Maximizing OpenCL Performance

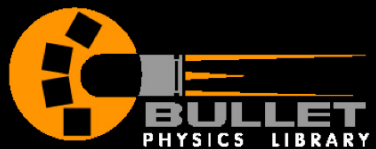
OpenCL
Accelerated Physics

Benedict Gaster
AMD OpenCL Architect

Bullet

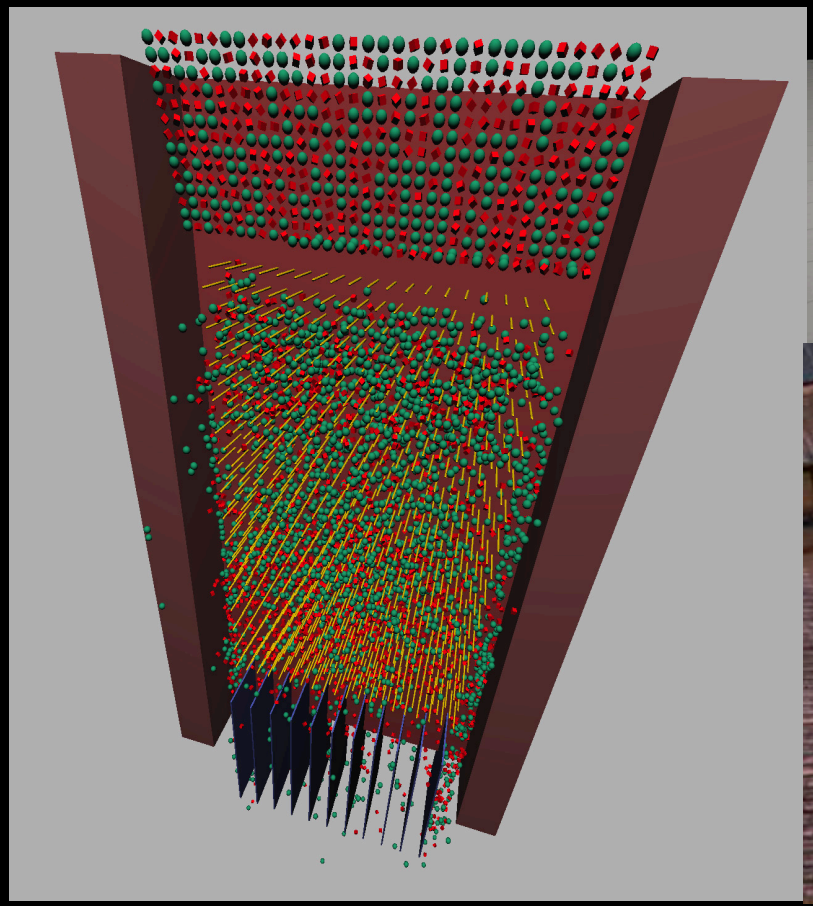
AMD

- An open source physics SDK
  - Third most used physics SDK
  - Zlib license for copy-and-use openness
  - Primary development by Erwin Coumans of Sony
- AMD collaborating on GPU acceleration
  - Cloth/soft body and fluids in OpenCL
  - Fully open-source contributions

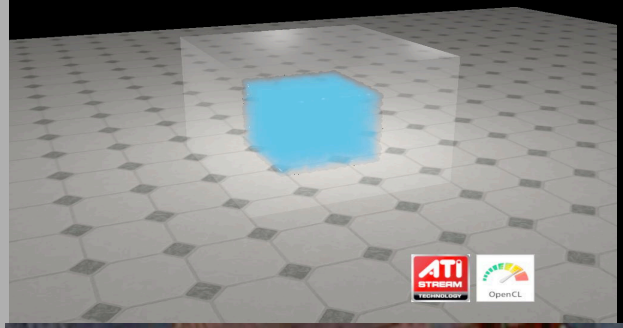


Bullet OpenCL Is…

AMD



Rigid bodies



Fluids

Cloth

An Introduction to Cloth Simulation

AMD

Masses and springs

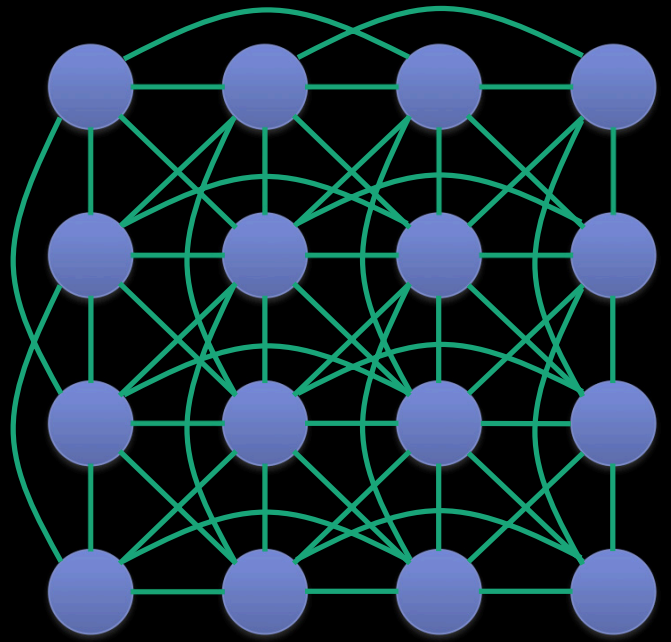
- A subset of the possible set of soft bodies
- Mass/spring system
  - Large collection of masses (particles)
  - Connect using spring constraints
  - Layout and properties change properties of cloth

An Introduction to Cloth Simulation

AMD

Springs with purpose

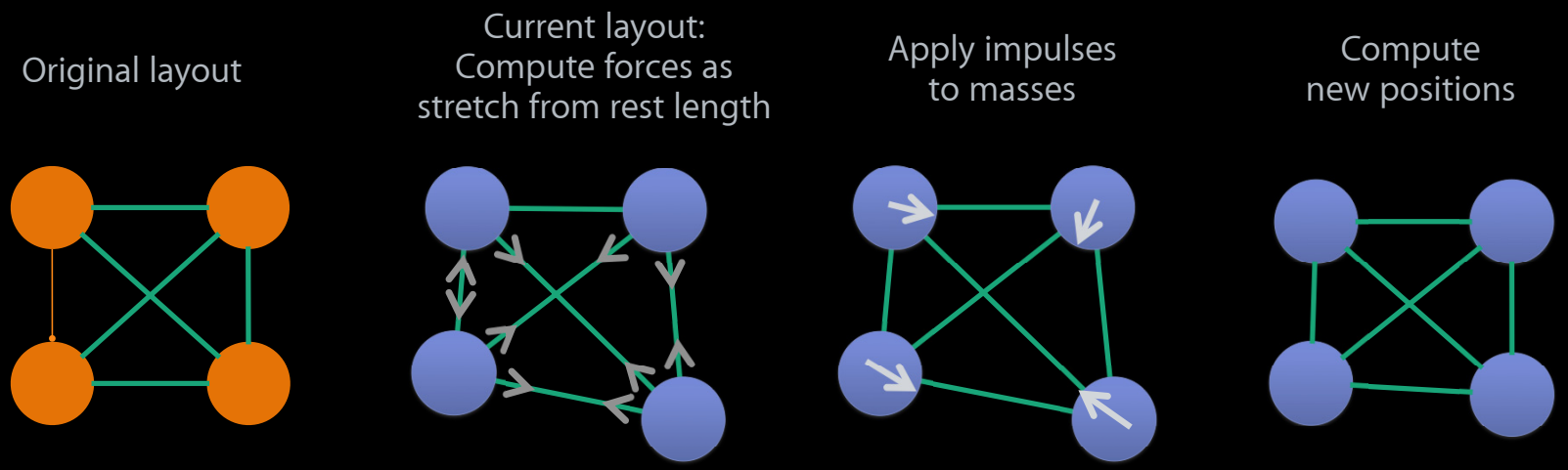
- Three main types of springs
  - Structural
  - Shearing
  - Bending



An Introduction to Cloth Simulation

Parallelism

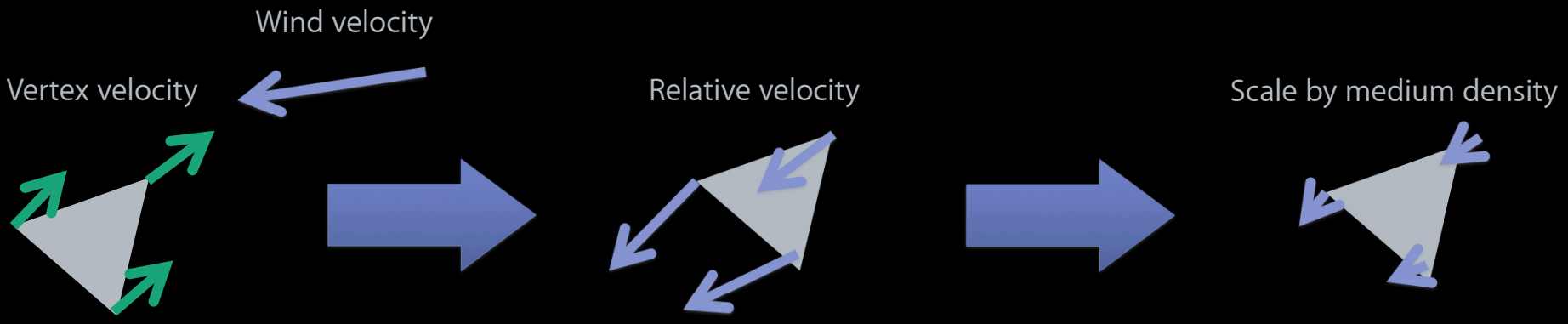
- Large number of particles
  - Appropriate for parallel processing
  - Force from each spring constraint applied to both connected particles



An Introduction to Cloth Simulation

Wind

- Wind interaction
  - Computed per-vertex
  - Uses force and density of air medium
  - Computes against normal and mass of vertex



Moving to GPU Acceleration

The CPU approach

AMD

- Iterative verlet integration over vertex positions
  - For each spring computes a force
  - Updates both vertices with a new position
  - Repeat n times where n is configurable

- Note that the computation is serial
  - Propagation of values through the solver is immediate

The CPU Approach

AMD

```
for each iteration
{
  for(int linkIndex = 0; linkIndex < numLinks; ++linkIndex)
  {
    float massLSC =
      (inverseMass0 + inverseMass1)/linearStiffnessCoefficient;
    float k = ((restLengthSquared – lengthSquared) /
      (massLSC * (restLengthSquared + lengthSquared) ) );

    vertexPosition0 –= length*(k*inverseMass0);
    vertexPosition1 += length *(k*inverseMass1);
  }
}
```

Moving to the GPU

Parallel execution

AMD

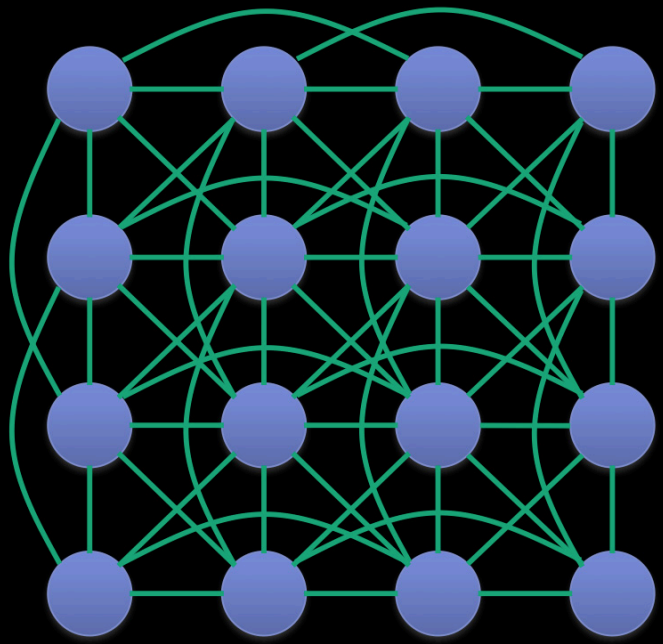
- The CPU implementation was serial
  - No atomicity issues
  - Value propagation immediate from a given update
- The GPU implementation is parallel within a cloth
  - Multiple updates to the same node create races

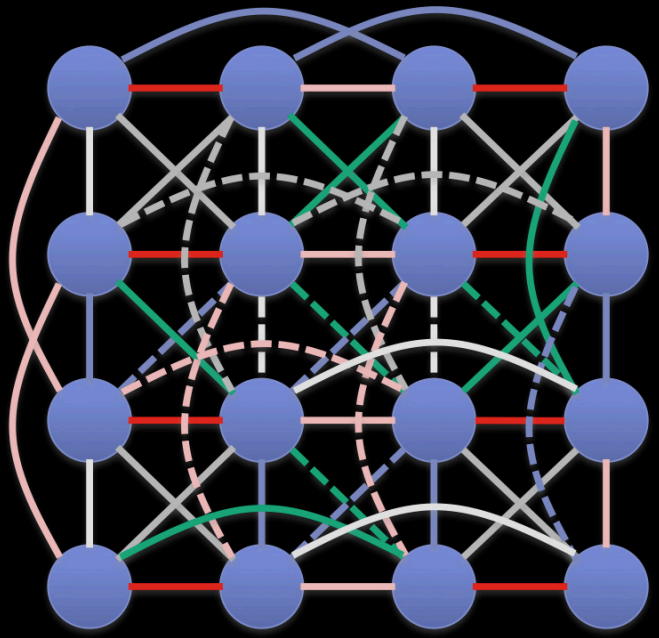
Moving to the GPU

Batching the simulation

AMD

- Create independent subsets of links through graph coloring
- Synchronize between batches





Driving Batches

AMD

```
for each iteration
{
  for( int i = 0; i < m_batchStartLengths.size(); ++i )
  {
    int start = m_linkData.m_batchStartLengths[i].first;
    int num   = m_linkData.m_batchStartLengths[i].second;
      for(int linkIndex = start;
          linkIndex < start + num;
          ++linkIndex)
    { … }
  }
}
```

Dispatching a Batch

AMD

```
cl::Kernel kernel =
 static_cast<const OpenCLDevice &>(m_device).getSolvePosFromLinksKernel();
cl::CommandQueue queue  =
  static_cast<const OpenCLDevice &>(m_device).getCLCommandQueue();
// Set resources and dispatch
kernel.setArg(0, startLink);
kernel.setArg(1, numLinks);
kernel.setArg(2, vertexIndicesForLinksSRV);
kernel.setArg(3, massSumLinearStiffnessCoefficientSRV);
kernel.setArg(4, restLengthSquaredSRV);
kernel.setArg(5, inverseMassSRV);
kernel.setArg(6, vertexPositions);
// Execute the kernel
queue->enqueueNDRangeKernel(
     kernel, cl::NullRange, cl::NDRange(numLinks), cl::NDRange(128) );
 …
```

Executing a Batch

AMD

```
__kernel void solvePositionsFromLinksKernel(
      const int startLink,
      const int numLinks,
      __global * int2  g_linksVertexIndices,
      __global * float g_linksMassLSC,
      __global * float g_linksRestLengthSquared,
      __global * float g_verticesInverseMass,
      __global * float4 g_verticesPositions)
   {
      int linkID = get_global_id(0) + startLink;
      if( get_global_id(0) < numLinks ) {
        float massLSC = g_linksMassLSC[linkID];
        float restLenSq = g_linksRestLengthSquared[linkID];
```

Executing a Batch

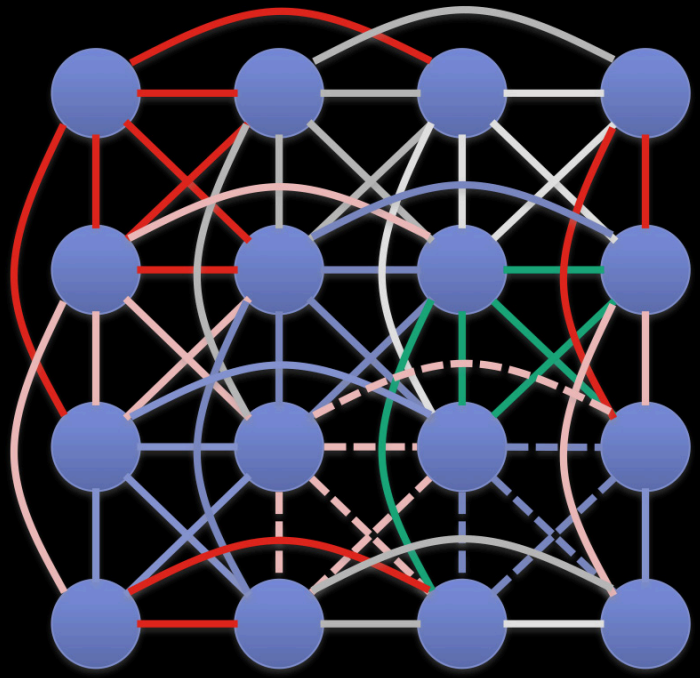
AMD

```
if( massLSC > 0.0f ) {
  int2 nodeIndices = g_linksVertexIndices[linkID];
  int node0 = nodeIndices.x; int node1 = nodeIndices.y;
  float3 position0 = g_vertexPositions[node0].xyz;
  float3 position1 = g_vertexPositions[node1].xyz;
  float inverseMass0 = g_verticesInverseMass[node0];
  float inverseMass1 = g_verticesInverseMass[node1];
  float3 del = position1 - position0;
  float len  = dot3(del, del);
  float k = ((restLenSq - len)/(massLSC*(restLenSq+len)));
  position0  = position0 - del*(k*inverseMass0);
  position1  = position1 + del*(k*inverseMass1);
  g_vertexPositions[node0] = (float4)(position0, 0.f);
  g_vertexPositions[node1] = (float4)(position1, 0.f);
```

Improving the Constraint Solver

Higher efficiency

- We saw the batched links earlier
  - Large number of batches needed
  - Low work density per-thread
- Can create larger batches
  - The cloth is fixed-structure
  - Can be preprocessed
- Fewer dispatches
  - Bottleneck in early version

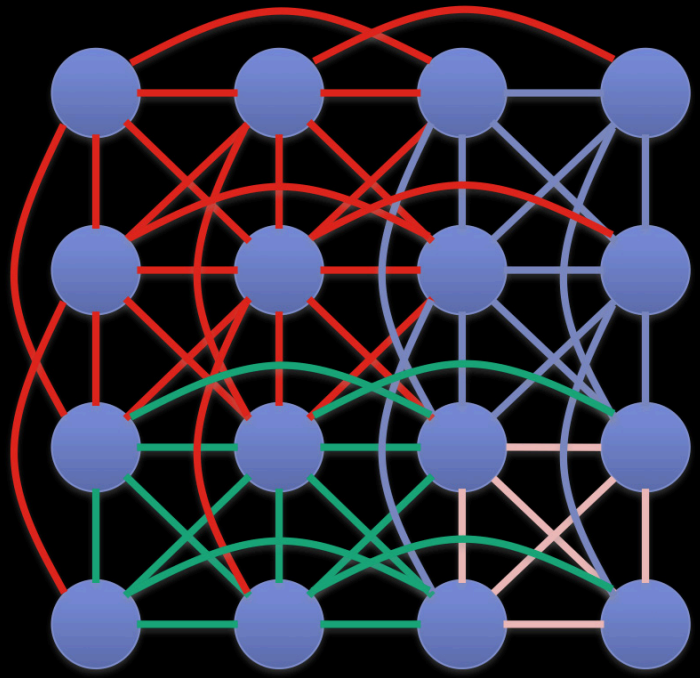


9 batches

Improving the Constraint Solver

Larger batches still

- We can move to much larger batches
  - Number of parallel instances reduced
  - On arbitrary meshes batches hard to create



4 batches

Improving the Constraint Solver

Remove determinism

- Large batches change behavior of solver
  - A lot of computation fed from previous iteration
  - In a serial implementation there is none
  - Propagation of updates is slower

- Relax a step further
  - Allow non-deterministic updates
  - Execute per-vertex and pull data, new or old, from neighbors link-by-link
  - No write-after-write hazard

Improving the Constraint Solver

Remove determinism

AMD

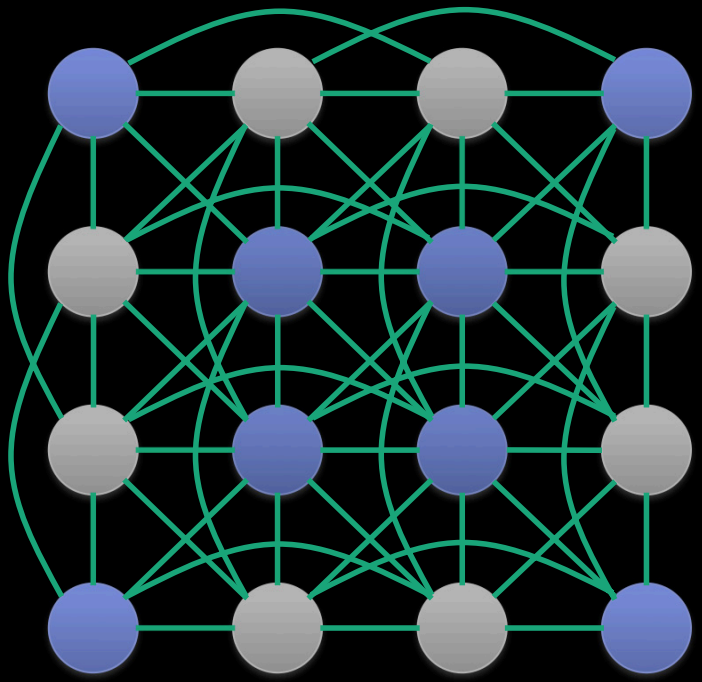
- Single batch
  - Highly efficient per solver iteration
  - Can re-use position data for central node
  - Slower convergence
  - Need to cleverly arrange data to allow efficient loop unrolling
- Scope for more iterations of solver to reduce effects

Improving the Constraint Solver

A branch divergence warning

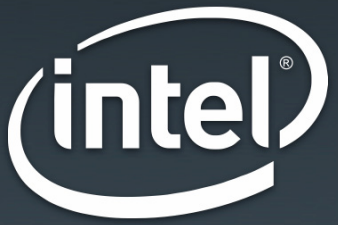
AMD

- The GPU is a collection of wide SIMD engines
  - Divergent branches hurt performance
  - Nodes have different degrees
  - Regular mesh
    - Low overhead
    - Similar degree throughout
  - Complicated mesh
    - Arbitrary numerous peaks
    - Pack vertices by degree



Demo

Writing OpenCL for Intel CPUs



Vinay Awasthi
Senior Software Engineer

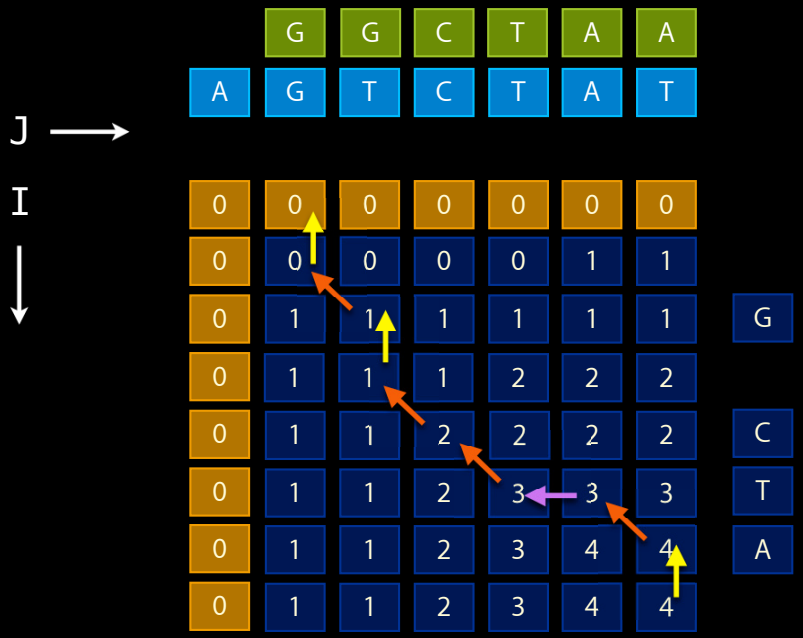
Goals of This Presentation

- Develop efficient OpenCL kernels for Intel CPUs
- Demonstrate CPU is excellent for complex, memory intensive algorithms
- Utilize multiple cores, vector data types, and large caches to hide data dependencies
- Case study
- Conclusions

Case Study

Longest common subsequence

- It is used to find the longest subsequence common to all sequences in a set of sequences
  - Commonly used to analyze DNA and protein sequence




```
If Data[i-1,j] == Data[i, j-1] (diag - Match)
    C[i,j] = C[i-1, j-1] + 1
else
    C[i,j]=MAX(C[i-1, j], C[i, j-1])
C[i,j] = C[i-1,j] (up - Insertion)
C[i,j] = C[i, j-1] (left - Deletion)
```

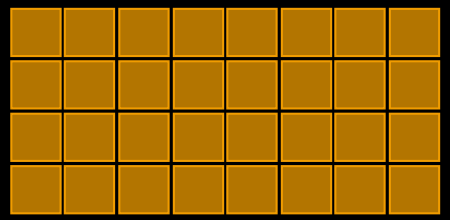
LCS = GCTA

Case Study

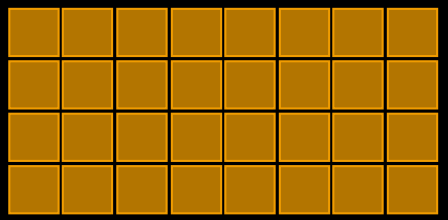
Wind

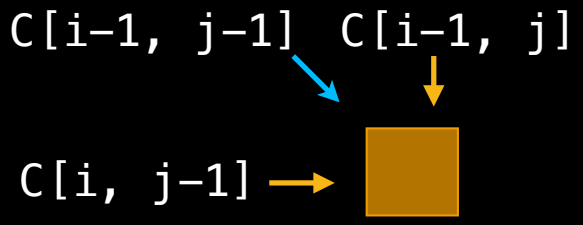
- Why is it interesting?
  - Dynamic programming is used extensively to solve hard problems
  - Method of solving complex combinational optimizations by decomposition and tabulation of intermediate results
  - Optimizations applied
    - Use of vector data types to take advantage of instruction level parallelism
    - Memory/cache optimizations
    - Multi-threading using OpenCL

Vector Data Type Optimizations



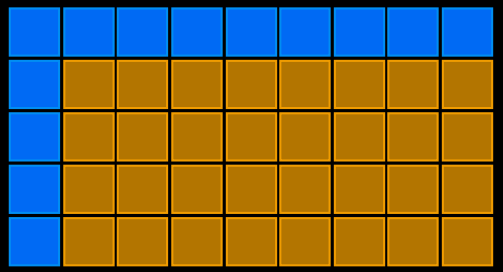
Vector Data Type Optimizations



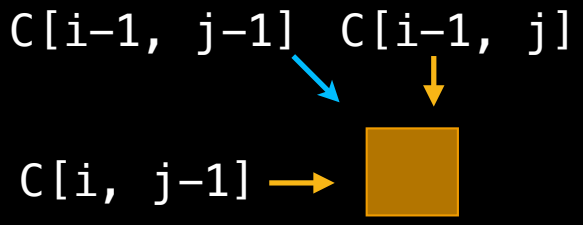



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

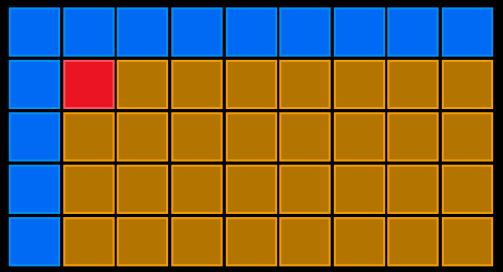


- Initialize with a Zero filled Band

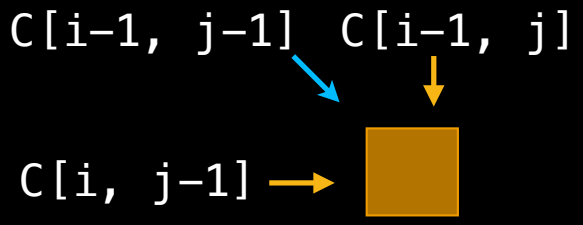



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

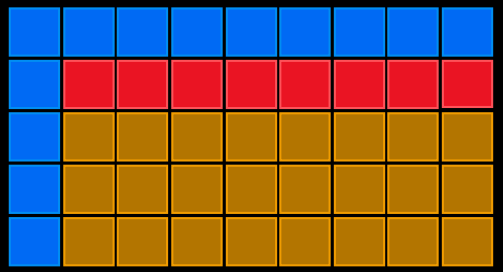


- Initialize with a Zero filled Band

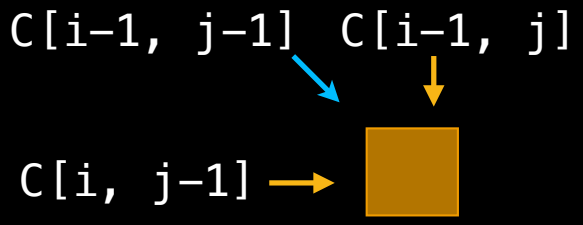



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

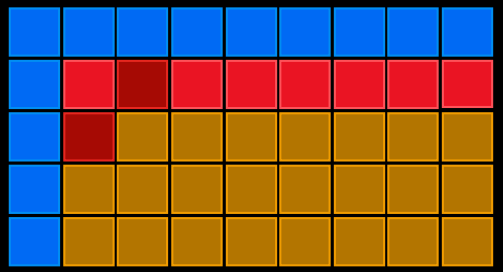


- Initialize with a Zero filled Band

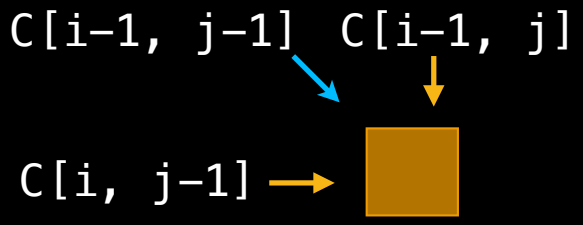



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

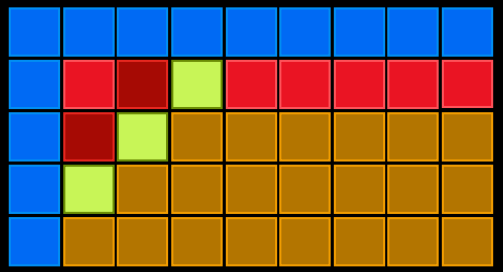


- Initialize with a Zero filled Band

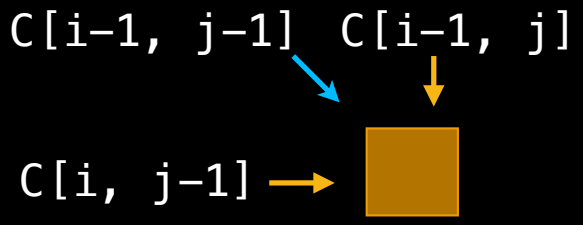



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

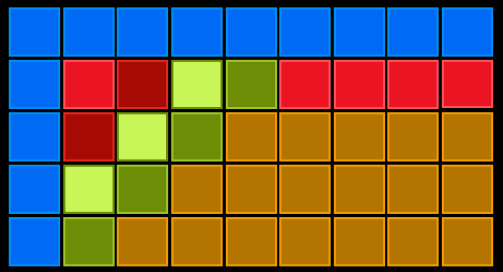


- Initialize with a Zero filled Band

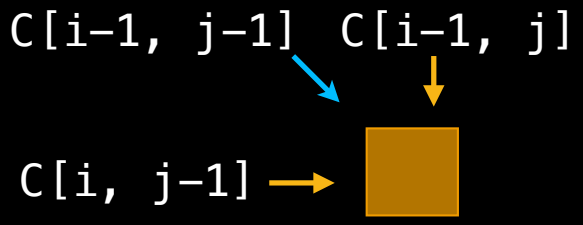



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

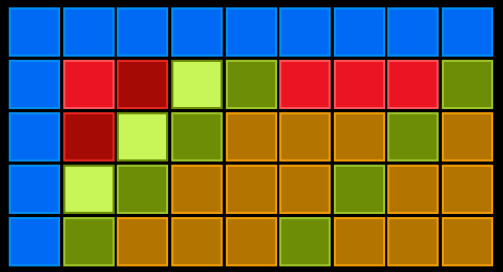


- Initialize with a Zero filled Band

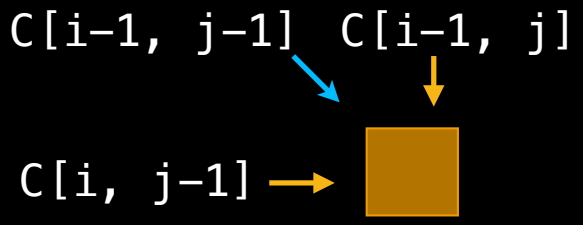



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

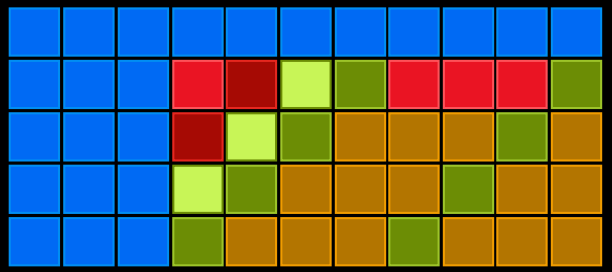


- Initialize with a Zero filled Band

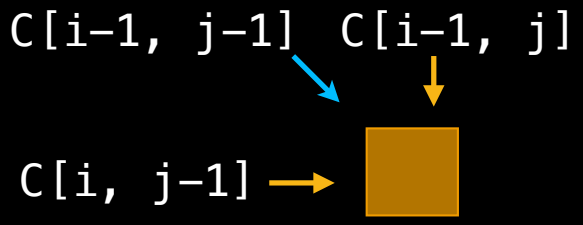



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

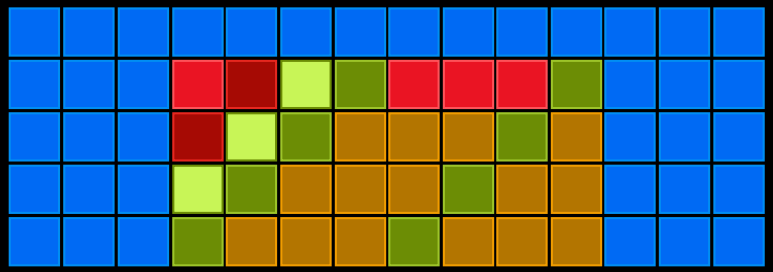


- Initialize with a Zero filled Band

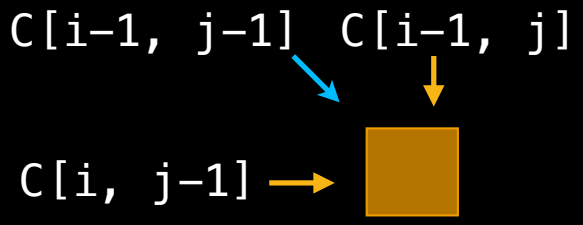



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

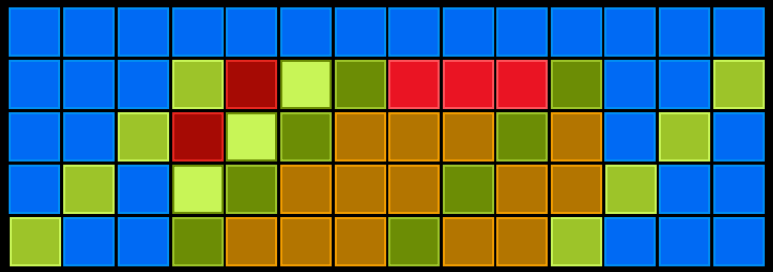


- Initialize with a Zero filled Band

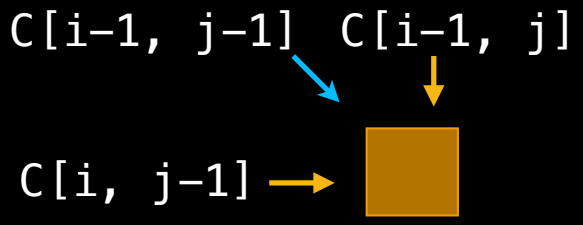



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

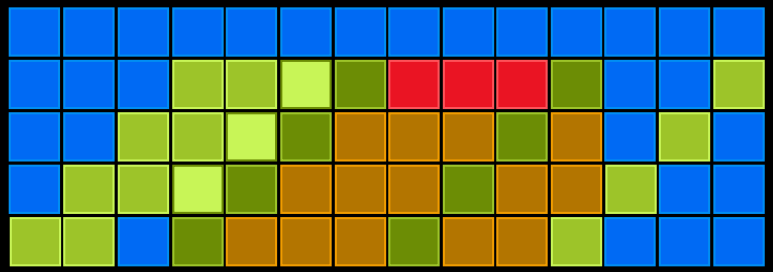


- Initialize with a Zero filled Band

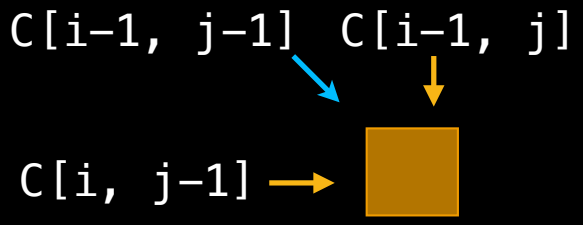



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

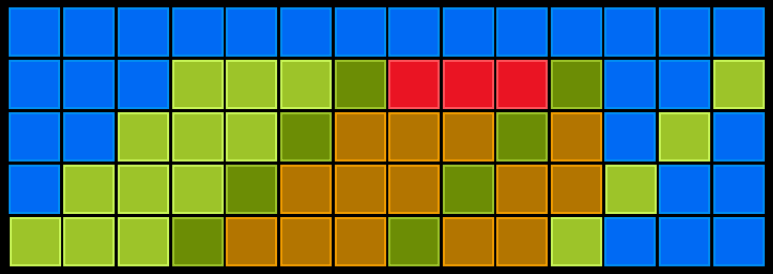


- Initialize with a Zero filled Band

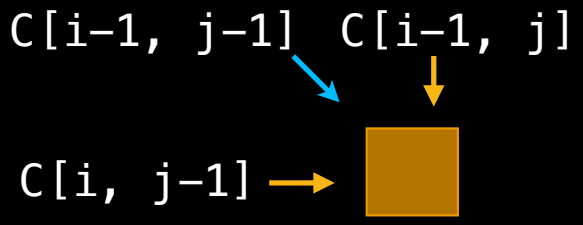



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

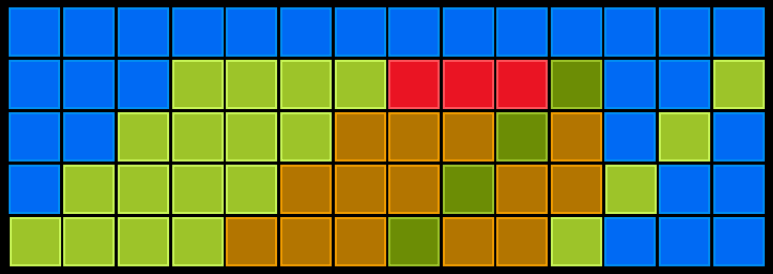


- Initialize with a Zero filled Band

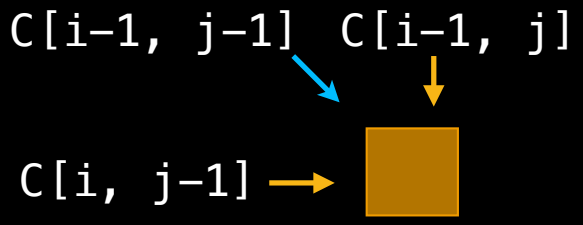



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

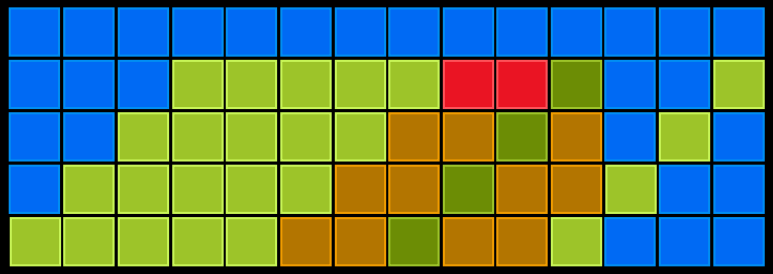


- Initialize with a Zero filled Band

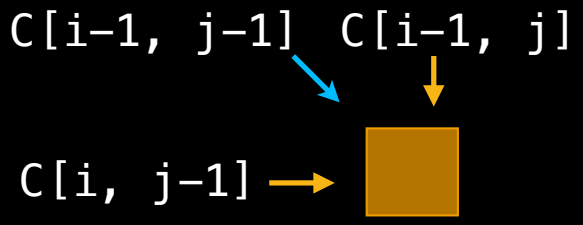



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

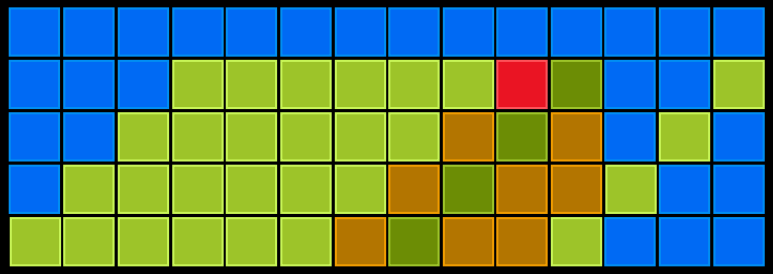


- Initialize with a Zero filled Band

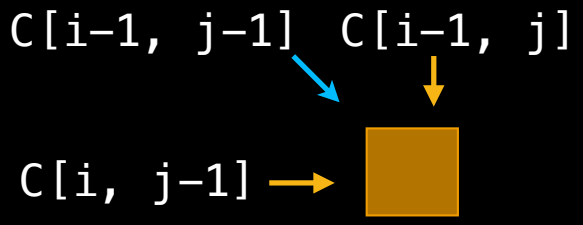



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

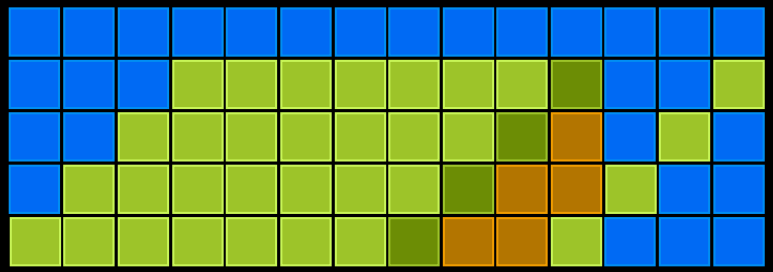


- Initialize with a Zero filled Band

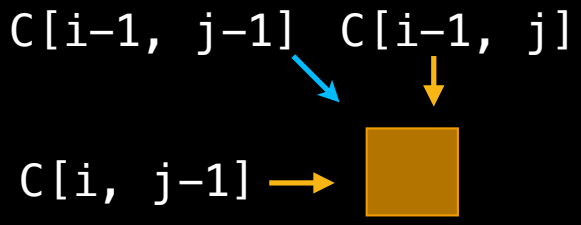



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

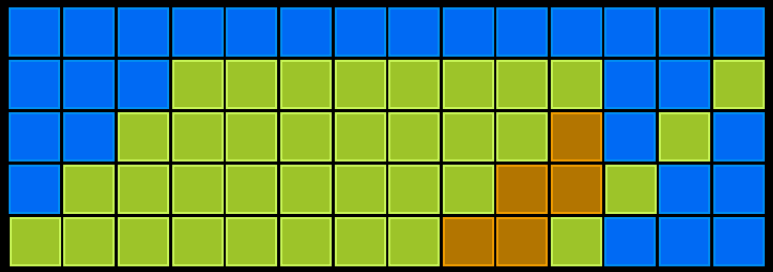


- Initialize with a Zero filled Band

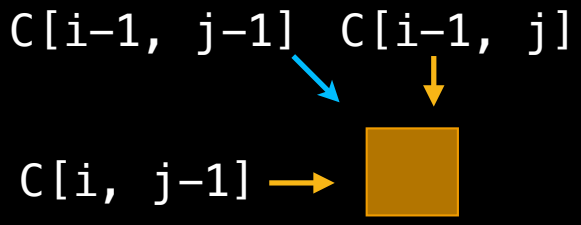



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

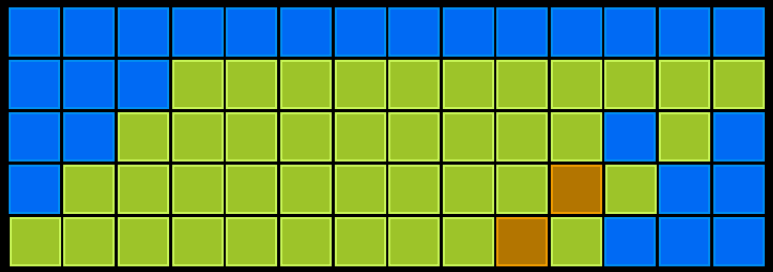


- Initialize with a Zero filled Band

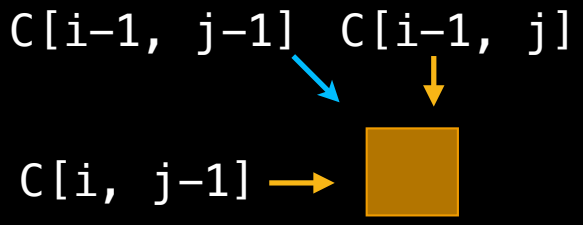



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

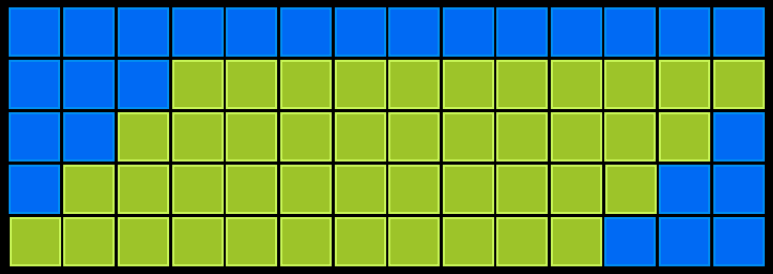


- Initialize with a Zero filled Band

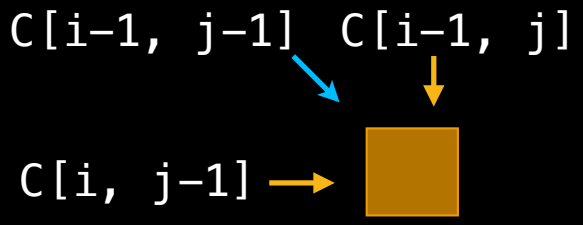



```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

Vector Data Type Optimizations

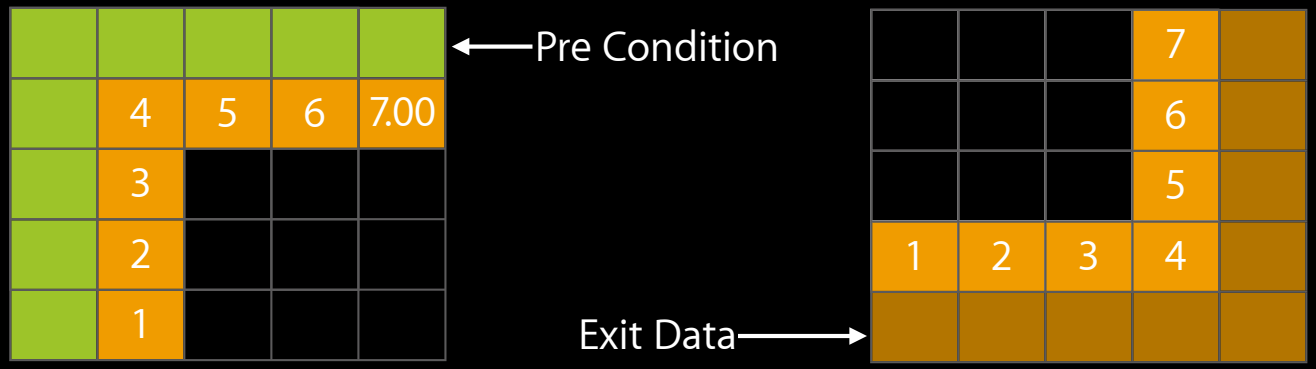


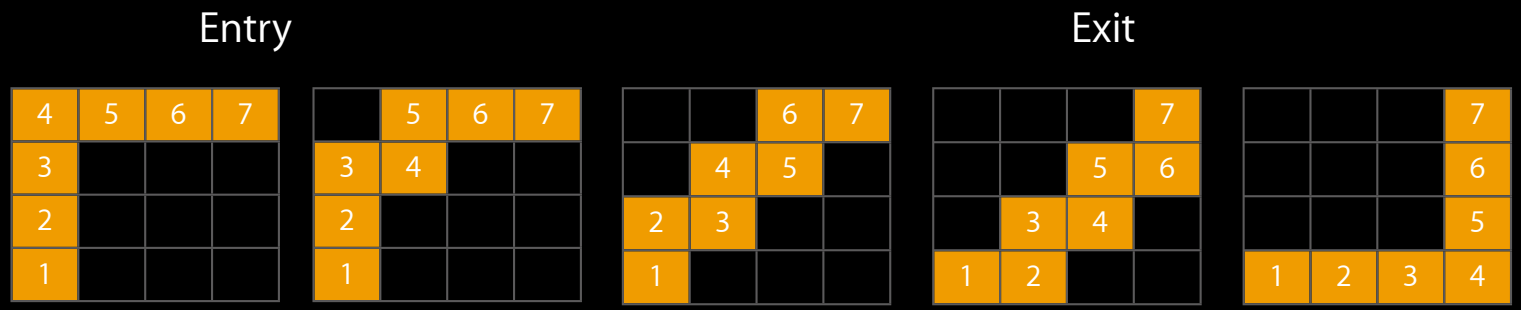
- Initialize with a Zero filled Band




```
If Data[i–1,j] == Data[i, j–1]
    C[i,j] = C[i–1, j–1] + 1
else
    C[i,j]=MAX(C[i–1, j], C[i, j–1])
```

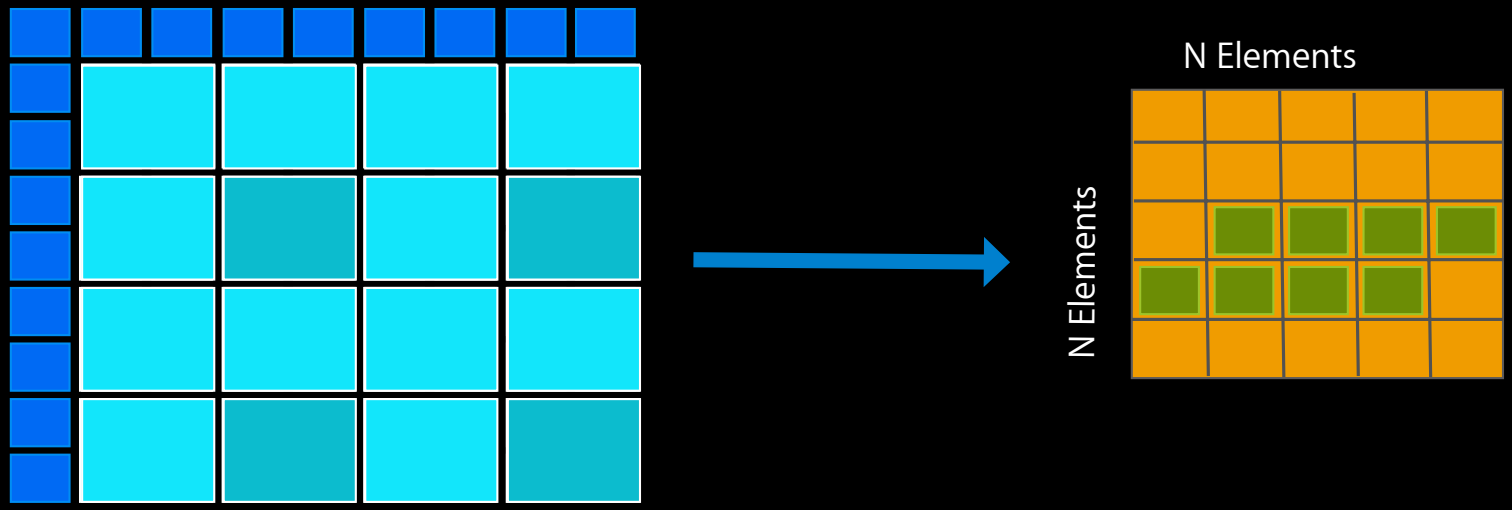
OpenCL Multithread Execution





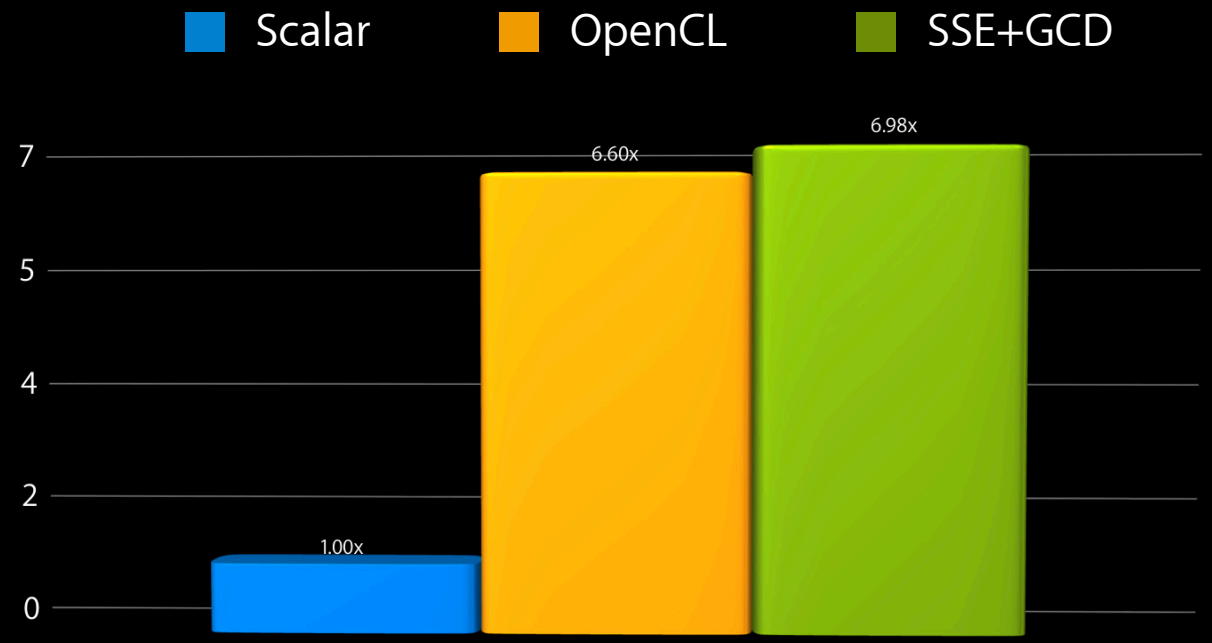
Number of Threads Executing in Parallel per Box

OpenCL Multithread Execution



(Cache Oblivious Algorithms) Execution Is Aware of Cache Hierarchy

Performance Results



Tests shown run on system with configuration as follows:
Mac OS X 10.6.3, 2.93 GHz Intel® Xeon® X5570 Processor, Intel Compiler 11.1.036

Lessons Learned

- Issue large work-items
  - Approximately 10,000–100,000 instructions
- Utilize map/unmap buffers
  - Only locks/unlocks the memory (light weight)
- Use CL_ALLOC_HOST_PTR for CPU memory
- Use command synchronizations such as clEnqueueWaitForEvents and barriers judiciously

Lessons Learned, continued

- Image read/write, channel/format support, sampler, etc.… Do not use any specialized logic/hw on CPUs
- Avoid handling special cases/edge conditions/boundary conditions in kernels (use padding)
- Use 1D Range if possible to take advantage of cache localities and avoid 2D–index calculations

```
__kernel  void  unrolled(const  __global  int*  data,  const  uint
dataSize)
{
    size_t  tid  =  get_global_id(0);
    size_t  gridSize  =  get_global_size(0);
    size_t  workPerItem  =  dataSize  /  gridSize;
    size_t  myStart  =  tid  *  workPerItem;
    for  (size_t  i  =  myStart;  i  <  myStart  +  workPerItem;  ++i)
    {
        //actual  work
    }
}
```

Avoid using such coding patterns

```
__kernel  void  Pythagorean(const  __global  f
                            const  __global  float*  b
                            __global  float*  c)
{
    size_t  tid  =  get_global_id(0);
    c[tid]  =  sqrt(a[tid]  *  a[tid]  +  b[tid]  *  b[tid]);
}
```

Use built-in function hypot

```
__kernel  void  exponentor(__global  int*  data,  const  uint
exponent)
{
    size_t  tid  =  get_global_id(0);
    int  base  =  data[tid];
    for  (int  i  =  1;;  i  <  exponent;;  ++i)
    {
        data[tid]  *=  base;
    }
}
```

Use global constant memory

```
__kernel  void  exponentor(__global  int*  data)
{
    size_t  tid  =  get_global_id(0);
    int  base  =  data[tid];
    for  (int  i  =  1;  i  <  EXPONENT;  ++i)
    {
        data[tid]  *=  base;
    }
}
```

Pass it as build configuration, compiler can then unroll loop

```
__kernel  __attribute__((vec_type_hint(float2)))
void  shift_by(__global  float2*  coords,  __global  float2*  deltas)
{
    uint  tid  =  get_global_id(0);
    coords[tid]  +=  deltas[tid];
}
```

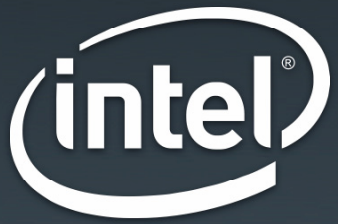
Use size_t, avoid down-casting

Use full length vectors, do more work

Kernel should now look like below

```
__kernel  __attribute__((vec_type_hint(float4)))
void  shift_by(__global  float2*  coords,  __global  float2*  deltas)
{
    size_t  tid  =  get_global_id(0);
    float4  my_coords  =  (float4)(coords[tid],  coords[tid  +  1]);
    float4  my_deltas  =  (float4)(deltas[tid],  deltas[tid  +  1]);
    my_coords  +=  my_deltas;
    vstore4(my_coords,  tid,  (__global  float*)coords);
}
```

Demo



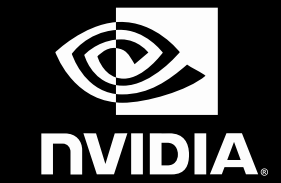
Conclusions

- OpenCL Framework allows you to harness the power of Intel CPUs
  - Intuitive, easy and maintainable
- OpenCL can help you create optimized code for the CPU
  - Almost on par with MT + SSE hand tuned code if coded optimally (SOA, larger work loads, use built-in functions)
    - Scales well across cores
    - Effectively utilizes SSE ISA
    - Well suited to serial parts using task parallelism
- Portable code
  - Across devices
  - Across device generations

Optimizing for GPUs with OpenCL

James Fung

Outline



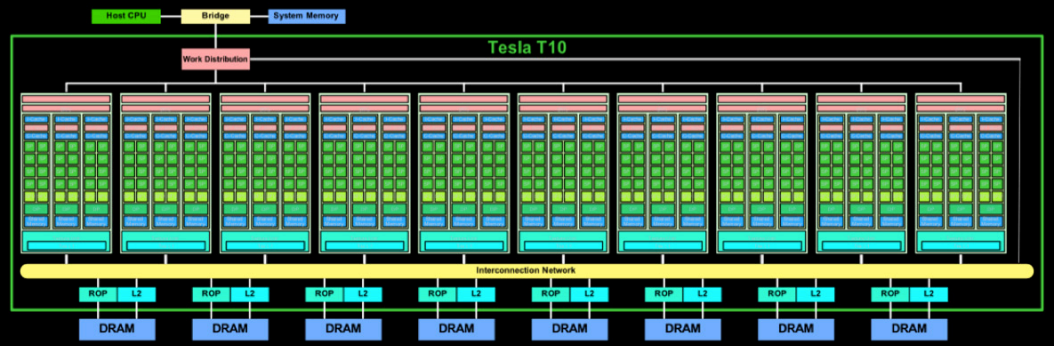
- Work group size heuristics: maximizing occupancy
- Instruction optimization: avoiding divergence
- Memory optimizations
  - Global memory coalescing
  - Local memory bank conflicts
- Using GPU texture hardware
  - Example: optical flow

Terminology Used

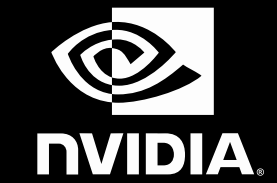
OpenCL	CUDA
Work Item	Thread
Work Group	Block

GPU Architecture (GTX285)

- 240 thread processors grouped into 30 streaming multiprocessors (SMs) @ 1.45 GHz with 4.0 GB of RAM
- 1 TFLOPS single precision (IEEE 754 precision)
- 87 GFLOPS double precision
- Each SM:
  - Eight thread processors
  - One double precision unit
  - 16 KB local memory, 16384 registers



Occupancy

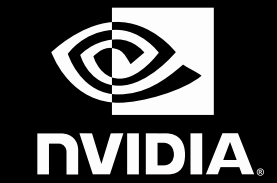


- With so many processors, its key to keep them all busy
- Work items (threads) are executed concurrently in "Warps" of 32 threads
- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

$$\text{Occupancy} = \frac{\text{\# of resident warps}}{\text{Max possible \# of resident warps}}$$

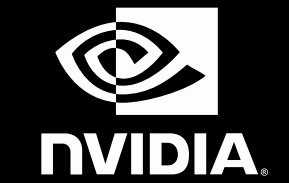
- Limited by resource usage:
  - Registers
  - Local memory

Measuring Occupancy



- Example: HW shader unit:
  - 8 work groups max
  - 32KB total local memory
  - 1024 work items max
- A work group size of 128 work items requiring 24KB of local memory
  - → only run one work group per shader unit (128 threads) BAD
- Check GPU documentation for details on HW

Global Workgroup Size Heuristics

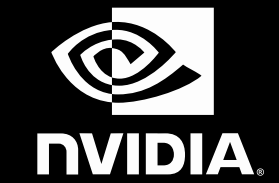


- # of workgroups > # of multiprocessors
  - So all multiprocessors have at least one workgroup to execute
- # of workgroups / # of multiprocessors > 2
  - Multiple workgroups can run concurrently in a multiprocessor
  - workgroups that aren't waiting at a barrier keep the hardware busy
  - Subject to resource availability—registers, local memory

Global Workgroup Size Heuristics, Cont.

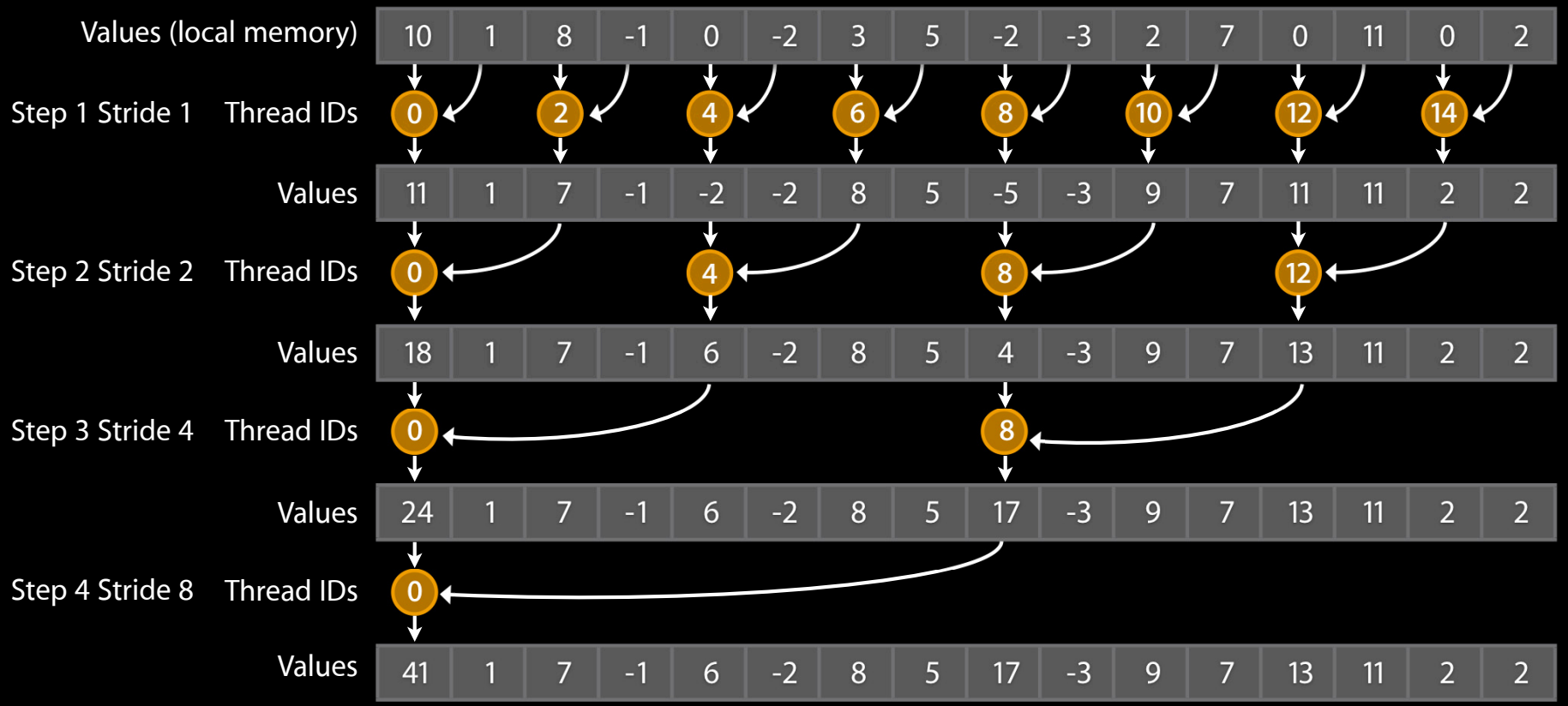
- # of workgroups > 100 to scale to larger devices
  - Workgroups executed in pipeline fashion
  - 1000 workgroups per kernel launch will scale across multiple generations
- # of work items/workgroup a multiple of warp size
  - So all threads in a warp are active

Control Flow: Divergence

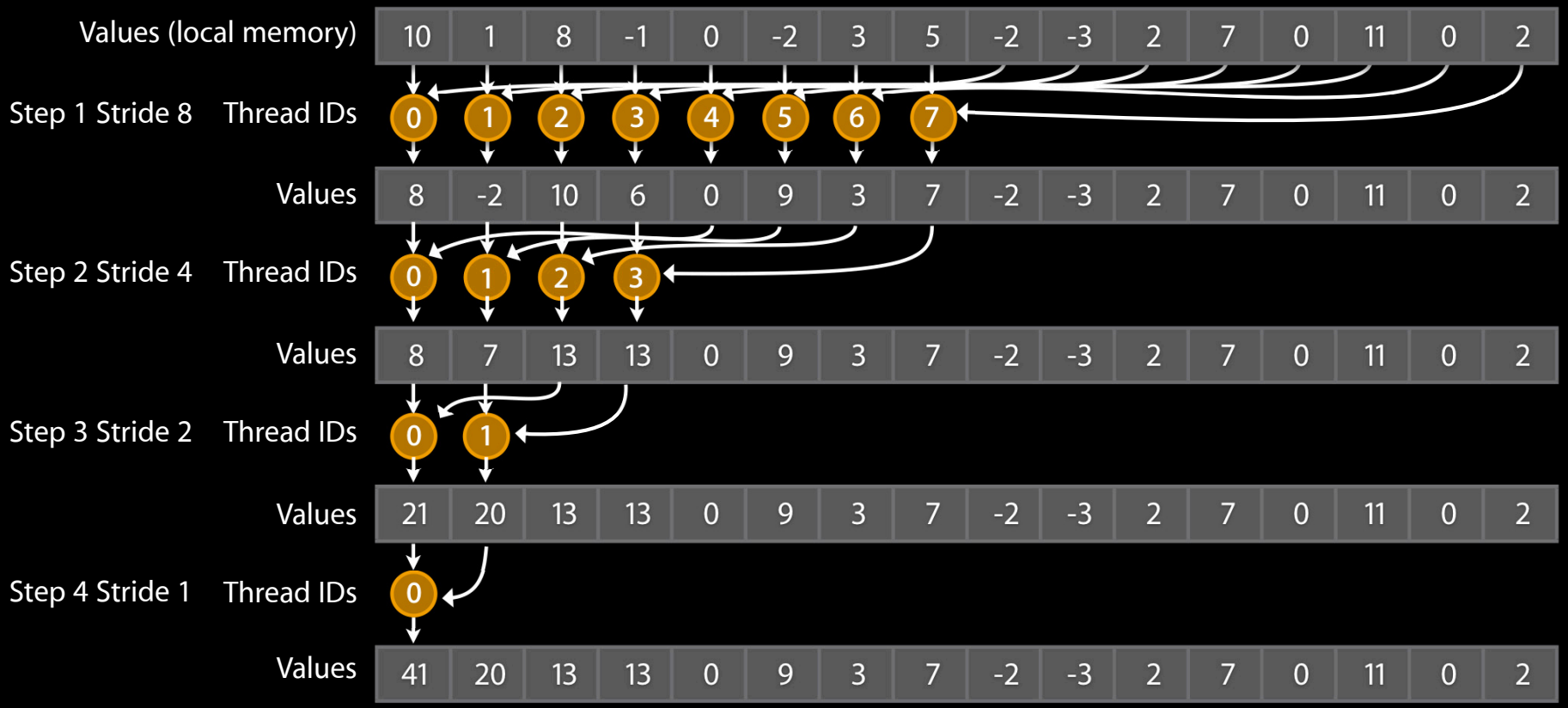


- Main performance concern with branching is divergence
- Work items within a single warp take different paths
- Different execution paths must be serialized
- Avoid divergence when branch condition is a function of work item ID
  - Example with divergence:
    - If (threadIdx.x > 2) { }
    - Branch granularity < warp size
  - Example without divergence:
    - If (threadIdx.x / WARP_SIZE > 2) { }
    - Branch granularity is a whole multiple of warp size

Parallel Reduction: Interleaved Addressing with Divergence (Poor Perf.)



Parallel Reduction: Sequential Addressing (Better Perf.)



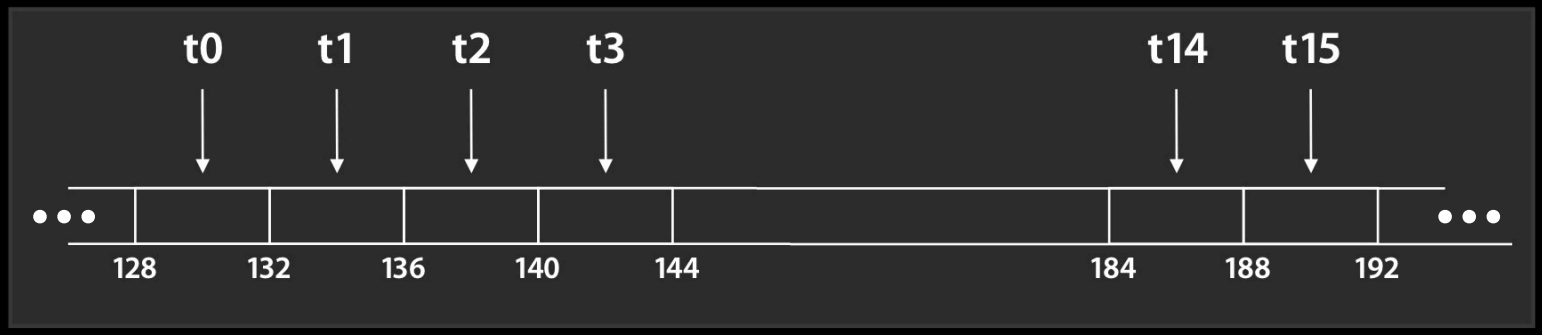
Fast Memory Access: Coalescing

- Compute 1.1
  - Applies to GeForce 8600/8800 GT, 9400M, 9600 GT
  - A coordinated read by a half-warp (16 threads)
  - A contiguous region of global memory:
    - 64 bytes — each thread reads a word: int, float, …
    - 128 bytes — each thread reads a double-word: int2, float2, …
    - 256 bytes — each thread reads a quad-word: int4, float4, …

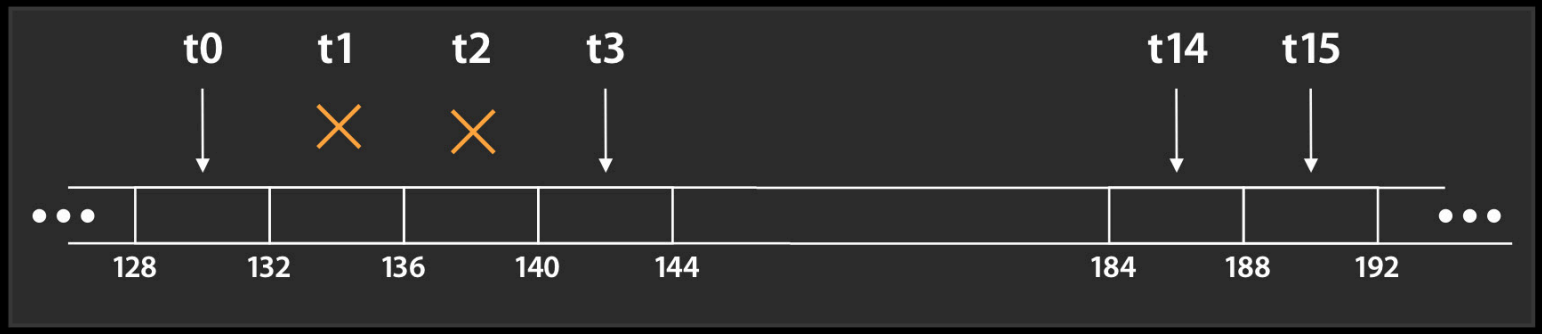
Fast Memory Access: Coalescing, Cont.

- Compute 1.1
  - Additional restrictions:
    - Starting address for a region must be a multiple of region size
    - The k^{th} thread in a half-warp must access the k^{th} element in a workgroup being read
  - Exception: not all threads must be participating
    - Predicated access, divergence within a halfwarp

Coalesced Access: Reading Floats

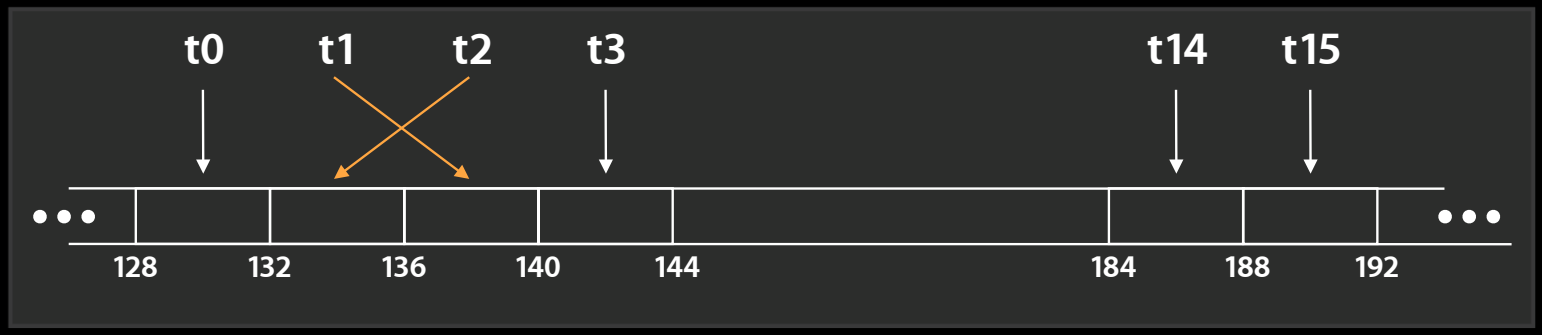


All Threads Participate

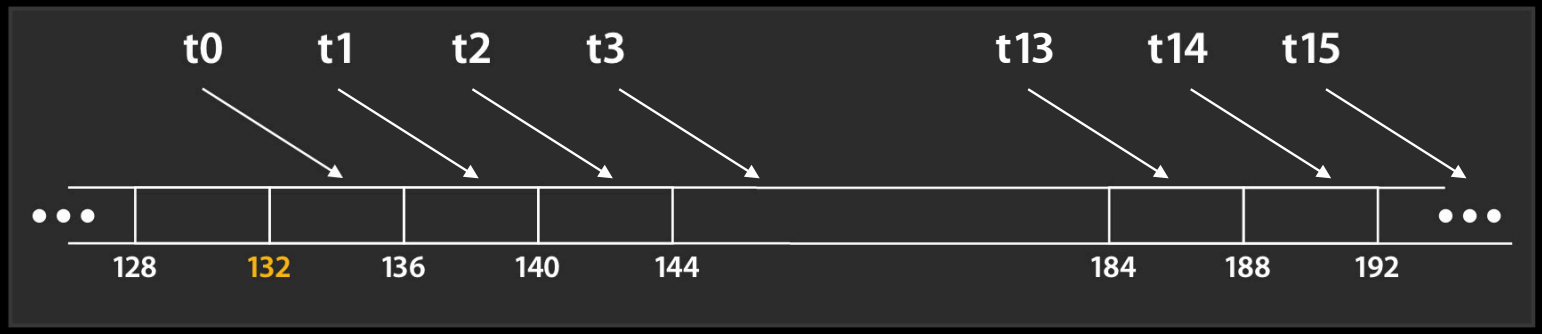


Some Threads Do Not Participate

Uncoalesced Access: Reading Floats



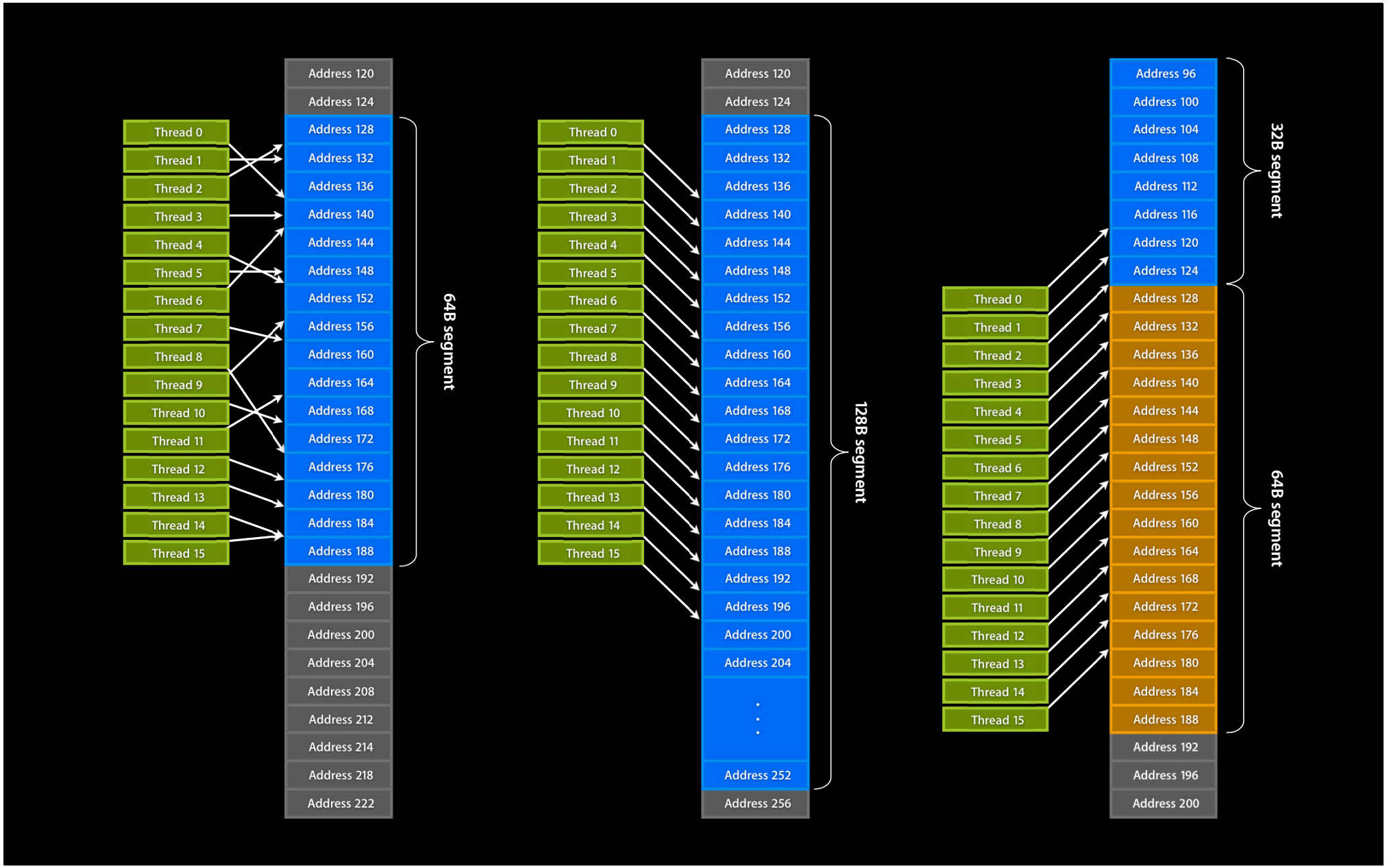
Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

Coalescing Compute >= 1.2

- Much improved coalescing capabilities in 10-series architecture
  - GeForce GTX285
  - GeForce 330M
- Hardware combines addresses within a half-warp into one or more aligned segments
  - 32, 64, or 128 bytes
- All threads with addresses within a segment are serviced with a single memory transaction
  - Regardless of ordering or alignment within the segment

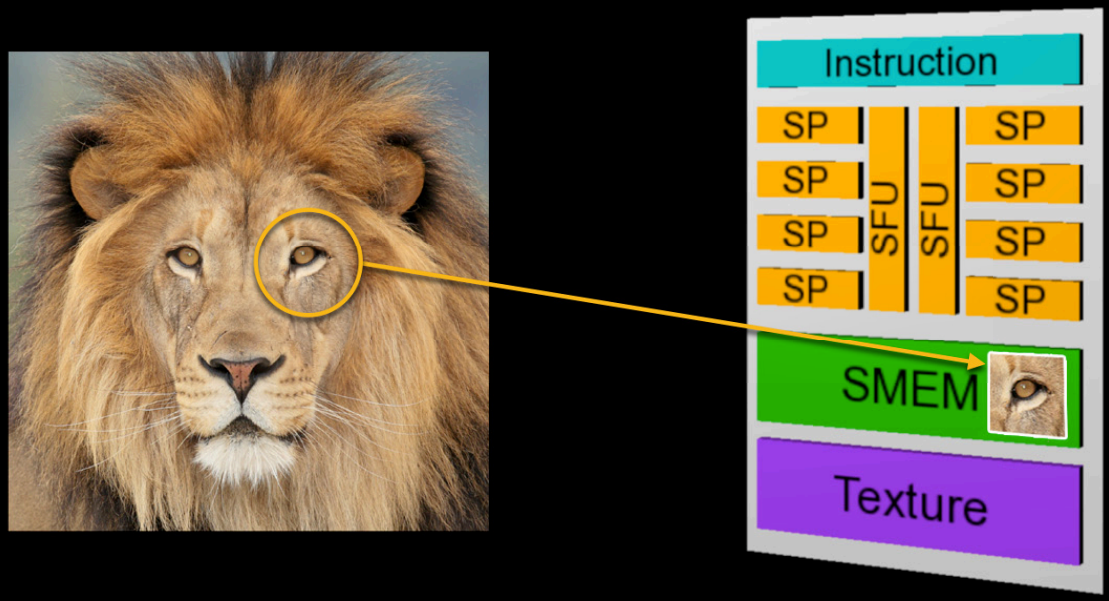

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 208
Address 212
Address 214
Address 218
Address 222
64B segment
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 252
Address 256
128B segment
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 96
Address 100
Address 104
Address 108
Address 112
Address 116
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
32B segment
64B segment

Coalescing: Timing Results (8800 GTX)

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads reading floats:
  - 356µs—coalesced
  - 357µs—coalesced, some threads don't participate
  - 3,494µs—permuted/misaligned thread access
- 4K blocks x 256 threads reading float[3] (e.g. RGB):
  - 3,302µs—float[3] uncoalesced
  - 359µs—float[3] coalesced through local memory

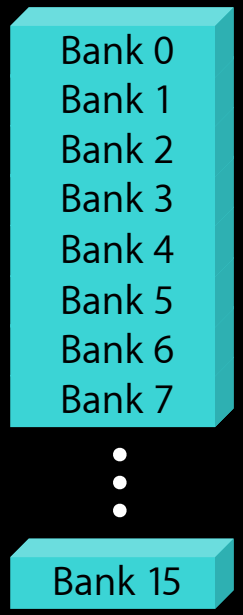
Use Local Memory

- Local memory
  - High speed, low latency on-chip memory
  - Load an image tile into local memory and share pixels between threads



Local Memory Architecture

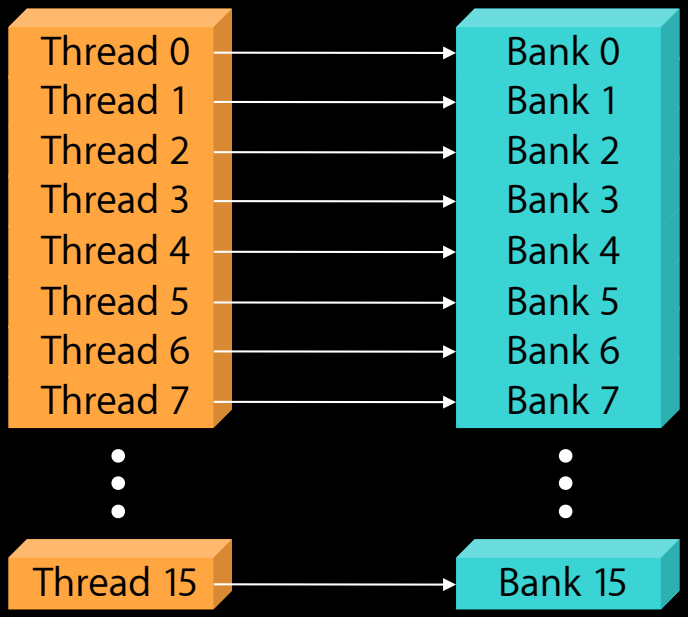
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Bank Addressing Examples

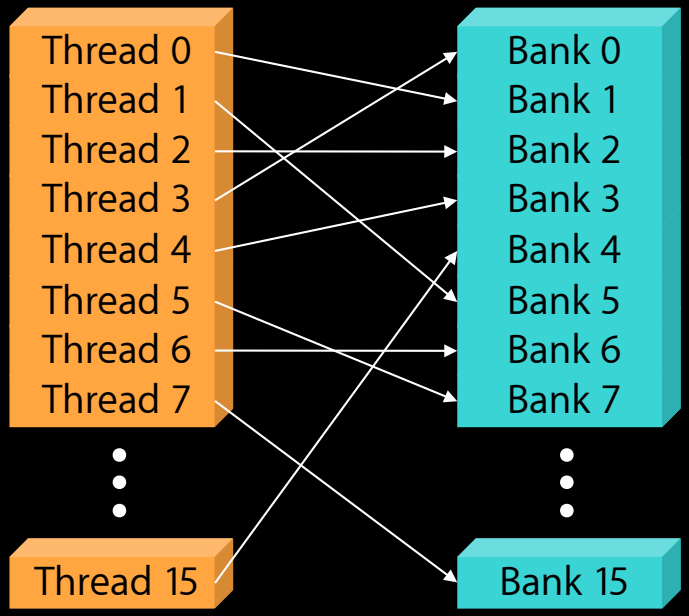
No Bank Conflicts

Linear addressing stride == 1



No Bank Conflicts

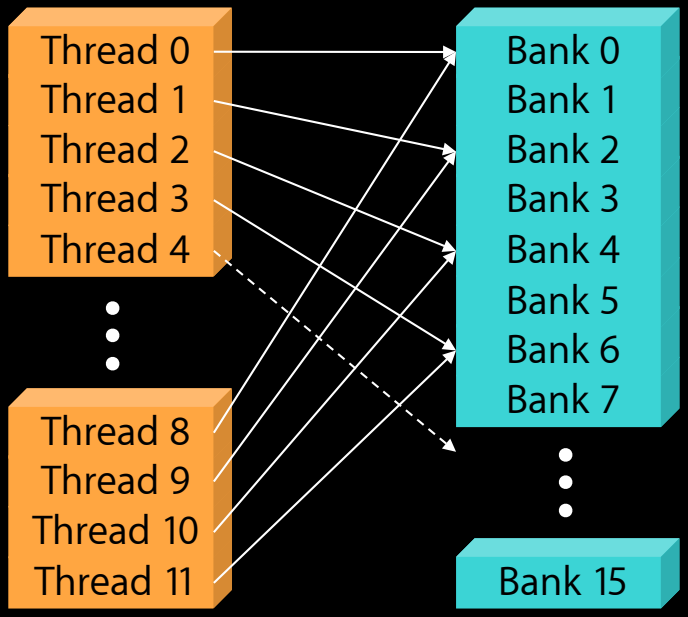
Random 1:1 Permutation



Bank Addressing Examples

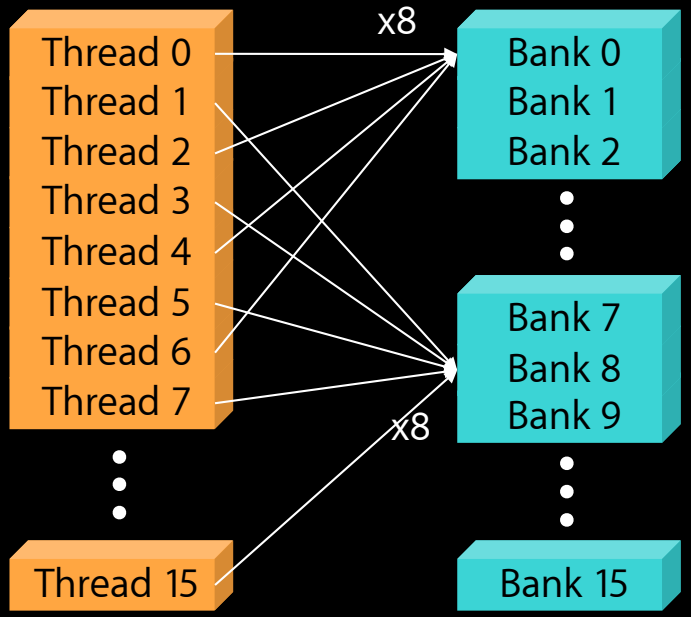
2-way Bank Conflicts

Linear addressing stride == 2



8-way Bank Conflicts

Linear addressing stride == 8



Applying Textures Optical Flow Example

- Optical Flow calculation motion of points in image pairs
- Embarrassingly parallel, compute intensive
- Applications:
  - Image stabilization, feature tracking, video encoding
- Optical flow makes use of:
  - Texture cache
  - Texture hardware bilinear interpolation

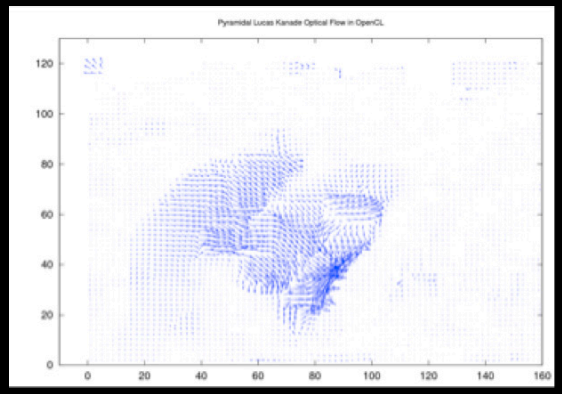
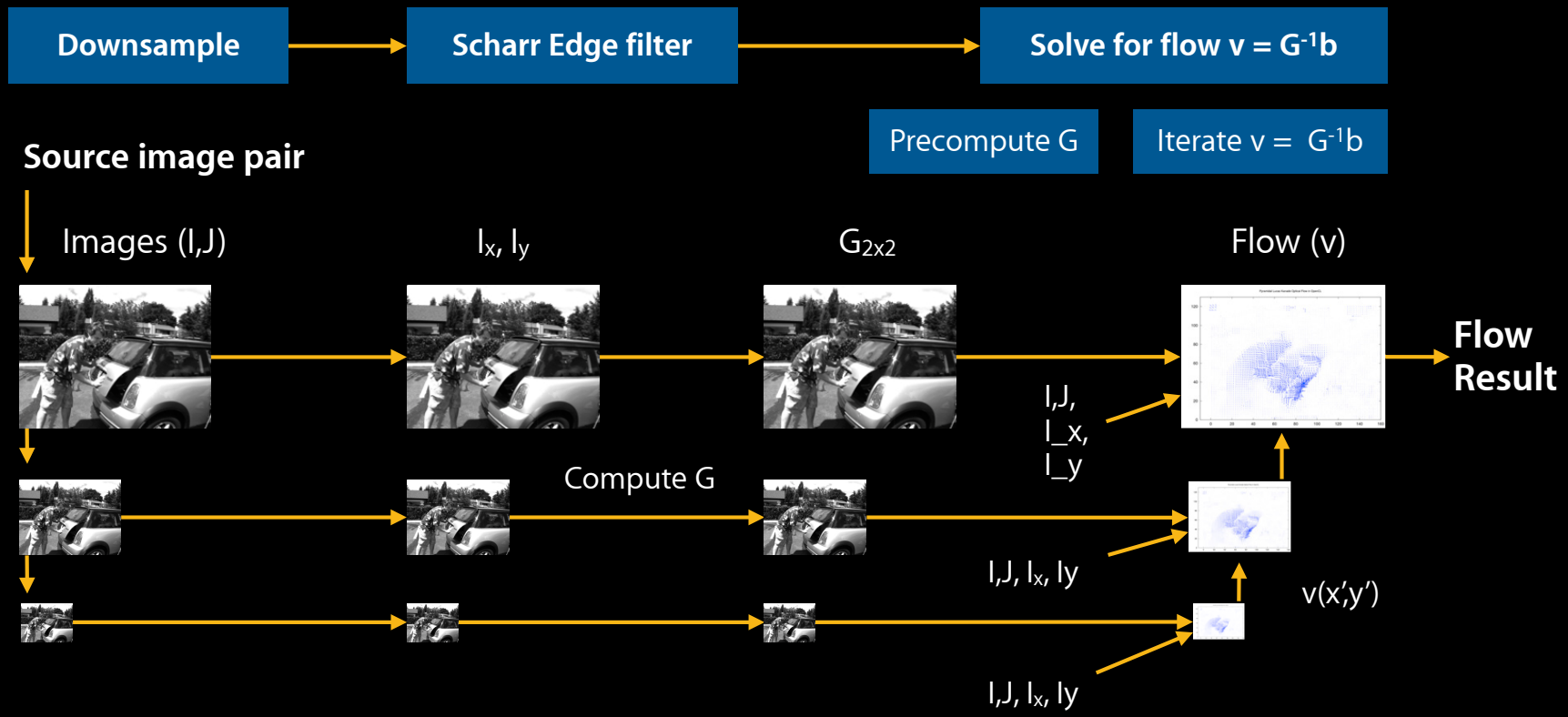


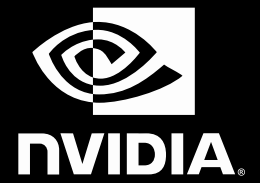
Image Sequence: Middlebury "Minicooper" sequence, frames 10-11

Architecture

Algorithm: Pyramidal Lucas Kanade Optical Flow



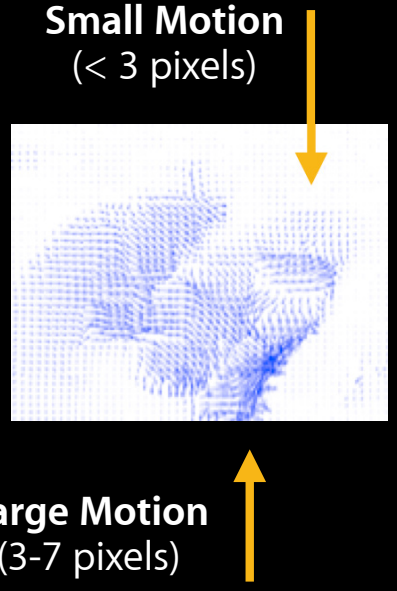
Texture Cache



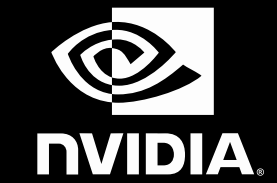
- Solving for flow $v = G^{-1}b$ is an iterative algorithm

Estimate motion $\boldsymbol{v} = \boldsymbol{G}^{-1}\,\boldsymbol{b}(x',y')$
Update position $(x',y') = (x'+v_x, y'+v_y)$
Repeat N iterations or until convergence

- Different areas have different amount of motion, but spatially coherent
  - Lookup window offset varies inside the image
  - Texturing hardware manages caches for you!



Hardware Interpolation



- Sub-pixel accuracy and sampling is crucial
  - Between iterations, x,y is non-integer

$$\partial I_k(x,y) = I^L(x,y) - J^L(x + g_x^L + v_x^L, y + g_y^L + v_y^L)$$

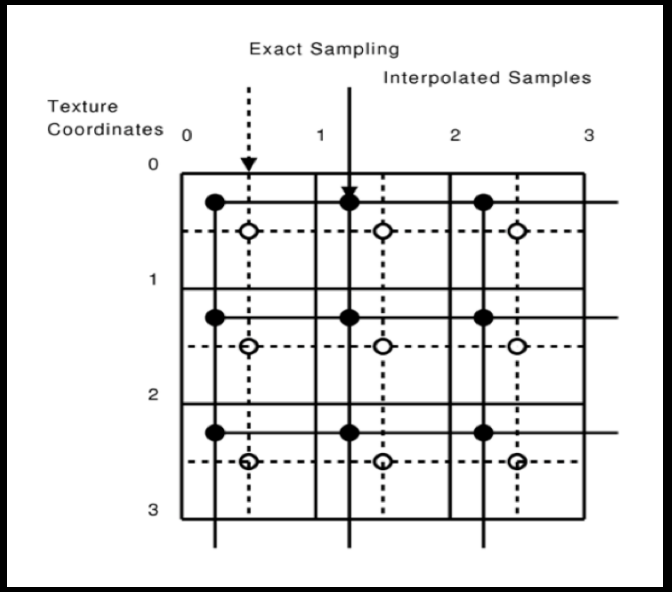
$$b_k = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \partial I_k(x,y) I_x(x,y) \\ \partial I_k(x,y) I_y(x,y) \end{bmatrix}$$

image level L
pixel centre p
window w,
guess g from previous level L+1

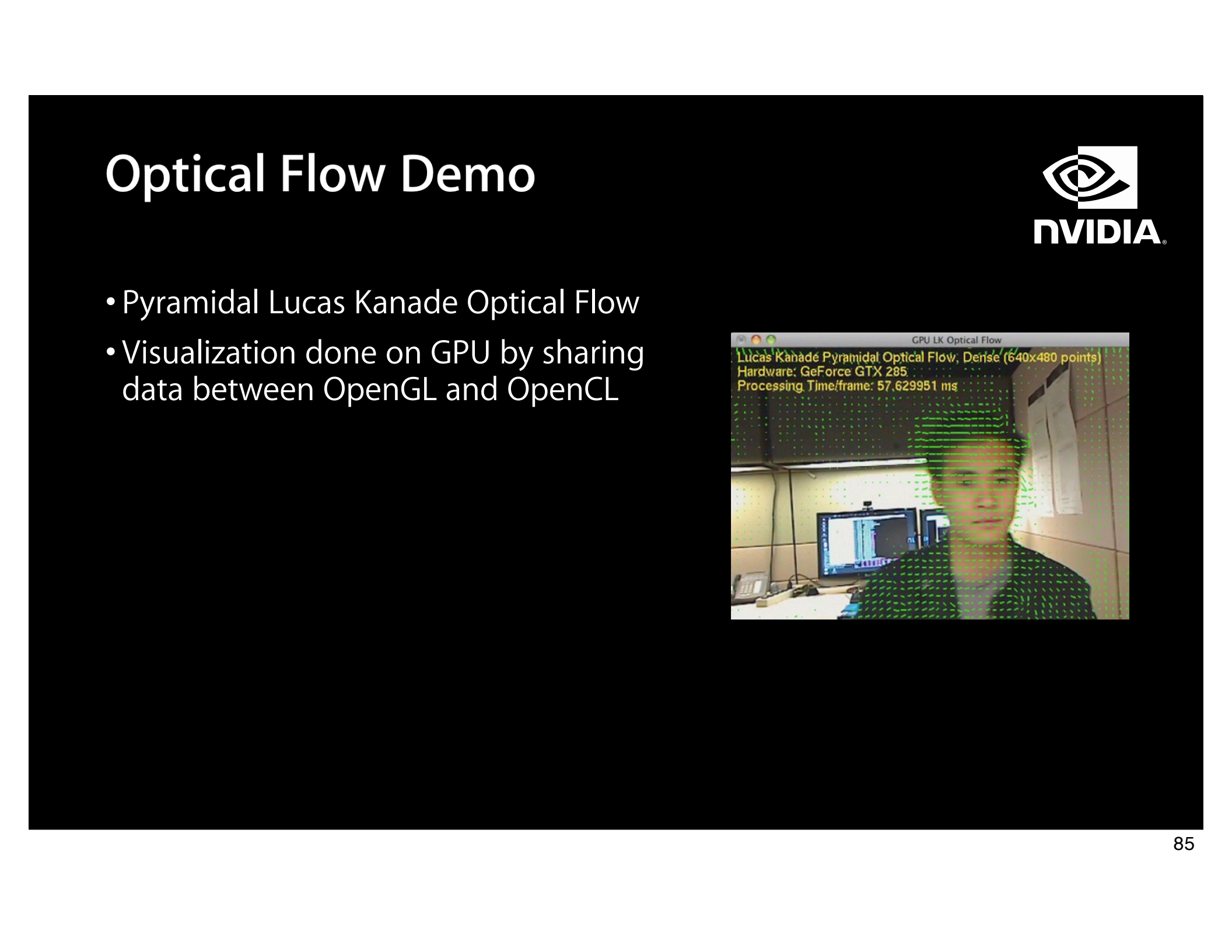
Hardware Interpolation

- Use Texture Hardware Linear Interpolation

```
sampler_t bilinSample =
    ... | CLK_FILTER_LINEAR;
…
float Jsample = read_imagef( J_float,
    bilinSampler, Jidx+(float2)(i,j) ).x;
…
```

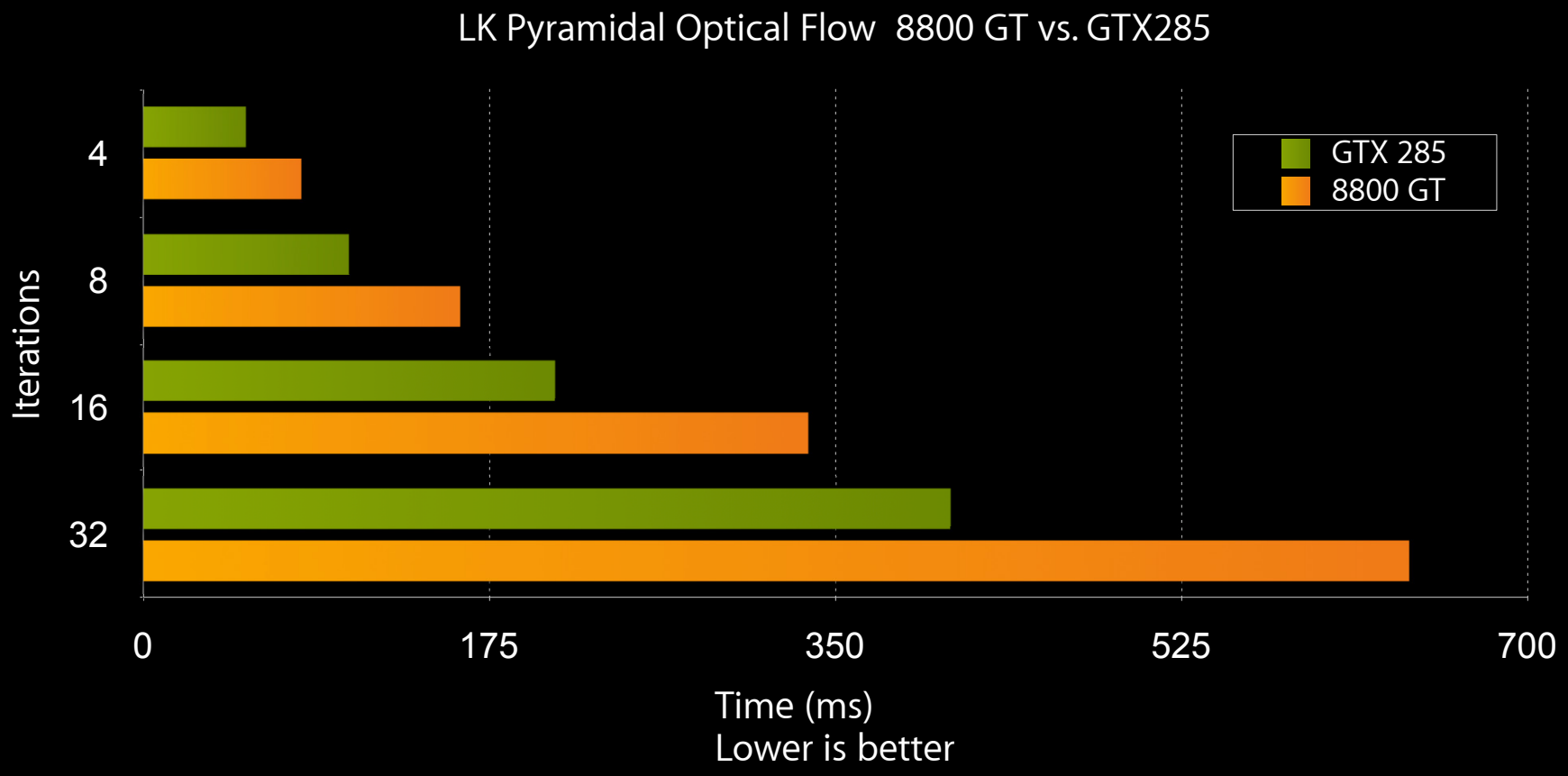


Optical Flow Demo



- Pyramidal Lucas Kanade Optical Flow
- Visualization done on GPU by sharing data between OpenGL and OpenCL

Optical Flow Performance



More Information

Allan Schaffer
Graphics and Game Technologies Evangelist
aschaffer@apple.com

Apple Developer Forums
http://devforums.apple.com

Labs

OpenCL Lab	Location Graphics & Media Lab C Thursday 9:00AM


WWDC10