OpenCL
A Standard Platform for programming Heterogeneous parallel computers

Tim Mattson, Intel
Ian Buck, Nvidia
Michael Houston, AMD
Ben Gaster, AMD

Preliminaries:

- **Disclosures**
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are not speaking for our employers.
    - We are not speaking for Khronos

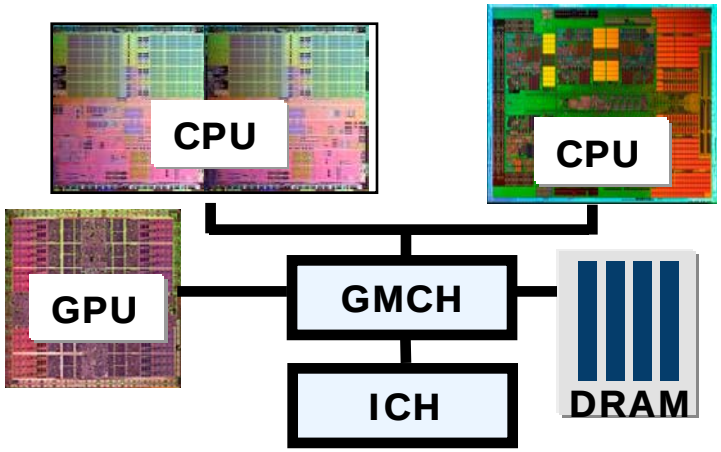
- **We take our tutorials VERY seriously:**
  - Help us improve … give us feedback and tell us how you would make this tutorial even better.

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

Heterogeneous computing

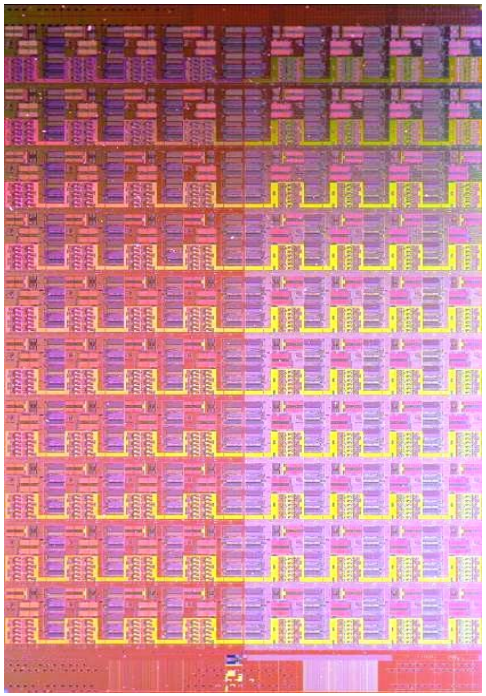
- **A modern platform has:**
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?



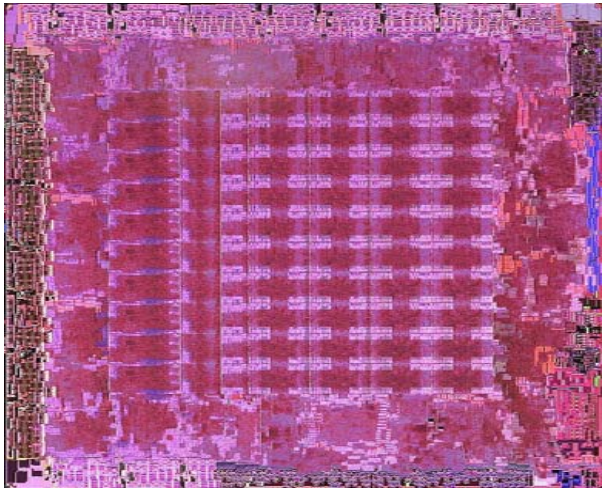
- **Programmers need to make the best use of <u>all</u> the available resources from within a <u>single</u> program:**
  - One program that runs well (i.e. reasonably close to “hand-tuned” performance) on a heterogeneous mixture of processors.

Microprocessor trends

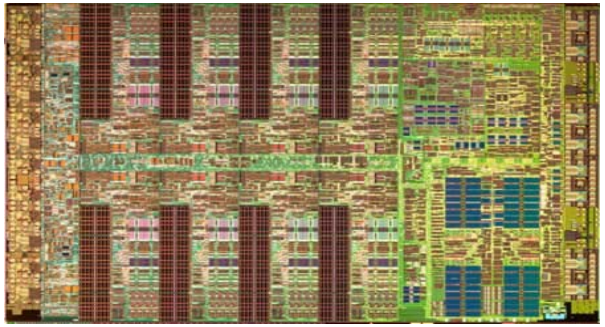
Individual processors are many core (and often heterogeneous) processors.



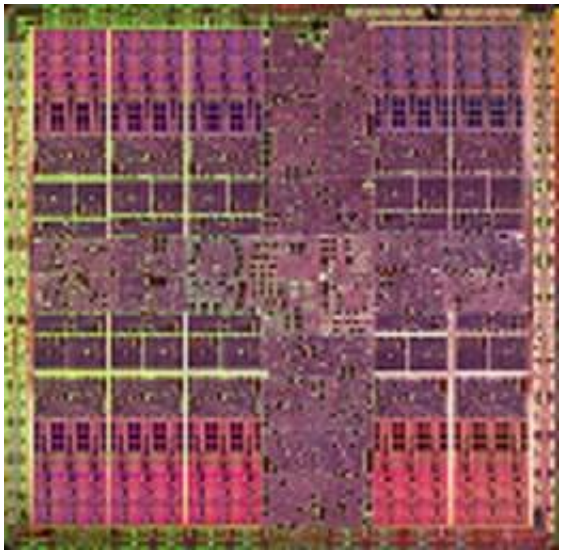
Intel 80 core research chip



ATI RV770

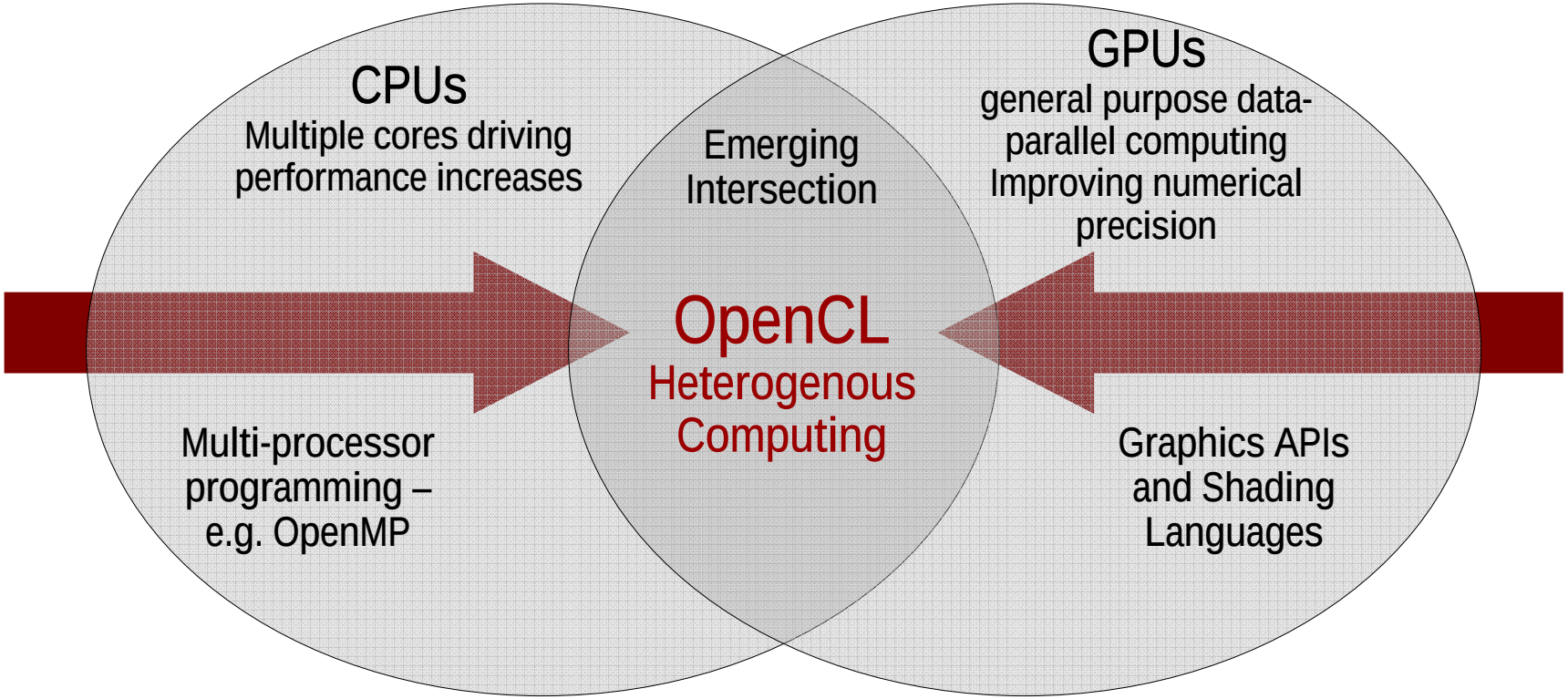


IBM Cell



NVIDIA Tesla C1060

How to program these new platforms?



OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL Working Group within Khronos

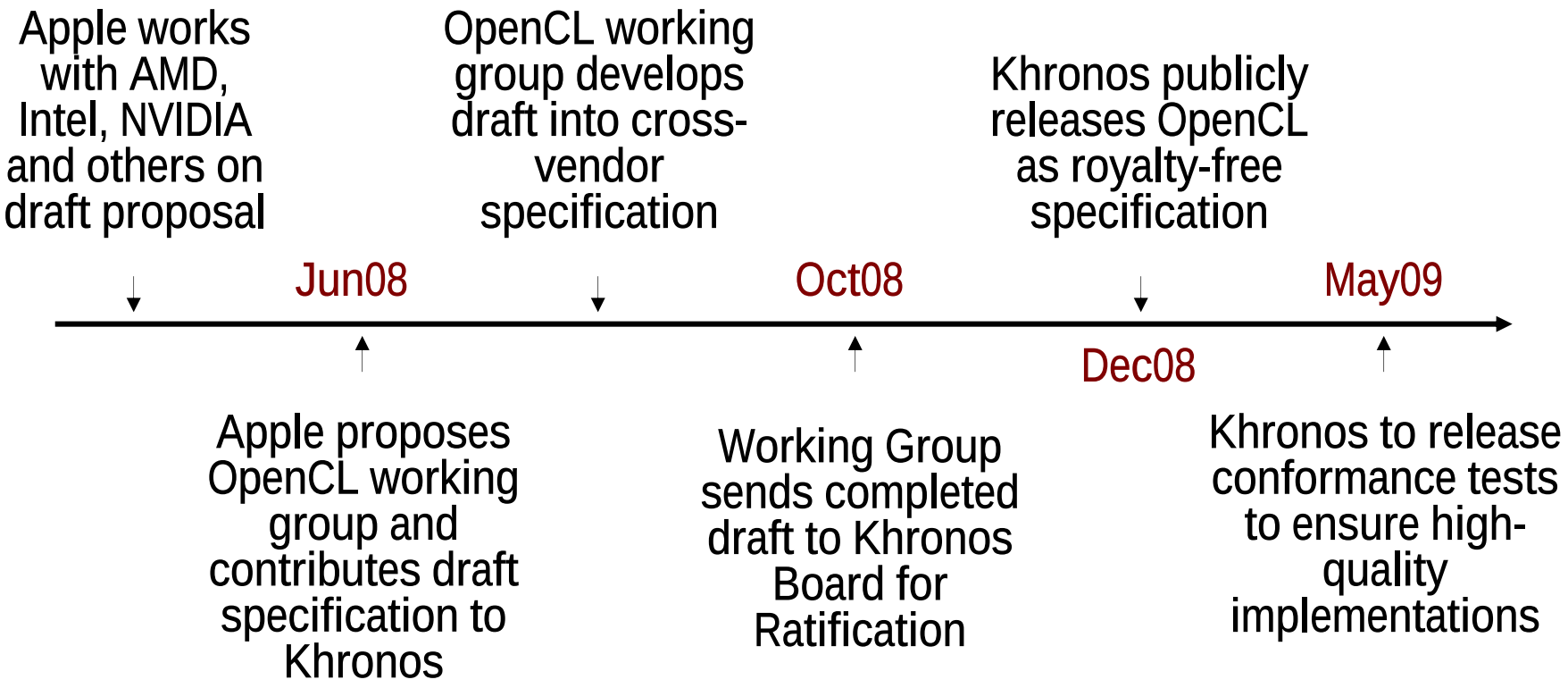
- **Diverse industry participation …**
  - Processor vendors (e.g. Apple), system OEMs, middleware vendors, application developers.
- **OpenCL became an important standard “on release” by virtue of the market coverage of the companies behind it.**

OpenCL: From cell phone to supercomputer

- **OpenCL Embedded profile for mobile and embedded silicon**
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate "ES" specification
- **Khronos APIs provide computing support for imaging & graphics**
  - Enabling advanced applications in, e.g., Augmented Reality
- **OpenCL will enable parallel computing in new markets**
  - Mobile phones, cars, avionics

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

OpenCL Timeline



OpenCL: Now and Future

- **Where to go to use OpenCL today!**
  - Apple's Mac OS X 10.6 (Snow Leopard) includes OpenCL
  - Nvidia GPU Release
  - AMD CPU/GPU Release
  - … and others over the next 12 months

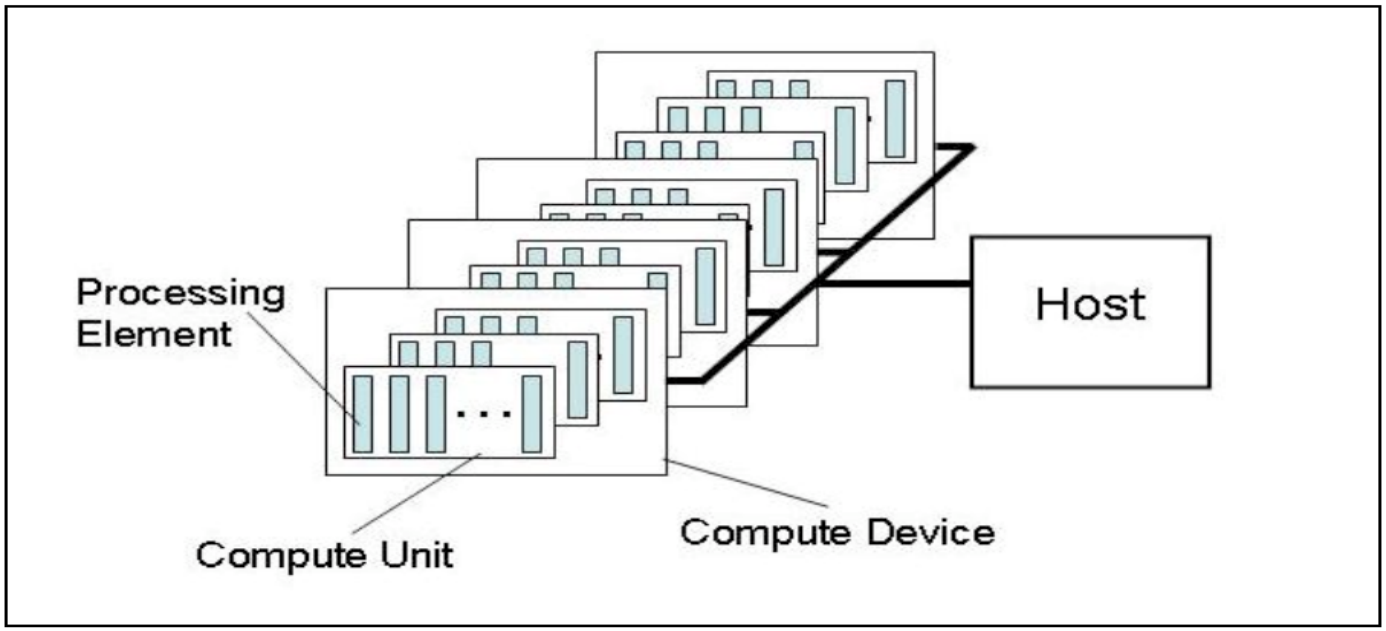
We need to update language for what to call the Nvidia and AMD releases

- **What next for OpenCL?**
  - OpenCL will evolve as processor architecture evolves
    - OpenCL 1.1 in late stages of definition … release first half of 2010.
    - OpenCL 2.0 work has begun … release around 2012
  - OpenCL will develop side by side with key graphics standards such as OpenGL

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

OpenCL Platform Model



- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

Execution model:

- OpenCL execution model … define a problem domain and execute a kernel invocation for each point in the domain
  - E.g., process a 1024 x 1024 image: Global problem dimensions:
    1024 x 1024 = 1 kernel execution per pixel: 1,048,576 total kernel executions

Scalar

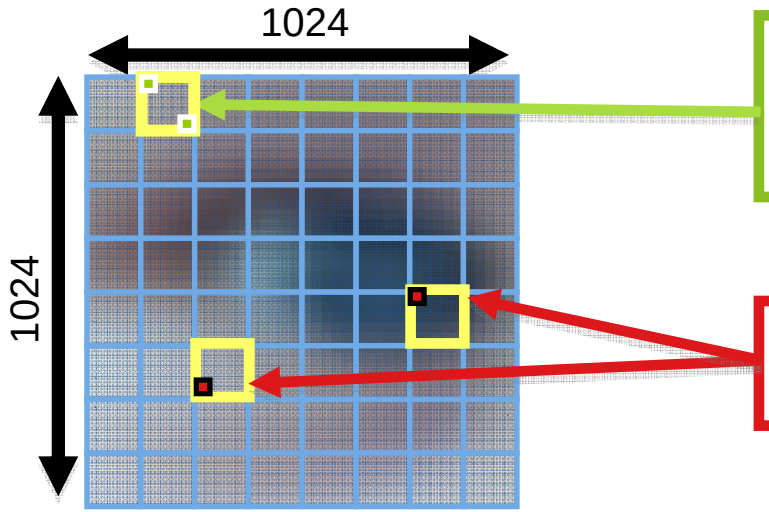
```
void
scalar_mul(int n,
           const float *a,
           const float *b,
           float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over “n” work-items
```

An N-dimension domain of work-items

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (executed together)

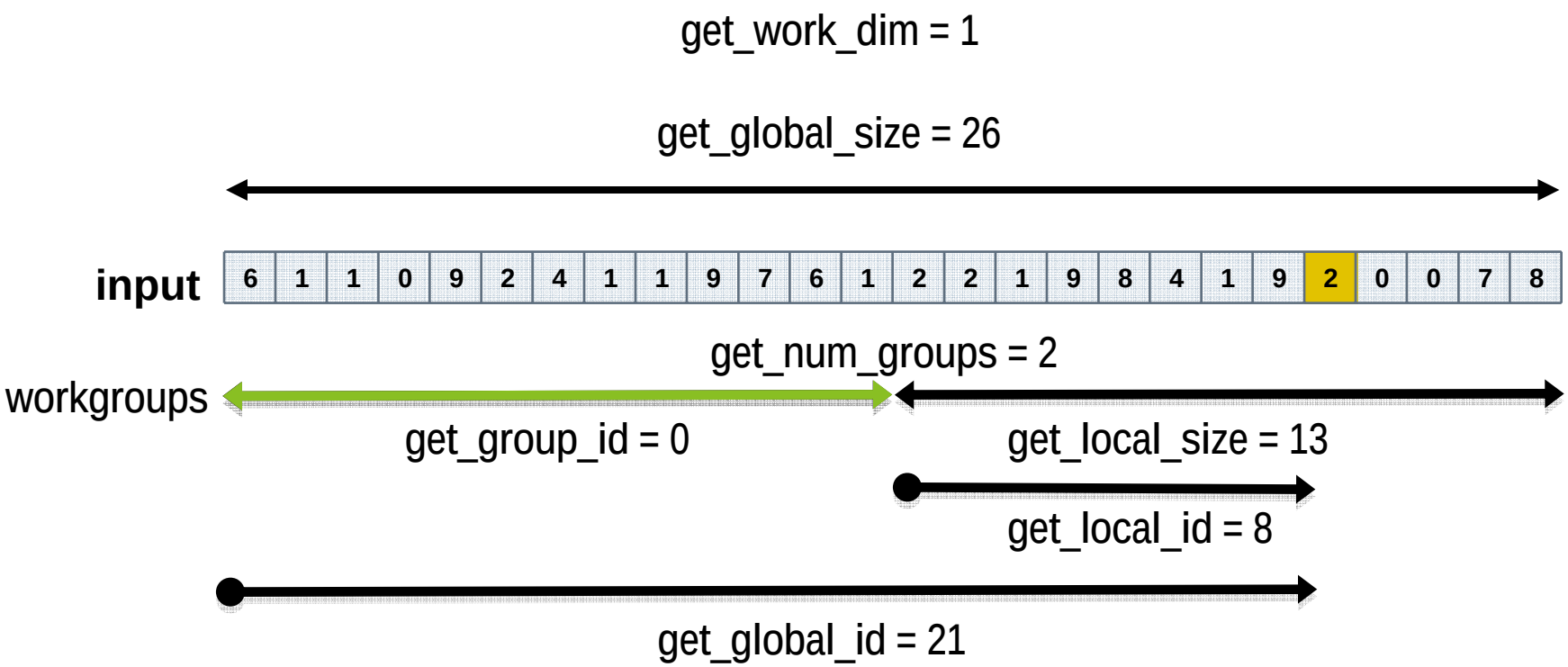


Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

- Choose the dimensions that are “best” for your algorithm

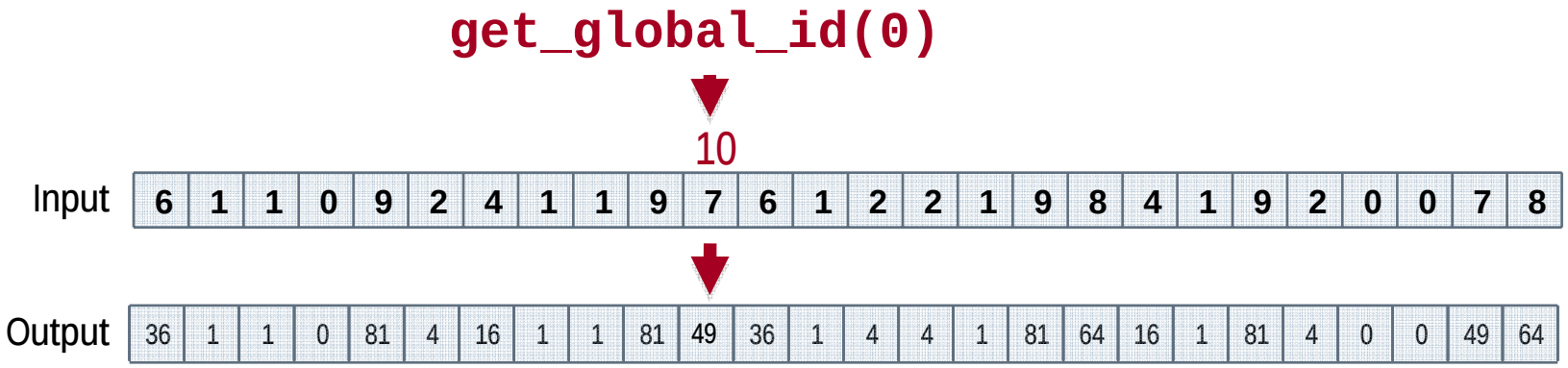
Examples: Work-Items and Workgroups



Kernel

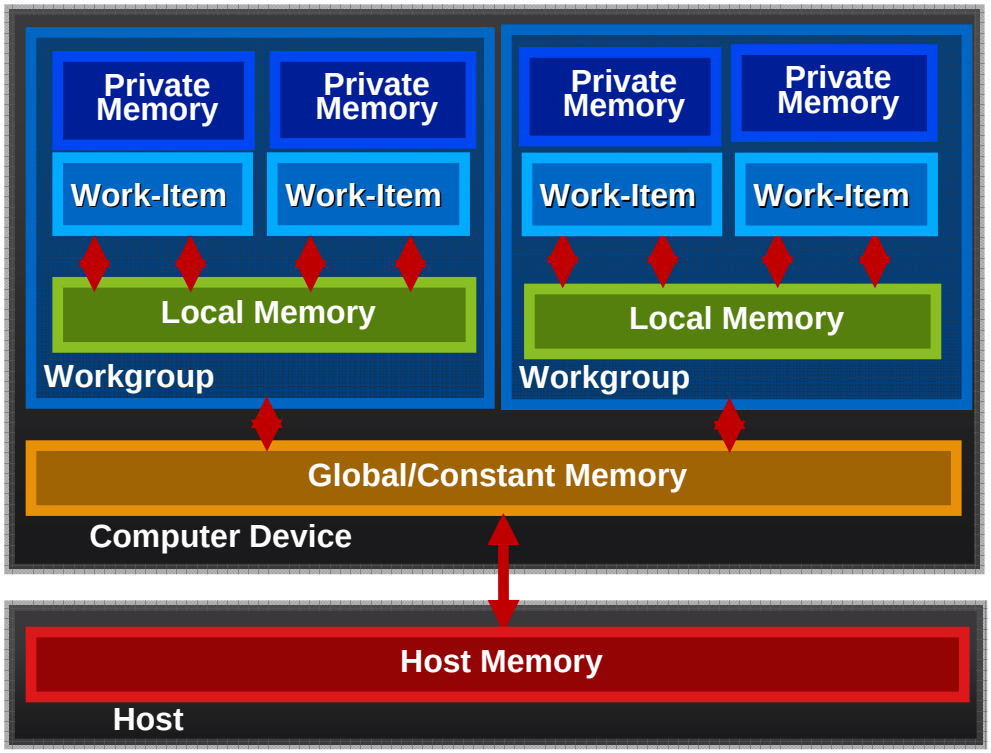
- A data-parallel function executed for each work-item

```
kernel void square(
       global float* input,
       global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```



OpenCL Memory Model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a workgroup (16Kb)
- Local Global/Constant Memory
  - Not synchronized
- Host Memory
  - On the CPU



- Memory management is explicit
  You must move data from host -> global -> local *and* back

Memory Consistency

- **“OpenCL uses a relaxed consistency memory model; i.e.**
  - the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.”
- **Within a work-item, memory has load/store consistency**
- **Within a work-group at a barrier, local memory has consistency across work-items**
- **Global memory is consistent within a work-group, at a barrier, but not guaranteed across different work-groups**
- **Consistency of memory shared between commands are enforced through synchronization**

OpenCL C Language

- Derived from ISO C99
  - No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- Additions to the language for parallelism
  - Work-items and workgroups
  - Vector types
  - Synchronization
- Address space qualifiers
- Optimized image access
- Built-in functions

Data Types

- Scalar data types
  - char , uchar, short, ushort, int, uint, long, ulong
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types

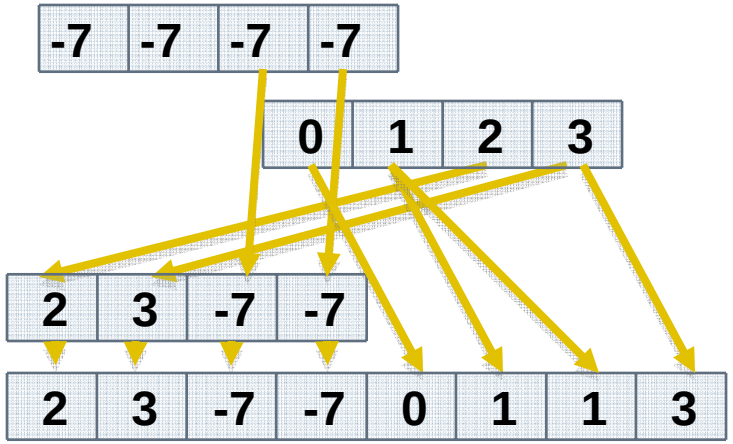
Vector Types

- Portable
- Vector length of 2, 4, 8, and 16
- char2, ushort4, int8, float16, double2, ...
- Endian safe
- Aligned at vector length
- Vector operations and built-in functions

Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

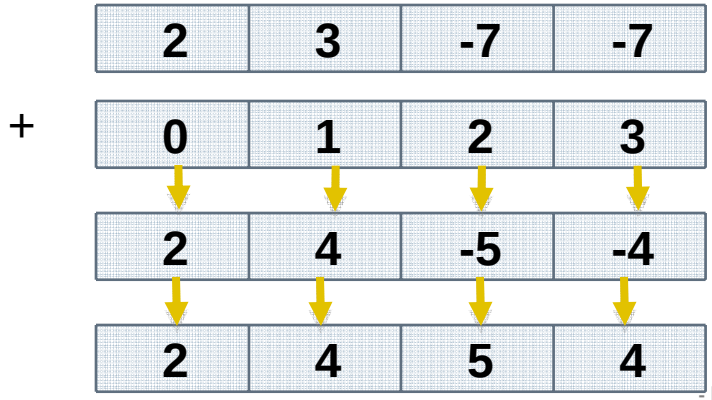


- Vector components

```
vi0.lo = vi1.hi;
int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```

- Vector ops

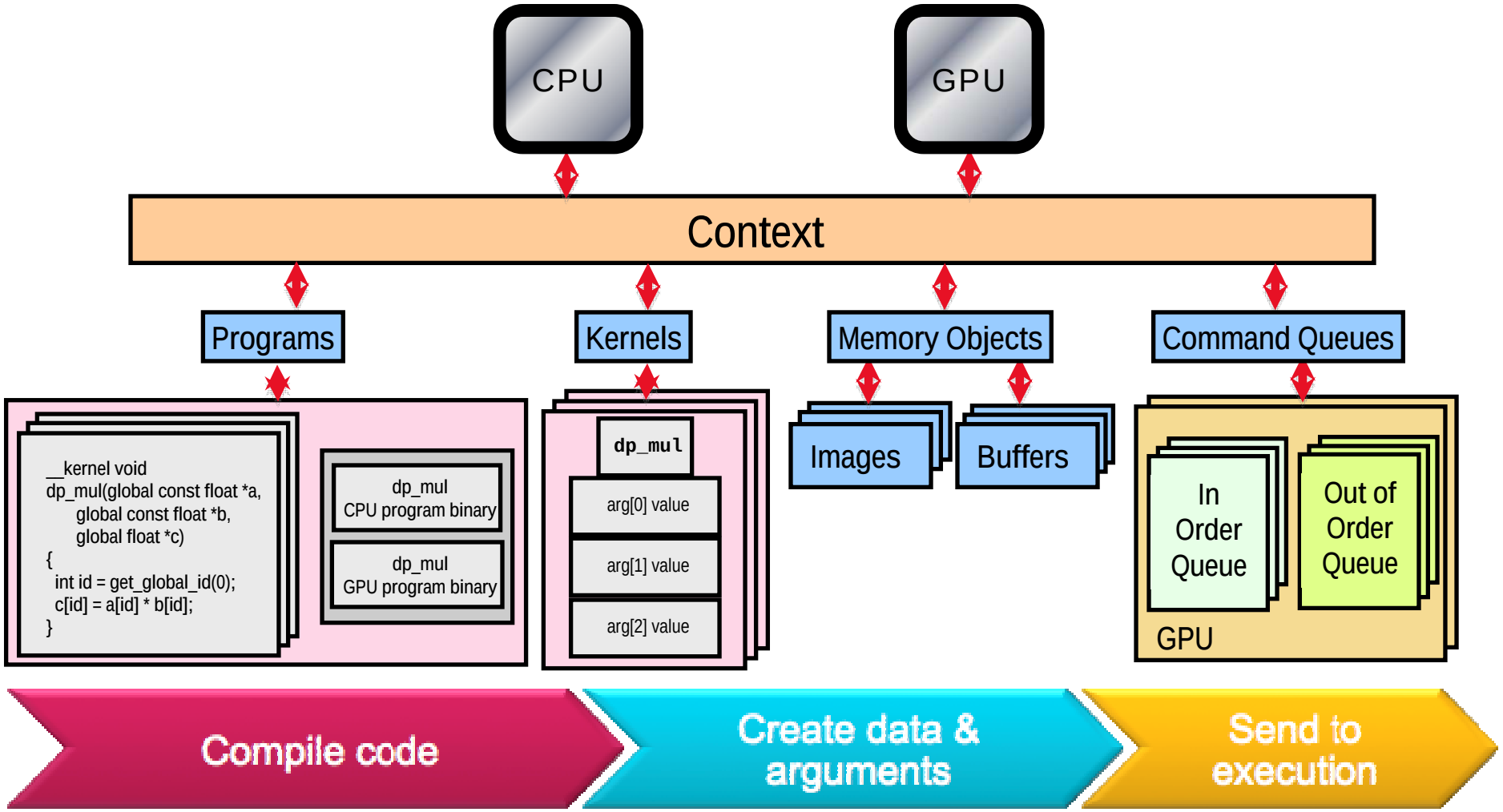
```
vi0 += vi1;
vi0 = abs(vi0);
```



Contexts and Queues

- **Contexts are used to contain and manage the state of the “world”**
- **Kernels are executed in contexts defined and manipulated by the host**
  - Devices
  - Kernels - OpenCL functions
  - Program objects - kernel source and executable
  - Memory objects
- **Command-queue** - coordinates execution of kernels
  - Kernel execution commands
  - Memory commands - transfer or mapping of memory object data
  - Synchronization commands - constrains the order of commands
- **Applications queue compute kernel execution instances**
  - Queued in-order
  - Executed in-order or out-of-order
  - Events are used to synchronize execution instances

OpenCL summary



Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

Example: vector addition

- **The “hello world” program of data parallel programming is a program to add two vectors**

  C[i] = A[i] + B[i] for i=1 to N

- **For the OpenCl solution, there are two parts**
  - Kernel code
  - Host code

Vector Addition - Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                       NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
                                       NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                       NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                          sizeof(cl_float)*n, NULL,
                                       NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                              NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                     sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE,
   0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Vector Addition - Host Program

Define platform and queues

Define Memory objects

```
READ_ONLY |
                                    srcA,
    NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
    NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL,
    NULL);
```

Create the program

Build the program

Create and setup kernel

```
// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                   sizeof(cl_mem));
```

Execute the kernel

```
// set work-item dimensions
// execu
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);
```

Read results on the host

```
// rea
err =
   0,
```

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

Platform Layer: Basic discovery

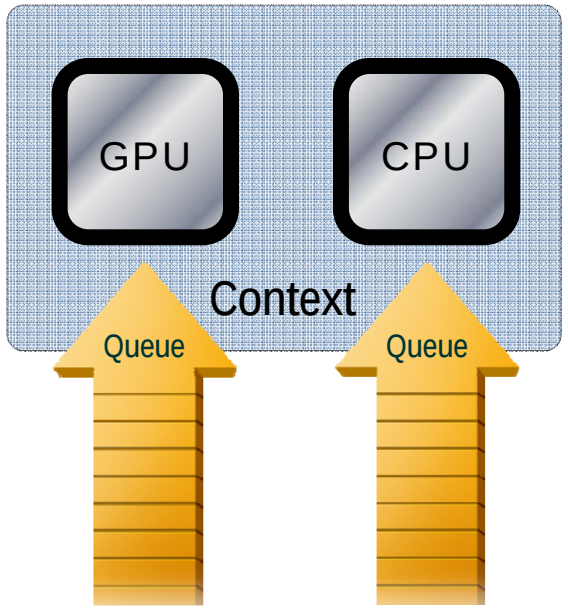
- **Platform layer allows applications to query for platform specific features**
- **Querying platform info Querying devices**
  - *clGetDeviceIDs*()
    - Find out what compute devices are on the system
    - Device types include CPUs, GPUs, or Accelerators
  - *clGetDeviceInfo*()
    - Queries the capabilities of the discovered compute devices such as:
      - Number of compute cores
      - Maximum work-item and work-group size
      - Sizes of the different memory spaces
      - Maximum memory object size

Platform Layer: Contexts

- **Creating contexts**
  - Contexts are used by the OpenCL runtime to manage objects and execute kernels on one or more devices
  - Contexts are associated to one or more devices
    - Multiple contexts could be associated to the same device
  - *clCreateContext*() and *clCreateContextFromType*() returns a *handle* to the created contexts

Platform layer: Command-Queues

- Command-queues store a set of operations to perform
- Command-queues are associated to a context
- Multiple command-queues can be created to handle independent commands that don’t require synchronization
- Execution of the command-queue is guaranteed to be completed at sync points



VecAdd: Context, Devices, Queue

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0, // (must be 0)
                       CL_DEVICE_TYPE_GPU,
                       NULL,  // error callback
                       NULL,  // user data
                       NULL); // error code

// get the list of GPU devices associated with context
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context,
                          devices[0],
                     0,     // default options
                     NULL); // error code
```

Memory Objects

- Buffers
  - Simple chunks of memory
  - Kernels can access however they like (array, pointers, structs)
  - Kernels can read and write buffers
- Images
  - Opaque 2D or 3D formatted data structures
  - Kernels access only via read_image() and write_image()
  - Each image can be read or written in a kernel, but not both

Creating Memory Objects

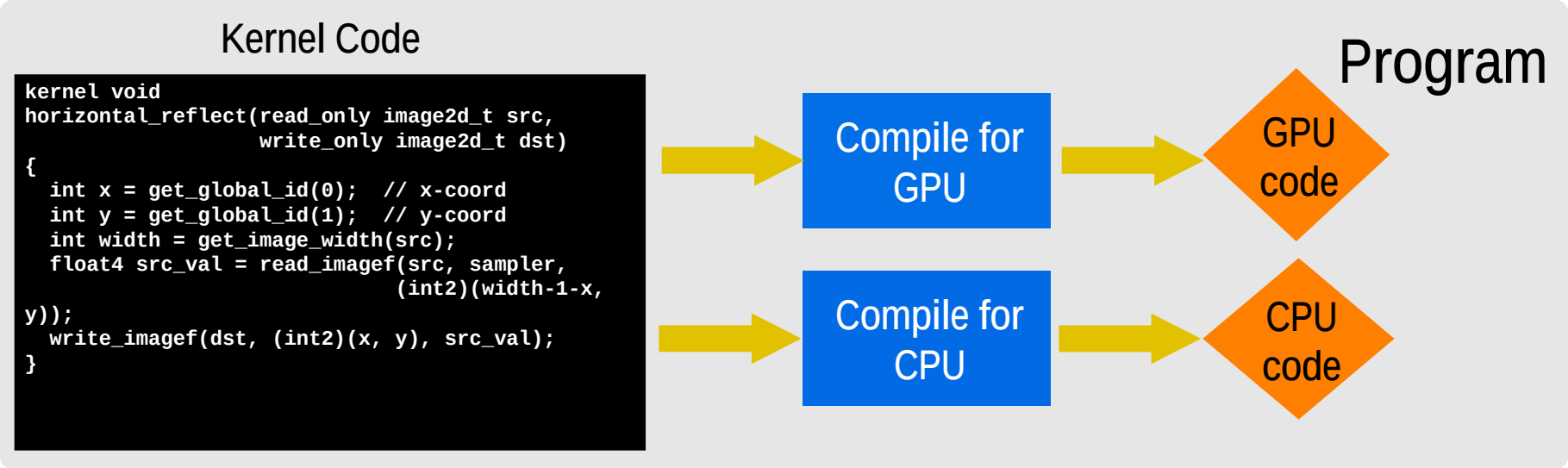
- **Memory objects are created with an associated context**
  - *clCreateBuffer()*, *clCreateImage2D*(), and *clCreateImage3D*()
- **Memory can be created as read only, write only, or read-write**
- **Where objects are created in the platform memory space can be controlled**
  - Device memory
  - Device memory with data copied from a host pointer
  - Host memory
  - Host memory associated with a pointer
    - Memory at that pointer is guaranteed to be valid at synchronization points

VecAdd: Create Memory Objects

```
cl_mem memobjs[3];
// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
                    CL_MEM_READ_ONLY |   // flags
                    CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float)*n,  // size
                    srcA,                // host pointer
                    NULL);               // error code
memobjs[1] = clCreateBuffer(context,
             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
             sizeof(cl_float)*n, srcB, NULL);

// allocate input buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);
```

Build the Program object

- **The program object encapsulates:**
  - A context
  - The program source/binary
  - List of target devices and build options
- **The Build process … to create a program object**
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()



VecAdd: Create and Build the Program

```
// create the program
cl_program program = clCreateProgramWithSource(
                context,
                1,                // string count
                &program_source, // program strings
                NULL,             // string lengths
                NULL);            // error code
// build the program
cl_int err = clBuildProgram(program,
                0,     // num devices in device list
                NULL,  // device list
                NULL,  // options
                NULL,  // notifier callback function ptr
                NULL); // user data
```

Kernel Objects

- **Kernel objects encapsulate**
  - Specific kernel functions declared in a program
  - Argument values used for kernel execution
- **Creating kernel objects**
  - *clCreateKernel*() - creates a kernel object for a single function in a program
- **Setting arguments**
  - *clSetKernelArg*(<kernel>, <argument index>)
  - Each argument data must be set for the kernel function
  - Argument values copied and stored in the kernel object
- **Kernel vs. program objects**
  - Kernels are related to program execution
  - Programs are related to program source

VecAdd: Create the Kernel and Set the Arguments

```
// create the kernel
cl_kernel kernel = clCreateKernel(program, “vec_add”, NULL);

// set “a” vector argument
err  = clSetKernelArg(kernel,
                0,                      // argument index
                (void *)&memobjs[0], // argument data
                sizeof(cl_mem));     // argument data size

// set “b” vector argument
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                 sizeof(cl_mem));

// set “c” vector argument
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                 sizeof(cl_mem));
```

Kernel Execution

- **A command to execute a kernel must be enqueued to the command-queue**
  - Command-queue could be explicitly flushed to the device
  - Command-queues execute in-order or out-of-order
    - In-order - commands complete in the order queued and correct memory is consistent
    - Out-of-order - no guarantee when commands are executed or memory is consistent without synchronization
- ***clEnqueueNDRangeKernel*()**
  - Data-parallel execution model
  - Describes the ***index space*** for kernel execution
  - Requires information on NDRange dimensions and work-group size
- ***clEnqueueTask*()**
  - Task-parallel execution model (multiple queued tasks)
  - Kernel is executed on a single work-item
- ***clEnqueueNativeKernel*()**
  - Task-parallel execution model
  - Executes a native C/C++ function not compiled using the OpenCL compiler
  - This mode does not use a kernel object so arguments must be passed in

VecAdd: Invoke Kernel

```
size_t global_work_size[1] = n; // set work-item dimensions
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                      1,          // Work dimensions
                      NULL,       // must be NULL (work offset)
                     global_work_size,
                      NULL,       // automatic local work size
                      0,          // no events to wait on
                      NULL,       // event list
                      NULL);      // event for this kernel
```

Synchronization

- **Synchronization**
  - Signals when commands are completed to the host or other commands in queue
  - Blocking calls
    - Commands that do not return until complete
    - clEnqueueReadBuffer() can be called as blocking and will block until complete
  - *Event objects*
    - Tracks execution status of a command
    - Some commands can be blocked until event objects signal a completion of previous command
      - clEnqueueNDRangeKernel() can take an event object as an argument and wait until a previous command (e.g., clEnqueueWriteBuffer) is complete
  - Queue barriers - queued commands that can block command execution

VecAdd: Read Output

```
// read output array
err = clEnqueueReadBuffer( context, memobjs[2],
                      CL_TRUE,              // blocking
                      0,                    // offset
                      n*sizeof(cl_float), // size
                      dst,                  // pointer
                      0, NULL, NULL);       // events
```

OpenCL C for Compute Kernels

- **Derived from ISO C99**
  - A few restrictions: recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported
- **Built-in Data Types**
  - Scalar and vector data types, Pointers
  - Data-type conversion functions: convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t
- **Built-in Functions — Required**
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
- **Built-in Functions — Optional**
  - double precision, atomics to global and local memory
  - selection of rounding mode, writes to image3d_t surface

OpenCL C Language Highlights

- **Function qualifiers**
  - "__kernel" qualifier declares a function as a kernel
  - Kernels can call other kernel functions
- **Address space qualifiers**
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier
- **Work-item functions**
  - Query work-item identifiers
    - get_work_dim(), get_global_id(), get_local_id(), get_group_id()
- **Synchronization functions**
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations

OpenCL C Language Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer of types less than 32-bit are not supported
- Double types are not supported, but reserved

Vector Addition Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
 }
```

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

Agenda

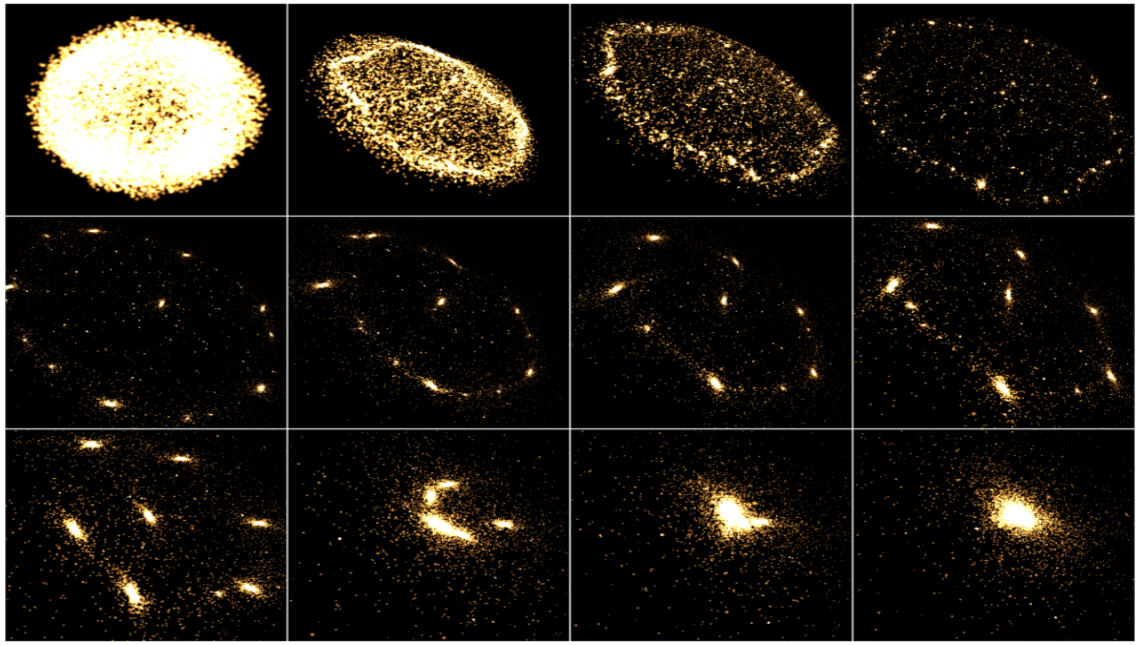
- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

N-Body Simulation

- **Numerically Simulate evolution of system of N bodies**
  - Each body continuously interacts with all other bodies
- **Examples:**
  - Astronomical and astrophysical simulation
  - Molecular dynamics simulation
  - Fluid dynamics simulation
  - Radiometric transfer (Radiosity)
- **N^2 interactions to compute per time step**
  - For the brute force all-pairs approach we discuss here

Astrophysics N-Body Simulation

- **OpenCL All-Pairs N-Body Gravitation Simulation**
- **NVIDIA GeForce GTX 280 GPU:**
  - More than 20B body-body interactions per second
  - 400+ GFLOP/s, 16K bodies at 75+ FPS
    - 20 FLOPS per interaction, $16K^2$ interactions per frame
- **Highly Parallel**
- **High Arithmetic Intensity**



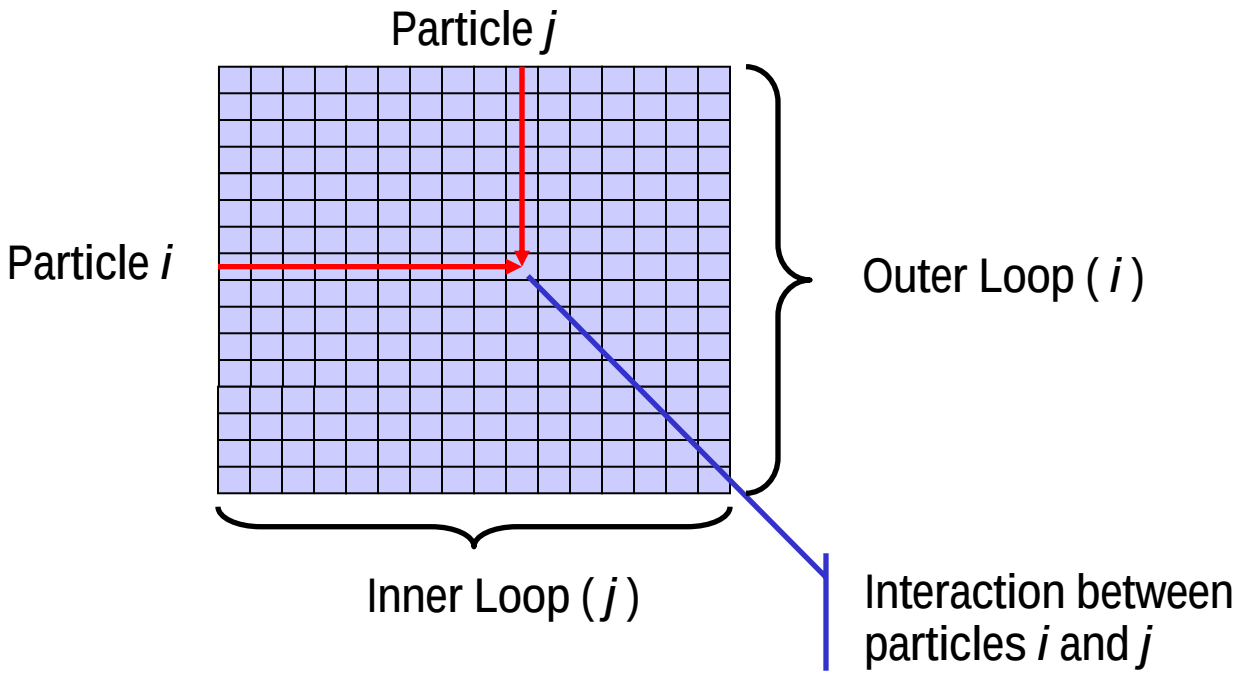
Sequential N-Body Algorithm

```
foreach body i {   // outer “i” loop
  accel = 0;
  pos = position[i];
  foreach body j { // inner “j” loop
    accel += computeAcceleration( pos,
                                  position[j]);
  }
  // Leapfrog-Verlet integration*
  velocity[i] += accel * timestep;
  position[i] += velocity[i] * timestep;
}
```

*May use other integration schemes

Sequential N-Body Algorithm

- **Conceptual grid of interactions between (*i*,*j*) pairs**



Approach to N-Body Parallelism

- **Acceleration on all bodies can be computed in parallel in OpenCL**
- **One work item per body**
  - *N* / *p* work groups of *p* work items process *p* bodies at a time

```
forall bodies i in parallel {
  accel = 0;
  pos = position[i];

  foreach body j {
    accel += computeAcceleration(pos, position[j]);
  }
}
```

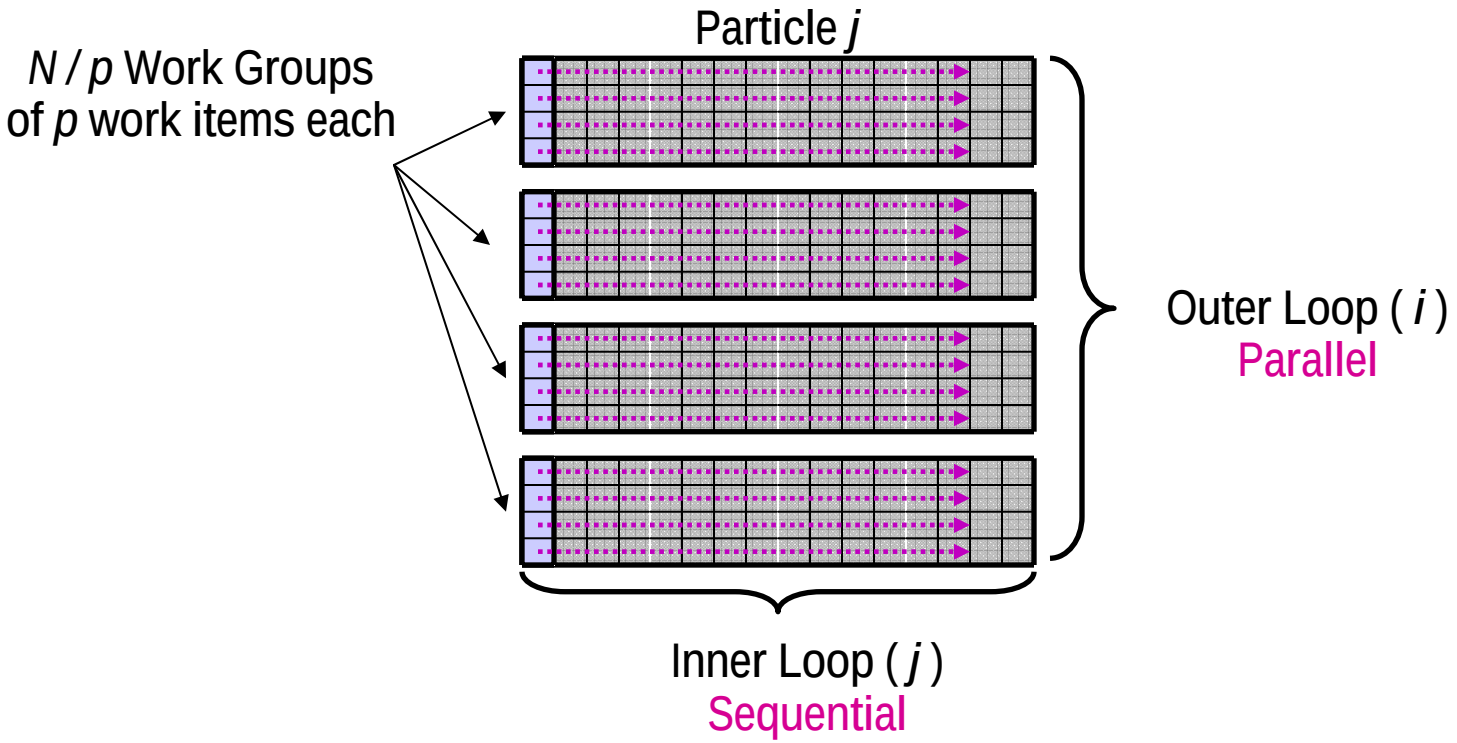
Naïve Parallel Approach

```
forall bodies i in parallel {
  accel = 0;
  pos = position[i]
  foreach body j {
    accel += computeAcceleration(pos,
                                 position[j])
  }
}
```

- Every thread loads all other body positions from off-chip memory
  - N^2 loads …. Would be bandwidth bound = poor performance!
  - 100 GB/s peak / 16 bytes per position = 6.25B interactions/s *theoretical* peak
  - =125 GFLOP/s ≈ 1/3 of what GeForce 280 GTX achieves on our code

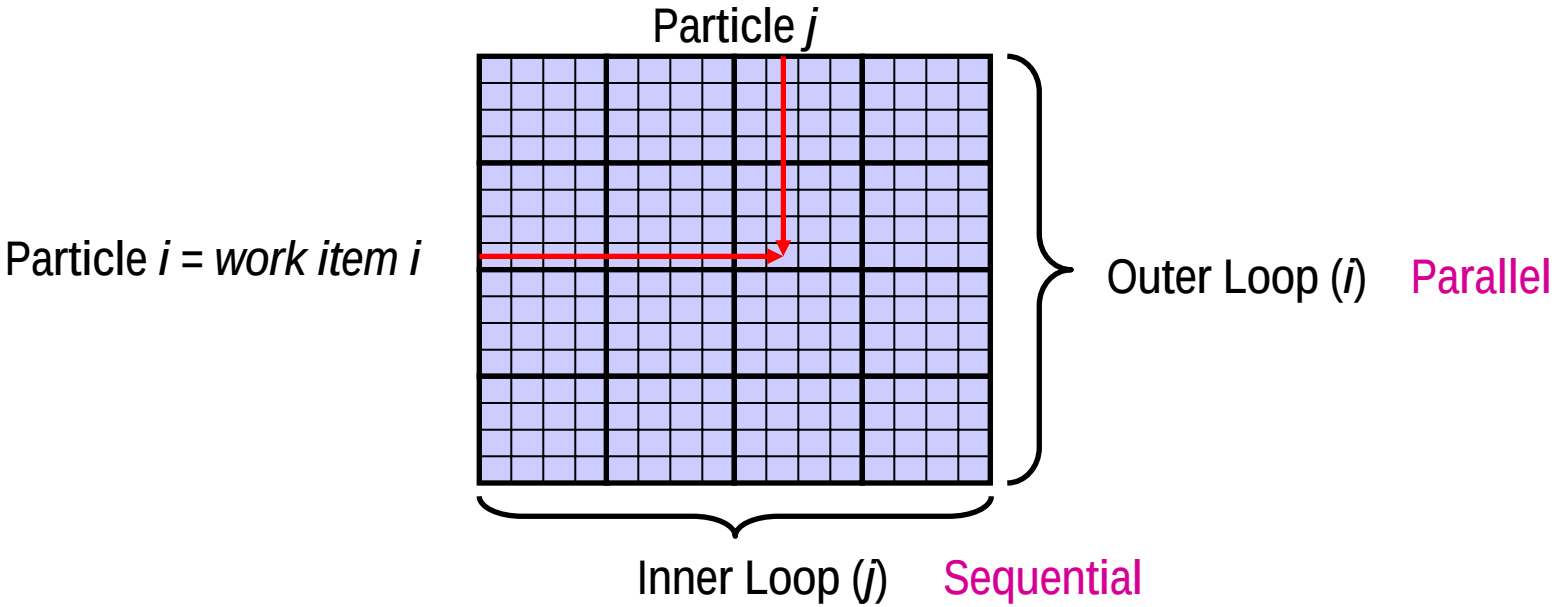
Naïve Parallel Implementation

- ***N* total work items divided into *N / p* work groups in OpenCL**



Tiling: Solving the B/W Bottleneck

- **Each body position is used by all work items**
  - Cache the loads on chip
  - GPUs have fast on-chip shared memory, CPUs have on-chip caches
- Idea: Break grid into conceptual *tiles*
  - share blocks of body positions between work items (threads)



Tiled Parallel Approach

```
forall bodies i in parallel {
  accel = 0;
  pos = position[i];
  foreach tile q {
    forall work_items p in work_group in parallel {
      local[p] = position[q*tile_size + p];
    }
    synchronize work items in work group
    foreach body j in tile q {
      accel += computeAcceleration(pos, local[j]);
    }
    synchronize work items in work group
  }
}
```

- **Sequential inner loop divided into *N* / *p* sub-loops over tiles**
  - Work items in a work group cooperatively load *p* positions within tile to local mem
- **Reduces number of loads to N^2 / *p***
  - Typically use *p* = 256 work items, so big savings!
  - Compute bound, good performance: 20B+ interactions/s = 400+ GFLOP/s

Body-Body Gravitation

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\left\| \mathbf{r}_{ij} \right\|^2} \times \frac{\mathbf{r}_{ij}}{\left\| \mathbf{r}_{ij} \right\|},$$

$$\mathbf{A}_i = G \sum_{0 < j < N} \frac{m_j \mathbf{r}_{ij}}{\left(\left\| \mathbf{r}_{ij} \right\|^2 + e^2 \right)^{\frac{3}{2}}}$$

- $\mathbf{A}_i$: Acceleration on body i
- m_j: Mass of body j
- $\mathbf{r}_{ij}$: Vector pointing from body i to body j
- ε: Softening Factor: used to preclude body-body collisions and divide by zero
- G: Gravitational Constant

OpenCL N-body Source Walkthrough

- **More involved than previous example**
  - **Demonstrates:**
    - Local memory
    - Explicit Work group sizing
    - OpenGL vertex buffer interop
  - **Outline of walkthrough**
    - Kernel code
    - Creation of CL memory objects from GL buffer objects
    - Mapping / unmapping GL buffer objects into OpenCL
    - Setting arguments, notably local memory
    - Kernel invocation

Kernel: Single Body Pair Gravitation

```
float4 bodyBodyInteraction(float4 bi, float4 bj, float softeningSq)
{    // r_ij  [3 FLOPS]
    float4 r;
    r.x = bi.x–b.x; r.y = bi.y–bj.y; r.z = bi.z–bj.z; r.w = 0;

    // distSqr = dot(r_ij, r_ij) + EPS^2  [6 FLOPS]
    float distSqr = bi.x*bj.x + bi.y*bj.z + bi.z*bj.z + softeningSquared;

    // invDistCube =1/distSqr^(3/2)  [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
    float invDist = rsqrt(distSqr);
    float invDistCube =  invDist * invDist * invDist;

    // s = m_j * invDistCube [1 FLOP]
    float s = bj.w * invDistCube;
    // accel =  s * r_ij [3 FLOPS]
    r.x *= s; r.y *= s; r.z *= s;
    return r * s;
    // + 3 FLOPS on return to accumulate acceration
 }
```

Total: 20 FLOPS

Kernel: Acceleration Within a Tile

```
float4 gravitation(float4 myPos,
                   __local float4* sharedPos,
                   float softeningSq)
{
    unsigned int i = 0;
    float4 accel = (float4){0.0f};

    for (unsigned int i = 0; i < get_local_size(0); i++;)
    {
        float4 a
        a = bodyBodyInteraction(sharedPos[i], myPos);
        accel.x += a.x; accel.y += a.y; accel.z += a.z;
    }
    return accel;
}
```

Kernel: Integrate All Bodies (1)

```
__kernel void
integrateBodies(__global float4* outPos, __global float4* outVel,
                __global float4* inPos,  __global float4* inVel,
                float deltaTime, float damping, float softSq
                __local  float4* sharedPos)
 {
    float4 pos    = inPos[get_global_id(0)];
   float4 accel  = (float4){0.0f};
   for (int tile = 0; tile < get_num_groups(0); tile++)
   {
      sharedPos[get_local_id(0)] =
         inPos[tile*get_local_size(0) + get_local_id(0)];
       barrier(CLK_LOCAL_MEM_FENCE);
       accel += gravitation(bodyPos, sharedPos);
       barrier(CLK_LOCAL_MEM_FENCE);
   }
```

Kernel: Integrate All Bodies (2)

```
    // integrate, using acceleration = force \ mass;
    // factor out body's mass from the equation,
    // so force == acceleration
    float4 vel = inVel[get_global_id(0)];

    vel += accel * deltaTime * damping;
    pos += vel * deltaTime;

    // store new position and velocity
    outPos[get_global_id(0)] = pos;
    outVel[get_global_id(0)] = vel;
}
```

Host: Set Kernel Arguments (1)

```
void integrateNbodySystem(
      cl_context ctx,  cl_command_queue cmd_queue,
      cl_mem newPos,   cl_mem newVel,
      cl_mem oldPos,   cl_mem oldVel,
      float deltaTime, float damping, float softSq,
      unsigned int numBodies, unsigned int workGroupSize)
{
    cl_int err;
    err = clSetKernelArg(kernel, 0,
                         sizeof(cl_mem), (void *)&newPos);
    err |= clSetKernelArg(kernel, 1,
                          sizeof(cl_mem), (void *)&newVel);
    err |= clSetKernelArg(kernel, 2,
                          sizeof(cl_mem), (void *)&oldPos);
    err |= clSetKernelArg(kernel, 3,
                          sizeof(cl_mem), (void *)&oldVel);
```

Host: Set kernel arguments (2)

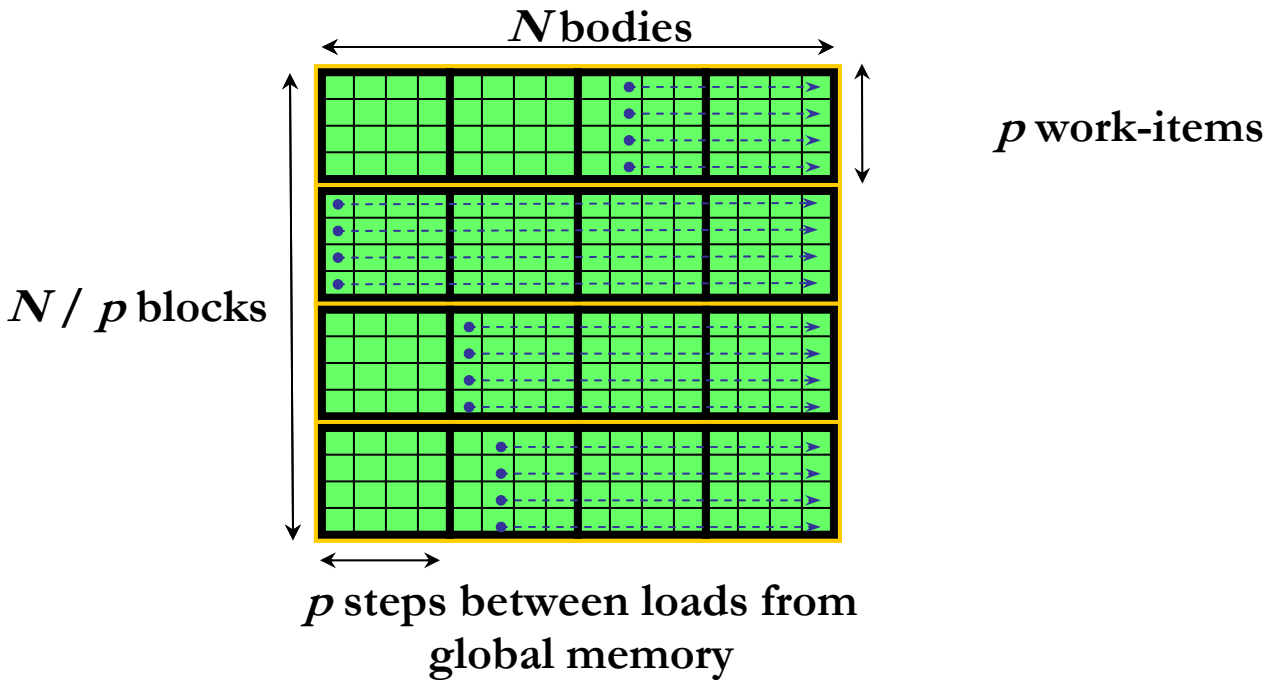
```
err |= clSetKernelArg(kernel, 4,
                      sizeof(cl_float), (void *)&deltaTime);
err |= clSetKernelArg(kernel, 5,
                      sizeof(cl_float), (void *)&damping);
err |= clSetKernelArg(kernel, 6,
                      sizeof(cl_float), (void *)&softSq);

// 4 floats for position
int localMemSize = workGroupSize * 4 * sizeof(float);

// Set the local memory size for __local ptr argument
err |= clSetKernelArg(kernel, 8, localMemSize, NULL);

if (err) { /* handle error */ }
```

Host: Invoke Kernel

```
size_t global_work_size = workGroupSize;
size_t local_work_size = numBodies/workGroupSize;

// execute the kernel:
clEnqueueNDRangeKernel(
    cmd_queue,
    kernel,
    1,                  //cl_uint work_dim
    NULL,               //const size_t *global_work_offset
    global_work_size,
    local_work_size,
    0,                  //cl_uint num_events_in_wait_list,
    NULL,               // const cl_event *event_wait_list,
    NULL                //cl_event *event
    );
```

Algorithmic Approaches

- **Nyland *et al.* (2007)**



Extending to protein folding

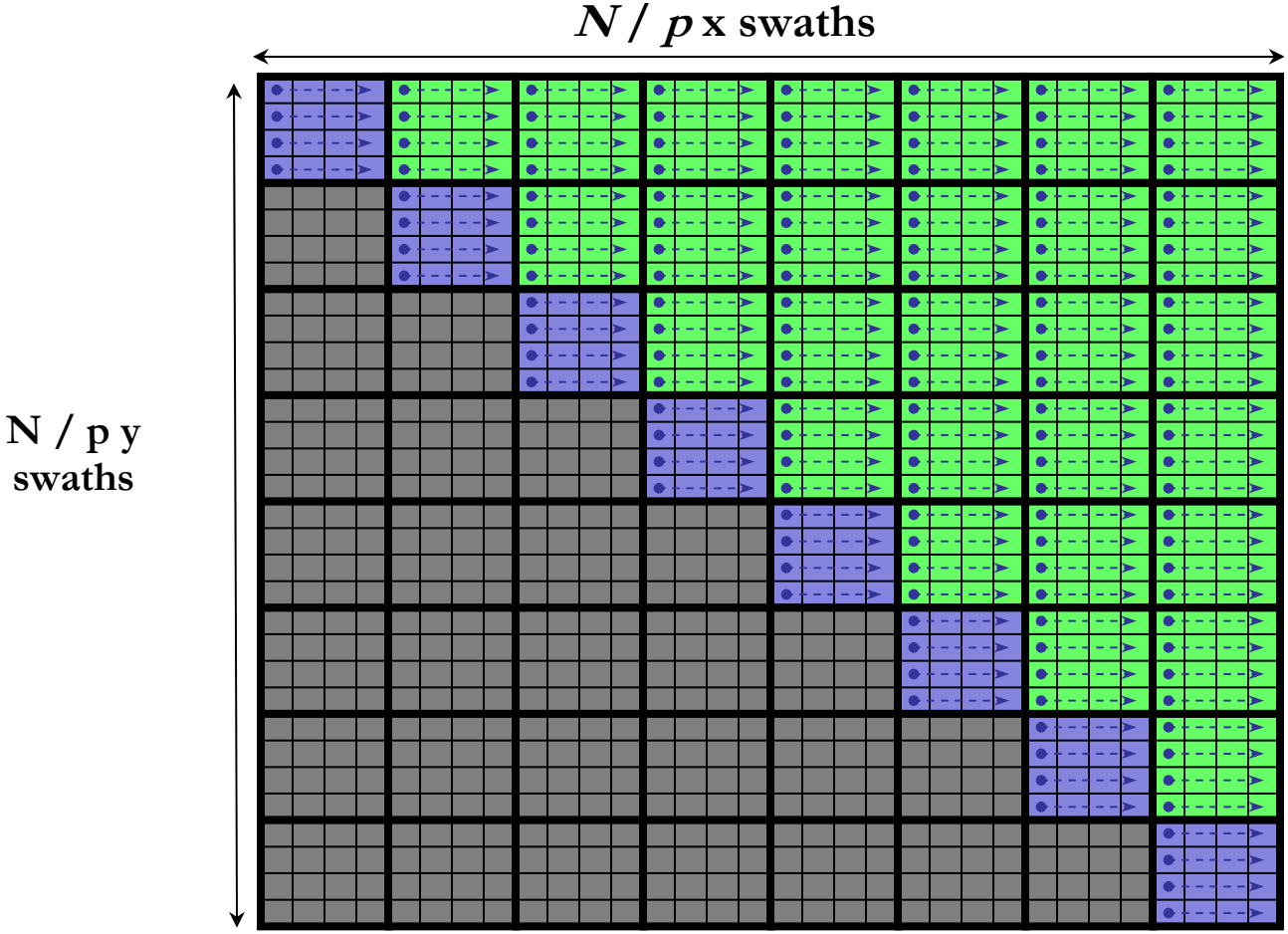
Note that f_{ij} uses the same data as f_{ji}

$$f_{ij} = -\frac{q_i q_j (1.0 - 0.25 * e^{-D_{ij}})}{\left(r_{ij}^2 + r_i^{Born} r_j^{Born} * e^{-D_{ij}}\right)^{1.5}}$$

So if we calculate them both at the same time:

$$\frac{n(n+1)}{2} \text{ as opposed to } n^2 \text{ calculations}$$

We really only need to process blue and green tiles

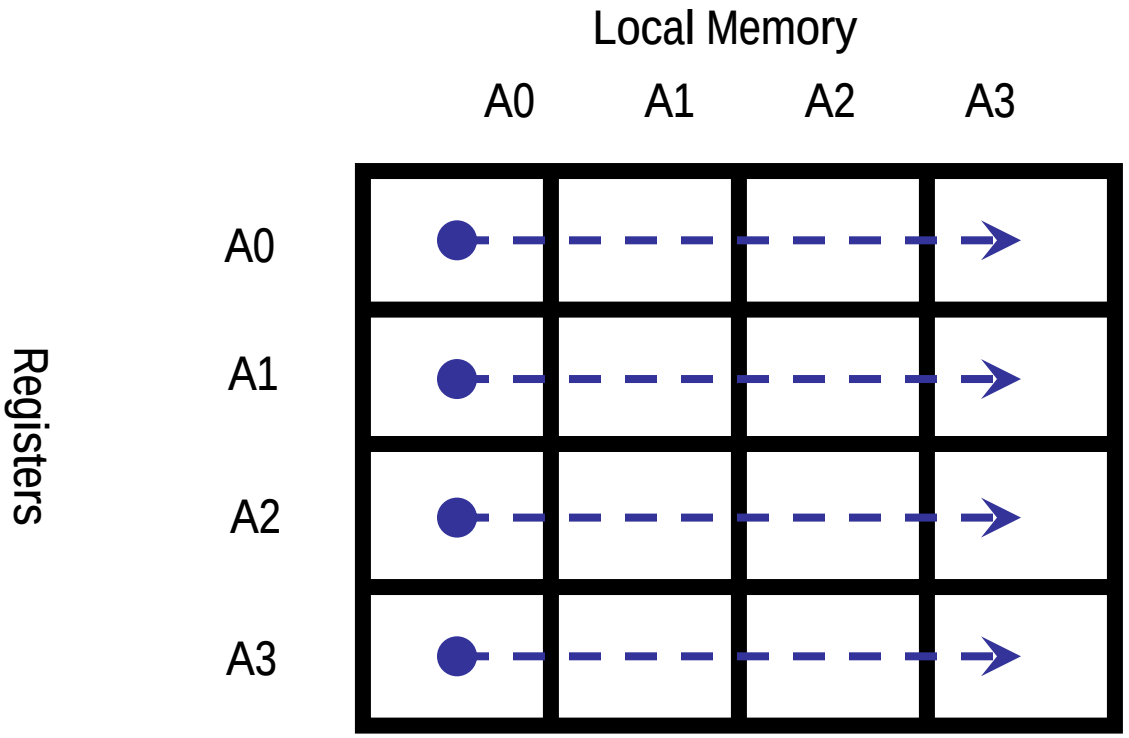


Ignore grey tiles – they're redundant

Overall Process

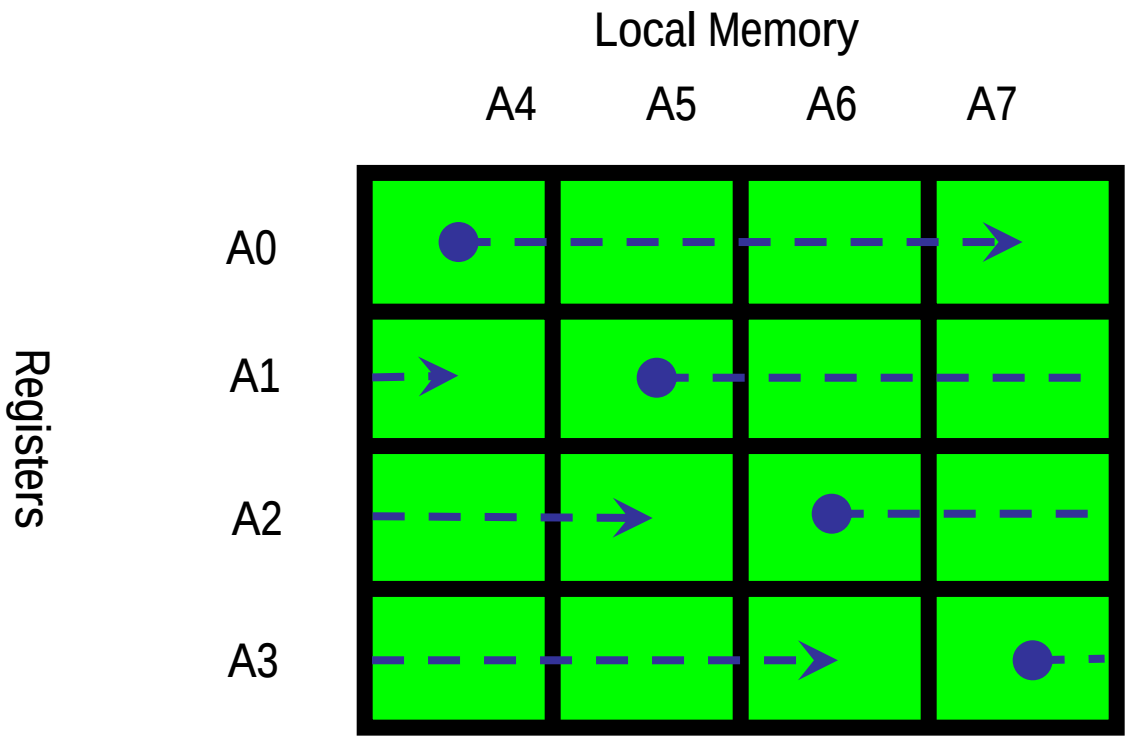
- Assemble all tiles into a work queue
- Query GPU features in order to equally distribute work queue tiles amongst its SMs
- Each tile will be handled by a warp (32 work-items) within a much larger work-group
- Work-items within a group operate in lockstep, avoiding any need for explicit synchronization – think of them as 32 lanes of SIMD.
- Launch appropriately sized work-groups on each SM (up to 256 work-items or 8 warps on G8x/G9x, and up to 320 work-items on GT2xx)

On-Diagonal Tile Calculation



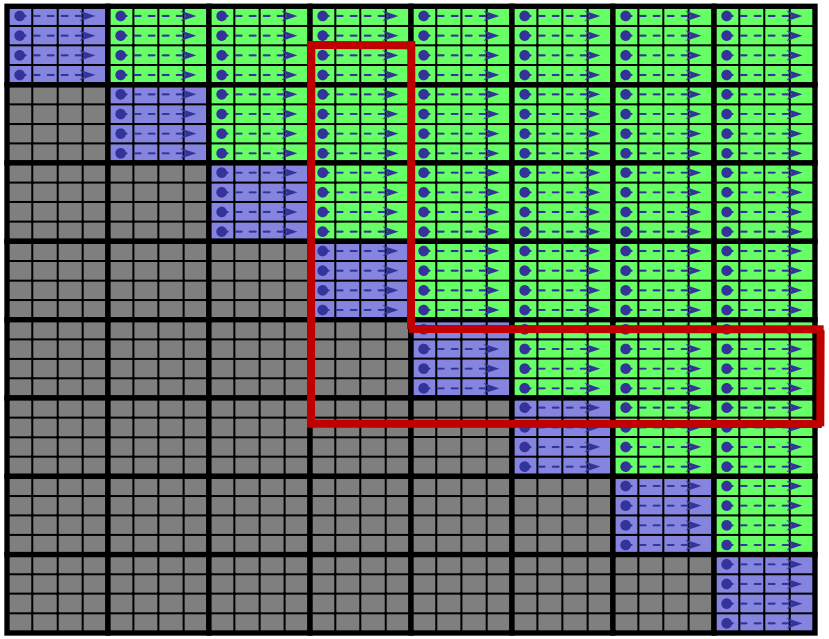
All work-items can operate on the same atom in local memory

Off-Diagonal Tile Calculation



Each work-item must operate on a different atom in Local memory

Output Management



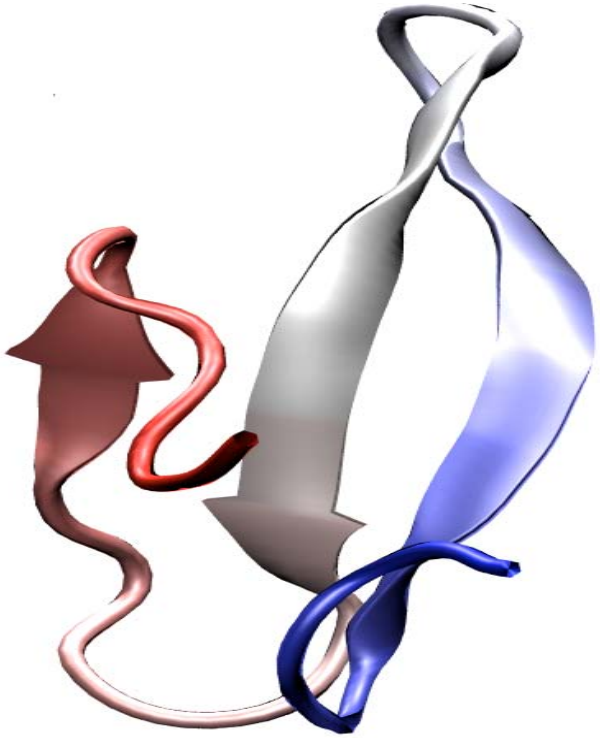
Output is complicated – possible overwrites without elaborate synchronization/scheduling

Synchronization/scheduling algorithm is tricky on small systems, but uses a small number of output buffers (1 per work group)

Solution – notice that each swath of atoms is only written 1+*N*/*p* times

So allocate (*N*/*p*)+1 output buffers and the process runs entirely asynchronously, independent of work group size and number at the expense of a short reduction at the end (there's no need to initialize them either BTW since they all get updated).

Test Case

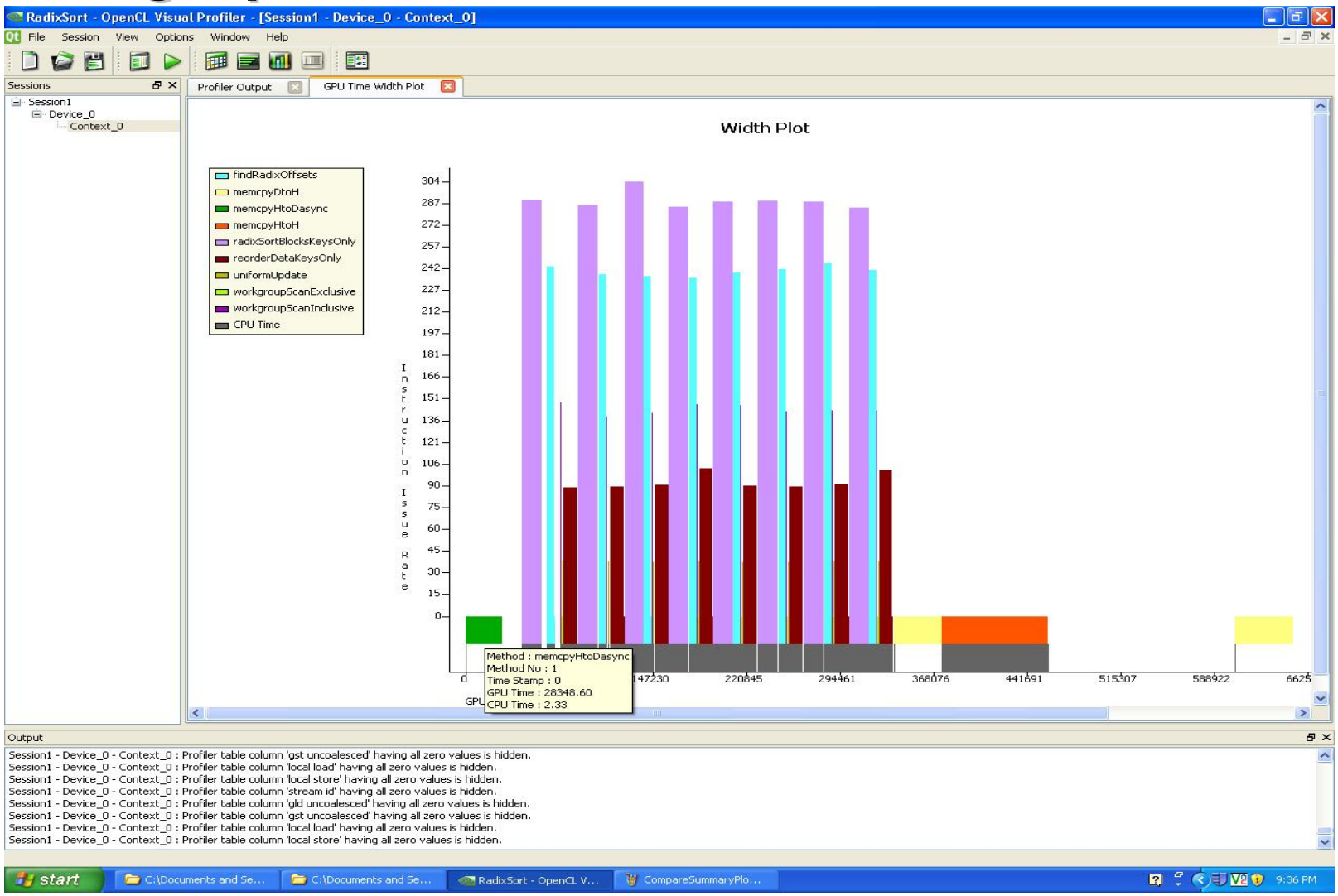


Fip35 WW Domain
544 Atoms
295,936 unique interactions

GTX280 Performance
4,864 work-items across 30 cores
218 μs per iteration
792 ns/day

Designed specifically to fold rapidly thus making it a good test case for protein folding code

Profiling OpenCL



Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

Simulating the world with Abstraction

C++ bindings for OpenCL

Motivation

"In my experience, C++ is alive and well -- thriving, even. This surprises many people. It is not uncommon for me to be asked, essentially, why somebody would choose to program in C++ instead of in a simpler language with more extensive "standard" library support, e.g., Java or C#."

Scott Meyer

- **Data abstraction**
- **Object-oriented programming**
- **Generic templates**

Goals

- Lightweight, providing access to the low-level features of the original OpenCL C API.
- Compatible with standard C++ compilers (GCC 4.x and VS 2008).
- C++ features that may be considered acceptable by all, e.g. exceptions, should not be required but may be supported by the use of polices that are not enabled by default.
- Should not require the use of the Standard Template Library.
- The bindings should be defined completely as with in a header, cl.hpp.

Something simple – hello from OpenCL C++

```
#define __CL_ENABLE_EXCEPTIONS
#define __NO_STD_VECTOR
#define __NO_STD_STRING
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif
#include <cstdio>
#include <cstdlib>
#include <iostream>

const char * helloStr = "__kernel void  hello(void) { }\n";
```

Something simple – hello from OpenCL

```
int main(void) {
    cl_int err = CL_SUCCESS;
    try {
        cl::Context context(CL_DEVICE_TYPE_GPU, 0, NULL, NULL, &err);
        cl::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
        cl::Program::Sources source(1, std::make_pair(helloStr,strlen(helloStr)));
        cl::Program program_ = cl::Program(context, source);
        program_.build(devices);
        cl::Kernel kernel(program_, "hello", &err);
        cl::CommandQueue queue(context, devices[0], 0, &err);
        cl::KernelFunctor func = kernel.bind(queue, cl::NDRange(4, 4), cl::NDRange(2, 2));
        func().wait();
    } catch (cl::Error err) {
        std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")“  << std::endl;
    }
        return EXIT_SUCCESS;
}
```

Real Time Ocean Simulation

- **FFTs are a common abstraction:**
  - Computationally expensive
  - Data parallel

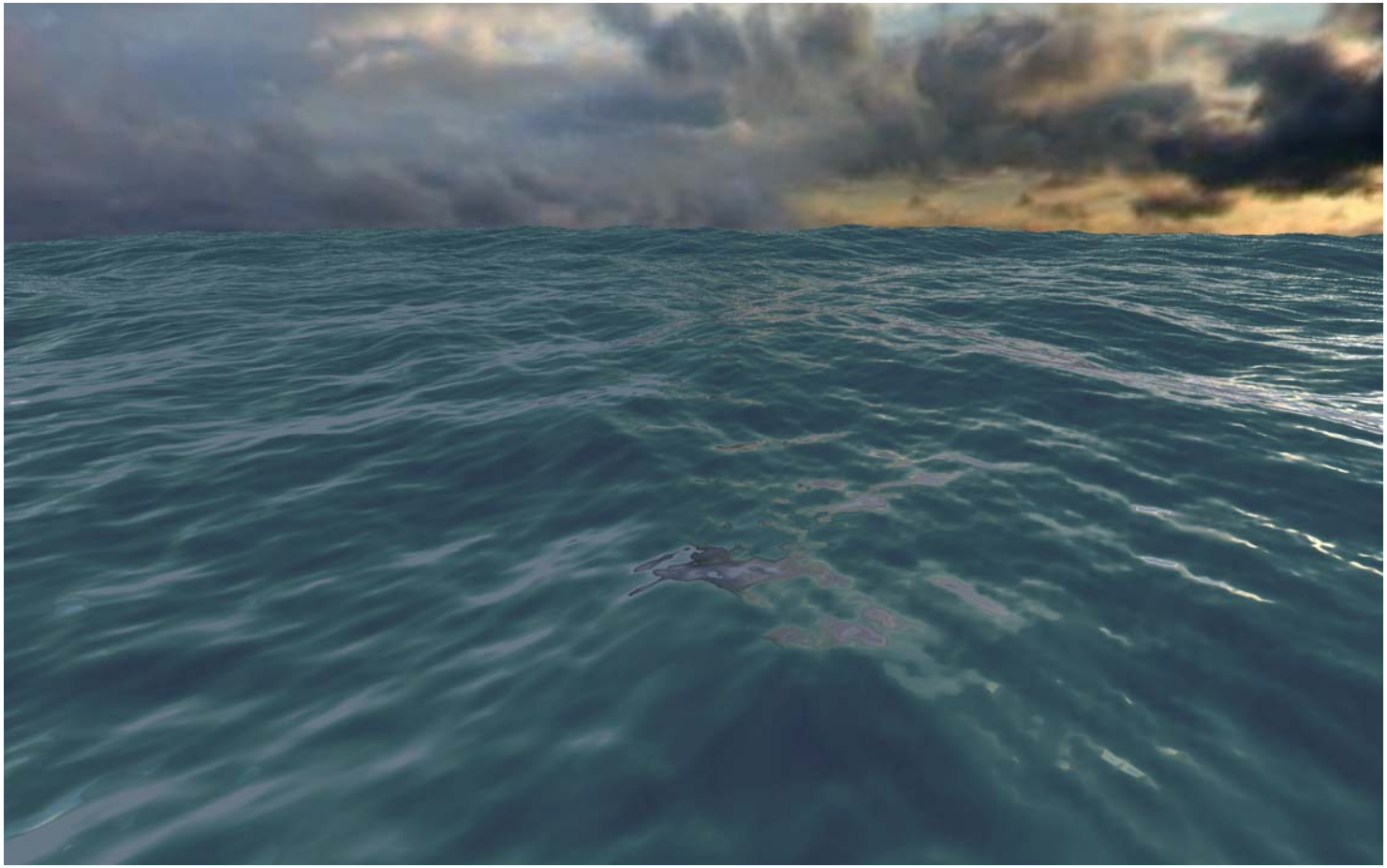
- **Today's GPUs are extremely good at executing FFTs:**
  - 1k x 1k is easy ☺
  - 2k x 2k is easy but real time rendering becomes interesting (~2Million polygons)

Real Time Ocean Simulation

- **Jerry Tenssendorf's work:**
  - Simulating Ocean Water, SIGGRAPH 1999
  - Waterworld, Titanic, and many others (2kx2k FFTs)
  - Works with sums of sinusoids but starts in Fourier domain
  - Can evaluate at any time t without having to evaluate other times
  - Use the Phillips Spectrum
    - Roughness of waves is a function of wind velocity

- **Jason L. Mitchell's work:**
  - Real-Time Synthesis and Rendering of Ocean Water, 2005
  - DX-9 Demo 256x256 FFTs, low and high frequencies separated

Real Time Ocean Simulation



Real Time Ocean Simulation

- **Starts with a platform…**

```
cl::vector<cl::Platform> platforms;
 err = cl::Platform::get(&platforms);

checkErr(err && (platforms.size() == 0 ? -1 : CL_SUCCESS), "cl::Platform::get()");

// As cl::vector (implements std::vector interface) straightforward to determine number of
// platforms, no need for additional variables as required by clGetPlatformIDs.
std::cout << "Number of platforms:\t “ << platforms.size() << std::endl;

for (cl::vector<cl::Platform>::iterator i = platforms.begin(); i != platforms.end(); ++i) {
    // pick a platform and do something
     std::cout << "  Platform Name: “ << (*i).getInfo<CL_PLATFORM_NAME>().c_str()
                << std::endl;
}
```

Real Time Ocean Simulation

- **clGetXInfo functions are provided in two flavors**
  - A static version of the form:
    - template <cl_int name> typename
    - detail::param_traits<detail::cl_device_info, name>::param_type
    - getInfo(cl_int* err = NULL) const
  - A dynamic version of the form:
    - template <typename T>
    - cl_int getInfo(cl_device_info name, T* param) const
- **Unlike the C API the C++ bindings return info values directly:**
  - No need to call info function to find memory requirements
  - Mapping from **cl_X_info** enums to C++ types, so for **cl_platform_info**:

```
F(cl_platform_info, CL_PLATFORM_PROFILE, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_VERSION, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_NAME, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_VENDOR, STRING_CLASS) \
F(cl_platform_info, CL_PLATFORM_EXTENSIONS, STRING_CLASS)
```

Real Time Ocean Simulation

- **Continue with a context of devices…**

```
cl::vector<cl::Platform>::iterator p = platforms.begin();
…
// Get all GPU devices supported by a particular platform
cl::vector<cl::Device> devices;
(*p).getDevices(CL_DEVICE_TYPE_GPU  | CL_DEVICE_TYPE_CPU, &devices);

// Create a single context for all devices
cl::Context context(devices, NULL, NULL, NULL, &err);
checkErr(err, "Conext::Context()");
// Create work-queues for CPU and GPU devices
queueCPU = cl::CommandQueue(context, devices[0], 0, &err);
checkErr(err, "CommandQueue::CommandQueue(CPU)");
queueGPU = cl::CommandQueue(context, devices[1], 0, &err);
checkErr(err, "CommandQueue::CommandQueue(GPU)");
```

Real Time Ocean Simulation

- **Load and build devices programs…**

```
std::ifstream file("ocean_kernels.cl");
checkErr(file.is_open() ? CL_SUCCESS : -1, "reading ocean_kernels.cl");
std::string prog(std::istreambuf_iterator<char>(file), (std::istreambuf_iterator<char>()));

cl::Program::Sources source(1, std::make_pair(prog.c_str(),prog.length()+1));
                                        cl::Program program(context, source);
err = program.build(devices);

if (err != CL_SUCCESS) {
   std::cout << "Info: " <<
program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices);
   checkErr(err, "Program::build()");
}
```

Real Time Ocean Simulation

- **Two (data) parallel kernels for FFT:**

```
__kernel __attribute__((reqd_work_group_size (64,1,1)))
void kfft(__global float *greal, __global float *gimag)

__kernel __attribute__((reqd_work_group_size (64,1,1)))
void  ktran(__global float *greal, __global float *gimag)
```

- **One kernel for calculating the partial differences, i.e slopes, that is used to calculate the normal's from the height map for light shading:**

```
__kernel void kPartialDiffs(__global float* h, __global float2 *slopeOut, uint width, uint height)
```

- **One (task) kernel for generating Phillips Spectrum:**

```
__kernel  void phillips(__global float2 * buffer, __global float2 *const randomNums,
                        float windSpeed, float windDir,  unsigned int height, unsigned int width)
```

Real Time Ocean Simulation

- **Build the kernels and set invariant arguments…**

```
kfftKernel = cl::Kernel(program, "kfft", &err);
checkErr(err, "Kernel::Kernel(kfft)");

ktranKernel = cl::Kernel(program, "ktran", &err);
checkErr(err, "Kernel::Kernel(ktrans)");

phillipsKernel = cl::Kernel(program, ”kphillips", &err);
checkErr(err, "Kernel::Kernel(kphillips)");

partialDiffsKernel = cl::Kernel(program, “kPartialDiffs", &err);
checkErr(err, "Kernel::Kernel(kPartialDiffs)");
```

Real Time Ocean Simulation

- **Allocate memory buffers…**

```
imag = cl::Buffer(context, CL_MEM_READ_WRITE,1024*1024*sizeof(float), 0, &err);
checkErr(err, "Buffer::Buffer(imag)");

spectrum = cl::Buffer(context, CL_MEM_READ_WRITE, 1024*1024*sizeof(cl_float2), 0,
&err);
checkErr(err, "Buffer::Buffer(spectrum)");

// Height map and partial differences (i.e. slopes) generated directly into GL for renderning
real = cl::BufferGL(context,CL_MEM_READ_WRITE, heightVBO,&err);
checkErr(err, "BufferGL::BufferGL(height)");

slopes = cl::BufferGL(context, CL_MEM_READ_WRITE, partialDiffsVBO, &err);
checkErr(err, "BufferGL::BufferGL(partialDiffs)");
```

Real Time Ocean Simulation

- **Do the work…**

```
phillipsEvent.wait(); // make sure spectrum is up-to-date, i.e. account for wind changes. in
                      // practice.  We double buffer to avoid causing delays due to
                      // continual wind changes
cl::vector<cl::Memory> v;
v.push_back(real); v.push_back(partialDifss);
err = queueGPU.enqueueAcquireGLObjects(&v);
checkErr(err, "Queue::enqueueAcquireGLObjects()");

err = kfftKernel.setArg(0, real); // other arguments are invariant, set once during setup

err = queueGPU.enqueueNDRangeKernel(kfftKernel, cl::NullRange,
                                    cl::NDRange(1024*64), cl::NDRange(64));

checkErr(err, "CommandQueue::enqueueNDRangeKernel(kfftKernel1)");

err = ktranKernel.setArg(0, real); // other arguments are invariant, set once during setup
err = queueGPU.enqueueNDRangeKernel(ktranKernel, cl::NullRange,
                                    cl::NDRange(128*129/2 * 64), cl::NDRange(64));

checkErr(err, "CommandQueue::enqueueNDRangeKernel(ktranKernel1)");
```

Real Time Ocean Simulation

- **Do the work (cond)…**

```
// note, no need to set argument as they persist from previous calls
err = queueGPU.enqueueNDRangeKernel(kfftKernel, cl::NullRange,
                                    cl::NDRange(1024*64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(kfftKernel2)");

err = queueGPU.enqueueNDRangeKernel(ktranKernel, cl::NullRange,
                                    cl::NDRange(128*129/2 * 64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(ktranKernel2)");

err  = calculateSlopeKernel.setArg(0, real); err |= partialDiffsKernel.setArg(1, slopes);
err |= calculateSlopeKernel.setArg(2, width); err |= partialDiffsKernel.setArg(3, height);
checkErr(err, "Kernel::setArg(partialDiffsKernel)");

err = queueGPU.enqueueNDRangeKernel(partialDiffsKernel, cl::NullRange,
                                    cl::NDRange(width,height), cl::NDRange(8,8));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(partialDiffsKernel)");
```

Real Time Ocean Simulation

- **Do the work (cond)…**

```
err = queueGPU.enqueueReleaseGLObjects(&v);
checkErr(err, "Queue::enqueueReleaseGLObjects(GPU)");
queueGPU.finish();
```

- **And when the wind changes…**

```
queueCPU.enqueueTask(phillipsKernel, NULL, &phillipsEvent);
```

Optional features - Exceptions

- **Not enabled by default**
- **To enable define the following before including cl.hpp:**
  - #define __CL_ENABLE_EXCEPTIONS
- **API calls that originally return an "*cl_int err*" will now throw the exception, *cl::Error*,on error:**

```
catch (cl::Error err)
{
        std::cerr << "ERROR: " << err.what()
                << "(" << err.err() << ")" << std::endl;
}
```

- **err.what() returns an error string.**
- **err.err() returns the error code.**

Optional features – No STL usage

- **Not everyone what's to use STL in their applications.**

- **OpenCL C++ bindings use std::string and std::vector.**

- **Provide alternative implementations:**
  - std::string and std::vector
- **Allow user defined versions.**

Optional features – std::vector replacement

- **Define the following before including cl.hpp:**
  - #define __NO_STD_VECTOR
- **New vector template, supporting the same interface as std::vector:**
  - template cl::vector< typename T, unsigned int N = __MAX_DEFAULT_VECTOR_SIZE>;
- **__MAX_DEFAULT_VECTOR_SIZE defaults to 10, can be overridden by user defined definition before cl.hpp, e.g.:**
  - __MAX_DEFAULT_VECTOR_SIZE 5
- **Developer can provide own version of vector by defining:**
  - #define __USE_DEV_VECTOR
  - #define VECTOR_CLASS userVector, where userVector must be a template of the form given above for cl::vector.

Optional features – std::string replacement

- **Define the following before including cl.hpp:**
  - #define __NO_STD_STRING
- **New string class, supporting the same interface a std::string:**
  - cl::string;
- **Developer can provide own version of vector by defining:**
  - #define __USE_DEV_STRING
  - #define STRING_CLASS userString, where userString must be a class supporting the same interface are std::string.

Try them out…

- **Planed for adoption as part of OpenCL 1.1**

- **Can be downloaded from Khronos OpenCL site:**
  - http://www.khronos.org/registry/cl/

- **Multiple platform and cross vendor:**
  - A little abstraction goes a long way!

Check out the Ocean waves

- **See the Ocean being simulated in real time**
  - AMD's Booth #1417

- **Down load the complete source for OpenCL Oceans at:**
  - http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

Hybrid Parallel Gas Dynamics

Michael Houston, AMD

Acknowledgements

Marcus Daniels
Los Alamos National Laboratory
T-6 Theoretical Biology and Biophysics

Paul Weber
Los Alamos National Laboratory
HPC-5 High Performance Systems Integraion

Ben Bergen
Los Alamos National Laboratory
CCS-2 Computational Physics

Special thanks:
Larry Cox
Los Alamos National Laboratory
CCS Deputy Division Leader

Sourceforge project – hypgad

http://sourceforge.net/projects/hypgad/

- **Project initiated by Los Alamos National Laboratory**
- **Grab the source and follow along:**

svn co https://hypgad.svn.sourceforge.net/svnroot/hypgad/trunk hypgad

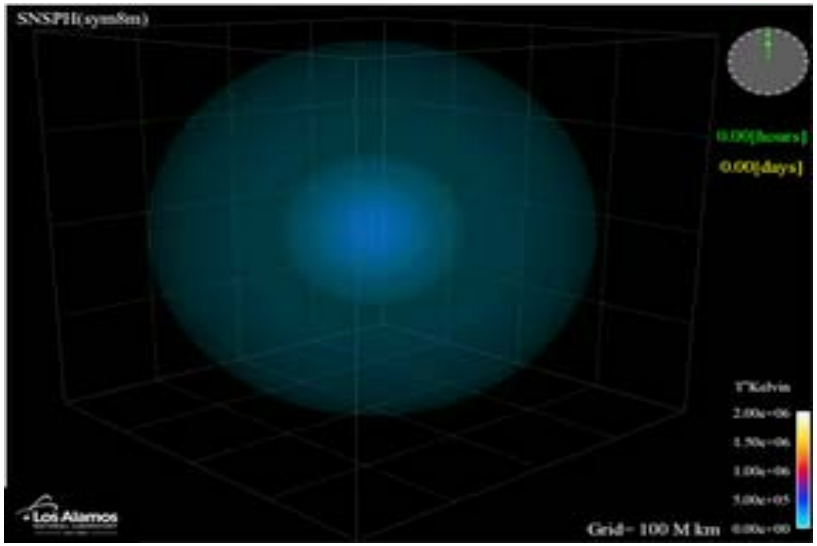
Outline

- What does this code do
- All the math
- The road to OpenCL

Supernovae Explosions

The final event in the evolution of a sufficiently massive star is a supernova

SNSPH, the code used to generate the animation shown on this slide, was developed at LANL by Chris Fryer and Mike Warren, with simulation runs performed by Gabriel Rockefeller



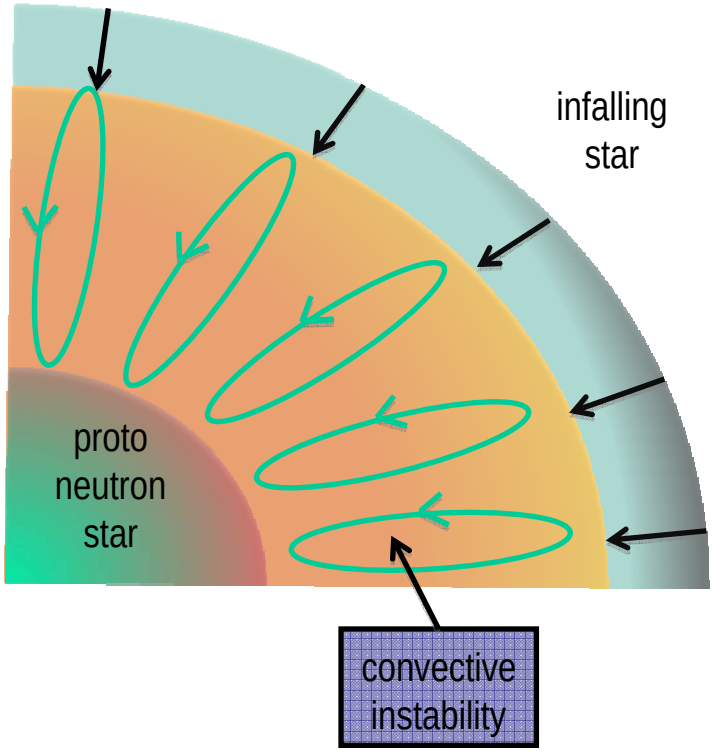
This simulation accurately captures important small-scale features of the explosion—Richtmeyer-Meshkov and Raleigh-Taylor instabilities—that can be observed at the leading edge of the shock wave.

Supernovae Explosions

The engine that drives these explosions—the reflected force of the infalling star—and the energy of the explosive shock wave that it generates depend on convective instability in a region close to the core

In the foreseeable future, we will not be able to observe or experimentally reproduce the hydrodynamic processes that lead to these phenomena

These processes can be accurately modeled by high-resolution Eulerian hydrodynamics techniques such as the MUSCL-Hancock method used in our OpenCL demo code



A Brief Trip Into The Math

Non-Linear Conservation Laws

The equation

$$q_t + f(q)_x = 0$$

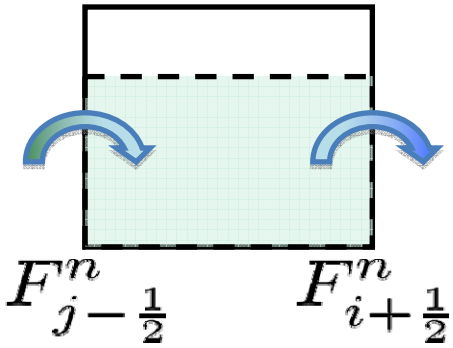
states that the change in time of a quantity *q* is only due to the flux *f*, where *f* is also a function of *q*

Numerical Method

Basic idea is simple:

Change in average amount of substance *Q* in cell *j* is determined by the flux of *Q* across the cell walls

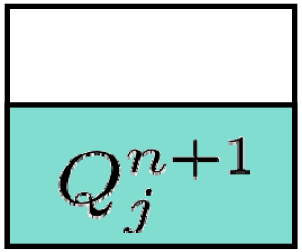
Numerical Method



Basic idea is simple:

Change in average amount of substance *Q* in cell *j* is determined by the flux of *Q* across the cell walls

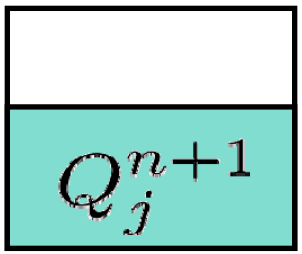
Numerical Method



Basic idea is simple:

Change in average amount of substance *Q* in cell *j* is determined by the flux of *Q* across the cell walls

Numerical Method



Numerically, this can be expressed as

$$Q_j^{n+1} = Q_j^n - \frac{\Delta t}{\Delta x}\left[F_{j+\frac{1}{2}}^n - F_{j-\frac{1}{2}}^n\right]$$

Finite Volume Method (FVM)

The flux form

$$Q_j^{n+1} = Q_j^n - \frac{\Delta t}{\Delta x}\left[F^n_{j+\frac{1}{2}} - F^n_{j-\frac{1}{2}}\right]$$

defines a generic *explicit* finite volume method, where the average flux *F* is usually a function of the cell average on either side of the face, e.g.,

$$F^n_{i-\frac{1}{2}} = \mathcal{F}(Q^n_{j-1}, Q^n_i)$$

Finite Volume Method (FVM)

A first attempt at defining *F* might be a simple arithmetic average

$$F^n_{i-\frac{1}{2}} = \frac{1}{2}\left[f(Q^n_{j-1}) + f(Q^n_j)\right]$$

This approach leads to the update rule

$$Q^{n+1}_j = Q^n_j - \frac{\Delta t}{2\Delta x}\left[f(Q^n_{j+1}) - f(Q^n_{j-1})\right]$$

Unfortunately, this rule is numerically UNSTABLE!

Lax-Friedrichs Method

A better approach is to *also* take an arithmetic average of the cell averages *Q*

$$Q_j^{n+1} = \frac{1}{2}\left(Q_{j-1}^n + Q_{j+1}^n\right) - \frac{\Delta t}{2\Delta x}\left[f(Q_{j+1}^n) - f(Q_{j-1}^n)\right]$$

This can be expressed in flux form as

$$F_{j-\frac{1}{2}}^n = \frac{1}{2}\left[f(Q_{i-1}^n) + f(Q_j^n)\right] - \frac{\Delta x}{2\Delta t}\left(Q_j^n - Q_{j-1}^n\right)$$

The Lax-Friedrichs method is stable provided that

$$\frac{\Delta t}{\Delta x}\max_p \left|\lambda^p\right| < 1$$

The Road To OpenCL

Quick explanation of source setup

hypgad/

src/

boundary/
data/
display/
grid/
io/
mp/
ocl/
pipeline/
simulation/
solve/ **this is where most of the CL code is located**
utils

See:
hypgad/src/solve/* for most of the “action” for OpenCL
hypgad/src/mp/* for the MPI code

Starting from the C++ code – the algorithm steps

slopesX()
reconstructionPredictorsX()
predictorsX(dt)
reconstructionX()
riemannFluxesX(dt)
averagesX(dt)
updateBoundary()
primitives()

slopesY()
reconstructionPredictorsY()
predictorsY(dt)
reconstructionY()
riemannFluxesY(dt)
averagesY(dt)
updateBoundary()
primitives()

Starting from the C++ code – creating the kernels

- **Find all of the parallel loops**
- **Make dimensionality of loops obvious**
  - Convert into 2D loops for 2D grid
  - This will become the NDRange for the kernel launch
- **Factor loop bodies into macros**
  - Code sharing via the macros across X/Y sweeps and offsets
  - Similar patterns in macros, so optimizations may be reused more easily
- **Create CL kernels**
  - Kernels for each part of the algorithm on previous page
  - Body of kernels use the corresponding macro
  - This makes it easier to switch between CL code and C code to verify and check along the way

Starting from the C++ code – setup OpenCL

- **Put all of the setup into the instantiation of the solver**
- **Get Platform information**
- **Get Device information**
- **Create context with platform and requested devices**
- **Create all the buffers**
- **Build program**
  - Load .cl file and pass string to clBuildProgram
- **Create Command Queue on device**
- **Get handles to individual kernels**

Starting from the C++ code – running

- **All of the "magic" is in the advance() method**
  - Call each step of the algorithm
    - Adjust kernel arguments as needed
      - Many arguments do not vary and are only setup once
        - Theoretically code may be specialized by the compiler at runtime
    - Enqueue kernel to device
  - Read/write buffers
    - getMax()
  - Exchange with neighbors (MPI)

Optimizations

- **Fuse kernels**
  - Many of the kernels use the same inputs and outputs
  - Running independently forces extra load/store of data

  - Advantages of fusing kernels
    - Increase arithmetic intensity by fusing kernels together

  - Disadvantages of fusing kernels
    - Increased register pressure, more complex code

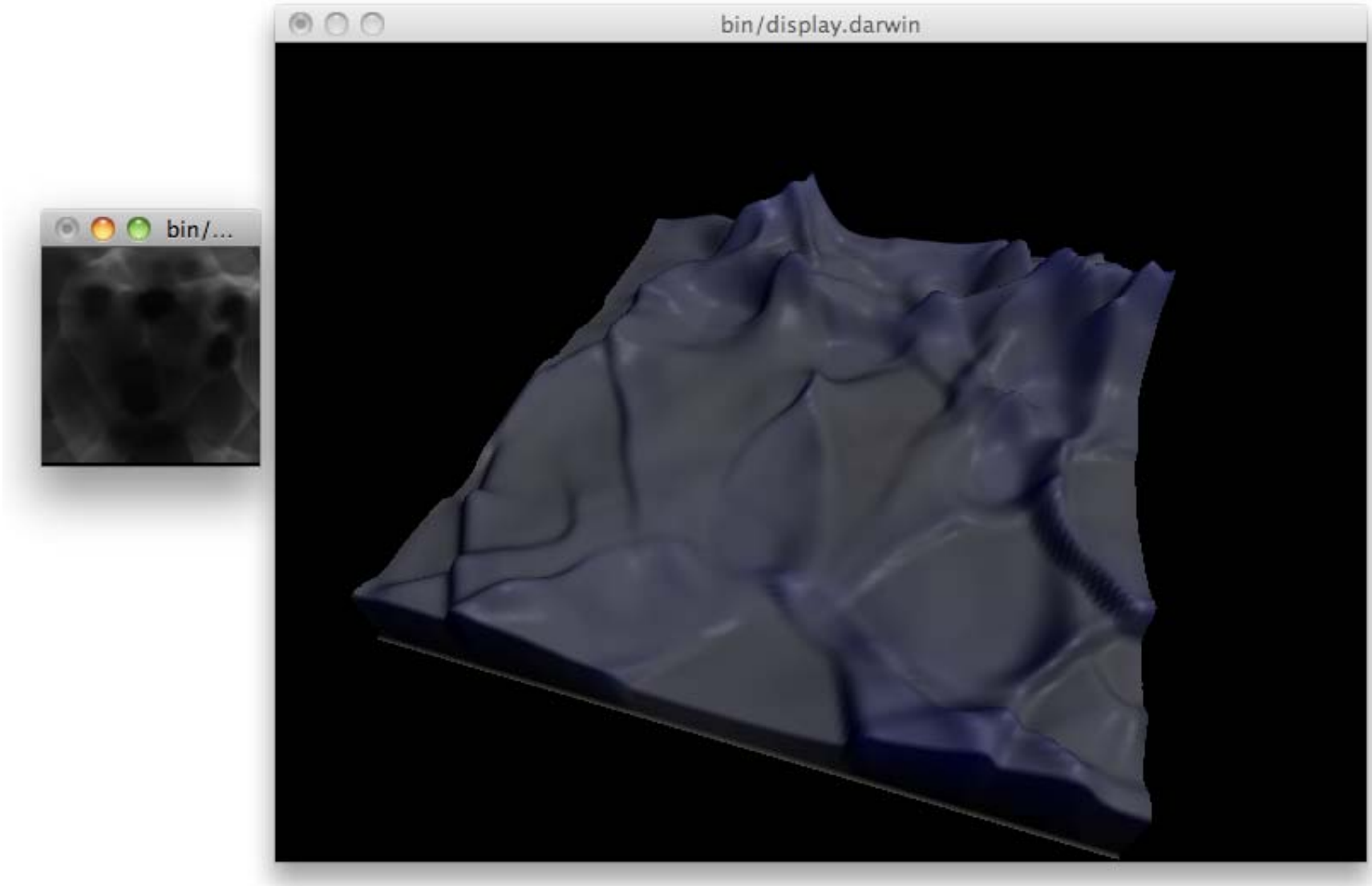
  - Current code base fuses the “easy” kernels that don’t require heavy code modification

Current state of code – 09/30/2009

- **Little tuning so far**
  - Main goal was to get it up and running on “everything”
  - No vector utilization, yet
  - Experimental use of __local memory
  - __constant not useful thus far because of limited size

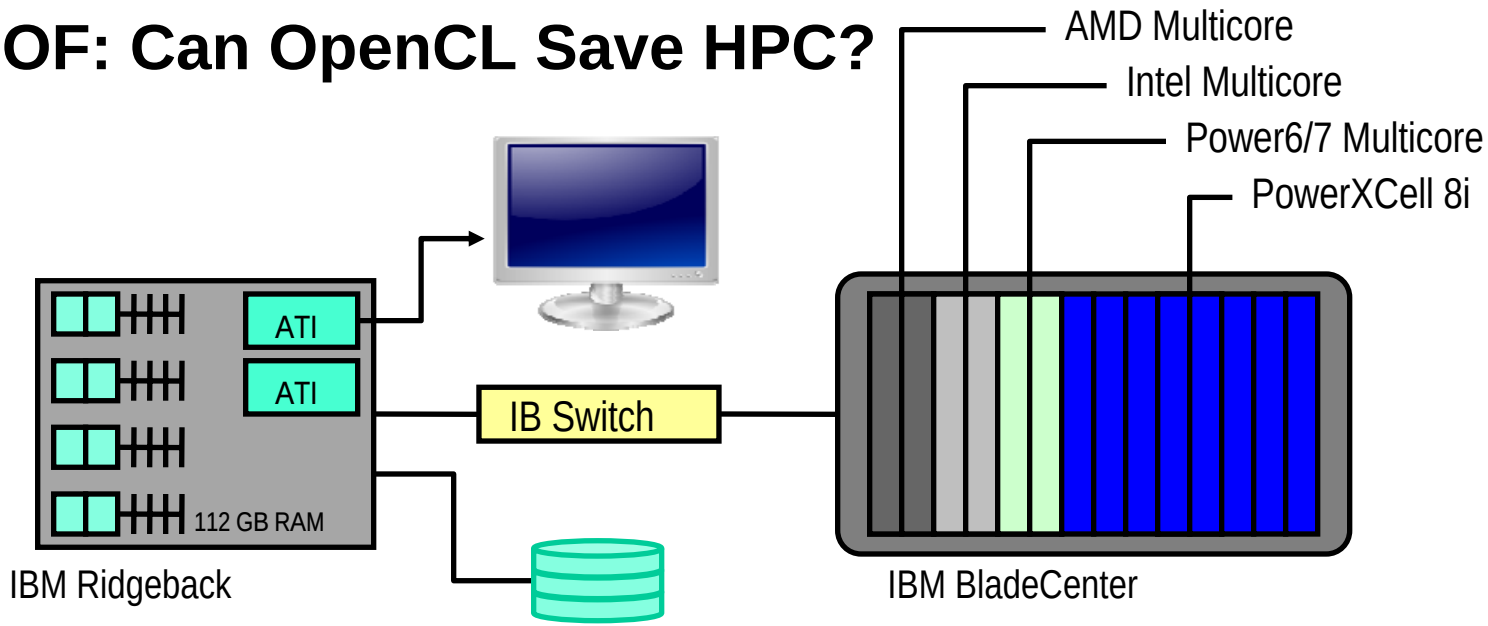
- **Up and running on multiple vendor’s implementations**
  - You will see it running in several places on the show flow
  - Multi-vendor demonstration where the code is running distributed via MPI across nodes with different OpenCL hardware

- **Hopefully some tuning for the final presentation and demo**


bin/...
bin/display.darwin

SC09 OpenCL Real-time Interactive Demo

- **Distributed-memory parallel using multiple context OpenMPI**
  - mpirun –np 1 display 1900 1200 : -np 8 cell_sim : -np 2 amd_sim
- **Data and task-parallel using OpenCL**
- **SC09 BOF: Can OpenCL Save HPC?**



Slide 128

Agenda

- **Heterogeneous computing and the origins of OpenCL**
- **Understanding OpenCL: fundamental models**
- **A simple example, vector addition**
- **OpenCL in Action (Case Studies)**
  - Basic OpenCL: N-body program
  - C++ and OpenCL: Ocean dynamics simulation
  - SuperComputing OpenCL: the demo from LANL
  - OpenCL and the CPU: video processing.

OpenCL*: it's not just a GPGPU Language

- **OpenCL defines a platform API to coordinate heterogeneous parallel computations**
  - Literature rich with parallel coordination languages/API
  - OpenCL unique in its ability to coordinate CPUs, GPUs, etc

- **Key coordination concepts**
  - Each device has its own asynchronous workqueue
  - Synchronize between OpenCL computations w/event handles from different (or same) devices
  - Enables algorithms and systems that use all available computational resources
  - Enqueue "native functions" for integration with C/C++ code

OpenCL's Two Styles of Data-Parallelism

- **Explicit SIMD data parallelism:**
  - The kernel defines one stream of instructions
  - Parallelism from using wide vector types
  - Size vector types to match native HW width
  - Combine with task parallelism to exploit multiple cores.

- **Implicit SIMD data parallelism (i.e. shader-style):**
  - Write the kernel as a "scalar program"
  - Use vector data types sized naturally to the algorithm
  - Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware.

Both approaches are viable CPU options

Data-Parallelism: options on IA processors

- **Explicit SIMD data parallelism**
  - Programmer chooses vector data type (width)
  - Compiler hints using attributes
    - vec_type_hint(type*n)*
- **Implicit SIMD Data parallel**
  - Map onto CPUs, GPUs, Larrabee, …
    - SSE/AVX/LRBni: 4/8/16 workitems in parallel
- **Hybrid use of the two methods**
    - AVX: can run two 4-wide workitems in parallel
    - LRBni: can run four 4-wide workitems in parallel

Explicit SIMD data parallelism

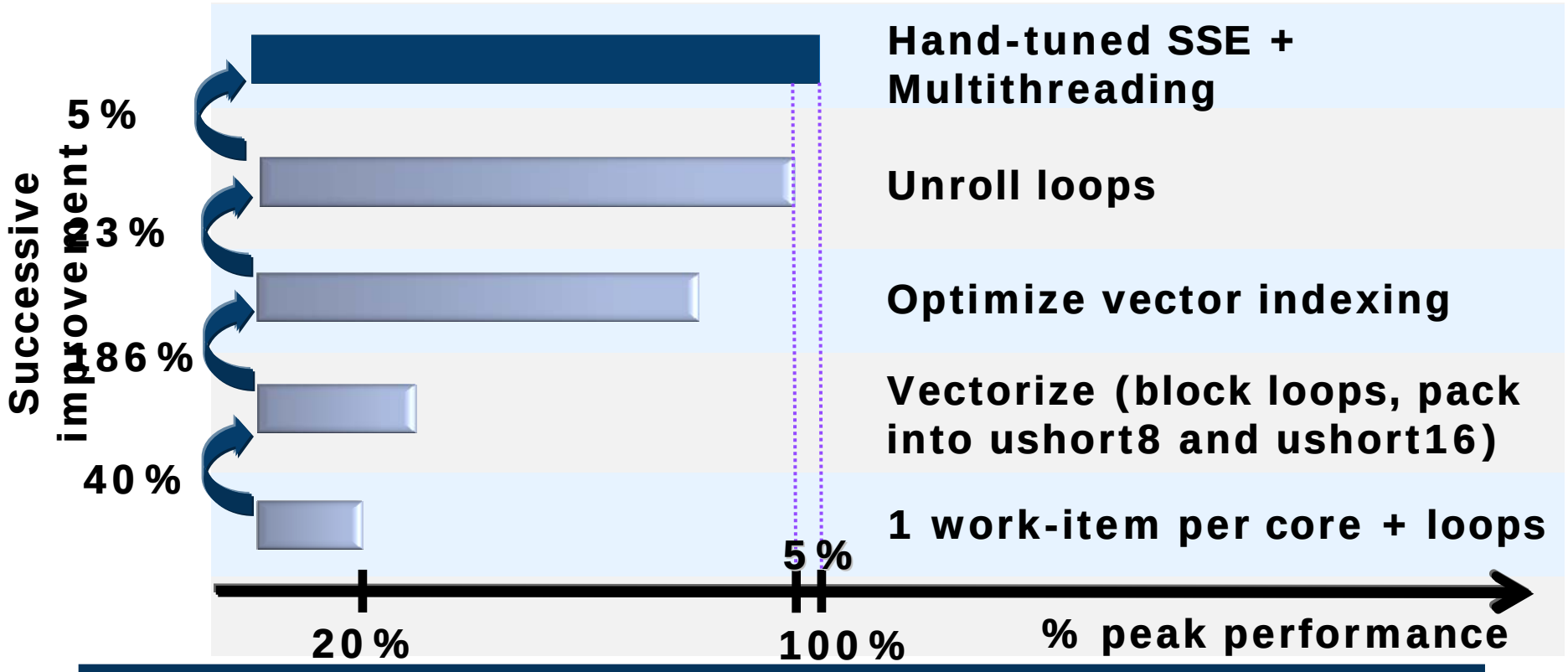
- **OpenCL as a portable interface to vector instruction sets.**
  - Block loops and pack data into vector types (float4, ushort16, etc).
  - Replace scalar ops in loops with blocked loops and vector ops.
  - Unroll loops, optimize indexing to match machine vector width

```
float a[N], b[N], c[N];
for (i=0; i<N; i++)
    c[i] = a[i]*b[i];

<<< the above becomes >>>>

float4 a[N/4], b[N/4], c[N/4];
for (i=0; i<N/4; i++)
     c[i] = a[i]*b[i];
```

Explicit SIMD data parallelism means you tune your code to the vector width and other properties of the compute device

Explicit SIMD data parallelism: Case Study

- Video contrast/color optimization kernel on a dual core CPU.



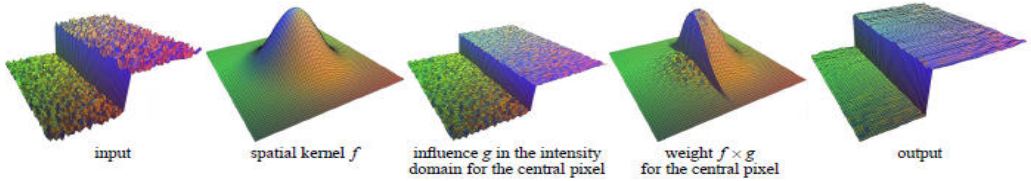
Good news: OpenCL code 95% of hand-tuned SSE/MT perf.
Bad news: New platform, redo all those optimizations.

3 Ghz dual core CPU
pre-release version of OpenCL
Source: Intel Corp.

* Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

Towards “Portable” Performance

- **The following C code is an example of a Bilateral 1D filter:**
- **Reminder: Bilateral filter is an edge preserving image processing algorithm.**
- **See more information here: http://scien.stanford.edu/class/psych221/projects/06/imagescaling/bilati.html**



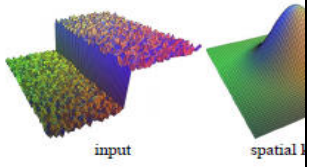

```
void P4_Bilateral9 (int start, int end, float v)
{
   int i, j, k;
   float w[4], a[4], p[4];
   float inv_of_2v = -0.5 / v;
   for (i = start; i < end; i++) {
      float wt[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
      for (k = 0; k < 4; k++)
         a[k] = image[i][k];
      for (j = 1; j <= 4; j++) {
         for (k = 0; k < 4; k++)
            p[k] = image[i - j*SIZE][k] - image[i][k];
         for (k = 0; k < 4; k++)
            w[k] = exp (p[k] * p[k] * inv_of_2v);
         for (k = 0; k < 4; k++) {
            wt[k] += w[k];
            a[k] += w[k] * image[i - j*SIZE][k];
         }
      }
      for (j = 1; j <= 4; j++) {
         for (k = 0; k < 4; k++)
            p[k] = image[i + j*SIZE][k] - image[i][k];
         for (k = 0; k < 4; k++;
            w[k] = exp (p[k] * p[k] * inv_of_2v);
         for (k = 0; k < 4; k++) {
            wt[k] += w[k];
            a[k] += w[k] * image[i + j*SIZE][k];
         }
      }
      for (k = 0; k < 4; k++) {
         image2[i][k] = a[k] / wt[k];
      }
   }
}
```

Source: Intel Corp.

Towards “Portable” Performance

- The following C code is an example of a Bilateral 1D
- Reminder: B preserving i
- See more in http://scien. projects/06/i

```
void P4_Bilateral9 (int start, int end, float v)
{
   <<< Declarations >>>
   for (i = start; i < end; i++) {
      for (j = 1; j <= 4; j++) {
      <<< a series of short loops >>>>
       }
      for (j = 1; j <= 4; j++) {
       <<< a 2nd series of short loops >>>
      }
 }
}
```



Source: Intel Corp.

“Implicit SIMD” data parallel code

- **“outer” loop replaced by work-items running over an NDRange index set.**
- **NDRange 4*image size … since each workitem does a color for each pixel.**
- **Leave it to the compiler to map work-items onto lanes of the vector units …**

```
__kernel void P4_Bilateral9 (__global float* inImage, __global float* outImage, float v)
{
    const size_t myID     = get_global_id(0);
    const float inv_of_2v = -0.5f / v;
    const size_t myRow    = myID / IMAGE_WIDTH;
         size_t maxDistance = min(DISTANCE, myRow);
          maxDistance = min(maxDistance, IMAGE_HEIGHT - myRow);
    float currentPixel, neighborPixel, newPixel;
    float diff;
    float accumulatedWeights, currentWeights;
    newPixel = currentPixel = inImage[myID];
    accumulatedWeights = 1.0f;
    for (size_t dist = 1; dist <= maxDistance; ++dist)
    {
          neighborPixel         = inImage[myID + dist*IMAGE_WIDTH];
          diff                  = neighborPixel - currentPixel;
          currentWeights        = exp(diff * diff * inv_of_2v);
          accumulatedWeights += currentWeights;
          newPixel              += neighborPixel * currentWeights;
          neighborPixel           = inImage[myID - dist*IMAGE_WIDTH];
          diff                    = neighborPixel - currentPixel;
          currentWeights          = exp(diff * diff * inv_of_2v);
          accumulatedWeights += currentWeights;
          newPixel              + = neighborPixel * currentWeights;
    }
    outImage[myID] = newPixel / accumulatedWeights;
}
```

“Implicit SIMD” data parallel code

- “o
by
ov
ind

- ND
…
wo
for

- Le
co
ite
the

```
__kernel void p4_bilateral9(__global float* inImage,
                            __global float* outImage, float v)
{
    const size_t myID    = get_global_id(0);
    <<< declarations >>>
    for (size_t dist = 1; dist <= maxDistance; ++dist){
       neighborPixel    = inImage[myID +
                                  dist*IMAGE_WIDTH];
       diff             = neighborPixel - currentPixel;
       currentWeights   = exp(diff * diff * inv_of_2v);
      << plus others to compute pixels, weights, etc >>
       accumulatedWeights += currentWeights;
    }
    outImage[myID] = newPixel / accumulatedWeights;
}
```

Portable Performance in OpenCL

- **Implicit SIMD code … where the framework maps work-items onto the "lanes of the vector unit" … creates the opportunity for portable code that performs well on full range of OpenCL compute devices.**

- **Requires mature OpenCL technology that "knows" how to do this:**
  - … But it is important to note …. we know this approach works since its based on the way shader compilers work today.

Conclusion

- **OpenCL defines a platform-API/framework for heterogeneous computing … not just GPGPU or CPU-offload programming.**

- **OpenCL has the potential to deliver portably performant code; but only if its used correctly:**
  - Implicit SIMD data parallel code has the best chance of mapping onto a diverse range of hardware … once OpenCL implementation quality catches up with mature shader languages.

- **The future is clear:**
  - Braided parallelism mixing task parallel and data parallel code in a single program … balancing the load among ALL OF the platform's available resources.