

## GPU Programming with CUDA (C and PGI CUDA Fortran)

Michael Wolfe  
[Michael.Wolfe@pgroup.com](mailto:Michael.Wolfe@pgroup.com)  
<http://www.pgroup.com>

March 2011

Part 2: CUDA

2-1

The Portland Group®

### VADD on Host

```
subroutine host_vadd(A,B,C,N)
  real(4) :: A(N), B(N), C(N)
  integer :: N, i
  do i = 1,N
    C(i) = A(i) + B(i)
  enddo
end subroutine
```

```
void host_vadd( float* A, float* B, float* C, int n ){
  int i;
  for( i = 0; i < n; ++i ){
    C[i] = A[i] + B[i];
  }
}
```

2-3

The Portland Group®

## CUDA C and CUDA Fortran

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

2-2

The Portland Group®

### CUDA C VADD Device Code

```
__global__ void vaddkernel( float* A, float* B,
                           float* C, int n ){
  int i;
  i = blockIdx.x*blockDim.x + threadIdx.x;
  if( i <= N ) C[i] = A[i] + B[i];
}
```

2-4

The Portland Group®

## CUDA Fortran VADD Device Code

```

module kmod
  use cudafor
  contains
  attributes(global) subroutine vaddkernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockid%x-1)*blockdim%x + threadid%x
    if( i <= N ) C(i) = A(i) + B(i)
  end subroutine
end module

```

2-5

The Portland Group®

## CUDA C VADD Host Code

```

void vadd( float* A, float* B, float* C, int n ){
  float *Ad, *Bd, *Cd;
  cudaMalloc( (void**)&Ad, n*sizeof(float) );
  cudaMalloc( (void**)&Bd, n*sizeof(float) );
  cudaMalloc( (void**)&Cd, n*sizeof(float) );
  cudaMemcpy( Ad, A, n*sizeof(float),
             cudaMemcpyHostToDevice );
  cudaMemcpy( Bd, B, n*sizeof(float),
             cudaMemcpyHostToDevice );
  vaddkernel<<< (n+31)/32, 32 >>>( Ad, Bd, Cd, n );
  cudaMemcpy( C, Cd, n*sizeof(float),
             cudaMemcpyDeviceToHost );
  cudaFree( Ad ); cudaFree( Bd ); cudaFree( Cd );
}

```

2-6

The Portland Group®

## CUDA Fortran VADD Host Code

```

subroutine vadd( A, B, C )
  use kmod
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable, dimension(:):: &
    Ad, Bd, Cd
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine

```

2-7

The Portland Group®

## CUDA Programming

### □ Host code

- Optional: select a GPU
- Allocate device memory
- Copy data to device memory
- Launch kernel(s)
- Copy data from device memory
- Deallocate device memory

### □ Device code

- Scalar thread code, limited operations
- Implicitly parallel
  - threads within a thread block, essentially SIMD
  - thread blocks within a grid, MIMD parallelism

2-8

The Portland Group®

### Elements of CUDA C - Host

```

void vadd( float* A, float* B, float* C )
{
    float *Ad, *Bd, *Cd;
    cudaMalloc( (void**)&Ad, n*sizeof(float) );
    cudaMalloc( (void**)&Bd, n*sizeof(float) );
    cudaMalloc( (void**)&Cd, n*sizeof(float) );
    cudaMemcpy( Ad, A, n*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( Bd, B, n*sizeof(float), cudaMemcpyHostToDevice );
    vaddkernel<<< (n+31)/32, 32 >>>( Ad, Bd, Cd, n );
    cudaMemcpy( C, Cd, n*sizeof(float), cudaMemcpyDeviceToHost );
    cudaFree( Ad ); cudaFree( Bd ); cudaFree( Cd );
}
    
```

2-9



### Elements of CUDA Fortran - Host

```

subroutine vadd( A, B, C )
    use kmod
    real(4), dimension(:) :: A, B, C
    real(4), device, allocatable, dimension(:):: &
        Ad, Bd, Cd
    integer :: N
    N = size( A, 1 )
    allocate( Ad(N), Bd(N), Cd(N) )
    Ad = A(1:N)
    Bd = B(1:N)
    call vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )
    C(1:N) = Cd
    deallocate( Ad, Bd, Cd )
end subroutine
    
```

2-10



### 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)
    - 3D in CUDA 4.0 (Fermi)
  - 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

2-11



### Elements of CUDA C - Kernel

```

__global__ void vaddkernel( float* A, float* B,
                           float* C, int n ) {
    int i;
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= N ) C[i] = A[i] + B[i];
}
    
```

2-12



## Elements of CUDA Fortran - Kernel

```

module kmod
  use cudafor
contains
  attributes(global) subroutine vaddkernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadidx%x
    if( i <= N ) C(i) = A(i) + B(i)
  end subroutine
end module
    
```

Annotations:

- global means kernel (points to `attributes(global)`)
- device attribute implied (points to `device ::`)
- value vs. Fortran default (points to `value ::`)
- blockidx from 1..(N+31)/32 (points to `blockidx%x`)
- threadidx from 1..32 (points to `threadidx%x`)
- array bounds test (points to `if( i <= N )`)

2-13



## Time for a Live Demo (2) CUDA Vector Add

2-14



## CUDA Elements

- Host code
  - Declaring and allocating device memory
  - Moving data to and from device memory
  - Pinned memory
  - Launching kernels
- Kernel code
  - `__global__`, `__device__` (C), Attributes clause (Fortran)
  - Kernel routines, device routines
  - Shared memory
  - What is and what is not allowed in a kernel
  - CUDA Runtime API
  - CUDA Driver API (C only)

2-15



## 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, no device locals
- Pointers are unattributed, stored on the host
  - `float* b;`

2-16



## 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

2-17



## 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

2-18



## Declaring Device Data - Fortran

- Data declared in a Fortran module
  - Device variables, arrays, allocatables allowed
  - Device variables, arrays, allocatables are accessible to device subprograms within that module
  - Also accessible to host subprograms in that module or which use that module
  - Constant attribute (not to be confused with parameter) puts variable or array in constant memory space (64KB limit)

2-19



## Allocating Device Data - C

- replace malloc calls
  - `float *a, *b;`  
`istat = cudaMalloc( (void*)&a, n*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

2-20



## 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

2-21



## 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

2-22



## 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

2-23



## Using the API - Fortran

```
use cudafor
real, allocatable, device :: a(:)
real :: b(10), b2(2), c(10)
...
istat = cudaMalloc( a, 10 )
istat = cudaMemcpy( a, b, 10 )
istat = cudaMemcpy( a(2), b2, 2 )

istat = cudaMemcpy( c, a, 10 )
istat = cudaFree( a )
```

2-24



## Host Pinned Memory - C

### allocate pinned memory

```

float *x, *a;
istat = cudaHostAlloc( (void**)&a, sizeof(float)*n );
istat = cudaMalloc( (void**)&x, sizeof(float) * n );
...
istat = cudaMemcpy( x, a, sizeof(float)*n,
    cudaMemcpyHostToDevice );
....

cudaHostFree( a );  cudaFree( x );
    
```

### Downsides

- Limited amount of pinned memory on the host
- May reduce performance of this or other applications

2-25



## Host Pinned Memory - Fortran

### Pinned attribute for host data

```

real, pinned, allocatable :: x(:, :)
real, device, allocatable :: a(:, :)
allocate( a(1:n,1:m), x(1:n,1:m) )
...
a(1:n,1:m) = x(1:n,1:m)      ! copies to device
....
x(1:n,1:m) = a(1:n,1:m)      ! copies from device
deallocate( a, b )
    
```

### Downsides

- Limited amount of pinned memory on the host
- May not succeed in getting pinned memory

2-26



## Behind the Scenes - Fortran

### allocates/deallocates turn into cudaMalloc and cudaFree calls

### Assignments turn into cudaMemcpy calls

- Non-contiguous data may be many many calls
- Scalar assignments as well!

### Declarations of device data become

- Static data is static on host + GPU
- Non-static data is dynamically allocated on GPU

2-27



## Time for a Live Demo (3)

### CUDA Vector Add

Pinned vs Non-pinned Data Bandwidth

2-28



## 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 );

```

2-29

The Portland Group®

## 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

2-30

The Portland Group®

## Launching Kernels

### 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 )

```

### launch configuration

- <<< grid, block >>>
- grid, block may be scalar integer expression, or struct dim3 (C) or type(dim3) (Fortran) variable

### The launch is asynchronous

- host program continues, may issue other launches

2-31

The Portland Group®

## 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) );

```

2-32

The Portland Group®

## Time for a Live Demo (4)

### CUDA Vector Add

### Timing a Kernel

2-33

The Portland Group®

## Writing a CUDA C Kernel (1)

- global attribute on the function header, must be void type
  - `__global__ void kernel ( ... ){ ... }`
- May declare scalars, fixed size arrays in local memory
- May declare shared memory arrays
  - `__shared__ float 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, ...

2-34

The Portland Group®

## Writing a CUDA C 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

2-35

The Portland Group®

## Writing a CUDA C Kernel (3)

- Disallowed statements include
  - `printf` (some now)
  - `malloc`, `new`, `free` (coming in 4.0)
  - `exit`
  - recursion, direct or indirect

2-36

The Portland Group®

## Writing a CUDA Fortran Kernel (1)

- 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
  - `real, shared :: sm(16,16)`
  - Limited amount of shared memory available
  - shared among all threads in the same thread block
- Data types allowed
  - `integer(1,2,4,8), logical(1,2,4,8), real(4,8), complex(4,8), character(len=1)`
  - Derived types

2-37



## Writing a CUDA Fortran Kernel (2)

- Predefined variables
  - `blockidx, threadidx, griddim, blockdim, warpsize`
- Executable statements in a kernel
  - assignment
  - `do, if, goto, case`
  - intrinsic function call, device subprogram call
  - `where, forall`

2-38



## Writing a CUDA Fortran Kernel (3)

- 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`

2-39



## Supported Intrinsic Functions

### □ Fortran numeric intrinsics

<code>abs</code>	<code>aimag</code>	<code>aint</code>	<code>anint</code>
<code>ceiling</code>	<code>cmplx</code>	<code>conjg</code>	<code>dim</code>
<code>floor</code>	<code>int</code>	<code>logical</code>	<code>max</code>
<code>min</code>	<code>mod</code>	<code>modulo</code>	<code>nint</code>
<code>real</code>	<code>sign</code>		

### □ Fortran mathematical intrinsics

<code>acos</code>	<code>asin</code>	<code>atan</code>	<code>atan2</code>
<code>cos</code>	<code>cosh</code>	<code>exp</code>	<code>log</code>
<code>log10</code>	<code>sin</code>	<code>sinh</code>	<code>sqrt</code>
<code>tan</code>	<code>tanh</code>		

2-40



## Supported Device Functions

□ use cudadevice

__mulhi	__umulhi	__mul64hi	__umul64hi
__int_as_float	__float_as_int	__saturate[f]	__sad
__usad	__fdividef	fdivide[f]	__sinf
__cosf	__tanf	__expf	__exp10f
__log2f	__log10f	__logf	__powf
__float2int_r*	__float2uint_r*	__int2float_r*	__uint2float_r*
__float2ll_r*	__float2ull_r*	__ll2float_r*	__ull2float_r*
__float2half_rn	__half2float	__double2int_r*	__double2uint_r*
__int2double_r*	__uint2double_r*	__double2ll_r*	__double2ull_r*
__ll2double_r*	__ull2double_r*	__fadd_r*	__fmul_r*
__fmaf_r*	__frcp_r*	__fsqrt_r*	__fdiv_r*
__dadd_r*	__dmul_r*	__dfma_r*	__dsqrt_r*
__drcp_r*	__ddiv_r*	__clz[ll]	__ffs[ll]
__popc[ll]	__bref[ll]		

2-41

## Supported libm Functions

□ use libm

acosh[f]	asinh[f]	asinf	acosf
atan2f	cbrt[f]	ceil[f]	copysign[f]
cosf	coshf	erf[f]	erfc[f]
expm1[f]	expf	exp10[f]	fabs[f]
floor[f]	fma[f]	fmin[f]	frexp[f]
ilogb[f]	ldexp[f]	lgamma[f]	llrint[f]
lrint[f]	llround[f]	lround[f]	logb[f]
log10f	logf	log1p[f]	log2[f]
modf[f]	nearbyint[f]	nextafter[f]	pow[f]
remainder[f]	rint[f]	scalbn[f]	scalbln[f]
sinf	sinhf	sqrtf	tanf
tanhf	tgamma[f]	trunc[f]	

2-42

## New Intrinsic Procedures

- `syncthreads()` or `__syncthreads()`
  - synchronizes all threads in the same thread block at a barrier
  - must reach the same lexical call
- `gpu_time(clock)`
  - returns clock cycle counter
- warp vote functions
  - `allthreads(a(i)<0.0)`, `anythread(a(i)<0.0)`
- atomic functions, int or integer(4) mostly, some float, long
  - = `atomicadd( a[j], 1 )`, `atomicsub`, etc.
  - = `atomicinc( a[j] )`, `atomicdec`
  - = `atomiccas( a[i], compare, val )`

2-43

## Modules and Scoping

- `attributes(global)` subroutine kernel in a module
  - can directly access device data in the same or another module
  - can call device subroutines / functions in the same module
- `attributes(device)` subroutine / function in a module
  - can directly access device data in the same or another module
  - can call device subroutines / functions in the same module
  - implicitly private to the module
- `attributes(global)` subroutine kernel outside of a module
  - can directly access device data in any module
- host subprograms
  - can call any kernel in any module or outside module
  - can access module data in any module
  - can call CUDA C kernels as well (explicit interface)

2-44

## 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:
  - `-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`)

2-45

## 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

2-46

## Time for a Live Demo (5)

### CUDA Vector Add

### Using CUDA Fortran Emulation Mode

2-47

## CUDA Runtime API Routines

- ❑ Device management
- ❑ Thread management
- ❑ Memory management
- ❑ Event management
- ❑ Fortran: use `cudafor`

2-48

## Device Management

```
❑ int cudaGetDeviceCount( int icount );  
  
❑ int cudaSetDevice( int inum );  
  
❑ int cudaGetDevice( int inum );
```

2-49

## Device Management

```
❑ integer cudaGetDeviceCount( icount )  
    integer, intent(out) :: icount  
  
❑ integer cudaSetDevice( inum )  
    integer, intent(in) :: inum  
  
❑ integer cudaGetDevice( inum )  
    integer, intent(out) inum
```

2-50

## Device Management

```
❑ int cudaGetDeviceProperties(  
    struct cudaDeviceProp* prop, int inum );  
  
❑ int cudaChooseDevice(  
    int* inum, struct cudaDeviceProp* prop );
```

2-51

## Device Management

```
❑ prop.name [char[256]]  
❑ prop.major  
❑ prop.minor  
❑ prop.totalGlobalMem  
❑ prop.regsPerBlock  
❑ prop.maxThreadsPerBlock  
❑ prop.maxThreadDim(3)  
❑ prop.maxGridSize(3)  
❑ prop.clockRate  
❑ prop.totalConstMem
```

2-52

## Device Management

- ❑ `prop.warpSize`
- ❑ `prop.textureAlignment`
- ❑ `prop.deviceOverlap`
- ❑ `prop.multiProcessorCount`
- ❑ `prop.kernelExecTimeoutEnabled`
- ❑ `prop.memPitch`
- ❑ `prop.integrated`
- ❑ `prop.canMapHostMemory`
- ❑ `prop.computeMode`

2-53

## Device Management

- ❑ `integer cudaGetDeviceProperties(prop, inum)`  
`type(cudaDeviceProp), intent(out) :: prop`  
`integer, intent(in) :: inum`
- ❑ `integer cudaChooseDevice( inum, prop )`  
`type(cudaDeviceProp), intent(in) :: prop`  
`integer, intent(out) :: inum`

2-54

## Device Management

- ❑ `prop%name [character]`
- ❑ `prop%major`
- ❑ `prop%minor`
- ❑ `prop%totalGlobalMem`
- ❑ `prop%regsPerBlock`
- ❑ `prop%maxThreadsPerBlock`
- ❑ `prop%maxThreadDim (3)`
- ❑ `prop%maxGridSize (3)`
- ❑ `prop%clockRate`
- ❑ `prop%totalConstMem`

2-55

## Device Management

- ❑ `prop%warpSize`
- ❑ `prop%textureAlignment`
- ❑ `prop%deviceOverlap`
- ❑ `prop%multiProcessorCount`
- ❑ `prop%kernelExecTimeoutEnabled`
- ❑ `prop%memPitch`
- ❑ `prop%integrated`
- ❑ `prop%canMapHostMemory`
- ❑ `prop%computeMode`

2-56

## Thread Management

- ❑ `integer cudaThreadSynchronize();`
- ❑ `integer cudaThreadExit();`
- ❑ `int cudaThreadSynchronize();`
- ❑ `int cudaThreadExit();`

2-57

## Stream Management

- ❑ `int cudaStreamCreate( int* stream );`
- ❑ `int cudaStreamQuery( int stream );`
- ❑ `int cudaStreamSynchronize( int stream );`
- ❑ `int cudaStreamDestroy( int stream );`

2-58

## Stream Management

- ❑ `integer cudaStreamCreate(stream)`  
`integer,intent(out) :: stream`
- ❑ `integer cudaStreamQuery(stream)`  
`integer,intent(in) :: stream`
- ❑ `integer cudaStreamSynchronize(stream)`  
`integer,intent(in) :: stream`
- ❑ `integer cudaStreamDestroy(stream)`  
`integer,intent(in) :: stream`

2-59

## Memory Management

- ❑ `int cudaMallocHost( void** ptr, size_t size );`
- ❑ `int cudaFreeHost( void* ptr );`
- ❑ `int cudaMalloc( void** ptr, size_t count );`
- ❑ `int cudaFree( void* ptr );`

2-60

## Memory Management

```
int cudaMemcpy( void* dst, void* src,
               size_t count, enum dir );
```

```
dir values:
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
```

2-61

The Portland Group®

## Memory Management

```
int cudaMallocPitch(
    void* ptr, size_t* pitch, size_t w, size_t h);
```

```
int cudaMemset( void* ptr, int val, size_t count);
```

```
int cudaMemset2D( void* ptr, size_t pitch,
                  int value, size_t w, size_t h );
```

2-62

The Portland Group®

## Memory Management

```
int cudaMemcpyToSymbol(
    char* symbol, void* src, size_t count,
    size_t offset, enum dir );
```

```
int cudaMemcpyFromSymbol(
    void* dst, char* symbol, size_t count,
    size_t offset, enum dir );
```

```
int cudaGetSymbolAddress(
    void** ptr, char* symbol );
```

```
int cudaGetSymbolSize(
    size_t* size, char* symbol );
```

2-63

The Portland Group®

## Memory Management

```
integer cudaMallocHost( ptr, size )
type(c_ptr),intent(out) :: ptr
integer,intent(in) :: size
```

```
integer cudaFreeHost( ptr )
type(c_ptr),intent(in) :: ptr
```

```
integer cudaMalloc( ptr, count )
<type>,device,allocatable,dimension(:)::&
ptr
integer,intent(in) :: count
```

```
integer cudaFree( ptr )
<type>,device,dimension(*) :: ptr
```

2-64

The Portland Group®

## Memory Management

- `integer cudaMemcpy(dst,src,count,dir)`  
`<type>,<device><,<dimension> :: dst,src`  
`integer, intent(in) :: count, dir`
- dir values:  
`cudaMemcpyHostToHost`  
`cudaMemcpyHostToDevice`  
`cudaMemcpyDeviceToHost`  
`cudaMemcpyDeviceToDevice`

2-65

The Portland Group®

## Memory Management

- `integer cudaMallocPitch(ptr, pitch, w, h)`  
`<type>,device,allocatable,dimension(:,:)&`  
`:: ptr`  
`integer,intent(out) :: pitch`  
`integer,intent(in) :: w,h`
- `integer cudaMemcpy(ptr, value, count)`  
`<type>,device,<,<dimension(*)> :: ptr`  
`<type> :: value`  
`integer :: count`
- `integer cudaMemcpy2D(ptr,pitch,value,w,h)`  
`<type>,device,<,<dimension> :: ptr`

2-66

The Portland Group®

## Memory Management

- `integer cudaMemcpyToSymbol &`  
`( symbol, src, count, offset, dir )`  
`<type> :: symbol`  
`<type> :: src`  
`integer,intent(in) :: count,offset,dir`
- `integer cudaMemcpyFromSymbol &`  
`( dst, symbol, count, offset, dir )`
- `integer cudaGetSymbolAddress(ptr,symbol)`  
`type(c_devptr) :: ptr`
- `integer cudaGetSymbolSize( size, symbol )`  
`integer, intent(out) :: size`

2-67

The Portland Group®

## Naming CUDA Fortran symbols

- module global symbols
  - `modulename_16`, common block for device symbols
  - `modulename_17`, common block for constant symbols

2-68

The Portland Group®

## Event Management

```

❑ int cudaEventCreate( cudaEvent_t* event );

❑ int cudaEventRecord
  ( cudaEvent_t event, cudaStream_t stream );

❑ int cudaEventSynchronize( cudaEvent_t event );

❑ int cudaEventElapsedTime
  ( float* time, cudaEvent_t e1, cudaEvent_t e2 );

❑ int cudaEventDestroy( cudaEvent_t event );

```

2-69

## Event Management

```

❑ integer cudaEventCreate( event )
  type(cudaEvent), intent(out) :: event

❑ integer cudaEventRecord( event, stream )
  type(cudaEvent), intent(in) :: event
  integer, intent(in) :: stream

❑ integer cudaEventSynchronize( event )
  type(cudaEvent), intent(in) :: event

❑ integer cudaEventElapsedTime( time,e1,e2 )
  real, intent(out) :: time
  type(cudaEvent), intent(in) :: e1, e2

❑ integer cudaEventDestroy( event )
  type(cudaEvent), intent(in) :: event

```

2-70

## Error Handling

```

❑ integer cudaGetLastError()

❑ character(128) cudaGetErrorString( err )
  integer err

❑ cudaError_t cudaGetLastError();

❑ char* cudaGetErrorString( cudaError_t err );

```

2-71

## Time for a Live Demo (6)

### CUDA Device Query

### Error Handling

### cudaSetDevice

2-72

## CUDA C vs CUDA Fortran

### □ CUDA C

- supports texture memory
- supports Runtime API
- supports Driver API
- cudaMalloc, cudaFree
- cudaMemcpy
- OpenGL interoperability
- Direct3D interoperability
- arrays zero-based
- threadIdx/blockIdx 0-based
- unbound pointers
- pinned allocate routines

### □ CUDA Fortran

- no texture memory (may come)
- supports Runtime API
- no support for Driver API
- allocate, deallocate
- assignments
- no OpenGL interoperability
- no Direct3D interoperability
- arrays one-based
- threadIdx/blockIdx 1-based
- allocatable are device/host
- pinned attribute

2-73



## 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 )
```

2-74



## 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
```

2-75



## Interoperability

### □ CUDA Fortran kernels can be linked with nvcc

- The kernels look to nvcc just like CUDA C kernels

### □ CUDA C kernels can be linked with pgfortran

- remember -Mcuda flag when linking object files
- PGI 10.9 CUDA Fortran release uses CUDA 2.3 by default
- PGI 11.x CUDA Fortran uses CUDA 3.1 by default
- CUDA 3.1 may be set as default or -Mcuda=cuda3.1
  - set DEFCUDAVERSION=3.1; # in siterc or .mypgirc

2-76



## Time for a Live Demo (7)

### CUDA Vector Add

CUDA C and CUDA Fortran

2-77

The Portland Group®

## VADD on Host

```
subroutine host_vadd(A,B,C,N)
  real(4) :: A(N), B(N), C(N)
  integer :: N, i
  do i = 1,N
    C(i) = A(i) + B(i)
  enddo
end subroutine
```

2-78

The Portland Group®

## CUDA Fortran VADD Host Code

```
subroutine vadd( A, B, C )
  use kmod
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable, dimension(:):: &
    Ad, Bd, Cd
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
```

2-79

The Portland Group®

## CUDA Fortran VADD Device Code

```
module kmod
  use cudafor
  contains
  attributes(global) subroutine vaddkernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*blockdim%x + threadidx%x
    if( i <= N ) C(i) = A(i) + B(i)
  end subroutine
end module
```

2-80

The Portland Group®

## CUF Kernel VADD Host Code

```

subroutine vadd( A, B, C )
  use kmod
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable, dimension(:):: &
    Ad, Bd, Cd
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  !$cuf kernel do(1) <<< (N+31)/32, 32>>>
  do i = 1, n
    Cd(i) = Ad(i) + Bd(i)
  enddo
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine

```

2-81

## CUF Kernels

- !\$cuf kernel
- do(loop depth)
  - do by itself defaults to do(1)
- <<< blocks, threads >>>
  - must have one entry for each loop, innermost first
  - blocks or threads may be \*
- body of the loop(s) become the body of the kernel
  - enable -Minfo to see compiler messages
- Must immediately precede tightly nested loop of given depth
- All arrays must be device arrays; scalars are auto-privatized

2-82

## Time for a Live Demo (7.1)

### CUF Kernels

2-83

## 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

2-84

```

__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;
}

```

2-85



```

__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
}

```

2-86



```

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;
}

```

2-87



```

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 );
}

```

2-88



```

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 );
}

```

2-89

## 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

2-90

```

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

```

2-91

```

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

```

2-92

```

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
end module

```

2-93

```

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

```

2-94

```

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

```

2-95

Time for a Live Demo (8)  
 CUDA C vs CUDA Fortran MATMUL  
 Using cudaprof

2-96

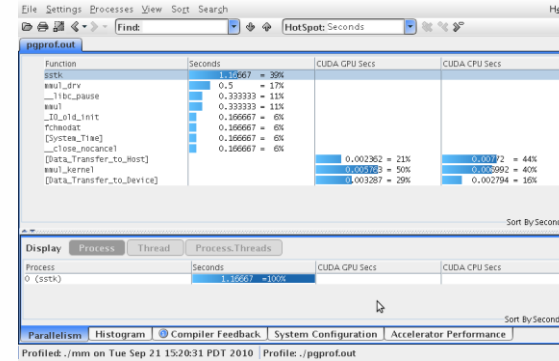
## Profiling CUDA Fortran

```
$ make
pgfortran -Minfo=ccff -Mcuda -c ../mm.cuf□
pgfortran -Minfo=ccff -Mcuda -c ../mmdrv.f90□
pgfortran -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
```

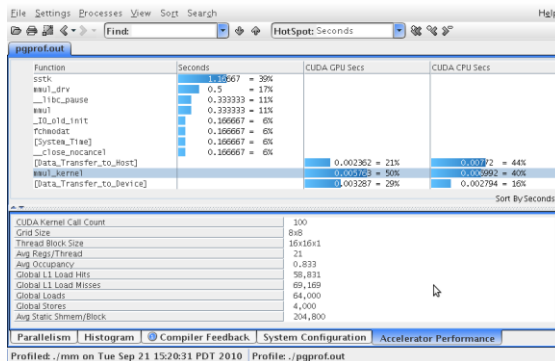
- Uses the same GPU counters as cudaprof
  - Relates to host performance and source code
- Build as usual
  - Here we added -Minfo=ccff to enhance profile data
- Run pgcollect
  - collect global memory data (gmem) on Fermi (cc20)
- Then invoke the PGPROF performance profiler



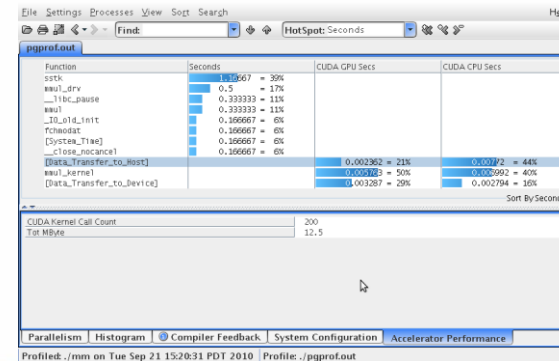
## CUDA Fortran Profile



## CUDA Fortran Profile - Kernel



## CUDA Fortran Profile - Data



## 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 shared memory, avoid bank conflicts
  - use constant memory
- ❑ Optimize kernel code
  - redundant code elimination
  - loop unrolling
  - Optimize compute intensity
    - unroll the parallel loop

2-101



## 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

2-102



## 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

2-103



## Execution Configuration

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

2-104



## 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

2-105

The Portland Group®

## Divergence

### Pad arrays to multiples of block size

```

▪ i = (blockidx%x-1)*64 + threadidx%x
  if( i <= N ) A(i) = ...

```

2-106

The Portland Group®

## Global Memory

### Stride-1, aligned accesses

- address is aligned to  $\text{mod}(\text{threadidx}\%x, 16)$
- $\text{threadidx}\%x$  and  $\text{threadidx}\%x+1$  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

2-107

The Portland Group®

## 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

```

2-108

The Portland Group®

## Redundant access within a GPU Thread

```

real, shared :: BB(256)
...
do ii = 0, 255, 64
  BB(threadidx%x+ii) = B(i1o+ii)
enddo
call syncthread()
do j = j1o, jhi
  do i = i1o+threadidx%x, ihi, 64
    A(i,j) = A(i,j) * BB(i-i1o)
  enddo
enddo

```

2-109

The Portland Group®

## Redundant access across GPU Threads

```

! threadidx%x from 1:64
i = (blockidx%x-1)*64 + threadidx%x
...
do j = j1o, jhi
  A(i,j) = A(i,j) * B(j)
enddo

```

2-110

The Portland Group®

## Redundant access across GPU Threads

```

real, shared :: BB(64)

i = (blockidx%x-1)*64 + threadidx%x
...
do jb = j1o, jhi, 64
  BB(threadidx%x) = B(jb+threadidx%x)
  call syncthread()
  do j = jb, min(jhi, jb+63)
    A(i,j) = A(i,j) * BB(j-jb+1)
  enddo
enddo

```

2-111

The Portland Group®

## Stride-1 Access within a GPU thread

```

! threadidx%x from 1:32
i = (blockidx%x-1)*32 + threadidx%x
...
ix = indx(i)
do j = j1o, jhi
  A(i,j) = A(i,j) * B(ix+j)
enddo

```

2-112

The Portland Group®

## Stride-1 Access within a GPU thread

```

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

```

2-113

## Stride-1 Access within a GPU thread

```

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

```

2-114

## 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

2-115

## 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

2-116

```

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

```

2-117



```

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
end module

```

2-118



```

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

```

2-119



```

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

```

2-120



## Jacobi Relaxation

```

change = tolerance + 1.0
do while(change > tolerance)
  change = 0
  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)
enddo

```

2-121



## Jacobi Relaxation Main Kernel

```

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()

```

2-122



## Jacobi Relaxation Main Kernel

```

mynewa = w0*aa(ii,jj) + &
  w1*(aa(ii-1,jj) + aa(ii,jj-1) + &
    aa(ii+1,jj) + aa(ii,jj+1)) + &
  w2*(aa(ii-1,jj-1) + aa(ii-1,jj+1) + &
    aa(ii+1,jj-1) + aa(ii+1,jj+1))
change = abs(mynewa-aa(ii,jj))
newa(i,j) = mynewa
! store change in shared array before reducing
k = threadidx%x + (threadidx%y-1)*16
mychange(k) = change
call syncthreads()

```

2-123



## 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

```

2-124



## Jacobi Relaxation Reduction Kernel

```

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

```

2-125

## 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

```

2-126

## Constant Memory

- ❑ Small (64KB), read-only in a kernel, written by the host
  - assignment or API
  - used for small coefficient tables, common pointers
- ❑ Hardware cached
  - separate 16KB constant cache

2-127

## Texture Memory

- ❑ Requires special memory allocation, binding to a texture handle, special data fetch routines
- ❑ Hardware cached
- ❑ Fermi data caches may be more effective

2-128

## Time for a Live Demo (9)

### CUDA Vector Add

#### Non-stride-1 Accesses

2-129

The Portland Group®

## 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`

2-130

The Portland Group®

## Time for a Live Demo (10)

### CUDA Matmul

#### Unrolled loops, Loop fusion example

2-131

The Portland Group®

## 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];
}
```

132

The Portland Group®

## 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 );
    
```

133

## 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

2-134

## 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
    
```

135

## 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()
    
```

136

### 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
    
```

137



### 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
endif
enddo
pi = 4.0*float(s) / float(n)
    
```

138



### Monte Carlo Example Walk-through

- ❑ The Monte Carlo algorithm is used to approximate an integral of a function over a multidimensional volume.
- ❑ Taking a large sample of uniformly random numbers, the algorithm checks if the numbers fall within the function. The arithmetic mean over N samples will be the integral.
- ❑  $\langle f \rangle \approx 1/N \sum f(x_i)$

- ❑ Pros: Highly parallel since each random point can be calculated independently.
- ❑ Cons: The points must be summed. Needs a large random number set.



### Computing PI

- ❑ To compute PI using the Monte Carlo Algorithm use the function:  $f(x,y) = (x^2 + y^2 < 1) ? 1 : 0$
- ❑ The points are then summed. The approximate value of pi can then be calculated by multiplying four times the volume of the square with the mean value for  $f(x,y)$ .

```

! Perform the function
! f(x,y) = (x^2 + y^2 < 1) ? 1 : 0
! X and Y are a set of random points
do i=1,N
  tempVal = X(i)*X(i) + Y(i)*Y(i)
  if (tempVal < 1) then
    temp(i) = 1
  else
    temp(i) = 0
  endif
endif
enddo

! Sum the results
sumA = 0
sumSq = 0
do i=1,N
  sumA = sumA + temp(i)
  sumSq = sumSq + (temp(i)*temp(i))
enddo

! calculate the mean
meanA = sumA / real(N)
meanSq = sumSq / real(N)

! approximate pi
results%estimate = meanA * volume * 4
results%variance = (meanSq - meanA*meanA) / (N - 1)
    
```

Demo: Review the CPU code and run baseline performance



## Simple Device Kernel

```

attributes(global)
subroutine montecarlo_cuf1_kernel(dX, dY, dTemp, N)
  use cudafor
  implicit none
  real, dimension(N), device :: dX, dY, dTemp
  integer, value :: N
  integer :: i
  real :: tempVal

  i = (blockIdx%x-1)*blockDim%x + threadIdx%x
  tempVal = dX(i)*dX(i) + dY(i)*dY(i)
  if (tempVal < 1.0) then
    dTemp(i) = 1.0
  else
    dTemp(i) = 0.0
  endif
end subroutine montecarlo_cuf1_kernel

```



## Host Code - Timers

```

! timer variables
real :: sum_start, sum_end, func_time, datat_time
type(cudaEvent) :: func_start, func_end, datat_start, datat_end

! Initialize our data xfer timing routines
istat = cudaEventCreate(datat_start)
istat = cudaEventCreate(datat_end)
istat = cudaEventRecord(datat_start, 0)

! Copy the random points from the host to device
dX = X
dY = Y

! get the data transfer time
istat = cudaEventRecord(datat_end, 0)
istat = cudaThreadSynchronize()
istat = cudaEventElapsedTime(datat_time, datat_start, datat_end)
results%time(5) = results%time(5) + (datat_time/1000)

```



## Host Code – Launching the kernel

```

! set our Grid and Block sizes
dimBlock = dim3(256,1,1)
dimGrid = dim3(N/dimBlock%x,1,1)

! Timing code omitted in the slide
! call our device kernel using the grid/block dimensions
! and passing in device pointers to our random point and temp
call montecarlo_cuf1_kernel<<<dimGrid,dimBlock>>>(dX,dY,dTemp,N)

! Check for errors
errCode = cudaGetLastError()
if (errCode .gt. 0) then
  print *, cudaGetErrorString(errCode)
endif

! copy the result temp array back to the host
temp = dTemp

```



## Running the simple example

- ❑ Example will fail due to a “ConfigureCall FAILED:9” error.
- ❑ The maximum number of threads per block is 512 on a Tesla and 1024 on a Fermi. The maximum number of blocks is 64K.
- ❑ The sample uses 67108860 with 256 threads per block. This is simply too big.
- ❑ We'll fix this later. For now lower the value of N to 16777215.
- ❑ In the Makefile, use -DUSE\_SMALL



## Sum Reduction

- To improve performance we want to reduce the data transfer between the host and GPU. Instead of copying back the results array, let's perform the sum reduction on the GPU.

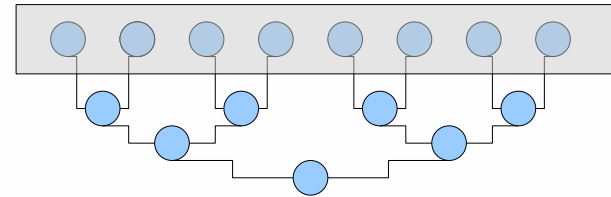
```
! Add a new device kernel
attributes(global) subroutine montecarlo_cuf2_sum(dTemp, N, &
  dSum, dSumSq)

implicit none
real, dimension(N), device :: dTemp
real, device :: dSum, dSumSq
integer, value :: N, i
dSum = 0
dSumSq = 0
do i=1,N
    dSum = dSum + dTemp(i)
    dSumSq = dSumSq + (dTemp(i)*dTemp(i))
enddo
end subroutine montecarlo_cuf2_sum
...
! Launch serially from the host
call montecarlo_cuf2_sum<<<1,1>>>(dTemp, N, dSum, dSumSq)
```



## Parallel Sum Reduction

- While the serial sum reduction did reduce the data transfer time, the serial sum reduction was very slow.
- Instead we should perform as much of the reduction in parallel using partial sums.



Reference:  
[http://developer.download.nvidia.com/compute/cuda/1\\_1/Website/projects/reduction/doc/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf)



## Parallel Sum Reduction - cont.

```
attributes(global) subroutine montecarlo_cuf3_sum0(dTemp, dSumA, dSumSqA, N)
implicit none
real, dimension(N), device :: dTemp
real, dimension(sumN), device :: dSumA, dSumSqA
integer, value :: N
integer :: i, ix, iy, nthrd, start, end, share, rem
real :: tempVal
ix = (blockIdx%x-1)*blockDim%x + threadIdx%x

! The total number of threads
nthrd = blockDim%x * gridDim%x
! Thread's share of the elements
share = N/nthrd
! Thread's starting and ending index
start = (share*(ix-1)) + 1
end = start + share - 1
! Add the extra elements to the end
if (ix .eq. sumN) then
    rem = N - (share*nthrd)
    end = end + rem
endif

dSumA(ix) = 0
dSumSqA(ix) = 0
do i=start,end
    dSumA(ix) = dSumA(ix) + dTemp(i)
    dSumSqA(ix) = dSumSqA(ix) + (dTemp(i)*dTemp(i))
enddo
end subroutine montecarlo_cuf3_sum0
```

Calculate the thread number as before, however each thread will now perform the sum on more than one element of the temporary array

Determine this thread's share of the work

perform the partial summation



## Improving Performance

```
attributes(global) subroutine montecarlo_cuf4_sum0(dTemp,
dSumA, dSumSqA, N)
implicit none
real, dimension(N), device :: dTemp
real, dimension(THREAD_N), device :: dSumA, dSumSqA
integer, value :: N
integer :: i, ix, iy, nthrd, start, end, share, rem
real :: tempVal, sum, sumSq

ix = (blockIdx%x-1)*blockDim%x + threadIdx%x
nthrd = blockDim%x * gridDim%x
sum = 0
sumsq = 0
do i=ix,N,nthrd
    sum = sum + dTemp(i)
    sumSq = sumSq + (dTemp(i)*dTemp(i))
enddo

dSumA(ix) = sum
dSumSqA(ix) = sumSq
end subroutine montecarlo_cuf4_sum0
```

Improved thread memory access

Use scalars to reduce access to global memory



## Fixing the main kernel

```

attributes(global) subroutine montecarlo_cuf5_kernel (dX, dY, dTemp, N)
  implicit none
  real, dimension(N), device :: dX, dY
  real, dimension(N), device :: dTemp
  integer, value :: N
  integer :: i, ix, iy, nthrd, start, end, share, rem
  real :: tempVal
  ix = (blockIdx%x-1)*blockDim%x + threadIdx%x
  nthrd = blockDim%x * gridDim%x
  do i=ix,N,nthrd
    tempVal = dX(i)*dX(i) + dY(i)*dY(i)
    if (tempVal < 1.0) then
      dTemp(i) = 1.0
    else
      dTemp(i) = 0.0
    endif
  enddo
end subroutine montecarlo_cuf5_kernel
    
```

Have each thread process more than one element!



## Calling a CUDA C Routine

Write an interface to the CUDA C Routines using F2003 ISO C Binding

```

interface
  ! C function to load the Mersenne Twister input data set
  subroutine loadMTGPU(dat_path) bind(c,name='loadMTGPU')
    use iso_c_binding
    character(len=80) :: dat_path
  end subroutine loadMTGPU
  ! C function to initialize the random seed on the GPU
  subroutine seedMTGPU(seed) bind(c,name='seedMTGPU')
    use iso_c_binding
    integer(c_int), value :: seed ! Pass by value
  end subroutine seedMTGPU
  ! CUDA C Mersenne Twister algorithm to generate an array of random
  ! numbers in parallel.
  subroutine RandomGPU(d_Rand, n_per_rng) bind(c,name='randomgpu')
    use iso_c_binding
    real(c_float), dimension(:), device :: d_Rand
    integer, value :: n_per_rng ! pass by value
  end subroutine RandomGPU
end interface
    
```



## Random Numbers

- ❑ Runtime is dominated by the time to generate two sets of random numbers and copy them to the host.
- ❑ calling RANDOM\_NUMBER from a device is not yet supported.
- ❑ Need to generate a list of random numbers on the GPU using a parallel algorithm.
- ❑ The NVIDIA CUDA SDK includes a CUDA C implementation of the Mersenne Twister parallel random number generator.
- ❑ Instead of re-writing this in CUDA Fortran, why not just call the CUDA C version?

Reference:

<http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf>



## Calling a CUDA C Routine - cont.

```

! initialize and then call the RNG for dX
write(dpath,'(a)') 'mtdata/MersenneTwister.dat'//char(0)
call loadMTGPU(dpath)
call seedmtgpu(777)
call randomgpu<<<32,128>>>(dX,N_PER_THD)

! Re-seed and call the RNG for dY
call seedmtgpu(123)
call randomgpu<<<32,128>>>(dY,N_PER_THD)
    
```

Calling a CUDA C kernel is the same as CUDA Fortran



## Fermi vs Tesla

- |                                                        |                                                         |
|--------------------------------------------------------|---------------------------------------------------------|
| <input type="checkbox"/> ECC                           | <input type="checkbox"/> no ECC                         |
| <input type="checkbox"/> double = float / 2            | <input type="checkbox"/> double = float / 8             |
| <input type="checkbox"/> two level hardware data cache | <input type="checkbox"/> no user-visible hardware cache |
| <input type="checkbox"/> constant memory cache         | <input type="checkbox"/> constant memory cache          |
| <input type="checkbox"/> 16/48KB shared memory         | <input type="checkbox"/> 16KB shared memory             |
| <input type="checkbox"/> <=16 kernels at a time        | <input type="checkbox"/> 1 kernel at a time             |
| <input type="checkbox"/> 2*16TP * 14SM                 | <input type="checkbox"/> 8TP * 30SM                     |
| <input type="checkbox"/> unified address space         | <input type="checkbox"/> shared/local/global ptrs       |
| <input type="checkbox"/> dynamic allocation (?)        | <input type="checkbox"/> allocate from host only        |
| <input type="checkbox"/> enhanced support for C++      |                                                         |

2-153



## Fermi vs ATI

- |                                                        |                                                         |
|--------------------------------------------------------|---------------------------------------------------------|
| <input type="checkbox"/> ECC                           | <input type="checkbox"/> EDC                            |
| <input type="checkbox"/> double = float / 2            | <input type="checkbox"/> double = float / 5             |
| <input type="checkbox"/> two level hardware data cache | <input type="checkbox"/> no user-visible hardware cache |
| <input type="checkbox"/> constant memory cache         | <input type="checkbox"/> constant memory cache          |
| <input type="checkbox"/> texture cache                 | <input type="checkbox"/> texture cache                  |
| <input type="checkbox"/> 16/48KB shared memory         | <input type="checkbox"/> 16KB local data store          |
| <input type="checkbox"/> <=16 kernels at a time        | <input type="checkbox"/> 1 kernel at a time?            |
| <input type="checkbox"/> 2*16TP * 16SM                 | <input type="checkbox"/> 16PE * 10SU * 5 "cores"        |
| <input type="checkbox"/> unified address space         | <input type="checkbox"/> ?                              |
| <input type="checkbox"/> dynamic allocation (?)        | <input type="checkbox"/> ?                              |
| <input type="checkbox"/> enhanced support for C++      |                                                         |

2-154



## Fermi vs Larrabee(Knight's Ferry)

- |                                                        |                                                        |
|--------------------------------------------------------|--------------------------------------------------------|
| <input type="checkbox"/> ECC                           | <input type="checkbox"/> ECC                           |
| <input type="checkbox"/> double = float / 2            | <input type="checkbox"/> double = float / 2            |
| <input type="checkbox"/> two level hardware data cache | <input type="checkbox"/> two level hardware data cache |
| <input type="checkbox"/> constant memory cache         |                                                        |
| <input type="checkbox"/> texture cache                 |                                                        |
| <input type="checkbox"/> 16/48KB shared memory         |                                                        |
| <input type="checkbox"/> <=16 kernels at a time        | <input type="checkbox"/> N kernels at a time           |
| <input type="checkbox"/> 2*16TP * 16SM                 | <input type="checkbox"/> 32 cores * 16-wide SSE        |
| <input type="checkbox"/> unified address space         | <input type="checkbox"/> unified address space         |
| <input type="checkbox"/> dynamic allocation (?)        | <input type="checkbox"/> full OS, dynamic allocation   |
| <input type="checkbox"/> enhanced support for C++      | <input type="checkbox"/> full language support         |

2-155



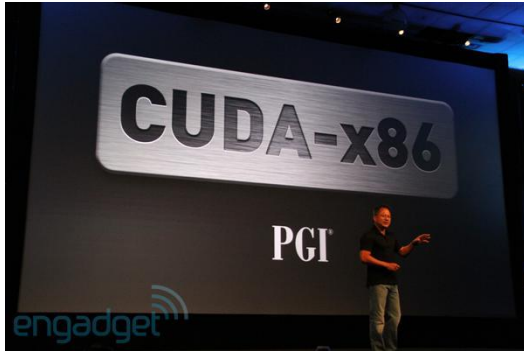
## CUDA 4.0 Features

- Easier to share a GPU across host threads
  - single thread-safe context shared across threads
- Easier to use multiple GPUs in a single host thread
- Easy to pin existing host memory
- GPUDirect v2.0, direct GPU-to-GPU memory (Fermi+)
- 3-dimensional grids (Fermi+)
- Unified virtual addressing (Fermi+, 64b mode, Linux/Windows)
- Thrust library, binary disassembler, cuda-gdb for Mac, ...

2-156



## NVIDIA Teams with PGI for CUDA-x86



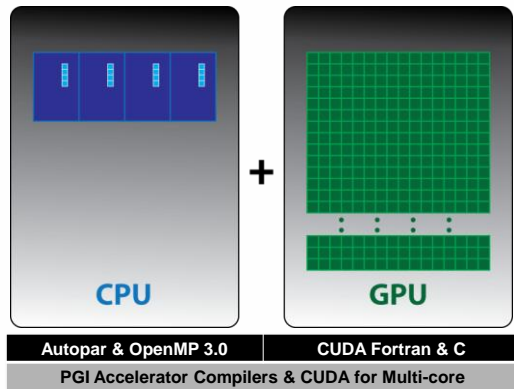
## CUDA C/C++ for x86 Motivation

Provide CUDA developers with a common code path for both  
NVIDIA GPU and x86 platform support

Run CUDA C applications on x86 clusters

## Integrated x64+GPU Optimization

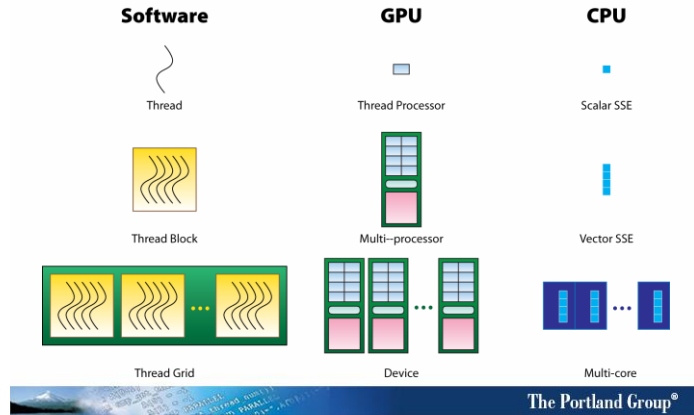
PGI Compilers for Heterogeneous Many-core Platforms



## PGI CUDA C/C++ for Multi-core x86

- ❑ Will track NVIDIA's definition and evolution of the CUDA C language for GPUs moving forward
- ❑ Implementation will proceed in phases
  - Phase 0 prototype – demo at SC10 in New Orleans
  - Phase 1 first production – in Q2 2011 with most CUDA C functionality; not a performance release
  - Phase 2 performance – in Q4 2011 leveraging multi-core and SSE/AVX to implement low-overhead native parallel/SIMD execution
  - Phase 3 unification – in H2 2012 with support for compiling device kernels for execution on either multi-core or NVIDIA GPUs
- ❑ PGI Unified Binary technology will enable one binary that uses NVIDIA GPUs when present or defaults to multi-core x86 if no GPU is present

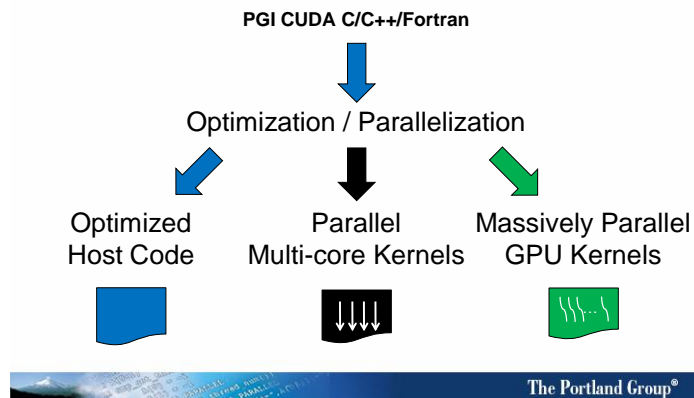
## CUDA for GPUs vs Multi-core x86



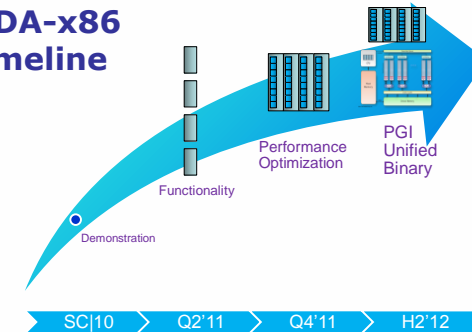
## Optimized CUDA C for Multi-core x86

- Process CUDA C/C++ as a native parallel programming language for multi-core x86
- Reconstitute chevron syntax into parallel/vector loops, use multiple cores and SSE/AVX instructions to effect parallel execution
- Execute each CUDA thread block using a single host core, eliminate synchronization where possible
- Host Code: all PGI optimizations for Intel/AMD host code will be supported

## PGI CUDA Compilers for Multi-core x86 & NVIDIA GPUs



## CUDA-x86 Timeline



- More information at [www.pgroup.com/cuda\\_x86.htm](http://www.pgroup.com/cuda_x86.htm)

## Copyright Notice

© 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.

2-165

