编译器优化那些事儿（13）：全函数向量化

背景

SIMT(Single Instruction Multiple Thread)和 SIMD(Single Instruction Multiple Data)是两种最为常见的并行计算架构。GPU厂商一般采用SIMT架构。英伟达的CUDA作为SIMT的代表，在AI和通用高性能计算中得到广泛应用。与此同时，CPU的向量处理器通常是SIMD架构。以下是相同简易逻辑在SIMT及SIMD架构上的伪代码：

```
// SIMT code snippet
...
int tid = threadIdx.x;
s_a = pA[tid];
s_b = pB[tid];
if (tid < 16) {
 s_c = s_a + s_b;
} else {
 s_c = s_a – s_b;
}
pC[tid] = s_c;
...
```

```
// SIMD code snippet
...
v_a = vload(pA);
v_b = vload(pB);
v_idx = [0…31] // vector sequence from 0 to 31.
v_br = [16…16] // broadcasted vector with 32 identical elements of 16
mask = vcmp(v_idx, v_br);
v_c0 = vadd(v_a, v_b);
v_c1 = vsub(v_a, v_b);
v_c = vselect(v_c0, v_c1, mask);
vstore(v_c, pC);
...
```

如上所示，SIMT是标量编程模型，编程者编写的单线程标量代码被直接“复制”到一组捆绑在一起的多线程同时执行，从而实现并行化。以CUDA为例，标量代码被一个warp中的32个线程执行。SIMT编程方式对编程者更友好，更具易用性。另一方面，SIMD直接暴露了向量语意，在一些复杂编程场景下，显得比较繁琐。在硬件仅支持SIMD架构的情况下，全函数向量化技术能够在编译层面实现SIMT到SIMD架构的转化，使SIMT代码也能高效运行在SIMD硬件上。

方法

1. Uniform analysis

首先，我们对SIMT的标量代码进行数据流分析。具体来说我们对程序中的每个量都推演一个shape属性。属性可以有以下几种情况：

(1)uniform：量在warp中所有线程中的值是统一的。比如在之前SIMT用例中(tid<16)这段，16这个常量的shape属性就是uniform。

(2)continuous：量在warp各线程中的值成连续递增态势。SIMT用例中的tid属性就是continuous。

(3)strided：量在warp各线程中成等差递增态势。比如，在各线程中一个量成0、stride、2*stride、3*stride…态势，就是strided属性的量。

(4)varying：量在warp各线程中的值不可预判。

shape属性对于全函数向量化有重要意义。主要可以表现在控制流的处理和访存的优化两方面。从控制流的角度展开，如果控制流基于某个uniform的量，那就不存在向量化的divergence。反之，如果量不是uniform的，那就会导致一部分线程走一个if分支，其余线程走else分支，这就形成了divergence的情况。在这种情况下向量宽度无法得到充分运用，导致计算资源的浪费。在访存场景下，如果load/store的下标量是continuous的shape属性，通常可以优化成coalesced的访存模式。反之，如果是stride比较大的或者varying的读写，通常只能做无规则的scatter/gather的访存。两者的效率通常有很大区别。

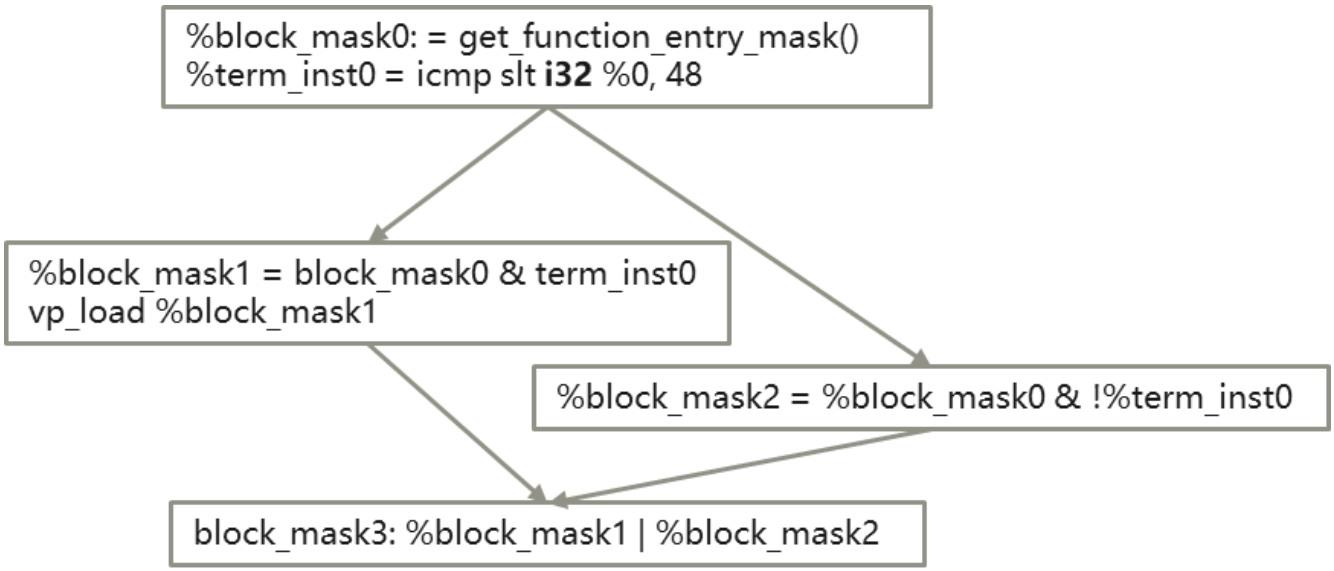
2. Mask generation

(1)if-else结构

在warp内出现divergence的情况下，需要对代码block生成mask。mask可以被用在select、load、store指令上确保程序最终结果正确。对于以下简单的if-else divergence的情况：

```
1. // simple if-else divergence case
2. ...
3. int tid = threadIdx.x;
4. if (tid < 48) {
5.  // block1
6. } else {
7.  // block2
8. }
9. ...
```

我们生成一下CFG：



对于每个block，mask由两部分组成：

1)predecessor block的block mask；

2)predecessor block的terminator instruction所生成的mask。

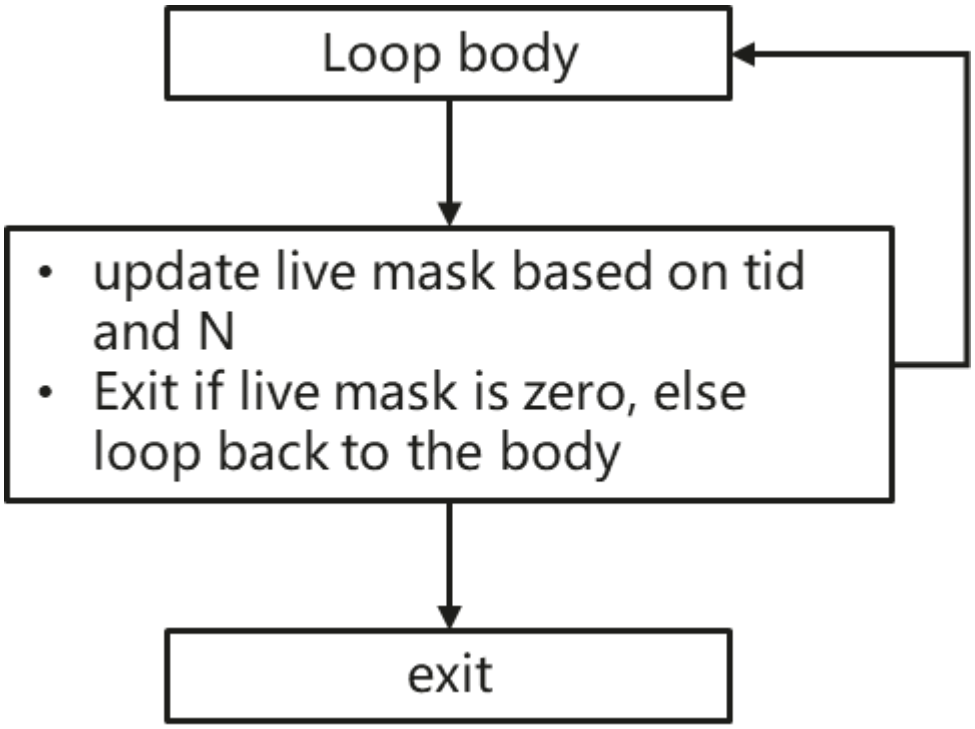
block mask由这两部分的与结果得到。上图CFG中，block1和block2是这种情况。与此同时，block3是block1和block2两个分支的合并点，block3的mask是block1和block2两个block mask的并集。通过这种mask的计算方式，我们同样可以支持nested if-else的场景。

(2)loop结构

除了if-else结构，loop结构也会造成warp内部的divergence，如果loop的induction variable或者bound是thread相关的，示例如下：

```
1. // loop divergence
2. ...
3. for (int i = tid; i < N; tid += threadDim) {
4.     // loop body
5. }
6. ...
```

我们可以看到不是每个线程都参与所有循环的计算，所以我们用一个live mask确定本循环参与的线程。只有当warp内所有线程都完成计算程序才会退出循环结构。



3. CFG Linearization

Linearization是将带mask的CFG序列化执行的过程。

4. Vectorization

对于很多指令，向量化是简单的从标量指令到向量指令的直接替换。对于下方含有uniform或continuous量的计算，我们可以做一些优化。具体来讲，int idx = tid + a; 这段是一个continuous的量 tid 和一个uniform的量 a 之间的运算，可以先不做向量化。计算可以以标量方式进行，结果 idx 是continuous的量，实际的向量化需要在 int d = b + tid; 这段发生。

```
1. void foo (int *src, int *dst) {
2.   int tid = threadIdx.x;
3.   int a = 1;
4.   int idx = tid + a; // no widen here
5.   int b = src[idx]; // b is a vector with W element
6.                     // load src behaves like load W elements starting from idx
7.                     // idx would not be widen!
8.   int c = b + a; // b is varying, a will be widen.
9.   int d = b + tid; // b is varying, tid will be widen to vector.
10.  dst[tid] = c + d;
11. }
```

同时，如果目标硬件上没有对应的向量指令，那只能通过循环执行标量指令代替。这种情况下的转换性能就不太理想。

总结

全函数向量化技术通过对变量及控制流的分析，完成SIMT标量到SIMD向量指令映射。该技术对SIMT代码中的复杂场景，包括嵌套的控制流逻辑、结构体变量等都有较为完备的处理能力。鉴于CUDA在AI领域的丰富生态，全函数向量化技术在SIMD硬件迁移CUDA生态的任务中有重要意义。

参考

(1)https://compilers.cs.uni-saarland.de/papers/karrenberg_wfv.pdf

(2)https://github.com/cdl-saarland/rv

(3)https://github.com/karrenberg/wfv