面向科学计算内核的可移植扩展精度浮点运算

Manas Vishal数学系 马萨诸塞大学达特茅斯分校美国马萨诸塞州北达特茅斯
vishalmanas28@gmail.com

arXiv:2611.28959v1 [cs.MS] 21 Nov 2026

摘要—许多科学计算需要比 IEEE 双精度更高的准确度，但任意精度运算在 CPU 和 GPU 上往往代价过高。多浮点算术通过将每个值表示为若干 IEEE 数的未求值和，弥补了这一差距。现有实现通常局限于 CPU、CUDA 或特定语言的编程模型。

我们介绍 MultiFloats.cpp，这是一个仅含头文件的 C++17 库，使用 Kokkos 为 Serial、OpenMP、CUDA 和 HIP 编译相同的扩展精度内核。该库支持从 Float64x2 到 Float64x4（106–212 位）的类型，以及相应的基于 Float32 的类型。我们使用七种 SHOC 风格的内核对其进行评估，并在 A100、GH200 和 AMD RX 780M 系统上将其与四精度、CAMPARY、MPFR 和 Julia MultiFloats.jl 进行比较。在 A100 上，Float64x4 加法的吞吐量达到 15,700 百万次运算/秒，即$1.5\times$CAMPARY；在 CPU 上，向量化的 Float64x4 运算比4.2–$6.6\times$QD 的 qd_real 快。对 10,000 个输入进行测试时，加法和乘法的 ULP 误差均为 0，而 Float64x4 除法和平方根的误差分别不超过 1.71 和 17.9 ULP。因此，无需为不同后端分别实现，扩展精度也能兼具实用性和可移植性。

索引词—扩展精度、双双精度、四倍双精度、GPU 计算、Kokkos、可移植性、SHOC 基准测试、HIP

I. 引言

科学计算的准确度往往并非受限于离散化，而是受限于消去和浮点舍入误差的累积。例如，IEEE 754 Float64 的 53 位有效数无法分辨彼此抵消、相差超过约 21 个数量级的中间量。Rump 多项式 [16]，$f(a,\ b) = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + a/(2b)$，便是一个典型例子。在 (77617，33096) 处，Float64 返回 -1.18×10^{21}，尽管真实值约为 -0.8274。即使是 Float64x2（106 位）也会给出错误的符号，而 Float64x4（212 位）则能得到正确结果。补偿求和可以修正这个特定表达式，但通用科学计算代码需要在整个计算过程中始终保持准确的算术运算。这些限制并非只存在于人为构造的例子中：我们的极端质量比旋近引力波求解器和自力计算同样需要高于 Float64 [1]–[3] 的精度。

MPFR [5]提供任意精度，但通常会使硬件算术运算的开销增加 10–$100\times$ 倍。多浮点算术运算 [4] 提供了一种互补方案：将数值存储为若干 k IEEE 数的未求值之和；对于基于双精度的类型，每个数值约可获得 $53k$ 位精度，同时每次运算均由原生浮点指令实现。QD [6]、CAMPARY [8], 和 MultiFloats.jl [10] 已证明，这种表示方式在 CPU、CUDA GPU 以及 Julia 中都具有实用性。目前仍缺少一种编译型 C++实现，能够通过同一套源代码路径同时支持 NVIDIA 和 AMD GPU。

我们通过 MultiFloats.cpp 弥补这一空白。这是一个基于 Kokkos [12] 的仅头文件 C++17 库。后端选择在配置时完成，而算术类型和内核保持不变。本文的主要贡献如下：

1) **面向 CPU 扩展精度的显式 SIMD 类型**。我们引入 Vec<T,W>，这是一种结构数组表示形式，可向编译器公开彼此独立的多浮点运算。这避免了 GCC 无法对相应结构数组的 TwoSum 图进行向量化的问题，并使加法、乘法和高位除法的性能较 GCC 标量代码提升 1.9–$4.7\times$ 倍。2) **CUDA 和 HIP 共用一套源代码路径**。同一组 Kokkos 内核可编译用于 NVIDIA 和 AMD GPU。与原生 CUDA 直接比较后发现，Kokkos 在 A100 和 GH200 上的开销不超过 1.7%。3)**经过验证的扩展精度基准测试套件**。我们对七种 SHOC 风格内核进行了评估，涵盖 Float64x1–Float64x4 和 Float32x2–Float32x4，并使用 MPFR 进行 ULP 测量，同时将性能与四精度、CAMPARY 和 MultiFloats.jl 进行比较。

II. 背景

多浮点算术。我们从 $x = x_h + x_l$ 开始，其中$|x_l| \le \frac{1}{2}$ ulp(x_h)；四倍双精度将这一扩展扩展为四个分量。TwoSum 返回舍入后的和及其精确残差，而 TwoProd 对乘积执行相同操作 [6]、[7], [17]。随后，重新归一化生成观测到的 0- 末位单位加法和乘法结果。成本大致随 $O(k^2)$ 增长；在 GPU 上，更大的 k 会增加每次运算量和每个值所需的字节数，我们将在第 IV-E 节中分别分析这两种影响。

性能可移植性。Kokkos 为 Serial、OpenMP、CUDA、HIP 和 SYCL [12] 提供并行循环、归约和视图。每个内核都使用 KOKKOS_LAMBDA，因此其 C++源代码独立于所选的执行空间。

M. Vishal，“可移植扩展精度浮点运算：科学计算内核”，载于*2026* 年 *IEEE* 高性能极限计算会议（*HPEC*），2026 年。© 2026 IEEE。允许个人使用本材料。对于所有其他用途，无论是在当前还是未来的任何媒体中，包括为广告或促销目的重新印刷或再出版本材料、创建新的合集、转售或重新分发至服务器或列表，或在其他作品中重复使用本作品中受版权保护的任何组成部分，均必须获得 IEEE 的许可。

基准层次结构。 SHOC 根据算法复杂度 [13] 组织加速器工作负载。AXPY、点积、归约、Triad 和快速傅里叶变换用于衡量算术运算吞吐量和内存吞吐量，而通用矩阵乘法和稀疏矩阵-向量乘法则增加了数据复用和不规则访问。我们使用这一系列基准来研究独立操作以外的扩展精度。

III. 库实现

核心类型和基本操作。定长聚合类型 MultiFloat<T,N> 存储N IEEE 有效数字，无需分配内存或使用运行时多态；例如，Float64x2 是 MultiFloat<double,2> 的别名。KOKKOS_INLINE_FUNCTION 使每个运算符和数学函数都可在主机和设备上使用。

算术运算遵循 Hida[6]。Knuth加减法序列使 two_sum 不受 FMA 收缩影响，而 two_prod 通过 fma(a,b,-prod) 计算其残差，而不是采用 Dekker 分割。因此，MultiFloats.cpp 可以使用 -ffp-contract=fast，而 CAMPARY 的分割序列无法使用该选项。固定的 two-sum 扫描会对每个展开进行重新规格化；Float64x4 执行五次扫描，以控制 GPU 寄存器使用量，同时不影响观测到的 ULP 误差。

除法和平方根通过牛顿–拉夫逊迭代实现。默认的 Float64x4 平方根会在一次 $1\to2\to4$ 细化中跳过三分量阶段，观测到的最大误差为 17.9 ULP。sqrt_accurate 将倒数细化扩展至四个分量，在成本约为两倍的情况下，将最大误差降至 1.2 ULP（见第 IV-H 节）。

Kokkos集成。 可在设备上调用的操作让多精度浮点值直接存储于 Kokkos::View 中；因此，扩展精度 AXPY 仍保留原生的并行_for 形式。仅更改 CMake 后端，即可为 Serial、OpenMP、CUDA 或 HIP 编译此源代码。在同一会话中进行的原生 CUDA 对比显示，在 A100 和 GH200 上抽象开销低于 1.7%（第 IV-G 节）。

向量化CPU类型。 结构数组布局会使 GCC 无法识别独立的肢体操作。Julia 的 LLVM 流水线似乎能够恢复打包的 AVX指令，而 GCC 通常无法从内联的 TwoSum 图中做到这一点。我们通过 Vec<T,W> 暴露这种并行性；它以结构数组布局存储 W 扩展。在 AVX2 上，Vec4<Float64x2> 可在无需运行时转置的情况下处理四个值，使 Float64x2 加法和乘法性能提升 1.9–$2.2\times$，并且在更高肢体数下仍能与 Julia 保持竞争力（第 IV-B 节）。

IV. 实验结果

现在，我们将评估该库的性能，涵盖从单项算术运算到多核和 GPU 内核。表 I 总结了七种 SHOC 风格的工作负载，以及每项实验所用的参数。

A. 实验设置

硬件和软件。我们的主要系统配备一颗 16 核 AMD EPYC 7313 CPU（AVX2+融合乘加）

表 I 基准测试参数。表中报告的是三次独立运行中各自五次测量的最佳结果的中位数。

基准测试	N 重复次数*指标		布局
CPU 微基准	10K	1 纳秒/次运算	AoS
GPU 微基准	10M	50 百万次运算/秒	AoS
一级	1M	10 百万次运算/秒	AoS
GEMM	64–4096	1 GFLOPS	左侧布局
SpMV	100K	10百万次运算/秒	压缩稀疏行（每行 5 个非零元素）
FFT	256–1M	1 百万次运算/秒	数组结构（复数）
准确度	10K	1 末位单位统计（主机）	

*每次计时试验中的内核调用次数。

以及一块 NVIDIA A100 80 GB PCIe GPU。我们使用 GCC 13.3.0 和 Clang 21.1.8 编译 CPU 代码，编译选项为 -O3 -march=native -ffp-contract=fast；EPYC 的频率上限为 3.0GHz。软件栈包括 CUDA 12.9（NVHPC 25.7）、Kokkos 4.7.02、Julia 1.12.5（搭配 MultiFloats.jl v3.2.6 和 CUDA.jl 6.2.0）、QD v2.3.22（搭配 QD_IEEE_加法），以及作为准确度参考的 MPFR。我们还单独在搭载 ROCm 6.3.1 的 RX 780 M（RDNA 3）上测试 HIP 的可移植性。

CPU 协议。 对于 CPU 微操作，我们使用 taskset -c 0 将执行固定到一个核心，进行三秒的计算预热，并使用 $N = 10{,}000$ 计时五次试验。OpenMP 实验使用 $N = 1$M，采用紧密线程绑定，使用 1–16 个核心，每次试验调用内核十次。我们报告三次独立运行的中位数。

GPU 协议。对于 GPU 微操作，我们使用 N =10M，进行一次预热，每次试验调用 50 次内核，使用显式栅栏，并进行五次试验。内核实验在一级运算中使用 $N = 1$M，在稀疏矩阵-向量乘法中使用 100K 行，并将通用矩阵乘法的矩阵尺寸设为 64–4096。我们使用 nvidia-smi 验证设备处于静默状态，并再次报告三次独立运行的中位数。

输入。C++ 和 Julia 性能测试使用完全相同的确定性输入，而准确度测试使用带种子的随机值。我们归档每次运行级别的范围；中位数聚合可降低对 CPU 频率爬升以及小型 GPU 工作负载启动延迟变化的敏感性。

B. CPU 微操作延迟

我们首先单独分析算术运算本身的开销。表 II 比较了标量和 Vec4 MultiFloats.cpp 运算与四精度、MultiFloats.jl 及 MPFR 的性能。

测量结果表明，数据布局和编译器代码生成与算术算法同等重要。与 GCC 标量代码相比，Vec4 可将加法、乘法以及三肢或四肢除法的延迟提升 1.9–$4.7\times$ 倍。二肢除法是例外：保持正确性的 GCC 反向量化保护机制使 Vec4 延迟升至 3.58 ns/op，而 GCC 标量代码为 2.39 ns/op；不过，加法和乘法仍受益于显式布局。

在 Float64x4 上，Vec4 的加法、乘法和除法分别需要 6.09、5.37 和 42.20 ns/op，与 Julia 的 6.32、6.38 和 43.29 ns/op 相当，甚至更快。GCC 标量实现分别需要 23.12、24.14 和 197.15 ns/op，而 Clang 弥补了其中的大部分差距。因此，我们认为标量实现的差异很大程度上源于代码生成，而非

表 II CPU 微操作延迟（纳秒/操作）。$N = 10,000$，AMD EPYC 7313，单核，Turbo 预热。GCC 和 Clang 表示分别来自 GCC 13.3 和 Clang 21.1.8 的标量代码；Vec4 是由 GCC 编译的 SOA 路径。粗体：每个精度类别中各列的最佳（最快）结果。

Type	Mode	Add	Mul	Div	Sqrt
Float64x1	GCC	**0.21**	**0.21**	**0.30**	**2.29**
Float64x1	Clang	**0.21**	0.22	0.31	2.42
Julia Float64x1	—	**0.21**	**0.21**	0.31	2.30
MPFR 53 位	—	8.01	11.47	15.49	15.83
Float64x2	GCC	1.42	1.02	2.39	**3.67**
Float64x2	Clang	1.25	0.90	**1.33**	3.70
Float64x2	Vec4	**0.73**	**0.47**	3.58	3.71
Julia Float64x2	—	1.25	0.93	5.18	19.28
dd_real（四精度）	—	1.53	2.36	6.24	8.32
MPFR 128 位	—	8.11	17.06	46.34	118.4
Float64x3	GCC	11.73	7.42	99.53	125.02
Float64x3	Clang	10.98	6.42	99.16	**107.69**
Float64x3	Vec4	**2.65**	**1.70**	**24.22**	109.01
Julia Float64x3	—	3.57	2.48	25.72	117.38
MPFR 160位	—	9.87	23.60	67.89	120.0
Float64x4	GCC	23.12	24.14	197.15	199.69
Float64x4	Clang	6.30	6.30	42.69	195.65
Float64x4	Vec4	**6.09**	**5.37**	**42.20**	190.63
Julia Float64x4	—	6.32	6.38	43.29	189.62
qd_实数（QD）	—	38.85	35.51	218.17	802.0
256位 MPFR	—	13.52	31.96	79.08	**165.2**

表 III 基于FLOAT32 的类型的 CPU 微操作延迟（纳秒/次），协议与表 II 相同。粗体表示每个精度类别中各列的最佳（最快）结果。

Type	Mode	Add	Mul	Div	Sqrt
Float32x1	GCC	0.11	**0.10**	**0.12**	1.35
Float32x1	Clang	0.11	**0.10**	**0.12**	1.35
Julia Float32x1	—	**0.10**	**0.10**	**0.12**	**1.34**
Float32x2	GCC	0.70	0.53	2.11	**2.29**
Float32x2	Clang	2.18	0.94	**1.85**	2.34
Float32x2	Vec4	**0.41**	**0.27**	3.46	2.35
Julia Float32x2	—	0.55	0.40	2.46	18.00
Float32x3	GCC	12.06	7.97	128.30	**106.86**
Float32x3	Clang	10.56	6.75	104.22	112.23
Float32x3	Vec4	2.86	**1.77**	23.96	107.62
Julia Float32x3	—	**2.48**	1.95	**13.81**	112.21
Float32x4	GCC	21.97	22.61	191.70	191.74
Float32x4	Clang	22.94	21.90	209.23	213.00
Float32x4	Vec4	6.22	5.75	41.91	186.84
Julia Float32x4	—	**4.87**	**5.03**	**23.38**	**181.03**

底层表示。平方根受益较少，因为其牛顿迭代是顺序执行的。与四精度相比，Vec4 实现的四种 Float64x4 运算速度提升了 4.2–6.6× 倍。

表 III 中基于 Float32 的结果遵循相同的总体规律。在两个分量时，Vec4 加法耗时为 0.41 ns/op，而基于双精度的类型耗时为 0.73ns/op。然而，在分量数较多时，重新归一化成为主要开销，Float32 和 Float64 的标量运算成本趋于接近。Julia

表 IV 超越函数性能：CPU 延迟（纳秒/次运算，标量）和 GPU 吞吐量（百万次运算/秒，$N = 10$M，A100）。

平台	Type	exp	log	sinpi	cbrt
CPU（纳秒/次操作）	Float64x2	186.6	141.9	227.6	79.04
	Float64x3	873.9	625.4	822.7	259.7
	Float64x4	2202	1425	1786	477.1
	Float32x2	100.6	102.1	88.9	75.29
	Float32x3	456.2	392.9	461.4	343.9
	Float32x4	923.9	827.3	1074	468.8
GPU（百万次运算/秒）	Float64x2	10,952	2,025	10,945	24,937
	Float64x3	2,181	979	2,262	6,706
	Float64x4	757	557	768	2,992
	Float32x2	35,223	29,494	38,591	46,952
	Float32x3	7,640	2,084	8,272	13,399
	Float32x4	2,802	1,235	2,804	5,985

表 V EPYC 7313 上的 OpenMP 扩展性：Float32x4 吞吐量（百万次运算/秒）。

内核	1 个线程	4 个线程	16 个线程	加速比	Eff.
AXPY	17.2	68.1	259	15.1×	94%
DOT	17.3	67.6	256	14.8×	93%
归约	27.1	106.5	402	14.8×	93%
Triad	17.1	68.0	258	15.1×	94%
SpMV	19.2	76.6	290	15.1×	94%
FFT	2.6	10.3	34.3	13.2×	82%

$N = 1$M；每次试验调用 10 次，取 5 次中的最佳结果；3 次运行的中位数。

在若干三分量和四分量运算中表现领先，而显式 Vec4 路径在三肢乘法中仍然表现最佳。

C. CPU 超越函数

超越函数除了多字长基本运算外，还需要进行自变量约简和多项式求值。表 IV 报告了它们的 CPU 延迟和 A100 吞吐量。

我们观察到 $O(k^2)$–$O(k^3)$ 的有效开销，因为自变量约简和霍纳求值会以扩展精度反复执行。在 GPU 上，Float64x2 指数函数的吞吐量达到 10,952 百万次运算/秒，而 Float64x4 达到 757 百万次运算/秒。因此，$14.5\times$减速幅度大于加法的 $4.3\times$ 减速幅度，体现了多项式求值成本的增加。与 1024 位 MPFR 相比，Float64x2–Float64x4 的指数函数、对数函数和 sinpi 的最大误差为 8–38 末位单位，中位误差低于一个末位单位。共用函数的结果与 Julia 一致；我们独立实现的 sinpi 也呈现出相同的误差范围。

D. OpenMP 多核扩展性

接下来，我们考察每个值增加的算术运算是否仍能在多个 CPU 核心上实现良好的扩展性。我们使用 Kokkos::OpenMP 并采用紧密的线程绑定，在 1–16 个 EPYC 7313 核心上运行 SHOC 内核。表 V 报告 Float32x4 的吞吐量，表 VI 则给出全部八个精度级别下 16 线程的效率。

在使用两个或更多分量时，AXPY、点积、归约、Triad 和稀疏矩阵-向量乘法可保持 88–97% 的并行效率。多肢重新归一化提高了每字节执行的计算量，

表 VI EPYC 7313 上各内核与精度的 OpenMP 16 线程并行效率（$S_{16}/16$，百分比）。

	浮点数64×				浮点数32×			
内核	1	2	3	4	1	2	3	4
AXPY	54	88	97	96	83	94	94	94
DOT	85	93	93	93	86	92	91	93
归约	86	92	93	93	87	92	93	93
Triad	46	93	95	95	57	93	94	94
SpMV	78	92	93	93	79	92	94	94
FFT	20	32	57	74	44	61	69	82

$N = 1$M，16 个线程，close 绑定；3 次运行的最佳 5 个值的中位数。

表 VII GPU 吞吐量（百万次运算/秒）。NVIDIA A100 上的 $N = 10$M 个元素。粗体：各精度类别中的最佳结果。CAMPARY 基于双精度 limb 构建，因此基于Float32的行仅比较 MultiFloats.cpp 和 Julia。

Type 库	Add	Mul	Div	Sqrt
Float64x1 MF.cpp	**67,579**	**67,556**	**67,198**	**96,356**
Float64x1 Julia	64,568	65,045	64,978	93,522
Float64x2 MF.cpp	**34,786**	**34,773**	**34,456**	**51,182**
Float64x2 Julia	34,338	34,414	34,202	47,601
Float64x2 CAMPARY	34,238	34,231	34,127	49,282
Float64x3 MF.cpp	**23,041**	**23,172**	**14,863**	**14,779**
Float64x3 Julia	22,927	23,021	14,317	13,899
Float64x3 CAMPARY	16,466	16,713	10,001	8,969
Float64x4 MF.cpp	15,726	16,605	**7,099**	**7,275**
Float64x4 Julia	**16,581**	**17,189**	6,977	7,067
Float64x4 CAMPARY	10,468	10,520	6,601	6,468
Float32x1 MF.cpp	**120,837**	**120,843**	**119,254**	**160,214**
Float32x1 Julia	117,359	118,562	117,167	151,685
Float32x2 MF.cpp	**66,684**	**67,480**	**62,189**	**95,161**
Float32x2 Julia	64,382	64,648	61,249	72,572
Float32x3 MF.cpp	**44,939**	**45,513**	**28,052**	**28,517**
Float32x3 Julia	44,134	44,762	25,104	24,728
Float32x4 MF.cpp	33,623	33,673	**13,915**	**14,487**
Float32x4 Julia	**33,816**	**33,826**	12,994	13,199

因此，在原生精度下受带宽限制的内核会转向计算受限状态，其扩展性也更接近核心数量。快速傅里叶变换呈现出相同趋势：随着肢体数增加，Float64 系列的效率从 20% 提升至 74%，Float32 系列则从 44% 提升至 82%。我们将这种单调改善解读为以下证据：限制低精度情形的因素是内存流量，而非 Kokkos 同步开销。

E. GPU 吞吐量

随后，我们在 A100 上重复了微操作研究。表 VII 比较了 MultiFloats.jl 的四种基本运算，以及基于双精度的类型在 CAMPARY 中的表现。

在一个和两个分量时，加法和乘法的算术强度约为 0.42 浮点运算次数/字节，远低于 A100 的屋顶线拐点 4.85 浮点运算次数/字节。因此，这些运算受带宽限制，并且在 Float64x2 下，三种实现的结果相差不超过 2%。

表 VIII A100 GPU 内核吞吐量（百万次运算/秒）：一级运算/快速傅里叶变换使用 $N = 1$M，稀疏矩阵-向量乘法使用 100K 行。当规模为 $N = 1$M 时，低位 limb 工作集（≤ 24MB；FLOAT64X2 为 48 MB）可容纳在 40 MB 的 L2 缓存中或与其容量相当。这解释了 AXPY/Triad 流量所隐含的 2.1–2.5 TB/s 吞吐量为何超过高带宽内存 2.0 TB/s 的峰值；表 VII 给出了高带宽内存驻留时的 $N = 10$M 速率。

	Float64×N			
内核	x1	x2	x3	x4
AXPY	102,469	42,802	19,969	15,334
DOT	41,384	27,526	14,491	8,334
归约	43,070	31,606	18,998	10,969
Triad	102,637	29,311	19,832	15,102
SpMV	73,001	50,190	27,383	15,508
FFT	5,684	2,545	1,736	1,330
	Float32×N			
	x1	x2	x3	x4
AXPY	104,722	101,998	52,734	28,689
DOT	43,786	35,565	22,601	14,181
归约	45,537	37,649	25,645	18,145
Triad	104,569	99,630	36,276	25,792
SpMV	77,952	64,573	42,596	25,716
FFT	8,532	4,496	3,194	2,306

在这种情况下，平方根的性能可能优于加法，因为它只读取一个输入数组，而不是两个；这种差异反映的是内存流量，而非算法更高效。

随着肢体数增加，算术运算强度会凸显不同实现之间的差异。在 Float64x4 下，Multi- Floats.cpp 的速度比 CAMPARY 快 1.1–1.6× 倍；例如，加法的性能达到 15,726，而不是 10,468 百万次运算/秒。我们认为，这一差距很大程度上源于显式融合乘加残差的使用，使 MultiFloats.cpp 能够保留融合运算，而 CAMPARY 的 Dekker 分裂则要求使用 --fmad=false。对于全部四种 Float64x4 运算，MultiFloats.cpp 与 Julia 的性能差距均在 6% 以内，这与指令调度方面的细微差异相符。直接的原生 CUDA 对比表明，在 A100 和 GH200 上，Kokkos 抽象层的开销均低于 1.7%。

F. SHOC 内核性能

仅凭微操作吞吐量无法描述完整内核内部的算术运算行为。因此，图 1 和表 VIII 报告了所有七种 SHOC 风格工作负载在所支持精度级别下的表现。

内核结果区分了内存主导型行为和计算主导型行为。AXPY 和 Triad 的性能下降约为 7×Float64x4/Float64x1，其中部分额外的算术运算开销被内存访问所掩盖。DOT 和归约的性能下降了 4–5×，而稀疏矩阵-向量乘法的性能下降了 4.7×。

通用矩阵乘法对精度最为敏感，因为每次乘加运算都会执行完整的多分量运算图。在$N = 4096$，朴素的 A100 内核性能从 Float64x1 的 283 GFLOPS 降至 Float64x4 的 38.8 GFLOPS。采用 16×16共享内存分块后，Float64x1 的性能提升至 2,049 GFLOPS，Float64x2 提升至 333 GFLOPS，但对于

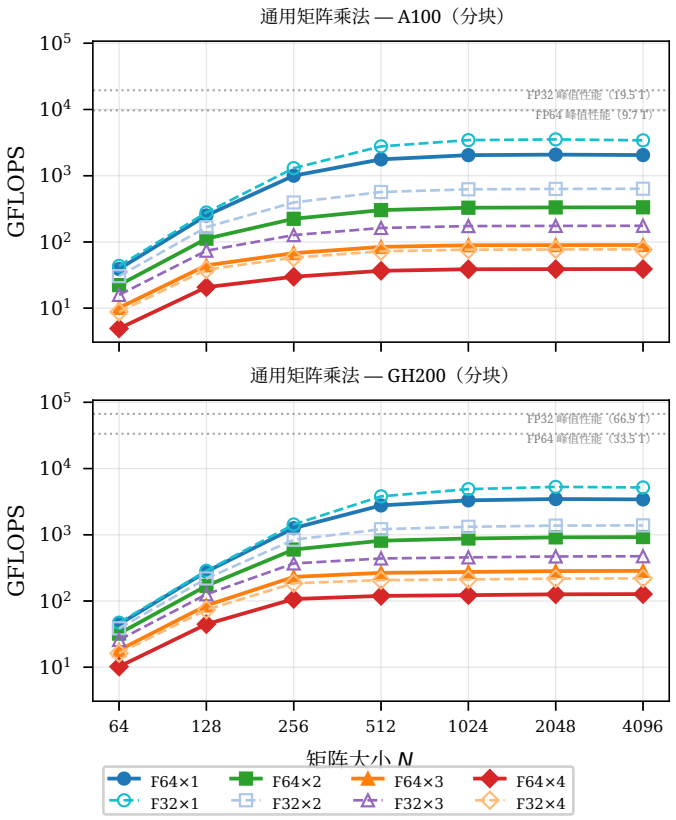


图 1. A100（上）和 GH200（下）上，基于 Float64 和 Float32 的类型在不同矩阵尺寸（$N = 64$–4096）下的分块 GEMM 吞吐量扩展。实线：Float64；虚线：Float32。灰色点线表示非张量核心 FP64/FP32 理论峰值性能。两款 GPU 使用相同的 Kokkos 源代码。

三分量和四分量配置，收益甚微，因为 $O(k^2)$算术运算的开销超过了数据重用带来的收益。在 GH200 上，相应分块配置下，Float64x1 和 Float64x4 的性能分别为 3,423 和127 GFLOPS。与 $N \leq 32$的 1,024 位 MPFR 对比时，Float64x2 的通用矩阵乘法误差仍低于 10 个末位单位，而 Float64x3– Float64x4 的 ULP 误差为 0.0。

快速傅里叶变换。对于快速傅里叶变换，我们预先计算扩展精度的旋转因子表，而不是在每次蝶形运算中计算 sincospi。在 $N = 1$M 规模下，Float64x1 到 Float64x4 的吞吐量从 5,684 降至 1,330 百万次运算/秒，性能降低4.3×倍。归档的单核测量结果显示，GPU 加速比从 74× 增至 649×，因为随着每次蝶形运算的开销增加，GPU 能够隐藏更多算术运算。与精度为 $O(N^2)$ 的 MPFR 离散傅里叶变换在$N = 256$下进行比较，平均误差分别为 5.4、3.2、1.5 和 1.0 末位单位，最大误差分别为 810、365、157 和 100 末位单位；这两项指标都随着肢体数增加而降低。

基于 Float32 的表示提供了一个实用的中间精度区间。对于带宽受限的 AXPY，Float32x2 的性能达到 101,998 百万次运算/秒，相当于原生 Float64x1 的 99%，同时提供约 48 位的有效数。其指数范围仍与 Float32 相同，因此适用于精度和带宽占主导的场景，但不能作为 Float64 的通用替代方案。

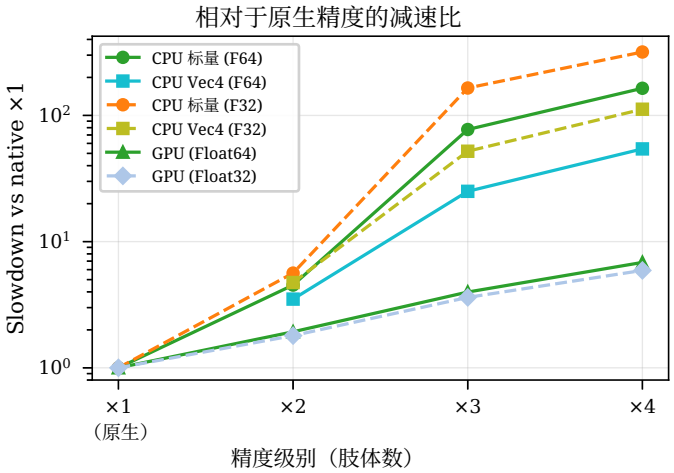


图 2。CPU 标量、CPU Vec4（均为基于 Float64 的类型）、GPU 基于 Float64 的类型和 GPU 基于 Float32 的类型相对于原生精度（Float64x1 或 Float32x1）的减速比。加法、乘法、除法和平方根的几何平均值。

G. 减速比与开销

图 2 总结了相对于原生精度的开销。在 A100 上，基于双精度的加法在两个、三个和四个分量时分别慢了 1.9×、2.9× 和 4.3× 。这些比率不应被视为纯粹的运算次数，因为原生基准受带宽限制；表 VII 中的绝对速率更能反映 GPU 性能。基于 Float32 的类型在四个分量时的几何平均减速为 5.9×，这与单精度融合乘加的开销相符。在 CPU 上，GCC 标量实现使用 Float64x4 时减速 87–657× ，而 Vec4 和 Clang 将这一范围缩小至 26–141×。这一差异再次表明，观测到的开销中有相当一部分来自编译器代码生成。

H. 准确度

我们使用 $[0.1, 1000]$ 中的 10,000 个随机输入，将 Float64x2–Float64x4 在其 $53k$ 位目标精度下与 1,024 位 MPFR 参考值进行评估（图 3）。在所有测试精度下，加法和乘法均产生 0.0 末位单位误差，验证了这些无误差变换。在 Float64x2 下，除法的最大误差为 2.15 末位单位（中位数为 0.11），而平方根达到 0.45 末位单位（中位数为 0.06）。在 Float64x4 下，相应的最大误差分别为 1.71 和 17.9 末位单位，中位数分别为 0.02 和 0.14 末位单位。

更大的 Float64x4 平方根最大误差源于该 $1 \to 2 \to 4$ 改进：跳过三字长步骤可使吞吐量近似翻倍，同时将观测到的误差保持在 18 个末位单位以内。sqrt_accurate 可将该最大误差降至 1.2 个末位单位，但代价是增加一倍。未进行 FMA 收缩时，MultiFloats.cpp 和 MultiFloats.jl 生成逐位相同的 Float64x3–Float64x4 输出。基于 Float32 的除法/平方根最大误差在两个分量时为 2.40/0.46 个末位单位，在四个分量时为 1.72/19.85 个末位单位。

底层 IEEE 表示带来了两项局限。次正规输入可能导致中间乘积下溢，而接近 Float64 溢出阈值的数值会降低

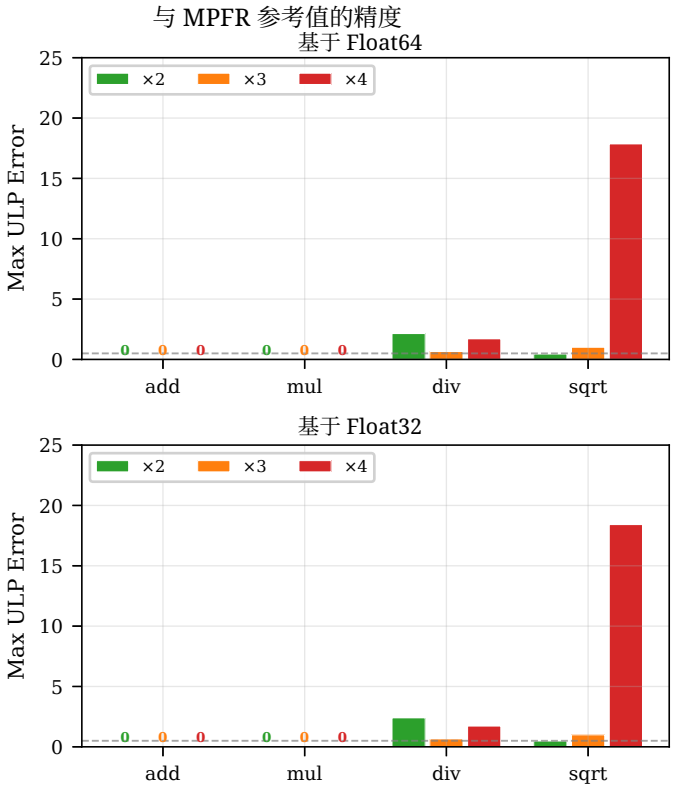


图 3. 基于 Float64（上）和基于 Float32（下）的类型相对于 1024 位 MPFR 参考值在目标精度下的 ULP 误差。

平方根细化的精度。我们还发现了编译器特有的敏感性：在所报告的优化标志下，Clang 21 的 SLP 向量化会破坏 Float64x2 快速傅里叶变换的低位分量。因此，我们仅针对 Clang 的快速傅里叶变换目标禁用 SLP。修正后的 Clang 构建通过了 MPFR 精度测试；本文报告的所有内核吞吐量数据均使用 CPU 上的 GCC 或 GPU 上的 CUDA 得出。

I. 跨厂商可移植性

最后，我们测试了编译之外的可移植性。只需更改 CMake 后端，完整测试套件即可通过 Kokkos::HIP 和 ROCm 6.3.1 在 AMD RX 780M 上运行。表 IX 对比了具有代表性的 Float64x2 CUDA 和 HIP 结果。尽管这些设备属于不同的硬件类别，相同的算术运算和内核仍可在两种环境中执行。

V. 相关工作

既有工作在平台范围、精度模型和语言运行时方面各不相同。

仅 CPU 库。Hida 等。 [6] 引入了 QD 的 dd_real 和 qd_real，作为我们的主要 CPU 基线；Float64x4 Vec4 的速度比 qd_real 快 4.2–6.6× 倍（表 II）。Mukunoki 等。 [14] 研究了双精度–单精度的性能权衡。

厂商专用 GPU 库。CAMPARY [8] 支持 CUDA 上可变数量的分量。MultiFloats.cpp 在 Float64x2 上与之性能相当，在 Float64x3 和 Float64x4 上分别快 1.4–1.6× 倍和 1.1–1.6× 倍

表 IX 跨厂商可移植性：NVIDIA CUDA 与 AMD HIP 上使用相同源代码的 FLOAT64x2 吞吐量（百万次运算/秒）。

内核	CUDA	HIP	比率
加法（微操作）	34,717	1,589	21.8×
乘法（微操作）	34,758	1,586	21.9×
AXPY	43,950	1,119	39.3×
DOT	28,113	952	29.5×
归约	26,226	1,086	24.1×
Triad	29,495	1,708	17.3×
SpMV	54,100	1,745	31.0×

A100 PCIe 与 RX 780M 集成显卡对比。两列数据分别在不同的测试活动中测得；A100 的主要中位数见表 VII 和表 VIII。这些比率源于硬件差距，而非代码质量。

（表 VII）。GQD [9] 将四精度引入 CUDA，而 Zhang 和 Aiken [11]为支撑 [10] 中无分支算法的浮点累加网络提供了计算机验证的误差界。

可复现累加。ExBLAS [15] 采用长累加器和补偿来实现按位可复现 BLAS，而非提高工作精度；这两种方法都能保留原生算术所丢弃的信息。

Ozaki 方案张量核心 GEMM。Ootomo 等人 [18]采用 Ozaki 分割，在整数张量核心上模拟双精度 GEMM [14]。对于合适的矩阵，其性能可超过原生 FP64 峰值性能，但准确度取决于条件数。相比之下，我们将一种 $O(k^2)$ 算术模型应用于 AXPY、快速傅里叶变换、稀疏矩阵-向量乘法、归约和通用矩阵乘法；Ozaki 调度仍是一项很有前景的扩展。

Julia MultiFloats.jl。MultiFloats.jl [10]通过 Julia 和 LLVM 实现了同一类算法。我们的 GPU 1.2×结果仍处于该范围内，CPU 性能也具有竞争力；而 MultiFloats.cpp 则以仅包含头文件的 C++17组件形式集成。

VI. 结论

我们展示了一个适用于 CPU/GPU 多浮点算术的 C++ 源代码。在 A100 上，Float64x4 加法的持续性能达到 15,700 百万次运算/秒（1.5× CAMPARY）；Float64x4 Vec4 比 QD 的 qd_real 快 4.2–6.6× 倍。在测试的输入上，加法和乘法的 ULP 误差均为 0，平方根的误差则不超过 17.9 ULP。一级内核在 16 个核心上达到约 93% 的效率；Kokkos 在 A100/GH200 上的开销低于 1.7%；且未经修改的源代码可通过 CUDA/HIP 运行。未来工作将聚焦于 Ozaki GEMM、SYCL、自适应精度，以及针对双黑洞问题的性能基准测试。

致谢

M.V. 感谢 Sigal Gottlieb、Gaurav Khanna、Aaron Jarmusch 和 Connor Kenyon 提供的见解，以及 Kokkos 和 MultiFloats.jl 的开发者。马萨诸塞大学达特茅斯分校通过 MGHPCC Unity 提供了 NVIDIA A100/GH200 的使用权限。代码、数据和制品位于 GitHub 仓库 https: //github.com/manasvishal/MultiFloats.cpp。

参考文献

[1] M. Vishal、S. E. Field、K. Rink、S. Gottlieb 和 G. Khanna，“迈向极端质量比旋近的指数收敛模拟：具有奇异源项的标量Teukolsky方程时域求解器”，*Phys. Rev. D*，第 110 卷，第 10 期，第 104009 页，2024 年。[2] M. Vishal、S. E. Field、S. Gottlieb 和 J. Ryan，“双曲面域上标量Teukolsky方程的超收敛间断Galerkin方法：高效波形与自力计算”，*General Relativity and Gravitation*，第 57 卷，第 7 期，第 104 页，2025 年。[3] M. Vishal，*PNP*问题：黑洞动力学中的精度与扰动：工程与应用科学博士论文，博士论文，马萨诸塞大学达特茅斯分校，2026 年。[4] D. H. Bailey，“科学计算中的高精度浮点运算”，*Computing in Science & Engineering*，第 7 卷，第 3 期，第 54–61 页，2005 年。[5] L. Fousse、G. Hanrot、V. Lef`evre、P. P´elissier 和 P. Zimmermann，“MPFR：支持正确舍入的多精度二进制浮点库”，*ACM Trans. Math. Software*，第 33 卷，第 15 期，2007 年，第 2、13 页。[6] Y. Hida、X. S. Li 和 D. H. Bailey，“四倍双精度浮点算术算法”，载于*Proc.15th IEEE Symp. Computer Arithmetic*，2001 年，第 155–162 页。[7] T. J. Dekker，“一种扩展可用精度的浮点技术”，*Numerische Mathematik*，第 18 卷，第 3 期，第 224–242 页，1971 年。[8] M. Joldes、J.-M. Muller、V. Popescu 和 W. Tucker，“CAMPARY：CUDA多精度运算库及其应用”，载于*Proc. 5th Int. Congress on Mathematical Software (ICMS)*，2016 年，第 232–240 页。[9] M. Lu、B. He 和 Q. Luo，“在图形处理器上支持扩展精度”，载于*Proc. 6th Int. Workshop on Data Management on New Hardware (DaMoN)*，2010 年，第 19–26 页。

[10] D. K. Zhang 和 A. Aiken，“用于扩展精度浮点运算的高性能无分支算法”，载于高性能计算、网络、存储与分析国际会议论文集（*SC*），2025 年，第 695–710 页，doi：10.1145/3712285.3759876。[11] D. K. Zhang 和 A. Aiken，“浮点累加网络的自动验证”，载于 第 *37* 届计算机辅助验证国际会议论文集（*CAV*），2025 年，第 215–237 页，doi：10.1007/978-3-031-98682- 6 12。[12] C. R. Trott 等，“Kokkos 3：面向百亿亿次计算时代的编程模型扩展”，*IEEE* 并行与分布式系统汇刊，第 33 卷，第 4 期，第 805–817 页，2022 年。[13] A. Danalis 等，“可扩展异构计算（SHOC）基准测试套件”，载于第 *3* 届通用图形处理器计算研讨会论文集（*GPGPU*），2010 年，第 63–74 页。[14] D. Mukunoki、T. Ogita 和 K. Ozaki，“采用双精度-单精度-双精度浮点格式的 Ozaki方案进行准确且高效的计算”，数值算法，第 83 卷，第 1431–1453 页，2020 年。[15] R. Iakymchuk、S. Collange、D. Defour 和 S. Graillat，“ExBLAS：可复现且快速的 BLAS”，载于*IEEE* 第 *22* 届计算机算术研讨会论文集，2015 年，第 30–37 页。[16] S. M. Rump，“验证包含的算法——理论与实践”，载于计算中的可靠性，R. E. Moore 编，Academic Press，1988 年，第 109–126 页。[17] M. Joldes、J.-M. Muller 和 V. Popescu，“双字运算基本构件的紧致且严格误差界”，*ACM* 数学软件汇刊，第 44 卷，第 2 期，第 15:1–15:27 页，2017 年。[18] H. Ootomo、K. Ozaki 和 R. Yokota，“整数矩阵乘法单元上的 DGEMM”，高性能计算应用国际期刊，第 38 卷，第 4 期，第 289–306 页，2024 年。