Proposal for
TLX: Tensor LLVM eXtensions

Rationale

Diverse hardware vendors are developing new hardware support for (mostly dense) tensor computations, which have become increasingly important for machine learning applications. These include both ISA extensions on CPUs and GPUs (such as Intel AMX, Power MMA, NVIDIA's tensor cores, AMD's matrix cores, and Qualcomm's HVX vector ISA) and dedicated accelerators for compute offload (such as NVIDIA's NVDLA, Amazon's Inferentia and Trainium, and numerous ML accelerators from smaller companies). While ML workloads are the primary motivation and likely to be the dominant use cases, other tensor-intensive application domains, such as image processing, scientific computing, quantum simulations, financial modeling, and others can benefit from this hardware support as well, via languages like C++, DPC++, Julia, Fortran, Halide, CUDA, OpenCL, and others.

LLVM can play a crucial role in making it easier for these vendors to create optimizing compiler back-ends for their emerging hardware (if the existing vector and matrix support in LLVM were generalized to support tensor operations). LLVM is already widely-used today by many of the vendors that develop these tensor architectures, e.g., to target CPUs and GPUs. LLVM is highly retargetable, by design. For the CPU targets, LLVM allows an integrated code generation framework for tensor operations with optimized intermixing of scalar, 1-D vector and 2-D matrix operations in the same code section (e.g., loop body). And LLVM has front-ends for a wide range of high-level languages, including essentially all the languages used widely for relevant application domains today.

No existing infrastructure we know of meets these needs. MLIR is likely the best option, and we believe it is entirely complementary to LLVM. MLIR provides strong support for high-level tensor operations in *TOSA*, relevant optimizations in *Affine* and *Linalg*, and lowering paths to accelerators, GPUs and (via the LLVM dialect) CPUs. Crucially, however, MLIR does not have a low-level code generation framework that is retargetable to diverse hardware: it relies on LLVM for this purpose. If LLVM could be extended with tensor operations and a corresponding retargetable tensor code generation framework, MLIR could leverage this as well. Moreover, there are enough vendors and also languages that rely heavily on LLVM (but don't use MLIR) that it seems worthwhile to have a high-quality tensor code generation framework in both LLVM as well as in MLIR. Ideally, both systems would largely share the same code.

The broad goal of our project is to add a retargetable tensor code generation framework to LLVM. We are currently working on a prototype implementation with our collaborators at Amazon AWS, Intel, IBM and Qualcomm. This RFC focuses on the first stage: extending the LLVM IR with tensor operations which we refer to as TLX (Tensor LLVM eXtensions).

Overall Project Objectives

- A unified retargetable code generation and optimization framework for LLVM to target diverse tensor architectures with a common set of IR extensions, instead of using target-specific solutions.
- (Subject of this RFC.) A single set of target-agnostic tensor extensions in LLVM IR that higher-level tensor code generation frameworks such as XLA, Halide, TVM, MLIR, etc. can target, instead of lowering to target-specific intrinsics in LLVM, while retaining the optimizations in these high-level frameworks.
- A pathway for LLVM-based languages such as C/C++, DPC++, Fortran, Rust, Julia, etc. that do not have front ends for compiler systems like MLIR, TVM, XLA, etc. to target modern tensor architectures by lowering to our tensor extensions in LLVM.
- Target-independent optimizations (e.g. peephole optimizations and generic SSA-based optimizations) and also flexible code generation capabilities in LLVM that could involve mixing instructions operating on vector and rectangular registers, and involve developing cost models which could help reduce register spills and maximize usage of available hardware resources.
- Contribute our tensor extensions (this RFC) and retargetable code generation framework (as a followup) to the LLVM project for the community to experiment with and provide feedback.

RFC: Introduction of Tensor Concept in LLVM

To achieve our objectives, we need to introduce the concept of tensors in LLVM, which are currently missing. To do this, we need to add a tensor type (i.e., an N-dimensional data type), generalizing 1-D vectors and 2-D matrices. We also need crucial tensor operations which front-ends for high-level languages can target, and which represent or can be implemented via ISAs of different tensor architectures.

Implementation of Tensor Type in LLVM

Overview:

The concept of dense tensors can be implemented as a new, first-class n-dimensional vector type in LLVM. However, doing this would be extremely intrusive since it will require changes to hundreds of files in LLVM. While this may be the correct option in the long term, once the design has been properly evaluated and refined, the effort required to do so for an initial prototype and evaluation is not justified. So we propose to implement the tensor concept as an LLVM intrinsic called **`llvm.tensor.typeinfo`**, while representing tensor data in “flattened” form as ordinary LLVM vector types. The intrinsic takes as operands a “flattened” LLVM vector, together with shape, layout and padding vectors, and returns a value of LLVM token type. By returning a value of token type, this intrinsic avoids the risk of being eliminated by optimizations (especially,

dead code elimination) when it has uses. This intrinsic is marked with the **'readnone'** and **'speculatable'** attributes so that it does not inhibit optimizations like redundancy elimination, dead code elimination, code motion, etc.

```
token llvm.tensor.typeinfo(<llvm-vector-type> %tensor,
<n x i32> %shape, <n x i32> %layout, <n x i32> %padding)
```

Operands:

Operand	Description
%tensor	n-dimensional tensor value represented as a "flattened" vector
%shape	Vector of dimension values of a tensor
%layout	Vector of permutation of dimension indices ranging from 0 to n-1
%padding	Vector of padding values along every dimension of a tensor

Result:

Result	Description
token value	LLVM value of token type associated with a tensor value

Semantics:

The '**llvm.tensor.typeinfo**' intrinsic is used to produce a unique token value associated with a tensor value represented as a "flattened" vector. The layout operand of this intrinsic is expressed as a permutation of dimension indices (from 0 to n-1 for an n-dimensional tensor). This represents tensor layouts in LLVM in a generic way. The number of elements in shape, layout and padding vectors must be the same and equal to the number of dimensions of the given tensor.
Note that this intrinsic is only meant to hold information such as shape, layout and padding of a tensor value in LLVM IR. It does not read nor write memory nor perform any computations, and it does not exhibit any kind of undefined behavior.

Example:

```
; The first argument (%tensor) is the tensor that is being modeled as a flattened vector. The second
argument is the shape (16 x 5 x 3), the third argument is layout (<0, 1, 2>) and the fourth argument is
padding (<3, 2, 1> along the corresponding dimensions) for the given tensor.
```

```
input = call token @llvm.tensor.typeinfo(<240 x float> %tensor, <3 x
i32> <i32 16, i32 5, i32 3>, <3 x i32> <i32 0, i32 1, i32 2>, <3 x
i32> <i32 3, i32 2, i32 1>)

; The first argument is the input virtual tensor register,  and the second argument is the new permutation
of the layout of the input tensor. This operation produces a tensor of layout  <2, 0, 1>.
%output = call <240 x float> @llvm.tensor.transpose(token %input, <3 x
i32> <i32 2, i32 0, i32 1>)

; The first argument (%output) is the output tensor that is being modeled as a flattened vector. The
second argument is the new shape (3 x 16 x 5), the third argument is layout (<2, 0, 1>) and the fourth
argument is the new padding (<1, 3, 4> along the corresponding dimensions) for the output tensor.
%typed_output = call token @llvm.tensor.typeinfo(<240 x float>
%output, <3 x i32> <i32 3, i32 16, i32 5>, <3 x i32> <i32 2, i32 0,
i32 1>, <3 x i32> <i32 1, i32 3, i32 2>)
```

Tensor Operations in LLVM

llvm.tensor.load

Overview:

This operation loads a tensor or sub-tensor with the given shape, layout and padding from memory into a register. This operation is strided, unlike the existing load instruction in LLVM, to be able to load sub-tensors from memory. This intrinsic is marked with `'speculatable'` attribute to prevent it from inhibiting optimizations like redundancy elimination, dead code elimination, code motion, etc.

```
token llvm.tensor.load(<element_type>* %mem_ptr, <n x i32> %shape, <n
x i32> %layout, <n x i32> %pad, <n xi32> %strides)
```

Operands:

Operand	Description
%mem_ptr	Starting address of a tensor/subtensor in memory
%shape	Vector of dimension values of the loaded tensor/sub-tensor
%layout	Vector of permutation of dimension indices ranging from 0 to n-1
%padding	Vector of padding values along every dimension of the loaded

	tensor/sub-tensor
%strides	Vector of strides in memory along every dimension of the loaded tensor/sub-tensor

Result:

Result	Description
%output	token value representing the output tensor

Semantics:

The '**`llvm.tensor.load`**' intrinsic loads a tensor or subtensor with the given shape, layout and padding from memory into a register. This operation is strided based on **`%strides`**, unlike the existing load instruction in LLVM, to be able to load subtensors from memory since sub-tensors are not laid out contiguously in memory. This intrinsic reads from memory, but does not write to memory.

Example:

```
; This loads a sub-tensor from the memory location pointed to by %mem_ptr. The sub-tensor has the shape <16 x 6 x 4> (second argument), layout <0, 1, 2>  (third argument) and zero padding (fourth argument). The strides in memory along every dimension are <0, 0, 8>, which means that the rows of the loaded sub-tensor have a distance of 8 bytes in memory. This produces a unique token %tensor..
%tensor = call token @llvm.tensor.load(i8* %mem_ptr, <3 x i32> <i32 16, i32 6, i32 4>, <2 x i32> <i32 0, i32 1, i32 2>, <3 x i32> <i32 0, i32 0, i32 0>, <3 x i32> <i32 0, i32 0, i32 8>)
```

llvm.tensor.store

Overview:

This operation stores a tensor or subtensor from a register into memory. This operation is strided, unlike the existing store instruction in LLVM, to be able to store sub-tensors into memory. This intrinsic is marked with **`'readnone'`** attribute to prevent it from inhibiting optimizations like redundancy elimination, dead code elimination, code motion, etc.

```
void llvm.tensor.store(<element_type>* %mem_ptr,token %tensor, <n xi32> %strides)
```

Operands:

Operand	Description
%mem_ptr	Starting address in memory where tensor is stored
%tensor	Stored tensor/subtensor
%strides	Vector of strides in memory along every dimension of the stored tensor/subtensor

Result:

Intrinsic does not return anything.

Semantics:

The '`llvm.tensor.store`' intrinsic stores a tensor or subtensor from a register into memory. This operation is strided based on `%strides`, unlike the existing store instruction in LLVM, to be able to store sub-tensors to memory since sub-tensors are not laid out contiguously in memory. This intrinsic writes to memory, but does not read from memory.

Example:

```
%tensor = call token @llvm.tensor.typeinfo(<240 x float> %tensor, <3 x i32> <i32 16, i32 6, i32 4>, <3 x i32> <i32 0, i32 1, i32 2>, <3 x i32> <i32 0, i32 0, i32 0>)

; This stores a tensor from the memory location pointed to by %mem_ptr and the second argument is the stored tensor itself. The strides in memory along every dimension are <0, 12, 10> (third argument), which means that the rows of %tensor are stored 10*sizeof(float) bytes apart and columns of %tensor are 12*sizeof(float) bytes apart in memory.
call void @llvm.tensor.store(float* %mem_ptr, token %tensor, <3 x i32> <i32 0, i32 12, i32 10>)
```

llvm.tensor.matmul

Overview:

This intrinsic performs batched matrix multiplication between the inner dimensions of two multidimensional tensors. This intrinsic is marked with the `'readnone'` and `'speculatable'` attributes to prevent it from inhibiting optimizations like redundancy elimination, dead code elimination, code motion, etc.

```
<vector_ty> llvm.tensor.matmul(token %input1, token %input2)
```

Operands:

Operand	Description
%input1	Token value representing the first input tensor
%input2	Token value representing the second input tensor

Result:

Result	Description
%output	Output tensor expressed as a “flattened” LLVM vector

Semantics:

The ‘`llvm.tensor.matmul`’ intrinsic performs batched matrix multiplication between two input tensors. The inner two dimensions of the input tensors must have valid matrix multiplication dimensions, and any further outer dimensions must be of matching batch size. This intrinsic does not read nor write memory, nor does it exhibit any kind of undefined behavior.

Example:

```
%input1 = call token @llvm.tensor.typeinfo(<12 x float> %tensor1, <2 x
i32> <i32 3, i32 4>, <2 x i32> <i32 0, i32 1>, <2 x i32> <i32 0, i32
0>)
%input2 = call token @llvm.tensor.typeinfo(<12 x float> %tensor2, <2 x
i32> <i32 4, i32 3>, <2 x i32> <i32 0, i32 1>, <2 x i32> <i32 0, i32
0>)

%output = call <9 x float> @llvm.tensor.matmul(token %input1, token
%input2)

%typed_output = call token @llvm.tensor.typeinfo(<9 x float> %output,
<2 x i32> <i32 3, i32 3>, <2 x i32> <i32 0, i32 1>, <2 x i32> <i32 0,
i32 0>)
```

llvm.tensor.transpose

Overview:

This intrinsic changes the layout of a given tensor by permuting the indices of its dimensions. This intrinsic is marked with the **'readnone'** and **'speculatable'** attributes to prevent it from inhibiting optimizations like redundancy elimination, dead code elimination, code motion, etc.

```
<vector_ty> llvm.tensor.transpose(token %input, <n x i32> %new_layout)
```

Operands:

Operand	Description
%input	Token value representing the input tensor
%new_layout	This is the new permutation of tensor layout

Result:

Result	Description
%output	Output tensor expressed as a “flattened” LLVM vector

Semantics:

The '**llvm.tensor.transpose**' intrinsic operates on the given tensor and produces an output tensor with the given layout. This operation changes the physical layout of the given tensor and leads to changes in the tensor shape and padding. Note that operation does not lead to any change in the number of dimensions.
Note that this intrinsic does not read nor write memory, nor does it exhibit any kind of undefined behavior.

Example:

```
%input = call token @llvm.tensor.typeinfo(<240 x float> %tensor, <3 x
i32> <i32 16, i32 5, i32 3>, <3 x i32> <i32 0, i32 1, i32 2>, <3 x
i32> <i32 3, i32 2, i32 1>)
```

```
; The first argument is the input virtual tensor register,  and the second argument is the new permutation of the layout of the input tensor. This operation produces a tensor of layout  <2, 0, 1>.
%output = call <240 x float> @llvm.tensor.transpose(token %input, <3 x i32> <i32 2, i32 0, i32 1>)

%typed_output = call token @llvm.tensor.typeinfo(<240 x float> %output, <3 x i32> <i32 3, i32 16, i32 5>, <3 x i32> <i32 2, i32 0, i32 1>, <3 x i32> <i32 1, i32 3, i32 2>)
```

Design of Tensor Extensions in LLVM

Tensor extensions we have added to LLVM are described in the document here.

LLVM Tensor Intrinsics	Frontend Equivalent	Target Equivalent
llvm.tensor.matmul	XLA dot op	
llvm.tensor.contract	XLA dot general op	
llvm.tensor.umma		Intel AMX mma instruction Power MMA instruction
llvm.tensor.smma		Intel AMX mma instruction Power MMA instruction
llvm.tensor.usmma		Intel AMX mma instruction Power MMA instruction
llvm.tensor.summa		Intel AMX mma instruction Power MMA instruction
llvm.tensor.convolution	XLA convolution op	NVDLA convolution instruction
llvm.tensor.tanh	XLA element-wise op	NVDLA element-wise instruction
llvm.tensor.sigmoid		NVDLA element-wise instruction
llvm.tensor.relu		NVDLA element-wise instruction
llvm.tensor.broadcast	XLA broadcast op	Intel AMX fill instruction
llvm.tensor.load		Intel AMX load instruction

llvm.tensor.store		Intel AMX store instruction
llvm.tensor.reduce.max	XLA reduce window op	NVDLA pooling instruction
llvm.tensor.reduce.min	XLA reduce window op	NVDLA pooling instruction
llvm.tensor.reduce.add	XLA reduce window op	
llvm.tensor.reduce.mul	XLA reduce window op	
llvm.tensor.reduce.and	XLA reduce window op	
llvm.tensor.reduce.or	XLA reduce window op	
llvm.tensor.reduce.xor	XLA reduce window op	
llvm.tensor.reshape.block	OneDNN Layouts	
llvm.tensor.reshape.permute	Tensorflow reshape op	
llvm.tensor.transpose	Tensorflow transpose op	NVDLA reshape instruction
llvm.tensor.pad	XLA pad op	
llvm.tensor.concat	XLA concat op	NVDLA reshape instruction
llvm.tensor.tovector		Power unprime instruction
llvm.vector.totensor		Power prime instruction

Compatibility with and Benefits over Matrix Extensions

The existing matrix extensions model vectors as matrices in LLVM can co-exist and can be used with the tensor extensions that we propose. We argue that our tensor extensions provide an extensible and flexible long-term solution that LLVM developers can experiment with and adopt overtime. We believe that our tensor extensions provide the following benefits over the existing matrix extensions:

- Our tensor extensions support an arbitrary number of dimensions for tensors. This affords LLVM developers the flexibility to use higher-dimensional tensors as opposed to confining to rigidly supporting two dimensional tensors only. This support for generality also makes the tensor extensions more easy to maintain in the future.
- Currently, information about matrix shapes and layouts is encoded within the matrix intrinsics in LLVM. They do not provide a separation between the matrix properties and matrix operations. This makes the existing matrix extensions rigid and difficult to extend

in the future because if developers decide to encode more matrix properties in the IR, they would have to modify all matrix intrinsics and modify several lines of code using these matrix extensions. Our tensor extensions provide a separation between the tensor concept and tensor operations, thereby providing the flexibility of extending the tensor properties represented in the IR without having to modify all the tensor operations that operate on tensors. Note that this flexibility also allows supporting new kinds of tensors (such as sparse and ragged tensors) more easily in the future as well.

- Matrix padding is modelled using vector shuffle instructions. This requires optimizations, analyses and transformations to infer padding information by carefully inspecting all the shuffle instructions and their masks. We encode tensor padding information as a set of tensor properties directly represented and readily available in the IR and we use an intrinsic to represent a padding operation.

Methodology to Extend TLX

Decoupling tensor type information and intrinsics for tensor operations, allows extending the tensor type information, if needed, without having to make any changes to other intrinsics for tensor operations. If sparse tensors need to be supported, a new variant of typeinfo intrinsic with information relevant to sparse tensors could be introduced while continuing using the intrinsics for tensor operations.
Note that if the typeinfo intrinsic described in this document needs to be modified, it can be modified without needing to modify other intrinsics for tensor operations.

Current Status of the Implementation

- Lowering of most high-level tensor operations to LLVM scalar and vector instructions is supported.
- Tensor code generation framework is capable of targeting Intel AMX. Support for targeting NVDLA and NVIDIA tensor cores is in progress.
- Lowering support to target Intel VNNI and Hexagon Vector Extension (HVX) is underway.
- Example of lowering from Julia to the proposed tensor extensions is in the design document.

Open Questions

- How can the Relu class of operations be supported in a general way in LLVM? Is there a need/desire for these operations to be predicated?
- How can vendors add custom types without making intrusive changes to LLVM's type system and LLVM files?