In-Place Matrix Transposition on GPUs

Presented by: Owen Roseborough

Based on the paper by Juan Gómez-Luna, I-Jui Sung, Li-Wen Chang, José M. González-Linares, Nicolás Guil, and Wen-Mei W. Hwu

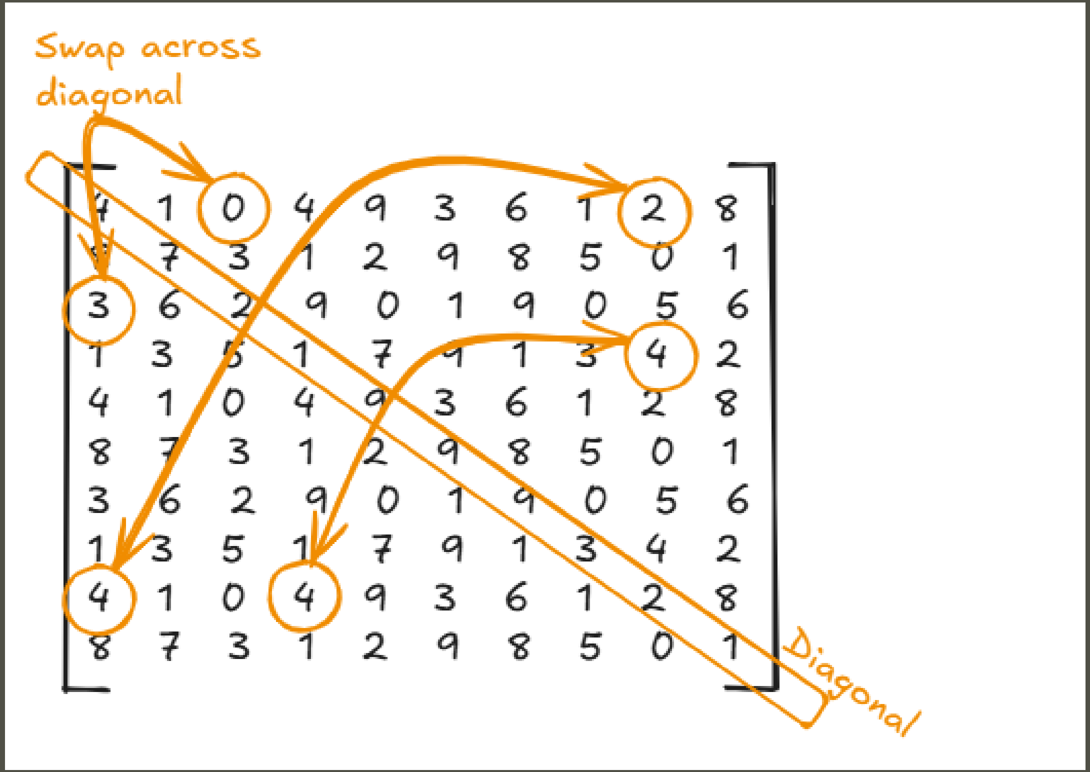
Introduction to the Research Area

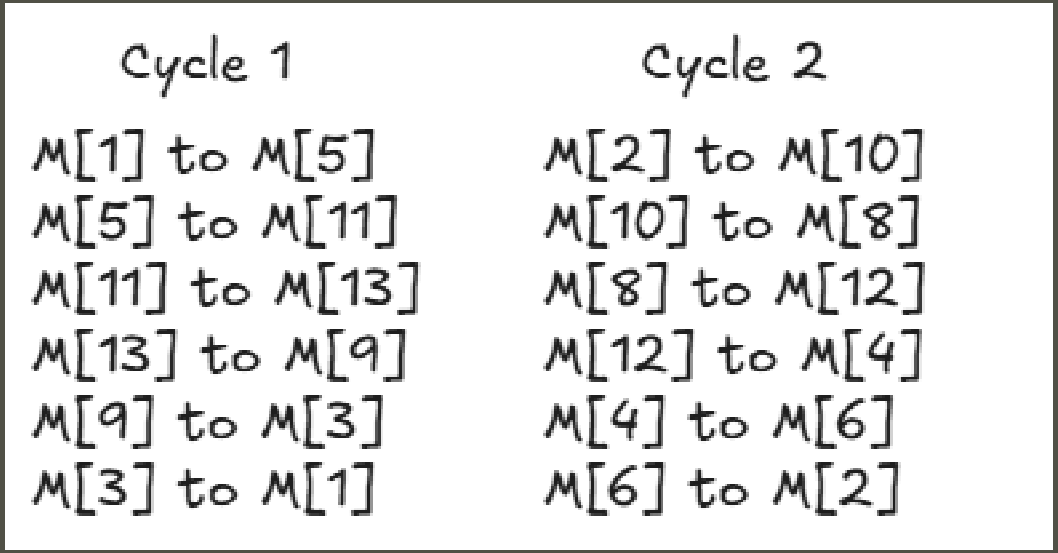
- **Matrix Transposition (MT)**: Rearranging a matrix such that rows become columns.
- Commonly used in:
  - **2D Fast Fourier Transform (FFT)** – iterates row major, then column major, but column major accesses inefficient – the solution is a transpose, iterate row major, then transpose back
  - **K Means Clustering** – the algorithm needs to perform a matrix multiplication with matching inner dimensions – solution is a transpose
- **Majority of work of MT is swapping elements**, so the speed of execution is limited/enhanced by the sustained memory bandwidth of GPU

Introduction to the Research Area

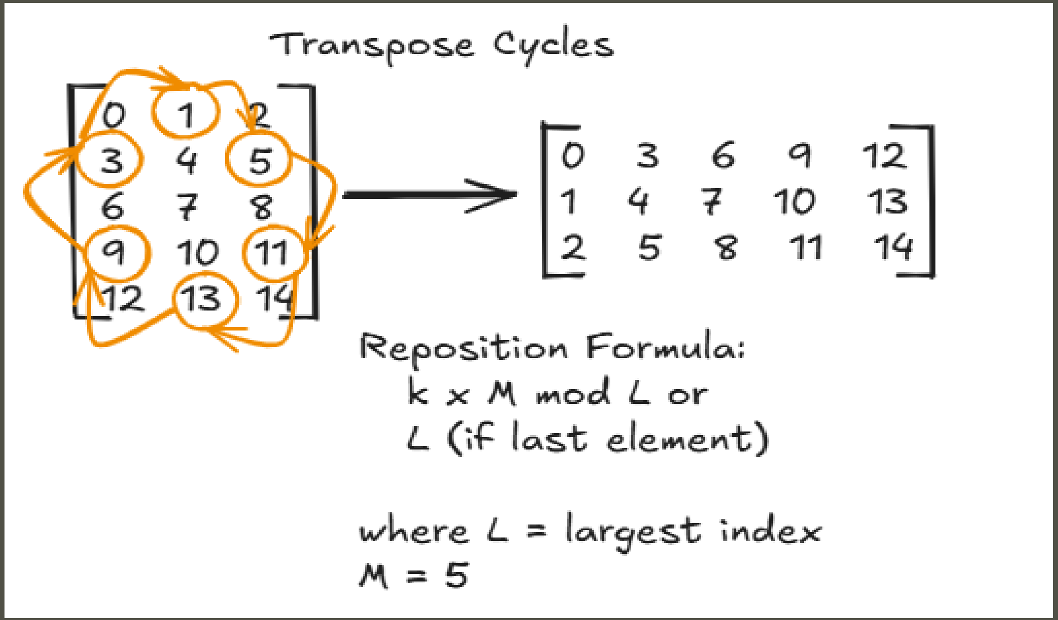
- **Out-of-Place vs. In-Place**
  - **Out-of-Place** transposition can only be used to transpose matrix(s) that are 50% of the size of the GPUs memory capacity, since the other 50% is needed for the transposed matrix
  - **In-Place** transposition is memory-efficient, critical for large matrices on GPUs.
- **Challenges of In-Place Transposition**
  - Requires memory-efficient algorithms.
  - GPUs have memory access constraints and bank conflicts.

Square Matrix Transposition has as many cycles as elements above or below diagonal – already have efficient implementations





Cycle 1	Cycle 2
M[1] to M[5]	M[2] to M[10]
M[5] to M[11]	M[10] to M[8]
M[11] to M[13]	M[8] to M[12]
M[13] to M[9]	M[12] to M[4]
M[9] to M[3]	M[4] to M[6]
M[3] to M[1]	M[6] to M[2]



Introduction to the Research Area

- Transposing Rectangular Matrices with **chains of shifting**
- Note we have **no shared** Matrix indices between cycles – **Parallelism!**
- “However, for massively parallel systems that require thousands of concurrently active threads to attain maximum parallelism, **this form of parallelism alone is neither sufficient nor regular”**
- Authors make note that Cate and Twigg have proven that “the length of the longest cycle is always **multiple times** the lengths of other cycles”

B. Catanzaro, A. Keller, and M. Garland, “A decomposition for in place matrix transposition,”

- A different approach by B.Catanzaro presents a method that decomposes the transpose into three steps: column rotation, row shuffle, and column shuffle.
- Performing the column rotate step ensures that the row shuffling maps to distinct locations.
- This enables concurrent execution without data conflicts. Each of the rotation and shuffles operate on separate rows or columns.

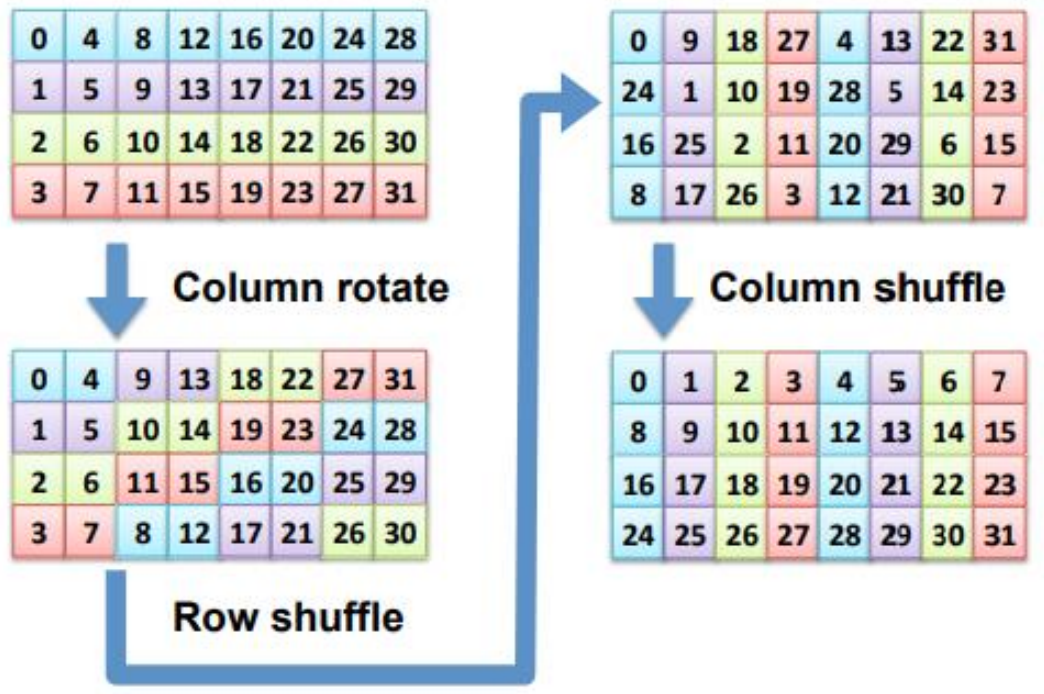


Figure 2: C2R transpose of 4 × 8 matrix

Full Transposition As a Sequence of Elementary Tiled Transpositions

- The dimensions of a M x N matrix are factorized as M x m x N x n, a 4-dimensional array

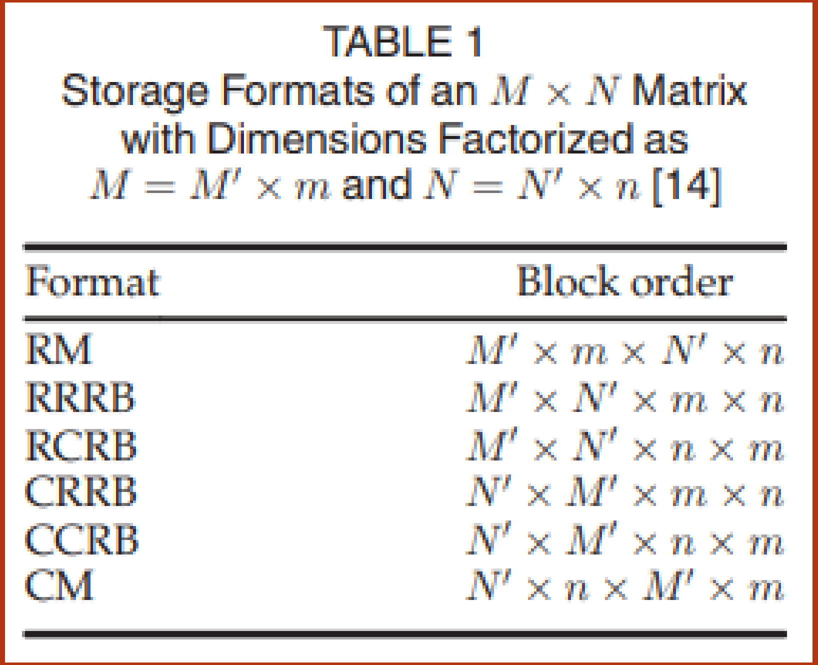


TABLE 1
Storage Formats of an $M \times N$ Matrix with Dimensions Factorized as $M = M' \times m$ and $N = N' \times n$ [14]

Format	Block order
RM	$M' \times m \times N' \times n$
RRRB	$M' \times N' \times m \times n$
RCRB	$M' \times N' \times n \times m$
CRRB	$N' \times M' \times m \times n$
CCRB	$N' \times M' \times n \times m$
CM	$N' \times n \times M' \times m$

- Then elementary transpositions are carried out with adjacent dimensions, the goal being to get from RM to CM

Full Transposition As a Sequence of Elementary Tiled Transpositions

- “Table 2 employs the factorial numbering system [18] to name each elementary transposition. This table lists possible permutations that refer to the swapping of adjacent dimensions”

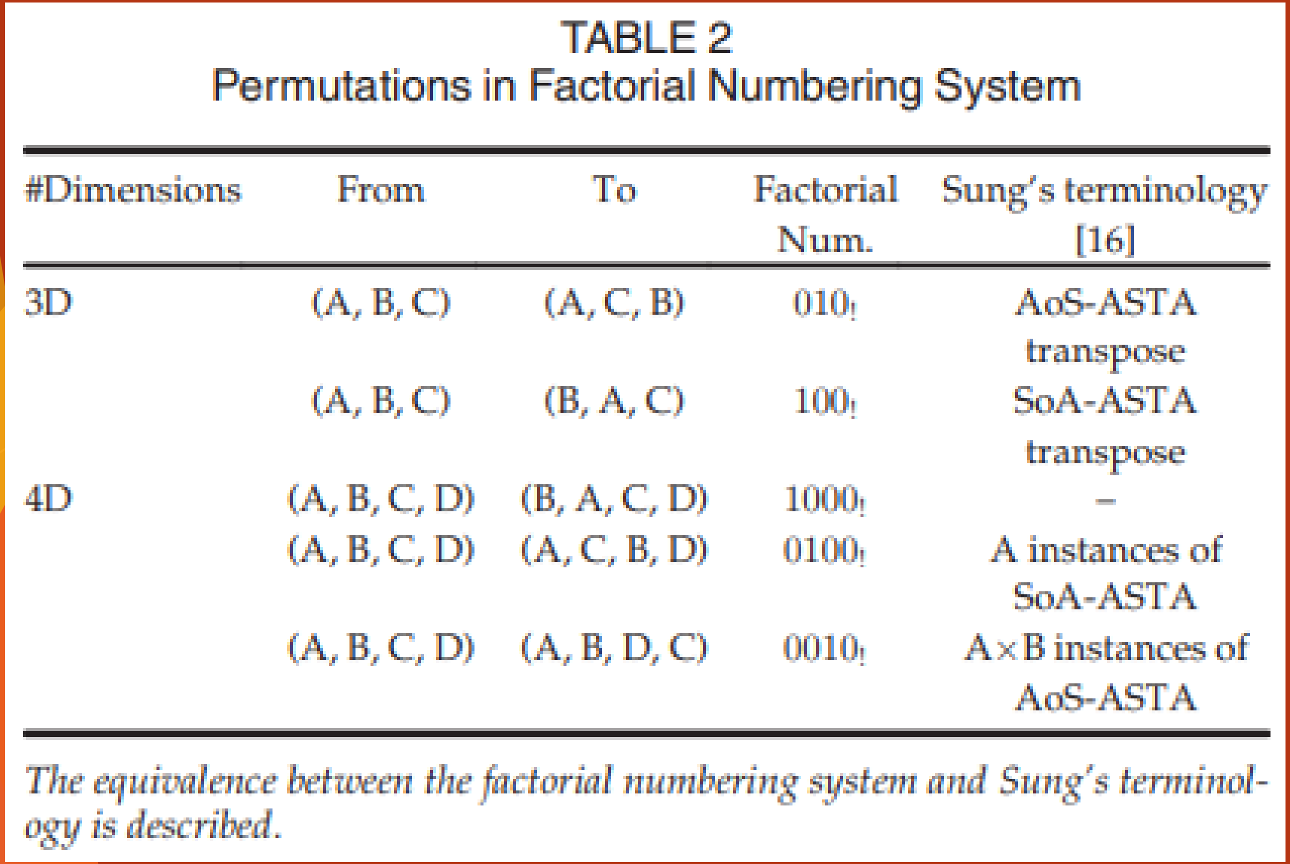


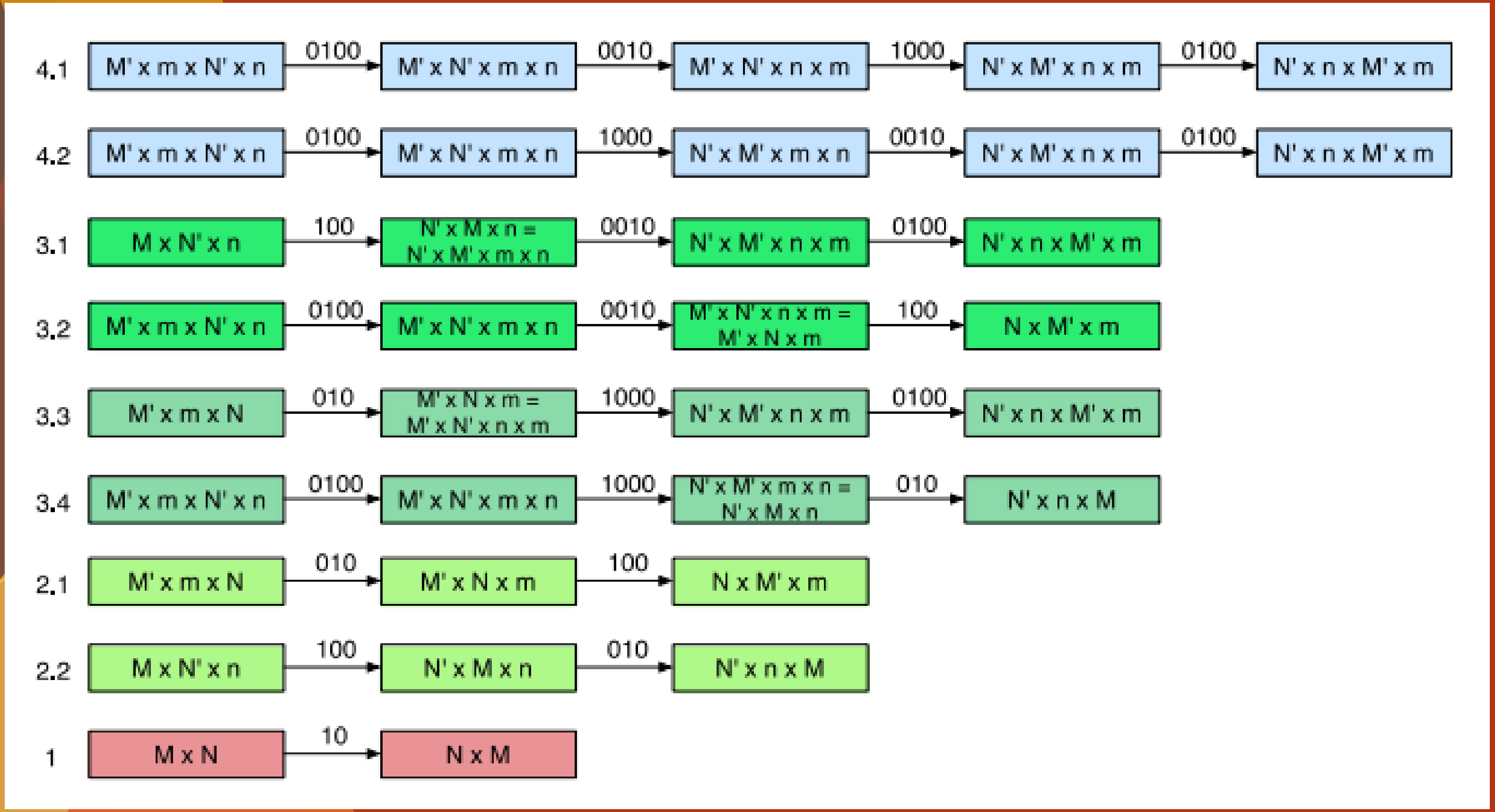
TABLE 2
Permutations in Factorial Numbering System

#Dimensions	From	To	Factorial Num.	Sung’s terminology [16]
3D	(A, B, C)	(A, C, B)	$010_!$	AoS-ASTA transpose
	(A, B, C)	(B, A, C)	$100_!$	SoA-ASTA transpose
4D	(A, B, C, D)	(B, A, C, D)	$1000_!$	–
	(A, B, C, D)	(A, C, B, D)	$0100_!$	A instances of SoA-ASTA
	(A, B, C, D)	(A, B, D, C)	$0010_!$	A$\times$B instances of AoS-ASTA

The equivalence between the factorial numbering system and Sung’s terminology is described.

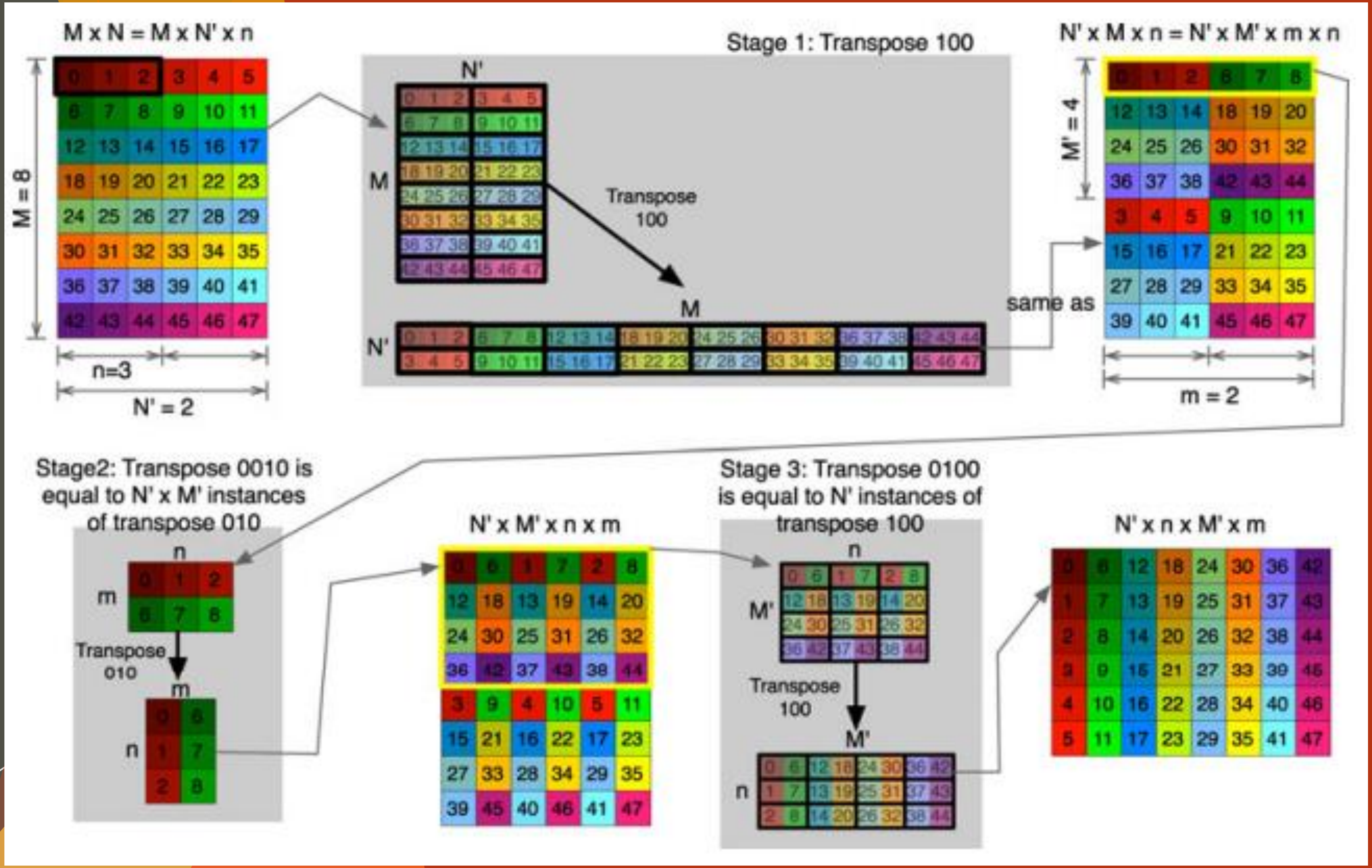
Full Transposition As a Sequence of Elementary Tiled Transpositions

- We have nine possible such factorized transformations:



Full Transposition As a Sequence of Elementary Tiled Transpositions

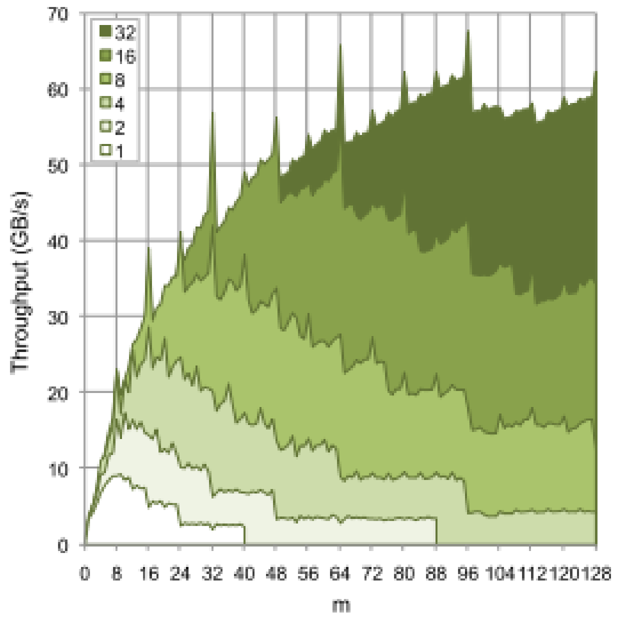
- Full Sequence 3.1 shown here:



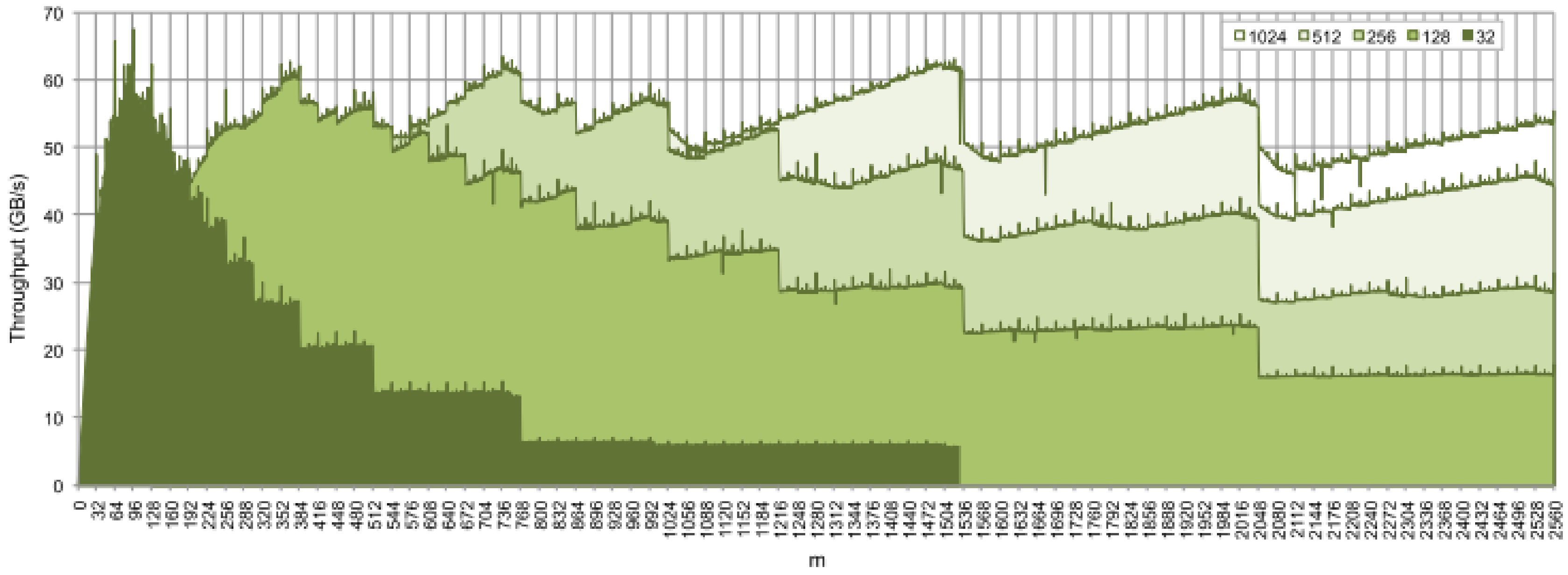
M x N = M x N' x n
M = 8
n=3
N' = 2
Stage 1: Transpose 100
N'
M
Transpose 100
M
N'
same as
N' x M x n = N' x M' x m x n
M' = 4
m = 2
Stage2: Transpose 0010 is equal to N' x M' instances of transpose 010
n
m
Transpose 010
m
n
N' x M' x n x m
Stage 3: Transpose 0100 is equal to N' instances of transpose 100
n
M'
Transpose 100
M'
n
N' x n x M' x m

Tuning SIMD and Thread Group Sizes

chunks too small -> too few threads working -> low parallelism -> low speed
chunks too big -> exceed GPU memory limits -> occupancy drops -> low speed



(a) $m \leq 128$



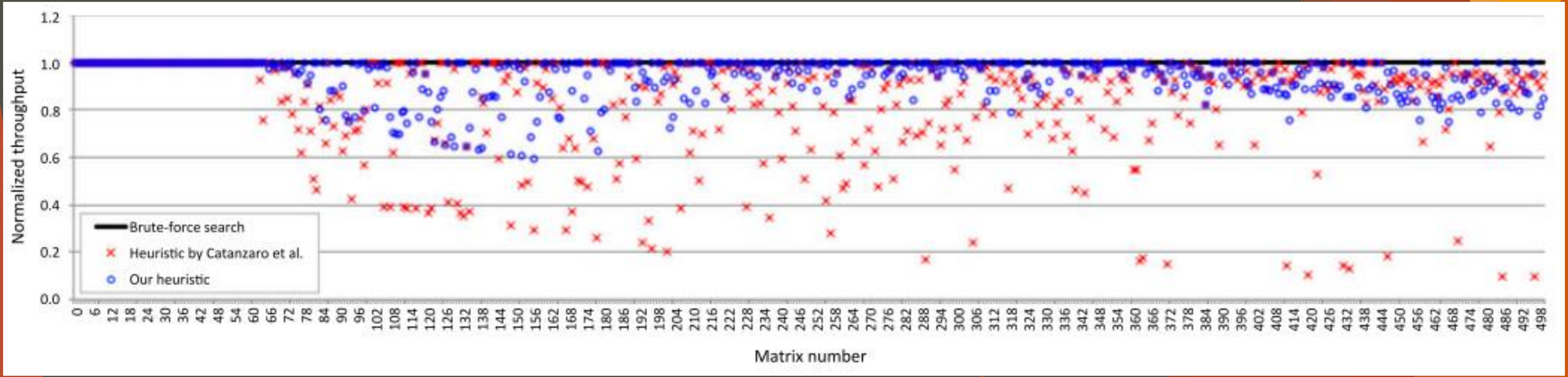
(b) $m \leq 2560$

- The above figure shows how fast in GB/s the 100! matrix transposition method runs on an NVIDIA K20 GPU when the height of each matrix chunk (m) is less than 2560. The left graph shows how performance changes when using different virtual SIMD unit sizes. The right graph shows how performance changes when using different work-group sizes.

Heuristic for Selecting Tile Size

Catanzaro heuristic picks largest possible tile size that divides the matrix and is no longer than 72

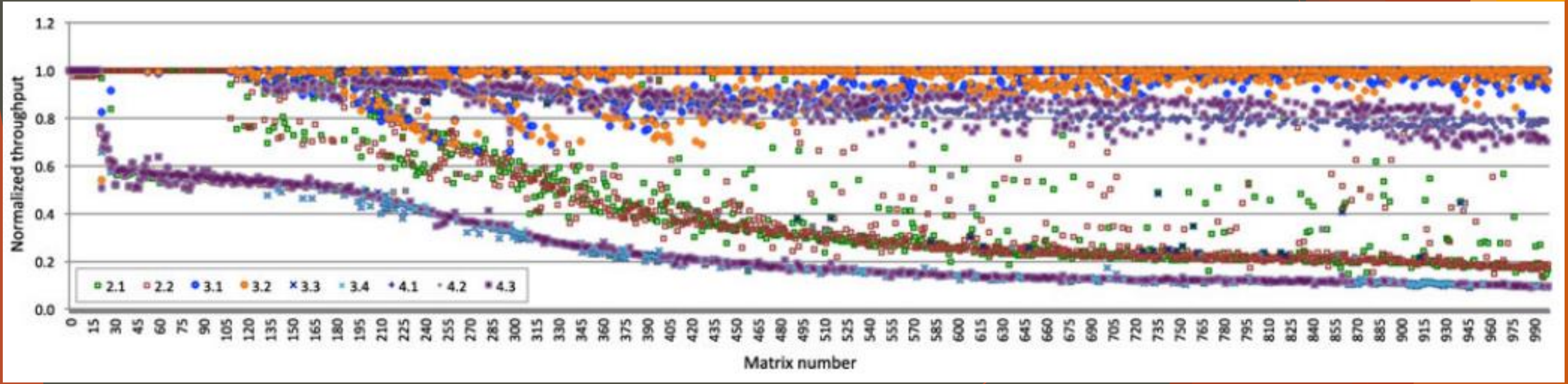
paper’s heuristic tries combinations of tile sizes within settable limits and fit in GPU local memory. It picks the combination that allows the most GPU threads to run in parallel.



Experimenting to Find Best Factorization

The authors tested different sequences of transpositions on an NVIDIA K20 GPU. They ran experiments on 1,000 random matrices, and chose optimal tile sizes based on previously mentioned heuristic

Sequence 3.1 and 3.2 consistently give best performance. Sequence 3.1 worked better when n > m, and the reverse for 3.2. This is because the super-element size for 3.1 is n and for 3.2 is m.



Very Small Super Element Size

"A very small super-element size m or n makes transposition 100! barely achieve 10GB/s or less ... Even worse would be the case of both m and n very small: less than 5 GB/s“

The example given is that of small prime dimensioned matrices, they will have super element sizes of 1 and 1. Solution to this is to use padding

each row is handled by a work-group, loads data into its memory, waits for next row/work-group, then writes back result. Synchronization flags and memory barriers ensure correct ordering

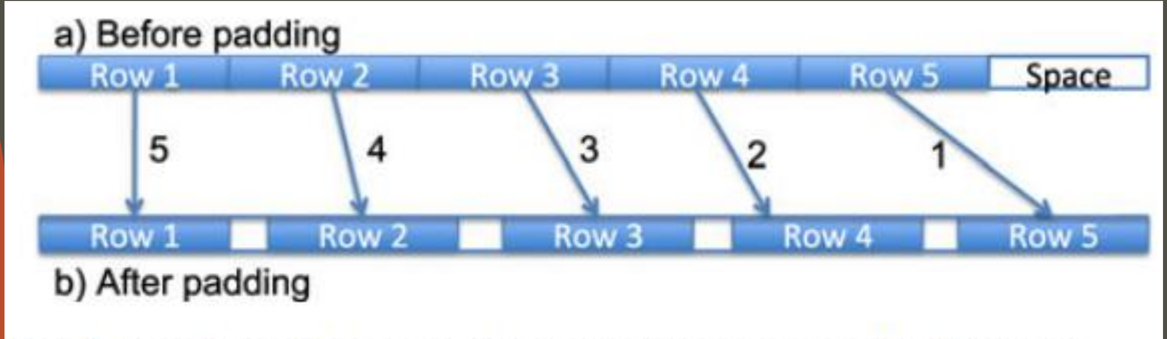
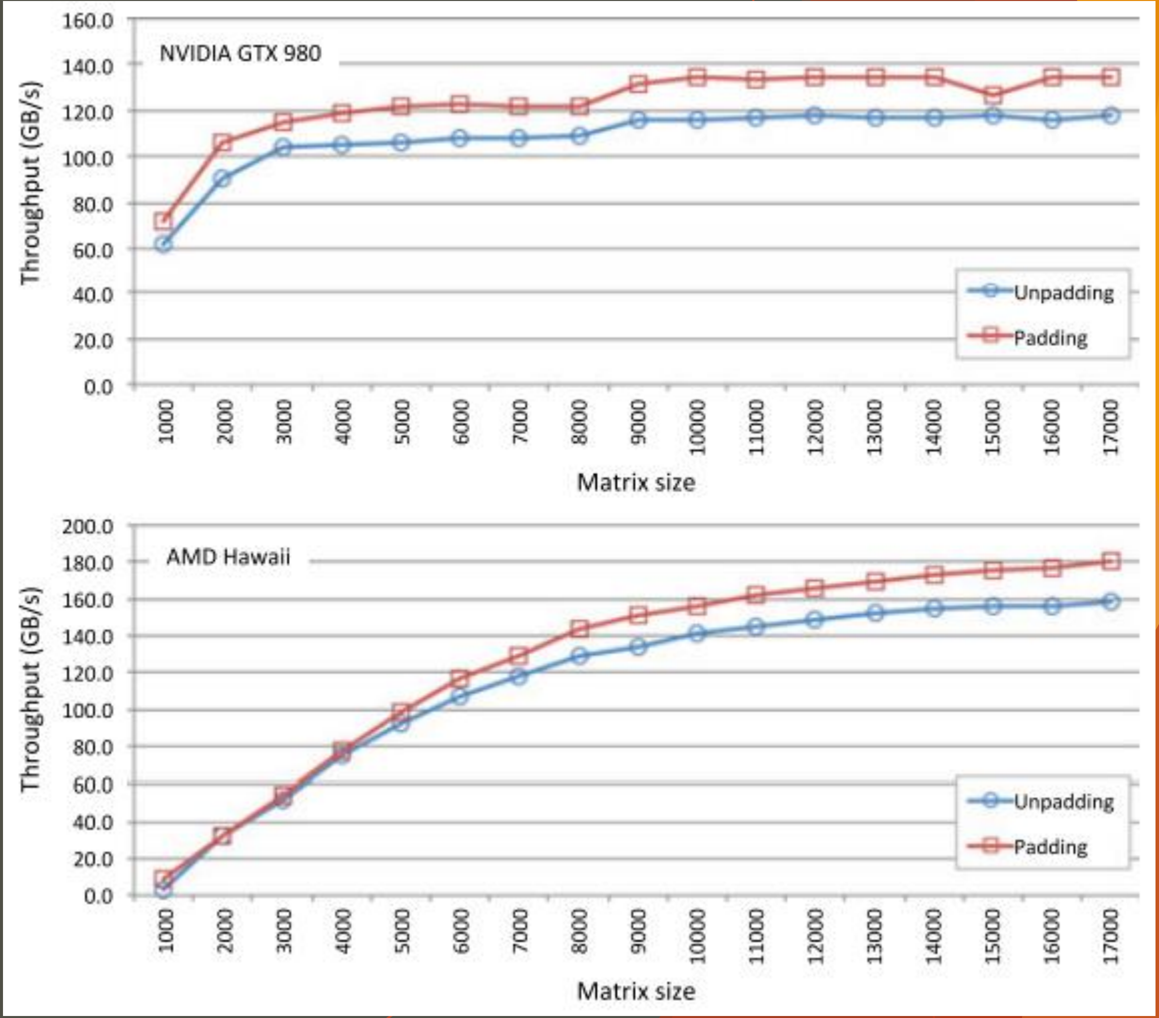
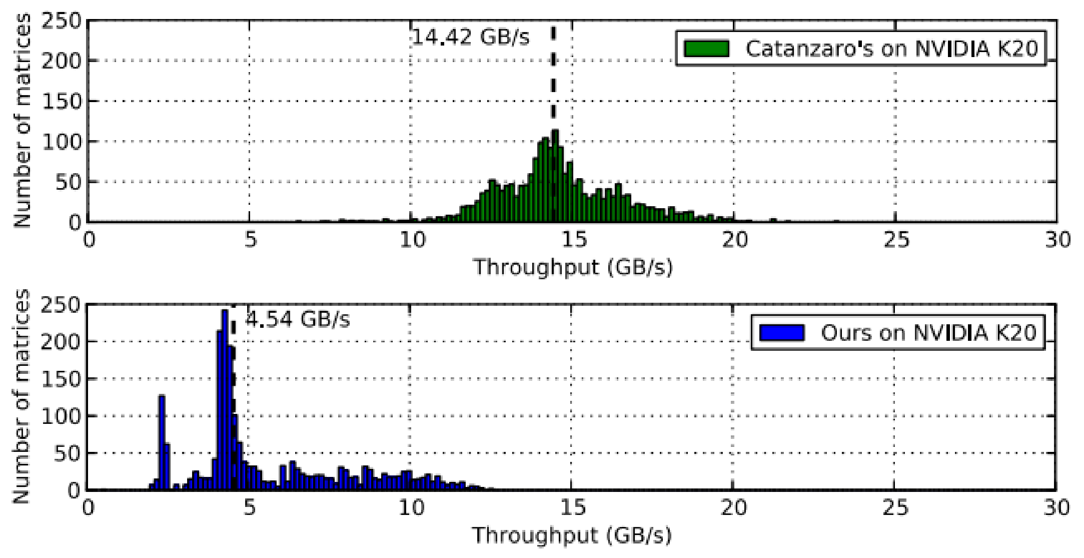


Fig. 7. Padding in-place a row-major matrix composed by five rows.

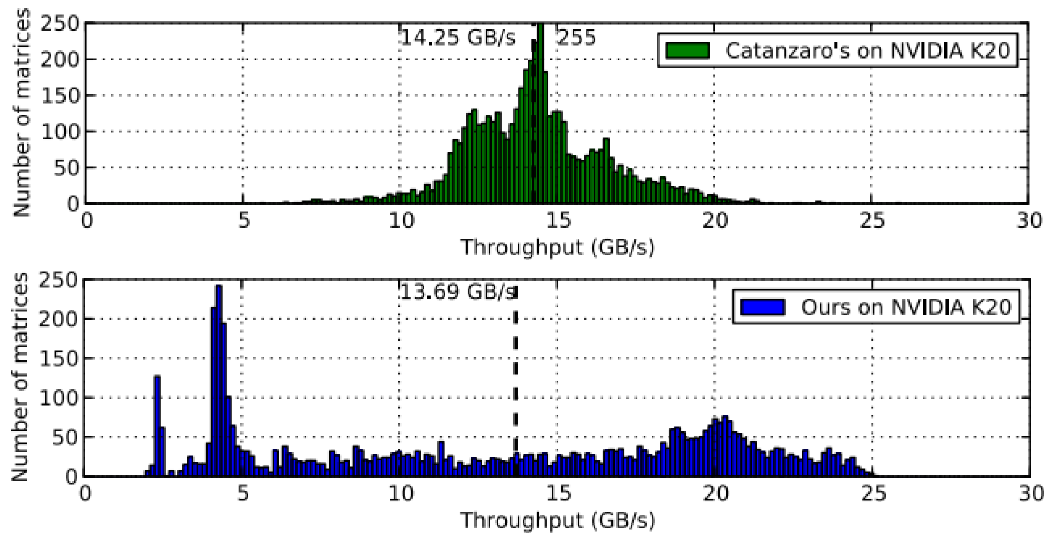


Experimental Results

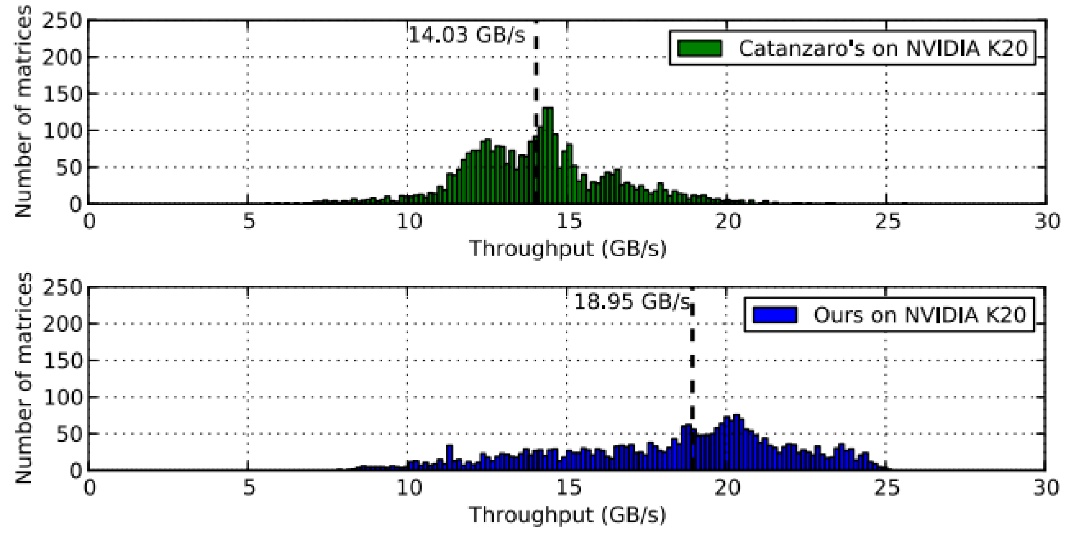
The authors compared their GPU matrix transposition implementation to Catanzaro’s on an NVIDIA K20 GPU. They found that for matrices with m and n greater than 4 and 6 respectively they achieved higher throughput (18.95 GB/s vs 14.03 GB/s Catanzaro)(no padding)



(c) 2074 matrices ($m \leq 4$ and $n \leq 6$)



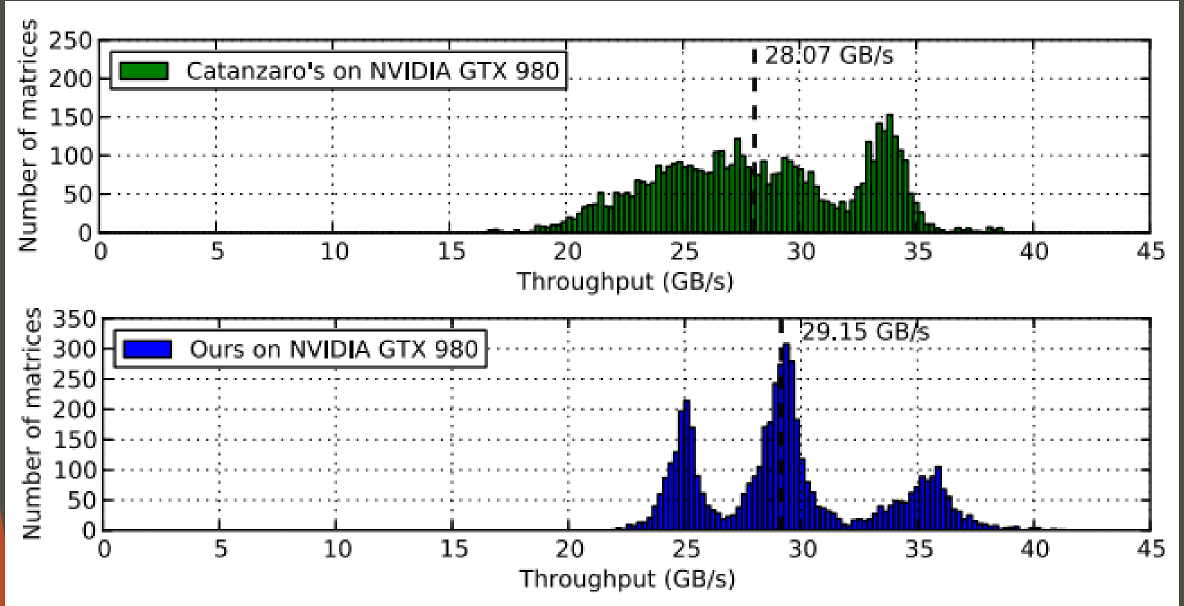
(a) 5000 matrices



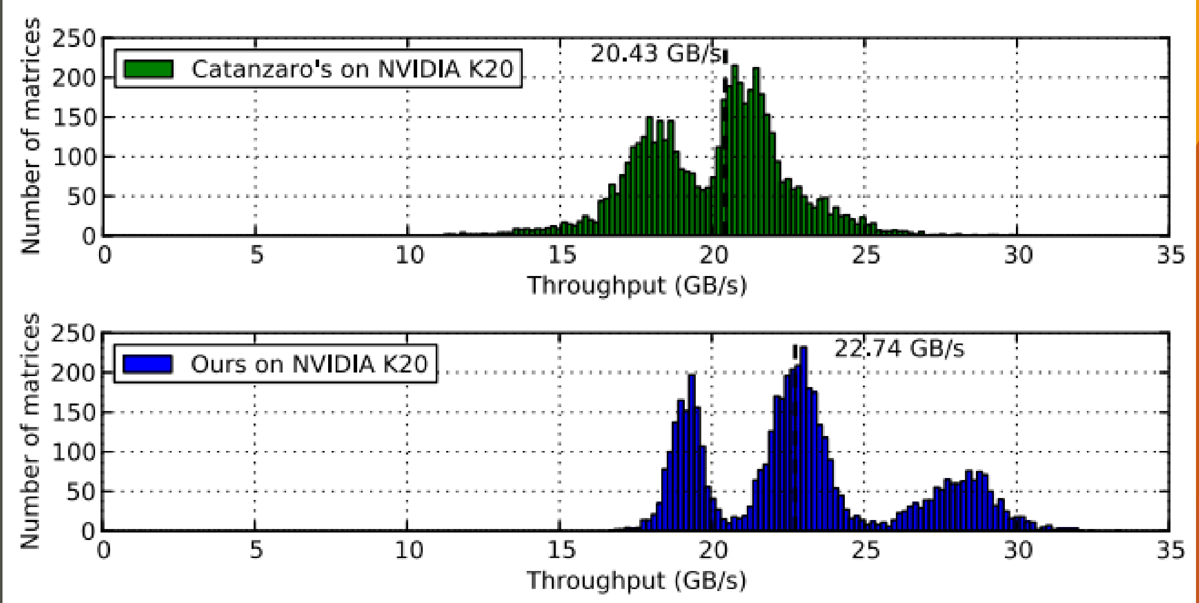
(b) 2926 matrices ($m > 4$ and $n > 6$)

Experimental Results

With one or two dimensions padded though the authors achieved a slightly better throughput than Catanzaro’s implementation -> 29.15 GB/s vs 28.07 GB/s and 22.74 GB/s vs 20.43 GB/s



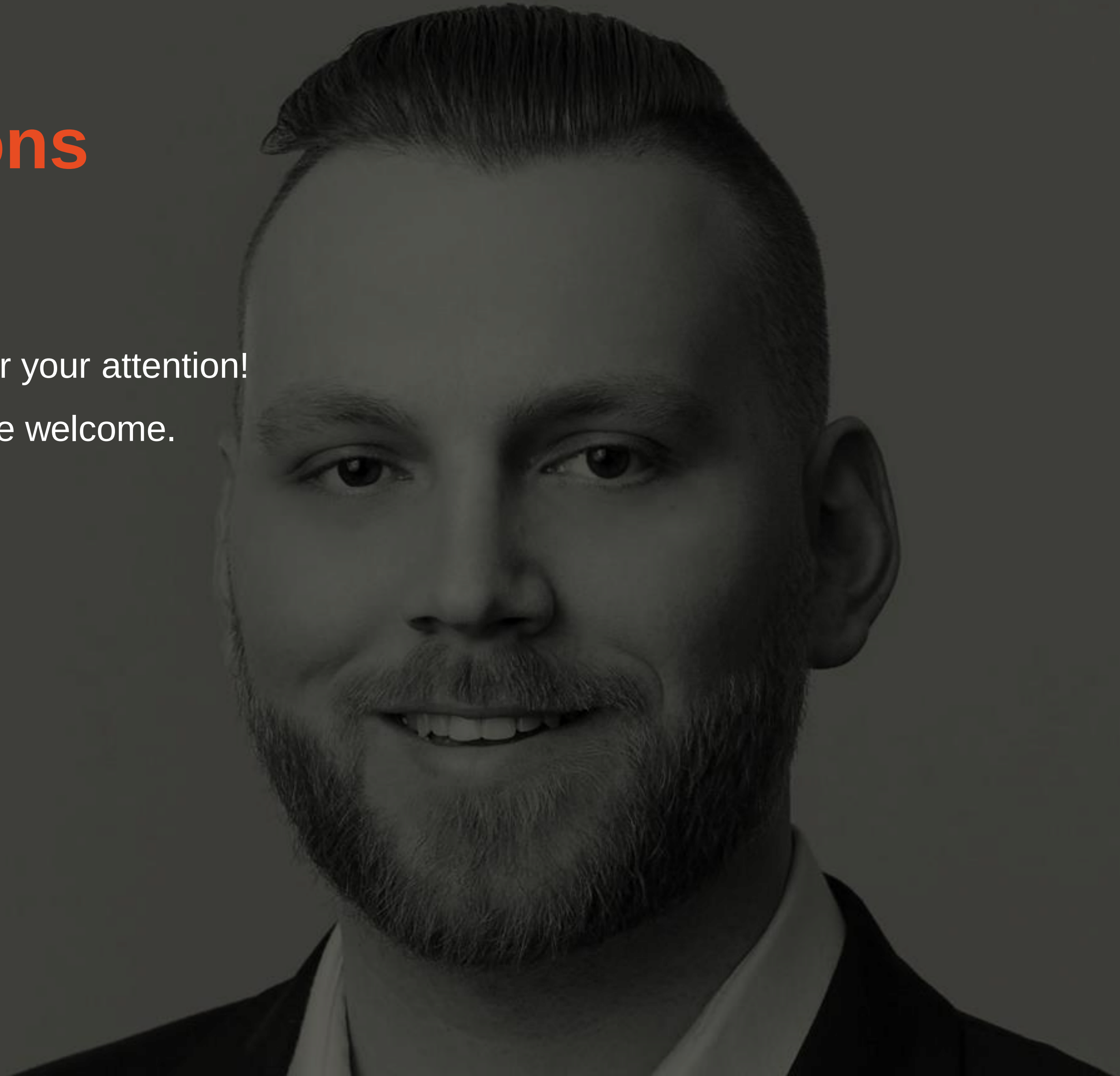
(a) Double precision



(a) Double precision

Conclusion

- The proposed in-place matrix transposition approach effectively leverages GPU memory bandwidth.
- Optimized tile selection and padding in special cases ensure superior performance.
- outperforms existing methods



Questions

- Thank you for your attention!
- Questions are welcome.

References

- **In-Place Matrix Transposition on GPUs** by Juan Gómez-Luna et al.
- **A decomposition for in place matrix transposition** by B. Catanzaro, A. Keller, and M. Garland