Tall Skinny QR decomposition

Pudit Tempattarachoke and Chutchatut Sutichavengkul

1 Summary

Overdetermined linear systems show up everywhere, and the bottleneck is usually the QR factorization. When the matrix is tall and skinny, classical Householder QR wastes parallelism because it is fundamentally sequential column by column. TSQR avoids that by organizing the factorization as a reduction tree, which exposes parallel work at every level and keeps communication costs low. For many downstream tasks, we only need the final R, not the full Q. That lets us ignore the expensive parts of the algorithm and focus strictly on the reduction path that produces R. We built three implementations to exploit different hardware models: an OpenMP version for shared memory, an OpenMPI version for distributed memory, and a CUDA version for GPUs. The project is available at `https://pudit.github.io/TSQR/`.

2 Background

Solving overdetermined linear systems relies on computing a QR factorization [1] of a tall and skinny matrix. For these shapes, the dominant cost is applying Householder reflectors one panel at a time, which is inherently sequential and leaves a lot of parallel capacity unused. TSQR [2] restructures the computation as a binary reduction over block rows. Each leaf node holds a block of the matrix. Every internal node merges two local R factors by stacking them and performing a small QR to produce a new R. This continues up the tree until the root produces the final R factor. The inputs are the block rows, and the output is a single upper triangular R. The key data structures are the local blocks and the intermediate R factors. The expensive part is the repeated QR on stacked R matrices at each merge step, but these operations are independent across tree levels, which exposes clean parallelism.

Dependencies only flow upward in the tree. Nodes at the same level never depend on each other, so the workload is naturally data parallel. Each merge touches only two small matrices, giving good locality. The operations are dense linear algebra on contiguous blocks, which maps well to SIMD units, thread-level parallelism on CPUs, and massive parallelism on GPUs. This structure is also communication efficient since the only data passed up the tree is an R factor, not a full block row.

3 Approach

This section describes the R-only TSQR pipeline implementation for three targets, the rationale behind its structure, data mapping to cores/threads/warps, modifications from naive serial QR, and optimization iterations.

3.1 Implementation Summary

All implementations are in C++ with small platform-specific kernels; no full-size vendor QR libraries are used.

- **OpenMP (shared memory)**: Uses `omp` pragmas for parallel work distribution and synchronization.
- **OpenMPI (distributed memory)**: Implements binary-tree reduction across MPI ranks using `MPI_Scatter`, `MPI_Send`, and `MPI_Recv`.
- **CUDA (GPU)**: Performs block-level local QR and a sequence of device kernels reducing stacked R blocks. The device-side QR is a Householder-based routine (`qr_device`).

3.2 Algorithmic Mapping

The TSQR uses a binary reduction tree. Matrix A is split into block rows. Each leaf computes a local QR on its block, storing only the $n \times n$ R. Internal nodes stack two child R matrices into a $2n \times n$ matrix and compute a small QR to produce a new R. The root R is the final output.

Inputs: full tall-and-skinny matrix A partitioned into block rows. **Outputs:** single upper-triangular R of size $n \times n$. **Intermediate values:** $n \times n$ R factors passed up the tree.

3.3 Key Data Structures

Local block rows: contiguous row-major buffers. Intermediate R buffers: contiguous arrays of size n^2. Ping-pong buffers: avoid allocation per merge step.

3.4 OpenMP Mapping

Partitioning: Matrix split into `num_threads` equal-height blocks; one block per thread.

Merge Phase: Iterative rounds: threads process independent merge pairs, stacking two $n \times n$ R matrices, performing a small QR, and writing the result to the alternate buffer. Barrier synchronization follows each round; a single thread swaps buffers and halves the active counter until one R remains.

Locality and SIMD: Barriers synchronize rounds; contiguous memory provides cache locality. Small $n \times n$ QR kernels fit in L1/L2 and vectorize well, using dense BLAS-1/2-style operations.

Implementation Details: Two preallocated buffers (`R_buffer1`, `R_buffer2`) implement ping-pong. Local QR writes to per-thread slices.

3.5 MPI Mapping

Partitioning: Matrix distributed via `MPI_Scatter`. Each rank computes a local QR and holds a local $n \times n$ R.

Tree Reduction: Binary tree: sending ranks transmit R to partner ranks; receivers stack two Rs, compute a QR, and produce a new R. Rounds continue until rank 0 holds the final R.

Constraints: Power-of-two world size, equal partitioning. Buffers sized $2n^2$ to receive partners' Rs. MPI_Finalize is called at the end.

3.6 CUDA Mapping

Kernel Decomposition: `qr_on_blocks` assigns one thread per block row; each runs `qr_device` producing an $n \times n$ R in global memory.

Reduction Kernels: `qr_on_stacks` merges two consecutive R blocks in shared memory, producing a new R. Host launches kernel iteratively until one R remains.

Memory and Performance: Shared memory checked against `THREAD_PER_BLOCK` times per-thread scratch memory. Global memory accesses are coalesced. QR on $2n \times n$ is compute-bound and fits in fast shared memory for small n.

Householder Kernel: `qr_device` implements classical Householder QR, explicitly computing norms, vector scaling, $I - 2vv^T$, and $H * R$ multiplies.

Constraints: Block-based splitting, round-up division, and sufficient shared memory per thread.

3.7 Changes from Serial QR

- Q is not formed, reducing memory and communication overhead.
- Tree-shaped reduction replaces serial Householder sweep; exposes leaf-level and internal parallelism.
- Power-of-two sizes and equal blocks simplify indexing and scheduling.

3.8 Optimization Iterations and Lessons

- Forming full Q caused memory and communication blow-up; dropped Q to focus on R.
- Ping-pong buffers replaced per-round allocations for performance.

3.9 Performance Characteristics

- **Parallelism:** Data-parallel across block rows and merge pairs; log(p) synchronization rounds.
- **Locality:** Dense, contiguous memory enables high cache efficiency.
- **Communication:** Only $n \times n$ R factors communicated; total $O(\log_2(p))$ rounds.
- **Load Balance:** Equal block partitioning ensures balance; remainder scheme not implemented.

3.10 Correctness, Stability, and Limitations

- Correctness: Stacking R_1 and R_2 and QR factorizing produces equivalent R as serial TSQR.
- Stability: Householder-based merges preserve numerical stability; tree order affects round-off only.
- Limitations: Power-of-two processor counts, m divisible by number of blocks, sufficient GPU shared memory required. Small-kernel QR used; highly tuned libraries could outperform for large n.
- Input Size Limitation: The GPU input size is limited compared to OpenMP and MPI because each thread computes partial results entirely in shared memory. This constraint restricts the maximum block width to 4 in our experiments. Increasing the width would require reducing `THREAD_PER_BLOCK`, but this is already 32; lowering it would reduce efficiency due to non-full warp utilization.

3.11 Code Provenance

Custom `qr` and `qr_cuda` implementations. Evolved in-house, not derived from external TSQR libraries.

3.12 Assumptions and Parameters

- `num_processors/world_size` is a power of two.
- m divisible by number of blocks; block_height = m/p.
- `THREAD_PER_BLOCK` sets thread-to-block mapping and shared memory usage checks.

4 Results

4.1 OpenMP parallelization

4.1.1 Speedup vs Number of Threads

Data and definitions
We define the *total speedup* measured experimentally as

$$S_{\text{total}}(p, n) \;=\; \frac{T(1, n)}{T(p, n)},$$

where $T(p, n)$ is the measured runtime (ms) for p threads and matrix width n. The dataset used (times in ms) corresponds to $m = 1600$ and $n \in \{2, 4, 8, 16, 32\}$:

Threads	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	19222	38725	77728	156111	311765
2	1057	2077	4142	8404	17241
4	114	232	470	955	1988
8	15	28	56	117	271
16	4	8	17	33	97
32	2	4	4	15	81

From this, the total speedups (rounded) are:

Threads	$S_{\text{total}}(p, 2)$	$S_{\text{total}}(p, 4)$	$S_{\text{total}}(p, 8)$	$S_{\text{total}}(p, 16)$	$S_{\text{total}}(p, 32)$
1	1.0	1.0	1.0	1.0	1.0
2	18.2	18.6	18.8	18.6	18.1
4	168.7	167.1	165.2	163.4	156.9
8	1281.5	1383.0	1388.0	1334.0	1150.2
16	4805.5	4840.6	4572.2	4727.6	3213.6
32	9611.0	9681.3	19432.0	10407.4	3849.5

Plots: Total Speedup and Computation Speedup

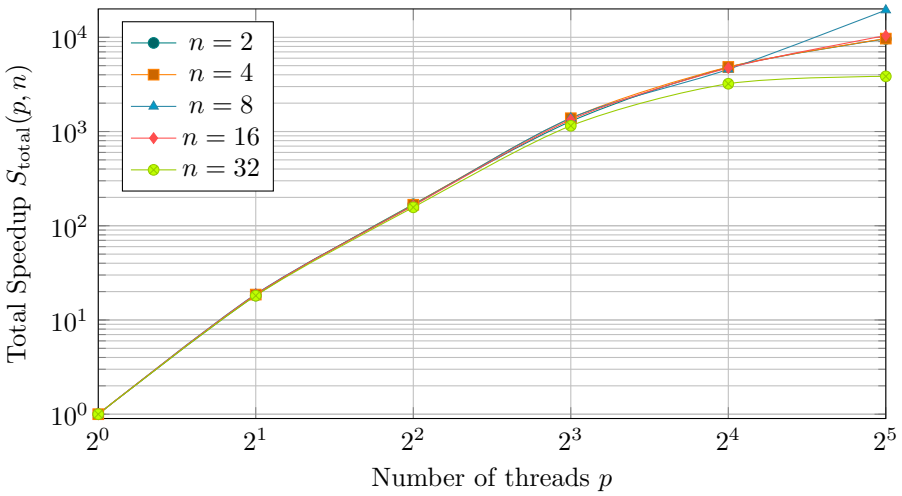

Figure 1: Measured **Total Speedup** $S_{\text{total}}(p, n)$ vs number of threads for each matrix width n. A log-log scaling is used because the measured speedups span many orders of magnitude.

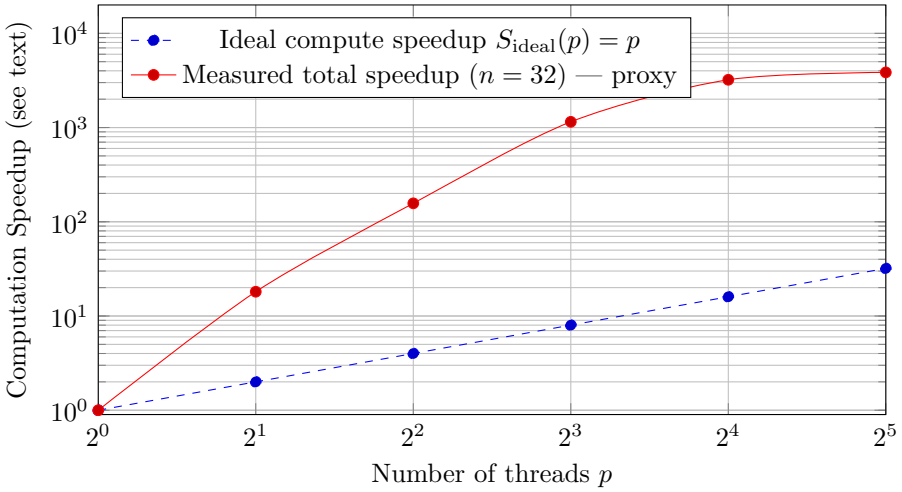

Figure 2: Comparison of ideal, linear **computation** speedup and the measured total speedup for $n = 32$. The measured curve is shown as a proxy to illustrate that the observed acceleration is much larger than pure compute-parallelism (see discussion).

Discussion and interpretation

Total speedup shape. The *total speedup* curves show extremely large increases as thread count grows. For each fixed matrix width n the measured speedup values are orders of magnitude larger than the corresponding number of threads p. Even at modest thread counts, the ratio $T(1)/T(p)$ becomes huge.

Why speedup is *superlinear*. The measured speedups are *superlinear* (i.e. $S_{\text{total}}(p) \gg p$). This arises from a combination of effects:

- **Algorithmic difference between serial and parallel code.** The parallel implementation uses TSQR (a tree-reduction QR suited to tall-skinny matrices) whereas the single-thread baseline is a plain serial QR. TSQR reorganizes work and reduces global memory traffic, performing fewer expensive memory accesses and fewer global synchronizations.
- **Cache and memory hierarchy effects.** Each thread works on a smaller local block which often fits in L1/L2 cache. This reduces cache misses and memory latency compared to the monolithic serial run.
- **Measurement baseline issues.** If the single-thread measurement includes extra overhead (initialization, I/O) that the parallel measurements avoid or amortize, the speedup will appear inflated.

Computation speedup panel and what it means. The second figure overlays the ideal linear compute speedup $S_{\text{ideal}}(p) = p$ with the measured total speedup for $n = 32$. Because we did not separately instrument computation vs communication, we cannot draw a true computation-only curve. The measured total speedup lies far above the ideal line, confirming that the observed acceleration comes from a combination of algorithmic improvements and hardware-cache effects. For tall-skinny matrices and many processes, the cost of MP communication can dominate, reducing scaling efficiency as the performance speed up when $n = 32$ does not superlinear and follow the previous iterations of trends.

4.2 OpenMPI parallelization

4.2.1 Speedup vs Number of Processors

Data and definitions
We define the *total speedup* measured experimentally as

$$S_{\text{total}}(p, n) \;=\; \frac{T(1, n)}{T(p, n)},$$

where $T(p, n)$ is the measured runtime (ms) for p MPI processes and matrix width n. The dataset used (times in ms) corresponds to $m = 1600$ and $n \in \{2, 4, 8, 16, 32\}$:

Processors	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	14656	29482	58859	118389	238073
2	1054	2017	3929	7856	15949
4	236	355	644	1106	2155
8	178	192	220	291	445
16	207	235	210	251	270
32	426	432	430	418	401

From this, the total speedups are:

Processors	$S_{\text{total}}(p, 2)$	$S_{\text{total}}(p, 4)$	$S_{\text{total}}(p, 8)$	$S_{\text{total}}(p, 16)$	$S_{\text{total}}(p, 32)$
1	1.0	1.0	1.0	1.0	1.0
2	13.9	14.6	15.0	15.1	14.9
4	62.0	83.0	91.5	107.0	110.4
8	82.3	153.7	267.5	406.9	535.0
16	70.8	125.4	280.3	471.9	881.7
32	34.4	68.3	136.9	283.3	593.2

Total Speedup vs. Number of Processors

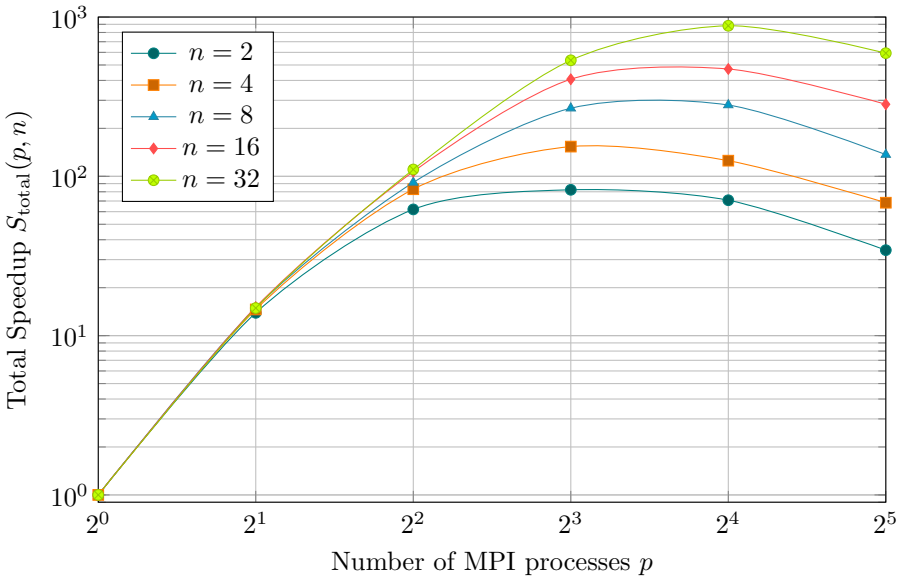

Figure 3: Measured **Total Speedup** $S_{\text{total}}(p, n)$ vs number of MPI processes for each matrix width n.

Discussion and interpretation

Total speedup shape. The total speedup curves initially grow with the number of MPI processes. For small matrix widths ($n = 2, 4$), speedup peaks at moderate process counts (8–16) and then decreases for more processes, showing a non-ideal scaling. For larger widths ($n = 16, 32$), speedup continues to grow with more processes up to 16, but eventually saturates or slightly declines.

Non-ideal behaviors.

- **Communication overhead:** For tall-skinny matrices and many processes, the cost of MPI communication can dominate, reducing scaling efficiency.
- **Load imbalance:** When the number of rows per process becomes very small, some processes are idle while others compute.

4.3 CUDA

The CUDA implementation of the TSQR algorithm was developed to accelerate the QR decomposition of tall-and-skinny matrices by exploiting fine-grained parallelism on the GPU. The input matrix $A \in \mathbb{R}^{h \times w}$ was partitioned into column blocks of height b_h and processed independently in the first stage of TSQR. Each block was assigned to a single CUDA block, with one thread performing the local QR factorization using a device-side Householder QR routine.

The kernel launch configuration for the first stage used: [`qr_on_blocks<<<num_blocks, 1, shared_mem>>>`] where `num_blocks = h / b_h`, and `shared_mem` was dynamically allocated to store intermediate matrices in shared memory. For a block height of 4 and a matrix of size 256×2, this resulted in 64 CUDA blocks being launched. The amount of shared memory was carefully tuned to stay within the hardware limits returned by `cudaGetDeviceProperties`, and explicitly checked against `prop.sharedMemPerBlock`.

During development, several common CUDA issues were encountered and resolved. These included illegal memory access errors caused by incorrect indexing, mismatches between host and device memory allocation sizes, and misconfigured `cudaMemcpy` directions. These problems were diagnosed using `cudaGetLastError()` and `cudaDeviceSynchronize()` after kernel launches to precisely identify failure points.

To quantify performance, both the CUDA TSQR implementation and a baseline serial CPU QR were benchmarked with all timings recorded in microseconds. Table 1 reports CUDA compute time, CUDA total time, and the CPU QR cost when available. CUDA total includes kernel launch overhead and device-host transfers.

Table 1: CUDA TSQR Performance and CPU QR Reference Times (µs)

Height	Width	CUDA Compute (µs)	CUDA Total (µs)	CPU Time (µs)
4	2	64	49414	2
4	4	93	49861	2
8	2	114	58061	3
8	4	264	50275	4
16	2	139	50483	8
16	4	412	50674	14
32	2	163	49748	41
32	4	561	51984	81
64	2	187	50713	337
64	4	722	50817	730
128	2	217	49714	3371
128	4	894	53088	7011
256	2	238	51693	40505
256	4	1090	58437	80276
512	2	260	50034	398335
512	4	1284	51120	794928
1024	2	288	49745	4094456
1024	4	1478	52835	6597709
2048	2	311	50468	99886803
2048	4	1673	51748	191214140
4096	2	336	52165	961167928
4096	4	2373	52150	2020257382
8192	2	858	50402	NA
8192	4	2094	53669	NA
16384	2	421	58708	NA
16384	4	2735	57435	NA

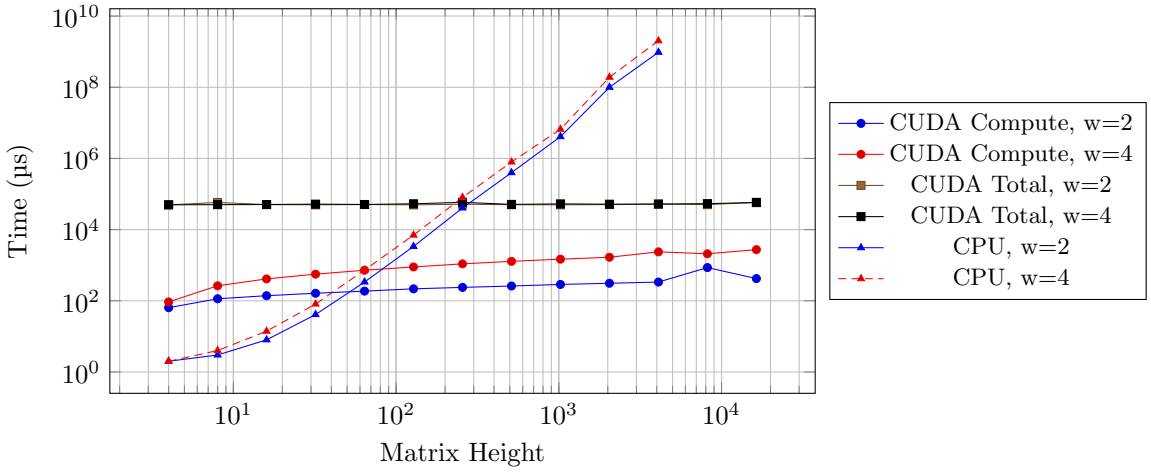

Figure 4: Performance comparison for CUDA TSQR and CPU QR with width 2 and 4. All values are in microseconds.

For sufficiently tall matrices, the GPU outperforms the serial CPU QR by a large margin. The CPU implementation is bottlenecked by its serial access pattern and cache inefficiency: it walks down rows with a single thread, producing poor locality and causing cache thrashing as height grows. That is why the CPU time explodes with height while CUDA compute remains relatively small.

CUDA compute time looks almost constant across heights because the current TSQR design assigns small local tiles to blocks and the per-block arithmetic cost is limited. CUDA total, however, is dominated by fixed launch and transfer overheads at small sizes. As the height grows large enough, the CPU cost becomes enormous and the GPU's parallel arithmetic dominates overall runtime, yielding significant end-to-end speedup. This is visible in Table 1 and in the accompanying figure.

Correctness was validated by copying R back to host and comparing against a CPU reference QR; the R factors agreed within floating point tolerance.

Practical next steps focus on further scaling and reducing overheads. Kernel launch and device-host transfer costs dominate for smaller matrices, so merging stages into a single kernel can reduce the number of launches and minimize transfer overhead. For very tall matrices, careful tuning of shared memory and block size ensures high SM occupancy and throughput. Implementing these optimizations will further amplify the GPU advantage at large sizes.

5 Contribution

Pudit Tempattarachoke and Chutchatut Sutichavengkul contributed to this project equally (50%-50%).

References

[1] Justin Solomon. *Numerical Algorithms: Methods for Computer Vision, Machine Learning, and Graphics*. USA: A. K. Peters, Ltd., 2015. ISBN: 1482251884.

[2] James Demmel et al. "Communication-optimal parallel and sequential QR and LU factorizations". In: *SIAM Journal on Scientific Computing* 34.1 (2012), A206–A239.