CUDA微架构与指令集（2）-SASS指令集概述

https://zhuanlan.zhihu.com/p/161624982

None Mon May, 24 03:20

今天来聊一聊CUDA的SASS指令集。官方没有看到SASS的全称，有人说是Streaming ASSembly。SASS是CUDA中对应GPU的机器码的硬件指令集。CUDA中还有另一个更上层的虚拟指令集PTX（**P**arallel **T**hread e**X**ecution）。我大概总结了两者的一些区别和联系：

指令集性质：SASS指令集与SM架构有直接对应关系，一旦硬件架构设计完成就不再改变。注意不一定是一一对应，因为一些架构的改变可能仅表现为某些指令的性能变化，但SASS指令集本身（包括编码和功能）并没有变化。典型的例子是Maxwell和Pascal两者的SASS几乎是完全一样的，当然其实二者的底层硬件架构可能也是高度雷同，但毕竟是两个版本。PTX与硬件架构只有比较弱的耦合关系，它本质上是从SASS上抽象出来的一种更上层的软件编程模型，介于CUDA C/C++和SASS之间。PTX也有版本，但只与PTX本身所支持的功能有关，更类似于C99，C++11这种语言版本，与硬件架构未必有对应关系。PTX是一种抽象语言，理论上说，每个版本都可以支持任意版本的SASS指令集，而且可以通过软件升级维护进行扩展和调整。但是由于某些PTX功能与硬件SASS指令的强相关性，导致某些特定架构上的实现可能会受到限制，甚至完全不支持。所以PTX除了自身的软件版本以外，也有 `.target` 语句来指定目标架构的sm版本（称为virtual architecture），相当于表示当前PTX文件只能使用某个sm支持的feature。

兼容性：CUDA C/C++程序编译完成后，一般NVCC会同时生成PTX和SASS代码，当然用户也可以指定只生成其中一种。SASS前面已经说过，是机器码的硬件指令集，编译的SM版本与当前GPU的SM版本不对应的话是不能运行的。但PTX可以被driver中的jit编译器编译成与当前GPU对应的SASS代码。这样就实现了代码的可移植性和向后兼容。前提是driver的版本要够新，能支持当前的GPU，同时PTX文件的版本要支持那个架构。所以买了一块最新的卡，以前的程序如果内嵌了PTX还是能跑的，只是需要更新一下驱动。从功能上讲，PTX是向后兼容的。但SASS不一定，有可能前一代架构的指令由于某种原因被废弃了。比如说32bit整数的乘法，在Kepler中有 `IMAD` 指令，但Maxwell和Pascal里一般都用三个16bit整数乘法指令 `XMAD` 来实现，在Turing中又用回了 `IMAD` 指令。Maxwell和Pascal也许还有 `IMAD` 指令（也许是性能不好，不确定），但 `XMAD` 应该是前后几代都没有用了。所以SASS的功能是可以随着需求而增删的。只要PTX提供了足够的向后兼容性（也就是这个功能可以由其他指令完成），那整个程序就可以实现向后兼容。**注意**：PTX可以向后兼容，那能不能兼容更早的架构呢？我测试过几次，感觉是不行，一般会报209错误 `cudaErrorNoKernelImageForDevice`，即使实际运行需要的feature在当前显卡上是能支持的。所以一般建议编译成PTX的时候，gencode的版本低一些比较好，现在CUDA 11好像最低支持到 `compute_30`（对应sm30），意味着更早的芯片就不能跑了。

当然，PTX的兼容性也有一些成本。尽管PTX是比较接近汇编的语言，其JIT编译还是会消耗一些时间。如果Kernel运行时间本来很短，那这个成本就会相对更高。不过driver会对之前的编译结果做一些cache，所以重复运行的overhead并不大。但是这个cache一般重启后也会消失，所以下次用还是要重新编译。而如果编译时在FatBinary中已经有对应的SASS版本，就不再有jit的这个overhead了。

官方支持：PTX是NVIDIA官方支持的最底层，有相关的文档（见Parallel Thread Execution ISA）和完善的工具链（NVCC，cuobjdump，PTXAS等等），也可以在driver api中load，甚至支持cuda C中inline PTX assembly。而SASS这层只有非常简略的介绍SASS Instruction Set Reference，虽然其中也提供了一些工具如nvdisasm和cuobjdump做一些分析，但也非常局限。Debug上两者倒是差别不大，NSight功能比较完善了，现在应该是可以支持cuda C/PTX/SASS三个层级的debug。

对于大多数用户来讲，如果需要基于PTX开发，是有完整的官方文档和工具链的，而且在官方论坛上也可以得到一定的支持。但是要基于SASS开发则基本需要白手起家，因为连基本的官方汇编器都没有。因为官方只提供了简单的反汇编器（`nvdisasm`和`cuobjdump`），control codes之类也不会显示。不过有一些第三方的汇编器，如`asfermi`，`maxas`等等，但因为是非官方版本，功能有限且容易出错，仅做研究用，产品代码一般并不推荐。

PTX的兼容性是NVIDIA能够进行快速架构迭代的重要手段。从某种意义上讲，功能是刚性需求，性能是弹性需求。所以兼容性都是保证功能可以延续，但性能则可以根据需要调整。SASS可以根据硬件实现和市场需求来选择最合适的指令集，而PTX则在它基础上构建相对稳定的feature列表。假如某个feature价值很高，SASS可以专门为他设计一条高性能指令，即使实现这个指令开销很大也值得。但如果后来这个功能重要性下降，那就可以把这条硬件指令删掉，用其他指令来凑成这个功能，从而把这个硬件指令的开销省下来。也有的时候是找到了某个功能更好的实现方式，从而替换了原来的指令，而两者的用法可以相同，也可以不同，但在PTX层是可以完全一致的。同时，编译器的发展也为SASS的发展演化提供了很大的帮助。从Kepler开始，NVIDIA就可以将一些控制逻辑交给编译器来做，有人称为control codes，将来会细讲。在Kepler中，每条指令是64bit，每8条指令中有一条会专门编码control codes。到了Maxwell和Pascal，则是每4条指令中有一条control codes。到了Volta和Turing架构，每条指令长度由64bit变成了128bit，这样每条指令都能够编码control codes。这些改变对大多数的用户程序几乎都是透明的，这就得益于PTX所提供的兼容性。如果说SASS要像X86那样必须完全支持先前版本的所有二进制程序，那势必背上沉重的历史包袱，功能更新和迭代速度上显然就会受到极大限制了。

我们可以写个简单的CUDA C程序（存为`cudatest.cu`）来看看具体的代码生成：

```
#include 'cuda_runtime.h'

__global__ void func(int c, int* a)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    a[idx] *= c;
}

int main()
{
    return 0;
}
```

然后用nvcc来编译。我这里用的cuda 10.2 on win8.1，我这里已经把nvcc所在目录加到了环境变量。注意环境变量中还要有相应的C/C++编译器，比如VS的cl等，否则nvcc会报错。编译命令如下：

```
nvcc cudatest.cu -o cudatest -gencode=arch=compute_30,code=\'sm_30,compute_30\' -gencode=arch=compute_52,code=\'sm_52,compute_52\' -gencode=arch=compute_75,code=\'sm_75,compute_75\'
```

这里写了很多`-gencode=*`，用来控制具体要生成哪些PTX和SASS代码。`arch=compute_30`表示基于`compute_30`的**virtual GPU architecture**，但它只是我们前面提到的控制使用feature的子集，并不控制是否生成具体PTX代码。后面的`code=\'sm_30,compute_30\'`才表示代码生成列表。其中`sm_30`表示基于`sm_30`的架构生成相应SASS代码，`compute_30`表示基于`compute_30`的虚拟架构生成相应PTX代码，这个必须与前面`arch=*`一致。前面也提到了PTX有向后兼容性，所以这里也可以基于`compute_30`生成多个架构的SASS代码，比如`code=\'sm_30,sm_50,sm_75\'`等等，注意这里不写`compute_30`表示不再生成对应PTX代码了，也就是说其他sm版本就跑不了这个程序了。

多个`-gencode=*`可以支持多个虚拟架构列表，而每个都可以按这个逻辑来控制代码生成。所有的代码生成后会被打包成FatBinary，内嵌在程序中供调用。程序运行时driver会去判断是否有编译好的对应架构的SASS版本，如果没有就从可选的PTX中JIT编译一个（印象中是挑可用的最高版本）。如果是没有合适的PTX文件，比如它最低支持的是`compute_50`，但是我只有sm_35的卡，那运行程序就会返回209错误`cudaErrorNoKernelImageForDevice`。**注意**：CUDA里很多错误是不造成CPU运行中断或抛出异常的，需要手动check返回值。运行kernel没有返回值，就只好用cudaGetLastError来检查错误，当然这里要记得要先做`cudaDeviceSynchronize()`。

NVCC支持的选项很多，有兴趣的同学可以自己去看文档。在VS里控制代码生成比较简单，只需要把项目属性中CUDA C/C++的device下的CodeGeneration改掉就行，多个就用分号隔开。比如上面的就可以直接写`compute_30,sm_30;compute_52,sm_52;compute_75,sm_75`。如果只是单个cu文件要改，那就在那个cu文件对应的属性中改。

编译完成后，我们可以把生成的SASS和PTX代码dump出来看一下：

```
cuobjdump -ptx cudatest.exe > cudatest.ptx
cuobjdump -sass cudatest.exe > cudatest.sass
```

其中PTX代码节选如下。因为这里没有用到太多版本相关的feature，所以对应compute_30/compute_52/compute_75的三个版本基本就没啥变化，只是`target`不一样而已，所以这里我只列了一个。最前面的`.version 6.5`表示PTX ISA的版本，具体版本变化可以看PTX的官方文档。

```
.version 6.5
.target sm_30
.address_size 64

.visible .entry _Z4funciPi(
.param .u32 _Z4funciPi_param_0,
.param .u64 _Z4funciPi_param_1
)
{
.reg .b32 %r<8>;
.reg .b64 %rd<5>;

ld.param.u32 %r1, [_Z4funciPi_param_0];
ld.param.u64 %rd1, [_Z4funciPi_param_1];
cvta.to.global.u64 %rd2, %rd1;

mov.u32 %r2, %tid.x;
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %ntid.x;

mad.lo.s32 %r5, %r4, %r3, %r2;
mul.wide.s32 %rd3, %r5, 4;
add.s64 %rd4, %rd2, %rd3;
ld.global.u32 %r6, [%rd4];
mul.lo.s32 %r7, %r6, %r1;
st.global.u32 [%rd4], %r7;

ret;
}
```

再来看生成的SASS代码，注意这里我们先只关注反汇编后的机器代码部分（相当于常说的`.text`部分）。实际上为了保证模块的正常载入和kernel的运行，还需要一些其他信息，这些其实是放在对应cubin文件的其他section中，以后有机会再讲。首先是`sm_30`也就是Kepler架构的SASS代码：

```
arch = sm_30
code version = [1,7]
producer = <unknown>
host = windows
compile_size = 64bit

    code for sm_30
        Function : _Z4funciPi
    .headerflags    @'EF_CUDA_SM30 EF_CUDA_PTX_SM(EF_CUDA_SM30)'
                                                                        /* 0x2282c28042823307 */
        /*0008*/                   MOV R1, c[0x0][0x44];                /* 0x2800400110005de4 */
        /*0010*/                   S2R R0, SR_TID.X;                    /* 0x2c00000084001c04 */
        /*0018*/                   S2R R3, SR_CTAID.X;                  /* 0x2c0000009400dc04 */
        /*0020*/                   IMAD R0, R3, c[0x0][0x28], R0;       /* 0x20004000a0301ca3 */
        /*0028*/                   MOV32I R3, 0x4;                      /* 0x180000001000dde2 */
        /*0030*/                   ISCADD R2.CC, R0, c[0x0][0x148], 0x2;  /* 0x4001400520009c43 */
        /*0038*/                   IMAD.HI.X R3, R0, R3, c[0x0][0x14c];   /* 0x208680053000dce3 */
                                                                        /* 0x20000002e04283f7 */
        /*0048*/                   LD.E R0, [R2];                       /* 0x8400000000201c85 */
        /*0050*/                   IMUL R4, R0, c[0x0][0x140];          /* 0x5000400500011ca3 */
        /*0058*/                   ST.E [R2], R4;                       /* 0x9400000000211c85 */
        /*0060*/                   EXIT;                                /* 0x8000000000001de7 */
        /*0068*/                   BRA 0x68;                            /* 0x4003ffffe0001de7 */
        /*0070*/                   NOP;                                 /* 0x4000000000001de4 */
        /*0078*/                   NOP;                                 /* 0x4000000000001de4 */
        ......................
```

然后是 `sm_52` 也就是Maxwell架构的SASS代码：

```
arch = sm_52
code version = [1,7]
producer = <unknown>
host = windows
compile_size = 64bit

    code for sm_52
        Function : _Z4funciPi
    .headerflags    @'EF_CUDA_SM52 EF_CUDA_PTX_SM(EF_CUDA_SM52)'
                                                                                /* 0x001c7c00e22007f6 */
        /*0008*/                   MOV R1, c[0x0][0x20] ;                       /* 0x4c98078000870001 */
        /*0010*/                   S2R R0, SR_TID.X ;                           /* 0xf0c8000002170000 */
        /*0018*/                   S2R R2, SR_CTAID.X ;                         /* 0xf0c8000002570002 */
                                                                                /* 0x001fd840fec20ff1 */
        /*0028*/                   XMAD R0, R2.reuse, c[0x0] [0x8], R0 ;        /* 0x4e00000000270200 */
        /*0030*/                   XMAD.MRG R3, R2.reuse, c[0x0] [0x8].H1, RZ ; /* 0x4f107f8000270203 */
        /*0038*/                   XMAD.PSL.CBCC R2, R2.H1, R3.H1, R0 ;         /* 0x5b30001800370202 */
                                                                                /* 0x001fc800fcc207f1 */
        /*0048*/                   SHR R0, R2.reuse, 0x1e ;                     /* 0x3829000001e70200 */
        /*0050*/                   ISCADD R2.CC, R2, c[0x0][0x148], 0x2 ;       /* 0x4c18810005270202 */
        /*0058*/                   IADD.X R3, R0, c[0x0][0x14c] ;               /* 0x4c10080005370003 */
                                                                                /* 0x081fd860fe2007b5 */
        /*0068*/                   LDG.E R0, [R2] ;                             /* 0xeed4200000070200 */
        /*0070*/                   XMAD R5, R0.reuse, c[0x0] [0x140], RZ ;      /* 0x4e007f8005070005 */
        /*0078*/                   XMAD.MRG R6, R0.reuse, c[0x0] [0x140].H1, RZ ; /* 0x4f107f8005070006 */
                                                                                /* 0x001ffc00fe2007f2 */
        /*0088*/                   XMAD.PSL.CBCC R0, R0.H1, R6.H1, R5 ;         /* 0x5b30029800670000 */
        /*0090*/                   STG.E [R2], R0 ;                             /* 0xeedc200000070200 */
        /*0098*/                   EXIT ;                                       /* 0xe30000000007000f */
                                                                                /* 0x001f8000fc0007ff */
        /*00a8*/                   BRA 0xa0 ;                                   /* 0xe2400fffff07000f */
        /*00b0*/                   NOP;                                         /* 0x50b0000000070f00 */
        /*00b8*/                   NOP;                                         /* 0x50b0000000070f00 */
        ......................
```

最后是 `sm_75`，也就是Turing架构的SASS代码：

```
arch = sm_75
code version = [1,7]
producer = <unknown>
host = windows
compile_size = 64bit

    code for sm_75
        Function : _Z4funciPi
    .headerflags    @'EF_CUDA_SM75 EF_CUDA_PTX_SM(EF_CUDA_SM75)'
        /*0000*/                   MOV R1, c[0x0][0x28] ;                 /* 0x00000a0000017a02 */
                                                                          /* 0x000fd00000000f00 */
        /*0010*/                   S2R R2, SR_TID.X ;                     /* 0x0000000000027919 */
                                                                          /* 0x000e220000002100 */
        /*0020*/                   MOV R5, 0x4 ;                          /* 0x0000000400057802 */
                                                                          /* 0x000fc60000000f00 */
        /*0030*/                   S2R R3, SR_CTAID.X ;                   /* 0x0000000000037919 */
                                                                          /* 0x000e240000002500 */
        /*0040*/                   IMAD R2, R3, c[0x0][0x0], R2 ;         /* 0x0000000003027a24 */
                                                                          /* 0x001fc800078e0202 */
        /*0050*/                   IMAD.WIDE R2, R2, R5, c[0x0][0x168] ;  /* 0x00005a0002027625 */
                                                                          /* 0x000fd400078e0205 */
        /*0060*/                   LDG.E.SYS R0, [R2] ;                   /* 0x0000000002007381 */
                                                                          /* 0x000ea400001ee900 */
        /*0070*/                   IMAD R5, R0, c[0x0][0x160], RZ ;       /* 0x0000580000057a24 */
                                                                          /* 0x004fd000078e02ff */
        /*0080*/                   STG.E.SYS [R2], R5 ;                   /* 0x0000000502007386 */
                                                                          /* 0x000fe2000010e900 */
        /*0090*/                   EXIT ;                                 /* 0x000000000000794d */
                                                                          /* 0x000fea0003800000 */
        /*00a0*/                   BRA 0xa0;                              /* 0xfffffff000007947 */
                                                                          /* 0x000fc0000383ffff */
        /*00b0*/                   NOP;                                   /* 0x0000000000007918 */
                                                                          /* 0x000fc00000000000 */
        /*00c0*/                   NOP;                                   /* 0x0000000000007918 */
                                                                          /* 0x000fc00000000000 */
        /*00d0*/                   NOP;                                   /* 0x0000000000007918 */
                                                                          /* 0x000fc00000000000 */
        /*00e0*/                   NOP;                                   /* 0x0000000000007918 */
                                                                          /* 0x000fc00000000000 */
        /*00f0*/                   NOP;                                   /* 0x0000000000007918 */
                                                                          /* 0x000fc00000000000 */
        .....................
```

对比一下可以发现三个版本的SASS的一些差异:

1. 一个显著的区别就是control codes的变化，Kepler是1+7，Maxwell是1+3，两者反汇编后指令编码前没有对应指令文本的那些行，就是control codes。Turing的control codes是内嵌在每条指令中，但并没有占用完整的64bit。所以Turing的反汇编中的无文本的指令行其实有很多bit也是参与指令编码的，不都是control codes。

2. 即使对于最简单的指令`NOP`，Kepler中的编码是`0x4000000000001de4`，Maxwell是`0x50b0000000070f00`，Turing是`0x000000000007918, 0x000fc00000000000`（第二个64bit中含有control code）。所以尽管反汇编后的指令助记词没变化，但实际上ISA还是不一样的，只是支持同样的指令功能而已。

3. Kepler中的int32的乘法用的是`IMAD`和`IMUL`，而Maxwell中都用的三个`XMAD`来组合，Turing中用回了`IMAD`。但如果看算地址常用的形式：uint64+int32*int32，Kepler和Maxwell都用的是类似`LEA`的`ISCADD`指令，Turing中用的是`IMAD.WIDE`。这些都是同样的功能在不同版本的SASS中采用了不同的实现，而它们对应的PTX代码是一模一样的。

这次就先讲这么多吧~ 下次讲讲SASS的分类和基本的指令发射逻辑。有些东西我也没有研究得很仔细，仅供参考~ 如果有什么问题，欢迎各位批评指正~