Precision Redefined: Unlocking and Delivering the Full Power of Modern GPUs for Scientific Computing

Harun Bayraktar, Senior Director – Math Libraries Engineering (presenter)

& a long list of colleagues who contributed to this work

Fast and Accurate Numerical Linear Algebra on Low-Precision Hardware: Algorithms and Error Analysis
PASC25 | June 19th, 2025 | Brugg, Switzerland

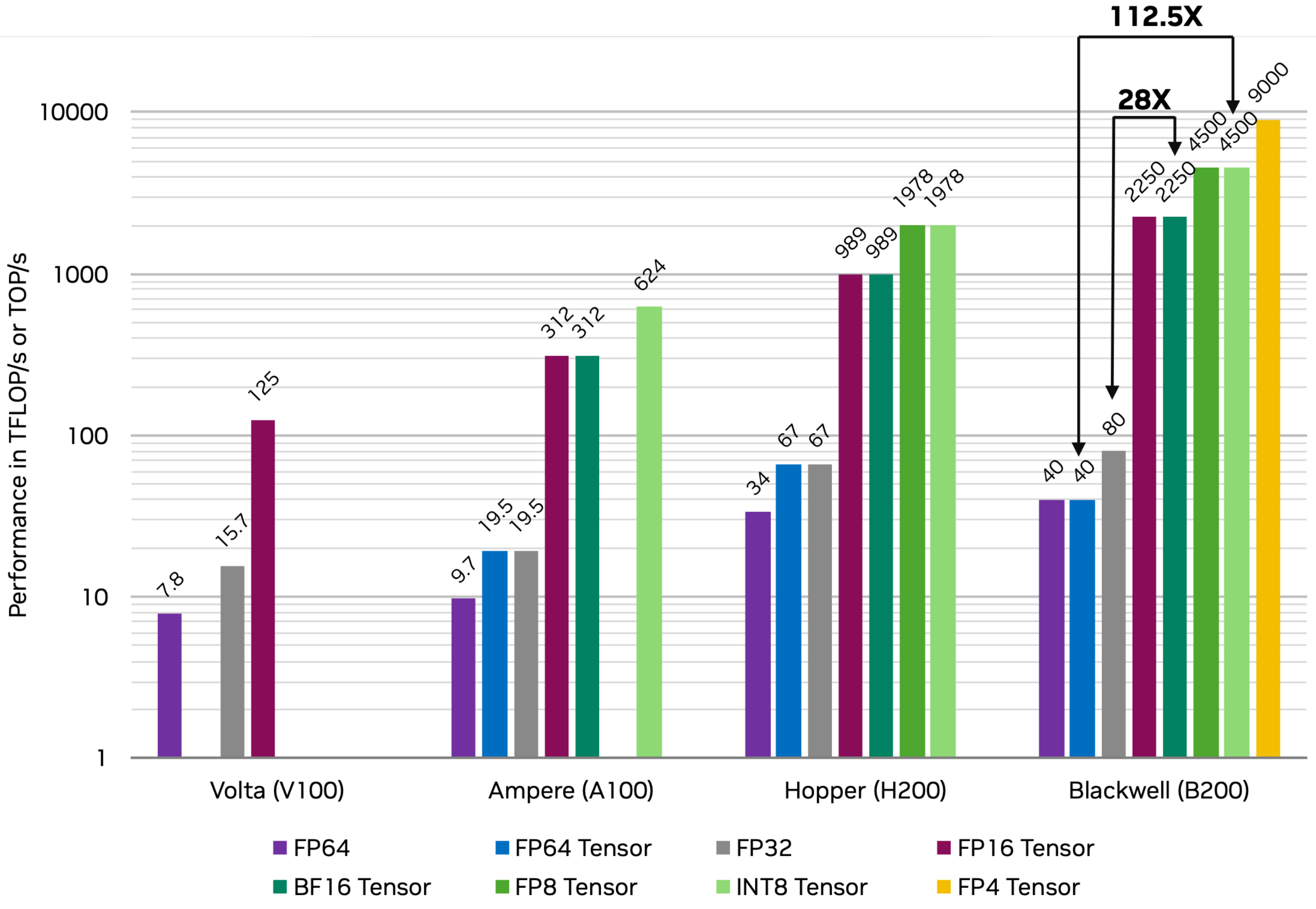
Agenda

- Motivation, History, and Productization Status
- Device Extension Libraries & Emulation Samples on Github
- FP64 Matrix Multiplication Emulation in cuBLAS
- Automatically Determining Emulation Parameters: Exponent Span Capacity (ESC) Algorithm
- Grading the cuBLAS DGEMM Implementation
- Closing Remarks & Future Work

Evolution of Peak Performance

Leveraging High Performance and Energy-Efficient Hardware for Higher Precision

- Architectures will have to accommodate both AI and scientific computing even as the fields become increasingly intertwined[1]
- Can we leverage reduced precision tensor cores to:
  - Accelerated mixed-precision algorithms?
  - Emulate FP64 and FP32 matrix multiplies without sacrificing accuracy for a performance gain?
  - **Can we realize the higher perf/Watt gain for a wide range of applications?**
  - **Can we do this in a non-intrusive way (i.e., not require any code changes)?**



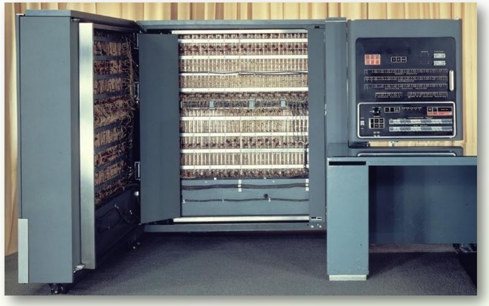
1 Hardware Trends Impacting Floating-Point Computations In Scientific Applications https://arxiv.org/abs/2411.12090

Historical Perspective on Emulation

The evolution of floating-point (FP) computation

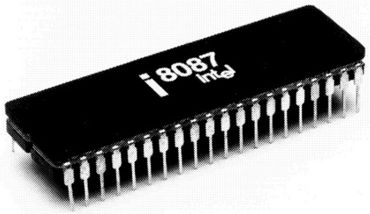
1950s
Simulated floating-point arithmetic utilizing fixed-point representations

IBM 701 Speedcoding System



1960s-70s
IBM Hexadecimal FP
Cray FP

Diversity in representations

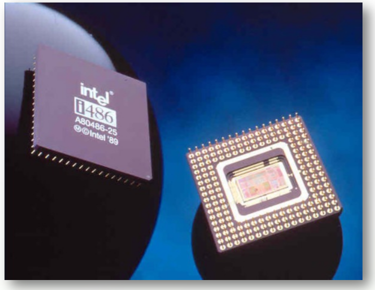


1980
FPU Coprocessor

Intel 8087

1985
IEEE 754

Standardization of FP

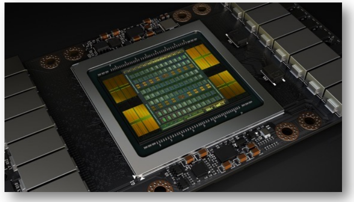


1989
Integrated FPU

Intel i486

2001
GPUs with programmable shaders

NVIDIA GeForce3



2017+
GPU Tensor Cores introduced for for reduced & mixed-precision

FP16, BF16, TF32, FP8, NVFP4, MXFP8

Today
FP emulation returns (e.g., Ozaki-I & II)

GPU Tensor Cores Accelerated Matrix Multiplication using AI FP types

Emulation Methods for Matrix Multiplication

- **FP32 using BF16 Tensor Cores** [1]
  - Tested for accuracy and performance impact in:
    - Weather, quantum circuit and condensed matter simulations
    - Dense Linear Algebra (QR, LU)
  - Uses 9 inner matrix multiplies in BF16 (BF16x9)
  - **Released with CUDA 12.9 for Blackwell GPUs**

- **FP64 using INT8 Tensor Cores** [2]
  - Coming soon for Ampere, Ada, Hopper, and Blackwell GPUs
  - Being tested for accuracy and performance impact in:
    - Materials Science, Electronic Structure
    - Molecular Dynamics, Computational Chemistry
    - HPL, Dense Linear Algebra (QR, LU)
  - Uses a variable number of inner matrix multiplies in INT8

Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance

Hiroyuki Ootomo[1] and Rio Yokota[2]

[1]School of Computing, Tokyo Institute of Technology
ootomo.h@rio.gsic.titech.ac.jp
[2]Global Scientific Information and Computing Center, Tokyo Institute of Technology
rioyokota@gsic.titech.ac.jp

Abstract

Tensor Core is a mixed-precision matrix-matrix multiplication unit on NVIDIA GPUs with a theoretical peak performance of more than 300 TFlop/s on Ampere architectures. Tensor Cores were developed in response to the high demand of dense matrix multiplication from machine learning. However, many applications in scientific computing such as preconditioners for iterative solvers and low-precision Fourier transforms can exploit these Tensor Cores. To compute a matrix multiplication on Tensor Cores, we need to convert input matrices to half-precision, which results in loss of accuracy. To avoid this, we can keep the mantissa loss in the conversion using additional half-precision variables and use them for

DGEMM on Integer Matrix Multiplication Unit

Hiroyuki Ootomo
ootomo.h@rio.gsic.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Katsuhisa Ozaki
ozaki@sic.shibaura-it.ac.jp
Shibaura Institute of Technology
Saitama, Japan

Rio Yokota
rioyokota@gsic.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

Deep learning hardware achieves high throughput and low power consumption by reducing computing precision and specializing in matrix multiplication. For machine learning inference, fixed-point value computation is commonplace, where the input and output values and the model parameters are quantized. Thus, many processors are now equipped with fast integer matrix multiplication units (IMMU). It is of significant interest to find a way to harness these IMMUs to improve the performance of HPC applications while maintaining accuracy. We focus on the Ozaki scheme, which computes a high-precision matrix multiplication by using lower-precision computing units, and show the advantages and disadvantages of using IMMU. The experiment using integer Tensor Cores shows that we can compute double-precision matrix multiplication faster than cuBLAS and an existing Ozaki scheme implementation on FP16 Tensor Cores on NVIDIA consumer GPUs. Furthermore, we demonstrate accelerating a quantum circuit simulation by up to

and FP32, the inference involves quantizing model parameters and input/output values to use lower-bit-length integer formats. This reduces data size, processor circuit area, and power consumption [14, 16]. As a result, high-performance processors like NVIDIA GPUs and edge devices like Google Coral Edge TPUs [33] are equipped with low-bit-length integer matrix multiplication units (IMMU). For instance, NVIDIA GPUs provide a DP4A instruction that can efficiently compute the inner product of two length-4 8-bit integer (INT8) vectors and accumulate the result in a 32-bit integer (INT32). In addition, NVIDIA Tensor Cores support the multiplication of INT8 matrices with INT32 accumulation from the Turing architecture and the multiplication of 4-bit integer (INT4) matrices from the Ampere architecture. These integer Tensor Cores can achieve a theoretical peak performance that is 2 ~ 4 times faster than floating-point Tensor Cores. Other processors, such as Google TPU v1 [18], Intel AMX-INT8 [5], and Groq TSP [2], also support the multiplication of INT8 matrices with INT32 ac-

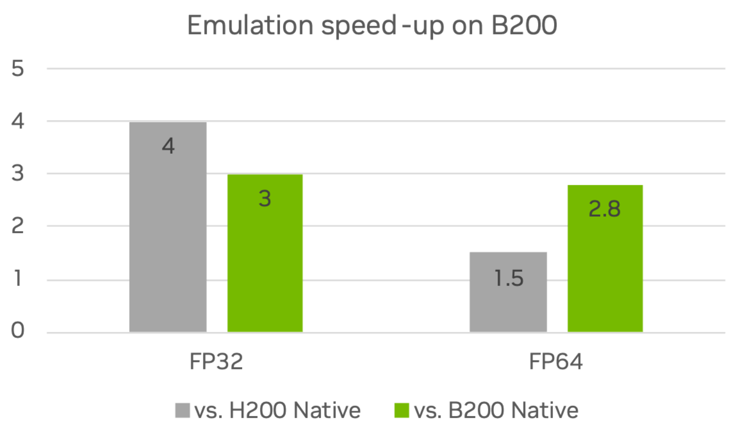
1 https://arxiv.org/pdf/2203.03341 and https://arxiv.org/pdf/1904.06376
2 https://arxiv.org/abs/2306.11975 and https://arxiv.org/abs/2409.13313
See GTC Session Energy-Efficient Supercomputing Through Tensor Core-Accelerated Mixed-Precision Computing and Floating-Point Emulation [S71487]

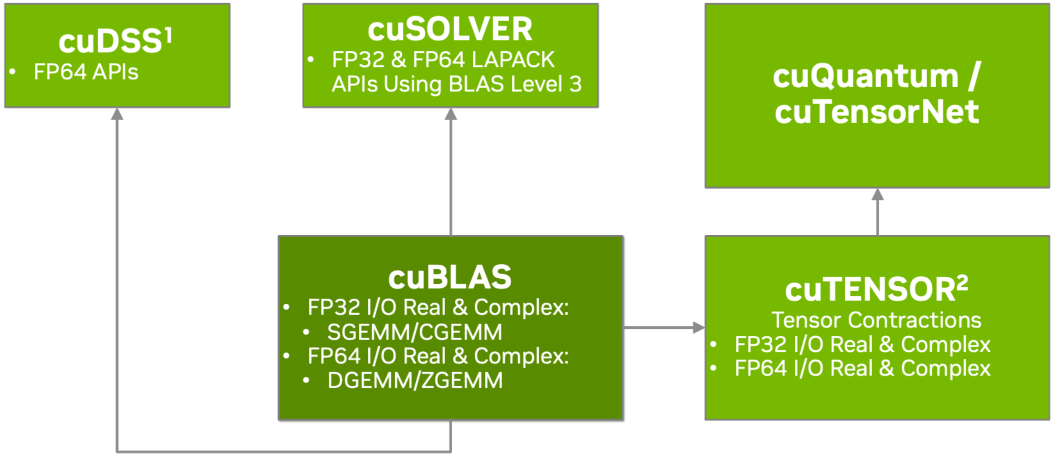
Productization of Emulation for Matrix Multiplications

Single- and Double-Precision Matrix Multiplications

- Initial release with cuBLAS as an **opt-in**
  - If you opt-in it will also apply to libraries that depend on cuBLAS where applicable
  - Environment variables and new APIs allow full control of emulation parameters
    - `CUBLAS_EMULATE_SINGLE_PRECISION=1|0`
    - `cublasEmulationStrategy_t`
  - **Safeguards are in place to fallback to native** HW-based kernels to guarantee results are always correct and corner cases are handled
- Single-precision (FP32)
  - **Publicly released with CUDA 12.9 May 2025**
- Double-precision (FP64)
  - First with Ozaki-I method
  - Release planned for second half of 2025
- Future releases will
  - Add opt-in through other libraries
  - After enough exposure will **switch to opt-out**

[Energy-Efficient Supercomputing Through Tensor Core-Accelerated Mixed-Precision Computing and Floating-Point Emulation [S71487]]





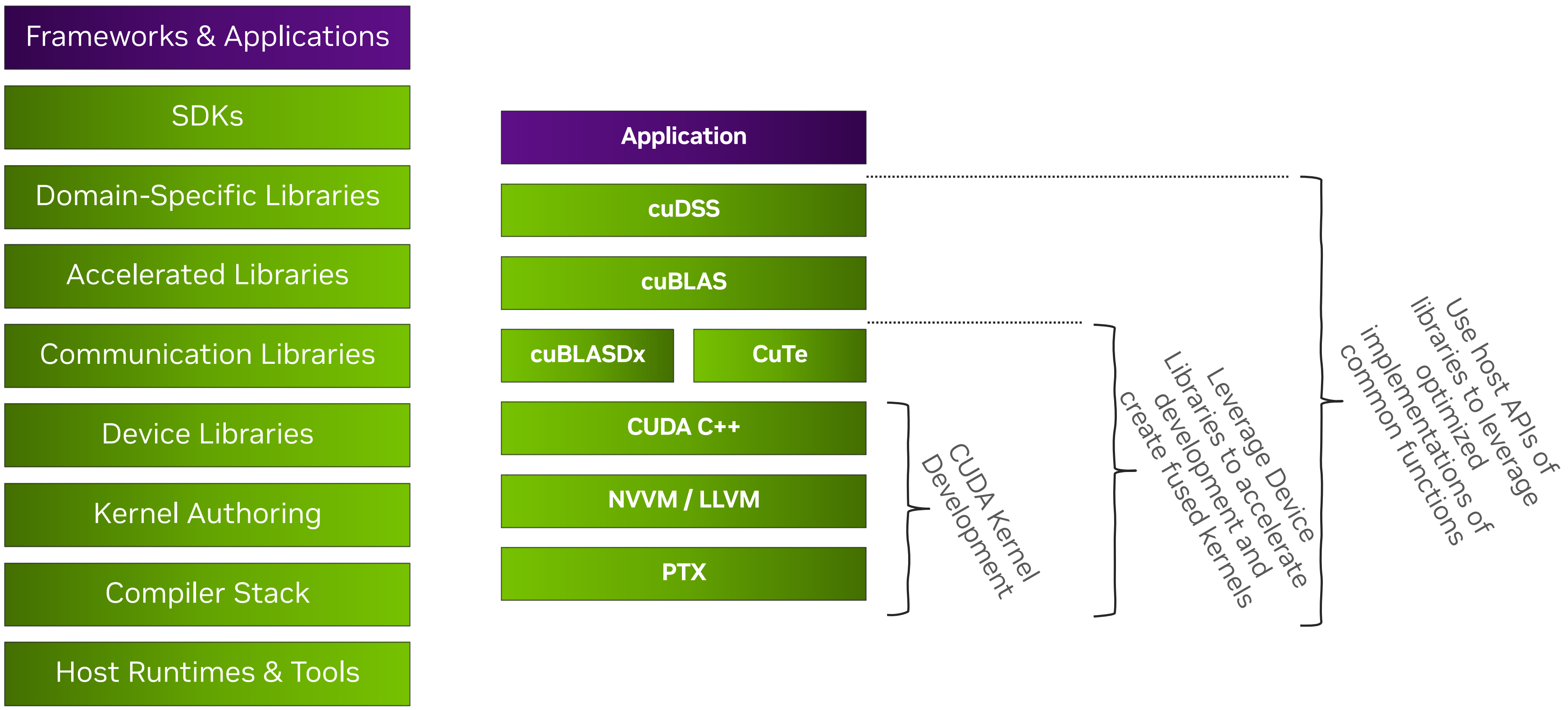
1. Direct Sparse Solvers will mostly benefit on non-100 class GPUs.
2. Initial emulation support in cuTENSOR will initially accelerate very large contractions on Blackwell and newer GPU architectures.

Device Extension Libraries & Emulation

Bringing advanced MathDx capabilities to ease writing high-performance emulation kernels in C++ and Python

Breaking Down the Compute Stack

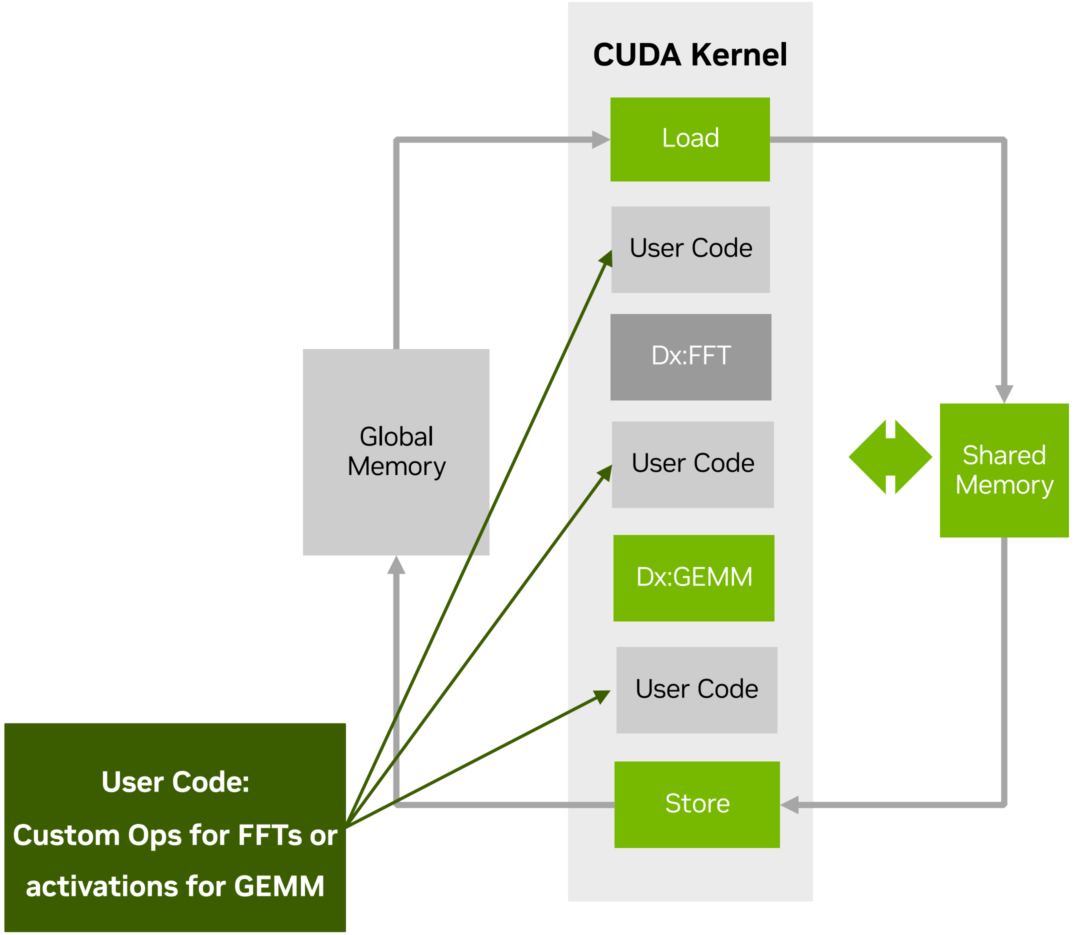
Example using an application that relies on a sparse direct solver



Device Extension Math Libraries

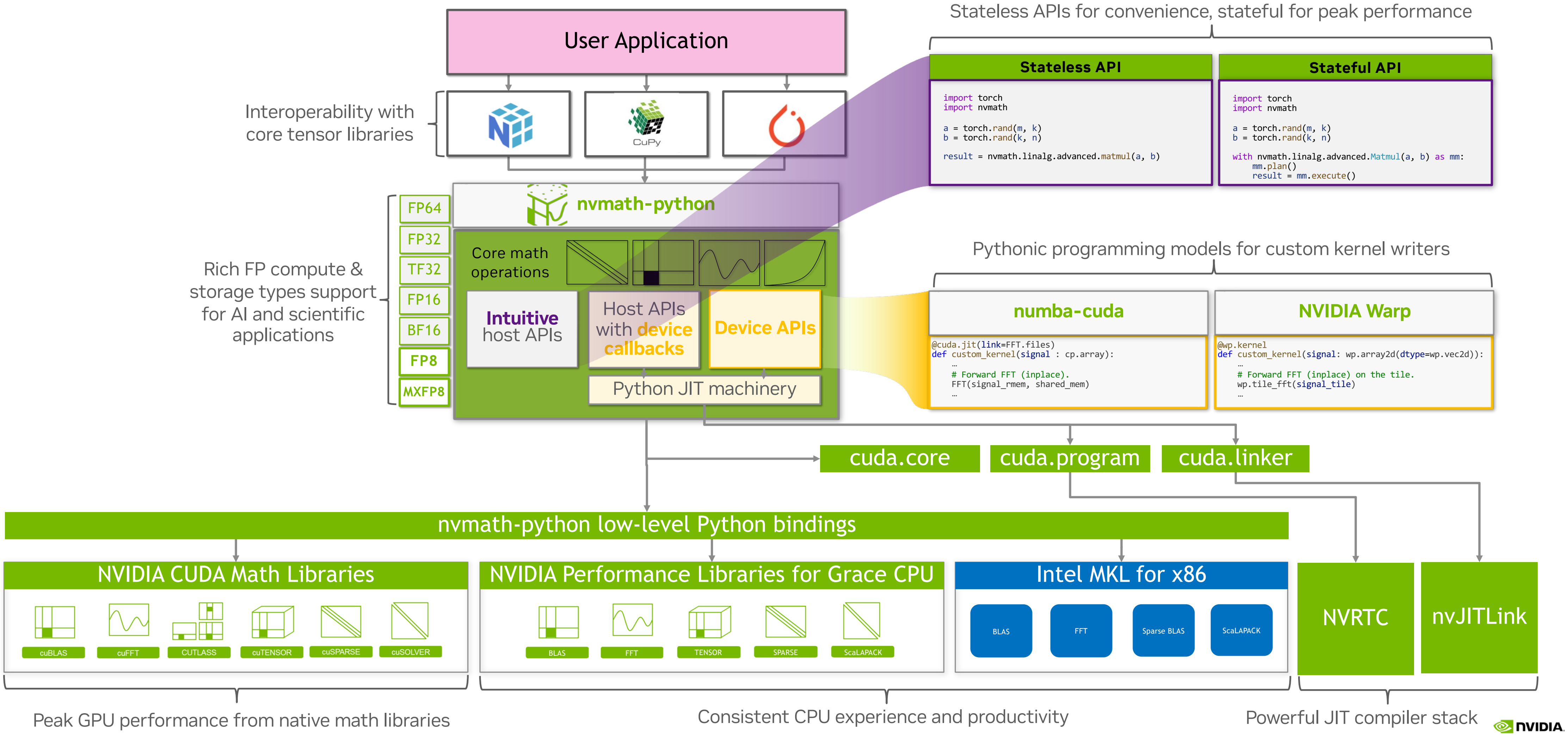
Maximum Flexibility of CUDA with Library Productivity

- CUDA kernels require handling CUDA memory and thread hierarchy
- Device Extension APIs provide configurable building blocks, with no call overhead for use in user CUDA kernel
- C++ and Python support
- Currently available:
  - **cuFFTDx**
  - **cuBLASDx**
  - **cuSOLVERDx**
  - **cuRANDDx**
  - **nvCOMPDx**



nvmath-python

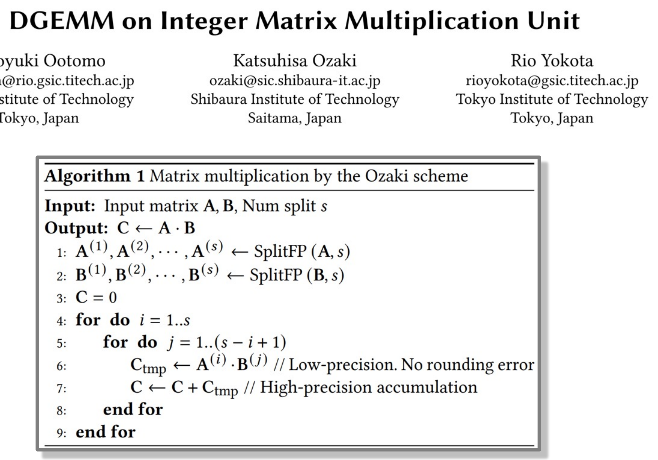
Reimagining math libraries for the Python ecosystem



NVIDIA

Floating-Point Emulation Using nvmath-python and Numba

GEMM emulation using INT8 tensor-core IMMA accessible through nvmath-python



DGEMM on Integer Matrix Multiplication Unit

Hiroyuki Ootomo
ootomo.h@rio.gsic.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Katsuhisa Ozaki
ozaki@sic.shibaura-it.ac.jp
Shibaura Institute of Technology
Saitama, Japan

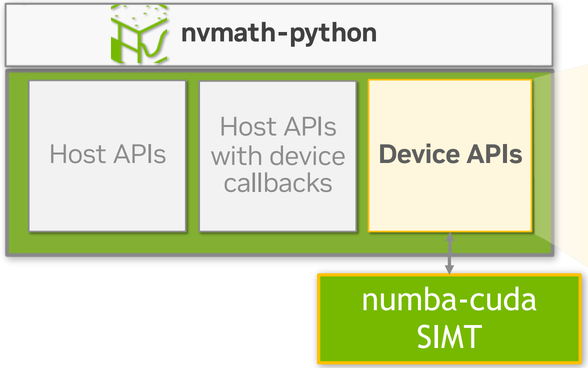
Rio Yokota
rioyokota@gsic.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Algorithm 1 Matrix multiplication by the Ozaki scheme

Input: Input matrix **A**, **B**, Num split s
Output: $\mathbf{C} \leftarrow \mathbf{A} \cdot \mathbf{B}$

```
A^(1), A^(2), ..., A^(s) ← SplitFP (A, s)
B^(1), B^(2), ..., B^(s) ← SplitFP (B, s)
C = 0
for do i = 1..s
    for do j = 1..(s − i + 1)
        C_tmp ← A^(i) · B^(j) // Low-precision. No rounding error
        C ← C + C_tmp // High-precision accumulation
    end for
end for
```

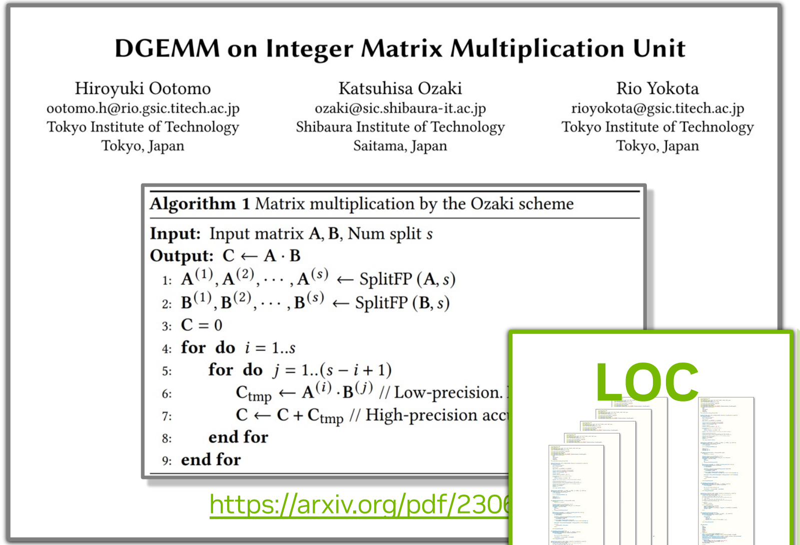
https://arxiv.org/pdf/2306.11975



Thanks to **NVIDIA MathDx**, developers can easily write high-performance numerical kernels in C++. Now **nvmath-python brings the MathDx goodness to Python** so that users can write high-performance kernels using **numba-cuda**

Floating-Point Emulation Using nvmath-python and Numba

GEMM emulation using INT8 tensor-core IMMA accessible through nvmath-python



DGEMM on Integer Matrix Multiplication Unit

Hiroyuki Ootomo
ootomo.h@rio.gsic.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

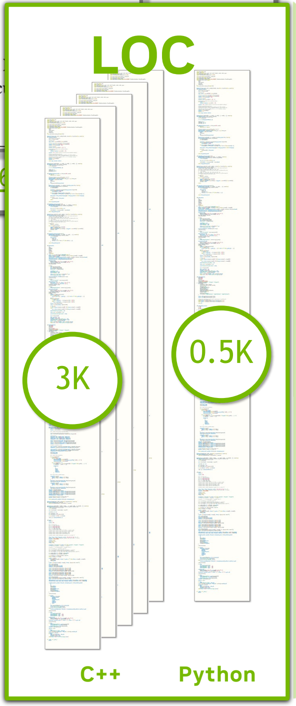
Katsuhisa Ozaki
ozaki@sic.shibaura-it.ac.jp
Shibaura Institute of Technology
Saitama, Japan

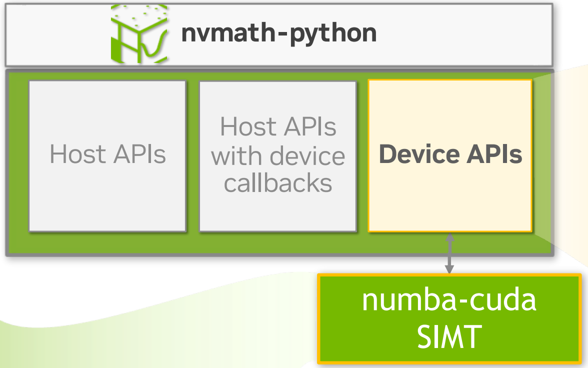
Rio Yokota
rioyokota@gsic.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Algorithm 1 Matrix multiplication by the Ozaki scheme

Input: Input matrix **A**, **B**, Num split s
Output: $\mathbf{C} \leftarrow \mathbf{A} \cdot \mathbf{B}$
$\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \cdots, \mathbf{A}^{(s)} \leftarrow \mathrm{SplitFP}(\mathbf{A}, s)$
$\mathbf{B}^{(1)}, \mathbf{B}^{(2)}, \cdots, \mathbf{B}^{(s)} \leftarrow \mathrm{SplitFP}(\mathbf{B}, s)$
$\mathbf{C} = 0$
for do $i = 1..s$
  for do $j = 1..(s - i + 1)$
    $\mathbf{C}_{tmp} \leftarrow \mathbf{A}^{(i)} \cdot \mathbf{B}^{(j)}$ // Low-precision.
    $\mathbf{C} \leftarrow \mathbf{C} + \mathbf{C}_{tmp}$ // High-precision acc
  end for
end for

https://arxiv.org/pdf/2306

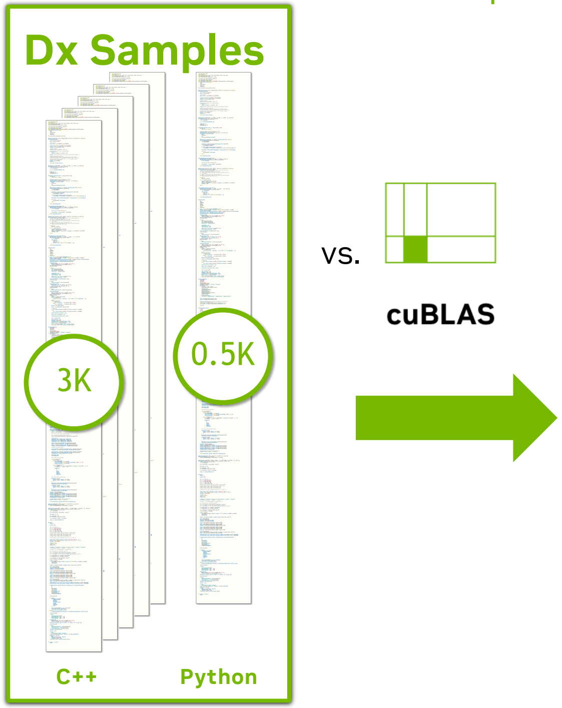


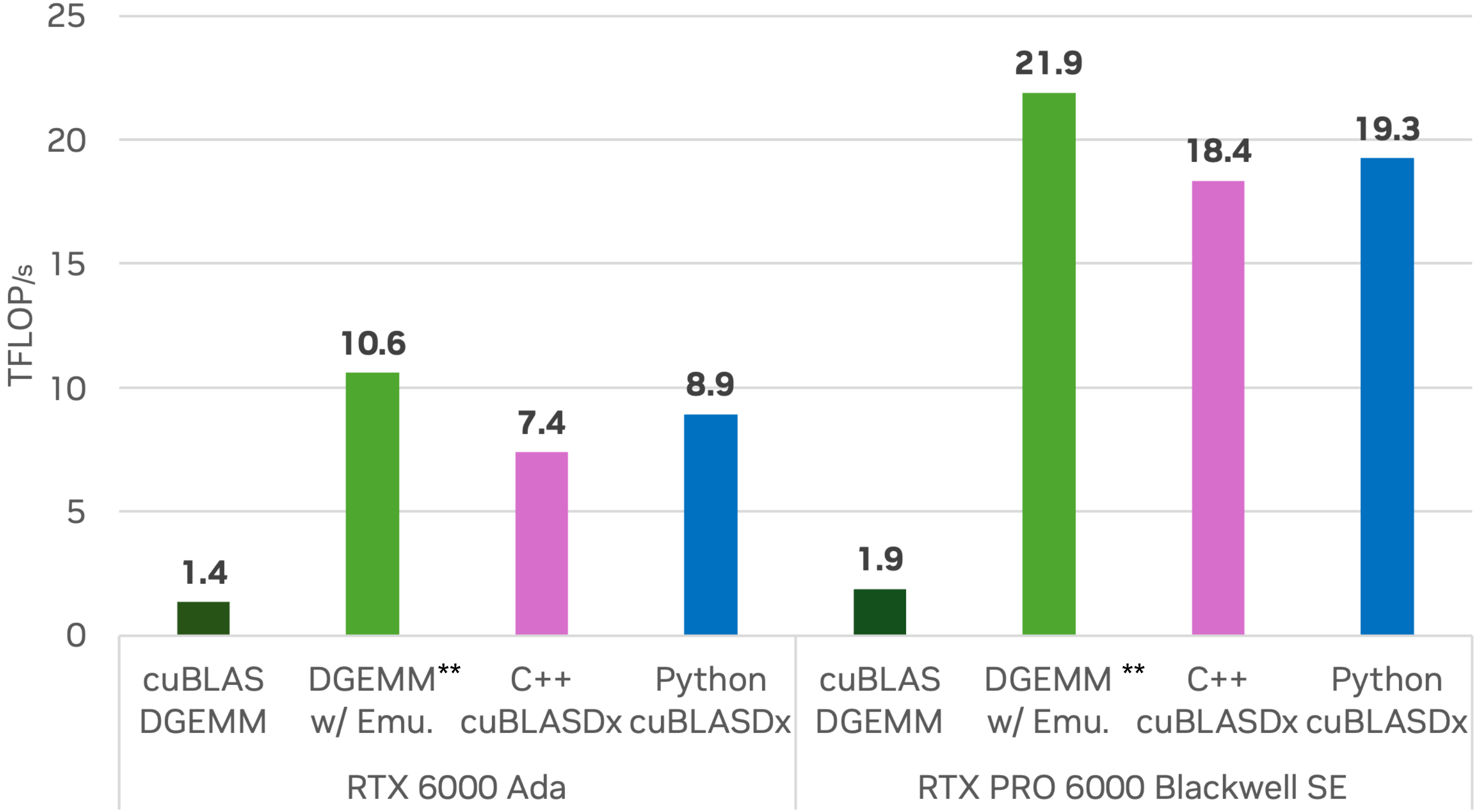


Thanks to **NVIDIA MathDx**, developers can easily write high-performance numerical kernels in C++ . Now **nvmath-python brings the MathDx goodness to Python** so that users can write high-performance kernels using **numba-cuda**

Floating-Point Emulation Using nvmath-python and Numba

GEMM emulation performance* using INT8 tensor-core IMMA accessible through nvmath-python





C++ Python

https://github.com/NVIDIA/CUDALibrarySamples/tree/master/MathDx/cuBLASDx/16_dgemm_emulation

https://github.com/NVIDIA/nvmath-python/blob/main/examples/device/cublasdx_fp64_emulation.py

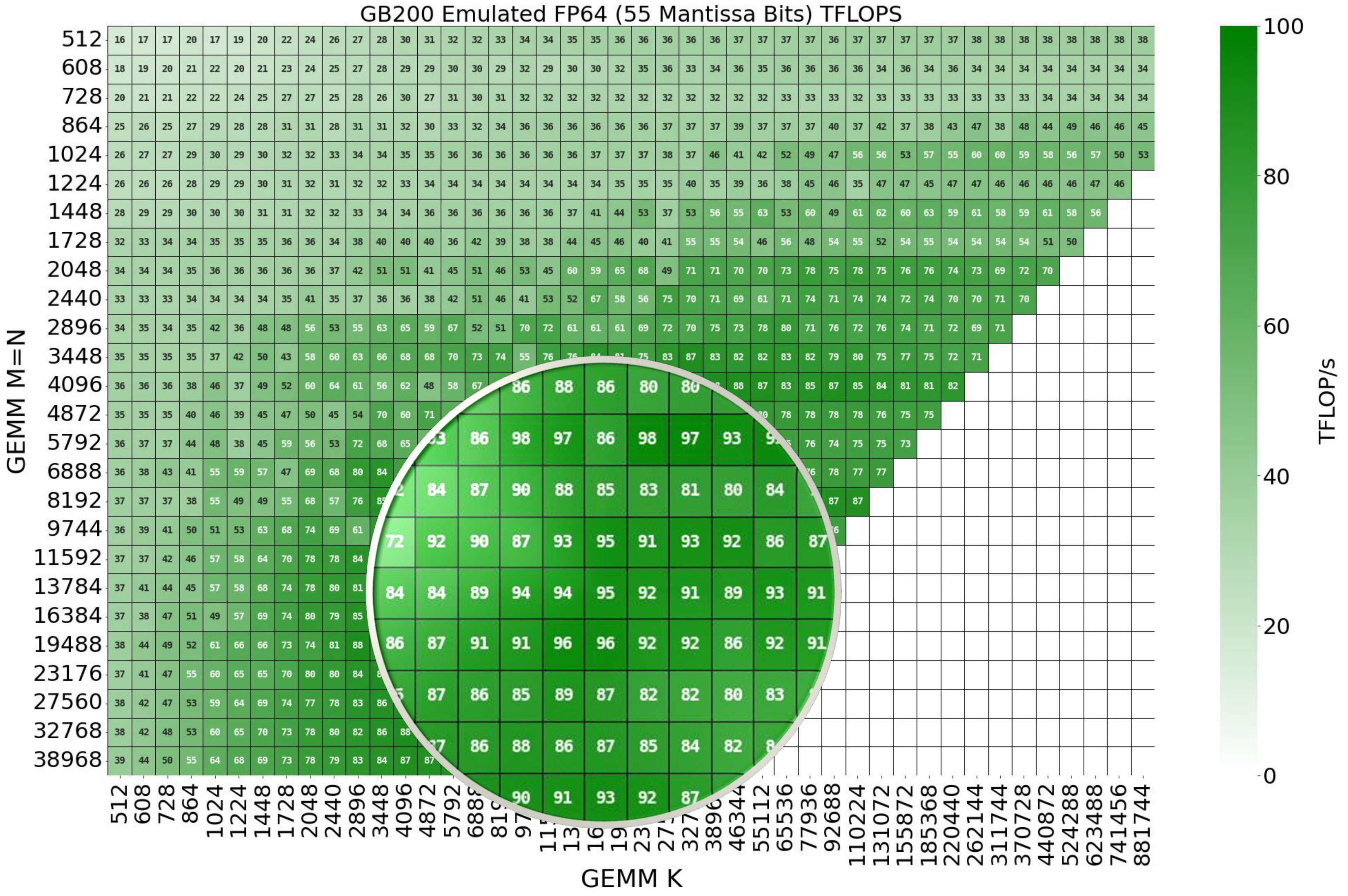
m=n=k=8192 problem size for $C_{mn}=A_{mk}B_{kn}$

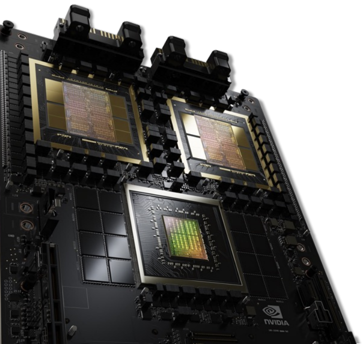
(*) Preliminary data, subject to change.

(**) cuBLAS DGEMM w/ Emulation not yet released

FP64 Matrix Multiplication Emulation in cuBLAS

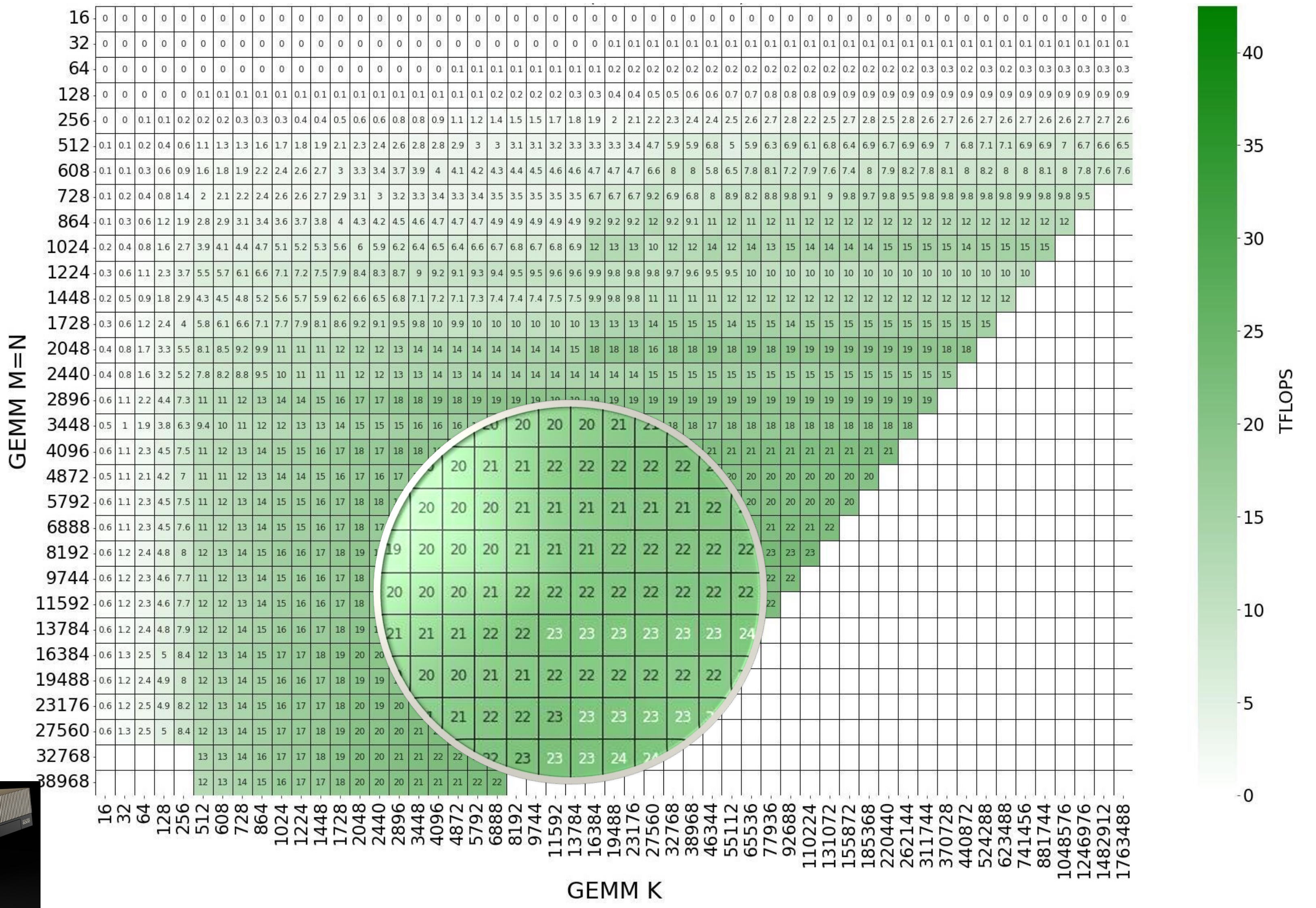
GB200 Emulated FP64 GEMM Performance [TFLOPS]

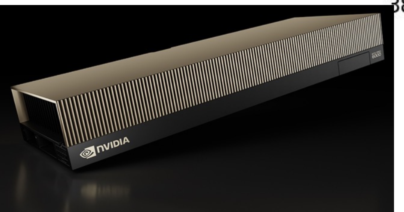




Blackwell RTX Pro 6000 SE Emulated FP64 GEMM Performance [TFLOPS]

55 Mantissa Bits Used For Emulation (s=7)



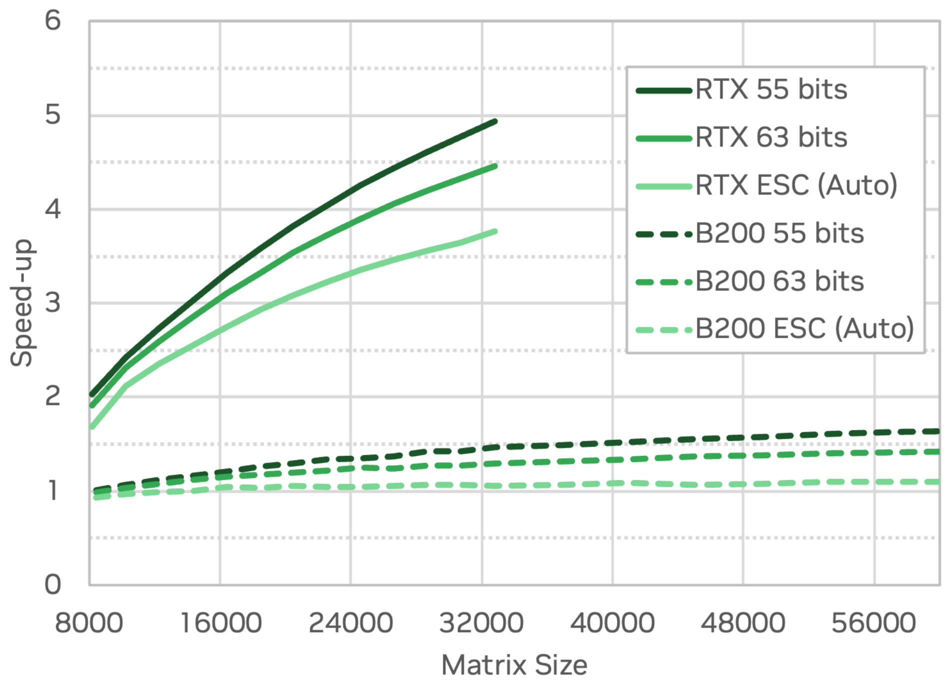


NVIDIA

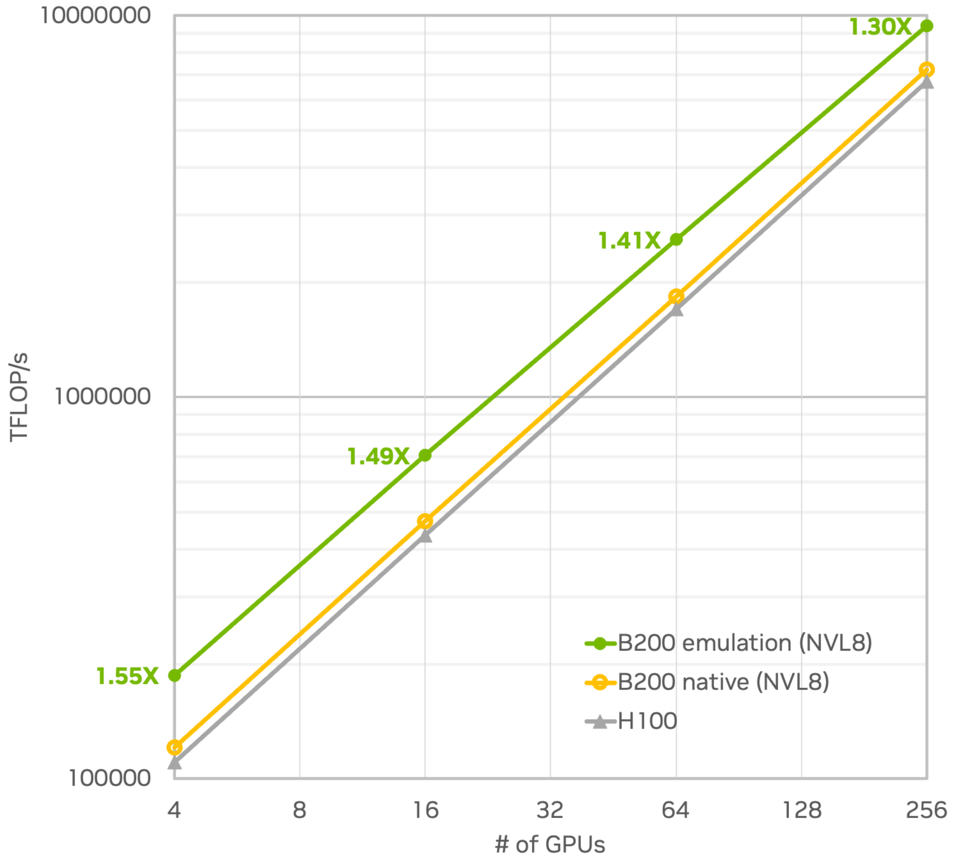
Performance Impact of Emulation in cuSOLVER

QR Factorization Study

cuSOLVER Single GPU Speed-ups for both RTX PRO 6000 Blackwell Server Edition and B200



cuSOLVERMp Weak-Scaling Study on DGX B200 vs H200 Cluster

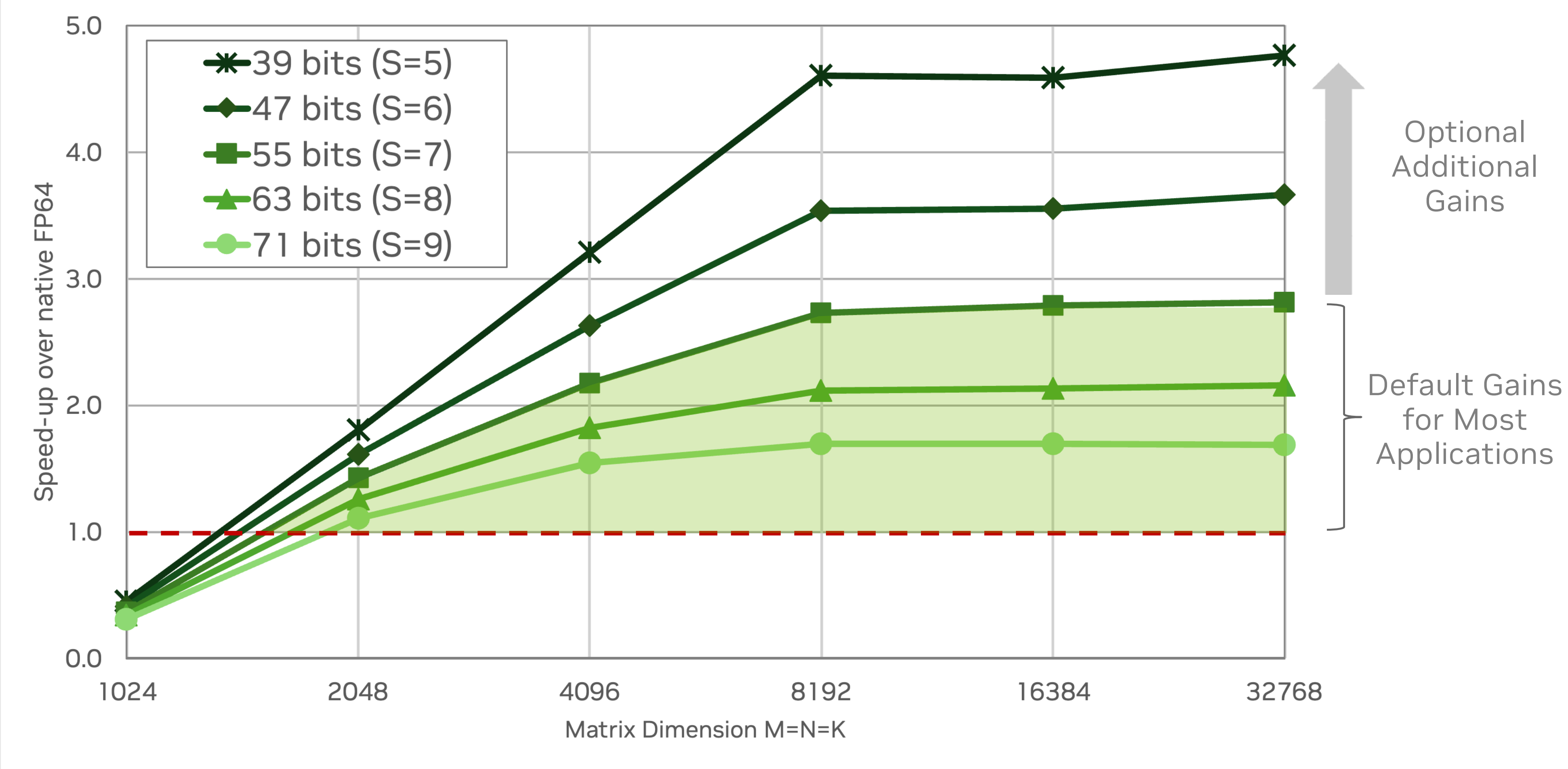


55 bits used for emulation

All data points shown pass cuSOLVER accuracy tests

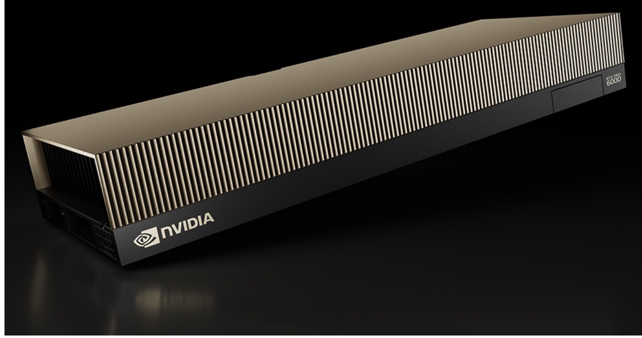
Potential Additional Efficiency Gains

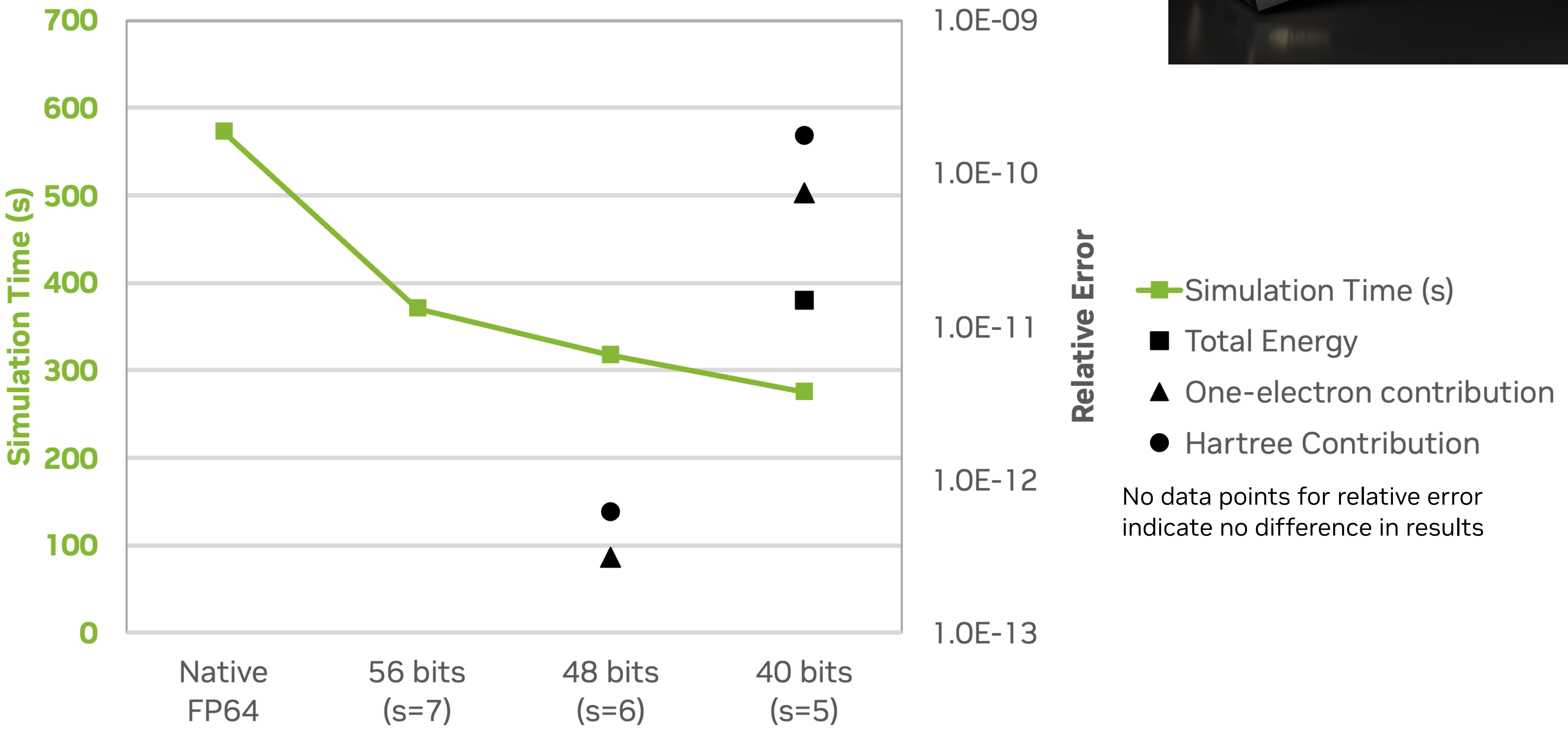
Performance of Emulated GEMM on B200 GPUs for various number of bits used



Quantum Espresso Performance with FP Emulation (Ozaki-I)

AuSurf Benchmark on RTX 6000 Pro Blackwell SE (Low native FP64 throughput)





No data points for relative error indicate no difference in results

Automatically Determining Emulation Parameters: Exponent Span Capacity (ESC) Algorithm

ESC (Exponent Span Capacity)

Number of (Additional) Bits Required in the Intermediate Representation to Maintain Desired Precision

- **What is the ESC?**
  - Recall that the fixed-point representation has no dynamic exponent
  - To represent values with different exponents, fixed-precision emulation, we:
    - Shift mantissa bits left (greater exponent) or shift mantissa bits right (lesser exponent)
  - ESC is the number of **extra** bits needed in the intermediate (fixed-point) representation to hold mantissa values
- Matrix multiplication is equivalent to a set of independent dot products
- Every dot product has a bit-shift requirement (ESC)
- The bit-shift requirement for matrix multiplication is the maximum of the constituent dot product bit-shift requirements

ESC (Exponent Span Capacity)

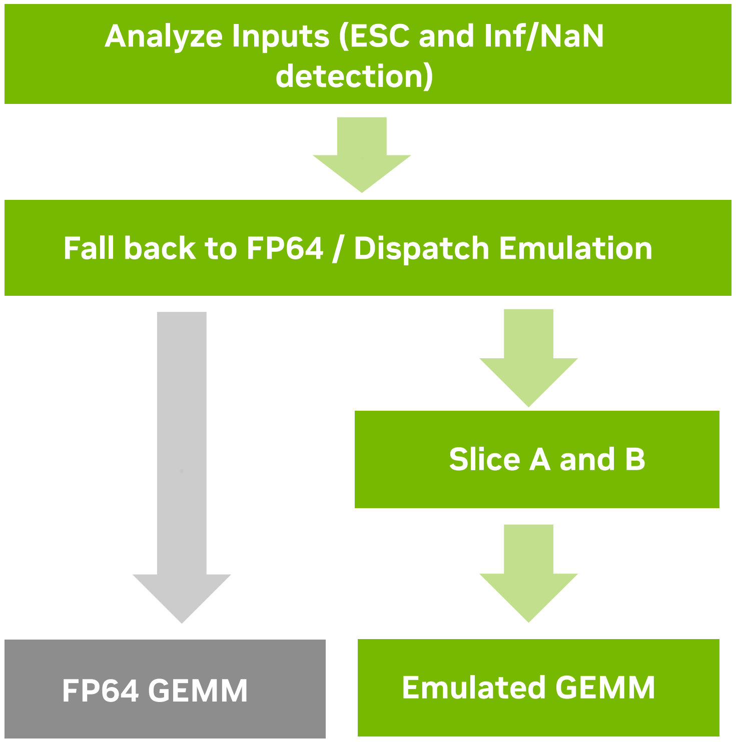
Number of (Additional) Bits Required in the Intermediate Representation to Maintain Desired Precision

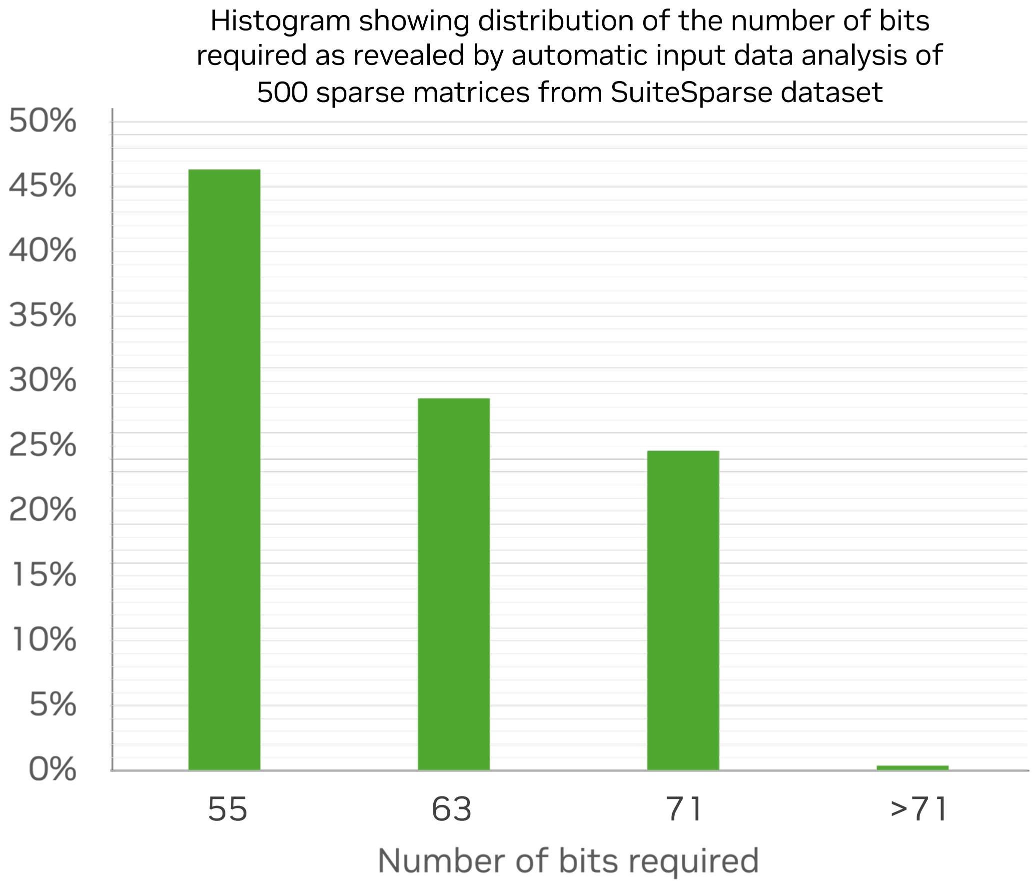
- Consider a Single Dot Product
  - $s = \vec{x} \times \vec{y}$
- View this Dot Product in terms of a Two Step Process, Consistent with the Underlying Mathematics
  - $\vec{z} = \vec{x} \odot \vec{y}$: Hadamard Product ($\vec{z}_i = \vec{x}_i\vec{y}_i \; \forall \; i, 1{:}n$)
  - $s = \sum_{i=1}^{n} \vec{z}_i$: Reduce
- There are 3 Special (Not Necessarily Unique) Values to Consider
  - (1) The maximum exponent in $\vec{x}$: ($Max(Exp(\vec{x}))$)
  - (2) The maximum exponent in $\vec{y}$: ($Max(Exp(\vec{y}))$)
  - (3) The maximum exponent in $\vec{z}$: ($Max(Exp(\vec{z}))$)
- The ESC is Defined as:
  - $(Max\big(Exp(\vec{x})\big) + Max(Exp(\vec{y})))$ - $Max(Exp(\vec{z}))$

- What Does This Definition Provide?
  - The largest contribution(s) to the dot product are captured in full-fidelity
    - Requested number of mantissa bits in $\vec{x}_i \; and \; \vec{y}_i$ that contribute to $Max(Exp(\vec{z}))$ are extracted from their FP representation and used
    - Regardless of “skew” (assumes maximal skew)
      - i.e. the contributions to $Max(Exp(\vec{z}))$ are implicitly assumed to be as asymmetric as they possibly could be
      - e.g., Assume that $Max(Exp(\vec{x}))$ $Max(Exp(\vec{y}))$ are both 100, ESC is 80, and $Max(Exp(\vec{z}))$ is 120
        - This approach assumes that the elements, $\vec{x}_i \; and \; \vec{y}_i$, that contribute to $Max(Exp(\vec{z}))$ could have an exponent of 20 (safe)
        - As opposed to assuming that both contributing exponents are 60 (unsafe) (i.e. 100/20 vs. 60/60)

FP64-like Accuracy and Potential Performance Boost

Implementing guardrails for seamless acceleration

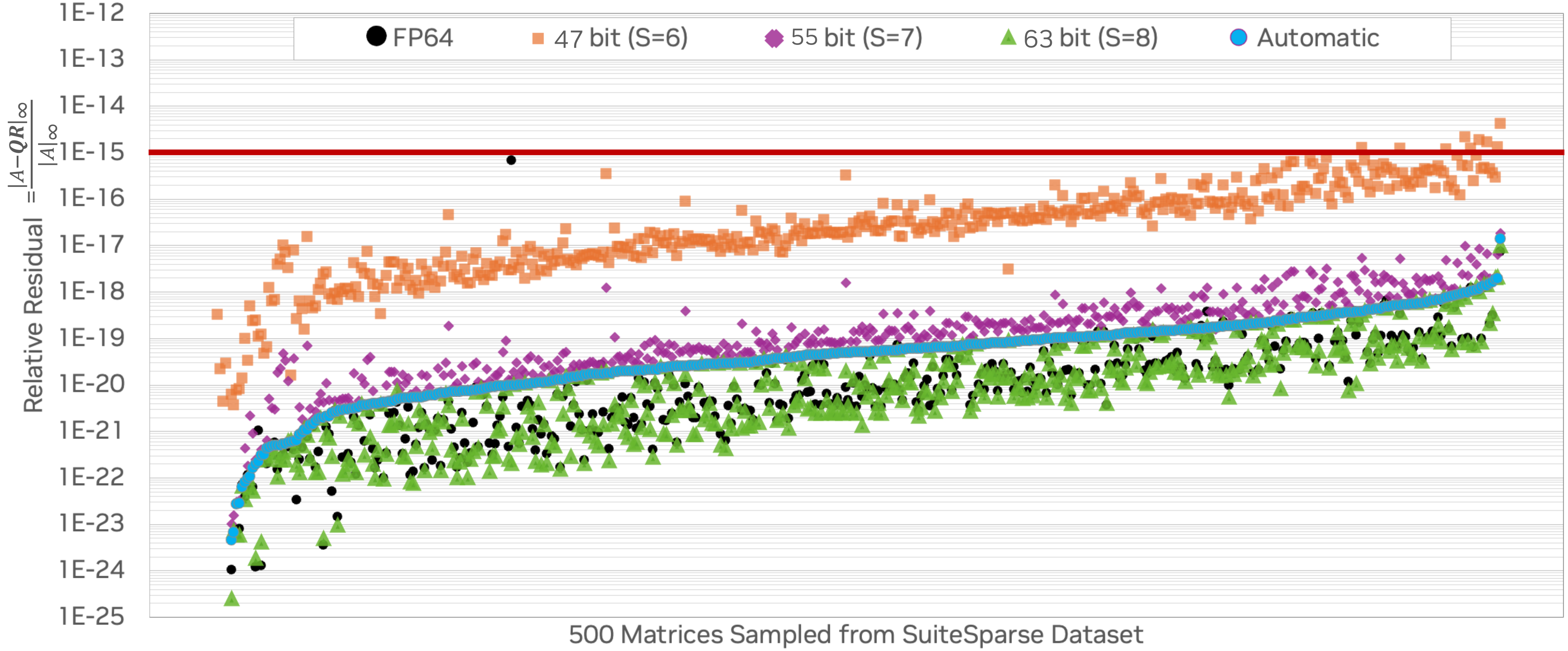




ESC (Exponent Span Capacity) calculates the number of extra mantissa bits required to accurately calculate the matrix multiplication result

Automatic Tunning of Emulation for Accuracy

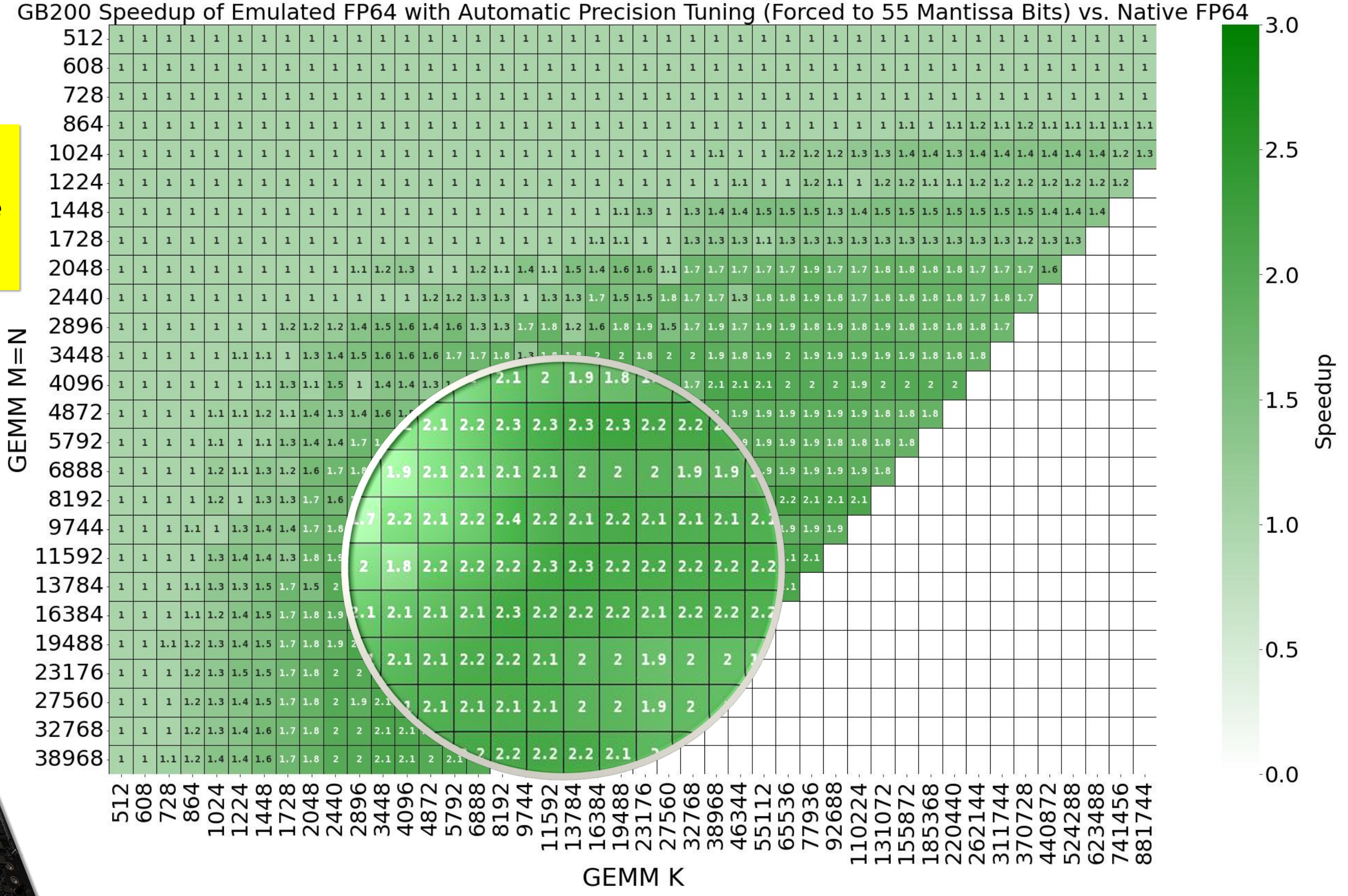
Library Selects the Number of Bits Automatically Based on Input Data with 10% overhead

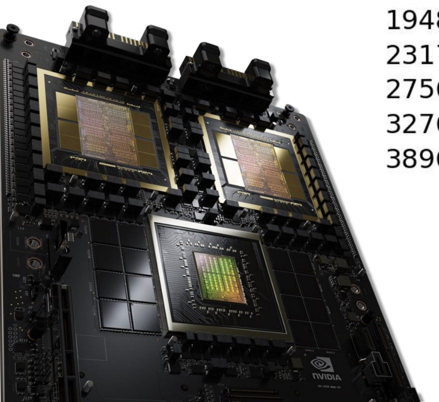


GB200 Emulated FP64 GEMM Speed-up vs Native FP64

With ESC (Automatic Precision Tuning) forced to 55 mantissa bits

For cases when emulation is faster, the geomean of the ESC overhead is 10.5%





Grading the cuBLAS DGEMM Implementation

Demonstration of Safety Using FP64 Emulation

How can we ensure the results are always correct? What are the limits of cases emulation can handle?

- Numerical linear algebra experts have developed tests that can detect “emulation” or push them beyond practical limits
  - “How to grade the accuracy of an implementation of the BLAS,” by Jim Demmel, Xiaoye Li, Julien Langou, Weslley Pereira, Mark Gates, and Cindy Rubio Gonzalez
  - https://www.cs.utexas.edu/~flame/BLISRetreat2024/slides/Grading_BLAS.pdf
- There are three tests (screens)
  - Strassen’s
  - Fixed-Point Strassen’s
  - **Fixed-Point Emulation**
- We implemented the not gameable versions (supersedes gameable)
  - We will make the testing source code available (in Q3’25)
- We are testing our libraries against these tests:
  - When emulation is turned on with no additional options:
    - The library automatically calculates the number of extra mantissa bits required and falls back to native FP64 to **deliver the correct result**
  - When emulation is turned on and forced to use a fixed number of mantissa bits by the user
    - The library will honor the user’s request and it will fail the **fixed-point emulation test**

Grading Test for Emulation Detection

Push Emulation to Its Limits to Test ESC

- Generate vector, x, using uniform random distribution with all elements in [1, 2)
- A Diagonal Matrix D is used to scale:
  - $y \leftarrow x \cdot D$ and $z \leftarrow x \cdot D^{-1}$
- Columns of A and rows of B matrices are circularly shifted "copies" of y and z, respectively
- We progressively increase the max. value of D from 2^0 to the 2^{512} limit to make the problem harder

Diagonal Exponent (base 2)

$$\begin{pmatrix} 507 \\ 363 \\ 218 \\ 73 \\ -72 \\ -217 \\ -362 \\ -507 \end{pmatrix}$$

Diagonal

$$\begin{pmatrix} 4.19\times10^{152} \\ 1.879\times10^{109} \\ 4.212\times10^{65} \\ 9.445\times10^{21} \\ 2.118\times10^{-22} \\ 4.748\times10^{-66} \\ 1.064\times10^{-109} \\ 2.387\times10^{-153} \end{pmatrix}$$

$x \cdot D$

A Matrix

$$\begin{pmatrix}
6.319\times10^{152} & 3.186\times10^{109} & 7.828\times10^{65} & 1.252\times10^{22} & 2.584\times10^{-22} & 8.124\times10^{-66} & 1.926\times10^{-109} & 3.219\times10^{-153} \\
3.186\times10^{109} & 7.828\times10^{65} & 1.252\times10^{22} & 2.584\times10^{-22} & 8.124\times10^{-66} & 1.926\times10^{-109} & 3.219\times10^{-153} & 6.319\times10^{152} \\
7.828\times10^{65} & 1.252\times10^{22} & 2.584\times10^{-22} & 8.124\times10^{-66} & 1.926\times10^{-109} & 3.219\times10^{-153} & 6.319\times10^{152} & 3.186\times10^{109} \\
1.252\times10^{22} & 2.584\times10^{-22} & 8.124\times10^{-66} & 1.926\times10^{-109} & 3.219\times10^{-153} & 6.319\times10^{152} & 3.186\times10^{109} & 7.828\times10^{65} \\
2.584\times10^{-22} & 8.124\times10^{-66} & 1.926\times10^{-109} & 3.219\times10^{-153} & 6.319\times10^{152} & 3.186\times10^{109} & 7.828\times10^{65} & 1.252\times10^{22} \\
8.124\times10^{-66} & 1.926\times10^{-109} & 3.219\times10^{-153} & 6.319\times10^{152} & 3.186\times10^{109} & 7.828\times10^{65} & 1.252\times10^{22} & 2.584\times10^{-22} \\
1.926\times10^{-109} & 3.219\times10^{-153} & 6.319\times10^{152} & 3.186\times10^{109} & 7.828\times10^{65} & 1.252\times10^{22} & 2.584\times10^{-22} & 8.124\times10^{-66} \\
3.219\times10^{-153} & 6.319\times10^{152} & 3.186\times10^{109} & 7.828\times10^{65} & 1.252\times10^{22} & 2.584\times10^{-22} & 8.124\times10^{-66} & 1.926\times10^{-109}
\end{pmatrix}$$

$x \cdot D^{-1}$

B Matrix

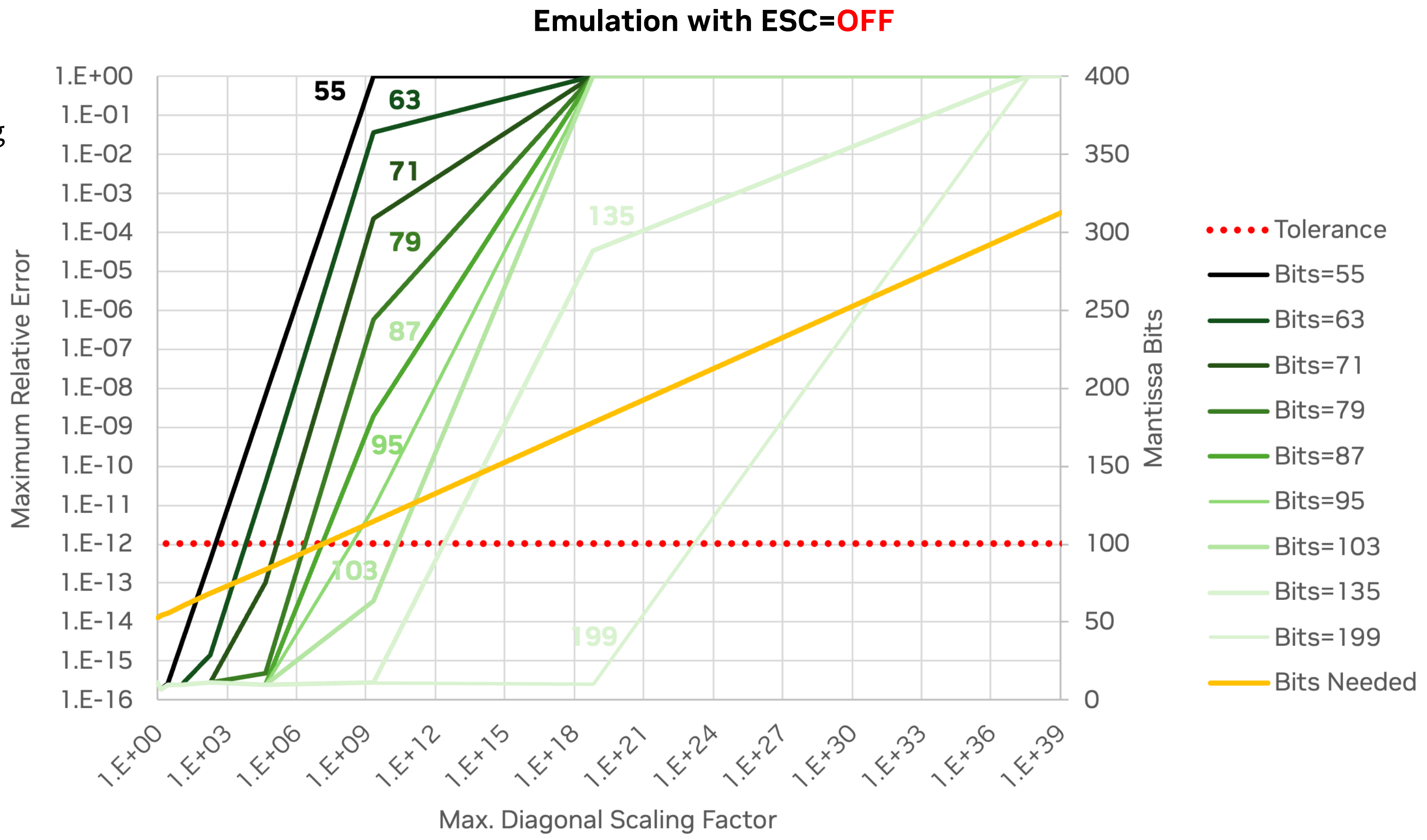
$$\begin{pmatrix}
3.6\times10^{-153} & 9.026\times10^{-110} & 4.412\times10^{-66} & 1.404\times10^{-22} & 5.762\times10^{21} & 3.604\times10^{65} & 1.7\times10^{109} & 5.651\times10^{152} \\
9.026\times10^{-110} & 4.412\times10^{-66} & 1.404\times10^{-22} & 5.762\times10^{21} & 3.604\times10^{65} & 1.7\times10^{109} & 5.651\times10^{152} & 3.6\times10^{-153} \\
4.412\times10^{-66} & 1.404\times10^{-22} & 5.762\times10^{21} & 3.604\times10^{65} & 1.7\times10^{109} & 5.651\times10^{152} & 3.6\times10^{-153} & 9.026\times10^{-110} \\
1.404\times10^{-22} & 5.762\times10^{21} & 3.604\times10^{65} & 1.7\times10^{109} & 5.651\times10^{152} & 3.6\times10^{-153} & 9.026\times10^{-110} & 4.412\times10^{-66} \\
5.762\times10^{21} & 3.604\times10^{65} & 1.7\times10^{109} & 5.651\times10^{152} & 3.6\times10^{-153} & 9.026\times10^{-110} & 4.412\times10^{-66} & 1.404\times10^{-22} \\
3.604\times10^{65} & 1.7\times10^{109} & 5.651\times10^{152} & 3.6\times10^{-153} & 9.026\times10^{-110} & 4.412\times10^{-66} & 1.404\times10^{-22} & 5.762\times10^{21} \\
1.7\times10^{109} & 5.651\times10^{152} & 3.6\times10^{-153} & 9.026\times10^{-110} & 4.412\times10^{-66} & 1.404\times10^{-22} & 5.762\times10^{21} & 3.604\times10^{65} \\
5.651\times10^{152} & 3.6\times10^{-153} & 9.026\times10^{-110} & 4.412\times10^{-66} & 1.404\times10^{-22} & 5.762\times10^{21} & 3.604\times10^{65} & 1.7\times10^{109}
\end{pmatrix}$$

Grading Test for Emulation Detection

Push Emulation to Its Limits to Test ESC

- We progressively increase the max. value of D from 2^0 to the 2^{512} limit to make the problem harder
- Reference matrix was calculated using `long double`
- Calculate the error as:

$$Max.Relative\ Error = \max_{i,j}\left|\frac{C - C_{ref}}{C_{ref}}\right|$$



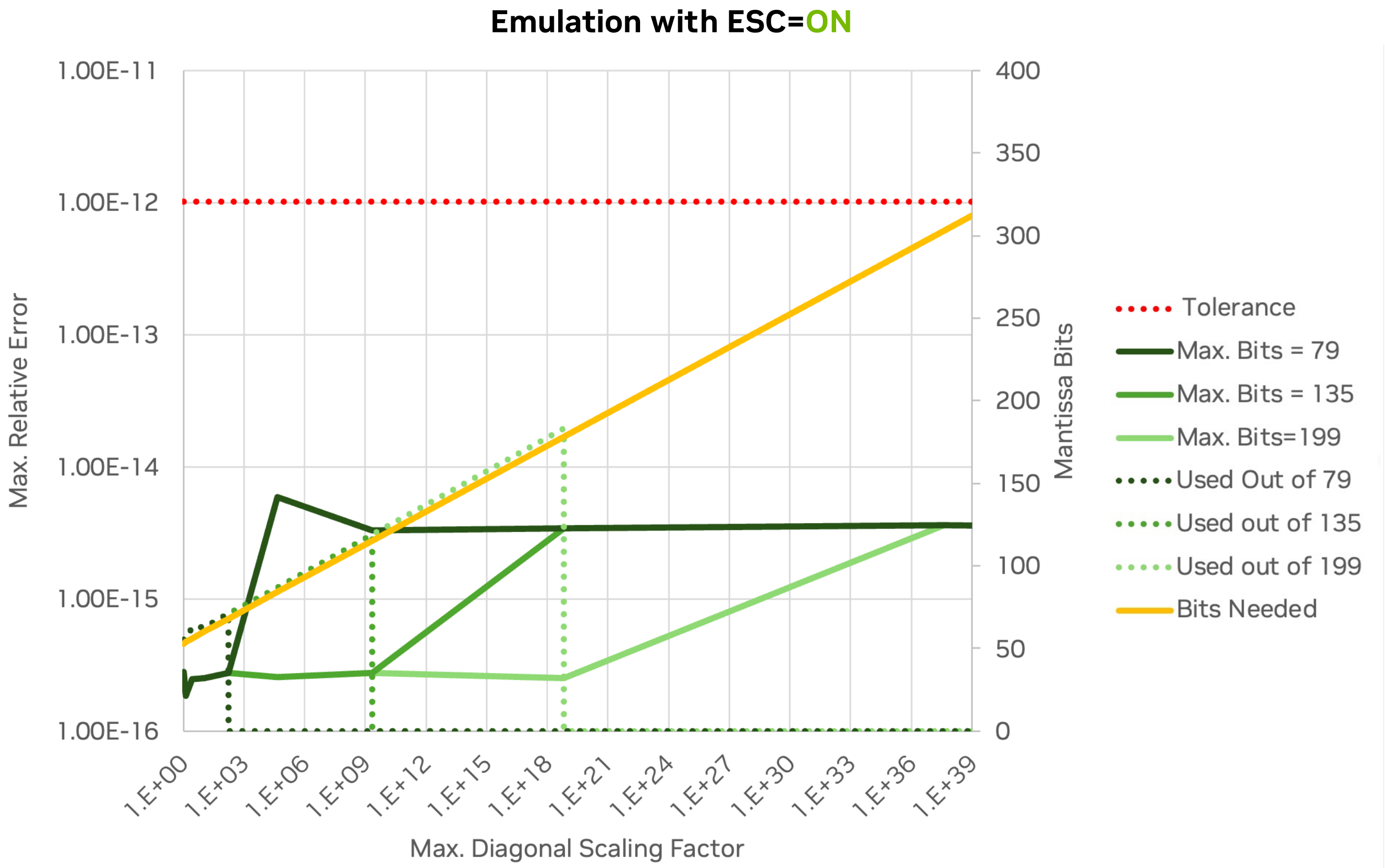
Grading Test for Emulation Detection

Push Emulation to Its Limits to Test ESC

- We progressively increase the max. value of D from 2^0 to the 2^{512} limit to make the problem harder
- Reference matrix was calculated using `long double`
- Calculate the error as:

$$Max.\,Relative\ Error = \max_{i,j}\left|\frac{C - C_{ref}}{C_{ref}}\right|$$

- With ESC the library **maintains accuracy throughout** the increasing difficulty of the problem, switching automatically from the emulation scheme to native FP64

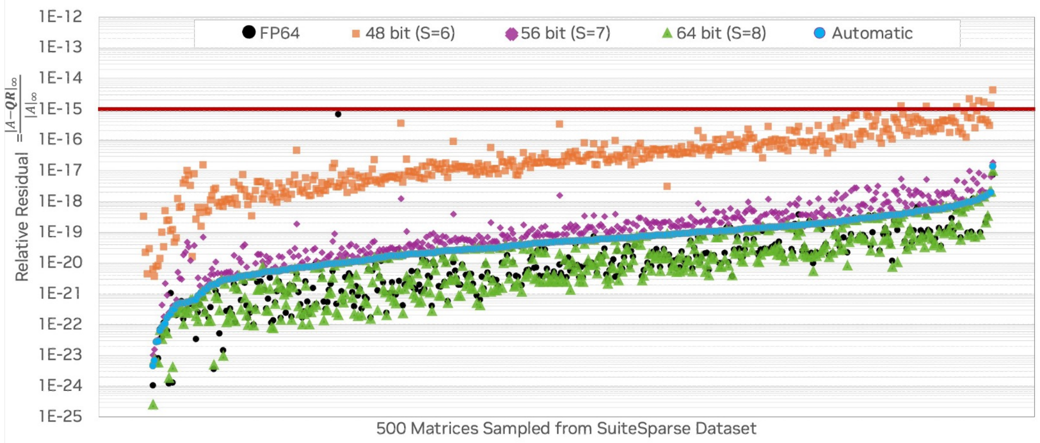


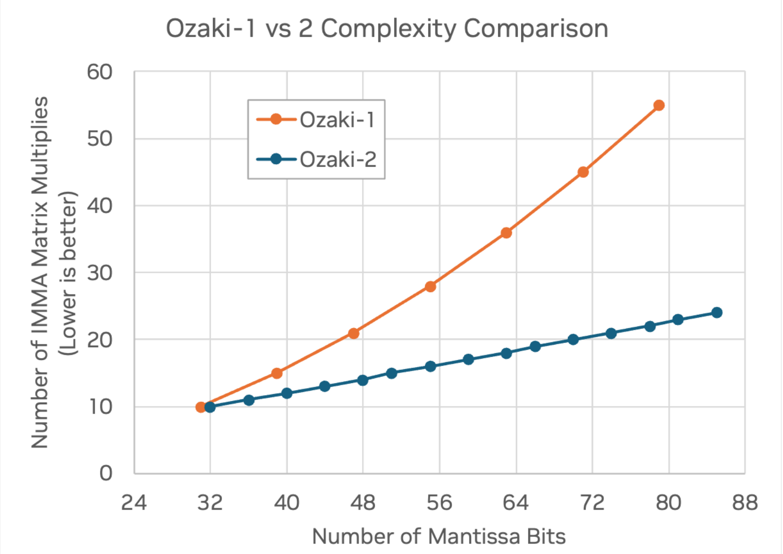
Closing Remarks & Future Work

Closing Remarks & Future Work

Productization of Emulation for Matrix Multiplications

- Single precision (FP32) matrix multiplication emulation using BF16x9 method is **available now**
- Double-precision (FP64) with Ozaki-I method will be released second half of 2025
- Environment variables and programmatic APIs are available to control emulation behavior
- **Safeguards (e.g., ESC)** are in place to fallback to native HW-based kernels to guarantee results are always correct and corner cases are handled
- cuBLASDx based implementation samples (without ESC) **are available on github** in both C++ and Python
- Future releases will
  - Add opt-in through other libraries that rely on matrix multiplications
  - **Ozaki-II** method will be added to is fundamentally different and is able to handle more difficult cases with less additional compute cost
  - Continue to improve performance and reduce memory requirements
  - After enough exposure will **switch to opt-out**





NVIDIA

NVIDIA