KokkosKernels: Compact Layouts for Batched Blas and Sparse Matrix-Matrix multiply

Siva Rajamanickam,
Kyungjoo Kim, Andrew Bradley, Mehmet Deveci, Christian Trott,
Si Hammond

Batched BLAS Workshop, 2017, Atlanta
KokkosKernels: Overview

- Layer of **performance portable** kernels on top of Kokkos
  - Sparse linear algebra kernels
  - Dense linear algebra kernels (Batched BLAS as well as traditional BLAS)
  - Graph kernels
  - Tensor Contraction kernels (upcoming)
KokkosKernels : Overview

- No dependencies other than Kokkos
- Node-level only (No MPI)
- Provide kernels for all-levels of parallelism (wherever applicable) : Device level, Team level, Thread level, Serial
- Copyright received last week
  - Will reside in the Kokkos github organization
KokkosKernels: Current Kernels

- Sparse linear algebra kernels
  - CrsMatrix – fill
  - Sparse Matrix Vector Multiply
  - Sparse Matrix Matrix Multiply – Mehmet Deveci
  - (Symmetric) Gauss Seidel
- Dense linear algebra kernels (BLAS)
  - BLAS1, some BLAS2
  - Batched BLAS – Kyungjoo Kim
- Graph kernels
  - Graph coloring
- Other Utilities
  - HashMap
  - Uniform Memory Allocator
Motivation for Batched BLAS with Compact Layouts

- Sandia application characteristics
  - One dimension of the mesh more important than the others when preconditioning
  - Multiple degrees of freedom per element gives rise to tiny blocks
Motivation for Batched BLAS/LAPACK

- Block Jacobi preconditioner where each block is a Tridiagonal matrix
- Every scalar in the tridiagonal matrix is a small block matrix
  - Block sizes 5x5, 9x9, 15x15 etc
- Typical number of diagonal blocks 512-1024
- Key kernels needed DGEMM, LU, TRSM

Algorithm 1: Reference impl. TriLU

```plaintext
for T in \{T_0, T_1, \ldots, 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}^rU^{-1};
    \hat{A}^{r+1} := \hat{C}^{r+1} - \hat{C}^r\hat{B}^r;
  end
  \hat{A}^{k-1} := \{L \cdot U\};
end
```
KokkosKernels Compact Layouts for Batched BLAS

### Algorithm 2: Batched impl. TriLU

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

- Data Layout for better vector intrinsics
  - Pack entries from up to vlen block diagonal matrices, vlen is the vector length (vector length = 2 shown)
  - Use vector intrinsics on the new data vector data with operator overloading
- Scalar Performance is due to explicit loop unrolling
Path Forward for Compact Layouts for Batched BLAS

- KokkosKernels:
  - A Performance-Portable Reference Implementation for compact layouts
- Collaborations would be ideal
  - Intel MKL team for compact layouts in MKL (ongoing)
    - Thanks to T. Costa, M. Guney, S. Knepper, S. Story
    - Disseminate the ideas to broader community
  - “Shared Fate Milestones” (Exascale Computing Project) with the MAGMA team
    - Thanks to S. Tomov, J. Dongarra
  - Extend the work to other kernels (E.g: Tensor contractions)
Compact Layouts for Batched BLAS: Experiments

MKL Test Setup

```cpp
Kokkos::parallel_for(Kokkos::RangePolicy(N),
    KOKKOS_LAMBDA(const int k) {
        auto aa = Kokkos::subview(a, k, Kokkos::ALL(), Kokkos::ALL());
        auto bb = Kokkos::subview(b, k, Kokkos::ALL(), Kokkos::ALL());
        auto cc = Kokkos::subview(c, k, Kokkos::ALL(), Kokkos::ALL());

        cblas_dgemm(CblasRowMajor,
            CblasNoTrans, CblasNoTrans,
            BlkSize, BlkSize, BlkSize,
            1.0,
            (double*)aa.data(), aa.stride_0(),
            (double*)bb.data(), bb.stride_0()
            1.0,
            (double*)cc.data(), cc.stride_0();
    });
```

```cpp
MKL_INT blksize[1] = { BlkSize };
MKL_INT lda[1] = { a.stride_1() };
MKL_INT ldb[1] = { b.stride_1() };
MKL_INT ldc[1] = { c.stride_1() };
CBLAS_TRANSPOSE transB[1] = { CblasNoTrans };
double one[1] = { 1.0 };
MKL_INT size_per_grp[1] = { N };
```

```cpp
cblas_dgemm_batch(CblasRowMajor,
    transA, transB,
    blksize, blksize, blksize,
    one,
    (const double**)aa, lda,
    (const double**)bb, ldb,
    one,
    cc, ldc,
    1,
    size_per_grp);
```
Compact Layouts for Batched BLAS: Experiments

KokkosKernels Test Setup

// Scalar version
Kokkos::View<double***,HostSpaceType>
   a("a", N, BlkSize, BlkSize);

// Vector version
Kokkos::View<Vector<VectorTag<AVX<double>,4> >***,HostSpaceType>
   a("a", N/VectorLength, BlkSize, BlkSize),

Kokkos::parallel_for( Kokkos::RangePolicy(/* N or N/VectorLength */),
   KOKKOS_LAMBDACONST(int k) {
      auto aa = Kokkos::subview(a, k, Kokkos::ALL(), Kokkos::ALL());
      auto bb = Kokkos::subview(b, k, Kokkos::ALL(), Kokkos::ALL());
      auto cc = Kokkos::subview(c, k, Kokkos::ALL(), Kokkos::ALL());

      KokkosKernels::Serial::
         Gemm<Trans::NoTranspose,Trans::NoTranspose,AlgoTagType>::
            invoke(1.0, aa, bb, 1.0, cc); }
});
KokkosKernels Batched BLAS : DGEMM Performance

KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>DGEMM GFLOP/s</th>
<th>Blocksize</th>
<th>Speedup w.r.t. MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>1.64</td>
<td>2.58</td>
<td>2.45</td>
</tr>
<tr>
<td>2</td>
<td>2.70</td>
<td>4.90</td>
<td>4.77</td>
</tr>
<tr>
<td>4</td>
<td>3.64</td>
<td>8.72</td>
<td>9.46</td>
</tr>
<tr>
<td>8</td>
<td>4.15</td>
<td>13.3</td>
<td>18.3</td>
</tr>
<tr>
<td>16</td>
<td>4.25</td>
<td>16.3</td>
<td>33.9</td>
</tr>
<tr>
<td>34</td>
<td>3.77</td>
<td>15.4</td>
<td>53.2</td>
</tr>
<tr>
<td>68</td>
<td>3.91</td>
<td>16.1</td>
<td>76.3</td>
</tr>
</tbody>
</table>

- Intel Knights Landing Architecture
- GFLOP/s (numbers) and speedup w.r.t MKL (colors) shown for 512 worksets
- Data flushed after each GEMM
## KokkosKernels Batched BLAS: TRSM Performance

**KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>TRSM GFLOP/s</th>
<th>Blocksize 5</th>
<th>Blocksize 9</th>
<th>Blocksize 15</th>
<th>Blocksize 20</th>
<th>Speedup w.r.t. MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.80</td>
<td>1.61</td>
<td>1.90</td>
<td>3.83</td>
<td></td>
<td>33.9</td>
</tr>
<tr>
<td>2</td>
<td>1.06</td>
<td>2.79</td>
<td>3.57</td>
<td>7.16</td>
<td></td>
<td>28.6</td>
</tr>
<tr>
<td>4</td>
<td>1.07</td>
<td>3.89</td>
<td>6.35</td>
<td>13.0</td>
<td></td>
<td>23.2</td>
</tr>
<tr>
<td>8</td>
<td>1.06</td>
<td>4.58</td>
<td>10.4</td>
<td>21.8</td>
<td></td>
<td>17.9</td>
</tr>
<tr>
<td>16</td>
<td>0.96</td>
<td>4.40</td>
<td>14.6</td>
<td>31.9</td>
<td></td>
<td>12.5</td>
</tr>
<tr>
<td>34</td>
<td>0.80</td>
<td>3.84</td>
<td>16.6</td>
<td>38.2</td>
<td></td>
<td>7.2</td>
</tr>
<tr>
<td>68</td>
<td>0.84</td>
<td>3.66</td>
<td>18.8</td>
<td>43.2</td>
<td></td>
<td>1.8</td>
</tr>
</tbody>
</table>

- Intel Knights Landing Architecture
- GFLOP/s (numbers) and speedup w.r.t MKL (colors) shown for 512 worksets
- Data flushed after each TRSM
### KokkosKernels Batched BLAS : LU Performance

**KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>LU GFLOP/s</th>
<th>Blocksize</th>
<th></th>
<th></th>
<th></th>
<th>Speedup w.r.t. MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>0.53</td>
<td>1.26</td>
<td>1.73</td>
<td>3.18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>0.69</td>
<td>1.93</td>
<td>3.25</td>
<td>6.01</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>0.68</td>
<td>2.68</td>
<td>5.60</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>0.69</td>
<td>3.04</td>
<td>8.52</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>5</td>
<td>0.64</td>
<td>3.03</td>
<td>11.2</td>
<td>24.2</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>5</td>
<td>0.53</td>
<td>2.57</td>
<td>11.6</td>
<td>27.6</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>5</td>
<td>0.59</td>
<td>2.52</td>
<td>13.3</td>
<td>29.8</td>
</tr>
</tbody>
</table>

- Intel Knights Landing Architecture
- GFLOP/s (numbers) and speedup w.r.t MKL (colors) shown for 512 worksets
- Data flushed after each LU
Performance comparisons for Large-Block Jacobi Small-Block Tridiagonal factorization and Triangular Solve

- One right hand side per solve
- Speedups against a hand-tuned version of the code within the application
Compact Layouts for Batched BLAS: Discussion

• Path forward
  – Batched implementation of other kernels than DGEMM
  – Integrating with other linear algebra codes (FASTILU, direct methods)
  – Implementation of Compact/Packed Layouts in other libraries

• Smaller block sizes are an important use case for Sandia applications

• Need careful interface design for reuse of the structure

• C++20 standardization of the “packed double” or SIMT vector
Sparse Matrix-Matrix Multiplication (SpGEMM)
Sparse Matrix-Matrix Multiplication Problem

• SPGEMM: fundamental block for
  – Algebraic multigrid
  – Various graph analytics problems: clustering, betweenness centrality...

• Extra irregularity: nnz of C is unknown beforehand.
SpGEMM: Previous Work

• **Distributed Memory algorithms:**
  – 1D Trilinos, 2D Combinatorial Blas [Buluç 12],
    3D [Azad 15], Hypergraph-based: [Akbudak 14], [Ballard 16]
• Most of the shared algorithms are based on 1D-Gustavson algorithm [Gustavson 78]
• **Multi-threaded algorithms:**
  – Dense Accumulator [Patwary 15]
  – Sparse Heap accumulators: ViennaCL, CommBlass
  – Sparse accumulators: MKL
• **GPUs:**
  – CUSP: 3D outer product $O$(FLOPS) memory
  – Hierarchical: cuSPARSE, bhSparse [Liu 14]
KokkosKernels Portable SPGEMM Method

- Two Phase Method: Symbolic and Numeric Phase
- Each team works on a bunch of rows
  - Team: Block (GPU), group of hyperthreads in core (CPU)
- Each worker in team works on consecutive rows.
  - Worker: Warp (GPUs), hyperthread (CPU)
  - More coalesced access on GPUs, better L1-cache usage on CPUs.
- Each vectorlane in a worker works on a different multiplication within a row:
  - Vectorlane: Threads in a Warp (GPUs), vector units (CPU)

*See Mehmet Deveci’s talk on Tuesday @CSE for more details*
• Comparing KokkosKernels SPGEMM and two SPGEMM in Intel MKL and ViennaCL on Intel Knights Landing
• Geometric Mean Speedups w.r.to sequential KokkosKernel SPGEMM for 20 different matrix multiplications
• Reusing the symbolic structure is key to better performance on applications
### KokkosKernels Portable SPGEMM Method on GPUs

<table>
<thead>
<tr>
<th>KKMEM</th>
<th>time</th>
<th>gflops</th>
<th>CUSP</th>
<th>bhSPARSE</th>
<th>ViennaCL</th>
<th>cuSPARSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2cubes_sphere</td>
<td>0.02</td>
<td>3.631</td>
<td>4.54</td>
<td>1.20</td>
<td>1.06</td>
<td>3.62</td>
</tr>
<tr>
<td>cage12</td>
<td>0.03</td>
<td>2.396</td>
<td>3.13</td>
<td>0.75</td>
<td>1.22</td>
<td>2.74</td>
</tr>
<tr>
<td>webbase</td>
<td>0.27</td>
<td>0.521</td>
<td>0.66</td>
<td>0.54</td>
<td>5.18</td>
<td>2.30</td>
</tr>
<tr>
<td>offshore</td>
<td>0.03</td>
<td>4.304</td>
<td>5.25</td>
<td>1.33</td>
<td>1.21</td>
<td>7.08</td>
</tr>
<tr>
<td>filter3D</td>
<td>0.03</td>
<td>4.918</td>
<td>5.78</td>
<td>0.83</td>
<td>1.47</td>
<td>4.30</td>
</tr>
<tr>
<td>hugebubbles20_0</td>
<td>0.10</td>
<td>3.804</td>
<td>4.99</td>
<td>4.81</td>
<td>1.94</td>
<td>12.14</td>
</tr>
<tr>
<td>Europe</td>
<td>0.18</td>
<td>2.669</td>
<td>3.41</td>
<td>5.57</td>
<td>2.57</td>
<td>2.50</td>
</tr>
<tr>
<td>cant</td>
<td>0.04</td>
<td>12.001</td>
<td>12.83</td>
<td>1.05</td>
<td>1.42</td>
<td>0.77</td>
</tr>
<tr>
<td>hood</td>
<td>0.08</td>
<td>13.944</td>
<td>14.22</td>
<td>0.97</td>
<td>1.77</td>
<td>1.72</td>
</tr>
<tr>
<td>ptk</td>
<td>0.07</td>
<td>17.717</td>
<td>17.88</td>
<td>1.13</td>
<td>2.06</td>
<td>1.53</td>
</tr>
<tr>
<td>Empire_R_AP</td>
<td>0.04</td>
<td>4.734</td>
<td>0.89</td>
<td>0.65</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>Empire_RA_P</td>
<td>0.08</td>
<td>2.316</td>
<td>1.03</td>
<td>0.41</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>Laplace_RA_A</td>
<td>0.39</td>
<td>2.041</td>
<td>0.68</td>
<td>0.73</td>
<td>2.71</td>
<td></td>
</tr>
<tr>
<td>Laplace_RA_P</td>
<td>0.15</td>
<td>3.398</td>
<td>2.57</td>
<td>1.00</td>
<td>11.65</td>
<td></td>
</tr>
<tr>
<td>Laplace_R_AP</td>
<td>0.19</td>
<td>5.466</td>
<td>2.36</td>
<td>1.24</td>
<td>5.24</td>
<td></td>
</tr>
<tr>
<td>Laplace_RA_P</td>
<td>0.47</td>
<td>2.203</td>
<td>1.67</td>
<td>0.65</td>
<td>3.32</td>
<td></td>
</tr>
<tr>
<td>Brick_R_A</td>
<td>0.64</td>
<td>2.381</td>
<td>1.16</td>
<td>1.82</td>
<td>4.91</td>
<td></td>
</tr>
<tr>
<td>Empire_R_A</td>
<td>0.43</td>
<td>5.934</td>
<td>1.09</td>
<td>1.06</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>Empire_A_P</td>
<td>0.30</td>
<td>8.463</td>
<td>3.60</td>
<td>1.05</td>
<td>1.48</td>
<td></td>
</tr>
<tr>
<td>Brick_RA_P</td>
<td>0.61</td>
<td>6.326</td>
<td>1.26</td>
<td>0.43</td>
<td>1.14</td>
<td></td>
</tr>
<tr>
<td>Idoor</td>
<td>0.32</td>
<td>14.910</td>
<td>1.09</td>
<td>1.88</td>
<td>1.76</td>
<td></td>
</tr>
<tr>
<td>delaunay_n24</td>
<td>0.41</td>
<td>3.086</td>
<td>1.74</td>
<td>1.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brick_R_AP</td>
<td>0.24</td>
<td>6.349</td>
<td>0.76</td>
<td>1.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>channel</td>
<td>0.43</td>
<td>7.054</td>
<td>1.51</td>
<td>3.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brick_AP</td>
<td>0.49</td>
<td>7.954</td>
<td>0.95</td>
<td>4.54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cage15</td>
<td>1.56</td>
<td>2.660</td>
<td>1.77</td>
<td>4.86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bump</td>
<td>0.88</td>
<td>13.126</td>
<td>1.58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>audit</td>
<td>1.31</td>
<td>12.345</td>
<td>1.54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dieFilterV3real</td>
<td>1.80</td>
<td>9.679</td>
<td>1.85</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Comparing KokkosKernels SPGEMM and four other GPU implementations (CUSPARSE, CUSP, bhSparse, ViennaCL)
- Both multigrid and data analysis style multiplications
- Reusing the symbolic structure is key to better performance on applications

The matrices and multiplications used throughout this paper. The (#rows, #cols, #nnz) of the input matrices and #multiplications performed are...
Sparse Matrix-Matrix multiplication Discussion

- Raising Importance of SPGEMM
  - Data Analysis community is driving lot of the work
  - Key for scalability of algebraic multigrid setup
- An opportunity to address a gap for important applications
- Addressing the symbolic reuse portion is an important usecase for several applications
- One performance-portable reference implementation available
  - Vendor collaborations and other reference implementation needed
Thank you
srajam@sandia.gov
KokkosKernels Portable SPGEMM Method

• 2 level Hashmap Accumulator:
  – 1\textsuperscript{st} level uses GPUs shared memory or a small memory that will fit in L1 cache
  – 2\textsuperscript{nd} level goes to global memory

• Uniform Memory Pool:
  – Only some of the workers need 2\textsuperscript{nd} level hash map. They request memory from memory pool.

• Compression: Symbolic works performs unions on rows. Binary relations that can be done with BitWiseO