SAND2019-2037C

Performance Portable SIMD Approach Implementing Block Line Solver For Coupled PDEs

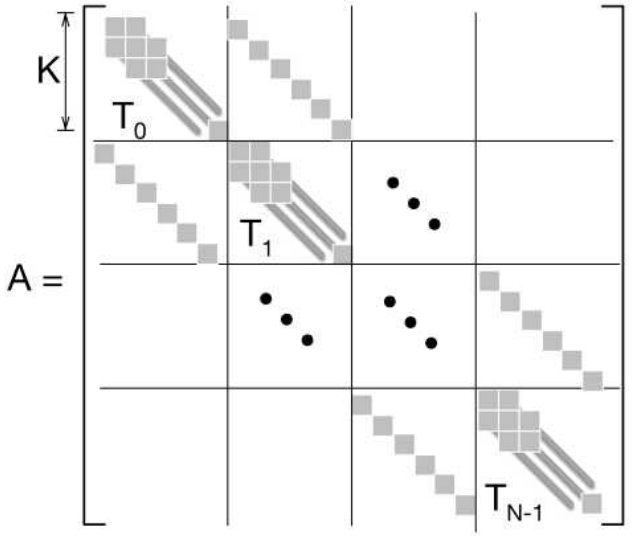
Kyungjoo Kim

Center for Computing Research, Sandia National Labs

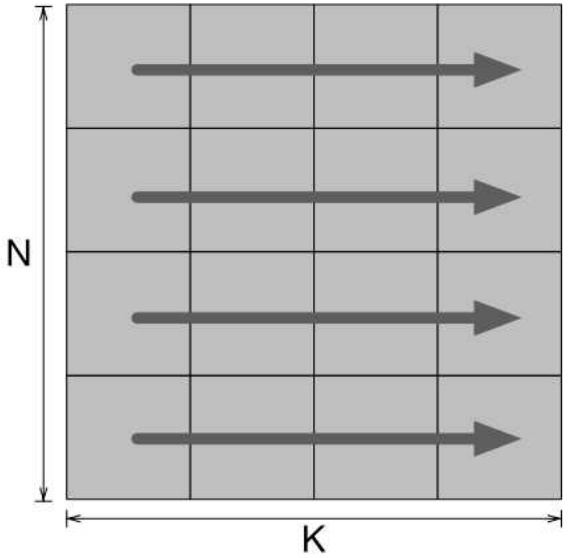
SIAM CSE, Feb 25 - Mar 1, 2019

Line Implicit Solver

- Consider a block sparse system of equations $Ax = b$ arising from coupled PDEs.
- Lines are usually formed in the boundary layer to resolve shocks.
- Small blocks represent interactions among a group of variables e.g., velocity, pressure, density and chemical species.
- Default solver for SPARC (Sandia production-level CFD code)



Linear system of equations



Problem domain and extracted lines

Line Implicit Solver

- By splitting $A = M - S$, we obtain

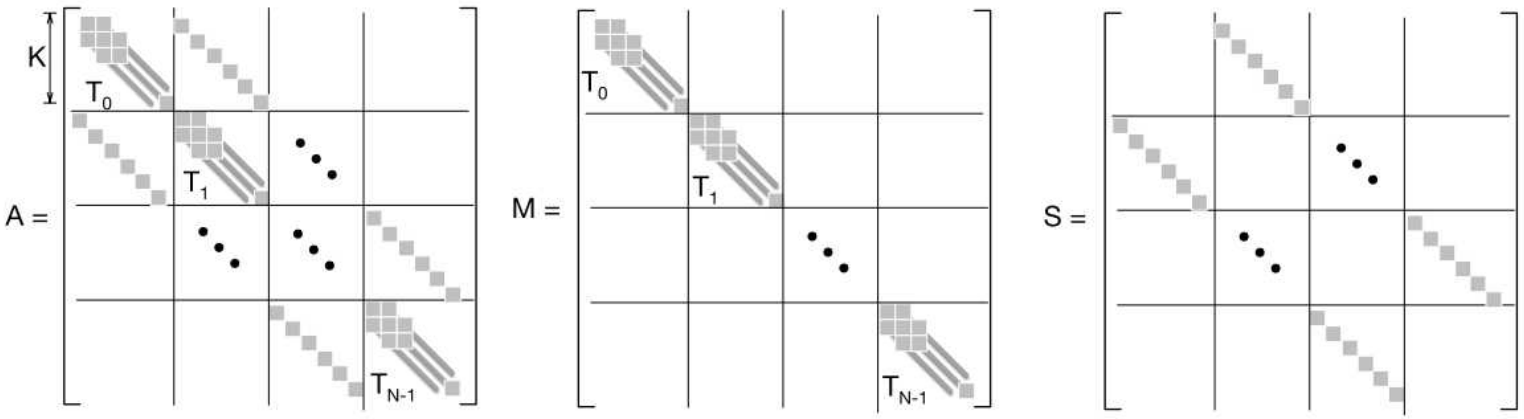
$$(M - S)x = b$$
$$Mx = b + Sx$$
$$x = M^{-1}(b + Sx) = x + M^{-1}(b - Ax),$$

  where M is a set of block tridiagonal matrices corresponding to the extracted lines of elements.

- Solve this iteratively

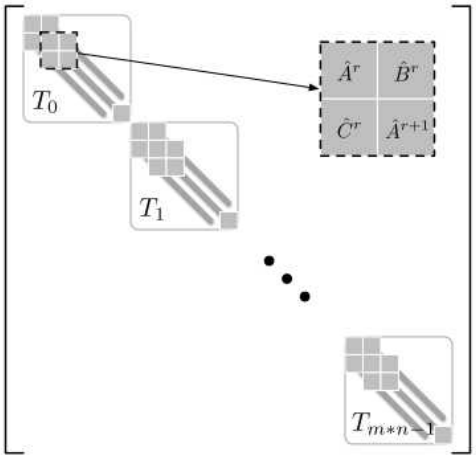
$$x^{k+1} = M^{-1}(b + Sx^k).$$

- This procedure requires solutions of many block tridiagonal systems and efficient block sparse matrix vector multiplication.



Solving Many Tridiagonal Systems

- A fixed number of iterations are applied $x^{k+1} = M^{-1}(b + Sx^k)$.
- M is factorized once per solution (or each non-linear iteration) and applied multiple times.
- Typical blocksizes b are 3, 5, 9 and 15, which are related to scientific applications e.g., elasticity, ideal gas and reactive fluid problems.
- Limit memory usage up to 16 GB i.e., MCDRAM on KNL and GPU device memory.
- A typical local problem size in a 3D domain ($m \times n \times k$) is selected as $128 \times 128 \times 128$ for $b = 3, 5$ and $64 \times 64 \times 128$ for $b = 10, 15$.
- Batch parallelism ($m \times n$) is considered for solving block tridiagonal systems.



for T *in* $\{T_0, T_1, \cdots, T_{m\times n-1}\}$ **do in parallel**
  for $r \leftarrow 0$ **to** $k-2$ **do**
    $\hat{A}^r := LU(\hat{A}^r)$;
    $\hat{B}^r := L^{-1}\hat{B}^r$;
    $\hat{C}^r := \hat{C}^r U^{-1}$;
    $\hat{A}^{r+1} := \hat{A}^{r+1} - \hat{C}^r\hat{B}^r$;
  $\hat{A}^{k-1} := LU(\hat{A}^{k-1})$;

Initialization of the line smoother

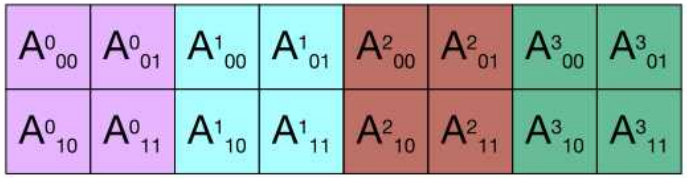
Efficient Implmentation using Compact Data Layout (SIMD)

Problems using vendor libraries

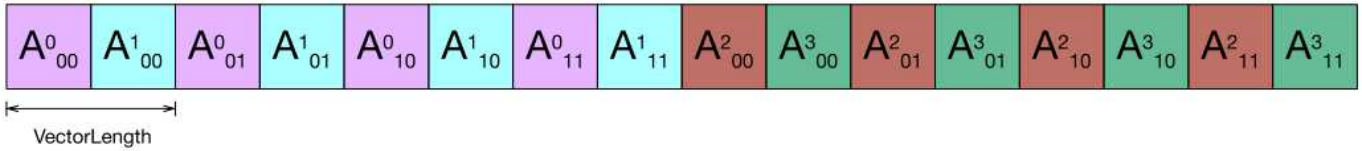
- Standard BLAS/LAPACK are not optimized for solving small problems.
- Using batched BLAS/LAPACK APIs, no data locality is exploited in a sequence of batch operations.

Our approach: Compact Data Layout

- A new interleaved data layout for batched BLAS/LAPACK for efficient vectorization on small problems.
- Solve a small number of problems in parallel using SIMD instructions.
- Array of blocks is represented by a multi-dimensional array.



- Compact layout packs data across batch instances with respect to SIMD length.



SIMD Approach To Solve Small Dense Problems

- To vectorize across batch instances, we use packed vectors as computing unit instead scalar.
- By overloading arithmatic operators, the scalar BLAS are reused and naturally vectorized.

```
template<typename T, int vl>
struct Vector<T,vl> {
  T _data[vl];
};
```

```
Vector<T,vl>
operator+(Vector<T,vl> a, Vector<T,vl> b) {
  Vector<T,vl> r_val;
  for (int i=0;i<vl;++i)
    r_val._data[i] = a._data[i] + b._data[i];
  return r_val;
}
```

- Multidimensional array:

```
typedef double scalar_type;
typedef Vector<scalar_type,VectorLength> vector_type;

// LayoutRight: most right consecutive index is stored in the memory contiguously
Kokkos::View<vector_type*** ,LayoutRight> Av(''A'',     N, Blk, Blk);
// Reinterpretation of rank 3 vector_type view with scalar rank 4 view
Kokkos::View<scalar_type****,LayoutRight> As(Av.data(), N, Blk, Blk, VectorLength);
```

- What about GPUs ?

Implicit and Explicit (Warp) Vectorizations

Sandia National Laboratories

SIMD instructions on CPUs

- A single thread exploits data-level parallelism.
- Vetor length is given from hardware specific vector instruction set e.g., AVX512.
- A compiler is responsible for vectorization.

Warp-based (explicit) vectorization on GPUs

- Warp: a set of concurrent threads executing the same instruction.
- Each thread in a warp process scalar instructions.
- Vector instructions are dynamically formed by a warp.
- Coalesced memory access by threads is essential for efficient memory operations.
- GPUs have 128 bit memory instruction set (`double2` – internal vector type)
  - A benefit using `double2` is to reduce the number of memory transactions.

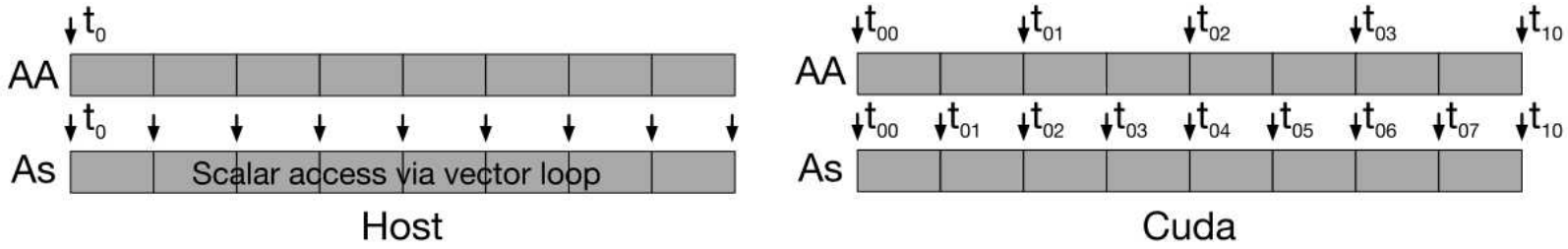
Kokkos Unified Programming Approach

```
using namespace Kokkos;
// Suppose that N = 64, b = 4

// Default values are pre-defined in KokkosBatched but a user can
// control the vector size for CUDA.
// vl: Host 8, Cuda 16, Vector length used for compact data format.
// il: Host 8, Cuda  2, Internal vector length used in a functor.
//                      Using internal vector type, we make sure
//                      using 128bit memory instructions.
constexpr int vl = DefaultVectorLength<exec_space>::value;
constexpr int il = DefaultInternalVectorLength<exec_space>::value;

typedef double scalar_type
typedef Vector<scalar_type,vl> vector_type;
typedef Vector<scalar_type,il> internal_vector_type;

// Av: Host 8x4x4, Cuda 4x4x4, Rank 3 Multi-dimensional array
//                             of vector_type.
View<vector_type*** ,LayoutRight> Av(''A'', N/vl, b, b);
```

Example: Set identity of N matrices $b \times b$

Kokkos Unified Programming Approach

```
using namespace Kokkos;
// Suppose that N = 64, b = 4

// Av: Host 8x4x4, Cuda 4x3x3, Rank 3 Multi-dimensional array
//                             of vector_type.
View<vector_type*** ,LayoutRight> Av(''A'', N/vl, b, b);

// Reinterpret the rank 3 vector view with the rank 4 internal vector type.
// AA: Host 8x4x4x1, Cuda 4x4x4x8
View<internal_vector_type****,LayoutRight> AA(Av.data(), N/vl, b, b, vl/il);

// One can also reinterpret the view with rank 4 scalar type if
// accessing idividual scalar entries is necessary.
// As: Host 8x4x4x8, Cuda 4x4x4x16
View<scalar_type****,LayoutRight> As(Av.data(), N/vl, b, b, vl);

// For this example, we use a view with internal vector type (AA).
```



Example: Set identity of N matrices $b \times b$; figure describes different data access patterns for a vector length 8.

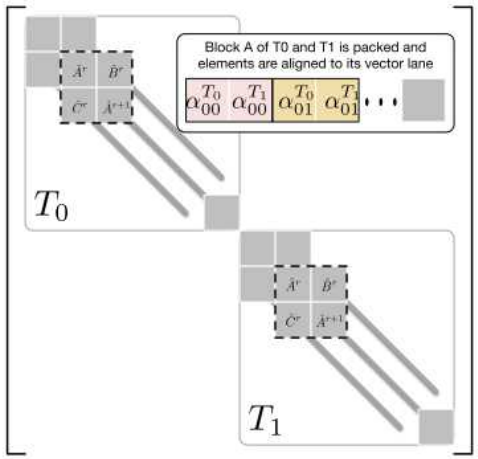
Kokkos Unified Programming Approach

```
using namespace Kokkos;
// Suppose that N = 64, b = 4

// Reinterpret the rank 3 vector view with the rank 4 internal vector type.
// AA: Host 8x4x4x1, Cuda 4x4x4x8
View<internal_vector_type****,LayoutRight> AA(Av.data(), N/vl, b, b, vl/il);

// Host policy(8,1,1), Cuda policy(gridSize=8,blockDim.y=8, blockDim.x=4)
// You can consider league_size equivalent to the number of independent works.
TeamPolicy<exec_space> policy(league_size = N/vl,
                              team_size = AUTO,
                              vector_loop_size = vl/il);
parallel_for(policy, KOKKOS_LAMBDA(const member_type member) {
  int p = member.league_rank();
  // CUDA: for (int v=threadIdx.x;v<vector_loop_size;v+=blockDim.x)
  parallel_for(ThreadVectorRange(member, vector_loop_size), [&](int v) {
    // Host A 4x4, Internal vector length 8.
    // Cuda A 4x4, Internal vector length 2.
    //             Consecutive indexing of v allows perfectly coalesced
    //             memory access along 16 doubles.
    auto A = subview(AA, p, ALL, ALL, v);
    // CUDA: for (int ij=threadIdx.y;v<b*b;ij+=blockDim.y)
    Kokkos::parallel_for(TeamThreadRange(member, b*b), [&](int ij) {
      int i = ij/b, j = ij%b;
      AA(i,j) = i == j ? one : zero;
    }); }); });
```

Example: Set identity of N matrices $b \times b$

Line Implicit Solver Implementation

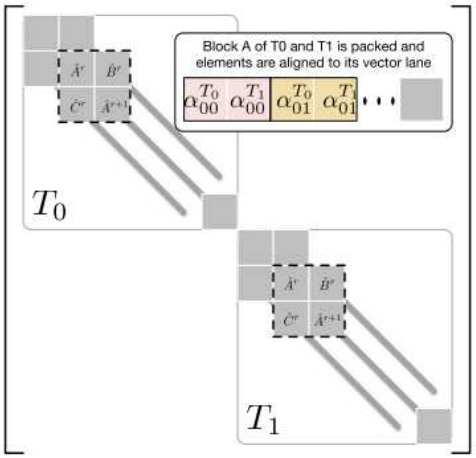
- KokkosKernels provides serial/team level dense linear algebra components so that users can compose their own batch operations.
- Solve phase of block tridiagonal matrices can be done in a similar fashion.



1 **for** *a pair* T *in* $\{(T_0,T_1),(T_2,T_3),\cdots,(T_{m\times n-2},T_{m\times n-1})\}$ **do in parallel**
2 **for** $r \leftarrow 0$ **to** $k-2$ **do**
3 $\hat{A}^r := LU(\hat{A}^r)$;
4 $\hat{B}^r := L^{-1}\hat{B}^r$;
5 $\hat{C}^r := \hat{C}^r U^{-1}$;
6 $\hat{A}^{r+1} := \hat{A}^{r+1} - \hat{C}^r\hat{B}^r$;
7 $\hat{A}^{k-1} := LU(\hat{A}^{k-1})$;

Line Implicit Solver Implementation

- KokkosKernels provides serial/team level dense linear algebra components so that users can compose their own batch operations.
- Solve phase of block tridiagonal matrices can be done in a similar fashion.




```
// Problem domain ( M x N ) x K, blocksize b
//   - M x N : the number of block tridiagonals (concurrency)
//   - K     : the length of block tridiagonals (workload per team)
View<vector_type*****,LayoutRight> Av(''A'', M*N/vl, K, 3, b, b);
View<internal_vector_type******,LayoutRight>
            AA(Av.data(), M*N/vl, K, 3, b, b, vl/il);

TeamPolicy<exec_space> policy(league_size = M*N/vl,
                              team_size = AUTO,
                              vector_loop_size = vl/il);
parallel_for(policy, KOKKOS_LAMBDA(const member_type member) {
  int p = member.league_rank();
  parallel_for(ThreadVectorRange(member, vector_loop_size), [&](int v) {
    for (int r=0;r<(K-1);++r) {
      auto A = Kokkos::subview(AA, p, r,   1, ALL(), ALL(), v);
      auto B = Kokkos::subview(AA, p, r,   2, ALL(), ALL(), v);
      auto C = Kokkos::subview(AA, p, r,   0, ALL(), ALL(), v);
      auto D = Kokkos::subview(AA, p, r+1, 1, ALL(), ALL(), v);
      TeamLU(member, A);
      TeamTrsm(member,  1.0, A, B);
      TeamTrsm(member,  1.0, A, C);
      TeamGemm(member, -1.0, C, B, 1.0, D);
    }
    auto A = Kokkos::subview(AA, p, K-1, 1, ALL(), ALL(), v);
    TeamLU(member, A);
  }); });
```

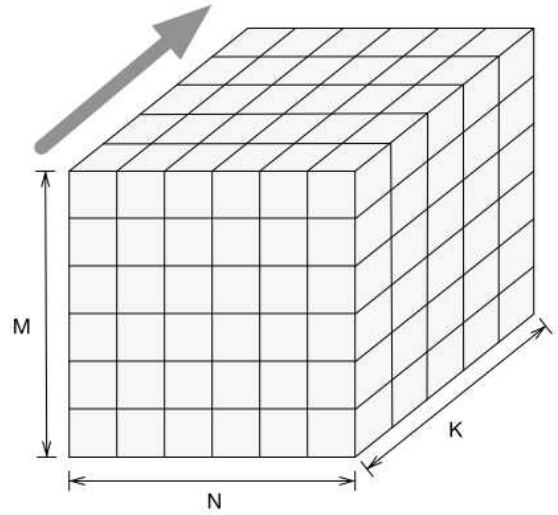
Numerical Experiments

Test Machines

- Intel Xeon Phi
- NVIDIA Tesla V100

Test Problems

- With limited device memory (16 GB), the following test problems are evaluated.

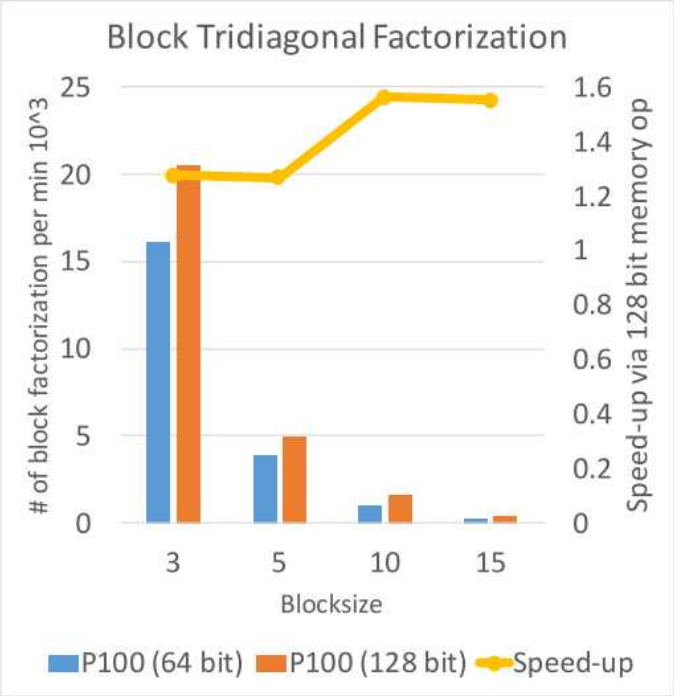


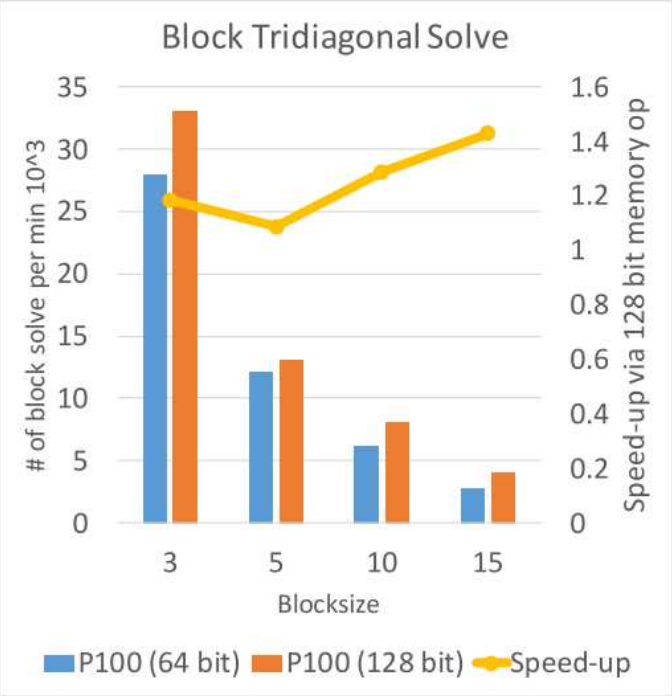
Blocksize	$M \times N \times K$
3	$128 \times 128 \times 128$
5	$128 \times 128 \times 128$
10	$64 \times 64 \times 128$
15	$64 \times 64 \times 128$

NVIDIA Tesla V100

- Both codes shows 100% memory store/load efficiency and performance is limited by L2.
- Bigger blocksize (more computational intensity) shows more benefits from 128 bit memory instructions.

Team (blockDim)	2×16	4×8
Global Store/Load Throughput (GB/s)	215.31 / 641.38	287.68 / 762.08
L2 Throughput Writes/Reads (GB/s)	215.31 / 530.37	287.68 / 677.41
Global Store/Load Transactions Per Request (Byte)	7.69 / 31.22	14.27 / 60.81

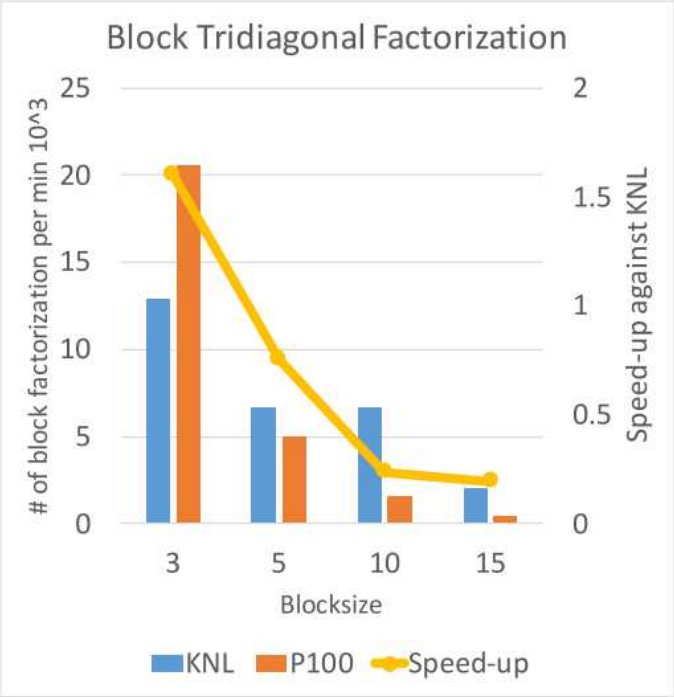


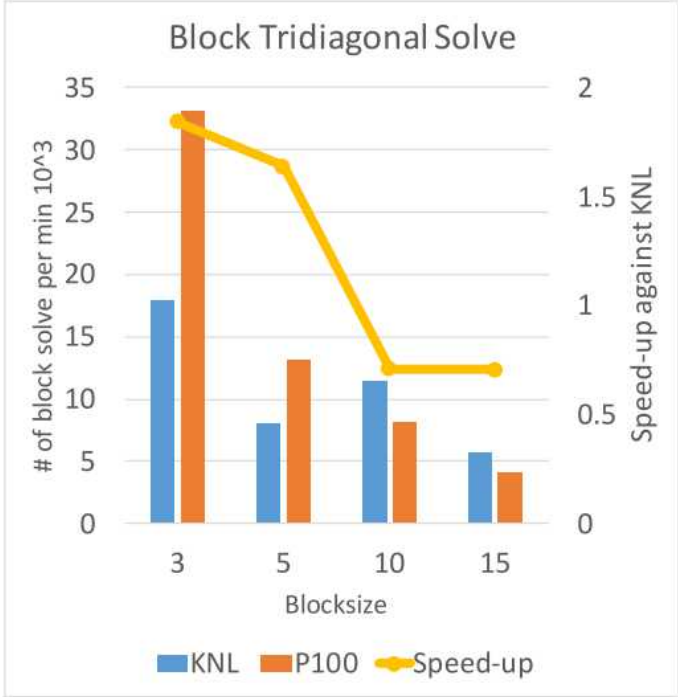


Intel KNL vs NVIDIA Tesla V100

- Block tridiagonal factorization/solves are inherently sequential.
- With increasing blocksize, the number of concurrent work (tridiagonals) is reduced while the workload per team increases.
- Need more parallelism for bigger problems.

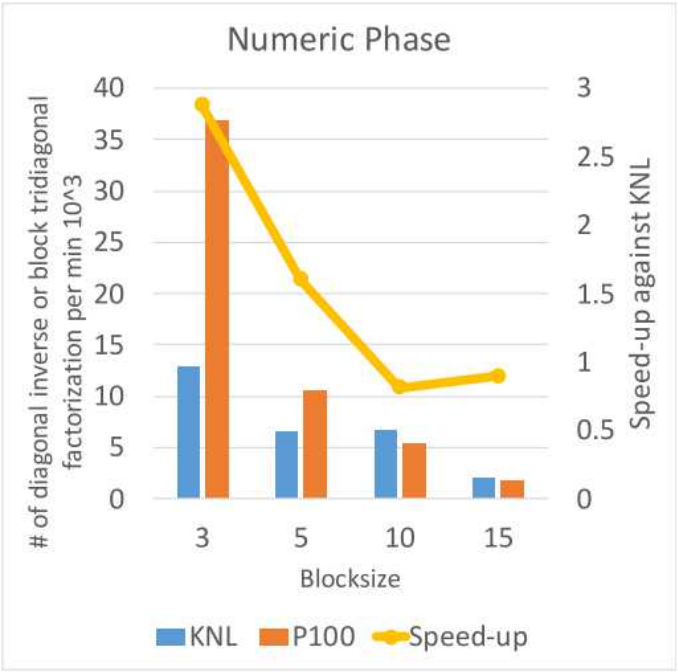
Cube Size($M \times N \times K$)	$128 \times 128 \times 128$	$64 \times 64 \times 128$
Multiprocessor Activity (%)	98.31	95.75
Achieved Occupancy	0.29	0.07

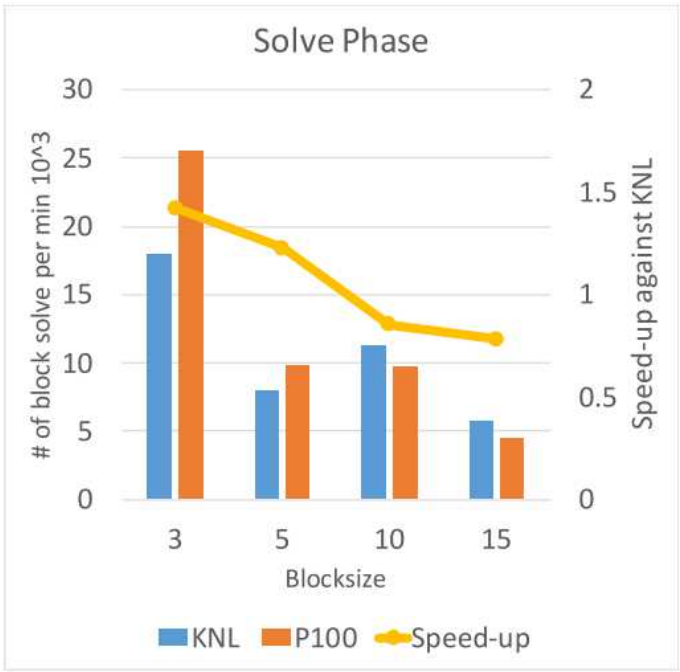




Block Jacobi Iterations for Solving Block Tridiagonals

- In transient flow, application (CFD) uses tiny time steps with relaxed error threshold - more nonlinear iterations and less number of iterations for solves.
- Apply a small number of Jacobi iterations instead of direct solutions (factorize, foward/backward solves).
- Improve performance on numeric phase (inverse diagonals).
- With increasing data movement during Jacobi iterations, there is not much benefit in the solve phase.





Comparison of Jacobi block tridiagonal solves (threshold 10^{-4}) on P100 against direct block tridiagonal solves on KNL

Summary

- Demonstrated performance portable SIMD approach for line implicit solver using Kokkos framework.
  - Prepare data for multidimensional array of SIMD type.
  - Reinterpret the multidimensional array based on execution space.
  - Construct `Kokkos::TeamPolicy` accordingly to exploit both compiler vectorization and dynamic vectorization.
- Improved GPU performance (10 - 50 %) by using 128 bit memory instructions (`double2` or (`float4`)); performance improvement is higher when computational intensity increases.
- Compared to highly optimized KNL code (same code), GPUs perform better for small blocksizes (3 and 5)
- Due to device memory constraint (lowering parallelism), it is difficult to use a GPU efficiently for bigger blocksizes (10 and 15).
- Easy solution: a GPU with a bigger device memory (32 or 64 GB).