Matrix Computation Acceleration in the Presence of Data Layout Conversion (work-in-progress)

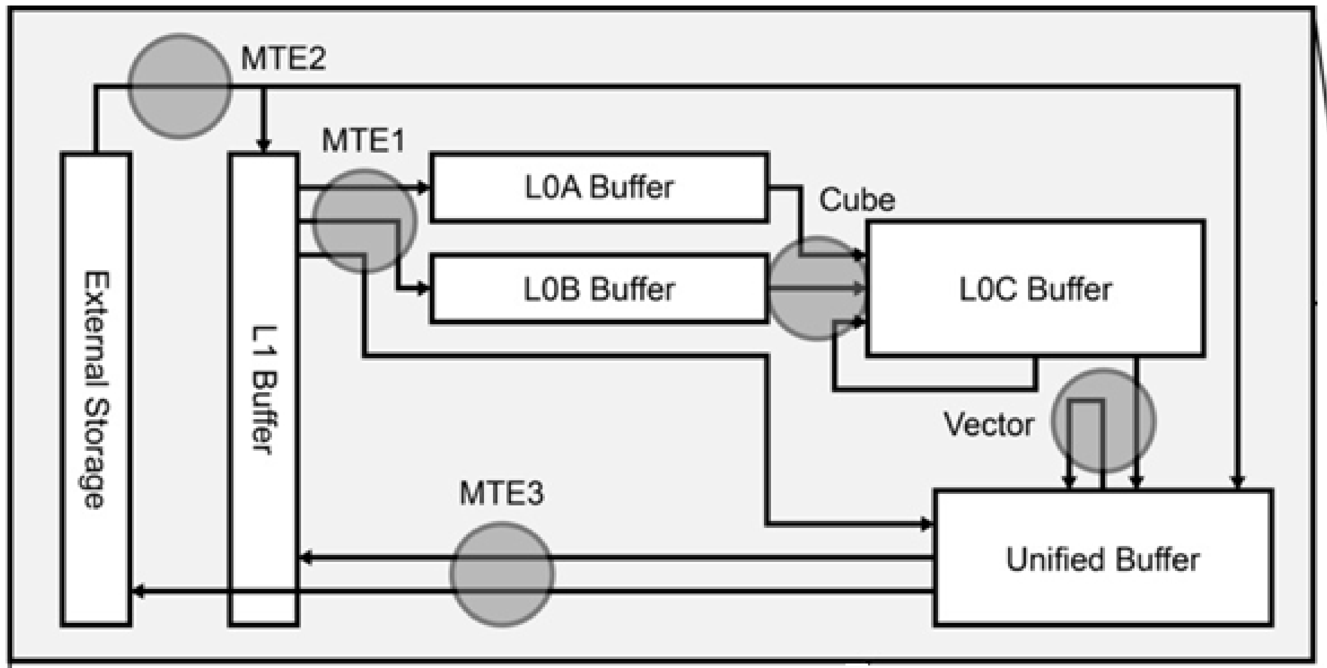
LATHC'23 Workshop

Chenchen Tang, Frank Gao, *Kai-Ting Amy Wang

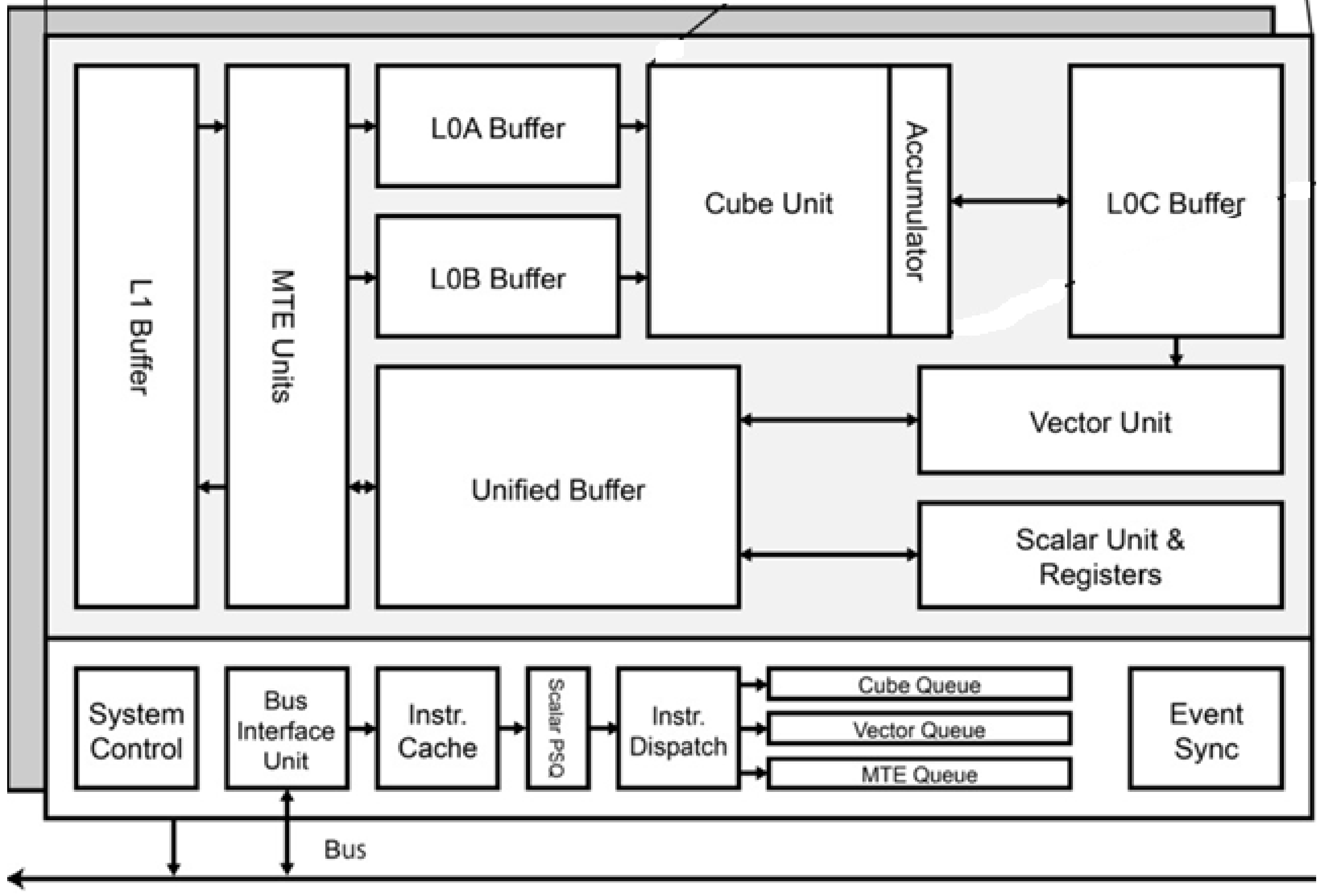
Feb 26th, 2023

HUAWEI

Background, DaVinci Architecture



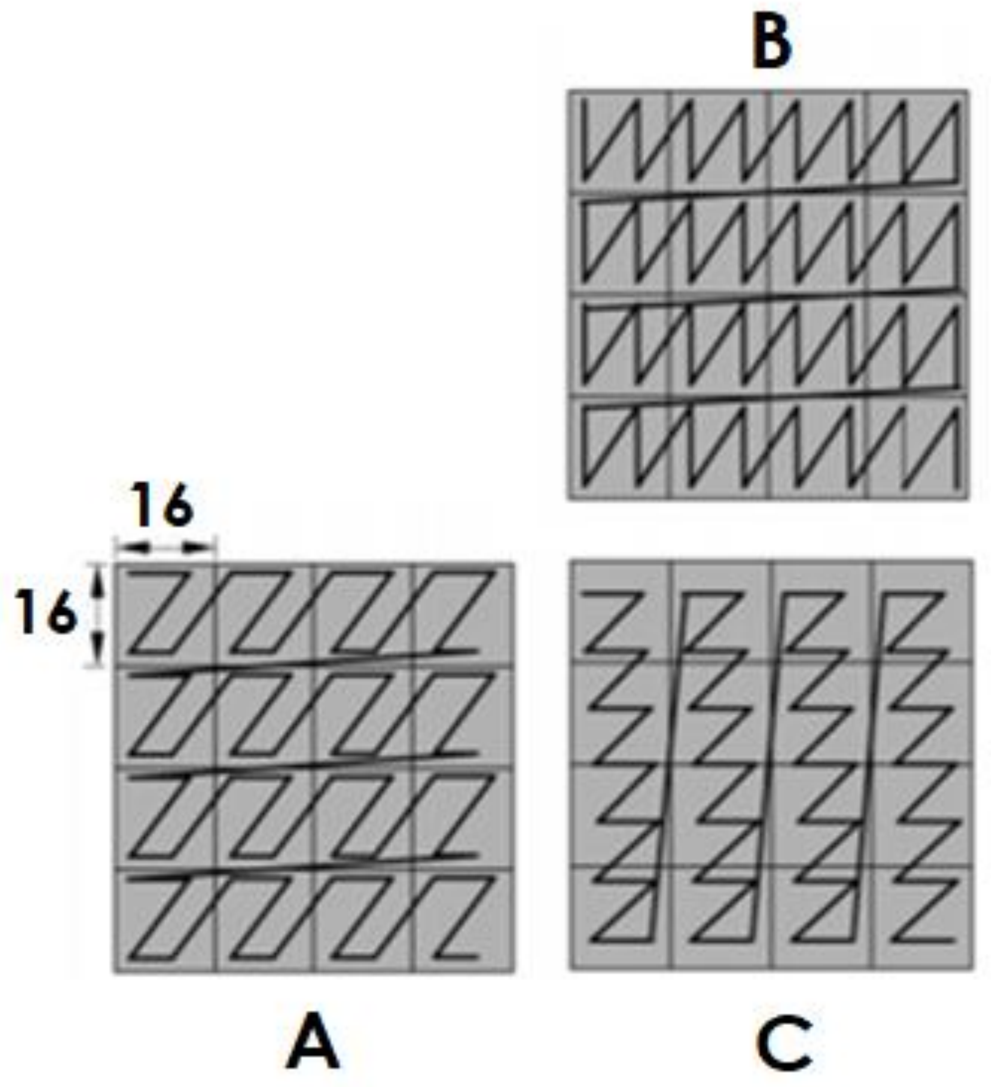
(b) Logic Diagram of DaVinci Storage Units and Data Path Control



- ❑ Scalar Unit, Vector Unit, Cube Unit
- ❑ 5 memory on-chip buffers
- ❑ 3 Memory Transfer Engines (MTEs)

Performance modeling on DaVinci AI core, Journal of Parallel and Distributed Computing, 2023
https://www.sciencedirect.com/science/article/pii/S074373152300014X

Background, DaVinci Architecture, GEMM

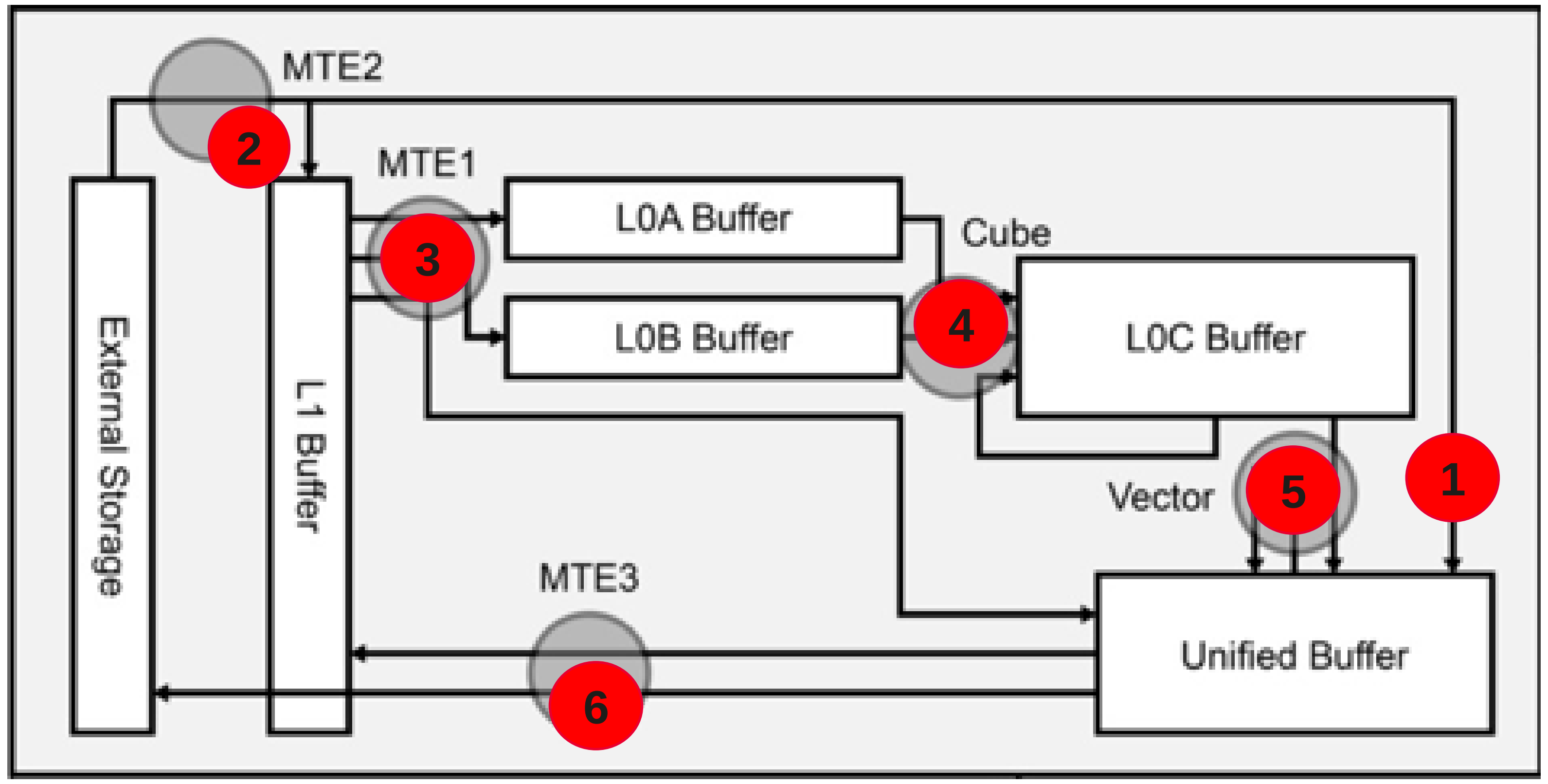


- **Fractal Layouts:**
  - A (zZ), B (nZ), C (zN)

GEMM Formula: C = αC + βAB

Dataflow (assume α = 1.0, β = 1.0) :

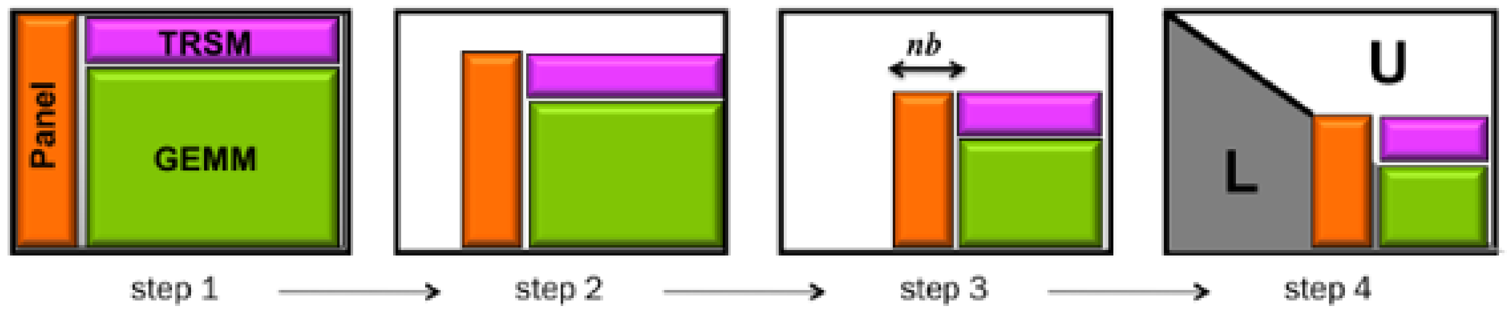
1. Copy initial Matrix C from GM to UB (MTE2).
2. Copy data of Matrix A and B from GM to L1A and L1B (MTE2).
3. Load data of Matrix A and B from L1A and L1B to L0A and L0B (MTE1).
4. Cube multiplies data from L0A and L0B; Stores results to L0C (Cube).
5. Copy results from L0C to UB (Vector).
6. Copy results from UB to GM (MTE3).



Motivating Use case
LU Factorization

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{44} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21}u_{11} & l_{21}u_{12}+u_{22} & l_{21}u_{13}+u_{23} & l_{21}u_{14}+u_{24} \\ l_{31}u_{11} & l_{31}u_{12}+l_{32}u_{22} & l_{31}u_{13}+l_{32}u_{23}+u_{33} & l_{31}u_{14}+l_{32}u_{24}+u_{34} \\ l_{41}u_{11} & l_{41}u_{12}+l_{42}u_{22} & l_{41}u_{13}+l_{42}u_{23}+l_{43}u_{33} & l_{41}u_{14}+l_{42}u_{24}+l_{43}u_{34}+u_{44} \end{bmatrix}$$



Many GEMMs with small K and large M, N, causing a large C

- Performing post layout conversion on C can be expensive.

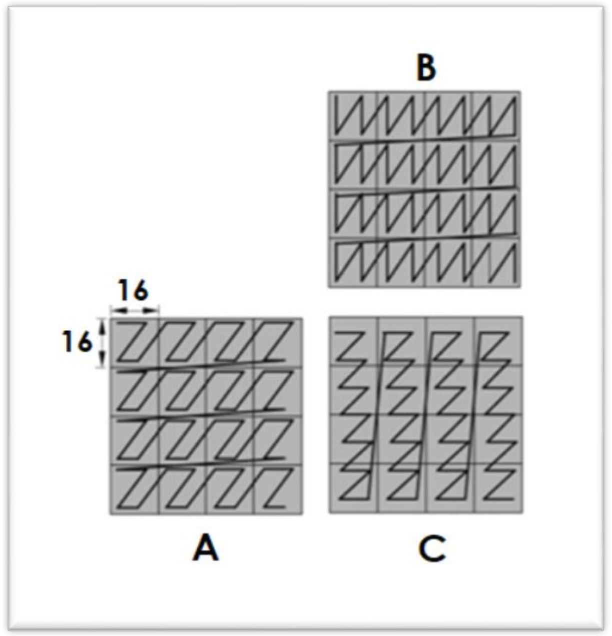
Problem & Existing Solution:

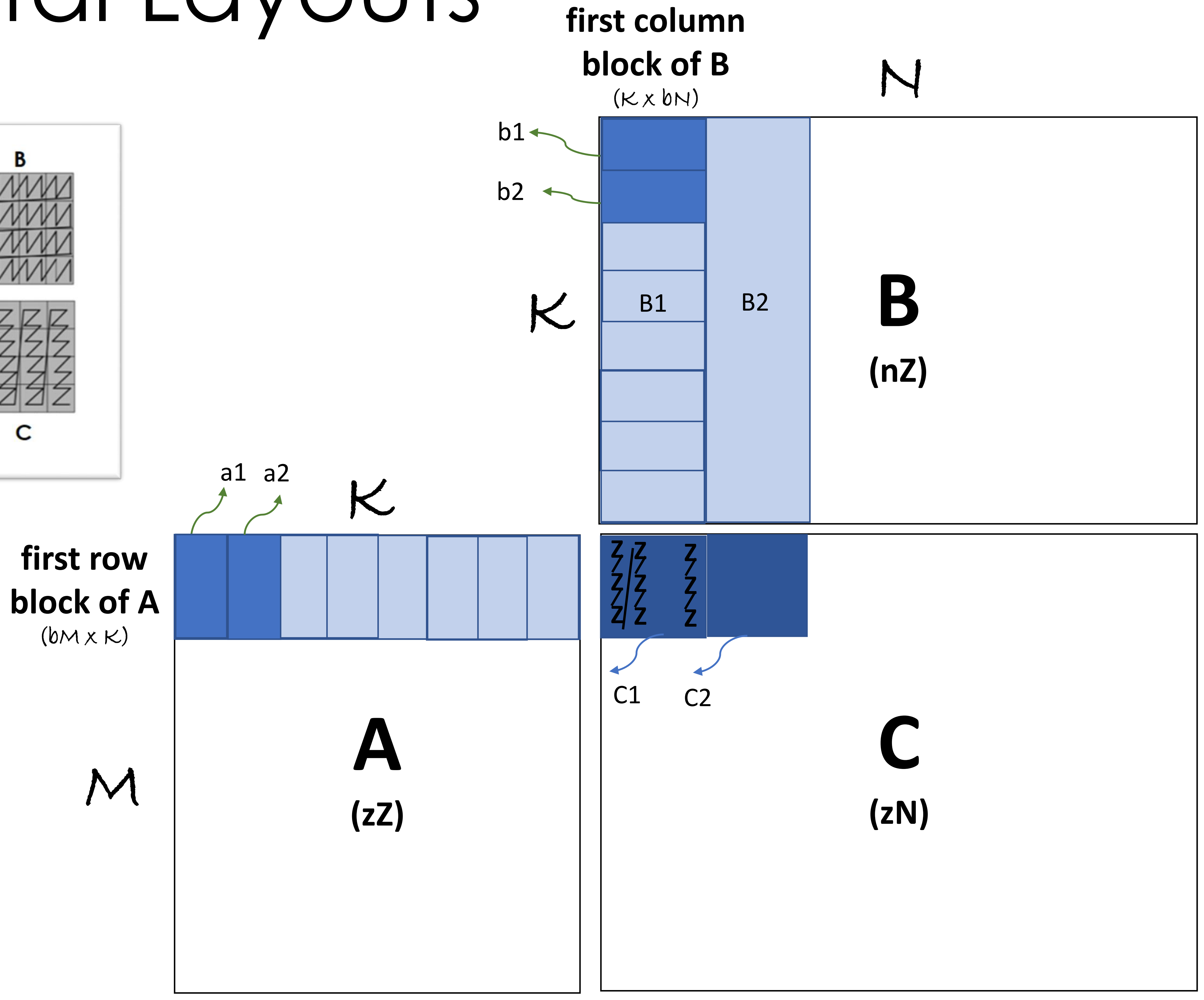
- Difficult to keep data of each LU step in fractal layouts since computing Linv/Uinv requires row-wise operations.
- Perform pre/post-layout conversions before/after each LU step.

Research Question:

- Can we **combine** data layout conversions with data movement operations (i.e. DMAs) **efficiently**?

Fractal Layouts

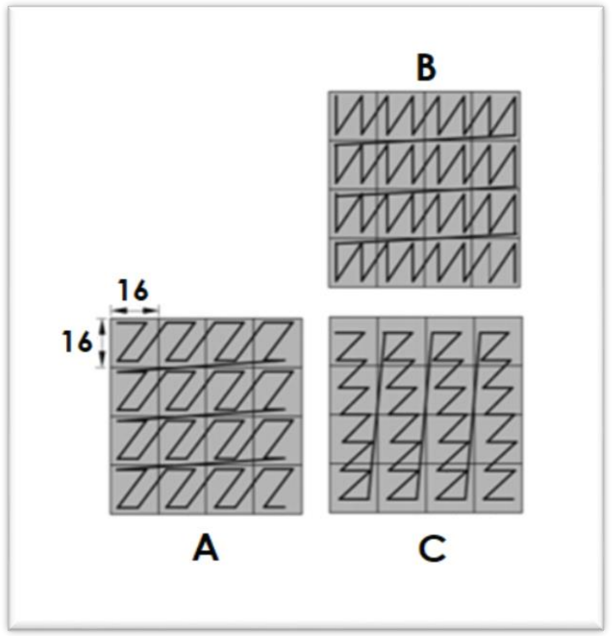


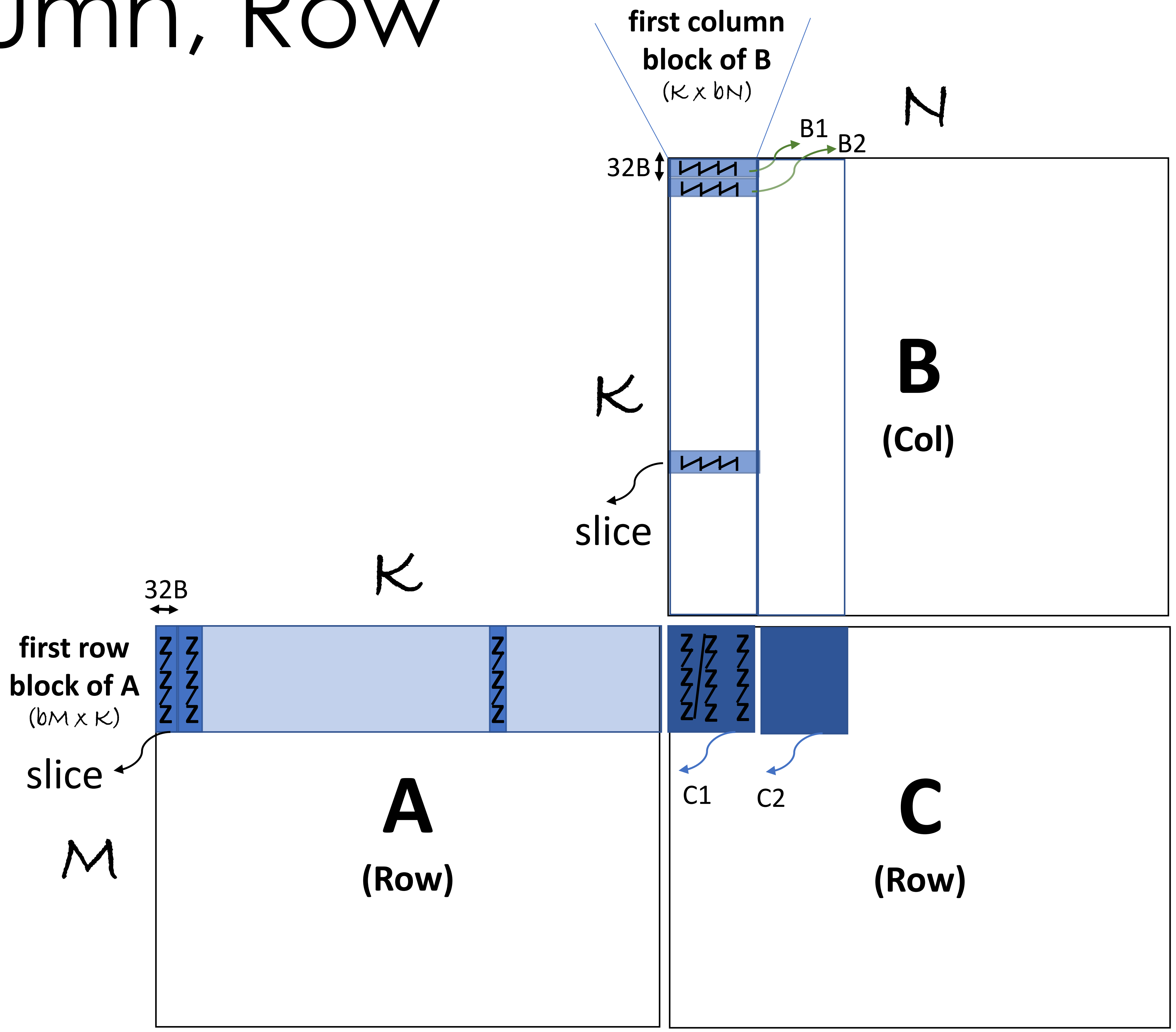


Matrices are pre/post processed, so all is well!

What happens with "fractalization-on-demand"?

Row, Column, Row

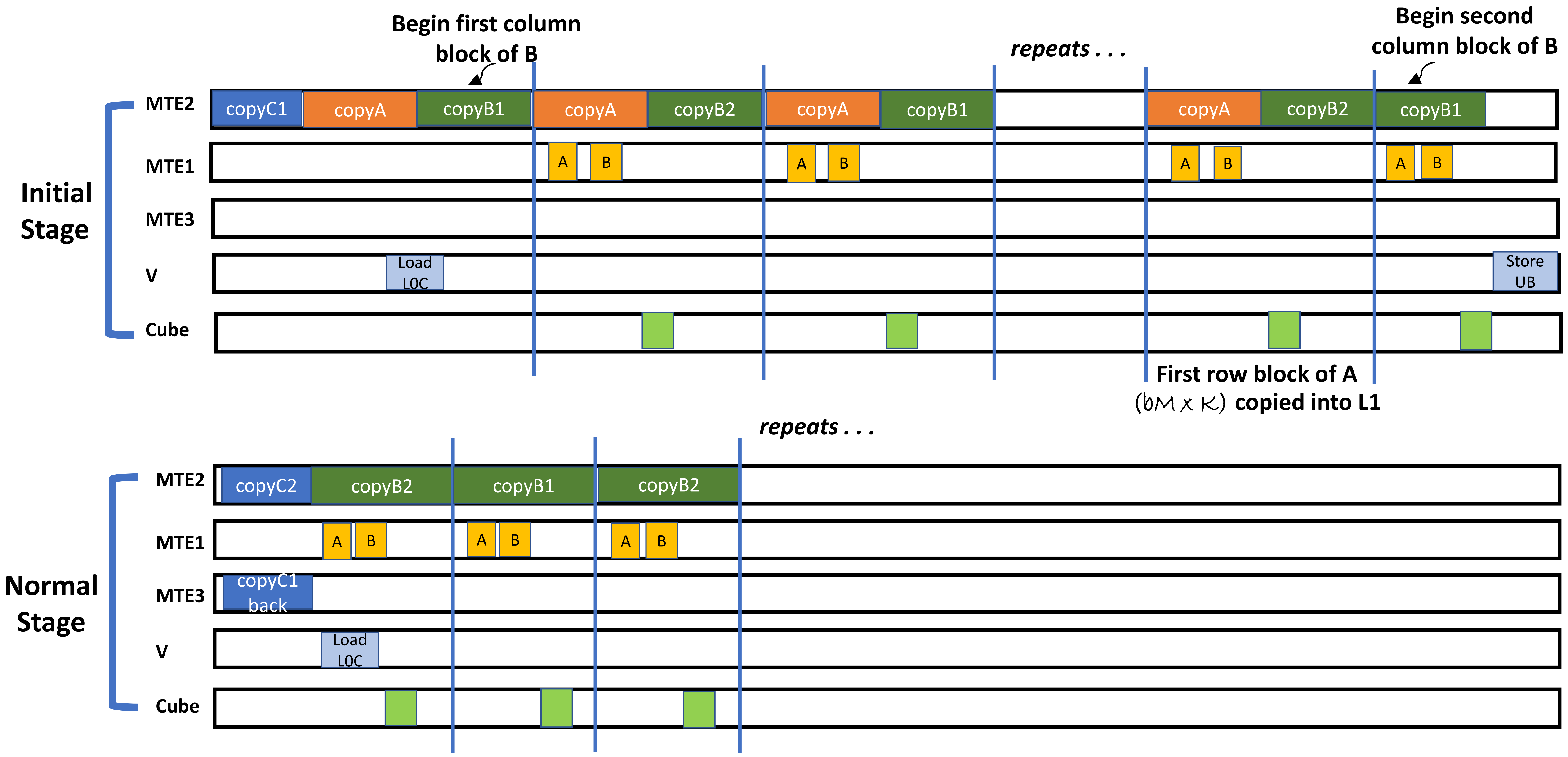




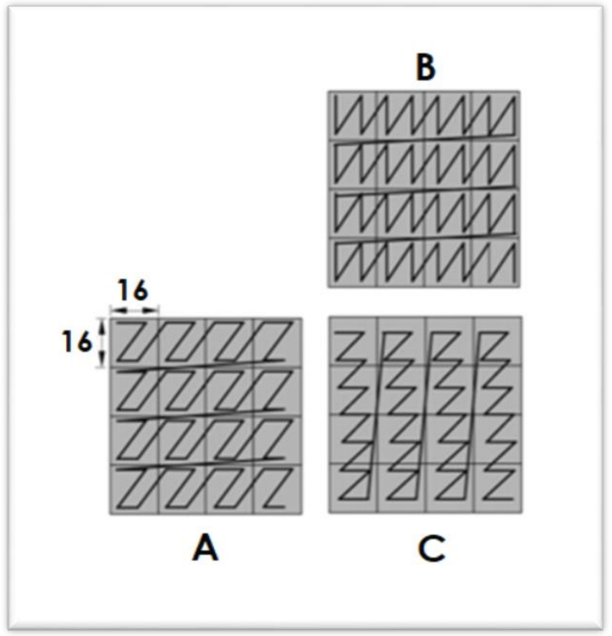
Consequence of "fractalization": bring only thin slices of A and B into L1 each DMA, and mad of 2 slices under utilizes cube unit and results in large C.

Row, Column, Row

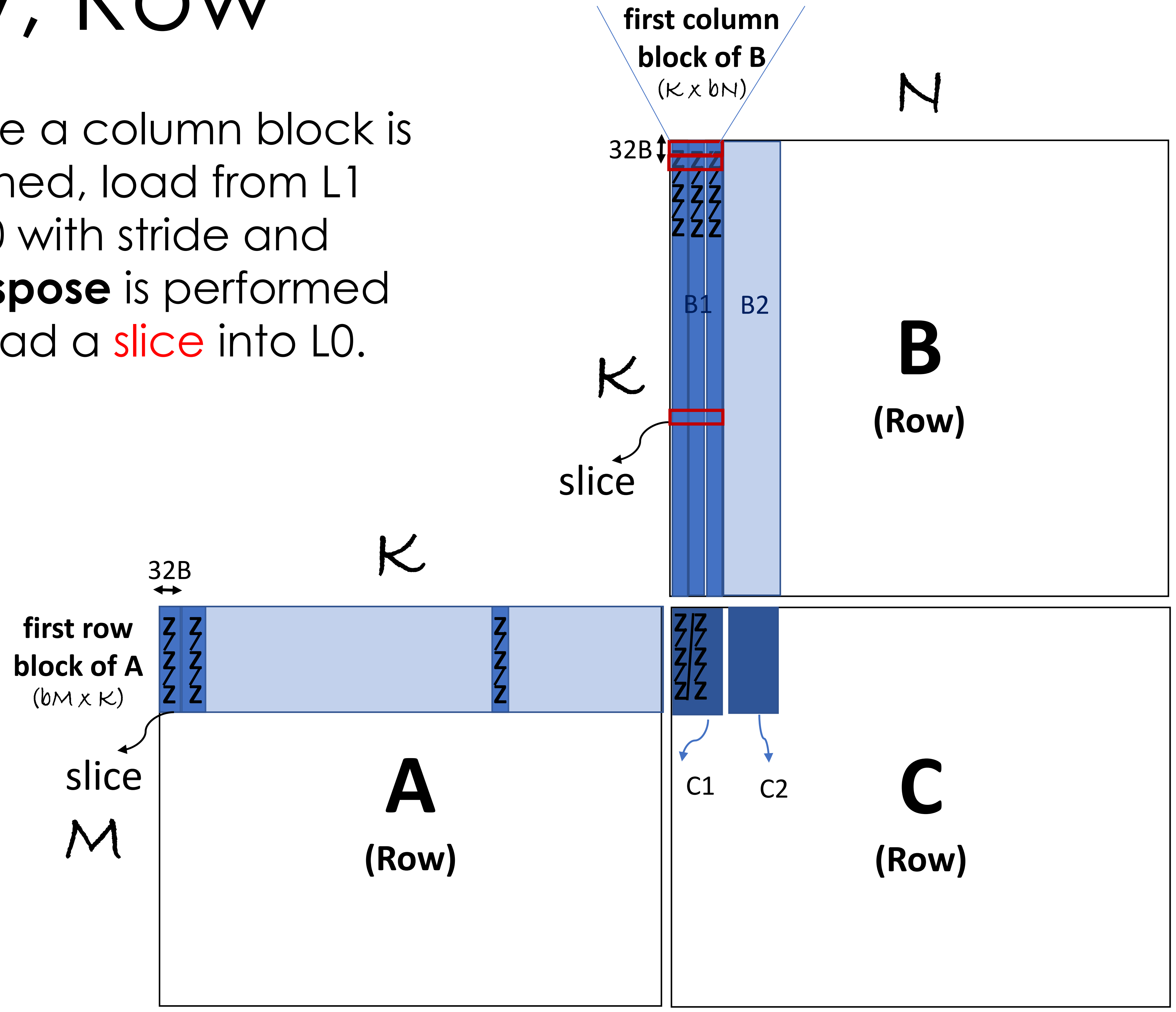
Execution Pipeline

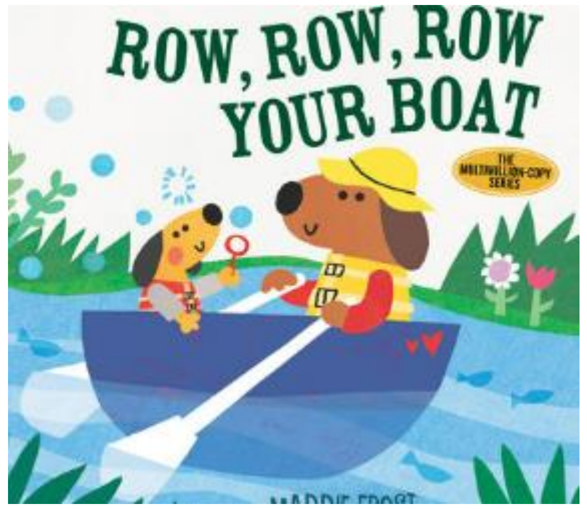


Row, Row, Row



Once a column block is fetched, load from L1 to L0 with stride and **transpose** is performed to load a slice into L0.

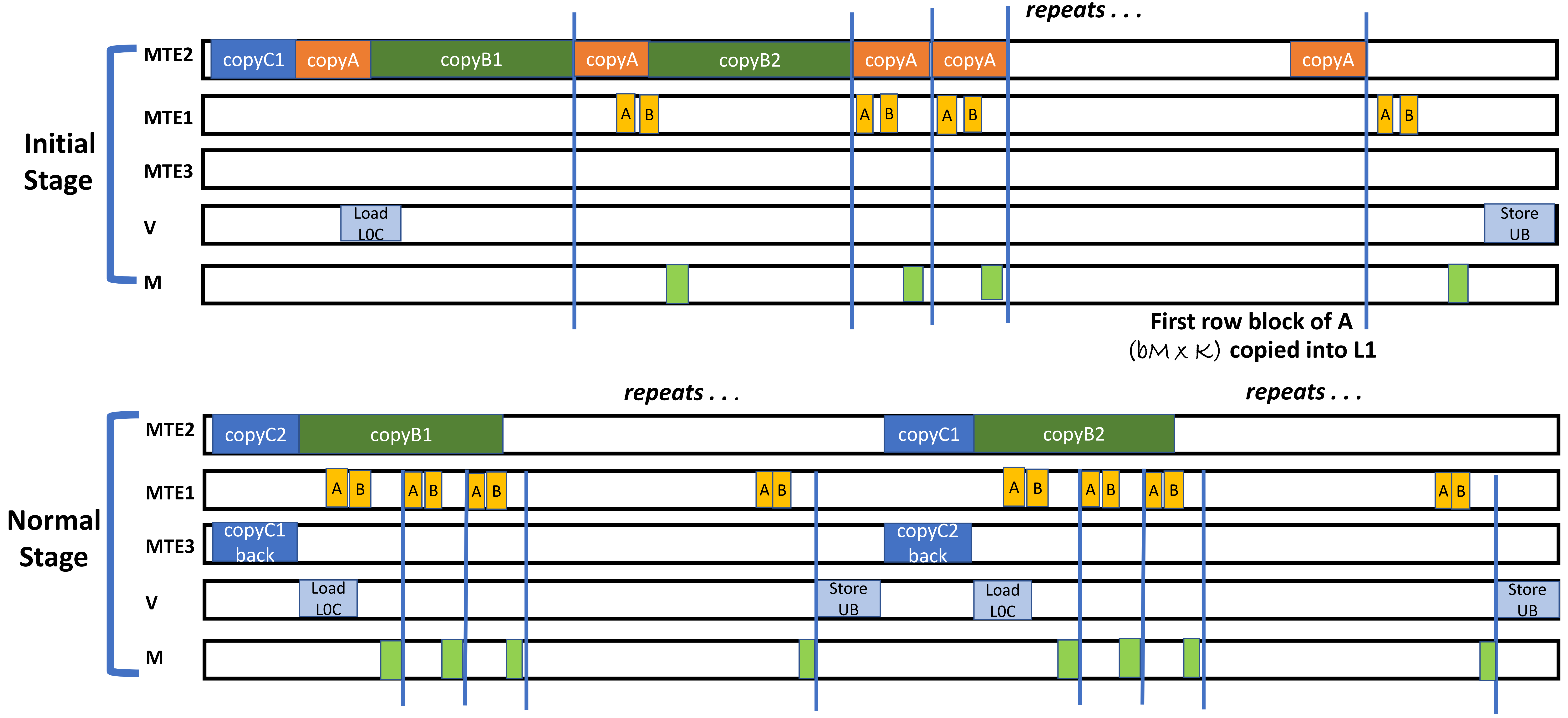




Now we need to do 3 DMAs before even getting a slice of B!

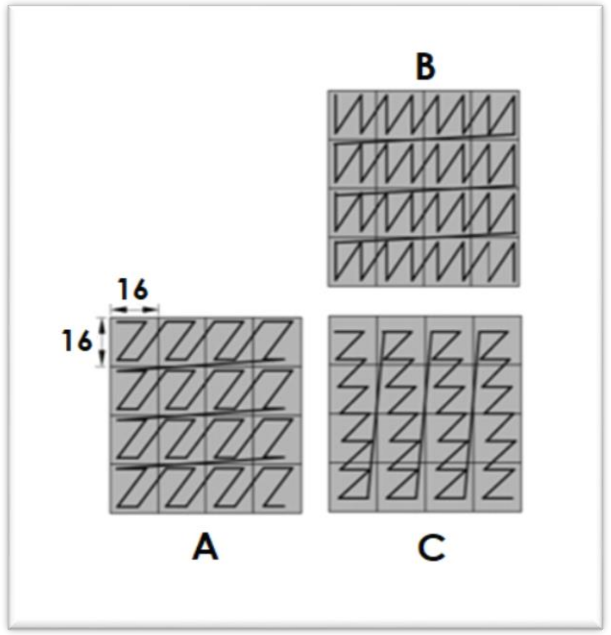
Row, Row, Row

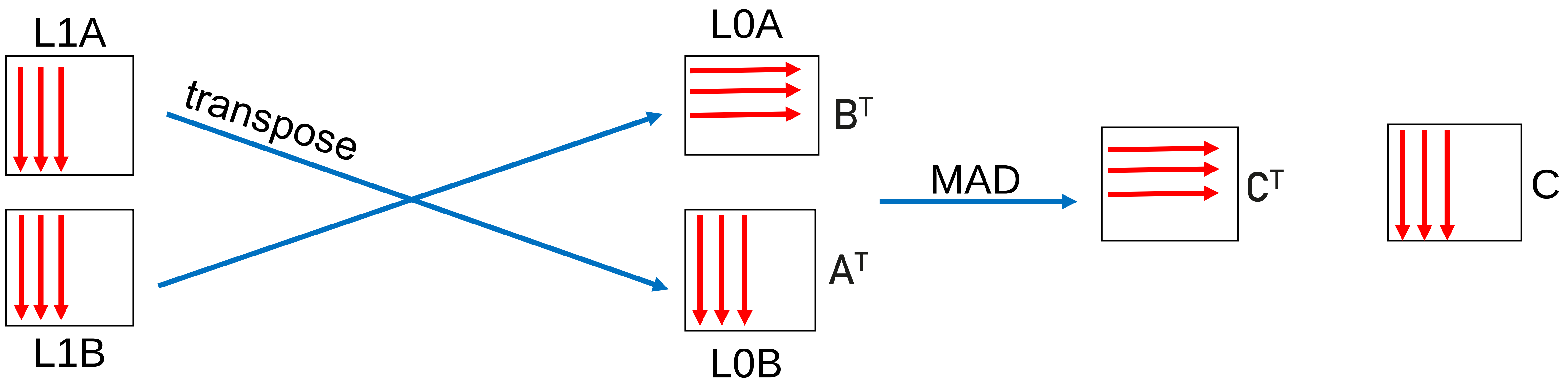
Execution Pipeline



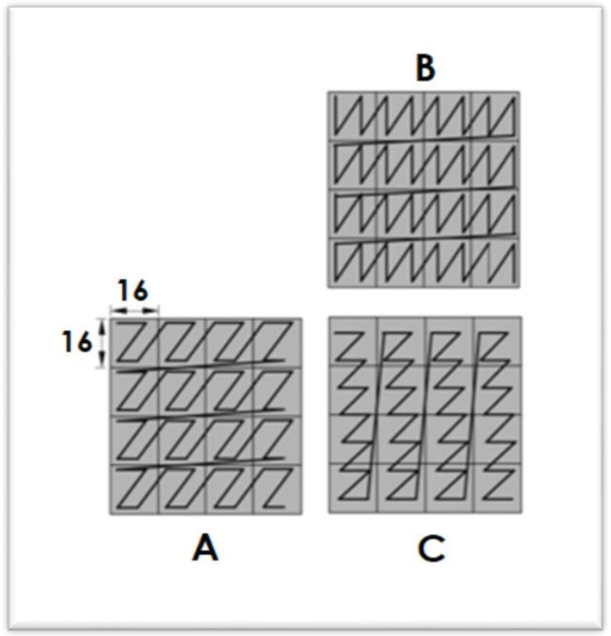
Column, Column, Column

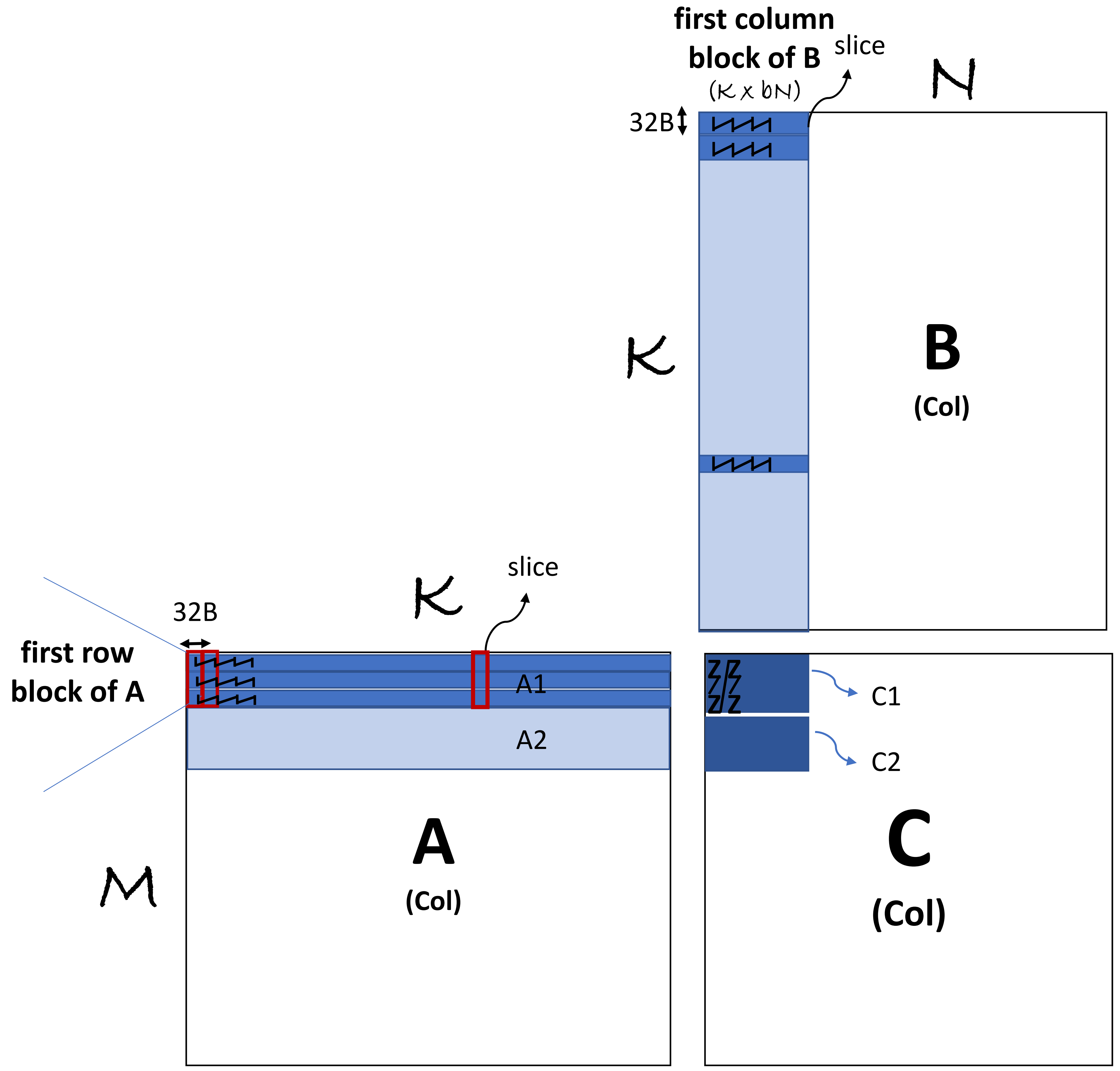
- C=AB, $C^T = B^T A^T$ (for each 16x16 fractal)
- **TODO:** After swap L0A and L0B, L0C size changes from bM*bN to bN*bM. UB is bM*bN. L0C and UB sizes don't match.





Column, Column, Column

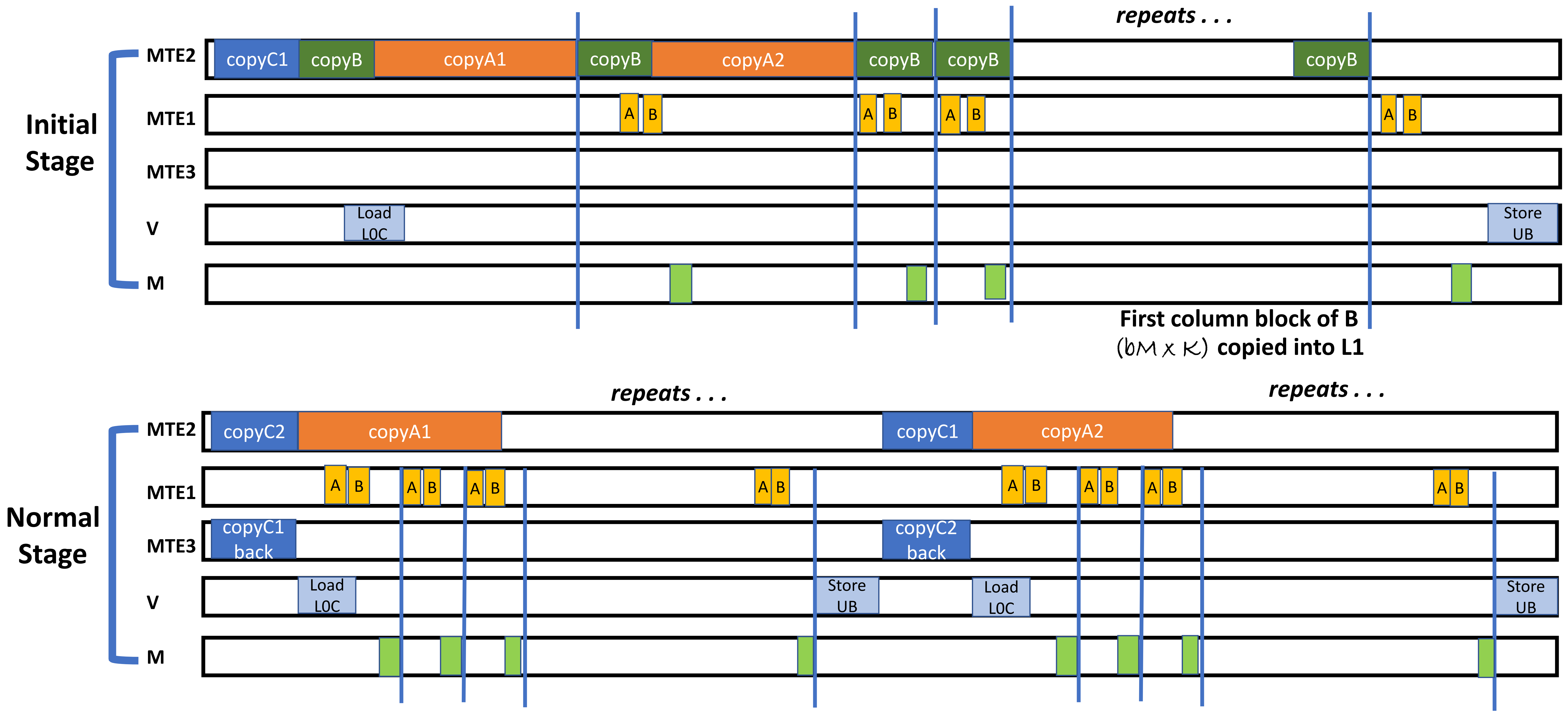




As bad as RRR, now we need to do 3 DMAs before getting a slice of A!

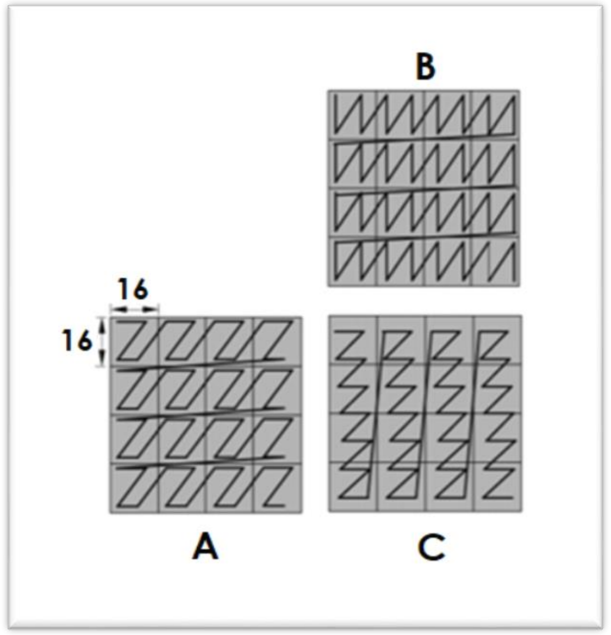
Column, Column, Column

Execution Pipeline



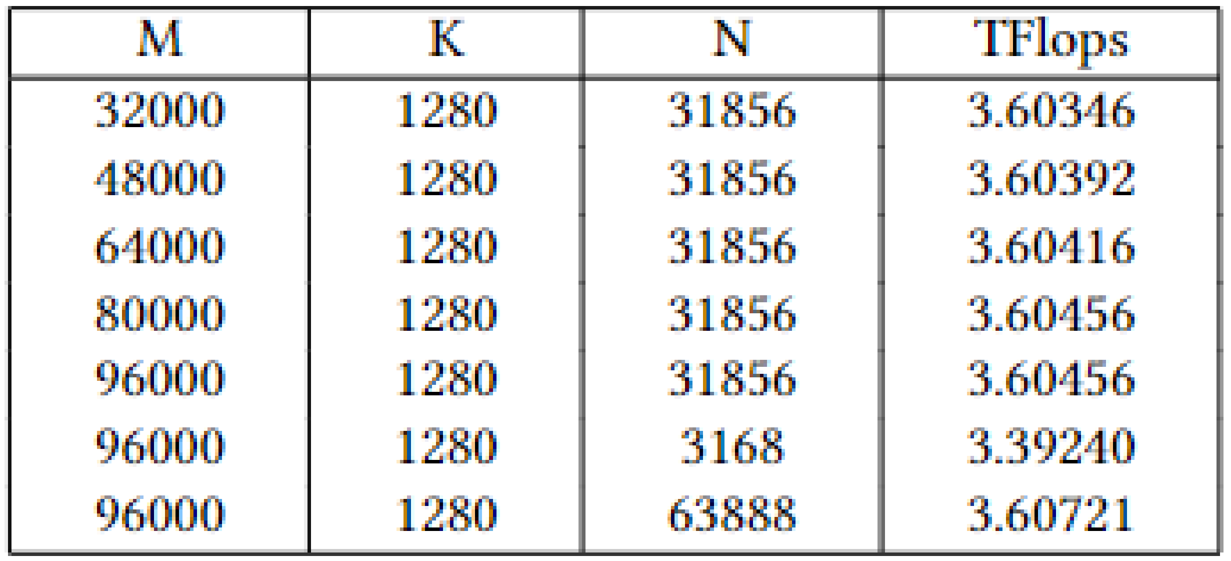
Performance Results

- Performance presented in half x half -> half, single aicore, ascend-910
- **"Fractalizing"** is limits the choices of tiling:
  - RCR
    - bM * K + 2 * (16 * bN) <= L1_size
    - bM * 16 <= L0A_size
    - 16 * bN <= L0B_size
    - bM * bN <= L0C_size
    - 2 * bM * bN <= UB_Size
  - RRR
    - bM * K + 2 * (K * bN) <= L1_size
    - bM * 16 <= L0A_size
    - 16 * bN <= L0B_size
    - bM * bN <= L0C_size
    - 2 * bM * bN <= UB_size



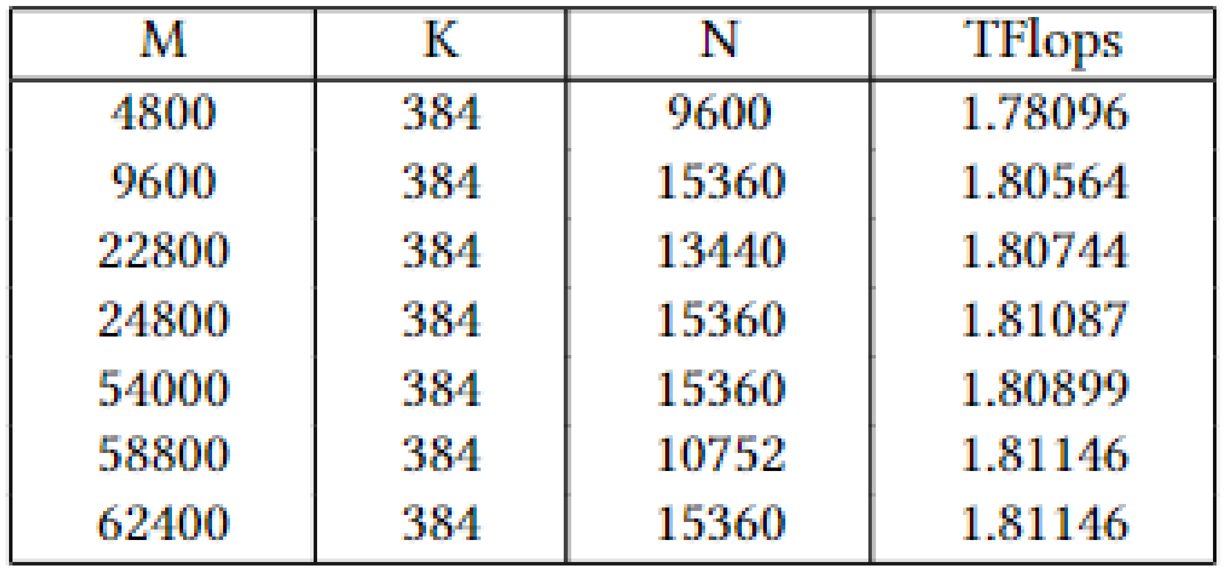
Performance Results

Table 1. RCR input data



M	K	N	TFlops
32000	1280	31856	3.60346
48000	1280	31856	3.60392
64000	1280	31856	3.60416
80000	1280	31856	3.60456
96000	1280	31856	3.60456
96000	1280	3168	3.39240
96000	1280	63888	3.60721

Table 2. RRR input data



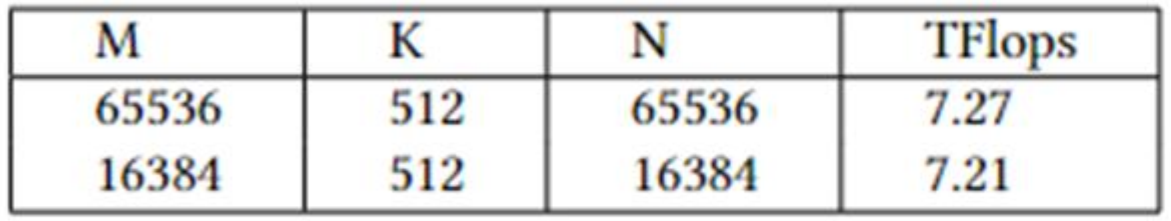
M	K	N	TFlops
4800	384	9600	1.78096
9600	384	15360	1.80564
22800	384	13440	1.80744
24800	384	15360	1.81087
54000	384	15360	1.80899
58800	384	10752	1.81146
62400	384	15360	1.81146

Why does RRR perform poorly?

1. Suffers from long start-up latency in the initial stage.
2. Unbalanced computation, DMA overlap in normal stage.
   - MTE2 not busy all the time, i.e. can not overlap with K/16 worth of the compute pipeline.

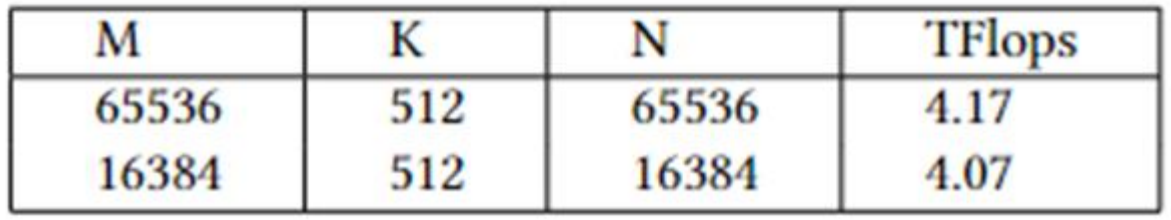
Performance Results

Table 1. 4D fractal layout input data with DB in L0



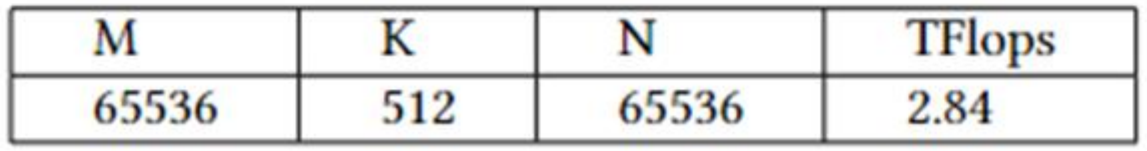
M	K	N	TFlops
65536	512	65536	7.27
16384	512	16384	7.21

Table 2. 4D fractal layout input data without DB in L0



M	K	N	TFlops
65536	512	65536	4.17
16384	512	16384	4.07

Table 3. 4D fractal layout input data with DB in L0 with pre post processing time



M	K	N	TFlops
65536	512	65536	2.84

Why does fractal layout perform so well?

- Fully utilize L0 to achieve largest MAD possible
- Double buffering at L0 makes a difference

But let's account of pre/post layout conversions with 16 CPU cores

- Performance drops from 7.2 to 2.8 TFlops

Conclusion: How do we improve it?
1.8 TFlops (RRR) < 2.8 Tflops < 3.6 (RCR) Tflops

..... To be continued..

Thank you!

Q & A