OpenCL: Molecular Modeling on Heterogeneous Computing Systems

John Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

University of Illinois at Urbana-Champaign

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

Fall National Meeting of the American Chemical Society,
Boston, MA, August 22, 2010

What is OpenCL?

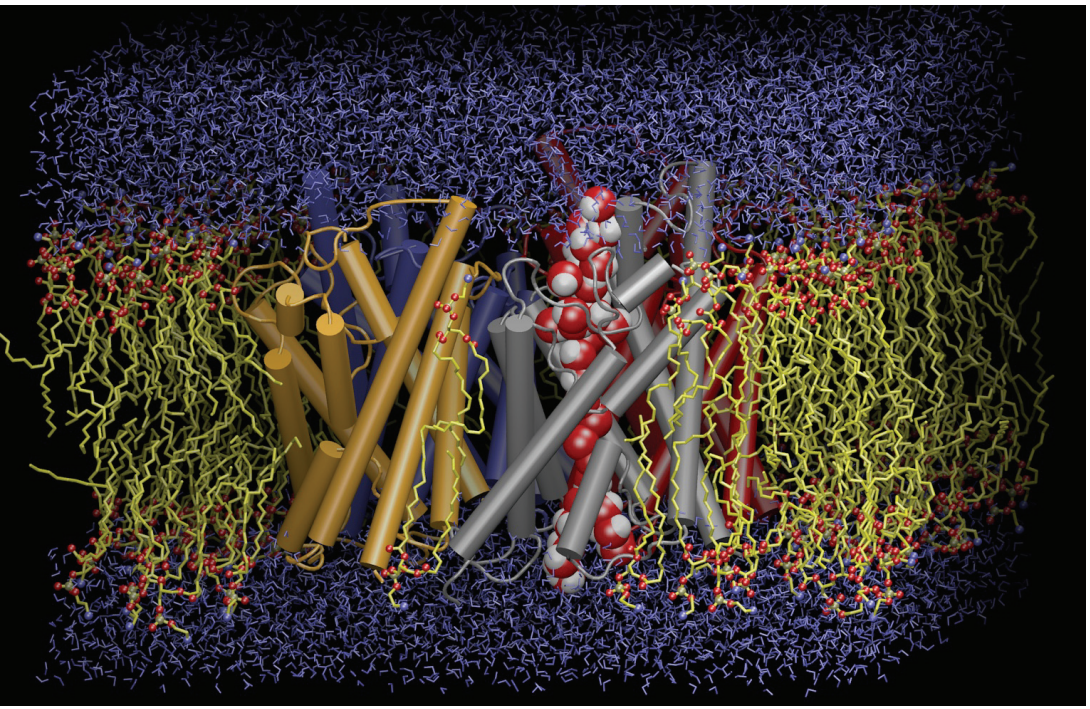
- Cross-platform parallel computing API and C-like language for heterogeneous computing devices
- Code is portable across various target devices:
  - Correctness is guaranteed
  - Performance of a given kernel is NOT guaranteed across differing target devices
- OpenCL implementations are available for AMD/Intel x86 CPUs, IBM Cell and Power, and both AMD and NVIDIA GPUs
- In principle, OpenCL could also target DSPs, and perhaps also FPGAs

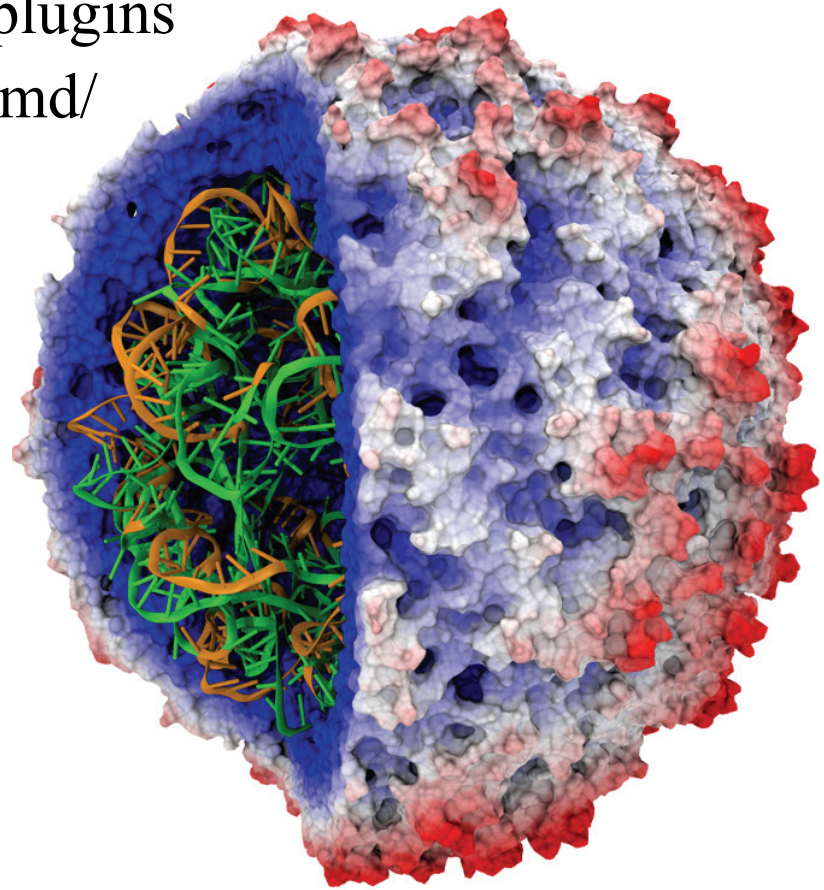
Current Impact and Future Outlook for Heterogeneous Computing

- 3 of the top 10 supercomputers in the latest Top500 list are heterogeneous systems
- Top 8 systems in the Green500 list are heterogeneous systems, top 8 are 3x more efficient than average of all Green500 systems!
- Performance and power efficiency appear to favor heterogeneous systems combining multi-core CPUs with GPUs or other massively parallel co-processors
- Supercomputer architects are forecasting that future petascale and exascale machines could be dominated by heterogeneous systems

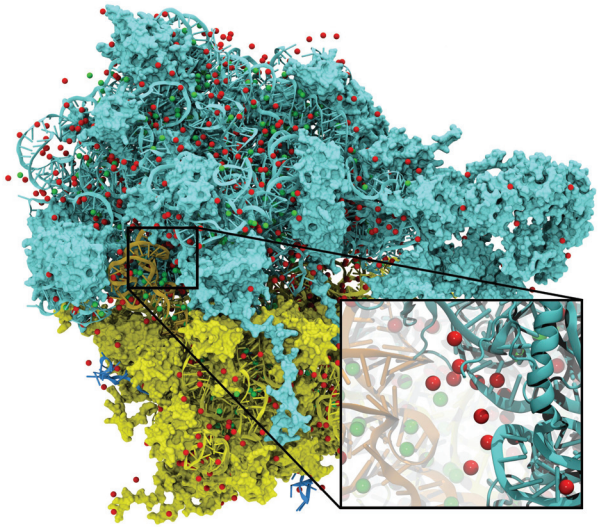
VMD – "Visual Molecular Dynamics"

- Visualization and analysis of molecular dynamics simulations, sequence data, volumetric data, quantum chemistry simulations, particle systems, …
- User extensible with scripting and plugins
- http://www.ks.uiuc.edu/Research/vmd/



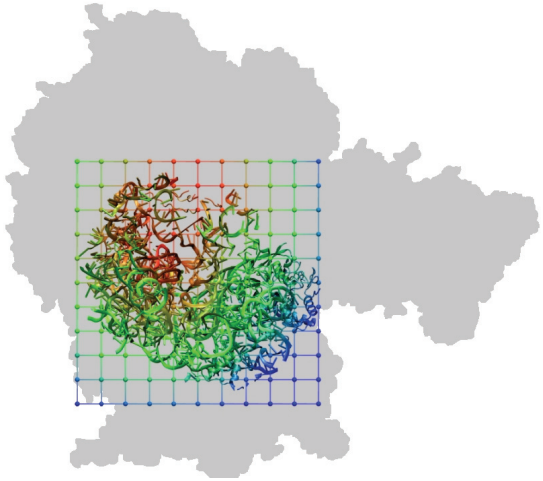


GPU Algorithms in VMD



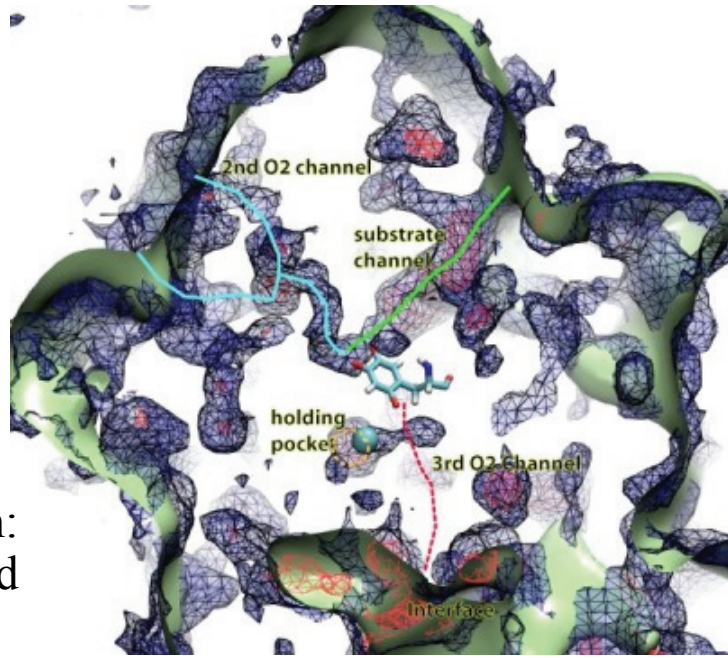
Ion placement

20x to 44x faster



Electrostatic field calculation: multilevel summation method

31x to 44x faster

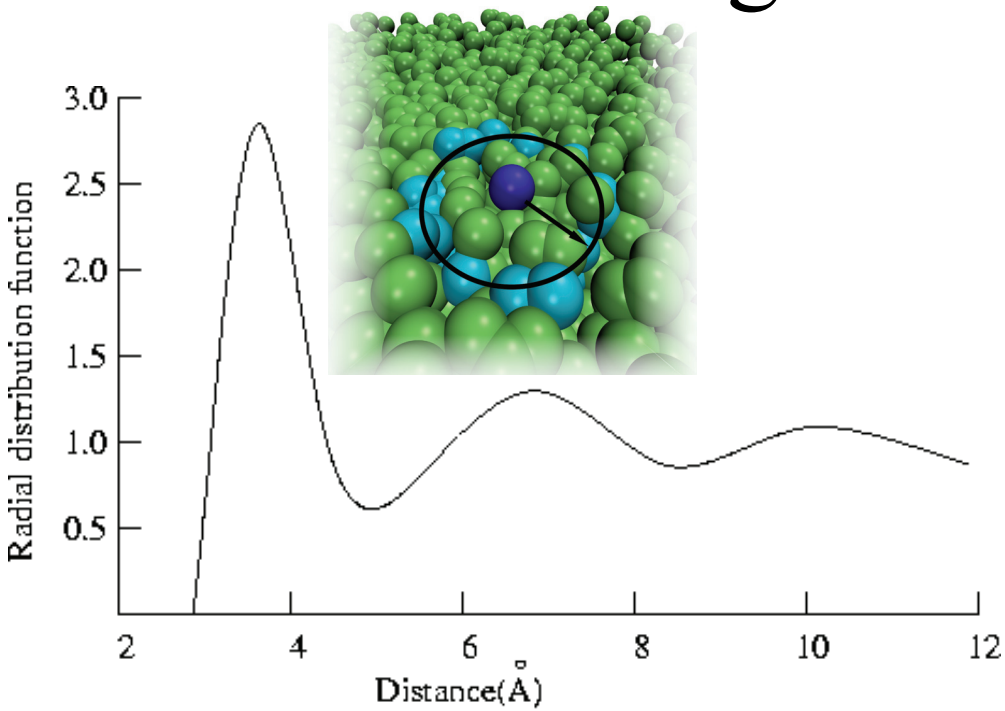


Imaging of gas migration pathways in proteins with implicit ligand sampling

20x to 30x faster

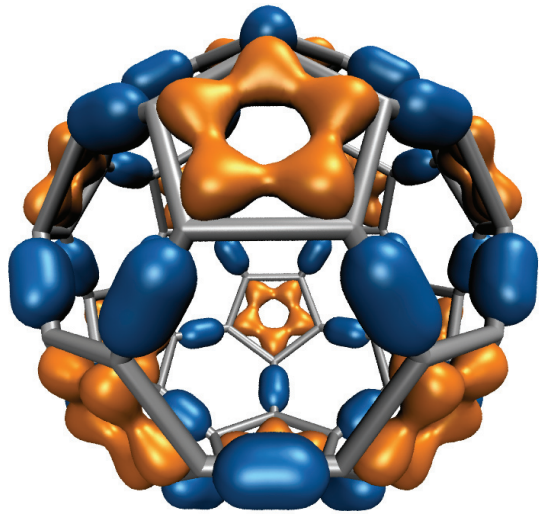
GPU: massively parallel co-processor

GPU Algorithms in VMD



Radial distribution functions

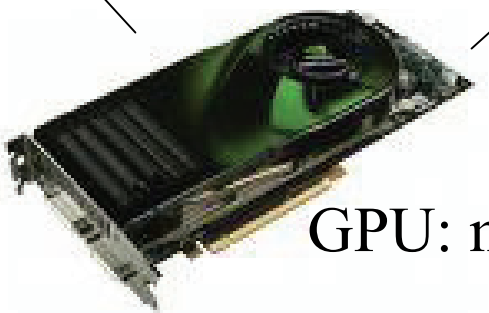
30x to 70x faster



Molecular orbital

calculation and display

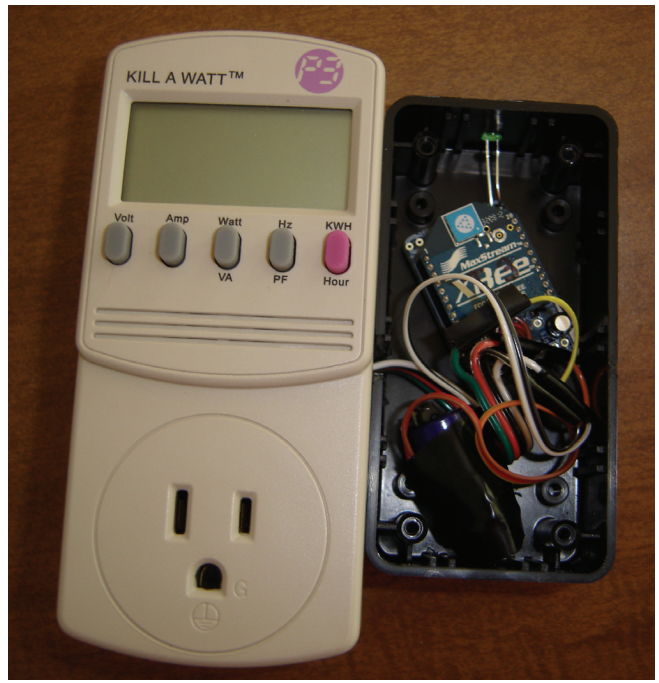
100x to 120x faster



GPU: massively parallel co-processor

Trajectory Analysis on NCSA GPU Cluster with MPI-enabled VMD

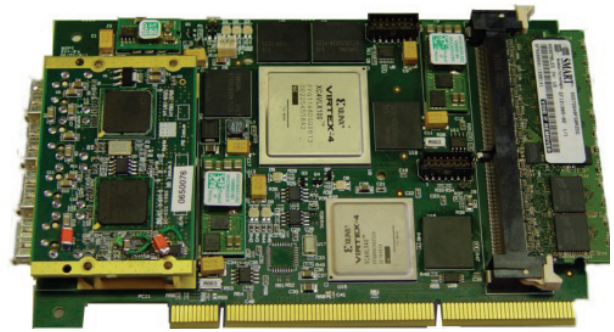
- Short time-averaged electrostatic field test case (few hundred frames, 700,000 atoms)
- 1:1 CPU/GPU ratio
- Power measured on a single node w/ NCSA monitoring tools
- CPUs-only: 1465 sec, 299 watts
- CPUs+GPUs: 57 sec, 742 watts
- Speedup 25.5 x
- Power efficiency gain: 10.5 x

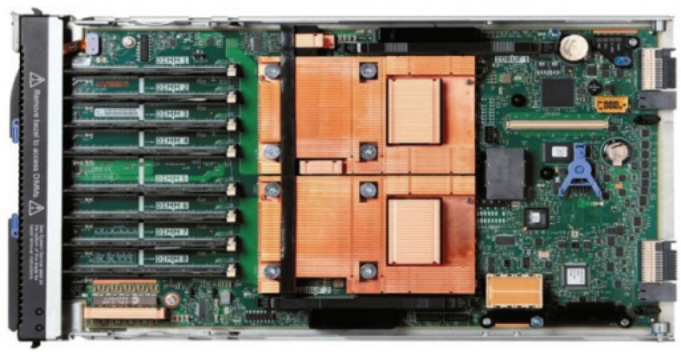


NCSA Tweet-a-watt power monitoring device

Supporting Diverse Co-Processor and Accelerator Hardware in Production Codes

- Development of HPC-oriented scientific software is already challenging
- Maintaining code in unique languages for each accelerator type is costly and impractical beyond a certain point
- Diversity and rapid evolution of CPU vector extensions and accelerators exacerbates these issues
- OpenCL ameliorates some key problems:
  - Supports CPUs, GPUs, other devices
  - Common language for writing computational "kernels"
  - Common API for managing execution on target device



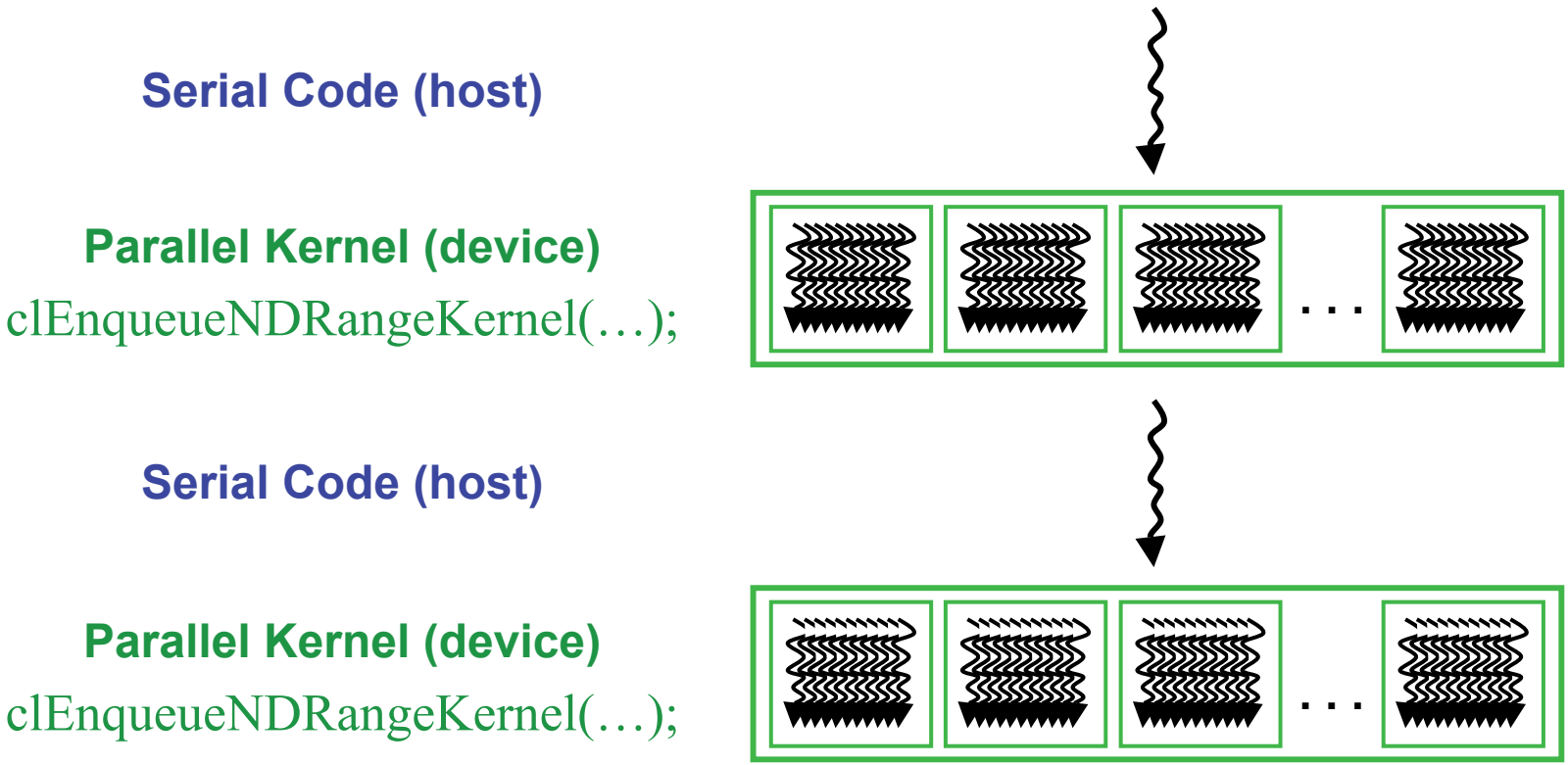


OpenCL Hardware Support

- Targets a broad range of CPU-like and GPU-like devices:
  - OpenCL now available for x86 CPUs, Power CPUs, Cell, and both AMD and NVIDIA GPUs
  - Some OpenCL features are optional and aren't supported on all devices, e.g. double-precision arithmetic
- OpenCL codes must be prepared to deal with much greater hardware diversity than conventional CPU, or even GPU codes
- A single OpenCL kernel will likely not achieve peak performance on all devices

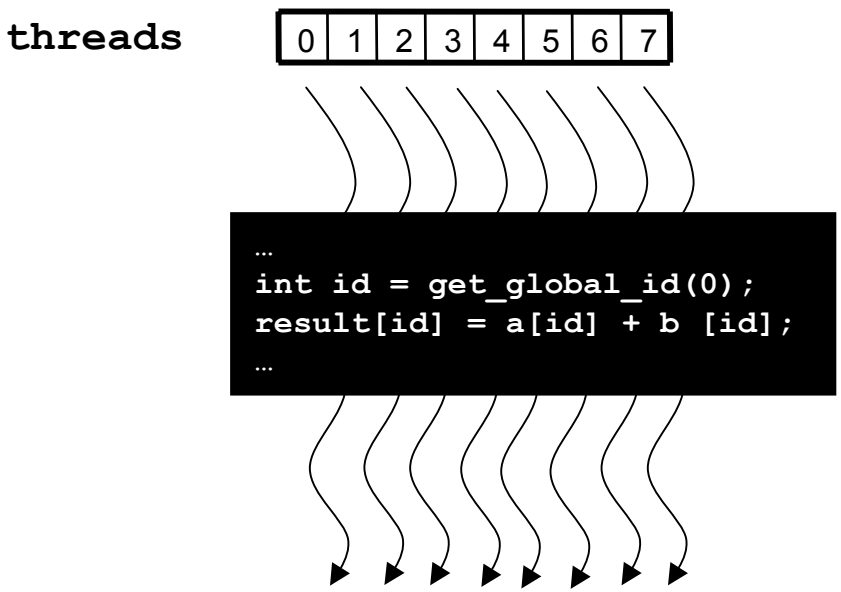
OpenCL Execution Model

- Integrated host+device application
  - Serial or modestly parallel parts in **host** code
  - Highly parallel parts in **device** SPMD kernel code



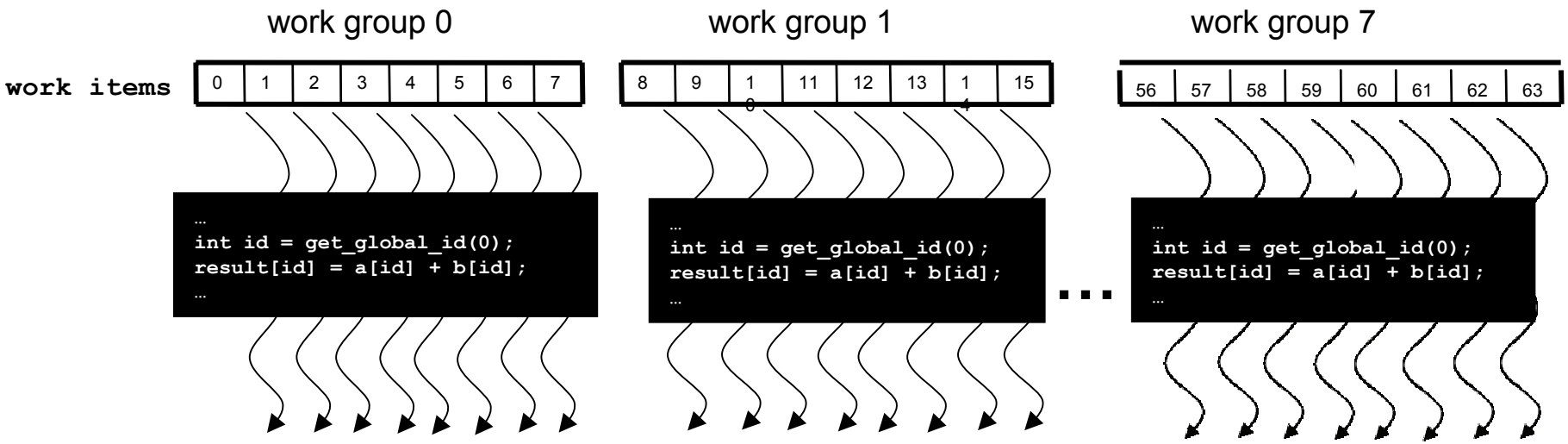
Array of Parallel Work Items

- An OpenCL kernel executes an array of work items
  - All work items run the same code (SPMD)
  - Each work item has an index that it uses to compute memory addresses and make control decisions

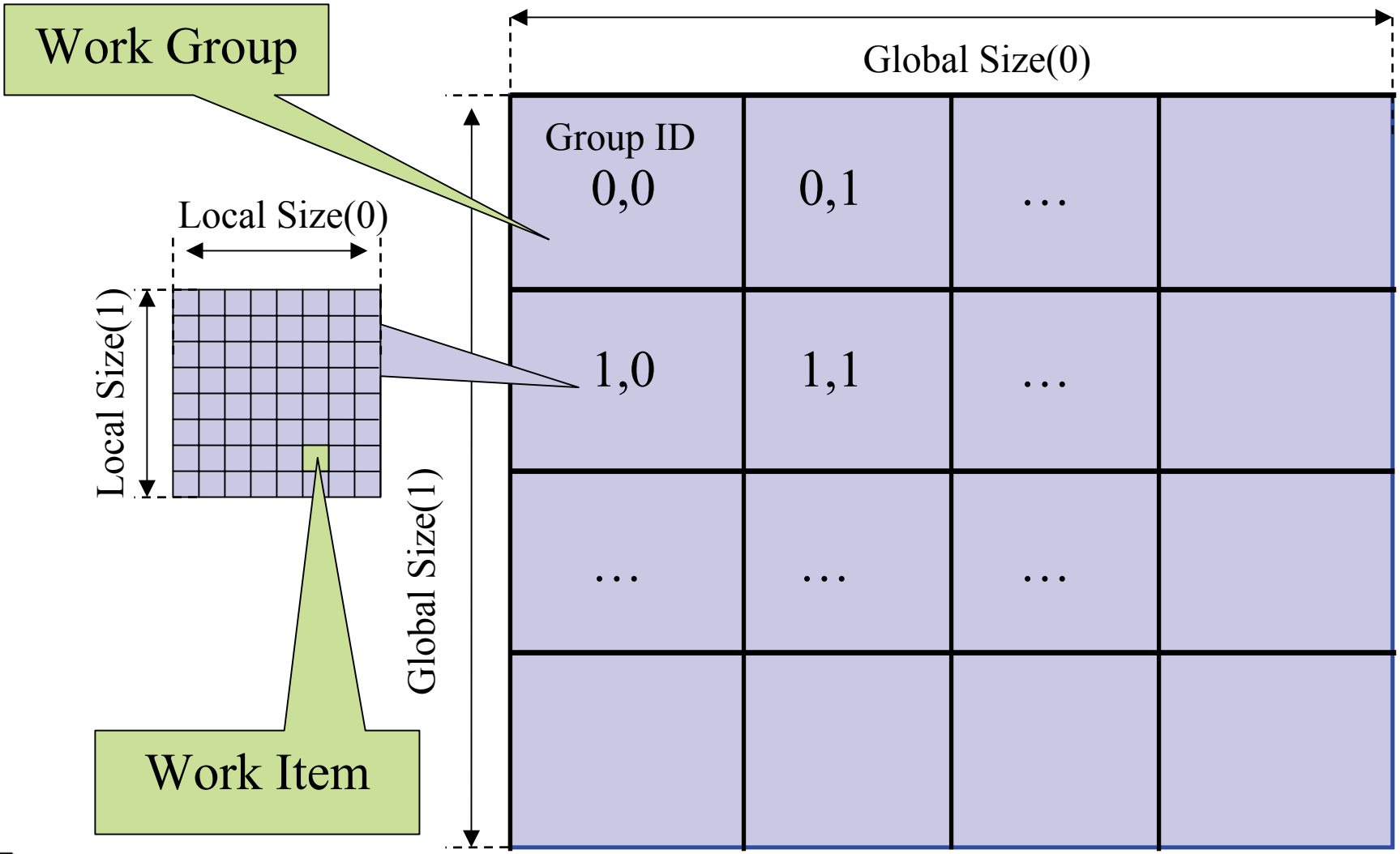


Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via **shared memory, atomic operations,** and **barrier synchronization**
  - Work items in different work groups cannot cooperate

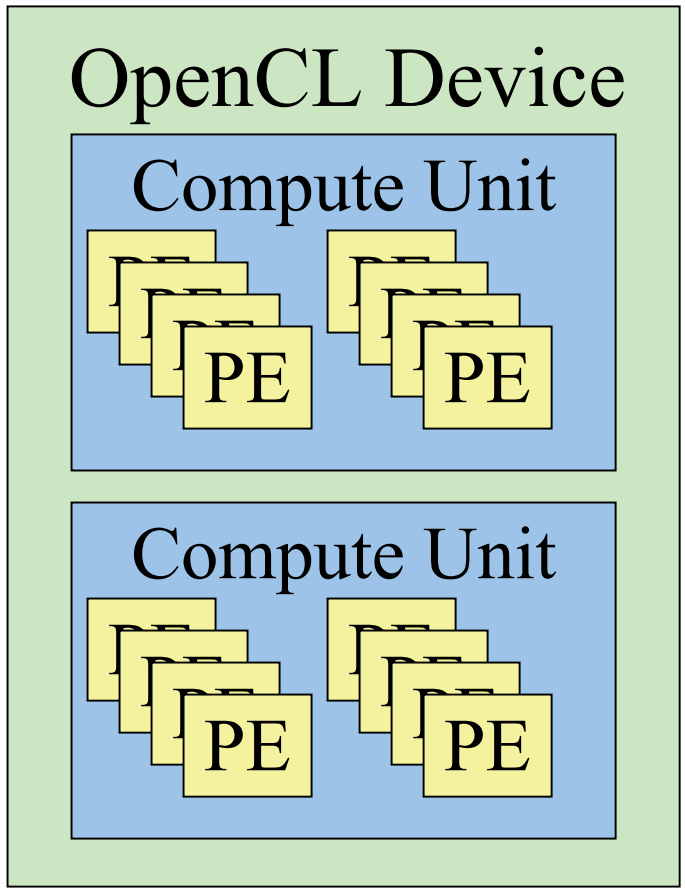


OpenCL NDRange Configuration



OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each “device” contains one or more “compute units”, i.e. cores, SMs, etc...
- Each “compute unit” contains one or more SIMD “processing elements”

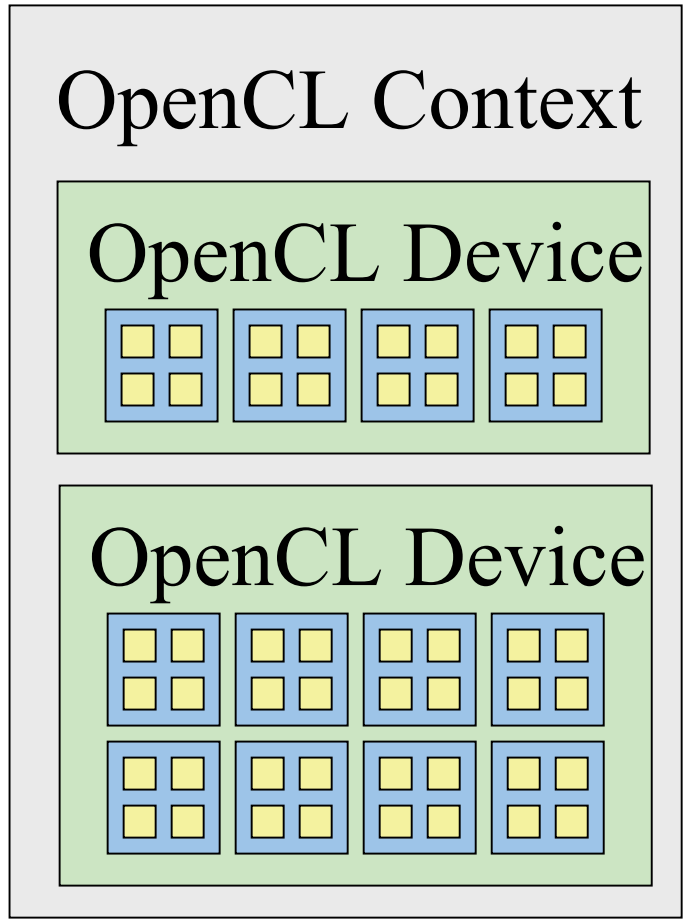


OpenCL Memory Systems

- __global – large, high 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
- Device memory is managed explicitly by the programmer, as with CUDA
- Pinned memory buffer allocations are created using the CL_MEM_USE_HOST_PTR flag

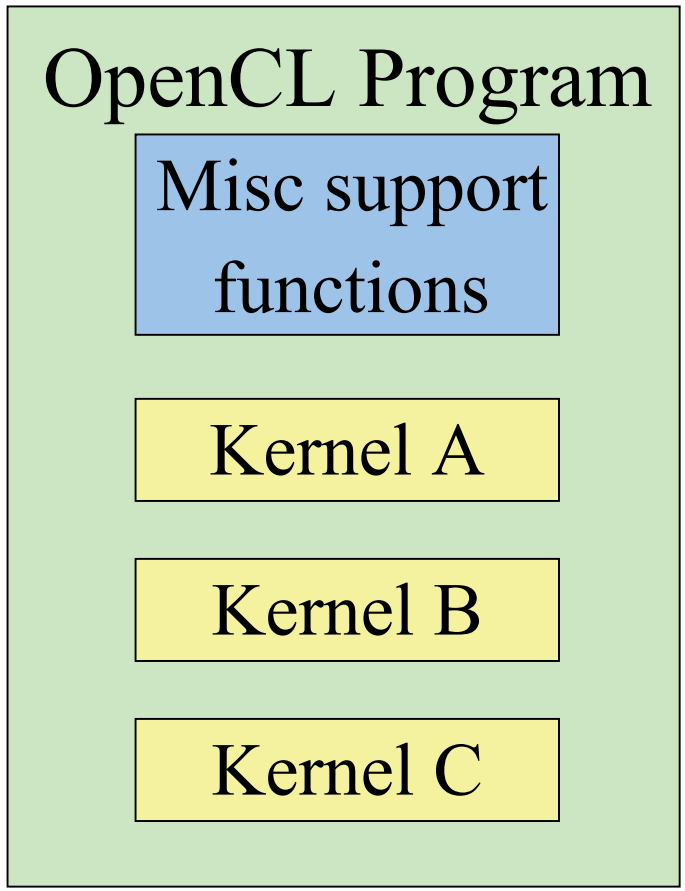
OpenCL Context

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



OpenCL Programs

- An OpenCL "program" contains one or more "kernels" and any supporting routines that run on a target device
- An OpenCL kernel is the basic unit of code that can be executed on a target device

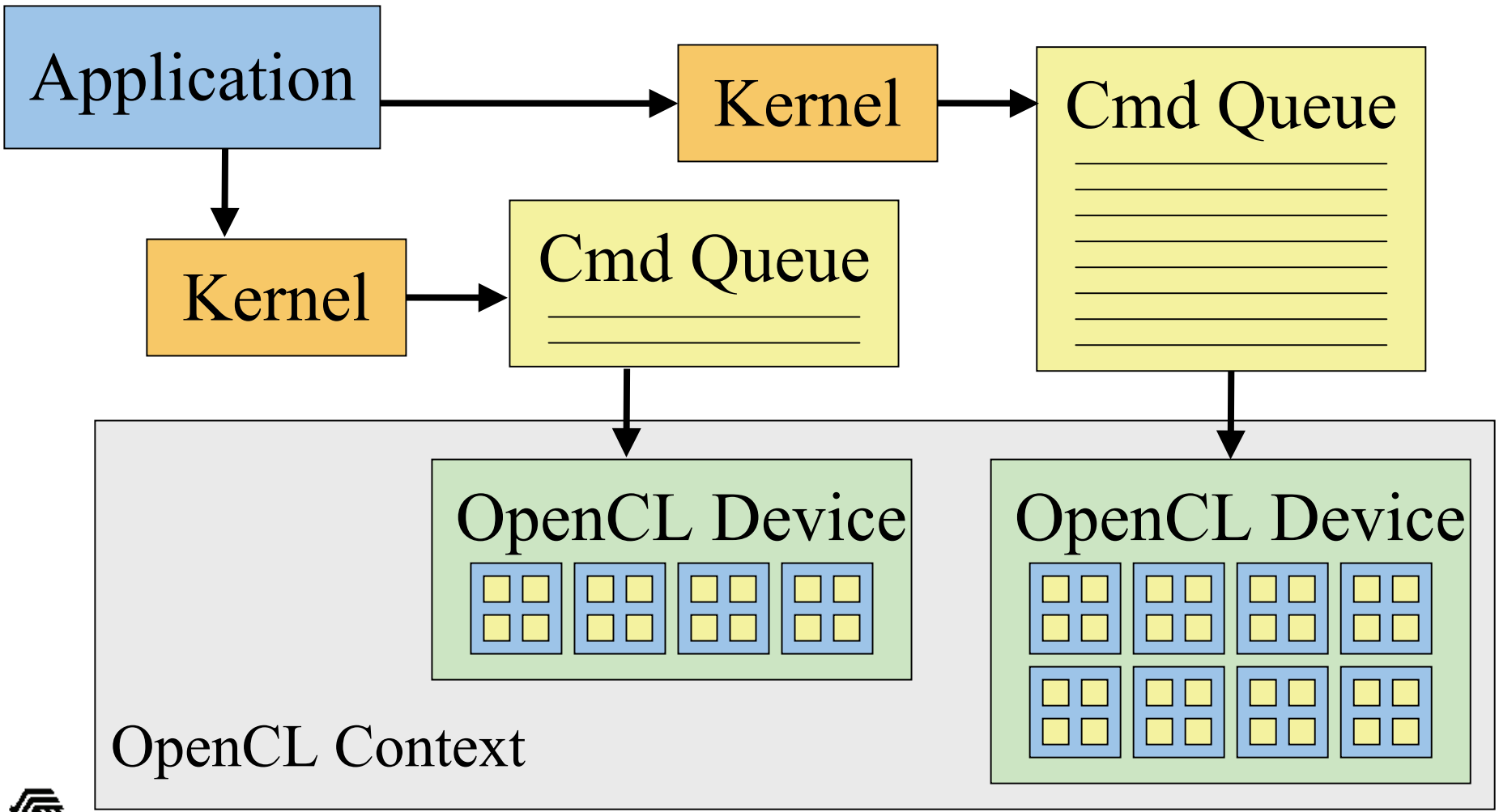


OpenCL Kernels

- Code that actually executes on target devices
- Analogous to CUDA kernels
- Kernel body is instantiated once for each work item
- Each OpenCL work item gets a unique index, like a CUDA thread does

```
__kernel void
vadd(__global const float *a,
     __global const float *b,
     __global float *result) {
  int id = get_global_id(0);
  result[id] = a[id] + b[id];
}
```

OpenCL Execution on Multiple Devices



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…]
```

Gives raw source code string(s) to OpenCL

```
cl_program clpgm;
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, and retreive a handle to the “clenergy” kernel

OpenCL Kernel Launch (abridged)

```
doutput = clCreateBuffer(clctx, CL_MEM_READ_WRITE, volmemsz, NULL, NULL);
datominfo = clCreateBuffer(clctx, CL_MEM_READ_ONLY, MAXATOMS * sizeof(cl_float4),
    NULL, NULL);
[…]
clerr = clSetKernelArg(clkern, 0, sizeof(int), &runatoms);
clerr = clSetKernelArg(clkern, 1, sizeof(float), &zplane);
clerr = clSetKernelArg(clkern, 2, sizeof(cl_mem), &doutput);
clerr = clSetKernelArg(clkern, 3, sizeof(cl_mem), &datominfo);
cl_event event;
clerr = clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL, Gsz, Bsz, 0, NULL, &event);
clerr = clWaitForEvents(1, &event);
clerr = clReleaseEvent(event);
[…]
clEnqueueReadBuffer(clcmdq, doutput, CL_TRUE, 0, volmemsz, energy, 0, NULL, NULL);
clReleaseMemObject(doutput);
clReleaseMemObject(datominfo);
```

Load OpenCL kernel parameters onto stack…

Launch kernel!

Read results back

OpenCL Application Example

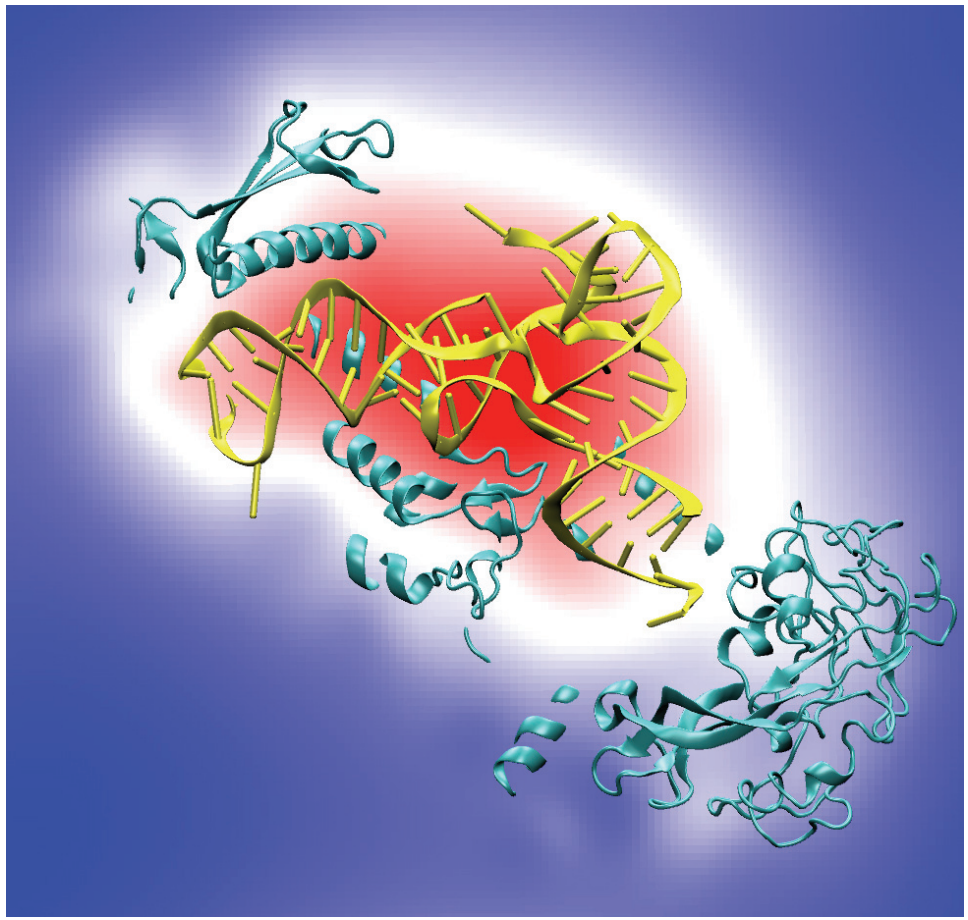
- The easiest way to illustrate how OpenCL works is to explore a very simple algorithm implemented using the OpenCL API
- Since many have been working with CUDA already, I'll use the simple direct Coulomb summation kernel we originally wrote in CUDA
- I'll show how CUDA and OpenCL have much in common, and also highlight some of the new issues one has to deal with in using OpenCL on multiple hardware platforms

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

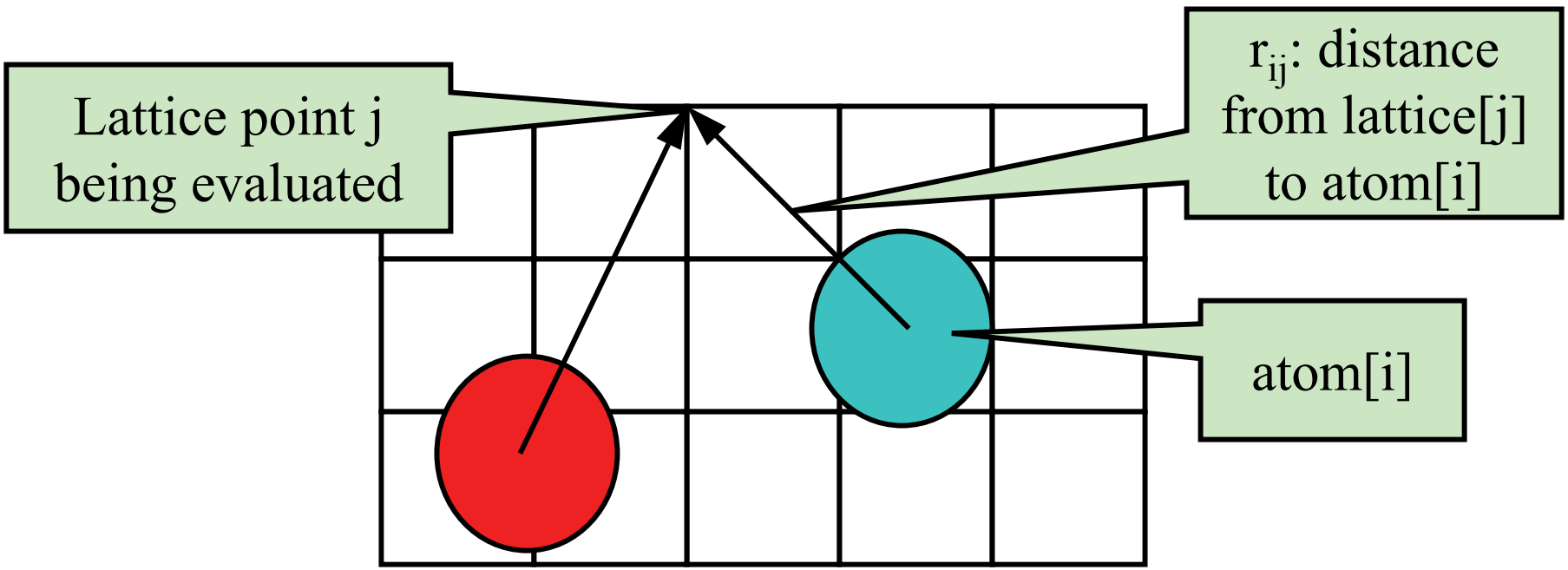


Isoleucine tRNA synthetase

Direct Coulomb Summation

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

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

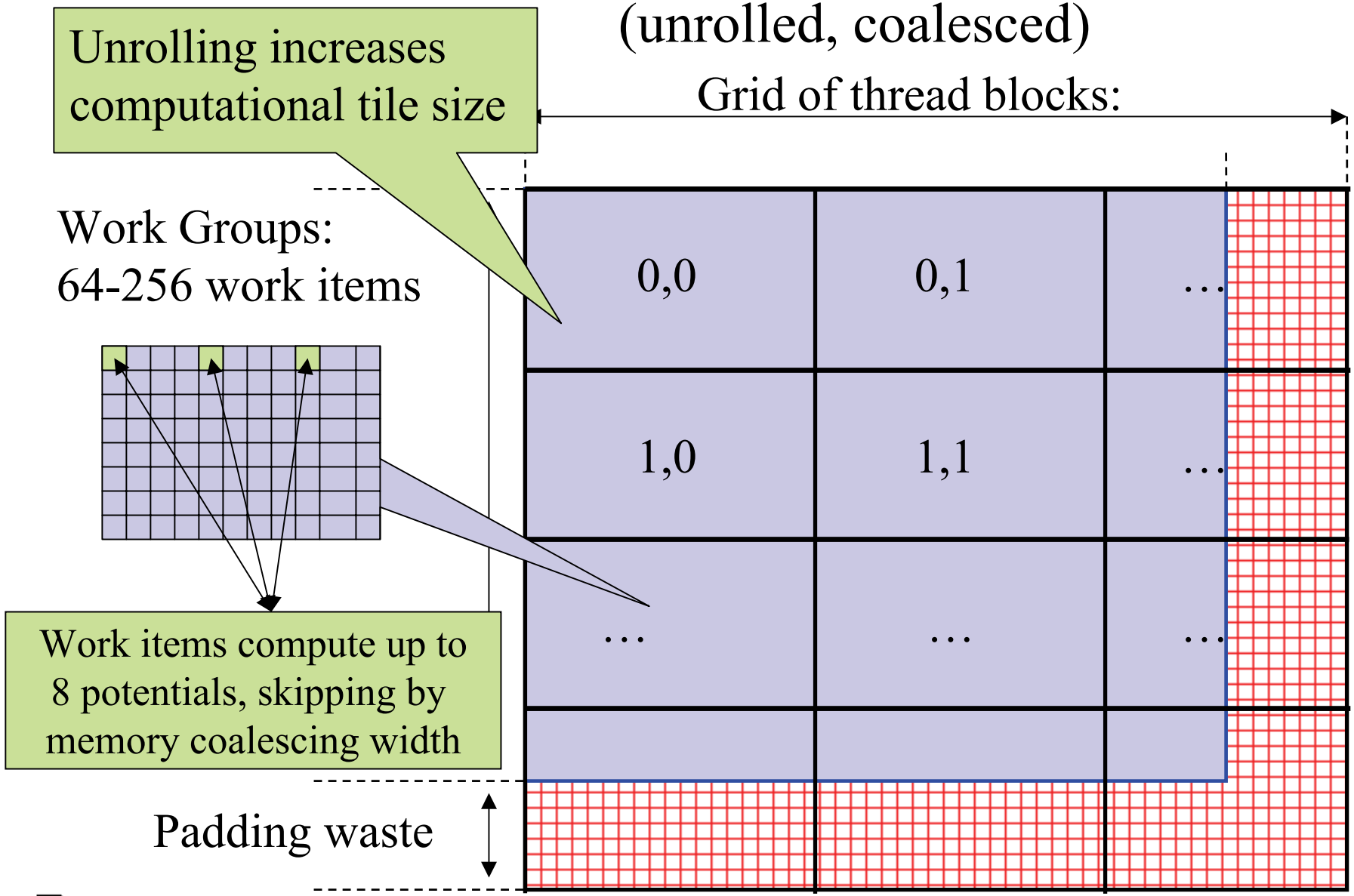


Single Slice DCS: Simple (Slow) C Version

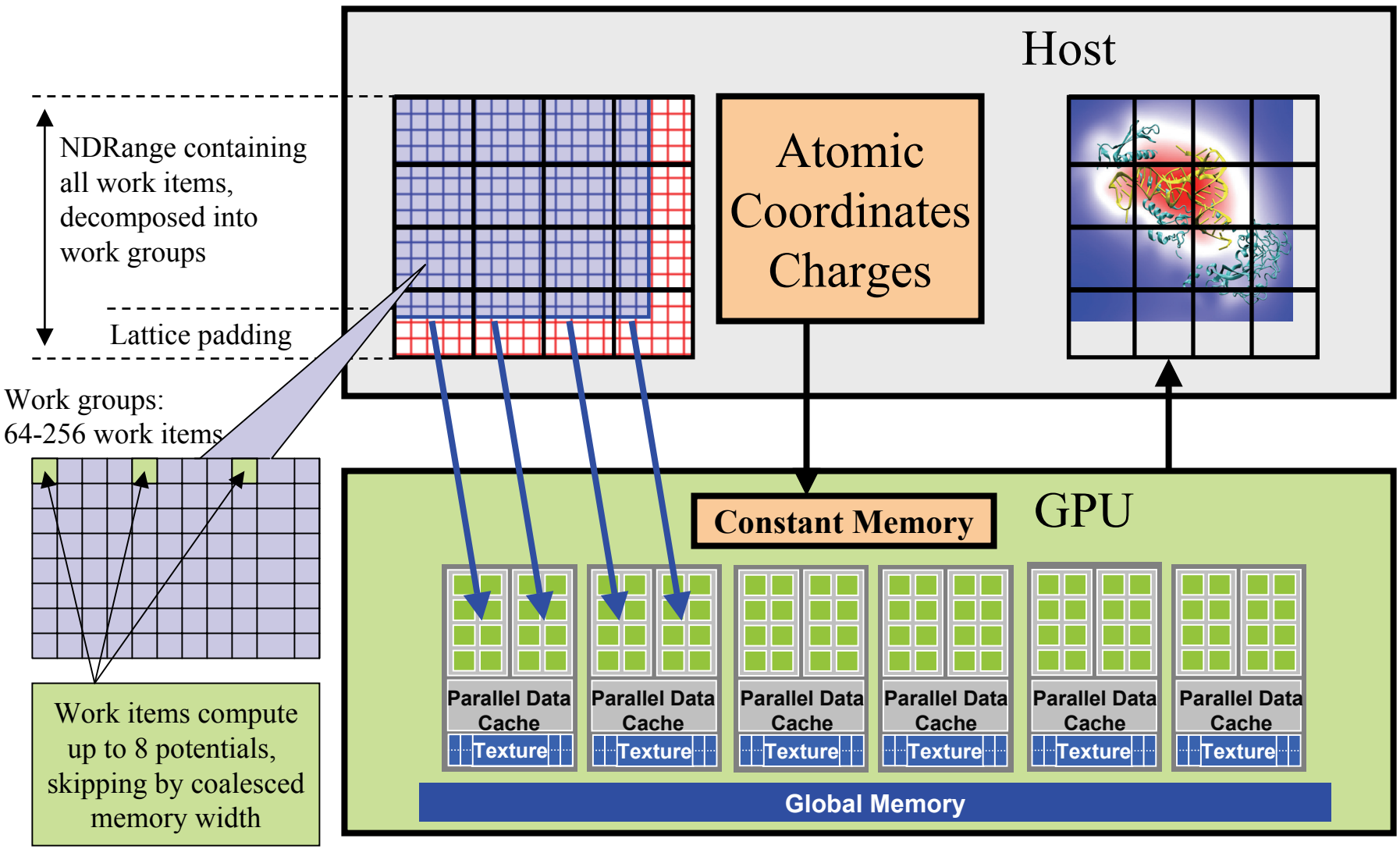
```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms, int numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each atom
        float dx = x - atoms[n    ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
```

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;
```

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 x86 CPU, AMDGPU)

```
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);
}
```

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

- Existing OpenCL implementations don't necessarily autovectorize your code to 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

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

Getting More Performance: Adapting DCS Kernel to OpenCL on Cell

OpenCL Target Device	Scalar Kernel: Ported directly from original CUDA kernel	4-Vector Kernel: Replaced manually unrolled loop iterations with float4 vector ops	Async Copy Kernel: Replaced __constant accesses with use of async_work_group_copy(), use float16 vector ops
IBM QS22 CellBE *** __constant not implemented	2.33 Bevals/sec, 17.16 GFLOPS ****	6.21 Bevals/sec, 45.81 GFLOPS ****	16.22 Bevals/sec, 119.65 GFLOPS

Replacing the use of constant memory with loads of atom data to __local memory via async_work_group_copy() increases performance significantly since Cell hadn't implemented __constant memory yet, at the time of these tests.

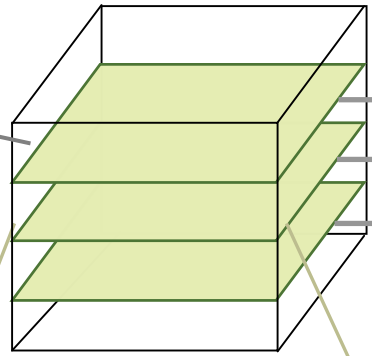
Tests show that the speed of native_rsqrt() is a performance limiter for Cell. Replacing native_rsqrt() with a multiply results in a ~3x increase in execution rate.

Experiments Porting VMD CUDA Kernels to OpenCL

- OpenCL is very similar to CUDA, though a few years behind in terms of HPC features
- Why support OpenCL if CUDA works well?
  - Potential to eliminate hand-coded SSE, or other chip-specific arithmetic intrinsics for CPU versions of compute intensive code
  - Looks more like C and is easier for non-experts to read than hand-coded SSE or other vendor-specific instruction set intrinsics, assembly code, etc
  - Easier to support rare, high-end machines like NCSA Blue Waters, in apps like VMD that are more typically used on commodity hardware

MO GPU Parallel Decomposition

MO 3-D lattice decomposes into 2-D slices. Each slice bcomes a CUDA grid or an OpenCL NDRange



…

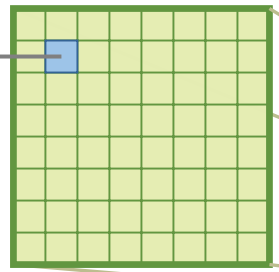
GPU 2

GPU 1

GPU 0

Lattice can be computed using multiple GPUs

Small 8x8 thread blocks afford large per-thread register count, shared memory



Each thread computes one MO lattice point.

Threads producing results that are used

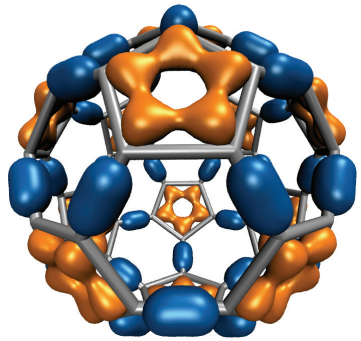
Threads producing results that are discarded

Padding optimizes global memory performance, guaranteeing coalesced global memory accesses

Grid of thread blocks

Molecular Orbital Inner Loop, Hand-Coded SSE Hard to Read, Isn't It? (And this is the "pretty" version!)

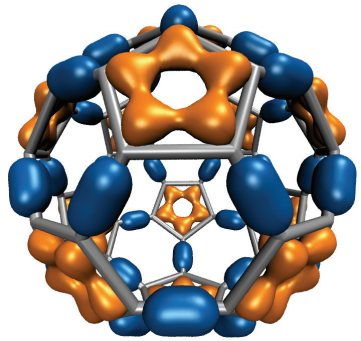
```
for (shell=0; shell < maxshell; shell++) {
  __m128 Cgto = _mm_setzero_ps();
  for (prim=0; prim<num_prim_per_shell[shell_counter]; prim++) {
    float exponent       = -basis_array[prim_counter    ];
    float contract_coeff =  basis_array[prim_counter + 1];
    __m128 expval = _mm_mul_ps(_mm_load_ps1(&exponent), dist2);
    __m128 ctmp = _mm_mul_ps(_mm_load_ps1(&contract_coeff), exp_ps(expval));
    Cgto = _mm_add_ps(contracted_gto, ctmp);
    prim_counter += 2;
  }
  __m128 tshell = _mm_setzero_ps();
  switch (shell_types[shell_counter]) {
    case S_SHELL:
      value = _mm_add_ps(value, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), Cgto));    break;
    case P_SHELL:
      tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), xdist));
      tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), ydist));
      tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), zdist));
      value = _mm_add_ps(value, _mm_mul_ps(tshell, Cgto));
      break;
```



Until now, writing SSE kernels for CPUs required assembly language, compiler intrinsics, various libraries, or a really smart autovectorizing compiler and lots of luck...

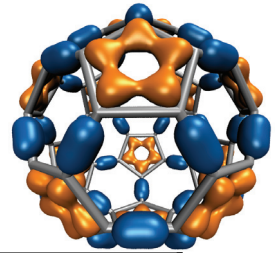
Molecular Orbital Inner Loop, OpenCL Vec4 Ahhh, much easier to read!!!

```
for (shell=0; shell < maxshell; shell++) {
  float4 contracted_gto = 0.0f;
  for (prim=0; prim < const_num_prim_per_shell[shell_counter];  prim++) {
    float exponent       = const_basis_array[prim_counter    ];
    float contract_coeff = const_basis_array[prim_counter + 1];
    contracted_gto += contract_coeff * native_exp2(-exponent*dist2);
    prim_counter += 2;
  }
  float4 tmpshell=0.0f;
  switch (const_shell_symmetry[shell_counter]) {
    case S_SHELL:
      value += const_wave_f[ifunc++] * contracted_gto;      break;
    case P_SHELL:
      tmpshell += const_wave_f[ifunc++] * xdist;
      tmpshell += const_wave_f[ifunc++] * ydist;
      tmpshell += const_wave_f[ifunc++] * zdist;
      value += tmpshell * contracted_gto;
      break;
```



OpenCL's C-like kernel language is easy to read, even 4-way vectorized kernels can look similar to scalar CPU code. All 4-way vectors shown in green.

Apples to Oranges Performance Results: OpenCL Molecular Orbital Kernels



Kernel	Cores	Runtime (s)	Speedup
Intel QX6700 CPU ICC-SSE (SSE intrinsics)	1	46.580	1.00
Intel Core2 Duo CPU OpenCL scalar	2	43.342	1.07
Intel Core2 Duo CPU OpenCL vec4	2	8.499	5.36
Cell OpenCL vec4*** no __constant	16	6.075	7.67
Radeon 4870 OpenCL scalar	10	2.108	22.1
Radeon 4870 OpenCL vec4	10	1.016	45.8
GeForce GTX 285 OpenCL vec4	30	0.364	127.9
GeForce GTX 285 CUDA 2.1 scalar	30	0.361	129.0
GeForce GTX 285 OpenCL scalar	30	0.335	139.0
GeForce GTX 285 CUDA 2.0 scalar	30	0.327	142.4

Minor varations in compiler quality can have a strong effect on "tight" kernels. The two results shown for CUDA demonstrate performance variability with compiler revisions, and that with vendor effort, OpenCL has the potential to match the performance of other APIs.

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)

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
  - Ben Levine, Axel Kohlmeyer, Temple University
  - Prof. Wen-mei Hwu, Christopher Rodrigues, UIUC IMPACT Group
  - UIUC NVIDIA CUDA Center of Excellence
  - CUDA team at NVIDIA
  - NIH support: P41-RR05969

Online OpenCL Materials

- 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/
- NVIDIA OpenCL Resources:
  http://www.nvidia.com/object/cuda_opencl.html

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.