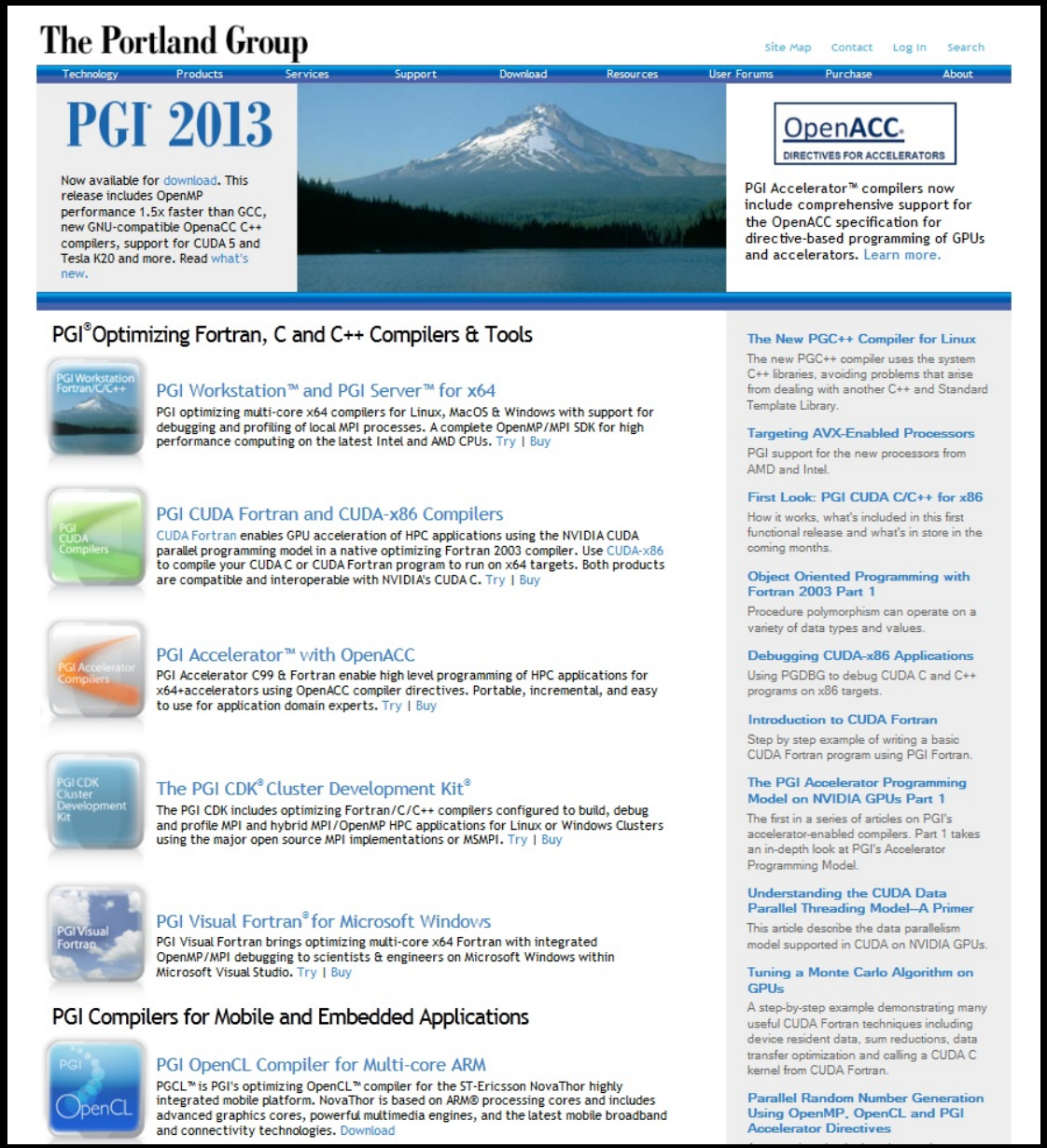
Directive-based Accelerator Programming With OpenACC

Mathew Colgrove
mathew.colgrove@pgroup.com

July 2013

PGI

- C99, C++, F2003 Compilers
  - Optimizing
  - Vectorizing
  - Parallelizing
- Graphical parallel tools
  - PGDBG® debugger
  - PGPROF® profiler
- AMD, Intel, NVIDIA
- PGI Unified Binary™
- Linux, OS X, Windows
- Visual Studio & Eclipse integration
- GPGPU Features
  - PGI Accelerator™, OpenACC
  - CUDA Fortran/C/C++
  - CUDA-x86
  - OpenCL (ARM CPUs only)



www.pgroup.com

Talk Roadmap

- Introduction to OpenACC
- Compute Constructs
- Loop Directives
- Demo – Himeno
- Data Region
- Porting Example – Seismic CPML
- Future OpenACC features

OpenACC

Open Programming Standard for Parallel Computing

"PGI OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan."

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

"OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP."

--Michael Wong, CEO OpenMP Directives Board

OpenACC Standard

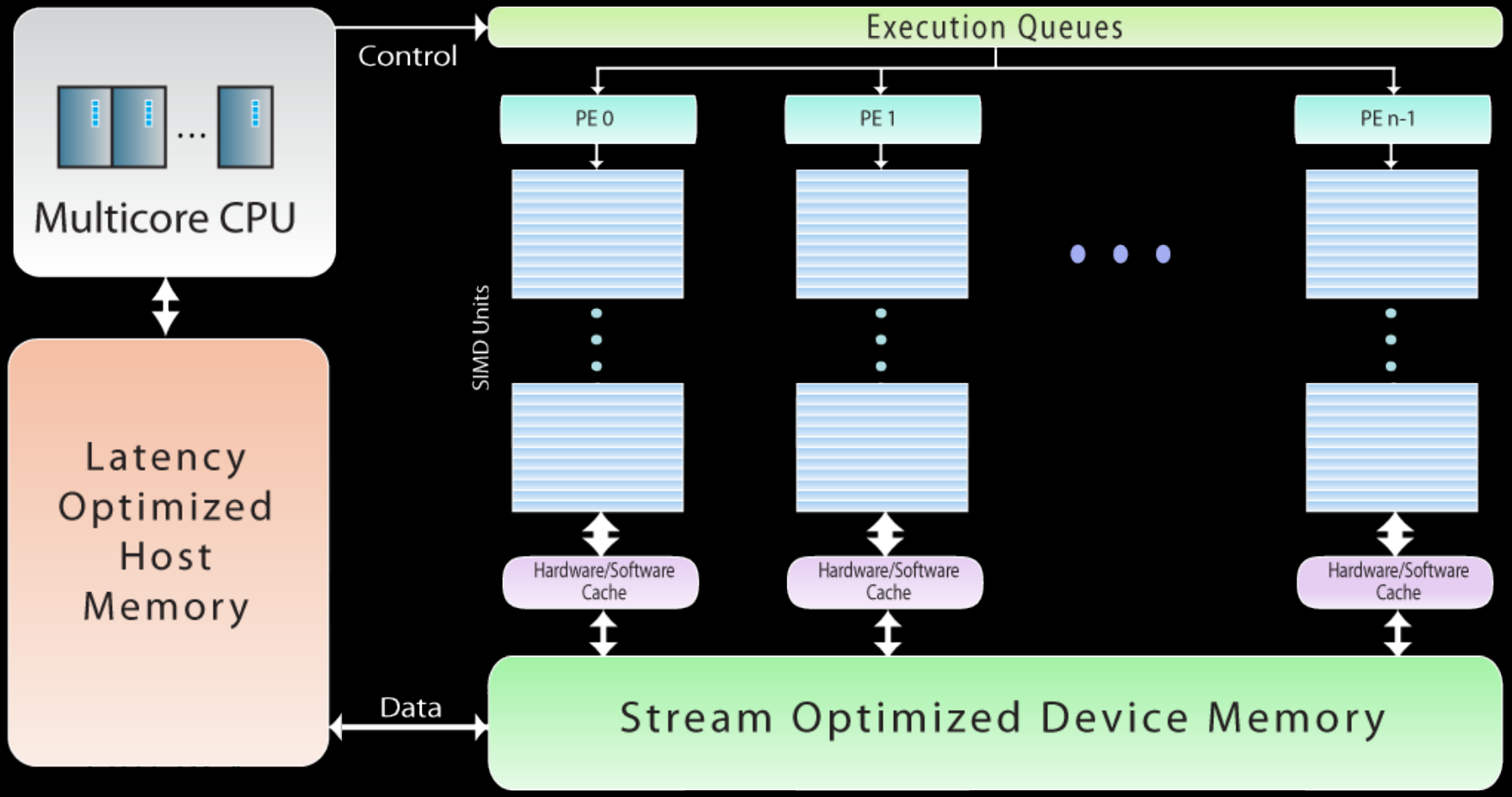
NVIDIA

CRAY THE SUPERCOMPUTER COMPANY

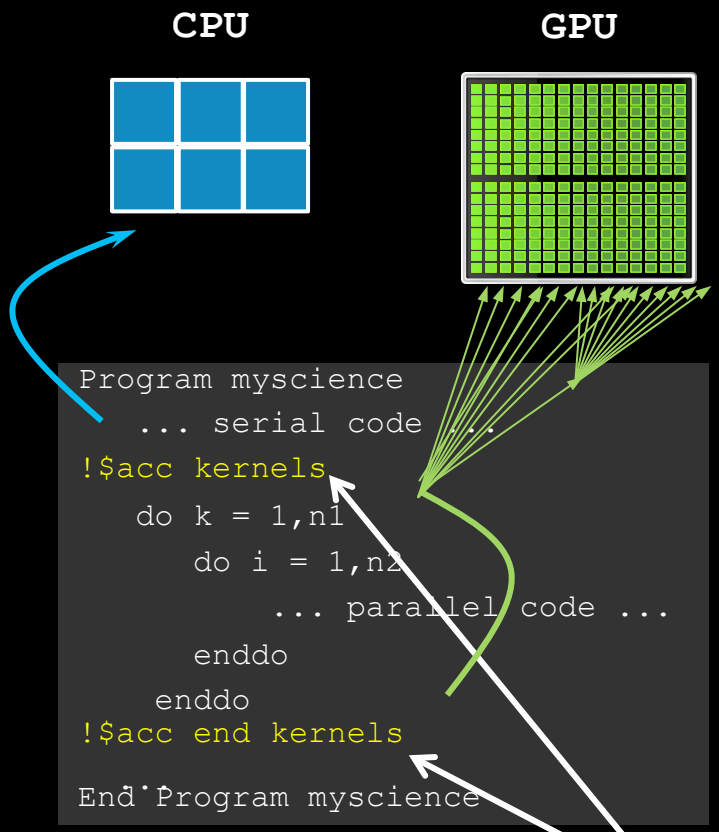
PGI

CAPS

OpenACC Abstract Machine Architecture



OpenACC Directives



Portable compiler hints

Compiler parallelizes code

Designed for multicore CPUs & many core GPUs / Accelerators

PGI Accelerator Directive-based Compilers

```
#pragma acc kernels loop
  for( i = 0; i < nrows; ++i ){
    float val = 0.0f;
    for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
        val += m[i+nrows*d] * v[j];
    }
    x[i] = val;
  }
```

compile

```
matvec:
        subq  $328, %rsp
...
        call  __pgi_cu_alloc
...
        call  __pgi_cu_uploadx
...
        call  __pgi_cu_launch2
...
        call  __pgi_cu_downloadx
...
        call  __pgi_cu_free
...
```

+

```
    .entry  matvec_14_gpu( ...
    .reg  .u32 %r<70> ...
    cvt.s32.u32  %r1, %tid.x;
    mov.s32      %r2, 0;
    setp.ne.s32  $p1, %r1, %r2
    cvt.s32.u32  %r3, %ctaid.x;
    cvt.s32.u32  %r4, %ntid.x;
    mul.lo.s32   %r5, %r3, %r4;
    @%p1 bra     $Lt_0_258;
    st.shared.s32 [__i2s], %r5
$Lt_0_258:
    bar.sync     0;
...
```

Link

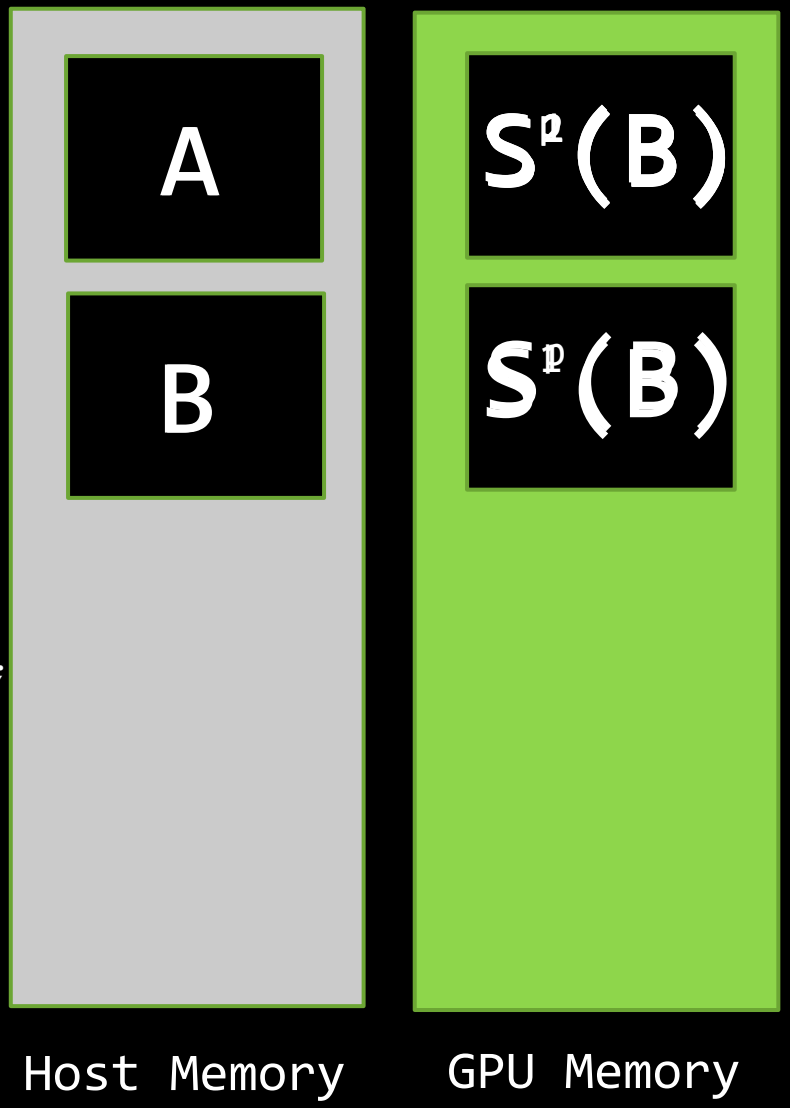
Unified Object

execute

… no change to existing makefiles, scripts, IDEs, programming environment, etc.

PGI Accelerator OpenACC example

```
#pragma acc data \
        copy(b[0:n][0:m]) \
        create(a[0:n][0:m])
{
for (iter = 1; iter <= p; ++iter){
  #pragma acc kernels
  {
  for (i = 1; i < n-1; ++i){
    for (j = 1; j < m-1; ++j){
      a[i][j]=w0*b[i][j]+
              w1*(b[i-1][j]+b[i+1][j]+
                  b[i][j-1]+b[i][j+1])+
              w2*(b[i-1][j-1]+b[i-1][j+1]+
                  b[i+1][j-1]+b[i+1][j+1]);
  } }
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      b[i][j] = a[i][j];
  }
}
}
```



Why use OpenACC Directives?

- Productivity
  - Higher level programming model
  - a la OpenMP
- Portability
  - ignore directives, portable to the host
  - portable across different accelerators
  - *performance portability*
- Performance feedback

Matrix Multiply Source Code Size Comparison:

```
1 void
2 computeMM0_saxpy(float C[][WB], float A[][WA], float B[][WB],
3                  int hA, int wA, int wB)
4 {
5 #pragma acc region
6 {
7 #pragma acc for parallel vector(16) unroll(4)
8     for (int i = 0; i < hA; ++i) {
9         for (int j = 0; j < wB; ++j) {
10            C[i][j] = 0.0 ;
11        }
12        for (int k = 0; k < wA; ++k) {
13            for (int j = 0; j < wB; ++j) {
14                C[i][j] = C[i][j]+A[i][k]*B[k][j];
15            }
16        }
17    }
18 }
19 }
}
```

Directives

```
1 void
2 __global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
3 {
4     int bx = blockIdx.x;
5     int by = blockIdx.y;
6     int tx = threadIdx.x;
7     int ty = threadIdx.y;
8     int aBegin = wA * BLOCK_SIZE * by;
9     int aEnd   = aBegin + wA - 1;
10    int aStep  = BLOCK_SIZE;
11    int bBegin = BLOCK_SIZE * bx;
12    int bStep  = BLOCK_SIZE * wB;
13    float Csub = 0;
14    for (int a = aBegin, b = bBegin;
15             a <= aEnd;
16             a += aStep, b += bStep) {
17        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
18        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
19        AS(ty, tx) = A[a + wA * ty + tx];
20        BS(ty, tx) = B[b + wB * ty + tx];
21        __syncthreads();
22        for (int k = 0; k < BLOCK_SIZE; ++k)
23            Csub += AS(ty, k) * BS(k, tx);
24        __syncthreads();
25    }
26    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
27    C[c + wB * ty + tx] = Csub;
28 }
29
30 void
31 domatmul( float* C, float* A, float* B, unsigned int hA, unsigned int wA , unsigned wB )
32 {
33     unsigned int size_A = WA * HA;
34     unsigned int mem_size_A = sizeof(float) * size_A;
35     unsigned int size_B = WB * HB;
36     unsigned int mem_size_B = sizeof(float) * size_B;
37     unsigned int size_C = WC * HC;
38     unsigned int mem_size_C = sizeof(float) * size_C;
39     float *d_A, *d_B, *d_C;
40
41     cudaMalloc((void**) &d_A, mem_size_A);
42     cudaMalloc((void**) &d_B, mem_size_B);
43     cudaMalloc((void**) &d_C, mem_size_C);
44     cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
45     cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
46
47     dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
48     dim3 grid(WC / threads.x, HC / threads.y);
49
50     matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
51
52     cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
53
54     cudaFree(d_A);
55     cudaFree(d_B);
56     cudaFree(d_C);
57 }
```

CUDA C

```
1 void matrixMulGPU(cl_uint ciDeviceCount, cl_mem h_A, float* h_B_data,
2                   unsigned int mem_size_B, float* h_C )
3 {
4     cl_mem d_A[MAX_GPU_COUNT];
5     cl_mem d_C[MAX_GPU_COUNT];
6     cl_mem d_B[MAX_GPU_COUNT];
7
8     cl_event GPUDone[MAX_GPU_COUNT];
9     cl_event GPUExecution[MAX_GPU_COUNT];
10
11    // Create buffers for each GPU
12    // Each GPU will compute sizePerGPU rows of the result
13    int sizePerGPU = HA / ciDeviceCount;
14
15    int workOffset[MAX_GPU_COUNT];
16    int workSize[MAX_GPU_COUNT];
17
18    workOffset[0] = 0;
19    for(unsigned int i=0; i < ciDeviceCount; ++i)
20    {
21        // Input buffer
22        workSize[i] = (i != (ciDeviceCount - 1)) ? sizePerGPU : (HA - workOffset[i]);
23
24        d_A[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, workSize[i] * sizeof(float) * WA, NULL,NULL);
25
26        // Copy only assigned rows from host to device
27        clEnqueueCopyBuffer(commandQueue[i], h_A, d_A[i], workOffset[i] * sizeof(float) * WA,
28                            0, workSize[i] * sizeof(float) * WA, 0, NULL, NULL);
29
30        // create OpenCL buffer on device that will be initiatlize from the host memory on first use
31        // on device
32        d_B[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
33                                mem_size_B, h_B_data, NULL);
34
35        // Output buffer
36        d_C[i] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,  workSize[i] * WC * sizeof(float), NULL,NULL);
37
38        // set the args values
39        clSetKernelArg(multiplicationKernel[i], 0, sizeof(cl_mem), (void *) &d_C[i]);
40        clSetKernelArg(multiplicationKernel[i], 1, sizeof(cl_mem), (void *) &d_A[i]);
41        clSetKernelArg(multiplicationKernel[i], 2, sizeof(cl_mem), (void *) &d_B[i]);
42        clSetKernelArg(multiplicationKernel[i], 3, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
43        clSetKernelArg(multiplicationKernel[i], 4, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
44
45        if(i+1 < ciDeviceCount)
46            workOffset[i + 1] = workOffset[i] + workSize[i];
47    }
48    // Execute Multiplication on all GPUs in parallel
49    size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
50    size_t globalWorkSize[] = {shrRoundUp(BLOCK_SIZE, WC), shrRoundUp(BLOCK_SIZE, workSize[0])};
51    // Launch kernels on devices
52        for(unsigned int i = 0; i < ciDeviceCount; i++)
53        {
54                        // Multiplication - non-blocking execution
55                        globalWorkSize[1] = shrRoundUp(BLOCK_SIZE, workSize[i]);
56                        clEnqueueNDRangeKernel(commandQueue[i], multiplicationKernel[i], 2, 0, globalWorkSize, localWorkSize,
57                                               0, NULL, &GPUExecution[i]);
58                }
59        for(unsigned int i = 0; i < ciDeviceCount; i++)
60    {
61                clFinish(commandQueue[i]);
62        }
63    for(unsigned int i = 0; i < ciDeviceCount; i++)
64    {
65 // Non-blocking copy of result from device to host
66        clEnqueueReadBuffer(commandQueue[i], d_C[i], CL_FALSE, 0, WC * sizeof(float) * workSize[i],
67                            h_C + workOffset[i] * WC, 0, NULL, &GPUDone[i]);
68    }
69 // CPU sync with GPU
70    clWaitForEvents(ciDeviceCount, GPUDone);
71
72 // Release mem and event objects
73    for(unsigned int i = 0; i < ciDeviceCount; i++)
74    {
75        clReleaseMemObject(d_A[i]);
76        clReleaseMemObject(d_C[i]);
77        clReleaseMemObject(d_B[i]);
78        clReleaseEvent(GPUExecution[i]);
79        clReleaseEvent(GPUDone[i]);
80    }
81 }
82 __kernel void
83 matrixMul( __global float* C, __global float* A, __global float* B,
84            __local float* As, __local float* Bs)
85 {
86    int bx = get_group_id(0), tx = get_local_id(0);
87    int by = get_group_id(1), ty = get_local_id(1);
88    int aEnd   = WA * BLOCK_SIZE * by + WA - 1;
89
90    float Csub = 0.0f;
91
92    for (int a = WA*BLOCK_SIZE*by , b = BLOCK_SIZE * bx;
93         a <= aEnd; a += BLOCK_SIZE, b += BLOCK_SIZE*WB) {
94        As[tx + ty * BLOCK_SIZE] = A[a + WA * ty + tx];
95        Bs[tx + ty * BLOCK_SIZE] = B[b + WB * ty + tx];
96        barrier(CLK_LOCAL_MEM_FENCE);
97        for (int k = 0; k < BLOCK_SIZE; ++k)
98            Csub += As[k + ty * BLOCK_SIZE]*Bs[tx + k * BLOCK_SIZE] ;
99        barrier(CLK_LOCAL_MEM_FENCE);
100   }
101   C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = Csub;
102
103 }
```

OpenCL

Talk Roadmap

- Introduction to OpenACC
- Compute Constructs
- Loop Directives
- Demo – Himeno
- Data Region
- Porting Example – Seismic CPML
- Future OpenACC features

Kernels Construct

- C

```
#pragma acc kernels clause...
{
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

```
!$acc kernels clause...
     do i = 1,n
          r(i) = a(i) * 2.0
     enddo
!$acc end kernels
```

Parallel Construct

- C

```
#pragma acc parallel clause...
{
    #pragma acc loop gang vector
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

```
!$acc parallel clause...
!$acc loop gang vector
    do i = 1,n
       r(i) = a(i) * 2.0
    enddo
!$acc end parallel
```

Kernels vs. Parallel

- Kernels Construct
  - Derived from PGI Accelerator Model
  - More implicit giving the compiler more freedom to create optimal code for a given accelerator
  - Works best for tightly nested loops
  - May require some additional 'hints' to the compiler
    - i.e. C99 restrict keyword
- Parallel Construct
  - Based on OpenMP "workshare"
  - Create parallel gangs that execute redundantly
  - Each gang executes a portion of a work-sharing loop
  - More explicit requiring some user intervention

http://www.pgroup.com/lit/articles/insider/v4n2a1.htm

Talk Roadmap

- Introduction to OpenACC
- Compute Constructs
- Loop Directives
- Demo – Himeno
- Data Region
- Porting Example – Seismic CPML
- Future OpenACC features

Loop Directive

- C

```
#pragma acc loop clause...
for( i = 0; i < n; ++i ){
    ....
}
```

- Fortran

```
!$acc loop clause...
do i = 1, n
```

Note: Compute Constructs and the Loop directive may be combined

Loop Directive Clauses

- `independent`
  - use with care, overrides compiler analysis for dependence, private variables (kernels only, implied with parallel)
- `private( list )`
  - private data for each iteration of the loop
- `reduction( red:var )`
  - reduction across the loop

- Scheduling Clauses
  - `vector` or `vector(width)`
  - `gang` or `gang(width)`
  - `worker` or `worker(width)`
  - `seq`

Loop Scheduling Clauses

- `!$acc loop gang`
  - runs in 'gang' mode only (`blockIdx`)
  - does not declare that the loop is in fact parallel (use independent)
- `!$acc loop gang(32)`
  - runs in 'parallel' mode only with `gridDim` == 32 (32 blocks)
- `!$acc loop vector(128)`
  - runs in 'vector' mode (`threadIdx`) with `blockDim` == 128 (128 threads)
  - vector size, if present, must be compile-time constant
- `!$acc loop gang vector(128)`
  - strip mines loop
  - inner loop runs in vector mode, 128 threads (`threadIdx`)
  - outer loop runs across thread blocks (`blockIdx`)

Time for a Demo

- Himeno
  - Compute constructs
  - Loop schedules
  - Compiler Feed-back messages
  - Profiling

Talk Roadmap

- Introduction to OpenACC
- Compute Constructs
- Loop Directives
- Demo – Himeno
- Data Region
- Porting Example – Seismic CPML
- Future OpenACC features

Data Region

- C

```
#pragma acc data
{
    ....
}
```

- Fortran

```
!$acc data
    ....
!$acc end data
```

- May span across host code and multiple compute regions
- May be nested
- May not be nested within a compute region
- Data is not implicitly synchronized between the host and device

Data Clauses

- Data clauses
  - `copy( list )`
  - `copyin( list )`
  - `copyout( list )`
  - `create( list )`
  - `present( list )`
  - `present_or_copy(list)` `pcopy(list)`
  - `present_or_copyin(list)` `pcopyin(list)`
  - `present_or_copyout(list)` `pcopyout(list)`
  - `present_or_create(list)` `pcreate(list)`
  - `deviceptr( list )`

Data Attributes

- predetermined data attributes
  - loop variables are private
- implicit data attributes
  - array, struct – `present_or_copy`
  - scalar – `firstprivate`
- explicit data attributes
  - in a data clause

Data Regions Across Procedures

```
subroutine sub( a, b )
 real :: a(:), b(:)
  !$acc kernels pcopyin(b)
    do i = 1,n
      a(i) = a(i) * b(i)
    enddo
  !$acc end kernels
  ...
end subroutine

subroutine bus(x, y)
 real :: x(:), y(:)
 !$acc data copy(x)
 call sub( x, y )
 !$acc end data
```

Data Regions Across Procedures

```
void sub( float* a, float* b, int n ){
  int i;
  #pragma acc kernels pcopyin(b[0:n])
    for( i = 0; i < n; ++i )
      a[i] *= b[i];
  ...
}

void bus( float* x, float* y, int n ){
  #pragma acc data copy(x[0:n])
  {
    sub( x, y, n );
  }
...
```

Update Directives

- `update host( list )`
- `update device( list )`
  - data must be in a data allocate clause for an enclosing data region
  - both may be on a single line
    - `update host(list) device(list)`
- Data update clauses on data construct (PGI)
  - `updatein( list )` or `update device( list )`
  - `updateout( list )` or `update host ( list )`
  - shorthand for update directive just inside data construct

Additional Concepts not covered today

- Asynchronous Data Movement
- Asynchronous Compute
- Wait Directive
- Cache Directive
- Host_data Directive
- OpenACC and CUDA interoperability
- OpenACC Runtime Library Calls
- OpenACC Environment Variables
- Multi-device programming
- Multi-target accelerators (Unified Binary)
- Performance Optimization
- Data Layout

Talk Roadmap

- Introduction to OpenACC
- Compute Constructs
- Loop Directives
- Demo – Himeno
- Data Region
- Porting Example – Seismic CPML
- Future OpenACC features

Porting SEISMIC CPML

- Set of ten open-source Fortran90 programs
- Solves two-dimensional or three-dimensional isotropic or anisotropic elastic, viscoelastic or poroelastic wave equation.
- Uses finite-difference method with Convolutional or Auxiliary Perfectly Matched Layer (C-PML or ADE-PML) conditions
- Developed by Dimitri Komatitsch and Roland Martin from University of Pau, France.
- Accelerated source used is taken from the 3D elastic finite-difference code in velocity and stress formulation with Convolutional-PML (C-PML) absorbing conditions.

http://www.geodynamics.org/cig/software/seismic_cpml

Full Article: http://www.pgroup.com/lit/articles/insider/v4n1a3.htm

Step 1: Evaluation

- Is my algorithm right for a GPU?
  - SEISMIC_CPML models seismic waves through the earth. Has an outer time step loop with 9 inner parallel loops. Uses MPI and OpenMP parallelization.
- Good candidate for the GPU, but not ideal.

Step 2: Add Compute Regions

```
!$acc kernels
  do k = kmin,kmax
    do j = NPOINTS_PML+1, NY-NPOINTS_PML
      do i = NPOINTS_PML+1, NX-NPOINTS_PML
      total_energy_kinetic = total_energy_kinetic + 0.5d0 * rho*(vx(i,j,k)**2 + vy(i,j,k)**2 + vz(i,j,k)**2)
      epsilon_xx = ((lambda + 2.d0*mu) * sigmaxx(i,j,k) - lambda * sigmayy(i,j,k) - lambda*sigmazz(i,j,k)) / (4.d0 * mu * (lambda + mu))
      epsilon_yy = ((lambda + 2.d0*mu) * sigmayy(i,j,k) - lambda * sigmaxx(i,j,k) - lambda*sigmazz(i,j,k)) / (4.d0 * mu * (lambda + mu))
      epsilon_zz = ((lambda + 2.d0*mu) * sigmazz(i,j,k) - lambda * sigmaxx(i,j,k) - lambda*sigmayy(i,j,k)) / (4.d0 * mu * (lambda + mu))
      epsilon_xy = sigmaxy(i,j,k) / (2.d0 * mu)
      epsilon_xz = sigmaxz(i,j,k) / (2.d0 * mu)
      epsilon_yz = sigmayz(i,j,k) / (2.d0 * mu)
      total_energy_potential = total_energy_potential + 0.5d0 * (epsilon_xx * sigmaxx(i,j,k) + epsilon_yy * sigmayy(i,j,k) + &
        epsilon_yy * sigmayy(i,j,k)+ 2.d0 * epsilon_xy * sigmaxy(i,j,k) + 2.d0*epsilon_xz * sigmaxz(i,j,k)+2.d0*epsilon_yz * sigmayz(i,j,k))
      enddo
    enddo
  enddo
!$acc end kernels
```

Compiler Feedback

```
% pgfortran -Mmpi=mpich2 -fast -acc -Minfo=accel
seismic_CPML_3D_isotropic_MPI_OACC_1.F90 -o gpu1.out
seismic_cpml_3d_iso_mpi_openmp:
   1107, Generating copyin(vz(11:91,11:631,kmin:kmax))
         Generating copyin(vy(11:91,11:631,kmin:kmax))
         Generating copyin(vx(11:91,11:631,kmin:kmax))
         Generating copyin(sigmaxx(11:91,11:631,kmin:kmax))
         Generating copyin(sigmayy(11:91,11:631,kmin:kmax))
         Generating copyin(sigmazz(11:91,11:631,kmin:kmax))
         Generating copyin(sigmaxy(11:91,11:631,kmin:kmax))
         Generating copyin(sigmaxz(11:91,11:631,kmin:kmax))
         Generating copyin(sigmayz(11:91,11:631,kmin:kmax))
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
```

```
1108, Loop is parallelizable
1109, Loop is parallelizable
1110, Loop is parallelizable
      Accelerator kernel generated
    1108, !$acc do gang, vector(4)
    1109, !$acc do gang, vector(4)
    1110, !$acc do vector(16)
1116, Sum reduction generated for total_energy_kinetic
1134, Sum reduction generated for total_energy_potential
```

Initial Timings

Version	MPI Processes	OpenMP Threads	GPUs	Time (sec)	Approx. Programming Time (min)
Original MPI/OMP	2	4	0	951	
ACC Step 1	2	0	2	3031	10

System Info:
4 Core Intel Core-i7 920 Running at 2.67Ghz
Includes 2 Tesla C2070 GPUs
Problem Size: 101x641x128

Why the slowdown?

Step 3: Optimize Data Movement

```
!$acc data                                                              &
!$acc      copyin(a_x_half,b_x_half,k_x_half,                           &
!$acc                  a_y_half,b_y_half,k_y_half,                      &
!$acc                  a_z_half,b_z_half,k_z_half,                      &
!$acc                  a_x,a_y,a_z,b_x,b_y,b_z,k_x,k_y,k_z,             &
!$acc                  sigmaxx,sigmaxz,sigmaxy,sigmayy,sigmayz,sigmazz, &
!$acc                  memory_dvx_dx,memory_dvy_dx,memory_dvz_dx,       &
!$acc                  memory_dvx_dy,memory_dvy_dy,memory_dvz_dy,       &
!$acc                  memory_dvx_dz,memory_dvy_dz,memory_dvz_dz,       &
!$acc                  memory_dsigmaxx_dx, memory_dsigmaxy_dy,          &
!$acc                  memory_dsigmaxz_dz, memory_dsigmaxy_dx,          &
!$acc                  memory_dsigmaxz_dx, memory_dsigmayz_dy,          &
!$acc                  memory_dsigmayy_dy, memory_dsigmayz_dz,          &
!$acc                  memory_dsigmazz_dz)

  do it = 1,NSTEP
.... Cut ....
  enddo
!$acc end data
```

Timings Continued

Version	MPI Processes	OpenMP Threads	GPUs	Time (sec)	Approx. Programming Time (min)
Original MPI/OMP	2	4	0	951	
ACC Step 1	2	0	2	3031	10
ACC Step 2	2	0	2	124	180

System Info:
4 Core Intel Core-i7 920 Running at 2.67Ghz
Includes 2 Tesla C2070 GPUs

Data movement time now only 5 seconds!

Step 4: Fine Tune Schedule

```
!$acc do vector(4)
  do k=k2begin,NZ_LOCAL
   kglobal = k + offset_k
!$acc do gang, vector(4)
   do j=2,NY
!$acc do gang, vector(16)
     do i=1,NX-1
      value_dvx_dx = (vx(i+1,j,k)-vx(i,j,k)) * ONE_OVER_DELTAX
      value_dvy_dy = (vy(i,j,k)-vy(i,j-1,k)) * ONE_OVER_DELTAY
      value_dvz_dz = (vz(i,j,k)-vz(i,j,k-1)) * ONE_OVER_DELTAZ
```

Final Timings

Version	MPI Processes	OpenMP Threads	GPUs	Time (sec)	Approx. Programming Time (min)
Original MPI/OMP	2	4	0	951	
ACC Step 1	2	0	2	3031	10
ACC Step 2	2	0	2	124	180
ACC Step 3	2	0	2	120	120

7x in 5 Hours!

Cluster Timings

Version	Size	MPI Processes	OpenMP Threads	GPUs	Time (sec)
MPI/OMP	101x641x3072	24	96	0	2081
MPI/ACC	101x641x3072	24	0	24	446

System Info: 8 Nodes
2 socket, 6 Core Intel X5675 Running at 3.06Ghz with 3 Tesla C2070 GPU
Compiler: PGI 2012 version 12.5

Just under 5x!

Talk Roadmap

- Introduction to OpenACC
- Compute Constructs
- Loop Directives
- Demo – Himeno
- Data Region
- Porting Example – Seismic CPML
- Future OpenACC features

OpenACC 2.0 Highlights

- Procedure calls, separate compilation
- Nested parallelism
- Data management features and global data
- atomic operations

Currently Procedure Calls in Compute Regions Must Be Ininled

```
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // must inline matvec
}
```

```
pgcc -Minline a.c
pgcc -Mextract=lib:mylib b.c
pgcc -Minline=lib:mylib a.c
pgcc -Minline=levels:10 a.c
pgcc -Minline=levels:2,foo,phoo,bar,lib:mylib a.c
```

OpenACC 2.0 Function Calling

```
#pragma acc routine worker
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // procedure call on the device
}
```

```
#pragma acc routine worker
void matvec( float* v, float* x,
             float* a, int i,
             int n ){
  float xx = 0;
  #pragma acc loop reduction(+:xx)
  for( int j = 0; j < n; ++j )
    xx += a[i*n+j]*v[j];
  x[i] = xx;
}
```

OpenACC 2.0 routine bind

```
#pragma acc routine worker bind(mvdev)
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

OpenACC 2.0 routine bind cont.

```
void matvec( float* v, float* x,
             float* a, int i, int n ){
    float xx=0.0;

    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}

#pragma acc routine worker nohost
void mvdev( float* v, float* x,
            float* a, int i, int n ){
    float xx = 0.0;
    #pragma acc loop reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}
```

Nested Parallelism

```
#pragma acc routine
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop …
for( int i = 0; i < n; ++i )
   matvec( v, x, i, n );


#pragma acc routine
matvec(…){
   #pragma acc parallel loop
   for( int i = 0; i < n; ++i ){…}
```

Nested Parallelism cont.

```
#pragma acc routine
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel num_gangs(1)
{
   matvec( v0, x0, i, n );
   matvec( v1, x1, i, n );
   matvec( v2, x2, i, n );
}

#pragma acc routine
  matvec(…){
#pragma acc parallel loop
   for( int i = 0; i < n; ++i ){…}
```

Dynamic Data Lifetimes

```
void init( int n ){
…
#pragma acc enter data copyin( x[0:100] )
…
#pragma acc enter data present_or_create( y[0:n] )
…
}


void fini( int n ){
…
#pragma acc exit data delete( x[0:100] )
…
#pragma acc exit data copyout( y[0:n] )
…
}
```

Global Data

```
float x[1000];
#pragma acc declare create(x)
// static allocation, host + device

float y[1000];
#pragma acc declare device_resident(y)
// static allocation, device-only

float z[10000];
#pragma acc declare link(z)
// static allocation on host, dynamic allocation on device
```

atomic operations

```
#pragma acc parallel loop
   for( j = 1; j < n-1; ++j ){
       y = x[j];
       i = y & 0xf;
       #pragma acc atomic update
           ++bin[i];
   }
```

// essentially the OpenMP atomic operations
// atomic update, read, write, capture
// only native data lengths supported

Other OpenACC 2.0 Features Not Covered Today

- Device-specific tuning, multiple devices
- Multiple host thread support
- Loop directive additions
- Asynchronous behavior additions
- New API routines

Key Value-add of the PGI Accelerator Compilers

PGI compilers are designed to enable performance-portable programming for all many-core and GPU Accelerator-based systems *without having to re-program for each new successive hardware advancement.*

Copyright Notice