Fast In-place Transposition

I-Jui Sung, University of Illinois
Juan Gómez-Luna, University of Córdoba (Spain)
Wen-Mei Hwu, University of Illinois
Full Transposition

- Full transposition is desired for many algorithms
  - E.g. FFT
Full Transposition

- Full transposition is desired for many algorithms
  - E.g. FFT
Full Transposition

- Full transposition is desired for many algorithms
- E.g. FFT
Full Transposition

- Full transposition is desired for many algorithms
  - E.g. FFT
Why Full In-Place Transposition on GPU?

- Transposition is obviously memory bound
  - Sheer raw memory bandwidth on GPU
- Big data sets
  - May not be able to afford the 2X spatial overhead for out-of-place transposition
Why Fast In-Place Transposition is Difficult on GPUs

- “Following-the-cycles”
- No locality

A \begin{align*}
(0,0) & (1,0) (2,0) (0,1) (1,1) (2,1) (0,2) (1,2) (2,2) (0,3) (1,3) (2,3) (0,4) (1,4) (2,4)
\end{align*}

A' \begin{align*}
(0,0) & (1,0) (2,0) (3,0) (4,0) (0,1) (1,1) (2,1) (3,1) (4,1) (0,2) (1,2) (2,2) (3,2) (4,2)
\end{align*}
Why Fast In-Place Transposition is Difficult on GPUs

- The amount of cycles is not enough (except square matrices)
Plan

- Let us look at transposing square matrices first
- Then near-square matrices, if we can pad them
- General transposition
In-Place Transposition of Square Matrices

Throughput of In-Place Transposing an $N \times N$ Square Matrix (Tesla K20)

- **CULA**
- **My Implementation**

Throughput in GB/s versus $N$
Near Square Matrices

- **Case I**: adding more rows (trivial for row-major layout)

- **Case II**: adding more columns
  - Observation: when there are spaces available, multiple rows can be moved in parallel without additional temporary storage
  - a) Before padding
  - b) After padding

  Starting from the last few rows, and then move the next few rows, so on so forth
Parallel In-Place Padding

- If space is enough we can move multiple rows w/o overlapping with other rows asynchronously

- In this case, Row 3 and Row 4 can be moved in parallel, but Row 2 needs to be moved in another iteration
  - \( \text{MovableRows} = \frac{(\text{TotalRows} - \text{RowsMoved}) \times C}{\text{RowSize}} \)
Parallel Padding for Near-Square Matrices

- Throughput is limited by spaces available for padding

Throughput of Parallel Padding 5Kx4.9K Matrix to Square, on Tesla K20 (Kepler)

Effective throughput = 38.2 GB/s
Plan

- Let us look at transposing full square matrices
  - Reached 45~51% peak on Kepler (208GB/s peak mem b/w)
- Maybe near-square matrices if we can pad the matrix
  - Parallelism can be limited.
- General transposition
Improving Locality by Tiled Transposition

moving small data pieces over large distances
Improving Locality by Tiled Transposition

Step 1: move large chunks over large distances

 transpose

8x6 matrix as 8x2x3 array. Transpose to 2x8x3 array
Improving Locality by Tiled Transposition

Step 1: move large chunks over large distances

transpose
Improving Locality by Tiled Transposition

Step 2: move small data pieces within small area
**Improving Locality by Tiled Transposition**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
</tbody>
</table>

**Step 3: move large chunks over large distances**

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>1</th>
<th>7</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>19</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
<td>41</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
</tbody>
</table>

**Transposing**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
<td>25</td>
<td>31</td>
<td>37</td>
<td>43</td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>34</td>
<td>40</td>
<td>46</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>12</th>
<th>18</th>
<th>24</th>
<th>30</th>
<th>36</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>13</td>
<td>19</td>
<td>25</td>
<td>31</td>
<td>37</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>14</td>
<td>20</td>
<td>26</td>
<td>32</td>
<td>38</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>15</td>
<td>21</td>
<td>27</td>
<td>33</td>
<td>39</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td>34</td>
<td>40</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>17</td>
<td>23</td>
<td>29</td>
<td>35</td>
<td>41</td>
<td>47</td>
</tr>
</tbody>
</table>
Are you able to reproduce that sequence of transposition?

We need a simpler way to refer to these transpositions

We are essentially permuting dimensions

Factorial numbering

- A B C → B A C = 100₁
  - Second, First, First
- A B C → A C B = 010₁
  - First, Second, First
Improving Locality by Tiled Transposition

[Submitted to EuroPar 2013]

Assume $M = M'm; N = N'n$:

1. $M \times N = M \times N' \times n$

2. Transpose 100

3. $N' \times M \times n = N' \times M' \times m \times n$

4. Transpose 0010 is equal to $N' \times M'$ instances of transpose 010

5. Transpose 0100 is equal to $N'$ instances of transpose 100

Assume $M = M'm; N = N'n$:
Tiled Transposition Exploits Locality

- 2 basic kind of transposition involved
  - Transposition 010!
    - E.g. Treat 8x2x3 array as 8 instances of 2x3 array
    - Transpose each of the 2x3 array
  - Transposition 100!
    - E.g. Treat 8x6 matrix as 8x2 array of super element of size 3
    - Transpose it to 2x8 array of super element of size 3
Transposition 010!

- Not too different from transposing a bunch of small tiles
Transposition 100!

- Not too different from transposing a matrix with super-elements

AxBxC

same as

super-elements of size C

transpose

same as

BxAxC
Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting

```
0 1 2 3 4
5 6 7 8 9
```

transpose

```
0 1
2 3
4 5
6 7
8 9
```
Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting

Cycles:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 1 \\
2 & 3 \\
4 & 5 \\
6 & 7 \\
8 & 9 \\
\end{array}
\]

\[
\text{curr} \rightarrow (\text{curr} \% \text{N}) \times \text{M} + \text{curr} / \text{N}; \quad \text{next}
\]

\[
\text{M} = 2, \text{N} = 5
\]
Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting

Cycles:
- \{0\}

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
5 & 6 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccc}
4 & 5 & 6 & 7 \\
8 & 9 & 0 & 1 \\
2 & 3 & 0 & 1 \\
4 & 5 & 6 & 7 \\
8 & 9 & 0 & 1 \\
\end{array}
\]

\[
\text{curr} = 0 \quad \rightarrow \quad \text{(curr \% N)*M + curr/N;} \quad \rightarrow \quad \text{next} \quad 0
\]

\(M = 2, N = 5\)
Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting

Cycles:
- \{0\}
- \{1, 2, 4, 8, 7, 5, 1\}

27
Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting

- Cycles:
  - \{0\}
  - \{1, 2, 4, 8, 7, 5, 1\}
  - \{3, 6, 3\}
Transposition - Cycles

- **Transposition is a permutation**
  - A permutation can be decomposed to independent cycles of shifting

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

- **Cycles:**
  - \{0\}
  - \{1, 2, 4, 8, 7, 5, 1\}
  - \{3, 6, 3\}
  - \{9\}

\[
\text{curr} \rightarrow (\text{curr} \mod N) \times M + \text{curr} / N; \quad \text{next}
\]

\[
M = 2, N = 5
\]
Cycle Following – Original

- **Cycles:**
  - thread 0: \{0\}
  - thread 1: \{1, 2, 4, 8, 7, 5, 1\}
  - thread 2: \{3, 6, 3\}
  - thread 3: \{9\}

**Imbalance**

*This is equivalent to a straightforward parallelism of the IPT algorithm in Gustavson et al, "In-place transposition of rectangular matrices." in PARA'06.*
Cycle Following – Load Balanced

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Cycle Following – Load Balanced

\{0\} \{1, 2, 4, 8, 7, 5, 1\} \{3, 6, 3\} \{9\}

\begin{array}{ccccc}
0 & 1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 & 9 \\
\end{array}
Cycle Following – Load Balanced

\{0\} \{1, 2, 4, 8, 7, 5, 1\} \{3, 6, 3\} \{9\}

\begin{array}{cccc}
0 & 1 & 2 & 3 \\
5 & 6 & 7 & 8 \\
\end{array}

\begin{array}{c}
7 \\
3 \\
4 \\
9 \\
\end{array}
Cycle Following – Load Balanced

\{0\} \quad \{1, 2, 4, 8, 7, 5, 1\} \quad \{3, 6, 3\} \quad \{9\}

\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 & 9 \\
\end{array}
Transposition 100!

- Original vs Load Balanced

[Temperature Graph Description]

- Transpose 100 on Tesla K20c
- Tile size = 32
- Bandwidth (GB/s)
- Percentage of non-trivial cycles
- Width

---

<table>
<thead>
<tr>
<th>Width</th>
<th>Bandwidth (GB/s)</th>
<th>Percentage of non-trivial cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>40</td>
<td>10%</td>
</tr>
<tr>
<td>24</td>
<td>60</td>
<td>20%</td>
</tr>
<tr>
<td>32</td>
<td>80</td>
<td>30%</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>256</td>
<td>120</td>
<td>60%</td>
</tr>
</tbody>
</table>
Some Performance Numbers

- Transposing an MxN matrix on an NVIDIA Tesla K20
  - Un-tiled transposition gives 1.5GB/s throughput
  - <2% of peak

- Tiled approach treat the MxN matrix as a 4D array, $M' \times m \times N' \times n$
  - Applying fast primitives that either
    - Transpose many instances of small matrix involving m and n only
      - ~100GB/s if tile is small
      - We want to somehow apply this only when small matrix = $m \times n$ or $n \times m$
    - Transpose one matrix of super-elements either m or n sized
      - ~70GBs, but depending on how big the super-element is
      - We want to somehow apply this only when element size = m or n
Performance of Transposition 100!

- Transposing an $N \times M' \times m$ array to $M' \times N \times m$ array on Tesla K20 (Kepler)
  - Peak = 208GB/s
  - ~33%
Performance of Transposition 100,

- Transposing an $\text{NxM'}x\text{m}$ array to $\text{M'}x\text{Nx}\text{m}$ array on AMD HD7750 (Cape Verde)
  - Peak = 72GB/s
  - $\sim 46\%$
Results: Choosing a Tile Size

- 80%+ performance can be obtained with statically-chosen tile sizes
- Search space = 1855 candidate tile sizes

Transposing a 7200x1800 Matrix of floats in-place on NVIDIA Tesla K20 (Kepler)

- 90%+ of 20.59GB/s
- 80%-90%
- 70%-80%
- 60%-70%
Results: Choosing a Tile Size

- 80%+ performance can be obtained with statically-chosen tile sizes
Results: vs. Padding

- Padding/Square Transposition involves:
  - Padding: ~38GB/s each for large padding factor
  - Square transposition: ~80GB/s
  - Packing: ~38GB/s (estimated)
  - Overall throughput: ~15GB/s

- 3-Stage approach
  - ~20GB/s with good m and n
Summary

- A full transposition algorithm library for the GPU is presented
  - First known in-place fast rectangular transposition for the GPUs
- Competitive performance even for near square matrices
Library

- We open-source the transposition library so that people don’t need to reinvent the wheel
  - OpenCL (most up-to-date), Python (PyOpenCL), CUDA
  - Mathematica version under working
- https://bitbucket.org/ijsung/libmarshal/wiki/Home

The user API is declared in
```
#include "cl_marsh.h"
```

Full In-Place Transposition on GPU
```
extern 'C' bool cl_transpose(cl_command_queue queue, cl_mem src, int A, int a, int B, int b);
```

This function implements in-place transposition of a Aa by Bb float matrix of an OpenCL buffer src. Returns false if there is no error.
Questions?
Heat-map of Cycles at (1K+M, 1K+N)
Why is This New?


Stage 1: Transpose 0100 is equal to $M'$ instances of transpose 100

Stage 2: Transpose 0010 is equal to $M' \times N'$ instances of transpose 010

Stage 3: Transpose 1000

Stage 4: Transpose 0100 is equal to $N'$ instances of transpose 100

$M \times N = M' \times m \times N' \times n$

$n = 3$

$m = 2$

$M' = 4$

$N' = 2$

$M' \times N' \times m \times n$

$M' \times N' \times n \times m$
Comparison of Methods, Using a 7200x180 Example

  - Tile sizes allowed in each step:
    - For 7200x1800 matrix, best (m, n) = (16, 20) on Tesla K20:
      - Transposition 0100 (derived from 100!): n (15.81 GB/s)
      - Transposition 0010 (derived from 010!): m x n (98.58 GB/s)
      - Transposition 1000 (derived from 100!): n x m (49.21 GB/s)
      - Transposition 0100 (derived from 100!): m (21.40 GB/s)
  - We proposed a 3-step approach
    - Larger tile sizes allowed in transposition 100 and 0100:
      - For 7200x1800 matrix, 20.59 GB/s for (m, n) = (32, 72) on Tesla K20:
        - Transposition 100: m (59.55 GB/s)
        - Transposition 0010: mxn (113.90 GB/s)
        - Transposition 0100 (derived from 100!): n (43.89 GB/s)
Results: vs. Prior Art
Low-level Optimizations

- **Transposition**: PTTWAC vs P-IPT (one-cycle-per-thread-block)
  - Bottleneck of PTTWAC was on barriers
  - Removed by using a warp to move a super-element instead of a thread-block
Low-level Optimizations Applied

- Transposition 010\texttt{: PTTWAC vs P-IPT (one-cycle-per-thread)}
  - Bottleneck of PTTWAC was on atomic operations to on-chip memory
Comparing to P-IPT (Cycle Following)

For transposition 010, re-layout atomic bit flags reduces contention and improves performance

(a) Spreading_factor = 1

(b) Spreading_factor = 32

(c) Spreading_factor = 32 & Padding

Element_position

Flag_word

Position conflict

Bank conflict: 16 mod 32 = 48 mod 32

Bank conflict: 512 mod 1024 = 1536 mod 1024

Bank conflict
Comparing to P-IPT (Cycle Following)

- Transposition 010, w/ layout adjustment for the atomic flags in on-chip memory