PGI® 2010 Compilers & Tools for x64+GPU Systems

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

January 28, 2010

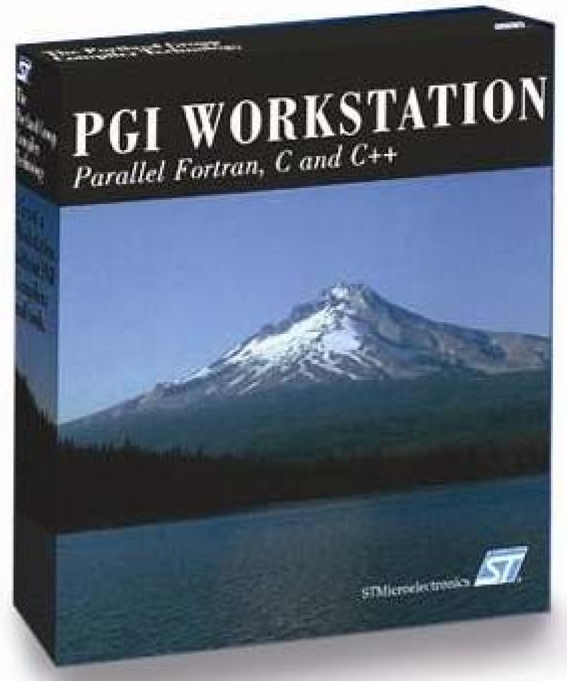
Talk Roadmap

- ❑Introduction to The Portland Group / PGI
- ❑Introduction to GPU programming
- ❑CUDA Fortran Overview
- ❑PGI Accelerator programming model
- ❑PGPROF and Compiler Feedback
- ❑Future directions, challenges, Q&A

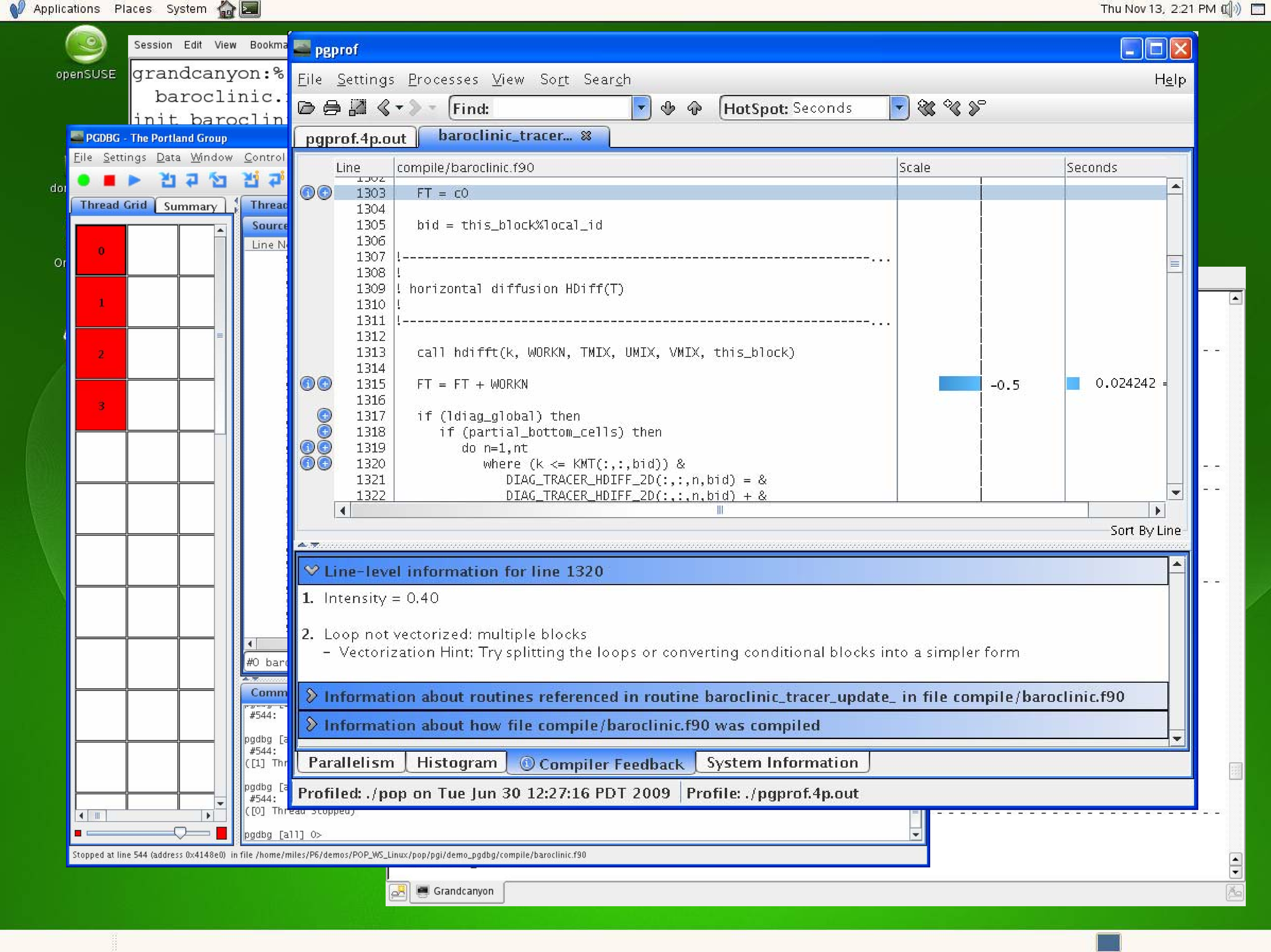
PGI Workstation / Server / CDK

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

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



Self-contained OpenMP/MPI/Accelerator Development Solution



```
grandcanyon:%
  baroclinic.
init_baroclin
```

Line	compile/baroclinic.f90	Scale	Seconds
1303	`FT = c0`		
1304			
1305	`bid = this_block%local_id`		
1306			
1307	`!-----------------------------------------------------------------...`		
1308	`!`		
1309	`! horizontal diffusion HDiff(T)`		
1310	`!`		
1311	`!-----------------------------------------------------------------...`		
1312			
1313	`call hdifft(k, WORKN, TMIX, UMIX, VMIX, this_block)`		
1314			
1315	`FT = FT + WORKN`	-0.5	0.024242
1316			
1317	`if (ldiag_global) then`		
1318	`   if (partial_bottom_cells) then`		
1319	`      do n=1,nt`		
1320	`         where (k <= KMT(:,:,bid)) &`		
1321	`            DIAG_TRACER_HDIFF_2D(:,:,n,bid) = &`		
1322	`            DIAG_TRACER_HDIFF_2D(:,:,n,bid) + &`		

Sort By Line

Line-level information for line 1320

1. Intensity = 0.40

2. Loop not vectorized: multiple blocks
   - Vectorization Hint: Try splitting the loops or converting conditional blocks into a simpler form

Information about routines referenced in routine baroclinic_tracer_update_ in file compile/baroclinic.f90

Information about how file compile/baroclinic.f90 was compiled

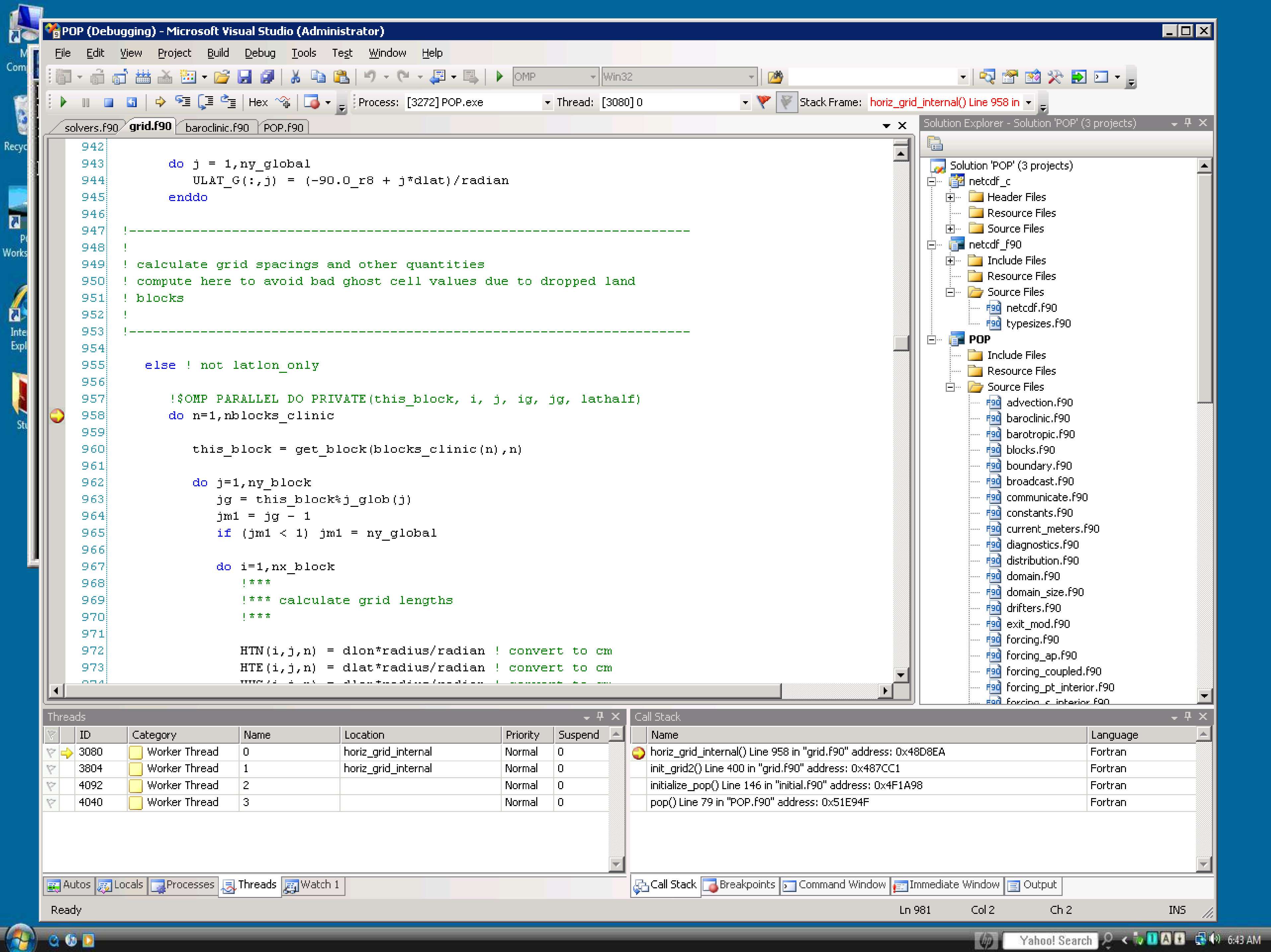
Parallelism | Histogram | Compiler Feedback | System Information

Profiled: ./pop on Tue Jun 30 12:27:16 PDT 2009 | Profile: ./pgprof.4p.out

```
#0 baro
#544:
pgdbg [a
#544:
([1] Thr
pgdbg [a
#544:
([0] Thread stopped)
pgdbg [all] 0>
```

Thread Grid: 0 1 2 3

Stopped at line 544 (address 0x4148e0) in file /home/miles/P6/demos/POP_WS_Linux/pop/pgi/demo_pgdbg/compile/baroclinic.f90


POP (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Tools Test Window Help
OMP
Win32
Hex
Process: [3272] POP.exe
Thread: [3080] 0
Stack Frame: horiz_grid_internal() Line 958 in
solvers.f90 grid.f90 baroclinic.f90 POP.f90
Solution Explorer - Solution 'POP' (3 projects)

```fortran
      do j = 1,ny_global
         ULAT_G(:,j) = (-90.0_r8 + j*dlat)/radian
      enddo

!-----------------------------------------------------------------------
!
! calculate grid spacings and other quantities
! compute here to avoid bad ghost cell values due to dropped land
! blocks
!
!-----------------------------------------------------------------------

   else ! not latlon_only

      !$OMP PARALLEL DO PRIVATE(this_block, i, j, ig, jg, lathalf)
      do n=1,nblocks_clinic

         this_block = get_block(blocks_clinic(n),n)

         do j=1,ny_block
            jg = this_block%j_glob(j)
            jm1 = jg - 1
            if (jm1 < 1) jm1 = ny_global

            do i=1,nx_block
               !***
               !*** calculate grid lengths
               !***

               HTN(i,j,n) = dlon*radius/radian ! convert to cm
               HTE(i,j,n) = dlat*radius/radian ! convert to cm
```

Solution 'POP' (3 projects)
- netcdf_c
  - Header Files
  - Resource Files
  - Source Files
- netcdf_f90
  - Include Files
  - Resource Files
  - Source Files
    - netcdf.f90
    - typesizes.f90
- POP
  - Include Files
  - Resource Files
  - Source Files
    - advection.f90
    - baroclinic.f90
    - barotropic.f90
    - blocks.f90
    - boundary.f90
    - broadcast.f90
    - communicate.f90
    - constants.f90
    - current_meters.f90
    - diagnostics.f90
    - distribution.f90
    - domain.f90
    - domain_size.f90
    - drifters.f90
    - exit_mod.f90
    - forcing.f90
    - forcing_ap.f90
    - forcing_coupled.f90
    - forcing_pt_interior.f90

Threads

	ID	Category	Name	Location	Priority	Suspend
⇨	3080	Worker Thread	0	horiz_grid_internal	Normal	0
	3804	Worker Thread	1	horiz_grid_internal	Normal	0
	4092	Worker Thread	2		Normal	0
	4040	Worker Thread	3		Normal	0

Call Stack

Name	Language
horiz_grid_internal() Line 958 in "grid.f90" address: 0x48D8EA	Fortran
init_grid2() Line 400 in "grid.f90" address: 0x487CC1	Fortran
initialize_pop() Line 146 in "initial.f90" address: 0x4F1A98	Fortran
pop() Line 79 in "POP.f90" address: 0x51E94F	Fortran

Autos Locals Processes Threads Watch 1
Call Stack Breakpoints Command Window Immediate Window Output
Ready Ln 981 Col 2 Ch 2 INS
Yahoo! Search 6:43 AM

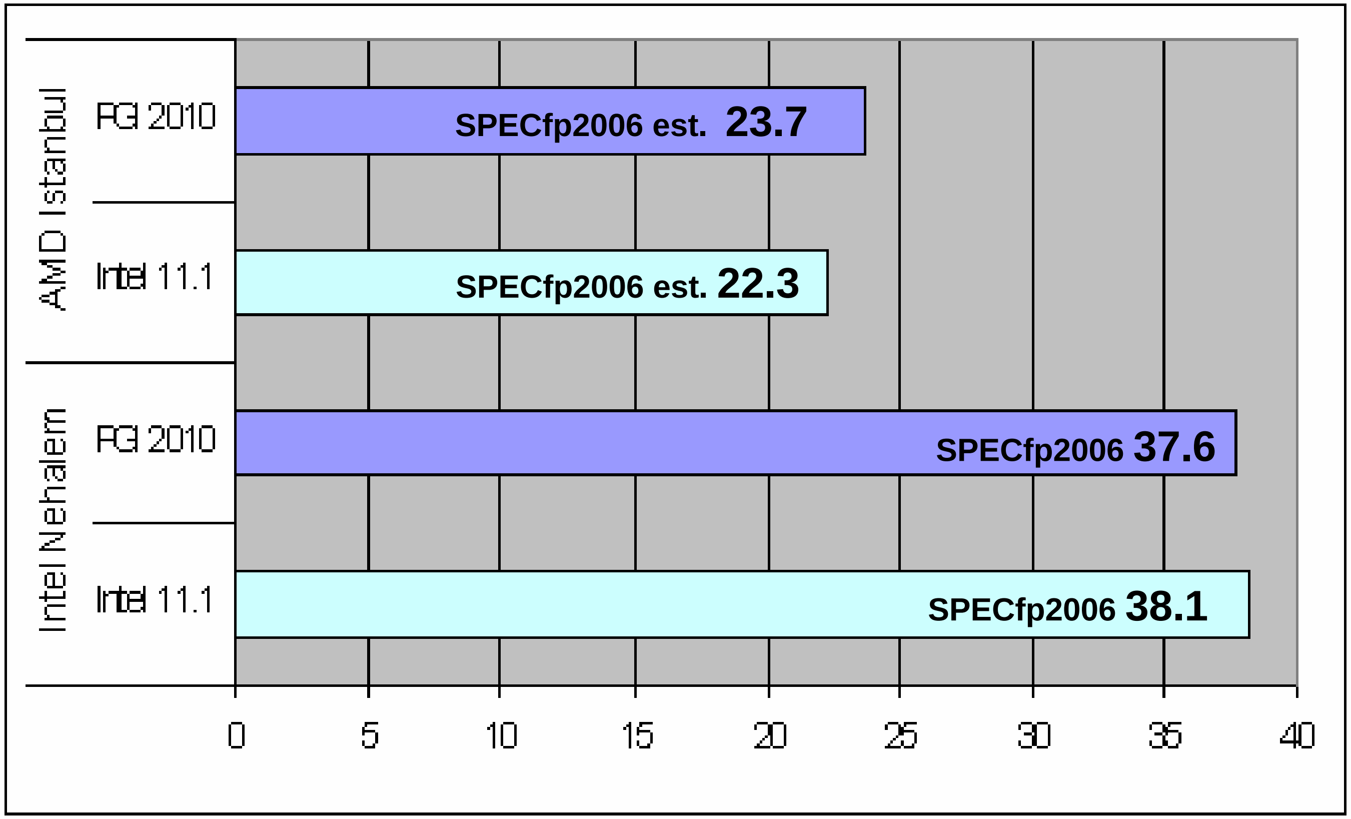
PGI® Compilers & Tools Positioning

- **Not intended to replace infrastructure compilers (GCC/VC++)**
  - Compilers & tools dedicated to scientific computing, where utilization of latest architecture features and speed on generated code is #1 criteria
- **HPC-focused Compilers & Tools technologies**
  - State of the art local, global and interprocedural 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
- **GPU/Accelerator Compilers & Tools**

PGI® 2010 New 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
- PVF Windows/MSMPI Cluster/Parallel Debugging
  - Debug Fortran & C MSMPI cluster applications within Visual Studio
  - PGI Accelerator and CUDA Fortran support
- Compiler Enhancements
  - F2003 – several new language features
  - Latest EDG 4.1 C++ front-end – more g++/VC++ compatible
  - 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)

Multicore X64 Performance



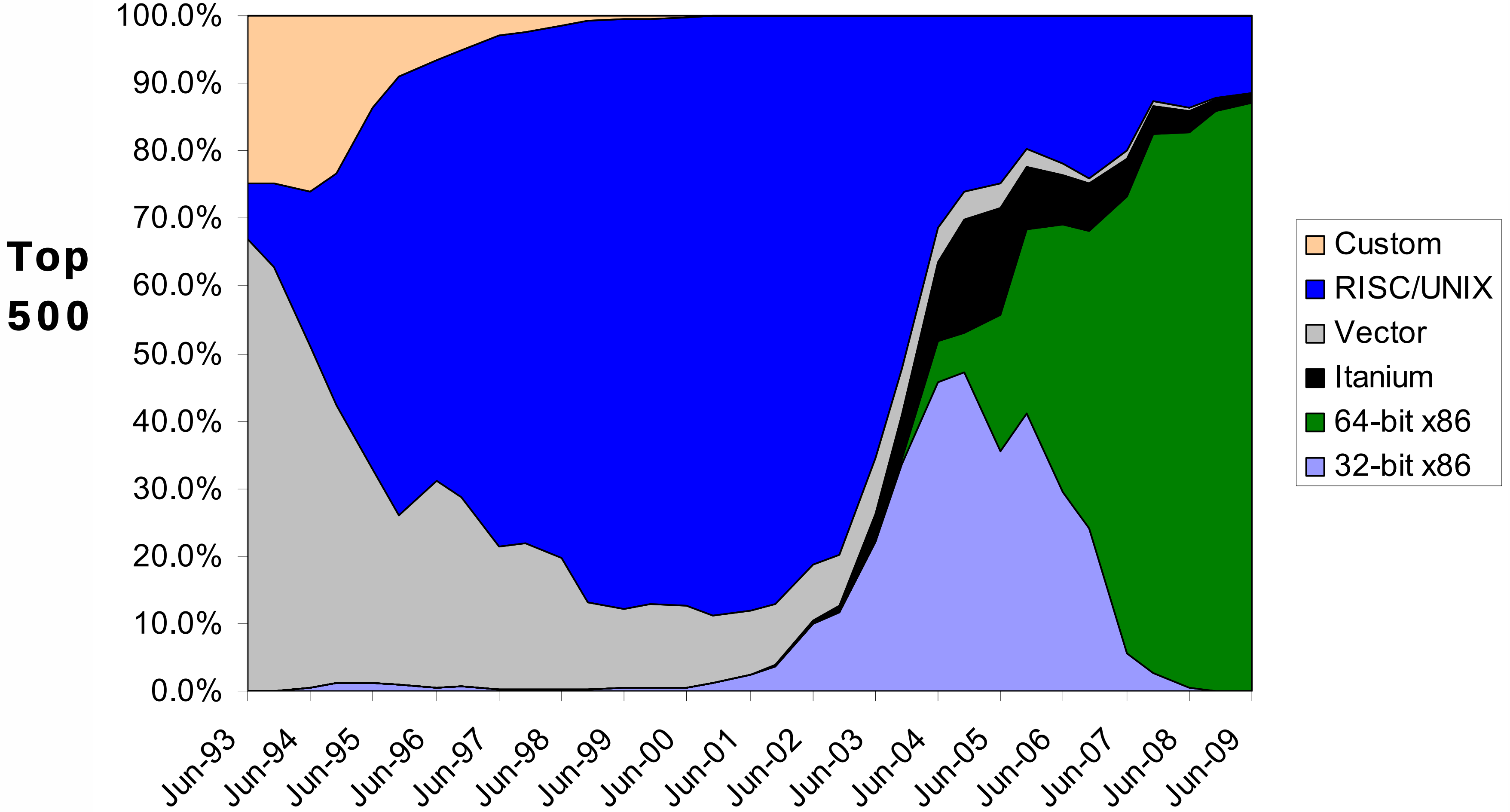
SPEC® and SPECfp® are registered trademarks of the Standard Performance Evaluation Corporation (SPEC) (www.spec.org)
Competitive benchmark results stated above reflect results performed by The Portland Group during the week of November 8th, 2009.
The Intel Nehalem system used is a Dell R610 using 2 Intel 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.

Talk Roadmap

- ❑ Introduction to The Portland Group / PGI
- ❑ Introduction to GPU programming
- ❑ CUDA Fortran Overview
- ❑ PGI Accelerator programming model
- ❑ PGPROF performance profiler
- ❑ Future directions, challenges, Q&A

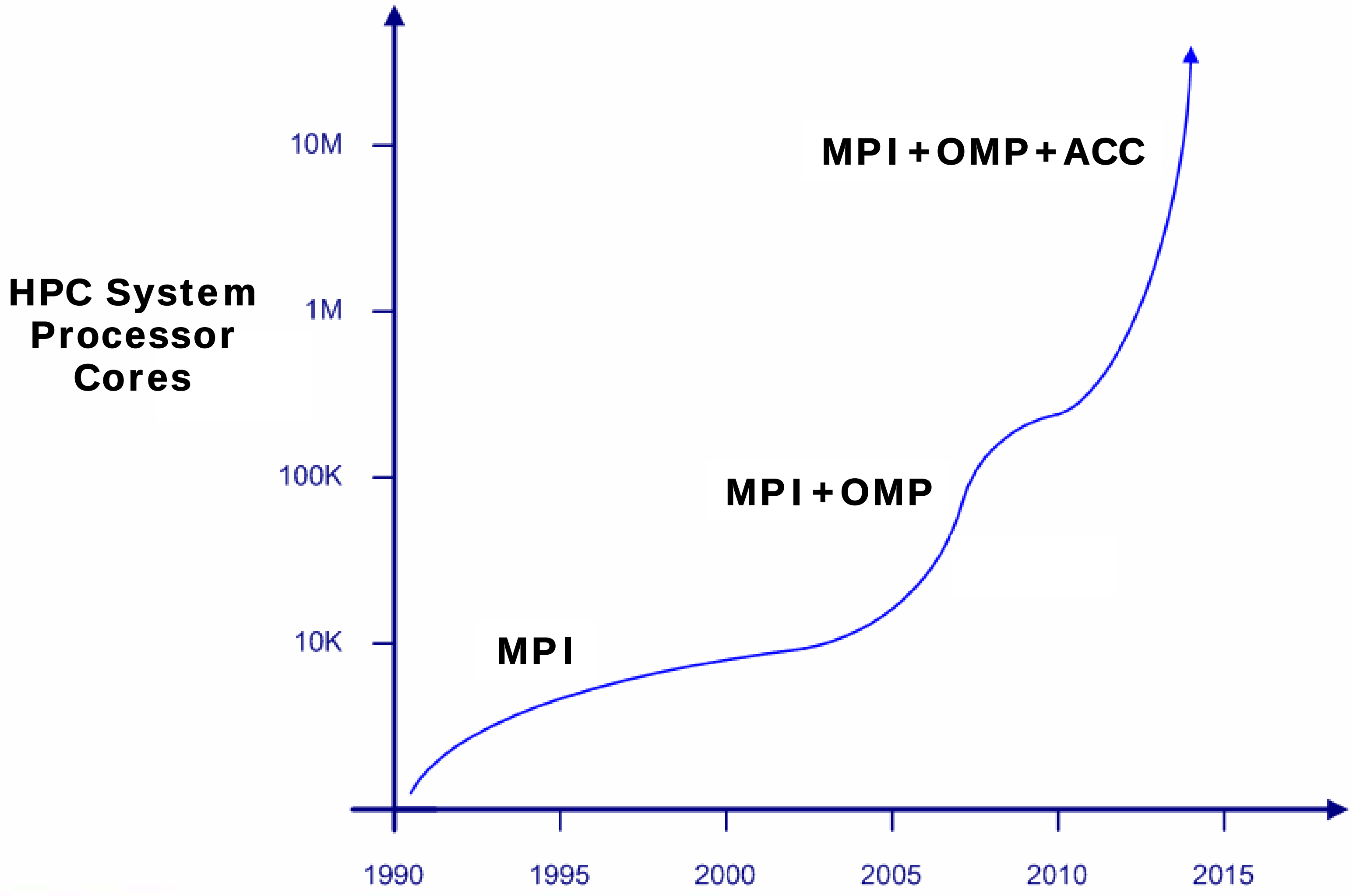
HPC Hardware Trends

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

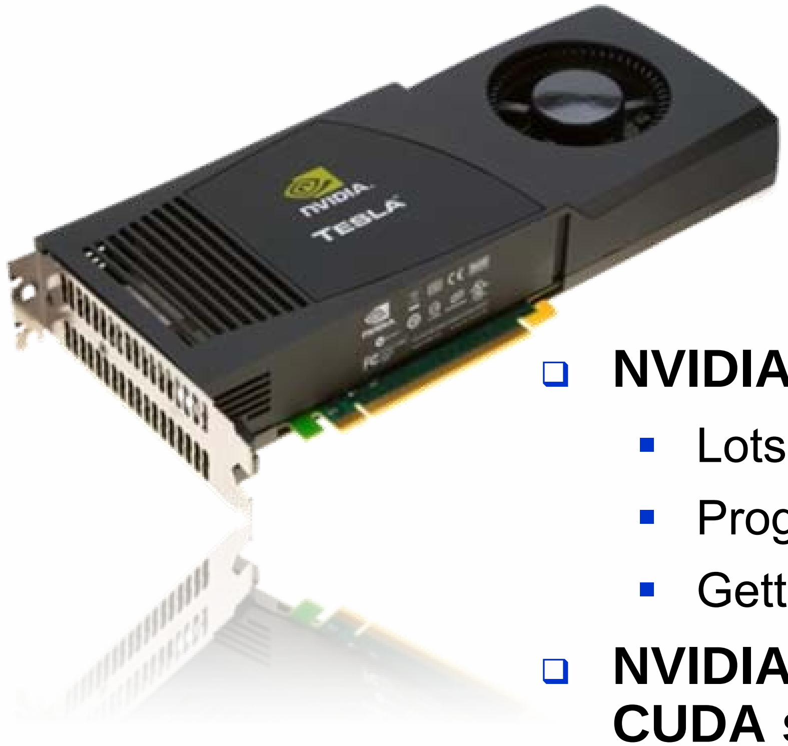


Compilers & Programming Models Must Evolve for Each New Generation of HPC Hardware

Expect 20M Core systems in the Next Few Years



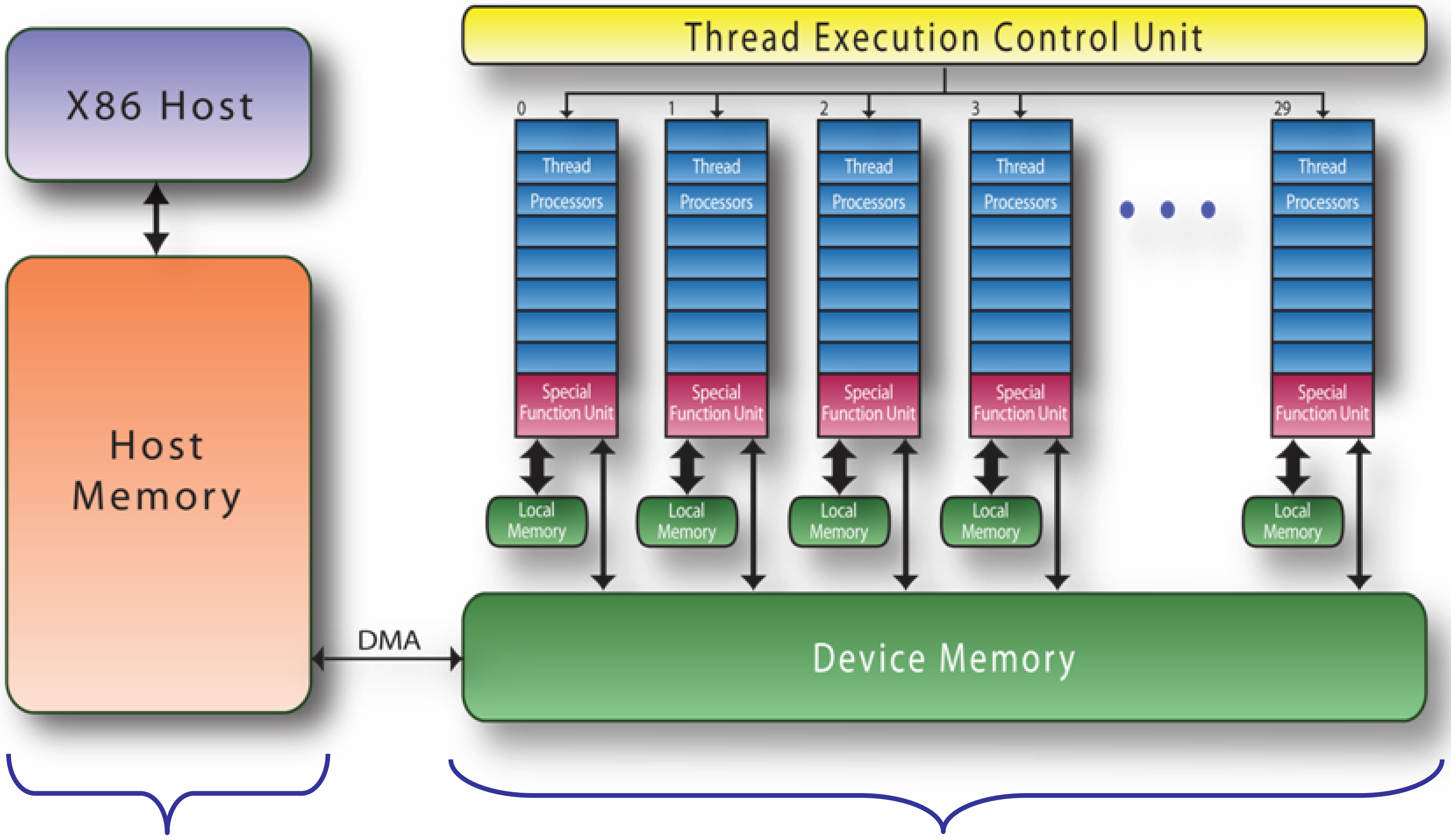
PGI Accelerator Compilers for x64+GPU systems



- **NVIDIA TESLA C1060**
  - 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 CUDA Fortran
- **Following slides describe how PGI Accelerator compilers address this challenge**
- **PGI goal is to do for GPU programming what OpenMP did for Posix Threads**

Emerging Cluster Node Architecture

Commodity Multicore x86 + Commodity Manycore GPUs



4 – 16 CPU Cores

128 – 2048 GPU/Accelerator Cores

Simple Fortran 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 Fortran 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
```

Basic CUDA C Matrix Multiply Kernel for an NVIDIA GPU

```
extern "C" __global__ void
mmkernel( float* a,float* b,float* c,
          int la,int lb,int lc,int n,
          int m,int p )
{
    int i = blockIdx.x*64+threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
      sum += b[i+lb*k] * c[k+lc*j];
    a[i+la*j] = sum;
}
```

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c, int la, int lb, int lc, int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*128 + tx;  int j = blockIdx.y*4;
    __shared__ float cb0[128], cb1[128], cb2[128], cb3[128];

    float sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
    for( int ks = 0; ks < p; ks += 128 ){
      cb0[tx] = c[ks+tx+lc*j];     cb1[tx] = c[ks+tx+lc*(j+1)];
      cb2[tx] = c[ks+tx+lc*(j+2)]; cb3[tx] = c[ks+tx+lc*(j+3)];
      __syncthreads();
      for( int k = 0; k < 128; k+=4 ){
        float rb = b[i+lb*(k+ks)];
         sum0 += rb * cb0[k];    sum1 += rb * cb1[k];
          sum2 += rb * cb2[k];    sum3 += rb * cb3[k];
        rb = b[i+lb*(k+ks+1)];
         sum0 += rb * cb0[k+1]; sum1 += rb * cb1[k+1];
          sum2 += rb * cb2[k+1]; sum3 += rb * cb3[k+1];
        rb = b[i+lb*(k+ks+2)];
         sum0 += rb * cb0[k+2]; sum1 += rb * cb1[k+2];
          sum2 += rb * cb2[k+2]; sum3 += rb * cb3[k+2];
        rb = b[i+lb*(k+ks+3)];
         sum0 += rb * cb0[k+3]; sum1 += rb * cb1[k+3];
          sum2 += rb * cb2[k+3]; sum3 += rb * cb3[k+3];
      }
      __syncthreads();
    }
    a[i+la*j] = sum0;     a[i+la*(j+1)] = sum1;
     a[i+la*(j+2)] = sum2; a[i+la*(j+3)] = sum3;
}
```

Optimized CUDA C Matrix Multiply Kernel

Host-side CUDA C Matrix Multiply GPU Control Code

```
cudaMalloc( &bp, memsize );
cudaMalloc( &ap, memsize );
cudaMalloc( &cp, memsize );

cudaMemcpy( bp, b, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( cp, c, memsize, cudaMemcpyHostToDevice );
cudaMemcpy( ap, a, memsize, cudaMemcpyHostToDevice );

dim3 threads( 128 );
dim3 blocks( matsize/128, matsize/4 );
mmkernel<<<blocks,threads>>>(ap,bp,cp,nsize,nsize,
                    nsize,matsize,matsize,matsize);

cudaMemcpy( a, ap, memsize, cudaMemcpyDeviceToHost );

cudaFree( ap );
cudaFree( bp );
cudaFree( cp );
```

Talk Roadmap

- ❑The Portland Group / PGI
- ❑Introduction to GPU programming
- ❑CUDA Fortran Overview
- ❑PGI Accelerator programming model
- ❑PGPROF performance profiler
- ❑Future directions, challenges, Q&A

What is CUDA Fortran?

- CUDA Fortran is an analog to NVIDIA's CUDA C language
- Fortran language extensions and CUDA API give HPC developers direct control over all aspects of GPU programming
- Co-defined by PGI and NVIDIA, implemented in the PGI 2010 Fortran 95/03 compiler
- Supported on Linux, MacOS and Windows

CUDA Fortran
Matrix Multiply Host Routine

```
   . . .
   subroutine mmul( A, B, C )                       ! Host routine to drive mmul_kernel
      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
       use cudafor                                     !    CUDA Fortran GPU kernel
    contains
       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
    . . .
```

CUDA C vs CUDA Fortran

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

- CUDA Fortran
  - NO texture memory
  - supports Runtime API
  - NO Driver API
  - allocate, deallocate
  - assignments
  - NO OpenGL interoperability
  - NO Direct3D interoperability
  - No textures
  - arrays one-based
  - threadidx/blockidx 1-based
  - allocatable are device/host
  - pinned attribute

Talk Roadmap

- ❑Introduction to The Portland Group / PGI
- ❑Introduction to GPU programming
- ❑CUDA Fortran Overview
- ❑PGI Accelerator programming model
- ❑PGPROF performance profiler
- ❑Future directions, challenges, Q&A

PGI Directive-based Fortran Matrix Multiply for x64+GPU

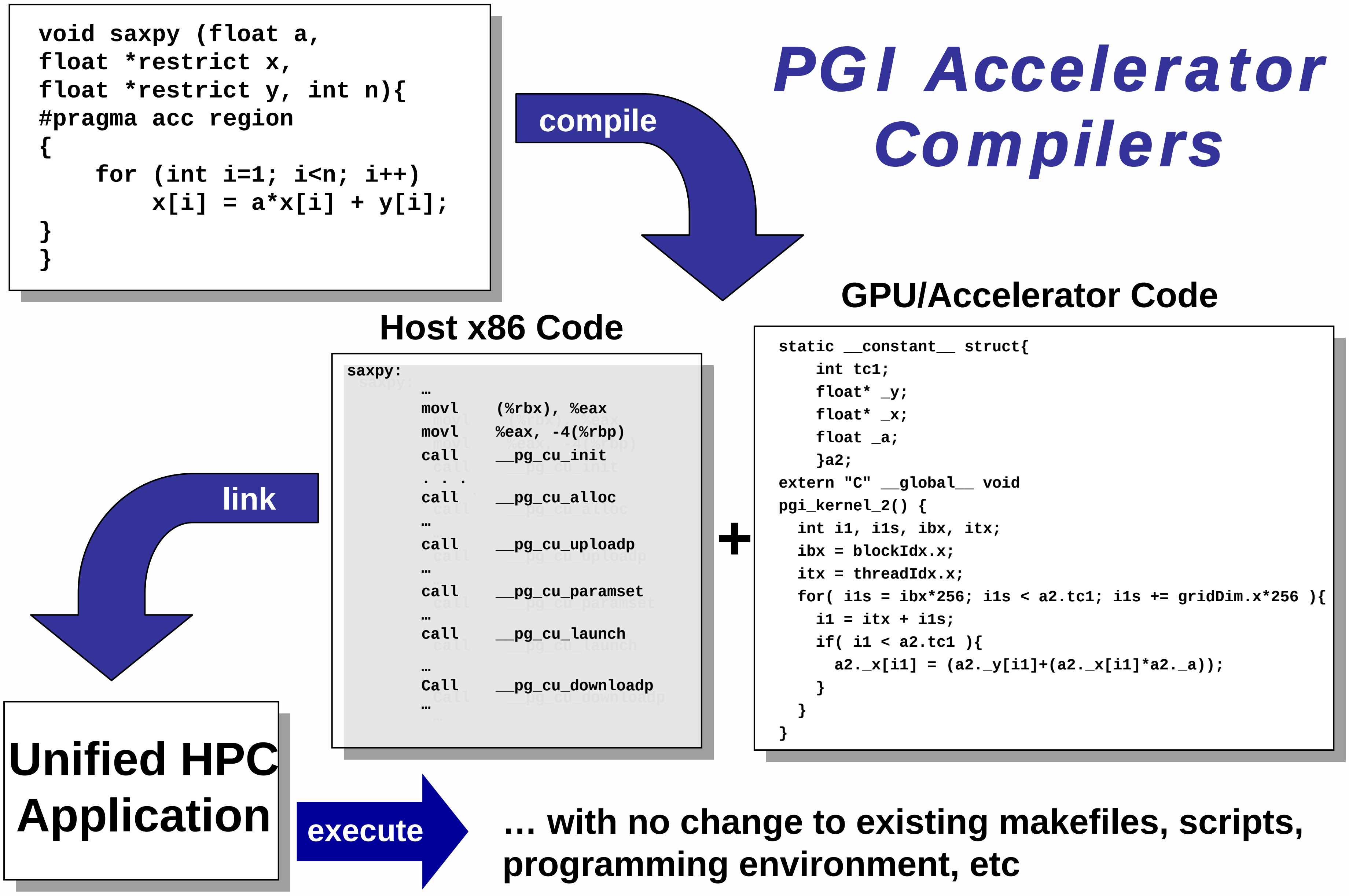
```
!$acc region
      do j = 1, m
         do k = 1, p
            do i = 1,n
               a(i,j) = a(i,j) + b(i,k)*c(k,j)
            enddo
         enddo
      enddo
!$acc end region
```

PGI Accelerator vs CUDA Fortran?

- The **PGI Accelerator** programming model is a high-level ***implicit*** model for x64+GPU systems, similar to OpenMP for multi-core
  - Supported in both the PGFORTRAN and PGCC compilers (no C++, yet)
  - Offload compute-intensive code to a GPU accelerator using directives
  - Programs remain 100% standard-compliant and portable
  - Makes GPGPU programming and optimization incremental, and easily approachable by application domain experts

- **PGI CUDA Fortran** is an ***explicit*** model requiring direct control of:
  - Splitting source into host code and GPU kernel subroutines/functions
  - Allocation of page-locked host memory, GPU device main memory, GPU constant memory and GPU shared memory
  - All data movement between host memory and GPU memory hierarchy
  - Definition of thread/block grids and launching of compute kernels
  - Synchronization of threads within a CUDA thread group
  - Asynchronous launch of GPU kernels, synchronization with host CPU
  - All CUDA Runtime API features and functions

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];
}
}
```

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));
    }
  }
}
```

Unified HPC Application

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

PGI Accelerator 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

Directives for Tuning Data Movement and Kernel Mapping

```
!$acc region copyin(b(1:n,1:p),c(1:p,1:m))
!$acc& copy(a(1:n,1:m))
!$acc do parallel, unroll(4)
      do j = 1, m
!$acc do parallel, vector(128)
         do i = 1, n
!$acc do seq, unroll(4), cache(c)
            do k = 1, p
               a(i,j) = a(i,j) + b(i,k)*c(k,j)
            enddo
         enddo
      enddo
!$acc end region
```

Same Basic Model for C - pragmas

```
#pragma acc region
{
    for(int opt = 0; opt < optN; opt++){
        float S = h_StockPrice[opt],
              X = h_OptionStrike[opt],
              T = h_OptionYears[opt];
        float sqrtT = sqrtf(T);
        float d1 = (logf(S/X) +
                (Riskfree + 0.5 * Volatility * Volatility) * T)
                / (Volatility * sqrtT);
        float    d2 = d1 - Volatility * sqrtT;
        float cndd1 = CND(d1);
        float cndd2 = CND(d2);
        float expRT = expf(- Riskfree * T);
        h_CallResult[opt] = (S*cndd1-X*expRT*cndd2);
        h_PutResult[opt] = (X*expRT*(1.0-cndd2)-S*(1.0-cndd1));
    }
}
```

PGI Accelerator Directives Syntax

- Accelerator compute region directive
  - Fortran syntax

```
!$acc region [clause [,clause]…]
```

  - C syntax

```
#pragma acc region [clause [,clause]…]
{
…
}
```

- Accelerator loop mapping directive
  - Fortran syntax

```
!$acc do [clause [,clause]…]
do-loop
```

  - C syntax

```
#pragma acc for [clause [,clause]…]
for-loop
```

PGI Accelerator Directives Syntax

- **Accelerator combined directive**
  - **Fortran syntax**

```
!$acc region do [clause [,clause]…]
do-loop
```

  - **C syntax**

```
#pragma region for [clause [,clause]…]
for-loop
```

PGI Accelerator Directives Syntax

- Accelerator data region directive
  - Fortran syntax

    ```
    !$acc data region [clause [,clause]…]
    ```

  - C syntax

    ```
    #pragma acc data region [clause [,clause]…]
    {
    …
    }
    ```

- Accelerator date update directive
  - Fortran syntax

    ```
    !$acc {updatein|updateout} clause [[,clause]…]
    ```

  - C syntax

    ```
    #pragma acc {updatein|updateout} clause [[,clause]…]
    ```

PGI Accelerator Region Declarative / Executable Clauses

Clause	Region Scope / Type
`if (`*`cond`*`)`	compute
`copy (`*`list`*`)`	compute, data, declaration
`copyin (`*`list`*`)`	compute, data, declaration
`copyout (`*`list`*`)`	compute, data, declaration
`local (`*`list`*`)`	compute, data, declaration
`mirror (`*`list`*`)`	data, declaration (Fortran)
`reflected (`*`list`*`)`	compute, data, declaration (Fortran)
`updatein (list)`	compute, data, executable
`updateout (list)`	compute, data, executable

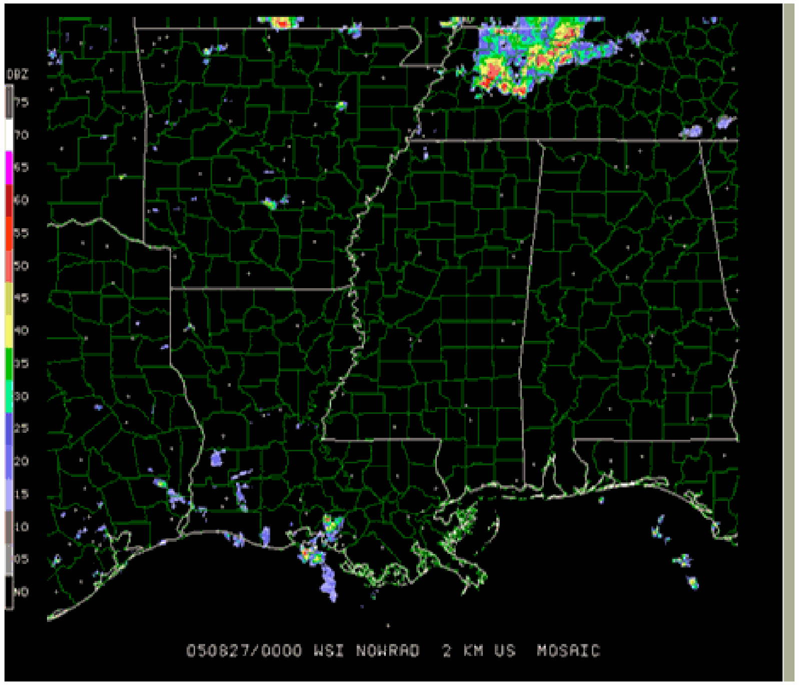
PGI Accelerator Loop Mapping Clauses

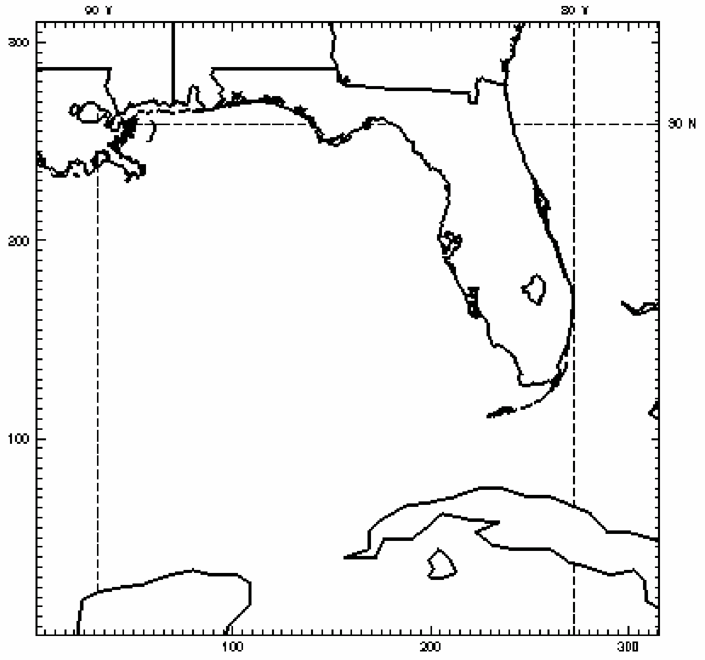
Clause	Scope
`host [(`*`width`*`)]`	loop
`parallel [(`*`width`*`)]`	loop
`seq [(`*`width`*`)]`	loop
`vector [(`*`width`*`)]`	loop
`private (`*`list`*`)`	loop
`kernel`	loop
`unroll (`*`width`*`)*`	loop
`cache (`*`list`*`)*`	loop

*** Not supported in PGI 10.1**

PGI Accelerator Programming Model Performance on a real Application

WRF 3.1.1 - Weather Research and Forecast Model





- Used at hundreds of sites in research and/or production
- Over 400,000 lines of Fortran & C source code
- Already MPI and OpenMP-enabled for multicore clusters
- Good candidate for a multi-core x86+NVIDIA port

```
#if ( RWORDSIZE == 4 )
#  define VREC vsrec
#  define VSQRT vssqrt
#else
#  define VREC vrec
#  define VSQRT vsqrt
#endif

!Including inline expansion statistical function
MODULE module_mp_wsm5
!
!
   REAL, PARAMETER, PRIVATE :: dtcldcr     = 120. ! maximum time step for minor loops
   REAL, PARAMETER, PRIVATE :: n0r = 8.e6         ! intercept parameter rain
   REAL, PARAMETER, PRIVATE :: avtr = 841.9       ! a constant for terminal velocity of
rain
   REAL, PARAMETER, PRIVATE :: bvtr = 0.8         ! a constant for terminal velocity of
rain
   REAL, PARAMETER, PRIVATE :: r0 = .8e-5         ! 8 microm  in contrast to 10 micro m
   REAL, PARAMETER, PRIVATE :: peaut = .55        ! collection efficiency
   REAL, PARAMETER, PRIVATE :: xncr = 3.e8        ! maritime cloud in contrast to 3.e8 in
tc80
   REAL, PARAMETER, PRIVATE :: xmyu = 1.718e-5    ! the dynamic viscosity kgm-1s-1
   REAL, PARAMETER, PRIVATE :: avts = 11.72       ! a constant for terminal velocity of
snow
   REAL, PARAMETER, PRIVATE :: bvts = .41         ! a constant for terminal velocity of
snow
   REAL, PARAMETER, PRIVATE :: n0smax =  1.e11    ! maximum n0s (t=-90C unlimited)
   REAL, PARAMETER, PRIVATE :: lamdarmax = 8.e4   ! limited maximum value for slope
parameter of rain
   REAL, PARAMETER, PRIVATE :: lamdasmax = 1.e5   ! limited maximum value for slope
parameter of snow
   REAL, PARAMETER, PRIVATE :: lamdagmax = 6.e4   ! limited maximum value for slope
parameter of graupel
   REAL, PARAMETER, PRIVATE :: dicon = 11.9       ! constant for the cloud-ice diamter
```

WRF WSM52D Performance (Seconds)

2.67Ghz Intel Nehalem Server vs NVidia Tesla C1060

Host Cores	GPUs	Total Time	GPU Data	GPU Compute	Notes
1	-	236	-	-	Nehalem, 1 core
2	-	125	-	-	Nehalem, 2 cores*
4	-	70	-	-	Nehalem, 4 cores*
1	1	36.15	10.64	25.40	Initial GPU kernel**
1	1	29.79	10.84	18.85	Tuned kernel schedule **
2	2	16.67	6.29	10.50	Tuned kernel schedule **
1	1	19.72	10.75	8.85	Work-in-progress kernel **
2	2	12.00	6.87	5.29	Work-in-progress kernel **
1	1	26.35	9.04	7.17	Hand-coded CUDA C

* Parallelized using OpenMP with the Intel 11.1 compiler

** Using PGI Accelerator directives to offload computations to the GPU

Porting WSM52D to NVIDIA Tesla
using PGI Accelerator Fortran vs CUDA C

- **1500** **Lines of code ported**
- **3** **weeks for the CUDA C Port**
- **4** **days for the PGI Accelerator Fortran port**
- **5x** **PGI Accelerator porting efficiency advantage**
- **1.23x** **CUDA C performance efficiency advantage**

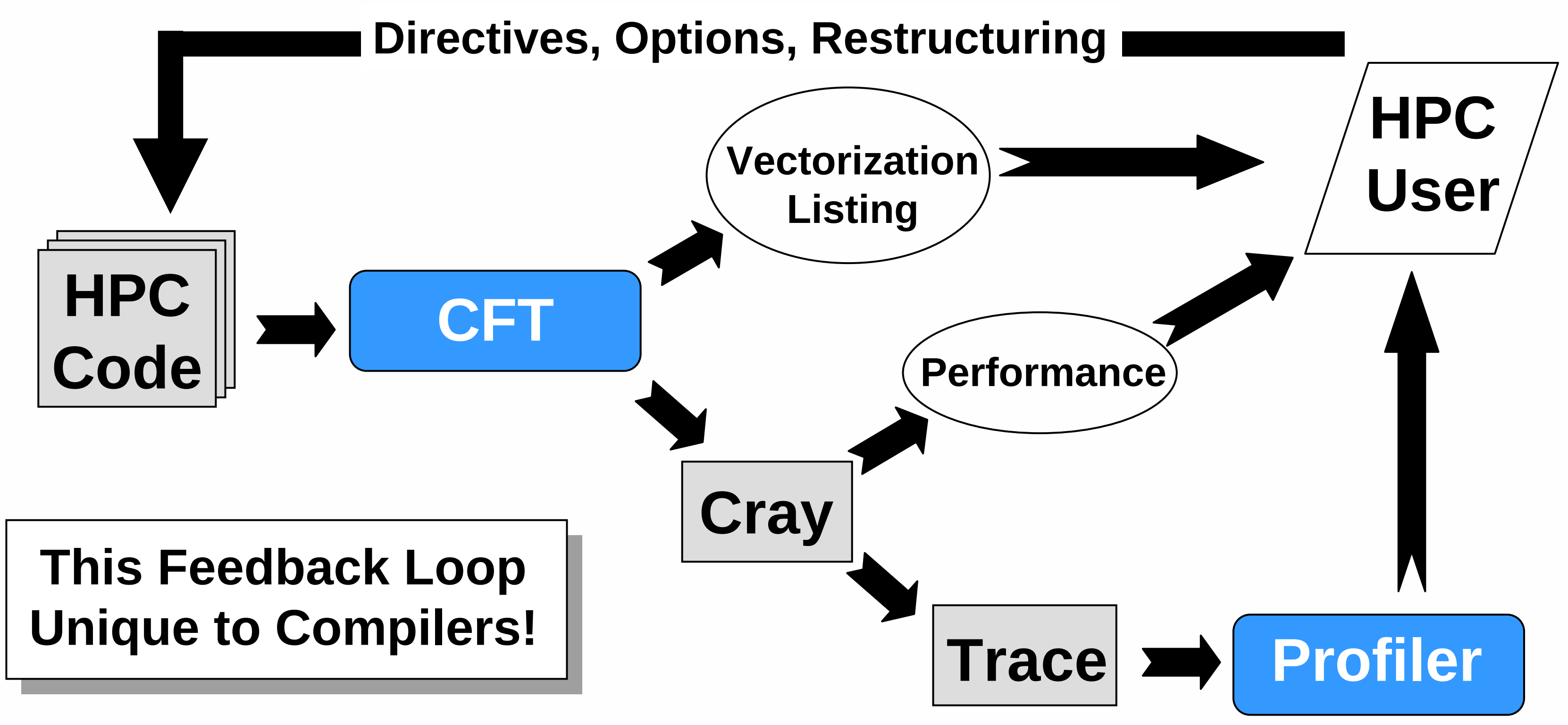
Using PGI Accelerator Fortran a full WRF port to NVIDIATesla GPUs is feasible in less than 1 programmer year

Talk Roadmap

- ❑The Portland Group / PGI
- ❑Introduction to GPU programming
- ❑CUDA Fortran Overview
- ❑PGI Accelerator programming model
- ❑PGPROF and compiler feedback
- ❑Future directions, challenges, Q&A

How did we make Vectors Work?

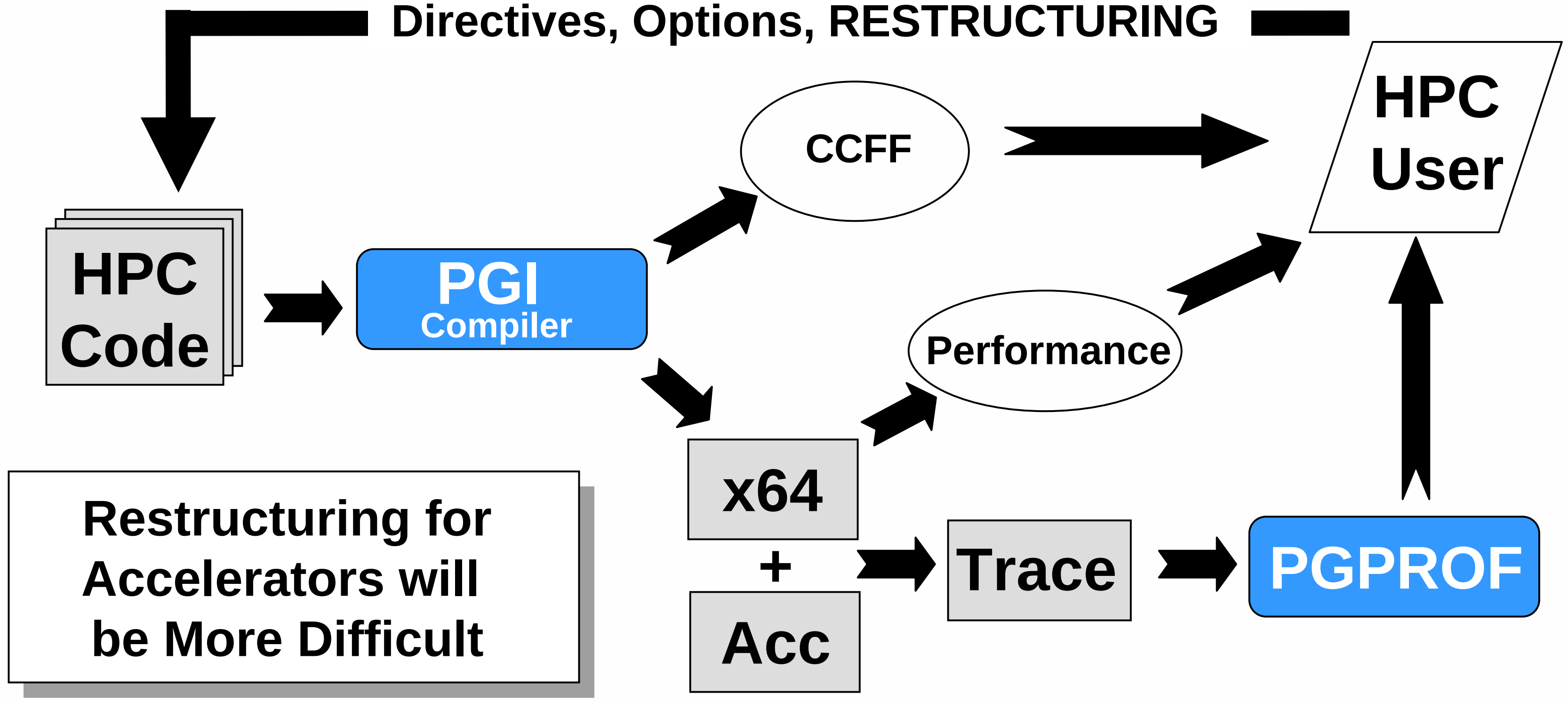
Compiler-to-Programmer Feedback – a classic “Virtuous Cycle”



Directives, Options, Restructuring
HPC Code
CFT
Vectorization Listing
HPC User
Performance
Cray
Trace
Profiler
This Feedback Loop Unique to Compilers!

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

Compiler-to-Programmer Feedback



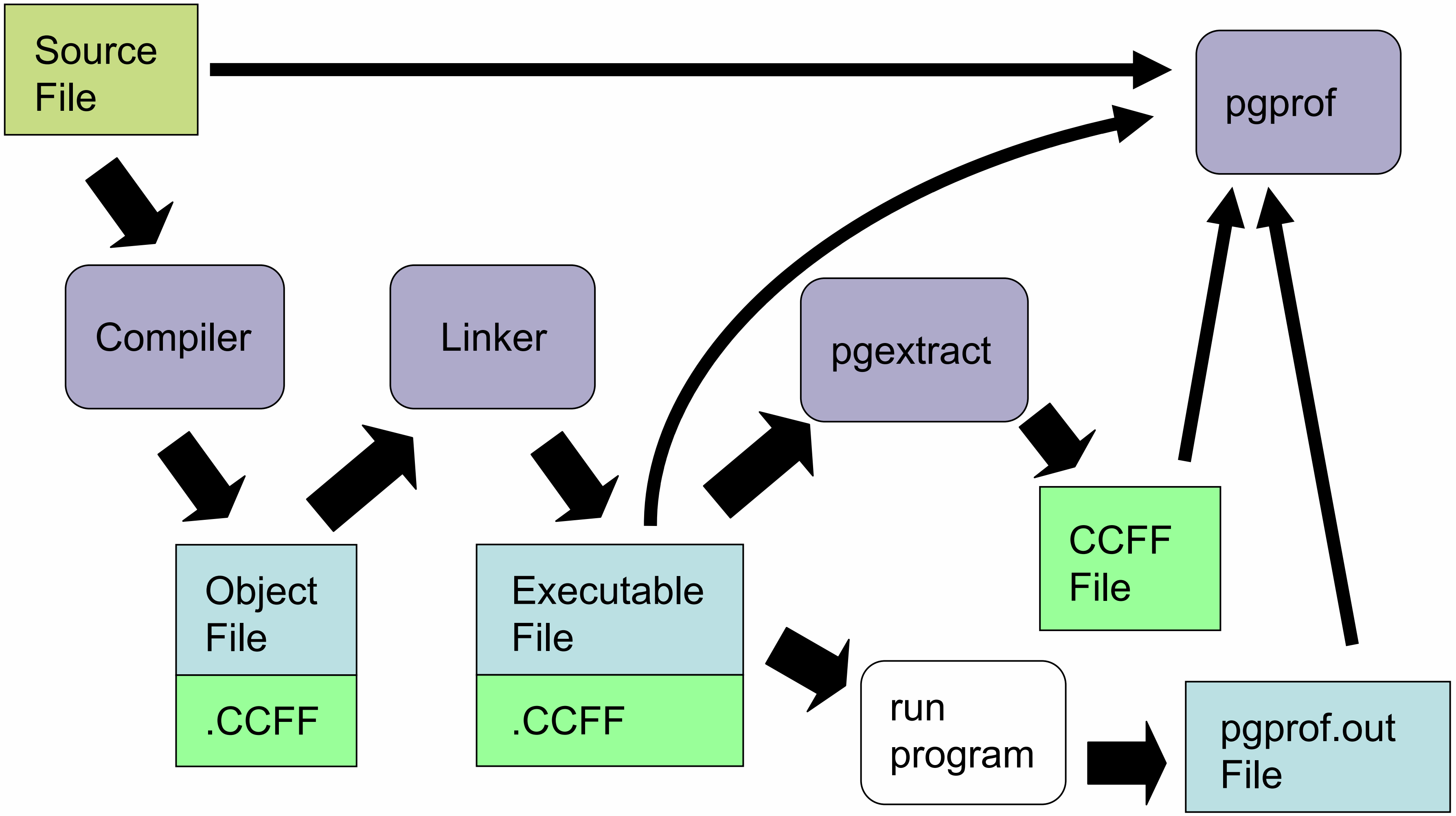
Restructuring for Accelerators will be More Difficult

Compiler-to-User Feedback

```
% pgfortran -fast -ta=nvidia -Minfo mm.F90
mm1:
      6, Generating copyout(a(1:m,1:m))
         Generating copyin(c(1:m,1:m))
         Generating copyin(b(1:m,1:m))
      7, Loop is parallelizable
      8, Loop is parallelizable
         Accelerator kernel generated
          7, !$acc do parallel, vector(16)
          8, !$acc do parallel, vector(16)
     11, Loop carried reuse of 'a' prevents parallelization
     12, Loop is parallelizable
         Accelerator kernel generated
          7, !$acc do parallel, vector(16)
         11, !$acc do seq
             Cached references to size [16x16] block of 'b'
             Cached references to size [16x16] block of 'c'
         12, !$acc do parallel, vector(16)
             Using register for 'a'
```

Common Compiler Feedback Format

http://www.pgroup.com/resources/ccff.htm



Profiling an Accelerator Enabled application

- Step 1: Compile using *–ta=nvidia* flag to specify that target accelerator is NVIDIA, and *–Minfo=ccff* to embed compiler feedback into generated binary.

```
pgfortran –ta=nvidia –Minfo=ccff –fast –DPAD –DACC
  himeno.F90 –o himeno.exe
```

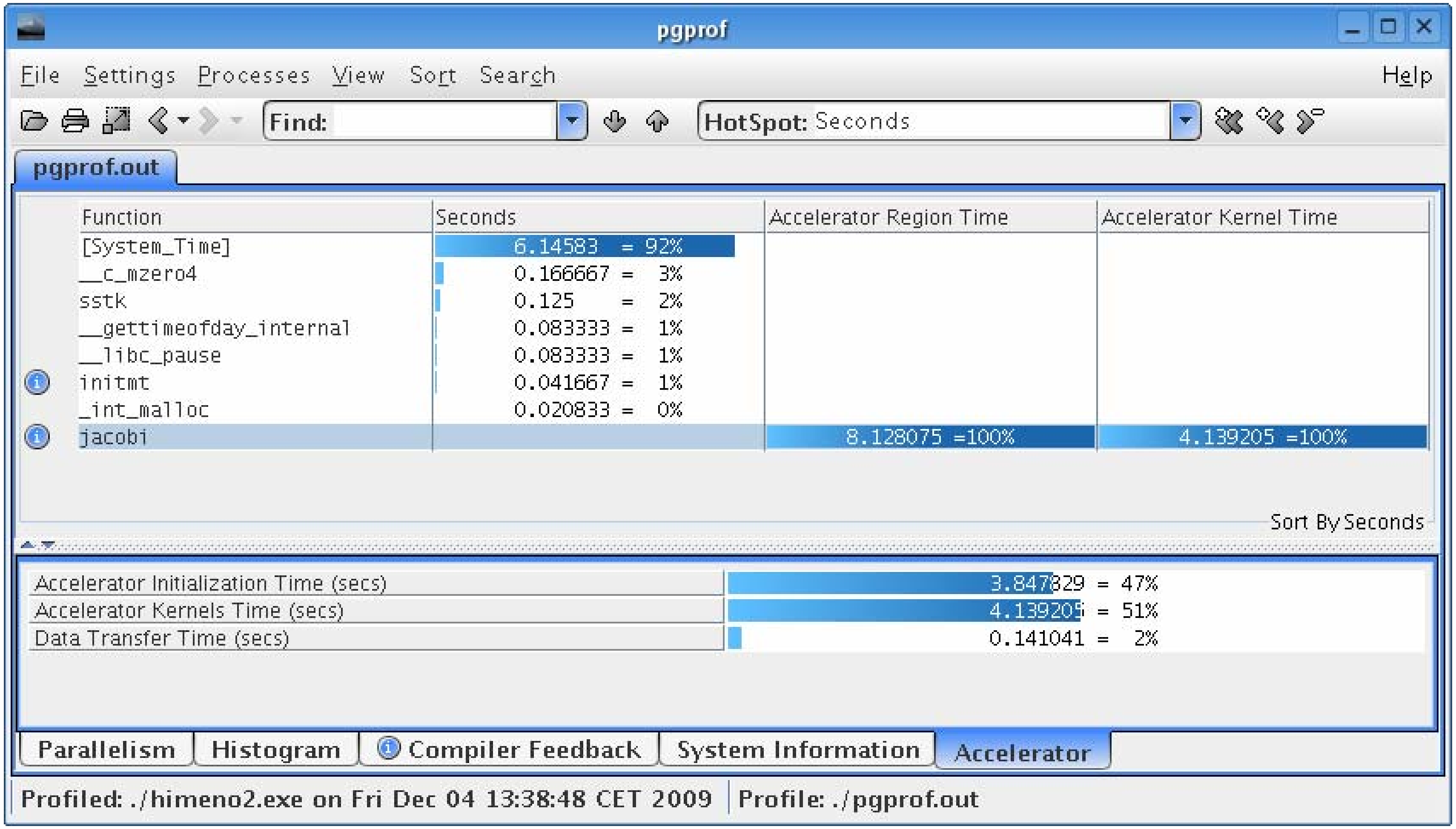
- Step 2: use *pgcollect* driver to launch your application and collect time spend on CPU/GPU.

```
pgcollect –time himeno.exe
```

- Step 3: use *pgprof* to browse profiling information collected by *pgcollect*.

```
pgprof –exe himeno.exe
```

Routine level Accelerator Region Profiling Information



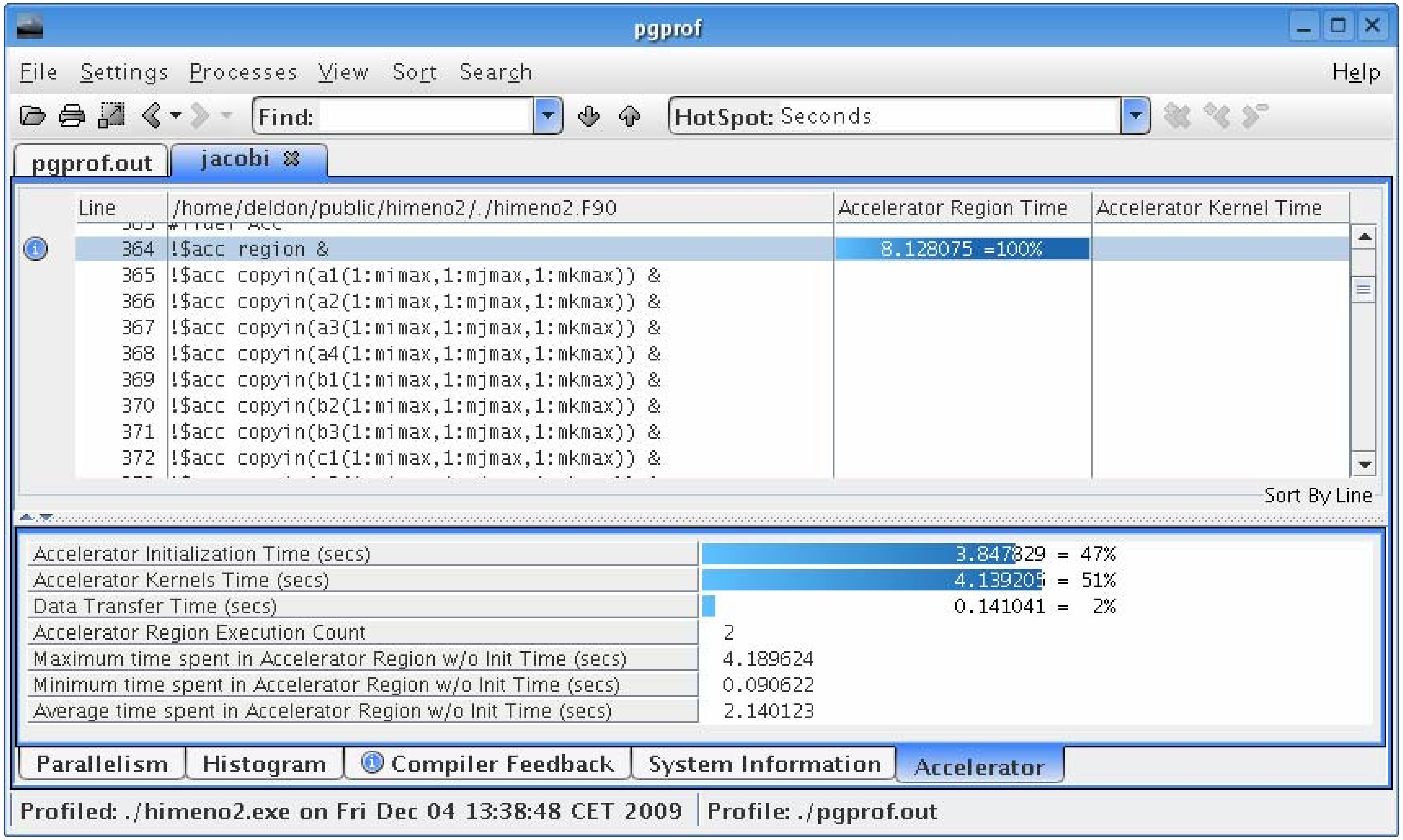
Function	Seconds	Accelerator Region Time	Accelerator Kernel Time
[System_Time]	6.14583 = 92%		
__c_mzero4	0.166667 = 3%		
sstk	0.125 = 2%		
__gettimeofday_internal	0.083333 = 1%		
__libc_pause	0.083333 = 1%		
initmt	0.041667 = 1%		
_int_malloc	0.020833 = 0%		
jacobi		8.128075 =100%	4.139205 =100%

Sort By Seconds

Accelerator Initialization Time (secs)	3.847829 = 47%
Accelerator Kernels Time (secs)	4.139205 = 51%
Data Transfer Time (secs)	0.141041 = 2%

Profiled: ./himeno2.exe on Fri Dec 04 13:38:48 CET 2009 | Profile: ./pgprof.out

Source level Accelerator Region Profiling Information

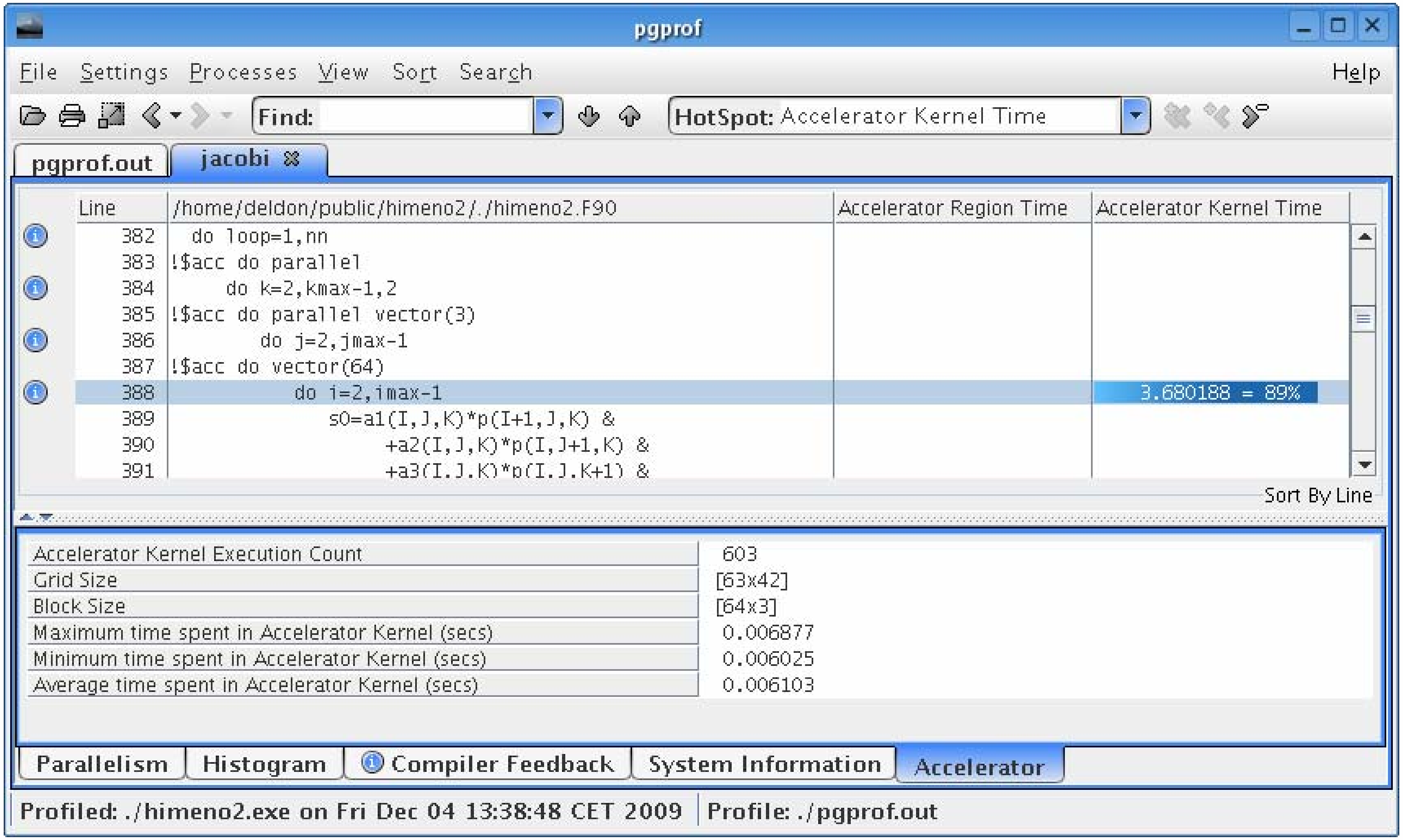


```
Line  /home/deldon/public/himeno2/./himeno2.F90            Accelerator Region Time   Accelerator Kernel Time
 364  !$acc region &                                        8.128075 =100%
 365  !$acc copyin(a1(1:mimax,1:mjmax,1:mkmax)) &
 366  !$acc copyin(a2(1:mimax,1:mjmax,1:mkmax)) &
 367  !$acc copyin(a3(1:mimax,1:mjmax,1:mkmax)) &
 368  !$acc copyin(a4(1:mimax,1:mjmax,1:mkmax)) &
 369  !$acc copyin(b1(1:mimax,1:mjmax,1:mkmax)) &
 370  !$acc copyin(b2(1:mimax,1:mjmax,1:mkmax)) &
 371  !$acc copyin(b3(1:mimax,1:mjmax,1:mkmax)) &
 372  !$acc copyin(c1(1:mimax,1:mjmax,1:mkmax)) &
```

Accelerator Initialization Time (secs)	3.847829 = 47%
Accelerator Kernels Time (secs)	4.139205 = 51%
Data Transfer Time (secs)	0.141041 = 2%
Accelerator Region Execution Count	2
Maximum time spent in Accelerator Region w/o Init Time (secs)	4.189624
Minimum time spent in Accelerator Region w/o Init Time (secs)	0.090622
Average time spent in Accelerator Region w/o Init Time (secs)	2.140123

Profiled: ./himeno2.exe on Fri Dec 04 13:38:48 CET 2009 | Profile: ./pgprof.out

Source level Accelerator Kernel Profiling Information

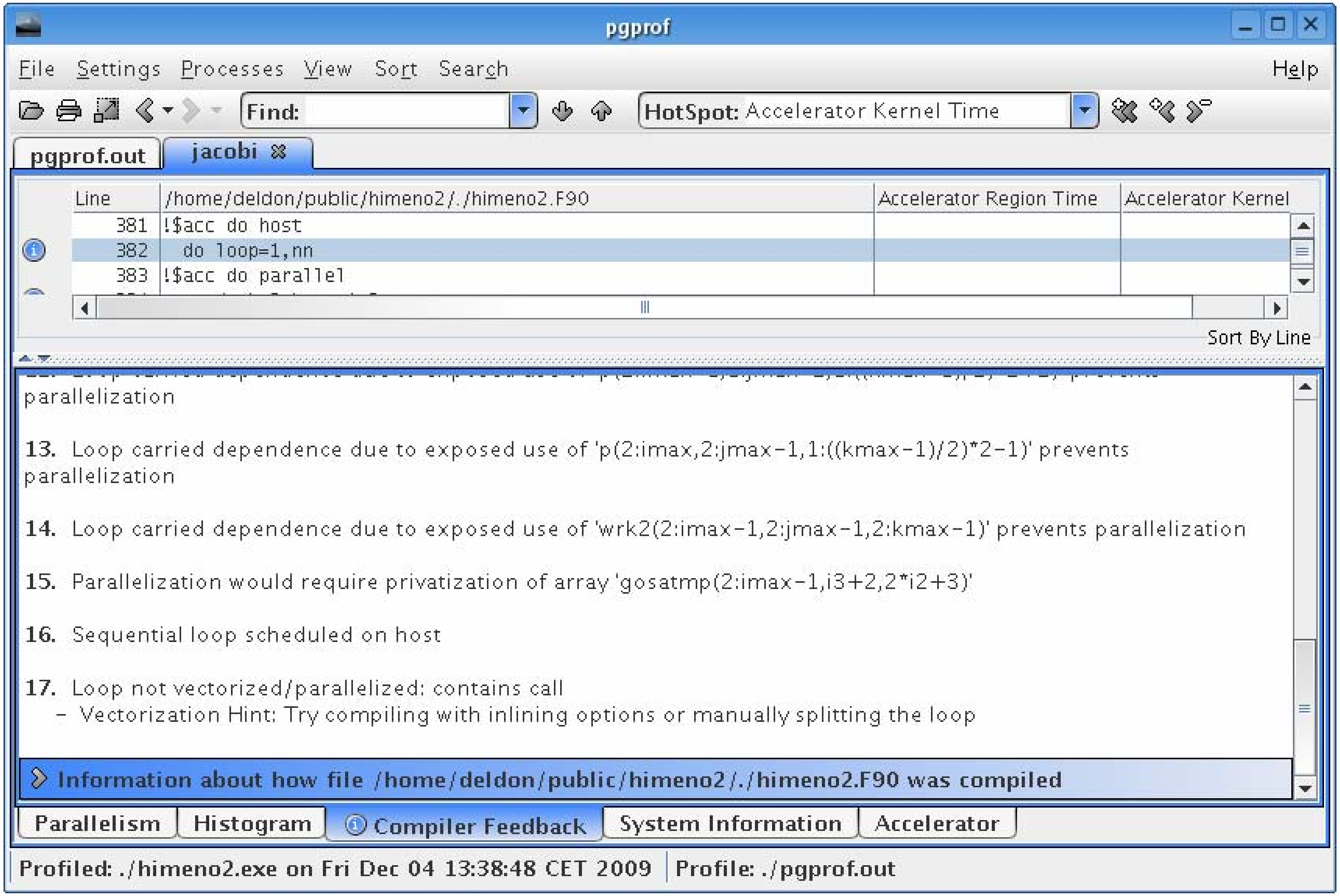


Line	/home/deldon/public/himeno2/./himeno2.F90	Accelerator Region Time	Accelerator Kernel Time
382	do loop=1,nn		
383	!$acc do parallel		
384	do k=2,kmax-1,2		
385	!$acc do parallel vector(3)		
386	do j=2,jmax-1		
387	!$acc do vector(64)		
388	do i=2,imax-1		3.680188 = 89%
389	s0=a1(I,J,K)*p(I+1,J,K) &		
390	+a2(I,J,K)*p(I,J+1,K) &		
391	+a3(I,J,K)*p(I,J,K+1) &		

Accelerator Kernel Execution Count	603
Grid Size	[63x42]
Block Size	[64x3]
Maximum time spent in Accelerator Kernel (secs)	0.006877
Minimum time spent in Accelerator Kernel (secs)	0.006025
Average time spent in Accelerator Kernel (secs)	0.006103

Profiled: ./himeno2.exe on Fri Dec 04 13:38:48 CET 2009 | Profile: ./pgprof.out

Accelerate using CCFF Information



Line	/home/deldon/public/himeno2/./himeno2.F90	Accelerator Region Time	Accelerator Kernel
381	!$acc do host		
382	do loop=1,nn		
383	!$acc do parallel		

parallelization

13. Loop carried dependence due to exposed use of 'p(2:imax,2:jmax-1,1:((kmax-1)/2)*2-1)' prevents parallelization

14. Loop carried dependence due to exposed use of 'wrk2(2:imax-1,2:jmax-1,2:kmax-1)' prevents parallelization

15. Parallelization would require privatization of array 'gosatmp(2:imax-1,i3+2,2*i2+3)'

16. Sequential loop scheduled on host

17. Loop not vectorized/parallelized: contains call
  - Vectorization Hint: Try compiling with inlining options or manually splitting the loop

Information about how file /home/deldon/public/himeno2/./himeno2.F90 was compiled

Profiled: ./himeno2.exe on Fri Dec 04 13:38:48 CET 2009 | Profile: ./pgprof.out

Availability and Additional Information

- **PGI 2010 Compilers & Tools** – available now! See www.pgroup.com for details
- **PGI Accelerator programming model** – supported for x64+NVIDIA targets in the PGI 2010 F95/03 and C99 compilers, available now; see http://www.pgroup.com/accelerate for a detailed specification, FAQ and related articles and white papers
- **CUDA Fortran** – supported on NVIDIA GPUs in PGI 2010 F95/03 compiler; see http://www.pgroup.com/resources/cudafortran.htm for a detailed specification

Talk Roadmap

- ❑Introduction to The Portland Group / PGI
- ❑Introduction to GPU programming
- ❑CUDA Fortran Overview
- ❑PGI Accelerator programming model
- ❑PGPROF performance profiler
- ❑Future directions, challenges, Q&A

Backup Slides

PGI 2010 Compilers F2003/C++ Language Support

- PGI Fortran 2003 incremental features
  - *Initial release of PGI 2010*: pointer reshaping, procedure pointers and statement, abstract interfaces, ieee_exceptions module, ieee_arithmetic module
  - *Coming later in PGI 2010*: Object-oriented features

- PGC++/ PGCC enhancements
  - EDG release 4.1 front-end with enhanced GNU and Microsoft compatibility, extern inline support, improved BOOST support, thread-safe exception handling

Intel AVX Support

- Intel Advanced Vector Extensions
  - New instructions
  - Wider vector registers, up to 2X floating point performance
- PGI F03/C/C++ compilers will be ready when AVX-enabled systems become available
- Can run with Intel simulator today
  - For those who like to experiment or test for correctness

AVX Example: Vector Add

FORTRAN

```fortran
subroutine vadd( a, b, c, n )
      real    a(n), b(n), c(n)
      integer i, n
      do i = 1, n
         c(i) = a(i) + b(i)
      enddo
end
```

SSE

```
x.LB1_438:
movups      (%r10,%rcx), %xmm0
movups      (%r9,%rcx), %xmm1
addps       %xmm0, %xmm1
movups      %xmm1, (%r8,%rcx)
movups      16(%r10,%rcx), %xmm0
movups      16(%r9,%rcx), %xmm1
addps       %xmm0, %xmm1
movups      %xmm1, 16(%r8,%rcx)
addq        $32, %rcx
subl        $8, %eax
testl       %eax, %eax
jg    .LB1_438
```

AVX

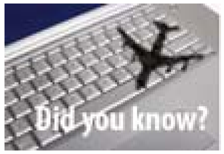
```
.LB1_477:
vmovups     (%r9,%rcx), %ymm0
vaddps      (%r10,%rcx), %ymm0, %ymm1
vmovups      %ymm1, (%r8,%rcx)
vmovups     32(%r9,%rcx), %ymm0
vaddps      32(%r10,%rcx), %ymm0, %ymm1
vmovups     %ymm1, 32(%r8,%rcx)
addq        $64, %rcx
subl        $16, %eax
testl       %eax, %eax
jg    .LB1_477
```

Invoking AVX on PGI 10.0

- pgfortran –tp sandybridge-64
- GNU Binutils 2.19.51 or newer
- Intel Software Development Emulator
  - http://software.intel.com/en-us/articles/intel-software-development-emulator/

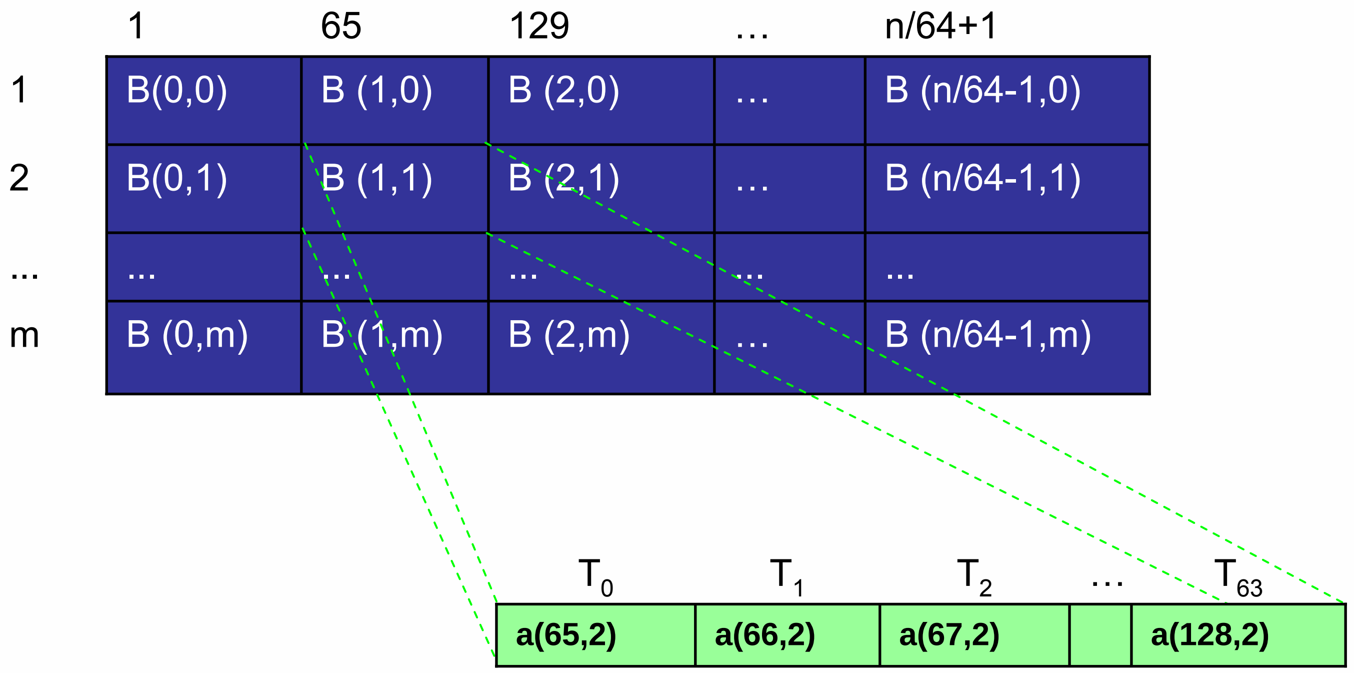
Cross-Platform and Mobile HPC Development

- PGI Workstation® on Linux, MacOS & Windows
  - Same C, C++, and Fortran compilers on all platforms
  - PGI Accelerator support and CUDA Fortran
  - MPI, OpenMP, PGDBG®, PGPROF®
- Cross-platform licensing
  - One license can cover all platforms
  - Floating license only
- 'Borrow' licensing

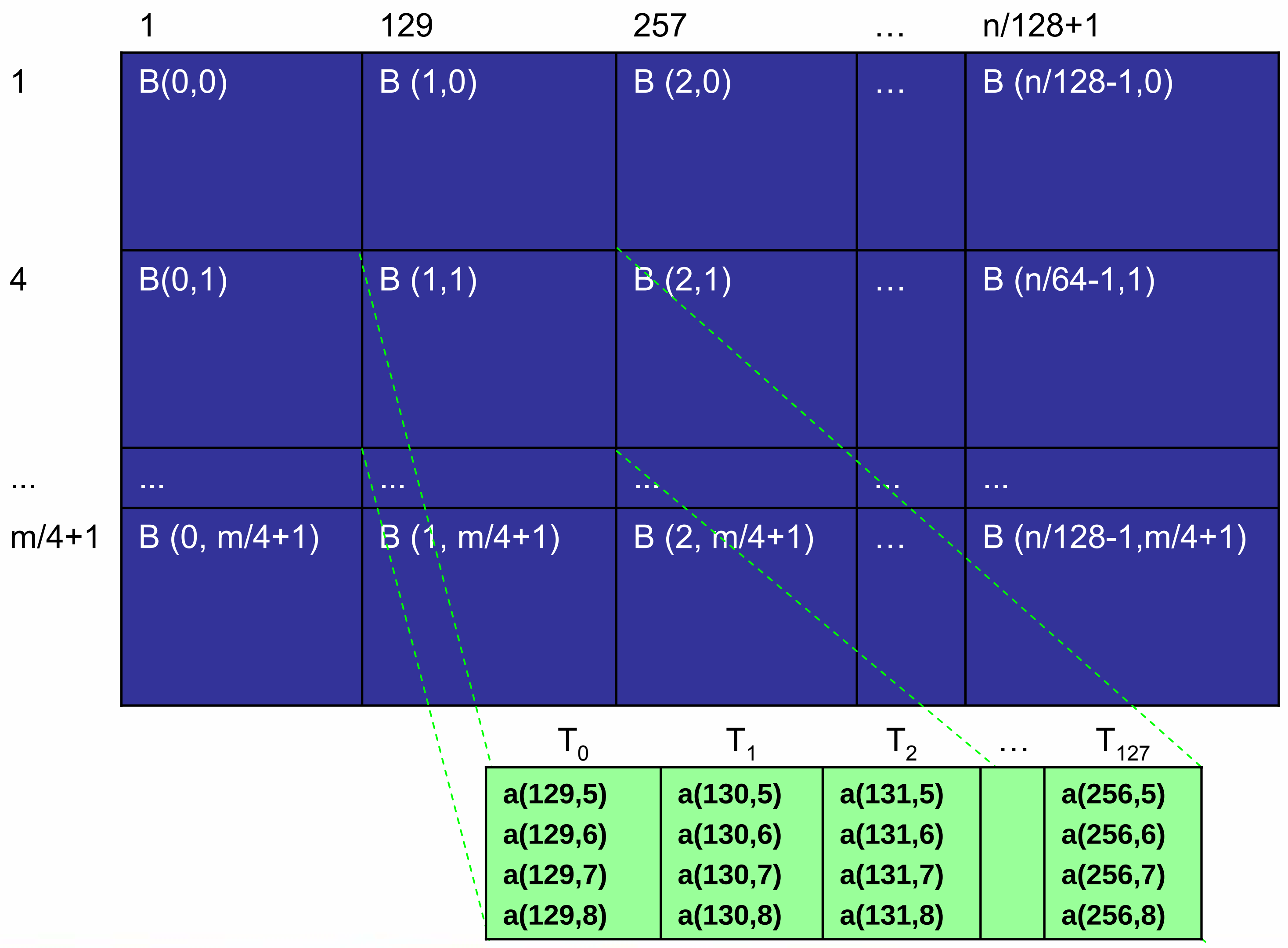


  - No separate license needed to work off-line on your notebook
  - Check out a floating license for your notebook for home or travel

Matrix Multiply Kernel Mapping



Optimized Matrix Multiply Kernel Mapping



	1	129	257	…	n/128+1
1	B(0,0)	B (1,0)	B (2,0)	…	B (n/128-1,0)
4	B(0,1)	B (1,1)	B (2,1)	…	B (n/64-1,1)
...	...	...	...	...	...
m/4+1	B (0, m/4+1)	B (1, m/4+1)	B (2, m/4+1)	…	B (n/128-1,m/4+1)

T_0	T_1	T_2	…	T_{127}

a(129,5)
a(129,6)
a(129,7)
a(129,8)
	a(130,5)
	a(130,6)
	a(130,7)
	a(130,8)
		a(131,5)
		a(131,6)
		a(131,7)
		a(131,8)
				a(256,5)
				a(256,6)
				a(256,7)
				a(256,8)

Porting WRF WS52D code to Accelerator model

- Ported an 800+ line loop from subroutine WSM52D in module WSM5, an ice microphysics model
- Code restructuring required
    - Step 1: move invariant computation out of the loop
    - Step 2: loop interchange on several inner loops
    - Step 3: move loop surrounding call to WSM52D into the routine itself. In fact, WSM52D was working on 2D slices of a 3D array, with this modification, WSM52D routine works on the complete 3D array.

Porting WRF WS52D code to PGI Accelerator model

- Experiment with accelerator directives
  - Step 1: add '!$acc do parallel' directives to 2 outer loops; compiler couldn't generate kernel due to arrays that need to be privatized
  - Step 2: add 'private(…)' clause to get arrays privatized. Compiler generates an accelerator kernel, but its mapping on GPU was ineficient due to selected block size and loop iteration count. We end up with a significant slow down.
  - Step 3: Tune data transfer using copyin(…)/copyout(…) clauses to ensure contiguous data transfers on large arrays
  - Step 4: Better use of parallel/vector clauses to tune the kernel mapping
- Combining OpenMP/Accelerator directives
  - Leveraged OpenMP directives to enable use of multiple GPU

Accelerator Directives Processing

- Live variable and array region analysis to augment information in region directives and determine in / out datasets for the region
- Dependence and parallelism analysis to augment information in loop directives and determine loops that can be executed in parallel
- Select mapping of loop(s) onto hardware parallelism, SIMD/vector and MIMD/parallel dimensions, strip mining if necessary
- Extract the kernel or kernels, generate GPU code for each kernel
- Lots of opportunity for optimization of kernel code - loop unrolling, software data cache usage, register lifetime management (minimize register usage to maximize multithreading potential)
- Generate host code to drive and launch kernels, allocate GPU memory, copy data from host to GPU, copy results back to host, continue on host, generate host version of loop(s)
- Generate feedback to programmer