RISC-V Matrix Extension Introduction

Zhiqiang Liu, Xin Ouyang

Stream Computing Inc.

Agenda

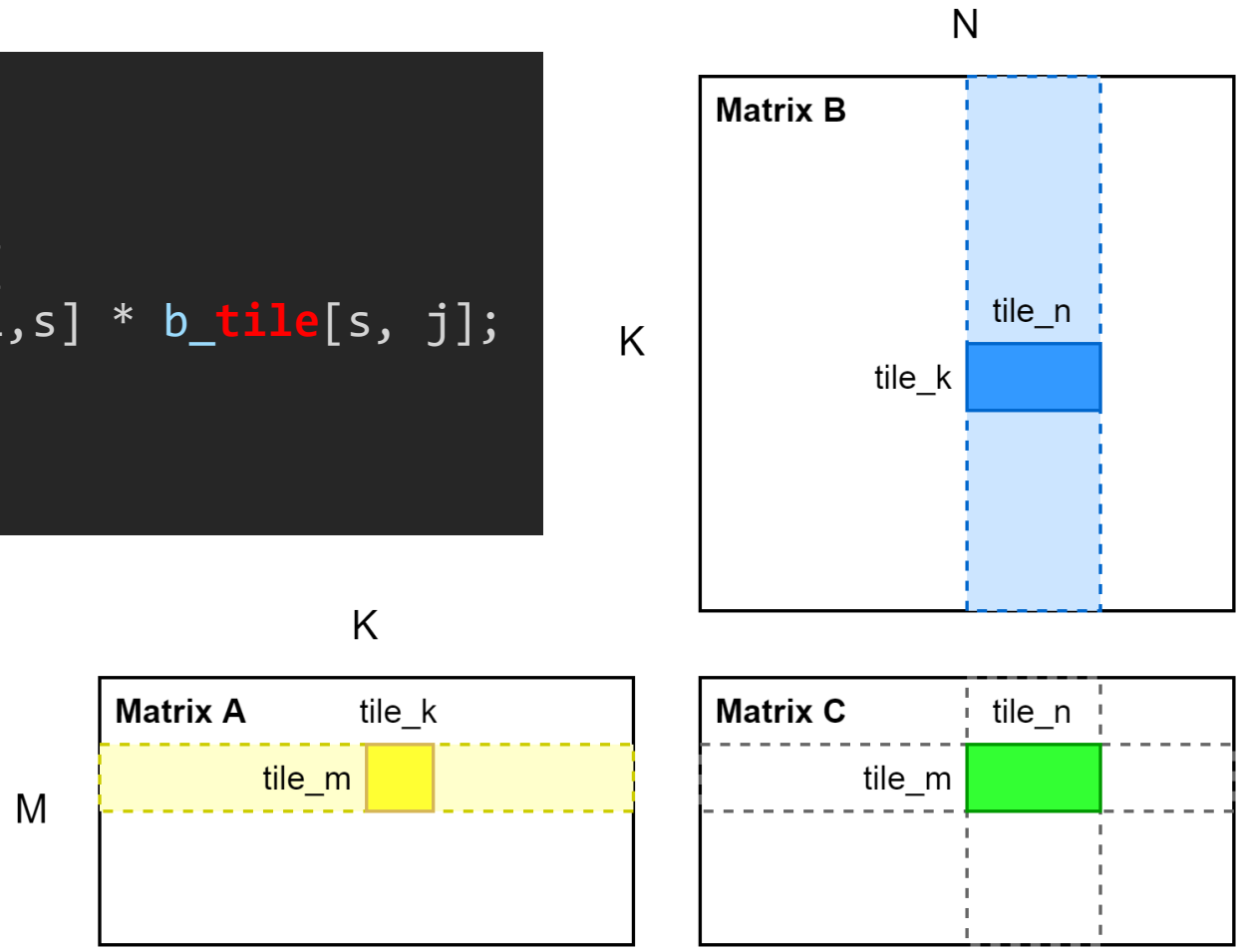
- Introduction
- Programmer's Model
- Instructions
- Standard Matrix Extensions
- Open Source Projects
- Future Works

Matrix Extension vs. Vector Extension

- Highly inspired by the RISC-V vector extension
- Standalone extension with no need of vector extension support
- Use Zmv extension to bridge vector and matrix extensions
  - exchange matrix data between vector registers and **memory**
  - exchange matrix data between vector registers and **matrix registers**

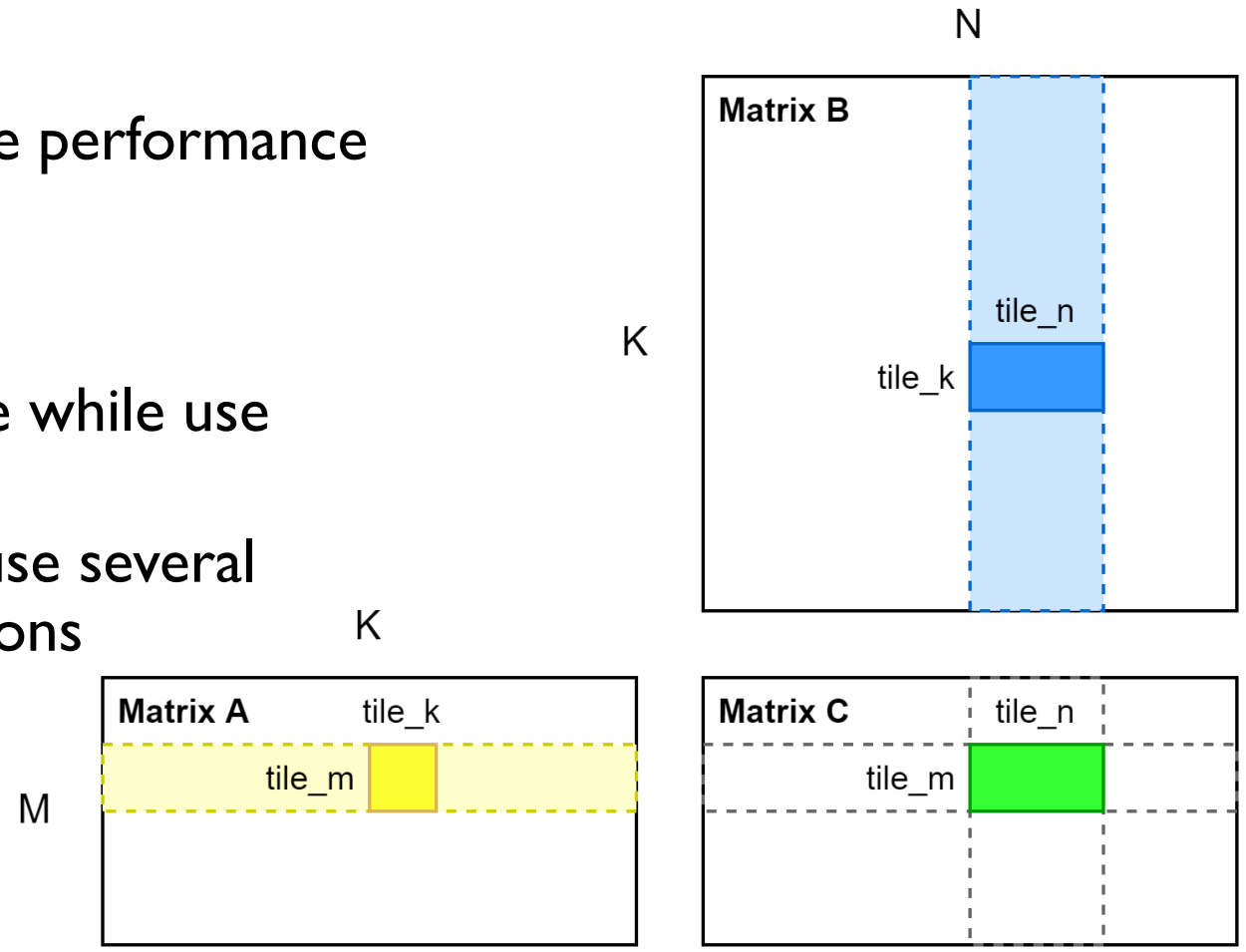
Tiled Matrix Multiplication

```
for (i=0; i<m; i+=tile_m) {
    for (j=0; j<n; j+=tile_n) {

        for (s=0; s<k; s+=tile_k) {
            c_tile[i,j] += a_tile[i,s] * b_tile[s, j];
        }
    }
}
```



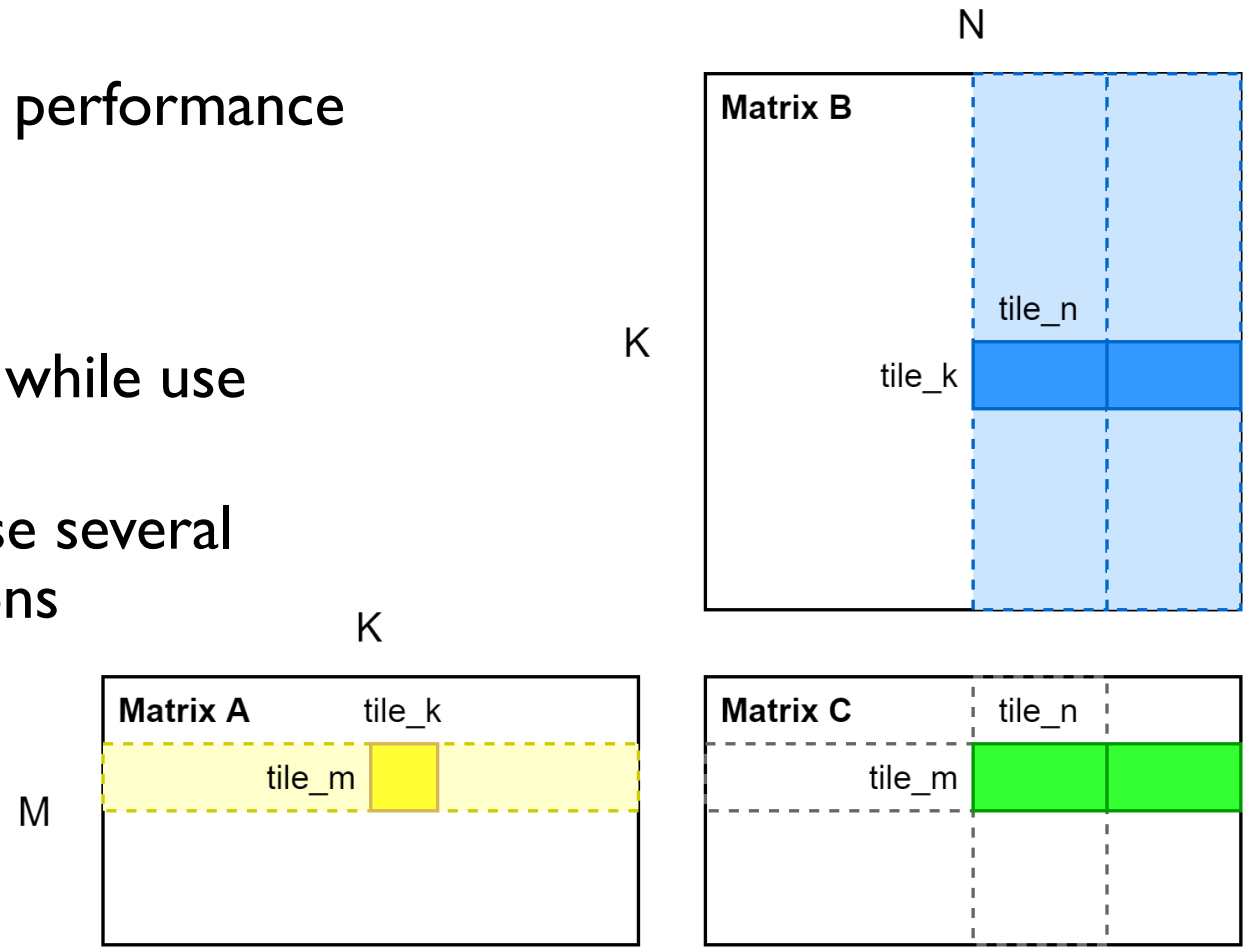
Tiled Matrix Multiplication

- Memory access friendly
  - Taking advantage of unit-stride performance
- Explore data reuse
  - Load an element in a tile once while use several times
  - Load an entire tile once and use several times in parallel implementations



Tiled Matrix Multiplication

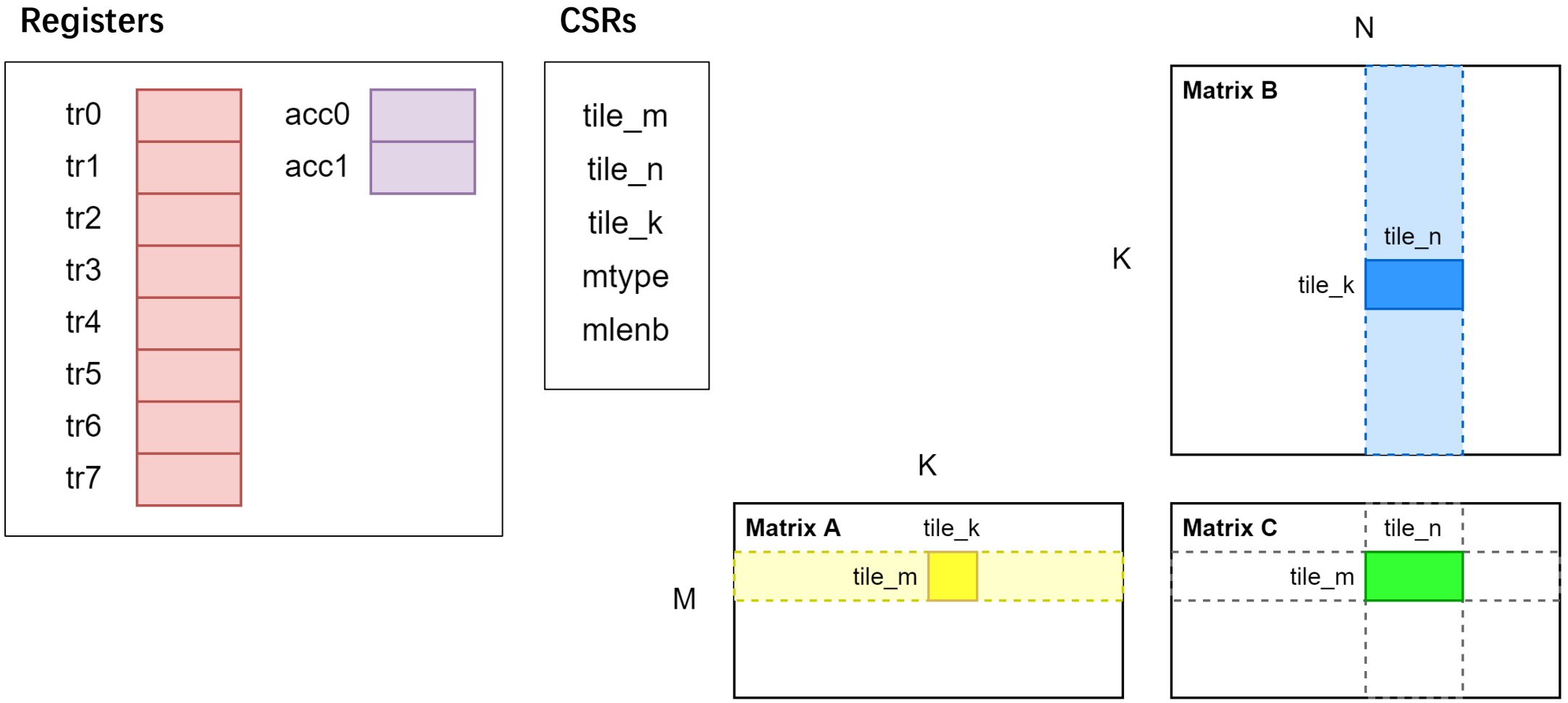
- Memory access friendly
  - Taking advantage of unit-stride performance
- Explore data reuse
  - Load an element in a tile once while use several times
  - Load an entire tile once and use several times in parallel implementations



Agenda

- Introduction
- Programmer's Model
- Instructions
- Standard Matrix Extensions
- Open Source Projects
- Future Works

Registers and CSRs



CSRs

- tile_m/tile_k/tile_n: Matrix tile configure registers
  - Unsigned integers specifying the shapes for tiled matrix
  - Updated by msettile[m|k|n]{i} instructions
- mtype: Matrix type register
  - Updated by msettype{i} instructions

Table 2. `mtype` register layout

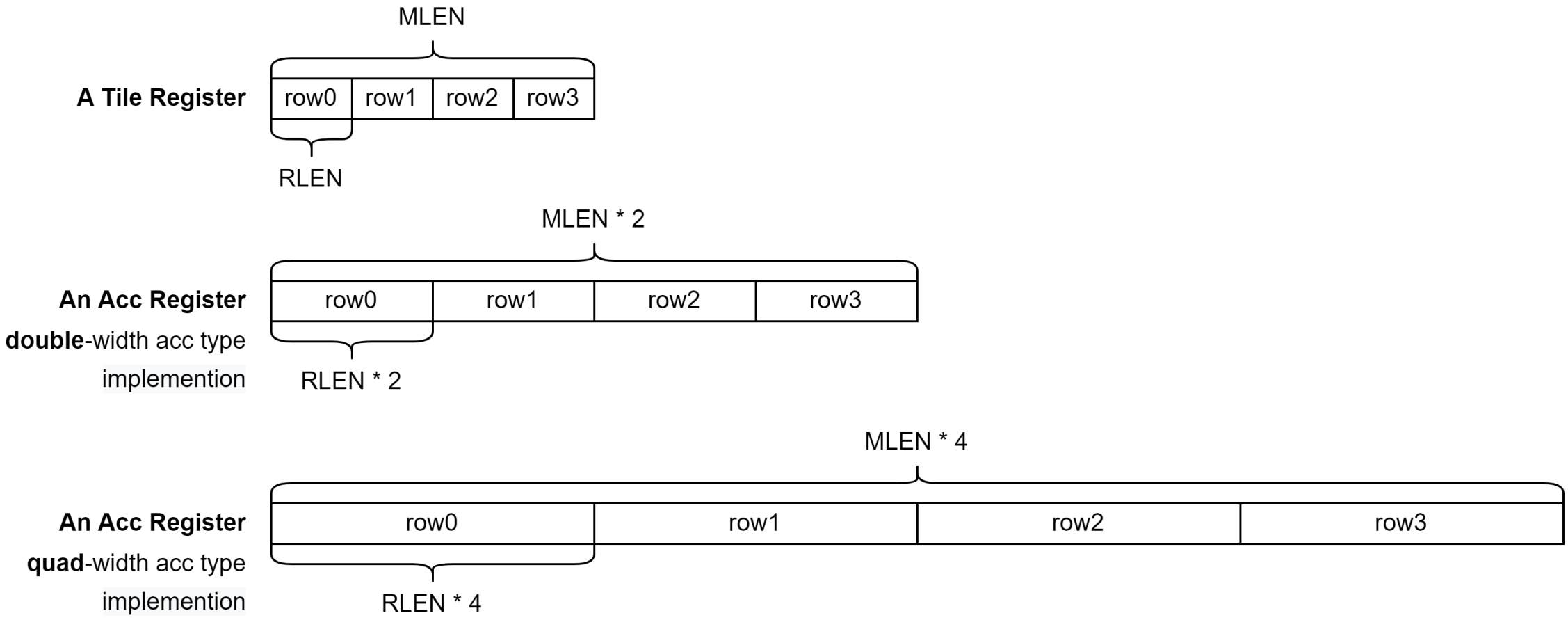
Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:4	0	Reserved if non-zero
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting

Implementation-defined Constant Parameters

- ELEN: Maximum bits of a matrix element, `ELEN>8`
- MLEN: Bits in a single matrix tile register, `MLEN<=`2^{32}
- RLEN: Bits in a row of a single matrix tile register, `RLEN<=`2^{16}

- `ELEN < RLEN < MLEN`
- Support matrix tile size from 2x2 to 2^{16} x 2^{16}

Tile and Accumulator Registers



Agenda

- Introduction
- Programmer's Model
- Instructions
- Standard Matrix Extensions
- Open Source Projects
- Future Works

Instruction Formats

- Opcode = 1110111

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

Instruction Formats

- Opcode = 1110111
- Type by `inst[14:12]`
  - 111: Configuration
  - 110: Arithmetic
  - 101: Data Move
  - `000-011`: Load & Store

Configuration instructions, funct3 = 111

31 28	27 20	19 15	14 12	11 7	6 0
funct4	imm13		funct3	rd	1110111
funct4	00000000	rs1	funct3	rd	1110111

Load & Store instructions, eew = 000 - 011

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	ls	rs2	rs1	eew	md	1110111

Arithmetic & Type-Convert instructions, funct3 = 110

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	fp	ms2	ms1	funct3	md	1110111

Data Move instructions, funct3 = 101

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	di	rs2	ms1	funct3	md	1110111

Configuration-Setting Instructions - msettype{i}

- msettype{i} instructions set the mtype CSR based on their arguments.

```
msettypei rd, mtypei    # rd = new mtype, mtypei = new mtype setting
msettype rd, rs1        # rd = new mtype, rs1 = new mtype value
```

Table 3. mtype register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:4	0	Reserved if non-zero
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting

Configuration-Setting Instructions - msettype{i}

```
Suggested assembler names used for msettypei mtypei immediate

    e8      # SEW = 8b
    e16     # SEW = 16b
    e32     # SEW = 32b
    e64     # SEW = 64b

    accq    # support 32-bit accumulator element

Examples:

    msettypei t0, e8            # SEW = 8
    msettypei t0, e32           # SEW = 16
    msettypei t0, e8, accq      # SEW = 8, support 32-bit accumulator
element
```

Configuration-Setting Instructions – msettile[m|k|n]{i}

- msettile[m|k|n]{i} instructions set the tile_m/tile_k/tile_n CSRs based on their arguments.

```
msettilemi rd, mleni    # rd = new tile_m, mleni = ATM
msettilem rd, rs1       # rd = new tile_m, rs1 = ATM

msettileki rd, mleni    # rd = new tile_k, mleni = ATN
msettilek rd, rs1       # rd = new tile_k, rs1 = ATN

msettileni rd, mleni    # rd = new tile_n, mleni = ATK
msettilen rd, rs1       # rd = new tile_n, rs1 = ATK
```

Configuration-Setting Instructions – msettile[m|k|n]{i}

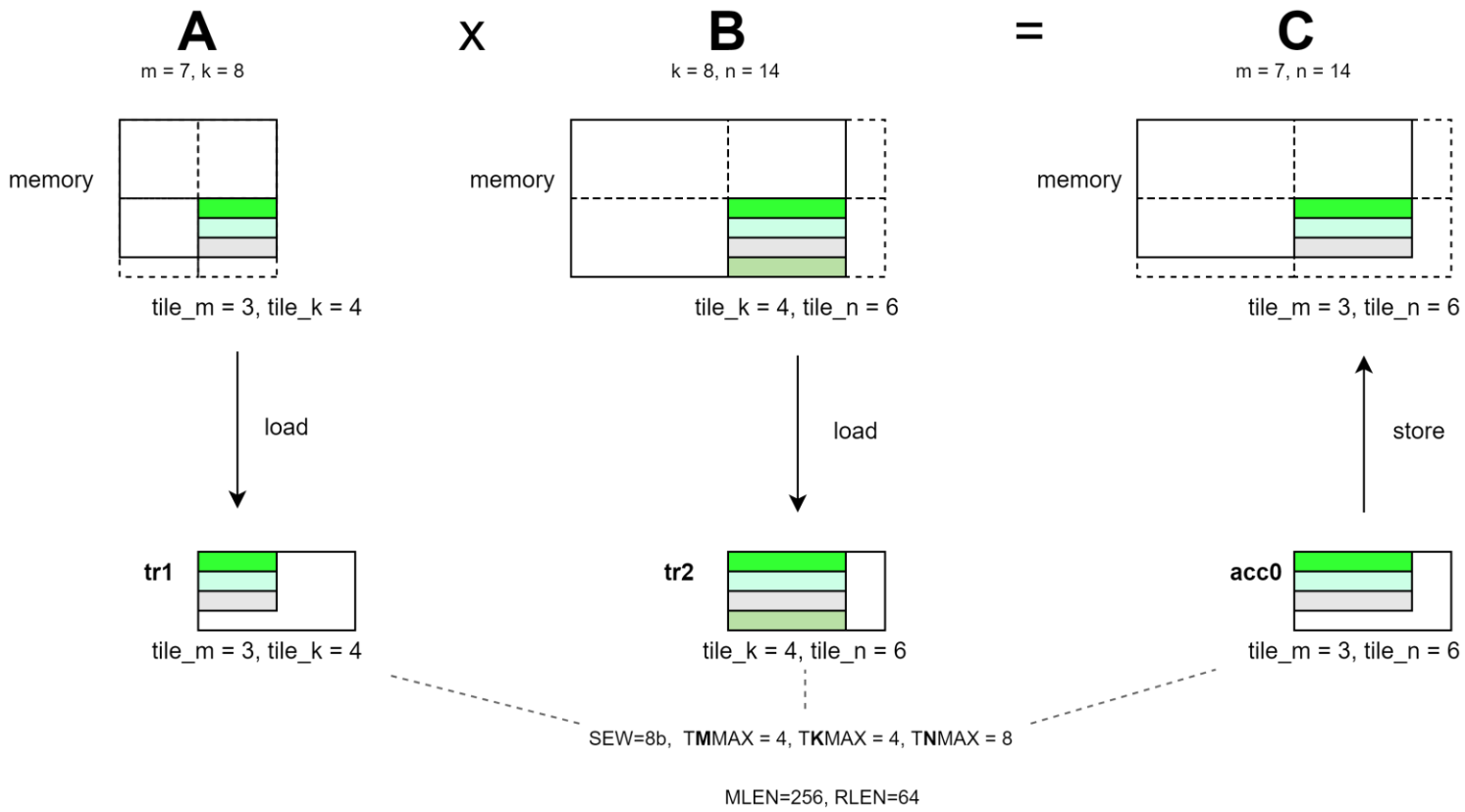
- TMMAX, TKMAX, TNMAX, represent the maximum shapes of matrix tiles that could be stored in matrix registers
  - TMMAX = MLEN / RLEN
  - TKMAX = min(MLEN / RLEN, RLEN / SEW)
  - TNMAX = RLEN / SEW
- MLEN=256, RLEN=64
  - When SEW=8b, TMMAX=4, TKMAX=4, TNMAX=8 # 4x4x8 8bit matmul
  - When SEW=16b, TMMAX=4, TKMAX=4, TNMAX=4 # 4x4x4 16bit matmul
  - When SEW=32b, TMMAX=4, TKMAX=2, TNMAX=2 # 4x2x2 32bit matmul

Configuration-Setting Instructions – msettile[m|k|n]{i}

- msettile[m|k|n]{i} set tile_m/tile_k/tile_n obeying the following constraints

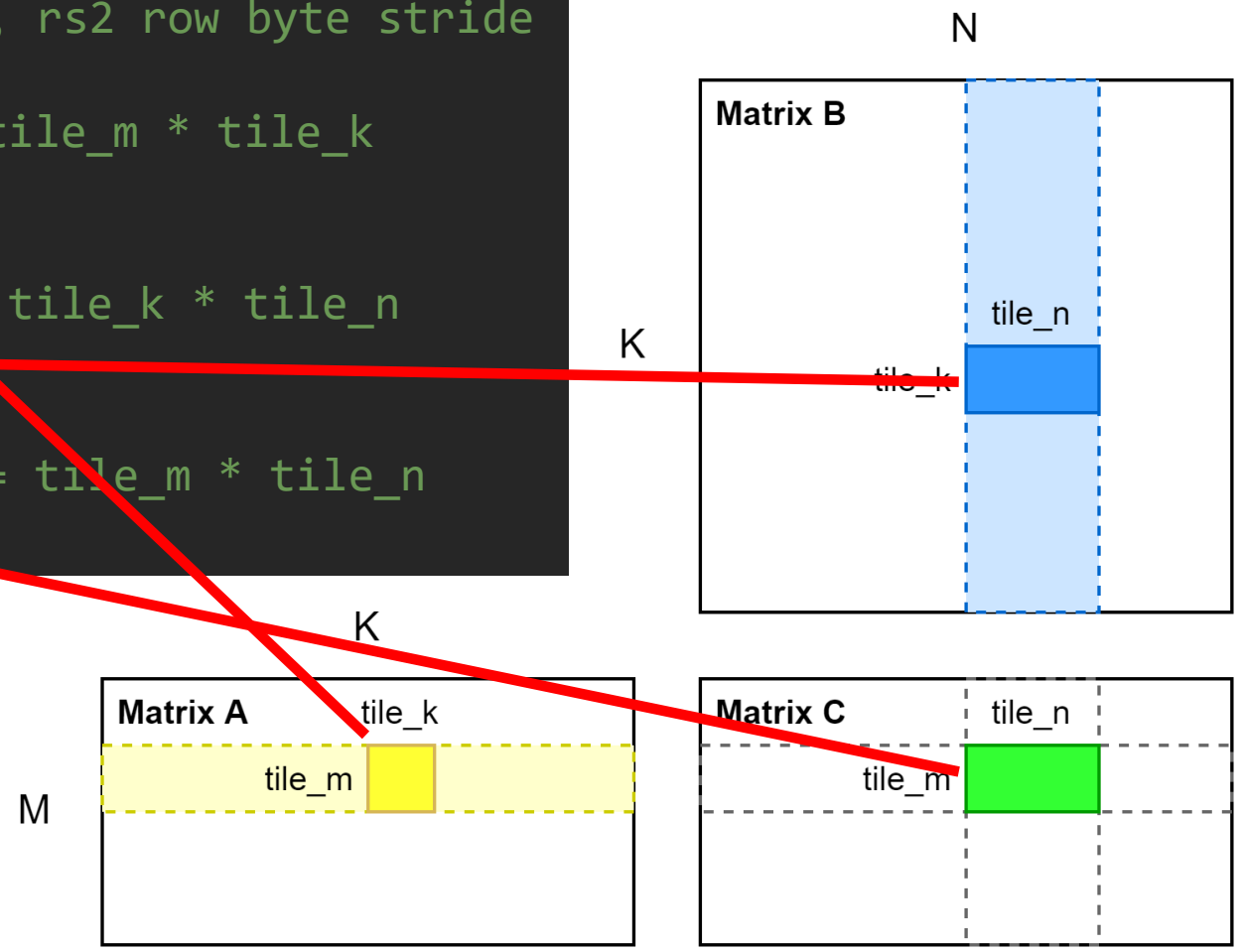
1. `tile_m = ATM` if `ATM <= TMMAX`
2. `ceil(ATM / 2) <= tile_m <= TMMAX` if `ATM < (2 * TMMAX)`
3. `tile_m = TMMAX` if `ATM >= (2 * TMMAX)`

Configuration-Setting Instructions Example



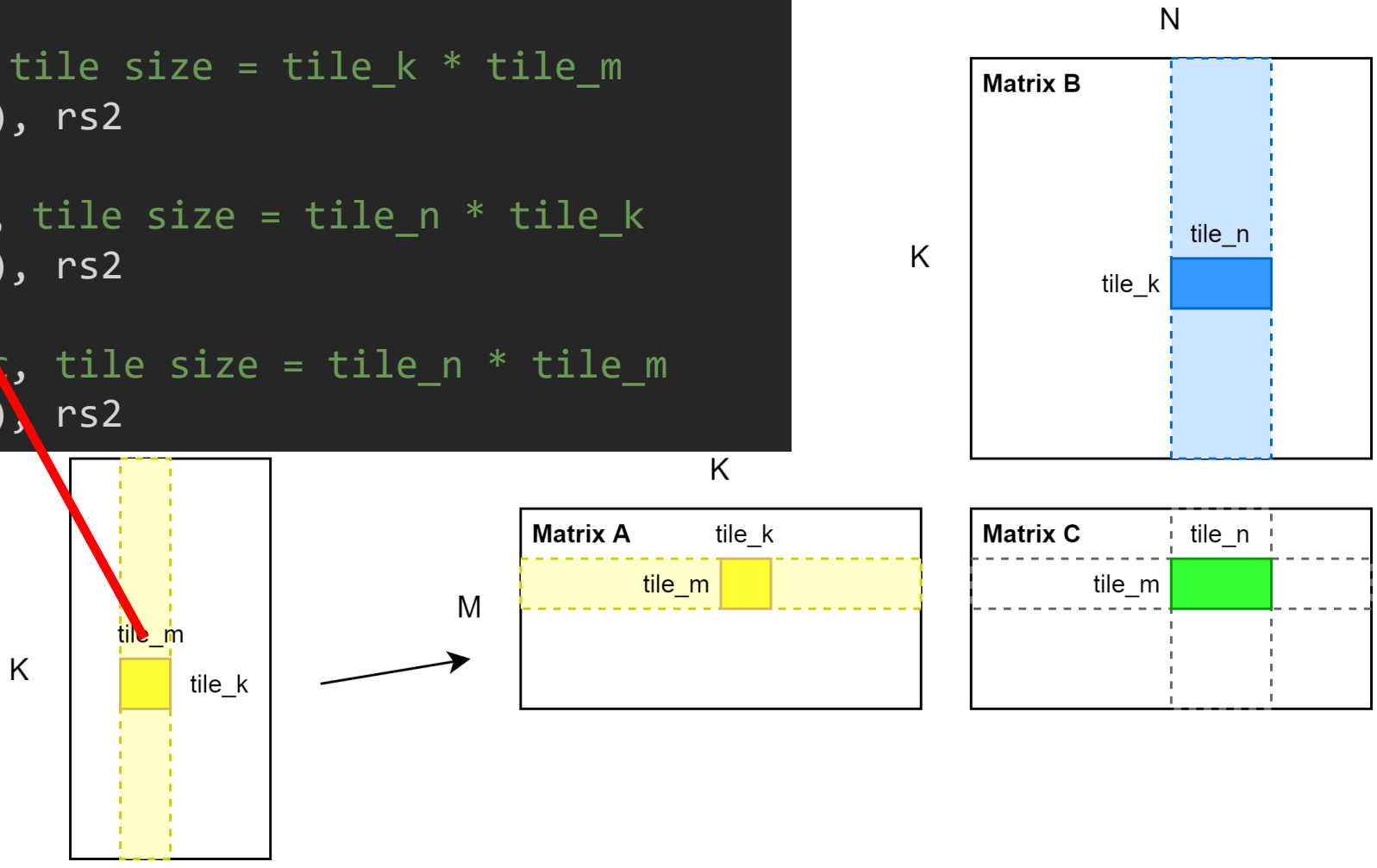
Load instructions

```
# md destination, rs1 base address, rs2 row byte stride

# for left matrix, a, tile size = tile_m * tile_k
mlae<eew>.m md, (rs1), rs2

# for right matrix, b, tile size = tile_k * tile_n
mlbe<eew>.m md, (rs1), rs2

# for output matrix, c, tile size = tile_m * tile_n
mlce<eew>.m md, (rs1), rs2
```



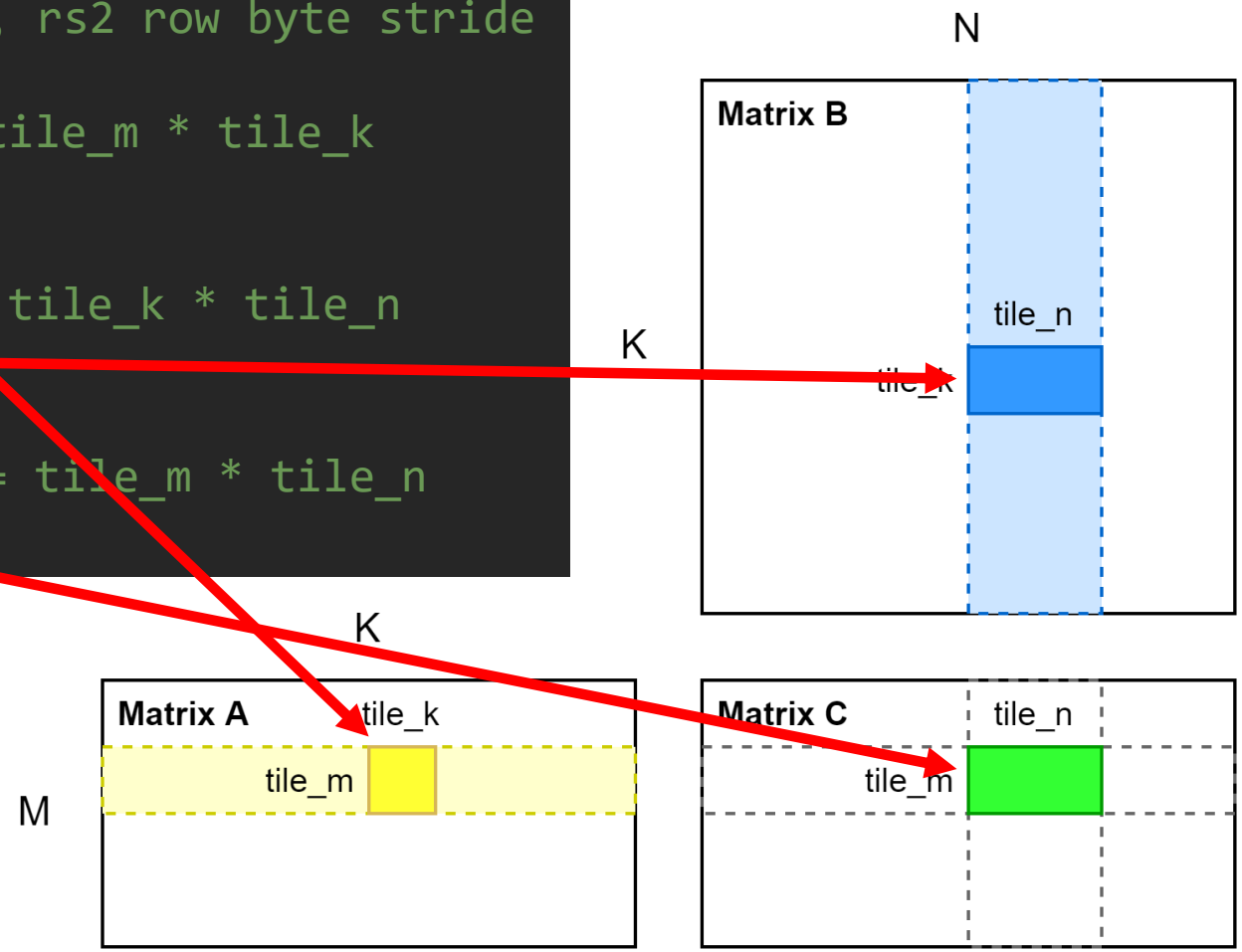
Load instructions with transposition

```
# md destination, rs1 base address, rs2 row byte stride

# for left matrix, a, tile size = tile_k * tile_m
mlate<eew>.m md, (rs1), rs2

# for right matrix, b, tile size = tile_n * tile_k
mlbte<eew>.m md, (rs1), rs2

# for output matrix, c, tile size = tile_n * tile_m
mlcte<eew>.m md, (rs1), rs2
```



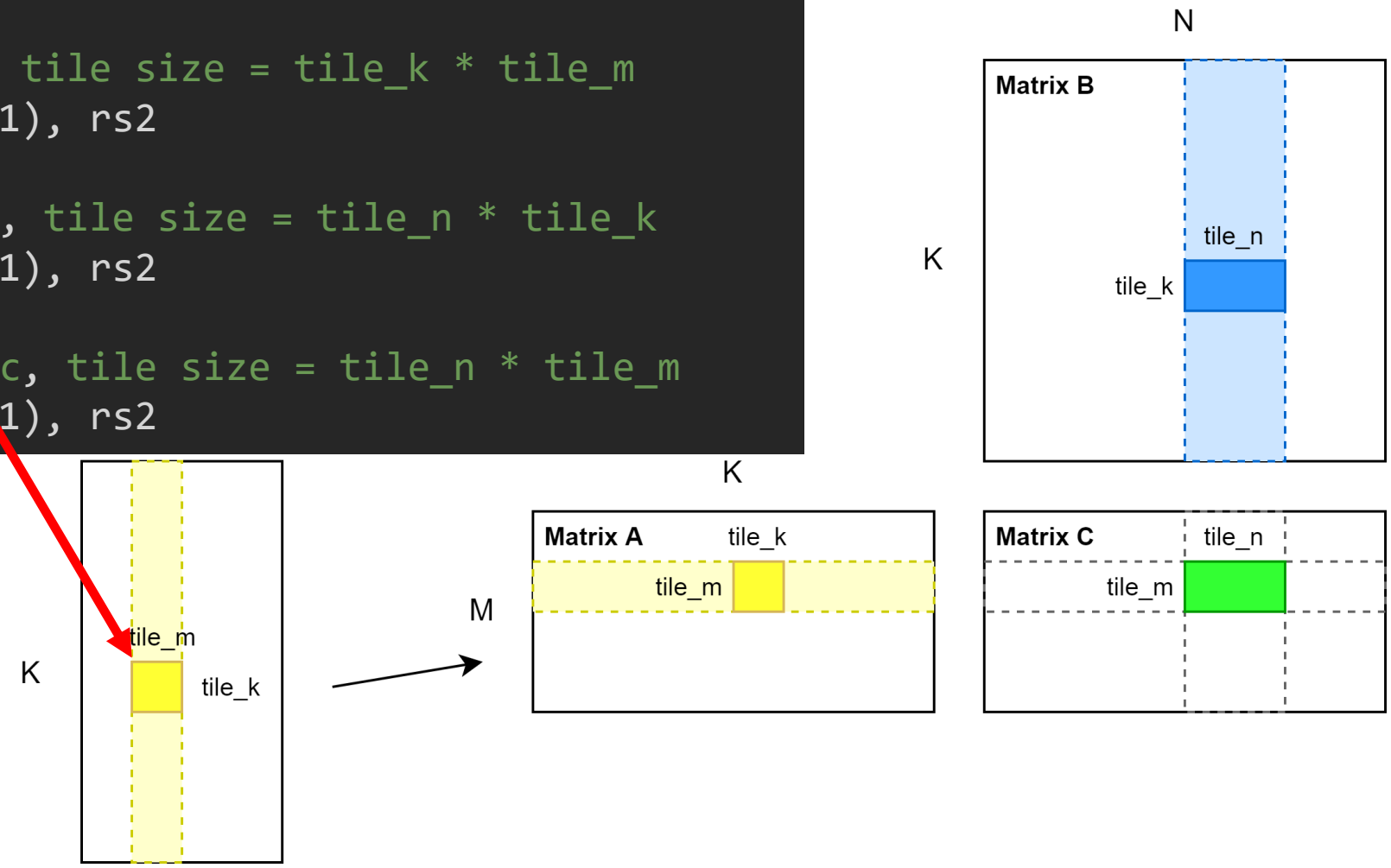
Store instructions

```
# md destination, rs1 base address, rs2 row byte stride

# for left matrix, a, tile size = tile_m * tile_k
msae<eew>.m md, (rs1), rs2

# for right matrix, b, tile size = tile_k * tile_n
msbe<eew>.m md, (rs1), rs2

# for output matrix, c, tile size = tile_m * tile_n
msce<eew>.m md, (rs1), rs2
```



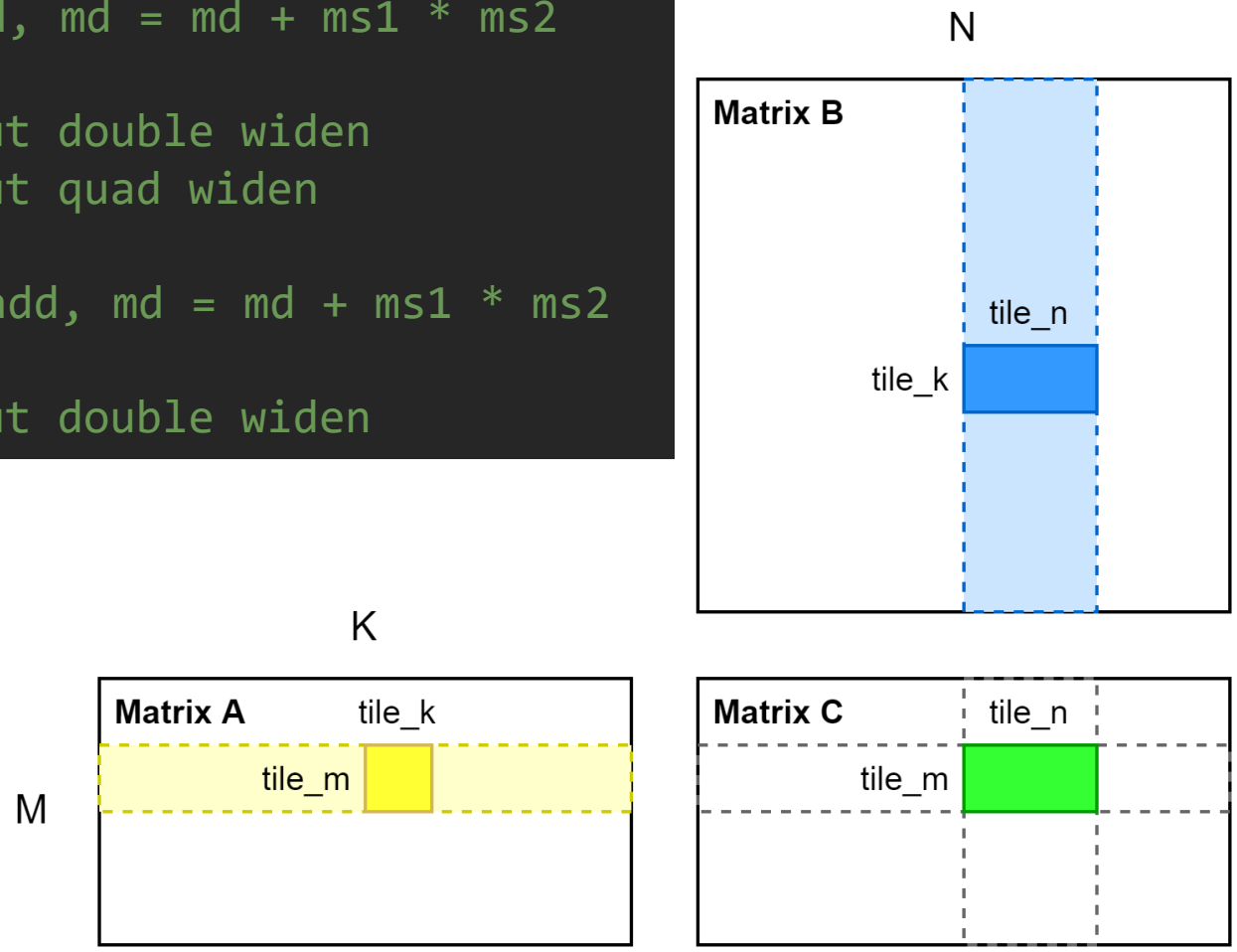
Store instructions with transposition

```
# ms3 store data, rs1 base address, rs2 row byte stride

# for left matrix, a, tile size = tile_k * tile_m
msate<eew>.m ms3, (rs1), rs2

# for right matrix, b, tile size = tile_n * tile_k
msbte<eew>.m ms3, (rs1), rs2

# for output matrix, c, tile size = tile_n * tile_m
mscte<eew>.m ms3, (rs1), rs2
```



Matrix Multiplication instructions

```
# int matrix multiplication and add, md = md + ms1 * ms2
mma.mm md, ms1, ms2
mwma.mm md, ms1, ms2          # output double widen
mqma.mm md, ms1, ms2          # output quad widen

# float matrix multiplication and add, md = md + ms1 * ms2
mfma.mm md, ms1, ms2
mfwma.mm md, ms1, ms2         # output double widen
```



Element-Wise Add/Sub/Multiply Instructions

```
# md[i,j] = md[i,j] + ms1[i,j]
m{w|q}addc.mm md, ms1
mf{w}addc.mm md, ms1

# md[i,j] = md[i,j] - ms1[i,j]
m{w|q}subc.mm md, ms1
mf{w}subc.mm md, ms1

# md[i,j] = ms1[i,j] - md[i,j]
m{w|q}rsubc.mm md, ms1
mf{w}rsubc.mm md, ms1

# md[i,j] = ms1[i,j] * rs2
m{w|q}emulc.mx md, ms1, rs2
mf{w}emulc.mx md, ms1, rs2

# md[i,j] = ms1[i,j] * imm
m{w|q}emulc.mi md, ms1, imm
```

Type-Convert Instructions

```
# convert float to float
mfncvtc.f.fw.m md, ms1   # double-width float to single-width float
mfwcvtc.fw.f.m md, ms1   # single-width float to double-width float

# convert int to float
mfcvtc.f.x.m md, ms1     # int to float
mfncvtc.f.xw.m md, ms1   # double-width int to float
mfncvtc.f.xq.m md, ms1   # quad-width int to float
mfwcvtc.fw.x.m md, ms1   # single-width int to double-width float
mfcvtc.fw.xw.m md, ms1   # double-width int to double-width float
mfncvtc.fw.xq.m md, ms1  # quad-width int to double-width float

# convert float to int
mfcvtc.x.f.m md, ms1     # float to int
mfwcvtc.xw.f.m md, ms1   # float to double-width int
mfwcvtc.xq.f.m md, ms1   # float to quad-width int
mfncvtc.x.fw.m md, ms1   # double-width float to single-width int
mfcvtc.xw.fw.m md, ms1   # double-width float to double-width int
mfwcvtc.xq.fw.m md, ms1  # double-width float to quad-width int
```

```
void matmul_float16(c, a, b, m, k, n) {
    msettype(e16);                              // use 16bit input matrix element

    for (i=0; i<m; i+=tile_m) {                 // loop at dim m with tiling
        tile_m = msettile_m(m-i);
        for (j=0; j<n; j+=tile_n) {             // loop at dim n with tiling
            tile_n = msettile_n(n-j);

            acc = mfemul_mf(acc, 0.f)           // clear acc reg
            for (s=0; s<k; s+=tile_k) {         // loop at dim k with tiling
                tile_k = msettile_k(k-s);

                tr1 = mlae16_m(&a[i][s], k*2);  // load left matrix a
                tr2 = mlbe16_m(&b[s][j], n*2);  // load right matrix b
                acc = mfwma_mm(tr1, tr2);       // tiled matrix multiply,
                                                // double widen output acc
            }

            acc = mfncvt_f_fw_m(acc);           // convert widen result
            msce16_m(acc, &c[i][j], n*2);       // store to matrix c
        }
    }
}
```

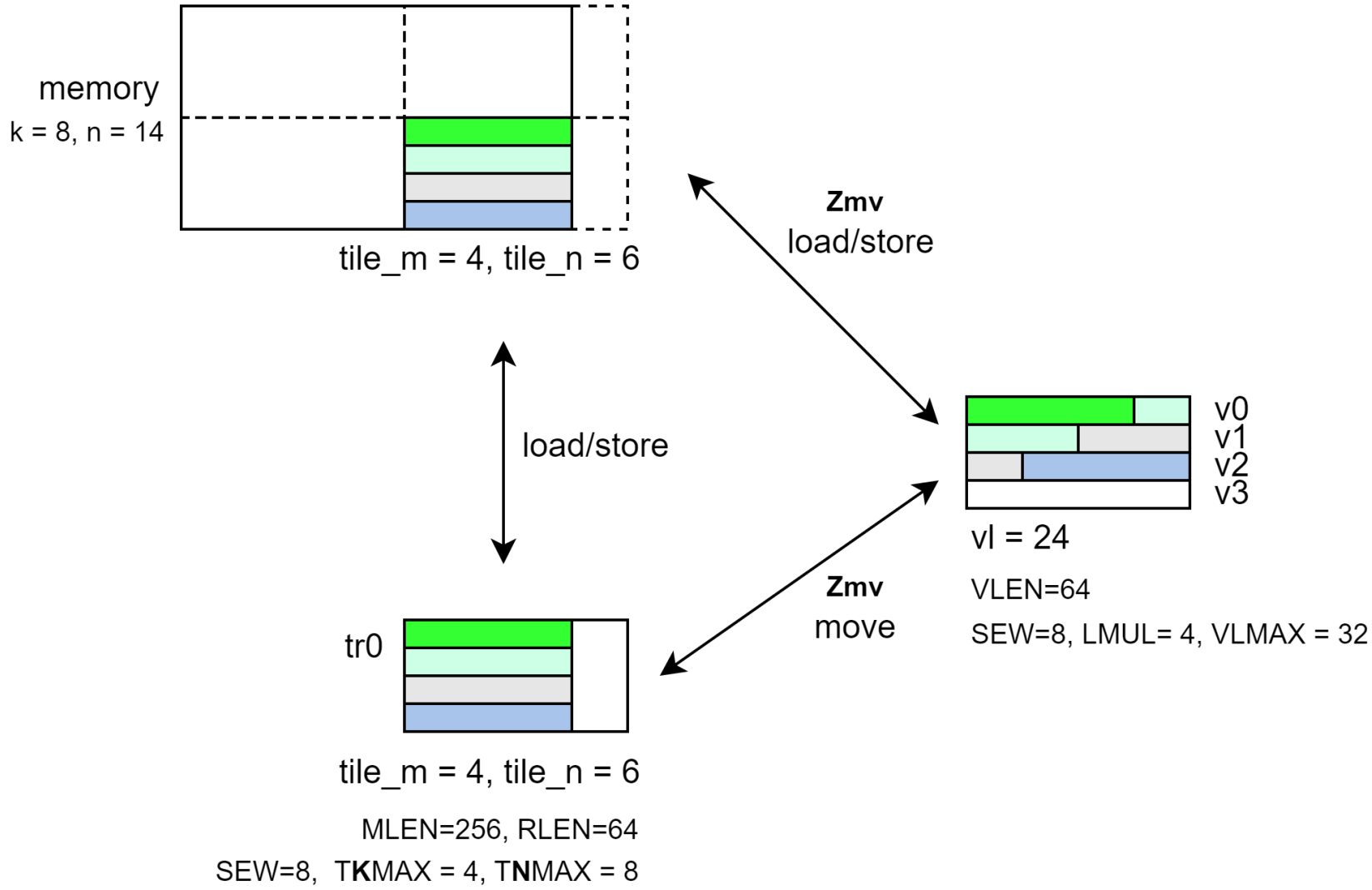
Agenda

- Introduction
- Programmer's Model
- Instructions
- Standard Matrix Extensions
- Open Source Projects
- Future Works

Zmv: Matrix for Vector operations

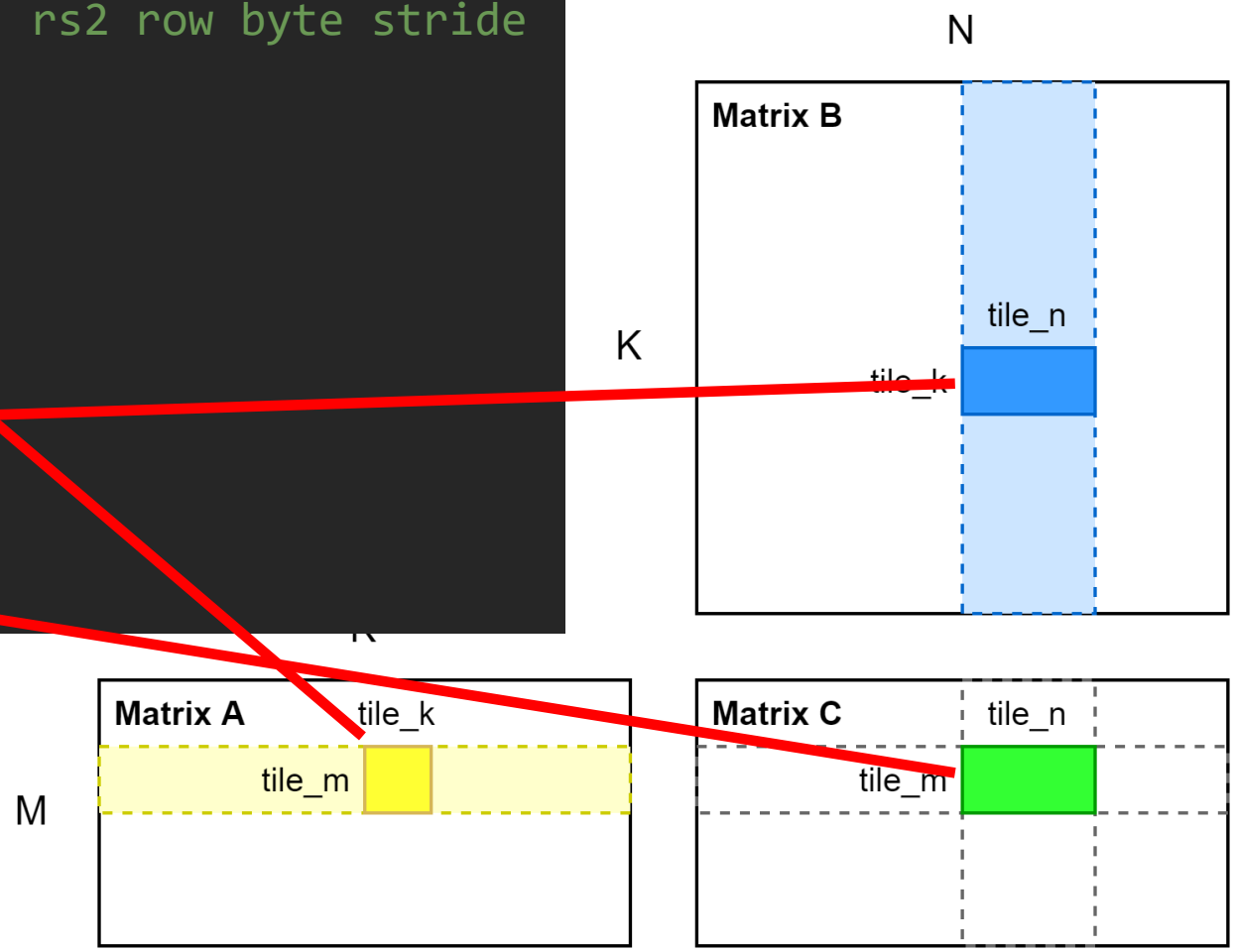
- Provide matrix support with the RISC-V Vector "V" extension.
  - Load matrix tile slices into vector registers
  - Store matrix tile slices from vector registers
  - Move data between slices of a matrix register and vector registers
  - Element-wise multiply with a matrix register and a vector register(broadcast to a matrix).

Load/store, data move example



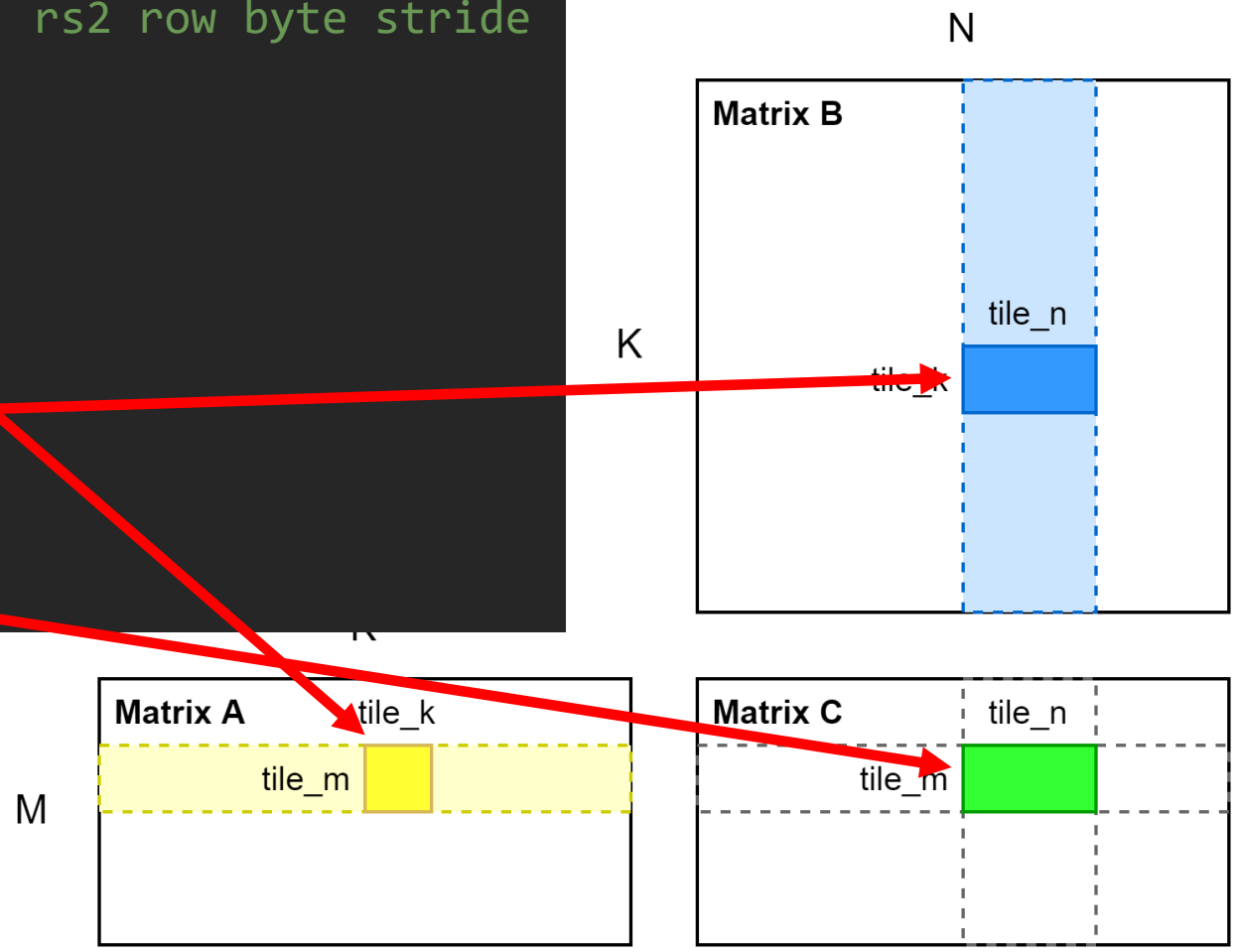
Load instructions

```
# vd destination, rs1 base address, rs2 row byte stride
# lmul / (eew/sew) rows or columns

# for left matrix, a
mlae<eew>.v vd, (rs1), rs2

# for right matrix, b
mlbe<eew>.v vd, (rs1), rs2

# for output matrix, c
mlce<eew>.v vd, (rs1), rs2
```



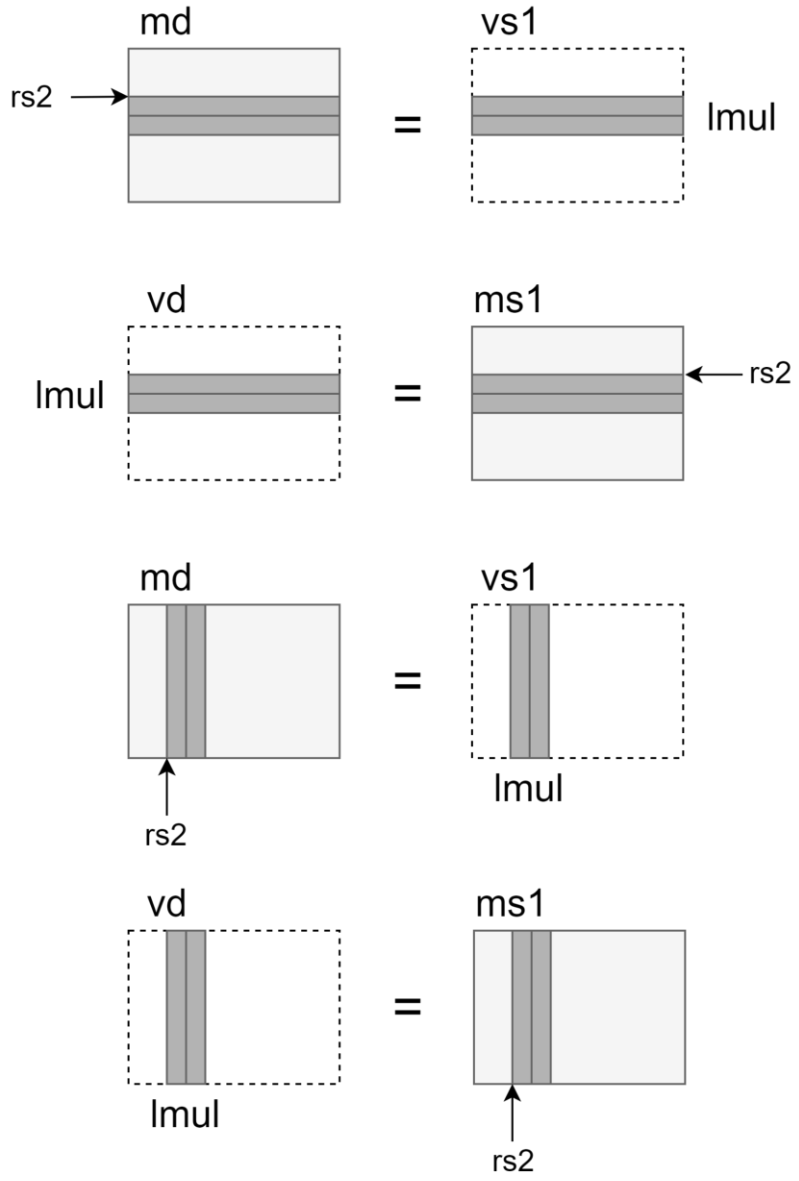
Store instructions

```
# vs3 store data, rs1 base address, rs2 row byte stride
# lmul / (eew/sew) rows or columns

# for left matrix, a
msae<eew>.v vs3, (rs1), rs2

# for right matrix, b
msbe<eew>.v vs3, (rs1), rs2

# for output matrix, c
msce<eew>.v vs3, (rs1), rs2
```



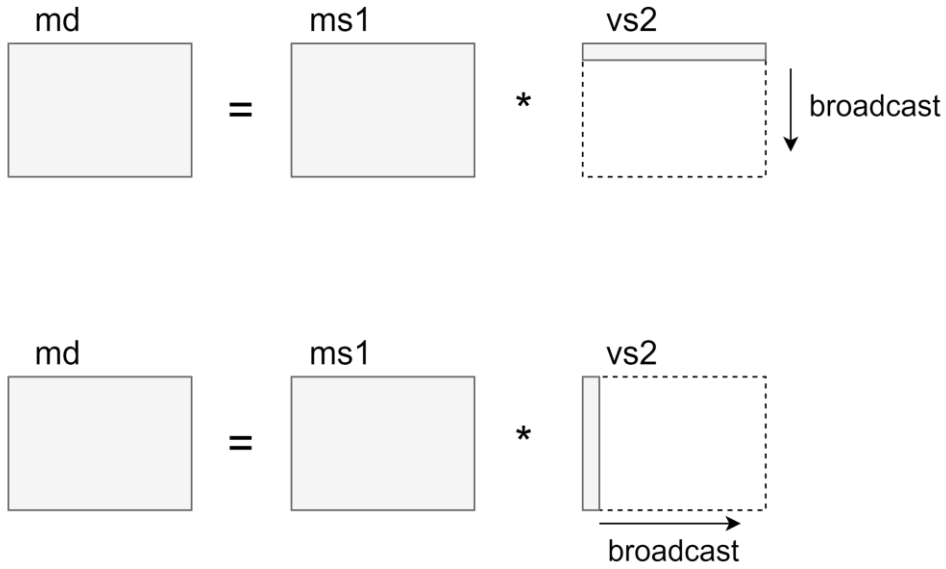
Data Move Instructions

```
# between matrix register rows and vector registers.

# vd[(i-rs2) * tile_n + j] = md[i, j], i = rs2 .. rs2 + lmul
mmvar.v.m  vd, ms1, rs2
mmvbr.v.m  vd, ms1, rs2
mmvcr.v.m  vd, ms1, rs2
# md[i, j] = vd[(i-rs2) * tile_n + j], i = rs2 .. rs2 + lmul
mmvar.m.v  md, vs1, rs2
mmvbr.m.v  md, vs1, rs2
mmvcr.m.v  md, vs1, rs2

# between matrix register columns and vector registers.

# vd[(j-rs2) * tile_m + i] = md[i, j], j = rs2 .. rs2 + lmul
mmvac.v.m  vd, ms1, rs2
mmvbc.v.m  vd, ms1, rs2
mmvcc.v.m  vd, ms1, rs2
# md[i, j] = vd[(j-rs2) * tile_m + i], j = rs2 .. rs2 + lmul
mmvac.m.v  md, vs1, rs2
mmvbc.m.v  md, vs1, rs2
mmvcc.m.v  md, vs1, rs2
```



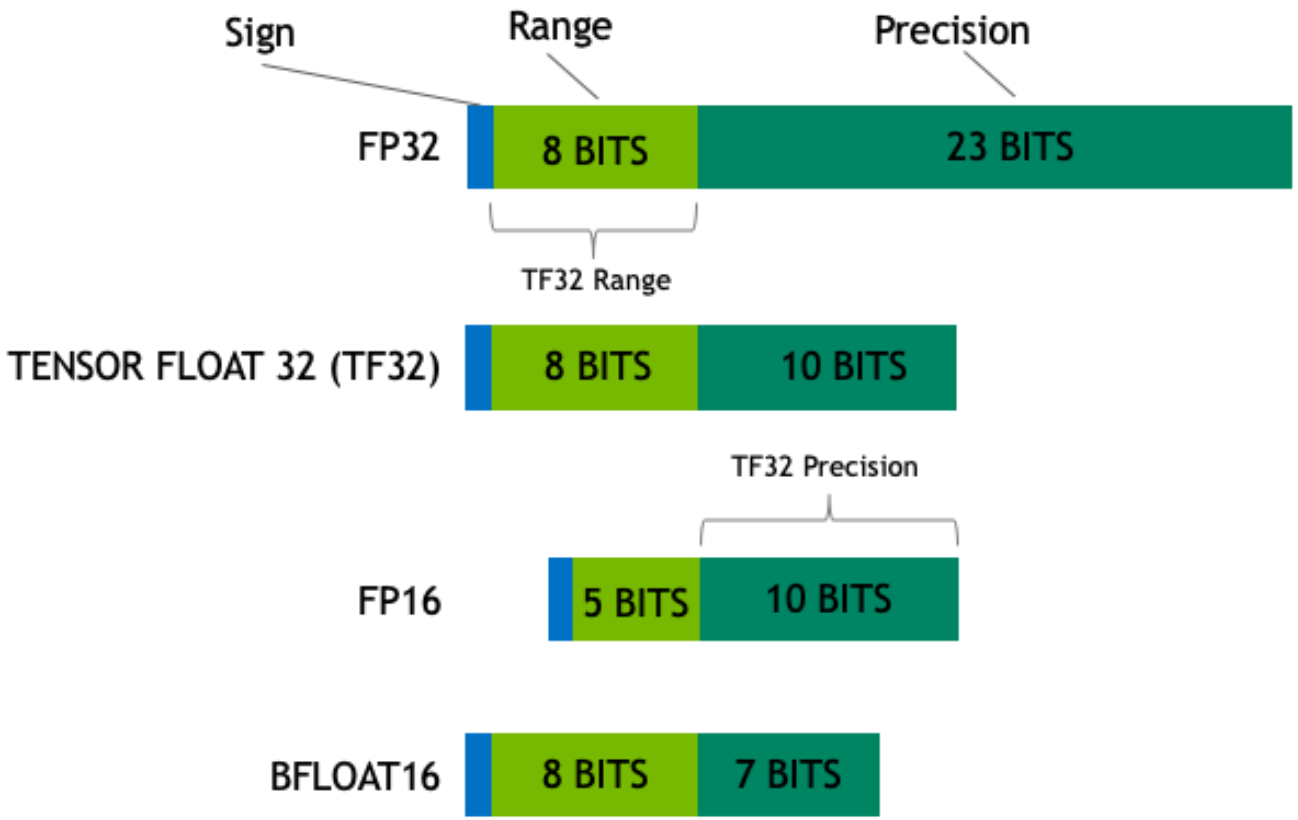
Matrix element-wise multiply

```
# matrix ele-wise multiply with a row of vector

# md[i,j] = ms1[i,j] * vs2[j]
memulcr.mv md, ms1, vs2
mwemulcr.mv md, ms1, vs2  # output double widen
mqemulcr.mv md, ms1, vs2  # output quadruple widen

mfemulcr.mv md, ms1, vs2
mfwemulcr.mv md, ms1, vs2  # output double widen

# matrix ele-wise multiply with a column of vector

# md[i,j] = ms1[i,j] * vs2[i]
memulcc.mv md, ms1, vs2
mwemulcc.mv md, ms1, vs2  # output double widen
mqemulcc.mv md, ms1, vs2  # output quadruple widen

mfemulcc.mv md, ms1, vs2
mfwemulcc.mv md, ms1, vs2  # output double widen
```



```
void fused_matmul_relu_float16(c, a, b, m, k, n) {
    msettype(e16);                              // use 16bit input matrix element
    for (i=0; i<m; i+=tile_m) {                 // loop at dim m with tiling
        tile_m = msettile_m(m-i);
        for (j=0; j<n; j+=tile_n) {             // loop at dim n with tiling
            tile_n = msettile_n(n-j);
            acc = mfemul_mf(acc, 0.f)           // clear acc reg
            for (s=0; s<k; s+=tile_k) {         // loop at dim k with tiling
                tile_k = msettile_k(k-s);
                tr1 = mlae16_m(&a[i][s]);       // load left matrix a
                tr2 = mlbe16_m(&b[s][j]);       // load right matrix b
                acc = mfwma_mm(tr1, tr2);       // tiled matrix multiply
            }

            acc = mfncvt_f_fw_m(acc);           // convert widen result to normal width

            for (s=0; s<tile_m; s+=rows) {
                rows = min(tile_m - s, 8*vlenb/rlenb); // max rows could move into 8 vregs

                vsetvl(tile_n*rows, e16, m8);
                v1 = mmvcr_v_m(acc, s);         // move acc rows to vreg
                v1 = vfmax_vf(0.f, v1);         // vfmax.vf for relu

                msce16_v(v1, &c[i+s][j], n);    // store output tile slices
            }
        }
    }
}
```

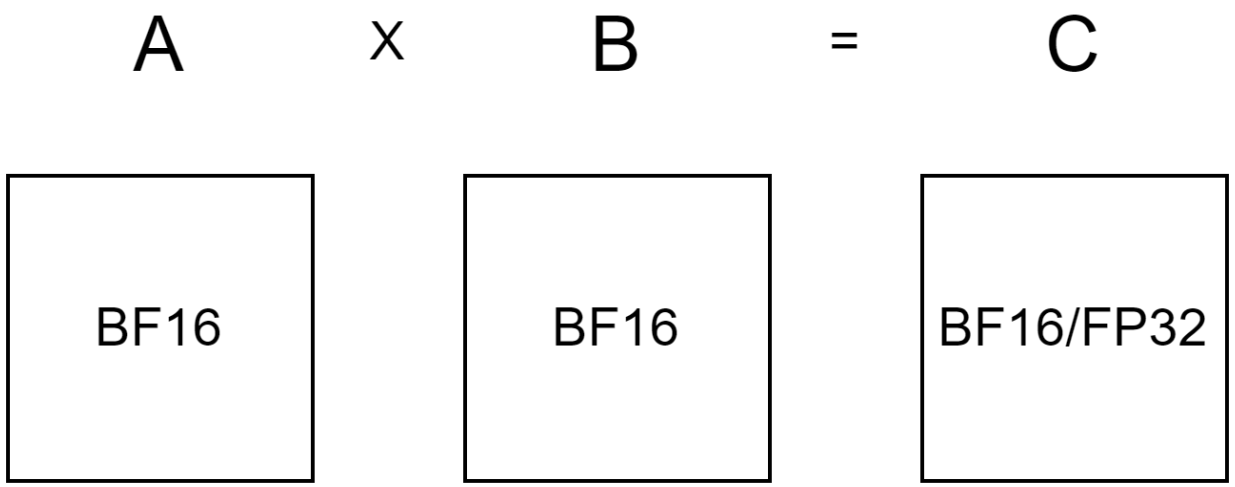
BF16, TF32 and FP16, FP32



- https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/

Zmbf16: Matrix Bfloat16(BF16) Extension

- Allows to use BF16 format as the data type of input/output elements.



Zmbf16: Matrix Bfloat16(BF16) Extension

- Add a bit mtype[4] in mtype register.

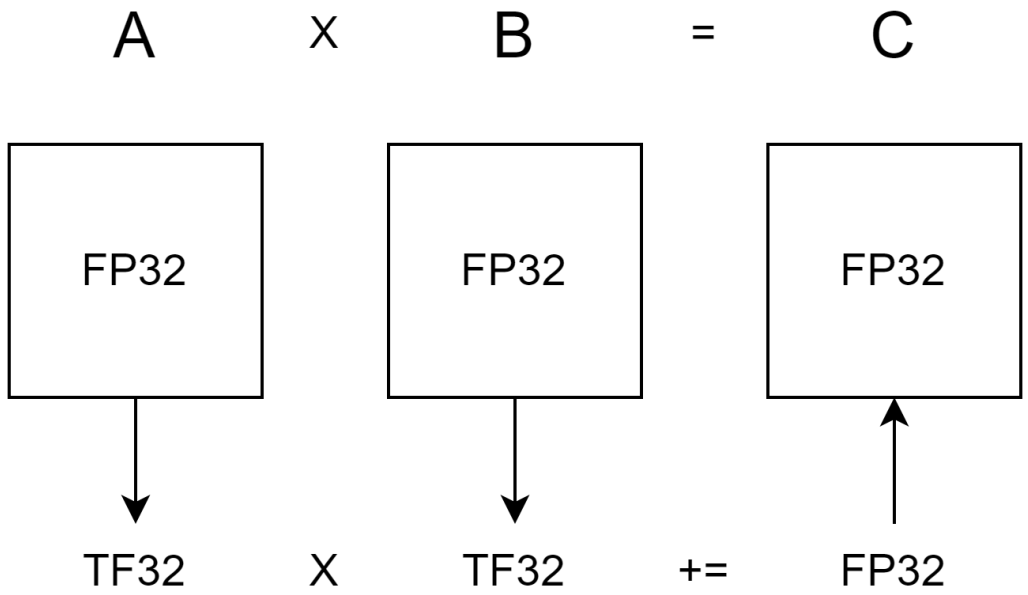
Table 5. mtype register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:6	0	Reserved if non-zero
5	**mtf32**	**Enable TF32 FMA for matrix multiplication**
4	mbf16	Use bfloat16 input format

```
Suggested bf16 assembler name used for msettypei mtypei immediate

    bf16  # Use BF16 format

Examples:

    msettypei t0, e16, bf16         # SEW = 16, use BF16 as input matrix element
```

Zmtf32: Matrix TensorFloat-32(TF32) Extension

- TF32 implementations are designed to achieve better performance on matrix multiplications and convolutions
- by rounding input Float32 data to have 10 bits of mantissa, and accumulating results with FP32 precision, maintaining FP32 dynamic range.



Zmtf32: Matrix TensorFloat-32(TF32) Extension

- Add a bit mtype[5] in mtype register.

Table 5. `mtype` register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:6	0	Reserved if non-zero
5	**mtf32**	**Enable TF32 FMA for matrix multiplication**
4	mbf16	Use bfloat16 input format

```
Suggested tf32 assembler name used for msettypei mtypei immediate

    tf32  # enable TF32 FMA

Examples:

    msettypei t0, e32, tf32         # SEW = 32, enable TF32 FMA
```

Agenda

- Introduction
- Programmer's Model
- Instructions
- Standard Matrix Extensions
- Open Source Projects
- Future Works

https://github.com/riscv-stc/riscv-matrix-project

- riscv-matrix-spec: CC-BY-4.0
  - The matrix extension proposal
- llvm-project: Apache License 2.0
  - llvm toolchain to support matrix extension proposal
- riscv-isa-sim: upstream modified BSD
  - Spike ISS to support matrix extension proposal
- chipyard: upstream BSD-3-Clause
  - Chipyard project to support matrix extension proposal with BOOM Core
- riscv-pvp-matrix: BSD-3-Clause
  - RISC-V Matrix Extension ISA Verification using RISC-V PVP
- riscv-dnn: BSD-3-Clause
  - A small DNN library for RISC-V, using RISC-V Vector and Matrix extensions

Agenda

- Introduction
- Programmer's Model
- Instructions
- Standard Matrix Extensions
- Open Source Projects
- Future Works

Future works

- Im2col Matrix Multiplication Extension
  - perform the im2col operation on-the-fly, by the new load instructions
  - to support convolution operation with matrix multiplication
- Chipyard simulation & verification & prototype documents
- Reference performance data of Chipyard simulation

Questions ?