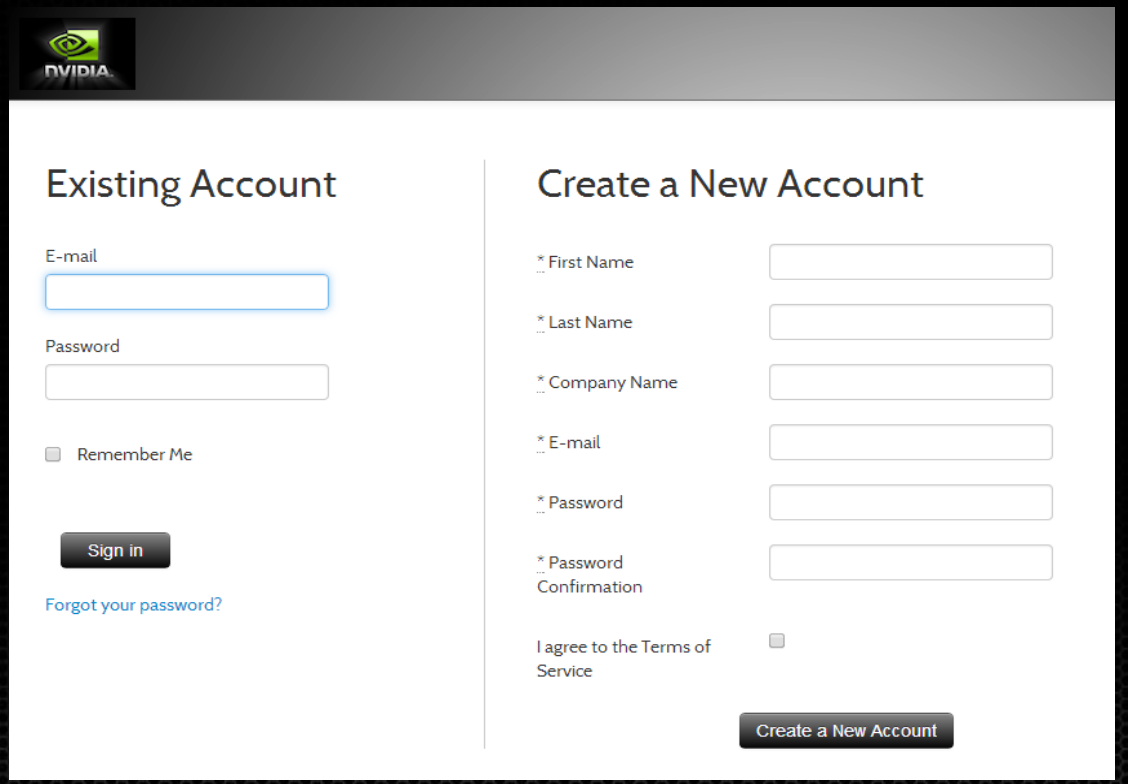
NVIDIA
OPENACC
Hands on Workshop

Before we begin…

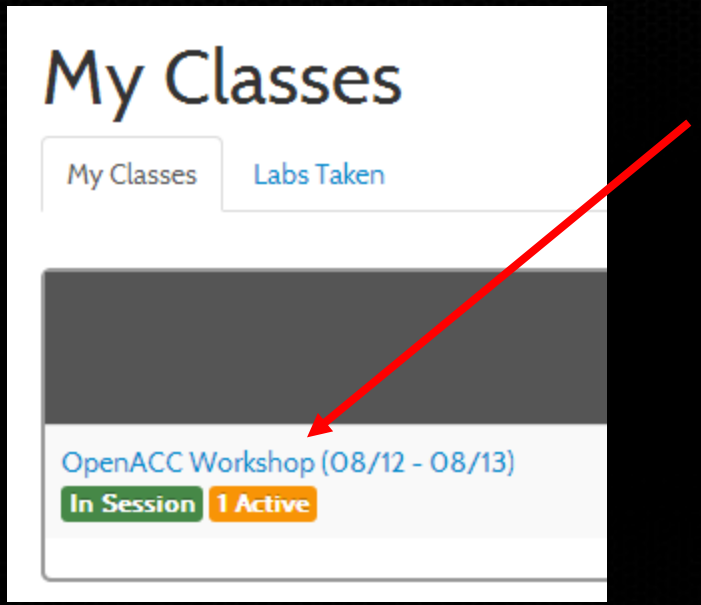
- Let’s get the AWS instance started

Getting access

- Goto nvlabs.qwiklab.com, log-in or create an account



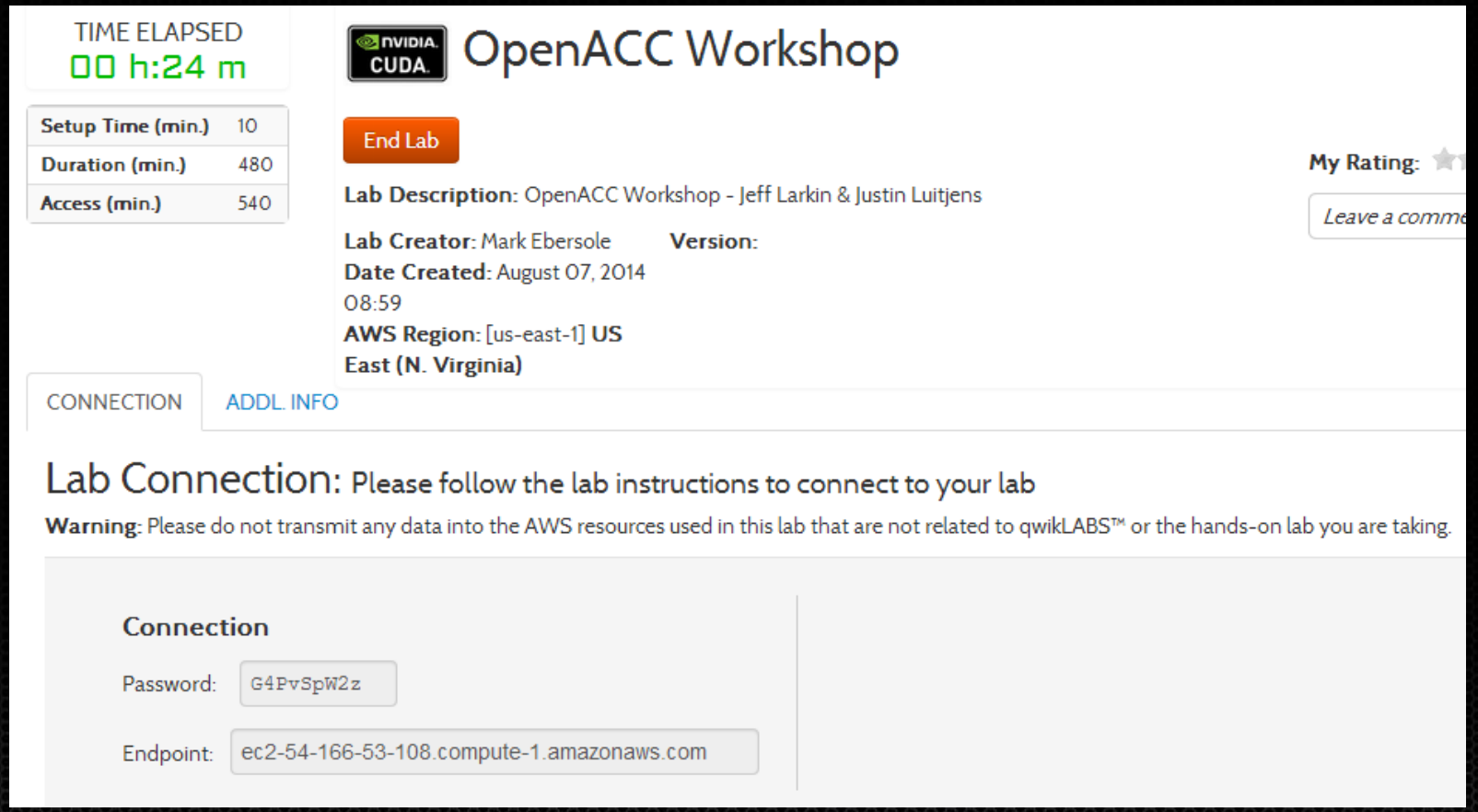
Select Openacc workshop link



Find lab and click start

Connection information

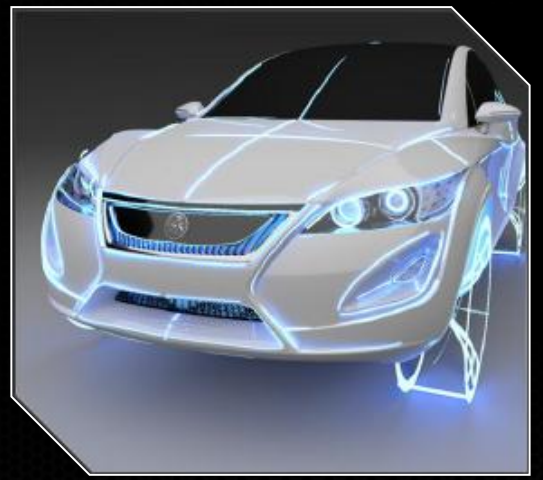
- After about a minute, you should see



World Leader in Visual Computing

GAMING

PRO VISUALIZATION



HPC & BIG DATA

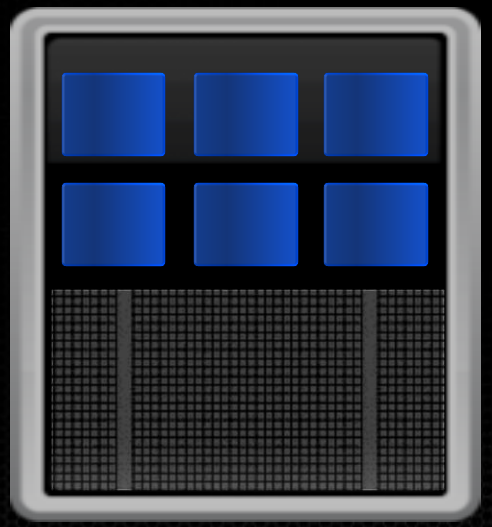
MOBILE COMPUTING

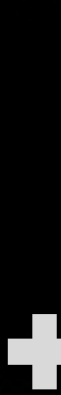
HPC's Biggest Challenge: Power

Accelerated Computing

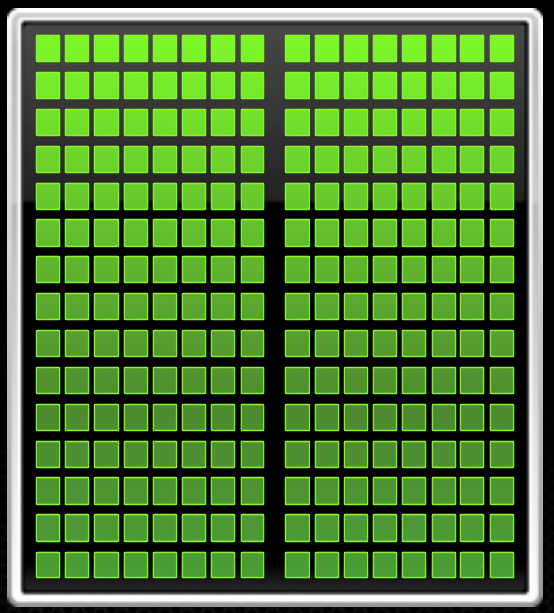
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for
Serial Tasks



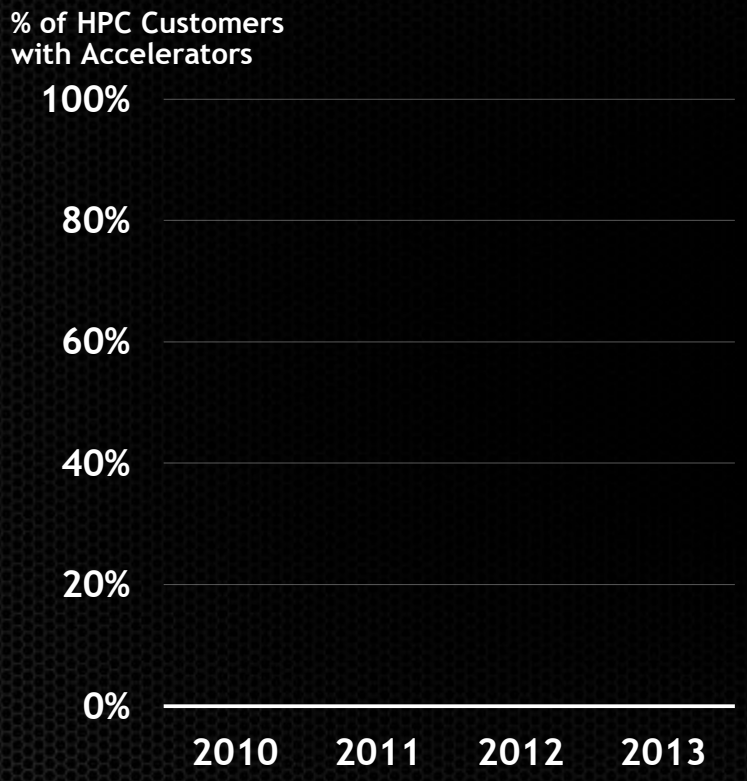


GPU Accelerator
Optimized for
Parallel Tasks



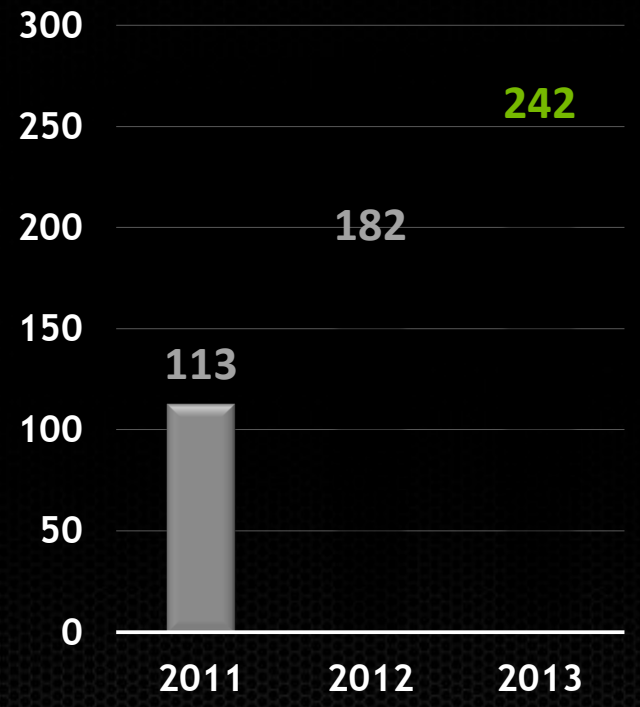
Accelerated Computing Growing Fast

Rapid Adoption of Accelerators

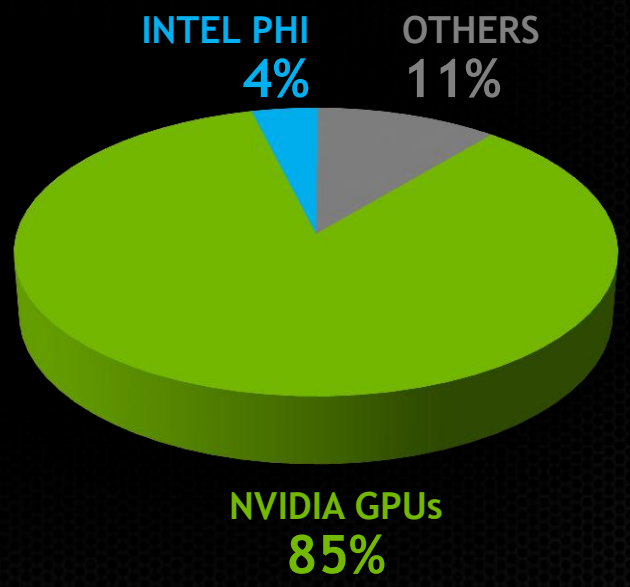


Intersect360 HPC User Site Census: Systems, July 2013
IDC HPC End-User MSC Study, 2013

Hundreds of GPU Accelerated Apps



NVIDIA GPU is Accelerator of Choice



Intersect360 Research
HPC User Site Census: Systems, July 2013

Data Clauses

`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to the host when exiting region.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

C99

```
#pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])
```

Fortran

```
!$acc data copyin(a(1:end)), copyout(b(s/4:s/4+3*s/4))
```

C99: var[first:count]
Fortran: var(first:last)

Hands On Activity (Example 1)

1. Modify the code to add a structured data region at the appropriate spot
   - How does the compiler output change?
2. Retime the code
   - Is it faster now?
3. Reprofile the code using NVVP
   - What is the distribution of transfers vs kernels now?
   - How far apart are consecutive kernels?

```
#pragma acc data copy(...)
{
    ...
}
```

OpenACC enter exit Directives

enter: Defines the start of an unstructured data region

clauses: `copyin(list), create(list)`

exit: Defines the end of an unstructured data region

clauses: `copyout(list), delete(list)`

- Used to define data regions when scoping doesn't allow the use of normal data regions (e.g. The constructor/destructor of a class).

```
#pragma acc enter data copyin(a)
...
#pragma acc exit data delete(a)
```

OpenACC enter exit Directives

Every variable in enter should also appear at exit

exit must appear before deallocation

Order is important
#pragma acc data enter (Error)

Data is not reference counted
(first exit will delete data)

Hands On Activity (Example 1)

1. Now use enter/exit data instead of a structured data region

```
#pragma acc enter data copyin(a)
...
#pragma acc exit data delete(a)
```

OpenACC update Directive

update: Explicitly transfers data between the host and the device

Useful when you want to update data in the middle of a data region

Clauses:

device: copies from the host to the device

self,host: copies data from the device to the host

```
#pragma acc update host(x[0:count])
MPI_Send(x,count,datatype,dest,tag,comm);
```

OpenACC `kernels` construct

The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```
#pragma acc kernels
{
  for(int i=0; i<N; i++)
  {
    a[i] = 0.0;
    b[i] = 1.0;
    c[i] = 2.0;
  }

  for(int i=0; i<N; i++)
  {
    a[i] = b[i] + c[i];
  }
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

OpenACC `parallel loop` vs. `kernels`

`PARALLEL LOOP`

- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP

`KERNELS`

- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with single directive
- Gives compiler additional leeway.

Both approaches are equally valid and can perform equally well.

Hands on Activity (Example 1)

1. Modify the code to the use kernels directive instead of parallel loop
   - Did it work?

```
#pragma acc kernels
{
  ...
}
```

Aliasing Rules Prevent Parallelization

```
23, Loop is parallelizable
        Accelerator kernel generated
        23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    25, Complex loop carried dependence of 'b->' prevents parallelization
        Loop carried dependence of 'a->' prevents parallelization
        Loop carried backward dependence of 'a->' prevents vectorization
        Accelerator scalar kernel generated
    27, Complex loop carried dependence of 'a->' prevents parallelization
        Loop carried dependence of 'b->' prevents parallelization
        Loop carried backward dependence of 'b->' prevents vectorization
        Accelerator scalar kernel generated
```

OpenACC independent clause

Specifies that loop iterations are data independent. This overrides any compiler dependency analysis

```
#pragma acc kernels
{
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  a[i] = 0.0;
  b[i] = 1.0;
  c[i] = 2.0;
}
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  a(i) = b(i) + c(i)
}
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

C99: `restrict` Keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.

```
float *restrict ptr
```

  - Meaning: "for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points"*
- OpenACC compilers often require `restrict` to determine independence
  - Otherwise the compiler can't parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

~~`float restrict *ptr`~~
`float *restrict ptr`

http://en.wikipedia.org/wiki/Restrict

Hands On Activity (Example 1)

1. Use either restrict or independent along with acc kernels
   - Did it work?
   - How is this different than acc parallel?

```
float *restrict ptr

#pragma acc loop independent
```

OpenACC private Clause

```
#pragma acc parallel loop
  for(int i=0;i<M;i++) {
    for(int jj=0;jj<10;jj++)
       tmp[jj]=jj;
    int sum=0;
    for(int jj=0;jj<N;jj++)
       sum+=tmp[jj];
    A[i]=sum;
  }
```

```
#pragma acc parallel loop \
  private(tmp[0:10])
  for(int i=0;i<M;i++) {
     for(int jj=0;jj<10;jj++)
       tmp[jj]=jj;
     int sum=0;
     for(int jj=0;jj<N;jj++)
       sum+=tmp[jj];
     A[i]=sum;
   }
```

- Compiler cannot parallelize because tmp is shared across threads
- Also useful for live-out scalars

Deploy

Productize

- Check API return values
- Run cuda-memcheck tools

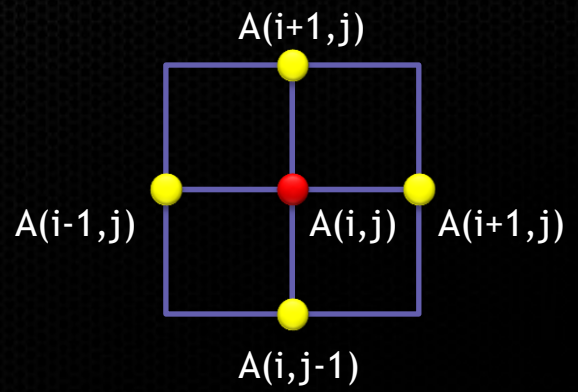
- Library distribution
- Cluster management

Early gains
Subsequent changes are evolutionary

Review

- APOD: Access Parallelize Optimize Deploy
- Use profile tools to guide your development
  - pgprof, nvvp, nvprof, etc
- Write kernels using the parallel loop or kernels constructs
- Minimize transfers using the data construct
- Use the copy clauses to control which data is transferred

Hands on Activity (Example 2)



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

- Given a 2D grid
  - Set every vertex equal to the average of neighboring vertices
    - Repeat until converged
  - Common algorithmic pattern

Hands on Activity (Example 2)

1. Build & Run
2. Switch compiler to use PGI instead of GCC
3. Use pgprof to identify the largest bottlenecks
4. Use what you have learned to parallelize the largest function
   - Create the data region within this function for now
   - Can the second largest function be parallelized?

OpenACC reduction Clause

reduction: specifies a reduction operation and variables for which that operation needs to be applied

```
int sum=0;
#pragma acc parallel loop reduction(+:sum)
for(int i=0; i<N; i++)
{
  ...
  sum+=…
}
```

Hands on Activity (Example 2)

1. Use the reduction clause to parallelize the error function
2. Optimize data movement to avoid unnecessary data copies
   - Hint: present clause

```
int sum=0;
#pragma acc parallel loop reduction(+:sum)
for(int i=0; i<N; i++)
{
  ...
  sum+=…
}
```

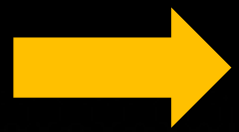
Nested Loops

- Currently we have only exposed parallelism on the outer loop
- We know that both loops can be parallelized
- Let’s look at methods for parallelizing multiple loops

OpenACC collapse Clause

collapse(n): Applies the associated directive to the following *n* tightly nested loops.

```
#pragma acc parallel
#pragma acc loop collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
    ...
```



```
#pragma acc parallel
#pragma acc loop
for(int ij=0; ij<N*N; ij++)
    ...
```

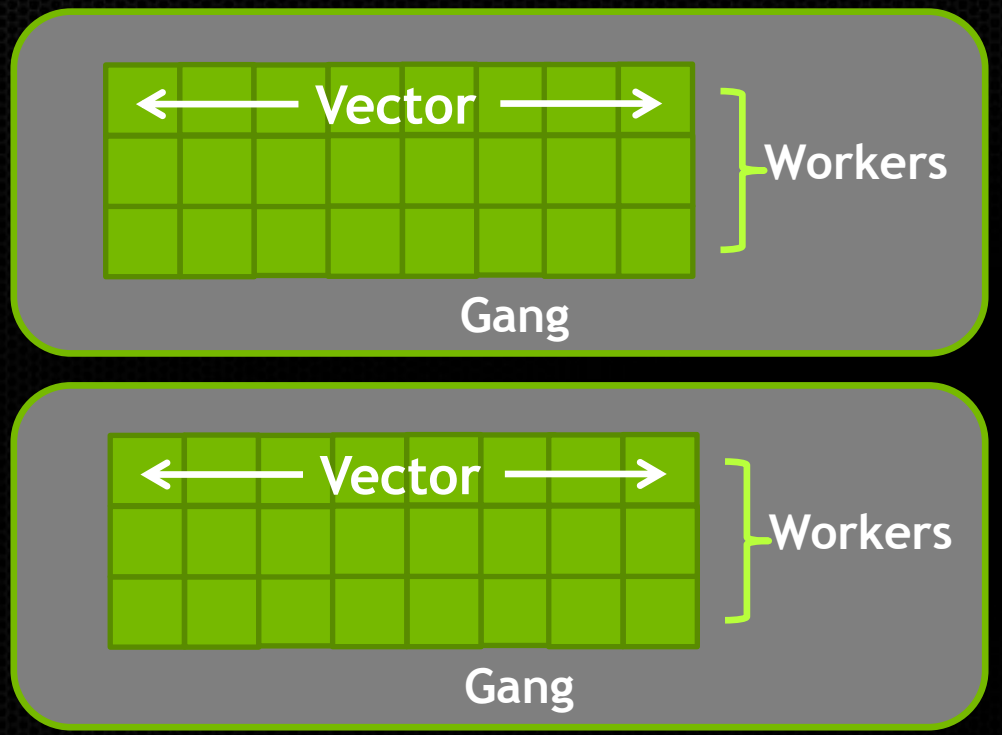
Loops must be tightly nested

Hands On Activity (Example 2)

1. Use the collapse clause to parallelize the inner and outer loops
   - Did you see any performance increase?

```
#pragma acc parallel
#pragma acc loop collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
    ...
```

OpenACC: 3 Levels of Parallelism



- *Vector* threads work in lockstep (SIMD/SIMT parallelism)
- *Workers* have 1 or more vectors.
- *Gangs* have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other

OpenACC gang, worker, vector Clauses

- gang, worker, and vector can be added to a loop clause
- Control the size using the following clauses on the parallel region
  - parallel: num_gangs(n), num_workers(n), vector_length(n)
  - Kernels: gang(n), worker(n), vector(n)

```
#pragma acc parallel loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop worker
  for (int j = 0; j < n; ++j)
    ...
```

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

parallel only goes on the outermost loop

gang, worker, vector appear once per parallel region

Hands On Activity (Example 2)

1. Replace collapse clause with some combination of gang/worker/vector
2. Experiment with different sizes using num_gangs, num_workers, and vector_length
   - What is the best configuration that you have found?

```
#pragma acc parallel loop gang num_workers(4) vector_length(32)
for (int i = 0; i < n; ++i)
  #pragma acc loop worker
  for (int j = 0; j < n; ++j)
    ...
```

Understanding Compiler Output

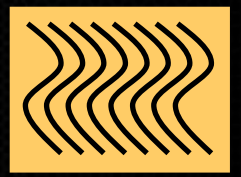
```
Accelerator kernel generated
            15, #pragma acc loop gang, worker(4) /* blockIdx.x threadIdx.y */
            17, #pragma acc loop vector(32) /* threadIdx.x */
```

- 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

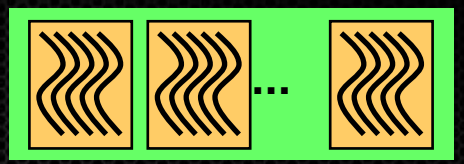
CUDA Execution Model

Software

Thread



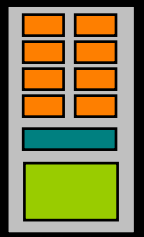
Thread Block



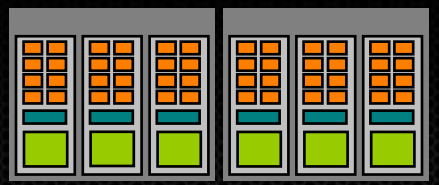
Grid

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)

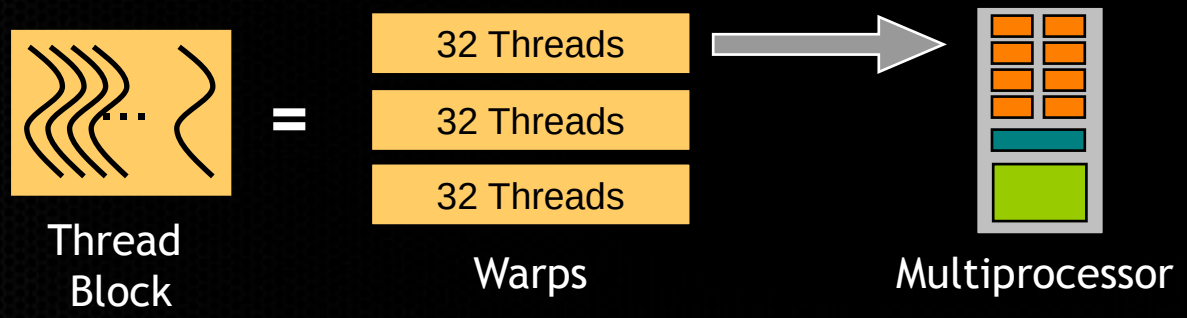
Understanding Compiler Output

```
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
  - worker 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 (COARSE GRAIN)
  - worker: mapped threads (FINE GRAIN)
  - vector: mapped to threads (FINE SIMD)
- Exact mapping is compiler dependent
- Performance Tips:
  - Use a vector size that is divisible by 32
  - Block size is num_workers * vector_length
    - Generally having the block size between 128 and 256 is ideal.

Understanding Compiler Output

```
IDX(int, int, int):
          4, Generating implicit acc routine seq
             Generating Tesla code
```

- Compiler is automatically generating a routine directive
- Some compilers may not do this
- Compiler may not be able to do it for some routines

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++)
    ...
}
```

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.

Hands On Activity (Example 2)

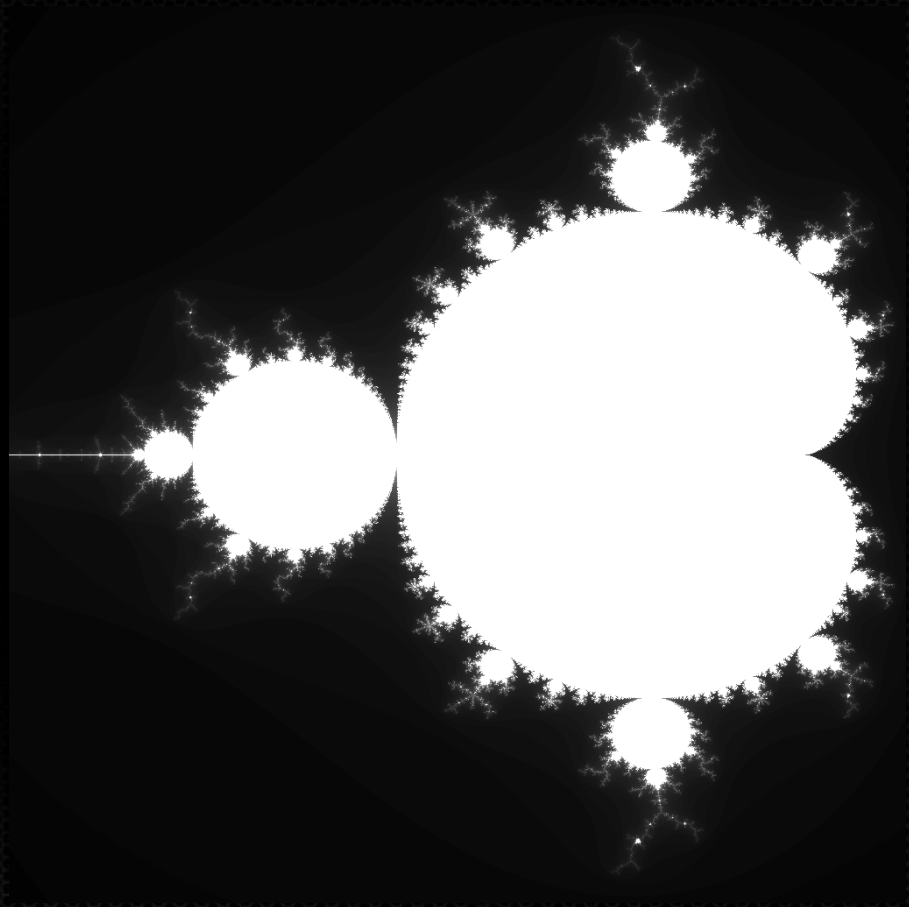
1. Modify the code to use an explicit routine
2. Rebuild and rerun

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

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

Hands On Activity (Example 3)

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



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

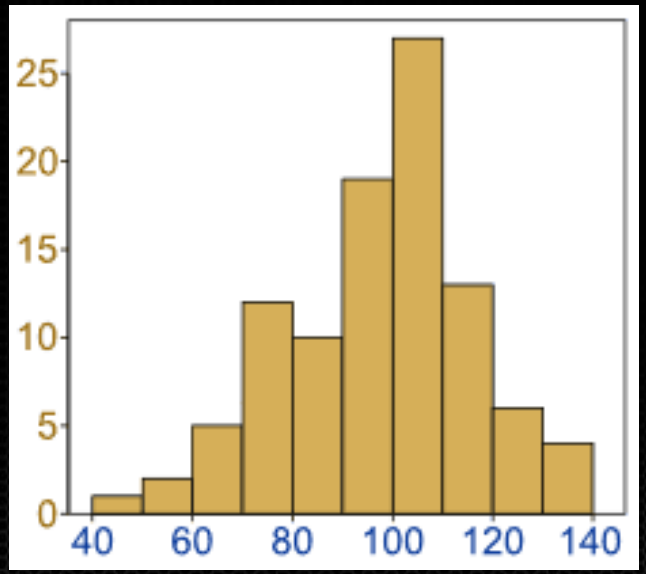
Clauses: read, write, update, capture

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

Hands On Activity (Exercise 4)

Exercise 4: Simple histogram creation

1. Use what you have learned to accelerate this code




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

OpenACC host_data directive

host_data use_device(list):

makes the address of the device data available on the host

Useful for GPU aware libraries (e.g. MPI, CUBLAS, etc)

```
#pragma acc data copy(x)
{
  // x is a host pointer here
  #pragma acc host_data use_device(x)
  {
    // x is a device pointer here
    MPI_Send(x,count,datatype,dest,tag,comm)
  }
  // x is a host pointer here
}
```

Host code that expects device pointers

CUBLAS Library & OpenACC

OpenACC Main Calling CUBLAS

```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...
cublasInit();
#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}
cublasShutdown();
```

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust
- …

http://www.pgroup.com/lit/articles/insider/v5n2a2.htm

Review

- Use atomic to parallelize codes with race conditions
- Use host_data to interoperate with cuda enabled libraries

Optimization Techniques

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

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

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;
}
```

Hands On Activity (Example 1)

1. Fuse nearby kernels
2. Rerun and profile
   - Did it get faster?
   - Do you see less launch latency?

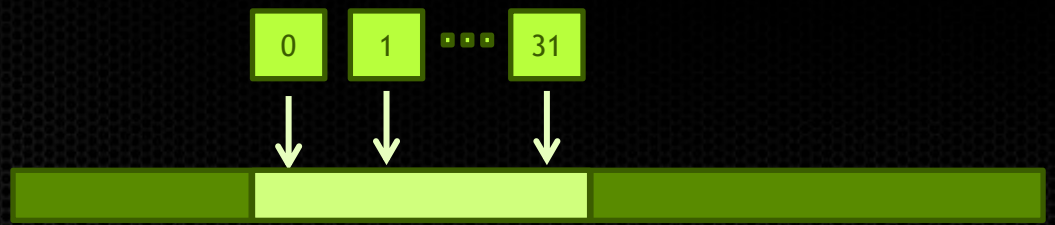
Hands On Activity (Example 2)

We are going to inspect kernel performance using the profiler

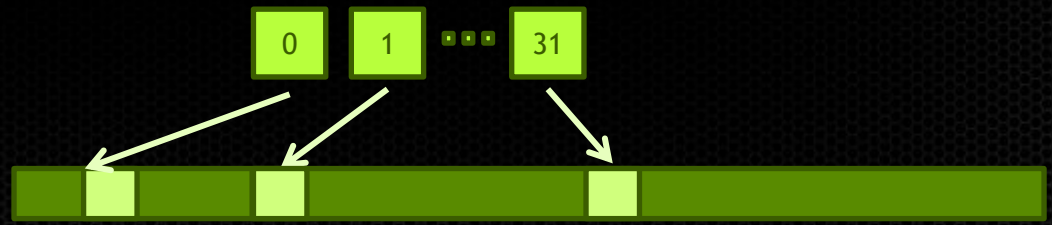
1. Edit main.cpp and reduce the number of iterations to 10.
2. Open nvvp and generate a new timeline with this example
3. Click on the first kernel
4. Click on the analysis tab
5. Click on unguided analysis
6. Click analyze all
7. Look at the properties window.
   - Do you see any warnings?

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

Hands On Activity (Example 2)

1. Find a way to fix the coalescing
   - Did we get better?
   - Why aren't we at 100%?
2. Apply this fix to both kernels
   - Verify your fix using nvvp
   - Did you see a performance improvement?

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)
```

Hands on Activity (Example 1)

1. Go back to example 1 and run it in nvvp
   - How much time is there between consecutive kernels?
2. Add the async and wait clauses
3. Recompile and rerun
   - Did the time between consecutive kernels improve?

```
#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)
```

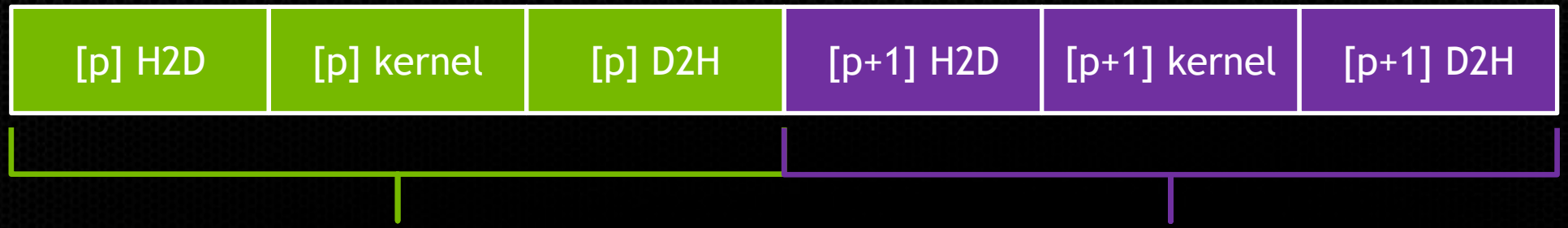
OpenACC Pipelining

```
#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])
}
```

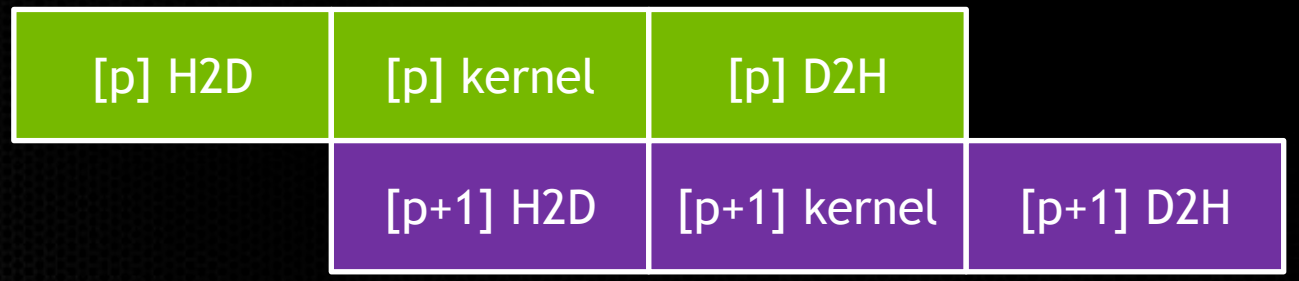
For this example, assume that each “plane” is completely independent and must be copied to/from the device.

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.)



P and P+1 Serialize



P and P+1 Overlap Data Movement

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

OpenACC Pipelining (cont.)

```
#pragma acc data create(plane)
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.

Hands On Activity (Example 3)

1. Pipeline the Mandelbrot code by batching rows
   - 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
  #pragma acc update host(...) async(...)
}
//wait for execution
#pragma acc wait
fwrite(...)
```

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

Additional Topics

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

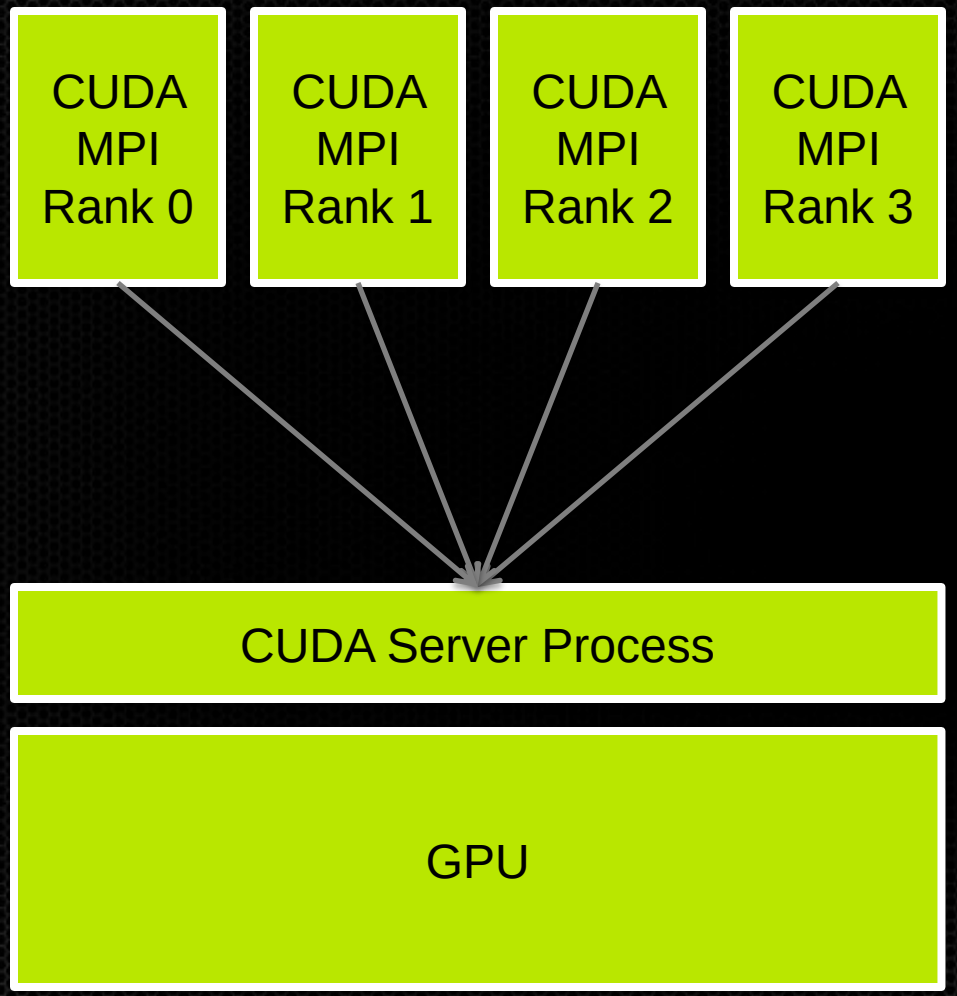
```
#include "openacc.h"

acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```

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)

Multi-Process Server Required for Hyper-Q / MPI

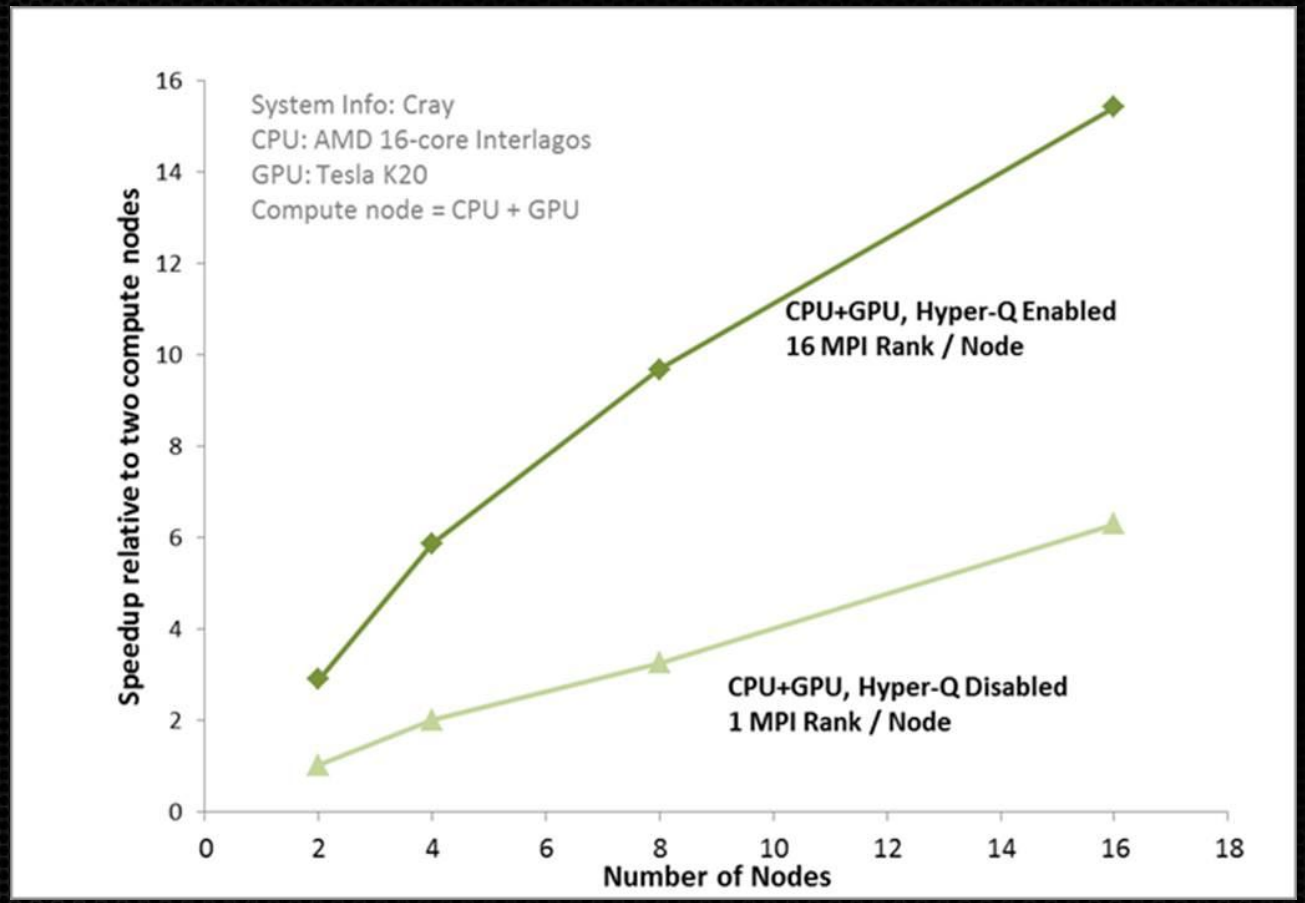


- `$ mpirun -np 4 my_cuda_app`
  - **No application re-compile to share the GPU**
- No user configuration needed
  - **Can be preconfigured by SysAdmin**

- MPI Ranks using CUDA are clients
- Server spawns on-demand per user

- One job per user
  - **No isolation between MPI ranks**
  - **Exclusive process mode enforces single *server***
- One GPU per rank
  - **No cudaSetDevice()**
  - **only CUDA device 0 is visible**

Strong Scaling of CP2K on Cray XK7



Hyper-Q with multiple MPI ranks leads to 2.5X speedup over single MPI rank using the GPU

http://blogs.nvidia.com/blog/2012/08/23/unleash-legacy-mpi-codes-with-keplers-hyper-q/

Advanced Data Layouts

- OpenACC works best with flat arrays
- Experimental support for objects is currently in PGI/14.4
  - Doesn't always work
  - Work around: Copy data to local pointers/variables (C99 & Fortran)

May work

```
#pragma acc data                        \
        copy(a[:],a.data[0:a.N]) \
        parallel loop
for(i=0;i<a.N;i++)
  a.data[i]=0;
```

Works Fine

```
int N=a.N;
float *data=a.data;
#pragma acc data                \
        copy(data[0:N]) \
        parallel loop
for(i=0;i<N;i++)
  data[i]=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

Challenge Problem: CG Solver

- Accelerate this application to the best of your ability
- Tips:
  - Matrix has at most 27 non-zeros per row (inner loop width is max 27)
- 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

Hands On Activity (Survey)

- Please help us make this workshop better in the future:

https://www.surveymonkey.com/s/XJ6GVSQ

- Questions?

Office Hours

- Let’s work on your codes now