Distributed-Memory Dense Linear Algebra Program Generation

Bryan Marker, Don Batory, Robert van de Geijn

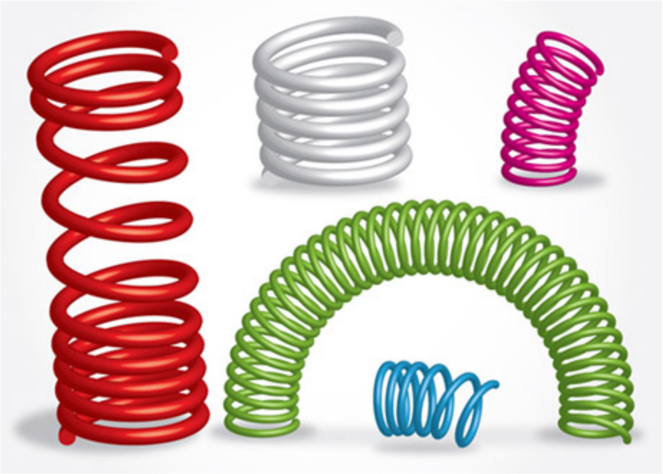
The University of Texas at Austin

A Product of Collaboration

- Don Batory
  - Software product lines
  - Relational query optimization
  - Software engineering
- Robert van de Geijn
  - Dense linear algebra
  - High performance computing
  - Many libraries targeting various architectures
- Bryan Marker
  - Undergrad research experience with Robert
  - Two years working as a software engineer on code generator and data-flow programming language
  - Co-advised by both professors

Let's get on the same page

- For **computation science and engineering (CSE)** and many other fields, domain EXPERTS are rare
  - It takes a lot of experience to become one
- We need domain experts to get high performance, trusted code
- They provide many libraries
  - Users expect many functions
  - Many target architectures
    - Distributed memory
    - Shared memory
    - Sequential
    - GPGPUs
    - Combinations of these



Let's get on the same page

- Knowledge is manually reapplied when
  - A new function implementation is needed
  - A new architecture comes out
  - A new optimization is discovered for a particular hardware stack

- Experts end up doing a lot of rote development to apply their rare knowledge repeatedly
  - Algorithmic knowledge
  - Hardware knowledge

Holy Grail

- Instead of encoding result of applied knowledge (code), encode expert knowledge
- Then, experts only concern themselves with
  - Sequential algorithms
  - Knowledge about implementing pieces on (parallel) architectures

- Automatically generate optimized implementations
- Get people out of software development loop as much as possible

Towards Program Generation

- Let’s work towards encoding expert knowledge and automatically applying it

- Let’s work towards leveraging the expert’s abilities
  - Automatically applying his/her knowledge

- Allow the expert to gain new knowledge
  - Or relax

DxT

Dense Linear Algebra

- **Dense linear algebra (DLA)** is a prime domain to explore these ideas
- Decades of engineering has led to well-layered software
  - Layering makes the expert more effective
  - He/she only needs to port some software components
  - Think: replace a sequential library with a shared-memory library
- Benefit to us: easier to encode with layering
  - Encode knowledge about pertinent software components instead of all code expressible in general-purpose language

WHO KNOWS ABOUT THE BLAS?

The Basic Linear Algebra Subroutines (BLAS)

Jargon

- Collection of commonly-used DLA operations
  - matrix-matrix multiplication
  - multiplication by a triangular matrix
- Often the bottom of a software stack
- Portability of user applications
  - An appropriate BLAS library can be (easily) linked in
  - This layering is common in DLA and higher-level applications using DLA
- We are in the habit of coding in terms of limited functionality (e.g. with the BLAS)

Distributed-Memory (Cluster) Architectures

Jargon

- Many computers connected via high-speed network
- Here, we use collective communication routines found in the **Message Passing Interface (MPI)**
  - (Another layer)
- Difficult to code for
- Layer on top of distributed-memory DLA libraries
  - Other libraries
  - User applications

DxT

NOW LET'S TALK ABOUT ENCODING EXPERT KNOWLEDGE TO AUTOMATICALLY GENERATE CODE…

Distributed-Memory DLA

Jargon

- We target the Elemental library
  - Distributed-memory DLA
  - Functionality similar to ScaLAPACK
  - C++ library with an API we use as a **domain-specific language (DSL)**

DxT

Elemental

Jargon

- View the *p* processes in the cluster as a 2-dimensional grid
- About 10 distributions of matrices onto the grid
  - Default (elemental) distribution
  - Other options enable parallelization
  - The expert knows which are valid for each input/output matrix for each operation
- Enable parallel computation by
  - Redistributing data from default distribution to other distributions
  - Performing locally sequential computation on all processes
  - Redistributing data back to default distribution (possibly with reduction)

Distributed-Memory DLA

- We want to encode the knowledge of Jack Poulson and Robert van de Geijn, the expert developers
- We want to use that knowledge to automatically generate the same or better code

- We use **Design by Transformation (DxT)** as our way to encode expert knowledge
  - Hint: we use graphs to encode algorithms/implementations and transformations on graphs to encode expert design knowledge

$A = L\,L^T$

Algorithm: $A := \text{CHOL\_BLK\_VAR3}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$A_{11} = \Gamma(A_{11})$

$A_{21} = A_{21}\,\text{TRIL}\,(A_{11})^{-T}$

$A_{22} = A_{22} - \text{TRIL}\left(A_{21}A_{21}^T\right)$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
PartitionDownDiagonal
    ( A, ATL, ATR,
         ABL, ABR, 0 );
    while( ABR.Height() > 0 )
    {
        RepartitionDownDiagonal
        ( ATL, /**/ ATR,        A00, /**/ A01, A02,
         /*************/        /***************/
               /**/             A10, /**/  A11, A12,
          ABL, /**/ ABR,        A20, /**/  A21, A22 );
        A21_VC_Star.AlignWith( A22 );
        A21_MC_Star.AlignWith( A22 );
        A21_MR_Star.AlignWith( A22 );
        //------------------------------------------------------//
        A11_Star_Star = A11;
        internal::LocalChol( Lower, A11_Star_Star );
        A11 = A11_Star_Star;

        A21_VC_Star = A21;
        internal::LocalTrsm
        ( Right, Lower, ConjugateTranspose, NonUnit,
          (F)1, A11_Star_Star, A21_VC_Star );

        A21_MC_Star = A21_VC_Star;
        A21_MR_Star = A21_VC_Star;

        internal::LocalTriangularRankK
        ( Lower, ConjugateTranspose,
          (F)-1, A21_MC_Star, A21_MR_Star, (F)1, A22 );

        A21 = A21_MC_Star;
        //------------------------------------------------------//
        A21_VC_Star.FreeAlignments();
        A21_MC_Star.FreeAlignments();
        A21_MR_Star.FreeAlignments();
        SlidePartitionDownDiagonal
        ( ATL, /**/ ATR,       A00, A01, /**/ A02,
               /**/            A10, A11, /**/ A12,
         /*************/       /****************/
          ABL, /**/ ABR,       A20, A21, /**/ A22 );
    }
```

$A = L L^T$

Algorithm: $A := \text{CHOL\_BLK\_VAR3}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$A_{11} = \Gamma(A_{11})$$
$$A_{21} = A_{21}\,\text{TRIL}\,(A_{11})^{-T}$$
$$A_{22} = A_{22} - \text{TRIL}\,(A_{21}A_{21}^T)$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
PartitionDownDiagonal
    ( A, ATL, ATR,
         ABL, ABR, 0 );
    while( ABR.Height() > 0 )
    {
        RepartitionDownDiagonal
        ( ATL, /**/ ATR,        A00, /**/ A01, A02,
         /*************/        /****************/
               /**/             A10, /**/  A11, A12,
          ABL, /**/ ABR,        A20, /**/  A21, A22 );
        A21_VC_Star.AlignWith( A22 );
        A21_MC_Star.AlignWith( A22 );
        A21_MR_Star.AlignWith( A22 );
        //------------------------------------------------------//
        A11_Star_Star = A11;
        internal::LocalChol( Lower, A11_Star_Star );
        A11 = A11_Star_Star;

        A21_VC_Star = A21;
        internal::LocalTrsm
        ( Right, Lower, ConjugateTranspose, NonUnit,
          (F)1, A11_Star_Star, A21_VC_Star );

        A21_MC_Star = A21_VC_Star;
        A21_MR_Star = A21_VC_Star;

        internal::LocalTriangularRankK
        ( Lower, ConjugateTranspose,
          (F)-1, A21_MC_Star, A21_MR_Star, (F)1, A22 );

        A21 = A21_MC_Star;
        //------------------------------------------------------//
        A21_VC_Star.FreeAlignments();
        A21_MC_Star.FreeAlignments();
        A21_MR_Star.FreeAlignments();
        SlidePartitionDownDiagonal
        ( ATL, /**/ ATR,       A00, A01, /**/ A02,
               /**/            A10, A11, /**/ A12,
         /*************/       /****************/
          ABL, /**/ ABR,       A20, A21, /**/ A22 );
    }
```

$$A = L\,L^T$$

Algorithm: $A := \text{CHOL\_BLK\_VAR3}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$A_{11} = \Gamma(A_{11})$

$A_{21} = A_{21}\,\text{TRIL}\,(A_{11})^{-T}$

$A_{22} = A_{22} - \text{TRIL}\,(A_{21} A_{21}^T)$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
PartitionDownDiagonal
     ( A, ATL, ATR,
          ABL, ABR, 0 );
while( ABR.Height() > 0 )
{
        RepartitionDownDiagonal
        ( ATL, /**/ ATR,        A00, /**/ A01, A02,
         /*************/        /****************/
                /**/            A10, /**/  A11, A12,
          ABL, /**/ ABR,        A20, /**/  A21, A22 );
        A21_VC_Star.AlignWith( A22 );
        A21_MC_Star.AlignWith( A22 );
        A21_MR_Star.AlignWith( A22 );
        //--------------------------------------------------------//
        A11_Star_Star = A11;
        internal::LocalChol( Lower, A11_Star_Star );
        A11 = A11_Star_Star;

        A21_VC_Star = A21;
        internal::LocalTrsm
        ( Right, Lower, ConjugateTranspose, NonUnit,
          (F)1, A11_Star_Star, A21_VC_Star );

        A21_MC_Star = A21_VC_Star;
        A21_MR_Star = A21_VC_Star;

        internal::LocalTriangularRankK
        ( Lower, ConjugateTranspose,
          (F)-1, A21_MC_Star, A21_MR_Star, (F)1, A22 );

        A21 = A21_MC_Star;
        //--------------------------------------------------------//
        A21_VC_Star.FreeAlignments();
        A21_MC_Star.FreeAlignments();
        A21_MR_Star.FreeAlignments();
        SlidePartitionDownDiagonal
        ( ATL, /**/ ATR,       A00, A01, /**/ A02,
                /**/           A10, A11, /**/ A12,
         /*************/       /****************/
          ABL, /**/ ABR,       A20, A21, /**/ A22 );
}
```

$A = L\,L^T$

Algorithm: $A := \text{CHOL\_BLK\_VAR3}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$A_{11} = \Gamma(A_{11})$$
$$A_{21} = A_{21}\,\text{TRIL}\,(A_{11})^{-T}$$
$$A_{22} = A_{22} - \text{TRIL}\,(A_{21}A_{21}^T)$$

Continue with

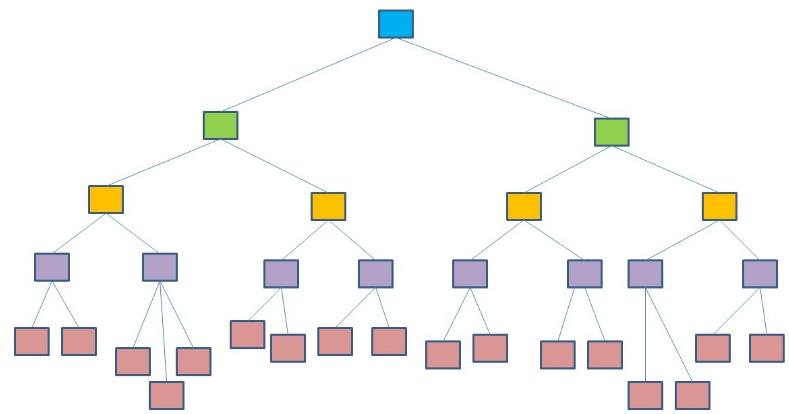
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
PartitionDownDiagonal
    ( A, ATL, ATR,
         ABL, ABR, 0 );
    while( ABR.Height() > 0 )
    {
        RepartitionDownDiagonal
        ( ATL, /**/ ATR,        A00, /**/ A01, A02,
         /*************/        /****************/
               /**/             A10, /**/  A11, A12,
          ABL, /**/ ABR,        A20, /**/  A21, A22 );
        A21_VC_Star.AlignWith( A22 );
        A21_MC_Star.AlignWith( A22 );
        A21_MR_Star.AlignWith( A22 );
        //------------------------------------------------------//
        A11_Star_Star = A11;
        internal::LocalChol( Lower, A11_Star_Star );
        A11 = A11_Star_Star;

        A21_VC_Star = A21;
        internal::LocalTrsm
        ( Right, Lower, ConjugateTranspose, NonUnit,
          (F)1, A11_Star_Star, A21_VC_Star );

        A21_MC_Star = A21_VC_Star;
        A21_MR_Star = A21_VC_Star;

        internal::LocalTriangularRankK
        ( Lower, ConjugateTranspose,
          (F)-1, A21_MC_Star, A21_MR_Star, (F)1, A22 );

        A21 = A21_MC_Star;
        //------------------------------------------------------//
        A21_VC_Star.FreeAlignments();
        A21_MC_Star.FreeAlignments();
        A21_MR_Star.FreeAlignments();
        SlidePartitionDownDiagonal
        ( ATL, /**/ ATR,       A00, A01, /**/ A02,
               /**/            A10, A11, /**/ A12,
         /*************/       /****************/
          ABL, /**/ ABR,       A20, A21, /**/ A22 );
    }
```

What does an expert need to do?

- Choose an algorithm
  - Cholesky has 3 basic algorithms



- Choose how to parallelize each operation
  - Which distributions are valid
  - Which distributions are efficient
- Choose alternate implementations for redistribution
  - E.g. can choose alternatives with intermediate distributions
- Optimize combinations of redistributions

DxT

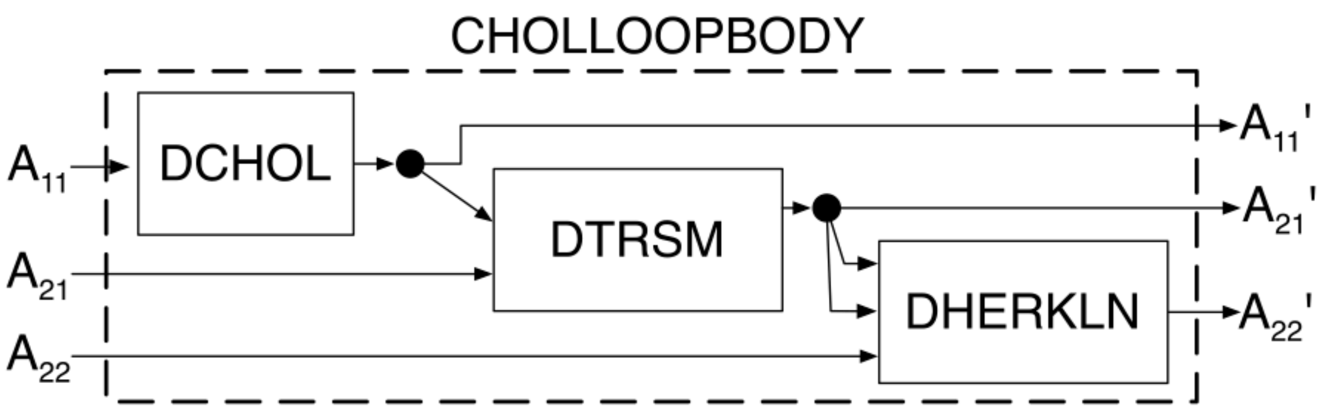
DESIGN BY TRANSFORMATION

Graphs

- Data-flow, directed acyclic graphs
  - Represents an algorithm or implementation

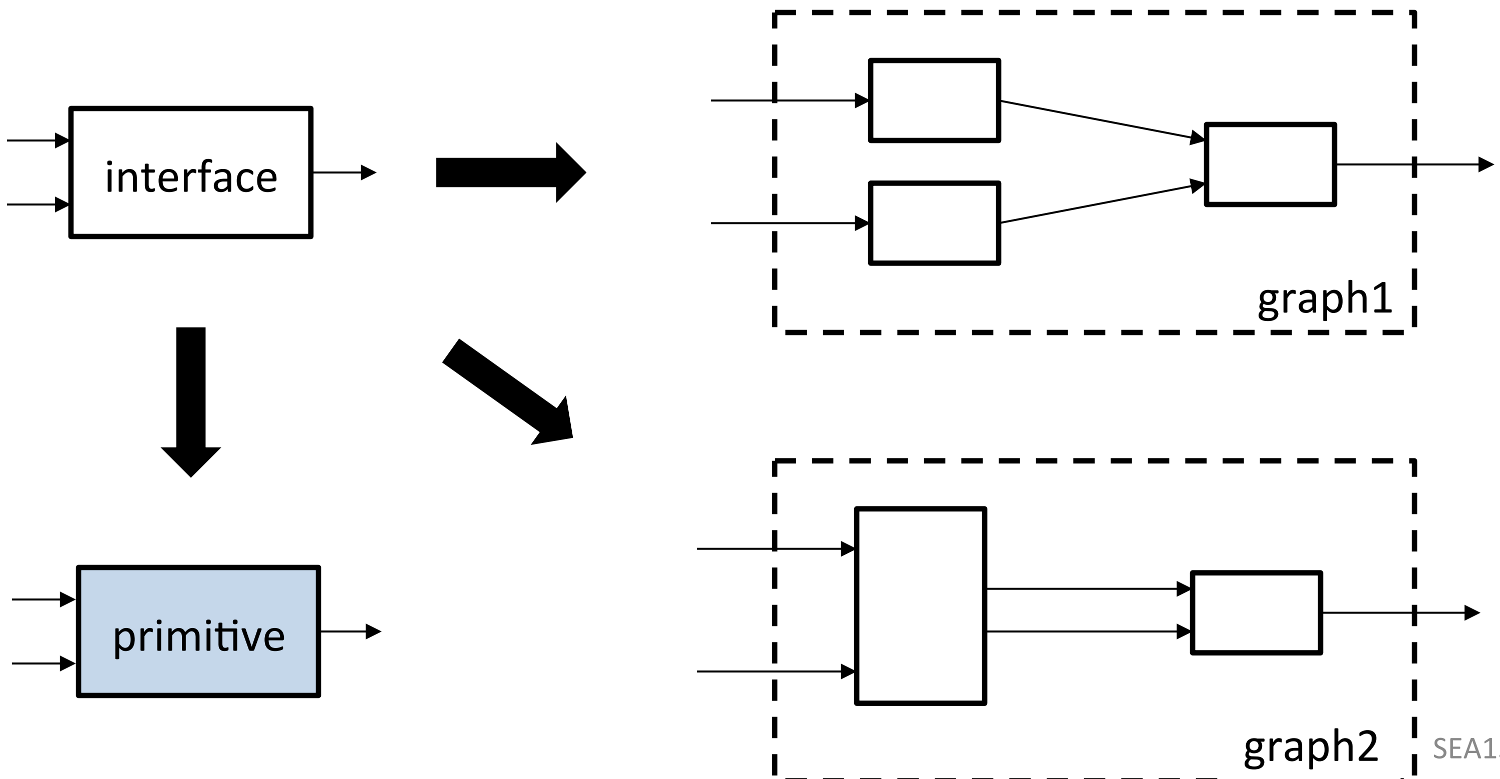
- A box or node represents an operation
  - An interface without implementation details
  - OR a primitive operation that maps to given code

View as DAG

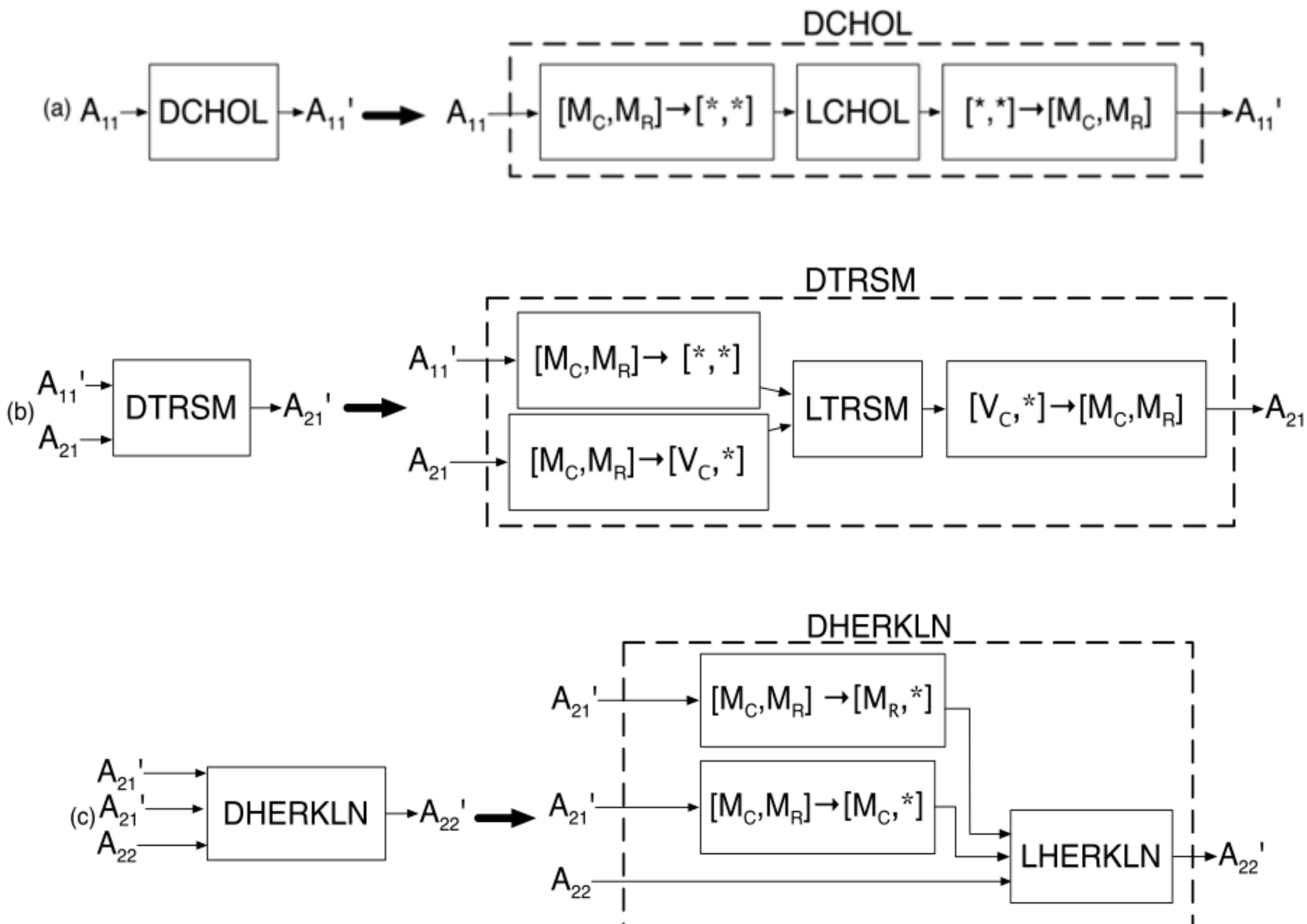


Transform with Implementations

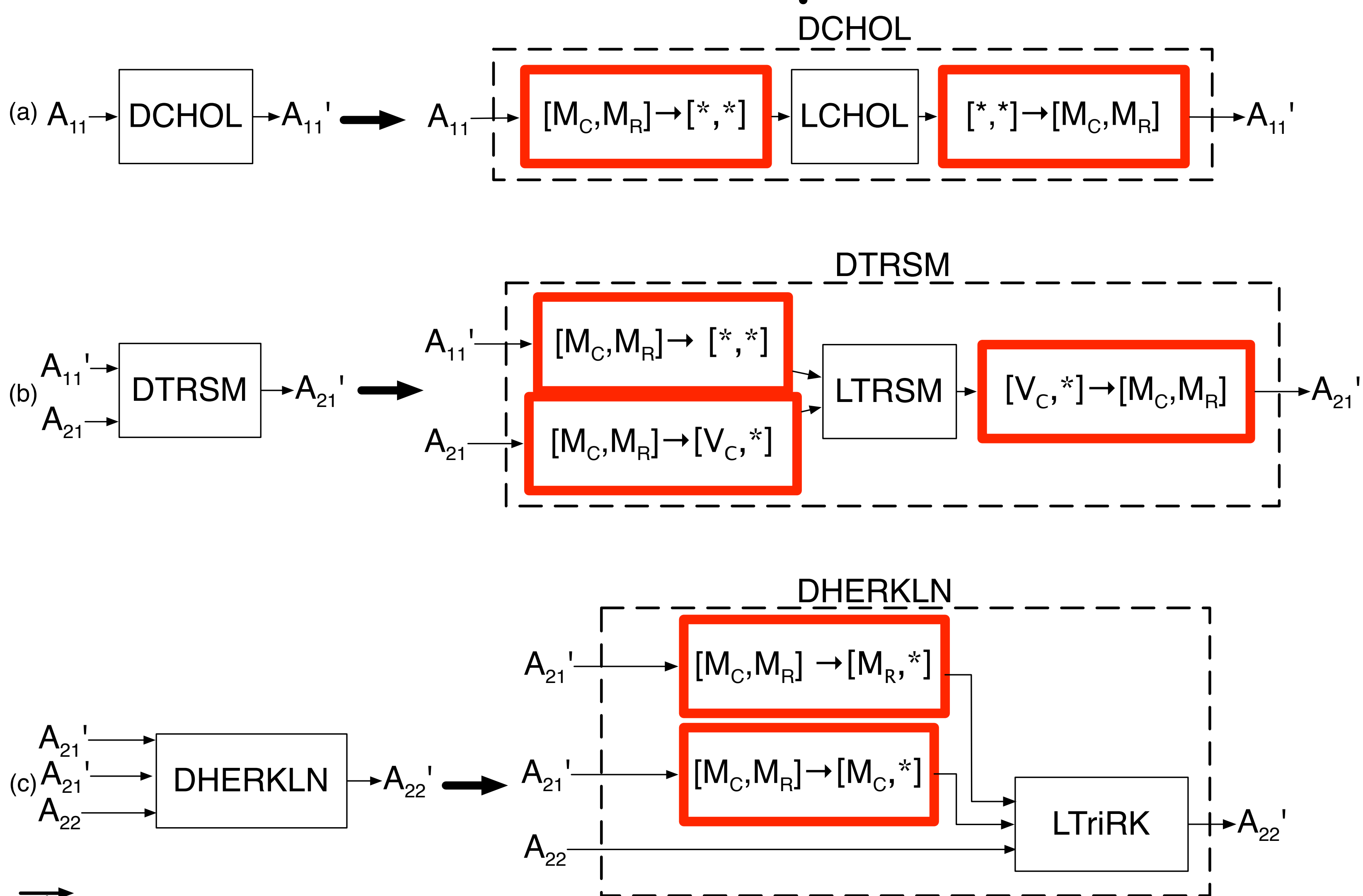
- **Refinements** replace a box without implementation details
  - Chooses a specific way to implement the box's functionality
  - E.g. choose an algorithmic variant or way to parallelize a loop body operation



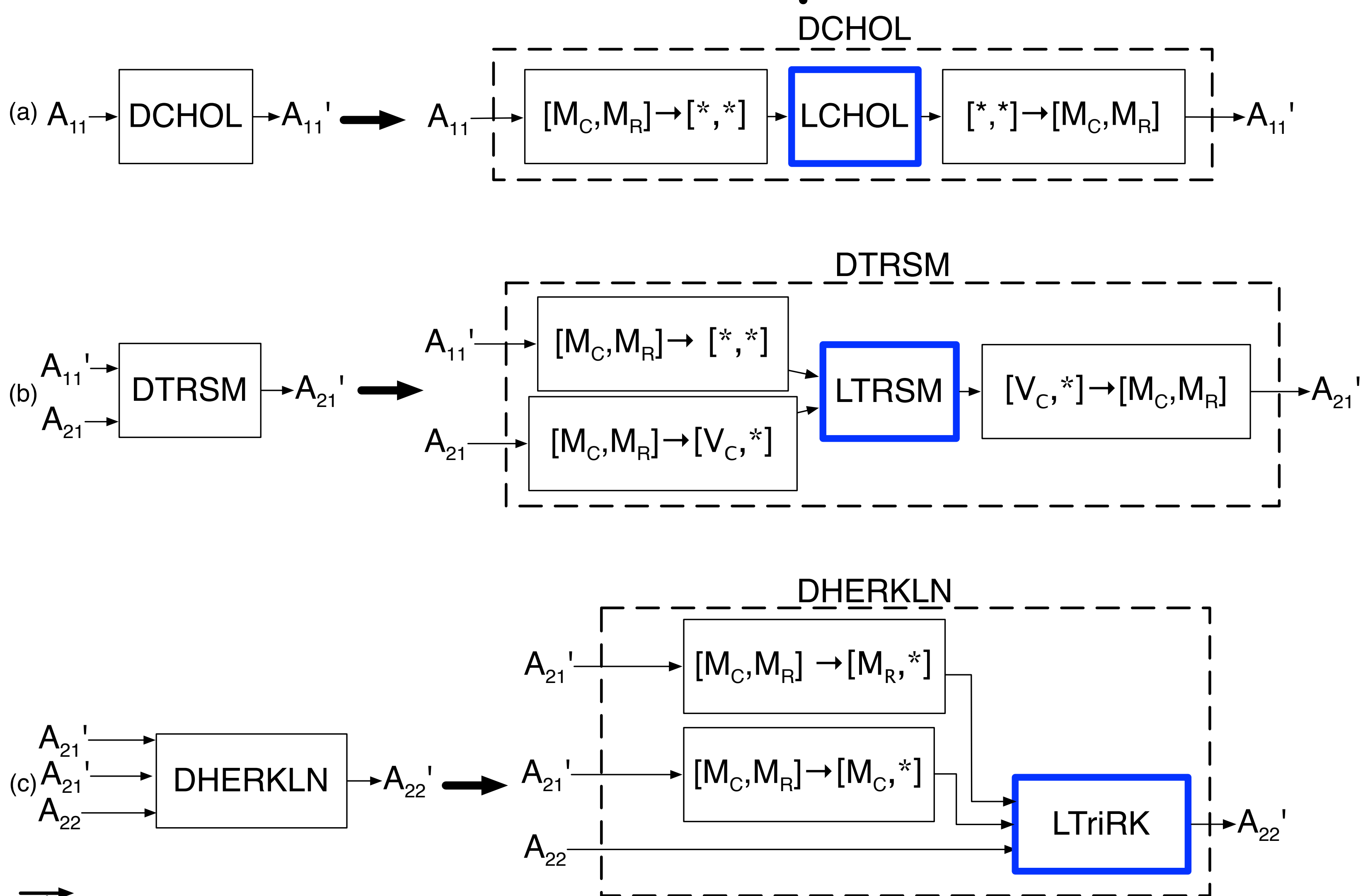
Transform with Implementations



Transform with Implementations

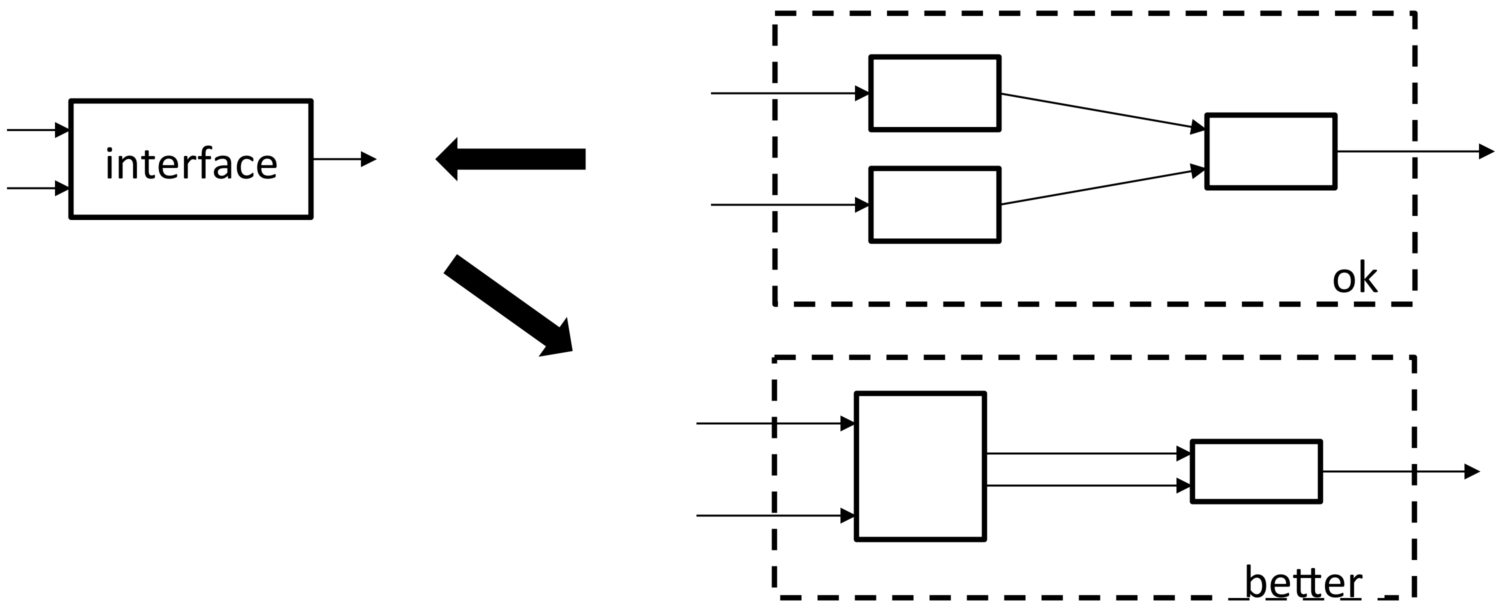


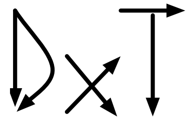
Transform with Implementations



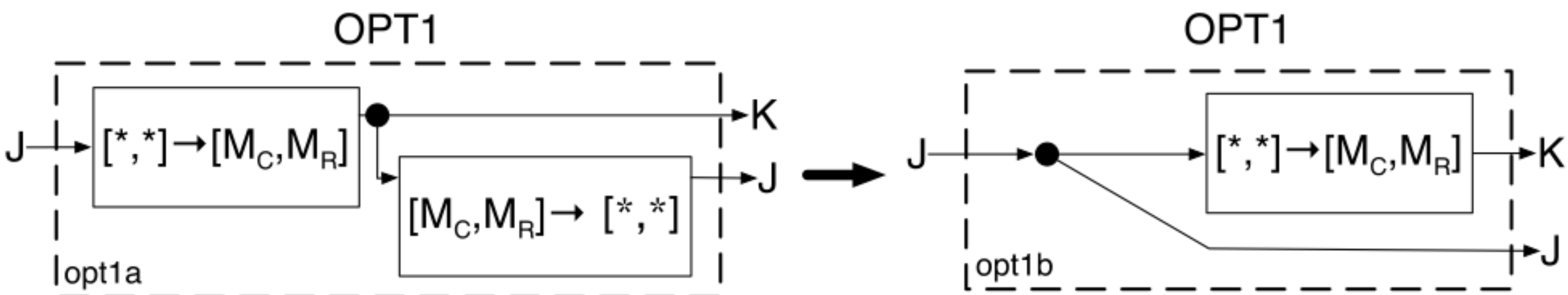
Transform to Optimize

- **Optimizations** replace a subgraph with another subgraph
  - Same functionality
  - A different way of implementing it
  - Optimizations are chained to improve performance





Transform to Optimize



DxT

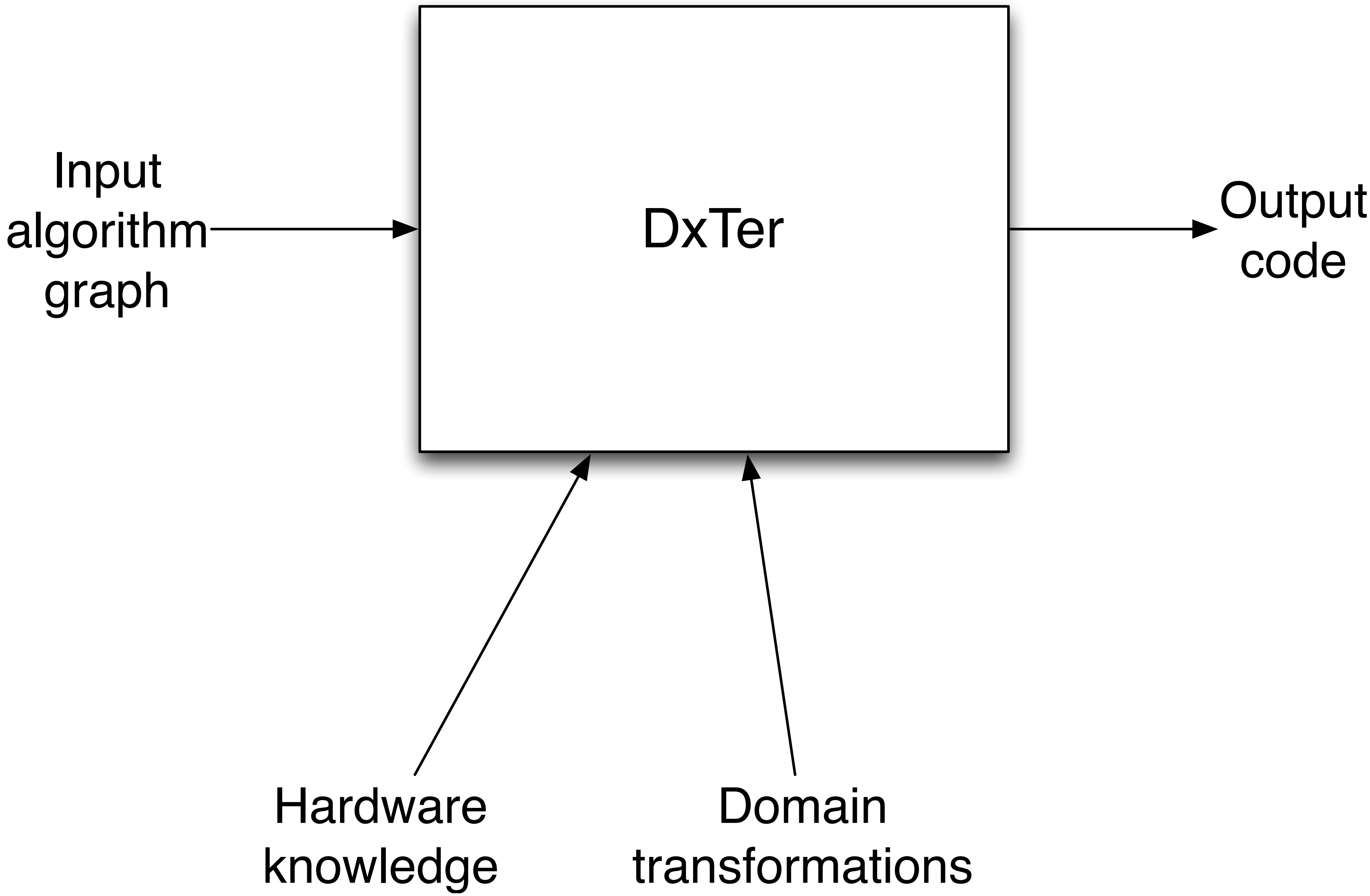
Transformations

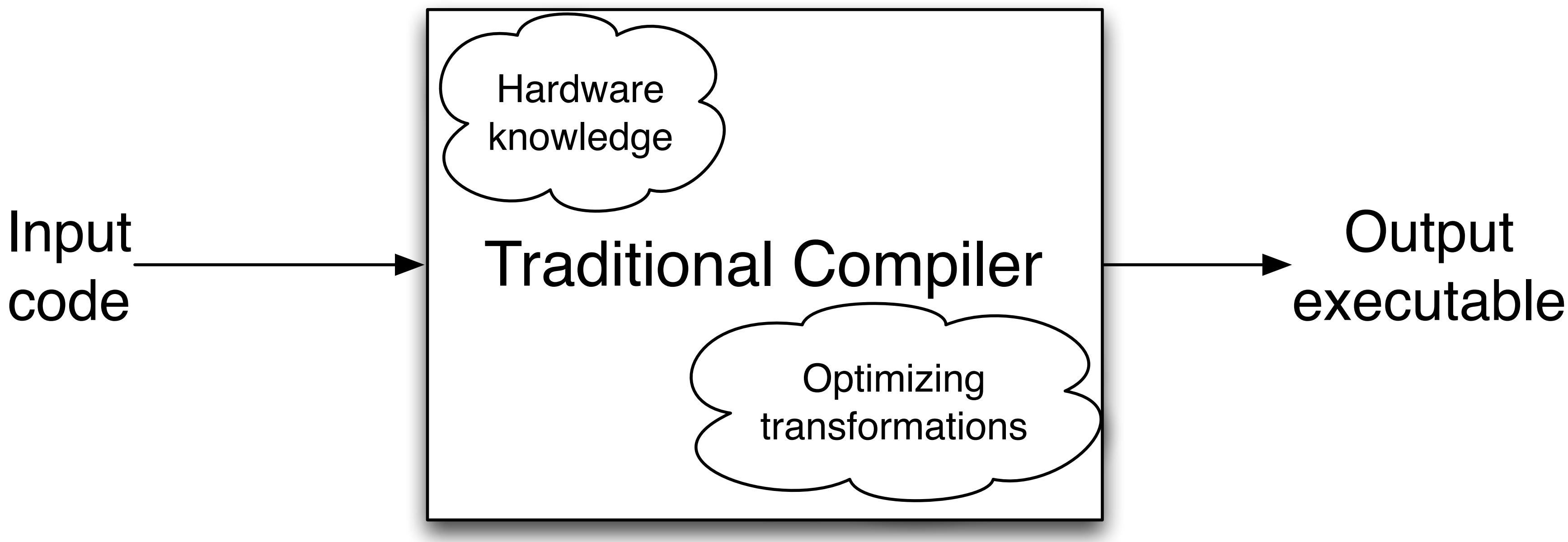
- We use correct transformations
- Final implementation is correct by construction

DxTer

- Prototype system

- Encode knowledge as transformations
- Input algorithm graph
- Applies all transformations it can
  - Combinatorial search space
- Outputs "best" implementation
  - Generates DSL code
  - E.g. targeting Elemental


Input
algorithm
graph
DxTer
Output
code
Hardware
knowledge
Domain
transformations


Input
code
Hardware
knowledge
Traditional Compiler
Optimizing
transformations
Output
executable

Cost Analysis

- DxTer estimates cost of all implementations
  - They're all valid implementations
  - Choose the best-performing
- Form of domain knowledge
- An expert manually coding has to estimate how good his/her implementation is

Cost Analysis

- For DLA, each box has a cost estimate
- First-order approximation
  - In terms of number of processes, problem size, communication and computation costs, etc.
- Using cost functions to mimic the heuristics experts use to make decisions
  - They're just as good as what an expert uses

Cost Analysis

- For DLA, each box has a cost estimate
- First-order approximation based on
  - Amount of data movement
  - Amount of computation
  - Rough estimate of cost of computation and communication
  - Number of processes
- Using cost functions to mimic the heuristics experts use to make decisions

Cost Analysis

Operation	Cost
`LocalChol` $(n \times n)$	$\gamma n^3/3$
`LocalTrsm` (Right, Lower, $n \times n$, $m \times n$)	γmnn
`A11_Star_Star = A11` $(m \times n)$	$\alpha\lceil \log_2 p \rceil + \beta \frac{p-1}{p} mn$
`A21_MC_Star = A21_VC_Star` $(m \times n)$	$\alpha\lceil \log_2 c \rceil + \beta \frac{c-1}{c} \frac{m}{r} n$

- Include machine-specific and problem-size parameters
- For now
  - First-order approximations
  - No running and timing necessary
  - Just meant to separate bad choices from good
- You can imagine
  - More complex cost functions
  - More complicated uses (e.g. multi-objective and/or hardware-software co-design)

RESULTS!

Level-3 BLAS

- Matrix-matrix operations

Matrix-matrix multiplication (Gemm)	$C := \alpha\, A * B + \beta\, C$
Triangular matrix multiply (Trmm)	$B := \alpha\, L * B$
Solve a triangular system of equations (Trsm)	$B := \alpha\, L^{-1} * B$
Hermitian matrix multiply (Hemm)	$C := \alpha\, A * B + \beta\, C$
Symmetric matrix multiply (Symm)	$C := \alpha\, A * B + \beta\, C$
Hermitian matrix rank-k update (Herk)	$C := \alpha\, A * A^H + \beta\, C$
Symmetric matrix rank-k update (Syrk)	$C := \alpha\, A * A^T + \beta\, C$
Hermitian matrix rank-2k update (Her2k)	$C := \alpha\, (A * B^H + B * A^H) + \beta\, C$
Symmetric matrix rank-2k update (Syrk)	$C := \alpha\, (A * B^T + B * A^T) + \beta\, C$

Basic Linear Algebra Subprograms

Operation	Versions needed	# Implementations Analyzed	DxTer vs. Expert
Gemm	12	378	Same or slightly better
Hemm	8	16,884	Same
Her2k	4	552,415	Same
Herk	4	1,252	Same
Symm	8	16,880	Same
Syr2k	4	295,894	Same
Syrk	4	1,290	Same
Trmm	16	3,352	Better algorithms
Trsm	16	1,012	Same, slightly better, or new code

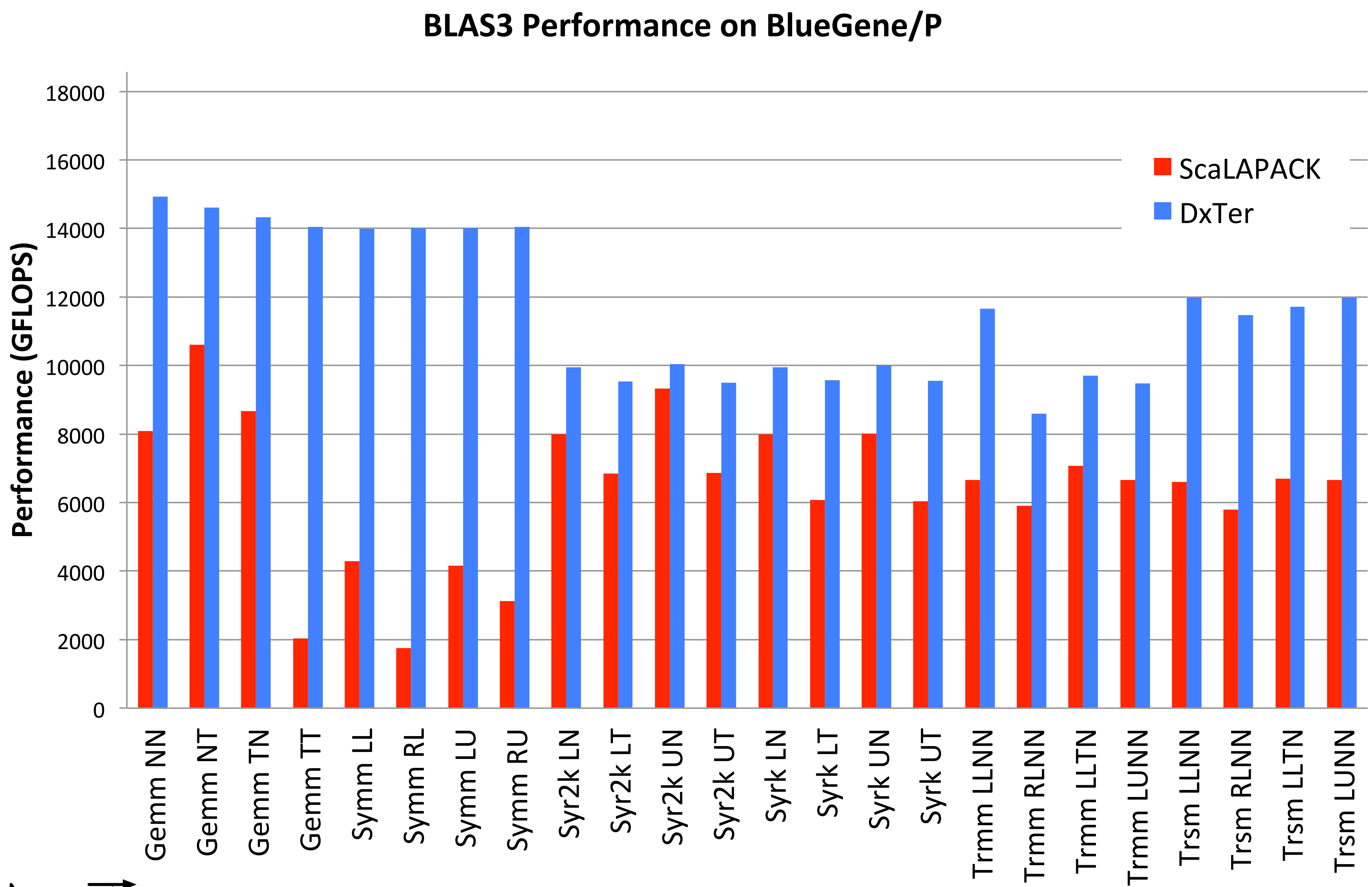
Transformations for the Level-3 BLAS

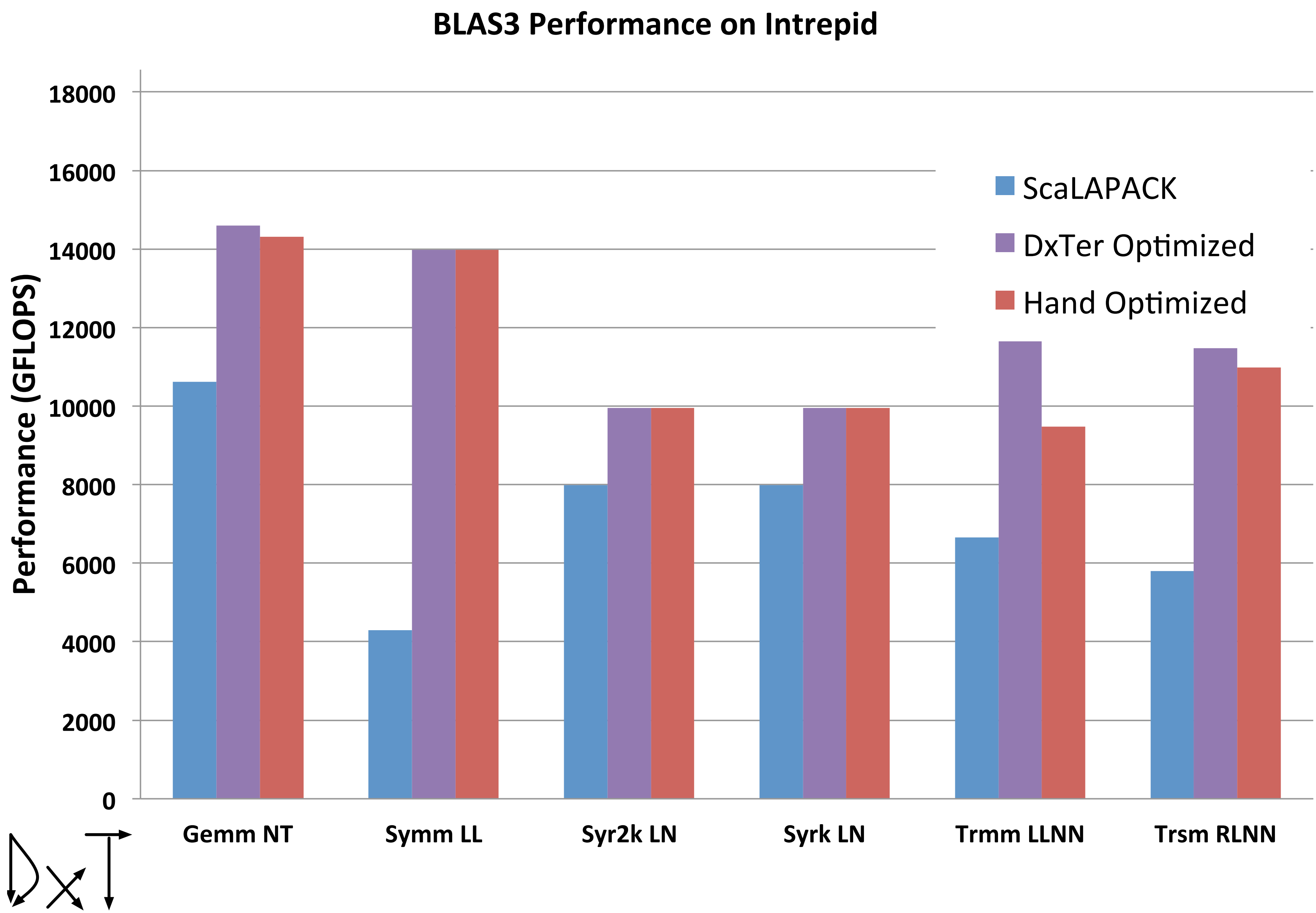
	Unique	Total
Algorithm refinements	19	30
Parallelizing refinements	14	31
Redistribution optimizations	38	780

DxT

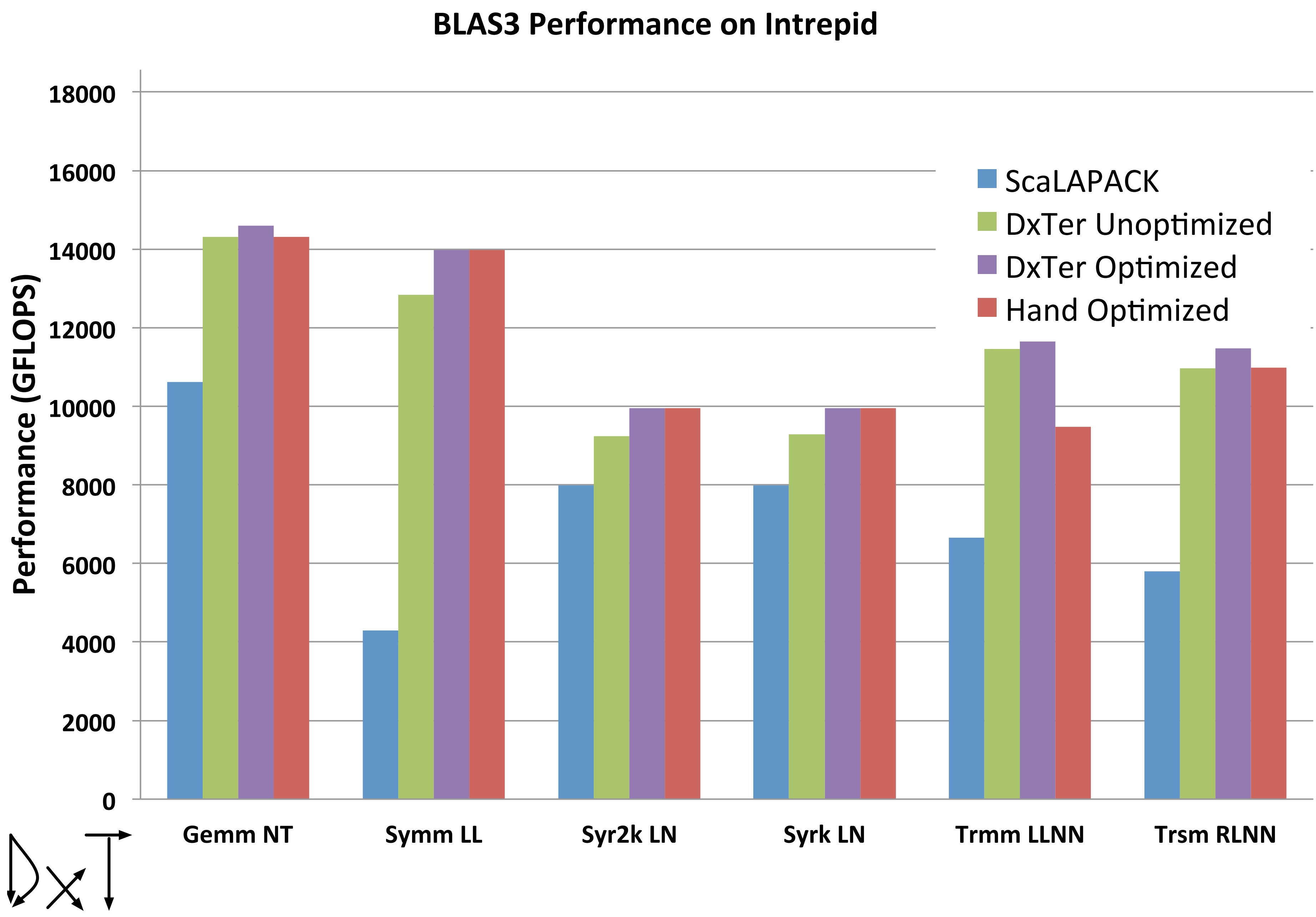
Performance Test

- Argonne's BlueGene/P machine Intrepid
- 8,192 cores
- Over 27 TFLOPS peak performance
- 2/3 of peak at top of graphs


BLAS3 Performance on BlueGene/P
Performance (GFLOPS)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
ScaLAPACK
DxTer
Gemm NN
Gemm NT
Gemm TN
Gemm TT
Symm LL
Symm RL
Symm LU
Symm RU
Syr2k LN
Syr2k LT
Syr2k UN
Syr2k UT
Syrk LN
Syrk LT
Syrk UN
Syrk UT
Trmm LLNN
Trmm RLNN
Trmm LLTN
Trmm LUNN
Trsm LLNN
Trsm RLNN
Trsm LLTN
Trsm LUNN



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


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

LET'S (AUTOMATICALLY) REAPPLY THAT KNOWLEDGE…

Algorithm: $A := L^{-1}AL^{-H}$(two-sided Trsm) and
$A := L^{H}AL$(two-sided Trmm)

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right)$

where A_{TL} and L_{TL} are 0×0.

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

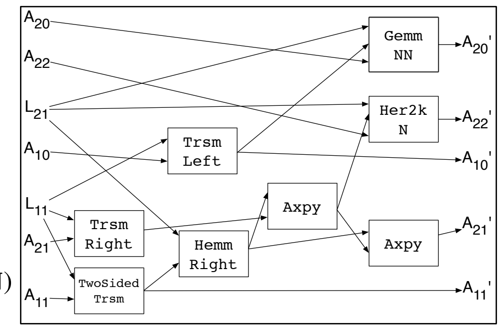
Repartition

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

where A_{11} **and** L_{11} **are** $b \times b$

Variant 4 for $L^{-1}AL^{-H}$

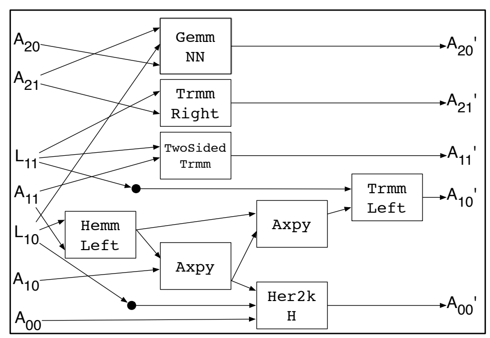
$A_{10} := L_{11}^{-1}A_{10}$ (Trsm Left)
$A_{20} := A_{20} - L_{21}A_{10}$ (Gemm NN)
$A_{11} := L_{11}^{-1}A_{11}L_{11}^{-H}$ (two-sided Trsm)
$Y_{21} := L_{21}A_{11}$ (Hemm Right)
$A_{21} := A_{21}L_{11}^{-H}$ (Trsm Right)
$A_{21} := A_{21} - \frac{1}{2}Y_{21}$ (Axpy)
$A_{22} := A_{22} - (L_{21}A_{21}^{H} + A_{21}L_{21}^{H})$ (Her2k N)
$A_{21} := A_{21} - \frac{1}{2}Y_{21}$ (Axpy)



Loop body graph input to DxTer for Trsm.

Variant 4 for $L^{H}AL$

$Y_{10} := A_{11}L_{10}$ (Hemm Left)
$A_{10} := A_{10} + \frac{1}{2}Y_{10}$ (Axpy)
$A_{00} := A_{00} + (A_{10}^{H}L_{10} + L_{10}^{H}A_{10})$ (Her2k H)
$A_{10} := A_{10} + \frac{1}{2}Y_{10}$ (Axpy)
$A_{10} := L_{11}^{H}A_{10}$ (Trmm Left)
$A_{11} := L_{11}^{H}A_{11}L_{11}$ (two-sided Trmm)
$A_{20} := A_{20} + A_{21}L_{10}$ (Gemm NN)
$A_{21} := A_{21}L_{11}$ (Trmm Right)



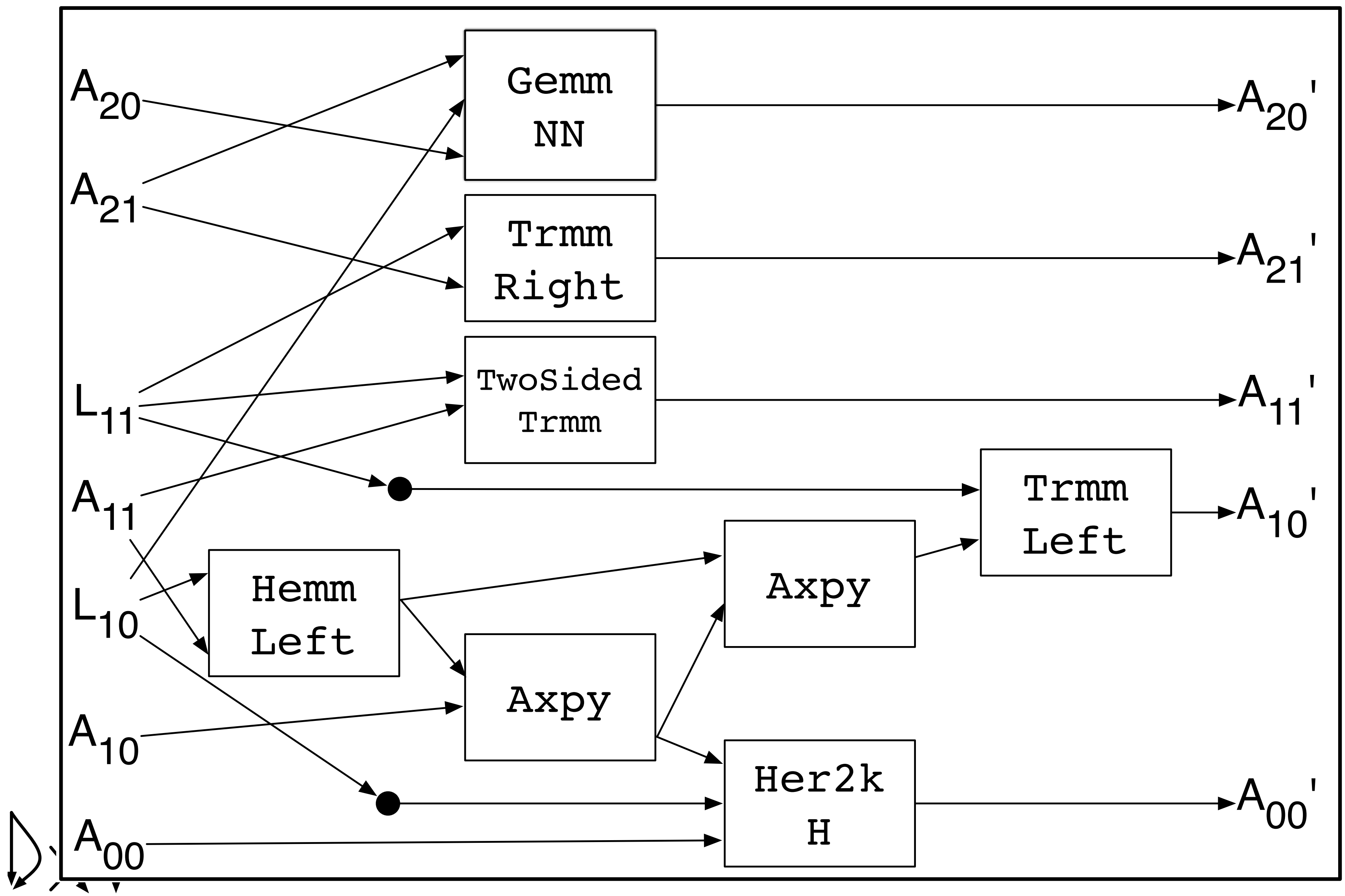
Loop body graph input to DxTer for Trmm.

Continue with

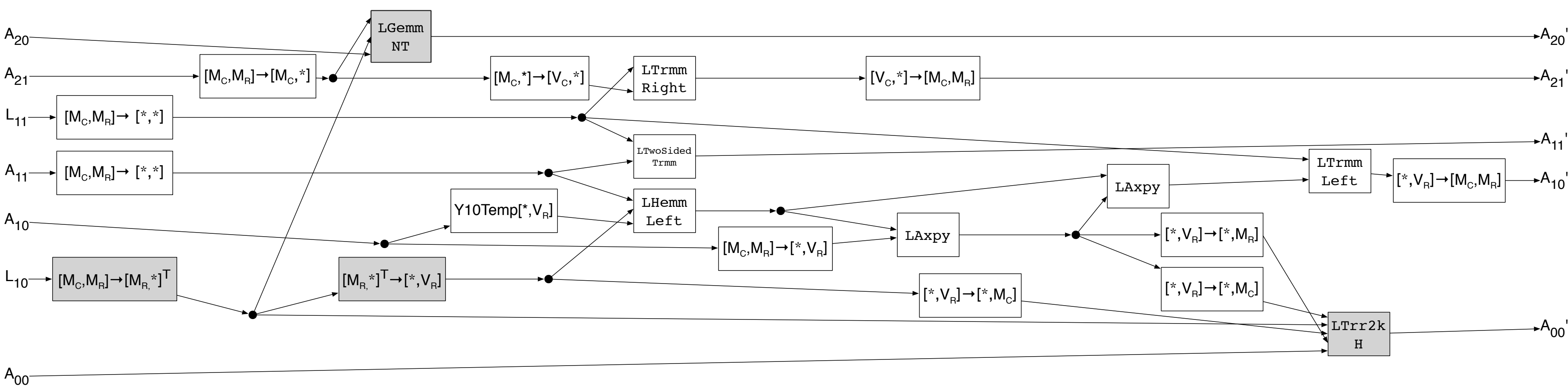
$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

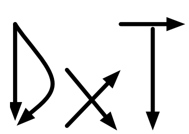
endwhile

Starting Graph

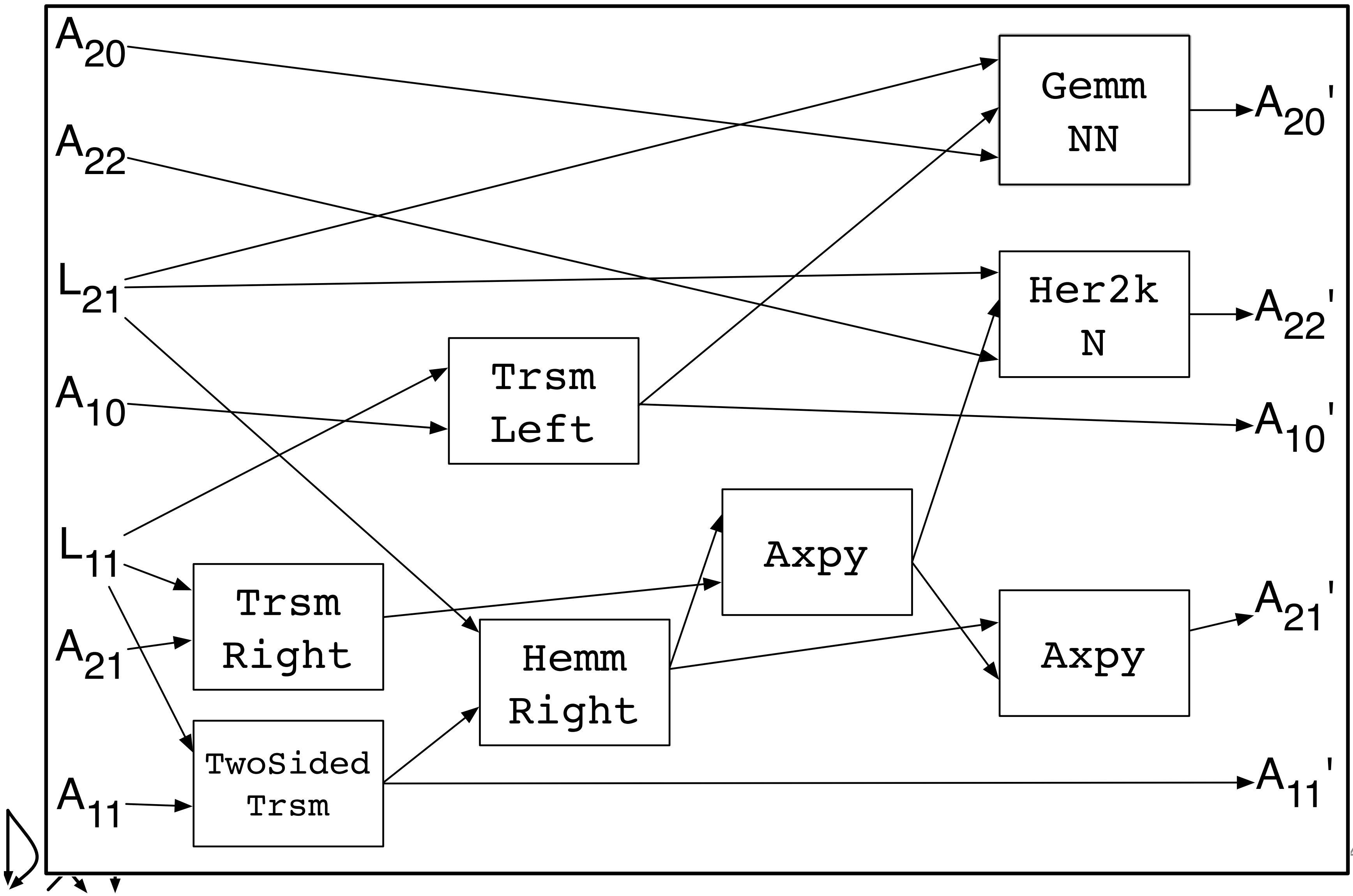


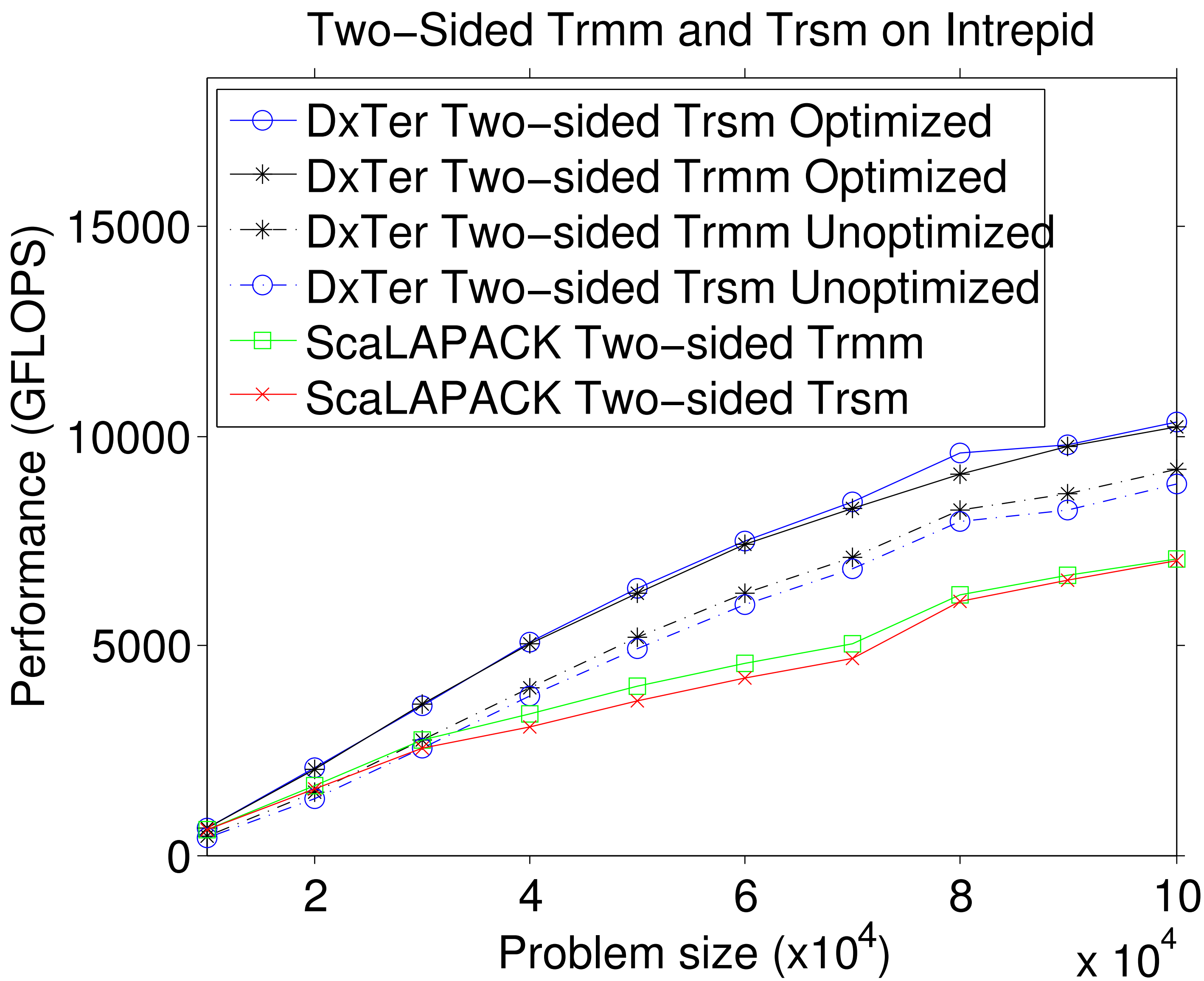
Final Implementation





The Other Input




Two−Sided Trmm and Trsm on Intrepid
DxTer Two−sided Trsm Optimized
DxTer Two−sided Trmm Optimized
DxTer Two−sided Trmm Unoptimized
DxTer Two−sided Trsm Unoptimized
ScaLAPACK Two−sided Trmm
ScaLAPACK Two−sided Trsm
Performance (GFLOPS)
0
5000
10000
15000
2
4
6
8
10
x 10^4
Problem size (x10^4)

Knowledge reuse!

- Many more operations implemented
  - Generated same or better than the expert
  - Generated correct code
  - Generated new operations, never optimized before (manual loop fusion is hard)

- Reused algorithm knowledge to generate code for sequential architectures
  - Just needed some additional sequential-specific knowledge
  - Target BLIS library as DSL

Related Work

- Spiral
- Built to Order (BTO) BLAS
- Tensor Contraction Engine (TCE)
- ATLAS / general autotuning

Some projects with similar goal at lower levels of stack

Ask me if you want to know more

Conclusion

- With the well-layered structure found in a modern distributed-memory DLA library, we can encode expert knowledge
  - Refinements to make implementation choices
  - Optimizations to improve performance
  - Cost estimates to choose "best" implementations

- We can automatically generate code that is the same as or better than an expert

- That knowledge can be reused automatically instead of forcing an expert to reapply it manually
  - Experts can forget (e.g. optimizations or entire algorithms)
  - A computer doesn't

Moving Forward

- We want to see DxT applied to other domains
  - Not to alleviate common user's burden
  - To enable the expert developer
- Surely, other domains have similar regularity
  - Relational query optimization (RQO) has done something similar to this for many years
  - There must be others
- We think domain software must be layered and well-designed
  - Expert knowledge is essential
  - Any ideas?

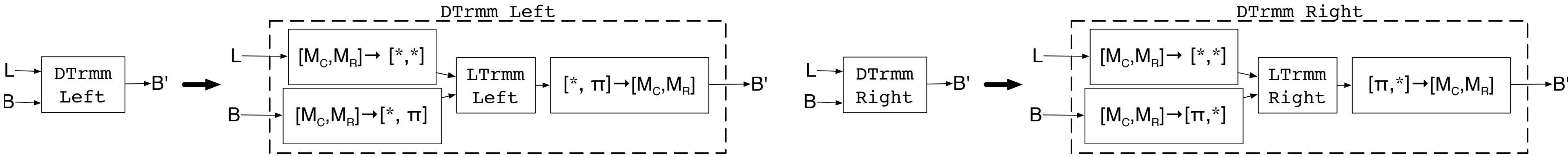
Thanks to…

- Sandia fellowship and NSF Graduate Research Fellowship under grant DGE-1110007
- NSF grants CCF-0917167 and OCI-1148125
- We used resources of the Argonne Leadership Computing Facility at Argonne National Lab, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357
- We are greatly indebted to Jack Poulson

Questions?

bamarker@cs.utexas.edu

www.cs.utexas.edu/~bamarker




```
void DistTrmmToLocalTrmm::Apply(Poss *poss, Node *node) const
{
  Trmm *trmm = (Trmm*)node;
  RedistNode *node1 = new RedistNode(D_STAR_STAR);
  RedistNode *node2 = new RedistNode(trmm->m_side == LEFT ? m_leftType: m_rightType);
  Trmm *node3 = new Trmm(trmm->m_side, trmm->m_tri,
                                        trmm->m_trans, trmm->m_coeff, trmm->m_type);
  RedistNode *node4 = new RedistNode(D_MC_MR);
  node1->AddInput(node->Input(0),node->InputConnNum(0));
  node2->AddInput(node->Input(1),node->InputConnNum(1));
  node3->AddInput(node1,0);
  node3->AddInput(node2,0);
  node4->AddInput(node3,0);
  poss->AddNodes(4, node1, node2, node3, node4);
  trmm->RedirectChildren(node4,0);
  trmm->m_poss->DeleteChildAndCleanUp(trmm);
}
```

Cost Analysis

Operation	Cost
LocalChol $(n \times n)$	$\gamma n^3/3$
LocalTrsm (Right, Lower, $n \times n$, $m \times n$)	γmnn
A11_Star_Star = A11 $(m \times n)$	$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} mn$
A21_MC_Star = A21_VC_Star $(m \times n)$	$\alpha \lceil \log_2 c \rceil + \beta \frac{c-1}{c} \frac{m}{r} n$

- Include machine-specific and problem-size parameters
- For now
  - First-order approximations
  - No running and timing necessary
  - Just meant to separate bad choices from good
- You can imagine
  - More complex cost functions
  - More complicated uses (e.g. multi-objective and/or hardware-software co-design)

Related Work

- Auto-tuning
  - Attempt to generate/choose best code for particular architecture
  - Sometimes chooses from a handful of algorithmic options
  - Tweak parameters
  - Explore space and run potential implementations
  - Often misses the optimal because it's only tweaking parameters
- "Automated Empirical Optimization of Software and the ATLAS project" by R. Clint Whaley, Antoine Petitet and Jack Dongarra. Parallel Computing, 27(1-2):3-35, 2001.

Related Work

- SPIRAL
  - Low-level kernels
    - Primarily digital signal processing (DSP)
    - Now moving into DLA
  - Compact mathematical notation
    - Re-writes for equivalent operations
  - Runtimes are small, so uses on-line learning techniques to find best implementations
- Markus Püschel et al. SPIRAL: Code Generation for DSP TransformsProceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation," Vol. 93, No. 2, 2005, pp. 232-275

Related Work

- Tensor Contraction Engine
  - One type of operation
  - Optimizes for space and time complexity
  - All about loop transformations
  - DxT could be used for tensor contractions, but TCE can't be used for DLA
- Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, A. Sibiryakov. Proceedings of the IEEE, Vol. 93, No. 2, February 2005, pp. 276-292.

Related Work

- Built to Order (BTO) BLAS
  - Automatically generates code for algorithms using level-1 and level-2 BLAS operations
  - Focuses on shared memory
  - Unique algorithm representation allowing for search using genetic algorithm
- Geoffrey Belter, Ian Karlin, Elizabeth Jessup, Jeremy Siek. Automating the Generation of Composed Linear Algebra Kernels. In SC09: the International Conference on High Performance Computing, Networking, Storage, and Analysis. November, 2009.