Optimization of Tele-Immersion Codes

Albert Sidelnik, I-Jui Sung, Wanmin Wu,
María Garzarán, Wen-mei Hwu, Klara Nahrstedt,
David Padua, Sanjay Patel

University of Illinois at Urbana-Champaign
Agenda

1. High-level goals
2. Tele-Immersion
3. GPU specific optimizations applied
4. Results of the optimization effort
5. Future work
6. Conclusion
Main Goals

• Find data-parallel primitives and apply tuning techniques
  – Adapts for **portability** across multiple target architectures
    • E.g. Multi-cores, Clusters, and GPUs
  – Adapts for **performance**
    • E.g. optimal tile sizes, unroll factors, scheduling
  – Enables **productivity**
    • Programmer express data parallel operations
    • Focus more on their algorithms

• To do this study, we need good representative applications
  – Apply above to the domain of **Tele-immersion**
Tele-Immersion

Photo courtesy of Prof. Ruzena Bajcsy.
Tele-Immersive Environment

3D Capturing Tier

End User

Internet2

Transmission Tier

Multi-Display Rendering/Displaying Tier

UPCRC Illinois
Universal Parallel Computing Research Center
Initial Strategy

• Profile existing code to find hotspots
• Restructure original code as a sequence of data parallel operations
• Express these operations using new data structures
  – This enables targeting of multiple platforms
• Perform tuning on these newly restructured kernels
# Overall Flow of TI Code

<table>
<thead>
<tr>
<th>Main Thread</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Post-processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Image Thread 0 (BW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 1 (BW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 2 (BW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 3 (Color)</td>
<td></td>
<td>Pre-processing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute Thread 0</td>
<td>Triangulation</td>
<td></td>
<td>MNCC</td>
<td>Homogen</td>
<td>Reconstruct Depth</td>
</tr>
<tr>
<td>Compute Thread 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute Thread N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>12.1</td>
<td>12.0</td>
<td>5.5</td>
<td>17.8</td>
<td>2</td>
</tr>
</tbody>
</table>
Compute MNCC

• MNCC = Modified Normalized Cross Correlation
  – Computes correlation of feature points across different images

• Consists of two (consecutive) data parallel operations
  – Computation of correlation values
  – Maximum reduction

• Very little control flow (outside of maximum reduction)
  – Good candidate for GPUs
High-Level View of MNCC

Original Code

```c
compute_mncc (data , Thread ID) {
    int start = start edge for ID
    int end = end edge for ID
    for i=start , end {
        x1= x_edge [i];
        y1= y_edge [i];
        for j=0, num_disp {
            // find corresponding edges in L and R cameras
            x1_eL = ( float *)( C2LX + x1* num_disp );
            y1_eL = ( float *)( C2LY + y1* num_disp );
            ...
        }
        maxcorr(i) =0;
        for j=0, NUM_DISP {
            ...
            corr1 = ...; corr2 = ....; corr3 = ....;
            // find maximum correlation
            corr [i* num_disp +j]= corr1 + corr2 + corr3 ;
            if ( corr [i* num_disp +j]> maxcorr [i]) then
                maxcorr [i] = corr [i* num_disp +j];
        }
    }
}
```

Restructured Code

```c
compute_mncc (data , Thread ID) {
    int start = start edge for ID
    int end = end edge for ID
    for i=start , end {
        for j=0, NUM_DISP {
            x1= x_edge [i];
            y1= y_edge [i];
            // find corresponding edges in L and R cameras
            x1_eL = ( float *)& C2LX [x1* num_disp ];
            y1_eL = ( float *)& C2LY [y1* num_disp ];
            ...
        }
        corr1 = ...; corr2 = ...; corr3 = ...;
        corr [i* num_disp +j]= corr1 + corr2 + corr3 ;
    }
}

find_maximum (data , Thread ID) {
    int start = start edge for ID
    int end = end edge for ID
    for i = start , end {
        maxcorr [i] =0;
        for j = 0, NUM_DISP {
            if ( corr [i* num_disp +j]> maxcorr [i]) then
                maxcorr [i] = corr [i* num_disp +j];
        }
    }
}
```
MNCC Optimizations (GPU)

1. Start with naïve (restructured) data parallel operation
   – Easy port of the code to use CUDA
   – Only outer loop is parallelized
   – Empirically search for best thread block size

2. Introduce multiple dimensions of parallelism
   – No dependences across loops
   – Empirically search for best 2D thread block size

3. Transpose the thread block structure (Loop Interchange)
   – Take advantage of memory coalescing
   – Empirically search the best transposed 2D thread block size

4. Utilize texture memory as a hardware cache
   – Frequent 2D table lookups
Compute Homogen

- Data Parallel routine
- Apply similar restructuring techniques as in MNCC
- Lots of control flow
  - Consists of many divergent branches
    - Very input dependent
  - Potentially bad candidate for GPU
  - Good for CPUs using dynamic scheduling
    - Load imbalance
    - Overdecomposition will help here
Homogen Optimizations (GPU)

1. Start with naïve data parallel implementation
   - Same as MNCC

2. Utilize texture memory
   - Same as MNCC

3. Compiler flags
   - Nvcc compiler flag \texttt{--maxregcount #}
   - Beneficial impact on performance by forcing compiler to spill registers earlier
Initial Results

• Test Platform 1:
  – Intel 4-Core Penryn 2.83ghz
  – 4GB memory/6MB L2 cache
  – Nvidia GTX280 (Cuda 2.0)
  – Intel ICC 10.1 Compiler/MS Visual C++

• Test Platform 2:
  – 4x6-Core Intel Dunnington Xeon 2.40ghz
  – 48GB memory/12MB L3 cache
  – Intel ICC 10.1
Compiler Results (4-Core Intel)

• Original Code
  – Microsoft Visual C++: 20fps
  – Intel ICC 10.1: 31fps

• Up to 35% speedup just from switching compilers
  – Mostly due to auto-vectorization
MNCC GPU Optimization Trend

- Naïve
- Multiple Dimensions
- Loop Interchange
- Texture Memory
MNCC Results (CPU)

24-Core Intel

Time (ms)

8 Threads

16 Threads

24 Threads

Original

Restructured

UPCRC Illinois
Universal Parallel Computing Research Center
Optimized MNCC Results

24-Core Intel vs Nvidia GTX280 GPU

Time (ms)

2.5
2
1.5
1
0.5
0

24-Core

GPU
Homogen GPU Optimization Trend
Homogen Results (CPU)

24-Core Intel

Number of Tiles

Time (ms)

8 24 64 128 256 512 1024

Original  Restructured
Optimized Homogen Results

24-Core Intel vs Nvidia GTX280 GPU

- Time (ms)
  - 0
  - 2
  - 4
  - 6
  - 8
  - 10
  - 12

- 24 Core CPU
- GPU

UPCCO Illinois
Universal Parallel Computing Research Center
## Overall Results (Modified)

<table>
<thead>
<tr>
<th>Main Thread</th>
<th>Pre-processing</th>
<th>Triangulation</th>
<th>Post-processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Image Thread 0 (BW)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 1 (BW)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 2 (BW)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 3 (Color)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute Thread 0</td>
<td></td>
<td>MNCC</td>
<td></td>
</tr>
<tr>
<td>Compute Thread 1</td>
<td></td>
<td>Homogen</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ReconstructDepth</td>
<td></td>
</tr>
<tr>
<td>Compute Thread N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>12.1</td>
<td>1.03</td>
<td>6.4</td>
</tr>
<tr>
<td></td>
<td>1.12</td>
<td>0.5</td>
<td>1.8</td>
</tr>
<tr>
<td>Total:</td>
<td>22.95 (~44fps)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![UPCORE Illinois](image-url)
Work in progress

• Port kernels to use new data structures (HTAs)
  – HTA = Hierarchically Tiled Array
  – Facilitates locality and parallelism
  – Provides a “map” primitive
    • Performs a user-defined operation on an element-by-element or tile-by-tile granularity
  – Encapsulate parallelism from programmer
  – Target for multiple classes of parallel architectures
    • E.g. multi-cores, clusters, GPUs

• Add GPU backend to HTAs
Work in progress (cont.)

• Investigation of parallelization of Delaunay triangulation
  – K. Pingali, et. al (Galois)
• Further GPU tuning of Homogen in progress
• Adding empirical auto-tuning framework
  – Tune for performance on multi-cores and GPUs
• Look at future architectures such as Intel’s Larabee
Conclusions

• Good performance from restructuring and tuning the kernels
• Switching compilers leads to large performance improvements
• Good scalability
  – For both large multi-cores and GPU platforms
  – GPU implementation of MNCC is up to 2x faster than a 24-core
• New bottlenecks appear after original optimizations
Questions?
Thread Block Size Impact on MNCC

![Graph showing the impact of thread block size on MNCC time (µs). The graph compares 2D Thread Block, Loop Interchange, and Texture Memory.]
Compute Kernels

- **MNCC** and **Homogen** are the two most computationally expensive sections of code (~68% total execution)
  - MNCC \( \rightarrow \) ~34% of total execution time
  - Compute Homogen \( \rightarrow \) ~34% of total execution time

- Delaunay Triangulation is purely sequential
  - Parallel implementations exist (K. Pingali et. al)
  - Becomes bottleneck as MNCC is improved
User Defined Operations

\texttt{hmap( }F(), X, Y)\texttt{)
Compute MNCC (cont.)

• We need to restructure original MNCC code
  – Allows for Hmap on element-by-element, or tile-by-tile
    • This can exploit more parallelism
  – Kernels are now simpler and easier to understand
  – Simpler code can possibly enable more compiler optimizations

• Perform traditional compiler optimizations on the kernels
  – Converting code to perfectly nested loops
  – Changing pointer arithmetic to array subscripts
    • Benefits readability, but might worsen performance
  – Loop fusion
  – Code movement
  – Dead code elimination
Compute MNCC Restructuring

**Original MNCC**

```c
compute_mncc(data, Thread ID) {
    int start = start of range for ID
    int end = end of range for ID
    for I = start, end
        ... 
        for J = 0, NUM_DISP {
            ... 
        }
    for J = 0, NUM_DISP {
        ... 
        corr_vals(I * NUM_DISP + J) = ...
    }
    find maximum value and index
}
```

**Restructured MNCC**

```c
compute_mncc(data, Thread ID) {
    int start = start of range for ID
    int end = end of range for ID
    for I = start, end
        for J = 0, NUM_DISP {
            ... 
            corr_vals(I * NUM_DISP + J) = ...
        }
    find_maximum(data, Thread ID) {
        int start = start of range for ID
        int end = end of range for ID
        for I = start, end
            for J = 0, NUM_DISP {
                ...
            }
        find maximum value and index
    }
```
Hmap conversion

Call hmap(ComputeMNCC(), DATA)

Call hmapReduce(MAX(), DATA)
## Overall Results (Original)

<table>
<thead>
<tr>
<th>Main Thread</th>
<th>Post-processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Image Thread 0 (BW)</td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 1 (BW)</td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 2 (BW)</td>
<td></td>
</tr>
<tr>
<td>Get Image Thread 3 (Color)</td>
<td></td>
</tr>
<tr>
<td>Compute Thread 0</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Compute Thread 1</td>
<td>Homogen</td>
</tr>
<tr>
<td>...</td>
<td>MNCC</td>
</tr>
<tr>
<td>Compute Thread 7</td>
<td>Reconstruct Depth</td>
</tr>
</tbody>
</table>

| Time (ms): | 12.1 | 12.0 | 5.5 | 17.8 | 2 | 1.8 | Total: 51.8 (~19.3fps) |
HTA Data Structure