait15,
52                                  req=0x0)
53          target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
54          i+=1
55          ii = sc.SASS_KK__MOVImm(kk_sm,
56                                  exec_pred_inv=False, exec_pred=PT,
57                                  target_reg=clk2o_R9, imm_val=0x0,
58                                  usched_info_reg=wait15,
59                                  req=0x0)
60          target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
61          i+=1
62          ii = sc.SASS_KK__MOVImm(kk_sm,
63                                  exec_pred_inv=False, exec_pred=PT,
64                                  target_reg=clk_diffo_R11, imm_val=0x0,
65                                  usched_info_reg=wait15,
66                                  req=0x0)
67          target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
68          i+=1
69
70          # Measure the first clock
71          ii = sc.SASS_KK__CS2R(kk_sm,
72                                target_reg=clk1_R6,
73                                usched_info_reg=wait_reg,
74                                req=0x0)
75          # don't forget to add
76          target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
77          # don't forget to increment
78          i+=1
79          # Move the clock value into another register: we want to
80          # know if the value is correct or not after waitX cycles
81          ii = sc.SASS_KK__IADD3_NOIMM_RRR_RRR(kk_sm,
82                                       target_reg=staging_R12,
```

```
                                negate_Ra=False, src_Ra=clk1_R6,
                                negate_Rb=False, src_Rb=RZ,
                                negate_Rc=False, src_Rc=RZ,
                                usched_info_reg=wait15)
        target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
        i+=1
        # Measure the second clock
        ii = sc.SASS_KK__CS2R(kk_sm,
                              target_reg=clk2_R8,
                              usched_info_reg=wait15, req=0x0)
        target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
        i+=1
        # Compute the difference of the two clocks.
        # Subtract 15 because we don't want the clocks of the MOV
        # instruction in the result
        ii = sc.SASS_KK__IADD3_IMM_RsIR_RIR(kk_sm,
                                        target_reg=clk_diff_R10,
                                        negate_Ra=False, Ra=clk2_R8,
                                        src_imm=-15,
                                        negate_Rc=True, Rc=clk1_R6,
                                        usched_info_reg=wait15)
        target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
        i+=1
        # Store the clock diff in the clock output array.
        # Set a barrier that we can wait for the next time we call this
        # method.
        ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=clk_output_ureg,
                                  offset=stg_offset,
                                  source_reg=clk_diff_R10,
                                  usched_info_reg=wait15,
                                  req=0x0, rd=0x0,
                                  size=32)
        target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=ui_output_ureg,
                                  offset=stg_offset,
                                  source_reg=staging_R12,
                                  usched_info_reg=wait15,
                                  req=0x0, rd=0x1,
                                  size=32)
        target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=ui_input_ureg,
                                  offset=stg_offset,
```

```
127                                     source_reg=clk1_R6,
128                                     usched_info_reg=wait15,
129                                     req=0x0, rd=0x1,
130                                     size=32)
131         target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
132         i+=1
133         return i
134
135     def create(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
136         target_cubin = SM_CuBin_File(kk_sm.sass, template, wipe=True)
137
138         nrc = target_cubin.reg_count()
139         nnrc = {n:100 for n,k in nrc.items()}
140         target_cubin.overwrite_reg_count(nnrc)
141
142         # Zero and True: we need them all the time
143         RZ = kk_sm.regs.Register__RZ__255
144         PT = kk_sm.regs.Predicate__PT__7
145         # Some temp blabla
146         staging_R2 = kk_sm.regs.Register__R2__2
147
148         # ====================================================
149         # It is a good idea to initialize registers in this
150         # way to make things more readable. The kk_sm.regs
151         # are tuples with 3 entries:
152         # UniformRegister__UR2__2 = ('UniformRegister', 'UR2', 2)
153         #  - register category
154         #  - register name
155         #  - register number (can be different than register name!)
156         wait1 = kk_sm.regs.USCHED_INFO__WAIT1_END_GROUP__1
157         wait2 = kk_sm.regs.USCHED_INFO__WAIT2_END_GROUP__2
158         wait3 = kk_sm.regs.USCHED_INFO__WAIT3_END_GROUP__3
159         wait4 = kk_sm.regs.USCHED_INFO__WAIT4_END_GROUP__4
160         wait5 = kk_sm.regs.USCHED_INFO__WAIT5_END_GROUP__5
161         wait6 = kk_sm.regs.USCHED_INFO__WAIT6_END_GROUP__6
162         wait7 = kk_sm.regs.USCHED_INFO__WAIT7_END_GROUP__7
163         wait8 = kk_sm.regs.USCHED_INFO__WAIT8_END_GROUP__8
164         wait9 = kk_sm.regs.USCHED_INFO__WAIT9_END_GROUP__9
165         wait10 = kk_sm.regs.USCHED_INFO__WAIT10_END_GROUP__10
166         wait11 = kk_sm.regs.USCHED_INFO__WAIT11_END_GROUP__11
167         wait12 = kk_sm.regs.USCHED_INFO__WAIT12_END_GROUP__12
168         wait13 = kk_sm.regs.USCHED_INFO__WAIT13_END_GROUP__13
169         wait14 = kk_sm.regs.USCHED_INFO__WAIT14_END_GROUP__14
170         wait15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
```

```
171
172         a_UR2 = kk_sm.regs.UniformRegister__UR2__2
173         control_UR4 = kk_sm.regs.UniformRegister__UR4__4
174         ui_output_UR6 = kk_sm.regs.UniformRegister__UR6__6
175         d_output_UR8 = kk_sm.regs.UniformRegister__UR8__8
176         ui_input_UR10 = kk_sm.regs.UniformRegister__UR10__10
177         d_input_UR12 = kk_sm.regs.UniformRegister__UR12__12
178         clk_UR14 = kk_sm.regs.UniformRegister__UR14__14
179         f_output_UR16 = kk_sm.regs.UniformRegister__UR16__16
180
181         # 1. load all ptr registers
182         # 2. load first value
183         # 3. deal with floats
184         # => create a two int kernel thing...
185
186         a_R4 = kk_sm.regs.Register__R4__4
187
188         # ui0_R10_f32 = kk_sm.regs.Register__R10__10
189         # ui0_R12_f64 = kk_sm.regs.Register__R12__12
190         # ui0_rcp32_R16 = kk_sm.regs.Register__R16__16
191         # ui0_rcp64_R18 = kk_sm.regs.Register__R18__18
192         # a32_d_ui032_R20 = kk_sm.regs.Register__R20__20
193         # a64_d_ui064_R22 = kk_sm.regs.Register__R22__22
194
195         # Kernel index
196         ki = 0
197         i=0
198         empty = sc.SASS_KK__Empty(kk_sm, wait15)
199         # ====================================================
200         # Add initial mov instruction
201         target_cubin.create_instr(ki, i,
202                                   empty.class_name__mov,
203                                   empty.enc_vals__mov)
204         i+=1
205
206         # ====================================================
207         # Load unsigned int once using ULDG and another time
208         # using IMAD (equivalent)
209         # With ULDC we can pass the size, which makes it
210         # easier to deal with larger data types, it is more
211         # cumbersome if we have value types. In this instance
212         # the value of a is stored into U2, and there is a lack
213         # of nice instructions to do things with it. Thus, IMAD
214         # is the better choice.
```

```
215         # Generally, the UniformRegister category is considered
216         # a 'distinct data path' and is most often used to deal
217         # with addresses.
218         ii = sc.SASS_KK__ULDC(kk_sm,
219                               uniform_reg=a_UR2,
220                               m_offset=0x160,
221                               usched_info_reg=wait15, size=32)
222         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
223         i+=1
224         ii = sc.SASS_KK__IMAD_RRC_RRC(kk_sm,
225                                       Rd=a_R4, Ra=RZ, Rb=RZ,
226                                       m_bank=0x0, m_bank_offset=0x160,
227                                       usched_info_reg=wait15)
228         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
229         i+=1
230
231         # ====================================================
232         # For pointer types, using ULDC is the best option. It
233         # exists on all architectures starting with SM 75 and
234         # creates an uniform register with the base address.
235         # There are a lot of useful instructions to do things
236         # with this construct.
237         ii = sc.SASS_KK__ULDC(kk_sm,
238                               uniform_reg=control_UR4,
239                               m_offset=0x168,
240                               usched_info_reg=wait15, size=64)
241         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
242         i+=1
243         ii = sc.SASS_KK__ULDC(kk_sm,
244                               uniform_reg=ui_output_UR6,
245                               m_offset=0x170,
246                               usched_info_reg=wait15, size=64)
247         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
248         i+=1
249         ii = sc.SASS_KK__ULDC(kk_sm,
250                               uniform_reg=d_output_UR8,
251                               m_offset=0x178,
252                               usched_info_reg=wait15, size=64)
253         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
254         i+=1
255         ii = sc.SASS_KK__ULDC(kk_sm,
256                               uniform_reg=ui_input_UR10,
257                               m_offset=0x180,
258                               usched_info_reg=wait15, size=64)
```

```
259         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
260         i+=1
261         ii = sc.SASS_KK__ULDC(kk_sm,
262                               uniform_reg=d_input_UR12,
263                               m_offset=0x188,
264                               usched_info_reg=wait15, size=64)
265         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
266         i+=1
267         ii = sc.SASS_KK__ULDC(kk_sm,
268                               uniform_reg=clk_UR14,
269                               m_offset=0x190,
270                               usched_info_reg=wait15, size=64)
271         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
272         i+=1
273         ii = sc.SASS_KK__ULDC(kk_sm, uniform_reg=f_output_UR16,
274                               m_offset=0x198,
275                               usched_info_reg=wait15, size=64)
276         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
277         i+=1
278
279         # ======================================================================
280         # ======================================================================
281         # == Compute a couple of clock cycles ==================================
282         # ======================================================================
283         # ======================================================================
284         # The process is:
285         #  - write clk1 using 15 cycles
286         #  - write clk2 using 1 to 15 cycles
287         #  - compute clk2 - clk1 using 15 cycles
288         #  - subtract 15 from the difference
289         #  - write the difference in the return array
290
291         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
292                                  wait_reg=wait1,
293                                  clk_output_ureg=clk_UR14,
294                                  ui_output_ureg=ui_output_UR6,
295                                  ui_input_ureg=ui_input_UR10,
296                                  set_req=False,
297                                  stg_offset=0x0)
298         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
299                                  wait_reg=wait2,
300                                  clk_output_ureg=clk_UR14,
301                                  ui_output_ureg=ui_output_UR6,
302                                  ui_input_ureg=ui_input_UR10,
```

```
303                                 set_req=True,
304                                 stg_offset=0x8)
305         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
306                                 wait_reg=wait3,
307                                 clk_output_ureg=clk_UR14,
308                                 ui_output_ureg=ui_output_UR6,
309                                 ui_input_ureg=ui_input_UR10,
310                                 set_req=True,
311                                 stg_offset=0x10)
312         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
313                                 wait_reg=wait4,
314                                 clk_output_ureg=clk_UR14,
315                                 ui_output_ureg=ui_output_UR6,
316                                 ui_input_ureg=ui_input_UR10,
317                                 set_req=True,
318                                 stg_offset=0x18)
319         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
320                                 wait_reg=wait5,
321                                 clk_output_ureg=clk_UR14,
322                                 ui_output_ureg=ui_output_UR6,
323                                 ui_input_ureg=ui_input_UR10,
324                                 set_req=True,
325                                 stg_offset=0x20)
326         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
327                                 wait_reg=wait6,
328                                 clk_output_ureg=clk_UR14,
329                                 ui_output_ureg=ui_output_UR6,
330                                 ui_input_ureg=ui_input_UR10,
331                                 set_req=True,
332                                 stg_offset=0x28)
333         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
334                                 wait_reg=wait7,
335                                 clk_output_ureg=clk_UR14,
336                                 ui_output_ureg=ui_output_UR6,
337                                 ui_input_ureg=ui_input_UR10,
338                                 set_req=True,
339                                 stg_offset=0x30)
340         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
341                                 wait_reg=wait8,
342                                 clk_output_ureg=clk_UR14,
343                                 ui_output_ureg=ui_output_UR6,
344                                 ui_input_ureg=ui_input_UR10,
345                                 set_req=True,
346                                 stg_offset=0x38)
```

```
347         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
348                                  wait_reg=wait9,
349                                  clk_output_ureg=clk_UR14,
350                                  ui_output_ureg=ui_output_UR6,
351                                  ui_input_ureg=ui_input_UR10,
352                                  set_req=True,
353                                  stg_offset=0x40)
354         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
355                                  wait_reg=wait10,
356                                  clk_output_ureg=clk_UR14,
357                                  ui_output_ureg=ui_output_UR6,
358                                  ui_input_ureg=ui_input_UR10,
359                                  set_req=True,
360                                  stg_offset=0x48)
361         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
362                                  wait_reg=wait11,
363                                  clk_output_ureg=clk_UR14,
364                                  ui_output_ureg=ui_output_UR6,
365                                  ui_input_ureg=ui_input_UR10,
366                                  set_req=True,
367                                  stg_offset=0x50)
368         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
369                                  wait_reg=wait12,
370                                  clk_output_ureg=clk_UR14,
371                                  ui_output_ureg=ui_output_UR6,
372                                  ui_input_ureg=ui_input_UR10,
373                                  set_req=True,
374                                  stg_offset=0x58)
375         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
376                                  wait_reg=wait13,
377                                  clk_output_ureg=clk_UR14,
378                                  ui_output_ureg=ui_output_UR6,
379                                  ui_input_ureg=ui_input_UR10,
380                                  set_req=True,
381                                  stg_offset=0x60)
382         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
383                                  wait_reg=wait14,
384                                  clk_output_ureg=clk_UR14,
385                                  ui_output_ureg=ui_output_UR6,
386                                  ui_input_ureg=ui_input_UR10,
387                                  set_req=True,
388                                  stg_offset=0x68)
389         i = self.add_clk_measure(kk_sm, i=i, target_cubin=target_cubin,
390                                  wait_reg=wait15,
```

```
391                                 clk_output_ureg=clk_UR14,
392                                 ui_output_ureg=ui_output_UR6,
393                                 ui_input_ureg=ui_input_UR10,
394                                 set_req=True,
395                                 stg_offset=0x70)
396
397         # ================================================================
398         # Write 815 in control array location index 0.
399         ii = sc.SASS_KK__MOVImm(kk_sm,
400                                 exec_pred_inv=False,
401                                 exec_pred=PT,
402                                 target_reg=staging_R2,
403                                 imm_val=815,
404                                 req=0x0,
405                                 usched_info_reg=wait15)
406         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
407         i+=1
408         ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
409                                   offset=0x0,
410                                   source_reg=staging_R2,
411                                   usched_info_reg=wait15,
412                                   req=0x0, rd=0x7,
413                                   size=32)
414         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
415         i+=1
416
417         # ================================================================
418         # Add final exit and bra instructions
419         target_cubin.create_instr(ki, i,
420                                   empty.class_name__exit,
421                                   empty.enc_vals__exit)
422         i+=1
423         target_cubin.create_instr(ki, i,
424                                   empty.class_name__bra,
425                                   empty.enc_vals__bra)
426         i+=1
427
428         return target_cubin
429
430
431     def __init__(self, sm_nr:int):
432         kk_sm = KK_SM(sm_nr, ip='127.0.0.1', port=8180, webload=True)
433         t_location = os.path.dirname(os.path.realpath(__file__))
434         temp_loc = '{0}/template_projects/template_1k_no_loop_240' \
```

```
435                         .format(t_location)
436         template = '{0}/benchmark_binaries/template_1k_no_loop_240_{1}' \
437                         .format(temp_loc, sm_nr)
438         mod_loc = '{0}/tutorials_binaries'.format(t_location)
439         exe_name = "{0}/tutorial_3_clocks_{1}".format(mod_loc, sm_nr)
440 
441         modified_file:SM_CuBin_File = self.create(kk_sm, template)
442         modified_file.to_exec(exe_name)
443         os.system('chmod +x {0}'.format(exe_name))
444 
445 
446 if __name__ == '__main__':
447     b = Kernel(86)
448     print("Finished")
449     pass
```

A.8 ISETP and Chaining

This is the full script for the Tutorial 6.10.

```
1  import os
2  from py_cubin import SM_CuBin_File
3  from kk_sm import KK_SM
4  import sass_create_86 as sc
5  
6  class Kernel:
7      def create(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
8          target_cubin = SM_CuBin_File(kk_sm.sass, template, wipe=True)
9  
10         nrc = target_cubin.reg_count()
11         nnrc = {n:100 for n,k in nrc.items()}
12         target_cubin.overwrite_reg_count(nnrc)
13 
14         # Zero and True: we need them all the time
15         RZ = kk_sm.regs.Register__RZ__255
16         PT = kk_sm.regs.Predicate__PT__7
17         # Some temp blabla
18         staging_R2 = kk_sm.regs.Register__R2__2
19 
20         # ====================================================
21         # It is a good idea to initialize registers in this
22         # way to make things more readable. The kk_sm.regs
```

```
23          # are tuples with 3 entries:
24          # UniformRegister__UR2__2 = ('UniformRegister', 'UR2', 2)
25          #  - register category
26          #  - register name
27          #  - register number (can be different than register name!)
28          wait15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15
29
30          a_UR2 = kk_sm.regs.UniformRegister__UR2__2
31          control_UR4 = kk_sm.regs.UniformRegister__UR4__4
32          ui_output_UR6 = kk_sm.regs.UniformRegister__UR6__6
33          d_output_UR8 = kk_sm.regs.UniformRegister__UR8__8
34          ui_input_UR10 = kk_sm.regs.UniformRegister__UR10__10
35          d_input_UR12 = kk_sm.regs.UniformRegister__UR12__12
36          clk_UR14 = kk_sm.regs.UniformRegister__UR14__14
37          f_output_UR16 = kk_sm.regs.UniformRegister__UR16__16
38
39          a_R4 = kk_sm.regs.Register__R4__4
40          P0 = kk_sm.regs.Predicate__P0__0
41          P1 = kk_sm.regs.Predicate__P1__1
42          P2 = kk_sm.regs.Predicate__P2__2
43          P3 = kk_sm.regs.Predicate__P3__3
44
45          op_LT = kk_sm.regs.ICmpAll__LT__1
46          op_EQ = kk_sm.regs.ICmpAll__EQ__2
47          op_LE = kk_sm.regs.ICmpAll__LE__3
48          op_GT = kk_sm.regs.ICmpAll__GT__4
49          op_NE = kk_sm.regs.ICmpAll__NE__5
50          op_GE = kk_sm.regs.ICmpAll__GE__6
51
52          fmt_U32 = kk_sm.regs.FMT__U32__0
53          fmt_S32 = kk_sm.regs.FMT__S32__1
54
55          bop_OR = kk_sm.regs.Bop__OR__1
56          bop_AND = kk_sm.regs.Bop__AND__0
57          bop_XOR = kk_sm.regs.Bop__XOR__2
58
59          # Kernel index
60          ki = 0
61          i=0
62          empty = sc.SASS_KK__Empty(kk_sm, wait15)
63          # =====================================================
64          # Add initial mov instruction
65          target_cubin.create_instr(ki, i,
66                                    empty.class_name__mov,
```

```
                                empty.enc_vals__mov)
        i+=1

        # ====================================================
        # Load unsigned int once using ULDG and another time
        # using IMAD (equivalent)
        # With ULDC we can pass the size, which makes it
        # easier to deal with larger data types, it is more
        # cumbersome if we have value types. In this instance
        # the value of a is stored into U2, and there is a lack
        # of nice instructions to do things with it. Thus, IMAD
        # is the better choice.
        # Generally, the UniformRegister category is considered
        # a 'distinct data path' and is most often used to deal
        # with addresses.
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=a_UR2,
                              m_offset=0x160,
                              usched_info_reg=wait15, size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__IMAD_RRC_RRC(kk_sm,
                                      Rd=a_R4, Ra=RZ, Rb=RZ,
                                      m_bank=0x0, m_bank_offset=0x160,
                                      usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ====================================================
        # For pointer types, using ULDC is the best option. It
        # exists on all architectures starting with SM 75 and
        # creates an uniform register with the base address.
        # There are a lot of useful instructions to do things
        # with this construct.
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=control_UR4,
                              m_offset=0x168,
                              usched_info_reg=wait15, size=64)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ULDC(kk_sm,
                              uniform_reg=ui_output_UR6,
                              m_offset=0x170,
                              usched_info_reg=wait15, size=64)
```

```
111         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
112         i+=1
113         ii = sc.SASS_KK__ULDC(kk_sm,
114                               uniform_reg=d_output_UR8,
115                               m_offset=0x178,
116                               usched_info_reg=wait15, size=64)
117         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
118         i+=1
119         ii = sc.SASS_KK__ULDC(kk_sm,
120                               uniform_reg=ui_input_UR10,
121                               m_offset=0x180,
122                               usched_info_reg=wait15, size=64)
123         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
124         i+=1
125         ii = sc.SASS_KK__ULDC(kk_sm,
126                               uniform_reg=d_input_UR12,
127                               m_offset=0x188,
128                               usched_info_reg=wait15, size=64)
129         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
130         i+=1
131         ii = sc.SASS_KK__ULDC(kk_sm,
132                               uniform_reg=clk_UR14,
133                               m_offset=0x190,
134                               usched_info_reg=wait15, size=64)
135         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
136         i+=1
137         ii = sc.SASS_KK__ULDC(kk_sm, uniform_reg=f_output_UR16,
138                               m_offset=0x198,
139                               usched_info_reg=wait15, size=64)
140         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
141         i+=1
142
143         # ======================================================================
144         # ======================================================================
145         # == Compute a couple of ISETP chainings ===============================
146         # ======================================================================
147         # ======================================================================
148         # First move a well defined value into the staging reg
149         ii = sc.SASS_KK__MOVImm(kk_sm,
150                                 exec_pred_inv=False,
151                                 exec_pred=PT,
152                                 target_reg=staging_R2,
153                                 imm_val=99,
154                                 req=0x0,
```

```
                                usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # P0: a == 1 or a == 3
        ii = sc.SASS_KK__ISETP_RsIR_RIR(kk_sm, target_pred=P0, aux_pred=PT,
                                        reg=a_R4, imm=1,
                                        comp_op=op_EQ,
                                        fmt=fmt_U32,
                                        bop_op=bop_AND,
                                        invert_Pp=False, Pp=PT,
                                        usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ISETP_RsIR_RIR(kk_sm, target_pred=P0, aux_pred=PT,
                                        reg=a_R4, imm=3,
                                        comp_op=op_EQ,
                                        fmt=fmt_U32,
                                        bop_op=bop_OR,
                                        invert_Pp=False, Pp=P0,
                                        usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # P1: a == 0 || a == 2
        ii = sc.SASS_KK__ISETP_RsIR_RIR(kk_sm, target_pred=P1, aux_pred=PT,
                                        reg=a_R4, imm=0,
                                        comp_op=op_EQ,
                                        fmt=fmt_U32,
                                        bop_op=bop_AND,
                                        invert_Pp=False, Pp=PT,
                                        usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__ISETP_RsIR_RIR(kk_sm, target_pred=P1, aux_pred=PT,
                                        reg=a_R4, imm=2,
                                        comp_op=op_EQ,
                                        fmt=fmt_U32,
                                        bop_op=bop_OR,
                                        invert_Pp=False, Pp=P1,
                                        usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

```

```
199         # depending on P0 and P1 write different values to the control reg
200         ii = sc.SASS_KK__MOVImm(kk_sm,
201                                 exec_pred_inv=False,
202                                 exec_pred=P0,
203                                 target_reg=staging_R2,
204                                 imm_val=31,
205                                 req=0x0,
206                                 usched_info_reg=wait15)
207         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
208         i+=1
209         ii = sc.SASS_KK__MOVImm(kk_sm,
210                                 exec_pred_inv=False,
211                                 exec_pred=P1,
212                                 target_reg=staging_R2,
213                                 imm_val=20,
214                                 req=0x0,
215                                 usched_info_reg=wait15)
216         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
217         i+=1
218
219         # Write the result back to memory at offset 0x8
220         ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
221                                   offset=0x8,
222                                   source_reg=staging_R2,
223                                   usched_info_reg=wait15,
224                                   req=0x0, rd=0x7,
225                                   size=32)
226         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
227         i+=1
228
229         # ================================================================
230         # Write 815 in control array location index 0.
231         ii = sc.SASS_KK__MOVImm(kk_sm,
232                                 exec_pred_inv=False,
233                                 exec_pred=PT,
234                                 target_reg=staging_R2,
235                                 imm_val=815,
236                                 req=0x0,
237                                 usched_info_reg=wait15)
238         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
239         i+=1
240         ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
241                                   offset=0x0,
242                                   source_reg=staging_R2,
```

```
243                                     usched_info_reg=wait15,
244                                     req=0x0, rd=0x7,
245                                     size=32)
246         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
247         i+=1
248 
249         # ================================================================
250         # Add final exit and bra instructions
251         target_cubin.create_instr(ki, i,
252                                     empty.class_name__exit,
253                                     empty.enc_vals__exit)
254         i+=1
255         target_cubin.create_instr(ki, i,
256                                     empty.class_name__bra,
257                                     empty.enc_vals__bra)
258         i+=1
259 
260         return target_cubin
261 
262 
263     def __init__(self, sm_nr:int):
264         kk_sm = KK_SM(sm_nr, ip='127.0.0.1', port=8180, webload=True)
265         t_location = os.path.dirname(os.path.realpath(__file__))
266         temp_loc = '{0}/template_projects/template_1k_no_loop' \
267                         .format(t_location)
268         template = '{0}/benchmark_binaries/template_1k_no_loop_{1}' \
269                         .format(temp_loc, sm_nr)
270         mod_loc = '{0}/tutorials_binaries'.format(t_location)
271         exe_name = "{0}/tutorial_4_isetp_chaining_{1}".format(mod_loc, sm_nr)
272 
273         modified_file:SM_CuBin_File = self.create(kk_sm, template)
274         modified_file.to_exec(exe_name)
275         os.system('chmod +x {0}'.format(exe_name))
276 
277 
278 if __name__ == '__main__':
279     b = Kernel(86)
280     print("Finished")
281     pass
```

A.9 Constructing a SASS Loop

This is the full script for Tutorial 6.11.

```
import os
from py_cubin import SM_CuBin_File
from kk_sm import KK_SM
import sass_create_86 as sc

class Kernel:
    def create(self, kk_sm:KK_SM, template:str) -> SM_CuBin_File:
        target_cubin = SM_CuBin_File(kk_sm.sass, template, wipe=True)

        nrc = target_cubin.reg_count()
        nnrc = {n:100 for n,k in nrc.items()}
        target_cubin.overwrite_reg_count(nnrc)

        # Zero and True: we need them all the time
        RZ = kk_sm.regs.Register__RZ__255
        PT = kk_sm.regs.Predicate__PT__7
        # Some temp blabla
        staging_R2 = kk_sm.regs.Register__R2__2

        # ====================================================
        # It is a good idea to initialize registers in this
        # way to make things more readable. The kk_sm.regs
        # are tuples with 3 entries:
        # UniformRegister__UR2__2 = ('UniformRegister', 'UR2', 2)
        #  - register category
        #  - register name
        #  - register number (can be different than register name!)
        wait15 = kk_sm.regs.USCHED_INFO__WAIT15_END_GROUP__15

        a_UR2 = kk_sm.regs.UniformRegister__UR2__2
        control_UR4 = kk_sm.regs.UniformRegister__UR4__4
        ui_output_UR6 = kk_sm.regs.UniformRegister__UR6__6
        d_output_UR8 = kk_sm.regs.UniformRegister__UR8__8
        ui_input_UR10 = kk_sm.regs.UniformRegister__UR10__10
        d_input_UR12 = kk_sm.regs.UniformRegister__UR12__12
        clk_UR14 = kk_sm.regs.UniformRegister__UR14__14
        f_output_UR16 = kk_sm.regs.UniformRegister__UR16__16

        a_R4 = kk_sm.regs.Register__R4__4
        a_iter_R6 = kk_sm.regs.Register__R6__6

```

```
42          # Kernel index
43          ki = 0
44          i=0
45          empty = sc.SASS_KK__Empty(kk_sm, wait15)
46          # ====================================================
47          # Add initial mov instruction
48          target_cubin.create_instr(ki, i,
49                                    empty.class_name__mov,
50                                    empty.enc_vals__mov)
51          i+=1
52
53          # ====================================================
54          # Load unsigned int once using ULDG and another time
55          # using IMAD (equivalent)
56          # With ULDC we can pass the size, which makes it
57          # easier to deal with larger data types, it is more
58          # cumbersome if we have value types. In this instance
59          # the value of a is stored into U2, and there is a lack
60          # of nice instructions to do things with it. Thus, IMAD
61          # is the better choice.
62          # Generally, the UniformRegister category is considered
63          # a 'distinct data path' and is most often used to deal
64          # with addresses.
65          ii = sc.SASS_KK__ULDC(kk_sm,
66                                uniform_reg=a_UR2,
67                                m_offset=0x160,
68                                usched_info_reg=wait15, size=32)
69          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
70          i+=1
71          ii = sc.SASS_KK__IMAD_RRC_RRC(kk_sm,
72                                        Rd=a_R4, Ra=RZ, Rb=RZ,
73                                        m_bank=0x0, m_bank_offset=0x160,
74                                        usched_info_reg=wait15)
75          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
76          i+=1
77
78          # ====================================================
79          # For pointer types, using ULDC is the best option. It
80          # exists on all architectures starting with SM 75 and
81          # creates an uniform register with the base address.
82          # There are a lot of useful instructions to do things
83          # with this construct.
84          ii = sc.SASS_KK__ULDC(kk_sm,
85                                uniform_reg=control_UR4,
```

```
86                              m_offset=0x168,
87                              usched_info_reg=wait15, size=64)
88          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
89          i+=1
90          ii = sc.SASS_KK__ULDC(kk_sm,
91                              uniform_reg=ui_output_UR6,
92                              m_offset=0x170,
93                              usched_info_reg=wait15, size=64)
94          target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
95          i+=1
96          ii = sc.SASS_KK__ULDC(kk_sm,
97                              uniform_reg=d_output_UR8,
98                              m_offset=0x178,
99                              usched_info_reg=wait15, size=64)
100         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
101         i+=1
102         ii = sc.SASS_KK__ULDC(kk_sm,
103                             uniform_reg=ui_input_UR10,
104                             m_offset=0x180,
105                             usched_info_reg=wait15, size=64)
106         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
107         i+=1
108         ii = sc.SASS_KK__ULDC(kk_sm,
109                             uniform_reg=d_input_UR12,
110                             m_offset=0x188,
111                             usched_info_reg=wait15, size=64)
112         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
113         i+=1
114         ii = sc.SASS_KK__ULDC(kk_sm,
115                             uniform_reg=clk_UR14,
116                             m_offset=0x190,
117                             usched_info_reg=wait15, size=64)
118         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
119         i+=1
120         ii = sc.SASS_KK__ULDC(kk_sm, uniform_reg=f_output_UR16,
121                             m_offset=0x198,
122                             usched_info_reg=wait15, size=64)
123         target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
124         i+=1
125
126         # ======================================================================
127         # ======================================================================
128         # == Compute a couple of things in a loop==============================
129         # ======================================================================
```

```
130         # ================================================================
131
132         # Init a_iter_R6 to Zero
133         ii = sc.SASS_KK__IADD3_NOIMM_RRR_RRR(kk_sm,
134                                              a_iter_R6,
135                                              negate_Ra=False, src_Ra=RZ,
136                                              negate_Rb=False, src_Rb=RZ,
137                                              negate_Rc=False, src_Rc=RZ,
138                                              usched_info_reg=wait15)
139         target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
140         i+=1
141
142         # This is the starting point of the loop. Use
143         # the instruction index i to calculate the jump size
144         i_start = i
145         # Increment the loop variable by 1: use Ra = Ra + 1
146         ii = sc.SASS_KK__IADD3_IMM_RsIR_RIR(kk_sm,
147                                             a_iter_R6,
148                                             negate_Ra=False, Ra=a_iter_R6,
149                                             src_imm=1,
150                                             negate_Rc=False, Rc=RZ,
151                                             usched_info_reg=wait15)
152         target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
153         i+=1
154
155         branch_pred = kk_sm.regs.Predicate__P0__0
156         ii = sc.SASS_KK__isetp__RRR_RRR_noEX(kk_sm,
157                                 pred_invert=False, pred=PT,
158                                 Pu=branch_pred,
159                                 Ra=a_R4,
160                                 icmp=kk_sm.regs.ICmpAll__GT__4,
161                                 Rb=a_iter_R6,
162                                 fmt=kk_sm.regs.FMT__S32__1,
163                                 usched_info_reg=wait15)
164         target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
165         i+=1
166
167         # Use the current instruction index to calculate the jump
168         # size!
169         i_end = i
170         ii = sc.SASS_KK__BRA(kk_sm,
171                              pred_invert=False, pred=branch_pred,
172                              Pp_invert=False,
173                              Pp=PT,
```

```
                            imm_val=int(-16*(i_end - i_start + 1)),
                            usched_info_reg=wait15)
        target_cubin.create_instr(0, i, ii.class_name, ii.enc_vals)
        i+=1

        # Write the iteration variable into second place of the
        # control register
        ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
                                  offset=0x8,
                                  source_reg=a_iter_R6,
                                  usched_info_reg=wait15,
                                  req=0x0, rd=0x7,
                                  size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================================
        # Write 815 in control array location index 0.
        ii = sc.SASS_KK__MOVImm(kk_sm,
                                exec_pred_inv=False,
                                exec_pred=PT,
                                target_reg=staging_R2,
                                imm_val=815,
                                req=0x0,
                                usched_info_reg=wait15)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1
        ii = sc.SASS_KK__STG_RaRZ(kk_sm, uniform_reg=control_UR4,
                                  offset=0x0,
                                  source_reg=staging_R2,
                                  usched_info_reg=wait15,
                                  req=0x0, rd=0x7,
                                  size=32)
        target_cubin.create_instr(ki, i, ii.class_name, ii.enc_vals)
        i+=1

        # ================================================================
        # Add final exit and bra instructions
        target_cubin.create_instr(ki, i,
                                  empty.class_name__exit,
                                  empty.enc_vals__exit)
        i+=1
        target_cubin.create_instr(ki, i,
                                  empty.class_name__bra,
```

```
                                    empty.enc_vals__bra)
        i+=1

        return target_cubin


    def __init__(self, sm_nr:int):
        kk_sm = KK_SM(sm_nr, ip='127.0.0.1', port=8180, webload=True)
        t_location = os.path.dirname(os.path.realpath(__file__))
        temp_loc = '{0}/template_projects/template_1k_no_loop' \
                        .format(t_location)
        template = '{0}/benchmark_binaries/template_1k_no_loop_{1}' \
                        .format(temp_loc, sm_nr)
        mod_loc = '{0}/tutorials_binaries'.format(t_location)
        exe_name = "{0}/tutorial_5_loop_{1}".format(mod_loc, sm_nr)

        modified_file:SM_CuBin_File = self.create(kk_sm, template)
        modified_file.to_exec(exe_name)
        os.system('chmod +x {0}'.format(exe_name))


if __name__ == '__main__':
    b = Kernel(86)
    print("Finished")
    pass
```