Jakub Kurzak

Parallel Linear Algebra Software for Multi-Core Architectures (PLASMA) for the CELL BE

Georgia Tech CELL Workshop

June 18, 2007

Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

Outline

- **New goals for dense linear algebra on multi-core processors**
- **Hand-crafted code example**
  - solving SPD systems of linear equations
- **Automated code generation**
  - DAG based execution with CELL SuperScalar
- **New algorithmic approaches**
  - Algorithms by tiles

Dense Linear Algebra Software Evolution

LINPACK

- Vector machines – BLAS 1, BLAS 2

LAPACK

- Cache-based machines – BLAS 3

ScaLAPACK

- Distributed memory machines – PBLAS, BLACS, MPI

PLASMA

- General framework for
  - Multi-core
  - CELL BE
  - distributed memory machines

The PLASMA Framework

PLASMA

- Mixed-precision algorithms to exploit SIMD ILP
- Dynamic DAG-based scheduling
- Non-blocking communication
- Algorithms by tiles
  - Maximum locality
  - Minimal synchronization
  - High performance data representation (BDL)

New BLAS required

- Block Data Layout
- Focus on tile kernel performance
- Implementation of new operations

Scalability

scalability

scalability

.....

.....

.....

Exploiting SIMD ILP in Single Precision

Mixed-precision Iterative Refinement (SPD)

Solve: *Ax = b, where A is SPD*

$L\,L^T$ = cholesky(A)	SINGLE	$O(n^3)$
$x = L\backslash(L^T\backslash b)$	SINGLE	$O(n^2)$
$r = b - Ax$	DOUBLE	$O(n^2)$
WHILE || r || not *small enough*		
$z = L\backslash(L^T\backslash r)$	SINGLE	$O(n^2)$
$x = x + z$	DOUBLE	$O(n^1)$
$r = b - Ax$	DOUBLE	$O(n^2)$
END		

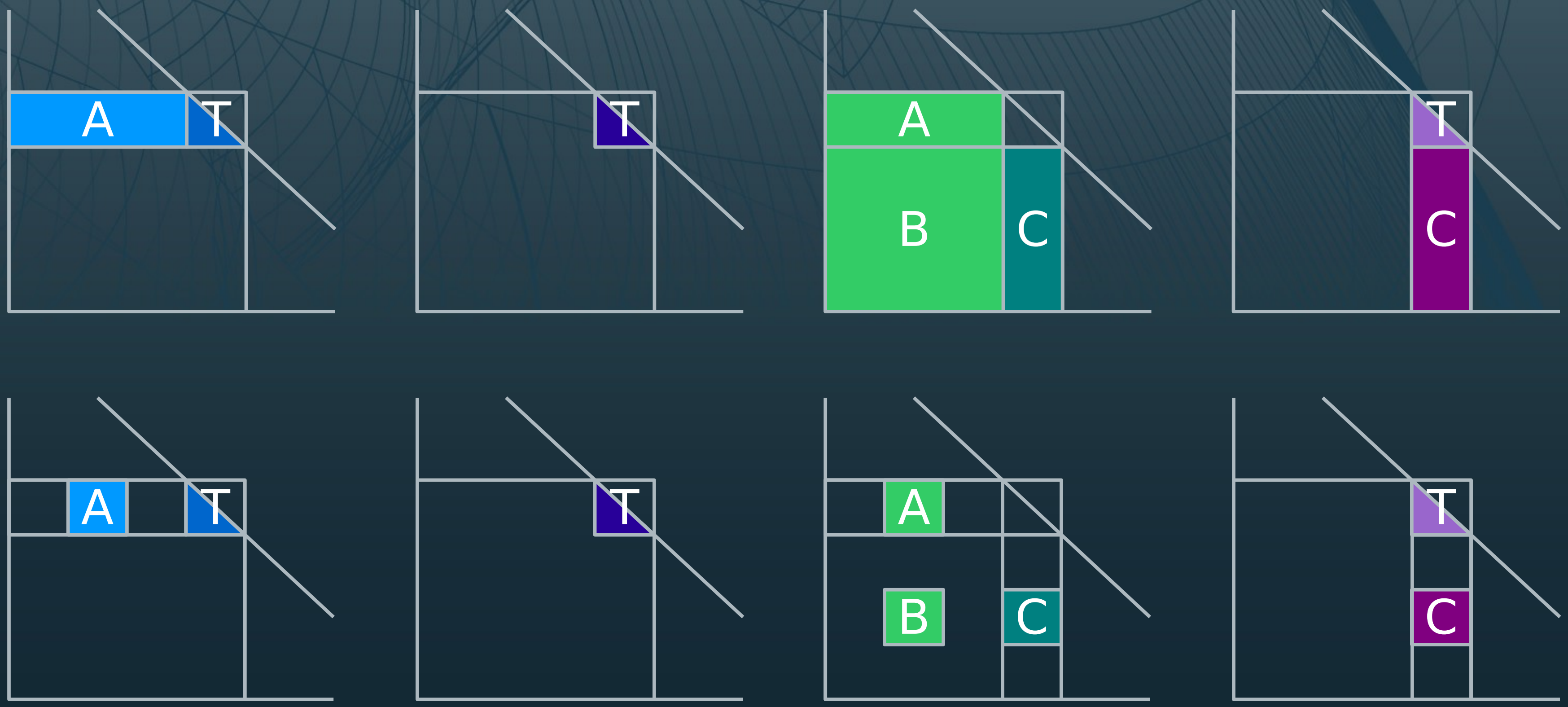
Algorithms by Tiles – Cholesky Factorization

$T = T - A \times A^T$
SYRK

$T = LL^T$
POTRF

$C = C - B \times A^T$
GEMM

$C = C \backslash T$
TRSM

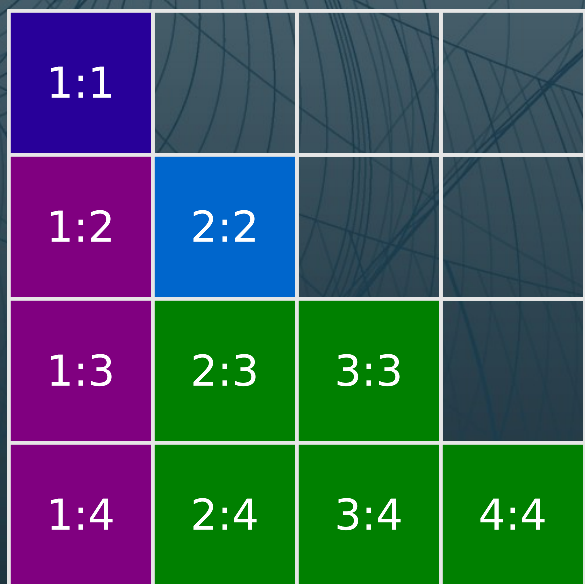


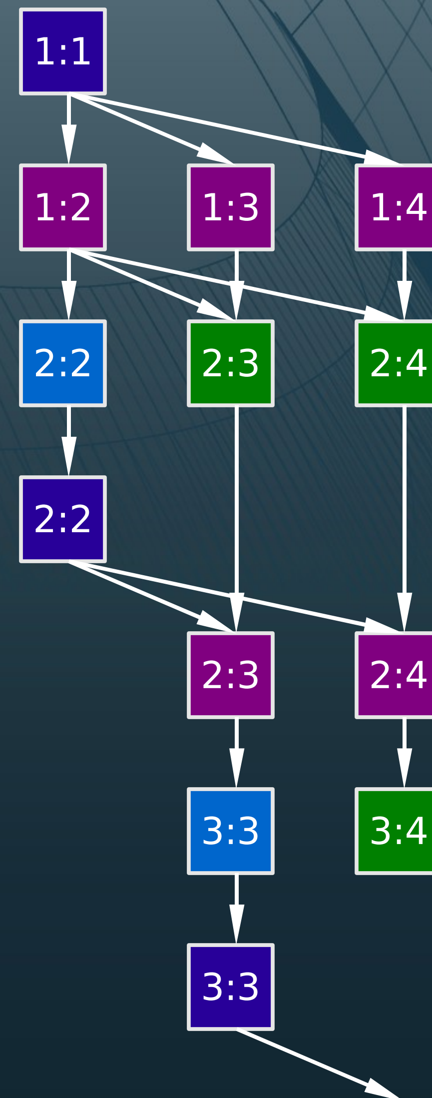
Building Blocks – SIMD Tile Kernels

Equivalent BLAS / LAPACK call	.c [LOC]	.o [KB]	Compilation	Execution time [µs]	Execution speed [Gflop/s]
cblas_**sgemm**(CblasRowMajor, CblasNoTrans, CblasTrans, 64, 64, 64, -1.0, A, 64, B, 64, 1.0, C, 64);	330	9.0	spu-gcc -Os	22.78	**23.01** **24.00**
cblas_**ssyrk**(CblasRowMajor, CblasLower, CblasNoTrans, 64, 64, -1.0, A, 64, 1.0, C, 64);	160	4.7	spuxlc -O3	13.23	**20.12**
cblas_**strsm**(CblasRowMajor, CblasRight, CblasLower, CblasTrans, CblasNonUnit, 64, 64, 1.0, T, 64, B, 64);	310	8.2	spuxlc -O3	16.47	**15.91**
lapack_**spotrf**(lapack_lower, 64, trans(A), 64, &info);	340	4.1	spu_gcc -O3	15.16	**5.84**

Real men program in assembly

DAG-based Dependency Tracking

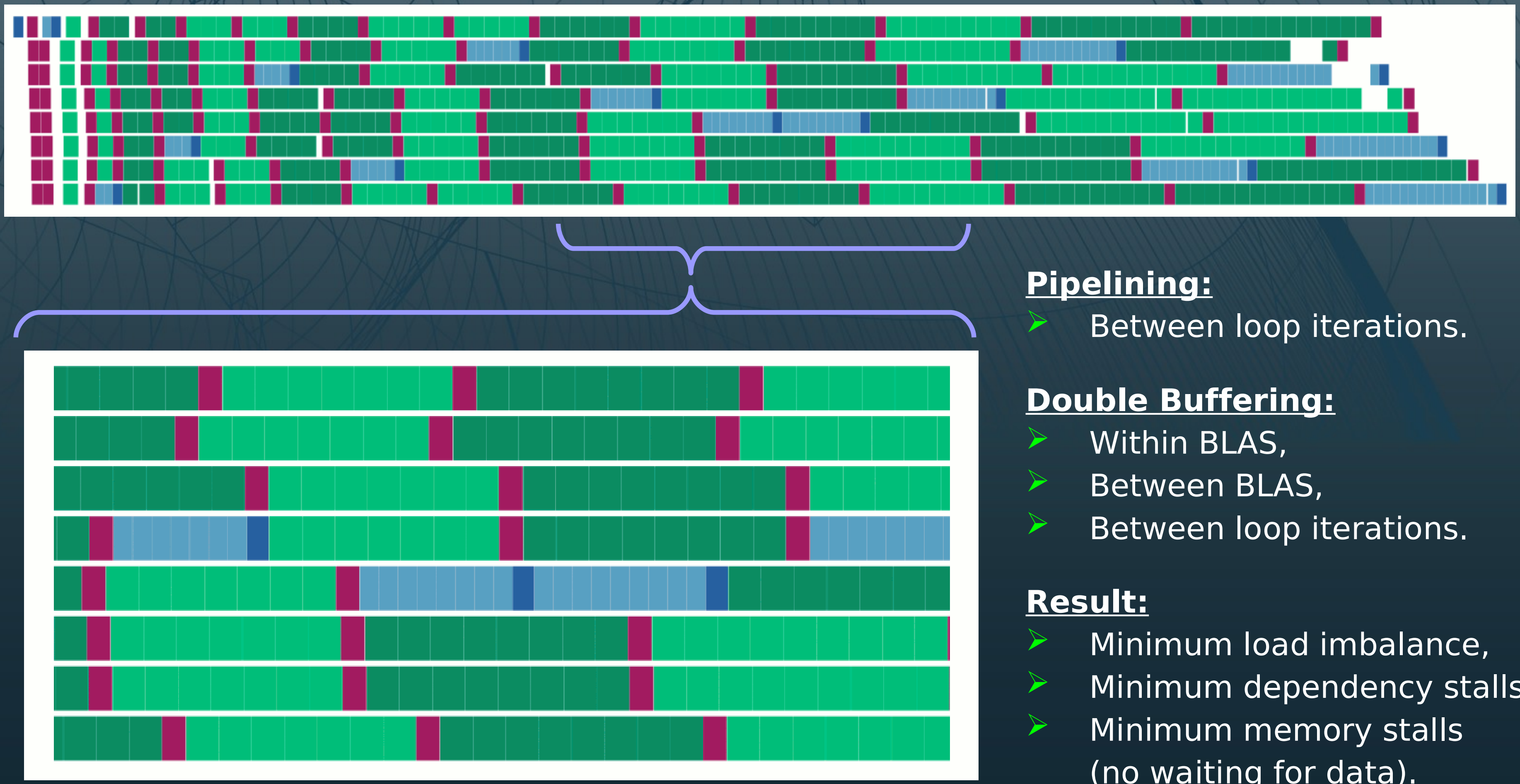




<u>Dependencies expressed by the DAG are enforced on a tile basis:</u>

- fine-grained parallelization
- flexible scheduling

Pipelining & Double Buffering



Pipelining:

- Between loop iterations.

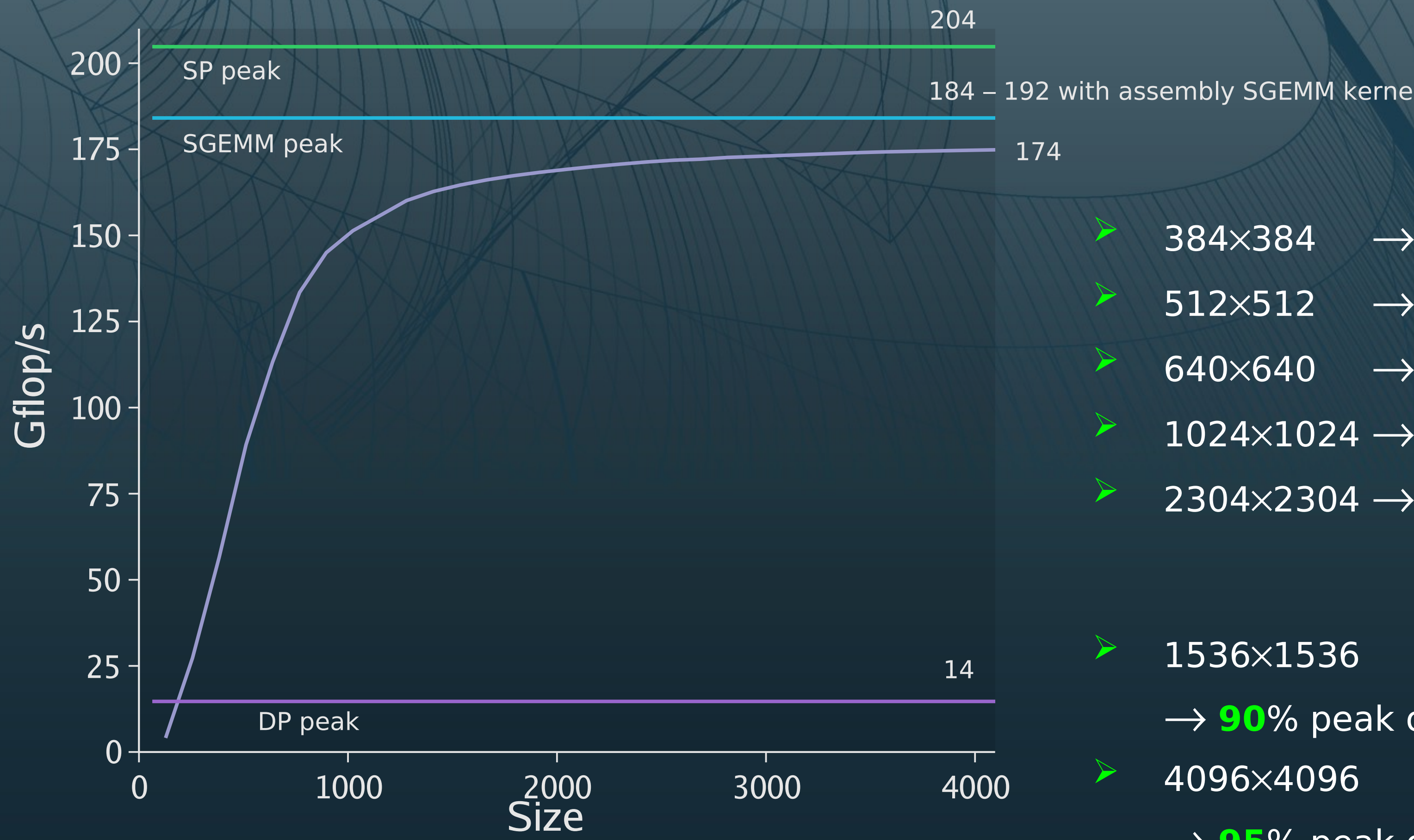
Double Buffering:

- Within BLAS,
- Between BLAS,
- Between loop iterations.

Result:

- Minimum load imbalance,
- Minimum dependency stalls,
- Minimum memory stalls (no waiting for data).

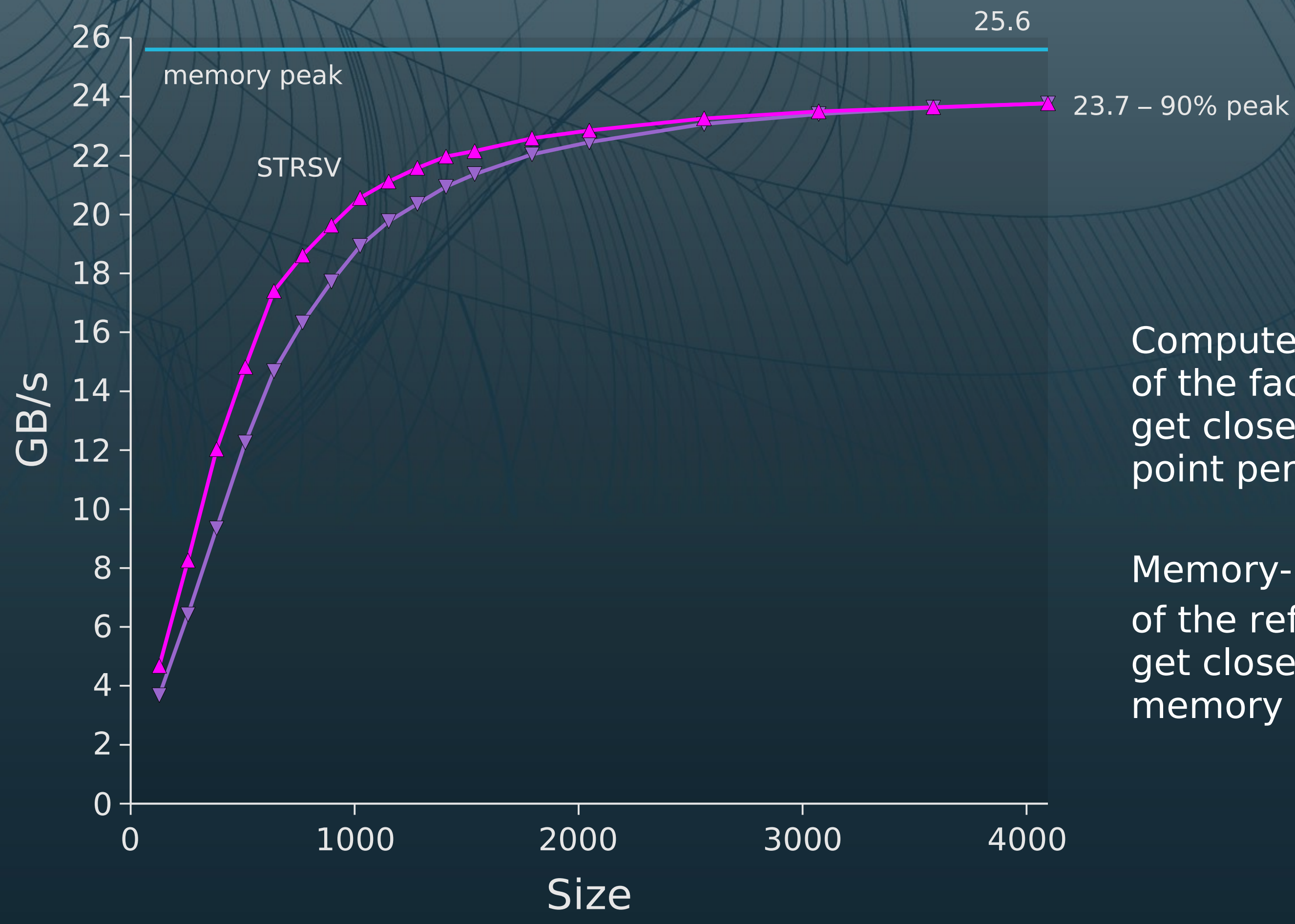
Cholesky Factorization (SPOTRF)



- 384×384 → >**50** Gflop/s
- 512×512 → ~**90** Gflop/s
- 640×640 → >**110** Gflop/s
- 1024×1024 → >**150** Gflop/s
- 2304×2304 → >**170** Gflop/s

- 1536×1536
  → **90**% peak of SGEMM
- 4096×4096
  → **95**% peak of SGEMM

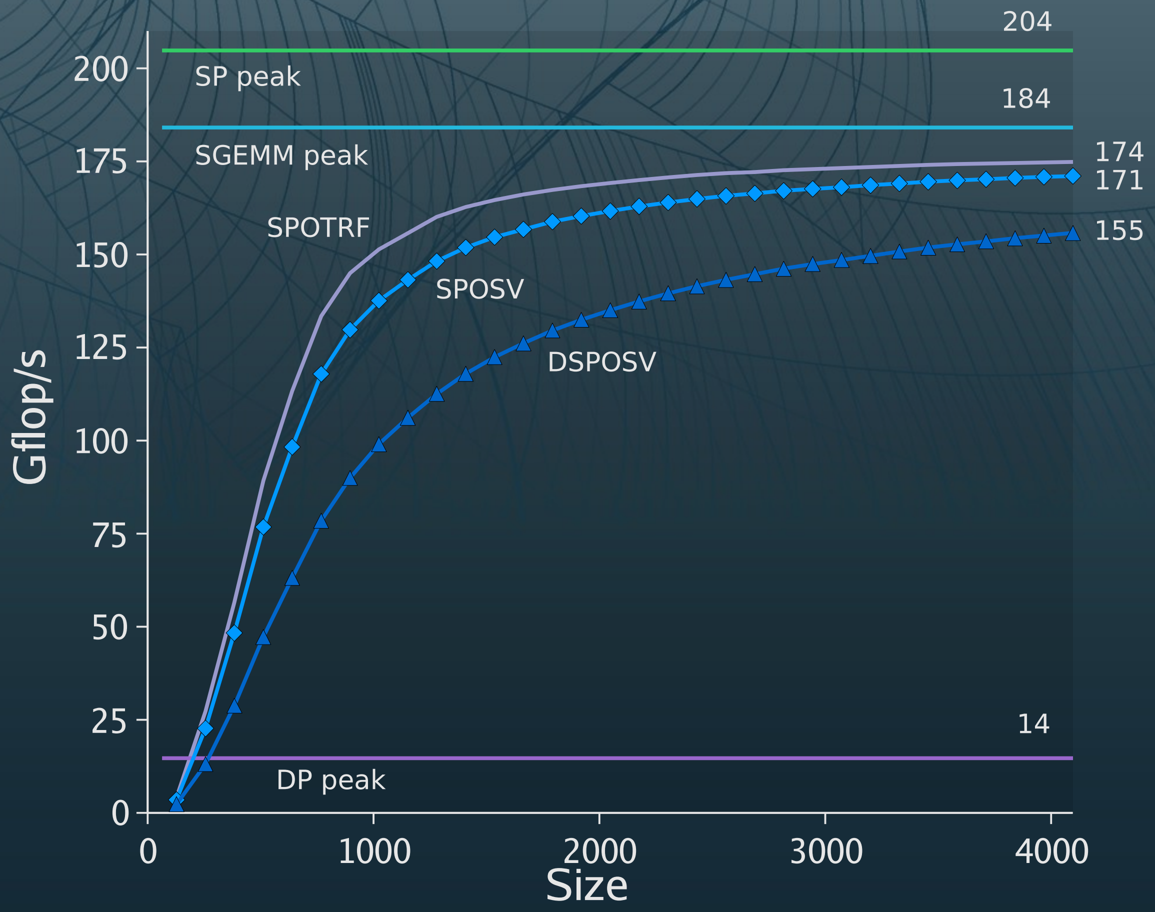
Triangular Solve (STRSV)



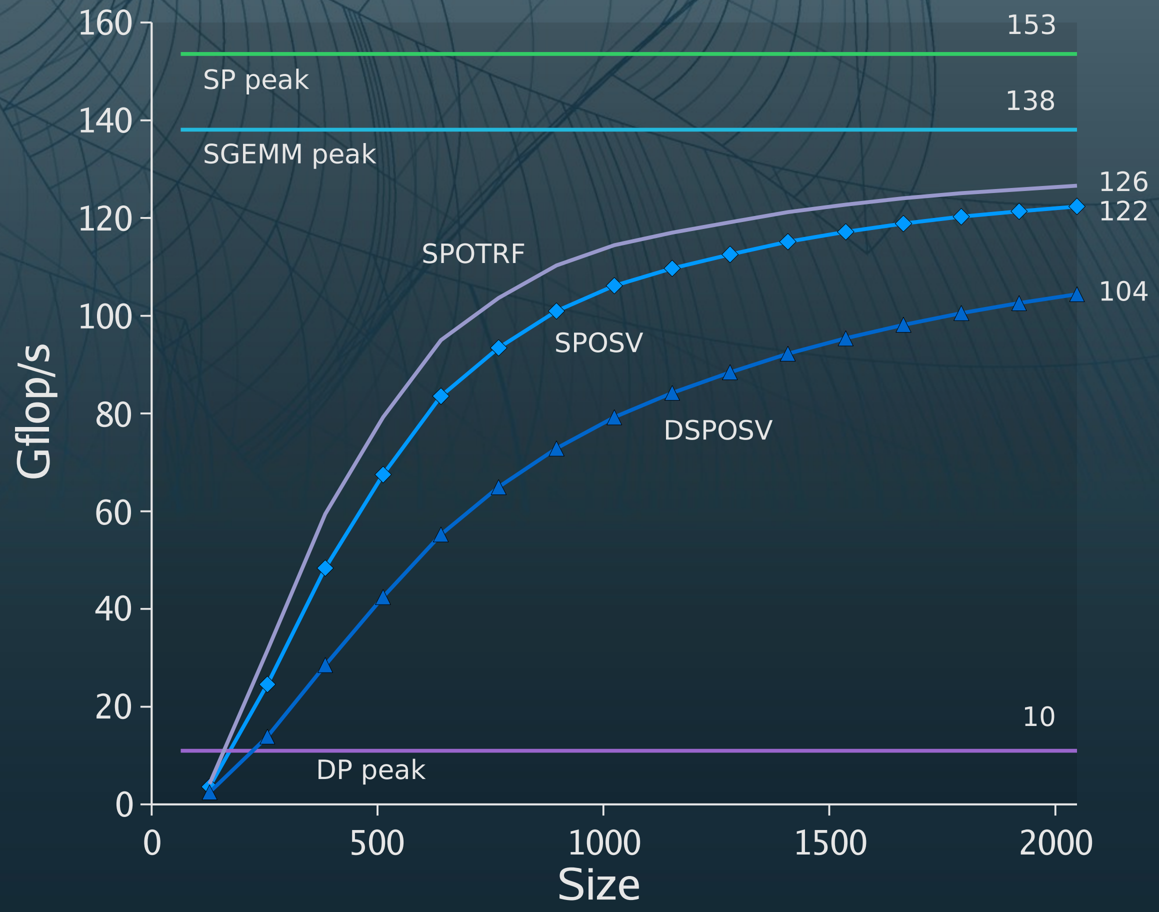
Compute-bound operations of the factorization (SPOTRF) get close to the peak floating point performance

Memory-bound operations of the refinement (STRSV) get close to the peak memory bandwidth

Performance – CELL Blade



Performance – Playstation 3



The Need for Automation

```
         DO 20 J = 1, N, NB
*
*           Update and factorize the current diagonal block and test
*           for non-positive-definiteness.
*
            JB = MIN( NB, N-J+1 )
            CALL DSYRK( 'Lower', 'No transpose', JB, J-1, -ONE,
     $                  A( J, 1 ), LDA, ONE, A( J, J ), LDA )
            CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
            IF( INFO.NE.0 )
     $         GO TO 30
            IF( J+JB.LE.N ) THEN
*
*              Compute the current block column.
*
               CALL DGEMM( 'No transpose', 'Transpose', N-J-JB+1, JB,
     $                     J-1, -ONE, A( J+JB, 1 ), LDA, A( J, 1 ),
     $                     LDA, ONE, A( J+JB, J ), LDA )
               CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
     $                     N-J-JB+1, JB, ONE, A( J, J ), LDA,
     $                     A( J+JB, J ), LDA )
            END IF
   20    CONTINUE
```

LAPACK FORTRAN 77 Cholesky factorization

- **Roughly 20 lines**

CELL Cholesky main factorization routine

- **Roughly 400 lines !!!**

Some automation needed !!!

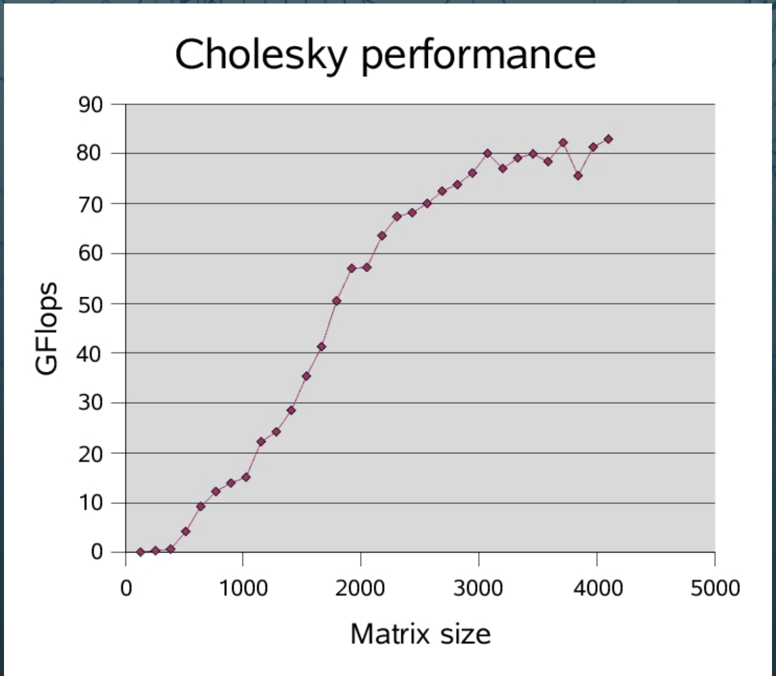
Cholesky – Sequential CELL Code

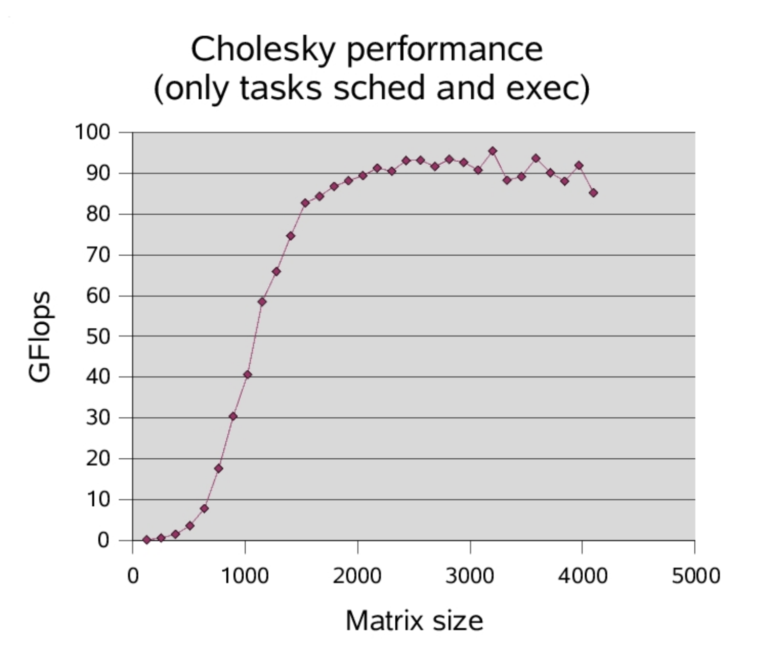
```
for (i = 0; i < DIM; i++) {
  for (j= 0; j< i-1; j++){
     for (k = 0; k < j-1; k++) {
         sgemm_tile( A[i][k], A[j][k], A[i][j] );
     }
     strsm_tile( A[j][j], A[i][j] );
  }
  for (j = 0; j < i-1; j++) {
     ssyrk_tile( A[i][j], A[i][i] );
  }
  spotrf_tile( A[i][i] );
}


void sgemm_tile(float *A, float *B, float *C)


void strsm_tile(float *T, float *B)


void ssyrk_tile(float *A, float *C)
```

Cholesky – CELL SupeScalar (BSC)

```
for (i = 0; i < DIM; i++) {
  for (j= 0; j< i-1; j++){
     for (k = 0; k < j-1; k++) {
         sgemm_tile( A[i][k], A[j][k], A[i][j] );
     }
     strsm_tile( A[j][j], A[i][j] );
  }
  for (j = 0; j < i-1; j++) {
     ssyrk_tile( A[i][j], A[i][i] );
  }
  spotrf_tile( A[i][i] );
}

#pragma css task input(A[64][64], B[64][64]) inout(C[64][64])
void sgemm_tile(float *A, float *B, float *C)

#pragma css task input (T[64][64]) inout(B[64][64])
void strsm_tile(float *T, float *B)

#pragma css task input(A[64][64], B[64][64]) inout(C[64][64])
void ssyrk_tile(float *A, float *C)
```

Cholesky – CELL SupeScalar – Performance





DAG construction on the fly

DAG construction before execution

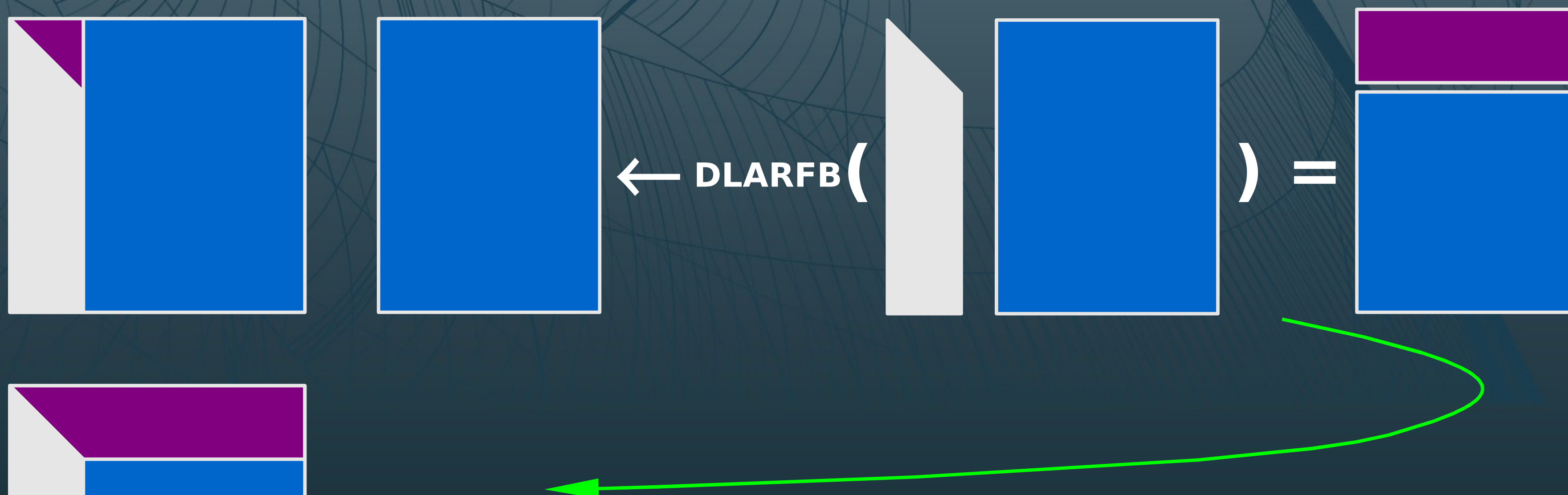
Block Algorithms – LAPACK QR Factorization

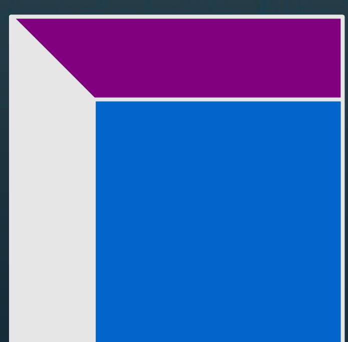
Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections

Factorize the panel – calculate small R (purple) and a block of Householder reflectors (white)

Block Algorithms – LAPACK QR Factorization

Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections

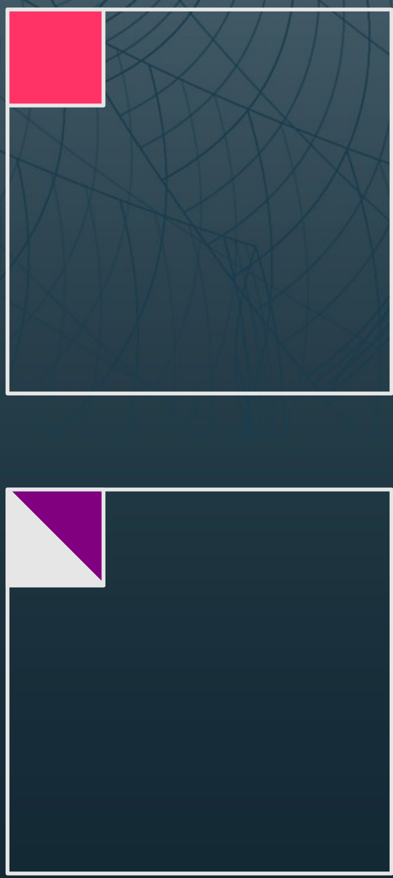


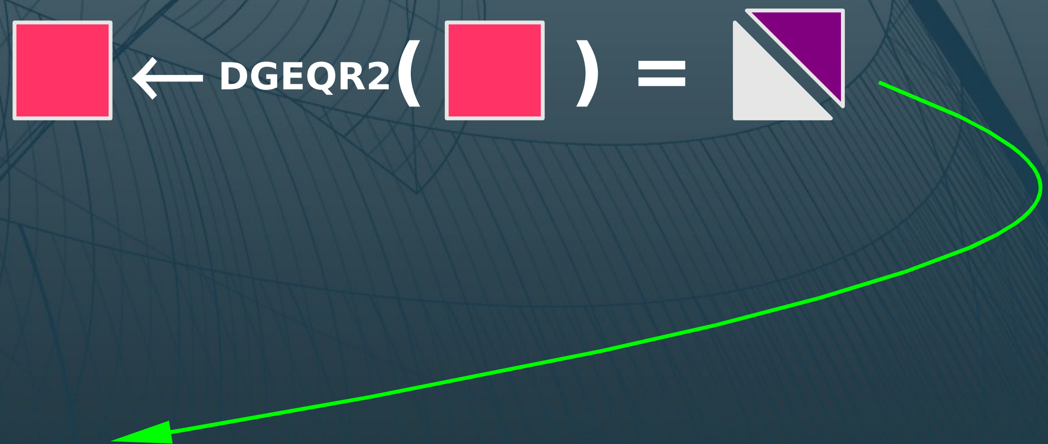


Update the trailing submatrix – apply the Householder reflectors to the submatrix to the right from the panel

Tile Algorithms – PLASMA QR Factorization

Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections

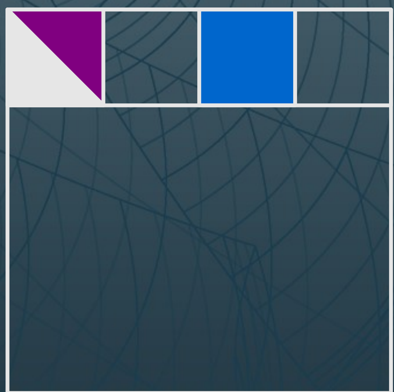


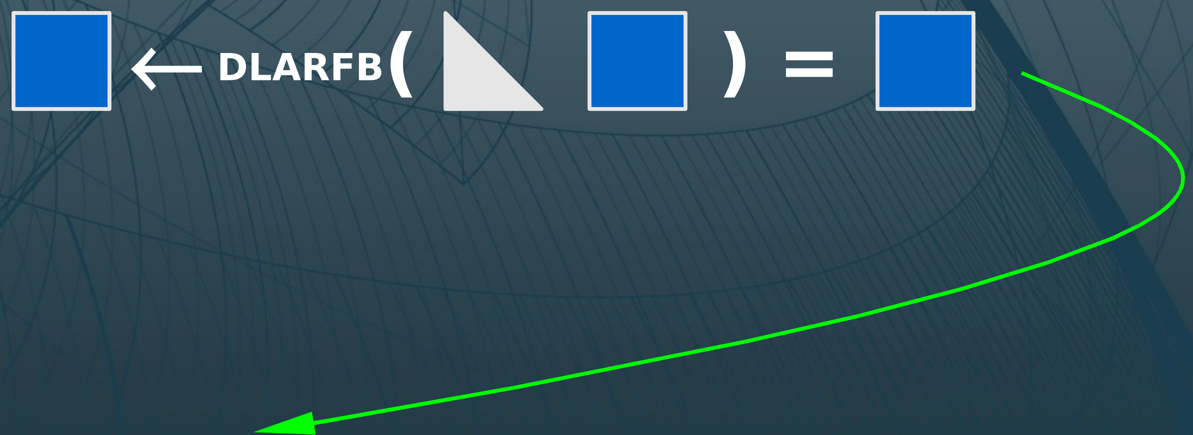


Factorize the firts tile – calculate small R (purple) and a small block of Householder reflectors (white)

Tile Algorithms – PLASMA QR Factorization

Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections

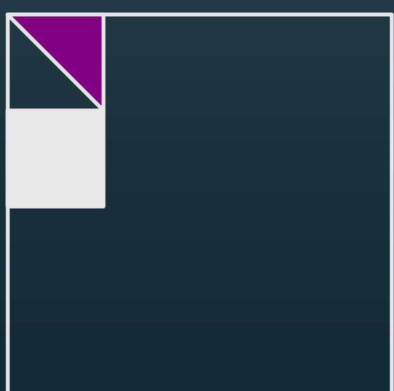




Update the first row of tiles

Tile Algorithms – PLASMA QR Factorization

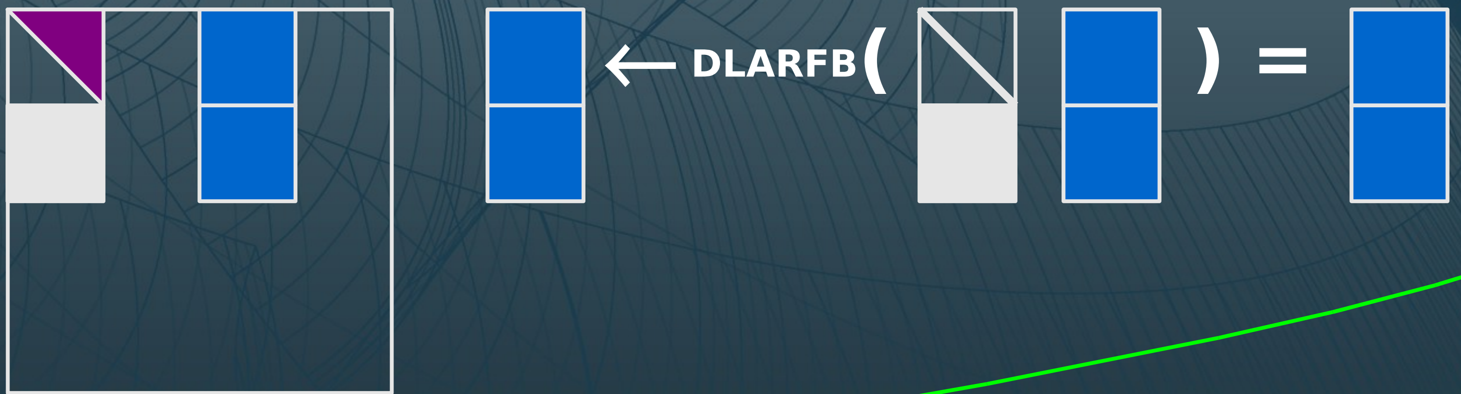
Decompose a matrix into factors Q and R, where Q is unitary
and R is upper triangular using Householder reflections

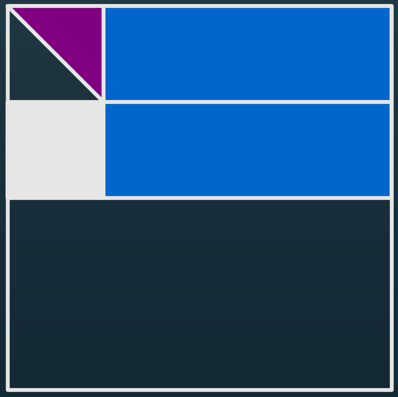


Couple the first R with the first tile in the same column
and compute the QR factorization

Tile Algorithms – PLASMA QR Factorization

Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections

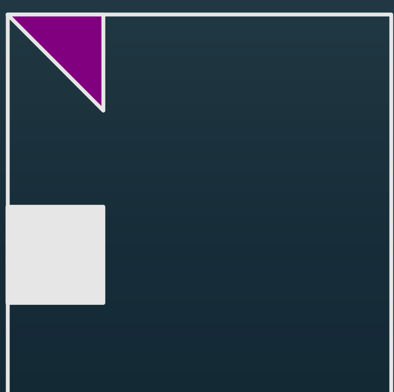




Update the two rows of tiles to the right

Tile Algorithms – PLASMA QR Factorization

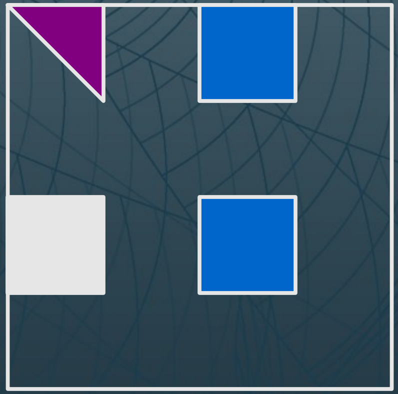
Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections

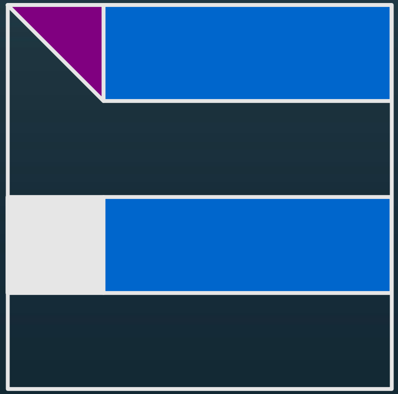


Couple the first R with the second tile in the same column and compute the QR factorization

Tile Algorithms – PLASMA QR Factorization

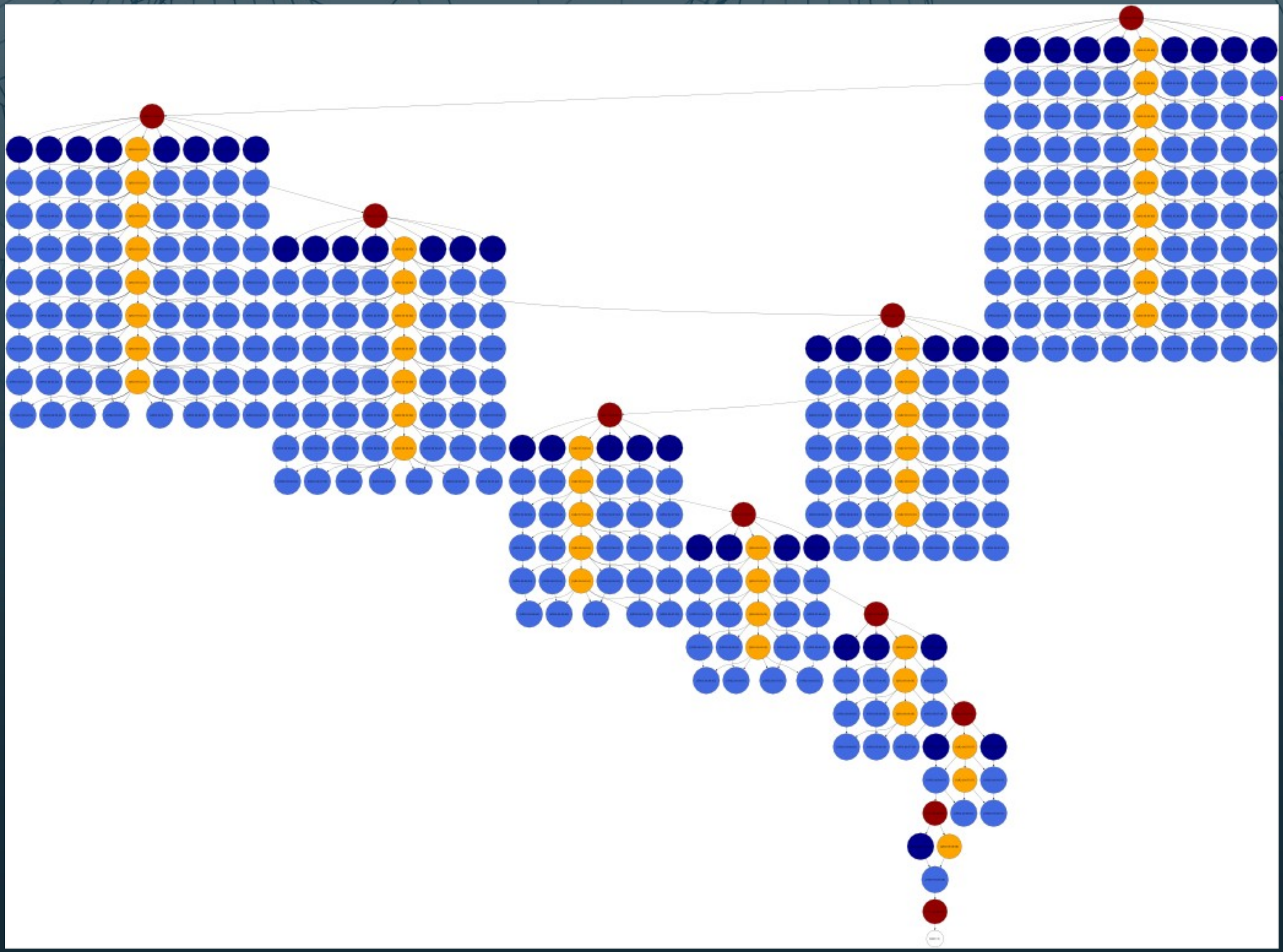
Decompose a matrix into factors Q and R, where Q is unitary and R is upper triangular using Householder reflections





Update the two rows of tiles to the right

PLASMA QR Factorization DAG



Each node is a tile operation

Future

- **CELL implementation of tile algorithms**
  - Cholesky (done), LU, QR,
  - Hessenbeg, bi-diagonal, tri-diagonal reduction

- **Efficient DAG construction and execution**