高性能计算基准测试深度研究报告：基于A100 GPU与Zen 4架构的2D泊松方程求解器性能分析

1. 引言

1.1 研究背景与科学意义

在计算物理与工程科学的广阔版图上，偏微分方程（Partial Differential Equations, PDEs）构成了描述自然界连续物理现象的数学基石。其中，泊松方程（Poisson Equation）$\nabla^{2}p=b$ 作为一种典型的椭圆型偏微分方程，其地位尤为核心。它不仅主导着静电学中电势与电荷分布的关系（如库仑定律的微分形式），还在牛顿万有引力理论中描述引力势与质量密度的联系[1]。更为关键的是，在计算流体动力学（CFD）领域，特别是在求解不可压缩纳维-斯托克斯方程（Navier-Stokes Equations）的投影法（Projection Method）中，压力泊松方程的求解往往占据了整个时间步迭代中最大的计算开销[3]。

随着科学模拟的规模不断扩大，从微米尺度的微流控芯片模拟到星系尺度的引力坍缩模拟，网格分辨率的提升导致计算量呈几何级数增长。传统的串行计算模式已无法满足现代科研对“求解时间（Time-to-Solution）”的迫切需求。因此，利用现代异构计算架构——特别是图形处理单元（GPU）和众核中央处理器（CPU）——来加速泊松求解器，已成为高性能计算（HPC）领域的关键研究课题。

本研究报告基于C3SE（Chalmers Centre for Computational Science and Engineering）Vera计算集群的实验数据，对2D泊松方程求解器在不同硬件架构与软件实现下的性能进行了详尽的基准测试与分析。实验不仅涵盖了从单核CPU到多核并行，再到GPU加速的跨越，更深入探讨了不同软件栈（Python原生、Numba、CuPy、PyTorch）及底层存储优化技术（Shared Memory Tiling）对性能的决定性影响[5]。

1.2 数学模型与离散化方法

本实验所求解的核心方程为二维笛卡尔坐标系下的泊松方程：

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = b(x,y)$$

为了在计算机上进行数值求解，必须将连续的物理域离散化为有限的网格点。采用有限差分法（Finite Difference Method, FDM），利用泰勒级数展开（Taylor Series Expansion）将二阶偏导数近似为中心差分格式 6。对于均匀网格（网格间距为 Δx 和 Δy），点 (i, j) 处的拉普拉斯算子可近似为：

$$ \nabla^2 p_{i,j} \approx \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\Delta x^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{\Delta y^2} $$

通过雅可比迭代法（Jacobi Iteration），我们可以得到显式的更新公式。这是一种经典的定常迭代

法，其计算过程具有高度的数据并行性，非常适合在向量处理器和GPU上实现。第 $k+1$ 次迭代的值 $p_{i,j}^{new}$ 仅依赖于第 k 次迭代的邻域值 7：
$$ p_{i,j}^{new} = \frac{\Delta y^{2}(p_{i+1,j} + p_{i-1,j}) + \Delta x^{2}(p_{i,j+1} + p_{i,j-1}) - b_{i,j}\Delta x^{2}\Delta y^{2}}{2(\Delta x^{2} + \Delta y^{2})} $$
这种计算模式被称为“五点模板”（5-point stencil），因为它涉及中心点及其东、西、南、北四个相邻点的数值 [6]。虽然雅可比方法的收敛速度通常不如高斯-赛德尔（Gauss-Seidel）或共轭梯度法（Conjugate Gradient），但其“新值完全依赖旧值”的特性避免了读写依赖冲突（Race Conditions），使其成为评估并行硬件峰值内存带宽性能的理想基准 [9]。

1.3 算术强度与内存墙挑战

理解本实验结果的一个关键理论框架是“算术强度”（Arithmetic Intensity），即每访问一个字节的内存所执行的浮点运算次数（FLOPs/Byte）。对于上述2D五点模板计算：

- 浮点运算：包含加法、乘法和减法，大约需要 7-10 次 FLOPs。
- 内存访问：在最朴素的实现中，需要读取5个浮点数（4个邻居+1个源项）并写入1个浮点数。假设使用单精度浮点数（4 Bytes），则总内存传输量为 $6 \times 4 = 24$ Bytes。
- 算术强度：$\approx 10 / 24 \approx 0.42$ FLOPs/Byte [11]。

根据 Roofline 模型 [13]，现代计算硬件（如A100 GPU或Zen 4 CPU）的计算能力极其强大，通常需要极高的算术强度（例如 >100 FLOPs/Byte）才能使计算单元满载。泊松求解器的极低算术强度意味着该算法是典型的**内存受限（Memory-Bound）**问题。性能的瓶颈不在于处理器的计算速度，而在于内存子系统将数据搬运到处理器的速度。这一理论背景是后续分析CPU核心扩展失效及GPU共享内存优化生效的基础。

---

2. 实验环境与硬件架构深度解析

为了准确解读基准测试数据，必须深入剖析实验所依托的硬件平台——C3SE Vera集群的计算节点配置。实验涉及两种截然不同的计算架构：基于x86-64指令集的AMD EPYC CPU和基于大规模并行架构的NVIDIA A100 GPU。

2.1 CPU架构：AMD EPYC 9354 "Genoa" (Zen 4)

虽然实验报告摘要中提及“AMD EPYC 7000 Series (Zen 4)” [5]，但这在命名规范上存在矛盾。AMD EPYC 7000系列对应的是Zen, Zen 2 (Rome) 或 Zen 3 (Milan) 架构，而 **Zen 4** 架构对应的是 **EPYC 9004**系列。结合C3SE官方文档确认，Vera集群配备的是 **AMD EPYC 9354** 处理器 [14]。

- 核心微架构：Zen 4 微架构采用了台积电5nm工艺制造，相比前代带来了显著的IPC（每时钟周期指令数）提升和更高的时钟频率（基频3.25GHz，加速频率3.8GHz）[16]。每个核心都配备了AVX-512向量指令集，理论上单核浮点性能极强。
- **Chiplet**设计与**I/O**瓶颈：EPYC 9354拥有32个物理核心（64线程）。这些核心分布在多个CCD（Core Complex Die）上，通过Infinity Fabric互连总线与中央I/O Die通信。
- 内存子系统：这是本实验中CPU性能分析的关键。EPYC 9004系列支持 12通道 DDR5 内存，

理论峰值带宽计算如下：
$$ 12 \text{ channels} \times 8 \text{ Bytes/transfer} \times 4800 \text{ MT/s} \approx 460.8 \text{ GB/s} $$
尽管460.8 GB/s的带宽在CPU领域已属顶尖 16，但在面对64个线程同时发起的密集访存请求（Stencil Computation）时，这一通道仍极易饱和。当每个核心试图全速运行内存密集型代码时，平均每个核心可用的带宽仅为 $460.8 / 64 \approx 7.2 \text{ GB/s}$。这甚至低于单根DDR4内存条的带宽，解释了为何多核扩展会遇到瓶颈。

2.2 GPU架构：NVIDIA A100-SXM4

实验使用了一块 NVIDIA A100-SXM4-40GB GPU，这是专为科学计算设计的顶级加速器，基于Ampere架构 [5]。

- 高带宽内存（**HBM2e**）：与CPU的DDR5不同，A100采用了HBM2e显存，通过硅中介层（Silicon Interposer）直接与GPU芯片封装在一起，实现了超宽的位宽（5120-bit）。其理论内存带宽高达 **1,555 GB/s**（40GB版本）甚至 **2,039 GB/s**（80GB版本）[19]。
  - 对比：A100的内存带宽是EPYC 9354的 **3.3**倍到**4.4**倍。对于内存受限的泊松求解器，这直接奠定了GPU性能碾压CPU的基础。
- 流式多处理器（**SM**）与共享内存：A100包含108个SM，每个SM配备了192KB的L1缓存/共享内存组合存储 [21]。共享内存（Shared Memory）是用户可编程的片上高速缓存（On-chip Scratchpad），其带宽高达数十TB/s，且延迟极低。在Stencil计算中，有效利用共享内存可以大幅减少对HBM显存的访问压力，是性能优化的核心手段 [22]。
- **SXM4**互连：SXM4板型支持NVLink高速互连，虽然本实验主要关注单卡性能，但SXM4版本通常拥有比PCIe版本更高的TDP（400W vs 250W），允许更持久的高频运行 [24]。

---

3. 技术实现与软件栈分析

为了全面评估不同开发模式的效率与性能，实验采用了五种不同的技术路径来实现泊松求解器。这些技术涵盖了从纯Python的解释执行到底层CUDA内核的编译执行。

3.1 CPU基准实现

- **CPU Single-Core (Python):** 作为最基础的对照组，使用纯Python编写。Python的解释器机制（CPython）导致循环执行效率极低，且全局解释器锁（GIL）限制了多线程并行。这种实现主要用于验证算法正确性和展示未经优化的性能下限 [5]。
- **CPU Multi-Core (Numba Parallel):** 利用 numba.jit(parallel=True) 装饰器。Numba 是一个即时编译器（JIT），它利用LLVM将Python字节码编译为机器码。
  - 技术机制：通过 numba.prange，它能自动将外层循环并行化，类似于C语言中的OpenMP #pragma omp parallel for。
  - 优势与局限：它消除了Python解释器的循环开销，并利用了多核CPU。然而，由于它直接操作平坦的内存数组，并不包含针对缓存局部性（Cache Blocking）的高级优化，因此极易受到内存带宽的限制 [5]。

3.2 GPU高层库实现 (High-Level Libraries)

这一类实现代表了现代数据科学家最常用的开发模式：利用现成的库，以极低的代码量换取GPU加速。

- **CuPy:** 专为GPU设计的NumPy兼容库。
  - 实现方式：依靠内核融合（Kernel Fusion）或预编译的CUDA内核。对于 p[1:-1, 1:-1] =... 这样的切片操作，CuPy会在后台调用CUDA内核。
  - 性能特征：虽然CuPy极大地简化了开发，但在处理复杂的逐元素（Element-wise）运算组合时，如果未能有效触发内核融合，可能会导致多次内核启动（Kernel Launch）和显存读写，产生额外开销 [26]。
- **PyTorch:** 原本为深度学习设计的框架，但同样适用于张量计算。
  - 技术优势：PyTorch 拥有非常成熟的 JIT 编译器（TorchScript）和急切执行（Eager Execution）优化器。对于Stencil计算中的张量加减乘除，PyTorch 往往能更智能地融合算子，减少显存访问次数。此外，PyTorch 的内存分配器（Caching Allocator）在处理临时张量时效率极高 [28]。

3.3 GPU底层优化实现 (Numba CUDA)

这一类实现利用 numba.cuda 接口，允许开发者在Python中直接编写类似CUDA C的内核代码，显式控制线程块（Block）和网格（Grid）。

- **Numba Basic (Global Memory):**
  - 实现逻辑：每个GPU线程对应网格中的一个点。线程直接从全局内存（Global Memory）读取邻居节点的值。
  - 缺陷：存在严重的重复读取。例如，计算 $p_{i,j}$ 需要读取 $p_{i,j+1}$，而计算 $p_{i,j+1}$ 时又需要读取一次 $p_{i,j+1}$。在五点模板中，每个数据平均被读取5次（1次作为中心，4次作为邻居），导致显存带宽的极大浪费 [23]。
- **Numba Shared Memory (Tiling):**
  - 核心技术：分块（**Tiling**）与光环交换（**Halo Exchange**）。
  - 实现细节：将大网格切割成若干个小块（Tile），例如 16×16 或 32×32。每个线程块（Thread Block）负责计算一个Tile。
  - 步骤：
    1. 协同加载（**Cooperative Loading**）：线程块中的所有线程协作，将当前Tile及其周边的“光环”数据（Halo/Ghost Cells）从慢速的全局内存加载到高速的共享内存（Shared Memory）中 [30]。
    2. 同步（**Synchronization**）：调用 cuda.syncthreads() 确保所有数据加载完毕。
    3. 高速计算：线程直接从共享内存读取邻居数据进行差分计算。由于共享内存位于片上，延迟极低，且避免了对显存的重复访问。
  - 技术门槛：这是最复杂的实现方式，需要处理复杂的索引映射、边界条件检查和不同内存层级之间的数据搬运，但能带来最高的性能收益 [5]。

---

4. 实验结果深度分析与讨论

基于实验报告提供的数据（Table 1, Figure 1-3 [5]），我们对各项技术的表现进行深入剖析。

4.1 CPU扩展性的崩溃：内存带宽饱和现象

实验数据揭示了一个反直觉的现象：在处理 2000×2000 的大网格时，**64**核的执行时间（**512.37s**）竟然比**32**核（**467.61s**）更慢。

网格尺寸	**CPU (32-Core)**	**CPU (64-Core)**	性能变化
800×800	319.35s	396.46s	变慢 **24%**
2000×2000	467.61s	512.37s	变慢 **9.5%**

- 原因分析：这是一个教科书式的**内存带宽饱和（Memory Bandwidth Saturation）**案例。
  - Stencil计算的算术强度极低（< 0.5 FLOPs/Byte）。32个Zen 4核心全速运行时，其数据吞吐需求已经接近或超过了EPYC 9354的12通道DDR5内存带宽上限（约460 GB/s）[31]。
  - 当开启64个线程时，实际上并没有更多的内存带宽可供使用。相反，过多的线程导致了严重的资源争抢。
  - 争抢开销：更多的线程意味着更频繁的上下文切换、更严重的L3缓存抖动（Cache Thrashing）以及内存控制器的排队延迟。当带宽成为硬约束时，增加计算单元（核心）不仅无效，反而因管理开销（Overhead）拖累了整体性能。
- 结论：对于内存受限型应用，盲目增加CPU核心数存在边际效用递减甚至负增长的临界点。在Vera集群的该节点上，32核可能是该算法的性能甜点（Sweet Spot）。

4.2 GPU与CPU的代际差距

在 2000×2000 网格下，单核CPU耗时 **9069.63**秒（约2.5小时），而优化后的GPU仅需 **12.00**秒。

- 加速比：$9069.63 / 12.00 \approx \mathbf{755\times}$。
- 多核对比：即便是表现最好的32核CPU（467.61s），也被GPU超越了约 **39**倍。
- 分析：这种巨大的差距主要归功于A100的HBM2e显存。其2039 GB/s的带宽是CPU DDR5带宽（460 GB/s）的4.4倍。考虑到CPU实际应用中很难跑满理论带宽（通常利用率在60%-80%），而GPU的流式架构更容易接近带宽峰值，实际的有效带宽差距可能更大。此外，A100能够同时调度成千上万个轻量级线程，通过零开销的线程切换有效隐藏了内存访问延迟（Latency Hiding），这是依靠乱序执行（Out-of-Order Execution）和深层缓存的CPU所无法比拟的 [20]。

4.3 高层库之争：PyTorch 为何优于 CuPy？

在所有测试规模下，PyTorch的表现均优于CuPy。在 2000×2000 网格下，PyTorch（

33.85s）比 CuPy（55.22s）快了约 **63%**。

网格尺寸	**CuPy (s)**	**PyTorch (s)**	**Numba Shared (s)**
2000×2000	55.22	33.85	12.00

- 深入剖析：
  - 通常认为CuPy作为轻量级库，开销应小于庞大的PyTorch。但在逐元素操作（Element-wise Operations）密集的Stencil计算中，PyTorch展现了优势。
  - 内核融合（**Kernel Fusion**）：Stencil更新公式涉及多个加法、乘法和减法。如果逐个执行（如CuPy的默认行为可能倾向于这样做），会产生大量的临时张量读写（Global Memory Traffic）。PyTorch 的 JIT 编译器或底层的 TensorIterator 机制能够更积极地将这些操作融合为一个单一的CUDA内核，使得数据只需从显存读取一次，计算完毕后写回，大幅减少了显存带宽占用 [28]。
  - 内存分配器：迭代求解器需要频繁更新数组。PyTorch 的 Caching Allocator 经过了深度优化，能够极快地复用显存块，减少了 cudaMalloc 和 cudaFree 带来的同步开销 [34]。

4.4 共享内存优化的决定性作用

实验结果中最引人注目的数据是 Numba Shared Memory 实现（12.00s）相比于 PyTorch（33.85s）和 CuPy（55.22s）的巨大优势。

- 性能提升：Numba Shared 比 PyTorch 快 **2.8**倍，比 CuPy 快 **4.6**倍。
- 技术原理：
  - 在 PyTorch/CuPy 的实现中，虽然可能利用了内核融合，但底层的通用内核（General Kernel）通常无法针对特定的Stencil模式进行极致的**数据复用（Data Reuse）**优化。它们仍然主要依赖L2缓存来缓解带宽压力。
  - Numba Shared Memory 实现显式地使用了 GPU 的 L1/Shared Memory。在加载 16×16 的数据块时，每个数据元素被协作线程读取到片上共享内存中。
  - 带宽倍增效应：一旦数据进入共享内存，后续的邻居访问（上、下、左、右）就完全发生在片上。共享内存的聚合带宽（Aggregate Bandwidth）在A100上可达 **19 TB/s** 级别，远超HBM2e的2 TB/s [21]。
  - 消除冗余：这种手动分块技术（Tiling）实际上将原本需要从显存重复读取多次的数据，变成了“读取一次，多次使用”，从而人为地提高了算法的算术强度，突破了内存墙的限制。

---

5. 结论与建议

5.1 核心结论

1. 架构决定性能上限：对于泊松求解器这类内存受限算法，GPU 凭借 HBM2e 带来的超高带宽，相对于顶级服务器 CPU 具有压倒性的性能优势（~40-700倍加速）。

2. **CPU**多核陷阱：在高带宽需求的计算任务中，增加 CPU 核心数并不总是带来性能提升。当内存通道饱和时，过多的线程不仅无益，反而有害。在 C3SE Vera 这样的节点上，应谨慎配置 MPI 进程或 OpenMP 线程数，避免 64 核全开导致的性能倒退。
3. 软件栈层级效应：
   - 高层库（**PyTorch**）：提供了极佳的“开箱即用”性能，适合快速原型开发。其内部优化（算子融合）使其在未做额外工作的情况下明显优于 CuPy。
   - 底层优化（**Numba Shared**）：展示了“极致性能”的代价与收益。通过增加代码复杂度和开发时间，显式管理 GPU 内存层级，可以获得比高层库再快 3 倍的性能。

5.2 针对演示（**Presentation**）的建议

在制作 Presentation 时，建议遵循以下叙事逻辑：

1. 从物理到数学：简述泊松方程的重要性及五点差分模板的稀疏性、局部性。
2. 从 **CPU** 到 **GPU** 的跨越：展示 Figure 1 [5]，用对数坐标轴突显 CPU 与 GPU 的数量级差异，强调“GPU 是解决大规模网格的唯一可行方案”。
3. 揭示**“内存墙”**：重点展示 CPU 32核 vs 64核 的对比图表 [5]。这是一个极佳的教育时刻，用于解释带宽饱和概念。
4. 剖析 **GPU** 优化：对比 PyTorch、CuPy 和 Numba Shared 的性能。使用图解（如 [30] 中的 Tiling 示意图 ）来解释为什么 Shared Memory 能带来额外的 3 倍加速——即“通过软件技巧弥补硬件瓶颈”。
5. 总结：高性能计算不仅是硬件的堆砌，更是对硬件架构特性的深刻理解与软件算法的精准适配。

参考文献引用说明：

- [5] 实验原始报告与代码列表。
- [1] 泊松方程背景与数值方法。
- [14] AMD EPYC 硬件规格与内存带宽分析。
- [19] NVIDIA A100 架构与显存特性。
- [31] 多核CPU带宽饱和理论。
- [23] GPU共享内存分块技术与Halo Exchange优化。
- [28] PyTorch 与 CuPy 性能差异分析。

Works cited

1. Poisson's equation - Wikipedia, accessed December 8, 2025, https://en.wikipedia.org/wiki/Poisson%27s_equation
2. Poisson's Equation in Electrostatics, accessed December 8, 2025, https://dasher.wustl.edu/chem430/readings/poisson-equation.pdf
3. 2D FD Poisson Example — ExaStencils documentation, accessed December 8, 2025, https://www.exastencils.fau.de/sphinx/tutorials/ExaStencils/2D_FD_Poisson_fromL4.html

4. 1 Poisson's equation, accessed December 8, 2025, https://acme.byu.edu/00000179-d3f1-d7a6-a5fb-ffff6a1c0001/poissonequation-1-pdf
5. TRA.pdf
6. Five-point stencil - Wikipedia, accessed December 8, 2025, https://en.wikipedia.org/wiki/Five-point_stencil
7. 8.1 Jacobi method - Numerical Analysis II - Fiveable, accessed December 8, 2025, https://fiveable.me/numerical-analysis-ii/unit-8/jacobi-method/study-guide/x5EgnOuloZs4JRLk
8. Jacobi method - Wikipedia, accessed December 8, 2025, https://en.wikipedia.org/wiki/Jacobi_method
9. Jacobi Solver with HIP and OpenMP offloading - ROCm™ Blogs - AMD, accessed December 8, 2025, https://rocm.blogs.amd.com/high-performance-computing/jacobi/README.html
10. USING JACOBI METHOD TO SOLVE THE TWO-EQUATION TURBULENCE MODEL FOR PARALLELIZATION ON GPU COMPUTING SYSTEM, accessed December 8, 2025, https://www.cmff.hu/papers25/CMFF25_Final_Paper_PDF_98.pdf
11. 2D Jacobi stencil pseudo-code and data access pattern. - ResearchGate, accessed December 8, 2025, https://www.researchgate.net/figure/D-Jacobi-stencil-pseudo-code-and-data-access-pattern_fig2_357417576
12. Estimating the upper bound on arithmetic intensity for a stencil algorithm, accessed December 8, 2025, https://impact-workshop.org/papers/eubaisa.pdf
13. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures - UC Berkeley EECS, accessed December 8, 2025, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.pdf
14. Vera @ C3SE - SUPR - NAISS, accessed December 8, 2025, https://supr.naiss.se/resource/vera/
15. Vera - C3SE, accessed December 8, 2025, https://www.c3se.chalmers.se/about/Vera/
16. AMD EPYC 9354 Specs - CPU Database - TechPowerUp, accessed December 8, 2025, https://www.techpowerup.com/cpu-specs/epyc-9354.c2923
17. AMD EPYC 9004 (4th Gen) 9354 Dotriaconta-core (32 Core) 3.25 GHz Processor - 256 MB L3 Cache - 64-bit Processing - 3.80 GHz Overclocking Speed - Socket SP5 No Graphics - 280 W - 64 Threads - Newegg.com, accessed December 8, 2025, https://www.newegg.com/amd-epyc-9354-socket-sp5/p/1WK-0184-00068
18. Optimizing Linux for AMD EPYC™ 9004 Series Processors with SUSE Linux Enterprise 15 SP4, accessed December 8, 2025, https://documentation.suse.com/sbp/tuning-performance/html/SBP-AMD-EPYC-4-SLES15SP4/index.html
19. Nvidia A100 SXM4 Tensor CORE GPU Accelerator 80GB HBM2 Memory Bandwidth 2039GB/S PCI-E 4.0 X16 General Purpose Graphics Processing Unit Gpgpu - IT Creations, accessed December 8, 2025, https://www.itcreations.com/product/139047

20. NVIDIA A100 | Tensor Core GPU, accessed December 8, 2025, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf
21. Revisiting Temporal Blocking Stencil Optimizations - OSTI.GOV, accessed December 8, 2025, https://www.osti.gov/servlets/purl/1994670
22. Using Shared Memory in CUDA C/C++ | NVIDIA Technical Blog, accessed December 8, 2025, https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/
23. CUDA – Stencils, Shared Memory and Reduction Operations, accessed December 8, 2025, https://indico.fysik.su.se/event/6743/contributions/10338/attachments/4175/4801/4.CUDA-StencilsSharedMemory-Markidis.pdf
24. NVIDIA A100 PCIe vs SXM: Comprehensive Performance Comparison - Hyperstack, accessed December 8, 2025, https://www.hyperstack.cloud/technical-resources/performance-benchmarks/nvidia-a100-pcie-vs-nvidia-a100-sxm-a-comprehensive-comparison
25. Exploring Numba and CuPy for GPU-Accelerated Monte Carlo Radiation Transport - MDPI, accessed December 8, 2025, https://www.mdpi.com/2079-3197/12/3/61
26. CuPy and Numba on the GPU – Lesson Title - The Carpentries Incubator, accessed December 8, 2025, https://carpentries-incubator.github.io/gpu-speedups/01_CuPy_and_Numba_on_the_GPU/index.html
27. Plug-and-Play CuPy on ROCm: Data Analytics Acceleration Made Simple, accessed December 8, 2025, https://rocm.blogs.amd.com/artificial-intelligence/cupy-v13/README.html
28. Speed Up PyTorch With Custom Kernels. But It Gets Progressively Darker - Dss Solutions, accessed December 8, 2025, https://dsssolutions.com/2025/01/09/speed-up-pytorch-with-custom-kernels-but-it-gets-progressively-darker/
29. RFC: The State of Custom CUDA extensions in PyTorch · Issue #152032 - GitHub, accessed December 8, 2025, https://github.com/pytorch/pytorch/issues/152032
30. CUDA Studylog 3 - Tiling and Shared Memory for Matrix Multiplication Optimization, accessed December 8, 2025, https://gitlostmurali.com/machine-learning/data-science/cuda-matmul-2
31. Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs - ResearchGate, accessed December 8, 2025, https://www.researchgate.net/publication/220727817_Realistic_Workload_Scheduling_Policies_for_Taming_the_Memory_Bandwidth_Bottleneck_of_SMPs
32. Estimating your memory bandwidth - Daniel Lemire's blog, accessed December 8, 2025, https://lemire.me/blog/2024/01/13/estimating-your-memory-bandwidth/
33. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling - Ethz, accessed December 8, 2025, https://people.inf.ethz.ch/omutlu/pub/large-gpu-warps-TR-HPS-2010-006.pdf
34. Performance measurements - `cp.matmul` slower than `torch.matmul` · Issue

#5075 · cupy/cupy - GitHub, accessed December 8, 2025, https://github.com/cupy/cupy/issues/5075
35. [D] Why is PyTorch as fast as (and sometimes faster than) TensorFlow? - Reddit, accessed December 8, 2025, https://www.reddit.com/r/MachineLearning/comments/cvcbu6/d_why_is_pytorch_as_fast_as_and_sometimes_faster/
36. CUDA Memory Architecture, accessed December 8, 2025, https://ajdillhoff.github.io/notes/cuda_memory_architecture/
37. Stencil - Khushi Agrawal, accessed December 8, 2025, https://khushi-411.github.io/stencil/