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)<br>Forschungszentrum Jülich |

Aachen, February 18, 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

# List of Figures

# List of Tables

# Listings

# List of Symbols

| | |
|---|---|
| $\mathcal{T}$ | Tensor |
| $\mathbf{A}, \mathbf{B}$ | Matrices |
| $\vec{v}$ | Vector |
| $\chi$ | Bond dimension |
| $d$ | Local (physical) dimension |
| $N$ | Matrix/problem size |
| $\mathcal{O}(\cdot)$ | 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 $\varepsilon$ 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. $4 \times 4$ or $8 \times 8$) 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

## 1.2. Problem Statement

## 1.3. Contributions

## 1.4. Outline

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

# 2. Background

## 2.1. Tensor Networks

### 2.1.1. Tensor Notation and Diagrams

### 2.1.2. Tensor Contraction

### 2.1.3. Tensor Network Structures

## 2.2. GPU Architecture

### 2.2.1. Streaming Multiprocessor and Warp Execution

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.

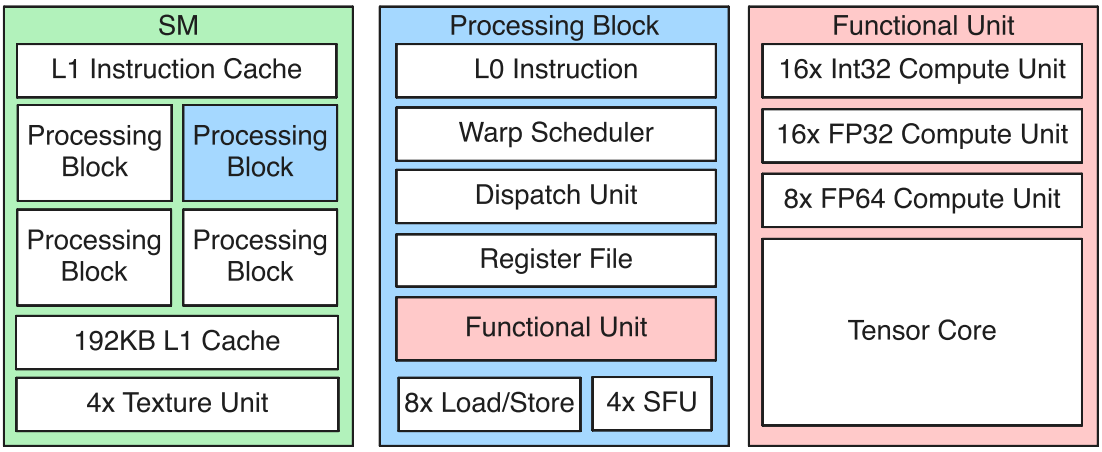


Figure 2.1.: Internal structure of an A100 Streaming Multiprocessor at three levels of detail. **Left:** the SM contains four processing blocks sharing an L1 data cache and shared memory. **Centre:** each processing block includes a warp scheduler, register file slice, execution units, LD/ST units, SFUs, 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 Figure 2.1, 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.
- One **texture unit** for read-only cached 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.

## 2.2.2. Floating-Point Formats and Precision Trade-offs

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

### IEEE 754 Binary Representation

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

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

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

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

where k is the number of exponent bits. For example for

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 |

Table 2.2.: Numerical properties of floating-point formats on the A100. Machine epsilon ($\varepsilon$) 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 | $\approx 6.1 \times 10^{-5}$ | $\approx 9.8 \times 10^{-4}$ |

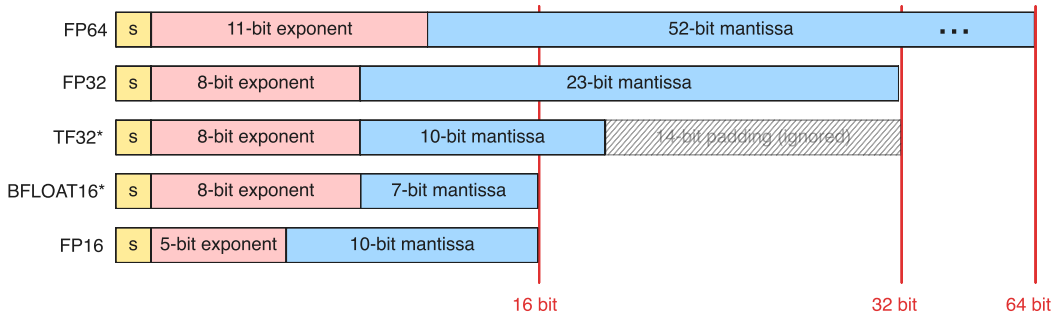


Figure 2.2.: Bit layout comparison of FP64, FP32, TF32, BF16, and FP16, with exponent and significand fields drawn to scale.

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

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

10 significand bits before performing the multiply-accumulate, with the accumulation itself carried out in full FP32 [NVI20]. The programmer does not need to convert data explicitly; the truncation is transparent.

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

### 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

$$\underbrace{0}_{\text{sign}} \mid \underbrace{01100}_{\text{exp}} \mid \underbrace{0100000000}_{\text{mantissa}}.15$$

**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 $2 \times 2$ 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 $5 \times 10^{-4}$ 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 [NVI20]; whether the same holds for a given scientific workload must be verified on a case-by-case basis, as we do in **??**.

**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

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) |

unit in the last place at magnitude $\approx 1$), 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.

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

### The Case for Single Precision in Tensor Network Computations

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

For tensor network algorithms, several factors favour FP32:

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

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

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

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

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

### 2.2.3. Memory Hierarchy

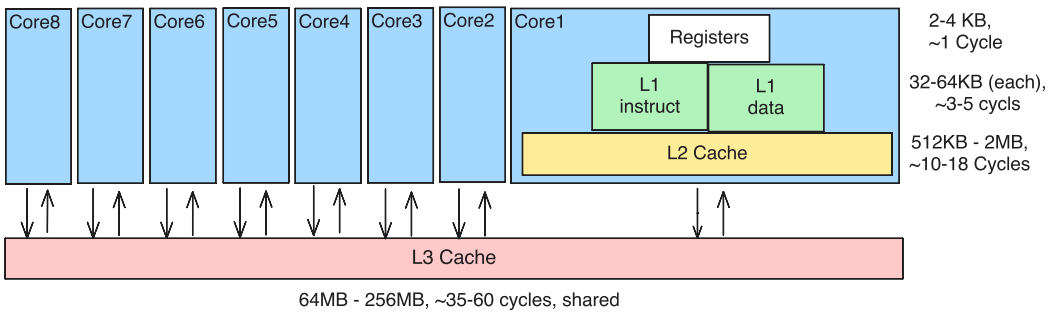


Figure 2.3.: Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM).

As shown in Figure 2.3, modern CPUs rely on a deep cache hierarchy to reduce effective memory latency.

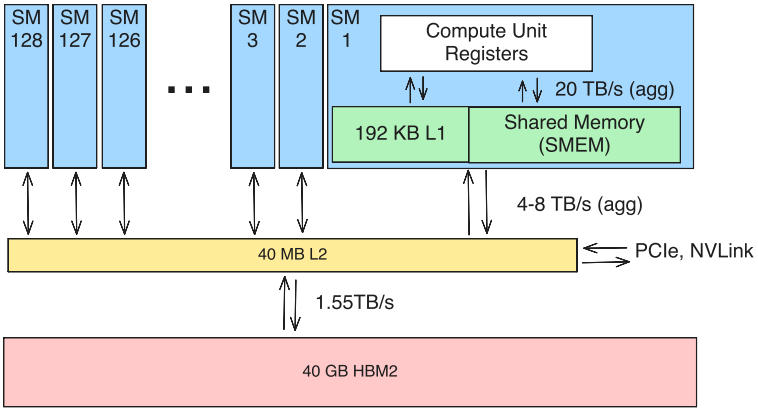


Figure 2.4.: Memory hierarchy of a modern CPU, showing the register file and cache levels (L1, L2, L3) before main memory (DRAM).

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

- **Registers** reside in the SM and offer single-cycle access. They are private to each thread and are the scarcest on-chip resource: the number of registers a kernel uses directly limits occupancy.
- **Shared memory** is also on-chip but visible to all threads in a block, making it the primary mechanism for explicit data reuse and inter-thread communication. Most high-performance kernels—tiled matrix multiplications, tensor contractions—stage data through shared memory to avoid repeated global loads.
- **L2 cache** is shared across all SMs and transparently caches global memory traffic. On the A100 it is 40 MB, modest relative to the core count, which reflects the throughput-oriented design.
- **HBM (global memory)** provides the bulk storage. The A100 uses HBM2e with 80 GB capacity and a peak bandwidth of 2039 GB/s (Table 2.4), achieved through a wide interface with thousands of parallel data lines. Access latency, however, is roughly 500× that of a register access (Table 2.7).

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

## 2.2.4. NVIDIA A100 Ampere Architecture

### Hardware Overview

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

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

| Property | Value |
|---|---|
| Streaming Multiprocessors (SMs) | 108 |
| CUDA cores per SM | 64 |
| Tensor cores per SM | 4 |
| Warp schedulers per SM | 4 |
| Maximum threads per SM | 2048 |
| Maximum warps per SM | 64 |
| Maximum blocks per SM | 32 |
| Register file size per SM (32-bit registers) | 65 536 |
| Shared memory per SM (KB) | 164 |
| L2 cache size (MB) | 40 |
| HBM memory capacity (GB) | 80 |
| Peak memory bandwidth (GB/s) | 2039 |
| Base clock frequency (GHz) | 1.41 |

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

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

#### Theoretical Peak Performance

#### Derived Performance Limits

#### Memory Latency

### 2.2.5. 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 are built from a chiplet design: each socket consists of a central I/O die (cIOD) connected to eight Core Complex Dies (CCDs). Each CCD contains two Core Complexes (CCXes), and each CCX contains four cores sharing a 16 MB L3 cache. The hierarchy is shown in Figure 2.5.

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

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

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

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

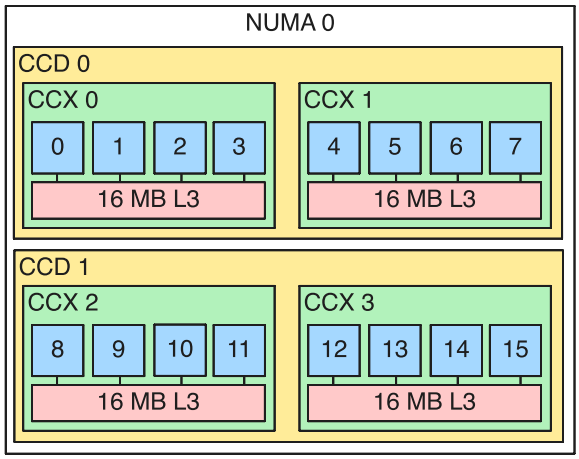


Figure 2.5.: NUMA and chiplet hierarchy of a single AMD EPYC 7742 socket. Each socket contains 8 CCDs grouped into 4 NUMA domains (NPS=4 configuration). Each CCD holds two CCXes, each with 4 cores and a private 16 MB L3 cache. All CCDs connect to a central I/O die that houses the memory controllers and 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. The latency hierarchy within a socket, from fastest to slowest, is:

1. **Intra-CCX:** cores sharing the same 16 MB L3 cache communicate through it at

low latency.

2. **Intra-CCD, cross-CCX:** cores in different CCXes of the same CCD must communicate through the I/O die, as each CCX has its own independent L3.

3. **Cross-CCD, same socket:** traffic between CCDs in different NUMA domains routes through the I/O die's internal Infinity Fabric switches.

4. **Cross-socket:** traffic between the two sockets traverses AMD's Infinity Fabric inter-socket links (xGMI), incurring the highest latency.

### 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.

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

Table 2.8.: 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 2.6.

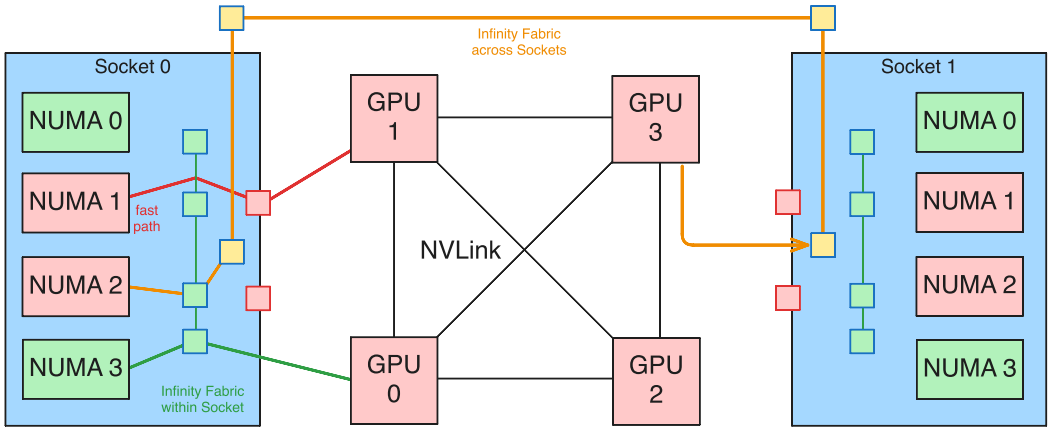


Figure 2.6.: Compute node topology showing the four A100 GPUs (centre) connected via NVLink 3.0, with the two CPU sockets on either side. Three representative host–device paths are highlighted: a fast local PCIe path (GPU 1 ↔ NUMA 1), an intra-socket Infinity Fabric path (GPU 0 ↔ NUMA 3), and a cross-socket Infinity Fabric path (GPU 3 ↔ 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.

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.

## 2.3. CUDA Programming Model

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

### 2.3.1. Thread Hierarchy and Kernel Launch

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

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

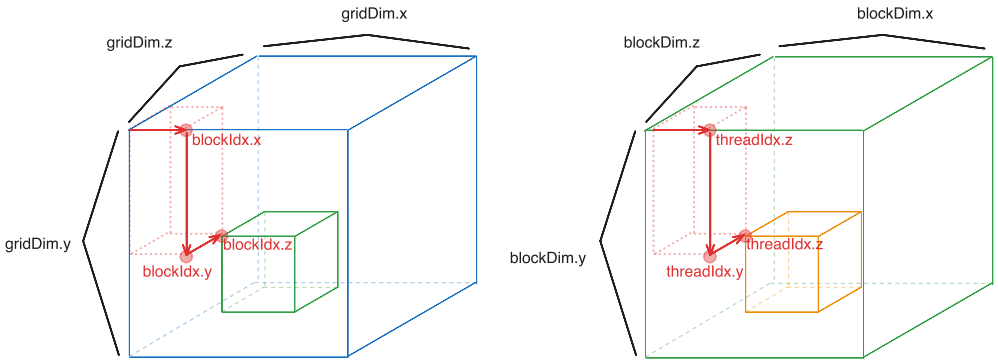


Figure 2.7.: CUDA execution hierarchy showing the mapping of grids to thread blocks and threads. Threads are organised in up to three dimensions and identified using the built-in variables `threadIdx` and `blockIdx`.

The hierarchical organisation of grids, thread blocks, and threads is illustrated in Figure 2.7. Within each block, threads are further grouped into *warps* of 32 threads that execute instructions in lock-step on the SM's SIMT (Single Instruction, Multiple Thread) execution units.

### Kernel Syntax and Launch Configuration

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

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

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

The global thread index is computed on line 2 as `blockIdx.x * blockDim.x + threadIdx.x`. This is the standard one-dimensional addressing pattern: `blockIdx.x` selects which block the thread belongs to, `blockDim.x` gives the number of threads per block, and `threadIdx.x` is the thread's position within its block. The bounds check on line 3 is necessary because the total number of threads launched (blocks $\times$ 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:

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

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

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

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

### Block Size Selection and Hardware Constraints

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

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

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

### Linearisation of Multidimensional Indices

GPU memory is addressed linearly, so multidimensional arrays must be mapped to one-dimensional offsets. For a tensor stored in row-major order, the linear index of

element $(i, j)$ in an $M \times N$ matrix is

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

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

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

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

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

## 2.3.2. Shared Memory and Synchronisation

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

### Static and Dynamic Allocation

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

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

Listing 2.3: Static shared memory allocation for a tile.

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

```
extern __shared__ float smem[];

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

Listing 2.4: Dynamic shared memory allocation.

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

### Tiled Matrix Multiplication

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

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

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

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

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

The key points of this kernel are:

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

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

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

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

## 2.3.3. Memory Coalescing and Bank Conflicts

### Global Memory Coalescing

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

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

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

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

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

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

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

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

### Shared Memory Bank Conflicts

Shared memory is divided into 32 *banks*, each 4 bytes wide, interleaved in a round-robin fashion: word $k$ resides in bank $k$ 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
```

```
4
5  // Fix: pad the leading dimension by one
6  __shared__ float tile[32][33];       // stride = 33, conflicts eliminated
7  float val = tile[threadIdx.x][col];
```

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

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

### 2.3.4. Performance Profiling with Nsight Compute

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

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

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

- **Achieved occupancy:** the ratio of active warps per cycle to the maximum the SM can support, indicating how effectively the kernel hides memory latency through parallelism.
- **Memory throughput:** the achieved bandwidth to each level of the memory hierarchy (HBM, L2, shared memory), compared against the theoretical peak. A kernel achieving close to peak HBM bandwidth is memory-bound.
- **Compute throughput:** the achieved FLOP/s relative to the peak, broken down by instruction type. A kernel well below peak compute with high memory throughput is bandwidth-limited.
- **Warp stall reasons:** a breakdown of why warps were stalled (e.g. waiting for memory, waiting at a barrier, instruction dependencies), which directly guides optimisation.
- **Shared memory bank conflicts:** the number of replayed shared memory accesses due to bank conflicts, indicating whether padding or access pattern changes are needed.
- **L2 cache hit rate:** the fraction of global memory requests served by the L2 cache, relevant for understanding data reuse across thread blocks.

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

### 2.3.5. 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 [NVI20]) 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 2.2.1.

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 $\mu$s 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.

#### 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 [NVI25], 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 2.4.1, and the performance impact of fusion is quantified experimentally in Section 5.2.

## 2.4. Related Work

### 2.4.1. cuBLAS, cuBLASDx, and cuTENSOR

### 2.4.2. ChASE Eigensolver

### 2.4.3. Existing GPU Tensor Network Implementations

# 3. Design and Methodology

## 3.1. Target Kernels

## 3.2. Algorithmic Approach

## 3.3. Data Layout and Memory Strategy

## 3.4. Baseline Selection

# 4. Implementation

## 4.1. Integration and Build System

## 4.2. Kernel Design

## 4.3. Occupancy and Launch Configuration

## 4.4. Shared Memory Tiling

# 5. Results

## 5.1. Experimental Setup

## 5.2. Kernel Launch Overhead Benchmark

As discussed in Section 2.3.5, 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 the total runtime. This section 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 [NVI25], 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.

### 5.2.3. Discussion

Two observations follow from the data. First, the total time for separate launches is nearly constant across all matrix sizes, varying only from 8022 $\mu$s at $n = 1$ to 9752 $\mu$s at $n = 32$. This confirms that the per-launch overhead of approximately 8 $\mu$s 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.

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 ($\mu$s) | Fused ($\mu$s) | 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 |

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, placing them squarely 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

## 5.4. Scaling Behaviour

## 5.5. Profiling Analysis

## 5.6. Comparison with cuBLAS and cuTENSOR

## 5.7. Discussion

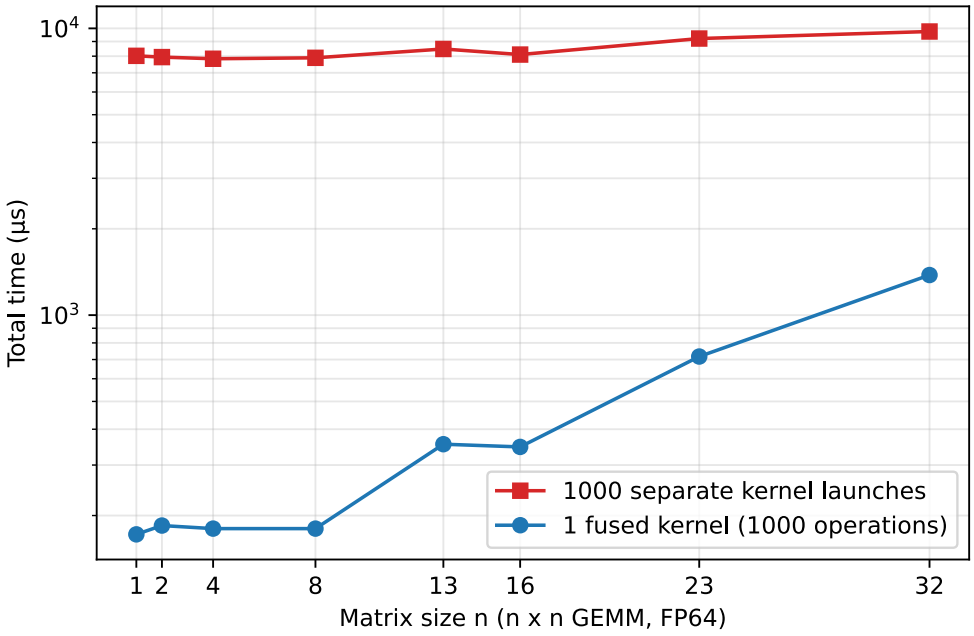


Figure 5.1.: Total time for 1000 GEMM operations as a function of matrix size, comparing 1000 separate kernel launches (red) against a single fused kernel (blue). The near-constant red curve below $n = 16$ confirms that launch overhead dominates over actual computation for small matrices.

# 6. Conclusion

## 6.1. Summary

## 6.2. Limitations

## 6.3. Future Work

# Bibliography


[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.

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

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

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

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

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

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

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

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

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

[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-sys

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

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

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

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

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

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

# 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 HBM2e |
| 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 18, 2026 Daniel Sinkin