ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GSRC

Gigascale Systems Research Center

Many-core Computing

Can compilers and tools do the heavy lifting?

Wen-mei Hwu

FCRP GSRC, Illinois UPCRC, Illinois CUDA CoE, IACAT, IMPACT

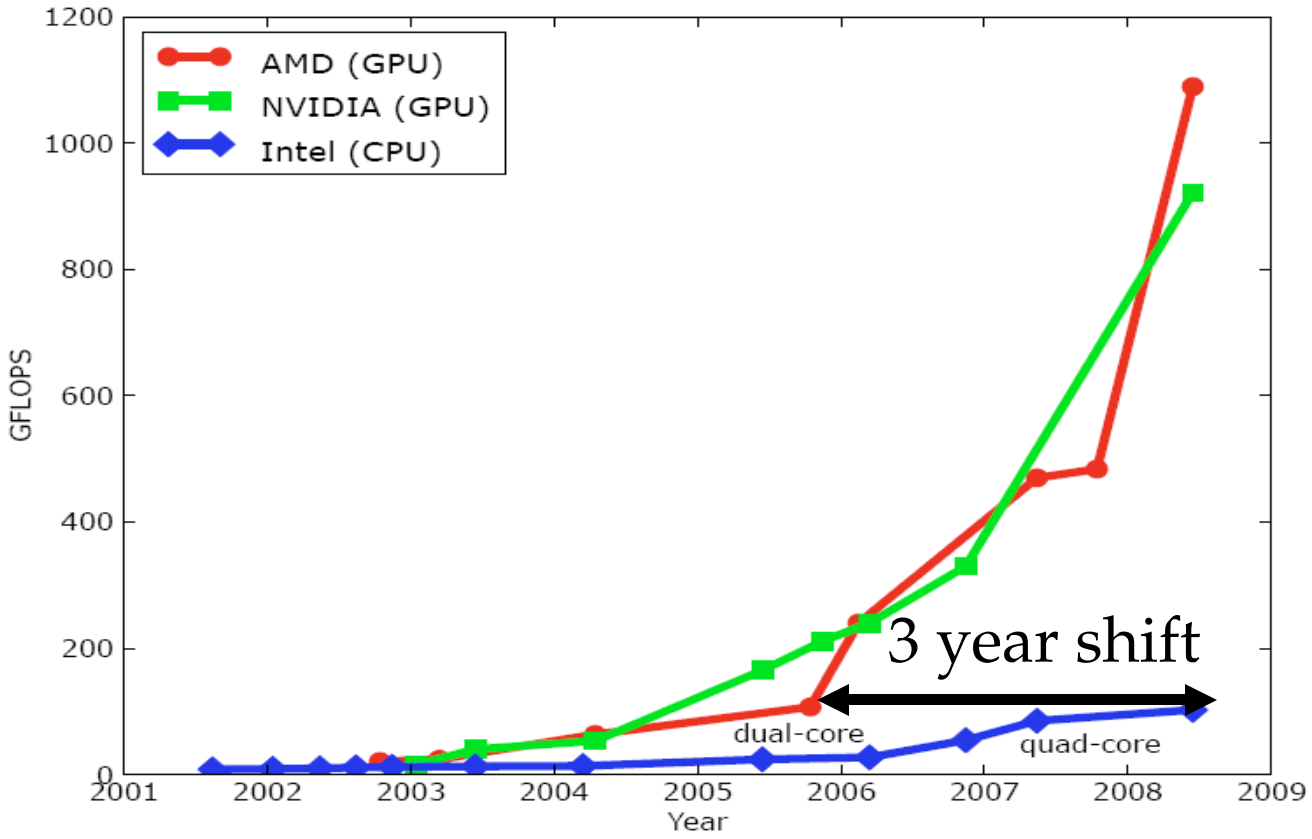
University of Illinois, Urbana-Champaign

Outline

- Parallel application outlook
- Heavy lifting in “simple” parallel applications
- Promising tool strategies and early evidence
- Challenges and opportunities

SoC specific opportnities and challenges?

The Energy Behind Parallel Revolution



Courtesy:
John Owens

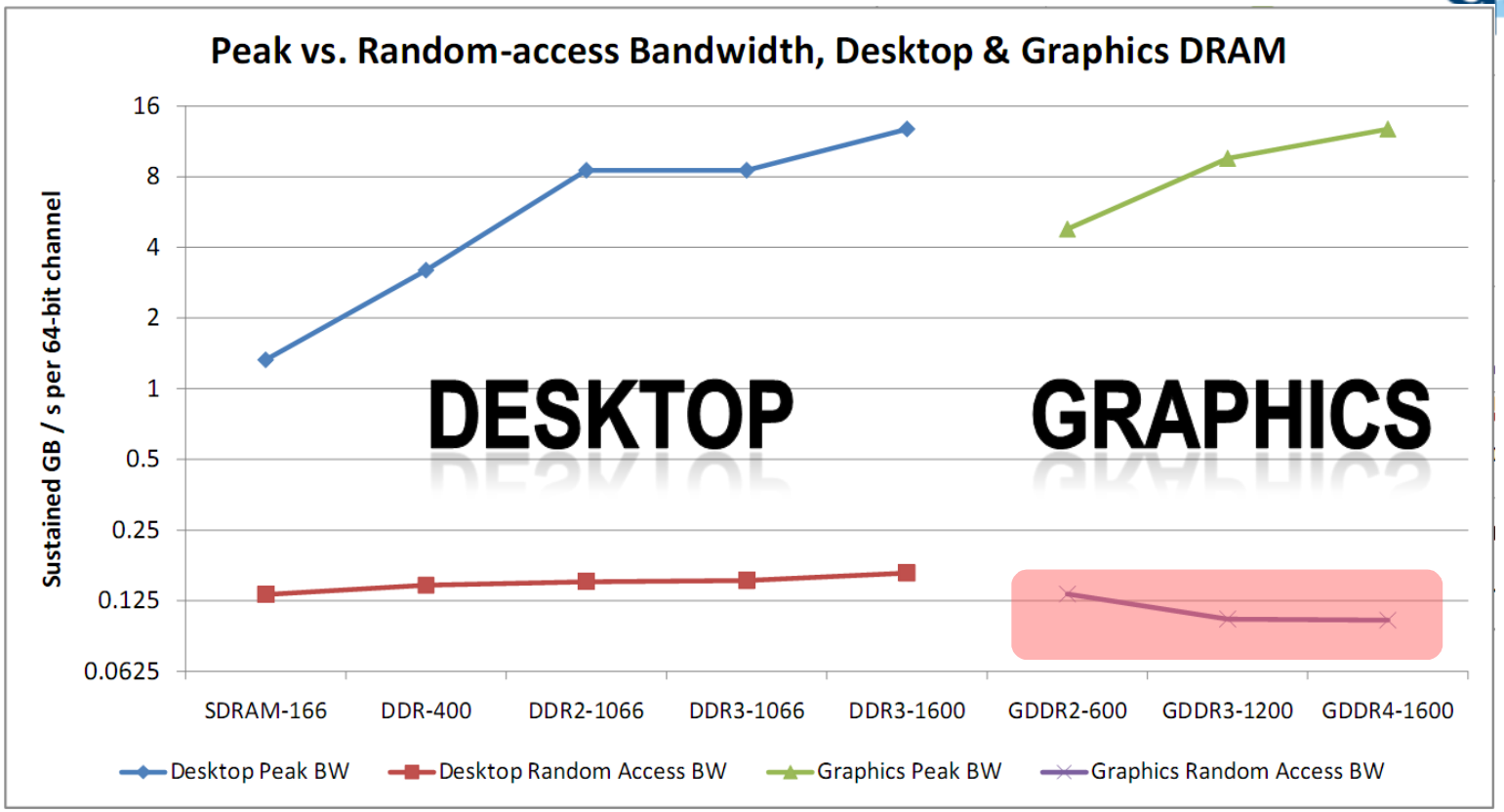
Courtesy: John Owens

My Predictions

- Mass market parallel apps will focus on many-core GPUs in the next three to four years
  - NVIDIA GeForce, ATI Radon, Intel Larrabee
  - “Simple” (vector) parallelism
  - Dense matrix, single/multi-grids, stencils, etc.

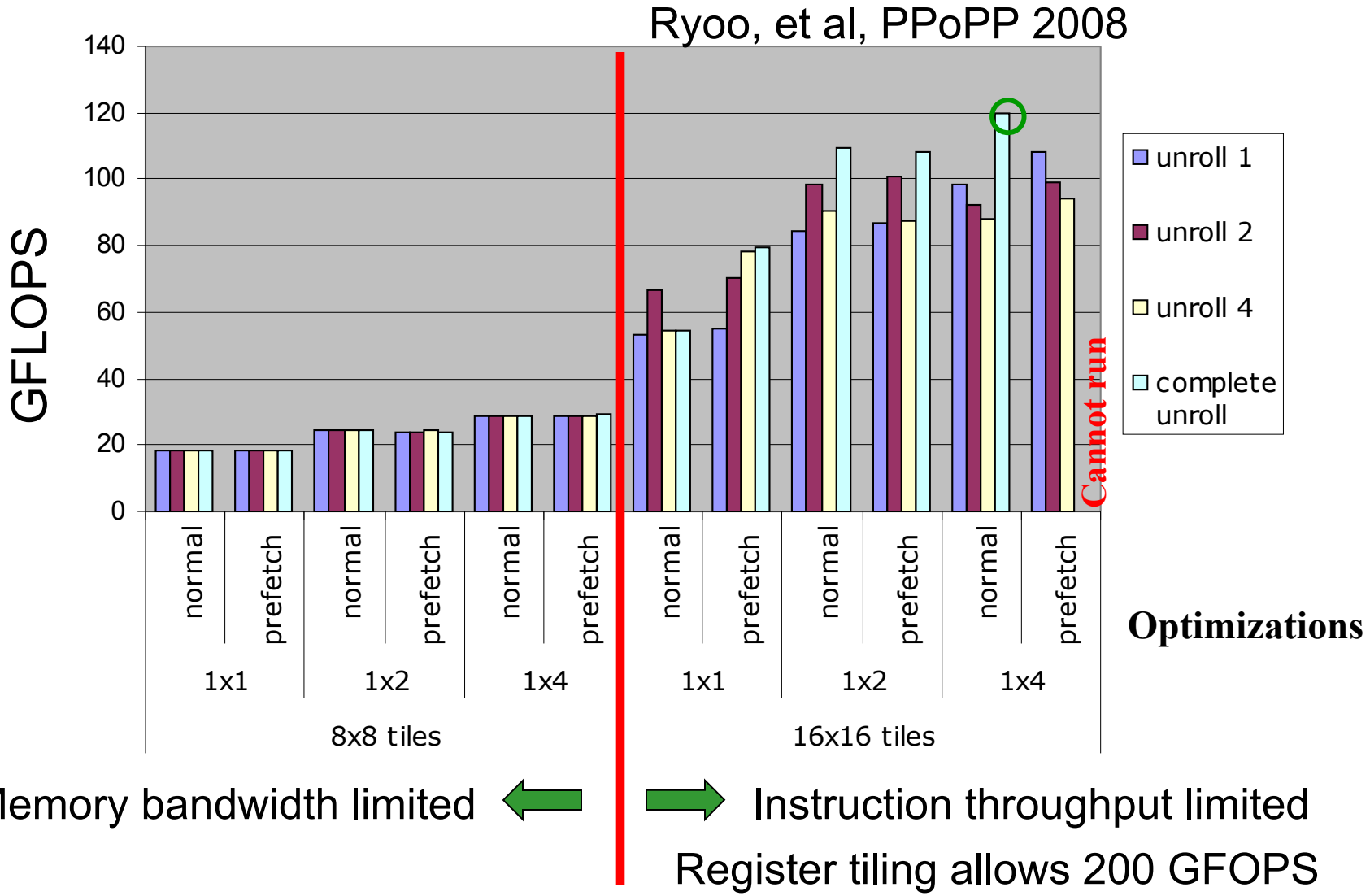
- Even “simple” parallelism can be challenging
  - Memory bandwidth limitation
  - Portability and scalability
  - Heterogeneity and data affinity

DRAM Bandwidth Trends



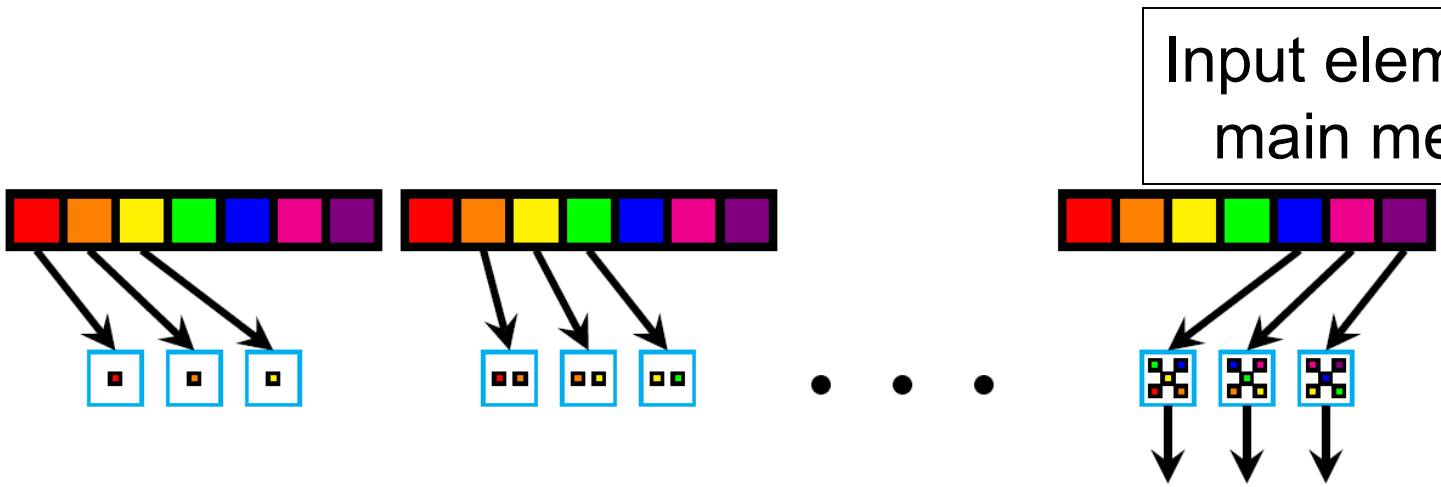
- Random access BW 1.2% of peak for DDR3-1600, 0.8% for GDDR4-1600 (and falling)
- 3D stacking and optical interconnects will unlikely help.

Dense Matrix Multiplication Example (G80)



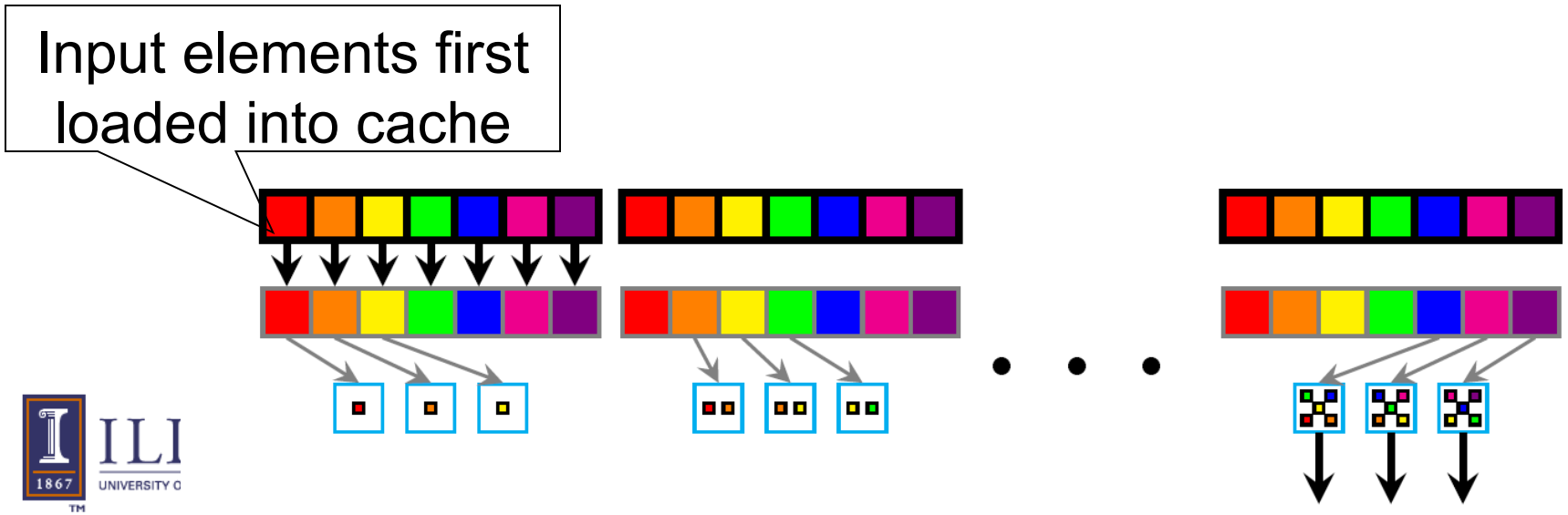
Example: Convolution – Base Parallel Code

- Each parallel task calculates an output element
- Figure shows
  - 1D convolution with K=5 kernel
  - Calculation of 3 output elements
- Highly parallel but memory bandwidth inefficient
  - Uses massive threading to tolerate memory latency
  - Each input element loaded up to K times



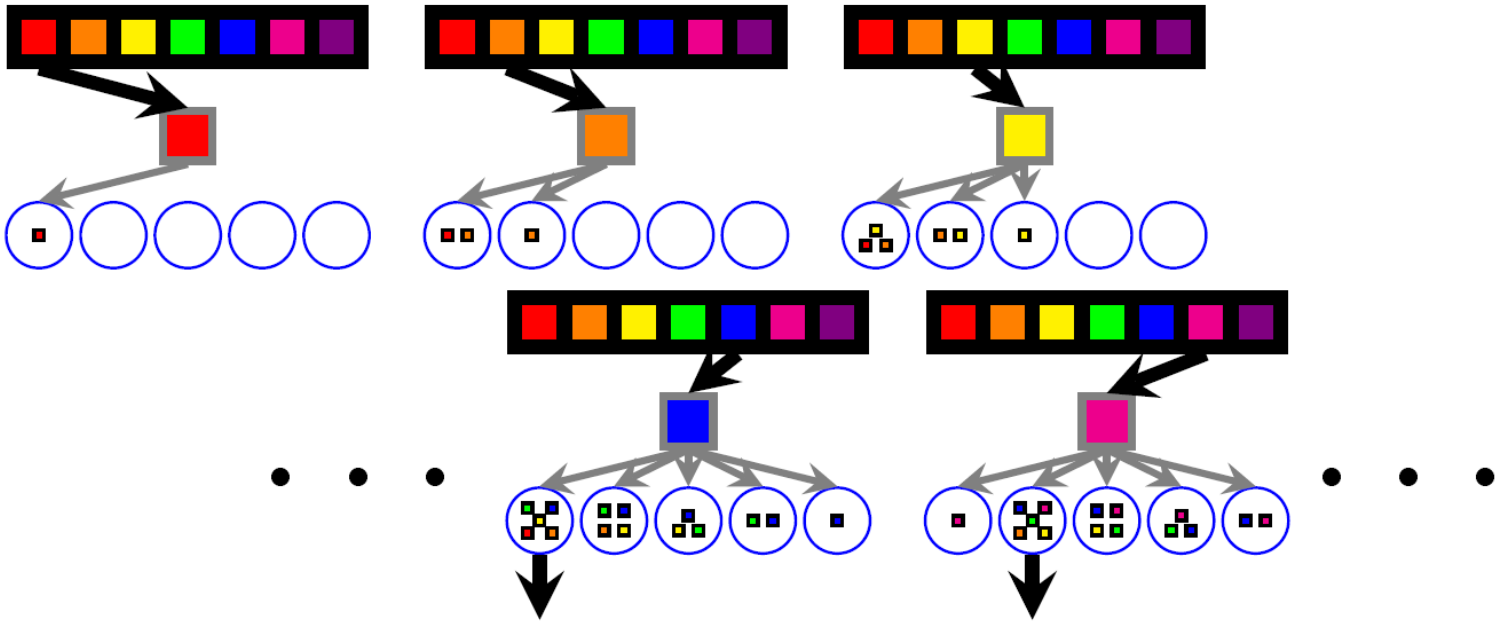
Example: convolution using on-chip caching

- Output elements calculated from cache contents
  - Each input element loaded only once
  - Cache pressure – (K-1+N) input elements needed for N output elements
    - $7/3 = 2.3$, $7^2/3^2 = 5.4$, $7^3 / 3^3 = 12$
    - For small caches, the benefit can be significantly reduced due to the high-ratio of additional elements loaded.



Example: Streaming for Reduced Cache Pressure

- Each input element is loaded into cache in turn
  - Or a (n-1)D slice in nD convolution
- All threads consume that input element
  - "loop skewing" needed to align the consumption of input elements
  - This stretches the effective size of the on-chip cache



Many-core GPU Timing Results

- Time to compute a 3D k^3-kernel convolution on 4 frames of a 720X560 video sequence
  - All times are in milliseconds
  - Timed on a Tesla S1070 using one G280 GPU

k	Baseline (3.1)	Shared Memory (3.2)	Streaming (3.3)	3D Fourier (3.4)	Hybrid Fourier (3.4)
5	16	11	4	24	15
7	44	15	8	34	17
9	96	48	16	39	20
11	180	77	27	44	23
13	295		45	74	24
15	454		75	56	26

Multi-core CPU Timing Results

- Time to compute a 3D k^3-kernel convolution on 4 frames of a 720X560 video sequence
  - All times are in milliseconds
  - Timed on a Dual-Socket Duo-Core 2.4 GHz Opteron system, all four cores used

k	Baseline (3.1)	Shared Memory (3.2)	Streaming (3.3)	3D Fourier (3.4)	Hybrid Fourier (3.4)
5	136	117	140	128	133
7	362	289	317	235	152
9	1018	597	614	208	213
11	1954	1065	1135	238	237
13	3590	1733	1771	267	271
15	6453	2676	2633	338	356

Application Example: Up-resolution of Video

Nearest & bilinear interpolation:
+ Fast but low quality

Bicubic interpolation:
+ Higher quality but computational intensive

Implementation Overview

- Step 1: Find the coefficients of the shifted B-Splines.
  - Two single pole IIR filters along each dimension
  - Implemented with recursion along scan lines
- Step 2: Use the coefficients to interpolate the image
  - FIR filter for bicubic interpolation implemented as a k=4 2D convolution with $(2+16+2)^2$ input tiles with halos
  - Streaming not required due to small 2D kernel, on-chip cache works well as is.
- Step 3: DirectX displays from the GPU

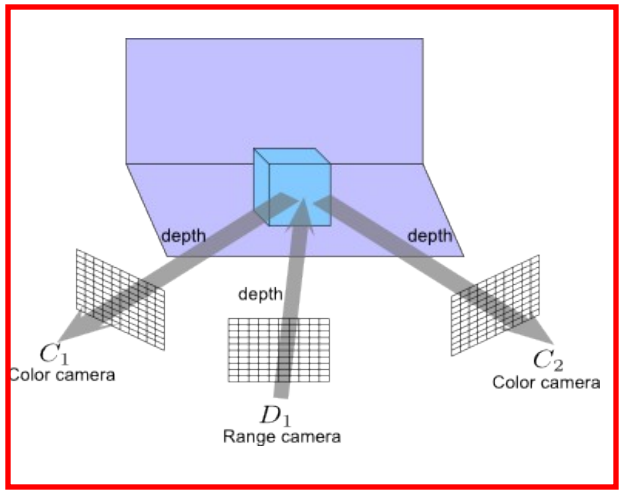
Upconversion Results

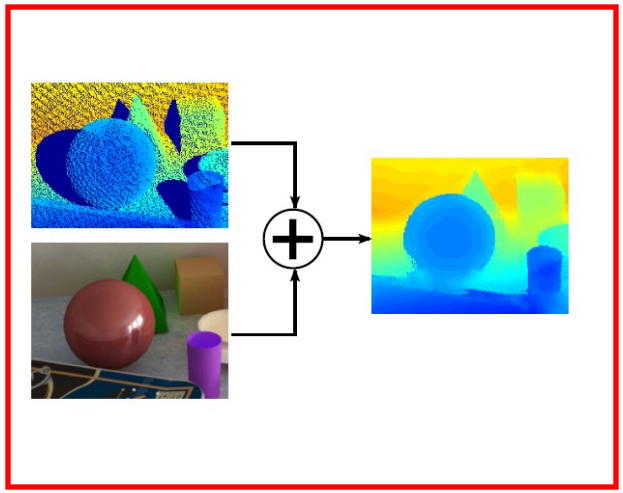
- Parallelize bicubic B-spline interpolation
  - Interpolate QCIF (176x144) to nearly HDTV (1232x1008)
  - Improved quality over typical bilinear interpolation
  - Improved speed over typical CPU implementations
    - Measured 350x speedup over un-optimized CPU code
    - Estimated 50x speedup over optimized CPU code from inspection of CPU code
  - Real-time!

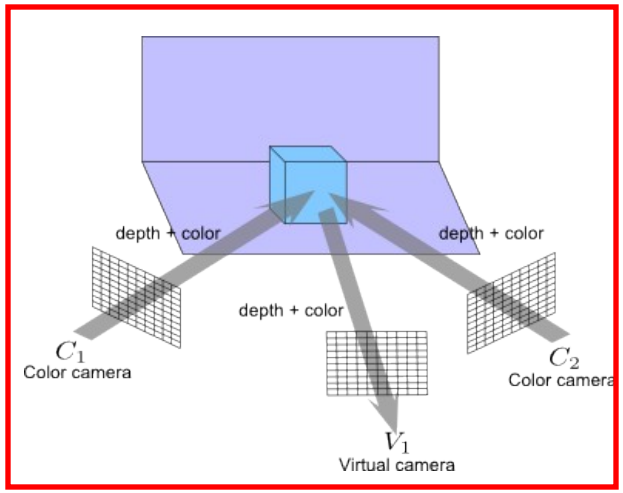
	Hardware	IIR	FIR
CPU	Intel Pentium D	5 ms	1689 ms
GPU	nVidia GeForce 8800 GTX	1 ms	4 ms

Application Example: Depth-Image Based Rendering

- Three main steps:
  - Depth propagation
  - Color-based depth enhancement
  - Rendering

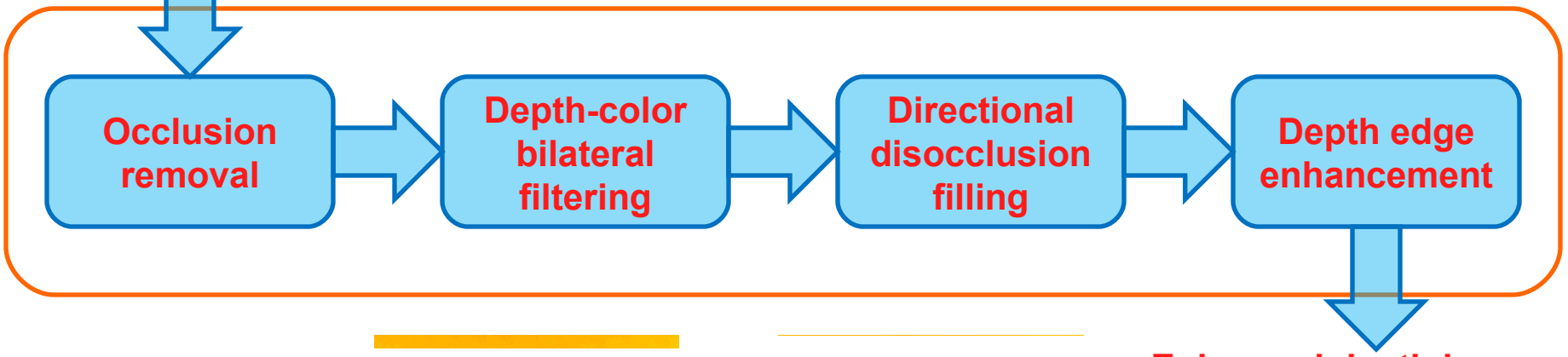




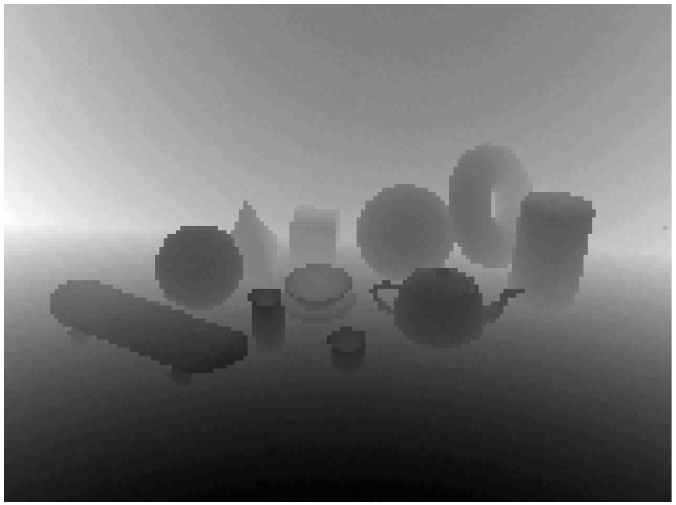


Color-based depth enhancement

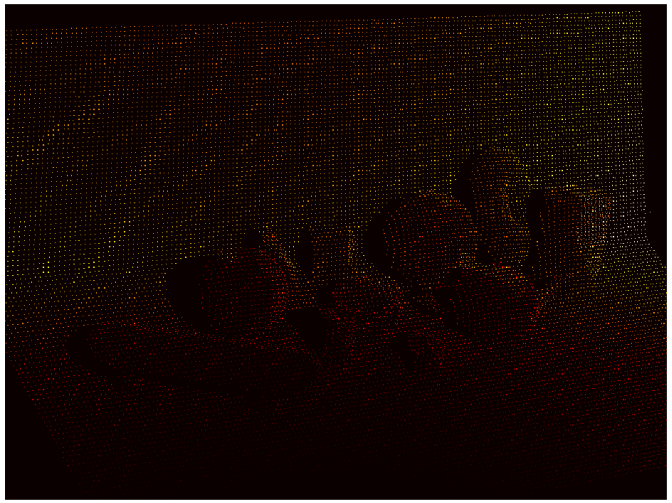
Propagated depth image at color view



Enhanced depth image



Propagated depth



Depth – color bilateral filteri

$$d_A = \frac{1}{W_A} \sum_{B \in S_A} G_{\sigma_s^2}(|\vec{x}_A - \vec{x}_B|) \cdot G_{\sigma_r^2}(|I_A - I_B|) \cdot d_B$$

$$W_A = \sum_{B \in S_A} G_{\sigma_s^2}(|\vec{x}_A - \vec{x}_B|) \cdot G_{\sigma_r^2}(|I_A - I_B|)$$

d_A : *depth value of point A.*

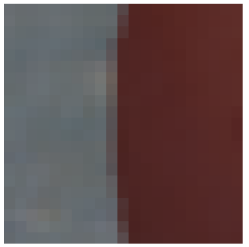
I_A : *color value of point A.*

$\vec{x}_A = [u_A, v_A]$: *2D coordinate of point A.*

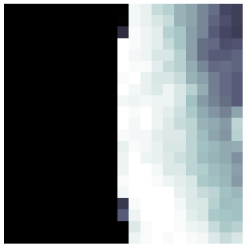
S_A : *set of A neighboring points.*

$G_\sigma(|\vec{x}|) = exp\left(\frac{-|\vec{x}|^2}{2\sigma^2}\right)$: *Gaussian kernel.*

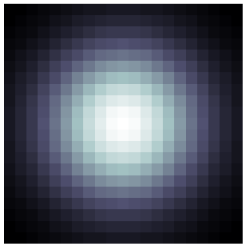
W_A : *normalizing term.*



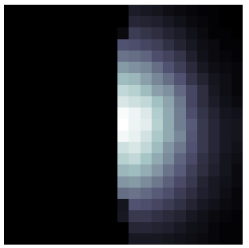
I



$G_{\sigma_r^2}(I_A - I_B)$

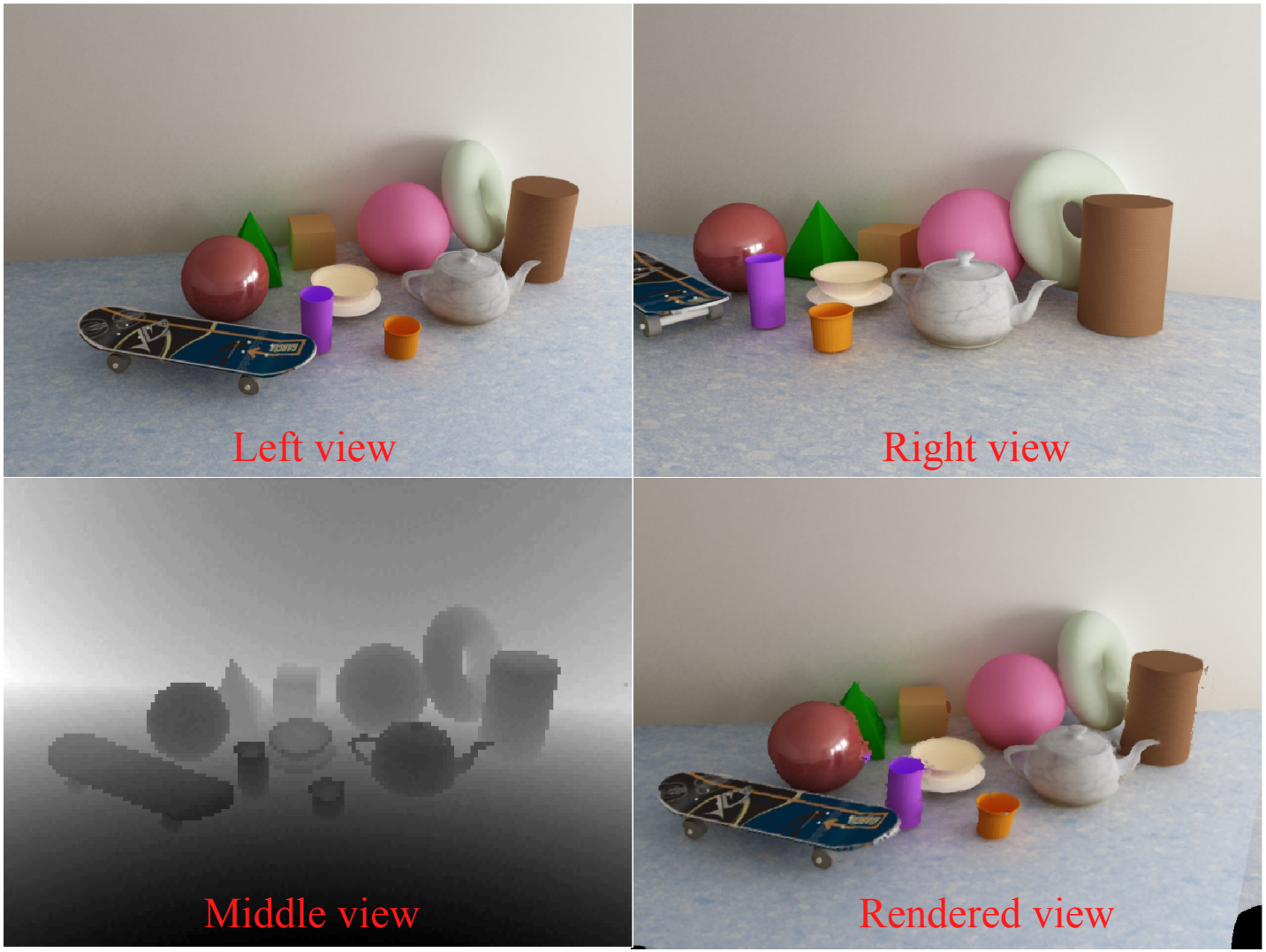


$G_{\sigma_s^2}(x_A - x_B)$



$G_{\sigma_s^2} * G_{\sigma_r^2}$

DIBR Visual Results



DIBR Time results

- Depth propagation.
  - Not computationally intensive but hard to parallelize
  - Each pixel in the depth view is be copied to the corresponding pixel in a different color view.
  - 3D-to-2D projection, many-to-one mapping.
  - Atomic functions are used, current work to improve with sort-scan and binning algorithms.
- Depth-color bilateral filter (DCBF)
  - Computational expensive.
  - Similar to 2D convolution. Similar parallelism techniques work well

	Hardware	Depth propagation	DCBF
CPU	Intel Core 2 Duo E8400 3.0GHz	38 ms	1041 ms
GPU	NVIDIA GeForce 9800 GT	24 ms	14 ms
Speedup		1.6x	74.4x

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GSRC

Gigascale Systems Research Center

Some upcoming tools

Gluon – specification information enables robust co-parallelization. (Illinois)

- Developers specify pivotal information at function boundaries
  - Heap data object shapes and sizes
  - Object access guarantees
  - Some can be derived from global analyses but others can be practically infeasible to extract from source code.

- Compilers leverage the information to
  - Expose and transform parallelism
  - Perform code and layout transformations for locality

Gluon Parallelism Exposure Example

```
struct data {
   float x; float y; float z;
};

int cal_bin(str
                struct data *b) {
1. __spec(*a: r, (data)[1]);
2. __spec(*b: r, (data)[1]);
3. __spec(ret_v: range(0,SZ));

 int bin = .                    *b*/
 return(bin
}
```

No side effect on d elements

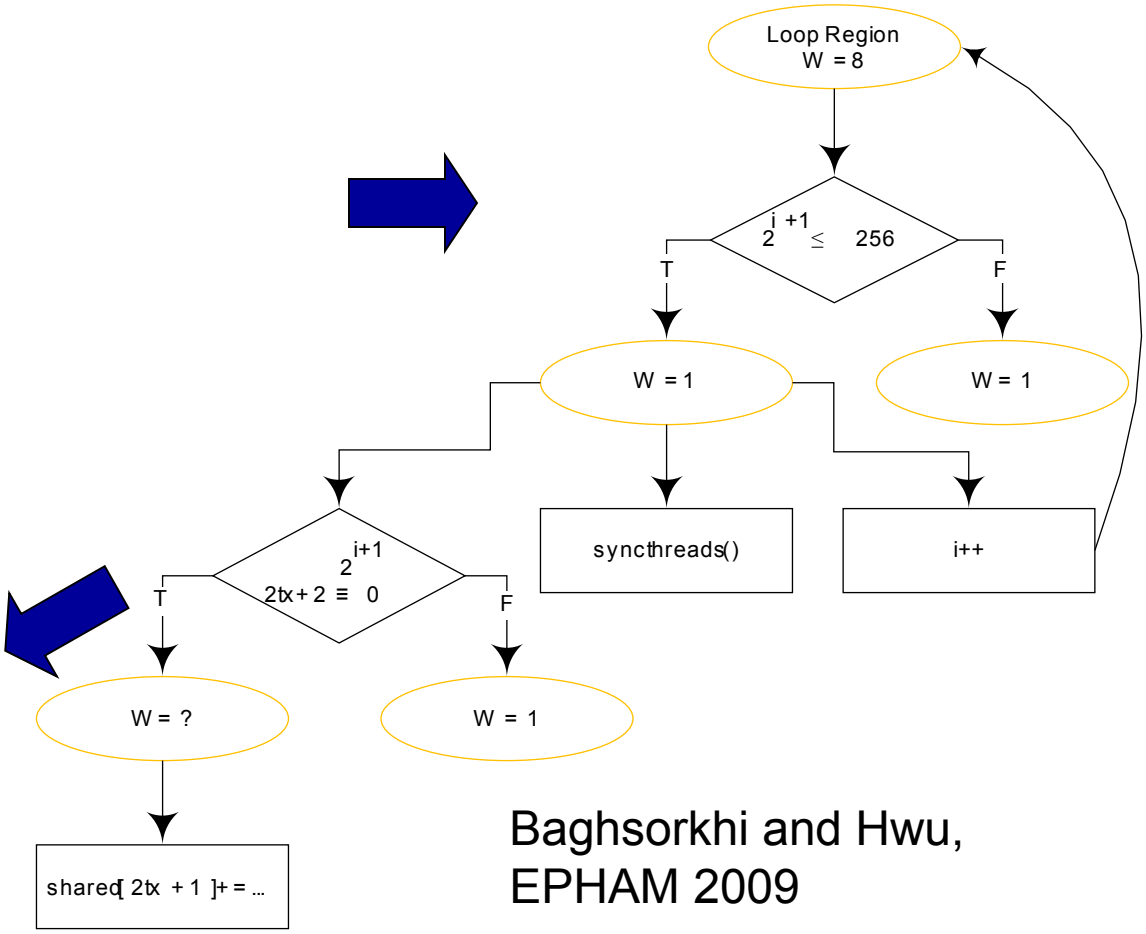
hist safe to privatize

data layout can be done safely

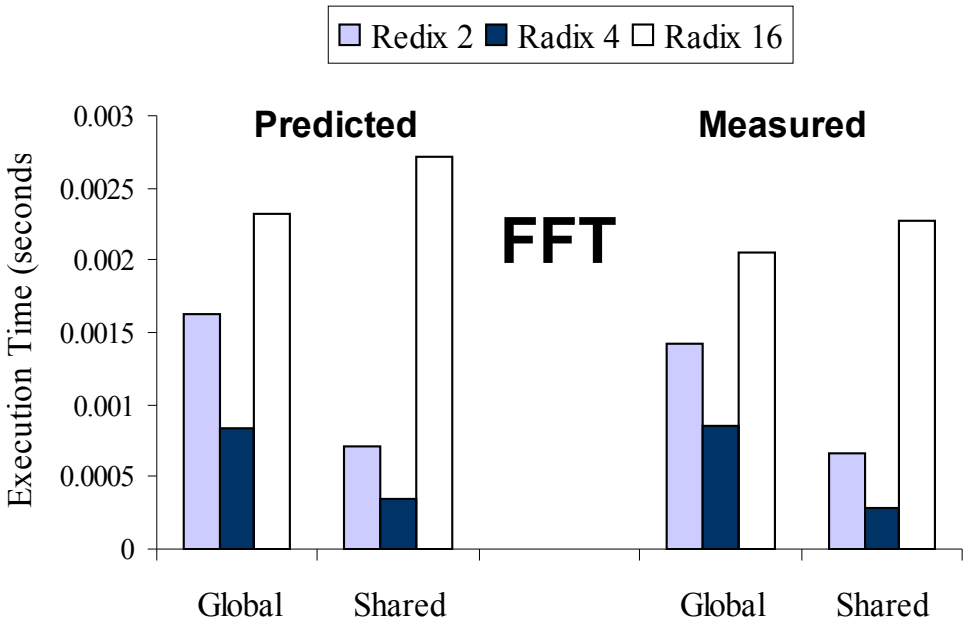
```
int *tpacf(int len, struct data *d) {
4. __spec(d: r, (int)[len]);

   int *hist = malloc(SZ*sizeof(int));
   __spec(hist: (int)[SZ]);

   for (i=0; i < len; i++) {
      for (j = 0; j < len; j++) {

6.       int bin = cal_bin(&d[i],&d[j]);

7.       hist[bin] += 1;
      }
   }
}
```

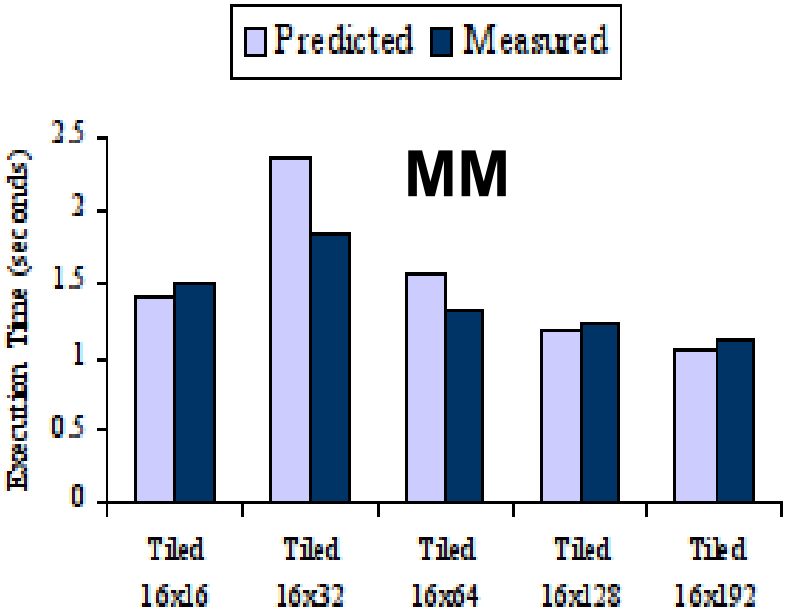
Program Dependence Graph Based Application Performance Prediction (Illinois)

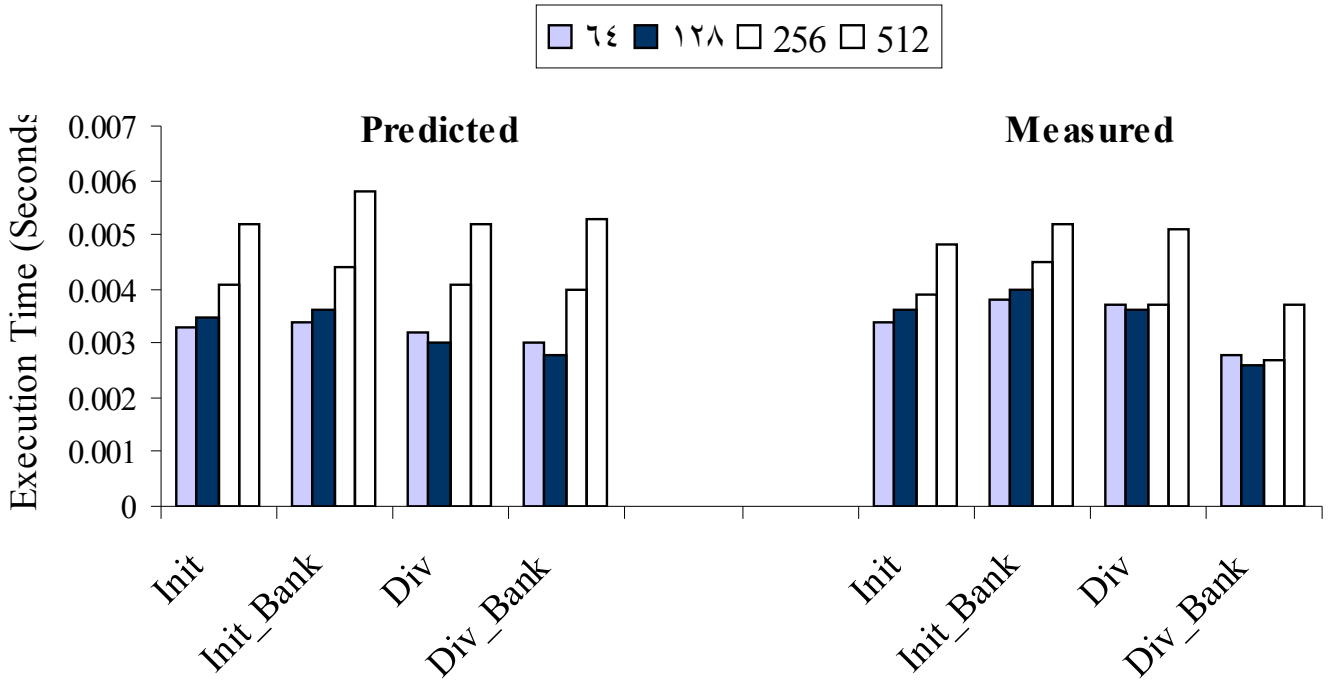
Predicting the performance effect of compiler transformations.



Baghsorkhi and Hwu, EPHAM 2009


Redix 2
Radix 4
Radix 16
Predicted
Measured
FFT
Execution Time (seconds
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
Global
Shared
Global
Shared


Predicted
Measured
MM
Execution Time (seconds)
2.5
2
1.5
1
0.5
0
Tiled 16x16
Tiled 16x32
Tiled 16x64
Tiled 16x128
Tiled 16x192


٦٤
١٢٨
256
512
Predicted
Measured
Execution Time (Seconds
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
Init
Init_Bank
Div
Div_Bank
Init
Init_Bank
Div
Div_Bank

Automating Memory Coalescing using Gluon and PDG

```
1   #define ASIZE 3000
    #define TPB 32

    void
5   kernel (float *a, float *b)
    {
      __annotation (L"__global__ TPB 1");
      __annotation (L"garray a 2 4 ASIZE ASIZE");
      __annotation (L"garray b 2 4 ASIZE ASIZE");
10
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
      int i;
15
      __annotation (L"BoundChk");
      if (bki * TPB + thi >= ASIZE)
        return;

20    for (i = 0; i < ASIZE; i++)
        {
          __annotation (L"loop i 0 ASIZE 1");
          b[(bki*TPB+thi)*ASIZE + i] =
            a[(bki*TPB+thi)*ASIZE + i] * t;
25      }
    }
```

```
1   #define ASIZE 3000
    #define TPB 32

    __global__ void
5   kernel (float *a, float *b)
    {
      int thi = threadIdx.x;
      int bki = blockIdx.x;
      float t = (float) thi + bki;
10    int i;

      int j, End, k;
      __shared__ float a_shared[TPB][TPB];
      __shared__ float b_shared[TPB][TPB];
15
      End = ASIZE % TPB == 0 ? ASIZE / TPB : (ASIZE/TPB)+1;
      for (j = 0; j < End; j++)
        {
          /* Coalesce loads */
20        __syncthreads();
          for (k = 0; k < TPB; k++)
            {
              if ((j*TPB + thi < ASIZE) &&
                  ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE * ASIZE))
25              a_shared[k][thi] = a[(bki*TPB + k)*ASIZE + j*TPB + thi];
            }
          __syncthreads();

          /* Conditions:
30         * TPB && obey original end && !(early exit condition)
           */
          for (i = 0;
           (i < TPB) && (j*TPB+i < ASIZE) && !(bki * TPB + thi >= ASIZE);
           i++)
35          {
              b_shared[thi][i] = a_shared[thi][i] * t;
            }

          /* Coalesce stores */
40        __syncthreads();
          for (k = 0; k < TPB; k++)
            {
              if ((j*TPB + thi < ASIZE) &&
                  ((bki*TPB+k)*ASIZE + j*TPB + thi < ASIZE * ASIZE))
45              b[(bki*TPB + k)*ASIZE + j*TPB + thi] = b_shared[k][thi];
            }
          __syncthreads();
        }
    }
```

Shared Memory

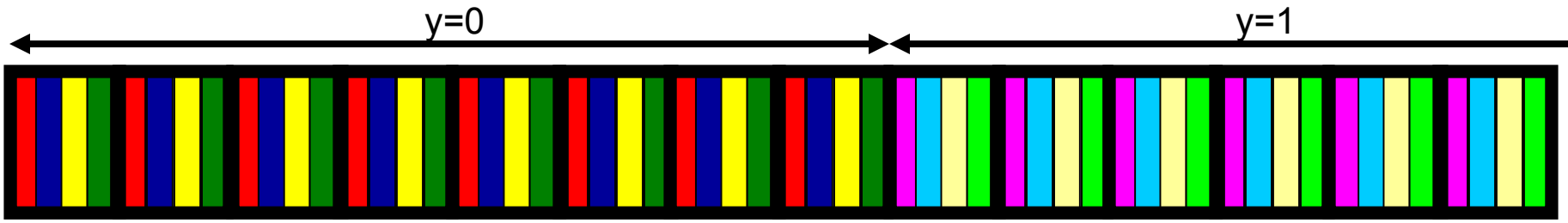
Loop Tiling

Coalesced Loads

Computation Kernel

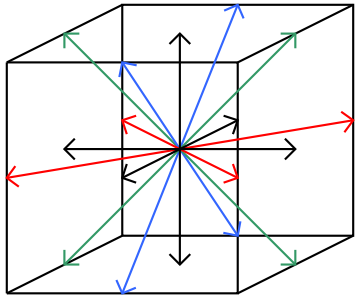
Coalesced Stores

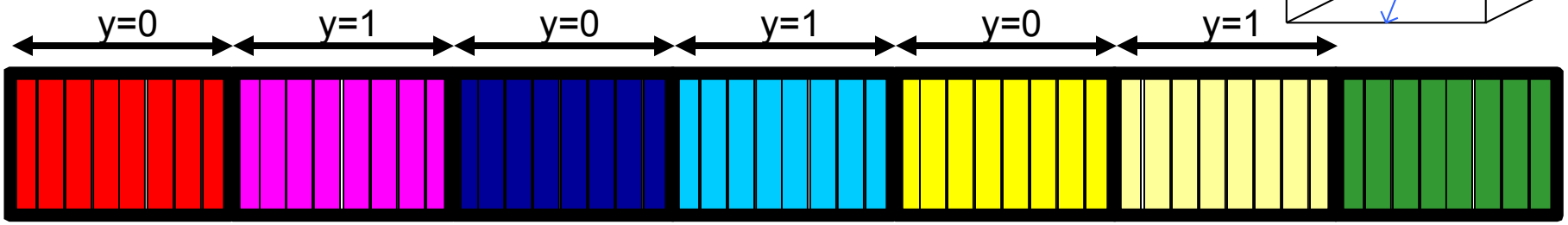
Memory Layout Transformation Lattice-Boltzmann Method Example



Array of Structure: [z][y][x][e]

F(z, y, x, e) = z * |Y| * |X| * |E| + y * |X| * |E| + x * |E| + e





Structure of Array: [e][z][y][x]

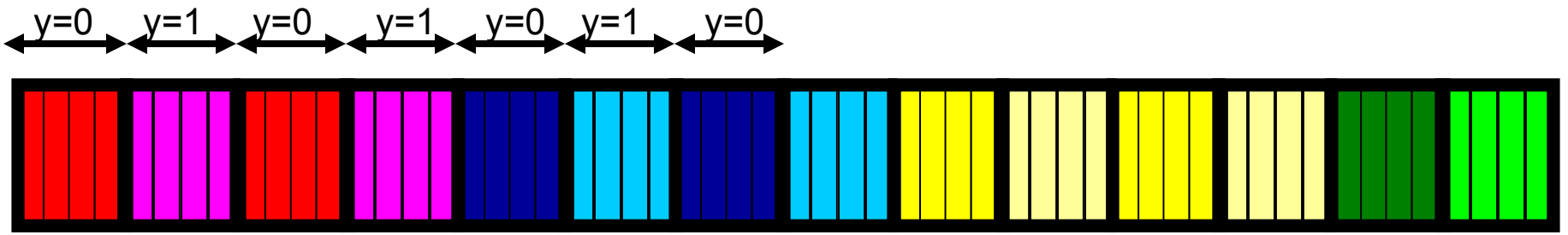
F(z, y, x, e) = e * |Z| * |Y| * |X|+ z * |Y| * |X| +y * |X| + x

4X faster than AoS on GTX280

The best layout is neither SoA nor AoS

- Tiled Array of Structure, using lower bits in x and y indices, i.e. $x_{3:0}$ and $y_{3:0}$ as lowest dimensions: [z][$y_{31:4}$][$x_{31:4}$][e][$y_{3:0}$][$x_{3:0}$]
  - $F(z, y, x, e) = z * \lceil |Y|/2^4 \rceil * \lceil |X|/2^4 \rceil * |E| * 2^4 * 2^4 + y_{31:4} * \lceil |X|/2^4 \rceil * |E| * 2^4 * 2^4 + x_{31:4} * |E| * 2^4 * 2^4 + e * 2^4 * 2^4 + y_{3:0} * 2^4 + x_{3:0}$

- 6.4X faster than AoS, 1.6X faster than SoA on GTX280:
  - Better utilization of data by neighboring cells
  - This is a scalable layout: same layout works for very large objects



Summary

- Tools must understand and manage data accesses
  - Partnership between developers and tools
  - Key to “good” parallelism
  - Must balance between developer specification and program analysis
  - Key to portability and productivity

- “Simple” many-core programming tools within reach
  - Memory bandwidth optimizations
  - Parallel execution granularity adjustments
  - Well-known algorithm changes
  - Heterogeneous computing mapping and data transfers
  - Haves and Have-Nots of many-core computing

- http://www.parallel.illinois.edu/
  - Courses, seminars, publications, tools,
  - UPCRC. CUDA Center of Excellence. IACAT. …

Current Challenges

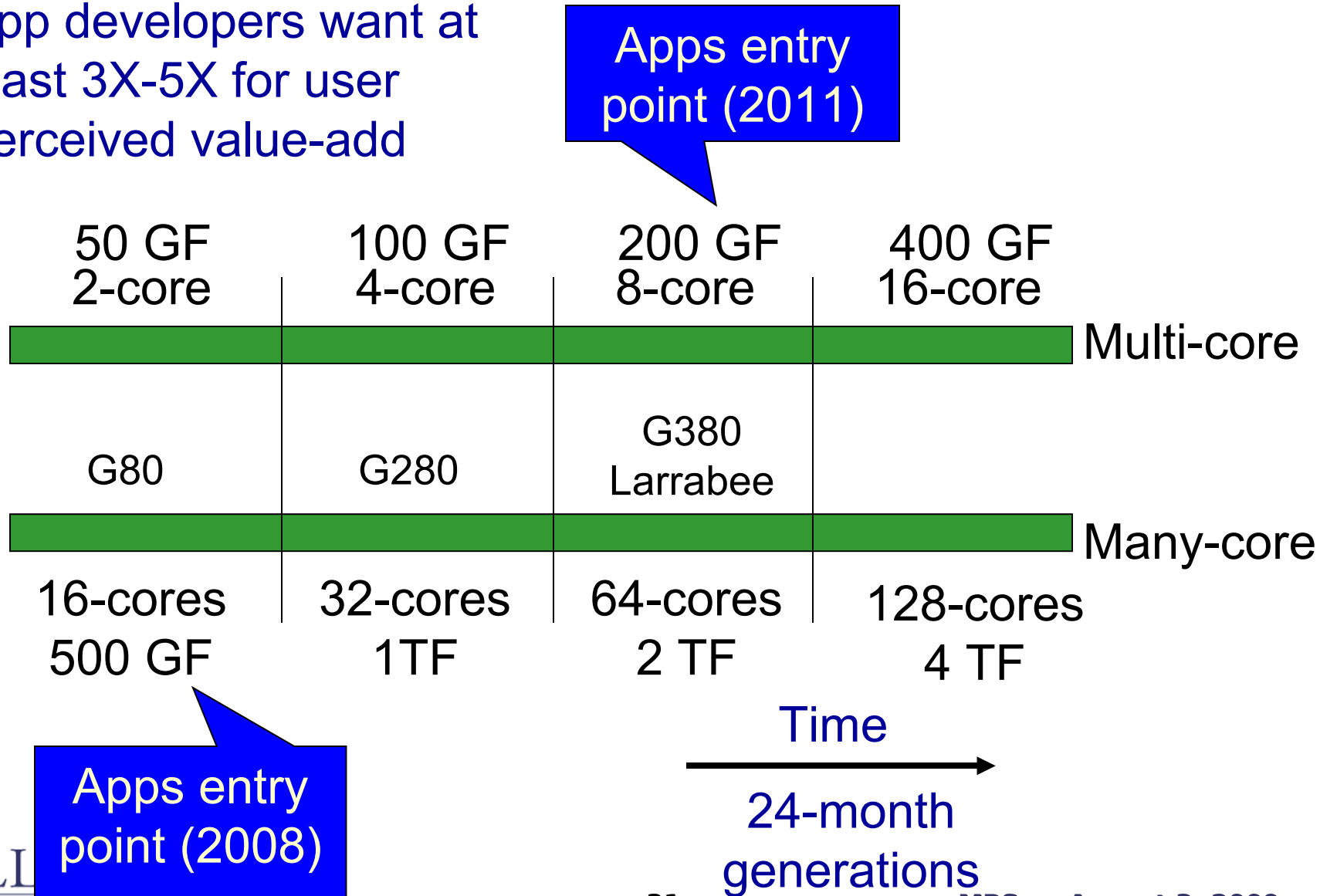
- Execution Models
  - Currently single kernel execution
  - Moving to multiple kernel steaming
- Irregular Algorithms and Data Structures
  - Data layout and tiling transformations for sparse matrices and spatial data structures need to be developed and automated
  - Graph algorithms lack conceptual foundation for locality
- Usability
  - Tools and interfaces may be still too tedious and confusing for application developers

GSRC

Thank you! Any questions?

Applications Entry Timeframes

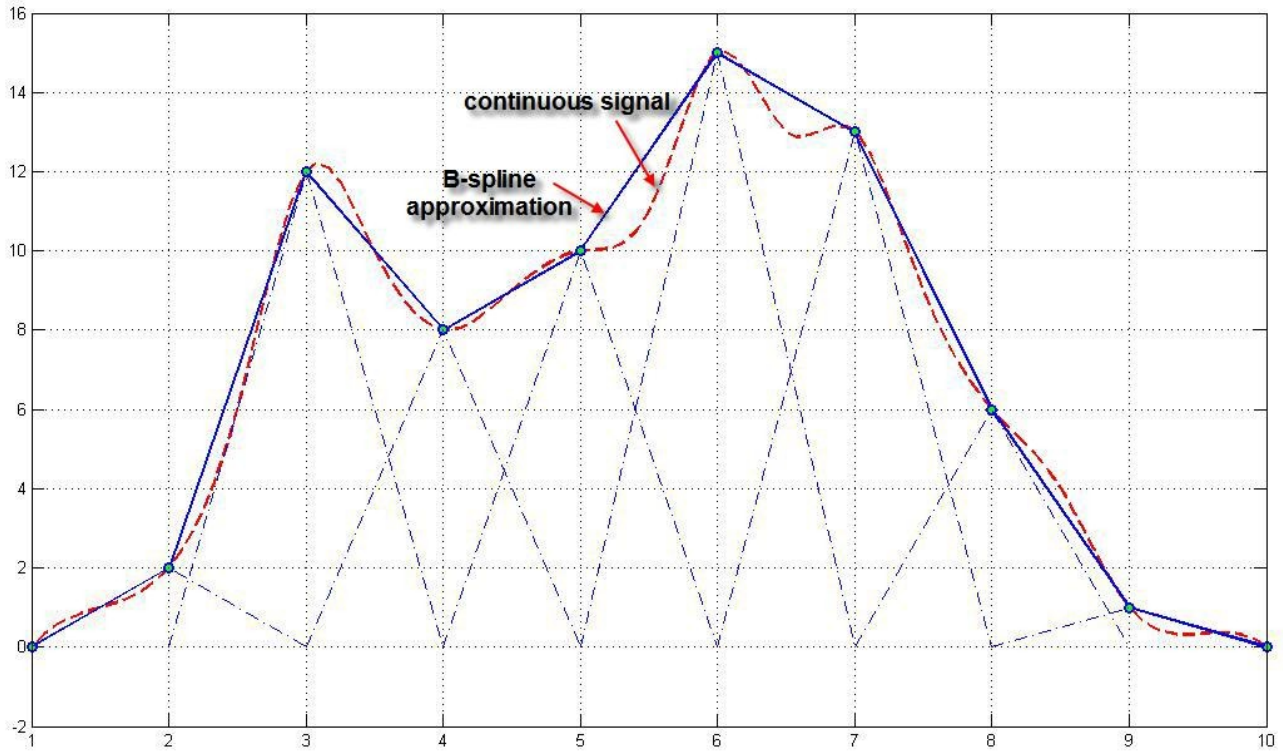
App developers want at least 3X-5X for user perceived value-add



FIR implementation

Linear interpolation for 1D case

Cubic interpolation for 1D case



$$k = x - \lfloor x/R \rfloor * R$$

$$g[x] = c[x-1]w0[k] + c[x]w1[k] + c[x+1]w2[k] + c[x+2]w3k]$$

Depth propagation

- Propagate depth information from the depth camera to each color camera.

- 2D point to 3D ray mapping relation:

$$\vec{r} = \begin{bmatrix} \vec{s}_{ijk} & \vec{t}_{ijk} & f * \vec{w}_{ijk} \end{bmatrix} \begin{bmatrix} x_s & x_t & x_w \end{bmatrix}^T = P\vec{x}$$

- Warping equation: (L. McMillan, 1997)

$$\vec{x}_d = P_d^{-1} \left(\frac{|P_r \vec{x}_r|}{d(\vec{x}_r)} (\vec{C}_r - \vec{C}_d) + P_r \vec{x}_r \right)$$

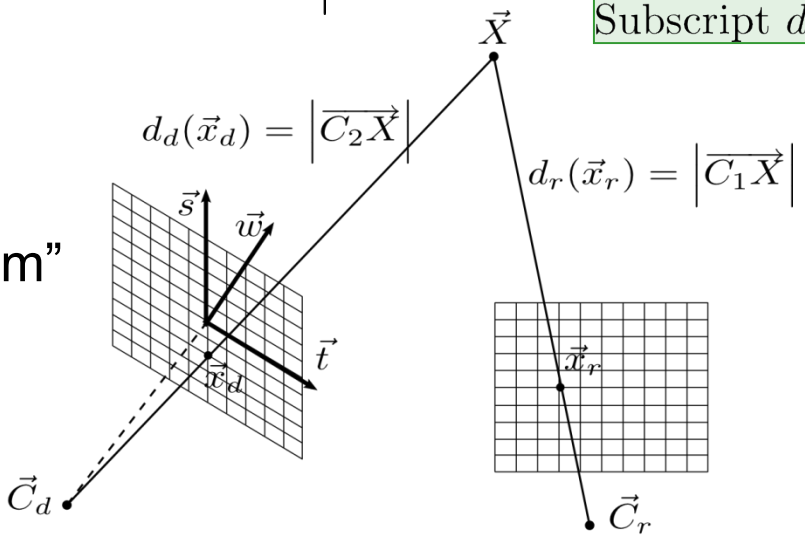
- Compute new depth values:

$$d_d(\vec{x}_d) = \left| \overrightarrow{C_2 X} \right| = \left| \overrightarrow{C_2 C_1} + \overrightarrow{C_1 X} \right|$$

Notation:

$\{\vec{s}, \vec{t}, \vec{w}\}$ = local view coordinates.
$\{\vec{i}, \vec{j}, \vec{k}\}$ = global coordinates.
f = focal length of the camera.
P = point-to-ray projection matrix.
$\vec{r}$ = 3D ray.
$\vec{x}$ = 2D coordinate of a pixel.
$\vec{X}$ = 3D projection of $\vec{x}$.
$\vec{C}$ = camera center.
Subscript r = reference view.
Subscript d = desired view.

A form of 2D "histogram" challenging for GPUs



Illinois Vision Video (ViVid) Framework

- Constructed by vision experts with parallel programming expertise
- For video analysis, enhancement, and synthesis apps
- Python module bindings for seamless CPU/GPU deployment
  - MPEG2 Video Decoder and file I/O- C++ (through OpenCV)
  - 2D Convolution - C++, Python, CUDA
  - 3D Convolution - C++, Python, CUDA
  - 2D Fourier Transform - C++, Python, CUDA
  - 3D Fourier Transform - C++, Python, CUDA
  - Optical Flow Computation - C++ (through OpenCV)
  - Motion Feature Extraction - C++, Python, CUDA
  - Pairwise distance between 2 collections of vectors - C++, Python, CUDA
- Domain knowledge capture for optimization and auto-tuning

M. Dikman, et al, University of Illinois, Urbana-Champaign

GMAC Heterogeneous Computing Runtime (UPC/Illinois)

- Software-Based Unified CPU/GPU Address Space
  - Same address/pointer used by CPU and GPU
  - No explicit data transfers
- Data reside mainly in GPU memory
  - Close to compute power
  - Occasional CPU access for legacy libraries and I/O
- Customizable automatic data transfers:
  - Transfer everything (safe mode)
  - Transfer dirty data before kernel execution
  - **Transfer data as being produced (default)**
- Multi-process / Multi-thread support
- CUDA compatible, Linux alpha version available soon.