arXiv:2305.01024v1 [cs.DC] 1 May 2023

GPU 上在线容错高性能 GEMM 剖析

Shixun Wu* Yujia Zhai* Jinyang Liu Jiajun Huang 加利福尼亚大学，加利福尼亚大学，加利福尼亚大学，加利福尼亚大学，河滨分校河滨分校河滨分校河滨分校美国加利福尼亚州河滨市美国加利福尼亚州河滨市美国加利福尼亚州河滨市美国加利福尼亚州河滨市swu264@ucr.edu yzhai015@ucr.edu jliu447@ucr.edu jhuan380@ucr.edu

Zizhe Jian
加利福尼亚大学河滨分校
美国加利福尼亚州河滨市
zjian106@ucr.edu

Bryan M. Wong 加
利福尼亚大学河滨分校 美
国加利福尼亚州河滨市
bryan.wong@ucr.edu

Zizhong Chen 加
利福尼亚大学河滨分校
美国加利福尼亚州河滨
市 chen@cs.ucr.edu

摘要

通用矩阵乘法（GEMM）是机器学习和科学计算等多种应用中的关键算法，因为高效的 GEMM 实现对于这些计算的性能至关重要。研究人员常常致力于借助大型计算平台提升性能，但系统规模扩大也会引发对硬件和软件可靠性的担忧。本文介绍了一种高性能基于 GPU 的通用矩阵乘法设计，该设计集成了基于算法的容错方案，可在线检测并纠正计算单元中的静默数据损坏。我们探索了线程级、线程束级和线程块级的 GEMM 容错设计，并提供了一种基准 GEMM 实现，其性能可与最先进的闭源 cuBLAS GEMM 媲美，甚至更快。我们提出了一种内核融合策略，以在原始 GEMM 计算期间隐藏并减轻容错所导致的内存延迟。为支持各种输入矩阵形状并降低开发成本，我们提出了一种基于模板的方法，可自动生成容错和非容错 GEMM 实现的代码。我们在 NVIDIA Tesla T4 和 A100 服务器 GPU 上评估了这项工作。实验结果表明，我们的基准 GEMM 性能与闭源 cuBLAS 相当或更优。与此前最先进的非融合容错 GEMM 相比，我们的最优融合策略平均实现了 39.04% 的加速比。此外，即使每分钟注入数百个错误，我们的容错 GEMM 相较于 cuBLAS 也仅产生极小开销（平均 8.89% ）。对于不规则形状输入，代码生成器生成的内核展现出显著的加速比分别达到容错和非容错通用矩阵乘法的 160% ~ 183.5%和 148.55% ~ 165.12%，性能最高比 cuBLAS 快 41.40%。

CCS 概念

• **计算方法** → **大规模并行算法**。

关键词

通用矩阵乘法、图形处理器、性能优化、可靠性、弹性

ACM 参考格式：

Shixun Wu、Yujia Zhai、Jinyang Liu、Jiajun Huang、Zizhe Jian、Bryan M. Wong 和 Zizhong Chen。2023年。GPU 上在线容错高性能 GEMM 剖析。载于*2023*年国际超级计算会议（*ICS '23*），*2023*年*6*月*21*日至*23*日，美国佛罗里达州奥兰多。 ACM，纽约州纽约，美国，13 页。
https://doi.org/10.1145/3577193.3593715

1 引言

现代计算系统日益增长的复杂性和规模使其更容易受到瞬态故障的影响。瞬态故障是指在信号传输或数值存储过程中可能发生的错误。它们可能由多种因素造成，例如晶体管宽度缩小、电路密度提高以及更低的近阈值电压操作 [27, 33, 36]。这些故障会对计算系统的可靠性产生重大影响，过去曾发生过大量瞬态故障导致服务器崩溃或使整个计算机集群无法使用的情况 [20, 28]。事实上，美国能源部已将可靠性确定为百亿亿次级计算面临的主要挑战之一，凸显了解决这一问题的重要性 [32]。

瞬态故障可能威胁到即使是最先进的计算系统的可靠性。例如，对一台拥有 190,000 个尖端 Xeon Phi 处理器的百亿亿次级计算机进行的模拟表明，即使采用纠错码进行保护，该系统仍可能每天发生瞬态错误 [40]。为了在运行时保护软件系统免受计算错误的影响并维持其可靠性，学术界和工业界已经提出并研究了各种容错技术 [3, 19, 34]。

瞬态故障影响应用程序时，可能会出现两种错误：失效停止错误会导致应用程序崩溃，而失效继续错误则允许应用程序

*Shixun Wu 和 Yujia Zhai 对本文贡献相同。

授予免费许可，允许为个人或课堂使用而制作本作品的部分或全部内容的数字副本或纸质副本，前提是不得为营利或商业利益而制作或分发副本，且副本须保留本声明及首页的完整引文。必须尊重本作品中第三方组件的版权。其他所有用途，请联系所有者/作者。*ICS '23*，*2023*年*6*月*21*日至*23*日，美国佛罗里达州奥兰多© 2023年版权归所有者/作者所有。ACM ISBN 979-8-4007-0056-9/23/06。https://doi.org/10.1145/3577193.3593715

继续运行，但产生错误结果。失效停止错误通常可以通过检查点/重启机制[1, 41, 42, 46] 或算法方法 [8, 11, 22], 来缓解，但失效继续错误可能更加危险，因为它们会在毫无预警的情况下破坏应用程序状态，并导致计算结果错误 [4, 13, 18, 35, 45]。这类错误在安全关键场景 [29]中尤其棘手。本文重点关注计算逻辑单元中的失效继续错误（例如计算产生错误结果的情况），并假设失效停止错误和内存错误分别由检查点/重启机制和纠错码防护。我们将这类错误称为软错误。

处理软错误的一种方法是使用双模块冗余（DMR）。DMR 通过复制计算指令并在原始程序中插入校验指令来实现，通常需要编译器的协助 [12, 38, 39, 43, 51]。虽然 DMR 是一种可应用于任何应用的通用方法，但它也可能带来显著的开销，尤其是对于需要复制所有计算的计算密集型应用。近年来，为了降低容错开销，研究人员针对各种应用开发了基于算法的容错（ABFT）方案。近年来，针对各种应用已经开发了大量 ABFT 方案。例如，Huang 和 Abraham 为矩阵乘法提出了首个 ABFT 方案 [24], ； Di 和 Cappello 提出了一种面向实际高性能计算应用的自适应影响驱动容错方法 [16], ； Chien 等人提出了 Global View Resilience 系统，这是一种使应用能够高效实现弹性的库 [14]。此外，针对排序 [30], 、快速傅里叶变换 [2, 31, 47], 、迭代求解器 [6, 9, 48], 和卷积神经网络 [55] 等广泛使用的算法，也有相应的 ABFT 方案。

通用矩阵乘法是机器学习和科学计算中各种应用的重要构建模块。这些应用通常会在配备图形处理器等加速器的大型计算设施上长时间运行。因此，确保图形处理器上的通用矩阵乘法能够抵御软错误至关重要，相关问题已在中央处理器 [10, 21, 50] 和图形处理器 [17, 26]上得到广泛研究。现有研究已证明在图形处理器上为通用矩阵乘法部署算法级容错是可行的，但这些研究要么缺乏架构感知设计和优化，无法降低算法级容错带来的内存开销 [17] ，要么无法即时纠正检测到的错误。为解决这些问题，我们首先从零开始构建了一个高性能通用矩阵乘法内核，并以此为基础提出了一种新颖高效的方法：通过实现定制的容错图形处理器内核，将算法级容错的内存操作与原始通用矩阵乘法的内存占用相融合。我们还设计了一种基于模板的代码生成方案，以减少内核开发工作量，并支持范围广泛的输入矩阵形状。具体而言，我们的贡献包括：

- 我们的工作从优化原始通用矩阵乘法开始。通过采用一系列调优策略，包括分块、寄存器复用，以及在寄存器和共享内存层级进行预取，我们的通用矩阵乘法性能达到了与闭源 cuBLAS 通用矩阵乘法相当甚至更高的水平。我们提供了一个公开的 GitHub 仓库，用于存档本文相关的源代码 [1]。
- 我们从线程级、线程束级和线程块级详细剖析了 GPU 上具有在线容错功能的高性能GEMM。
- 我们提出了一种基于模板的代码生成策略，可针对各种输入形状自动生成具有或不具有容错功能的高性能GEMM内核。
- 我们在 NVIDIA Tesla T4 GPU 和 NVIDIA A100 GPU 上对本工作进行了基准测试。与此前最先进的非融合容错基线相比，我们的内核融合策略在保持在线错误校正功能的同时，平均实现了 39.04% 的加速比。即使每分钟注入数百个错误，具有容错功能的通用矩阵乘法版本相较于闭源 cuBLAS 的平均开销也仅为 8.89%，微乎其微。此外，对于不规则形状输入，代码生成器生成的内核在具有和不具有容错功能的通用矩阵乘法中，分别实现了 160% ~ 183.5 % 和 148.55% ~ 165.12% 的显著加速比，最高超越 cuBLAS 41.40%。

本文其余部分安排如下：第 2 节介绍背景和相关工作，第 3 节和第 4 节分别介绍我们针对原始通用矩阵乘法和容错通用矩阵乘法的设计与优化。第 5 节给出评估结果。第 6 节总结本文并展望未来工作。

2 背景与相关工作

本章概述两类主要的软件方法——基于复制的方法和基于算法的方法，以应对运行时计算错误。我们还介绍了采用内核融合策略来提升应用性能的先前工作，尤其是那些围绕通用矩阵乘法操作展开的工作。

2.1 基于复制的容错 nce

基于复制的容错也称为双模块冗余（DMR），多年来一直是防范软件错误的常见保护方案。这种技术源于编译器辅助方法，已得到广泛研究，并根据复制域（SoR），即冗余执行的逻辑域进行分类。以往关于基于复制的容错的工作可分为三类：（1）线程级复制（TLD），复制整个处理器和内存系统；（2）采用纠错码假设的 TLD（TLD+ECC），复制指令，但仅从相同的内存地址加载操作数；（3）仅针对计算错误的 DMR，只复制计算指令，以防止错误结果写入内存。

2.2 基于算法的容错

软错误保护算法旨在检测并纠正迭代型或计算密集型应用在计算过程中可能发生的错误。这些算法有着悠久的成功历史，第一个此类算法的开发】【。

[1] https://github.com/shixun404/Fault-Tolerant-SGEMM-on-NVIDIA-GPUs.git

用于在 1984 [24] 中进行矩阵乘法。这些算法的基本思想是分别使用以下公式将矩阵 A 和 B 编码为校验和形式 A^c 和 B^r：

$$A \xrightarrow{encode} A^c := \begin{bmatrix} A \\ e^T A \end{bmatrix}, \tag{1}$$

$$B \xrightarrow{encode} B^r := \begin{bmatrix} B & Be \end{bmatrix}, \tag{2}$$

其中，e 是转置的单位向量，$[1, 1, \ldots, 1]^T$。随后将编码矩阵相乘，得到一个矩阵 C^f，其中同时包含正确结果和校验和信息：

$$C^f = A^c \cdot B^r = \begin{bmatrix} C & Ce \\ e^T C & \end{bmatrix} = \begin{bmatrix} C & C^r \\ C^c & \end{bmatrix}. \tag{3}$$

通过比较矩阵 C 中的值与 C^r 和 C^c 中的校验和值，可以验证最终结果的准确性。如果这些值之间的差异超过预定阈值，则表明计算过程中发生了错误。使用校验和对矩阵进行编码和验证的成本通常远低于矩阵乘法本身的成本，因此这是一种轻量级的错误检测方法。这些校验和算法可用于任何矩阵乘法算法，并且可以在线（计算过程中）或离线（计算完成后）验证最终结果的准确性。

Chen 等人 [10] 提出了一种外积矩阵-矩阵乘法，其中在计算过程中可以维持校验和关系：

$$C^f = \sum_s A^c(:, s) \cdot B^r(s, :) = \sum_s \begin{bmatrix} C_s & C_s e \\ e^T C_s & \end{bmatrix}. \tag{4}$$

这里，s 是矩阵 C 外积更新的步长，C_s 表示外积乘法 $A^c(:, s) \cdot B^r(s, :)$ 的每一步结果。双校验和方案的离线版本在完整执行过程中只能纠正一个错误，而在线版本可在外积更新的每个步骤中纠正一个错误，因此能够处理整个程序中的多个错误。后来，在2011年，Ding及其同事将基于算法级容错的外积通用矩阵乘法扩展到了图形处理器上 [17]。为进一步隐藏算法级容错引入的校验和相关内存延迟，Zhai 等人为支持 AVX-512 的中央处理器上的通用矩阵乘法设计了融合计算内核 [52]。在图形处理器上，Kosaian 和 Rashmi 展示了首次尝试将容错用于通用矩阵乘法，但该方法只能检测错误，无法纠正错误 [26]。尽管他们成功地将校验和计算的内存延迟与原始通用矩阵乘法操作的数据移动融合起来，但要纠正检测到的错误，必须预先分配明确建立的额外内存空间，这不可避免地需要修改相关源代码。为弥补这一不足，本文提出了首个完全融合且具备在线容错能力的通用矩阵乘法，可高效地检测并纠正计算错误。

2.3 内核融合

内核融合是一种性能优化技术，已在中央处理器和图形处理器领域得到广泛研究。内核融合的主要目标是将多个计算内核合并为一个更高效的内核，从而减少与内核启动、内存传输及其他性能瓶颈相关的总体开销。通过利用数据局部性、减少内存访问并提升缓存利用率，事实证明，内核融合能够在各种应用和计算领域中带来显著的性能提升。

多项研究探讨了内核融合在中央处理器和图形处理器上各种应用中的潜在优势。例如，Chen 等人 [7] 提出了一种编译方法，可自动融合中央处理器上线性代数和深度学习操作中的数值内核。在图形处理器领域，Jia 等人 [25] 将内核融合集成到广受欢迎的深度学习框架 Caffe 中，而 Rhu 等人 [44] 则提出了一种融合深度神经网络中卷积层的技术，以减少内存占用并提升性能。Dao 等人 [15] 和 Zhai 等人 [54] 提出了融合方案，显著提升了 Transformer 模型核心算法——多头注意力——在图形处理器上的性能。内核融合在同态加密 [53]和科学计算 [5] 等其他领域也展现出优势。这些研究凸显了内核融合作为一种性能优化策略，在中央处理器和图形处理器上的各类应用中都十分有效，并强调了其在追求高性能计算方面的重要性。本文通过利用内核融合策略，针对图形处理器上的容错通用矩阵乘法性能优化提出了新的见解。

3 不带容错的单精度通用矩阵乘法优化

高效的容错通用矩阵乘法实现，需要将算法级容错的内存占用与原始通用矩阵乘法操作融合。这意味着，高效的通用矩阵乘法代码库是实现算法级容错仪器融合内存操作的基础。由于 cuBLAS 通用矩阵乘法是闭源的，从头实现高性能单精度通用矩阵乘法已不可避免。本节将详细介绍单精度通用矩阵乘法（SGEMM）逐步优化的过程：从朴素基线出发，通过利用共享内存、向量化内存访问和预取，使性能超越闭源的 cuBLAS。随后，我们将介绍自动代码生成策略，以便将这一性能推广到各种输入矩阵形状。

3.1 单精度通用矩阵乘法的逐步优化

图 1 展示了图形处理器上单精度通用矩阵乘法的整体概览。在设备级别，单精度通用矩阵乘法执行 $C+=A\cdot B$，其中 A、B 和 C 分别是 $M\times K$、$K\times N$ 和 $M\times N$ 矩阵。在线程块级别，一个线程块累加 $C_s+=A_s\cdot B_s$，其中 A_s、B_s 和 C_s 分别是 $m_s\times k_s$、$k_s\times n_s$ 和 $m_s\times n_s$ 矩阵。在线程束级别，一个线程束累加 $C_w+=A_w\cdot B_w$，其中 A_w、B_w、C_w $m_w\times k_w$ $k_w\times n_w$ $m_w\times n_w$ 分别是 、 和 矩阵。在线程级别，一个线程累加 $C_t+=A_t\cdot B_t$，其中 A_t、B_t 和 C_t 分别是 $m_t\times k_t$、$k_t\times n_t$ 和 $m_t\times n_t$ 矩阵。在本小节中，我们从头介绍通用矩阵乘法的优化技术，详细阐述每个步骤的优势和理论依据，以揭开闭源

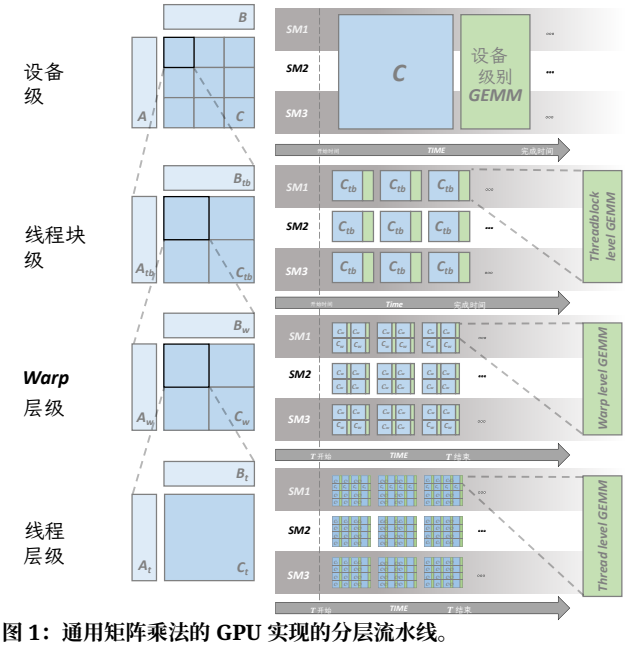


图 1：通用矩阵乘法的 GPU 实现的分层流水线。

cuBLAS 库，为后续的容错通用矩阵乘法优化奠定基础。

3.1.1 朴素实现。我们从朴素基线开始介绍逐步进行的优化。这里，将 $C+=AB$ 的工作负载沿行和列两个维度进行划分，使每个线程块计算一个分块 C_{tb} C $C_{tb}+=$，即 $\cdot$。因此，A_{tb} B_{tb} 每个线程计算一个元素 $C_{tb}[i,j]$ $+=A_{tb}[i,k]B_{tb}[k,j]$ $k=0\ 1$ $K-1$ $C_{tb}[i,j]$，其中，，$\cdot\cdot\cdot$。在整个计算过程中，的结果都累加到一个寄存器中。对于尺寸从 1024_2 到 6144_2 的方形矩阵输入，这一基线变体在 NVIDIA Tesla T4 GPU 上的平均性能为 611 每秒十亿次浮点运算。

3.1.2 线程块级分块。朴素基线方法存在问题：它在速度较慢的全局内存中进行大量内存访问，其复杂度为 $O(n^3)$，且没有利用速度更快的共享内存中的任何数据复用。在这一改进变体中，引入分块策略，以便在 GPU 共享内存中复用数据，从而降低通用矩阵乘法操作中高昂的全局内存延迟成本。该过程首先将分块 A_{tb} 和 B_{tb} 从全局内存加载到共享内存中。随后同步线程块，确保所有线程都已完成数据存储。每个线程从共享内存中加载元素 $A_{tb}[i,k]$ 和 $B_{tb}[k,j]$ 到寄存器 A_t 和 B_t，并计算 $C_t+=A_t\cdot B_t$。评估结果显示，这一步的平均性能提升了 11.3%，达到 679 每秒十亿次浮点运算。

3.1.3 线程级分块。 鉴于通用矩阵乘法算法的特点，即在 O（n^3）矩阵元素之间执行 O（{v6）计算，我们希望在每个线程的寄存器级别进一步实现数据复用，也就是在此步骤采用线程级分块。线程级分块并非将 C_{tb} 中的一个目标元素分配给每个线程，而是为每个线程引入更多工作负载，从而实现更多积极的寄存器级数据重用。线程级分块步骤能够更充分地利用寄存器级的架构资源，从而显著减少全局内存和共享内存中的内存访问。我们通过为每个线程分配 C 矩阵中的 4、16 和 64 个元素，探索并测试了一系列线程级分块变体。SGEMM 性能较上一步最高提升 4.62×，将每秒十亿次浮点运算提升至 3822。

3.1.4 线程束级分块。 还可以在另一个架构层级采用分块策略——线程束级。一个线程束中的 32 个线程可按 4×8 或 8×4 进行划分，分别对应为行或列维度分配 4 个或 8 个线程。对于 4×8 的线程束级分块设置，以及每个线程块中静态初始化 256 个线程（8 个线程束）的 128×128 线程块级分块，每个线程束负责一个大小为 64×32 的子矩阵 C，每个线程则负责一个 16×4 子块。由于线程束按 4×8 的方式划分，前 4 个线程共享线程束子矩阵中第一个八分之一的列分块，接下来的 4 个线程负责该子矩阵中第二个八分之一的列分块，依此类推。因此，精心设计的线程束级分块有助于改善共享内存层级的内存访问。在 NVIDIA GPU 中，当不同线程尝试访问共享内存中的同一地址时，硬件会将这些内存访问合并为一次访问。这种线程束级分块策略也不会产生共享内存 bank 冲突，能够显著降低共享内存层级的内存延迟，使性能提升至每秒 4331 十亿次浮点运算，较上一步加速 13%。

3.1.5 向量化加载/存储。 不同于以标量方式访问全局内存和共享内存（每次内存事务仅涉及一个数据元素），我们可以采用向量化内存访问，以更充分地利用内存带宽。采用 128 位向量化内存访问后，性能较之前有所提升，平均每秒浮点运算次数从 4331 GFLOPS 提高到 4381 GFLOPS。

3.1.6 预取：从共享内存到寄存器。 为了进一步隐藏访问共享内存时的内存延迟，我们将寄存器利用率提高一倍，并将共享内存访问与通用矩阵乘法计算进行流水线化，即在此步骤进行预取。寄存器利用率翻倍打破了原有的加载-计算依赖，使待计算的数据能够在前一次迭代中加载到寄存器中。每次迭代中，一个线程预取两个片段，$A_{tb}[i:i+8,\ k+1]$ 和 $B_{tb}[k+1,j:j+8]$。随后，该线程使用当前存放在寄存器中的片段 $A_{tb}[0:8,\ k]$ 和 $B_{tb}[k,0:8]$ 执行累加 $C_t[0:8,\ 0:8]+=A_t[0:8]\times B_t[0:8]$。累加完成后，将预取的片段加载到寄存器 $A_t[0:8]$ 和 $B_t[0:8]$中。这项优化使性能提升了 5.6%，每秒十亿次浮点运算从 4381 提高到 4625。

3.1.7 预取：全局内存到共享内存。预取技术也可用于在全局内存和共享内存之间传输数据，方法是将共享内存缓冲区的大小加倍。这样可以确保共享内存中所需的所有数据都存储在上一轮迭代的正确位置。例如，预取的全局内存分块 $A[i:i+128,\ k+8:k+2\times8]$，$B[k+8:k+2\cdot8,\ j:j+128]$ 会在当前

累加 $C_{tb} += A_{tb}B_{tb}$ 完成后，加载到 : 128，0 : 和 : 8，0 : 中。累加使用当前位于共享内存中的分块 $A[i: i+128,\ k: k+8]$ 和 $B[k: k+8,\ j: j+128]$。使用 __syncthreads() 函数可确保所有线程都已完成累加。将共享内存缓冲区从 $A_{tb}[0:128,\ 0: 8]$ 和 $B_{tb}[0:8,\ 0: 128]$ 加倍至 $A_{tb}[0:128,\ 0: 2\times 8]$ 和 $B_{tb}[0: 2\times 8,\ 0:128]$, 后，预取的分块便可直接存储在共享内存当前未占用的部分，无需等待累加完成，也无需等待 __syncthreads() 函数执行。在此步骤中，SGEMM 性能进一步提升了 0.6%，达到 4654 每秒十亿次浮点运算。

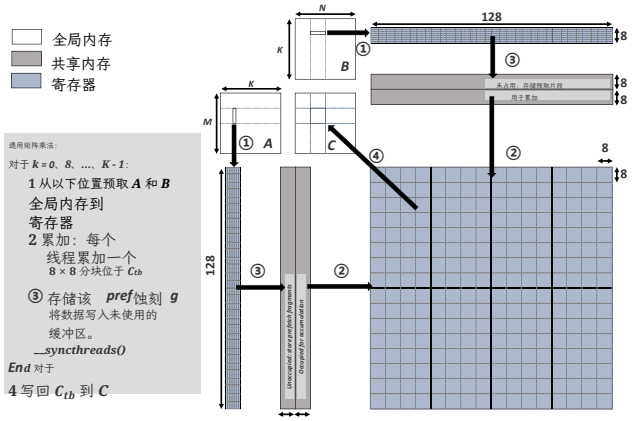


图 2：优化后的 SGEMM 内核概览。

3.2 SGEMM 的自动代码生成

静态硬编码的参数选择机制应用于更广泛的输入形状时，可能会降低性能。例如，对输出矩阵采用 128×128 的二维分块方式时，在较大的矩阵上性能令人满意。然而，对于较小的输入矩阵，相同的参数会导致运行时启动的活跃线程块数量不足，无法充分利用 GPU。为了解决这一问题，我们提出了一种更灵活的参数选择机制。然而，集成一系列硬编码内核会产生高昂的开发成本。因此，我们对 SGEMM 内核采用模板化方法，根据输入形状提供通过半经验方式选取的内核参数，以便在运行时生成高性能的参数化内核。这种在线代码生成方案使我们能够将 SGEMM 在方阵上的出色性能推广到不规则形状输入，同时保持较低的开发成本。

表 1：Tesla T4 GPU 上的 SGEMM 内核参数配置。

	m_{tb}	n_{tb}	k_{tb}	m_w	n_w	m_t	n_t
小型	16	16	16	8	16	2	2
中型	32	32	8	16	32	4	4
大型	64	64	8	32	64	8	8
高瘦型	32	128	8	16	64	4	8
huge	128	128	8	32	64	8	8

3.2.1 代码生成策略。除了遵循逐步优化单精度通用矩阵乘法的流程外，代码生成方案还接收 7 个参数作为输入，并生成相应的高性能单精度通用矩阵乘法内核。内核参数为 n_{tb}，m_{tb}，k_{tb}，m_w，n_w，m_t 和 n_t。这些参数分别对应线程块级（tb）、线程束级（w）和线程级（

t），分别。在代码生成模板中，内存操作经过精心设计，以避免存储体冲突。其他参数则由分块大小直接决定，例如双缓冲区大小以及向量化加载/存储的数据类型。图 8 展示了单精度通用矩阵乘法的代码生成模板。

3.2.2 内核参数。我们使用代码生成模板生成了具有不同参数的单精度通用矩阵乘法内核。通过实证分析，我们分别针对输入形状 1 ~ 128、128 ~ 256、256 ~ 512 和 > 512，确定了四种性能最佳的内核。这些内核分别命名为 small、medium、large 和 huge。此外，我们还开发了一种针对不规则输入形状优化的内核，命名为 tall and skinny。在此，我们采用一种受一系列基于 NVIDIA CUTLASS [23, 37] 的研究启发的参数选择方案。表 1 总结了相应内核参数的详细信息。

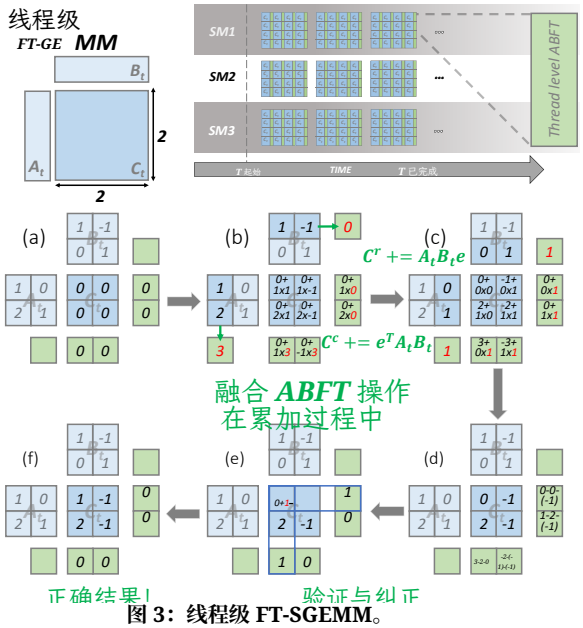


图 3：线程级 FT-SGEMM。

4 优化具备容错能力的 SGEMM

自行实现的高效 SGEMM 内核为我们提供了一个高性能代码库，用于在计算内核中构建轻量级容错方案。本节将针对三种不同的硬件层级——线程级、线程束级和线程块级——探讨用于通用矩阵乘法的融合式算法级容错方案。

4.1 故障模型

在深入探讨故障容错设计之前，我们先详细说明故障模型。本研究聚焦于检测和纠正

计算单元中发生的错误可能会影响最终输出矩阵的结果。我们不关注内存中的故障，因为在 ECC 的保护下，内存相对更具韧性。为在运行时处理计算错误，我们在单粒子翻转（SEU）假设下设计容错方案，即每个错误检测与纠正周期内仅发生一次软错误。SEU 假设在许多研究工作中被广泛用作故障模型 [17, 43, 49, 52], ，这是合理的，因为短故障检测间隔使多个软错误同时发生的概率较低。

4.2 探索各种算法级容错单精度通用矩阵乘法方案

4.2.1 线程级算法级容错单精度通用矩阵乘法。在最初的尝试中，我们旨在在线程级实现容错功能。在线程级，算法级容错相关操作可以融合在线程级累加过程中，无需额外的线程束内或线程块间通信。

图 3 展示了线程级 FT-GEMM 检测并纠正故障的过程。在图 3(a) 中，一个线程为 C_t（蓝色标记）初始化 4 个寄存器，并为校验和（绿色标记）初始化 6 个寄存器。 A_t 和 B_t 存储在共享内存中（白色标记）。在图 3(b) 和 (c) 中，该线程加载 A_t 和 B_t 的片段以执行矩阵乘法。同时更新这些片段及 C_t 的校验和，即 e^TA_t、 B_te、 $C^c+=e^TA_tB_t$ $C^r+=A_tB_te$ 3 C_t 和 。在图 (d) 中，使用校验和对 进行验证。在图 3(e) 中，检测并纠正了一个故障。故障位置由两个校验和中的相对位置确定；另一方面，纠正值由校验和中的偏移量确定。最后，我们在图 3(f) 中得到无故障的 C_t 。

线程级算法级容错方案虽然不需要线程束内或线程块内通信的成本，但会带来相当大的计算开销。这是因为标准通用矩阵乘法的计算复杂度为三次方，而校验和编码的计算复杂度为二次方。受保护的矩阵越大，校验和编码带来的开销就越低。然而，由于寄存器资源有限，线程级算法级容错只能处理小型矩阵（例如， 8×8），从而导致显著的开销。在实践中，在 NVIDIA Tesla T4 GPU 上，线程级算法级容错

对于大小从 1024^2 到 6144^2的方阵，平均开销为 25%。

4.2.2 线程束级算法级容错单精度通用矩阵乘法。 虽然线程级算法级容错不会增加内存操作，但算法级容错编码会使总计算量增加 $(4n_t)/(2n_t^2)=2/n_t$，其中 n_t是线程的分块大小。根据表 1，这会导致计算量增加 25% ~ 100%。考虑到通用矩阵乘法的计算强度，计算量的增加会带来不可忽视的开销。因此，将算法级容错的编码单元从线程提升到线程束是一个不错的选择。理想情况下，线程束级算法级容错可将计算量增幅降至 5%，但会引入若干线程束级通信。这些通信包括对 e^TA_w和 B_we的编码进行线程束级归约、分发 e^TA_w和 B_we，以及从不同线程收集 C_w^c和 C_w^r。首先，我们发现，线程束级归约可以与 A_{tb}和 B_{tb}的预取融合（图 2 中的步骤 3）。每个线程束持有一个或多个 A_w和 B_w的片段。因此， e^TA_w和 B_we 可以通过调用 2 ~ 5 次线程束级归约原语来计算

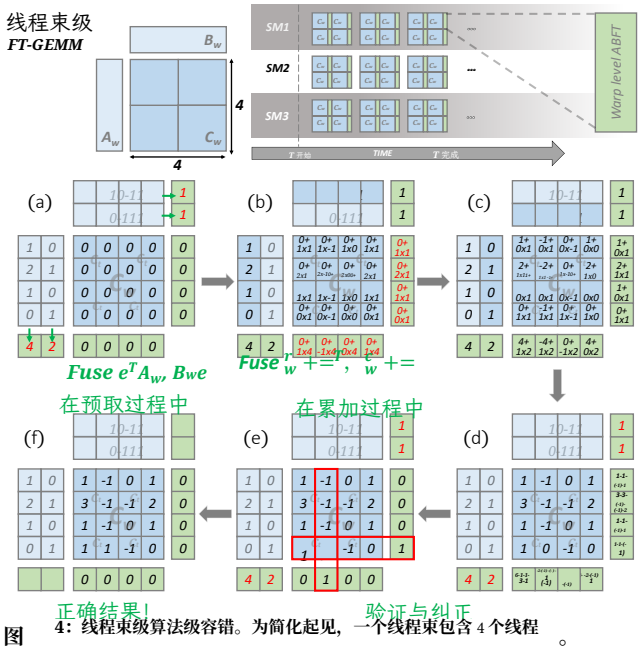


图 4：线程束级算法级容错。为简化起见，一个线程束包含 4 个线程 。

原语。该原语在寄存器级交换数据，无需额外的同步延迟。接下来，我们复用双缓冲的空闲部分来分发 e^TA_w和 B_we，无需额外的内存空间。之后， C_w^c和 C_w^r会与通用矩阵乘法同时更新。在验证过程中，首先将 C_w^c和C_w^r从线程收集到共享内存中，然后与 C_w进行比较。最后，每个线程检查 C_w^c和 C_w^r，以验证并纠正结果。

图 4 展示了线程束级ABFT的一个简单示例。在图 4(a) 中，计算 e^TA_w 和 B_we 的编码并将其存储在共享内存中（以白色标示）。在图 4(b) 和 (c) 中，每个线程加载 A_w 和 B_w 的片段以执行矩阵乘法。同时加载编码 e^TA_w 和 B_we，以便随后更新： $C_w^c+=e^TA_wB_w$ 和 $C_w^r+=A_wB_we$。在图 4(d) 中，先将 C_w^r 和 C_w^c 加载到共享内存中，然后与 C_w 进行比较。在图 4(e) 中，检测到故障并将其纠正。最后，我们在图 3(f) 中得到无故障的 C_w。尽管线程束级ABFT的计算开销较低，但每次更新 C_w 时都需要额外读取两次共享内存。虽然这些读取操作无需同步，但仍会限制通用矩阵乘法的效率。因此，我们提出线程块级基于算法的容错方案，进一步减少额外计算，并将所有通信融合到预取过程中，以最大限度地降低开销。

4.2.3 线程块级基于算法的容错单精度通用矩阵乘法。 为了进一步隐藏算法级容错的开销，我们在线程块级应用基于算法的容错编码。为避免通用矩阵乘法累加期间产生额外延迟，我们将所有编码与预取阶段融合。图 5 展示了线程块级基于算法的容错方案示例。如图 5(a) 所示，所有基于算法的容错编码都与预取阶段融合。通过精心设计的预取策略，无需额外的全局读取操作，即可通过线程束级归约获取 e^TA 和 Be 中的每个元素。随后，我们的预取策略使得 和 的编码能够在线程内完成，而无需线程块级通信。

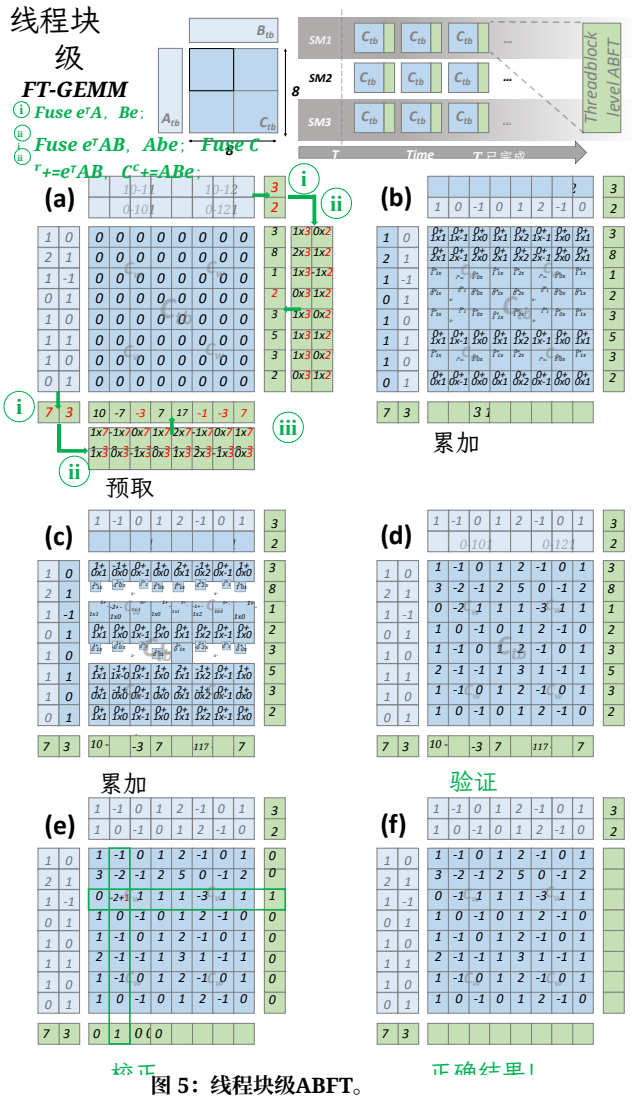


图 5：线程块级ABFT。

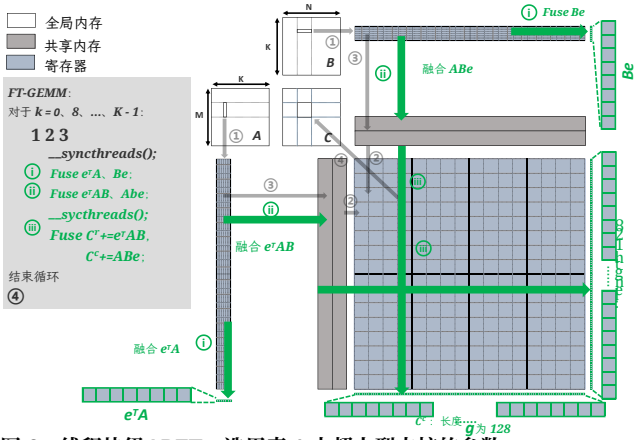


图 6：线程块级ABFT。选用表 1 中超大型内核的参数。

将其与图 7 中的普通 GEMM基线进行比较。借助我们从头构建的高效通用矩阵乘法内核，校验和编码和在线更新的内存占用已与通用矩阵乘法计算完全融合，从而将容错开销降至最低。

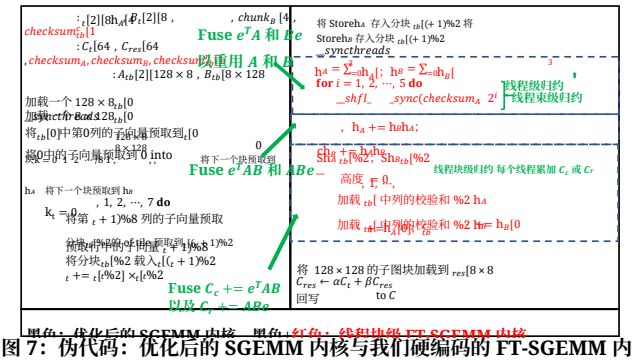


图 7：伪代码：优化后的 SGEMM 内核与我们硬编码的 FT-SGEMM 内核。算法级容错相关操作以红色标出。

最后，通过线程块级归约更新目标校验和 C^r 和 C_c。在图 5(b-c) 中，通用矩阵乘法累加无需额外操作即可完成。图 5(d-f) 遵循与线程级ABFT和线程束级ABFT相同的验证与纠错流程。

图 6 进一步展示了线程块级 FT-SGEMM 的工作流程详情。上述校验和编码与解码方案应用于每个线程块级外积 k 循环，以在线更新校验和。与前两种方案相比，线程块级基于算法的容错充分利用了片段 A_{tb} 和 B_{tb} 的预取阶段。由于额外计算开销可以忽略不计，且精心设计的融合策略能够隐藏编码开销，线程块级基于算法的容错在另外两种融合式算法级容错方案中性能最佳。接下来，我们介绍一种基于线程块级基于算法的容错的自动 FT-SGEMM代码生成方法。我们总结了线程块级基于算法的容错伪代码，并

4.3 自动代码生成

如第2.3节所述，通过开发一种代码生成方案，使我们的高性能内核能够适用于广泛的输入形状，同时将开发成本维持在合理范围内。这里，我们提出一种线程块级 FT-SGEMM 的代码生成策略。尽管采用了第 3.2 节中相同的代码生成逻辑，FT-SGEMM 的代码生成策略仍需为算法级容错编码进行额外设计。考虑到内核参数范围广泛，编码位数与线程数之比会有所变化。这导致不同内核规模下每个线程承担的工作负载各不相同。为保持卓越性能，我们首先确保线程访问合并，并避免存储体冲突。其次，将工作负载平均分配给不同线程。图 8 展示了 FT-SGEMM 代码生成模板的伪代码。算法级容错操作以红色标出。单精度通用矩阵乘法的内核参数沿用表 1 中的设置。

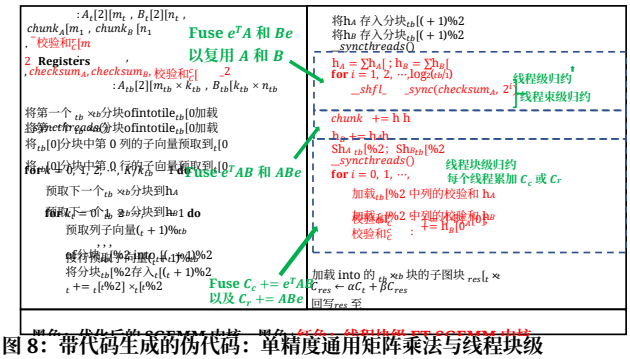


图 8：带代码生成的伪代码：单精度通用矩阵乘法与线程块级 FT-SGEMM。算法级容错相关操作以红色标记。

5 性能评估

我们在两款 NVIDIA GPU 上评估了所提出的优化方案：Tesla T4 和 A100。Tesla T4 GPU 所在节点配备两颗 16 核 Intel Xeon Silver 4216 中央处理器，最高睿频为 3.2 GHz。该节点的中央处理器主存容量为 512 GB，频率为 2400 MHz。我们使用带有 –O3 优化标志的 CUDA 11。4 编译程序。A100 GPU 所在节点配备一颗 64 核 AMD EPYC 7742 中央处理器，睿频为 3.4 GHz。我们评估了逐步进行的单精度通用矩阵乘法优化、不同级别的算法级容错单精度通用矩阵乘法策略，以及生成的内核的性能。我们首先将 FT-SGEMM 与 Ding 等人在 2011 [17] 中首次提出的、此前最先进的容错通用矩阵乘法实现进行比较。我们还将 FT-SGEMM 与 CUDA 11。6 内置的 cuBLAS SGEMM 进行比较。报告的性能数据取数十次运行的平均值，以尽量减小波动。

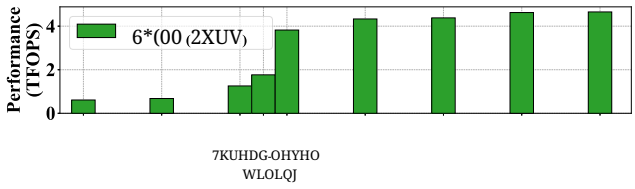


图 9：逐步优化单精度通用矩阵乘法。

5.1 单精度通用矩阵乘法（无容错）基准测试

5.1.1 单精度通用矩阵乘法的逐步优化。图 9 展示了在 NVIDIA Tesla T4 GPU 上对不具备容错能力的单精度通用矩阵乘法所做的逐步优化。基准性能为 611 每秒十亿次浮点运算；采用线程块级分块后，由于共享内存中的数据复用增加，性能提升了 11.3%。通过寄存器级数据重用，性能进一步提升至 3822 每秒十亿次浮点运算。值得一提的是，这些简单的优化已经优于 Tesla T4 GPU 上的 cuBLAS SGEMM。此外，我们提出的流水线策略对计算与数据传输进行重叠，传输路径包括从全局内存到共享内存以及从共享内存到寄存器；采用该策略后，性能达到 4654 每秒十亿次浮点运算，比初始版本快 7.62 倍。

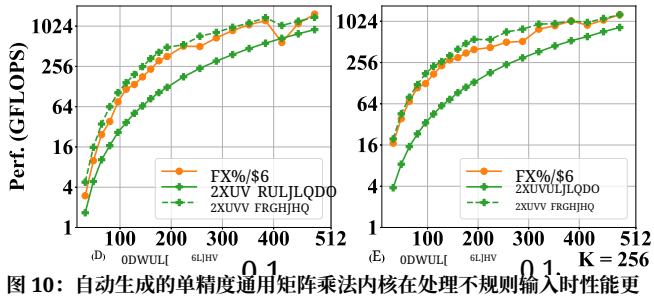


图 10：自动生成的单精度通用矩阵乘法内核在处理不规则输入时性能更佳。

5.1.2 非容错单精度通用矩阵乘法的自动代码生成。除了提供硬编码的单精度通用矩阵乘法实现外，我们还将其内核模板化，使其能够将方阵上的性能推广至各种不规则形状的输入，同时无需在内核开发上投入大量人工。图 10 比较了生成的非容错单精度通用矩阵乘法内核与
在同一块 Tesla T4 GPU 上与 cuBLAS 和原始内核进行比较。通过在运行时自动选择一组更灵活的分区参数，生成的内核相比硬编码基线最高可提升 230。96%。与闭源的 cuBLAS 单精度通用矩阵乘法相比，我们自动生成的内核平均加速 18。21%。

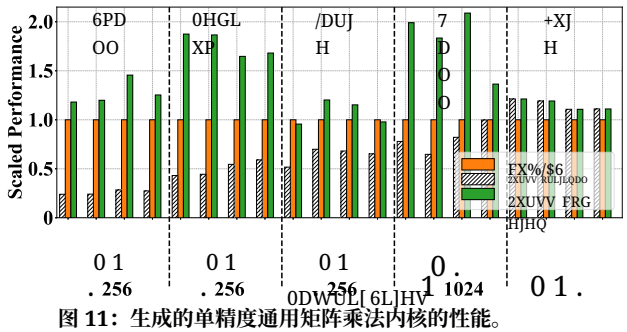


图 11：生成的单精度通用矩阵乘法内核的性能。

我们进一步将基于模板的代码生成策略从小型方阵推广到一系列不规则形状输入，如图 11 所示，包括小型、中型、大型、高矩阵和超大型矩阵。通过采用表 1 中的参数选择策略，我们生成的 SGEMM 内核在各种输入形状下的性能均领先于大型方阵。将 K 维固定为 256，并将 M 和 N 从 64 逐步增加到 490，步长为 32，代码生成器会采用不同的内核参数并生成相应的 SGEMM 内核。对于小型矩阵（M 和 N 均介于 64 到 112 之间），我们的代码生成器平均比 cuBLAS 快 27.23%。对于中型矩阵（M 和 N 均为 160），代码生成器会切换到另一种参数配置，比 cuBLAS 快 76.72%。对于更大的矩阵（M 和 $N \geq 384$），代码生成器会再次采用不同的参数，性能超过 cuBLAS 7.22%。对于 K 固定为 1024 的更宽矩阵，代码生成器比 cuBLAS 快 81.95%。值得一提的是，对于不规则形状矩阵，参数化内核的性能显著优于硬编码的 SGEMM，平均提升 160%。

5.2 单精度通用矩阵乘法容错基准测试

我们从头开始对单精度通用矩阵乘法进行优化，包括分块、寄存器级和共享内存级数据重用、流水线化，以及基于模板的灵活内核参数选择，使我们能够进一步探索适用于单精度通用矩阵乘法的轻量级容错方案。

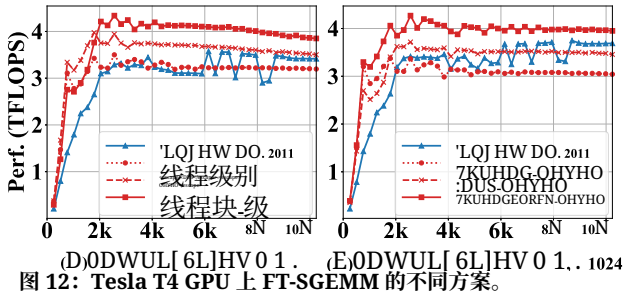


图 12：Tesla T4 GPU 上 FT-SGEMM 的不同方案。

5.2.1 单精度通用矩阵乘法容错的各种方案基准测试。图 12 对我们提出的线程级、线程束级和线程块级容错方案的性能进行了基准测试，并与先前最先进的非融合基线进行了比较。我们提出的所有融合式算法级容错方案的表现都优于非融合 FT-SGEMM。在 3 种不同的融合内核中，线程块级 FT-SGEMM 的性能最佳。对于 $M = N = K$ 情况，线程块级 FT-SGEMM 的性能分别比非融合内核、线程级 FT-SGEMM 和线程束级 FT-SGEMM 高 25.98%、19.55% 和 6.49%。对于 $M = N$，$K = 1024$ 情况，线程块级 FT-SGEMM 的性能分别比非融合内核、线程级 FT-SGEMM 和线程束级 FT-SGEMM 高 23.14%、24.85% 和 13.88%。

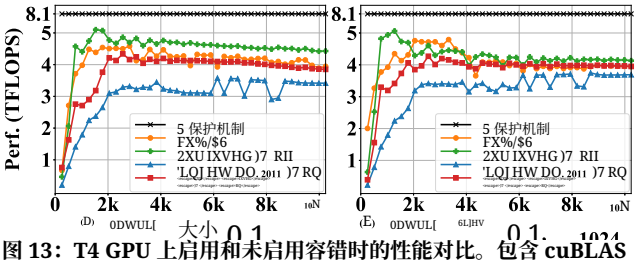


图 13：T4 GPU 上启用和未启用容错时的性能对比。包含 cuBLAS 和非融合 FT-SGEMM 内核的基准测试。启用和禁用表示是否启用了容错功能。

在融合式 FT-SGEMM 算法中启用容错能力会产生平均 11.31% 的开销；然而，该算法的性能仍可与业界最先进的 cuBLAS 库相媲美。图 13 比较了 cuBLAS SGEMM、非融合 FT-SGEMM 以及启用和未启用容错的融合式 FT-SGEMM 的性能。对于 $M = N = K$ 方阵，容错会增加 14.85% 的开销；而对于 $M = N$，$K = 1024$，开销为 8.55%。值得注意的是，与 cuBLAS SGEMM 相比，启用容错的融合式 FT-SGEMM 的开销处于 5.33% 至 7.71% 之间，属于合理范围。这些结果证明了所提出的融合式算法级容错方案的有效性。

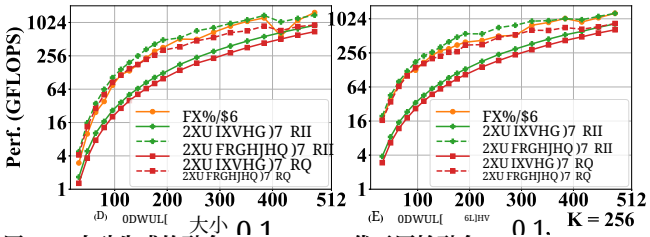


图 14：自动生成的融合 FT-SGEMM 优于原始融合 FT-SGEMM。

5.2.2 使用代码生成技术的 *FT-GEMM*。代码生成技术进一步提升了我们的 FT-SGEMM 实现的性能。图 14 比较了自动生成的内核与 cuBLAS 和原始内核。关闭容错时，自动生成的 FT-SGEMM 内核平均性能分别比原始内核和 cuBLAS SGEMM 高 183.50% 和 27.81%。开启容错时，自动生成的 FT-SGEMM 内核相较于原始内核的性能提升降至 165.12%，但相较于 cuBLAS SGEMM 的开销则从 59.23% 大幅降至 4.88%。

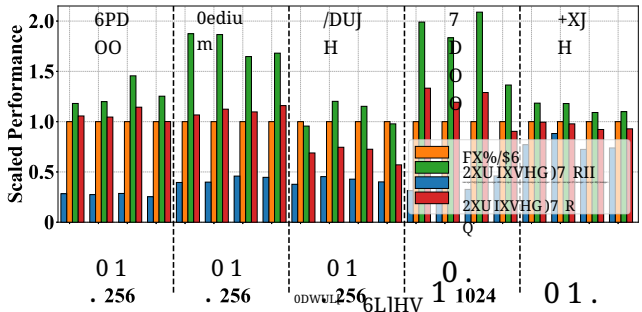


图 15：Tesla T4 GPU 上启用容错的生成式单精度通用矩阵乘法性能。

图 15 比较了针对非规则输入形状（超出方形输入范围）生成的五个融合容错 SGEMM 内核与 cuBLAS 和非融合 FT-SGEMM 的性能。将基线 SGEMM 内核推广到容错内核时，我们采用了相同的模板化、参数化和代码生成技术。我们的线程块级容错方案仅为基线带来微乎其微的性能开销，并保持与闭源 cuBLAS SGEMM 相当或更快的性能。具体而言，在启用容错功能后，生成的内核根据内核大小不同，性能比 cuBLAS 高 7.22% 至 81.95%，比非融合 FT-SGEMM 高 64.69% 至 287.06%。

5.3 错误注入下的容错单精度通用矩阵乘法基准测试

我们通过向每个计算内核注入多个计算错误来验证所提出的容错方案的有效性，并将最终计算结果与 cuBLAS 进行对比验证。我们选择在源代码级别注入错误，以最大限度地降低对原生程序性能的影响。由于本文主要关注计算错误，而非内存错误，因此，错误

通过向累加结果的寄存器添加数值偏移量来模拟寄存器位翻转。对于每个线程块，在整个计算过程中，将一定数量的错误均匀注入随机线程中。注入的错误会导致校验和验证步骤中出现不匹配，因此可以根据校验和关系计算出错误元素及错误幅度。随后，通过从错误位置减去错误幅度来纠正检测到的错误。纠正所有检测到的错误后，我们将最终计算结果与 cuBLAS 进行比较，以验证结果的正确性，并确保容错方案合理有效。

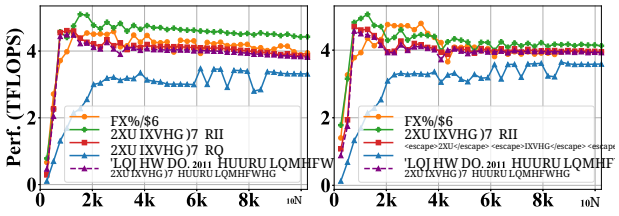


图 16：在 T4 GPU 上进行错误注入。

图 16 比较了错误注入条件下容错单精度通用矩阵乘法的性能。由于 Ding 的算法级容错采用基于外积的通用矩阵乘法，即输出矩阵通过一系列 M-by-$K_s \times K_s$-by-N 通用矩阵乘法累加得到，因此会在 K 维度上每隔固定步长 $K_s = 256$ 注入错误。为确保公平比较，我们采用与 Ding 等人在 2011 [17] 中提出的非融合算法级容错方案相同的错误注入间隔 K_s。也就是说，对于每个外积矩阵乘法子问题，注入（并纠正）1、2、.....、40 个错误，其 K 维度范围为 256 至 10240。实验结果验证了，与仅检测故障的 FT-SGEMM 方案相比，我们的错误纠正功能仅增加了极少的额外时钟周期。与 Ding 的非融合算法级容错方案相比，我们的融合 FT-SGEMM 平均加速比达到 38.8%，同时开销可忽略不计，不到 10%。与 cuBLAS 相比，我们的融合 FT-SGEMM 在注入错误时仅有 3.22% ～ 4.9% 的开销。

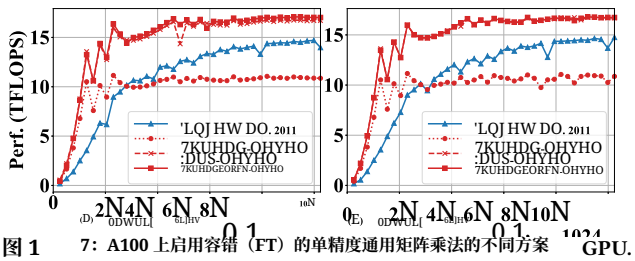


图 17：A100 上启用容错（FT）的单精度通用矩阵乘法的不同方案 GPU.

5.4 A100 上的性能评估

除了 NVIDIA Tesla T4 GPU，我们还在 NVIDIA A100 GPU 上进一步评估了这些优化的效果。图 17 比较了非融合 FT-SGEMM、线程级 FT-SGEMM、线程束级 FT-SGEMM 和线程块级 FT-SGEMM。线程块级 FT-SGEMM 的性能优于非融合 FT-SGEMM、线程级 FT-SGEMM 和线程束级 FT-SGEMM。对于 $M = N = K$ 情况，线程块级 FT-SGEMM 的性能优于非融合、线程级和线程束级方案分别提高了 52.39%、47.21% 和 1.02%。对于 $M = N$, $K = 1024$ 情况，线程块级 FT-SGEMM 的性能分别优于非融合、线程级和线程束级方案 54.93%、47.18% 和 0.04%。

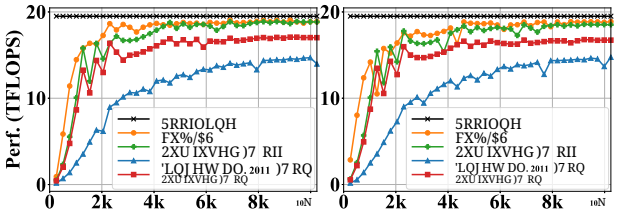


图 18：A100 GPU 上的单精度通用矩阵乘法与 FT-SGEMM 对比。

图 18 对比了 cuBLAS SGEMM、基线算法级容错单精度通用矩阵乘法、我们优化后的 SGEMM 和我们的 FT-SGEMM 的性能。对于 $M = N = K$ 情况，我们的 SGEMM 内核相比 cuBLAS SGEMM 的开销为 6.29%。我们的算法级容错单精度通用矩阵乘法相比 cuBLAS SGEMM 的开销为 15.32%，相比我们的 SGEMM 的开销为 9.93%。对于 $M = N$, $K = 1024$ 情况，我们的 SGEMM 内核相比 cuBLAS SGEMM 的开销为 3.27%。我们的算法级容错单精度通用矩阵乘法相比 cuBLAS SGEMM 的开销为 15.15%，相比我们的 SGEMM 的开销为 9.44%。

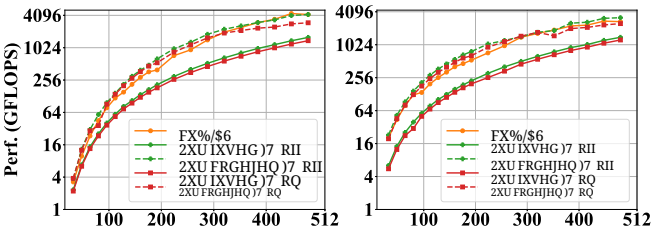


图 19：A100 GPU 上的代码生成。

图 19 比较了自动生成内核与 cuBLAS 及原始内核的性能。对于 $M = N = K$ 场景，自动生成的 SGEMM 内核性能比 cuBLAS SGEMM 高 20.26%，比我们的原始 SGEMM 高 160.10%。自动生成的 FT-SGEMM 内核性能比 cuBLAS SGEMM 高 5.94%，比原始 FT-SGEMM 内核高 148.55%。对于 $M = N$, $K = 256$ 的情况，自动生成的 SGEMM 内核性能比 cuBLAS SGEMM 高 22.45%，比我们的原始 SGEMM 高 197.78%。自动生成的 FT-SGEMM 内核性能比 cuBLAS SGEMM 高 7.07%，比原始 FT-SGEMM 内核高 195.07%。

图 20 比较了生成的内核与 cuBLAS 和算法容错基线的性能。图中包含五种不同尺寸的生成内核。对于形状不规则的通用矩阵乘法，如表 1 所示，我们的启发式参数选择机制相比硬编码方案有显著提升。此外，我们的容错方案仅产生轻微开销——在

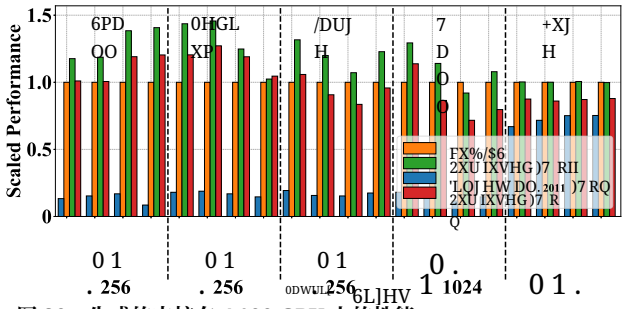


图 20：生成的内核在 A100 GPU 上的性能。

一系列输入矩阵上的平均开销为 15.68%。与 cuBLAS SGEMM 相比，我们生成的内核在小型输入上的性能略快，在较大矩阵上的性能则相当。同时，与非融合的算法容错基线相比，我们融合的算法容错内核表现出显著提升：对于从小型到超大型的输入形状，平均速度提高了 462.56%。

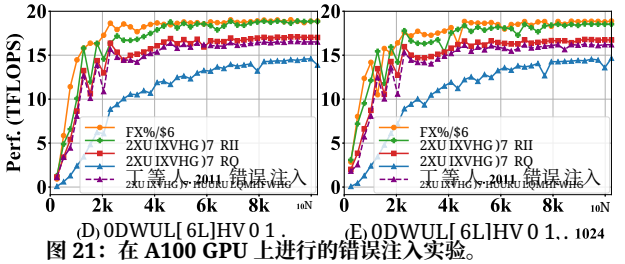


图 21：在 A100 GPU 上进行的错误注入实验。

在数十次错误注入的情况下，我们的 FT-SGEMM 性能优于非融合 ABFT SGEMM，提升了 56.12%。如图 21 所示，在发生错误注入时，FT-SGEMM 的性能比不具备任何容错能力且未进行错误注入的 cuBLAS SGEMM 慢 18%。对于 $M = N = K$ 情况，我们在错误注入条件下的 ABFT-SGEMM 比 cuBLAS SGEMM 慢 18.17%，与我们优化后的 SGEMM 相比，开销为 12.97%；与未进行错误注入的 ABFT-SGEMM 相比，开销为 3.39%。对于$M = N, K = 1024$ 情况，我们在错误注入条件下的 ABFT-SGEMM 比 cuBLAS SGEMM 慢 18.40%，与我们优化后的 SGEMM 相比，开销为 13.03%；与未进行错误注入的 ABFT-SGEMM 相比，开销为 3.96%。

5.5 在线算法容错与离线算法容错

除了详细介绍算法级容错的优化技术外，我们还通过将容错方法与仅检测方案进行比较，阐述了我们对容错方法的见解。我们的容错方法不仅能即时检测错误，还能纠正检测到的错误。因此，这种容错方案称为在线算法级容错方案。这种在线纠错机制需要付出代价。如果只检测错误而不进行纠正，就可以释放用于跟踪行列校验和编码的寄存器预算，从而显著降低性能开销。然而，这种仅检测的容错方案在检测到错误时需要重新计算，即采用离线算法级容错，这种情况下会带来 100% 的开销。

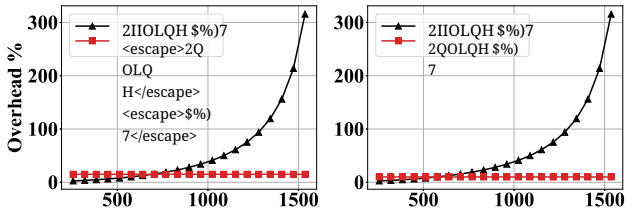


图 22：在线算法容错通用矩阵乘法与离线算法容错通用矩阵乘法。

在实践中，在线算法级容错是否比离线版本更有优势，取决于实际错误率。考虑以下情况：$C+ = AB$大小为 M，N，K。每个线程块都会累加一个大小为 m_{tb} 的分块 C_{tb}，n_{tb}。线程块中的累加错误率为 $\gamma_0 \in (0, 1)$，也就是说，每次累加发生错误的概率为 γ_0。因此，总体错误率为 $\gamma = 1 - (1 - \gamma_0)^{m_{tb} \times n_{tb}}$。由于离线算法级容错在重新计算期间发生错误时需要重新启动，因此，要获得正确的单精度通用矩阵乘法结果，预期重新启动次数为 $(1-\gamma) + 2\gamma((1-\gamma) + 2\gamma(\cdots)) = (1-\gamma)(1+2\gamma+(2\gamma)^2+\cdots) = (1-\gamma)^{1-2\gamma}$。相比之下，在线算法级容错方案能够即时精确纠错，因此获得正确结果所需的预期计算次数仅为 1。针对上述错误模型，我们在图 22 中比较了错误率为 $\gamma_0 = \frac{1}{256}$ 时在线 FT-SGEMM 与离线 FT-SGEMM 的开销。按照 [26]，中提出的思路，我们为单精度通用矩阵乘法实现了一种离线算法级容错方法，只在运行过程中检测错误。由于运行时容错所需的寄存器使用量显著减少，我们观察到其性能开销接近 1%。

6 结论

本研究提出了一种专为图形处理器设计的高性能GEMM方案，并配备了基于算法的容错功能。我们的设计涵盖线程级、线程束级和线程块级的GEMM容错方案，以及一种基准GEMM实现，其性能可与当前最先进的闭源 cuBLAS GEMM 媲美，甚至更胜一筹。此外，我们还提出了一种经济高效的基于模板的方法，用于代码生成，并支持广泛的输入矩阵形状。未来的工作将着重拓展这些策略的适用范围，纳入更多数据类型，例如用于双精度复数矩阵的 ZGEMM，以及 AMD 和 Intel 的图形处理器。

7 致谢

本研究得到了美国能源部科学办公室高级科学计算研究办公室通过高级计算科学发现（SciDAC）程序提供的资助，资助编号为 DE-SC0022209。我们感谢匿名审稿人提出的富有见地的意见。

参考文献

[1] Martín Abadi、Ashish Agarwal、Paul Barham、Eugene Brevdo、Zhifeng Chen、Craig Citro、Greg S. Corrado、Andy Davis、Jeffrey Dean、Matthieu Devin、Sanjay Ghemawat、Ian Goodfellow、Andrew Harp、Geoffrey Irving、Michael Isard、Yangqing Jia、Rafal Jozefowicz、Lukasz Kaiser、Manjunath Kudlur、Josh Levenberg、Dandelion Mané、Rajat Monga、Sherry Moore、Derek Murray、Chris Olah、Mike Schuster、Jonathon Shlens、Benoit Steiner、Ilya Sutskever、Kunal Talwar、Paul

Tucker、Vincent Vanhoucke、Vijay Vasudevan、Fernanda Viégas、Oriol Vinyals、Pete Warden、Martin Wattenberg、Martin Wicke、Yuan Yu 和 Xiaoqiang Zheng。2015年。TensorFlow：异构系统上的大规模机器学习。https://www.tensorflow.org/ 软件可从 tensorflow.org 获取。[2] Anna Antola、Roberto Negrini、MG Sami 和 Nello Scarabottolo。1992年。FFT 阵列中的容错：时间冗余方法。《VLSI 信号、图像与视频技术信号处理系统期刊》4, 4（1992年），295–316。[3] Robert Baumann。2002年。商用半导体技术中的软错误：概述与缩放趋势。IEEE 2002年可靠性物理教程笔记，《可靠性基础》7（2002年）。[4] Jon Calhoun、Marc Snir、Luke N Olson 和 William D Gropp。2017年。深入理解静默数据损坏传播。载于《第26届高性能并行与分布式计算国际研讨会论文集》。ACM，131–142。[5] Hongwei Chen、Yujia Zhai、Joshua J Turner 和 Adrian Feiguin。2023年。x86 CPU 上原子自旋动力学模拟的高性能实现。arXiv 预印本 arXiv:2304.10966（2023年）。[6] Longxiang Chen、Dingwen Tao、Panruo Wu 和 Zizhong Chen。2014年。将基于校验和的算法级容错扩展至在线容忍迭代方法中的软错误。载于《2014年第20届 IEEE 并行与分布式系统国际会议（ICPADS）》。IEEE，344–351。[7] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Meghan Cowan、Haichen Shen、Leyuan Wang、Yuwei Hu、Luis Ceze 等。2018年。TVM：面向深度学习的自动化端到端优化编译器。arXiv 预印本 arXiv:1802.04799（2018年）。[8] Zizhong Chen。2008年。将基于算法的容错扩展至高性能分布式环境中的停机故障。载于《2008 IEEE 并行与分布式处理国际研讨会》。IEEE，1–8。[9] Zizhong Chen。2013年。在线算法级容错：一种用于迭代方法软错误检测的在线算法容错方案。载于《ACM SIGPLAN 通知》，第48卷。ACM，167–176。[10] Zizhong Chen 和 Jack Dongarra。2008年。面向停机故障的基于算法的容错。《IEEE 并行与分布式系统汇刊》19, 12（2008年），1628–1641。[11] Zizhong Chen 和 Jack Dongarra。2008年。一种可扩展的无盘检查点编码算法。载于《2008年第11届 IEEE 高保障系统工程研讨会》。IEEE，71–79。[12] Zhi Chen、Alexandru Nicolau 和 Alexander V Veidenbaum。2016年。基于 SIMD 的软错误检测。载于 ACM 计算前沿国际会议论文集。ACM，45–54。[13] Chen-Yong Cher、Meeta S Gupta、Pradip Bose 和 K Paul Muller。2014年。通过硬件质子辐照和软件故障注入，了解 Blue Gene/Q 计算芯片的软错误弹性。载于《SC'14：高性能计算、网络、存储与分析国际会议论文集》。IEEE，587–596。[14] Andrew Chien、Pavan Balaji、Peter Beckman、Nan Dun、Aiman Fang、Hajime Fujita、Kamil Iskra、Zachary Rubenstein、Ziming Zheng、Rob Schreiber 等。2015年。用于科学应用弹性的版本化分布式数组：全局视图弹性。《计算机科学 Procedia》51（2015年），29–38。[15] Tri Dao、Daniel Y Fu、Stefano Ermon、Atri Rudra 和 Christopher Ré。2022年。FlashAttention：具备 I/O 感知能力的快速、内存高效精确注意力机制。arXiv 预印本 arXiv:2205.14135（2022年）。[16] Sheng Di 和 Franck Cappello。2016年。面向高性能计算应用的静默数据损坏自适应影响驱动检测。《IEEE 并行与分布式系统汇刊》27, 10（2016年），2809–2823。[17] Chong Ding、Christer Karlsson、Hui Liu、Teresa Davies 和 Zizhong Chen。2011年。具有在线容错能力的 GPU 矩阵乘法。载于《2011年 IEEE 第九届并行与分布式处理及应用国际研讨会》。IEEE，311–317。[18] Jack Dongarra、Pete Beckman、Terry Moore、Patrick Aerts、Giovanni Aloisio、Jean-Claude Andre、David Barkai、Jean-Yves Berthou、Taisuke Boku、Bertrand Braunschweig 等。2011年。国际百亿亿次级软件项目路线图。《高性能计算应用国际期刊》25, 1（2011年），3–60。[19] Al Geist。2016年。超级计算领域隐藏的怪物。《IEEE Spectrum》53, 3（2016年），30–35。[20] Leonardo Arturo Bautista Gomez 和 Franck Cappello。2015年。通过多变量插值检测并纠正模板应用中的数据损坏。载于《2015年 IEEE 集群计算国际会议》。IEEE，595–602。[21] John A Gunnels、Daniel S Katz、Enrique S Quintana-Orti 和 RA Van de Gejin。2001年。容错高性能矩阵乘法：理论与实践。载于《2001年可靠系统与网络国际会议》。IEEE，47–56。[22] Doug Hakkarinen、Panruo Wu 和 Zizhong Chen。2014年。面向 Cholesky 分解的停机故障基于算法的容错。《IEEE 并行与分布式系统汇刊》26, 5（2014年），1323–1335。[23] Jianyu Huang、Chenhan D Yu 和 Robert A van de Geijn。2020年。GPU 上的 Strassen 算法再探。《ACM 数学软件汇刊》（TOMS）49, 1（2020），1–22。[24] Kuang-Hua Huang 和 Jacob A Abraham。1984年。矩阵运算的基于算法的容错。《IEEE计算机汇刊》100, 6（1984年），518–528。[25] Yangqing Jia、Evan Shelhamer、Jeff Donahue、Sergey Karayev、Jonathan Long、Ross Girshick、Sergio Guadarrama 和 Trevor Darrell。2014年。Caffe：用于快速特征嵌入的卷积架构。载于第22届ACM国际多媒体会议论文集。675–678。[26] Jack Kosaian 和 KV Rashmi。2021年。算术强度引导的图形处理器神经网络推理容错。载于高性能计算、网络、存储与分析国际会议论文集。1–15。[27] Jean-Claude Laprie。1985年。可信计算与容错。FTCS-15论文摘要集（1985年），2–11。[28] Dong Li、Jeffrey S Vetter 和 Weikuan Yu。2012年。使用二进制插桩工具对极端规模科学应用中的软错误易受影响性进行分类。载于高性能计算、网络、存储与分析国际会议论文集。IEEE计算机学会出版社，57。[29] Guanpeng Li、Siva Kumar Sastry Hari、Michael Sullivan、Timothy Tsai、Karthik Pattabiraman、Joel Emer 和 Stephen W Keckler。2017年。理解深度学习神经网络（DNN）加速器及应用中的错误传播。载于高性能计算、网络、存储与分析国际会议论文集。ACM，8。[30] Sihuan Li、Hongbo Li、Xin Liang、Jieyang Chen、Elisabeth Giem、Kaiming Ouyang、Kai Zhao、Sheng Di、Franck Cappello 和 Zizhong Chen。2019年。FT-iSort：高效的内省排序容错。载于高性能计算、网络、存储与分析国际会议论文集。ACM，71。[31] Xin Liang、Jieyang Chen、Dingwen Tao、Sihuan Li、Panruo Wu、Hongbo Li、Kaiming Ouyang、Yuanlai Liu、Fengguang Song 和 Zizhong Chen。2017年。在快速傅里叶变换中在线纠正软错误。载于高性能计算、网络、存储与分析国际会议论文集。ACM，30。[32] Robert Lucas、James Ang、Keren Bergman、Shekhar Borkar、William Carlson、Laura Carrington、George Chiu、Robert Colwell、William Dally、Jack Dongarra 等。2014年。美国能源部高级科学计算咨询小组委员会（ASCAC）报告：百亿亿次级研究的十大挑战。技术报告。美国能源部科学办公室（美国）。[33] Robyn R Lutz。1993年。分析安全关键型嵌入式系统中的软件需求错误。载于 *[1993]* IEEE需求工程国际研讨会论文集。IEEE，126–133。[34] Timothy C May 和 Murray H Woods。1979年。动态存储器中的α粒子诱发软错误。《IEEE电子器件汇刊》26, 1（1979年），2–9。[35] Subhasish Mitra、Pradip Bose、Eric Cheng、Chen-Yong Cher、Hyungmin Cho、Rajiv Joshi、Young Moon Kim、Charles R Lefurgy、Yanjing Li、Kenneth P Rodbell 等。2014年。弹性之墙：跨层解决策略。载于2014年超大规模集成电路技术、系统与应用国际研讨会技术项目论文集（VLSI-TSA）。IEEE，1–11。[36] Michael Nicolaidis。1999年。基于时间冗余的软错误容忍，以应对纳米技术的挑战。载于第17届IEEE超大规模集成电路测试研讨会论文集（目录号 PR00146）。IEEE，86–94。[37] NVIDIA。2022年检索。https://github.com/NVIDIA/cutlass。在线。[38] Nahmsuk Oh、Philip P Shirvani 和 McCluskey。2002年。通过软件签名进行控制流检查。《IEEE可靠性汇刊》51, 1（2002年），111–122。[39] Nahmsuk Oh、Philip P Shirvani 和 Edward J McCluskey。2002年。通过超标量处理器中的重复指令进行错误检测。《IEEE可靠性汇刊》51, 1（2002年），63–75。[40] Daniel Oliveira、Laércio Pilla、Nathan DeBardeleben、Sean Blanchard、Heather Quinn、Israel Koren、Philippe Navaux 和 Paolo Rech。2017年。Xeon Phi 可靠性的实验与分析研究。载于高性能计算、网络、存储与分析国际会议论文集。ACM，28。[41] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Kopf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。2019年。PyTorch：命令式风格的高性能深度学习库。载于《神经信息处理系统进展》第32卷，H. Wallach、H. Larochelle、A. Beygelzimer、F. d'Alché-Buc、E. Fox 和 R. Garnett（编）。Curran Associates, Inc.，8024–8035。[42] James C Phillips、Rosemary Braun、Wei Wang、James Gumbart、Emad Tajkhorshid、Elizabeth Villa、Christophe Chipot、Robert D Skeel、Laxmikant Kale 和 Klaus Schulten。2005年。使用 NAMD 实现可扩展的分子动力学。《计算化学杂志》26, 16（2005年），1781–1802。[43] George A Reis、Jonathan Chang、Neil Vachharajani、Ram Rangan 和 David I August。2005年。SWIFT：软件实现的容错。载于国际代码生成与优化研讨会论文集。IEEE计算机学会，243–254。[44] Minsoo Rhu、Mike O'Connor、Niladrish Chatterjee、Jeff Pool、Youngeun Kwon 和 Stephen W Keckler。2018年。压缩 DMA 引擎：利用激活

稀疏性训练深度神经网络。载于*2018*年*IEEE*国际高性能计算机体系结构研讨会（*HPCA*）。IEEE，78–91。[45] Marc Snir、Robert W Wisniewski、Jacob A Abraham、Sarita V Adve、Saurabh Bagchi、Pavan Balaji、Jim Belak、Pradip Bose、Franck Cappello、Bill Carlson 等。2014年。应对百亿亿次级计算中的故障。《国际高性能计算应用期刊》28, 2（2014年），129–173。[46] Dingwen Tao、Sheng Di、Xin Liang、Zizhong Chen 和 Franck Cappello。2018年。通过有损检查点提升迭代方法的性能。载于第*27*届国际高性能并行与分布式计算研讨会论文集。52–65。[47] DL Tao 和 Carlos RP Hartmann。1993年。一种用于快速傅里叶变换网络的新型并发错误检测方案。《*IEEE* 并行与分布式系统汇刊》4, 2（1993年），198–221。[48] Dingwen Tao、Shuaiwen Leon Song、Sriram Krishnamoorthy、Panruo Wu、Xin Liang、Eddy Z Zhang、Darren Kerbyson 和 Zizhong Chen。2016年。New-sum：一种适用于通用迭代方法的新型在线算法容错方案。载于第*25*届*ACM*国际高性能并行与分布式计算研讨会论文集。ACM，43–55。[49] Panruo Wu 和 Zizhong Chen。2014年。FT-ScaLAPACK：在线纠正 ScaLAPACK Cholesky、QR 和 LU分解例程中的软错误。载于第*23*届国际高性能并行与分布式计算研讨会论文集。ACM，49–60。

[50] Panruo Wu、Chong Ding、Longxiang Chen、Teresa Davies、Christer Karlsson 和 Zizhong Chen。2013。矩阵乘法中的在线软错误纠正。《计算科学期刊》4，6（2013），465–472。[51] Jing Yu、Maria Jesus Garzaran 和 Marc Snir。2009。Esoftcheck：移除非关键检查以实现容错。载于《2009年国际代码生成与优化研讨会》。IEEE，35–46。[52] Yujia Zhai、Elisabeth Giem、Quan Fan、Kai Zhao、Jinyang Liu 和 Zizhong Chen。2021。FT-BLAS：具有在线容错功能的高性能基本线性代数子程序实现。载于《ACM国际超级计算大会论文集》。127–138。[53] Yujia Zhai、Mohannad Ibrahim、Yiqin Qiu、Fabian Boemer、Zizhong Chen、Alexey Titov 和 Alexander Lyashevsky。2022。加速英特尔图形处理器上的加密计算。载于《2022年IEEE国际并行与分布式处理研讨会（IPDPS）》。IEEE，705–716。[54] Yujia Zhai、Chengquan Jiang、Leyuan Wang、Xiaoying Jia、Shang Zhang、Zizhong Chen、Xin Liu 和 Yibo Zhu。2022。ByteTransformer：针对变长输入加速的高性能 Transformer。arXiv 预印本 arXiv:2210.03052（2022）。[55] Kai Zhao、Sheng Di、Sihuan Li、Xin Liang、Yujia Zhai、Jieyang Chen、Kaiming Ouyang、Franck Cappello 和 Zizhong Chen。2020。卷积神经网络的算法级容错。《IEEE 并行与分布式系统汇刊》（2020）。