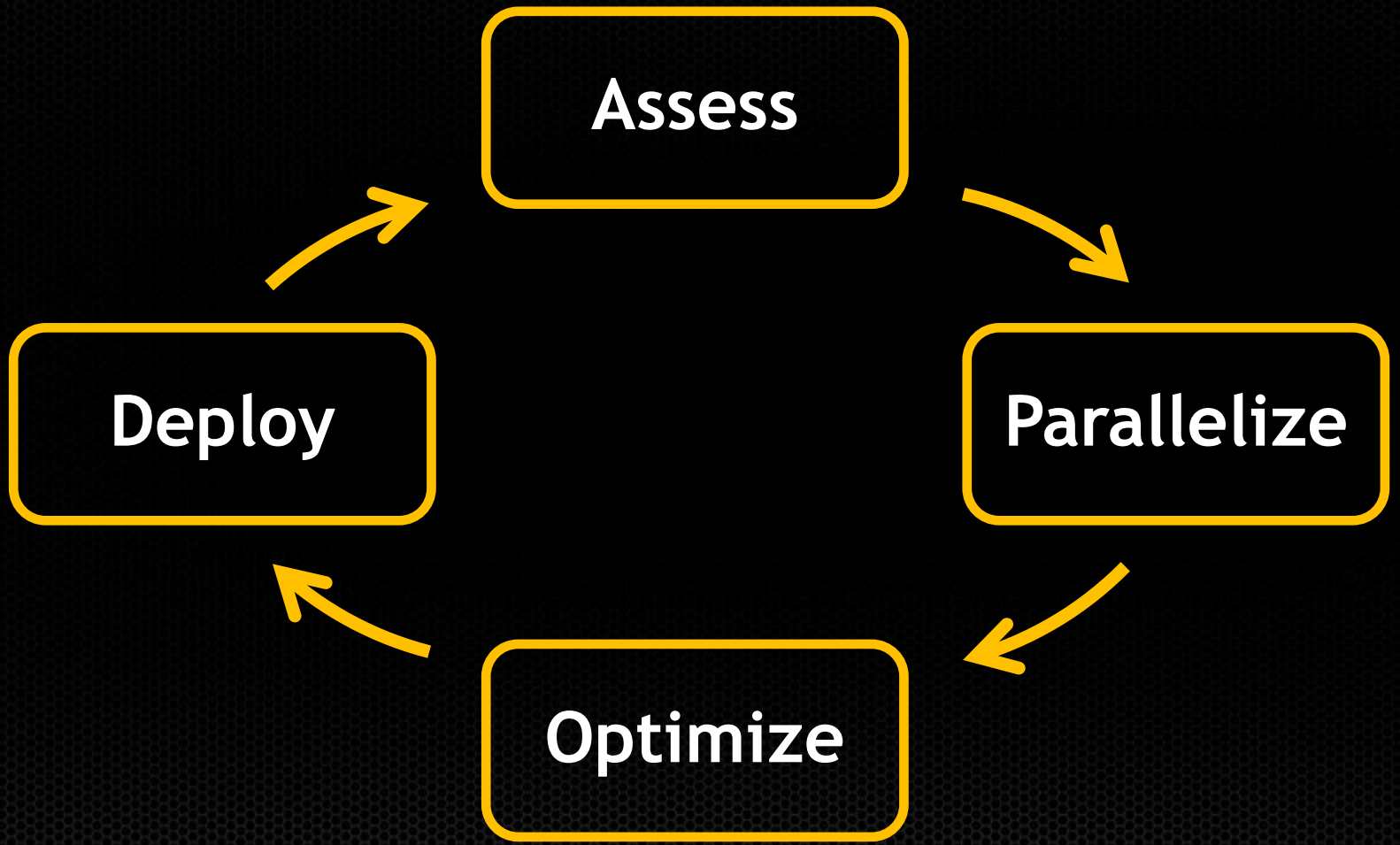

NVIDIA
OPENACC MINI-HACKATHON
ORNL – November 2016

Optimization Techniques and Strategies

- Use a disciplined approach
- Locate a profiler you know and trust
- Probably need some understanding of the underlying hardware
- Lots of information available:
  - http://www.pgroup.com/resources/openacc_tips_fortran.htm
  - http://www.nvidia.fr/content/EMEAI/tesla/openacc/pdf/Top-12-Tricks-for-Maximum-Performance-C.pdf
  - http://www.pgroup.com/resources/...
  - Kepler_Tuning_Guide.pdf

APOD: A Systematic Path to Performance

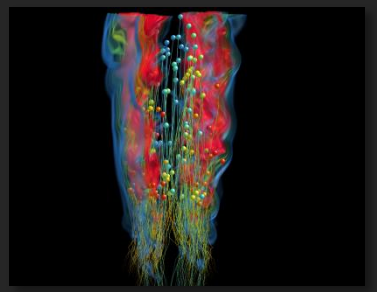


Focus on Parallelism and Data locality

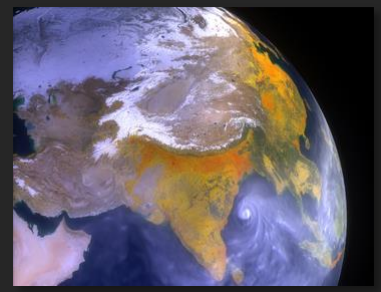
With directives, tuning work focuses on *exposing parallelism* and *expressing data locality*, which makes codes inherently better

Example: Application tuning work using directives for Titan system at ORNL

S3D
Research more efficient combustion with next-generation fuels



- **Tuning top 3 kernels (90% of runtime)**
- ***3 to 6x faster on CPU+GPU vs. CPU+CPU***
- **But also improved all-CPU version by 50%**



CAM-SE
Answer questions about specific climate change adaptation and mitigation scenarios

- **Tuning top key kernel (50% of runtime)**
- **6.5x faster on CPU+GPU vs. CPU+CPU**
- **Improved performance of CPU version by 100%**

Performance Goals

- Data movement between Host and Accelerator
  - minimize amount of data
  - minimize number of data moves
  - minimize frequency of data moves
  - optimize data allocation in device memory
- Parallelism on Accelerator
  - Lots of MIMD parallelism to fill the multiprocessors
  - Lots of SIMD parallelism to fill cores on a multiprocessor
  - Lots more MIMD parallelism to fill multithreading parallelism
- Multiprocessor resources

Performance Measurement

- PGI_ACC_NOTIFY
- PGI_ACC_TIME
- pgprof / nvprof
- nvvp Visual Profiler
- others (TAU, Vampir, ...)

PGI_ACC_NOTIFY Bit Mask

1 - launch

```
launch CUDA kernel  file=smooth4.c function=smooth_acc line=17 device=0 num_gangs=98
num_workers=1 vector_length=128 grid=1x98 block=128
```

2 - data upload/download

```
upload CUDA data  file=smooth4.c function=smooth_acc line=12 device=0 variable=a
bytes=40000
download CUDA data  file=smooth4.c function=smooth_acc line=23 device=0 variable=a
bytes=40000
```

4 - wait (explicit or implicit) for device

```
Implicit wait  file=smooth4.c function=smooth_acc line=17 device=0
Implicit wait  file=smooth4.c function=smooth_acc line=23 device=0
```

8 - data/compute region enter/leave

```
Enter data region file=smooth4.c function=smooth_acc line=12 device=0
Enter compute region file=smooth4.c function=smooth_acc line=14 device=0
Leave compute region file=smooth4.c function=smooth_acc line=17 device=0
```

16 - data create/allocate/delete/free

```
create CUDA data  bytes=40000 file=smooth4.c function=smooth_acc line=12 device=0
alloc  CUDA data  bytes=40000 file=smooth4.c function=smooth_acc line=12 device=0
delete CUDA data  bytes=40448 file=smooth4.c function=smooth_acc line=23 device=0
```

PGI_ACC_TIME environment variable

```
Accelerator Kernel Timing data
/proj/scratch/mwolfe/test/openacc/src/smooth4.c
  smooth_acc  NVIDIA  devicenum=0
    time(us): 317
    12: data region reached 5 times
        12: data copyin reached 10 times
             device time(us): total=121 max=19 min=11 avg=12
        23: data copyout reached 5 times
             device time(us): total=63 max=14 min=12 avg=12
    14: compute region reached 5 times
        17: kernel launched 5 times
            grid: [1x98]  block: [128]
             device time(us): total=133 max=90 min=9 avg=26
            elapsed time(us): total=176 max=99 min=17 avg=35
```

PGI_ACC_TIME environment variable

- Data collected per host thread and summed across threads
- Not valid with async; set `PGI_ACC_SYNCHRONOUS=1`

```
Accelerator Kernel Timing data
    Timing may be affected by asynchronous behavior
    set PGI_ACC_SYNCHRONOUS to 1 to disable async() clauses
/proj/scratch/mwolfe/test/openacc/src/async1.f90
  testasync  NVIDIA  devicenum=0
    time(us): 304
    19: compute region reached 1 time
        21: kernel launched 1 time
            grid: [977]  block: [256]
             device time(us): total=84 max=84 min=84 avg=84
            elapsed time(us): total=94 max=94 min=94 avg=94
```

PGPROF Command Line Profiler

- pgprof ./exe
  - Report kernel and transfer times directly
- Collect profiles for PGPROF GUI
  - %> pgprof -o profile.out ./exe
- Collect for MPI processes

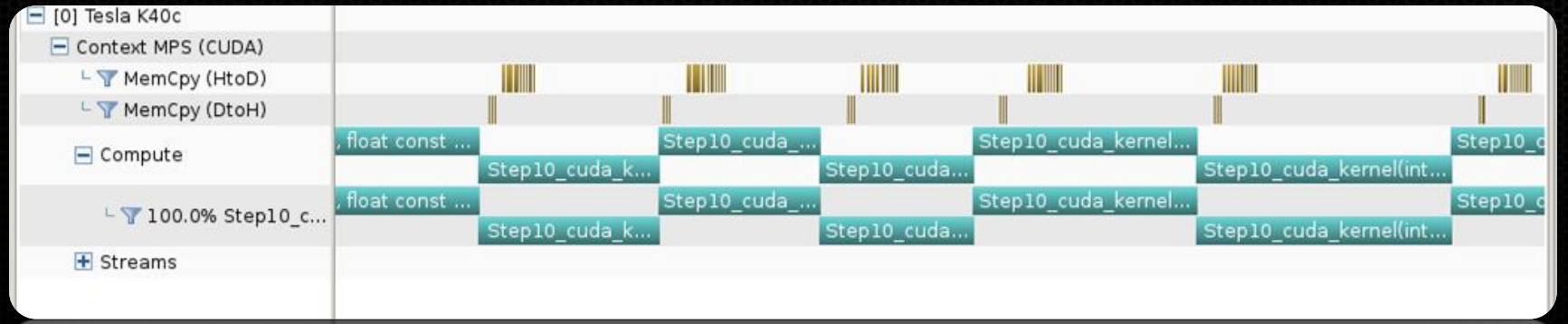
  %> mpirun –np 2 pgprof -o profile.%p.out ./exe
- Collect profiles for complex process hierarchies
  - --profile-child-processes, --profile-all-processes
- Collect key events and metrics

  %> pgprof –events all flops_sp ./exe
- Trace stream usage

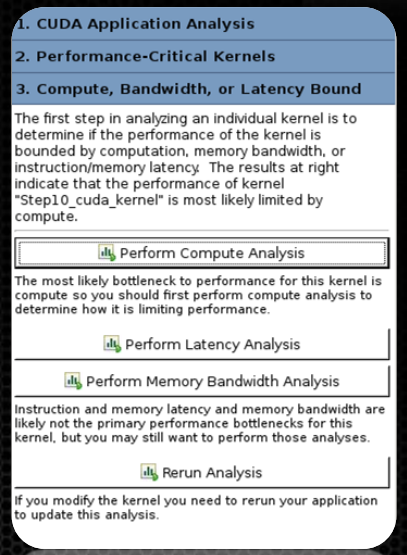
  %> pgprof –print-gpu-trace ./exe
- Full listing of options see: pgprof --help

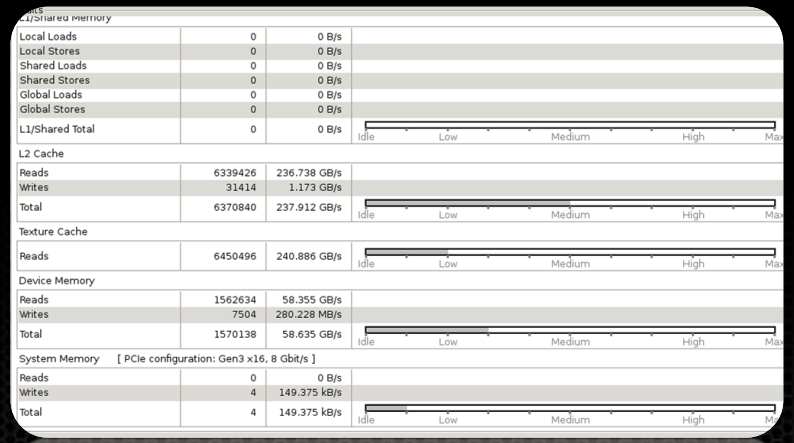
PGI's Visual Profiler

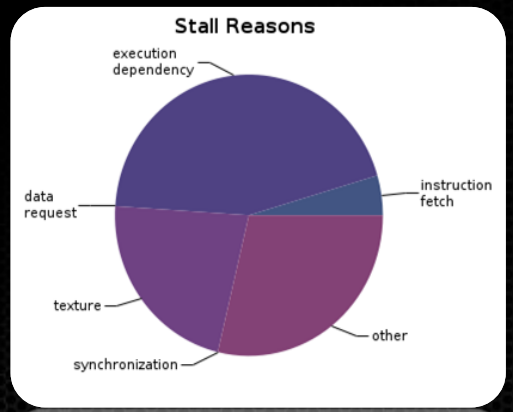
Timeline



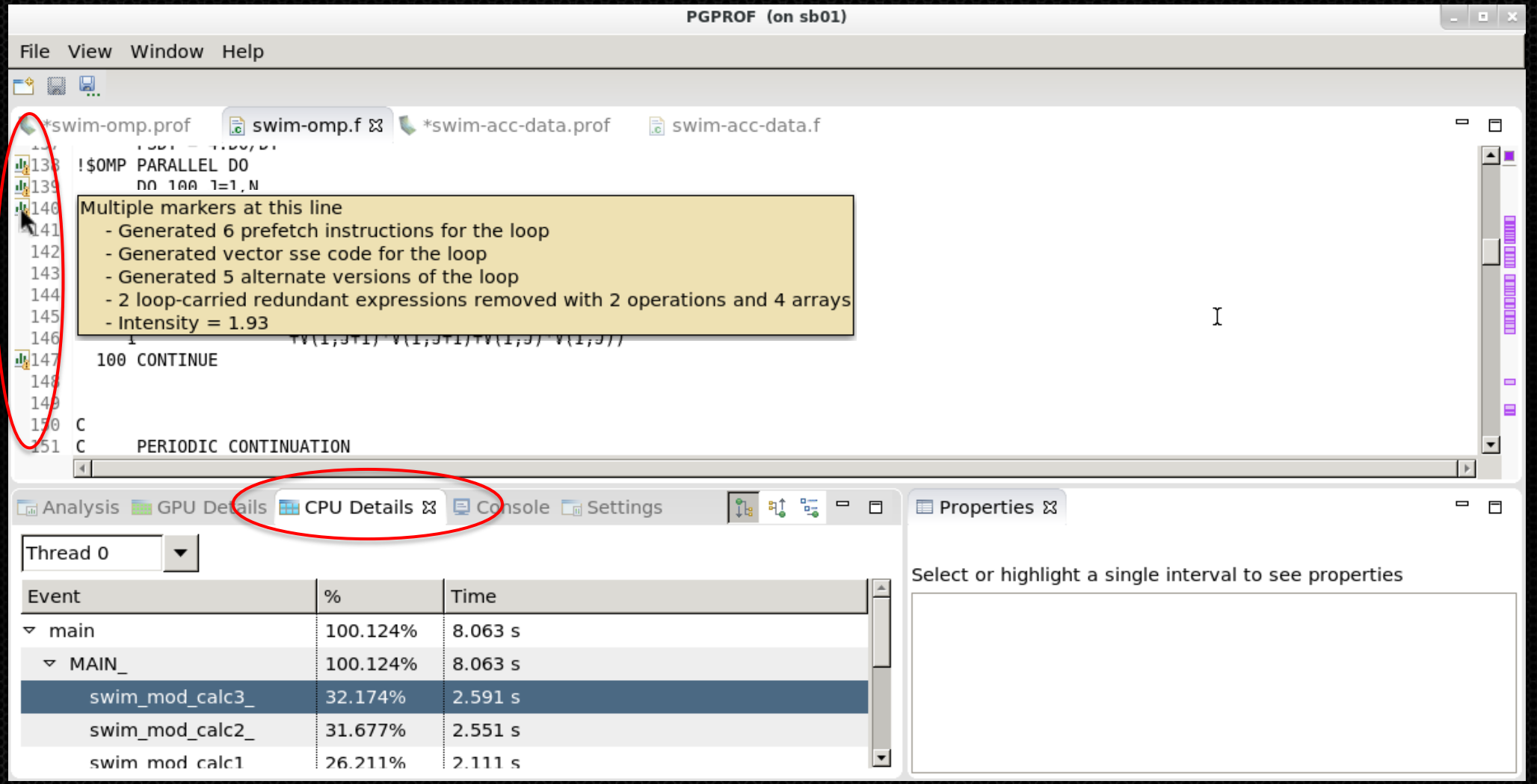
Guided Analysis System



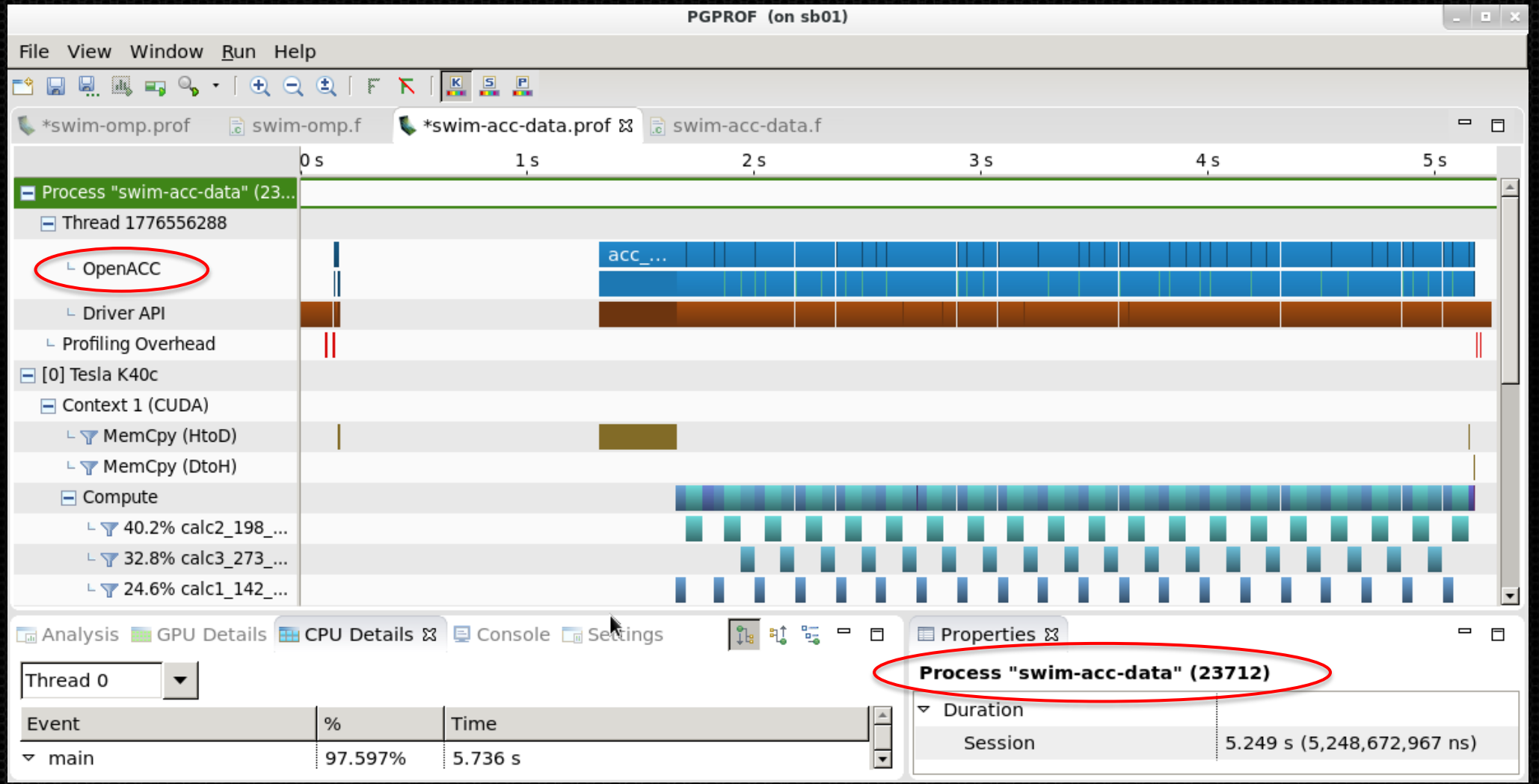




PGPROF 2016: CPU Profiling, Compiler Feedback



PGPROF 2016: OpenACC Profiling



Other New PGPROF Profiling Features

For GPU Performance Optimization

- Guided Analysis
- Utilization Analysis
- Memory Bandwidth Analysis
- Compute Analysis
- Latency Analysis
- NVLink Analysis
- http://www.pgroup.com/doc/pgprof16tut.pdf

Understanding Compiler Output

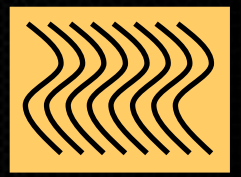
```
Accelerator kernel generated
            15, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
            17, #pragma acc loop vector(32) /* threadIdx.x */
```

- Compiler is reporting how it is assigning work to the device
  - gang is being mapped to blockIdx.x
  - worker is being mapped to threadIdx.y
  - vector is being mapped to threadIdx.x
- Unless you have used CUDA before this should make absolutely no sense to you

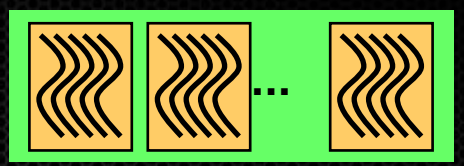
CUDA Execution Model

Software

Thread



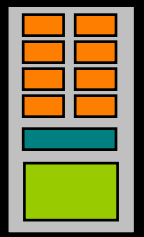
Thread Block



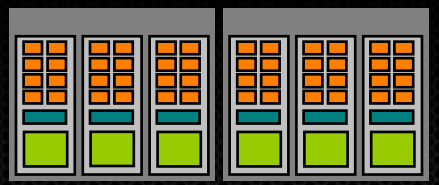
Grid

Hardware

Scalar Processor



Multiprocessor



Device

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

blocks and grids can be multi dimensional (x,y,z)

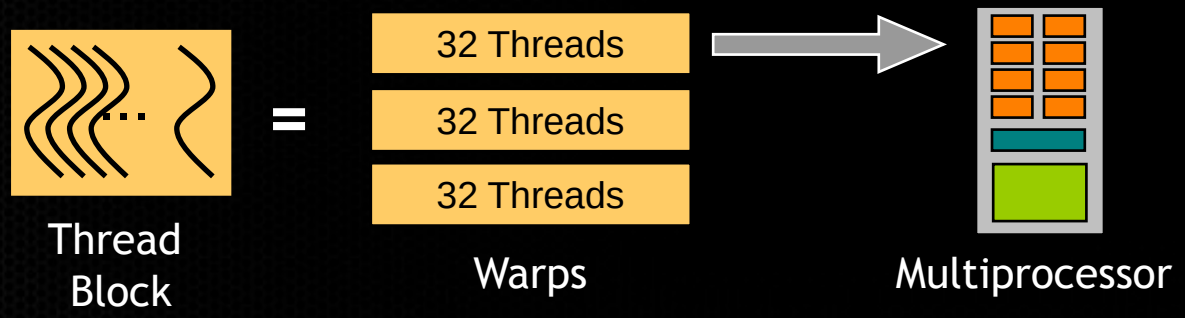
Understanding Compiler Output

```
Accelerator kernel generated
            15, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
            17, #pragma acc loop vector(32) /* threadIdx.x */
```

- Compiler is reporting how it is assigning work to the device
  - gang is being mapped to blockIdx.x
  - woker is being mapped to threadIdx.y
  - Vector is being mapped to threadIdx.x

- This application has a thread block size of 4x32 and launches as many blocks as necessary

CUDA Warps



A thread block consists of a groups of warps

A warp is executed physically in parallel (SIMD) on a multiprocessor

Currently all NVIDIA GPUs use a warp size of 32

Mapping OpenACC to CUDA

- The compiler is free to do what they want
- In general
  - gang: mapped to blocks
  - worker: mapped threads
  - vector: mapped to threads
- Mapping to blocks/threads is highly compiler dependent
- Performance Tips:
  - Use a vector size that is divisible by 32
  - Generally having the block size between 128 and 256 is ideal.
    - 128<= num_workers * vector_length <= 256

Minimize Data Transfers

- Avoid unnecessary data transfers
  - Use the most appropriate data clause (don't transfer if you don't need to)
  - Leave data on the device if possible

```
==22104== NVPROF is profiling process 22104, command: ./a.out
==22104== Profiling application: ./a.out
==22104== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.04%  3.16076s      5000  632.15us  630.45us  649.59us  [CUDA memcpy HtoD]
 36.56%  1.95739s      3000  652.46us  618.74us  672.95us  [CUDA memcpy DtoH]
  1.90%  101.98ms      1000  101.97us  79.874us  104.00us  main_24_gpu
  1.42%  75.930ms      1000  75.929us  75.170us  76.930us  main_21_gpu
  1.08%  57.828ms      1000  57.827us  57.538us  59.106us  main_18_gpu
```

Write Parallelizable Loops

Use countable loops
- C99: while->for
- Fortran: while->do

Avoid pointer arithmetic

Write rectangular loops (compiler cannot parallelize triangular lops)

```
bool found=false;
while(!found && i<N) {
  if(a[i]==val) {
    found=true
    loc=i;
  }
  i++;
}
```

```
bool found=false;
for(int i=0;i<N;i++) {
  if(a[i]==val) {
    found=true
    loc=i;
  }
}
```

```
for(int i=0;i<N;i++) {
  for(int j=i;j<N;j++) {
    sum+=A[i][j];
  }
}
```

```
for(int i=0;i<N;i++) {
  for(int j=0;j<N;j++) {
    if(j>=i)
      sum+=A[i][j];
  }
}
```

Inlining

- When possible aggressively inline functions/routines
  - This is especially important for inner loop calculations

```
#pragma acc routine seq
inline
int IDX(int row, int col, int LDA) {
  return row*LDA+col;
}
```

Kernel Fusion

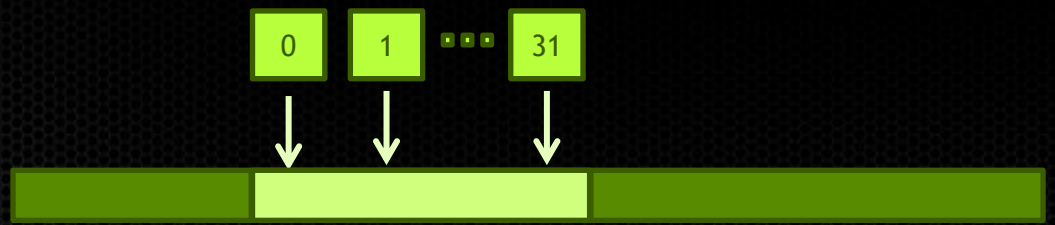
- Kernel calls are expensive
  - Each call can take over 10us in order to launch
  - It is often a good idea to generate large kernels is possible
- Kernel Fusion (i.e. Loop fusion)
  - Join nearby kernels into a single kernel

```
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    a[i]=0;
}
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    b[i]=0;
}
```

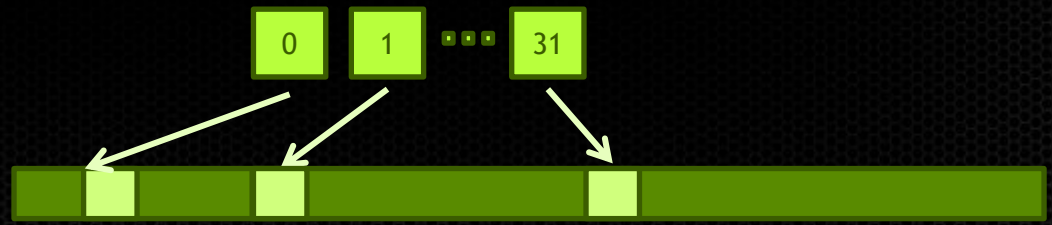
```
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {
    a[i]=0;
    b[i]=0;
}
```

Memory Coalescing

- *Coalesced* access:
  - A group of 32 contiguous threads (“warp”) accessing adjacent words
  - Few transactions and high utilization
- *Uncoalesced* access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization
- For best performance threadIdx.x should access contiguously



Coalesced



Uncoalesced

Loop Bounds and Array Access Patterns

```
do iz = zmin, zmax
  do iy = ymin, ymax
    do ix = xmin, xmax
       s3 = 0.0_8
       do j = 1, nx  ! 4 <= nx <= 8
         s1 = 0.0_8; s2 = 0.0_8
         do i = 1, nx
            s1 = s1 + delta(i,j,ix)*flx(i,j,ix,iy,iz)
            s2 = s2 + delta(j,i,ix)*fly(i,j,ix,iy,iz)
         end do  ! i loop
         s3 = s3 + s1*wgtx(j) + s2*wgty(j)
      end do ! j loop
      grad(ix,iy,iz) = s3
   end do ! ix loop
  end do ! iy loop
end do ! iz loop
```

OpenACC async and wait clauses

async(n): launches work asynchronously in queue *n*

wait(n): blocks host until all operations in queue *n* have completed

Can significantly reduce launch latency, enables pipelining and concurrent operations

```
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc wait(1)
```

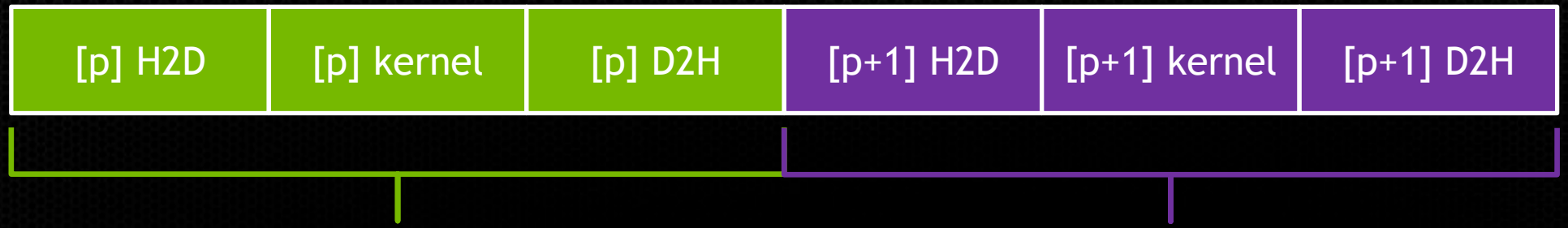
OpenACC Pipelining

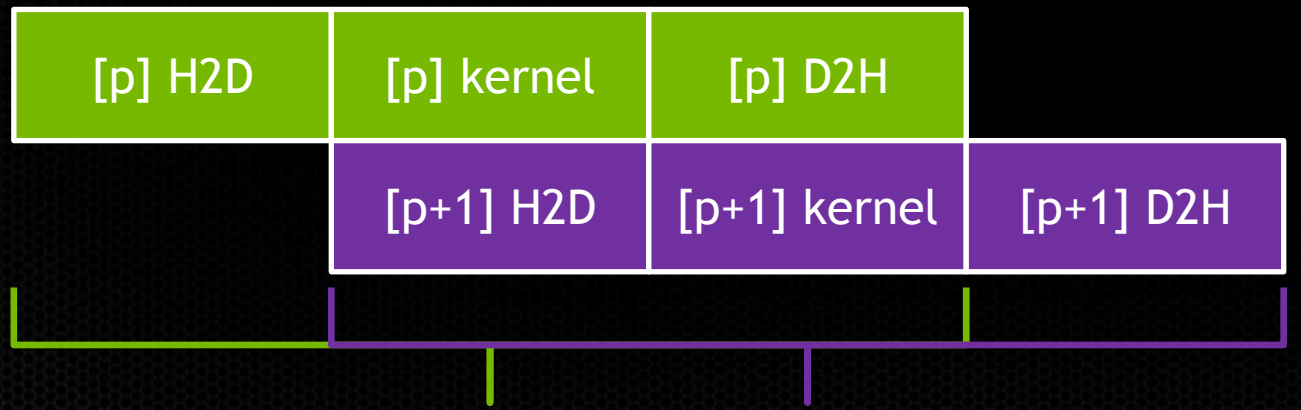
```
#pragma acc data
for(int p = 0; p < nplanes; p++)
{
  #pragma acc update device(plane[p])
  #pragma acc parallel loop
  for (int i = 0; i < nwork; i++)
  {
    // Do work on plane[p]
  }
  #pragma acc update host(plane[p])
}
```

For this example, assume that each “plane” is completely independent and must be copied to/from the device.

As it is currently written, plane[p+1] will not begin copying to the GPU until plane[p] is copied from the GPU.

OpenACC Pipelining (cont.)





NOTE: In real applications, your boxes will not be so evenly sized.

OpenACC Pipelining (cont.)

```
#pragma acc data
for(int p = 0; p < nplanes; p++)
{
  #pragma acc update device(plane[p]) async(p)
  #pragma acc parallel loop async(p)
  for (int i = 0; i < nwork; i++)
  {
    // Do work on plane[p]
  }
  #pragma acc update host(plane[p]) async(p)
}
#pragma acc wait
```

Enqueue each plane in a queue to execute in order

Wait on all queues.

Async Additions

```
#pragma acc parallel async(5) wait(4,3)
{
   #pragma acc loop collapse(2) gang vector
   for( j = 1; j < n-1; ++j )
      for( i = 1; i < m-1; ++i )
         x[i][j] = 0.25*( y[i-1][j] + y[i+1][j] +
                          y[i][j-1] + y[i][j+1]);

}
// this parallel region goes on queue 5
// this parallel region waits for existing events on queues 4,3
```

Async Additions

```
#pragma acc wait(3)
// this thread waits for queue 3 to complete

#pragma acc wait(3) async(5)
// queue 5 waits for all events on queue 3 to complete
```

OpenACC cache directive

cache: on NVIDIA GPUs, this is meant to read data from global memory and store it in shared memory, for multiple uses, by multiple threads.

Thus far, in our PGI implementation, we've seen many slowdowns, and some speedups.

Stay tuned.

Cache Directive Example

```
!$acc parallel loop gang collapse(3) private(grads,dinv_s,vtemp,dvv_s,spheremp_s)
present(dinv,tensorvisc,spheremp,variable_hyperviscosity,deriv_dvv,s,laplace)
vector_length(kchunk*np*np)
    do ie = 1 , nelemd
      do q = 1 , qsize
        do ks = 1 , nlev, kchunk
          !$acc cache(grads,dinv_s,vtemp,dvv_s,spheremp_s)

          !$acc loop vector collapse(3)
          do k = 1 , kchunk
            do j = 1 , np
              do i = 1 , np
                if (k == 1) then
                  dvv_s(i,j) = deriv_dvv(i,j)
                  . . .
```

OpenACC `routine` directive

routine: Compile the following function for the device (allows a function call in device code)

Clauses: gang, worker, vector, seq

```
#pragma acc routine seq
void fun(…) {

  for(int i=0;i<N;i++)
    ...
}
```

```
#pragma acc routine vector
void fun(…) {
  #pragma acc loop vector
  for(int i=0;i<N;i++)
    ...
}
```

The OpenACC Routine Directive

- **routine:** Compile the following function for the device (allows a function call in device code) Clauses: gang, worker, vector, seq

```
#pragma acc parallel loop gang \
            vector_length(VL)
   for(int i=0;i<N;i++)
    fun_vec(…);
}
```

```
#pragma acc routine vector
void fun_vec(…) {
  #pragma acc loop vector
  for(int i=0;i<N;i++)
    fun_seq(…);
}
```

```
#pragma acc routine seq
void fun_seq(…) {

  for(int i=0;i<N;i++)
    ...
}
```

OpenACC routine: Fortran

```
subroutine foo(v, i, n) {
  use …
  !$acc routine vector
  real :: v(:,:)
  integer, value :: i, n
  !$acc loop vector
  do j=1,n
    v(i,j) = 1.0/(i*j)
  enddo
end subroutine

!$acc parallel loop
do i=1,n
  call foo(v,i,n)
enddo
!$acc end parallel loop
```

The routine directive may appear in a fortran function or subroutine definition, or in an interface block.

Nested acc routines require the routine directive within each nested routine.

The save attribute is not supported.

Note: Fortran, by default, passes all arguments by reference. Passing scalars by value will improve performance of GPU code.

Routine Directive Example

```
      subroutine transprt ()
!$acc routine(photprod) vector
!$acc routine(chemrate) vector
!$acc routine(chempl) vector
. . .
!$acc routine(vsisolv) vector
!$acc routine(densolv2) vector
!$acc routine(smoothz) vector
. . .
!$acc parallel
!$acc loop gang
      do nfl = 1, nf
!$acc loop seq
        do ni = nion1,nion2
!         update variables
          call densolv2 ( ni,denin(1,ni,nfl),
     .       prod(1,ni,nfl), loss(1,ni,nfl), deni_old(1,ni,nfl), nfl )

        end do
      end do
!$acc end parallel
```

Review

- Use the reduction clause to parallelize reductions
- Use routine to parallelize subroutines
- Compiler output explicitly tells you what it is doing
  - Watch out for implicit parallelization, it may not be portable
    - e.g. reduction, routine, etc
- Use collapse or gang, worker, and vector to parallelize nested loops

OpenACC `atomic` directive

atomic: subsequent block of code is performed atomically with respect to other threads on the accelerator

Clauses: read, write, update, capture

```
#pragma acc parallel loop
for(int i=0; i<N; i++) {
  #pragma acc atomic update
  a[i%100]++;
}
```

Atomic Operations

```
#pragma acc parallel loop
   for( j = 1; j < n-1; ++j ){
        y = x[j];
        i = y & 0xf;
        #pragma acc atomic update
            ++bin[i];
   }
// essentially the OpenMP atomic operations
// atomic update, read, write, capture
// only native data lengths supported
```

OpenACC Atomic Operations

miniMD contains a race condition

```
#pragma acc data copyout(f[0:3*nall]) copyin(x[0:3*nall],
numneigh[0:nlocal], neighbors[0:nlocal*maxneighs])
  {
  // clear force on own and ghost atoms
    #pragma acc kernels loop
    for(int i = 0; i < nall; i++) {
……
    }
    #pragma acc kernels loop independent
    {
    for(int i = 0; i < nlocal; i++) {
……
      for(int k = 0; k < numneighs; k++) {
        j = neighs[k];
……
            if(GHOST_NEWTON || j < nlocal) {
              f[3 * j + 0] -= delx * force;
              f[3 * j + 1] -= dely * force;
              f[3 * j + 2] -= delz * force;
            }
```

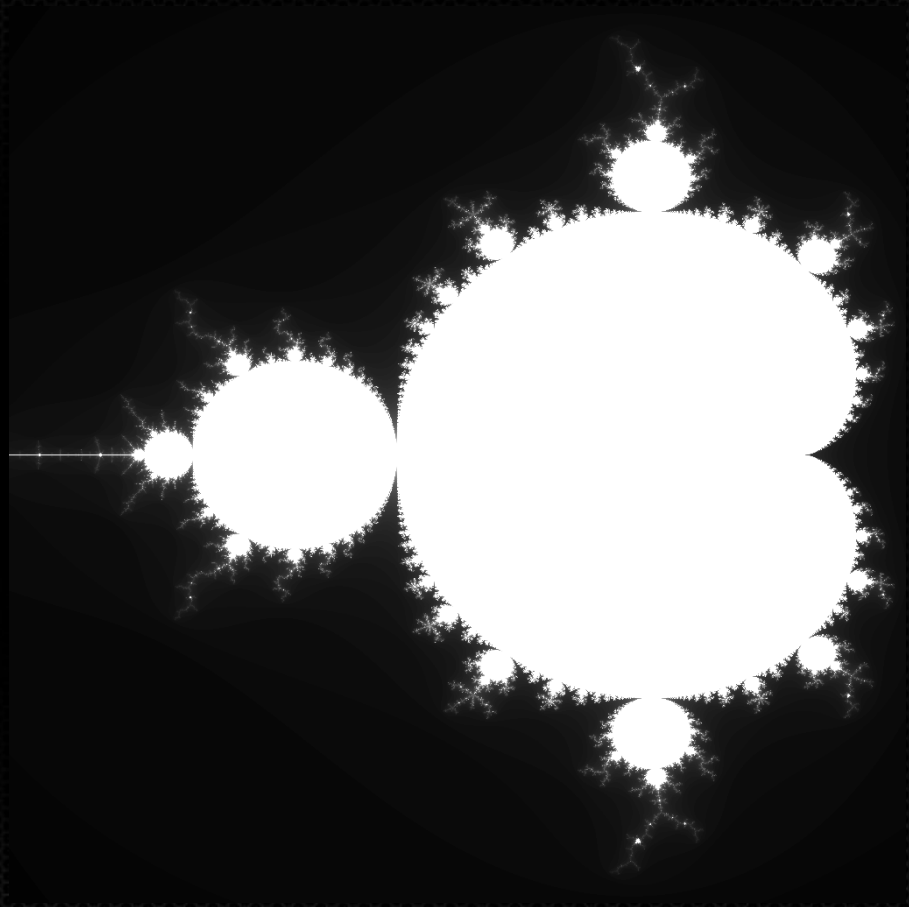
Fails – requires total re-write without support for OpenACC 2.0 atomic directives.

With OpenACC atomic

```
#pragma acc data copyout(f[0:3*nall]) copyin(x[0:3*nall],
numneigh[0:nlocal], neighbors[0:nlocal*maxneighs])
  {
  // clear force on own and ghost atoms
    #pragma acc kernels loop
    for(int i = 0; i < nall; i++) {
……
    }
    #pragma acc kernels loop independent
    {
    for(int i = 0; i < nlocal; i++) {
……
      for(int k = 0; k < numneighs; k++) {
        j = neighs[k];
……
            if(GHOST_NEWTON || j < nlocal) {
          #pragma acc atomic update
              f[3 * j + 0] -= delx * force;
          #pragma acc atomic update
              f[3 * j + 1] -= dely * force;
          #pragma acc atomic update
              f[3 * j + 2] -= delz * force;
            }
```

Hands On Activity (Example 3)

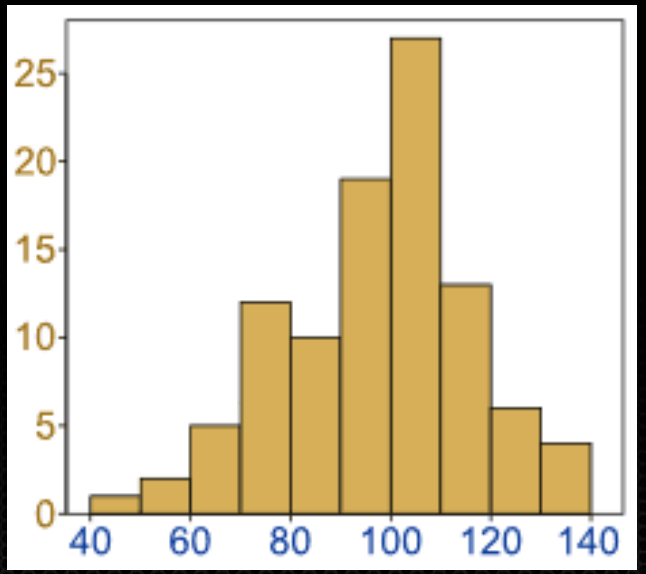
1. Accelerate the Mandelbrot code
2. Validate results using gthumb



Hands On Activity (Exercise 4)

Exercise 4: Simple histogram creation

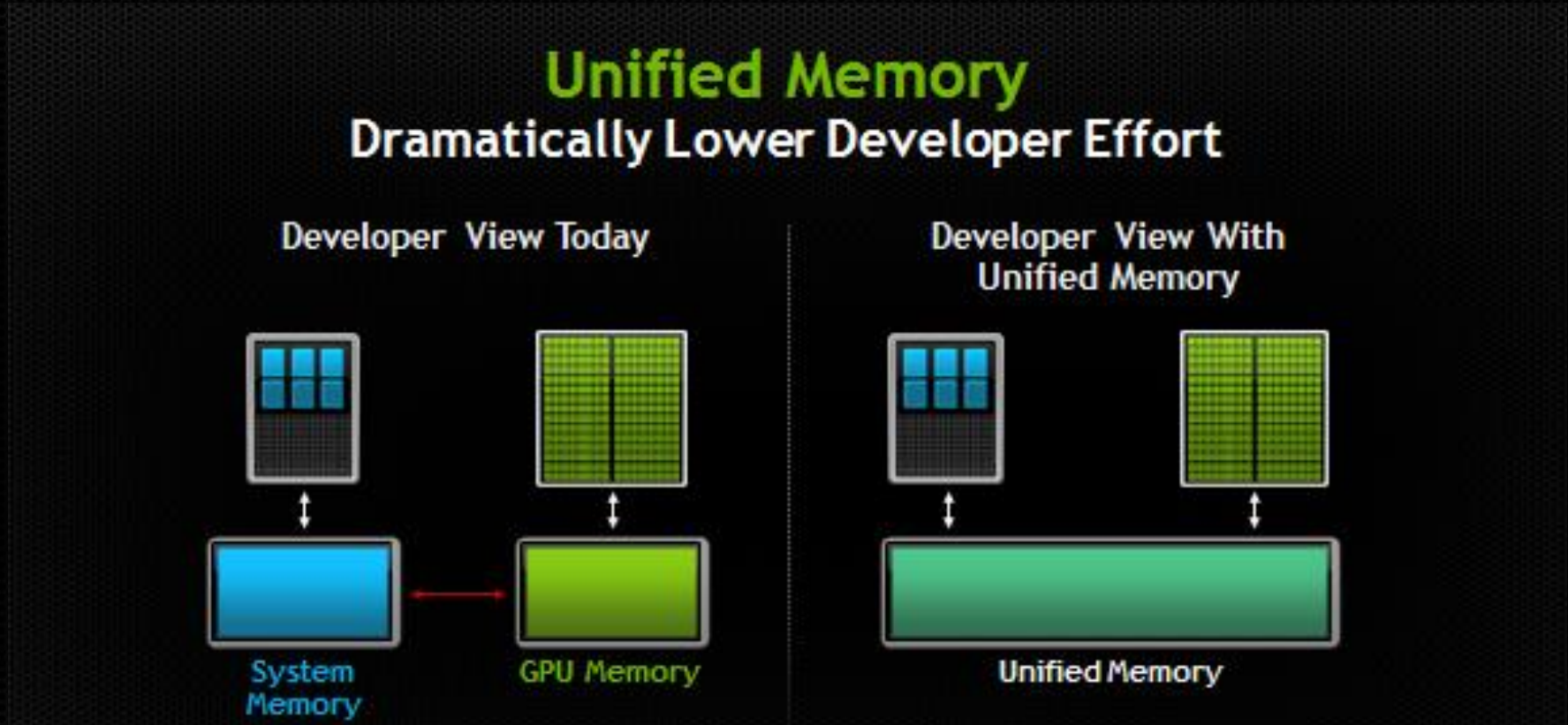
1. Use what you have learned to accelerate this code



Review

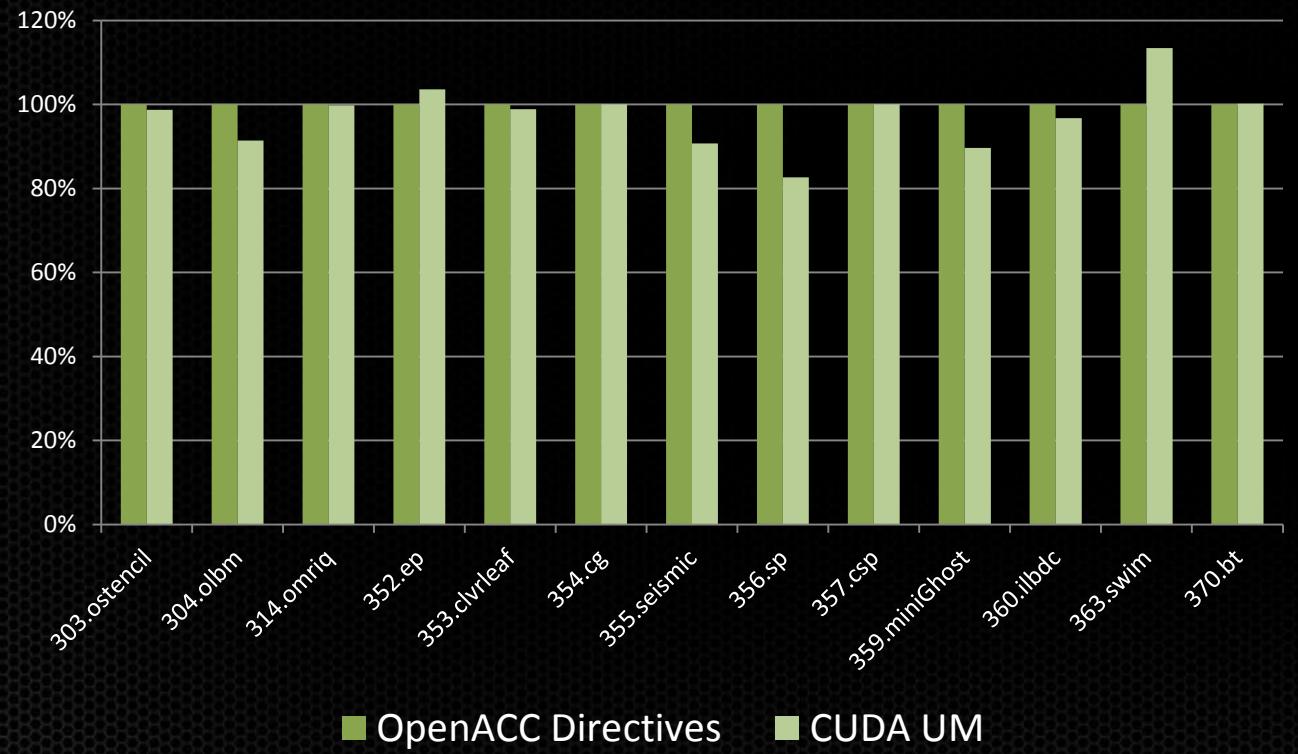
- Minimize data transfers
- Avoid loops structures that are not parallelizable
  - While loop & triangular loops
- Inline function calls within kernels when possible
- Fuse nearby kernels to minimize launch latency
- Optimize memory access pattern to achieve coalesced access
  - threadIdx.x should be the contiguous dimension
- Use async and wait to reduce launch latency and enable pipelining

NVIDIA Unified Memory



OpenACC and CUDA Unified Memory

PGI 15.1: OpenACC directive-based data movement vs OpenACC w/CUDA 6.5 Unified Memory on Kepler



Features:

- Fortran ALLOCATE and C/C++ malloc/calloc/new can automatically use CUDA Unified Memory
- No explicit transfers needed for dynamic data (or allowed, for now)

Limitations:

- Supported only for dynamic data
- Program dynamic memory size is limited by UM data size
- UM data motion is synchronous
- Can be unsafe

OpenACC w/Unified Memory

- Advantages
  - No data clauses needed for dynamic data
  - Simpler on-ramp to performance
  - Ready for future architectures
  - Avoid deep copy problem
- Limitations
  - Applies only to heap allocations. Not available yet for stack, static, or global data
  - Limited to available device memory
  - Cannot access unified memory in host code when a kernel is running. If you do, likely you will see a bus error
  - All data movement is synchronous
- https://www.pgroup.com/lit/articles/insider/v6n2a4.htm

OpenACC Technical Report TR-14-1

- Deep Copy: What if we need stl::vector or other container template classes where the data type is not known?
- The OpenACC 2.0 Specification considerably improved C++ support with the addition of “routine” and unstructed data regions, but falls short on managing complex data management.
- It’s a very difficult problem to solve with a general solution.
- The OpenACC committee is investigating solutions and has published two reports
  - http://www.openacc.org/sites/default/files/TR-14-1.pdf
  - http://www.openacc.org/sites/default/files/TR-16.1.pdf

The Deep Copy Problem in Fortran

```
    type county_t
        character*32 :: cname
        integer(4)   :: countyid
        real(4)      :: Latitude, Longitude
        integer(4), allocatable :: population(:)
    end type
    type state_t
        character*2  :: code
        integer(4)   :: stateid
        character*32 :: cname
        type(county_t), allocatable :: county(:)
        integer(4), allocatable :: county_rlkup(:)
    end type
    type country_t
        character*32 :: cname
        type(state_t), allocatable :: state(:)
        integer(4), allocatable :: state_rlkup(:)
    end type
    type(country_t) :: homeland
end module
```

A Manual Deep Copy Solution in Fortran

```
subroutine copy_deep_in
use geography

!$acc enter data copyin(homeland, homeland%state, homeland%state_rlkup)
do i = 1, size(homeland%state)
  !$acc enter data copyin(homeland%state(i)%county)
  do j = 1, size(homeland%state(i)%county)
    !$acc enter data copyin(homeland%state(i)%county(j)%population)
  enddo
  !$acc enter data copyin(homeland%state(i)%county_rlkup)
enddo
return
end subroutine
```

Using the deep data in Fortran

```
!$acc parallel present(homeland)
!$acc loop gang
do k = 1, size(homeland%state)
  !$acc loop vector
  do j = 1, size(homeland%state(k)%county)
    !$acc loop seq
    do i = 1, pop_future_years
      ipop = pop_data_years+i
      homeland%state(k)%county(j)%population(ipop) = &
      . . .
    end do
  end do
end do
!$acc end parallel
```

PGI C++ with OpenACC

- Initially, OpenACC could only be used in C++ if it looked like C
- As of the PGI 14.4 release support was added for
  - C++ Classes, “this” pointers, Data Members, Class Methods
  - Template classes
- This was an important first step. PGI is collaborating with customers such as Sandia National Labs on improving support for C++ with OpenACC.

Use pgc++

PGI pgc++ features

- GNU compatibility
- Uses GNU STL and headers
- Supported on Linux and OSX
- C++11 Support. Some C++14
- On-going performance tuning

Compiler Options

```
% pgc++ --c++11 -acc -Minfo ex1.cpp

will produce the following compiler feedback (CCFF) messages:

main:
  11, Generating copyout(A[:][:])
      Generating copy(sum)
      Generating Tesla code
  13, Loop is parallelizable
  14, Loop is parallelizable
      Accelerator kernel generated
  13, #pragma acc loop gang, vector(128) collapse(2)
  14, /* blockIdx.x threadIdx.x collapsed*/
      main::[lambda(int, int) (instance 1)] ::operator  ()(int, int)
const:
  10, Generating implicit acc routine seq
      Generating Tesla code
```

Unstructured Data Regions

```
template<typename T> class vtype {
    long _size;
    T* _data;
public:
    explicit vtype(long size) : _size(size) {
        _data = new T[_size];
          // Copy the ‘this’ pointer and shallow copy of data members
        #pragma acc enter data copyin(this)
          // Create the _data vector on device and ‘attach’ to the class
        #pragma acc enter data create(_data[0:_size])
    }
    ~vtype() {
        delete [] _data;
          // Delete the device data
        #pragma acc exit data delete(_data,this)
    }
```

Seq Routine Auto-Generation

- In a C++ program, many functions, certainly class member functions, appear as source code in header files included in the program. Oftentimes, the functions are defined in system headers or other application packages, and modifying those headers is either unwise or impossible.

- The PGI C++ compiler will take note of functions called in compute regions and implicitly add the pragma *acc routine seq* if there is no explicit routine directive.

PGI's acc_attach() Extension

- In our experiences, we've found that we need to expose the attach operation to the user.
- PGI has implemented a new API function, acc_attach

```
// copy constructor
dupvector( const dupvector &copyof ) {
   size = copyof.size;
   data = copyof.data;
   #pragma acc enter data copyin(this[0:1])
   acc_attach( (void**)&data );
}
```

Hands On Activity (Example 1)

1. Modify the Makefile to build with OpenACC Unified Memory Support

   `-ta=tesla:managed` Target NVIDIA GPUS

2. Remove the data regions
3. Run again:

   ```
   %> time ./a.out
   ```

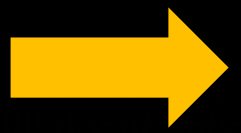
   Do you understand the performance?

Hands On Activity (Example 3)

1. Pipeline the Mandelbrot code by batching rows
    1. What was the time for compute + copy before & after?

```
#pragma acc ...
for rows
  for cols
      …
//copy image to host

fwrite(...);
```

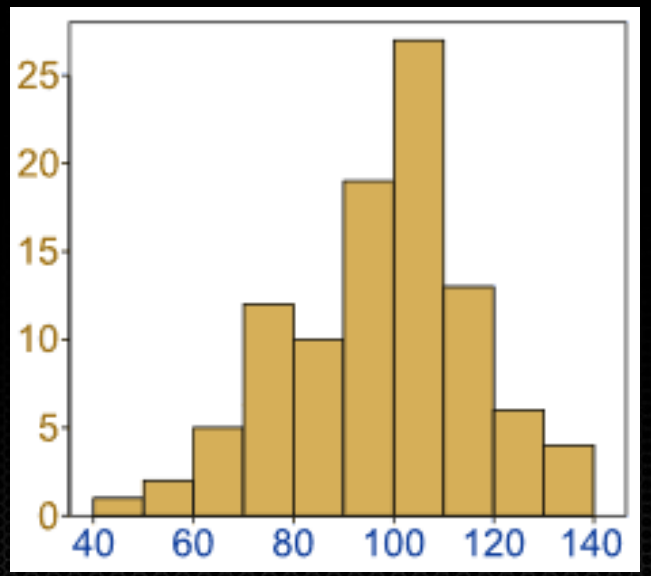


```
for batches
  #pragma acc ... async(...)
  for rows in batch
    for cols
      ...
  //copy batch to host async
  //wait for execution
  fwrite(...)
```

Hands On Activity (Exercise 4)

Exercise 4: Simple histogram creation

1. Use the curand library to fill the array on the device



Challenge Problem: CG Solver

- Files:
  - main.cpp: the high level cg solve algorithm
  - matrix.h: matrix definition and allocation routines
  - vector.h: vector definition and allocation routines
  - matrix_functions.h: the matrix kernels
  - vector_functions.h: the vector kernels

1. Accelerate this application to the best of your ability

CUDA Fortran / OpenACC Interoperability

- Call CUDA Fortran kernels from OpenACC data regions
- Use CUDA Fortran device and managed data in compute regions

```
module mymod
  real, dimension(:), allocatable, device :: xDev
end module
...
use mymod
...a
allocate( xDev(n) ) ! allocates xDev in GPU memory
call init_kernel <<<dim3(n/128),dim3(128)>>> (xDev, n)
...
!$acc data copy( y(:) )          ! no need to copy xDev
  ...
!$acc kernels loop
  do i = 1, n
    y(i) = y(i) + a*xDev(i)
  enddo
  ...
!$acc end data
```

- Support of arguments with the device attribute in the PGI OpenACC 2.0 Runtime Library Routines

CUDA / OpenACC Interoperability

- The acc host_data use_device construct is useful for calling CUDA functions that take device pointers.
- Force CUDA libraries to use the default OpenACC stream by the sequence:
  - C:
    - cu***SetStream(handle, (cudaStream_t) acc_get_cuda_stream(acc_async_sync));
  - Fortran:
    - istat = cu***SetStream(handle, acc_get_cuda_stream(acc_async_sync))
- See the example codes in 2016/examples/CUDA-Libraries

Runtime Library Routines

Fortran

```
use openacc
#include "openacc_lib.h"
```

```
acc_get_num_devices
acc_set_device_type
acc_get_device_type
acc_set_device_num
acc_get_device_num
acc_async_test
acc_async_test_all
```

C

```
#include "openacc.h"
```

```
acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```

New OpenACC 2.0 API calls

```
acc_copyin( x, sizeof(int)*n );
call acc_copyin( x(1:n,1:m) )
acc_present_or_copyin( x, sizeof(int)*n );
call acc_present_or_copyin( x(1:n,1:m) )
acc_create( x, sizeof(int)*n );
call acc_create( x(1:n,1:m) )
acc_present_or_create( x, sizeof(int)*n )
call acc_present_or_create( x(1:n,1:m) )
acc_delete( x, sizeof(int)*n );
call acc_delete ( x(1:n,1:m) )
acc_copyout( x, sizeof(int)*n );
call acc_copyout( x(1:n,1:m) )
```

New OpenACC 2.0 API calls

```
acc_update_device( x, sizeof(int)*n )
call acc_update_device( x(1:n,1:m) )
acc_update_self( x, sizeof(int)*n )
call acc_update_self( x(1:n,1:m) )
acc_map_data( devptr, hostptr, sizeof(float)*n )
acc_unmap_data( hostptr )
devptr = acc_deviceptr( hostptr )
hostptr = acc_hostptr( devptr )
acc_is_present( x, sizeof(float)*n )
acc_is_present( a(1:n,1:m) )
acc_memcpy_to_device( devptr, src, bytes )
acc_memcpy_from_device( dest, devptr, bytes )
```

Platform-Specific API calls

```
acc_get_current_cuda_device()
acc_get_current_cuda_context()
acc_get_cuda_stream( i )
acc_set_cuda_stream( i, void* )
```

The PGI OpenACC conditional sentinel

```
!@acc call acc_set_device_num(n)

is equivalent to

#ifdef _OPENACC
call acc_set_device_num(n)
#endif

Also, !@cuf for CUDA Fortran
```

MPI Parallelization Strategies

- One MPI process per GPU
  - Multi-GPU: use acc_set_device_num to control GPU selection per rank
- Multiple MPI processes per GPU
  - Use NVIDIA's Multi-Process Service (MPS)
  - Documentation: man nvidia-cuda-mps-control
  - Currently only supports a single GPU per node (multi-GPU POR in 7.0)

Review

- OpenACC is open, simple, and portable
- Assess, Parallelize, Optimize, Deploy
  - Assess: Find limiters
  - Parallelize & Optimize: Target limiters
  - Deploy: Get changes out the door
- Fine grained parallelism is key
  - Expose parallelism where ever it may be