Background on Tensor Contraction Kernels in Uni20

Uni20 Developer Documentation

November 4, 2025

1 Introduction

Tensor contraction is the multilinear generalisation of matrix multiplication. Uni20 exposes contraction kernels that operate on tensor operands described by `mdspan`-compatible interfaces. This note collects the background theory and implementation strategy used by Uni20's current kernels so that new contributors can reason about correctness and performance trade-offs. We focus on the CPU and BLAS backends, where the core strategy is to decompose an arbitrary contraction into groups of indices that resemble the (M, N, K) structure of a GEMM operation.

2 Terminology and Layout Conventions

The C++ `mdspan` reference implementation distinguishes between `layout_right`—where the right-most index varies fastest and therefore has stride 1—and `layout_left`, whose left-most index is contiguous. This is analagous to *row-major* and *column-major* layouts for matrices, but it is confusing to refer to higher order tensors as row- or column-major, hence we use the right or left layout terminology. There are several ways to remember this convention: `layout_left` means that the smallest stride (often but not necessarily stride 1) is the left-most index. It also means that the strides increase from left-to-right. Conversely, for `layout_right`, the right-most stride is the smallest, and the strides increase from right to left.

Fortran traditionally uses `layout_left` (column-major), while other languages typically use `layout_right` (row-major). Uni20 is agnostic to which style is used – in algorithms often *left* layout is more natural since we often want to look at the lowest stride, which will be the first entry in the array rather than the last, and we can arrange the indices in left layout by ordering the strides from smallest to largest.

Throughout the remainder of this document we use *merging* as the canonical term for combining adjacent stride descriptors whose outer stride equals the inner stride multiplied by the extent. This replaces the earlier "coalescing" wording and aligns the prose with the `merge_strides_*` helper names.

3 From `mdspan` to Concrete Tensors

Uni20 adopts the C++ reference implementation of `mdspan`, placed in the `stdex` namespace, as `stdex::mdspan`. The canonical way to include the `mdspan` library is to include `src/common/mdspan.hpp`, although arguably that should move to `src/mdspan/mdspan.hpp` (but this might introduce some ambiguity between `include <mdspan/mdspan.hpp>` for the reference version, versus `include "mdspan/mdspan.hpp"` for the Uni20 header).

An `mdspan` couples a raw pointer ("data handle") with an *extents* object, a layout mapping, and an accessor. Within Uni20 we wrap that interface in convenience aliases and concepts (see `src/common/mdspan.hpp` and `src/mdspan/concepts.hpp`). Any type that models the `StridedMdspan` concept exposes the rank, extents, and per-index strides required to navigate strided storage. The kernel front-end therefore accepts operands as `StridedMdspan` instances, enabling it to operate on plain `mdspan`, custom views, and tensor wrappers alike.

A concrete owning tensor pairs storage with such an `mdspan`. The class template `Tensor<...>` stores an owning buffer and an `mdspan` view over that buffer (Fig. 1). The kernel path only interacts with the non-owning view: when a tensor is passed to a contraction routine we immediately retrieve its `mdspan` via `view()` and use `data_handle()`, extents, and strides to drive the computation. This makes the implementation agnostic to how memory is allocated while keeping the backend code focused on pointer arithmetic.

```
class Tensor {
  using mdspan_type = stdex::mdspan<T, extents_type, layout_type,
     accessor_type>;
  std::vector<T> storage_;   // owning buffer
  mdspan_type    view_;      // mdspan over that buffer
};
```

Figure 1: A Uni20 tensor owns storage but exposes an `mdspan` view. Backend kernels consume the `mdspan` interface only.

4 Stride Groups

Consider a contraction of rank-A tensor A and rank-B tensor B over N contracted index pairs. We reorganise the indices into three groups, each represented by an array of `extent_strides` structures:

- The *M-group* stores all uncontracted indices sourced from A (which also appear in the output tensor C).
- The *N-group* stores all uncontracted indices sourced from B (also present in C).
- The *K-group* contains the contracted index pairs between A and B.

Each descriptor captures the extent of the grouped dimension and the strides in both participating operands. The helper `extract_strides()` walks the contracted index pairs, checks that extents agree, and fills the K-group with matching stride metadata. It then visits the remaining indices of A and B, checking that they match the layout of C while filling the M- and N-groups respectively. Finally, each group is merged: adjacent descriptors combine when both operands present mergeable strides (the outer stride equals the inner stride scaled by the extent), effectively fusing neighbour indices that already form a single logical block. We then call `merge_strides_right()` on each group, giving a compact representation of the contraction that resembles GEMM's (M, N, K) tuple. (The choice of `merge_strides_right()` here means that unmergable indices will be in the layout right order for the *first* set of strides in the group. This is a rather arbitrary choice, that we will probably want to change later.)

The descriptors themselves are instances of `extent_strides<2>` from `src/mdspan/strides.hpp`. Each record stores a common `extent` alongside a two-element `strides` array capturing the contri-

bution of the dimension in the A and B operands. Conceptually, the structure follows the following sketch:

```
struct extent_strides<2> {
  index_type extent;                        // logical length of the merged
      axis
  std::array<std::ptrdiff_t,2> strides; // the corresponding strides of
      the two groups
};
```

One way to visualise the metadata is to compare a logical axis before and after merging. The table below shows how two adjacent descriptors representing axes i and j fuse into a single descriptor when both operands present compatible strides:

State	**Extent**	**Strides** (A, B)
Before merge	(e_i, e_j)	$((s_i^A, s_i^B), (s_j^A, s_j^B))$
After merge	$e_i e_j$	(s_i^A, s_i^B)

Here the stride pair (s_i^A, s_i^B) is reused for the merged axis because the merge predicate requires $s_j^A = e_i \cdot s_i^A$ and $s_j^B = e_i \cdot s_i^B$.

5 CPU Reference Kernel

The CPU reference backend executes the grouped contraction via recursive loops. The M-group is iterated outer-most, the N-group governs the middle loops, and the K-group contributes to the innermost dot product. Each recursion level adjusts the active pointers by the precomputed strides so that no additional index arithmetic is required. This implementation works for arbitrary strided tensors that satisfy the `StridedMdspan` concept, even if they are not in a form that can readily be contracted using `GEMM` or other kernels.

6 Delegating to BLAS

When the grouped representation aligns with the requirements of a GEMM, the contraction can be mapped to a single BLAS call:

$$C_{MN} \leftarrow \beta C_{MN} + \alpha A_{MK} B_{KN}. \tag{1}$$

A BLAS GEMM operates on column-major matrices with the following key properties:

- Each operand must present at least one axis with unit stride. The `TRANS` parameter selects whether that axis corresponds to rows or columns, effectively allowing either row-major (*right*) or column-major (*left*) interpretation for each input matrix.
- The leading dimensions (strides between consecutive columns/rows) may exceed the logical extents, allowing batched or padded storage.
- The output matrix C likewise requires a unit-stride axis, but by choosing the transpose flags consistently we can always align the stored unit stride with BLAS's expectation. When C itself is row-major we can compute $C^\top = B^\top A^\top$ instead and transpose the interpretation of every operand simultaneously.

Uni20's stride descriptors already capture both extents and strides. A direct GEMM is therefore possible when:

1. Each of the M-, N-, and K-groups collapses to a single descriptor so that every operand presents exactly two logical indices.

2. For A either the M- or the K-descriptor has stride 1. The transpose flag determines which logical axis BLAS treats as contiguous. If the K-descriptor is stride 1, it is `'N'` (Normal) mode. Otherwise, if the M-group descriptor is stride 1, it is `'T'` (Transpose).

3. For B either the K- or the N-descriptor has stride 1, again selecting the appropriate transpose mode. If the N-descriptor is stride 1, it is `'N'` (Normal) mode. Otherwise, if the K-group descriptor is stride 1, it is `'T'` (Transpose).

4. For C either the M- or the N-descriptor has stride 1. If the M-descriptor of C has stride 1, then we can call BLAS directly with the transpose modes for A and B as above. If the N-desriptor of C has stride 1, then we need to instead interpret the operation as $C^\top = B^\top A^\top$, effectively interchanging MN, and reinterpret the transpose flags for A and B.

Because BLAS exposes two transpose choices per operand there are $2 \times 2 = 4$ combinations for the inputs and, once the result orientation is taken into account, eight total possibilities when deciding which physical axis supplies the unit stride. When these criteria hold, the grouped contraction represents a simple matrix multiplication with matrices laid out in memory exactly as BLAS expects. Uni20 can then forward the contraction to the BLAS backend, providing extents as matrix sizes and the outer strides as leading dimensions.

7 Rearrangement Strategies

Many practical contractions violate at least one of the BLAS-friendly criteria. The two common issues are:

1. **Grouped indices that are not contiguous.** After merging we may still retain multiple descriptors per group because strides disagree. For instance, an M-group may contain two indices where the stride of the inner index is not the extent-scaled stride of the outer index. This prevents the fused dimension from being interpreted as a single matrix axis.

2. **Operand has no stride-1 index.** If a tensor has no stride-1 index (for example, it might be obtained by *slicing* a tensor, e.g. taking every second element of a tensor[1]) then that tensor is not compatible with BLAS.

To exploit BLAS in these scenarios we must rearrange storage so that each group behaves like a single contiguous axis. Uni20 can deploy several strategies:

Looped GEMM. If only one group fails to merge completely, we can treat the remaining dimensions as batch loops around a GEMM call. For example, suppose the M-group contains two descriptors but the N- and K-groups each reduce to one. We can iterate over the extra M-group descriptor, adjusting pointers by its stride, and issue a GEMM for every slice. This preserves GEMM's efficiency while avoiding a full rearrangement.

[1] A practical example of this is taking the real components of a complex tensor.

Rearrange → GEMM → Rearrange. When multiple groups have non-contiguous descriptors or when unit stride axes do not line up, we may materialise temporary buffers whose layout is explicitly column-major. The sequence proceeds as follows:

1. Copy or pack the relevant operand(s) into temporaries with contiguous grouped axes.
2. Call GEMM on the packed operands.
3. Unpack or scatter the result back to the original layout of C.

The packing step often leverages the same stride metadata to generate efficient nested loops or to drive vectorised copy kernels.

The choice between these approaches depends on the relative size of the non-contiguous groups, cache behaviour, and the overhead of packing. Ultimately we might want to implement some benchmarking tests to select at compile or runtime which approach to use. But we will want kernels that implement both approaches – the decision as to which one to use will be at a higher level.

Practical implementations of GEMM usually achieve their highest throughput when the contracting (K) group is contiguous in both operands. In column-major Fortran notation this corresponds to computing $C \leftarrow A^\top B$, so that the transpose flag exposes the K-group as the unit-stride axis in A while B already presents a unit stride along its rows. Or in modern notation, A is *layout right*, and B and C are *layout left*. Modern BLAS libraries pack input panels into tiled contiguous buffers, which reduces the penalty when one operand lacks stride-1 access; nevertheless, presenting contiguous memory along K maximizes the efficiencies. But this means that if we do need to copy the memory for any of the operands to rarrange the stride layouts, a rearrangement of A should be done so that it results in layout right, whereas a rearrangement of B is best done as layout left. If we need to rearrange C, then we have a choice, whether to use layout left, interpretting the operation as $C = AB$, or layout right, interpreting the operation as $C^\top = B^\top A^\top$; we can choose whichever version results in the most favorable layout of A and B.

Note that rearranging the memory layout of tensors has no connection to the logical ordering of indices. In Uni20, the order of indices is irrelevant, only the extents and the corresponding strides matter. Once the indices involved in a contraction are split into the M, N, K groups, the order of indices within each group can be changed freely.

8 Putting It All Together

A general tensor contraction can therefore follow one of several execution paths:

1. **Direct GEMM.** When every group collapses to a single contiguous descriptor with unit stride in the contracting operands, Uni20 maps the operation to a single BLAS GEMM.

2. **Batched GEMM.** When a subset of groups contains descriptors that could not be merged but the remaining ones do merge, Uni20 can run GEMM inside outer loops indexed by the residual descriptors.

3. **Pack–GEMM–Unpack.** When neither of the above is feasible, Uni20 can rearrange the operands and/or result into contiguous temporaries, dispatch GEMM, and scatter the result back.

4. **Reference Loop Kernel.** As a fallback or correctness reference, the CPU recursive kernel executes the contraction directly from the stride descriptors without relying on GEMM. This path might also be useful for layouts that cannot be cheaply rearranged. For very small tensors, the cost of rearranging is likely to be higher than just calling the reference kernel.

9 Worked Example

To illustrate the mechanics of stride grouping and rearrangement, consider the contraction

$$C_{aij} = \sum_b A_{abi} B_{bj} \tag{2}$$

with the following layouts (extents and per-index strides):

$$\begin{aligned} C &\in \mathbb{R}^{3\times4\times5}, && \text{strides} = (20, 5, 1), \\ A &\in \mathbb{R}^{3\times7\times4}, && \text{strides} = (28, 4, 1), \\ B &\in \mathbb{R}^{7\times5}, && \text{strides} = (5, 1). \end{aligned}$$

The index order in the explicit expression is not relevant; only the mapping between indices and tensor layouts matters when extracting stride groups. For example, the permutation C_{iaj} with extents $(4, 3, 5)$ and strides $(5, 20, 1)$ is equivalent from the contraction engine's perspective.

Initial stride grouping. Running `extract_strides()` identifies two potential assignments for the M-group (the uncontracted indices sourced from A):

- $M = (a, i)$ with extents $(3, 4)$, strides $(28, 1)$ in A, and corresponding strides $(20, 5)$ in C.
- $M = (i, a)$ with extents $(4, 3)$, strides $(4, 28)$ in A, and strides $(5, 20)$ in C.

This group is not mergable. The strides in C are mergable in isolation (since the other stride is equal to the product of the inner stride and its extent), but the strides in A are not.

The K-group contains the single contracted index b with extent 7, stride 4 in A, and stride 5 in B. The N-group consists of the index j with extent 5, stride 1 in B, and stride 1 in C.

From the critera of when we can delegate to BLAS, from above we have 3 criteria:

1. *Each of the M-, N-, and K-groups collapses to a single descriptor*: **No**. The M group contains 2 descriptors.
2. *For A either the M- or the K-descriptor has stride* 1.: **No**. Neither the K or M group has stride 1 for the A tensor. (The stride 1 as part of the M group for A does not count here since there is more than one stride.)
3. *For B either the K- or the N-descriptor has stride* 1.: **Yes**. In this example, the N group for B has stride 1.
4. *For C either the M- or the N-descriptor has stride* 1.: **Yes**. The N group for C has stride 1.

We therefore need to examine the M group and find a way to merge these indices, and we also need to rearrange the memory for the A tensor so that either M or K groups for A has stride 1. The first step, examine how to merge the indices in the M group, reveals that the descriptors for C *are* mergable, hence we do not need to rearrange C. We only need to rearrange A, and ensure that we do it in such a way that we can merge the strides of A and C in the same way. Following the discussion above, when we rearrange the memory for A, we are best doing so such that the contraction stride (the K group) is stride 1.

Rearranging operand A. The kernel therefore decides to pack A into a temporary layout where the K-group is contiguous. One suitable rearrangement stores the indices in the order (b, i, a) in the left layout so that the packed tensor has strides $(1, 7, 28)$: the b index (contracting group) has unit stride, while (i, a) form the M-group with extents $(4, 3)$ and strides $(7, 28)$. After grouping into the M, N, K groups we have:

$$\begin{aligned} &M : \text{extent } (4,3), && \text{stride}_A = (7,28), && \text{stride}_C = (5,20), \\ &N : \text{extent } 5, && \text{stride}_B = 1, && \text{stride}_C = 1, \\ &K : \text{extent } 7, && \text{stride}_A = 1, && \text{stride}_B = 5, \end{aligned}$$

We can then merge the indices in the M group, to

$$M : \text{extent } 12, \qquad \text{stride}_A = 7, \qquad \text{stride}_C = 5,$$

which fits the GEMM interface with $C_{MN} = A_{MK}B_{KN}$. The K group is unit stride in A, but not in B, which means that the operation is effectively $C^\top = BA$, a $12 \times 7 \times 5$ multiply, so we call GEMM with B as the first operand with leading dimension 5, and A as the second operand with leading dimension 7, and C has leading dimension 5. Note that the leading dimensions coincide with the extents here, but that is not necessary: it would also work if the matrices are not contiguous: the leading dimensions arise from the remaining *stride*, not the *extent*. This example would also work, for example, if C had strides $(32, 8, 1)$.

Decision test. The rule of thumb is:

1. Check whether the K-group in C is mergeable (contiguous after merging strides). If it is, choose the permutation of packed strides in A that mirrors C's grouping so that no additional work on C is required.

2. If the K-group is not mergeable in C, C must be rearranged regardless. In that case the packed layout for A (and subsequently C) can be chosen freely because both operands will be copied into temporaries.

10 Future work

We now need to to turn this into a full algorithm. This requires:

1. Finalize the stride merging API. The question here is how to handle cases where one tensor has mergable strides but the other tensor in the group does not. We perhaps want to quickly check and see whether the first or second tensor has mergable indices, and then reorder the strides based on whether we want the unmergable group to be layout left or layout right.

2. Implement the `rearrange` function, to copy a tensor to different strides. This could optionally use the HPTT library, although that library uses a permutation-based API rather than stride-based. So in the first instance, we should write our own `rearrange` function, since we need this as a fallback anyway. This function would look something like

```
void rearrange(extents_type Extents, stride_type InputStrides,
   stride_type OutputStrides, T const* In, T* Out)
```

We could re-use the `extent_strides` structure, which means we can use the existing functions to merge indices if possible. We probably don't want to use `extent_strides` in the API interface though, since we won't naturally have the strides in that structure yet. But maybe...

3. Implement the pack–GEMM–unpack function, with unit tests for each possible case. There are many different possibilties that requre rearranging one or more tensors, so the tests should be careful to cover all possibilities.

4. Write some benchmark comparisons of the GEMM kernel versus the reference loop kernel.

As additions/extensions, we could consider:

1. Implement the loop-over-GEMM algorithm, with some benchmarks to see if/when this is more efficient than pack–GEMM–unpack.

2. Implement batched GEMM, for BLAS libraries that offer this as an extension (eg MKL). Probably the only one worth implementing is a strided batched gemm, which allows for the case where either the M, N (or both?) groups have more than 1 index.