

# Optimization of Tele-Immersion Codes

**Albert Sidelnik**

**I-Jui Sung**

In collaboration with:

Wanmin Wu, María Garzarán, Wen-mei Hwu,  
Klara Nahrstedt, David Padua, Sanjay Patel  
University of Illinois at Urbana-Champaign

**UPERC Illinois**  
Universal Parallel Computing  
Research Center



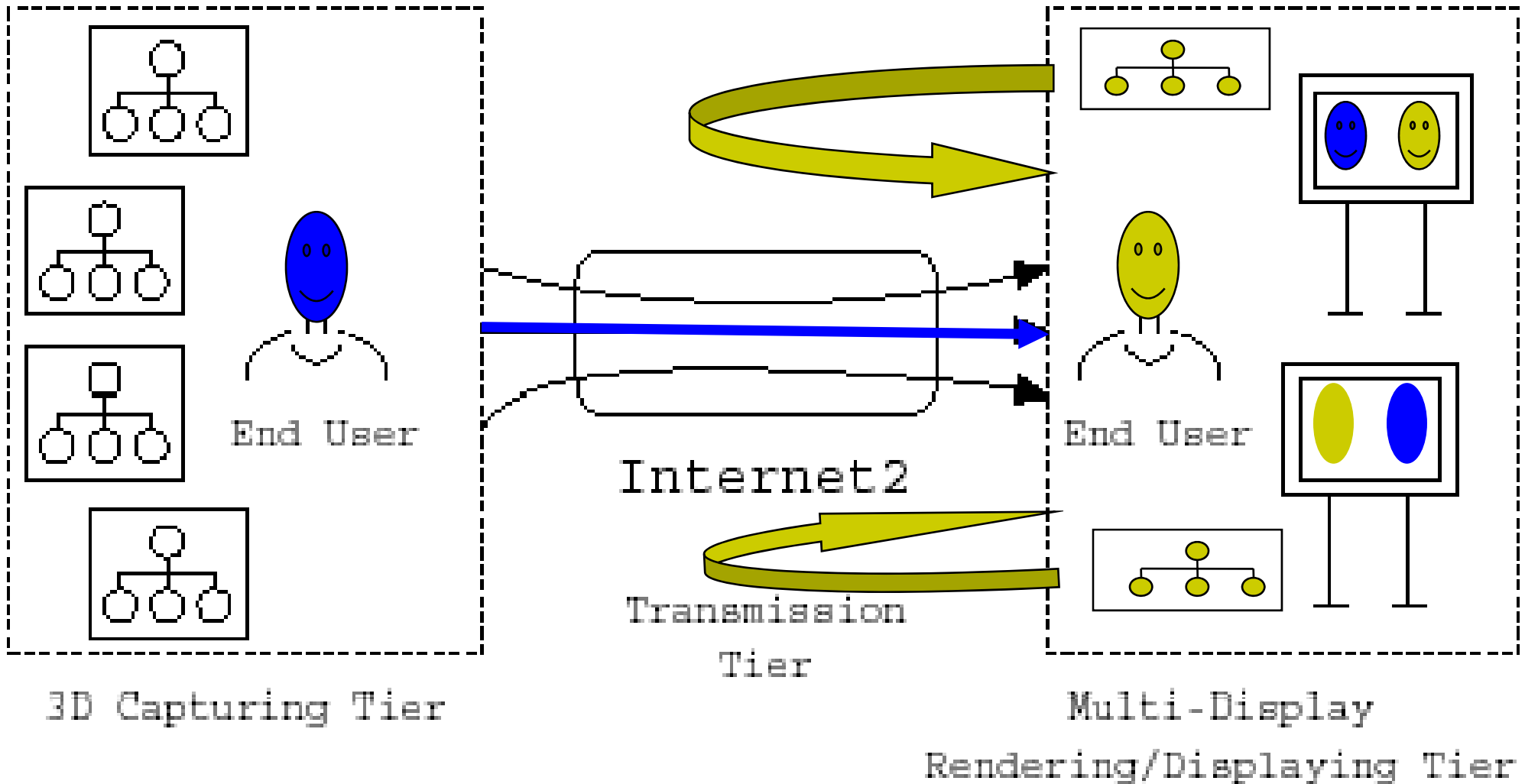
# Tele-Immersion



Photo courtesy of Prof. Ruzena Bajcsy.



# Tele-Immersive Environment



# Main Goals

- Find data-parallel primitives and apply auto-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 a good application
  - Apply above to the domain of **Tele-immersion**



# 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 auto-tuning on these newly restructured kernels

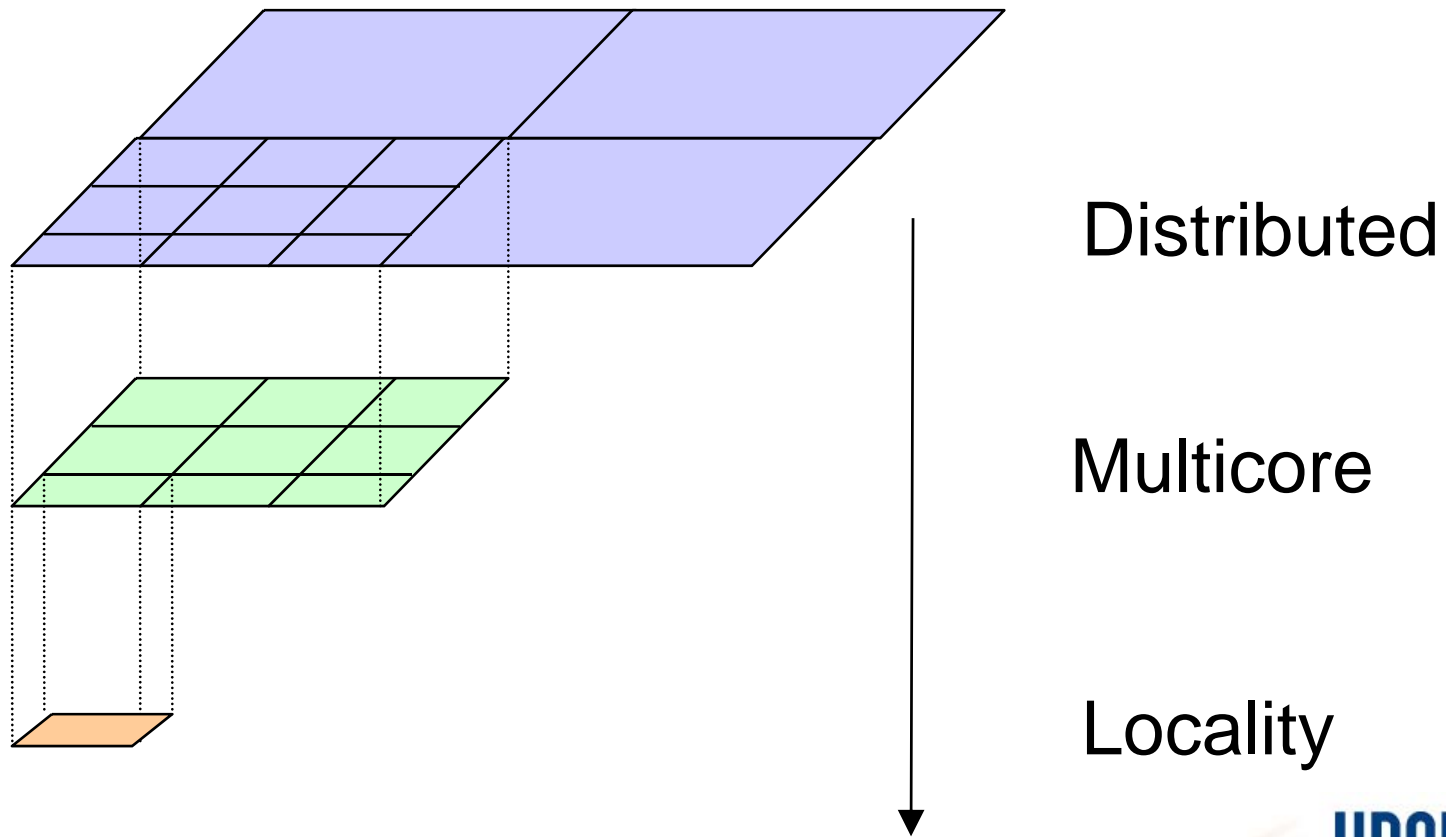


# HTA Data Structure

- 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
- HTAs encapsulate parallelism from programmer
- Can target for multiple classes of parallel architectures
  - E.g. multi-cores, clusters, GPUs



# HTA Data Structure



# Overall Flow of TI Code

Main Thread						Post-processing		
Get Image Thread 0 (BW)	Pre-processing							
Get Image Thread 1 (BW)								
Get Image Thread 2 (BW)								
Get Image Thread 3 (Color)								
Compute Thread 0		Triangulation		Homogen	Reconstruct Depth			
Compute Thread 1		MNCC						
...								
Compute Thread 7								
Time (ms) :	12.1	12.0	5.5	17.8	2	1.8	Total: 51.2	



# Compute Kernels

- MNCC and Homogen are the two most computationally expensive sections of code (~68% total execution)
  - MNCC → ~34% of total execution time
  - Compute Homogen → ~34% of total execution time
- Delaunay Triangulation is purely sequential
  - Parallel implementations exist (K. Pingali et. al)
  - Becomes bottleneck as MNCC is improved



# Compute MNCC

- MNCC = Modified Normalized Cross Correlation
  - Computes correlation of feature points across different images
- Very little control flow
  - Good candidate for GPUs
- Consists of two (consecutive) data parallel operations
  - Computation of correlation values
  - Maximum reduction



# Compute Homogen

- Data Parallel routine
- Apply similar restructuring techniques as in MNCC
- Lots of control flow
  - Potentially bad candidate for GPU
  - Good for CPU
    - Load imbalance
    - Overdecomposition will help here



# 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



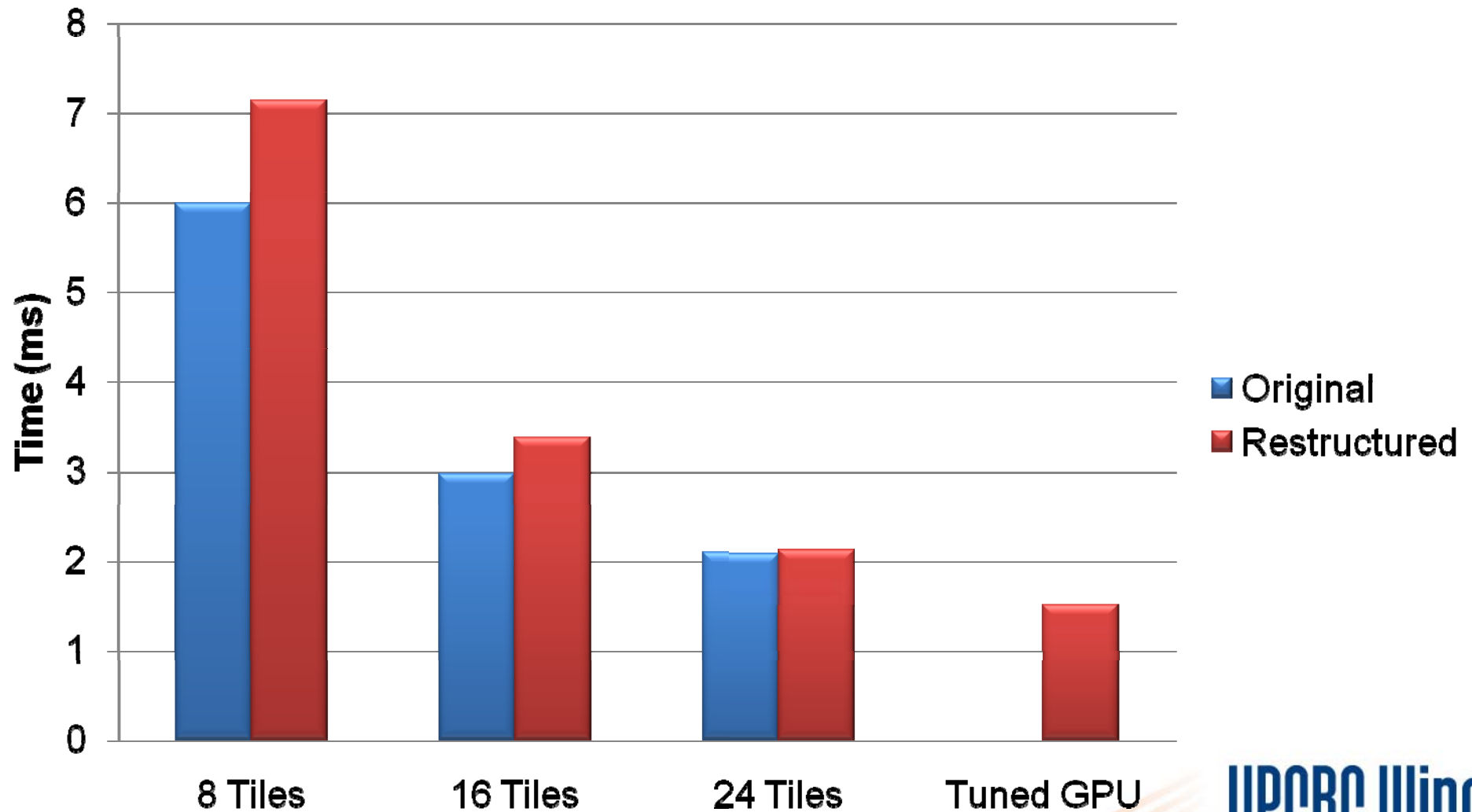
# Compiler Results (4-Core Intel)

- Original Code
  - Microsoft Visual C++: 20fps
  - Intel ICC 10.1: **31fps**
- Up to 35% speedup just from switching compilers
  - Mostly due to auto-vectorization



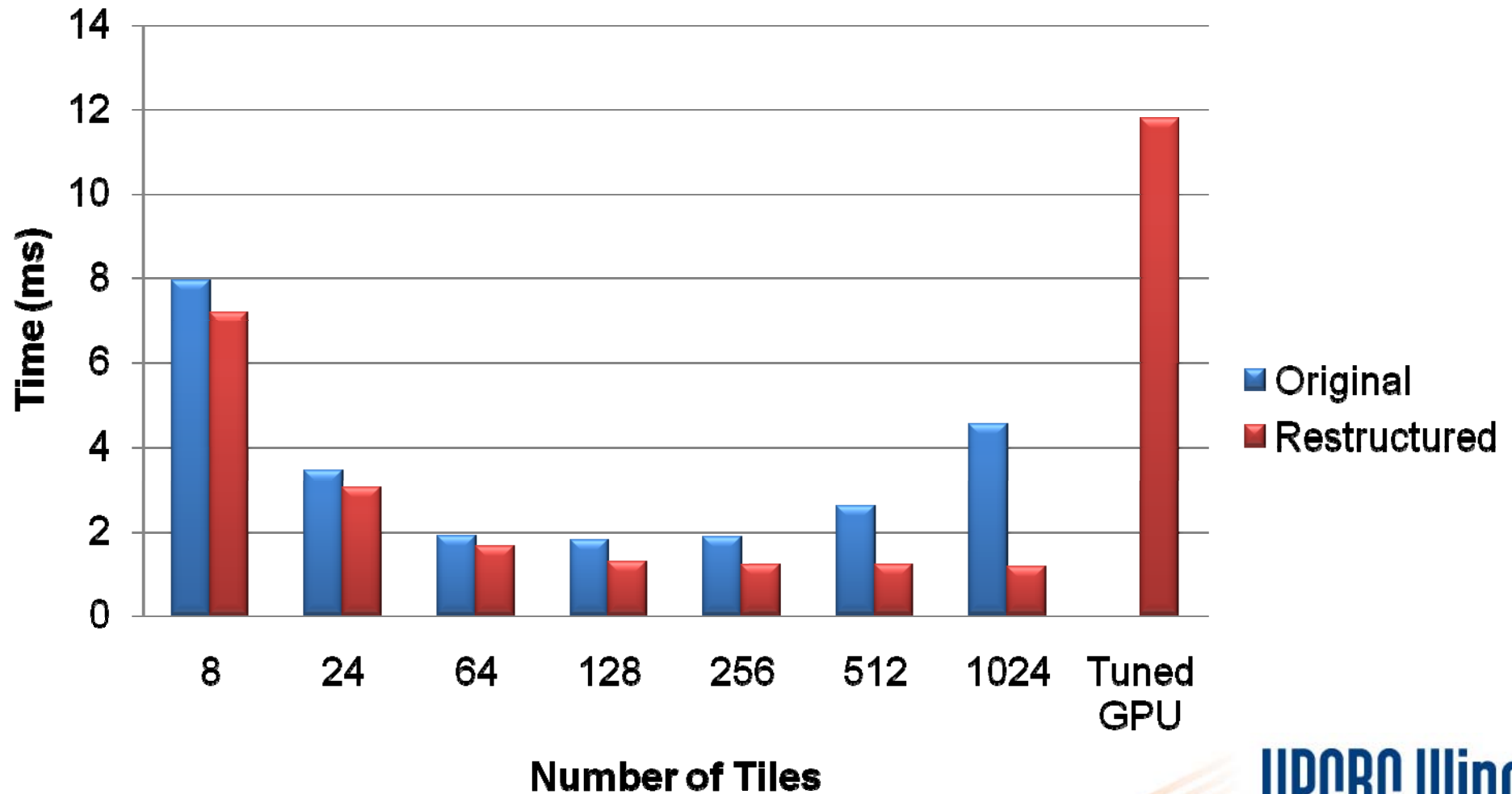
# Optimized MNCC Results

## 24-Core Intel



# Optimized Homogen Results

## 24-Core Intel



# Overall Results (Modified)

Main Thread						Post-processing	
Get Image Thread 0 (BW)	Pre-processing						
Get Image Thread 1 (BW)							
Get Image Thread 2 (BW)							
Get Image Thread 3 (Color)							
Compute Thread 0		Triangulation		Homogen	Reconstruct Depth		
Compute Thread 1		MNCC					
...							
Compute Thread N							
Time (ms) :	12.1	2.13	5.3	1.12	0.5	1.8	Total: 22.95 (~44fps)



# Conclusions

- Good performance from restructuring and tuning the kernels
- Switching compilers leads to large performance improvements
- Good scalable speedups
  - For both large multi-cores and GPU platforms
  - GPU implementation of MNCC is ~30% faster than a 24-core
- New bottlenecks appear after original optimizations



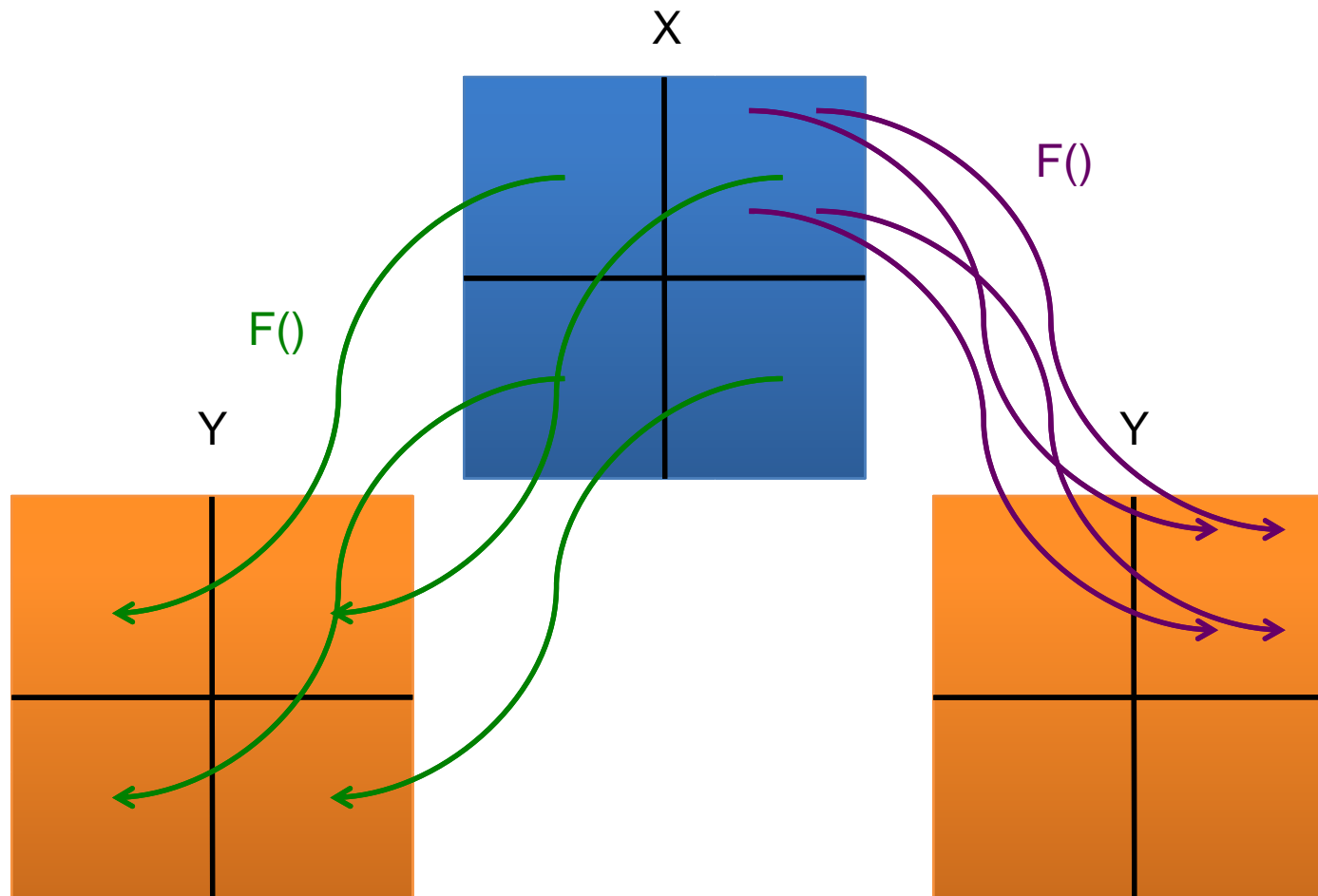
- Work to be presented at:
  - A. Sidelnik, I. Sung, W. Wu, M. Garzaran, W. Hwu, K. Nahrstedt, D. Padua, S. Patel, “**Optimization of Tele-Immersion Codes**”, Second Workshop on General Purpose Processing on Graphics Processing Units, Washington D.C, March 2009
- For more questions:
  - [asideln2@uiuc.edu](mailto:asideln2@uiuc.edu) or [sung10@uiuc.edu](mailto:sung10@uiuc.edu)

# Backup Slides



# User Defined Operations

`hmap(F(), X, Y)`



# 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

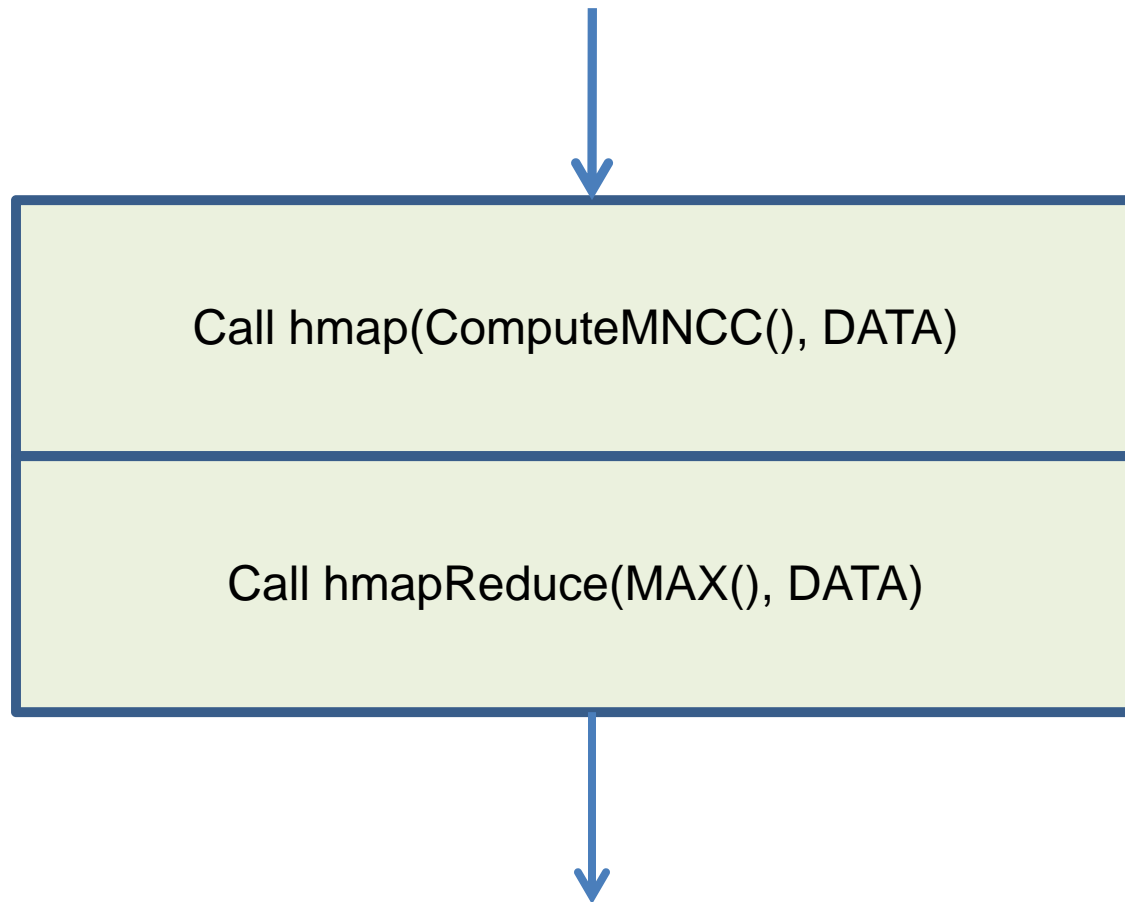
## Original MNCC

```
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

```
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



# Overall Results (Original)

Main Thread						Post-processing	
Get Image Thread 0 (BW)	Pre-processing						
Get Image Thread 1 (BW)							
Get Image Thread 2 (BW)							
Get Image Thread 3 (Color)							
Compute Thread 0		Triangulation		Homogen	Reconstruct Depth		
Compute Thread 1		MNCC					
...							
Compute Thread 7							
Time (ms) :	12.1	12.0	5.5	17.8	2	1.8	Total: 51.8 (~19.3fps)

# Work in progress

- Porting kernels to use HTAs
- Adding empirical auto-tuning framework
  - Tune for locality
  - Tune for performance on multi-cores and GPUs
  - Look at future architectures such as Intel's Larabee
- Further tuning of homogen using GPU in progress
- Investigation of parallelization of Delaunay triangulation
- Hybrid computing techniques
  - Don't let CPU/GPU cycles sit idle

