Matrix Tile Extension
Portable ISA for Vector-Integrated Matrix Units

Erich Focht (NEC), Marc Casas-Guix (BSC)

Scope and Principles

Provide a simple and portable way to use tensor units linked to vector units for performing the operations **C = A·B** and **C = C+A·B** where **A**, **B**, **C** are dense matrices of the dimensions, respectively, $m \times k$, $k \times n$, $m \times n$.

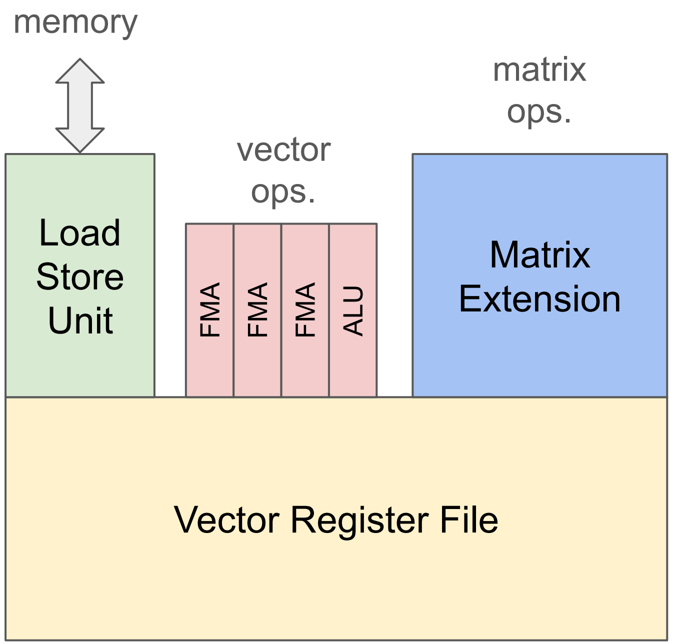
Portability

Few Commands

Simple Usage

The names of the instructions have been chosen to describe their function and are by no means “final”.

Accommodate Diversity



Short Vectors (e.g. VLEN=256)

- fp64: **2x2** or **4x1** matrix per VREG
- fp16: **4x4** or **8x2** matrix per VREG
- VRF: 4-8 read ports (?)
- 4 lanes (or less)

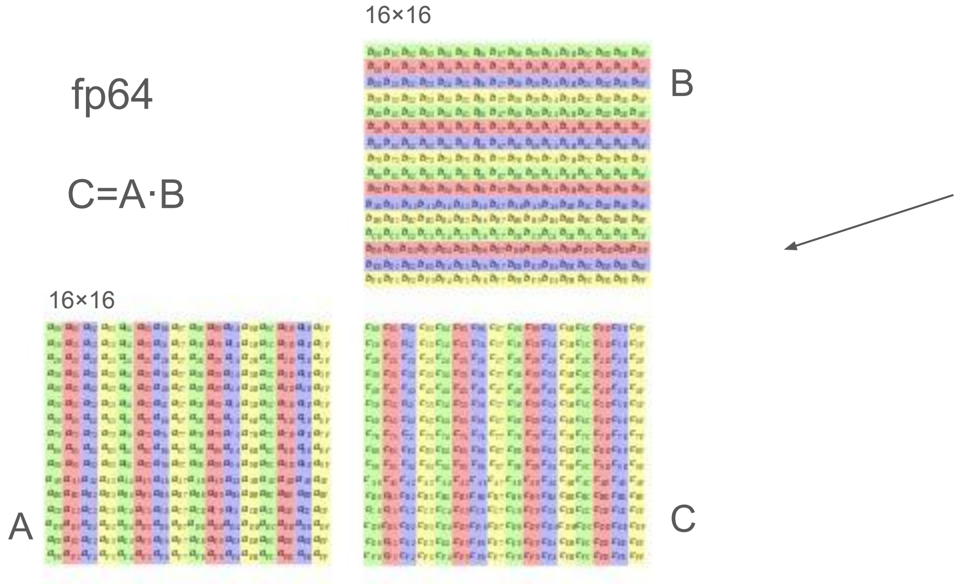
Long Vectors (e.g. VLEN=16384)

- fp64: **16x16, 32x8, 64x4, 128x2, 256x1** matrix
- fp16: **32x32, 64x16, 128x8, 256x4, 512x2, 1024x1** matrix
- VRF: 5-16 read ports per lane, eg. 512 on SX-Aurora VE3
- 8 to 32 lanes
- many possible matrix multiplier geometries!

Square Tiles

fp64

C=A·B



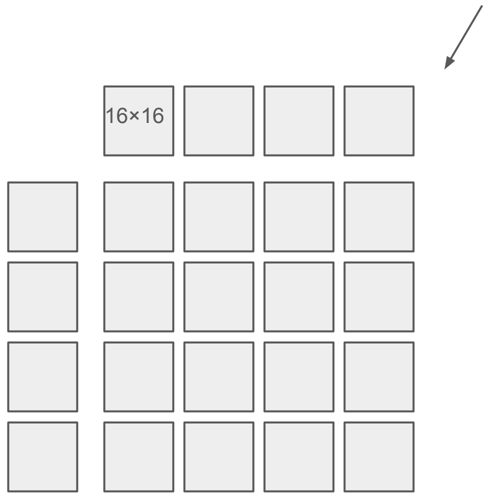
Arithmetic Intensity:

Load: 2·256 = 512

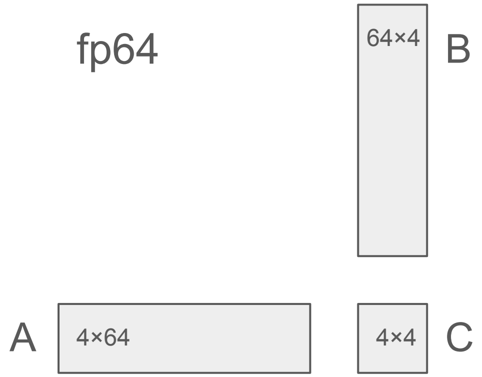
Matmult: 2·16·16·16 = 8192 ops

Arithmetic intensity = 8192 / 512 = 16

With 4x4 accumulators: a.i. = 64



Narrow Tiles



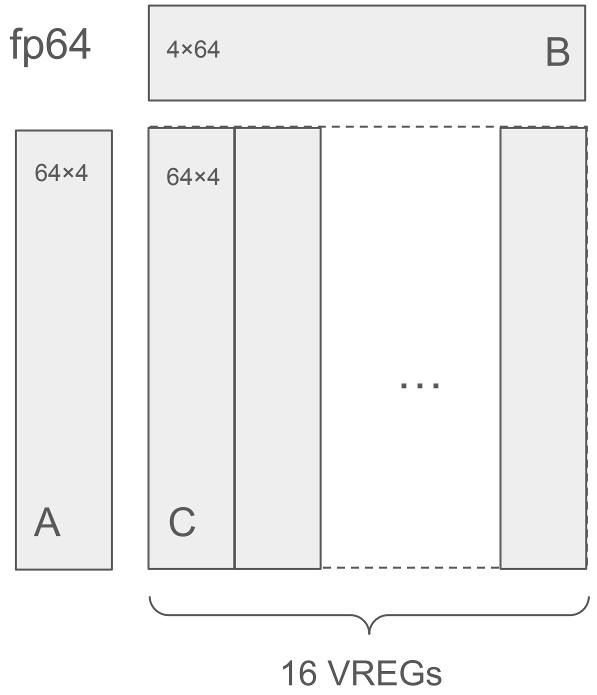
Arithmetic Intensity:

Load: 2·4·64 = 512

Matmult: 2·4·4·64 = 2048 ops

Arithmetic intensity = 2048 / 512 = 4

Narrow Tiles



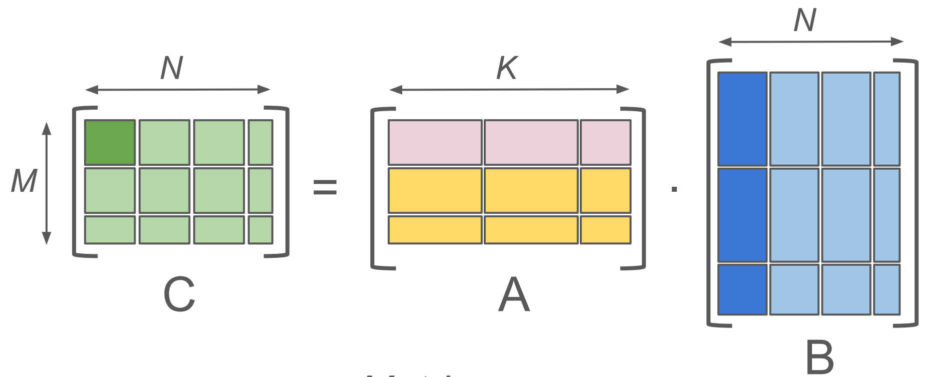
Arithmetic Intensity:

Load: 64·4·2 = 512
Matmult: 2·64·64·4 = 32768 ops
Arithmetic intensity = 64

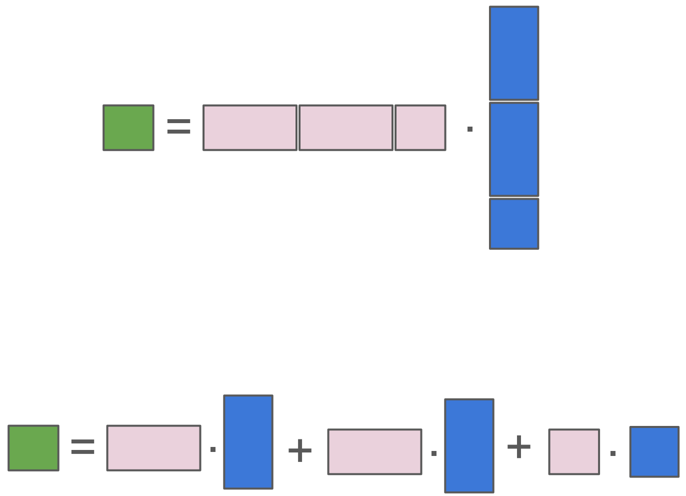
A, B: 1 VREG each
C: 16 VREGs

Fundamentals

Matrix - Tile



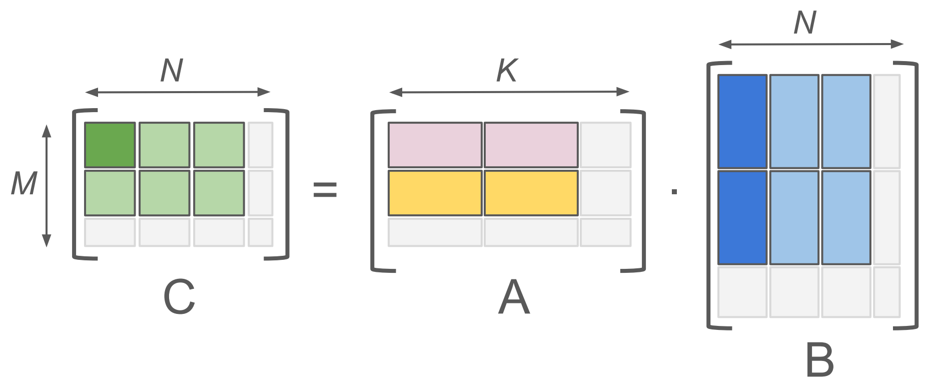
Matrices



Tiles

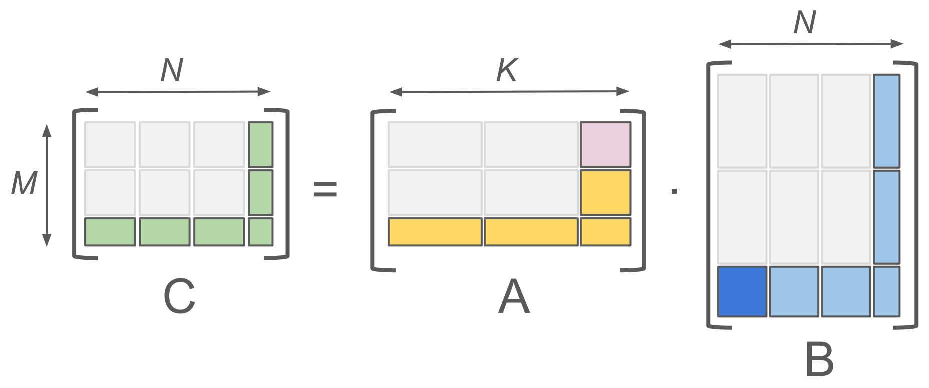
Multiplication of the large matrices C = A·B is decomposed into sums of their multiplied tiles. Tiles are multiplied and added by the tensor unit.

Matrix - Tile



Maximum tile sizes, as supported by the tensor unit. Dimensions:

- C-tiles: $m_{max} \times n_{max}$
- A-tiles: $m_{max} \times k_{max}$
- B-tiles: $k_{max} \times n_{max}$



Border tiles have smaller dimensions. Tensor units may or may not be able to operate on smaller tiles.

Tile - Matrix Unit - Vector Register

Matrix / tensor unit operates on tiles.

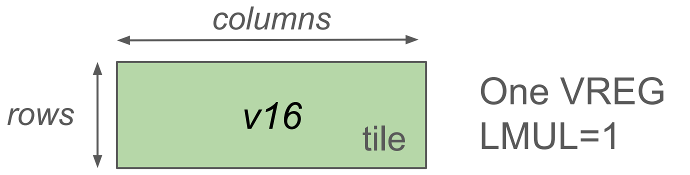
Tile dimensions are constrained by

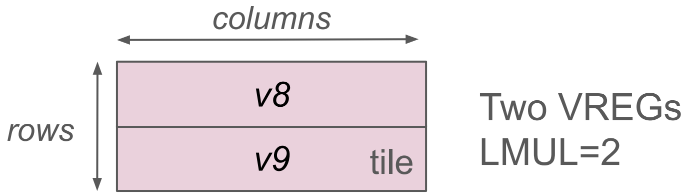
- Tensor unit geometry: m_{max}, k_{max}, n_{max}
  - May depend on SEW, data type
- Vector register size (VLEN, SEW)

Tiles are stored in vector registers.

- One or several (LMUL)
- Different tiles can have different LMUL

Matrix tile data layout in vector register is **<u>not</u>** specified and left to the implementation!



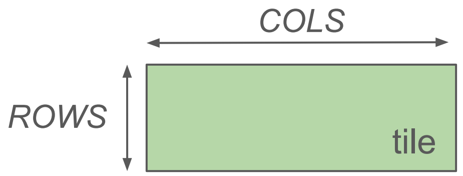


Tile Shape

A matrix tile is characterized by:

- VREG vector register number
- ROWS number of rows
- COLS number of columns
- SEW single element width
- MTYPE matrix type (A, B, C)
- (LMUL)
- (DTYPE data type (signed/unsigned/float/alt float))

$\left.\rule{0pt}{3em}\right\}$ ***tile shape***



MAXROWS, MAXCOLS depends on tensor unit geometry and tile type (A, B, C).

Tile Shape Data

The Tile Shape is a 32 bit entity that contains all information needed by the tensor unit to interpret correctly the tile data stored in a vector register or a vector register group.

- **rows/cols**: number of rows, columns of the tile
  - Values can vary widely: eg. 1 to 4096
    - Maximum VLEN=32768: 4096 elements in int8
  - Odd tail / border values should be handled as well
- **sew**: single element width
- **mtype**: tile type enum A, B, C
  - Load unit must recognize whether internal tile representation is in column-major-order or row-major-order.
- (**dtype**: Data type of tile: unsigned/signed/float/alt float)

Example:

```
typedef struct {
  unsigned int rows : 12;
  unsigned int cols : 12;
  unsigned int sew : 3;
  unsigned int mtype : 2;
} TileShape_32b;
```

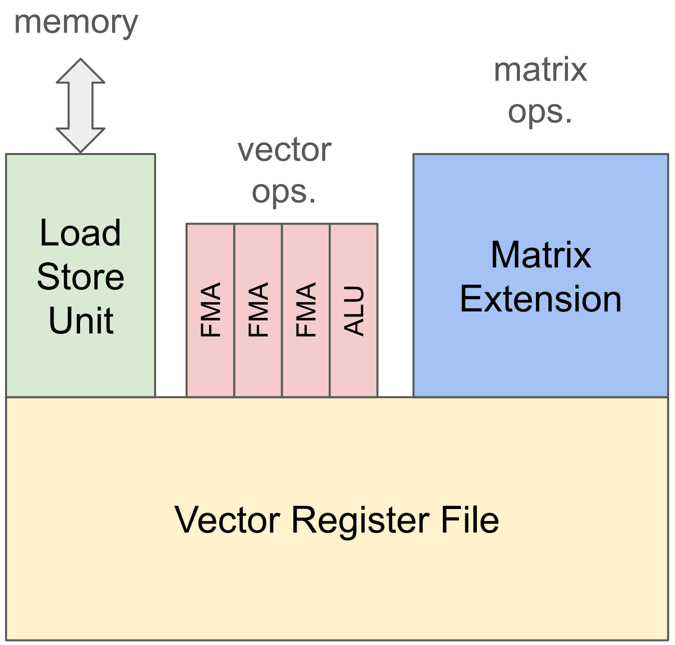
```
typedef enum {
  MTYPE_A, MTYPE_B, MTYPE_C,
} mtype_t;
```

Tensor Unit Configuration Needs

(1) Load matrix tile into VREG
Store matrix tile to memory

- Tile dimensions
- SEW
- Matrix type (A, B, C) for possible internal conversion

=> tile shape



(2) Multiply matrix tiles

- m×k×n,
- For each tile A, B, C:
  - SEW & DTYPE

or

- The 3 tile shapes of A, B, C
- (DTYPE, if not in tile shape)

(1) and (2) should be able to run simultaneously

Element-wise Matrix Operations

With matrix tiles stored in vector registers element-wise matrix operations can be done by using vector instructions.

- The effective or granted vector length depends on the size and shape of the tile.
- The tensor unit implementation can use masking to represent tiles smaller than the maximum tile size.
- Element-wise operations may need to use the corresponding, implementation-specific, vector mask.

Examples:

- Add two tiles of same shape
- Multiply each element of a tile with scalar value
- Initialize tile
- Apply activation function
- …

Intrinsics/ISA

Core Logic

1. Get Tile Shape(s) for A, B, C

Iterate, strip-mine.
Like ***vl = vsetvl(avl)*** but for 2D tiles

ask tensor unit for the largest tile shape needed

tensor unit replies with largest tile shape it supports

2. Load A, B, C from memory

Like ***vleNN.v*** but for 2D tiles

Use tile shape data

3. Multiply C=C+A·B

Like ***vfmacc.vv*** but for 2D tiles

Use tile shape data

4. Store C to memory

Like ***vseNN.v*** but for 2D tiles

Use tile shape data

Get Tile Shape

```
uint32_t tsget_T(int8_t mtype, uint32_t max_rows, uint32_t max_columns)
```

Retrieve the largest possible Tile Shape for a particular tile type A, B or C and the data type T.
T is one of {**i8**, i16, i32, i64, u8, u16, u32, u64, **f16**, f32, f64}.
Equivalent to vsetvl intrinsic.

Arguments:

- **mtype:** matrix type enum value for A, B, C type.
- **max_rows**: maximum number of rows needed in the tile.
- **max_columns**: maximum number of columns needed in the tile.

Returns:

- A Tile Shape value that is supported by the tensor unit implementation.

Get Tile Shape

Remarks:

The tile shape value removes the need to have state stored either in the tensor unit or along the vector register containing a tile.

It eases optimization in the compiler like unrolling and reordering tile operations.

Requesting an unsupported data type T could either raise an exception or return a Zero tile shape value.

Letting the tensor implementation return the appropriate tile shape relieves us of the burden to manage various $\mathbf{m_{max} \times k_{max} \times n_{max}}$ combinations for different **data types** and **SEWs**.

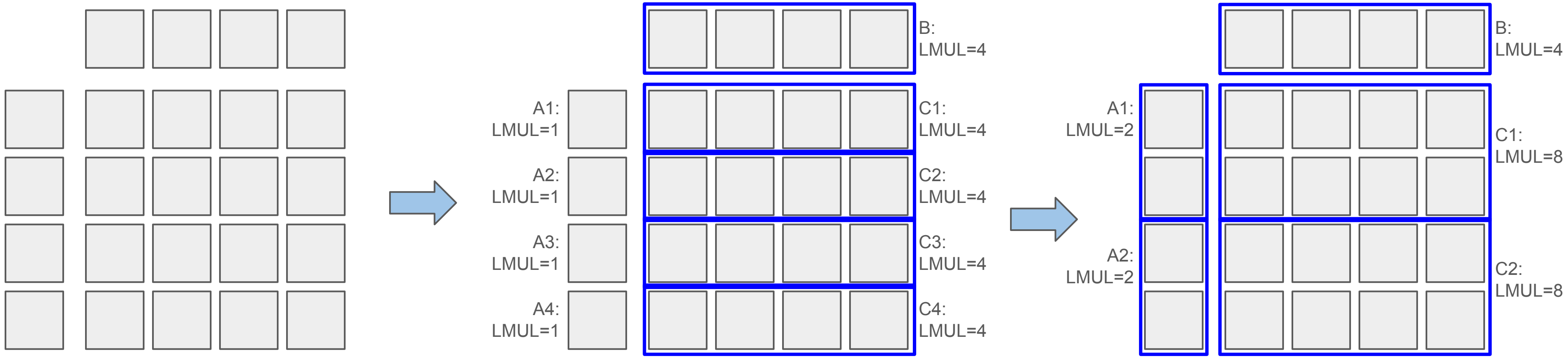
Querying tile shapes simplifies iterating over the tiles when multiplying large matrices.

Add LMUL to type T? Eg. ***tsget_f16m4()***

Get Tile Shape

Remarks:

Add LMUL to type T? Eg. ***tsget_f16m4()*** : Looks useful, needs further discussion.



LMUL=1 : 16 tile multiplies

4 tile multiplies

2 tile multiplies
(but less regular pattern in C)

Load Tile From Memory

`vT_t tload_T(vT_t vd, uint64_t addr, uint32_t str_col, uint32_t str_row, uint32_t ts)`

Load a tile shaped according to the tile shape ***ts*** from memory address ***addr*** into the vector register ***vd*** considering the column and row strides ***str_col***, ***str_row***.

T is the tile element data type (eg. f16 or f16m1).

Arguments:

- **vd**: target vector register
- **addr**: start address of the tile in memory
- **str_col**: column stride of the tile data in memory
- **str_row**: row stride of the tile data in memory
- **ts**: tile shape formerly retrieved through call to ***tile_get_shape()***

On NEC SX-Aurora VE:

VLD2D	**vld2d**[.*nc*]	{%vx | %vix}, {%sy | I}, {%sz | Z}	Vector Load 2D

Store Tile To Memory

`tstore_T(vT_t vs, uint64_t addr, uint32_t str_col, uint32_t str_row, uint32_t ts)`

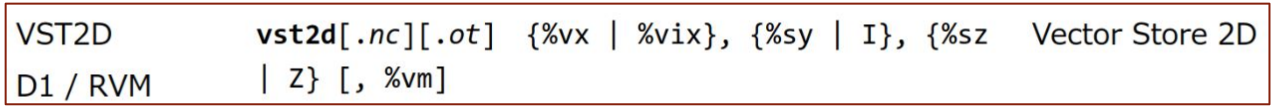
Store a tile shaped according to the tile shape ***ts*** from the vector register ***vs*** to memory address ***addr*** considering the column and row strides ***str_col***, ***str_row***.

T is the tile element data type(eg. f32 or f32m1).

Arguments:

- **vs**: source vector register
- **addr**: start address of the tile in memory
- **str_col**: column stride of the tile data
- **str_row**: row stride of the tile data
- **ts**: tile shape retrieved through ***tile_get_shape()***

On NEC SX-Aurora VE:



VST2D D1 / RVM	**vst2d**[.*nc*][.*ot*] {%vx | %vix}, {%sy | I}, {%sz | Z} [, %vm]	Vector Store 2D

Load / Store Tile

Remarks:

- The way how tile data is laid out in the vector register(s) is not specified and should remain implementation specific. It can be column-major-order or row-major-order or depend on the A, B, C matrix type.
- Passing both strides (row/column) helps the tensor unit transpose the tile when loading it, eg. if the internal representation requires column-major-order but data is row-major-order in memory. Order in memory doesn’t matter.
- 2D slices of higher dimensional tensors can be loaded/stored.
- Strides in bytes or bytes in SEW?
- The intrinsics get transformed into (at least) 2 instructions: a ***vsetvli*** and the actual tile load instruction.

Multiply And Accumulate Tiles

$vT_C m1\_t$ $\mathbf{tXmacc\_T_C}(vT_C m1\_t$ $\mathbf{vc}$, $vT_{AB} m1\_t$ $\mathbf{va}$, $vT_{AB} m1\_t$ $\mathbf{vb}$, uint32_t **tc**, uint32_t **ta**, uint32_t **tb**)

Perform the multiply and accumulate operation ***C=C+A·B*** with tile ***A*** in vector register ***va***, having shape ***ta,*** B stored in ***vb***, shape ***tb*** and the in/out tile ***C*** in ***vc*** with shape ***tc***.
X specifies operation type and widening. Eg. tfmacc, twmacc, tfwmacc, tqmacc, etc.

Arguments:

- vc: vector register that contains tile C.
- va: vector register containing the tile A.
- vb: vector register containing the tile B.
- tc: tile shape of C.
- ta: tile shape of A.
- tb: tile shape of B.

Multiply Tiles

Remarks:

- The intrinsics have 6 arguments: 3 x VREG + 3 x TileShape
  - Not feasible for assembler instructions
- Intrinsics get converted into at least two assembler instructions
  - Setup Tensor Unit: takes the three tile shapes as arguments
    - New instruction **tmulset**, similar to **vsetvli**
  - Matrix Multiplication instruction: takes the three VREGs as arguments
- Handling LMUL is more complicated than in the vector case. LMUL in T_C with narrower T_A, T_B does not imply that $LMUL_{A,B} < LMUL_C$

Tile Shape Helpers

Tile Shape Rows And Columns

```
uint16_t tsrows(uint32_t ts)
```

```
uint16_t tscols(uint32_t ts)
```

Extract ***rows*** and ***columns*** values from tile shape ***ts***. This function can be a macro or compiler builtin, as it only does an AND and a SHIFT operation on the tile shape value. Expands to existing instructions.

Arguments:

- ts: tile shape.

Returns:

- Rows, columns values encoded in the tile shape variable.

Tile Shape Vector Length

```
size_t tsvl(uint32_t ts)
```

Return implementation specific vector length in elements that needs to be used for element-wise vector operations on the tile with the tile shape ***ts***.

Arguments:

- ts: tile shape

Returns:

- Vector length for element-wise operations on tile elements.

Tile Shape Vector Mask

```
vboolX_t tsmask(vboolX_t vd, uint32_t ts)
```

The implementation specific vector mask that needs to be used for element-wise vector operations on the tile with the tile shape ***ts*** is stored in the destination vector register ***vd***. Typically masks are needed when the tile shape doesn't have the maximum size.

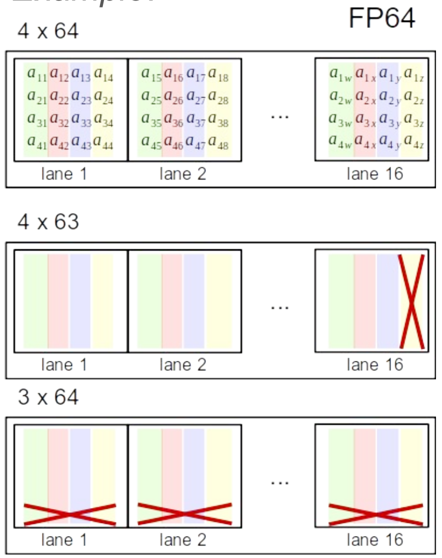
Arguments:

- vd: destination vector register for vector mask.
- ts: tile shape.

Returns:

- Vector mask for element-wise operations on tile elements.

Example:



Assembler

tsget	Tile shape get	
tload	Tile load	
tstore	Tile store	
tmacc	Tile multiply-accumulate	tfmacc, tfwmacc, twmacc, tqmacc, tfqmacc, tmaccu,...
tmulset	Tile multiplier set	
tsvl	Tile shape vector length	
tsmask	Tile shape mask	

Tensor Unit CSR (example)

(Could fit into 64 bits)

```
struct TensorUnitCSR {
  // – tile shapes –
  unsigned int m : 12;
  unsigned int k : 12;
  unsigned int n : 12;
  unsigned int sew_a : 3;
  unsigned int sew_b : 3;
  unsigned int sew_c : 3;
  // – data type –
  unsigned int float : 1;
  unsigned int alt_float : 1;
  unsigned int unsig : 1;
  // – further config –
  unsigned int sparse : 1;
  // – status –
  unsigned int status : X;
  // – implementation specific –
  unsigned int impl : Y;
};
```

NOTE: Adding data type information to Tile Shape reduces number of *tmacc* instruction variants.

Example

GEMM C = A · B

Matrix data types: C: FP32 A,B: FP32

```
for m = 0, M, m += tm do
    for n = 0, N, n += tn do
        tc = tsget_f32(MTYPE_C, M - m, N - n)
        tm = tsrows(tc)
        tn = tscols(tc)
        gvl = tsvl(tc)
        c = __riscv_vfmv_v_f_f32m1(0.0f, gvl)
        for k = 0, K, k += tscols(ta) do
            ta = tsget_f32(MTYPE_A, tm, K - k)
            tb = tsget_f32(MTYPE_B, K - k, tn)
            a = tload_f32(&A[m * LDA + k], LDA, 1, ta)
            b = tload_f32(&B[k * LDB + n], LDB, 1, tb)
            c = tfmacc_f32(c, a, b, tc, ta, tb)
        tstore_f32(c, &C[m * LDC + n], LDC, 1, tc)
```

GEMM $C = \text{alpha} \cdot A \cdot B + \text{beta} \cdot C$

Matrix data types: C: FP32 A,B: FP16

```
for m = 0, M, m += tm do
    for n = 0, N, n += tn do
        tc = tsget_f32(MTYPE_C, M - m, N - n)
        tm = tsrows(tc)
        tn = tscols(tc)
        gvl = tsvl(tc)
        cmask = tsmask(tc)
        c = __riscv_vfmv_v_f_f32m1(0.0f, gvl)
        for k = 0, K, k += tscols(ta) do
            ta = tsget_f16(MTYPE_A, tm, K - k)
            tb = tsget_f16(MTYPE_B, K - k, tn)
            a = tload_f16(&A[m * LDA + k], LDA, 1, ta)
            b = tload_f16(&B[k * LDB + n], LDB, 1, tb)
            c = tfwmacc_f32(c, a, b, tc, ta, tb)
        t = tload_f32(&C[m * LDC + n], LDC, 1, tc)
        c = __riscv_vfmul_vf_f32m1(c, alpha, gvl, cmask)
        c = __riscv_vfadd_vf_f32m1(t, beta, gvl, cmask)
        tile_store_fp32(c, &C[m * LDC + n], LDC, 1, tc)
```

TODO

- Discover supported data formats (!)
- Multiple VREG Tiles
  - Code for square matrices pattern (A) with 4x4 C-type + 4 A-type + 4 B-type = 24 VREGs
  - Not optimal for narrow matrices (B): 16 C-type VREGs + 1 A-type + 1 B-type = 18 VREGs
  - Pattern (A) probably still good enough to reach peak performance/throughput!
  - Add new Multi-Tile-Shape to describe optimal topology?
    - Eg. add rows, columns of tiles for optimal performance
    - Generate code dynamically at runtime for presented topology?
    - Or let compiler generate alternative code for extreme cases and chose path according to retrieved shape?
  - LMUL in tile shape: not sufficient for 2D tile patterns
    - If LMUL is hidden in an argument (ts): not possible to do register allocation
    - Helps reduce number of instructions.
- Sparse matrix operations
- Other matrix operations

THE END