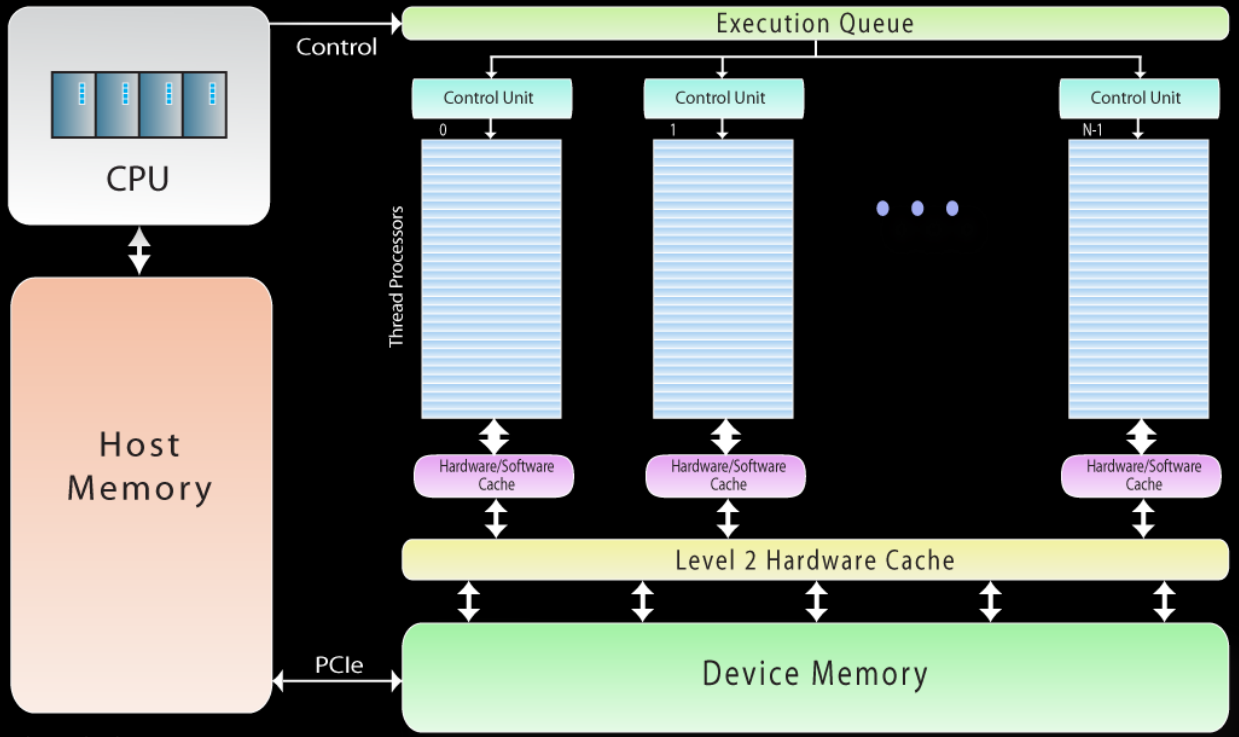
Programming GPUs for Performance and Portability

Michael Wolfe

michael.wolfe@pgroup.com

http://www.pgroup.com/accelerate

Multi-core x64 + NVIDIA GPU Architecture



```
void mmul( float* A, float* B, float* C, int N, int M, int L ){
   float *Ad, *Bd, *Cd;
   dim3 dimGrid, dimBlock;
   cudaMalloc( (void**)&Ad, N*L*sizeof(float) );
   cudaMalloc( (void**)&Bd, L*M*sizeof(float) );
   cudaMalloc( (void**)&Cd, N*M*sizeof(float) );
   cudaMemcpy( Ad, A, N*L*sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( Bd, B, N*L*sizeof(float), cudaMemcpyHostToDevice );
   dimGrid = dim3( N/16, M/16 );
   dimBlock = dim3( 16, 16, 1 );
   kmmul<<<dimGrid,dimBlock>>>( Ad,Bd,Cd,N,M,L );
   cudaMemcpy( C, Cd, N*L*sizeof(float), cudaMemcpyDeviceToHost );
   cudaFree( Ad );
   cudaFree( Bd );
   cudaFree( Cd );
}
```

```
void mmul( float* A, float* B, float* C, int N, int M, int L )
   float *Ad, *Bd, *Cd;
   dim3 dimGrid, dimBlock;
   cudaMalloc( (void**)&Ad, N*L*sizeof(float) );
   cudaMalloc( (void**)&Bd, L*M*sizeof(float) );
   cudaMalloc( (void**)&Cd, N*M*sizeof(float) );
   cudaMemcpy( Ad, A, N*L*sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( Bd, B, N*L*sizeof(float), cudaMemcpyHostToDevice );
   dimGrid = dim3( N/16, M/16 );
   dimBlock = dim3( 16, 16, 1 );
   kmmul<<<dimGrid,dimBlock>>>( Ad,Bd,Cd,N,M,L );
   cudaMemcpy( C, Cd, N*L*sizeof(float), cudaMemcpyDeviceToHost );
   cudaFree( Ad );
   cudaFree( Bd );
   cudaFree( Cd );
}
```

manage memory allocation

```
void mmul( float* A, float* B, float* C, int N, int M, int L )
   float *Ad, *Bd, *Cd;
   dim3 dimGrid, dimBlock;
   cudaMalloc( (void**)&Ad, N*L*sizeof(float) );
   cudaMalloc( (void**)&Bd, L*M*sizeof(float) );
   cudaMalloc( (void**)&Cd, N*M*sizeof(float) );
   cudaMemcpy( Ad, A, N*L*sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( Bd, B, N*L*sizeof(float), cudaMemcpyHostToDevice );
   dimGrid = dim3( N/16, M/16 );
   dimBlock = dim3( 16, 16, 1 );
   kmmul<<<dimGrid,dimBlock>>>( Ad,Bd,Cd,N,M,L );
   cudaMemcpy( C, Cd, N*L*sizeof(float), cudaMemcpyDeviceToHost );
   cudaFree( Ad );
   cudaFree( Bd );
   cudaFree( Cd );
}
```

manage data movement

```
void mmul( float* A, float* B, float* C, int N, int M, int L )
   float *Ad, *Bd, *Cd;
   dim3 dimGrid, dimBlock;
   cudaMalloc( (void**)&Ad, N*L*sizeof(float) );
   cudaMalloc( (void**)&Bd, L*M*sizeof(float) );
   cudaMalloc( (void**)&Cd, N*M*sizeof(float) );
   cudaMemcpy( Ad, A, N*L*sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( Bd, B, N*L*sizeof(float), cudaMemcpyHostToDevice );
   dimGrid = dim3( N/16, M/16 );
   dimBlock = dim3( 16, 16, 1 );
   kmmul<<<dimGrid,dimBlock>>>( Ad,Bd,Cd,N,M,L );
   cudaMemcpy( C, Cd, N*L*sizeof(float), cudaMemcpyDeviceToHost );
   cudaFree( Ad );
   cudaFree( Bd );
   cudaFree( Cd );
}
```

manage kernel configuration and launch

```
__global__ void matmul_kernel(float* A, float* B, float* C,
                              int N, int M, int L ){
   int i,j,k;
   float Cij;
   int tx = threadIdx.x, ty = threadidx.y;

   i = blockIdx.x * 16 + tx;
   j = blockIdx.y * 16 + ty;
   Cij = 0.0f;
   if( i < N && j < M ){
      for( k = 0; k < L; ++k )
        Cij += A[i+k*N] * B[k+j*L];
      C[i+j*N] = Cij;
   }
}
```

simple matmul kernel

```
__global__ void matmul_kernel(float* A, float* B, float* C,
                              int N, int M, int L ){
   int i,j,k,kb;
   float Cij;
   int tx = threadIdx.x, ty = threadIdx.y;
   __shared__ float Ab[16][16], Bb[16][16];
   i = blockidx.x * 16 + tx;
   j = blockidx.y * 16 + ty;
   Cij = 0.0f;
   for( kb = 0; kb < L; kb += 16 ){
      if( i<N && kb+ty<L ) Ab[tx][ty] = A[i+N*(kb+ty)];
      if( kb+tx<L && j<M ) Bb[tx][ty] = B[kb+tx+L*j];
      __syncthreads();
      if( i<N && j<M )
         for( k = 0; k < 16; ++k )
            if( kb+k < L ) Cij = Cij + Ab[tx][k] * Bb[k][ty];
      __syncthreads();
   }
   if( i < N && j < M ) C[i+N*j] = Cij;
}
```

faster version

```
__global__ void matmul_kernel(float* A, float* B, float* C,
                              int N, int M, int L ){
   int i,j,k,kb;
   float Cij;
   int tx = threadIdx.x, ty = threadIdx.y;
   __shared__ float Ab[16][16], Bb[16][16];
   i = blockidx.x * 16 + tx;
   j = blockidx.y * 16 + ty;
   Cij = 0.0f;
   for( kb = 0; kb < L; kb += 16 ){
      if( i<N && kb+ty<L ) Ab[tx][ty] = A[i+N*(kb+ty)];
      if( kb+tx<L && j<M ) Bb[tx][ty] = B[kb+tx+L*j];
      __syncthreads();
      if( i<N && j<M )
         for( k = 0; k < 16; ++k )
            if( kb+k < L ) Cij = Cij + Ab[tx][k] * Bb[k][ty];
      __syncthreads();
   }
   if( i < N && j < M ) C[i+N*j] = Cij;
}
```

split loop into chunks of 16

```
__global__ void matmul_kernel(float* A, float* B, float* C,
                              int N, int M, int L ){
   int i,j,k,kb;
   float Cij;
   int tx = threadIdx.x, ty = threadIdx.y;
   __shared__ float Ab[16][16], Bb[16][16];
   i = blockidx.x * 16 + tx;
   j = blockidx.y * 16 + ty;
   Cij = 0.0f;
   for( kb = 0; kb < L; kb += 16 ){
      if( i<N && kb+ty<L ) Ab[tx][ty] = A[i+N*(kb+ty)];
      if( kb+tx<L && j<M ) Bb[tx][ty] = B[kb+tx+L*j];
      __syncthreads();
      if( i<N && j<M )
         for( k = 0; k < 16; ++k )
            if( kb+k < L ) Cij = Cij + Ab[tx][k] * Bb[k][ty];
      __syncthreads();
   }
   if( i < N && j < M ) C[i+N*j] = Cij;
}
```

use "shared memory" as data cache

```
__global__ void matmul_kernel(float* A, float* B, float* C,
                              int N, int M, int L ){
   int i,j,k,kb;
   float Cij;
   int tx = threadIdx.x, ty = threadIdx.y;
   __shared__ float Ab[16][16], Bb[16][16];
   i = blockidx.x * 16 + tx;
   j = blockidx.y * 16 + ty;
   Cij = 0.0f;
   for( kb = 0; kb < L; kb += 16 ){
      if( i<N && kb+ty<L ) Ab[tx][ty] = A[i+N*(kb+ty)];
      if( kb+tx<L && j<M ) Bb[tx][ty] = B[kb+tx+L*j];
      __syncthreads();
      if( i<N && j<M )
         for( k = 0; k < 16; ++k )
            if( kb+k < L ) Cij = Cij + Ab[tx][k] * Bb[k][ty];
      __syncthreads();
   }
   if( i < N && j < M ) C[i+N*j] = Cij;
}
```

conditionals to prevent matrix overrun

PGI Accelerator Directives

```
#pragma acc region for
   for( i = 0; i < N; ++i )
      for( j = 0; j < M; ++j )
         for( k = 0; k < L; ++k )
            C[i][j] += A[i][k] * B[k][j];
```

PGI Accelerator Directives

```
#pragma acc region for
   for( i = 0; i < N; ++i )
      for( j = 0; j < M; ++j )
         for( k = 0; k < L; ++k )
            C[i][j] += A[i][k] * B[k][j];
```

```
!$acc region do
   do i = 1, N
      do j = 1, M
         do k = 1, L
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
         enddo
      enddo
   enddo
```

PGI Accelerator Compilers

```
#pragma acc region for
  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;
...
```

Unified Object

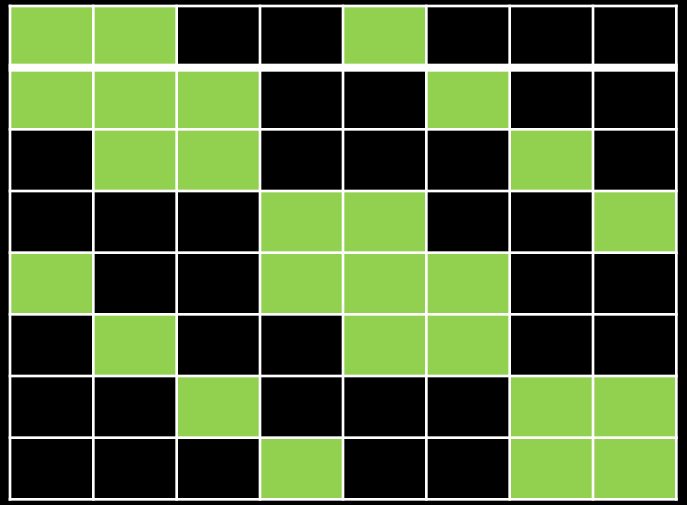
execute

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

Creative Similarities

- Select Algorithm
- Select Data Structure
- CUDA / OpenCL
  - encourages algorithm / data restructuring
- Directives
  - Faster to get started

Diagonal Representation



7	2			5			
9	6	4			7		
	8	2	0			9	
		0	7	8			6
1			7	5	4		
	3			3	3	0	
		4			0	4	2
			1			1	3

Diagonal Representation

		7	2	5
	9	6	4	7
	8	2	0	9
	0	7	8	6
1	7	5	4	
3	3	3	0	
4	0	4	2	
1	1	3		

-4	-1	0	1	4

7	2			5			
9	6	4			7		
	8	2	0			9	
		0	7	8			6
1			7	5	4		
	3			3	3	0	
		4			0	4	2
			1			1	3

Diagonal Representation

		7	2	5
	9	6	4	7
	8	2	0	9
	0	7	8	6
1	7	5	4	
3	3	3	0	
4	0	4	2	
1	1	3		

-4	-1	0	1	4

```
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*nzeros+d] * v[j];
  }
  x[i] = val;
}
```

Sparse Matrix-Vector Multiply

```
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*nzeros+d] * v[j];
  }
  x[i] = val;
}
```

Sparse Matrix-Vector Multiply

```
#pragma acc region for copyin( m[0:nzeros*nrows-1], v[0:nrows-1] )
  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*nzeros+d] * v[j];
    }
    x[i] = val;
  }
```

Sparse Matrix-Vector Multiply

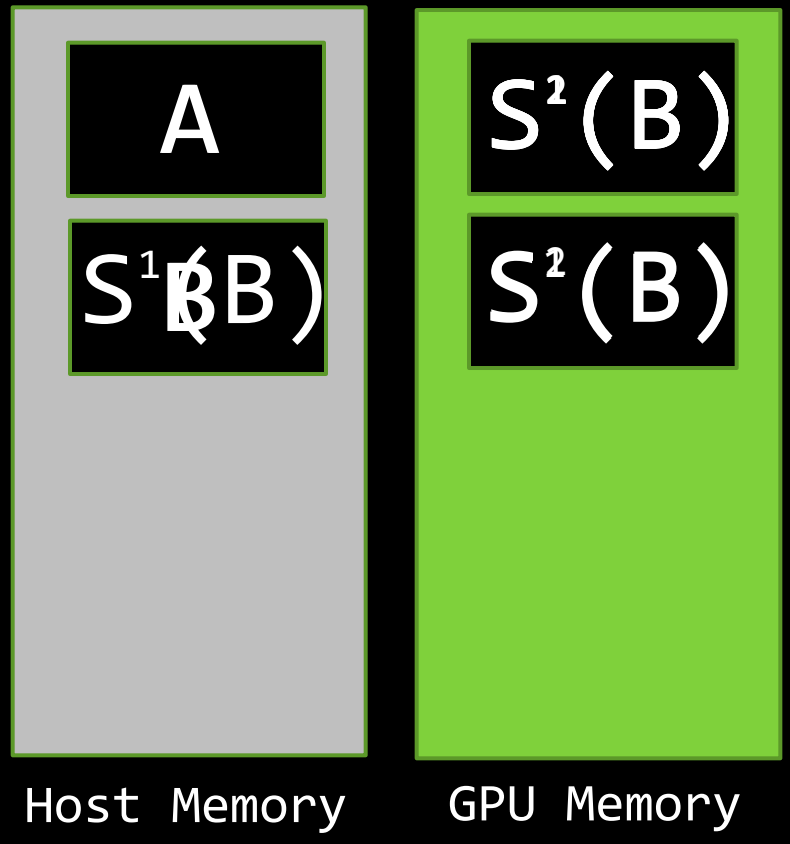
```
#pragma acc region for copyin( m[0:nzeros*nrows-1], v[0:nrows-1] )
  for( i = 0; i < nrows; ++i ){
    float val = 0.0f;
    for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
        val += m[i+nrows*d] * v[j];
    }
    x[i] = val;
  }
```

Programming Effort

- Select parts of the application to accelerate
- Select what data to move to GPU, and when
- CUDA/OpenCL
  - more explicit control
  - more intrusive program restructuring
- Directives
  - incremental, implicit
  - compiler feedback

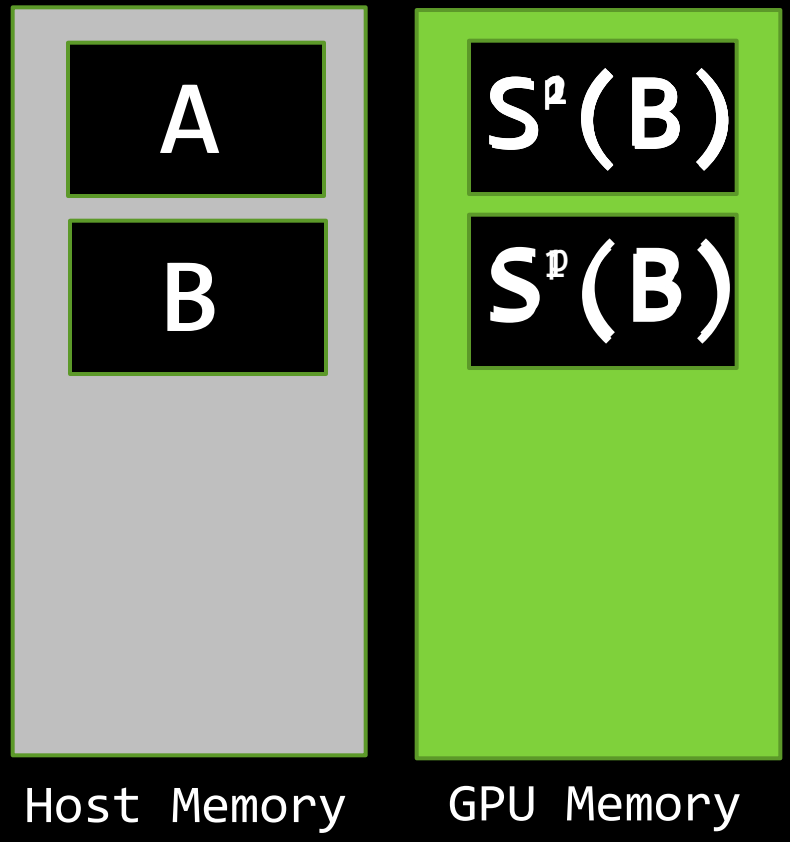
PGI Accelerator compute region

```
for (iter = 1; iter <= niters; ++iter){
  #pragma acc region
  {
  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];
  }
}
```



PGI Accelerator data region

```
#pragma acc data region \
        copy(b[0:n-1][0:m-1]) \
        local(a[0:n-1][0:m-1])
{
for (iter = 1; iter <= p; ++iter){
  #pragma acc region
  {
  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];
  }
}
}
```



Performance

- Directives equivalent to straightforward CUDA/OpenCL
- CUDA/OpenCL
  - texture memory
  - manual loop unrolling
- Directives
  - automatic use of "shared memory"
  - automatic selection of thread block shape/size

SEISMIC_CMPL Step 1: Add Compute Regions

```
!$acc region do
  do k = kmin,kmax
    do j = NPOINTS_PML+1, NY-NPOINTS_PML
      do i = NPOINTS_PML+1, NX-NPOINTS_PM:
      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
```

Compiler Feedback

```
% pgfortran -Mmpi=mpich2 -fast -ta=nvidia -Minfo=accel
seismic_CPML_3D_isotropic_MPI_ACC_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
```

SEISMIC_CMPL Step 2: Data Movement

```
!$acc data region                                                    &
!$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 region
```

SEISMIC_CMPL Step 3: Tuning

```
!$acc do vector(4)
  do k=k2begin,NZ_LOCAL
   kglobal = k + offset_k
!$acc do parallel, vector(4)
   do j=2,NY
!$acc do parallel, 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	948	
ACC Step 1	2	0	2	3599	10
ACC Step 2	2	0	2	194	180
ACC Step 3	2	0	2	**167**	120

System Info:

4 Core Intel Core-i7 920 Running at 2.67Ghz

Includes 2 Tesla C2070 GPUs

Problem Size: 101x641x128

5x in 5 hours!

Cluster Timings

Version	Size	MPI Processes	OpenMP Threads	GPUs	Time (sec)
MPI/OMP	101x641x512	16	256	0	607
MPI/ACC	101x641x512	16	0	16	186
MPI/OMP	101x641x1024	16	256	0	1920
MPI/ACC	101x641x1024	16	0	16	**335**

System Info: 16 Nodes
Four socket AMD 8356 @2.3GHZ (16 cores per node)
Tesla C1060

Still 5x!

Portability

- OpenCL provides functional, not performance portability
  - Du, Weber, Luszcek, Tomov, Peterson, Dongarra, *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming*, August 2010, preprint, submitted for publication
  - Vendors allow OpenCL extensions and extra functionality
  - ATI-optimized kernel reaches 50% peak on ATI, 4% on Fermi
  - Fermi-optimized kernel reaches 40% peak on Fermi, <1% on ATI
    - (slight modification reaches 22% on ATI)
- Higher level, directive programming insulates you from target-specific tuning

OpenACC™ API

PGI, Cray, NVIDIA, CAPS

Directives similar to PGI Accelerator, OpenMP

PGI Accelerator Directives

```
#pragma acc data region copy(b[0:n*m-1]) local(a[0:n*m-1])

{
  for (iter = 1; iter <= p; ++iter){
    #pragma acc region

    {
      for (i = 1; i < n-1; ++i)
        for (j = 1; j < m-1; ++j){
          a[i*m+j]=w0*b[i*m+j]+
                   w1*(b[(i-1)*m+j]+b[(i+1)*m+j]+
                       b[i*m+j-1]+b[i*m+j+1])+
                   w2*(b[(i-1)*m+j-1]+b[(i-1)*m+j+1]+
                       b[(i+1)*m+j-1]+b[(i+1)*m+j+1]);
        }
      for( i = 1; i < n-1; ++i )
        for( j = 1; j < m-1; ++j )
          b[i*m+j] = a[i*m+j];
    }
  }
```

PGI Accelerator vs. OpenACC

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

OpenACC™ API

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

Programming NVIDIA GPUs with PGI Accelerator Directives

- Appropriate algorithm (think nested parallel loops)
- Appropriate data structure (vectors, arrays, simple indexing)
- Read the –Minfo messages
- Manage data moving to and from GPU (CUDA or data regions)
- Optimize, tune for strides, locality
- Hindrances and Workarounds
- PGI Unified Binary

www.pgroup.com/accelerate