PGI® 2011 Compilers and Tools Fortran 2003, CUDA Fortran, CUDA C for X86, PGI Accelerator

Dave Norton
dave.norton@pgroup.com
norton@hpfa.com
www.hpfa.com

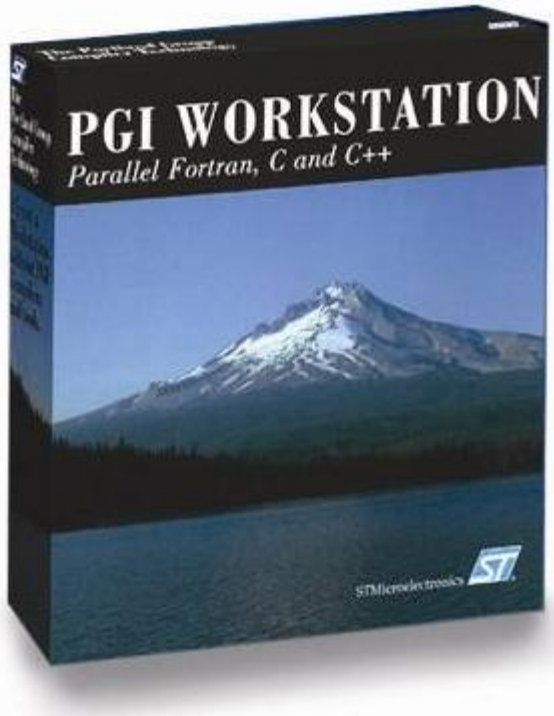
14th Workshop on the Use of High Performance Computing in Meteorology

3 NOV 10

PGI Workstation / Server / CDK

Linux, Windows, MacOS, 64-bit, 32-bit, AMD, Intel, Nvidia Compilers and Graphical Tools

Compiler	Language	Command
PGFORTRAN™	Fortran 95, F2003, !$acc, CUDA Fortran	pgfortran
PGCC®	C99, K&R *C* , #pragma acc *gcc Extensions* , CUDA C	pgcc
PGC++®	ANSI/ISO *C++*	pgCC
PGDBG®	MPI/OpenMP debugger	pgdbg
PGPROF®	MPI/OpenMP/ACC profiler	pgprof

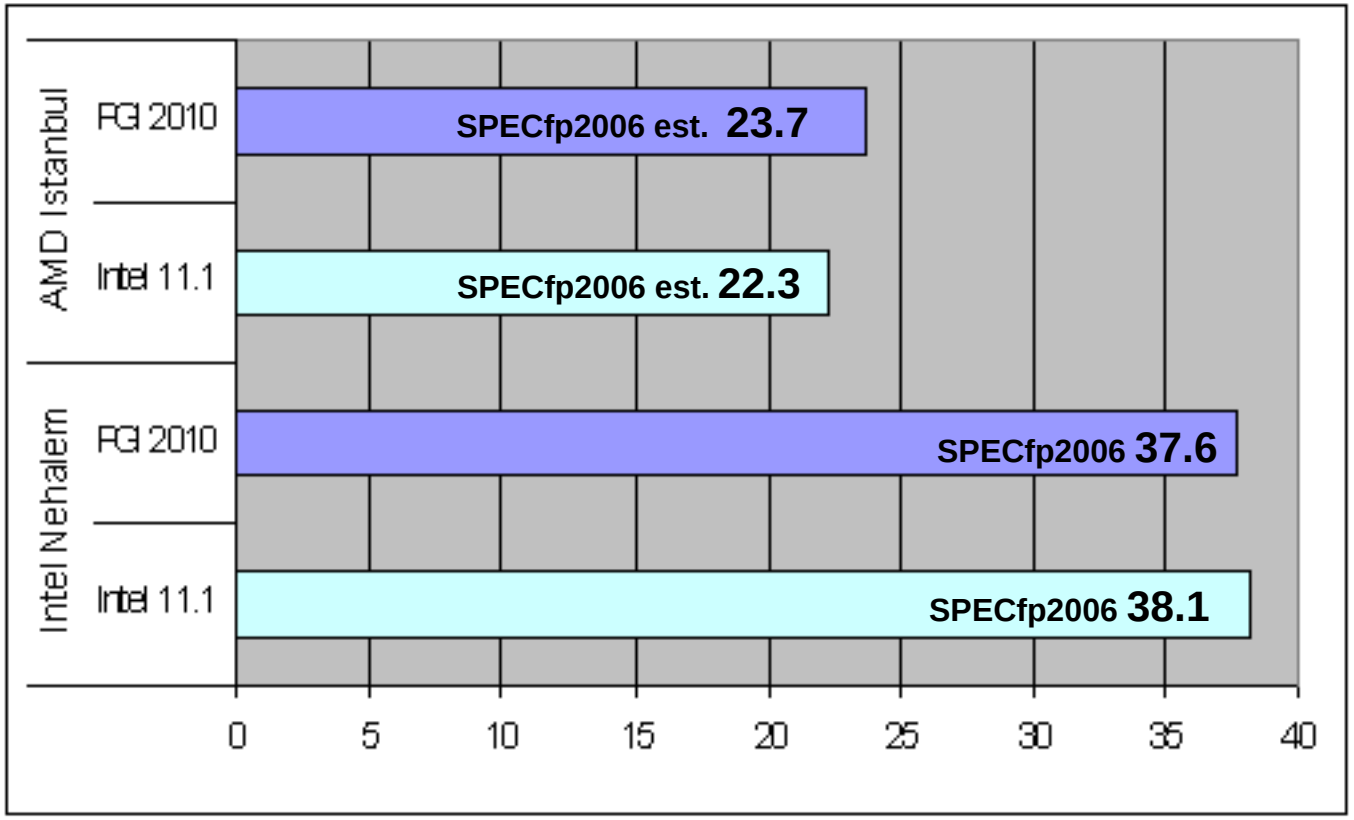


Self-contained OpenMP/MPI/Accelerator Parallel SW Development Solution

PGI® Compilers & Tools Positioning

- **HPC-focused Compilers & Tools technologies**
  - State of the art local, global and inter-procedural optimizations
  - Automatic vectorization and SIMD/SSE code generation
  - Support of OpenMP 3.0 standard
  - Automatic loop parallelization
  - Profile-guided optimization
  - PGI Unified Binary technology to target different ‘flavors’ of same architecture or heterogeneous architectures
  - Graphical tools to Debug/Profile multithreaded/multiprocess hybrid applications
  - Compilers & tools dedicated to scientific computing, where utilization of latest architecture features and speed on generated code is #1 criteria
- **GPU/Accelerator Compilers & Tools**
  - CUDA Fortran, and PGI Accelerator compilers for Nvidia
  - CUDA C compilers for PGI Accelerator for X86 and MIC

Multicore X64 Performance vs Intel



SPEC® and SPECfp® are registered trademarks of the Standard Performance Evaluation Corporation (SPEC) (www.spec.org)
Competitive benchmark results stated above reflect results performed by The Portland Group during the week of November 8th, 2009.
The Intel Nehalem system used is a Dell R610 using 2 Intel Xeon X5550 with 24GB DDR3-1333. The AMD Istanbul system is a kit built 2 Opteron 2431 system with 32GB DDR2-800. Since this system is not generally available, the AMD results should be considered estimates.

PGI Milestones

- **1989 PGI Formed**
- 1991 Pipelining i860 Compiler
- Intel Paragon support
- 1994 Parallel i860 Compiler
- 1996 ASCI Red TFLOPS Compiler
- 1997 Linux/x86 Compiler
- 1998 OpenMP for Linux/x86
- 1999 SSE/SIMD Auto-vectorization
- 2001 VLIW ST100 Compiler
- 2003 64-bit Linux/x86 Compiler
- 2004 ASCI Red Storm Compiler
- 2005 PGI Unified Binary
- 2006 PGI Visual Fortran
- 2007 64-bit MacOS Compiler
- **2008 PGI Accelerator Compiler**
- **2009 CUDA Fortran**
- **2011 CUDA C for x64 and MIC**

PGI® 2011 Features

- PGI Accelerator™ Programming Model
  - High-level, Portable, Directive-based Fortran & C extensions (no C++, yet)
  - Supported on NVIDIA CUDA GPUs
- PGI CUDA Fortran
  - Extended PGI Fortran, co-defined by PGI and NVIDIA
  - Lower-level explicit NVIDIA CUDA GPU programming
- **PGI CUDA C for x64 and MIC**
  - Demonstration of CUDA C for x64 at SC 2010 in New Orleans
  - Mixing of CUDA C, CUDA Fortran and PGI Accelerator directives
- Compiler Enhancements
  - **F2003 – object oriented features**
  - Latest EDG 4.1 C++ front-end – more g++/VC++ compatible, zero cost exception handling (-zc_eh)
  - **AVX code generation, code generator tuning**
- PGPROF Enhancements
  - Uniform performance profiling across Linux, MacOS and Windows
  - **x64+GPU performance profiling**
  - Updated Graphical User Interface (GUI)

Fortran 2003 Features in Current PGI Compiler Release

IEEE_EXCEPTIONS module	IEEE_ARITHMETIC module	Allocatable Array Extensions
ISO_C_Binding	c_f_pointer	c_f_procpointer
c_associated	Enumerators	Procedure Pointers
Interface procedure	Pass and Nopass Attribute	allocatable scalars
move_alloc()	Pointer Reshaping	Square brackets
volatile attribute and stmt	IMPORT statement	iso_fortran_env module
Access to environment	Length of names and statements	
Optional Kind to Intrinsics	Asynchronous I/O'	Wait Statement
PENDING specifier for INQUIRE		Access = 'stream’
POS specifier for INQUIRE	IOSTAT kind in all i/o stmts	SIZE kind in read/write stmts
Allow NAMELIST w/internal file		IEEE_ARITHMETIC large arrays
Classes	Type Extension(not polymorphic)	polymorphic entities
type uses CONTAINS declaration		Inheritance
EXTENDS_TYPE_OF intrinsic	SAME_TYPE_AS intrinsic	Typed allocation

Fortran 2003 Features in Current PGI Compiler Release

READ blank specifier
READ pad specifier
WRITE delim specifer
NEW_LINE intrinsic
IS_IOSTAT_END intrinsic
IS_IOSTAT_EOR intrinsic
SYSTEM_CLOCK COUNT_RATE is real
abstract interfaces
Type-bound procedures
PASS attribute
NOPASS attribute
NON_OVERRIDABLE attribute
PRIVATE and PUBLIC attributes
PRIVATE statement for type bound procedures
deferred type-bound procedures
ABSTRACT types
i/o keyword encoding
Decimal comma for i/o, dc, dp
ASYNCHRONOUS attribute and stmt
IEEE_FEATURES module
Max, Min take character
errmsg on allocate/deallocate
Mixed component accessibility
Sourced allocation (non-polymorphic)
Associate Construct
Sourced allocation (polymorphic types)

Fortran 2003 Object Oriented Features

	Compiler Release
generic type-bound procedures	11.0
select type construct	11.0
unlimited polymorphic entities	11.0
typed allocation for unlimited polymorphic entities	11.0
sourced allocation for unlimited polymorphic entities	11.0
deferred type parameters (requires MRC 15.2 & MRC 16.2.1)	11.x
select type construct for unlimited polymorphic entities	11.x
parameterized derived types (MRC 16.2.1)	11.x
final procedures	11.x

Fortran 2003 I/O Features

	Compiler Release
i/o of inf and nan (fs#3962)	11.0
round i/o specifier, ru,rd,etc.	11.0
non-default derived type I/O	11.x
non-default derived type I/O (type-bound procedures)	11.x
recursive I/O w/external file	11.x
recursive I/O w/internal file	11.x
SIGN= Specifier	11.0
NEXTREC, NUMBER, RECL, SIZE kind	11.0
DECIMAL in INQUIRE stmt	11.0
F2003 NAMELIST group entities	11.0

Fortran 2003 Remaining Features

	Compiler Release
deferred-character-length	11.0
generic & derived type the same	11.0
sourced allocation (deferred character)	11.0
PROTECTED attribute and stmt	11.0
stop stmt warns about FP exc.	11.0
rename user-defined operators	11.0
array constructor syntax	11.0
structure constructors	11.0
SELECTED_CHAR_KIND intrinsic	11.0

RAPS Fortran 2003 Status

Code Name	Feature	Status, FS, Comments
tr15581-1	Allocatable array arguments	PASSES
tr15581-2	Allocatable function results	PASSES
tr15581-3	Allocatable array components	PASSES
access	Module access control	PASSES
import	Import specified	PASSES
allocate_1	Allocate enhancements	Doesn't compile - uses F2008 feature
array_1	Array constructors	PASSES
constants	Complex constants	PASSES
intrinsics_1	Intrinsic enhanments	PASSES
iomsg	IOMSG specifier	PASSES
move_alloc	Intrinsic movealloc	PASSES
realloc	Allocatable array assignment	PASSES, NOTE: needs Mallocatable=03
pointer_assign	Pointer remapping	PASSES
pointer_intent	Pointer intent	PASSES
procptr_1	Procedure pointer	PASSES
procptr_2	Abstract interface	PASSES
value	Value attribute	PASSES
ifs_mk2.tar	cg minimization of quadradic cost	PASSES with rewrite
test_01	ISO fortran environment	PASSES
test_02	IEEE arithmetic	PASSES
test_03	Enumerator example	PASSES
test_04	ISO binding example	PASSES
pthread	Pthread example	PASSES (Linux)
linked_list	Unlimited polymorphism	Doesn't compile FS#17055 (11.0)

AVX Support in 11.0

The next generation of processors from both Intel and AMD will support AVX instructions.

AVX doubles the width of the floating point registers to 256 bits and adds 3 operand instructions resulting in more then a 2X decrease in assembly language instructions in performance critical sections of code

AVX are *vector* instructions where one instruction operates on 8 sp, or 4 dp words at the same time, effectively doubling the performance of the CPU.

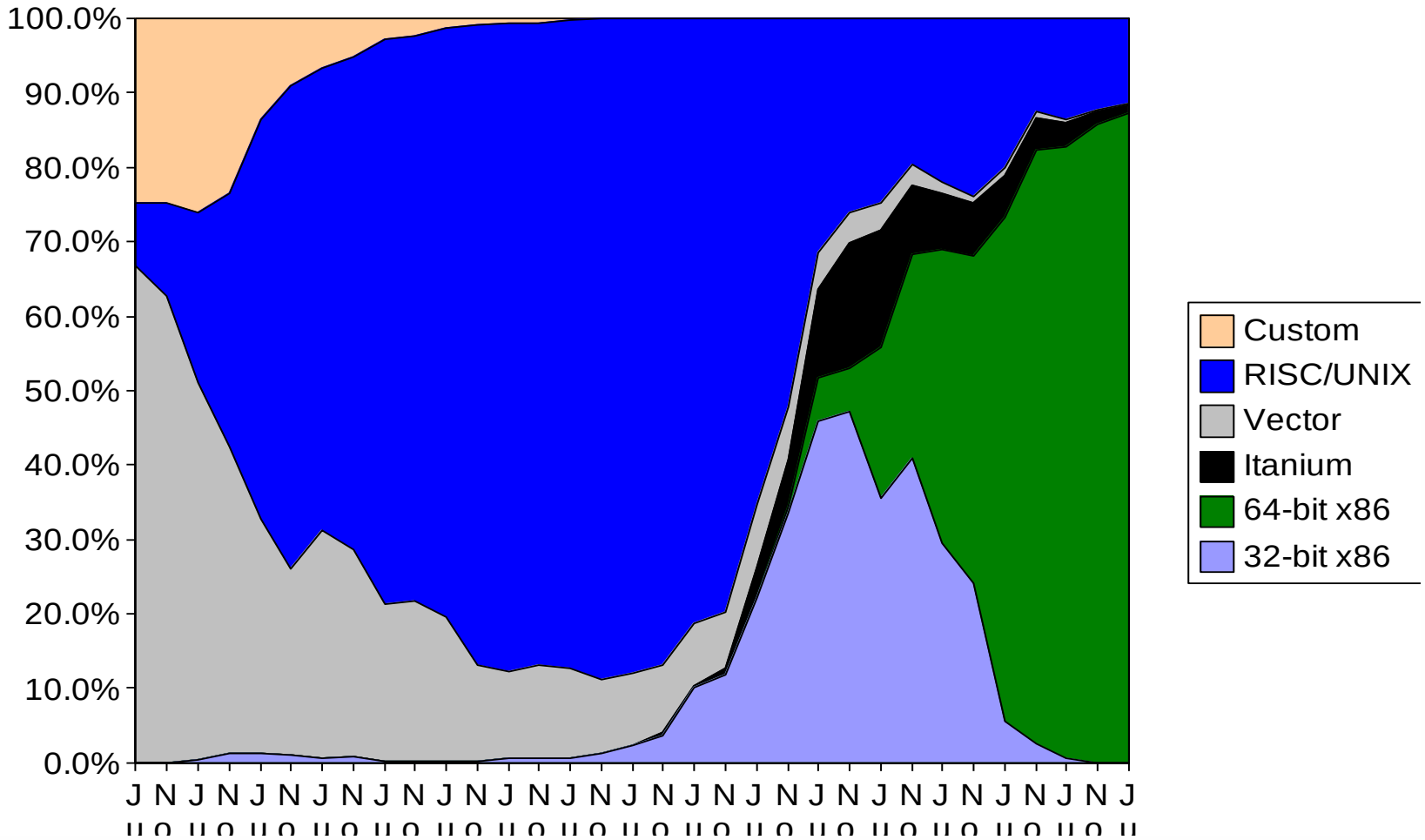
Codes should be compiled with –fast for vectorization and –Minfo to get compiler feedback

PGI compiled codes can made use of the Intel AVX simulator

HPC Hardware Trends

Today: Clusters of Multicore x86
Tomorrow? Clusters of Multicore x86 + Accelerators

Top 500



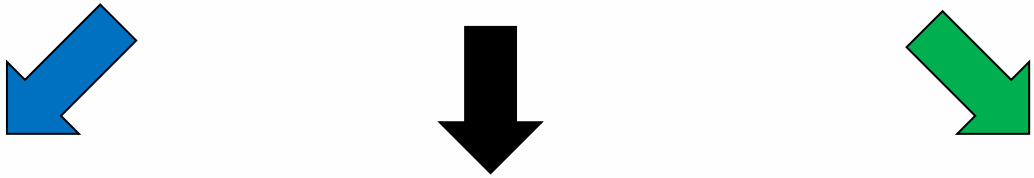
PGI CUDA C for Multi-core x86

- **Will track NVIDIA's definition and evolution of the CUDA C language for GPUs moving forward**
- **Implementation will proceed in phases**
  - Phase 1 prototype demonstration at SC10 in New Orleans (November)
  - Phase 2 first production release in Q2 2011 with most CUDA C functionality; not a performance release
  - Phase 3 performance release in Q4 2011 leveraging multi-core and SSE/AVX to implement low-overhead native parallel/SIMD execution
- **Will eventually support execution of Device kernels on NVIDIA CUDA-enabled GPUs as well**
- **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**

PGI CUDA Compilers for Multi-core x86 & NVIDIA GPUs

PGI CUDA C/Fortran

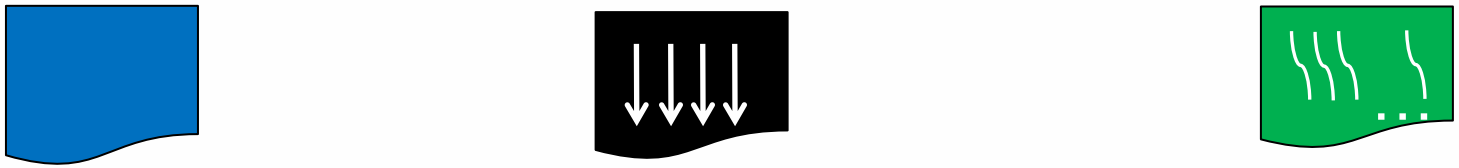
Optimization / Parallelization



Optimized Host Code

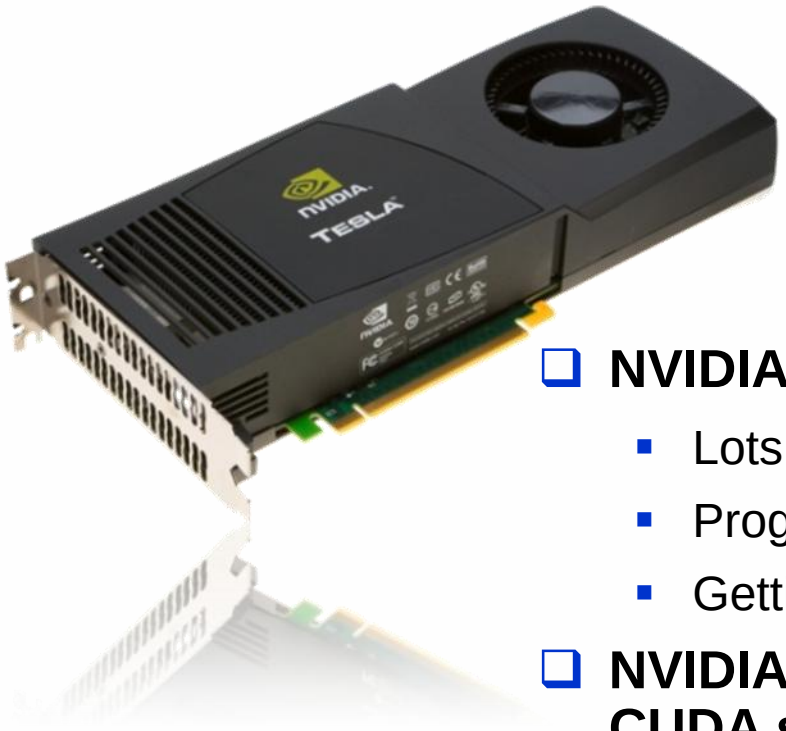
Parallel Multi-core Kernels

Massively Parallel GPU Kernels



Optimized CUDA C for Multi-core x86

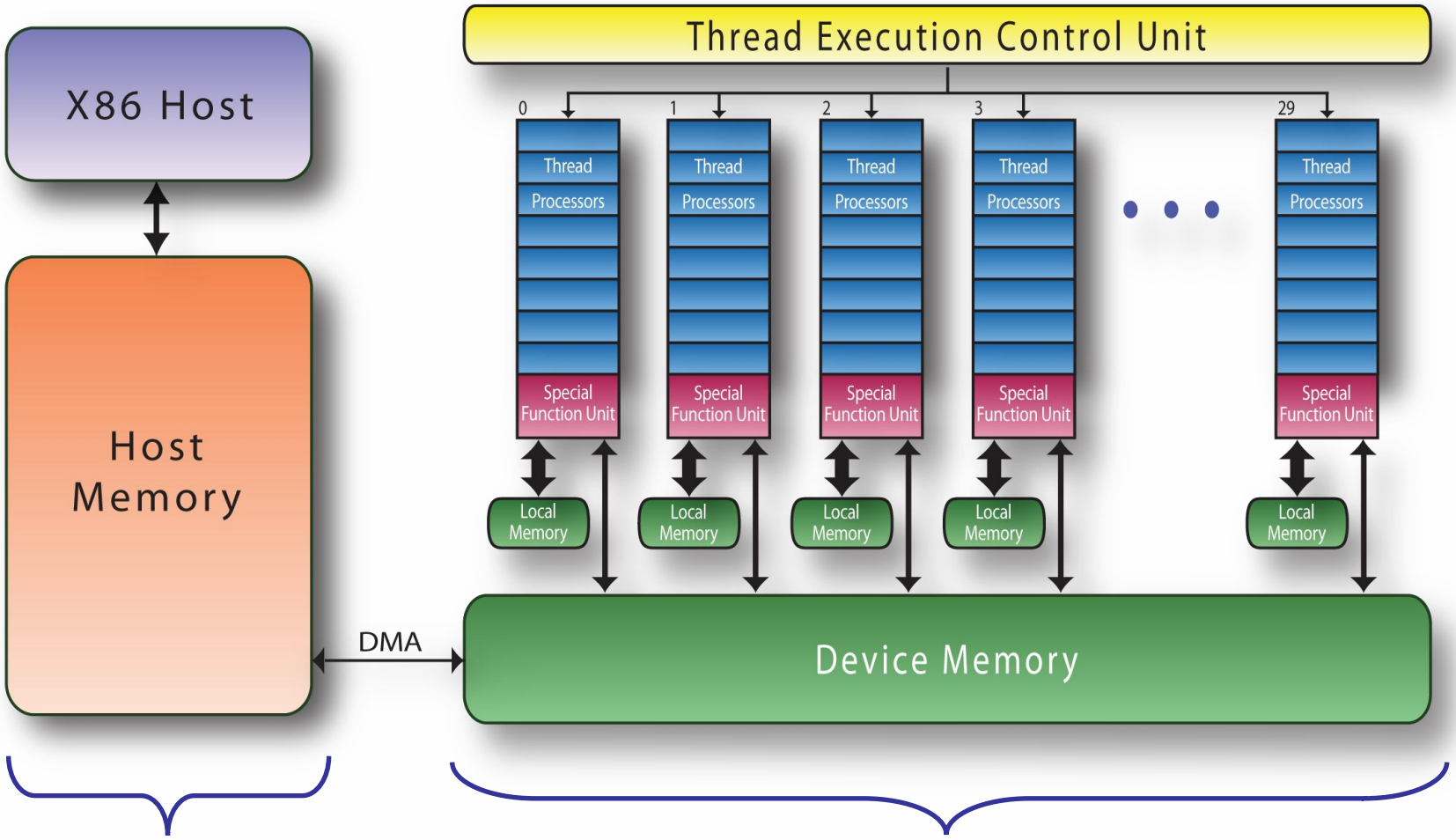
- Process CUDA C as a native parallel programming language for multi-core x86
- Inline Device kernel functions, translate chevron syntax to parallel/vector loops, use multiple cores and SSE/AVX instructions
- Execute each CUDA thread block using a single host core, eliminate synchronization where possible
- Host Code: all PGI optimizations for Intel/AMD host code will be supported
- Performance Goal: Well-structured CUDA C for multi-core x86 programs approach the efficiency of the same algorithm written in OpenMP



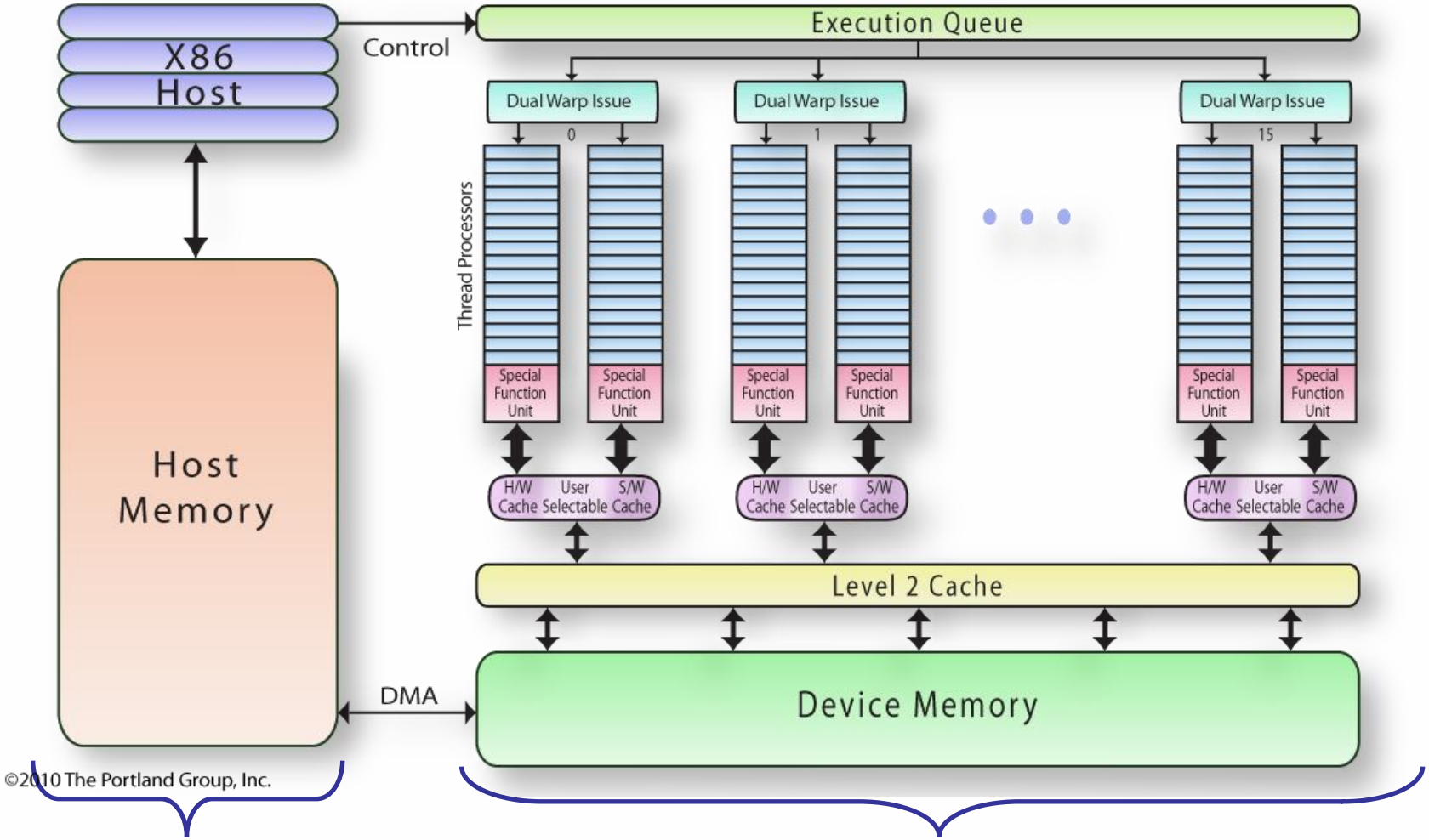
- **NVIDIA TESLA C1060/C2050**
  - Lots of available performance ~1 TFlops peak SP
  - Programming is a challenge
  - Getting high performance is lots of work
- **NVIDIA CUDA programming model and C for CUDA simplify GPGPU programming**
  - Much easier than OpenGL/DirectX, still challenging
- **PGI's CUDA Fortran provides an a Fortran based analog to CUDA C**
- **PGI's Accelerator Directive compilers for C and Fortran provide a higher level, OpenMP style of programming NVIDIA GPU's.**

Tesla C1060

Commodity Multicore x86 + Commodity Manycore GPUs



Tesla C2050 "Fermi"



4 – 24 CPU Cores

448/480/512 GPU/Accelerator Cores

Keys to performance on GPUs

- Lots of MIMD (Multiple Instruction, Multiple Data) parallelism to fill the multiprocessors
- Lots of SIMD (Single Instruction, Multiple Data) parallelism to fill cores on a multiprocessor
- High compute intensity (flops to memory access ratio)
- Minimize data movement between host and GPU
- Make use of memory hierarchy on the GPU
  - Local memory (thread registers)
  - Shared memory (local to an SM)
  - Constant memory (less necessary on the Fermi because of hardware cache)
- Make ample use of pinned memory on the host

CUDA Fortran 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

Fortran VADD on Host

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

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
```

Declare device array

Allocate device memory

Copy data to device

Launch a kernel

Copy data back from device

Deallocate device memory

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 callable from the CPU

device attribute implied

value vs. Fortran default

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

threadidx from 1..32

array bounds test

CUDA Fortran Language

- Host code
  - Declaring and allocating device memory
  - Moving data to and from device memory
  - Pinned memory
  - Launching kernels
- Kernel code
  - Attributes clause
  - Kernel subroutines, device subprograms
  - Shared memory
  - What is and what is not allowed in a kernel
  - CUDA Runtime API

Declaring Device Data

- 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

- 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 (10.5 or later)

Pinned Memory

- 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

Allocating Device Data

- 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

- 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 pinned memory will be faster

Using the API

```
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 )
```

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 )
```

- 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

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 type(dim3) variable
- The launch is asynchronous
  - host program continues, may issue other launches

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
  - Limited amount of shared memory available
  - shared among all threads in the same thread block

```
real, shared :: sm(16,16)
```

- 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**
  - assignment
  - do, if, goto, case
  - call (to device subprogram, must be inlined)
  - intrinsic function call (inlined)
  - where, forall

Modules and Scoping

- attributes(global) subroutine kernel in a module
  - can directly access device data in the same 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 module
  - can call device subroutines / functions in the same module
  - implicitly private
- attributes(global) subroutine kernel outside of a module
  - cannot directly access any global device data (just arguments)
- 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)

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

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
  - 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

Interoperability with CUDA C

- 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 with CUDA C

- 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_( a, x, y, n );
```

```
attributes(global) subroutine saxpy(a,x,y,n)
real, value :: a
real :: x(*), y(*)
integer, value :: n
```

Interoperability with CUDA C

- 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
  - This CUDA Fortran release uses CUDA 2.3 by default.
  - CUDA 3.0 is available in 10.4 or later (-Mcuda=cuda3.0)
  - CUDA 3.1 is available in 10.8 or later (-Mcuda=cuda3.1)

The PGI Accelerator Programming Model

Simple Matrix Multiply for an x64 Host

```
do j = 1, m
  do k = 1, p
    do i = 1,n
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

Parallel Matrix Multiply for a Multi-core x64 Host

```
!$omp parallel do
  do j = 1, m
    do k = 1, p
      do i = 1,n
        a(i,j) = a(i,j) + b(i,k)*c(k,j)
      enddo
    enddo
  enddo
```

PGI Directive-based Matrix Multiply for x64+GPU

```
!$acc region
      do j = 1, m
        do k = 1, p
          do i = 1,n
            a(i,j) = a(i,j) + b(i,k)*c(k,j)
          enddo
        enddo
      enddo
!$acc end region
```

PGI Accelerator Compilers

```
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

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
        …
```

+

GPU/Accelerator Code

```
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

Important Terms

- **Accelerator Directive**
  - **in C: #pragma acc ...**
  - **in Fortran: !$acc ...**
- **Region**
  - **structured block (single entry/single exit)**
  - **in C: surrounded by braces {}**
  - **in Fortran: surrounded by begin/end directives**
- **Compute Region**
  - **Region with loops targeted for accelerator**
- **Kernel**
  - **Code executed on accelerator**
  - **Kernel executes on a multidimensional rectangular parallel domain**

Performance Goals

- **Data movement between Host and Accelerator**
  - Minimize amount of data moved, number of data moves, frequency of data moves
  - Maximize bandwidth 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 and hide long device memory latency

Performance Goals (2)

- 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 (CUDA “shared” memory)
- 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

Getting Started

- Install a CUDA-enabled NVIDIA GPU
- Install latest CUDA-enabled NVIDIA driver
- Install the PGI 10.0 or later Compilers
- Test connection to GPU
  pgaccelinfo ; pgcpuid
- Try sample programs
  cd testdir
  cp /opt/pgi/linux86-64/10.8/etc/samples .
  make f1.exe c1.exe

Building Accelerator Programs

- **pgfortran -ta=nvidia a.f90**
- **pgcc -ta=nvidia a.c**
- Sub-options, host option:
  - **-ta=nvidia:{analysis | nofma | keepbin | keepptx | keepgpu | maxregcount:<n> | cc10 | cc11 | cc13 | cc20 | fastmath | mul24 | cuda2.3| cuda3.0 | cuda3.1 | time}**
  - **-ta=host**
- Build for GPU or host execution with **-ta=nvidia,host**
- Enable compiler feedback with **-Minfo** or **-Minfo=accel**
- Must have appropriate gcc for preprocessor (Linux,OSX)
  - CL, NVCC tools bundled with compiler, cross-dev environment

PGI Accelerator Directives and Pragmas

PGI Accelerator Directives

- C
  - #pragma acc *directive-name* [*clause* [[,]*clause*]... \
    continuation to next line
- Fortran
  - Fortran 90 free form

    !$acc *directive-name* [*clause* [[,]*clause*]... &
    continuation to next line

  - Fortran 77 fixed form

    !$acc *directive-name* [*clause* [[,]*clause*]...
    !$acc* *continuation to next line*

Accelerator 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
conditional execution

- C

```
#pragma acc region if(n > 100)
{
   for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

```
!$acc region if( n.gt.100 )
   do i = 1,n
      r(i) = a(i) * 2.0
   enddo
!$acc end region
```

Compute Region Clauses
data copy clauses

- C

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

- Fortran

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

Compute Region Clauses
data copy clauses – array sections

- C

```
#pragma acc region copyin(a[0:n-1])
{
   for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

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

Compute Region Clauses
data copy clauses - multiple

- C

```
#pragma acc region copyin(a[0:n-1]) copyout(r)
{
   for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

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

Compute Region Clauses
data copy clauses – multi-D arrays

- C

```
#pragma acc region copyin(a[0:n-1][0:m-1])
{
   for( i = 1; i < n-1; ++i )
     for( j = 1; j < m-1; ++j )
       r[i][j] = a[i][j]*2.0f;
}
```

- Fortran

```
!$acc region copyin(a(1:m,1:n))
   do i = 2,n-1
      do j = 2,m-1
         r(j,i) = a(j,i) * 2.0
      enddo
   enddo
!$acc end region
```

Compute Region Clauses
data copy clauses – bidirectional copy

- C

```
#pragma acc region copyin(a[0:n-1][0:m-1]) \
                  copy(r[0:n-1][0:m-1])
{
   for( i = 1; i < n-1; ++i )
     for( j = 1; j < m-1; ++j )
       r[i][j] = a[i][j]*2.0f;
}
```

- Fortran

```
!$acc region copyin(a(1:m,1:n)) copy(r(:,:))
   do i = 2,n-1
      do j = 2,m-1
         r(j,i) = a(j,i) * 2.0
      enddo
   enddo
!$acc end region
```

Compute Region Clauses
device local data

```
!$acc region copyin(a(1:m,1:n)) local(r)
    do times = 1,niters
     do i = 2,n-1
       do j = 2,m-1
          r(j,i) = 0.25*(a(j-1,i)+a(j,i-1)+a(j+1,i)+a(j,i+1))
       enddo
     enddo
     do i = 2,n-1
       do j = 2,m-1
          a(j,i) = r(j,i)
       enddo
     enddo
   enddo
!$acc end region
```

Compute Region Clauses
device local data – array sections

```
!$acc region copyin(a(1:m,1:n)) local(r(2:m-1,2:n-1))
    do times = 1, niters
      do i = 2,n-1
        do j = 2,m-1
          r(j,i) = 0.25*(a(j-1,i)+a(j,i-1)+a(j+1,i)+a(j,i+1))
        enddo
      enddo
      do i = 2,n-1
        do j = 2,m-1
          a(j,i) = r(j,i)
        enddo
      enddo
    enddo
!$acc end region
```

Compute Region Clauses
when does data copy actually occur?

- C

```
#pragma acc region copyin(a[0:n-1][0:m-1])
  /* data copied to Accelerator here */
{
    for( i = 1; i < n-1; ++i )
      for( j = 1; j < m-1; ++j )
        r[i][j] = a[i][j]*2.0f;
} /* data copied to Host here */
```

- Fortran

```
!$acc region copyin(a(1:m,1:n))
  ! data copied to Accelerator here
    do i = 2,n-1
      do j = 2,m-1
        r(j,i) = a(j,i) * 2.0
      enddo
    enddo
  ! data copied to Host here
!$acc end region
```

What can appear in an Accelerator Compute Region?

- Arithmetic
  - C: int, float, double
  - F: integer, real, double precision, complex
- Loops, IFs
  - Kernel loops must be rectangular: trip count is invariant
- Obstacles with C
  - unbound pointers – use restrict keyword, or –Msafeptr, or

  –Mipa=fast
  - default is double – use float constants (0.0f), or –Mfcon, and float intrinsics
- Obstacles with Fortran
  - Fortran pointer attribute is not supported

Supported C Intrinsics

- C: #include <accelmath.h>

acos	asin	atan	atan2
cos	cosh	exp	fabs
fmax	fmin	log	log10
pow	sin	sinh	sqrt
tan	tanh		
acosf	asinf	atanf	atan2f
cosf	coshf	expf	fabsf
fmaxf	fminf	logf	log10f
powf	sinf	sinhf	sqrtf
tanf	tanhf		

Supported Fortran Intrinsics

abs	acos	aint	asin
atan	atan2	cos	cosh
dble	exp	iand	ieor
int	ior	log	log10
max	min	mod	nint
not	real	sign	sin
sinh	sqrt	tan	tanh

Device Resident Data

- Accelerator data region
  - moves data, like a compute region
  - compute regions within a data region will not need to move that data
  - subprogram argument passing may include hidden pointers to accelerator copies
- Implicit data region for each function, subroutine, program
- Leave data on the device between accelerator compute regions, even across subroutine boundaries

Accelerator 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 )`**
  - **`updateout( list )`**
  - Data must be in a data allocate clause for an enclosing data region

Accelerator Data Region

- !$acc data region copy(a)

```
...
!$acc region copyin(b)
  do i = 1,n
    a(i) = a(i) * b(i)
  enddo
!$acc end region
...
!$acc end data region
```

- copy, copyin, copyout, local clauses allowed on data region

Implicit Data Region

- subroutine sub(a)
  real a(:)
  !$acc copy(a)
  ...
  !$acc region copyin(b)
  do i = 1,n
  a(i) = a(i) * b(i)
  enddo
  !$acc end region
  ...
  end subroutine
- copy, copyin, copyout, local data directives allowed
- list may include any variable or array

Update directive

- !$acc data region copyin(a), local(b)
  call getvalues(b)
  !$acc updatein(b)
  !$acc region
  do i = 1,n
  a(i) = a(i) * b(i)
  enddo
  !$acc end region
  !$acc updateout(a)
  call outputvalues(a)
  !$acc end data region
- Executable directive to copy data to/from device-resident arrays

Data Region Clauses
Mirroring Fortran Allocatables on the Device

- Fortran mirror clause for allocatable arrays
- float, dimension(:), allocatable :: a

```
!$acc data region mirror(a)
  allocate(a(1:n))
  !$acc region copyin(b)
    do i = 1,n
      a(i) = b(i) * 2.0
    enddo
  !$acc end region
  ...
  deallocate(a)
!$acc end data region
```

- Mirrored arrays will match allocation state on host and accelerator

Data Region Clauses
Passing Device Copies as Arguments

```
      subroutine sub( a, b )
      real :: a(:), b(:)
!$acc reflected(a)
!$acc region copyin(b)
      do i = 1,n
        a(i) = a(i) * b(i)
      enddo
!$acc end region
      ...
      end subroutine

      subroutine bus(x, y)
      real :: x(:), y(:)
!$acc data region copy(x)
      call sub( x, y )
      ...
```

Passing Device Copies

- REFLECTED clause only available in Fortran
- List of argument arrays
- Caller must have a visible device copy at the call site
- Subprogram interface must be explicit
  - interface block or module
- Compiler will pass a hidden argument corresponding to the device copy

*NOTE: MIRROR & REFLECTED are not supported in PGI 10.8! They are expected to be available in PGI 2011

PGI Accelerator Region Clauses
Summary

Clause	Availability	Region Scope
if (*cond*)	PGI 10.0	compute
copy (*list*)	PGI 10.0	compute, data, declaration
copyin (*list*)	PGI 10.0	compute, data, declaration
copyout (*list*)	PGI 10.0	compute, data, declaration
local (*list*)	PGI 10.0	compute, data, declaration
mirror (*list*)	Est PGI 11.0	data, decl (Fortran)
reflected(*list*)	Est PGI 11.0	compute, data, decl (Fortran)
updatein(list)	PGI 10.0	compute, data, executable
updateout(list)	PGI 10.0	compute, data, executable

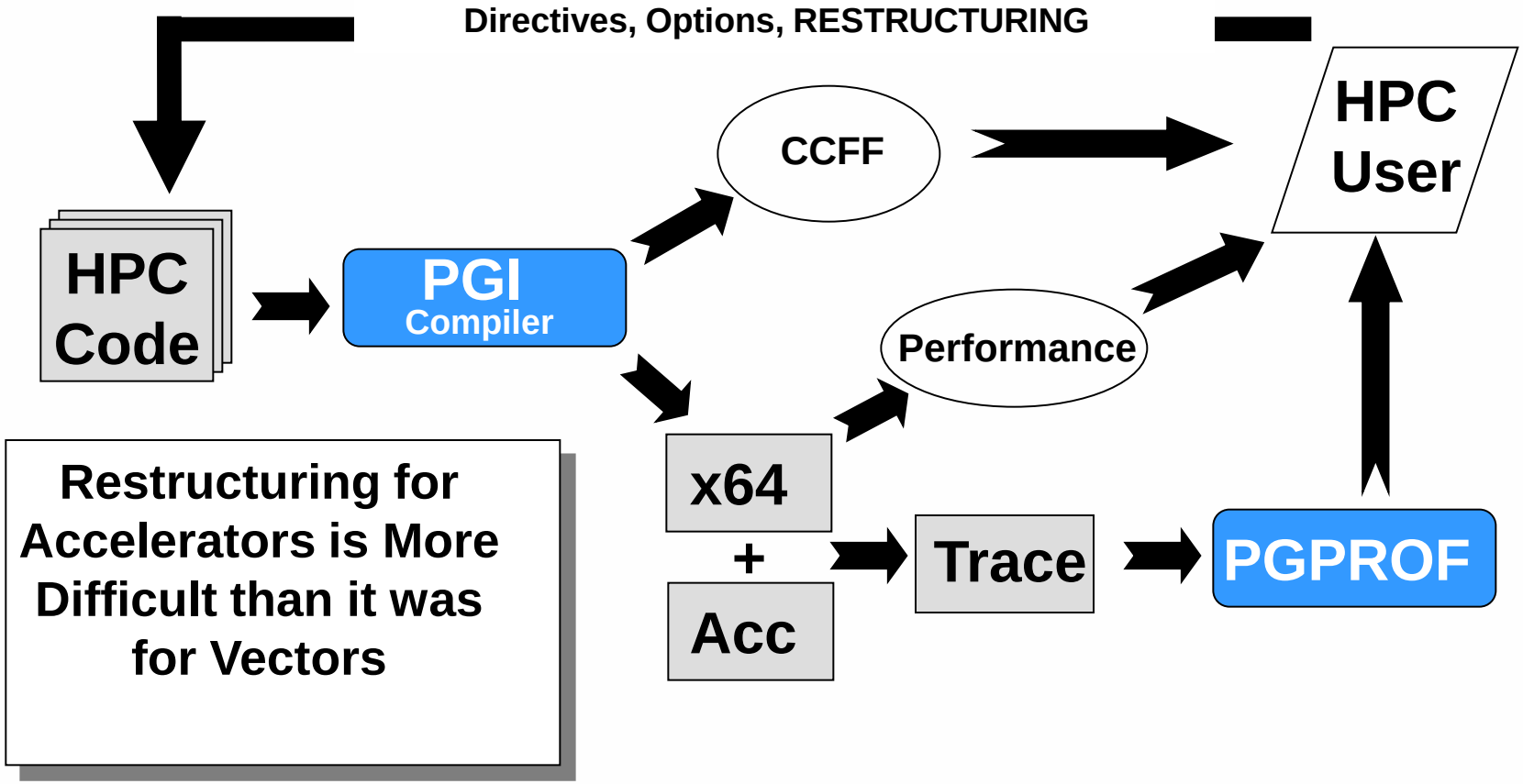
PGI Accelerator Loop Mapping Clauses

Summary

Clause	Availability	Scope
host [(*width*)]	PGI 10.0	loop
parallel [(*width*)]	PGI 10.0	loop
seq [(*width*)]	PGI 10.0	loop
vector [(*width*)]	PGI 10.0	loop
private (*list*)	PGI 10.0	loop
kernel	PGI 10.0	loop
unroll (*width*)	Est. PGI 11.0	loop
cache (*list*)	Est. PGI 11.0	loop

Understanding & Using Compiler Feedback

Programmer Productivity: Compiler-to-Programmer Feedback



Compiler Feedback Messages

- **Data related**
  - Generating copyin(b(1:n,1:m))
  - Generating copyout(b(2:n-1,2:m-1))
  - Generating copy(a(1:n,1:n))
  - Generating local(c(1:n,1:n))
- **Loop or kernel related**
  - Loop is parallelizable
  - Accelerator kernel generated
- **Barriers to GPU code generation**
  - No parallel kernels found, accelerator region ignored
  - Loop carried dependence due to exposed use of ... prevents parallelization
  - Parallelization would require privatization of array ...

Compiler Messages Continued

- Memory optimization related
  - Cached references to size [18x18] block of ‘b’
  - Non-stride-1 memory accesses for ‘a’

Availability and Additional Information

- **PGI Accelerator Programming Model** – supported for x64+NVIDIA targets in the PGI 2010 Fortran and C compilers; see www.pgroup.com/accelerate for a detailed specification of the PGI Accelerator model, an FAQ, and related articles and white papers

- **CUDA Fortran** – supported on NVIDIA GPUs in PGI 2010 Fortran 95/03 compiler; see http://www.pgroup.com/resources/cudafortran.htm for a detailed specification

- **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

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