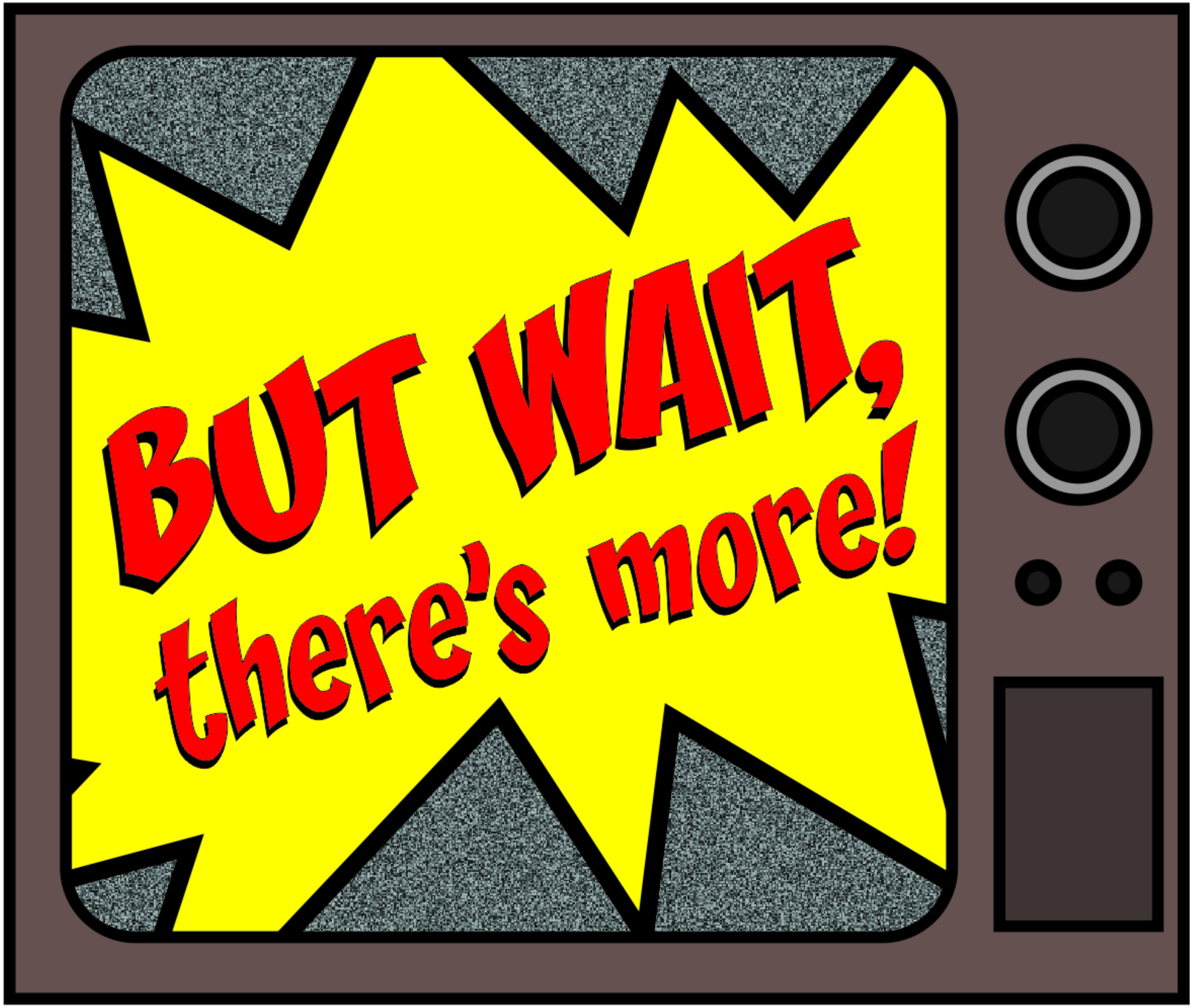
BLIS as a Research Vehicle

Bryan Marker
The University of Texas at Austin

Non-Traditional

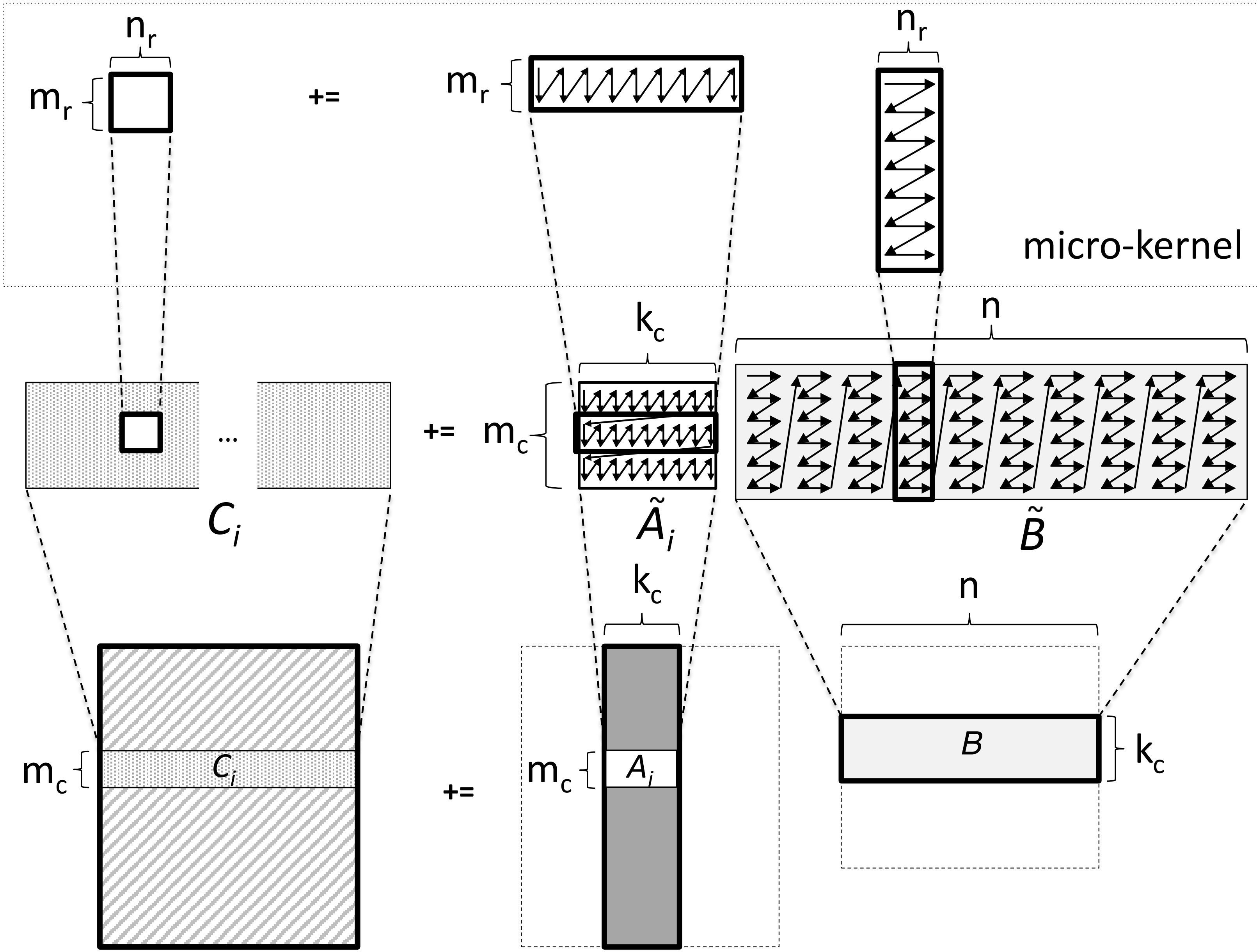
- Traditional function interfaces are limiting
- If you want non-supported behavior
  - You have to write (inefficient) wrapper code to the BLAS operations
  - OR you have to write your own BLAS-like operation and suffer bad performance or spend A LOT of time porting to each architecture
- With BLIS, as we know, you have more options for high-level functionality
- We expect this will allow users to optimize code like never before
- This will enable new research and code development


BUT WAIT,
there's more!

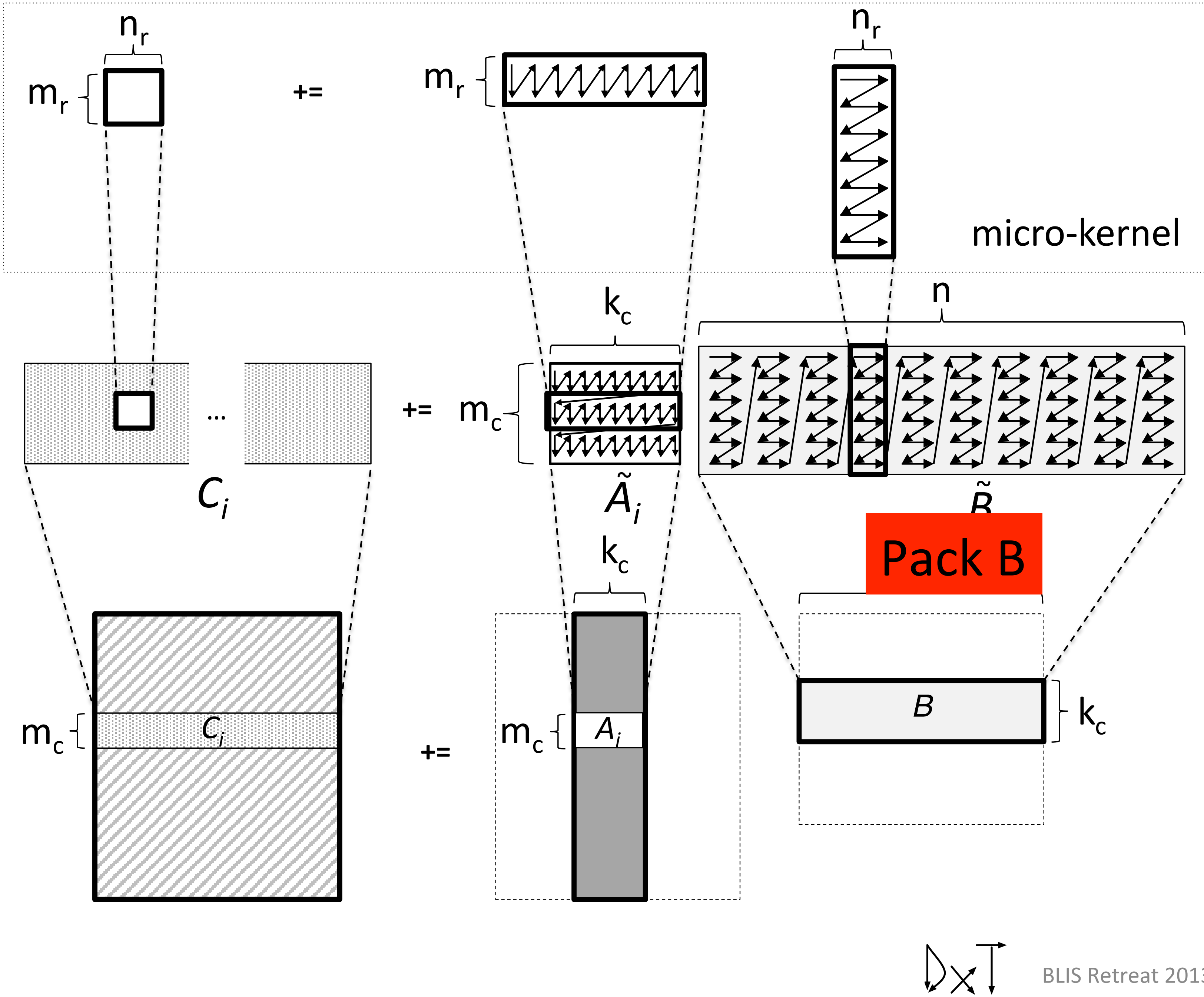
Low-Level Operations!

- We know all BLAS3 operations are implemented in terms of
  - Loops that partition matrices in specific ways
  - Kernels that copy and permute data into packed buffers
  - Kernels that compute on packed buffer

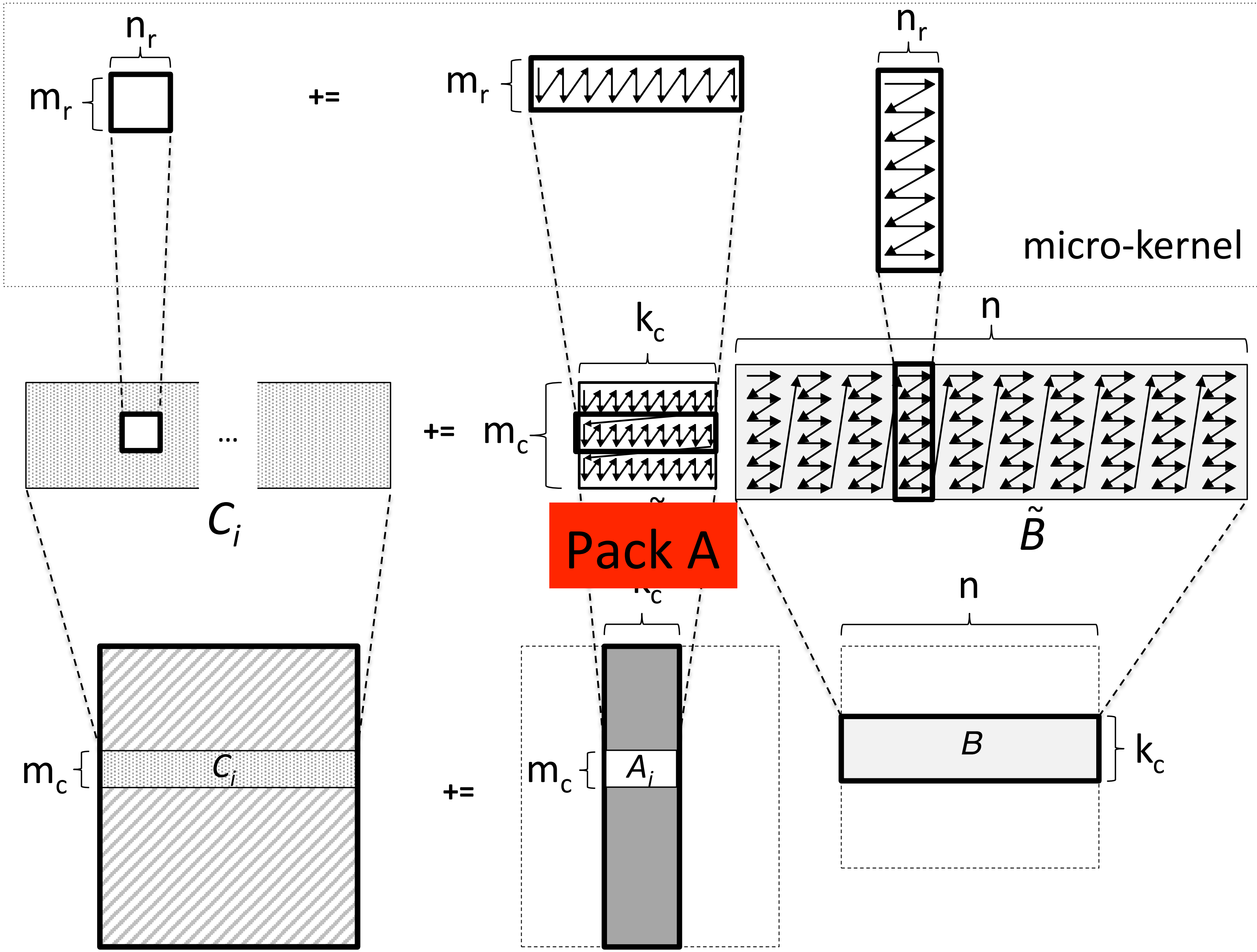
- The packing and computation kernels are low-level operations that are hidden from general BLAS/BLIS users
  - They're currently only used by the BLAS developer



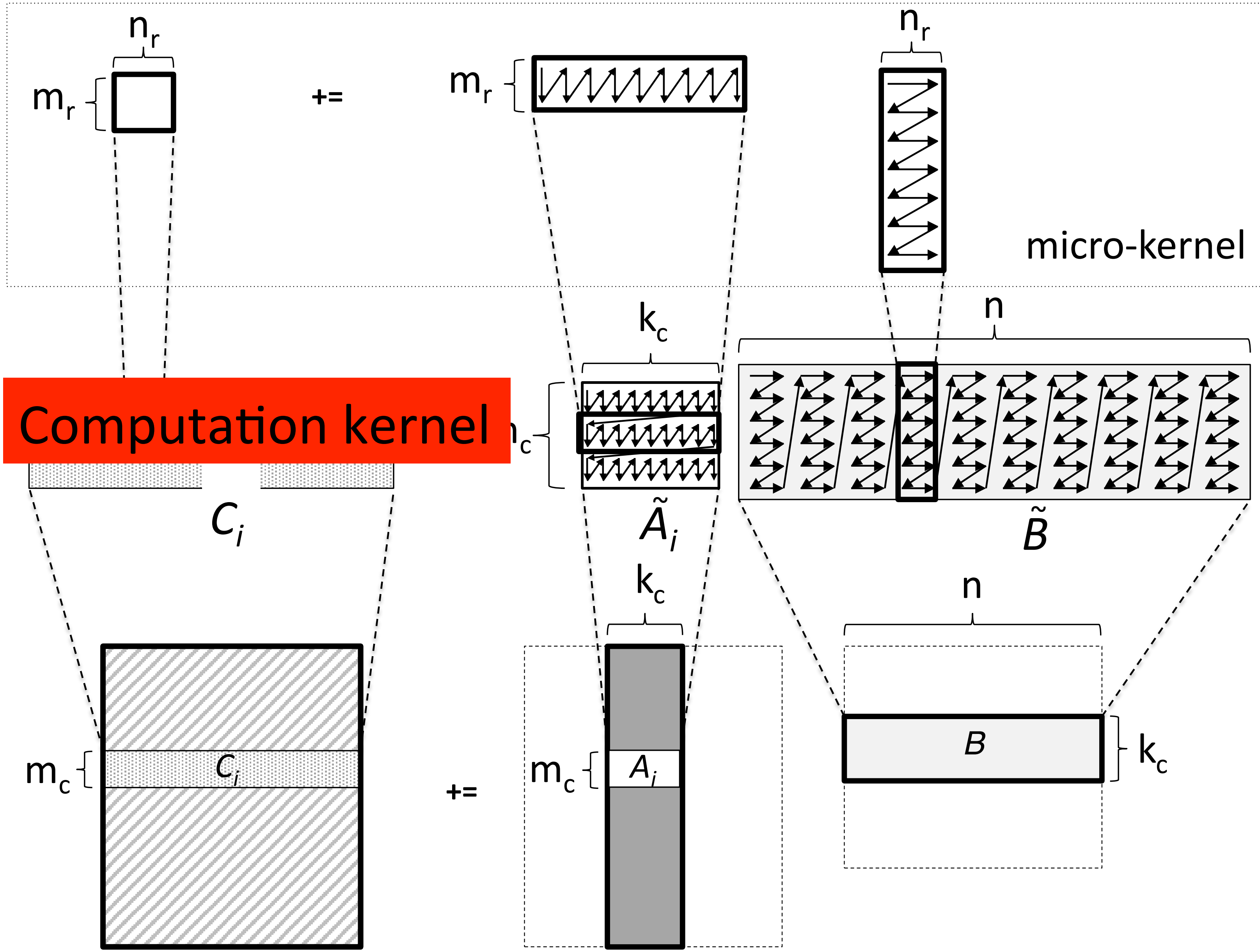
n_r
m_r
+=
m_r
n_r
micro-kernel
k_c
n
m_c
...
+=
C_i
$\tilde{A}_i$
$\tilde{B}$
k_c
n
m_c
C_i
+=
m_c
A_i
B
k_c



n_r
m_r
+=
m_r
n_r
micro-kernel
k_c
n
m_c
C_i
Ã_i
B̃
Pack B
k_c
m_c
C_i
+=
m_c
A_i
B
k_c



n_r
m_r
+=
m_r
n_r
micro-kernel
k_c
n
C_i
...
+=
m_c
Pack A
B̃
k_c
n
m_c
C_i
+=
m_c
A_i
B
k_c



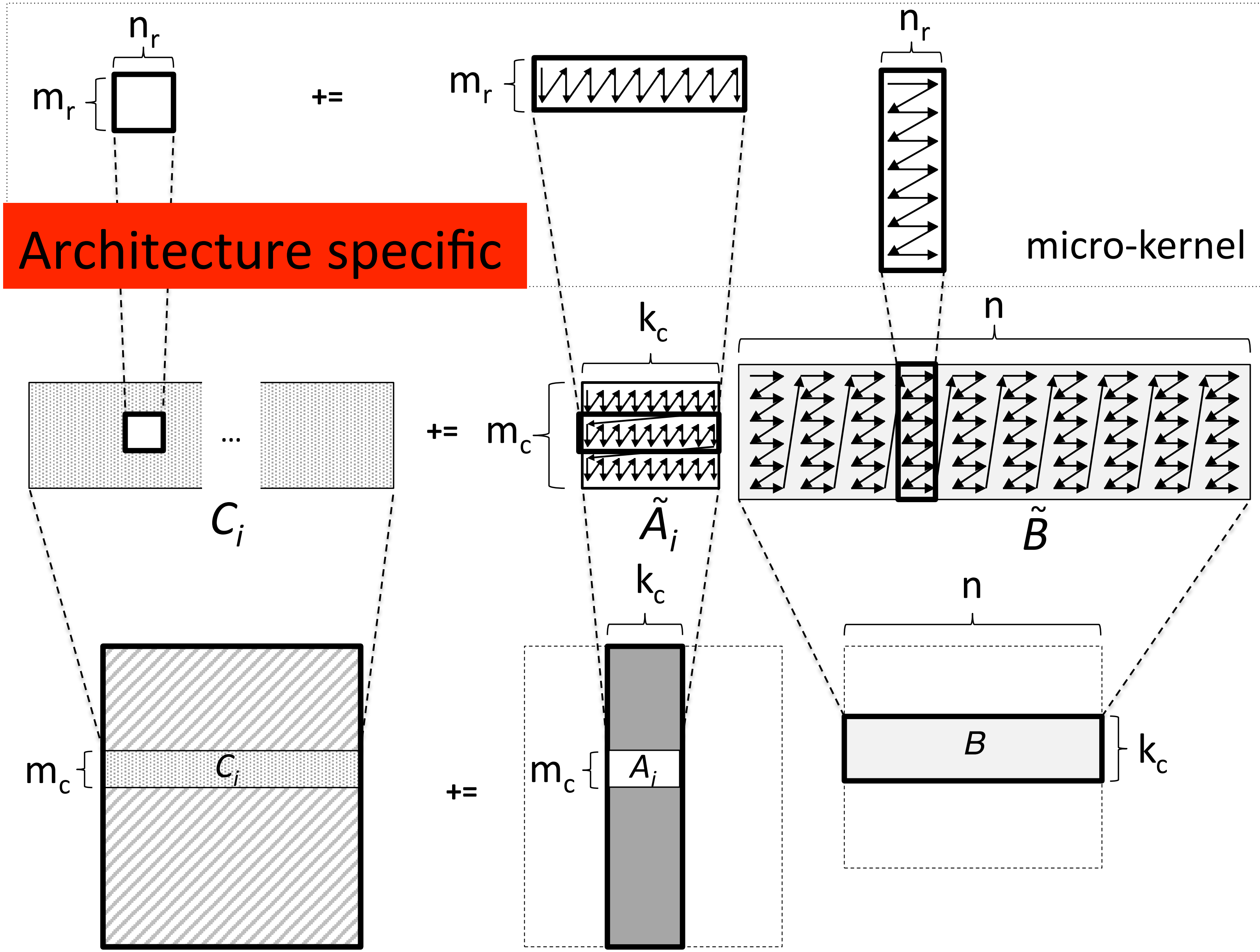
n_r
m_r
+=
m_r
n_r
micro-kernel
Computation kernel
k_c
n
C_i
Ã_i
B̃
k_c
n
m_c
C_i
+=
m_c
A_i
B
k_c

Low-Level Operations

- BLAS experts code using calls to these low-level operations

- Experts know the preconditions/postconditions of these functions
  - Input/output sizes (blocksizes)
  - Expectations on levels of cache in which data reside

- Operations above a certain level are portable
  - E.g. the entire stack of code for Gemm isn't ported to each architecture
  - Below some level, architecture-specific code is used
  - Above that level, the same code is used for all architectures
  - E.g. BLIS microkernels are architecture specific, but the macrokernels built from them are are not



Architecture specific
micro-kernel
n_r
m_r
+=
k_c
n
m_c
...
C_i
$\tilde{A}_i$
$\tilde{B}$
A_i
B

Low-Level Operations

- This approach to implementing the BLAS3 is fairly standard across BLAS libraries
- For closed-source libraries, you can't see/access the necessary operations
- For existing open-source libraries the operations are buried and complicated to understand unless you are an expert or close to an expert
  - Developing good low-level operations was not a design goal
  - No need to export requirements on and interfaces to low-level operations

Low-Level Operations

- The result is that you cannot easily understand the BLAS3 code
  - You have to become an expert in the particular BLAS library
- You certainly cannot teach the code to students
  - You can understand the general algorithms
  - You cannot point to specific lines of the GotoBLAS or MKL and say "this is how it's done in practice"
- You cannot code and optimize your own BLAS-like operations
  - E.g. Gemm followed by Trsm with the same "B"
  - Combinations of BLAS operations can incur inefficiencies hidden within the library
  - You need low-level operations to optimize this in a portable way
  - You want to be able to implement the algorithm with low-level operations and just change the microkernels for each architecture – portability!

BLIS

- A major design goal of BLIS is to change traditional BLAS layering
  - FLAME research has demonstrated the performance and pedagogical utility in exposing low-level kernels – more on this shortly
- In prototyping our ideas, we used complicated GotoBLAS low-level operations
  - I learned A LOT about the GotoBLAS
- Now, BLIS improves on GotoBLAS operations
  - Similar computation pattern with more readable code

- Some BLIS design goals
  - Develop the low-level operations to be understandable and usable without hindering performance
  - Implement BLAS3 algorithms using these operations in an understandable way
  - Code operations in terms of microkernels to enable easy portability

BLIS

- Now, you have a high-performance library built from low-level operations
  - You can understand the algorithms
  - You can explain the algorithms to novices
- You can use the low-level kernels for exploring / implementing new BLAS-like algorithms
  - By replacing micro-kernels with platform-tuned implementation, your algorithms are portable
- BLIS's low-level operations will enable
  - Higher performance, portable code for unique BLAS-like algorithms that show up repeatedly in DLA libraries
  - New research into DLA software engineering
  - New research into software built on DLA

LET'S SHIFT GEARS

My Research

- Encode knowledge about software instead just the result of applying knowledge (code)
  - We shouldn't just store code because we loose too much information about the software

- Many high level goals, including
  - Automatic program generation/derivation
  - Better understood code
  - More trusted code
  - Easier adaptation to changing architectures

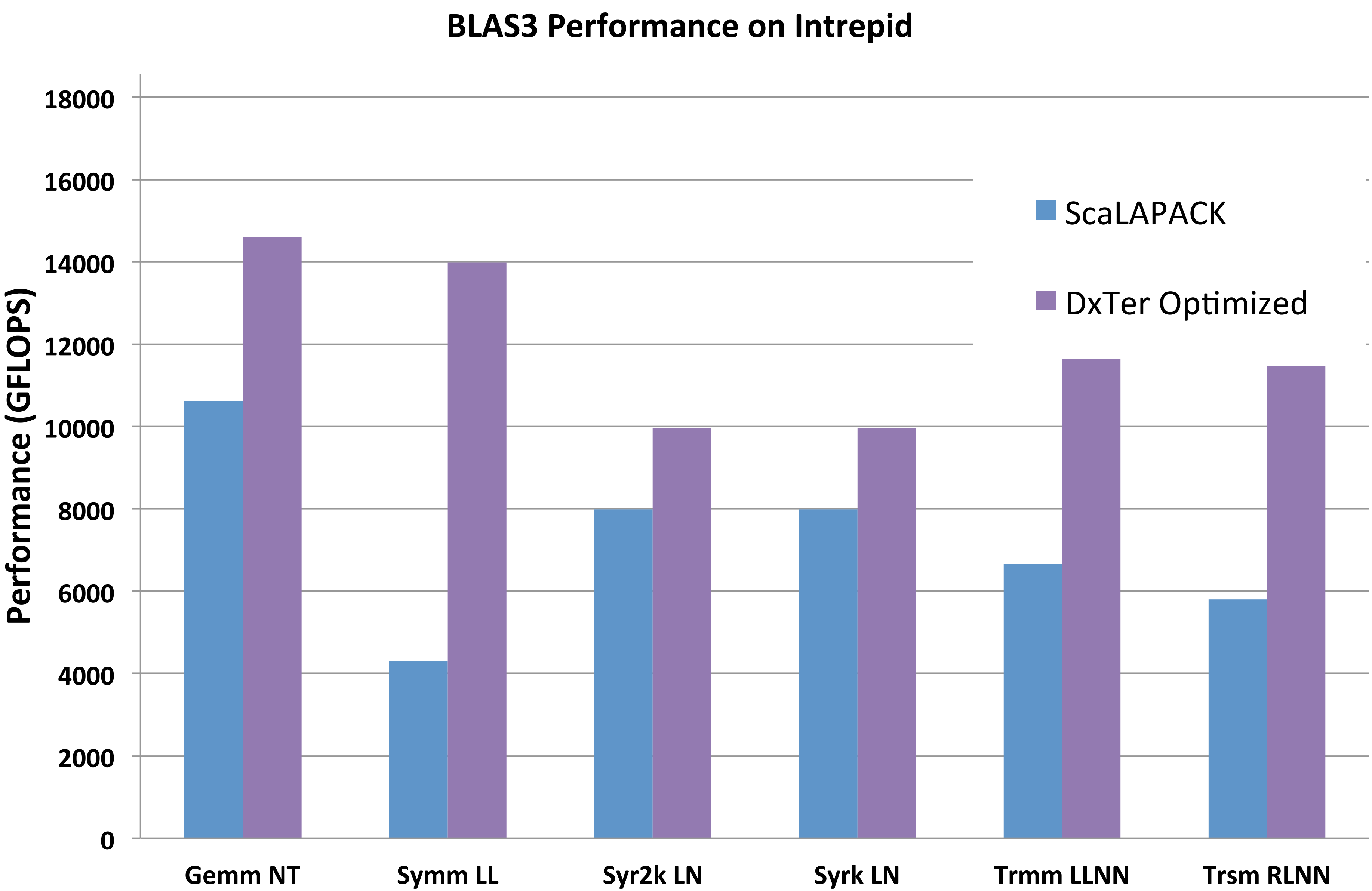
- Dense linear algebra (DLA) is a well-understood domain to start my research and I am using BLIS as a research vehicle
  - The results contribute to BLIS's development

Design by Transformation

- Design by Transformation (DxT)
  - Way to encode the expert knowledge about a domain (like dense linear algebra) and software to implement domain's functionality
  - Knowledge is encoded as graph transformations where graphs represent functionality
  - With knowledge encoded, it can be automatically applied to implement and optimize algorithms for a target architecture
  - I'll explain the basics

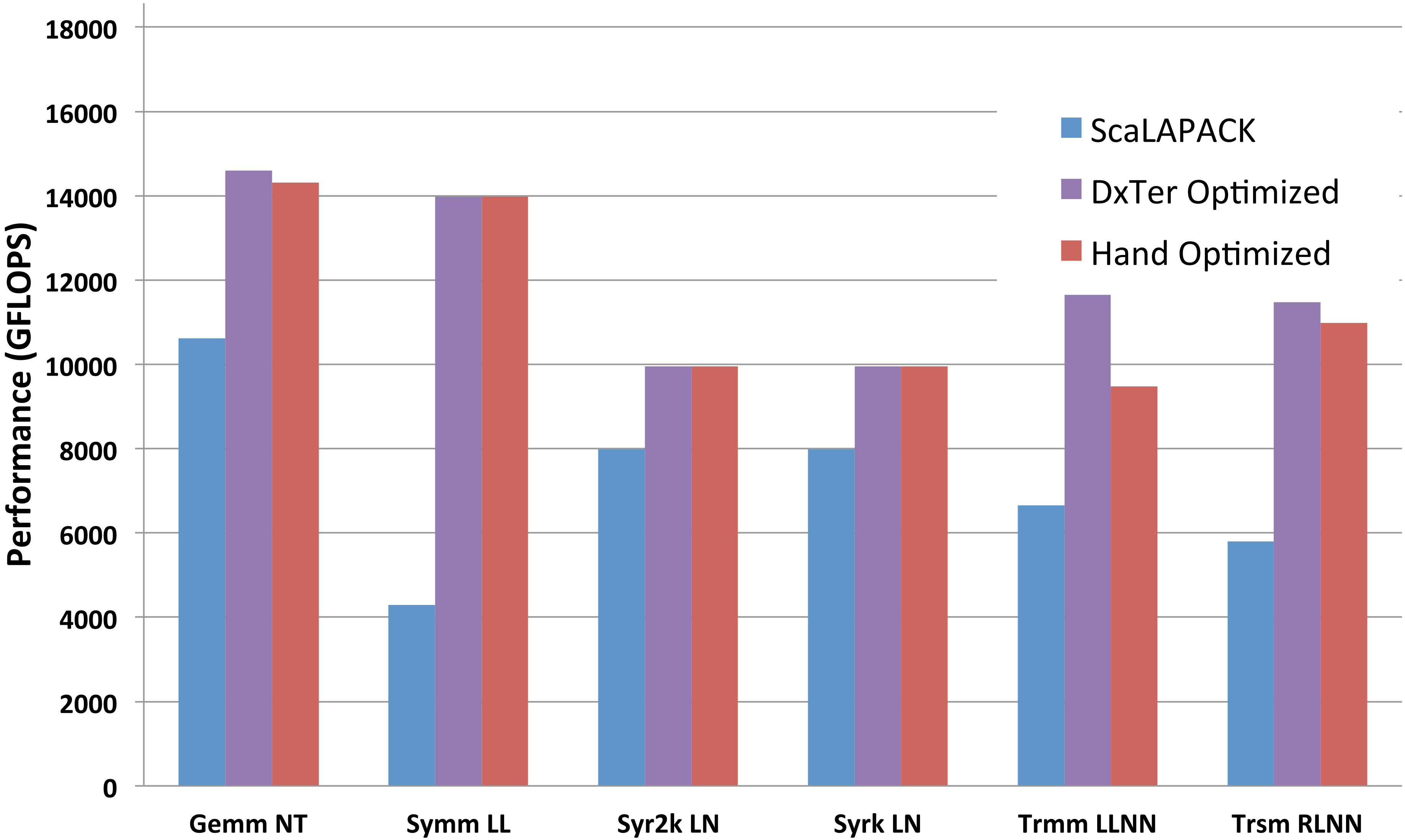
DxT

- DxT was first applied to the distributed-memory library Elemental
- DxT automatically explores distribution/parallelization options and algorithmic variants that a person would explore manually
- There are many cases where DxT-generated code is better performing than hand-implemented
- There is one case where the expert made a coding mistake
  - DxT generates correct code by design
- DxT generated code has been incorporated into the Elemental library
- Now, DxT is being applied to BLIS



BLAS3 Performance on Intrepid
Performance (GFLOPS)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
ScaLAPACK
DxTer Optimized
Gemm NT
Symm LL
Syr2k LN
Syrk LN
Trmm LLNN
Trsm RLNN

BLAS3 Performance on Intrepid



Graphs

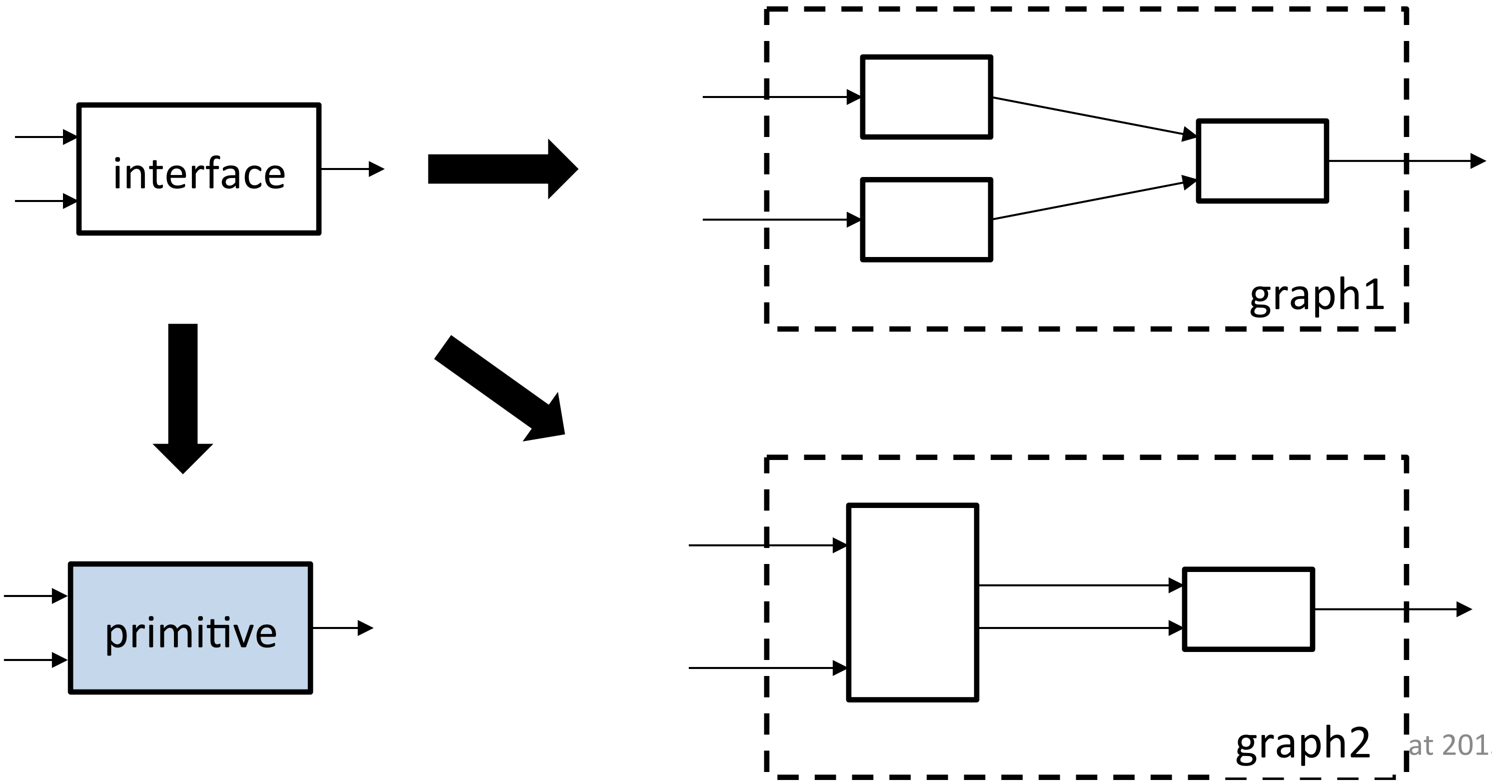
- Data-flow, directed acyclic graphs (DAGs) encode algorithms and implementations

- A box or node represents an operation
  - An interface without implementation details
  - OR a primitive operation that maps to given code

- A starting algorithm without implementation details is encode as a graph of interfaces
- We want to transform it into a graph with complete implementation details
  - Transform into a graph of primitives that represent BLIS low-level kernels
  - Convert the graph to BLIS code, calling low-level kernels

Transform with Implementations

- **Refinements** replace a box without implementation details
  - Chooses a specific way to implement the box's functionality
  - E.g. choose a loop-based algorithm to implement Gemm



BLIS Implementations

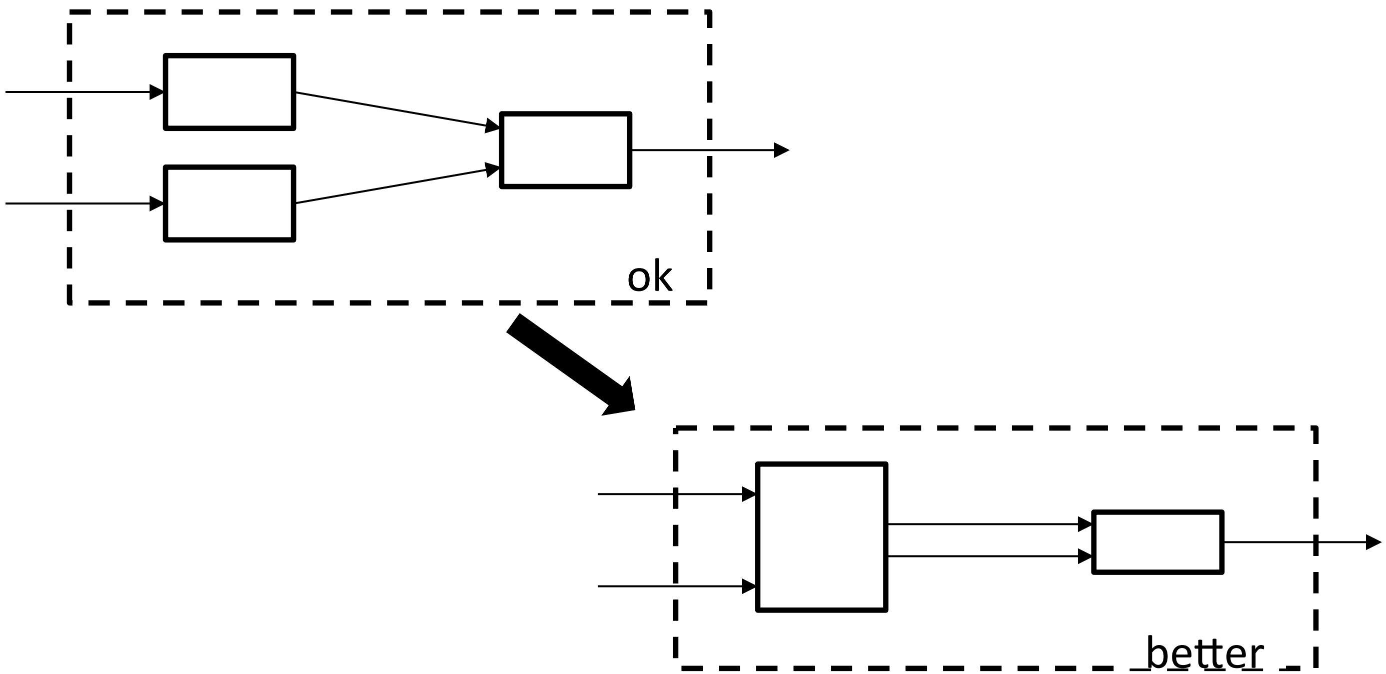
- Gemm implemented
  - In terms of a loop around smaller Gemm subproblems
  - OR in terms of packing operations and computation kernels
- BLIS makes these relationships clear (understandable)

- We can encode such implementation knowledge
- That knowledge can be used for the BLAS3 and BLAS-like operations
  - Remember: all other BLAS3 operations are built on Gemm

- Notice that there is DLA knowledge overlap between Elemental code and BLIS
  - Implementing Gemm in terms of small Gemm is useful on any architecture
  - DxT makes this overlap of domain knowledge explicit by reusing architecture-agnostic transformations

Transform to Optimize

- **Optimizations** replace a subgraph with another subgraph
  - Same functionality
  - A different way of implementing it



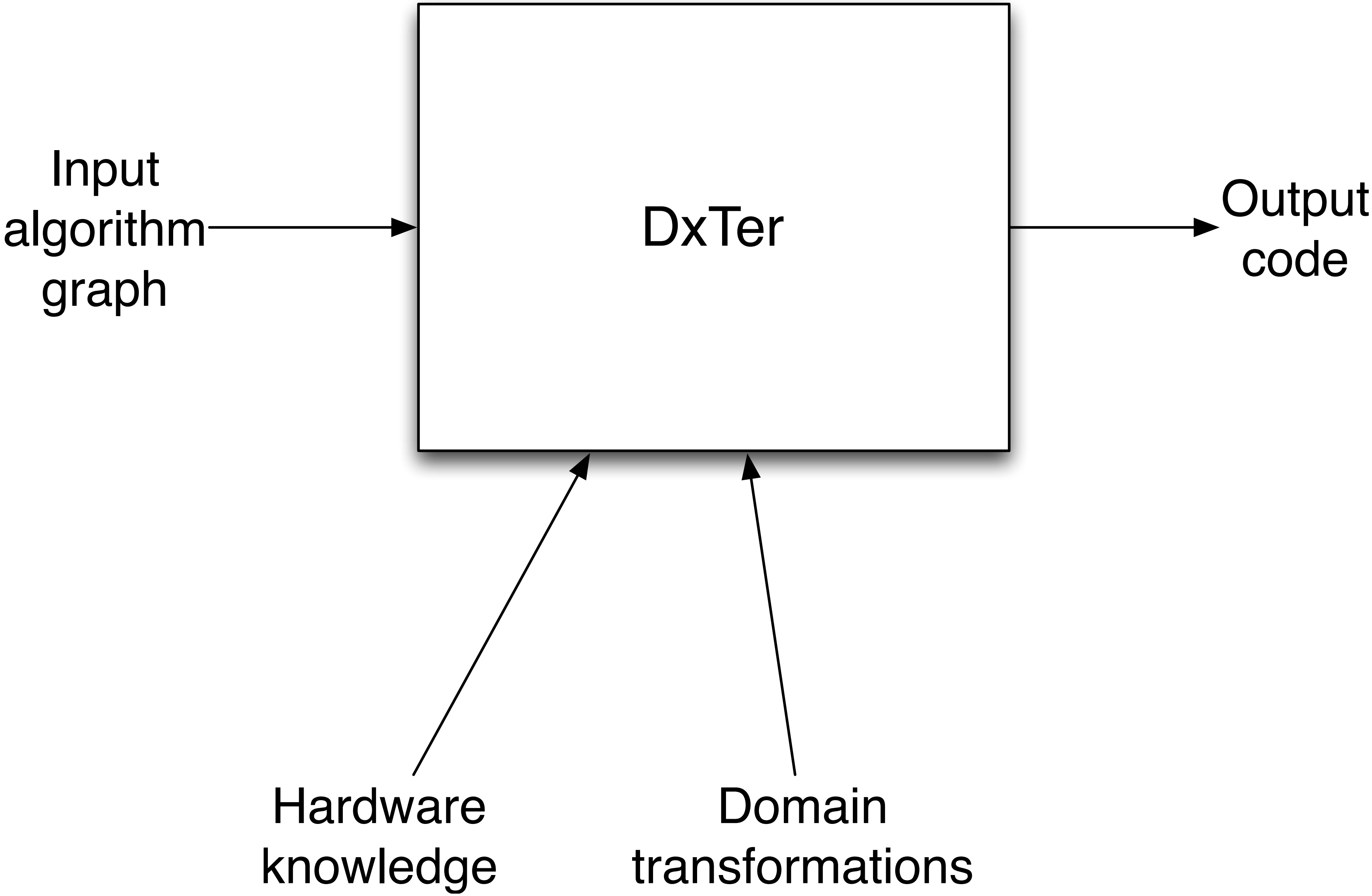
Optimizations

- One does not pack data twice
  - An expert knows that this is unnecessary – get rid of one pack operation
  - Encode such expert knowledge as an optimization
  - This optimizes the Gemm+Trsm algorithm

- Optimizations can also encode ways to parallelize loops / macrokernels

With Knowledge Encoded

- We can use a mechanical system to explore implementation options and generate code for the BLAS3
  - Apply transformations to generate a search space of implementation options
  - Use an estimate of implementation costs to rank-order the implementations
  - Choose the "best" graph and output code by mapping each box to a BLIS call


Input
algorithm
graph
DxTer
Output
code
Hardware
knowledge
Domain
transformations

Results

- BLAS3 can all be generated by DxTer
  - Basically, this verifies Field's code

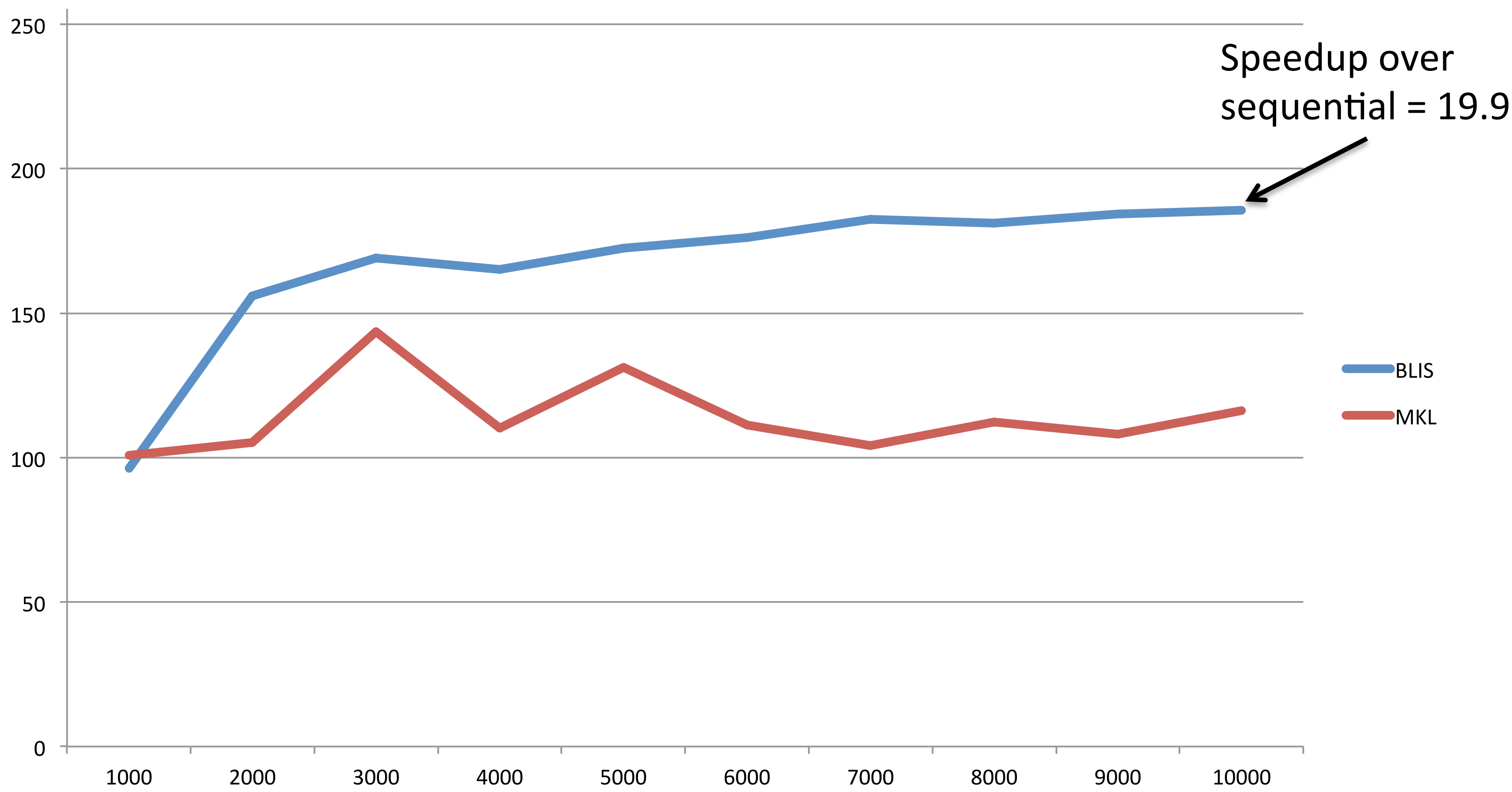
- Use the same knowledge on more complicated operations (like the BLAS calls in QR or two-sided Trsm/Trmm)
  - Allow DxTer to fuse loops and remove unnecessary packing automatically
  - Get speedup from removing unnecessary packing

Results

- Tyler has demonstrated how to parallelize Gemm
  - Rules about which loops to parallelize and to what degree
- Other BLAS3 operations have similar structure to Gemm
  - n-dimension loop
  - k-dimension loop
  - m-dimension loop

- I have encoded knowledge about Gemm parallelism
  - Take the DxTer-derived sequential code and tags loops/operations with different amounts of parallelism
- DxTer applies the same knowledge to parallelize other BLAS3 operations
  - Only does so when “legal”
- DxTer generates parallel code for all BLAS3 operations automatically

Trmm (Left, Lower, Non-Trans)



BLIS as a Research Vehicle

- This work could not have been done with traditional BLAS software

- With BLIS
  - We can understand the algorithms and how they're used/implemented
  - We can use the low-level kernels to construct our own BLAS or BLAS-like operations
  - We can optimize beyond what the traditional BLAS functions allow

- With software that is so understandable at all layers, we can teach about the software and we can even automate its construction

Questions?

bamarker@cs.utexas.edu

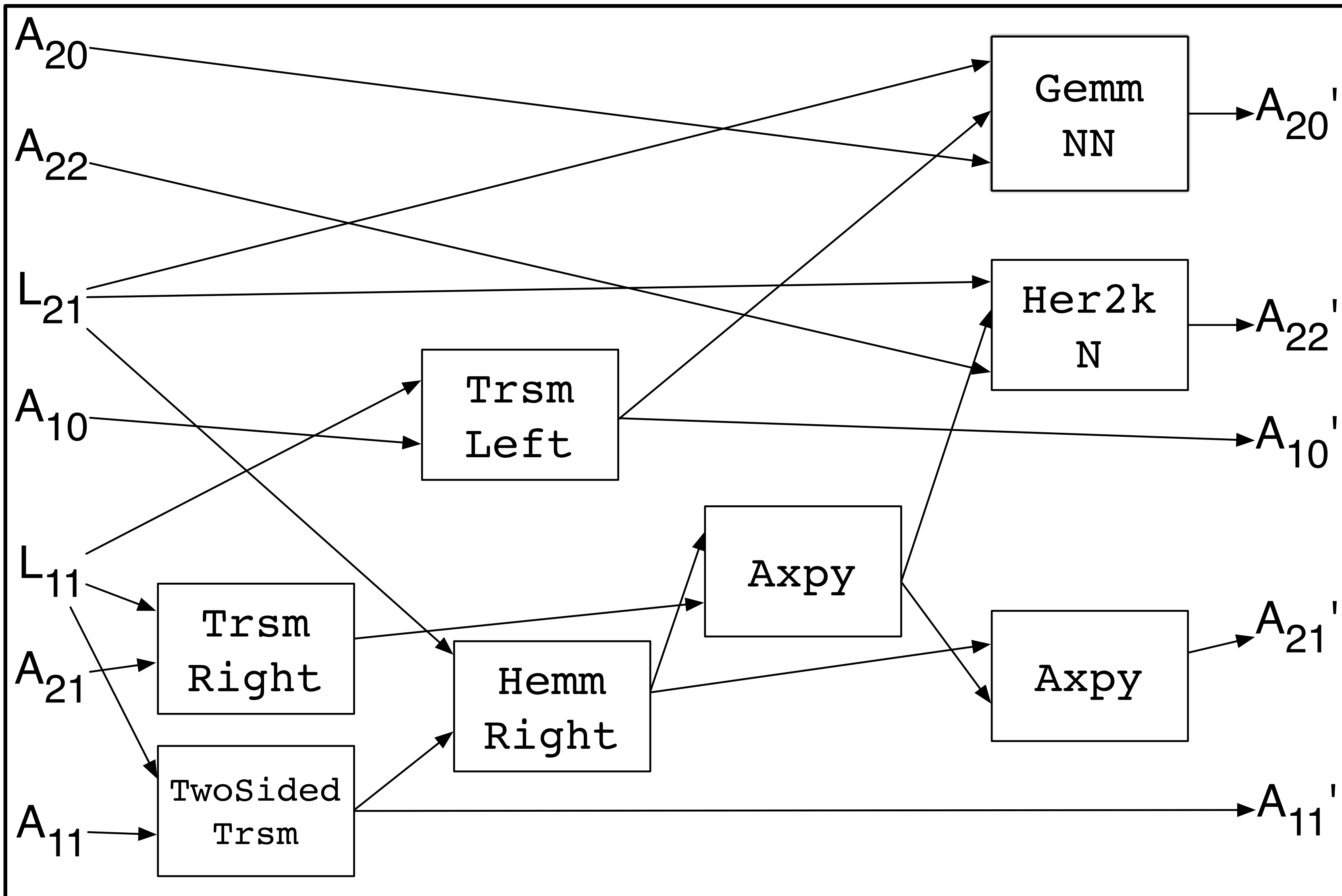
www.cs.utexas.edu/~bamarker

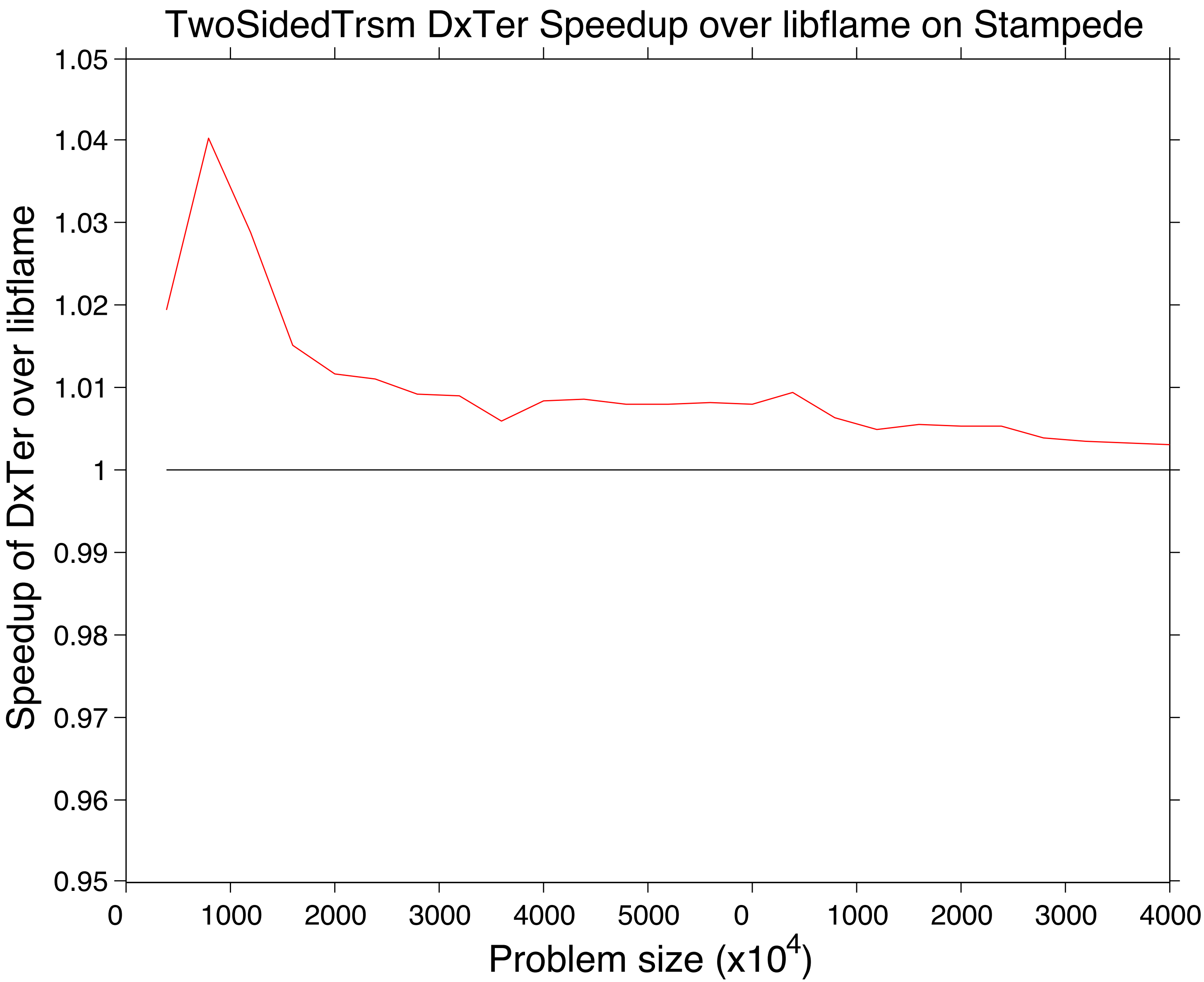
code.google.com/p/dxter/

I was funded by NSF and Sandia Graduate Research Fellowships

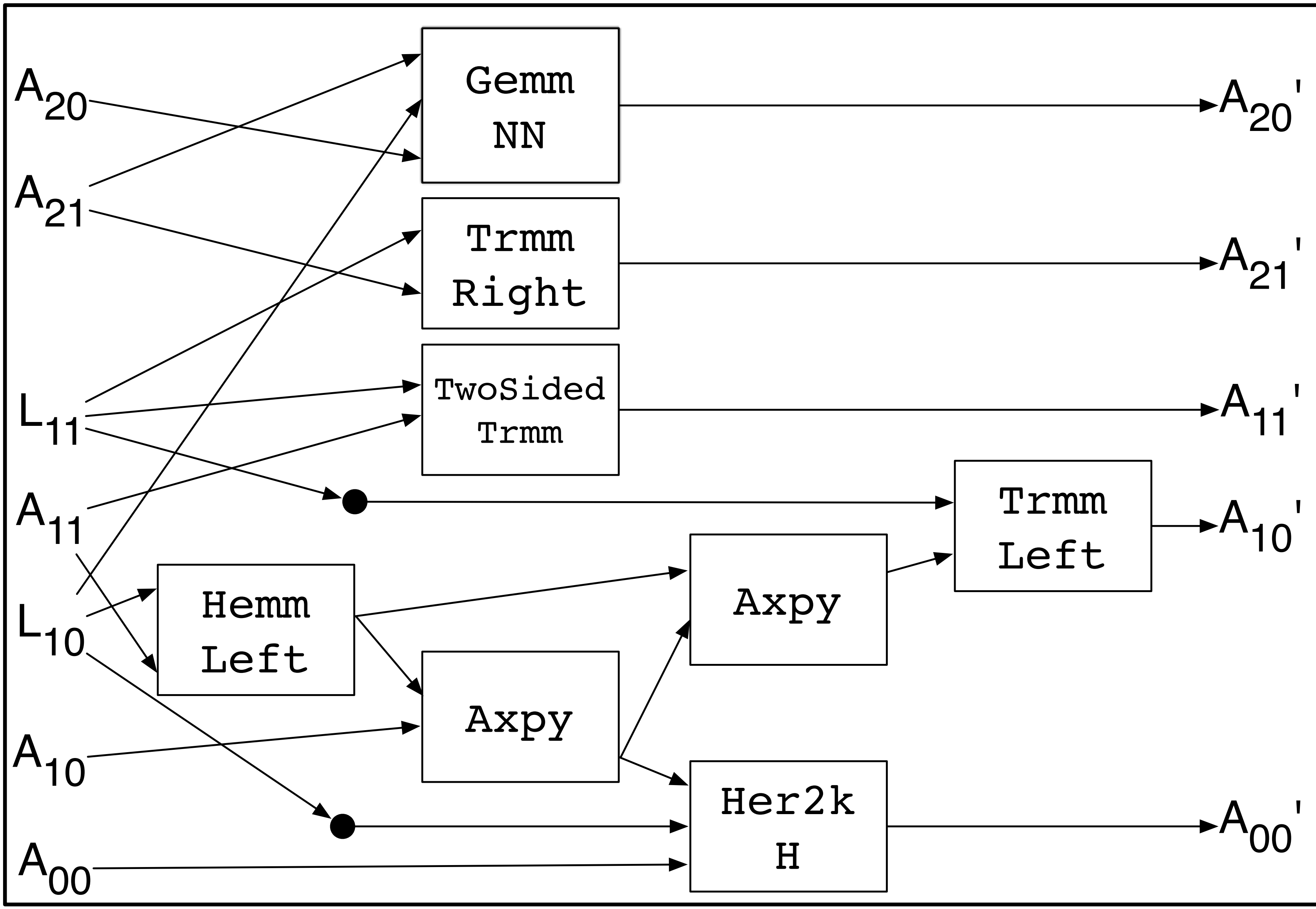
Thanks!

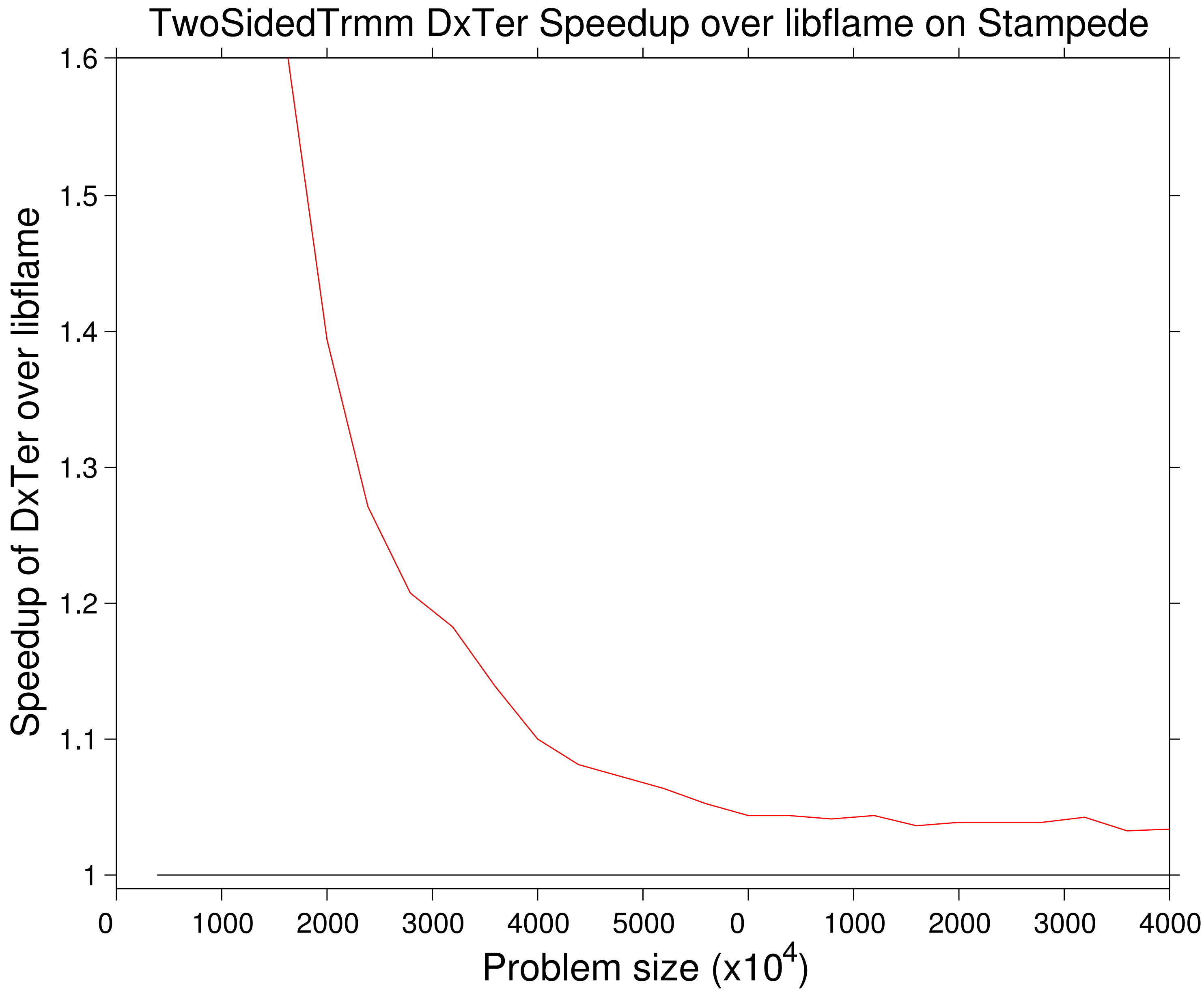
Starting Graph

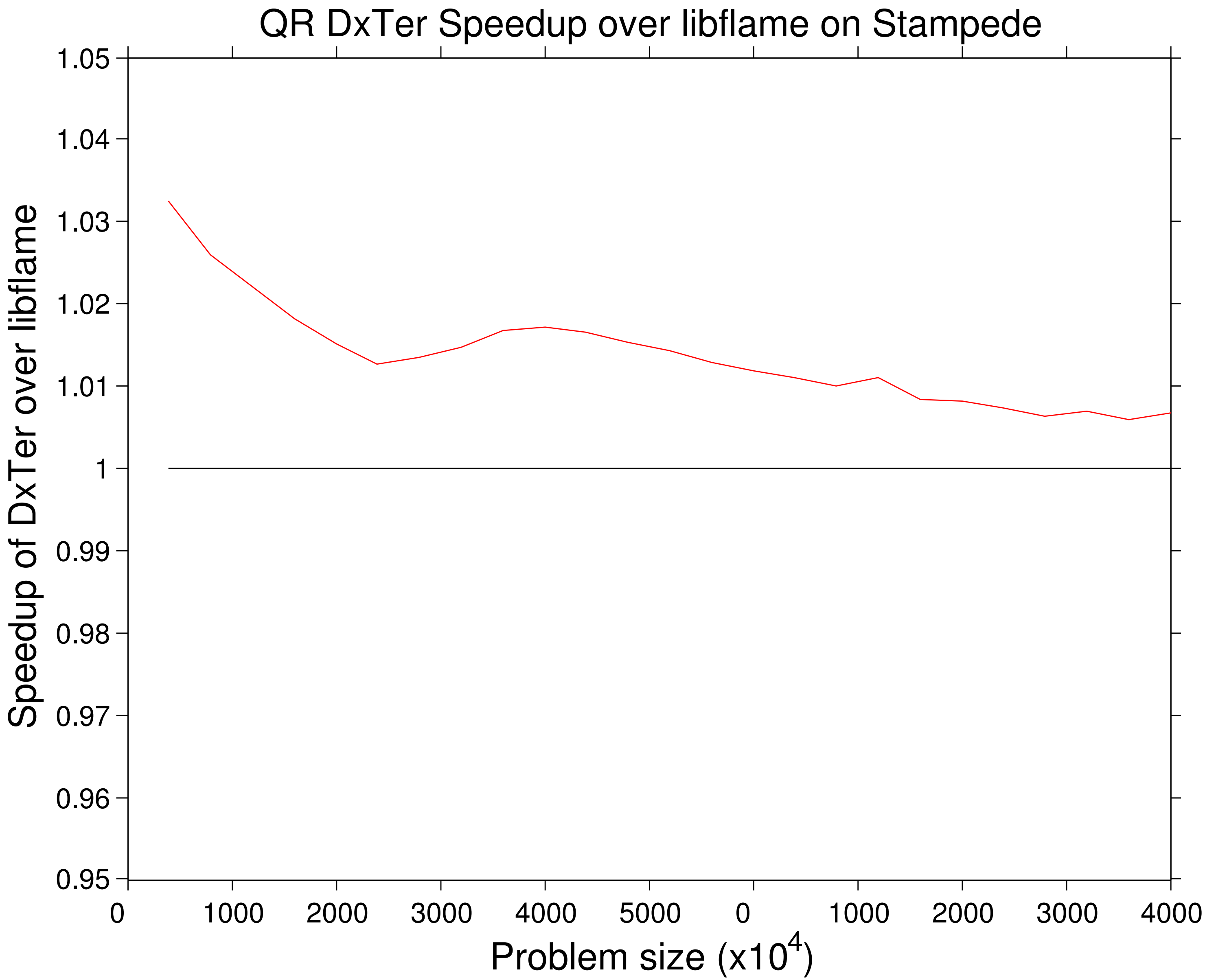



TwoSidedTrsm DxTer Speedup over libflame on Stampede
Speedup of DxTer over libflame
Problem size (x10^4)
1.05
1.04
1.03
1.02
1.01
1
0.99
0.98
0.97
0.96
0.95
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000

One Starting Graph




TwoSidedTrmm DxTer Speedup over libflame on Stampede
Speedup of DxTer over libflame
1.6
1.5
1.4
1.3
1.2
1.1
1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
Problem size (x10^4)


QR DxTer Speedup over libflame on Stampede
Speedup of DxTer over libflame
1.05
1.04
1.03
1.02
1.01
1
0.99
0.98
0.97
0.96
0.95
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
Problem size (x10^4)