深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler 第10篇 Tensor IR语义详解 (2)

关于作者以及免责声明见序章开头。

题图源自网络，侵删。

本篇继续上一篇的主题，继续讨论Tensor IR的语义、语法。

预定义的运算符

Tensor IR预定义了一些运算符，用以对数据进行计算。这些运算符以expr节点的子类的形式暴露给Tensor IR的用户。主要有：

常量

表示了在Graph Compiler编译Tensor IR阶段的常量。`constant_node`节点内部存放了一个`std::vector<union_val>`用于存放“常量”当中的值。`union_val`是Graph Compiler定义的`union`类型，里面有float，uint64_t，int64_t三种可能的类型。`constant_node`节点还记录了这个常量的数据类型。Graph Compiler本身是一个C++项目，对于编译器本身，Tensor IR的`constant_node`节点里面存放的数值仍然是“变量”，所谓“常量”是对Tensor IR组成的“里世界”程序而言的。有读者可能会问，为什么需要用`std::vector<union_val>`，而不是单个`union_val`记录常量的值，这是因为Tensor IR允许SIMD（向量）常量。一个向量常量就需要多个`union_val`来保存值。

类型转换（cast）

Tensor IR支持将expr的数据类型进行转换，类似C语言中的cast操作：`(int)sin(x)`。这是通过`cast_node` expr节点完成的。cast节点接受一个expr输入，和一个目标类型。它的作用是保留输入expr数值语义的情况下，转换输入expr的类型，返回同样数值的新类型的expr。文本形式如下：`target_type(in_expr)`

例如，我们可以将一个f32类型的expr（例如1.5f），转换为s32类型，对于浮点数转换为整数，结果应该是浮点数去除小数点后面小数的值(例如1)。对于整数转换为浮点数，如果没有超出浮点数表示范围情况下，则可以无损地进行类型转换。

双目运算符

即两个输入一个输出的expr运算符。Tensor IR支持以下这些：加（add），减（sub），乘（mul），除（div），取余（mod）。其中取余运算不支持浮点数，其他所有的这些运算都支持整数或浮点数运算。如果双目运算符的两个输入操作数类型不一致，那么需要通过自动类型转换（auto cast，见上一篇文章）来让它们转换为相同类型。

比较运算符

即比较两个数值操作数，返回布尔值的expr运算符。支持以下这些运算：>(cmp_gt), <(cmp_lt), >= (cmp_ge), <= (cmp_le), == (cmp_eq), !=(cmp_ne)。比较运算符支持输入操作数为SIMD向量，返回的是同样长度的布尔值向量，将会在一条指令中进行多个值的比较。

逻辑运算符

即对于布尔值（boolean）的运算。支持以下这些运算：与（and）、或（or）、非（not）。这些运算在Tensor IR中主要用在if语句的条件判断和select表达式的条件中。如果输入的操作数是布尔值组成的SIMD向量，那么运算的输出也应该是同样长度的布尔值向量。

取函数地址 (func_addr)

这个expr将一个IR函数的指针转换为expr，返回一个`pointer`类型的值。

选择运算符（select）

即C语言的问号冒号表达式:`AAA?BBB:CCC`。接受三个输入，第一个是选择的“条件”，应为布尔类型或布尔向量类型，第二、三个输入应该是相同类型的两个expr，如果为向量类型，应该与第一个“条件”输入的向量长度相同。如果“条件”中的值为true，那么选择第一个输入中的值作为返回值。反正选择第二个输入的值。对于向量类型的输入，返回值向量的每个元素都类似地基于“条件”值中的对应元素在第二、三号输入expr的元素中选择。

Intrinsic

Intrinsic在传统编译器中是常见的概念。它表示编译器“认识”的一类函数，编译器由于知道这类函数的语义，可以对这类函数进行优化。在Graph Compiler中，Intrinsic和其他普通的运算符在使用上没有本质区别。我们把一部分运算符作为Intrinsic的主要原因在于添加expr子类的开发成本比较大，需要添加的代码比较多，所以我们添加了一个expr子类`intrin_call`表示一个expr大类，然后再在`intrin_call`类中区分各种不同的运算。我们支持的Intrinsic有：

min - 取两数最小值
max - 取两数最大值
abs - 取输入的绝对值
round - 将浮点数取整到最近的整数浮点数
floor - 向下取整
ceil - 向上取整

exp - 指数运算
sqrt - 平方根
rsqrt - 平方根倒数
reduce_add - 对一个向量输入的所有元素求和，返回标量
reduce_mul - 对一个向量输入的所有元素求乘积，返回标量
fmadd - fused multiply add, out = a * b + c
unpack_low - x86 vector unpack_low
unpack_high - x86 vector unpack_high
shuffle - x86 vector shuffle
permute - x86 vector permute
int_and - 整数按位与
int_or - 整数按位或
int_xor - 整数按位异或
reinterpret - 将输入值的二进制表示重新解释为目标类型。与“cast” expr不同，cast保留了数值不变，而reinterpret保留二进制数据不变
broadcast - 将一个标量输入转换为向量，向量中每个元素的值都是输入的标量值
isnan - 判断浮点数是不是NaN
saturated_cast - 饱和数值转换，将f32（向量类型）或s32（向量类型）转换为u8（向量）或s8（向量）。若转换为u8，输出为`max(0, min(input,255))`。若转换为s8，输出为`max(-128, min(input,127))`。
round_and_cast - 浮点数就近取整，返回整数类型，语义与std::roundf一致
shl - 整数左移运算（C语言中的<<) shr - 整数右移运算（C语言中的>>)
permutex2var - x86 vector permutex2var
brgemm - Batch reduce GEMM, 进行小矩阵运算的micro kernel
list_brgemm - List Batch reduce GEMM, 进行小矩阵运算的micro kernel

调用函数

Tensor IR通过`call_node`来实现对IR function的调用。`call_node`接收一个IR function作为函数调用的目标。它还需要一个`std::vector<expr>`作为实参列表。实参需要与被调用的函数的形参数量、类型一致。如果函数有返回值，那么`call_node`这个expr节点的数据类型是函数的返回值类型。`call_node`作为expr可以参与其他expr的运算和使用中。如果我们仅仅想要调用一个函数，而且丢弃它的返回值，那么需要用`evaluate_node_t`包裹这个`call_node`，然后将`evaluate_node_t`节点加入到stmts代码块中。`evaluate_node_t`的作用可以类比到C语言中函数调用本身是表达式，而不是语句，所以C语言中如果只是调用函数，不关心返回值，不能直接写为：

```
int main() {
    printf("hello world")
}
```

（注意上面的代码后面没有分号）而是要为printf这行加上分号，表示这是一个“语句”，不是“表达式”。

Tensor IR还允许通过函数指针间接调用一个函数。函数指针应该是pointer类型的expr，编译器需要通过这个expr的attr（关于attr可以在这篇找到它的用法）中的"prototype"找到函数指针指向的IR函数的原型。

控制流和并行化

Tensor IR支持的控制流主要是分支（if）和循环（for）。

if语句对应的stmt节点类型是`if_else_node_t`。它相比C语言没有什么特别之处。`if_else_node_t`接收一个布尔类型的expr作为判断条件，还接收一个stmts节点作为“then block”，如果条件为true将会跳转到“then block”。`if_else_node_t`还可以可选地接收一个“else block”（stmts），用于处理条件为false的情况。没有else block的if的文本形式为：

```
if(cond) {
    // true block
}
```

若有else block，则为

```
if(cond) {
    // then block
} else {
    // else block
}
```

Tensor IR的for循环节点`for_loop_node_t`主要接收以下几个参数：loopvar, start, end, step。loopvar是for循环的循环变量，应该是一个之前没有定义过（通过`define_node_t`或者函数参数列表）的var。这个var将会被for循环内部使用，它的作用域就在这个for循环的循环体当中。start、end、step是三个expr，表示for循环的循环变量值的起始、结束（不包括）和步长。Tensor IR中for循环的文本形式为：

```
for(loop_var, start, end, step) {
    ...
}
```

它相当于C语言的代码：

```
for(int64_t loop_var = start; i < end; i += step) {
    ...
}
```

for循环中还有一个特殊的参数，可以用于并行执行for循环。这也是Tensor IR利用到多核CPU、GPU的主要接口。如果读者了解过OpenMP，那么应该对OpenMP的`#pragma omp parallel for`语句不陌生，在C语言代码的for循环之前加上这个代码，就能利用多核执行这个循环，循环体中的代码可以并行地被每个核心执行，而且各个核心将会均摊需要执行的loopvar所有的值。在Tensor IR的`for_loop_node_t`中，如果设置`kind_`成员变量为`PARALLEL`，那么这个for循环将如同OpenMP parallel for一样被并行执行。这个成员变量的默认值为`NORMAL`，即单线程执行for循环。`kind_`成员变量还有一个可能的值为`GROUPED_PARALLEL`，这是为GPU代码预留的值，Graph Compiler暂时不支持GPU代码，不再赘述。

如果for循环标记了并行执行，具体在运行中有多少线程会被使用、每个线程会分配到多少个loopvar的值作为任务，这些都要看线程池底层的实现。Graph Compiler的Runtime API支持设置底层线程池的最大线程数量。

parallel for循环的文本描述为：

```
for(loop_var, start, end, step) parallel {
    ...
}
```

parallel for循环的循环体内部创建的本地变量和Tensor都是各个线程私有、独立的。parallel for循环体内可以访问循环体外面的全局和本地Tensor。例如如下代码并行地将本地Tensor A清零：

```
tensor A: f32[100]
for(i, 0, 100, 1) parallel {
    A[i] = 0.0f
}
```

parallel for循环体可以只读访问循环体外的本地变量（不能写入），以下代码是合法的：

```
var value: f32 = 1.0f
tensor A: f32[100]
for(i, 0, 100, 1) parallel {
    A[i] = value + i
}
```

对于全局变量，parallel for循环体可以自由修改，但是需要注意并发写入内存造成的冲突问题。

在Graph Compiler底层，会把parallel for的循环体提取出来作为一个新的函数，在原来parallel for的位置，会通过Runtime API来调用线程池创建线程。Tensor IR用户无需关心这些细节，可以直接使用parallel for表示并行执行的代码。编译器中将会自动将parallel for的IR进行转换。例如，原有使用parallel for的IR如下：

```
func main(): void {
    var value: f32 = 1.0f
    tensor A: f32[100]
    for(i, 0, 100, 1) parallel {
        A[i] = value + i
    }
}
```

编译器底层将自动转换为：

```
func for_body(i: index, value: f32, A: f32[100]): void {
    A[i] = value + i
}

func main(): void {
    var value: f32 = 1.0f
    tensor A: f32[100]
    evaluate{sc_parallel_call_cpu(func_addr(for_body), 0, 100, 1, value, A)}
}
```

Tensor IR和C/C++交互

有关如何从C++编译和调用Tensor IR的函数，我们将在之后有关JIT编译的文章中讨论。这里我们主要讨论Tensor IR如何访问C/C++的函数、以及C++代码如何访问到编译后Tensor IR中的全局变量。

首先需要厘清，Tensor IR本身是“代码”，其中的变量、Tensor仅仅是代码的一部分，在编写Tensor IR的时候，是不可能获得IR中Var和Tensor的具体的值的（如果不考虑它们是常量的情况下）。经常有开发者问，某个Tensor IR编写代码中，这个Tensor当中的值是多少呢？类比C语言，我们不可能指着一个还没有编译的C语言源文件里面的`float* A`说，`A[100]`的具体值是多少？因为这个代码根本还没有执行，所以不可能有具体的值。只有在编译（JIT）然后执行时，Var和Tensor中才能有具体的值。有这样的困惑和混乱是正常的，因为Graph Compiler有两次“运行”的时间，第一，是编译器本身运行的时候，这个时候编译器内部所有的变量、内存的值是

可以调试、确定的，但是编写的Tensor IR本身还只是C++的对象，尚未编译；第二，是Tensor IR编译后实际运行的时候，这个时候Var和Tensor才有具体的值。

首先讨论Tensor IR中如何调用C/C++的函数。我们在上一篇文章讨论IR function的时候已经说到，IR function可以申明但不定义一个函数。被申明的IR函数只是一个空壳（没有body），Graph Compiler在编译Tensor IR到可执行二进制代码的时候将会从函数申明的函数名字，找到Graph Compiler runtime当中定义的函数地址，进行“链接”。生成的代码将可以调用链接到的Runtime函数。具体有哪些函数可以被调用、具体的“函数名”到函数指针的转换，可以参考symbol_resolver.cpp。在Tensor IR这一侧，如果想要调用Runtime提供的函数，可以使用builtin.hpp中预定义的IR函数申明。例如这个头文件中提供了`print_int`函数，用于在运行的时候打印s32类型的expr到stdout中。之前提到的内存alloc、free，以及线程池调用，都是通过IR函数申明的方式进行的。

接下来我们讨论C++中如何获取编译后的Tensor IR中全局变量、Tensor的值。JIT引擎在编译IR module之后，会返回一个JIT module对象。其中的`globals_`成员指向原来IR module中全局数据（变量、Tensor）的数据表。这个数据表的类型是`statics_table_t`，可以使用其中的`get`方法通过变量、Tensor的名字查找到它的指针。

还有一个我常被问到的问题是，如何将一个编译器中的`std::vector`当中的值存入一个IR的Tensor中。Graph Compiler允许全局Tensor（也就是定义于整个IR module的Tensor）拥有初始值。可以通过`tensor_node`的`init_value_`成员来设置。我们可以将`std::vector`当中的值存入`init_value_`中，完成C++当中的值到Tensor IR中的值的转化。

到这里我们终于将Tensor IR的“语法规则”大致描述了一遍。希望可以帮助到读者理解Tensor IR的，为之后通过Tensor IR编写高性能kernel打下基础。