Illinois UPCRC Summer School 2010

The OpenCL Programming Model

Part 2: Case Studies

Wen-mei Hwu and John Stone
with special contributions from
Deepthi Nandakumar

UPCRC Illinois
Universal Parallel Computing Research Center

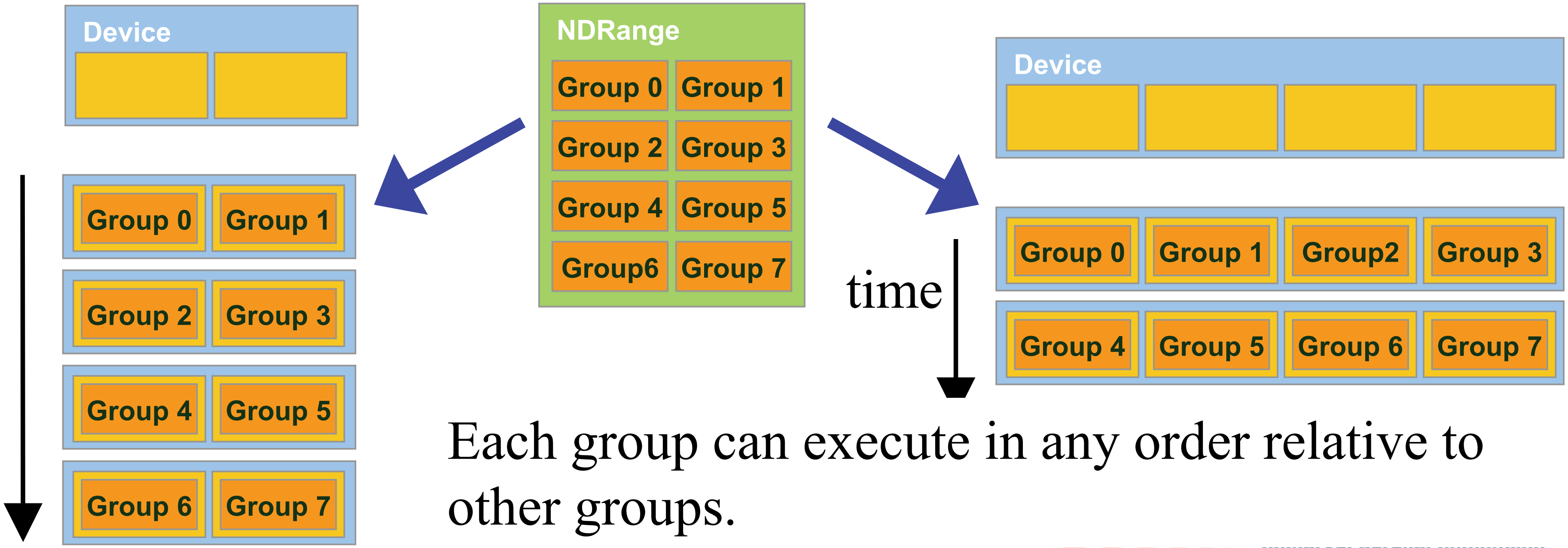
OpenCL Data Parallel Model

- 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 with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by launching a new kernel

UPCRC Illinois
Universal Parallel Computing Research Center

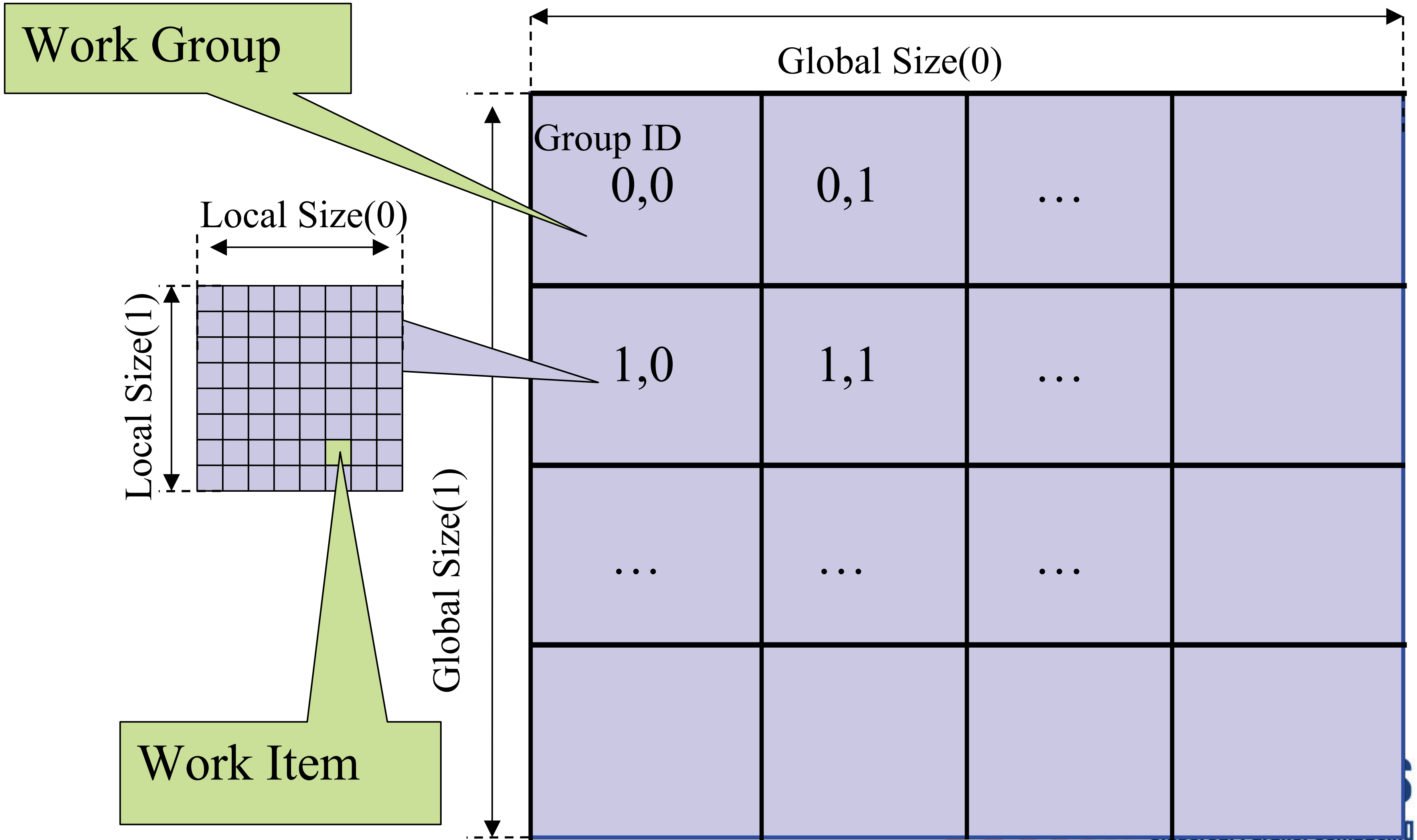
Transparent Scalability

- Hardware is free to assigns work groups to any processor at any time
  - A kernel scales across any number of parallel processors



Each group can execute in any order relative to other groups.

OpenCL NDRange Configuration



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

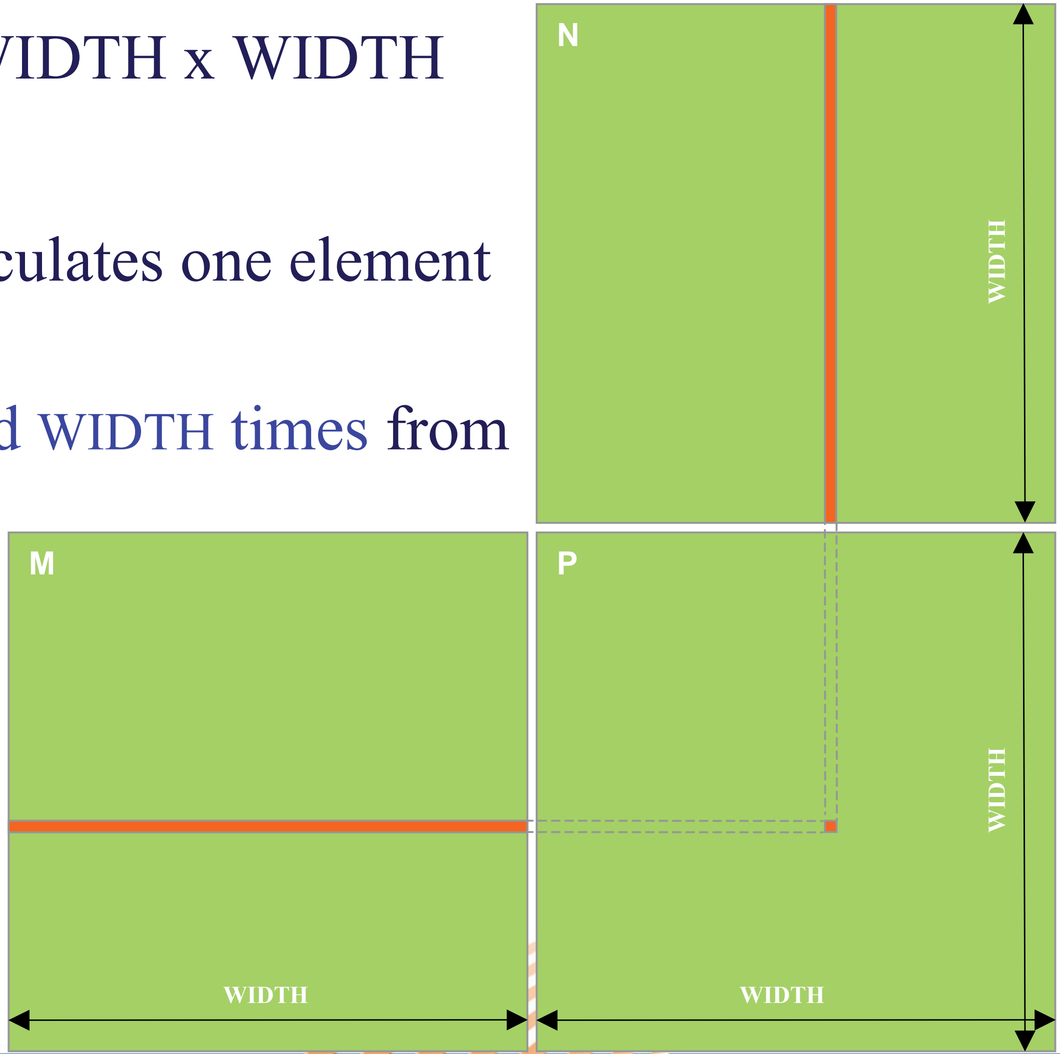
UPCRC Illinois
Universal Parallel Computing Research Center

A Simple Running 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

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One work item calculates one element of P
  - M and N are loaded WIDTH times from global memory



Memory Layout of a Matrix in C

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M

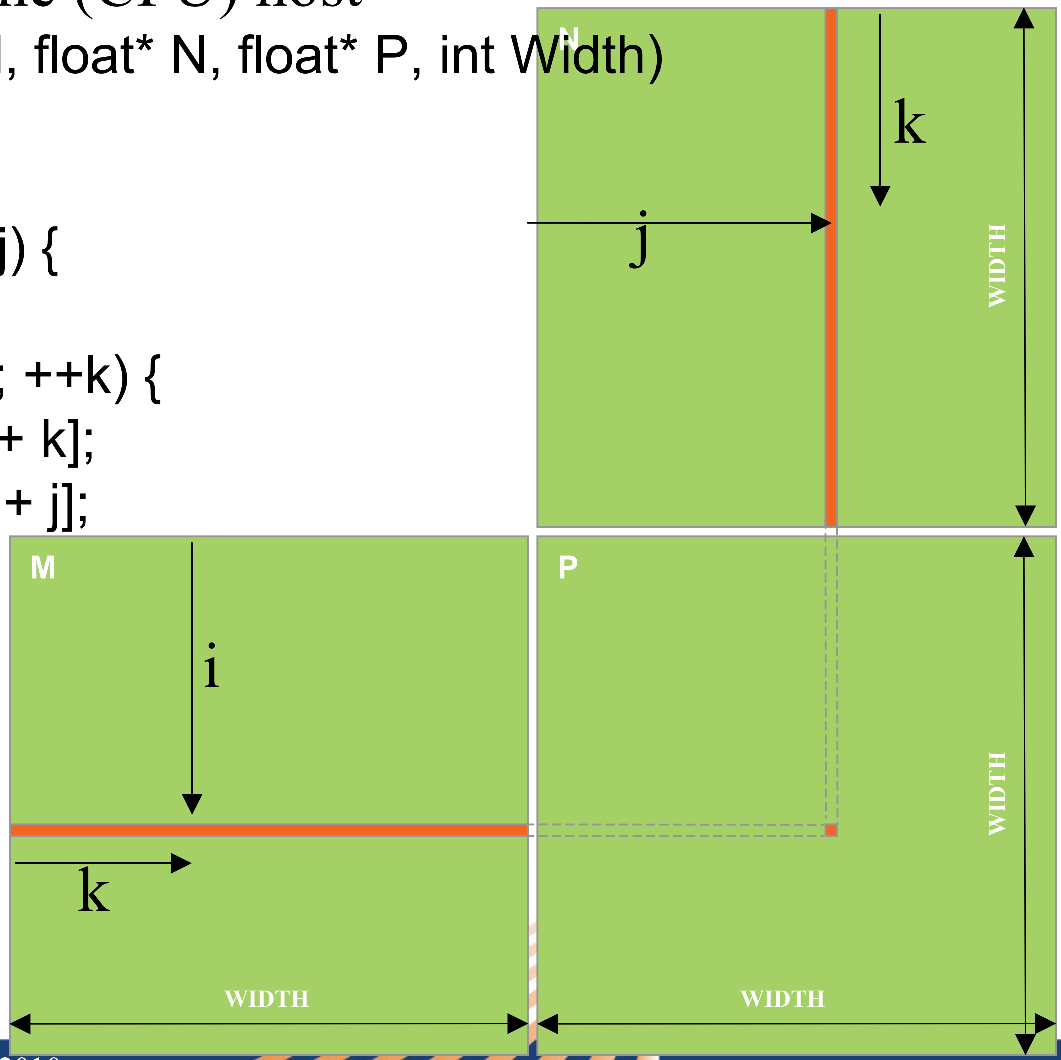
$\downarrow$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

UPCRC Illinois
Universal Parallel Computing Research Center

Step 1: Matrix Multiplication A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Step 2: 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);
```

UPCRC Illinois
Universal Parallel Computing Research Center

Step 3: 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);
```

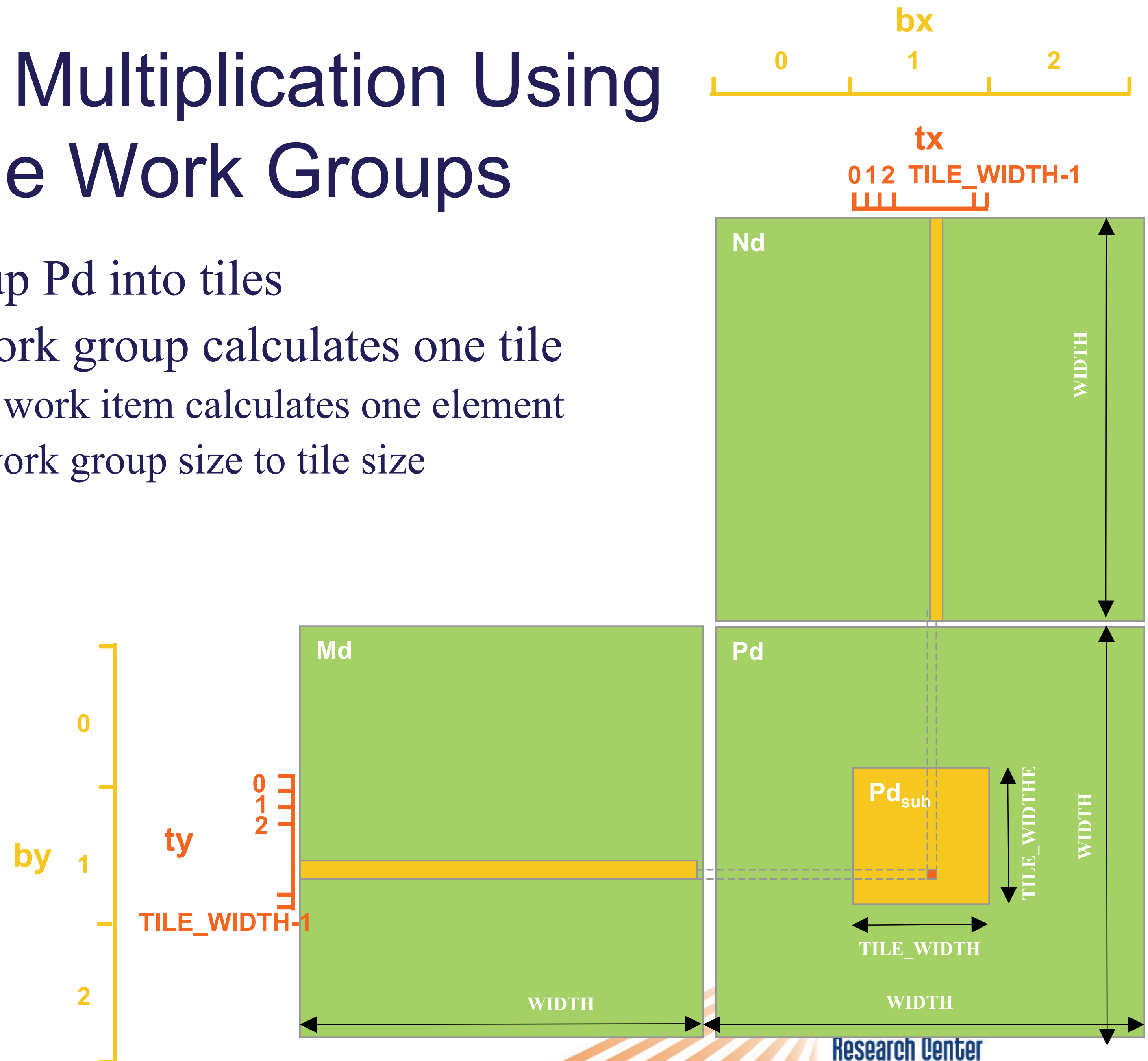
// Free device matrices

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

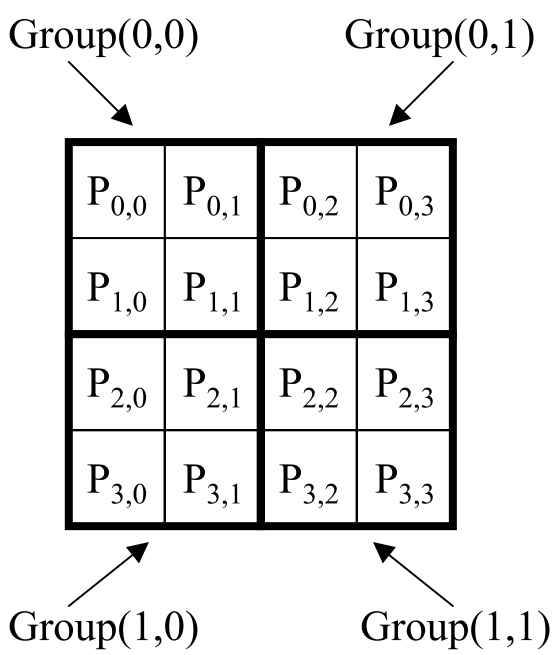
UPCRC Illinois
Universal Parallel Computing Research Center

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

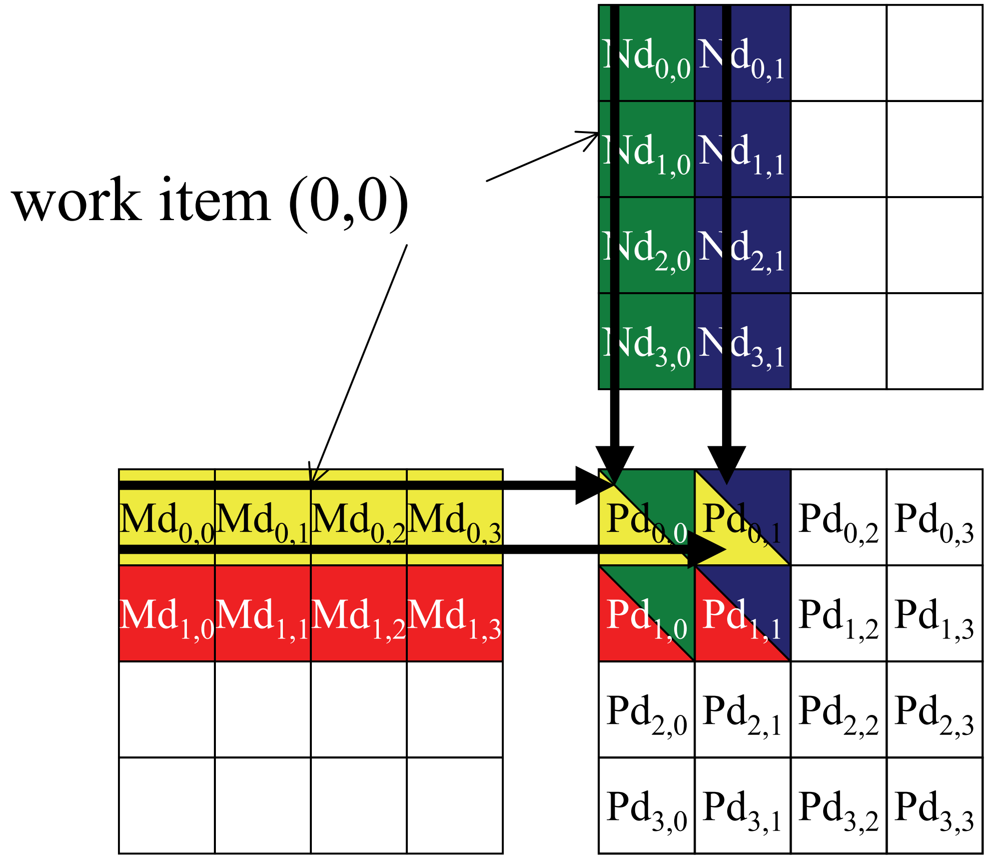


A Very Small Example



TILE_WIDTH = 2

A Very Small Example: Multiplication



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;
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Revised Step 5: 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);
```

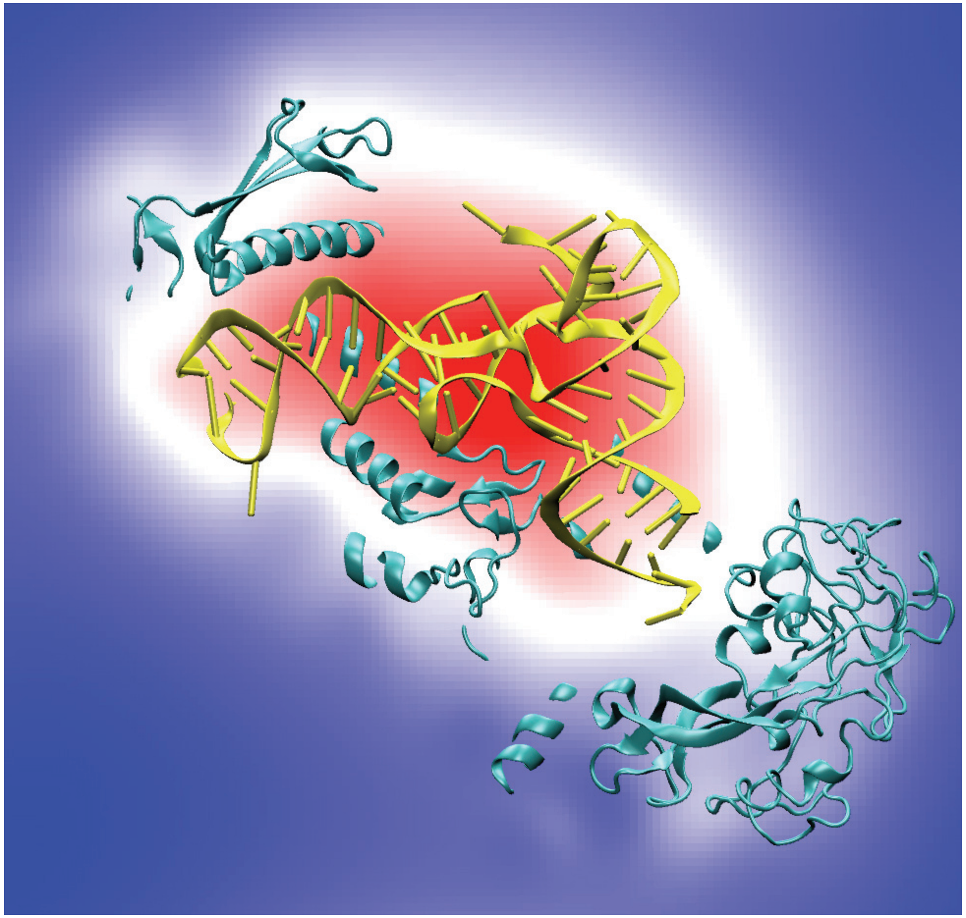
A Real Application Example

Electrostatic Potential Maps

- Electrostatic 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

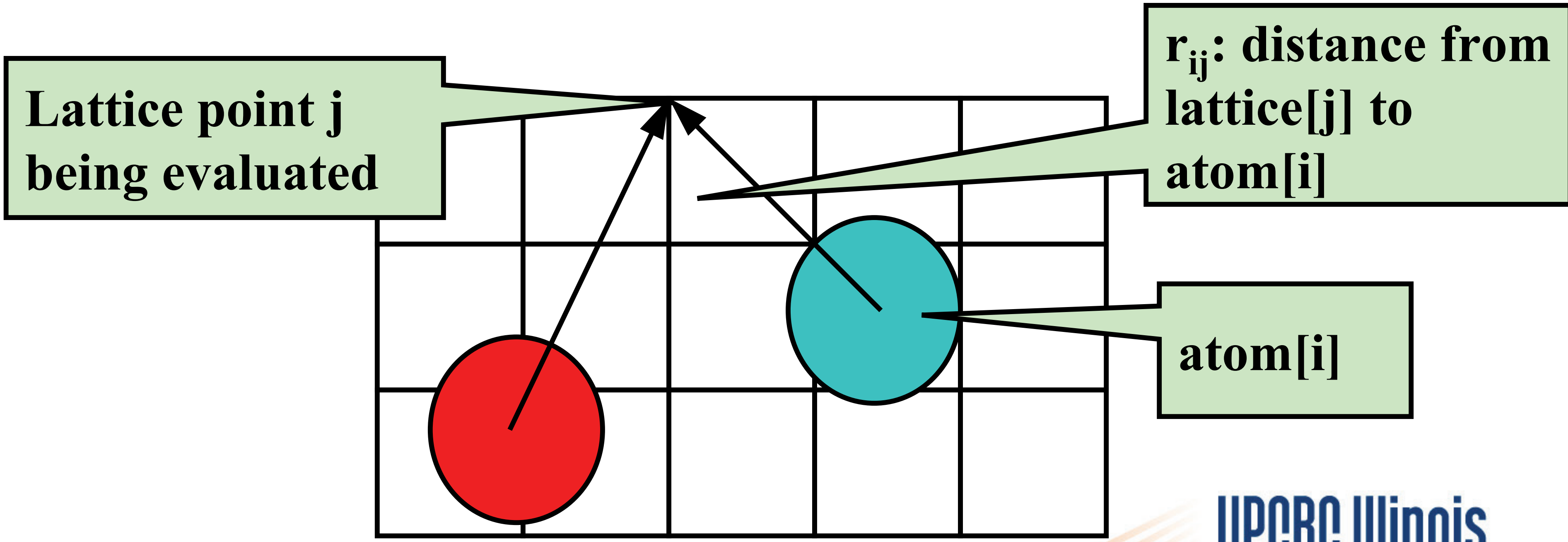


Isoleucine tRNA synthetase

Direct Coulomb Summation

- Each lattice point accumulates electrostatic potential contribution from all atoms:

  potential[j] += charge[i] / r_{ij}

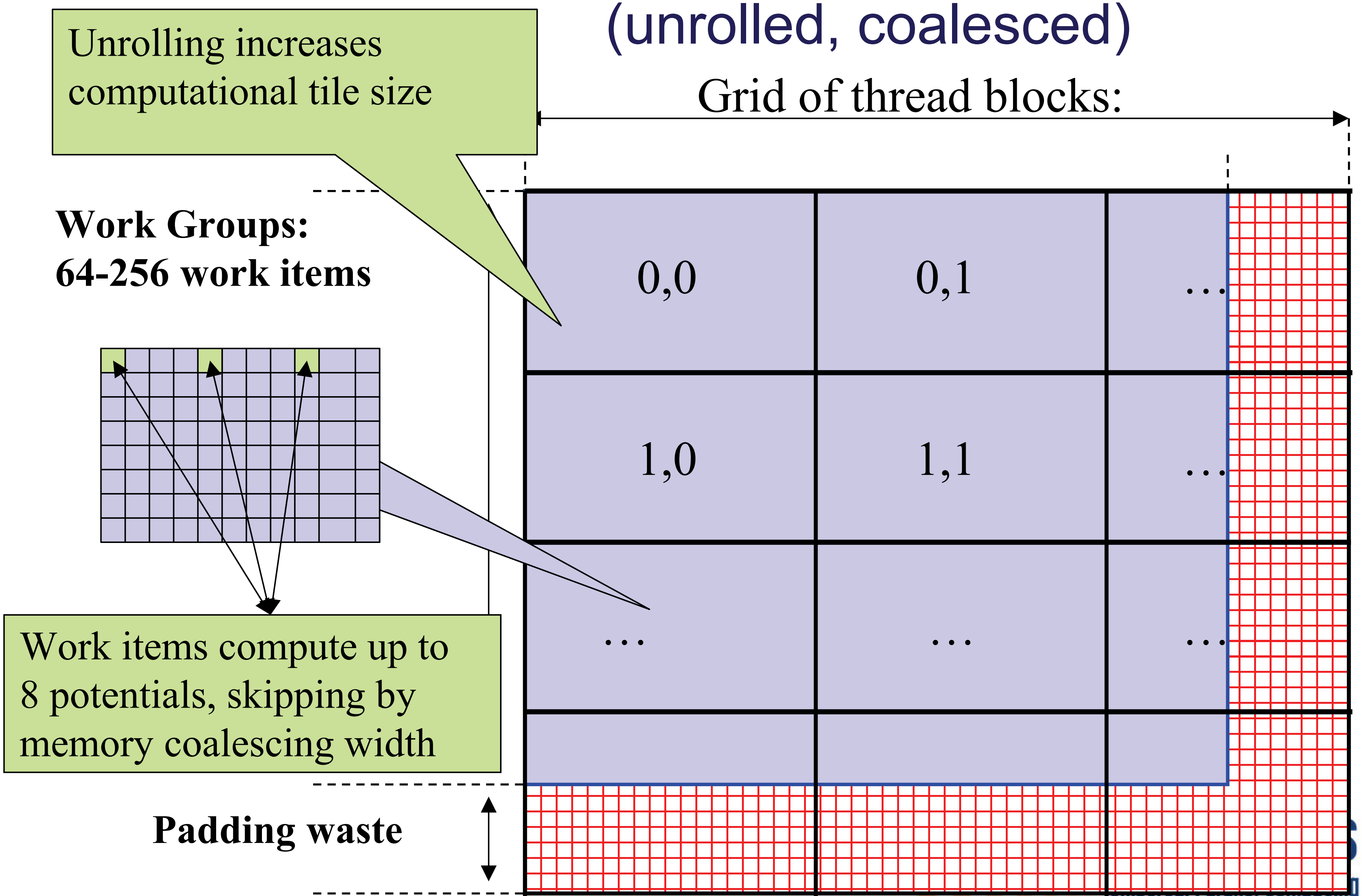


Data Parallel Direct Coulomb Summation Algorithm

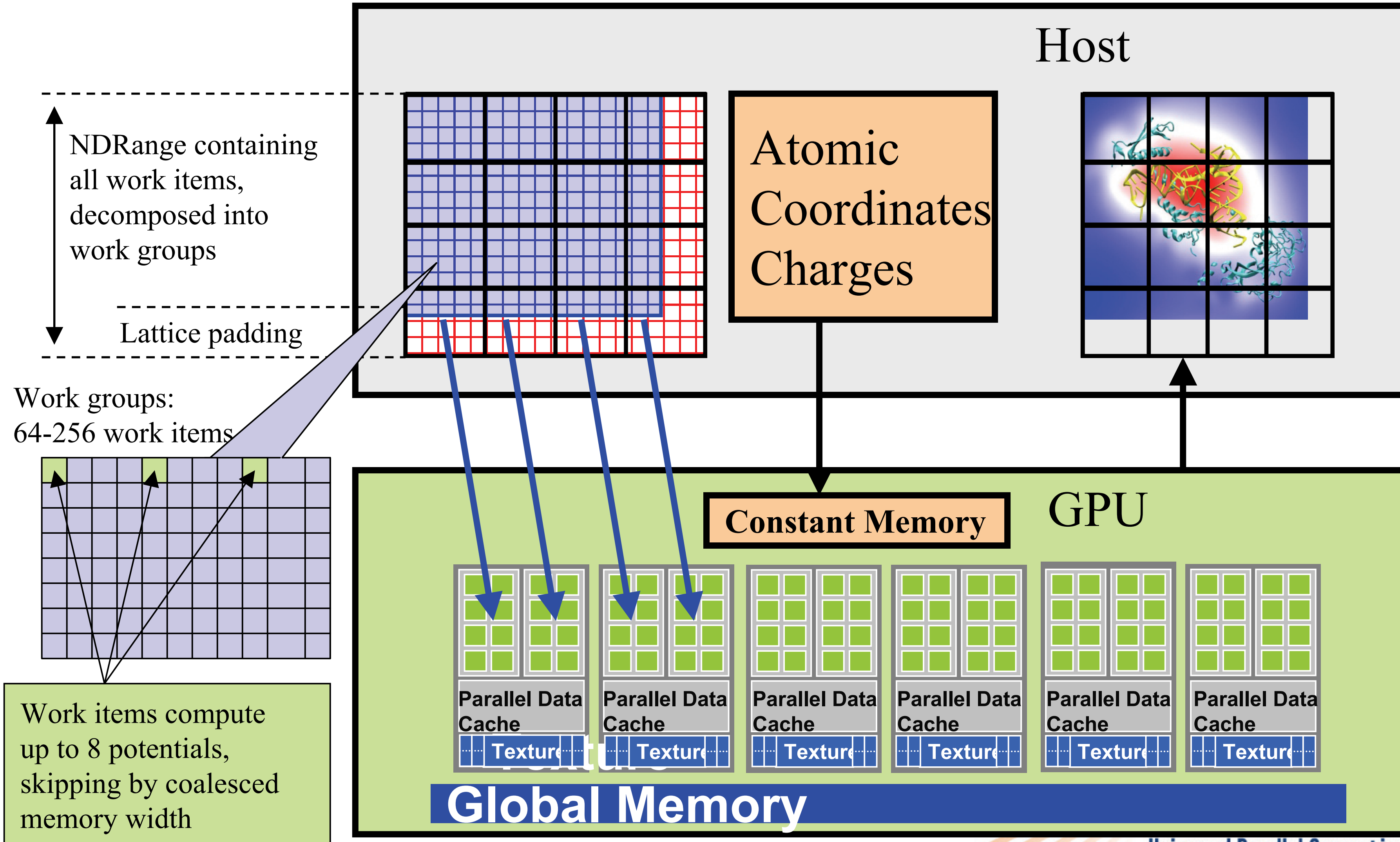
- Work is decomposed into tens of thousands of independent calculations
  - multiplexed onto all of the processing units on the target device (hundreds in the case of modern GPUs)
- Single-precision FP arithmetic is adequate for intended application
- Numerical accuracy can be improved by compensated summation, spatially ordered summation groupings, or accumulation of potential in double-precision
- Starting point for more sophisticated linear-time algorithms like multilevel summation

UPCRC Illinois
Universal Parallel Computing Research Center

DCS Data Parallel Decomposition (unrolled, coalesced)



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;
```

UPCRC Illinois
Universal Parallel Computing Research Center

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);
  }
```

UPCRC Illinois
Universal Parallel Computing Research Center

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);
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Wait a Second, Why Two Different OpenCL Kernels???

- Existing OpenCL implementations don't necessarily autovectorize 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 will presently require multiple OpenCL kernels

OpenCL Host Code

- Roughly analogous to CUDA driver API:
  - Memory allocations, memory copies, etc
  - Create and manage device context(s) and associate 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

UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;
cl_context clctx = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL, NULL,
    NULL, &clerr);

size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs,
    NULL);

cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0,
    &clerr);
```

UPCRC Illinois
Universal Parallel Computing Research Center

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 f
cl_program clpgm;
clpgm = clCreateProgramWithSource(c tx, 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);
```

Gives raw source code string(s) to OpenCL

Set compiler flags, compile source, and retreive a handle to the “clenergy” kernel

Host Code for 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);
```

Apples to Oranges Performance Results: OpenCL Direct Coulomb Summation Kernels

OpenCL Target Device	OpenCL "cores"	Scalar Kernel: Ported from original CUDA kernel	4-Vector Kernel: Replaced manually unrolled loop iterations with float4 vector ops
AMD 2.2GHz Opteron 148 CPU (a very old Linux test box)	1	0.30 Bevals/sec, 2.19 GFLOPS	0.49 Bevals/sec, 3.59 GFLOPS
Intel 2.2Ghz Core2 Duo, (Apple MacBook Pro)	2	0.88 Bevals/sec, 6.55 GFLOPS	2.38 Bevals/sec, 17.56 GFLOPS
IBM QS22 CellBE *** __constant not implemented yet	16	2.33 Bevals/sec, 17.16 GFLOPS ****	6.21 Bevals/sec, 45.81 GFLOPS ****
AMD Radeon 4870 GPU	10	41.20 Bevals/sec, 303.93 GFLOPS	31.49 Bevals/sec, 232.24 GFLOPS
NVIDIA GeForce GTX 285 GPU	30	75.26 Bevals/sec, 555.10 GFLOPS	73.37 Bevals/sec, 541.12 GFLOPS

MADD, RSQRT = 2 FLOPS All other FP instructions = 1 FLOP

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/TutorialOpenCL.aspx
- NVIDIA OpenCL Resources: http://www.nvidia.com/object/cuda_opencl.html
- Kirk and Hwu, "Programming Massively Parallel Processors – a Hands-on Approach," Morgan-Kaufman, ISBN: 978-0-12-381472-2

Summary

- Incorporating OpenCL into an application requires adding far more "plumbing" in an application than for the CUDA Runtime API
- Although OpenCL code is portable in terms of correctness, performance of any particular kernel is not guaranteed across different device types/vendors
- Apps have to check performance-related properties of target devices, e.g. whether __local memory is fast/slow (query CL_DEVICE_LOCAL_MEM_TYPE)
- It remains to be seen how OpenCL "platforms" will allow apps to concurrently use an AMD CPU runtime and NVIDIA GPU runtime (may already work on MacOS X?)

Acknowledgements

- Additional Information and References:
  - http://www.ks.uiuc.edu/Research/gpu/
- Questions, source code requests:
  - John Stone: johns@ks.uiuc.edu
- Acknowledgements:
  - J. Phillips, D. Hardy, J. Saam,
    UIUC Theoretical and Computational Biophysics Group,
    NIH Resource for Macromolecular Modeling and Bioinformatics
  - Christopher Rodrigues, UIUC IMPACT Group
  - CUDA team at NVIDIA
  - UIUC NVIDIA CUDA Center of Excellence
  - NIH support: P41-RR05969

UPCRC Illinois
Universal Parallel Computing Research Center

Publications

http://www.ks.uiuc.edu/Research/gpu/

- Probing Biomolecular Machines with Graphics Processors. J. Phillips, J. Stone. *Communications of the ACM,* 52(10):34-41, 2009.
- GPU Clusters for High Performance Computing. V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu. *Workshop on Parallel Programming on Accelerator Clusters (PPAC),* IEEE Cluster 2009. In press.
- Long time-scale simulations of in vivo diffusion using GPU hardware. E. Roberts, J. Stone, L. Sepulveda, W. Hwu, Z. Luthey-Schulten. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Computing*, pp. 1-8, 2009.
- High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs. J. Stone, J. Saam, D. Hardy, K. Vandivort, W. Hwu, K. Schulten, *2nd Workshop on General-Purpose Computation on Graphics Pricessing Units (GPGPU-2), ACM International Conference Proceeding Series*, volume 383, pp. 9-18, 2009.
- Multilevel summation of electrostatic potentials using graphics processing units. D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

Publications (cont)
http://www.ks.uiuc.edu/Research/gpu/

- Adapting a message-driven parallel application to GPU-accelerated clusters. J. Phillips, J. Stone, K. Schulten. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, 2008.
- GPU acceleration of cutoff pair potentials for molecular modeling applications. C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W. Hwu. *Proceedings of the 2008 Conference On Computing Frontiers*, pp. 273-282, 2008.
- GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.
- Accelerating molecular modeling applications with graphics processors. J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.
- Continuous fluorescence microphotolysis and correlation spectroscopy. A. Arkhipov, J. Hüve, M. Kahms, R. Peters, K. Schulten. *Biophysical Journal*, 93:4006-4017, 2007.