Cascading GEMM: High Precision from Low Precision

Devangi N. Parikh and Greg Henry

UT Austin
Intel

BLIS Retreat 2022

Outline

Introduction

Motivation

- Increase in new low-precision formats and accelerators to keep up with the high computation demand.

Motivation

- Increase in new low-precision formats and accelerators to keep up with the high computation demand.
- Can we leverage low-precision compute to obtain high-precision accuracy?

Contributions

- Present a strategy for cascading `FP64x2` matrices into multiple `FP64` matrices that enables computing of `FP64x2` GEMM with ten `FP64` GEMMs.

Contributions

- Present a strategy for cascading `FP64x2` matrices into multiple `FP64` matrices that enables computing of `FP64x2` GEMM with ten `FP64` GEMMs.
- Describe a prototype implementation that makes the proposed strategy practical.

Contributions

- Present a strategy for cascading `FP64x2` matrices into multiple `FP64` matrices that enables computing of `FP64x2` GEMM with ten `FP64` GEMMs.
- Describe a prototype implementation that makes the proposed strategy practical.
- Present performance and accuracy results.

Contributions

- Present a strategy for cascading `FP64x2` matrices into multiple `FP64` matrices that enables computing of `FP64x2` GEMM with ten `FP64` GEMMs.
- Describe a prototype implementation that makes the proposed strategy practical.
- Present performance and accuracy results.
- Discuss opportunities of the proposed techniques.

Cascading

FP64x2 Scalars

Consider a 106-bit mantissa number χ.

$$\chi = \pm . \beta_0 \cdots \beta_{D-1} \beta_D \cdots \beta_{2D-1} \times 2^e,$$

If χ is normalized[1], then $D >= 53$.

[1] A FP64x2 number χ is normalized when the rounding to double is just the high part.

FP64x2 Scalars

Consider a 106-bit mantissa number χ.

$$\chi = \pm . \beta_0 \cdots \beta_{D-1} \beta_D \cdots \beta_{2D-1} \times 2^e,$$

If χ is normalized[1], then $D >= 53$.
We can represent χ exactly as two FP64 numbers (with the same exponent):

$$\chi = \chi_0 + \chi_1 \times 2^{-D},$$

where $\chi_0 = \pm . \beta_0 \beta_1 \cdots \beta_{D-1} \times 2^e$ and $\chi_1 = \pm . \beta_D \cdots \beta_{2D-1} \times 2^e$.

[1] A FP64x2 number χ is normalized when the rounding to double is just the high part.

FP64x2 Multiplication

Consider a second FP64x2, $\psi = \psi_0 + \psi_1 \times 2^{-D}$.

FP64x2 Multiplication

Consider a second FP64x2, $\psi = \psi_0 + \psi_1 \times 2^{-D}$.

The multiplication can be written as

$$\chi\psi = \chi_0\psi_0 + \chi_0\psi_1 \times 2^{-D} + \chi_1\psi_0 \times 2^{-D} + \chi_1\psi_1 \times 2^{-2D}.$$

FP64x2 Multiplication

Consider a second FP64x2, $\psi = \psi_0 + \psi_1 \times 2^{-D}$.

The multiplication can be written as

$$\chi\psi = \chi_0\psi_0 + \chi_0\psi_1 \times 2^{-D} + \chi_1\psi_0 \times 2^{-D} + \chi_1\psi_1 \times 2^{-2D}.$$

Each of these terms may incur floating point error, and thus to get an accurate result, we must capture these errors.

Cascading Scalars

Instead, we consider

$$\chi = \pm . \beta_0 \cdots \beta_{D_0-1} \beta_{D_0} \cdots \beta_{D_1-1} \beta_{D_1} \cdots \beta_{D_2-1} \beta_{D_2} \cdots \beta_{2D-1} \times 2^e,$$

Cascading Scalars

Instead, we consider

$$\chi = \pm . \beta_0 \cdots \beta_{D_0-1} \beta_{D_0} \cdots \beta_{D_1-1} \beta_{D_1} \cdots \beta_{D_2-1} \beta_{D_2} \cdots \beta_{2D-1} \times 2^e,$$

and "cascade" this `FP64x2` number into four chunks,

$$\chi = \pm . \boxed{\beta_0 \cdots \beta_{D_0-1}} \boxed{\beta_{D_0} \cdots \beta_{D_1-1}} \boxed{\beta_{D_1} \cdots \beta_{D_2-1}} \boxed{\beta_{D_2} \cdots \beta_{2D-1}} \times 2^e,$$

Cascading Scalars (contd.)

Rewrite χ in terms of four FP64s:

$$\begin{aligned}
\chi &= \underbrace{\pm.\beta_0 \cdots \beta_{D_0-1} \times 2^e} \\
&+ \underbrace{\pm.\beta_{D_0} \cdots \beta_{D_1-1} \times 2^e} \times \underbrace{2^{-D_0}} \\
&+ \underbrace{\pm.\beta_{D_1} \cdots \beta_{D_2-1} \times 2^e} \times \underbrace{2^{-D_1}} \\
&+ \underbrace{\pm.\beta_{D_2} \cdots \beta_{2D-1} \times 2^e} \times \underbrace{2^{-D_2}} \ .
\end{aligned}$$

Cascading Scalars (contd.)

Rewrite χ in terms of four FP64s:

$$\begin{aligned}
\chi &= \underbrace{\pm.\beta_0 \cdots \beta_{D_0-1} \times 2^e}_{\chi_0} \\
&+ \underbrace{\pm.\beta_{D_0} \cdots \beta_{D_1-1} \times 2^e}_{\chi_1} \times \underbrace{2^{-D_0}}_{\sigma_1} \\
&+ \underbrace{\pm.\beta_{D_1} \cdots \beta_{D_2-1} \times 2^e}_{\chi_2} \times \underbrace{2^{-D_1}}_{\sigma_2} \\
&+ \underbrace{\pm.\beta_{D_2} \cdots \beta_{2D-1} \times 2^e}_{\chi_3} \times \underbrace{2^{-D_2}}_{\sigma_3} .
\end{aligned}$$

Cascading Scalars (contd.)

More concisely,

$$\chi = \chi_0\sigma_0 + \chi_1\sigma_1 + \chi_2\sigma_2 + \chi_3\sigma_3,$$

where for simplicity $\sigma_0 = 1$ in our discussion.

Cascading Scalars (contd.)

More concisely,

$$\chi = \chi_0\sigma_0 + \chi_1\sigma_1 + \chi_2\sigma_2 + \chi_3\sigma_3,$$

where for simplicity $\sigma_0 = 1$ in our discussion.

Similarly, ψ:

$$\psi = \psi_0\tau_0 + \psi_1\tau_1 + \psi_2\tau_2 + \psi_3\tau_3,$$

Scalar Multiplication with Cascading Scalars

The product of these two scalars:

$$\begin{aligned}
\chi\psi = \quad & \underbrace{\chi_0\psi_0\sigma_0\tau_0}_{\text{bin 0}} + \underbrace{\chi_0\psi_1\sigma_0\tau_1 + \chi_1\psi_0\sigma_1\tau_0}_{\text{bin 1}} + \underbrace{\chi_0\psi_2\sigma_0\tau_2 + \chi_1\psi_1\sigma_1\tau_1 + \chi_2\psi_0\sigma_2\tau_0}_{\text{bin 2}} \\
& + \underbrace{\chi_0\psi_3\sigma_0\tau_3 + \chi_1\psi_2\sigma_1\tau_2 + \chi_2\psi_1\sigma_2\tau_1 + \chi_3\psi_0\sigma_3\tau_0}_{\text{bin 3}} + \underbrace{\chi_1\psi_3\sigma_1\tau_3 + \chi_2\psi_2\sigma_2\tau_2 + \chi_3\psi_1\sigma_3\tau_1}_{\text{bin 4}} \\
& + \underbrace{\chi_2\psi_3\sigma_2\tau_3 + \chi_3\psi_2\sigma_3\tau_2}_{\text{bin 5}} + \underbrace{\chi_3\psi_3\sigma_3\tau_3}_{\text{bin 6}}.
\end{aligned}$$

Scalar Multiplication with Cascading Scalars

The product of these two scalars:

$$\begin{aligned}\chi\psi = \quad & \underbrace{\chi_0\psi_0\sigma_0\tau_0}_{\text{bin 0}} + \underbrace{\chi_0\psi_1\sigma_0\tau_1 + \chi_1\psi_0\sigma_1\tau_0}_{\text{bin 1}} + \underbrace{\chi_0\psi_2\sigma_0\tau_2 + \chi_1\psi_1\sigma_1\tau_1 + \chi_2\psi_0\sigma_2\tau_0}_{\text{bin 2}} + \\ & \underbrace{\chi_0\psi_3\sigma_0\tau_3 + \chi_1\psi_2\sigma_1\tau_2 + \chi_2\psi_1\sigma_2\tau_1 + \chi_3\psi_0\sigma_3\tau_0}_{\text{bin 3}} + \underbrace{\chi_1\psi_3\sigma_1\tau_3 + \chi_2\psi_2\sigma_2\tau_2 + \chi_3\psi_1\sigma_3\tau_1}_{\text{bin 4}} + \\ & \underbrace{\chi_2\psi_3\sigma_2\tau_3 + \chi_3\psi_2\sigma_3\tau_2}_{\text{bin 5}} + \underbrace{\chi_3\psi_3\sigma_3\tau_3}_{\text{bin 6}}.\end{aligned}$$

Bin p includes the product terms where $i + j = p$.

Scalar Multiplication with Cascading Scalars

The product of these two scalars:

$$\begin{aligned}\chi\psi = \quad & \underbrace{\chi_0\psi_0\sigma_0\tau_0}_{\text{bin 0}} + \underbrace{\chi_0\psi_1\sigma_0\tau_1 + \chi_1\psi_0\sigma_1\tau_0}_{\text{bin 1}} + \underbrace{\chi_0\psi_2\sigma_0\tau_2 + \chi_1\psi_1\sigma_1\tau_1 + \chi_2\psi_0\sigma_2\tau_0}_{\text{bin 2}} + \\ & \underbrace{\chi_0\psi_3\sigma_0\tau_3 + \chi_1\psi_2\sigma_1\tau_2 + \chi_2\psi_1\sigma_2\tau_1 + \chi_3\psi_0\sigma_3\tau_0}_{\text{bin 3}} + \underbrace{\chi_1\psi_3\sigma_1\tau_3 + \chi_2\psi_2\sigma_2\tau_2 + \chi_3\psi_1\sigma_3\tau_1}_{\text{bin 4}} + \\ & \underbrace{\chi_2\psi_3\sigma_2\tau_3 + \chi_3\psi_2\sigma_3\tau_2}_{\text{bin 5}} + \underbrace{\chi_3\psi_3\sigma_3\tau_3}_{\text{bin 6}}.\end{aligned}$$

Bin p includes the product terms where $i + j = p$.

If each scalar has 4 chunks, then there are a total of 7 bins, and 16 products.

Scalar Multiplication with Cascading Scalars

The product of these two scalars:

$$\begin{aligned}\chi\psi = \quad & \underbrace{\chi_0\psi_0\sigma_0\tau_0}_{\text{bin 0}} + \underbrace{\chi_0\psi_1\sigma_0\tau_1 + \chi_1\psi_0\sigma_1\tau_0}_{\text{bin 1}} + \underbrace{\chi_0\psi_2\sigma_0\tau_2 + \chi_1\psi_1\sigma_1\tau_1 + \chi_2\psi_0\sigma_2\tau_0}_{\text{bin 2}} + \underbrace{\chi_0\psi_3\sigma_0\tau_3 + \chi_1\psi_2\sigma_1\tau_2 + \chi_2\psi_1\sigma_2\tau_1 + \chi_3\psi_0\sigma_3\tau_0}_{\text{bin 3}} \\ & + \underbrace{\chi_1\psi_3\sigma_1\tau_3 + \chi_2\psi_2\sigma_2\tau_2 + \chi_3\psi_1\sigma_3\tau_1}_{\text{bin 4}} + \underbrace{\chi_2\psi_3\sigma_2\tau_3 + \chi_3\psi_2\sigma_3\tau_2}_{\text{bin 5}} + \underbrace{\chi_3\psi_3\sigma_3\tau_3}_{\text{bin 6}}.\end{aligned}$$

Bin p includes the product terms where $i + j = p$.

If each scalar has 4 chunks, then there are a total of 7 bins, and 16 products.

If D_0, D_1, and D_2 are chosen carefully, the first three bins can be computed and stored exactly in FP64.

Scalar Multiplication with Cascading Scalars (contd.)

In our discussion here, $\sigma_i = \tau_i$ for $i \in \{0, 1, 2, 3\}$.

$$
\begin{array}{lrcl}
\text{bin 0:} & \sigma_0\tau_0 & = & 1 \\
\text{bin 1:} & \sigma_0\tau_1 = \sigma_1\tau_0 & = & 2^{-D_0} \\
\text{bin 2:} & \sigma_1\tau_1 \approx \sigma_0\tau_2 = \sigma_2\tau_0 & = & 2^{-D_1} \\
\text{bin 3:} & \sigma_1\tau_2 = \sigma_2\tau_1 \approx \sigma_0\tau_3 = \sigma_3\tau_0 & = & 2^{-D_2} \\
\text{bin 4:} & \sigma_2\tau_2 \approx \sigma_1\tau_3 = \sigma_3\tau_1 & \ll & 2^{-D_2} \\
\text{bin 5:} & \sigma_2\tau_3 = \sigma_3\tau_2 & \ll & 2^{-D_2} \\
\text{bin 6:} & \sigma_3\tau_3 & \ll & 2^{-D_2}.
\end{array}
$$

Cascading Vectors and Dot Products

Let x and y be vectors of size k with `FP64x2` numbers as their entries.

Cascading Vectors and Dot Products

Let x and y be vectors of size k with `FP64x2` numbers as their entries.

We wish to compute $\alpha = x^T y$.

Cascading Vectors and Dot Products

Let x and y be vectors of size k with `FP64x2` numbers as their entries.

We wish to compute $\alpha = x^T y$.

Cascading each of the vectors:

$$\begin{aligned} x &= x_0\sigma_0 + x_1\sigma_1 + x_2\sigma_2 + x_3\sigma_3 \\ y &= y_0\,\tau_0 + y_1\,\tau_1 + y_2\,\tau_2 + y_3\,\tau_3 \end{aligned}$$

Cascading Vectors and Dot Products

$$
\begin{aligned}
x^T y = \quad & x_0^T y_0 \sigma_0 \tau_0 + x_0^T y_1 \sigma_0 \tau_1 + x_0^T y_2 \sigma_0 \tau_2 + x_0^T y_3 \sigma_0 \tau_3 \\
& + x_1^T y_0 \sigma_1 \tau_0 + x_1^T y_1 \sigma_1 \tau_1 + x_1^T y_2 \sigma_1 \tau_2 + x_1^T y_3 \sigma_1 \tau_3 + \\
& + x_2^T y_0 \sigma_2 \tau_0 + x_2^T y_1 \sigma_2 \tau_1 + x_2^T y_2 \sigma_2 \tau_2 + x_2^T y_3 \sigma_2 \tau_3 + \\
& + x_3^T y_0 \sigma_3 \tau_0 + x_3^T y_1 \sigma_3 \tau_1 + x_3^T y_2 \sigma_3 \tau_2 + x_3^T y_3 \sigma_3 \tau_3 .
\end{aligned}
$$

$$
\underbrace{\qquad}_{\text{bin 0}} \quad \underbrace{\qquad}_{\text{bin 1}} \quad \underbrace{\qquad}_{\text{bin 2}} \quad \underbrace{\qquad}_{\text{bin 3}} \quad \underbrace{\qquad}_{\text{bin 4}} \quad \underbrace{\qquad}_{\text{bin 5}} \quad \underbrace{\qquad}_{\text{bin 6}}
$$

Computing $x_i^T y_j$ Exactly

To compute $x_i^T y_j$ exactly, for chunks $i, j \in \{0, 1, 2\}$:

$$x_i^T y_j = \chi_{0,i}\psi_{0,j} + \chi_{1,i}\psi_{1,j} + \cdots + \chi_{k-1,i}\psi_{k-1,j},$$

Computing $x_i^T y_j$ Exactly

To compute $x_i^T y_j$ exactly, for chunks $i, j \in \{0, 1, 2\}$:

$$x_i^T y_j = \chi_{0,i}\psi_{0,j} + \chi_{1,i}\psi_{1,j} + \cdots + \chi_{k-1,i}\psi_{k-1,j},$$

- Chunk each element in x_i into the same ranges.

Computing $x_i^T y_j$ Exactly

To compute $x_i^T y_j$ exactly, for chunks $i, j \in \{0, 1, 2\}$:

$$x_i^T y_j = \chi_{0,i}\psi_{0,j} + \chi_{1,i}\psi_{1,j} + \cdots + \chi_{k-1,i}\psi_{k-1,j},$$

- Chunk each element in x_i into the same ranges.
- Track the $\lceil \log_2 k \rceil$ additional bits that the addition of k terms in the dot product that might yield.

Computing $x_i^T y_j$ Exactly

To compute $x_i^T y_j$ exactly, for chunks $i, j \in \{0, 1, 2\}$:

$$x_i^T y_j = \chi_{0,i}\psi_{0,j} + \chi_{1,i}\psi_{1,j} + \cdots + \chi_{k-1,i}\psi_{k-1,j},$$

- Chunk each element in x_i into the same ranges.
- Track the $\lceil \log_2 k \rceil$ additional bits that the addition of k terms in the dot product that might yield.
- Track any additional bits that the addition of the terms within a bin may result in.
  For e.g. Bin 2 is when $i + j = 2$, and there may be several different x_i or y_j to add into this bin.

Tracking bits

- Given these constraints, each chunk can accommodate $22, 21, 21$, and 53 bits respectively. The first chunk is slightly larger because there's only one product in Bin 0.

Tracking bits

- Given these constraints, each chunk can accommodate $22, 21, 21$, and 53 bits respectively. The first chunk is slightly larger because there's only one product in Bin 0.
- Therefore, the total number of bits is now actually $22 + 21 + 21 + 53 = 117$ which is more than the number of bits in the mantissa of a FP64x2 number.

Tracking bits

- Given these constraints, each chunk can accommodate $22, 21, 21$, and 53 bits respectively. The first chunk is slightly larger because there's only one product in Bin 0.
- Therefore, the total number of bits is now actually $22 + 21 + 21 + 53 = 117$ which is more than the number of bits in the mantissa of a FP64x2 number.
- This means intermediate results are potentially accumulated in a precision higher than FP64x2 accommodates. But we may also potentially lose bits because of fixed point.

Additional Details

- If we chose our bit-contraints carefully (22,21,21,53), then summation within bins can be in FP64 arithmetic.

Additional Details

- If we chose our bit-contraints carefully (22,21,21,53), then summation within bins can be in FP64 arithmetic.
- Summation across bins 0 through 2 must be performed in FP64x2 addition or with "quick two-sum arithmetic"[1].

[1] T. Dekker, Numer. Math., 18 (1971), pp. 224–242.

Additional Details

- If we chose our bit-contraints carefully (22,21,21,53), then summation within bins can be in FP64 arithmetic.
- Summation across bins 0 through 2 must be performed in FP64x2 addition or with "quick two-sum arithmetic"[1].
- Summation across bins 3 through 6 can be done in FP64 arithmetic, since the last chunk is 53 bits, which implies that these terms are not even attempted to be done error-free.

[1] T. Dekker, Numer. Math., 18 (1971), pp. 224–242.

Additional Details

- If we chose our bit-contraints carefully (22,21,21,53), then summation within bins can be in FP64 arithmetic.
- Summation across bins 0 through 2 must be performed in FP64x2 addition or with "quick two-sum arithmetic"[1].
- Summation across bins 3 through 6 can be done in FP64 arithmetic, since the last chunk is 53 bits, which implies that these terms are not even attempted to be done error-free.
- To further preserve accuracy, the adding of contributions across bins starts with bin 6 and ends with bin 0.

[1] T. Dekker, Numer. Math., 18 (1971), pp. 224–242.

Reformulating the Cascading Dot Product

$$\left(\begin{array}{c} \sigma_0 x_0^T \\ \hline \sigma_1 x_1^T \\ \hline \sigma_2 x_2^T \\ \hline \sigma_3 x_3^T \end{array}\right) \left(\begin{array}{c|c|c|c} \tau_0 y_0 & \tau_1 y_1 & \tau_2 y_2 & \tau_3 y_3 \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \sigma_0\tau_0 x_0^T y_0 & \sigma_0\tau_1 x_0^T y_1 & \sigma_0\tau_2 x_0^T y_2 & \sigma_0\tau_3 x_0^T y_3 \\ \hline \sigma_1\tau_0 x_1^T y_0 & \sigma_1\tau_1 x_1^T y_1 & \sigma_1\tau_2 x_1^T y_2 & \sigma_1\tau_3 x_1^T y_3 \\ \hline \sigma_2\tau_0 x_2^T y_0 & \sigma_2\tau_1 x_2^T y_1 & \sigma_2\tau_2 x_2^T y_2 & \sigma_2\tau_3 x_2^T y_3 \\ \hline \sigma_3\tau_0 x_3^T y_0 & \sigma_3\tau_1 x_3^T y_1 & \sigma_3\tau_2 x_3^T y_2 & \sigma_3\tau_3 x_3^T y_3 \end{array}\right)$$

Ten Dot Products

$$
\begin{aligned}
\text{bin 3–6} = {} & x_0^T y_3 \sigma_0 \tau_3 + \\
& x_1^T y_2 \sigma_1 \tau_2 + \quad x_1^T y_3 \sigma_1 \tau_3 \quad + \\
& x_2^T y_1 \sigma_2 \tau_1 + \quad x_2^T y_2 \sigma_2 \tau_2 \quad + x_2^T y_3 \sigma_2 \tau_3 + \\
& \underbrace{x_3^T y_0 \sigma_3 \tau_0}_{\text{bin 3}} + \quad \underbrace{x_3^T y_1 \sigma_3 \tau_1}_{\text{bin 4}} \quad + \underbrace{x_3^T y_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{x_3^T y_3 \sigma_3 \tau_3}_{\text{bin 6}}
\end{aligned}
$$

Ten Dot Products

$$
\begin{aligned}
\text{bin 3–6} = {} & x_0^T y_3 \sigma_0 \tau_3 + \\
& x_1^T y_2 \sigma_1 \tau_2 + \; x_1^T y_3 \sigma_1 \tau_3 \; + \\
& x_2^T y_1 \sigma_2 \tau_1 + \; x_2^T y_2 \sigma_2 \tau_2 \; + x_2^T y_3 \sigma_2 \tau_3 + \\
& \underbrace{x_3^T y_0 \sigma_3 \tau_0}_{\text{bin 3}} + \; \underbrace{x_3^T y_1 \sigma_3 \tau_1}_{\text{bin 4}} \; + \underbrace{x_3^T y_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{x_3^T y_3 \sigma_3 \tau_3}_{\text{bin 6}} \\
= {} & x_0^T \sigma_0 y_3 \tau_3 + \\
& x_1^T \sigma_1 \; \underbrace{(y_2 \tau_2 + y_3 \tau_3)}_{y_4 \tau_4} \; + \\
& x_2^T \sigma_2 \; \underbrace{(y_1 \tau_1 + y_2 \tau_2 + y_3 \tau_3)}_{y_5 \tau_5} \; + \\
& x_3^T \sigma_3 \; \underbrace{(y_0 \tau_0 + y_1 \tau_1 + y_2 \tau_2 + y_3 \tau_3)}_{y_6 \tau_6} .
\end{aligned}
$$

Ten Dot Products (contd.)

$$\left(\begin{array}{c} \sigma_0 x_0^T \\ \hline \sigma_1 x_1^T \\ \hline \sigma_2 x_2^T \\ \hline \sigma_3 x_3^T \end{array}\right) \quad \left(\begin{array}{c|c|c|c||c|c|c} \tau_0 y_0 & \tau_1 y_1 & \tau_2 y_2 & \tau_3 y_3 & \tau_4 y_4 & \tau_5 y_5 & \tau_6 y_6 \end{array}\right)$$

Ten Dot Products (contd.)

$$\left(\begin{array}{c} \sigma_0 x_0^T \\ \hline \sigma_1 x_1^T \\ \hline \sigma_2 x_2^T \\ \hline \sigma_3 x_3^T \end{array}\right) \left(\begin{array}{c|c|c|c||c|c|c} \tau_0 y_0 & \tau_1 y_1 & \tau_2 y_2 & \tau_3 y_3 & \tau_4 y_4 & \tau_5 y_5 & \tau_6 y_6 \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c||c|c|c} \sigma_0\tau_0 x_0^T y_0 & \sigma_0\tau_1 x_0^T y_1 & \sigma_0\tau_2 x_0^T y_2 & \sigma_0\tau_3 x_0^T y_3 & \star & \star & \star \\ \hline \sigma_1\tau_0 x_1^T y_0 & \sigma_1\tau_1 x_1^T y_1 & \star & \star & \sigma_1\tau_4 x_1^T y_4 & \star & \star \\ \hline \sigma_2\tau_0 x_2^T y_0 & \star & \star & \star & \star & \sigma_2\tau_5 x_2^T y_5 & \star \\ \hline \star & \star & \star & \star & \star & \star & \sigma_3\tau_6 x_3^T y_6 \end{array}\right)$$

Cascading matrices and GEMM

Now, consider $C = AB$, where the "inner size" of the matrix is restricted to k.

Cascading matrices and GEMM

Now, consider $C = AB$, where the "inner size" of the matrix is restricted to k.

Matrices A and B can be chunked as follows:

Cascading matrices and GEMM

Now, consider $C = AB$, where the "inner size" of the matrix is restricted to k.

Matrices A and B can be chunked as follows:

$$A = A_0\sigma_0 + A_1\sigma_1 + A_2\sigma_2 + A_3\sigma_3$$
$$B = B_0\,\tau_0 + B_1\,\tau_1 + B_2\,\tau_2 + B_3\,\tau_3$$

Cascading Matrices and GEMM (contd.)

$$
\begin{aligned}
AB = {} & \underbrace{A_0 B_0 \sigma_0 \tau_0}_{\text{bin } 0} + \underbrace{A_0 B_1 \sigma_0 \tau_1 + A_1 B_0 \sigma_1 \tau_0}_{\text{bin } 1} + \underbrace{A_0 B_2 \sigma_0 \tau_2 + A_1 B_1 \sigma_1 \tau_1 + A_2 B_0 \sigma_2 \tau_0}_{\text{bin } 2} \\
& + \underbrace{A_0 B_3 \sigma_0 \tau_3 + A_1 B_2 \sigma_1 \tau_2 + A_2 B_1 \sigma_2 \tau_1 + A_3 B_0 \sigma_3 \tau_0}_{\text{bin } 3} + \underbrace{A_1 B_3 \sigma_1 \tau_3 + A_2 B_2 \sigma_2 \tau_2 + A_3 B_1 \sigma_3 \tau_1}_{\text{bin } 4} \\
& + \underbrace{A_2 B_3 \sigma_2 \tau_3 + A_3 B_2 \sigma_3 \tau_2}_{\text{bin } 5} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin } 6}.
\end{aligned}
$$

Cascading Matrices and GEMM (contd.)

$$
\begin{aligned}
AB = {} & \underbrace{A_0 B_0 \sigma_0 \tau_0}_{\text{bin 0}} + \underbrace{A_0 B_1 \sigma_0 \tau_1 + A_1 B_0 \sigma_1 \tau_0}_{\text{bin 1}} + \underbrace{A_0 B_2 \sigma_0 \tau_2 + A_1 B_1 \sigma_1 \tau_1 + A_2 B_0 \sigma_2 \tau_0}_{\text{bin 2}} \\
& + \underbrace{A_0 B_3 \sigma_0 \tau_3 + A_1 B_2 \sigma_1 \tau_2 + A_2 B_1 \sigma_2 \tau_1 + A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \underbrace{A_1 B_3 \sigma_1 \tau_3 + A_2 B_2 \sigma_2 \tau_2 + A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} \\
& + \underbrace{A_2 B_3 \sigma_2 \tau_3 + A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}}.
\end{aligned}
$$

We can approximate the FP64x2 GEMM in terms of 16 FP64 GEMMs, extending the observations about dot products.
This means that

Cascading Matrices and GEMM (contd.)

$$
\begin{aligned}
AB = {} & \underbrace{A_0 B_0 \sigma_0 \tau_0}_{\text{bin 0}} + \underbrace{A_0 B_1 \sigma_0 \tau_1 + A_1 B_0 \sigma_1 \tau_0}_{\text{bin 1}} + \underbrace{A_0 B_2 \sigma_0 \tau_2 + A_1 B_1 \sigma_1 \tau_1 + A_2 B_0 \sigma_2 \tau_0}_{\text{bin 2}} \\
& + \underbrace{A_0 B_3 \sigma_0 \tau_3 + A_1 B_2 \sigma_1 \tau_2 + A_2 B_1 \sigma_2 \tau_1 + A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \underbrace{A_1 B_3 \sigma_1 \tau_3 + A_2 B_2 \sigma_2 \tau_2 + A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} \\
& + \underbrace{A_2 B_3 \sigma_2 \tau_3 + A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}}.
\end{aligned}
$$

We can approximate the FP64x2 GEMM in terms of 16 FP64 GEMMs, extending the observations about dot products.
This means that

- D_0, D_1, and D_2 are picked as discussed in previous slides.

Cascading Matrices and GEMM (contd.)

$$
\begin{aligned}
AB = {} & \underbrace{A_0 B_0 \sigma_0 \tau_0}_{\text{bin 0}} + \underbrace{A_0 B_1 \sigma_0 \tau_1 + A_1 B_0 \sigma_1 \tau_0}_{\text{bin 1}} + \underbrace{A_0 B_2 \sigma_0 \tau_2 + A_1 B_1 \sigma_1 \tau_1 + A_2 B_0 \sigma_2 \tau_0}_{\text{bin 2}} \\
& + \underbrace{A_0 B_3 \sigma_0 \tau_3 + A_1 B_2 \sigma_1 \tau_2 + A_2 B_1 \sigma_2 \tau_1 + A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \underbrace{A_1 B_3 \sigma_1 \tau_3 + A_2 B_2 \sigma_2 \tau_2 + A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} \\
& + \underbrace{A_2 B_3 \sigma_2 \tau_3 + A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}}.
\end{aligned}
$$

We can approximate the FP64x2 GEMM in terms of 16 FP64 GEMMs, extending the observations about dot products.
This means that

- D_0, D_1, and D_2 are picked as discussed in previous slides.
- Elements within rows of A must be chunked conformally.

Cascading Matrices and GEMM (contd.)

$$
\begin{aligned}
AB = {} & \underbrace{A_0 B_0 \sigma_0 \tau_0}_{\text{bin 0}} + \underbrace{A_0 B_1 \sigma_0 \tau_1 + A_1 B_0 \sigma_1 \tau_0}_{\text{bin 1}} + \underbrace{A_0 B_2 \sigma_0 \tau_2 + A_1 B_1 \sigma_1 \tau_1 + A_2 B_0 \sigma_2 \tau_0}_{\text{bin 2}} + \\
& \underbrace{A_0 B_3 \sigma_0 \tau_3 + A_1 B_2 \sigma_1 \tau_2 + A_2 B_1 \sigma_2 \tau_1 + A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \underbrace{A_1 B_3 \sigma_1 \tau_3 + A_2 B_2 \sigma_2 \tau_2 + A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} + \\
& \underbrace{A_2 B_3 \sigma_2 \tau_3 + A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}}.
\end{aligned}
$$

We can approximate the FP64x2 GEMM in terms of 16 FP64 GEMMs, extending the observations about dot products.
This means that

- D_0, D_1, and D_2 are picked as discussed in previous slides.
- Elements within rows of A must be chunked conformally.
- Elements within columns of B must be chunked conformally.

Cascading Matrices and GEMM (contd.)

$$
\begin{aligned}
AB = {} & \underbrace{A_0 B_0 \sigma_0 \tau_0}_{\text{bin 0}} + \underbrace{A_0 B_1 \sigma_0 \tau_1 + A_1 B_0 \sigma_1 \tau_0}_{\text{bin 1}} + \underbrace{A_0 B_2 \sigma_0 \tau_2 + A_1 B_1 \sigma_1 \tau_1 + A_2 B_0 \sigma_2 \tau_0}_{\text{bin 2}} + \\
& \underbrace{A_0 B_3 \sigma_0 \tau_3 + A_1 B_2 \sigma_1 \tau_2 + A_2 B_1 \sigma_2 \tau_1 + A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \underbrace{A_1 B_3 \sigma_1 \tau_3 + A_2 B_2 \sigma_2 \tau_2 + A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} + \\
& \underbrace{A_2 B_3 \sigma_2 \tau_3 + A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}}.
\end{aligned}
$$

We can approximate the FP64x2 GEMM in terms of 16 FP64 GEMMs, extending the observations about dot products.
This means that

- D_0, D_1, and D_2 are picked as discussed in previous slides.
- Elements within rows of A must be chunked conformally.
- Elements within columns of B must be chunked conformally.
- The terms in bins 0–2 are computed exactly.

Ten GEMMs

$$
\begin{aligned}
\text{bin 3–6} = {} & A_0 B_3 \sigma_0 \tau_3 + \\
& A_1 B_2 \sigma_1 \tau_2 + \; A_1 B_3 \sigma_1 \tau_3 \; + \\
& A_2 B_1 \sigma_2 \tau_1 + \; A_2 B_2 \sigma_2 \tau_2 \; + A_2 B_3 \sigma_2 \tau_3 + \\
& \underbrace{A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \; \underbrace{A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} \; + \underbrace{A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}}
\end{aligned}
$$

Ten GEMMs

$$
\begin{aligned}
\text{bin 3–6} = {} & A_0 B_3 \sigma_0 \tau_3 + \\
& A_1 B_2 \sigma_1 \tau_2 + A_1 B_3 \sigma_1 \tau_3 + \\
& A_2 B_1 \sigma_2 \tau_1 + A_2 B_2 \sigma_2 \tau_2 + A_2 B_3 \sigma_2 \tau_3 + \\
& \underbrace{A_3 B_0 \sigma_3 \tau_0}_{\text{bin 3}} + \underbrace{A_3 B_1 \sigma_3 \tau_1}_{\text{bin 4}} + \underbrace{A_3 B_2 \sigma_3 \tau_2}_{\text{bin 5}} + \underbrace{A_3 B_3 \sigma_3 \tau_3}_{\text{bin 6}} \\
= {} & A_0 \sigma_0 B_3 \tau_3 + \\
& A_1 \sigma_1 \underbrace{(B_2 \tau_2 + B_3 \tau_3)}_{B_4 \tau_4} + \\
& A_2 \sigma_2 \underbrace{(B_1 \tau_1 + B_2 \tau_2 + B_3 \tau_3)}_{B_5 \tau_5} + \\
& A_3 \sigma_3 \underbrace{(B_0 \tau_0 + B_1 \tau_1 + B_2 \tau_2 + B_3 \tau_3)}_{B_6 \tau_6} .
\end{aligned}
$$

Ten GEMMs

$$\left(\begin{array}{c} \sigma_0 A_0 \\ \hline \sigma_1 A_1 \\ \hline \sigma_2 A_2 \\ \hline \sigma_3 A_3 \end{array}\right) \left(\begin{array}{c|c|c|c||c|c|c} \tau_0 B_0 & \tau_1 B_1 & \tau_2 B_2 & \tau_3 B_3 & \tau_2 B_4 & \tau_1 B_5 & \tau_0 B_6 \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c||c|c|c} \sigma_0\tau_0 A_0 B_0 & \sigma_0\tau_1 A_0 B_1 & \sigma_0\tau_2 A_0 B_2 & \sigma_0\tau_3 A_0 B_3 & \star & \star & \star \\ \hline \sigma_1\tau_0 A_1 B_0 & \sigma_1\tau_1 A_1 B_1 & \star & \star & \sigma_1\tau_4 A_1 B_4 & \star & \star \\ \hline \sigma_2\tau_0 A_2 B_0 & \star & \star & \star & \star & \sigma_2\tau_5 A_2 B_5 & \star \\ \hline \star & \star & \star & \star & \star & \star & \sigma_3\tau_6 A_3 B_6 \end{array}\right)$$

Implementation

A Näive Approach

- Convert A and B into cascading matrices upfront.
- Use GEMM to compute the various cross products.
- Compute the resulting C.

This approach does not fully reuse data brought through the memory hierarchy.

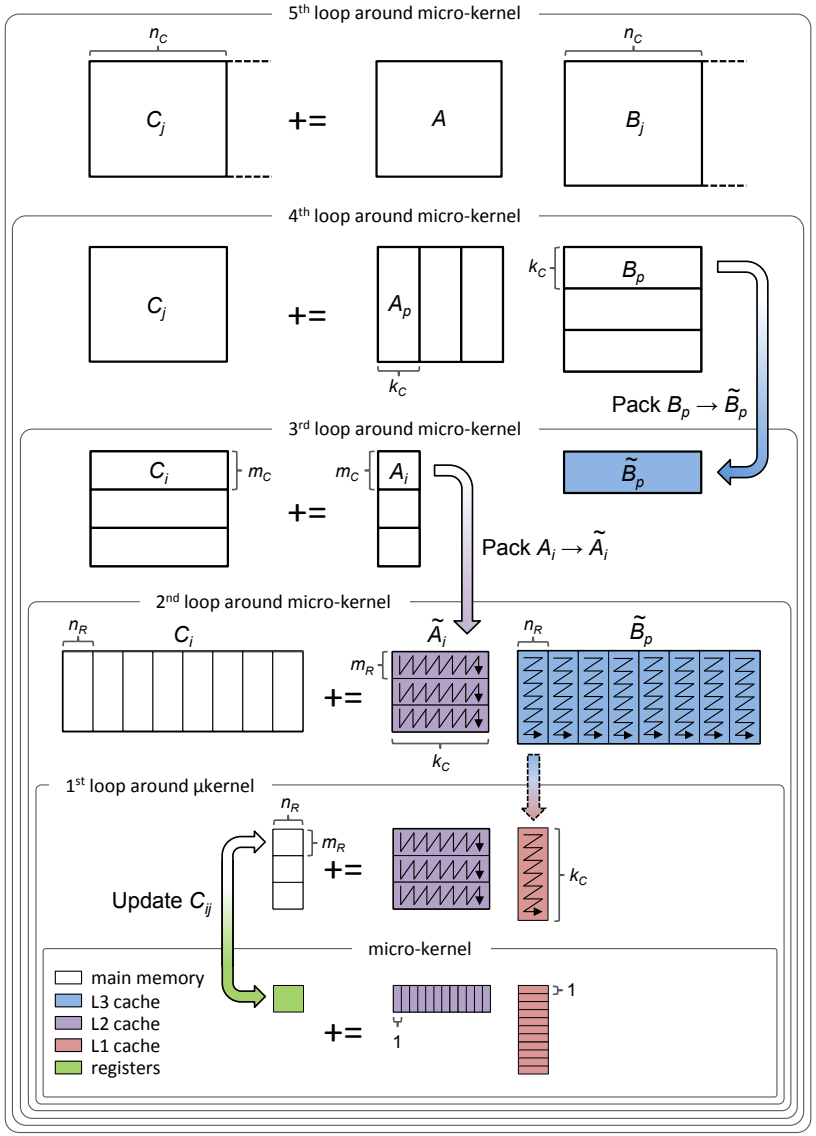
A Näive Approach

- Convert A and B into cascading matrices upfront.
- Use GEMM to compute the various cross products.
- Compute the resulting C.

This approach does not fully reuse data brought through the memory hierarchy.

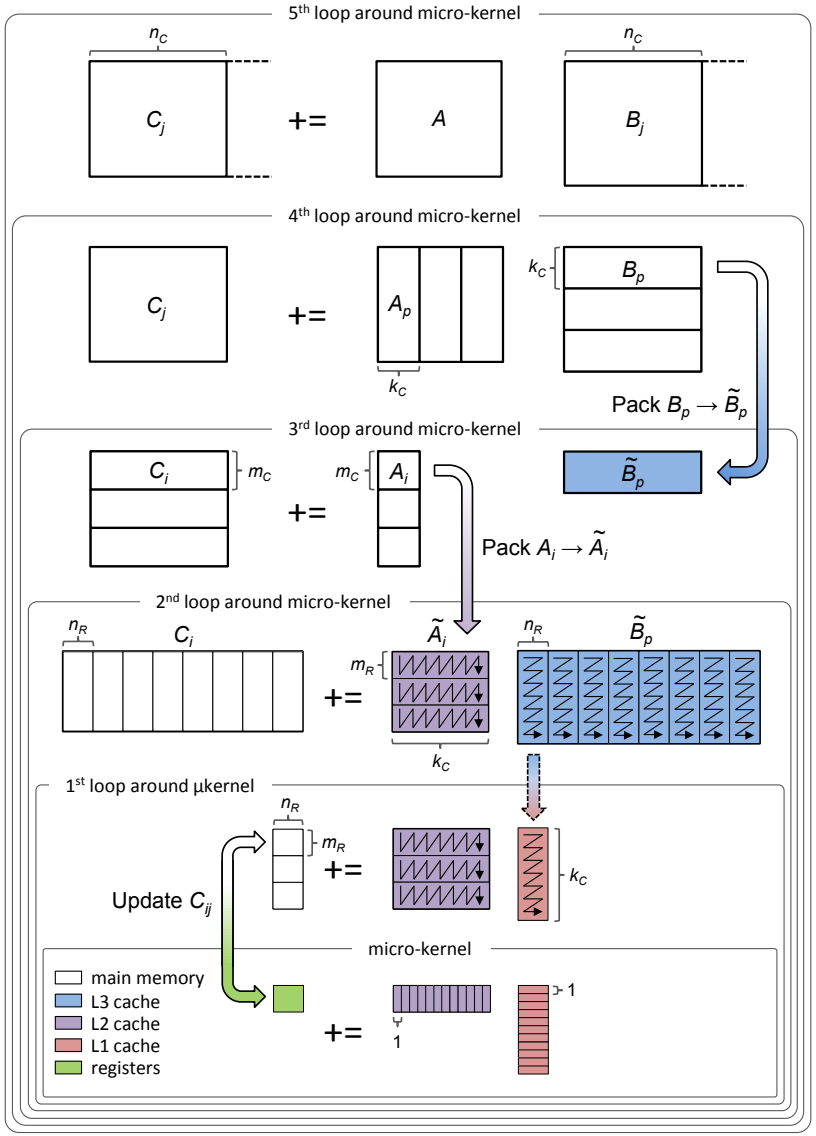
Using BLIS, we can do better!

High-Performance GEMM



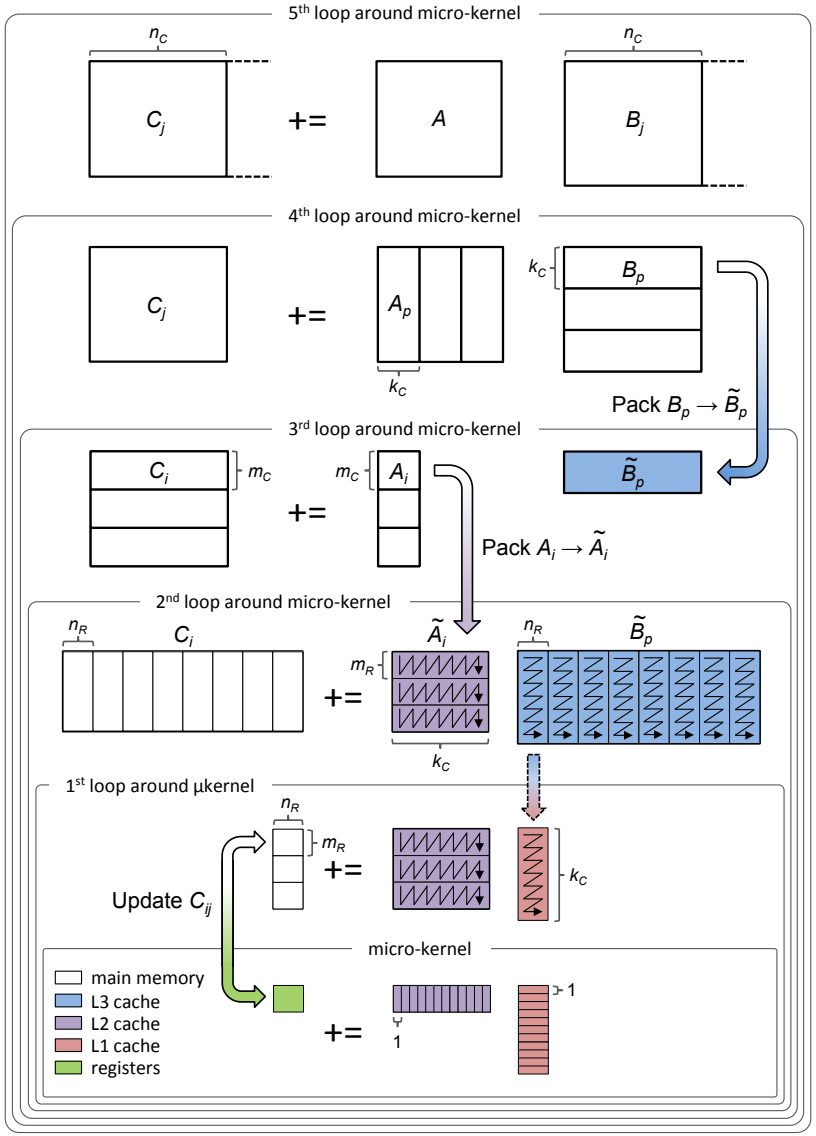
- Most of the library is written in C99.

High-Performance GEMM



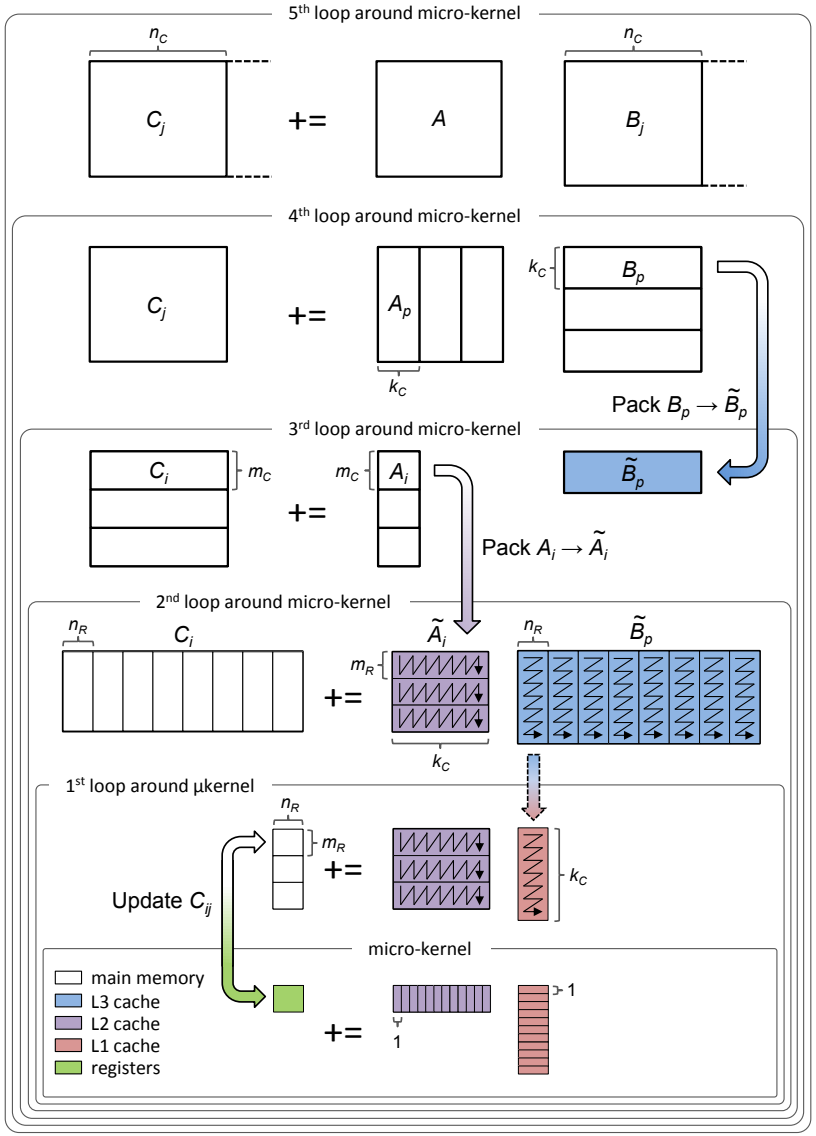
- Most of the library is written in C99.
- BLIS isolates the computation kernel (microkernel) to one small loop around a rank-1 update written in assembly.

High-Performance GEMM



- Most of the library is written in C99.
- BLIS isolates the computation kernel (microkernel) to one small loop around a rank-1 update written in assembly.
- Blocks for various levels of cache.

High-Performance GEMM



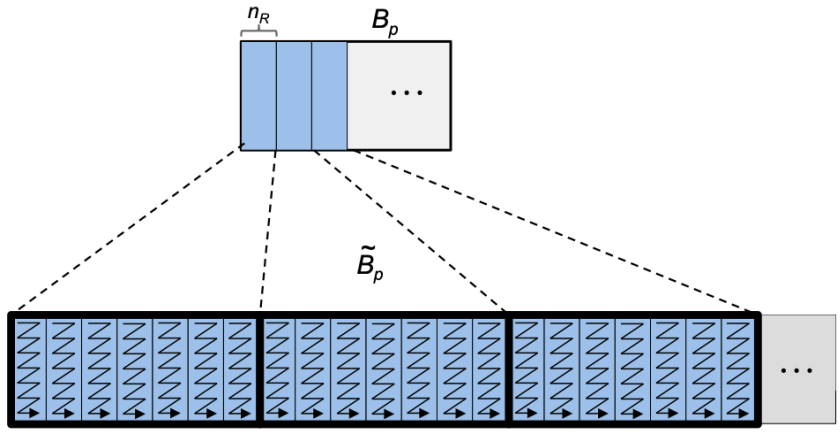
- Most of the library is written in C99.
- BLIS isolates the computation kernel (microkernel) to one small loop around a rank-1 update written in assembly.
- Blocks for various levels of cache.
- Packs blocks of A and B to improve data locality.

Phase 1: Converting and Packing the Cascading Matrices

- Phase 0: Compute the maximum absolute value for each row of A and column of B. Each row of A and column of B is quantized independently.

Phase 1: Converting and Packing the Cascading Matrices

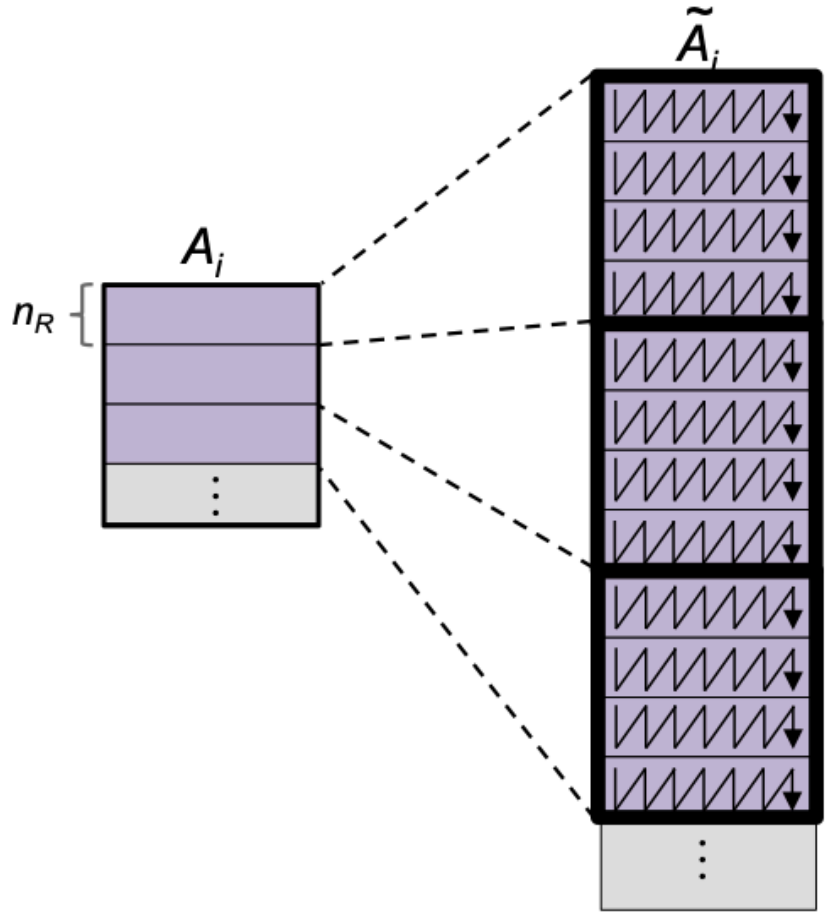
- Phase 0: Compute the maximum absolute value for each row of A and column of B. Each row of A and column of B is quantized independently.
- Phase 1: Convert each entry of `FP64x2` B and A into its corresponding entry of the cascading matrix and then pack these cascading matrices in the data layout required for BLIS.



Packing layout for a panel of B.

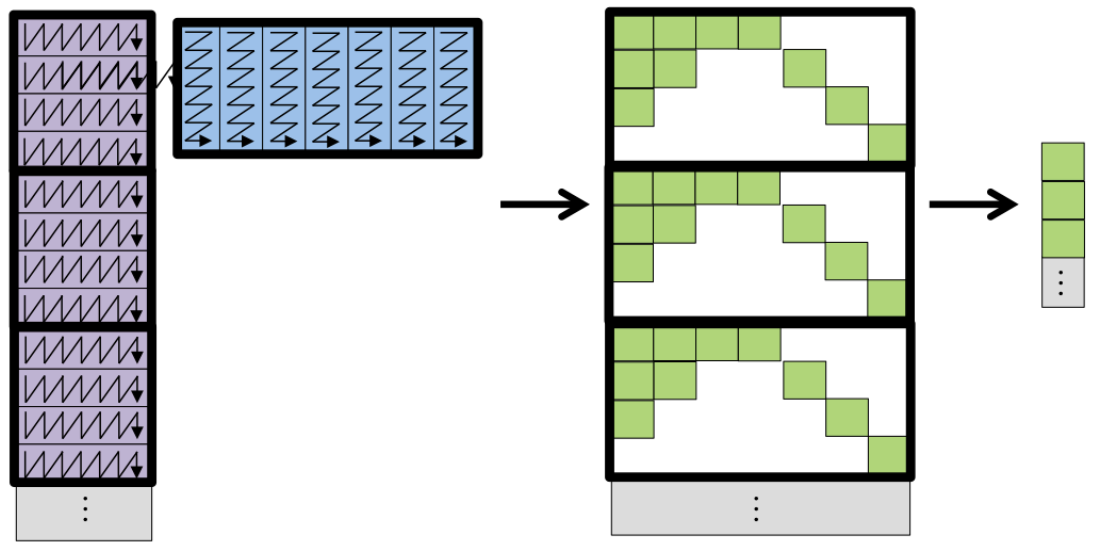
Phase 1: Converting and Packing the Cascading Matrices

- Phase 0: Compute the maximum absolute value for each row of A and column of B. Each row of A and column of B is quantized independently.
- Phase 1: Convert each entry of `FP64x2` B and A into its corresponding entry of the cascading matrix and then pack these cascading matrices in the data layout required for BLIS.



Packing layout for a block of A.

Phase 2: `dgemm()`



Operation performed with cascaded matrices by first loop around the micro-kernel.

Phase 3: Putting it back together

- The accumulation back to C is done after the first loop around the microkernel.

Phase 3: Putting it back together

- The accumulation back to C is done after the first loop around the microkernel.
- This loop is selected to ensure the temporary buffers of C fit in the L2 cache.

Phase 3: Putting it back together

- The accumulation back to C is done after the first loop around the microkernel.
- This loop is selected to ensure the temporary buffers of C fit in the L2 cache.
- The accumulation is done using a "quick two-sum"[2] addition.

[2] T. Dekker, Numer. Math., 18 (1971), pp. 224–242.

Performance experiments

Setup

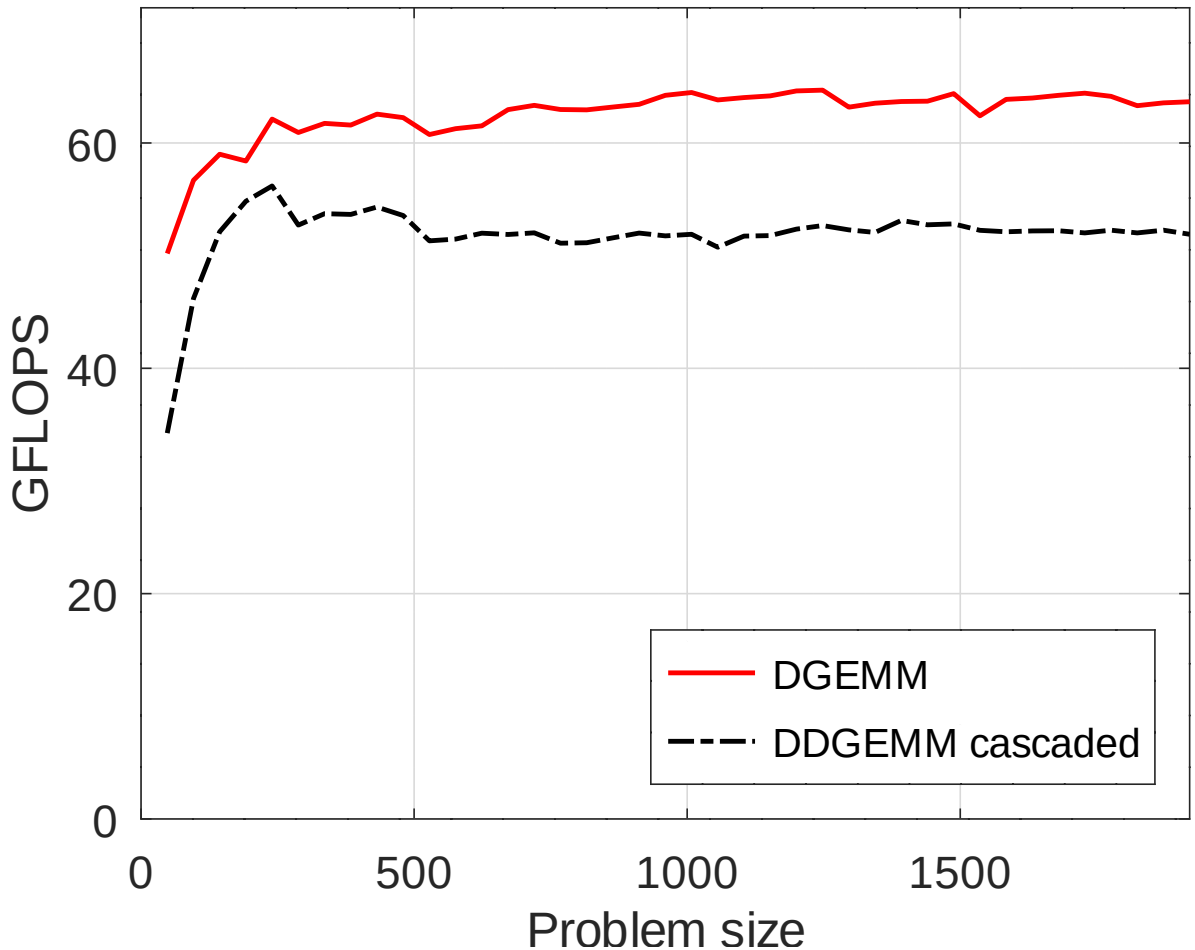
Platform

- Intel Core i7-7700K CPU with 4 cores.
- Each core runs at 4.20 GHz with a max turbo frequency of 4.50 GHz.
- A single-core peak performance of 72 GFLOPS in double precision.

Comparison

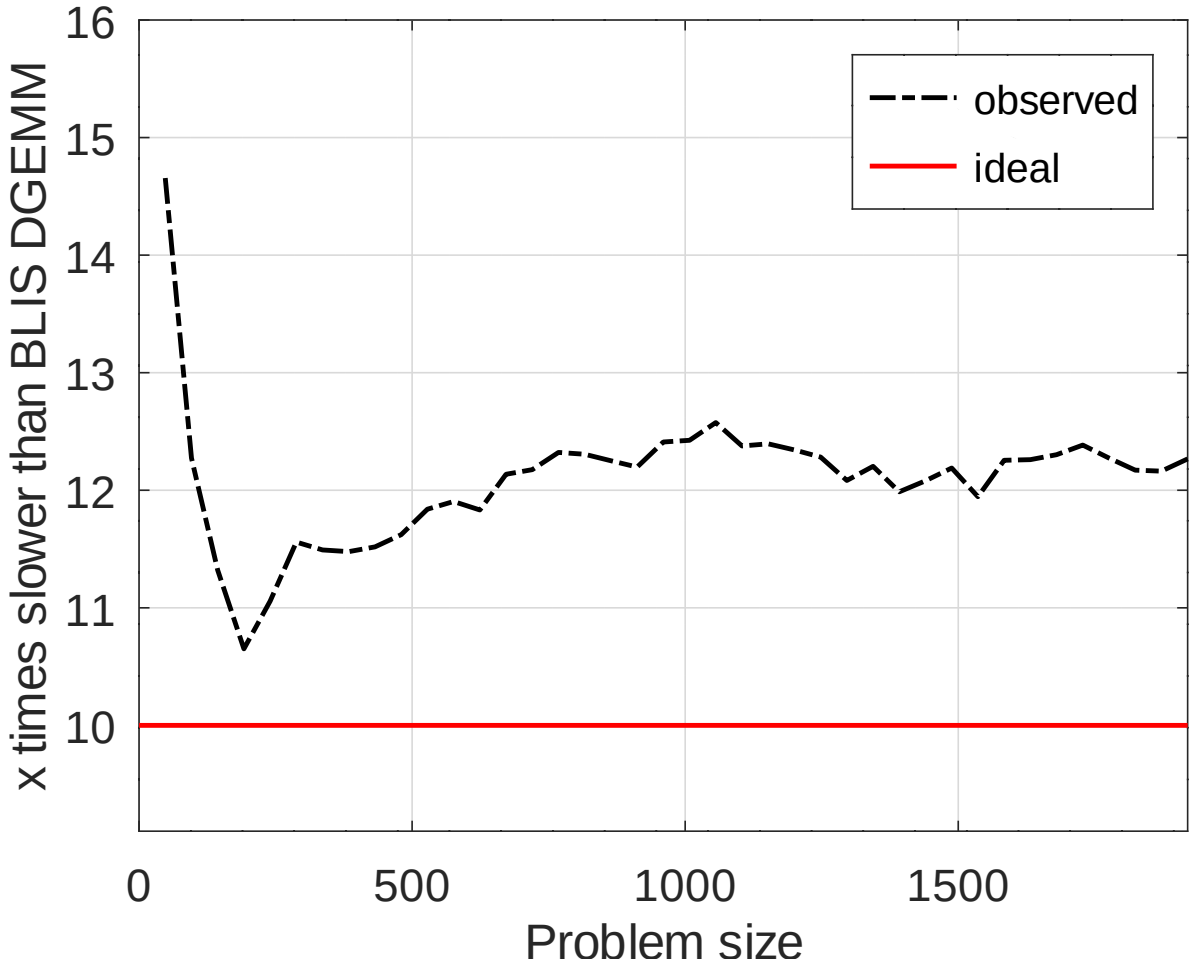
- The best comparison is against DGEMM performance.
- if our algorithm runs 10x slower than DGEMM, this has a lot more meaning than just saying it has 10x the flop count.

Performance (GFLOPS)



GFLOPS attained for both BLIS DGEMM and FP64x2 cascaded matrices at various problem sizes ($m = n = k$).

Performance–Slowdown compared to DGEMM



Ratio of the execution time of FP64x2 cascaded matrix GEMM vs BLIS DGEMM. Ideally, 10x slowdown is expected.

Accuracy experiments

Types of Accuracy Experiments

- GEMM with matrices filled with uniformly distributed random data in a given range.
- Matrices constructed so that GEMM is ill-conditioned.

Uniformly Random Data

Generating Data

- Pick a random `FP64` number in a given range.
- Add random bits in the mantissa past the 53-rd bit, upto 113-th bit.
- This becomes an `FP128` number in the prescribed range.
- This `FP128` number is converted into non-overlapping `FP64x2` format.

Ill-Conditioning Experiments

- Expect error to be worst when so-called catastrophic cancellation is encountered.

Ill-Conditioning Experiments

- Expect error to be worst when so-called catastrophic cancellation is encountered.
- Generate A and B s.t we get ill-conditioned results at most entries of $C = AB$.

Ill-Conditioning Experiments

- Expect error to be worst when so-called catastrophic cancellation is encountered.
- Generate A and B s.t we get ill-conditioned results at most entries of $C = AB$.
- Compare our algorithm with a triple-nested loop `FP64x2` GEMM.

Ill-Conditioning Experiments

- Expect error to be worst when so-called catastrophic cancellation is encountered.
- Generate A and B s.t we get ill-conditioned results at most entries of $C = AB$.
- Compare our algorithm with a triple-nested loop `FP64x2` GEMM.
- Gold standard: `FP128x2` GEMM.

Ill-Conditioning Experiments

- Expect error to be worst when so-called catastrophic cancellation is encountered.
- Generate A and B s.t we get ill-conditioned results at most entries of $C = AB$.
- Compare our algorithm with a triple-nested loop `FP64x2` GEMM.
- Gold standard: `FP128x2` GEMM.
- To show the results, we show the ratio of the maximum relative component-wise computed error.

Ill-conditioned Experiments

- Each entry of $C = AB$ is a dot product of a row of A with a column of B.
- Examine the case when a dot product of two vector, x and y, with reasonable length yields a small result. Recall that this happens when, for example, $\|x\|_2 \approx \|y\|_2 = 1$ and $x^T y$ is small.
- Existing (XBLAS) generators that create matrices so that multiplication with them is ill-conditioned are dot product generators. Therefore, the matrices generated via this technique leads to problems on the diagonal alone.

Generating Ill-Conditioned Matrices

- Assume that the matrices are square ($m = n = k$).

Generating Ill-Conditioned Matrices

- Assume that the matrices are square ($m = n = k$).
- Create A to have mutually orthogonal rows.
  - Generate a random matrix, compute its QR factorization, and setting A equal to the resulting Q.

Generating Ill-Conditioned Matrices

- Assume that the matrices are square ($m = n = k$).
- Create A to have mutually orthogonal rows.
  - Generate a random matrix, compute its QR factorization, and setting A equal to the resulting Q.
- Generate a matrix $\widehat{C}$
  - Generate each element such that it has magnitude in the range $(t, 10t)$, where t is some small positive tolerance.
  - Randomly pick the sign of each element.
  - Randomly pick one location in every column of $\widehat{C}$ to be one.

Generating Ill-Conditioned Matrices

- Assume that the matrices are square ($m = n = k$).
- Create A to have mutually orthogonal rows.
  - Generate a random matrix, compute its QR factorization, and setting A equal to the resulting Q.
- Generate a matrix $\widehat{C}$
  - Generate each element such that it has magnitude in the range $(t, 10t)$, where t is some small positive tolerance.
  - Randomly pick the sign of each element.
  - Randomly pick one location in every column of $\widehat{C}$ to be one.
- Set $B = Q^T \widehat{C}$.

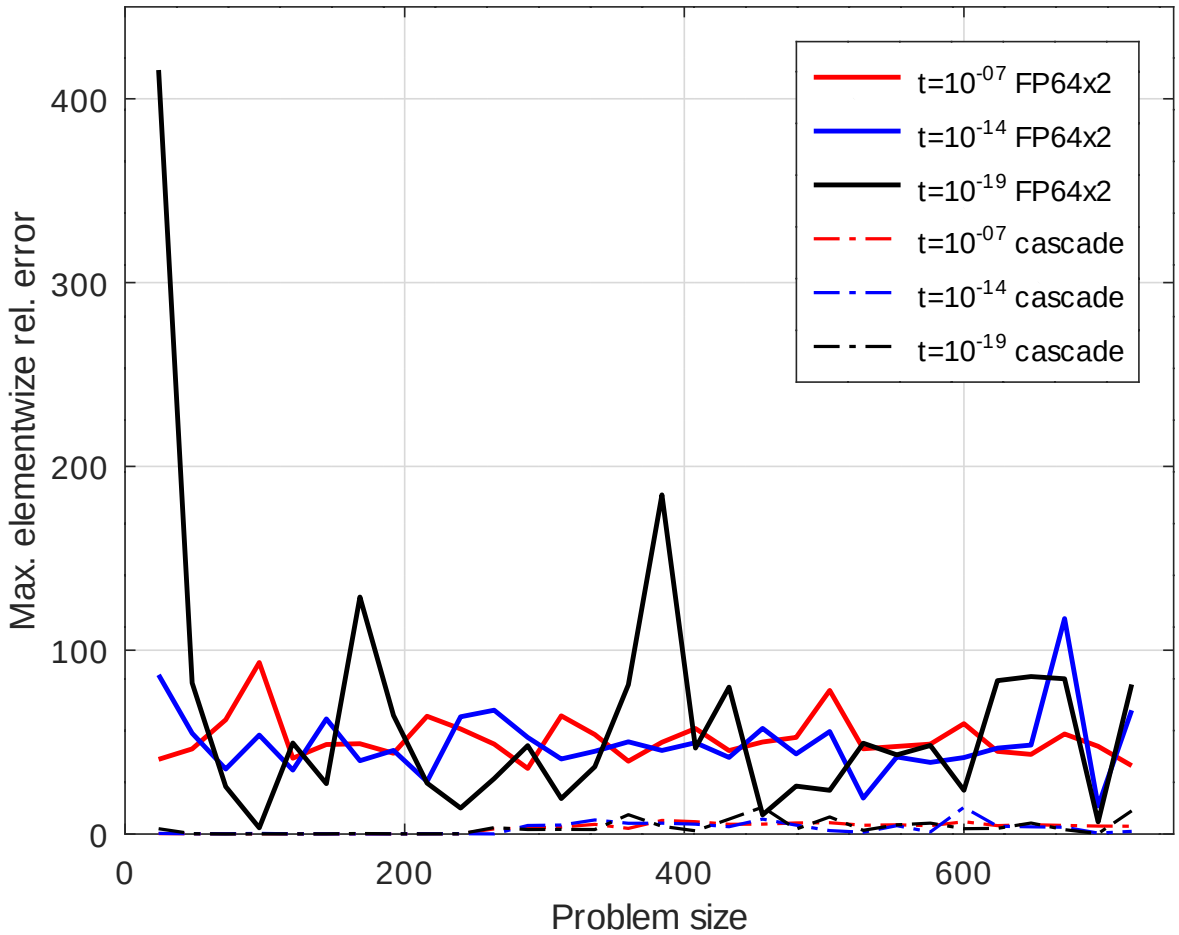
Ill-Conditioned Experiments

In our experiments, we focus on three choices for tolerance t:

- $t = 10^{-9}$, which can wipe out all accuracy when computing in single precision (FP32).
- $t = 10^{-14}$, which can wipe out all accuracy when computing in double precision (FP64).
- $t = 10^{-19}$, which can lead to significant loss of accuracy when computing in double-double precision (FP64x2), but will leave some accuracy.

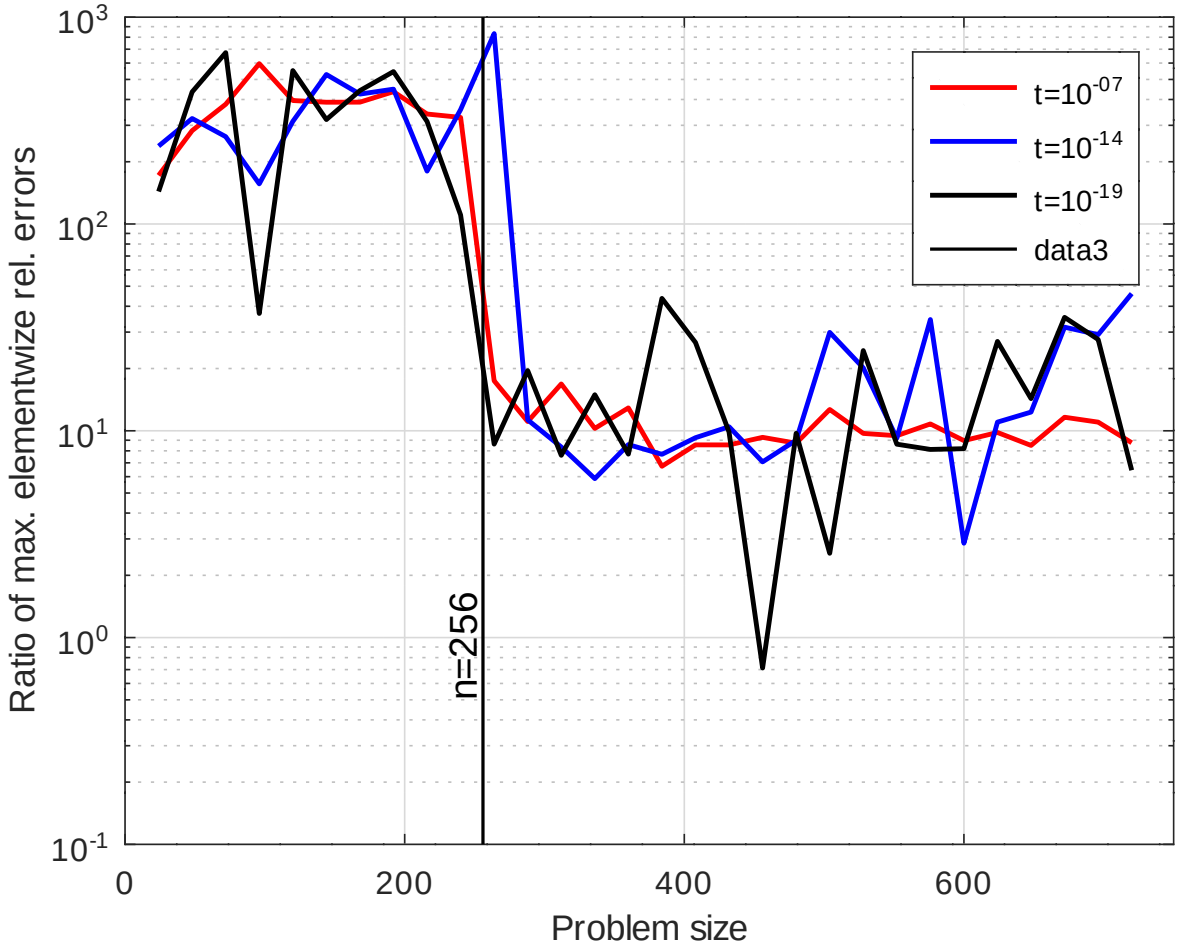
Ran experiments for a range of problem sizes, reporting the average worst case component-wise errors over many runs.

Results



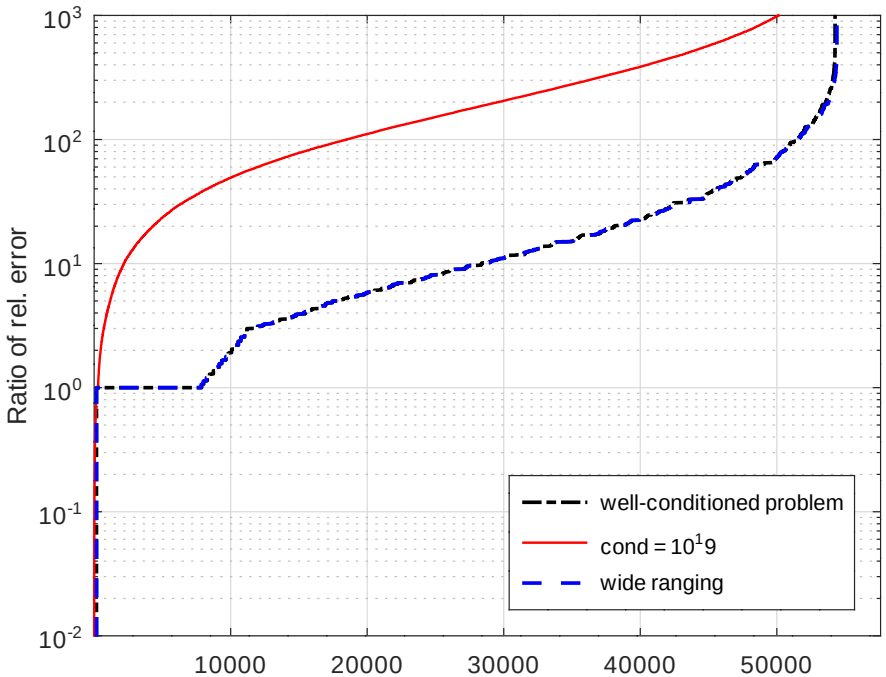
Maximum component-wise error for various choices of tolerance, for computation with `FP64x2` arithmetic and with cascaded matrices.

Results



Same data, presented as the ratio of error when computing with `FP64x2` arithmetic and with cascaded multiplication.

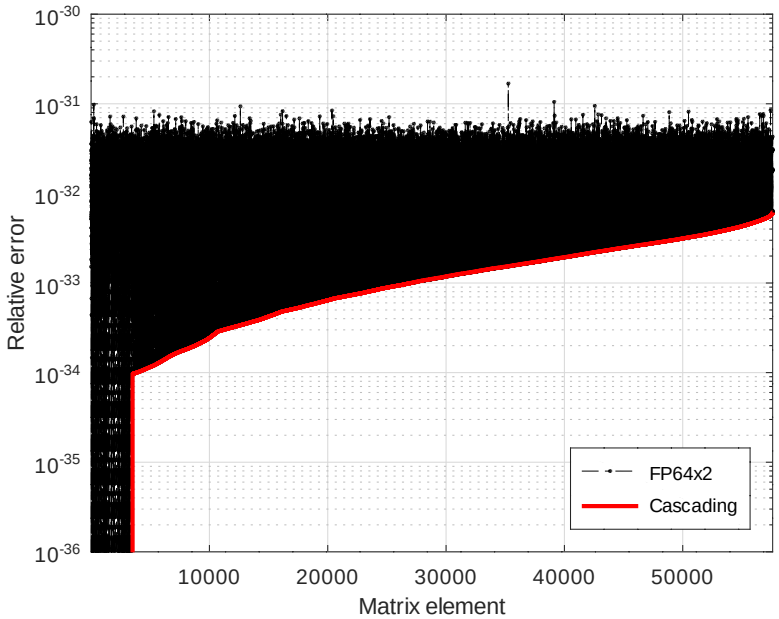
Element-wise Accuracy Results



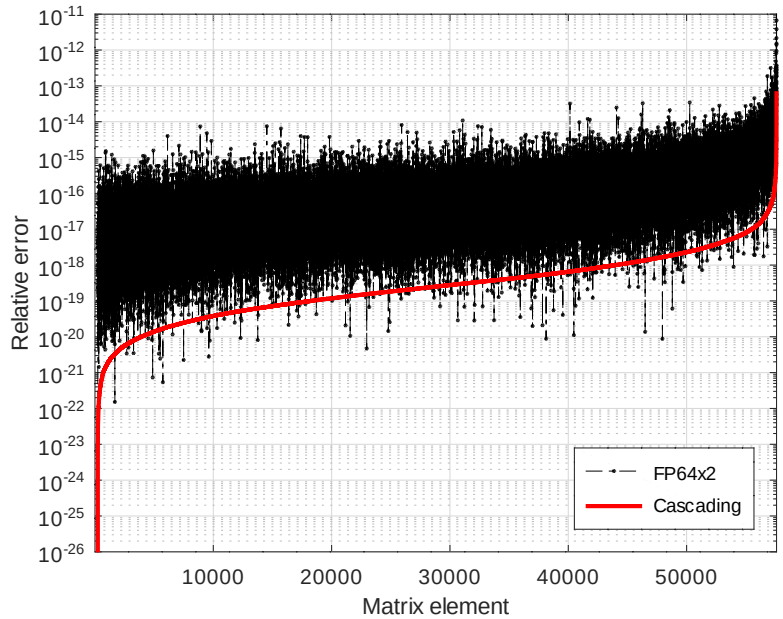
- The ratios (Cascaded divided by `FP64x2`) of relative accuracies of all elements of matrix $C = AB$ are reported, where all matrices involved are 240×240.
- The elements of the results in each experiment are sorted by the value of the ratio.
- Values less than 10^0 mean that `FP64x2` is more accurate.

Element-wise Accuracy Results

- We report the relative error of all elements of the result for $m = n = k = 240$.
- The elements are sorted by the relative error incurred by the cascading matrix multiplication.
- The corresponding error incurred by FP64x2 is also reported.



well-conditioned experiments



experiments with conditioning in the 10^{19} range

Opportunities

Opportunities

- More generally cast high precision in terms of low precision.
- Mix operand precision within cascaded matrix multiplication.
- Scale and balance the matrices to limit absorption errors.
- Drop lower bins when doing lower precision.
- Auto-correction.
- Threading opportunities.

Summary

- Cascading matrices provides the opportunities to compute BLAS operations at higher precision, using lower precision computations.
- What would we need to support cascading matrices in BLIS?

Thank you!

Backup Slides

Generalizing and Mixing Precisions

- `FP32` in terms of `FP16`: cascading matrices of `FP16x2` or `BFLOAT16x3`.
- `FP64` in terms of lower precision: cascading matrices of `FP32x2`, `INT8x7`, `FP16x5`, `BFLOAT16x6`.
- Mixed Precision: If A is `FP16`, and B is `FP64` , BLIS normally would promote A to FP64, and call DGEMM. Instead, we could transform B into `FP16x5`, and do 5 `FP16`-GEMMs instead.

Scale and Balance

Let,

$$A = \begin{pmatrix} 1 & \beta \\ 1 & \epsilon \end{pmatrix} B = \begin{pmatrix} 1 & \delta \\ 0.5 & 1 \end{pmatrix}$$

Now AB has $\delta + \epsilon$ in the (2,2) position.

If these terms are too small, then we may end up dropping bits and the relative accuracy of this computation will be lost.

But if we scale the columns of A by a diagonal matrix F (consisting of powers of two) and the rows of B by F^{-1}, we might achieve a better balance.

Drop Lower Bins

If A, B is `BFLOAT16x3` with `FP32` accumulation, then AB can simulate SGEMM.
The same "cascading matrix" code can be used to mimic more than just QGEMM, but SGEMM or DGEMM or even BF16GEMM.

Auto-correction

Threading Concepts

if A and B are too small to thread, perhaps one can also thread over the number of multiplies when one breaks this into cascading equations.

Generated Ill-Conditioned Matrix Observations

- In exact arithmetic $\widehat{C} = AB$.
- The columns of $\widehat{C}$ have length approximately equal to one and hence so do the columns of B, since multiplication by a unitary matrix Q^T preserves length.
- The computation of AB inherently involves many dot products that have the desired property of triggering cancellation.

Badness Case

Consider, $A = \begin{pmatrix} 1 & \epsilon \\ \epsilon & 1 \end{pmatrix}$ and $B = \begin{pmatrix} \epsilon & \epsilon \\ 1 & 1 \end{pmatrix}$.

When using these proposed techniques, such a scenario can result in catastrophically inaccurate answers.

We can minimize hitting this badness case in general by quantizing every row of A and every column of B separately.

We can auto-detect this badness case (computationally free), and report this has occurred.