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

Lab Connection: Please follow the lab instructions to connect to your lab

Warning: Please do not transmit any data into the AWS resources used in this lab that are not related to qwikLABS™ or the hands-on lab you are taking.

Connection

Password: 64EvSp92z

Endpoint: ec2-54-166-53-108.compute-1.amazonaws.com
World Leader in Visual Computing

GAMING

PRO VISUALIZATION

HPC & BIG DATA

MOBILE COMPUTING
Power for CPU-only Exaflop Supercomputer = Power for the Bay Area, CA (San Francisco + San Jose)

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

Hundred of GPU Accelerated Apps

NVIDIA GPU is Accelerator of Choice

% of HPC Customers with Accelerators

0% 20% 40% 60% 80% 100%

2010 2011 2012 2013

2011 2012 2013

INTEL PHI 4% OTHERS 11%

NVIDIA GPUs 85%

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

Intersect360 Research
HPC User Site Census: Systems, July 2013
Diverse Markets

FY14 Segments

- Supercomputing: 23%
- Defense/Federal: 13%
- Higher Ed/Research: 15%
- Media & Entertainment: 9%
- Consumer Web: 6%
- Med Image/Instrumentation: 11%
- CAE/MFG: 7%
- Oil & Gas: 12%
- Finance: 4%

NVIDIA estimates
Solid Growth of GPU Accelerated Apps

# of GPU-Accelerated Apps

<table>
<thead>
<tr>
<th>Year</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>113</td>
<td>182</td>
<td>272</td>
</tr>
</tbody>
</table>

Top HPC Applications

<table>
<thead>
<tr>
<th>Category</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Molecular Dynamics</td>
<td>AMBER, CHARMM, DESMOND, GROMACS, LAMMPS, NAMD</td>
</tr>
<tr>
<td>Quantum Chemistry</td>
<td>Abinit, Gaussian, GAMESS, NWChem</td>
</tr>
<tr>
<td>Material Science</td>
<td>CP2K, QMCPACK, Quantum Espresso, VASP</td>
</tr>
<tr>
<td>Weather &amp; Climate</td>
<td>COSMO, GEOS-5, HOMME, CAM-SE, NEMO, NIM, WRF</td>
</tr>
<tr>
<td>Lattice QCD</td>
<td>Chroma, MILC</td>
</tr>
<tr>
<td>Plasma Physics</td>
<td>GTC</td>
</tr>
<tr>
<td>Structural Mechanics</td>
<td>ANSYS Mechanical, LS-DYNA Implicit, MSC Nastran</td>
</tr>
<tr>
<td>Fluid Dynamics</td>
<td>ANSYS Fluent, OptiStruct, Culises (OpenFOAM)</td>
</tr>
</tbody>
</table>
Conclusion

Accelerators are the future of high performance computing

Now we have to learn how program them...
What is Heterogeneous Programming?

Application Code

- Compute-Intensive Functions: 5% of Code
- Rest of Sequential CPU Code

GPU

CPU
3 Ways to Accelerate Applications

Applications

Libraries
- Easy to use
- Most Performance

Compiler Directives
- Easy to use
- Portable code

Programming Languages
- Most Performance
- Most Flexibility
GPU Accelerated Libraries

Linear Algebra
FFT, BLAS, SPARSE, Matrix

Numerical & Math
RAND, Statistics

Data Struct. & AI
Sort, Scan, Zero Sum

Visual Processing
Image & Video
GPU Programming Languages

Numerical analytics
MATLAB, Mathematica, LabVIEW

Fortran
CUDA Fortran

C
CUDA C

C++
CUDA C++

Python
PyCUDA, Copperhead

F#
Alea.cuBase
OpenACC: Open, Simple, Portable

- Open Standard
- Easy, Compiler-Driven Approach
- Portable on GPUs and Xeon Phi

```c
main() {
    ...
    <serial code>
    ...
    #pragma acc kernels
    {
        <compute intensive code>
    }
    ...
}
```

CAM-SE Climate
6x Faster on GPU
Top Kernel: 50% of Runtime
OpenACC
The Standard for GPU Directives

Simple: Directives are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

Powerful: GPU Directives allow complete access to the massive parallel power of a GPU
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%
• Work was done in CUDA Fortran (not OpenACC)
Back to Heterogeneous Computing

Application Code

Compute-Intensive Functions

Rest of Sequential CPU Code

5% of Code

GPU

CPU
Low Latency or High Throughput?

CPU
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution
- 10’s of threads

GPU
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
- 10000’s of threads
Low Latency or High Throughput?

- **CPU** architecture must **minimize latency** within each thread
- **GPU** architecture **hides latency** with computation from other thread warps

<table>
<thead>
<tr>
<th>CPU core – Low Latency Processor</th>
<th>Computation Thread/Warp</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram of CPU core" /></td>
<td><img src="image" alt="Diagram of Computation Thread/Warp" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU Stream Multiprocessor – High Throughput Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram of GPU Stream Multiprocessor" /></td>
</tr>
</tbody>
</table>

- **Processing**
- **Waiting for data**
- **Ready to be processed**
- **Context switch**
Accelerator Fundamentals

We must expose enough parallelism to saturate the device
- Accelerator threads are slower than CPU threads
- Accelerators have orders of magnitude more threads

Fine grained parallelism is good

Coarse grained parallelism is bad
- Lots of legacy apps have only exposed coarse grain parallelism
  - i.e. MPI and possibly OpenMP
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

Lab Connection: Please follow the lab instructions to connect to your lab

**Warning:** Please do not transmit any data into the AWS resources used in this lab that are not related to qwikLABS™ or the hands-on lab you are taking.

**Connection**

- **Password:** G4Ev9p4l2z
- **Endpoint:** ec2-54-166-53-108.compute-1.amazonaws.com
Connection information

Password to your GPU Instance
Address of your GPU Instance
how to connect - nx

With NoMachine NX client 3.5

Click Configure
How to connect - NX

1. Cut and Paste address into the Host box
2. Set Desktop to Unix & GNOME
3. Choose an appropriate display size
4. Click Ok
how to connect - nx

1. Login is gpudev1
2. Copy & Paste password
3. Click Login
how to connect - nx

- If prompted, click yes
Hands On Activity (Example 1)

1. Download and untar hands on zip
   %> tar -xzf OpenAccHandsOn.tgz
   %> cd OpenAccHandsOn
   %> cd {LANGUAGE}
   %> cd example1
   %> make
   %> time ./a.out

2. Edit the makefile and switch to PGI compiler
   C++: pgCC
   Fortran: pgf90

3. Add optimization flag
   -fast
A POD: A Systematic Path to Performance

- Assess
- Parallelize
- Optimize
- Deploy
Profile the code, find the hotspot(s)
Focus your attention where it will give the most benefit
Hands On Activity (Example 1)

1. Profile the current application using pgprof
   *
   $>$ pgcollect ./a.out  
   $>$ pgprof -exe a.out  

   • For source in Fortran compile with -g

2. Double click on main
   - Which loops are the limiter?
   - Which loops are parallelizable?
```c
for(int i=0; i<N; i++)
    a[i]=1;
for(int i=0; i<N; i++)
    b[i]=2;
for(int iter=0; iter<ITERS; iter++) {
    for(int i=0; i<N; i++)
        b[i]=b[i]*a[i];
    for(int i=0; i<N; i++)
        a[i]=b[i]+a[i];
    for(int i=0; i<N; i++)
        b[i]=b[i]/a[i];
}
return 0;
```
Parallelize

Applications

Libraries
Easy to use
Most Performance

Compiler Directives
Easy to use
Portable code

Programming Languages
Most Performance
Most Flexibility
Common Mistakes

- We will highlight common mistakes people make throughout this presentation.
- Look for the ⚠️ symbol to indicate common errors.
OpenACC Directive Syntax

- **C/C++**

  ```c
  #pragma acc directive [clause [,,] clause] ...
  ```
  ...

  ...often followed by a structured code block

- **Fortran**

  ```fortran
  !$acc directive [clause [,,] clause] ...
  ```
  ...

  ...often paired with a matching end directive surrounding a structured code block:

  ```fortran
  !$acc end directive
  ```

  **Don’t forget acc**
OpenACC Example: SAXPY

**SAXPY in C**

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(n), y(n), a
    integer :: n, i

    !$acc parallel loop
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end parallel loop
end subroutine saxpy

...!
Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...```
**OpenACC parallel loop Directive**

**parallel**: a parallel region of code. The compiler generates a parallel *kernel* for that region.

**loop**: identifies a loop that should be distributed across threads

parallel & loop are often placed together

```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
    y[i] = a*x[i]+y[i];
}
```

**Kernel:**
A function that runs in parallel on the GPU
Hands On Activity (Example 1)

1. Modify the Makefile to build with OpenACC
   -acc            Compile with OpenACC
   -ta=tesla      Target NVIDIA GPUS

2. Add parallel loop directives to parallelizable loops

3. Run again:
   %> time ./a.out
   Did the application get faster or slower?

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

Remove -g from the compile flags
Hands On Activity (Example 1)

1. How do we know what happened?
2. Modify the Makefile again
   -Minfo=accel Verbose OpenACC Output
3. Rebuild the application
pgCC -acc -Minfo=accel -ta=nvidia main.cpp

main:

18, Accelerator kernel generated
20, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
18, Generating present_or_copy(b[:N])
Generating Tesla code
21, Accelerator kernel generated
23, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
21, Generating present_or_copyin(b[:N])
Generating present_or_copy(a[:N])
Generating Tesla code
24, Accelerator kernel generated
26, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
24, Generating present_or_copyin(a[:N])
Generating present_or_copy(b[:N])
Generating Tesla code

Generated 3 Kernels
Optimize

Profile-driven optimization

CPU Tools:
- gprof
- pgprof
- vampir
- TAU

GPU Tools:
- nvidia-nsight NVIDIA Nsight IDE
- nvvp NVIDIA Visual Profiler
- nvprof Command-line profiling
NVIDIA’s Visual Profiler
Timeline

Guided System

Analysis

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

Perform Compute Analysis
The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

Perform Latency Analysis
Instructions and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform these analyses.

Renew Analysis
If you modify the kernel you need to rerun your application to update this analysis.

Stall Reasons
- execution dependency
- data request
- texture
- synchronization
- other
**NVPROF**

**Command line profiler**

- `nvprof ./exe`
  - Report kernel and transfer times directly
- Collect profiles for NVVP
  - `%> nvprof -o profile.out ./exe`
  - `%> nvprof --analysis-metrics -o profile.out ./exe`
- Collect for MPI processes
  - `%> mpirun -np 2 nvprof -o profile.%p.out ./exe`
- Collect profiles for complex process hierarchies
  - `--profile-child-processes, --profile-all-processes`
- Collect key events and metrics
  - `%> nvprof --metrics flops_sp ./exe`
  - `--query-metrics --query-events`
Hands On Activity (Example 1)

1. Profile using PGI's built in OpenACC profiling
   %> PGI_ACC_TIME=1 ./a.out

2. Run the application with nvprof and inspect output

3. Create a new NVVP session
   - Click on File
   - Select the executable
   - Click Next -> Finish

4. Explore the profile
   - Is the GPU busy?
   - What is the GPU doing?
   - How much time do we spend in kernels vs transfers?
23: compute region reached 1000 times

   23: kernel launched 1000 times
   grid: [3907]  block: [256]
       device time(us): total=21,135 max=493 min=2 avg=21
       elapsed time(us): total=53,352 max=561 min=30 avg=53

23: data region reached 1000 times

   23: data copyin transfers: 2000
       device time(us): total=18,899 max=51 min=5 avg=9
   26: data copyout transfers: 1000
       device time(us): total=6,812 max=47 min=  avg=6

26: data region reached 1000 times

   26: data copyin transfers: 2000
       device time(us): total=18,900 max=50 min=2 avg=9
   29: data copyout transfers: 1000
NVPROF Output

==22104== NVPROF is profiling process 22104, command: ./a.out
==22104== Profiling application: ./a.out
==22104== Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>59.04%</td>
<td>3.16076s</td>
<td>5000</td>
<td>632.15us</td>
<td>630.45us</td>
<td>649.59us</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>36.56%</td>
<td>1.95739s</td>
<td>3000</td>
<td>652.46us</td>
<td>618.74us</td>
<td>672.95us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>1.90%</td>
<td>101.98ms</td>
<td>1000</td>
<td>101.97us</td>
<td>79.874us</td>
<td>104.00us</td>
<td>main_24_gpu</td>
</tr>
<tr>
<td>1.42%</td>
<td>75.930ms</td>
<td>1000</td>
<td>75.929us</td>
<td>75.170us</td>
<td>76.930us</td>
<td>main_21_gpu</td>
</tr>
<tr>
<td>1.08%</td>
<td>57.828ms</td>
<td>1000</td>
<td>57.827us</td>
<td>57.538us</td>
<td>59.106us</td>
<td>main_18_gpu</td>
</tr>
</tbody>
</table>
NVVP Output

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as...
1. Copy input data from CPU memory/NIC to GPU memory
1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC
The `data` construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```c
#pragma acc data
{
    #pragma acc parallel loop
    ...
    #pragma acc parallel loop
    ...
}
```

Array used within the data region will remain on the GPU until the end of the data region.

Be careful with scoping rules
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**

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

**Fortran**

```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?

```c
#pragma acc data copy(...)
{
    ...
}
```
**OpenACC enter exit Directives**

**enter**: Defines the start of an unstructured data region  
clauses: \texttt{copyin(list)}, \texttt{create(list)}

**exit**: Defines the end of an unstructured data region  
clauses: \texttt{copyout(list)}, \texttt{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).

```c
#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)
1. Now use enter/exit data instead of a structured data region

```c
#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

```c
#pragma acc update host(x[0:count])
MPI_Send(x,count,datatype,dest,tag,comm);
```
The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```c
#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];
  }
}
```

The compiler identifies 2 parallel loops and generates 2 kernels.
### OpenACC parallel loop vs. kernels

<table>
<thead>
<tr>
<th>PARALLEL LOOP</th>
<th>KERNELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires analysis by programmer to ensure safe parallelism</td>
<td>Compiler performs parallel analysis and parallelizes what it believes safe</td>
</tr>
<tr>
<td>Straightforward path from OpenMP</td>
<td>Can cover larger area of code with single directive</td>
</tr>
<tr>
<td></td>
<td>Gives compiler additional leeway.</td>
</tr>
</tbody>
</table>

Both approaches are equally valid and can perform equally well.
1. Modify the code to use \texttt{kernels} directive instead of \texttt{parallel} loop

   Did it work?

\begin{verbatim}
#pragma acc kernels
{
  ...
}
\end{verbatim}
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.

```c
#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)
}
}
```

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.

```c
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

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**?

```c
float *restrict ptr
#pragma acc loop independent
```
OpenACC `private` Clause

```c
#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;
}
```

```c
#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
Given a 2D grid

- Set every vertex equal to the average of neighboring vertices
  - Repeat until converged

Common algorithmic pattern

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

```c
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

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

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

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

**Loops must be tightly nested**
1. Use the `collapse` clause to parallelize the inner and outer loops
   Did you see any performance increase?

```c
#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 parallel 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 \texttt{collapse} clause with some combination of \texttt{gang/worker/vector}

2. Experiment with different sizes using \texttt{num\_gangs}, \texttt{num\_workers}, and \texttt{vector\_length}
   
   What is the best configuration that you have found?

\begin{verbatim}
#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)
    ...
\end{verbatim}
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

**Hardware**

- Thread
- Scalar Processor
- Thread Block
- Multiprocessor
- Grid
- 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
A thread block consists of a group 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.
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

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

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

```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
```
Hands On Activity (Example 2)

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

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

Clauses: read, write, update, capture

```c
#pragma acc parallel loop
for(int i=0; i<N; i++) {
    #pragma acc atomic update
    a[i%100]++;
}
```
Exercise 4: Simple histogram creation

1. Use what you have learned to accelerate this code

```c
#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)

```c
#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
```
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
• …
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 loops)
Inlining

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

```c
#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

```c
#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;
}
```
Hands On Activity (Example 1)

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

```c
#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

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.)

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

```plaintext
#pragma acc ...
for rows
  for cols
    ...
//copy image to host
fwrite(...);
```

```plaintext
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

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

```c
#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

```c
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