OpenCL

What is OpenCL?

- Cross-platform parallel computing API and C-like language for heterogeneous computing devices
- Code is portable across various target devices:
  - Correctness is guaranteed
  - Performance of a given kernel is not guaranteed across differing target devices
- OpenCL implementations already exist for AMD, ATI, and NVIDIA GPUs, x86 CPUs
- In principle, OpenCL could also target DSPs, Cell, and perhaps also FPGAs

More on Multi-Platform Targeting

- **Targets a broader range of CPU-like and GPU-like devices than CUDA**
  - Targets devices produced by multiple vendors
  - Many features of OpenCL are optional and may not be supported on all devices
- **OpenCL codes must be prepared to deal with much greater hardware diversity**
- **A single OpenCL kernel will likely not achieve peak performance on all device types**

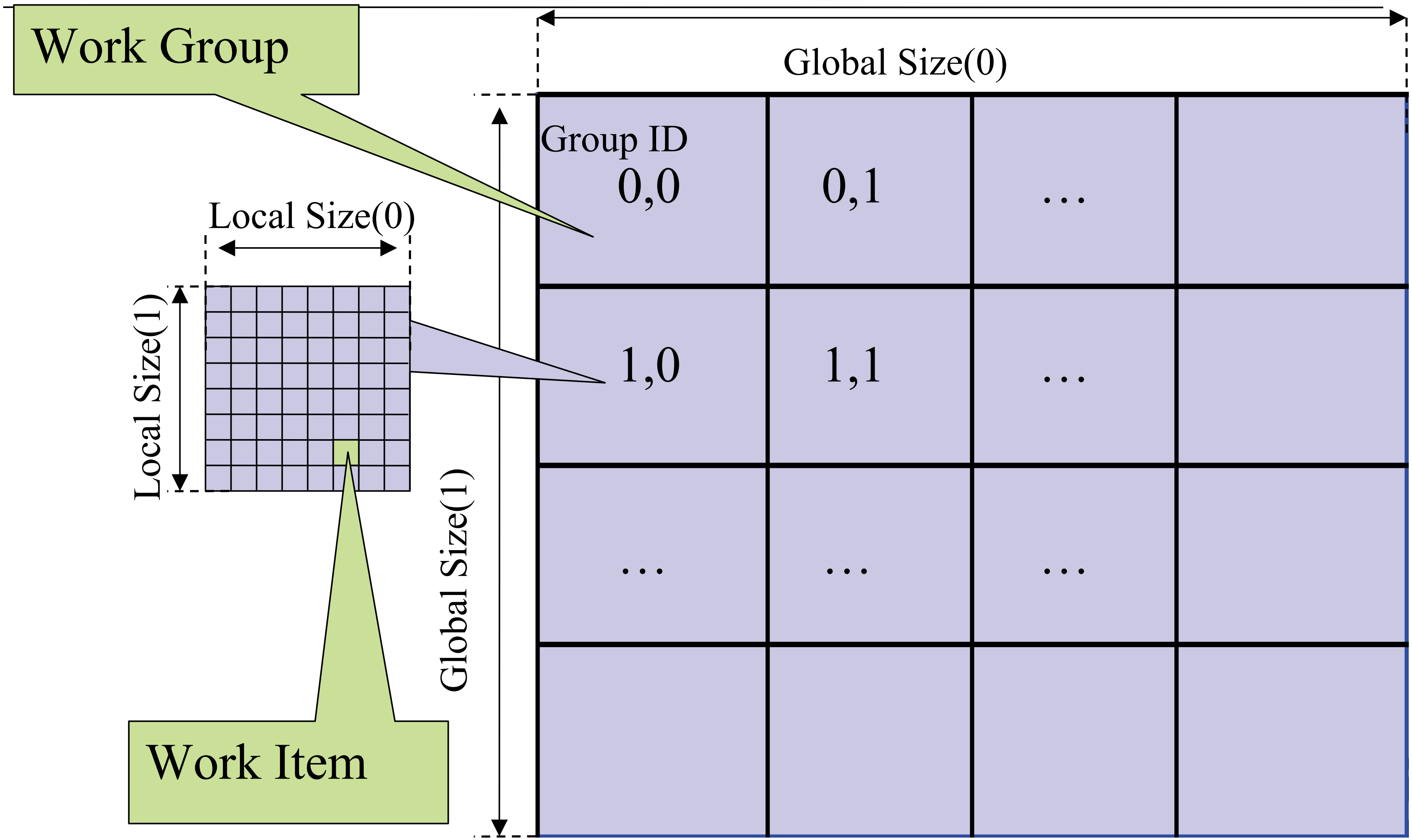
OpenCL Data Parallel Model Summary

- Parallel work is submitted to devices by launching **kernels**
- Kernels run over global dimension index ranges (**NDRange**), broken up into "**work groups**", and "**work items**"
- Work items executing within the same work group can synchronize with each other using barriers or memory fences
- Work items in different work groups can only sync with each other by launching a new kernel

Mapping Data Parallelism Models: OpenCL to CUDA

OpenCL Parallelism Concept	CUDA Equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

OpenCL NDRange Configuration



Mapping OpenCL indices to CUDA

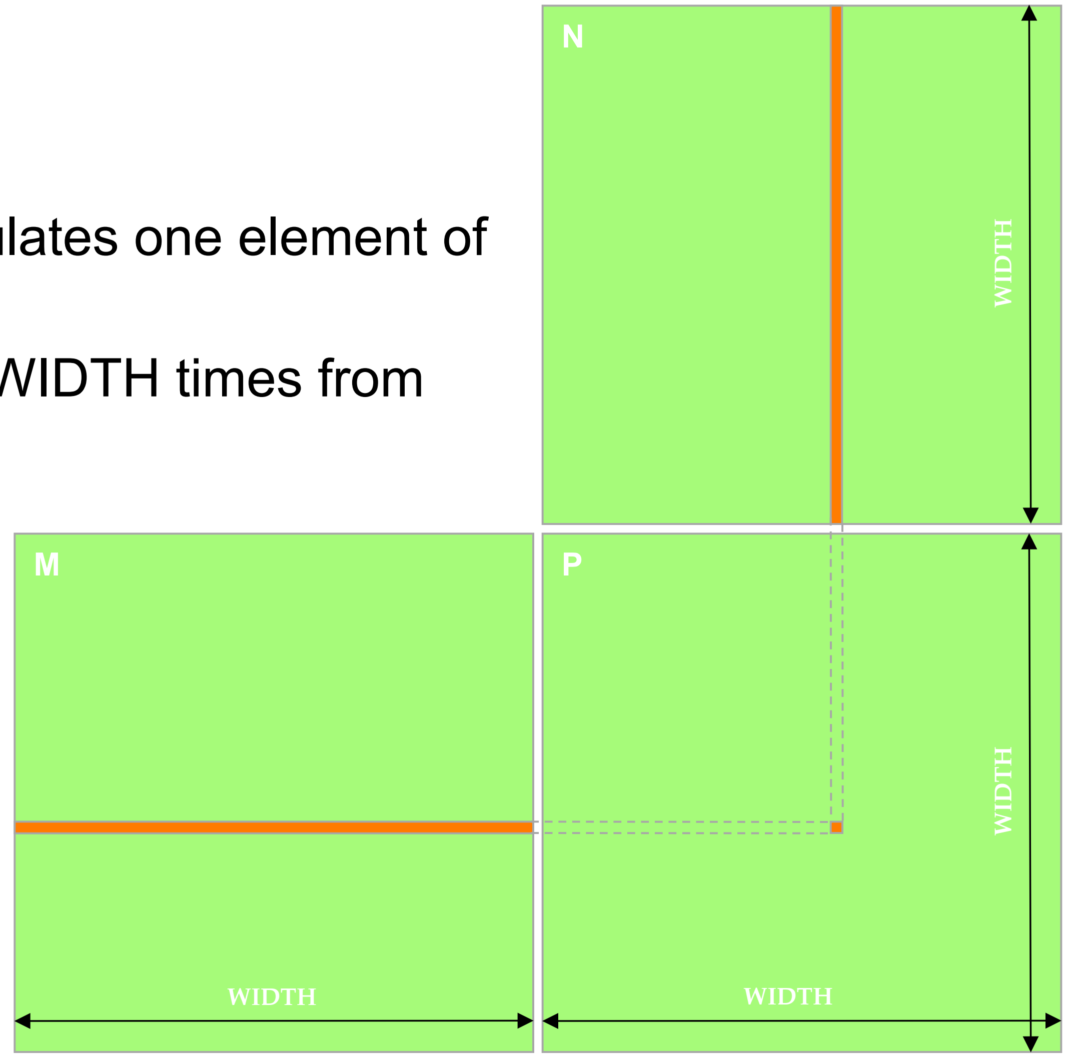
OpenCL API call	Explanation	CUDA equivalent
get_global_id(0);	Global index of the work item in the *x*-dimension	blockIdx.x×blockDim.x+threadIdx.x
get_local_id(0)	Local index of the work item within the work group in the *x*-dimension	threadIdx.x
get_global_size(0);	Size of NDRange in the *x*-dimension	gridDim.x ×blockDim.x
get_local_size(0);	Size of each work group in the *x*-dimension	blockDim.x

A Simple Example Matrix Multiplication

- **A simple matrix multiplication example that illustrates the basic features of memory and thread management in OpenCL programs**
  - Private register usage
  - Work item ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

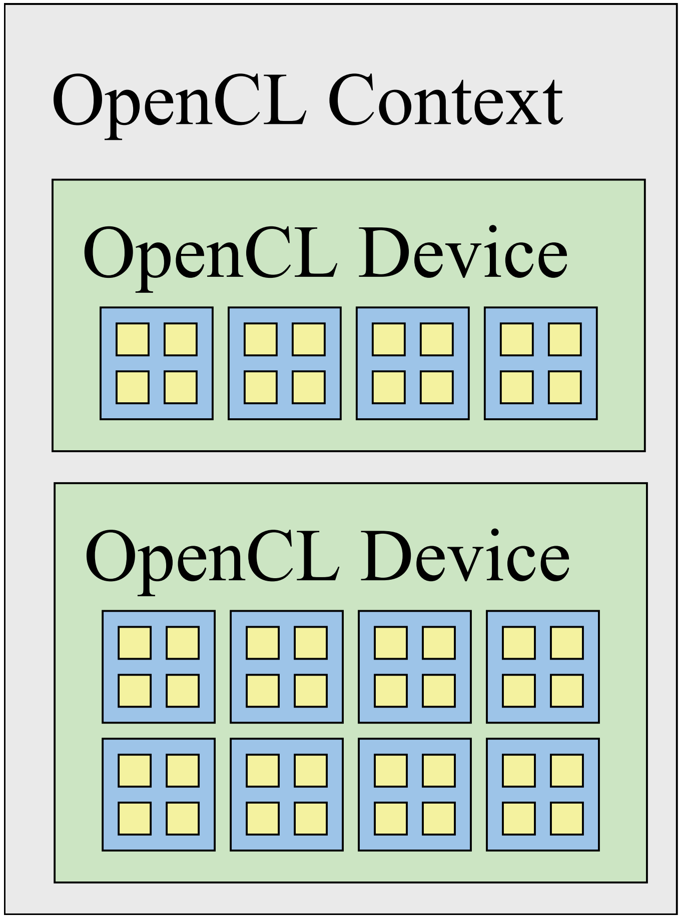
Square Matrix-Matrix Multiplication

- **P = M * N of size WIDTH x WIDTH**
  - Each work item calculates one element of P
  - M and N are loaded WIDTH times from global memory

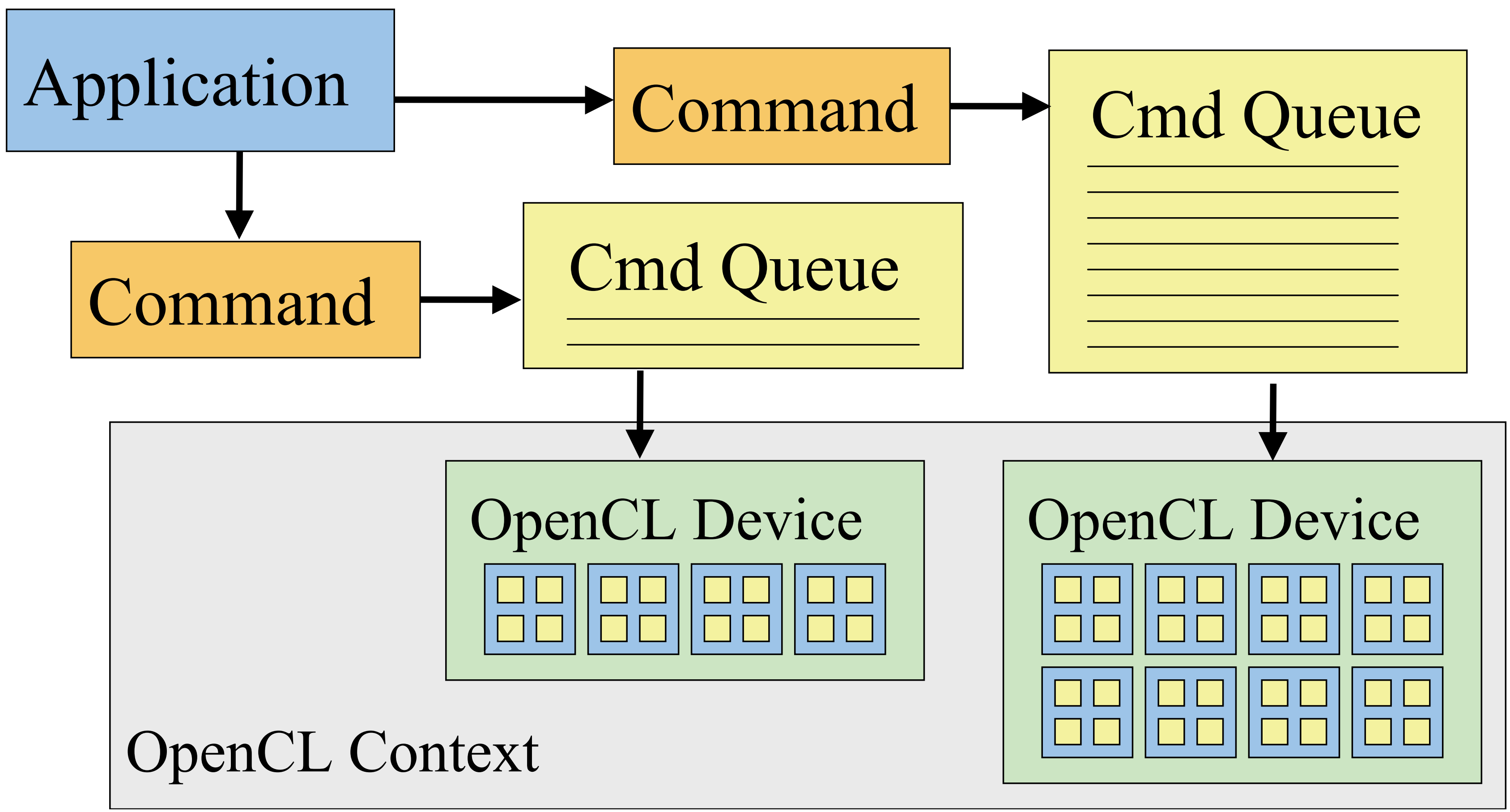


OpenCL Context

- **Contains one or more devices**
- **OpenCL memory objects associated with a context, not a specific device**
- **clCreateBuffer() is the main data object allocation function**
  - error if an allocation is too large for any device in the context
- **Each device needs its own work / command queue(s)**
- **Memory transfers are associated with a command queue (thus a specific device)**



OpenCL Device Command Execution



OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;
// create context including all available OpenCL devices
cl_context clctx = clCreateContextFromType(
                      0, CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);
size_t parmsz;
// query number of devices in context
clerr = clGetContextInfo(
                      clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);
// now that size is known, allocate list for device info
cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
// query device info
clerr = clGetContextInfo(
               clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);
// create command queue for first OpenCL device
cl_command_queue clcmdq = clCreateCommandQueue(
       clctx, cldevs[0], 0, &clerr);
```

Data Allocation

- **clCreateBuffer();**
  - Requires five parameters
    - OpenCL context
    - Allocation and usage flags
    - Size in bytes
    - Host memory pointer
    - Returned error code

Host-to-Device Data Transfer

- **clEnqueueWriteBuffer();**
  - memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size in bytes of written data
    - Host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

Device-to-Host Data Transfer

- **clEnqueueReadBuffer();**
  - memory data transfer to host
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size in bytes of written data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

OpenCL Memory Systems

- __global – large, long latency
- __private – on-chip device registers
- __local – memory accessible from multiple PEs or work items
  - May be SRAM or DRAM, must query…
- __constant – read-only constant cache
- Programmer manages device memory explicitly

OpenCL Memory Types	CUDA Equivalent
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	Local memory

Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   cl_mem Md, Nd, Pd;
   Md=clCreateBuffer(clctxt, CL_MEM_READ_WRITE,
                            mem_size_M, NULL, NULL);
   Nd=clCreateBuffer(clctxt, CL_MEM_READ_WRITE,
                            mem_size_N, NULL, &ciErrNum);

   clEnqueueWriteBuffer(clcmdque, Md, CL_FALSE, 0, mem_size_M,
                                   (const void * )M, 0, 0, NULL);
   clEnqueueWriteBuffer(clcmdque, Nd, CL_FALSE, 0, mem_size_N,
                                   (const void *)N, 0, 0, NULL);

   Pd=clCreateBuffer(clctxt, CL_MEM_READ_WRITE, mem_size_P,
                            NULL, NULL);
```

Output Matrix Data Transfer (Host-side Code)

2. // Kernel invocation code – to be shown later

3. // Read P from the device

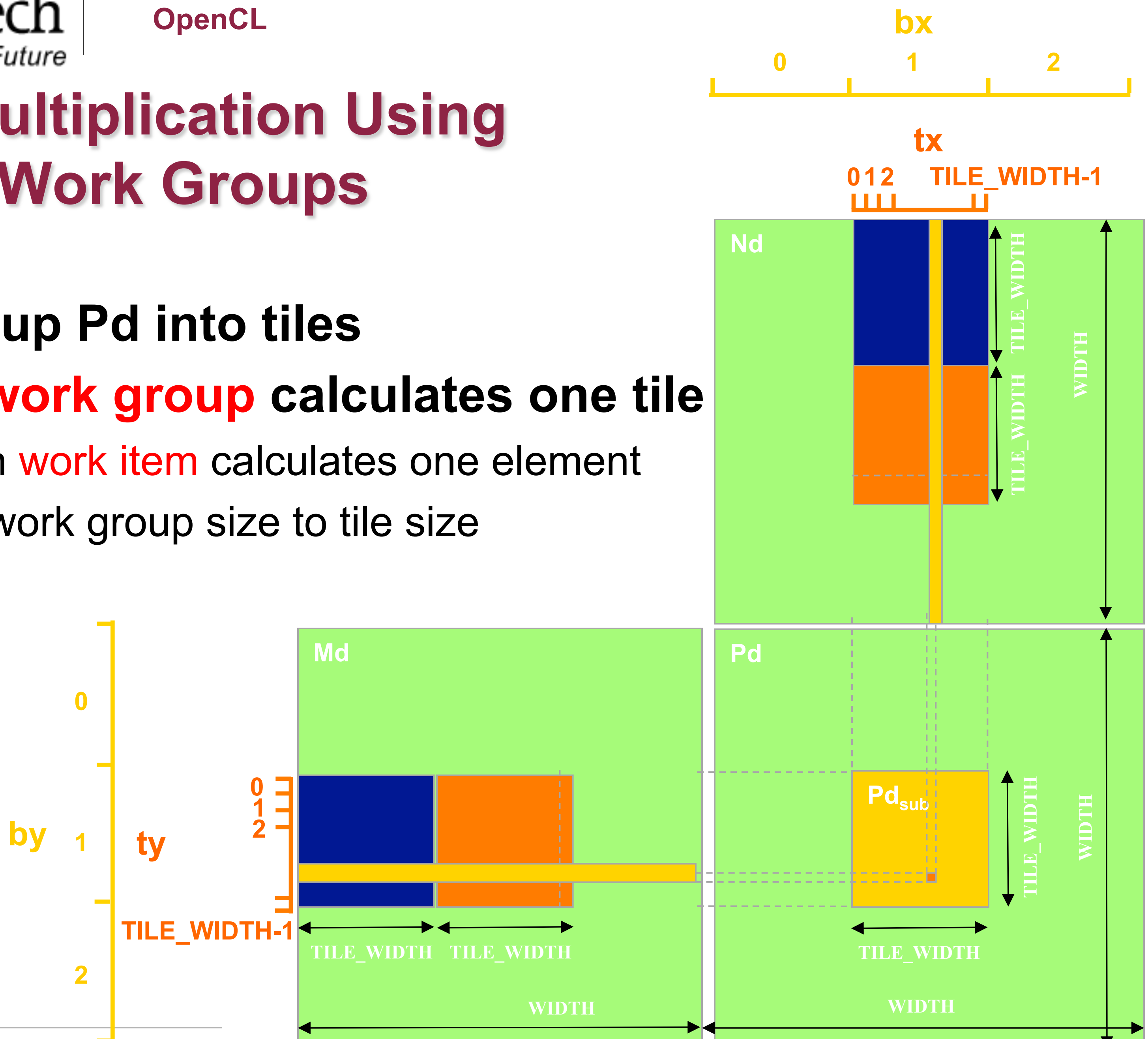
```
clEnqueueReadBuffer(clcmdque, Pd, CL_FALSE,
    0, mem_size_P,(void*)P), 0, 0, &ReadDone);
```

// Free device matrices

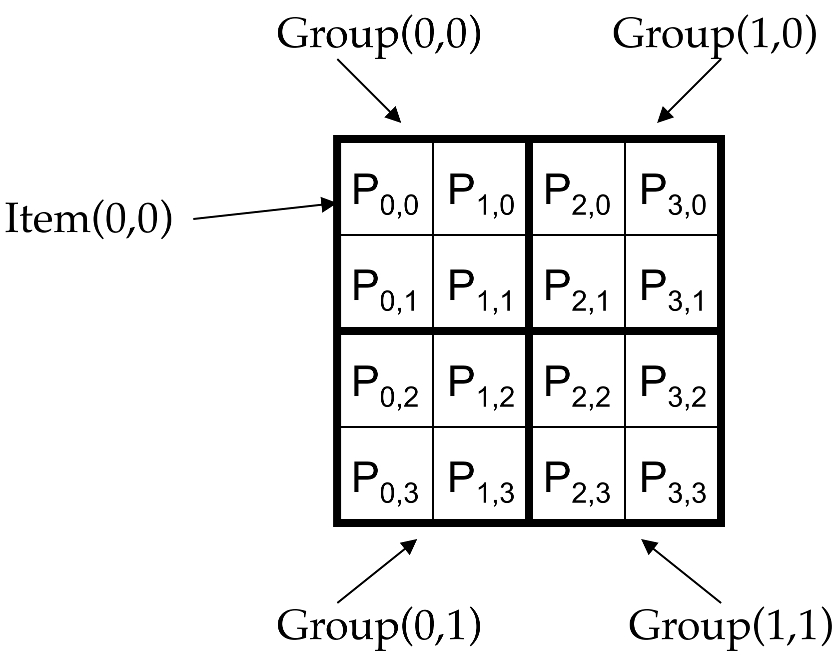
```
clReleaseMemObject(Md);
clReleaseMemObject(Nd);
clReleaseMemObject(Pd);
}
```

Matrix Multiplication Using Multiple Work Groups

- Break up Pd into tiles
- Each work group calculates one tile
  - Each work item calculates one element
  - Set work group size to tile size



A Very Small Example



WIDTH = 4; TILE_WIDTH = 2
Each work group has 2*2 = 4 work items

WIDTH/TILE_WIDTH = 2
Use 2*2 = 4 work groups

OpenCL Matrix Multiplication Kernel

```
__kernel void MatrixMulKernel(__global float* Md, __global
   float* Nd, __global float* Pd, int Width)
{
  // Calculate the row index of the Pd element and M
  int Row = get_global_id(1);
  // Calculate the column idenx of Pd and N
  int Col = get_global_id(0);

  float Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

  Pd[Row*Width+Col] = Pvalue;
}
```

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
size_t cl_DimBlock[2], cl_DimGrid[2];
cl_DimBlock[0] = TILE_WIDTH;
cl_DimBlock[1] = TILE_WIDTH;
cl_DimGrid[0] = Width;
cl_DimGrid[1] = Width;
clSetKernelArg(clkern, 0, sizeof (cl_mem), (void*)(&deviceP));
clSetKernelArg(clkern, 1, sizeof (cl_mem), (void*)(&deviceM));
clSetKernelArg(clkern, 2, sizeof (cl_mem), (void*)(&deviceN));
clSetKernelArg(clkern, 3, sizeof (int), (void *)(&Width));

// Launch the device kernel
clEnqueueNDRangeKernel(clcmdque, clkern, 2, NULL,
                       cl_DimGrid, cl_DimBlock, 0, NULL,
                       &DeviceDone);
```

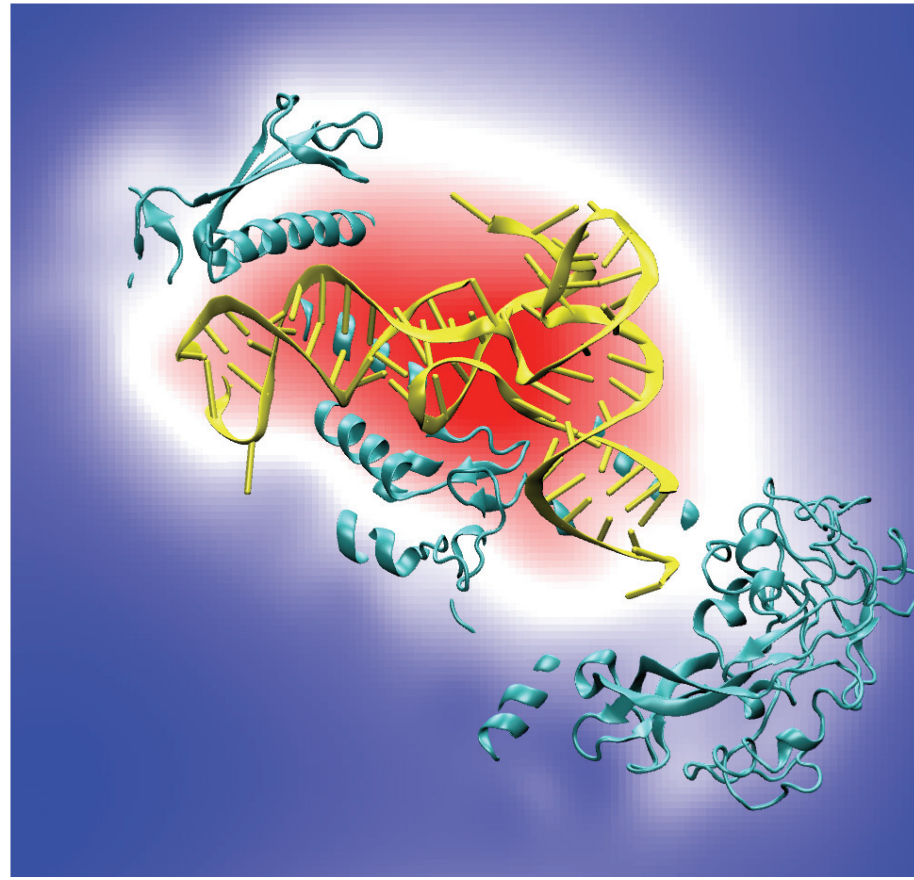
A Real Application Example -- Electrostatic Potential Maps

Electrostatic Potential Maps

- Electrostatic potentials evaluated on 3-D lattice:

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0 |\mathbf{r}_j - \mathbf{r}_i|}$$

- Applications include:
  - Ion placement for structure building
  - Time-averaged potentials for simulation
  - Visualization and analysis

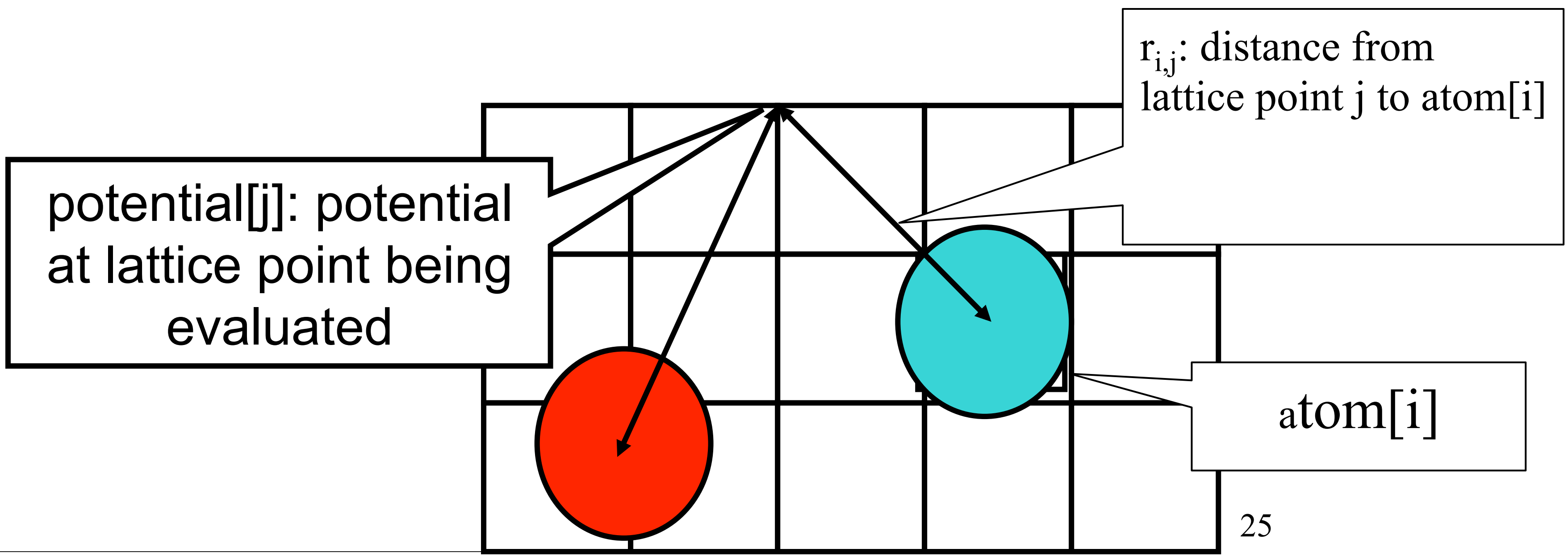


Isoleucine tRNA synthetase

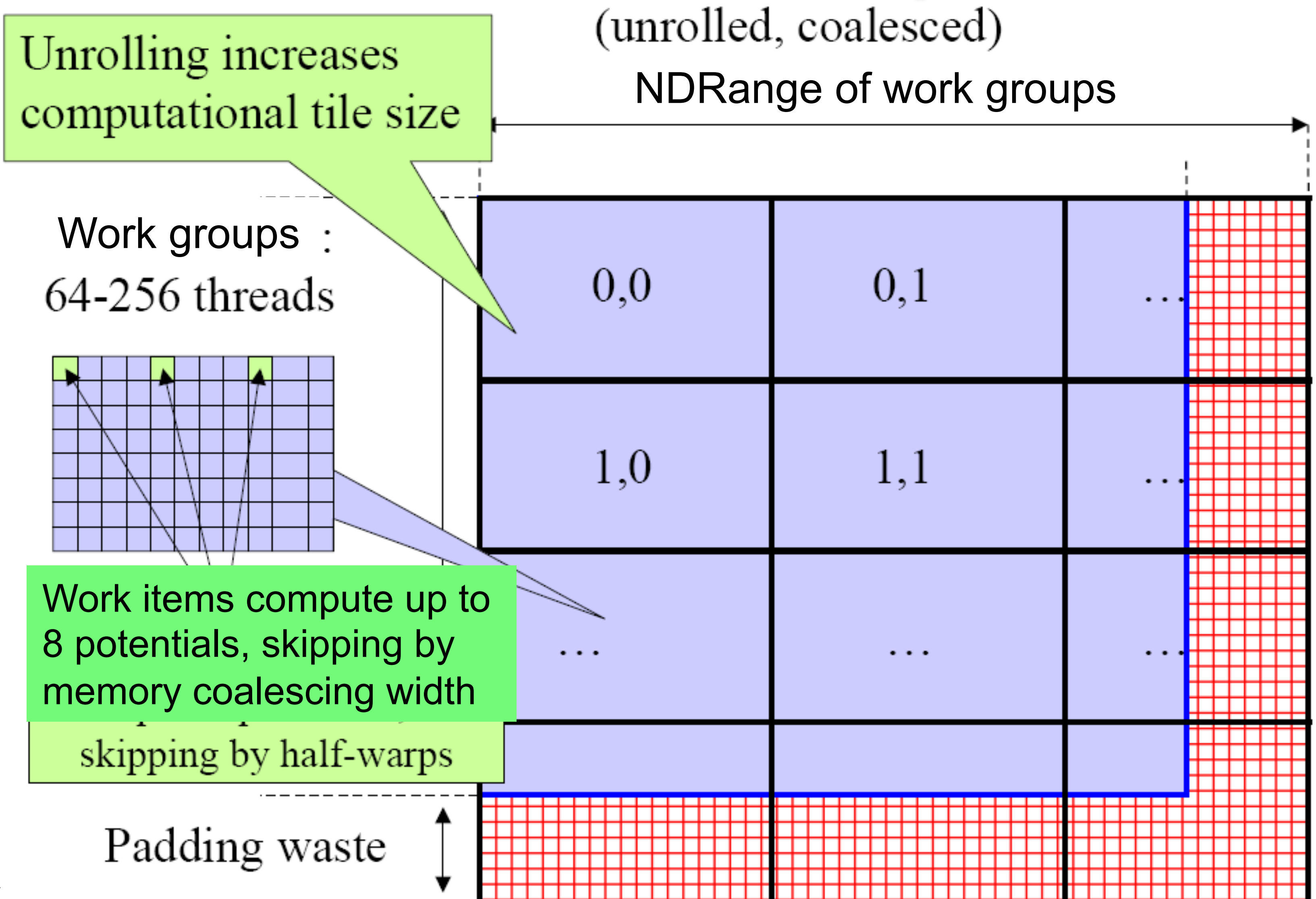
Direct Coulomb Summation

- **At each lattice point, sum potential contributions for all atoms in the simulated structure:**

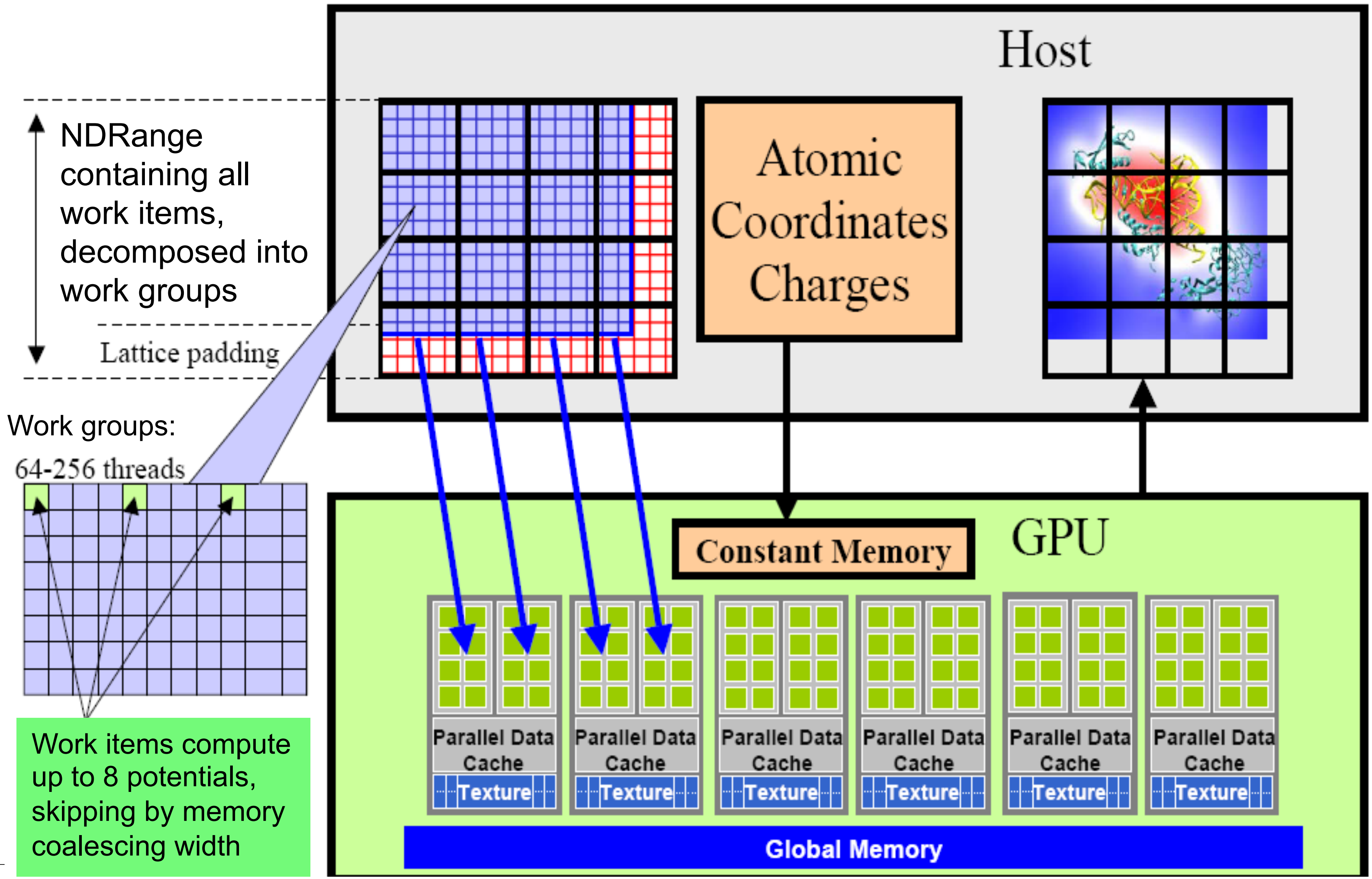
  $\text{potential[j]} \mathrel{+}= \text{charge[i]} / r_{ij}$



DCS Data Parallel Decomposition



Direct Coulomb Summation in OpenCL



Direct Coulomb Summation Kernel Setup

OpenCL:

```
__kernel void clenergy(…) {
  unsigned int xindex = (get_global_id(0) -
     get_local_id(0)) * UNROLLX +
     get_local_id(0);
  unsigned int yindex = get_global_id(1);
  unsigned int outaddr = get_global_size(0) *
     UNROLLX * yindex + xindex;
```

CUDA:

```
__global__ void cuenergy (…) {
  unsigned int xindex = blockIdx.x *
     blockDim.x * UNROLLX +
     threadIdx.x;
  unsigned int yindex = blockIdx.y *
     blockDim.y + threadIdx.y;
  unsigned int outaddr = gridDim.x *
     blockDim.x * UNROLLX *
     yindex + xindex;
```

DCS Inner Loop (CUDA)

```
…for (atomid=0; atomid<numatoms; atomid++) {
   float dy = coory - atominfo[atomid].y;
   float dyz2 = (dy * dy) + atominfo[atomid].z;
   float dx1 = coorx – atominfo[atomid].x;
   float dx2 = dx1 + gridspacing_coalesce;
   float dx3 = dx2 + gridspacing_coalesce;
   float dx4 = dx3 + gridspacing_coalesce;
   float charge = atominfo[atomid].w;
   energyvalx1 += charge * rsqrtf(dx1*dx1 + dyz2);
   energyvalx2 += charge * rsqrtf(dx2*dx2 + dyz2);
   energyvalx3 += charge * rsqrtf(dx3*dx3 + dyz2);
   energyvalx4 += charge * rsqrtf(dx4*dx4 + dyz2);
  }
```

DCS Inner Loop (OpenCL on NVIDIA GPU)

```
…for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx – atominfo[atomid].x;
    float dx2 = dx1 + gridspacing_coalesce;
    float dx3 = dx2 + gridspacing_coalesce;
    float dx4 = dx3 + gridspacing_coalesce;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
    energyvalx2 += charge * native_rsqrt(dx2*dx2 + dyz2);
    energyvalx3 += charge * native_rsqrt(dx3*dx3 + dyz2);
    energyvalx4 += charge * native_rsqrt(dx4*dx4 + dyz2);
  }
```

DCS Inner Loop (OpenCL on AMD CPU)

```
float4 gridspacing_u4 = { 0.f, 1.f, 2.f, 3.f };
gridspacing_u4 *= gridspacing_coalesce;
float4 energyvalx=0.0f;
…
for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float4 dx = gridspacing_u4 + (coorx – atominfo[atomid].x);
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
}
```

Why Two Different OpenCL Kernels???

- Existing OpenCL implementations don't necessarily auto-vectorize your code for the native hardware's SIMD vector width
- Although you can run the same code on very different devices and get the correct answer, performance will vary wildly...
- In many cases, getting peak performance on multiple device types or hardware from different vendors currently requires multiple OpenCL kernels

OpenCL Host Code

- **Roughly analogous to CUDA driver API:**
  - Memory allocations, memory copies, etc
  - Create and manage device context(s) and associated work queue(s), etc...
  - OpenCL uses reference counting on all objects
- **OpenCL programs are normally compiled entirely at runtime, which must be managed by host code**

OpenCL Kernel Compilation Example

OpenCL kernel source code as a big string

```
const char* clenergysrc =
  "__kernel __attribute__((reqd_work_group_size_hint(BLOCKSIZEX,
          BLOCKSIZEY, 1))) \n"
  "void clenergy(int numatoms, float gridspacing, __global float
      *energy, __constant float4 *atominfo) { \n" [...etc and so forth...]
cl_program clpgm;
```

Gives raw source code string(s) to OpenCL

```
clpgm = clCreateProgramWithSource(clctx, 1, &clenergysrc, NULL, &clerr);
char clcompileflags[4096];
sprintf(clcompileflags, "-DUNROLLX=%d -cl-fast-relaxed-math -cl-single-
precision-constant -cl-denorms-are-zero -cl-mad-enable", UNROLLX);
clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL, NULL);
cl_kernel clkern = clCreateKernel(clpgm, "clenergy", &clerr);
```

Set compiler flags, compile source, retreive handle to the "clenergy" kernel

Host Code: OpenCL Kernel Launch

```
1. doutput = clCreateBuffer(clctx, CL_MEM_READ_WRITE,volmemsz, NULL,      NULL);
2. datominfo = clCreateBuffer(clctx, CL_MEM_READ_ONLY, MAXATOMS
              *sizeof(cl_float4), NULL, NULL);
...
3. clerr= clSetKernelArg(clkern, 0,sizeof(int), &runatoms);
4. clerr= clSetKernelArg(clkern, 1,sizeof(float), &zplane);
5. clerr= clSetKernelArg(clkern, 2,sizeof(cl_mem), &doutput);
6. clerr= clSetKernelArg(clkern, 3,sizeof(cl_mem), &datominfo);
7. cl_event event;
8. clerr= clEnqueueNDRangeKernel(clcmdq,clkern, 2, NULL, Gsz,Bsz, 0,      NULL,
&event);
9. clerr= clWaitForEvents(1, &event);
10. clerr= clReleaseEvent(event);
...
11. clEnqueueReadBuffer(clcmdq,doutput, CL_TRUE, 0, volmemsz, energy, 0,      NULL,
NULL);
12. clReleaseMemObject(doutput);
13. clReleaseMemObject(datominfo);
```

To Learn More

- Khronos OpenCL headers, specification, etc: http://www.khronos.org/registry/cl/
- Khronos OpenCL samples, tutorials, etc: http://www.khronos.org/developers/resources/opencl/
- AMD OpenCL Resources: http://developer.amd.com/gpu/ATIStreamSDK/pages/Tutorial OpenCL.aspx
- NVIDIA OpenCL Resources: http://www.nvidia.com/object/cuda_opencl.html
- Chapter 11 of our textbook