CUDA SASS汇编器实现笔记（2）-自动指令编码

https://zhuanlan.zhihu.com/p/296908633

None Fri Jun, 04 04:30

今天分享通过简单的线性代数方程组来求解SASS指令编码的自动化算法。今天这部分主要讲相对通用的部分，一些需要特殊处理的部分将会留到下期再讨论。

指令编码的基本逻辑

历史上存在过的指令集很多，支持的功能和形式也多种多样，但指令集本身的编码方式还是有很多的共通之处。CISC如x86，长度不定，编码格式复杂，不同指令的编码整齐度也比较差，所以解码会比较费事。RISC指令集则一般定长，格式整齐，一般可分为一些格式相近的组，每个组的操作数位置相同，支持的modifier也相似，所以解码会比较简单。当然，RISC能简化的另一个重要原因是非load-store指令通常不支持memory操作数，省去了x86那些复杂寻址模式的困扰。

由于NV没有提供SASS的官方编码文档，我节选了RISC-V中的基础编码格式来简单的阐述一下其中的逻辑（选自RISC-V官方Spec，Vol I: Unprivileged ISA, 版本V20191213, P16）：

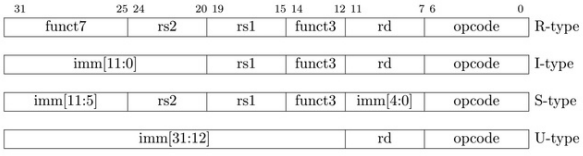

Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

指令编码分为几个类（R/I/S/U）。每个类分为多个不相交的域，比如R型有opcode域，目标寄存器域rd，源寄存器域rs1、rs2，功能指定域funct7、funct3等等。不同类指令的域划分方式是不一样的，这通常意味着解码的时候不同类要走不同分支。注意不同类未必是不同的功能单元，比如整数add功能单元可以接受R型的两寄存器输入（ADD），也可以接受I型的寄存器+立即数输入（ADDI），两者类型不同，但完全可能用同一个功能单元来执行。这里和NV的一个区别是指令的操作不一定都由opcode指定，而是与funct7、funct3共同来指定。编码的方式会决定很多操作数的取值范围（或者说是表达能力），比如指定寄存器的域为5bit，那最多编码32个寄存器。再比如跳转指令一般带立即数操作数（类似短跳），如果目标地址超出立即数范围，那就不能短跳，可能需要直接操作PC（相当于长跳）。一些load/store指令的立即数offset范围也取决于编码位数。诸如此类，等等。

CUDA SASS的编码逻辑与前述RISC-V大致相似，只是具体的分类形式以及各个域的含义不同。SASS在Fermi及以前好像是不定长的，但到Kepler以后就定长了，另外还加入了control codes。control codes在Kepler、Maxwell/Pascal、Volta/Turing/Ampere分别有不同的形式，但是与原本的汇编指令部分的编码基本是独立的。具体格式之前的专栏文章已经讲过，不再赘述。这里我们只讨论SASS中描述指令操作部分的编码。简单起见，这里只讨论Turing的格式，其他微架构具体编码上有差别，但指导思想是一样的。

单类指令的编码方法

正常来讲，要做一个汇编器，要先获得各种指令类型的分类方式和各个域的含义。但是NV并没有提供这个（会提供这个估计就会放出汇编器了）。所以这其中的对应关系需要我们自己搜集。这其中有不少各显神通的探测方法。上期已经讲过，我的需求很简单，就是能完成汇编功能就行，不想花太多力气研究具体编码和格式。所以我这里采用了一种简明直接的方式。

NV提供了反汇编的工具 `cuobjdump`，可以看到指令汇编的文本和指令编码之间的对应关系，但并不会具体给出编码的逻辑。例如，某个dump的sass指令为：

```
/*1190*/    @P0 FADD.FTZ R13, -R14, -RZ ;    /* 0x800000ff0e0d0221 */
                                             /* 0x000fc80000010100 */
```

其中前面的 `/*1190*/` 是指令地址（相对于Kernel起始位置），`@P0` 是谓词（Predicate），表示 `P0` 为 `True` 时这条指令才起作用（**注**：我一直以为Predicate不影响指令发射，只影响写回。不过我最近看到Ampere有些memory指令加了 `@!PT` 后发射stall的cycle数变成了1，也许有更复杂的机制在，等我有空研究下）。`FADD` 就是通常说的操作码（Opcode）。`.FTZ` 是Opcode `FADD` 的一个modifier，表示对subnormal做Flush-To-Zero。`R13`，`-R14`，`-RZ` 是 `FADD` 的操作数（operand），其中 `R13` 为目的操作数，其他两个是源操作数。所以这个指令的操作逻辑为：`R13 = (-R14) + (-RZ)`。`RZ` 是SASS中一个输出恒0的寄存器（做输入永远为0，做输出则丢弃）。另外 `R14` 和 `RZ` 前面都带有一个负号 `-`，这是我们之前提到的operand modifier，浮点操作数还可以取绝对值 `|*|`，或者同时取负和取绝对值。具体各种指令分类和解释可以参考之前的介绍文章：

cloudcore：CUDA微架构与指令集（3）-SASS指令集分类zhuanlan.zhihu.com

要编码一个指令，就需要知道它属于哪个大类（相当于域的bit划分方式），以及每个域的编码方式。比如第一个源操作数 R14 要放在哪，前面加负号需要设置哪个bit，第二个源操作数 RZ 又放在哪，加负号又是哪个bit，等等。这里，我们先来研究同样opcode且同类operand输入时的编码逻辑。

以前面提到 FADD 指令为例。显然，汇编文本中的每个token都有自己一个编码（如 FADD ， RZ ， R13 等）。而最终的指令编码就是各部分编码的和：

```
c(*) = c('@P0') + c('FADD.FTZ') + c('0_R13') + c('1_-R14') + c('2_-RZ').
```

其中操作数部分加了序号前缀，因为同样是 R13 ，做第一个还是第二个操作数显然会对应不同的编码。然后我们还可以接着把所有的modifier拆分出来，注意操作数的modifier也要加相应序号。另外， RZ 被转成了 R255 （可以认为二者就是alias的关系）。

```
c(*) = c('@P0') + c('FADD') + c('FTZ') + c('0_R13') + c('1_Neg')
    + c('1_R14') + c('2_Neg') + c('2_R255').
```

这样，指令编码的搜集过程就简化为求解各个域对应的编码的过程，还有各个不同指令的格式分类过程。

之前多数第三方汇编器在编码这一块都投入了大量的精力，通过人力分析整理出了一大堆复杂的编码逻辑。我比较懒，这个方式对我来讲太痛苦了。人肉总结这些规则还是太烦了，我又不关心具体每个指令是怎么编码的，我只想复现就行。关键是这个方法还存在很多不方便解决的问题，特别是输入不充分的时候。比如 PLOP3 指令：

```
PLOP3.LUT P0, PT, PT, PT, PT, 0x8, 0x0 ;
PLOP3.LUT P2, PT, P2, P3, PT, 0xa8, 0x0 ;
PLOP3.LUT P1, PT, P1, P2, PT, 0x80, 0x0 ;
```

里面经常几个 PT 同时出现，容易造成困惑，很难直接得出哪个 PT 对应哪几个bit。这就需要需要大量的高质量输入去做人脑分析（有些可以从标准库中dump，有些就需要自己写各种case去触发对应的指令）。这些工作掺在大量的指令格式的整理工作中，导致工作量巨大，大量精力被浪费在这些价值不高的重复工作上。我不太接受这种方式，所以我采用了一种更自动化的方法来做指令编码。这里我们暂且先不管不同类型的指令编码的差别，只管同opcode和同样operand类型输入的编码。

首先，我把输入的汇编文本拆分成多个token（比如`FADD`，`R13`等），每个token对应一个域。我把各个域的编码分为两部分：`Code = Value*Weight`，`Value`是与上下文无关的文本本身的值，而`Weight`则取决于这个域的位置（多数时候可以看成域对应bit中最低位的值，例如第1位就是1，第9位就是256）。同样的`R13`出现在不同操作数位置时，`Value`是一样的，只是`Weight`不同。如果说我们可以通过简单的规则得到每个指令的`Value`的序列，然后又能收集到足够多不同`Value`组合（比如dump所有的标准库），而最终的编码又都是已知的，这不就转化成了一个简单的线性代数方程组的求解，那不就可以算出对应的`Weight`了吗？

那每个指令的`Value`怎么得到呢？我们先把modifier放一边，先来说操作数的`Value`。对于Turing架构，操作数的`Value`的取值方式有下面几种（为了区分类型，每种操作数有一个名字label来表示这个值的类型，后面会用到这个label去做分类）：

1. **Indexed type**: 所有的索引类型都是一个前缀带一个数字。比如register `R#`，predicate `P#`，uniform register/predicate `UR#/UP#`, barrier `B#`,以及 scoreboard `SB#`。它的值就是后面的数字，名字label就是前面的前缀。比如`R13`的值就是13，label是`R`，`UP2`的值就是2，label是`UP`。这里也包括一些alias的情况，比如`RZ=R255`，`PT=P7`，`URZ=UR63`等等，相互之间完全等价。
2. **Integer Immediate**：整数立即数，比如`0x0`，`0xff`这种。它的值是它本身代表的值，label为`II`。那这就有一个问题，负数怎么办呢？简单，把负号拆出来作为一个modifier，值仍然是整数本身的值，但是多了一个负数的offset，正好让这个负号的modifier来承担。
3. **Float Immediate**: 浮点立即数与整数类似，但是浮点数的编码会稍微复杂些。这个与具体架构有关，有很多问题暂时无法放进通用框架内处理，下期讲特殊处理的时候再细说。浮点数的值就是其相应的二进制表示（可能有些bit会截掉，有的符号位也会转移），label为`FI`。注意在cuobjdump的sass中，整数立即数必须以`0x`开头，如果写成`1`、`42`这种则是浮点立即数。这个惯例也被继承下来。
4. **Address**: 内存地址类型，一般都在中括号内：`[*]`。里面的表达式不一定是单纯的立即数如`[0x18]`，可能有寄存器值如`[R#+0x####]`，甚至更复杂的如这种`[R10.X8+UR4+0x10]`。所以它的`Value`会是一个数组，label是`A`（address）加上内部子串label的组合。例如`[R10.X8+UR4+0x10]`的值为`[10, 4, 0x10]`，label为`ARURII`。那其中的`.X8`怎么办呢？这个会被剥离出来作为modifier，另有任用~
5. **Constant memory**: Constant memory的形式为`c[0x##][Address]`，第一个中括号是bank。后面则是bank内的相应地址，它的值为bank的值与后面Address值的组合，label是`c`再加上后面Address的label。比如`c[0x2][R4+0x8000]`的值为`[2, 4, 0x8000]`，label为`cARII`。
6. **Scoreboard Set**：这个只是单纯支持DEPBAR指令的特殊操作数，类似`DEPBAR.LE SB1, 0x0, {4,3,2} ;`中的`{4,3,2}`。这个下次讲特殊处理的时候再细讲。
7. **Label**: 这类就专门放其他不属于上面那些类的部分，比如`SR_TID.X`，`SR_LEMASK`，`3D`，`ARRAY_2D`这种。这个处理起来自由度很大，也留到下期再讲。

除了operand外，其他部分如predicate也有个值，这个现在都是4bit，所以`@P0`、`@!P3`这种会直接转成对应的二进制表示，只是不做操作数的predicate不出现在label里。把所有出现的modifier都赋值为1，相当于把编码值都交给`Weight`来承担（当然，操作数位置等仍然要区分）。每个指令可能会出现不同的modifier，则相应出现的值为1，不出现则值为0。Opcode的部分我们先让`FADD`来承担，相当于让其值为1，权为相应的opcode的编码。这样，前述指令编码就可以表示为：

```
c(*) = c('@P0') + c('FADD') + c('FTZ') + c('0_R13') + c('1_Neg') + c('1_R14') + c('2_Neg') + c('2_R255').
     = [0] * W('@P#') + [1] * W('FADD') + [1] * W['FTZ'] + [13]*W['0_R#'] + [1] * W['1_Neg']
       + [14] * W['1_R#'] + [1] * W['2_Neg'] + [255] * W['2_R#']
     = [0, 1, 1, 13, 1, 14, 1, 255]
       * [w('@P#'), w('FADD'), w('FTZ'), w('0_R#'), w('1_Neg'), w('1_R#'), w('2_Neg'), w('2_R#')]^T
     = v * w^T
```

这样，通过收集不同的`FADD`指令（注意所有操作数类型必须匹配，这样权重含义才一致），就可以得到一系列的`v`值，而`c`值已知。把所有`v`的值作为行向量组成系数矩阵`A`，原编码值组成右端项`b`，这样求解`A*w^T = b`就可以得到`w`的值，这样权重就可以得到了。给出一个新的待编码的指令，用同样的方法很容易算出其值序列`v`，然后与求出的权重`w`点乘即可算出指令的编码值。

不同类型编码的处理

那前面所说的方程组能解出所有指令对应的权重吗？理论上说，你可以永远把指令当成一个整体的label，把值设为1，把完整的指令文本作为weight。这样上面的逻辑仍然成立。但是这样，你只能汇编出所有你精确见过的指令，任何一丁点修改都不行。这显然适用性很差，而且编码字典太占空间了，实现上也不现实。我们需要找到一个方法，既能够最大化通用性（保持规则越少越好，减少人工干预的成分），又能最小化输入所需的指令数目，权重之间最大程度复用。

对于一个指令而言，最简单的莫过于一个predicate（隐式@PT或显式@P0这种）加opcode，比如`@P1 EXIT`，`YIELD`。对于所有带操作数的指令，value长度最少要包括predicate、opcode再加上每个operand的值。其他的modifier则属于可选部分（有就是值为1，没有就值为0）。因此，我们把一个opcode下同样类型输入的指令分为一类，相当于这一类的指令的值序列才会放到同一个系数矩阵`A`里。不同的opcode或是同样的opcode不同operand类型的矩阵是分开的，域的含义不一样，权重`w`自然也会不一样。这种方式既可以最大限度的容纳操作数和modifier内容的变化，又能兼顾指令大类或操作数类型不同导致的格式差异。

我们给每个这样的类起个名字（我这里称为**Key**），用来区分各个权重所属的类型。Key的具体组成方式就是把opcode和所有operand的类型的label用下划线连接起来。比如：

```
LDC_R_cAR               :     LDC.64 R6, c[0x2][R2] ;
LDS_R_ARI               :     LDS.U R9, [R0.X4+0x1000] ;
IMAD_R_R_II_R           :     IMAD.WIDE R2, R5, 0x8, R2 ;
IADD3_R_P_R_II_R        :     IADD3 R4, P2, R4, 0x4, RZ ;
FFMA_R_R_R_FI           : @P1 FFMA R3, R12, R9, -0.0013887860113754868507 ;
FFMA_R_R_R_R            :     FFMA R3, R12, R3, R4 ;
TEX_R_R_R_R_II_II_2D_II :     TEX.SCR.LL RZ, R6, R4, R11, 0x0, 0x5e, 2D, 0x1 ;
EXIT                    :     EXIT ;
```

这样，只有具有同样Key的指令才会共享权重序列。当然，这显然是有些浪费的。有很多同类型的指令会支持同样的modifier和operand类型，只是opcode不同，但是我们把这些都分开了。这么做，首先是基于我们只想复现汇编码，不关心具体编码过程，汇编器性能也关系不大。其次，就是不想花精力自己去做各种指令的分类，这对我们价值不大，能自动就自动了，浪费点空间关系不大。再次，这样可以把通用性做到很高的程度。将来即使是SASS的编码方式和分类有变化，它仍然可以很好的处理。实际上这套流程不仅适用于Turing，对于其他的SM架构几乎也可以不需要过多的人工干预。当然，需要特殊处理的地方也有，但都是个别的，工作量还可以接受。

非完备编码的处理

如果能得到完备的输入，前述线性代数方程组 `A*w^T=b` 就是可逆的，我们可以直接把权重序列 `w` 解出来。在大部分情况下，`w` 中的值就可以完备的对应到各个域的编码方式上（特殊情况也是有的，下期再议）。可惜的是，如前所述的 `PLOP3` 的那种总是看到两个PT结伴而行的情况，这时系数矩阵 `A` 是亏秩的（相当于有两列值总是一样，必定亏秩），无法解出 `w` 的每个值！然而，有趣的是，我们其实并不总是需要解出每个值！其实只要是需要编码的指令系数值能够被现有的系数矩阵 `V` 的行线性表示，那这个指令就能够进行确定性的编码！

这里我做了一点小的数学变换，检查是否能线性表示比较耗时。更直接的方法是先求出 `A` 的零空间nullspace作为行向量组成的矩阵：`N=null(A)`，如果待编码的系数向量 `v` 位于其零空间（也即有 `N*v^T=0`），那 `v` 就可以被 `A` 的行向量线性表示，也即该指令就可以被编码。

例如有两个这样的输入：

```
FADD R10, R1, R2;  /* c0 */
FADD R11, R1, R2;  /* c1 */
```

其值系数矩阵为[1, 10, 1, 2; 1, 11, 1, 2]，显然是亏秩的。但是如果想编码：

则发现它可以被二者线性表示，也就是 `c1-c0` 其实可以标定第一个操作数值R10->R11中 `+1` 的变化，那R11->R12再加一次也一样。那最终编码就可以确定为 `c1 + (c1-c0)`。

这也为系数矩阵`A`的构建提供了一个重要思路：我们其实没必要保存所有收集到的系数，因为大部分的输入指令都可能是冗余的，只需要保存那些提供新信息的指令系数值（相当于当前不能被编码的才会用于更新`A`）。所以，我们可以迭代的输入一系列的指令，首先检查有没有出现新的没见过的modifier，因为有新modifier相当于原`A`的对应列原先都为0，肯定有新信息。如果没有新modifier，再看`v`是否在当前`A`的零空间，如果不在，说明它不能被当前系数编码，则把`v`作为做为新的一行加入`A`中，并更新对应的零空间即可。如此循环。

这里还有另一个有趣的点。在系数矩阵`A`亏秩不可逆的情况下，要求解`w`一般是不行的，相当于`w`有无穷多组解。但是，只要保证待编码指令的系数值`v`在`A`的零空间内，哪个解算出来的`c=v*w^T`都是一样的！相当于`w`可以灵活选择的那些分量其实是与`v`垂直的，求内积后为0，不会对最后的编码有任何贡献！所以只要找到任意一个`w`的解就行。这样编码的时候就可以不用去凑`A`的线性组合，直接算内积就可以了，计算量上还是可以省不少。

这个方法另一个重大的好处是可以把大部分编码的搜集工作直接推到用户端去。也就是说，假如用户用CUDA C或PTX生成了一个当前不支持的指令，则在不涉及特殊处理的情况下，用户只需要自己运行一下系数矩阵的更新过程，就可以自动生成能编码该指令的规则，而不用去改汇编器本身。实际上，在我现在的处理中，只需要选择对应的系数矩阵文件，甚至都可以实现多SM版本的支持。这在需要做多个版本的汇编的时候会省力许多。

注：我最近才看到KeplerAs采用了非常逆向的bit翻转的方式来穷举每个域的编码位置。我这里没有这么做，我也不确定这种方式产生的输入是不是一定正确的。我这里的所有输入要么就是NV标准库的dump，要么是CUDA C或PTX编译生成后dump的代码。应该说是来源很正义了，如果有bug完全可以找官方解决~ 当然，如果对bit翻转的方式很信任，它也可以作为我这里的输入，对完备性应该会有帮助。不过我感觉标准库都没用到，且用很多C或PTX都无法触发的指令形式，估计要么应用有限，要么有别的实现路子，所以也不用太纠结了吧……

小结

这篇文章写了这么长，但其实python代码实现很短，关键代码除掉注释估计就三五百行而已，算得上是很简洁了。当然，这也得益于矩阵部分用了sympy库，而且py自带大整数支持。

其实这里整个流程的通用性很强。如有是有其他的指令集能提供反汇编而不提供汇编的，基本上也可以这么弄。只是具体的value的处理方式上有些变化，其他流程都是一样的。

不过，我们并不是SASS汇编规则的制定者，要想完备的支持它的编码，光有这个通用规则并不能100%的支持所有指令。所以下期我会讲一讲需要特殊处理的指令，以及不同SM版本之间需要区别处理的地方。这其中也会碰到不少令我倍感迷惑的指令，下期再分享一下~ 也期待各位有了解的到时能答疑解惑~