Insight

- Advantages of tiling:
  - Increased locality
  - Improves parallelism

- But, most programming languages lack language constructs for tiling
Application Areas

- **Multi-level tiling:**
  - Cache-oblivious / recursive algorithms
    - Numerical/linear algebra
    - Sorting
    - Scanning
  - Stencil codes
    - ODEs and PDEs

- **Single-level tiling**
  - Wide range of applications
Impact of Work

- Good number of citations ~150
- But, idea didn’t really catch on
  - Not much work on multiple level tiling since 2010
- Work has been much more focused on single-tiling:
  - Automatic tiling
  - Optimizing & Dynamic Tiling
  - Tiling for GPUs & Distributed Systems
  - Overlapping tiling
  - Tiling across the memory hierarchy
  - Edge cases
What is an HTA?

- An array partitioned into tiles.
  - Tiles are either conventional arrays or lower level HTAs
  - Can have any number of dimensions
- Tiles can be distributed across processors or stored locally

Local HTA

\[ \text{hta}( \text{M}, \{ [1,3,5], [1,3,5] \} ) \]

(Ref: Construction of an HTA [1, p.49])

Distributed HTA

\[ \text{hta}( \text{M}, \{ [1,3,5], [1,3,5] \}, [2,2] , \text{“cyclic”} ) \]

(1) Construction of an HTA [1, p. 49]
Distributed Programming Model

- Follows SPMD model
  - Communication: 2-sided message passing (MPI)
  - Computation: each processor applies on locally owned tiles

\[ C = \begin{array}{ccc}
P1 & P2 & P1 \\
P3 & P4 & P3 \\
P1 & P2 & P1 \\
\end{array} \times 2 \]
Distribution Types

Processor Grid (1x2)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>

HTA (3x8)

Block

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P1</th>
<th>P1</th>
<th>P1</th>
<th>P2</th>
<th>P2</th>
<th>P2</th>
<th>P2</th>
</tr>
</thead>
</table>

Cyclic

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P1</th>
<th>P2</th>
<th>P1</th>
<th>P2</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>

Block-Cyclic

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P1</th>
<th>P2</th>
<th>P2</th>
<th>P1</th>
<th>P1</th>
<th>P2</th>
<th>P2</th>
</tr>
</thead>
</table>
Distribution Types (Continued)

Processor Grid

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
<td>P4</td>
</tr>
</tbody>
</table>

HTA (4x4)

Cyclic

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
<td>P4</td>
<td>P3</td>
<td>P4</td>
</tr>
<tr>
<td>P1</td>
<td>P2</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
<td>P4</td>
<td>P3</td>
<td>P4</td>
</tr>
</tbody>
</table>

Block

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P1</td>
<td>P2</td>
<td>P2</td>
</tr>
<tr>
<td>P1</td>
<td>P1</td>
<td>P2</td>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
<td>P3</td>
<td>P4</td>
<td>P4</td>
</tr>
<tr>
<td>P3</td>
<td>P3</td>
<td>P4</td>
<td>P4</td>
</tr>
</tbody>
</table>
Machine Mapping

- Not addressed in this paper

- HTAs have a machine mapping that specifies:
  - where the HTA is allocated in a distributed system
    - *Distribution* class: specifies the home location of the scalar data for each of the tiles of an HTA
  - the memory layout of the scalar data array underlying the HTA
    - *MemoryMapping* class: specifies the layout (row-major across tiles, row-major per tile etc.), size and stride of the flat array data underlying the HTA
Accessing HTAs

3 methods
- Hierarchically – addressing using each level of tile
- Flat – addresses the elements of an HTA by their absolute indices, as a normal array
- A combination of the two – applying flattening at any level of the hierarchy

Takeaway – provides a simple method for selecting elements of HTAs that bridges the gap between HTA and non-HTA applications

(1) Accessing the contents of an HTA [1, p.49]
Accessing Regions of HTAs

- 3 methods
  - Can use `begin:end` indexing for any access method
  - Can use `:` notation to refer to the whole range of values for an index
  - Can use Boolean arrays for logical indexing

- Takeaway: access methods cover all use cases

(1) Accessing the contents of an HTA [1, p. 49]
(2) Logical indexing in HTA [1, p. 50]
Rules for Binary Operations & Assignment

- HTA $\oplus$ Scalar
  - each scalar of the HTA is operated with the scalar

- HTA $\oplus$ Matrix
  - each lowest level tile of the HTA is operated with the matrix

- HTA $\oplus$ HTA
  - Same topology -> corresponding tiles are operated on
    - Produces an HTA with the same topology
  - Otherwise, the operation acts like HTA $\oplus$ Matrix
HTA Methods

- Overloaded array operations that, when applied to HTAs, operate on the tile level (instead of individual array elements)
  - Assignment
  - Binary operations
  - Indexing
  - Other frequently-used array functions
    - transpose
    - permute
    - circshift
    - repmat
- Methods that apply only to HTAs
  - reduceHTA - a generalized reduction method that operates on HTA tiles
  - parHTA - applies in parallel the same function to each tile of an HTA
Example – transpose and permute

- dpermute – the data permuted, but the shape of the containing HTA remains the same (# of tiles in each dimension)

(1) Transpose and dpermute [1, p. 50]
Example – 3D matrix & dpermute

- Implicit parallel communication from HTA assignment
- fft is applied in parallel on local tiles
- 1st and 2nd dimensions are local, so use dpermute to make the 3rd dimension local to the processor for fft can be applied

X = hta(A, { [1], [1], [partition-z] }, [1,1,nprocs])
X = fft(X, [], 1)
X = fft(X, [], 2)
X = dpermute(X, [3,1,2])
X = fft(X, [], 1)

(1) Data Permutation in FFT. (a)-Pictorial view. (b)-code [1, p. 50]
Example – circshift

function C = cannon(A,B,C)
    for i=2:m
        A{i,:) = circshift(A{i,:}, [0, -(i-1)]);
        B(:,i) = circshift(B(:,i), [-(i-1), 0]);
    end
    for k=1:m-1
        C = C + A * B;
        A = circshift(A, [0, -1]);
        B = circshift(B, [-1, 0]);
    end

- Cannon’s Algorithm – MMM
- Shifts tiles in row i of A to the left i-1 times
- Shifts tiles in column i of B up i-1 times
- Matrix multiplication is done locally -> C is left distributed
Advantages of Tiled Cannon’s and parHTA Example

- Aggregates data into a tile for communication
- Increased locality from matrix-matrix multiplication (instead of element by element multiplication)
- Can further increase cache locality by using HTAs with more levels, and applying matrix multiplication recursively
  - \[ C = \text{parHTA}(\text{matmul}, A, B, C) \]

```matlab
function C = matmul (A, B, C)
    if (level(A) == 0)
        C = C + A * B;
    else
        for i=1:size(A,1)
            for k=1:size(A,2)
                for j=1:size(B,2)
                    C{i, j} = matmul(A{i,k}, B{k,j}, C{i,j});
```
Example - repmat

- Normal Summa algorithm

  For (k = 1 ... M)
    For (i = 1 ... M)
      For (j = 1 ... M)
        \[ C(i, j) = C(i, j) + a(i, k) \times b(k, j) \]

- Tiled version:

  ```matlab
  function C = summa(A, B, C)
    for k=1:m
      T1 = repmat(A(:, k), 1, m);
      T2 = repmat(B{k, :}, m, 1);
      C = C + T1 \times T2;
    end
  ```
Example – repmat (continued)

- **Tiled version:**

  ```matlab
  function C = summa (A, B, C)
  for k=1:m
      T1 = repmat(A{:, k}, 1, m);
      T2 = repmat(B{k, :], m, 1);
      C = C + T1 * T2;
  end
  ```

- **Multiplication is then done locally**
Example – logical indexing

- Wavefront computation - Normal code:

```
for i=2:m-1
    for j=2:n-1
        A(i,j)= A(i-1,j) + A(i,j-1);
```

- Can parallelize by computing in parallel the element of each diagonal of the matrix:

```
(1) 2-D wavefront computation [1, p. 52]
```
Example – logical indexing

- Wavefront computation
  
  \[
  \begin{array}{c}
  \text{for } k=2:m+n \\
  \text{for } i=2:dimx-1 \\
  \text{for } j=2:dimy-1 \\
  \quad A\{x+y == k\}(i, j) = A\{x+y == k\}(i-1, j) + \\
  \quad \quad A\{x+y == k\}(i, j-1);
  \end{array}
  \]

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4
\end{bmatrix}
\]

- Select tiles on the diagonal using “x+y == k”

- Implicit communication

\[1\) 2-D wavefront computation [1, p. 52]\]
parHTA and reduceHTA example

\[
A = \text{hta}(\text{MX, \{partition\_A\}, [m n]}); \\
V = \text{hta}(\text{VX, \{partition\_B\}, [m n]}); \\
B = \text{repmat}(V, m, 1) \\
B = \text{parHTA(@transpose, B)} \\
C = \text{reduceHTA('sum', A \ast B, 2, true});
\]

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

\[
\begin{array}{ccc}
a & b & c \\
a & b & c \\
a & b & c \\
\end{array}
\]

\[
\begin{array}{ccc}
a & a & a \\
b & b & b \\
c & c & c \\
\end{array}
\]

\[
\begin{array}{ccc}
1*a & 2*a & 3*a \\
4*b & 5*b & 6*b \\
7*c & 8*c & 9*c \\
\end{array}
\]

\[
\begin{array}{ccc}
1*a+2*a+3*a \\
4*b+5*b+6*b \\
7*c+8*c+9*c \\
\end{array}
\]

all-to-all

\[
\begin{array}{ccc}
d & d & d \\
e & e & e \\
f & f & f \\
\end{array}
\]
Implementations

- Implemented as a class for:
  - MATLAB
  - C++
- Uses object oriented capabilities of these languages
Goals

- To identify the set of operations on tiles needed to develop parallel programs
  - Examples seem fairly comprehensive
- To implement these operations so that they are:
  - Readable / easy to use
  - Efficient
Results – Performance of MATLAB Implementation

- NAS Performance Benchmarks

<table>
<thead>
<tr>
<th>Nprocs</th>
<th>EP (CLASS C)</th>
<th>FT (CLASS B)</th>
<th>CG (CLASS C)</th>
<th>MG (CLASS B)</th>
<th>LU (CLASS B)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fortran + MPI</td>
<td>Matlab + HTA</td>
<td>Fortran + MPI</td>
<td>Matlab + HTA</td>
<td>Fortran + MPI</td>
</tr>
<tr>
<td>1</td>
<td>901.6</td>
<td>3556.9</td>
<td>136.8</td>
<td>657.4</td>
<td>3606.9</td>
</tr>
<tr>
<td>4</td>
<td>273.1</td>
<td>888.8</td>
<td>109.1</td>
<td>274.0</td>
<td>362.0</td>
</tr>
<tr>
<td>8</td>
<td>136.3</td>
<td>447.0</td>
<td>65.5</td>
<td>159.3</td>
<td>123.4</td>
</tr>
<tr>
<td>16</td>
<td>68.6</td>
<td>224.8</td>
<td>37.2</td>
<td>87.2</td>
<td>89.5</td>
</tr>
<tr>
<td>32</td>
<td>34.7</td>
<td>112.0</td>
<td>20.7</td>
<td>42.9</td>
<td>48.4</td>
</tr>
<tr>
<td>64</td>
<td>17.1</td>
<td>56.7</td>
<td>10.4</td>
<td>24.0</td>
<td>44.5</td>
</tr>
<tr>
<td>128</td>
<td>8.5</td>
<td>29.1</td>
<td>5.9</td>
<td>15.6</td>
<td>30.8</td>
</tr>
</tbody>
</table>

(1) Execution times in seconds for some of the applications in the NAS benchmarks [1, p. 53]

- Takeaways:
  - For each configuration, MATLAB+HTA is slower
  - EP, FT, CG: 128 parallel MATLAB performs 30.9, 8.8, and 29.3X faster than sequential Fortran
  - MG, LU, BT(not shown): slow sequential, 128 parallel MATLAB performs close to the same or worse than sequential Fortran
C++ Implementation

- Largely the same
- Differences
  - Focused on performance
  - Array operators not overloaded (fixed later)
C++ Example

**Takeaway:** C++ implementation, with type declarations and without array operators, is much less readable

(1) Recursive matrix multiplication in C++ using HTAs [1, p. 54]
Results - Performance of C++ Implementation

- One level of HTA introduces an overhead of between 8 – 13.5%
- Naïve use produces little benefit
- INTEL MKL is the only version that uses INTEL SSE2 vector extensions, all others use scalar code

Performance in MFLOPS for different versions of matrix-matrix multiplication. (1) MKL uses SSE2 vector extension. (1)

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Naïve 3 loops</th>
<th>Tiled 6 loops</th>
<th>HTA naïve</th>
<th>HTA+ATLAS</th>
<th>ATLAS</th>
<th>Intel MKL(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>504</td>
<td>161</td>
<td>657</td>
<td>675</td>
<td>2069</td>
<td>2387</td>
<td>3624</td>
</tr>
<tr>
<td>1008</td>
<td>150</td>
<td>649</td>
<td>679</td>
<td>2192</td>
<td>2384</td>
<td>3762</td>
</tr>
<tr>
<td>2016</td>
<td>133</td>
<td>632</td>
<td>675</td>
<td>2216</td>
<td>2492</td>
<td>3821</td>
</tr>
<tr>
<td>3024</td>
<td>135</td>
<td>644</td>
<td>668</td>
<td>2245</td>
<td>2509</td>
<td>3716</td>
</tr>
<tr>
<td>4032</td>
<td>36</td>
<td>588</td>
<td>613</td>
<td>2217</td>
<td>2519</td>
<td>3752</td>
</tr>
</tbody>
</table>

(1) Performance in MFLOPS for different versions of matrix-matrix multiplication [1, p. 55]
Results – Program Complexity

- Reduces complexity (measured in terms of lines of code)
  - Communication: assignments vs. message passing
  - Data decomposition: single HTA constructor vs. computing assignment

(1) Linecount of key sections of HTA and MPI programs [1, p. 55]
Advantages of HTAs

- Library approach
  - Easy to use, can make small modifications to existing code

- Global data view & single threaded view
  - Reduced program complexity

- Hierarchical tiling
  - Recursion
  - Can use multiple levels of parallelism
Disadvantages of HTAs

- **Programming burden:**
  - Creating distributed, tiled algorithms is challenging
  - Insufficient documentation

- **Difficulty optimizing code:**
  - Interprocess communication is hidden from the user
  - SPMD Model limits ability to use irregular parallelism
  - Library overhead
Future Work

- C++ implementation
  - Operator overloading
  - Optimizations
  - Additional functionality
    - Asynchronous communication
    - Map-reduce framework
    - Overlapped tiling
    - Data layering
References