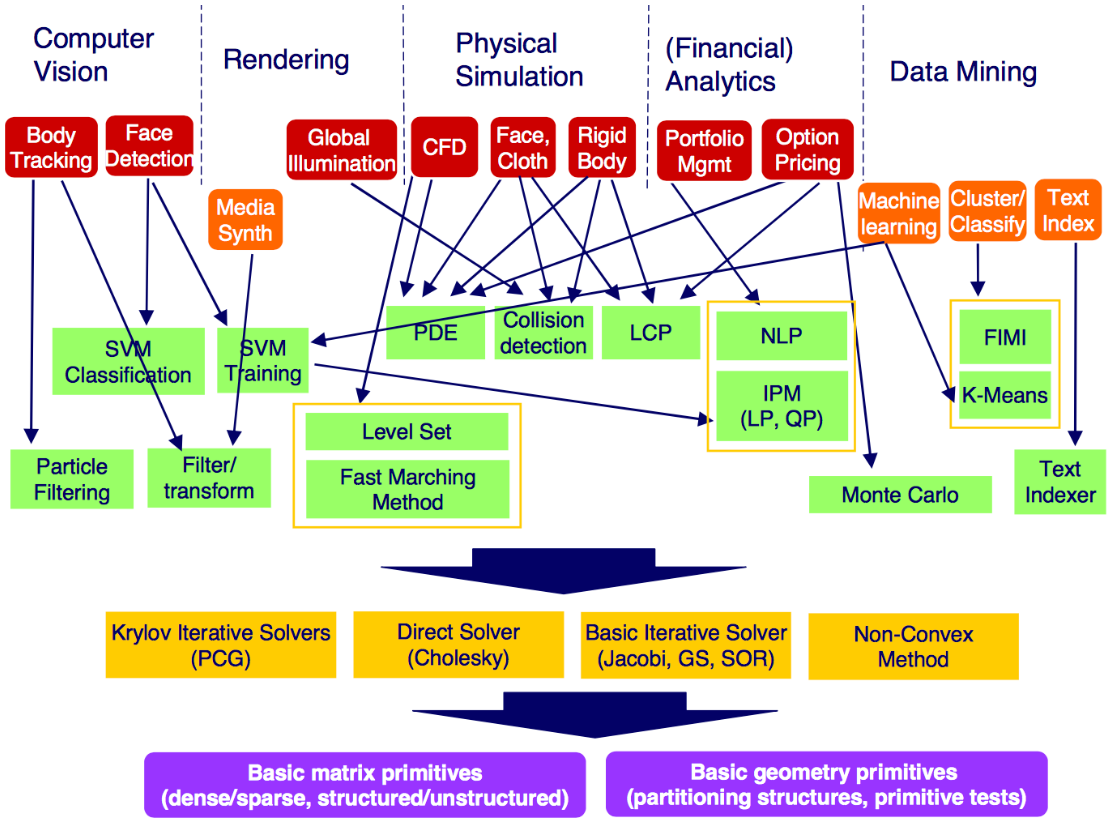
TSM2: Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs

Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao*, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben**, Qiang Guan***, Zizhong Chen

University of California, Riverside   **University of Alabama*
***Los Alamos National Laboratory*   ****Kent State University*

Linear algebra kernels are widely used

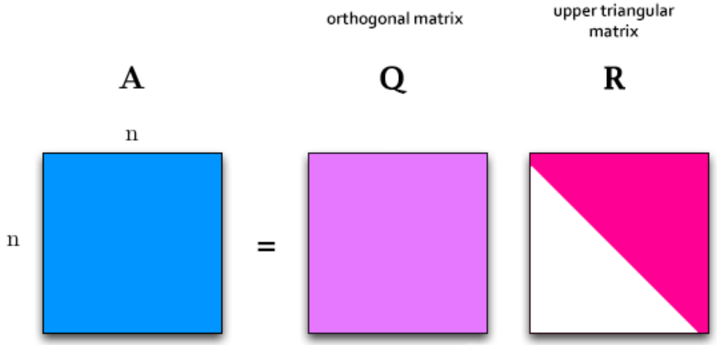
- Linear algebra kernels have been widely used.
  - *E.g., scientific simulation, big data analytics, machine leaning, etc.*
- Matrix-matrix multiplication (**GEMM**)
  - One of the most fundamental computation kernel that is used to build up other kernels
  - Core computation of many applications.
  - Cost most of the computation time of applications



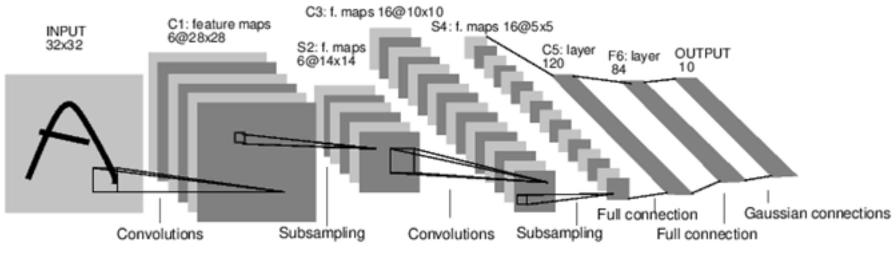
(Source: Berkeley Dwarfs Report)

Input shape of GEMM can varies from application to application

Relative regular shape input



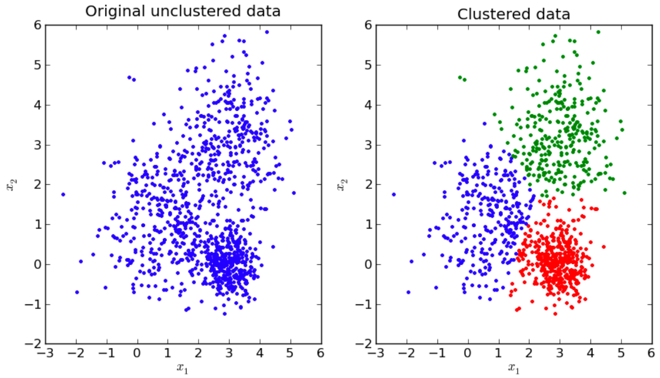
Dense Matrix Decompositions



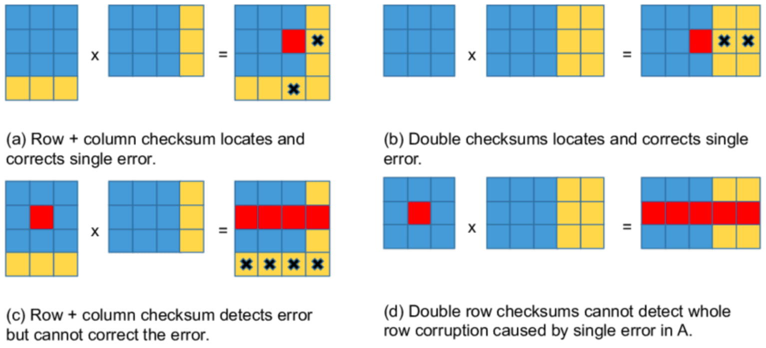
A Full Convolutional Neural Network (LeNet)

Deep Neural Networks

Tall-and-skinny shape input



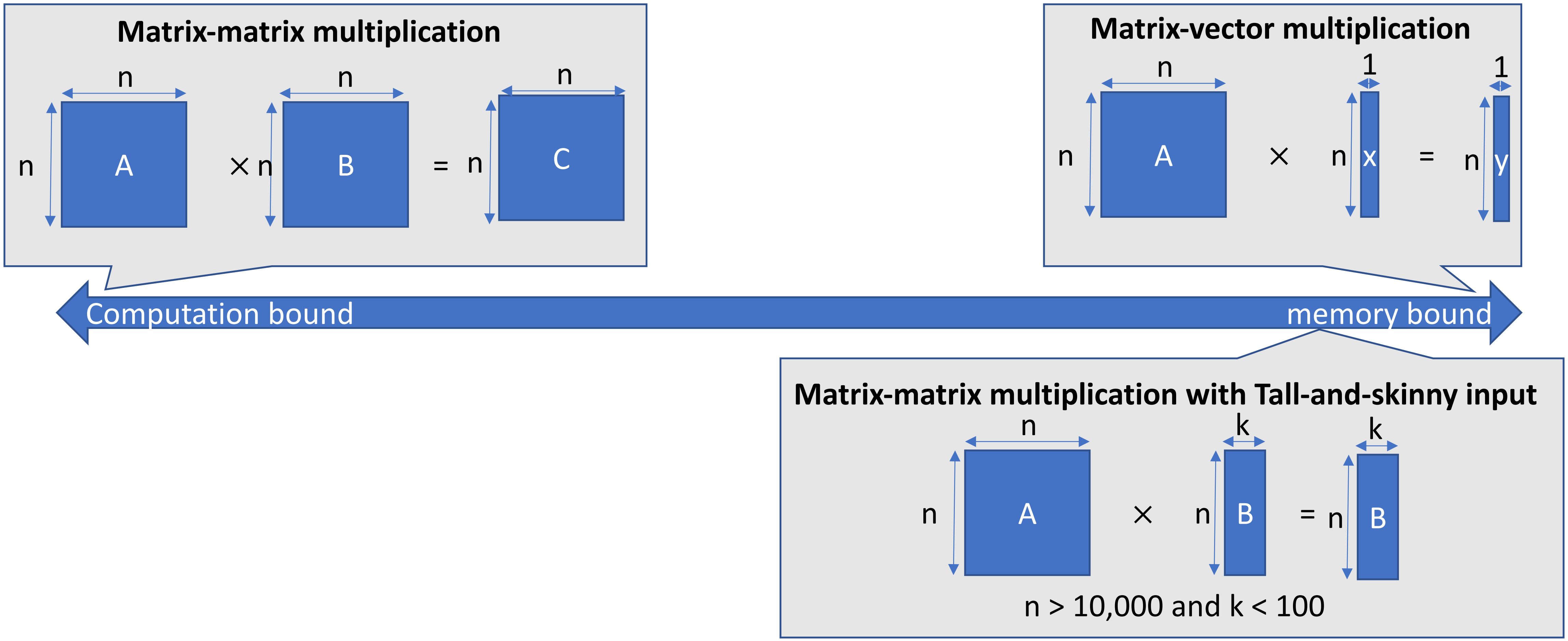
K-means



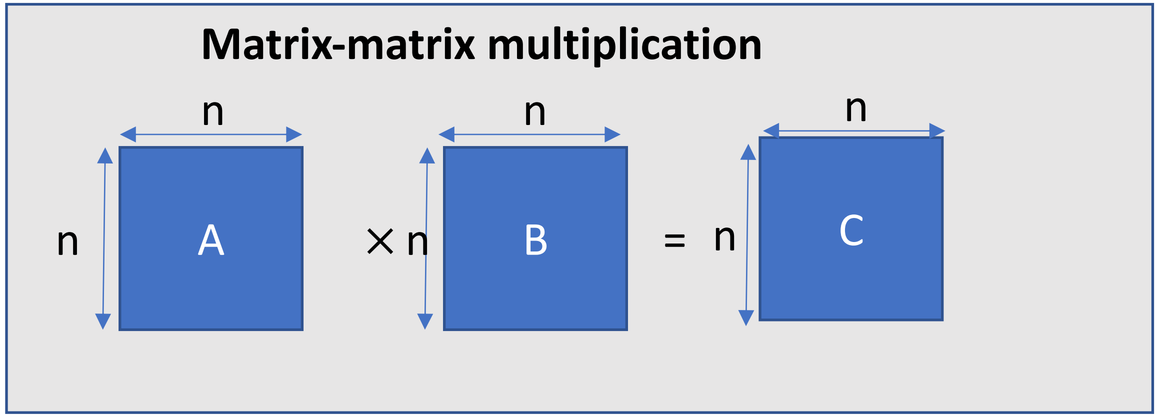
Algorithm Based Fault Tolerance

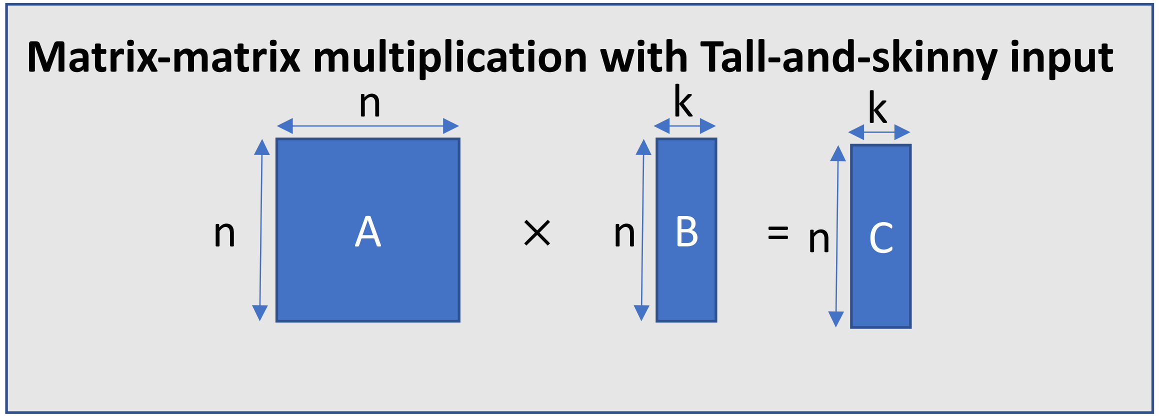
Two Kinds of Computations

- Computation bound → Performance of application is bounded by **the computation power**.
- Memory bound → Performance of application is bounded by **the memory bandwidth.**



Why tall-and-skinny behaves differently than regular shape input?



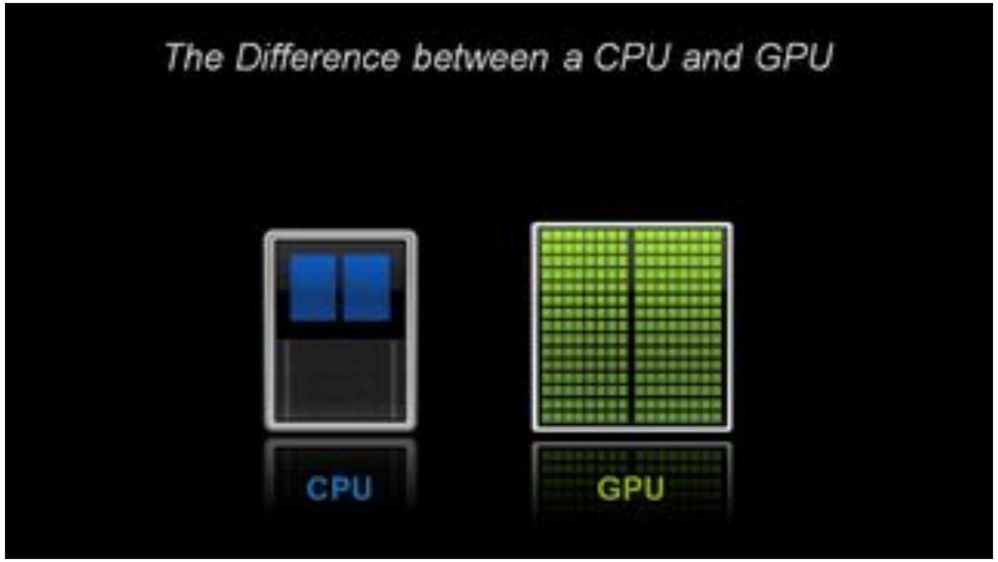


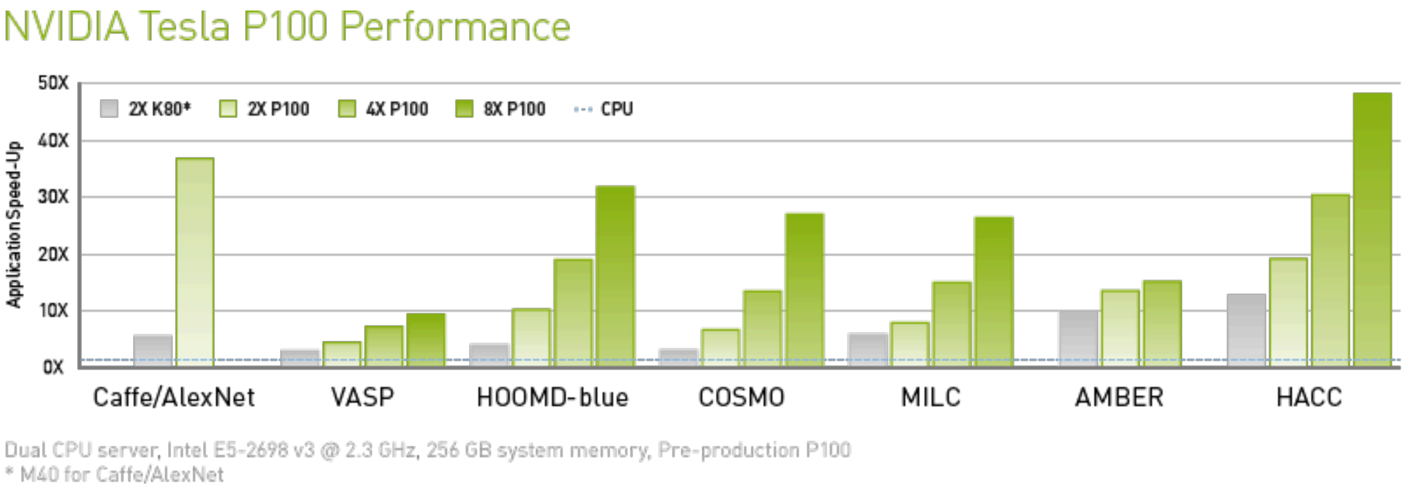
Input matrices size is $O(n^2)$.
Computing time complexity is $O(n^3)$.
Each element is used n times.

Input matrices size is $O(n^2)$.
Computing time complexity is $O(n^2k)$
Each element is used k times on average

- So for tall and skinny matrix input, depending on the k and the ratio between target GPU's peak computation power and peak memory throughput, it is usually memory bound.

GPUs are widely used for accelerating applications





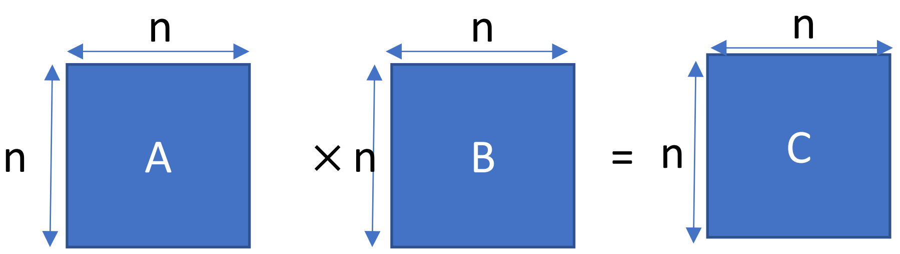
- Good at parallelized computations.
- Higher computation power and memory throughput.
- Commonly used for accelerating matrix-related computations.

cuBLAS library

- One of the most commonly used standard linear algebra libraries optimized for GPUs, which is developed by Nvidia.
- The core computing library of many big data and scientific computing applications.
- With deep optimization by Nvidia, the cuBLAS library is able to provide state-of-the-art performance in regular-shaped input matrix cases.
- But not fully optimized for tall-and-skinny matrix cases.

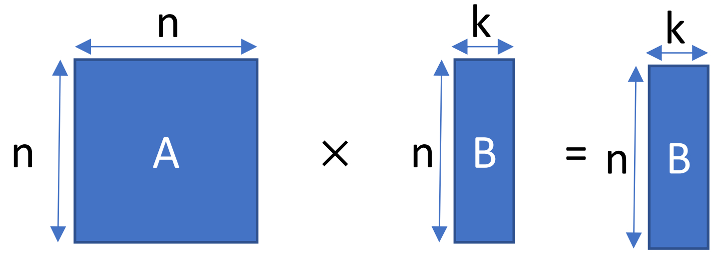
Poor Performance on Current State-of-the-Art Design:

Regular-sized matrix multiplication



With large n, k in similar magnitude

Tall-and-skinny matrix multiplication



n >> k

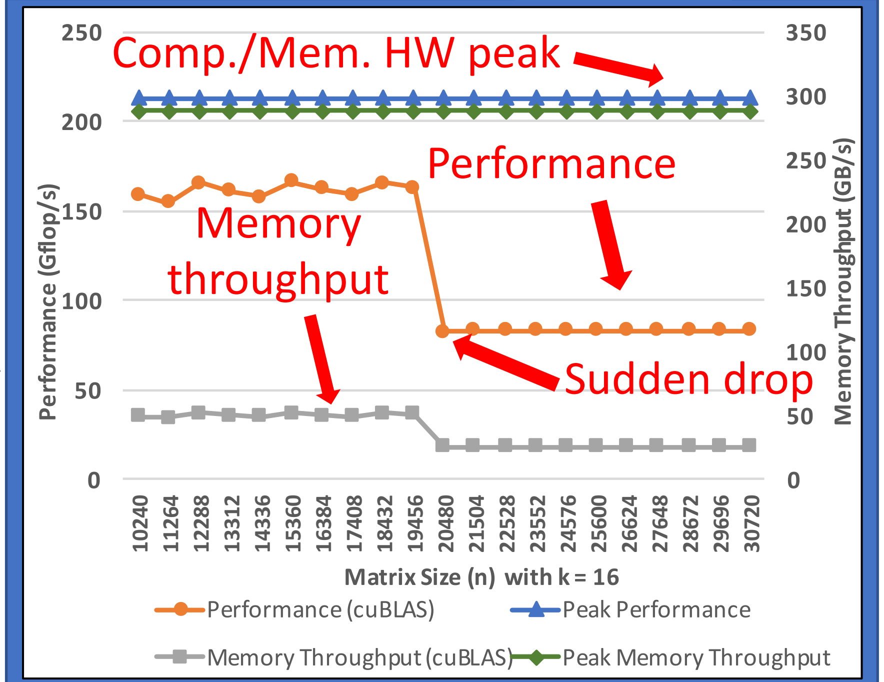
Regular size: 80%-90% of the peak computation power

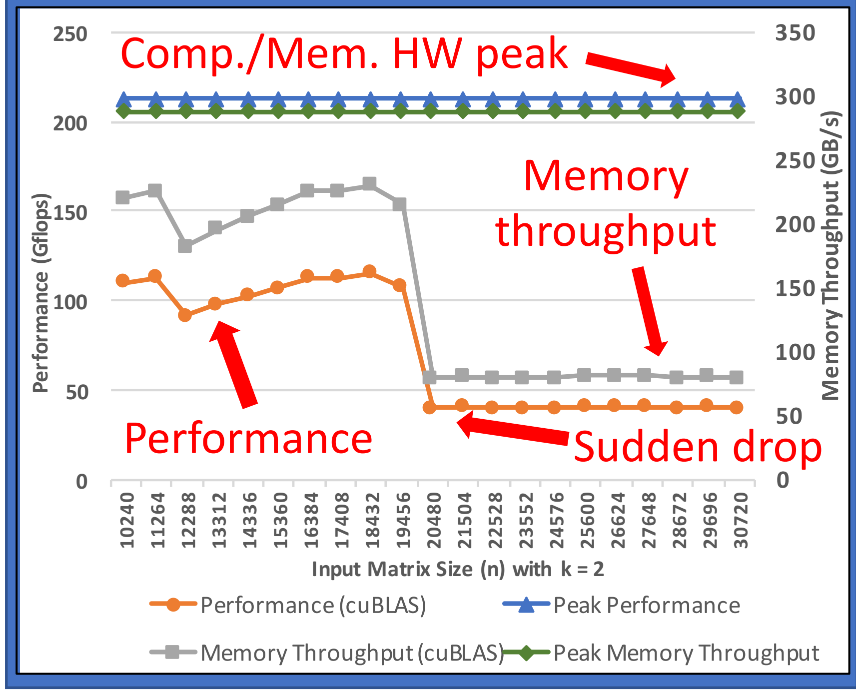
Computation bound ⟷ memory bound

Current state-of-the-art design only optimized for computation bound case

Low GPU utilization:

- K = 2:
  - 49.9% memory band.
  - 37.9% peak comp. power
- K=16:
  - 31.1% memory band.
  - 56.6% peak comp. power



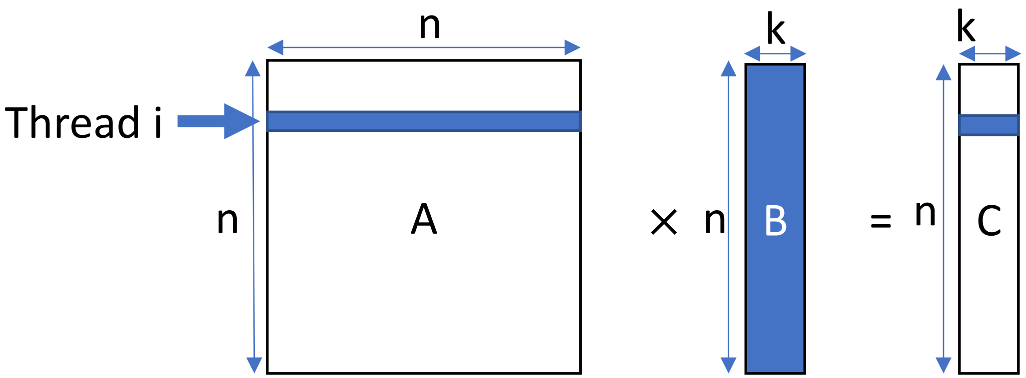


TSM2: redesigned matrix-matrix multiplication for tall-and-skinny input

- Several factors are considered:
  1) Total number of global memory accesses.
  2) Efficiency on global memory throughput.
  3) Parallelism of overall workload.
  4) On-chip memory utilization.
  5) Streaming Multiprocessor (SM) utilization.

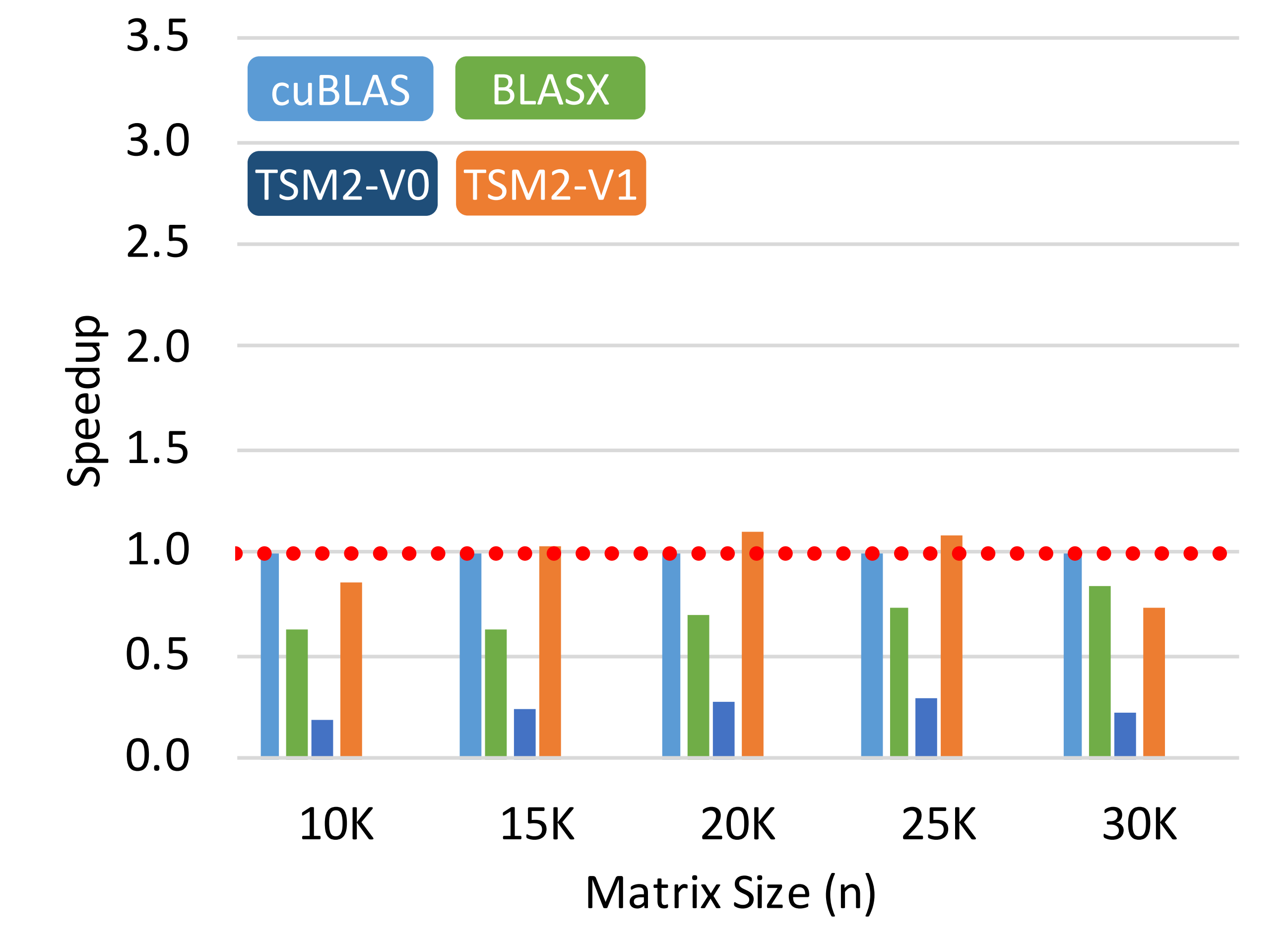
Algorithm design: how to fit the workload into the programming model of CUDA(Continued)

- We divide the workload by assigning n rows of matrix A to n different threads. Each vector-matrix multiplication is assigned to one thread.
    i. To ensure high parallelism and high Streaming Multiprocessor occupancy.
    ii. To ensure minimum number of memory access in favor of matrix A.
    iii. To enable high memory accesses efficiency.



Redesigning matrix-matrix multiplication for tall-and-skinny input

- Rethinking algorithm design – aiming to reduce total number of memory access
  - Inner produ



Tall-and-skinny GEMM with K=8 on Nvidia Tesla K40c

Versi

Require: input matrix
Require: output matri
1: **for** $i = 1$ to k **do**
2: **for** $j = 1$ to n **do**
3: $C[thread\_id,$
4: **end for**
5: **end for**
Algorithm 1: Work

Outer Product

) and B ($n \times k$)
k)

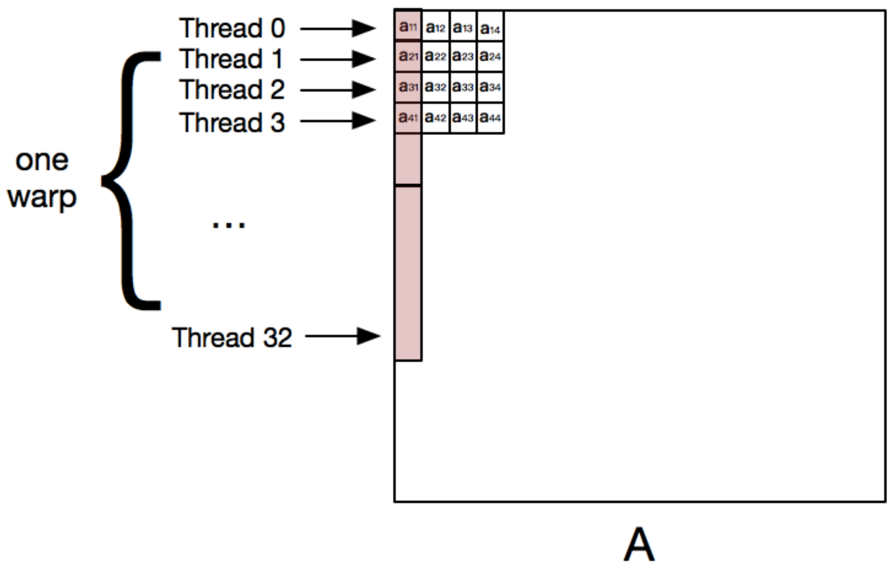
each thread with outer product

- Memory access
- Memory access to each element of B: n times
- Total number of accesses: $2kn^2$

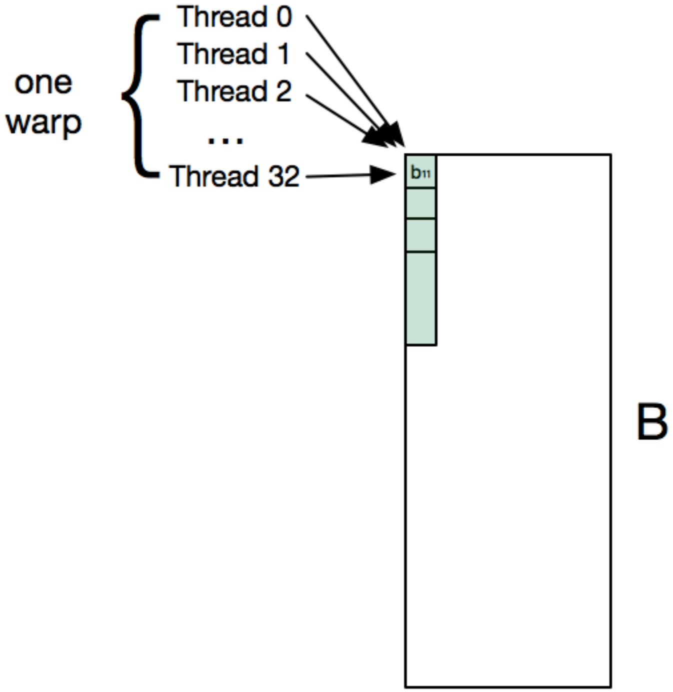
- Memory access to each element of A: 1 time
- Memory access to each element of B: n times
- Total number of accesses: $(k+1)n^2$

Global memory access efficiency analysis

- Global memory access efficiency per transaction = useful data/cache line size
  - Affect overall application memory access efficiency
  - Determined by the memory access pattern and the algorithm
  - Can be challenging to improve without modifying the algorithm design
- For outer product GEMM:



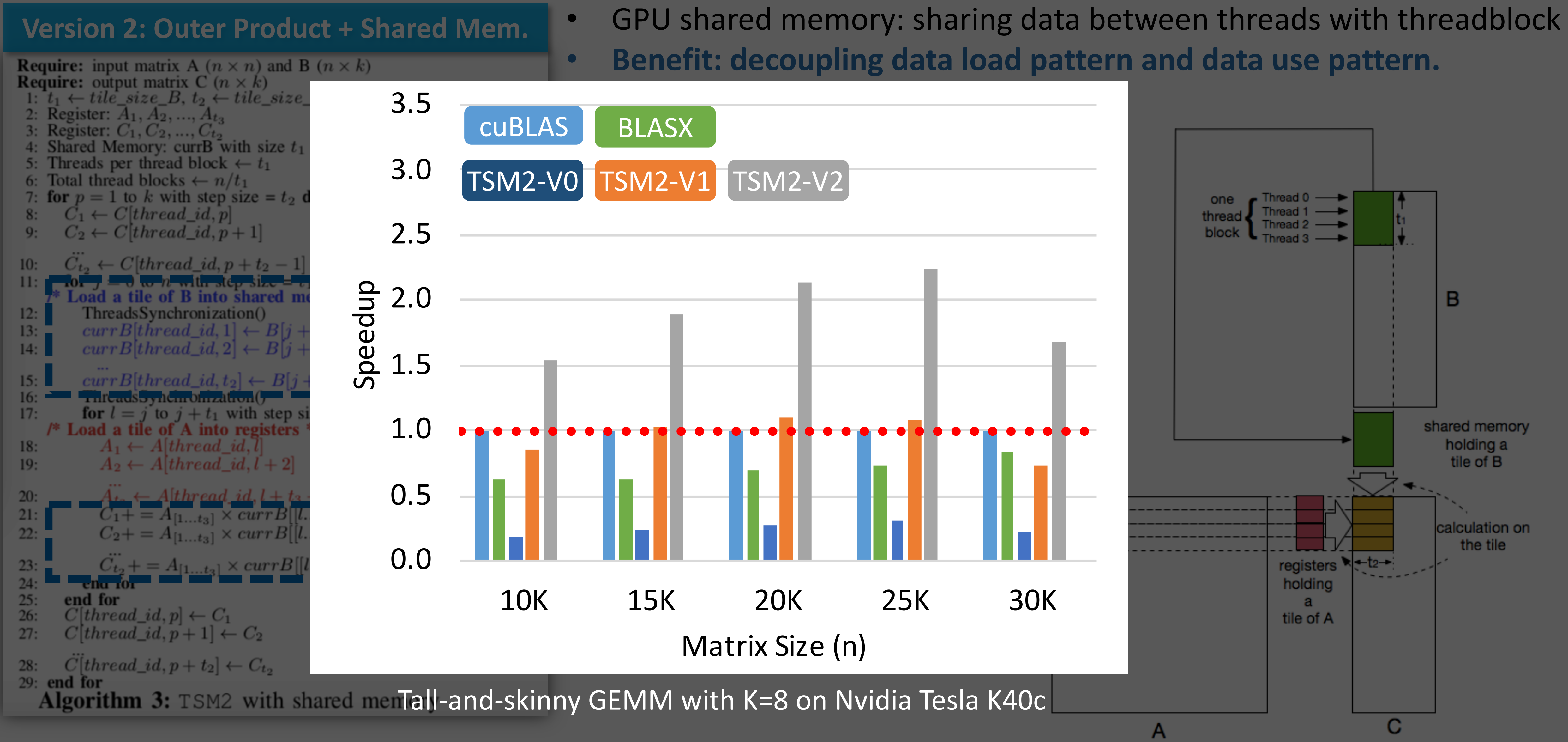
$$\frac{128\ bytes}{128\ bytes} = \mathbf{100\%}\ or\ \frac{32\ bytes}{32\ bytes} = \mathbf{100\%}$$



$$\frac{8\ bytes}{128\ bytes} = \mathbf{6.25\%}\ or\ \frac{8\ bytes}{32\ bytes} = \mathbf{25\%}$$

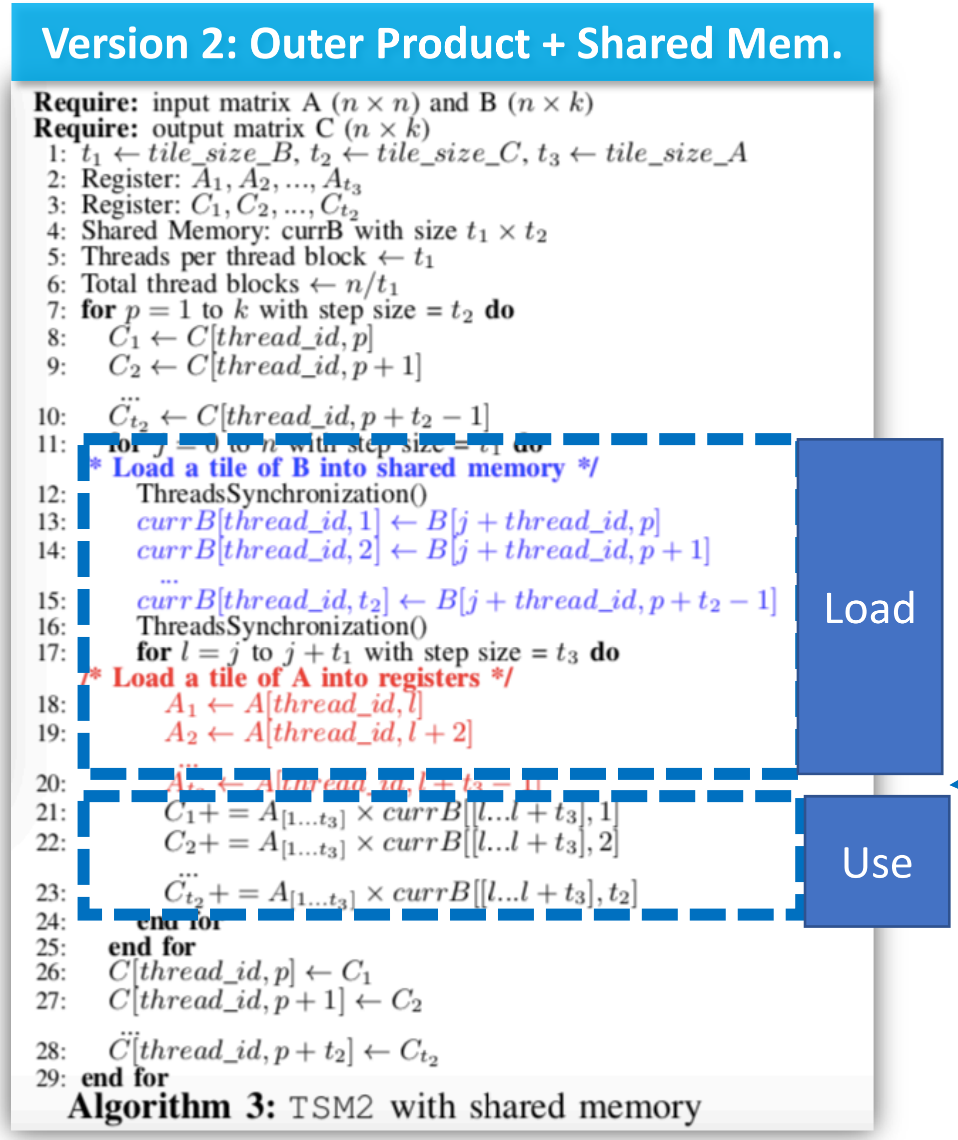
Improving global memory access efficiency

- GPU shared memory: sharing data between threads with threadblock
- **Benefit: decoupling data load pattern and data use pattern.**



Tall-and-skinny GEMM with K=8 on Nvidia Tesla K40c

Improving global memory access efficiency



Version 2: Outer Product + Shared Mem.

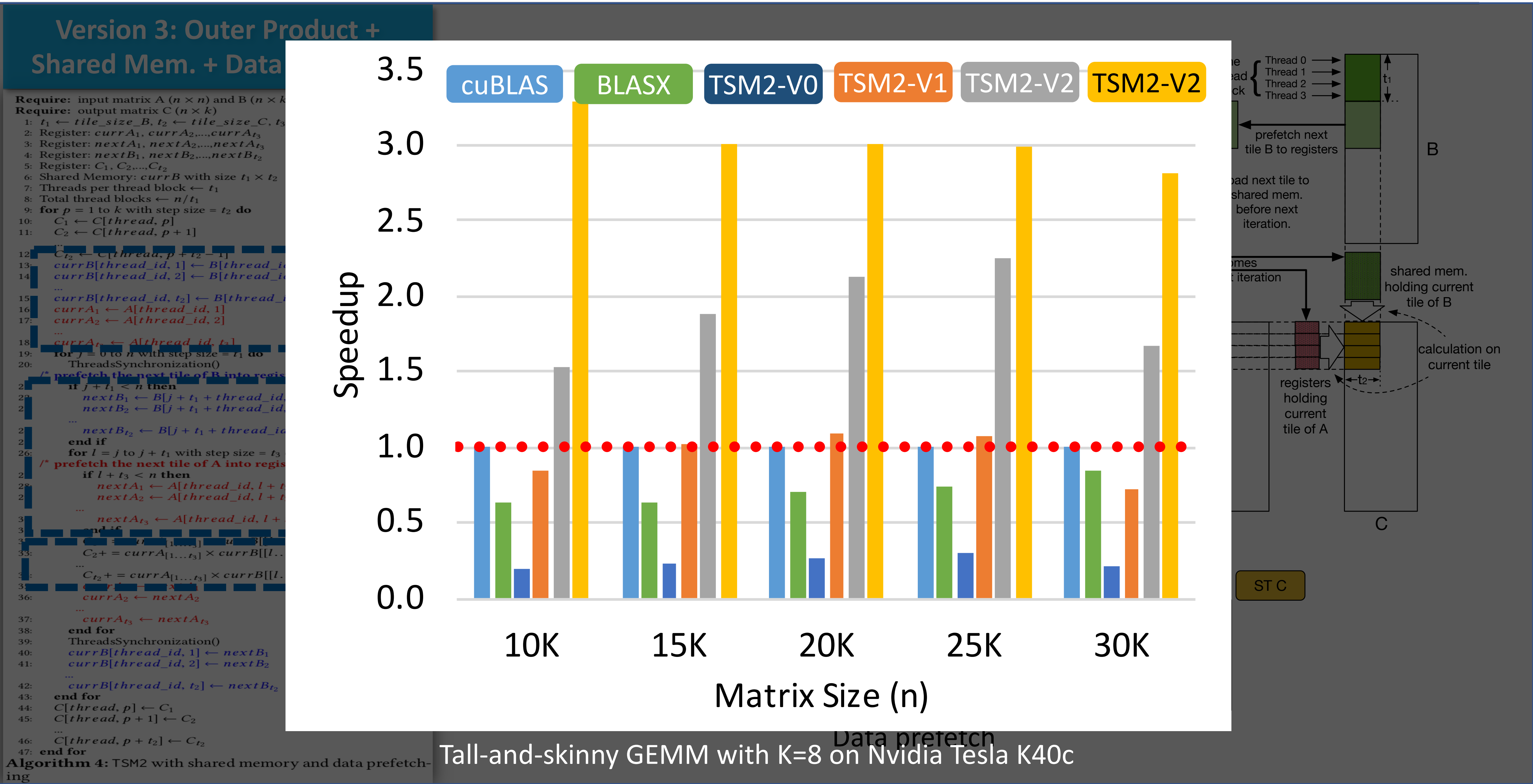
Require: input matrix A ($n \times n$) and B ($n \times k$)
Require: output matrix C ($n \times k$)
$t_1 \leftarrow tile\_size\_B$, $t_2 \leftarrow tile\_size\_C$, $t_3 \leftarrow tile\_size\_A$
Register: $A_1, A_2, ..., A_{t_3}$
Register: $C_1, C_2, ..., C_{t_2}$
Shared Memory: currB with size $t_1 \times t_2$
Threads per thread block $\leftarrow t_1$
Total thread blocks $\leftarrow n/t_1$
for $p = 1$ to k with step size = t_2 **do**
  $C_1 \leftarrow C[thread\_id, p]$
  $C_2 \leftarrow C[thread\_id, p+1]$
  ...
  $C_{t_2} \leftarrow C[thread\_id, p+t_2-1]$
  for $j = 0$ to n with step size = t_1 **do**
/* Load a tile of B into shared memory */
    ThreadsSynchronization()
    $currB[thread\_id, 1] \leftarrow B[j + thread\_id, p]$
    $currB[thread\_id, 2] \leftarrow B[j + thread\_id, p+1]$
    ...
    $currB[thread\_id, t_2] \leftarrow B[j + thread\_id, p+t_2-1]$
    ThreadsSynchronization()
    for $l = j$ to $j + t_1$ with step size = t_3 **do**
/* Load a tile of A into registers */
      $A_1 \leftarrow A[thread\_id, l]$
      $A_2 \leftarrow A[thread\_id, l+2]$
      ...
      $A_{t_3} \leftarrow A[thread\_id, l+t_3-1]$
      $C_1 += A_{[1...t_3]} \times currB[[l...l+t_3], 1]$
      $C_2 += A_{[1...t_3]} \times currB[[l...l+t_3], 2]$
      ...
      $C_{t_2} += A_{[1...t_3]} \times currB[[l...l+t_3], t_2]$
    end for
  end for
  $C[thread\_id, p] \leftarrow C_1$
  $C[thread\_id, p+1] \leftarrow C_2$
  ...
  $C[thread\_id, p+t_2] \leftarrow C_{t_2}$
end for

Algorithm 3: TSM2 with shared memory

Data dependency between data load and data use instructions

- Even with efficient global memory loading pattern, it still brings high GPU underutilization
  - Main cause: long memory access latency can be hard to hide.

Data prefetch: Improving GPU utilization

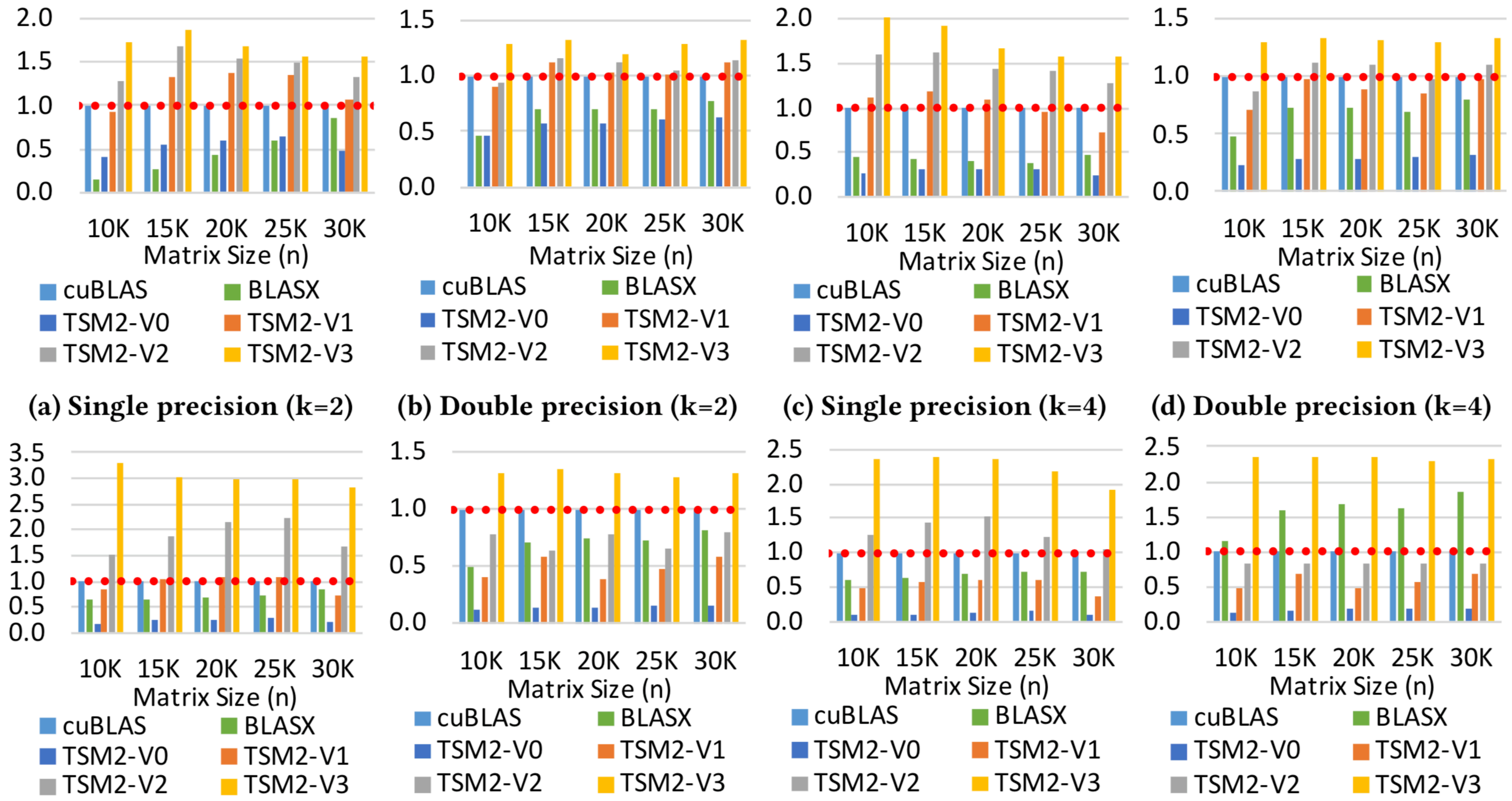


Tall-and-skinny GEMM with K=8 on Nvidia Tesla K40c

Experimental evaluation:

GPU Model	Micro-architectures	Memory	Peak performance	Peak memory bandwidth
Tesla K40c	Kepler	12 GB	1430 GFLOPS	288 GB/s
Tesla M40	Maxwell	24 GB	213 GFLOPS	288 GB/s
Tesla P100	Pascal	16 GB	4600 GFLOPS	720 GB/s

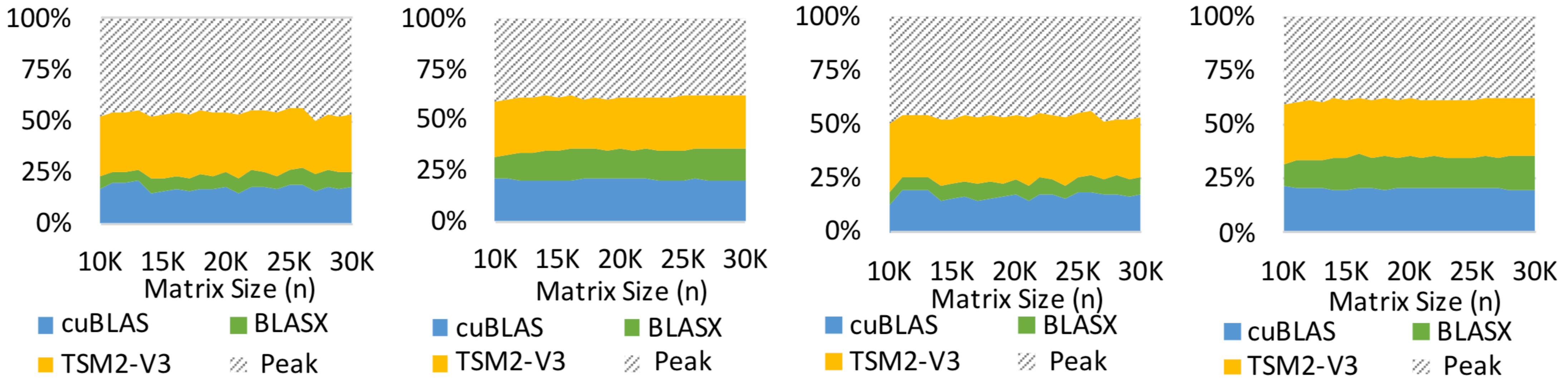
Experimental evaluation: Speedup (on Nvidia Tesla K40c)



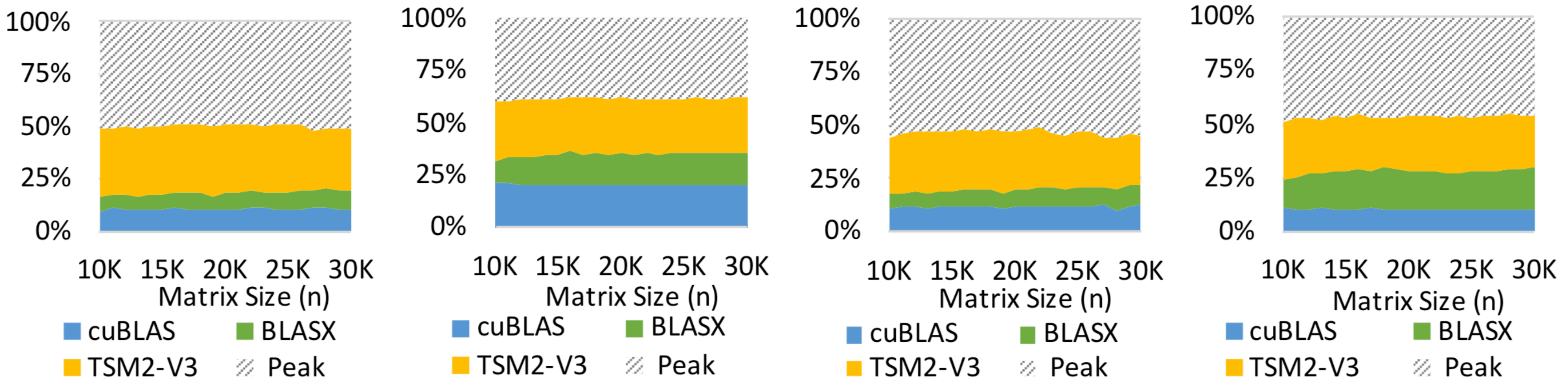
(a) Single precision (k=2) (b) Double precision (k=2) (c) Single precision (k=4) (d) Double precision (k=4)

(e) Single precision (k=8) (f) Double precision (k=8) (g) Single precision (k=16) (h) Double precision (k=16)

Experimental evaluation: Memory bandwidth (on Nvidia Tesla K40c)

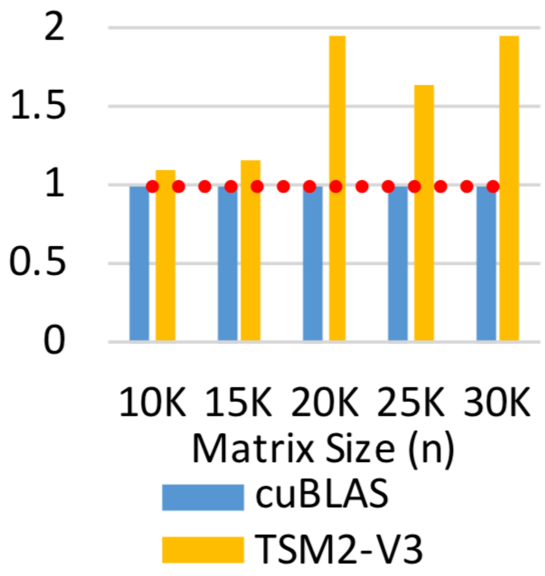


(a) Single precision (k=2) (b) Double precision (k=2) (c) Single precision (k=4) (d) Double precision (k=4)

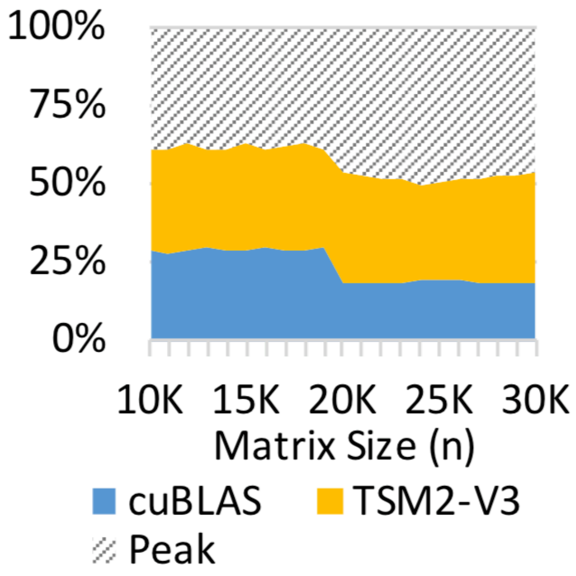


(e) Single precision (k=8) (f) Double precision (k=8) (g) Single precision (k=16) (h) Double precision (k=16)

Experimental evaluation on Nvidia Tesla M40 and P100

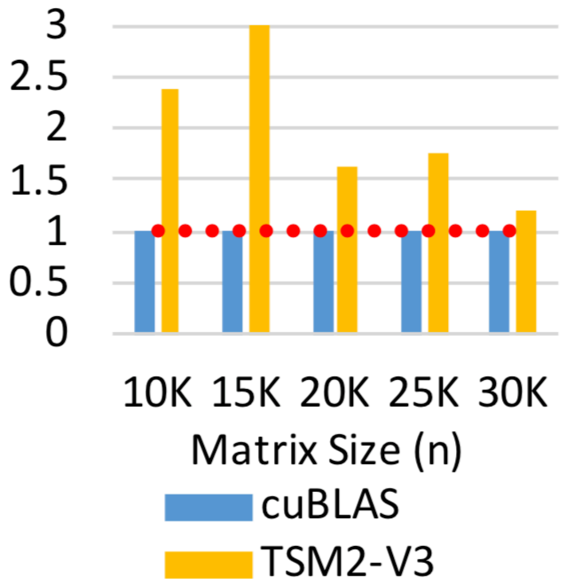


(a) Speedup

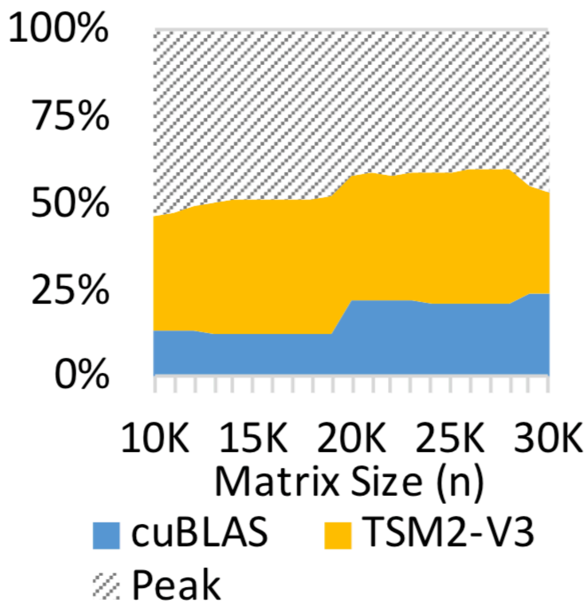


(b) Computational power util.

Tesla M40



(a) Speedup



(b) Memory throughput util.

Tesla P100

Showcase 1: K-means

Core computation of Lloyd's K-means: **distance calculation.**
Common choice: **Euclidean Distance**

$$||x - y||^2 = ||x||^2 + ||y||^2 - 2xy$$

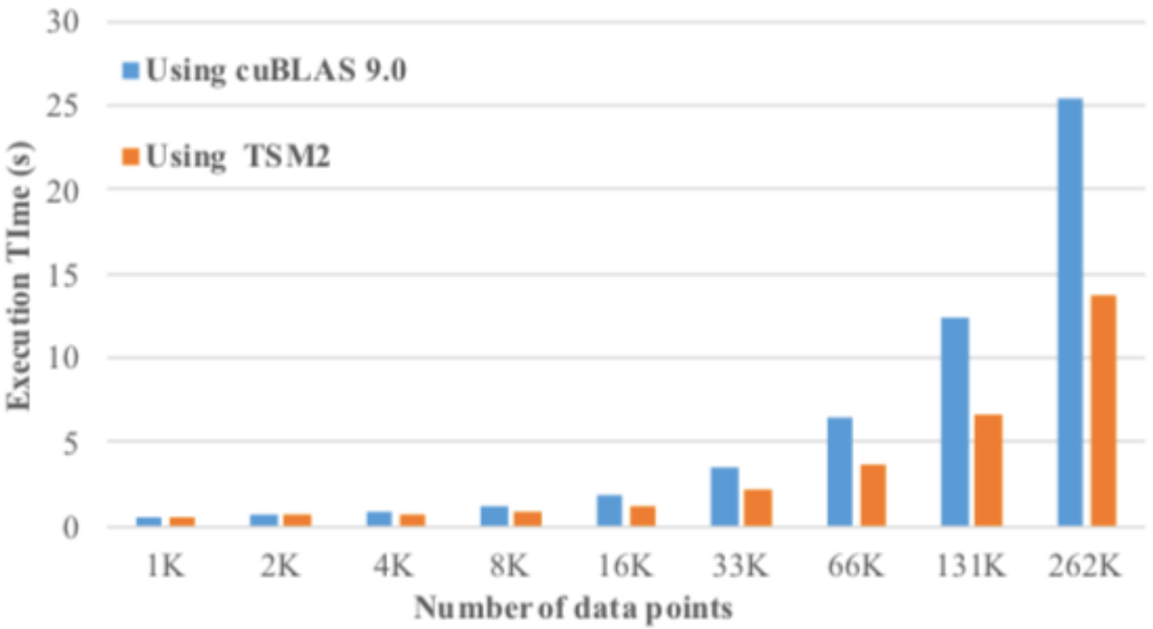
When we have multiple x and y:

Group x → matrix X
Group y → matrix Y

calculating xy → XY
(matrix matrix multiplication)

Calculating distance between:

- Data points X (n points with d dimensions);
- Centroids C (k centroids with d dimensions);
- → matrix-matrix multiplication: (n*d) times (d*k).
- Usually k << n,d → tall-and-skinny



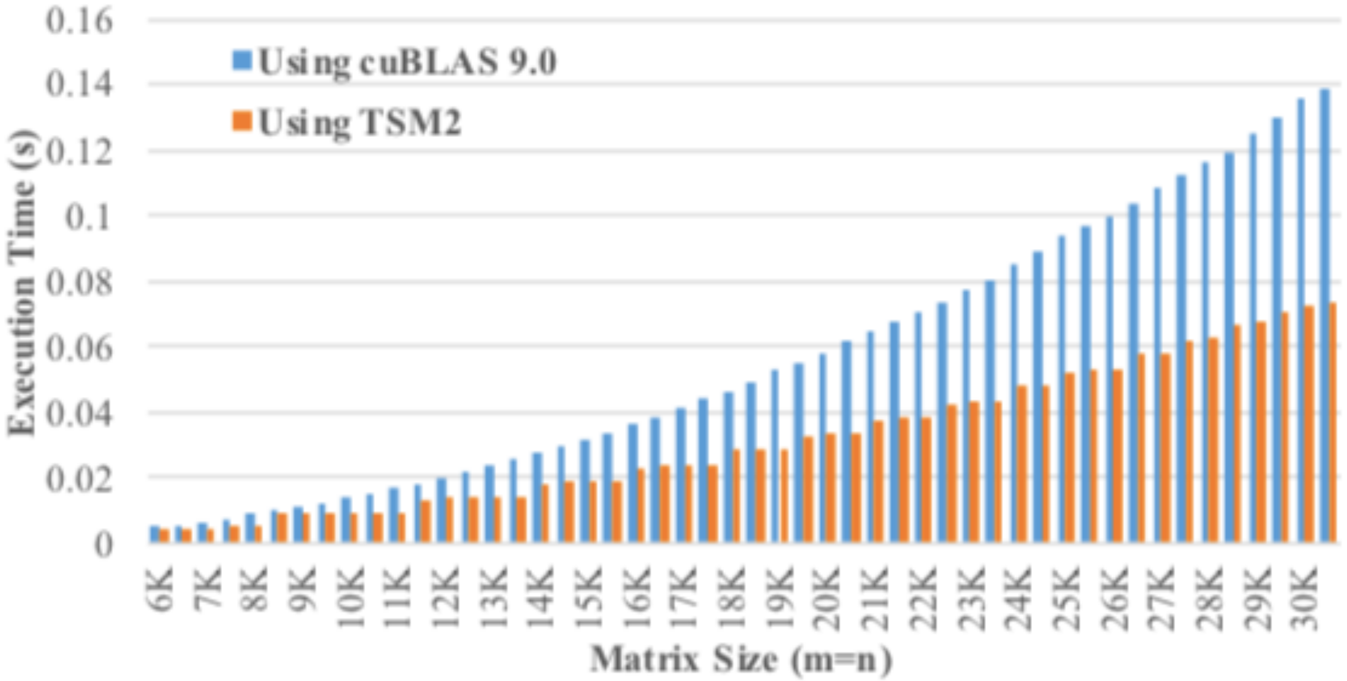
- Execution time of the first 100 iterations of Lloyd's K-means algorithm on K40c (d = 4096, k = 16).
- Using our TSM2, we speedup K- means by 1.06x - 1.89x (**avg. 1.53x**).
- GPU version K-means originally developed by NVIDIA: https://github.com/NVIDIA/kmeans

Showcase 2: *ABFT Matrix Checksum Encoding*

- Core computation of ABFT: **calculating checksum** *(encode redundant info)*
- E.g., calculate the checksum of matrix A with checksum weight vector v:

$$checksum(A) = Av$$

- Usually use multiple different checksum weight vectors.
- If we use c different checksum weight vectors → (m-by-n) times (n-by-c)
- Common choice: c = 2 << m,n → tall-and skinny



We compare the checksum encoding performance by using cuBLAS and TSM2 on K40c. As we can see, our TSM2 significantly improve the checksum encoding calculation with 1.10x to 1.90x speedup (**avg. 1.67x**).

Conclusion:

- We first analyzed the performance of current GEMM in the latest cuBLAS library.
- We discovered the potential challenges of optimizing tall-and-skinny GEMM since its workload is memory bound.
- We redesigned an optimized tall-and-skinny GEMM with several optimization techniques focusing on GPU resource utilization.
- Experiment results show that our optimized implementation can achieve better performance on three modern GPU micro-architectures.

We have an optimized design, but when do we use it?

Algorithm 4 Tall-and-skinny matrix-matrix multiplication with data prefetching

Require: input matrix A ($n \times n$) and B ($n \times k$)
Require: output matrix C ($n \times k$)

$t_1 \leftarrow tile\_size\_B$, $t_2 \leftarrow tile\_size\_C$, $t_3 \leftarrow tile\_size\_A$
Register: $currA_1, currA_2,...,currA_{t_3}$
Register: $nextA_1, nextA_2,...,nextA_{t_3}$
Register: $nextB_1, nextB_2,...,nextB_{t_2}$
Register: $C_1, C_2,...,C_{t_2}$
Shared Memory: $currB$ with size $t_1 \times t_2$
Threads per thread block $\leftarrow t_1$
Total thread blocks $\leftarrow n/t_1$
for $p = 1$ to k with step size = t_2 **do**
  $C_1 \leftarrow C[thread, p]$
  $C_2 \leftarrow C[thread, p+1]$
  ...
  $C_{t_2} \leftarrow C[thread, p + t_2 - 1]$
  $currB[thread\_id, 1] \leftarrow B[thread\_id, p]$
  $currB[thread\_id, 2] \leftarrow B[thread\_id, p+1]$
  ...
  $currB[thread\_id, t_2] \leftarrow B[thread\_id, p + t_2 - 1]$
  $currA_1 \leftarrow A[thread\_id, 1]$
  $currA_2 \leftarrow A[thread\_id, 2]$
  ...
  $currA_{t_3} \leftarrow A[thread\_id, t_3]$
  for $j = 0$ to n with step size = t_1 **do**
    ThreadsSynchronization()
/* prefetch the next tile of B into registers */
    if $j + t_1 < n$ **then**
      $nextB_1 \leftarrow B[j + t_1 + thread\_id, p]$
      $nextB_2 \leftarrow B[j + t_1 + thread\_id, p+1]$
      ...
      $nextB_{t_2} \leftarrow B[j + t_1 + thread\_id, p + t_2 - 1]$
    end if
    for $l = j$ to $j + t_1$ with step size = t_3 **do**
/* prefetch the next tile of A into registers */
      if $l + t_3 < n$ **then**
        $nextA_1 \leftarrow A[thread\_id, l + t_3]$
        $nextA_2 \leftarrow A[thread\_id, l + t_3 + 1]$
        ...
        $nextA_{t_3} \leftarrow A[thread\_id, l + t_3 + t_3 - 1]$
      end if
      $C_1 += currA_{[1...t_3]} \times currB[[l...l + t_3], 1]$
      $C_2 += currA_{[1...t_3]} \times currB[[l...l + t_3], 2]$
      ...
      $C_{t_2} += currA_{[1...t_3]} \times currB[[l...l + t_3], t_2]$
      $currA_1 \leftarrow nextA_1$
      $currA_2 \leftarrow nextA_2$
      ...
      $currA_{t_3} \leftarrow nextA_{t_3}$
    end for
    ThreadsSynchronization()
    $currB[thread\_id, 1] \leftarrow nextB_1$
    $currB[thread\_id, 2] \leftarrow nextB_2$
    ...
    $currB[thread\_id, t_2] \leftarrow nextB_{t_2}$
  end for
  $C[thread, p] \leftarrow C_1$
  $C[thread, p+1] \leftarrow C_2$
  ...
  $C[thread, p + t_2] \leftarrow C_{t_2}$
end for

Tuning parameters

How to determine when the computation is memory bound and when it is not?

GPU Peak Perf.

GPU Peak Mem. Band.

Hardware parameters

Algorithm 5 Parameter Optimization for Tall-and-skinny matrix-matrix multiplication

if $k \leq t_2^{threshold}$ **then**
  $Total\_memory \approx n \times n \times \frac{k}{t_2} \times bytes\_per\_elem.$
  $Bandwidth = PeakBand. \times Util_{mem}$
  Use Gradient Descent to Optimize (t_2 and t_3): $Time = \frac{Total\_memory}{Bandwidth}$ with $1 \leq t_2 \leq k$ and $1 \leq t_3$
  Output: t_2 and t_3
else
  $Total\_flops = n \times n \times k \times 2$
  $Compute\_power = PeakPerf. \times Util_{comp}$
  Use Gradient Descent to Optimize (t_2 and t_3): $Time_1 = \frac{Total\_flops}{Compute\_power}$ with $t_2^{threshold} \leq t_2 \leq k$ and $1 \leq t_3$
  $t_{2(time_1)} \leftarrow t_2$
  $t_{3(time_1)} \leftarrow t_3$
  $Total\_memory \approx n \times n \times \frac{k}{t_2} \times bytes\_per\_elem.$
  $Bandwidth = PeakBand. \times Util_{mem}$
  Use Gradient Descent to Optimize (t_2 and t_3) in $Time_2 = \frac{Total\_memory}{Bandwidth}$ with $1 \leq t_2 \leq t_2^{threshold}$ and $1 \leq t_3$
  $t_{2(time_2)} \leftarrow t_2$
  $t_{3(time_2)} \leftarrow t_3$
  if $Time_1 < Time_2$ **then**
    Output: $t_{2(time_1)}$ and $t_{3(time_1)}$
  else
    Output: $t_{2(time_2)}$ and $t_{3(time_2)}$
  end if
end if