The BLIS Approach to Skinny Matrix Multiplication

Field G. Van Zee

Science of High Performance Computing

The University of Texas at Austin

September 19, 2019

Science of High Performance Computing (SHPC) research group

- Led by Robert A. van de Geijn
- Contributes to the science of DLA and instantiates research results as open source software
- Long history of support from National Science Foundation
- Website: https://shpc.ices.utexas.edu/

SHPC Funding (BLIS)

- NSF
  - 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.)

SHPC Funding (BLIS)

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

Publications

- *"BLIS: A Framework for Rapid Instantiation of BLAS Functionality"* (TOMS; in print)
- *"The BLIS Framework: Experiments in Portability" (TOMS; in print)*
- *"Anatomy of Many-Threaded Matrix Multiplication"* (IPDPS; in proceedings)
- *"Analytical Models for the BLIS Framework"* (TOMS; in print)
- *"Implementing High-Performance Complex Matrix Multiplication via the 3m and 4m Methods"* (TOMS; in print)
- *"Implementing High-Performance Complex Matrix Multiplication via the 1m Method"* (~~TOMS~~ SISC; submitted)
- "Supporting Mixed-Domain Mixed-Precision Matrix Multiplication within the BLIS Framework" (TOMS; under revision)

Review

- 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?
  - BLAS constitute the “bottom of the food chain” for most dense linear algebra applications, as well as other HPC libraries
  - LAPACK, `libflame`, MATLAB, PETSc, numpy, gsl, etc.

Review

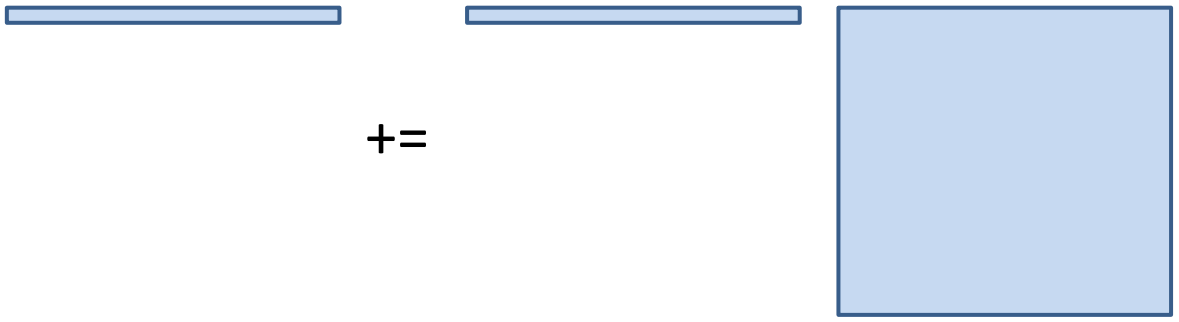
- What is BLIS?
  - A framework for instantiating BLAS libraries (ie: fully compatible with BLAS)
- What else is BLIS?
  - Provides alternative BLAS-like (C friendly) API that fixes deficiencies in original BLAS
  - Provides an object-based API
  - Provides a superset of BLAS functionality
  - A productivity multiplier
  - A research environment

Motivation

- Consider the classic gemm operation
- Typical HPC problems are “large”: what does this mean?
  - ALL matrix dimensions (m, n, k) are “large”
- BLIS’s Achilles heel: “small” matrix multiplication: why?
  - There isn’t enough computation (flops) engendered by small matrix multiplication to justify the overhead in BLIS
    - Object management, use of internal packing buffers

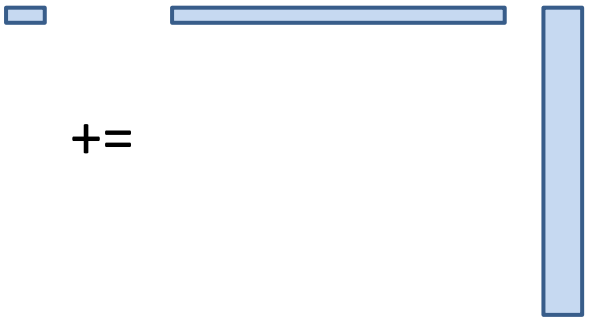
Motivation

- What happens if we consider a hybrid situation?
  - Instead of ALL matrix dimensions being small, what happens if ONE matrix dimension is small (and the other two dimensions are potentially still large-ish)?
  - How small is small? Potentially very small: ≈10 or less.
  - Example:



Motivation

- Alternatively…
  - What happens if TWO matrix dimensions are small (and the other dimension is potentially still large or large-ish)?
  - Example:



Specification

- Let’s start by specifying what a skinny gemm implementation should support

Specification

- What should a skinny gemm implementation support?
  - Various problem shape scenarios

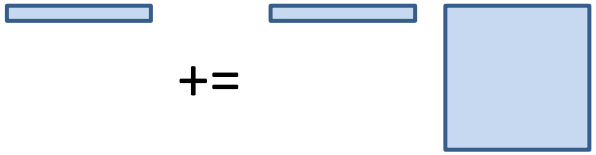
Shape Scenarios

- Six problem shape scenarios (mnk):

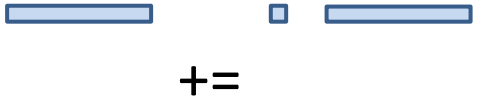
Shape Scenarios

- Six problem shape scenarios (mnk):

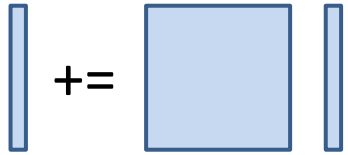
SLL: small m



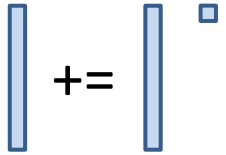
SLS: small m, k



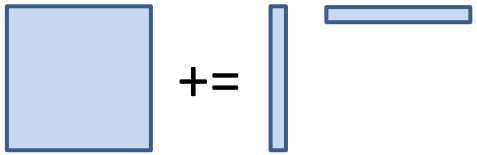
LSL: small n



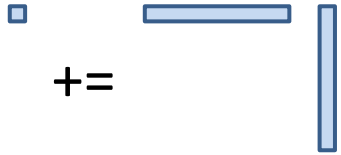
LSS: small n, k



LLS: small k



SSL: small m, n



Shape Scenarios

- Six problem shape scenarios (mnk):
- Ideally, our solution would work across as many of these shape scenarios as possible

Specification

- What should a skinny gemm implementation support?
  - Various problem shape scenarios (mnk)
    - SLL, LSL, LLS, SSL, SLS, SSL
  - Transposition on A and/or B (transA, transB)
    - NN, NT, TN, TT
    - Complex domain: conjA, conjB
  - Row and column storage (CAB)
    - RRR, RRC, RCR, RCC, CRR, CRC, CCR, CCC

Specification

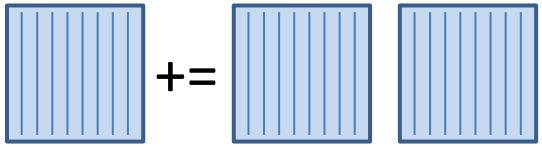
- What should a skinny gemm implementation support?
  - Avoid: assumption that A and B are packed
  - This makes supporting all eight storage combinations harder! Why? Two reasons:
    - We can't assume contiguous/unit stride on A and B
    - We have to handle edge cases explicitly rather. (Reminder: BLIS computes edge cases to temporary storage, then copies appropriate elements back to C.)
  - General stride should be supported, even if it's slow

The BLIS Approach

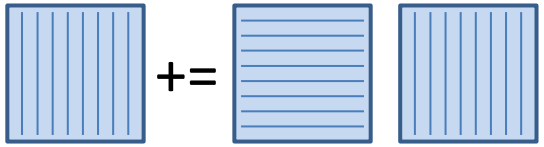
- Today, let's consider double-precision real domain only
  - Complex is possible, but more involved due to conjugation on A and/or B
- Note that transposition on A, B can be interpreted as changing the *effective* storage combination
  - Example: An m-by-n row-stored matrix with a transpose is equivalent to an n-by-m column-stored matrix (with no transpose)
  - This reduces 32 parameter cases (4 transAB x 8 storage) to 8 effective cases

Storage Combinations

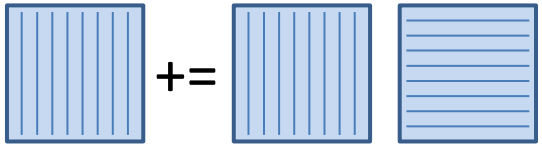
CCC



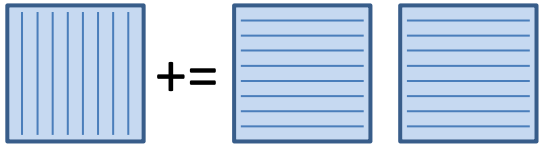
CRC



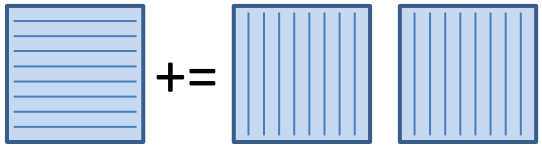
CCR



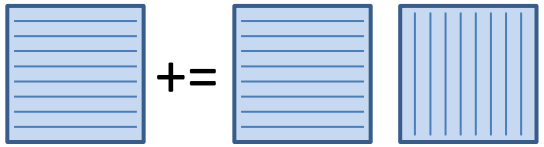
CRR



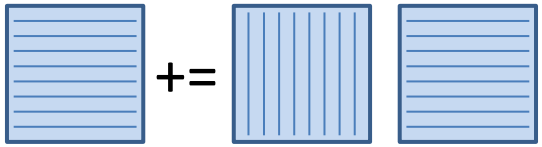
RCC



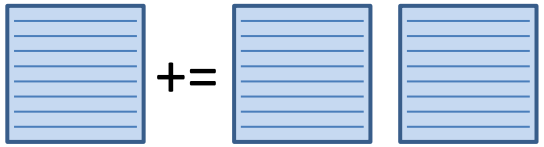
RRC



RCR



RRR



Storage Combinations

CCC

CRC

CCR

CRR

RCC

RRC

RCR

RRR

Storage Combinations

CCC

CRC

CCR

CRR

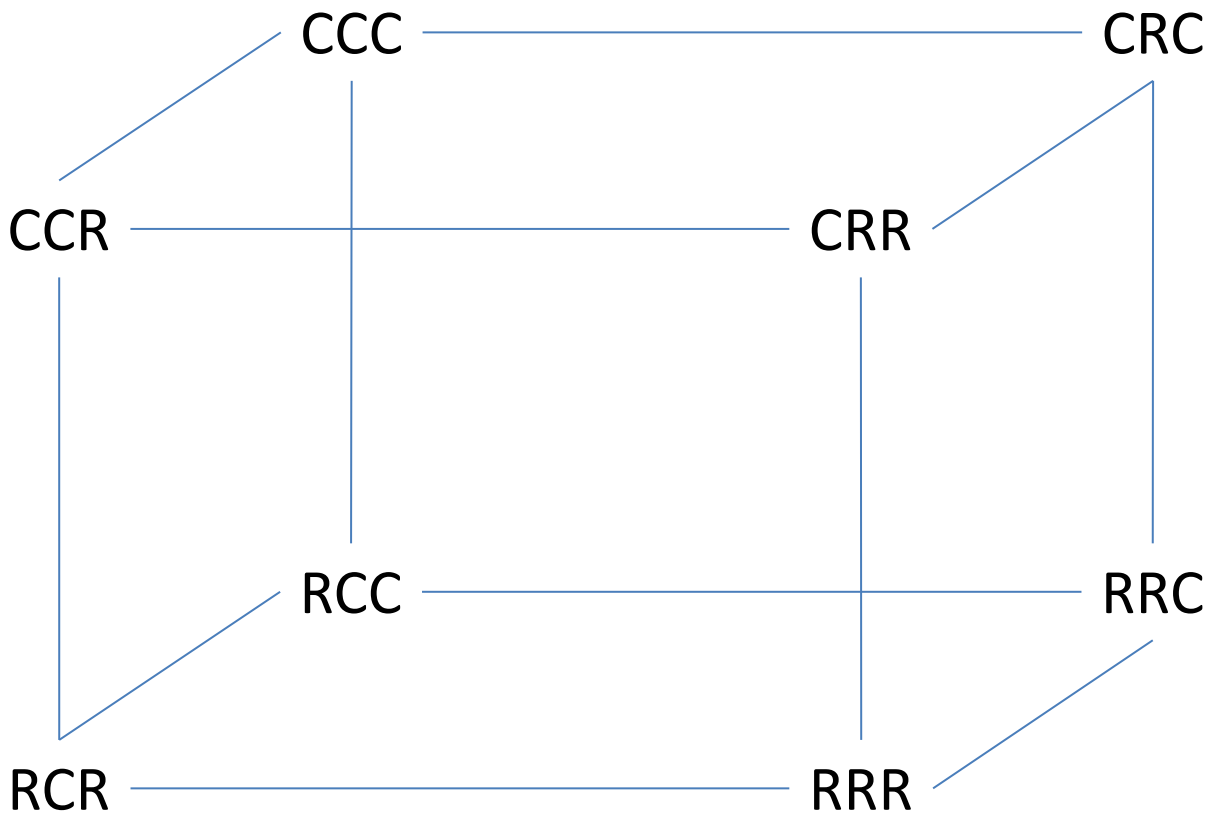
RCC

RRC

RCR

RRR

Storage Combinations

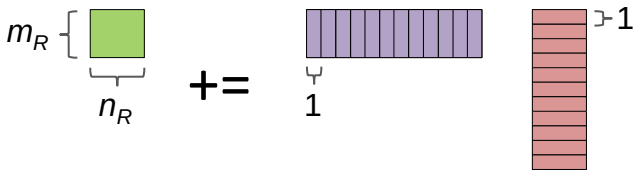


Storage Combinations

- How do we support all eight effective storage combinations?
  - Remember: we can’t assume A or B is packed

Revisiting the microkernel

- Let's review the conventional BLIS microkernel
- What do we like about it?
  - Achieves a high fraction of peak
  - Able to work with m, n dimensions that are small
- What don't we like about it?
  - Inherently has an affinity for large k dimensions
  - Depends on contiguous/packed A and B



Revisiting the microkernel

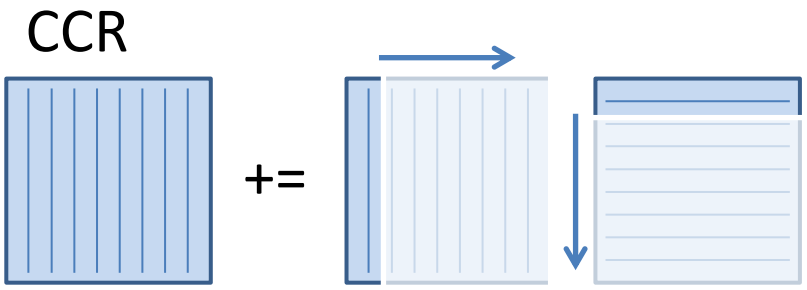
- Comments
  - Can't do much about affinity for large k
  - It's unclear how important packing really is
- Verdict
  - Let's stick with the same microkernel design
  - One big caveat: either A or B (or both) may have large leading dimensions (row stride for row storage; column stride for column storage)
    - In other words, we can't assume A or B is packed

Microkernel implementation

- Turns out that the storage of A, B, and C affects how the microkernel can be practically implemented
- Let's look at an example

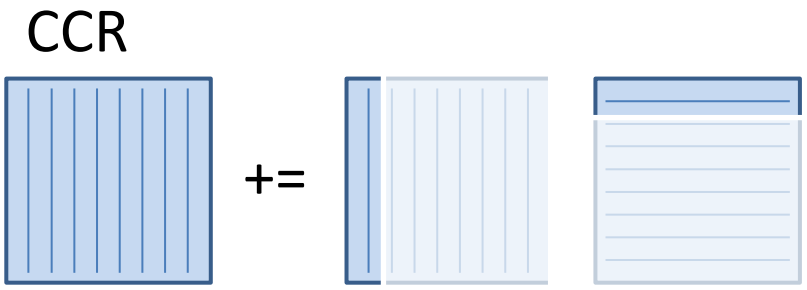
Microkernel implementation

- Microkernel consists of a loop over k dimension



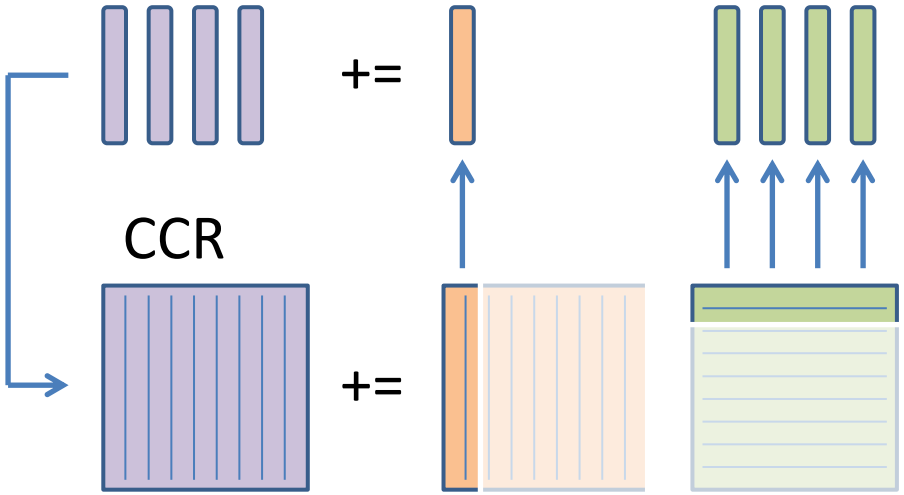
Microkernel implementation

- Two implementation options



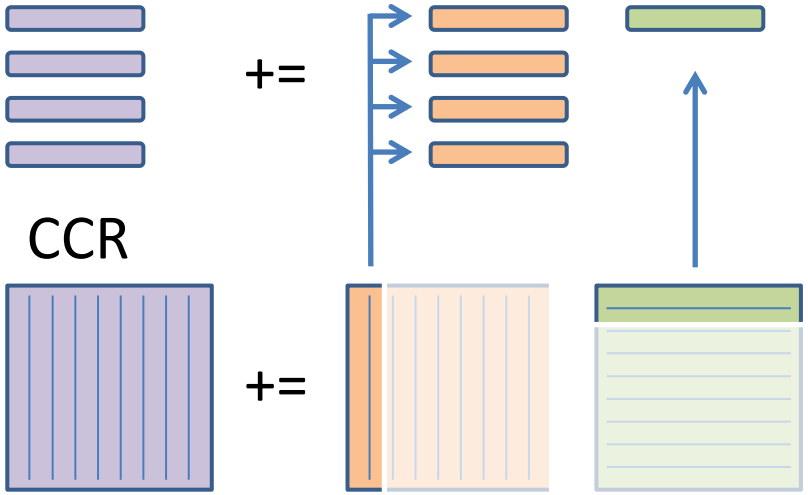
Microkernel implementation

- Two implementation options
  - Load contiguous vectors of A and broadcast from B



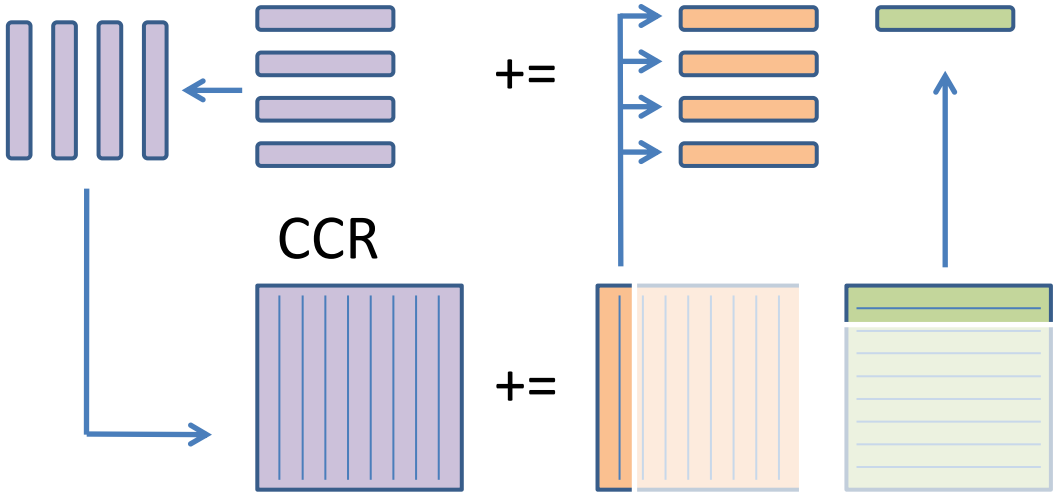
Microkernel implementation

- Two implementation options
  - Load contiguous vectors of A and broadcast from B
  - Load contiguous vectors of B and broadcast from A



Microkernel implementation

- Two implementation options
  - Load contiguous vectors of A and broadcast from B
  - Load contiguous vectors of B and broadcast from A
    - In this case, requires in-register transpose prior to I/O on C



Microkernel implementation

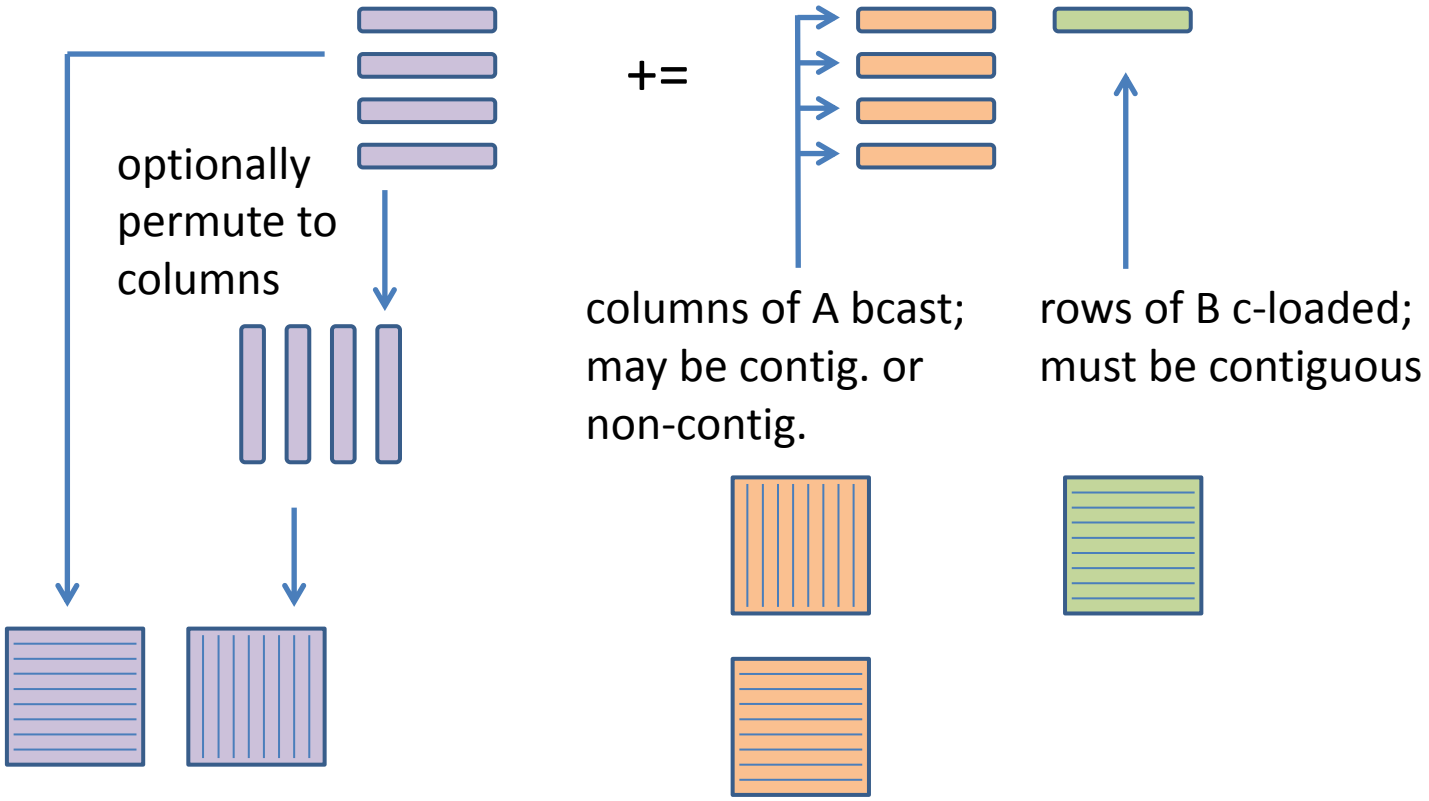
- There are other implementation strategies
- Two (somewhat orthogonal) properties:
  - The orientation of the microtile registers
    - And whether in-register transpose is needed for I/O on C
  - The instruction types used to load elements of A and B
- We want to avoid in-register transposition if possible
  - We will see that the latter component affects the former

Microkernel implementation

- So let’s enumerate the family of kernel implementation types

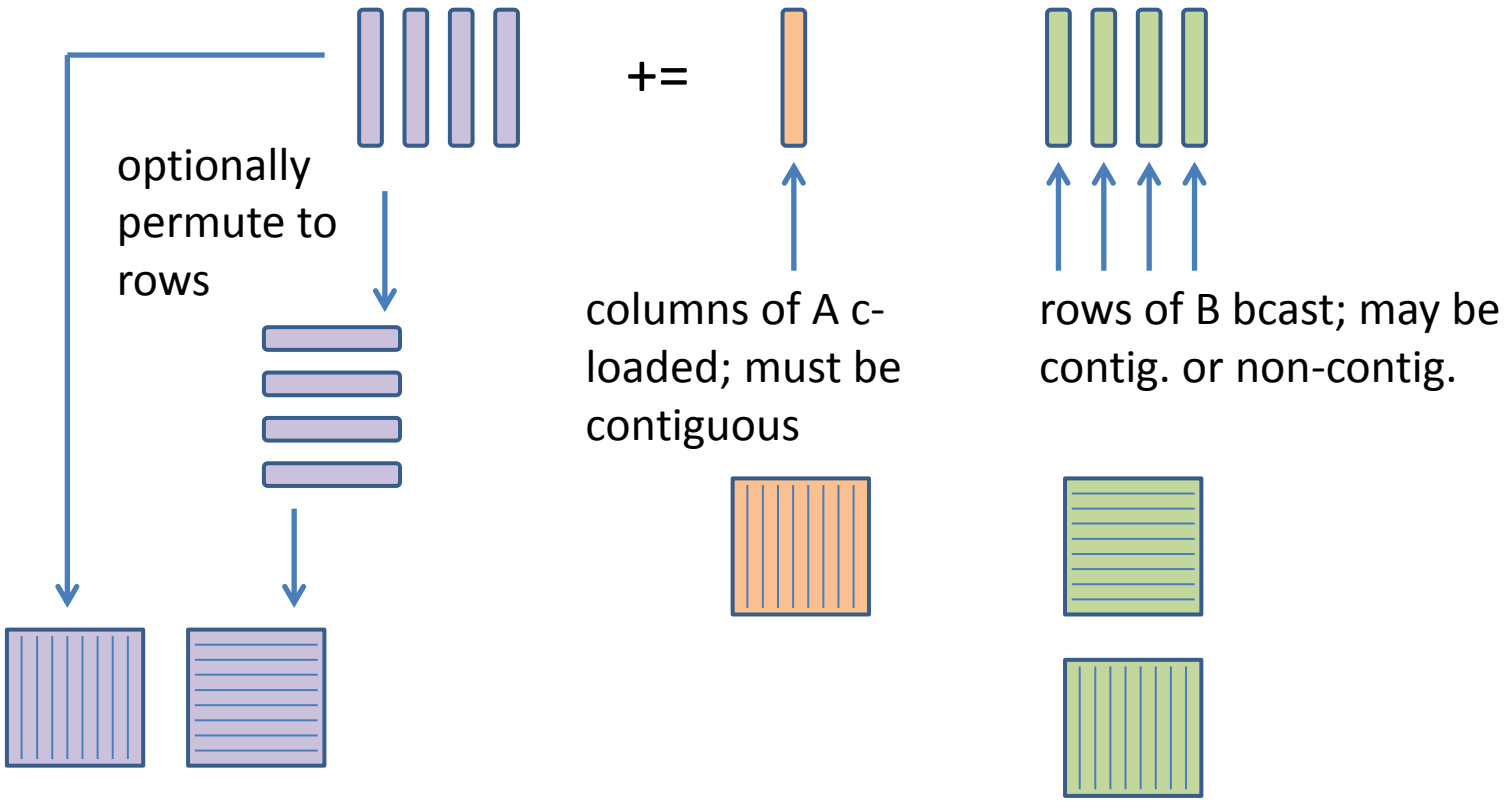
Microkernel implementation

- Row-oriented, contiguous axpy (rca)



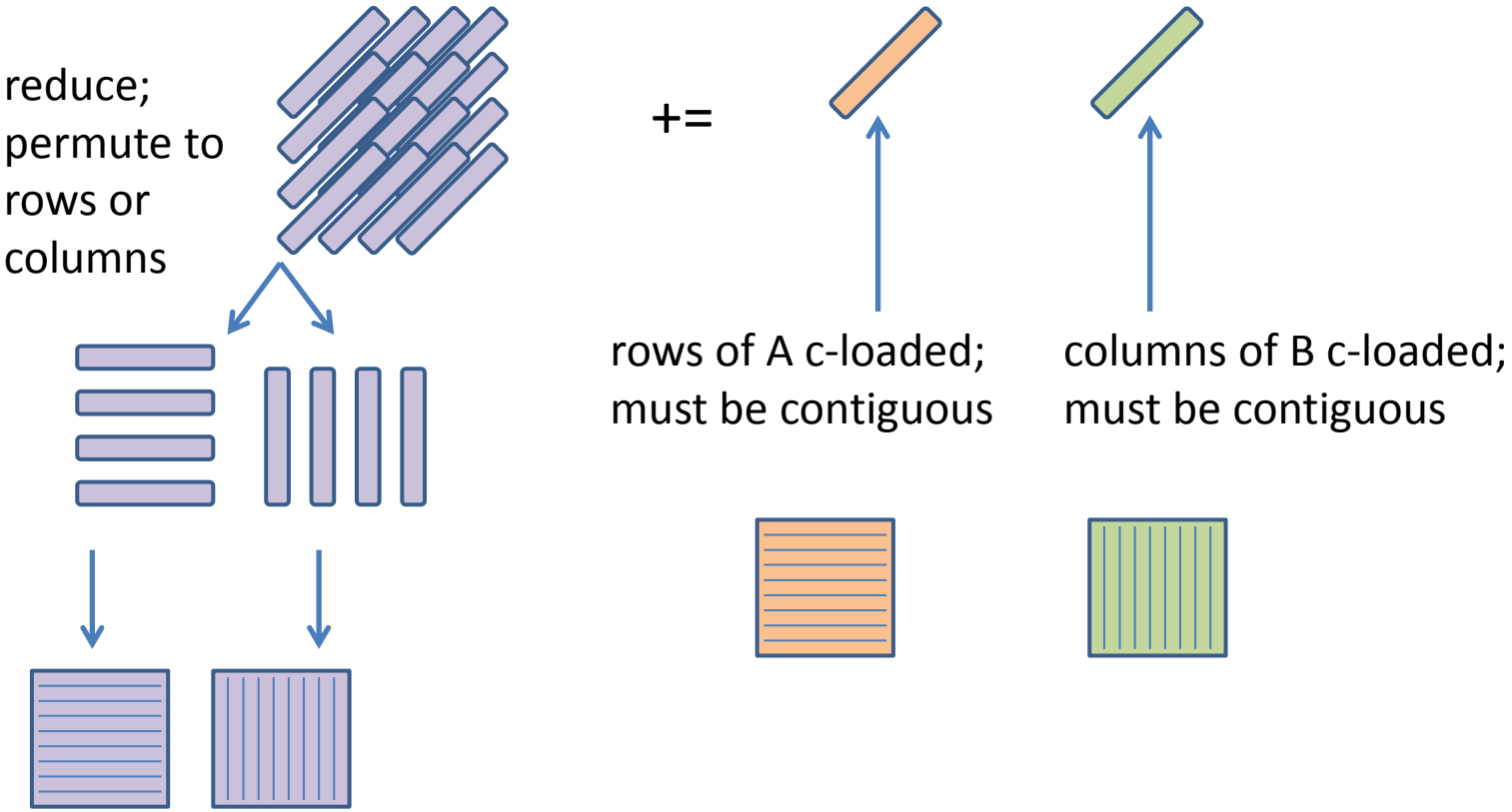
Microkernel implementation

- Column-oriented, contiguous axpy (cca)



Microkernel implementation

- K-oriented, contiguous dot (kcd)

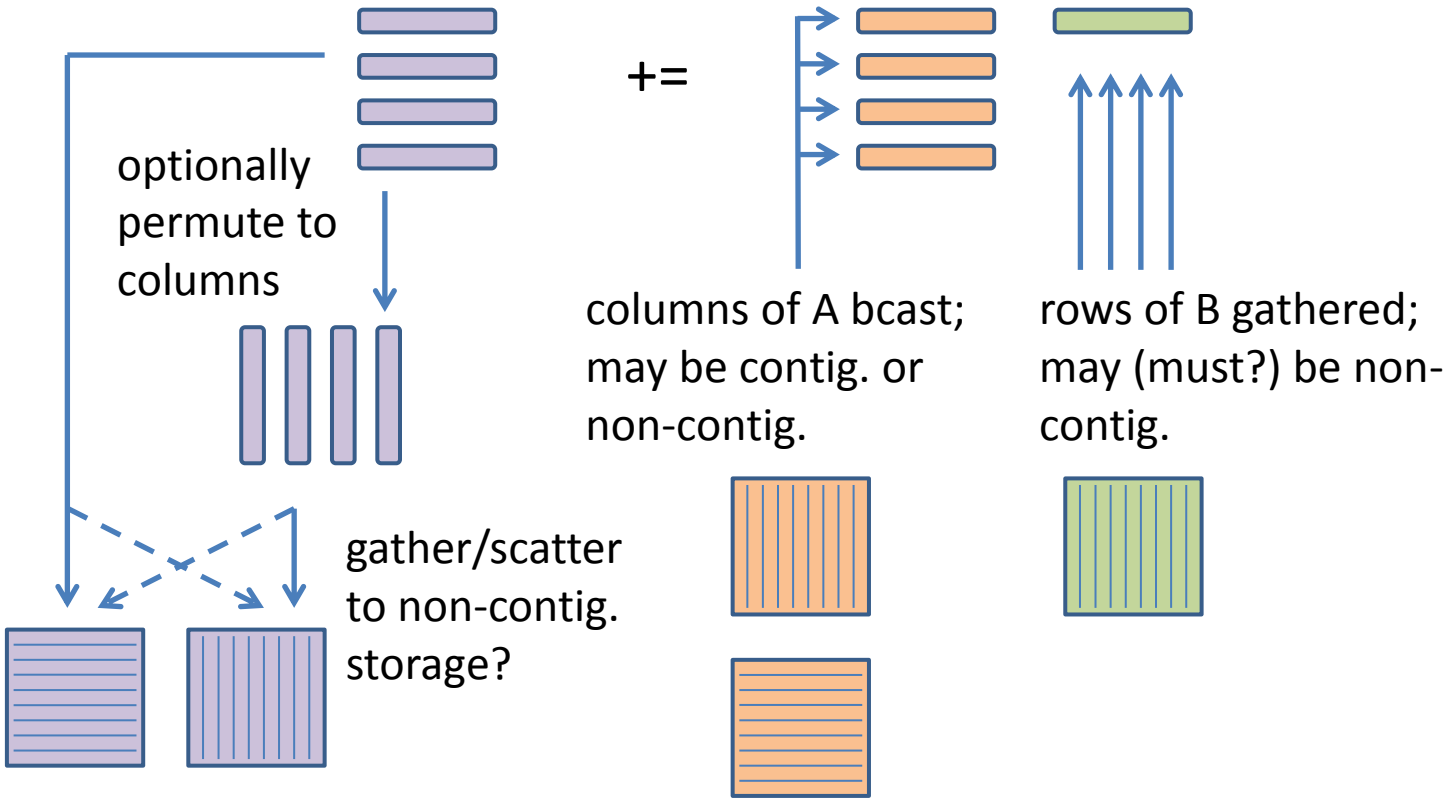


Microkernel implementation

- These three implementation types have bizarro twins that prefer (need?) non-contiguous access
  - Don't know of any existing hardware that meets this criteria, but maybe someday?
  - Notice that this preference for non-contiguous access could affect both input of A and B (gather) *and* input/output on C (gather/scatter)

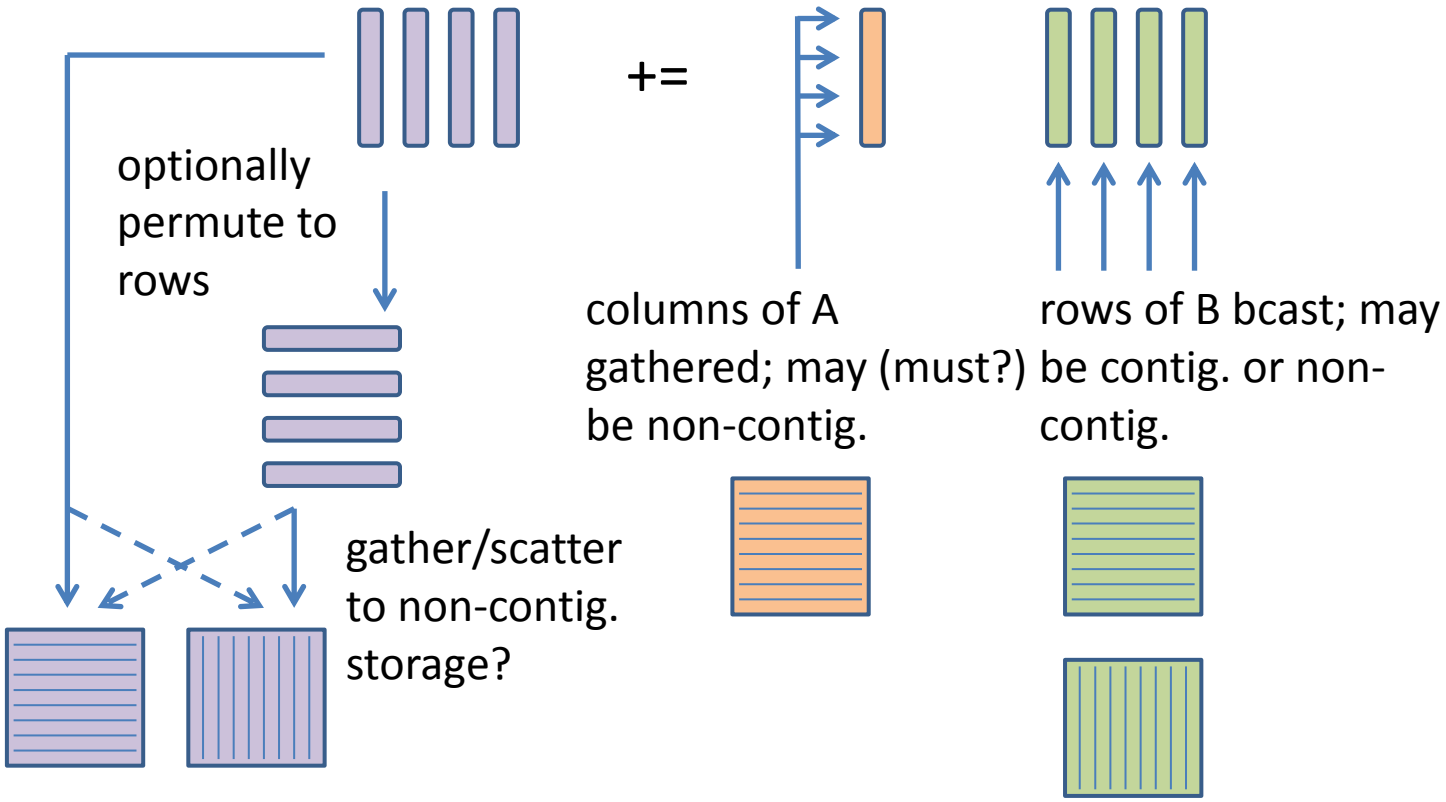
Microkernel implementation

- Row-oriented, non-contiguous axpy (rga)



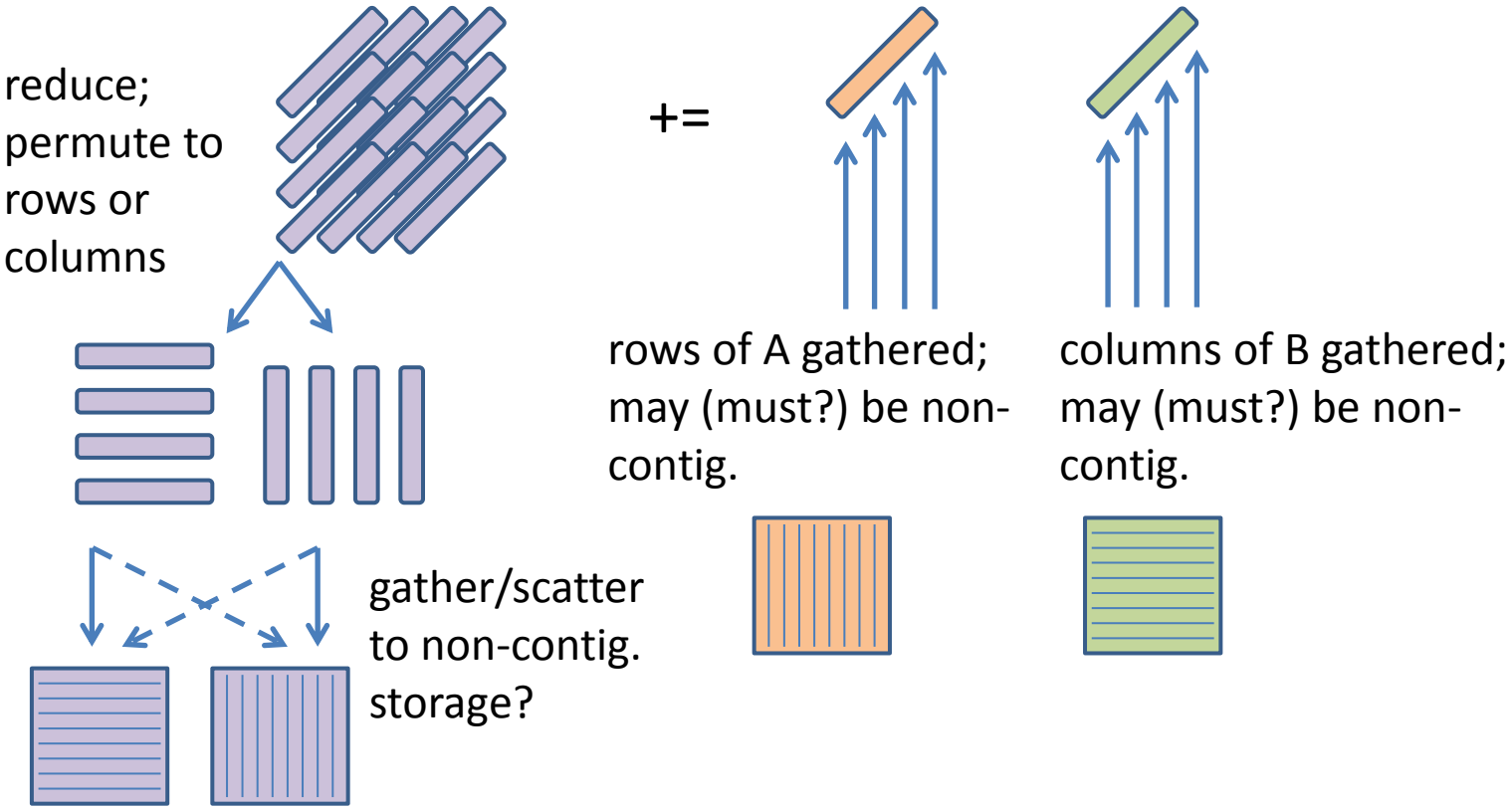
Microkernel implementation

- Column-oriented, non-contiguous axpy (cga)



Microkernel implementation

- K-oriented, non-contiguous dot (kgd)

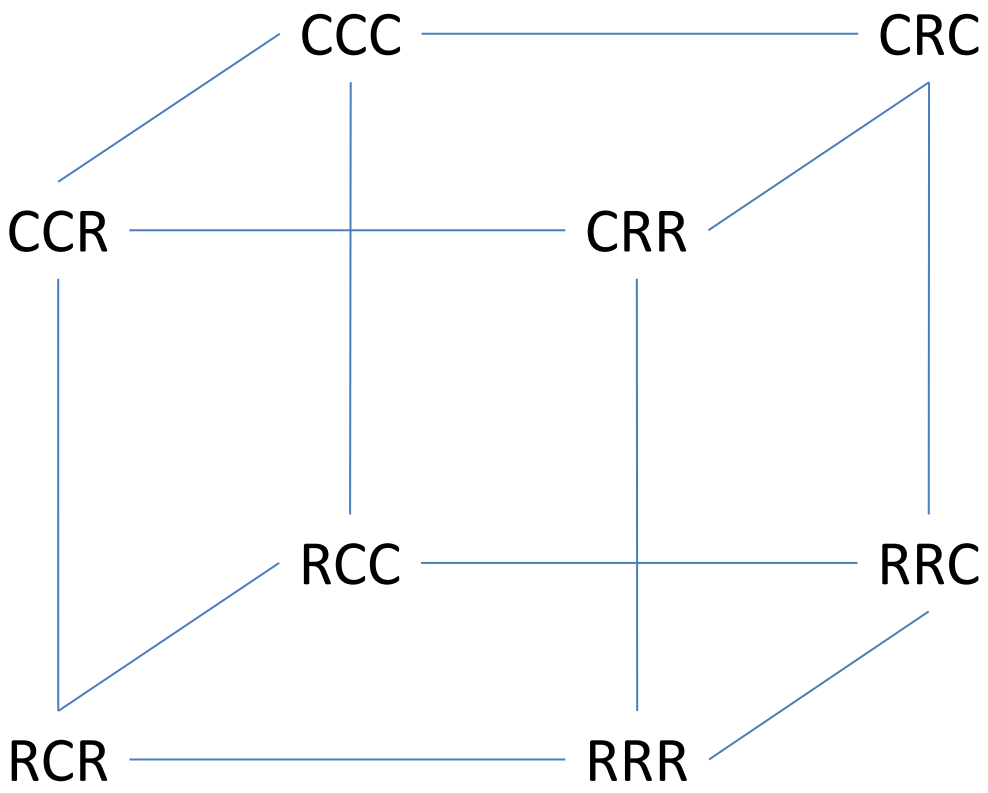


Microkernel implementation

- To summarize…
- Conventional kernel types
  - row/col-oriented, contiguous axpy (rca, cca)
  - k-oriented, contiguous dot (kcd)
- Bizarro twins
  - row/col-oriented, non-contiguous axpy (rga, cga)
  - k-oriented, non-contiguous dot (kgd)

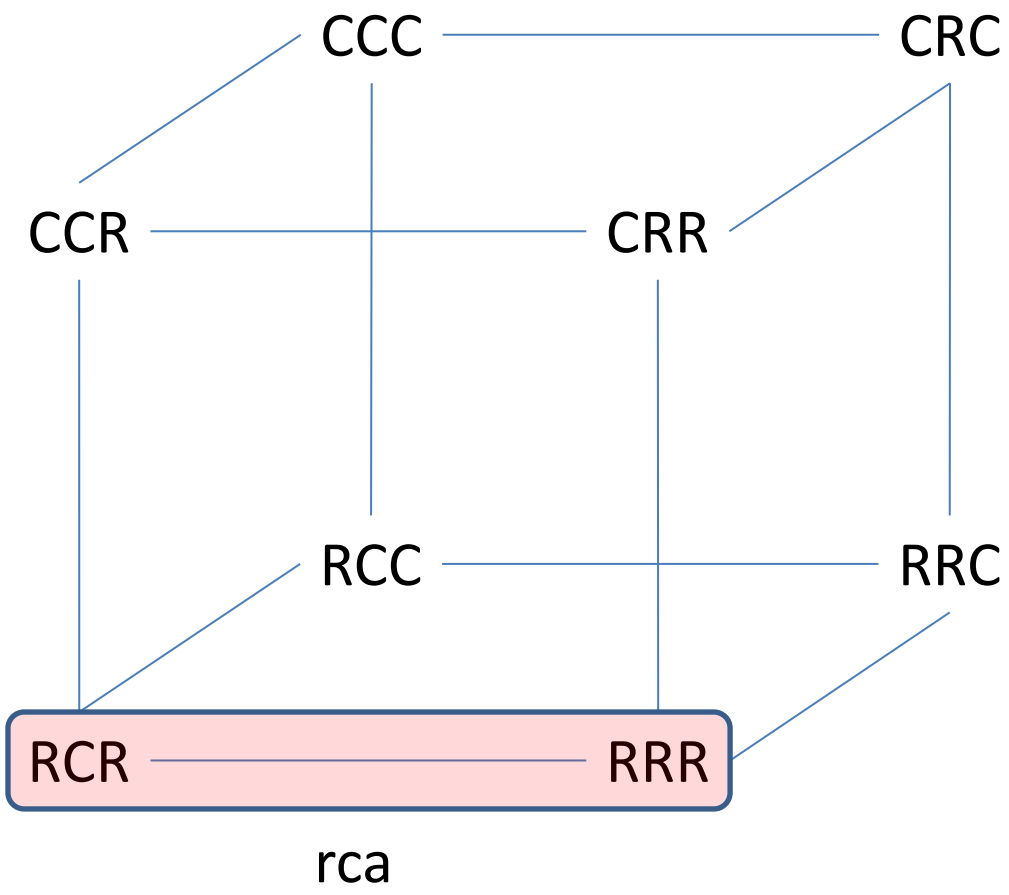
Microkernel application

- Now let's revisit the storage combination cube
  - idea: inspect applicability of each µkernel type



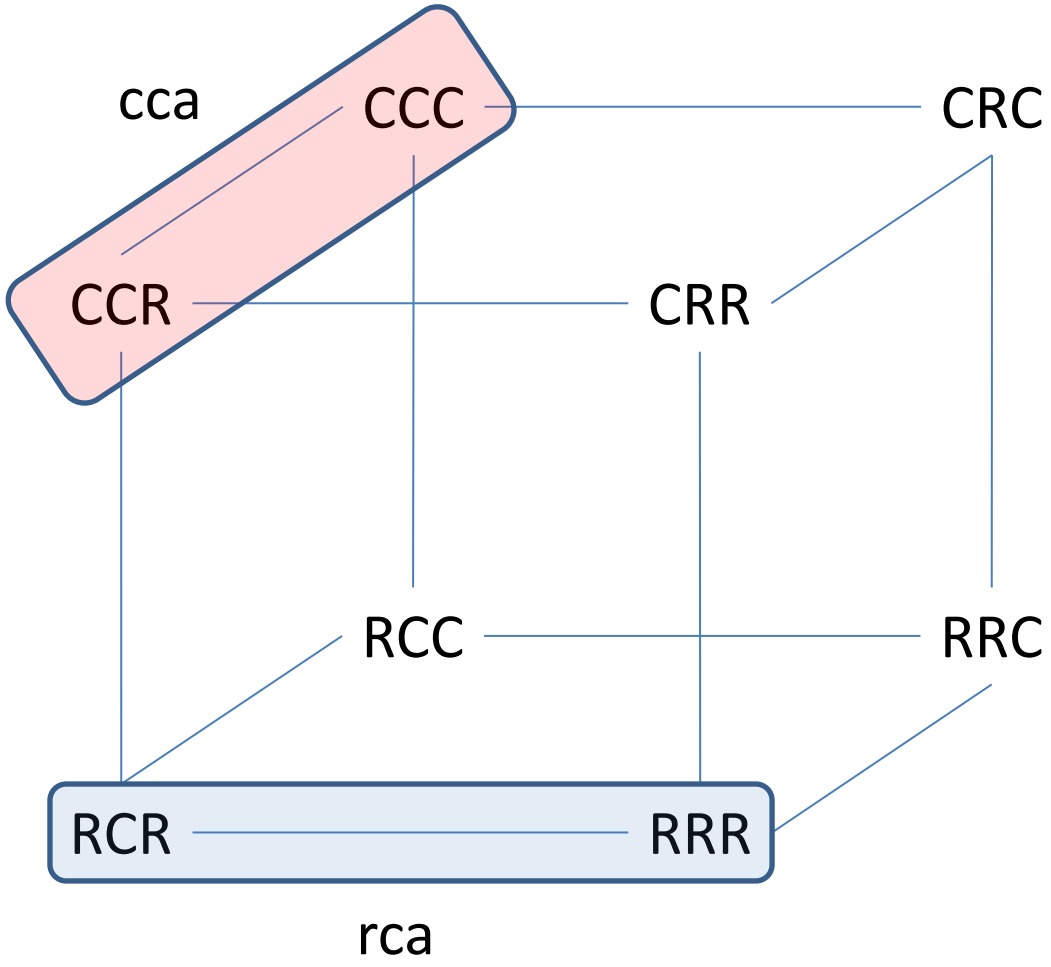
Microkernel application

- Row-oriented contig. axpy (rca)



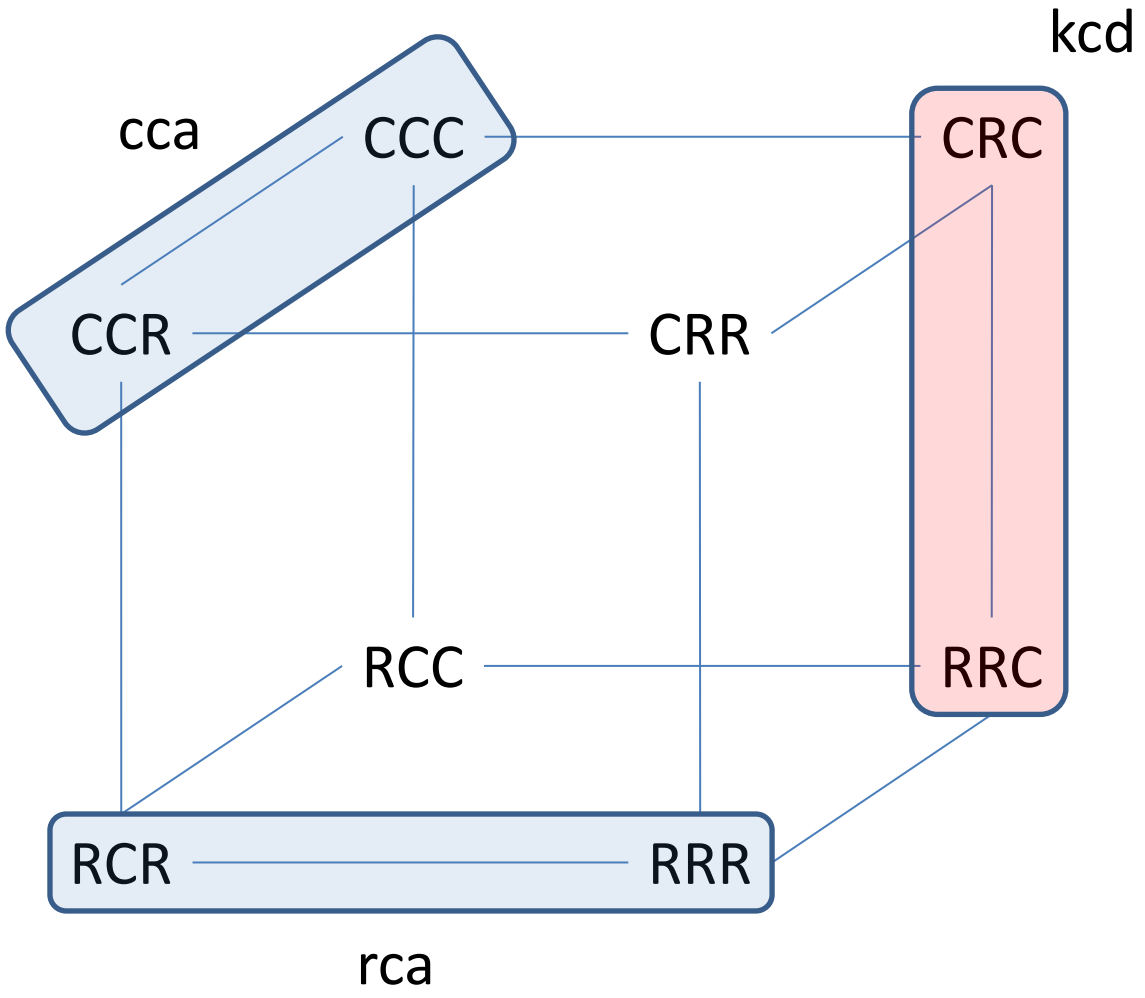
Microkernel application

- Row-oriented contig. axpy (rca)
- Col-oriented contig. axpy (cca)



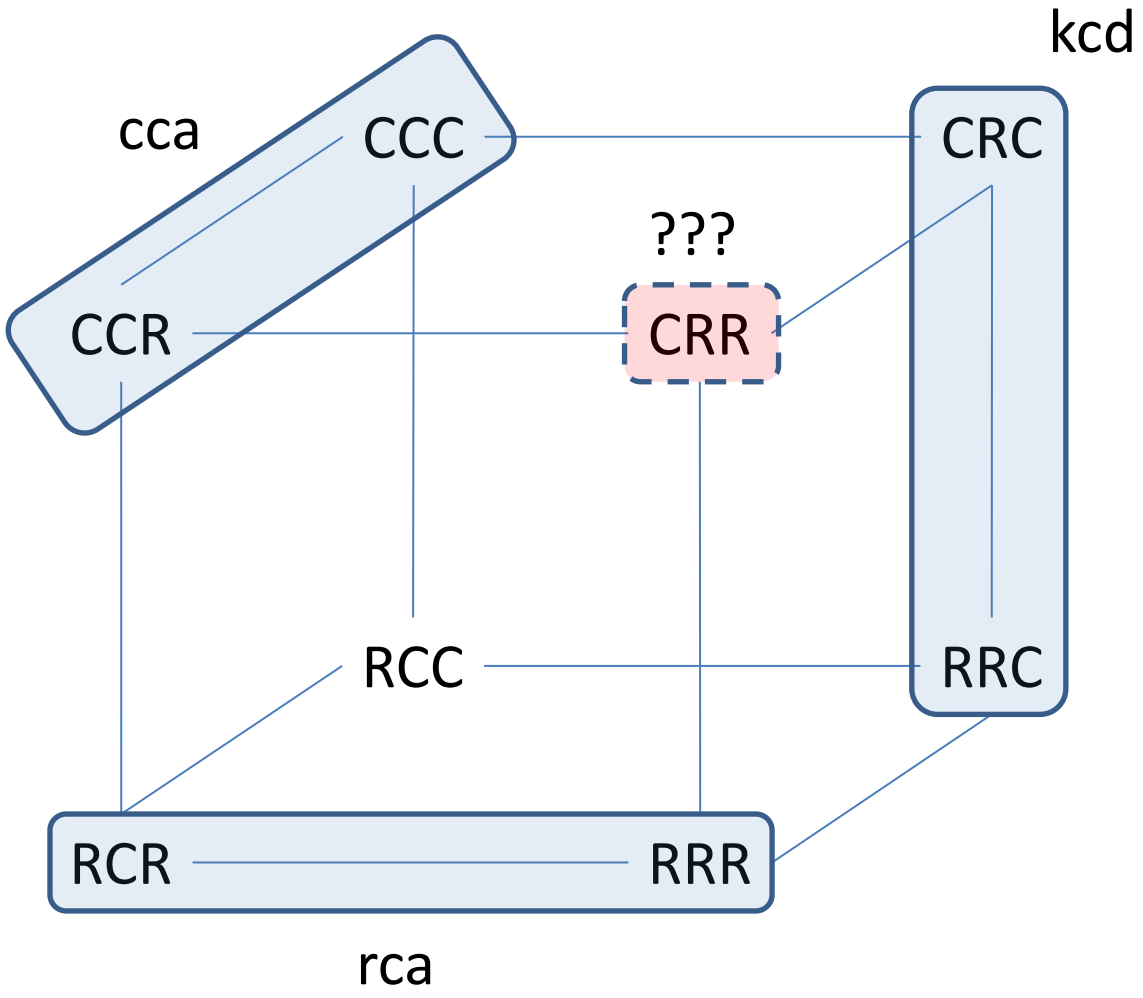
Microkernel application

- Row-oriented contig. axpy (rca)
- Col-oriented contig. axpy (cca)
- K-oriented contig. dot (kcd)



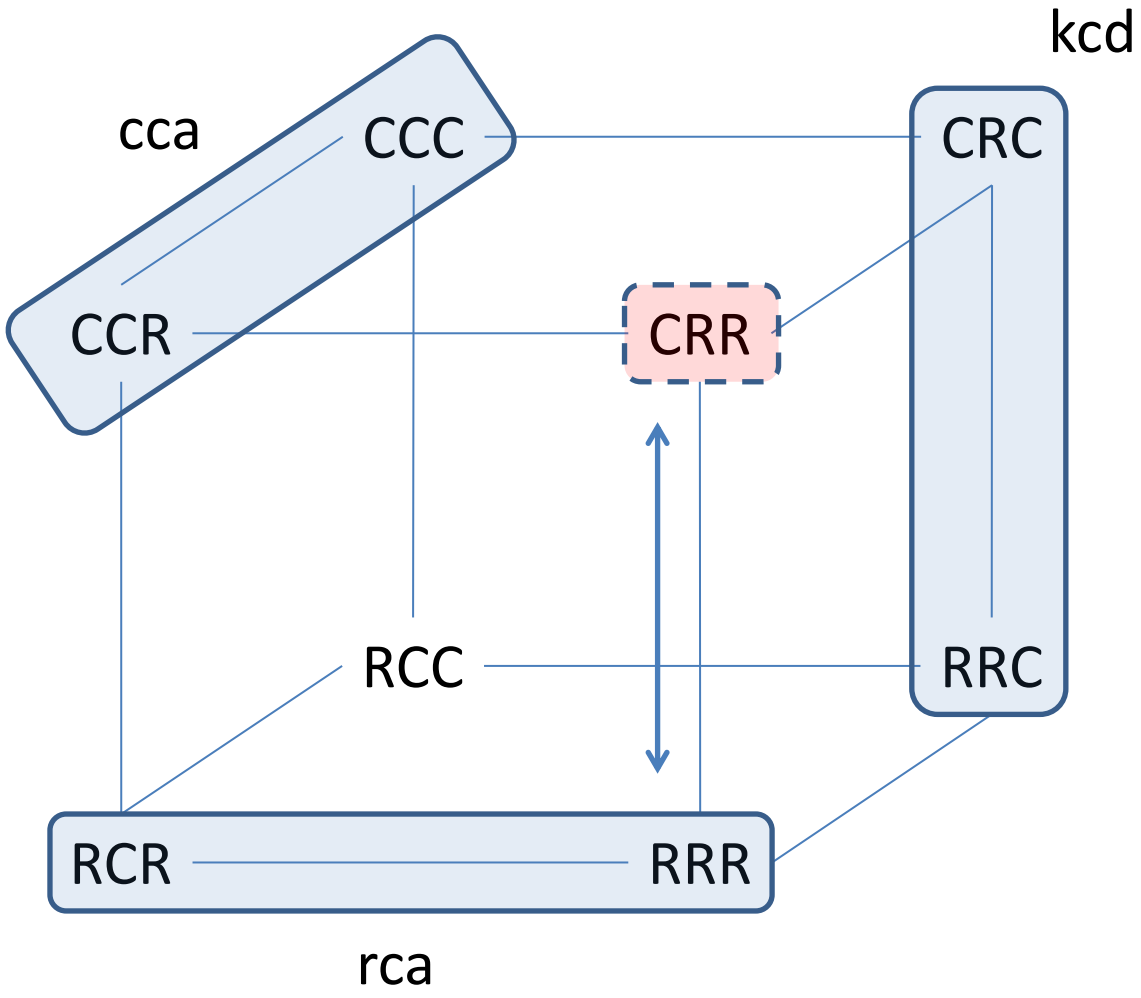
Microkernel application

- How to handle CRR?



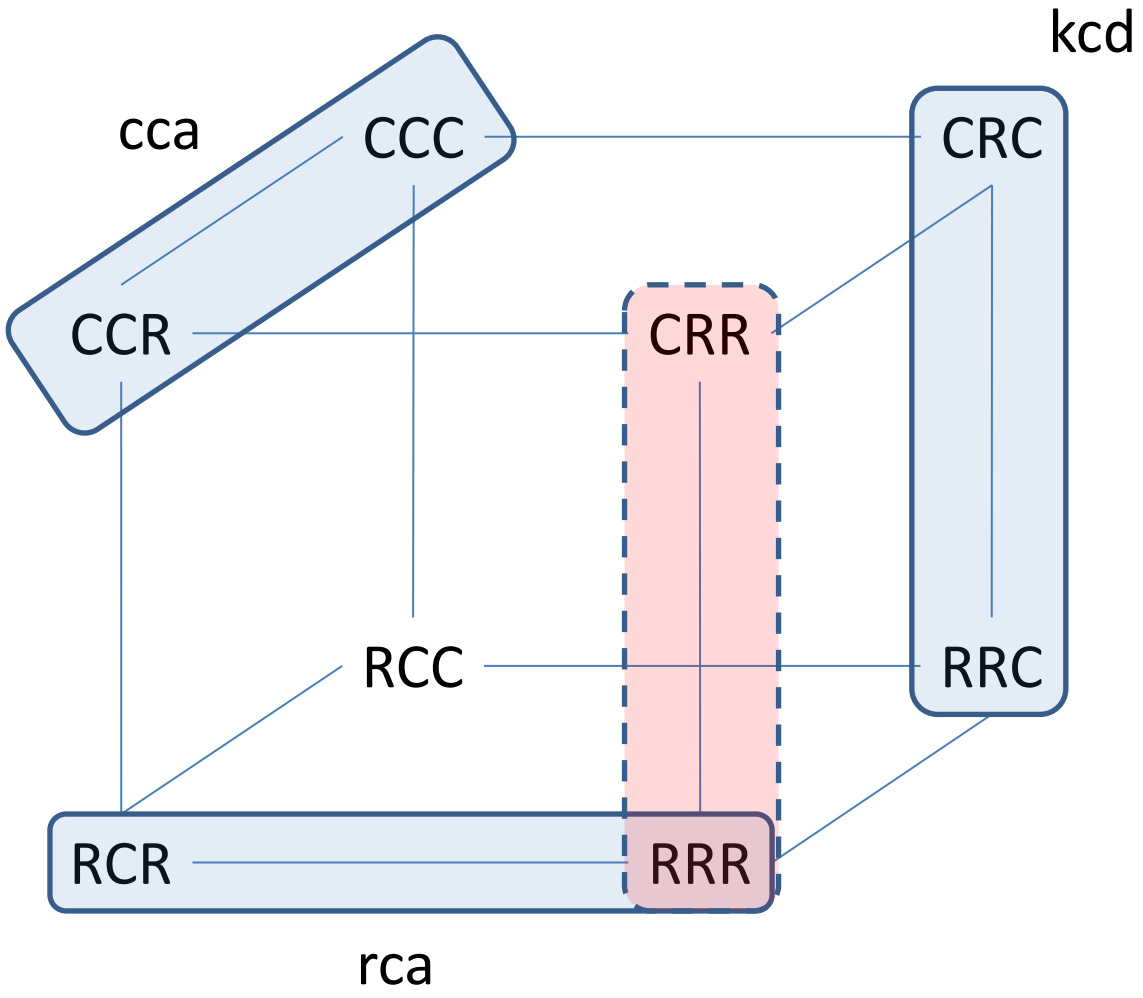
Microkernel application

- How to handle CRR?
  - CRR = RRR + in-register transpose of microtile of matrix C
- What does this mean?



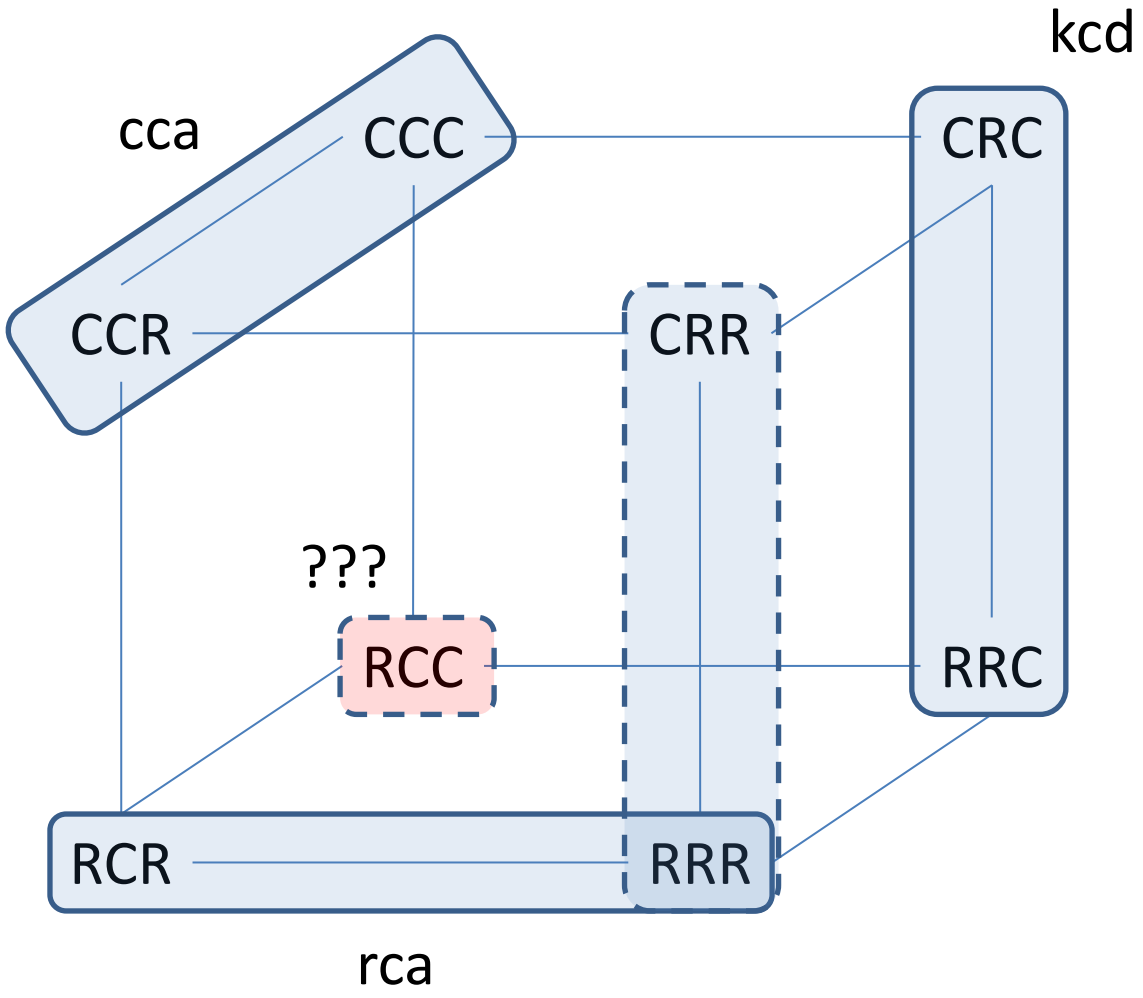
Microkernel application

- How to handle CRR?
  - CRR = RRR + in-register transpose of microtile of matrix C
- This means we can use the rca kernel



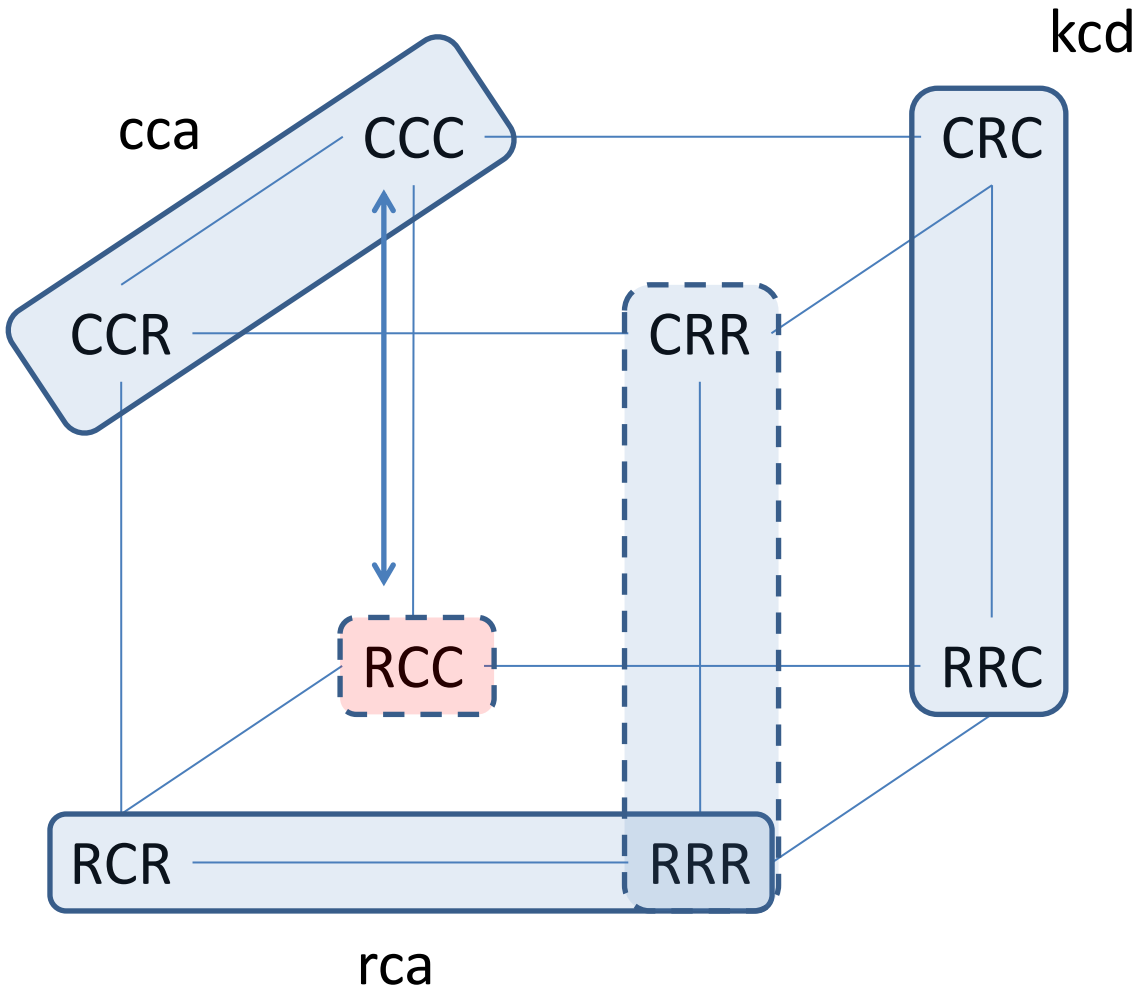
Microkernel application

- How to handle RCC?



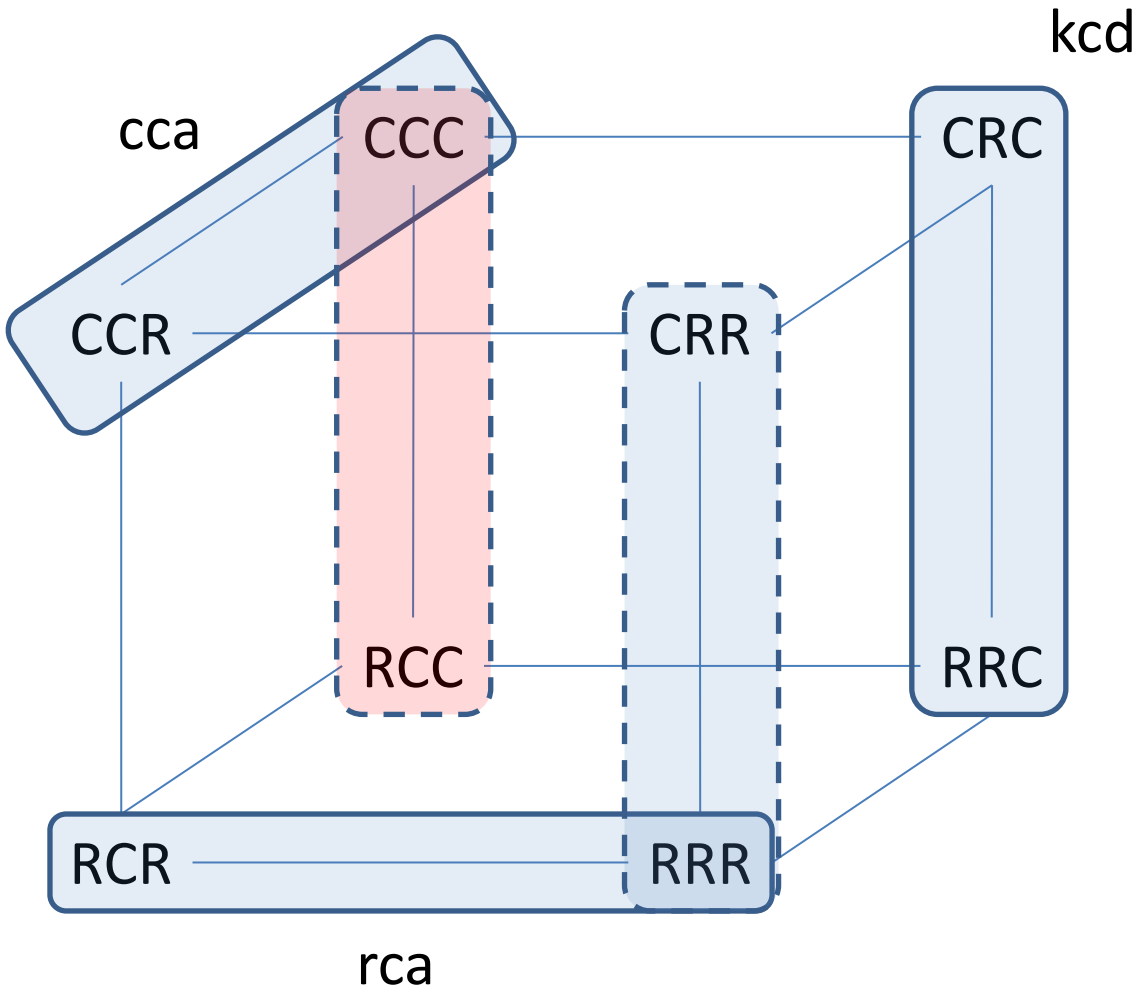
Microkernel application

- How to handle RCC?
  - RCC = CCC + in-register transpose of microtile of matrix C
- What does this mean?



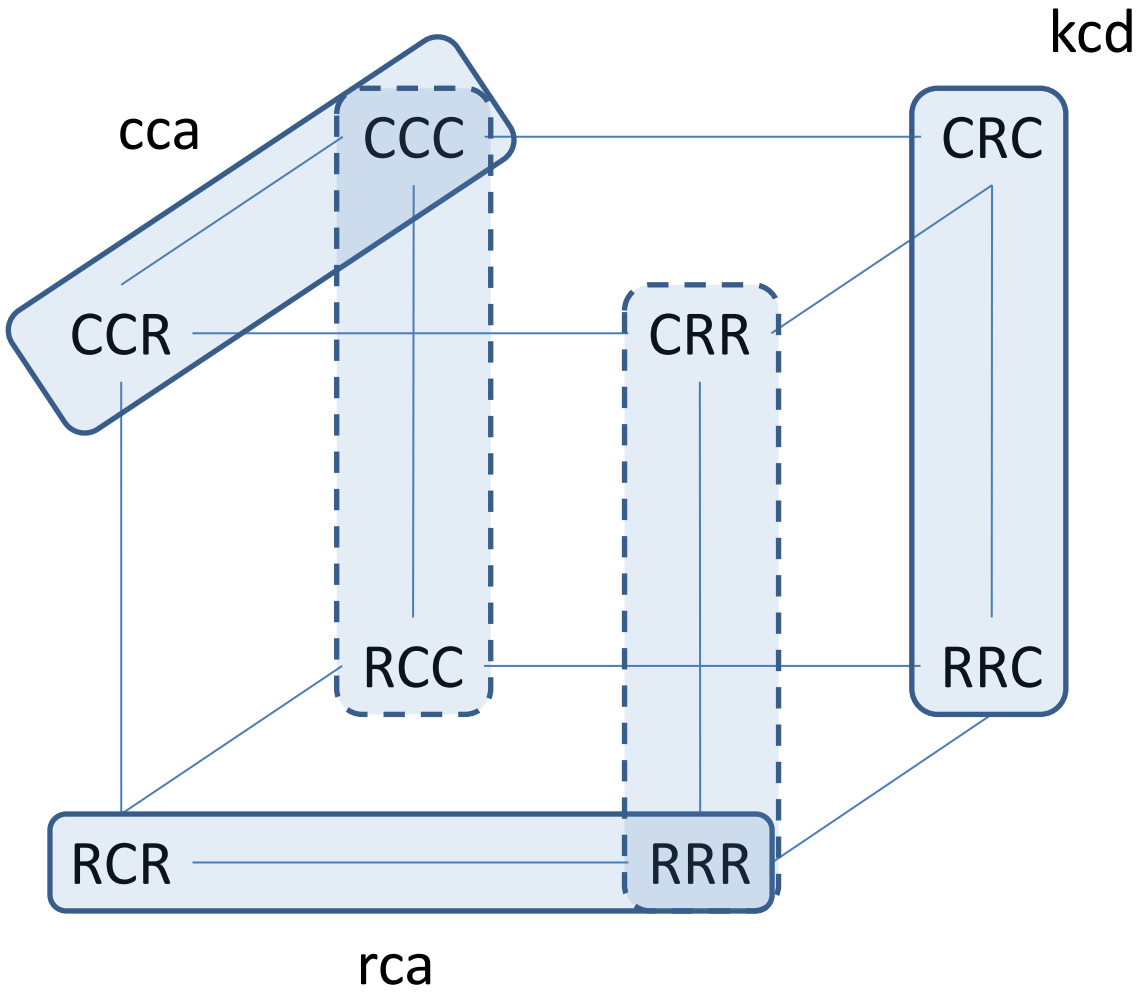
Microkernel application

- How to handle RCC?
  - RCC = CCC + in-register transpose of microtile of matrix C
- This means we can use the cca kernel



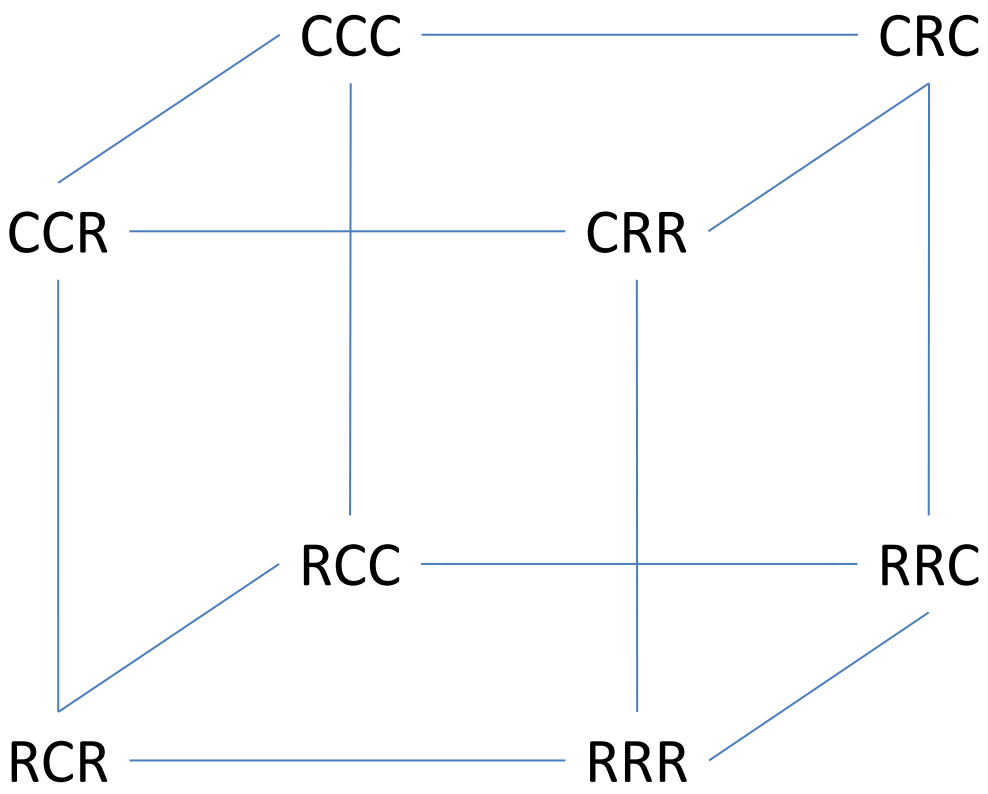
Microkernel application

- What did we just do?
  - Provided support for all combinations of storage (and transposition) with three kernels: rca, cca, kcd



Microkernel application

- Exercise for the audience
  - Repeat this analysis for the bizarro kernels!



Edge case handling

- Consider a basic double-precision real gemm microkernel for Haswell/Zen and newer
  - MR = 6 NR = 8
  - “Interior” case will be 6x8
  - Edge cases will be… smaller
  - Let’s consider only variations in MR < 6 for now
  - So, how do we handle these edge cases?

Edge case handling

- So, how do we handle edge cases (MR < 6; NR = 8)?
- Option 0: Copy A, B, and C to temporary storage and then use BLIS's current edge case strategy
  - "You seem to be avoiding your problems. Tell me about your childhood."
  - This is probably never advantageous, though we haven't investigated it yet
- Option 1: Use reference code
  - 6x8 may be fast, but all edge cases (5x8, 4x8, 3x8, 2x8, 1x8) will be very slow
  - This largely defeats the purpose of targeting skinny matrices

Edge case handling

- So, how do we handle edge cases (MR < 6; NR = 8)?
- Option 2: Implement/combine kernels for powers of 2
  - Implement only 4x8, 2x8, 1x8 and combine as needed. Much faster than reference, but 5x8 and 3x8 will suffer from redundant function call, integer typecasting overhead
- Option 3: Implement all edge kernels
  - All cases are fast, but requires writing full slate of kernels (5x8, 4x8, 3x8, 2x8, 1x8)

Edge case handling

- Also, we'll need to fill in the whole grid of kernel types (both MR and NR dimensions)
- For example, assuming we choose Option 2:
  - 6x8, 6x4, 6x2, 6x1
  - 4x8, 4x4, 4x2, 4x1
  - 2x8, 2x4, 2x2, 2x1
  - 1x8, 1x4, 1x2, 1x1

Edge case handling

- Also, we'll need to fill in the whole grid of kernel types (both MR and NR dimensions)
- For example, assuming we choose Option 2:
  - 6x8, 6x4, 6x2, 6x1
  - 4x8, 4x4, 4x2, 4x1
  - 2x8, 2x4, 2x2, 2x1
  - 1x8, 1x4, 1x2, 1x1
- We may not need to implement 1xN or 1xM kernels since they may not vectorize easily

Odds and ends

Optional: “smart” edge blocking

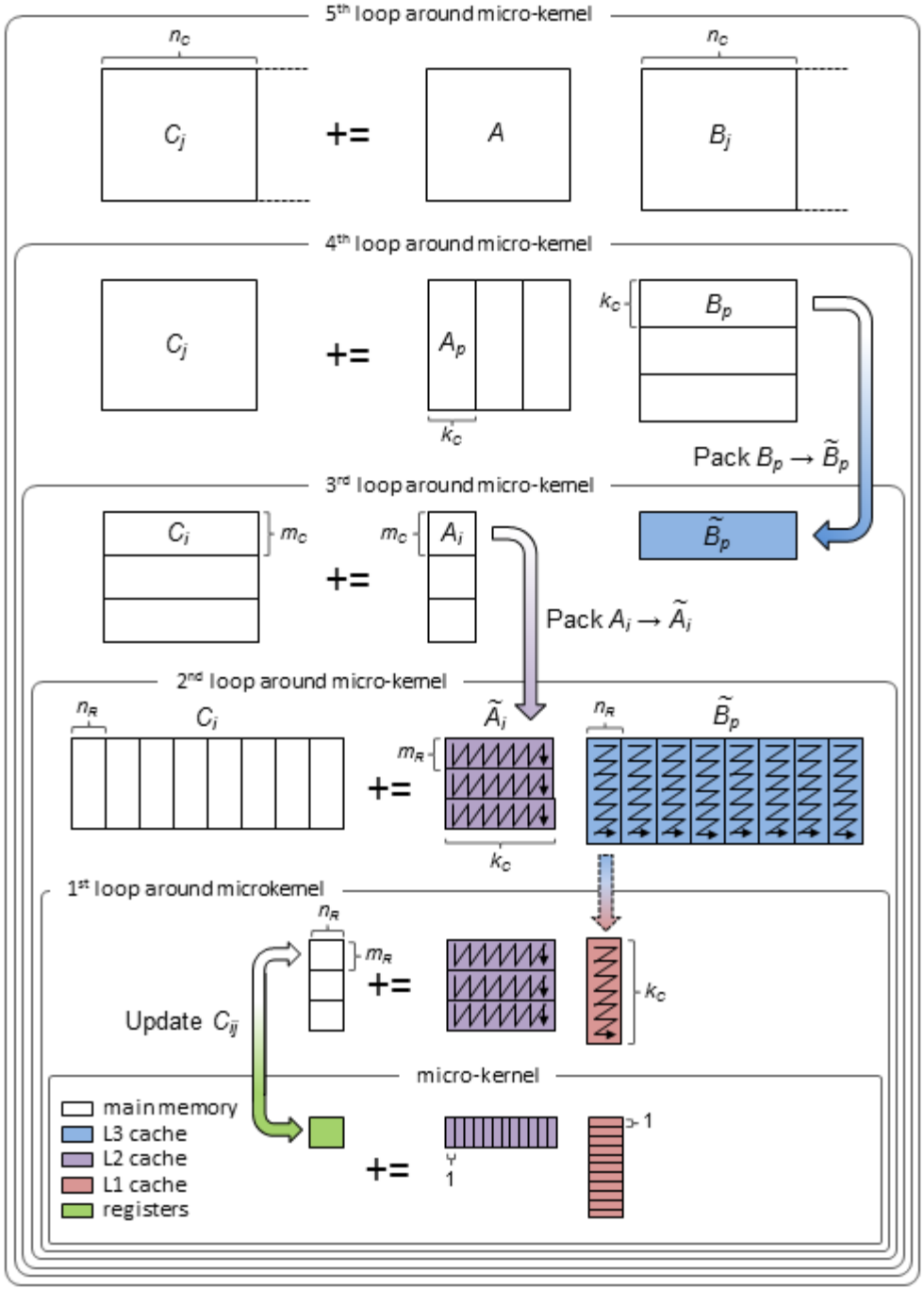
- Consider when m is just a little larger than MR
  - Example: 7x8 (assuming a target microtile of 6x8)
  - Even if we choose Option 3 and implement all edge case kernels, 7x8 decomposes into 6x8 + 1x8
    - Performance drops sharply because of the 1x8 call
  - Turns out 4x8 + 3x8 yields higher aggregate performance

Optional: “smart” edge blocking

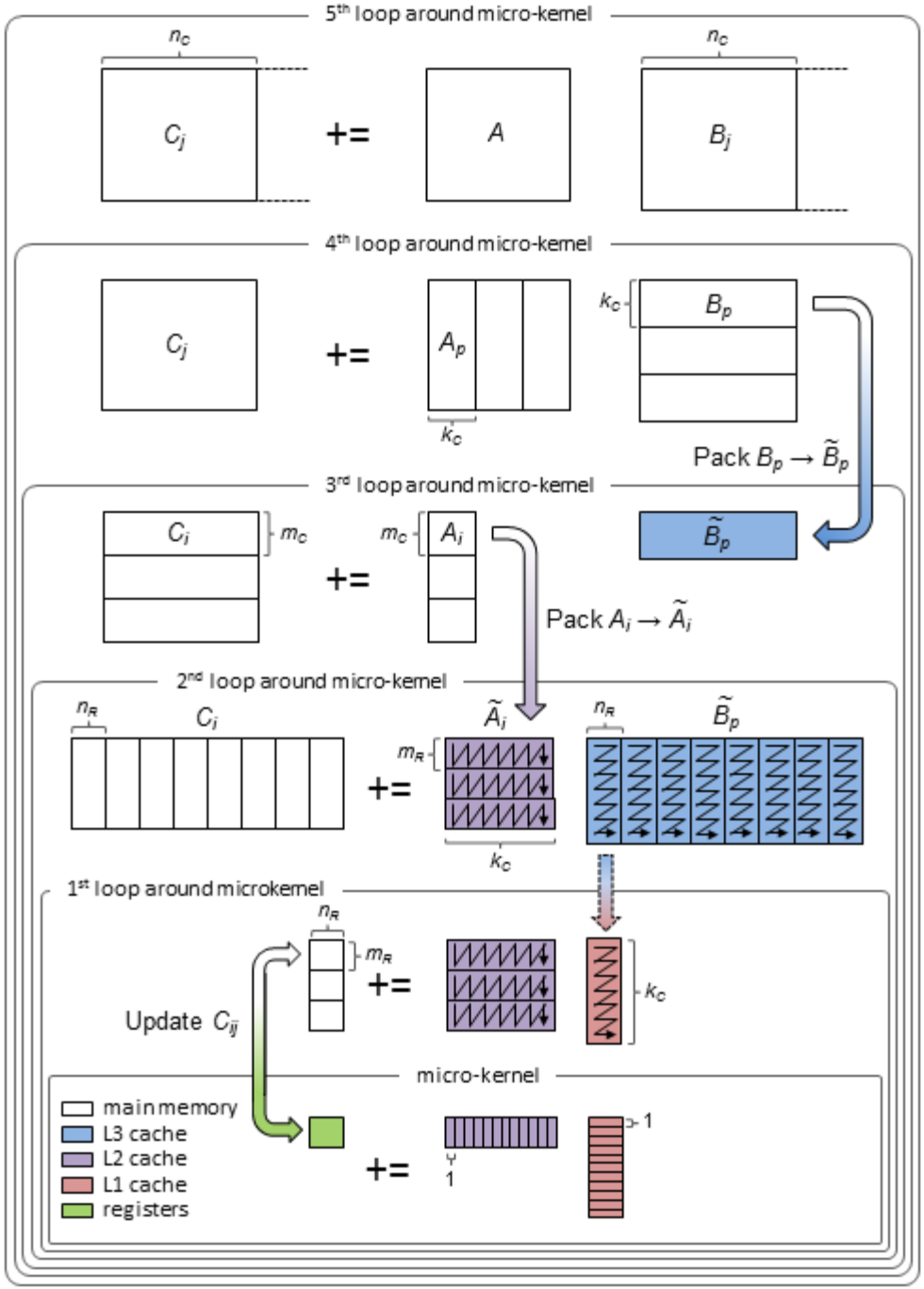
- How do we implement this alternate kernel decomposition?
  - Define a maximum edge case dimension: $ME_{max} = 9$
  - Allow smaller edge cases to be absorbed into the last full “interior” kernel invocation
  - This means the merged problem can be more favorably decomposed into two smaller kernel calls
  - Examples:
    - 9x8: Executed as 5x8 + 4x8 (instead of 6x8 + 3x8)
    - 8x8: Executed as 4x8 + 4x8 (instead of 6x8 + 2x8)
    - 7x8: Executed as 4x8 + 3x8 (instead of 6x8 + 1x8)

Optional: millikernels

- Conventional design
  - microkernel contained within the IR loop



- Conventional design
  - microkernel contained within the IR loop



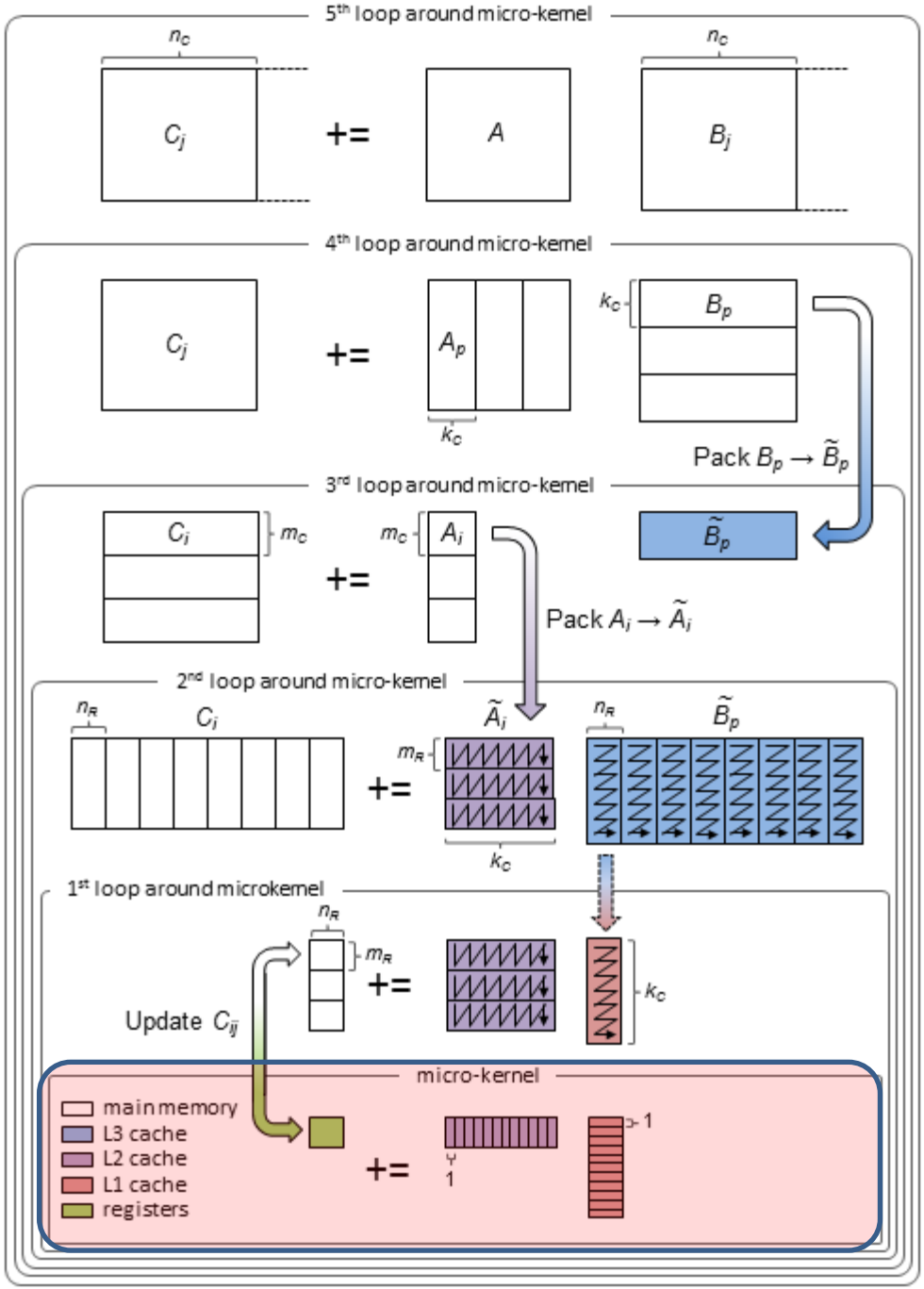
IR loop

microkernel

- Conventional design
  - microkernel contained within the IR loop
  - assembly code highlighted in pink

IR loop

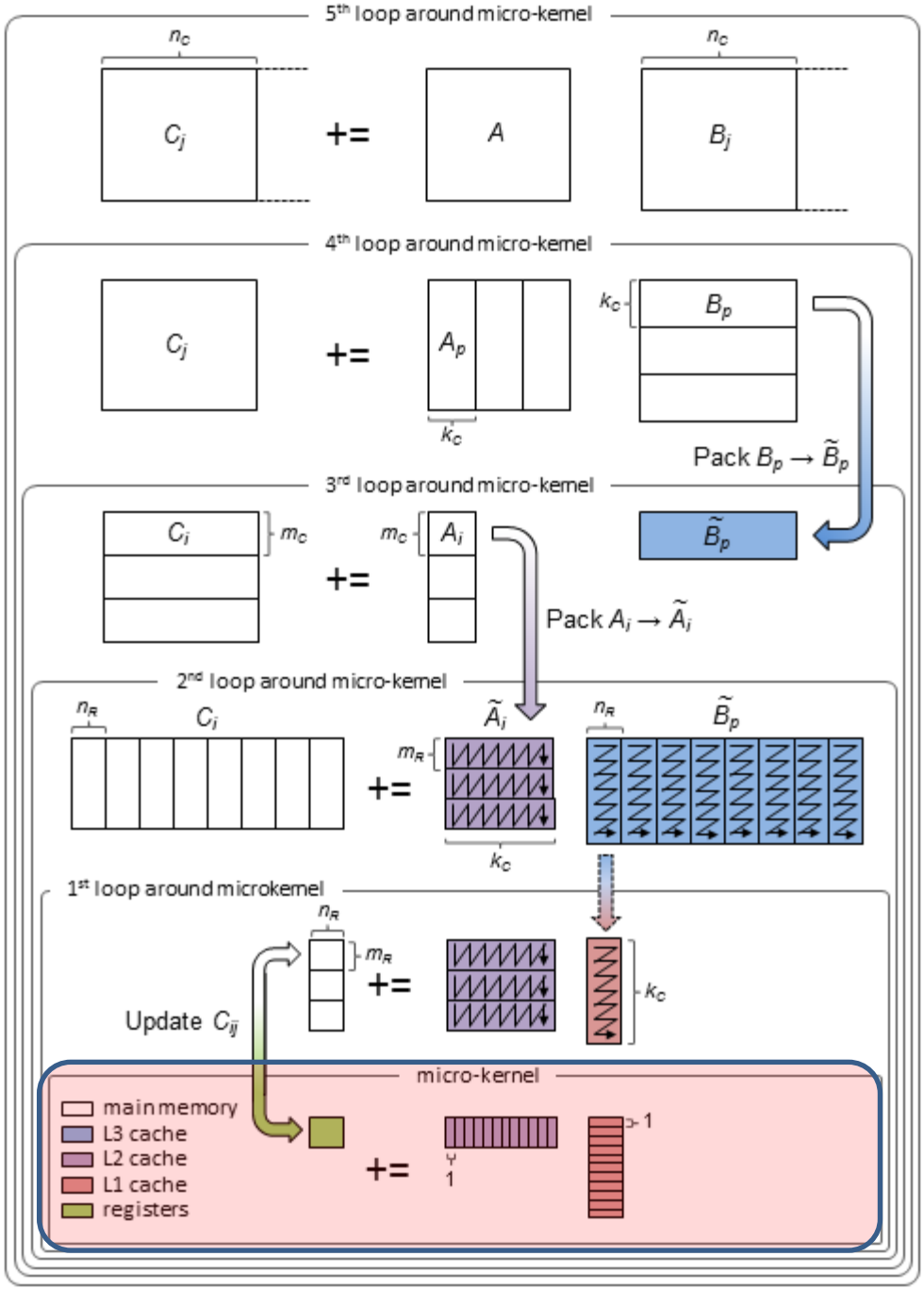
microkernel



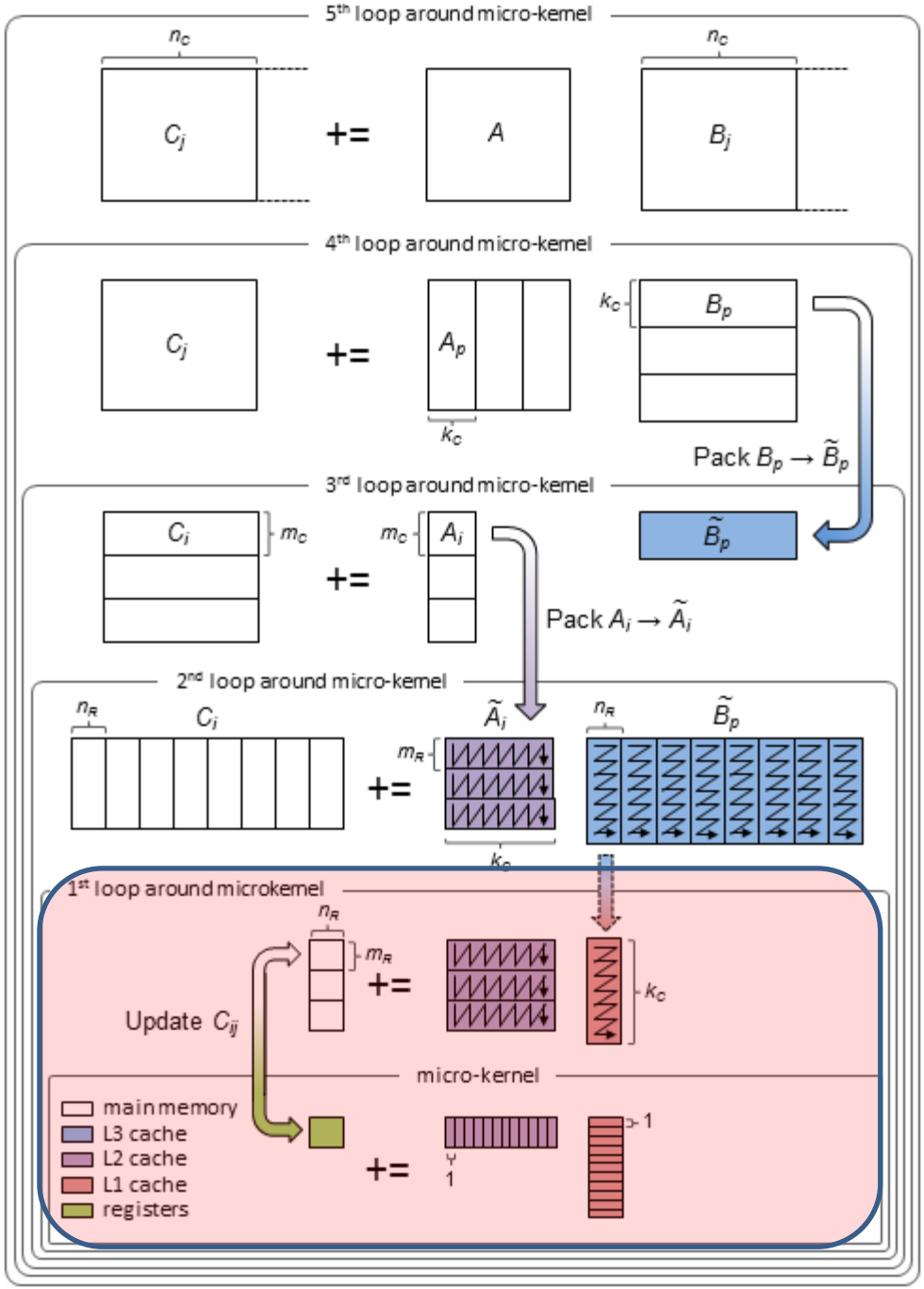
- Conventional design
  - microkernel contained within the IR loop
  - problem? function call + integer casting overhead **per** µkernel call

IR loop →

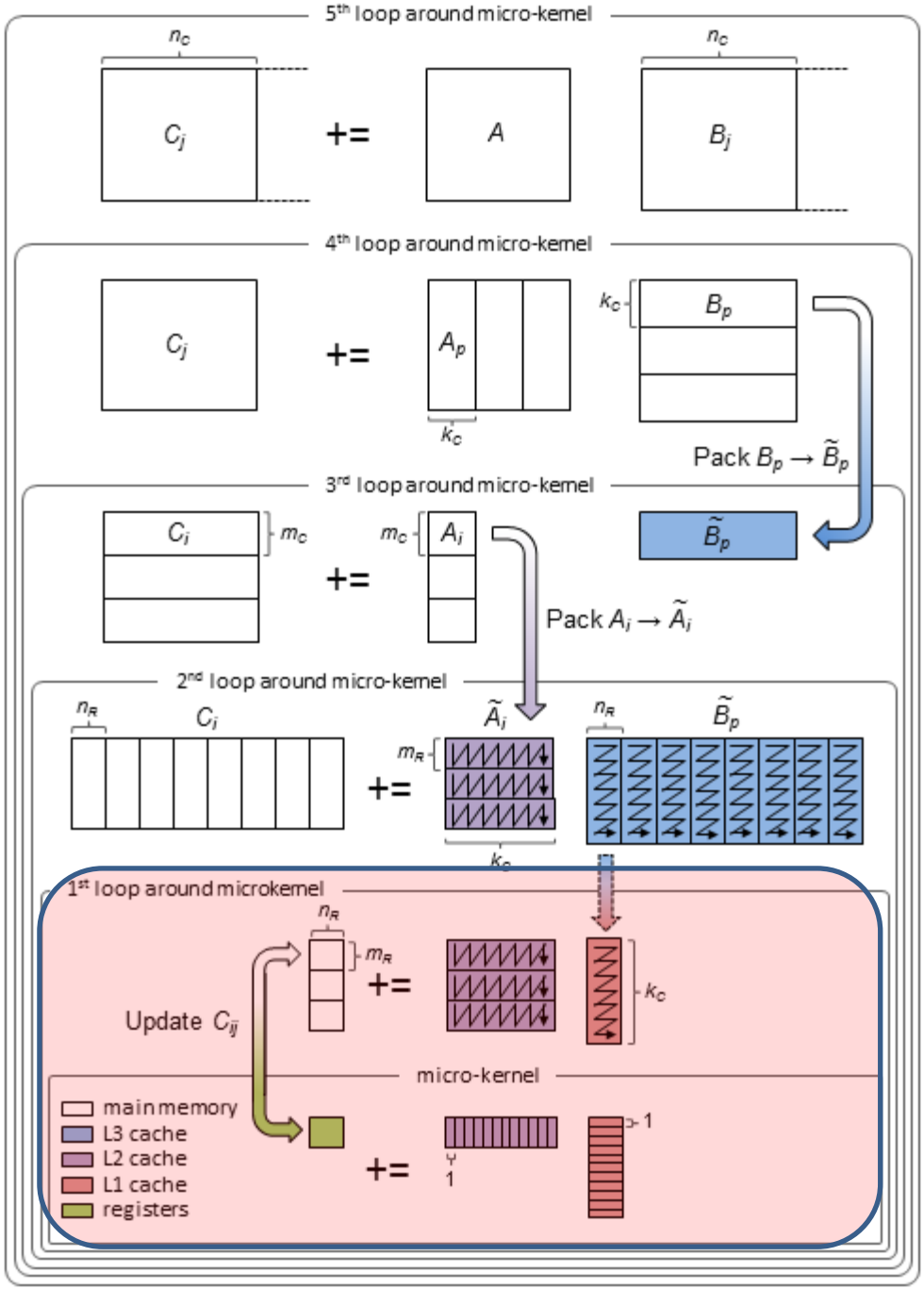
microkernel →



- Conventional design
  - microkernel contained within the IR loop
  - solution? millikernels!



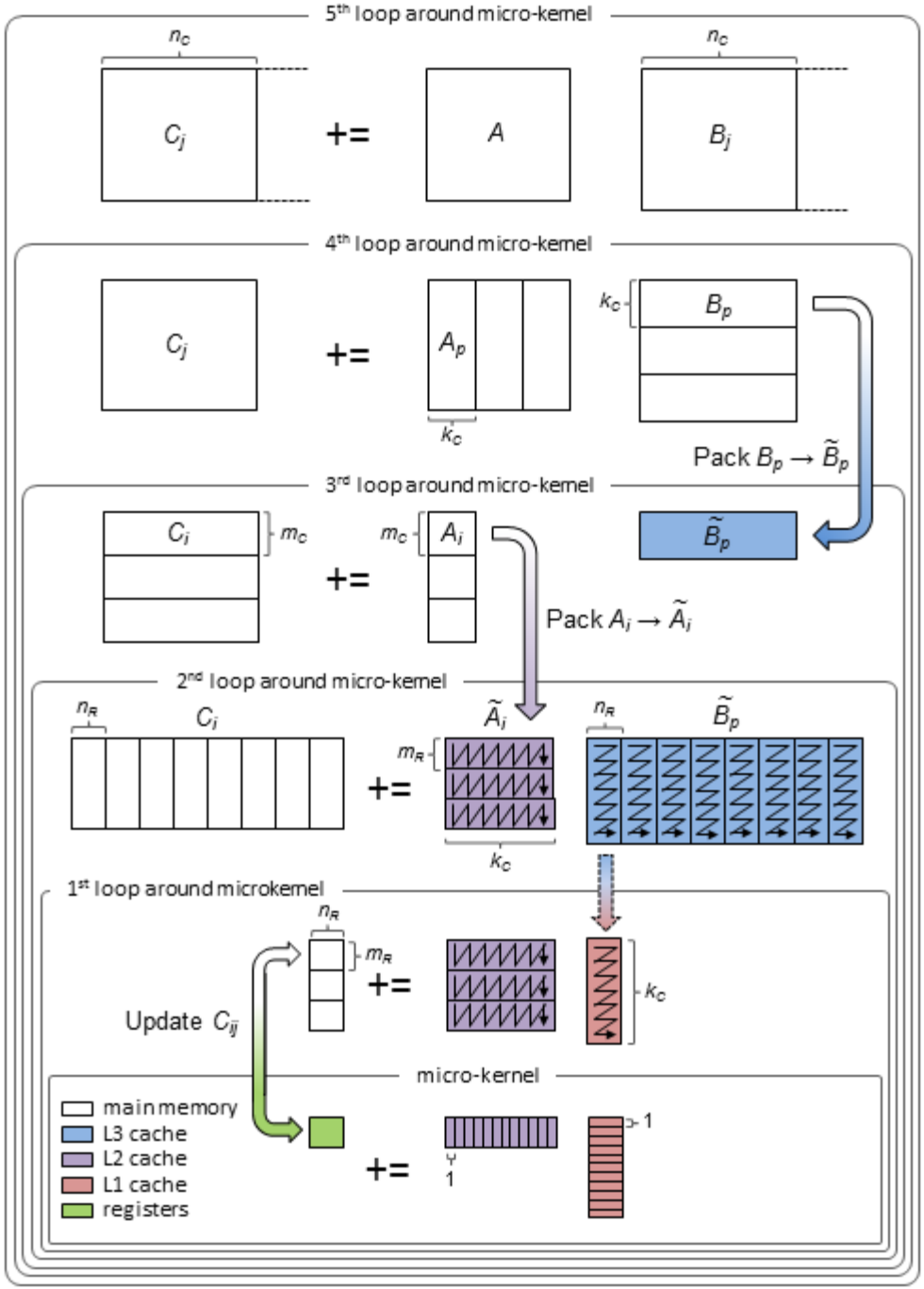
- Conventional design
  - microkernel contained within the IR loop
  - solution? millikernels!
  - Reduces function call, integer casting overhead by factor of m/MR (or n/MR… yes, I mean MR)



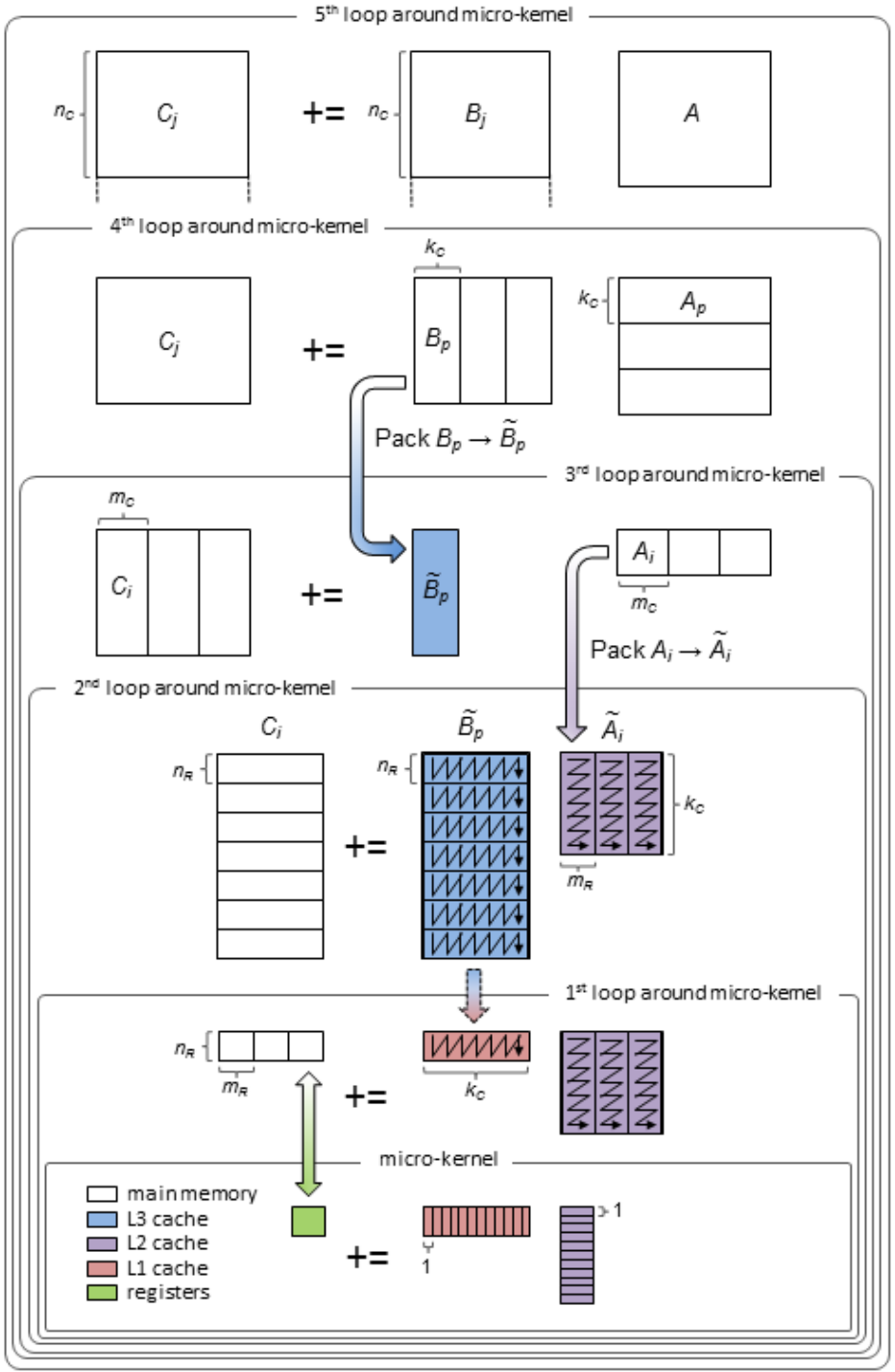
Algorithmic loop structure

- What higher level loops do we use around these kernels?

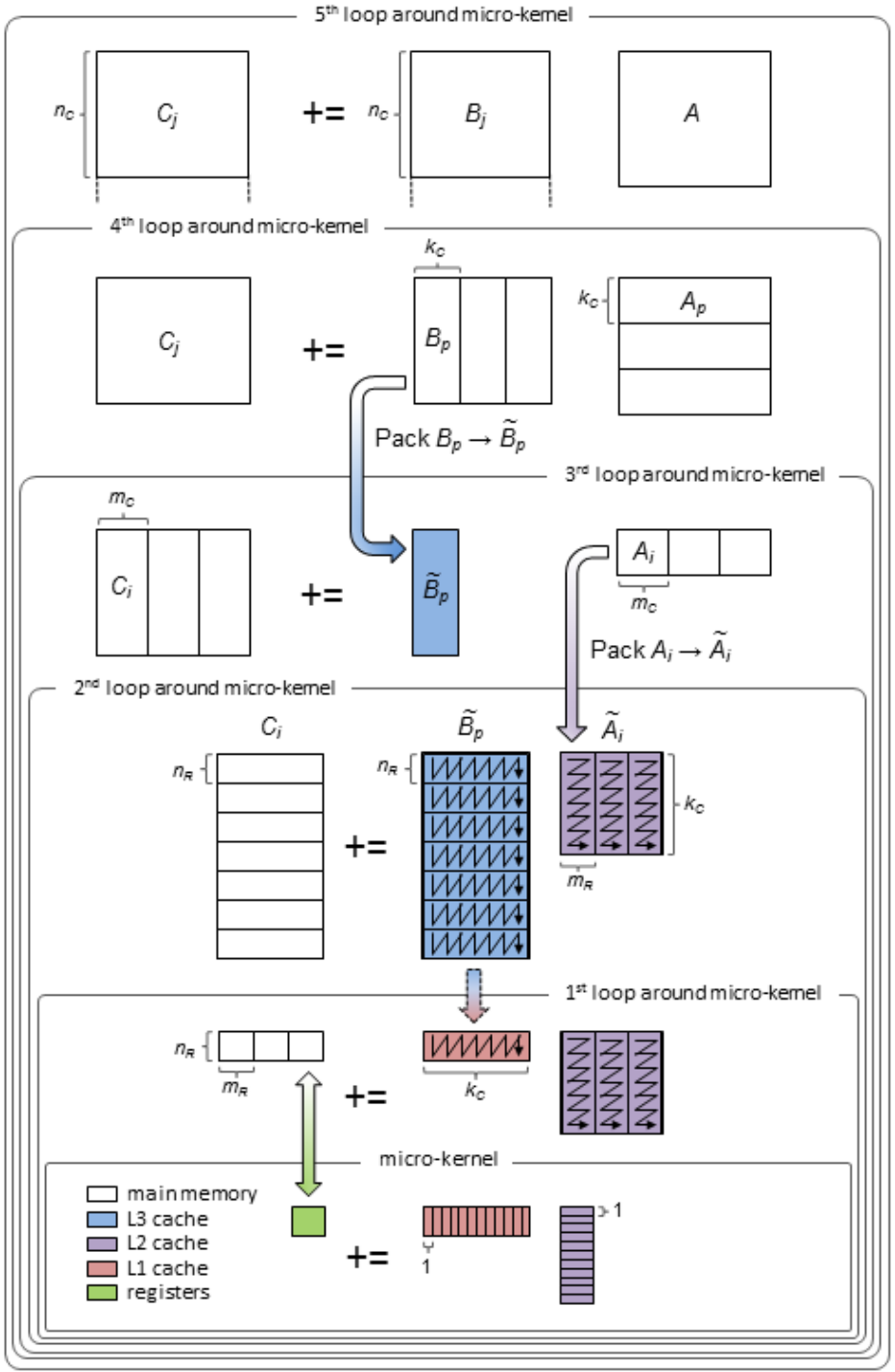
- We use classic block-panel algorithm



- We use classic block-panel algorithm
- And its block-panel counterpart!



- We use classic block-panel algorithm
- And its block-panel counterpart!
  - Recycles the same MRxNR µkernel, which may or may not be called with an induced transposition (NRxMR)



Thresholds & Handlers

- When do we switch between “skinny” and “large” code paths?
  - MT, NT, KT thresholds per datatype. If **any** dimension (m, n, k) is less than than its respective threshold (MT, NT, KT), skinny implementation is called
  - **However**, the skinny implementation “handler” can perform further heuristics and reject the problem
    - If rejected, execution returns to the “large” code path
  - Handler can be thought of as the high-level entry point for the skinny code path
  - Thresholds and handlers set per subconfiguration

Skinny implementation status

- Introduced on April 27, 2019
  - Core changes: b9c9f035
- Currently implemented for double-precision real gemm only
  - Intel: Haswell, Broadwell, Skylake, Kaby Lake, Coffee Lake, etc. (AVX2 + FMA)
  - AMD: Zen1, Zen1+, Zen2
- Currently single-threaded only
  - Multithreaded is possible, but not yet implemented

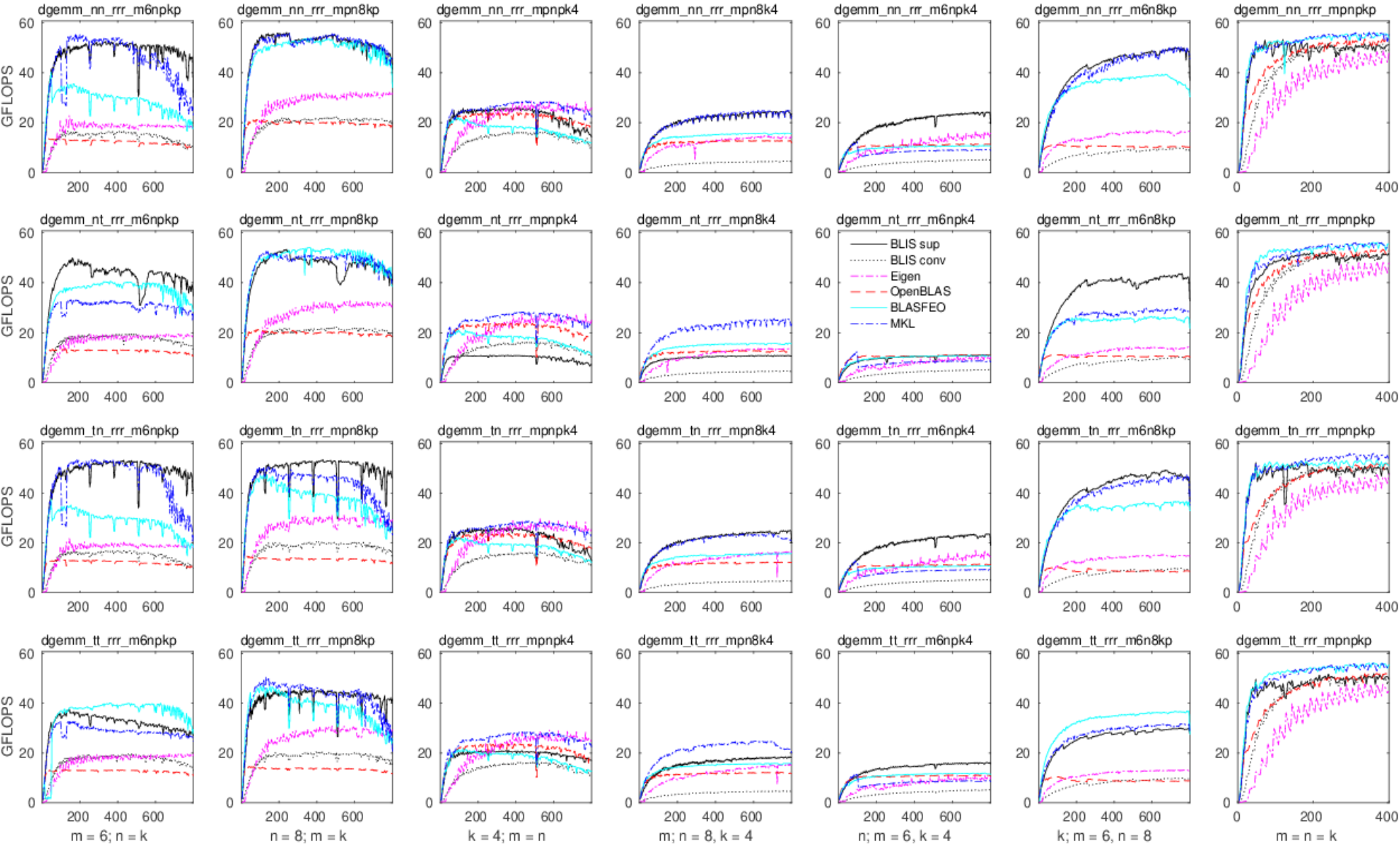
Performance

- Added PerformanceSmall.md document to 'docs' directory
  - Skinny performance results currently shown for:
    - Haswell
    - Kaby Lake
    - Zen1 (Epyc)
  - Results shown for
    - Four transA/B cases: NN, NT, TN, TT
    - Two storage combinations: RRR and CCC
      - CBLAS API allows RRR or CCC, but no mixing of formats
    - Seven shape scenarios: SLL, LSL, LLS, SSL, SLS, SSL, LLL

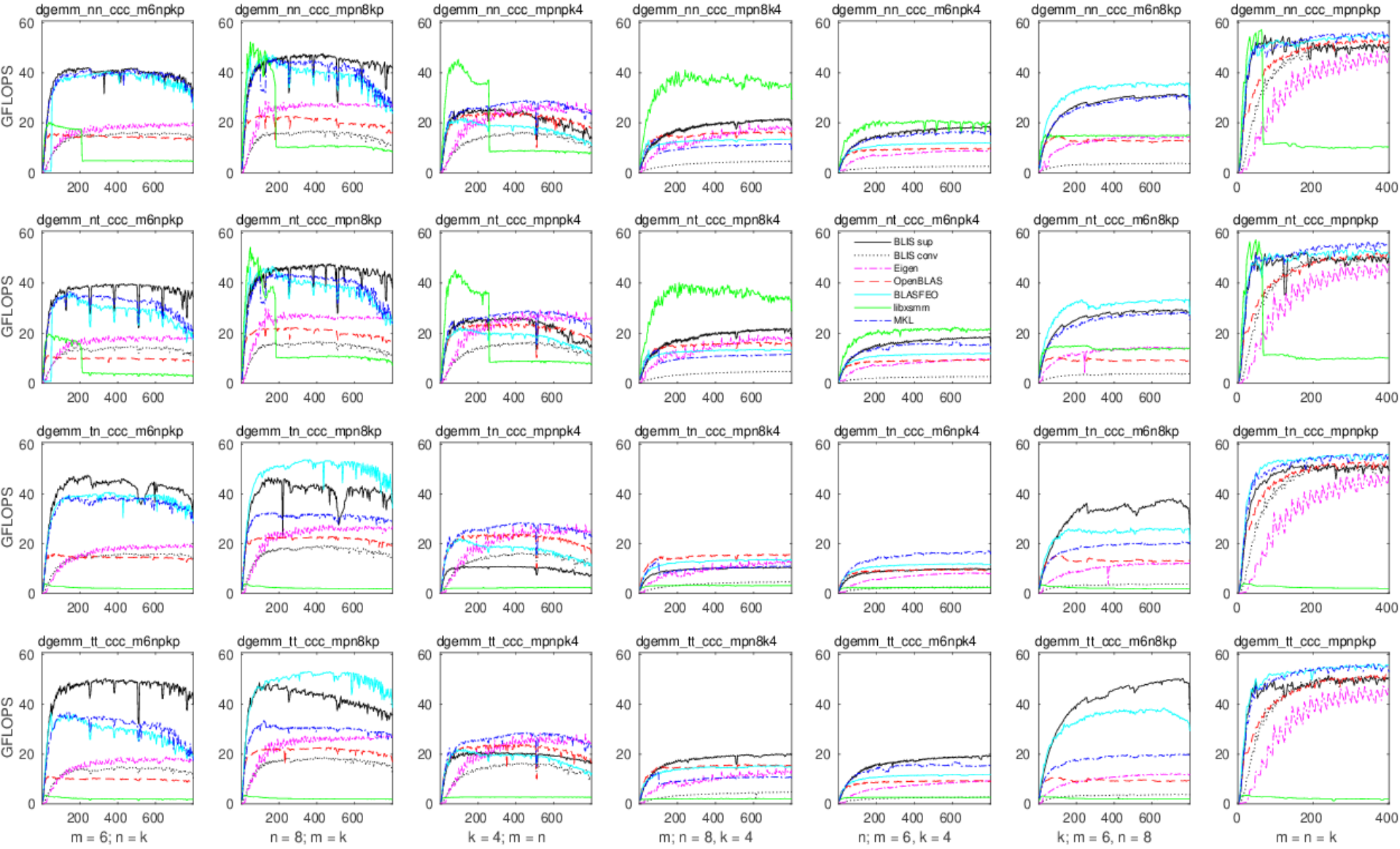
Performance

- Following results were gathered on
  - 3.8GHz Intel Kaby Lake (i5-7500)
- Implementations:
  - BLIS “sup”(skinny kernels; no packing)
  - BLIS “conv” (conventional kernels; with packing)
  - OpenBLAS
  - Eigen
  - MKL
  - BLASFEO
  - libxsmm
- So how did we do?

Performance: row storage



Performance: column storage



Conclusion

- Skinny gemm is definitely more complex than large gemm
  - But those complexities can be managed using BLIS's "principles" of DLA design and code management
- And what about the "dumb" benchmark (ie: all dimensions relatively small and square)?
  - Turns out to work pretty well here, too!
- Is the "BLIS approach" optimal? No.
  - Do we care? Not really.
  - Why? We're happy to give up the last 5% in the name of productivity

Conclusion

- Bottom line
  - BLIS now provides a unified BLAS-like framework for (typically) achieving 90-95% of attainable peak performance for both small and large problem domains across a wide swath of storage and transposition scenarios
  - No other open source project provides this :)

Acknowledgements

- This work made possible thanks to collaborative partnership with AMD
  - Kiran Varaganti

Further Information

- Website:
  - http://github.com/flame/blis/
- Discussion:
  - http://groups.google.com/group/blis-devel
  - http://groups.google.com/group/blis-discuss
- Contact:
  - field@cs.utexas.edu

Thank you!