Computing with Interlagos and Nvidia processors

Titan Summit, Aug 15-17, 2011

Dave Norton, Craig Toepfer
dave.norton@pgroup.com
craig.toepfer@pgroup.com
http://www.pgroup.com

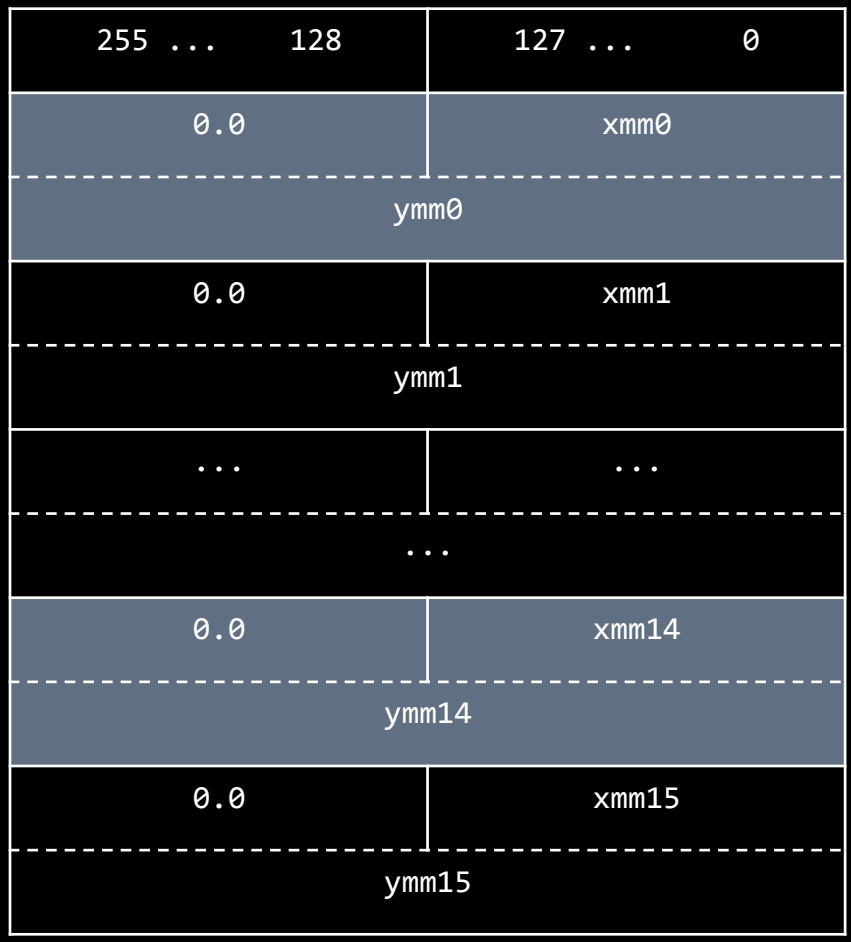
What is AVX?

Before VEX:

```
movsd (%rax, %r9), %xmm0
movsd (%rax, %r8), %xmm1
movsd %xmm1, %xmm2
addsd %xmm0, %xmm2
```

After VEX:

```
vmovsd  (%rax, %r9), %xmm0
vmovsd  (%rax, %r8), %xmm1
vaddsd %xmm0, %xmm1, %xmm2
```



Importance of Vectorization

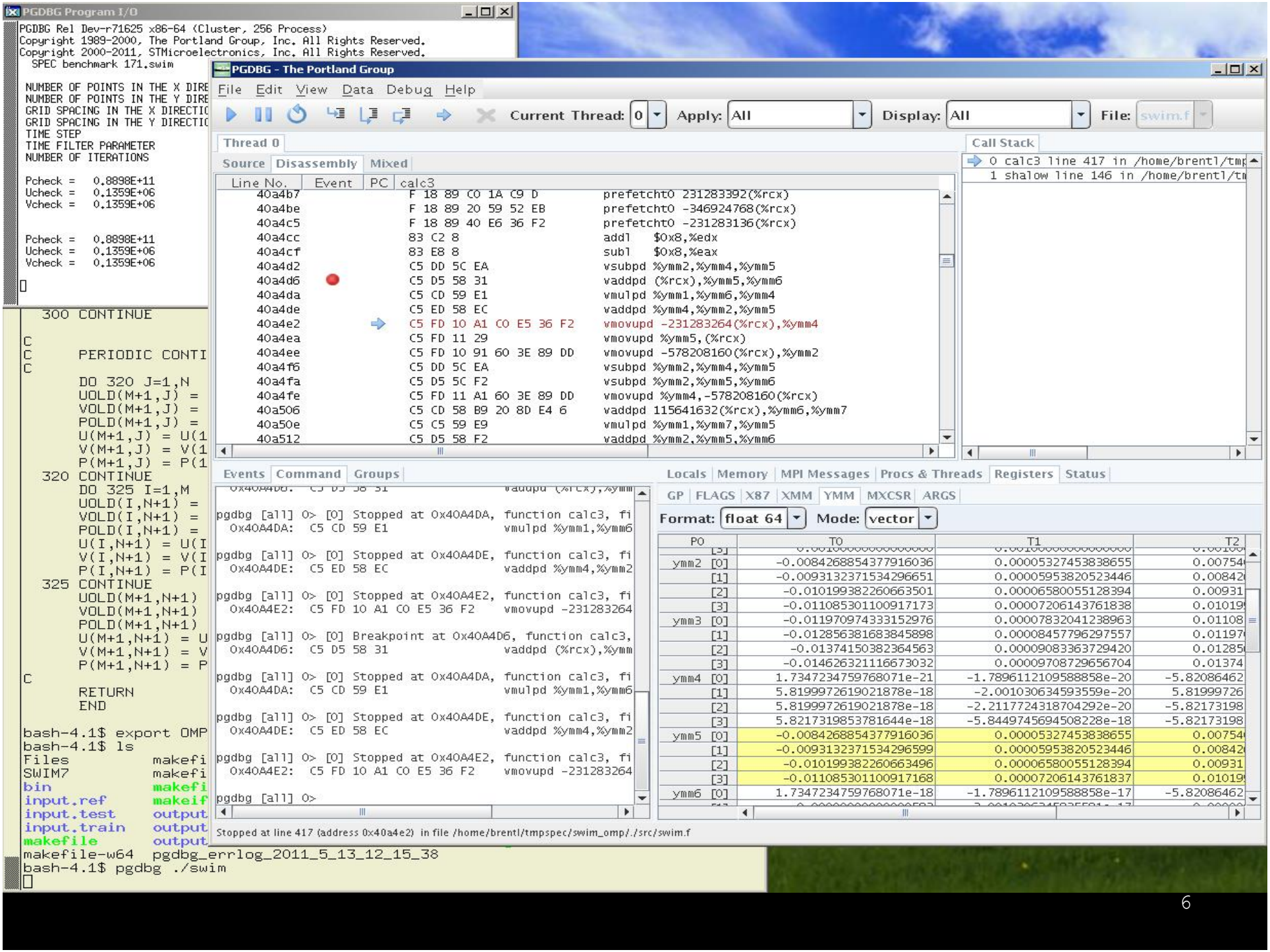
	255...192	191...128	127...64	63...0
	ymm0[3]	ymm0[2]	ymm0[1]	ymm0[0]
vbroadcast	a	a	a	a
vmovapd	x[3]	x[2]	x[1]	x[0]
vmulpd	a*x[3]	a*x[2]	a*x[1]	a*x[0]
vmovapd	y[3]	y[2]	y[1]	y[0]
vaddpd	a*x[3]+y[3]	a*x[2]+y[2]	a*x[1]+y[1]	a*x[0]+y[0]
vmovapd	y[3]	y[2]	y[1]	y[0]

Know Your Target Processors

- AMD Bulldozer PGI target processor flag : –tp bulldozer
- Specify size of SIMD instructions : -Mvect=simd:[128|256]
- Enable/Disable generation of FMA instructions: -[no]fma
- Running FMA4 code on anything but Bulldozer will yield:
  Illegal instruction (core dumped)
- Make use of PGI Unified Binary technology to produce optimal code paths for multiple x64 architectures within a single executable.

vzeroupper instruction generation

- This instruction zeroes out the upper 128 bits of all the ymm registers and marks them as clean.
- If you mix 256-bit AVX instructions with legacy SSE instructions that use xmm registers, you will incur performance penalties of roughly one hundred cycles at the transition points.
- The PGI compiler currently generates the vzeroupper instruction right before a call is made. This is because we cannot be sure how the callee has been compiled.
- When compiling functions that perform AVX instruction sequences, the PGI compiler generates a vzeroupper instruction right before returning, again because we cannot make assumptions about how the caller was compiled.


PGDBG Program I/O
PGDBG Rel Dev-r71625 x86-64 (Cluster, 256 Process)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2011, STMicroelectronics, Inc. All Rights Reserved.
SPEC benchmark 171.swim
NUMBER OF POINTS IN THE X DIRE
NUMBER OF POINTS IN THE Y DIRE
GRID SPACING IN THE X DIRECTIO
GRID SPACING IN THE Y DIRECTIO
TIME STEP
TIME FILTER PARAMETER
NUMBER OF ITERATIONS
Pcheck = 0.8898E+11
Ucheck = 0.1359E+06
Vcheck = 0.1359E+06
Pcheck = 0.8898E+11
Ucheck = 0.1359E+06
Vcheck = 0.1359E+06
PGDBG - The Portland Group
File Edit View Data Debug Help
Current Thread: 0 Apply: All Display: All File: swim.f
Thread 0
Source Disassembly Mixed
Line No. Event PC calc3
40a4b7 F 18 89 C0 1A C9 D prefetcht0 231283392(%rcx)
40a4be F 18 89 20 59 52 EB prefetcht0 -346924768(%rcx)
40a4c5 F 18 89 40 E6 36 F2 prefetcht0 -231283136(%rcx)
40a4cc 83 C2 8 addl $0x8,%edx
40a4cf 83 E8 8 subl $0x8,%eax
40a4d2 C5 DD 5C EA vsubpd %ymm2,%ymm4,%ymm5
40a4d6 C5 D5 58 31 vaddpd (%rcx),%ymm5,%ymm6
40a4da C5 CD 59 E1 vmulpd %ymm1,%ymm6,%ymm4
40a4de C5 ED 58 EC vaddpd %ymm4,%ymm2,%ymm5
40a4e2 C5 FD 10 A1 C0 E5 36 F2 vmovupd -231283264(%rcx),%ymm4
40a4ea C5 FD 11 29 vmovupd %ymm5,(%rcx)
40a4ee C5 FD 10 91 60 3E 89 DD vmovupd -578208160(%rcx),%ymm2
40a4f6 C5 DD 5C EA vsubpd %ymm2,%ymm4,%ymm5
40a4fa C5 D5 5C F2 vsubpd %ymm2,%ymm5,%ymm6
40a4fe C5 FD 11 A1 60 3E 89 DD vmovupd %ymm4,-578208160(%rcx)
40a506 C5 CD 58 B9 20 8D E4 6 vaddpd 115641632(%rcx),%ymm6,%ymm7
40a50e C5 C5 59 E9 vmulpd %ymm1,%ymm7,%ymm5
40a512 C5 D5 58 F2 vaddpd %ymm2,%ymm5,%ymm6
Call Stack
0 calc3 line 417 in /home/brentl/tmp
1 shalow line 146 in /home/brentl/tm
300 CONTINUE
C
C PERIODIC CONTI
C
DO 320 J=1,N
UOLD(M+1,J) =
VOLD(M+1,J) =
POLD(M+1,J) =
U(M+1,J) = U(1
V(M+1,J) = V(1
P(M+1,J) = P(1
320 CONTINUE
DO 325 I=1,M
UOLD(I,N+1) =
VOLD(I,N+1) =
POLD(I,N+1) =
U(I,N+1) = U(I
V(I,N+1) = V(I
P(I,N+1) = P(I
325 CONTINUE
UOLD(M+1,N+1)
VOLD(M+1,N+1)
POLD(M+1,N+1)
U(M+1,N+1) = U
V(M+1,N+1) = V
P(M+1,N+1) = P
C
RETURN
END
bash-4.1$ export OMP
bash-4.1$ ls
Files makefi
SWIM7 makefi
bin makefi
input.ref makeif
input.test output
input.train output
makefile output
makefile-w64 pgdbg_errlog_2011_5_13_12_15_38
bash-4.1$ pgdbg ./swim
Events Command Groups
pgdbg [all] 0> [0] Stopped at 0x40A4DA, function calc3, fi
0x40A4DA: C5 CD 59 E1 vmulpd %ymm1,%ymm6
pgdbg [all] 0> [0] Stopped at 0x40A4DE, function calc3, fi
0x40A4DE: C5 ED 58 EC vaddpd %ymm4,%ymm2
pgdbg [all] 0> [0] Stopped at 0x40A4E2, function calc3, fi
0x40A4E2: C5 FD 10 A1 C0 E5 36 F2 vmovupd -231283264
pgdbg [all] 0> [0] Breakpoint at 0x40A4D6, function calc3,
0x40A4D6: C5 D5 58 31 vaddpd (%rcx),%ymm
pgdbg [all] 0> [0] Stopped at 0x40A4DA, function calc3, fi
0x40A4DA: C5 CD 59 E1 vmulpd %ymm1,%ymm6
pgdbg [all] 0> [0] Stopped at 0x40A4DE, function calc3, fi
0x40A4DE: C5 ED 58 EC vaddpd %ymm4,%ymm2
pgdbg [all] 0> [0] Stopped at 0x40A4E2, function calc3, fi
0x40A4E2: C5 FD 10 A1 C0 E5 36 F2 vmovupd -231283264
pgdbg [all] 0>
Locals Memory MPI Messages Procs & Threads Registers Status
GP FLAGS X87 XMM YMM MXCSR ARGS
Format: float 64 Mode: vector
P0 T0 T1 T2
ymm2 [0] -0.0084268854377916036 0.00005327453838655 0.00754
[1] -0.0093132371534296651 0.00005953820523446 0.00842
[2] -0.010199382260663501 0.00006580055128394 0.00931
[3] -0.011085301100917173 0.00007206143761838 0.01019
ymm3 [0] -0.011970974333152976 0.00007832041238963 0.01108
[1] -0.012856381683845898 0.00008457796297557 0.01197
[2] -0.01374150382364563 0.00009083363729420 0.01285
[3] -0.014626321116673032 0.00009708729656704 0.01374
ymm4 [0] 1.7347234759768071e-21 -1.7896112109588858e-20 -5.82086462
[1] 5.8199972619021878e-18 -2.001030634593559e-20 5.81999726
[2] 5.8199972619021878e-18 -2.2117724318704292e-20 -5.82173198
[3] 5.8217319853781644e-18 -5.8449745694508228e-18 -5.82173198
ymm5 [0] -0.0084268854377916036 0.00005327453838655 0.00754
[1] -0.0093132371534296599 0.00005953820523446 0.00842
[2] -0.010199382260663496 0.00006580055128394 0.00931
[3] -0.011085301100917168 0.00007206143761837 0.01019
ymm6 [0] 1.7347234759768071e-18 -1.7896112109588858e-17 -5.82086462
Stopped at line 417 (address 0x40a4e2) in file /home/brentl/tmpspec/swim_omp/./src/swim.f

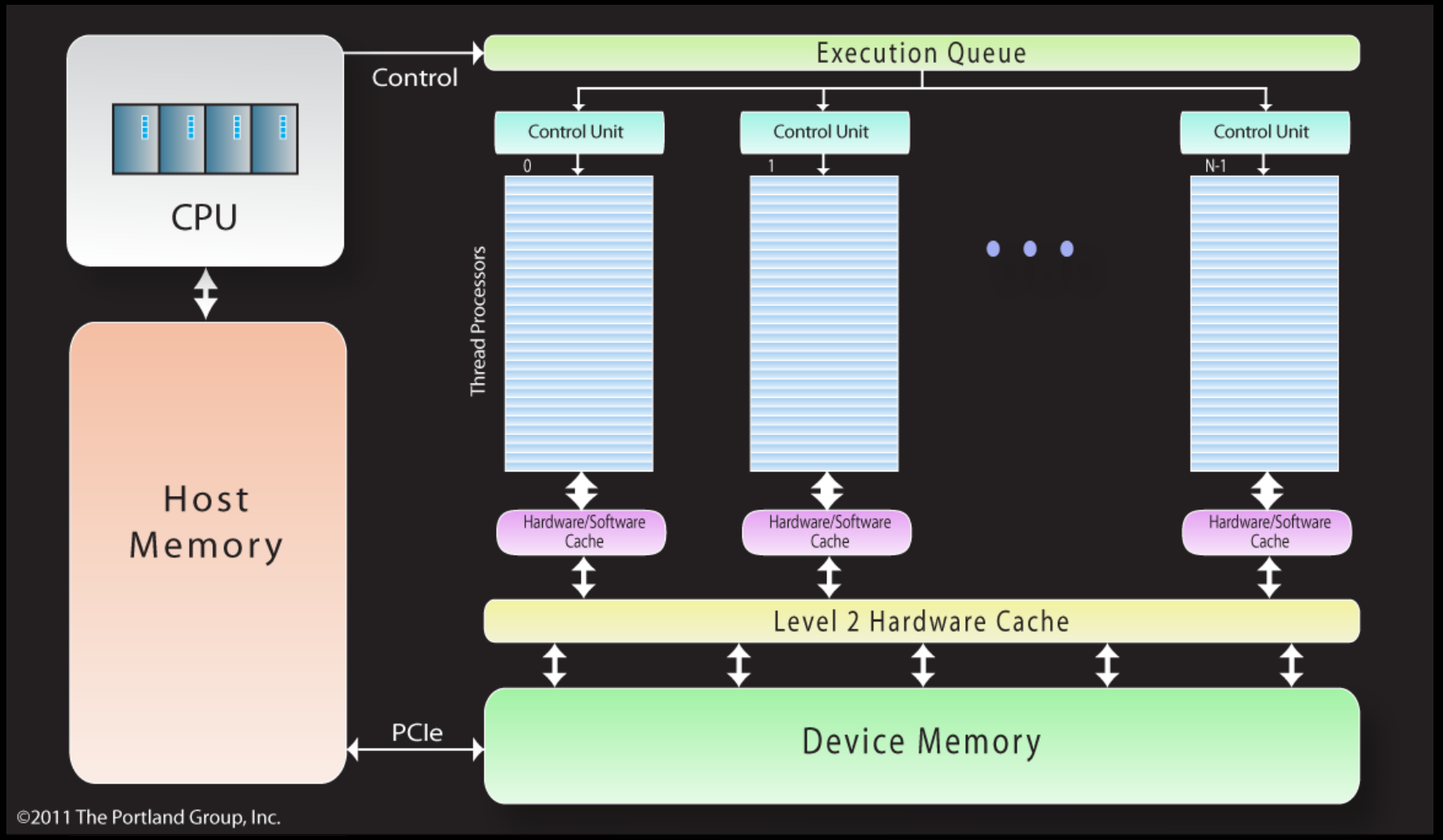
Questions/Comments about programming with AVX on Interlagos processors?

PGI

- C99, C++, F2003 Compilers
  - Optimizing
  - Vectorizing
  - Parallelizing
- Graphical parallel tools
  - PGDBG® debugger
  - PGPROF® profiler
- AMD, Intel, NVIDIA
- 64-bit / 32-bit
- PGI Unified Binary™
- Linux, MacOS, Windows
- Visual Studio integration
- GPGPU Features
  - CUDA Fortran/C/C++
  - PGI Accelerator™
  - CUDA-x86

www.pgroup.com

The emerging HPC architecture is multi-core x64 + manycore GPUs



Candidate GPU Codes

- Code has lots of SPMD type parallelism
- Performance profile:
  - Several "hot spots" that can make good use of acceleration
  - Flat profile with significant parallel computing in successive subroutines

Basic Porting Approach

- Allocate arrays on GPU
- Move data from host to GPU
- Launch compute kernel on GPU
- Move results from GPU to host
- Deallocate arrays on GPU

What is CUDA Fortran?

- CUDA Fortran is an analog to NVIDIA's C for CUDA
- CUDA Fortran was co-defined by PGI and NVIDIA and implemented in the PGI 2010 Fortran 95/03 compiler
- Includes support for the full CUDA programming model API and introduces intuitive Fortran language extensions to simplify host vs GPU data management
- Is supported on Linux, MacOS and Windows, including support within PGI Visual Fortran on Windows

Fortran?!? Really?

- Clear, straight-forward syntax
- Long legacy in the scientific community
- Large existing code base
- Semantics make it simpler to vectorize and parallelize
- Array descriptors have been implemented since F90 and allow high-level operations
- We can leverage descriptor extensibility to offload data and work to a GPU
- Modules add flexibility to overcome some CUDA limitations
- Fortran 2003 improves encapsulation, adds type extension and polymorphism
- New abstraction features, high-level syntax, along with a strongly-typed language lead to programmer productivity gains, with no sacrifice in performance

CUDA Fortran in 2 slides

```
subroutine vadd( A, B, C )
  use cudafor
  use kmod
  real, dimension(:) :: A, B
  real, pinned, dimension(:) :: C
  real, device, allocatable:: 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 VADD Device Code

```
module kmod
 use cudafor
contains
 attributes(global) subroutine vaddkernel(A,B,C,N)
  real, 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
```

CUDA Fortran
Matrix Multiply Host Routine

```
   subroutine mmul( A, B, C )                          ! Host routine to drive mmul_kernel
      use cudafor                                      ! Use module containing CUDA definitions
      real, dimension(:,:) :: A, B, C
                                                       ! Declare allocatable device arrays
      real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
      type(dim3) :: dimGrid, dimBlock                  ! Define thread grid, block shapes
! Begin execution
      N = size( A, 1 )
      M = size( A, 2 )
      L = size( B, 2 )
      allocate (Adev(N,M), Bdev(M,L), Cdev(N,L)) ! Allocate device arrays in GPU memory
      Adev = A(1:N,1:M)                                ! Copy input A to GPU device memory
      Bdev = B(1:M,1:L)                                ! Copy input B to GPU device memory
      dimGrid = dim3( N/16, M/16, 1 )                  ! Define thread grid dimensions
      dimBlock = dim3( 16, 16, 1 )                     ! Define thread block dimensions
                                                       ! Launch mmul_kernel on GPU
      call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, M, L)

      C(1:N,1:L) = Cdev                                ! Copy result C back to host memory
      deallocate( Adev, Bdev, Cdev )                   ! Free device arrays
   end subroutine mmul
end module mmul_mod
```

CUDA Fortran
Matrix Multiply GPU Kernel

```
  module mmul_mod                                  ! Module containing matrix multiply
  contains                                         !   CUDA Fortran GPU kernel
    attributes(global) subroutine mmul_kernel( A, B, C, N, M, L )
    real :: A(N,M), B(M,L), C(N,L)
    integer, value :: N, M, L
    integer :: i, j, kb, k, tx, ty
    real, shared :: Asub(16,16), Bsub(16,16)   ! Declare shared memory submatrix temps
    real :: Cij                                ! Declare C(i,j) temp for accumulations
! Begin execution
    tx = threadidx%x                           ! Get my thread indices
    ty = threadidx%y                           !
    i = blockidx%x * 16 + tx                   ! This thread computes
    j = blockidx%y * 16 + ty                   !   C(i,j) = sum(A(i,:) * B(:,j))
    Cij = 0.0
    do kb = 1, M, 16
      Asub(tx,ty) = A(i,ks+tx-1)               ! Each of 16x16 threads loads one
      Bsub(tx,ty) = B(ks+ty-1,j)               !   one element of ASUB & BSUB into
      call syncthreads()                       !   shared memory
      do k = 1,16                              ! Each thread accumulates length 16
        Cij = Cij + Asub(tx,k) * Bsub(k,ty)    !   partial dot product into its Cij
      enddo
      call syncthreads()
    enddo
    C(i,j) = Cij                               ! Each thread stores its element
                                               !   to the global C array
  end subroutine mmul_kernel                   ! End CUDA Fortran GPU kernel routine
  . . .
```

Threads

- Each thread is assigned a thread block index accessed through the built-in blockidx variable, and a thread index accessed through threadidx.
- The thread index may be a one-, two-, or three-dimensional index.
- In CUDA Fortran, the thread index for each dimension starts at one. A unique thread ID is assigned to each thread, computed from the thread index.
  - For a one-dimensional thread block, the thread index is equal to the thread ID.
  - For a two-dimensional thread block of size *(Dx,Dy), the thread ID is equal to (x+Dx (y-1)).*
  - *For a three-dimensional thread block of size (Dx,Dy,Dz), the thread ID is (x+Dx (y-1)+Dy(z-1)).*
- Threads in the same thread block may cooperate by using *shared memory, and by synchronizing at a barrier* using the SYNCTHREADS() intrinsic. Each thread in the block waits at the call to SYNCTHREADS() until all threads have reached that call. The shared memory acts like a low-latency, high bandwidth software managed cache memory.
- Currently, the maximum number of threads in a thread block is 1024 for Fermi.

Blocks

- A kernel may be invoked with many thread blocks, each with the same thread block size.
- The thread blocks are organized into a one- or two-dimensional *grid of blocks, so each thread has a thread index within the* block, and a block index within the grid.
- When invoking a kernel, the first argument in the chevron <<<>>> syntax is the grid size, and the second argument is the thread block size.
- Thread blocks must be able to execute independently; two thread blocks may be executed in parallel or one after the other, by the same core or by different cores. This behavior is controlled by the hardware rather then the programmer.
- There are currently a maximum of 65535blocks allowed. Beyond this, the programmer must stripmine data

Declaring Fortran 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 Fortran Module 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

Host Memory

- On the host side, the host program can directly access data in the host main memory.
- It can also directly copy data to and from the device global memory; such data copies require DMA access to the device, so are slow relative to the host memory.
- The host can also set the values in the device constant memory, again implemented using DMA access.

Device Memory

- On the device side, data in global device memory can be read or written by all threads.
- Data in constant memory space is initialized by the host program; all threads can read data in constant memory. Accesses to constant memory are typically faster than accesses to global memory, but it is read-only to the threads and limited in size. On Fermi – constant memory may not be faster then simply using the global device memory cache, however managing what data is in constant memory will free more space in the hardware cache.
- Threads in the same thread block can access and share data in shared memory; data in shared memory has a lifetime of the thread block.
- Each thread can also have private local memory; data in thread local memory may be implemented as processor registers or may be allocated in the global device memory; best performance will often be obtained when thread local data is limited to a small number of scalars that can be allocated as processor registers.

Subroutine/function attributes

- attributes(host) : The host attribute, specified on the subroutine or function statement, declares that the subroutine or function is to be executed on the host. Such a subprogram can only be called from another host subprogram. The default is attributes(host), if none of the host, global, or device attributes is specified.
- Attributes(global) : The global attribute may only be specified on a subroutine statement; it declares that the subroutine is a kernel subroutine, to be executed on the device, and may only be called from the host using a kernel call containing the chevron syntax and runtime mapping parameters.
- Attributes(device): The device attribute, specified on the subroutine or function statement, declares that the subprogram is to be executed on the device; such a routine must be called from a subprogram with the global or device attribute.

Variable Qualifiers

- Attributes (device) A variable with the device attribute is called a *device variable*, and is allocated in the device global memory.
  - A device array may be an explicit-shape array, an allocatable array, or an assumed-shape dummy array. An allocatable device variable has a dynamic lifetime, from when it is allocated until it is deallocated. Other device variables have a lifetime of the entire application.
- Attributes(constant) A variable with the constant attributes is called a *device constant variable.* Device constant variables are allocated in the device constant memory space. Device constant data may not be assigned or modified in any device subprogram, but may be modified in host subprograms. Device constant variables may not be allocatable, and have a lifetime of the entire application.

Variable Qualifiers (cont)

- Attributes(shared) A variable with the shared attributed is called a device shared variable or a *shared variable. A shared* variable may only be declared in a device subprogram, and may only be accessed within that subprogram, or by other device subprograms to which it is passed as an argument. A shared variable may not be data initialized.
- Attributes(pinned) A variable with the pinned attributes is called a *pinned variable. A pinned variable must be an allocatable* array. When a pinned variable is allocated, it will be allocated in host pagelocked memory. The advantage of using pinned variables is that copies from page-locked memory to device memory are faster than copies from normal paged host memory.

Allocating 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 host pinned memory will be faster
- Asynchronous copies currently require API interface

Concurrent Stream Execution

- Operations involving the device, including kernel execution and data copies to and from device memory, are implemented using stream queues. An operation is placed at the end of the stream queue, and will only be initiated when all previous operations on that queue have been completed.
- An application can manage more concurrency by using multiple streams.
- Each user-created stream manages its own queue; operations on different stream queues may execute out-of-order with respect to when they were placed on the queues, and may execute concurrently with each other.
- The default stream, used when no stream identifier is specified, is stream zero; stream zero is special in that operations on the stream zero queue will begin only after all preceding operations on all queues are complete, and no subsequent operations on any queue begin until the stream zero operation is complete.

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
- The launch is asynchronous
  - host program continues, may issue other launches

CUDA Errors

- Out of memory
- Launch failure (array out of bounds, …)
- No device found
- Invalid device code (compute capability mismatch)

Test for error:

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

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

Writing a CUDA Kernel (1)

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

Writing a CUDA Kernel (2)

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

Writing a CUDA Kernel (3)

- Fortran disallowed statements include
  - read, write, print, open, close, inquire, format, other IO except now some limited support for list-directed (print *)
  - allocate, deallocate
  - Fortran pointer assignment, pointers in general
  - recursive procedure calls, direct or indirect
  - ENTRY statement, optional arguments, alternate return
  - SAVEd data
  - assigned goto, ASSIGN statement
  - stop, pause

!$CUF Kernel Loop Directive

- CUDA Fortran allows automatic kernel generation and invocation from a region of host code containing one or more tightly nested loops.
- Launch configuration and mapping of the loop iterations onto the hardware is controlled and specified as part of the directive body using the familiar CUDA chevron syntax.
- As with any kernel, the launch is asynchronous. The program can use cudaThreadSynchronize() or CUDA Events to wait for the completion of the kernel.
- The work in the loops specified by the directive is executed in parallel, across the thread blocks and grid; it is the programmer's responsibility to ensure that parallel execution is legal and produces the correct answer.
- The one exception to this rule is a scalar reduction operation, such as summing the values in a vector or matrix. For these operations, the compiler handles the generation of the final reduction kernel, inserting synchronization into the kernel as appropriate

!$CUF kernel examples

- The general form of the kernel directive is:

  ```
  !$cuf kernel do[(n)] <<< grid, block >>>
  ```

- The compiler maps the launch configuration specified by the grid and block values onto the outermost *n loops*, starting at loop n and working out. The grid and block values can be an integer scalar or a parenthesized list. Alternatively, using asterisks tells the compiler to choose a thread block shape and/or compute the grid shape from the thread block shape and the loop limits.

```
!$cuf kernel do(2) <<< (*,*), (32,4) >>>
do j = 1, m
   do i = 1, n
      a(i,j) = b(i,j) + c(i,j)
   end do
end do
```

Using the CUF Kernel directive

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

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

db = b
dc = c

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

a = da

deallocate( da, db, dc )
```

Building a CUDA Fortran Program

- CUDA Fortran is supported by the PGI Fortran compilers when the filename uses a CUDA Fortran extension. The .cuf extension specifies that the file is a free-format CUDA Fortran program;
- The .CUF extension may also be used, in which case the program is processed by the preprocessor before being compiled.
- To compile a fixed-format program, add the command line option –Mfixed.
- CUDA Fortran extensions can be enabled in any Fortran source file by adding the –Mcuda command line option.

CUDA C vs CUDA Fortran

- CUDA C
  - supports Runtime API
  - supports Driver API
  - cudaMalloc, cudaFree
  - cudaMemcpy
  - OpenGL interoperability
  - Direct3D interoperability
  - Supports texture memory
  - arrays zero-based
  - threadidx/blockidx zero-based
  - unbound pointers
  - pinned allocate routines
- CUDA Fortran
  - 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

Generic interfaces and overloading

Allows programmers to define Fortran-like operations:

```
module dev_transpose
  interface transpose
    module procedure real4devxspose
    module procedure int4devxspose
  end interface
  contains
    function real4devxpose(adev) result(b)
    real, device :: adev(:,:)
    real  b(ubound(adev,2),ubound(adev,1))
    <add your choice of transpose kernel>
    return
    end
end module dev_transpose
```

At the site of the function reference, the look is normal Fortran:

```
subroutine s1(a,b,n,m)
use dev_transpose
real, device :: a(n,m)
real b(m,n)
b = transpose(a)
end
```

BLAS overloading

```
use cublas

real(4), device :: xd(N)
real(4) x(N)
call random_number(x)

! On the device
allocate(xd(N))
xd = x
j = isamax(N,xd,1)

! On the host, same name
k = isamax(N,x,1)
```

```
module cublas
! isamax
  interface isamax
    integer function isamax &
       (n, x, incx)
    integer :: n, incx
    real(4) :: x(*)
    end function

    integer function isamaxcu &
       (n, x, incx) bind(c, &
            name='cublasIsamax')
    integer, value :: n, incx
    real(4), device :: x(*)
    end function
  end interface
. . .
```

Calling CUDA Fortran subroutines from PGI Accelerator programs

```
use cublas

!$acc data region copyin(x)

! some compute regions . . .

k = isamax(N,x,1)

! maybe some other compute regions. . .

!$acc end data region
```

- You can call CUDA global routines by creating explicit interfaces to host-resident data and device-resident data specific functions for a generic call

Object-oriented features

Type extension allows polymorphism:

```
type dertype
integer  id, iop, npr
real, allocatable :: rx(:)
contains
  procedure :: init => init_dertype
  procedure :: print => print_dertype
  procedure :: find  => find_dertype
end type dertype

type, extends(dertype) :: extdertype
real, allocatable, device :: rx_d(:)
contains
  procedure :: init  => init_extdertype
  procedure :: find  => find_extdertype
end type extdertype
```

The class statement allows arguments of the base or extended type:

```
subroutine init_dertype(this, n)
class(dertype) :: this
```

You optimize data movement in CUDA Fortran by manipulating F90 syntax and/or inserting API calls

- Can you eliminate Host/Device array assignments?
- Can you place host data in Pinned memory?
- Can you re-use GPU memory and data across kernels?
- You must use API calls to overlap data movement with kernel invocations

Using 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)
```

Taking CUDA to another level

- The higher-level PGI Accelerator programming model lags behind CUDA C and CUDA Fortran in supporting some features (e.g. full support for device resident data, asynchronous data transfers, etc)
- Writing CUDA C or CUDA Fortran device kernels can be difficult and time-consuming (e.g. cutting the relevant loops or code segments out into a separate kernel routine, porting code involving reductions, etc).
- We address these two issues by borrowing technologies between models

CUF Kernel-based matrix multiply using CUDA Fortran device arrays

Fortran code

```
subroutine mmul(a,b,c,n,m,l)
real, device :: a(n,*),b(m,*),c(n,*)
!$cuf kernel do(2) <<<(*,*),(*,*)>>>
do k = 1, l
   do i = 1, n
     c(i,k) = 0.0
     do j = 1, m
       c(i,k) = c(i,k)+a(i,j)*b(j,k)
     end do
   end do
end do
return
end
```

Generated GPU code description

```
5, CUDA kernel generated

5, !$cuf kernel do <<< (*,*),
       (16,16) >>>
       Using register for 'c'
```

PGI directive-based matrix multiply using CUDA Fortran device arrays

Fortran code

```
subroutine mmul(a,b,c,n,m,l)
real, device :: a(n,*),b(m,*),c(n,*)
!$acc region
   do k = 1, l
      do j = 1, m
         do i = 1, n
            c(i,k) = c(i,k) +
                     a(i,j) * b(j,k)
         end do
      end do
   end do
!$acc end region
return
end
```

Generated GPU code description

```
4, Loop is parallelizable
   Accelerator kernel generated
    4, !$acc do parallel, vector(16)
    5, !$acc do seq
       Cached references to size
       [16x16] block of 'a'
       Cached references to size
       [16x16] block of 'b'
   6, !$acc do parallel, vector(16)
```

GPU programming constants

The Program must:

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

CUDA Fortran Host Code

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

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

db = b
dc = c

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

a = da

deallocate( da, db, dc )
```

Using PGI Accelerator directives

```
!$acc region do
do i = 1, n
   a(i) = b(i) + c(i)
enddo
```

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

End of the world as we know it?

5.4 cublas<t>axpy()

```
cublasStatus_t cublasSaxpy(cublasHandle_t handle, int n,
                           const float           *alpha,
                           const float           *x, int incx,
                           float                 *y, int incy)
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy)
cublasStatus_t cublasCaxpy(cublasHandle_t handle, int n,
                           const cuComplex       *alpha,
                           const cuComplex       *x, int incx,
                           cuComplex             *y, int incy)
cublasStatus_t cublasZaxpy(cublasHandle_t handle, int n,
                           const cuDoubleComplex *alpha,
                           const cuDoubleComplex *x, int incx,
                           cuDoubleComplex       *y, int incy)
```

This function multiplies the vector **x** by the scalar α and adds it to the vector **y** overwriting the latest vector with the result. Hence, the performed operation is $\mathbf{y}[j] = \alpha \times \mathbf{x}[k] + \mathbf{y}[j]$ for $i = 1, \ldots, n$, $k = 1 + (i - 1) * \text{incx}$ and $j = 1 + (i - 1) * \text{incy}$. Notice that the last two equations reflect 1-based indexing used for compatibility with Fortran.

I was looking at a solver and I was missing slaswp. . .

```
#ifdef _CUDA
SUBROUTINE SLASWP_CUF( N, A, LDA, K1, K2, IPIV, INCX )
#else
SUBROUTINE SLASWP( N, A, LDA, K1, K2, IPIV, INCX )
#endif
*
*  -- LAPACK auxiliary routine (version 3.2) --
*     Univ. of Tennessee, Univ. of California Berkeley and NAG Ltd..
*     November 2006
*
*     .. Scalar Arguments ..
      INTEGER :: INCX, K1, K2, LDA, N
*     ..
*     .. Array Arguments ..
#ifdef _CUDA
      INTEGER, DEVICE :: IPIV(N)
      REAL, DEVICE    :: A( LDA, N)
#else
      INTEGER            IPIV( * )
      REAL               A( LDA, * )
#endif
```

```
. . .

IF( N32.NE.0 ) THEN
#ifdef _CUDA
!$cuf kernel do <<< *, 32 >>>
#endif
         DO 30 J = 1, N32, 32
            IX = IX0
            DO 20 I = I1, I2, INC
               IP = IPIV( IX )
               IF( IP.NE.I ) THEN
                  DO 10 K = J, J + 31
                     TEMP = A( I, K )
                     A( I, K ) = A( IP, K )
                     A( IP, K ) = TEMP
   10             CONTINUE
               END IF
               IX = IX + INCX
   20       CONTINUE
   30    CONTINUE
END IF
```

```
      IF( N32.NE.N ) THEN
         N32 = N32 + 1
         IX = IX0
#ifdef _CUDA
!$cuf kernel do <<< *, 1 >>>
#endif
         DO 50 I = I1, I2, INC
            IP = IPIV( IX )
            IF( IP.NE.I ) THEN
               DO 40 K = N32, N
                  TEMP = A( I, K )
                  A( I, K ) = A( IP, K )
                  A( IP, K ) = TEMP
   40          CONTINUE
            END IF
            IX = IX + INCX
   50    CONTINUE
      END IF
```

- Compile it:

```
PGI$ pgfortran -c -Mfixed -Mcuda –Mpreprocess -Minfo slaswp.f
slaswp:
     92, CUDA kernel generated
         92, !$cuf kernel do <<< (*), (32) >>>
    113, CUDA kernel generated
        113, !$cuf kernel do <<< (*), (1) >>>
```

- Write a generic interface:

```
INTERFACE SLASWP
  SUBROUTINE SLASWP( N, A, LDA, K1, K2, IPIV, INCX )
    INTEGER INCX, K1, K2, LDA, N
    INTEGER IPIV( * )
    REAL    A( LDA, * )
  END SUBROUTINE
  SUBROUTINE SLASWP_CUF( N, A, LDA, K1, K2, IPIV, INCX )
    INTEGER INCX, K1, K2, LDA, N
    INTEGER, DEVICE ::  IPIV( * )
    REAL, DEVICE  :: A( LDA, * )
  END SUBROUTINE
END INTERFACE
```

- Use it in CUDA Fortran:

```
USE MY_SLASWP
. . .
CALL SLASWP(N, A_DEV, LDA, K1, K2, IPIV_DEV, INCX)
```

- Use it in the PGI Accelerator Model:

```
use my_slaswp
!$acc data region copy(a, b, x) local(ipiv)
. . .
call slaswp(n, a, lda, k1, k2, ipiv, incx)
```

- Not overly concerned with performance of these small routines at this point; mainly we've avoided data transfers between the device and host
- Go on to the next one. Impress your friends with how fast you've ported your code to GPUs.

Building a CUDA Fortran Program

- **`pgfortran a.cuf`**
  - **`.cuf`** suffix implies CUDA Fortran (free form)
  - **`.CUF`** suffix runs preprocessor
  - Use the **`-Mfixed`** option for F77-style fixed format
- **`pgfortran -Mcuda a.f90`**
  - **`pgfortran -Mcuda[=[emu|cc10|cc11|cc12|cc13|cc20]]`**
- Must use **`-Mcuda`** when linking from object files
  - Compiler driver pulls in correct path and libraries

Latest PGI 11.x Features in CUDA Fortran & PGI Accelerator Compilers

- CUDA 4.0 Support
- Some PRINT * support in CUDA Fortran device routines
- CUBLAS and CUFFT interface modules
- Global subroutine shared memory automatic array support
- Calling generic host/device functions from within a PGI Accelerator data region
- OpenMP parallel regions containing CUDA Fortran calls and PGI Accelerator regions

Programming with Accelerator Directives

Implicit Programming of Accelerators

- The PGI Accelerator directive based approach to programming.
- Maximize the work that the compiler is able to do
- Concentrate programmer efforts on performance of kernels rather then management and placement of data

What parts can the compiler do?

1. **Split code between Host and GPU**
   - CUDA Fortran, CUDA and OpenCL – function level, done manually by programmer
   - Modern Compilers – can do this just as well as you can, and a lot faster, and enable offloading of regions within functions
2. **Manage data allocation/movement between Host and Device**
   - CUDA Fortran does this implicitly through language syntax. Code looks similar to standard Fortran
   - CUDA and OpenCL – do this manually with API calls, one or more per argument to the device kernel, host code nearly unrecognizable compared to original
   - Modern Compilers – can do this almost as well you can, user-driven tuning is required, but can and should be quick and easy
3. **Tune Device Kernels**
   - CUDA Fortran, CUDA and OpenCL – this step is both time-consuming and difficult; must optimize grid/thread geometry, optimize memory placement/accesses, etc
   - Modern Compilers – can help a little here and make the code portable, but this step is probably always going to be ***hard***

Accelerator VADD Device Code
(two dimensional array example)

```
module kmod
 contains
 subroutine vaddkernel(A,B,C)
  real :: A(:,:), B(:,:), C(:,:)
!$acc region
  C(:,:) = A(:,:) + B (:,:)
!$acc end region
 end subroutine
end module
```

!$acc region clauses can surround many individual loops and compute kernels. There is no implicit GPU/CPU data movement within a region

Compiling the subroutine:

```
PGI$ pgfortran -Minfo=accel -ta=nvidia -c vadd.F90

vaddkernel:
     5, Generating copyout(c(1:z_b_14,1:z_b_17))
        Generating copyin(a(1:z_b_14,1:z_b_17))
        Generating copyin(b(1:z_b_14,1:z_b_17))
        Generating compute capability 1.0 binary
        Generating compute capability 1.3 binary
        Generating compute capability 2.0 binary
     6, Loop is parallelizable
        Accelerator kernel generated
         6, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
            !$acc do parallel, vector(16) ! blockidx%y threadidx%y
            CC 1.0 : 7 registers; 64 shared, 8 constant, 0 local memory bytes; 100% occupancy
            CC 1.3 : 8 registers; 64 shared, 8 constant, 0 local memory bytes; 100% occupancy
            CC 2.0 : 15 registers; 8 shared, 72 constant, 0 local memory bytes; 100% occupancy
```

Tuning the compute kernel

Accelerator VADD Device Code

```
module kmod
 contains
 subroutine vaddkernel(A,B,C)    ! We know array size
  real :: A(:,:), B(:,:), C(:,:)! dimension(2560,96)
  integer :: i,j
!$acc region
!$acc do parallel
  do j = 1,size(A,1)
!$acc do vector(96)
    do i = 1,size(A,2)
      C(j,i) = A(j,i) + B (j,i)
    enddo
  enddo
!$acc end region
 end subroutine
end module
```

Keeping the data on the GPU

Accelerator VADD Device Code

```
module kmod
 contains
 subroutine vaddkernel(A,B,C)
  real :: A(:,:), B(:,:), C(:,:)
!$acc reflected (A,B,C)
!$acc region
   C(:,:) = A(:,:) + B (:,:)
!$acc end region
 end subroutine
end module
```

The !$reflected clause must be visible to the caller so it knows to pass pointers to arrays on the GPU rather then copyin actual array data.

Compiling the subroutine:

```
PGI$ pgfortran -Minfo=accel -ta=nvidia -c vadd.F90
vaddkernel:
     5, Generating reflected(c(:,:))
        Generating reflected(b(:,:))
        Generating reflected(a(:,:))
     6, Generating compute capability 1.0 binary
        Generating compute capability 1.3 binary
        Generating compute capability 2.0 binary
     7, Loop is parallelizable
        Accelerator kernel generated
         7, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
            !$acc do parallel, vector(16) ! blockidx%y threadidx%y
            CC 1.0 : 11 registers; 80 shared, 8 constant, 0 local memory bytes; 66% occupancy
            CC 1.3 : 11 registers; 80 shared, 8 constant, 0 local memory bytes; 100% occupancy
            CC 2.0 : 17 registers; 8 shared, 88 constant, 0 local memory bytes; 100% occupancy
```

Allocating/Deallocating GPU Arrays

Accelerator VADD Device Code

```
subroutine vadd(M,N)
  use kmod ! Visibility of !$acc reflected
  real, dimension(:,:) :: A, B, C
  integer :: N
!$acc mirror(A,B,C)
  allocate(A(M,N),B(M,N),C(M,N))
  A = 1.0
  B = 2.0
!$acc update device(A,B)
  call vaddkernel (A,B,C)
  call kernel2 (A,B,C)
  call kernel3 (A,B,C)
  call kernel4 (A,B,C)
!$acc update host(C)
  deallocate( A, B)
 end subroutine
```

```
% pgfortran -help -ta
-ta=nvidia:{analysis|nofma|[no]flushz|keepbin|keepptx|keepgpu|maxregcount:<n>|
           c10|cc11|cc12|cc13|cc20|fastmath|mul24|time|cuda2.3|cuda3.0|
           cuda3.1|cuda3.2|cuda4.0|[no]wait}|host
                  Choose target accelerator
   nvidia         Select NVIDIA accelerator target
   analysis       Analysis only, no code generation
   nofma          Don't generate fused mul-add instructions
   [no]flushz     Enable flush-to-zero mode on the GPU
   keepbin        Keep kernel .bin files
   keepptx        Keep kernel .ptx files
   keepgpu        Keep kernel source files
   maxregcount:<n> Set maximum number of registers to use on the GPU
   cc10           Compile for compute capability 1.0
   ...
   cc20           Compile for compute capability 2.0
   fastmath       Use fast math library
   mul24          Use 24-bit multiplication for subscripting
   time           Collect simple timing information
   cuda2.3        Use CUDA 2.3 Toolkit compatibility
   ...
   cuda4.0        Use CUDA 4.0 Toolkit compatibility
   [no]wait       Wait for each kernel to finish; overrides nowait clause
   host           Compile for the host, i.e. no accelerator target
```

Obstacles to GPU code generation

- Loop nests to be offloaded to the GPU must be rectangular
- At least some of the loops to be offloaded must be fully data parallel with no synchronization or dependences across iterations
- Computed array indices should be avoided
- All function calls must be inlined within loops to be offloaded
- In Fortran, the pointer attribute is not supported; pointer arrays may be specified, but pointer association is not preserved in GPU device memory
- In C
  - Loops that operate on structs can be offloaded, but those that operate on nested structs cannot
  - Pointers used to access arrays in loops to be offloaded must be declared with C99 restrict (or compiled w/-Msafeptr, but it is file scope)
  - Pointer arithmetic is not allowed within loops to be offloaded

Obstacles to loop parallelization or vectorization in a compute region

- Computed Index (linearization, look-up) ➡
- While loops ➡
- Triangular loops ➡
- "live-out" variables ➡
- Local arrays that must be privatized ➡
- Function calls that cannot be inlined ➡
- Device runtime errors – failure to launch ➡
- Compiler errors ➡

Computed Index – Linearization

```
!$acc region
   do i = 1, M
      do j = 1, N
         idx = ((i-1)*M)+j
         A(idx) = B(i,j)
      enddo
   enddo
!$acc end region
```

```
% pgfortran linearization.f90 -ta=nvidia -Minfo=accel
linear:
   16, No parallel kernels found, accelerator
       region ignored
   17, Complex loop carried dependence of 'a'
       prevents parallelization
   18, Complex loop carried dependence of 'a'
       prevents parallelization
       Parallelization would require  privatization of
       array 'a(:)
```

To fix, remove the linearization or use the "independent" clause:

```
!$acc region
   do i = 1, M
      do j = 1, N
         A(i,j) = B(i,j)
      enddo
   enddo
!$acc end region
```

```
!$acc region
!$acc do independent
   do i = 1, M
      do j = 1, N
         idx = ((i-1)*M)+j
         A(idx) = B(i,j)
      enddo
   enddo
!$acc end region
```

Computed Index - Look-up

```
!$acc region
do i = 1, M
   idx = lookup(i)
   do j = 1, N
      A(idx,j) = ((i-1)*M)+j
   enddo
enddo
!$acc end region
```

```
% pgfortran lookup.f90 -ta=nvidia -Minfo=accel
lookup_test:
   16, Generating copyout(a(:,1:1024))
   17, Parallelization would require privatization of array
       'a(:,1:1024)'
       Sequential loop scheduled on host
   19, Loop is parallelizable
       Accelerator kernel generated
      19, !$acc do parallel, vector(256)
```

```
!$acc region
do i = 1, M
   do j = 1, N
      idx = lookup(j)
      A(i,idx) = ((i-1)*M)+j
   enddo
enddo
!$acc end region
```

```
% pgfortran lookup1.f90 -ta=nvidia -Minfo=accel
lookup_test:
   16, Generating copyout(a(1:1024,:))
       Generating copyin(cell(1:1024))
   17, Loop is parallelizable
       Accelerator kernel generated
       17, !$acc do parallel, vector(256)
   18, Loop carried reuse of 'a' prevents parallelization
       Inner sequential loop scheduled on accelerator
```

The Independent or parallel clauses could be used to force parallelization but is not recommended

While Loops

```
!$acc region
   i = 0
   do, while (.not.found)
      i = i + 1
      if (A(i) .eq. 102) then
         found = i
      endif
   enddo
!$acc end region
```

```
% pgf90 -ta=nvidia -Minfo=accel while.f90
while1:
    17, Accelerator region ignored
    19, Accelerator restriction: invalid loop
```

Convert to a rectangular loop:

```
!$acc region
   do i = 1, N
      if (A(i) .eq. 102) then
         found(i) = i
      else
         found(i) = 0
      endif
   enddo
!$acc end region
print *, 'Found at ', minval(found)
```

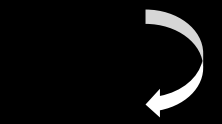
```
% pgf90 -ta=nvidia -Minfo=accel while2.f90
while2:
    18, Generating copyin(a(1:1024))
        Generating copyout(found(1:1024))
        Generating compute capability 1.0 binary
        Generating compute capability 1.3 binary
    19, Loop is parallelizable
        Accelerator kernel generated
        19, !$acc do parallel, vector(256)
            Using register for 'found'
```

Triangular Loops

```
!$acc region copyout(A)
do i = 1, M
   do j = i, N
      A(i,j) = i+j
   enddo
enddo
!$acc end region
```

All loop schedules must be rectangular. For triangular loops, the compiler will either serialize the inner loop or make the inner loop rectangular and add an implicit if statement to skip the lower part of the triangle.

Problem: The compiler will copy out the entire array A. The lower triangle contains garbage since it was not initialized. Use "copy(A)" to initialize the values.



"live-out" Variables

```
!$acc region
do i = 1, M
   do j = 1, N
      idx = i+j
      A(i,j) = idx
   enddo
enddo
!$acc end region
print *, idx, A(1,1), A(M,N)
```

```
% pgf90 -ta=nvidia,time -Minfo=accel  liveout.f90
liveout:
   11, Generating copyout(a(1:1024,1:1024))
   12, Loop is parallelizable
       Accelerator kernel generated
       12, !$acc do parallel, vector(256)
   13, Inner sequential loop scheduled on accelerator
   14, Accelerator restriction: induction variable
       live-out from loop: idx
   15, Accelerator restriction: induction variable
       live-out from loop: idx
```

Privatize the scalar:

```
!$acc region
do i = 1, M
   !$acc do private(idx)
   do j = 1, N
      idx = i+j
      A(i,j) = idx
   enddo
enddo
!$acc end region
print *, idx, A(1,1), A(M,N)
```

```
% pgf90 -ta=nvidia,time -Minfo=accel liveout2.f90
liveout2:
   10, Generating copyout(a(1:1024,1:1024))
   11, Loop is parallelizable
   13, Loop is parallelizable
       Accelerator kernel generated
       11, !$acc do parallel, vector(16)
       13, !$acc do parallel, vector(16)
```

Privatization of Local Arrays

```
!$acc region
  do i = 1, M
    do j = 1, N
      do jj = 1, 10
        tmp(jj) = jj
      end do
      A(i,j) = sum(tmp)
    enddo
  enddo
!$acc end region
```

```
% pgf90 -ta=nvidia -Minfo=accel  private.f90
privatearr:
     10, Generating copyout(tmp(1:10))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
     11, Parallelization would require privatization of array 'tmp(1:10)'
     13, Parallelization would require privatization of array 'tmp(1:10)'
         Sequential loop scheduled on host
     14, Loop is parallelizable
         Accelerator kernel generated
         14, !$acc do parallel, vector(10)
     17, Loop is parallelizable
         Accelerator kernel generated
         17, !$acc do parallel, vector(10)
             Sum reduction generated for tmp$r
```

Privatization of Local Arrays - cont.

```
!$acc region
  do i = 1, M
!$acc do private(tmp)
    do j = 1, N
      do jj = 1, 10
        tmp(jj) = jj
      end do
      A(i,j) = sum(tmp)
    enddo
  enddo
!$acc end region
```

```
% pgf90 -ta=nvidia,time -Minfo=accel private2.f90
privatearr2:
     10, Generating copyout(a(1:1024,1:1024))
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
     11, Loop is parallelizable
     13, Loop is vectorizable
         Accelerator kernel generated
         11, !$acc do parallel, vector(16)
         13, !$acc do vector(16)
     14, Loop is parallelizable
     17, Loop is parallelizable
```

Need to privatize local temporary arrays.
Default is to assume that they are shared.

Function Calls

- Function calls are not allowed within a compute region
- Restriction is due to lack of a device linker and hardware support
- Functions must be inlined, either manually or by the compiler using –Minline or –Mipa=inline

Managing data allocation and movement between Host and GPU memories

You optimize data movement in CUDA C by manipulating API calls

- Can you eliminate or tune API calls?
- Can you place host data in Pinned memory?
- Can you re-use GPU memory and data across kernels?
- Can you overlap data movement with kernel invocations?

You optimize data movement with PGI Accelerator compilers using directives

- Add clauses to compute REGION directives to minimize data movement ➡
- Use DATA REGIONs to re-use data across kernels ➡
- Use MIRROR, REFLECTED and UPDATE to re-use data across subroutine boundaries in Fortran ➡
- NOTE: there is no support for asynchronous data movement in the PGI Accelerator model (yet) ➡

Compute region directive clauses for tuning data allocation and movement

Clause	Meaning
`if (`*`condition`*`)`	Execute on GPU conditionally
`copy (`*`list`*`)`	Copy in and out of GPU memory
`copyin (`*`list`*`)`	Only copy in to GPU memory
`copyout (`*`list`*`)`	Only copy out of GPU memory
`local (`*`list`*`)`	Allocate locally on GPU
`deviceptr (`*`list`*`)`	C pointers in *list* are device pointers
`update device (`*`list`*`)`	Update device copies of the arrays
`update host (`*`list`*`)`	Update host copies of the arrays

Use compute region clauses to override compiler decisions on data movement

For example, by default the compiler will move the minimum amount of data required for correct execution:

```
% pgfortran -ta=nvidia -fast -c -o J1.o J1.f90 -Minfo=accel
jacobi:
     18, Generating copyin(a(1:m,1:n))
         Generating copyout(a(2:m-1,2:n-1))
         Generating copyout(newa(2:m-1,2:n-1))
...
```

Use *copy* clause to force one contiguos copyin/copyout of entire *a* array, *local* clause to eliminate copyout of *newa*:

```
% pgfortran -ta=nvidia -fast -c -o J1.o J1.f90 -Minfo=accel
jacobi:
     18, Generating copy(a(:,:))
         Generating local(newa(:,:))
...
```

Data regions enable re-use of GPU data across compute regions

- Data region in C

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

- Data region in Fortran

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

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

Using GPU device-resident data across subroutines

```
subroutine timestep(Input,Result,M,N)
   use kmod ! Make reflected var's visible
   real, dimension(M,N) :: Input,Result

   integer :: M,N
   real, allocatable :: B,C,D
   dimension(:,:) :: B,C,D
!$acc mirror(B,C,D)
   allocate(B(M,N),C(M,N),D(M,N))
   B = 2.0
!$acc update device(Input,B)
   call vaddkernel (Input,B,C)
   ...
   call kernel2 (C,D)
   ...
   call kernel3 (D,Result)
!$acc update host(Result)
   deallocate(B,C,D)
end subroutine
```

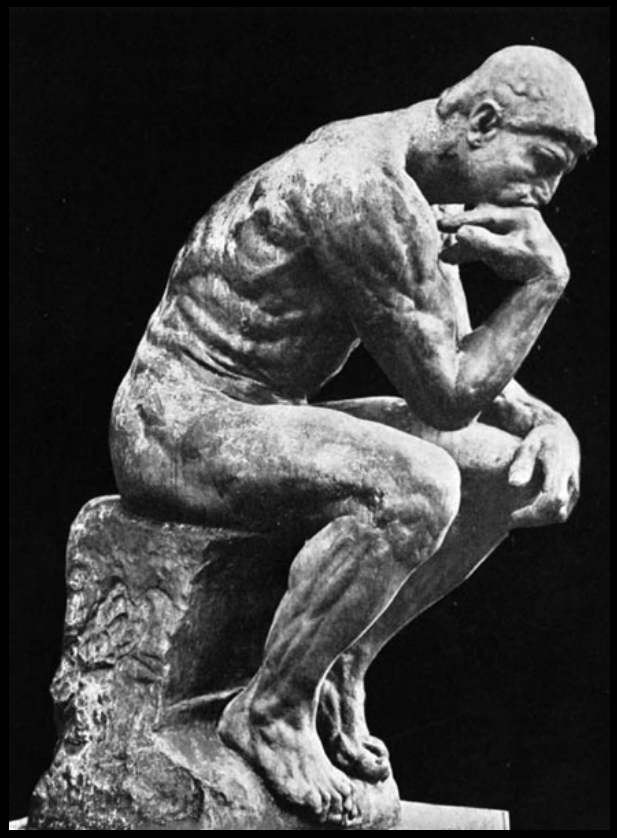
CPU Code

```
module kmod
Contains
!
   subroutine vaddkernel(A,B,C)
   real :: A(:,:),B(:,:),C(:,:)
!$acc reflected (A,B,C)
!$acc region
      C(:,:) = A(:,:) + B (:,:)
!$acc end region
   end subroutine
!
   subroutine kernel2(C,D)
   real :: C(:,:),D(:,:)
!$acc reflected (C,D)
!$acc region
      < compute-intensive loops >
!$acc end region
   end subroutine
   ...
end module
```

GPU Code

Tuning GPU kernel schedules and memory usage

You optimize kernels in CUDA C/Fortran using heuristics and experimentation



You optimize kernels with PGI Accelerator compilers by adding directives ...

```
void
computeMM(float C[][WB], float A[][WA], float B[][WB],
          int hA, int wA, int wB)
{
#pragma acc region
{
    int i, j, k;
#pragma acc for parallel vector(16)
    for (int i = 0; i < hA; ++i) {
        C[i][j] = 0.0;
        for (int k = 0; k < wA; ++k) {
#pragma acc for parallel vector(16)
            for (int j = 0; j < wB; ++j) {
               C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
}
}
```

... interpreting compiler feedback, and restructuring loops

```
% pgcc -fast -ta=nvidia -Minfo mm.c
...
  62, Loop is parallelizable
  64, Loop carried dependence of 'C' prevents parallelization
      Loop carried backward dependence of 'C' prevents vectorization
  66, Loop is parallelizable
      Accelerator kernel generated
      62, #pragma acc for parallel, vector(16) /* blockIdx.y threadIdx.y */
      64, #pragma acc for seq(16)
          Cached references to size [16x16] block of 'A'
          Cached references to size [16x16] block of 'B'
      66, #pragma acc for parallel, vector(16) /* blockIdx.x threadIdx.x */
          Using register for 'C'
          CC 1.3 : 27 registers; 2264 shared, 24 constant,
                   0 local memory bytes; 50% occupancy
...
```

Loop directive clauses for tuning GPU kernel schedules

Clause	Meaning
`parallel [(`*`width`*`)]`	Parallelize the loop across the multi-processors
`vector [(`*`width`*`)]`	SIMD vectorize the loop within a multi-processor
`seq [(`*`width`*`)]`	Execute the loop sequentially on each thread processor
`independent`	Iterations of this loop are data independent of each other
`unroll (`*`factor`*`)`	Unroll the loop *factor* times
`cache (`*`list`*`)`	Try to place these variables in shared memory
`private (`*`list`*`)`	Allocate a copy of each variable in *list* for each loop iteration

Loop Schedules

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

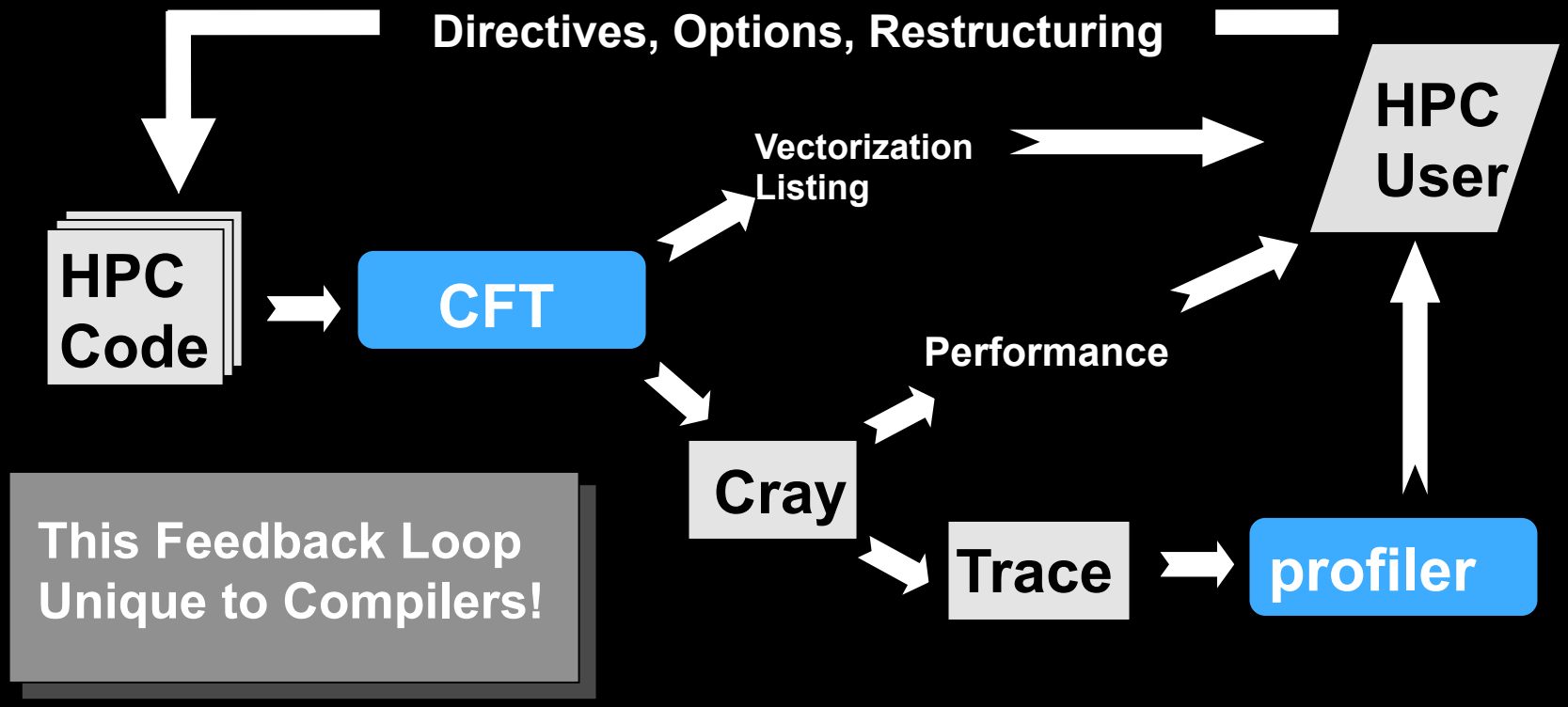
- Vector loops correspond to threadidx indices (SIMD)
- Parallel loops correspond to blockidx indices (MIMD)
- The loop nest above has a CUDA schedule:

```
<<< dim3(ceil(N/16),ceil(M/16)), dim3(16,16) >>>
```

- The compiler strip-mines to protect against very long loop limits
- It is possible to create any legal CUDA schedule using the PGI Accelerator loop scheduling clauses

How did we make Vectors Work?

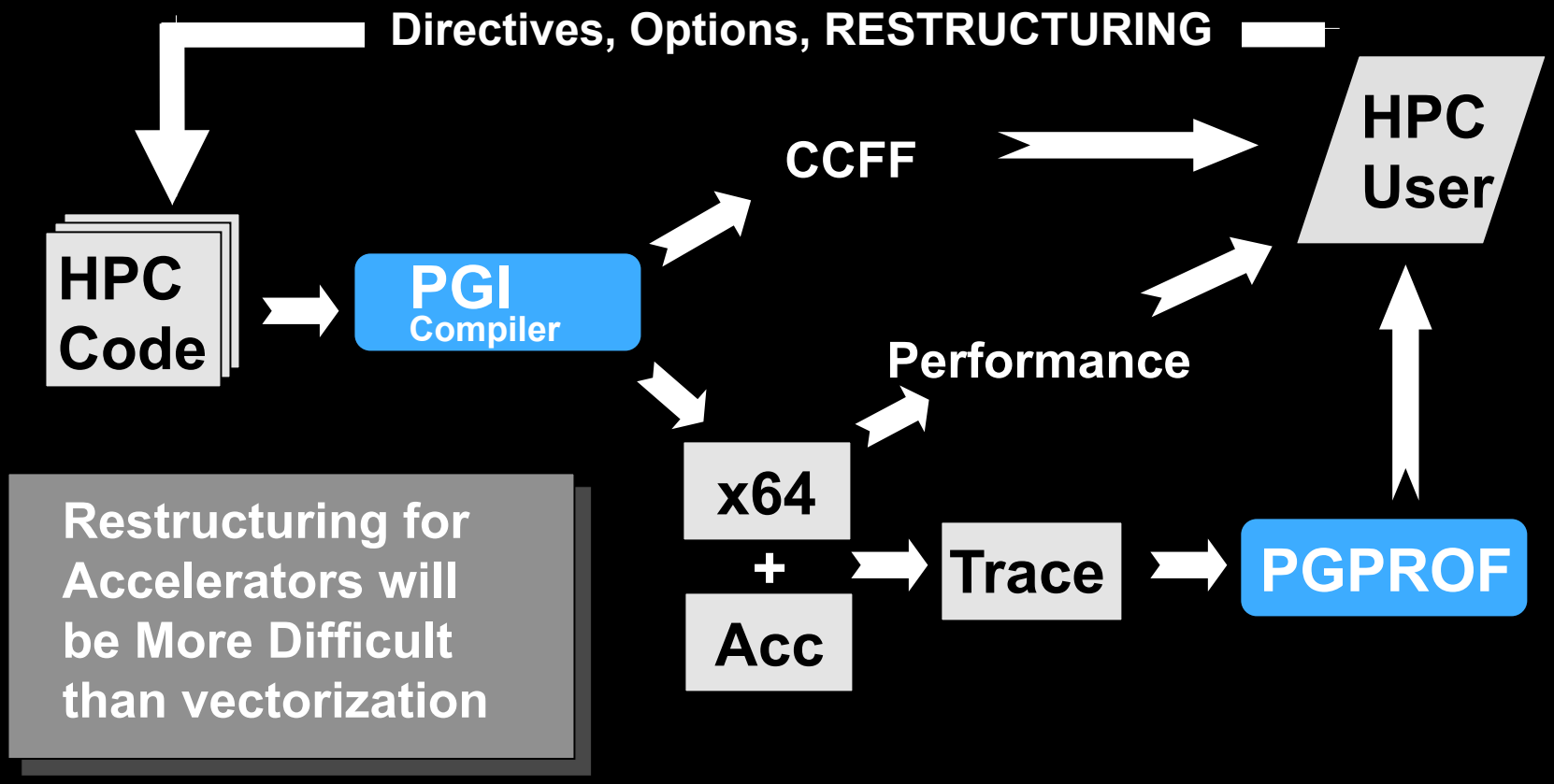
Compiler-to-Programmer Feedback – a classic "Virtuous Cycle"



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

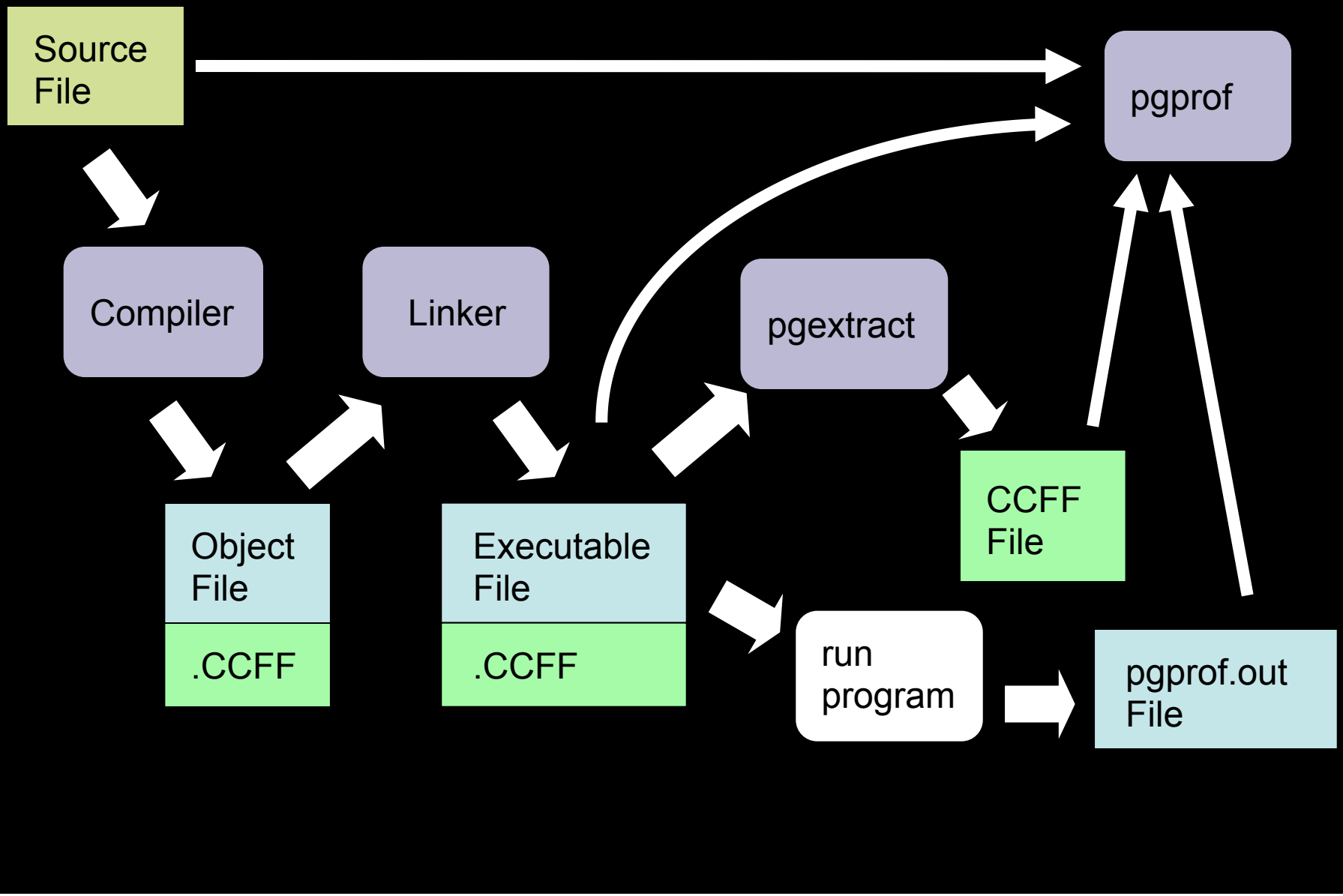
Compiler-to-Programmer Feedback

Incremental porting/tuning for GPUs

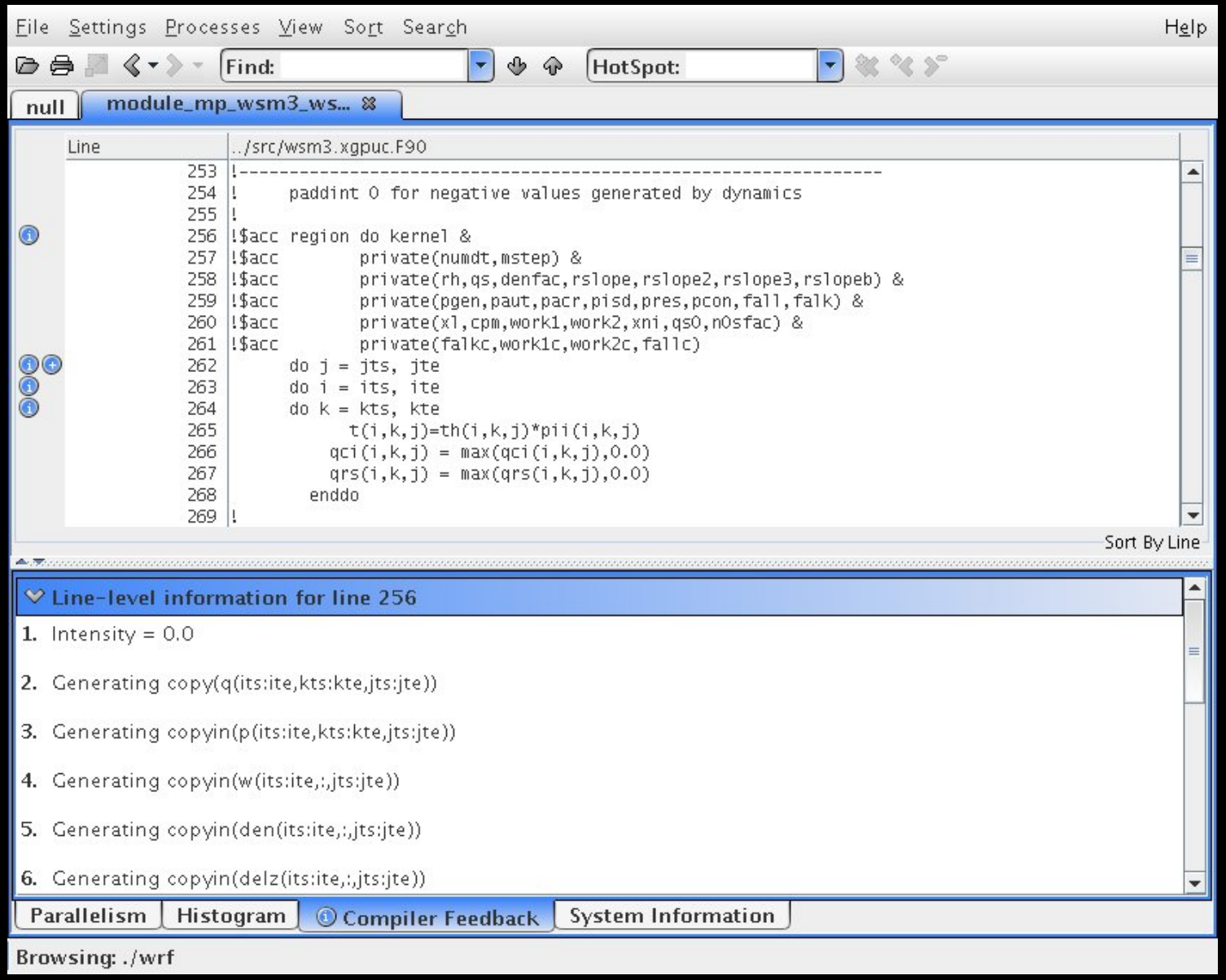


Common Compiler Feedback Format

http://www.pgroup.com/ccff



PGPROF with CCFF Messages



General Compiler Feedback

- How the function was compiled
- Interprocedural optimizations
- Profile-feedback runtime data
  - Block execution counts
  - Loop counts, range of counts
- Compiler optimizations, missed opportunities
  - Vectorization, parallelization
  - Altcode, re-ordering of loops, inlining
  - X64+GPU code generation, GPU kernel mapping, data movement
- Compute intensity – important for GPUs & Multi-core

Example PGI Accelerator compiler feedback messages

- `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))`
- `Generating local(c(1:n,1:n))`
- `Generating reflected(d(1:n,1:n))`
- `Loop is parallelizable`
- `Accelerator kernel generated`
- `No parallel kernels found, accelerator region ignored`
- `Loop carried dependence due to exposed use of ... prevents parallelization`
- `Parallelization would require privatization of array ...`
- `Accelerator restriction: invalid loop`
  `Loop not vectorized/parallelized: not countable`

Categories of PGI Accelerator compiler feedback

- Hindrances
  - live out variables, non-private arrays
  - loop-carried dependences
  - unsupported data type, unsupported operation
  - unknown array bounds
- Data movement
  - copyin, copyout, local
- Optimizations performed, potential performance problems
  - cached data, register usage, local /constant memory usage
  - non-stride-1 accesses (non-coalesced)
- Loop schedules
  - mapping of loop iterations to thread/block indices
  - occupancy calculation

Device Errors

Call to cuMemcpyDtoH returned error 700: Launch failed

```
!$acc region
do i = 1, M
   do j = 1, N
      A(i,j) = B(i,j+1) << out-of-bounds
   enddo
enddo
!$acc end region
```

Typically occurs when the device kernel gets an execution error, such as a out-of-bounds or other memory access violation.

Call to cuMemcpy2D returned error 1: Invalid value

```
parameter(N=1024,M=512)
real :: A(M,N), B(M,N)
...
!$acc region copyout(A), copyin(B(0:N,1:M+1))  <<< Bad bounds
   do i = 1, M                                      for copyin
      do j = 1, N
         A(i,j) = B(i,j+1)
      enddo
   enddo
!$acc end region
```

Typically occurs if there is an error copying data to/from the device

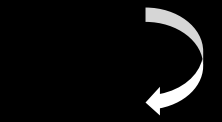
Compiler Errors

- No software is without bugs, including the compiler
- If you encounter a problem that seems to be the fault of the compiler, please send a report to PGI Customer Service (trs@pgroup.com)
- Oak Ridge has special reporting website

Example of an Internal Compiler Error (ICE):

```
% pgf90 -ta=nvidia -c bug.f90
/tmp/pgaccrRVcZyn0dlyP.gpu(20): error: identifier "z0" is undefined

1 error detected in the compilation of "/tmp/pgnvdASVco8Pmxo9p.nv0".
PGF90-F-0000-Internal compiler error. pgnvd job exited with nonzero
status code       0 (bug.f90: 22)
```



Timing / Profiling

- How long does my program take to run?
  - time ./myprogram
- How long do my kernels take to run?
  - pgfortran –ta=nvidia,time
- Environment variables:

- export ACC_NOTIFY=0
- export NVDEBUG=0
- # cuda profiler settings
- #export CUDA_PROFILE=1
- #export CUDA_PROFILE_CONFIG=cudaprof.cfg
- #export CUDA_PROFILE_CSV=1
- #export CUDA_PROFILE_LOG=cudaprof.log

OpenMP and Accelerator Directives

```
program Main
use accel_lib
 …
!$omp parallel private(ilo, iho, k, flux) num_threads(2)
  do iter = 1, 100
     call acc_set_device_num(omp_get_thread_num(), ACC_DEVICE_NVIDIA)
     if ( first ) then
       dkm = km/omp_get_thread_num()
       ilo = dkm *omp_get_thread_num() + 1
       if (omp_get_thread_num() + 1 == omp_get_num_threads()) then
         iho = km
       else
         iho = dkm*(omp_get_thread_num() + 1)
       endif
     endif
!$acc region
     do k=ilo,iho
          …
     end do
!$acc end region
!$omp end parallel
   end do
end program Main
```

Automate the mechanical ... focus on the creative

1. Split code between Host and GPU
2. Manage data allocation & movement between Host and GPU memories
3. Tune GPU Kernel Schedules and Memory Usage

Mechanical

Creative

The PGI Accelerator compilers provide some advantages over CUDA

- They automate the mechanical aspects of GPU programming
- Your programs remain standard-compliant and portable
- The programming model and porting process is more incremental than CUDA

Matrix Multiply Source Code Size Comparison:

```
1 void
2 computeMM0_saxpy(float C[][WB], float A[][WA], float B[][WB],
3                  int hA, int wA, int wB)
4 {
5 #pragma acc region
6 {
7 #pragma acc for parallel vector(16) unroll(4)
8     for (int i = 0; i < hA; ++i) {
9         for (int j = 0; j < wB; ++j) {
10            C[i][j] = 0.0 ;
11        }
12        for (int k = 0; k < wA; ++k) {
13            for (int j = 0; j < wB; ++j) {
14                C[i][j] = C[i][j]+A[i][k]*B[k][j];
15            }
16        }
17    }
18 }
19 }
}
```

Directives

```
1 void
2 __global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
3 {
4     int bx = blockIdx.x;
5     int by = blockIdx.y;
6     int tx = threadIdx.x;
7     int ty = threadIdx.y;
8     int aBegin = wA * BLOCK_SIZE * by;
9     int aEnd   = aBegin + wA - 1;
10    int aStep  = BLOCK_SIZE;
11    int bBegin = BLOCK_SIZE * bx;
12    int bStep  = BLOCK_SIZE * wB;
13    float Csub = 0;
14    for (int a = aBegin, b = bBegin;
15             a <= aEnd;
16             a += aStep, b += bStep) {
17        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
18        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
19        AS(ty, tx) = A[a + wA * ty + tx];
20        BS(ty, tx) = B[b + wB * ty + tx];
21        __syncthreads();
22        for (int k = 0; k < BLOCK_SIZE; ++k)
23            Csub += AS(ty, k) * BS(k, tx);
24        __syncthreads();
25    }
26    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
27    C[c + wB * ty + tx] = Csub;
28 }
29
30 void
31 domatmul( float* C, float* A, float* B, unsigned int hA, unsigned int wA , unsigned wB )
32 {
33     unsigned int size_A = WA * HA;
34     unsigned int mem_size_A = sizeof(float) * size_A;
35     unsigned int size_B = WB * HB;
36     unsigned int mem_size_B = sizeof(float) * size_B;
37     unsigned int size_C = WC * HC;
38     unsigned int mem_size_C = sizeof(float) * size_C;
39     float *d_A, *d_B, *d_C;
40
41     cudaMalloc((void**) &d_A, mem_size_A);
42     cudaMalloc((void**) &d_B, mem_size_B);
43     cudaMalloc((void**) &d_C, mem_size_C);
44     cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
45     cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
46
47     dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
48     dim3 grid(WC / threads.x, HC / threads.y);
49
50     matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
51
52     cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
53
54     cudaFree(d_A);
55     cudaFree(d_B);
56     cudaFree(d_C);
57 }
```

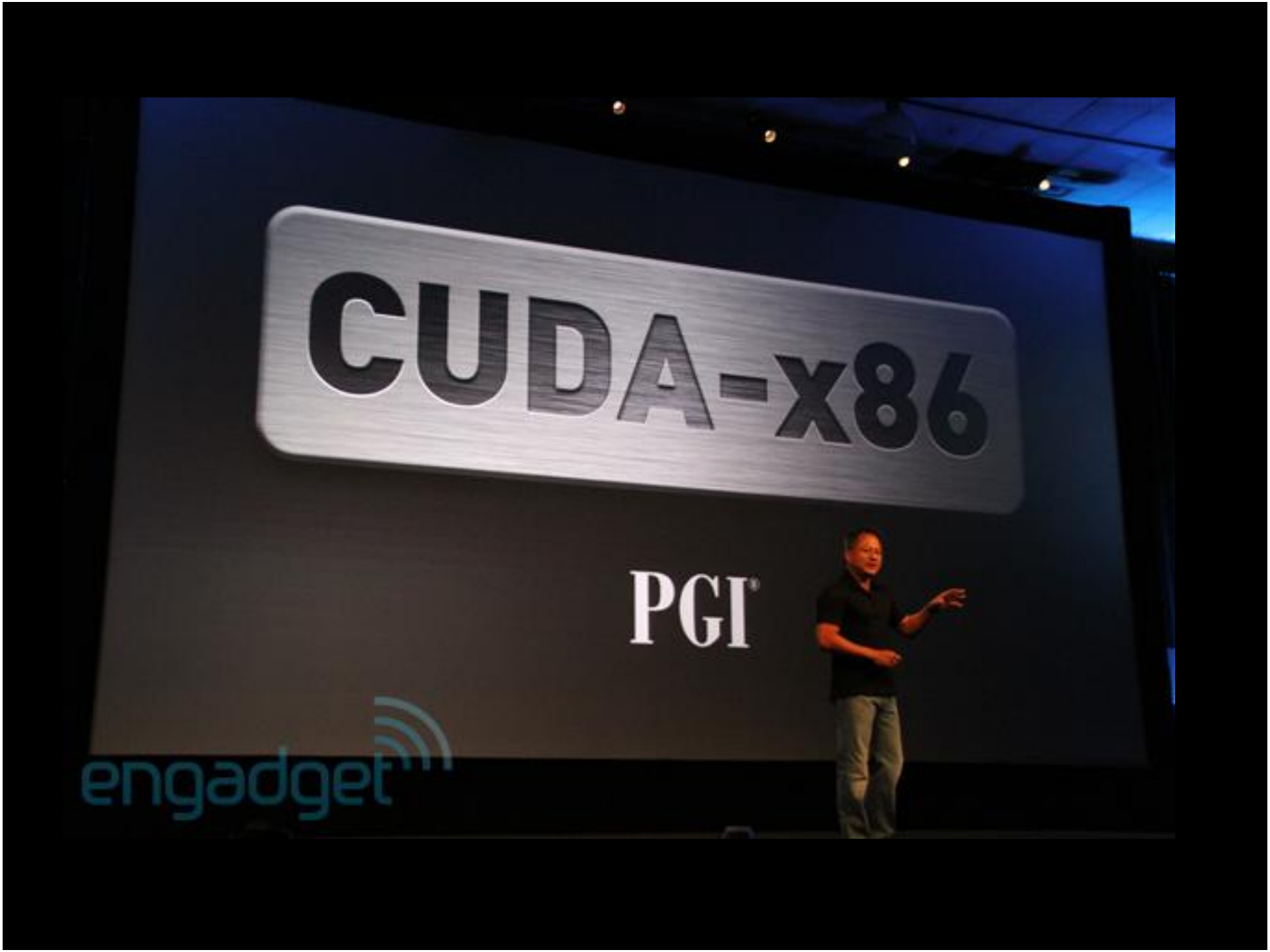
CUDA C

```
1 void matrixMulGPU(cl_uint ciDeviceCount, cl_mem h_A, float* h_B_data,
2                   unsigned int mem_size_B, float* h_C )
2 {
3     cl_mem d_A[MAX_GPU_COUNT];
4     cl_mem d_C[MAX_GPU_COUNT];
5     cl_mem d_B[MAX_GPU_COUNT];
6
7     cl_event GPUDone[MAX_GPU_COUNT];
8     cl_event GPUExecution[MAX_GPU_COUNT];
9
12    // Create buffers for each GPU
13    // Each GPU will compute sizePerGPU rows of the result
14    int sizePerGPU = HA / ciDeviceCount;
15
16    int workOffset[MAX_GPU_COUNT];
17    int workSize[MAX_GPU_COUNT];
18
19    workOffset[0] = 0;
20    for(unsigned int i=0; i < ciDeviceCount; ++i)
21    {
22        // Input buffer
23        workSize[i] = (i != (ciDeviceCount - 1)) ? sizePerGPU : (HA - workOffset[i]);
24
25        d_A[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, workSize[i] * sizeof(float) * WA, NULL,NULL);
26
27        // Copy only assigned rows from host to device
28        clEnqueueCopyBuffer(commandQueue[i], h_A, d_A[i], workOffset[i] * sizeof(float) * WA,
29                            0, workSize[i] * sizeof(float) * WA, 0, NULL, NULL);
30
31        // create OpenCL buffer on device that will be initiatlize from the host memory on first use
32        // on device
33        d_B[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
34                                mem_size_B, h_B_data, NULL);
35
36        // Output buffer
37        d_C[i] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,  workSize[i] * WC * sizeof(float), NULL,NULL);
38
39        // set the args values
40        clSetKernelArg(multiplicationKernel[i], 0, sizeof(cl_mem), (void *) &d_C[i]);
41        clSetKernelArg(multiplicationKernel[i], 1, sizeof(cl_mem), (void *) &d_A[i]);
42        clSetKernelArg(multiplicationKernel[i], 2, sizeof(cl_mem), (void *) &d_B[i]);
43        clSetKernelArg(multiplicationKernel[i], 3, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
44        clSetKernelArg(multiplicationKernel[i], 4, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
45
46        if(i+1 < ciDeviceCount)
47            workOffset[i + 1] = workOffset[i] + workSize[i];
48    }
49    // Execute Multiplication on all GPUs in parallel
50    size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
51    size_t globalWorkSize[] = {shrRoundUp(BLOCK_SIZE, WC), shrRoundUp(BLOCK_SIZE, workSize[0])};
52    // Launch kernels on devices
53        for(unsigned int i = 0; i < ciDeviceCount; i++)
54        {
55                        // Multiplication - non-blocking execution
56                        globalWorkSize[1] = shrRoundUp(BLOCK_SIZE, workSize[i]);
57                        clEnqueueNDRangeKernel(commandQueue[i], multiplicationKernel[i], 2, 0, globalWorkSize, localWorkSize,
58                                               0, NULL, &GPUExecution[i]);
59                }
60        for(unsigned int i = 0; i < ciDeviceCount; i++)
61    {
62                clFinish(commandQueue[i]);
63        }
64    for(unsigned int i = 0; i < ciDeviceCount; i++)
65    {
66 // Non-blocking copy of result from device to host
67        clEnqueueReadBuffer(commandQueue[i], d_C[i], CL_FALSE, 0, WC * sizeof(float) * workSize[i],
68                            h_C + workOffset[i] * WC, 0, NULL, &GPUDone[i]);
69    }
70 // CPU sync with GPU
71    clWaitForEvents(ciDeviceCount, GPUDone);
72
73 // Release mem and event objects
74    for(unsigned int i = 0; i < ciDeviceCount; i++)
75    {
76        clReleaseMemObject(d_A[i]);
77        clReleaseMemObject(d_C[i]);
78        clReleaseMemObject(d_B[i]);
79        clReleaseEvent(GPUExecution[i]);
80        clReleaseEvent(GPUDone[i]);
81    }
82 }
83 __kernel void
84 matrixMul( __global float* C, __global float* A, __global float* B,
85            __local float* As, __local float* Bs)
86 {
87     int bx = get_group_id(0), tx = get_local_id(0);
88     int by = get_group_id(1), ty = get_local_id(1);
89     int aEnd   = wA * BLOCK_SIZE * by + wA - 1;
90
91     float Csub = 0.0f;
92
93     for (int a = wA*BLOCK_SIZE*by , b = BLOCK_SIZE * bx;
94          a <= aEnd; a += BLOCK_SIZE, b += BLOCK_SIZE*wB) {
95         As[tx + ty * BLOCK_SIZE] = A[a + wA * ty + tx];
96         Bs[tx + ty * BLOCK_SIZE] = B[b + wB * ty + tx];
97         barrier(CLK_LOCAL_MEM_FENCE);
98         for (int k = 0; k < BLOCK_SIZE; ++k)
99             Csub += As[k + ty * BLOCK_SIZE]*Bs[tx + k * BLOCK_SIZE] ;
101        barrier(CLK_LOCAL_MEM_FENCE);
102    }
103    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = Csub;
104
105 }
```

OpenCL

The PGI Accelerator compilers have limitations relative to CUDA

- No support for PINNED memory
- No support for C++

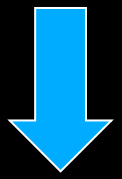

CUDA-x86
PGI
engadget

PGI CUDA C/C++ for x86 - Motivation

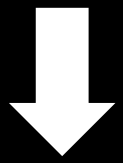
- Provide CUDA developers with a common code path for both NVIDIA GPU and multi-core x86 platform support
- Run CUDA C/C++ & Fortran applications on x86 clusters

PGI CUDA compilers for multi-core x86 and NVIDIA GPUs

PGI CUDA C/C++/Fortran



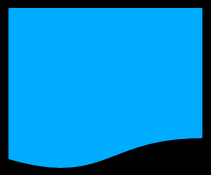
Optimization / Parallelization

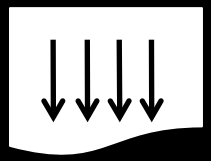


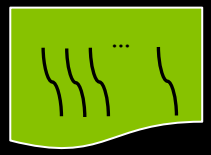
Optimized Host Code

Parallel Multi-core Kernels

Massively Parallel GPU Kernels



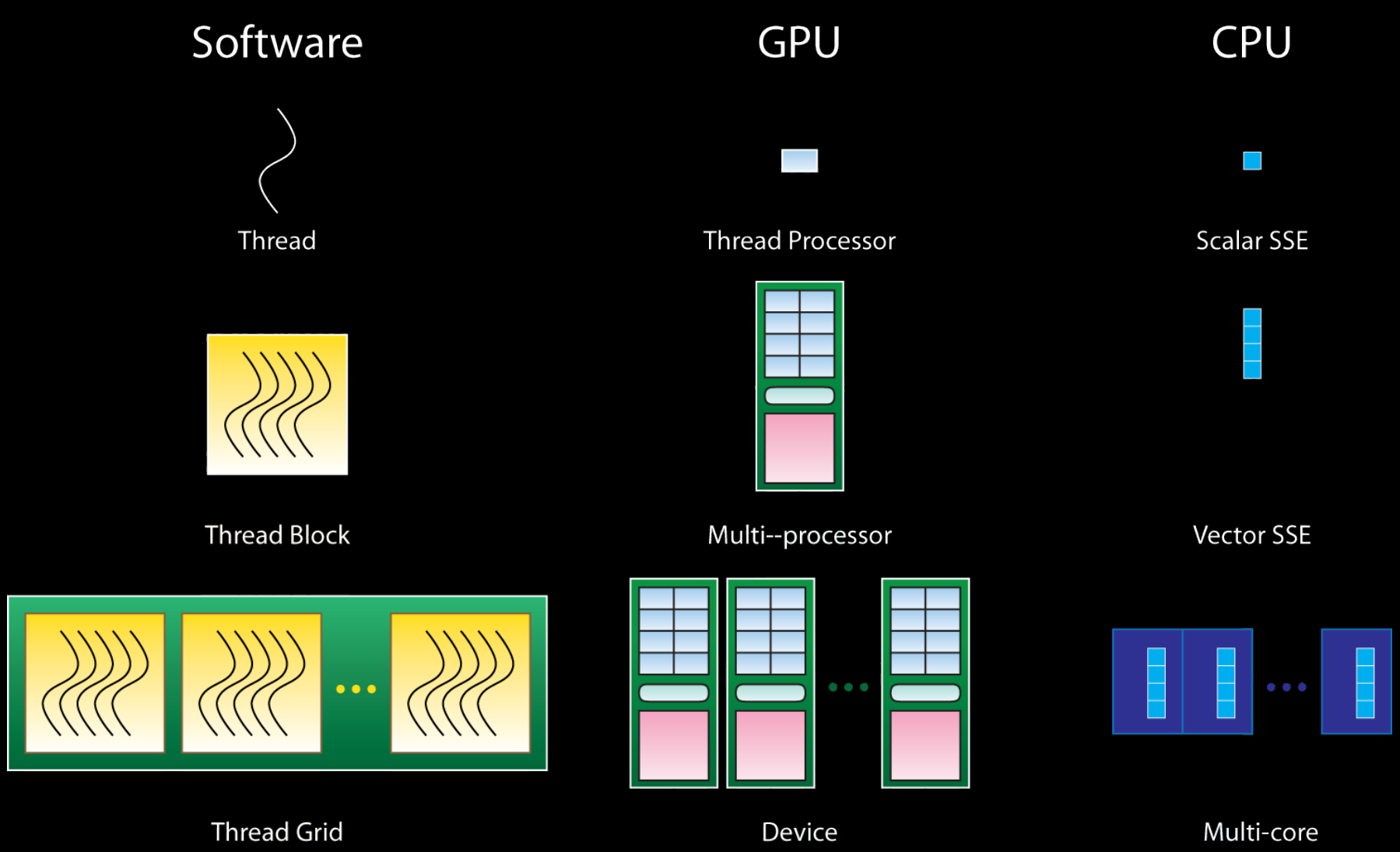




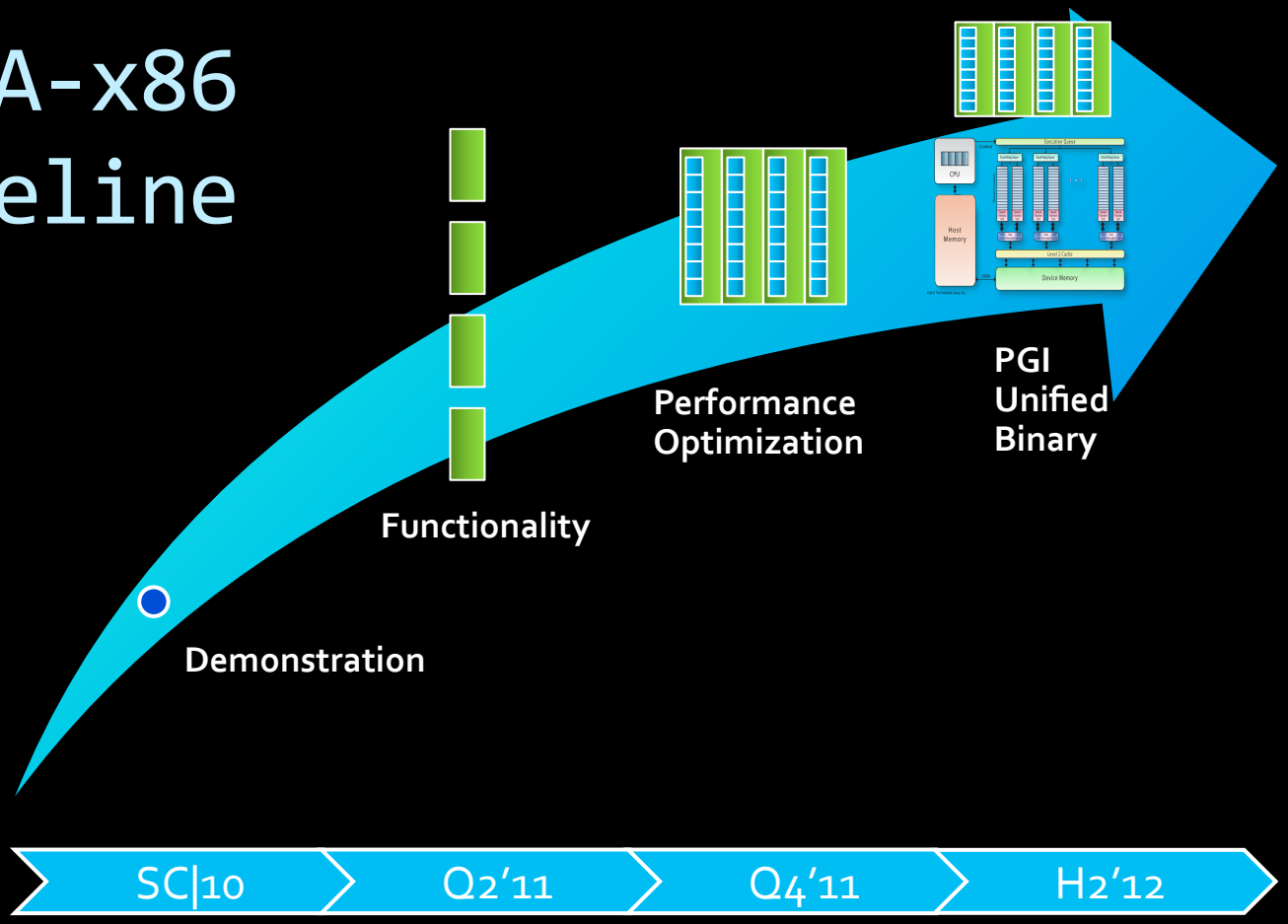
Optimized CUDA C/C++ for x86

- Process CUDA C/C++ as a native parallel programming language
  - 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: full PGI Intel/AMD optimizations support
  - Common compiler back-end provides code generation for new platforms (ie SSE/AVX extensions) and optimization for all languages
  - Tuned compilation delivers maximum performance

CUDA for GPUs vs Multi-core x86



CUDA-x86 Timeline



The Functionality release is available as of PGI 11.5. More info at pgroup.com/cuda_x86.htm

Some Reference Materials

- **PGI Accelerator programming model**
  - http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf
- **CUDA Fortran**
  - http://www.pgroup.com/lit/whitepapers/pgicudaforug.pdf
- **CUDA-x86**
  - http://www.pgroup.com/resources/cuda-x86.htm
- **Understanding the CUDA Threading Model**
  - http://www.pgroup.com/lit/articles/insider/v2n1a5.htm

Copyright Notice