IN-PLACE ARRAY TRANSPOSITION AND FAST ARRAY OF STRUCTURES ACCESSES

Bryan Catanzaro
Transposition shows up in many places

- Linear algebra
- Image processing
- Databases
- FFTs

It is the core of data layout choices

- Array of Structures versus Structures of Arrays
- Huge performance impact for SIMD processors
In-place transposition

Don’t you just swap $A[i,j]$ with $A[j,i]$?
- Yes, iff matrix is square
- Otherwise...

![Logical view and row-major linearization diagrams]
Cycle shifting

Break permutation into independent cycles

Input

Output

For more, see: [http://en.wikipedia.org/wiki/In-place_matrix_transposition](http://en.wikipedia.org/wiki/In-place_matrix_transposition)
Problems with cycle shifting

- Storing vs. recomputing cycles
  - Leads to $O(mn \log mn)$ complexity for less than $O(mn)$ auxiliary space [Knuth, TAOCP Volume 3, 1973]

- Cycle lengths are badly distributed
  - Leads to load imbalance if parallelizing cycles

- Complicated!
  - Tricky, irregular, limits application scope
Decomposition

Let’s keep the 2-d view of the original matrix

Decompose the permutation into independent row and column permutations

Like the puzzle
Decomposition

- Within each row, send each element to its destination column
- Within each column, send each element to its destination row
- Decomposes the large permutation into many small permutations
  - No cycle shifting!
Removing Conflicts

If $m$, $n$ are not coprime, row shuffle has conflicts.
Removing Conflicts

We prove these conflicts can be removed.

If $m$, $n$ are not coprime, row shuffle has conflicts.

Column rotation.
Index Equations

\[ c = \gcd(m, n) \quad a = \frac{m}{c} \quad b = \frac{n}{c} \]

Conflict removal (Column rotation)

\[ r_j(i) = \left( i + \left\lfloor \frac{j}{b} \right\rfloor \right) \mod m \]

Row shuffle

\[ d'_i(j) = \left( \left( i + \left\lfloor \frac{j}{b} \right\rfloor \right) \mod m + jm \right) \mod n \]

Column shuffle

\[ s'_j(i) = \left( j + in - \left\lfloor \frac{i}{a} \right\rfloor \right) \mod m \]

- We prove these functions are bijective
  - This enables our decomposition
  - We give their inverses in [PPoPP2014]
```python
def c2r(A):
    """Algorithm 1""
    m, n = A.shape

    #Allocate temporary space
    #with max(m, n) elements
    tmp = np.zeros(max(m, n))

    if gcd(m, n) > 1:  #Removing conflicts
        for j in range(n):
            #Gather with r(i, j)
            for i in range(m):
                tmp[i] = A[r(i, j), j]
        for i in range(m):
            A[i, j] = tmp[i]

    for i in range(m):  #Row shuffles
        #Scatter with d_prime(i, j)
        for j in range(n):
            tmp[d_prime(i, j)] = A[i, j]
        for j in range(n):
            A[i, j] = tmp[j]

    for j in range(n):  #Column shuffles
        #Gather with s_prime(i, j)
        for i in range(m):
            tmp[i] = A[s_prime(i, j), j]
        for i in range(m):
            A[i, j] = tmp[i]
```
Implementation, optimizations

- Algorithm is naturally parallel & load balanced
- Use only gathers, no scatters
  - Requires analytic inverses of all index equations
- Strength reduction for integer division & modulus
- Decompose column shuffle
  - Row permute
  - Column rotate
- Cache aware primitives
  - Row permute and column rotate
Experimental Results - CPU

- Random sized DP matrices
  - \( m, n \in [1000, 10000) \)
- Simple CPU implementation
  - (No SIMD or cache opt.)
- Intel Core i7 950 (4 C, 8 T)
- Median throughput
  - MKL: 0.07 GB/s
  - C2R, 1 Thread: 0.34 GB/s
  - C2R, 8 Threads: 1.26 GB/s
  - [Gustavson 2012] : 1.27 GB/s
Experimental Results - GPU

- Uses all optimizations
- NVIDIA Tesla K20c
- Random matrices
  - $m, n \in [1000, 20000)$
- Median performance
  - [Sung 2013]: 5.33 GB/s (SP)
  - C2R: 14.2 GB/s (SP)
  - C2R: 19.5 GB/s (DP)
Arrays of Structures

- Transposition converts between
  - Arrays of Structures
  - Structures of Arrays

- Specialized C2R transpose implementation for small $m$ or $n$

- Median performance on Tesla K20c: 34.3 GB/s
**SIMD Vector Memory Accesses**

**Transpose without on-chip memory!**

Possible because our algorithm is so simple

Allows transpose to be hidden in C++ pointer type:

- `coalesced_ptr<T>`
- Dereferencing this type invokes transpose
**SIMD Vector Memory Accesses**

- C2R algorithm can operate in-place on the register file
  - Uses SIMD shuffle, no SIMD divergence or bank conflicts

- Enables arbitrary length SIMD vector loads and stores
  - Up to 45x faster than direct access

**Unit-stride AoS Accesses**

- Diagram showing GB/s vs. Size of structure (bytes)

**Random AoS Accesses**

- Diagram showing GB/s vs. Size of structure (bytes)
Summary

- New algorithm for in-place matrix transposition
  - Optimal work complexity: $O(mn)$ with only $O(\max(m, n))$ space

- Unoptimized CPU implementation: 1.26 GB/s
  - Competitive with optimized implementation from [Gustavson 2012]

- Median performance of 19.5 GB/s on Tesla K20c GPU
  - 3x faster than [Sung 2013] implementation (SP)

- Fast in-place conversions between AoS and SoA
  - 34.3 GB/s median performance on K20c

- New technique for dealing with vector memory accesses on SIMD
  - Up to 180 GB/s on K20c, 45x faster than direct accesses
Contacts

Matrix transposition routines:
- [http://github.com/BryanCatanzaro/inplace.git](http://github.com/BryanCatanzaro/inplace.git)

AoS routines:
- [http://github.com/BryanCatanzaro/trove.git](http://github.com/BryanCatanzaro/trove.git)

bcatanzaro@nvidia.com