Mixing domains and precisions in BLIS: Initial thoughts

Field G. Van Zee

Science of High Performance Computing

The University of Texas at Austin

The Problem

- gemm
  - $C := \beta C + \alpha AB$
- Let's simplify by omitting scalars
  - $C := C + AB$
- Recall: BLAS requires A, B, and C to be stored as the **same datatype** (precision and domain)
  - single real, double real, single complex, double complex
- What if we could lift this constraint?

The Precedent

- gemm
  - $C := \beta C + \alpha AB$
- BLAS requires
  - A, B, and C to be column-stored
- CBLAS requires
  - A, B, and C to be column-stored, OR…
  - A, B, and C to be row-stored
- BLIS allows
  - Each of {A, B, C} to be column-stored, row-stored, or stored with general stride (like tensors)
- Bottom line: we've already solved a similar combinatoric problem

A closer look

- gemm
  - $C := C + AB$
- What do we want?
  - To allow A, B, or C to be stored as any supported datatype (storage datatype)
- Actually we want more than that
  - To allow the A*B to be performed in a precision different (potentially) than the storage precision of either A or B (computation precision)
  - Potentially same for domain (computation domain)

Combinatoric Analysis

- Each of the three operands may be stored as one of t storage datatypes
- Assuming two domains, the operation may be computed in one of t/2 precisions.
- Total number of possible cases to implement
  - In general: $N=(t/2)t^3 = t^4/2$
  - For BLIS (currently): $N=(4/2)4^3 = 128$
  - Notice that BLAS implements only $4/128$

Combinatoric Analysis

- ssss, sssd, ssds, ssdd, sscs, sscd, … zzzs, zzzd.
- But wait! We don't need to implement them all… do we?
    - Okay, which ones do we omit?
- We must implement all cases because we can only identify cases that are *currently* useful to *one* or more parties, not cases that *will never* be useful to *any* party.

Combinatoric Analysis

- What about the other gemm parameters?
  - Each of three operands can be stored according to one of three storage formats: $3 \uparrow 3$
  - A and B can take one of four conjugation/ transposition arguments: $2 \uparrow 4$
- Total:
  - $N = (4/2\ )\, 4 \uparrow 3 \cdot 3 \uparrow 3 \cdot 2 \uparrow 4 = 55{,}296$

Combinatoric Analysis

- What if we hypothetically add a precision?
  - Ex: half-precision real; half-precision complex
- Total number of datatype cases to implement
  - $N=(6/2)6^{3}=648$
- When combined with storage, conjugation/transposition parameters
  - $N=(6/2)6^{3}\cdot 3^{3}\cdot 2^{4}=279{,}936$

Combinatoric Analysis

- Don’t try that with auto code generation!

The Path Forward

- So…
  - 128 datatype cases (for gemm)
  - 55,296 total uses cases
- How will we tackle this with BLIS?

The Path ~~Forward~~ Behind Us

- So…
  - 128 datatype cases (for gemm)
  - 55,296 total uses cases
- How ~~will~~ did we tackle this with BLIS?
- Surprise! It’s already done
  - How much? All of it (for gemm)

Mixed domain+precision

- You must have been working at this non-stop for months!
  - 14 calendar days for mixed domain (June 1 – June 14)
  - 14 calendar days for mixed precision, and mixed domain+precision (June 15 – June 28)
  - That includes retrofitting testsuite to test all cases
  - And no, I'm not a laser-focused robot
    - I sleep and take weekends off
    - I go to PhD dissertation defenses
    - I help others in our group at UT
    - I help others on GitHub

Mixed domain+precision

- Surely this must have exploded BLIS source!
  - No.

Source code (framework)	Total lines	Total size (KB)
BLIS pre-mixed dt	148,646	4,699
BLIS post-mixed dt	153,071 (+4,425)	4,840 (+141)

Source code (testsuite)	Total lines	Total size (KB)
BLIS pre-mixed dt	22,816	678
BLIS post-mixed dt	23,928 (+1,112)	710 (+32)

Mixed domain+precision

- Okay, what about the object code footprint?
  - Not really:

BLIS library size (KB)	**Static library**	**Shared library**	**Statically-linked testsuite**
BLIS pre-mixed dt	3,138	2,285	1,631
BLIS post-mixed dt (disabled)	3,142 (+4)	2,285 (+0)	1,661 (+30)
BLIS post-mixed dt (enabled)	3,255 (+117)	2,389 (+104)	1,757 (+126)

Mixed domain: How did we do it?

Mixed domain case: C += A B	**Notes**
R += R R	Already implemented.
R += R C	Pair 1C: project B to real domain.
R += C R	Pair 1C: project A to real domain.
R += C C	Pack to 1r format and compute/accumulate in real domain.
C += R R	Project C to real domain and compute/accumulate in real domain. (Requires support for general stride storage.)
C += R C	Pair 2C: Treat B as $k \times 2n$ real matrix and pack accordingly; accumulate to C (by rows) via virtual µkernel.
C += C R	Pair 2C: Treat A as $2m \times k$ real matrix and pack accordingly; accumulate to C (by columns) via virtual µkernel.
C += C C	Already implemented.

Mixed precision: How did we do it?

Mixed precision case: C += A B | cp	Implementation notes
s += s s | s	Already implemented.
s += s d | s	Cast (demote) B to single-precision during packing.
s += d s | s	Cast (demote) A to single-precision during packing.
s += d d | s	Cast (demote) A, B to single-precision during packing.
d += s s | s	Use special update in macrokernel (or virtual μkernel) to accumulate result to C.
d += s d | s	Cast (demote) B to single during packing. Use special update in macrokernel (or virtual μkernel) to cast/accumulate result to C.
d += d s | s	Cast (demote) A to single during packing. Use special update in macrokernel (or virtual μkernel) to cast/accumulate result to C.
d += d d | s	Cast (demote) A, B to single during packing. Use special update in macrokernel (or virtual μkernel) to cast/accumulate result to C.

Mixed precision: How did we do it?

Mixed precision case: C += A B | cp	**Implementation notes**
s += s s | d	Cast (promote) A, B to double-precision during packing. Use special update in macrokernel (or virtual µkernel) to cast/accumulate result to C.
s += s d | d	Cast (promote) A to double-precision during packing. Use special update in macrokernel (or virtual µkernel) to cast/accumulate result to C.
s += d s | d	Cast (promote) B to double-precision during packing. Use special update in macrokernel (or virtual µkernel) to cast/accumulate result to C.
s += d d | d	Use special update in macrokernel (or virtual µkernel) to cast/accumulate result to C.
d += s s | d	Cast (promote) A and B to double-precision during packing.
d += s d | d	Cast (promote) A to double-precision during packing.
d += d s | d	Cast (promote) B to double-precision during packing.
d += d d | d	Already implemented.

Mixed domain: How did we do it?

- So what do we need? The ability to…
  - project complex matrices to real domain (in-place)
  - pack to 1r format
  - accumulate matrix products to C with general stride
  - “spoof” complex blocksizes for partitioning and then use real blocksizes in macrokernel
  - accumulate to C via virtual microkernels
  - nearly indispensable: encapsulation via objects

Mixed precision: How did we do it?

- So what do we need? The ability to…
  - Track at least three datatypes per object
    - storage, target, computation
  - Cast (promote or demote) a matrix from its storage datatype to the target datatype during packing
  - Cast (promote or demote) an intermediate matrix product from the computation datatype to the storage datatype of C during accumulation

Mixing domain+precision: How did we do it?

- Implementing full mixed datatype
  - Once you've implemented mixed domain and mixed precision separately, this is nearly free!
    - Domain and precision are mostly orthogonal

Performance

- Sorry, I didn’t have time.

Performance

- ~~Sorry, I didn't have time.~~
  - Kidding. Of course I have performance results!
- Poster: sequential performance
  - https://www.cs.utexas.edu/~field/retreat/2018/mdst.pdf
- Web-only bonus: multithreaded performance
  - https://www.cs.utexas.edu/~field/retreat/2018/mdmt.pdf

Performance

- Hardware
  - Intel Xeon E3-1271 v3 (Haswell) 3.6GHz (4 cores)
- Software
  - Ubuntu 16.04
  - GNU gcc 5.4.0
  - OpenBLAS 0.2.20 (latest stable release)
  - BLIS 0.4.1-15/c03728f1 + mixed-dt extensions

Performance

- Implementations tested
  - BLIS: implemented within `bli_gemm()`
    - Mixed domain/precision logic is hidden
  - OpenBLAS: implemented within a “dumb wrapper” around `[sdcz]gemm_()`
    - Mixed domain/precision logic is exposed
- Labeling example: **zcds**gemm
  - Interpretation: **cabx**
    - **C** is double complex (z)
    - **A** is single complex (c)
    - **B** is double real (d)
    - computation is e**x**ecuted in single-precision (s)

Performance

- Results
  - x-axis: problem size: m = n = k
    - Sequential: 40 to 2000 in increments of 40
    - Multithreaded: 80 to 4000 in increments of 80
  - y-axis: GFLOPS/core
    - Top of graph is machine (theoretical) peak
  - Each data point is best of three trials

Performance

- General characterization
  - mixed-datatype BLIS performs typically 75-95% of [sdcz]gemm
  - mixed-datatype BLIS almost universally outperforms the “dumb wrapper” alternative
  - *and* BLIS requires less workspace
  - *and* BLIS still provides features and options not present in the BLAS
    - row/column strides; extra support for complex domain, object API, more multithreading options, comprehensive testsuite, lots of documentation, etc.

What's next?

- Other operations?
  - hemm, symm, herk, syrk, trmm, etc.
- Other precisions?
  - bfloat16
  - quad-precision
  - double double
- Start from scratch?
  - C++

Thank you!