OpenCL Programming & Optimization Case Study

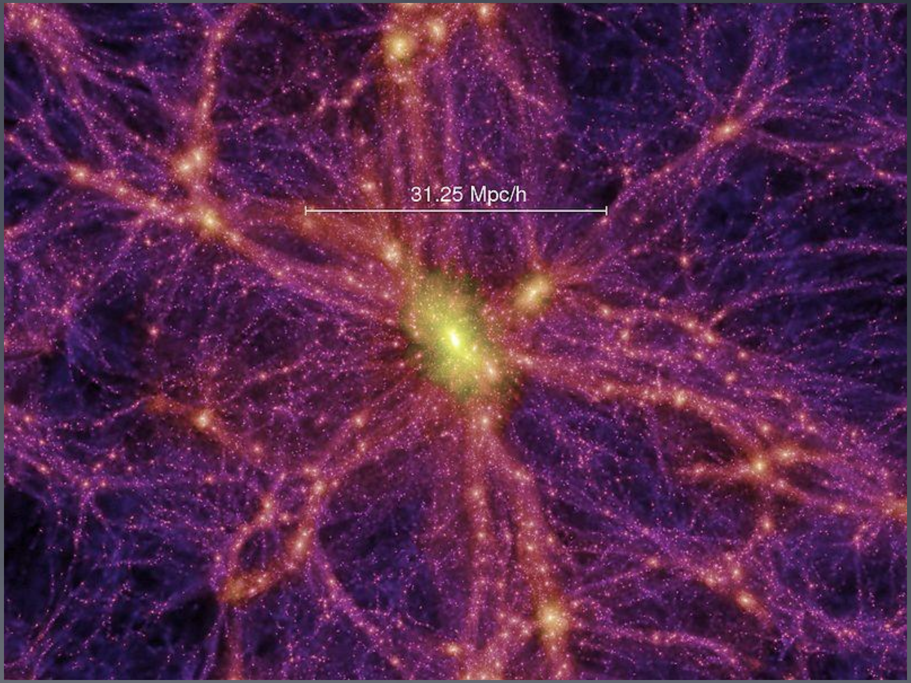
A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

Instructor Notes

- Lecture discusses parallel implementation of a simple embarrassingly parallel nbody algorithm
  - We aim to provide some correspondence between the architectural techniques discussed in the optimization lectures and a real scientific computation algorithm
  - We implement a N-body formulation of gravitational attraction between bodies
- Two possible optimizations on a naïve nbody implementation
  - Data reuse vs. non data reuse
  - Unrolling of inner loop
- Simple example based on the AMD Nbody SDK example
- Example provided allows user to:
  - Change data size to study scaling
  - Switch data reuse on or off
  - Change unrolling factor of n-body loop
  - Run example without parameters for help on how to run the example and define optimizations to be applied to OpenCL kernel

Topics

- N-body simulation algorithm on GPUs in OpenCL
- Algorithm, implementation
- Optimization Spaces
- Performance



The Millennium Run simulates the universe until the present state, where structures are abundant, manifesting themselves as stars, galaxies and clusters

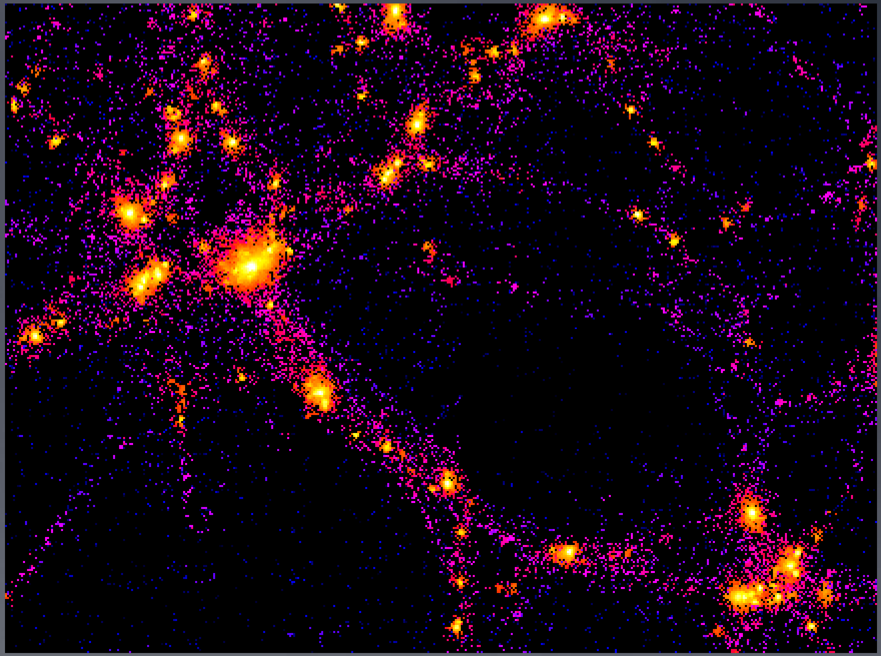
Source: http://en.wikipedia.org/wiki/N-body_simulation

Motivation

- We know how to optimize a program in OpenCL by taking advantage of the underlying architecture
  - We have seen how to utilize threads to hide latency
  - We have also seen how to take advantage of the different memory spaces available in today's GPUs.
- How do we apply this to an real world useful scientific computing algorithm?
- Our aim is to use the architecture specific optimizations discussed in previous lectures

N-body Simulation

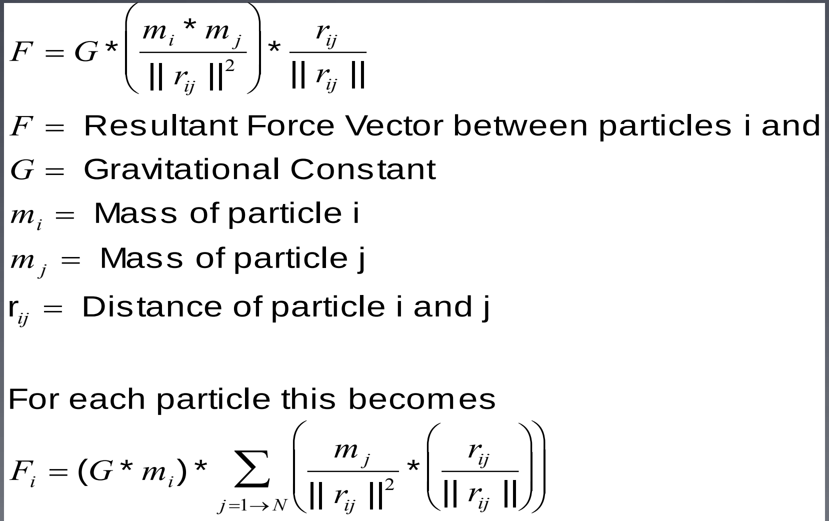
- An n-body simulation is a simulation of a system of particles under the influence of physical forces like gravity
  - E.g.: An astrophysical system where a particle represents a galaxy or an individual star
- N^2 particle-particle interactions
  - Simple, highly data parallel algorithm
- Allows us to explore optimizations of both the algorithm and its implementation on a platform



Source: THE GALAXY-CLUSTER-SUPERCLUSTER CONNECTION
http://www.casca.ca/ecass/issues/1997-DS/West/west-bil.html

Algorithm

- The gravitational attraction between two bodies in space is an example of an N-body problem
  - Each body represents a galaxy or an individual star, and bodies attract each other through gravitational force
- Any two bodies attract each other through gravitational forces (F)



$$F = G * \left(\frac{m_i * m_j}{\| r_{ij} \|^2} \right) * \frac{r_{ij}}{\| r_{ij} \|}$$

$F =$ Resultant Force Vector between particles i and

$G =$ Gravitational Constant

$m_i =$ Mass of particle i

$m_j =$ Mass of particle j

$\mathrm{r}_{ij} =$ Distance of particle i and j

For each particle this becomes

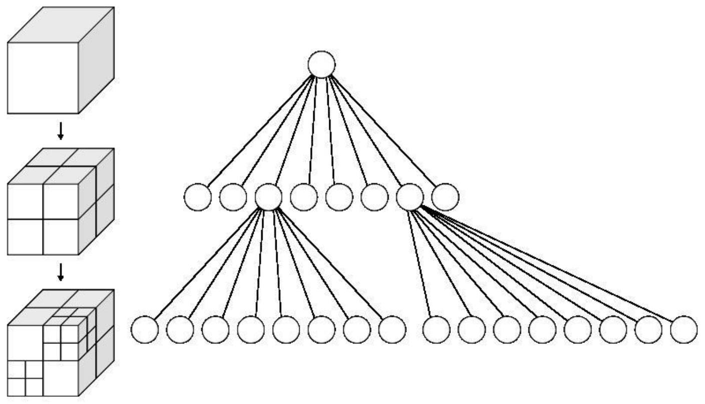
$$F_i = (G * m_i) * \sum_{j=1 \to N} \left(\frac{m_j}{\| r_{ij} \|^2} * \left(\frac{r_{ij}}{\| r_{ij} \|} \right) \right)$$

- An O(N^2) algorithm since N*N interactions need to be calculated
- This method is known as an all-pairs N-body simulation

A Collaboration Between David Kaeli, Northeastern University
and Benedict R. Gaster, AMD © 2011

N-body Algorithms

- For large counts, the previous method calculates of force contribution of distant particles
  - Distant particles hardly affect resultant force
- Algorithms like Barnes Hut reduce number of particle interactions calculated
  - Volume divided into cubic cells in an octree
  - Only particles from nearby cells need to be treated individually
  - Particles in distant cells treated as a single large particle
- In this lecture we restrict ourselves to a simple all pair simulation of particles with gravitational forces



- A octree is a tree where a node has exactly 8 children
- Used to subdivide a 3D space

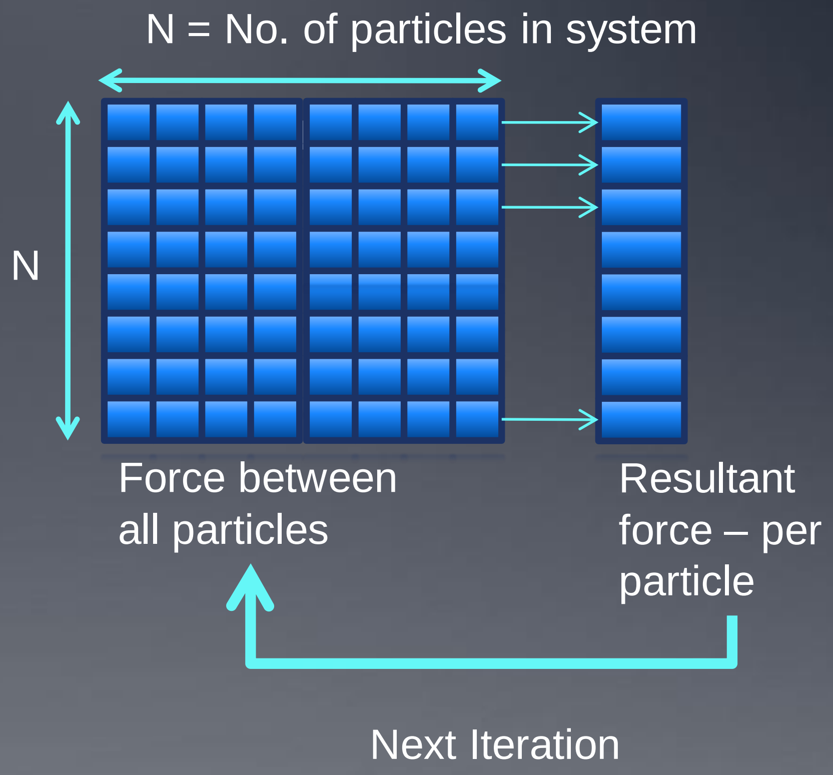
Basic Implementation – All pairs

- All-pairs technique is used to calculate close-field forces
- Why bother, if infeasible for large particle counts ?
  - Algorithms like Barnes Hut calculate far field forces using near-field results
  - Near field still uses all pairs
  - So, implementing all pairs improves performance of both near and far field calculations
- Easy serial algorithm
  - Calculate force by each particle
  - Accumulate of force and displacement in result vector

```
for(i=0; i<n; i++)
{
    ax = ay = az = 0;
   // Loop over all particles "j”
    for (j=0; j<n; j++)  {

        //Calculate Displacement
        dx=x[j]-x[i];
        dy=y[j]-y[i];
        dz=z[j]-z[i];

        // small eps is delta added for dx,dy,dz
= 0
        invr= 1.0/sqrt(dx*dx+dy*dy+dz*dz
+eps);
        invr3 = invr*invr*invr;
        f=m[ j ]*invr3;

        // Accumulate acceleration
        ax += f*dx;
        ay += f*dy;
        az += f*dx;
    }
   // Use ax, ay, az to update particle
positions
}
```

Parallel Implementation

- Forces of each particle can be computed independently
  - Accumulate results in local memory
  - Add accumulated results to previous position of particles
- New position used as input to the next time step to calculate new forces acting between particles



N = No. of particles in system

N

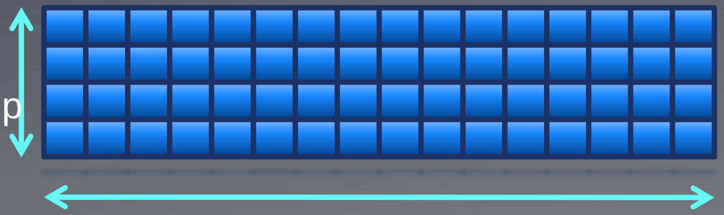
Force between all particles

Resultant force – per particle

Next Iteration

Naïve Parallel Implementation

- Disadvantages of implementation where each work item reads data independently
  - No reuse since redundant reads of parameters for multiple work-items
  - Memory access= N reads*N threads= N^2
- Similar to naïve non blocking matrix multiplication in Lecture 5



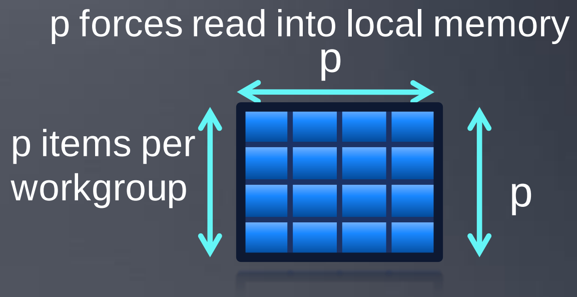
p items /workgroup

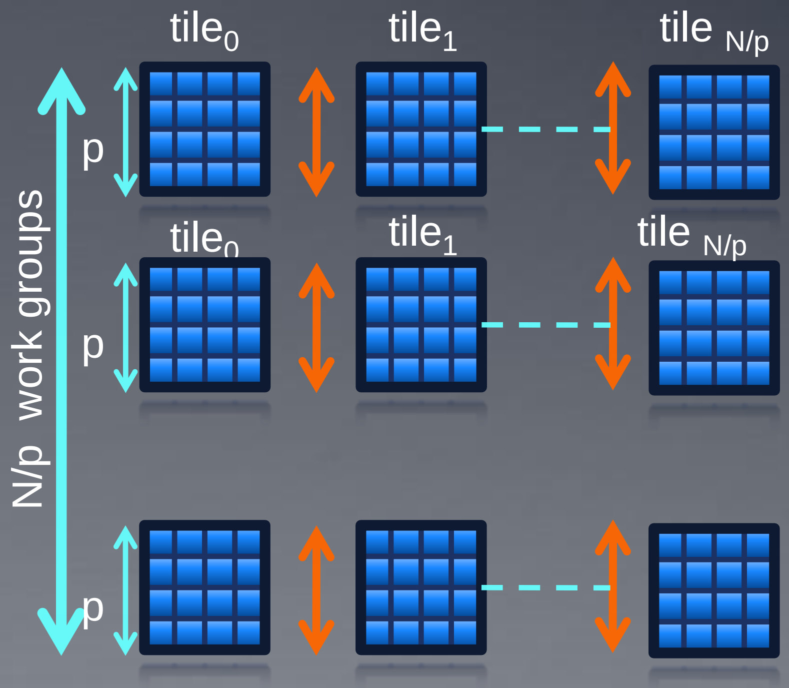
N = No. of particles
All N particles read in by each work item

```
__kernel void nbody(
        __global float4 * initial_pos,
        __global float4 * final_pos,
        Int N, __local float4 * result) {

    int localid = get_local_id(0);
    int globalid = get_global_id(0);
    result [localid] = 0;

    for( int i=0 ; i<N;i++)  {
        //! Calculate interaction between
        //! particle globalid and particle i
                GetForce(
                        globalid, i,
                        initial_pos, final_pos,
                        &result [localid]) ;
        }
        finalpos[ globalid] = result[ localid];
}
```

Local Memory Optimizations

- Data Reuse
  - Any particle read into compute unit can be used by all p bodies
- Computational tile:
  - Square region of the grid of forces consisting of size p
  - 2p descriptions required to evaluate all p^2 interactions in tile
  - p work items (in vertical direction) read in p forces
- Interactions on p bodies captured as an update to p acceleration vectors
- Intra-work group synchronization shown in orange required since all work items use data read by each work item





OpenCL Implementation

- Data reuse using local memory
  - Without reuse N*p items read per work group
  - With reuse p*(N/p) = N items read per work group
  - All work items use data read in by each work item
- SIGNIFICANT improvement: p is work group size (at least 128 in OpenCL, discussed in occupancy)
- Loop nest shows how a work item traverses all tiles
- Inner loop accumulates contribution of all particles within tile

Kernel Code

```
for (int i = 0; i < numTiles; ++i)
   {
      // load one tile into local memory
      int idx = i * localSize + tid;
      localPos[tid] = pos[idx];

     barrier(CLK_LOCAL_MEM_FENCE);

     // calculate acceleration effect due to each body
     for( int j = 0; j < localSize; ++j )    {
         // Calculate acceleration caused by particle j on i
         float4 r = localPos[j] – myPos;

         float distSqr = r.x * r.x  +  r.y * r.y  +  r.z * r.z;
         float invDist = 1.0f / sqrt(distSqr + epsSqr);
         float s = localPos[j].w * invDistCube;

         // accumulate effect of all particles
         acc += s * r;
      }
      // Synchronize so that next tile can be loaded
      barrier(CLK_LOCAL_MEM_FENCE);
   }
}
```

Performance

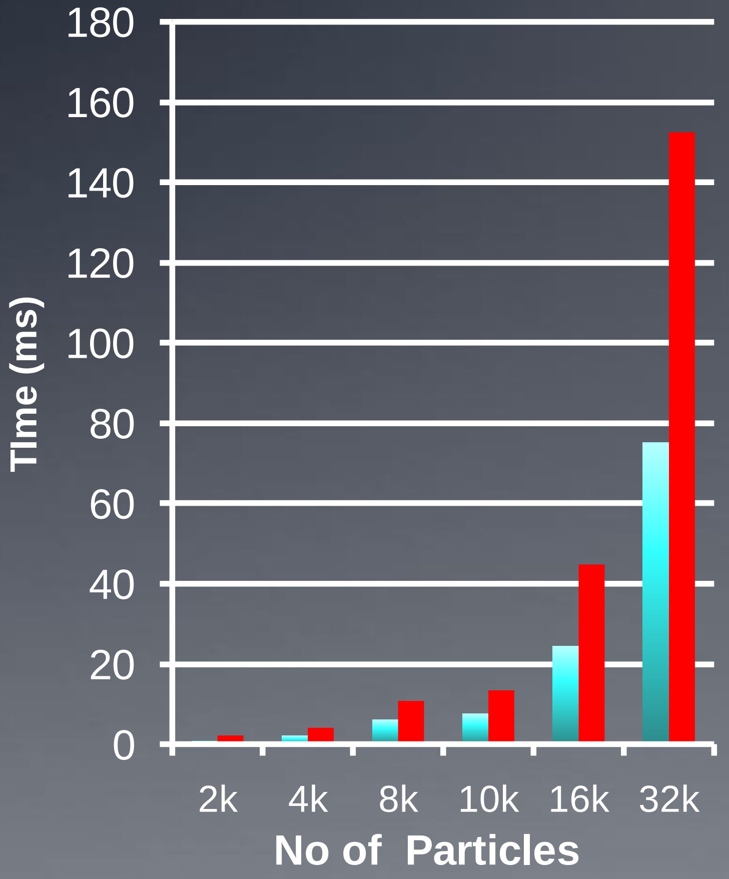
- Effect of optimizations compared for two GPU platforms
  - **Exactly same** code, only recompiled for platform

- Devices Compared
  - AMD GPU = 5870 Stream SDK 2.2
  - Nvidia GPU = GTX 480 with CUDA 3.1

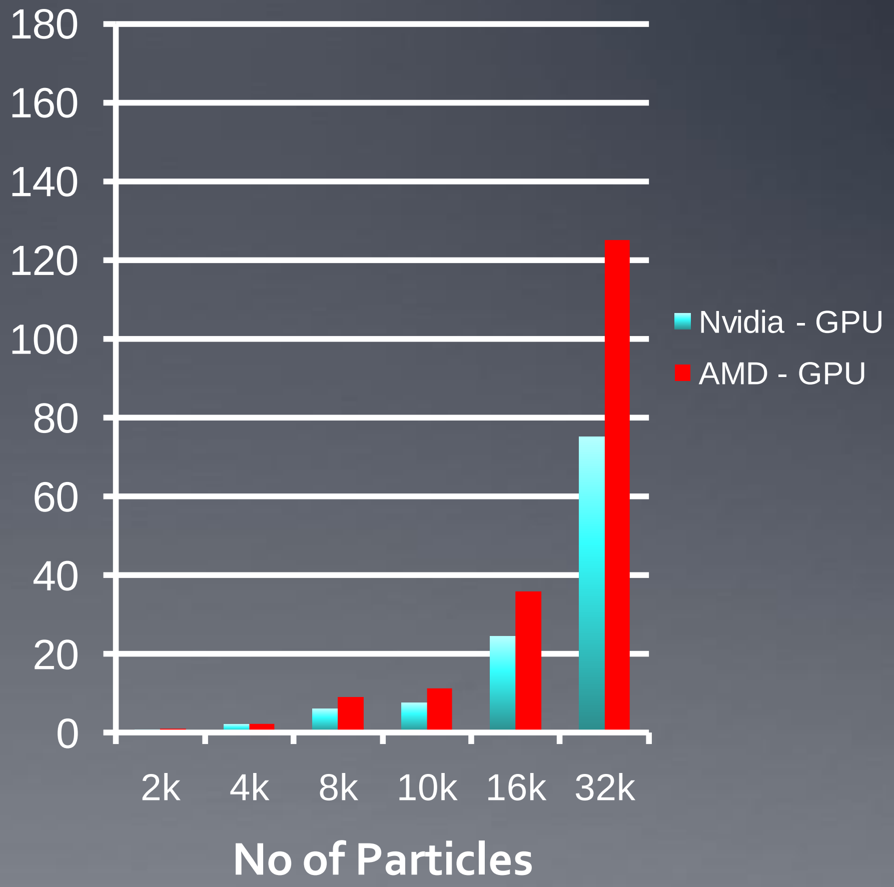
- Time measured for OpenCL kernel using OpenCL event counters (covered in in Lecture 11)
  - Device IO and other overheads like compilation time are not relevant to our discussion of optimizing a compute kernel
  - Events are provided in the OpenCL spec to query obtain timestamps for different state of OpenCL commands

Effect of Reuse on Kernel Performance

Execution Time – Non Reuse



Execution Time – Reuse



Thread Mapping and Vectorization

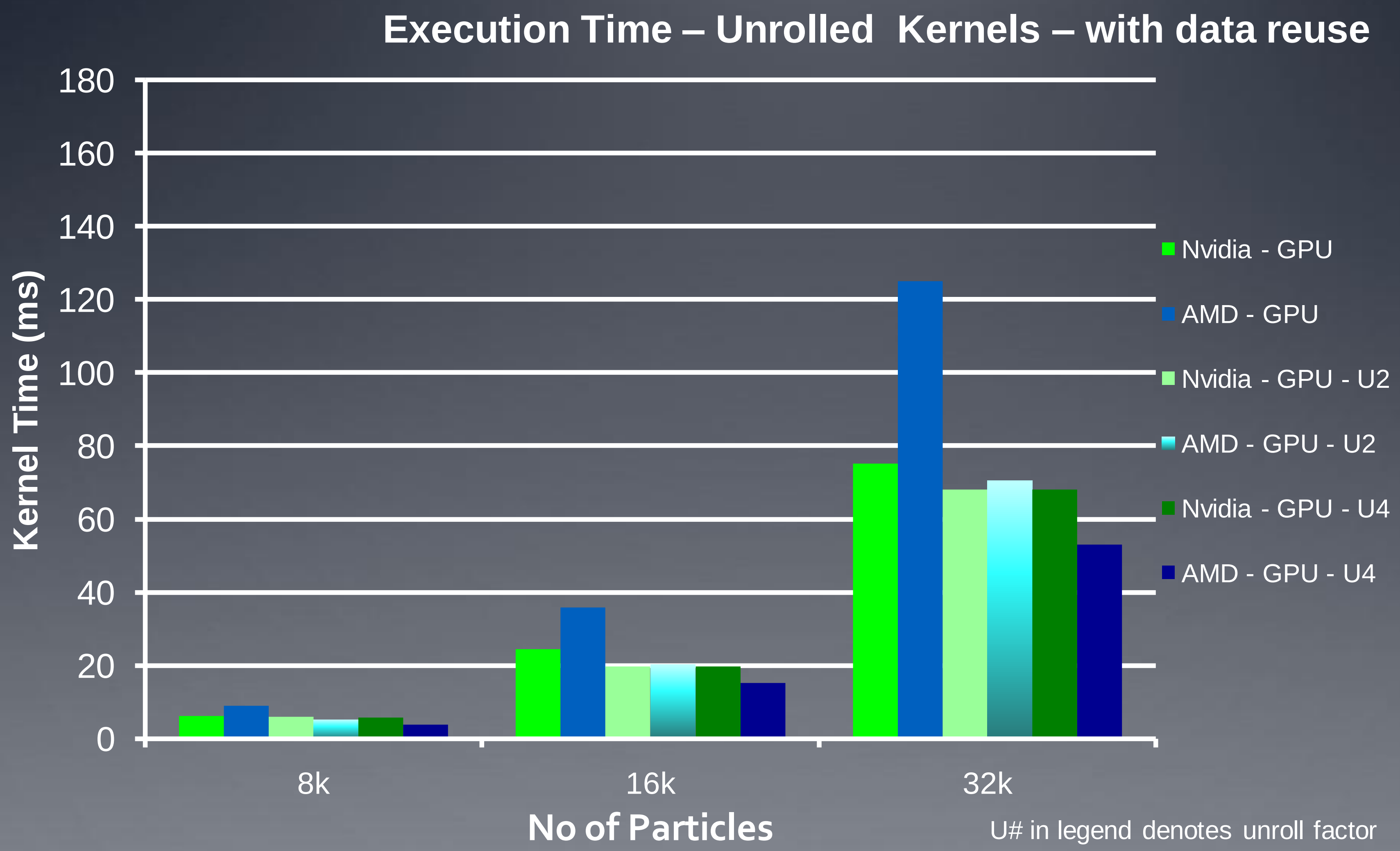
- As discussed in Lecture 8
  - Vectorization allows a single thread to perform multiple operations in one instruction
  - Explicit vectorization is achieved by using vector data-types (such as float4) in the source
- Vectorization using float4 enables efficient usage of memory bus for AMD GPUs
  - Easy in this case due to the vector nature of the data which are simply spatial coordinates
- Vectorization of memory accesses implemented using float4 to store coordinates

```
__kernel void nbody(
        __global float4 * pos,
        __global float4 * vel,
//float4 types enables improved
//usage of memory bus
) {

// Loop Nest Calculating per tile interaction
// Same loop nest as in previous slide
for (int i=0 ; i< numTiles; i++)
{
        for( int j=0; j<localsize; j ++)
}

}
```

Performance - Loop Unrolling

- We also attempt loop unrolling of the reuse local memory implementation
  - We unroll the innermost loop within the thread
- Loop unrolling can be used to improve performance by removing overhead of branching
  - However this is very beneficial only for tight loops where the branching overhead is comparable to the size of the loop body
  - Experiment on optimized local memory implementation
  - Executable size is not a concern for GPU kernels
- We implement unrolling by factors of 2 and 4 and we see substantial performance gains across platforms
  - Decreasing returns for larger unrolling factors seen

Effect of Unroll on Kernel Performance



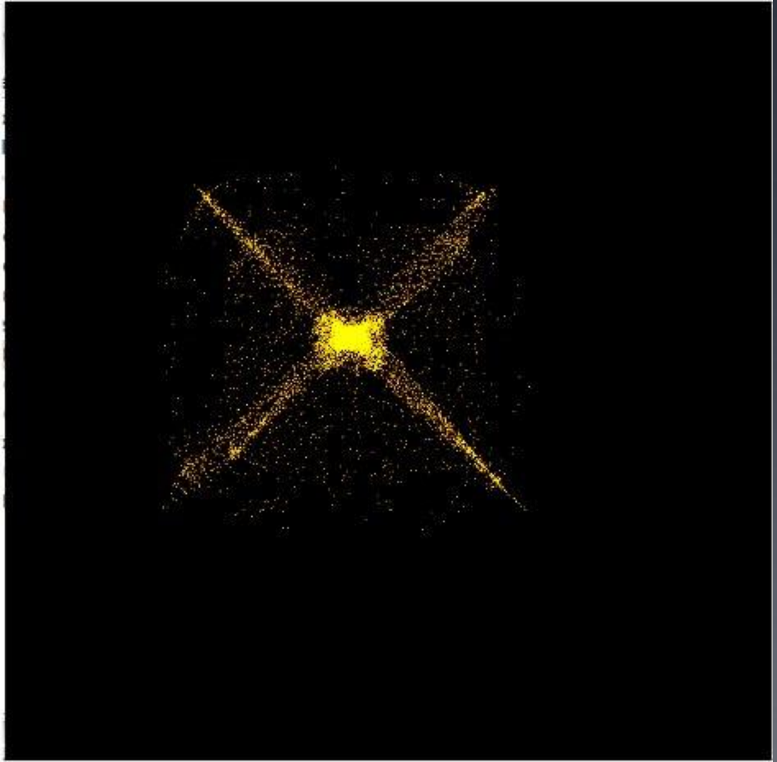
U# in legend denotes unroll factor

Performance Conclusions

- From the performance data we see:
- Unrolling greatly benefits the AMD GPU for our algorithm
  - This can be attributed to better packing of the VLIW instructions within the loop
  - Better packing is enabled by the increased amount of computation uninterrupted by a conditional statement when we unroll a loop
- The Nvidia Fermi performance doesn't benefit as much from the reuse and tiling
  - This can be attributed to the caching which is possible in the Fermi GPUs. The caching could enable data reuse.
  - As seen the Fermi performance is very close for both with and without reuse
- Note: You can even run this example on the CPU to see performance differences

Provided Nbody Example

- A N-body example is provided for experimentation and explore GPU optimization spaces
- Stand-alone application based on simpler on AMD SDK formulation
  - Runs correctly on AMD and Nvidia hardware
- Three kernels provided
  - Simplistic formulation
  - Using local memory tiling
  - Using local memory tiling with unrolling
- **Note:** Code is not meant to be a high performance N-body implementation in OpenCL
  - The aim is to serve as an optimization base for a data parallel algorithm



Screenshot of provided N-body demo running 10k particles on a AMD 5870

References for Nbody Simulation

- AMD Stream Computing SDK and the Nvidia GPU Computing SDK provide example implementations of N-body simulations
- Our implementation extends the AMD SDK example. The Nvidia implementation implementation is a different version of the N-body algorithm
- Other references
  - Nvidia GPU Gems
    - http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html
  - Brown Deer Technology
    - http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody.html

Summary

- We have studied a application as an optimization case study for OpenCL programming
- We have understood how architecture aware optimizations to data and code improve performance
- Optimizations discussed do not use device specific features like Warp size, so our implementation yields performance improvement while maintaining correctness on both AMD and Nvidia GPUs

Other Resources

- Well known optimization case studies for GPU programming
- CUDPP and Thrust provide CUDA implementations of parallel prefix sum, reductions and parallel sorting
  - The parallelization strategies and GPU specific optimizations in CUDA implementations can usually be applied to OpenCL as well
  - CUDPP http://code.google.com/p/cudpp/
  - Thrust http://code.google.com/p/thrust/
- Histogram on GPUs
  - The Nvidia and AMD GPU computing SDK provide different implementations for 64 and 256 bin Histograms
- Diagonal Sparse Matrix Vector Multiplication - OpenCL Optimization example on AMD GPUs and multi-core CPUs
  - http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study.aspx
- GPU Gems which provides a wide range of CUDA optimization studies