

# **PGI® 2010 Compilers & Tools for x64+GPU Systems**

**Douglas Miles**  
**douglas.miles@pgroup.com**  
**www.pgroup.com**

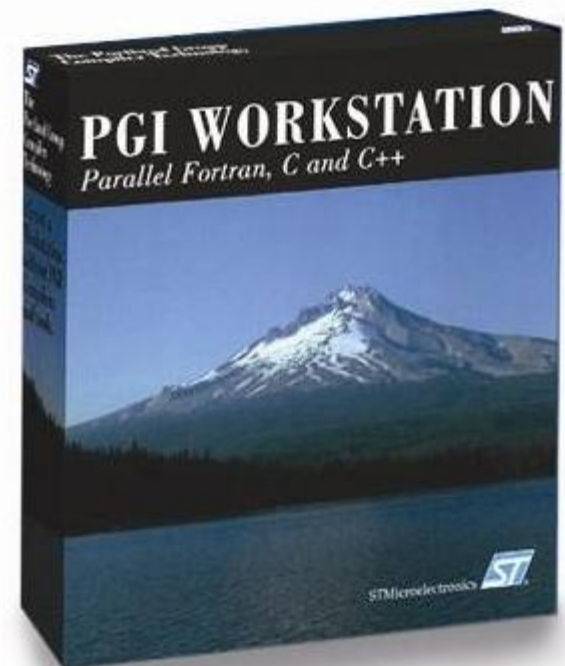
**September, 2010**



# PGI Workstation / Server / CDK

Linux, Windows, MacOS, 32/64-bit, AMD64, Intel 64, NVIDIA  
UNIX-heritage Command-level Compilers + Graphical Tools

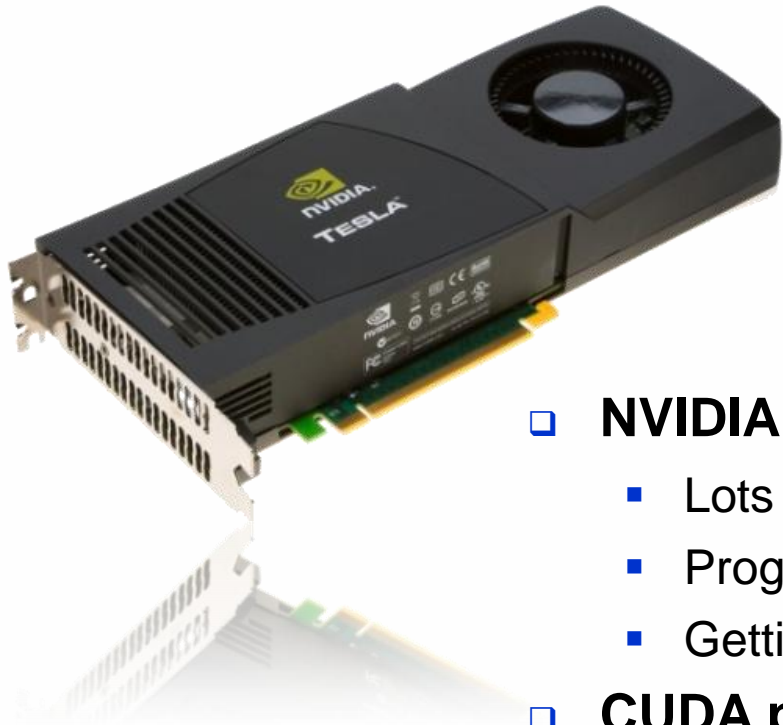
Compiler	Language	Command
<b>PGFORTRAN™</b>	Fortran 95, partial F2003, CUDA Fortran	pgfortran
<b>PGCC®</b>	ANSI C99, K&R C and <i>GNU gcc Extensions</i>	pgcc
<b>PGC++®</b>	ANSI/ISO C++	pgCC
<b>PGDBG®</b>	MPI/OpenMP debugger	pgdbg
<b>PGPROF®</b>	MPI/OpenMP/ACC profiler	pgprof



***A Self-contained OpenMP/MPI/Accelerator  
Compiler & Tools Solution for Scientists & Engineers***



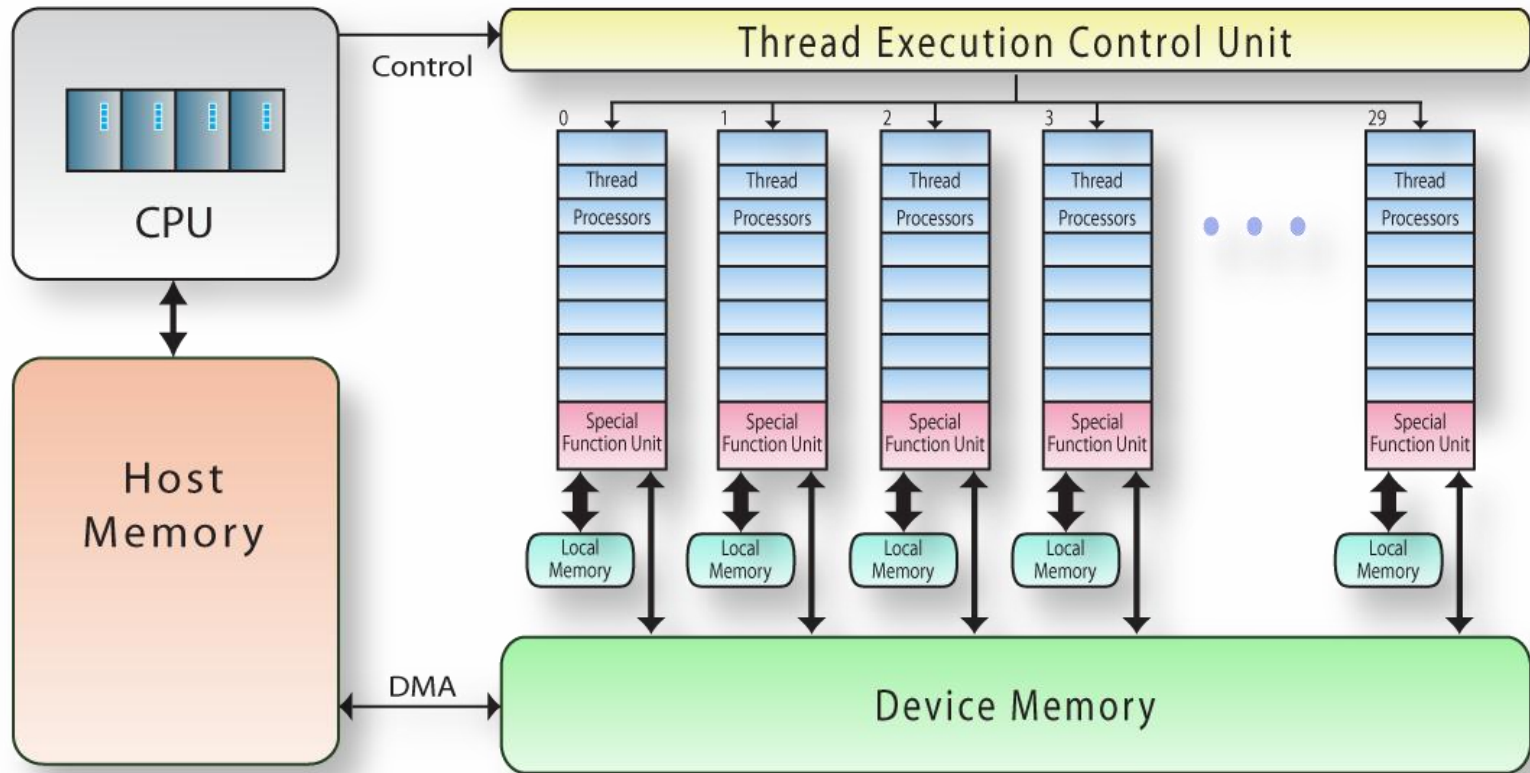




- **NVIDIA TESLA C1060 and C2050 (Fermi)**
  - Lots of available performance 1 - 2 TFlops peak SP
  - Programming is a challenge
  - Getting high performance is lots of work
- **CUDA programming model and compilers simplify GPGPU programming**
  - **NVIDIA CUDA C** *much* easier than OpenGL, but still challenging especially for porting legacy codes
  - **PGI CUDA Fortran** – a Fortran analog to CUDA C
- **A PGI Goal: do for GPU programming what OpenMP did for Posix Threads**

# Emerging Cluster Node Architecture

## Commodity Multicore x86 + Commodity Manycore GPUs

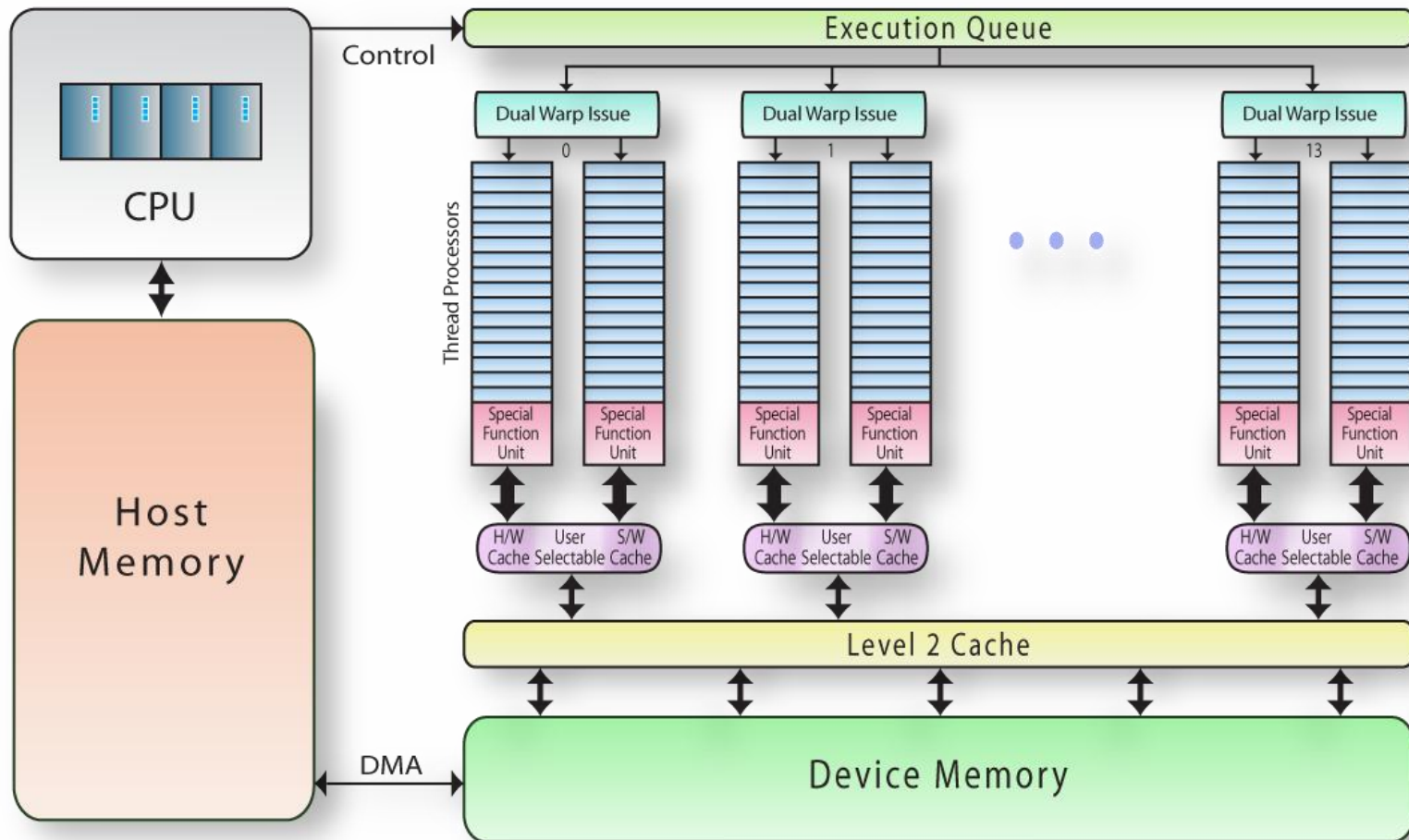


©2010 The Portland Group, Inc.

**4 – 32 CPU Cores**

**240 – 1792 GPU/Accelerator Cores**

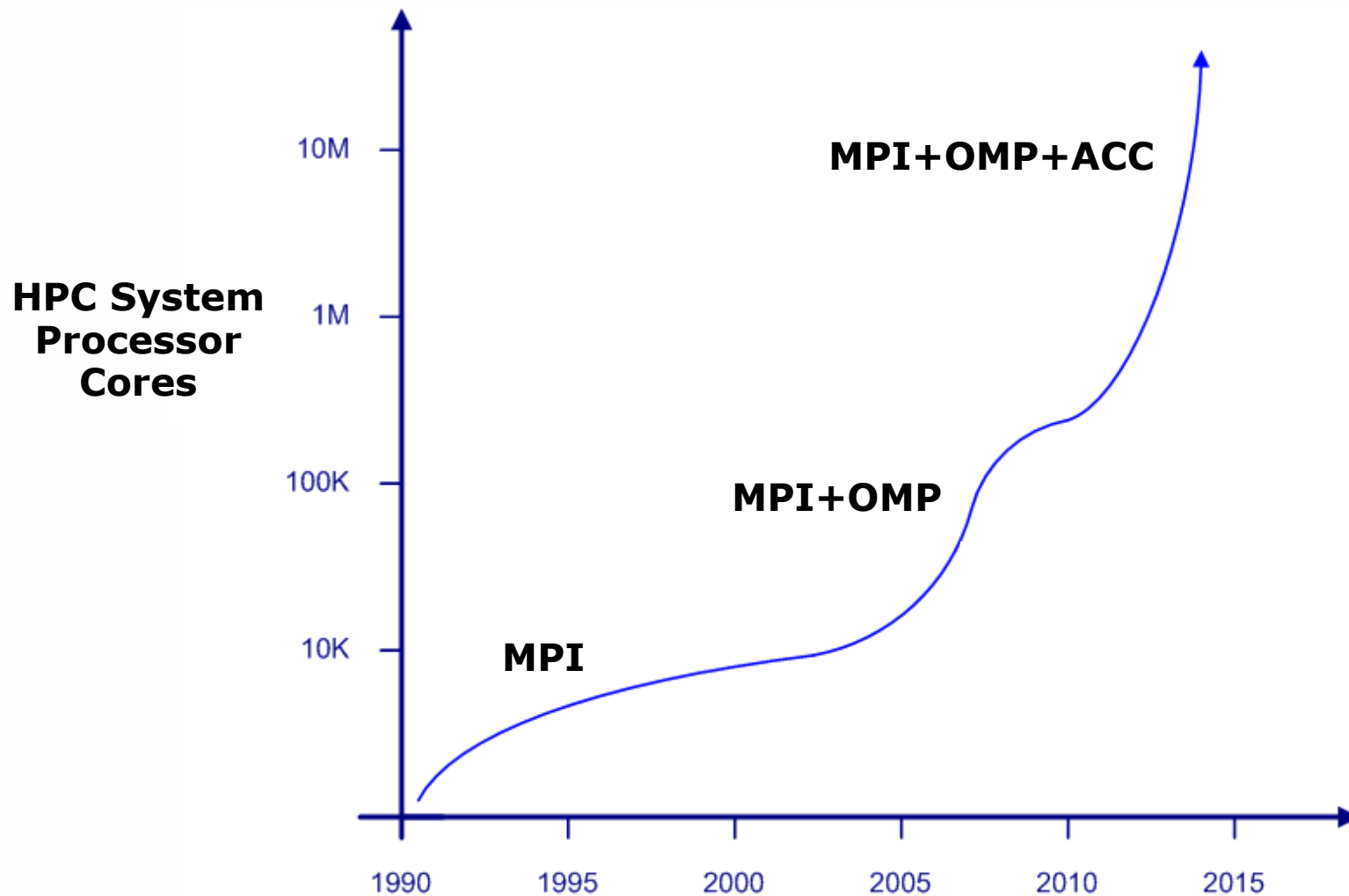
# Abstracted x64+Fermi Architecture



©2010 The Portland Group, Inc.

# Compilers & Programming Models Must Evolve for Each New Generation of HPC Hardware

Expect 20M Core systems in the Next Few Years



# PGI Compiler Features for GPU Programming

- ❑ **CUDA Fortran** – leverages standard F95 language and a few extensions to simplify GPU programming – array syntax, typing, data allocation
- ❑ **PGI Accelerator C99 and Fortran** – implements most features of CUDA C and CUDA Fortran as directives to enable incremental development of standard-compliant and fully-portable GPU-enabled applications

# CUDA Fortran

- ❑ CUDA Fortran is an analog to NVIDIA's C for CUDA
- ❑ Co-defined by PGI and NVIDIA and implemented in the PGI Fortran compiler
- ❑ Supports the full CUDA API plus intuitive Fortran language extensions to simplify Host/GPU data management
- ❑ Supported on Linux, MacOS and Windows, including support within PGI Visual Fortran on Windows

# 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
```

***Fortran***

```
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];
  }
}
```

***C***

# 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];  
}
```

# 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
```

# 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 );
}
```

# 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
```

# CUDA Fortran Programming

- Host code
  - Optional: select a GPU
  - Allocate device data
  - Copy data to device memory
  - Launch kernel(s)
  - Copy data from device memory
  - Deallocate device data
- Device code
  - Scalar thread code, limited operations
  - Implicitly parallel

# 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
```

Allocate device memory

Copy data to device

Launch a kernel

Copy data back from device

Deallocate device memory

# 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

# CUDA Programming: the GPU

- A scalar program, runs on one CUDA thread
  - All threads run the same code
  - Executed in thread groups called “thread blocks”
  - grid may be 1D or 2D (max 65535x65535)
  - thread block may be 1D, 2D, or 3D
    - max total thread block 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

# 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
```

global means kernel

device attribute implied

blockidx from 1..(N+31)/32

threadidx from 1..32

array bounds test

# 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

# Writing a CUDA Fortran Kernel (2)

- Predefined variables
  - `blockidx`, `threadidx`, `griddim`, `blockdim`, `warpsize`
- Executable statements in a kernel
  - arithmetic operations and assignment
  - `do`, `if`, `goto`, `case`
  - `call` (to device subprogram, must be inlined)
  - intrinsic function call, device subprogram call (inlined)
  - `where`, `forall`

# 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
- where, forall

# Building a CUDA Fortran Program

- ❑ `% pgfortran a.cuf`
  - `.cuf` suffix implies CUDA Fortran (free form)
  - `.CUF` suffix runs preprocessor
  - `-Mfixed` for F77-style fixed format
  
- ❑ `% pgfortran -Mcuda [= [emu | cc10 | cc11 | cc12 | cc13 | cc20] ] . . .`
  - To specify compute capability of target device
  - Use `-Mcuda=emu` for debugging kernels on host
  
- ❑ **Must use `-Mcuda` when linking from object files**
  - Compiler driver pulls in correct path and libraries
  
- ❑ **All other PGI Fortran compiler options are accepted and apply to host code (`-fast`, etc)**

# You can use the CUDA API

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

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

istat = cudaMemcpyAsync(a, x, 10, istrm)
```

# CUDA C vs CUDA Fortran

## □ CUDA C

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

## □ CUDA Fortran

- NO texture memory
- supports Runtime API
- NO Driver API
- allocate, deallocate
- assignments
- NO OpenGL interoperability
- NO Direct3D interoperability
- NO textures (yet)
- arrays one-based
- threadIdx/blockIdx 1-based
- allocatable are device/host
- pinned attribute

# PGI Accelerator Fortran and C

- ❑ Built on lessons from Vector and SMP programming (the latter standardized as OpenMP)
- ❑ Directive-based method to delineate compute kernels to be offloaded to a GPU, manage data movement between Host and GPU, and map loop parallelism onto a GPU
- ❑ Enables use in Fortran and C99 today, eventually C++
- ❑ Programs remain 100% standard compliant and portable to other compilers and HW
- ❑ Enables incremental porting/tuning of applications to heterogeneous Host+Accelerator targets
- ❑ Designed to enable development of applications that are performance portable to multiple types of accelerators

```

void saxpy (float a,
float *restrict x,
float *restrict y, int n){
#pragma acc region
{
    for (int i=1; i<n; i++)
        x[i] = a*x[i] + y[i];
}
}

```

compile

# PGI Accelerator Compilers

## GPU/Accelerator Code

## Host x86 Code

```

saxpy:
...
    movl    (%rbx), %eax
    movl    %eax, -4(%rbp)
    call   __pg_cu_init
    . . .
    call   __pg_cu_alloc
...
    call   __pg_cu_uploadp
...
    call   __pg_cu_paramset
...
    call   __pg_cu_launch
...
    Call   __pg_cu_downloadp
...

```

+

```

static __constant__ struct{
    int tc1;
    float* _y;
    float* _x;
    float _a;
}a2;

extern "C" __global__ void
pgi_kernel_2() {
    int i1, i1s, ibx, itx;
    ibx = blockIdx.x;
    itx = threadIdx.x;
    for( i1s = ibx*256; i1s < a2.tc1; i1s += gridDim.x*256 ){
        i1 = itx + i1s;
        if( i1 < a2.tc1 ){
            a2._x[i1] = (a2._y[i1]+(a2._x[i1]*a2._a));
        }
    }
}

```

link

Unified HPC Application

execute

... with no change to existing makefiles, scripts, programming environment, etc

# PGI Accelerator Directives

## 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

# Jacobi Relaxation in OpenMP

```
change = tolerance + 1.0
!$omp parallel shared(change)
do while(change > tolerance)
  change = 0
!$omp do reduction(max:change) private(i,j)
  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
```

# Jacobi Relaxation for GPU

```
change = tolerance + 1.0
!$acc data region local(newa(1:m,1:n)) &
    copy(a(1:m,1:n))
do while(change > tolerance)
    change = 0
!$acc region
    do j = 2, n-1
    do i = 2, m-1
        newa(i,j) = w0 * a(i,j) + &
            w1 * (a(i-1,j) + a(i,j-1) + &
                a(i+1,j) + a(i,j+1)) + &
            w2 * (a(i-1,j-1) + a(i-1,j+1) + &
                a(i+1,j-1) + a(i+1,j+1))
        change = max(change, abs(newa(i,j) - a(i,j)))
    enddo
    enddo
    a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!$acc end region
enddo
!$acc end data region
```

# Loop Schedules

**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 Schedules

```
change = tolerance + 1.0
!$acc data region local(newa(1:m,1:n)) &
    copy(a(1:m,1:n))
do while(change > tolerance)
    change = 0
!$acc region
!$acc do parallel, vector(8)
    do j = 2, n-1
!$acc do parallel, vector(32)
    do i = 2, m-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
...

```

# Compiler-to-User Feedback

```
% pgfortran -fast -ta=nvidia -Minfo mm.c
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'
```

# PGI Accelerator vs CUDA

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

# Near Term Future Additions

- ❑ Remaining loop clauses (unroll, cache)
- ❑ Implicit data regions
- ❑ Passing device data to subroutines, **mirror**, **reflected**
- ❑ Code optimization
  - aggressive procedure inlining in compute regions
  - instruction count optimizations
  - low-level code improvements
- ❑ Automatic private array detection
- ❑ Improvements to kernel schedule selection
- ❑ Direct calling of CUDA kernels inside accelerator regions
- ❑ Direct use of CUDA data

# Longer Term Evolution

- ❑ C++
- ❑ New targets – Multicore, ATI? Larrabee? Other?
- ❑ Overlap compute / data movement - pipelined loops
- ❑ More tools support
- ❑ Libraries of device routines
- ❑ PGI Accelerator programming model evolution
  - Asynchronous kernel execution / data transfer
  - Multiple homogeneous Accelerators
  - Multiple heterogeneous Accelerators
  - Standardization
  - Concurrent kernels, a la OpenCL / Fermi

# Reference Materials

- **PGI Accelerator programming model** – supported for x64+NVIDIA targets in the PGI 2010 Fortran 95/03 and C99 compilers
  - [http://www.pgroup.com/lit/whitepapers/pgi\\_accel\\_prog\\_model\\_1.2.pdf](http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf)
- **CUDA Fortran** – supported on NVIDIA GPUs in PGI 2010 Fortran 95/03 compiler
  - <http://www.pgroup.com/lit/whitepapers/pgicudaforug.pdf>
- **Understanding the CUDA Data Parallel Threading Model**
  - <http://www.pgroup.com/lit/articles/insider/v2n1a5.htm>