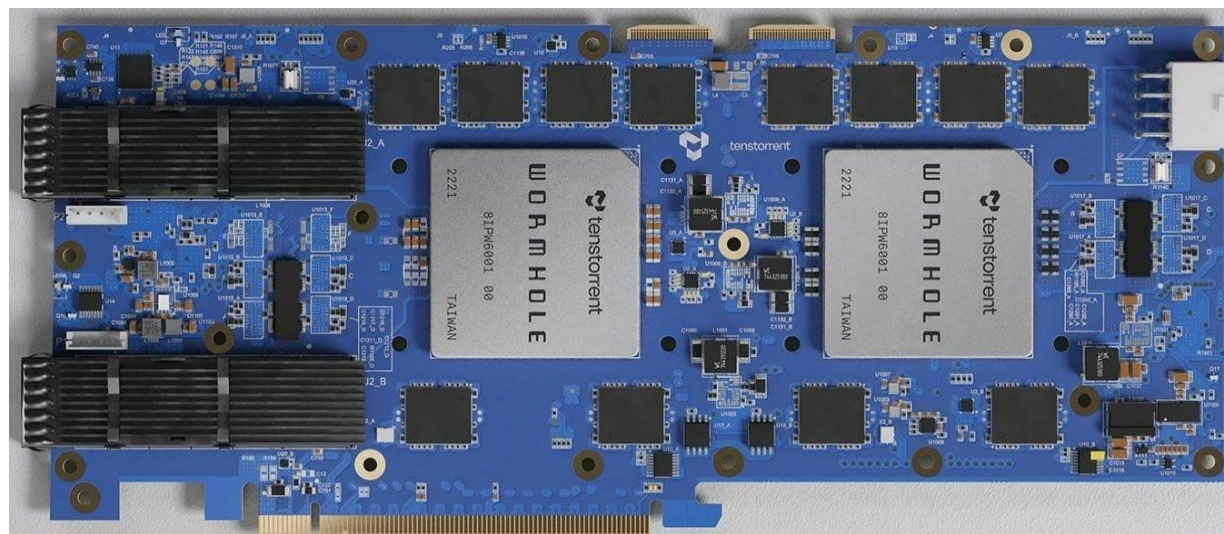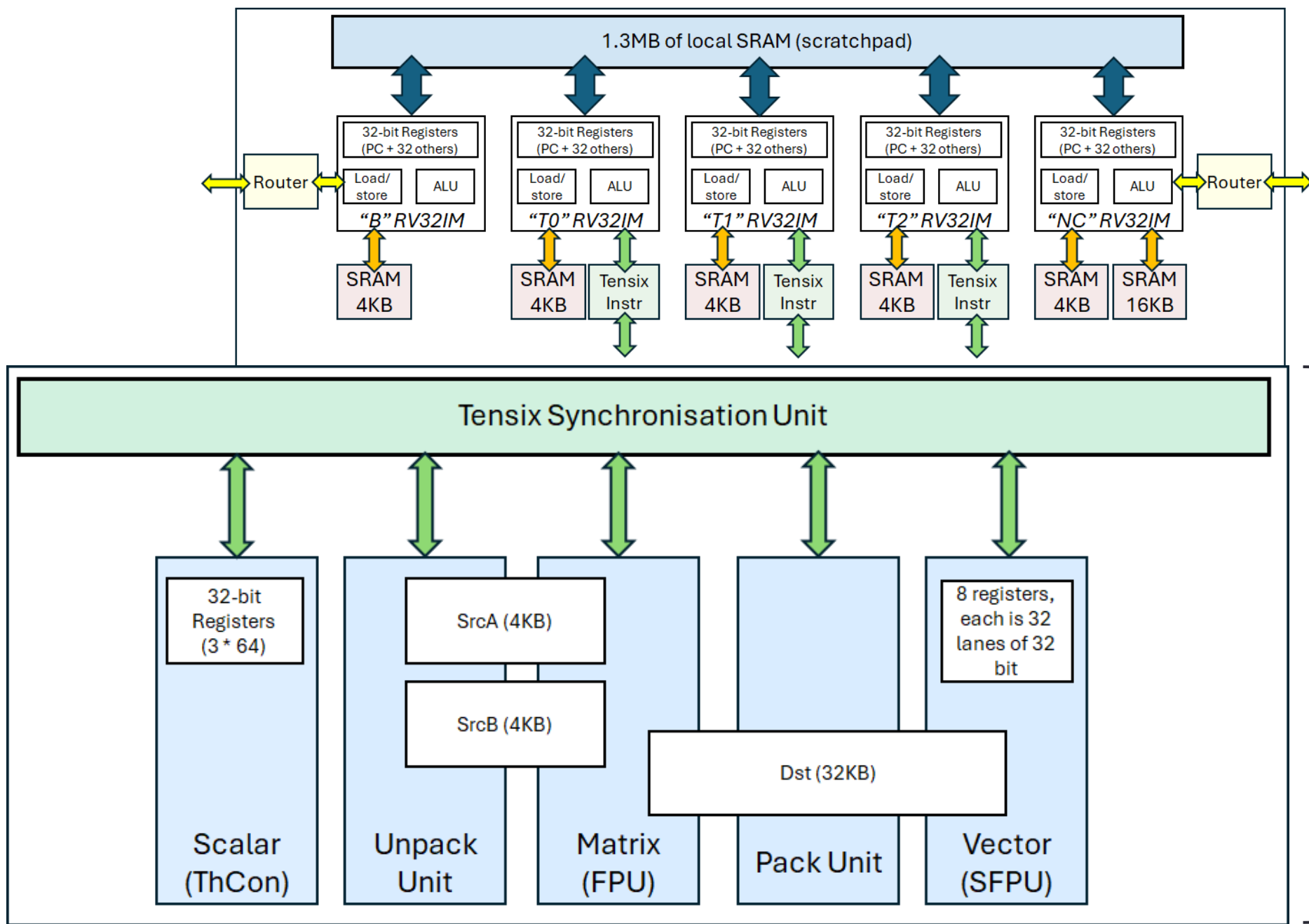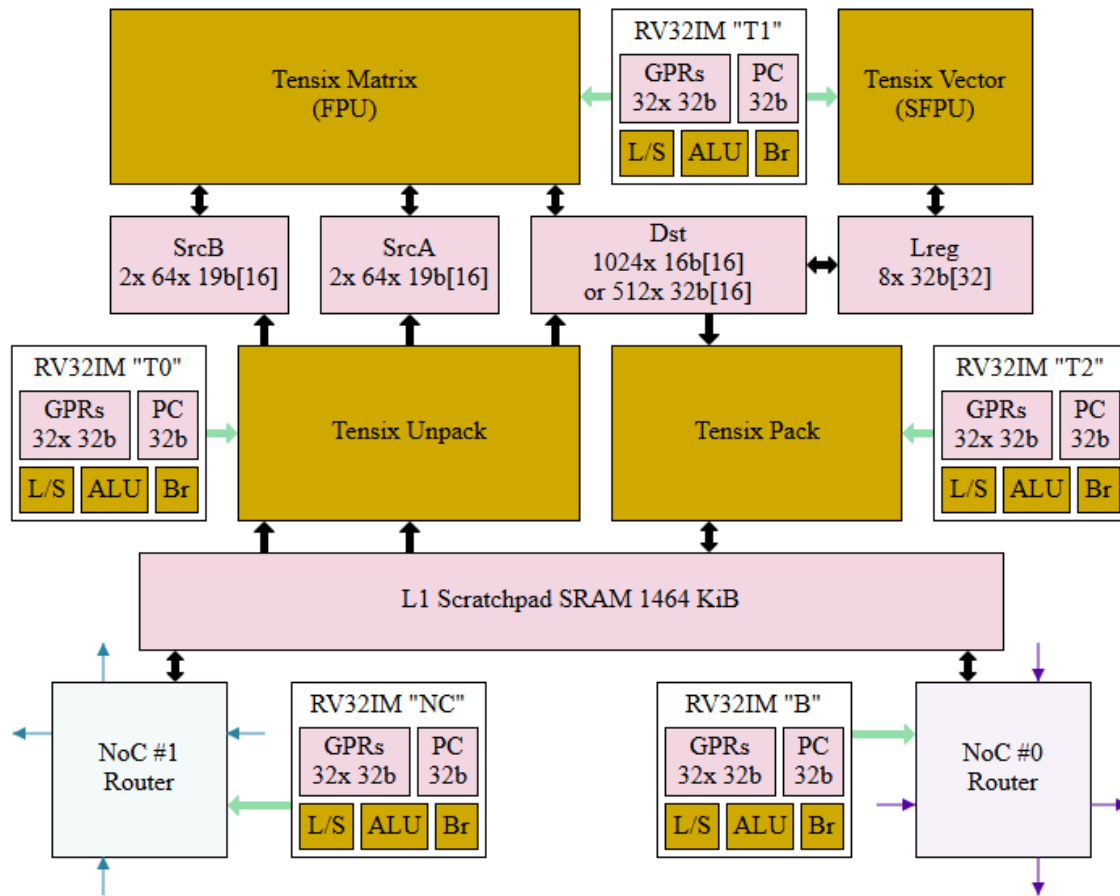# Overview of TT-Metalium SDK: Compute

The compute engine has a scalar, matrix and vector unit
- SrcA and SrcB are input registers
- Dst is an output register (and input register for Vector unit too)

# A more accurate diagram….



*From Pete Cawley's blog at [https://www.corsix.org/content/tt-wh-part7](https://www.corsix.org/content/tt-wh-part7)*

- The matrix unit can perform up to 4.096 TFLOP/s
- The vector unit can perform 32 FP32 maths operations per cycle

- This is FP32(ish)
  - The matrix unit makes several sacrifices to IEEE compliance to achieve this performance
    - Maximum of 19 bits for each element, so supports a maximum of TF32
  - The vector unit provides full FP32 support, so for HPC workloads this is probably the one we would use
    - However, performance is slower

# Data supported by the matrix unit (FPU)

- Inputs to the matrix unit (srcA and srcB) are two banks of 64 rows by 16 columns of 19-bit data

  - 1024 elements computed with by the FPU, the majority of compute FPU operations have a latency of five cycles but are pipelined and can be issued each cycle
  - Matrix multiplication can deliver up to 4.096 TFLOP/s and element wide operations 0.256 TFLOP/s
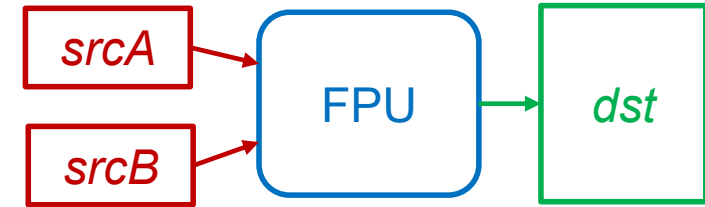
| Dst data type | += | SrcB data type | @ | SrcA data type |
|---|---|---|---|---|
| 8x16 matrix of either FP32 or BF16 | += | 8x16 matrix of either TF32 or BF16 | @ | 16x16 matrix of either TF32 (†) or BF16 |
| 8x16 matrix of either FP32 or FP16 | += | 8x16 matrix of FP16 | @ | 16x16 matrix of FP16 (†) |
| 8x16 matrix of integer "32" | += | 8x16 matrix of integer "8" | @ | 16x16 matrix of integer "8" (‡) |

From https://github.com/tenstorrent/tt-isa-documentation/blob/main/WormholeB0/TensixTile/TensixCoprocessor/SrcASrcB.md

- Whilst the FPU provides the potential for performance, the data types that are supported are rather limited

# Issuing compute operations to matrix unit
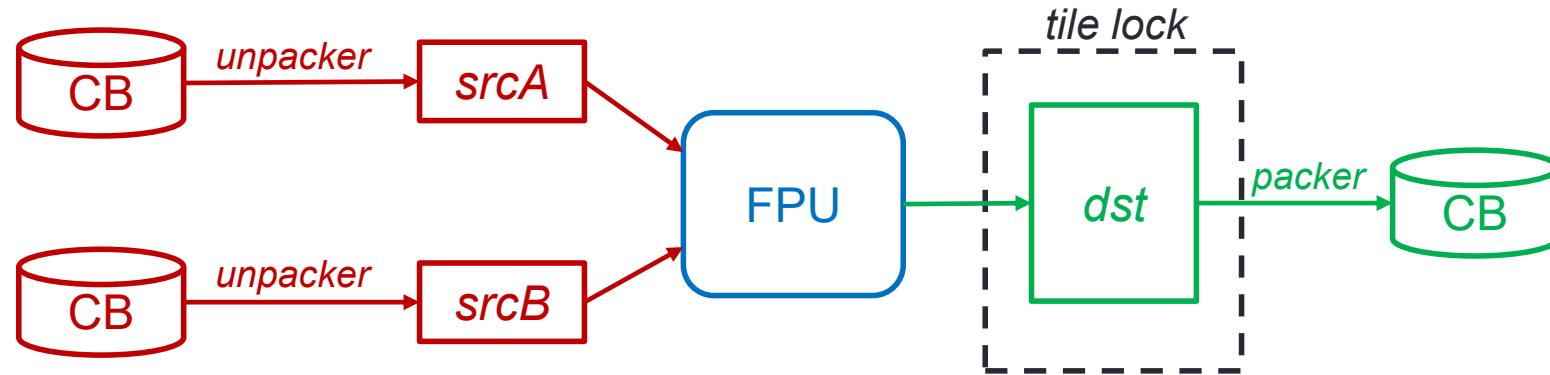


**To get input data in**

- Wait for two CBs (LHS and RHS) to be available via *cb_wait_front* API call

**To compute**

- Acquire exclusive compute lock on DST (target) registers using *tile_regs_aqcuire* API call

- Issue corresponding matrix API call such as *add_tiles, sub_tiles, mul_tiles* with CB index as input

- Release exclusive compute lock on DST (target) registers using *tile_regs_commit* API call

**To get results out**

- Acquire exclusive pack lock on DST (target) registers using *tile_regs_wait* API call

- Copy results from dst register to target CB via *pack_tile* API call

- Release exclusive pack lock on DST (target) registers using *tile_regs_release* API call

# Data supported by the vector unit (SFPU)

- Inputs to the vector unit are from the *dst* register
  - 1024 rows of 16 columns of 16-bit data, or 512 rows of 16 columns of 32-bit data
  - BF16, FP16, FP32, int8, int16 and int32 are supported
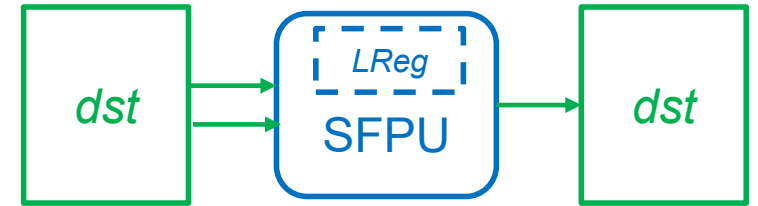


- The vector unit has 32 lanes, each of 32 bit
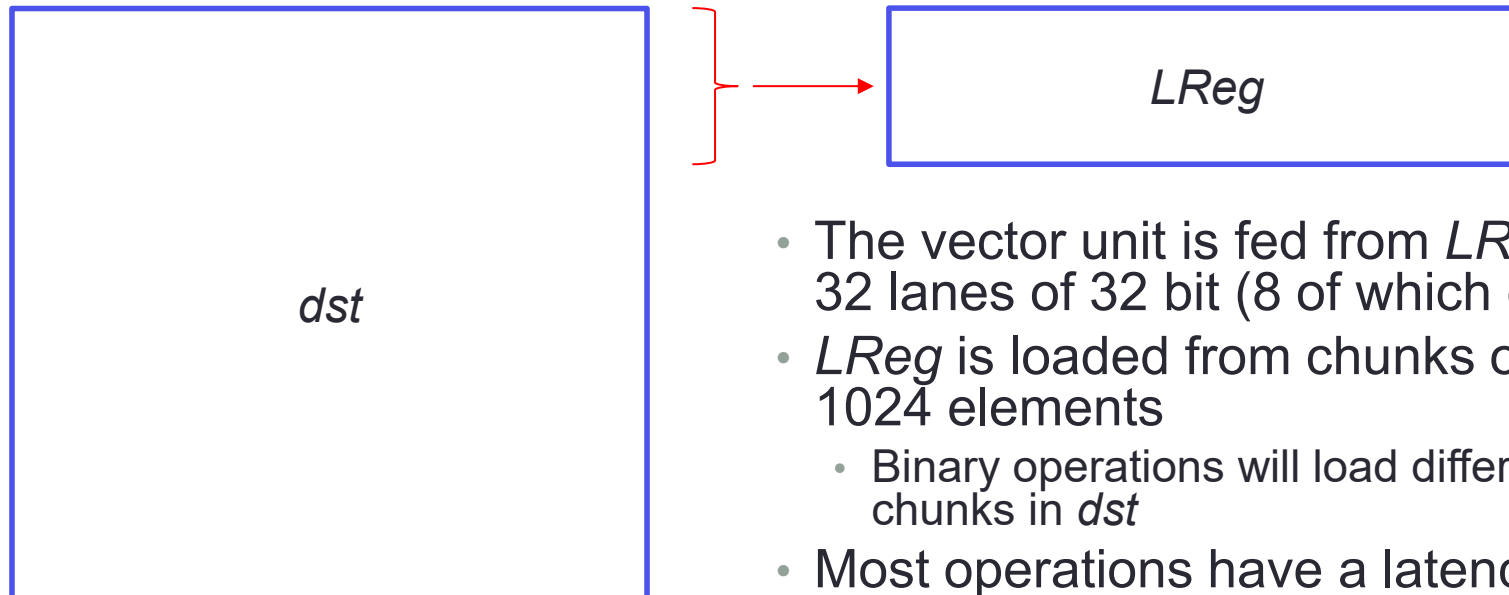


- The vector unit is fed from *LReg*, which contains 17 times, 32 lanes of 32 bit (8 of which can be used for inputs)
- *LReg* is loaded from chunks of *dst* , from a maximum of 1024 elements
  - Binary operations will load different lanes of *LReg* from different chunks in *dst*
- Most operations have a latency of 2 cycles and can be issued each cycle – but only work on 32 elements

# Issuing compute operations to vector unit
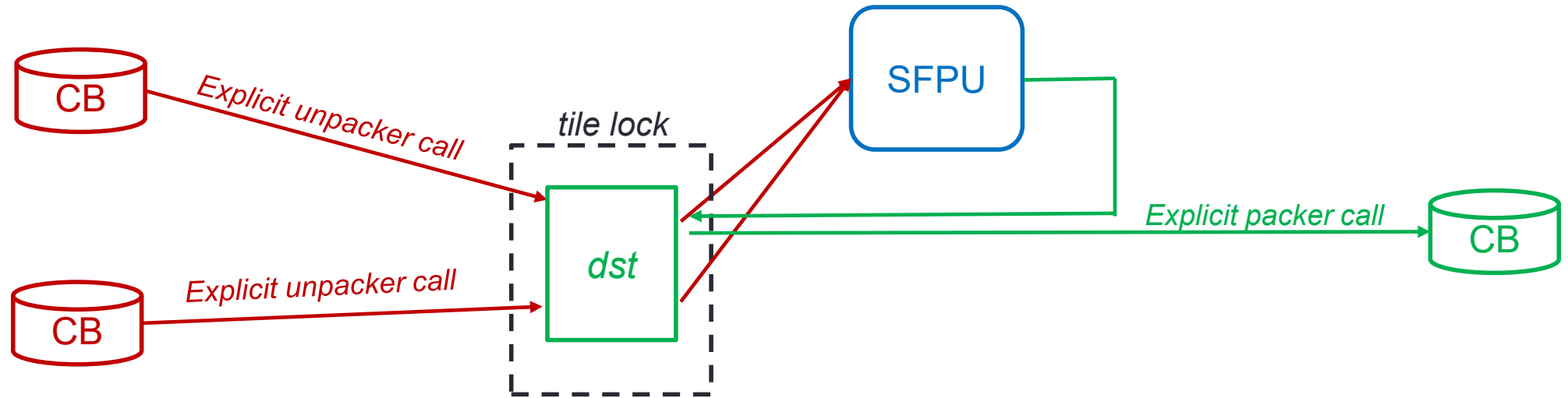


## To get input data in

- Wait for two CBs (LHS and RHS) to be available via *cb_wait_front* API call
- Acquire exclusive pack lock on DST (target) registers using *tile_regs_wait* API call
- Copy both input tiles into *dst* register using segment index
- Release exclusive pack lock on DST (target) registers using *tile_regs_release* API call

## To compute

- Acquire exclusive compute lock on DST (target) registers using *tile_regs_aqcuire* API call
- Issue corresponding vector API call such as *add_binary_tile, sub_binary_tile* with segment index determining inputs (first input overwritten with results)
- Release exclusive compute lock on DST (target) registers using *tile_regs_commit* API call

## To get results out

- Acquire exclusive pack lock on DST (target) registers using *tile_regs_wait* API call
- Copy results from dst register to target CB via *pack_tile* API call
- Release exclusive pack lock on DST (target) registers using *tile_regs_release* API call

# The key points

- Inputs are CBs and the output is a CB

- *dst* register is split into 16 segments
  - Matters more when using the vector unit

- Interaction with *dst* is always explicit and you need to acquire locks on the *dst* register as this coordinates instructions from the pack, compute and unpack RISC-V cores
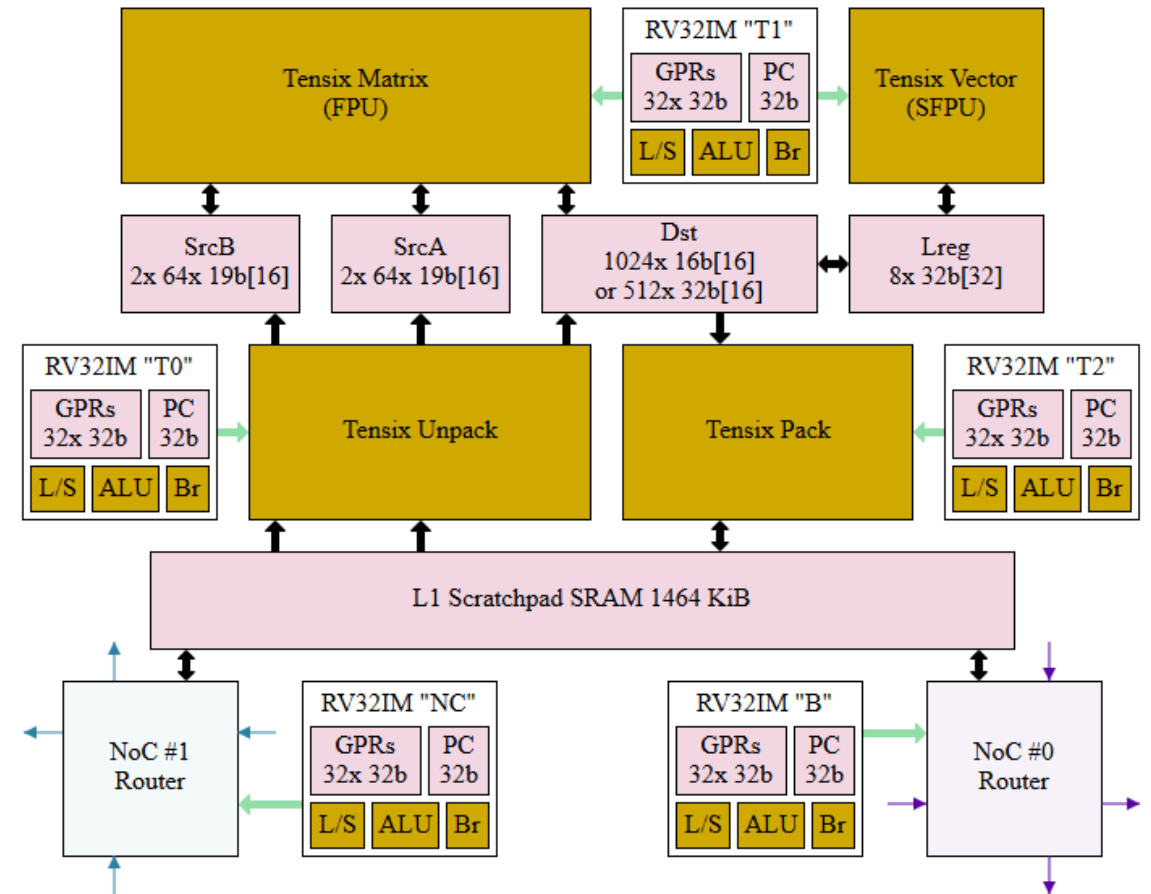
# Initialisation

- The compute engine must be initialised, taking the input and output circular buffers as arguments
    - This configures the unpacker, packer, and FPU for the specific operation being performed.
    - Re-initialization is not required for repeated operations with the same source, destination, and data type parameters.

Image from https://docs.tenstorrent.com/tt-metal/latest/tt-metalium/tt_metal/advanced_topics

- In practical four (FPU) you will see we use *binary_op_init_common* (with input and output CBs) and *add_tiles_init* (with input CBs)
- In practical five (SFPU) we use *init_sfpu* (with an input and output CB) and *add_int_tile_init* (with no arguments)
    - *init_sfpu* sets up the packer and unpackers, but there are limits (assumes all CBs the same type and doesn't support FP32) and *copy_tile_init* can be used instead to get round these
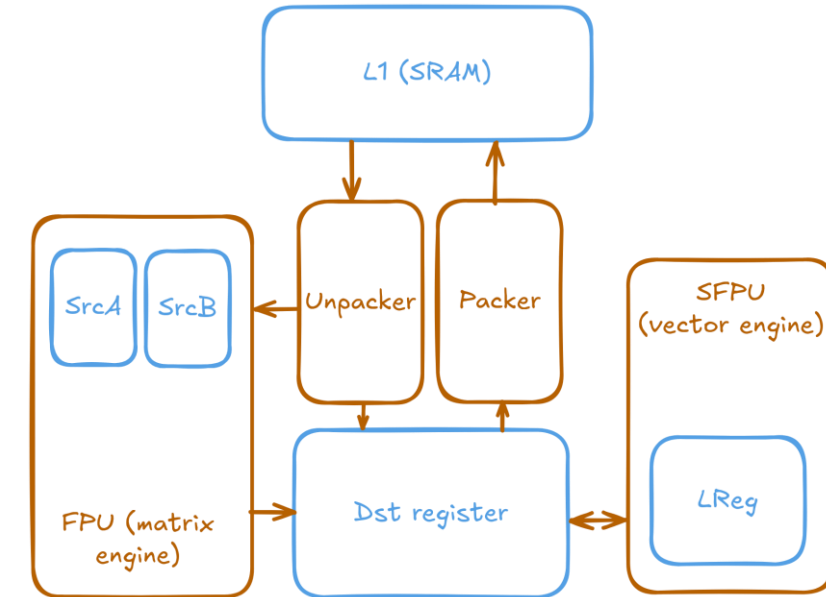
# Most common maths calls

## Matrix unit

- add_tiles
- sub_tiles
- mul_tiles
- matmul_tiles
- reduce_tile
- transpose_wh_tile

*This also explains why the vector unit consumes from dst, as a common ML use-case is to execute with the matrix unit and then run another operation on results via the vector unit*

## Vector unit

- add_binary_tile
- sub_binary_tile
- mul_binary_tile

*And integer variants*

- abs_tile
- exp_tile
- Isinf_tile
- Isfinite_tile
- Isnan_tile
- sqrt_tile
- square_time
- tan_tile
- sin_tile
- cos_tile
- atan_tile
- acos_tile
- asin_tile

- ltz_tile
- eqz_tile
- lez_tile
- gtz_tile
- neq_tile
- gez_tile
- unary_ne_tile
- unary_gt_tile
- unary_lt_tile
- unary_max_tile
- unary_min_tile

*This column operations are for comparison*

# Three compute cores…
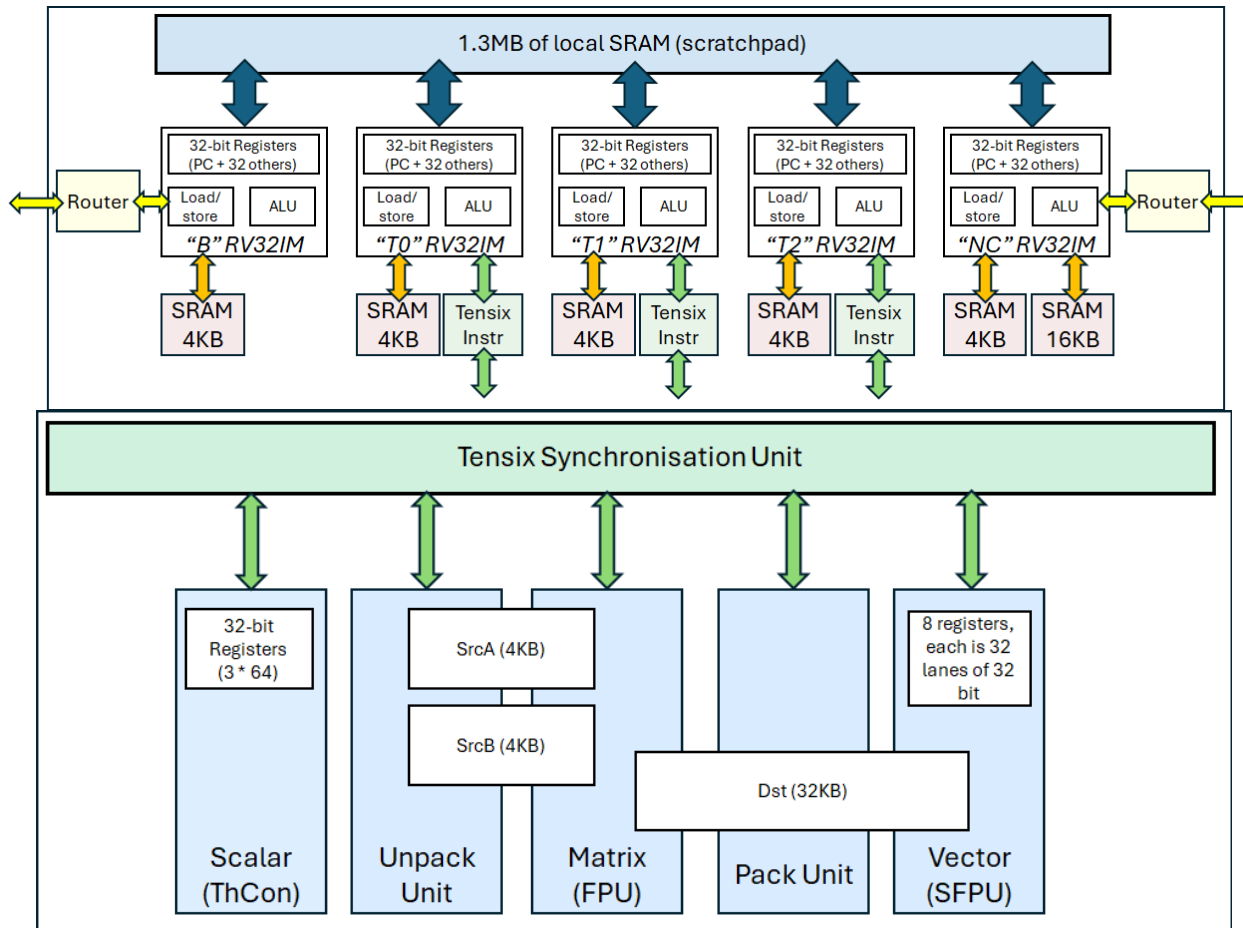


- Each Tensix unit has three RISC-V baby cores for compute
  - Unpacker drives the unpack unit
  - Maths drives FPU, SFPU, ThCon
  - Packer drives the pack unit

- Programmer's compute kernel is launched on all three cores which execute it concurrently
  - In the Metalium API there are explicit sections for different cores, where one (or more) cores will execute some code and additional synchronisation
    - But we don't really need to worry about this, however, it explains why there are locks on the dst register to avoid conflict between the math and pack cores

# Tiling data to drive compute

- We have talked about bringing the FPU into play but the registers are only of a certain size
  - srcA and srcB contain a maximum of 1024 elements and similar if you use the SFPU

- Therefore need to tile data across chunks
  - Tenstorrent use the terminology *tile* due to the architecture being designed for matrix multiplications, and chunk would be better as a tile can be 1D

- Practical three will explore how to do this, before using the matrix multiplication engine in practical four and vector unit in practical five to perform the compute

| Tile 1 | Tile 2 |
|--------|--------|
| Tile 3 | Tile 4 |

| Tile 1 |
|--------|
| Tile 2 |
| Tile 3 |
| Tile 4 |
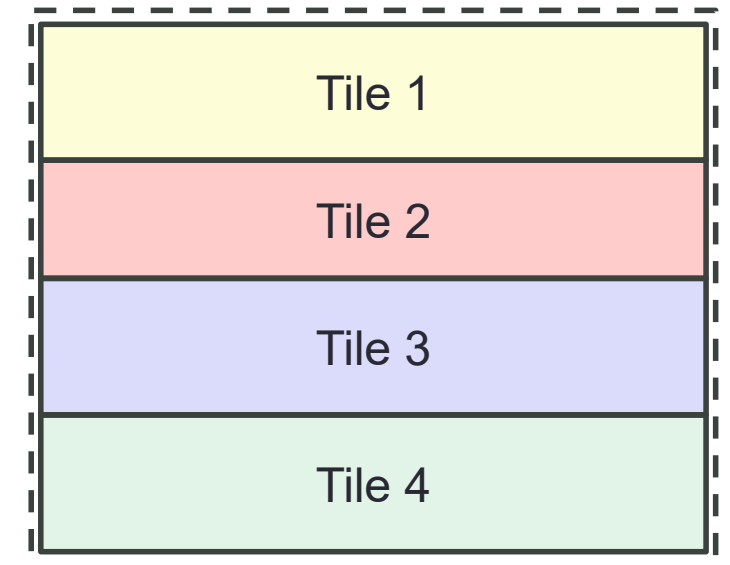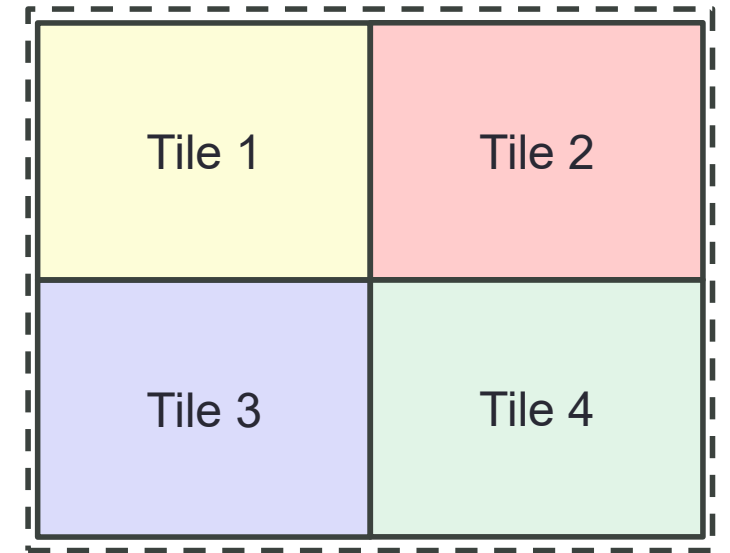
# What now: Practicals 3,4 & 5 and more information

- We are now going to move onto looking at practicals 3, 4, and 5
  - Practical three chunks up data (into tiles) and operates on each chunk in tern. This is in preparation to use the FPU/SFPU as they have a maximum of 1024 elements at a time
  - Practical four explores using the FPU (matrix unit) to undertake element wise addition. We sacrifice accuracy (can only use int8) but get performance.
  - Practical five explores using the SFPU (vector unit) to undertake element wise addition. We sacrifice performance but get accuracy (can use int32).


- There is API based documentation at https://docs.tenstorrent.com/tt-metal/latest/tt-metalium/tt_metal/apis/index.html
  - But this is somewhat incomplete, the API header files at https://github.com/tenstorrent/tt-metal/tree/main/tt_metal/include/compute_kernel_api tend to be more useful


- Detailed architecture documentation can be found at https://github.com/tenstorrent/tt-isa-documentation/tree/main/WormholeB0