Performance Optimization on GPUs

Michael Wolfe
Michael.Wolfe@pgroup.com
http://www.pgroup.com

November 2011
AMD “Magny-Cours”
Host Architecture Features

- Register files (how many? what kind?)
- Functional units (how many?), cache (I, D)
- Execution pipeline (pipeline stages?)
  - branch prediction, hazards (types?)
  - pipelined functional units, superpipelining, register bypass
  - stalls, avoiding stalls
- Multiscalar execution (superscalar, control unit lookahead)
  - LIW (long instruction word)
- Multithreading, Simultaneous multithreading
- Vector instruction set
- Multiprocessor, Multicore, coherent caches (MESI protocols)
Making It Faster

- **Processor:**
  - 
  - 
  - 

- **Memory**
  - Latency
  - Bandwidth
Abstracted x64+Tesla Architecture
Abstracted x64+Fermi Architecture
GPU Architecture Features

- Optimized for high degree of regular parallelism
- Classically optimized for low precision
  - Fermi supports double precision at ½ single precision bandwidth
- High bandwidth memory (Fermi supports ECC)
- Highly multithreaded (slack parallelism)
- Hardware thread scheduling
- Non-coherent software-managed data caches
  - Fermi has two-level hardware data cache
- No multiprocessor memory model guarantees
  - some guarantees with fence operations
Tesla-10 Features Summary

- Massively parallel thread processors
  - Organized into multiprocessors
    - up to 30, see deviceQuery or pgaccelinfo
  - Physically: 8 thread processors per multiprocessor
  - ISA: 32 threads per warp
  - Logically: Thread block is quasi-SIMD
- Memory hierarchy
  - host memory, device memory, constant memory, shared memory, register
- Queue of operations (kernels) on device
Fermi (Tesla-20) Features Summary

- Massively parallel thread processors
  - Organized into multiprocessors
    - up to 16, see deviceQuery or pgaccelinfo
    - Physically: two groups of 16 thread processors per multiprocessor
    - ISA: still 32 threads per warp, dual issue for 32-bit code

- Memory hierarchy
  - host memory, device memory (two level hardware cache), constant memory, (configurable) shared memory, register

- Queue of operations (kernels) on device

- ECC memory protection (supported, not default)

- Much improved double precision performance

- Hardware 32-bit integer multiply
Parallel Programming on CPUs

- **Instruction level parallelism (ILP)**
  - Loop unrolling, instruction scheduling
- **Vector parallelism**
  - Vectorized loops (or vector intrinsics)
- **Thread level / Multiprocessor / multicore parallelism**
  - Parallel loops, parallel tasks
  - Posix threads, OpenMP, Cilk, TBB, ..... 
- **Large scale cluster / multicomputer parallelism**
  - MPI (& HPF, co-array Fortran, UPC, Titanium, X10, Fortress, Chapel)
Parallel Programming on GPUs

- Instruction level parallelism (ILP)
  - Loop unrolling, instruction scheduling
- Vector parallelism
  - CUDA Thread Blocks, OpenCL Work Groups
- Thread level / Multiprocessor / multicore parallelism
  - CUDA Grid, OpenCL
Performance Tuning

- Performance Measurement
- Choose an appropriately parallel algorithm, appropriate data structure
- Optimize data movement between host and GPU
  - frequency, volume, regularity
- Optimize device memory accesses
  - strides, alignment
  - use shared memory
  - use constant memory, texture memory
- Optimize kernel code
  - redundant code elimination
  - loop unrolling
  - Optimize compute intensity
    - unroll the parallel loop
Host-GPU Data Movement

- Avoid altogether
- Move outside of loops
- Better to move a whole array than subarray
- Update halo regions rather than whole array
  - use GPU to move halo region to contiguous area?
- Use streams, overlap data / compute
  - requires pinned host memory
Occupancy

- How many simultaneously active warps / maximum (maximum is 24, 32 (Tesla-10), 48 (Fermi))

- Limits
  - threads per multiprocessor
  - threads per thread block (512 / 1024)
  - thread blocks per multiprocessor (8)
  - register usage (8K / 16K / 32K registers / multiprocessor)
    - each warp uses 32n, so 16Kreg = 512wreg
  - shared memory usage (16KB / 48KB)

- Low occupancy often leads to low performance

- High occupancy does not guarantee high performance
Execution Configuration

- Execution configuration affects occupancy
- Want many threads per thread block
  - multiple of 32
  - 64, 128, 192, 256
- Want many many thread blocks
Divergence

- Scalar threads executing in SIMD mode
  - if( threadIdx%x <= 10 ) then
    foo = foo * 2
  else
    foo = 0
  endif

- Each path taken
  - do i = 1, threadIdx%x
    a(threadIdx%x,i) = 0
  enddo

- Only matters within a warp
Divergence

- Pad arrays to multiples of block size
  
  \[ i = (\text{blockidx} \% x - 1) \times 64 + \text{threadidx} \% x \]
  
  \[ \text{if}( i \leq N ) \ A(i) = \ldots \]
Global Memory

- **Stride-1, aligned accesses**
  - address is aligned to mod(threadidx%x,16)
  - threadIdx%x and threadIdx%x+1 access consecutive addresses
  - alignment critical for Compute Capability 1.0, 1.1

- **Using shared memory as data cache**
  - Redundant data access within a thread
  - Redundant data access across threads
  - Stride-1 data access within a thread
Redundant access within a GPU Thread

! threadIdx%x from 1:64
! this thread block does 256 ‘i’ iterations
ilo = (blockidx%x-1)*256
ihi = blockidx*256 - 1
...
do j = jlo, jhi
   do i = ilo+threadidx%x, ihi, 64
      A(i,j) = A(i,j) * B(i)
   enddo
enddo
Redundant access within a GPU Thread

real, shared :: BB(256)
...
do ii = 0, 255, 64
    BB(threadidx%x+ii) = B(ilo+ii)
enddo
call syncthreads()
do j = jlo, jhi
    do i = ilo+threadidx%x, ihi, 64
        A(i,j) = A(i,j) * BB(i-ilo)
    enddo
enddo
Redundant access across GPU Threads

! threadidx%x from 1:64
i = (blockidx%x-1)*64 + threadIdx%x

... 
do j = jlo, jhi
   A(i,j) = A(i,j) * B(j)
enddo
Redundant access across GPU Threads

real, shared :: BB(64)

i = (blockidx%x-1)*64 + threadIdx%x
...

do jb = jlo, jhi, 64
  BB(threadidx%x) = B(jb+threadidx%x)
  call syncthreads()
  do j = jb, min(jhi,jb+63)
    A(i,j) = A(i,j) * BB(j-jb+1)
  enddo
  call syncthreads()
enddo
Stride-1 Access within a GPU thread

! threadidx%x from 1:32
i = (blockidx%x-1)*32 + threadidx%x

... ix = indx(i)
do j = jlo, jhi
    A(i,j) = A(i,j) * B(ix+j)
endo
real, shared :: BB(32,32)
integer, shared :: IXX(32)
i = (blockidx%x-1)*32 + threadIdx%x
...
ix = indx(i)
IXX(threadidx%x) = ix
call syncthreads()
do jb = jlo, jhi, 32
   do j = 1, 32
      BB(threadidx%x,j) = B(IXX(j)+threadidx%x)
   enddo
   do j = jb, min(jhi,jb+31)
      A(i,j) = A(i,j) * BB(j,threadidx%x)
   enddo
enddo
Stride-1 Access within a GPU thread

```fortran
real, shared :: BB(33,32)
integer, shared :: IXX(32)
i = (blockidx%x-1)*32 + threadIdx%x
...
ix = indx(i)
IXX(threadidx%x) = ix
call syncthreads()
do jb = jlo, jhi, 32
  do j = 1, 32
    BB(threadidx%x,j) = B(IXX(j)+threadidx%x)
  enddo
  do j = jb, min(jhi,jb+31)
    A(i,j) = A(i,j) * BB(j,threadidx%x)
  enddo
enddo
```
Shared Memory

- 16 memory banks

- Use `threadidx%x` in leading (stride-1) dimension

- Avoid stride of 16

- Shared memory also used to pass kernel arguments, affects occupancy
Unroll the Parallel Loop

- If thread ‘j’ and ‘j+1’ share data, where
  - j is a parallel index
  - j is not the stride-1 index

- Unroll two or more iterations of ‘j’ into the kernel
Low-level Optimizations

- instruction count optimizations
  - loop unrolling (watch memory access patterns)
  - loop fusion

- minimize global memory accesses
  - use scalar temps
  - scalarizing arrays
  - downsides:
    increased register usage
    spills to “local memory”

- use shared memory for threadIdx-invariant values
  - nvcc -Xptxas=-v
  - pgfortran -Mcuda=ptxinfo
Other Hacks

- Low precision transcendental functions (\_sinf, ...)
  - Compiler Option?
- 24-bit integer multiply (Tesla only)
  - Compiler Option?
- atomic operations
- warp-vote functions, warp-level programming
SAXPY on Host

subroutine saxpy (A,X,Y,N)
    real(4) :: A, X(N), Y(N)
    integer :: N, i
    do i = 1,N
        X(i) = A * X(i) + Y(i)
    enddo
end subroutine

void saxpy( float a, float* x, float* y, int n ){
    int i;
    for( i = 0; i < n; ++i ){
        x[i] = a*x[i] + y[i];
    }
}
__global__ void
saxpy_kernel( float a, float* x, float* y, int n ){
    int i;
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= n ) x[i] = a*x[i] + y[i];
}
module kmod
  use cudafor
contains
  attributes(global) subroutine saxpy_kernel(A,X,Y,N)
    real(4), device :: A, X(N), Y(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*blockdim%x + threadIdx%x
    if( i <= N ) X(i) = A*X(i) + Y(i)
  end subroutine
end module
CUDA C SAXPY Host Code

```c
void saxpy( float a, float* x, float* y, int n ){
    float *xd, *yd;
    cudaMemcpy( (void**)&xd, x, n*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( (void**)&yd, y, n*sizeof(float), cudaMemcpyHostToDevice );
    saxpy_kernel<<<(n+31)/32, 32>>>( a, xd, yd, n );
    cudaMemcpy( x, xd, n*sizeof(float), cudaMemcpyDeviceToHost );
    cudaFree( xd ); cudaFree( yd );
}
```
subroutine saxpy( A, X, Y, N )
  use kmod
  real(4) :: A, X(N), Y(N)
  integer :: N
  real(4), device, allocatable, dimension(:):: &
    Xd, Yd
  allocate( Xd(N), Yd(N) )
  Xd = X(1:N)
  Yd = Y(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd, Yd, N)
  X(1:N) = Xd
  deallocate( Xd, Yd )
end subroutine
Measuring Performance

- event timer
- profiler
- trace
event timers

cudaEvent_t e1, e2;
float usec;

cudaEventRecord( e1, 0 );
saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
cudaEventRecord( e2, 0 );

cudaEventSynchronize( e2 );
cudaEventElapsedTime( &usec, e1, e2 );
## Improving Performance

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Upside</th>
<th>Downside</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
real(4) function summit(X,N)
  real(4) :: X(N), sum
  integer :: N, I
  sum = 0.0
  do I = 1,N
    sum = sum + X(I)
  enddo
  summit = sum
end subroutine

float summit( float* x, int n ){
  int i;
  float sum = 0.0;
  for( i = 0; i < n; ++i )
    sum = sum + x[i];
  return sum;
}
CUDA C SUM reduction Code

#define BSIZE 128
__global__ void
sum_kernel( float* x, int n ){
    int i, k;
    float sum = 0.0f;
    __shared__ float lsum[BSIZE];
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i < n ) sum = x[i];
    i = threadIdx.x;
    lsum[i] = sum;
    __syncthreads();
}
CUDA C SUM reduction Code (2)

```c
k = blockDim.x;
while( k > 1 ){
    k >>= 1;
    if( i < k ) lsum[i] += lsum[i+k];
    __syncthreads();
}
/* partial sum in lsum[0]. Now what? */
...
CUDA Fortran SUM Reduction Code

module kmod
  use cudafor
contains
  attributes(global) subroutine sum_kernel(X,N)
    real(4), device :: X(N)
    integer, value :: N
    integer :: i, k
    real(4) :: sum
    real(4), shared :: lsum[128]
    i = (blockidx%x-1)*blockdim%x + threadIdx%x
    sum = 0.0
    if( i <= N ) sum = X(i)
  call syncthreads()
CUDA Fortran SUM reduction (2)

```fortran
k = blockDim%x
do while( k > 1 )
    k = k / 2
    if( i < k ) lsum(i) = lsum(i) + lsum(i+k)
call syncthreads()
endo
d! partial sum in lsum[0]. Now what?
...
end subroutine
end module
```
```c
#define BSIZE 128
__global__ void
sum_kernel( float* x, int n, float* psum ){
    int i, k;
    float sum = 0.0f;
    __shared__ float lsum[BSIZE];
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= n ) sum = x[i];
    i = threadIdx.x;
    lsum[i] = sum;
    __syncthreads();
    psum[i] = sum;
}```
CUDA C SUM reduction Code (2)

```c
k = blockDim.x;
while( k > 1 ){
    k >>= 1;
    if( i < k ) lsum[i] += lsum[i+k];
    __syncthreads();
}
/* partial sum in lsum[0].  Now what? */
psum[blockIdx.x] = lsum[0];
```
CUDA Fortran SUM Reduction Code

```fortran
module kmod

  use cudafor

contains

  attributes(global) subroutine sum_kernel(X,N)
  real(4), device :: X(N)
  integer, value :: N
  integer :: i, k
  real(4) :: sum
  real(4), shared :: lsum[128]

  i = (blockidx%x-1)*blockdim%x + threadIdx%x
  sum = 0.0
  if ( i <= N ) sum = X(i)
  call syncthreads()

end subroutine sum_kernel
```

CUDA Fortran SUM reduction (2)

```fortran
k = blockDim%x
do while( k > 1 )
    k = k / 2
    if( i < k ) lsum(i) = lsum(i) + lsum(i+k)
    call syncthreads()
endo
! partial sum in lsum[0]. Now what?
...
end subroutine
end module
```
CUDA C SUM Host Code

```c
float summit(float* x, int n ){
    float *xd;
    float sum;
    cudaMalloc( (void**)&xd, n*sizeof(float) );
    cudaMemcpy( xd, x, n*sizeof(float),
                cudaMemcpyHostToDevice );
    sum_kernel<<< (n+BSIZE-1)/BSIZE, BSIZE >>>
        ( xd, n );
    cudaMemcpy( &sum, xd, sizeof(float),
                cudaMemcpyHostToDevice );
    cudaFree( xd );
}
```
subroutine summit( X, N )
  use kmod
  real(4) :: X()
  integer :: N
  real(4), device, allocatable, dimension(:):: Xd
  real(4) :: sum
  allocate( Xd(N) )
  Xd = X(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>( Xd, N )
  sum = Xd(1)
  deallocate( Xd )
end subroutine
Measuring Performance

- event timer
- profiler
- trace
## Improving Performance

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Upside</th>
<th>Downside</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-50
CUDA C Matrix Multiplication
Code Walkthrough

- for( i = 0; i < N; ++i )
  for( j = 0; j < M; ++j ){
    C[i][j] = 0.0;
    for( k = 0; k < L; ++k )
      C[i][j] = C[i][j] + A[k][i]*B[j][k];
  }

- Kernel computes a 16x16 submatrix
  - assume matrix sizes are divisible by 16
- thread block is (16,16), grid is (N/16,M/16)
  - each thread accumulates one element of the 16x16 block of C
  - k loop is strip mined in strips of size 16
  - threads cooperatively load a 16x16 block of A and B
__global__ void kmmul(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;
    for( k = 0; k < L; ++k )
        Cij += A[i+k*N] * B[k+j*L];
    C[i+j*N] = Cij;
}

__global__ void kmmul(float* A, float* B, float* C, 
        int N, int M, int L ){
    int i,j,k,kb,tx,ty;
    __shared__ float Ab[16][16], Bb[16][16];
    float Cij;
    tx = threadIdx.x
    ty = threadIdx.y;
    i = blockIdx.x * 16 + tx;
    j = blockIdx.y * 16 + ty;
    Cij = 0.0f;
    ! continued
for( kb = 0; kb < L; kb += 16 ){
    Ab[tx][ty] = A[i+N*(kb+ty-1)];
    Bb[tx][ty] = B[kb+tx-1+N*j];
    __syncthreads();
    for( k = 0; k < 16; ++k )
        Cij = Cij + Ab[tx][k] * Bb[k][ty];
    __syncthreads();
}
C[i+N*j] = Cij;
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;

dimGrid = dim3( N/16, M/16 );
dimBlock = dim3( 16, 16, 1 );
kmmul<<<dimGrid,dimBlock>>>( Ad, Bd, Cd, N, M, L );
}
CUDA Fortran Matrix Multiplication
Code Walkthrough

- `do i = 1, N
  do j = 1, M
    C(i,j) = 0.0
    do k = 1, L
      C(i,j) = C(i,j) + A(i,k)*B(k,j)`

- **Kernel computes a 16x16 submatrix**
  - initially, assume matrix sizes are divisible by 16
- **thread block is (16,16), grid is (N/16,M/16)**
  - each thread accumulates one element of the 16x16 block of C
  - k loop is strip mined in strips of size 16
  - threads cooperatively load a 16x16 block of A and B
module mmulmod
contains
  attributes(global) subroutine kmmul( A,B,C,N,M,L)
    real,device :: A(N,L),B(L,M),C(N,M)
    integer,value :: N,M,L
    integer :: i,j,k
    real :: Cij
    i = (blockidx%x-1) * 16 + tx
    j = (blockidx%y-1) * 16 + ty
    Cij = 0.0
    do k = 1, L
      Cij = Cij + A(i,k) * B(k,j)
    enddo
    C(i,j) = Cij
  end subroutine
end module
module mmulmod
contains
  attributes(global) subroutine kmmul( A,B,C,N,M,L)
  real,device :: A(N,L),B(L,M),C(N,M)
  integer,value :: N,M,L
  integer :: i,j,k,kb,tx,ty
  real,shared :: Ab(16,16), Bb(16,16)
  real :: Cij
  tx = threadidx%x ; ty = threadidx%y
  i = (blockidx%x-1) * 16 + tx
  j = (blockidx%y-1) * 16 + ty
  Cij = 0.0
  ! continued
do kb = 1, L, 16
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j)
    call syncthreads()
    do k = 1,16
        Cij = Cij + Ab(tx,k) * Bb(k,ty)
    enddo
    call syncthreads()
endo
do kb = 1, L, 16
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j)
    call syncthreads()
endo
call syncthreads()
endo
C(i,j) = Cij
endo subroutine
do kb = 1, L, 16
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j)
    call syncthreads()
    do k = 1,16
        Cij = Cij + Ab(tx,k) * Bb(k,ty)
    enddo
    call syncthreads()
endo
call syncthreads()
endo
C(i,j) = Cij
endo subroutine
end module
subroutine mmul( A, B, C )
   use cudafor
   use mmulmod
   real, dimension(:,:) :: A, B, C
   real, device, allocatable, dimension(:,:): : Ad,Bd,Cd
   type(dim3) :: dimGrid, dimBlock
   integer :: N, M, L
   N = size(C,1) ; M = size(C,2) ; L = size(A,2)
   allocate( Ad(N,L), Bd(L,M), Cd(N,M) )
   Ad = A(1:N,1:L)
   Bd = B(1:L,1:M)
   dimGrid = dim3( N/16, M/16 )
   dimBlock = dim3( 16, 16, 1 )
   call mmul<<<dimGrid,dimBlock>>>( Ad,Bd,Cd,N,M,L )
   C(1:N,1:M) = Cd
   deallocate( Ad, Bd, Cd )
end subroutine
subroutine mmul( A, B, C )
  use cudafor
  use mmulmod
  real, dimension(:,:), device :: A, B, C
  type(dim3) :: dimGrid, dimBlock
  integer :: N, M, L
  N = size(C,1) ; M = size(C,2) ; L = size(A,2)
  dimGrid = dim3( N/16, M/16 )
  dimBlock = dim3( 16, 16, 1 )
  call mmul<<<dimGrid,dimBlock>>>( A,B,C,N,M,L )
end subroutine
Profiling CUDA Fortran

$ make
pgfortran -Minfo=ccff -Mcuda -c ../mm.cuf
gfortran -Minfo=ccff -Mcuda -c ../mmdrv.f90
gfortran -Minfo=ccff -Mcuda -o mm mmdrv.o mm.o
$
$ pgcollect -cuda=gmem,cc20 ./mm
[...program output...]
target process has terminated, writing profile data
$
$ pgprof -exe ./mm

- Uses the same GPU counters as cudaprof
  - Relates to host performance and source code
- Build as usual
  - Here we added -Minfo=ccff to enhance profile data
- Run pgcollect
  - collect global memory data (gmem) on Fermi (cc20)
- Then invoke the PGPROF performance profiler
## CUDA Fortran Profile

### Function Times

<table>
<thead>
<tr>
<th>Function</th>
<th>Seconds</th>
<th>CUDA GPU Secs</th>
<th>CUDA CPU Secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstk</td>
<td>1.16667</td>
<td>0.002362 = 21%</td>
<td>0.00772 = 44%</td>
</tr>
<tr>
<td>mmul_driv</td>
<td>0.5</td>
<td>0.000780 = 50%</td>
<td>0.000992 = 40%</td>
</tr>
<tr>
<td>__libc_pause</td>
<td>0.333333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmul</td>
<td>0.333333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_IO_old_init</td>
<td>0.16667</td>
<td>0.003287 = 29%</td>
<td>0.002794 = 16%</td>
</tr>
<tr>
<td>fchmodat</td>
<td>0.16667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[System_Time]</td>
<td>0.16667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__close_nocancel</td>
<td>0.16667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Data_Transfer_to_Host]</td>
<td>0.16667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmul_kernel</td>
<td>0.16667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Data_Transfer_to_Device]</td>
<td>0.16667</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CUDA Fortran Profile - Kernel

<table>
<thead>
<tr>
<th>Function</th>
<th>Seconds</th>
<th>CUDA GPU Secs</th>
<th>CUDA CPU Secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstk</td>
<td>1.16667</td>
<td>0.002362</td>
<td>0.00772</td>
</tr>
<tr>
<td>mmul_drv</td>
<td>0.5</td>
<td>0.005763</td>
<td>0.005992</td>
</tr>
<tr>
<td>_libc_pause</td>
<td>0.33333</td>
<td>0.005763</td>
<td></td>
</tr>
<tr>
<td>mmul</td>
<td>0.33333</td>
<td>0.005763</td>
<td></td>
</tr>
<tr>
<td>_IO_old_init</td>
<td>0.16667</td>
<td>0.005763</td>
<td>0.005992</td>
</tr>
<tr>
<td>fchmodat</td>
<td>0.16667</td>
<td>0.005763</td>
<td></td>
</tr>
<tr>
<td>[System_Time]</td>
<td>0.16667</td>
<td>0.005763</td>
<td></td>
</tr>
<tr>
<td>__close_nocancel</td>
<td>0.16667</td>
<td>0.005763</td>
<td></td>
</tr>
<tr>
<td>[Data_Transfer_to_Host]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmul_kernel</td>
<td>0.16667</td>
<td>0.005763</td>
<td>0.005992</td>
</tr>
<tr>
<td>[Data_Transfer_to_Device]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CUDA Kernel Call Count: 100
Grid Size: 8x8
Thread Block Size: 16x16x1
Avg Regs/Thread: 21
Avg Occupancy: 0.833
Global L1 Load Hits: 58,831
Global L1 Load Misses: 69,169
Global Loads: 64,000
Global Stores: 4,000
Avg Static Shmem/Block: 204,800

Profiled: ./mm on Tue Sep 21 15:20:31 PDT 2010 | Profile: ./pgprof.out
## CUDA Fortran Profile - Data

<table>
<thead>
<tr>
<th>Function</th>
<th>Seconds</th>
<th>CUDA GPU Secs</th>
<th>CUDA CPU Secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstk</td>
<td>1.16667 = 39%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmul_drv</td>
<td>0.5     = 17%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__libc_pause</td>
<td>0.33333 = 11%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmul</td>
<td>0.33333 = 11%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_IO_old_init</td>
<td>0.16667 = 6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fchmodat</td>
<td>0.16667 = 6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[System_Time]</td>
<td>0.16667 = 6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__close_nocancel</td>
<td>0.16667 = 6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Data_Transfer_to_Host]</td>
<td></td>
<td>0.002362 = 21%</td>
<td>0.00772 = 44%</td>
</tr>
<tr>
<td>mmul_kernel</td>
<td></td>
<td>0.005763 = 50%</td>
<td>0.005992 = 40%</td>
</tr>
<tr>
<td>[Data_Transfer_to_Device]</td>
<td></td>
<td>0.003287 = 29%</td>
<td>0.002794 = 16%</td>
</tr>
</tbody>
</table>

CUDA Kernel Call Count: 200
Total MB/sec: 12.5
module mmulmod
contains
  attributes(global) subroutine mmul( A,B,C,N,M,L)
    real,device :: A(N,L),B(L,M),C(N,M)
    integer,value :: N,M,L
    integer :: i,j,kb,k,tx,ty
    real,shared :: Ab(16,16), Bb(16,16)
    real :: Cij
    tx = threadidx%x ; ty = threadidx%y
    i = (blockidx%x-1) * 16 + tx
    j = (blockidx%y-1) * 16 + ty
    Cij = 0.0
    ! continued
do kb = 1, L, 16
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j)
    call syncthreads()
    do k = 1,16
        Cij = Cij + Ab(tx,k) * Bb(k,ty)
    enddo
    call syncthreads()
enddo
C(i,j) = Cij
end subroutine
module mmulmod
contains
  ! unroll two 'j' iterations
attributes(global) subroutine mmul( A,B,C,N,M,L)
  real,device :: A(N,L),B(L,M),C(N,M)
  integer,value :: N,M,L
  integer :: i,j,kb,k,tx,ty
  real,shared :: Ab(16,16), Bb(16,16)
  real :: Cij1, Cij2
  tx = threadIdx%x ; ty = threadIdx%y
  i = (blockIdx%x-1) * 16 + tx
  j1 = (blockIdx%y-1) * 32 + ty
  j2 = j1+16
  Cij1 = 0.0
  Cij2 = 0.0
  ! continued
do kb = 1, L, 16
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j1)
    call syncthreads()
    do k = 1,16
        Cij1 = Cij1 + Ab(tx,k) * Bb(k,ty)
    enddo
    call syncthreads()
    Bb(tx,ty) = B(kb+tx-1,j2)
    call syncthreads()
    do k = 1,16
        Cij2 = Cij2 + Ab(tx,k) * Bb(k,ty)
    enddo
    call syncthreads()
enddo
C(i,j1) = Cij1
C(i,j2) = Cij2
end subroutine
end module
Jacobi Relaxation

\[
\text{change} = \text{tolerance} + 1.0 \\
do \text{ while}(\text{change} > \text{tolerance}) \\
\quad \text{change} = 0 \\
\quad \text{do } i = 2, m-1 \\
\quad \quad \text{do } j = 2, n-1 \\
\quad \quad \quad \text{newa}(i,j) = w_0 \ast a(i,j) + \& \\
\quad \quad \quad \quad \quad w_1 \ast (a(i-1,j) + a(i,j-1) + \& \\
\quad \quad \quad \quad \quad \quad a(i+1,j) + a(i,j+1)) + \& \\
\quad \quad \quad \quad \quad \quad \quad w_2 \ast (a(i-1,j-1) + a(i-1,j+1) + \& \\
\quad \quad \quad \quad \quad \quad \quad \quad a(i+1,j-1) + a(i+1,j+1)) \\
\quad \quad \quad \quad \text{change} = \max(\text{change}, \abs{\text{newa}(i,j)-a(i,j)}) \\
\quad \quad \quad \text{enddo} \\
\quad \quad \text{endo} \\
\quad \text{a}(2:m-1,2:n-1) = \text{newa}(2:m-1,2:n-1) \\
\text{endo}
\]
attributes(global) subroutine jkernel( a, anew & cchange, n, m, w0, w1, w2 )
real, device :: a(m,n), anew(m,n), cchange(*)
integer, value :: n, m
real, value :: w0, w1, w2
real, shared :: aa(18,18), mychange(256)
real :: mynewa, change
integer :: ii, jj, i, j, k, kr
ii = threadIdx%x+1 ; jj = threadIdx%y+1
i = (blockIdx%x-1)*16 + ii
j = (blockIdx%y-1)*16 + jj
aa(ii-1,jj-1) = a(i-1,j-1)
if( ii<=3 ) aa(ii+15,jj) = a(i+15,j-1)
if( jj<=3 ) aa(ii,jj+15) = a(i-1,j+15)
if( ii<=3.and.jj<=3 ) aa(ii+15,jj+15) = a(i+15,j+15)
call syncthreads()
Jacobi Relaxation Main Kernel

\[
\text{mynewa} = w_0 \times aa(ii,jj) + \& \\
\text{w}_1 \times (aa(ii-1,jj) + aa(ii,jj-1) + \& \\
\quad aa(ii+1,jj) + aa(ii,jj+1)) + \& \\
\text{w}_2 \times (aa(ii-1,jj-1) + aa(ii-1,jj+1) + \& \\
\quad aa(ii+1,jj-1) + aa(ii+1,jj+1))
\]

\[
\text{change} = \text{abs}(\text{mynewa} - \text{aa}(ii,jj))
\]

\[
\text{newa}(i,j) = \text{mynewa}
\]

! store change in shared array before reducing

\[
k = \text{threadidx} \times x + (\text{threadidx} \times y - 1) \times 16
\]

\[
\text{mychange}(k) = \text{change}
\]

\[
\text{call syncthreads()}
\]
Jacobi Relaxation Main Kernel

! reduce all 'mychange' values to a single value
kr = 256
do while( kr > 1 )
   kr = kr / 2
   if( k <= kr ) then
      mychange(k) = max(mychange(k), mychange(k+kr))
   call syncthreads()
enddo

kk = (blockidx%y-1) * blockdim%x + blockidx%x
if( k .eq. 1 ) cchange(kk) = mychange(1)
end subroutine
attributes(global) subroutine jreduce( &
     lchange, n )
real, device :: lchange(*)
real, shared :: mychange(256)
integer, value :: n
integer :: k, kk, m
real :: change
k = threadIdx%x
! first, reduce lchange to just 256 values
if( k <= n ) change = lchange(k)
m = k+256
do while( m <= n )
    change = max( change, lchange(m) )
    m = m + 256
enddo
mychange(k) = change
Jacobi Relaxation Reduction Kernel

! reduce all 'mychange' values to a single value
call syncthreads()
kr = 256
do while( kr > 1 )
    kr = kr / 2
    if( k <= kr ) then
        mychange(k) = max(mychange(k),mychange(k+kr))
    call syncthreads()
enddo

if( k .eq. 1 ) cchange(1) = mychange(1)
end subroutine
Sparse Matrix Vector Multiply

- Diagonal Representation
- ELLPack Representation
- Compressed Sparse Row (CSR) representation

```c
for( i = 0; i < nrows; ++i ){
    val = 0.0;
    for( j = 0; j < ncols; ++j ){
        val += A[i*ncols+j] * x[j];
    }
    y[i] = val;
}
```
Diagonal Representation

\[
\begin{array}{ccc}
7 & 5 \\
9 & 6 & 4 & 7 \\
8 & 2 & 0 & 9 \\
0 & 7 & 8 & 6 \\
1 & 7 & 5 & 4 \\
3 & 3 & 3 & 0 \\
4 & 0 & 4 & 2 \\
1 & 1 & 3 \\
\end{array}
\]
# Diagonal Representation

\[
\begin{array}{ccc}
7 & 2 & 5 \\
9 & 6 & 4 & 7 \\
8 & 2 & 0 & 9 \\
0 & 7 & 8 & 6 \\
1 & 7 & 5 & 4 \\
3 & 3 & 3 & 0 \\
4 & 0 & 4 & 2 \\
1 & 1 & 3 \\
\end{array}
\]

\[
\begin{array}{ccc}
7 & 2 & 5 \\
9 & 6 & 4 & 7 \\
8 & 2 & 0 & 9 \\
0 & 7 & 8 & 6 \\
1 & 7 & 5 & 4 \\
3 & 3 & 3 & 0 \\
4 & 0 & 4 & 2 \\
1 & 1 & 3 \\
\end{array}
\]
Diagonal Representation

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

for( i = 0; i < nrows; ++i ){
    val = 0.0;
    for( d = 0; d < nzeros; ++d ){
        j = i + offset[d];
        if( j >= 0 && j < ncols )
            val += data[i*nzeros+d] * x[j];
    }
    y[i] = val;
}
## Diagonal Representation

```
i = blockDim.x*blockIdx.x+threadIdx.x;
if( i < nrows ){
    val = 0.0;
    for( d = 0; d < nzeros; ++d ){
        j = i + offset[d];
        if( j >= 0 && j < ncols )
            val += data[i+nrows*d] * x[j];
    }
    y[i] = val;
}
```

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

-4 -1 0 1 4
www.pgroup.com/lit/samples/sc11.tar
ELLPACK Representation

7  2  4  5
9  6  4  7
  2  9
  7  8  6
  7  5  4
 3  3  3  4
 4  4  2
 1  4  1  3
ELLPACK Representation

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>*</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
ELLPACK Representation

```
for( i = 0; i < nrows; ++i ){
    val = 0.0;
    for( n = 0; n < nzeros; ++n ){
        j = colndx[i*nzeros+n];
        if( j >= 0 && j < ncols )
            val += data[i*nzeros+n] * x[j];
    }
    y[i] = val;
}
```
ELLPACK Representation

\[
i = blockIdx.x*blockDim.x + threadIdx.x;
\]

if( i < nrows ){
    val = 0.0;
    for( n = 0; n < nzeros; ++n ){
        j = colndx[i+nrows*n];
        if( j >= 0 && j < ncols )
            val += data[i+nrows*n] * x[j];
    }
    y[i] = val;
}
CSR Representation

```
7  2  4  5
9  6  4  7
  2     9
  7  8  6
  7  5
  3  3  3  4
  4     2
  1  4  1  3
```
## CSR Representation

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>13</th>
<th>15</th>
<th>19</th>
<th>22</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>j</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>9</td>
<td></td>
<td></td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td></td>
<td>3</td>
<td>4</td>
<td>7</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>j</td>
<td>7</td>
<td>5</td>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>j</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td></td>
<td>2</td>
<td>6</td>
<td>7</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## CSR Representation

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>9</th>
<th>6</th>
<th>4</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

The Portland Group®
CSR Representation

for( i = 0; i < nrows; ++i ){
    val = 0.0;
    nstart = rowindex[i];
    nend = rowindex[i+1];
    for( n = nstart; n<nend; ++n ){
        j = colndx[n];
        val += data[n] * x[j];
    }
    y[i] = val;
}
CSR Representation

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if( i < nrows ){
    val = 0.0;
    nstart = rowindex[i];
    nend = rowindex[i+1];
    for( n = nstart; n<nend; ++n ){
        j = colndx[n];
        val += data[n] * x[j];
    }
    y[i] = val;
}
```
CSR Representation

tid = threadIdx.x;
gid = blockIdx.x*blockDim.x + threadIdx.x;
i = gid / GRPSIZE; // GRPSIZE is power of 2
lane = gid & (GRPSIZE-1);
if( i < nrows ){
    vals[tid] = 0.0; // vals is __shared__
nstart = rowindex[i];
nend = rowindex[i+1];
for( n = nstart+lane; n<nend; n += GRPSIZE )
    vals[tid] += data[n] * x[colndx[n]];
__syncthreads();
for( ln = GRPSIZE>>1; ln > 0; ln >>= 1){
    if( lane < ln ) vals[tid] += vals[tid+ln];
    __syncthreads();
}
if( lane == 0 ) y[i] = vals[tid];
}
Performance Tuning (directives)

- Performance Measurement
- Choose an appropriately parallel algorithm, appropriate data structure
- Optimize data movement between host and GPU
  - frequency, volume, regularity
- Optimize device memory accesses
  - strides, alignment
- Optimize kernel code
  - loop unrolling
  - Optimize compute intensity
    - unroll the parallel loop
change = tolerance + 1.0

!$acc data region local(newa(1:m,1:n)) &
    copy(a(1:m,1:n))

do while(change > tolerance)
    change = 0
    !$acc region
    do i = 2, m-1
        do j = 2, n-1
            newa(i,j) = w0*a(i,j) + &
                w1 * (a(i-1,j) + a(i,j-1) + &
                    a(i+1,j) + a(i,j+1)) + &
                w2 * (a(i-1,j-1) + a(i-1,j+1) + &
                    a(i+1,j-1) + a(i+1,j+1))
            change = max(change,abs(newa(i,j)-a(i,j)))
        enddo
    enddo
    a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
    !$acc end region
enddo

!$acc end data region
#pragma acc data region local(newa[0:n-1][0:m-1])
copy(a[0:n-1][0:m-1])
{
    do{
        change = 0;
        #pragma acc region
        {
            for( i = 1; i < m-1; ++i )
                for( j = 1; j < n-1; ++j ){
                    newa[j][i] = w0*a[j][i] +
                       w1 * (a[j][i-1] + a[j-1][i] +
                            a[j][i+1] + a[j+1][i]) +
                       w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j-1][i+1] + a[j+1][i+1]);
                    change = fmax(change,fabs(newa[j][i]-a[j][i]));
                }
            for( i = 1; i < m-1; ++i )
                for( j = 1; j < n-1; ++j )
                    a[j][i] = newa[j][i];
        }
    }while( change > tolerance ); }
}
% pgfortran -fast -ta=nvidia -Minfo mm.F90

mm1:

6, Generating copyout(a(1:m,1:m))
   Generating copyin(c(1:m,1:m))
   Generating copyin(b(1:m,1:m))
7, Loop is parallelizable
8, Loop is parallelizable
   Accelerator kernel generated
    7, !$acc do parallel, vector(16)
    8, !$acc do parallel, vector(16)
11, Loop carried reuse of 'a' prevents parallelization
12, Loop is parallelizable
   Accelerator kernel generated
    7, !$acc do parallel, vector(16)
    11, !$acc do seq
       Cached references to size [16x16] block of 'b'
       Cached references to size [16x16] block of 'c'
    12, !$acc do parallel, vector(16)
       Using register for 'a'
Performance Profiling

- TAU (Tuning and Analysis Utilities, University of Oregon)
  - collects performance information
- cudaprof (NVIDIA)
  - gives a trace of kernel execution
- `pgfortran -ta=nvidia,time` (on link line)
  - dump of region-level and kernel-level performance
  - upload/download data movement time
  - kernel execution time
- `pgcollect a.out`
- `PGI_ACC_PROFILE` environment variable
  - enables profile data collection for accelerator regions
- `ACC_NOTIFY` environment variable
  - prints one line for each kernel invocation
Performance Profiling

Accelerator Kernel Timing data
f3.f90
smooth

24: region entered 1 time
  time(us): total=1116701 init=1115986 region=715
  kernels=22 data=693
  w/o init: total=715 max=715 min=715 avg=715

27: kernel launched 5 times
  grid: [7x7] block: [16x16]
  time(us): total=17 max=10 min=1 avg=3

34: kernel launched 5 times
  grid: [7x7] block: [16x16]
  time(us): total=5 max=1 min=1 avg=1
Profiling an Accelerator Model Program

$ make
pgf90 -fast -Minfo=ccff ../mm2.f90 ../mmdriv.f90 -o mm2 -ta=nvidia
../mm2.f90:
../mmdriv.f90:
$
$ pgcollect -time ./mm2 < ../in
[...program output...]
target process has terminated, writing profile data
$
$ pgprof -exe ./mm2

- Build as usual
  - Here we add -Minfo=ccff to enhance profile data
- Just invoke with pgcollect
  - Currently collects data similar to -ta=time
- Invoke the PGPROF performance profiler
Dynamic Profile Data Collection

- `acc_enable_time(acc_device_nvidia);`
- `long etime = acc_exec_time(acc_device_nvidia);`
- `long ttime = acc_total_time(acc_device_nvidia);`
- `acc_disable_time(acc_device_nvidia);`

- `unsigned int allocs = acc_allocs();`
- `unsigned int frees = acc_frees();`
- `unsigned int copyins = acc_copyins();`
- `unsigned int copyouts = acc_copyouts();`
- `unsigned long bytesalloc = acc_bytesalloc();`
- `unsigned long bytesin = acc_bytesin();`
- `unsigned long bytesout = acc_bytesout();`
- `unsigned int kernels = acc_kernels();`
- `unsigned int regions = acc_regions();`
- `unsigned long totmem = acc_get_memory();`
- `unsigned long freemem = acc_get_free_memory();`
### Accelerator Model Profiling

**Function** | **Seconds** | **Accelerator Region Time** | **Accelerator Kernel Time**
---|---|---|---
`sstk` | 1 = 43% | | |
`[System_Time]` | 0.5 = 21% | | |
`__libc_pause` | 0.333333 = 14% | | |
`__open_nocancel` | 0.166667 = 7% | | |
`fchmodat` | 0.166667 = 7% | | |
`_int_malloc` | 0.166667 = 7% | | |
`mm` | | 4.122981 =100% | |
`0 (sstk)` | 1 =100% | | |

Profiled: ./mm2 on Tue Sep 21 15:37:52 PDT 2010  Profile: ./pgprof.out
## Accelerator Profiling - Region

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><code>!!$acc region</code></td>
</tr>
<tr>
<td>8</td>
<td><code>do j = 1,m</code></td>
</tr>
<tr>
<td>9</td>
<td><code>do i = 1,m</code></td>
</tr>
<tr>
<td>10</td>
<td><code>a(i,j) = 0.0</code></td>
</tr>
<tr>
<td>11</td>
<td><code>enddo</code></td>
</tr>
<tr>
<td>12</td>
<td><code>do k = 1,m</code></td>
</tr>
<tr>
<td>13</td>
<td><code>do i = 1,m</code></td>
</tr>
<tr>
<td>14</td>
<td><code>a(i,j) = a(i,j) + b(i,k) * c(k,j)</code></td>
</tr>
</tbody>
</table>

### Performance Metrics

- **Accelerator Initialization Time (secs):** 3.937971 = 96%
- **Accelerator Kernels Time (secs):** 0.056836 = 1%
- **Data Transfer Time (secs):** 0.010859 = 0%
- **Accelerator Region Execution Count:** 4
- **Maximum time spent in Accelerator Region w/o Init Time (secs):** 0.132479
- **Minimum time spent in Accelerator Region w/o Init Time (secs):** 0.01746
- **Average time spent in Accelerator Region w/o Init Time (secs):** 0.046252
### Compiler Feedback

#### Line 13

```
13  do j = 1, m
14  do k = 1, m
15  a(i,j) = a(i,j) + b(i,k) * c(k,j)
```

**Seconds**

- 4.122981 = 100%
- 0.000631 = 1%
- 0.056205 = 99%

**Line-level information for line 13**

1. Intensity = 0.67
2. Loop is parallelizable
3. Accelerator kernel generated

**Information about how file ./mm2.f90 was compiled**

**Profiled**

- ./mm2 on Tue Sep 21 15:37:52 PDT 2010
- Profile: ./pgprof.out
Performance Tuning

- Performance Measurement
- Choose an appropriately parallel algorithm
- Optimize data movement between host and GPU
  - frequency, volume, regularity
- Optimize device memory accesses
  - strides, alignment
  - use data cache
- Optimize kernel code
- Optimize compute intensity
- Linux only: pgcudainit
- Windows: must be on the console
- OSX: two GPUs doesn’t mean you can use the idle one
Writing Accelerator Programs

- Step 1: Appropriate algorithm: lots of parallelism
  - Lots of MIMD parallelism to fill the multiprocessors
  - Lots of SIMD parallelism to fill cores on a multiprocessor
  - Lots more MIMD parallelism to fill multithreading parallelism
  - High compute intensity
  - Insert directives, read feedback for parallelism hindrances
    - Dependence relations, pointers
    - Scalars live out from the loop
    - Arrays that need to be private
    - Iterate
Writing Accelerator Programs

- Step 2: Tune data movement between Host and Accelerator
  - read compiler feedback about data moves
  - minimize amount of data moved
  - minimize frequency of data moves
  - minimize noncontiguous data moves
  - optimize data allocation in device memory
  - optimize allocation in host memory (esp. for C)
  - insert local, copyin, copyout clauses
  - use data regions, update clauses
Manually managing memory (CUDA)

- CUDA Fortran device arrays
  - CUDA Fortran device arrays can be used in accelerator regions
  - compile with .cuf suffix or –Mcuda in addition to –ta=nvidia

- CUDA C device arrays
  - no attribute to tell the compiler where a pointer points
  - new in PGI 11.4
    - #pragma acc [data] region deviceptr(ap)
    - tells the compiler that ap was allocated on the GPU
    - ap = acc_malloc( nbytes ); ... acc_free( ap );
    - allocates / frees ap on the current device
    - You may also use data allocated by cudaMalloc or cuMemAlloc
Tuning Accelerator Programs

- Step 3: Tune data movement between device memory and cores
  - minimize frequency of data movement
  - optimize strides – stride-1 in vector dimension
  - optimize alignment – 16-word aligned in vector dimension
  - look at cache feedback from compiler
Tuning Accelerator Programs

- Step 4: Tune kernel code
  - profile the code (cudaprof, pgcollect, -ta=nvidia, time)
  - experiment with kernel schedule using loop directives
    - unroll clauses
  - enable experimental optimizations (-ta=nvidia,O3)
  - single precision vs. double precision
  - 24-bit multiply for indexing (-ta=nvidia,mul24)
  - low precision transcendental (-ta=nvidia,fastmath)
unroll clauses

- Consider each vector space as a ‘tile’ of iterations
  - !$acc do parallel unroll(2)
    - each thread block does two tiles
    - !$acc do parallel unroll(4) does four tiles
  - !$acc do vector(128) unroll(2)
    - tile size is 128 in this dimension
    - use thread block of ½ tile size (64) to do this tile
    - !$acc do vector(128) unroll(4) uses ¼ tile size (32)
  - !$acc do parallel unroll(2) vector(128) unroll(4)
    - strip mines loop to tile size 128
    - each thread block does two tiles
    - thread block size is (32 in this dimension)
- unroll width must be compile-time constant
© Contents copyright 2009-2011, The Portland Group, Inc. This material may not be reproduced in any manner without the expressed written permission of The Portland Group.