GPU Architecture

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

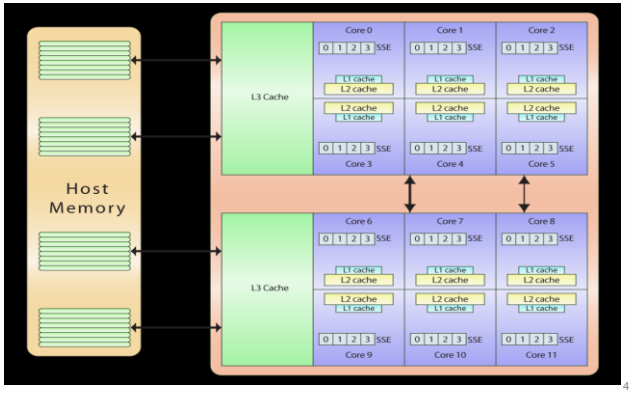
CPU Architecture Features

- Register files (integer, float)
- Functional units (integer, float, address), Icache, Dcache
- Execution pipeline (fetch,decode,issue,execute,cache,commit)
  - branch prediction, hazards (control, data, structural)
  - pipelined functional units, superpipelining, register bypass
  - stall, scoreboarding, reservation stations, register renaming
- 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:
  - Faster clocks
  - More work per clock:
    - superscalar
    - VLIW
    - more cores
    - vector / SIMD instructions
- Memory
  - Latency reduction
  - Latency tolerance

AMD "Magny-Cours"



pthreads parallel vector add

```
for( i = 1; i < ncores; ++i )
  pthread_create( &th[i], NULL, vadd, i );
vadd( 0 );
for( i = 1; i < ncores; ++i )
  pthread_join( th[i], NULL );
...
void vadd( int threadnum ){
 int work = (n+numthreads-1)/numthreads;
 int ifirst = threadnum*work;
 int ilast = min(n,(threadnum+1)*work);
 for( int i = ifirst; i < ilast; ++i )
  a[i]= b[i] + c[i];
}
```

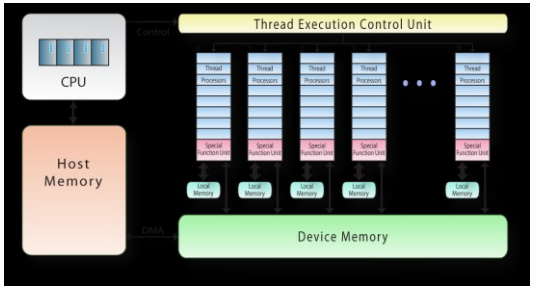

OpenMP parallel vector add

```
#pragma omp parallel do private(i)
for( i = 0; i < n; ++i )
 a[i] = b[i] + c[i];
```

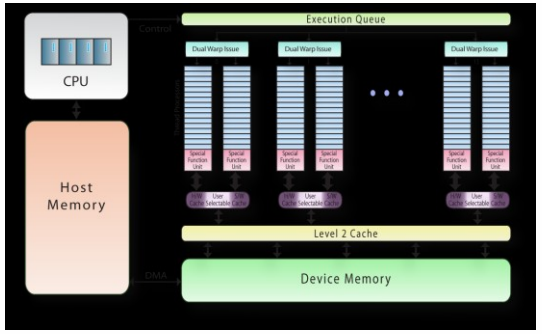

OpenMP Behind the Scenes

- Compiler generates code for N threads:
  - split up the iterations across N threads
- Assumptions
  - uniform memory access costs
  - coherent cache mechanism
- Virtualization penalties
  - load balancing
  - cache locality
  - vectorization within the threads
  - thread management
  - loop scheduling (which thread does what iteration)
  - NUMA memory access penalty

Abstracted x64+Tesla-10 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

Parallel Programming on GPUs

- High degree of regular parallelism
  - lots of scalar threads
  - threads organized into thread groups / blocks
    - SIMD, pseudo-SIMD
  - thread groups organized into grid
    - MIMD
- Languages
  - CUDA, OpenCL, (Brook, Brook+), graphics: OpenGL, DirectX
  - may include vector datatypes (float4, int2)
- Platforms
  - ArBB (Rapidmind, now owned by Intel)

GPU Programming

- Allocate data on the GPU
- Move data from host, or initialize data on GPU
- Launch kernel(s)
  - GPU driver can generate ISA code at runtime
  - preserves forward compatibility without requiring ISA compatibility
- Gather results from GPU
- Deallocate data

Host-side CUDA C Control Code

```
memsize = sizeof(float)*n;
cudaMalloc( &da, memsize );
cudaMalloc( &db, memsize );
cudaMalloc( &dc, memsize );

cudaMemcpy( db, b, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( dc, c, memsize, cudaMemcpyHostToDevice );

dim3 threads( 256 );
dim3 blocks( n/256 );
vaddkernel<<<blocks,threads>>>( da, db, dc, n );

cudaMemcpy( a, da, memsize, cudaMemcpyDeviceToHost );

cudaFree( da );
cudaFree( db );
cudaFree( dc );
```


Device-side CUDA C GPU Code

```
extern "C" __global__ void
vaddkernel( float* a, float* b, float* c, int n )
{
  int ti = threadIdx.x;              /* local index */
  int i = blockIdx.x*blockDim.x+ti;  /* global index */

  a[i] = b[i] + c[i];
}
```


Host-side CUDA C Control Code

```
memsize = sizeof(float)*n;
cudaMalloc( &da, memsize );
cudaMalloc( &db, memsize );
cudaMalloc( &dc, memsize );

cudaMemcpy( db, b, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( dc, c, memsize, cudaMemcpyHostToDevice );

dim3 threads( 256 );
dim3 blocks( (n+255)/256 );
vaddkernel<<<blocks,threads>>>( da, db, dc, n );

cudaMemcpy( a, da, memsize, cudaMemcpyDeviceToHost );

cudaFree( da );
cudaFree( db );
cudaFree( dc );
```


Device-side CUDA C GPU Code

```
extern "C" __global__ void
vaddkernel( float* a, float* b, float* c, int n )
{
  int ti = threadIdx.x;              /* local index */
  int i = blockIdx.x*blockDim.x+ti;  /* global index */

  if( i < n ) a[i] = b[i] + c[i];
}
```

Host-side CUDA C Control Code

```
memsize = sizeof(float)*n;
cudaMalloc( &da, memsize );
cudaMalloc( &db, memsize );
cudaMalloc( &dc, memsize );

cudaMemcpy( db, b, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( dc, c, memsize, cudaMemcpyHostToDevice );

dim3 threads( 256 );
dim3 blocks( min((n+255)/256,65535) );
vaddkernel<<<blocks,threads>>>( da, db, dc, n );

cudaMemcpy( a, da, memsize, cudaMemcpyDeviceToHost );

cudaFree( da );
cudaFree( db );
cudaFree( dc );
```


Device-side CUDA C GPU Code

```
extern "C" __global__ void
vaddkernel( float* a, float* b, float* c, int n )
{
  int ti = threadIdx.x;              /* local index */
  int i = blockIdx.x*blockDim.x+ti;  /* global index */

  for( ; i < n; i += blockDim.x*gridDim.x )
    a[i] = b[i] + c[i];
}
```


CUDA Behind the Scenes

- What you write is what you get
- Implicitly parallel
  - threads into warps
  - warps into thread groups
  - thread groups into a grid
- Hardware thread scheduler
- Highly multithreaded

Appropriate GPU programs

- Characterized by nested parallel loops
- High compute intensity
- Regular data access
- Isolated host/GPU data movement

Device-side OpenCL GPU Code

```
__kernel void
vaddkernel( __global float* a, __global float* b,
            __global float* c, int n )
{
  int i = get_global_id(0);
  for( ; i < n; i += get_global_size(0) )
    a[i] = b[i] + c[i];
}
```


Host-side OpenCL Control Code

```
hContext = clCreateContextFromType( 0, CL_DEVICE_TYPE_GPU, 0,0,0);
clGetContextInfo( hContext, CL_CONTEXT_DEVICES, sizeof(Dev), 0 );
hQueue = clCreateCommandQueue( hContext, Dev[0], 0, 0 );
hProgram = clCreateProgramWithSource( hContext, 1, sProgram, 0, 0 );
clBuildProgram( hprogram, 0, 0, 0, 0, 0 );
hKernel = clCreateKernel( hProgram, "vaddkernel", 0 );

ha = clCreateBuffer( hContext, 0, memsize, 0, 0 );
hb = clCreateBuffer( hContext, 0, memsize, 0, 0 );
hc = clCreateBuffer( hContext, 0, memsize, 0, 0 );

clEnqueueWriteBuffer( hQueue, hb, 0, 0, memsize, b, 0, 0, 0 );
clEnqueueWriteBuffer( hQueue, hc, 0, 0, memsize, c, 0, 0, 0 );

clSetKernelArg( hKernel, 0, sizeof(cl_mem), (void*)&ha );
clSetKernelArg( hKernel, 1, sizeof(cl_mem), (void*)&hb );
clSetKernelArg( hKernel, 2, sizeof(cl_mem), (void*)&hc );
clSetKernelArg( hKernel, 3, sizeof(int), (void*)&n );
clEnqueueNDRangeKernel( hQueue, hKernel, 1, 0, dims, &n, &bsize, 0, 0, 0 );

clEnqueueReadBuffer( hContext, hc, CL_TRUE, 0, memsize, a, 0, 0, 0 );

clReleaseMemObject( hc );
clReleaseMemObject( hb );
clReleaseMemObject( ha );
```


Host-side CUDA Fortran Control Code

```
use vaddmod
real, device, dimension(:), allocatable :: da, db, dc

allocate( da(1:n), db(1:n), dc(1:n) )

db = b
dc = c

call vaddkernel<<<min((n+255)/256,65535),256>>>( da, db, dc, n )

a = da

deallocate( da, db, dc )
```


Device-side CUDA Fortran GPU Code

```
module vaddmod
contains
 attributes(global) subroutine vaddkernel( a, b, c, n )
  real, dimension(*) :: a, b, c
  integer, value :: n
  integer :: i
  i = (blockidx%x-1)*blockdim%x + threadidx%x
  do i = i, n, blockdim%x*griddim%x
   a(i) = b(i) + c(i)
  enddo
 end subroutine
end module
```

Time for a Live Demo

Find PI – Host code

```
use MersenneTwisterMod
use findpimod
implicit none
real, dimension(:), allocatable, device :: x, d
real, dimension(128) :: dd
real :: pi
integer :: nrand
integer :: i, n
nrand = 30000000
n = nrand/2

allocate( x( nrand ), d( 128 ) )
call initrand()
call grand( x, nrand, 777 )
call findpi<<< 128, 128 >>>( x, d, n )
dd = d
pi = 4.0*sum(dd) / float(n)
print *, pi
end program
```


Find PI – Device code part 1

```
module findpimod
contains
 attributes(global) subroutine findpi( x, d, n )
  implicit none
  real, dimension(*) :: x, d
  integer, value :: n
  integer :: i, t, s
  integer, shared :: ss(128)
  s = 0
  t = (blockidx%x-1)*blockdim%x + threadidx%x
  do i = t, n, blockdim%x*griddim%x
   if( x(i)*x(i) + x(i+n)*x(i+n) <= 1.0 ) then
    s = s + 1
   endif
  enddo
  ss(threadidx%x) = s
  call syncthreads()
```


Find PI – Device code part 2

```
 call syncthreads()
  i = 64
  do while( i >= 1 )
   if( threadidx%x <= i )then
    ss(threadidx%x) = ss(threadidx%x) + ss(threadidx%x+i)
   endif
   call syncthreads()
   i = i / 2
  enddo
  if( threadidx%x == 1 )then
   d(blockidx%x) = ss(1)
  endif
 end subroutine
end module
```

GPU Programming with CUDA

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

CUDA C and CUDA Fortran

- Simple introductory program
- Programming model
- Low-level Programming
- Building CUDA programs
- Performance Tuning

- Device management
- Device capabilities
- Declaring data
- Types of GPU memory
- Host pinned memory
- Allocating data
- Copying data
- Launching kernels
- Error detection and management
- Builtin and intrinsic functions
- Testing, timing kernels
- CUDA C & Fortran Interoperability
- Detailed Performance Tuning
- CUDA Runtime vs. Driver API

Host-side CUDA C Control Code

```
memsize = sizeof(float)*n;
cudaMalloc( &da, memsize );
cudaMalloc( &db, memsize );
cudaMalloc( &dc, memsize );

cudaMemcpy( db, b, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( dc, c, memsize, cudaMemcpyHostToDevice );

dim3 threads( 256 );
dim3 blocks( min((n+255)/256,65535) );
vaddkernel<<<blocks,threads>>>( da, db, dc, n );

cudaMemcpy( a, da, memsize, cudaMemcpyDeviceToHost );

cudaFree( da );
cudaFree( db );
cudaFree( dc );
```

Allocate device memory

Copy data to device

Launch a kernel

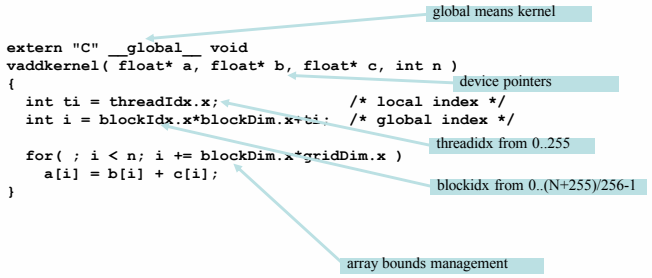
Copy data back from device

Deallocate device memory

CUDA Programming: the GPU

- A scalar program, runs on one thread
  - All threads run the same code
  - Executed in thread groups
  - grid may be 1D or 2D (max 65535x65535)
  - thread block may be 1D, 2D, or 3D
    - max total size 512 (Tesla) or 1024 (Fermi)
  - blockidx gives block index in grid (x,y)
  - threadidx gives thread index within block (x,y,z)
- Kernel runs implicitly in parallel
  - thread blocks scheduled by hardware on any multiprocessor
  - runs to completion before next kernel

Device-side CUDA C GPU Code




```
extern "C" __global__ void
vaddkernel( float* a, float* b, float* c, int n )
{
  int ti = threadIdx.x;                 /* local index */
  int i = blockIdx.x*blockDim.x+ti;     /* global index */

  for( ; i < n; i += blockDim.x*gridDim.x )
    a[i] = b[i] + c[i];
}
```


Declaring Device Data - C

- File static variables / arrays with device attribute are allocated in device memory
  - `__device__ float a[10];`
  - `__device__ int n;`
- Constant attribute for small coefficient arrays
  - `__constant__ float k[10] = { 1.0, 2.0, 2.1, 2.5, 2.9, 3.1, 3.14, 3.141, 3.14159, 3.1415926535 };`
  - stored in constant memory space, 64KB limit
- No device globals (across files), no device locals (to a function)
- Pointers are unattributed, stored on the host
  - `float* b;`

Declaring Device Data - Fortran

- Variables / arrays with device attribute are allocated in device memory
  - `real, device, allocatable :: a(:)`
  - `real, allocatable :: a(:)`
    `attributes(device) :: a`
- In a host subroutine or function
  - device allocatables and automatics may be declared
  - device variables and arrays may be passed to other host subroutines or functions (explicit interface)
  - device variables and arrays may be passed to kernel subroutines

Declaring Device Data - Fortran

- Variables / arrays with device attribute are allocated in device memory
  - ```
    module mm
      real, device, allocatable :: a(:)
      real, device :: x, y(10)
      real, constant :: c1, c2(10)
      integer, device :: n
    contains
      attributes(global) subroutine s( b )
    ...
    ```
- Module data must be fixed size, or allocatable
    ```

Allocating Device Data - C

- replace malloc calls
  - ```
    float *a, *b;
    istat = cudaMalloc( (void**)&a, n*m*sizeof(float) );
    istat = cudaMalloc( (void**)&b, n*sizeof(float) );
    ....
    cudaFree( a ); cudaFree( b );
    ```
- dynamic allocation
  - Allocate is done by the host subprogram
  - Memory is not virtual, you can run out
  - Device memory is shared among users / processes
    - you can have deadlock
  - `istat` return value to catch and test for errors

Allocating Device Data - Fortran

- Fortran allocate / deallocate statement
  - ```
    real, device, allocatable :: a(:,:), b
    allocate( a(1:n,1:m), b )
    ....
    deallocate( a, b )
    ```
- arrays or variables with device attribute are allocated in device memory
  - Allocate is done by the host subprogram
  - Memory is not virtual, you can run out
  - Device memory is shared among users / processes, you can have deadlock
  - `STAT=ivar` clause to catch and test for errors

Copying Data to / from Device - C

- Assignment statements

```
float *a, *b, *b2, *c;
. . .
istat = cudaMalloc( (void**)&a, 10*sizeof(float) );
istat = cudaMemcpy( a, b, 10*sizeof(float),
  cudaMemcpyHostToDevice );
istat = cudaMemcpy( a+2, b2, 2*sizeof(float),
  cudaMemcpyHostToDevice );
...
istat = cudaMemcpy( c, a, 10*sizeof(float),
  cudaMemcpyDeviceToHost );
istat = cudaFree( a );
```

- Data copy to / from host pinned memory will be faster

Copying Data to / from Device - Fortran

- Assignment statements
  - ```
    real, device, allocatable :: a(:,:), b
    allocate( a(1:n,1:m), b )
    a(1:n,1:m) = x(1:n,1:m)      ! copies to device
    b = 99.0
    ....
    x(1:n,1:m) = a(1:n,1:m)      ! copies from device
    y = b
    deallocate( a, b )
    ```
- Data copy may be noncontiguous, but will then be slower (multiple DMAs)
- Data copy to / from host pinned memory will be faster
    ```

Launching Kernels - C

- Function call with chevron syntax for launch configuration
  - `vaddkernel <<< (n+31)/32, 32 >>> ( A, B, C, n );`
  - ```
    dim3 g, b;
    g = dim3((n+31)/32, 1, 1);
    b = dim3( 32, 1, 1 );
    vaddkernel <<< g, b >>> ( A, B, C, n );
    ```


Launching Kernels - Fortran

- Subroutine call with chevron syntax for launch configuration
  - `call vaddkernel <<< (N+31)/32, 32 >>> ( A, B, C, N )`
  - ```
    type(dim3) :: g, b
    g = dim3((N+31)/32, 1, 1)
    b = dim3( 32, 1, 1 )
    call vaddkernel <<< g, b >>> ( A, B, C, N )
    ```
- Interface must be explicit
  - In the same module as the host subprogram
  - In a module that the host subprogram uses
  - Declared in an interface block
- The launch is asynchronous
  - host program continues, may issue other launches

CUDA Errors

- Out of memory
- Launch failure (array out of bounds, ...)
- No device found
- Invalid device code (compute capability mismatch)
- Test for error:

```
ir = cudaGetLastError()
if( ir ) print *, cudaGetErrorString( ir )

ir = cudaGetLastError();
if( ir ) printf( "%s\n", cudaGetErrorString(ir) );
```


Writing a CUDA Kernel (1)

- C: global attribute on the function header, must be void type
  - `__global__ void kernel ( ... ){...}`
- F: global attribute on the subroutine statement
  - `attributes(global) subroutine kernel ( A, B, C, N )`
- May declare scalars, fixed size arrays in local memory
- May declare shared memory arrays
  - `C: __shared__ float sm(16,16);`
  - `F: real, shared :: sm(16,16)`
  - Limited amount of shared memory available (16KB, 48KB)
  - shared among all threads in the same thread block
- Data types allowed
  - int (long,short,char), float, double, struct, union, ...
  - integer(1,2,4,8), logical(1,2,4,8), real(4,8), complex(4,8), derivedtype

Writing a CUDA Kernel (2)

- Predefined variables
  - `blockIdx, threadIdx, gridDim, blockDim, warpSize`
- Executable statements in a kernel
  - assignment
  - for, do, while, if, goto, switch
  - function call to device function
  - intrinsic function call
  - most intrinsics implemented in header files

Writing a CUDA Kernel (3)

- C: Disallowed statements include
  - printf (soon)
  - malloc, new, free, exit
  - recursion, direct or indirect
- F: Disallowed statements include
  - read, write, print, open, close, inquire, format, any IO at all
  - allocate, deallocate, adjustable-sized arrays
  - pointer assignment
  - recursive procedure calls, direct or indirect
  - ENTRY statement, optional arguments, alternate return
  - data initialization, SAVEd data
  - assigned goto, ASSIGN statement
  - stop, pause

Building a CUDA C Program

- nvcc a.cu
  - `.cu` suffix implies CUDA C
- Must use `nvcc` when linking from object files
- Must have appropriate system gcc for preprocessor (Linux, Mac OSX) or CL (Windows)
- flags include:
  - `-ptx` – compile to .ptx portable assembly
  - `-m32` or `–m64` – use 32-bit or 64-bit host + GPU code
  - `-arch compute_13 -code sm_13,sm_20,compute_20`
  - `-maxrregcount 20`
  - `-ftz=true -prec-div=true -prec-sqrt=true`
  - `-use_fast_math (-ftz=true -prec_div=false -prec_sqrt=false)`

Building a CUDA Fortran Program

- pgfortran –Mcuda a.f90
  - `pgfortran -Mcuda[=[emu|cc10|cc11|cc12|cc13|cc20]]`
  - `pgfortran a.cuf`
    - `.cuf` suffix implies CUDA Fortran (free form)
    - `.CUF` suffix runs preprocessor
    - `-Mfixed` for F77-style fixed format
- Must use `–Mcuda` when linking from object files
- Must have appropriate gcc for preprocessor (Linux, Mac OSX)
  - CL, NVCC tools bundled with compiler

Interoperability, C and Fortran

- CUDA Fortran uses the Runtime API
  - `use cudafor` gets interfaces to the runtime API routines
  - CUDA C can use Runtime API (cuda...) or Driver API (cu...)
- CUDA Fortran calling CUDA C kernels
  - explicit interface (interface block), add BIND(C)
  -
```
interface
 attributes(global) subroutine saxpy(a,x,y,n) bind(c)
  real, device :: x(*), y(*)
  real, value :: a
  integer, value :: n
 end subroutine
end interface
call saxpy<<<grid,block>>>( aa, xx, yy, nn )
```


Interoperability

- CUDA C calling CUDA Fortran kernels
  - Runtime API
  - make sure the name is right
    - `module_subroutine_` or `subroutine_`
  - check value vs. reference arguments
  -
```
extern __global__ void saxpy_( float a,
  float* x, float* y, int n );
…
saxpy_<<<grid,block>>>( a, x, y, n );
```
  -
```
attributes(global) subroutine saxpy(a,x,y,n)
  real, value :: a
  real :: x(*), y(*)
  integer, value :: n
```


Performance Tuning

- Performance Measurement
- Choose an appropriately parallel algorithm
- Optimize data movement between host and GPU
  - minimize frequency, volume, irregularity
- Optimize device memory accesses
  - optimize strides, alignment, compute intensity
  - use shared memory, but avoid bank conflicts
  - use constant memory for small coefficient vectors
- Optimize kernel code
  - redundant code elimination
  - loop unrolling
  - Keep the processor busy
    - unroll the parallel or vector “loops”


```
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,tx,ty; real :: Cij
   tx = threadidx%x ; ty = threadidx%y
   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 :: Cij
   real,shared :: Bb(16,16)
   tx = threadidx%x ; ty = threadidx%y
   i=(blockidx%x-1)*16+tx; j=(blockidx%y-1)*16+ty
   Cij = 0.0
   do kb = 0, L-1, 16
    Bb(tx,ty) = B(kb+tx,j)
    call syncthreads()
    do k = 1,16
     Cij = Cij + A(i,kb+k) * Bb(k,ty)
    enddo
    call syncthreads()
   enddo
   C(i,j) = Cij
  end subroutine
```



```
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 :: Cij
   real,shared :: Bb(16,16), Ab(16,16)
   tx = threadidx%x ; ty = threadidx%y
   i=(blockidx%x-1)*16+tx; j=(blockidx%y-1)*16+ty
   Cij = 0.0
   do kb = 0, L-1, 16
    Ab(tx,ty) = A(i,kb+ty)
    Bb(tx,ty) = B(kb+tx,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
```


Time for a Live Demo

CUF Kernels

```
use vaddmod
real, device, dimension(:), allocatable :: da, db, dc

allocate( da(1:n), db(1:n), dc(1:n) )

db = b
dc = c

!$cuf kernel do(1) <<< *, 256 >>>
do i = 1, n
 da(i) = db(i) + dc(i)
enddo

a = da

deallocate( da, db, dc )
```

Find PI using CUF Kernels

```
use MersenneTwisterMod
real, dimension(:), allocatable, device :: x, d
real, dimension(128) :: dd
real :: pi
integer :: nrand, i, n
nrand = 30000000
n = nrand/2

allocate( x( nrand ), d( 128 ) )
call initrand()
call grand( x, nrand, 777 )
s = 0
!$cuf kernel do(1) <<< *, 128 >>>
do i = 1, n
 if( x(i)*x(i) + x(i+n)*x(i+n) <= 1.0 ) then
   s = s + 1
  endif
enddo
pi = 4.0*float(s) / float(n)
```


GPU Programming with the PGI Accelerator Programming Model

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

Accelerator Programming

- Simple introductory program
- Programming model
- Building Accelerator programs
- High-level Programming
- Interpreting Compiler Feedback
- Tips and Techniques
- Performance Tuning

Accelerator vector add

```
#pragma acc region for
 for( i = 0; i < n; ++i )
  a[i] = b[i] + c[i];
```

Why use Accelerator Directives?

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

Basic Syntactic Concepts

- Fortran accelerator directive syntax
  - `!$acc directive [clause]...`
  - `&` continuation
  - Fortran-77 syntax rules
    - `!$acc` or `C$acc` or `*$acc` in columns 1-5
    - continuation with nonblank in column 6
- C accelerator directive syntax
  - `#pragma acc directive [clause]... eol`
  - continue to next line with backslash

Region

- region is single-entry/single-exit region
  - in Fortran, delimited by begin/end directives
  - in C, a single statement, or {...} region
  - no jumps into/out of region, no return
- compute region contains loops to send to GPU
  - loop iterations translated to GPU threads
    - loop indices become threadidx/blockidx indices
- data region encloses compute regions
  - data moved at region boundaries

Appropriate Algorithm

- Nested parallel loops
  - iterations map to threads
  - parallelism means threads are independent
  - nested loops means lots of parallelism
- Regular array indexing
  - allows for stride-1 array access

```
#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 ); }
```


Behind the Scenes

- compiler determines parallelism
- compiler generates thread code
  - split up the iterations into threads, thread groups
  - inserts code to user software data cache
  - accumulate partial sum
  - second kernel to combine final sum
- compiler also inserts data movement
  - compiler or user determines what data to move
  - data moved at boundaries of data/compute region

Building Accelerator Programs

- `pgfortran -ta=nvidia a.f90`
- `pgcc -ta=nvidia a.c`
- Other options:
  - `-ta=nvidia[,cc10|cc11|cc12|cc13|cc20]`
    - default in siterc file:
      - `set COMPUTECAP=13;`
  - `-ta=nvidia[,cuda2.3|cuda3.1|cuda3.2]`
    - default in siterc file:
      - `set DEFCUDAVERSION=3.1;`
  - `-ta=nvidia,time`
  - `-ta=nvidia,host`
- Enable compiler feedback with `-Minfo` or `-Minfo=accel`

Performance Goals

- Data movement between Host and Accelerator
  - minimize amount of data
  - minimize number of data moves
  - minimize frequency of data moves
  - optimize data allocation in device memory
- Parallelism on Accelerator
  - 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

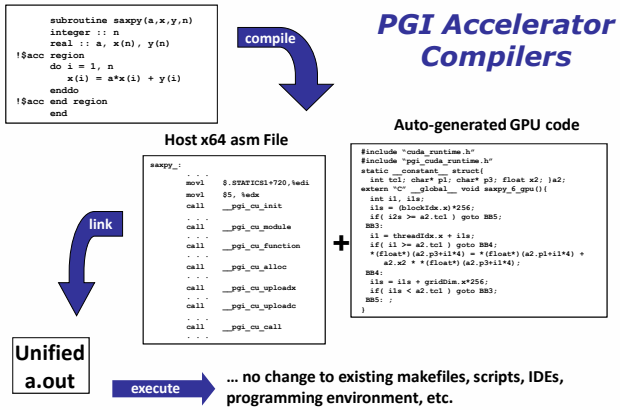
Performance Goals

- 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
  - store array blocks in data cache
- Other goals?
  - minimize register usage?
  - small kernels vs. large kernels?
  - minimize instruction count?
  - minimize synchronization points?

Program Execution Model

- Host
  - executes most of the program
  - allocates accelerator memory
  - initiates data copy from host memory to accelerator
  - sends kernel code to accelerator
  - queues kernels for execution on accelerator
  - waits for kernel completion
  - initiates data copy from accelerator to host memory
  - deallocates accelerator memory
- Accelerator
  - executes kernels, one after another
  - concurrently, may transfer data between host and accelerator

PGI Accelerator Compilers



Data Region

- C

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

- Fortran

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

- May be nested and may contain compute regions
- May not be nested within a compute region

Data Region Clauses

- Data allocation clauses
  - `copy( list )`
  - `copyin( list )`
  - `copyout( list )`
  - `local( list )`
  - data in the lists must be distinct (data in only one list)
  - may not be in a data allocate clause for an enclosing data region
- Data update clauses
  - `updatein( list )` or `update device( list )`
  - `updateout( list )` or `update host ( list )`
  - data must be in a data allocate clause for an enclosing data region

Data Region 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)`

Compute Region

- C

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

- Fortran

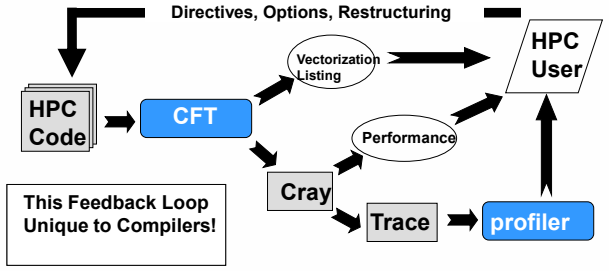
```
!$acc region
    do i = 1,n
        r(i) = a(i) * 2.0
    enddo
!$acc end region
```


Compute Region Clauses

- Data allocation clauses
  - `copy( list )`
  - `copyin( list )`
  - `copyout( list )`
  - `local( list )`
  - data in the lists must be distinct (data in only one list)
- Data update clauses
  - `updatein( list )` or `update device( list )`
  - `updateout( list )` or `update host ( list )`
  - data must be in a data allocate clause for an enclosing data region

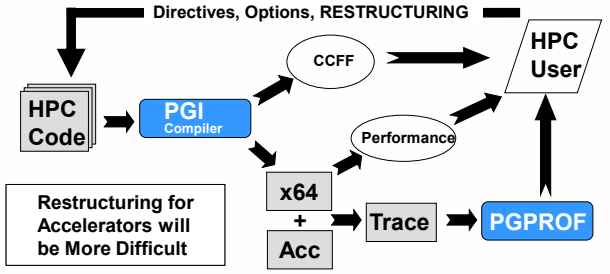
How did we make Vectors Work?

Compiler-to-Programmer Feedback – a classic "Virtuous Cycle"



We can use this same methodology to enable effective migration of applications to Multi-core and Accelerators

Compiler-to-Programmer Feedback



Compiler-to-User Feedback

```
% 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'
```


Loop Schedules

```
27, Accelerator kernel generated
    26, !$acc do parallel, vector(16)
    27, !$acc do parallel, vector(16)
```

- vector loops correspond to `threadidx` indices
- parallel loops correspond to `blockidx` indices
- this schedule has a CUDA schedule
  `<<< dim3(ceil(N/16),ceil(M/16)), dim3(16,16) >>>`
- Compiler strip-mines to protect against very long loop limits

Loop Directive

- C

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

- Fortran

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


Loop Directive Clauses

- Scheduling Clauses
  - `vector` or `vector(n)`
  - `parallel` or `parallel(n)`
  - `seq` or `seq(n)`
  - `host` or `host(n)`
- `independent`
  - use with care, overrides compiler analysis for dependence, private variables
- `private( list )`
  - private data for each iteration of the loop
  - different from local (how?)

PGI Unified Binary

- One binary that executes on GPU or on host
  - default: auto-detect presence of GPU, use GPU if present
  - override default with environment variable `ACC_DEVICE`
  - override within program with call to `acc_set_device`
- Building PGI Unified Binary
  - `pgfortran -ta=nvidia,host -fast program.f90`
- Running PGI Unified Binary
  - `a.out`
  - `setenv ACC_DEVICE host ; a.out`
  - `setenv ACC_DEVICE nvidia ; a.out`
  - before first region

```
call acc_set_device( acc_device_nvidia )
call acc_set_device( acc_device_host )
```


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 ; pgprof -exe a.out pgprof.out`
- `ACC_ACC_PROFILE` environment variable
  - enables profile data collection for accelerator regions

Performance Profiling

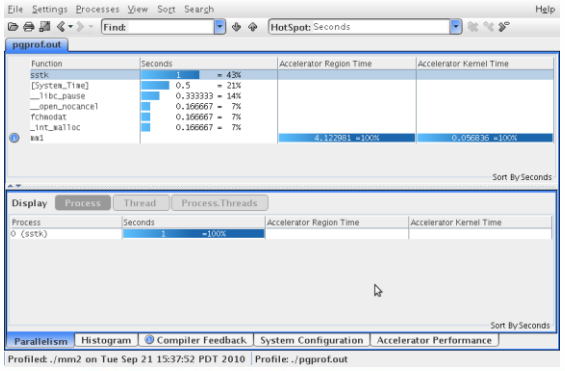
```
Accelerator Kernel Timing data
f3.f90
  smooth
    24: region entered 1 time
        time(us): total=1116701 init=1115986 region=715
                  kernels=22 data=693
        w/o init: total=715 max=715 min=715 avg=715
        27: kernel launched 5 times
            grid: [7x7]  block: [16x16]
            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

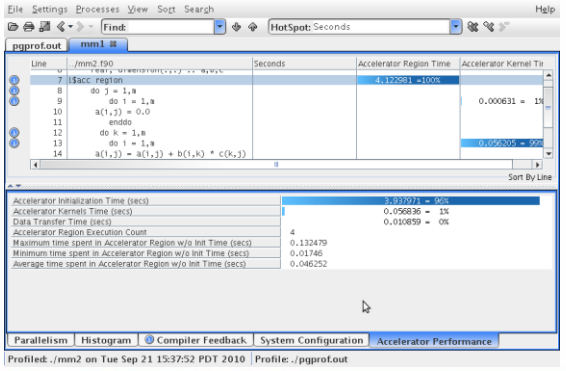
```
$ make
pgf90 -fast -Minfo=ccff ../mm2.f90 ../mmdriv.f90 -o mm2 -ta=nvidia
../mm2.f90:
../mmdriv.f90:
$
$ pgcollect -time ./mm2 < ../in
[...program output...]
target process has terminated, writing profile data
$
$ pgprof -exe ./mm2
```

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

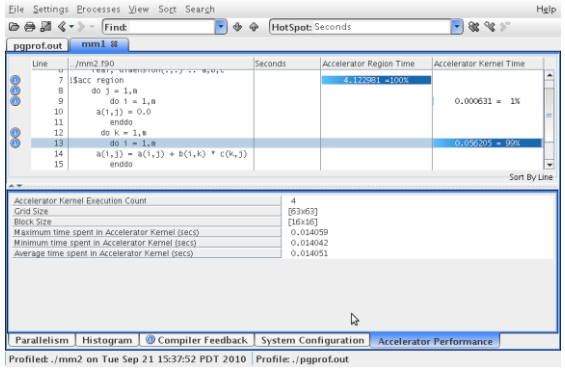
Accelerator Model Profiling



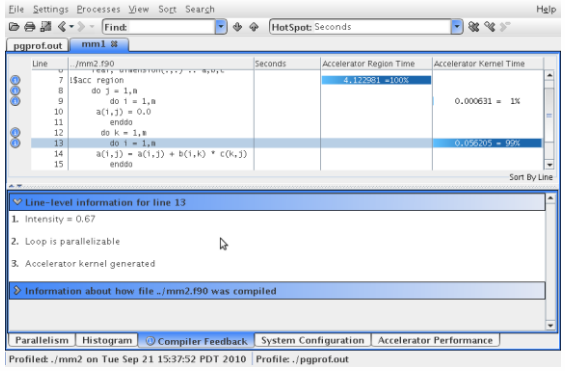
Accelerator Profiling - Region



Accelerator Profiling - Kernel



Compiler Feedback



Longer Term Evolution

- C++
- New targets
  - multicore, Larrabee?, ATI?, other?
- Overlap compute / data movement
  - pipelined loops
- More tools support
- Libraries of device routines
- Programming model evolution
  - multiple GPUs
  - standardization
  - Concurrent kernels, a la OpenCL / Fermi

PGI Accelerator vs CUDA/OpenCL

- **The** PGI Accelerator **programming model is a high-level** *implicit* **programming model for x64+GPU systems, similar to OpenMP for multi-core x64. The PGI Accelerator model:**
  - **Enables offloading of compute-intensive loops and code regions from a host CPU to a GPU accelerator using simple compiler directives**
  - **Implements directives as Fortran comments and C pragmas, so programs can remain 100% standard-compliant and portable**
  - **Makes GPGPU programming and optimization incremental and accessible to application domain experts**
  - **Is supported in both the PGF95 and PGCC C99 compilers**

PGI Accelerator vs CUDA/OpenCL

- CUDA **is a lower-level** explicit **programming model with substantial runtime library components that give expert programmers direct control of:**
  - **Splitting up of source code into host CPU code and GPU compute kernel code in appropriately defined functions and subprograms**
  - **Allocation of page-locked host memory, GPU device main memory, GPU constant memory and GPU shared memory**
  - **All data movement between host main memory and the various types of GPU memory**
  - **Definition of thread/block grids and launching of compute kernels**
  - **Synchronization of threads within a CUDA thread group**
  - **Asynchronous launch of GPU compute kernels, synchronization with host CPU**
- OpenCL **is similar to CUDA, adds some functionality, even lower level**

Availability and Additional Information

- PGI Accelerator Programming Model **– is supported for x64+NVIDIA targets in the PGI Fortran and C compilers, available now**
- Other GPU and Accelerator Targets **– are being studied by PGI, and may be supported in the future as the necessary low-level software infrastructure (e.g. OpenCL) becomes more widely available**
- Further Information **– see www.pgroup.com/accelerate for a detailed specification of the PGI Accelerator model, an FAQ, and related articles and white papers**

Where to get help

- PGI Customer Support – trs@pgroup.com
- PGI User's Forum – http://www.pgroup.com/userforum/index.php
- PGI Articles – http://www.pgroup.com/resources/articles.htm http://www.pgroup.com/resources/accel.htm
- PGI User's Guide – http://www.pgroup.com/doc/pgiug.pdf
- CUDA Fortran Reference Guide – http://www.pgroup.com/doc/pgicudafortug.pdf

Copyright Notice