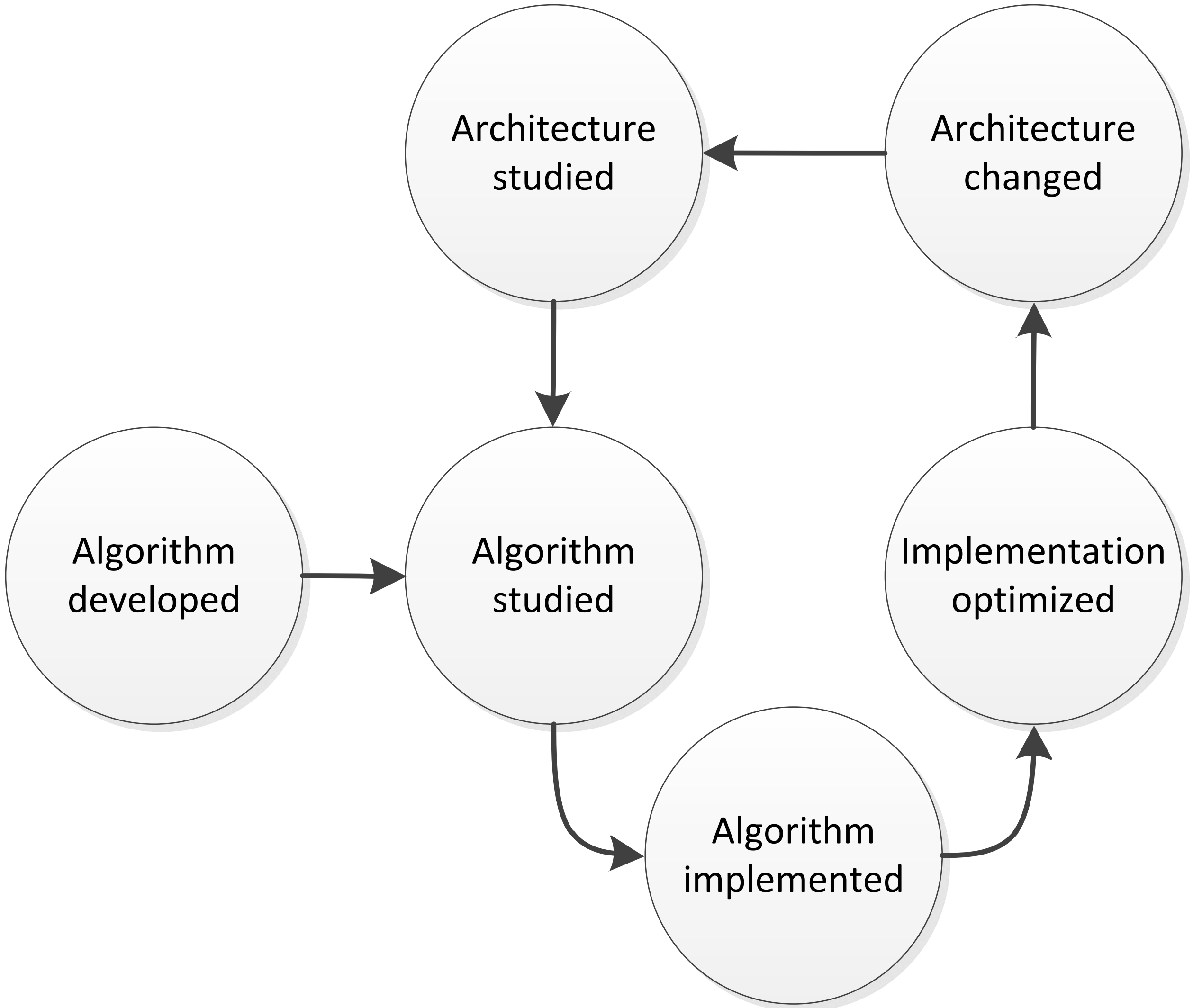
How a Domain-Specific Language Enables the Automation of Optimized Code for Dense Linear Algebra

DxT – Design by Transformation

Bryan Marker, Don Batory,

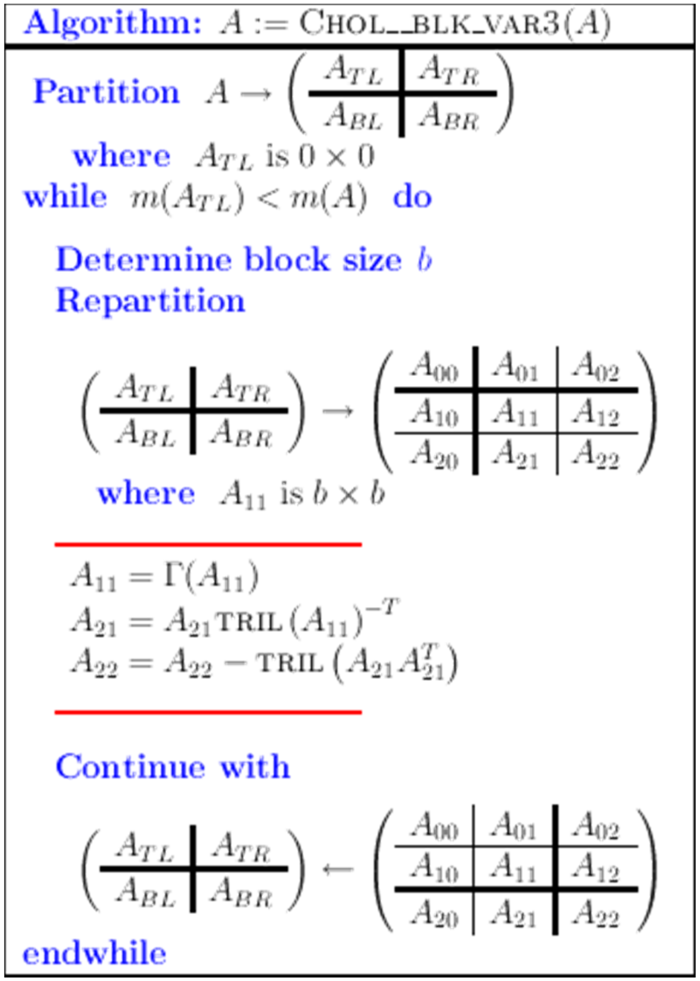
Jack Poulson, Robert van de Geijn

The Development Cycle



Why?

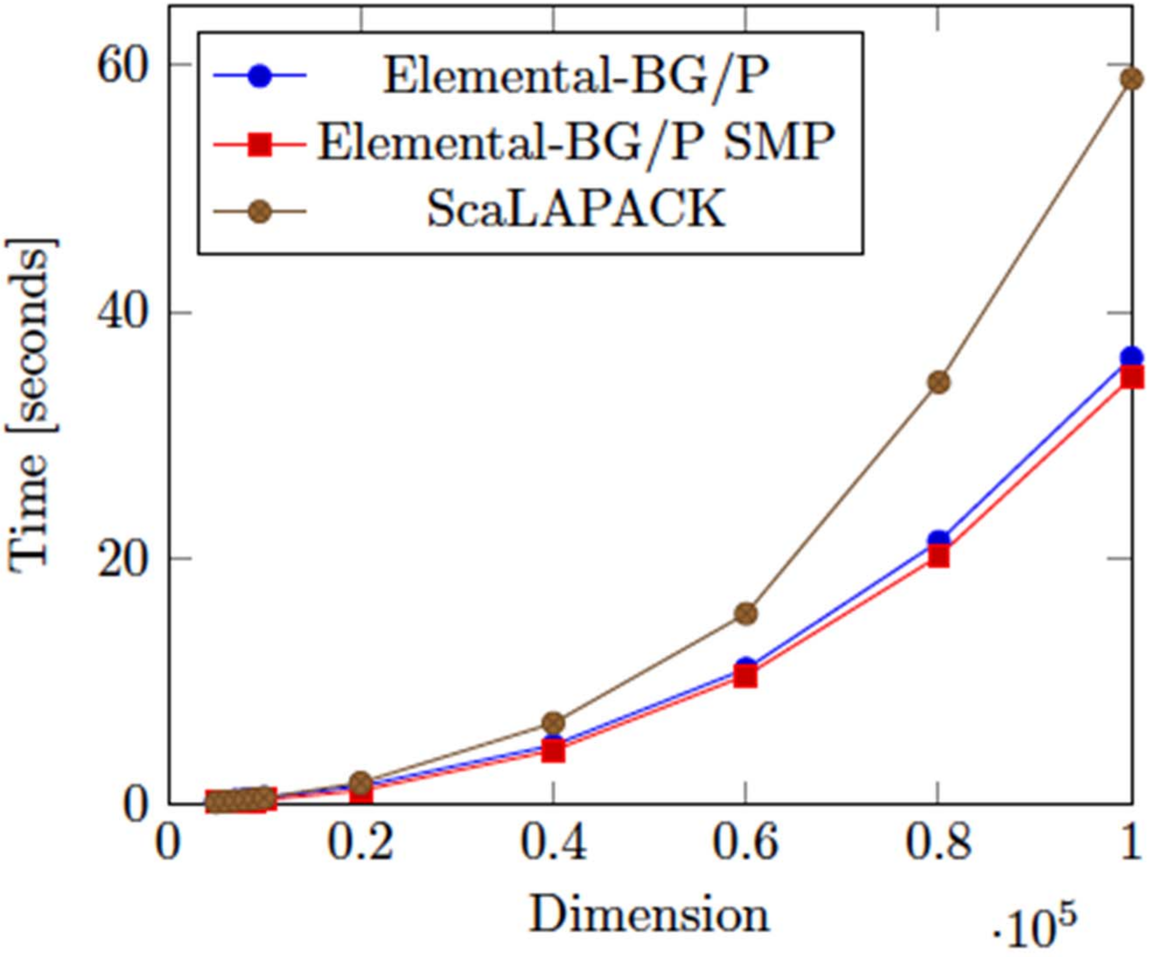
- We already modularize
  - Functions, libraries, features
  - You don’t inline re-used code, you put it in a function
  - Develop DSLs to reduce redundant development for domains
- We still encode architecture/algorithm specifics
  - Can’t re-use next time around
  - Doable a few times, not thousands
  - Manually break through layers to optimize
- Why don’t we encode reusable, high-level knowledge?
  - About algorithms, operations, or architectures



```
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 $0 \times 0$
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;
    lapack::internal::LocalChol( Lower, A11_Star_Star );
    A11 = A11_Star_Star;

    A21_VC_Star = A21;
    blas::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;

    // (A21^T[* ,MC])^T A21^H[* ,MR] = A21[MC,* ] A21^H[* ,MR]
    //                         = (A21 A21^H)[MC,MR]
    blas::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 );
  }
```

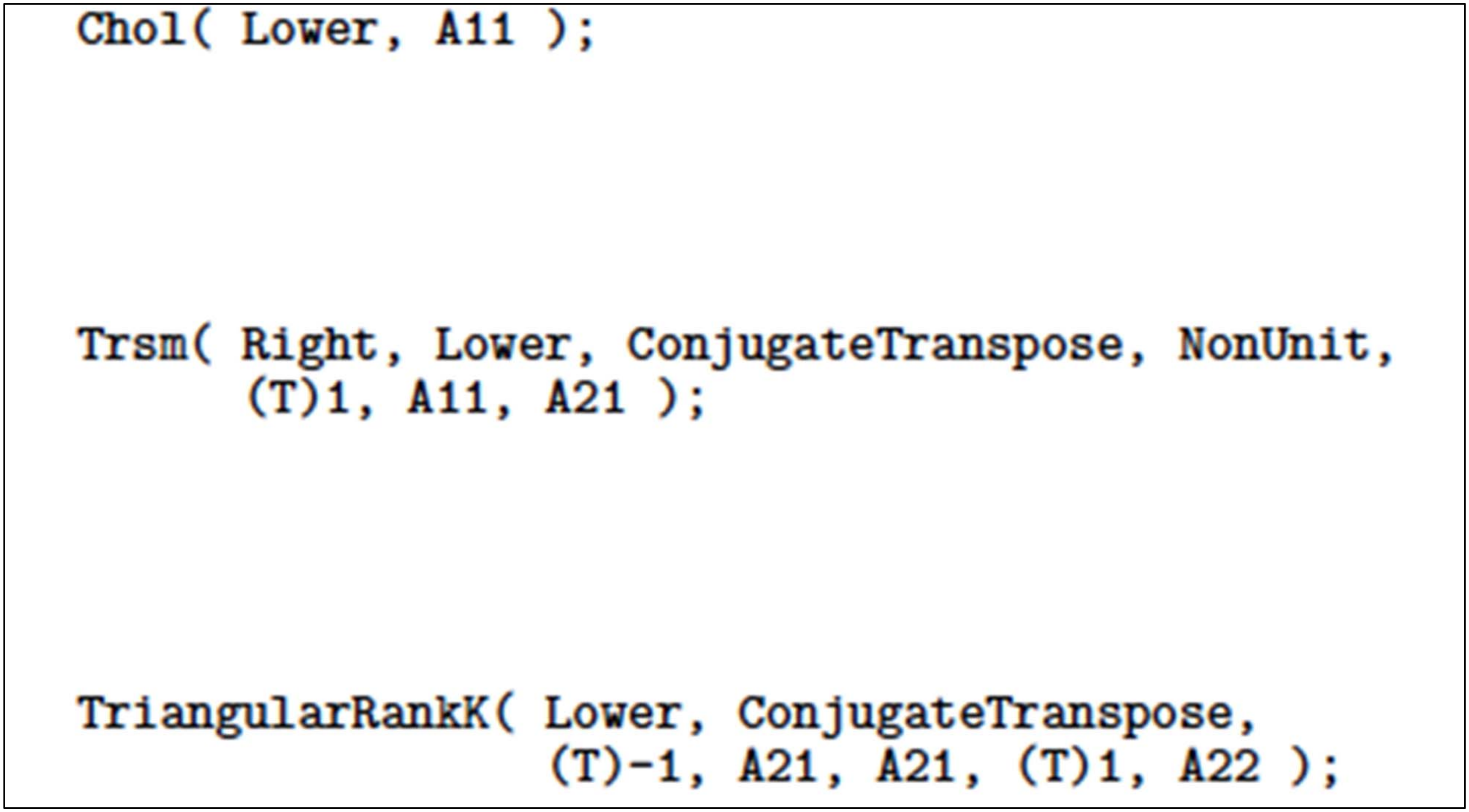
Performance of Elemental on 8192 cores



What Does an Expert Do?

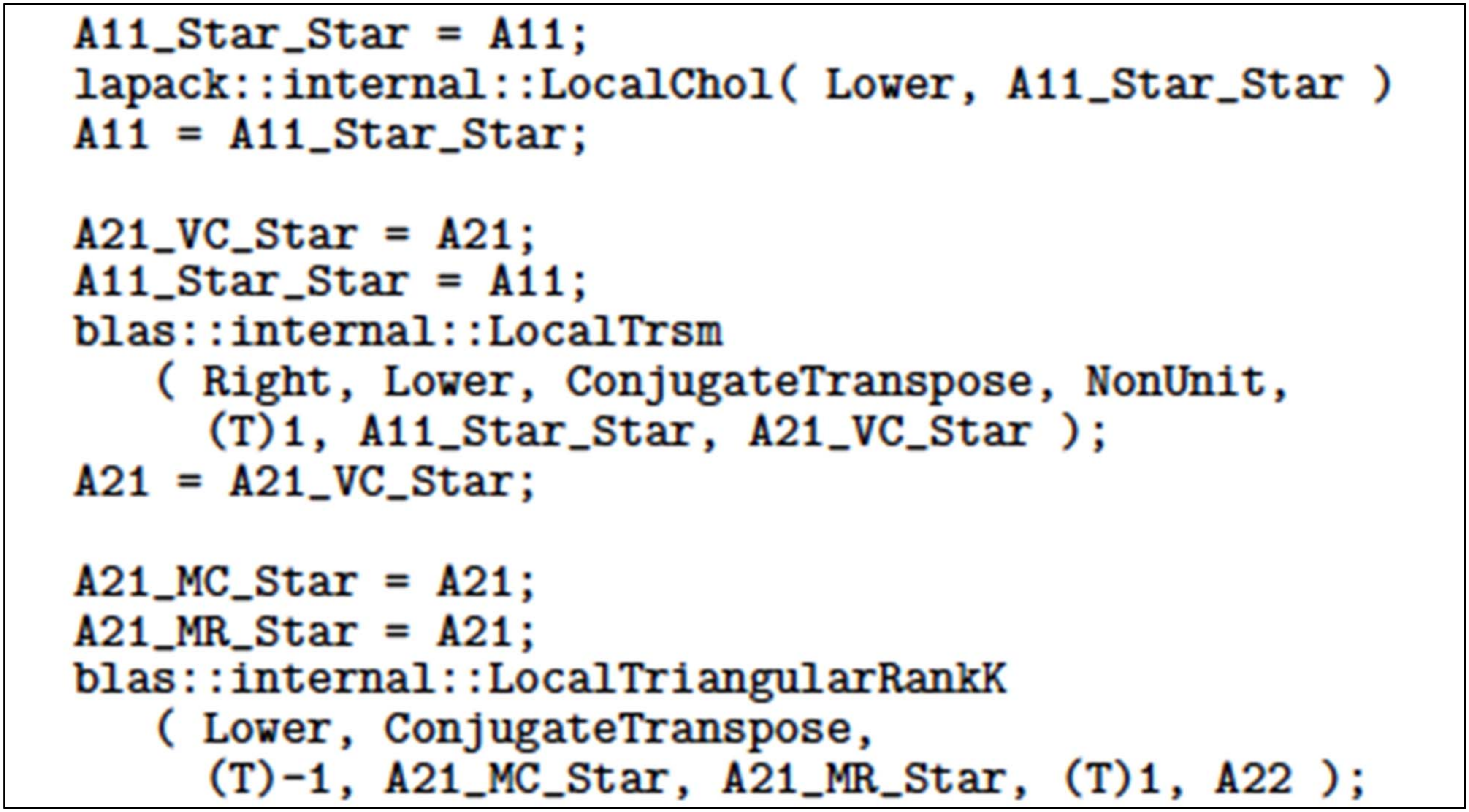
- Starts with algorithm
- Chooses architecture-specific implementation
  - Break through abstraction boundaries (when brave enough)
  - In-line code to expose details and inefficiencies
- Optimizes the code
- Transforms from algorithm to high-performance code
- Uses a DSL (when possible)
  - Abstract functionality into DSL
  - Maintain structure in code
  - Make transformations common across algorithms

Start with the algorithm



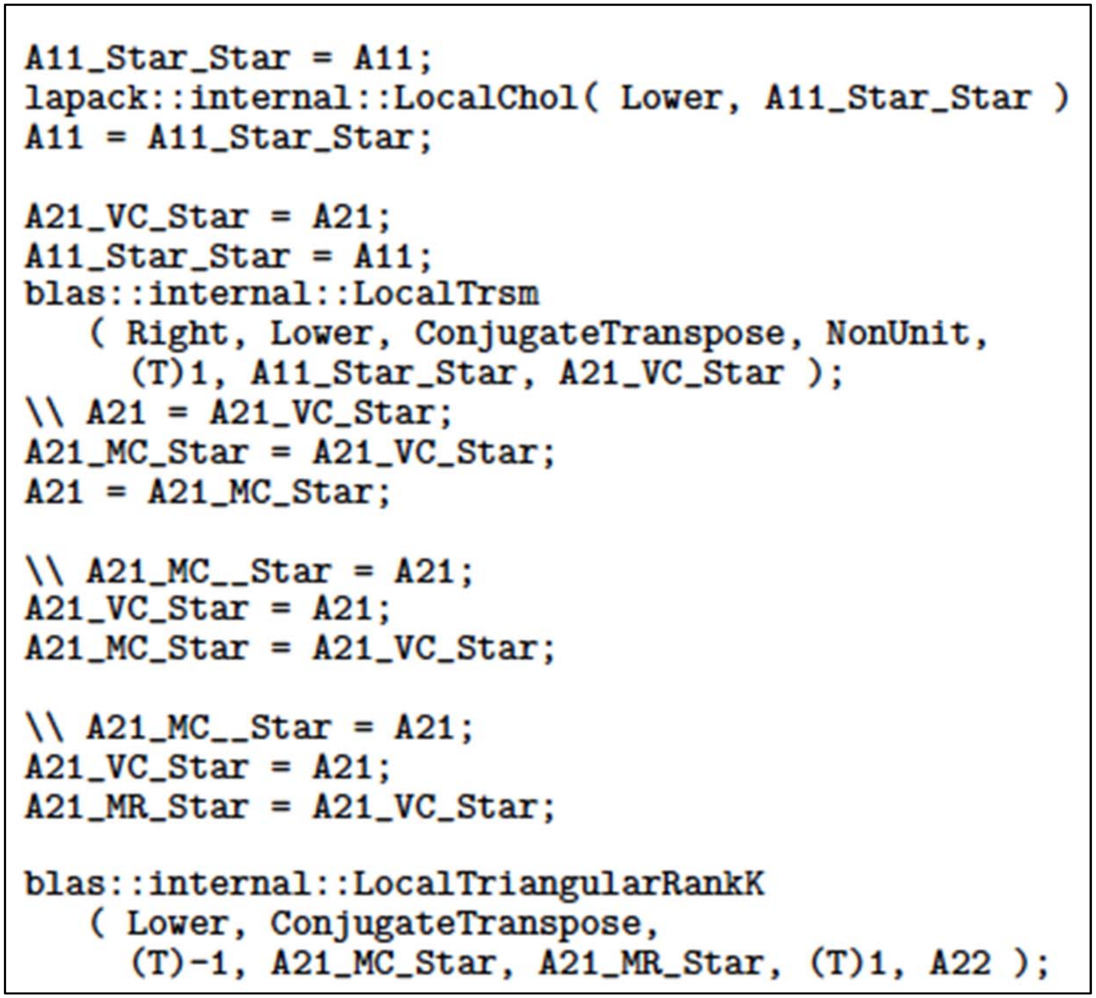
```
Chol( Lower, A11 );



Trsm( Right, Lower, ConjugateTranspose, NonUnit,
      (T)1, A11, A21 );



TriangularRankK( Lower, ConjugateTranspose,
                 (T)-1, A21, A21, (T)1, A22 );
```

Inline Some Parallelization Choices



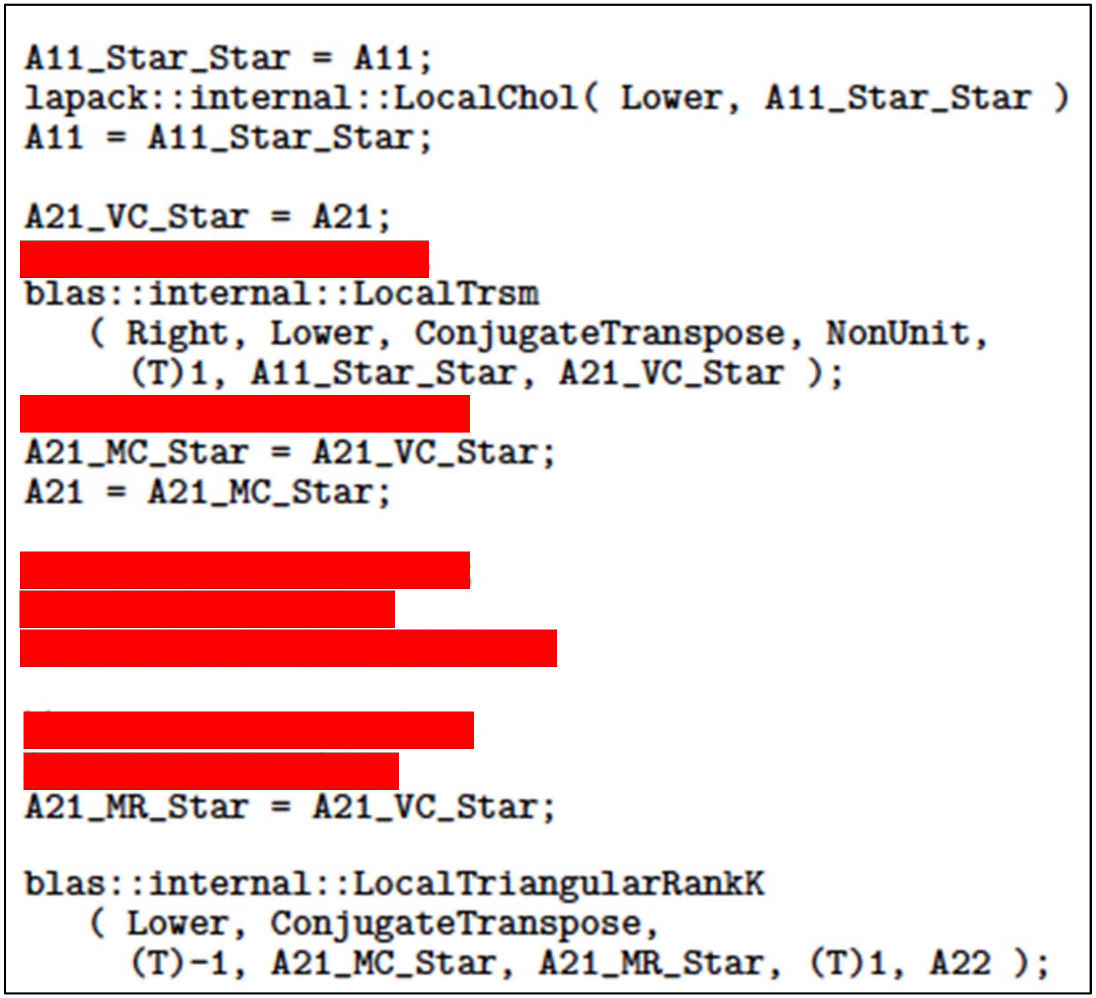
```
A11_Star_Star = A11;
lapack::internal::LocalChol( Lower, A11_Star_Star )
A11 = A11_Star_Star;

A21_VC_Star = A21;
A11_Star_Star = A11;
blas::internal::LocalTrsm
   ( Right, Lower, ConjugateTranspose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );
A21 = A21_VC_Star;

A21_MC_Star = A21;
A21_MR_Star = A21;
blas::internal::LocalTriangularRankK
   ( Lower, ConjugateTranspose,
     (T)-1, A21_MC_Star, A21_MR_Star, (T)1, A22 );
```

Inline Some Redistribution Choices



```
A11_Star_Star = A11;
lapack::internal::LocalChol( Lower, A11_Star_Star )
A11 = A11_Star_Star;

A21_VC_Star = A21;
A11_Star_Star = A11;
blas::internal::LocalTrsm
   ( Right, Lower, ConjugateTranspose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );
\\ A21 = A21_VC_Star;
A21_MC_Star = A21_VC_Star;
A21 = A21_MC_Star;

\\ A21_MC__Star = A21;
A21_VC_Star = A21;
A21_MC_Star = A21_VC_Star;

\\ A21_MC__Star = A21;
A21_VC_Star = A21;
A21_MR_Star = A21_VC_Star;

blas::internal::LocalTriangularRankK
   ( Lower, ConjugateTranspose,
     (T)-1, A21_MC_Star, A21_MR_Star, (T)1, A22 );
```

Optimize



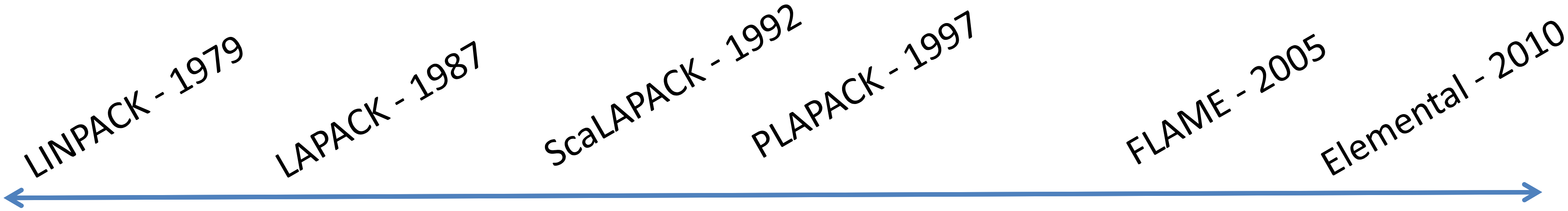
```
A11_Star_Star = A11;
lapack::internal::LocalChol( Lower, A11_Star_Star )
A11 = A11_Star_Star;

A21_VC_Star = A21;
[redacted]
blas::internal::LocalTrsm
   ( Right, Lower, ConjugateTranspose, NonUnit,
     (T)1, A11_Star_Star, A21_VC_Star );
[redacted]
A21_MC_Star = A21_VC_Star;
A21 = A21_MC_Star;

[redacted]

[redacted]
A21_MR_Star = A21_VC_Star;

blas::internal::LocalTriangularRankK
   ( Lower, ConjugateTranspose,
     (T)-1, A21_MC_Star, A21_MR_Star, (T)1, A22 );
```

Automate the Process!

- For operations in DSL, give implementations options
  - Architecture-dependent
  - Code an expert would inline
- Give transformations to optimize patterns found in DSL
- Transform just like an expert
- Use Model-Drive Engineering (MDE)
- Design by Transformation (DxT)

DSLs Enable This

- Many years of software engineering efforts
- DSLs provide well-layered and abstracted codes
- DSLs define and limit set of operations to be performed
- DSL make it easier to see common transformations in domain



LINPACK - 1979

LAPACK - 1987

ScaLAPACK - 1992

PLAPACK - 1997

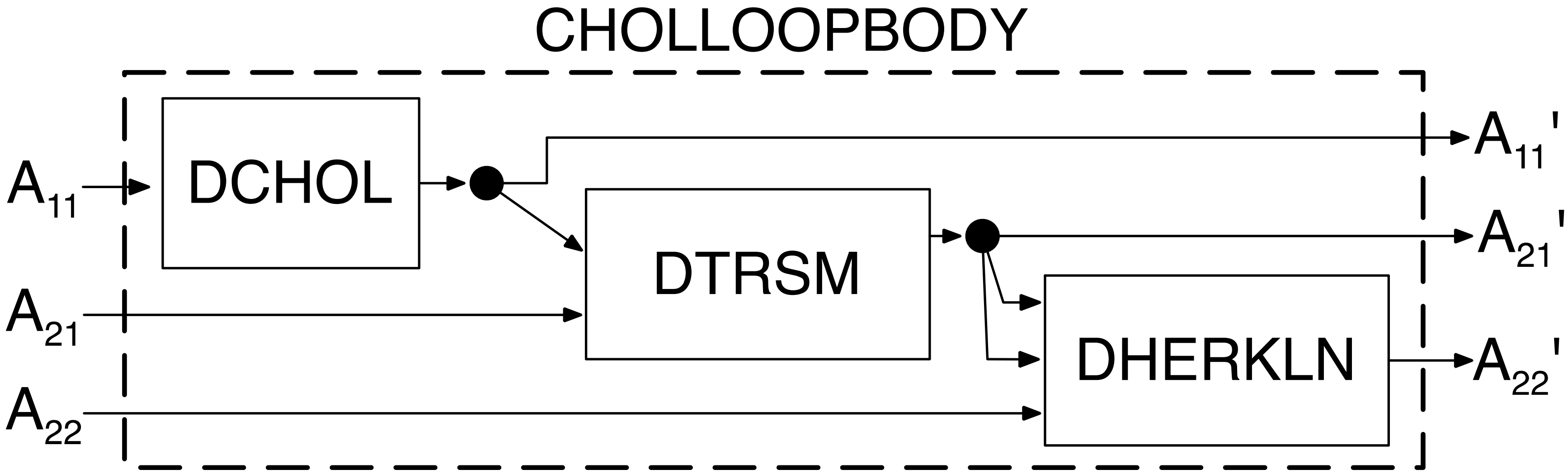
FLAME - 2005

Elemental - 2010

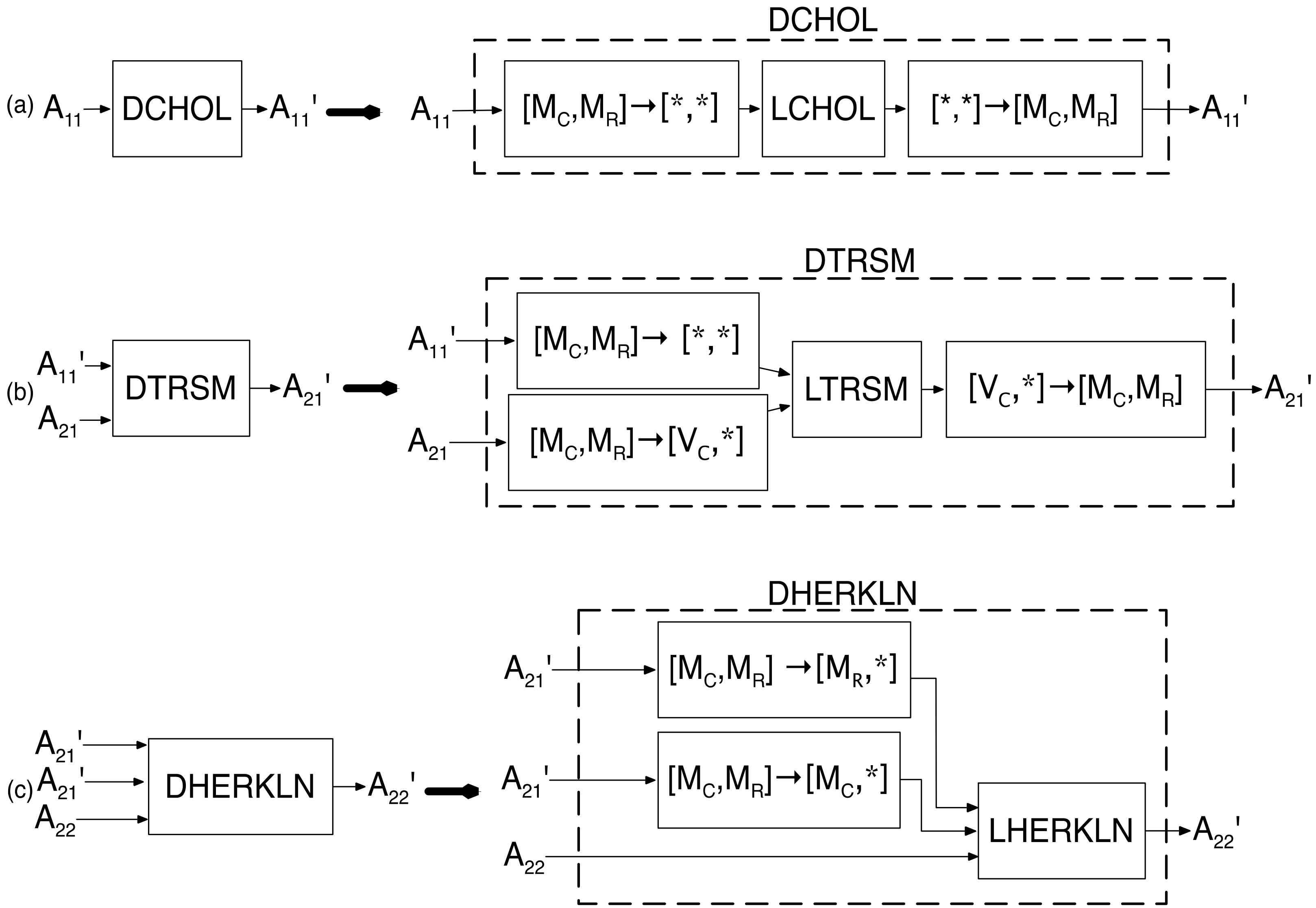
Speaking of DSLs

- Distinguished by constructs specific to domain
- Allows definition of domain-specific relationships compactly (preferably declaratively)
- In our opinion, this rules out traditional view of libraries
  - (Controversial)
- E.g. relational SQL (database queries)

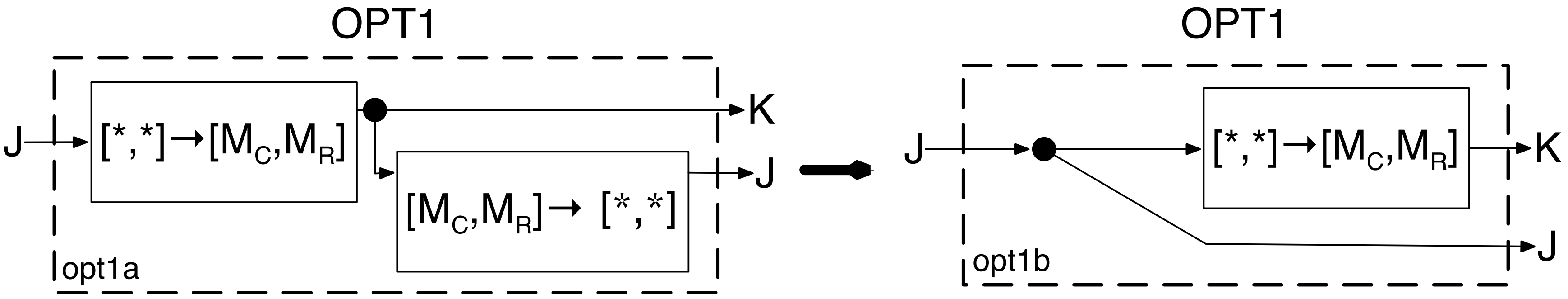
View as DAG in the spirit of MDE (this is a DSL)



Transform with Implementations (choose refinement of abstractions)



Transform to Optimize



OPT1

J

$[*,*]\rightarrow[M_C,M_R]$

K

J

$[M_C,M_R]\rightarrow [*,*]$

opt1a

OPT1

J

$[*,*]\rightarrow[M_C,M_R]$

K

J

opt1b

Grammars and Meta-Models

- Models are algorithms/code
- Meta-model defines correct models
- Ensures proper types and properties in code
  - Code compiles
- Meta-model for domain is grammar for DSL

Design by Transformation

- Two types of transformations come naturally
  - Box rewrites to specify abstraction implementations
  - Optimizations
- API for common abstractions already developed
  - BLAS, LAPACK, MPI
  - These are the abstraction boxes
  - Implementations/refinements are known
- Optimizations known to experts

Correct by Construction

- Start with correct model
  - E.g. derived to be correct with FLAME
- Apply correct transformations
- End with correct model
  - Compiles for target architecture
  - Gets same answer as starting model

The big idea…

Encode transformations to be reused not code that is disposable

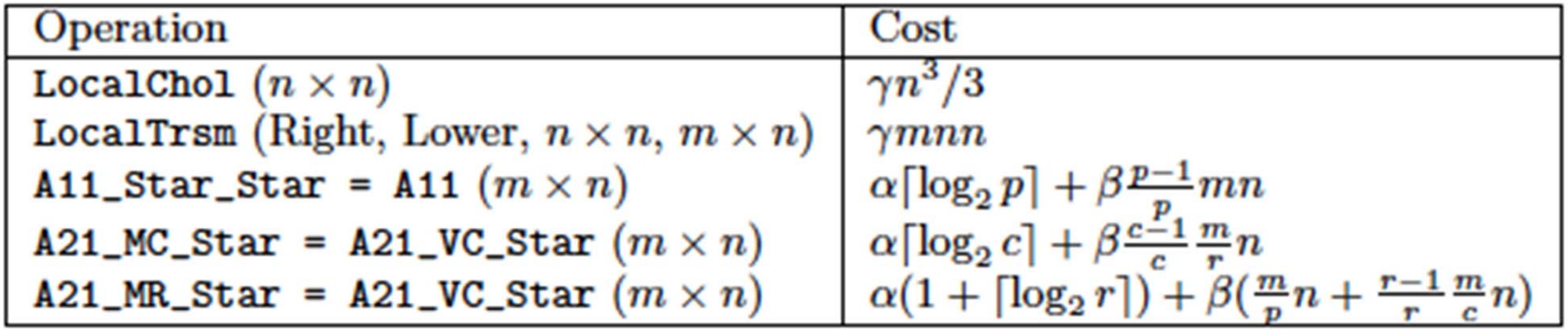
A Mechanical System Would…

- Have many instances of the two transformations
- Apply these to an input algorithm
- Transform the algorithm to many implementations of varying efficiency
  - Combinatorial explosion

How Much Does it Cost?

- An expert uses rough idea of runtimes to make implementation choices
  - Know which implementations are better than others
- For DxT generate all implementation and estimate costs (e.g. runtime or power consumption)
  - Search the space of possibilities
  - Attach cost to each of DSL's possible operations
- Choose "best" implementations from the entire space

Cost Functions



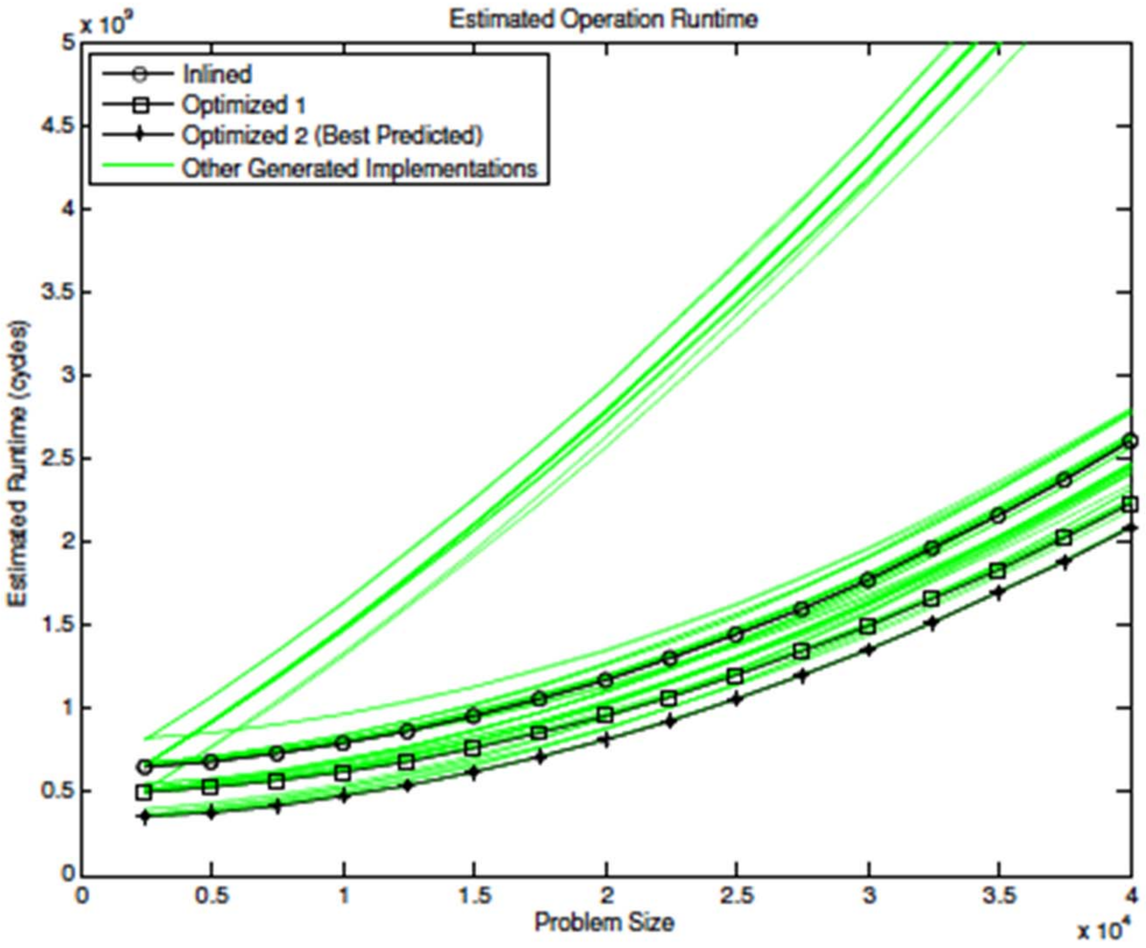
Operation	Cost
`LocalChol` $(n \times n)$	$\gamma n^3/3$
`LocalTrsm` (Right, Lower, $n \times n$, $m \times n$)	γmnn
`A11_Star_Star = A11` $(m \times n)$	$\alpha\lceil\log_2 p\rceil + \beta\frac{p-1}{p}mn$
`A21_MC_Star = A21_VC_Star` $(m \times n)$	$\alpha\lceil\log_2 c\rceil + \beta\frac{c-1}{c}\frac{m}{r}n$
`A21_MR_Star = A21_VC_Star` $(m \times n)$	$\alpha(1+\lceil\log_2 r\rceil) + \beta(\frac{m}{p}n + \frac{r-1}{r}\frac{m}{c}n)$

- Include machine-specific and problem-size parameters
- First-order approximations
- Just meant to separate bad choices from good

Prototype System

- Takes input algorithm graph
- Generates all implementations from known transformations
- 10s-10,000s of implementations
  - 2 Cholesky variants
  - 1 TRSM variant
  - 3 GEMM variants
  - 1 variant of preprocessor operation for generalized eigenvalue problem
- Same or better implementations as hand-generated
- These are indicative of many more operations that can be supported by DxT

Cholesky Cost Estimates



What DSLs Do For Us

- Enable us to layer code/functionality
  - Understand the layers
  - Encode transformations to break through layers
- Enable us to define meta-models/grammar to guide transformations and ensure correctness
- Limit the amount of operations (and cost functions) we need to support because functionality abstracted and re-used

DSLs Enable Us To…

Encode transformations to be reused not code that is disposable

Future Work

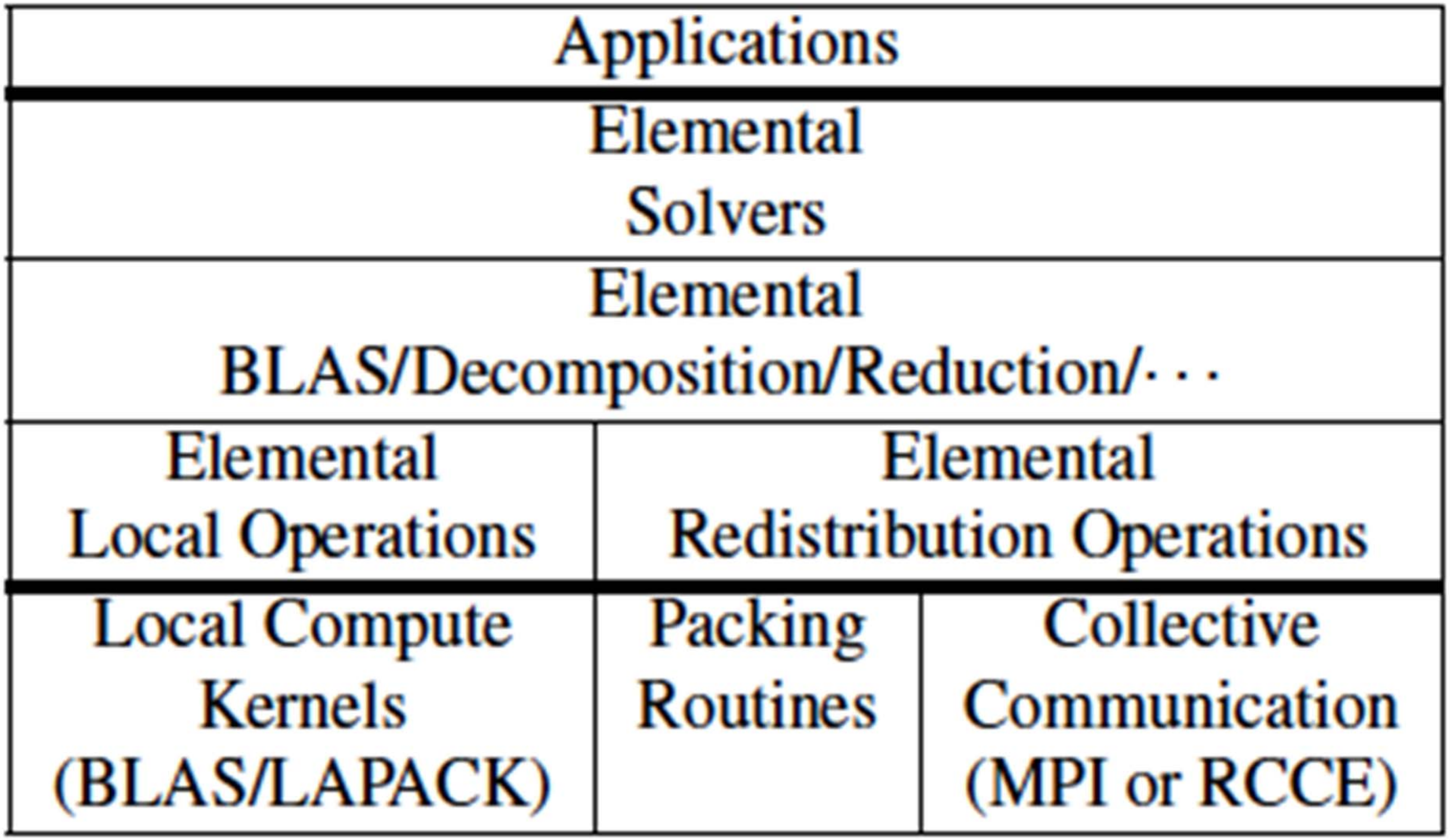
- Encode more transformations
- Target SMP and sequential algorithms
  - Low-level BLAS kernels
- Improve cost estimates
- Algorithmic transformations (variants)
- Try other domains in HPC
- Replace libraries like libflame and Elemental with libraries of algorithms and transformations
  - Not just auto-tune

Questions?

- Read our SC11 Submission
- FLAME and libflame
  - www.cs.utexas.edu/~flame
- Elemental
  - code.google.com/p/elemental
- bamarker@cs.utexas.edu
- Thanks to NSF and Sandia fellowships
- Rui Gonçalves, Taylor Riche, Andy Terrel

```
//-------------------------------------------------------------------------//
    //$A_{11}$ = Chol($A_{11}$)
    A11_Star_Star = A11;
    lapack::internal::LocalChol( Lower, A11_Star_Star );
    A11 = A11_Star_Star;

    //$A_{21}$ = $A_{21}$ TRIL($A_{11}$)$^{-T}$
    A21_VC_Star = A21;
    blas::internal::LocalTrsm
    ( Right, Lower, ConjugateTranspose, NonUnit,
      (F)1, A11_Star_Star, A21_VC_Star );

    //$A_{22}$ = $A_{22}$ – TRIL($A_{21}A_{21}^{T}$)
    A21_MC_Star = A21_VC_Star;
    A21_MR_Star = A21_VC_Star;
    blas::internal::LocalTriangularRankK
    ( Lower, ConjugateTranspose,
      (F)-1, A21_MC_Star, A21_MR_Star, (F)1, A22 );

    A21 = A21_MC_Star;
//-------------------------------------------------------------------------//
```

Elemental’s Layering



<table>
<tr><td colspan="3">Applications</td></tr>
<tr><td colspan="3">Elemental
Solvers</td></tr>
<tr><td colspan="3">Elemental
BLAS/Decomposition/Reduction/· · ·</td></tr>
<tr><td>Elemental
Local Operations</td><td colspan="2">Elemental
Redistribution Operations</td></tr>
<tr><td>Local Compute
Kernels
(BLAS/LAPACK)</td><td>Packing
Routines</td><td>Collective
Communication
(MPI or RCCE)</td></tr>
</table>

What This Gets Us

- Encode knowledge about component operations
  - Generate implementations and optimize with transformations
- Libraries of transformations used to generate libraries of code (DxT)
  - Not libraries of specific code for operation A on architecture B with characteristics C
  - Libraries of how to use many A’s, B’s, and C’s
  - Next generation libflame and Elemental consist of algorithms and transformations