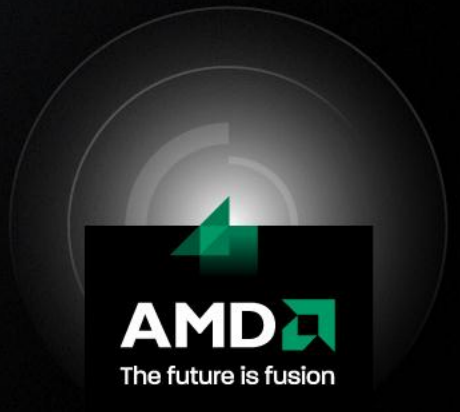
# OpenCL Do’s and Don’ts

Ofer Rosenberg
PMTS, OpenCL SW Dev.

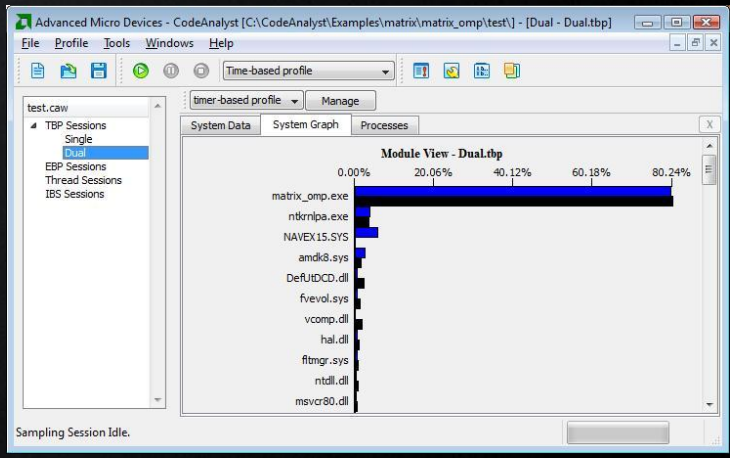
19/12/2011



# Application code do's and don'ts

Use OpenCL where its right

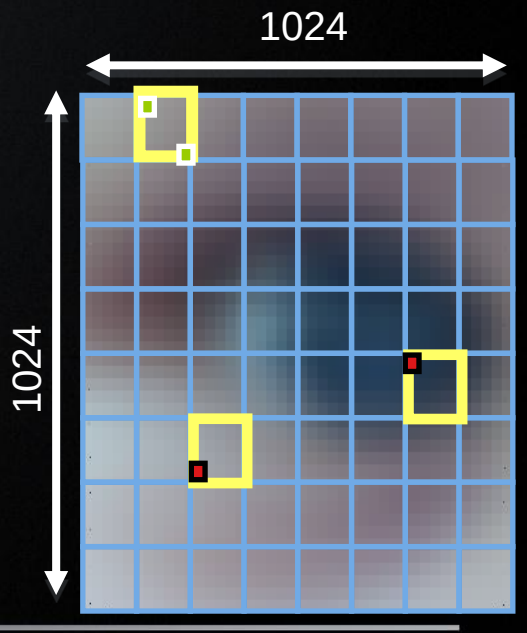
- Analyze the application code to find "hotspots"
- The hotspot code should be:
  - Highly Parallelized
  - Each instance is independent
- Remember the dispatch overhead
  - GPU ~ a few 10us
  - CPU also got "dispatch" overhead




# Application code do's and don'ts

Choosing the work size

- Global – use the largest possible
- Local – it's complicated ☺
  - Meet algorithm requirements on behavior
  - Key element for optimization on GPU
  - On CPU relevant only if there are barriers (fiber switch)
  - Too large – register spill to memory, cache misses, etc.
  - Too small – inefficient use of local memory, not hiding latency
  - No rule of thumb here – need experiments



# Memory Allocation

- Choose the right type based on usage and device
- USE_HOST_PTR is highly suitable for CPU only execution
- ALLOC_HOST_PTR is highly suitable for multi-device execution
- Some HW vendors offer special modes

Table 4.3 OpenCL Memory Object Properties

| clCreateBuffer/ clCreateImage Flags Argument | Device Type | Location | clEnqueueMapBuffer/ clEnqueueMapImage/ clEnqueueUnmapMemObject | |
|---|---|---|---|---|
| | | | Map Mode | Map Location |
| Default (none of the following flags) | Discrete GPU | Device memory | Copy | Mapped data size:<br>• <=32MiB: Pinned host memory<br>• >32MiB: Host memory (different memory area can be used on each map) |
| | Fusion APU | Device-visible host memory | | |
| | CPU | Use *Map Location* directly | Zero copy | |
| CL_MEM_ALLOC_HOST_PTR (clCreateBuffer on Windows 7 and Vista) | Discrete GPU | Pinned host memory shared by all devices in context (unless only device in context is CPU; then, host memory) | Zero copy | Use Location directly (same memory area is used on each map) |
| | Fusion APU | | | |
| | CPU | | | |
| CL_MEM_ALLOC_HOST_PTR (clCreateImage on Windows 7, Vista & Linux; clCreateBuffer on Linux) | Discrete GPU | Device memory | Copy | Pinned host memory (unless only device in context is CPU; then, host memory (same memory area is used on each map) |
| | Fusion APU | Device-visible memory | | |
| | CPU | | Zero copy | |
| CL_MEM_USE_HOST_PTR | Discrete GPU | Device memory | Copy | Pinned Host Memory (host memory is passed to host_ptr argument of clCreateBuffer / clCreateImage is pinned; it is unpinned when memory object is deleted (unless the only device in context is CPU; then, no pinning is done, and host memory is used) |
| | Fusion APU | Device-visible host memory | | |
| | CPU | Use *Map Location* directly | Zero copy | |
| CL_MEM_USE_PERSISTENT_MEM_AMD (Windows 7, Vista) | Discrete GPU | Host-visible device memory | Zero copy | Use *Location* directly (different memory area can be used on each map) |
| | Fusion APU | Device-visible host memory | | |
| | CPU | Host memory | | |
| CL_MEM_USE_PERSISTENT_MEM_AMD (Linux) | Same as Default. | | | |

Table taken from AMD APP Programming guide
http://developer.amd.com/sdks/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

# Memory Access

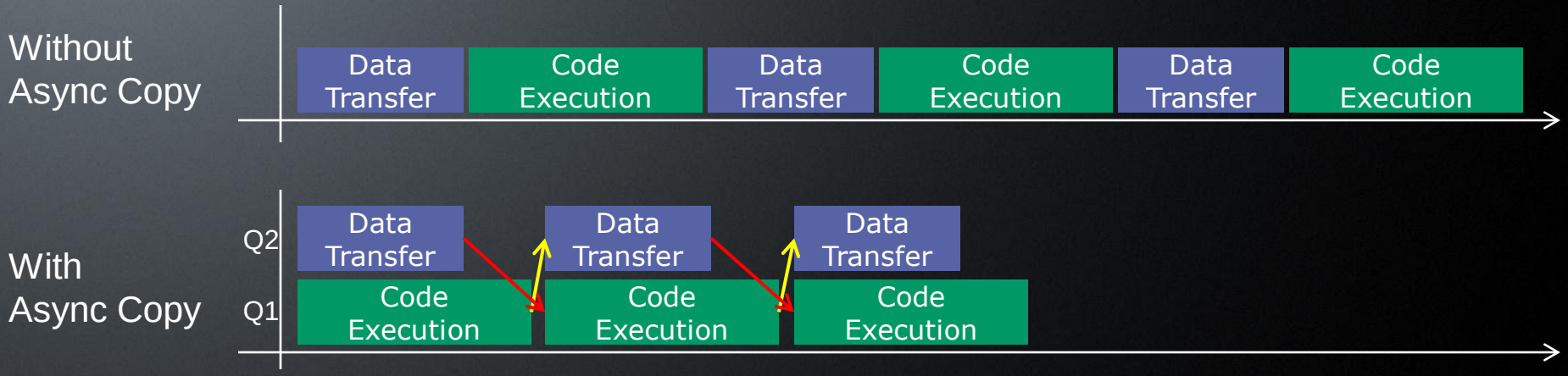
- OpenCL supports two patterns of Memory Access:
  - Write/Execute/Read
  - Unmap/Execute/Map
- Choosing a pattern is based on Application needs – the goal is to minimize copies/allocations. Examples:
  - If the Application receives and sends buffers with varying addresses, choose read and writes
  - If the Application processes the buffer (for example, analyze it), choose map/unmap to avoid additional memory allocation
- Caution! Asynchronous operation

```
clEnqueueRead/Write/Map/Unmap (queue, object, blocking, offset, size, *ptr, ...)
```

  - Choose "blocking" to ensure memory is copied when the operation is done
  - Otherwise, monitor the event… (using "wait for event", or event callback)

# Asynchronous Copy (Transfer)

- Relevant only for GPUs (or non-Host devices)
- Available on some Vendor solutions (HW/SW)
- The basic idea is overlap of data transfer and code execution




- How to enable Async Copy ?
  - Use two queues, one for data transfer and one for execution
  - Create the right event dependency between them
  - The SW & HW will utilize Async Copy automatically

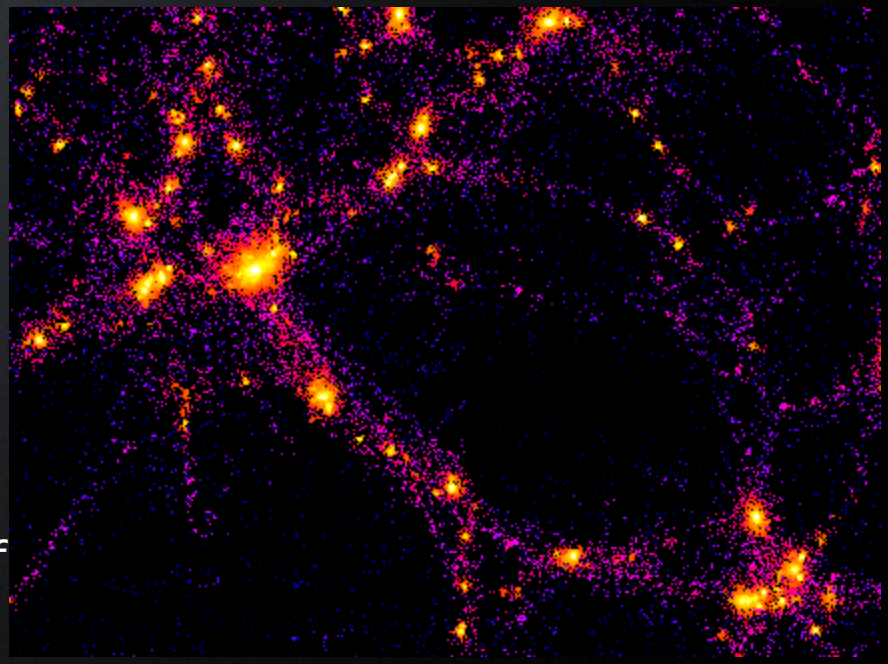
# The following slides are taken from AMD OpenCL University Kit

http://developer.amd.com/zones/openclzone/universities/pages/default.aspx

Perhaad Mistry & Dana Schaa,
Northeastern University Computer Architecture Research Lab,
with Benedict R. Gaster, AMD


# 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 j

$G$ = Gravitational Constant

$m_i$ = Mass of particle i

$m_j$ = Mass of particle j

$r_{ij}$ = Distance of particle i and j

For each particle this becomes

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

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

# 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

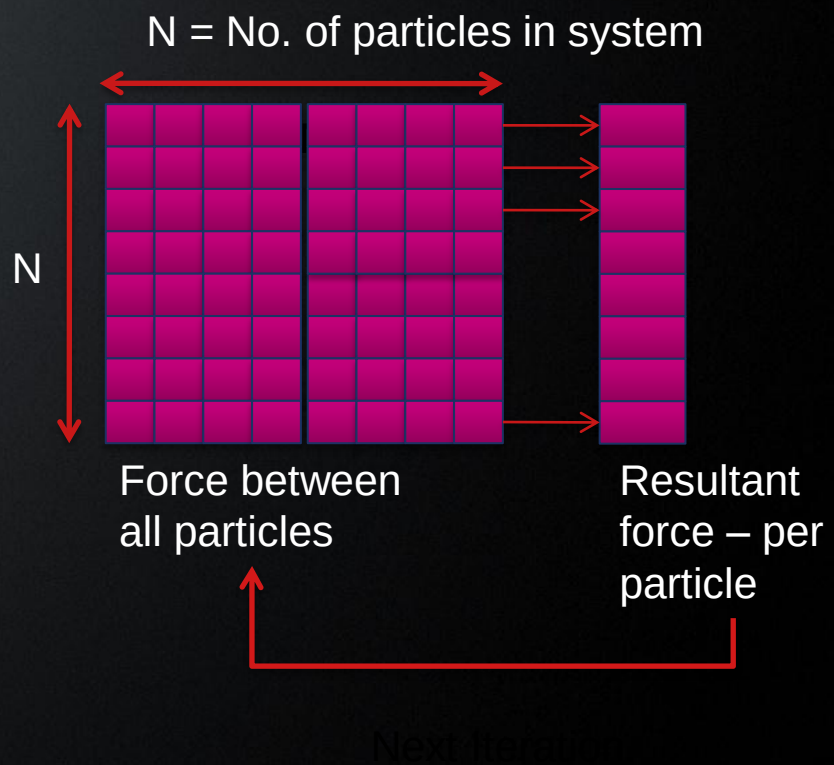
```
for(i=0; i<n; i++)
{
    ax = ay = az = 0;
   // Loop over all particles "j"
    for (j=0; j<n; j++)  {

       //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
}
```

# All Pairs – full implementation

```
Void NBody::nBodyCPUReference()
{
    //Iterate for all samples
    for(int i = 0; i < numBodies; ++i)
    {
        int myIndex = 4 * i;
        float acc[3] = {0.0f, 0.0f, 0.0f};
        for(int j = 0; j < numBodies; ++j)
        {
            float r[3];
            int index = 4 * j;
            float distSqr = 0.0f;
            for(int k = 0; k < 3; ++k)
            {
                r[k] = refPos[index + k] - refPos[myIndex + k];
                distSqr += r[k] * r[k];
            }

            float invDist = 1.0f / sqrt(distSqr + espSqr);
            float invDistCube =  invDist * invDist * invDist;
            float s = refPos[index + 3] * invDistCube;

            for(int k = 0; k < 3; ++k)  {  acc[k] += s * r[k]; }
        }

        for(int k = 0; k < 3; ++k)
        {
            refPos[myIndex + k] += refVel[myIndex + k] * delT + 0.5f * acc[k] * delT * delT;
            refVel[myIndex + k] += acc[k] * delT;
        }
    }
}
```

# 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



# Hands on #1

Application code envelope is provided.

Partial kernel called “NBody_Kernels_basic.cl” is provided

1. Use the basic implementation on previous slide to create OpenCL Kernel that implements N-body

2. Run on CPU device and GPU device, and compare

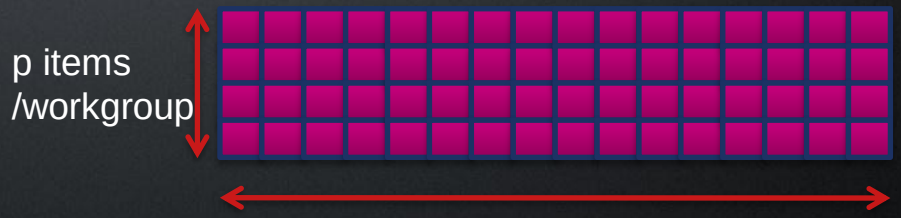
3. The code uses device-local buffers, and copy. Convert to USE_HOST_PTR and Map/Unmap, run the two versions on the CPU and compare

# 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

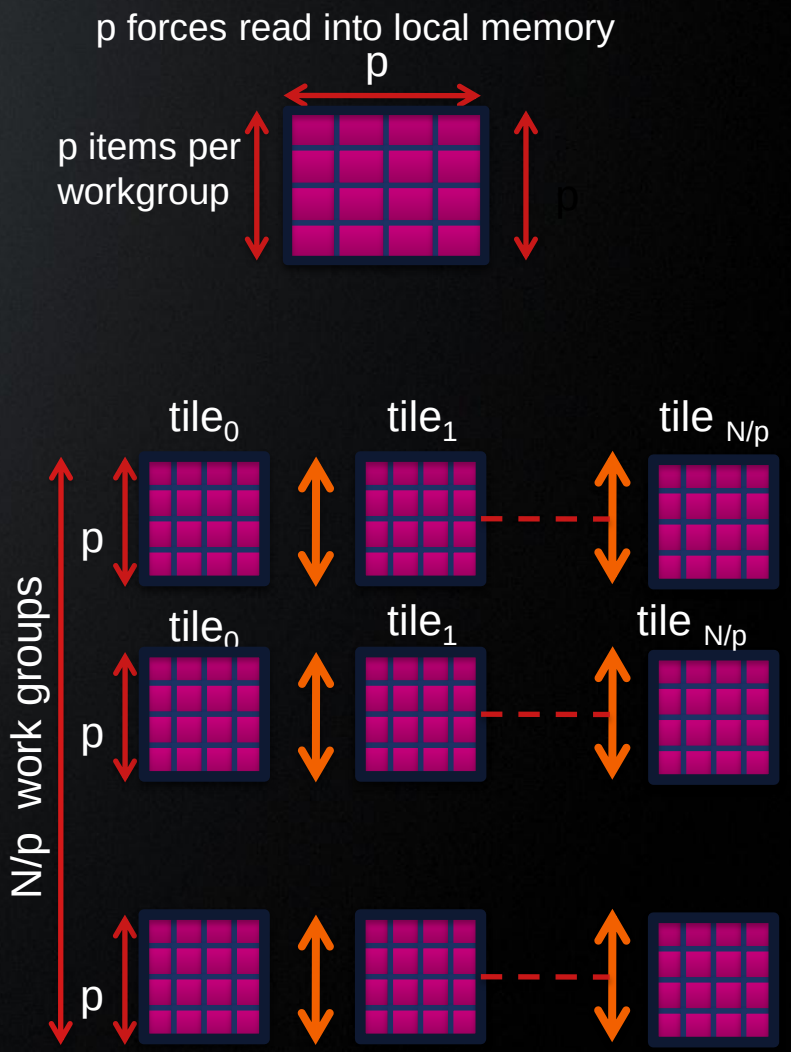



N = No. of particles
All N particles read in by each work item

```
__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



# Hands on #2

Application code and kernel code for tiled access is provided

1. Use the description to convert the kernel to use local memory

2. Run on CPU device and GPU device, with and without local memory and compare

# 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

```
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);
   }
}
```

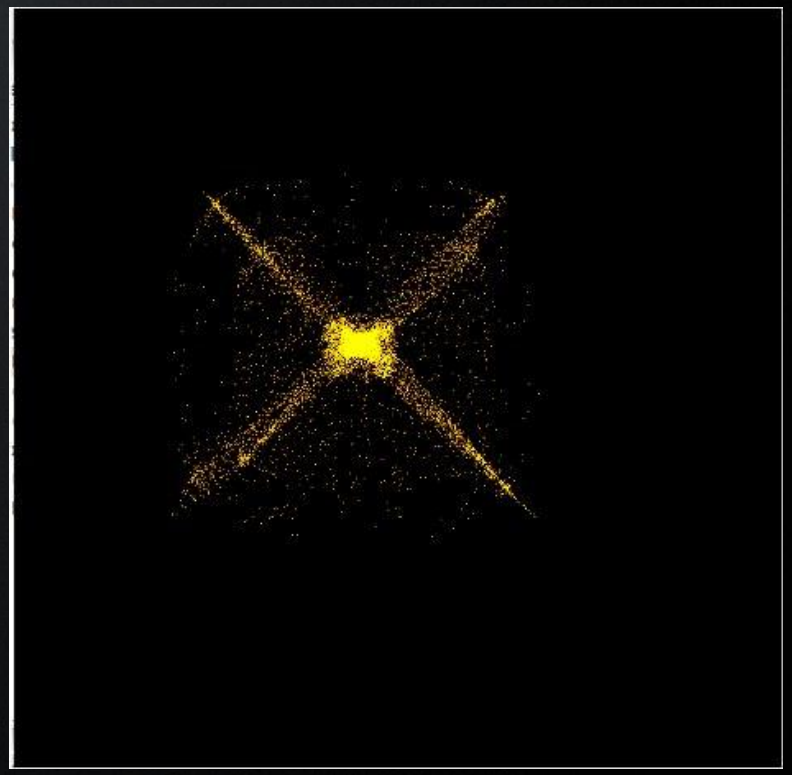
# Performance - Loop Unrolling

- 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


Perhaad Mistry & Dana
Schaa, Northeastern Univ

# 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
- 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 example running 10k particles on a Radeon 6970