深入了解 oneDNN 神经网络计算图编译模块 – oneDNN Graph Compiler Tensor IR语义详解 (1)

关于作者以及免责声明见序章开头。

题图源自网络，侵删。

我们已经花了两篇文章的篇幅来介绍Tensor IR：

之前的这两篇文章主要解答的问题是Tensor IR是什么，以及在Graph Compiler的C++代码中如何表示和生成Tensor IR。有关Tensor IR还有一个重要的问题本系列文章没有说明清楚：如何组织、编写Tensor IR，编写出一个想要的计算函数（或者Op）。

GraphCompiler是一个编译器，而Tensor IR是编译器内部的用户代码表示。所以Tensor IR本身可以认为是一种编程语言。常见的编程语言通常是通过文本来告诉编译器想要执行或编译的代码的。例如对于Python编译器（解释器）来说`print("Hello world")`这样的代码会先认为它是字符串，然后经过parser解析成语法分析树（AST），然后进行进一步的翻译和执行。对于Graph Compiler的Tensor IR部分来说，我们不需要将代码parse成语法分析树的过程，因为Tensor IR已经是结构化的用户程序表示了。我们之前提到Tensor IR本身的表达形式是对标C语言的，每种IR类型基本上对应了C语言中的某种语法元素，所以GraphCompiler在解析Tensor IR的时候也需要有一定的语法规则。每种Tensor IR都有不同的语义，Tensor IR的使用者（实现Op内部算法的开发者，或者直接使用Tensor IR的用户）需要合理地组织Tensor IR来编写Op的实现程序，然后将Tensor IR作为GraphCompiler后端的输入来生成代码。所以在这一点上可以看到，Tensor IR本身也是一种编程语言。

本文开始，我们将分两篇文章来从“编程语言”的角度展开描述Tensor IR的“语法”和各个常用Tensor IR的语义。

另外，由于Tensor IR本身是C++对象，我们难以在文章中通过文字直接形容一个复杂的IR对象。还好我们在之前两篇关于Tensor IR的文章中已经提到，Tensor IR是可以通过`to_string()`方法转换为字符串表达形式。这种字符串表达形式和常见的编程语言的代码较为类似，本文在描述Tensor IR的会经常使用字符串表达形式来表示一个复杂的Tensor IR。还有Tensor IR在设计时对标了C语言，所以本文中对一些IR语法概念的描述将会类比到C语言中对应的概念。

IR Module和IR function

我们先从编程语言的角度来“复习”一下IR Module和IR function这两个概念。IR Module可以类比为C语言中单个".c"文件。我们知道C语言中一个".c"源文件中可以存放两类对象，即函数和全局变量。类似地，在Tensor IR的IR Module中，我们也可以放入多个IR function和全局变量的定义。同一个IR module中的IR function之间可以互相直接调用，而且IR function也可以访问同一个IR Module中的全局变量。IR module是Graph Compiler后端编译代码的最小粒度——每次调用Graph Compiler生成代码都需要传入一个IR module。

IR function对应了C语言的函数概念。函数可以有参数列表作为它的形式参数。和C语言相同，IR函数可以有0个或者1个参数。Tensor IR目前不支持直接通过返回值返回Tensor。如果需要一个IR函数通过Tensor返回结果，可以将函数外部预先的分配的Tensor通过参数传给函数内部，函数内部直接对Tensor写入值。用户定义的IR function的`body_`成员指向了组成函数代码的`stmts`对象，这可以类比到C语言函数最外层的"{}"花括号。Tensor IR还支持“声明”外部函数。外部函数在IR中的`body_`成员指向了空指针，表示这个函数是在IR module之外定义的，需要Graph Compiler通过在编译IR阶段通过函数申明的名字，来寻找到这个外部函数的二进制实现，然后“链接”到JIT编译之后的IR module中。Tensor IR中有许多依赖外部函数的地方。例如Tensor IR底层申请Tensor内存和调用线程池并行运行函数时，都需要依赖Graph Compiler提供的Runtime函数，这些函数通过IR function申明的方式提供给Tensor IR使用。

IR function转换为文本形式之后，有如下表示形式：

```
func func_name(var1: type1, tensor2: type2[A * B * C], ...): type3 {
    ...
}
```

其中`fun_name`表示这个IR function的名字，`var1`和`tensor2`是函数的参数，它们分别是var和tensor，将在下一节详细讨论。`var1`是`type1`类型的变量。`tensor2`是`type1`类型的的Tensor，维度是`A*B*C`。函数返回值类型为`type3`。

变量：var

Tensor IR支持定义和使用变量（var）。变量表示了一个计算机中存储的单个数据值。R中`var_node`节点本身是expr（表达式）类型，所以可以将一个var和其他expr一起组成复杂的expr，var出现在其他表达式中，代表了读取这个var的值。变量也可以出现在“赋值语句”中，表示对这个变量进行赋值。Tensor IR中的赋值语句就是`assign_node_t`。文本形式的对变量赋值的`assign_node_t`如下所示：

```
var_name = some_expr
```

其中`var_name`表示了变量的名字。`some_expr`是变量中的新值。

在使用变量之前需要先定义变量。Tensor IR有三种方式定义变量。

1）将`var_node`放在IR function的参数列表中。我们在上一节IR function的讨论中已经看到了这样定义var的方式。
2）通过在函数体中添加`define_node_t`来定义函数内部的本地变量。Graph Compiler中通常会把本地变量对应到CPU、GPU中的寄存器或者栈内存上。`define_node_t`是stmt节点，表示定义了一个Tensor或者Var。在这个define节点之后，同一个stmts中可以使用定义的Var或者Tensor。在define节点之前、或者是define节点所在的stmts之外，则不能使用它定义的Tensor或者Var。这与C语言中变量的定义是类似的：一个变量的“声明周期”在它所在的花括号"{}"block结束为止。

3）通过在IR Module中添加`define_node_t`来定义全局变量。全局变量可以被IR module中所有的IR function使用。

定义变量的`define_node_t`节点的文本形式如下：
```
var a: type [= init_value]
```

在定义变量的时候可以赋给变量一个可选的初值（即上面的`= init_value`）部分。Var有一个重要的属性，就是它的数据类型，也就是上面文本形式的`type`部分。有关数据类型将在下一节展开描述。

Tensor IR的类型系统和向量化

每个表达式（expr），包括var，都有它的数据类型。我们先从简单的标量（Scalar）数据类型来理解。Tensor IR中支持以下这些基础数据类型（括号中的是这个数据类型在Tensor IR中的名字）：

1. 浮点数：32位（f32），bfloat16（bf16）
2. 无符号整数：8位（u8），16位（u16），32位（u32），64位（index）
3. 有符号整数：8位（s8），32位（s32）
4. 通用指针(pointer)，数据指针（f32*, *u8**, …）
5. 布尔类型（boolean）

选择这些数据类型作为基础类型，主要考虑到深度学习计算中对于数据的计算主要以32位浮点数为基础（几乎不会用到64位double浮点数）。在量化模型中常用8位整数和bfloat16类型表示压缩后的数据，用于节约内存吞吐和计算。另外我们通常使用64位整数作为Tensor索引。通用指针(pointer)对应了C语言中的`void*`，表示一块指向未知类型（或者用户根本不关心的类型）的内存。编译器中还定义了指向基础数据类型的指针。注意，Graph Compiler不支持更复杂的指针类型，例如指向指针的指针等。

GraphCompiler允许用户输入的Tensor IR中的数据类型不完全匹配。例如一个IR function（例如名字叫`func_a`）的形式参数中声明了`f32`的参数，但Graph Compiler允许用户在调用这个函数的时候传入`s32`的值。Graph Compiler内部会自动为传入的`s32`类型的expr外面包裹一个`cast_node`节点，将它转换为`f32`类型。这在C语言中也有类似做法，编译器将会自动插入一些合法的转换。在Graph Compiler中允许进行的自动类型转换被称为Upcast，也就是原则上，允许将数值范围较小的数据类型自动根据需要转换为数据范围更大的类型（有符号整数可以自动转换为无符号整数）。而Graph Compiler不允许将大的数据范围的数据类型自动转换为小范围的数据类型。例如`s32`可以自动转换为`f32`、`index`、`u32`等，但是反过来`f32`无法自动转换为`s32`。

在什么情况下需要用到自动类型转换呢？上面说到了在函数调用时，数据类型不匹配的情况，例如上文例子中的函数调用`fun_a(1)`的实际参数为"1"这个`s32`类型的常量，那么Graph Compiler将会自动将代码转换为`fun_a(f32(1))`，其中`f32(1)`表示一个`cast_node`，将"1"这个常量转换为`f32`类型。另外还有一个例子就是加法expr的计算需要两个输入值的类型相同。Graph Compiler将会判断两个输入expr的类型哪个需要进行类型转换，例如`1.0f + 2`这个加法expr，用户想要的计算是将1.0这个32位浮点数和2这个整数常量相加，那么编译器会自动为整数expr添加cast，使得加法节点满足加号左右的数据类型一致：`1.0f + f32(2)`。

如果编译器判断需要的数据类型和输入的数据类型不一致，而且不能进行自动类型转换，那么编译器将会报错。插入自动类型转换的工作由Graph Compiler的"auto caster"这个pass来完成。

现代CPU和GPU为了提高指令级并行度，引入了SIMD指令（Single Instruction Multiple Data），通过一条指令可以操作多个数据。我们把不使用SIMD的、每条指令（或者IR代码）只操作一个数据的程序称为标量（Scalar）程序；如果将标量程序改为使用SIMD，我们称之为向量化的过程（Vectorize）。GCC和LLVM编译器都在一定程度上支持“自动向量化”功能，可以在编译器内部自动将标量程序进行向量化改写，但是自动向量化仍然有一定的局限性，对于一些复杂情况仍然无法做到向量化。Graph Compiler没有自动向量化的功能，它需要用户（使用Tensor IR的开发者）手动标注哪些部分的IR是使用SIMD操作多个数据的。Graph Compiler在类型系统中添加了一个“维度”——lanes，用于表示类型的SIMD长度，也就是将多少个标量数据打包到一起。SIMD类型的expr也可以正常参与各种expr的运算，例如加减乘除等等。如果一个expr的数据类型的lanes大于1，那么有关这个expr的计算都需要通过SIMD指令进行处理，CPU、GPU底层将会把lanes个数据打包到一起同时处理。Tensor IR中的变量（var）的数据类型也支持SIMD类型，这些变量如果不在栈中，就可以被映射到硬件的SIMD寄存器中。由于硬件上能处理的SIMD长度是有限的，所以针对不同的底层硬件，Tensor IR的SIMD lanes长度只能允许硬件支持的大小。例如对于支持AVX512的CPU来说，它支持128位（SSE指令集），256位（AVX，AVX2指令集）和512位（AVX512指令集）向量寄存器。所以对于`f32`数据类型来说，AVX512的CPU硬件支持的Tensor IR SIMD lanes有4，8，16这三种，因为有`4=128/32`，以此类推有8和16。在这样的硬件平台上，其他长度的`f32`的SIMD lanes是非法的。

在文本形式的Tensor IR中，用`{base_type}x{lanes}`表示SIMD数据类型。例如`f32x16`表示16个`f32`打包在一起的512位的SIMD类型。

Tensor的定义和访问

在第一篇介绍Tensor IR的文章中已经定义了Tensor IR中的Tensor概念。我们可以认为Tensor IR中的Tensor是行优先的多维数组（如果不考虑非默认stride的情况下），在内存中每一行是连续排列的，Tensor与Var有相似之处，它们都用于IR中表示数据存放的地方。不同之处在于，Var只能存放单个（Scalar）数据，或者SIMD支持的长度的数据，而不能支持任意长度的存取。Tensor则可以在内存用量允许的情况下支持任意长度的数据存取。Tensor本身是多维数组，可以与Graph IR中的Tensor对应起来；而var只是Tensor IR内部处理、暂存数据的地方，也只支持1维（SIMD）和0维（Scalar）。有的开发者可能会将Tensor和SIMD类型的var搞混，因为它们都能表示多个数据的组合。这两者最大的不同是，Tensor必定存放在内存之中，而Var理想情况下使用寄存器来存放，还有区别在于Tensor支持多维度，支持从Tensor中索引取出元素，而SIMD Var暂时不支持这些。

在使用Tensor之前，需要事先定义它。和Var类似，定义Tensor有三种方式：

1）将`tensor_node`放在IR function的参数列表中。这表示将实际参数Tensor的引用传入函数中。跨函数传递Tensor都是通过传引用方式处理的。
2）通过在函数体中添加`define_node_t`来定义函数内部的本地Tensor。和Var一样，一个本地Tensor的“声明周期”在它所在的花括号"{}"block结束为止。对于本地Tensor，编译器在生成代码的时候，会在创建Tensor的位置为Tensor分配内存（对于小Tensor，会在栈上分配，对于大Tensor，会在堆上分配）。在本地Tensor生命周期结束的时候（花括号的"}"位置，也就是stmts节点结束的时候），编译器会自动生成释放内存的代码。这样函数内部的Tensor声明周期可以

交由编译器自动管理，使用者无需手动调用内存申请或者释放的函数。
3）通过在IR Module中添加`define_node_t`来定义全局Tensor，可以被IR module中所有的IR function使用。

`define_node_t`定义Tensor的文本形式是：
```
tensor tsr_name: type[A*B*...]
```

其中`tsr_name`是定义Tensor的名字，`type`是Tensor元素的类型，`A*B*...`表示Tensor每一维的长度。需要注意的是，Tensor的元素类型表示的是Tensor中每个元素的类型。它只能是标量类型，不存在类似`f32x16`的Tensor。

为了操作Tensor中的某个元素，可以对Tensor进行索引操作，Tensor IR中称之为`indexing`。`indexing_node`是expr，表示取出Tensor中某个位置的元素。它的内部注意由两部分组成：被索引的Tensor和Tensor内部的位置（index）。对于N维的Tensor来说，它的index也应该有N维，并且每个维度的数值不应该超过Tensor相应维度的长度。如果不涉及赋值号“=”，那么indexing会被编译为对Tensor内存相应位置的内存读取。和Var一样，indexing也可以在赋值号（`assign_node_t`）左边，表示对Tensor某个位置元素进行赋值。下面的例子是，定义3维Tensor A，长度是[100,200,300]。对它的[0,100,200]位置赋值1.0，然后计算Relu，然后写回同样位置：
```
tensor A: f32[100 * 200 * 300]
A[0,100,200] = 1.0f
A[0,100,200] = max(A[0,100,200], 0.0f)
```

我们在上一节讨论了SIMD数据类型。对Tensor的读写也支持一条IR进行多个元素读写的SIMD操作。indexing节点本身是expr，所以我们可以设定一个访问Tensor的indexing节点的SIMD lanes，来实现向量化的读写Tensor内存。对于SIMD lanes为N的indexing，表示了访问这个索引位置的N个连续元素。文本形式的Tensor IR中，向量化的indexing节点会在index后面用`@ N`来标注lanes数量为N，例如：`A[0,100,200 @ 16]`表示的是从Tensor A的`0,100,200`位置开始的16个相邻连续元素。虽然不存在向量类型元素的Tensor，但是我们可以从Tensor中读取出SIMD类型的元素。

Tensor在Graph Compiler底层会被翻译为指针，对于tensor的indexing则是基于指针进行偏移后的内存访问。Tensor本身也是expr，可以被用于计算之中。Tensor IR中，Tensor本身的数据类型是对应元素的指针类型。例如`f32`的Tensor本身的类型是`f32*`。这应该比较好理解，可以类比到C语言的“数组名”本质上的类型指针，可以自动“退化”到数组元素类型的指针类型。Tensor IR也允许将一个Tensor作为指针传递给其他IR function。

还有一点需要注意：Graph Compiler中假设了同一个IR function中，不同Tensor的实际内存地址是不重叠的。也就是说在编写Tensor IR程序的时候需要避免将同一块tensor传入一个函数的两个不同参数。这个假设有利于编译器进行内存访问的优化。对应于C语言的“restrict”关键字和LLVM的“noalias”属性。

Tensor指针偏移运算和切片：Tensorptr

对于Tensor来说，Tensor IR的使用者还有一种常见的操作：获取Tensor某个元素的地址。类比到C语言中，相当于`&A[i]`这样的操作。Tensor IR中提供了`tensorptr`这种expr来表达这样的地址偏移运算。`tensorptr`中主要是由两部分组成，一个是“base”，表示内存地址的起始位置，另一个是“index”，表示对“base”地址进行偏移的索引。其中base可以是Tensor或者另一个Tensorptr，而

index部分则是对这个Tensor或者Tensorptr的索引号。和`indexing`节点一样，index和base的维数需要一致。例如我们取上文中三维Tensor A的`[0,100,200]`位置的地址，可以用Tensorptr表示，对应的文本形式是：

```
&A[0,100,200]
```

Tensorptr的数据类型应该和base Tensor或者Tensorptr一样，也是元素的指针类型。我们可以将计算到的Tensorptr结果传给另一个IR function，这个IR function的参数列表中可以使用Tensor作为参数来接受这个指针，进行进一步访问。

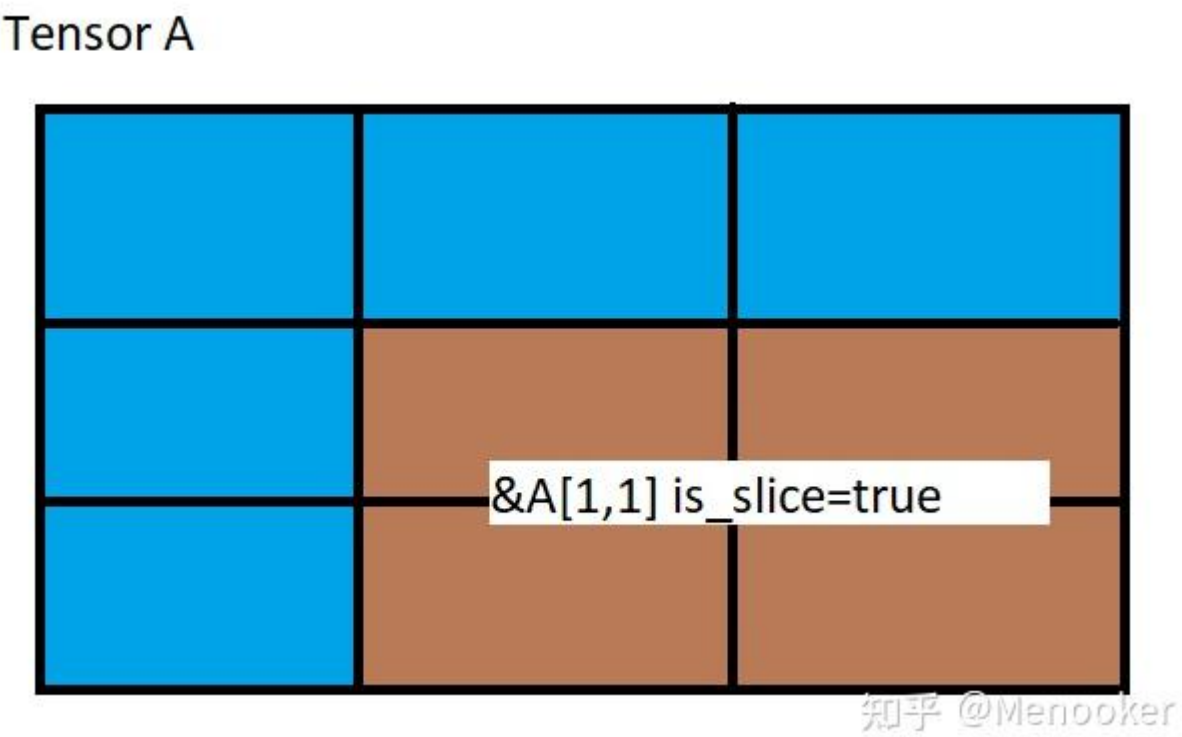
可以认为Tensorptr是对原有base tensor的“切片”，它从base tensor的某个位置开始（由index指定）将tensor的后面部分“切”了下来。对于被切下的部分Tensor我们可以可以通过`indexing`节点来访问Tensor切片指向的元素内容。对于“切片”的存取会映射到原始base tensor的相应位置。Tensorptr的“切片”可以有两种不同的“切法”，对应于对于`is_slice=true/false`两种模式。Graph Compiler中最为常用的`is_slice=true`模式中，如果对上文出现的Tensorptr: `&A[0,100,200]`进行indexing访问，访问这个切片的第`[16,0,32]`号位置，那么对应的indexing节点会是：

```
&A[0,100,200][16,0,32]
```

它其实是在访问原base tensor A中的元素：

```
A[0+16,100+0,200+32]
```

`is_slice=true`模式相当于在原始base tensor的某个起始位置开始横竖“切”几刀。后续对于这个Tensorptr的indexing索引访问可以把每个维度的indexing节点索引和Tensorptr起始偏移的索引相加，映射到原始base tensor上。如下图所示：



本篇先告一段落。我们下一篇继续介绍Tensor IR支持的各种计算、控制流，如何在Tensor IR中使用并行化处理数据、调用函数和访问C++的数据和runtime函数。