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 20, 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 ii

Acknowledgements iii

List of Figures viii

List of Tables x

Listings xii

List of Symbols xiii

List of Abbreviations xiv

Glossary xv

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Problem Statement . . . 1
- 1.3. Contributions . . . 1
- 1.4. Outline . . . 2

2. Background 3
- 2.1. Tensor Networks . . . 3
  - 2.1.1. Historical Background . . . 3
  - 2.1.2. Why Tensor Networks Are of Interest . . . 3
  - 2.1.3. Computational Context . . . 4
  - 2.1.4. Matrix Product States (MPS) from a Computational View . . . 4
  - 2.1.5. Kernel Patterns Relevant to This Thesis . . . 5
  - 2.1.6. Scope Boundary . . . 6
- 2.2. Mathematical Foundations for Performance Evaluation . . . 6
  - 2.2.1. Work, Data Movement, and Runtime Bounds . . . 6
  - 2.2.2. Parallel Scaling Metrics . . . 7
  - 2.2.3. Floating-Point Error Model . . . 7
  - 2.2.4. Timing Statistics . . . 8
- 2.3. GPU Architecture . . . 9
  - 2.3.1. Streaming Multiprocessor and Warp Execution . . . 9
  - 2.3.2. What Is New in A100 for This Thesis . . . 10
  - 2.3.3. Floating-Point Formats and Precision Trade-offs . . . 11
  - 2.3.4. Memory Hierarchy . . . 16
  - 2.3.5. NVIDIA A100 Ampere Architecture . . . 17
  - 2.3.6. Compute Node Topology . . . 21

2.4. CUDA Programming Model . . . 26
2.4.1. Thread Hierarchy and Kernel Launch . . . 27
2.4.2. Shared Memory and Synchronisation . . . 30
2.4.3. Memory Coalescing and Bank Conflicts . . . 31
2.4.4. Performance Profiling with Nsight Compute . . . 32
2.4.5. Kernel Launch Mechanics . . . 33
2.5. Related Work . . . 36
2.5.1. cuBLAS, cuBLASDx, and cuTENSOR . . . 36
2.5.2. ChASE Eigensolver . . . 36
2.5.3. Existing GPU Tensor Network Implementations . . . 36
2.5.4. Selected Work in Collaborating Research Directions . . . 37
2.5.5. Representative CUDA Implementation Patterns . . . 39

3. Design and Methodology 42
3.1. Target Kernels . . . 42
3.1.1. Selection Criteria . . . 42
3.1.2. SOTA-Informed Initial Backlog (Working Notes) . . . 42
3.2. Case Study: Contraction Order in a Two-Site TN Update . . . 43
3.2.1. Problem Definition . . . 43
3.2.2. How to Read This as a Practical HPC Task . . . 43
3.2.3. Plausible Baseline (Bad) . . . 44
3.2.4. Improved Algorithm (Better) . . . 44
3.2.5. Worked Numerical Example (Shape Used in This Thesis) . . . 45
3.2.6. Practice Problems (Implementation-Oriented) . . . 45
3.2.7. Diagram Blueprint for Manual Drawing . . . 46
3.2.8. Design Decisions Behind the Better Variant . . . 46
3.2.9. Optimisation Ladder (Structured After SGEMM Worklogs) . . . 47
3.2.10. Profiling Reproducibility . . . 47
3.3. Algorithmic Approach . . . 47
3.3.1. Optimisation Axes . . . 48
3.3.2. Acceptance Criteria . . . 48
3.3.3. Analytical Performance Models . . . 48
3.4. Data Layout and Memory Strategy . . . 54
3.4.1. Layout Principles . . . 54
3.4.2. Memory-Level Strategy . . . 55
3.5. Baseline Selection . . . 55
3.5.1. Primary Baselines . . . 55
3.5.2. Fairness Criteria . . . 55
3.5.3. Profiling Evidence Gate . . . 56
3.6. Methodology and Reporting Standards . . . 56
3.6.1. Profiling Workflow Standard . . . 56
3.6.2. Benchmark Design Standard . . . 57
3.6.3. Mandatory Reporting Items . . . 58
3.6.4. Claim-to-Evidence Rules . . . 58
3.6.5. Threats-to-Validity Template . . . 58

4. Implementation **59**
4.1. Integration and Build System . . . 59
4.1.1. Build Configuration . . . 59
4.1.2. Runtime Integration . . . 59
4.2. Kernel Design . . . 59
4.2.1. Design Variants . . . 60
4.2.2. Correctness and Numerical Checks . . . 60
4.3. Occupancy and Launch Configuration . . . 60
4.3.1. Tuning Procedure . . . 60
4.3.2. Launch Overhead Considerations . . . 60
4.4. Shared Memory Tiling . . . 61
4.4.1. Tile Design Choices . . . 61
4.4.2. Boundary Handling . . . 61

5. Results **62**
5.1. Experimental Setup . . . 62
5.1.1. Hardware/Software Baseline . . . 62
5.1.2. Workload Definition . . . 62
5.1.3. Measurement Protocol . . . 62
5.2. Kernel Launch Overhead Benchmark . . . 63
5.2.1. Setup . . . 63
5.2.2. Results . . . 63
5.2.3. Discussion . . . 64
5.3. Single-GPU Performance . . . 64
5.3.1. Metrics . . . 65
5.3.2. Evaluation Dimensions . . . 65
5.4. Scaling Behaviour . . . 65
5.4.1. Strong and Weak Scaling Views . . . 65
5.4.2. Communication Effects . . . 65
5.5. Profiling Analysis . . . 66
5.5.1. KPI Pack Used for Attribution . . . 66
5.5.2. Interpretation Rules . . . 67
5.5.3. Application to Benchmark Pairs . . . 68
5.5.4. Global-Memory Coalescing Sweep . . . 68
5.5.5. Register Pressure and Occupancy . . . 69
5.6. Comparison with cuBLAS and cuTENSOR . . . 69
5.6.1. Comparison Protocol . . . 69
5.6.2. Interpretation Strategy . . . 70
5.7. Energy Efficiency Considerations . . . 70
5.7.1. Why Energy is in Scope for This Thesis . . . 71
5.7.2. Metrics . . . 71
5.7.3. First-Order Model for Contraction Kernels . . . 71
5.7.4. Implications for This Thesis . . . 72
5.7.5. Measurement Protocol on A100 Nodes . . . 72
5.8. Discussion . . . 73
5.8.1. What Improved and Why . . . 73
5.8.2. How Well Results Transfer . . . 73
5.8.3. Microbenchmark Implications for Tensor Networks . . . 73

6. Conclusion **74**
6.1. Summary 74
6.2. Limitations 74
6.3. Future Work 74

Bibliography **75**

7. Experimental Environment and Reproducibility **81**
7.1. Hardware Configuration 81
7.2. Software Environment 82
7.3. GPU Topology 82
7.4. Build and Run Procedure 83
7.5. Measurement Methodology 83
7.6. Data Availability 83

A. Supplementary Benchmarks **84**

Declaration of Authorship **85**

List of Figures

2.1. Two-site MPS update pipeline and its kernel-level mapping. This is a direct bridge between tensor-network algorithm steps (top) and GPU optimisation targets (bottom). 5

2.2. Full GA100 floorplan schematic with graphics processing clusters (GPCs), L2 cache fabric, memory controllers, and NVLink/PCIe interfaces. This chip-level view provides the context for the SM-level detail in Figure 2.3. 9

2.3. Internal structure of an A100 Streaming Multiprocessor at three levels of detail. **Left:** the SM contains four processing blocks sharing an L1 data cache and shared memory. **Centre:** each processing block includes a warp scheduler, register file slice, execution units, LD/ST units, SFUs, and a texture unit. **Right:** the execution units comprise 16 INT32 cores, 16 FP32 cores, 8 FP64 cores, and one tensor core. 9

2.4. Bit layout comparison of FP64, FP32, TF32, BF16, and FP16, with exponent and significand fields drawn to scale. 11

2.5. Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM). 16

2.6. Memory hierarchy of an A100 GPU, showing registers, shared memory/L1, L2 cache, and HBM global memory. 16

2.7. First-order defect-limited yield sensitivity for a GA100-sized die ($A = 8.26\,\mathrm{cm}^2$) using $Y_{\mathrm{full}} \approx e^{-AD_0}$. Marker points use industry-reported N7 anchor values ($D_0 \approx 0.33$ and $D_0 \approx 0.09$). 18

2.8. NUMA and chiplet hierarchy of a single AMD EPYC 7742 socket. Each socket contains 8 CCDs grouped into 4 NUMA domains (NPS=4 configuration). Each CCD holds two CCXes, each with 4 cores and a private 16 MB L3 cache. All CCDs connect to a central I/O die that houses the memory controllers and PCIe root complexes. 22

2.9. Compute node topology showing the four A100 GPUs (centre) connected via NVLink 3.0, with the two CPU sockets on either side. Three representative host–device paths are highlighted: a fast local PCIe path (GPU 1 $\leftrightarrow$ NUMA 1), an intra-socket Infinity Fabric path (GPU 0 $\leftrightarrow$ NUMA 3), and a cross-socket Infinity Fabric path (GPU 3 $\leftrightarrow$ NUMA 2). 24

2.10. 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`. . 27

2.11. Asynchronous host–device timeline using `cudaMemcpyAsync`, streams, and event-based ordering. Host work overlaps queued device work until explicit synchronisation. 34

2.12. Synchronous host–device timeline using blocking `cudaMemcpy`. The host is stalled during H2D and D2H transfers. 35

3.1. Per-batch matrix view of the two-site update. The tensor expression is equivalent to a two-stage GEMM chain with a shared G and M across batches. 44
3.2. Algorithmic structure of the two-site case study: direct contraction (left) versus contraction-order decomposition into two GEMM-like stages (right). 44
3.3. Arithmetic-intensity map for rectangular GEMM-like kernels in FP32 at fixed $K = 64$. Left: ideal intensity without padding. Right: effective intensity with 16-element padding on M, N, K (useful FLOPs over padded traffic). 51
3.4. Storage model $M(k; s) = 3k^2 s$ for square GEMM in FP32 and FP64. Left: intersection with A100 HBM capacity (40 GB). Right: same model against per-SM shared-memory capacity (164 KiB). 52
3.5. Roofline envelopes for A100-SXM4-80GB. Relative to A100-40GB, the compute ceilings are unchanged while higher memory bandwidth shifts the knees to lower operational intensity. 53
3.6. Roofline envelopes for H100-SXM5-80GB using representative peak throughput and bandwidth values from NVIDIA's Hopper documentation. 53
3.7. Amdahl speedup bounds for representative parallel fractions ($f \in \{0.90, 0.95, 0.99\}$). 54
3.8. Warp-level address mapping as a layout design rule: contiguous thread-to-address mapping improves global-memory transaction efficiency, while large strides fragment traffic. 55
3.9. Standard profiling-and-reporting pipeline used in this thesis. Timeline localisation precedes counter collection, and claim validation loops back when mechanism evidence is inconsistent. 57

5.1. Total time for 1000 GEMM operations as a function of matrix size, comparing 1000 separate kernel launches (red) against a single fused kernel (blue). The near-constant red curve below $n = 16$ confirms that launch overhead dominates over actual computation for small matrices. 64
5.2. Global-memory coalescing sweep over stride. 69
5.3. Register pressure versus throughput and occupancy. 70

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 mantissa column lists only the explicitly stored fraction bits; all formats carry an additional implicit leading bit for normal numbers. 12
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). 12
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]. 15
2.4. Key hardware specifications of the NVIDIA A100 (SXM4-40GB baseline). 17
2.5. A100 SXM4 variant comparison relevant to performance modeling. Data compiled from NVIDIA's A100 architecture whitepaper and product specifications [NVI20a, NVI25c]. 20
2.6. Theoretical peak floating-point throughput of the A100 GPU. 20
2.7. Derived theoretical limits of the A100 architecture. 21
2.8. Approximate memory access latency at different hierarchy levels. . . . 21
2.9. GPU-to-NUMA affinity on a JURECA-DC compute node. 23
2.10. Path-class mapping used for topology-aware cost evaluation. 26
2.11. Working notes: transfer from selected literature to CUDA optimisation actions in this thesis. 39

3.1. Work comparison for the thesis reference shape ($B = 1$, $d_p = 16$, $\chi = 256$). 45
3.2. Main design decisions for the two-site contraction case study. 46
3.3. Representative arithmetic-intensity values for fixed $K = 64$ in FP32 with $t = 16$ padding. 50
3.4. Capacity-limited square dimensions for the GEMM storage model $M(k; s) = 3k^2 s$. 51
3.5. Amdahl speedup bounds for representative parallel fractions. 52
3.6. Mandatory disclosure checklist for this thesis. 58

5.1. Total execution time for 1000 GEMM operations ($n \times n$, FP64) on an A100 GPU. *Separate* denotes 1000 individual kernel launches; *Fused* denotes a single kernel performing all 1000 operations internally. The overhead column gives the fraction of the separate-launch time attributable to launch overhead rather than computation. 63
5.2. KPI identifier to primary metric mapping used for reporting. Scripts keep compatibility fallbacks for Nsight Compute version differences. . . 66
5.3. Coalescing sweep results from the synthetic stride microbenchmark on A100 (FP32 loads/stores). 68

5.4. Register-pressure microbenchmark results on A100 (FP32 arithmetic core work). 69

7.1. Compute node hardware specification. 81
7.2. Software versions used for all experiments. Components marked with † are loaded via the JSC module system. 82
7.3. GPU interconnect topology. NV4 = four bonded NVLinks; PIX = single PCIe bridge; SYS = traverses PCIe and inter-NUMA interconnect. . . 82

Listings

2.1. Element-wise vector addition kernel and its host-side launch. 27
2.2. Matrix transpose using two-dimensional grid and block addressing. . . 28
2.3. Static shared memory allocation for a tile. 30
2.4. Dynamic shared memory allocation. 30
2.5. Tiled matrix multiplication using shared memory. Each block computes one tile of the output matrix C. 30
2.6. Coalesced access pattern: consecutive threads read consecutive columns. 32
2.7. Non-coalesced access pattern: consecutive threads read elements separated by stride N. 32
2.8. Bank conflict when accessing a column of a 32-wide shared array, and the padding fix. 32
2.9. Asynchronous pattern with `cudaMemcpyAsync`, stream ordering, and explicit final synchronisation. 35
2.10. Shape-bucketed small contractions using strided batched GEMM. Representative kernel-orchestration pattern inspired by massively parallel DMRG implementations. 39
2.11. Operator-sum contraction sketch. The core idea is to accumulate local terms directly instead of materialising a large explicit operator tensor first. 40
2.12. Two-stream overlap pattern for distributed dense kernels. The communication stream runs asynchronous collectives while compute proceeds on the main stream. 40

7.1. Build commands for JURECA-DC. 83

List of Symbols

Symbol	Meaning
$\mathcal{T}$	Generic tensor
$\mathbf{A}, \mathbf{B}, \mathbf{C}$	Matrices
$\vec{v}$	Vector
$\Psi(i_1, \ldots, i_L)$	Many-body wavefunction amplitude
L	Number of lattice sites / modes
d	Local physical dimension per site
χ	Bond dimension (MPS rank limit)
$A^{[\ell]}$	Local MPS tensor at site ℓ
α_ℓ	Virtual bond index between neighboring tensors
$N_{\text{state}} = d^L$	Number of coefficients in full-state representation
$N_{\text{MPS}} \sim L\, d\, \chi^2$	Parameter count of a uniform-bond MPS
$S = -\sum_a s_a^2 \log s_a^2$	Entanglement entropy from Schmidt coefficients
ϵ_{trunc}	Truncation error after rank reduction
M, N, K	GEMM dimensions ($\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{K\times N}$)
$F = 2MNK$	Floating-point operation count for dense GEMM
Q	Data movement volume (bytes)
$I = F/Q$	Arithmetic intensity (FLOPs per byte)
P	Achieved performance (FLOPs/s)
$P_{\max}$	Theoretical peak compute throughput
$B_{\max}$	Peak memory bandwidth
$I^\star = P_{\max}/B_{\max}$	Roofline ridge point
$\eta = P/P_{\max}$	Compute efficiency relative to peak
$T_{\text{H2D}}(\pi, V)$	Host-to-device transfer time for path class π and volume V
$T_0(\pi)$	Fixed path-dependent transfer latency term
$B_{\text{eff}}(\pi)$	Effective end-to-end transfer bandwidth
V	Transfer volume in bytes
π	Path class (local, same-socket remote, cross-socket)
n_{IF}	Number of additional Infinity Fabric segments in the path
τ_{IF}	Average latency penalty per Infinity Fabric segment
$\mathcal{O}(\cdot)$	Asymptotic upper bound

List of Abbreviations

BF16	Brain Floating Point (16-bit)
BLAS	Basic Linear Algebra Subprograms
CCX	Core Complex (AMD)
CCD	Core Complex Die (AMD)
CUDA	Compute Unified Device Architecture
DFT	Density Functional Theory
DMRG	Density Matrix Renormalization Group
FMA	Fused Multiply-Add
FP8/16/32/64	IEEE 754 Quarter/Half/Single/Double Precision
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 (NVIDIA CUDA)
NUMA	Non-Uniform Memory Access
NVLink	NVIDIA GPU Interconnect
PCIe	Peripheral Component Interconnect Express
SFU	Special Function Unit
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SVD	Singular Value Decomposition
TF32	TensorFloat-32
TN	Tensor Network
ULP	Unit in the Last Place

Glossary

Arithmetic intensity
: The ratio of floating-point operations to bytes transferred from memory, typically measured in FLOPs/byte. Determines whether a kernel is compute-bound or memory-bound (see *roofline model*).

Bank conflict
: A shared memory access pattern in which multiple threads in a warp address different words in the same memory bank, forcing the accesses to be serialised.

Bond dimension
: In tensor networks, the size of an index shared between two tensors. Larger bond dimensions permit more accurate representations but increase computational and memory cost.

Coalescing
: The hardware mechanism by which individual memory requests from threads in a warp are combined into a minimal number of cache-line transactions. Requires consecutive threads to access consecutive addresses.

Contraction
: The generalisation of matrix multiplication to tensors: a pairwise summation over one or more shared indices between two tensors, producing a new tensor.

Kernel
: A function written in CUDA that executes in parallel across many GPU threads. Launched from the host (CPU) and runs on the device (GPU).

Machine epsilon
: The smallest floating-point number ε such that $1 + \varepsilon \neq 1$ in a given format. Characterises the relative spacing of representable values near 1.

Mixed precision
: A computational strategy that uses lower-precision arithmetic (e.g. FP16, BF16, TF32) for the bulk of computation while maintaining higher-precision accumulators (e.g. FP32) to preserve numerical accuracy.

Occupancy
: The ratio of active warps on an SM to the maximum number of warps the SM supports. Higher occupancy generally improves latency hiding but is not always necessary for peak performance.

Roofline model
: A performance model that bounds achievable throughput by the minimum of peak compute throughput and peak memory bandwidth multiplied by arithmetic intensity. Used to classify kernels as compute-bound or memory-bound.

Shared memory Fast, on-chip memory visible to all threads within a CUDA thread block. Used as a programmer-managed cache and for inter-thread communication.

SIMT Single Instruction, Multiple Threads. NVIDIA's execution model in which a warp of 32 threads executes the same instruction simultaneously, similar to SIMD in Flynn's taxonomy but with the ability for individual threads to follow divergent control-flow paths (at the cost of serialisation).

Tensor core A specialised hardware unit on NVIDIA GPUs that performs small matrix multiply-accumulate operations (e.g. 4×4 or 8×8) in a single cycle, providing significantly higher throughput than standard CUDA cores for supported precisions.

Tiling A loop transformation that partitions a computation into smaller blocks (tiles) to improve data locality and reuse in caches or shared memory.

Warp A group of 32 threads that execute in lock-step on the same SM processing block. The warp is the fundamental scheduling unit on NVIDIA GPUs.

Warp divergence A performance penalty that occurs when threads within a warp take different branches of a conditional. The hardware serialises the divergent paths, reducing effective parallelism.

1. Introduction

1.1. Motivation

Tensor-network simulations increasingly rely on GPU-accelerated dense kernels, yet practical workloads often run in a regime where library defaults are not fully efficient: matrix blocks are small or irregular, kernel sequences are short, and memory-layout transforms are frequent. In this regime, launch overhead, data movement, and occupancy limits can dominate the total runtime.

This thesis is conducted in collaboration with the Jülich Supercomputing Centre (JSC), where partner teams in computational physics require reliable performance gains on production HPC systems. The immediate hardware target is Ampere-class GPUs, in particular A100 nodes, with a conservative baseline of one node (four A100-SXM4-40GB GPUs connected via NVLink).

The goal is practical: build a profiling-driven kernel-optimisation workflow that produces measurable speedups for tensor-network contraction workloads on current JSC hardware and can be carried over to next-generation accelerators.

1.2. Problem Statement

The central problem is to identify and optimise the CUDA kernels and execution patterns that dominate runtime in tensor-network contraction workflows, while keeping the scope on HPC implementation aspects rather than physics modelling.

The work is organised around the following technical questions:

1. Which kernel patterns are the primary bottlenecks on A100 for the target workload distribution (small/medium contractions, layout transforms, short kernel sequences)?

2. How much performance can be recovered through launch reduction, kernel fusion, data-layout control, and occupancy-aware tuning?

3. How closely do optimised kernels approach hardware limits predicted by bandwidth and throughput models, and where do residual stalls remain?

The thesis addresses these questions with a measurement-first methodology based on reproducible benchmarks and Nsight profiling, and compares custom kernels against established vendor baselines (cuBLAS/cuTENSOR).

1.3. Contributions

The contributions of this thesis are:

1. A hardware-aware characterisation of tensor-network-relevant kernel patterns on Ampere GPUs, with emphasis on launch-dominated and bandwidth-limited regimes.

2. An implementation strategy for fused and layout-aware CUDA kernels, including explicit occupancy and memory-hierarchy trade-off analysis.

3. A profiling workflow that links observed performance counters to concrete optimisation decisions and documents where bottlenecks shift after each optimisation step.

4. A benchmark and comparison framework against cuBLAS/cuTENSOR baselines on a realistic single-node A100 environment.

1.4. Outline

Chapter 2 introduces the CUDA and hardware concepts required for the optimisation work, and frames tensor networks only as computational motivation. Chapter 3 defines the target kernels, baseline choices, and data-layout strategy. Chapter 4 details the kernel design and integration choices used to realise the optimisation approach. Chapter 5 reports benchmark and profiling results, including launch overhead experiments and comparison against library baselines. Chapter 6 summarises findings, limitations, and next optimisation steps.

2. Background

Chapter 2 covers the HPC background used in the optimisation work. Tensor networks appear only as computational motivation; the focus is GPU architecture, CUDA execution, and prior kernel-engineering results.

2.1. Tensor Networks

2.1.1. Historical Background

Tensor networks entered computational physics through the density-matrix renormalization group (DMRG) in the early 1990s, where the key practical idea was to keep only the most relevant subspace during iterative updates [Whi92, Whi93]. During the 2000s, this was reformulated in explicit tensor-network language, especially as matrix product states (MPS), which clarified the relation between approximation quality and bond dimension [Sch11, Oru14].

For high-performance computing, the first generation of tensor-network codes was CPU-focused and built around BLAS/LAPACK kernels plus MPI-based distribution. Over time, the contraction-heavy parts moved to GPUs, where cuBLAS/cuTENSOR-style kernels and communication-aware scheduling became central for strong scaling across devices [SSK17, ML$^+$23b, ML$^+$24]. The algorithmic core stayed similar, but the main performance bottlenecks shifted toward memory movement, launch overhead, and multi-GPU orchestration.

2.1.2. Why Tensor Networks Are of Interest

Tensor networks are useful because they replace exponential state representations with structured low-rank factorizations when entanglement is limited. For a chain of L sites with local dimension d, the full state vector requires

$$N_{\text{state}} = d^L$$

complex amplitudes. In contrast, a uniform-bond MPS uses

$$N_{\text{MPS}} \sim L\, d\, \chi^2,$$

which is polynomial in system size for fixed bond dimension χ.

This reduction is directly tied to entanglement structure. Across one bipartite cut, an MPS corresponds to a truncated Schmidt decomposition

$$|\Psi\rangle = \sum_{a=1}^{\chi} s_a \, |a_L\rangle \, |a_R\rangle \,,$$

and the von Neumann entropy

$$S = -\sum_a s_a^2 \log s_a^2$$

is effectively controlled by the retained rank χ.

From an HPC perspective, tensor networks are also interesting because they are a stress test for modern GPUs: many dense contractions, repeated reshapes, and long iterative workflows with short kernels. This combination makes them a good case for studying both kernel-level efficiency and system-level orchestration costs [ZSW20, A$^+$23].

2.1.3. Computational Context

Tensor-network methods arise in quantum many-body simulation, where the full state space of the Schrödinger equation is reduced to a structured set of lower-rank objects. In this thesis, tensor networks are treated as an *application source of computational patterns*, not as a physics topic. Detailed notation, model assumptions, and algorithmic physics choices are outside scope here and are introduced separately with domain experts.

From a high-performance computing perspective, the relevant point is that tensor-network codes repeatedly execute dense linear-algebra-like kernels on moderate to small tensor blocks. Those kernels are often embedded in long iterative workflows, so both per-kernel efficiency and orchestration overhead matter.

2.1.4. Matrix Product States (MPS) from a Computational View

Matrix Product States (MPS) are a common tensor-network representation where a high-order object is factorised into a chain of local tensors. For this thesis, the relevant point is not the physics interpretation but the resulting compute structure:

- repeated contraction of small and medium dense tensors,
- frequent reshape/permutation steps between contraction phases,
- recurring low-rank truncation steps (typically SVD/QR-based),
- many short kernel sequences within iterative sweeps.

Using a minimal notation, an order-L MPS can be written as

$$\Psi(i_1, \ldots, i_L) = \sum_{\alpha_1, \ldots, \alpha_{L-1}} A^{[1]}_{i_1,\alpha_1} A^{[2]}_{\alpha_1,i_2,\alpha_2} \cdots A^{[L]}_{\alpha_{L-1},i_L},$$

where i_ℓ is a physical index and α_ℓ is a bond index. The bond dimension χ controls both approximation quality and computational cost.

From an HPC perspective, two scaling facts are useful planning rules:

1. storage for a uniform MPS scales as $\mathcal{O}(L\, d\, \chi^2)$ rather than exponentially in L,
2. dominant kernels in common MPS updates scale roughly with $\mathcal{O}(d\, \chi^3)$ per local update step, so increasing χ quickly shifts pressure to memory traffic and tensor-factorisation kernels.

A typical two-site update can be sketched with a simple cost model:

$$\text{FLOPs}_{\text{update}} \approx c_1\, d\, \chi^3 + c_2\, d^2\, \chi^2,$$

where the first term represents contraction work and the second represents truncation/-factorization work. Constants c_1, c_2 depend on layout and update scheme, but the scaling makes clear why χ is the dominant performance driver.

In practical implementations, local updates are often executed as:

1. fuse neighboring local tensors into an effective two-site tensor,
2. contract with environment/operator terms,
3. reshape to a matrix and run SVD/QR-style truncation,
4. redistribute factors back into MPS form.

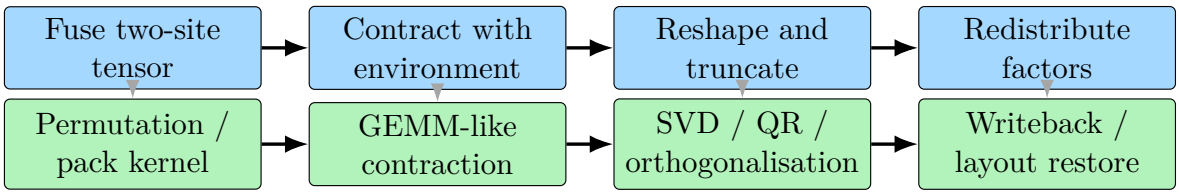

Figure 2.1.: Two-site MPS update pipeline and its kernel-level mapping. This is a direct bridge between tensor-network algorithm steps (top) and GPU optimisation targets (bottom).

The same structure appears in the optimisation targets of later chapters: GEMM-like contractions, layout transforms, launch-overhead control, and profiling-guided tuning of memory/occupancy trade-offs.

2.1.5. Kernel Patterns Relevant to This Thesis

The optimisation work in this thesis is driven by three recurring patterns:

1. **Small and medium dense contractions.** After index reshaping, many contractions reduce to GEMM-like operations with dimensions that are too small to saturate the GPU when launched individually.
2. **Layout transforms.** Permutations and reshapes are required to match library expectations (e.g. column-major interfaces), and these transforms can dominate runtime if memory access is not coalesced.
3. **Fine-grained kernel sequences.** Practical workflows execute large numbers of short kernels; launch overhead and synchronisation overhead become first-order effects in this regime (see Sections 2.4.5 and 5.2).

This computational profile aligns directly with CUDA optimisation topics such as occupancy control, shared-memory tiling, register pressure, launch fusion, and host-device scheduling.

2.1.6. Scope Boundary

To keep the thesis focused and technically coherent, we explicitly exclude:

- derivations of tensor-network physics models,
- detailed notation systems used in specific communities,
- algorithmic comparisons driven primarily by physics accuracy criteria.

Instead, we evaluate kernels by HPC criteria: runtime, achieved throughput, memory efficiency, and scaling behaviour on Ampere-class GPUs.

2.2. Mathematical Foundations for Performance Evaluation

Core mathematical tools for performance interpretation and numerical reliability are collected in this section. It is not a full theory treatment; the goal is a compact framework that links measurements to well-defined models.

2.2.1. Work, Data Movement, and Runtime Bounds

For a kernel with floating-point work F (FLOPs) and main-memory traffic Q (bytes), a first-order runtime model is

$$T \approx \max\left(\frac{F}{P_{\text{eff}}},\ \frac{Q}{B_{\text{eff}}}\right) + T_{\text{launch}},$$

where P_{eff} and B_{eff} are achieved compute and bandwidth rates, and T_{launch} is host/device dispatch overhead. The first term is a roofline-style throughput bound, while the additive launch term follows the latency-plus-throughput structure used in communication models such as Hockney's [WWP09, HJ88].

The corresponding arithmetic intensity

$$I = \frac{F}{Q}$$

is a primary explanatory variable because it determines whether a kernel is expected to be memory-bound or compute-bound [WWP09]. Under ideal sustained ceilings $(P_{\max}, B_{\max})$, throughput is bounded by

$$P \le \min(P_{\max},\ B_{\max} I)\,.$$

This gives the standard ridge point

$$I^{\star} = \frac{P_{\max}}{B_{\max}},$$

which separates memory-bound and compute-bound regimes.

2.2.2. Parallel Scaling Metrics

Given runtime T_p on p processing units, speedup and parallel efficiency are

$$S_p = \frac{T_1}{T_p}, \qquad E_p = \frac{S_p}{p}.$$

For strong scaling with serial fraction f_s, Amdahl's bound is

$$S_p \le \frac{1}{f_s + \frac{1-f_s}{p}}.$$

For weak scaling with scaled problem size, Gustafson's relation is

$$S_p \approx p - f_s(p-1).$$

These bounds are used to interpret node-level scaling results in Section 5.4 [Amd67, Gus88].

2.2.3. Floating-Point Error Model

For one floating-point operation, we use the standard model

$$\mathrm{fl}(a \circ b) = (a \circ b)(1+\delta), \qquad |\delta| \le u,$$

with unit roundoff u and $\circ \in \{+, -, \times, /\}$.

Conditioning explains why addition/subtraction can be much more sensitive than multiplication. A local relative condition number for addition is

$$\kappa_{\mathrm{add}}(x, y) = \frac{|x| + |y|}{|x + y|},$$

which becomes large under cancellation ($x \approx -y$). By contrast, multiplication has $\kappa_{\mathrm{mul}}(x, y) = 1$ under standard relative perturbation analysis [Hig02]. A concise engineering-focused discussion of conditioning and cancellation is also given in [DR22]. For dot products, a useful problem-condition measure is

$$\kappa_{\mathrm{dot}}(x, y) = \frac{|x|^\top |y|}{|x^\top y|},$$

which again grows when positive and negative contributions cancel.

Two concrete values make the conditioning effect explicit:

$$\kappa_{\mathrm{add}}(1, 0.8) = \frac{|1| + |0.8|}{|1 + 0.8|} = \frac{1.8}{1.8} = 1,$$

while under near-cancellation

$$\kappa_{\mathrm{add}}(1, -0.999999) = \frac{|1| + |-0.999999|}{|1 - 0.999999|} = \frac{1.999999}{10^{-6}} \approx 2 \times 10^6.$$

The corresponding multiplication stays locally well-conditioned:

$$\kappa_{\mathrm{mul}}(1, -0.999999) = 1.$$

For a length-k dot product, a common bound is

$$\left|\mathrm{fl}(x^\top y) - x^\top y\right| \le \gamma_k \, |x|^\top |y|, \qquad \gamma_k = \frac{ku}{1-ku},$$

which makes explicit why long reduction chains are sensitive to precision choice. This is one reason why FP32 accuracy checks are kept in the workflow even when throughput is the primary objective [HM22].

This also motivates fused multiply-add and FP32 accumulation in tensor-core paths: IEEE-754 FMA semantics avoid intermediate rounding between multiply and add [IEE19], and Ampere TF32 tensor-core execution keeps accumulation in FP32 [NVI20a], reducing the impact of repeated low-precision reduction steps.

2.2.4. Timing Statistics

For repeated runtime samples $\{t_i\}_{i=1}^n$, this thesis reports the sample median

$$\tilde{t} = \mathrm{median}(t_i)$$

as the primary point estimate, because it is robust to occasional outliers from scheduler and system noise. This is preferable to the arithmetic mean when runtime distributions are skewed or contain sporadic long-tail events from OS, driver, or queue interference [Jai91, KJ13].

Dispersion is tracked with the median absolute deviation (MAD):

$$\mathrm{MAD} = \mathrm{median}\Big(|t_i - \tilde{t}|\Big)\,,$$

and a robust spread estimate

$$\hat{\sigma}_{\mathrm{rob}} \approx 1.4826 \cdot \mathrm{MAD}\,.$$

The scaling factor makes $\hat{\sigma}_{\mathrm{rob}}$ comparable to a standard deviation under a normal reference model, while retaining robustness under heavy-tailed noise [HR09].

Uncertainty of the median is reported with a nonparametric bootstrap confidence interval. With B resamples (typically $B = 1000$), let $\tilde{t}^{(b)}$ be the median of bootstrap replicate b. The percentile interval is

$$\mathrm{CI}_{95\%} = \left[q_{0.025}\Big(\{\tilde{t}^{(b)}\}_{b=1}^B\Big)\,, q_{0.975}\Big(\{\tilde{t}^{(b)}\}_{b=1}^B\Big)\right],$$

which avoids normality assumptions on runtime samples [ET94].

For reporting stability, this thesis also tracks the robust relative spread

$$\mathrm{RRS} = \frac{\hat{\sigma}_{\mathrm{rob}}}{\tilde{t}},$$

and increases repetition count when RRS is high. This keeps timing conclusions stable in short benchmark campaigns.

2.3. GPU Architecture

2.3.1. Streaming Multiprocessor and Warp Execution

The Streaming Multiprocessor (SM) is the fundamental compute unit of the GPU. Each A100 GPU contains 108 SMs, and all CUDA threads ultimately execute on one of them. Understanding the internal structure of an SM is essential for reasoning about occupancy, register pressure, and warp scheduling.

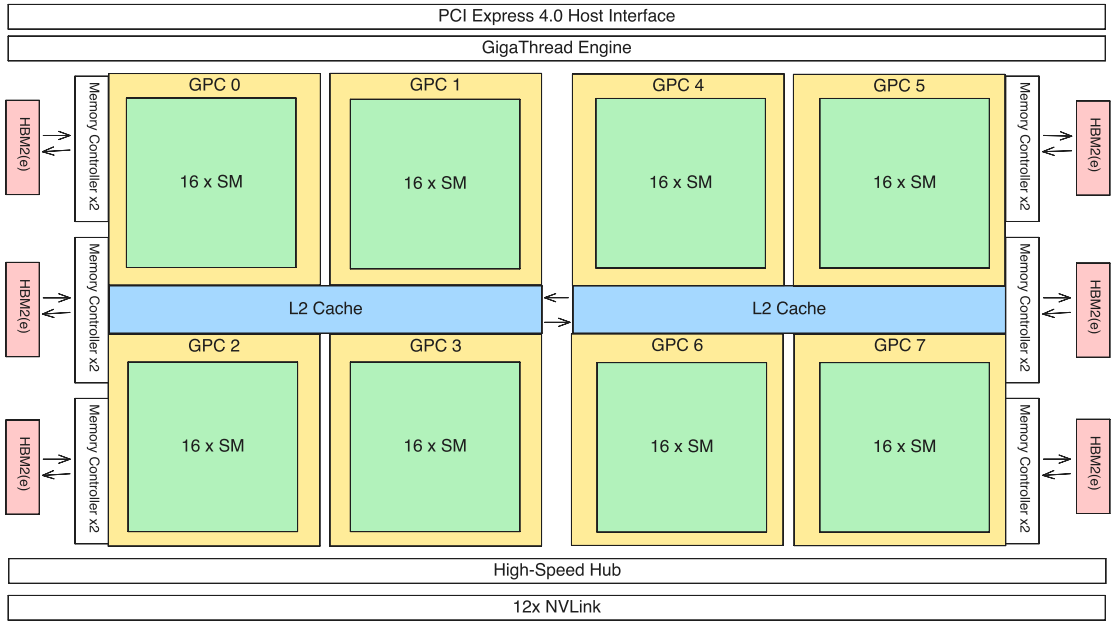

Figure 2.2.: Full GA100 floorplan schematic with graphics processing clusters (GPCs), L2 cache fabric, memory controllers, and NVLink/PCIe interfaces. This chip-level view provides the context for the SM-level detail in Figure 2.3.

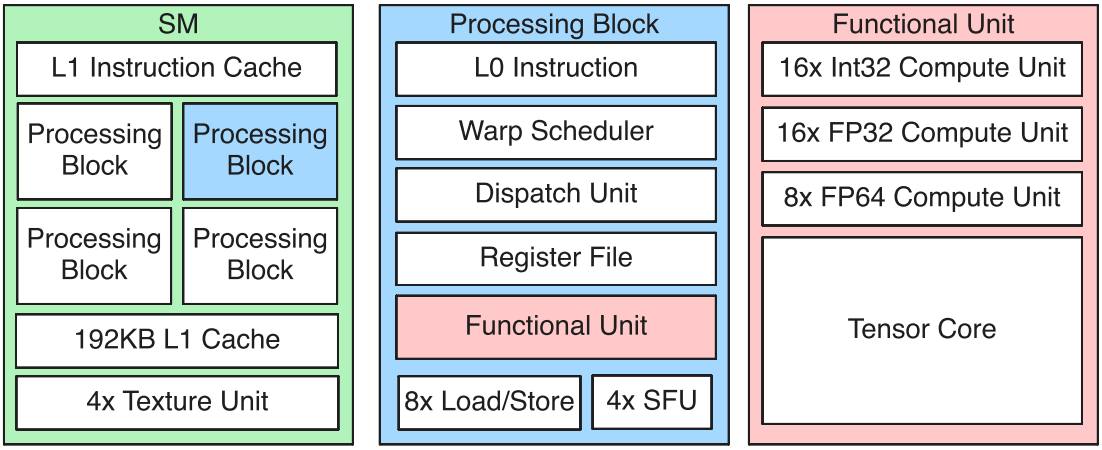

Figure 2.3.: Internal structure of an A100 Streaming Multiprocessor at three levels of detail. **Left:** the SM contains four processing blocks sharing an L1 data cache and shared memory. **Centre:** each processing block includes a warp scheduler, register file slice, execution units, LD/ST units, SFUs, and a texture unit. **Right:** the execution units comprise 16 INT32 cores, 16 FP32 cores, 8 FP64 cores, and one tensor core.

Each SM is partitioned into four *processing blocks* (also called sub-partitions). As shown in Figures 2.2 and 2.3, GA100 is organized hierarchically from chip-level GPC groupings down to the internals of each SM. At the SM level, each processing block contains:

- One **warp scheduler** with a single dispatch unit, capable of issuing one instruction per cycle from the warp it selects.
- A slice of the **register file** (16 384 32-bit registers per block, 65 536 per SM).
- **Execution units:** 16 INT32 cores, 16 FP32 cores, 8 FP64 cores, and one third-generation tensor core.
- **8 LD/ST units** for issuing memory load and store operations.
- **4 SFUs** (Special Function Units) for transcendental operations such as `sin`, `exp`, and reciprocal square root.
- A **read-only/texture path** for cached read-mostly memory access.
- An **L0 instruction cache** private to the processing block.

The four processing blocks share the SM's L1 data cache and its configurable shared memory (up to 164 kB on the A100). Threads are grouped into *warps* of 32 and assigned to processing blocks at launch time. Each warp scheduler selects one of its resident warps per cycle and issues a single instruction to the appropriate execution unit. Because the four schedulers operate independently, an SM can have four instructions from four different warps in flight simultaneously, which is the primary mechanism for hiding memory and instruction latency.

2.3.2. What Is New in A100 for This Thesis

Relative to the previous Volta generation, the A100 changes several hardware limits that matter directly for tensor-network kernels [NVI20a, NVI25d]:

- **Third-generation tensor cores and TF32 mode.** FP32 inputs can be accelerated on tensor cores through TF32 execution without changing storage format, which improves throughput for GEMM-like contractions.
- **Larger on-chip memory budget.** The A100 provides up to 164 kB shared memory per SM and a 40 MB L2 cache, increasing tile reuse opportunities and reducing pressure on HBM.
- **Asynchronous global-to-shared copy.** Ampere introduces hardware support for overlap-friendly copy pipelines (`cp.async`), which is important for latency hiding in custom kernels.
- **Stronger inter-GPU fabric.** NVLink 3.0 bandwidth makes multi-GPU contraction decomposition practical with lower communication penalty.

For performance modeling, these changes can be summarized with a roofline bound

$$P \leq \min\Big(P_{\max},\ B_{\max}\, I\Big),$$

where I is arithmetic intensity (FLOPs/byte). The ridge point

$$I^{\star} = \frac{P_{\max}}{B_{\max}}$$

is the minimum intensity needed for compute-bound execution. Using A100 FP32 peaks ($P_{\max} \approx 19.5\,\text{TFLOPS}$, $B_{\max} \approx 1555\,\text{GB/s}$), we get $I^{\star} \approx 12.5$ FLOPs/byte, which is the practical threshold used throughout this thesis.

2.3.3. Floating-Point Formats and Precision Trade-offs

Scientific GPU workloads have often defaulted to FP64, following long-standing CPU-side numerical practice. A100 supports multiple floating-point formats with very different throughput, and for many contraction workloads FP32 or reduced precision is accurate enough at much lower cost. The following subsections summarize the relevant formats and motivate the precision choices used here.

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 bias can be computed as

$$2^{k-1} - 1$$

where k is the number of exponent bits.

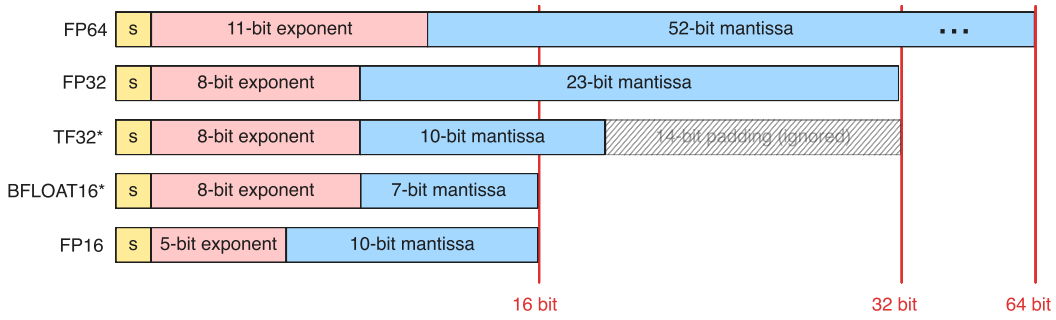

Figure 2.4.: Bit layout comparison of FP64, FP32, TF32, BF16, and FP16, with exponent and significand fields drawn to scale.

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 mantissa column lists only the explicitly stored fraction bits; all formats carry an additional implicit leading bit for normal numbers.

Format	Total bits	Sign	Exponent	Bias	Mantissa[1]
FP64 (double)	64	1	11	1023	52
FP32 (single)	32	1	8	127	23
TF32	19	1	8	127	10
BF16 (bfloat16)	16	1	8	127	7
FP16 (half)	16	1	5	15	10
FP8 (E5M2)[2]	8	1	5	15	2
FP8 (E4M3)[2]	8	1	4	7	3

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.

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	$\approx 6.1 \times 10^{-5}$	$\approx 9.8 \times 10^{-4}$

Two observations are relevant here. First, TF32 and BF16 share the same 8-bit exponent as FP32, so they retain the FP32 dynamic range despite having fewer significand bits. TF32 was introduced for Ampere tensor cores: inputs are stored as FP32 values, while the tensor core internally truncates to 10 significand bits before multiply-accumulate, with accumulation in FP32 [NVI20a]. No explicit data conversion is required in user code.

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).

Examples

Example: Encoding a decimal number in FP16. decimal number 0.15625_{10} has the binary representation 0.00101_2 because

$$0.15625 = \frac{1}{8} + \frac{1}{32} = 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}$$

holds. To express this as an FP16 number we write it in normalised form,

$$0.00101_2 = 1.01_2 \times 2^{-3},$$

and match it against Equation (2.1): the sign is $s = 0$ (positive), the true exponent is -3, and the stored exponent is $e = -3 + 15 = 12 = 01100_2$. The fraction field stores the bits after the implicit leading 1, i.e. 01 padded with zeros to 10 bits. The complete bit representation is therefore

$$\underbrace{0}_{\text{sign}} \mid \underbrace{01100}_{\text{exp}} \mid \underbrace{0100000000}_{\text{mantissa}}.$$

Example: Absorption in FP16 addition. Consider adding 1024.0 and 0.5 in half precision. The value $1024.0 = 1.0 \times 2^{10}$ is stored as

$$\underbrace{0}_{\text{sign}} \mid \underbrace{11001}_{\text{exp=25}} \mid \underbrace{0000000000}_{\text{mantissa}},$$

while $0.5 = 1.0 \times 2^{-1}$ is stored as

$$\underbrace{0}_{\text{sign}} \mid \underbrace{01110}_{\text{exp=14}} \mid \underbrace{0000000000}_{\text{mantissa}}.$$

To perform the addition, the hardware aligns the exponents by shifting the smaller operand's significand to the right by $10 - (-1) = 11$ positions:

$$0.5 = \underbrace{0.00000000001}_{\text{11 places after the radix point}} \times 2^{10}.$$

Since FP16 retains only 10 fraction bits after the implicit leading 1, the shifted significand falls entirely outside the representable precision and is rounded to zero. The result is 1024.0—the smaller operand has been *absorbed*.

Had the same computation been carried out in FP32 (23 fraction bits), the shift of 11 places is well within the available precision and the result is correctly obtained as 1024.5. This example illustrates why mixed-precision strategies—performing arithmetic at higher precision than the storage format—can recover accuracy at modest cost.

Example: TF32 truncation in a matrix multiply. The A100 tensor cores accept FP32 inputs but internally truncate the significand from 23 to 10 fraction bits before performing the multiply, while the accumulation remains in full FP32. To see the effect, consider two 2×2 matrices with entries drawn from familiar constants:

$$A = \begin{pmatrix} 3.1415927 & 1.2345679 \\ 2.7182817 & 0.9876543 \end{pmatrix}, \qquad B = \begin{pmatrix} 0.9876543 & 2.7182817 \\ 1.2345679 & 3.1415927 \end{pmatrix}.$$

After TF32 truncation, the entries lose their lower 13 significand bits:

$$\tilde{A} = \begin{pmatrix} 3.1406250 & 1.2343750 \\ 2.7167969 & 0.9873047 \end{pmatrix}, \qquad \tilde{B} = \begin{pmatrix} 0.9873047 & 2.7167969 \\ 1.2343750 & 3.1406250 \end{pmatrix}.$$

The tensor core computes $C_{\mathrm{TF32}} = \tilde{A}\,\tilde{B}$ with the multiplies at TF32 precision and the accumulation in FP32, giving

$$C_{\mathrm{TF32}} = \begin{pmatrix} 4.6244 & 12.4091 \\ 3.9010 & 10.4817 \end{pmatrix},$$

whereas exact FP32 arithmetic yields

$$C_{\mathrm{FP32}} = A\,B = \begin{pmatrix} 4.6270 & 12.4182 \\ 3.9040 & 10.4919 \end{pmatrix}.$$

The relative error in each entry is on the order of 5×10^{-4} to 10^{-3}, consistent with TF32's machine epsilon of $\varepsilon \approx 9.8 \times 10^{-4}$.

In practice, the entries of typical tensor contractions are not deliberately crafted to maximise the impact of truncation, and the FP32 accumulation across many products causes individual truncation errors to partially cancel. Empirical studies consistently report that TF32 is a drop-in replacement for FP32 in deep-learning training [NVI20a]; whether the same holds for a given scientific workload must still be validated case by case in the benchmark chapter.

Remark: Fused multiply-add and single rounding. Modern GPU floating-point units implement the *fused multiply-add* (FMA) operation

$$\mathrm{fma}(a, b, c) = \mathrm{round}(a \times b + c),$$

where the product $a \times b$ is computed to *full* (unrounded) precision before the addition and only a single rounding step is applied to the final result. By contrast, the non-fused sequence

$$t = \mathrm{round}(a \times b), \qquad r = \mathrm{round}(t + c)$$

incurs two rounding errors. The difference is most visible when $a \times b$ and c are of similar magnitude but opposite sign, so that their sum is much smaller than either operand.

As a concrete example, let $a = b = 1 + 2^{-12}$ and $c = -(1 + 2^{-11})$, all exactly representable in FP32. The exact product is

$$a \times b = (1 + 2^{-12})^2 = 1 + 2^{-11} + 2^{-24},$$

so the exact result is $a \times b + c = 2^{-24} \approx 5.96 \times 10^{-8}$. In the non-fused path, the intermediate rounding of $a \times b$ to FP32 discards the 2^{-24} term (which is below one unit in the last place at magnitude ≈ 1), yielding $t = 1 + 2^{-11}$, and consequently $r = t + c = 0$—a complete loss of the true result. The FMA, retaining the full product before the addition, returns 2^{-24} correctly.

This property is directly relevant to tensor core computation: the multiply-accumulate $D = A \times B + C$ is implemented as a sequence of fused multiply-adds, and the single-rounding semantics mean that the accumulated dot products are more accurate than a naïve loop of separate multiplies and additions would be.

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.

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 throughput differences are large. Moving from FP64 CUDA core arithmetic (9.7 TFLOPS) to FP32 (19.5 TFLOPS) doubles throughput at no algorithmic cost. Using tensor cores in TF32 mode raises peak throughput to 156 TFLOPS, and FP16/BF16 tensor-core throughput reaches 312 TFLOPS (a 32× factor over FP64). Many kernels remain memory-bandwidth-bound, but switching from 64-bit to 32-bit values still halves memory traffic and directly benefits bandwidth-limited sections.

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 2.3.4), 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 Chapter 5. 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.

2.3.4. Memory Hierarchy

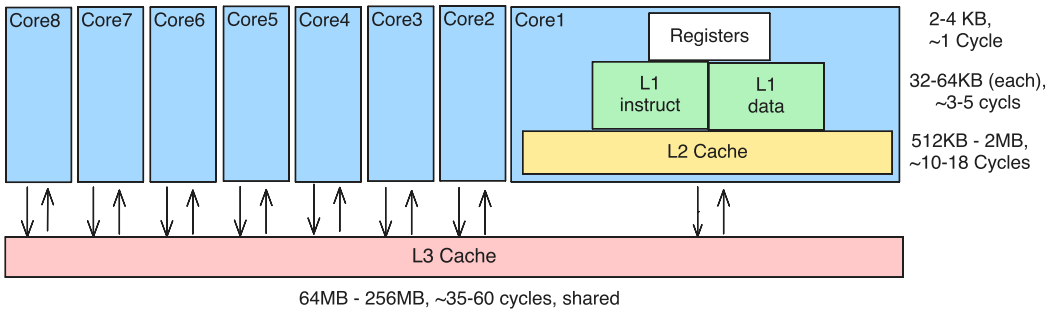

Figure 2.5.: Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM).

As shown in Figure 2.5, modern CPUs rely on a deep cache hierarchy to reduce effective memory latency.

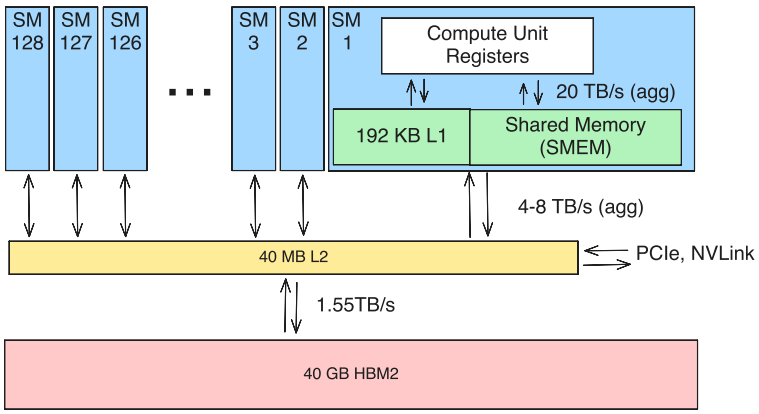

Figure 2.6.: Memory hierarchy of an A100 GPU, showing registers, shared memory/L1, L2 cache, and HBM global memory.

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.

- **HBM (global memory)** provides the bulk storage. The A100-SXM4-40GB used in this thesis provides 40 GB HBM with a peak bandwidth of approximately 1555 GB/s (Table 2.4). Access latency, however, is roughly 500× that of a register access (Table 2.8).

Most scientific GPU kernels, including tensor contractions, are memory-bandwidth-bound rather than compute-bound. Performance depends on reuse in registers and shared memory, plus coalesced, bandwidth-efficient HBM access patterns.

2.3.5. NVIDIA A100 Ampere Architecture

Hardware Overview

Table 2.4.: Key hardware specifications of the NVIDIA A100 (SXM4-40GB baseline).

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)	65 536
Shared memory per SM (KB)	164
L2 cache size (MB)	40
HBM memory capacity (GB)	40
Peak memory bandwidth (GB/s)	1555
Base clock frequency (GHz)	1.41

The A100 employs high-bandwidth memory as its main device memory, providing the throughput required to sustain dense linear algebra kernels.

Large-Die Yield Context (TSMC N7, Approximate)

GA100 is a very large monolithic die (about 826 mm^2) with 128 physical SM partitions in the full design, while shipping A100 parts expose 108 active SMs in the SXM4-40GB product configuration [NVI20a, NVI20b]. This subsection gives a first-order yield sensitivity estimate to provide economic context for why large dies are commonly harvested/binned.

Using the standard first-order Poisson defect model for full-die yield,

$$Y_{\text{full}}(A, D_0) \approx e^{-AD_0},$$

where A is die area in cm^2 and D_0 is defect density in defects/cm^2. For GA100, $A = 8.26$.

Absolute TSMC N7 defect-density time series are not publicly released in full. However, industry-reported anchor values from public disclosures place $D_0 \approx 0.33$ defects/cm^2 in a pre-MP reference regime and $D_0 \approx 0.09$ defects/cm^2 for N7 around three quarters after high-volume manufacturing ramp [Mel24, Fru20]. Figure 2.7 plots the resulting full-die-yield sensitivity.

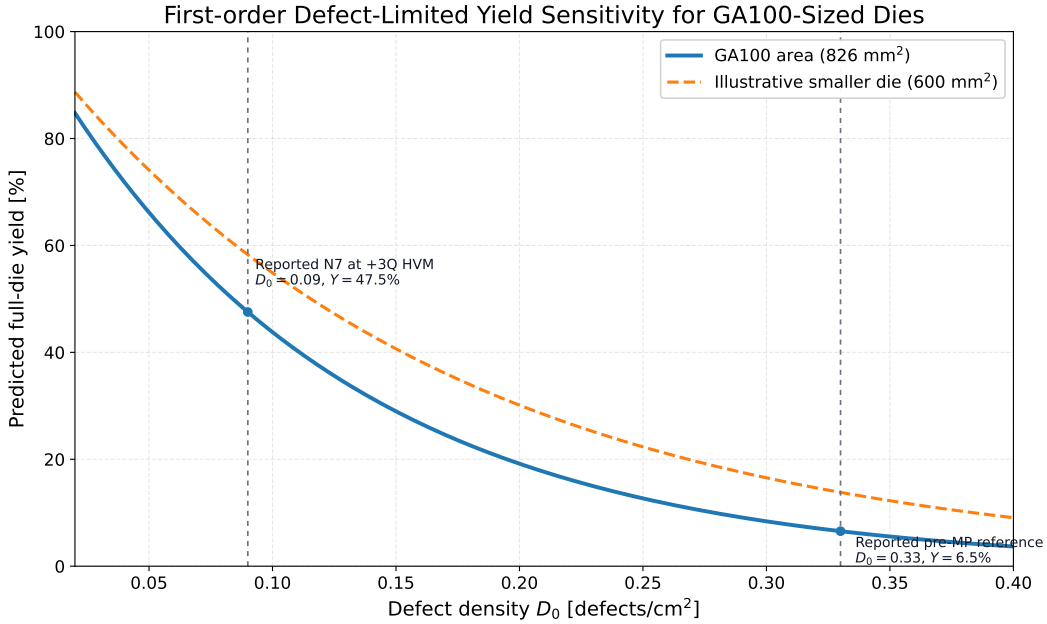

Figure 2.7.: First-order defect-limited yield sensitivity for a GA100-sized die ($A = 8.26\,\text{cm}^2$) using $Y_{\text{full}} \approx e^{-AD_0}$. Marker points use industry-reported N7 anchor values ($D_0 \approx 0.33$ and $D_0 \approx 0.09$).

At those two anchor points, the model gives

$$Y_{\text{full}}(D_0 = 0.33) \approx 6.6\%, \qquad Y_{\text{full}}(D_0 = 0.09) \approx 47.5\%.$$

This should not be interpreted as NVIDIA's product yield: shipping yield is higher due to redundancy, harvesting (disabled units), and performance binning. The model is used here only to quantify the sensitivity of large-die economics to defect density.

Area increment sanity check. For a die-area increase ΔA, Poisson yield scales as

$$\frac{Y(A + \Delta A, D_0)}{Y(A, D_0)} = e^{-D_0 \Delta A}.$$

With $\Delta A = 0.1\,\mathrm{cm}^2$, this multiplier is about 0.968 at $D_0 = 0.33$ and 0.991 at $D_0 = 0.09$, i.e. a 3.2 % and 0.9 % relative yield drop, respectively.

GA100 Die vs A100 Module: What Is Around the Chip

It is useful to distinguish the *GA100* silicon die from the complete *A100 accelerator module*. GA100 is the GPU chip itself (compute cores, caches, memory controllers, and I/O logic). The A100 is the packaged and integrated product used in systems (SXM4 or PCIe form factor), which adds memory stacks, package/interposer, power delivery, and board/module interfaces.

For architectural reasoning in this thesis, the main package-level components around GA100 are:

- **HBM stacks on the same package:** A100 places high-bandwidth memory adjacent to the GPU die in-package, enabling a very wide memory interface with short electrical paths.
- **Memory controllers and links:** the A100 implementation exposes five active HBM stacks and ten 512-bit memory controllers (effective 5120-bit memory interface) [NVI20b, NVI20a].
- **Interposer/package fabric:** GPU die and HBM stacks are integrated in one package so bandwidth is far higher than off-package GDDR designs at comparable power.
- **Module-level interfaces:** outside the package, the accelerator provides PCIe host connectivity and NVLink links for GPU-GPU communication, depending on form factor and server integration [NVI25c].

This distinction matters methodologically: when a kernel is memory-bound, the limiting path is often the GA100↔HBM package subsystem; when multi-GPU scaling is analyzed, the dominant path may shift to NVLink; and when host staging dominates, PCIe plus CPU-NUMA placement becomes critical.

A100 40GB vs 80GB: HBM2 vs HBM2e

For this thesis, the key architectural difference between A100-SXM4-40GB and A100-SXM4-80GB is the memory subsystem, not the SM compute front end. The 40GB card uses HBM2, while the 80GB card uses HBM2e with higher aggregate bandwidth and doubled capacity. A concise comparison is given in Table 2.5.

Table 2.5.: A100 SXM4 variant comparison relevant to performance modeling. Data compiled from NVIDIA's A100 architecture whitepaper and product specifications [NVI20a, NVI25c].

Property	A100-SXM4-40GB	A100-SXM4-80GB
Memory technology	HBM2	HBM2e
HBM capacity	40 GB	80 GB
Peak HBM bandwidth	1555 GB/s	2039 GB/s
FP32 peak throughput	19.5 TFLOPS	19.5 TFLOPS
L2 cache size	40 MB	40 MB
NVLink aggregate bandwidth per GPU (bidirectional)	600 GB/s	600 GB/s

Two immediate modeling consequences follow:

1. **Higher memory roofline.** For memory-bound kernels, the best-case throughput scaling from 40GB to 80GB is approximately the bandwidth ratio $2039/1555 \approx 1.31$ (about 31%).

2. **Lower ridge intensity.** With unchanged FP32 peak but higher bandwidth, the FP32 roofline ridge point shifts from

$$I^{\star}_{40} = \frac{19.5}{1555} \approx 12.5 \text{ FLOPs/byte}$$

to

$$I^{\star}_{80} = \frac{19.5}{2039} \approx 9.6 \text{ FLOPs/byte.}$$

This means more kernels fall into the compute-bound region on A100-80GB compared with A100-40GB at the same arithmetic intensity.

A100-80GB mainly increases headroom for memory-bound and capacity-limited workloads. Compute-pipeline ceilings stay almost unchanged, so compute-bound kernels should not see comparable speedups.

Theoretical Peak Performance

Table 2.6.: 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×

Derived Performance Limits

Table 2.7.: 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)	14.40
Total CUDA cores	6912
Total FP64 cores	3456
Maximum resident warps	6912
Maximum resident threads	221 184
Arithmetic intensity threshold FP32 (FLOPs/byte)	12.5
Arithmetic intensity threshold FP64 (FLOPs/byte)	6.2

Memory Latency

Table 2.8.: Approximate memory access latency at different hierarchy levels.

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

2.3.6. Compute Node Topology

The performance of GPU-accelerated applications depends not only on the GPU itself but also on the topology of the compute node—how CPUs, GPUs, and memory are interconnected. This subsection describes the node architecture of the JURECA-DC system used throughout this thesis, which is representative of modern multi-GPU HPC nodes.

CPU and NUMA Topology

Each compute node contains two AMD EPYC 7742 processors (Rome, Zen 2 microarchitecture), each with 64 physical cores. The chips follow a chiplet design. Since the terms are easy to mix up, we fix the naming used in this thesis:

- **CCD** (*Core Complex Die*): a compute chiplet containing CPU cores.
- **CCX** (*Core Complex*): a core cluster inside a CCD. On EPYC 7742, each CCD has two CCXes.
- **cIOD** (*central I/O die*): the central die that provides memory controllers, PCIe root complexes, and fabric routing.

On this Rome generation, each CCX contains four cores with a private 16 MB L3 cache. The hierarchy is shown in Figure 2.8.

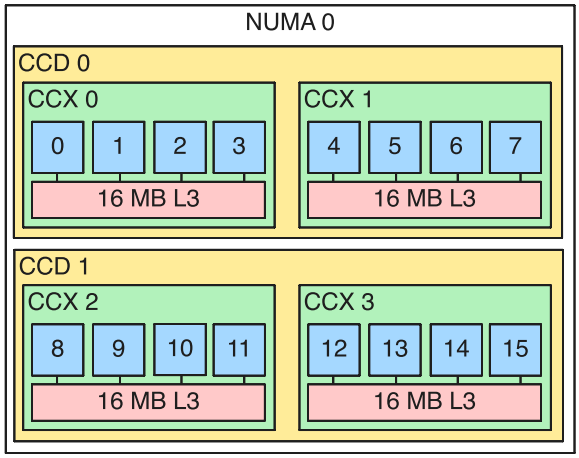

Figure 2.8.: NUMA and chiplet hierarchy of a single AMD EPYC 7742 socket. Each socket contains 8 CCDs grouped into 4 NUMA domains (NPS=4 configuration). Each CCD holds two CCXes, each with 4 cores and a private 16 MB L3 cache. All CCDs connect to a central I/O die that houses the memory controllers and PCIe root complexes.

The BIOS is configured with NPS=4, which exposes four NUMA domains per socket (eight in total across the node). Each NUMA domain encompasses two CCDs (16 cores, 32 threads with SMT) and a quarter of the socket's memory controllers. This configuration allows the operating system and runtime to make NUMA-aware allocation decisions.

Infinity Fabric and latency classes. AMD's Infinity Fabric connects CCX/CCD resources to the cIOD and also connects the two sockets. The practical latency hierarchy for software placement decisions is:

1. **Intra-CCX:** cores sharing the same 16 MB L3 cache communicate through it at low latency.
2. **Cross-CCX / cross-CCD, same socket:** communication no longer benefits from a shared L3 and is routed via Infinity Fabric through cIOD resources, with higher latency.
3. **Cross-socket:** traffic traverses inter-socket Infinity Fabric links (xGMI), which is the highest-latency path.

Why this matters for GPU work. The GPU is not attached directly to a random core; it is attached to a PCIe root complex on the cIOD, and each GPU therefore has a *local* NUMA region on the host side. A host-to-device transfer is fastest when both the launching thread and the source host memory are local to the GPU-affine NUMA domain.

If either thread placement or host-memory placement is remote, the transfer must take additional Infinity Fabric hops before reaching PCIe, which increases latency and can reduce effective bandwidth.

GPU Interconnect Topology

Each node has four NVIDIA A100-SXM4-40GB GPUs arranged in a fully connected mesh via NVLink 3.0. Every GPU pair is connected by four NVLink bridges (denoted NV4 in NVIDIA's topology notation), providing 100 GB/s of uni-directional bandwidth between any two devices. This symmetric, all-to-all connectivity means that multi-GPU tensor contractions do not suffer from topology-dependent bandwidth asymmetries.

PCIe and NVLink in this thesis. Although both are high-speed interconnects, they serve different roles:

- **PCIe Gen4 ×16** is the primary host-device path. On A100 nodes, it carries control traffic and host staging transfers, with a nominal bidirectional bandwidth of about 64 GB/s.

- **NVLink 3.0** is the primary GPU-GPU path. It is used for peer-to-peer exchange and collective communication, with up to 600 GB/s bidirectional bandwidth per A100 SXM4 GPU.

Practical rule: use PCIe-local NUMA placement to minimise host-device staging costs, and rely on NVLink for inter-GPU tensor redistribution whenever possible [NVI25c, NVI24b].

Each GPU is physically attached via PCIe Gen 4 ×16 to a specific NUMA domain on one of the two CPU sockets:

Table 2.9.: GPU-to-NUMA affinity on a JURECA-DC compute node.

GPU	Socket	NUMA domain	CPU cores (physical)
GPU 0	0	3	48–63
GPU 1	0	1	16–31
GPU 2	1	7	112–127
GPU 3	1	5	80–95

The full topology, including the three classes of CPU–GPU data paths, is illustrated in Figure 2.9.

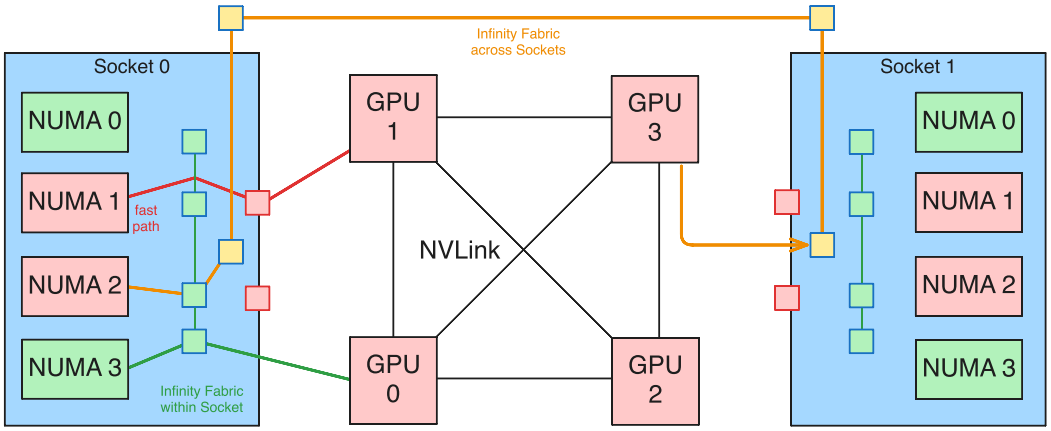

Figure 2.9.: Compute node topology showing the four A100 GPUs (centre) connected via NVLink 3.0, with the two CPU sockets on either side. Three representative host–device paths are highlighted: a fast local PCIe path (GPU 1 $\leftrightarrow$ NUMA 1), an intra-socket Infinity Fabric path (GPU 0 $\leftrightarrow$ NUMA 3), and a cross-socket Infinity Fabric path (GPU 3 $\leftrightarrow$ NUMA 2).

The distinction between these paths matters for host–device data transfers. A `cudaMemcpy` initiated from a CPU thread pinned to a GPU's local NUMA domain takes the shortest PCIe path through the I/O die. If the source thread or memory allocation resides on the wrong NUMA domain—or worse, the wrong socket—the transfer must additionally traverse Infinity Fabric, increasing latency. For GPU-to-GPU communication, NVLink bypasses the CPU subsystem entirely, making the CPU topology irrelevant for peer-to-peer transfers.

Fast-path policy used in this thesis. For runs that include host-device staging, the following policy is applied:

1. pin the orchestration thread to cores in the GPU-affine NUMA domain,
2. allocate pinned host buffers on the same NUMA domain,
3. execute `cudaMemcpyAsync` from that thread/context to preserve the local PCIe path.

This is a standard NUMA-locality rule for PCIe-attached accelerators and is independent of tensor-network physics details. More tightly integrated CPU–GPU systems (e.g. Grace-Hopper style designs) use different host-device topology assumptions and are therefore outside the scope of this node-level model.

Host-device path cost model. For a transfer of size V bytes, we use a first-order latency+bandwidth model in the standard Hockney form [HJ88]:

$$T_{\mathrm{H2D}}(\pi, V) \approx T_0(\pi) + \frac{V}{B_{\mathrm{eff}}(\pi)},$$

where π is the path class (local, same-socket remote, or cross-socket remote), T_0 is the fixed setup/latency term, and B_{eff} is the effective end-to-end bandwidth.

To make the topology penalty explicit, we can write

$$T_0(\pi) = T_0^{\text{local}} + n_{\text{IF}}(\pi)\,\tau_{\text{IF}},$$

with n_{IF} the number of additional Infinity Fabric segments and τ_{IF} an average per-segment latency penalty. In practice, this gives the expected ordering

$$T_{\text{local}} < T_{\text{same socket remote}} < T_{\text{cross socket}}.$$

The model is simple but sufficient for placement policy decisions: thread pinning and host-buffer NUMA placement should minimise n_{IF} and maximise B_{eff}. This is consistent with CUDA placement recommendations for multi-socket hosts [NVI24b, NVI25b].

Concrete mapping for this node. Using the measured affinity in Table 2.9, the path classes can be made explicit. For example, for GPU 1 (local NUMA 1):

- **Local path** ($\pi = \pi_{\text{local}}$): thread and pinned host buffer on NUMA 1 (cores 16–31), modeled with $n_{\text{IF}} = 0$.
- **Same-socket remote path** ($\pi = \pi_{\text{ss}}$): thread/buffer on NUMA 3 (same socket, cores 48–63), modeled with one additional fabric segment ($n_{\text{IF}} \approx 1$).
- **Cross-socket path** ($\pi = \pi_{\text{xs}}$): thread/buffer on NUMA 5 or NUMA 7 (other socket), modeled with inter-socket traversal plus local fabric ($n_{\text{IF}} \approx 2$ in this simplified model).

The same local/same-socket/cross-socket classification applies to GPU 0, GPU 2, and GPU 3 by replacing NUMA IDs with their entries from Table 2.9. The practical expectation is

$$B_{\text{eff}}(\pi_{\text{local}}) > B_{\text{eff}}(\pi_{\text{ss}}) > B_{\text{eff}}(\pi_{\text{xs}}),$$

and correspondingly higher transfer time for the same V.

Cost instantiation for this architecture. For topology-aware staging costs, we model a sequence of N host-device transfers with total volume $V_{\text{tot}} = \sum_{i=1}^{N} V_i$ as

$$T_{\text{stage}}(\pi) = \sum_{i=1}^{N} T_{\text{H2D}}(\pi, V_i) \approx N\,T_0(\pi) + \frac{V_{\text{tot}}}{B_{\text{eff}}(\pi)}.$$

With the path classes used in this node model ($n_{\text{IF}}(\pi_{\text{local}}) = 0$, $n_{\text{IF}}(\pi_{\text{ss}}) = 1$, $n_{\text{IF}}(\pi_{\text{xs}}) = 2$), this becomes

$$\begin{aligned}
T_{\text{stage}}(\pi_{\text{local}}) &\approx N\,T_0^{\text{local}} + \frac{V_{\text{tot}}}{B_{\text{eff}}(\pi_{\text{local}})}, \\
T_{\text{stage}}(\pi_{\text{ss}}) &\approx N\left(T_0^{\text{local}} + \tau_{\text{IF}}\right) + \frac{V_{\text{tot}}}{B_{\text{eff}}(\pi_{\text{ss}})}, \\
T_{\text{stage}}(\pi_{\text{xs}}) &\approx N\left(T_0^{\text{local}} + 2\tau_{\text{IF}}\right) + \frac{V_{\text{tot}}}{B_{\text{eff}}(\pi_{\text{xs}})}.
\end{aligned}$$

Relative to local placement, the predicted penalties are therefore

$$\Delta T_{\mathrm{ss}} = T_{\mathrm{stage}}(\pi_{\mathrm{ss}}) - T_{\mathrm{stage}}(\pi_{\mathrm{local}}) \approx N\,\tau_{\mathrm{IF}} + V_{\mathrm{tot}}\left(\frac{1}{B_{\mathrm{eff}}(\pi_{\mathrm{ss}})} - \frac{1}{B_{\mathrm{eff}}(\pi_{\mathrm{local}})}\right),$$
$$\Delta T_{\mathrm{xs}} = T_{\mathrm{stage}}(\pi_{\mathrm{xs}}) - T_{\mathrm{stage}}(\pi_{\mathrm{local}}) \approx 2N\,\tau_{\mathrm{IF}} + V_{\mathrm{tot}}\left(\frac{1}{B_{\mathrm{eff}}(\pi_{\mathrm{xs}})} - \frac{1}{B_{\mathrm{eff}}(\pi_{\mathrm{local}})}\right).$$

This turns placement into a computable policy decision: for each staging phase, evaluate ΔT_{ss} and ΔT_{xs} from measured T_0/B_{eff} and choose local NUMA placement whenever possible.

How parameters are obtained in practice. For each path class π, measure transfer times for multiple sizes $V \in [V_{\min}, V_{\max}]$ and fit

$$T(V) = a_\pi + b_\pi V, \qquad T_0(\pi) = a_\pi, \quad B_{\mathrm{eff}}(\pi) = \frac{1}{b_\pi}.$$

This linear fit is the standard way to instantiate latency+bandwidth models in system-performance analysis [HJ88, Jai91]. Once fitted, the above expressions directly provide predicted staging overhead for any (N, V_{tot}) in the workload.

Per-GPU path classes used in this thesis. The local and same-socket "remote" NUMA domains used for orchestration are:

GPU	π_{local}	π_{ss} (used)	π_{xs} (used)
GPU 0	NUMA 3	NUMA 1	NUMA 5 or NUMA 7
GPU 1	NUMA 1	NUMA 3	NUMA 5 or NUMA 7
GPU 2	NUMA 7	NUMA 5	NUMA 1 or NUMA 3
GPU 3	NUMA 5	NUMA 7	NUMA 1 or NUMA 3

Table 2.10.: Path-class mapping used for topology-aware cost evaluation.

For each row in Table 2.10, the same $n_{\mathrm{IF}} = \{0, 1, 2\}$ model applies, so the transfer-cost expressions above are directly reusable across all four GPUs.

In practice, this topology has two implications for the work in this thesis:

- **Single-GPU kernels:** the CPU topology is largely invisible, since kernel launch overhead is small and tensor data resides in GPU memory throughout the computation.
- **Multi-GPU contractions:** data distribution and inter-GPU communication use NVLink exclusively. Host-side orchestration threads are pinned to the appropriate NUMA domain to minimise any residual host–device transfer overhead.

2.4. 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.

2.4.1. 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`.

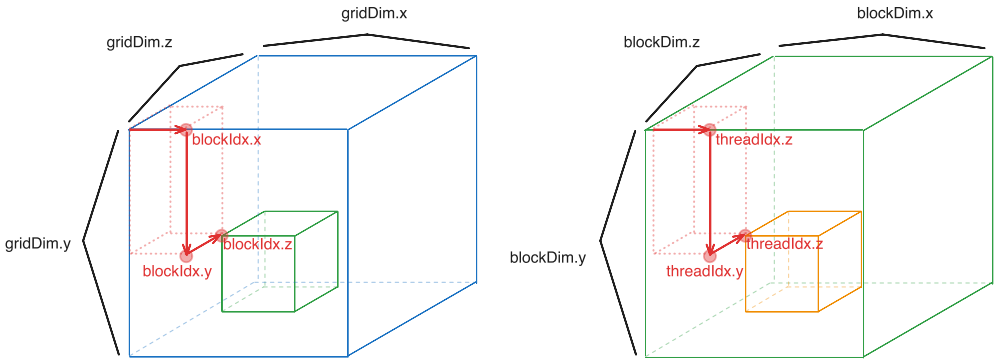

Figure 2.10.: CUDA execution hierarchy showing the mapping of grids to thread blocks and threads. Threads are organised in up to three dimensions and identified using the built-in variables `threadIdx` and `blockIdx`.

The hierarchical organisation of grids, thread blocks, and threads is illustrated in Figure 2.10. 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:

```
__global__ void vecAdd(const float *a, const float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

// Host launch
int n = 1 << 20;                    // 1,048,576 elements
int threadsPerBlock = 256;
```

```
int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
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 $\times$ 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. For a 3D tensor with logical indices (x, y, z) and extents (s_x, s_y, s_z), a common row-major linearisation is

$$\text{offset}(x, y, z) = x + s_x\, y + s_x s_y\, z. \tag{2.3}$$

Equivalently, using row-major index names (i, j, k) for a shape (D_0, D_1, D_2):

$$\text{offset}(i, j, k) = ((iD_1) + j)D_2 + k.$$

In compact form for rank-d tensors with extents $(D_0, \ldots, D_{d-1})$,

$$\text{offset}(i_0, \ldots, i_{d-1}) = \sum_{m=0}^{d-1} i_m \prod_{n=m+1}^{d-1} D_n.$$

In column-major (Fortran) order, the convention used by cuBLAS and most BLAS libraries, the 2D index is instead

$$\text{offset} = j \cdot M + i. \tag{2.4}$$

For the same 3D extents (s_x, s_y, s_z) and indices (x, y, z), the column-major form is

$$\text{offset}(x, y, z) = z + s_z\, y + s_z s_y\, x.$$

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 2.4.3.

2.4.2. 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 2.8), 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 2.7).

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.

```
#define TILE 16

__global__ void matmul(const float *A, const float *B, float *C,
                       int M, int N, int K) {
    __shared__ float As[TILE][TILE];
    __shared__ float Bs[TILE][TILE];

    int row = blockIdx.y * TILE + threadIdx.y;
    int col = blockIdx.x * TILE + threadIdx.x;
    float sum = 0.0f;

    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 kernel structure has three important properties:

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 $16\times$ 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.

2.4.3. 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 mod 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.

2.4.4. 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.

2.4.5. Kernel Launch Mechanics

The triple-chevron syntax `<<<gridDim, blockDim>>>` conceals a multi-stage pipeline that spans both host and device [NVI24b]. Understanding this pipeline is essential for reasoning about the cost of launching many small kernels, which is the regime encountered in tensor network contractions.

When the host executes a kernel launch, the following sequence of operations takes place:

1. **Argument marshalling.** The kernel's parameters—device pointers, scalars, dimensions—are packed into a parameter buffer and validated against the current CUDA context.
2. **Command enqueueing.** The launch command, containing the kernel function pointer, grid and block dimensions, dynamic shared memory size, and the parameter buffer, is placed into the stream's command queue. The host-side API call returns immediately; the launch is *asynchronous* from the host's perspective.

3. **Driver serialisation.** The CUDA driver serialises the queued command into the GPU's hardware command buffer.

4. **Block distribution.** The GPU's global block scheduler (referred to as the *GigaThread engine* in NVIDIA's architecture whitepapers [NVI20a]) dequeues the launch and begins distributing thread blocks to SMs. For each block, the scheduler verifies that the target SM has sufficient free resources—registers, shared memory, warp slots, and block slots—before making the assignment. Once assigned, a block remains on its SM until all of its warps complete execution; blocks do not migrate between SMs [NVI24b].

5. **Warp execution.** Within the assigned SM, the block's threads are partitioned into warps and handed to the processing block's warp scheduler, which begins issuing instructions as described in Section 2.3.1.

Steps 1–4 incur a fixed cost that is largely independent of the kernel's computational workload. On the A100, this cost is typically in the range of 5–10 μs per launch. For large kernels that execute for milliseconds, the overhead is negligible. For small kernels, however, the launch cost can exceed the computation time by an order of magnitude or more. This has a direct implication for workloads that require many small operations, such as batches of low-dimensional GEMM calls arising from tensor contractions: launching each operation as a separate kernel wastes the majority of the wall-clock time on launch overhead rather than useful computation.

MemcpyAsync vs Memcpy in Host–Device Timelines

Figures 2.11 and 2.12 illustrate two common execution patterns. The asynchronous version enqueues transfers and kernels and lets the host continue with CPU work until an explicit synchronisation point. The synchronous version uses blocking `cudaMemcpy`, which stalls the host thread during each transfer.

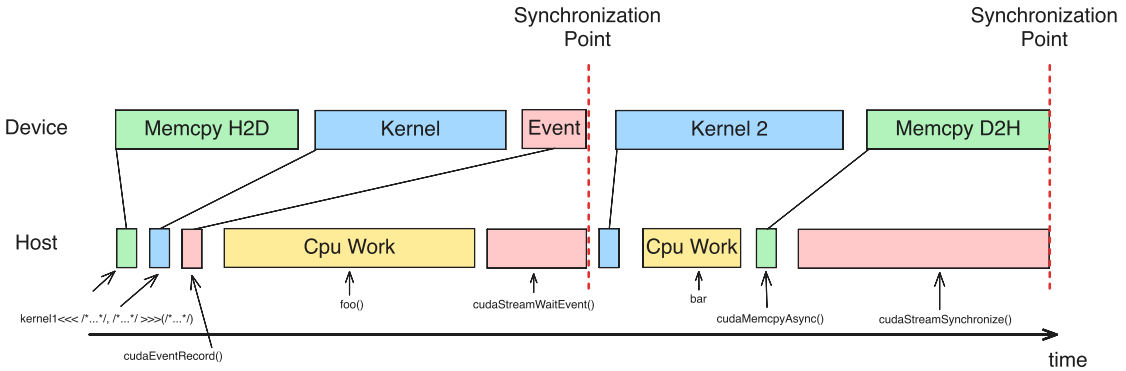

Figure 2.11.: Asynchronous host–device timeline using `cudaMemcpyAsync`, streams, and event-based ordering. Host work overlaps queued device work until explicit synchronisation.

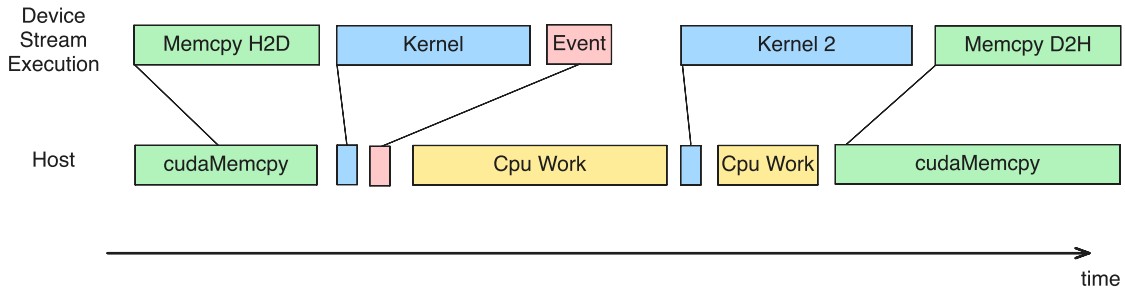

Figure 2.12.: Synchronous host–device timeline using blocking `cudaMemcpy`. The host is stalled during H2D and D2H transfers.

The corresponding asynchronous CUDA pattern is shown below. For true asynchronous host-device copies, host buffers should be page-locked (pinned), e.g. via `cudaMallocHost` [NVI24b, NVI25b].

```
cudaStream_t s1, s2;
cudaEvent_t done1;
cudaStreamCreate(&s1);
cudaStreamCreate(&s2);
cudaEventCreate(&done1);

// h_in / h_out should be pinned for true async copies.
cudaMemcpyAsync(d_in, h_in, bytes, cudaMemcpyHostToDevice, s1);
kernel1<<<grid1, block1, 0, s1>>>(d_in, d_tmp);
cudaEventRecord(done1, s1);

foo();  // host CPU work overlaps queued device work

cudaStreamWaitEvent(s2, done1, 0);
kernel2<<<grid2, block2, 0, s2>>>(d_tmp, d_out);

bar();  // more host CPU work

cudaMemcpyAsync(h_out, d_out, bytes, cudaMemcpyDeviceToHost, s2);
cudaStreamSynchronize(s2);  // explicit synchronization point
```

Listing 2.9: Asynchronous pattern with `cudaMemcpyAsync`, stream ordering, and explicit final synchronisation.

Practical interpretation. The asynchronous pattern is usually preferred when host-side orchestration can be overlapped with device execution. The synchronous pattern is simpler and often acceptable for single-shot transfers, but it introduces avoidable host idle time in pipelines with many transfer/compute stages.

Kernel Fusion

The standard mitigation for launch overhead is *kernel fusion*: combining multiple logically independent operations into a single kernel, so that the launch cost is paid once rather than once per operation. In the context of this thesis, fusion is achieved using cuBLASDx [NVI25a], which provides a device-side GEMM API that can be called from within a user-written kernel. This allows multiple GEMM operations to be executed sequentially (or cooperatively) inside a single kernel invocation, eliminating the

per-operation launch cost entirely. The cuBLASDx library is introduced in Section 2.5.1, and the performance impact of fusion is quantified experimentally in Section 5.2.

2.5. Related Work

2.5.1. cuBLAS, cuBLASDx, and cuTENSOR

NVIDIA's vendor libraries define the practical baseline for dense kernels on Ampere GPUs. cuBLAS provides highly optimised BLAS routines and remains the reference point for GEMM-like workloads [NVI24a]. cuTENSOR generalises this to tensor contractions and permutation-heavy primitives [NVI24c].

For the small-kernel regime relevant to tensor-network workflows, cuBLASDx is especially important: it enables device-side GEMM composition and supports launch-amortisation strategies that are difficult to implement with host-only library calls [NVI25a]. The launch-overhead benchmark in this thesis (Section 5.2) uses this capability directly.

In this thesis, custom kernels are evaluated against these libraries as performance baselines rather than replacements. The objective is to understand where workload-specific structure can beat general-purpose interfaces.

2.5.2. ChASE Eigensolver

The ChASE line of work provides relevant experience in GPU-oriented numerical software engineering, especially for hybrid CPU–GPU execution and scalable performance tuning [WSDN19, WDADN22, WDN23]. Although ChASE targets eigenvalue problems rather than tensor networks, it shares core HPC concerns with this thesis:

- balancing algorithmic structure with hardware-aware implementation,
- minimising communication and orchestration overhead,
- validating performance claims with rigorous benchmarking.

These principles influence the methodological choices in later chapters, particularly the emphasis on bottleneck-driven optimisation.

2.5.3. Existing GPU Tensor Network Implementations

Prior work on tensor contractions shows that performance is sensitive to index ordering, data layout, and mapping of contractions to BLAS kernels ([SSK17] provides an early and still useful reference point). Across implementations, two recurring observations are relevant here:

1. **Kernel granularity matters.** Many contractions are too small to fully utilise GPU throughput when issued individually.
2. **Data movement dominates.** Layout transforms and non-coalesced accesses can erase gains from nominally fast compute kernels.

This thesis builds on those observations with a stricter focus on Ampere-era execution details: launch overhead, occupancy limits, memory hierarchy effects, and profile-guided kernel fusion on realistic node topology.

2.5.4. Selected Work in Collaborating Research Directions

The following review focuses on work that is directly relevant to this thesis context: Wu, Di Napoli, Rizzi, Waintal, Legeza, and close collaborators.

Wu and Di Napoli: GPU-Distributed Dense Kernels and Communication

Wu et al. report a distributed hybrid CPU–GPU eigensolver where the dominant kernel is a distributed Hermitian matrix–matrix multiplication step (`HEMM`), and show that replacing vendor `PZGEMM` with a custom GPU-distributed implementation significantly improves strong scaling on JUWELS-Booster [WDADN22]. The SC-W extension further improves distributed behaviour with NCCL-based communication optimisations [WDN23].

A practical architectural lesson is clear: once local kernels are efficient, communication orchestration and overlap become major bottlenecks.

Legeza Group: Massively Parallel DMRG/MPS on CPU–GPU Systems

Menczer et al. present a sequence of works on massively parallel DMRG-style workloads that are highly relevant for kernel engineering: lock-free scheduling, task buffering, explicit overlap of communication and computation, and mixed-precision/tensor-core acceleration paths ([ML+23b, ML+23a, ML+24]).

The 2023 and 2024 studies also report good scaling across heterogeneous CPU–GPU systems and explicitly motivate mixed-precision/tensor-core execution for contraction-heavy workloads [ML+23b, ML+24]. For this thesis, the direct transfer is straightforward: reduce launch count for small contractions, bucket by shape, and tune stream-level scheduling.

Rizzi and Waintal: Operator-Sum Formulations and Practical TN Performance

Krinitsin, Rizzi, and Waintal present a TTN time-evolution implementation with two-level parallelisation (shared and distributed memory) and report order of magnitude speedups from GPU acceleration in the targeted workload regime [KRW25]. A practical point for this thesis is their emphasis on local operator-sum formulations, which can avoid expensive explicit intermediate constructions in contraction pipelines.

Waintal and collaborators provide broader algorithmic limits and scalable simulation perspectives for circuit-style tensor-network workloads ([ZSW20, A+23]). Even where papers are not CUDA-kernel papers, the computational patterns they identify (many medium/small contractions, truncation-heavy iterative sweeps) match the exact regime where launch overhead, layout efficiency, and batching matter most on A100-class hardware.

A100 CUDA Optimisation Practice (Documentation SOTA)

For low-level CUDA engineering on Ampere, the most useful guidance comes from NVIDIA's architecture and profiling documentation rather than from application papers. The recurring recommendations are:

1. treat memory traffic and reuse as first-order constraints (coalescing, shared-memory tiling, and cache-aware layouts),

2. use occupancy as a constraint, not a sole objective; register and shared-memory pressure must be balanced against instruction-level efficiency,

3. reduce launch overhead for short kernels through batching, fusion, and stream-graph style execution where appropriate,

4. validate each optimisation with metric-level profiling instead of relying on throughput numbers alone.

These points are consistent across the Ampere tuning guide, CUDA best-practice guide, and Nsight Compute profiling guide ([NVI25d, NVI25b, NVI25e]).

Integration plan used in this thesis. Given the current project scope (single A100 node, contraction-heavy short kernel sequences), the most relevant near-term integration steps are:

1. build a shape-bucketed benchmark suite for representative contraction sizes,

2. maintain a per-kernel profiler checklist (occupancy, memory throughput, stall breakdown, launch cost),

3. prioritise launch amortisation before aggressive micro-optimisation for kernels that run in the sub-millisecond regime,

4. keep FP32 as baseline and treat TF32 as an opt-in acceleration path with explicit numerical checks.

Recurring Implementation Patterns

Across the above works, four recurring implementation patterns emerge:

1. **Task bucketing by contraction shape.** Grouping same-shape small contractions enables batched GEMM and reduces launch count.

2. **Operator-sum execution paths.** Avoiding explicit large operator objects can reduce memory traffic and intermediate-kernel overhead.

3. **Communication–computation overlap.** Multi-GPU execution benefits from explicit stream partitioning and asynchronous collectives.

4. **Mixed-precision acceleration with guarded validation.** FP32/TF32 tensor-core execution is used for throughput, with strict numerical checks.

Paper Notes for Implementation Planning (Working Draft)

The table below is kept as implementation-oriented notes. It maps paper-level results and methodology to concrete optimisation actions for this thesis.

Table 2.11.: Working notes: transfer from selected literature to CUDA optimisation actions in this thesis.

Paper (focus)	Reported methodology/result signal	Direct action for this thesis
[ML+23b] (massively parallel TN state algorithms on CPU–GPU systems)	explicit multi-level parallelisation and mixed CPU/GPU execution for contraction-heavy workloads	keep kernels bucketed by shape and preserve stream-level concurrency; avoid one-kernel-per-contraction execution for small blocks
[ML+23a] (symmetry-aware massively parallel TN implementation)	algorithmic structure strongly influences effective performance, not just raw kernel throughput	separate profiling by contraction class and layout class; avoid aggregating all kernels into one average metric
[ML+24] (DMRG implementation/tuning with tensor-core usage)	strong emphasis on mixed precision and throughput-oriented kernel organisation	keep FP32 baseline, treat TF32 as opt-in path with fixed numerical validation; prioritise high-call-count kernels first
[KRW25] (TTN time evolution, Rizzi/Waintal collaboration)	two-level parallelisation and GPU acceleration provide major practical gains in targeted regimes	retain local-operator-sum style execution when possible and minimise explicit large intermediates that create extra memory traffic
[WDADN22, WDN23] (GPU-distributed dense eigensolver lineage at JSC)	once local kernels are efficient, communication/orchestration limits dominate scaling	for one-node scope, prepare kernels and APIs so later overlap of local compute and communication is possible without redesign

2.5.5. Representative CUDA Implementation Patterns

The following snippets are representative implementation skeletons derived from the above literature patterns. They are intentionally simplified to expose the kernel-architecture idea.

Pattern A: Bucketed Small-GEMM Execution (DMRG/MPS Sweeps)

```
struct BatchBucket {
    int m, n, k, count;
    const float* A;   // strided batch base
    const float* B;
    float* C;
    long long strideA, strideB, strideC;
};

void run_bucket(const BatchBucket& b, cublasHandle_t h, cudaStream_t s) {
    cublasSetStream(h, s);
```

```
    const float alpha = 1.0f, beta = 0.0f;
    cublasGemmStridedBatchedEx(
        h, CUBLAS_OP_N, CUBLAS_OP_N,
        b.m, b.n, b.k,
        &alpha,
        b.A, CUDA_R_32F, b.m, b.strideA,
        b.B, CUDA_R_32F, b.k, b.strideB,
        &beta,
        b.C, CUDA_R_32F, b.m, b.strideC,
        b.count,
        CUBLAS_COMPUTE_32F_FAST_TF32, CUBLAS_GEMM_DEFAULT_TENSOR_OP);
}
```

Listing 2.10: Shape-bucketed small contractions using strided batched GEMM. Representative kernel-orchestration pattern inspired by massively parallel DMRG implementations.

Why it matters. Instead of launching one kernel per tiny contraction, this pattern executes many contractions per call and shifts the regime away from launch-dominated runtime.

Pattern B: Operator-Sum Path Without Explicit MPO Materialisation

```
// Pseudocode-level sketch: each term is mapped to a GEMM-like contraction.
for (int t = 0; t < num_terms; ++t) {
    // left[t], right[t], and local_op[t] encode one local contribution
    contract_term_kernel<<<grid, block, shmem, stream>>>(
        left[t], local_op[t], right[t], state_in, state_tmp);
}

// Optional: fuse accumulation to reduce global-memory traffic.
accumulate_terms_kernel<<<grid2, block2, 0, stream>>>(state_tmp, state_out);
```

Listing 2.11: Operator-sum contraction sketch. The core idea is to accumulate local terms directly instead of materialising a large explicit operator tensor first.

Why it matters. It follows the local-operator-sum idea discussed in recent TTN benchmarking: intermediate tensor construction is reduced, and cache/memory-traffic behavior often improves [KRW25].

Pattern C: Overlap of Local Compute and Multi-GPU Reduction

```
cudaStream_t s_compute, s_comm;
cudaStreamCreate(&s_compute);
cudaStreamCreate(&s_comm);

// 1) Local block update on each GPU
local_update_kernel<<<grid, block, 0, s_compute>>>(local_A, local_B, local_C);

// 2) Asynchronous inter-GPU reduction / exchange
ncclAllReduce(local_C, global_C, count, ncclFloat, ncclSum, comm, s_comm);

// 3) Synchronize only when data dependency requires it
cudaEvent_t done_comm;
cudaEventCreate(&done_comm);
cudaEventRecord(done_comm, s_comm);
```

```
cudaStreamWaitEvent(s_compute, done_comm, 0);
postprocess_kernel<<<grid2, block2, 0, s_compute>>>(global_C);
```

Listing 2.12: Two-stream overlap pattern for distributed dense kernels. The communication stream runs asynchronous collectives while compute proceeds on the main stream.

Why it matters. It mirrors the communication/computation orchestration emphasis in Wu–Di Napoli style distributed GPU kernels and in massively parallel DMRG implementations [WDADN22, WDN23, ML+23b].

3. Design and Methodology

Chapter 3 defines the optimisation target and methodology before the implementation details. It specifies which kernel patterns are in scope, which baselines are used, and how data-layout decisions are tied to performance hypotheses.

3.1. Target Kernels

The target workload is defined by contraction patterns that can be reduced to dense linear algebra kernels after index permutation and reshaping. We focus on kernel classes that are both frequent in tensor-network workflows and sensitive to GPU execution overhead:

1. **Small/medium GEMM-like contractions** (single and batched), including cases where dimensions are not multiples of warp tile sizes.
2. **Layout-transformation kernels** (permute/pack/unpack) required before and after contractions.
3. **Fused kernel sequences** that execute several contractions (and optional lightweight post-processing) inside one launch.

The baseline precision target is FP32 for compute and storage, with optional TF32 tensor-core execution where numerical tolerance permits it. FP64 remains a reference configuration for accuracy and performance comparison.

3.1.1. Selection Criteria

A kernel is included in the optimisation set if it satisfies at least two of the following conditions:

- high call frequency in representative traces,
- measurable contribution to end-to-end runtime,
- clear exposure to launch overhead or memory-traffic bottlenecks,
- portability to both single-GPU and one-node multi-GPU execution.

3.1.2. SOTA-Informed Initial Backlog (Working Notes)

Before full application traces are fixed, the optimisation backlog is organised as a set of generic kernel classes that repeatedly appear in the cited literature and in early profiling:

1. **Shape-bucketed small GEMM path.** Group contractions by (m, n, k) class and run batched/device-side execution to reduce launch count.

2. **Layout-heavy micro-pipeline path.** Fuse reshape/pack/contract segments when data dependencies allow it, and track whether reduced launch count offsets higher register/shared-memory pressure.

3. **Local-operator-sum contraction path.** Prefer execution plans that avoid explicit large intermediate operators when equivalent local-term accumulation is available.

For each class, the first profiling pass records the same minimum metric set: kernel runtime, launch count, achieved occupancy, DRAM throughput, and top warp stall reasons. This keeps early optimisation choices comparable even before the final production kernel list is fixed.

3.2. Case Study: Contraction Order in a Two-Site TN Update

A concrete tensor-network-style case shows how a reasonable first implementation can be much slower than a mathematically equivalent alternative.

3.2.1. Problem Definition

Consider a batched two-site update with fused physical dimension $d_p = d^2$:

$$Y_{b,p,r} = \sum_{q=0}^{d_p-1} \sum_{c=0}^{\chi-1} G_{p,q}\, X_{b,q,c}\, M_{c,r},$$

where:

- $b \in [0, B)$ is a batch index,
- $p, q \in [0, d_p)$ are fused physical indices,
- $c, r \in [0, \chi)$ are bond-space indices.

The tensor structure is representative of two-site gate application and local environment projection steps in MPS-style workflows.

3.2.2. How to Read This as a Practical HPC Task

The expression can be read as a matrix-chain computation repeated over a batch:

- for each fixed batch index b, interpret $X_{b,:,:}$ as a matrix $X_b \in \mathbb{R}^{d_p \times \chi}$,
- interpret G as $d_p \times d_p$ and M as $\chi \times \chi$,
- compute

$$Y_b = (G\, X_b)\, M, \qquad b = 0, \ldots, B-1.$$

From an implementation point of view, the core is a repeated dense linear-algebra pattern where contraction order decides both arithmetic work and memory traffic.

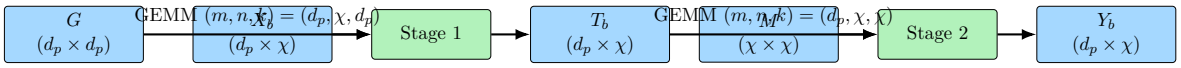

Figure 3.1.: Per-batch matrix view of the two-site update. The tensor expression is equivalent to a two-stage GEMM chain with a shared G and M across batches.

3.2.3. Plausible Baseline (Bad)

A straightforward GPU implementation computes one output element $Y_{b,p,r}$ per thread and evaluates both sums directly. This gives a direct cost

$$\mathcal{O}\Big(B\, d_p^2\, \chi^2\Big)\,,$$

with weak reuse of X and M across threads and heavy global-memory traffic. This baseline is implemented in `code/profiling/tn_two_site_bad_direct.cu`.

3.2.4. Improved Algorithm (Better)

The key change is contraction order:

$$T_{b,p,c} = \sum_{q=0}^{d_p-1} G_{p,q} X_{b,q,c}, \qquad Y_{b,p,r} = \sum_{c=0}^{\chi-1} T_{b,p,c} M_{c,r}.$$

This reduces asymptotic work to

$$\mathcal{O}\Big(B\,(d_p^2\chi + d_p\chi^2)\Big)\,,$$

and maps directly to two strided-batched GEMMs. The implementation is `code/profiling/tn_two_site_good_batched_gemm.cu`.

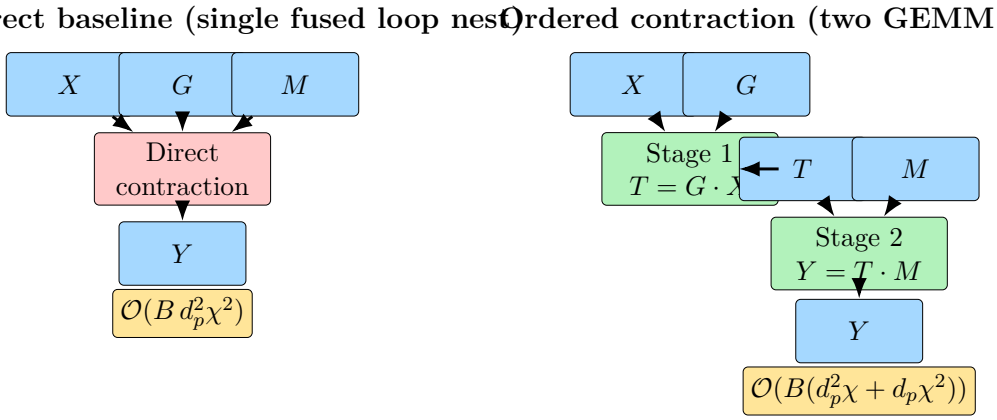

Figure 3.2.: Algorithmic structure of the two-site case study: direct contraction (left) versus contraction-order decomposition into two GEMM-like stages (right).

Theoretical operation-ratio improvement (direct vs ordered) is

$$R = \frac{d_p^2\chi^2}{d_p^2\chi + d_p\chi^2} = \frac{d_p\chi}{d_p + \chi}.$$

For the benchmarked shape $d_p = 16$, $\chi = 256$, this gives $R \approx 15.1$, before any hardware-level optimisations.

3.2.5. Worked Numerical Example (Shape Used in This Thesis)

Using the benchmarked shape $B = 1$, $d_p = 16$, $\chi = 256$:

- output size is $d_p\chi = 4096$ elements,
- direct variant evaluates $d_p\chi = 4096$ inner terms per output element,
- total direct term evaluations are

$$N_{\text{direct}} = d_p^2\chi^2 = 16{,}777{,}216.$$

For ordered contraction:

$$N_{\text{stage1}} = d_p^2\chi = 65{,}536, \qquad N_{\text{stage2}} = d_p\chi^2 = 1{,}048{,}576,$$

so

$$N_{\text{ordered}} = N_{\text{stage1}} + N_{\text{stage2}} = 1{,}114{,}112.$$

This gives

$$\frac{N_{\text{direct}}}{N_{\text{ordered}}} \approx 15.06,$$

which matches the asymptotic ratio above.

Variant	Term evaluations	Relative to ordered
Direct contraction	16,777,216	15.06×
Ordered contraction (two-stage)	1,114,112	1.00×

Table 3.1.: Work comparison for the thesis reference shape ($B = 1$, $d_p = 16$, $\chi = 256$).

3.2.6. Practice Problems (Implementation-Oriented)

1. **Work counting.** For $B = 4$, $d_p = 32$, $\chi = 64$, compute N_{direct} and N_{ordered}.
2. **GEMM mapping.** Write the two GEMMs in (m, n, k) form for stage 1 and stage 2, and identify which operands are shared across the batch.
3. **Shape intuition.** Using $R = \frac{d_p\chi}{d_p+\chi}$, determine whether increasing d_p or increasing χ helps more when $\chi \gg d_p$.
4. **Memory layout choice.** Pick a layout where the hottest contracted index is unit stride in both stages, and state one layout transform needed between stage 1 and stage 2.

Short answer key.

1. $N_{\text{direct}} = 16{,}777{,}216$, $N_{\text{ordered}} = 786{,}432$ ($\approx 21.3\times$ reduction).
2. Stage 1: $(m, n, k) = (d_p, \chi, d_p)$, Stage 2: $(m, n, k) = (d_p, \chi, \chi)$; G and M are batch-shared.
3. For $\chi \gg d_p$, ratio $R \approx d_p$, so increasing d_p gives the stronger marginal gain.
4. Any valid answer should keep the reduction index contiguous in the innermost access pattern of each stage.

3.2.7. Diagram Blueprint for Manual Drawing

To explain this section in one slide or one page, a compact three-panel diagram works well:

1. **Panel A: expression view.** Write $Y_{b,p,r} = \sum_{q,c} G_{p,q} X_{b,q,c} M_{c,r}$ and highlight contracted indices (q, c) in one color.
2. **Panel B: direct implementation.** Draw one big "direct contraction" block with arrows from G, X, and M to Y; annotate $\mathcal{O}(B d_p^2 \chi^2)$.
3. **Panel C: ordered implementation.** Draw two blocks: $T = G \cdot X$ then $Y = T \cdot M$; annotate $\mathcal{O}(B(d_p^2 \chi + d_p \chi^2))$ and the ratio $\frac{d_p \chi}{d_p + \chi}$.

For readability in Excalidraw, keep only five tensor nodes (G, X, M, T, Y), color contracted indices consistently, and place the work numbers directly below each panel.

3.2.8. Design Decisions Behind the Better Variant

The implementation sequence follows the staged optimisation style from Simon Böhm's SGEMM worklog [Boe22]: start with algorithmic structure, then improve memory behavior, then improve execution mapping.

Design decision	Why it helps
Change contraction order first	Removes redundant work ($\mathcal{O}(B d_p^2 \chi^2) \rightarrow \mathcal{O}(B(d_p^2 \chi + d_p \chi^2))$).
Map to strided-batched GEMM	Uses highly tuned kernels instead of manual scalar loops.
Reuse shared operands with zero stride	G and M are reused across the whole batch without host-side relaunch loops.
Keep FP32 pedantic math mode	Preserves strict FP32 behavior while comparing algorithmic variants.
Profile with fixed KPI pack	Separates true algorithm gains from incidental launch/measurement noise.

Table 3.2.: Main design decisions for the two-site contraction case study.

3.2.9. Optimisation Ladder (Structured After SGEMM Worklogs)

Beyond the single "bad vs good" comparison, this case follows a staged optimisation structure similar to the SGEMM worklog style in [Boe22]: fix algorithmic work first, then remove memory pathologies, then improve scheduling and occupancy, and finally tune.

1. **Stage 0 (correct baseline).** Start from a slow but clearly correct direct contraction to define a reproducible reference.
2. **Stage 1 (contraction order).** Remove redundant arithmetic in naive loop nests first; this is often the largest single gain for TN updates.
3. **Stage 2 (global-memory access).** Fix non-coalesced and heavily strided operand traversal so traffic scales with useful work.
4. **Stage 3 (on-chip reuse).** Avoid re-reading hot operands from HBM by increasing L2/shared-memory reuse where possible.
5. **Stage 4 (scheduler feed / ILP).** Increase independent work per warp to improve eligible-warps and issue efficiency.
6. **Stage 5 (resource balance).** Control register growth so occupancy does not collapse under aggressive unrolling/blocking.
7. **Stage 6 (launch cost).** Fuse tiny pre/post kernels around the contraction to amortise host launch overhead.
8. **Stage 7 (parameter search).** Tune tile/shape mappings per regime, since good choices are hardware- and size-dependent.

3.2.10. Profiling Reproducibility

The complete compile-and-profile job is provided as `code/profiling/ncu_tn_contraction_profile.slurm`. It collects the same KPI pack used in the rest of this thesis and writes separate CSV files for both variants.

3.3. Algorithmic Approach

The optimisation strategy follows a profile-driven iterative loop:

1. establish a reproducible baseline with library implementations,
2. measure kernel-level behaviour with Nsight metrics,
3. construct a bottleneck hypothesis (launch-bound, bandwidth-bound, or compute-bound),
4. implement one targeted optimisation,
5. re-profile and accept/reject the change based on quantitative criteria.

This keeps low-level tuning tied to measured bottlenecks instead of guesswork.

3.3.1. Optimisation Axes

Four axes are prioritised throughout the implementation chapters:

- **Launch amortisation:** reduce host-side orchestration overhead through kernel fusion and device-side composition.
- **Memory-efficiency optimisation:** maximise coalesced access and data reuse via layout control, shared-memory tiling, and register blocking.
- **Occupancy-resource balance:** tune block sizes and launch-bounds against register and shared-memory budgets.
- **Precision-performance trade-offs:** evaluate FP64, FP32, and TF32 modes under fixed correctness checks.

3.3.2. Acceptance Criteria

An optimisation is considered successful only if:

- it improves runtime or throughput on the target workload set,
- the profiling counters match the intended mechanism (not incidental noise),
- numerical deviation remains within predefined tolerance,
- implementation complexity remains maintainable for further tuning.

3.3.3. Analytical Performance Models

Before implementation-level tuning, this thesis uses two compact analytical models to classify bottlenecks and to set realistic speedup expectations: roofline analysis for single-kernel limits and Amdahl's law for global parallel-scaling limits.

Roofline Model

Let

$$I = \frac{\text{FLOPs}}{\text{bytes moved to/from main memory}}$$

denote operational intensity (FLOP/byte). For a device with peak compute throughput $P_{\max}$ and peak memory bandwidth B, the roofline bound is

$$P(I) = \min\left(P_{\max},\ B \cdot I\right).$$

This separates the two regimes:

- **memory-bound:** $I < I^{\star}$ and $P(I) \approx B \cdot I$,
- **compute-bound:** $I \geq I^{\star}$ and $P(I) \approx P_{\max}$,

where the crossover intensity is

$$I^\star = \frac{P_{\max}}{B}.$$

Using the A100-SXM4-40GB values from Tables 2.4 and 2.6, we take $B = 1555\,\mathrm{GB/s} = 1.555\,\mathrm{TB/s}$. The resulting crossover points are:

$$I^\star_{\mathrm{FP64}} \approx \frac{9.7}{1.555} = 6.2, \quad I^\star_{\mathrm{FP32}} \approx \frac{19.5}{1.555} = 12.5, \quad I^\star_{\mathrm{TF32}} \approx \frac{156}{1.555} = 100.3.$$

To quantify hardware sensitivity, two additional roofline references are used. For A100-SXM4-80GB, the compute ceilings stay unchanged while bandwidth rises to $B = 2.039\,\mathrm{TB/s}$ [NVI20a], which shifts the crossover points to

$$I^\star_{\mathrm{FP64}} \approx 4.8, \quad I^\star_{\mathrm{FP32}} \approx 9.6, \quad I^\star_{\mathrm{TF32}} \approx 76.5.$$

For H100-SXM5-80GB, using representative peak values $B = 3.35\,\mathrm{TB/s}$, $P_{\mathrm{FP64}} = 33.5\,\mathrm{TFLOPS}$, $P_{\mathrm{FP32}} = 67.0\,\mathrm{TFLOPS}$, and $P_{\mathrm{TF32}} = 494.0\,\mathrm{TFLOPS}$ [NVI22], the crossover points become

$$I^\star_{\mathrm{FP64}} \approx 10.0, \quad I^\star_{\mathrm{FP32}} \approx 20.0, \quad I^\star_{\mathrm{TF32}} \approx 147.5.$$

Small and medium contractions with low reuse remain memory-bound unless layout/tiling increases operational intensity.

Roofline in Practice

For this project, roofline is used as a measurement workflow, not just an analytical plot. The practical loop is:

1. estimate FLOPs and bytes from kernel geometry (static model),
2. collect measured FLOPs, memory traffic, and runtime from Nsight Compute [NVI25e],
3. place each kernel on the roofline and compare against the bound,
4. choose the next optimisation based on the dominant gap.

With measured FLOPs F, measured runtime t, and measured DRAM traffic Q_{dram}, we use

$$I_{\mathrm{meas}} = \frac{F}{Q_{\mathrm{dram}}}, \qquad P_{\mathrm{meas}} = \frac{F}{t}, \qquad \eta = \frac{P_{\mathrm{meas}}}{\min(P_{\max}, B\, I_{\mathrm{meas}})}.$$

Here η is a roofline efficiency ratio relative to the active bound.

Interpretation for optimisation decisions:

- if $I_{\mathrm{meas}} < I^\star$ and η is low, prioritise memory coalescing, reuse, and layout changes;
- if $I_{\mathrm{meas}} \geq I^\star$ and η is low, prioritise instruction-level efficiency, occupancy/resource balance, and launch overhead reduction;
- if end-to-end speedup is still low while kernel η improves, serial orchestration is likely the limiting factor (Amdahl regime).

Arithmetic-Intensity Map for Rectangular Kernels

To isolate shape effects before full kernel implementation, we model a GEMM-like kernel

$$C_{M\times N} \leftarrow A_{M\times K}B_{K\times N} + C_{M\times N}$$

with element size s bytes (FP32: $s = 4$). Under an ideal one-pass traffic model (read A, read B, read+write C), the arithmetic intensity is

$$I_{\text{ideal}}(M, N, K) = \frac{2MNK}{s\,(MK + KN + 2MN)}.$$

If dimensions are padded to a tile size t for tensor-core-friendly execution, define

$$\hat{M} = \left\lceil \frac{M}{t} \right\rceil t, \quad \hat{N} = \left\lceil \frac{N}{t} \right\rceil t, \quad \hat{K} = \left\lceil \frac{K}{t} \right\rceil t.$$

Using useful FLOPs ($2MNK$) over padded traffic gives the effective intensity

$$I_{\text{eff}}(M, N, K; t) = \frac{2MNK}{s\,(\hat{M}\hat{K} + \hat{K}\hat{N} + 2\hat{M}\hat{N})}.$$

For the target shape regime, we use $K = 64$ and $t = 16$. Figure 3.3 maps I_{ideal} and I_{eff} across $M, N \in [16, 256]$. The trend is consistent with an area-to-perimeter argument: larger and more balanced rectangles increase intensity, while slender shapes and heavy padding reduce it. Under the same traffic model, moving from FP32 to FP64 doubles bytes per value and therefore halves arithmetic intensity at every point of the map.

Table 3.3.: Representative arithmetic-intensity values for fixed $K = 64$ in FP32 with $t = 16$ padding.

Shape (M, N)	I_{ideal}	I_{eff}	Drop (%)	Padded $(\hat{M}, \hat{N}, \hat{K})$
$(47, 84)$	7.76	6.85	11.7	$(48, 96, 64)$
$(63, 95)$	8.67	8.50	2.0	$(64, 96, 64)$
$(65, 130)$	9.20	7.23	21.4	$(80, 144, 64)$
$(111, 73)$	9.27	8.58	7.4	$(112, 80, 64)$
$(128, 128)$	10.67	10.67	0.0	$(128, 128, 64)$

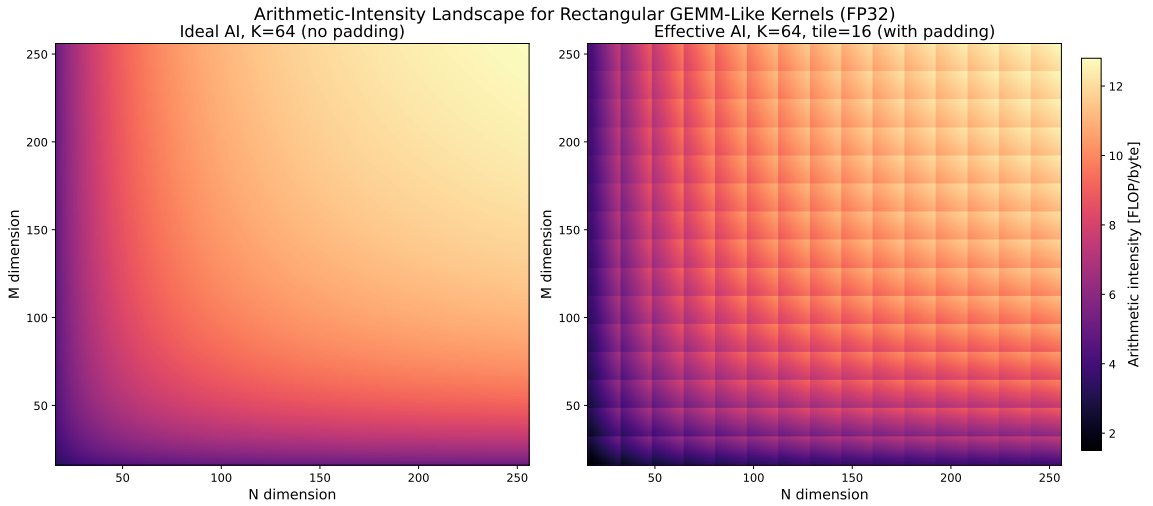

Figure 3.3.: Arithmetic-intensity map for rectangular GEMM-like kernels in FP32 at fixed $K = 64$. Left: ideal intensity without padding. Right: effective intensity with 16-element padding on M, N, K (useful FLOPs over padded traffic).

GEMM Storage Footprint vs A100 Capacity

For square GEMM dimensions (k, k, k), storing input and output matrices (A, B, C) requires

$$M(k; s) = 3k^2 s$$

bytes, where s is bytes per element ($s = 4$ for FP32, $s = 8$ for FP64). This is a storage model only (no temporary workspace, no extra staging buffers). The capacity-limited crossover dimension for a memory budget C is

$$k_{\max}(C; s) = \sqrt{\frac{C}{3s}}.$$

Table 3.4.: Capacity-limited square dimensions for the GEMM storage model $M(k; s) = 3k^2 s$.

Memory limit C	$k_{\max}$ (FP32)	$k_{\max}$ (FP64)
A100 HBM (40 GB)	57 735	40 825
A100 shared memory per SM (164 KiB)	118	84

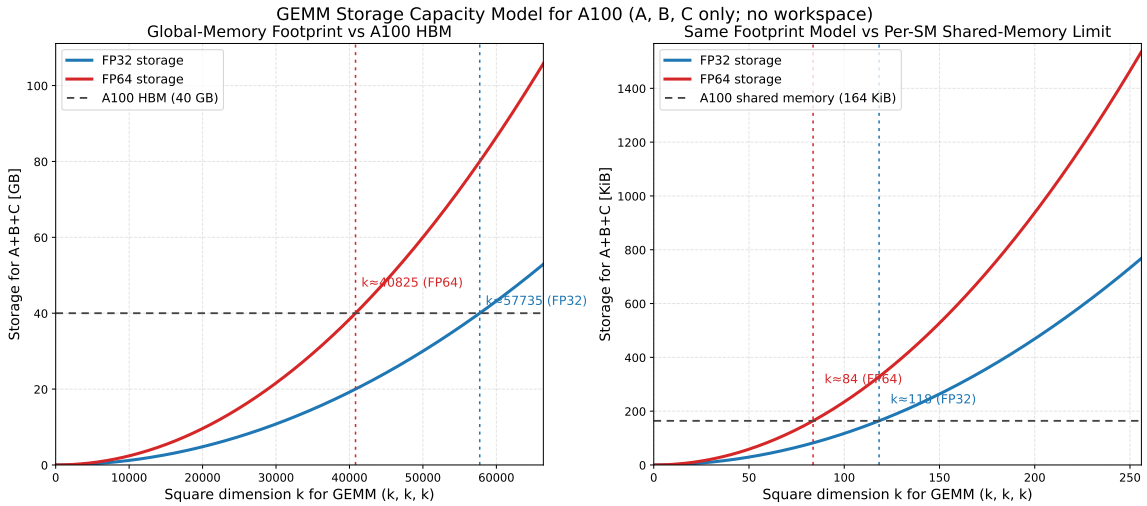

Figure 3.4.: Storage model $M(k; s) = 3k^2 s$ for square GEMM in FP32 and FP64. Left: intersection with A100 HBM capacity (40 GB). Right: same model against per-SM shared-memory capacity (164 KiB).

Amdahl's Law

Let f be the parallelisable fraction of a workload and p the number of parallel workers. Amdahl's law gives the idealised speedup bound

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}.$$

The asymptotic limit as $p \to \infty$ is

$$S_\infty = \frac{1}{1-f}.$$

Even efficient kernels are capped by serial orchestration, launch overhead, synchronisation, and data movement outside the parallel kernel body [Amd67].

Table 3.5.: Amdahl speedup bounds for representative parallel fractions.

Parallel fraction f	$S(4)$	S_∞
0.90	3.08	10.0
0.95	3.48	20.0
0.99	3.88	100.0

These bounds are used as planning constraints: kernel-local optimisation is necessary, but end-to-end scaling also requires reducing serial fractions in launch and orchestration paths.

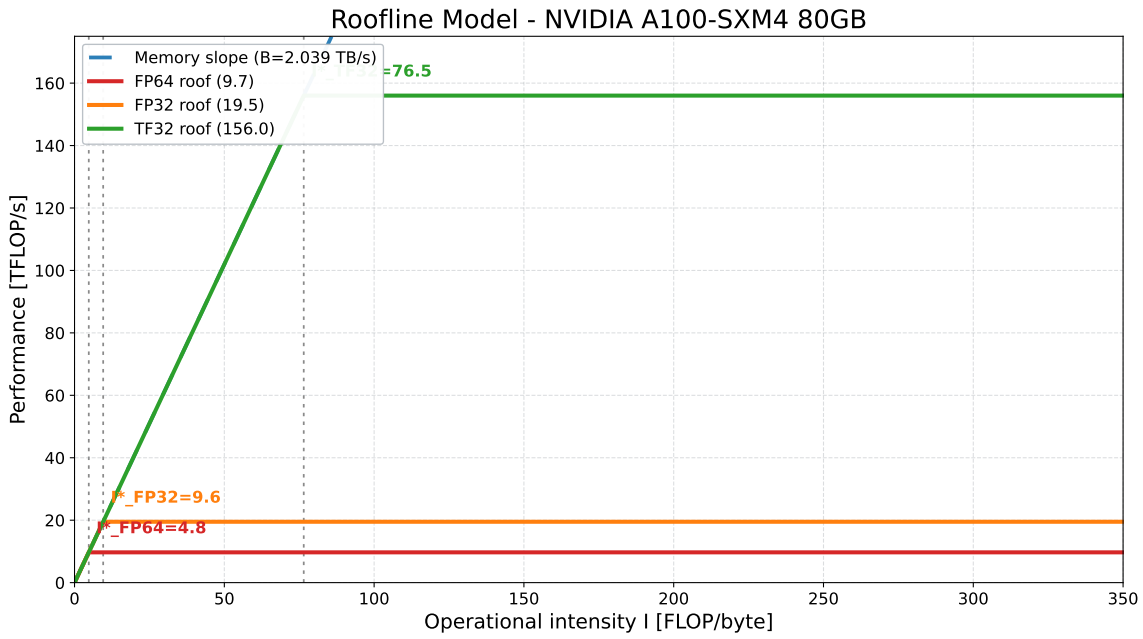

Figure 3.5.: Roofline envelopes for A100-SXM4-80GB. Relative to A100-40GB, the compute ceilings are unchanged while higher memory bandwidth shifts the knees to lower operational intensity.

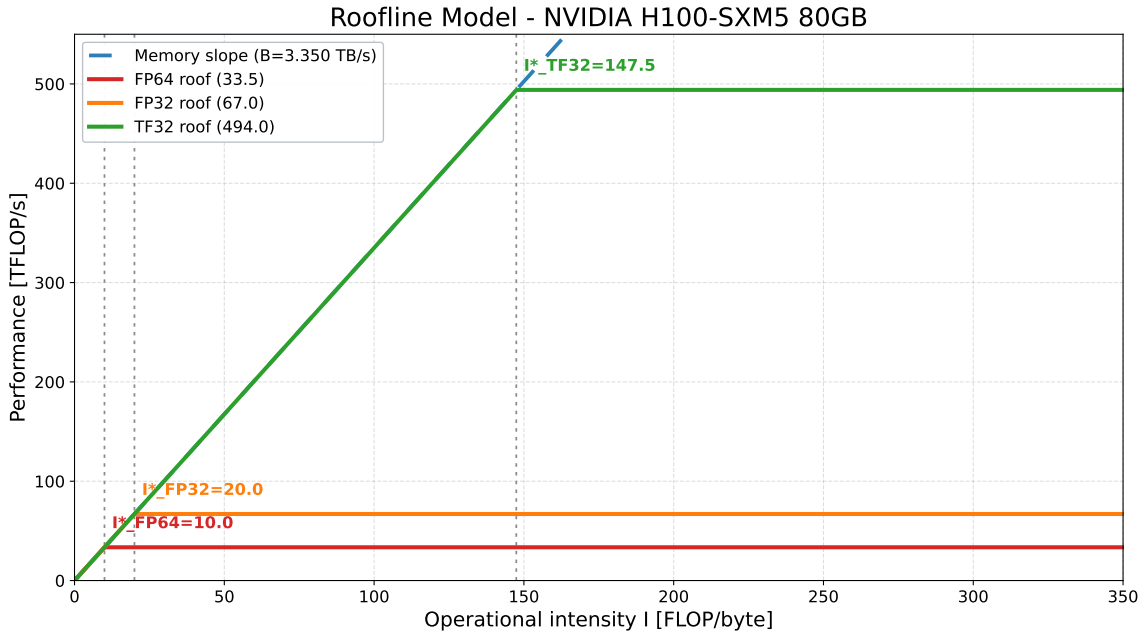

Figure 3.6.: Roofline envelopes for H100-SXM5-80GB using representative peak throughput and bandwidth values from NVIDIA's Hopper documentation.

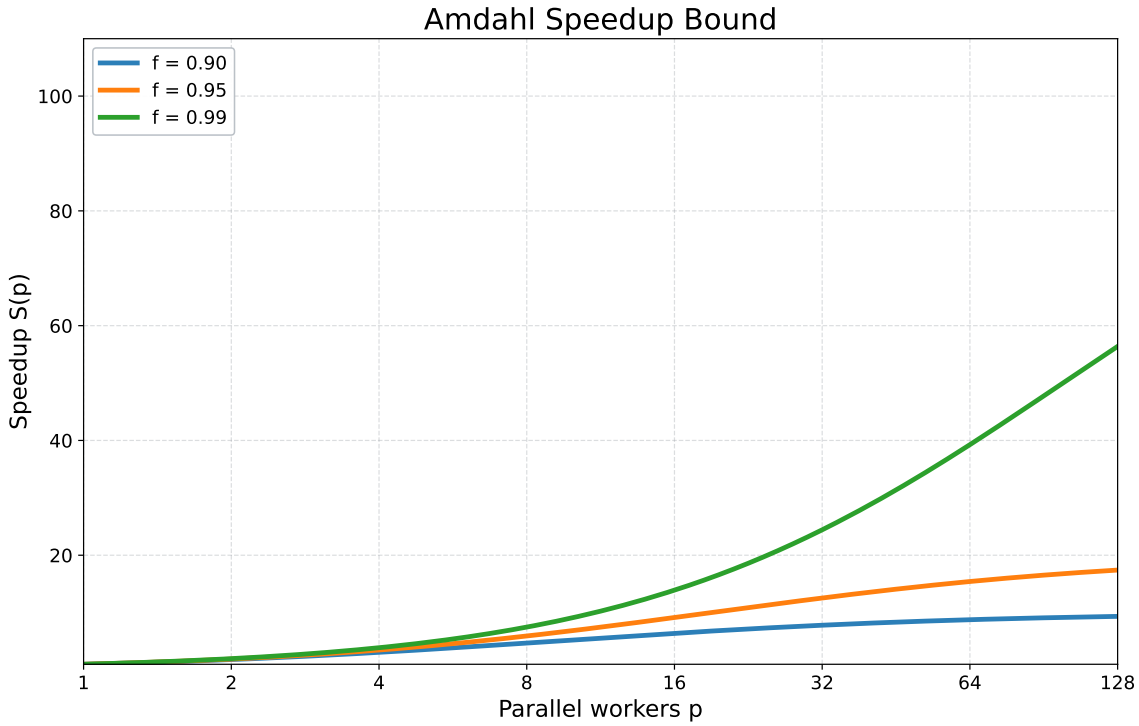

Figure 3.7.: Amdahl speedup bounds for representative parallel fractions ($f \in \{0.90, 0.95, 0.99\}$).

3.4. Data Layout and Memory Strategy

Data layout is treated as a first-class optimisation variable rather than a post-processing detail. For contraction-heavy workloads, memory access patterns often dominate arithmetic cost, so layout decisions are made jointly with kernel design.

3.4.1. Layout Principles

The implementation follows four layout rules:

1. keep the innermost thread-mapped dimension contiguous in memory to preserve coalesced global accesses,

2. minimise explicit transpose and permutation kernels by selecting a stable canonical in-memory layout,

3. perform reshape/pack operations in fused paths when possible,

4. use alignment-friendly leading dimensions to improve transaction efficiency and vectorised loads.

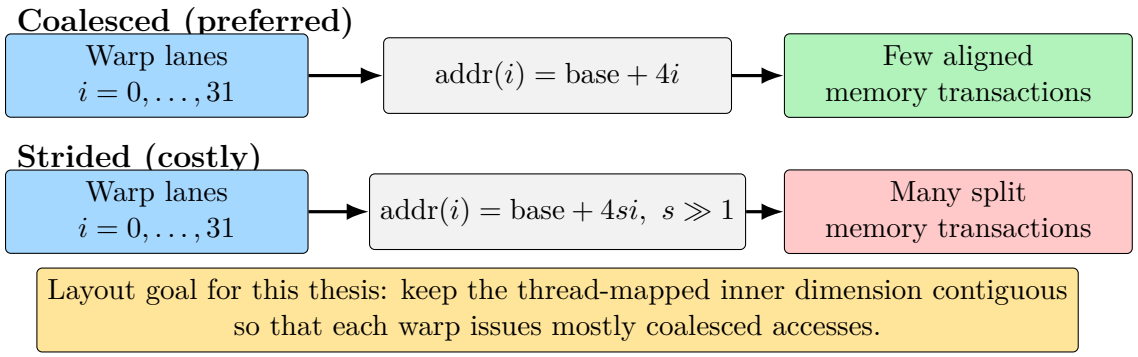

Figure 3.8.: Warp-level address mapping as a layout design rule: contiguous thread-to-address mapping improves global-memory transaction efficiency, while large strides fragment traffic.

3.4.2. Memory-Level Strategy

The memory strategy mirrors the A100 hierarchy:

- **Registers:** accumulate partial results and keep loop-invariant scalars local.
- **Shared memory:** stage reused tiles and remove redundant global loads.
- **L2/HBM:** organise global accesses as contiguous bursts and avoid strided warp patterns unless unavoidable.

When tensors require non-trivial index reorderings, preprocessing overhead is quantified explicitly and accounted for in end-to-end performance claims.

3.5. Baseline Selection

To make performance claims meaningful, all custom kernels are compared against well-defined baselines under matched problem sizes, precision, and memory layout assumptions.

3.5.1. Primary Baselines

- **cuBLAS:** reference for GEMM-like contraction kernels.
- **cuTENSOR:** reference for direct tensor contraction and permutation-centric workflows.
- **cuBLASDx compositions:** reference for device-side small-GEMM execution where launch amortisation is central.

3.5.2. Fairness Criteria

Each comparison uses:

1. identical input tensors and dimensions,

2. identical precision modes and accumulation rules,

3. warm-up runs and repeated measurements for variance control,

4. consistent stream and synchronisation semantics.

Reported metrics include median runtime, throughput (GFLOP/s or TFLOP/s), effective memory bandwidth, and speedup relative to the strongest baseline per workload.

3.5.3. Profiling Evidence Gate

To keep optimisation claims defensible, each accepted speedup is paired with a matching profiler signal:

1. **Launch-amortisation claim:** lower launch count and lower host or launch-overhead fraction at similar numerical output.

2. **Memory-efficiency claim:** higher effective throughput and/or improved coalescing/reuse indicators with matching or lower runtime.

3. **Occupancy/resource claim:** improved runtime with a coherent change in achieved occupancy and stall breakdown.

4. **Precision-path claim (FP32/TF32):** runtime gain plus explicit numerical-deviation check against the FP64/FP32 reference path.

This gate is applied uniformly to custom kernels and library-based baselines, so comparison quality is maintained while the kernel set is still evolving.

3.6. Methodology and Reporting Standards

The following rules define experiment design, profiling, and result reporting. The goal is to keep optimisation claims causal, reproducible, and comparable across kernels and optimisation stages.

3.6.1. Profiling Workflow Standard

Profiling is executed in two stages:

1. **System-level trace first (Nsight Systems).** Identify launch gaps, synchronisation bottlenecks, and overlap failures before collecting kernel microarchitecture counters. Traces are restricted to explicit regions of interest and short wall-clock windows to limit profiling perturbation and trace bloat [NVI25f, Pur24].

2. **Kernel-level trace second (Nsight Compute).** After the timeline bottleneck is localised, collect a fixed KPI pack for attribution. For routine iteration, use a minimal metric set and kernel filters; broad metric sweeps are reserved for focused deep dives because replay passes increase overhead and can bias short kernels [NVI25e].

The workflow is deliberately ordered: first find *where* time is spent, then explain *why* with counters.

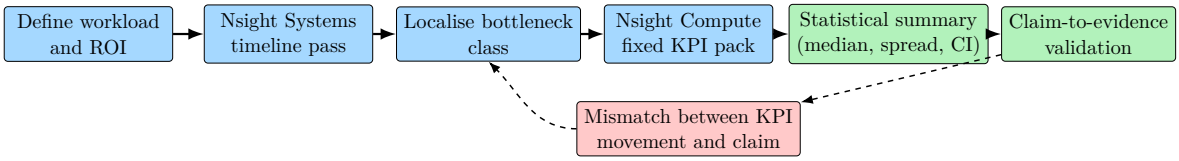

Figure 3.9.: Standard profiling-and-reporting pipeline used in this thesis. Timeline localisation precedes counter collection, and claim validation loops back when mechanism evidence is inconsistent.

3.6.2. Benchmark Design Standard

Each benchmark campaign follows these rules:

1. **Fixed conditions of observation:** hardware partition, software stack, compile flags, precision mode, and CPU/GPU affinity remain constant.

2. **Warm-up and steady-state timing:** warm-up iterations remove first-launch effects; measured iterations are reported separately.

3. **Replicated measurements:** repeated runs are required, with robust summary statistics (median and dispersion).

4. **Bias control:** run-order effects and hidden configuration dependencies are treated as validity risks and explicitly checked.

5. **Cross-workload aggregation:** when normalised speedups are aggregated across workloads, geometric means are used.

These choices follow established benchmarking guidance from systems literature [KJ13, MDHS09, FW86] and reporting standards in benchmark suites [SPE21].

3.6.3. Mandatory Reporting Items

Item	Minimum required disclosure
Platform	GPU model, interconnect/topology, CPU model, memory capacity.
Software stack	Driver, CUDA toolkit, compiler, library versions, profiler version.
Build configuration	Optimisation flags, debug/lineinfo flags, math mode (FP32/TF32/BF16/FP64).
Workloads	Exact tensor dimensions, batch sizes, data types, and input-generation policy.
Timing protocol	Warm-up count, measured iterations, repetitions, synchronisation boundaries.
Profiling protocol	Tool commands, kernel filters, metric set, launch-skip/count, replay-relevant options.
Statistics	Point estimate definition (e.g. median) and spread/uncertainty summary.
Correctness criteria	Numerical tolerance, reference implementation, validation frequency.
Artifacts	Script paths for run submission, CSV extraction, and figure generation.

Table 3.6.: Mandatory disclosure checklist for this thesis.

3.6.4. Claim-to-Evidence Rules

To avoid over-claiming, each optimisation claim must satisfy:

1. **Effect:** measurable runtime/throughput change on the target workload distribution.
2. **Mechanism match:** KPI movement is consistent with the proposed mechanism (e.g. memory, scheduler, occupancy, divergence).
3. **Numerical validity:** output deviation remains within tolerance.
4. **Reproducibility:** the result can be regenerated from documented scripts and environment settings.

If an effect is observed but mechanism counters do not match, the result is reported as empirical but non-attributed; no causal interpretation is claimed.

3.6.5. Threats-to-Validity Template

Each results subsection reports the following validity classes:

- **Internal validity:** profiler perturbation, run-to-run noise, and hidden synchronisation.
- **Construct validity:** mismatch between chosen KPIs and the intended bottleneck interpretation.
- **External validity:** portability limits across architectures, software versions, and tensor-shape distributions.

Mitigations are stated per experiment, using the mandatory reporting items in Table 3.6.

4. Implementation

Chapter 4 translates the design choices into concrete CUDA implementation decisions, with an emphasis on resource-aware kernel engineering and reproducible integration.

4.1. Integration and Build System

The implementation is integrated as a reproducible CUDA/C++ project with clear separation between benchmark harness, kernel library, and profiling scripts. This structure is necessary to compare optimisation variants consistently across multiple datasets and precision modes.

4.1.1. Build Configuration

All kernels target Ampere compute capability (`sm_80`) and are compiled in release mode with architecture-specific optimisation enabled. The build configuration records:

- compiler and CUDA toolkit versions,
- target architecture flags,
- enabled precision modes and feature toggles,
- linked baseline libraries (cuBLAS, cuTENSOR, cuBLASDx where used).

This metadata is logged with benchmark output so each result can be traced to a specific binary configuration.

4.1.2. Runtime Integration

Kernel entry points are exposed through a common interface that supports:

1. baseline execution,
2. custom unfused kernel execution,
3. fused-kernel execution paths.

Using a unified interface allows controlled A/B testing with minimal host-side differences between variants.

4.2. Kernel Design

Kernel design follows a bottom-up strategy: start from a minimal correct implementation, then introduce one optimisation mechanism at a time (tiling, fusion, vectorised loads, launch configuration tuning), validating each step by profiling.

4.2.1. Design Variants

For each target contraction pattern, three variants are maintained:

1. **Reference variant:** straightforward kernel for correctness and baseline profiling.
2. **Memory-optimised variant:** improved access locality and reduced redundant global loads.
3. **Launch-optimised variant:** fused execution that amortises host-side launch overhead.

Maintaining explicit variants makes it easier to attribute performance deltas and prevents "all-at-once" changes that are hard to analyse.

4.2.2. Correctness and Numerical Checks

Each variant is checked against a trusted baseline (library result or high precision reference) with fixed tolerance settings per precision mode. Any performance result is accepted only after numerical agreement is verified.

4.3. Occupancy and Launch Configuration

Launch configuration is tuned as a constrained resource-allocation problem on each SM. The central trade-off is that aggressive register/shared-memory usage can increase per-thread efficiency while reducing occupancy, which may hurt latency hiding.

4.3.1. Tuning Procedure

The tuning sequence is:

1. choose an initial block size (typically 128 or 256 threads),
2. inspect register use and shared-memory use per block,
3. estimate achievable occupancy,
4. benchmark and profile stall reasons,
5. retune block size and launch bounds based on measured bottlenecks.

If memory stalls dominate, higher occupancy is often preferred. If execution is compute-bound with low dependency stalls, lower occupancy with higher per-thread work can be superior.

4.3.2. Launch Overhead Considerations

For small kernels, launch latency can dominate irrespective of occupancy. Therefore occupancy tuning is combined with launch reduction techniques (kernel fusion or persistent execution) rather than treated in isolation.

4.4. Shared Memory Tiling

Shared-memory tiling is used to raise arithmetic intensity by reusing data that would otherwise be fetched repeatedly from global memory. The design objective is to maximise reuse while staying within per-block shared-memory limits and avoiding bank conflicts.

4.4.1. Tile Design Choices

Tile sizes are selected based on:

- contraction dimensions and divisibility,
- register pressure from accumulator blocking,
- shared-memory footprint per block,
- expected occupancy after resource allocation.

Padding is introduced where needed to avoid systematic bank conflicts in column-wise shared-memory accesses.

4.4.2. Boundary Handling

Real workloads frequently use dimensions that are not multiples of tile sizes. Boundary tiles are handled with predicated loads/stores to preserve correctness without launching separate cleanup kernels.

5. Results

Chapter 5 presents benchmark and profiling results for the optimisation process. It starts with the experimental protocol, then reports performance, scaling, profiler evidence, and baseline comparisons.

5.1. Experimental Setup

All measurements are performed on the A100-based JSC environment documented in Chapter 7, with emphasis on reproducibility and fair comparison between custom kernels and vendor-library baselines.

5.1.1. Hardware/Software Baseline

Unless explicitly stated otherwise, experiments use a single node with 4× A100-SXM4-40GB GPUs. Single-GPU measurements are pinned to one device; multi-GPU measurements use NVLink-connected devices within the same node.

The software stack (CUDA toolkit, driver, compiler, profiling tools) is kept fixed across all experiments in a campaign.

5.1.2. Workload Definition

The workload suite includes:

- small to medium GEMM-like contractions representative of target traces,
- a two-site tensor-network contraction-order case study (Section 3.2),
- layout-transform kernels required by contraction workflows,
- fused-kernel variants for launch amortisation studies.

Dimensions are selected to cover both launch-dominated and compute-dominated regions. Problem sets include non-power-of-two sizes to reflect realistic tensor shapes.

5.1.3. Measurement Protocol

Each data point is measured with:

1. warm-up iterations to remove first-launch effects,
2. repeated timed iterations (CUDA events) for stable statistics,
3. Nsight Compute passes with a fixed ten-counter KPI pack for hardware-counter attribution (Table 5.2).

Reported values use median runtime by default; spread (e.g. min/max or percentiles) is included when variance is non-negligible.

5.2. Kernel Launch Overhead Benchmark

As discussed in Section 2.4.5, each CUDA kernel launch incurs a fixed overhead of several microseconds regardless of the kernel's computational workload. For the small GEMM operations typical of tensor network contractions, this overhead can dominate total runtime. The following benchmark quantifies the effect and validates the kernel fusion strategy adopted in this thesis.

5.2.1. Setup

We compare two strategies for executing 1000 identical $n \times n$ double-precision GEMM operations on a single A100-SXM4-40GB GPU:

1. **Separate launches:** each GEMM is dispatched as an individual cuBLASDx kernel, incurring the full launch overhead 1000 times.

2. **Fused kernel:** a single kernel is launched once and performs all 1000 GEMM operations sequentially using cuBLASDx's device-side API [NVI25a], amortising the launch cost to a single invocation.

Matrix sizes range from $n = 1$ to $n = 32$ (square matrices, $m = n = k$), covering the regime of small GEMMs representative of tensor contractions with low to moderate bond dimensions. All matrices use column-major layout and FP64 precision. Timings are recorded using CUDA events with device synchronisation.

5.2.2. Results

Table 5.1 reports the measured total execution time for both strategies, along with the derived speedup factor and the fraction of the separate-launch time attributable to overhead. Figure 5.1 visualises the same data.

Table 5.1.: Total execution time for 1000 GEMM operations ($n \times n$, FP64) on an A100 GPU. *Separate* denotes 1000 individual kernel launches; *Fused* denotes a single kernel performing all 1000 operations internally. The overhead column gives the fraction of the separate-launch time attributable to launch overhead rather than computation.

n	Separate (μs)	Fused (μs)	Speedup	Overhead (%)
1	8022	172	46.7×	97.9
2	7941	185	43.0×	97.7
4	7835	180	43.5×	97.7
8	7895	180	43.8×	97.7
13	8477	355	23.9×	95.8
16	8104	347	23.4×	95.7
23	9217	717	12.9×	92.2
32	9752	1380	7.1×	85.9

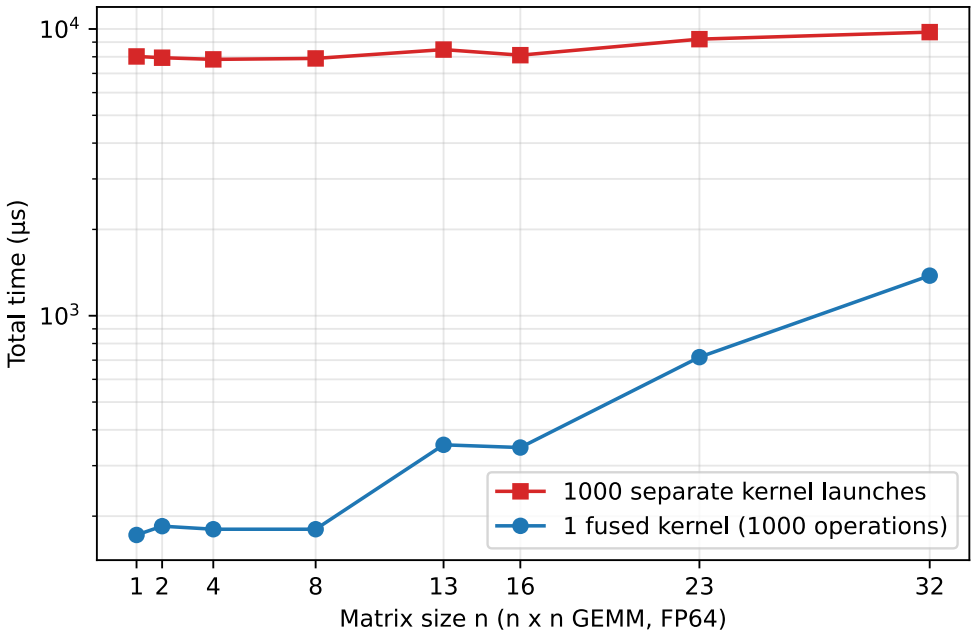

Figure 5.1.: Total time for 1000 GEMM operations as a function of matrix size, comparing 1000 separate kernel launches (red) against a single fused kernel (blue). The near-constant red curve below $n = 16$ confirms that launch overhead dominates over actual computation for small matrices.

5.2.3. Discussion

Two points are clear from the data. First, the total time for separate launches is nearly constant across all matrix sizes, varying only from 8022 μs at $n = 1$ to 9752 μs at $n = 32$. This confirms that the per-launch overhead of approximately 8 μs dominates the total runtime: the actual GEMM computation contributes less than 3% of the measured time for $n \leq 8$. Only at $n = 32$ does the computation begin to represent a meaningful fraction (14%) of the total.

Second, the fused kernel eliminates this overhead almost entirely. At $n = 1$, where the GEMM itself is trivial, fusion yields a 46.7× speedup. As the matrix size increases and the per-GEMM computation grows, the relative benefit of fusion decreases—but even at $n = 32$, the fused kernel remains 7.1× faster.

These results directly motivate the kernel fusion strategy employed throughout this thesis. Tensor network algorithms typically require many contractions of modest size, which places them in the regime where launch overhead dominates. By fusing these contractions into a single kernel using cuBLASDx's device-side GEMM API, the launch cost is paid once rather than once per contraction. The design and implementation of the fused kernels are described in Section 4.2.

5.3. Single-GPU Performance

Single-device performance is reported for the target kernel set, with custom implementations compared against cuBLAS/cuTENSOR baselines.

5.3.1. Metrics

The primary metrics are:

- runtime per operation,
- achieved throughput (GFLOP/s or TFLOP/s),
- effective memory bandwidth,
- speedup relative to the strongest baseline.

5.3.2. Evaluation Dimensions

Results are stratified by:

1. problem size regime (small, medium),
2. precision mode (FP64, FP32, optional TF32),
3. kernel variant (baseline, memory-optimised, fused).

The split separates gains from arithmetic optimisation and gains from reduced launch overhead.

5.4. Scaling Behaviour

The analysis tracks how performance evolves when moving from one GPU to multiple GPUs within a single NVLink-connected node.

5.4.1. Strong and Weak Scaling Views

- **Strong scaling:** fixed total problem size, increasing GPU count.
- **Weak scaling:** per-GPU problem size fixed, total size increases with GPU count.

Both views are needed because kernel-level speedups may not translate linearly once communication and synchronisation costs are included.

5.4.2. Communication Effects

Scaling analysis explicitly separates:

1. pure compute time on each GPU,
2. peer-to-peer transfer time over NVLink,
3. host-side orchestration and synchronisation overhead.

The decomposition identifies when scaling is limited by communication topology rather than kernel efficiency.

ID	Primary counter name (Nsight Compute)
K1	`sm__throughput.avg.pct_of_peak_sustained_elapsed`
K2	`dram__throughput.avg.pct_of_peak_sustained_elapsed`
K3	`lts__throughput.avg.pct_of_peak_sustained_elapsed`
K4	`l1tex__t_sector_hit_rate.pct`
K5	`smsp__pipe_fma_cycles_active.avg.pct_of_peak_sustained_active`
K6	`sm__warps_active.avg.pct_of_peak_sustained_active`
K7	`launch__occupancy_limit_registers`
K8	`smsp__warps_eligible.sum.per_cycle_active`
K9	`smsp__issue_active.avg.pct_of_peak_sustained_active`
K10	`smsp__thread_inst_executed_per_inst_executed.pct`

Table 5.2.: KPI identifier to primary metric mapping used for reporting. Scripts keep compatibility fallbacks for Nsight Compute version differences.

5.5. Profiling Analysis

Nsight Compute data is used to explain *why* performance changes after each optimisation, not only *how much* it changes.

5.5.1. KPI Pack Used for Attribution

To keep profiling interpretable and comparable, all Nsight Compute runs in this thesis use a fixed ten-counter KPI pack. The profiling scripts in `code/profiling/` use exact metric names when available and otherwise fall back to equivalent names for the installed Nsight Compute version.

Interpretation by KPI.

K1 (SM throughput). K1 reports how much total SM work was completed relative to the sustained peak for the device. It is a top-level progress indicator across execution pipelines, so high values mean the SMs are busy doing useful work. Low K1 indicates unused compute capacity, but it does not by itself say whether the cause is memory stalls, low occupancy, or poor issue efficiency.

K2 (DRAM throughput). K2 measures bandwidth at the HBM interface as a percentage of sustained peak. High K2 means the kernel is pushing the DRAM subsystem hard, often indicating a bandwidth-sensitive region. If K2 is high while compute-side counters stay modest, the kernel is usually memory-bound.

K3 (L2 throughput). K3 measures traffic pressure through the L2 cache fabric relative to peak. It helps distinguish kernels limited by on-chip cache traffic from kernels limited in the core pipelines. High K3 with weak compute counters often means data movement inside the memory hierarchy dominates.

K4 (L1 hit rate). K4 is the fraction of L1/texture requests served by L1 instead of missing to lower levels. Higher values usually mean better locality and lower latency per memory access. Low K4 is not automatically bad for streaming kernels with little temporal reuse, so it must be read together with K2/K3 and runtime.

K5 (FP32 FMA pipe active). K5 reports how active the FP32 FMA pipelines are relative to their sustained peak activity. It is the most direct signal for whether FP32 arithmetic hardware is actually being used. High K1 but low K5 usually means time is spent in non-FMA instructions or waiting, not in dense FP32 math.

K6 (warps active). K6 is runtime occupancy: active warps as a percentage of the architectural peak during kernel execution. High K6 means many warps are resident and participating, which improves latency hiding potential. However, high K6 alone does not guarantee speed if those warps are mostly stalled.

K7 (register occupancy limit). K7 captures how strongly register allocation constrains theoretical occupancy. A stronger register limit means fewer blocks/warps can reside on an SM at once, reducing latency-hiding headroom. It should be interpreted with K6 and K8 to separate static resource limits from dynamic scheduler starvation.

K8 (warps eligible per cycle). K8 measures how many warps are ready to issue in each active cycle of an SM sub-partition. High K8 means the scheduler has choices and can keep issue slots fed. Low K8 is a direct starvation signal, typically from dependencies or long-latency memory waits.

K9 (issue active). K9 reports how often issue slots actually dispatch instructions relative to peak sustained activity. High K9 means the front end is productive and instruction delivery is healthy. Low K9 together with low K8 indicates not enough ready warps, while low K9 with high K8 can indicate other front-end bottlenecks.

K10 (thread-inst per inst). K10 approximates average participating threads per issued instruction as a percentage of full-warp execution. Values near 100% indicate mostly uniform control flow, while values around 50% indicate substantial lane under-utilisation from divergence or predication. It is therefore the primary SIMT efficiency counter for branch-heavy kernels.

5.5.2. Interpretation Rules

Attribution follows these rules under identical problem sizes and launch configurations:

1. **Compute-limited pattern:** K1 and K5 are high, while K2/K3 are not dominant.
2. **Memory-limited pattern:** K2 and/or K3 are high, K8/K9 are reduced due to memory waiting, and throughput scales weakly with more FMAs.
3. **Occupancy/resource-limited pattern:** K7 indicates a register limit and K6 remains below expected active-warp levels.
4. **Scheduler starvation pattern:** K8 and K9 drop together, indicating insufficient eligible warps or long-latency dependencies.
5. **Divergence/predication loss pattern:** K10 decreases while kernel time increases under otherwise similar arithmetic work.

Stride	Avg. kernel time (ms)	Effective BW (GB/s)	$BW/BW_{s=1}$
1	0.107776	1245.34	1.000
2	0.152576	879.68	0.706
4	0.242893	552.58	0.444
8	0.432947	310.01	0.249
16	0.822886	163.11	0.131
32	0.841216	159.55	0.128
64	0.600320	223.58	0.180
128	0.237824	564.36	0.453

Table 5.3.: Coalescing sweep results from the synthetic stride microbenchmark on A100 (FP32 loads/stores).

5.5.3. Application to Benchmark Pairs

The same KPI pack is applied to both benchmark pairs used in this chapter:

- **FP32 GEMM pair (cuBLAS vs naive):** expected separation is high K1/K5/K9 for cuBLAS and lower compute-issue efficiency for the naive kernel, with different memory-pressure signatures in K2–K4.
- **Warp-divergence pair (uniform vs divergent):** expected separation is primarily in K10, with secondary reductions in K8/K9 and effective throughput for the divergent variant.
- **TN two-site contraction pair (ordered vs direct):** expected separation is higher K1/K5/K9 for the ordered strided-batched GEMM mapping and lower scheduler efficiency plus heavier memory pressure for the direct contraction-order baseline.

Measured example values. For the FP32 GEMM benchmark pair, the measured counters show: K1 98.45% (cuBLAS) vs 54.32% (naive), K5 98.67% vs 28.05%, K9 56.15% vs 31.50%, with higher memory pressure in the naive kernel (K2 16.54% vs 3.62%). The numbers indicate that the naive kernel is not occupancy-limited (K6 is high) but issue/compute-efficiency-limited.

For the warp-divergence pair, the key discriminator is K10: 100% (warp-uniform) vs 50.35% (forced divergent split), matching the expected half-warp execution efficiency for a lane-based 16/16 branch split.

Using a fixed KPI pack reduces trial-and-error profiler exploration and keeps profile evidence comparable across optimisation steps.

5.5.4. Global-Memory Coalescing Sweep

From stride 1 to stride 32, average kernel time increases from 0.108 ms to 0.841 ms (7.8×), while the effective bandwidth proxy drops from 1245 to 160 GB/s (0.13×). This is the expected signature of reduced coalescing: lanes in a warp map to more memory sectors, increasing memory transactions per useful word. For tensor network kernels this result shows that operand layouts in critical kernels should preserve unit-stride accesses for warp-contiguous dimensions whenever possible.

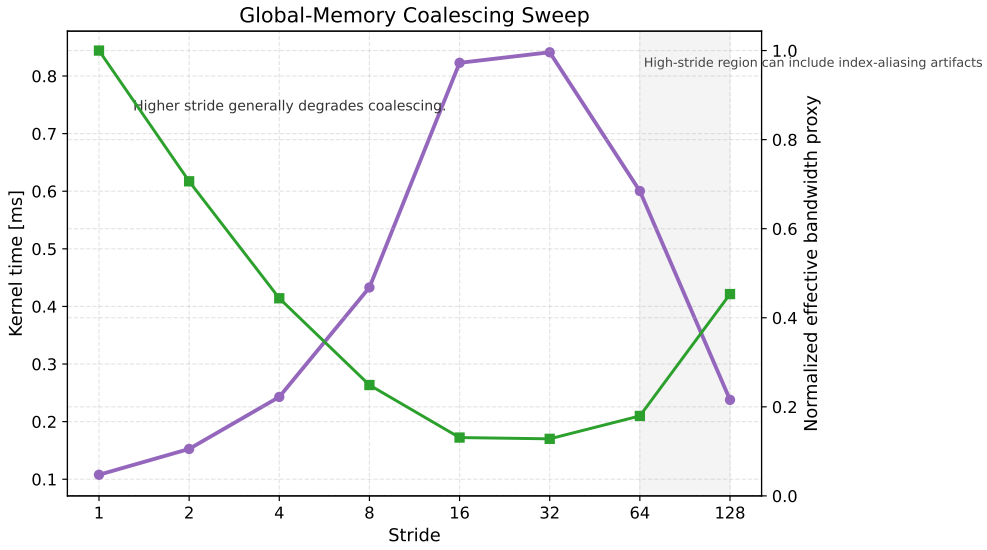

Figure 5.2.: Global-memory coalescing sweep over stride.

Variant	Avg. kernel time (ms)	Core effective TFLOP/s	Theoretical occupancy (%)
Low register pressure	0.594150	14.46	100.0
High register pressure	0.671002	12.80	62.5

Table 5.4.: Register-pressure microbenchmark results on A100 (FP32 arithmetic core work).

The partial recovery at stride 64 and 128 is treated as a benchmark artifact from the masked synthetic index mapping (`idx = (tid * stride) & (N-1)`), which introduces aliasing/reuse at large power-of-two strides. Therefore, interpretation is based primarily on the monotonic degradation regime up to stride 32.

5.5.5. Register Pressure and Occupancy

Increasing live register state reduces theoretical occupancy from 100.0% to 62.5%, increases runtime by about 12.9%, and reduces core effective throughput from 14.46 to 12.80 TFLOP/s (about 11.5%). This indicates that the high-register variant exposes less latency hiding due to fewer concurrently resident warps per SM. For contraction kernels, this supports limiting accumulator blocking and aggressive unrolling when they push register usage past the occupancy-efficient range.

5.6. Comparison with cuBLAS and cuTENSOR

Optimised kernels are compared with vendor-library baselines under matched precision, layout assumptions, and workload definitions.

5.6.1. Comparison Protocol

For each workload class:

1. run cuBLAS/cuTENSOR baseline,

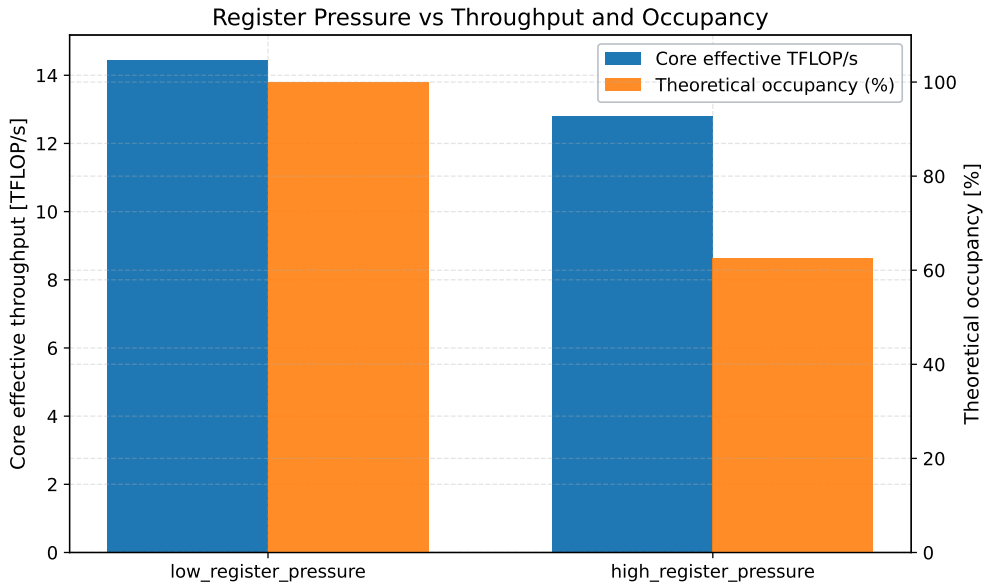

Figure 5.3.: Register pressure versus throughput and occupancy.

2. run custom unfused variant,
3. run custom fused/layout-aware variant,
4. report speedups and counter-based explanation.

The strongest baseline per workload is used as denominator for reported speedup to avoid optimistic bias.

5.6.2. Interpretation Strategy

Outperformance is interpreted by regime:

- if custom kernels win in small sizes, launch reduction is the likely mechanism,
- if wins appear in medium sizes with high bandwidth utilisation, memory access improvements are the likely mechanism,
- if libraries remain dominant at larger sizes, this indicates the practical limit of workload-specific optimisation.

5.7. Energy Efficiency Considerations

Performance is the primary optimisation target in this thesis, but energy-to-solution is also an operational objective in HPC: cluster power budgets are finite, and lower joules per job improve both operating cost and system throughput under queue load [KPB+23, TOP25]. For this reason, the same optimisation decisions used for speed (contraction order, coalescing, launch reduction, occupancy control) are evaluated here from an energy perspective as well.

5.7.1. Why Energy is in Scope for This Thesis

The target workload class (tensor-network-style contractions) executes many short and medium kernels in iterative loops. In this regime, wasted launches, redundant arithmetic, and avoidable memory traffic increase not only runtime but also board energy. On A100-class GPUs, where arithmetic throughput is high and kernel orchestration is often the bottleneck for moderate sizes, runtime improvements frequently translate directly into energy improvements at fixed clock/power settings [NVI20a, NVI25b].

Scope note: the measurements and models in this section focus on GPU board energy for relative comparison between algorithmic variants on the same node. Whole-facility effects (cooling plant, PUE, rack-level sharing) are out of scope.

5.7.2. Metrics

For one benchmark run of duration T, instantaneous board power $P(t)$, and total floating-point work F, the core quantities are

$$E = \int_0^T P(t)\,\mathrm{d}t \approx \bar{P}\,T, \qquad \eta_{\mathrm{F}} = \frac{F}{E}, \qquad \eta_{\mathrm{T}} = \frac{\text{time-to-solution}^{-1}}{\bar{P}},$$

where E is energy-to-solution (J), η_{F} is computational energy efficiency (FLOP/J), and η_{T} is throughput per watt for a fixed workload.

In addition, a standard trade-off metric is energy-delay product (EDP):

$$\mathrm{EDP} = E \cdot T,$$

and, when latency is weighted more strongly, energy-delay-squared product:

$$\mathrm{ED^2P} = E \cdot T^2.$$

These metrics are useful when comparing alternatives that exchange small runtime gains for larger power increases (or vice versa) [HTHW16].

For memory-intensive kernels with transferred bytes Q, a useful auxiliary metric is

$$\eta_{\mathrm{B}} = \frac{Q}{E} \quad [\text{bytes/J}],$$

which helps identify memory-system inefficiency.

5.7.3. First-Order Model for Contraction Kernels

A practical starting point is

$$E = \int_0^T P(t)\,\mathrm{d}t = P_{\mathrm{base}}\,T + \int_0^T P_{\mathrm{dyn}}(t)\,\mathrm{d}t.$$

Approximating the dynamic term by compute- and data-movement work yields

$$E \approx P_{\mathrm{base}}\,T + e_{\mathrm{F}}\,F + e_{\mathrm{B}}\,Q,$$

with:

- P_{base}: baseline board power while the GPU is active,

- e_{F}: effective energy per floating-point operation,
- e_{B}: effective energy per byte moved through memory.

The model is intentionally coarse, but it is sufficient for optimisation decisions and consistent with the runtime decomposition used in Section 2.2 [WWP09, HJ88]. It captures the central trade-off in this thesis: optimisations that reduce runtime T, redundant work F, or memory traffic Q usually improve energy efficiency simultaneously.

5.7.4. Implications for This Thesis

The profiling results already support concrete energy-efficiency expectations:

1. **Contraction-order improvement** (TN case study) reduces arithmetic work from $\mathcal{O}(Bd_p^2\chi^2)$ to $\mathcal{O}(B(d_p^2\chi + d_p\chi^2))$, so both time-to-solution and expected energy-to-solution decrease.
2. **Coalescing improvements** reduce excess memory transactions, improving both runtime and bytes/J in memory-sensitive regimes.
3. **Launch overhead reduction** is energy-relevant: many tiny launches spend time at non-trivial board power without proportional progress.
4. **Register-pressure control** avoids occupancy collapse that prolongs runtime and therefore increases $P_{\mathrm{base}}\,T$.

The same mechanisms that improve throughput in Section 5.5 and Tables 5.3 and 5.4 also dominate energy-to-solution.

5.7.5. Measurement Protocol on A100 Nodes

On the JSC platform used here, practical energy measurement can be added with minimal workflow change:

1. sample board power during each run via `nvidia-smi` (`power.draw`) telemetry,
2. align measurement start/stop with explicit CUDA synchronisation points so the integration window matches kernel execution,
3. integrate sampled power over wall-clock time (trapezoidal rule) to estimate E,
4. repeat runs and report robust summaries (median and spread) together with runtime and Nsight KPIs.

With samples $(t_i, P_i)_{i=1}^{m}$, the numerical estimate is

$$E \approx \sum_{i=1}^{m-1} \frac{P_i + P_{i+1}}{2}\,(t_{i+1} - t_i).$$

Representative telemetry fields and command syntax are documented in the `nvidia-smi` documentation [NVI25g]. The workflow is appropriate for relative algorithmic comparisons, but absolute values should be interpreted with care because board-level telemetry has finite sampling granularity and excludes non-GPU node power. Repeated measurements and fixed protocols mitigate this risk [KJ13].

5.8. Discussion

Kernel-level and system-level findings are combined here into practical guidance for future optimisation cycles.

5.8.1. What Improved and Why

The main question is not only whether performance improved, but which mechanism was responsible (launch reduction, memory locality, occupancy tuning, or precision mode selection). Profiling evidence is used to support each claim, using the fixed KPI pack defined in Table 5.2 so that attributions remain comparable across kernels and optimisation stages.

5.8.2. How Well Results Transfer

Improvements are classified as:

- **likely to carry over** (expected to transfer across Ampere-like systems),
- **topology dependent** (depends on node NUMA/NVLink structure),
- **workload dependent** (effective only for the studied contraction distribution).

5.8.3. Microbenchmark Implications for Tensor Networks

The two most useful profiling microbenchmarks are the coalescing sweep (Table 5.3 and Figure 5.2) and the register pressure study (Table 5.4 and Figure 5.3).

For coalescing, the stride-1 to stride-32 degradation quantifies how quickly memory efficiency collapses when warp lanes lose contiguous access patterns. The result supports prioritising layout transforms and contraction ordering that keep the hottest contracted index unit-stride in memory.

For register pressure, the occupancy drop from 100% to 62.5% with an accompanying throughput loss shows that "more in-register work" is not monotonically beneficial. In practice, this motivates tuning unroll depth and accumulator count jointly with occupancy targets rather than maximising instruction-level reuse in isolation.

6. Conclusion

6.1. Summary

This thesis treats tensor-network workloads from an HPC perspective: identify kernel bottlenecks on A100-class GPUs, apply profile-guided optimisations, and quantify performance relative to vendor-library baselines. The main control variables are launch overhead, memory-hierarchy efficiency, and occupancy-resource trade-offs.

6.2. Limitations

The current scope has three deliberate limitations:

- physics-specific algorithm derivations and notation are out of scope,
- optimisation focus is on Ampere-era kernels and one-node execution,
- coverage of workload diversity is bounded by available benchmark traces.

6.3. Future Work

Natural continuation points are:

1. extending the same methodology to Hopper/H100-class hardware while preserving comparability with A100 results,
2. deeper multi-GPU overlap strategies (communication/computation pipelining),
3. automated kernel-configuration search constrained by profiler-derived bottleneck models.

Bibliography

[A+23] Thomas Ayral et al. Density matrix renormalization group algorithm for simulating quantum circuits with a finite fidelity. *PRX Quantum*, 4(3):030307, 2023. https://arxiv.org/abs/2302.01941.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[Boe22] Simon Boehm. How to optimize a cuda matmul kernel for cublas-like performance: a worklog, 2022. https://siboehm.com/articles/22/CUDA-MMM.

[DR22] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer Spektrum, 3 edition, 2022.

[ET94] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall/CRC, 1994.

[Fru20] Andrei Frumusanu. TSMC details its N5 process technology: Performance, density, and defect improvements, 2020. https://www.anandtech.com/show/16028/tsmc-details-n5.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[GWMW20] Elijah Gilman, Samuel Walls, Alexander Merritt, and Bryan C. Ward. Demystifying the placement policies of the NVIDIA GPU thread block scheduler. In *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2020. https://cake.wpi.edu/assets/papers/gilman20_performance.pdf.

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2 edition, 2002.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*. Adam Hilger, 1988.

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

[HR09] Peter J. Huber and Elvezio M. Ronchetti. *Robust Statistics*. Wiley, 2 edition, 2009.

[HTHW16] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multicore chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210, 2016.

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

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[KJ13] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 63–74. ACM, 2013.

[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.

[KPB+23] Jan Koćot, Dmitrii Pogodaev, Damian Bajer, Miroslaw Lisowski, Marek Gorgon, and Jacek Leszczynski. Energy-aware computing and practical measures for energy efficiency: A comprehensive survey. *Energies*, 16(2):890, 2023.

[KRW25] Aleksei V. Krinitsin, Matteo Rizzi, and Xavier Waintal. Time evolution of the quantum ising model in two dimensions using tree tensor networks. *arXiv preprint arXiv:2505.07612*, 2025. https://arxiv.org/abs/2505.07612.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–276. ACM, 2009.

[Mel24] Chris Mellor. Intel says TSMC 2nm defect density better than 3nm, 5nm, and 7nm at same stage, 2024. https://www.tomshardware.com/tech-industry/intel-says-tsmc-2nm-defect-density-better-than-3nm-5nm-and-7nm-at-same-stage.

[ML+23a] Tamas Menczer, Ors Legeza, et al. Boosting the effective performance of massively parallel tensor network state algorithms on hybrid CPU-GPU based architectures via non-Abelian symmetries. *arXiv preprint arXiv:2309.16724*, 2023. https://arxiv.org/abs/2309.16724.

[ML+23b] Tamas Menczer, Ors Legeza, et al. Massively parallel tensor network state algorithms on hybrid CPU-GPU based architectures. *arXiv preprint arXiv:2305.05581*, 2023. https://arxiv.org/abs/2305.05581.

[ML+24] Tamas Menczer, Ors Legeza, et al. Parallel implementation of the density matrix renormalization group method achieving a quarter petaflops performance on a single DGX-H100 GPU node. *arXiv preprint arXiv:2407.07411*, 2024. https://arxiv.org/abs/2407.07411.

[NVI20a] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*, 2020. Whitepaper v1.0.

[NVI20b] NVIDIA Developer Blog. NVIDIA ampere architecture in-depth, 2020. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[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.

[NVI25a] NVIDIA Corporation. cuBLASDx: Device side BLAS extensions, 2025. https://docs.nvidia.com/cuda/cublasdx/.

[NVI25b] NVIDIA Corporation. *CUDA C++ Best Practices Guide*, 2025. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[NVI25c] NVIDIA Corporation. NVIDIA a100 tensor core GPU: Product specifications, 2025. https://www.nvidia.com/en-us/data-center/a100/.

[NVI25d] NVIDIA Corporation. *NVIDIA Ampere GPU Architecture Tuning Guide*, 2025. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html.

[NVI25e] NVIDIA Corporation. *NVIDIA Nsight Compute Profiling Guide*, 2025. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html.

[NVI25f] NVIDIA Corporation. *NVIDIA Nsight Systems User Guide*, 2025. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[NVI25g] NVIDIA Corporation. *NVIDIA System Management Interface (nvidia-smi)*, 2025. https://docs.nvidia.com/deploy/nvidia-smi/index.html.

[Oru14] Roman Orus. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[Pur24] Budirijanto Purnomo. Understanding the visualization of overhead and latency in NVIDIA Nsight systems. NVIDIA Technical Blog, 2024. https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/.

[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.

[Sch11] Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of Physics*, 326(1):96–192, 2011.

[SPE21] SPEC Open Systems Group. *SPEC CPU2017 Run and Reporting Rules*, 2021. https://www.spec.org/cpu2017/Docs/runrules.html.

[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.

[TOP25] TOP500 Project. The Green500 List, 2025. https://www.top500.org/lists/green500/.

[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.

[Whi92] Steven R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69(19):2863–2866, 1992.

[Whi93] Steven R. White. Density-matrix algorithms for quantum renormalization groups. *Physical Review B*, 48(14):10345–10356, 1993.

[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.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[ZSW20] Shangnan Zhou, E. Miles Stoudenmire, and Xavier Waintal. What limits the simulation of quantum computers? *Physical Review X*, 10(4):041038, 2020. `https://arxiv.org/abs/2002.07730`.

Repro

7. Experimental Environment and Reproducibility

All experiments reported in this thesis were conducted on a single compute node of the JURECA-DC cluster at the Jülich Supercomputing Centre (JSC). The hardware and software configuration is detailed below to facilitate reproducibility.

7.1. Hardware Configuration

Table 7.1.: Compute node hardware specification.

Component	Specification
GPUs	4× NVIDIA A100-SXM4-40GB
GPU memory per device	40 GB HBM2
GPU interconnect	NVLink 3.0, 4 links per GPU pair
NVLink bandwidth per link	25 GB/s (uni-directional)
PCIe	Gen 4 ×16
CPUs	2× AMD EPYC 7742, 64 cores per socket
CPU threads (total)	256 (SMT-2)
NUMA domains	8
Host memory	503 GiB DDR4
Network	2× Mellanox ConnectX (mlx5) HDR InfiniBand (200 Gbit/s)

The four GPUs are fully connected via NVLink 3.0 with four links between every pair (NV4 topology), providing 100 GB/s of uni-directional bandwidth between any two devices. Each GPU is associated with a distinct NUMA domain; GPU-affine memory allocation was used where applicable.

7.2. Software Environment

Table 7.2.: Software versions used for all experiments. Components marked with † are loaded via the JSC module system.

Software	Version
Operating system	Rocky Linux 9.7 (Blue Onyx)
Linux kernel	5.14.0-611.16.1.el9_7.x86_64
NVIDIA driver	590.48.01
CUDA Toolkit†	12.6.20
GCC†	13.3.0
CMake†	3.29.3
C++ standard	C++20
Python	3.9.25
NVIDIA Libraries	
MATHDx†	25.06.0
cuBLASDx	0.4.0 (bundled with MATHDx)
CUTLASS	3.9.0 (bundled with MATHDx)
cuBLAS	Bundled with CUDA Toolkit 12.6
Profiling Tools	
Nsight Compute (ncu)†	2024.3.0.0 (build 34567288)
Nsight Systems (nsys)†	TBD

The NVIDIA driver (590.48.01) reports forward-compatibility with CUDA 13.1; however, all code was compiled against the CUDA 12.6 toolkit. CUTLASS and cuBLASDx require compute capability 8.0 (Ampere) or higher; all kernels were compiled with `-DCMAKE_CUDA_ARCHITECTURES=80`.

7.3. GPU Topology

Table 7.3 summarises the interconnect topology as reported by `nvidia-smi topo -m`. All GPU pairs are connected via NV4 (four bonded NVLinks). The two network interfaces (mlx5_0, mlx5_1) are each PCIe-local to one GPU (GPU 1 and GPU 2 respectively), which is relevant for multi-node communication patterns.

Table 7.3.: GPU interconnect topology. NV4 = four bonded NVLinks; PIX = single PCIe bridge; SYS = traverses PCIe and inter-NUMA interconnect.

	GPU 0	GPU 1	GPU 2	GPU 3	CPU affinity	NUMA
GPU 0	—	NV4	NV4	NV4	48–63, 176–191	3
GPU 1	NV4	—	NV4	NV4	16–31, 144–159	1
GPU 2	NV4	NV4	—	NV4	112–127, 240–255	7
GPU 3	NV4	NV4	NV4	—	80–95, 208–223	5

7.4. Build and Run Procedure

The project uses CMake as its build system. A minimal build on JURECA-DC proceeds as follows:

```
# Load required modules (JURECA-DC, Stages/2025)
module load Stages/2025
module load CUDA/12.6 GCC/13.3.0 CMake/3.29.3 MATHDx/25.06.0

git clone https://github.com/<your-repo>.git && cd <your-repo>
mkdir build && cd build
cmake .. -DCMAKE_CUDA_ARCHITECTURES=80 -DCMAKE_BUILD_TYPE=Release
make -j$(nproc)
```

Listing 7.1: Build commands for JURECA-DC.

7.5. Measurement Methodology

7.6. Data Availability

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 20, 2026 Daniel Sinkin