An Introduction to OpenACC
Part II

Wei Feinstein
HPC User Services@LSU

LONI Parallel Programming Workshop 2015
Louisiana State University
Roadmap

• Recap of OpenACC
• OpenACC with GPU-enabled library
• Code profiling
• Code tuning with performance tool
• Programming on multi-GPUs
Large Scale of applications

- Genomics
- Deep Learning
- Coastal Storm Prediction

Multi-core CPU

GPU

Many-core CPU

OpenMP

OpenACC

Pragmas-based

Compiler-directives

high programming level
Heterogeneous Programming on GPUs

Applications

Libraries

“Drop-in” Acceleration

Compiler Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
OpenACC Execution Model

Application Code

$acc parallel

$acc end parallel

Compute-Intensive Functions

Generate Parallel Code for GPU

Rest of Sequential CPU Code

GPU

CPU
OpenACC Memory Model

Two separate memory spaces between host and accelerator
- Data transfer by DMA transfers
- Hidden from the programmer in OpenACC, so beware:
  - Latency
  - Bandwidth
  - Limited device memory size

Accelerator:
- No guarantee for memory coherence → beware of race conditions
- Cache management done by compiler, user may give hints
1. Copy input data from CPU memory to GPU memory
Data Flow

1. Copy input data from CPU memory to GPU memory
2. Execute GPU Kernel
1. Copy input data from CPU memory to GPU memory
2. Execute GPU Kernel
3. Copy results from GPU memory to CPU memory

Data Flow
Basic OpenACC directives

C/C++

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

Fortran

```fortran
!$acc directive-name [clause [[,] clause]...]
```
“Kernels / Parallel” Constructs

- **Kernels**
  - C/C++
    - `#pragma acc kernels [clauses]`
  - Fortran
    - `!$acc kernels [clauses]`

- **Parallel**
  - C/C++
    - `#pragma acc parallel loop [clauses]`
  - Fortran
    - `!$acc parallel loop [clauses]`
“Data” Construct

Data: management of data transfer between host and device

C/C++

```c
#pragma acc data [clauses]
```

Fortran

```fortran
!$acc data [clauses]
```
“host_data” Construct

C/C++

```c
#pragma acc kernels host_data use_device(list)
```

Fortran

```fortran
!$acc kernels host_data use_device(list)
```

- Make the address of device data available on host
- Specified variable addresses refer to device memory
- Variables must be present on device
- Can only be used within a data region
OpenACC compilers

- PGI compiler for C, C++ and Fortran
- Cray CCE compilers for Cray systems
- CAPS compilers
GPU Tools

Code performance increases with the deployment of GPU tools.
void saxpy(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
void saxpy_acc(int n, float a, float *x, float *y) {
    #pragma acc kernels
    for (int i = 0; i < n; ++i){
        y[i] = a * x[i] + y[i];
    }
}

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc kernels
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
}
Saxpy_openacc_v1

```c
void saxpy_acc(int n, float a, float *x, float *y) {
    #pragma acc kernels
    for (int i = 0; i < n; ++i){
        y[i] = a * x[i] + y[i];
    }
}

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
    }
    ...

Parallel the loop
```
void saxpy_acc(int n, float a, float *x, float *y) {
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    } a
}

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
}...
cublasSaxpy from cuBLAS library

void cublasSaxpy( int n,
                 const float *alpha,
                 const float *x,
                 int incx,
                 float *y,
                 int incy)

• A function in the standard Basic Linear Algebra Subroutines (BLAS) library
• cuBLAS: GPU-accelerated drop-in library ready to be used on GPUs.


### Saxpy_openacc_v2

```c
void saxpy_acc(int n, float a, float *x, float *y) {
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        y[i] = a * x[i] + y[i];
    } a
}

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
    }
    ...
```

### Saxpy_cuBLAS

```c
extern void
cublasSaxpy(int, float, float*, int, float*, int);

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i) {
        x[i] = 1.0f; y[i] = 0.0f;
    }
    // Perform SAXPY
    #pragma acc host_data use_device(x,y)
cublasSaxpy(n, 2.0, x, 1, y, 1);
    ...
```

[http://docs.nvidia.com/cuda](http://docs.nvidia.com/cuda)
void saxpy_acc(int n, float a, float *x, float *y) {
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i){
        y[i] = a * x[i] + y[i];
    } a
}

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
        #pragma acc parallel loop
        for (int i = 0; i < n; ++i) {
            x[i] = 1.0f; y[i] = 0.0f;
        }
    // Perform SAXPY
    saxpy_acc(n, a, x, y);
    }

extern void
cublasSaxpy(int,float,float*,int,float*,int);

int main(){
    ...
    // Initialize vectors x, y
    #pragma acc data create(x[0:n]) copyout(y[0:n])
        #pragma acc parallel loop
        for (int i = 0; i < n; ++i) {
            x[i] = 1.0f; y[i] = 0.0f;
        }
    // Perform SAXPY
    #pragma acc deviceptr (x,y)
cublasSaxpy(n, 2.0, x, 1, y, 1);
    ...

http://docs.nvidia.com/cuda
GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

Linear Algebra
- FFT, BLAS, SPARSE, Matrix
  - NVIDIA cuFFT, cuBLAS, cuSPARSE

Numerical & Math
- RAND, Statistics
  - NVIDIA Math Lib

Data Struct. & AI
- Sort, Scan, Zero Sum
  - Thrust
  - GPU AI - Board Games

Visual Processing
- Image & Video
  - NVIDIA NPP
  - NVIDIA Video Encode

4th HPC Parallel Programming Workshop
An Introduction to OpenACC-PartII
Optimize Data Locality

Optimize Loop Performance

Parallelize Loops with OpenACC

Application Profiling
The Himeno code

- 3D Poisson equation solver
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound

- Fortran and C implementations are available from http://accc.riken.jp/2467.htm

- The scalar version for simplicity
  - We will discuss the parallel version using OpenACC
**Optimize Loop Performance**

**Parallelize Loops with OpenACC**

**Optimize Data Locality**

**Application Profiling**
Application Profiling

• pgprof - PGI performance profiler

  pgcc -Minfo=ccff -o yourcode_exe yourcode.c
  pgcollect yourcode_exe
  pgprof -exe yourcode_exe

• gprof - GNU command line profiler

  gcc -pg -o yourcode_exe yourcode.c
  ./yourcode_exe
  gprof yourcode_exe gmon.out > yourcode_pro.output

• nvprof - command line profiler -nvprof
### An Introduction to OpenACC-PartII

#### Process/Thread Browser for application `./himeno.pg`

<table>
<thead>
<tr>
<th>Profile</th>
<th>Seconds</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pprof.out</td>
<td>8.6020</td>
<td>100%</td>
</tr>
<tr>
<td>P 0</td>
<td>8.6020</td>
<td>100%</td>
</tr>
</tbody>
</table>

#### Process/Thread Viewer for routine 'jacobi'

<table>
<thead>
<tr>
<th>Routine</th>
<th>Seconds</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi</td>
<td>8.3674</td>
<td>97%</td>
</tr>
<tr>
<td>P 0</td>
<td>8.3674</td>
<td>97%</td>
</tr>
</tbody>
</table>
4th HPC Parallel Programming Workshop

An Introduction to OpenACC-PartII
Application Profiling

• pgprof - PGI visual profiler
  
  pgcc –Minfo=ccff –o yourcode_exe yourcode.c
  pgcollect yourcode_exe
  pgprof –exe yourcode_exe

• gprof - GNU command line profiler
  
  gcc –pg –o yourcode_exe yourcode.c
  ./yourcode_exe
  gprof yourcode_exe gmon.out > yourcode_pro.output

• nvprof - command line profiler -nvprof
Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td>42.39</td>
<td>0.39</td>
<td>0.39</td>
</tr>
<tr>
<td>26.09</td>
<td>0.63</td>
<td>0.24</td>
</tr>
<tr>
<td>16.30</td>
<td>0.78</td>
<td>0.15</td>
</tr>
<tr>
<td>7.61</td>
<td>0.85</td>
<td>0.07</td>
</tr>
<tr>
<td>7.61</td>
<td>0.92</td>
<td>0.07</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
</tr>
</tbody>
</table>

- **%**: The percentage of the total running time of the program used by this function.
- **Cumulative**: A running sum of the number of seconds accounted for by this function and those listed above it.
- **Self**: The number of seconds accounted for by this function alone. This is the major sort for this listing.
- **Calls**: The number of times this function was invoked, if this function is profiled, else blank.
Amdahl’s Law
Application Profiling

Parallelize Loops with OpenACC

Optimize Loop Performance

Optimize Data Locality

Optimize Data Locality

An Introduction to OpenACC-PartII
Performance Profiling via NVVP
Performance Profiling via NVVP

1. CUDA Application Analysis
2. Check Overall GPU Usage

Results:
- Low Compute / Memcpy Efficiency [2.112 ms / 277.225 ms = 0.008]
  The amount of time performing compute is low relative to the amount of time required for memcpy.
- Low Memcpy/Compute Overlap [0 ns / 2.112 ms = 0%]
  The percentage of time when memcpy is being performed in parallel with compute is low.
Performance Profiling via NVVP
Performance

Why did OpenACC slow down here?

![Graph showing speedups (x fold) for serial, 2 threads, 4 threads, 8 threads, 16 threads, and OpenACC. The graph indicates a significant drop in speedup at 16 threads.](image-url)
Application Profiling

Optimize Loop Performance

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Data Locality

Parallelize Loops with OpenACC

Optimize Loop Performance

Application Profiling
Data transfer

• Data movement is expensive causing bottleneck to performance
• Minimize data movement
• Data caching
  – #pragma acc data copyin/copyout/copy
    • Allocate memory on device and copy data from host to device, or back, or both
  – #pragma acc data present
Improved performance with better data locality

1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to:

- **Low Memcpy/Compute Overlap** [0 ns / 314.434 μs = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low.

- **Low Kernel Concurrency** [0 ns / 3.563 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low.

- **Inefficient Memcpy Size**
Small memory copies do not enable the GPU to fully use the host to device bandwidth.

- **Low Memcpy Throughput** [5.241 MB/s avg, for memcpy accounting for 100% of all memcpy]
The memory copies are not fully using the available host to device bandwidth.
Improved performance with better data locality
Improved performance with better data locality

![Graph showing speedups (x fold) for Serial, 2 threads, 4 threads, 8 threads, 16 threads, and OpenACC. The y-axis represents speedup, and the x-axis represents the number of threads. The graph shows a significant increase in speedup with more threads and OpenACC compared to serial execution.](image-url)
Optimize Loop Performance → Parallelize Loops with OpenACC → Optimize Data Locality → Application Profiling → Optimize Loop Performance
Three Levels of Parallelism

OpenACC provides more detailed control over parallelization via gang, worker, and vector clauses

• Gang:
  – Share iterations across the gangs (grids) of a parallel region

• Worker:
  – Share iterations across the workers (warps) of the gang

• Vector:
  – Execute the iterations in SIMD mode
CUDA Kernels: Parallel Threads

A kernel is a function executed on the GPU as an array of threads in parallel.

All threads execute the same code, can take different paths.

Each thread has an ID Select input/output data
Control decisions.

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
- **Blocks** are grouped into a grid
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
- **Blocks** are grouped into a **grid**
- A **kernel** is executed as a **grid of blocks of threads**
MAPPING OPENACC TO CUDA
OpenACC Execution Model on CUDA

The OpenACC execution model has three levels:
*gang*, *worker*, and *vector*

For GPUs, the mapping is implementation-dependent. Some possibilities:

*gang*=block, *worker*=warp, and *vector*=threads of a warp

Depends on what the compiler thinks is the best mapping for the problem
OpenACC Execution Model on CUDA

The OpenACC execution model has three levels:
gang, worker, and vector

For GPUs, the mapping is implementation-dependent.

...But explicitly specifying that a given loop should map to
gangs, workers, and/or vectors is optional anyway

Further specifying the number of gangs/workers/vectors is also optional
So why do it? To tune the code to fit a particular target architecture in a
straightforward and easily re-tuned way.
Three Levels of Parallelism

C/C++

```c
#pragma acc parallel [num_gangs() / num_workers() / vector_length()]
```

Fortran

```fortran
!$acc parallel [num_gangs() / num_workers() / vector_length()]
```

C/C++

```c
#pragma acc loop[(num_gangs) / (num_workers) / (vector_length)]
```

Fortran

```fortran
!$acc parallel [(num_gangs) / (num_workers) / (vector_length)]
```
Multiple GPUs card on a single node?
Device Management

• Internal control variables (ICVs):
  – *Acc-device-type-var* → Controls which type of accelerator is used
  – *Acc-device-num-var* → Controls which accelerator device is used

• Setting ICVs by API calls
  – *acc_set_device_type()* *acc_set_device_num()*

• Querying of ICVs
  – *acc_get_device_type()* *acc_get_device_num()*
Device Management

acc_get_num_devices

- Returns the number of accelerator devices attached to host and the argument specifies type of devices to count

C:
- int acc_get_num_devices(acc_device_t)

Fortran:
- Integer function acc_get_num_devices(devicetype)
Device Management

acc_set_device_num

- Sets ICV ACC_DEVICE_NUM
- Specifies which device of given type to use for next region. Can not be called in a parallel, kernels or data region

C:

- Void acc_set_device_num(int,acc_device_t)

Fortran:

- Subroutine
  acc_set_device_num(devicenum,devicetype)
Device Management

• **Acc_get_device_num**
  - Return value of ICV ACC_DEVICE_NUM
  - Return which device of given type to use for next region
  - Can not be called in a parallel, kernels or data region

• C:
  - Void acc_get_device_num(acc_device_t)

• Fortran:
  - Subroutine acc_get_device_num(devicetype)
Directives-based programming on single node with multi-GPU cards

SAXPY Code

```c
// initialization
for (i = 0; i < n; i++){
    x[i] = 1.0; y[i] = 2.0;
}
// calculation
for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i]*2;
}
```
Directive-based programming on single node with multi-GPU cards

```c
// get # of GPU cards on this node
int gpu_ct=acc_get_num_devices(acc_device_nvidia);
// create one thread for each GPU kernel
#pragma omp parallel private(tid) num_threads(gpu_ct)
{
    // Obtain thread id
    tid = omp_get_thread_num();
    // assign one kernel to one OpenMP thread
    acc_set_device_num(tid + 1, acc_device_nvidia);
    #pragma acc data create(x[0:n],y[0:n]) copyin(a)
    #pragma acc kernels loop
        for (i = 0; i < n; i++){
            x[i] = 1.0; y[i] = 2.0;
        }
    #pragma acc kernels loop
        for (i = 0; i < n; i++){
            y[i] = a*x[i] + y[i]*2;
        }
} //end of omp parallel
```
Directive-based programming on single node with multi-GPU cards

- OpenACC only supports one GPU
- Hybrid model:
  - OpenACC + OpenMP to support multi-GPU parallel programming
- Limitations
  - Lack direct device-to-device communications
Conclusions

- OpenACC is a powerful programming model using compiler directives
- Progressive, productive code porting
- Portable and easy to maintain
- Interoperability
- Advanced features provide deeper control
Introduction to OpenACC PartII Lab
Getting Started

Connect to shelob cluster:

```
ssh username@shelob.hpc.lsu.edu
```

Extract the lab to your account:

```
tar xzvf /home/user/himeno.tar.gz
```

Change to the lab directory:

```
cd himeno
```

Request a interactive node

```
qsub –l –A allocation –lwalltime=2:00:00 –lnodes=1:ppn=16
```

Login in to the interactive node

```
ssh –X shelobxxx
```
Goal: code profiling to identify the target for parallelization 
(use your own code would be great)

```
cd Practical1
pgprof: PGI visual profiler
pgcc -Minfo=ccff mycode.c -o mycode
pgcollect mycode
pgprof -exe mycode
```
Exercise 1

Goal: code profiling to identify the target for parallelization (use your own code would be great)

gprof: GNU profiler

```bash
gcc mycode.c -o mycode -pg
./mycode
gprof mycode gmon.out >mycode_profile.output
```
Exercise 2

Goal: Identify hot spots in your code to improve performance

```bash
  cd Practical2
Compile source code (e.g. version 1)
  make ver=01
Check performance at command line (-Minfo=accel is turned on)
  ./himeno.01
Use nvvp visual profiler by typing:
  nvvp
```
Exercise 3

Goal: use nvvp to fine tune your code for better performance via using more OpenACC directives

```bash
cd Practical3
```

Compile the source code (e.g. version 1)

```bash
make ver=01
```

Use nvvp visual profiler by typing:

```bash
nvvp
```
Exercise 4

Goal: use OpenACC with GPU-enabled library

```bash
cd Practical4
pgprof: PGI visual profiler
pgcc -Minfo=ccff mycode.c -o mycode
pgcollect mycode
pgprof --exe mycode
```
Exercise 5

Goal: use multi-GPUs cards on a single node

```
  cd Practical5
```