

OPENCL调试与性能优化

汤善江 副教授

天津大学智能与计算学部

tashj@tju.edu.cn

<http://cic.tju.edu.cn/faculty/tangshanjiang/>

Outline

- OpenCL调试
 - CPU调试
 - GPU调试
- OpenCL性能优化
 - 线程映射
 - 设备占用
 - 向量化
- N-body仿真实例

Outline

- **OpenCL调试**
 - **CPU调试**
 - GPU调试
- OpenCL性能优化
 - 线程映射
 - 设备占用
 - 向量化
- N-body仿真实例

OpenCL调试技术

- Compiling for x86 CPU
 - Debugging with GDB
- GPU printf
- Live debuggers
 - gDEBugger

CPU端程序调试

- OpenCL allows the same code to run on different types of devices
 - Compiling to run on a CPU provides some extra facilities for debugging
 - Additional forms of IO (such as writing to disk) are still not available from the kernel
- AMD's OpenCL implementation recognizes any x86 processor as a target device
 - Simply select the CPU as the target device when executing the program
- NVIDIA's OpenCL implementation can support compiling to x86 CPUs if AMD's installable client driver is installed

CPU端程序调试——GDB

- Setting up for GDB
 - Pass the compiler the “-g” flag
 - Pass “-g” to `clBuildProgram()`
 - Set an environment variable `CPU_COMPILER_OPTIONS="-g"`
 - Avoid non-deterministic execution by setting an environment variable `CPU_MAX_COMPUTE_UNITS=1`

CPU端程序调试——GDB

- Run gdb with the OpenCL executable

```
> gdb a.out
```
- Breakpoints can be set by line number, function name, or kernel name
- To break at the kernel `hello` within gdb, enter:

```
(gdb) b __OpenCL_hello_kernel
```

 - The prefix and suffix are required for kernel names
- OpenCL kernel symbols are not known until the kernel is loaded, so setting a breakpoint at `clEnqueueNDRangeKernel()` is helpful

```
(gdb) b clEnqueueNDRangeKernel
```

CPU端程序调试——GDB

- To break on a certain thread, introduce a conditional statement in the kernel and set the breakpoint inside the conditional body
 - Can use gdb commands to view thread state at this point

...

```
if (get_global_id(1) == 20 &&  
    get_global_id(0) == 34) {  
    ; // Set breakpoint on this line  
}
```


Outline

- **OpenCL调试**
 - CPU调试
 - **GPU调试**
- OpenCL性能优化
 - 线程映射
 - 设备占用
 - 向量化
- N-body仿真实例

GPU程序调试——GPU Printf

- AMD GPUs support printing during execution using `printf()`
 - NVIDIA does not currently support printing for OpenCL kernels (though they do with CUDA/C)
- AMD requires the OpenCL extension `cl_amd_printf` to be enabled in the kernel
- `printf()` closely matches the definition found in the C99 standard

GPU程序调试—— GPU Printf

- `printf()` can be used to print information about threads or check help track down bugs
- The following example prints information about threads trying to perform an improper memory access

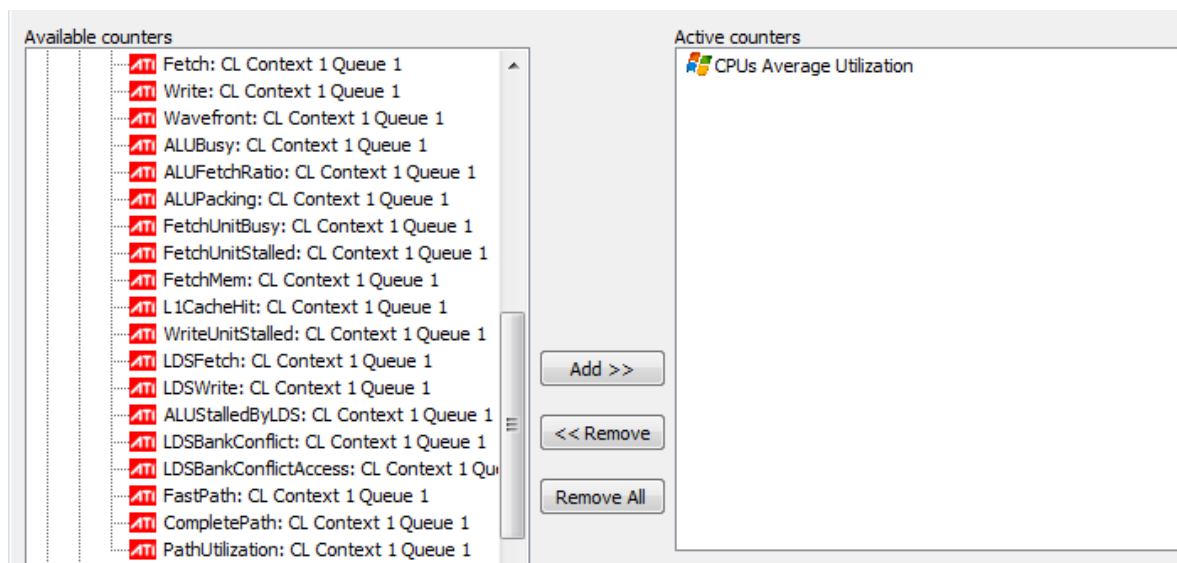
```
int myIdx = ... // index for addressing a matrix
if(myIdx < 0 || myIdx >= rows || myIdx >= cols) {
    printf("Thread %d,%d: bad index (%d)\n",
        get_global_id(1), get_global_id(0), myIdx);
}
```

GPU程序调试——GPU Printf

- `printf()` works by buffering output until the end of execution and transferring the output back to the host
 - It is important that a kernel completes in order to retrieve printed information
 - Commenting out code following `printf()` is a good technique if the kernel is crashing

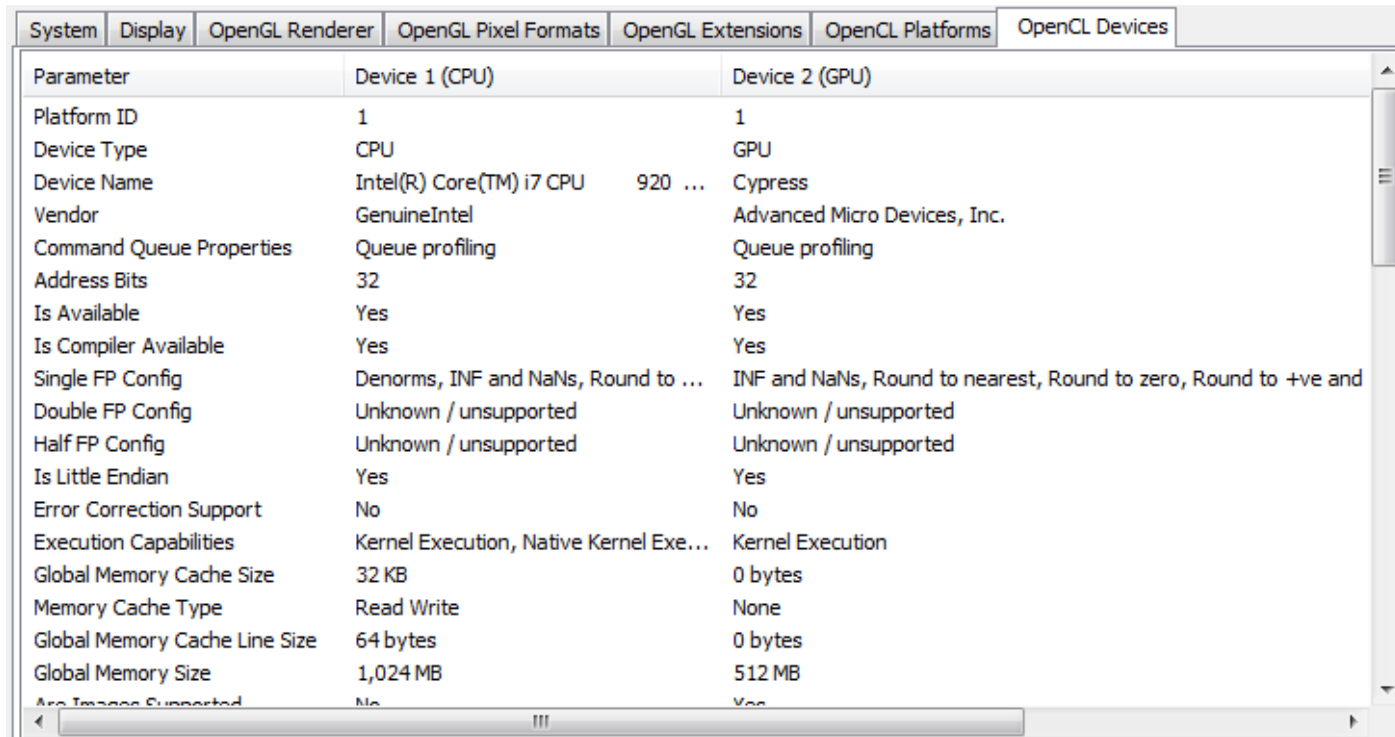
GPU程序调试——gDEBugger

- Developed by Graphic Remedy
 - Cost: not free
- Debugger, profiler, memory analyzer
- Integrated with AMD/ATI and NVIDIA performance counters



GPU程序调试——gDEBugger

- Displays information about OpenCL platforms and devices present in the system



The screenshot shows the 'OpenCL Devices' tab in the gDEBugger application. It displays a table with two columns: 'Device 1 (CPU)' and 'Device 2 (GPU)'. The table lists various parameters for each device, such as Platform ID, Device Type, Device Name, Vendor, and memory capabilities. The CPU device is an Intel(R) Core(TM) i7 CPU, and the GPU device is a Cypress GPU from Advanced Micro Devices, Inc.

Parameter	Device 1 (CPU)	Device 2 (GPU)
Platform ID	1	1
Device Type	CPU	GPU
Device Name	Intel(R) Core(TM) i7 CPU 920 ...	Cypress
Vendor	GenuineIntel	Advanced Micro Devices, Inc.
Command Queue Properties	Queue profiling	Queue profiling
Address Bits	32	32
Is Available	Yes	Yes
Is Compiler Available	Yes	Yes
Single FP Config	Denorms, INF and NaNs, Round to ...	INF and NaNs, Round to nearest, Round to zero, Round to +ve and
Double FP Config	Unknown / unsupported	Unknown / unsupported
Half FP Config	Unknown / unsupported	Unknown / unsupported
Is Little Endian	Yes	Yes
Error Correction Support	No	No
Execution Capabilities	Kernel Execution, Native Kernel Exe...	Kernel Execution
Global Memory Cache Size	32 KB	0 bytes
Memory Cache Type	Read Write	None
Global Memory Cache Line Size	64 bytes	0 bytes
Global Memory Size	1,024 MB	512 MB
Are Images Supported	No	Yes

GPU程序调试——gDEBugger

- Can step through OpenCL calls, and view arguments
 - Links to programs, kernels, etc. when possible in the function call view

CL Context 1 - 6 OpenCL function calls

```
clCreateCommandQueue(Context 1, Device 1, CL_NONE
clCreateBuffer(Context 1, CL_MEM_READ_ONLY | CL_ME
clCreateBuffer(Context 1, CL_MEM_READ_ONLY | CL_ME
clCreateBuffer(Context 1, CL_MEM_READ_WRITE, 4 KB, 0
clCreateProgramWithSource(Context 1, 1, 0x002FCE8, 0
➔ clBuildProgram(Program 1, 1, 0x003E9680, , 0x00000000,
```

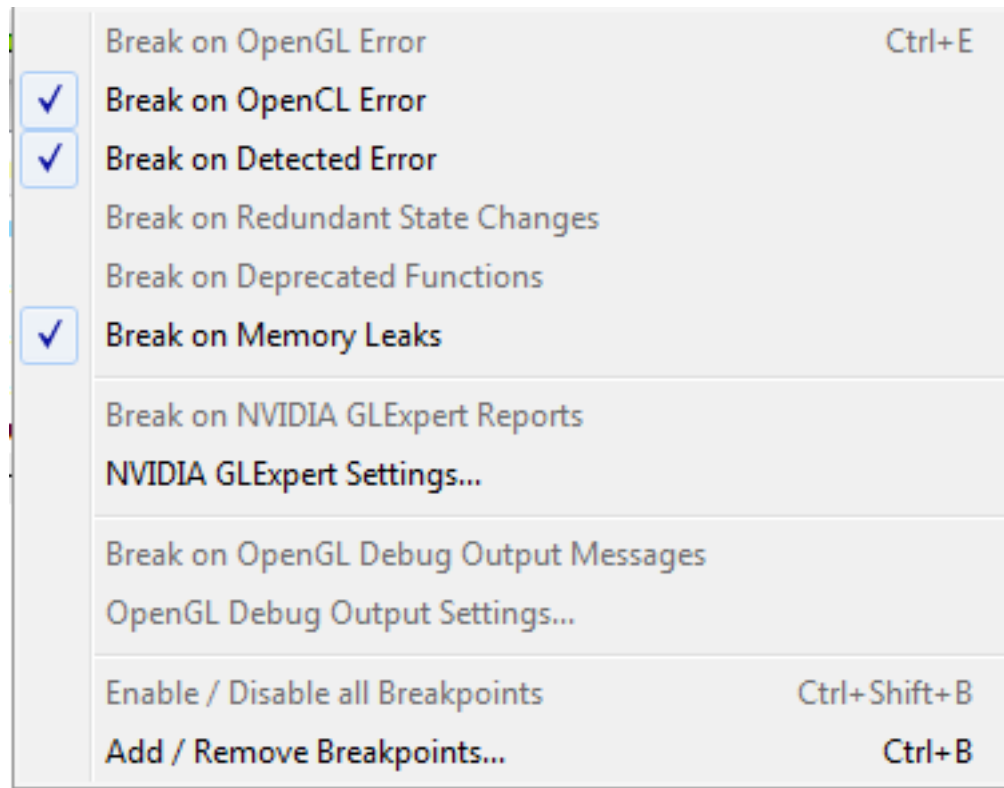
Properties

OpenCL Function Call

Name	clBuildProgram
Arguments	(0x05E77A20 - Program 1, 1, 0x003E9680, "", 0x00000000, 0x00000000)

GPU程序调试——gDEBugger

- Automatically detects OpenCL errors and memory leaks



GPU程序调试——gDEBugger

- Displays contents of buffers and images present on OpenCL devices
 - View live
 - Export to disk

The screenshot displays the gDEBugger interface with several panels:

- Graphic Objects:** A tree view on the left showing the hierarchy of OpenCL objects. It includes 'VA_gdebugger', 'CL Context 1', 'Buffers' (containing 'CL Buffer 1', 'CL Buffer 2', and 'CL Buffer 3'), 'Command Queues', and 'Computation Programs'. 'CL Buffer 1' is currently selected.
- Table:** A table in the upper right lists the buffers and their properties.

Name	Memory Size	Memory Flags
Buffer 1	4 KB	CL_MEM_READ_O...
Buffer 2	4 KB	CL_MEM_READ_O...
Buffer 3	4 KB	CL_MEM_READ_W...
Total	12 KB	
- Object Creation Calls Stack:** A panel at the bottom left showing the stack of calls that created the selected object. It lists two entries: '0x00fdb75f -' and '0x1cf7dc82 -'.
- Graph View:** A panel at the bottom center showing a 3D pie chart with three segments of different colors (cyan, blue, and dark blue).
- Properties View:** A panel on the bottom right showing the detailed properties of the selected 'CL Buffer 1'.

CL Buffer 1	
General	
Buffer Name	CL Buffer 1
Buffer Handle	0x02e36a80
Size	4KB
Flags	CL_MEM_READ_ONLY CL_MEM_COPY_HOST_PTR

At the bottom of the interface, there is a status bar with a checked box for 'Break On Memory Leaks' and the text 'CL Buffer 1'.

Outline

- OpenCL调试
 - CPU调试
 - GPU调试
- **OpenCL性能优化**
 - **线程映射**
 - 设备占用
 - 向量化
- N-body仿真实例

线程映射

- Thread mapping determines which threads will access which data
 - Proper mappings can align with hardware and provide large performance benefits
 - Improper mappings can be disastrous to performance

线程映射

- By using different mappings, the same thread can be assigned to access different data elements
 - The examples below show three different possible mappings of threads to data (assuming the thread id is used to access an element)

```
int group_size =
get_local_size(0) *
get_local_size(1);
```

```
int tid =
get_group_id(1) *
get_num_groups(0) *
group_size +
get_group_id(0) *
group_size +
get_local_id(1) *
get_local_size(0) +
get_local_id(0)
```

Mapping

```
int tid =
get_global_id(1) *
get_global_size(0) +
get_global_id(0);
```

```
int tid =
get_global_id(0) *
get_global_size(1) +
get_global_id(1);
```

Thread IDs

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

*assuming 2x2 groups

线程映射

- Consider a serial matrix multiplication algorithm

```
for(i1=0; i1 < M; i1++)  
    for(i2=0; i2 < N; i2++)  
        for(i3=0; i3 < P; i3++)  
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
 - We will create NM threads
 - Effectively removing the outer two loops
 - Each thread will perform P calculations
 - The inner loop will remain as part of the kernel
- Should the index space be $M \times N$ or $N \times M$?

线程映射

- Thread mapping 1: with an $M \times N$ index space, the kernel would be:

```
int tx = get_global_id(0);  
int ty = get_global_id(1);  
for(i3=0; i3<P; i3++)  
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

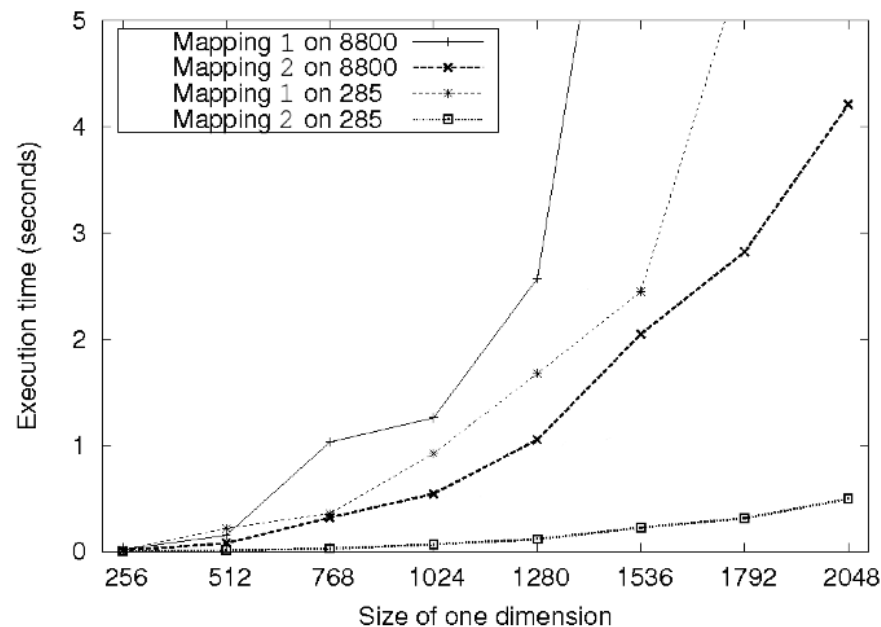
- Thread mapping 2: with an $N \times M$ index space, the kernel would be:

```
int tx = get_global_id (0);  
int ty = get_global_id (1);  
for(i3=0; i3<P; i3++)  
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

- Both mappings produce functionally equivalent versions of the program

线程映射

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs



- Notice that mapping 2 is far superior in performance for both GPUs

线程映射

- The discrepancy in execution times between the mappings is due to data accesses on the global memory bus
 - Assuming row-major data, data in a row (i.e., elements in adjacent columns) are stored sequentially in memory
 - To ensure coalesced accesses, consecutive threads in the same wavefront should be mapped to columns (the second dimension) of the matrices
 - This will give coalesced accesses in Matrices B and C
 - For Matrix A, the iterator $i3$ determines the access pattern for row-major data, so thread mapping does not affect it

线程映射

- In mapping 1, consecutive threads (tx) are mapped to different rows of Matrix C, and non-consecutive threads (ty) are mapped to columns of Matrix B
 - The mapping causes inefficient memory accesses

```
int tx = get_global_id(0);  
int ty = get_global_id(1);  
for(i3=0; i3<P; i3++)  
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

线程映射

- In mapping 2, consecutive threads (tx) are mapped to consecutive elements in Matrices B and C
 - Accesses to both of these matrices will be coalesced
 - Degree of coalescence depends on the workgroup and data sizes

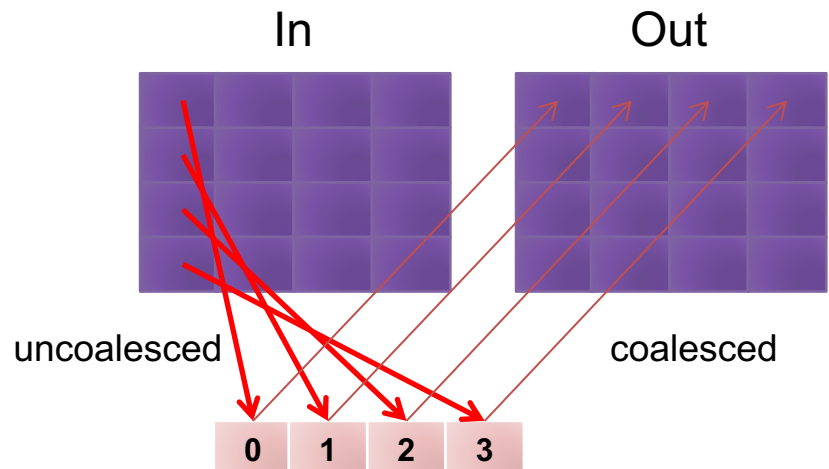
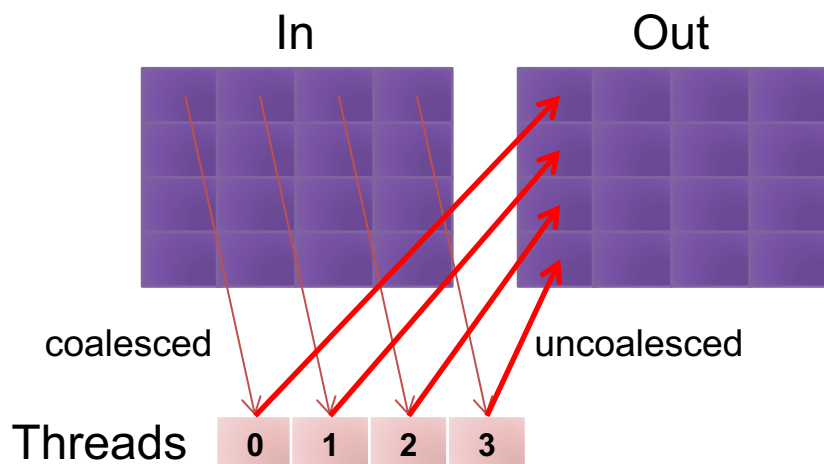
```
int tx = get_global_id (0);  
int ty = get_global_id (1);  
for(i3=0; i3<P; i3++)  
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

线程映射

- In general, threads can be created and mapped to any data element by manipulating the values returned by the thread identifier functions
- The following matrix transpose example will show how thread IDs can be modified to achieve efficient memory accesses

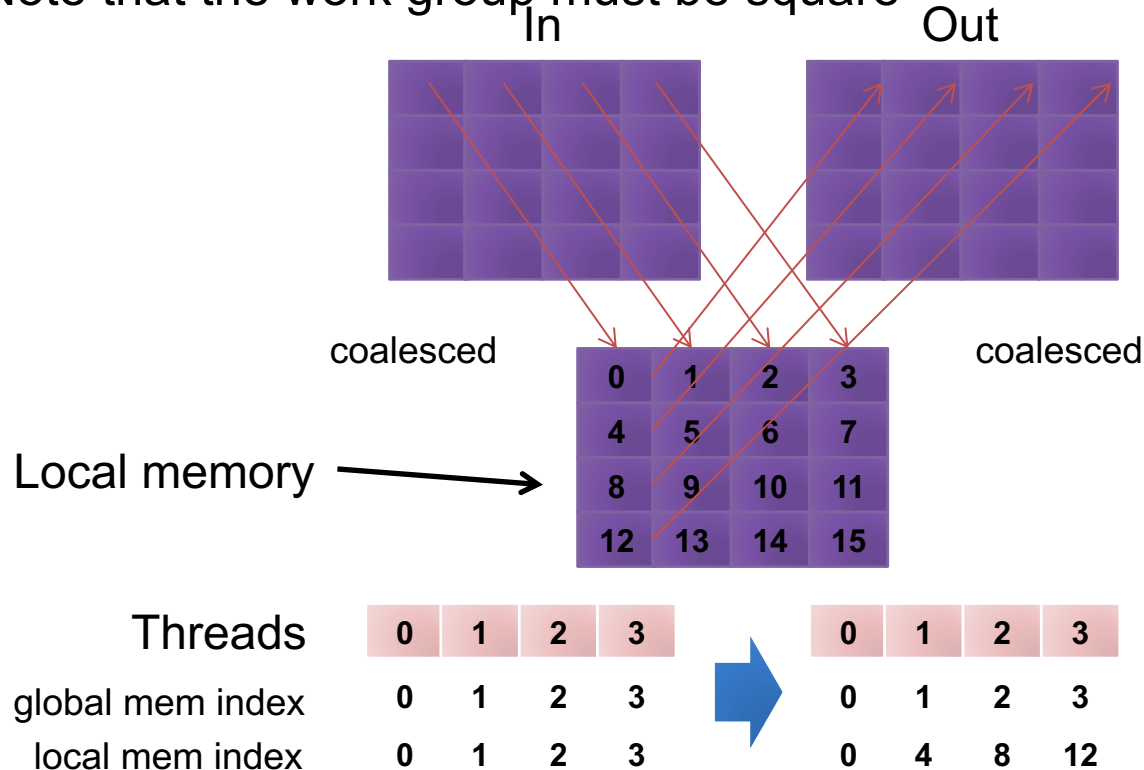
Matrix Transpose

- A matrix transpose is a straightforward technique
 - $\text{Out}(x,y) = \text{In}(y,x)$
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
 - Note that data must be read to a temporary location (such as a register) before being written to a new location



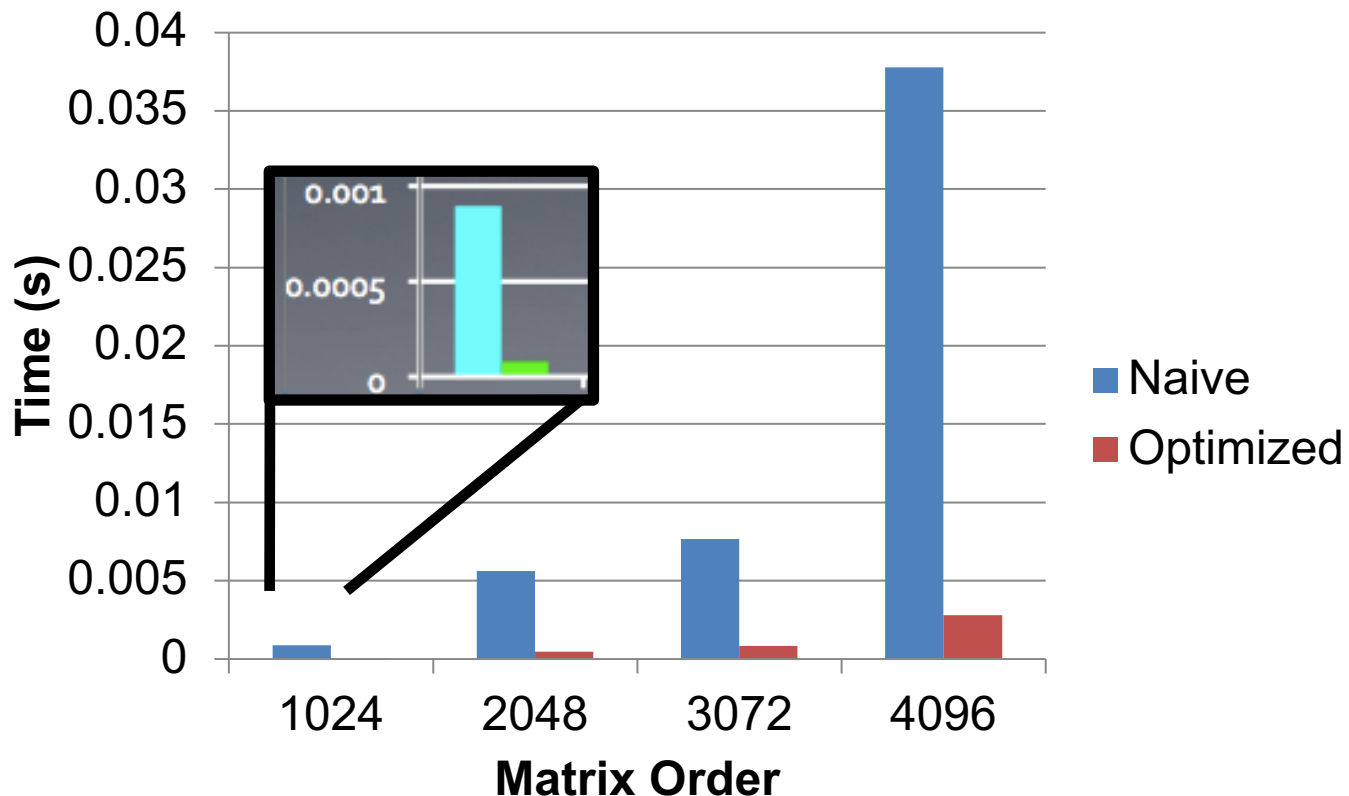
Matrix Transpose

- If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions
 - Note that the work group must be square



Matrix Transpose

- The following figure shows a performance comparison of the two transpose kernels for matrices of size $N \times M$ on an AMD 5870 GPU
 - “Optimized” uses local memory and thread remapping



Outline

- OpenCL调试
 - CPU调试
 - GPU调试
- **OpenCL性能优化**
 - 线程映射
 - **设备占用**
 - 向量化
- N-body仿真实例

设备占用

- On current GPUs, work groups get mapped to compute units
 - When a work group is mapped to a compute unit, it cannot be swapped off until all of its threads complete their execution
- If there are enough resources available, multiple work groups can be mapped to the same compute unit at the same time
 - Wavefronts from another work group can be swapped in to hide latency
- Resources are fixed per compute unit (number of registers, local memory size, maximum number of threads)
 - Any one of these resource constraints may limit the number of work groups on a compute unit
- The term *occupancy* is used to describe how well the resources of the compute unit are being utilized

设备占用——寄存器

- The availability of registers is one of the major limiting factor for larger kernels
- The maximum number of registers required by a kernel must be available for all threads of a workgroup
 - Example: Consider a GPU with 16384 registers per compute unit running a kernel that requires 35 registers per thread
 - Each compute unit can execute at most 468 threads
 - This affects the choice of workgroup size
 - A workgroup of 512 is not possible
 - Only 1 workgroup of 256 threads is allowed at a time, even though 212 more threads could be running
 - 3 workgroups of 128 threads are allowed, providing 384 threads to be scheduled, etc.

设备占用——寄存器

- Consider another example:
 - A GPU has 16384 registers per compute unit
 - The work group size of a kernel is fixed at 256 threads
 - The kernel currently requires 17 registers per thread
- Given the information, each work group requires 4352 registers
 - This allows for 3 active work groups if registers are the only limiting factor
- If the code can be restructured to only use 16 registers, then 4 active work groups would be possible

设备占用——本地内存

- GPUs have a limited amount of local memory on each compute unit
 - 32KB of local memory on AMD GPUs
 - 32-48KB of local memory on NVIDIA GPUs
- Local memory limits the number of active work groups per compute unit
- Depending on the kernel, the data per workgroup may be fixed regardless of number of threads (e.g., histograms), or may vary based on the number of threads (e.g., matrix multiplication, convolution)

设备占用——线程资源

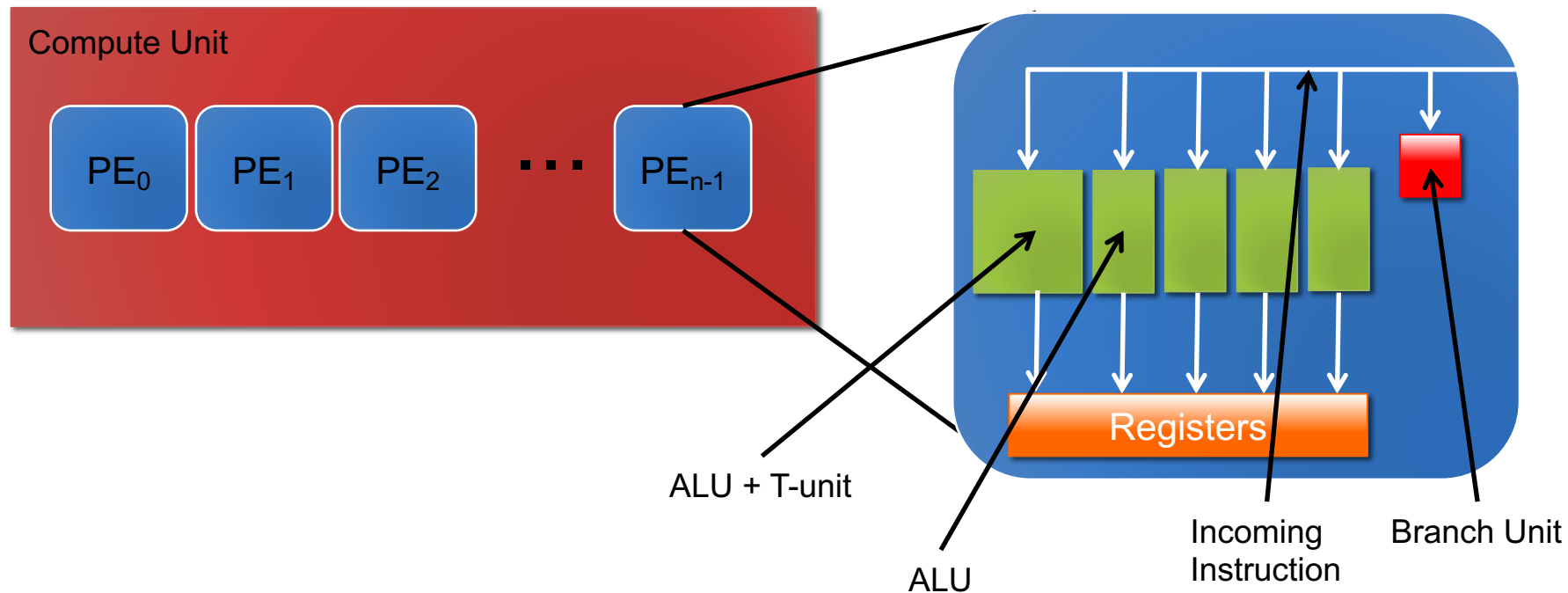
- GPUs have hardware limitations on the maximum number of threads per work group
 - 256 threads per WG on AMD GPUs
 - 512 threads per WG on NVIDIA GPUs
- NVIDIA GPUs have per-compute-unit limits on the number of active threads and work groups (depending on the GPU model)
 - 768 or 1024 threads per compute unit
 - 8 or 16 warps per compute unit
- AMD GPUs have GPU-wide limits on the number of wavefronts
 - 496 wavefronts on the AMD 5870 GPU (~25 wavefronts or ~1600 threads per compute unit)

Outline

- OpenCL调试
 - CPU调试
 - GPU调试
- **OpenCL性能优化**
 - 线程映射
 - 设备占用
 - **向量化**
- N-body仿真实例

向量化

- On AMD GPUs, each processing element executes a 5-way VLIW instruction
 - 5 scalar operations or
 - 4 scalar operations + 1 transcendental operation



向量化

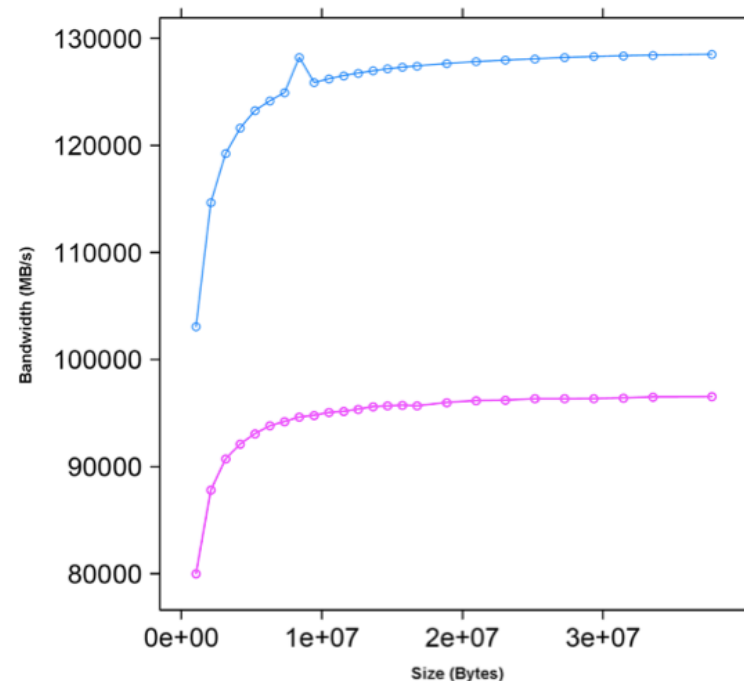
- Vectorization allows a single thread to perform multiple operations at once
- Explicit vectorization is achieved by using vector datatypes (such as `float4`) in the source program
 - When a number is appended to a datatype, the datatype becomes an array of that length
 - Operations can be performed on vector datatypes just like regular datatypes
 - Each ALU will operate on different element of the `float4` data

向量化

- Vectorization improves memory performance on AMD GPUs
 - The *AMD Accelerated Parallel Processing OpenCL Programming Guide* compares `float` to `float4` memory bandwidth

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return;
}

__kernel void
Copy1(__global const float * input,
      __global float * output)
{
    int gid = get_global_id(0);
    output[gid] = input[gid];
    return;
}
```

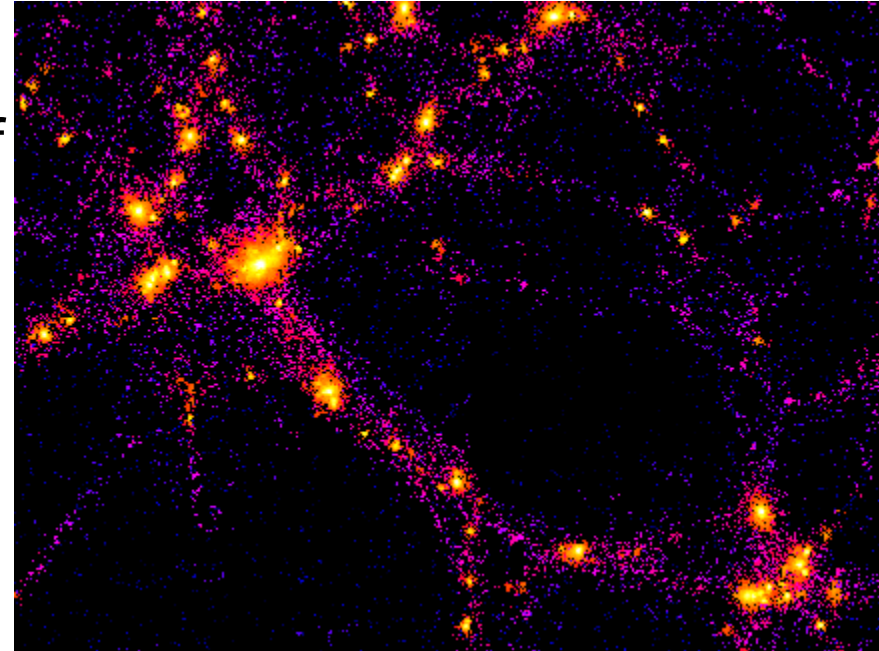


Outline

- OpenCL调试
 - CPU调试
 - GPU调试
- OpenCL性能优化
 - 线程映射
 - 设备占用
 - 向量化
- **N-body仿真实例**

N-body 仿真

- 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

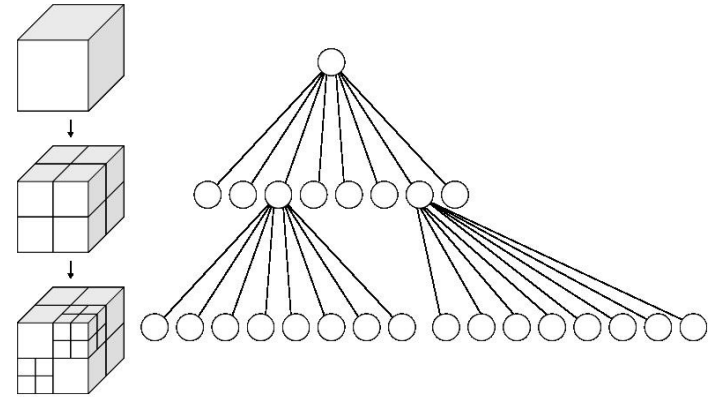
For each particle this becomes

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

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

N-body Algorithms

- For large counts, the previous method calculates the 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];

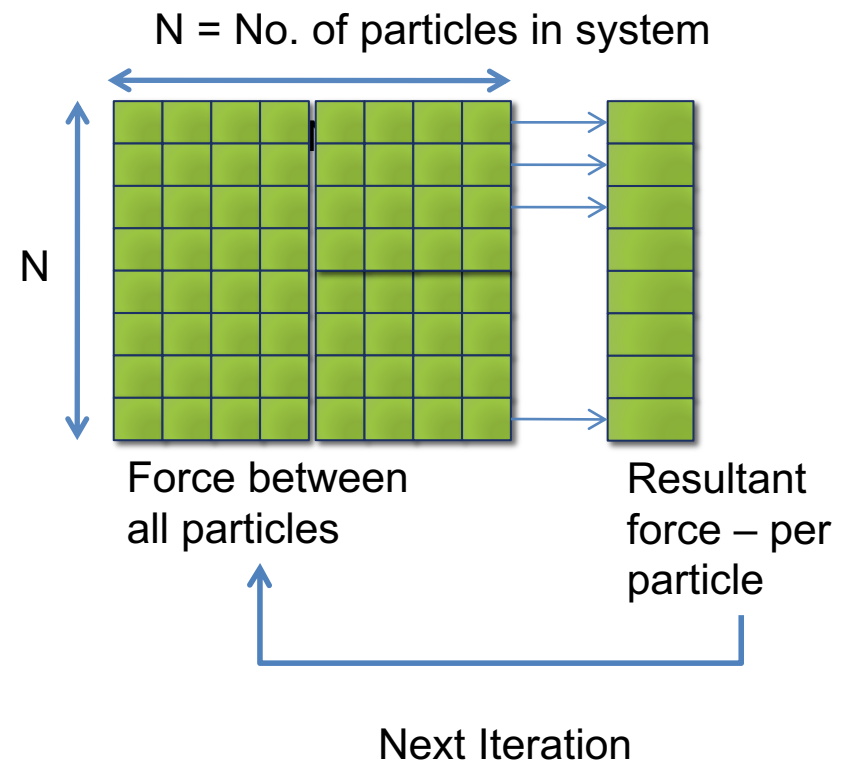
        // small eps is delta added for
        dx,dy,dz = 0
        invr= 1.0/sqrt(dx*dx+dy*dy+dz*dz
+eps);

        invr3 = invr*invr*invr;
        f=m[ j ]*invr3;

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

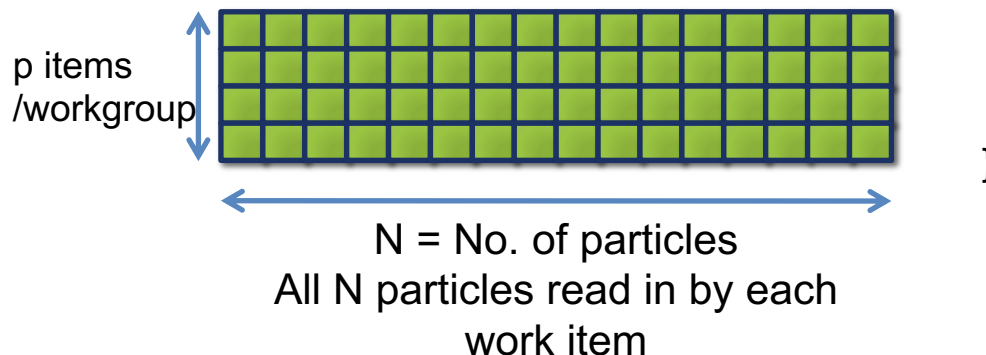
Parallel Implementation

- 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



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 \times N threads = N^2



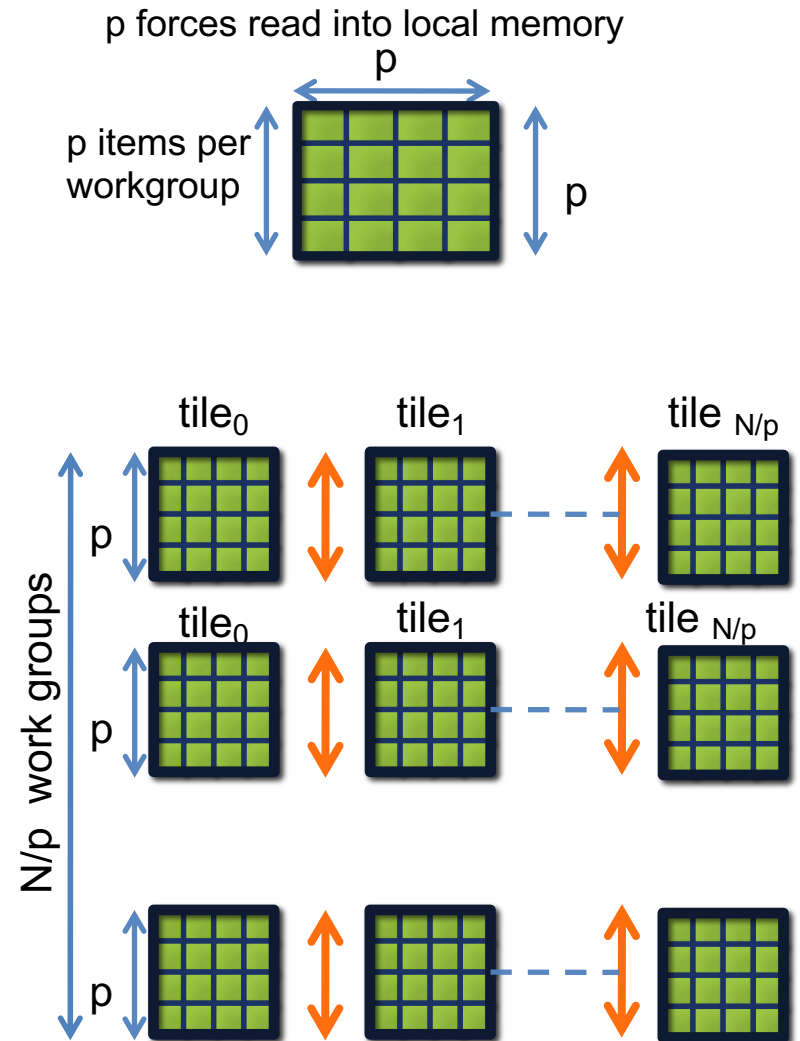
```
__kernel void nbody(
    __global float4 * initial_pos,
    __global float4 * final_pos,
    Int N, __local float4 * result) {

    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 $N \cdot p$ items read per work group
 - With reuse $p \cdot (N/p) = N$ 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);
}
```

Thread Mapping and Vectorization

- As discussed previously,
 - 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

Execution Time – Unrolled Kernels – with data reuse

