Solving Many Small Matrix Problems using Kokkos and KokkosKernels

PRESENTED BY
KokkosKernels Team (UUR, SAND2019-4542 PE)
Standard Kokkos Approach using Hierarchical Parallelism

- Multidimensional array for many small matrices: \( N \times B \times B \).
  
  \[
  A = \text{Kokkos::View<double***, LayoutRight>}(\text{“A”}, N, B, B); 
  \]

- To exploit matrix level parallelism, use \text{Kokkos::TeamPolicy} mapping a chunk of workload to a team of threads.

  **Example:** \text{doSomething}
  
  \[
  \text{Kokkos::TeamPolicy policy(league_size=}=N, \text{ team_size=}=\text{AUTO, vector_size=}=\text{AUTO});} 
  \]
  \[
  \text{Kokkos::parallel_for(policy, LAMBDAGmember_type member) \{ \} } 
  \]
  \[
  \text{int p = member.league_rank();} 
  \]
  \[
  \text{auto A_at_p = Kokkos::subview(A, p, ALL, ALL);} 
  \]
  \[
  \text{Kokkos::parallel_for(TeamThreadRange(member, B), \[&\](int i) \{ \} } 
  \]
  \[
  \text{Kokkos::parallel_for(_ThreadVectorRange(member, B), \[&\](int j) \{ \} } 
  \]
  \[
  \text{A_at_p(i, j) = doSomething();} \]

  A team of threads (team x vector) is formed and perform \text{doSomething} in parallel.

- **Host:** \text{vector_size=1} and compiler auto-vectorization is expected for the inner most parallel loop.

- **CUDA:** team and vector corresponds to \text{blockDim.y} and \text{blockDim.x} respectively.

  **Challenge:** Small matrix sizes \((B = 3, 5, 7, 9, 15)\) are difficult to use wide vector units efficiently e.g., Intel AVX512 and 32 CUDA threads in a warp.
Compact Data Layout

- Array of matrices using 3-d scalar view:

\[ A = \text{Kokkos::View\{double***, LayoutRight\}("A", N, B, B); } \]

- Interleaved data layout using 4-d scalar view with a given vector length (VL):
  - Entries are **contiguous** along the depth direction (VL)

\[ A_s = \text{Kokkos::View\{double***[VL], LayoutRight\}("A", N/VL, B, B); } \]

- Two views can be mapped each other by \[ A_s(p/VL, i, j, p\%VL) = A(p, i, j). \]
Consider a Vector as Computing Unit (SIMD type)

- By overloading arithmetic operators, scalar operations are naturally auto-vectorized.

```cpp
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;
}

typedef vector_type Vector<double,VL>;
Av = Kokkos::View<vector_type***,LayoutRight>("Av", N/VL, B, B);
```

- The same numeric functor used for 3-d scalar view can be used for 3-d vector view.
How can we maintain a single version of code to vectorize on both CPUs and GPUs?
Compiler Vectorization and Dynamic (Warp) Vectorization

- **SIMD instructions on CPUs**
  - A single thread exploits data-level parallelism via SIMD instructions
  - Vector length is given from hardware specific vector instruction set e.g., AVX512.
  - A compiler is responsible for vectorization; using pragma or vector intrinsics, a user can force the vectorization.

- **Warp-based vectorization on GPUs**
  - Warp: a set of concurrent threads executing the same instructions.
  - 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.
  - Note that GPUs have 128 bit memory instruction set (double2 – built-in vector type).
    - Use of double2 type can improve memory efficiency by reducing the number of memory transactions.

- **We want to maintain a single vector parallel loop to express compiler vectorization and warp-based vectorization as well as the combination of the both.**
Define two vector types.

\[
\begin{align*}
\text{constexpr int VL} &= \text{VectorLength<exec_space>::value;} \\
\text{constexpr int IL} &= \text{InternalVectorLength<exec_space>::value;}
\end{align*}
\]

typedef Vector<double,VL> vector_type;
typedef Vector<double,IL> internal_vector_type;

Re-interpret a scalar view as vector views.

typedef LayoutRight LR;
View<double***[VL],LR> As("As", N/VL, B, B);
View<vector_type***,LR> Av(As.data(), N/VL, B, B);
View<internal_vector_type***[VL/IL],LR> AA(As.data(), N/VL, B, B);

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>VL</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>IL</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>AA.extent(3)</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
Illegal Kokkos Hierarchical Parallelism

- Team Policy
  TeamPolicy<exec_space> policy(league_size = N/VL,
    team_size = AUTO,
    vector_size = VL/IL );

- Dispatch nested parallel loop.
  
  \[
  \text{Kokkos::parallel_for(policy, LAMBDA(member_type member) { \\
  \text{int p = member.league_rank();} \\
  \text{Kokkos::parallel_for(ThreadVectorRange(member, VL/IL), [\&](int v) { \\
  \text{auto Ap = Kokkos::subview(AA, p, ALL, ALL, v);} \\
  \text{Kokkos::parallel_for(TeamThreadRange(member, B*B), [\&](int ij) { \\
  \text{const int i = ij/B, j = ij}\%B;} \\
  \text{Ap(i, j) = doSomething();} \\
  \text{Ap(i, j) = doSomething();} \\
  \text{Ap(i, j) = doSomething();} \\
  \text{Ap(i, j) = doSomething();}}} \\
  \})});}}\]

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>team size</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>vector size</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
Why do we want to put the vector loop first?

- KokkosKernels provides layered functor level interface for BLAS and LAPACK functions allowing users to compose their own batched dense linear algebra.
  - Serial: no nested parallel_for is used inside of the functions;
  - Vector: a nested parallel_for with ThreadVectorRange is used;
  - Team: a nested parallel_for with TeamThreadRange is used;
  - TeamVector: two nested parallel_for with both TeamThreadRange and ThreadVectorRange are used inside.

Example:

```cpp
Kokkos::parallel_for(policy, LAMBDA(member_type member) {
    int p = member.league_rank();
    Kokkos::parallel_for(ThreadVectorRange(member, VL/IL), [&](int v) {
        auto Ap = Kokkos::subview(AA, p, ALL, ALL, v); //Extract a matrix view
        Kokkos::parallel_for(TeamThreadRange(member, B*B), [&](int ij) {
            const int i = ij/B, j = ij%B;
            Ap(i, j) = doSomething(); //This can be replaced by a function
            //KokkosBatched::TeamDoSomething(Ap)
        });
    });
});
```
Example: Line Implicit Solver (Block Tridiagonal Solver)

- A block sparse system of equations arises from CFD problems.
- Lines are formed in boundary layers to resolve shocks.
- Small blocks (3, 5, 9, and 15) represent interactions among a group of variables.

```
// Problem domain: (MxN)xK, blocksize B
View<internal_vector_type>*****[VL/IL],LR> AA("Av", MN/VL, K, 3, B, B);
TeamPolicy<exec_space> policy(MN/VL, AUTO, VL/IL); // Compact Layout
parallel_for(policy, LAMBDA(member_type member) {
  const double one(1);
  int p = member.league_rank();
  parallel_for(ThreadVectorRange(VL/IL)
    for (int r=0; r<(K-1); ++r) {
      auto A = subview(AA, p, r, 1, ALL, ALL, v);
      auto B = subview(AA, p, r, 2, ALL, ALL, v); // Extract
      auto C = subview(AA, p, r, 0, ALL, ALL, v); // matrix views
      auto D = subview(AA, p, r+1, 1, ALL, ALL, v);
      TeamLU(member, A);
      TeamTrsm(member, one, A, B);
      TeamTrsm(member, one, A, C);
      TeamGemm(member, -one, C, B, one, D);
    }
  auto A = subview(AA, p, K-1, 1, ALL, ALL, v);
  TeamLU(member, A);
}); });
```

Vectorized Block Tridiagonal Factorization
Numerical Experiments

- **Test Machines**
  - Intel Xeon Phi 7250, Knight Landing - 34 Tiles, 2 cores per tile @ 1.4GHz, w/ HBM.
  - NVIDIA P100 - 56 SMs, 32 FP64 CUDA cores per SM @ 1.126 GHz.

- **Test Problems**

<table>
<thead>
<tr>
<th>Blocksize</th>
<th>$M \times N \times K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$128 \times 128 \times 128$</td>
</tr>
<tr>
<td>5</td>
<td>$128 \times 128 \times 128$</td>
</tr>
<tr>
<td>10</td>
<td>$64 \times 64 \times 128$</td>
</tr>
<tr>
<td>15</td>
<td>$64 \times 64 \times 128$</td>
</tr>
</tbody>
</table>
Both codes show 100% memory store/load efficiency and performance is limited by L2.
NVIDIA P100 vs Intel KNL

- With an increasing blocksize, the number of block tridiagonals (concurrency) decreases while the parallelism within the matrix increases.

- Instead of using AUTO variables in determining team size, a heuristic is required to determine the optimal team size for different block sizes.
Summary

- Demonstrated performance portable compact batched linear algebra approach to develop a line implicit solver.

- Efficient vectorization can be achieved by blending compiler vectorization and warp-level vectorization.

- Kokkos programming model officially does not support what I presented but it works.

- KokkosKernels provides:
  - `struct Vector<T,VL>` and its specialization for Intel and NVIDIA architectures.
  - Serial, Vector, Team and TeamVector level functor interface for BLAS and LAPACK functions.

- Codes are available for testing
  - `github.com/kokkos/kokkos-kernels/perf_test/batched/KokkosBatched_Test_BlockTridiagDirect.cpp`