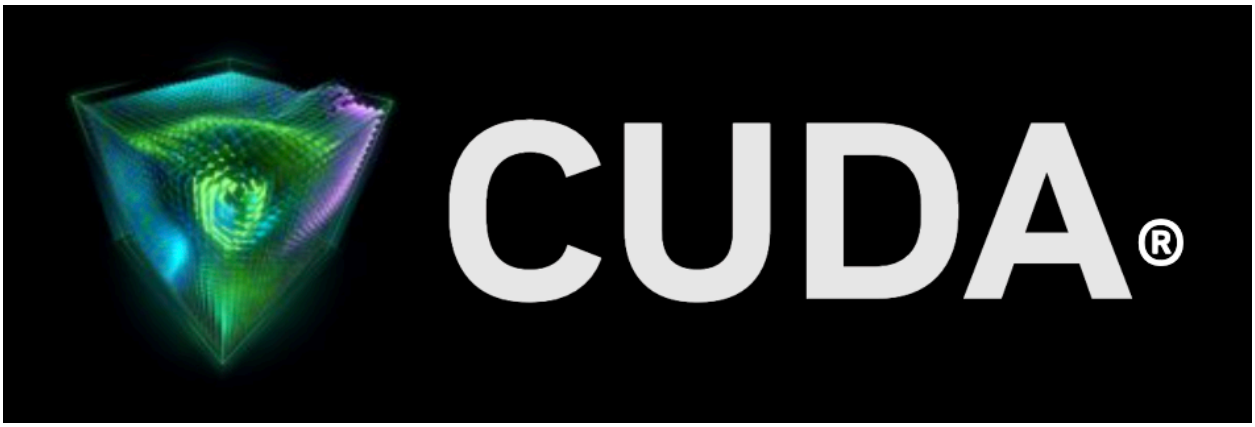

CUDA Tile C++ API Reference

Release 13.3

NVIDIA Corporation

May 21, 2026

Contents:

CUDA Tile C++ is a tile programming extension to the CUDA C++ language. In Tile C++, intra-block parallelism is expressed through elementwise operations on **tiles**. The compiler automatically parallelizes tile code across multiple threads while utilizing advanced hardware capabilities such as Tensor Memory Accelerators (TMA) and Tensor Cores. By leveraging the high level abstractions provided by Tile C++, a tile kernel can remain performance portable across NVIDIA GPU architectures without modification.

```
#include "cuda_tile.h"

__tile_global__ void vectorAdd(float* __restrict__ a, float* __restrict__ b, float* __
↪restrict__ out, std::size_t n) {
  namespace ct = cuda::tiles;
  using namespace ct::literals;

  a = assume_aligned(a, 16_ic);
  b = assume_aligned(b, 16_ic);
  out = assume_aligned(out, 16_ic);

  auto idx = ct::bid().x;

  auto view_a = ct::partition_view{ct::tensor_span{a, ct::extents{n}}, ct::shape{1024_
↪ic}};
  auto view_b = ct::partition_view{ct::tensor_span{b, ct::extents{n}}, ct::shape{1024_
↪ic}};
  auto view_out = ct::partition_view{ct::tensor_span{out, ct::extents{n}}, ct::shape
↪{1024_ic}};

  auto tile_a = view_a.load_masked(idx);
  auto tile_b = view_b.load_masked(idx);

  auto tile_out = tile_a + tile_b;

  view_out.store_masked(tile_out, idx);
}
```

This reference manual describes the foundational APIs for working with CUDA Tile C++. For a tutorial on the tile language extensions and tile programming paradigm, see the CUDA Programming Guide.

Chapter 1. General Principles

1.1. Header and Namespace

The Tile C++ APIs are available in the `cuda_tile.h` header. This header does not depend on the C++ standard library headers or on other CUDA Toolkit headers.

Unless otherwise specified, the Tile C++ APIs inhabit the `cuda::tiles` namespace. The `cuda::tiles` qualification may be abbreviated as `ct` in examples and documentation.

Example

Basic kernel returning the sum of two vectors using the `cuda_tile.h` header and `cuda::tiles` namespace.

```
#include "cuda_tile.h"

__tile_global__ void kernel(int* a, int* b, int* c) {
  namespace ct = ::cuda::tiles;

  auto idx = ct::bid().x;

  c[idx] = ct::add(a[idx], b[idx]);
}
```

The NVCC command line should target `sm_80` or higher and should include the `-std=c++20` and `--enable-tile` flags:

```
nvcc --enable-tile -std=c++20 -arch sm_80 main.cu
```

1.2. Language and Architecture Support

Use the `--enable-tile` flag to enable support for tile programming in NVCC and NVRTC. While tile kernels can be written with any C++ language standard, the CUDA Tile C++ APIs in `cuda_tile.h` including the `ct::tile` type require C++20 or higher.

The minimum supported target architecture for tile programming is `sm_80`. Tile code generation is implicitly disabled for earlier targets.

Note: NVCC targets `sm_75` as the default architecture if no architecture is provided. In this configuration, no tile code will be generated and a tile kernel launch will fail at runtime.

1.3. API and ABI Stability

The initial release of CUDA Tile C++ is CUDA 13.3.

Idiomatic usage of the Tile C++ APIs will not be broken between minor CUDA Toolkit versions. The APIs may change between major CUDA Toolkit versions. Certain non-idiomatic uses of the APIs could be broken between minor releases. These scenarios are noted in the API reference where relevant.

The ABI of any type under the `cuda::tiles` namespace will not change between minor versions of the CUDA Toolkit. The ABI of these types may change between major versions of the CUDA Toolkit. If an ABI change occurs, the ABI version tag for the inline namespace containing the affected type will also be incremented. As a result, code using the new ABI may fail to link with host or device libraries expecting the old ABI. This ensures early detection of some ABI mismatch issues.

Only the APIs which are explicitly documented in this API reference have a stable interface. No guarantee is made about the behavior or availability of entities with a double underscore prefix or entities under the `cuda::tiles::detail` namespace.

1.4. Scalars

The *scalars* are fundamental data types that are natively supported by the tile programming model. The Tile C++ APIs operate either on scalars directly or on dense arrays of scalars known as *tiles*.

The following types are known as *scalars*:[1]

Integral Scalars

An *integral scalar* is a possibly cv-qualified integral[5] type T whose bit size `CHAR_BIT * sizeof(T)` is 8, 16, 32, or 64.

Example

All of the following types are *integral scalars* on every platform supported by NVCC:

- `char`, `unsigned int`, `long long`, `char32_t`, `wchar_t`, `bool`

The extended integer type `int128_t` which is available on some platforms is not an *integral scalar* because its bit size is greater than 64.

The type `std::byte` is not an *integral scalar* because enumerations are not integral[5].

Basic Floating Point Scalars

The following types and cv-qualified variants thereof are known as *basic floating point scalars*:

[1] Within this document the term *scalar* and *scalar type* shall refer to this definition and not the term of the same name in the C++ standard.

[5] See § 6.8.2.11 [basic.fundamental] of ISO/IEC 14882:2024

`double`	Models the IEEE 754[2] 64-bit type.
`float`	Models the IEEE 754[Page 5, 2] 32-bit type.
__half	Models the IEEE 754[Page 5, 2] 16-bit type. Available in the `cuda_fp16.h` header.
__nv_bfloat16	Models the bfloat16 floating point format. In this format, 1 bit is stored for the sign, 8 bits for the exponent, and 7 bits for the mantissa. Available in the `cuda_bf16.h` header.

The basic floating point types may be used in the majority of Tile C++ arithmetic and math APIs.

The __nv_bfloat16 is considered to have appropriate IEEE 754 semantics for the purposes of defining the arithmetic and math APIs and associated rounding mode behaviors.

Note: The C++23 extended floating point types[6] are not considered to be *basic floating point scalars* even when provided by the platform. For example, none of the following types are *basic floating point scalars*:

- `std::bfloat16_t`, `std::float16_t`, `std::float32_t`, `std::float64_t`, `std::float128_t`

Restricted Floating Point Scalars

The following data types and their cv-qualified variants are known as *restricted floating point scalars*:

__nv_fp8_e4m3	Models a floating point format with 1 sign bit, 4 exponent bits and 3 mantissa bits. Available in the `cuda_fp8.h` header.
__nv_fp8_e5m2	Models a floating point format with 1 sign bit, 5 exponent bits and 2 mantissa bits. Available in the `cuda_fp8.h` header.
`__nv_tf32`	Models the tensor float 32 floating point format. This format has 1 sign bit, 8 exponent bits, and 10 or more mantissa bits. The total storage size is 32 bits and the alignment is 4 bytes. Available in the `cuda_tf32.h` header.

The restricted floating point types may be loaded and stored from memory and used in matrix multiplication functions, however their support in arithmetic and math operations is limited.

The __nv_fp8_e4m3 and __nv_fp8_e5m2 types are supported only for sm_90 and later architectures in tile code.

Pointer Scalars

The following types and their cv-qualified variants are known as *pointer scalars*:

1. Pointers to *numeric scalar* types
2. Pointers to possibly cv-qualified `void`

Example

Multi-level pointers and pointers to structs, arrays, unions, and functions are not considered to be *pointer scalars* and may not be used as tile elements.

[2] See IEEE 754-2019

[6] See § 6.8.2.12 [basic.fundamental] of ISO/IEC 14882:2024

Pointer Scalars	Not Pointer Scalars
`int* const`	`int**`
`double const*`	`int Foo::*`
`void*`	`void (*)(int, double)`
`__half volatile const*`	`Foo const*`

Floating Point Scalars
The *basic floating point scalars* and *restricted floating point scalars* are collectively known as *floating point scalar* types.

Arithmetic Scalars
The *integral scalars* and *basic floating point scalars* are collectively known as *arithmetic scalar* types. The arithmetic scalars can be used in most arithmetic APIs.

Numeric Scalars
The *integral scalars* and *floating point scalars* are collectively known as *numeric scalar* types.

1.4.1. Properties of Integral Scalars

Within this document, the *integral scalars* are understood to have the following properties which are useful when defining the arithmetic operations:

Numeric Value
The *numeric value* of an object of *integral scalar* type is the integer value used for performing arithmetic on that object. For all objects of integral type except `bool`, the numeric value is the value of the object. For objects of type `bool`, the numeric value is 0 if the object is `false` and 1 if the object is `true`.

Bitwidth
The *bitwidth* of an integral scalar type is the number of bits needed to encode the set of values representable by that type. For `bool`, the bitwidth is 1. For all other integral scalar types T, the bitwidth is `sizeof(T) * CHAR_BIT`.

Base Two Representation
The *base two representation* of an integer value x is the unique string of N bits $b_0 b_1 \dots b_{N-1}$ such that

$$x \bmod 2^N = \sum_{k=0}^{N-1} b_k \cdot 2^k$$

where each b_i has value 0 or 1 and N is the *bitwidth* of x.

Note: The arithmetic operations do not perform integral promotion, so it is necessary to define a numerical and bitwise representation for type `bool`.

1.5. Extents

The Tile C++ APIs use *ct::extents* to describe the shape of multi-dimensional arrays and tiles. A *ct::extents* object specifies a length for each dimension of the shape. Information about a dimension's length may be stored in the object at runtime or may be encoded in the object's type at compile time.

Example

Three extents objects each describing a shape of 4×8. The non-type template parameters of *ct::extents* determine which dimensions are stored at runtime.

```
namespace ct = ::cuda::tiles;

// All compile time dimensions
ct::extents<uint32_t, 4, 8> a{};

// One compile time, one runtime dimension
ct::extents<uint32_t, ct::dynamic_extent, 8> b{4};

// All runtime dimensions
ct::extents<uint32_t, ct::dynamic_extent, ct::dynamic_extent> c{4, 8};
```

When a dimension's length is stored at runtime, the first template parameter of *ct::extents* determines the integral type used for storing the dimension value. This type is called the *index type* and it is also used when performing computations involving the dimension's length.

Example

Extents objects with various index types.

```
namespace ct = ::cuda::tiles;

// Uses int16_t as the index type
ct::extents<int16_t, 3, ct::dynamic_extent> a{2};

// Uses uint64_t as the index type
ct::extents<uint64_t, ct::dynamic_extent, ct::dynamic_extent> b{4, 5};
```

Further information on the *ct::extents* type can be found in the *extents API reference*.

1.5.1. Extents Like

In addition to *ct::extents*, any type which satisfies the requirements of *extents like* can be used in Tile C++ APIs that expect a shape. This allows Tile C++ to interoperate with extents implementations that are provided by other C++ libraries.

Let E be a possibly cv-qualified type and e a glvalue expression of type `E const`. Let `dim` be a glvalue expression of type `E::rank_type const` when that is well-formed.

E is *extents like* if E models the standard library concept `std::copyable`[22] and the following expressions are well formed and fulfill the indicated requirements:

[22] See § 18.6 [concepts.object] of ISO/IEC 14882:2024

Expression	Requirements
`E::rank_type`	A cv-unqualified integral? type. **Semantics** Type used for identifying a dimension by its index.
`E::index_type`	A cv-unqualified integral? type. **Semantics** Type used to represent the length of a runtime dimension.
`E::rank()`	Static member function returning a prvalue of type `E::rank_type`. Must be a constant expression. **Semantics** Indicates the number of dimensions encoded in the extents type. The return value shall be non-negative and the expression must be equality-preserving.[23]
`E::rank_dynamic()`	Static member function returning a prvalue of type `E::rank_type`. Must be a constant expression. **Semantics** Indicates the number of dimensions which are known only at runtime. The expression shall be equality-preserving.[Page 10, 23] The return value must be exactly the number of dimensions whose length is as reported as *`ct::dynamic_extent`* by `E::static_extent(i)` for $0 \leq i <$ E::rank().
`E::static_extent(dim)`	Static member function returning a prvalue of type `size_t`. **Semantics** Yields the statically known length at dimension `dim`. If the length at `dim` is dynamic, yields *`ct::dynamic_extent`*. No requirements are placed on the behavior for values of `dim` that are not in the range $0 \leq dim <$ E::rank(). If `dim` is a constant expressions, the expression as a whole must be a constant expression. The expression must be equality-preserving.[Page 10, 23]
`e.extent(dim)`	Non-static member function returning a prvalue of type `E::index_type`. **Semantics** Yields the length at dimension `dim`. The expression must be equality-preserving.[Page 10, 23] The return value shall be non-negative. For each value $0 \leq dim <$ E::rank(), one of the following conditions holds: 1. `e.extent(dim)` == `E::static_extent(dim)`, or 2. `E::static_extent(dim)` == `ct::dynamic_extent` No requirements are placed on the behavior for values of `dim` that are not in the range $0 \leq dim <$ E::rank().

If E does not fulfill the semantic requirements above, any usage of E in a Tile C++ API results in undefined behavior.

Note: The `size_type` member alias is not required to be present. E need not be equality comparable or default constructible. Comparisons of extents are done via *`ct::extents_equal()`*.

1.5.2. Properties of Extents

This section describes properties which are common to all *extents like* types.

Let e be an object of *extents like* type E.

extents rank
The *rank* of E is given by `E::rank()` and indicates the number of dimensions encoded in the extents.

Example

The rank of `ct::extents<uint32_t, 3, 4, 5>` is 3.

The rank of `ct::extents<uint32_t>` is 0. A rank 0 extents is considered to have exactly one element.

---

static dimension

dynamic dimension
A *static dimension* is a dimension of E which is known at compile time. A *dynamic dimension* is a dimension of E known only at runtime. A dimension dim is dynamic if `E::static_extent(dim)` yields *`ct::dynamic_extent`*. Otherwise it is static.

Example

In the type `ct::extents<uint32_t, 4, ct::dynamic_extent, 7>`, dimensions 0 and 2 are static and dimension 1 is dynamic.

---

dynamic rank
The *dynamic rank* of E is the number of dynamic dimensions of E. The dynamic rank is given by `E::dynamic_rank()`.

extents shape
The *shape* of e is described by the `E::static_extent` and `e.extent` member functions. When `E::static_extent(dim)` returns the special sentinel value *`ct::dynamic_extent`*, the length of dimension dim is known only at runtime and may be retrieved with `e.extent(dim)`. Otherwise, `E::static_extent(dim)` returns the statically known length of dimension dim.

The shape of e is the collection of lengths given by:

$$e.\text{extent}(0) \times e.\text{extent}(1) \times \cdots \times e.\text{extent}(E::\text{rank}() - 1)$$

[23] See § 18.2 [concepts.equality] of ISO/IEC 14882:2024

The notation e_i refers to the length of dimension i of object e. In the case where all the dimensions are statically known, the notation E_i may be used to refer to the statically known length of dimension i given the type E.

Example

The following objects all have shape $3 \times 4 \times 7$:

- `ct::extents<uint32_t, 3, 4, 7>{}`
- `ct::extents<uint32_t, 3, ct::dynamic_extent, 7>{4}`
- `ct::extents<uint32_t, ct::dynamic_extent, 4, ct::dynamic_extent>{3, 7}`

singleton dimension

A *singleton dimension* of e is a dimension of length *1*.

Example

In the following object, dimension 1 is singleton

- `ct::extents<uint64_t, 4, 1>{}`

extents size

The *size* of e is the product of its dimensions and represents the number of elements in a multi-dimensional array whose shape is e. If the rank of e is 0 its size is 1.

When an e has no dynamic dimensions, its size is a property of the type E and may be computed at compile time.

The *size* is understood to be a mathematical property of e; it is a well formed value even if its computation in finite precision would trigger integer overflow.

When not clear from context, the term *object size* shall refer to the byte size of the object e.

Example

The size of `ct::extents<uint32_t, 4, 8>` is 32.

The size of `ct::extents<uint32_t, 5>` is 5.

The size of `ct::extents<uint32>` is 1.

extent equivalent

Two objects of *extents like* type are said to be *extent equivalent* if they have the same *rank* and their corresponding extent values are equal, regardless of their index types, rank types, or whether a given dimension is static or dynamic.

Example

The following two objects are *extent equivalent* despite having different types:

- `ct::extents<int16_t, 4, 8>{}`
- `ct::extents<uint32_t, ct::dynamic_extent, 8>{4}`

extents index space

Given N as the *rank* of e, the *index space* of e is the set of integer N-tuples that form valid indices into a multi-dimensional array described by e:

$$[0, e_0) \times [0, e_1) \times \cdots \times [0, e_{N-1})$$

In the case where E has only statically known dimensions, the *index space* of E refers to the index space of any instance of E.

Example:

The index space of the object `ct::extents<2, ct::rank_dynamic>{3}` includes the following indices:

- $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$, $(1, 2)$

---

shape like

The type E is said to be *shape like* if all of its dimensions are statically known.

For convenience, the alias template `ct::shape` can be used to specify a `ct::extents` with a `uint32_t` index type and all static dimensions.

Example

The following expressions produce the same type:

- `ct::shape<4, 5>`
- `ct::extents<uint32_t, 4, 5>`

---

shape equivalent

Two *shape like* types T and U are said to be *shape equivalent* if every instance a of T is *extent equivalent* to every instance b of U.

Example

The following two types are *shape equivalent*:

- `ct::extents<uint32_t, 4, 8>`
- `ct::extents<uint64_t, 4, 8>`

1.6. Tiles

The fundamental unit of tile programming is the `ct::tile<E, S>` data type. This type represents an immutable multi-dimensional array of *scalar* elements with a compile-time known shape. The template parameter E specifies the element type and while S specifies a *shape like* `ct::extents` describing the tile shape.

Example

A 4×8 tile object with `int` elements. Note that `ct::shape` is an alias for `ct::extents` with a `uint32_t` index type.

```
namespace ct = ::cuda::tiles;

ct::tile<int, ct::shape<4, 8>> obj;
```

For additional information about the *ct::tile* type, see the *tile type reference*.

1.6.1. Tile Like Types

The *scalar* types and the possibly cv-qualified specializations of *ct::tile* are together known as *tile like types*. These types implement a common abstraction that can be handled uniformly in Tile APIs.

The Tile C++ APIs generally treat *scalars* the same way they would handle rank 0 tiles. For example, the types `int` and `ct::tile<int, ct::shape<>>` behave similarly in Tile C++.

Example

The following types are all *tile like* types:

1. `double`
2. `int const`
3. `ct::tile<double, ct::shape<>>`
4. `ct::tile<float, ct::shape<4, 8>>`
5. `ct::tile<float, ct::shape<1, 1>> const volatile`

1.6.2. Properties of Tiles

This section describes the properties common to all *tile like* types. Let T be a *tile like* type.

notation

Let a be an object of *tile like type* T with *rank* N.

The notation T_i indicates the length of dimension i in the *shape* of T.

For an index $I = (i_0, i_1, \ldots, i_{N-1})$ in the *index space* of T, the notation $a(i_0, i_1, ..., i_{N-1})$ and $a(I)$ denote the element of a at index I.

tile element type

The *element type* of T is

- *remove-cv-t*<T> if T is a *scalar*.
- `T::element_type` if T is a specialization of *ct::tile*.

The element type of T is always a cv-unqualified *scalar* type.

Example

The *element type* of `ct::tile<double, ct::shape<4>>` is `double`.

The *element type* of `int` is `int`.

tile compatible shape

A *shape like* type S with rank N is said to be *tile compatible* if all the following hold:

1. `S::index_type` is `uint32_t`
2. Each dimension length S_i is a power of 2 for $0 \leq i < N$.
3. Each dimension length S_i does not exceed an implementation defined limit for $0 \leq i < N$.
4. The *size* of S does not exceed an implementation defined limit.
5. S is a specialization of *`ct::extents`*.

Example

`ct::shape<4, 7>` is not *tile compatible* because 7 is not a power of two.

`ct::shape<0>` is not *tile compatible* because 0 is not a power of two.

`ct::extents<int16_t, 4, 8>` is not *tile compatible* because its index type is not `uint32_t`.

`ct::shape<>` *is tile compatible*.

---

tile shape

The *shape type* of T is

- `ct::extents<uint32_t>` if T is a *scalar*.
- `T::shape_type` if T is a specialization of *`ct::tile`*.

The shape type of T is always *tile compatible*.

Example

The *shape type* of `ct::tile<int, ct::shape<4, 8>>` is `ct::shape<4, 8>`.

The *shape type* of `int` is `ct::shape<>`.

---

tile rank

tile size

tile index space

The *rank*, *size*, and *index space* of T refer to the *rank*, *size*, and *index space* of the *shape type* of T.

row major arrangement

Let a be an instance of T and N be its *rank*.

The *row major arrangement* of a is a sequence of elements s such that the following statement holds for any index $I = (i_0, i_1, ...i_{N-1})$ in the *index space* of a:

$$a(I) = s(i_0 \cdot (T_1 \cdot \ldots \cdot T_{N-1}) + i_1 \cdot (T_2 \cdot \ldots \cdot T_{N-1}) + \cdots + i_{N-1})$$

Equivalently:

$$a(I) = \sum_{j=0}^{N-1} i_j \cdot \prod_{k=j+1}^{N-1} T_k$$

Example

Consider a 4×2 tile `ct::tile<int, ct::shape<4, 2>>` representing the elements:

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix}$$

Its row major arrangement is the sequence

$$(0, 1, 2, 3, 4, 5, 6, 7)$$

---

singleton tile

If T has exactly one element it is known as a *singleton tile*.

Example

All the following types are *singleton tiles*:

- `double`
- `ct::tile<double, ct::shape<>>`
- `ct::tile<double, ct::shape<1, 1>>`

A specialization of `ct::tile` containing exactly one element is not a *scalar* but it is a *singleton tile*.

---

integral tile

basic floating point tile

restricted floating point tile

floating point tile

arithmetic tile

numeric tile

pointer tile

These terms indicate a *tile like type* whose *element type* is a *scalar* of the appropriate category.

Example

The following types are *integral tiles*:

- `int`
- `ct::tile<int, ct::shape<4>>`

The following types are *pointer tiles*:

- `double*`
- `ct::tile<double*, ct::shape<8>>`

elementwise operation
Consider a set of k *tile like* operands $a_1, a_2, \ldots, a_k$ of the same type T. An *elementwise operation* is the application of some operator $\text{op}(e_1, e_2, \ldots, e_k)$ to the corresponding elements of each operand to form a new result of type T. The result r is defined as

$$r(I) = \text{op}(a_1(I), a_2(I), \ldots, a_k(I))$$

for each index I in the *index space* of T.

1.7. Conversions

Many Tile C++ APIs convert between *tile like types* before performing a computation. These conversions are broadly categorized into *shape conversions*, *element type conversions*, and *arithmetic conversions*.

1.7.1. Extended Floating Point Types

For the purposes of defining the conversions of this section, the following *floating point scalar* types are considered to be extended floating point types[7]:

- __nv_fp8_e4m3
- __nv_fp8_e5m2
- __half
- __nv_bfloat16
- `__nv_tf32`

The floating point conversion ranks of these types form a partial order such that the each type on the left side of the following table has a lesser conversion rank than the corresponding type on the right:

Lesser Rank	Greater Rank
__nv_fp8_e4m3	__half
__nv_fp8_e4m3	__nv_bfloat16
__nv_fp8_e5m2	__half
__nv_fp8_e5m2	__nv_bfloat16
__half	`__nv_tf32`
__nv_bfloat16	`__nv_tf32`
`__nv_tf32`	`float`
`float`	`double`

Of the floating point types which are supported as *scalars*, no two types have the same floating point conversion rank and no subrank ordering is defined.

Example

The conversion ranks of __nv_fp8_e4m3 and __nv_fp8_e5m2 are unordered with respect to each other.

The conversion ranks of __half and __nv_bfloat16 are unordered with respect to each other.

The C++23 extended floating point types (for example `std::float16_t`) are not supported as *scalars* and no conversion rank or subrank for them is defined with respect to the *floating point scalar* types.

1.7.2. Element Type Conversions

scalar conversion

The *scalar conversions* are a set of behaviors for converting an object of one *scalar* type to another *scalar* type. A scalar conversion from an object of type T to an object of type U is a standard conversion sequence[7] which converts T to U, except that the following additional behaviors are specified:

1. Any *floating point scalar* type may be converted to any other *floating point scalar* type as part of the standard floating-point conversions[8]. The conversion exists regardless of whether the source and destination types are standard or extended floating point types.

   Note: This rule relaxes the floating-point conversions[8] by allowing conversions to and from extended floating point types of lower or unordered conversion rank.

   A conversion from a *floating point scalar* to a *floating point scalar* of a lower or unordered conversion rank is still considered to be a narrowing conversion.

   Example

   A scalar conversion exists from `double` to `__half`.

2. As part of the floating-point conversions[8], the *convertFormat*[24] operation with the *roundTiesToEven*[3] rounding mode is used to convert the source to the destination, except that:

   1. If the target type is `__nv_fp8_e5m2` or `__nv_fp8_e4m3` and the source value is non-finite or lies outside the representable finite range of the target type, the result value is unspecified.

   Note: This rule specifies an implementation defined behavior of the standard floating-point conversions[?].

3. As part of the floating-integral conversions[9], if the destination is a *floating point scalar* the

[7] See § 7.3.1.1 [conv.general] of ISO/IEC 14882:2024
[8] The standard floating-point conversions are defined in § 7.3.10 [conv.double] of ISO/IEC 14882:2024
[24] See § 5.4.2 of IEEE 754-2019
[3] See § 4.3 of IEEE 754-2019
[9] The standard floating-integral conversions are defined in § 7.3.11 [conv.fpint] of ISO/IEC 14882:2024

convertToIntegerTiesToEven[25] operation is used to convert the *numeric value* of the source to the destination.

If the source integer value is outside the representable finite range of the destination type, then

1. If the target type is `__nv_fp8_e5m2` or `__nv_fp8_e4m3`, the result is unspecified.
2. Otherwise, the result is an infinity whose sign matches the sign of the (non-zero) source value.

Note: This rule specifies an implementation defined behavior of the standard floating-integral conversions[Page 17, 9] and removes undefined behavior for out of range source values.

For the purposes of this conversion, the *restricted floating point scalars* and `__nv_bfloat16` are considered to be IEEE 754 arithmetic formats of appropriate range and precision.

Note: The precision of `__nv_tf32` is not fully specified, so conversions involving `__nv_tf32` are not fully specified.

If no standard conversion sequence[?] with the above modifications exists from an object of type T to the object of type U, no *scalar conversion* exists for those objects. If the standard conversion sequence from T to U would be narrowing, we say the scalar conversion from T to U is narrowing.

The above rules only affect certain tile C++ APIs that make use of the *scalar conversions*. The behavior of core C++ language semantics or builtin arithmetic operations is not affected by the above rules.

Example

There exists a *scalar conversion* from `__nv_bfloat16` to `__half`. The round nearest even rounding mode is used to select the result value. The conversion is narrowing because the conversion rank of `__nv_bfloat16` is unordered with respect to `__half`.

tile conversion

The *tile conversions* are a set of behaviors for converting an object a of *tile like type* T to a *tile like type* U. A tile conversion exists from T to U if there exists a *scalar conversion* from the *element type* of T to the *element type* of U and the *shapes* of T and U are *shape equivalent*.

The result of the conversion is a new object of type U formed by the *elementwise application* of *scalar conversion* to the *element type* of U.

A tile conversion is said to be narrowing if it invokes a narrowing *scalar conversion*.

Example

There exists a *tile conversion* from `int` to `ct::tile<int, ct::shape<>>`.

There does not exist a *tile conversion* from `int` to `ct::tile<int, ct::shape<1>>`.

bool tile conversion

The *bool tile conversion* of an object a of *tile like type* is the process of converting a to type `ct::tile_with_element_t<T, bool>` using *tile conversion*.

[25] See § 5.8 of IEEE 754-2019

Note: All *tile like* are bool tile convertible.

The result of bool tile conversion of a *scalar* is a *scalar*.

1.7.3. Shape Conversions

A tile may be converted to a tile of a different shape through *broadcasts*. A broadcast will replicate elements of the tile along a given dimension to expand the tile to the desired shape. There are two styles for this conversion:

1. One tile can be broadcasted to match a specific shape. This is know as *broadcast conversion*.
2. Two tiles can be broadcasted simultaneously so that their shapes match. This is called *mutual broadcast conversion*.

The following sections describe these conversions along with supporting definitions.

broadcastable to

Let S and B be two *shape like* types with ranks N and M respectively. We say that S is *broadcastable to* B if $N \leq M$ and for each dimension $0 \leq i < N$ one of the following conditions holds:

- $S_i = B_{i+M-N}$ or
- $S_i = 1$

For a *tile like* type T, we say that T is *broadcastable to* B if its *shape* is broadcastable to B and B is *tile compatible*.

Example

The shape on the left hand side of the table is *broadcastable to* the corresponding shape on the right:

Source	Broadcast Target
`ct::shape<4, 1>`	`ct::shape<4, 8>`
`ct::shape<2>`	`ct::shape<4, 2>`
`ct::shape<5, 2>`	`ct::shape<5, 2>`

Each tile like type on the left is broadcastable to the shape on the right:

Source	Broadcast Target
`double`	`ct::shape<4, 8>`
`ct::tile<float, ct::shape<2>>`	`ct::shape<8, 2>`
`ct::tile<float, ct::shape<2, 1, 8>>`	`ct::shape<2, 16, 8>`

broadcast conversion

Let a be an object of *tile like* type T with *shape* S and *rank* N. Let B be a *tile compatible shape* of rank $M \geq N$ such that T is *broadcastable to* B.

The *broadcast conversion* of a to shape B replicates the values of a along its *singleton dimensions* to match the shape B.

The broadcast conversion of a to B is a `ct::tile` object b whose *element type* matches that of a and whose shape is B. For each index $I = (i_0, i_1, \cdots, i_{M-1})$ in the *index space* of B,

$$b(i_0, i_1, \ldots, i_{M-1}) = a(f_0(i_{M-N}), f_1(i_{1+M-N}), \ldots, f_{N-1}(i_{M-1}))$$

For each source dimension $0 \leq k < N$, the function f_k is defined as

$$\begin{cases} f_k(i) = 0 & \text{if} S_k = 1 \\ f_k(i) = i & \text{otherwise} \end{cases}$$

The *broadcast conversion* of an object is always a specialization of `ct::tile` even if the source is a *scalar* type.

Example 1

Consider the following tile of type `ct::tile<int, ct::shape<4, 1>>`:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

The broadcast conversion of this object to the shape `ct::shape<4, 8>` is the tile object `ct::tile<int, ct::shape<4, 8>>`:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \end{pmatrix}$$

Example 2

Consider the following tile of type `ct::tile<int, ct::shape<2>>`:

$$\begin{pmatrix} 1 & 2 \end{pmatrix}$$

The broadcast conversion of this object to the shape `ct::shape<4, 2>` is the tile object `ct::tile<int, ct::shape<4, 2>>`:

$$\begin{pmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \\ 1 & 2 \end{pmatrix}$$

---

shape mutual broadcast compatible

Let T and U be two *shape like* types with ranks N and M respectively. Consider the common suffix of their dimensions formed by T_{N-1-i} and U_{M-1-i} for $0 \leq i < \min(N, M)$.

We say that T and U are *mutual broadcast compatible* if for each i, any of the following hold:

- $T_{N-1-i} = U_{M-1-i}$ or
- $T_{N-1-i} = 1$ or
- $U_{M-1-i} = 1$

Example

The following pairs of types are mutual broadcast compatible:

- `ct::shape<5, 1>` and `ct::shape<1, 3>`
- `ct::shape<2, 4, 1>` and `ct::shape<4, 3>`
- `ct::shape<>` and `ct::shape<3, 4, 5>`

The types `ct::shape<4, 2>` and `ct::shape<5, 2>` are not broadcast compatible because the first dimensions are non-singleton and do not match.

---

mutual broadcast shape

Let T and U be *broadcast compatible* with ranks N and M respectively. Without loss of generality, assume $N \leq M$. The *mutual broadcast shape* of T and U is a cv-unqualified specialization of `ct::extents` B satisfying the following conditions:

- The rank of B is M
- The index type of B is `uint32_t`
- $B_i = U_i$ for each i in $0 \leq i < M - N$
- $B_i = \max(T_{i-(M-N)}, U_i)$ for each i in $M - N \leq i < M$.

The broadcast type of T and U is a specialization of `ct::extents` even if neither T nor U are specializations of `ct::extents`.

Example

The third column in the following table is the *mutual broadcast shape* of the types in the first two columns:

Type 1	Type 2	Broadcast Type
`ct::shape<4, 1>`	`ct::shape<4, 8>`	`ct::shape<4, 8>`
`ct::shape<1, 5>`	`ct::shape<3, 1>`	`ct::shape<3, 5>`
`ct::shape<4, 2, 1>`	`ct::shape<2, 6>`	`ct::shape<4, 2, 6>`
`ct::shape<4>`	`ct::shape<1, 2, 1>`	`ct::shape<1, 2, 4>`

---

tile mutual broadcast compatible

Let T and U denote two *tile like* types. We say that T and U are *mutual broadcast compatible* if:

1. The *shapes* of T and U are *mutual broadcast compatible*, and

2. The resulting *mutual broadcast shape* is a *tile compatible shape*.

The *mutual broadcast shape* might not be *tile compatible* if it exceeds the implementation defined maximum size for a tile shape.

Example

The following types are all pairwise broadcast compatible

1. `ct::tile<int, ct::shape<2, 1, 8>>`
2. `ct::tile<int, ct::shape<1, 4, 1>>`
3. `ct::tile<float, ct::shape<4, 8>>`
4. `double`

---

mutual broadcast conversion

Let a and b be objects of *broadcast compatible tile like types* T and U respectively. The *mutual broadcast conversion* of a and b is a process for converting a and b to a common shape. It proceeds as follows:

1. Let B denote the *broadcast shape* for the shapes of T and U. Perform *broadcast conversion* to B on the objects a and b to yield two new objects of `ct::tile` type. For the remaining stage, let a and b denote the result of the preceding conversion.
2. If either T or U are specializations of `ct::tile`, the process is done and the converted results are a and b.
3. Otherwise, T and U are both *scalar* types and the result of the preceding step are two *singleton tiles*. Form two *scalar* objects by extracting the single element from a and b respectively. The two *scalar* objects are the result of the conversion.

Note: The result of *mutual broadcast conversion* between a scalar and a tile is a pair of tiles.

Example

Consider the following two tiles whose types are `ct::tile<int, ct::shape<1, 4>>` and `ct::tile<int, ct::shape<2, 1>>` respectively:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \qquad \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

The mutual broadcast conversion of these objects are two tiles of type `ct::tile<int, ct::shape<2, 4>>`:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \qquad \begin{pmatrix} 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 \end{pmatrix}$$

1.7.4. Arithmetic Conversions

The arithmetic conversions are used for converting among *arithmetic tiles* in binary operations. In general, an arithmetic conversion will first broadcast the operands to a common shape, then perform tile conversion to make a common element type. Unlike standard C++ arithmetic conversions, these arithmetic conversions do not perform integer promotion.

Two variants of arithmetic conversions are used depending on the operation. For most binary arithmetic operations, the data type of a tile operand is preferred over scalar operands. However for comparison operations, whether or not the operands are tiles or scalars is not considered when determining the common data type to convert to.

The following section describes the behavior of arithmetic conversions.

arithmetic common type

Let T and U be *arithmetic scalar* types.

The *arithmetic common type* of T and U is a type C to which both types may be *scalar converted*. This conversion roughly corresponds to the usual C++ arithmetic conversions without integer promotion behavior.

For the purposes of determining C, the non-standard *floating point scalar* types are considered to be extended floating point types of appropriate conversion ranks as described in *extended floating point types*.

C is determined as follows:

1. If either T or U are *floating point scalar*, C is the common type determined by the C++ usual arithmetic conversions[10]. If C could not be determined according to these rules, the *arithmetic common type* does not exist.
2. Otherwise, both T and U are *integral scalar* types. If T and U are the same type, C is that type.
3. Otherwise, if one of T or U is signed and the other is unsigned, the following rules apply. Let S be the signed type and U the unsigned type:
    1. If the conversion rank of U is greater than the conversion rank of S, C is U
    2. Otherwise, if S can represent all values of U, C is S.
    3. Otherwise, C is the unsigned type corresponding to S.
4. Otherwise, T and U are have the same signedness. If one has a greater conversion rank than the other, C is the type of the greater rank.
5. Otherwise, T and U are distinct integral types of the same signedness and conversion rank. C is the type with the greater conversion subrank defined as follows:
    1. The subranks of `char`, `char8_t`, `char16_t`, `char32_t`, and `wchar_t` are each less than the subrank of their corresponding underlying integer[26] type.
    2. If `char` has the same underlying type as any of `char8_t`, `char16_t`, `char32_t`, or `wchar_t`, the subrank of `char` is less than the subrank of the other type.

Example

The table below shows examples of the *arithmetic common type* on a typical target system.

[10] See § 7.4 of [expr.arith.conv] ISO/IEC 14882:2024
[26] See § 6.8.2 [basic.fundamental] of ISO/IEC 14882:2024

Type 1	Type 2	Common Type
`int`	`double`	`double`
`__half`	`float`	`float`
`__half`	`__nv_bfloat16`	*<illformed>*
`short`	`short`	`short`
`char16_t`	`unsigned short`	`unsigned short`

arithmetic comparison conversion

arithmetic tile conversion

The *arithmetic tile conversion* and *arithmetic comparison conversion* receive two *tile like* operands and convert them to a common element type and shape.

Let a and b be objects of *tile like* type T and U with *element types* TE and UE respectively. A common element type E is determined as follows:

1. For *arithmetic comparison conversion*, E is the *arithmetic common type* of TE and UE.
2. For *arithmetic tile conversion*, E is the *arithmetic common type* of TE and UE if T and U are both *scalars* or both non-scalars. Otherwise, exactly one of T or U is a scalar and E is the *element type* of the non-scalar.

If E could not be determined because the *arithmetic common type* doesn't exist, the conversion is ill-formed.

After E is determined, the conversion proceeds as follows:

1. Operands a and b undergo *mutual broadcast conversion* to form new objects a' and b' of the same shape B. If the mutual broadcast conversion is ill-formed, the arithmetic conversion as a whole is ill-formed.
2. If a and b are both *scalars*, a' and b' undergo *tile conversion* to E.
3. Otherwise, a' and b' undergo *tile conversion* to `ct::tile<E, B>`.

The conversion as a whole is ill-formed if the final *tile conversion* would be narrowing unless the source element type of the narrowing conversion is *integral* and the target is *floating point*.

Example 1

In the following code, *arithmetic tile conversion* is used to convert the operands prior to performing the addition. The `float` values are converted to `double` and both arguments are broadcasted to a common shape:

$$\begin{pmatrix} 2 & 6 \end{pmatrix} - \begin{pmatrix} 4 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 & 2 \\ 1 & 5 \end{pmatrix}$$

```
namespace ct = ::cuda::tiles;
using i32x1x2 = ct::tile<int, ct::shape<1, 2>>;
using i32x2x1 = ct::tile<int, ct::shape<2, 1>>;
using f32x1x2 = ct::tile<float, ct::shape<1, 2>>;
```

(continues on next page)

(continued from previous page)

```
using f64x2x1 = ct::tile<double, ct::shape<2, 1>>;
using f64x2x2 = ct::tile<double, ct::shape<2, 2>>;

float xData[1][2] = {
  {2, 6},
};

double yData[2][1] = {
  {4},
  {1},
};

f32x1x2 x = ct::load(&xData[0][0] + ct::iota<i32x1x2>());
f64x2x1 y = ct::load(&yData[0][0] + ct::iota<i32x2x1>());
f64x2x2 result = x - y;
```

Example 2

In (1), *arithmetic tile conversion* prefers the `ct::tile` argument when selecting a target element type. The `double` scalar would need to be narrowed to `int` making the conversion ill-formed.

In (2), *arithmetic comparison conversion* does not prefer either argument when determining the target element type. The elements of the `ct::tile` are converted to `double` and the comparison is well-formed.

```
namespace ct = ::cuda::tiles;
using i32x8 = ct::tile<int, ct::shape<8>>;

i32x8 x = ct::full<i32x8>(42);
2.0 * x;  // (1) ill-formed
2.0 == x; // (2) OK
```

Example 3

Example A	
T	`ct::tile<int, ct::shape<4, 1>>`
U	`ct::tile<float, ct::shape<1, 8>>`
Arith. Tile Conversion	`ct::tile<float, ct::shape<4, 8>>`
Arith. Comp. Conversion	`ct::tile<float, ct::shape<4, 8>>`

Example B	
T	`float`
U	`ct::tile<int, ct::shape<4, 8>>`
Arith. Tile Conversion	ill-formed: `float` -> `int` narrowing
Arith. Comp. Conversion	`ct::tile<float, ct::shape<4, 8>>`

Example C	
T	`__nv_bfloat16`
U	`ct::tile<__half, ct::shape<4, 8>>`
Arith. Tile Conversion	illformed: `__nv_bfloat16` -> `__half` narrowing
Arith. Comp. Conversion	ill-formed: Unordered conversion ranks

Example D	
T	`unsigned int`
U	`ct::tile<int, ct::shape<4, 8>>`
Arith. Tile Conversion	ill-formed: `unsigned int` -> `int` narrowing
Arith. Comp. Conversion	ill-formed: `int` -> `unsigned int` narrowing

arithmetic tile promotion
The *arithmetic tile promotion* of an object a of *tile like* type is the *elementwise application* of the standard integral promotions rules[11] to a. The result is a new object of the same *shape* but possibly different *element type* as a. The kind of object is unchanged by the operation: if a was originally a *scalar* it remains a *scalar*. If a was original a `ct::tile` it remains a `ct::tile`.

1.8. Tensor Span

CUDA Tile C++ uses *tensor span like* types to describe multi-dimensional arrays in memory. A tensor span consists of

1. A pointer to the base location of the array in memory.
2. A *layout mapping* describing the array shape and the mapping of logical multi-dimensional indices to linear indices into memory.
3. An *accessor policy* which decorates the tensor span with additional information about how elements of the array may be accessed.

For performance, data referenced by a tensor span should not be indexed directly. Instead, the tensor span should be wrapped in a view type that represents a tiling of the data. Tiles may be loaded from this view type directly.

Example

A 2×4 tensor span wrapping a pointer to memory. It uses the default accessor policy and the default row-major layout.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
```

(continues on next page)

[11] The integral promotion rules are defined in § 7.3.7 [conv.prom] of ISO/IEC 14882:2024

(continued from previous page)

```
float data[2][4] = {
  {0, 1, 2, 3},
  {4, 5, 6, 7}
};

auto span = ct::tensor_span{&data[0][0], ct::extents{2_ic, 4_ic}};
```

The *ct::tensor_span* type implements *tensor span like* and is similar to the C++23 `std::mdspan` type.

1.8.1. Layout Mapping

A *layout mapping* is an object describing the shape of a multi-dimensional array and its in-memory arrangement of elements. The shape is specified using an *extents like* type consisting of static or dynamic dimension values. The arrangement of elements is specified by a collection of statically or dynamically known stride values for each dimension which indicate the distance in memory between consecutive elements along that dimension. The strides determine a mapping from the multi-dimensional index space of array elements to their linear position in memory.

Let M denote a possibly cv-qualified type, m a const glvalue expression of type M and dim a const glvalue expression of type `M::rank_type`, when that is well-formed. M is a *layout mapping* type if M models the C++ standard library concept `std::copyable` and expressions in the following table are well formed and meet the indicated requirements:

Expression	Requirements
`M::extents_type`	A cv-unqualified *extents like* type. **Semantics** Indicates the static components of the multi-dimensional array shape represented by this layout mapping.
`M::rank_type`	Same type as `M::extents_type::rank_type`.
`M::index_type`	Same type as `M::extents_type::index_type`.
`M::layout_type`	A cv-unqualified *layout policy* type such that `M::layout_type::mapping<M::extents_type>` is the same type as `M` **Semantics** Indicates the *layout policy* type which may be used to rebuild this layout mapping given only an *extents like* type.
`M::is_always_strided()`	Static member function returning a prvalue of type `bool`. The expression `M::is_always_strided()` shall be a constant expression and shall always yield `true`. **Semantics** Indicates that the mapping describes a strided layout. Currently, only strided layouts are supported in *tensor span like* types.
`ct::layout_mapping_static_stride<M>{}(dim)`	Static member function of traits class *`ct::layout_mapping_static_stride`* implementing the function call operator. The expression must yield a prvalue of type `size_t`. **Semantics** Yields the statically known stride for dimension `dim`. If that stride is dynamic, returns *`ct::dynamic_extent`*. The expression shall be equality-preserving.[?] No requirements are placed on the behavior for values of `dim` that are not in the range $0 \leq dim < N$ where N is the rank of `M::extents_type`. For all other values of dim which are themselves constant expressions, the expression as a whole must be a constant expression.
`m.stride(dim)`	Non-static member function returning a prvalue of type `M::index_type`. **Semantics** Yields the runtime stride for dimension dim. The expression shall be equality-preserving.[?] The return value shall be non-negative. No requirements are placed on the behavior for values of `dim` that are not in the range $0 \leq dim < N$ where N is the rank of `M::extents_type`. The value of the expression shall be `ct::layout_mapping_static_stride<M>{}(dim)` when dimension dim corresponds to a static stride. In this case, the behavior is undefined if this value is not representable in `M::index_type`.
`m.extents()`	Member function yielding a const glvalue expression of type `M::extents_type`. **Semantics** Indicates the shape of the array. The expression shall be equality-preserving.[?]

Note: A layout mapping need not model equality comparable. Comparison of layout mappings is done by *ct::layout_mapping_equal()*.

If M does not fulfill the semantic requirements above, any usage of M in a Tile C++ API results in undefined behavior.

The remainder of this section describes the properties common to all *layout mapping* types. Let m be an object of *layout mapping* type M and let N be the *rank* of `M::extents_type`.

layout mapping shape

layout mapping rank

layout mapping size

layout mapping index space
The *shape, rank*, *size* and *index space* of m refers to the *shape*, *rank*, *size*, and *index space* of the *extents like* object `m.extents()`.

layout mapping function
The *layout mapping function* of m is a function $m : \mathbb{I} \to \mathbb{N}$ from the *index space* $\mathbb{I}$ of m to natural numbers representing the location of each array index $I = (i_0, i_1, \ldots, i_{N-1}) \in \mathbb{I}$ in a linear index space. The layout mapping function is defined as:

$$m(I) = \text{m.stride}(0) \cdot i_0 + \text{m.stride}(1) \cdot i_1 + \ldots + \text{m.stride}(N-1) \cdot i_{N-1}$$

The mapping function is said to be *injective* if for every pair of of distinct indices I and I' in the *index space* of m, their mappings are distinct: $m(I) \neq m(I')$.

layout mapping equivalent
Objects x and y are said to be *layout mapping equivalent* if all of the following hold:

1. They are both *layout mappings* of rank N.
2. The objects `x.extents()` and `y.extents()` are *extent equivalent*.
3. For each dimension dim in the range $0 \leq dim < N$, the strides `x.stride(dim)` and `y.stride(dim)` are the same value irrespective of index type.

layout policy

layout policy mapping type
A *layout policy* is a factory for producing *layout mapping types* given an *extents like* type describing an array shape. Let P be a possibly cv-qualified type and E a cv-unqualified *extents like* type. P is a *layout policy* for extents type E if the expression `P::mapping_type<E>` is well formed and yields a *layout mapping* type such that `P::mapping_type<E>::extents_type` is E.

1.8.2. Accessor Policy

An *accessor policy* describes how the elements of a multi-dimensional array are accessed. A type A is an *accessor policy* if A models the C++ standard library concept `std::copyable` and the following expressions are well-formed and have the indicated requirements:

Expression	Requirements
`A::element_type`	Possibly cv-qualified *scalar* type. **Semantics** Indicates the element type yielded by an access through A.
`A::data_handle_type`	Same type as `A::element_type*`
`A::reference`	Same type as `A::element_type&`
`ct::enable_contiguous_accessor_policy<A>`	Specialization of variable template *`ct::enable_contiguous_accessor_policy`* yielding `true`. **Semantics** Indicates that accessor policy behaves like a simple pointer dereference into a contiguous region of memory. Attests that an access at `size_t` index idx from base pointer p of type `A::data_handle_type` is a read or write through the glvalue `p[idx]`.

If A does not fulfill the semantic requirements above, any usage of A in a Tile C++ API results in undefined behavior.

1.8.3. Tensor Span Like

A *tensor span like* type represents a handle to a multi-dimensional array in memory. Let T be a type and t a const glvalue expression of type T. T is *tensor span like* if it models the C++ standard library concept `std::copyable` and the expressions in the following table are well formed and fulfill the indicated requirements:

Expression	Requirements
`T::mapping_type`	A cv-unqualified *layout mapping* type
`T::accessor_type`	A cv-unqualified *accessor policy* type
`T::extents_type`	Same type as `T::mapping_type::extents_type`
`T::layout_type`	Same type as `T::mapping_type::layout_type`
`T::index_type`	Same type as `T::extents_type::index_type`
`T::rank_type`	Same type as `T::extents_type::rank_type`
`T::element_type`	Same type as `T::accessor_type::element_type`
`T::value_type`	Same type as `remove-cv-t<T::element_type>`
`T::data_handle_type`	Same type as `T::accessor_type::data_handle_type`
`T::reference`	Same type as `T::accessor_type::reference`
`t.data_handle()`	Non static member function yielding a prvalue of type `T::data_handle_type`. **Semantics** Produces a pointer to the start of a multi-dimensional array. The expression must be equality-preserving.?
`t.mapping()`	Non static member function yielding a const lvalue of type `T::mapping_type`. **Semantics** Produces the mapping object describing the layout of the multi-dimensional array. The expression must be equality-preserving.?
`t.accessor()`	Non static member function yielding a const lvalue of type `T::accessor_type`. **Semantics** Produces the accessor policy describing how elements of the multi-dimensional array are to be accessed.

If T does not fulfill the semantic requirements above, any usage of T in a Tile C++ API results in undefined behavior.

The remainder of this section describes properties common to all *tensor span like* types. Let T be a *tensor span like* type and t an object of that type.

tensor span element type

tensor span value type

The type `T::element_type` is called the *element type* of T and is the type of the elements of the multi-dimensional array referenced by t. A const qualification of `T::element_type` indicates whether the underlying array may be written through an instance of T. The *value type* of t is the element type without cv-qualifiers.

tensor span shape

tensor span rank

tensor span size

tensor span index space
The *shape, rank, size,* and *index space* of t refer to the *shape*, *rank*, *size*, and *index space* of the *layout mapping* object `t.mapping()`.

tensor span notation
The notation t_k designates the length of dimension k for the *shape* of `t`.

tensor span function
Let m be the *layout mapping function* of `t.mapping()` and $\mathbb{I}$ the *index space* of t. The *tensor span function* $t : \mathbb{I} \to P$ of t associates indices in $\mathbb{I}$ to pointers. If p is the pointer produced by `t.data_handle()`, the mapping is defined as follows:

$$t(I) = p + m(I)$$

1.9. Numeric Modifiers

The following modifiers are provided as arguments to influence the numerical behavior of certain arithmetic and math APIs.

1.9.1. Rounding Mode

A *rounding mode* is a policy for selecting a floating point value to represent the result of a mathematical computation. This section describes the rounding mode behaviors while `ct::rounding_mode` documents the APIs for selecting rounding modes.

The *precise rounding modes* are rounding modes with well defined IEEE 754[?] semantics. These rounding modes yield an exact result if possible. Otherwise, one of the two consecutive floating point values which bound the infinitely precise result is yielded (subject to appropriate handling for non-finite values). The following *precise rounding modes* are defined:

Round Ties to Even
Indicates the *roundTiesToEven*[?] IEEE 754 rounding attribute. The value closest to the infinitely precise result is yielded (subject to certain corner cases which are documented in the IEEE 754 specification).

Round Toward Zero
Indicates the *roundTowardZero*[?] IEEE 754 rounding attribute. The value closest to but not greater in magnitude to the infinitely precise result is yielded.

Round Toward Negative
Indicates the *roundTowardNegative*[?] IEEE 754 rounding attribute. The value closest to but not greater than the infinitely precise result is yielded.

Round Toward Positive
Indicates the *roundTowardPositive*[?] IEEE 754 rounding attribute. The value closest to but not less than the infinitely precise result is yielded.

Note: The above listing is a summary of the rounding mode behavior and does not represent the exact semantics. Consult the IEEE 754[?] specification for the exact behavior of precise rounding modes.

Some operations support *imprecise rounding modes*. The imprecise rounding modes are not guaranteed to yield floating point values which are adjacent to the infinitely precise result. Maximum error

bounds may be documented for the results of operations using imprecise rounding modes. The following imprecise rounding modes are defined:

Round Approximate
Indicates a rounding policy that prioritizes computation speed over precision.

Round Full
Indicates a rounding policy that balances computation speed and precision.

The semantics of the imprecise rounding modes depend on the operation. See the documentation of the relevant operation for details.

Default Rounding Mode
A distinguished rounding mode called the *default rounding mode* determines the rounding behavior for most floating point APIs when one is not directly provided by the user. The default rounding mode is *Round Ties to Even*.

Example

In the following code, the *Round Toward Negative* rounding mode is selected for the addition. The result is `8.0f` instead `8.0f + 8 * eps` which would be produced by the default behavior.

```
namespace ct = ::cuda::tiles;
float eps = 0x0.000002p0f;
float result = ct::add(8.0f, 5 * eps, ct::round_toward_negative_t{});
```

Note: Not all rounding modes are supported by all APIs or operand types. Consult the relevant API for details.

1.9.2. Subnormals Rounding Mode

A *subnormals rounding mode* is a policy for handling subnormal[4] inputs and results in arithmetic and math APIs. This section describes the subnormals rounding mode behaviors while *`ct::subnormals_rounding_mode`* documents the APIs for selecting subnormals rounding modes.

The function subround accepts and returns a floating point value and applies a subnormals rounding mode as defined below:

Preserve Subnormals

$$\text{subround}(x) = x$$

Subnormal values are not modified when passed as arguments or returned as results of arithmetic and math APIs

Round Subnormals to Zero

$$\text{subround}(x) = \begin{cases} 0.0 & \text{x is subnormal and has positive sign} \\ -0.0 & \text{x is subnormal and has negative sign} \\ x & \text{otherwise} \end{cases}$$

[4] See § 3.3 of IEEE 754-2019

Subnormal operands and results are flushed to sign preserving zero in arithmetic and math APIs.

Default Subnormals Rounding Mode
A distinguished subnormals rounding mode called the *default subnormals rounding mode* determines the subnormals rounding behavior for operations where the mode is not directly specified by the user. The default subnormals rounding mode is *Preserve Subnormals*.

Example

In the following code, the *Round Subnormals to Zero* subnormals rounding mode is selected for the subtraction. The result of the subtraction would normally yield a subnormal value, however, because *Round Subnormals to Zero* is used, 0.0 is returned instead.

```
namespace ct = ::cuda::tiles;
float result = ct::sub(0x1.1p-126f, 0x1.0p-126f,
                       ct::round_ties_to_even_t{},
                       ct::round_subnormals_to_zero_t{});
```

Note: Not all subnormals rounding modes are supported by all APIs or operand types. Consult the relevant API for details.

1.9.3. NaN Propagation Mode

A *NaN propagation mode* determines how maximum and minimum operations should handle NaN inputs. This section describes the NaN propagation mode behaviors while `ct::nan_propagation_mode` documents the APIs for selecting a NaN propagation mode.

Suppress NaN
In this mode, maximum and minimum operations yield a NaN value only when both of their inputs are NaN. When one input is NaN and the other is a number, the number is returned.

Propagate NaN
In this mode, maximum and minimum operations yield a NaN value when either of their inputs are NaN.

Default NaN Propagation Mode
When a NaN propagation mode is not explicitly specified, the *default nan propagation mode* is used. The default NaN propagation mode is *suppress NaN*.

1.9.4. View Padding

Masked load operations on tile views such as `ct::partition_view` may optionally specify a padding value which determines the elements of the result tile corresponding to out of bounds loads. This section documents the view padding options while `ct::view_padding` documents the APIs for selecting view padding.

Zero View Padding
Indicates the value 0 for loads of integer values and positive zero for loads of *floating point* values.

Positive Infinity View Padding
Indicates the value ∞.

Negative Infinity View Padding
Indicates the value $-\infty$.

NaN View Padding
Indicates the value NaN.

Default View Padding
Indicates a default view padding value that is used when one is not explicitly specified. The default view padding is *zero view padding*.

1.10. Memory Model

The purpose of a memory model is to constrain the set of observable values exhibited by memory operations in a multi-threaded program. The Tile C++ memory model is based on the C++ standard memory model with modifications.

Informally, a Tile C++ program behaves as if each tile block is executed by a separate thread except that multiple load and store operations dispatched by a single Tile C++ API may execute simultaneously on different threads. Additionally, the visibility of atomic memory operations are limited to the threads of the specified *thread scope*.

Example

In the following example, a call to one of the *store* APIs will dispatch multiple memory operations that write to a single location. In the *`ct::atomic_store()`* invocation, no undefined behavior occurs because the writes are atomic and occur in the same thread scope (tile block scope). In the *`ct::store()`* invocation, undefined behavior occurs because the writes constitute a *data race*.

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;

int x = 0;

auto ptrs = ct::full<ct::tile<int*, ct::shape<4>>>(&x);

// No UB
ct::atomic_store(ptrs, ct::iota<i32x4>(),
                 ct::memory_order_relaxed_t{},
                 ct::thread_scope_block_t{});

// Value of 'x' is either 0, 1, 2, or 3 at this point
printf("%i\n", x);

// UB, due to data race
ct::store(ptrs, ct::iota<i32x4>());
```

The modifications to the C++ memory model are described below:

Tile Threads
A single thread[12] of execution is generated for each tile block that is launched by a tile kernel.

[12] See § 6.9.2.1 [intro.multithread.general] of ISO/IEC 14882:2024

These threads provide the *parallel forward progress guarantee*[13].

Memory Operation

A *memory operation* is an evaluation executed by some thread that modifies a memory location[14]. Memory operations are classified as either a read, a write, or a read-write. They may be further classified as either weak or strong. Strong memory operations are endowed with a *memory order* and *thread scope*.

The following constructs generate one or more memory operations:

1. A read or write through a glvalue generates one weak read (respectively write) memory operation on the memory location of the glvalue. The memory operation executes in the thread which evaluates the glvalue access.
2. A call to certain Tile C++ API functions may generate one or more memory operations. The relevant API function specifies:
    1. How many memory operations are generated by a single invocation of the API.
    2. The memory locations accessed by the generated memory operations.
    3. Whether the operations are read, write, or read-write.
    4. Whether the operations are weak or strong.
    5. If the operations are strong, what their *memory order* and *thread scope* is.

When a Tile C++ API generates memory operations, each memory operation is evaluated on a separate thread of execution. The beginning of the invocation of the API synchronizes with[15] the beginning of the evaluation of the memory operation in each thread. The end of the evaluation of the memory operation in each thread synchronizes with[15] the end of the API invocation.

Memory Order

The *memory order* of a *strong memory operation* corresponds to a C++ memory order[18] and indicates how the memory operation may synchronize with? other memory operations. Tile C++ supports the following memory orders:

Relaxed Memory Order
: The operation does not imply any synchronization relationship.

Release Memory Order
: Certain memory accesses occurring prior to this operation may not be reordered after it. Applicable only to write and read-write operations.

Acquire Memory Order
: Certain memory accesses occurring after this operation may not be reordered before it. Applicable only to read and read-write operations.

Acquire Release Memory Order
: Certain memory operations may not be reordered before or after this operation. Applicable only to read-write operations.

The *relaxed* and *acquire* are known as *read memory orders*. The *relaxed* and *release* are known as *write memory orders*.

APIs for selecting a memory order are documented in `ct::memory_order`.

[13] See § 6.9.2.3 [forward.progress] of ISO/IEC 14882:2024
[14] See § 6.7.1 [intro.memory] of ISO/IEC 14882:2024
[15] See § 6.9.2.2 [intro.race] of ISO/IEC 14882:2024
[18] See § 33.5.4 [atomics.order] of ISO/IEC 14882:2024

Thread Scope
The *thread scope* of a *strong memory operation* A constrains the strong memory operations which may *synchronize with* A as well as the *data races* that A may participate in.

Tile Block Scope
The set of threads that execute the *memory operations* generated by the the Tile C++ API that generated A.

Device Scope
The threads of the GPU device in which A executes including threads from other SIMT or Tile kernels.

System Scope
The threads of the whole system, including all GPU devices and the CPU host.

APIs for selecting a thread scope are documented in `ct::thread_scope`.

Default Thread Scope
A distinguished thread scope called the *default thread scope* specifies the scope used for invocations of Tile C++ atomic memory APIs when a thread scope is not explicitly specified. The default thread scope is *system scope*.

Strongly Scoped Operations
Two *memory operations* A and B are said to be *strongly scoped* if they access the same memory location[?], they are both strong memory operations, and A's *thread scope* contains the thread executing B and B's *thread scope* contains the thread executing A.

Atomic Objects
An object is atomic[20] and has a modification order[19] as specified by the C++ memory model if all *memory operations* which access that object are strong. The operations which access this object are considered to be atomic operations[21] with the specified memory order[?].

Synchronizes With
Memory operation A accessing the same *atomic object* as memory operation B shall synchronize with[?] B according to the rules for atomic operations[21] only if A and B are *strongly scoped*.

Data Races
In addition to the situations described in § 6.9.2.2 of [intro.race] ISO/IEC 14882:2024, a data race[16] occurs if two *memory operations* which are not *strongly scoped* access the same memory location[?] and neither happens before[17] the other.

[20] See § 6.9.2.2 [intro.race] of ISO/IEC 14882:2024
[19] See § 6.9.2.2 [intro.race] of ISO/IEC 14882:2024
[21] See § 33.5.4 [atomics.order] of ISO/IEC 14882:2024
[16] See § 6.9.2.2 [intro.race] of ISO/IEC 14882:2024
[17] See § 6.9.2.2 [intro.race] of ISO/IEC 14882:2024

Chapter 2. API Reference

2.1. General Operations

2.1.1. Thread Functions

2.1.1.1 cuda::tiles::bid

__tile__ uint3 **bid**() noexcept;

Yields the block index of the actively executing tile block. The block indices are organized into x, y, and z and are determined by the launch parameters of the tile kernel.

Example

The following code copies the elements from array `in` to `out` using 16 blocks arranged in a 4×4 grid. Each block handles 8 elements of data at a time. The *`ct::bid()`* function is used to select the starting offset of the data that will be handled by the block.

```
#include "cuda_tile.h"

namespace ct = ::cuda::tiles;
using i32x8 = ct::tile<int, ct::shape<8>>;

__tile_global__ void blockCopy(int* in, int* out) {
  int offset = ct::bid().x * ct::num_blocks().y * 8 + ct::bid().y * 8;

  auto data = ct::load(in + offset + ct::iota<i32x8>());
  ct::store(out + offset + ct::iota<i32x8>(), data);
}

void callBlockCopy() {
  int* pOut, *pIn;
  cudaMalloc(&pIn, 16 * 8 * sizeof(int));
  cudaMemset(pIn, 0, 16 * 8 * sizeof(int));
  cudaMalloc(&pOut, 16 * 8 * sizeof(int));
  blockCopy<<<dim3(4, 4, 1), 1>>>(pIn, pOut);
}
```

2.1.1.2 cuda::tiles::num_blocks

__tile__ dim3 **num_blocks**( ) noexcept;

Yields the number of blocks in the tile grid. The number of blocks is determined by the kernel launch parameters.

2.1.2. Type Classification

2.1.2.1 cuda::tiles::integral

template<typename **T**>
concept **integral** = /* atomic constraint */;

Determines if T is a possibly cv-qualified integral type.

2.1.2.2 cuda::tiles::tile_like

template<typename **T**>
concept **tile_like** = /* atomic constraint */;

Indicates whether T is a *tile like* type.

2.1.2.3 cuda::tiles::pointer_tile

template<typename **T**>
concept **pointer_tile** = ct::*tile_like*<*T*> && /* atomic constraint */

Indicates whether T is a *pointer tile* type.

2.1.2.4 cuda::tiles::numeric_tile

template<typename **T**>
concept **numeric_tile** = ct::*tile_like*<*T*> && /* atomic constraint */;

Indicates whether T is a *numeric tile* type.

2.1.2.5 cuda::tiles::arithmetic_tile

template<typename **T**>
concept **arithmetic_tile** = ct::*numeric_tile*<*T*> && /* atomic constraint */;

Indicates whether T is an *arithmetic tile* type.

2.1.2.6 cuda::tiles::floating_point_tile

template<typename **T**>
concept **floating_point_tile** = ct::*numeric_tile*<*T*> && /* atomic constraint */;

Indicates whether T is a *floating point tile* type.

2.1.2.7 cuda::tiles::basic_floating_point_tile

template<typename **T**>
concept **basic_floating_point_tile** = ct::*arithmetic_tile*<*T*> && ct::*floating_point_tile*<*T*> && /* atomic constraint */;

Indicates whether T is a *basic floating point tile* type.

2.1.2.8 cuda::tiles::restricted_floating_point_tile

template<typename **T**>
concept **restricted_floating_point_tile** = ct::*floating_point_tile*<*T*> && /* atomic constraint */;

Indicates whether T is a *restricted floating point tile* type.

2.1.2.9 cuda::tiles::integral_tile

template<typename **T**>
concept **integral_tile** = ct::*arithmetic_tile*<*T*> && /* atomic constraint */;

Indicates whether T is an *integral tile* type.

2.1.2.10 cuda::tiles::scalar

template<typename **T**>
concept **scalar** = ct::*tile_like*<*T*> && /* atomic constraint */;

Indicates whether T is a *scalar* type.

2.1.2.11 cuda::tiles::pointer_scalar

template<typename **T**>
concept **pointer_scalar** = ct::*pointer_tile*<*T*> && ct::*scalar*<*T*>;

Indicates whether T is a *pointer scalar* type.

2.1.2.12 cuda::tiles::numeric_scalar

template<typename **T**>
concept **numeric_scalar** = ct::*numeric_tile*<*T*> && ct::*scalar*<*T*>;

Indicates whether T is a *numeric scalar* type.

2.1.2.13 cuda::tiles::arithmetic_scalar

template<typename **T**>
concept **arithmetic_scalar** = ct::*arithmetic_tile*<*T*> && ct::*scalar*<*T*>;

Indicates whether T is an *arithmetic scalar* type.

Note: `ct::arithmetic_scalar` subsumes `ct::numeric_tile` and `ct::numeric_scalar`.

2.1.2.14 cuda::tiles::floating_point_scalar

template<typename **T**>
concept **floating_point_scalar** = ct::*floating_point_tile*<*T*> && ct::*scalar*<*T*>;

Indicates whether T is a *floating point scalar* type.

Note: `ct::floating_point_scalar` subsumes `ct::numeric_tile` and `ct::numeric_scalar`.

2.1.2.15 cuda::tiles::basic_floating_point_scalar

template<typename **T**>
concept **basic_floating_point_scalar** = ct::*basic_floating_point_tile*<*T*> && ct::*scalar*<*T*>;

Indicates whether T is a *basic floating point scalar* type.

Note: `ct::basic_floating_point_scalar` subsumes all of the following:

- `ct::floating_point_tile`
- `ct::arithmetic_tile`
- `ct::floating_point_scalar`
- `arithmetic_scalar`

2.1.2.16 cuda::tiles::restricted_floating_point_scalar

template<typename **T**>
concept **restricted_floating_point_scalar** = ct::*restricted_floating_point_tile*<*T*> &&
ct::*scalar*<*T*>;

Indicates whether T is a *restricted floating point scalar* type.

Note: `ct::restricted_floating_point_scalar` subsumes `ct::floating_point_tile` and `ct::floating_point_scalar`.

2.1.2.17 cuda::tiles::integral_scalar

template<typename **T**>
concept **integral_scalar** = ct::*integral_tile*<*T*> && ct::*scalar*<*T*> && ct::*integral*<*T*>;

Indicates whether T is an *integral scalar* type.

Note: *ct::integral_scalar* subsumes *ct::arithmetic_tile* and *ct::arithmetic_scalar*.

2.1.3. Extents Like Properties

2.1.3.1 cuda::tiles::extents_like

template<typename **T**>
concept **extents_like** = /* atomic constraint */

Indicates whether T is an *extents like* type.

2.1.3.2 cuda::tiles::extents_equal

template<ct::*extents_like* **T**, ct::*extents_like* **U**>
__tile__ __host__ __device__ constexpr bool **extents_equal**(*T* const &x, *U* const &y) noexcept;

Indicates whether the two *extents like* objects x and y are *extent equivalent*.

2.1.3.3 cuda::tiles::shape_like

template<typename **T**>
concept **shape_like** = ct::*extents_like*<*T*> && /* atomic constraint */;

Indicates whether T is a *shape like* type.

2.1.3.4 cuda::tiles::shape_size_v

template<ct::*shape_like* **T**>
inline constexpr size_t **shape_size_v** = /* see below */;

Yields the *size* of the *shape like* type T. If the size is not representable within the type `size_t`, the behavior is undefined.

2.1.3.5 cuda::tiles::same_shape

template<typename **T**, typename **U**>
concept **same_shape** = ct::*shape_like*<*T*> && ct::*shape_like*<*U*> && /* atomic constraint */;

Indicates whether *shape like* types T and U are *shape equivalent*.

2.1.4. Tile Like Properties

2.1.4.1 cuda::tiles::tile_element_t

template<ct::*tile_like* **T**>
using **tile_element_t** = /* see below */;

Yields the *element type* of a *tile like* type T.

2.1.4.2 cuda::tiles::tile_shape_t

template<ct::*tile_like* **T**>
using **tile_shape_t** = /* see below */;

Yields the *shape type* of a *tile like* type T.

2.1.4.3 cuda::tiles::tile_size_v

template<ct::*tile_like* **T**>
inline constexpr size_t **tile_size_v** = ct::*shape_size_v*<ct::*tile_shape_t*<*T*>>;

Yields the *tile size* of *tile like* type T.

2.1.4.4 cuda::tiles::tile_rank_v

template<ct::*tile_like* **T**>
inline constexpr size_t **tile_rank_v** = ct::*tile_shape_t*<*T*>::rank();

Yields the *rank* of *tile like* type T.

2.1.4.5 cuda::tiles::tile_with_element_t

template<ct::*tile_like* **T**, ct::*scalar* **E**>
using **tile_with_element_t** = /* see below */;

Yields a *tile like* type whose *element type* is E and whose *shape* matches that of T. If T is a specialization of `ct::tile`, the result is also a specialization of `tile`. If T is a *scalar* type, the result is a *Scalars* type. The result type is never cv-qualified.

2.1.4.6 cuda::tiles::tile_shape

template<typename **T**>
concept **tile_shape** = ct::*shape_like*<*T*> && /* atomic constraint */

Indicates whether T is a *shape like* type that is *tile compatible*.

2.1.5. Conversions

2.1.5.1 cuda::tiles::scalar_convertible_to

template<typename **T**, typename **U**>
concept **scalar_convertible_to** = ct::*scalar*<*T*> && ct::*scalar*<*U*> && /* atomic constraint */

Indicates whether there exists a *scalar conversion* from objects of type T to type U.

2.1.5.2 cuda::tiles::non_narrowing_scalar_convertible_to

template<typename **T**, typename **U**>
concept **non_narrowing_scalar_convertible_to** = ct::*scalar_convertible_to*<*T*, *U*> && /* atomic constraint */

Indicates whether there exists a non-narrowing *scalar conversion* from objects of type T to type U.

2.1.5.3 cuda::tiles::tile_convertible_to

template<typename **T**, typename **U**>
concept **tile_convertible_to** = ct::*tile_like*<*T*> && ct::*tile_like*<*U*> && /* atomic constraint */;

Indicates whether there exists a *tile conversion* from an object of type T to type U.

2.1.5.4 cuda::tiles::non_narrowing_tile_convertible_to

template<typename **T**, typename **U**>
concept **non_narrowing_tile_convertible_to** = ct::*tile_convertible_to*<*T*, *U*> && /* atomic constraint */;

Indicates whether there exists a non-narrowing *tile conversion* from an object of type T to type U.

2.1.5.5 cuda::tiles::bool_tile_convertible

template<typename **T**>
concept **bool_tile_convertible** = ct::*tile_like*<*T*> && ct::*tile_convertible_to*<*T*, ct::*tile_with_element_t*<*T*, bool>>;

Indicates whether T *bool tile convertible*.

2.1.5.6 cuda::tiles::shape_broadcastable_to

template<typename **T**, typename **U**>
concept **shape_broadcastable_to** = ct::*shape_like*<*T*> && ct::*shape_like*<*U*> && /* atomic constraint */

Indicates whether *shape like* type T may be *broadcasted to* type U.

2.1.5.7 cuda::tiles::broadcastable_to

template<typename **T**, typename **U**>
concept **broadcastable_to** = ct::*tile_like*<*T*> && ct::*tile_shape*<*U*> && ct::*shape_broadcastable_to*<ct::*tile_shape_t*<*T*>, *U*>;

Indicates whether *tile like* type T is *broadcastable to* shape U.

2.1.5.8 cuda::tiles::shape_broadcast_compatible

template<typename **T**, typename **U**>
concept **shape_broadcast_compatible** = ct::*shape_like*<*T*> && ct::*shape_like*<*U*> && /* atomic constraint */;

Indicates whether *shape like* types T and U are *mutual broadcast compatible*.

2.1.5.9 cuda::tiles::shape_broadcast_t

template<ct::*shape_like* **T**, ct::*shape_like* **U**>
requires ct::*shape_broadcast_compatible*<*T*, *U*>
using **shape_broadcast_t** = /* see below */;

Yields the *mutual broadcast shape* of *shape like* types T and U.

2.1.5.10 cuda::tiles::broadcast_compatible

template<typename **T**, typename **U**>
concept **broadcast_compatible** = ct::*tile_like*<*T*> && ct::*tile_like*<*U*> && /* atomic constraint */;

Indicates whether T and U are *broadcast compatible*.

2.1.5.11 cuda::tiles::mutual_broadcast_t

template<ct::*tile_like* **T**, ct::*tile_like* **U**, ct::*scalar* **E**>
requires ct::*broadcast_compatible*<*T*, *U*>
using **mutual_broadcast_t** = /* see below */;

Yields a cv-unqualified *tile like* type whose shape matches the *mutual broadcast shape* of the shapes of T and U and whose *element type* is E. The result type is a scalar if and only if both T and U are scalars.

2.1.5.12 cuda::tiles::arithmetic_tile_convertible

template<typename **T**, typename **U**>
concept **`arithmetic_tile_convertible`** = ct::*arithmetic_tile*<*T*> && ct::*arithmetic_tile*<*U*> && ct::*broadcast_compatible*<*T*, *U*> && /* atomic constraint */

Tests whether an *arithmetic tile conversion* exists between objects of types T and U.

2.1.5.13 cuda::tiles::arithmetic_tile_conversion_t

template<ct::*arithmetic_tile* **T**, ct::*arithmetic_tile* **U**>
requires ct::*arithmetic_tile_convertible*<*T*, *U*>
using **`arithmetic_tile_conversion_t`** = /* see below */;

Yields the common type C after running *arithmetic tile conversion* on two objects of types T and U.

2.1.5.14 cuda::tiles::arithmetic_tile_comparable

template<typename **T**, typename **U**>
concept **`arithmetic_tile_comparable`** = ct::*arithmetic_tile*<*T*> && ct::*arithmetic_tile*<*U*> && *broadcast_compatible*<*T*, *U*> && /* atomic constraint */

Tests whether an *arithmetic comparison conversion* exists between objects of types T and U.

2.1.5.15 cuda::tiles::arithmetic_tile_comparison_t

template<ct::*arithmetic_tile* **T**, ct::*arithmetic_tile* **U**>
requires ct::*arithmetic_tile_comparable*<*T*, *U*>
using **`arithmetic_tile_comparison_t`** = /* see below */;

Yields the result type C after running *arithmetic comparison conversion* on two objects of types T and U.

2.1.5.16 cuda::tiles::arithmetic_tile_promotion_t

template<ct::*arithmetic_tile* **T**>
using **`arithmetic_tile_promotion_t`** = /* see below */;

Yields the result type after applying *arithmetic tile promotion* to a hypothetical object a of type T.

2.1.6. Tensor Span Like Properties

2.1.6.1 cuda::tiles::layout_mapping

template<typename **T**>
concept **`layout_mapping`** = /* atomic constraint */

Indicates whether T is a *layout mapping* type.

2.1.6.2 cuda::tiles::accessor_policy

template<typename **T**>
concept **accessor_policy** = /* atomic constraint */

Indicates whether T is an *accessor policy* type.

2.1.6.3 cuda::tiles::tensor_span_like

template<typename **T**>
concept **tensor_span_like** = /* atomic constraint */

Indicates whether T is a *tensor span like* type.

2.1.6.4 cuda::tiles::layout_mapping_equal

template<ct::*layout_mapping* **Lhs**, ct::*layout_mapping* **Rhs**>
__tile__ __host__ __device__ constexpr bool **layout_mapping_equal**(*Lhs* const &x, *Rhs* const &y) noexcept;

Indicates whether *layout mappings* x and y are *layout mapping equivalent*.

2.1.7. Exposition Only Entities

The following *exposition only* entities are defined below in terms of a C++23[1] standard library API. These APIs are not necessarily defined or available when including the `cuda_tile.h` header.

template<typename **T**>
using **remove-cv-t** = std::remove_cv_t<*T*>;

template<typename **T**>
using **remove-pointer-t** = std::remove_pointer_t<*T*>;

template<typename **T**>
using **make-signed-t** = std::make_signed_t<*T*>;

using **nullptr-t** = decltype(nullptr);

template<typename **T**>
inline constexpr bool **is-const-v** = std::is_const_v<*T*>;

template<typename **T**>
inline constexpr bool **is-volatile-v** = std::is_volatile_v<*T*>;

template<typename **T**, typename **U**>
inline constexpr bool **is-convertible-v** = std::is_convertible_v<*T*, *U*>;

template<typename **T**, typename **U**>
inline constexpr bool **is-nothrow-convertible-v** = std::is_nothrow_convertible_v<*T*, *U*>;

template<typename **T**, typename ...**Args**>
inline constexpr bool **is-constructible-v** = std::is_constructible_v<*T*, *Args*...>;

[1] See ISO/IEC 14882:2024

2.2. Arithmetic Operations

This section documents the APIs for performing arithmetic on tiles.

2.2.1. Addition

```
template<
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs>
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> operator+(Lhs lhs, Rhs rhs) noexcept;
template<
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs>
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> add(Lhs lhs, Rhs rhs) noexcept;
template<
ct::rounding_mode Mode = ct::default_rounding_mode(),
ct::subnormals_rounding_mode SubMode = ct::default_subnormals_rounding_mode(),
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs> && /* atomic constraint */
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> add(Lhs lhs, Rhs rhs,
                                                        ct::rounding_mode_constant<Mode>,
                                                        ct::subnormals_rounding_mode_constant<SubMode>
                                                        = {}) noexcept;
```

Performs *elementwise* addition on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands and let T be their type.

If T is an unsigned *integral scalar*, the result of each addition is $a+b \bmod 2^n$ where n is the *bitwidth* of T.

If T is a signed *integral scalar*, the result of each addition is $a+b$. The behavior is undefined if $a+b$ is not representable in T for any pair of corresponding elements in the converted operands.

If T is a *basic floating point scalar*, the result of each addition is

$$\text{subround}(\textbf{addition}(\text{subround}(a), \text{subround}(b)))$$

where **addition** is the IEEE 754 *addition*[1] with *rounding mode* and subround applies a *subnormals rounding mode*. For overloads (1) and (2), the *default rounding mode* and *default subnormals rounding mode* are used. For overload (3), `Mode` determines the rounding mode and `SubMode` determines the subnormals rounding mode.

The atomic constraint of overload (3) validates that:

1. T is a *basic floating point scalar*

[1] See *Arithmetic operations* § 5.4.1 of IEEE 754-2019

2. Mode is a *precise rounding mode*
3. If SubMode is *round subnormals to zero*, then T is `float`.
4. The values Mode and SubMode are enumerators of their respective types.

Example

The following example shows how *arithmetic tile conversion* is used in the addition APIs. The first operand (an integer scalar) is broadcasted and converted to match the type of the second operand (a 2×2 floating point tile).

```
namespace ct = ::cuda::tiles;
using i32x2x2 = ct::tile<int, ct::shape<2, 2>>;
using f32x2x2 = ct::tile<float, ct::shape<2, 2>>;

float data[2][2] = {
  {0.0, 1.5},
  {3.0, 3.5},
};

f32x2x2 x = ct::load(&data[0][0] + ct::iota<i32x2x2>());
f32x2x2 result = 5 + x;
```

The `result` holds the following value after the code runs:

$$\begin{pmatrix} 5.0 & 6.5 \\ 8.0 & 8.5 \end{pmatrix}$$

Example

The following example uses the *`ct::round_toward_negative_t`* helper type to select a rounding mode for the addition. Note that the `ct::add` API must be used instead of the operator overload when specifying an explicit rounding mode:

```
namespace ct = ::cuda::tiles;
float eps = 0x0.000002p0f;
float result = ct::add(8.0f, 5 * eps, ct::round_toward_negative_t{});
```

The result of the addition $8 + 5\varepsilon$ is rounded toward negative infinity to yield the value `8.0f`.

2.2.2. Arithmetic Promotion

template<ct::*arithmetic_tile* **T**>
__tile__ ct::*arithmetic_tile_promotion_t*<*T*> **operator+**(*T* x) noexcept;

Yields the result of *arithmetic tile promotion* on the operand x.

Example

In the following example, the tile of `char` elements is promoted to a tile of `int` elements. The numeric values are unchanged after the operation:

```
namespace ct = ::cuda::tiles;
using i8x2x2 = ct::tile<char, ct::shape<2, 2>>;
using i32x2x2 = ct::tile<int, ct::shape<2, 2>>;

char data[2][2] = {
  {0, 1},
  {2, 3},
};

i8x2x2 x = ct::load(&data[0][0] + ct::iota<i32x2x2>());
i32x2x2 result = +x;
```

2.2.3. Subtraction

template<
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **operator-**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **sub**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<
ct::*rounding_mode* **Mode** = ct::*default_rounding_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **sub**(*Lhs* lhs, *Rhs* rhs,
ct::*rounding_mode_constant*<*Mode*>,
ct::*subnormals_rounding_mode_constant*<*SubMode*>
= {}) noexcept;

Performs *elementwise* subtraction of the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b denote corresponding elements of the converted operands `lhs` and `rhs` respectively and let T be their type.

If T is an unsigned *integral scalar*, the result of each subtraction is $a - b \bmod 2^n$ where n is *bitwidth* of T.

If T is a signed *integral scalar*, the result of each subtraction is $a - b$. The behavior is undefined if $a - b$ is not representable in T for any pair of corresponding elements in the converted operands.

If T is a *basic floating point scalar*, the result of each subtraction is

$$\text{subround}(\textbf{subtraction}(\text{subround}(a), \text{subround}(b)))$$

where **sub** is the IEEE 754 *subtraction*[Page 49, 1] operation with *rounding mode* and subround applies a *subnormals rounding mode*. For overloads (1) and (2), the *default rounding mode* and *default subnormals rounding mode* are used. For overload (3), `Mode` determines the rounding mode and `SubMode` determines the subnormals rounding mode.

The atomic constraint of overload (3) validates that:

1. T is a *basic floating point scalar*
2. `Mode` is a *precise rounding mode*
3. If `SubMode` is *round subnormals to zero*, then T is `float`.
4. The values `Mode` and `SubMode` are enumerators of their respective types.

Example

The following example shows how to use the `ct::round_subnormals_to_zero_t` helper type to specify a flush to zero behavior. Note that the `ct::sub` function must be used instead of the operator overload when specifying an explicit subnormals rounding mode. The result of the computation below is `0.0f`.

```
namespace ct = ::cuda::tiles;
float result = ct::sub(0x1.1p-126f, 0x1.0p-126f,
                       ct::round_ties_to_even_t{},
                       ct::round_subnormals_to_zero_t{});
```

2.2.4. Negation

template<ct::*arithmetic_tile* **U**>
__tile__ *U* **operator-**(*U* x) noexcept;

Performs *elementwise* negation of operand x.

Let a be an element of the operand x and T its type.

If T is an unsigned *integral scalar*, the result of the negation is $2^n - a$, where n is the *bitwidth* of T.

If T is a signed *integral scalar*, the result of the negation is $-a$. If the result value is not representable in T for any element of the operand x, the behavior is undefined.

If T is a *basic floating point scalar*, the result of the negation is **negate**(a) where **negate** is the IEEE 754 *negate*[2] operation.

[2] See *Sign bit operations* § 5.5.1 of IEEE 754-2019

2.2.5. Multiplication

```
template<
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs>
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> operator*(Lhs lhs, Rhs rhs) noexcept;
template<
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs>
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> mul(Lhs lhs, Rhs rhs) noexcept;
template<
ct::rounding_mode Mode = ct::default_rounding_mode(),
ct::subnormals_rounding_mode SubMode = ct::default_subnormals_rounding_mode(),
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs> && /* atomic constraint */
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> mul(Lhs lhs, Rhs rhs,
                                                        ct::rounding_mode_constant<Mode>,
                                                        ct::subnormals_rounding_mode_constant<SubMode>
                                                        = {}) noexcept;
```

Performs *elementwise* multiplication on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b denote corresponding elements of the converted operands `lhs` and `rhs` respectively and let T be their type.

If T is an unsigned *integral scalar*, the result of each multiplication is $a \cdot b \bmod 2^n$ where n is the *bitwidth* of T.

If T is a signed *integral scalar*, the result of each multiplication is $a \cdot b$. If this result is not representable in T for any pair of corresponding elements, the behavior is undefined.

If T is a *basic floating point scalar*, the result of each addition is

$$\text{subround}(\textbf{multiplication}(\text{subround}(a), \text{subround}(b)))$$

where **multiplication** is the IEEE 754 *multiplication*[?] operation with *rounding mode* and subround applies a *subnormals rounding mode*. For overloads (1) and (2), the *default rounding mode* and *default subnormals rounding mode* are used. For overload (3), `Mode` specifies the rounding mode and `SubMode` specifies the subnormals rounding mode.

The atomic constraint of overload (3) validates that:

1. T is a *basic floating point scalar*
2. `Mode` is a *precise rounding mode*
3. If `SubMode` is *round subnormals to zero*, then T is `float`.
4. The values `Mode` and `SubMode` are enumerators of their respective types.

2.2.6. Division

template<
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **operator/**(*Lhs* dividend, *Rhs* divisor) noexcept;

template<
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **div**(*Lhs* dividend, *Rhs* divisor) noexcept;

template<
ct::*rounding_mode* **Mode** = ct::*default_rounding_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **div**(*Lhs* dividend, *Rhs* divisor,
ct::*rounding_mode_constant*<*Mode*>,
ct::*subnormals_rounding_mode_constant*<*SubMode*>
= {}) noexcept;

Performs *elementwise* division of the *arithmetic tile converted* operands `dividend` and `divisor`.

Let a and b denote corresponding elements of the converted operands `dividend` and `divisor` respectively and let T be their type.

If T is an *integral scalar*, the result of each division is

$$\text{trunc}\left(\frac{a}{b}\right)$$

where the truncation function $\text{trunc}(x)$ yields the nearest integer value to x that does not exceed x in magnitude. If the resulting value is not representable in T or if b is zero for any pair of corresponding operands, the behavior is undefined.

If T is a *basic floating point scalar*, the result of each division is

$$\text{subround}(\textbf{div}(\text{subround}(a), \text{subround}(b)))$$

where subround applies a *subnormals rounding mode* and the behavior of **div** is determined by the selected *rounding mode*.

If the selected rounding mode is a *precise rounding mode*, then **div** is the IEEE 754 *division*[?] operation with that rounding mode.

If the selected rounding mode is *round approximate*, then $\textbf{div}(x, y)$ performs a fast approximation of division using a multiplication by reciprocal.

If the selected rounding mode is *round full*, then $\textbf{div}(x, y)$ performs a relatively fast approximation of division that has better accuracy across the full input range than the *round approximate* division.

Note: Neither the round approximate nor the round full division is IEEE 754 compliant.

For overloads (1) and (2), the *default rounding mode* and *default subnormals rounding mode* are used. For overload (3), `Mode` specifies the rounding mode and `SubMode` specifies the subnormals rounding mode.

The atomic constraint of overload (3) validates that:

1. T is a *basic floating point scalar*
2. If `Mode` is a *round full* or *round approximate*, then T is `float`.
3. If `SubMode` is *round subnormals to zero*, then T is `float`.
4. The values `Mode` and `SubMode` are enumerators of their respective types.

2.2.7. Ceiling Integer Division

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **`ceildiv`**(*Lhs* dividend, *Rhs* divisor) noexcept;

Performs *elementwise* integral ceiling division of the *arithmetic tile converted* operands `dividend` and `divisor`.

Let a and b denote corresponding elements of the converted operands `dividend` and `divisor` respectively and let T be their type.

The result of each division is

$$\left\lceil \frac{a}{b} \right\rceil$$

If the resulting value is not representable in T or if b is zero for any pair of corresponding operands, the behavior is undefined.

2.2.8. Floor Integer Division

template<ct::*integral_tile* **`Dividend`**, ct::*integral_tile* **`Divisor`**>
requires ct::*arithmetic_tile_convertible*<*Dividend*, *Divisor*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Dividend*, *Divisor*> **`floordiv`**(*Dividend* dividend, *Divisor* divisor) noexcept;

Performs *elementwise* integral floor division of the *arithmetic tile converted* operands `dividend` and `divisor`.

Let a and b denote corresponding elements of the converted operands `dividend` and `divisor` respectively and let T be their type.

The result of each division is

$$\left\lfloor \frac{a}{b} \right\rfloor$$

If the resulting value is not representable in T or if b is zero for any pair of corresponding operands, the behavior is undefined.

2.2.9. Remainder

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **operator%**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **remainder**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* remainder on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively and let T be their type.

If T is an *integral scalar*, each remainder is computed as

$$a - \text{trunc}\left(\frac{a}{b}\right) * b$$

where the truncation function trunc(x) yields the nearest integer value to x that does not exceed x in magnitude. The behavior is undefined if b is zero or if trunc $\left(\frac{a}{b}\right)$ is not representable in T for any pair of corresponding elements in the converted operands.

If T is a *basic floating point scalar*, each remainder is computed as follows:

$$a - \text{trunc}\left(\frac{a}{b}\right) \cdot b$$

where the truncation function trunc(x) yields the nearest integer value to x that does not exceed x in magnitude. If the resulting value is 0, the sign shall match the sign of a. If b is zero the result is NaN.

Non finite values for a or b are handled as follows:

1. If either a or b is NaN, the result is NaN.
2. If a is infinite and b is finite, the result is NaN.
3. If a is finite and b is infinite, the result is a.
4. If a and b are both infinite, the behavior is unspecified.

Note: The C++ STL function `std::remainder` on integral arguments performs a floating point remainder whereas `ct::remainder` performs integral remainder in this case.

2.2.10. Comparison Operators

template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_comparable*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator==**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_comparable*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator!=**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_comparable*<*Lhs*, *Rhs*>

__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator<**(*Lhs* lhs, *Rhs* rhs) noexcept;
template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_comparable*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator<=**(*Lhs* lhs, *Rhs* rhs) noexcept;
template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_comparable*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator>**(*Lhs* lhs, *Rhs* rhs) noexcept;
template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_comparable*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator>=**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* comparison of the *arithmetic comparison converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively and let T be their type.

If T is an *integral scalar*, the result of the comparison is one of the following mathematical predicates according to the selected operator:

Operator	Result
==	$a = b$
!=	$a \neq b$
<	$a < b$
<=	$a \leq b$
>	$a > b$
>=	$a \geq b$

If T is a *basic floating point scalar*, the result of each comparison is one of the following IEEE 754 comparison predicates[4] determined according to the operator:

Operator	Result
==	compareQuietEqual(a, b)
!=	compareQuietNotEqual(a, b)
<	compareSignalingLess(a, b)
<=	compareSignalingLessEqual(a, b)
>	compareSignalingGreater(a, b)
>=	compareSignalingGreaterEqual(a, b)

Note: For all the above predicates except compareQuietNotEqual, a `NaN` compares `false` to all other values. For details, refer to the IEEE 754 comparison predicates[4].

[4] See *Details of comparison predicates* § 5.11 of IEEE 754-2019

2.2.11. Bitwise And

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **operator&**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* bitwise AND on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands, and let a_i and b_i be the i^{th} bit of the *base two representation* of a and b respectively. The result of each computation is determined by:

$$r_i = \text{AND}(a_i, b_i)$$

where r_i is the i^{th} bit of the base two representation of the result and $\text{AND}(x, y)$ is defined by the following truth table:

x	y	$\text{AND}(x, y)$
0	0	0
0	1	0
1	0	0
1	1	1

2.2.12. Bitwise Or

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **operator|**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* bitwise OR on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands, and let a_i and b_i be the i^{th} bit of the *base two representation* of a and b respectively. The result of each computation is determined by:

$$r_i = \text{OR}(a_i, b_i)$$

where r_i is the i^{th} bit of the base two representation of the result and $\text{OR}(x, y)$ is defined by the following truth table:

x	y	$\text{OR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	1

2.2.13. Bitwise Xor

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **operator^**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* bitwise XOR on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands, and let a_i and b_i be the i^{th} bit of the *base two representation* of a and b respectively. The result of each computation is determined by:

$$r_i = \text{XOR}(a_i, b_i)$$

where r_i is the i^{th} bit of the base two representation of the result and $\text{XOR}(x, y)$ is defined by the following truth table:

x	y	$\text{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

2.2.14. Bitwise Complement

template<ct::*integral_tile* **T**>
requires /* atomic constraint */
__tile__ *T* **operator~**(*T* in) noexcept;

Performs *elementwise* complement on the operand `in`.

Let a be an element of `in` and let a_i be the i^{th} bit of the *base two representation* of a. The i^{th} bit r_i in the base two representation of the each result is determined as

$$r_i = \begin{cases} 1 & \text{if} \quad a_i = 0 \\ 0 & \text{if} \quad a_i = 1 \end{cases}$$

The atomic constraint validates that the *element type* of T is not `bool`.

2.2.15. Left Bitshift

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, ct::*tile_element_t*<*Lhs*>> **operator<<**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* left bitshift of the *mutual broadcast converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively. Let T be the type of a and N its *bitwidth*.

The result of the operation is the unique integer representable in T which is congruent to $a \cdot 2^b$ modulo 2^N.

If any element of the converted `rhs` is negative, the behavior is undefined. If any element of the converted `rhs` is greater than or equal to the N, the behavior is undefined.

The atomic constraint validates that T is not `bool`.

2.2.16. Right Bitshift

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, ct::*tile_element_t*<*Lhs*>> **operator>>**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* right bitshift of the *mutual broadcast converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively. Let T be the type of a and N its *bitwidth*.

The result of each operation is

$$\left\lfloor \frac{a}{2^b} \right\rfloor$$

If any element of the converted `rhs` is negative, the behavior is undefined. If any element of the converted `rhs` is greater than or equal to the N, the behavior is undefined.

The atomic constraint validates that T is not `bool`.

2.2.17. Logical Conjunction

template<ct::*bool_tile_convertible* **Lhs**, ct::*bool_tile_convertible* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator&&**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* logical conjunction of the *mutual broadcasted bool tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands. The result of each operation is determined by the following table:

a	b	**Result**
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

2.2.18. Logical Disjunction

template<ct::*bool_tile_convertible* **Lhs**, ct::*bool_tile_convertible* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*>
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator||**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise* logical disjunction of the *mutual broadcasted bool tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands. The result of each operation is determined by the following table:

a	b	**Result**
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

2.2.19. Logical Negation

template<ct::*bool_tile_convertible* **T**>
__tile__ ct::*tile_with_element_t*<*T*, bool> **operator!**(*T* x) noexcept;

Performs *elementwise* logical negation on the *bool tile converted* operand x.

Let a be an element of x. The result of each operation is

$$\begin{cases} \text{false} & \text{if a is true} \\ \text{true} & \text{if a is false} \end{cases}$$

2.2.20. Maximum

template<
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **max**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<
ct::*nan_propagation_mode* **NanMode** = ct::*default_nan_propagation_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**
>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **max**(*Lhs* lhs, *Rhs* rhs,
ct::*nan_propagation_mode_constant*<*NanMode*>,
ct::*subnormals_rounding_mode_constant*<*SubMode*>
= {}) noexcept;

Performs *elementwise* maximum on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively and let T be their type.

If T is an *integral scalar*, the result of each computation is

$$\begin{cases} a & a > b \\ b & a \leq b \end{cases}$$

If T is a *basic floating point scalar*, the result of each computation is

$$\text{subround}(\textbf{maximum}(a, b))$$

where subround applies a *subnormals rounding mode* and the behavior of **maximum** is determined according to the selected *NaN propagation mode*.

If the selected NaN propagation mode is *suppress NaN*, then **maximum** refers to the IEEE 754 *maximumNumber*[3] function.

If the selected NaN propagation mode is *propagate NaN*, then **maximum** refers to the IEEE 754 *maximum*[3] function.

For overload (1), the *default nan propagation mode* and *default subnormals rounding mode* are selected. For overload (2), the *NaN propagation mode* and *subnormals rounding mode* are determined by `NanMode` and `SubMode` respectively.

The atomic constraint of overload (2) validates that:

1. T is a *basic floating point scalar*
2. If `SubMode` is *round subnormals to zero*, then T is `float`.
3. The values `NanMode` and `SubMode` are enumerators of their respective types.

2.2.21. Minimum

```
template<
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs>
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> min(Lhs lhs, Rhs rhs) noexcept;
template<
ct::nan_propagation_mode NanMode = ct::default_nan_propagation_mode(),
ct::subnormals_rounding_mode SubMode = ct::default_subnormals_rounding_mode(),
ct::arithmetic_tile Lhs,
ct::arithmetic_tile Rhs
>
requires ct::arithmetic_tile_convertible<Lhs, Rhs> && /* atomic constraint */
__tile__ ct::arithmetic_tile_conversion_t<Lhs, Rhs> min(Lhs lhs, Rhs rhs,
                                                        ct::nan_propagation_mode_constant<NanMode>,
                                                        ct::subnormals_rounding_mode_constant<SubMode>
                                                        = {}) noexcept;
```

[3] See *Minimum and maximum operations* § 9.6 of IEEE 754-2019

Performs *elementwise* minimum on the *arithmetic tile converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively and let T be their type.

If T is an *integral scalar*, the result of each computation is

$$\begin{cases} b & a > b \\ a & a \leq b \end{cases}$$

If T is a *basic floating point scalar*, the result of each computation is

$$\text{subround}(\textbf{min}(a, b))$$

where subround applies a *subnormals rounding mode* and the behavior of **min** is determined according to the selected *NaN propagation mode*.

If the selected nan propagation mode is *suppress NaN*, then **min** refers to the IEEE 754 *minimumNumber*[Page 62, 3] function.

If the selected nan propagation mode is *propagate NaN*, then **min** refers to the IEEE 754 *minimum*[Page 62, 3] function.

For overload (1), the *default nan propagation mode* and *default subnormals rounding mode* are selected. For overload (2), the *NaN propagation mode* and *subnormals rounding mode* are determined by `NanMode` and `SubMode` respectively.

The atomic constraint of overload (2) validates that:

1. T is a *basic floating point scalar*
2. If `SubMode` is *round subnormals to zero*, then T is `float`.
3. The values `NanMode` and `SubMode` are enumerators of their respective types.

2.2.22. Absolute Value

template<ct::*arithmetic_tile* **U**>
__tile__ *U* **abs**(*U* x) noexcept;

Yields the *elementwise* absolute value of the operand `x`.

Let a be an element of `x` and T its type.

If T is an *integral scalar*, the result of each computation is absolute value $|a|$. If this value is not representable in T the behavior is undefined.

If T is a *basic floating point scalar*, the result of each computation is

$$\textbf{abs}(a)$$

where **abs** is the IEEE 754 *abs*? function.

2.2.23. Fused Multiply Add

template<
ct::*rounding_mode* **Mode** = ct::*default_rounding_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **Lhs**,
ct::*arithmetic_tile* **Rhs**,
ct::*basic_floating_point_tile* **Acc**
>
requires /* atomic constraint */
__tile__ *Acc* **fma**(*Lhs* lhs, *Rhs* rhs, *Acc* acc, ct::*rounding_mode_constant*<*Mode*> = {},
ct::*subnormals_rounding_mode_constant*<*SubMode*> = {}) noexcept;

Performs *elementwise* fused multiplication-addition of the operands `lhs` `rhs` and `acc`. The `lhs` and `rhs` operands are *broadcast converted* to the *shape* of `acc`, then *tile converted* the type `Acc`.

Let a and b and c be corresponding elements of the converted operands `lhs`, `rhs`, and `acc` respectively. The result of each computation is

$$\text{subround}(\textbf{fusedMultiplyAdd}(\text{subround}(a), \text{subround}(b), \text{subround}(acc)))$$

where **fusedMultiplyAdd** is the IEEE 754 *fusedMultiplyAdd*? operation with *rounding mode* and subround applies a *subnormals rounding mode*. `Mode` specifies the rounding mode and `SubMode` specifies the subnormals rounding mode.

The atomic constraint validates that

1. `lhs` and `rhs` are *broadcast convertible* to the *shape* of `acc`.
2. The *element types* of `lhs` and `rhs` are *non-narrowing scalar convertible* to the *element type* of `acc`.
3. `Mode` is a *precise rounding mode*
4. If `SubMode` is *round subnormals to zero*, then T is `float`.
5. The values `Mode` and `SubMode` are enumerators of their respective types.

2.2.24. Multiply High Bits

template<ct::*integral_tile* **Lhs**, ct::*integral_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **mulhi**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise operation* multiplication of the *arithmetic tile converted* operands `lhs` and `rhs` and yields only the upper N bit result of the $2N$ bit product.

Let a and b be corresponding elements of the converted operands and let T be their type and N their *bitwidth*.

If T is an unsigned *integral scalar*, the result of each computation is

$$\textbf{mulhiUnsigned(a, b)} = \left\lfloor \frac{a \cdot b}{2^N} \right\rfloor$$

This corresponds to multiplication of a and b in an integer type of twice the *bitwidth* of T followed by a N bit right shift.

If T is a signed *integral scalar*, the result of each computation is the unique integer representable in T that is congruent module 2^N to

$$\textbf{mulhiUnsigned}(a \bmod 2^N, b \bmod 2^N)$$

This corresponds to an unsigned mulhi operation where the operands are reinterpreted as unsigned integers.

2.2.25. Pointer No-Op

template<ct::*pointer_tile* **Tile**>
__tile__ *Tile* **operator+**(*Tile* x) noexcept;

Yields the value x without modification.

2.2.26. Pointer Addition

template<ct::*integral_tile* **Offset**, ct::*pointer_tile* **Ptr**>
requires ct::*broadcast_compatible*<*Offset*, *Ptr*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Offset*, *Ptr*, ct::*tile_element_t*<*Ptr*>> **operator+**(*Offset* offset, *Ptr* ptr) noexcept;

template<ct::*pointer_tile* **Ptr**, ct::*integral_tile* **Offset**>
requires ct::*broadcast_compatible*<*Ptr*, *Offset*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Ptr*, *Offset*, ct::*tile_element_t*<*Ptr*>> **operator+**(*Ptr* ptr, *Offset* offset) noexcept;

Performs *elementwise operation* pointer addition of the *mutual broadcast converted* operands `ptr` and `offset`.

Let p and x denote corresponding elements of the converted operands `ptr` and `offset`. The result of each computation is the result of evaluating the C++ pointer addition operation[5] `p + x`.

If this expression would yield undefined behavior for any pair of corresponding elements, the behavior of the operation as a whole is undefined.

The atomic constraint tests that the *element type* of `ptr` is not a pointer to (possibly cv-qualified) `void`.

Note: The builtin C++ pointer addition operation is well defined only if the resulting pointer is valid in the C++ object model. For details see [expr.add] § 7.6.6 of ISO/IEC 14882:2024.

---

[5] See *Additive operators* [expr.add] § 7.6.6 of ISO/IEC 14882:2024

2.2.27. Pointer Subtraction

template<ct::*pointer_tile* **Ptr**, ct::*integral_tile* **Offset**>
requires ct::*broadcast_compatible*<*Ptr*, *Offset*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Ptr*, *Offset*, ct::*tile_element_t*<*Ptr*>> **operator-**(*Ptr* ptr, *Offset* offset) noexcept;

Performs *elementwise operation* pointer subtraction of the *mutual broadcast converted* operands `ptr` and `offset`.

Let p and x denote corresponding elements of the converted operands `ptr` and `offset`. The result of each computation is the result of evaluating the C++ pointer subtraction operation[Page 65, 5] `p - x`.

If this expression would yield undefined behavior for any pair of corresponding elements, the behavior of the operation as a whole is undefined.

The atomic constraint tests that the *element type* of `ptr` is not a pointer to (possibly cv-qualified) `void`.

Note: The builtin C++ pointer subtraction operation is well defined only if the resulting pointer is valid in the C++ object model. For details see [expr.add] § 7.6.6 of ISO/IEC 14882:2024.

2.2.28. Pointer Difference

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, ptrdiff_t> **operator-**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise operation* pointer difference on the *mutual broadcast converted* operands `lhs` and `rhs`.

Let a and b denote corresponding elements of the converted operands `lhs` and `rhs` respectively. The result of each computation is the result of evaluating the C++ pointer difference operation[Page 65, 5] `a - b`.

If this expression would yield undefined behavior for any pair of corresponding elements, the behavior of the operation as a whole is undefined.

The atomic constraint validates that the *element type* of both `lhs` and `rhs` may participate in the builtin difference operation.

Note: The builtin C++ pointer difference operation is well defined only if the arguments are appropriate pointers in the C++ object model. For details see [expr.add] § 7.6.6 of ISO/IEC 14882:2024.

2.2.29. Pointer Comparison

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator==**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator!=**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator<**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator<=**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator>**(*Lhs* lhs, *Rhs* rhs) noexcept;

template<ct::*pointer_tile* **Lhs**, ct::*pointer_tile* **Rhs**>
requires ct::*broadcast_compatible*<*Lhs*, *Rhs*> && /* atomic constraint */
__tile__ ct::*mutual_broadcast_t*<*Lhs*, *Rhs*, bool> **operator>=**(*Lhs* lhs, *Rhs* rhs) noexcept;

Performs *elementwise operation* pointer comparison of the *mutual broadcast converted* operands `lhs` and `rhs`.

Let a and b be corresponding elements of the converted operands `lhs` and `rhs` respectively and let `@` denote the selected overloaded operator. The result of each computation is the result of evaluating the expression `a @ b` which is either a builtin relation operator expression[6] or a builtin equality operator expression[7].

The atomic constraint validates the *element types* of `Lhs` and `Rhs` may participate in the builtin C++ relational and comparison operators.

Note: The result of a the builtin relational and equality comparison operators may be unspecified if the pointer arguments are not appropriately arranged in the C++ object model. For details, see [expr.rel] § 7.6.9 and [expr.eq] § 7.6.10 of ISO/IEC 14882:2024.

2.2.30. Null Pointer Equality

template<ct::*pointer_tile* **Ptrs**>
__tile__ ct::*tile_with_element_t*<*Ptrs*, bool> **operator==**(*Ptrs* ptrs, *nullptr-t*) noexcept;

template<ct::*pointer_tile* **Ptrs**>
__tile__ ct::*tile_with_element_t*<*Ptrs*, bool> **operator==**(*nullptr-t*, *Ptrs* ptrs) noexcept;

Performs *elementwise operation* equality comparison to a null pointer value. Let a be an element of operand `ptrs`.

The result of each computation is `true` if a is a null pointer value and `false` otherwise.

[6] See *Relational operators* [expr.rel] § 7.6.9 of ISO/IEC 14882:2024
[7] See *Equality operators* [expr.eq] § 7.6.10 of ISO/IEC 14882:2024

2.2.31. Null Pointer Inequality

template<ct::*pointer_tile* **Ptrs**>
__tile__ ct::*tile_with_element_t*<*Ptrs*, bool> **operator!=**(*Ptrs* ptrs, *nullptr-t*) noexcept;
template<ct::*pointer_tile* **Ptrs**>
__tile__ ct::*tile_with_element_t*<*Ptrs*, bool> **operator!=**(*nullptr-t*, *Ptrs* ptrs) noexcept;

Performs *elementwise operation* inequality comparison to a null pointer value. Let a be an element of operand `ptrs`.

The result of each computation is `false` if a is a null pointer value and `true` otherwise.

2.3. Tile Operations

This section documents various operations for manipulating *tile like* objects.

2.3.1. cuda::tiles::full

template<ct::*tile_like* **T**>
__tile__ *T* **full**(ct::*tile_element_t*<*T*> x) noexcept;

Yields a *tile like* object of type T whose elements all have value x.

Example

The code `ct::full<ct::tile<int, ct::shape<2, 2>>>(42)` produces the following matrix:

$$\begin{pmatrix} 42 & 42 \\ 42 & 42 \end{pmatrix}$$

2.3.2. cuda::tiles::iota

template<ct::*integral_tile* **T**>
requires /* atomic constraint */
__tile__ *T* **iota**() noexcept;

Yields an *integral tile* of type T whose *row major arrangement* is the sequence $(0, 1, ..., N-1)$ where N is the *tile size* of T.

The atomic constraint validates that $N - 1$ is representable in the *element type* of T.

Example

The code `ct::iota<ct::tile<int, ct::shape<2, 4>>>()` produces the following matrix:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

2.3.3. cuda::tiles::ones

template<ct::*numeric_tile* **T**>
__tile__ *T* **ones**() noexcept;

Yields a *numeric tile* object of type T whose elements all have value 1 or `true` in the case of `bool` elements.

Example

The code `ct::ones<ct::tile<int, ct::shape<2, 2>>>(42)` produces the following matrix:

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

2.3.4. cuda::tiles::zeros

template<ct::*numeric_tile* **T**>
__tile__ *T* **zeros**() noexcept;

Yields a *numeric tile* object of type T whose elements all have value 0 or `false` in the case of `bool` elements.

When T is a *basic floating point tile*, the sign bit of each element of the result shall be positive.

Example

The code `ct::zeros<ct::tile<double, ct::shape<2, 2>>>(42)` produces the following matrix:

$$\begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix}$$

2.3.5. cuda::tiles::isinf

template<ct::*basic_floating_point_tile* **T**>
__tile__ ct::*tile_with_element_t*<*T*, bool> **isinf**(*T* x);

Performs an *elementwise* check for infinities in x.

For each element a of x, the result is `true` if a is positive or negative infinity and `false` otherwise.

Example

The following code checks for infinite values in x:

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;

float xData[4] = {
  1.0  / 0.0,  // positive inf
  -1.0 / 0.0,  // negative inf
  0.0  / 0.0,  // NaN
  1.0,         // finite
};

auto x = ct::load(&xData[0] + ct::iota<i32x4>());
auto r = ct::isinf(x);
```

$$\begin{pmatrix} \infty \\ -\infty \\ \text{NaN} \\ 1.0 \end{pmatrix} \rightarrow \begin{pmatrix} \text{true} \\ \text{true} \\ \text{false} \\ \text{false} \end{pmatrix}$$

2.3.6. cuda::tiles::isnan

template<ct::*basic_floating_point_tile* **T**>
__tile__ ct::*tile_with_element_t*<*T*, bool> **isnan**(*T* x);

Performs an *elementwise* check for NaN values in x.

For each element a of x, the result is `true` if a is not a number and `false` otherwise.

Example

The following code checks for infinite values in x:

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;

float xData[4] = {
  1.0  / 0.0,  // positive inf
  -1.0 / 0.0,  // negative inf
  0.0  / 0.0,  // NaN
  1.0,         // finite
};

auto x = ct::load(&xData[0] + ct::iota<i32x4>());
auto r = ct::isnan(x);
```

$$\begin{pmatrix} \infty \\ -\infty \\ \text{NaN} \\ 1.0 \end{pmatrix} \rightarrow \begin{pmatrix} \text{false} \\ \text{false} \\ \text{true} \\ \text{false} \end{pmatrix}$$

2.3.7. cuda::tiles::reshape

template<ct::*tile_shape* **Shape**, ct::*tile_like* **Tile**>
requires (ct::*tile_size_v*<*Tile*> == ct::*shape_size_v*<*Shape*>)
__tile__ ct::*tile*<ct::*tile_element_t*<*Tile*>, *remove-cv-t*<*Shape*>> **reshape**(*Tile* x, *Shape* = {}) noexcept;

Reshapes argument x to match the shape Shape.

The result is a `ct::tile` object of *shape* Shape whose *row major arrangement* of elements matches the *row major arrangement* of x.

Example

The following example reshapes a 2×4 matrix into a 4×2 matrix:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto x = ct::iota<ct::tile<int, ct::shape<2, 4>>>();
auto y = ct::reshape(x, ct::shape{4_ic, 2_ic});
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix}$$

2.3.8. cuda::tiles::tile_permutation_t

template<ct::*tile_like* **T**, typename **Map**>
requires /* atomic constraint */
using **tile_permutation_t** = /* see below */

Yields the type formed by permuting the dimensions of T according to the *ct::dimension_map* Map.

If T has *rank* less than 2, the result type is T. Otherwise, the result U is a specialization of *ct::tile* satisfying the following:

1. The *element type* and *rank* of U matches that of T.
2. Let P_i be `Map::mapping(i)`, denoting the permuted dimension at index i. The length of U at i is the length of the permuted dimension of T: $U_i = T_{P_i}$.

The atomic constraint validates that:

1. Map is a (possibly cv-qualified) specialization of *ct::dimension_map*.
2. The *rank* of Tile is equal to `Map::rank()`.

Examples

Example 1	Tile	`int`
	Map	`ct::dimension_map<>`
	Result	`int`
Example 2	Tile	`ct::tile<int, ct::shape<>>`
	Map	`ct::dimension_map<>`
	Result	`ct::tile<int, ct::shape<>>`
Example 3	Tile	`ct::tile<int, ct::shape<4, 2, 16, 8>>`
	Map	`ct::dimension_map<2, 1, 3, 0>`
	Result	`ct::tile<int, ct::shape<16, 2, 8, 4>>`

2.3.9. cuda::tiles::permute

template<typename **Map**, ct::*tile_like* **Tile**>
requires /* atomic constraint */
__tile__ ct::*tile_permutation_t*<*Tile*, *Map*> **permute**(*Tile* in, *Map* = {}) noexcept;

Yields the *permutation* of `in` according to the zero-based permutation vector described by the provided *`ct::dimension_map`*.

The atomic constraint validates that:

1. `Map` is a specialization of *`ct::dimension_map`*
2. The *rank* of `Tile` is `Map::rank()`.

Example

The following code permutes a $4 \times 2 \times 2$ tile to a $2 \times 4 \times 2$ tile according to the dimension map `ct::dimension_map<2, 0, 1>`:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
auto x = ct::iota<ct::tile<int, ct::shape<4, 2, 2>>>();
auto r = ct::permute(x, ct::dimension_map{2_ic, 0_ic, 1_ic});
```

$$\begin{pmatrix} \begin{pmatrix}0 & 1\end{pmatrix} & \begin{pmatrix}2 & 3\end{pmatrix} \\ \begin{pmatrix}4 & 5\end{pmatrix} & \begin{pmatrix}6 & 7\end{pmatrix} \\ \begin{pmatrix}8 & 9\end{pmatrix} & \begin{pmatrix}10 & 11\end{pmatrix} \\ \begin{pmatrix}12 & 13\end{pmatrix} & \begin{pmatrix}14 & 15\end{pmatrix} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix}0 & 2\end{pmatrix} & \begin{pmatrix}4 & 6\end{pmatrix} & \begin{pmatrix}8 & 10\end{pmatrix} & \begin{pmatrix}12 & 14\end{pmatrix} \\ \begin{pmatrix}1 & 3\end{pmatrix} & \begin{pmatrix}5 & 7\end{pmatrix} & \begin{pmatrix}9 & 11\end{pmatrix} & \begin{pmatrix}13 & 15\end{pmatrix} \end{pmatrix}$$

The behavior of the permutation is described precisely below:

tile projection

Let a be a *tile like* object of rank N, *shape* S and *index space* $I = [0, S_0) \times [0, S_1) \times ... \times [0, S_{N-1})$.

We may interpret a as a function $a : I \to \mathbb{E}$ describing the association of indices to elements $\mathbb{E}$.

The *projection* of a along dimension $0 \leq d < N$ at index $i \in [0, S_d)$ is a new *tile like* object of index space $I_p = [0, S_0) \times [0, S_1) \times ... \times [0, S_{d-1}] \times [0, 1) \times [0, S_{d+1}] \times ... \times [0, S_{N-1}]$:

$$\text{Proj}(a, i, d) : I_p \to \mathbb{E}$$

whose values correspond to a when dimension d is fixed at index i:

$$\text{Proj}(a, i, d)(x_0, x_1, \ldots, x_{d-1}, 0, x_{d+1}, \ldots x_{N-1}) = a(x_0, x_1, ..., x_{d-1}, i, x_d, ..., x_{N-2})$$

iterated tile projection

The *iterated projection* of a for indices $(i_0, i_1, ..., i_{N-1}) \in I$ at dimensions $(d_0, d_1, ..., d_n)$ is formed by the repeated application of Proj:

$$\text{Proj}(...\text{Proj}(\text{Proj}(a, i_0, d_0), i_1, d_1), ..., i_{N-1}, d_{N-1})$$

The *iterated projection* of a is a singleton tile whose value is the value of a at index i_0 along dimension d_0, i_1 along d_1, etc...

tile permutation

Let $P : [0, N) -> [0, N)$ be the bijective function describing a permutation of dimensions.

The *permutation* r of a is the unique *tile-like* object whose value at indices $(i_0, i_1, ..., i_{N-1})$ is the *iterated projection* of $(i_0, i_1, ..., i_{N-1})$ along dimensions $(P(0), P(1), ..., P(N-1))$.

2.3.10. cuda::tiles::tile_transpose_t

template<ct::*tile_like* **T**>
using **`tile_transpose_t`** = /* see below */;

Yields the result type when interchanging the first two dimensions of an object of type T.

If T has rank less than 2, the resulting type is T. Otherwise, the resulting type U is a specialization of `ct::tile` satisfying the following:

1. The *element type* and *rank* of U matches that of T
2. The lengths of U satisfy:
   1. $U_0 = T_1$
   2. $U_1 = T_0$
   3. $U_i = T_i$ for each $2 \leq i < N$ where N is the rank of T.

Examples

Example 1	Tile	`int`
	Result	`int`
Example 2	Tile	`ct::tile<int, ct::shape<>>`
	Result	`ct::tile<int, ct::shape<>>`
Example 3	Tile	`ct::tile<int, ct::shape<4, 2, 16, 8>>`
	Result	`ct::tile<int, ct::shape<2, 4, 16, 8>>`

2.3.11. cuda::tiles::transpose

template<ct::*tile_like* **Tile**>
__tile__ ct::*tile_transpose_t*<*Tile*> **transpose**(*Tile* in) noexcept;

Interchanges the first two dimensions of `in`.

If the *rank* of `in` is less than 2, `in` is returned unmodified. Otherwise, the resulting tile is produced as if by invoking:

```
ct::permute(src, ct::dimension_map<1, 0, 2, 3, ..., N-1>{});
```

where N is the rank of `src`.

Example

The following code transposes a $4 \times 2 \times 2$ tile to a $2 \times 4 \times 2$ tile:

```
namespace ct = ::cuda::tiles;
auto x = ct::iota<ct::tile<int, ct::shape<4, 2, 2>>>();
auto r = ct::transpose(x);
```

$$\begin{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} & \begin{pmatrix} 2 & 3 \end{pmatrix} \\ \begin{pmatrix} 4 & 5 \end{pmatrix} & \begin{pmatrix} 6 & 7 \end{pmatrix} \\ \begin{pmatrix} 8 & 9 \end{pmatrix} & \begin{pmatrix} 10 & 11 \end{pmatrix} \\ \begin{pmatrix} 12 & 13 \end{pmatrix} & \begin{pmatrix} 14 & 15 \end{pmatrix} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} & \begin{pmatrix} 4 & 5 \end{pmatrix} & \begin{pmatrix} 8 & 9 \end{pmatrix} & \begin{pmatrix} 12 & 13 \end{pmatrix} \\ \begin{pmatrix} 2 & 3 \end{pmatrix} & \begin{pmatrix} 6 & 7 \end{pmatrix} & \begin{pmatrix} 10 & 11 \end{pmatrix} & \begin{pmatrix} 14 & 15 \end{pmatrix} \end{pmatrix}$$

2.3.12. concatenation_compatible

template<typename **T**, typename **U**, size_t **D**>
concept **concatenation_compatible** = ct::*tile_like*<*T*> && ct::*tile_like*<*U*> && /* atomic constraint */;

Indicates whether *tile like* types T and U are concatenation compatible along dimension D.

T and U are concatenation compatible along D if:

1. T and U have the same *rank* and the same *element type*
2. $0 \leq D < N$ where N is the *rank* of T and U
3. The lengths satisfy $T_i = U_i$ for each $0 \leq i < N, \quad i \neq D$.
4. The shape S formed by $S_i = T_i$ for $i \neq D$ and $S_d = T_d + U_d$ is a *tile compatible shape*.

Note: Rank 0 tiles are never concatenation compatible.

2.3.13. cuda::tiles::concatenation_t

template<ct::*tile_like* **T**, ct::*tile_like* **U**, size_t **Dim**>
requires ct::*concatenation_compatible*<*T*, *U*, *Dim*>
using **concatenation_t** = /* see below */;

Yields the result of concatenating *tile like* types T and U along dimension D.

The result type is a specialization of *ct::tile* whose *element type* and *rank* matches that of T and U and whose shape S satisfies:

1. $S_i = T_i$ for $0 \leq i < N, \quad i \neq D$ where N is the rank of T and U.
2. $S_D = T_D + U_D$

Example

Example 1	T	`int`
	U	`int`
	Dimension	0
	Result	Incompatible
Example 2	T	`ct::tile<int, ct::shape<2, 4>`
	U	`ct::tile<int, ct::shape<2, 4>`
	Dimension	0
	Result	`ct::tile<int, ct::shape<4, 4>>`
Example 3	T	`ct::tile<int, ct::shape<2, 4>`
	U	`ct::tile<int, ct::shape<2, 4>`
	Dimension	1
	Result	`ct::tile<int, ct::shape<2, 8>>`

2.3.14. cuda::tiles::cat

template<ct::*integral* auto **D**, ct::*tile_like* **T**, ct::*tile_like* **U**>
requires (*D* >= 0) && ct::*concatenation_compatible*<*T*, *U*, *D*>
__tile__ ct::*concatenation_t*<*T*, *U*, *D*> **cat**(*T* x, *U* y, ct::*integral_constant*<*D*> = {}) noexcept;

Concatenates x and y along dimension D.

Let N be the *rank* of T and U. The resulting tile object r has values defined below

$$r(i_0, i_1, ..., i_D, ..., i_{N-1}) = \begin{cases} x(i_0, i_1, ..., i_D, ..., i_{N-1}) & i_D < T_D \\ y(i_0, i_1, ..., i_D - T_D, ..., i_{N-1}) & \text{otherwise} \end{cases}$$

Example

The following code concatenates a matrix of zeros with a matrix of ones along the second dimension.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
auto x = ct::full<ct::tile<int, ct::shape<4, 2>>>(0);
auto y = ct::full<ct::tile<int, ct::shape<4, 2>>>(1);
auto r = ct::cat(x, y, 1_ic);
```

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

2.3.15. cuda::tiles::select

template<ct::*tile_like* **T**, ct::*bool_tile_convertible* **C**>
requires ct::*broadcastable_to*<*C*, ct::*tile_shape_t*<*T*>>
__tile__ *T* **select**(*C* condition, *T* lhs, *T* rhs) noexcept;

Performs an *elementwise* selection of the values from `lhs` where `condition` is `true` and `rhs` where `condition` is `false`.

`condition` undergoes *bool tile conversion* followed by *broadcast conversion* to match the shape of T.

Let a, b and c denote corresponding elements of the converted `lhs`, `rhs` and `condition` arguments. The result of each selection is a if c is `true` and b otherwise.

Example

In the following example, the even elements of the result are derived from x while the odd elements are derived from y:

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;
bool cData[4] = {true, false, true, false};

auto c = ct::load(&cData[0] + ct::iota<i32x4>());
auto x = ct::iota<i32x4>();
auto y = -ct::iota<i32x4>();

auto r = ct::select(c, x, y);
```

$$\begin{pmatrix} \text{true} \\ \text{false} \\ \text{true} \\ \text{false} \end{pmatrix} \quad \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \quad \begin{pmatrix} 0 \\ -1 \\ -2 \\ -3 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ -1 \\ 2 \\ -3 \end{pmatrix}$$

2.3.16. cuda::tiles::extractable_from

template<typename **S**, typename **T**>
concept **`extractable_from`** = ct::*tile_shape*<*S*> && ct::*tile_like*<*T*> && /* see below */

Determines if *tile compatible shape* S is an extractable shape of T. S is extractable from T if:

1. S and T have the same *rank* denoted N.
2. For each i in $0 \leq i < N$, T_i is divisible by S_i.

Example

Example 1	T	`int`
	S	`ct::shape<>`
	Extractible	Yes
Example 2	T	`ct::tile<int, ct::shape<32, 8>>`
	S	`ct::shape<16, 2>`
	Extractible	Yes
Example 3	T	`ct::tile<int, ct::shape<32, 8>>`
	S	`ct::shape<2, 16>`
	Extractible	No

2.3.17. cuda::tiles::extract

template<ct::*tile_shape* **S**, ct::*tile_like* **T**, typename ...**`Indices`**>
requires ct::*extractable_from*<*S*, *T*> && /* atomic constraint */
__tile__ ct::*tile*<ct::*tile_element_t*<*T*>, *S*> **`extract`**(*T* x, *S*, *Indices*... idx) noexcept;

Extracts a partition of shape S from *tile like* object `x`.

The elements of `x` are partitioned into equally sized sections of shape S and each partition is assigned a zero-based index. The `idx` argument specifies the index of the partition that will be returned.

Let N be the *rank* of T and let $i_0, i_1, \ldots, i_{N-1}$ be the values of the `idx` pack after conversion to `size_t`. The resulting extraction r has the following value:

$$r(k_0, k_1, \ldots, k_{N-1}) = x(i_0 \cdot S_0 + k_0, i_1 \cdot S_1 + k_1, \ldots, i_{N-1} \cdot S_{N-1} + k_{N-1})$$

If the index $I = (i_0 \cdot S_0, \ldots i_0 \cdot S_0)$ is not in the *index space* of `x`, the behavior is undefined.

The atomic constraint validates that:

1. The size of the parameter pack `Indices` is N
2. For each type U in the parameter pack `Indices`, `is-convertible-v<U, size_t>` holds.

Example

In the following code, the partition in top right quadrant of the matrix is extracted:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
auto x = ct::iota<ct::tile<int, ct::shape<4, 4>>>();
auto r = ct::extract(x, ct::shape{2_ic, 2_ic}, 0, 1);
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix}$$

2.3.18. cuda::tiles::broadcast

template<ct::*tile_shape* **S**, ct::*tile_like* **T**>
requires ct::*broadcastable_to*<*T*, *S*>
__tile__ ct::*tile*<ct::*tile_element_t*<*T*>, *S*> **broadcast**(*T* x, *S* = {}) noexcept;

Yields the *broadcast conversion* of x to shape S.

Example

The following code broadcasts a 4×1 tile to a 4×4 tile:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
auto x = ct::iota<ct::tile<int, ct::shape<4, 1>>>();
auto r = ct::broadcast(x, ct::shape{4_ic, 4_ic});
```

$$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

2.3.19. cuda::tiles::element_cast

template<ct::*scalar* **E**, ct::*tile_like* **T**>
requires ct::*tile_convertible_to*<*T*, ct::*tile_with_element_t*<*T*, *E*>>
__tile__ ct::*tile_with_element_t*<*T*, *E*> **element_cast**(*T* x) noexcept;

Yields the *tile conversion* of x to the type `ct::tile_with_element_t<T, E>`.

Example

The following code converts a tile of integers to a tile of doubles:

```
namespace ct = ::cuda::tiles;
auto x = ct::iota<ct::tile<int, ct::shape<4, 1>>>();
auto r = ct::element_cast<double>(x);
```

$$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \rightarrow \begin{pmatrix} 0.0 \\ 1.0 \\ 2.0 \\ 3.0 \end{pmatrix}$$

2.3.20. cuda::tiles::element_bitcast

template<ct::*scalar* **E**, ct::*tile_like* **T**>
requires /* atomic constraint */
__tile__ ct::*tile_with_element_t*<*T*, *E*> **element_bitcast**(*T* x) noexcept;

Yields the *elementwise* bitcast of x to the type E.

For each element a in x, the result of the computation is produced as if by executing `std::bitcast<E>(a)`[1].

The atomic constraint validates that the object size of E is the same as the object size of the *element type* of T.

Note: The result of this operation depends on the value and object representations of the source and result types, and may generate undefined behavior, see § 22.15.3 of ISO/IEC 14882:2024 for details.

Example

```
namespace ct = ::cuda::tiles;
auto x = ct::full<ct::tile<unsigned char, ct::shape<4, 1>>>(255);
auto r = ct::element_bitcast<signed char>(x);
```

$$\begin{pmatrix} 255 \\ 255 \\ 255 \\ 255 \end{pmatrix} \rightarrow \begin{pmatrix} -1 \\ -1 \\ -1 \\ -1 \end{pmatrix}$$

[1] See *Function template bit_cast* [bit.cast] § 22.15.3 of ISO/IEC 14882:2024

2.4. Math Operations

The APIs in this section provide *elementwise* mathematical computations on *tile like* arguments. Each function provides an approximation of an infinitely precise mathematical result.

The behavior of these APIs is not yet exhaustively specified and may not follow IEEE 754 semantics. Information about the error bounds, rounding modes, and the behavior of non-finite, signed zero, and subnormal arguments may be absent.

Warning: Some of the mathematical APIs may be modified to accept *rounding mode* and *subnormals rounding mode* template arguments in future releases. This change may break existing code that uses explicit template arguments in math API invocations.

To maintain compatibility, avoid specifying an explicit template argument when invoking the math functions:

```
ct::exp(0.0);          // Preferred syntax
ct::exp<double>(0.0); // Discouraged syntax
```

2.4.1. cuda::tiles::ceil

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **ceil**(*Tile* in) noexcept;

Performs *elementwise* ceiling on the operand `in`. For each element x in operand `in`, the result of the computation is $\lceil x \rceil$.

2.4.2. cuda::tiles::floor

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **floor**(*Tile* in) noexcept;

Performs *elementwise* floor on the operand `in`. For each element x in operand `in`, the result of the computation is $\lfloor x \rfloor$.

2.4.3. cuda::tiles::pow

template<ct::*basic_floating_point_tile* **B**, ct::*basic_floating_point_tile* **E**>
requires ct::*arithmetic_tile_convertible*<*B*, *E*>
__tile__ ct::*arithmetic_tile_conversion_t*<*B*, *E*> **pow**(*B* base, *E* exponent) noexcept;

Performs *elementwise* exponentiation on the *arithmetic tile converted* operands `base` and `exponent`.

Let a and b be corresponding elements of the converted operands `base` and `exponent` respectively.

The result of each computation is a^b.

2.4.4. cuda::tiles::exp2

template<
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*basic_floating_point_tile* **Tile**
>
requires /* atomic constraint */
__tile__ *Tile* **exp2**(*Tile* in, ct::*subnormals_rounding_mode_constant*<*SubMode*> = {}) noexcept;

Performs *elementwise* base two exponentiation of operand `in`.

For each element x in operand `in`, the result of the computation is

$$\text{subround}(2^{\text{subround}(x)})$$

where subround applies a *subnormals rounding mode* as determined by the `SubMode` template argument.

The atomic constraint validates that:

1. If `SubMode` is *round subnormals to zero*, then T is `float`.
2. The value `SubMode` is an enumerator of `ct::subnormals_rounding_mode`.

2.4.5. cuda::tiles::exp

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **exp**(*Tile* in) noexcept;

Performs *elementwise* base e exponentiation on operand `in`. For each element x of `in`, the result of the computation is e^x.

2.4.6. cuda::tiles::log2

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **log2**(*Tile* in) noexcept;

Performs *elementwise* base 2 logarithm on operand `in`. For each element x of `in`, the result of the computation is $\log_2(x)$.

2.4.7. cuda::tiles::log

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **log**(*Tile* in) noexcept;

Performs *elementwise* natural logarithm on operand `in`. For each element x of `in`, the result of the computation is $\ln(x)$.

2.4.8. cuda::tiles::sqrt

template<
ct::*rounding_mode* **Mode** = ct::*default_rounding_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*basic_floating_point_tile* **Tile**
>
requires /* atomic constraint */
__tile__ *Tile* **sqrt**(*Tile* x, ct::*rounding_mode_constant*<*Mode*> = {},
ct::*subnormals_rounding_mode_constant*<*SubMode*> = {}) noexcept;

Performs *elementwise* square root on operand `in`.

For each element x of `in`, the result of the computation is

$$\text{subround}\left(\sqrt{\text{subround}(x)}\right)$$

where subround applies a *subnormals rounding mode* as specified by SubMode.

The atomic constraint validates that:

1. Mode is a *precise rounding mode* or *round approximate*.
2. If Mode is *round approximate*, then T is float.
3. If SubMode is *round subnormals to zero*, then T is `float`.
4. The value Mode and SubMode are enumerators of their respective types.

2.4.9. cuda::tiles::rsqrt

template<
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*basic_floating_point_tile* **Tile**
>
requires /* atomic constraint */
__tile__ *Tile* **rsqrt**(*Tile* in, ct::*subnormals_rounding_mode_constant*<*SubMode*> = {}) noexcept;

Performs *elementwise* reciprocal square root on operand `in`. For each element x of `in`, the result of the computation is

$$\text{subround}\left(\frac{1}{\sqrt{\text{subround}(x)}}\right)$$

where subround applies a *subnormals rounding mode* as specified by SubMode.

The atomic constraint validates that:

1. If SubMode is *round subnormals to zero*, then T is `float`.
2. The value SubMode is an enumerator of `ct::subnormals_rounding_mode`.

2.4.10. cuda::tiles::cosh

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **cosh**(*Tile* in) noexcept;

Performs *elementwise* hyperbolic cosine on the input `in`. For each element x of `in`, the result of the computation is $\cosh(x)$.

2.4.11. cuda::tiles::cos

template<ct::*basic_floating_point_tile* **Tile**>
__tile__ *Tile* **cos**(*Tile* in) noexcept;

Performs *elementwise* cosine on the input `in`. For each element x of `in`, the result of the computation is $\cos(x)$.

2.4.12. cuda::tiles::sinh

template<ct::*basic_floating_point_tile*>
__tile__ Tile **sinh**(Tile in) noexcept;

Performs *elementwise* hyperbolic sine on the input `in`. For each element x of `in`, the result of the computation is $\sinh(x)$.

2.4.13. cuda::tiles::sin

template<ct::*basic_floating_point_tile*>
__tile__ Tile **sin**(Tile in) noexcept;

Performs *elementwise* sine on the input `in`. For each element x of `in`, the result of the computation is $\sin(x)$.

2.4.14. cuda::tiles::tanh

template<ct::*basic_floating_point_tile*>
__tile__ Tile **tanh**(Tile in) noexcept;

Performs *elementwise* hyperbolic tangent on the input `in`. For each element x of `in`, the result of the computation is $\tanh(x)$.

2.4.15. cuda::tiles::tan

template<ct::*basic_floating_point_tile*>
__tile__ Tile **tan**(Tile in) noexcept;

Performs *elementwise* tangent on the input `in`. For each element x of `in`, the result of the computation is $\tan(x)$.

2.4.16. cuda::tiles::atan2

template<ct::*arithmetic_tile* **Lhs**, ct::*arithmetic_tile* **Rhs**>
requires ct::*arithmetic_tile_convertible*<*Lhs*, *Rhs*> &&
ct::*basic_floating_point_tile*<ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*>>
__tile__ ct::*arithmetic_tile_conversion_t*<*Lhs*, *Rhs*> **atan2**(*Lhs* y, *Rhs* x) noexcept;

Performs *elementwise* arctangent for ratio of the *arithmetic tile converted* operands y and x.

Let a and b be corresponding elements of the converted operands y and x respectively. The result of each computation is the angle between the positive x axis and the ray connecting the origin and the coordinate (a, b).

2.5. Matrix Multiplication

Matrix multiplication of tiles is supported through the *`ct::matmul()`* and *`ct::mma()`* functions. The `ct::mma` function accepts an accumulator argument that is added to the multiplication result. These functions can perform batched matrix multiplies along a third dimension when the arguments have rank 3.

The result of the matrix multiplication operations is an approximation of an infinitely precise mathematical computation. The exhaustive numerical behavior for these operation is not currently specified. Information about the error bounds, rounding modes, and the behavior of non-finite, signed zero, and subnormal arguments may be absent.

Additionally, there are unspecified scenarios wherein overflow may occur while performing matrix multiplication of integral operands. In these scenarios, the result of the operation is unspecified.

2.5.1. cuda::tiles::mma_compatible

template<typename **L**, typename **R**, typename **A**>
concept **mma_compatible** = ct::*numeric_tile*<*L*> && ct::*numeric_tile*<*R*> && ct::*numeric_tile*<*A*> && /* atomic constraint */;

Indicates whether the types L (the left matrix), R (the right matrix) and A (the accumulator matrix) can participate in a matrix multiply accumulate operation.

The types L, R and A can participate in a matrix multiply accumulate if

1. The *element types* of L and R are *integral scalars* of the same *bitwidth* or they are *floating point scalars* of the same conversion rank

2. The *element types* of L, R and A are are selected from a row in the following table:

Element Type of L and R	**Element type of A**
8 bit integral types	32 bit signed integral types
▶ `__nv_fp8_e4m3` ▶ `__nv_fp8_e5m2` ▶ `__half`	▶ `__half` ▶ `float`
▶ `__nv_bfloat16` ▶ `__nv_tf32` ▶ `float`	▶ `float`
▶ `double`	▶ `double`

3. All of L, R and A have the same *rank* and the rank is either 2 or 3.
4. In the case where the rank is 2, the *shapes* of L, R and A are taken from the table for some integers N, K and M:

Shape of L	**Shape of R**	**Shape of A**
$N \times K$	$K \times M$	$N \times M$

5. In the case where the rank is 3, the *shapes* of L, R and A are taken from the table for some integers A, B, C, N, K and M:

Shape of L	**Shape of R**	**Shape of A**
$A \times N \times K$	$B \times K \times M$	$C \times N \times M$

Additionally A and B satisfy the following predicates:

$$\begin{cases} A = C \\ A = 1 \end{cases} \text{ or } \qquad \begin{cases} B = C \\ B = 1 \end{cases} \text{ or}$$

2.5.2. cuda::tiles::matmul_compatible

template<typename **L**, typename **R**>
concept **`matmul_compatible`** = ct::*numeric_tile*<Lhs> && ct::*numeric_tile*<Rhs> && /* atomic constraint */;

Indicates whether the types L and R may participate in a matrix multiply operation.

The types L and R may participate in a matrix multiply operation if:

1. The element types of L and R are *integral scalars* whose *bitwidth* is 8 or they are *floating point scalars* of the same conversion rank.
2. L and R have the same *rank* and that rank is either 2 or 3.
3. In the case where the rank is 2, the *shapes* of L and R are taken from the table below for some integers N, K and M:

Shape of L	**Shape of** R
$N \times K$	$K \times M$

   The hypothetical `ct::extents` specialization of dimensions $N \times M$ must be *tile compatible*.
4. In the case where the rank is 3, the *shapes* of L and R are taken from the table below for some integers A, B, N, K and M:

Shape of L	**Shape of** R
$A \times N \times K$	$B \times K \times M$

   Additionally, either $A = B$ or at least one of them is equal to 1. The hypothetical `ct::extents` specialization whose dimensions are $\max(A, B) \times N \times M$ must be *tile compatible*.

2.5.3. cuda::tiles::matmul_result_t

template<ct::*numeric_tile* **L**, ct::*numeric_tile* **R**>
requires ct::*matmul_compatible*<*L*, *R*>
using **`matmul_result_t`** = /* see below */;

Yields the result type of performing matrix multiplication on operands of types L and R

The result type is a a specialization of `ct::tile`. The *element type* of the result is determined by the element types of L and R according to the following table:

L **and** R **element type**	**Result Element Type**
8 bit integral types	`int32_t`
▶ `__nv_fp8_e4m3` ▶ `__nv_fp8_e5m2` ▶ `__half`	`__half`
▶ `__nv_bfloat16` ▶ `__nv_tf32` ▶ `float`	`float`
▶ `double`	`double`

The shape of the result depends on the *rank* of L and R:

1. If L and R have rank is 2 with shapes $N \times K$ and $K \times M$ respectively, the result shape is $N \times M$.
2. If L and R have rank 3 with shapes $A \times N \times K$ and $B \times K \times M$, the result shape is $C \times N \times M$ where C is the larger of A and B.

Note: The result element type for matmul cannot be configured. To use a different element type, use `ct::mma()` with a zeroed accumulator.

2.5.4. cuda::tiles::mma

template<ct::*numeric_tile* **L**, ct::*numeric_tile* **R**, ct::*numeric_tile* **A**>
requires ct::*mma_compatible*<*L*, *R*, *A*>
__tile__ *A* **mma**(*L* lhs, *R* rhs, *A* acc) noexcept;

Performs matrix multiply on `lhs` and `rhs` and adds the result to `acc`. For *rank* 3 arguments, the matrix multiply accumulate is performed for each set of matrices along the first dimension of the operands.

If the operands have rank 3, the `lhs` undergoes *broadcast conversion* to the shape $A_0 \times L_1 \times L_2$ and `rhs` undergoes *broadcast conversion* to the shape $A_0 \times R_1 \times R_2$.

Let a, b and c denote the converted operands `lhs`, `rhs`, and `acc` respectively.

For rank 2 arguments, the result is an approximation of the matrix r determined by

$$r(i_0, i_1) = \sum_{k=0}^{k<L_1} a(i_0, k) \cdot b(k, i_1) + c(i_0, i_1)$$

For rank 3 arguments, the result is an approximation of the matrix r determined by

$$r(i_0, i_1, i_2) = \sum_{k=0}^{k<L_2} a(i_0, i_1, k) \cdot b(i_0, k, i_2) + c(i_0, i_1, i_2)$$

Example

The following example shows a matrix multiply accumulate computation for rank 2 arguments:

```
namespace ct = ::cuda::tiles;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;
using i32x4x2 = ct::tile<int, ct::shape<4, 2>>;
using i32x2x2 = ct::tile<int, ct::shape<2, 2>>;

auto lhs = ct::element_cast<float>(ct::iota<i32x2x4>());
auto rhs = ct::element_cast<float>(ct::iota<i32x4x2>());
auto acc = ct::element_cast<float>(ct::iota<i32x2x2>());

auto r = ct::mma(lhs, rhs, acc);
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 28 & 35 \\ 78 & 101 \end{pmatrix}$$

Example

The following example shows a batched matrix multiply accumulate computation for rank 3 arguments:

```
namespace ct = ::cuda::tiles;
using i32x1x2x4 = ct::tile<int, ct::shape<1, 2, 4>>;
using i32x1x4x2 = ct::tile<int, ct::shape<1, 4, 2>>;
using i32x1x2x2 = ct::tile<int, ct::shape<1, 2, 2>>;

auto lhs = ct::element_cast<float>(
            ct::cat<0>(ct::iota<i32x1x2x4>(), ct::iota<i32x1x2x4>()));
auto rhs = ct::element_cast<float>(
            ct::cat<0>(ct::iota<i32x1x4x2>(), -ct::iota<i32x1x4x2>()));
auto acc = ct::element_cast<float>(
            ct::cat<0>(ct::iota<i32x1x2x2>(), -ct::iota<i32x1x2x2>()));

auto r = ct::mma(lhs, rhs, acc);
```

$$\begin{pmatrix} \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \\ \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \end{pmatrix} \times \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix} \\ \begin{pmatrix} 0 & -1 \\ -2 & -3 \\ -4 & -5 \\ -6 & -7 \end{pmatrix} \end{pmatrix} + \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \\ \begin{pmatrix} 0 & -1 \\ -2 & -3 \end{pmatrix} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} 28 & 35 \\ 78 & 101 \end{pmatrix} \\ \begin{pmatrix} -28 & -35 \\ -78 & -101 \end{pmatrix} \end{pmatrix}$$

2.5.5. cuda::tiles::matmul

template<ct::*numeric_tile* **L**, ct::*numeric_tile* **R**>
requires ct::*matmul_compatible*<*L*, *R*>
__tile__ ct::*matmul_result_t*<*L*, *R*> **matmul**(*L* lhs, *R* rhs) noexcept;

Performs matrix multiply on `lhs` and `rhs`. For *rank* 3 arguments, the matrix multiply is performed for each set of matrices along the first dimension of the operands.

If the operands have rank 3, the `lhs` undergoes *broadcast conversion* to the shape $\max(L_0, R_0) \times L_1 \times L_2$ and `rhs` undergoes *broadcast conversion* to the shape $\max(L_0, R_0) \times R_1 \times R_2$.

Let a and b denote the converted operands `lhs` and `rhs` respectively.

For rank 2 arguments, the result is an approximation of the matrix r determined by

$$r(i_0, i_1) = \sum_{k=0}^{k<L_1} a(i_0, k) \cdot b(k, i_1)$$

For rank 3 arguments, the result is an approximation of the matrix r determined by

$$r(i_0, i_1, i_2) = \sum_{k=0}^{k<L_2} a(i_0, i_1, k) \cdot b(i_0, k, i_2)$$

Example

The following example shows a matrix multiply computation for rank 2 arguments:

```
namespace ct = ::cuda::tiles;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;
using i32x4x2 = ct::tile<int, ct::shape<4, 2>>;

auto lhs = ct::element_cast<float>(ct::iota<i32x2x4>());
auto rhs = ct::element_cast<float>(ct::iota<i32x4x2>());

auto r = ct::matmul(lhs, rhs);
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 28 & 34 \\ 76 & 98 \end{pmatrix}$$

Example

The following example shows a batched matrix multiply computation for rank 3 arguments:

```
namespace ct = ::cuda::tiles;
using i32x1x2x4 = ct::tile<int, ct::shape<1, 2, 4>>;
using i32x1x4x2 = ct::tile<int, ct::shape<1, 4, 2>>;

auto lhs = ct::element_cast<float>(ct::cat<0>(ct::iota<i32x1x2x4>(), ct::iota
↪<i32x1x2x4>()));
auto rhs = ct::element_cast<float>(ct::cat<0>(ct::iota<i32x1x4x2>(), -ct::iota
↪<i32x1x4x2>()));

auto r = ct::matmul(lhs, rhs);
```

$$\left(\begin{pmatrix}0&1&2&3\\4&5&6&7\end{pmatrix}\right) \times \left(\begin{pmatrix}0&1\\2&3\\4&5\\6&7\end{pmatrix}\right) \to \left(\begin{pmatrix}28&34\\76&98\end{pmatrix}\right)$$

$$\left(\begin{pmatrix}0&1&2&3\\4&5&6&7\end{pmatrix}\right) \times \left(\begin{pmatrix}0&-1\\-2&-3\\-4&-5\\-6&-7\end{pmatrix}\right) \to \left(\begin{pmatrix}-28&-34\\-76&-98\end{pmatrix}\right)$$

2.6. Reductions and Scans

A *reduction* on a *tile like* object along a dimension is the result of repeatedly accumulating the elements of that dimension into a single value according to a binary operation. The resulting tile contains the accumulated result of each vector along the operating dimension. A *scan* is similar to a reduction except it preserves partial accumulation results in the final tile.

2.6.1. Definitions

In the following definitions, let a be a *tile like* object of *rank* N, *shape* S, and *element type* E. Let $0 \leq d < N$ be a dimension of a and op : $E \times E \to E$ denote a binary operation with identity element id of *scalar* type E.

tile reduction

A *reduction* of a along d for operation op is any *tile like* object r satisfying the following constraints.

The shape R of r matches S except at dimension d where its length is 1:

$$R_k = \begin{cases} S_k & k \neq d \\ 1 & k = d \end{cases}$$

Let

$$I = (i_0, i_1, ..., i_{d-1}, 1, i_{d+1}, \ldots, i_{N-1})$$

be an element in the *index space* of r. Consider a collection consisting of the elements of a along d at I:

$$V = \{a(i_0, i_1, ..., i_{d-1}, j, i_{d+1}, \ldots, i_{N-1}) \quad | \quad 0 \leq j < S_d\}$$

Let $v = (v_0, v_1, \ldots, v_k)$ be any sequence consisting of the elements of V and 0 or more instances of id.

The value of r at I is any application of op to the sequence v:

$$r(I) = v_0 \text{ op } v_1 \text{ op} \ldots \text{op } v_k$$

where the grouping of the application of op and the ordering of the elements of v is unspecified.

If op would incur undefined behavior on any possible grouping or ordering of the elements of v, the computation of the reduction as a whole is undefined behavior.

Note: If op is not commutative or not associative or if id is not a true identity element, there may be multiple possible reductions of a.

If a particular grouping or ordering of v is observed at index I in the result, this does not imply any constraint on the grouping or ordering of v for any other index of r.

---

tile scan

A *scan* of a along d for operation op is any *tile like* object r of shape S satisfying the following constraints:

Let

$$I = (i_0, i_1, \ldots, i_{d-1}, i_d, i_{d+1}, \ldots, i_{N-1})$$

be an element in the *index space* of r.

Consider a collection consisting of the elements of a up to index i_d along d:

$$V = \{a(i_0, i_1, ..., i_{d-1}, j, i_{d+1}, ..., i_{N-1}) \quad | \quad 0 <= j <= i_d\}$$

Let $v = (v_0, v_1, \ldots, v_k)$ be any sequence consisting of the elements of V and 0 or more instances of id.

The value of r at I is any application of op to the sequence v

$$r(I) = v_0 \,\text{op}\, v_1 \,\text{op} \ldots \text{op}\, v_k$$

where the grouping of the application of op and the ordering of the elements of v is unspecified.

If op would incur undefined behavior on any possible grouping or ordering of the elements of v, the computation of the scan as a whole is undefined behavior.

Note: If op is not commutative or not associative or if id is not a true identity element, there may be multiple possible scans for a.

If a particular grouping or ordering of v is observed at index I in the result, this does not imply any constraint on the grouping or ordering of v for any other index of r.

2.6.2. cuda::tiles::reduction_result_t

template<ct::*tile_like* **T**, size_t **D**>
requires (*D* < ct::*tile_rank_v*<*T*>)
using **`reduction_result_t`** = /* see below */;

Yields the result type of applying a *reduction* to an object of type T along dimension D.

2.6.3. cuda::tiles::reduce_max

```
template<
ct::integral auto D,
ct::arithmetic_tile T
>
requires (D >= 0) && (D < tile_rank_v<T>)
__tile__ ct::reduction_result_t<T, D> reduce_max(T a, ct::integral_constant<D> = {}) noexcept;
template<
ct::integral auto D,
ct::nan_propagation_mode Nan = ct::default_nan_propagation_mode(),
ct::subnormals_rounding_mode SubMode = ct::default_subnormals_rounding_mode(),
ct::arithmetic_tile T
>
requires (D >= 0) && (D < tile_rank_v<T>) && /* atomic constraint */
__tile__ ct::reduction_result_t<T, D> reduce_max(T a, ct::integral_constant<D>,
                                                 ct::nan_propagation_mode_constant<Nan>,
                                                 ct::subnormals_rounding_mode_constant<SubMode>
                                                 = {}) noexcept;
```

Yields an unspecified *reduction* of operand `a` along dimension `D` according to binary operator $\mathrm{op}(x, y)$ defined as:

Overload 1: `ct::max(x, y)`

Overload 2: `ct::max<Nan, SubMode>(x, y)`

Let E be the *element type* of T.

If E is an *integral scalar*. the identity element id is the minimal representable value of E.

If E is a *basic floating point scalar*. the identity element id is negative infinity.

The atomic constraint of overload (2) validates that

1. T is a *basic floating point scalar*
2. If `SubMode` is *round subnormals to zero*, then T is `float`.
3. The values `NanMode` and `SubMode` are enumerators of their respective types.

Note: The identity element for floating point arguments id $= -\infty$ may not be a true identity of op depending on the provided numeric flags. For example, when *suppress NaN* (the default) is used:

$$\mathrm{op}(-\infty, \mathrm{NaN}) = -\infty$$

In this scenario, multiple valid reduction results are possible when any element of the tile is a NaN.

Example

The following example performs a max reduction:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
```

(continues on next page)

(continued from previous page)

```
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

float xData[2][4] = {
  {0, 10, 2, 5},
  {-3, 2, 22, 7},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r0 = ct::reduce_max(x, 1_ic); // No flags

auto r1 = ct::reduce_max(x, 1_ic,  // With flags
                         ct::suppress_nan_t{},
                         ct::preserve_subnormals_t{});
```

$$\begin{pmatrix} 0 & 10 & 2 & 5 \\ -3 & 2 & 22 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} 10 \\ 22 \end{pmatrix}$$

2.6.4. cuda::tiles::reduce_min

template<
ct::*integral* auto **D**,
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<*T*, *D*> **reduce_min**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

template<
ct::*integral* auto **D**,
ct::*nan_propagation_mode* **Nan** = ct::*default_nan_propagation_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>) && /* atomic constraint */
__tile__ ct::*reduction_result_t*<*T*, *D*> **reduce_min**(*T* a, ct::*integral_constant*<*D*>,
ct::*nan_propagation_mode_constant*<*Nan*>,
ct::*subnormals_rounding_mode_constant*<*SubMode*>
= {}) noexcept;

Yields an unspecified *reduction* of operand a along dimension D according to binary operator $\mathrm{op}(x, y)$ defined as:

Overload 1: `ct::min(x, y)`

Overload 2: `ct::min<Nan, SubMode>(x, y)`

Let E be the *element type* of T.

If E is an *integral scalar*. the identity element id is the maximal representable value of E.

If E is a *basic floating point scalar*. the identity element id is positive infinity.

The atomic constraint of overload (2) validates that

1. T is a *basic floating point scalar*

2. If SubMode is *round subnormals to zero*, then T is `float`.
3. The values NanMode and SubMode are enumerators of their respective types.

Note: The identity element for floating point arguments id $= \infty$ may not be a true identity of op depending on the provided numeric flags. For example, when *suppress NaN* (the default) is used:

$$\text{op}(\infty, \text{NaN}) = \infty$$

In this scenario, multiple valid reduction results are possible when any element of the tile is a NaN.

Example

The following example performs a min reduction:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

float xData[2][4] = {
  {0, 10, 2, 5},
  {-3, 2, 22, 7},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r0 = ct::reduce_min(x, 0_ic); // No flags

auto r1 = ct::reduce_min(x, 0_ic,  // With flags
                         ct::suppress_nan_t{},
                         ct::preserve_subnormals_t{});
```

$$\begin{pmatrix} 0 & 10 & 2 & 5 \\ -3 & 2 & 22 & 7 \end{pmatrix} \rightarrow \begin{pmatrix} -3 & 2 & 2 & 5 \end{pmatrix}$$

2.6.5. cuda::tiles::all_of

template<ct::*integral* auto **D**, ct::*bool_tile_convertible* **T**>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<ct::*tile_with_element_t*<*T*, bool>, *D*> **all_of**(*T* a,
ct::*integral_constant*<*D*>
= {}) noexcept;

Yields the logical AND *reduction* of the *bool tile converted* operand a along dimension D. The reduction operator $\text{op}(x, y)$ is the expression `x && y` and the identity element is `true`.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

bool xData[2][4] = {
  {true, true, false, false},
  {true, false, true, false},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r = ct::all_of(x, 0_ic);
```

$$\begin{pmatrix} \text{true} & \text{true} & \text{false} & \text{false} \\ \text{true} & \text{false} & \text{true} & \text{false} \end{pmatrix} \to \begin{pmatrix} \text{true} & \text{false} & \text{false} & \text{false} \end{pmatrix}$$

2.6.6. cuda::tiles::any_of

template<ct::*integral* auto **D**, ct::*bool_tile_convertible* **T**>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<ct::*tile_with_element_t*<*T*, bool>, *D*> **any_of**(*T* a,
ct::*integral_constant*<*D*>
= {}) noexcept;

Yields the logical OR *reduction* of the *bool tile converted* operand a along dimension D. The reduction operator $\text{op}(x, y)$ is the expression `x || y` and the identity element is `false`.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

bool xData[2][4] = {
  {true, true, false, false},
  {true, false, true, false},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r = ct::all_of(x, 0_ic);
```

$$\begin{pmatrix} \text{true} & \text{true} & \text{false} & \text{false} \\ \text{true} & \text{false} & \text{true} & \text{false} \end{pmatrix} \to \begin{pmatrix} \text{true} & \text{true} & \text{true} & \text{false} \end{pmatrix}$$

2.6.7. cuda::tiles::sum

```
template<
ct::integral auto D,
ct::arithmetic_tile T
>
requires (D >= 0) && (D < tile_rank_v<T>)
__tile__ ct::reduction_result_t<T, D> sum(T a, ct::integral_constant<D> = {}) noexcept;
template<
ct::integral auto D,
ct::rounding_mode Mode = ct::default_rounding_mode(),
ct::subnormals_rounding_mode SubMode = ct::default_subnormals_rounding_mode(),
ct::arithmetic_tile T
>
requires (D >= 0) && (D < tile_rank_v<T>) && /* atomic constraint */
__tile__ ct::reduction_result_t<T, D> sum(T a, ct::integral_constant<D>,
                                          ct::rounding_mode_constant<Mode>,
                                          ct::subnormals_rounding_mode_constant<SubMode> = {})
                                          noexcept;
```

Yields an unspecified summation *reduction* of operand a along dimension D. The binary operation $\mathrm{op}(x, y)$ is defined as

Overload 1: `ct::add(x, y)`

Overload 2: `ct::add<Mode, SubMode>(x, y)`

The identity element id is 0 for *integral scalar* elements and positive zero for floating point elements of a.

The atomic constraint of overload (2) validates that:

1. T is a *basic floating point scalar*
2. `Mode` is a *precise rounding mode*
3. If `SubMode` is *round subnormals to zero*, then T is `float`.
4. The values `Mode` and `SubMode` are enumerators of their respective types.

Note: The binary operation op is not associative for floating point arguments. There may be multiple valid results of the reduction.

Undefined behavior may occur if any ordering or grouping of elements triggers signed integer overflow. See *undefined behavior annex* for an example.

Example

The following example sums the rows of a tile:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

float xData[2][4] = {
  {3, 2, 1, 4},
```

(continues on next page)

(continued from previous page)

```
  {-3, 2, 1, 5},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r0 = ct::sum(x, 1_ic); // No flags

auto r1 = ct::sum(x, 1_ic,  // With flags
                  ct::round_ties_to_even_t{},
                  ct::preserve_subnormals_t{});
```

$$\begin{pmatrix} 3 & 2 & 1 & 4 \\ -3 & 2 & 1 & 5 \end{pmatrix} \to \begin{pmatrix} 10 \\ 5 \end{pmatrix}$$

2.6.8. cuda::tiles::prod

template<
ct::*integral* auto **D**,
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<*T*, *D*> **prod**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

template<
ct::*integral* auto **D**,
ct::*rounding_mode* **Mode** = ct::*default_rounding_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>) && /* atomic constraint */
__tile__ ct::*reduction_result_t*<*T*, *D*> **prod**(*T* a, ct::*integral_constant*<*D*>,
ct::*rounding_mode_constant*<*Mode*>,
ct::*subnormals_rounding_mode_constant*<*SubMode*> = {})
noexcept;

Yields an unspecified product *reduction* of operand a along dimension D. The binary operation $\mathrm{op}(x, y)$ is defined as

Overload 1: `ct::mul(x, y)`

Overload 2: `ct::mul<Mode, SubMode>(x, y)`

The identity element id is 1.

The atomic constraint of overload (2) validates that:

1. T is a *basic floating point scalar*
2. `Mode` is a *precise rounding mode*
3. If `SubMode` is *round subnormals to zero*, then T is `float`.
4. The values `Mode` and `SubMode` are enumerators of their respective types.

Note: The binary operation op is not associative for floating point arguments. There may be multiple valid results of the reduction.

Undefined behavior may occur if any ordering or grouping of elements triggers signed integer overflow. See the *undefined behavior annex* for an example.

Example

The following example performs a product along the rows of a tile:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

float xData[2][4] = {
  {3, 2, 1, 4},
  {-3, 2, 1, 5},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r0 = ct::prod(x, 1_ic); // No flags

auto r1 = ct::prod(x, 1_ic,  // With flags
                   ct::round_ties_to_even_t{},
                   ct::preserve_subnormals_t{});
```

$$\begin{pmatrix} 3 & 2 & 1 & 4 \\ -3 & 2 & 1 & 5 \end{pmatrix} \to \begin{pmatrix} 24 \\ -30 \end{pmatrix}$$

2.6.9. cuda::tiles::reduce_bitand

template<
ct::*integral* auto **D**,
ct::*integral_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<*T*, *D*> **reduce_bitand**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

Yields the unique bitwise AND *reduction* of the operand a along dimension D.

The binary operator $\mathrm{op}(x, y)$ is the expression `ct::operator&(x, y)`.

Let E be the *element type* of a. If E is a signed *integral scalar*, the identity element id is -1. If E is an unsigned *integral scalar* the identity element is the maximum integer value representable in E.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

unsigned char xData[2][4] = {
  {0b00001111, 0b10101010},
  {0b01010101, 0b11110000},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r = ct::reduce_bitand(x, 0_ic);
```

$$\begin{pmatrix} 0b00001111 & 0b10101010 \\ 0b01010101 & 0b11110000 \end{pmatrix} \rightarrow \begin{pmatrix} 0b00000101 & 0b10100000 \end{pmatrix}$$

2.6.10. cuda::tiles::reduce_bitor

template<ct::*integral* auto **D**, ct::*integral_tile* **T**>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<*T*, *D*> **reduce_bitor**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

Yields the unique bitwise OR *reduction* of the operand a along dimension D.

The binary operator op(x, y) is the expression `ct::operator|(x, y)`.

The identity element id is 0.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

unsigned char xData[2][4] = {
  {0b00001111, 0b10101010},
  {0b01010101, 0b11110000},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r = ct::reduce_bitor(x, 0_ic);
```

$$\begin{pmatrix} 0b00001111 & 0b10101010 \\ 0b01010101 & 0b11110000 \end{pmatrix} \rightarrow \begin{pmatrix} 0b01011111 & 0b11111010 \end{pmatrix}$$

2.6.11. cuda::tiles::reduce_bitxor

template<ct::*integral* auto **D**, ct::*integral_tile* **T**>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ ct::*reduction_result_t*<*T*, *D*> **reduce_bitxor**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

Yields the unique bitwise XOR *reduction* of the operand a along dimension D.

The binary operator $\mathrm{op}(x, y)$ is the expression `ct::operator^(x, y)`.

The identity element id is 0.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

unsigned char xData[2][4] = {
  {0b00001111, 0b10101010},
  {0b01010101, 0b11110000},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r = ct::reduce_bitxor(x, 0_ic);
```

$$\begin{pmatrix} 0b00001111 & 0b10101010 \\ 0b01010101 & 0b11110000 \end{pmatrix} \rightarrow \begin{pmatrix} 0b01011010 & 0b01011010 \end{pmatrix}$$

2.6.12. cuda::tiles::partial_sum

template<
ct::*integral* auto **D**,
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ *T* **partial_sum**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

template<
ct::*integral* auto **D**,
ct::*rounding_mode* **Mode** = ct::*default_rounding_mode*(),
ct::*subnormals_rounding_mode* **SubMode** = ct::*default_subnormals_rounding_mode*(),
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>) && /* atomic constraint */
__tile__ *T* **partial_sum**(*T* a, ct::*integral_constant*<*D*>, ct::*rounding_mode_constant*<*Mode*>,
ct::*subnormals_rounding_mode_constant*<*SubMode*> = {}) noexcept;

Yields an unspecified *scan* summation of operand a along dimension D. The binary operation $\mathrm{op}(x, y)$ is defined as

Overload 1: `ct::add(x, y)`

Overload 2: `ct::add<Mode, SubMode>(x, y)`

The identity element id is 0 for *integral scalar* elements and positive zero for floating point elements of a.

The atomic constraint of overload (2) validates that:

1. T is a *basic floating point scalar*
2. Mode is a *precise rounding mode*
3. If SubMode is *round subnormals to zero*, then T is `float`.
4. The values Mode and SubMode are enumerators of their respective types.

Note: The binary operation op is not associative for floating point arguments. There may be multiple valid results of the scan.

Undefined behavior may occur if any ordering or grouping of elements triggers signed integer overflow. See *undefined behavior annex* for an example.

Example

The following example performs a partial sum of the rows of a tile:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

float xData[2][4] = {
  {3, 2, 1, 4},
  {-3, 2, 1, 5},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r0 = ct::partial_sum(x, 1_ic); // No flags

auto r1 = ct::partial_sum(x, 1_ic,  // With flags
                          ct::round_ties_to_even_t{},
                          ct::preserve_subnormals_t{});
```

$$\begin{pmatrix} 3 & 2 & 1 & 4 \\ -3 & 2 & 1 & 5 \end{pmatrix} \to \begin{pmatrix} 3 & 5 & 6 & 10 \\ -3 & -1 & 0 & 5 \end{pmatrix}$$

2.6.13. cuda::tiles::partial_prod

template<
ct::*integral* auto **D**,
ct::*arithmetic_tile* **T**
>
requires (*D* >= 0) && (*D* < *tile_rank_v*<*T*>)
__tile__ *T* **partial_prod**(*T* a, ct::*integral_constant*<*D*> = {}) noexcept;

```
template<
ct::integral auto D,
ct::rounding_mode Mode = ct::default_rounding_mode(),
ct::subnormals_rounding_mode SubMode = ct::default_subnormals_rounding_mode(),
ct::arithmetic_tile T
>
requires (D >= 0) && (D < tile_rank_v<T>) && /* atomic constraint */
__tile__ T partial_prod(T a, ct::integral_constant<D>, ct::rounding_mode_constant<Mode>,
                        ct::subnormals_rounding_mode_constant<SubMode> = {}) noexcept;
```

Yields an unspecified product *scan* of operand a along dimension D. The binary operation op(x, y) is defined as

Overload 1: `ct::mul(x, y)`

Overload 2: `ct::mul<Mode, SubMode>(x, y)`

The identity element id is 1.

The atomic constraint of overload (2) validates that:

1. T is a *basic floating point scalar*
2. `Mode` is a *precise rounding mode*
3. If `SubMode` is *round subnormals to zero*, then T is `float`.
4. The values `Mode` and `SubMode` are enumerators of their respective types.

Note: The binary operation op is not associative for floating point arguments. There may be multiple valid results of the scan.

Undefined behavior may occur if any ordering or grouping of elements triggers signed integer overflow. See the *undefined behavior annex* for an example.

Example

The following example performs a product scan along the rows of a tile:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
using i32x2x4 = ct::tile<int, ct::shape<2, 4>>;

float xData[2][4] = {
  {3, 2, 1, 4},
  {-3, 2, 1, 5},
};

auto x = ct::load(&xData[0][0] + ct::iota<i32x2x4>());

auto r0 = ct::partial_prod(x, 1_ic); // No flags

auto r1 = ct::partial_prod(x, 1_ic,  // With flags
                           ct::round_ties_to_even_t{},
                           ct::preserve_subnormals_t{});
```

$$\begin{pmatrix} 3 & 2 & 1 & 4 \\ -3 & 2 & 1 & 5 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & 6 & 6 & 24 \\ -3 & -6 & -6 & -30 \end{pmatrix}$$

2.7. Memory Operations

This section describes the operations for performing scatter and gather style memory accesses. A gather load or scatter store operation receives a tile of pointers which may reference discontiguous regions of memory. Each pointer is loaded or stored by the operation simultaneously.

Example

A sparse collection of elements in the 4×4 array is loaded into a dense tile.

```
namespace ct = ::cuda::tiles;

int x[4][4] {
  {0,  1,  2,  3},
  {4,  5,  6,  7},
  {8,  9,  10, 11},
  {12, 13, 14, 15}
};

using i32x2x2 = ct::tile<int, ct::shape<2, 2>>;
using ptr_2x2 = ct::tile<int*, ct::shape<2, 2>>;

// [2, 11, 4, 13]
i32x2x2 idx = (2 + 9 * ct::iota<i32x2x2>()) % 16;

// [p + 2, p + 11, p + 4, p + 13]
ptr_2x2 ptrs = &x[0][0] + idx;

// [*(p + 2), *(p + 11), *(p + 4), *(p + 13)]
i32x2x2 r = ct::load(ptrs);
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & 11 \\ 4 & 13 \end{pmatrix}$$

2.7.1. cuda::tiles::loadable_tile

template<typename **T**>
concept **`loadable_tile`** = ct::*pointer_tile*<*T*> && /* atomic constraint */

Indicates whether a load may occur on a *pointer tile* object of type T.

The atomic constraint validates that:

1. The *element type* of T is not a pointer to (possibly cv-qualified) `void`.
2. The pointee type of the *element type* of T is not volatile qualified.

2.7.2. cuda::tiles::storeable_tile

template<typename **T**>
concept **storeable_tile** = ct::*loadable_tile*<*T*> && /* atomic constraint */

Indicates whether a store may occur on a *pointer tile* object of type T.

The atomic constraint validates that the pointee type of the *element type* of T is not const qualified.

2.7.3. cuda::tiles::tile_load_t

template<ct::*loadable_tile* **T**>
using **tile_load_t** = ct::*tile_with_element_t*<*T*,
remove-cv-t<*remove-pointer-t*<ct::*tile_element_t*<*T*>>>>

Yields the result type when loading from a *pointer tile* object of type T.

2.7.4. Load Operations

template<
ct::*loadable_tile* **Tile**
>
__tile__ ct::*tile_load_t*<*Tile*> **load**(*Tile* ptrs) noexcept;

template<
ct::*loadable_tile* **Tile**,
ct::*bool_tile_convertible* **Mask**
>
requires ct::*broadcastable_to*<*Mask*, ct::*tile_shape_t*<*Tile*>>
__tile__ ct::*tile_load_t*<*Tile*> **load_masked**(*Tile* ptrs, *Mask* mask) noexcept;

template<
ct::*loadable_tile* **Tile**,
ct::*bool_tile_convertible* **Mask**,
ct::*tile_like* **Padding**
>
requires ct::*broadcastable_to*<*Mask*, ct::*tile_shape_t*<*Tile*>> && ct::*broadcastable_to*<*Padding*,
ct::*tile_shape_t*<*Tile*>> && /* atomic constraint*/
__tile__ ct::*tile_load_t*<*Tile*> **load_masked**(*Tile* ptrs, *Mask* mask, *Padding* padding) noexcept;

template<
ct::*memory_order* **Order**,
ct::*thread_scope* **Scope** = ct::*default_thread_scope*(),
ct::*loadable_tile* **Tile**
>
requires ct::*read_memory_order*<*Order*>
__tile__ ct::*tile_load_t*<*Tile*> **atomic_load**(*Tile* ptrs, ct::*memory_order_constant*<*Order*> = {},
ct::*thread_scope_constant*<*Scope*> = {}) noexcept;

template<
ct::*memory_order* **Order**,
ct::*thread_scope* **Scope** = ct::*default_thread_scope*(),
ct::*loadable_tile* **Tile**,

```
ct::bool_tile_convertible Mask
>
requires (ct::broadcastable_to<Mask, ct::tile_shape_t<Tile>> && ct::read_memory_order<Order>)
__tile__ ct::tile_load_t<Tile> atomic_load_masked(Tile ptrs, Mask mask,
                                                  ct::memory_order_constant<Order> = {},
                                                  ct::thread_scope_constant<Scope> = {})
                                                  noexcept;

template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::loadable_tile Tile,
ct::bool_tile_convertible Mask,
ct::tile_like Padding
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Tile>> && ct::broadcastable_to<Padding,
ct::tile_shape_t<Tile>> && ct::read_memory_order<Order> && /* atomic constraint */
__tile__ ct::tile_load_t<Tile> atomic_load_masked(Tile ptrs, Mask mask, Padding padding,
                                                  ct::memory_order_constant<Order> order = {},
                                                  ct::thread_scope_constant<Scope> scope = {})
                                                  noexcept;
```

Loads the values specified by the pointers in `ptrs` into a tile. A load may be masked by `mask` in which case the corresponding result value is either unspecified or supplied by the `padding` argument.

When present, the `mask` argument undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

When present, the `padding` argument is *broadcast converted* to the *shape* of `ptrs` and then *tile converted* to the type `ct::tile_load_t<Tile>`.

Let m and p denote the converted `mask` and `padding`.

The result is a tile a whose value at index J is determined as follows:

1. If the mask $m(J)$ is `true` or `mask` is absent, the value is the result of dereferencing the pointer ptrs(J). If this results in undefined behavior for any index J, the behavior of the operation as a whole is undefined.
2. If $m(J)$ is `false` and `padding` is absent, the value is unspecified.
3. Otherwise, $m(J)$ is `false` and `padding` is specified. The result is the padding value $p(J)$.

An invocation generates a read *memory operation* on each location specified by `ptrs` for which the corresponding value in m is `true` or otherwise not present.

For the `atomic_load` and `atomic_load_masked` overloads, the generated memory operations are strong and have a *memory order* and *thread scope* specified by `Order` and `Scope` respectively.

The *latency* optimization hint may appertain to direct call expressions of the above load APIs.

The atomic constraint validates that the *element type* of `padding` is *scalar convertible* to the *element type* of `ct::tile_load_t<Tile>`.

Example

The following example shows a masked load with both atomic and non-atomic variants. The pointers for which `mask` is `false` are not loaded and their values in the result are taken from the `padding`.

```cpp
namespace ct = ::cuda::tiles;

int data[4] = { 2, 7, 5, 8 };
bool maskData[4] = {true, false, false, true};
int padData[4] = {-7, -3, -22, -100};

using i32x4 = ct::tile<int, ct::shape<4>>;

auto mask = ct::load(&maskData[0] + ct::iota<i32x4>());
auto padding = ct::load(&padData[0] + ct::iota<i32x4>());
auto ptrs = &data[0] + ct::iota<i32x4>();

// Non-atomic
auto r0 = ct::load_masked(ptrs, mask, padding);

// Atomic
auto r1 = ct::atomic_load_masked(ptrs, mask, padding,
                                 ct::memory_order_relaxed_t{},
                                 ct::thread_scope_device_t{});
```

$$\begin{pmatrix} 2 & 7 & 5 & 8 \end{pmatrix} \rightarrow \begin{pmatrix} 2 & -3 & -22 & 8 \end{pmatrix}$$

2.7.5. Store Operations

template<
ct::*storeable_tile* **Ptrs**,
ct::*non_narrowing_tile_convertible_to*<ct::*tile_load_t*<*Ptrs*>> **Value**
>
__tile__ void **store**(*Ptrs* ptrs, *Value* value) noexcept;

template<
ct::*storeable_tile* **Ptrs**,
ct::*non_narrowing_tile_convertible_to*<ct::*tile_load_t*<*Ptrs*>> **Value**,
ct::*bool_tile_convertible* **Mask**
>
requires ct::*broadcastable_to*<*Mask*, ct::*tile_shape_t*<*Ptrs*>>
__tile__ void **store_masked**(*Ptrs* ptrs, *Value* value, *Mask* mask) noexcept;

template<
ct::*memory_order* **Order**,
ct::*thread_scope* **Scope** = ct::*default_thread_scope*(),
ct::*storeable_tile* **Ptrs**,
ct::*non_narrowing_tile_convertible_to*<ct::*tile_load_t*<*Ptrs*>> **Value**
>
requires ct::*write_memory_order*<*Order*>
__tile__ void **atomic_store**(*Ptrs* ptrs, *Value* value, ct::*memory_order_constant*<*Order*> = {},
ct::*thread_scope_constant*<*Scope*> = {}) noexcept;

template<
ct::*memory_order* **Order**,
ct::*thread_scope* **Scope** = ct::*default_thread_scope*(),
ct::*storeable_tile* **Ptrs**,
ct::*non_narrowing_tile_convertible_to*<ct::*tile_load_t*<*Ptrs*>> **Value**,

ct::*bool_tile_convertible* **Mask**
>
requires ct::*broadcastable_to*<*Mask*, ct::*tile_shape_t*<*Ptrs*>> && ct::*write_memory_order*<*Order*>
__tile__ void **atomic_store_masked**(*Ptrs* ptrs, *Value* value, *Mask* mask,
ct::*memory_order_constant*<*Order*> = {},
ct::*thread_scope_constant*<*Scope*> = {}) noexcept;

Stores the elements specified by `value` to the memory locations of the corresponding elements of `ptrs`. A store may be inhibited by specifying `mask` values for the corresponding elements in `ptrs`.

The `value` argument undergoes *tile conversion* to the type *`tile_load_t`*<*`Ptrs`*>.

When present, `mask` undergoes *bool tile conversion* followed by *broadcast conversion* to the shape of *`ptrs`*.

Let v and m denote the converted `value` and `mask` arguments respectively and let J be an index into `ptrs`.

If `mask` is absent or $m(J)$ is `true`, the value $v(J)$ is stored at the memory location specified by ptrs(J). Otherwise, no store occurs for that index.

An invocation generates a write *memory operation* for each location specified by `ptrs` whose corresponding element in m is not `false`. For the `atomic_store` and `atomic_store_masked` variants, the generated memory operations are strong and have *memory order* and *thread scope* specified by `Order` and Scope respectively.

The *latency* optimization hint may appertain to direct call expressions of the above store APIs.

Note: Multiple *memory operations* may be generated on a single memory location if the same pointer value is present more than once in `ptrs`. This results in undefined behavior for the non-atomic overloads.

Example

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;

int data[4] = { 0, 1, 2, 3 };
bool maskData[4] = {true, false, false, true};

using i32x4 = ct::tile<int, ct::shape<4>>;

auto mask = ct::load(&maskData[0] + ct::iota<i32x4>());
auto value = ct::full<i32x4>(-1);
auto ptrs = &data[0] + ct::iota<i32x4>();

// Non-atomic
ct::store_masked(ptrs, value, mask);

// Atomic
ct::atomic_store_masked(ptrs, value, mask,
                        ct::memory_order_relaxed_t{},
                        ct::thread_scope_device_t{});
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} -1 & 1 & 2 & -1 \end{pmatrix}$$

2.7.6. cuda::tiles::atomic_compare_exchange

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Cmp,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Val
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_compare_exchange(Ptrs ptrs, Cmp cmp, Val val,
                                                       ct::memory_order_constant<Order> = {},
                                                       ct::thread_scope_constant<Scope> = {})
                                                       noexcept;

template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Cmp,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Val,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_compare_exchange_masked(Ptrs ptrs, Cmp cmps, Val vals,
                                                              Mask mask,
                                                              ct::memory_order_constant<Order>
                                                              = {},
                                                              ct::thread_scope_constant<Scope>
                                                              = {}) noexcept;
```

Performs an *elementwise* atomic compare exchange of the memory locations of `ptrs`.

The values `cmps` and `vals` undergo *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is specified, it undergoes *bool tile conversion* followed by *broadcast conversion* to the shape of `ptrs`.

Let p, c, v, and m denote corresponding elements of the converted operands `ptrs`, `cmp`, `val`, and `mask` respectively.

When m is `true` or `mask` is not specified, the value representations of c and the object at location p are compared. If they are bitwise identical, v is stored to the address p. The corresponding element of the result is the value under p that was used for the comparison check.

Otherwise, m is `false` and no read, comparison, or store occurs. The corresponding element of the result is the comparison value c.

A call to this API generates a strong read-write *memory operation* for each memory location p for which the mask m is `true` or not specified. The value read by the memory operation is the value under p that was used in the comparison check. The value written by the memory operation is v if the comparison succeeds or the value under p that was used for the comparison check otherwise.

The memory order and scope of these memory operations are specified by `Order` and `Scope` respectively.

The atomic constraint validates that the *element type* of `ct::tile_load_t<Ptrs>` is one of

1. A 32 bit or 64 bit *integral scalar*

2. A `float` or `double`

Note: A read-write memory operation is generated for a given element even if the comparison failed. In this scenario, the value written is the value that was read for the comparison check.

A caller may test if the compare and exchange succeeded by performing a bitwise comparison of the return value with the converted `cmp`. Equality comparison may behave differently than bitwise comparison for floating point values.

2.7.7. cuda::tiles::atomic_and

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_and(Ptrs ptrs, Values values,
                                          ct::memory_order_constant<Order> = {},
                                          ct::thread_scope_constant<Scope> = {}) noexcept;

template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_and_masked(Ptrs ptrs, Values values, Mask mask,
                                                 ct::memory_order_constant<Order> = {},
                                                 ct::thread_scope_constant<Scope> = {})
                                                 noexcept;
```

Performs *elementwise* atomic bitwise AND on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The bitwise AND between v and k is computed as if by invoking `ct::operator&(k, v)` and the result is stored to p. The corresponding element in the return value is k

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and Scope respectively.

The atomic constraint validates that the *element type* of `ct::tile_load_t<Ptrs>` is a 32 or 64 bit *integral scalar* type.

2.7.8. cuda::tiles::atomic_or

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_or(Ptrs ptrs, Values values, ct::memory_order_constant<Order>
                                         = {}, ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_or_masked(Ptrs ptrs, Values values, Mask mask,
                                                ct::memory_order_constant<Order> = {},
                                                ct::thread_scope_constant<Scope> = {}) noexcept;
```

Performs *elementwise* atomic bitwise OR on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The bitwise OR between v and k is computed as if by invoking `ct::operator|(k, v)` and the result is stored to p. The corresponding element in the return value is k

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and `Scope` respectively.

The atomic constraint validates that the *element type* of `ct::tile_load_t<Ptrs>` is a 32 or 64 bit *integral scalar* type.

2.7.9. cuda::tiles::atomic_xor

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_xor(Ptrs ptrs, Values values,
                                          ct::memory_order_constant<Order> = {},
                                          ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_xor_masked(Ptrs ptrs, Values values, Mask mask,
                                                 ct::memory_order_constant<Order> = {},
                                                 ct::thread_scope_constant<Scope> = {})
                                                 noexcept;
```

Performs *elementwise* atomic bitwise XOR on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The bitwise XOR between v and k is computed as if by invoking `ct::operator^(k, v)` and the result is stored to p. The corresponding element in the return value is k

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and `Scope` respectively.

The atomic constraint validates that the *element type* of `ct::tile_load_t<Ptrs>` is a 32 or 64 bit *integral scalar* type.

2.7.10. cuda::tiles::atomic_max

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_max(Ptrs ptrs, Values values,
                                          ct::memory_order_constant<Order> = {},
                                          ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_max_masked(Ptrs ptrs, Values values, Mask mask,
                                                 ct::memory_order_constant<Order> = {},
                                                 ct::thread_scope_constant<Scope> = {})
                                                 noexcept;
```

Performs *elementwise* atomic maximum on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The maximum between v and k is computed as if by invoking `ct::max(k, v)` and the result is stored to p. The corresponding element in the return value is k.

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and `Scope` respectively.

The atomic constraint validates that the *element type* of `ct::tile_load_t<Ptrs>` is a 32 or 64 bit *integral scalar* type.

2.7.11. cuda::tiles::atomic_min

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_min(Ptrs ptrs, Values values,
                                          ct::memory_order_constant<Order> = {},
                                          ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_min_masked(Ptrs ptrs, Values values, Mask mask,
                                                 ct::memory_order_constant<Order> = {},
                                                 ct::thread_scope_constant<Scope> = {})
                                                 noexcept;
```

Performs *elementwise* atomic minimum on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The minimum between v and k is computed as if by invoking `ct::min(k, v)` and the result is stored to p. The corresponding element in the return value is k.

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and `Scope` respectively.

The atomic constraint validates that the *element type* of `ct::tile_load_t<Ptrs>` is a 32 or 64 bit *integral scalar* type.

2.7.12. cuda::tiles::atomic_add

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_add(Ptrs ptrs, Values values,
                                          ct::memory_order_constant<Order> = {},
                                          ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_add_masked(Ptrs ptrs, Values values, Mask mask,
                                                 ct::memory_order_constant<Order> = {},
                                                 ct::thread_scope_constant<Scope> = {})
                                                 noexcept;
```

Performs *elementwise* atomic addition on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The sum between v and k is computed as if by invoking

1. If v is a *integral scalar*: `ct::add(k, v)`. If this produces undefined behavior, the behavior of the operation as a whole is undefined.
2. If v is a *basic floating point scalar*:

   `ct::add<ct::rounding_mode::round_ties_to_even, SubMode>(k, v)`

   where the value of SubMode is not specified.

The result of the computation is stored to p. The corresponding element in the return value is k.

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and Scope respectively.

The atomic constraint validates that *element type* of `ct::tile_load_t<Ptrs>` is one of

1. A 32 or 64 bit *integral scalar* type
2. `double`, `float`, or `__half`

2.7.13. cuda::tiles::atomic_sub

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_sub(Ptrs ptrs, Values values,
                                          ct::memory_order_constant<Order> = {},
                                          ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_sub_masked(Ptrs ptrs, Values values, Mask mask,
                                                 ct::memory_order_constant<Order> = {},
                                                 ct::thread_scope_constant<Scope> = {})
                                                 noexcept;
```

Performs *elementwise* atomic subtraction on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k. The difference between v and k is computed as if by invoking

1. If v is a *integral scalar*: `ct::add(k, -v)`. If this produces undefined behavior, the behavior of the operation as a whole is undefined.

   Note: The behavior is undefined if v is a signed integral type and overflow occurs in the unary negation even if the difference between k and v is representable in the target integer type.

2. If v is a *basic floating point scalar*:

   `ct::sub<ct::rounding_mode::round_ties_to_even, SubMode>(k, v)`

   where the value of SubMode is not specified.

The result of the computation is stored to p. The corresponding element in the return value is k.

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the

result of the computation. The memory order and thread scope of the operation is determined by `Order` and Scope respectively.

The atomic constraint validates that *element type* of `ct::tile_load_t<Ptrs>` is one of

1. A 32 or 64 bit *integral scalar* type
2. `double`, `float`, or `__half`

2.7.14. cuda::tiles::atomic_xchg

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values
>
requires /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_xchg(Ptrs ptrs, Values values,
                                           ct::memory_order_constant<Order> = {},
                                           ct::thread_scope_constant<Scope> = {}) noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::storeable_tile Ptrs,
ct::non_narrowing_tile_convertible_to<ct::tile_load_t<Ptrs>> Values,
ct::bool_tile_convertible Mask
>
requires ct::broadcastable_to<Mask, ct::tile_shape_t<Ptrs>> && /* atomic constraint */
__tile__ ct::tile_load_t<Ptrs> atomic_xchg_masked(Ptrs ptrs, Values values, Mask mask,
                                                  ct::memory_order_constant<Order> = {},
                                                  ct::thread_scope_constant<Scope> = {})
                                                  noexcept;
```

Performs *elementwise* atomic exchange on the memory locations of `ptrs`.

The `values` operand undergoes *tile conversion* to the type `ct::tile_load_t<Ptrs>`. If `mask` is present, it undergoes *bool tile conversion* followed by *broadcast conversion* to the *shape* of `ptrs`.

Let p, v, and m be corresponding elements of the converted operands `ptrs`, `values`, and `mask` when present.

If m is `true` or not present, p is loaded to produce a value k and v is subsequently stored to p. The corresponding element in the return value is k.

Otherwise, m is `false` and no read, computation, or write is performed. The corresponding element in the return value is unspecified.

A call to this API generates a strong *read-write memory operation* for each address of `ptrs` which is not masked. The value read is the value used for the computation and the value written is the result of the computation. The memory order and thread scope of the operation is determined by `Order` and Scope respectively.

The atomic constraint validates that *element type* of `ct::tile_load_t<Ptrs>` is one of

1. A 32 or 64 bit *integral scalar* type
2. `double`, or `float`

2.8. Extents

A *ct::extents* is an *extents like* type suitable for describing the dimensions of a *ct::tile* or *ct::tensor_span*.

Example

In the following code, the extents object x and y represents the shape 4×7. where the first dimension is known at compile time and the second is determined at runtime.

The extents object z represents the shape 33×0 where both dimensions are known at compile time.

In all cases, the index type used for index computations is `uint32_t`.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

ct::extents                                    x{4_ic, 7};
ct::extents<uint32_t, 4, ct::dynamic_extent>   y{7};
ct::shape<33, 0>                               z;
```

2.8.1. cuda::tiles::extents

template<ct::*integral* **I**, size_t... **Exts**>
struct **extents**;

An *extents like* type with static or dynamic extents specified by `Exts` and index type specified by `I`.

A specialization T of *extents* models *extents like* and satisfies the following requirements:

- `std::regular`[1]
- `std::is_nothrow_move_constructible_v`[2]
- `std::is_nothrow_move_assignable_v`[2]
- `std::is_nothrow_swappable_v`[2]

T is trivially copyable[3].

The program is ill-formed if I is cv-qualified.

The *shape* described by an *extents* object is determined by the *static_extent*() and *extent*() member functions.

[1] See § 18.6 [concepts.object] of ISO/IEC 14882:2024
[2] See § 21.3.3 [meta.type.synop] of ISO/IEC 14882:2024
[3] See § 6.8.1 [basic.types.general] of ISO/IEC 14882:2024

Exposition Only Member

index_type **__dynamic_extents**[*rank_dynamic*()];

For the purposes of defining the object size, alignment, and behavior of implicitly generated member functions, T has an exposition only member variable *__dynamic_extents*.

The member variable determines the value of the dynamic extents of an object of type T.

Member Aliases

using **index_type** = *I*;

The index type used for storing the dynamic extent values.

using **rank_type** = size_t;

The rank type which is suitable for selecting a particular extent in the *static_extent()* and *extent()* APIs.

rank

__tile__ __host__ __device__ static constexpr *rank_type* **rank**() noexcept;

Yields the size of the *Exts* parameter pack.

rank_dynamic

__tile__ __host__ __device__ static constexpr *rank_type* **rank_dynamic**() noexcept;

Yields the number of *ct::dynamic_extent* values in the *Exts* parameter pack.

static_extent

__tile__ __host__ __device__ static constexpr size_t **static_extent**(*rank_type* i) noexcept;

Yields the value of the i^{th} element of the *Exts* parameter pack. The behavior is undefined if i is not less than the length of the *Exts* pack.

extent

__tile__ __host__ __device__ constexpr *index_type* **extent**(*rank_type* i) const noexcept;

Yields the value of the i^{th} element of the *Exts* pack or the runtime value for the i^{th} dimension if i corresponds to a *ct::dynamic_extent* value in the *Exts* pack.

When i^{th} references a dynamic extent, the result is the N^{th} element of *__dynamic_extents* where N is the number of dynamic extents whose pack index in *Exts* is less than i.

The behavior is undefined if i is not less than *rank*() or if i is a static extent whose value is not representable in *index_type*.

Default Construction

constexpr **extents**( ) noexcept = default;

Constructs a *ct::extents* object where each element of *__dynamic_extents* is initialized to 0.

Construction From Dynamic Extents

template<typename ...**OtherExtents**>
requires /* atomic constraint */
__tile__ __host__ __device__ explicit constexpr **extents**(*OtherExtents*... other) noexcept;

Constructs a *ct::extents* from a pack specifying the dynamic extent values. The *__dynamic_extents* member is initialized as if by *__dynamic_extents*{*index_type*(*other*)...}.

The atomic constraint validates that:

1. Each of `OtherExtents` is convertible to *index_type*.
2. The size of the `OtherExtents` pack equals *rank_dynamic*( ) and is not 0.

Example

The following code constructs an extents object modeling the shape $8 \times 42 \times 3$.

```
namespace ct = ::cuda::tiles;
ct::extents<int32_t, 8, ct::dynamic_extent, 3> e{42};
```

Construction From Extents

template<typename ...**OtherExtents**>
requires /* atomic constraint */
__tile__ __host__ __device__ explicit constexpr **extents**(*OtherExtents*... other) noexcept;

Constructs a *ct::extents* from a pack specifying all the extent values. The *__dynamic_extents* member is initialized as by considering each element of the *other* pack whose index N corresponds to a dynamic extent value in *Exts*.

Each such element, after conversion to *index_type*, is placed at index M of *__dynamic_extents* where M is the number of dynamic extent values whose pack index in *Exts* is less than N.

The behavior is undefined if any element of *other* corresponding to a static extent does not equal the static extent value after conversion to *index_type*.

The atomic constraint validates that:

1. Each of `OtherExtents` is convertible to *index_type*.
2. The size of the `OtherExtents` pack equals *rank*( ), does not equal *rank_dynamic*( ) and is not 0.

Example

The following code constructs an extents object modeling the shape $8 \times 42 \times 3$.

```
namespace ct = ::cuda::tiles;
ct::extents<int32_t, 8, ct::dynamic_extent, 3> e{8, 42, 3};
```

Deduction Guide

template<typename ...**Ts**>
extents(*Ts*...) -> *extents*<uint32_t, extent-constant-or-dynamic<*Ts*>...>;

Deduction guide enabling CTAD from a collection of *`ct::integral_constant`* or integral arguments.

Example

The following code produces an extents object of type `ct::extents<uint32_t, 4, ct::dynamic_extent>`.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
ct::extents x{4_ic, 7};
```

2.8.2. cuda::tiles::dynamic_extent

enum **`/* unspecified identifier */`** : size_t

enumerator **`dynamic_extent`** = /* see below */

Yields the sentinel value for encoding a dynamic extent in an *extents like* type. The value is the maximum value which can be represented by `size_t`.

Note: The program is ill-formed when attempting to form the address of `ct::dynamic_extent`.

2.8.3. cuda::tiles::shape

template<ct::size_t... **Ts**>
requires ct::*shape_like*<ct::*extents*<uint32_t, *Ts*...>>
using **shape**;

Convenience alias for producing a *shape like* extents whose index type is `uint32_t`.

2.8.4. Equality Comparison

template<ct::*extents_like* **Lhs**, ct::*extents_like* **Rhs**>
__tile__ __host__ __device__ constexpr bool **operator==**(*Lhs* const &lhs, *Rhs* const &rhs) noexcept;

Indicates whether `lhs` is *extent equivalent* to `rhs`.

Note: A corresponding overload for `operator!=` is available via rewritten operator candidates[4].

2.8.5. extent-constant-or-dynamic

template<typename **T**>
inline constexpr size_t **extent-constant-or-dynamic**<*T*>

Exposition only variable template defined as follows:

- If T is a specialization of *ct::integral_constant*, yields the result of converting `T::value` to `size_t`.
- Otherwise, yields *ct::dynamic_extent*.

2.9. Tile

The *ct::tile* type represents an immutable multi-dimensional array of *scalars* with a statically known *shape*. Tiles have value semantics.

Example

The following code builds a tile containing the following elements:

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$$

The tile can be copied and the two copies remain independent. There is typically no performance overhead for copying tiles even if the tile contains a large number of elements.

```
namespace ct = ::cuda::tiles;
using i32x2x2 = ct::tile<int, ct::shape<2, 2>>;

auto x = ct::iota<i32x2x2>();

// Performs a deep copy of 'x' to 'y'.
auto y = x;
```

---

[4] See § 12.2.2.3 [over.match.oper] of ISO/IEC 14882:2024

2.9.1. cuda::tiles::tile

template<ct::*scalar* **E**, ct::*tile_shape* **S**>
struct **`tile`** final;

A specializations T of *`ct::tile`* satisfies the following requirements:

- `std::copyable<T>`[1]
- `std::is_nothrow_move_constructible_v<T>`[2]
- `std::is_nothrow_move_assignable_v<T>`[2]
- `std::is_nothrow_swappable_v<T>`[2]

All specializations of *`ct::tile`* are trivially copyable.[3]

The object size of a specialization `ct::tile<E, S>` is the product of the object size of E and the *shape size* of S. The alignment of `ct::tile<E, S>` is the alignment of E.

The program is ill-formed if E or S are cv-qualified.

The values of a tile are the possible combinations of the values of E that may be arranged into a shape specified by S. The value representation of the tile (for example, the in memory arrangement of elements) is unspecified.

Exposition Only Member

unsigned char **`__data`**[/* see below */];

For the purposes of defining the implicitly generated special member functions, the tile type contains an exposition only member *`__data`*. The number of elements of *`__data`* and its alignment are equal to the object size and alignment of the tile as specified in *`ct::tile`*.

Unless constructed through the default constructor, objects[4] of type E exist in the storage of *`__data`* and such objects are appropriately copied, moved or destroyed when the relevant special member function is invoked.

Member Aliases

using **`shape_type`** = *S*;
: Indicates the *tile shape* of T.

using **`element_type`** = *E*;
: Indicates the *element type* of T.

using **`rank_type`** = typename *S*::rank_type;
: Indicates type used for indexing the dimensions of T.

[1] See § 18.6 [concepts.object] of ISO/IEC 14882:2024
[2] See § 21.3.3 [meta.type.synop] of ISO/IEC 14882:2024
[3] See § 6.8.1 [basic.types.general] of ISO/IEC 14882:2024
[4] See § 6.7.2 [intro.object] of ISO/IEC 14882:2024

Default Constructor

tile( ) = default;

Constructs an instance of T whose elements are uninitialized. The only valid operation on the constructed object is destruction or overwrite by copy or move assignment. All other operations generate undefined behavior.

Conversions from Tile Like

template<ct::*tile_like* **0**>
requires ct::*tile_convertible_to*<*O*, *tile*>
__tile__ explicit(!ct::*non_narrowing_tile_convertible_to*<*O*, *tile*>) **tile**(*O* other) noexcept;

Constructs an instance of T from a *tile like* object `other` which is *tile convertible* to T. The constructed tile's value is the value produced by the (possibly narrowing) *tile conversion* of `other` to T.

Example

In the following example, the first tile is constructed from a *scalar* and the second tile is constructed from the conversion of another tile.

```
namespace ct = ::cuda::tiles;
ct::tile<int, ct::shape<>> x{2};

auto y = ct::full<ct::tile<int, ct::shape<4, 4>>>(2);

// Tile conversion from integers to floats
ct::tile<float, ct::shape<4, 4>> z{y};
```

Conversions to Scalars

template<ct::*scalar* **0**>
requires ct::*scalar_convertible_to*<*E*, *O*> && /* atomic constraint */
__tile__ explicit(!ct::*non_narrowing_scalar_convertible_to*<*E*, *O*>) **operator** ***0***( ) noexcept;

Converts the *singleton tiles* T to the *scalar* type O. The value of the resulting object is the *scalar conversion* of the unique element of the source tile. The atomic constraint validates that the *tile size* of T is 1.

Example

The following example converts a two dimensional singleton tile to a scalar of differing *element type*.

```
namespace ct = ::cuda::tiles;
using f32x1x1 = ct::tile<float, ct::shape<1, 1>>;
f32x1x1 x = ct::full<f32x1x1>(2.0f);

double y{x}; // Performs scalar conversion
```

Deduction Guide

template<ct::*scalar* **0**>
tile(*O*) -> *tile*<*remove-cv-t*<*O*>, ct::*shape*<>>;

Deduction guide which enables CTAD from *scalar* arguments.

Example

The following example invokes the *tile conversion* constructor to produce a tile of type `ct::tile<int, ct::shape<>>`.

```
namespace ct = ::cuda::tiles;
ct::tile x{2};
```

2.10. Layout Mappings

A *layout mapping* describes how the elements of a *ct::tensor_span* will be arranged in memory.

2.10.1. cuda::tiles::layout_right

struct **layout_right**

A *layout policy* representing the row-major element order.

The last multi-dimensional index varies the fastest with the increasing elements in memory.

Example

The following code creates 2×3 row-major *ct::tensor_span* over the elements $(0, 1, \ldots, 5)$:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

int x[] = {0, 1, 2, 3, 4, 5};

// Syntax 1
ct::tensor_span t1{&x[0], ct::extents{2_ic, 3_ic}, ct::layout_right{}};

// Syntax 2
ct::layout_right_mapping m{ct::extents{2_ic, 3_ic}};
ct::tensor_span t2{&x[0], m};
```

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

template<ct::*extents_like* **E**>

using **mapping** = ct::*layout_right_mapping*<*E*>

Alias template yielding the associated *layout mapping* type for representing the row-major arrangement of an array whose shape is an instance of E.

2.10.2. cuda::tiles::layout_right_mapping

template<ct::*extents_like* **E**>
struct **layout_right_mapping**

Layout mapping representing a row-major arrangement of an array whose shape is an instance of E. See `ct::layout_right` for an example.

A specialization T of `ct::layout_right_mapping` models the requirements of *layout mapping*. Additionally, T satisfies the following constraints if the corresponding constraint is satisfied by the underlying *extents like* type E:

- `std::is_nothrow_move_constructible_v<T>`[1]
- `std::is_nothrow_move_assignable_v<T>`[1]
- `std::is_nothrow_swappable_v<T>`[1]

The program is ill-formed if E is cv-qualified.

Exposition Only Members

extents_type **__extents**

T contains an exposition only member variable `__extents` denoting this object's *layout mapping shape*. This member exists for the purposes of defining the behavior of the implicitly generated special member functions.

Member Aliases

using **extents_type** = *E*

using **rank_type** = typename *extents_type*::rank_type

using **index_type** = typename *extents_type*::index_type

using **layout_type** = ct::*layout_right*

Constructor

__tile__ __host__ __device__ explicit **layout_right_mapping**(*extents_type* const &other) noexcept

Constructs an instance of this layout mapping by direct-list-initializing `__extents` with `other`.

[1] See § 21.3.3 [meta.type.synop] of ISO/IEC 14882:2024

is_always_strided

__tile__ __host__ __device__ static constexpr bool **is_always_strided**( ) noexcept

Yields `true`.

static_stride

__tile__ __host__ __device__ static constexpr size_t **static_stride**(*rank_type* i) noexcept

Returns the static stride at dimension i for a row-major layout of shape *__extents*.

Let E denote the *extents_type* and let N be its rank. The behavior is undefined if i is not in the range $[0, N)$.

If the value *extents_type*`::static_extent(k)` is *ct::dynamic_extent* for some value $i < k < N$, the result is *ct::dynamic_extent*. Otherwise, the result is

$$\prod_{k=i+1}^{N-1} E_k$$

and the behavior is undefined if this value is not representable in `size_t` or if the value equals *ct::dynamic_extent*.

stride

__tile__ __host__ __device__ *index_type* **stride**(*rank_type* i) const noexcept

Returns the stride at dimension i for a row major layout of shape *__extents*.

Let e denote *__extents* and let N be its rank. The behavior is undefined if i is not in the range $[0, N)$.

The result is

$$\prod_{k=i+1}^{N-1} e_k$$

The behavior is undefined if this value is not representable in *index_type*.

extents

__tile__ __host__ __device__ *extents_type* const &**extents**( ) const noexcept

Yields the glvalue *__extents*.

2.10.3. cuda::tiles::layout_left

struct **layout_left**

Layout policy representing a column-major arrangement of elements. The first multi-dimensional index varies the fastest with the increasing elements in memory.

Example

The following code creates 2×3 column-major *ct::tensor_span* over the elements $(0, 1, \ldots, 5)$:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

int x[] = {0, 1, 2, 3, 4, 5};

// Syntax 1
ct::tensor_span p1{&x[0], ct::extents{2_ic, 3_ic}, ct::layout_left{}};

// Syntax 2
ct::layout_left_mapping m{ct::extents{2_ic, 3_ic}};
ct::tensor_span p2{&x[0], m};
```

$$\begin{pmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix}$$

---

template<ct::*extents_like* **E**>
using **mapping** = ct::*layout_left_mapping*<*E*>

Alias template yielding the associated *layout mapping* type for representing the column-major arrangement of an array whose shape is an instance of E.

2.10.4. cuda::tiles::layout_left_mapping

template<ct::*extents_like* **E**>
struct **layout_left_mapping**

Layout mapping representing a column-major arrangement of an array whose shape is an instance of E. For an example, see *ct::layout_left*.

A specialization T of *ct::layout_left_mapping* models the requirements of *layout mapping*. Additionally, T satisfies the following constraints if the corresponding constraint is satisfied by the underlying *extents like* type E:

- `std::is_nothrow_move_constructible_v<T>`[Page 125, 1]
- `std::is_nothrow_move_assignable_v<T>`[Page 125, 1]
- `std::is_nothrow_swappable_v<T>`[Page 125, 1]

The program is ill-formed if E is cv-qualified.

Exposition Only Members

extents_type **__extents**

T contains an exposition only member variable *__extents* denoting this object's *layout mapping shape*. This member exists for the purposes of defining the behavior of the implicitly generated special member functions.

Types

using **extents_type** = *E*
using **rank_type** = typename *extents_type*::rank_type
using **index_type** = typename *extents_type*::index_type
using **layout_type** = ct::*layout_left*

Constructor

__tile__ __host__ __device__ explicit **layout_left_mapping**(*extents_type* const &other) noexcept

Constructs an instance of this layout mapping by direct-list-initializing `__extents` with `other`.

is_always_strided

__tile__ __host__ __device__ static constexpr bool **is_always_strided**() noexcept

Yields `true`.

static_stride

__tile__ __host__ __device__ static constexpr size_t **static_stride**(*rank_type* i) noexcept

Returns the static stride at dimension i for a column major layout of shape `__extents`.

Let E denote `extents_type` and let N be its rank. The behavior is undefined if i is not in the range $[0, N)$.

If `extents_type::static_extent(i)` is `ct::dynamic_extent` for some value $0 \leq i < dim$, the result is `ct::dynamic_extent`. Otherwise, the result is

$$\prod_{k=0}^{i-1} E_k$$

and the behavior is undefined if this value is not representable in `size_t` or equals `ct::dynamic_extent`.

stride

__tile__ __host__ __device__ *index_type* **stride**(*rank_type* i) const noexcept

Returns the stride at dimension i for a column major layout of shape `__extents`.

Let e denote `__extents` and let N be its rank. The behavior is undefined if i is not in the range $[0, N)$.

The returned value is

$$\prod_{k=0}^{i-1} e_k$$

The behavior is undefined if this value is not representable in `index_type`.

extents

__tile__ __host__ __device__ *extents_type* const &**extents**( ) const noexcept

Yields the glvalue *__extents*.

2.10.5. cuda::tiles::layout_right_padded

template<size_t **A**>
struct **layout_right_padded**

Layout policy representing a row-major element order whose last dimension is padded to ensure alignment of consecutive rows. The template parameter A specifies the desired alignment of each row. If A is *ct::dynamic_extent*, the alignment is instead supplied as a runtime argument when constructing an instance of *layout_right_padded_mapping*.

In a padded row-major layout, the last multi-dimensional index varies the fastest with the increasing elements in memory. Additionally, the position of each element anchoring a row is divisible by the desired alignment.

Example

Creates a 2×3 *ct::tensor_span* object over the elements $0, 1, \ldots, 7$ where each row is aligned to 4.

In the second case, the specified alignment of 2 is less then the row length of 3. The alignment is rounded up to the smallest multiple that is greater than the row length.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
int x[] = {0, 1, 2, 3, 4, 5, 6, 7};

// Syntax 1
ct::layout_right_padded_mapping m1{ct::extents{2_ic, 3_ic}, 4_ic};
ct::tensor_span p1{&x[0], m1};

// Syntax 2
ct::layout_right_padded_mapping m2{ct::extents{2_ic, 3_ic}, 2_ic};
ct::tensor_span p2{&x[0], m2};
```

$$\begin{pmatrix} 0 & 1 & 2 \\ 4 & 5 & 6 \end{pmatrix}$$

---

template<ct::*extents_like* **E**>
using **mapping** = ct::*layout_right_padded_mapping*<*E*, Alignment>

Alias template yielding the associated *layout mapping* type for representing a padded row-major arrangement of an array whose shape is an instance of E.

2.10.6. cuda::tiles::layout_right_padded_mapping

template<ct::*extents_like* **E**, size_t **A**>
struct **layout_right_padded_mapping**

Layout mapping representing a row-major element order whose shape is an instance of E. The last dimension is padded to ensure alignment of consecutive rows. For an example, see `ct::layout_right_padded`.

The template parameter A specifies the desired alignment of each row. If A is `ct::dynamic_extent`, the alignment is instead supplied as a runtime argument in the constructor `ct::layout_right_padded_mapping::layout_right_padded_mapping()`.

A specialization T of `ct::layout_right_padded_mapping` models the requirements of *layout mapping*. Additionally, T satisfies the following constraints if the corresponding constraint is satisfied by the underlying *extents like* type E:

- `std::is_nothrow_move_constructible_v<T>`[Page 125, 1]
- `std::is_nothrow_move_assignable_v<T>`[Page 125, 1]
- `std::is_nothrow_swappable_v<T>`[Page 125, 1]

The program is ill-formed if E is cv-qualified.

Exposition Only Members

extents_type **__extents**

Exposition-only member denoting this object's *layout mapping shape*.

index_type **__alignment**;

Exposition-only member denoting the desired alignment of each row specified at runtime. This member is present only when *A* is `ct::dynamic_extent`.

using **__padded_extents_t** = /* see below */

Exposition-only member denoting the type of a *padded extents* object. Let E denote `extents_type` and let N be its rank.

The *padded extents* type is the unique `ct::extents` specialization S satisfying:

1. The index type and *rank* of S match those of E.
2. The static extents of S match those of E with the possible exception of the dimension $N-1$ when $N \neq 0$.
3. If $N \neq 0$, the value S_{N-1} is
   1. `ct::dynamic_extent` if either A or E_{N-1} is `ct::dynamic_extent`.
   2. The smallest multiple of A that is not less than E_{N-1} otherwise. If this value is not representable in the type `size_t`, the behavior is undefined when constructing an instance of this object.

__tile__ __host__ __device__ constexpr *__padded_extents_t* **__padded_extents**( ) const noexcept

Exposition only function that yields a *padded extents* object based on the values of `__extents` and `__alignment` when present. Let e denote `__extents` and let N be its rank. Let a denote `__alignment` when present and *A* otherwise. The returned value is the unique instance s of `__padded_extents_t` satisfying:

1. The extent values of s match those of e with the possible exception of of dimension $N-1$ when $N \neq 0$.
2. When $N \neq 0$, the value s_{N-1} is the smallest multiple of a that is not less than e_{N-1}. If this value is not representable in the type *index_type*, the behavior of this function is undefined.

Member Aliases

using **extents_type** = *E*

using **rank_type** = typename *extents_type*::rank_type

using **index_type** = typename *extents_type*::index_type

using **layout_type** = ct::*layout_right_padded*<Alignment>

layout_right_padded_mapping

template<typename **T**>
requires *is-convertible-v*<*T*, *index_type*>
__tile__ __host__ __device__ **layout_right_padded_mapping**(*extents_type* const &other, *T* alignment) noexcept

Constructs an instance of this layout mapping by direct-list-initializing *__extents* with *other*. When *__alignment* is present, it is direct-initialized with *alignment*.

is_always_strided

__tile__ __host__ __device__ static constexpr bool **is_always_strided**() noexcept

Returns `true`.

static_stride

__tile__ __host__ __device__ static constexpr size_t **static_stride**(*rank_type* i) noexcept

Returns the static stride at dimension i. Behaves as if by invoking

`ct::layout_right_mapping<__padded_extents_t>::static_stride(i)`

stride

__tile__ __host__ __device__ *index_type* **stride**(*rank_type* i) const noexcept

Returns the stride at dimension i. Behaves as if by invoking

`ct::layout_right_mapping{__padded_extents()}.stride(i)`.

extents

__tile__ __host__ __device__ *extents_type* const &**extents**() const noexcept
Yields the glvalue *__extents*.

Deduction Guides

template<typename **E**, typename **A**>
layout_right_padded_mapping(*E* const&, *A*) -> *layout_right_padded_mapping*<*E*, extent-constant-or-dynamic<*A*>>;

Deduction guide enabling class template argument deduction from an *ct::integral_constant* or integral alignment argument.

2.10.7. cuda::tiles::layout_left_padded

template<size_t **A**>
struct **layout_left_padded**

Layout policy representing a column-major element order whose first dimension is padded to ensure alignment of consecutive columns. The template parameter A specifies the desired alignment of each column. If A is *ct::dynamic_extent*, the alignment is instead supplied as a run-time argument when constructing an instance of *layout_left_padded_mapping*.

In a padded column-major layout, the first multi-dimensional index varies the fastest with the increasing elements in memory. Additionally, the position of each element that anchors a column is divisible by the desired alignment.

Example

Constructs a 2×3 *ct::tensor_span* over the elements $0, 1, \ldots, 7$ where each column is aligned to 3.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
int x[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Syntax 1
ct::layout_left_padded_mapping m1{ct::extents{4_ic, 2_ic}, 6_ic};
ct::tensor_span p1{&x[0], m1};

// Syntax 2
ct::layout_left_padded_mapping m2{ct::extents{4_ic, 2_ic}, 3_ic};
ct::tensor_span p2{&x[0], m2};
```

$$\begin{pmatrix} 0 & 6 \\ 1 & 7 \\ 2 & 8 \\ 3 & 9 \end{pmatrix}$$

---

template<ct::*extents_like* **E**>

using **mapping** = ct::*layout_left_padded_mapping*<*E*, Alignment>

Alias template yielding the associated *layout mapping* type for representing a padded column-major arrangement of an array whose shape is an instance of E.

2.10.8. cuda::tiles::layout_left_padded_mapping

template<ct::*extents_like* **E**, size_t **Alignment**>
struct **layout_left_padded_mapping**

Layout mapping representing a column-major element order whose shape is an instance of E. The first dimension is padded to ensure alignment of consecutive columns. For an example, see `ct::layout_left_padded`.

The template parameter A specifies the desired alignment of each column. If A is `ct::dynamic_extent`, the alignment is instead supplied as a runtime argument when constructing an instance of `layout_left_padded_mapping`.

A specialization T of `ct::layout_left_padded_mapping` models the requirements of *layout mapping*. Additionally, T satisfies the following constraints if the corresponding constraint is satisfied by the underlying *extents like* type E:

- `std::is_nothrow_move_constructible_v<T>`[Page 125, 1]
- `std::is_nothrow_move_assignable_v<T>`[Page 125, 1]
- `std::is_nothrow_swappable_v<T>`[Page 125, 1]

The program is ill-formed if E is cv-qualified.

Exposition Only Members

extents_type **__extents**

Exposition-only member denoting this object's *layout mapping shape*.

index_type **__alignment**;

Exposition-only member denoting the desired alignment of each column specified at runtime. This member is present only when A is `ct::dynamic_extent`.

using **__padded_extents_t** = /* see below */

Exposition-only member denoting the type of a *padded extents* object. Let E denote `extents_type` and let N be its rank.

The *padded extents* type is the unique `ct::extents` specialization S satisfying:

1. The index type and *rank* of S match those of E.
2. The static extents of S match those of E with the possible exception of the dimension 0 when $N \neq 0$.
3. If $N \neq 0$, the value S_0 is
   1. `ct::dynamic_extent` if either A or E_0 is `ct::dynamic_extent`.
   2. The smallest multiple of A that is not less than E_0 otherwise. If this value is not representable in the type `size_t`, the behavior is undefined when constructing an instance of this object.

__tile__ __host__ __device__ constexpr *__padded_extents_t* **__padded_extents**( ) const noexcept

Exposition only function that yields a *padded extents* object based on the values of *__extents* and *__alignment* when present. Let e denote *__extents* and let N be its rank. Let a denote *__alignment* when present and `A` otherwise. The returned value is the unique instance s of *__padded_extents_t* satisfying:

1. The extent values of s match those of e with the possible exception of of dimension 0 when $N \neq 0$.
2. When $N \neq 0$, the value s_0 is the smallest multiple of a that is not less than e_0. If this value is not representable in the type *index_type*, the behavior of this function is undefined.

Member Aliases

using **extents_type** = *E*

using **rank_type** = typename *extents_type*::rank_type

using **index_type** = typename *extents_type*::index_type

using **layout_type** = ct::*layout_left_padded*<*Alignment*>

Constructor

template<typename **T**>
requires *is-convertible-v*<*T*, *index_type*>
__tile__ __host__ __device__ **layout_left_padded_mapping**(*extents_type* const &other, *T* alignment) noexcept

Constructs an instance of this layout mapping by direct-list-initializing *__extents* with *other*. When *__alignment* is present, it is direct-initialized with *alignment*.

is_always_strided

__tile__ __host__ __device__ static constexpr bool **is_always_strided**( ) noexcept

Returns `true`.

static_stride

__tile__ __host__ __device__ static constexpr size_t **static_stride**(*rank_type* i) noexcept

Returns the static stride at dimension i. Behaves as if by invoking

`ct::`*layout_left_mapping*`<`*__padded_extents_t*`>::static_stride(`*i*`)`

stride

__tile__ __host__ __device__ *index_type* **stride**(*rank_type* i) const noexcept

Returns the stride at dimension i.

Behaves as if by invoking

`ct::layout_left_mapping{__padded_extents()}.stride(i)`

extents

__tile__ __host__ __device__ *extents_type* const &**extents**() const noexcept

Yields the glvalue `__extents`.

Deduction Guides

template<typename **E**, typename **A**>
layout_left_padded_mapping(*E* const&, *A*) -> *layout_left_padded_mapping*<*E*, extent-constant-or-dynamic<*A*>>;

Enables class template argument deduction from an `ct::integral_constant` or integral alignment argument.

2.10.9. cuda::tiles::layout_strided

template<ct::*extents_like* **S**>
struct **layout_strided**

Layout policy representing a strided arrangement of elements. The stride values are given by an instance of S when constructing the corresponding `layout_strided_mapping` and may include statically known or dynamically known stride values.

Example

The code below creates a 2×3 tensor span over a sparse set of elements in the underlying array. Increment the first dimension moves 6 elements in memory while incrementing the second dimension moves 2 elements in memory.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
int x[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

ct::layout_strided_mapping m{ct::extents{2_ic, 3_ic}, ct::extents{6_ic, 2_ic}};
ct::tensor_span p{&x[0], m};
```

$$\begin{pmatrix} 0 & 2 & 4 \\ 6 & 8 & 10 \end{pmatrix}$$

---

template<ct::*extents_like* **E**>

using **mapping** = ct::*layout_strided_mapping*<*E*, Strides>

Alias template yielding the associated *layout mapping* type for representing a strided layout for an array whose shape is an instance of E.

2.10.10. cuda::tiles::layout_strided_mapping

template<ct::*extents_like* **E**, ct::*extents_like* **S**>
requires /* atomic constraint */
struct **layout_strided_mapping**

Layout mapping representing a layout with arbitrary strides for a shape that is an instance of E. The strides for each dimension may be known either statically or at runtime and are determined by an instance of S provided at construction.

A specialization T of *ct::layout_strided_mapping* models the requirements of *layout mapping*. Additionally, T satisfies the following constraints if the corresponding constraint is satisfied by the underlying *extents like* type E:

- `std::is_nothrow_move_constructible_v<T>`[Page 125, 1]
- `std::is_nothrow_move_assignable_v<T>`[Page 125, 1]
- `std::is_nothrow_swappable_v<T>`[Page 125, 1]

The program is ill-formed if E or S is cv-qualified.

The atomic constraint validates that E and S have the same *rank* and index type.

Exposition Only Members

The following exposition only members exist for the purposes of defining the implicitly defined behavior of the special member functions.

extents_type **__extents**

Exposition-only member denoting this object's *layout mapping shape*.

S **__strides**

Exposition-only member denoting the strides of this layout mapping.

Types

using **extents_type** = *E*
using **rank_type** = typename *extents_type*::rank_type
using **index_type** = typename *extents_type*::index_type
using **layout_type** = ct::*layout_strided*<*S*>

layout_strided_mapping

__tile__ __host__ __device__ **layout_strided_mapping**(*extents_type* const &other, *S* const &strides) noexcept

Constructs an instance of this layout mapping by performing direct-list-initializing of *__extents* with `other` and direct-list-initialization of *__strides* with `strides`.

is_always_strided

__tile__ __host__ __device__ static constexpr bool **is_always_strided**() noexcept

Returns `true`.

static_stride

__tile__ __host__ __device__ static constexpr size_t **static_stride**(*rank_type* i) noexcept

Returns the static stride at dimension i as if by invoking *S*`::static_extent(`*i*`)`. The behavior is undefined if i is not in the range $[0, N)$ where N is the *rank* of *E*.

stride

__tile__ __host__ __device__ *index_type* **stride**(*rank_type* i) const noexcept

Returns the stride at dimension i as if by invoking *__strides*`.extent(`*i*`)`. The behavior is undefined if i is not in the range $[0, N)$ where N is the rank of *E*.

extents

__tile__ __host__ __device__ *extents_type* const &**extents**() const noexcept

Yields the glvalue *__extents*.

2.10.11. Layout Comparison

template<ct::*layout_mapping* **Lhs**, ct::*layout_mapping* **Rhs**>
__tile__ __host__ __device__ constexpr bool **operator==**(*Lhs* const &lhs, *Rhs* const &rhs) noexcept;

Indicates whether `lhs` and `rhs` are *layout mapping equivalent*.

Note: A corresponding overload for `operator!=` is available via rewritten operator candidates[2].

---

[2] See § 12.2.2.3 [over.match.oper] of ISO/IEC 14882:2024

2.10.12. cuda::tiles::layout_mapping_static_stride

template<typename **T**>
struct **layout_mapping_static_stride**

Traits class which may be specialized to provide the static stride values when implementing a custom *layout mapping* type. The primary template's implementation delegates to a `T::static_stride` member function if available.

__tile__ __host__ __device__ constexpr size_t **operator()**(typename *T*::rank_type idx) noexcept;

Yields the value `T::static_stride(idx)`. This function participates in overload resolution if the expression `T::static_stride(idx)` is well formed and yields a prvalue of type `size_t`.

2.11. Tensor Span

A *`ct::tensor_span`* is a *tensor span like* type representing a view to a multi-dimensional array in memory.

Example

The following example creates a 2×4 row major view of elements of the x array. The length 2 is supplied at runtime while 4 is supplied at compile time.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
int x[8] = {0, 1, 2, 3, 4, 5, 6, 7};

ct::tensor_span t{&x[0], ct::extents{2, 4_ic}};
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

2.11.1. cuda::tiles::tensor_span

template<
ct::*scalar* **E**,
ct::*extents_like* **Extents**,
typename **Layout** = ct::*layout_right*,
ct::*accessor_policy* **Accessor** = ct::*default_accessor*<*E*>
>
requires ct::*layout_mapping*<*Layout*::mapping<*Extents*>> && is-same-v<*E*, *Accessor*::element_type>
struct **tensor_span**

A *`ct::tensor_span`* is an implementation of the *tensor span like* abstraction. It is a handle to an in-memory multi-dimensional array along with a *layout mapping* object describing the array's arrangement of elements and an *accessor policy* object providing customization for how elements are accessed.

A *ct::tensor_span* can represent arbitrary array shapes with mixed static and dynamic size and stride information.

All specializations of *ct::tensor_span* model *tensor span like*.

A specialization of *ct::tensor_span* may additionally satisfy the following constraints if the underlying layout mapping type *Layout*::mapping<*Extents*> and accessor policy *Accessor* satisfies them:

1. std::is_nothrow_move_constructible_v
2. std::is_nothrow_move_assignable_v
3. std::is_nothrow_swappable_v

The specialization is ill-formed if *Extents*, *Layout*, or *Accessor* is cv-qualified.

Exposition Only Helpers

For the purposes of defining the behavior of the implicitly defined special member functions, the following members exist.

[[no_unique_address]] *mapping_type* **__mapping**

Exposition only member variable indicating the layout of the multi-dimensional array referenced by this object.

[[no_unique_address]] *accessor_type* **__accessor**

Exposition only member describing the accessor policy for this tensor span.

[[no_unique_address]] *data_handle_type* **__handle**

Exposition only pointer to the multi-dimensional array referenced by this object.

Member Aliases

using **element_type** = *E*;
using **value_type** = *remove-cv-t*<*E*>
using **extents_type** = *Extents*
using **layout_type** = *Layout*
using **mapping_type** = *layout_type*::mapping<*extents_type*>
using **index_type** = *extents_type*::index_type
using **rank_type** = *extents_type*::rank_type
using **accessor_type** = *Accessor*
using **data_handle_type** = *accessor_type*::data_handle_type
using **reference** = *accessor_type*::reference

Construction From Mapping

__tile__ __host__ __device__ constexpr **tensor_span**(*data_handle_type* handle, *mapping_type* const &mapping, *accessor_type* const &accessor = {}) noexcept;

Direct list initializes *__handle* from *handle*, *__mapping* from *mapping*, and *__accessor* from *accessor*.

Construction From Policy

__tile__ __host__ __device__ constexpr **tensor_span**(*data_handle_type* handle, *extents_type* const &extents, *layout_type* const &layout = {}, *accessor_type* const &accessor = {})

Direct list initializes *__handle* from *handle*, *__mapping* from *extents*, and *__accessor* from *accessor*. This function participates in overload resolution only if *is-constructible-v*<*extents_type* const&, *mapping_type*> is `true`.

Note: The argument *layout* is functionally ignored but may be used for the purposes of class template argument deduction.

---

rank

__tile__ __host__ __device__ static constexpr *rank_type* **rank**() noexcept;

Yields *mapping_type*`::rank()`.

Rank Dynamic

__tile__ __host__ __device__ static constexpr *rank_type* **rank_dynamic**() noexcept

Yields *mapping_type*`::rank_dynamic()`.

static_extent

__tile__ __host__ __device__ static constexpr ct::size_t **static_extent**(*rank_type* dim) noexcept

Yields *extents_type*`::static_extent(`*dim*`)`.

extent

__tile__ __host__ __device__ constexpr *index_type* **extent**(*rank_type* dim) const noexcept
Yields *__mapping*.extents().extent(*dim*).

extents

__tile__ __host__ __device__ constexpr *extents_type* const &**extents**() const noexcept;
Yields *__mapping*.extents().

mapping

__tile__ __host__ __device__ constexpr *mapping_type* const &**mapping**() const noexcept;
Yields *__mapping*.

accessor

__tile__ __host__ __device__ constexpr *accessor_type* const &**accessor**() const noexcept;
Yields *__accessor*.

data_handle

__tile__ __host__ __device__ constexpr *data_handle_type* **data_handle**() const noexcept;
Yields *__handle*.

Deduction Guides

template<typename **T**, typename **M**, typename **A** = ct::*default_accessor*<*T*>>
tensor_span(*T**, *M* const&, *A* const& = {}) -> *tensor_span*<*T*, *M*::extents_type, *M*::layout_type, *A*>;
template<typename **T**, typename **E**, typename **L** = ct::*layout_right*, typename **A** = ct::*default_accessor*<*T*>>
tensor_span(*T**, *E* const&, *L* const& = {}, *A* const& = {}) -> *tensor_span*<*T*, *E*, *L*, *A*>;

Enables deduction of class template arguments from constructor arguments.

2.11.2. cuda::tiles::storeable_tensor_span

template<typename **T**>
concept **storeable_tensor_span** = *tensor_span_like*<*T*> && /* atomic constraint */

Indicates whether T is a *tensor span like* type that may participate in a store operation. The atomic constraint validates that the *element type* of T is not cv-qualified.

2.11.3. cuda::tiles::default_accessor

template<ct::*scalar* **E**>
struct **default_accessor**

A *default_accessor* is an *accessor policy* for simple contiguous memory accesses.

Aliases

using **element_type** = *E*
using **data_handle_type** = *E**
using **reference** = *E*&

Specializations

template<ct::*scalar* **E**>
constexpr bool **enable_contiguous_accessor_policy**<ct::*default_accessor*<*E*>> = true

Specialization of variable template `enable_contiguous_accessor_policy` for *ct::default_accessor*.

2.11.4. cuda::tiles::enable_contiguous_accessor_policy

template<typename **T**>
inline constexpr bool **enable_contiguous_accessor_policy** = false

Trait variable which may be specialized to mark custom types as a contiguous *accessor policy*.

2.12. Partition View

A *ct::partition_view* is a wrapper around a *tensor span like* type that divides it into statically sized tiles which may be loaded and stored from memory.

Example

The following code subdivides the provided 4×8 tensor span into partitions of size 2×2. The partition identified by index $(1, 2)$ is loaded.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
int x[4][8] = {
  {0, 1, 2, 3, 4, 5, 6, 7},
  {8, 9, 10, 11, 12, 13, 14, 15},
  {16, 17, 18, 19, 20, 21, 22, 23},
  {24, 25, 26, 27, 28, 29, 30, 31}
};
```

(continues on next page)

(continued from previous page)

```
ct::tensor_span t{&x[0][0], ct::extents{4_ic, 8_ic}};
ct::partition_view p{t, ct::shape{2_ic, 2_ic}};
auto r = p.load(1, 2);
```

$$\left(\begin{array}{cc|cc|cc|cc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \end{array}\right) \rightarrow \begin{pmatrix} 20 & 21 \\ 28 & 29 \end{pmatrix}$$

2.12.1. cuda::tiles::partition_view

template<ct::*tensor_span_like* **Span**, ct::*tile_shape* **Shape**>
requires (*Shape*::rank() == *Span*::extents_type::rank())
struct **partition_view**

All specializations T of *ct::partition_view* model `std::copyable`.

T satisfies the following constraints if they are satisfied by Span:

1. `std::is_nothrow_move_constructible_v<T>`
2. `std::is_nothrow_move_assignable_v<T>`
3. `std::is_nothrow_swappable_v<T>`

Aliases

using **span_type** = *Span*

The type of the wrapped *tensor span*.

using **view_shape_type** = *Shape*

The *shape* of the tile that will be loaded or stored.

using **element_type** = typename *span_type*::element_type

using **value_type** = typename *span_type*::value_type

using **index_type** = typename *span_type*::index_type

using **view_tile_type** = ct::*tile*<*value_type*, *view_shape_type*>

Exposition Only Members

span_type **__span**

Exposition only member containing the wrapped *tensor span* object. This member exists for the purpose of defining the behavior of the implicitly defined special member functions.

Exposition Only Definitions

partition view mapping

The *partition view mapping* is a function that associates an index of a partition to potential set of indices of the underlying *tensor span* which are to be loaded or stored.

Let S denote *view_shape_type* and let N be its rank.

Let $I = (i_0, i_1, \ldots, i_{N-1})$ be a *partition index* and $J = (j_0, j_1, \ldots, j_{N-1})$ be an index in the *index space* of S.

The mapping is a new index

$$p(I, J) = (i_0 \cdot S_0 + j_0, \ldots, i_{N-1} \cdot S_{N-1} + j_{N-1})$$

Note: The result of the mapping might not be an element in the tensor span's index space.

---

partition view index space

The index space of the partition view is the set of partition indices which correspond to in bounds or partially out of bound partitions.

Let S denote *view_shape_type*, let e denote *span_type*`::extents_type` and let N be their rank. The index space is the set of non-negative indices $I = (i_0, i_1, \ldots, i_{N-1})$ satisfying

$$i_k \cdot S_k < e_k \quad 0 \leq k < N$$

Constructor

__tile__ __host__ __device__ **partition_view**(*span_type* span, *view_shape_type*)

Constructs this object by direct-list-initializing *__span* from span.

Loads

```
template<
typename ...Idx
>
requires /* atomic constraint */
__tile__ view_tile_type load(Idx...) const noexcept;
template<
ct::view_padding Pad = ct::default_view_padding(),
typename ...Idx
>
requires /* atomic constraint */
__tile__ view_tile_type load_masked(Idx...) const noexcept;
template<
ct::view_padding Pad = ct::default_view_padding(),
typename ...Idx
>
requires /* atomic constraint */
__tile__ view_tile_type load_masked(ct::view_padding_constant<Pad>, Idx...) const noexcept;
template<
```

```
    ct::memory_order Order,
    ct::thread_scope Scope = ct::default_thread_scope(),
    typename ...Idx
    >
    requires ct::read_memory_order<Order> && /* atomic constraint */
    __tile__ view_tile_type atomic_load(Idx...) const noexcept;
    template<
    ct::memory_order Order,
    ct::thread_scope Scope = ct::default_thread_scope(),
    typename ...Idx
    >
    requires ct::read_memory_order<Order> && /* atomic constraint */
    __tile__ view_tile_type atomic_load(ct::memory_order_constant<Order>, Idx...) const
                                        noexcept;
    template<
    ct::memory_order Order,
    ct::thread_scope Scope = ct::default_thread_scope(),
    typename ...Idx
    >
    requires ct::read_memory_order<Order> && /* atomic constraint */
    __tile__ view_tile_type atomic_load(ct::memory_order_constant<Order>,
                                        ct::thread_scope_constant<Scope>, Idx...) const
                                        noexcept;
    template<
    ct::view_padding Pad = ct::default_view_padding(),
    ct::memory_order Order,
    ct::thread_scope Scope = ct::default_thread_scope(),
    typename ...Idx
    >
    requires ct::read_memory_order<Order> && /* atomic constraint */
    __tile__ view_tile_type atomic_load_masked(Idx...) const noexcept;
    template<
    ct::view_padding Pad = ct::default_view_padding(),
    ct::memory_order Order,
    ct::thread_scope Scope = ct::default_thread_scope(),
    typename ...Idx
    >
    requires ct::read_memory_order<Order> && /* atomic constraint */
    __tile__ view_tile_type atomic_load_masked(ct::view_padding_constant<Pad>, Idx...) const
                                               noexcept;
    template<
    ct::view_padding Pad = ct::default_view_padding(),
    ct::memory_order Order,
    ct::thread_scope Scope = ct::default_thread_scope(),
    typename ...Idx
    >
    requires ct::read_memory_order<Order> && /* atomic constraint */
    __tile__ view_tile_type atomic_load_masked(ct::view_padding_constant<Pad>,
                                               ct::memory_order_constant<Order>, Idx...) const
                                               noexcept;
    template<
    ct::view_padding Pad = ct::default_view_padding(),
    ct::memory_order Order,
```

```
ct::thread_scope Scope = ct::default_thread_scope(),
typename ...Idx
>
requires ct::read_memory_order<Order> && /* atomic constraint */
__tile__ view_tile_type atomic_load_masked(ct::view_padding_constant<Pad>,
                                           ct::memory_order_constant<Order>,
                                           ct::thread_scope_constant<Scope>, Idx...) const
                                           noexcept;
```

Loads a tile from the partition specified by the indices `Idx`.

Example

In the following example, a 4×11 tensor span is partitioned into 2×4 chunks. A partially out of bounds chunk is loaded using NaN padding values.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
float x[4][11] = {
  {0  , 1  , 2  , 3  , 4  , 5  , 6  , 7  , 8  , 9  , 10},
  {11 , 12 , 13 , 14 , 15 , 16 , 17 , 18 , 19 , 20 , 21},
  {22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 , 32},
  {33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 , 41 , 42 , 43}
};

ct::tensor_span t{&x[0][0], ct::extents{4_ic, 11_ic}};
ct::partition_view p{t, ct::shape{2_ic, 4_ic}};

auto r = p.load_masked(ct::view_padding_nan_t{}, 0, 2);
```

$$\left(\begin{array}{cccc|cccc|ccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ \hline 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 \end{array}\right) \rightarrow \begin{pmatrix} 8 & 9 & 10 & \text{NaN} \\ 19 & 20 & 21 & \text{NaN} \end{pmatrix}$$

Let I denote the values specified by the pack `Idx` and let J be an index in the *index space* of *`view_tile_type`*.

The value returned by the load is a tile object a satisfying

$$a(J) = t(p(I, J))$$

where p is the *partition view mapping* of this object and t is the *tensor span function* of *`__span`*.

If the value $p(I, J)$ is not in *index space* of *`__span`*, the behavior depends on the selected overload:

- For the non-masked overloads, the behavior is undefined.
- For masked overloads, the value of $a(J)$ is the *view padding* specified by Pad.

The behavior is undefined if any of the following hold:

1. a value of the `Idx` pack is not representable in *`index_type`*
2. I is outside the partition view's *index space*

3. The *tensor span function* is not injective

An invocation generates a *read memory operation* at the location $t(p(I, J))$ for each J in the index space of *__span* which is not masked.

For the atomic overloads, the memory operations are strong and have *thread scope* specified by Scope and *memory order* specified by `Order`

The *latency* and *allow_tma* optimization hints may appertain to direct call expressions of the above load APIs.

The atomic constraint requires that

1. The size of the `Idx` pack matches the rank of *view_shape_type*.
2. Each element of the `Idx` pack *scalar convertible* to *index_type*
3. If Pad is present and its value is not *zero view padding*, then *element_type* is a *basic floating point scalar*.
4. When specified, the values Pad, `Order`, and Scope are all enumerators of their respective types.

Note: The indices *Idx* specify which partition should be loaded. A fully out of bounds partition always yields undefined behavior. A partially out of bounds partition yields undefined behavior for the non-masked variants.

Stores

template<
ct::*tile_like* **Value**,
typename ...**Idx**
>
requires /* atomic constraint */ && ct::*non_narrowing_tile_convertible_to*<*Value*, *view_tile_type*>
&& ct::*storeable_tensor_span*<*span_type*>
__tile__ void **store**(*Value* a, *Idx*...) const noexcept;

template<
ct::*memory_order* **Order**,
ct::*thread_scope* **Scope** = ct::*default_thread_scope*(),
ct::*tile_like* **Value**,
typename ...**Idx**
>
requires ct::*write_memory_order*<*Order*> && ct::*non_narrowing_tile_convertible_to*<*Value*,
view_tile_type> && ct::*storeable_tensor_span*<*span_type*> && /* atomic constraint */
__tile__ void **atomic_store**(*Value* a, *Idx*...) const noexcept;

template<
ct::*memory_order* **Order**,
ct::*thread_scope* **Scope** = ct::*default_thread_scope*(),
ct::*tile_like* **Value**,
typename ...**Idx**
>
requires ct::*write_memory_order*<*Order*> && ct::*non_narrowing_tile_convertible_to*<*Value*,
view_tile_type> && ct::*storeable_tensor_span*<*span_type*> && /* atomic constraint*/
__tile__ void **atomic_store**(*Value* a, ct::*memory_order_constant*<*Order*>, *Idx*...) const
noexcept;

```
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::tile_like Value,
typename ...Idx
>
requires ct::write_memory_order<Order> && ct::non_narrowing_tile_convertible_to<Value,
view_tile_type> && ct::storeable_tensor_span<span_type> && /* atomic constraint */
__tile__ void atomic_store(Value a, ct::memory_order_constant<Order>,
                           ct::thread_scope_constant<Scope>, Idx...) const noexcept;
template<
ct::tile_like Value,
typename ...Idx
>
requires ct::non_narrowing_tile_convertible_to<Value, view_tile_type> &&
ct::storeable_tensor_span<span_type> && /* atomic constraint */
__tile__ void store_masked(Value a, Idx...) const noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::tile_like Value,
typename ...Idx
>
requires ct::write_memory_order<Order> && ct::non_narrowing_tile_convertible_to<Value,
view_tile_type> && ct::storeable_tensor_span<span_type> && /* atomic constraint */
__tile__ void atomic_store_masked(Value a, Idx...) const noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::tile_like Value,
typename ...Idx
>
requires ct::write_memory_order<Order> && ct::non_narrowing_tile_convertible_to<Value,
view_tile_type> && ct::storeable_tensor_span<span_type> && /* atomic constraint */
__tile__ void atomic_store_masked(Value a, ct::memory_order_constant<Order>, Idx...) const
                                  noexcept;
template<
ct::memory_order Order,
ct::thread_scope Scope = ct::default_thread_scope(),
ct::tile_like Value,
typename ...Idx
>
requires ct::write_memory_order<Order> && ct::non_narrowing_tile_convertible_to<Value,
view_tile_type> && ct::storeable_tensor_span<span_type> && /* atomic constraint */
__tile__ void atomic_store_masked(Value a, ct::memory_order_constant<Order>,
                                  ct::thread_scope_constant<Scope>, Idx...) const
                                  noexcept;
```

Stores the *tile converted* operand `a` to the partition specified by indices `Idx`.

Example

In the following example, a 4×8 tensor span is partitioned into 2×2 chunks and the bottom

right partition is updated with a new value.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
int x[4][8] = {
  {0, 1, 2, 3, 4, 5, 6, 7},
  {8, 9, 10, 11, 12, 13, 14, 15},
  {16, 17, 18, 19, 20, 21, 22, 23},
  {24, 25, 26, 27, 28, 29, 30, 31}
};

ct::tensor_span t{&x[0][0], ct::extents{4_ic, 8_ic}};
ct::partition_view p{t, ct::shape{2_ic, 2_ic}};

auto a = 100 * ct::iota<ct::tile<int, ct::shape<2, 2>>>();
p.store(a, 1, 3);
```

$$\begin{pmatrix} 0 & 100 \\ 200 & 300 \end{pmatrix} \rightarrow \left(\begin{array}{cc|cc|cc|cc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline 16 & 17 & 18 & 19 & 20 & 21 & 0 & 100 \\ 24 & 25 & 26 & 27 & 28 & 29 & 200 & 300 \end{array}\right)$$

Let I denote the values specified by the pack `Idx` and let J be an index in the *index space* of *`view_tile_type`*. Let a denote the value of a after *tile conversion* to *`view_tile_type`*.

The value $a(J)$ is stored to the memory location

$$t(p(I, J))$$

where p is the *partition view mapping* of this object and t is the *tensor span function* of *`__span`*.

If the value $p(I, J)$ is not in *index space* of *`__span`*, the behavior depends on the selected overload:

- For non-masked overloads, the behavior is undefined
- For masked overloads, no store occurs at that memory location

The behavior is undefined if any of the following hold:

1. a value of the `Idx` pack is not representable in *`index_type`*
2. I is outside the partition view's *index space*
3. The *tensor span function* is not injective

An invocation generates *write memory operations* for the values at the addresses $t(p(I, J))$ for each $J \in \mathbb{J}$ which is not masked.

For the atomic variants, the memory operations are strong and have *thread scope* specified by *`Scope`* and *memory order* specified by *`Order`*

The *latency* and *allow_tma* optimization hints may appertain to direct call expressions of the above store APIs.

The atomic constraint requires that

1. The size of the `Idx` pack matches the rank of *`view_shape_type`*.

2. Each element of the `Idx` pack *scalar convertible* to *`index_type`*
3. When specified, the values `Order`, and `Scope` are all enumerators of their respective types.

span

span_type const &**span**() const noexcept;

Yields the glvalue *`__span`*.

2.13. Constant Wrappers

This section describes wrapper types that carry constant values as well as API modifier flags. Using these types, one can pass compile time information to an API through a function argument instead of an explicit template argument.

Example

The following code shows how to use pass compile time values using both the explicit template argument and constant wrapper techniques.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

// Example 1
ct::shape<2, 4> x;
ct::shape       y{2_ic, 4_ic};

// Example 2
ct::dimension_map<2, 0, 1> w;
ct::dimension_map          z{2_ic, 0_ic, 1_ic};

// Example 3
ct::add<ct::rounding_mode::round_toward_zero>(0.0, 1.0, {});
ct::add(0.0, 1, ct::round_toward_negative_t{});

// Example 4
double* ptr{nullptr};
ct::atomic_load<ct::memory_order::relaxed>(ptr);
ct::atomic_load(ptr, ct::memory_order_relaxed_t{});

// Example 5
ct::partition_view P{
  ct::tensor_span{ptr, ct::shape{4_ic, 8_ic}},
  ct::shape{2_ic, 2_ic}};
P.load_masked<ct::view_padding::positive_inf>(0, 1);
P.load_masked(ct::view_padding_positive_inf_t{}, 0, 1);
```

2.13.1. cuda::tiles::integral_constant

template<ct::*integral* auto **V**>
struct **integral_constant**

A *ct::integral_constant* is a stateless empty type that encodes a compile time integral value.

Aliases

using **type** = *integral_constant*
using **value_type** = decltype(*V*)

Member Variables

static constexpr *value_type* **value** = *V*

Conversion Operator

__tile__ __host__ __device__ constexpr **operator** ***value_type***( ) const noexcept;

Conversion operator yielding the value *V*.

Function Call Operator

__tile__ __host__ __device__ constexpr **operator()**( ) const noexcept;

Yields the value *V*.

2.13.1.1 Overloaded Operators

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* + *y*> **operator+**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* * *y*> **operator***(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* - *y*> **operator-**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* / *y*> **operator/**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>

__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* % *y*> **operator%**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* & *y*> **operator&**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* | *y*> **operator|**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<*x* ^ *y*> **operator^**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<(*x* << *y*)> **operator<<**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**, auto **y**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<(*x* >> *y*)> **operator>>**(ct::*integral_constant*<*x*>, ct::*integral_constant*<*y*>) noexcept;

template<auto **x**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<~x> **operator~**(ct::*integral_constant*<*x*>) noexcept;

template<auto **x**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<+*x*> **operator+**(ct::*integral_constant*<*x*>) noexcept;

template<auto **x**>
__tile__ __host__ __device__ constexpr ct::*integral_constant*<-*x*> **operator-**(ct::*integral_constant*<*x*>) noexcept;

Overloaded operators for *ct::integral_constant*. Each operator yields default constructed instance of its return type.

2.13.1.2 Literal Operator Template

template<char...>
__tile__ __host__ __device__ constexpr ct::*integral_constant*</* see below */> **operator""_ic**() noexcept;

A literal operator template[1] for integer literals suffixed with `_ic`. This function inhabits the `cuda::tiles::literals` namespace.

Yields a default constructed instance of *ct::integral_constant* whose constant value matches the value of the provided integer literal. The value type of the result matches the type the integer literal would have had the `_ic` suffix been omitted[2] except the program is ill-formed if that type would be an extended integer type.

The program is ill-formed if the provided literal is not a decimal integer literal.

[1] See § 12.6 [over.literal] of ISO/IEC 14882:2024
[2] See § 5.13.2 [lex.icon] of ISO/IEC 14882:2024

Example

The following code shows two `_ic` literals and their corresponding types.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

// Type is `int`
22_ic;

// Type is either `long int` or `long long int`.
4294967296_ic;
```

2.13.2. cuda::tiles::dimension_map

template<size_t... **Idx**>
requires /* atomic constraint */
struct **dimension_map**;

Stateless empty type representing a permutation of the integers 0 through `sizeof...(Idx)` exclusive. Element k of the `Idx` pack indicates which dimension of the source should be placed at dimension k in the target during a *ct::permute()* operation. The *rank* of a `ct::dimension_map` is the size of the `Idx` pack.

The atomic constraint validates the `Idx` pack is a permutation of the integers 0 through $N-1$ inclusive where N is the rank. That is, each of the `Idx` values lies in the half open range $[0N)$ and there are no duplicates. An empty `Idx` pack represents a valid permutation.

Default Construction

__tile__ __host__ __device__ constexpr **dimension_map**() noexcept = default;

Default constructs an instance of this type.

Construction From Arguments

template<typename ...**Ts**>
requires (sizeof...(*Ts*) > 0)
__tile__ __host__ __device__ explicit constexpr **dimension_map**(*Ts*...) noexcept;

Constructs an instance of this type, discarding any arguments that were passed to it. This constructor is used in conjunction with the deduction guide to enable CTAD based construction from *ct::integral_constant* values.

rank

__tile__ __host__ __device__ static constexpr size_t **rank**() noexcept;

Yields the size of the *Idx* pack.

mapping

__tile__ __host__ __device__ static constexpr size_t **mapping**(rank_type i) noexcept;

Yields the i^{th} element of pack *Idx*. The behavior is undefined if $i \geq N$ where N is parameter pack size.

Deduction Guide

template<auto... **Vs**>
requires /* atomic constraint */
dimension_map(*integral_constant*<*Vs*>...) -> *dimension_map*<size_t(*Vs*)...>;

Deduction guide enabling CTAD from *ct::integral_constant* arguments. The atomic constraint validates that the expression `size_t(Vs)` is well formed for each element of the `Vs` pack.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

// Type: ct::dimension_map<2, 0, 1>
ct::dimension_map{2_ic, 0_ic, 1_ic};
```

2.13.3. Rounding Mode

APIs for working with *round modes*.

2.13.3.1 cuda::tiles::rounding_mode

enum class **rounding_mode** : /* unspecified */

enumerator **round_ties_to_even** = 0
enumerator **round_toward_zero** = 1
enumerator **round_toward_negative** = 2
enumerator **round_toward_positive** = 3
enumerator **round_approximate** = 4
enumerator **round_full** = 5

Type enumerating the available *rounding modes*.

2.13.3.2 cuda::tiles::rounding_mode_constant

template<ct::*rounding_mode* **Value**>
requires /* atomic constraint */
struct **rounding_mode_constant**;

A specialization of `rounding_mode_constant` encodes a *rounding mode* in its type. The atomic constraint validates that `Value` is an enumerator of *`ct::rounding_mode`*.

Example

Example usage of `rounding_mode_constant` to infer the rounding mode non-type template parameter of *`ct::add()`* using a function argument.

```
namespace ct = ::cuda::tiles;
ct::rounding_mode_constant<
    ct::rounding_mode::round_toward_negative> mode;
auto result = ct::add(1.0, 2.5, mode);
```

Member Aliases

using **type** = *rounding_mode_constant*;

using **value_type** = ct::*rounding_mode*;

Member Variables

static constexpr ct::*rounding_mode* **value** = *Value*;

Member Functions

__tile__ __host__ __device__ constexpr **operator ct::*rounding_mode*()** const noexcept;

Conversion operator yielding *`Value`*.

__tile__ __host__ __device__ constexpr ct::*rounding_mode* **operator()**() const noexcept;

Call operator yielding *`Value`*.

2.13.3.3 Constant Aliases

using **round_ties_to_even_t** = ct::*rounding_mode_constant*<ct::*rounding_mode*::*round_ties_to_even*>;

using **round_toward_zero_t** = ct::*rounding_mode_constant*<ct::*rounding_mode*::*round_toward_zero*>;

using **round_toward_negative_t** = ct::*rounding_mode_constant*<ct::*rounding_mode*::*round_toward_negative*>;

using **round_toward_positive_t** = ct::*rounding_mode_constant*<ct::*rounding_mode*::*round_toward_positive*>;

using **round_full_t** = ct::*rounding_mode_constant*<ct::*rounding_mode*::*round_full*>;

using **round_approximate_t** = ct::*rounding_mode_constant*<ct::*rounding_mode*::*round_approximate*>;

using **default_rounding_mode_t** = ct::*rounding_mode_constant*<ct::*default_rounding_mode*()>;

Convenience type aliases for inferring the rounding mode non-type template parameter of arithmetic and math APIs.

Example

Example usage of the rounding mode constant aliases:

```
namespace ct = ::cuda::tiles;
auto result = ct::add(1.0, 2.5, ct::round_toward_negative_t{});
```

2.13.3.4 cuda::tiles::default_rounding_mode

__tile__ __host__ __device__ constexpr ct::*rounding_mode* **default_rounding_mode**() noexcept;

Yields the *default rounding mode* which is *Round Ties to Even*.

2.13.4. Subnormals Rounding Mode

APIs for working with *subnormals rounding modes*.

2.13.4.1 cuda::tiles::subnormals_rounding_mode

enum class **subnormals_rounding_mode** : /* unspecified */

enumerator **preserve_subnormals** = 0

enumerator **round_subnormals_to_zero** = 1

Type enumerating the available *subnormals rounding modes*.

2.13.4.2 cuda::tiles::subnormals_rounding_mode_constant

template<ct::*subnormals_rounding_mode* **Value**>
requires /* atomic constraint */
struct **subnormals_rounding_mode_constant**;

A specialization of `subnormals_rounding_mode_constant` encodes a *subnormals rounding mode* in its type. The atomic constraint validates that `Value` is one of the enumerators of *`ct::subnormals_rounding_mode`*.

Example

Example usage of `subnormals_rounding_mode_constant` to infer the subnormals rounding mode non-type template parameter of *`ct::add()`* using a function argument.

```
namespace ct = ::cuda::tiles;
ct::subnormals_rounding_mode_constant<
   ct::subnormals_rounding_mode::round_subnormals_to_zero> mode;
auto result = ct::add(1.0f, 2.5f, {}, mode);
```

Member Aliases

using **type** = *subnormals_rounding_mode_constant*;

using **value_type** = ct::*subnormals_rounding_mode*;

Member Variables

static constexpr ct::*subnormals_rounding_mode* **value** = *Value*;

Member Functions

__tile__ __host__ __device__ constexpr **operator ct::*subnormals_rounding_mode***() const noexcept;

Conversion operator yielding *Value*.

__tile__ __host__ __device__ constexpr ct::*subnormals_rounding_mode* **operator()**() const noexcept;

Call operator yielding *Value*.

2.13.4.3 Constant Aliases

using **preserve_subnormals_t** = ct::*subnormals_rounding_mode_constant*<ct::*subnormals_rounding_mode*::*preserve_subnormals*>;

using **round_subnormals_to_zero_t** = ct::*subnormals_rounding_mode_constant*<ct::*subnormals_rounding_mode*::*round_subnormals_to_zero*>;

using **default_subnormals_rounding_mode_t** = ct::*subnormals_rounding_mode_constant*<ct::*default_subnormals_rounding_mode*()>;

Convenience type aliases for inferring the subnormals rounding mode non-type template parameter of arithmetic and math APIs.

Example

Example usage of the subnormals rounding mode constant aliases:

```
namespace ct = ::cuda::tiles;
auto result = ct::add(1.0f, 2.5f, {},
                      ct::round_subnormals_to_zero_t{});
```

2.13.4.4 cuda::tiles::default_subnormals_rounding_mode

__tile__ __host__ __device__ constexpr ct::*subnormals_rounding_mode* **default_subnormals_rounding_mode**( ) noe

Returns the *default subnormals rounding mode* which is *Preserve Subnormals*.

2.13.5. NaN Propagation Mode

APIs for working with *NaN propagation modes*.

2.13.5.1 cuda::tiles::nan_propagation_mode

enum class **nan_propagation_mode** : /* unspecified */

enumerator **suppress_nan** = 0

enumerator **propagate_nan** = 1

Type enumerating the available *NaN propagation modes*.

2.13.5.2 cuda::tiles::nan_propagation_constant

template<ct::*nan_propagation_mode* **Value**>
requires /* atomic constraint */
struct **nan_propagation_mode_constant**;

A specialization of `ct::nan_propagation_mode` encodes a *NaN propagation modes* in its type. The atomic constraint validates that *`Value`* is an enumerator of *`ct::nan_propagation_mode`*.

Member Aliases

using **type** = *nan_propagation_mode_constant*;

using **value_type** = ct::*nan_propagation_mode*;

Member Variables

static constexpr ct::*nan_propagation_mode* **value** = *Value*;

Member Functions

__tile__ __host__ __device__ constexpr **operator ct::*nan_propagation_mode***( ) const noexcept;

Yields the propagation mode constant *`Value`*.

__tile__ __host__ __device__ constexpr ct::*nan_propagation_mode* **operator()**( ) const noexcept;

Yields the propagation mode constant *`Value`*.

2.13.5.3 Constant Aliases

using **suppress_nan_t** = ct::*nan_propagation_mode_constant*<ct::*nan_propagation_mode*::*suppress_nan*>;

using **propagate_nan_t** = ct::*nan_propagation_mode_constant*<ct::*nan_propagation_mode*::*propagate_nan*>;

using **default_nan_propagation_mode_t** = ct::*nan_propagation_mode_constant*<ct::*default_nan_propagation_mode*()>;

Convenience type aliases for inferring the NaN propagation mode non-type template parameter of arithmetic APIs.

Example

Example usage of the NaN propagation mode constant aliases:

```
namespace ct = ::cuda::tiles;
auto result = ct::max(1.0, 2.5,
                      ct::propagate_nan_t{});
```

2.13.5.4 cuda::tiles::default_nan_propagation_mode

__tile__ __host__ __device__ inline constexpr ct::*nan_propagation_mode* **default_nan_propagation_mode**( );

Yields the *default nan propagation mode*.

2.13.6. View Padding

APIs for working with *view padding* values.

2.13.6.1 cuda::tiles::view_padding

enum class **view_padding** : /* unspecified */

enumerator **zero** = 0

enumerator **negative_zero** = 1

enumerator **positive_inf** = 2

enumerator **negative_inf** = 3

enumerator **nan** = 4

Enumeration of the supported *padding values* for masked view loads.

2.13.6.2 cuda::tiles::view_padding_constant

template<ct::*view_padding* **Value**>
requires /* atomic constraint */
struct **view_padding_constant**;

A specialization of `ct::view_padding_constant` encodes a *view padding* value in its type. The atomic constraint validates that `Value` is an enumerator of *`ct::view_padding`*.

Member Aliases

using **type** = *view_padding_constant*;
using **value_type** = ct::*view_padding*;

Member Variables

static constexpr ct::*view_padding* **value** = *Value*;

Member Functions

__tile__ __host__ __device__ constexpr **operator ct::*view_padding***( ) const noexcept;
Yields the value *`Value`*.

__tile__ __host__ __device__ constexpr ct::*view_padding* **operator()**( ) const noexcept;
Yields the value *`Value`*.

2.13.6.3 Constant Aliases

using **view_padding_zero_t** = ct::*view_padding_constant*<ct::*view_padding*::*zero*>;
using **view_padding_negative_zero_t** = ct::*view_padding_constant*<ct::*view_padding*::*negative_zero*>;
using **view_padding_positive_inf_t** = ct::*view_padding_constant*<ct::*view_padding*::*positive_inf*>;
using **view_padding_negative_inf_t** = ct::*view_padding_constant*<ct::*view_padding*::*negative_inf*>;
using **view_padding_nan_t** = ct::*view_padding_constant*<ct::*view_padding*::*nan*>;
using **default_view_padding_t** = ct::*view_padding_constant*<ct::*default_view_padding*()>;

Convenience aliases for specifying a view padding constant in view operations.

Example

Example usage of the view padding constant aliases:

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

int x = 0;
ct::tensor_span t{&x, ct::shape{1_ic}};
ct::partition_view p{t, ct::shape{1_ic}};

p.load_masked(ct::view_padding_zero_t{}, 0);
```

2.13.6.4 cuda::tiles::default_view_padding

__tile__ __host__ __device__ inline constexpr ct::*view_padding* **default_view_padding**();

Yields the *default view padding*.

2.13.7. Memory Order

APIs for working with *memory orders*.

2.13.7.1 cuda::tiles::memory_order

enum class **memory_order** : /* unspecified */

enumerator **relaxed** = 0

enumerator **acquire** = 1

enumerator **release** = 2

enumerator **acq_rel** = 3

Enumeration specifying the supported *memory orders*.

2.13.7.2 cuda::tiles::memory_order_constant

template<*memory_order* **Value**>
requires /* atomic constraint */
struct **memory_order_constant**;

A specialization of `ct::memory_order_constant` encodes a *memory order* value in its type. The atomic constraint validates that `Value` is an enumerator of *`ct::memory_order`*.

Member Aliases

using **type** = *memory_order_constant*;

using **value_type** = ct::*memory_order*;

Member Variables

static constexpr ct::*memory_order* **value** = *Value*;

Member Functions

__tile__ __host__ __device__ constexpr **operator ct::*memory_order*()** const noexcept;
Yields *Value*.

__tile__ __host__ __device__ constexpr ct::*memory_order* **operator()**() const noexcept;
Yields *Value*.

2.13.7.3 Constant Aliases

using **memory_order_relaxed_t** = ct::*memory_order_constant*<ct::*memory_order*::*relaxed*>;
using **memory_order_acquire_t** = ct::*memory_order_constant*<ct::*memory_order*::*acquire*>;
using **memory_order_release_t** = ct::*memory_order_constant*<ct::*memory_order*::*release*>;
using **memory_order_acq_rel_t** = ct::*memory_order_constant*<ct::*memory_order*::*acq_rel*>;
Convenience aliases for specifying a memory order in atomic memory APIs.

Example

Example usage of the memory order constant aliases:

```
namespace ct = ::cuda::tiles;
ct::atomic_load(ptr, ct::memory_order_relaxed_t{});
```

2.13.7.4 cuda::tiles::read_memory_order

template<ct::*memory_order* **Order**>
concept **read_memory_order** = /* atomic constraint */;
Indicates whether `Order` is a *read memory order*.

2.13.7.5 cuda::tiles::write_memory_order

template<ct::*memory_order* **Order**>
concept **write_memory_order** = /* atomic constraint */;
Indicates whether `Order` is a *write memory order*.

2.13.8. Thread Scope

APIs for working with *thread scopes*.

2.13.8.1 cuda::tiles::thread_scope

enum class **thread_scope** : /* unspecified */

enumerator **system** = 0
enumerator **device** = 1
enumerator **block** = 2

Enumeration specifying the supported *thread scopes*.

2.13.8.2 cuda::tiles::thread_scope_constant

template<ct::*thread_scope* **Value**>
requires /* atomic constraint */
struct **thread_scope_constant**;

A specialization of `ct::thread_scope_constant` encodes a *thread scope* value in its type. The atomic constraint validates that `Value` is an enumerator of *`ct::thread_scope`*.

Member Aliases

using **type** = *thread_scope_constant*;
using **value_type** = ct::*thread_scope*;

Member Variables

static constexpr ct::*thread_scope* **value** = *Value*;

Member Functions

__tile__ __host__ __device__ constexpr **operator ct::*thread_scope***() const noexcept;
Yields *`Value`*.

__tile__ __host__ __device__ constexpr ct::*thread_scope* **operator()**() const noexcept;
Yields *`Value`*.

2.13.8.3 Constant Aliases

using **thread_scope_system_t** = ct::*thread_scope_constant*<ct::*thread_scope*::*system*>;
using **thread_scope_device_t** = ct::*thread_scope_constant*<ct::*thread_scope*::*device*>;
using **thread_scope_block_t** = ct::*thread_scope_constant*<ct::*thread_scope*::*block*>;
using **default_thread_scope_t** = ct::*thread_scope_constant*<ct::*default_thread_scope*()>;

Convenience aliases for specifying a thread scope in atomic memory APIs.

Example

Example usage of the thread scope constant aliases:

```
namespace ct = ::cuda::tiles;
ct::atomic_load(ptr,
                ct::memory_order_acquire_t{},
                ct::thread_scope_device_t{});
```

2.13.8.4 cuda::tiles::default_thread_scope

__tile__ __host__ __device__ constexpr ct::*thread_scope* **default_thread_scope**() noexcept;

Yields the *default thread scope* for atomic memory APIs.

2.14. Integer Range

ct::irange is a forward range representing an increasing sequence of integers separated by a step. The range is constructed with a lower bound L, exclusive upper bound U, and a step $S > 0$. The range sweeps through the following values in increasing order:

$$\{L + i \cdot S \quad | \quad i \in \mathbb{Z} \quad i \geq 0 \quad L + i \cdot S < U\}$$

A *ct::irange* provides the compiler with structured information about the loop iteration bounds which may be used to better optimize the code.

Example

The following code prints the values from the sequence $(5, 7, 9, 11)$. The lower bound is 5, upper bound is 12 and step is 2.

```
namespace ct = ::cuda::tiles;
for (auto idx : ct::irange(5, 12, 2)) {
  printf("%i\n", idx);
}
```

2.14.1. irange-sentinel

template<ct::*integral* **T**>
struct **irange-sentinel**

Exposition only sentinel type corresponding to *ct::irange-iterator*. It models the end position of a *ct::irange*.

The program is ill-formed if T is cv-qualified.

Exposition Only Members

T **__end**
Exposition only member variable modeling the upper bound of a *ct::irange*. This member exists for the purposes of describing the implicitly defined special member functions.

Default Construction

constexpr **irange-sentinel**( ) noexcept = default
Constructs an instance of *ct::irange-sentinel* in a valid but unspecified state. The behavior is undefined if any operation other than destruction or copy/move assignment is performed on such an object.

Construction From End

__tile__ __host__ __device__ constexpr **irange-sentinel**(*T* end) noexcept
Direct-list-initializes *__end* from *end*.

2.14.2. cuda::tiles::irange_iterator

template<ct::*integral* **T**>
struct **irange_iterator**
Forward iterator of a *ct::irange*. The integer results and step computation is performed in the integral type *T*.

The program is ill-formed if *T* is const qualified.

Exposition Only Members

The following members exist for the purposes of describing the implicitly defined special member functions.

T **__current**
Exposition only member variable modeling the current value of the iterator yielded by a dereference.

T **__step**
Exposition only member modeling the value to add to this iterator during an increment.

Member Aliases

using **difference_type** = int64_t
using **value_type** = *T*

Default Construction

constexpr **irange_iterator**() noexcept = default
Constructs this object in a valid but unspecified state. The behavior is undefined if any operation other than destruction or copy/move assignment is performed on such an object.

Construction From Current and Step

__tile__ __host__ __device__ constexpr **irange_iterator**(*T* current, *T* step) noexcept
Direct-list-initializes *__current* from *current* and *__step* from *step*.

Dereference

__tile__ __host__ __device__ constexpr *T* **operator***() const noexcept
Yields *__current*.

Pre-Increment

__host__ __device__ __tile__ constexpr *irange_iterator* &**operator++**() noexcept
Advances *__current* by *__step* as if by *__current* += *__step*. Yields reference to *this.

Post-Increment

__host__ __device__ __tile__ constexpr *irange_iterator* **operator++**(int) noexcept
Advances *__current* by *__step* as if by *__current* += *__step*. Yields a copy of the iterator prior to the increment operation.

Iterator Comparison

__host__ __device__ __tile__ constexpr bool **operator==**(*irange_iterator* const &other) const noexcept
Compares the *__current* value of *this and *other* for equality. The behavior is undefined if *other* was derived from a different *ct::irange* instance than *this.

Note: operator!= is available by rewritten operator candidates.

Sentinel Comparison

__host__ __device__ __tile__ constexpr bool **operator==**(*irange-sentinel*<*T*> const &other) const noexcept

Returns `true` if `__current` is not less than `other.__end`.

Note: `operator!=` is available by rewritten operator candidates.

2.14.3. cuda::tiles::irange

template<ct::*integral* **T**>
struct **irange**

Forward range modeling an increasing sequence of integers separated by a step. For an example, see the *introduction*.

The program is ill-formed if *T* is const qualified.

Exposition Only Members

The following exposition only members exist for the purposes of describing the implicitly defined special member functions.

T **__lb**

Exposition only member variable modeling the inclusive lower bound of the range.

T **__ub**

Exposition only member variable modeling the exclusive upper bound of the range.

T **__step**

Exposition only member variable modeling the step size of the range.

Aliases

using **iterator** = *irange_iterator*<*T*>

using **const_iterator** = *irange_iterator*<*T*>

Construction From Bounds and Step

__host__ __device__ __tile__ constexpr **irange**(*T* lb, *T* ub, *T* step = 1) noexcept

Direct-list-initializes `__ub` from `ub`, `__lb` from `lb` and `__step` from `step`. Let N denote the largest value representable by *T*. The behavior is undefined if any of the following conditions hold after the initialization of members:

1. `__step <= 0`
2. `__ub > N - __step + 1`.

Example

The following iteration sweeps through the sequence $(-10, -9, -8, -7)$. The step is not explicitly specified and defaults to 1.

```
namespace ct = ::cuda::tiles;
for (auto idx : ct::irange(-10, -6)) {
  printf("%i\n", idx);
}
```

begin

__host__ __device__ __tile__ constexpr *iterator* **begin**() const noexcept

Yields *iterator*{*__lb*, *__step*}.

end

__host__ __device__ __tile__ constexpr *irange-sentinel*<*T*> **end**() const noexcept

Yields *irange-sentinel*<*T*>{*__ub*}.

empty

__host__ __device__ __tile__ constexpr bool **empty**() const noexcept

Determines if iteration over this range would yield zero elements.

Example

The following code yields `true` in both calls `empty()` because the lower bound is greater than or equal to the upper bound.

```
namespace ct = ::cuda::tiles;
ct::irange x(20, 10);
auto r0 = x.empty();

ct::irange y(20, 20);
auto r1 = y.empty();
```

size

__host__ __device__ __tile__ constexpr *iterator*::difference_type **size**() const noexcept

Determines the number of elements this range would produce during iteration. This value direct-initializes the return type, possibly triggering overflow.

lower_bound

__host__ __device__ __tile__ constexpr *T* **lower_bound**( ) const noexcept
Yields *__lb*.

upper_bound

__host__ __device__ __tile__ constexpr *T* **upper_bound**( ) const noexcept
Yields *__ub*.

step

__host__ __device__ __tile__ constexpr *T* **step**( ) const noexcept
Yields *__step*.

Deduction Guides

template<typename **U1**, typename **U2**>
irange(*U1*, *U2*) -> ct::*irange*<ct::*arithmetic_tile_conversion_t*<*U1*, *U2*>>;

template<typename **U1**, typename **U2**, typename **U3**>
irange(*U1*, *U2*, *U3*) ->
ct::*irange*<ct::*arithmetic_tile_conversion_t*<ct::*arithmetic_tile_conversion_t*<*U1*, *U2*>, *U3*>>;

Deduction guides enabling CTAD from integer arguments.

2.15. Assumptions

An assumption introduces constraints on program values that the compiler may use for optimization purposes. If a value violates an assumption, the behavior is undefined. Assumptions are not checked at runtime.

The assumption functions return the same value as their argument. For the compiler to effectively leverage an assumption, the return value of the assumption API should be used in subsequent operations.

Example

The following code tells the compiler to assume that the value of y is greater than or equal to 0. This allows the compiler to execute the first `printf` without performing any runtime check on the value of y.

The second `if` condition checks the value of x. Since x was not derived from the return value of an assumption, the compiler might not eliminate the runtime check of x.

```
__tile__ void foo(int x) {
  namespace ct = ::cuda::tiles;
  using namespace ct::literals;

  int y = ct::assume_bounded_below(x, 0_ic);

  if (y >= 0) { // conditional check eliminated by compiler
    printf("y is non-negative!\n");
  }

  if (x >= 0) { // conditional checked at runtime
    printf("x is non-negative!\n");
  }
}
```

Warning: The only effect of an assumption is to introduce undefined behavior. If the assumption is violated at runtime, the program's behavior is undefined.

2.15.1. cuda::tiles::assume_blocked

template<ct::*shape_like* **S**, ct::*tile_like* **T**>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_blocked**(*T* a, *S* = {}) noexcept;

Indicates that *tile like* argument a can be represented by partitions of shape S where all elements within a given partition have the same value.

The atomic constraint validates that:

1. S has no zero length extents.
2. The *rank* of T matches the *rank* of S.
3. T is a *pointer tile* or *integral tile*.

Let N denote the *rank* of T. A partition P_J at indices $J = (j_0, j_1, \ldots, j_{N-1})$ is defined as the set of values

$$P_J = \{a(i_0, i_1, \ldots, i_{N-1}) \quad | \quad j_k \cdot S_k \leq i_k < \min((j_k + 1) \cdot S_k, T_k) \quad 0 \leq k < N\}$$

This operation introduces undefined behavior if the set P_J has more than one distinct element for any combination of non-negative indices $J = (j_0, j_1, \ldots, j_{N-1})$.

Example

The following code specifies partitions of shape 3×2 must all have the same elements.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_blocked(x, ct::shape{3_ic, 2_ic});
```

The following 8×8 tile satisfies this assumption:

$$\begin{pmatrix} 42 & 42 & 5 & 5 & 1 & 1 & -2 & -2 \\ 42 & 42 & 5 & 5 & 1 & 1 & -2 & -2 \\ 42 & 42 & 5 & 5 & 1 & 1 & -2 & -2 \\ 3 & 3 & 2 & 2 & 4 & 4 & -5 & -5 \\ 3 & 3 & 2 & 2 & 4 & 4 & -5 & -5 \\ 3 & 3 & 2 & 2 & 4 & 4 & -5 & -5 \\ 7 & 7 & 8 & 8 & 3 & 3 & -6 & -6 \\ 7 & 7 & 8 & 8 & 3 & 3 & -6 & -6 \end{pmatrix}$$

2.15.2. cuda::tiles::assume_bounded

template<ct::*integral* auto **LB**, ct::*integral* auto **UB**, ct::*integral_tile* **T**>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_bounded**(*T* a, ct::*integral_constant*<*LB*> = {},
ct::*integral_constant*<*UB*> = {}) noexcept;

Indicates that the elements of a inhabit the inclusive range $[LB, UB]$.

Let E be the *element type* of T, let L be the least value representable in E, and let U be the greatest value representable in *make-signed-t*<E>. The atomic constraint validates that:

1. E is not `bool`
2. $L \leq LB \leq UB \leq U$

This operation introduces undefined behavior if any element e of a satisfies $e < LB$ or $e > UB$.

Example

The following example assumes that every element of x is bounded between -10 and 100 inclusive.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_bounded(x, -10_ic, 100_ic);
```

2.15.3. cuda::tiles::assume_bounded_above

template<ct::*integral* auto **UB**, ct::*integral_tile* **T**>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_bounded_above**(*T* a, ct::*integral_constant*<*UB*> = {}) noexcept;

Indicates that the elements of a are less than or equal to the upper bound UB.

Let E be the *element type* of a, let L be the least value representable in E and let U be the greatest value representable by *make-signed-t*<E>. The atomic constraint validates that:

1. E is not `bool`
2. $L \leq UB \leq U$

This operation introduces undefined behavior if any element e of a satisfies $e > UB$.

Example

The following example assumes that every element of x is less than or equal to 100.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_bounded_above(x, 100_ic);
```

2.15.4. cuda::tiles::assume_bounded_below

template<ct::*integral* auto **LB**, ct::*integral_tile* **T**>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_bounded_below**(*T* a, ct::*integral_constant*<*LB*> = {}) noexcept;

Indicates that the elements of a are greater than or equal to the lower bound LB.

Let E be the *element type* of a, let L be the least value representable by E and U be the greatest value representable by E. The atomic constraint validates that:

1. E is not `bool`
2. E is not an unsigned integral type
3. $L \leq LB \leq U$

This operation introduces undefined behavior if any element e of a satisfies $e < LB$.

Example

The following example assumes that every element of x is greater than or equal to -10.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_bounded_below(x, -10_ic);
```

2.15.5. cuda::tiles::assume_divisible

template<ct::*integral* auto **Div**, ct::*integral_tile* **T**>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_divisible**(*T* a, ct::*integral_constant*<*Div*> = {}) noexcept;

Indicates that the elements of a are divisible by Div.

The atomic constraint validates that Div is a power of two.

This operation introduces undefined behavior if any element of a is not divisible by Div.

Example

The following example assumes that every element of x is divisible by 16.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_divisible(x, 16_ic);
```

2.15.6. cuda::tiles::assume_divisible_strided

template<
ct::*integral* auto **Div**,
ct::*integral* auto **Stride**,
ct::*integral* auto **D**,
ct::*integral_tile* **T**
>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_divisible_strided**(*T* a, ct::*integral_constant*<*Div*> = {},
ct::*integral_constant*<*Stride*> = {},
ct::*integral_constant*<*D*> = {}) noexcept;

Indicates that every Stride elements along D is divisible by Div. Between the divisible elements, the values increase sequentially along D.

The atomic constraint validates that:

1. Div is a power of two
2. Stride > 0
3. $0 \leq D < N$ where N is the *rank* of *T*
4. The *element type* of T is signed

A strided partition at index $J = (j_0, j_1, \ldots, j_{N-1})$ is defined as a sequence

$$P_k = a(j_0, ..., j_D * \text{Stride} + k, \ldots, j_{N-1}) \quad 0 \leq k < \min(\text{Stride}, T_D - j_D \cdot \text{Stride})$$

This operation introduces undefined behavior if any such partition is not of the format

$$(n, n+1, n+2, \ldots, n+m)$$

where n is divisible by Div.

Example

The following example assumes that the start of each partition is divisible by 16. The partitions have stride 3 along dimension 1.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_divisible_strided(x, 16_ic, 3_ic, 1_ic);
```

The following matrix satisfies these assumptions:

$$\begin{pmatrix} 32 & 33 & 34 & 0 & 1 & 2 & 16 & 17 \\ 64 & 65 & 66 & -16 & -15 & -14 & 0 & 1 \end{pmatrix}$$

2.15.7. cuda::tiles::assume_aligned

template<ct::*integral* auto **Align**, ct::*pointer_tile* **T**>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_aligned**(*T* a, ct::*integral_constant*<*Align*> = {}) noexcept;

Indicates that the elements of a have alignment Align.

The atomic constraint validates that Align is a power of two.

This operation introduces undefined behavior if any element of a is not aligned to Align.

Example

The following example assumes that every element of x is 16 byte aligned.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_aligned(x, 16_ic);
```

2.15.8. cuda::tiles::assume_aligned_strided

template<
ct::*integral* auto **Align**,
ct::*integral* auto **Stride**,
ct::*integral* auto **D**,
ct::*pointer_tile* **T**
>
requires /* atomic constraint */
[[nodiscard]] __tile__ *T* **assume_aligned_strided**(*T* a, ct::*integral_constant*<*Align*> = {},
ct::*integral_constant*<*Stride*> = {},
ct::*integral_constant*<*D*> = {}) noexcept;

Indicates that every Stride elements along D is a pointer aligned to Align. Between the aligned elements, the pointer values increase sequentially according to the object size of the pointee type along D.

The atomic constraint validates that:

1. Div is a power of two
2. Stride > 0
3. $0 \leq D < N$ where N is the *rank* of *T*

A strided partition at index $J = (j_0, j_1, \ldots, j_{N-1})$ is defined as a sequence

$$P_k = a(j_0, ..., j_D * \text{Stride} + k, \ldots, j_{N-1}) \quad 0 \leq k < \min(\text{Stride}, T_D - j_D \cdot \text{Stride})$$

This operation introduces undefined behavior if any such partition is not of the format

$$(p, p+1, p+2, \ldots, p+m)$$

where p is a pointer value aligned to Align.

Example

The following example assumes that the partitions of x begin with pointers aligned to 8. The partitions have a stride of 3 along dimension 1.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

auto y = ct::assume_aligned_strided(x, 8_ic, 3_ic, 1_ic);
```

The following matrix satisfies the assumption for a divisibility of 8, a stride of 3, and a dimension of 1 given some pointer p aligned to 8.

$$\begin{pmatrix} p & p+1 & p+2 & p+8 & p+9 & p+10 & p+16 & p+17 \\ p+64 & p+65 & p+66 & p-16 & p-15 & p-14 & p & p+1 \end{pmatrix}$$

2.16. Optimization Hints

A *tile optimization hint* is metadata attached to a C++ source construct such as an entity, statement, or expression which is used to guide compiler optimizations related to that construct. Hints do not affect the semantics of the program and may be ignored by the compiler.

Example

The first hint below tells the compiler to prefer 4 thread blocks for each thread block cluster when launching the kernel. The second hint tells the compiler to avoid using TMA when loading data from x.

Both hints might be ignored by the compiler.

```
[[ cutile::hint(0, num_cta_per_cga=4) ]]
__tile_global__ void kernel(int* x) {
  namespace ct = ::cuda::tiles;
  using namespace ct::literals;
  auto S = ct::tensor_span{x, ct::shape{512_ic, 128}};
  auto P = ct::partition_view{S, ct::shape{32_ic, 64_ic}};

  ct::tile<int, ct::shape<32, 64>> y;

  [[ cutile::hint(0, allow_tma=false) ]]
  y = P.load(0, 1);
}
```

2.16.1. Hint Specification

A tile optimization hint contains the following information:

1. A numeric encoding of the *target architecture* for which the hint will be applicable. The architecture's numeric encoding is the same as that used by the `__CUDA_ARCH__` macro. When compiling for a specific target, only hints matching that target will be considered. The special value of `0` indicates a hint that applies to all architectures. Such hints are called "architecture agnostic".
2. A *hint kind* which indicates the type of information that the hint specifies. The supported hint kinds are specified below.
3. A *hint value* whose interpretation depends on the hint kind.

A tile optimization hint may *appertain* to one or more C++ constructs. When a hint appertains to a construct, the implementation may use the hint to make optimization decisions related to that construct.

Tile optimization hints are specified using the C++ attribute `cutile::hint` in the `cutile` attribute namespace. The *attribute-argument-clause*[1] of a `cutile::hint` attribute shall satisfy the following grammar production rules in addition to those production rules required by the C++ standard[2]:

```
attribute-argument-clause  ::=  "(" constant-expression "," tile-hint-kv-pair-list ")"
tile-hint-kv-pair-list     ::=  tile-hint-kv-pair
                                tile-hint-kv-pair "," tile-hint-kv-pair-list
tile-hint-kv-pair          ::=  tile-hint-kind "=" constant-expression
tile-hint-kind             ::=  identifier | keyword
```

Note: The grammar is unambiguous because *constant-expression* cannot produce a top-level comma expression.

[1] See § 9.12.1 [dcl.attr.grammar] of ISO/IEC 14882:2024

[2] See § 5.10 [lex.name], § 5.11 [lex.key], and § 7.7 [expr.const] of ISO/IEC 14882:2024 for definitions of the grammar symbols `identifier`, `keyword` and `constant-expression`.

```
// Syntax error, '3' is not a tile-hint-kv-pair
[[ cutile::hint(0, latency=4, 3) ]]

// OK '(4, 3)' is an integral constant expression yielding '3'.
[[ cutile::hint(0, latency=(4, 3) )]]
```

The first argument of the *attribute-argument-clause* shall be an integral constant expression whose value is the attribute's *target architecture*. For each *tile-hint-kv-pair* the *constant-expression* shall be an integral constant expression designating the key-value pair's value.

The target architecture and the value of each key-value pair may be type or value dependent. A `cutile::hint` attribute may participate in a pack expansion in which case a separate instance of the attribute is created for each element of the pack.

Example

Zero or more `cutile::hint` attributes may be generated for the function entity yielded by a given template instantiation of `kernel`:

```
template<size_t> struct ints { };

template<size_t... Archs, size_t... Xs>
[[ cutile::hint(Archs, occupancy = Xs) ... ]]
__tile_global__ void kernel(ints<Archs...>, ints<Xs...>) { }
```

Each *tile-hint-kv-pair* of a `cutile::hint` attribute generates a tile optimization hint if

1. The target architecture of the enclosing attribute is recognized by the implementation. This becomes the generated hint's architecture.
2. The *tile-hint-kind* of the key-value pair is one of the known list of hint kinds specified below. This becomes the generated hint's kind.
3. The key-value pair's value is valid for the associated hint kind and architecture as specified below. This becomes the generated hint's value.

Note: If a *tile-hint-kv-pair* does not generate a tile optimization hint, it is ignored.

A tile optimization *appertains* to a construct if:

1. For function entities, there exists a `cutile::hint` attribute which appertains[Page 176, 1] to the function and generates the optimization hint.
2. For expressions, the expression inhabits an *expression-statement*, and there exists a `cutile::hint` attribute which appertains[Page 176, 1] to the enclosing statement and generates the optimization hint.
3. The tile optimization hint is relevant to this construct according to the rules specific to that hint kind defined below.

Note: If a generated hint does not appertain to any entities, it is ignored.

```
namespace ct = ::cuda::tiles;
```

(continues on next page)

(continued from previous page)

```
// Ignored, not an expression-statement
[[ cutile::hint(0, latency=4) ]]
auto x = ct::load(y);

// Ignored, not an expression-statement
[[ cutile::hint(0, latency=4) ]]
if (auto x = ct::load(y)) { }

// Ignored, not an expression-statement
[[ cutile::hint(0, latency=4) ]]
return ct::load(y);
```

If two or more tile hints with the same kind and architecture appertain to the same construct, only the one generated by the *tile-hint-kv-pair* and attribute that are last in lexical order is considered. All others are ignored.

Example

In the following code, only the `latency=5` hint is applied in all three examples.

```
namespace ct = ::cuda::tiles;

// Example 1
[[ cutile::hint(0, latency=4, latency=5) ]]
x = ct::load(y);

// Example 2
[[ cutile::hint(0, latency=4) ]]
[[ cutile::hint(0, latency=5) ]]
x = ct::load(y);

// Example 3
[[ using cutile :
  hint(0, latency=4),
  hint(0, latency=5)
]]
x = ct::load(y);
```

If an architecture agnostic hint does not have a supported kind or value for the targeted architecture, the hint is ignored for that architecture.

Example

The hint below is ignored when targeting sm_80 or below, but is active for other architectures.

```
[[ cutile::hint(0, num_cta_in_cga=2) ]]
__tile__ void kernel0() { }
```

When both an architecture agnostic and an architecture specific hint with the same kind appertain to the same entity, the architecture specific hint takes precedence when the matching architecture is targeted.

Example

In the code below, the latency is specified to be 4 for all architectures except sm_100 where it is specified as 5.

```
namespace ct = ::cuda::tiles;

[[ using cutile : hint(0, latency=4), hint(1000, latency=5) ]]
ct::load(x);
```

2.16.2. Hint Kinds

2.16.2.1 num_cta_in_cga

The `num_cta_in_cga` hint kind indicates the recommended number of cooperative thread arrays that should be allocated per cooperative group array when launching a kernel. The supported values for this hint are 1, 2, 4, 8, and 16. This hint may appertain only to tile kernel functions.

For a `num_cta_in_cga` hint with architecture sm_80, the only applicable value is 1.

Example

In the following example, the compiler is instructed to prefer 4 thread blocks per cluster for sm_90 targets and 8 thread blocks per cluster for sm_100 targets when launching the kernel.

```
[[ using cutile :
  hint(900, num_cta_in_cga=4),
  hint(1000, num_cta_in_cga=8) ]]
__tile_global__ void kernel1() { }
```

2.16.2.2 occupancy

The `occupancy` hint kind indicates the recommended occupancy when launching a kernel. The supported values for this hint are the inclusive integer range $[1, 32]$. This hint may appertain only to tile kernel functions.

Example

In the following example, the compiler is instructed prefer an occupancy of 15 for all architectures except for sm_90 where it will prefer 22.

```
[[ using cutile :
  hint(0, occupancy=15),
  hint(900, occupancy=22) ]]
__tile_global__ void kernel2() { }
```

2.16.2.3 latency

The `latency` hint kind indicates the relative memory access latency of a memory operation. The supported values for this hint are the inclusive integer range $[1, 10]$. This hint may appertain only to direct call expressions to:

1. *pointer tile loads*
2. *pointer tile stores*
3. *partition view loads*
4. *partition view stores*

Example

In the following example, a latency hint of 7 is specified on all the call expressions.

```
namespace ct = ::cuda::tiles;

// Example 1
[[ cutile::hint(0, latency=7) ]]
z = ct::load(x) + ct::load(y);

// Example 2
[[ cutile::hint(0, latency=7) ]]
ct::store(ptr, z);

using namespace ct::literals;
auto t = ct::tensor_span{ptr, ct::shape{128_ic, 4_ic}};
auto p = ct::partition_view{t, ct::shape{4_ic, 4_ic}};

ct::tile<int, ct::shape<4, 4>> value;

[[ cutile::hint(0, latency=7) ]]
value = p.load(0, 0);

[[ cutile::hint(0, latency=7) ]]
p.store(value, 0, 0);
```

2.16.2.4 allow_tma

The `allow_tma` hint kind instructs the compiler to prefer traditional memory operations over Tensor Memory Accelerators (TMA) when loading or storing to view memory. The supported values for this hint are `true` and `false`. This hint may appertain only to direct call expressions to:

1. *partition view loads*
2. *partition view stores*

Example

In the following example, the hints instruct the compiler to prefer not to use TMA for the partition view load and store.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
auto t = ct::tensor_span{ptr, ct::shape{128_ic, 4_ic}};
auto p = ct::partition_view{t, ct::shape{4_ic, 4_ic}};

ct::tile<int, ct::shape<4, 4>> value;

[[ cutile::hint(0, allow_tma=false) ]]
value = p.load(0, 0);

[[ cutile::hint(0, allow_tma=false) ]]
p.store(value, 0, 0);
```

Chapter 3. Appendices

3.1. Undefined Behavior Annex

This section summarizes the scenarios where undefined behavior may occur in Tile C++ APIs. Please see the API reference for details on the specific behavior of any given operation.

- Undefined behavior may occur due to signed integer overflow in arithmetic addition APIs. This allows the compiler to optimize signed integer operations by assuming the absence of integer overflow. See *addition* for details.

  Example

  No integer promotion occurs as part of the *arithmetic tile conversions*. As a result, the addition below is performed in the type `signed char` and overflow occurs triggering undefined behavior.

  ```
  namespace ct = ::cuda::tiles;
  signed char x = 127;
  signed char y = 1;
  auto z = ct::add(x, y); // UB
  ```

- Undefined behavior may occur due to signed integer overflow in arithmetic subtraction APIs. This allows the compiler to optimize signed integer operations by assuming the absence of integer overflow. See *subtraction* for details.

  Example

  No integer promotion occurs as part of the *arithmetic tile conversions*. As a result, the subtraction below is performed in the type `signed char` and overflow occurs triggering undefined behavior.

  ```
  namespace ct = ::cuda::tiles;
  signed char x = -128;
  signed char y = 1;
  auto z = ct::sub(x, y); // UB
  ```

- Undefined behavior may occur due to signed integer overflow in arithmetic negation APIs. This allows the compiler to optimize signed integer operations by assuming the absence of integer overflow. See *negation* for details.

  Example

No integer promotion occurs in the negation APIs. As a result, the negation below is performed in the type `signed char` and overflow occurs triggering undefined behavior.

```
namespace ct = ::cuda::tiles;
using i8 = ct::tile<signed char, ct::shape<>>;
auto x = ct::full<i8>(-128);
auto y = -x; // UB
```

- Undefined behavior may occur due to signed integer overflow in multiplication APIs. This allows the compiler to optimize signed integer operations by assuming the absence of integer overflow. See *multiplication* for details.

Example

No integer promotion occurs as part of the *arithmetic tile conversions*. As a result, the multiplication below is performed in the type `signed char` and overflow occurs triggering undefined behavior.

```
namespace ct = ::cuda::tiles;
signed char x = -128;
signed char y = -1;
auto z = ct::mul(x, y); // UB
```

- Undefined behavior may occur due to signed integer overflow in division, ceil division, floor division, and remainder APIs. This allows the compiler to optimize signed integer operations by assuming the absence of integer overflow. See *division*, *ceiling division*, *floor division*, and *remainder* for details.

Example

No integer promotion occurs as part of the *arithmetic tile conversions*. As a result, the division operations below are performed in the type `signed char` and overflow occurs triggering undefined behavior.

```
namespace ct = ::cuda::tiles;
signed char x = -128;
signed char y = -1;
auto z0 = ct::div(x, y); // UB
auto z1 = ct::ceildiv(x, y); // UB
auto z2 = ct::floordiv(x, y); // UB
auto z3 = ct::remainder(x, y); // UB
```

- Undefined behavior may occur in integer division, ceil division, floor division, and remainder APIs when the divisor is zero. See *division*, *ceiling division*, *floor division*, and *remainder* for details.

Example

```
namespace ct = ::cuda::tiles;
int x = 10;
int y = 0;
auto z0 = ct::div(x, y); // UB
auto z1 = ct::ceildiv(x, y); // UB
```

(continues on next page)

(continued from previous page)

```
auto z2 = ct::floordiv(x, y); // UB
auto z3 = ct::remainder(x, y); // UB
```

- Undefined behavior may occur in left or right bitshift operations when the right hand side operand is negative. See *left bitshift* and *right bitshift* for details.

Example

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;
i32x4 x = ct::iota<i32x4>();
i32x4 y0 = x << -2; // UB
i32x4 y1 = x >> -2; // UB
```

- Undefined behavior may occur in left or right bitshift operations when the right hand side operand is greater than the bitwidth of the left hand side type. See *left bitshift* and *right bitshift* for details.

Example

```
namespace ct = ::cuda::tiles;
using i32x4 = ct::tile<int, ct::shape<4>>;
i32x4 x = ct::iota<i32x4>();
i32x4 y0 = x << 32; // UB
i32x4 y1 = x >> 32; // UB
```

- Undefined behavior may occur when computing the absolute value of a minimal integer value. See *ct::abs()* for details.

Example

```
namespace ct = ::cuda::tiles;
signed char x = -128;
auto y = ct::abs(x); // UB
```

- Undefined behavior may occur in pointer addition and subtraction operations if the resulting pointer does not refer to a valid object in the C++ object model or if other conditions of the builtin pointer arithmetic operations are not satisfied. See *pointer addition* and *pointer subtraction* for details.

Example

In the following example, the pointer values do not refer to objects in the original C++ array referenced by p. The behavior is undefined even though the pointer values are never dereferenced.

```
namespace ct = ::cuda::tiles;
int x[10];
auto p = ct::full<ct::tile<int*, ct::shape<4>>>(&x[0]);
auto q0 = p + 100; // UB
auto q1 = p - 100; // UB
```

- Undefined behavior may occur in pointer difference operations if the arguments do not refer to the same array object or if other conditions of the builtin pointer difference operation are not satisfied. See *pointer difference* for details.

Example

In the following example, the two pointer arguments refer to different array objects yielding undefined behavior.

```
namespace ct = ::cuda::tiles;
struct Obj { int x[4]; int y[4]; };
Obj obj;
auto p0 = ct::full<ct::tile<int*, ct::shape<4>>>(&obj.x[0]);
auto p1 = ct::full<ct::tile<int*, ct::shape<4>>>(&obj.y[0]);
auto q = p1 - p0; // UB
```

- Undefined behavior may occur when retrieving the size of a *shape like* type whose total size cannot be represented in `size_t`. For details see *`ct::shape_size_v`*.

Example

```
namespace ct = ::cuda::tiles;
using T = ct::shape<65536, 65536, 65536, 65536>;
auto s = ct::shape_size_v<T>; // UB
```

- Undefined behavior may occur when extracting an out of bounds partition from a tile. For details, see *`ct::extract()`*.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
auto x = ct::iota<ct::tile<int, ct::shape<4, 4>>>();
auto r = ct::extract(x, ct::shape{2_ic, 2_ic}, 0, 2); // UB
```

- Undefined behavior may occur when bitcasting the elements of a tile if the resulting value representation does not correspond to a value of the destination type. For details, see *`ct::element_bitcast()`*.

Example

```
namespace ct = ::cuda::tiles;
unsigned char x = 2;
auto r0 = ct::element_bitcast<bool>(x); // UB
```

- Undefined behavior may occur during a *reduction* or *scan* if there exists a possible grouping or order of elements that could trigger undefined behavior in the binary operation.

Example

The *ct::prod()* function is UB because the grouping $((127 \text{ op } 2) \text{ op } 0) \text{ op } 1$ triggers signed integer overflow.

The *ct::partial_prod()* invocation is undefined behavior because the grouping $(127 \text{ op } 2) \text{ op } 0$ for computing the third element of the scan result triggers undefined behavior.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
signed char xData[4] = {127, 0, 2, 1};
auto x = ct::load(&xData[0] + ct::iota<ct::tile<int, ct::shape<4>>>());
auto r0 = ct::prod(x, 0_ic); // UB
auto r1 = ct::partial_prod(x, 0_ic); // UB
```

- Undefined behavior may occur due to an assumption API that is violated at runtime. For details, see *Assumptions*.

Example

In the following code, the variable x violates the bounds assumption.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;

int x = 50;
int y = ct::assume_bounded_above(x, 10_ic); // UB
```

- Undefined behavior may occur when accessing the elements of a default constructed *ct::tile* object. For details, see *ct::tile::tile()*.

Example

Undefined behavior occurs at the addition operation because x is default constructed.

```
namespace ct = ::cuda::tiles;
ct::tile<int, ct::shape<4>> x;

auto y = x + 3; // UB
```

- Undefined behavior may occur when accessing a non-existent extent value via the *ct::extents::static_extent()* or *ct::extents::extent()* API.

Example

```
namespace ct = ::cuda::tiles;
using Exts = ct::extents<uint32_t, 4, ct::dynamic_extent>;

Exts::static_extent(2); // UB

Exts e{8};
e.extent(2); // UB
```

- Undefined behavior may occur when accessing a static extent value via the *ct::extents::extent()* function if the value is not representable in the index type.

Example

```
namespace ct = ::cuda::tiles;
ct::extents<uint16_t, 65536> e;
e.extent(0); // UB
```

- Undefined behavior may occur when accessing a non-existent dimension value via the *ct::dimension_map::mapping()* API.

Example

```
namespace ct = ::cuda::tiles;
using Map = ct::dimension_map<1, 0>;
Map::mapping(2); // UB
```

- Undefined behavior may occur when accessing a non-existent stride value via the `static_stride` or `stride` member functions of *layout mappings*.

Example

```
namespace ct = ::cuda::tiles;
using L = ct::layout_right_mapping<ct::extents<uint32_t, 4, 8>>;
L::static_stride(2); // UB
L{{}}.stride(2);       // UB
```

- Undefined behavior may occur if overflow happens while accessing the `static_stride` or `stride` member functions of *layout mappings*.

Example

```
namespace ct = ::cuda::tiles;
using L = ct::layout_right_mapping<ct::extents<uint32_t, 65536, 65536, 65536,
↪65536, 65536>>;
L::static_stride(0); // UB
L{{}}.stride(0);       // UB
```

- Undefined behavior may occur when loading or storing an out-of-bounds partition from a *ct::partition_view*. For details see *partition view loads* and *stores*.

Example

In this example, the behavior is undefined even though the load and store are both masked because the partition index is fully out of bounds.

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
ct::tensor_span t{ptr, ct::extents{16, 32}};
ct::partition_view p{t, ct::shape{4_ic, 2_ic}};
```

(continues on next page)

(continued from previous page)

```
p.load_masked(4, 0); // UB
p.store_masked(x, 0, 16); // UB
```

- Undefined behavior may occur when loading or storing a partially out-of-bounds partition from a *ct::partition_view* if the load or store is not masked.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
ct::tensor_span t{ptr, ct::extents{5, 6}};
ct::partition_view p{t, ct::shape{4_ic, 4_ic}};

p.load(0, 2); // UB
p.store(x, 2, 0); // UB
```

- Undefined behavior may occur when loading or storing a partition view whose underlying tensor span is not injective.

Example

```
namespace ct = ::cuda::tiles;
using namespace ct::literals;
ct::layout_strided_mapping m{ct::extents{8, 16}, ct::extents{8, 1}};

ct::tensor_span t{ptr, m};
ct::partition_view p{t, ct::shape{4_ic, 4_ic}};

p.load(0, 0); // UB
p.store(x, 0, 0); // UB
```

- Undefined behavior may occur when providing a negative step value during *ct::irange* construction.

Example

```
namespace ct = ::cuda::tiles;
ct::irange x(20, 10, -1); // UB
```

- Undefined behavior may occur when constructing an *ct::irange* whose upper bound is close to the max integer value representable in the range's value type. For details, see the constructor *ct::irange::irange()*.

Example

```
namespace ct = ::cuda::tiles;
ct::irange<unsigned char> x(0, 251, 10); // UB
```

- Undefined behavior may occur during a *scalar conversion* from a *floating point scalar* to an *integer scalar* if the source value is not (after truncation of its fractional part) within the representable range of the target. This behavior is described in 7.3.1 [conv.fpint] of ISO/IEC 14882:2024.

  Note that this kind of *scalar conversion* can occur in several APIs including the `ct::tile::tile()`, `ct::element_cast()`, and the *tile to scalar conversion operator*.

  Conversions to `bool` are handled by 7.3.15 [conv.bool] of ISO/IEC 14882:2024 and are not subject to this undefined behavior.

  Example

  ```
  namespace ct = ::cuda::tiles;
  unsigned char x{ct::tile<float, ct::shape<>>{256.0f}}; // UB
  ct::tile<unsigned char, ct::shape<>> y{256.0f}; // UB
  unsigned char z = ct::element_cast<unsigned char>(256.0f); // UB
  ```

- Undefined behavior may occur due to a *data race* when a single API generates multiple weak *memory operations* on the same memory location.

  Example

  ```
  namespace ct = ::cuda::tiles;
  using i32x4 = ct::tile<int, ct::shape<4>>;
  using ptr64x4 = ct::tile<int*, ct::shape<4>>;

  int x = 0;

  auto ptrs = ct::full<ptr64x4>(&x);
  ct::store(ptrs, ct::iota<i32x4>()); // UB
  ```

- Undefined behavior may occur due to signed integer overflow in the `ct::atomic_add()` API.

  Example

  ```
  namespace ct = ::cuda::tiles;
  int x = 0x7FFFFFFF;
  int y = 1;
  auto z = ct::atomic_add(&x, y, ct::memory_order_relaxed_t{}); // UB
  ```

- Undefined behavior may occur due to signed integer overflow in the `ct::atomic_sub()` API, including due to the internal unary negation performed on the right hand side operand.

  Example

  ```
  namespace ct = ::cuda::tiles;
  int x = -1;
  int y = 1 << 31;
  auto z = ct::atomic_sub(&x, y, ct::memory_order_relaxed_t{}); // UB
  ```

3.2. Mathematical Conventions

When using mathematical notation in this reference document, the following conventions are observed:

- A sequence $s = (x_0, x_1, \ldots, x_k)$ uses zero-based indexing: $s(i) = x_i$.
- An empty product (a product over a sequence of length zero) is understood to have value 1.
- An empty sum (a sum over a sequence of length zero) is understood to have value 0.
- A computation described using mathematical notation is understood to occur in infinite precision and is not an evaluation in the C++ abstract machine. When discussing C++ expressions that are evaluated with C++ semantics, this is specifically noted and the expression is typeset as inline code.
- When an *integral scalar* object is typeset in mathematical notation, the value corresponds to the *numeric value* of the object.

Chapter 4. Release Notes

4.1. CUDA 13.3

- Initial release

genindex

Copyright

Index

D

E

F

I

L

M

N

P

R

S

T

Z