Accelerating Matrix Multiplication in Multiple Double Arithmetic with Tensor Cores of a NVIDIA A100 Graphics Processing Unit

Jan Verschelde
(joint with Howard Chen)

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
https://github.com/janverschelde
janv@uic.edu

SIAM Conference on Parallel Processing for Scientific Computing
Berlin, 3-6 March 2026

Accurate Matrix Multiplication on Tensor Cores

1. Problem Statement
   - robust numerical algorithms
   - multiple double arithmetic
   - NVIDIA A100 tensor cores FP64
2. Data Staging Algorithms
   - splitting fractions of doubles
   - rewriting products of splitted matrices
   - error free transformations
3. Computational Experiments
   - staging data to apply `dmmaTensorCoreGemm`
   - conclusions and future directions

introduction

Definition (robust)

An algorithm is *robust* if it does not fail for small perturbations of degenerate inputs.

Floating-point arithmetic with 64-bit doubles can be extended to gain more accuracy than what only hardware arithmetic gives.

- Algorithms to extend 32-bit floating-point arithmetic originated in the late sixties [Dekker, *Numerische Mathematik* 1971].
- The arithmetic is provided in software packages such as
  - QDlib [Hida, Li, Bailey, 2001], and
  - CAMPARY [Joldes, Muller, Popescu, Tucker, 2016].

Quality Up: Apply acceleration on Graphics Processing Units (GPUs) to offset the cost overhead of multiple double arithmetic.

multiple double arithmetic

Definition (multiple double)

A *multiple double* is an unevaluated sum of nonoverlapping doubles.

Take 64 random complex numbers on the unit circle.
The 2-norm of this vector is 8, computed with multiple doubles:

```
double double : 8.00000000000000E+00 - 4.46815747097839E-32
  quad double : 8.00000000000000E+00 + 8.23258305145073E-65
  octo double : 8.00000000000000E+00 - 5.56764060802733E-128
  hexa double : 8.00000000000000E+00 - 1.54394135726410E-257
```

Cost overhead	add	mul	div	avg
2	20	23	70	37.7
4	89	336	893	439.3
8	269	1742	5126	2379.0
16	925	11499	33041	15155.0

The table lists the number of operations with doubles for a multiple double addition (`add`), multiplication (`mul`), and division (`div`).

NVIDIA A100 Tensor Cores FP64

Definition (NVIDIA Tensor Core)

An *NVIDIA Tensor Core* is a specialized high-performance compute core for matrix multiplication.

- 8th generation data center GPU, introduced in 2020.
- IEEE-compliant FP64 tensor core instructions for HPC, FP64 is short for 64-bit floating point.

Theoretical peak performance of FP64 tensor cores: 19.5 TFLOPS, theoretical peak performance of FP64 cores: 9.7 TFLOPS.

The `dmmaTensorCoreGemm` program of the CUDA samples multiplies a random 8192-by-4096 with a random 4096-by-8192 matrix in 42 milliseconds, achieving a performance of 12.96 FP64 TFLOPS.

Are tensor cores useful for multiple double arithmetic?

problem statement and relation to mixed precision

Problem Statement

Multiple double arithmetic requires a renormalization after every arithmetical operation. This renormalization involves branching.

Our solution is similar to the application of the Ozaki scheme [K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump, *Numer. Algor* 2012].

Connection to mixed precision:

- In mixed precision, the goal is to obtain double precision accuracy with lower precision floating-point or integer processors.
- With multiple double precision, we aim for high precision accuracy with double precision floating-point processors.

splitting fractions of doubles

The fraction `f` of a double x has 52 bits,
and we can distribute the bits into 4 parts `f0`, `f1`, `f2`, `f3`, each of 13 bits:

```
 f : 1001 0001 1010 0110 0110 0100 1110 1101 0010 0011 0100 1100 1100
f0 : 1001 0001 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
f1 : 0000 0000 0000 0110 0110 0100 1100 0000 0000 0000 0000 0000 0000
f2 : 0000 0000 0000 0000 0000 0000 0010 1101 0010 0010 0000 0000 0000
f3 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0100 1100 1100
```

Then we compute with 4 doubles `x0`, `x1`, `x2`, `x3`, we call them *quarters*,
with exponents `0`, `-13`, `-26`, `-39`.

```
x0 : 1 0010 0011 0100 0000000000000000000000000000000000000000000000000     0
x1 : 1 1001 1001 0011 0000000000000000000000000000000000000000000000000   -13
x2 : 1 0110 1001 0001 0000000000000000000000000000000000000000000000000   -26
x3 : 1 0100 1100 1100 0000000000000000000000000000000000000000000000000   -39
```

The zero bits at the end allow for error free computations:

- Multiplying two quarters leaves at least 26 zero bits at the end.
- Long sequences of inner products can be made *without loosing any bits.*

rewriting products of splitted matrices, first way

We focus on the four quarters of the leading doubles:

$$a_{i,k} = (a_{i,k,0}, a_{i,k,1}, a_{i,k,2}, a_{i,k,3}), \quad b_{k,j} = (b_{k,j,0}, b_{k,j,1}, b_{k,j,2}, b_{k,j,3}),$$

mapped to $A_{i,k}$ and $B_{j,k}$ as

$$A_{i,k} = \begin{bmatrix} a_{i,k,0} & 0 & 0 & 0 \\ a_{i,k,1} & a_{i,k,0} & 0 & 0 \\ a_{i,k,2} & a_{i,k,1} & a_{i,k,0} & 0 \\ a_{i,k,3} & a_{i,k,2} & a_{i,k,1} & a_{i,k,0} \end{bmatrix}, \quad B_{k,j} = \begin{bmatrix} b_{k,j,0} \\ b_{k,j,1} \\ b_{k,j,2} \\ b_{k,j,3} \end{bmatrix},$$

contributing to the product

$$C_{i,j} = \begin{bmatrix} a_{i,k,0}b_{k,j,0} \\ a_{i,k,1}b_{k,j,0} + a_{i,k,0}b_{k,j,1} \\ a_{i,k,2}b_{k,j,0} + a_{i,k,1}b_{k,j,1} + a_{i,k,0}b_{k,j,2} \\ a_{i,k,3}b_{k,j,0} + a_{i,k,2}b_{k,j,1} + a_{i,k,1}b_{k,j,2} + a_{i,k,0}b_{k,j,3} \end{bmatrix}.$$

dimensions of the rewritten products, first way

Let $AA = (A_h, A_\ell)$ and $BB = (B_h, B_\ell)$ be two double matrices:

$$AA \text{ is } M\text{-by-}K \quad \text{and} \quad BB \text{ is } K\text{-by-}N.$$

Each double double is splitted in 12 parts:

1. leading doubles x_h are quartered, and
2. second doubles x_ℓ are split in eight parts, $52 = 4 \times 7 + 4 \times 6$.

As each element of AA is replaced by a 12-by-12 matrix, and each element of BB is replaced by a 12-by-1 matrix, after the rewrite, the dimensions of the matrices A and B are

$$A \text{ is } 12M\text{-by-}12K \quad \text{and} \quad B \text{ is } 12K\text{-by-}N.$$

Running example: $(M, K, N) = (8192, 4096, 8192)$ on tensor cores.
Input double double AA is 682-by-341 and BB is 341-by-8192,
the product requires $682 \times 8192 = 5,586,944$ inner products.

rewriting products of splitted matrices, second way

We focus on the four quarters of the leading doubles:

$$a_{i,k} = (a_{i,k,0}, a_{i,k,1}, a_{i,k,2}, a_{i,k,3}), \quad b_{k,j} = (b_{k,j,0}, b_{k,j,1}, b_{k,j,2}, b_{k,j,3}),$$

mapped to $A_{i,k}$ and $B_{j,k}$ as

$$A_{i,k} = \begin{bmatrix} a_{i,k,0} \\ a_{i,k,1} \\ a_{i,k,2} \\ a_{i,k,3} \end{bmatrix}^T, \quad B_{k,j} = \begin{bmatrix} b_{k,j,0} & b_{k,j,1} & b_{k,j,2} & b_{k,j,3} \\ 0 & b_{k,j,0} & b_{k,j,1} & b_{k,j,2} \\ 0 & 0 & b_{k,j,0} & b_{k,j,1} \\ 0 & 0 & 0 & b_{k,j,0} \end{bmatrix},$$

contributing to the product

$$C_{i,j} = \begin{bmatrix} a_{i,k,0}b_{k,j,0} \\ a_{i,k,0}b_{k,j,1} + a_{i,k,1}b_{k,j,0} \\ a_{i,k,0}b_{k,j,2} + a_{i,k,1}b_{k,j,1} + a_{i,k,2}b_{k,j,0} \\ a_{i,k,0}b_{k,j,3} + a_{i,k,1}b_{k,j,2} + a_{i,k,2}b_{k,j,1} + a_{i,k,3}b_{k,j,0} \end{bmatrix}.$$

dimensions of the rewritten products, second way

Let $AA = (A_h, A_\ell)$ and $BB = (B_h, B_\ell)$ be two double matrices:

$$AA \text{ is } M\text{-by-}K \quad \text{and} \quad BB \text{ is } K\text{-by-}N.$$

Each double double is splitted in 12 parts:

1. leading doubles x_h are quartered, and
2. second doubles x_ℓ are split in eight parts, $52 = 4 \times 7 + 4 \times 6$.

As each element of AA is replaced by a 1-by-12 matrix, and
each element of BB is replaced by a 12-by-12 matrix,
after the rewrite, the dimensions of the matrices A and B are

$$A \text{ is } M\text{-by-}12K \quad \text{and} \quad B \text{ is } 12K\text{-by-}12N.$$

Running example: $(M, K, N) = (8192, 4096, 8192)$ on tensor cores.
Input double double AA is 8192-by-341 and BB is 341-by-682,
the product requires $8192 \times 682 = 5,586,944$ inner products.

overview of the algorithm

The double double matrix multiplication on tensor cores runs through the following steps:

1. Split all double doubles in 12 parts.
2. Rewrite into one single matrix product of double matrices.
3. Run the matrix multiplication on the tensor cores.
   Remains the most computationally intensive step.
4. Extract the parts from the product.
5. Add the parts of the product into a double double matrix.

The last step must be done in double double arithmetic, summing 12 doubles into one double double for each element of the product.

This double double summation is massively parallel and can be executed on regular cores.

The main question then remains: *Is the result accurate?*

grouping numbers by sign and size

- *Subtracting floating-point numbers of the same magnitude may lead to losing many significant bits in the fraction.*

  Therefore, group elements by sign as follows:
  - let A_+ contain all positive numbers of A, and
  - let A_- contain all negative numbers of A,

  so $A = A_+ + A_-$, and likewise $B = B_+ + B_-$, then

$$AB = (A_+ + A_-)(B_+ + B_-) = A_+B_+ + A_+B_- + A_-B_+ + A_-B_-.$$

- *Adding numbers with exponents of different magnitude may lead to losing many significant bits in the fraction.*

  Therefore, write A as $A_0 + A_1 + \cdots + A_{12}$, where the index is the exponent of the leading quarter of the double.
  Do likewise for B and execute many matrix multiplications.

equalizing the signs of the doubles

A double double x is representated by

- a high double x_h, and
- a low double x_ℓ.

Problem: x_h and x_ℓ have opposite signs.

Assuming $x_h > 0$ and the exponent of x_h is zero:

1. $b := 1 \times 2^{-52}$
2. $x_h := x_h - b$
3. $x_\ell := x_\ell + b$

Properties:

1. As $x_\ell < 0$, and moreover $x_\ell < -1 \times 2^{-52}$, we have $x_\ell + b > 0$.
2. As x_ℓ grows in size, we may loose the last bit of x_ℓ.

Subtracting and adding a bit b generalizes for any multiple double.

balancing the exponents of the doubles

If we quarter a double x in four parts: x_0, x_1, x_2, and x_3,
we expect the exponents to be 0, -13, -26, and -39, respectively.

Problem: x_1, x_2, and x_3 may have many leading zeros in their fractions.

Therefore, their exponents may be much less than -13, -26, and -39,
in the worst case: -25, -38, and -51, causing great loss of accuracy.

If the exponents of x_0 and x_1 differ by more than 13, do as follows:

1. $b := 1 \times 2^{-14}$
2. $x_0 := x_0 - b$
3. $x_1 := x_1 + b$

Properties:

1. As the exponent of x_0 is zero, subtracting b does not alter this.
2. As the exponent of x_1 is less than -13, $x_1 + b$ has exponent -13.

staging data to apply `dmmaTensorCoreGemm`

Using CUDA Software Development Kit, `nvcc` version 12.4.

- Microway GPU Server with NVIDIA Ampere A100 80GB GPU.
  Rocky Linux 9.3; host code compiled with `g++` 11.4.1
  Using shared memory, reached 13.75 FP64 TFLOPS.
- NVIDIA RTX 4080 housed in 2023 Alienware gaming laptop.
  Windows 11; 2022 Community Visual Studio version.
  The FP64 Tensor Cores on RTX 4080 give 0.50 TFLOPS.

Software under GPU GPL license, on `https://github.com/`

- `janverschelde/PHCpack/src/GPU/TensorCores`
- `hchen221/multiple-double-arithmetic-matrix-matrix-multiplication`

still under development . . .

conclusions and future directions

- Postponing normalizations allows for better use of floating-point pipelines and benefits also sequential processing.
- Promising for double double arithmetic.
- For multiple double arithmetic, tradeoffs:
  - $-$ more parts may lead to more exponent shifts,
  - $+$ smaller problems can already fully occupy tensor cores.
- The FP64 cores have to be programmed to collaborate with the FP64 tensor cores, for best performance.