RWTH Aachen University
Department of Computer Science

Design and Performance Engineering of GPU-Accelerated Tensor Network Algorithms for Large-Scale Scientific Simulations

Master Thesis

submitted by

Daniel Sinkin
Matriculation Number: 367316

First Examiner:	Prof. Dr. TODO
Second Examiner:	Prof. Dr. TODO
Advisors:	Dr. Edoardo Di Napoli
External Institution:	Jülich Supercomputing Centre (JSC) Forschungszentrum Jülich

Aachen, February 17, 2026

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Acknowledgements

TODO

Contents

Abstract iii

Acknowledgements v

List of Figures ix

List of Tables xi

Listings xiii

List of Symbols xv

List of Abbreviations xvii

1. Introduction 1
- 1.1. Motivation 1
- 1.2. Problem Statement 1
- 1.3. Contributions 1
- 1.4. Outline 1

2. Background 3
- 2.1. Tensor Networks 3
  - 2.1.1. Tensor Notation and Diagrams 3
  - 2.1.2. Tensor Contraction 3
  - 2.1.3. Tensor Network Structures 3
- 2.2. GPU Architecture 3
  - 2.2.1. Streaming Multiprocessor and Warp Execution 3
  - 2.2.2. Floating-Point Formats and Precision Trade-offs 3
  - 2.2.3. Memory Hierarchy 6
  - 2.2.4. NVIDIA A100 Ampere Architecture 7
- 2.3. CUDA Programming Model 8
  - 2.3.1. Thread Hierarchy and Kernel Launch 8
  - 2.3.2. Shared Memory and Synchronisation 11
  - 2.3.3. Memory Coalescing and Bank Conflicts 13
  - 2.3.4. Performance Profiling with Nsight Compute 14
- 2.4. Related Work 15
  - 2.4.1. cuBLAS and cuTENSOR 15
  - 2.4.2. ChASE Eigensolver 15
  - 2.4.3. Existing GPU Tensor Network Implementations 15

3. Design and Methodology 17
- 3.1. Target Kernels 17
- 3.2. Algorithmic Approach 17

3.3. Data Layout and Memory Strategy 17
3.4. Baseline Selection 17

4. Implementation 19
4.1. Kernel Design 19
4.2. Shared Memory Tiling 19
4.3. Occupancy and Launch Configuration 19
4.4. Integration and Build System 19

5. Results 21
5.1. Experimental Setup 21
5.2. Single-GPU Performance 21
5.3. Profiling Analysis 21
5.4. Comparison with cuBLAS and cuTENSOR 21
5.5. Scaling Behaviour 21
5.6. Discussion 21

6. Conclusion 23
6.1. Summary 23
6.2. Limitations 23
6.3. Future Work 23

Bibliography 25

A. Supplementary Benchmarks 27

Declaration of Authorship 29

List of Figures

2.1. Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM). 6

2.2. Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM). 6

2.3. CUDA execution hierarchy showing the mapping of grids to thread blocks and threads. Threads are organised in up to three dimensions and identified using the built-in variables `threadIdx` and `blockIdx`. . 9

List of Tables

2.1. Bit layout of floating-point formats relevant to GPU computing. FP64, FP32, and FP16 are defined by IEEE 754 [IEE19]; BF16 and TF32 are industry-defined formats (see text). The significand column lists only the explicitly stored fraction bits; all formats carry an additional implicit leading bit for normal numbers. 4
2.2. Numerical properties of floating-point formats on the A100. Machine epsilon (ε) is the smallest increment to 1.0 that produces a distinct value, i.e. $\varepsilon = 2^{-p}$ where p is the number of significand bits (including the implicit bit). 4
2.3. Peak floating-point throughput (TFLOPS) of the A100 by format and execution unit. Tensor core figures in parentheses include sparsity acceleration (2:4 structured sparsity). Data from [RH20]. 5
2.4. Key hardware specifications of the NVIDIA A100 (SXM4-80GB). . . . 7
2.5. Theoretical peak floating-point throughput of the A100 GPU. 7
2.6. Derived theoretical limits of the A100 architecture. 8
2.7. Approximate memory access latency at different hierarchy levels. . . . 8

Listings

2.1. Element-wise vector addition kernel and its host-side launch. 9
2.2. Matrix transpose using two-dimensional grid and block addressing. . . 10
2.3. Static shared memory allocation for a tile. 11
2.4. Dynamic shared memory allocation. 11
2.5. Tiled matrix multiplication using shared memory. Each block computes one tile of the output matrix C. 12
2.6. Coalesced access pattern: consecutive threads read consecutive columns. 13
2.7. Non-coalesced access pattern: consecutive threads read elements separated by stride N. 13
2.8. Bank conflict when accessing a column of a 32-wide shared array, and the padding fix. 13

List of Symbols

$\mathcal{T}$	Tensor
$\mathbf{A}, \mathbf{B}$	Matrices
$\vec{v}$	Vector
χ	Bond dimension
d	Local (physical) dimension
N	Matrix/problem size
$\mathcal{O}(\cdot)$	Asymptotic upper bound

List of Abbreviations

BLAS	Basic Linear Algebra Subprograms
DFT	Density Functional Theory
GEMM	General Matrix Multiply
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HPC	High Performance Computing
MPI	Message Passing Interface
MPS	Matrix Product State (tensor networks)
MPS	Multi-Process Service (Nvida CUDA)
SM	Streaming Multiprocessor
TN	Tensor Network

1. Introduction

1.1. Motivation

1.2. Problem Statement

1.3. Contributions

1.4. Outline

Chapter 2 introduces ...Chapter 3 presents ...Chapter 4 details ...Chapter 5 evaluates ...Chapter 6 summarises ...

2. Background

2.1. Tensor Networks

2.1.1. Tensor Notation and Diagrams

2.1.2. Tensor Contraction

2.1.3. Tensor Network Structures

2.2. GPU Architecture

2.2.1. Streaming Multiprocessor and Warp Execution

2.2.2. Floating-Point Formats and Precision Trade-offs

Scientific computing on GPUs has historically defaulted to 64-bit double precision (FP64), matching the convention established by decades of CPU-based numerical software. However, the A100 architecture supports a range of floating-point formats with dramatically different throughput characteristics, and for many workloads—including tensor contractions—FP32 or even reduced-precision formats offer sufficient accuracy at a fraction of the cost. This section reviews the relevant formats and motivates the precision choices made in this thesis.

IEEE 754 Binary Representation

The IEEE 754 standard [IEE19] defines the binary floating-point formats used across virtually all modern hardware. Each format encodes a real number as

$$(-1)^s \times 2^{e-\text{bias}} \times (1+f), \tag{2.1}$$

where s is a sign bit, e is an unsigned integer stored in the exponent field, the *bias* centres the exponent range around zero, and f is the fractional part of the significand (with an implicit leading 1 for normal numbers). The three fields are packed into a fixed-width bit string as shown in Table 2.1.

The number of exponent bits determines the *dynamic range* (the ratio of the largest to smallest representable magnitudes), while the number of significand bits determines the *precision* (the spacing between adjacent representable values). These properties are summarised in Table 2.2.

Two aspects of this table merit attention. First, TF32 and BF16 share the same 8-bit exponent as FP32, so they retain the full FP32 dynamic range despite having far fewer significand bits. TF32 was introduced specifically for the Ampere tensor cores: inputs are stored as ordinary FP32 values, but the tensor core hardware internally truncates to 10 significand bits before performing the multiply-accumulate, with the accumulation

Table 2.1.: Bit layout of floating-point formats relevant to GPU computing. FP64, FP32, and FP16 are defined by IEEE 754 [IEE19]; BF16 and TF32 are industry-defined formats (see text). The significand column lists only the explicitly stored fraction bits; all formats carry an additional implicit leading bit for normal numbers.

Format	Total bits	Sign	Exponent	Significand (fraction)
FP64 (double)	64	1	11	52
FP32 (single)	32	1	8	23
TF32	19	1	8	10
BF16 (bfloat16)	16	1	8	7
FP16 (half)	16	1	5	10

Table 2.2.: Numerical properties of floating-point formats on the A100. Machine epsilon (ε) is the smallest increment to 1.0 that produces a distinct value, i.e. $\varepsilon = 2^{-p}$ where p is the number of significand bits (including the implicit bit).

Format	Largest value	Smallest normal > 0	Machine epsilon
FP64	$\approx 1.8 \times 10^{308}$	$\approx 2.2 \times 10^{-308}$	$\approx 2.2 \times 10^{-16}$
FP32	$\approx 3.4 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 1.2 \times 10^{-7}$
TF32	$\approx 3.4 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 9.8 \times 10^{-4}$
BF16	$\approx 3.3 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 7.8 \times 10^{-3}$
FP16	65504	$\approx 6.1 \times 10^{-5}$	$\approx 9.8 \times 10^{-4}$

itself carried out in full FP32 [NVI20]. The programmer does not need to convert data explicitly; the truncation is transparent.

Second, FP16 has a severely limited dynamic range (maximum value 65504), which makes it unsuitable for many scientific workloads without careful scaling. BF16 ("brain floating point"), originally developed for deep learning training on Google's TPUs [KMM+19], avoids this limitation by retaining the FP32 exponent range at the cost of reduced precision (7 fraction bits versus 10 for FP16).

A100 Throughput by Precision

The performance gap between precisions is substantial. Table 2.3 reproduces the A100 throughput figures from NVIDIA's Ampere optimisation guide [RH20], broken down by execution unit.

The ratios are stark. Moving from FP64 CUDA core arithmetic (9.7 TFLOPS) to FP32 (19.5 TFLOPS) doubles throughput at no algorithmic cost. Engaging the tensor cores in TF32 mode yields a further 8× increase to 156 TFLOPS, and FP16/BF16 tensor core throughput reaches 312 TFLOPS—a 32× factor over FP64. Even accounting for the fact that many kernels are memory-bandwidth-bound rather than compute-bound, the reduced data movement from using 32-bit instead of 64-bit values halves the memory traffic, which directly benefits bandwidth-limited operations.

Table 2.3.: Peak floating-point throughput (TFLOPS) of the A100 by format and execution unit. Tensor core figures in parentheses include sparsity acceleration (2:4 structured sparsity). Data from [RH20].

Format	Scalar (CUDA cores)	Vector (CUDA cores)	Tensor cores
FP64	9.7	9.7	19.5
FP32	19.5	19.5	156 (TF32)
FP16	19.5	78	312 (624)
BF16	19.5	39	312 (624)

The Case for Single Precision in Tensor Network Computations

The choice of FP64 in scientific software is often driven by convention rather than necessity [HM22]. Double precision provides roughly 15–16 decimal digits of accuracy, while single precision provides 7–8. Whether the additional digits matter depends on the conditioning of the computation and the accuracy actually required by the application. Higham and Mary [HM22] survey a broad class of numerical linear algebra algorithms where mixed- or reduced-precision arithmetic achieves results of comparable quality to FP64 at substantially lower cost—an observation that applies directly to the tensor contractions considered here.

For tensor network algorithms, several factors favour FP32:

1. **Physical observables are approximate.** Tensor network methods are inherently approximate—the bond dimension truncation introduces controlled errors that typically far exceed single-precision rounding. When the truncation error is $\mathcal{O}(10^{-4})$ to $\mathcal{O}(10^{-6})$, carrying 10^{-16} precision in every floating-point operation is wasteful.

2. **Iterative refinement.** Many tensor network algorithms are iterative (e.g. DMRG sweeps, variational optimisation). Rounding errors from FP32 arithmetic are corrected at each iteration, so they do not accumulate in the same way as in a single long computation.

3. **Memory pressure.** Tensors with large bond dimensions consume substantial memory. Halving the per-element storage from 8 bytes to 4 bytes doubles the tensor sizes that fit in GPU memory, or equivalently allows larger bond dimensions for the same memory budget.

4. **Bandwidth amplification.** Since tensor contractions are frequently memory-bandwidth-bound (Section 2.2.3), the 2× reduction in data movement from FP32 translates almost directly into a 2× speedup for bandwidth-limited kernels, before accounting for any compute throughput gains.

The combination of these factors yields a practical speedup well in excess of the raw 2× compute ratio, as demonstrated in the benchmarks in **??**. Where necessary, mixed-precision strategies—accumulating in FP32 while storing in FP16 or BF16—can push performance further (see [HM22] for a rigorous treatment), though this thesis focuses on FP32 as the primary working precision.

2.2.3. Memory Hierarchy

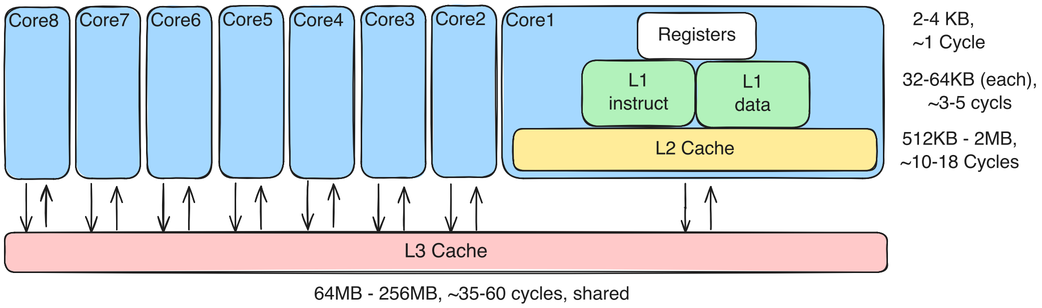
Core8
Core7
Core6
Core5
Core4
Core3
Core2
Core1
Registers
L1 instruct
L1 data
L2 Cache
2-4 KB, ~1 Cycle
32-64KB (each), ~3-5 cycls
512KB - 2MB, ~10-18 Cycles
L3 Cache
64MB - 256MB, ~35-60 cycles, shared

Figure 2.1.: Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM).

As shown in Figure 2.1, modern CPUs rely on a deep cache hierarchy to reduce effective memory latency.

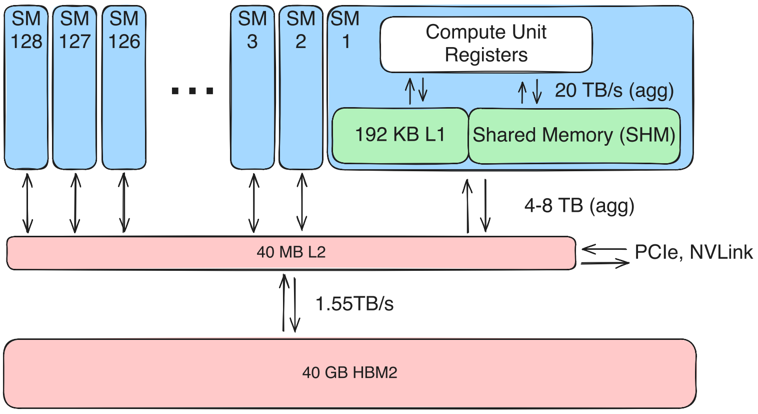
SM 128
SM 127
SM 126
...
SM 3
SM 2
SM 1
Compute Unit Registers
20 TB/s (agg)
192 KB L1
Shared Memory (SHM)
4-8 TB (agg)
40 MB L2
PCIe, NVLink
1.55TB/s
40 GB HBM2

Figure 2.2.: Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM).

GPU Memory Hierarchy Where CPUs use deep cache hierarchies to hide latency, GPUs prioritise bandwidth and expose a shallower hierarchy tuned for throughput. Four levels are relevant:

- **Registers** reside in the SM and offer single-cycle access. They are private to each thread and are the scarcest on-chip resource: the number of registers a kernel uses directly limits occupancy.
- **Shared memory** is also on-chip but visible to all threads in a block, making it the primary mechanism for explicit data reuse and inter-thread communication. Most high-performance kernels—tiled matrix multiplications, tensor contractions—stage data through shared memory to avoid repeated global loads.
- **L2 cache** is shared across all SMs and transparently caches global memory traffic. On the A100 it is 40 MB, modest relative to the core count, which reflects the throughput-oriented design.

Table 2.4.: Key hardware specifications of the NVIDIA A100 (SXM4-80GB).

Property	Value
Streaming Multiprocessors (SMs)	108
CUDA cores per SM	64
Tensor cores per SM	4
Warp schedulers per SM	4
Maximum threads per SM	2048
Maximum warps per SM	64
Maximum blocks per SM	32
Register file size per SM (32-bit registers)	256 000
Shared memory per SM (KB)	164
L2 cache size (MB)	40
HBM memory capacity (GB)	80
Peak memory bandwidth (GB/s)	2039
Base clock frequency (GHz)	1.41

Table 2.5.: Theoretical peak floating-point throughput of the A100 GPU.

Precision	Peak TFLOPS	Relative speed
FP64 (CUDA cores)	9.7	1.0×
FP64 (Tensor cores)	19.5	2.0×
FP32	19.5	2.0×
TF32 (Tensor cores)	156.0	16.1×
FP16 (Tensor cores)	312.0	32.2×

- **HBM (global memory)** provides the bulk storage. The A100 uses HBM2e with 80 GB capacity and a peak bandwidth of 2039 GB/s (Table 2.4), achieved through a wide interface with thousands of parallel data lines. Access latency, however, is roughly 500× that of a register access (Table 2.7).

The practical consequence is that most scientific GPU kernels, including tensor contractions, are memory-bandwidth-bound rather than compute-bound. Performance therefore hinges on maximising reuse in registers and shared memory and accessing HBM in coalesced, bandwidth-efficient patterns.

2.2.4. NVIDIA A100 Ampere Architecture

Hardware Overview

The A100 employs HBM2e as its main device memory, providing the high memory bandwidth necessary to sustain its computational throughput.

Table 2.6.: Derived theoretical limits of the A100 architecture.

Metric	Value
Peak FP32 performance per SM (TFLOPS)	0.18
Peak FP64 performance per SM (TFLOPS)	0.09
Tensor core FP16 performance per SM (TFLOPS)	2.89
Memory bandwidth per SM (GB/s)	18.88
Total CUDA cores	6912
Total FP64 cores	3456
Maximum resident warps	6912
Maximum resident threads	221 184
Arithmetic intensity threshold FP32 (FLOPs/byte)	9.6
Arithmetic intensity threshold FP64 (FLOPs/byte)	4.8

Table 2.7.: Approximate memory access latency at different hierarchy levels.

Memory level	Latency (cycles)
Registers	1
Shared memory	20
L2 cache	200
HBM global memory	500

Theoretical Peak Performance

Derived Performance Limits

Memory Latency

2.3. CUDA Programming Model

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel programming platform for general-purpose computation on GPUs. A CUDA program consists of host code, which runs on the CPU, and *kernels*, which are functions launched from the host but executed in parallel across many GPU threads. The programmer specifies the parallelism by choosing a grid of thread blocks at launch time; the hardware then schedules those blocks onto the available SMs.

2.3.1. Thread Hierarchy and Kernel Launch

CUDA organises parallel execution into a three-level hierarchy: *grids*, *thread blocks* (or simply *blocks*), and *threads*. A kernel launch creates a single grid, which is partitioned into blocks. Each block contains a fixed number of threads that execute concurrently on the same SM and can cooperate through shared memory and synchronisation barriers. Threads in different blocks cannot synchronise with each other during kernel execution.

Both the grid and each block can be specified with up to three dimensions, which provides a natural mapping for problems defined over multidimensional arrays. The dimensions are set via the `dim3` type, and each thread identifies its position using the built-in variables `threadIdx`, `blockIdx`, `blockDim`, and `gridDim`.

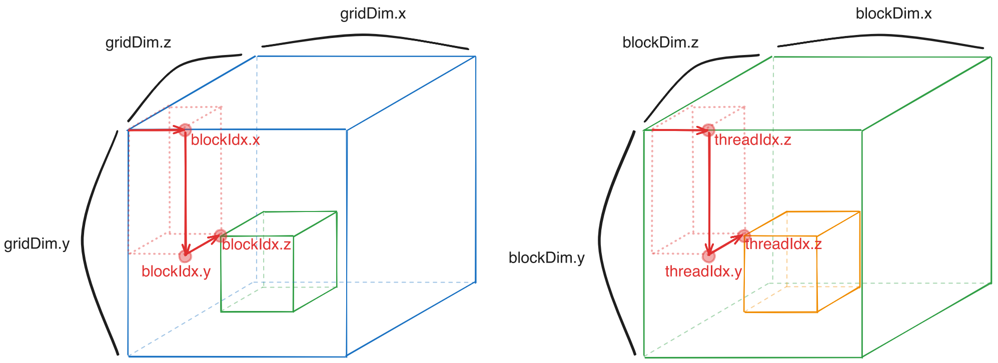

Figure 2.3.: CUDA execution hierarchy showing the mapping of grids to thread blocks and threads. Threads are organised in up to three dimensions and identified using the built-in variables `threadIdx` and `blockIdx`.

The hierarchical organisation of grids, thread blocks, and threads is illustrated in Figure 2.3. Within each block, threads are further grouped into *warps* of 32 threads that execute instructions in lock-step on the SM's SIMT (Single Instruction, Multiple Thread) execution units.

Kernel Syntax and Launch Configuration

A kernel is declared with the `__global__` qualifier and launched using the triple-chevron syntax `<<<gridDim, blockDim>>>`. The following example illustrates a minimal kernel that adds two vectors element-wise:

```
1  __global__ void vecAdd(const float *a, const float *b, float *c, int n) {
2      int i = blockIdx.x * blockDim.x + threadIdx.x;
3      if (i < n) {
4          c[i] = a[i] + b[i];
5      }
6  }
7
8  // Host launch
9  int n = 1 << 20;                      // 1,048,576 elements
10 int threadsPerBlock = 256;
11 int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
12 vecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);
```

Listing 2.1: Element-wise vector addition kernel and its host-side launch.

The global thread index is computed on line 2 as `blockIdx.x * blockDim.x + threadIdx.x`. This is the standard one-dimensional addressing pattern: `blockIdx.x` selects which block the thread belongs to, `blockDim.x` gives the number of threads per block, and `threadIdx.x` is the thread's position within its block. The bounds check on line 3 is necessary because the total number of threads launched (blocks × threads per block) is rounded up to a multiple of the block size and may therefore exceed `n`.

Two-Dimensional Grid Addressing

For problems with a natural two-dimensional structure, such as matrix operations or image processing, both the grid and block dimensions can be specified in two (or three) dimensions. The following example demonstrates a kernel that transposes an $M \times N$ matrix:

```
__global__ void transpose(const float *in, float *out, int M, int N) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < M && col < N) {
        out[col * M + row] = in[row * N + col];
    }
}

// Host launch
dim3 blockDim(16, 16);          // 256 threads per block
dim3 gridDim(
    (N + blockDim.x - 1) / blockDim.x,
    (M + blockDim.y - 1) / blockDim.y
);
transpose<<<gridDim, blockDim>>>(d_in, d_out, M, N);
```

Listing 2.2: Matrix transpose using two-dimensional grid and block addressing.

Here each thread is addressed by a `(row, col)` pair derived from the two-dimensional block and thread indices. The grid is sized so that at least $M \times N$ threads are launched, again with bounds checking to handle dimensions that are not exact multiples of the block size.

Block Size Selection and Hardware Constraints

The choice of block size affects both correctness and performance. The CUDA programming model imposes an upper limit of 1024 threads per block. Beyond this hard limit, several performance-relevant factors guide the choice:

- **Warp granularity.** Because threads are scheduled in warps of 32, the block size should be a multiple of 32 to avoid partially filled warps whose unused lanes still consume scheduling resources.
- **Occupancy.** Each SM has a finite number of registers, a fixed amount of shared memory, and a maximum number of resident threads (2048 on the A100). If a kernel uses many registers per thread, fewer threads can reside on the SM simultaneously, which may leave the hardware underutilised. The CUDA occupancy calculator relates block size and per-thread resource usage to the fraction of the SM's capacity that is occupied.
- **Shared memory per block.** Shared memory is partitioned among the blocks resident on an SM. A block that allocates a large amount of shared memory limits the number of blocks that can coexist, potentially reducing occupancy.

In practice, block sizes of 128 or 256 threads are common defaults that balance these constraints. More performance-critical kernels tune the block size empirically or use the `__launch_bounds__` qualifier to give the compiler additional information for register allocation.

Linearisation of Multidimensional Indices

GPU memory is addressed linearly, so multidimensional arrays must be mapped to one-dimensional offsets. For a tensor stored in row-major order, the linear index of element (i, j) in an $M \times N$ matrix is

$$\text{offset} = i \cdot N + j, \tag{2.2}$$

which generalises to higher-order tensors by successive multiplication with trailing dimensions. In column-major (Fortran) order, the convention used by cuBLAS and most BLAS libraries, the index is instead

$$\text{offset} = j \cdot M + i. \tag{2.3}$$

Listing 2.2 uses row-major layout for the input (`in[row * N + col]`) and column-major layout for the output (`out[col * M + row]`), which is precisely the transpose operation.

A concrete example: for a 4×3 matrix stored in row-major order, the element at row 2, column 1 maps to linear offset $2 \cdot 3 + 1 = 7$. Understanding this mapping is essential for implementing coalesced memory access patterns, discussed in Section 2.3.3.

2.3.2. Shared Memory and Synchronisation

Shared memory is an on-chip, programmer-managed memory space visible to all threads within a block. It serves two purposes: as a software-managed cache to stage data from global memory, and as a communication channel between threads in the same block. Because shared memory has much lower latency than global memory (approximately 20 cycles versus 500 cycles on the A100, cf. Table 2.7), its effective use is often the difference between a bandwidth-bound kernel and one that approaches peak compute throughput.

Static and Dynamic Allocation

Shared memory can be allocated statically at compile time or dynamically at launch time. Static allocation uses the `__shared__` qualifier with a fixed array size:

```
__shared__ float tile[TILE_SIZE][TILE_SIZE];
```

Listing 2.3: Static shared memory allocation for a tile.

Dynamic allocation is specified as a third argument in the launch configuration and accessed through an `extern __shared__` declaration:

```
extern __shared__ float smem[];

// Host launch with dynamic shared memory size in bytes
kernel<<<grid, block, sharedBytes>>>(args);
```

Listing 2.4: Dynamic shared memory allocation.

Dynamic allocation is useful when the tile size depends on runtime parameters, which is common in autotuned kernels.

Tiled Matrix Multiplication

The canonical use of shared memory is tiled (or blocked) matrix multiplication. A naïve matrix multiplication kernel that computes $C = AB$ with $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$ has each thread compute one element of C by reading an entire row of A

and column of B from global memory. This results in $\mathcal{O}(K)$ global loads per thread and an arithmetic intensity of roughly 2 FLOPs/byte—well below the A100's crossover point of ≈ 9.6 FLOPs/byte for FP32 (Table 2.6).

Tiling reduces global memory traffic by loading the input matrices into shared memory in small blocks (tiles), computing partial dot products from the tile, and then advancing to the next tile. Listing 2.5 shows the structure of this approach.

```
 1 #define TILE 16
 2
 3 __global__ void matmul(const float *A, const float *B, float *C,
 4                        int M, int N, int K) {
 5     __shared__ float As[TILE][TILE];
 6     __shared__ float Bs[TILE][TILE];
 7
 8     int row = blockIdx.y * TILE + threadIdx.y;
 9     int col = blockIdx.x * TILE + threadIdx.x;
10     float sum = 0.0f;
11
12     for (int t = 0; t < (K + TILE - 1) / TILE; t++) {
13         // Cooperative load: each thread loads one element of each tile
14         int aCol = t * TILE + threadIdx.x;
15         int bRow = t * TILE + threadIdx.y;
16         As[threadIdx.y][threadIdx.x] = (row < M && aCol < K)
17                                        ? A[row * K + aCol] : 0.0f;
18         Bs[threadIdx.y][threadIdx.x] = (bRow < K && col < N)
19                                        ? B[bRow * N + col] : 0.0f;
20
21         __syncthreads();  // ensure the tile is fully loaded
22
23         for (int k = 0; k < TILE; k++) {
24             sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];
25         }
26
27         __syncthreads();  // safe to overwrite tile in next iteration
28     }
29
30     if (row < M && col < N) {
31         C[row * N + col] = sum;
32     }
33 }
```

Listing 2.5: Tiled matrix multiplication using shared memory. Each block computes one tile of the output matrix C.

The key points of this kernel are:

1. **Cooperative loading (lines 14–19).** Every thread in the block loads exactly one element of the A-tile and one element of the B-tile. The 256 threads in a 16×16 block therefore load the entire 16×16 tile cooperatively. Boundary conditions are handled by loading zero for out-of-range indices.

2. **Barrier synchronisation (line 21).** The `__syncthreads()` call ensures that all threads have finished writing to shared memory before any thread begins reading from the tile. A second barrier after the computation (line 27) prevents any thread from overwriting the tile before all threads have finished using it.

3. **Data reuse.** Each element loaded into `As` is read by all 16 threads in its row; each element of `Bs` is read by all 16 threads in its column. This 16× reuse from shared memory instead of global memory raises the effective arithmetic intensity by the tile size factor.

For a tile size of T, the number of global memory loads per output element drops from $2K$ (naïve) to $2K/T$, and the arithmetic intensity increases proportionally by T. Larger tiles improve reuse but require more shared memory per block, which can limit occupancy.

2.3.3. Memory Coalescing and Bank Conflicts

Global Memory Coalescing

When a warp executes a load or store instruction, the hardware combines the 32 individual thread addresses into as few memory transactions as possible. If consecutive threads access consecutive memory addresses (i.e. thread i accesses address base $+ i$), the accesses are *coalesced* into a minimal number of 128-byte cache line transactions. Non-coalesced access patterns—strided or scattered—require multiple transactions for the same warp, wasting bandwidth.

Consider accessing a row-major $M \times N$ matrix. Iterating over columns (adjacent elements in memory) with consecutive threads produces coalesced accesses:

```
// Coalesced: threads in a warp read adjacent elements
float val = matrix[row * N + threadIdx.x];
```

Listing 2.6: Coalesced access pattern: consecutive threads read consecutive columns.

Iterating over rows with consecutive threads instead produces strided accesses with stride N, which is poorly coalesced:

```
// Non-coalesced: stride-N access
float val = matrix[threadIdx.x * N + col];
```

Listing 2.7: Non-coalesced access pattern: consecutive threads read elements separated by stride N.

This distinction is particularly relevant for tensor contractions, where the choice of loop ordering and index layout determines whether the innermost memory accesses are contiguous or strided.

Shared Memory Bank Conflicts

Shared memory is divided into 32 *banks*, each 4 bytes wide, interleaved in a round-robin fashion: word k resides in bank $k \bmod 32$. When multiple threads in a warp access different words in the same bank simultaneously, the accesses are serialised into multiple rounds, creating a *bank conflict*. The worst case is a 32-way bank conflict, where all threads hit the same bank, serialising the access entirely.

A common source of bank conflicts arises when accessing columns of a shared memory array whose leading dimension is a multiple of 32:

```
// 32-way bank conflict: column access, stride = 32
__shared__ float tile[32][32];
float val = tile[threadIdx.x][col];  // threads 0..31 all hit same bank
```

```

// Fix: pad the leading dimension by one
__shared__ float tile[32][33];       // stride = 33, conflicts eliminated
float val = tile[threadIdx.x][col];
```

Listing 2.8: Bank conflict when accessing a column of a 32-wide shared array, and the padding fix.

Adding one element of padding to the inner dimension changes the stride to 33, which is coprime to 32, so consecutive rows map to distinct banks. This is a standard optimisation in tiled kernels.

2.3.4. Performance Profiling with Nsight Compute

NVIDIA Nsight Compute is a kernel-level profiling tool for CUDA applications. It collects hardware performance counters and presents them as high-level metrics and roofline-model analyses, making it the primary tool for identifying performance bottlenecks in GPU kernels.

A typical profiling workflow proceeds as follows. First, the application is run under `ncu` (the Nsight Compute command-line interface), which replays each kernel multiple times to collect a full set of counters. Then the resulting report is examined either in the Nsight Compute GUI or by querying specific metrics from the command line.

Key metrics reported by Nsight Compute that are relevant to the optimisations discussed in subsequent chapters include:

- **Achieved occupancy:** the ratio of active warps per cycle to the maximum the SM can support, indicating how effectively the kernel hides memory latency through parallelism.

- **Memory throughput:** the achieved bandwidth to each level of the memory hierarchy (HBM, L2, shared memory), compared against the theoretical peak. A kernel achieving close to peak HBM bandwidth is memory-bound.

- **Compute throughput:** the achieved FLOP/s relative to the peak, broken down by instruction type. A kernel well below peak compute with high memory throughput is bandwidth-limited.

- **Warp stall reasons:** a breakdown of why warps were stalled (e.g. waiting for memory, waiting at a barrier, instruction dependencies), which directly guides optimisation.

- **Shared memory bank conflicts:** the number of replayed shared memory accesses due to bank conflicts, indicating whether padding or access pattern changes are needed.

- **L2 cache hit rate:** the fraction of global memory requests served by the L2 cache, relevant for understanding data reuse across thread blocks.

Throughout the implementation chapters of this thesis, Nsight Compute profiles are used to validate performance models, confirm that kernels are operating in the expected regime (compute-bound or memory-bound), and guide iterative optimisation of tensor contraction kernels.

2.4. Related Work

2.4.1. cuBLAS and cuTENSOR

2.4.2. ChASE Eigensolver

2.4.3. Existing GPU Tensor Network Implementations

3. Design and Methodology

3.1. Target Kernels

3.2. Algorithmic Approach

3.3. Data Layout and Memory Strategy

3.4. Baseline Selection

4. Implementation

4.1. Kernel Design

4.2. Shared Memory Tiling

4.3. Occupancy and Launch Configuration

4.4. Integration and Build System

5. Results

5.1. Experimental Setup

5.2. Single-GPU Performance

5.3. Profiling Analysis

5.4. Comparison with cuBLAS and cuTENSOR

5.5. Scaling Behaviour

5.6. Discussion

6. Conclusion

6.1. Summary

6.2. Limitations

6.3. Future Work

Bibliography

[HM22] Nicholas J. Higham and Theo Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[IEE19] IEEE. IEEE standard for floating-point arithmetic, 2019.

[KMM$^+$19] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of BFLOAT16 for deep learning training. In *arXiv preprint arXiv:1905.12322*, 2019.

[NVI20] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*, 2020. Whitepaper v1.0.

[NVI22] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Architecture*, 2022. Whitepaper.

[NVI24a] NVIDIA Corporation. cuBLAS library, 2024. `https://developer.nvidia.com/cublas`.

[NVI24b] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2024. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[NVI24c] NVIDIA Corporation. cuTENSOR: A high-performance tensor primitives library, 2024. `https://developer.nvidia.com/cutensor`.

[NVI25] NVIDIA Corporation. cuBLASDx: Device side BLAS extensions, 2025. `https://docs.nvidia.com/cuda/cublasdx/`.

[RH20] Ronny Ramirez and William Hsu. Optimizing applications for NVIDIA Ampere GPU architecture. GTC 2020, Session S21819, 2020. `https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf`.

[SSK17] Paul Springer, Tong Su, and Tamara G. Kolda. Tensor contractions with extended BLAS kernels on CPU and GPU. In *IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 2017. `https://research.nvidia.com/sites/default/files/pubs/2017-10_Tensor-Contractions-with/tensors_hipc.pdf`.

[WDADN22] Xinzhe Wu, Davor Davidović, Sebastian Achilles, and Edoardo Di Napoli. ChASE: a distributed hybrid CPU-GPU eigensolver for large-scale hermitian eigenvalue problems. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '22)*, pages 1–12. ACM, 2022.

[WDN23] Xinzhe Wu and Edoardo Di Napoli. Advancing the distributed multi-GPU ChASE library through algorithm optimization and NCCL library. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23)*, pages 1688–1696. ACM, 2023.

[WSDN19] Jan Winkelmann, Paul Springer, and Edoardo Di Napoli. ChASE: Chebyshev accelerated subspace iteration eigensolver for sequences of Hermitian eigenvalue problems. *ACM Transactions on Mathematical Software*, 45(2):1–34, 2019.

[Wu19] Xinzhe Wu. *Contribution to the Emergence of New Intelligent Parallel and Distributed Methods Using a Multi-level Programming Paradigm for Extreme Computing.* PhD thesis, University of Lille, 2019.

A. Supplementary Benchmarks

TODO

Declaration of Authorship

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any quotes accordingly.

Aachen, February 17, 2026 Daniel Sinkin