The BLIS Framework

Field G. Van Zee
Science of High-Performance Computing
The University of Texas at Austin

6th EasyBuild User Meeting
January 27, 2021

What is BLIS?

- BLAS-like Library Instantiation Software
- BLIS is a framework for
  - Quickly instantiating high-performance BLAS-like libraries
- "Why 'BLAS-like'?"…
  - For now, just assume BLAS-like = BLAS

What is BLAS?

- Basic Linear Algebra Subprograms
  - Level 1: vector-vector [Lawson et al. 1979]
  - Level 2: matrix-vector [Dongarra et al. 1988]
  - Level 3: matrix-matrix [Dongarra et al. 1990]
- Why are BLAS important?

Why are BLAS important?

- BLAS constitute the “bottom of the food chain” for most dense linear algebra applications, as well as other libraries
  - LAPACK, `libflame`, MATLAB, PETSc, etc.
- The idea is simple:
  - if the BLAS interface is “standardized”, and
  - if an optimized implementation exists for your architecture
  - then higher-level applications can portably access high performance

More about the BLAS

- level-1 operations
  - i?amax: find index of element with largest absolute value
  - ?asum: compute absolute sum
  - ?axpy: scale vector and accumulate
  - ?copy: copy vector
  - ?dot: compute dot (inner) product
  - ?nrm2: compute vector 2-norm
  - ?scal: apply a scalar to a vector
  - ?swap: swap two vectors

More about the BLAS

- level-2 operations
  - ?gemv: general matrix-vector multiply
  - ?ger: general rank-1 update
  - ?hemv: Hermitian matrix-vector multiply
  - ?her: Hermitian rank-1 update
  - ?her2: Hermitian rank-2 update
  - ?symv: symmetric matrix-vector multiply
  - ?syr: symmetric rank-1 update
  - ?syr2: symmetric rank-2 update
  - ?trmv: triangular matrix-vector multiply
  - ?trsv: triangular solve (with one right-hand side)

More about the BLAS

- level-3 operations
  - **?gemm: general matrix multiply**
  - ?hemm: Hermitian matrix multiply
  - ?herk : Hermitian rank-k update
  - ?her2k : Hermitian rank-2k update
  - ?symm: symmetric matrix multiply
  - ?syrk: symmetric rank-k update
  - ?syr2k: symmetric rank-2k update
  - ?trmm: triangular matrix multiply
  - ?trsm: triangular solve (with multiple right-hand sides)

More about the BLAS

- Plenty of BLAS implementations available
  - Vendor
    - MKL (Intel), cuBLAS (NVIDIA), ARMPL (ARM), ESSL (IBM), Accelerate (Apple), etc.
  - Open source
    - netlib, OpenBLAS, Eigen, BLASFEO, libxsmm, ATLAS, etc.
- So why do we need BLIS?

Why do we need BLIS?

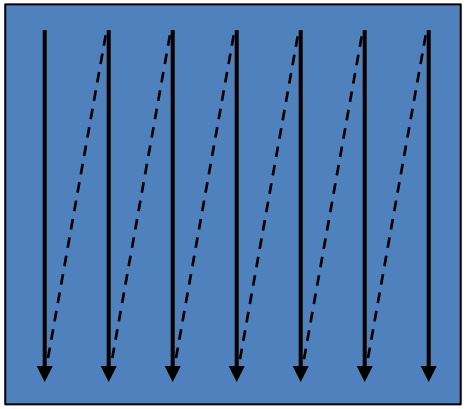
- Actually, there are two questions
  - Why do we **need** BLIS?
  - Why should we **want** BLIS?
    - (Even if we don't need it)
- Let's look at the first question

Why do we **need** BLIS?

- The BLAS interface is limiting for some applications
  - Not surprising – it was finalized 30 years ago!
- How exactly is the BLAS interface limiting?

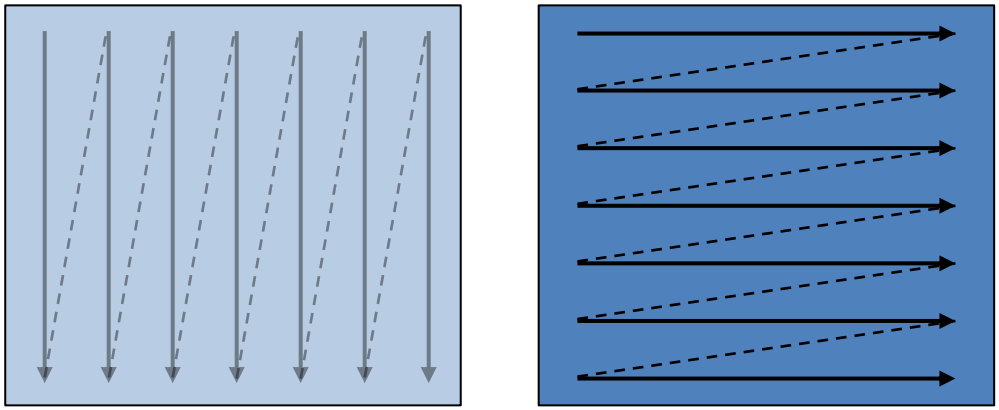
Limitations of BLAS interface

- Interface only allows column-major storage



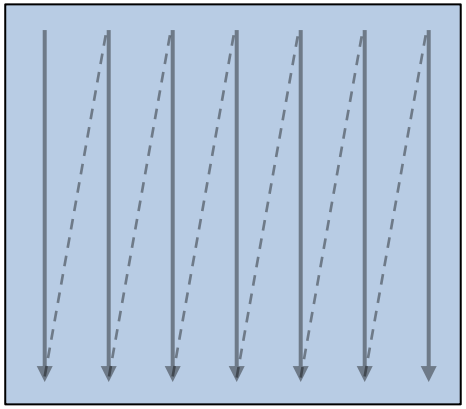
Limitations of BLAS interface

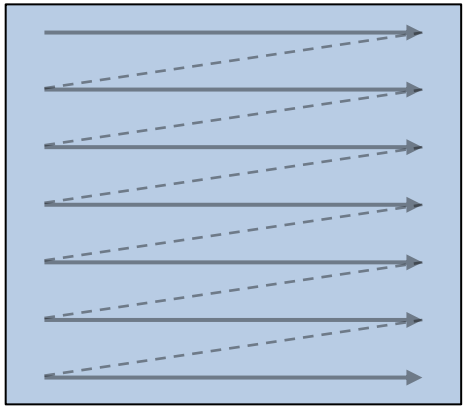
- Interface only allows column-major storage
  - We also want row-major storage

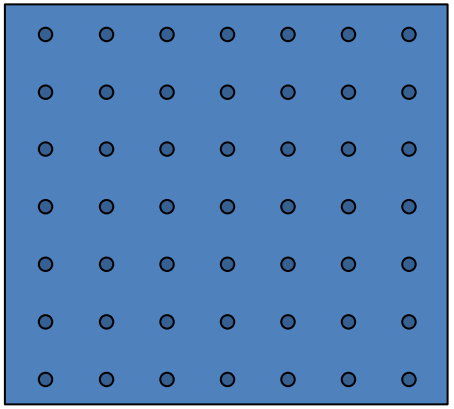


Limitations of BLAS interface

- Interface only allows column-major storage
  - We also want row-major storage and general stride storage



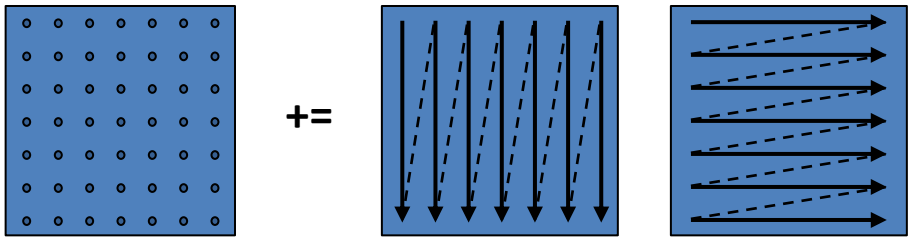




Limitations of BLAS interface

- Interface only allows column-major storage
  - We also want row-major storage and general stride storage
  - Further yet, we want to support computation on operands of ***mixed*** storage formats. Example:
    - $C := C + AB$

      where A is column-stored, B is row-stored, and C has general stride.

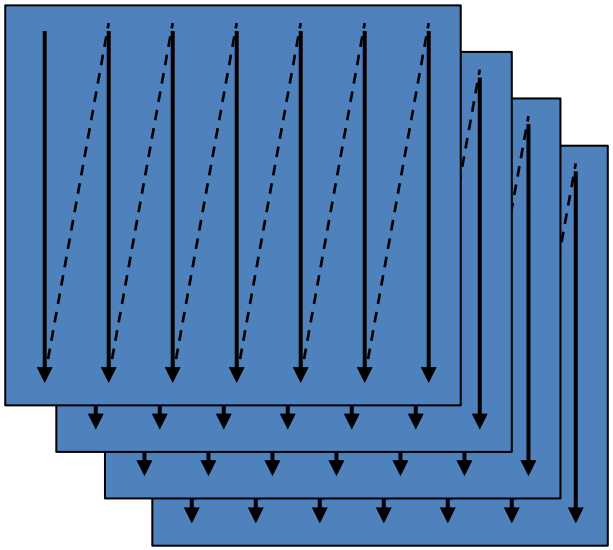


Limitations of BLAS interface

- Why do we need general stride storage?

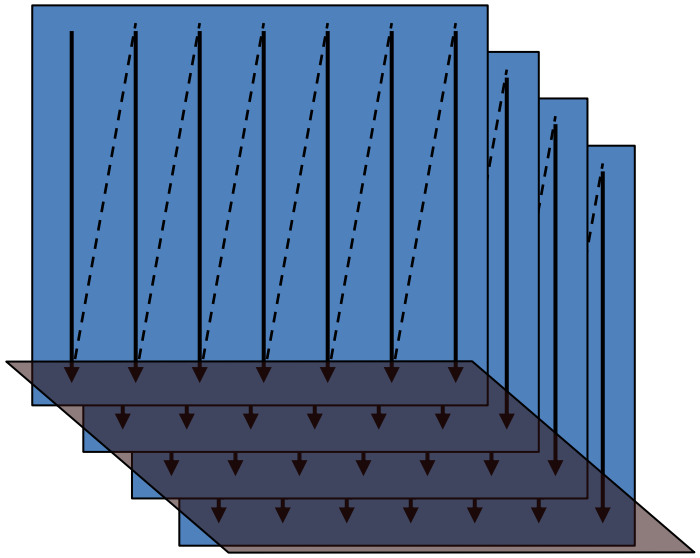
Limitations of BLAS interface

- Why do we need general stride storage?
- Example: three-dimensional tensor



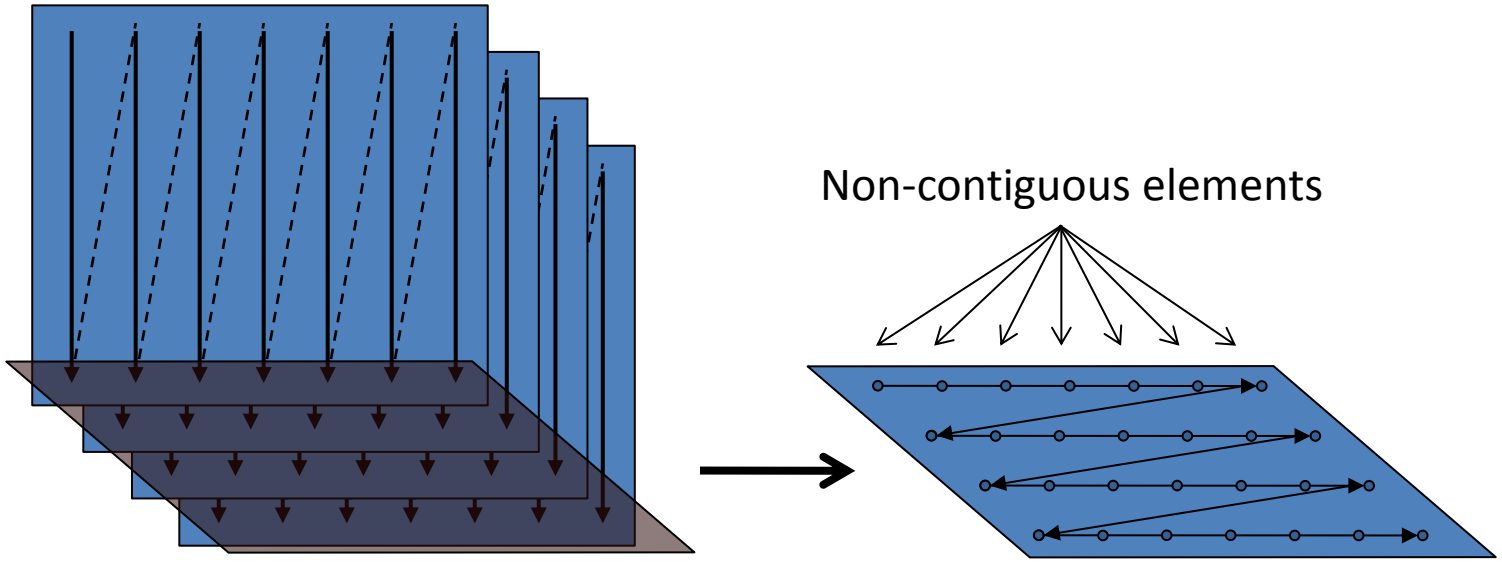
Limitations of BLAS interface

- Why do we need general stride storage?
- Example: three-dimensional tensor
  - How do we take an arbitrary slice?



Limitations of BLAS interface

- Why do we need general stride storage?
- Example: three-dimensional tensor
  - How do we take an arbitrary slice?
    - It may be non-contiguous in both dimensions



Limitations of BLAS interface

- Incomplete support for complex operations
  - Vector conjugation, matrix "conjugation without transposition" options are missing from BLAS

  Examples:

  - $y := y + \alpha\bar{x}$ axpy
  - $y := y + A\bar{x}$ gemv
  - $C := C + \bar{A}B$ gemv, gemm
  - $C := C + \bar{A}A^T$ her, herk
  - $B := \bar{L}B$ trmv, trmm
  - $B := \bar{L}^{-1}B$ trsv, trsm

Limitations of BLAS interface

- BLAS API is opaque
  - No uniform way to access lower-level kernels
- Why would one want access to these kernels? Some possibilities:
  - Optimize operations in higher-level libraries or applications
  - Implement new BLAS-like operations (without “reinventing the wheel”)
  - Conduct research or measurements on the kernels themselves

Limitations of BLAS interface

- Operation support has not changed in three decades
  - BLAST Technical Forum attempted to ratify some improvements
  - Revisions largely ignored by implementors. Why?
    - Best guess: No official reference implementation

Why do we **need** BLIS?

- So why does any of this mean we need BLIS?
  - The BLAS API is static and cannot be improved
  - We can't gain access to a better interface by building a better BLAS – we need something else altogether
  - This was one of the primary motivations for developing BLIS

Why do we **need** BLIS?

- BLIS addresses the BLAS' interface issues
  - Independent row and column stride properties allow flexible matrix storage
  - Any input operand can be conjugated
  - Experts can directly call lower-level packing, computation kernels
  - Operation support can grow over time, as needed
    - Hence why BLIS is "BLAS-like"

Why do we **need** BLIS?

- BLIS addresses the BLAS' interface issues
  - Independent row and column stride properties allow flexible matrix storage
  - Any input operand can be conjugated
  - Experts can directly call lower-level packing, computation kernels
  - Operation support can grow over time, as needed
    - Hence why BLIS is "BLAS-like"
- This is why BLIS needs to exist
  - *i.e., These features are largely absent from other BLAS implementations*

Why should we **want** BLIS?

- What if I don't need any of these features unique to BLIS?
  - You still might want to use BLIS!

Why should we **want** BLIS?

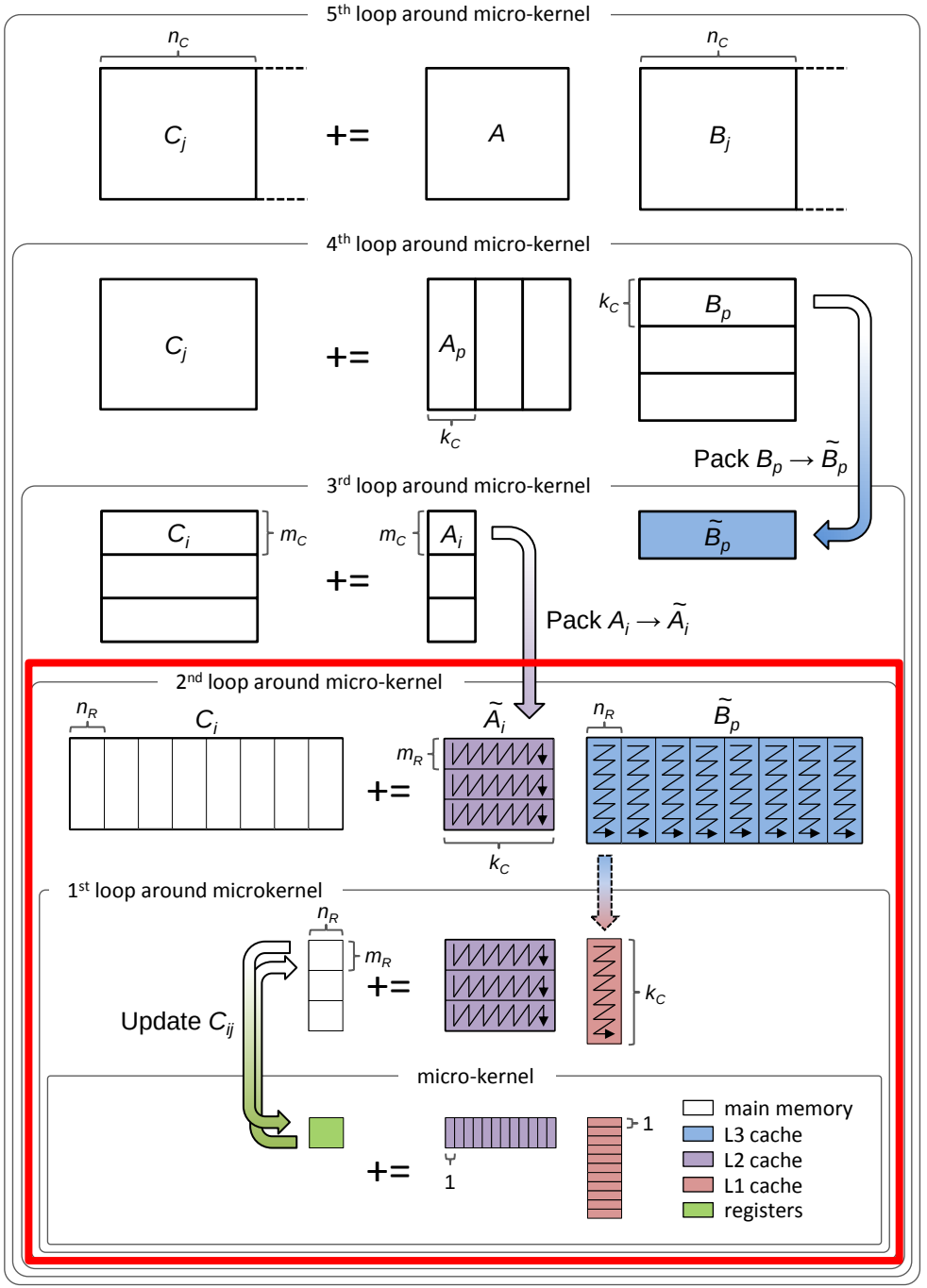
- What if I don't need any of these features unique to BLIS?
  - You still might want to use BLIS!
- If you're an end-user
  - Improved APIs
  - You can still use BLAS (or CBLAS) compatibility layer
- If you're a developer
  - BLIS makes it easier to implement high-performance BLAS libraries on new hardware

Why should we **want** BLIS?

- So how does BLIS make it easier to implement high-performance BLAS?
  - Let's first look at general matrix-matrix multiplication (gemm) as implemented by Kazushige Goto in GotoBLAS
    - [Goto and van de Geijn 2008]
  - Note: This same approach was inherited into OpenBLAS

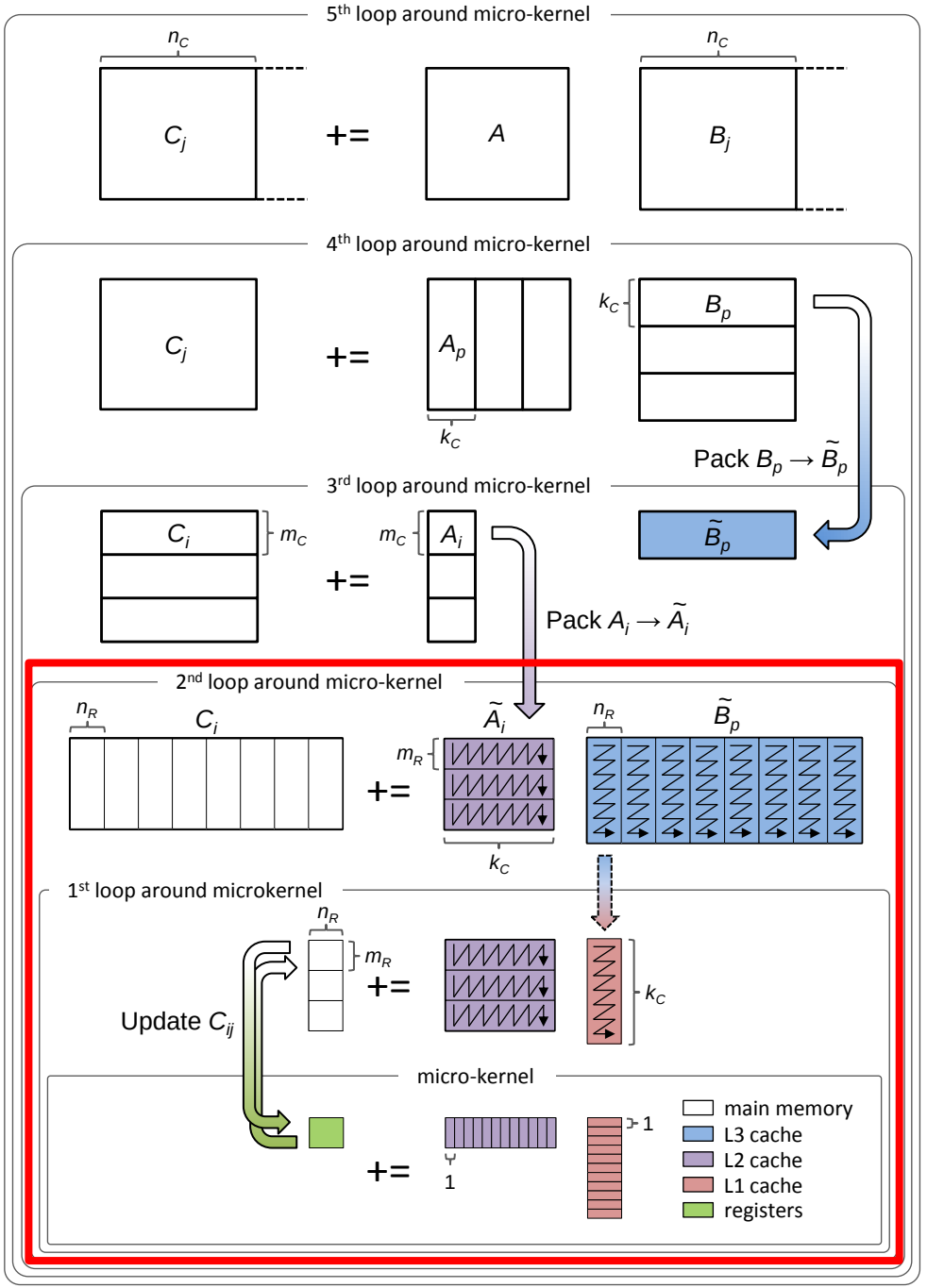
Goto Algorithm

- Built atop Goto's "inner kernel"
  - Three loops around a tiny outer product
  - Written entirely in assembly language



Goto Algorithm

- Built atop Goto's "inner kernel"
  - Three loops around a tiny outer product
  - Written entirely in assembly language
- Drawbacks
  - Cannot always recycle gemm inner kernel for other level-3 operations
  - Difficult to read: three loops in ~5000 lines!
  - Edge cases handled explicitly
  - Can't parallelize within assembly region

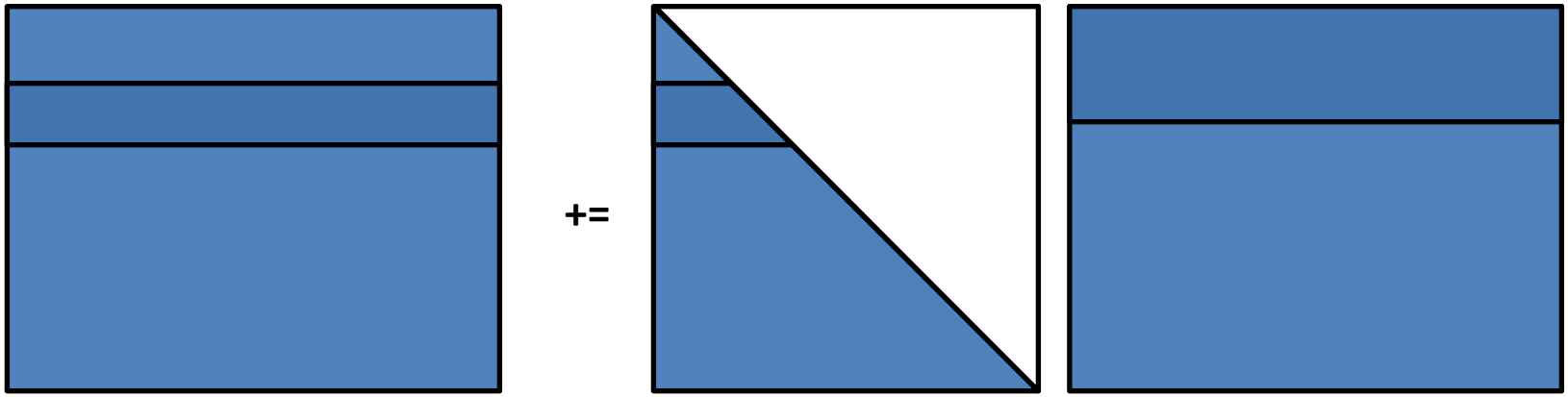


Goto Algorithm: Recycling Kernels

- Why can't we always recycle the Goto inner kernel?
  - It has to do with differences between the level-3 operations

Goto Algorithm: Recycling Kernels

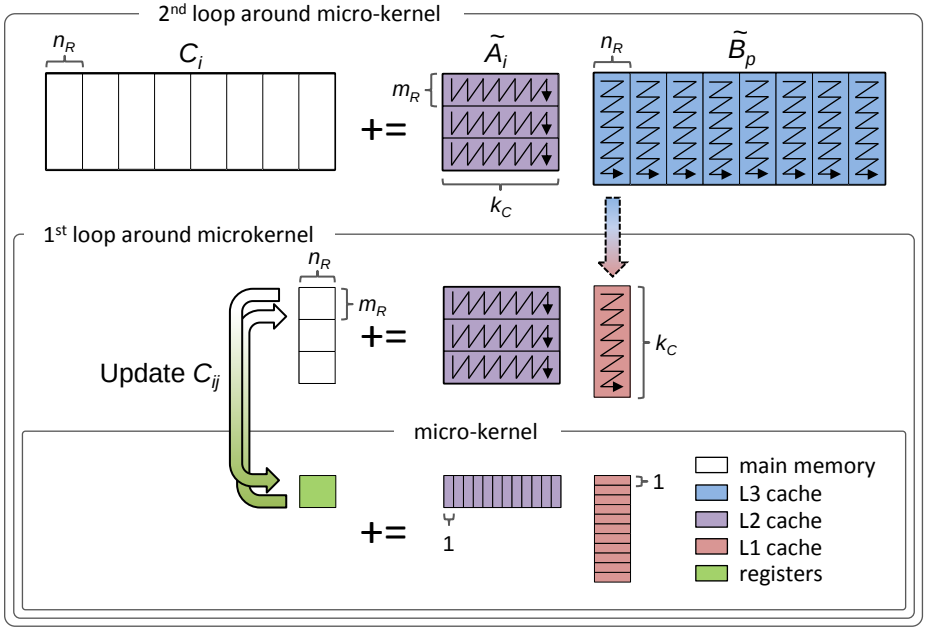
- Example: Triangular matrix multiplication (trmm)



- Needs at least two special inner kernels
  - Variants for lower- and upper-stored matrix A move in opposite directions
  - Inner-most loop bound varies as a function middle loop's current iteration (index) for blocks that intersect diagonal

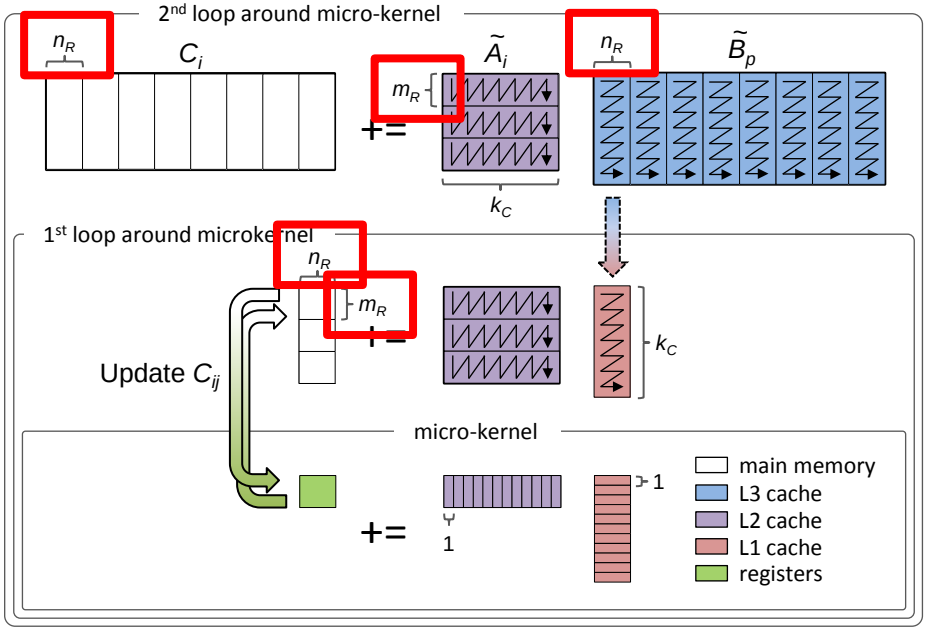
Goto Algorithm: Assembly Footprint

- Three loops encoded in assembly language
  - With lots of unrolling
  - And edge case handling
  - It gets complicated in a hurry!



Goto Algorithm: Edge Cases

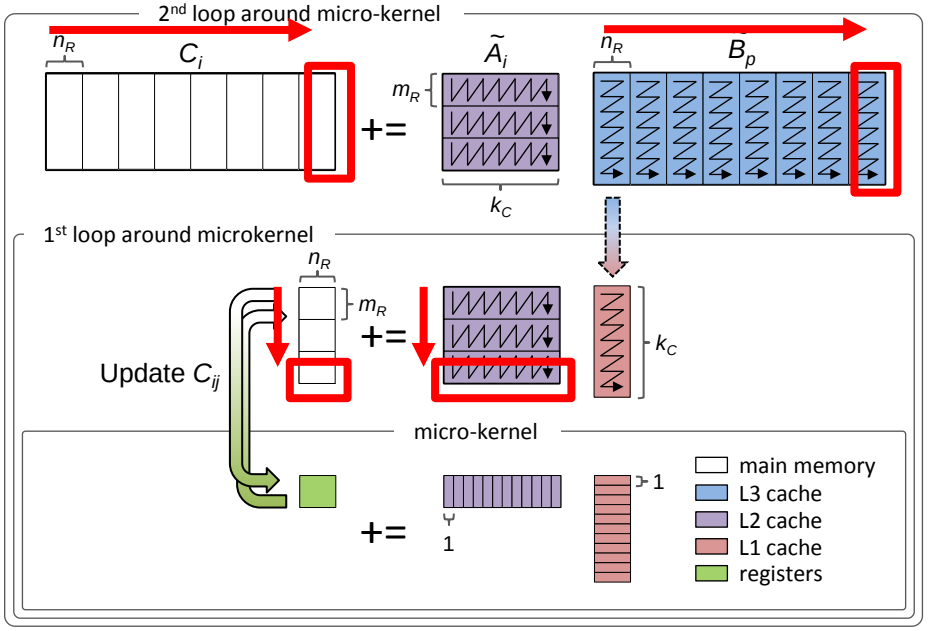
- What are kernel edge cases?
  - Inner kernels have a “fundamental size” (MR x NR) based on register allocation and data rearrangement (packing)
  - Examples
    - MR = 6
    - NR = 8



Goto Algorithm: Edge Cases

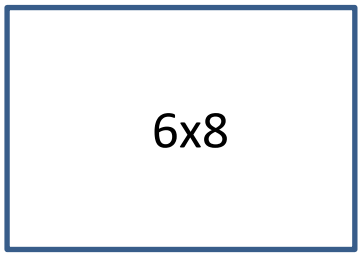
- What are kernel edge cases?
  - For each of the outer two loops, the inner kernel must handle “leftover” matrix blocks

Possible edge cases highlighted in red



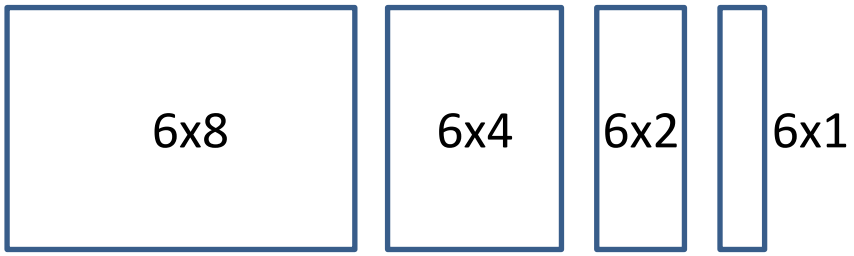
Goto Algorithm: Edge Cases

- What does this mean for the assembly kernel?
  - Lots of extra logic after each loop to handle specific edge case sizes



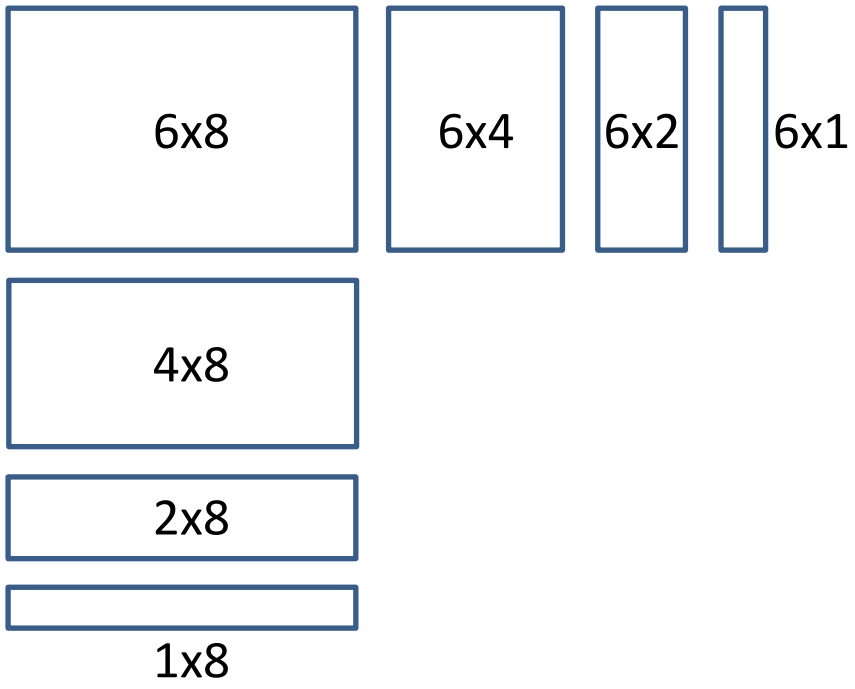
Goto Algorithm: Edge Cases

- What does this mean for the assembly kernel?
  - Lots of extra logic after each loop to handle specific edge case sizes



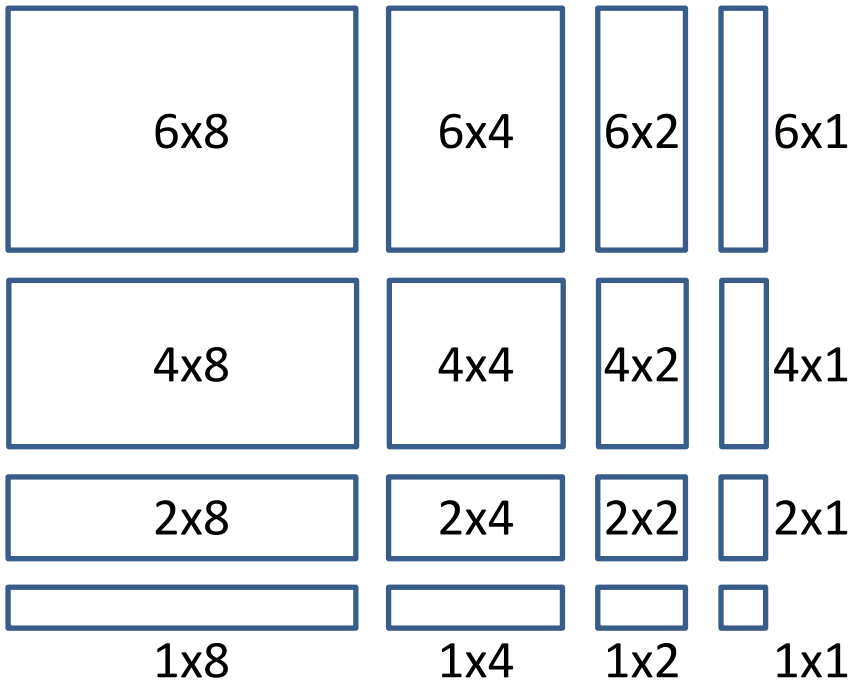
Goto Algorithm: Edge Cases

- What does this mean for the assembly kernel?
  - Lots of extra logic after each loop to handle specific edge case sizes



Goto Algorithm: Edge Cases

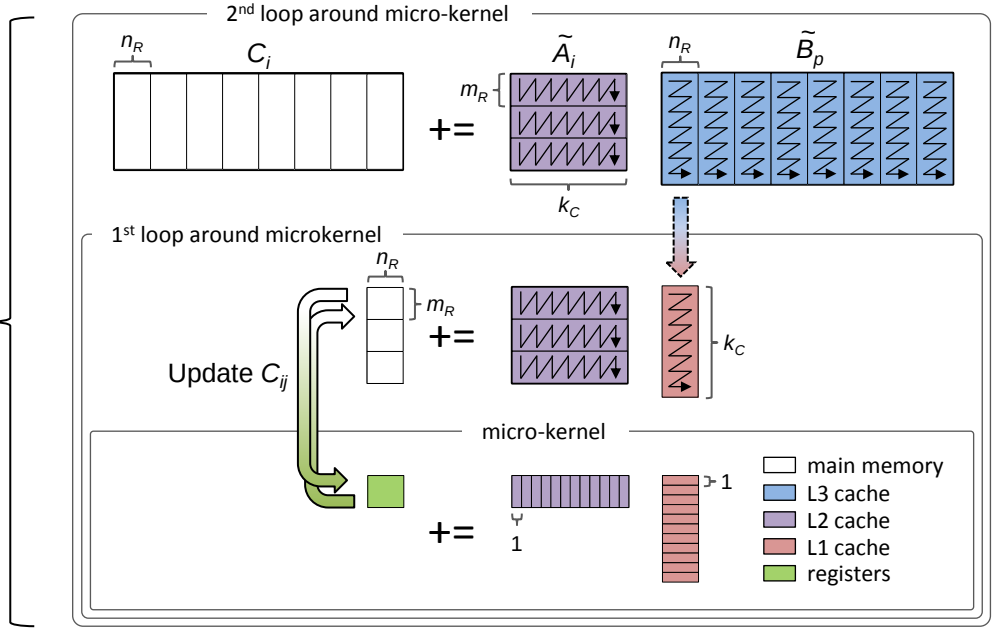
- What does this mean for the assembly kernel?
  - Lots of extra logic after each loop to handle specific edge case sizes



Goto Algorithm: Parallelization

- Inner kernel is an indivisible unit
  - Multithreaded parallelism cannot be easily encoded within either of the outer two loops
  - Parallelism must instead be obtained at a higher level

Can't easily parallelize this region

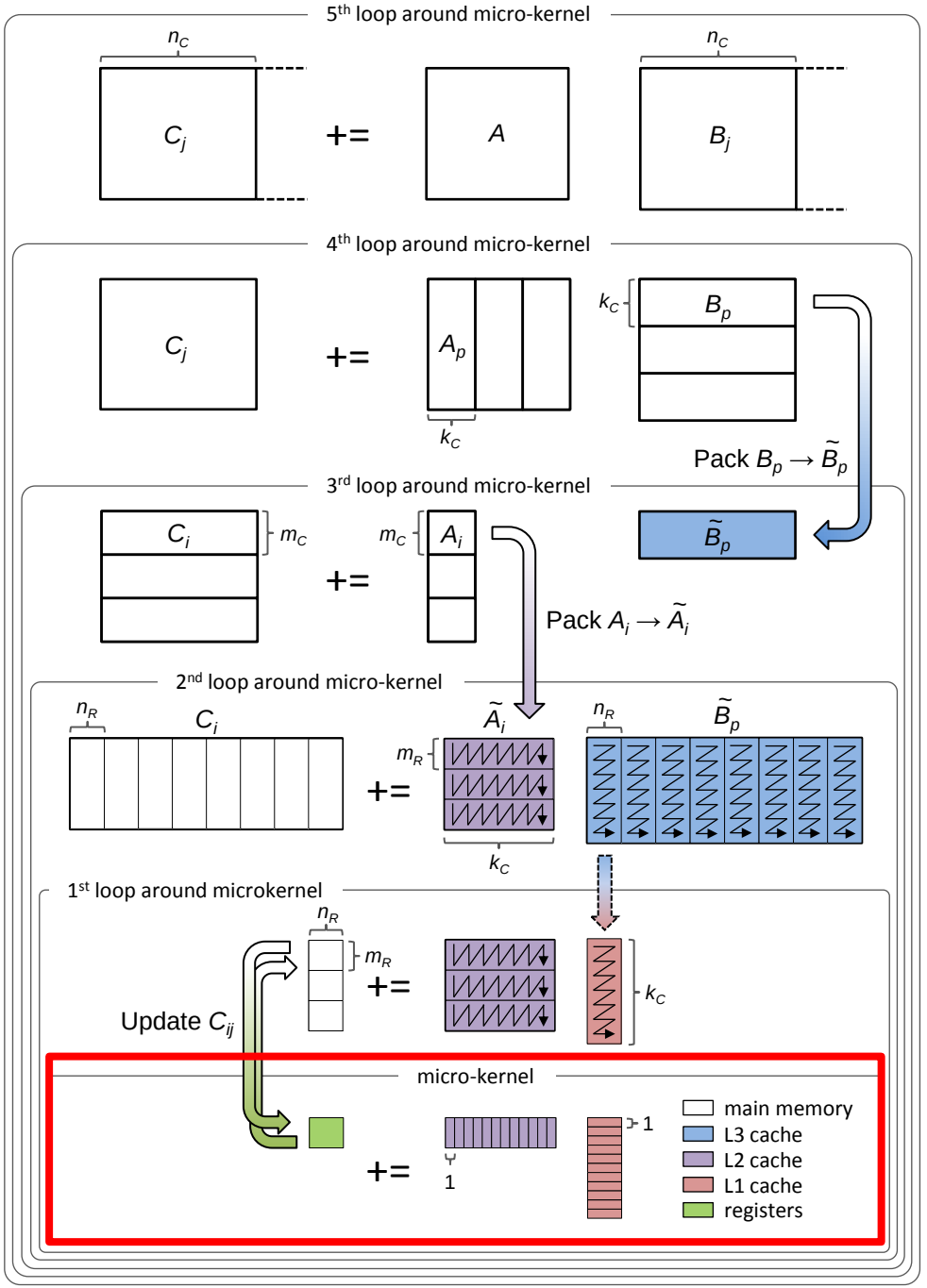


Why should we **want** BLIS?

- So how does BLIS make it easier to implement high-performance BLAS?
  - Now that we've looked at the software architecture for level-3 operations in GotoBLAS, let's look at the same for BLIS
    - [Van Zee and van de Geijn 2015]

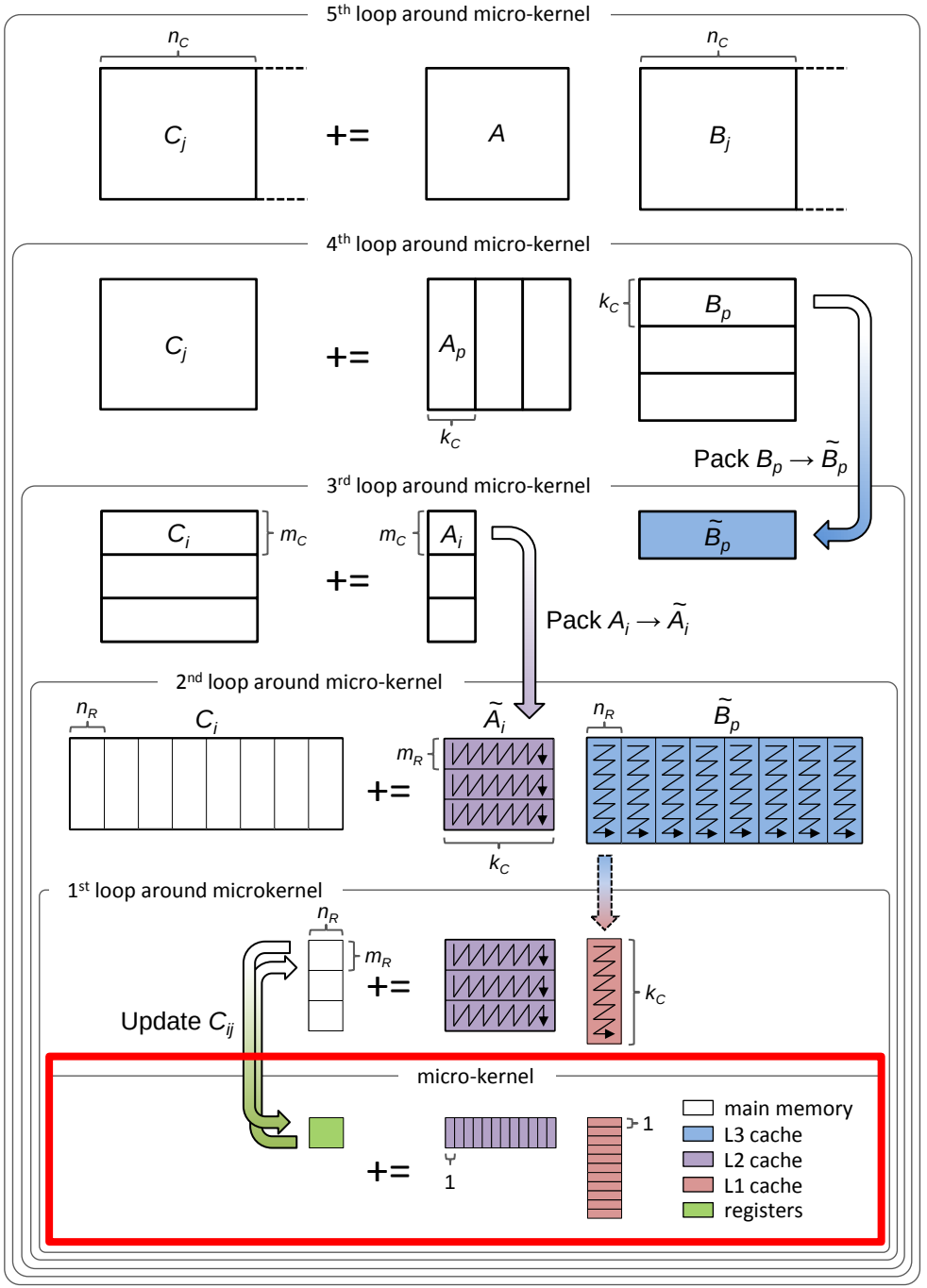
BLIS Algorithm

- Isolates assembly region to a smaller “micro-kernel”
  - One loop around a tiny outer product



BLIS Algorithm

- Isolates assembly region to a smaller “micro-kernel”
  - One loop around a tiny outer product
- Confers several benefits over Goto’s inner kernel
  - Micro-kernel is more easily recycled between level-3 inner kernels
  - Requires fewer lines of assembly
  - Allows edge cases to be handled portably
  - Exposes more opportunities for parallelism

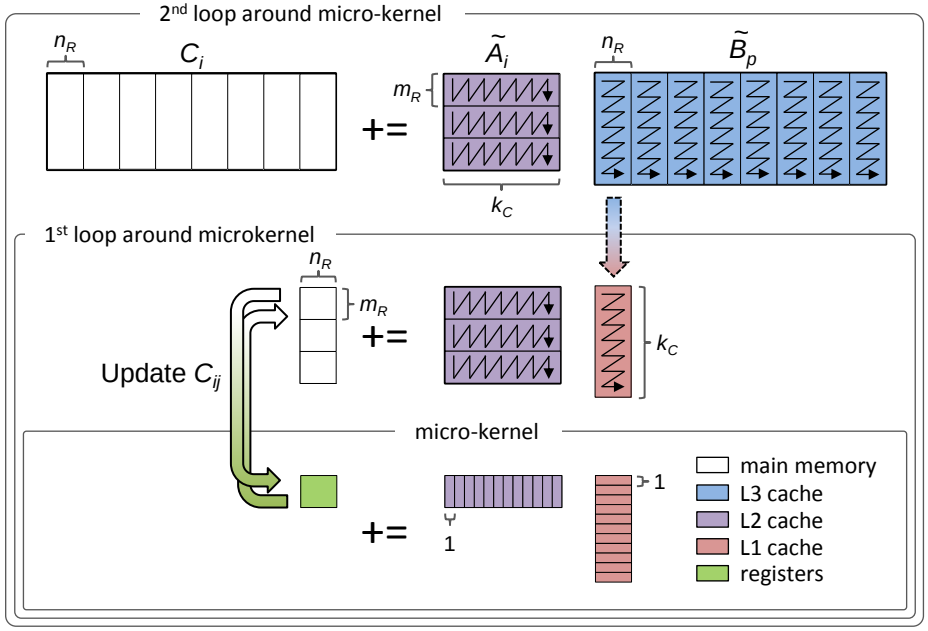


BLIS Algorithm: Recycling Kernels

- BLIS exposes the outer two loops of the inner kernel
  - These two outer loops were previously buried in assembly
  - Key observation: Virtually *all* of the differences between level-3 operations' inner kernels reside in these two loops
  - These loops ("macro-kernels") in BLIS are written in C99 for each of the level-3 operations

macro-kernel +
call to micro-kernel

micro-kernel

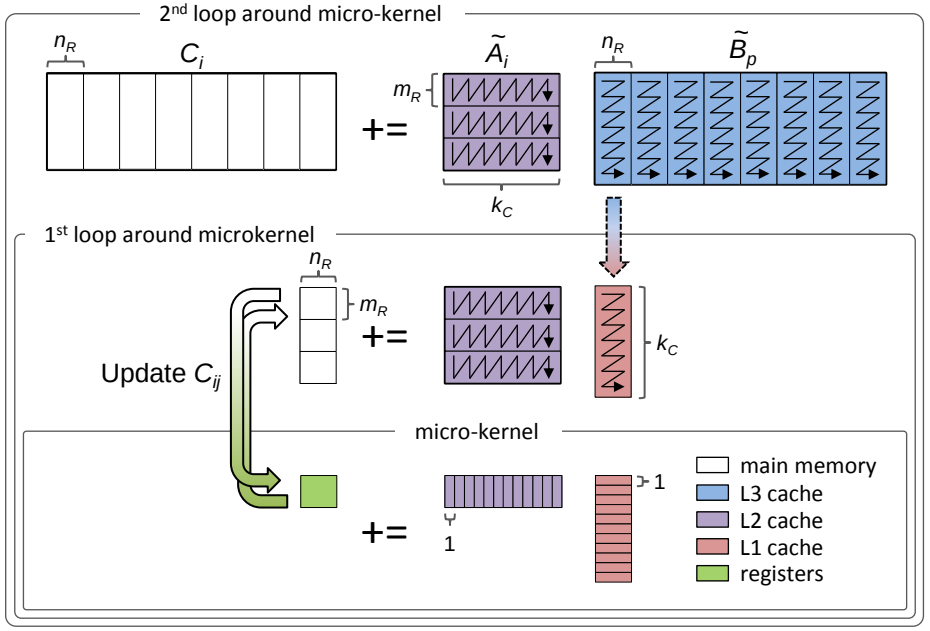


BLIS Algorithm: Assembly Footprint

- BLIS exposes the outer two loops of the inner kernel
  - Inner-most loop of macro-kernel simply calls micro-kernel
  - So the micro-kernel now consists of only one loop and no edge cases for MR, NR
  - This shrinks its size down from ≈5000 lines to ≈2000 lines

≈ 200 lines of C99

≈ 2000 lines of assembly



BLIS Algorithm: Edge Cases

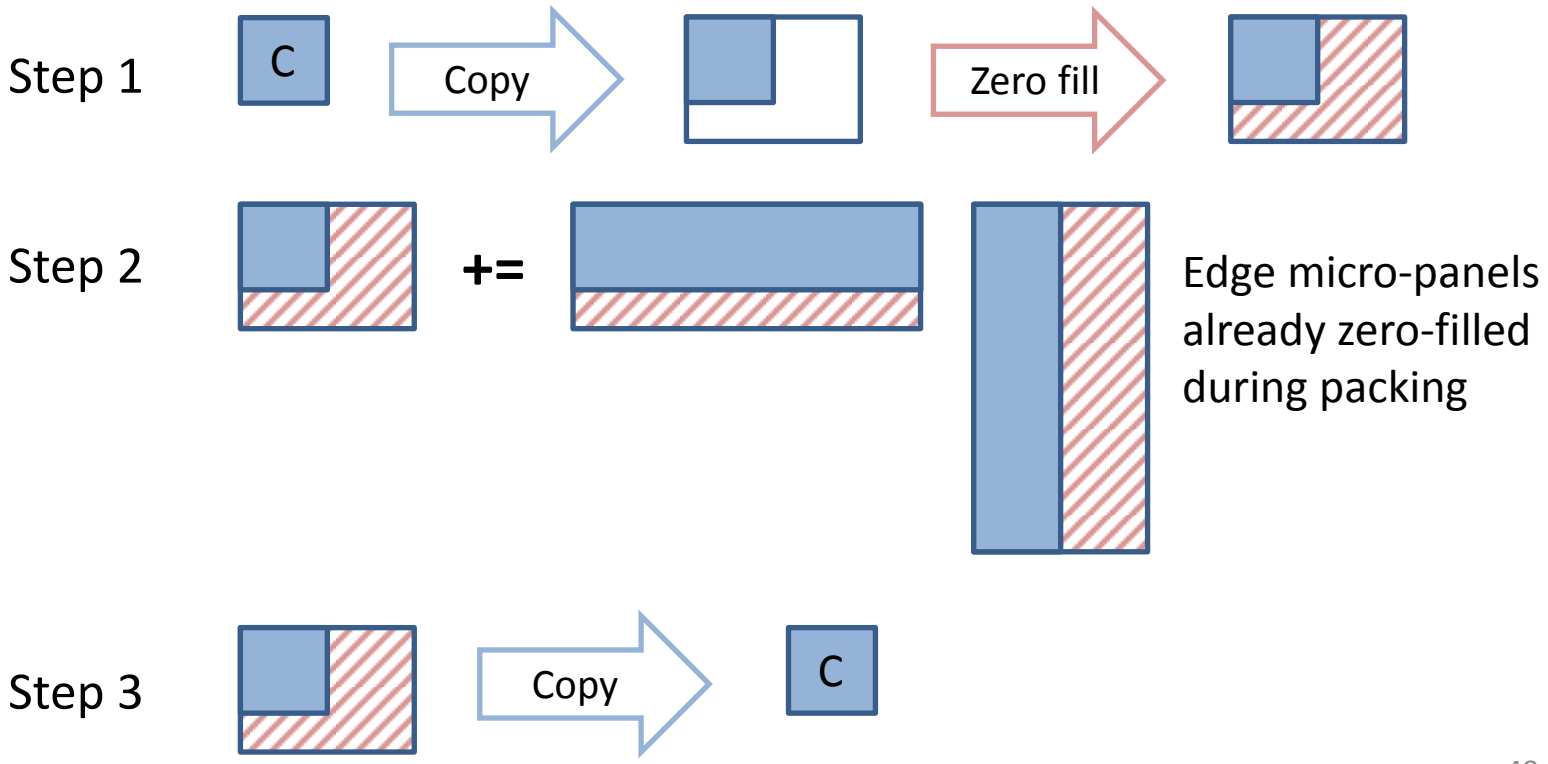
- Wait, if edge cases aren't handled in the microkernel, then how are they handled?

BLIS Algorithm: Edge Cases

- BLIS handles edge cases differently than the Goto approach
  - BLIS requires that the kernel developer implement only the (full-sized) micro-kernel

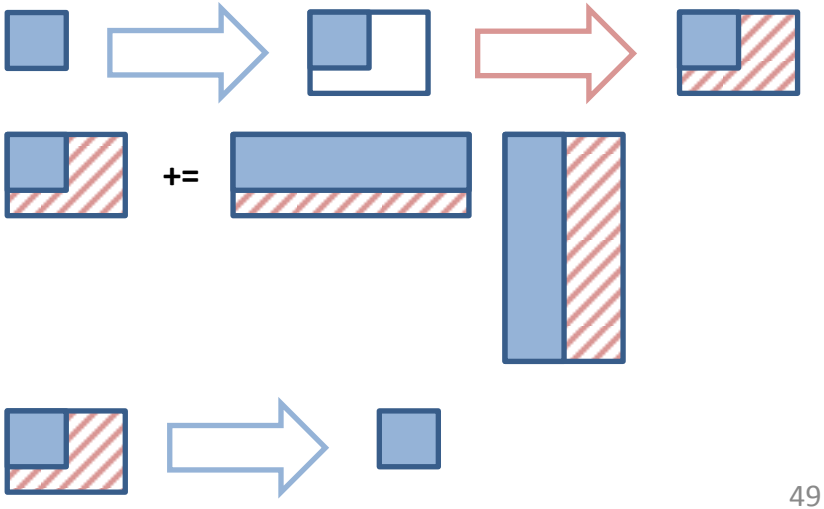
BLIS Algorithm: Edge Cases

- When an edge case is encountered in a level-3 operation
  - BLIS copies matrix C to temporary memory, zero-fills edges, computes on temporary, then copies back to C



BLIS Algorithm: Edge Cases

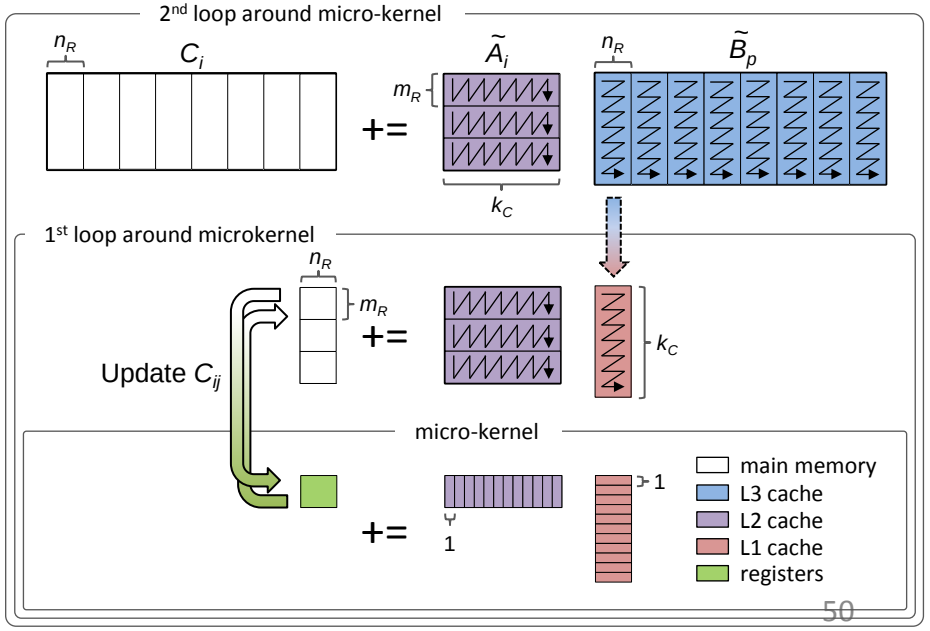
- This does come with a small performance penalty
  - First, the copies and zero-filling are not free
  - Second, the micro-kernel must perform extra computations with zeros, and does not get any credit for doing so
  - Manifests as a “saw tooth” pattern in performance graphs
- So why do we do this?
  - We think this trade-off between performance and productivity is worth it



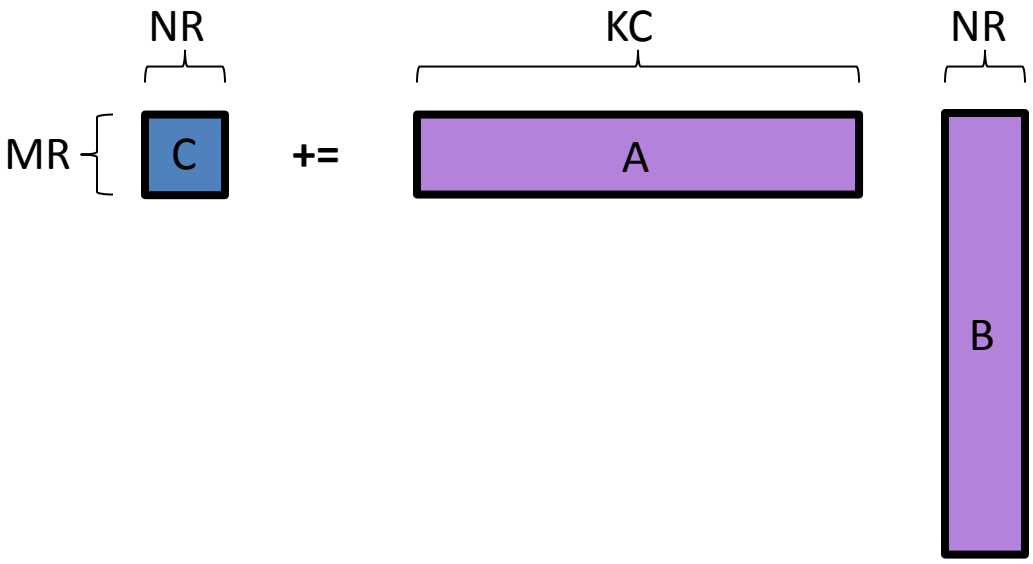
BLIS Algorithm: Parallelization

- BLIS macro-/micro-kernel design exposes additional opportunities for parallelism
  - Previously, Goto inner kernel was smallest unit of computation
  - Parallelism was typically extracted at higher (coarser grain) levels
  - Now we can parallelize at lower (finer grain) levels
    - Tends to produce smoother results and better load balancing
    - [Smith et al. 2014]

We can now easily parallelize these loops

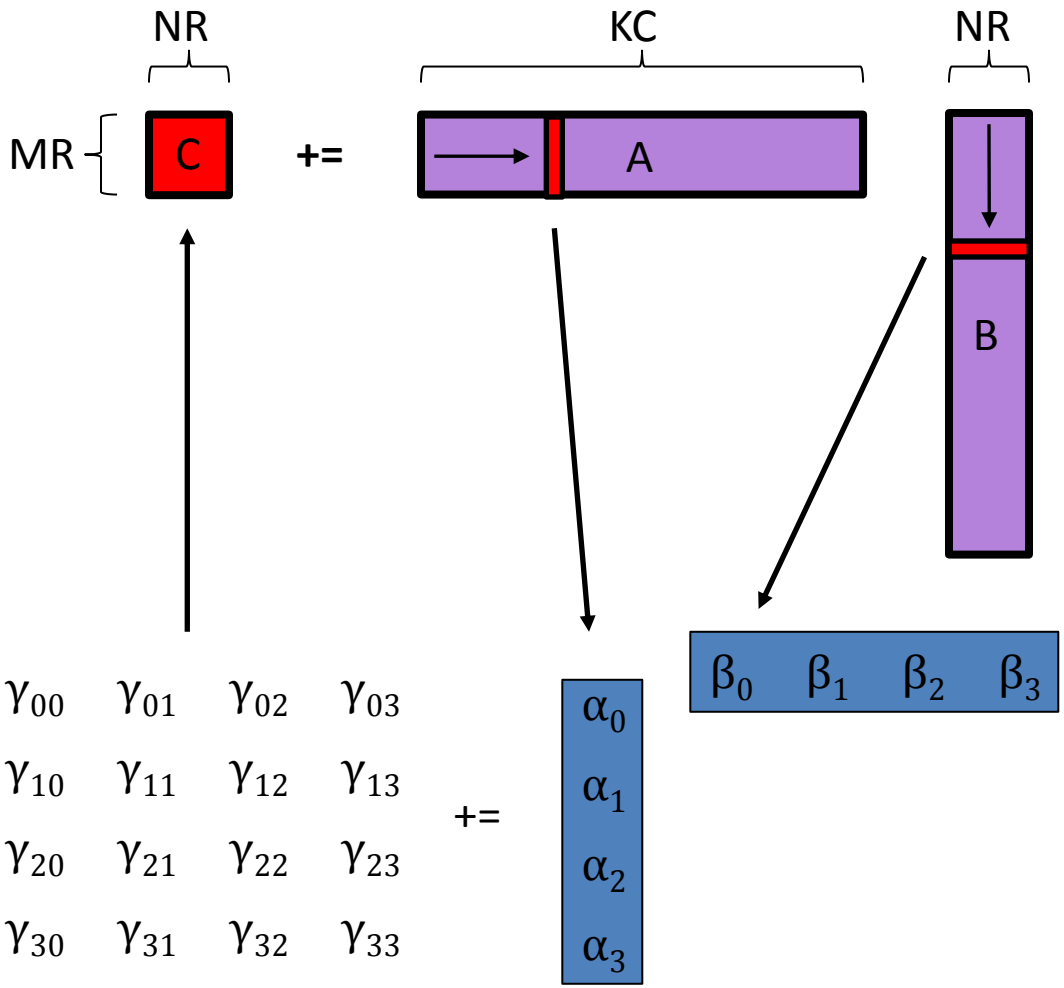


BLIS Algorithm: Micro-kernel



How is the micro-kernel typically implemented?

BLIS Algorithm: Micro-kernel

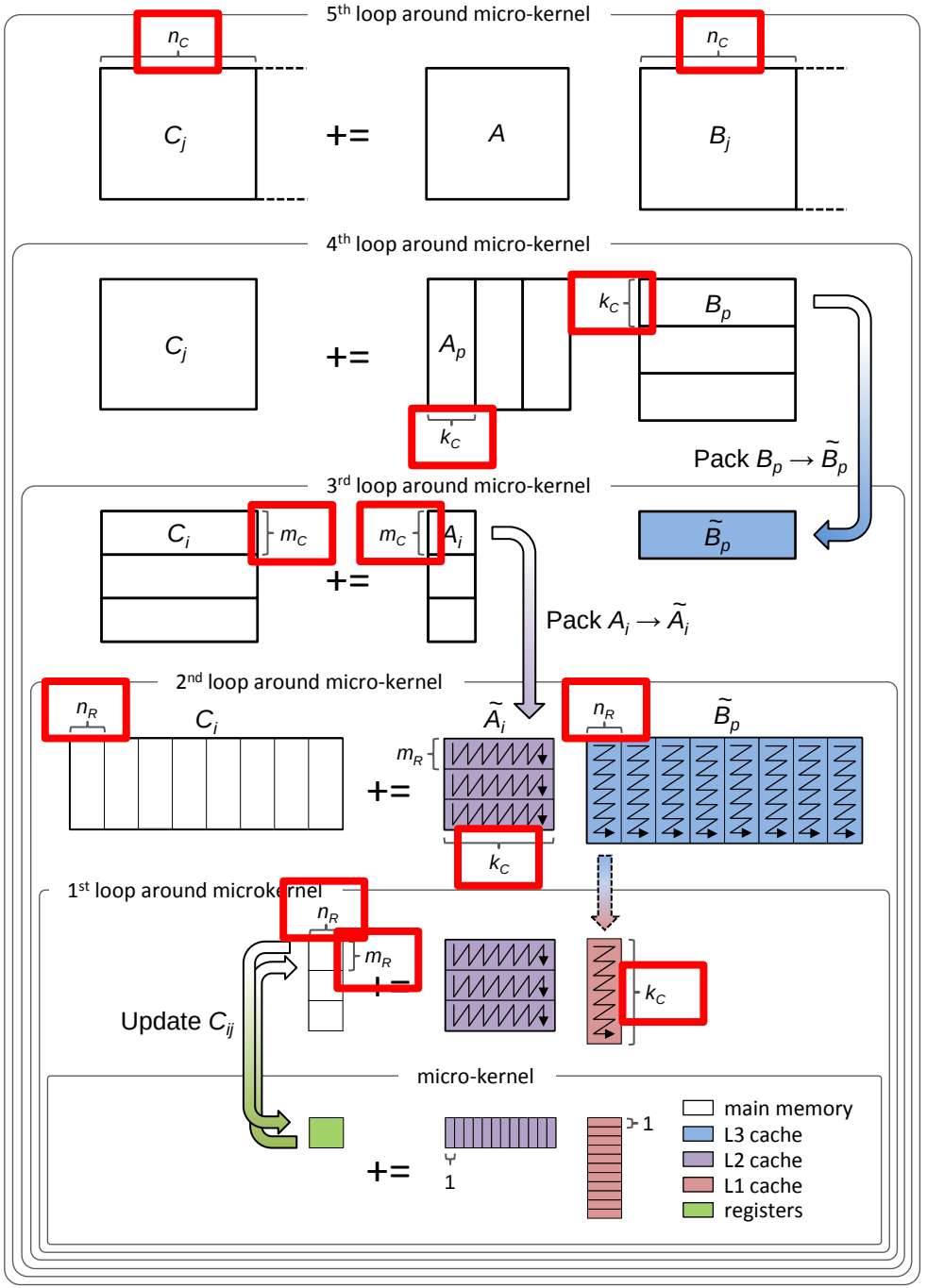


Typical micro-kernel loop iteration

- Load column of packed A
- Load row of packed B
- Compute outer product
- Update C (kept in registers)

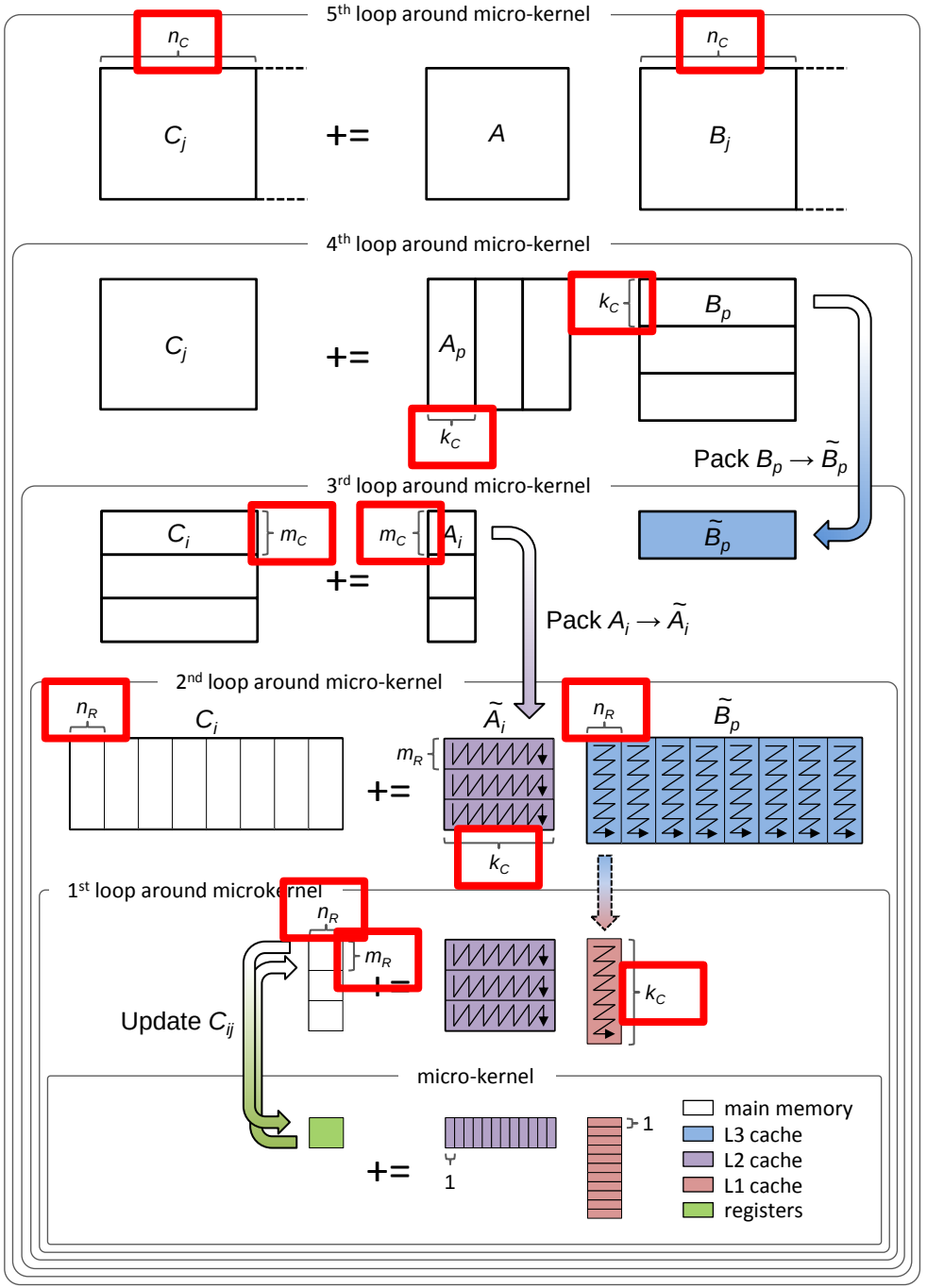
BLIS Blocksizes

- BTW: How are cache and register blocksizes chosen?



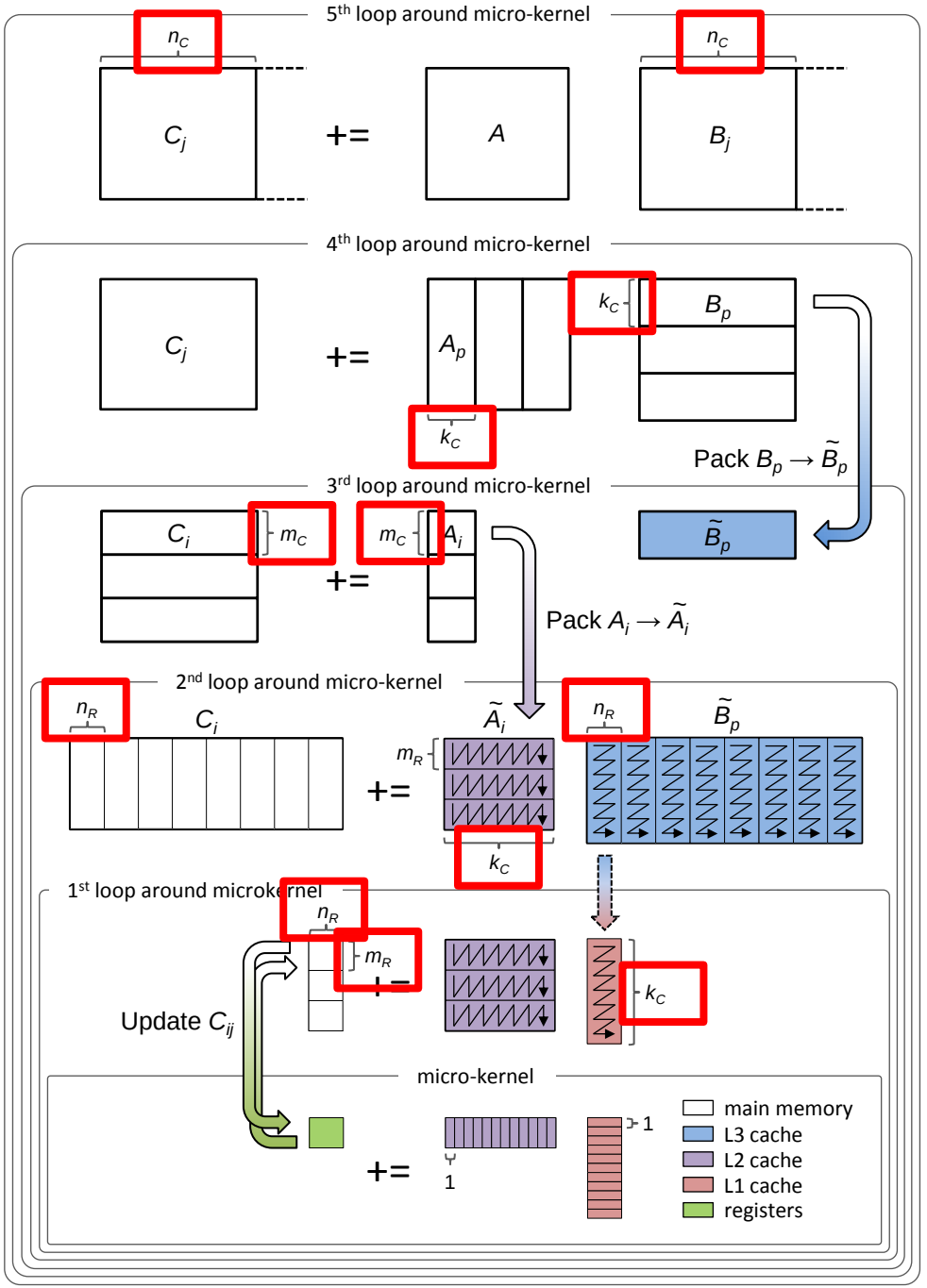
BLIS Blocksizes

- BTW: How are cache and register blocksizes chosen?
  - Empirical search?
    - [Whaley 1998]



BLIS Blocksizes

- BTW: How are cache and register blocksizes chosen?
  - ~~– Empirical search?~~
    - [Whaley 1998]
  - – Analytical model is sufficient
    - [Low et al. 2016]



BLIS Algorithm

- What about packing?
  - BLIS unifies packing for three different matrix structures into one interface
    - General matrices
    - Symmetric/Hermitian matrices
    - Triangular matrices
  - Highly-parameterized and reusable for variety of parameter cases
    - e.g. side/uplo/trans parameters, matrix storage formats
  - How?
    - Short answer: Separate row, column strides go a long way!
- Let's review and summarize

Benefits of BLIS

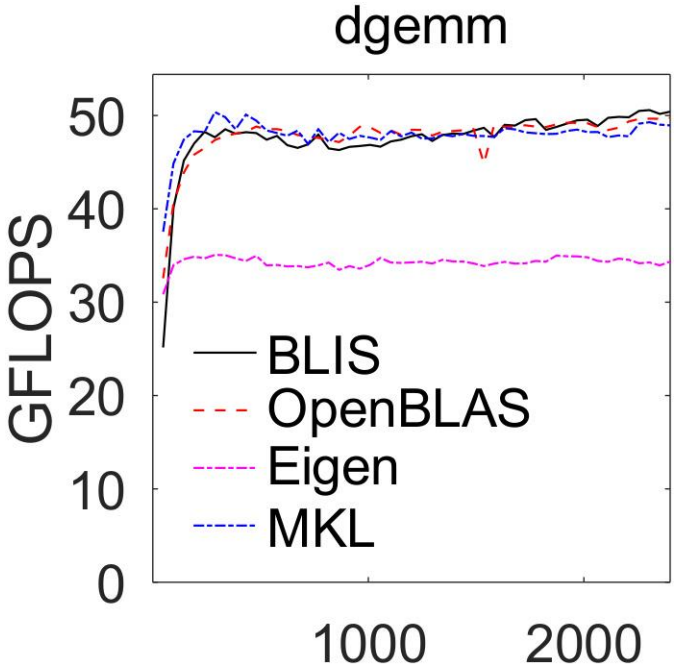
- BLIS…
  - Factors out as much complexity as possible from performance-sensitive kernel code, leaving only the micro-kernel
  - Significantly reduces the size and complexity of the kernels that must be optimized to achieve high performance
  - Provides generic, portable instances of factored codes (macro-kernels) as well as the higher-level blocked algorithms
  - Provides all packing functionality (no modification required)

BLIS performance

- *Okay, enough talk. Show me high performance!*
- Results gathered on AMD Epyc 7742 “Rome” Zen2 server
  - Before I show you full results, let’s review how to interpret each graph

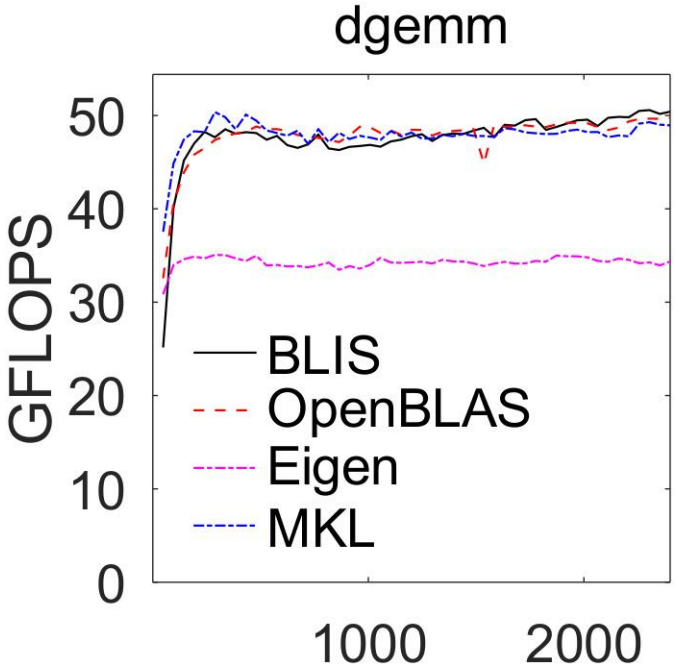
BLIS performance

- *Okay, enough talk. Show me high performance!*
- Results gathered on AMD Epyc 7742 “Rome” Zen2 server
  - *x*-axis shows problem size (all dimensions equal)
  - *y*-axis shows GFLOPS (top of graph = theoretical peak performance)
  - We compare BLIS to other BLAS implementations provided by
    - OpenBLAS 0.3.10
    - Eigen 3.3.90
    - Intel MKL 2020 Update 3

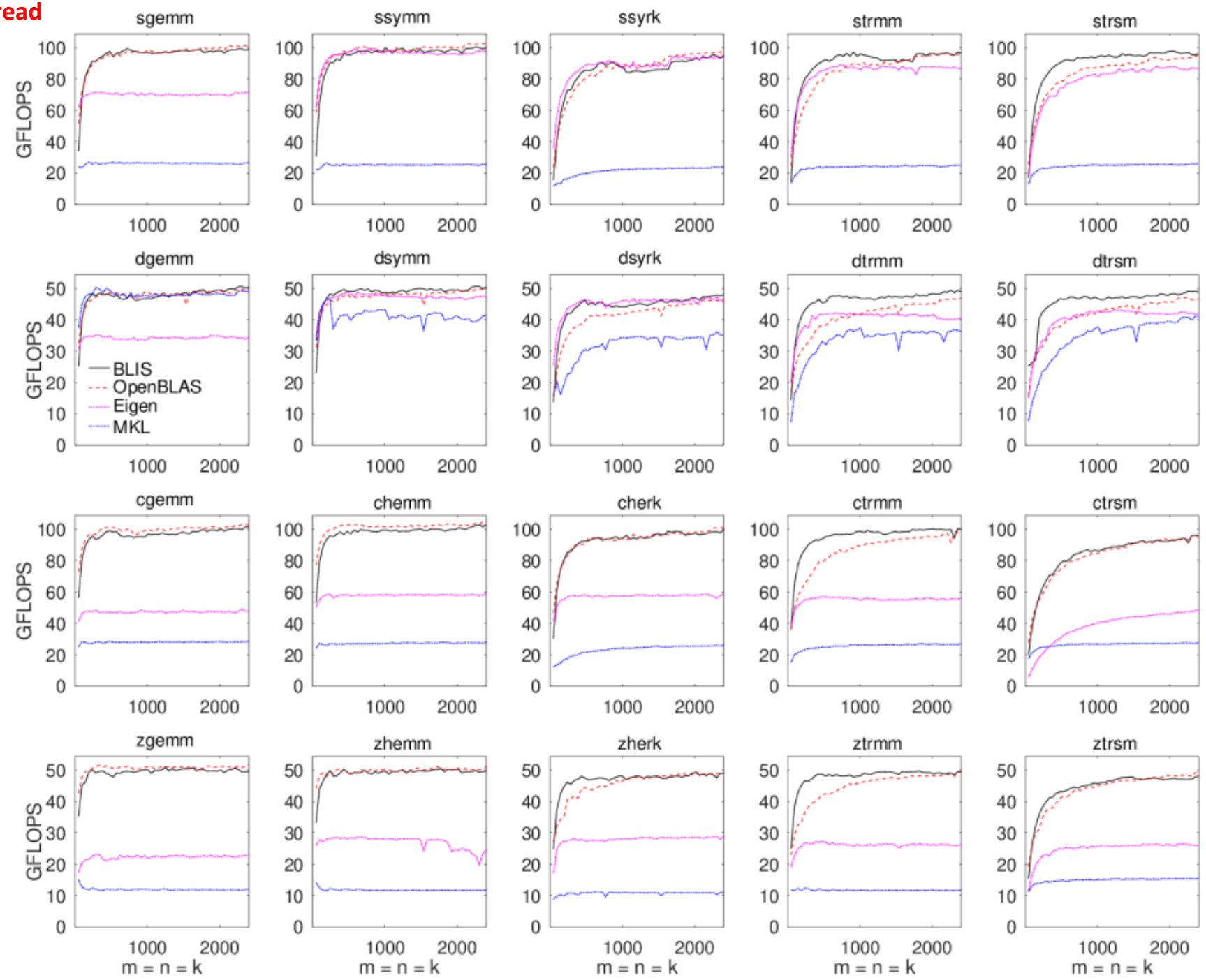


BLIS performance

- *Okay, enough talk. Show me high performance!*
- Results gathered on AMD Epyc 7742 “Rome” Zen2 server
  - *x*-axis shows problem size (all dimensions equal)
  - *y*-axis shows GFLOPS (top of graph = theoretical peak performance)
  - We compare BLIS to other BLAS implementations provided by
    - OpenBLAS 0.3.10
    - Eigen 3.3.90
    - Intel MKL 2020 Update 3
  - We do this for…
    - a representative sample of level-3 operations
    - on four floating-point datatypes (s = float, d = double, c = single complex z = double complex)

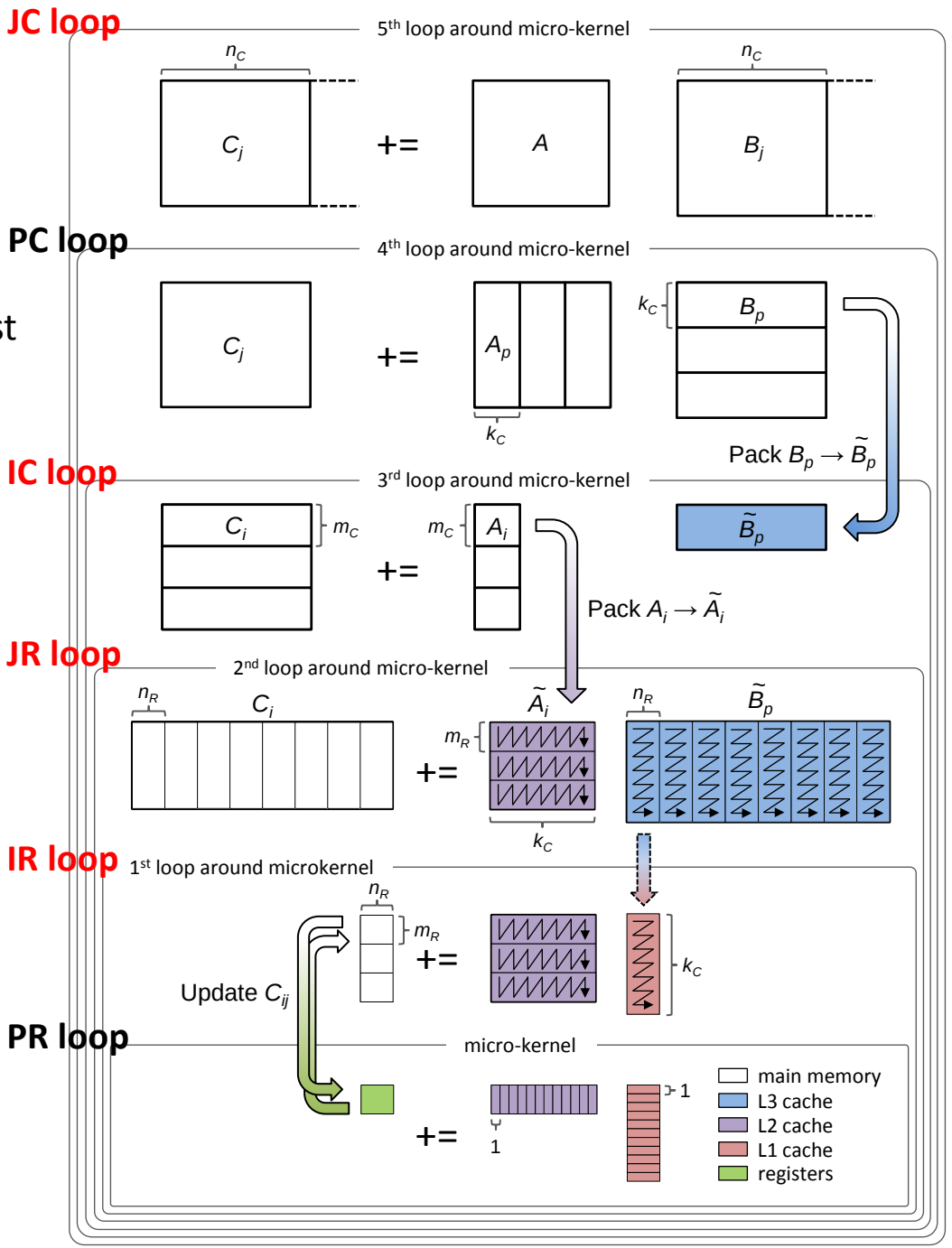


1 thread


sgemm
ssymm
ssyrk
strmm
strsm
dgemm
dsymm
dsyrk
dtrmm
dtrsm
cgemm
chemm
cherk
ctrmm
ctrsm
zgemm
zhemm
zherk
ztrmm
ztrsm
GFLOPS
m = n = k
BLIS
OpenBLAS
Eigen
MKL

Multithreading

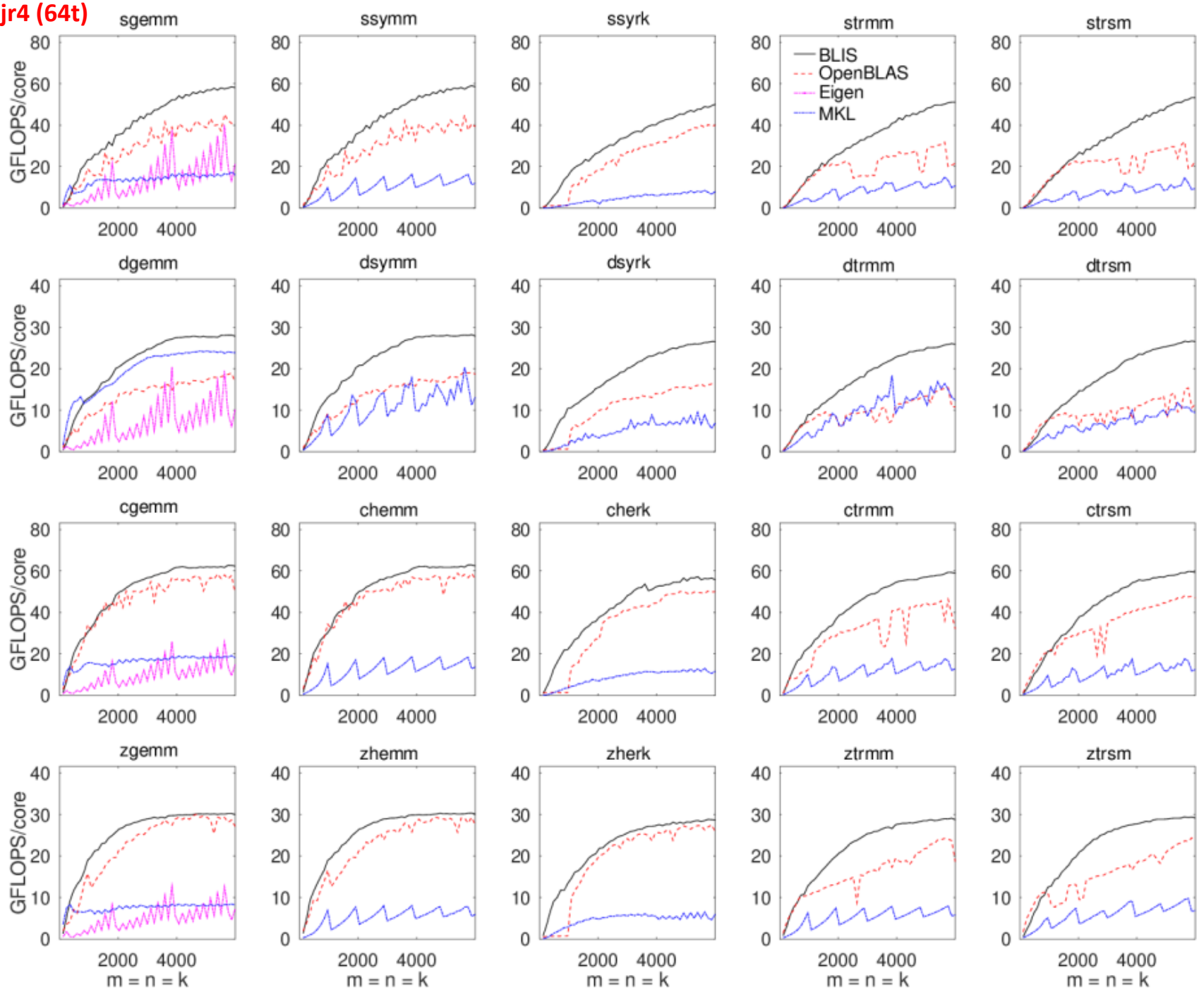
- Loops eligible for parallelism: 5th, 3rd 2nd, 1st
  - Parallelize two or more loops simultaneously
  - Ideal loops to target depend on which caches are shared vs. private
- Controlled via environment variables
  - `BLIS_JC_NT`
  - `BLIS_IC_NT`
  - `BLIS_JR_NT`
  - `BLIS_IR_NT`
- Can use either OpenMP or POSIX threads



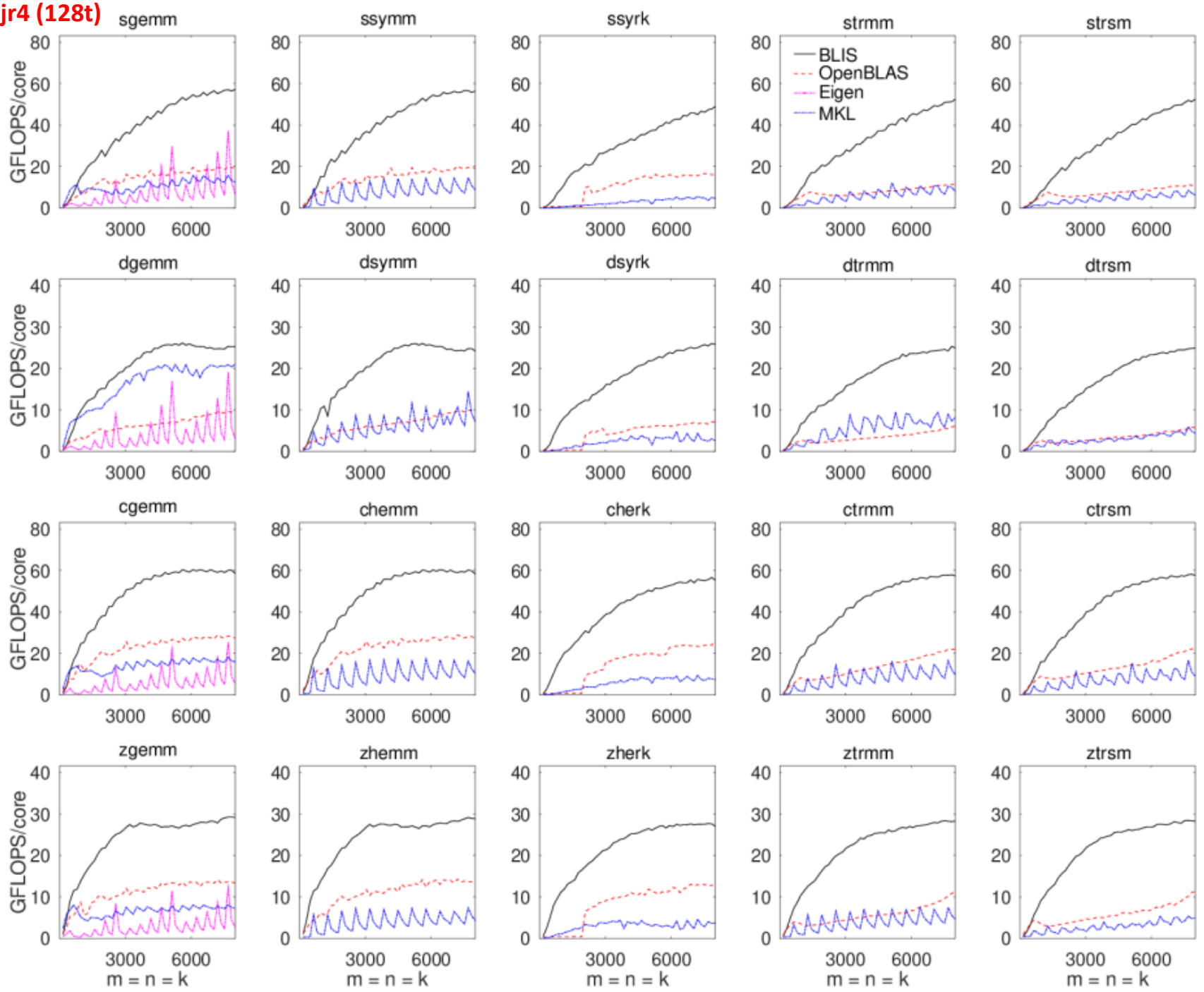
BLIS multithreading

- Multithreaded performance in BLIS
  - How does it compare to alternatives?
    - OpenBLAS, Intel's MKL, Eigen
  - Note: y-axis now shows GFLOPS/core
    - Top of graph still represents peak performance
  - Test hardware
    - AMD Epyc 7742 "Rome" Zen2 (2 sockets, 64 cores each)

jc4ic4jr4 (64t)



jc8ic4jr4 (128t)



Publications

- BLIS
  - Van Zee and van de Geijn. *"BLIS: A Framework for Rapid Instantiation of BLAS Functionality"* (ACM TOMS 2015)
  - Van Zee et al. *"The BLIS Framework: Experiments in Portability"* (ACM TOMS 2016)
  - Smith et al. *"Anatomy of Many-Threaded Matrix Multiplication"* (IPDPS 2014; in proceedings)
  - Low et al. *"Analytical Modeling is Enough for High-Performance BLIS"* (ACM TOMS 2016)
  - Van Zee and Smith. *"Implementing High-Performance Complex Matrix Multiplication via the 3m and 4m Methods"* (ACM TOMS 2017)
  - Van Zee. *"Implementing High-Performance Complex Matrix Multiplication via the 1m Method"* (SISC 2020)
  - Van Zee et al. *"Supporting Mixed-Domain Mixed-Precision Matrix Multiplication within the BLIS Framework"* (ACM TOMS; to appear)

Publications

- BLIS spin-offs and related efforts by collaborators
  - Devin A. Matthews. *"High-Performance Tensor Contraction Without Transposition."* (SISC 2015)
  - Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, George Biros. *"Performance Optimization for the K Nearest-Neighbors Kernel on x86 Architectures."* (SC 2015)
  - Jianyu Huang, Tyler M. Smith, Greg M. Henry, Robert A. van de Geijn. *"Strassen's Algorithm Reloaded."* (SC 2016)
  - Jianyu Huang, Leslie Rice, Devin A. Matthews, Robert A. van de Geijn. *"Generating Families of Practical Fast Matrix Multiplication Algorithms."* (IPDPS 2017)
  - Jianyu Huang, Chenhan D. Yu, Robert A. van de Geijn. *"Strassen's algorithm reloaded for GPUs."* (TOMS 2020)
  - Tyler M. Smith, Robert A. van de Geijn. *"The MOMMS Family of Matrix Multiplication Algorithms."* (arXiv 2019)

Investing Organizations

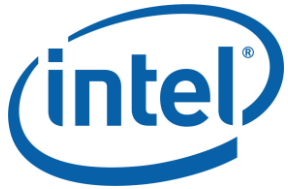
- NSF, 2012-present
  - Award ACI-1148125/1340293: *SI2-SSI: A Linear Algebra Software Infrastructure for Sustained Innovation in Computational Chemistry and other Sciences.* (Funded June 1, 2012 - May 31, 2015.)
  - Award CCF-1320112: *SHF: Small: From Matrix Computations to Tensor Computations.* (Funded August 1, 2013 - July 31, 2016.)
  - Award ACI-1550493: *SI2-SSI: Sustaining Innovation in the Linear Algebra Software Stack for Computational Chemistry and other Sciences*. (Funded July 15, 2016 – June 30, 2018.)
  - Award CSSI-2003921: *Collaborative Research: Frameworks: Beyond BLAS: A Framework for Accelerating Computational and Data Science.* (Funded May 1, 2020 - April 30, 2023.)

Investing Organizations

- Industry (gifts, grants, and hardware), 2011-present
  - Microsoft
  - Texas Instruments
  - Intel
  - AMD
  - HP Enterprise
  - Oracle
  - Huawei
  - Facebook
  - ARM







Takeaways

- BLIS is more than BLAS!
- BLIS benefits basic end-users
  - More flexible interface
- BLIS benefits developers
  - Provides a portable framework and reduces amount of code to be optimized and maintained
  - Allows rapid instantiation to new hardware
  - Contains infrastructure for implementing new operations

Takeaways

- BLIS benefits experts, SC/HPC Researchers
  - Access to low-level routines/kernels
  - Provides a platform for experimentation and prototyping
  - Foundation for mixed-domain, mixed-precision operations
- BLIS benefits everyone
  - BLIS facilitates high performance
  - Reasonably compact, readable code
  - Free / open source software – available under BSD license

Thank you!

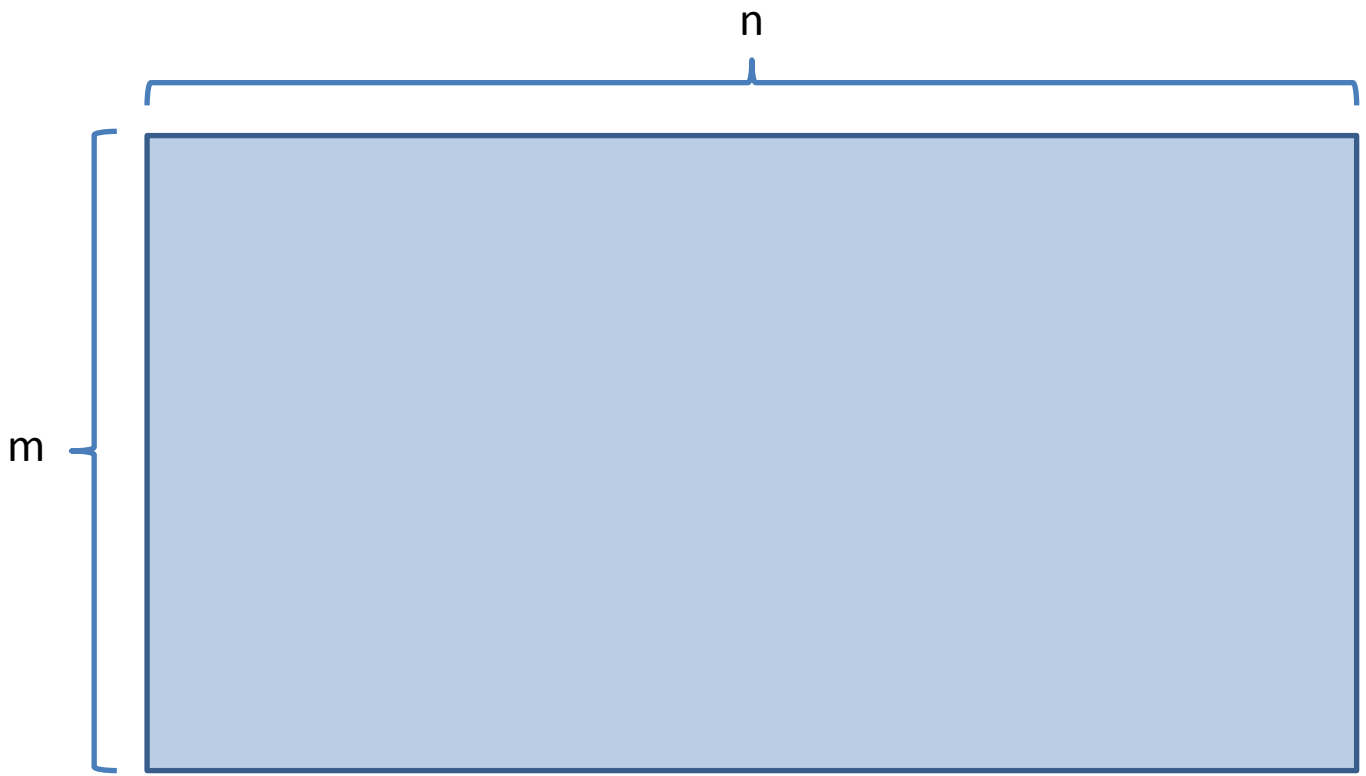
- Questions?

Bonus Topics

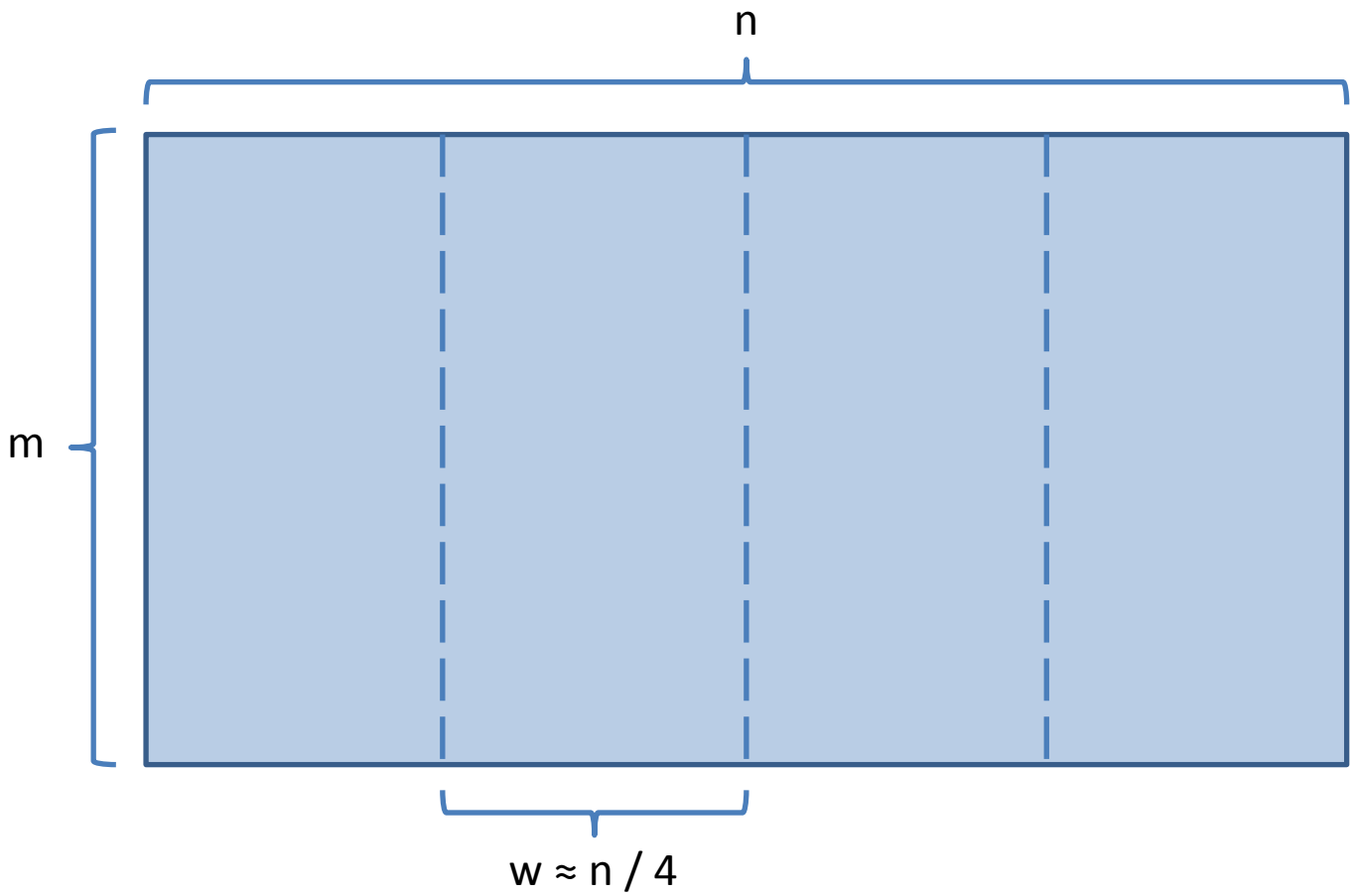
BLIS multithreading

- Quadratic partitioning
  - The topic: partitioning a submatrix for the purposes of multithreaded parallelism
  - The question: how to determine subpartition dimensions
  - For the following illustrative examples, assume:
    - We want four ways of parallelism (four threads)
    - We only partition in one dimension at a time

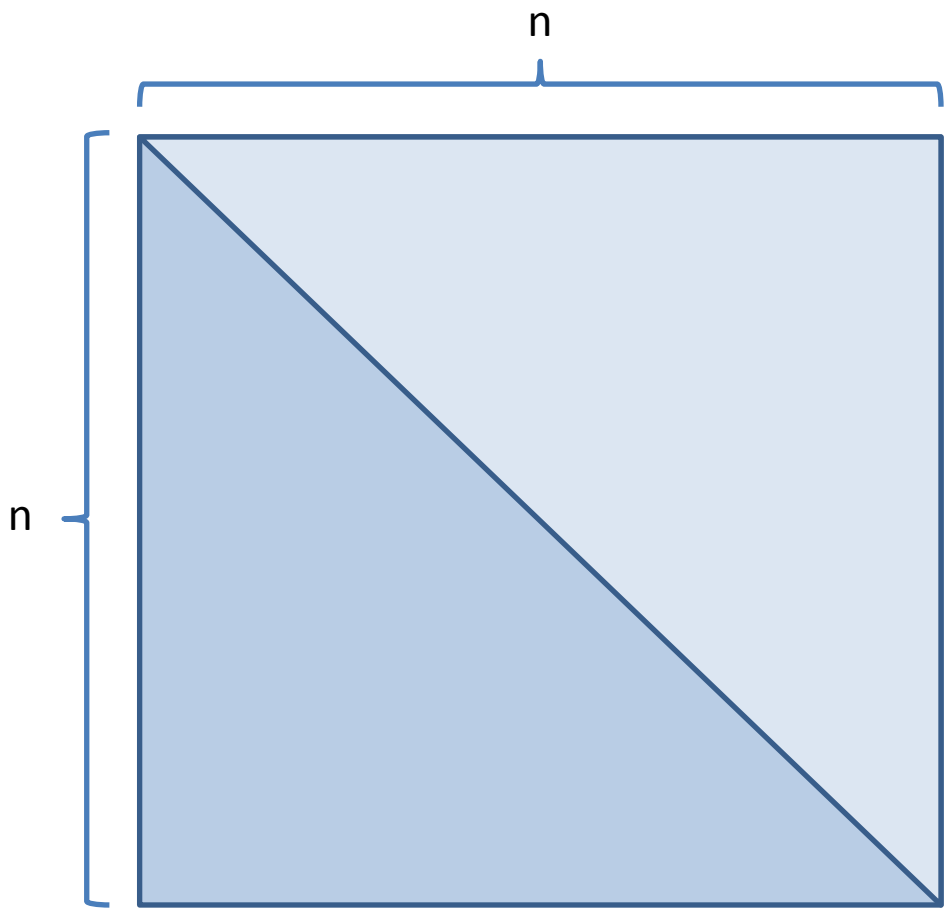
BLIS multithreading



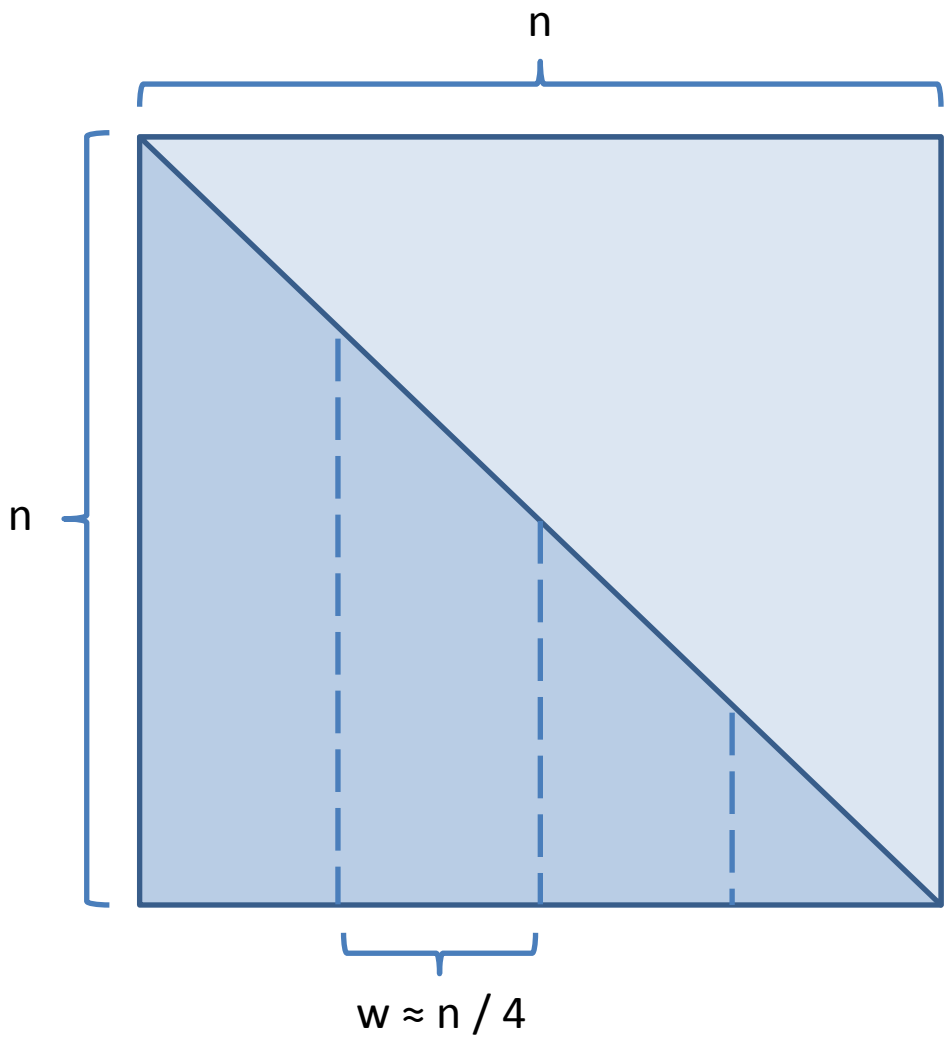
BLIS multithreading



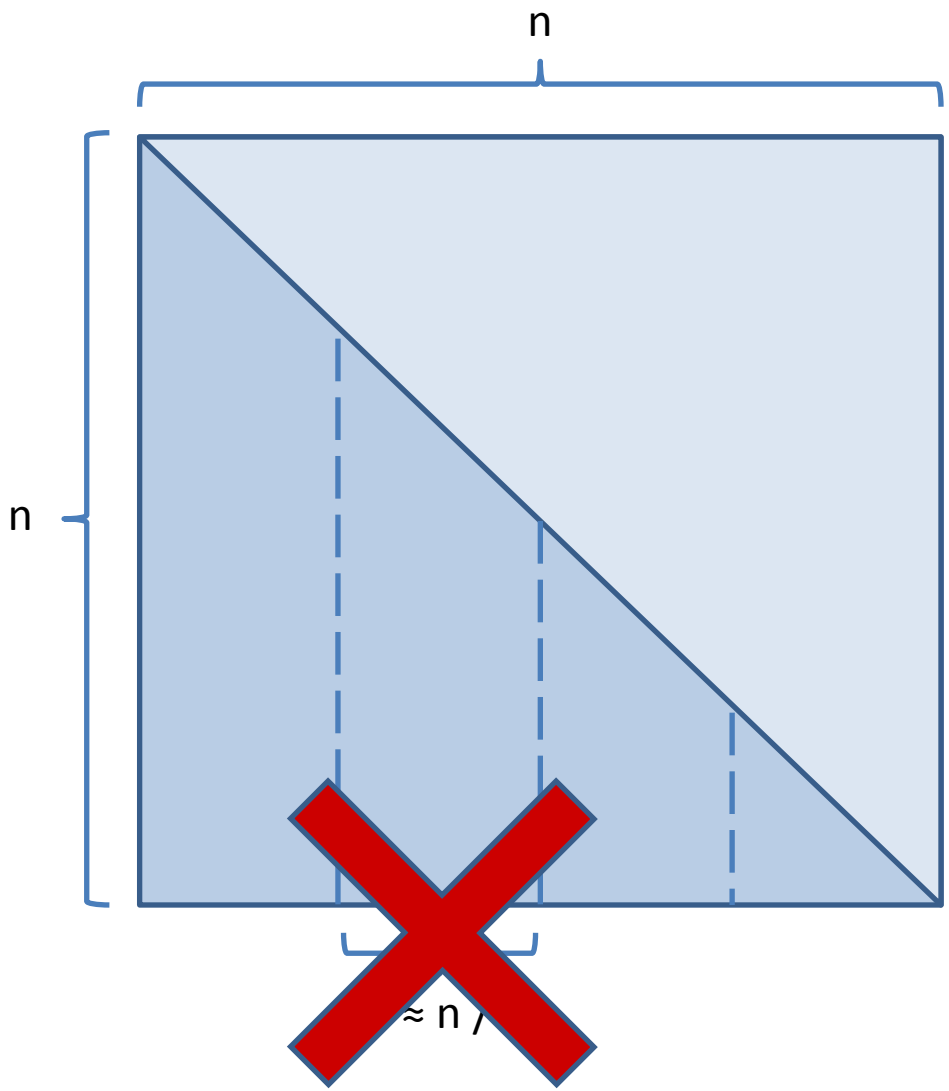
BLIS multithreading



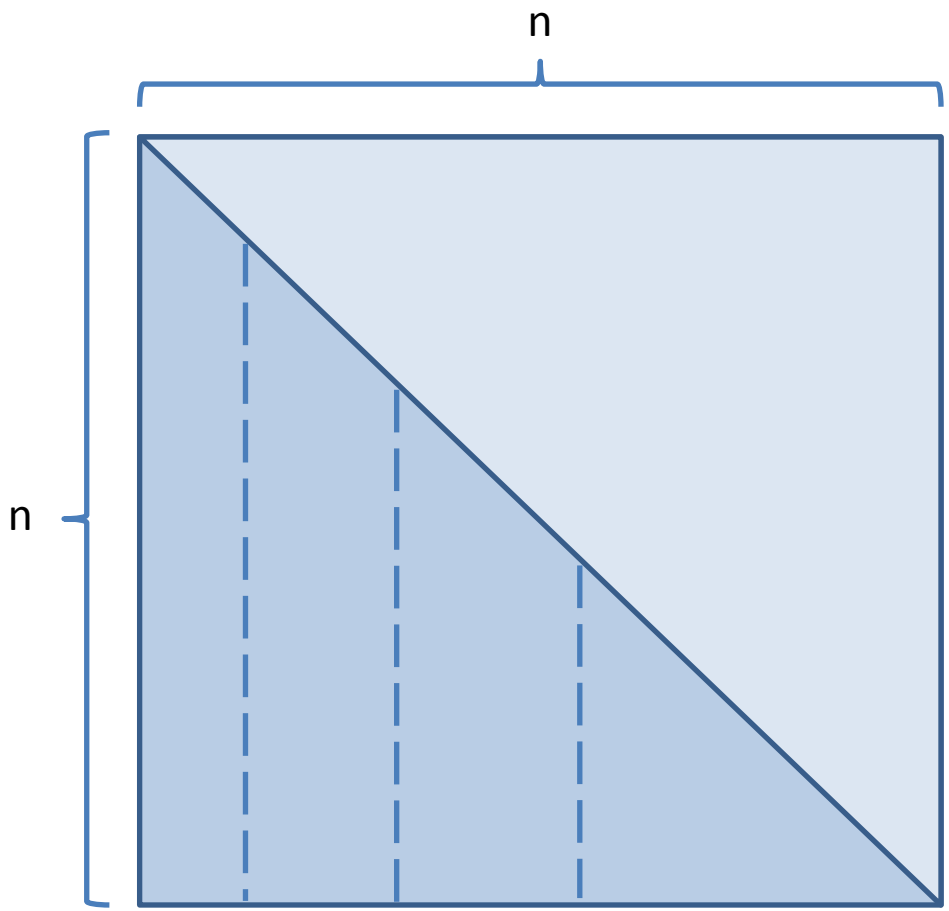
BLIS multithreading



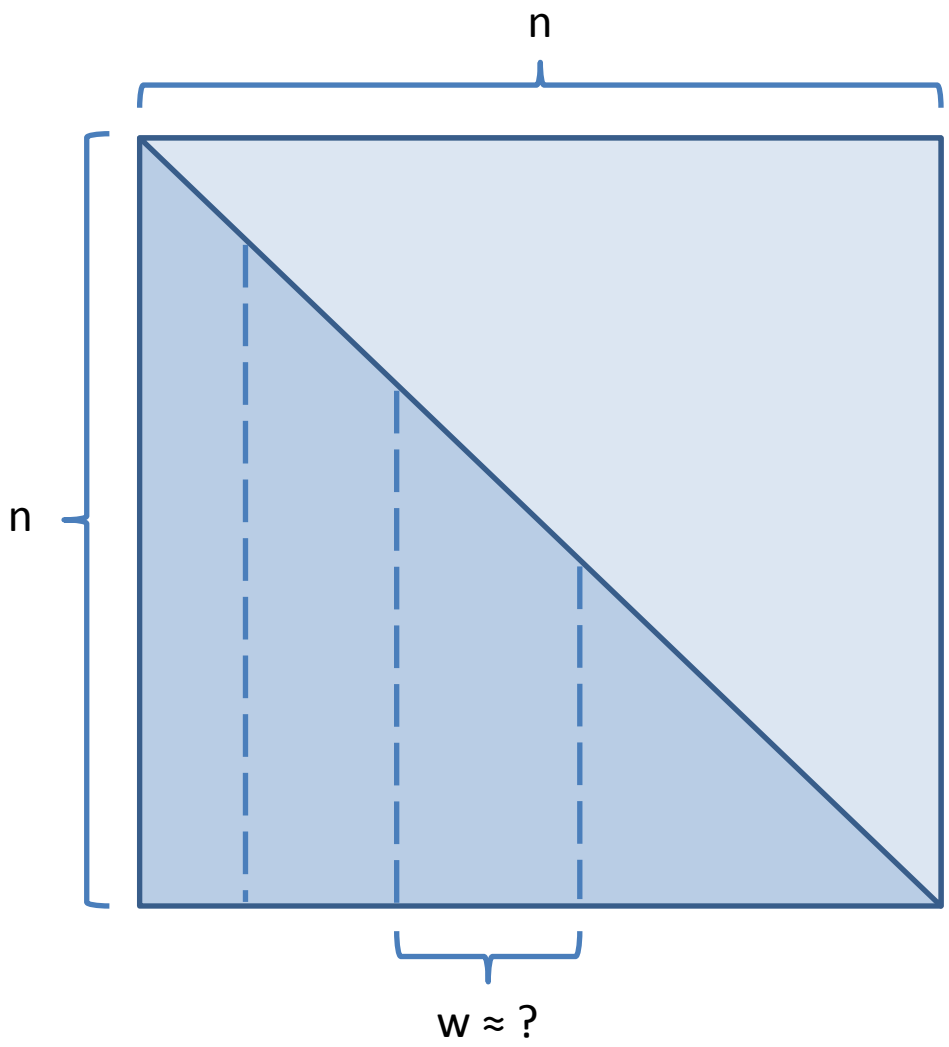
BLIS multithreading



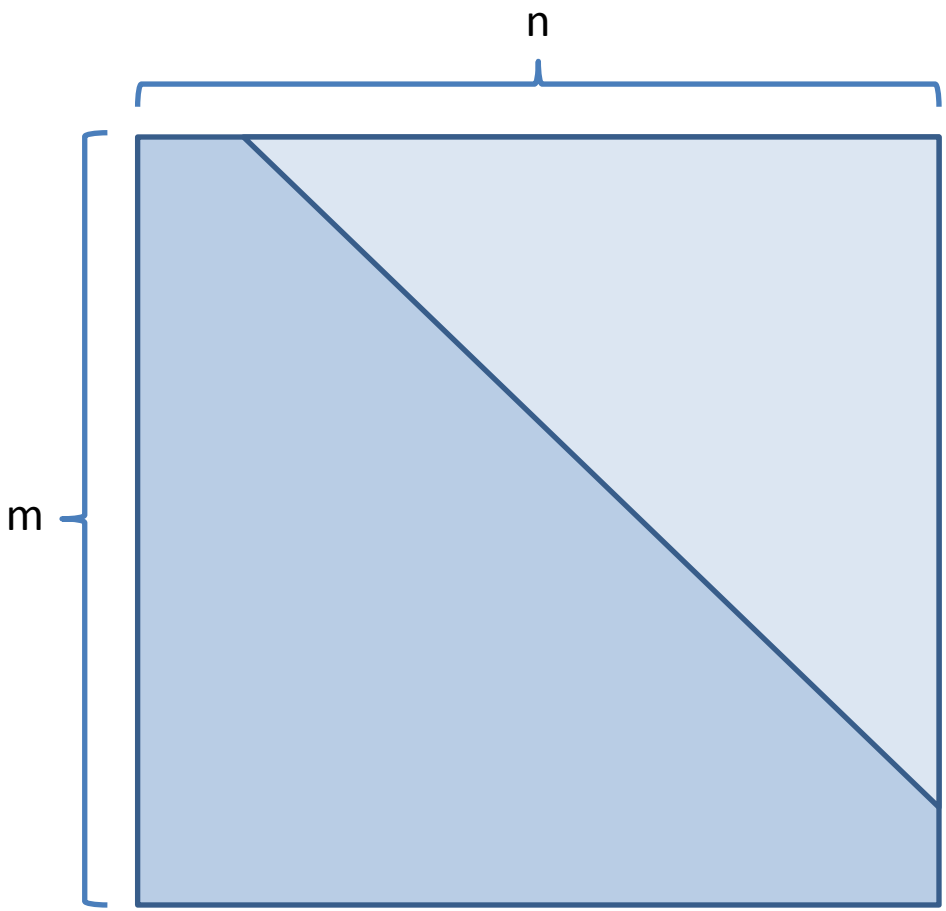
BLIS multithreading



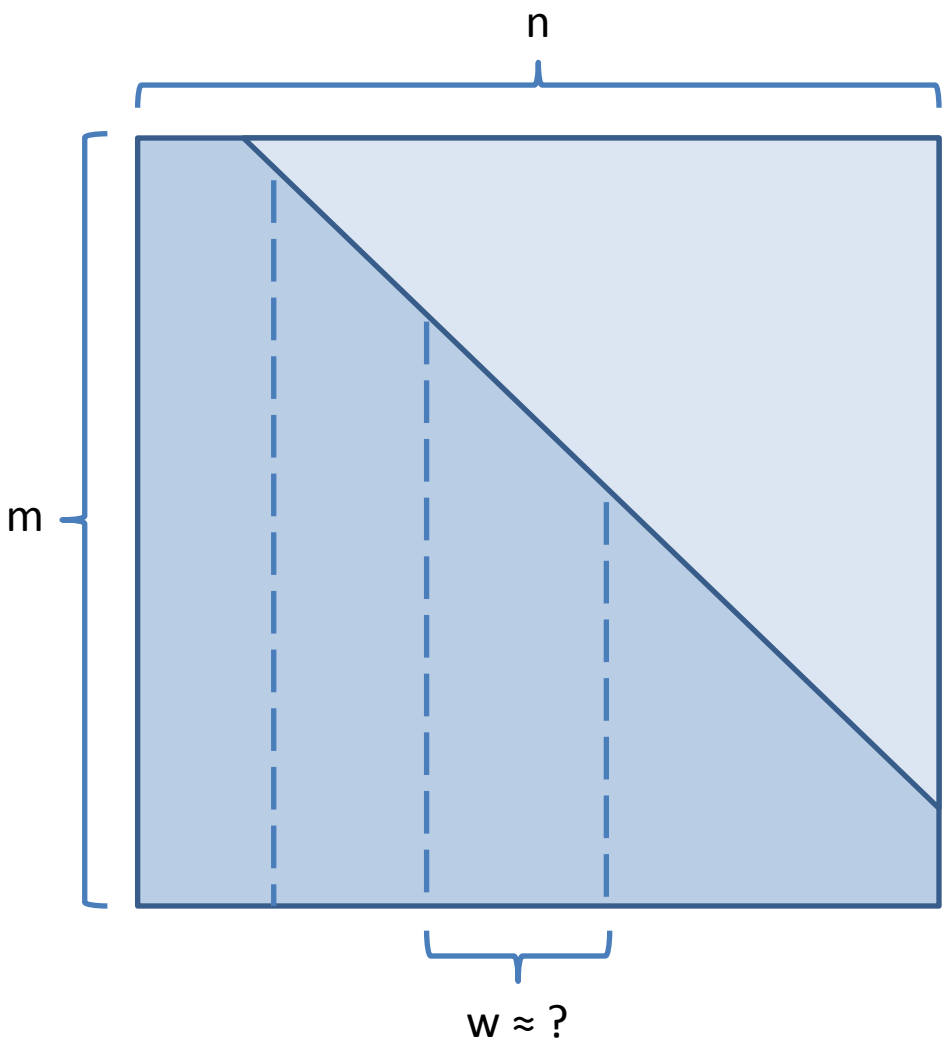
BLIS multithreading



BLIS multithreading



BLIS multithreading

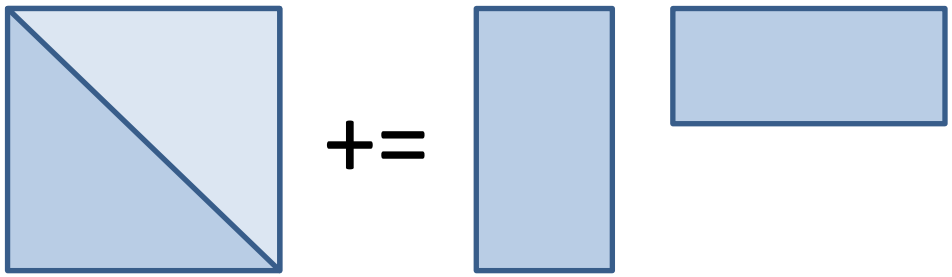


BLIS multithreading

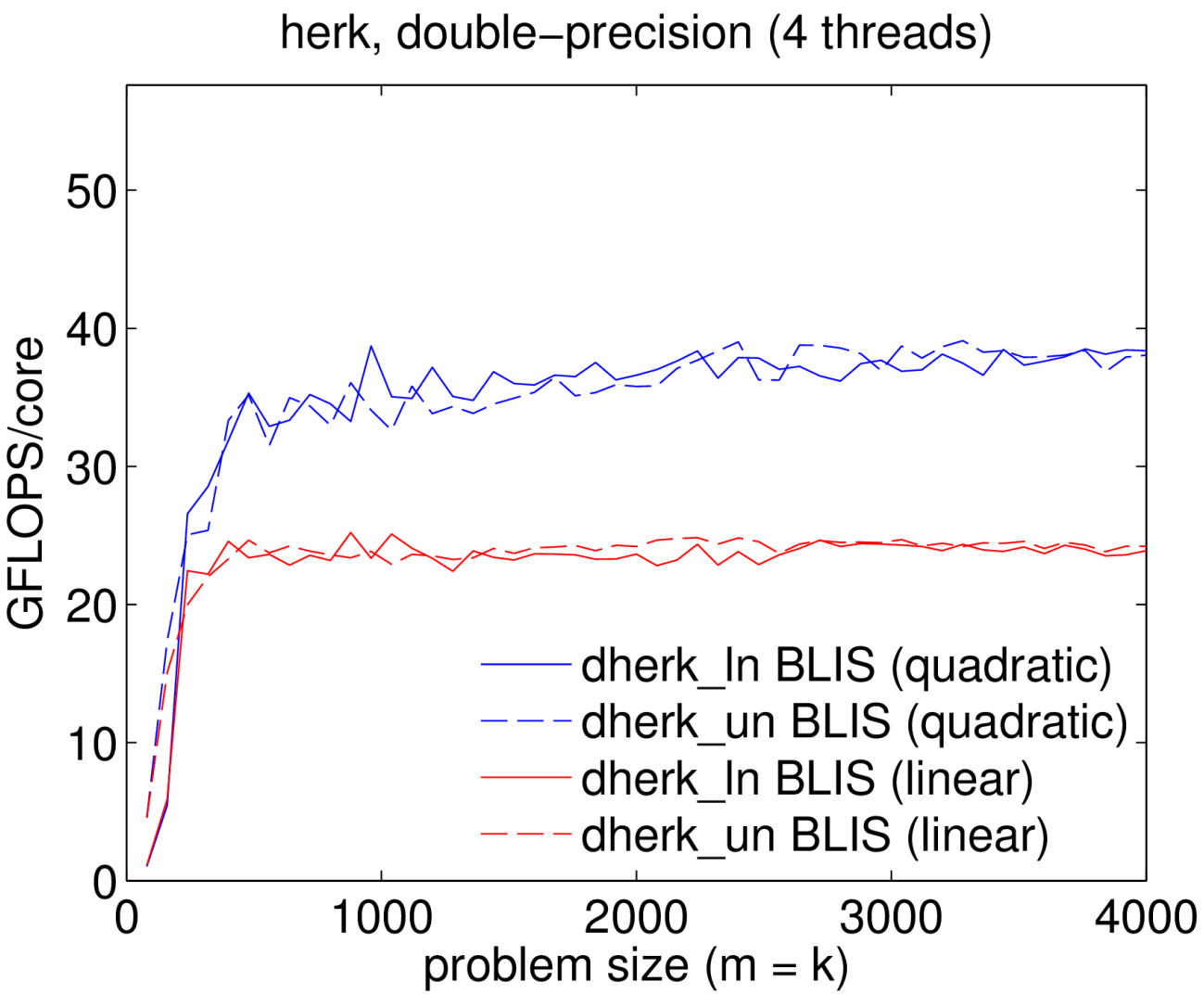
- Quadratic partitioning
  - Affects: herk, her2k, syrk, syr2k, trmm, trmm3
  - Arbitrary quasi-trapezoids (trapezoid-oids?)
  - Arbitrary diagonal offsets
  - Lower- or upper-stored Hermitian/symmetric or triangular matrices
  - Partition along m or n dimension, forwards or backwards
    - This matters because of edge case placement
  - Subpartitions must be multiples of “blocking factors” (ie: register blocksizes), except the subpartition containing edge case, if it exists

BLIS multithreading

- Quadratic partitioning
  - How much does it matter? Let's find out!
  - Test hardware
    - 3.6 GHz Intel Haswell (4 cores)
  - Test operation
    - Hermitian rank-k update: $C := C + AAH$



BLIS multithreading



APIs

- BLAS (Fortran-compatible)
- CBLAS (C conventions + row-major support)
- BLIS
  - Object API, e.g. `bli_gemm()`
  - Typed API, e.g. `bli_dgemm()`
  - Both offer basic + expert sub-interfaces
- And all APIs can be called from C++

BLAS vs CBLAS interfaces

```
// BLAS
void dgemm_
(
  char*   transa,
  char*   transb,
  int*    m,
  int*    n,
  int*    k,
  double* alpha,
  double* a, int* lda,
  double* b, int* ldb,
  double* beta,
  double* c, int* ldc
);
```

```
// CBLAS
void cblas_dgemm
(
  enum CBLAS_ORDER     order,
  enum CBLAS_TRANSPOSE transa,
  enum CBLAS_TRANSPOSE transb,
  int            m,
  int            n,
  int            k,
  double         alpha,
  const double* a, int lda,
  const double* b, int ldb,
  double         beta,
  double*        c, int ldc
);
```

BLAS vs Typed BLIS Interfaces

```
// BLAS
void dgemm_
(
  char*   transa,
  char*   transb,
  int*    m,
  int*    n,
  int*    k,
  double* alpha,
  double* a, int* lda,
  double* b, int* ldb,
  double* beta,
  double* c, int* ldc
);
```

```
// Typed API (basic)
void bli_dgemm
(
  trans_t transa,
  trans_t transb,
  dim_t   m,
  dim_t   n,
  dim_t   k,
  double* alpha,
  double* a, inc_t rsa, inc_t csa,
  double* b, inc_t rsb, inc_t csb,
  double* beta,
  double* c, inc_t rsc, inc_t csc
);
```

Object vs Typed BLIS Interfaces

```
// Object API (basic)
void bli_gemm
(
  obj_t*  alpha,
  obj_t*  a,
  obj_t*  b,
  obj_t*  beta,
  obj_t*  c
);
```

```
// Typed API (basic)
void bli_dgemm
(
  trans_t transa,
  trans_t transb,
  dim_t   m,
  dim_t   n,
  dim_t   k,
  double* alpha,
  double* a, inc_t rsa, inc_t csa,
  double* b, inc_t rsb, inc_t csb,
  double* beta,
  double* c, inc_t rsc, inc_t csc
);
```

Object vs Typed BLIS Interfaces

```
// Object API (expert)
void bli_gemm_ex
(
  obj_t*  alpha,
  obj_t*  a,
  obj_t*  b,
  obj_t*  beta,
  obj_t*  c,
  cntx_t* cntx,
  rntm_t* rntm
);
```

```
// Typed API (expert)
void bli_dgemm_ex
(
  trans_t transa,
  trans_t transb,
  dim_t   m,
  dim_t   n,
  dim_t   k,
  double* alpha,
  double* a, inc_t rsa, inc_t csa,
  double* b, inc_t rsb, inc_t csb,
  double* beta,
  double* c, inc_t rsc, inc_t csc,
  cntx_t* cntx,
  rntm_t* rntm
);
```

Implementation language

- BLIS is implemented in ISO C99
  - "Why not Fortran?"
  - Performance-critical kernels are expressed in assembly code or intrinsics
  - Thus, the higher-level framework could be anything
- BLIS makes ample use of the C preprocessor for source code templatization
  - Write one cpp macro and invoke once per datatype

Controlling Multithreading

- Reminder
  - BLIS's gemm algorithm has five loops outside the micro-kernel
  - Four of these loops may be parallelized in BLIS
    - JC
    - PC (parallelism not yet enabled)
    - IC
    - JR
    - IR
    - PR (microkernel)

Controlling Multithreading

- Three methods of specifying multithreading
  - Global specification via environment variables
    - Affects all threads
  - Global specification via runtime API
    - Affects all threads
  - Thread-local specification via runtime API
    - Affects only the calling thread!

Controlling Multithreading

- Global specification via environment variables
  - Example:

```
# Use either the automatic way or manual way of requesting
# parallelism.

# Automatic way.
$ export BLIS_NUM_THREADS = 6

# Expert way.
$ export BLIS_IC_NT = 2; export BLIS_JR_NT = 3

// Call a level-3 operation (basic interface is enough).
// Typed API responds similarly.
bli_gemm( &alpha, &a, &b, &beta, &c );
```

Controlling Multithreading

- Global specification via runtime API
  - Example:

```
// Use either the automatic way or manual way of requesting
// parallelism.

// Automatic way.
bli_thread_set_num_threads( 6 );

// Manual way.
bli_thread_set_ways( 1, 1, 2, 3, 1 );

// Call a level-3 operation (basic interface is still enough).
// Typed API responds similarly.
bli_gemm( &alpha, &a, &b, &beta, &c );
```

Controlling Multithreading

- Thread-local specification via runtime API
  - Example:

```
// Declare and initialize a rntm_t object.
rntm_t rntm = BLIS_RNTM_INITIALIZER;

// Call ONE (not both) of the following to encode your
// parallelization into the rntm_t.
bli_rntm_set_num_threads( 6, &rntm );       // automatic way
bli_rntm_set_ways( 1, 1, 2, 3, 1, &rntm );  // manual way

// Call a level-3 operation via an expert interface and pass
// in your rntm_t. (NULL below requests default context.)
// Typed API responds similarly.
bli_gemm_ex( &alpha, &a, &b, &beta, &c, NULL, &rntm );
```

Controlling Multithreading

- For more details:
  - `docs/Multithreading.md`

Thread Safety

- BLIS provides unconditional thread safety*
- What does this mean?
  - BLIS always uses mechanisms provided by pthreads API to ensure synchronous access to globally-shared data structures
  - Independent of multithreading option

    `--enable-threading={pthreads|openmp}`
    - Works with OpenMP
    - Works when multithreading is disabled entirely

*Under normal usage conditions.

The 1m Method

- Goal: Reuse real-domain kernels to induce complex-domain operations
  - This would avoid the need for addition complex microkernels, which tend to be more difficult to program in assembly code

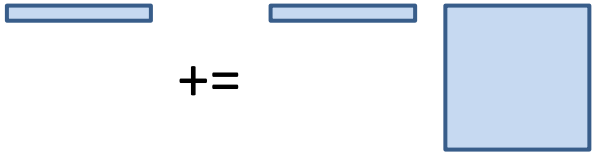
$$\begin{pmatrix} \gamma_{00}^r & \gamma_{01}^r \\ \gamma_{00}^i & \gamma_{01}^i \\ \gamma_{10}^r & \gamma_{11}^r \\ \gamma_{10}^i & \gamma_{11}^i \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{00}^r & -\alpha_{00}^i & \alpha_{01}^r & -\alpha_{01}^i \\ \alpha_{00}^i & \alpha_{00}^r & \alpha_{01}^i & \alpha_{01}^r \\ \alpha_{10}^r & -\alpha_{10}^i & \alpha_{11}^r & -\alpha_{11}^i \\ \alpha_{10}^i & \alpha_{10}^r & \alpha_{11}^i & \alpha_{11}^r \end{pmatrix} \begin{pmatrix} \beta_{00}^r & \beta_{01}^r \\ \beta_{00}^i & \beta_{01}^i \\ \beta_{10}^r & \beta_{11}^r \\ \beta_{10}^i & \beta_{11}^i \end{pmatrix}$$

- Solution: the 1m method [Van Zee 2020]
  - A is packed to micro-columns with real and imaginary elements duplicated and swapped to next column (with imaginary negated)
  - B is packed to micro-rows where imaginary elements are reordered to next row
  - Multiply A and B using normal real-domain microkernel

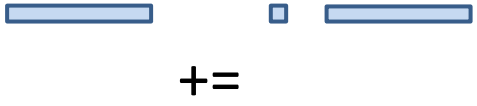
Small/skinny gemm

- Separate “sub-framework” intercepts these cases
  - Multithreading, optional packing supported

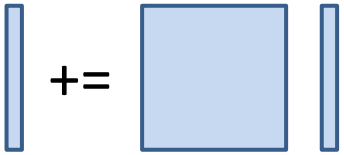
small m



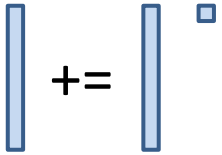
small m, k



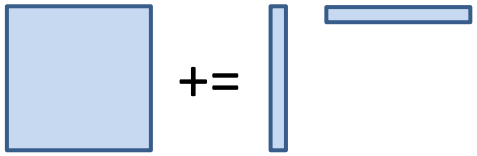
small n



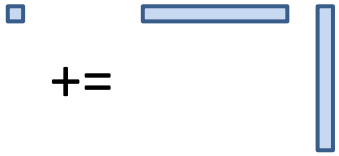
small n, k



small k



small m, n



Mixed datatype gemm

- Consider simplified gemm (no 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
  - BLIS has a gemm implementation that lifts this constraint!
  - Total number of possible cases to implement
    - Assume each operand stored as one of t storage datatypes
    - Operation may be computed in one of $t/2$ precisions (two domains)
    - In general: $N = \left(\frac{t}{2}\right) t^3 = \frac{t^4}{2}$
    - For BLIS (currently): $N = \left(\frac{4}{2}\right) 4^3 = 128$
    - Notice that BLAS implements only 4 out of the 128

Hardware Support

- AMD
  - Bulldozer, Piledriver, Steamroller, Excavator, Zen, Zen+, Zen2
- Intel
  - Core2, Sandy/Ivy Bridge, Haswell/Broadwell, Skylake[X], KNL, Kaby/Coffee Lake, and beyond
- IBM BlueGene/Q, Power9, Power10
- ARM (v7a, v8a, SVE)

Build system features

- Based on GNU build process
  - configure; make; make install
- Hardware detection
  - Determines kernel and cache blocksize selections
  - Happens at configure-time (slim libraries) or runtime (fat libraries)
- Compiler flags set on a per-subconfiguration basis
  - Flags may differ from `haswell` to `skx` to `zen` to `zen2`
- Dynamically generated makefile fragments
  - Shouldn't have to edit a makefile just because I rename foo.c to bar.c
- Monolithic header generation
  - All headers (≈500) recursively inlined into `blis.h`
  - Faster compilation time
  - Easier to distribute build products

Testing

- Unified testsuite
  - correctness and performance, BLIS-only
- netlib BLAS test drivers
  - correctness, BLAS-only
- Standalone (comparative) performance drivers
  - BLIS vs OpenBLAS, MKL, Eigen (large and small)
  - libxsmm, BLAFEO (small only)
- Continuous integration
  - Travis CI (including Intel SDE), AppVeyor

OS Support

- Debian/Ubuntu
- Fedora/EPEL
- Gentoo
- OpenSUSE
- GNU Guix
- OS X
- Windows
  - clang via AppVeyor

User Statistics

- GitHub provides two-week rolling averages
  - 120+ unique clones
  - 600+ unique visitors
  - This doesn't count .zip file downloads or OS-specific packages (e.g. Ubuntu installs)
  - Nor does it count clones/downloads of AMD BLIS
- How many total users do we have?
  - No idea!
  - But we think it could be quite high

Community Support

- Example code (tutorial)
- Documentation
- Performance graphs
  - http://github.com/flame/blis/
- Mailing lists
  - https://groups.google.com/group/blis-devel
  - https://groups.google.com/group/blis-discuss
- GitHub issues
  - https://github.com/flame/blis/issues

What's next?

- Future directions
  - Support new datatypes
    - bfloat16, fp16, int16, etc.
  - Refactor for more exotic expert usage
  - Implement new operations
  - Support new hardware (as it becomes available)

Thank you!

- Questions?