总体流程设计（2）-thread

https://zhuanlan.zhihu.com/p/137793216

None Mon May, 24 03:29

上次总体流程设计里已经介绍了CUDA程序的等级结构，今天可以开始聊一聊一些具有实用价值的设计思想。点比较散，有些并不是绝对的，要辩证的看。今天先说thread内的安排。

Thread内GPR的重用

Thread私有的GPR（General Purpose Register，通用寄存器，有时也直接称为Register，即寄存器）是GPU中最快也是最直接的资源。多使用GPR缓存中间数据，减少memory读写和重新计算，是最常用也是最初级的优化手段。常见的比如一些链式求值：v=f(g(h(x)))，就不要分成三个kernel去分别计算t0=h(x),t1=g(t0),v=f(t1). 再比如a=f(x), b=g(x), 也最好把a,b的对应元素放到同一个线程里计算，就不用重新把`x`读进来了。这不但可以减少memory的读写，同时也省去了临时变量的内存开销。当然这些其实都是挺初级的形式，也有一些更高级或者说是更隐蔽的形式。比如最简单的向量相加：

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

//...
vecadd<<<1024,1024>>>(a,b,c);
//...
```

其实也是有一些重用可以利用。因为64bit的机器上，`a,b,c`都是64bit的指针，计算`a[idx]`其实是需要计算`(*a) + idx*sizeof(float)`。这是一个64bit的无符号加法，每个线程的每个元素（三个变量）都要算一遍。而把形式稍微改一下：

```
__global__ void vecadd(float* a, float* b, float* c)
{
    int idx = threadIdx.x + 2 * blockDim.x * blockIdx.x;
    c[idx] = a[idx] + b[idx];
    c[idx+1024] = a[idx+1024] + b[idx+1024];
}

//...
vecadd<<<512,1024>>>(a,b,c);
//...
```

每个线程算了两个元素。这样 `a[idx]` 算地址的部分是可以复用的，多出来的 `1024` 可以加在Load或store指令的立即数的offset里，并不需要额外的指令。注意：这里的 `gridDim` 发生了变化。因为每个线程算了两个元素，总线程数自然要折半。把 `blockDim` 或是 `gridDim` 折半都是可以的，只是index的计算方式要相应调整。

然后，不仅可以展开2次，更多次也行。相当于每个线程做了以前几个线程的事。其中有很多公共的计算就通过线程内的GPR传递而省下了。当然，最后线程数不能太少，否则不能充分占满所有SM的话，效率就下降了。另外，这只是举个简单的例子，重点在于通过在GPR中共享中间结果来减少一些操作。实际上向量相加是一个典型的memory bound的问题，减少这些计算量对运行时间的影响还是比较小的。

GPR使用的控制

单个线程的GPR使用需要有所控制。GPR使用太多首先就会导致SM能容纳的warp数减少（Occupancy降低），其次如果需要做register spill，GPR的数据会被交换到local memory里（也许会在L1 Cache中缓存），这会极大的影响性能。GPR的使用一般是由编译器控制的，但是用户的一些编程方式会影响GPR的使用。以下是一些控制GPR使用的方式：

- 变量的生命周期要尽量控制为使用时间段。没有必要提前声明好所有临时变量，也没有必要复用之前申请过的临时变量。要用时再申请，用完就可以收回。把变量的生命周期做短，而且相互重叠的部分尽量少。这是从寄存器分配的角度来讲比较容易让编译器看懂的方式。当然现代编译器其实已经比较聪明了，但从编程的角度讲，只要不是对可读性影响太大，这个习惯其实是个好习惯。
- 选择合适的数据类型。比如每个double要2个GPR（8Byte），而float只需要一个。在精度允许的情况下，用float代替double不仅可以省很多GPR，还能大大减小ALU的负担。因为大多数消费级GPU的double ALU指令都比float ALU指令慢很多。半精度half类型如果一个GPR存两个（每个half是2Byte），可以用一些intrinsic的指令同时计算两个half，这样也可以省GPR。但不用那些指令时，单个half在运算时也是占用一个完整GPR的，并不节省。同理，一些short或是char也是一样。
- 不要用 `#pragma unroll` 去展开一些复杂循环。循环展开的目的是减少循环索引更新和跳转开销，有时也可以增加指令并行度。但是循环展开往往也意味着有更多的变量同时存在GPR中。一个简单的规律是，永远只展开一些简单循环（循环体只有几个指令）。
- 从当前的CUDA实现来看，单个kernel内所有可及的代码都是连续存储的，共用一套GPR编码。这和许多CPU那种通过保存register后完成跳转的方式是不一样的。所以，kernel内所有device函数（比如 `sinf`，`expf`）原则上都是inline的（也可能有些例外，比如printf，device的malloc和free，dynamic parallelism的启动kernel等），这样就需要在总分配的GPR中占用资源。一些复杂数学函数占用的GPR很多，有时为了实现简单，甚至会主动

用local memory代替。精度允许的情况下，用intrinsic(如 `__sinf`，`__expf`)代替精确实现，不仅可以大大减少中间GPR的使用，也可以把实际指令缩减到很少的几条。

- 上一条已经说过，kernel里几乎所有函数都是inline的。那即使是无法运行到的死代码或是不可及分支，也可能会增加GPR占用。尽管这些代码不会被运行，但它们在代码中引用到的GPR资源仍然需要在运行前就被分配好。所以，尽量去除一切死代码和不可及分支。如果是为了调试方便，可以用宏来做编译期的代码调整，或者是C++模板，但是release的版本不要出现这些代码。包括printf之类的函数也是一样，不需要的时候就不要出现在kernel内。
- 各种流控制（比如if，for等）嵌套的层数不能太多，流控制中的临时变量要尽量缩减。每深入下一层，上一层的很多变量一般都需要留存在GPR中，层数越多，需要留存的变量就越多，GPR的占用就也越多。有时编译器能做一些优化，但有时候还是需要用户自己进行调整。特别是浮点数操作，编译器为了满足C语言和IEEE floating point运算相关的规范，受到的约束很多，不如用户自己调整方便。如果需要很多重循环才能做的事，可以考虑把其中几重循环展开到多个线程去。这里有一个常见的难点是GPU对递归的支持其实是比较困难的，毕竟函数inline做递归还是很不方便，所以一些递归算法都需要用循环来实现，这个常常会导致GPR使用暴增。
- CUDA的大部分指令，除了可以使用GPR当操作数，也支持使用立即数和constant memory。GPR也有另一种uniform register的形式。例如 `FFMA` 指令（float fused multiply-add, 也即 `d=a*b+c`），就支持以下这些指令格式：

```
FFMA R0, R1, R2, R3 ;                          // R0 = R1*R2 + R3，全是普通GPR输入
FFMA R5, R10, 1.8446744073709551616e+19, RZ ; // a或b是立即数，a、b等价，可互换，所以一个支持就够了
FFMA R2, R10, R3, -1 ;                         // c是立即数
FFMA R5, R5, c[0x0][0x160], R6 ;               // a或b来自constant memory
FFMA R0, R5, R6 , c[0x0][0x168];               // c来自constant memory
FFMA R14, R19, UR6, R0 ;                       // a或b来自uniform register
FFMA R18, R16, R13, UR6 ;                      // c来自uniform register
```

这也就意味着如果这些操作数可以从立即数、constant memory来，就可以不占用GPR。uniform register虽然也是GPR，但因为是warp内共用，与thread私有的GPR不竞争资源，所以也可以起到节省每个线程GPR使用量的目的。立即数（immediate）是直接编码在指令里的常数，是编译期已知的常数，一旦编译完成就不能更改。比如 `float a=b*1.6f;` 这种，其中的 `1.6f` 就可能会被编译器作为立即数输编码到指令里。constant memory可以在运行kernel前进行赋值，但kernel运行中不能修改。constant memory有两个来源，一是用户申请。二是函数输入参数这种由驱动配置的变量。比如一个kernel的声明形式为：

```
__global__ void test(int a, float f[4],const int* in, int* out)
```

那在kernel内，参数`a`，`f[4]`和`*in`，`*out`（注意是8B指针地址）其实都在constant memory里。但`in`和`out`的内容(比如`in[0], out[3]`)则不是。善用立即数和constant memory输入对减少GPR使用还是有不少帮助的。

- uniform register是Turing才开始引入的，可以用来存储每个warp内的公共值。这可以减少一部分GPR的消耗。但当前uniform register的使用是编译器决定的，用高级语言的话用户无法直接控制。编译器要通过推理才能确定某个变量是不是整个warp是一个值，然后才有可能把它放到uniform register里。比如`v=a[blockIdx.x]`这种，编译器就能判断出来它是公共的。
- GPR的使用在编译时可以有一些控制手段。比如nvcc有`-maxrregcount`选项可以控制当前文件所有kernel最多可以使用多少GPR。如果想要每个kernel使用不同的限制，可以参考：Launch Bounds。它不是一个硬限制，更多的是通过提示编译器该kernel可能会用什么样的block配置来让编译器控制具体的GPR使用量，从而影响occupancy。

Local memory使用的控制

Local memory是线程里比较不受欢迎的资源。某种意义上讲，能用GPR解决的问题尽量不要用local memory。但有些情况下确实也很难避免。Local memory的来源一种是无法在编译期已知索引的地址。例如：

```
__global__ void addKernel(int c, const int *a, int *b)
{
    int i = threadIdx.x;
    int v[2];
    v[c] = c;
    v[2 - c] = 2 - c;

    b[i] = a[i] + v[i%2];
}
```

得到的汇编(sm_75, CUDA 10.2)如下：

```
_Z9addKerneliPKiPi:
.text._Z9addKerneliPKiPi:
        /*0000*/                   IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28] ;
        /*0010*/                   S2R R5, SR_TID.X ;
        /*0020*/                   IMAD.MOV.U32 R10, RZ, RZ, c[0x0][0x160] ;
        /*0030*/                   MOV R12, 0x4 ;
        /*0040*/                   IADD3 R1, R1, -0x8, RZ ;
        /*0050*/                   IADD3 R6, -R10, 0x2, RZ ;
        /*0060*/                   IMAD R7, R6, 0x4, R1 ;
        /*0070*/                   LEA.HI R0, R5.reuse, R5, RZ, 0x1 ;
        /*0080*/                   IMAD.WIDE R2, R5, R12, c[0x0][0x168] ;
        /*0090*/                   LOP3.LUT R0, R0, 0xfffffffe, RZ, 0xc0, !PT ;
        /*00a0*/                   IMAD.IADD R4, R5, 0x1, -R0 ;
        /*00b0*/                   LEA R0, R10, R1.reuse, 0x2 ;
        /*00c0*/                   LDG.E.SYS R2, [R2] ;
        /*00d0*/                   LEA R8, R4, R1, 0x2 ;
        /*00e0*/                   STL [R0], R10 ;                 // store v[c]
        /*00f0*/                   STL [R7], R6 ;                  // store v[2-c]
        /*0100*/                   LDL R9, [R8] ;                  // load v[i%2]
        /*0110*/                   IMAD.WIDE R4, R5, R12, c[0x0][0x170] ;
        /*0120*/                   IMAD.IADD R9, R2, 0x1, R9 ;
        /*0130*/                   STG.E.SYS [R4], R9 ;
        /*0140*/                   EXIT ;
```

可以看到，由于局部数组`v`的索引`c`是输入参数，无法在编译期得到，这样`v`只能放在local memory里。因为**CUDA的GPR是不能够索引的**，要动态索引只能通过memory的方式来访问。

我们再来看另一个稍微有点变化的例子：

```
__global__ void addKernel2(int c, const int* a, int* b)
{
    int i = threadIdx.x;
    int v[2];
    v[0] = c;
    v[1] = 2 - c;

    int vv = (i % 2 == 0) ? v[0] : v[1];
    b[i] = a[i] + vv;
}
```

这和上面的逻辑有些不同，但我们只看局部数组的处理方式。在这个kernel里，所有的v的索引都是可以在编译期预知的。这样v的元素就可以放在GPR里了。汇编确认一下，确实就没有local memory的读写了：

```
_Z10addKernel2iPKiPi:
.text._Z10addKernel2iPKiPi:
        /*0000*/                   IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28] ;
        /*0010*/                   S2R R5, SR_TID.X ;
        /*0020*/                   IMAD.MOV.U32 R6, RZ, RZ, 0x4 ;
        /*0030*/                   IMAD.WIDE R2, R5, R6, c[0x0][0x168] ;
        /*0040*/                   LDG.E.SYS R3, [R2] ;
        /*0050*/                   MOV R0, c[0x0][0x160] ;
        /*0060*/                   LOP3.LUT R4, R5.reuse, 0x1, RZ, 0xc0, !PT ;
        /*0070*/                   IADD3 R0, -R0, 0x2, RZ ;
        /*0080*/                   ISETP.NE.U32.AND P0, PT, R4, 0x1, PT ;
        /*0090*/                   IMAD.WIDE R4, R5, R6, c[0x0][0x170] ;
        /*00a0*/                   SEL R0, R0, c[0x0][0x160], !P0 ;
        /*00b0*/                   IMAD.IADD R7, R0, 0x1, R3 ;
        /*00c0*/                   STG.E.SYS [R4], R7 ;
        /*00d0*/                   EXIT ;
```

Local memory另一个使用场景是GPR实在是不够时，GPR内的值会被Spill到local memory里。如果更好的occupancy对性能帮助更大，这样做也许是有益的。另外，前面也讲过了，有些复杂函数为了实现方便，自带了local memory，这个用户就很难控制了。

从总体流程的影响上讲，Thread内的很多优化其实都是资源的分配问题。当然，更多的时候，thread会比较关心指令上是不是够精简，计算上是不是优化的够好。这个很多时候是编译器做了大部分，但用户有时也需要自己调整。这就涉及到计算怎么实现的问题，不是调度问题，就不太是我们这里讨论的重点了。