OPENACC ONLINE COURSE 2018
Week 1 – Introduction to OpenACC
Jeff Larkin, Senior DevTech Software Engineer, NVIDIA
ABOUT THIS COURSE

3 Part Introduction to OpenACC

- Week 1 – Introduction to OpenACC
- Week 2 – Data Management with OpenACC
- Week 3 – Optimizations with OpenACC

Each week will have a corresponding lab, only an hour and a web browser is required.

Please ask questions in the Q&A box, our TA’s will answer as quickly as possible.
Enable YOU to accelerate YOUR applications with OpenACC.
WEEK 1 OUTLINE
Topics to be covered

- What is OpenACC and Why Should You Care?
- Profile-driven Development
- First Steps with OpenACC
- Week 1 Lab
- Where to Get Help
INTRODUCTION TO OPENACC
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

OpenACC
OPENACC IS…

a directives-based parallel programming model designed for performance and portability.

```c
main()
{
    <serial code>
    #pragma acc kernels
    {
        <parallel code>
    }
}
```
OpenACC Directives

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

```c
#pragma acc data copyin(a,b) copyout(c) 
{
    ...
    #pragma acc parallel 
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            c[i] = a[i] + b[i];
            ...
        }
    }
    ...
}
```
OPENACC

**Incremental**
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

**Single Source**
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

**Low Learning Curve**
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.
**OPENACC**

**Incremental**
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

**Enhance Sequential Code**

```c
#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
    < loop code >
}

#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
    < loop code >
}
```

**Begin with a working sequential code.**
- Parallelize it with OpenACC.
- Rerun the code to verify correctness and performance
OPENACC

**Supported Platforms**
- POWER
- Sunway
- x86 CPU
- AMD GPU
- NVIDIA GPU
- PEZY-SC

**Single Source**
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

```c
int main(){
    ...
    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        <loop code>
}
```
OPENACC

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

```c
int main()
{
    <sequential code>
    #pragma acc kernels
    {
        <parallel code>
    }
}
```

The programmer will give hints to the compiler.

The compiler parallelizes the code.
OPENACC

**Incremental**
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

**Single Source**
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

**Low Learning Curve**
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.
DIRECTIVE-BASED HPC PROGRAMMING
Who’s Using OpenACC

- 3 OF TOP 5 HPC APPS
- 5 OF 13 CAAR CODES
- 2 OF LAST 9 FINALISTS

- 450 DOMAIN EXPERTS
- ACCELERATED APPS
- 100,000 DOWNLOADS
Using OpenACC allowed us to continue development of our fundamental algorithms and software capabilities simultaneously with the GPU-related work. In the end, we could use the same code base for SMP, cluster/network and GPU parallelism. PGI’s compilers were essential to the success of our efforts.
For VASP, OpenACC is the way forward for GPU acceleration. Performance is similar and in some cases better than CUDA C, and OpenACC dramatically decreases GPU development and maintenance efforts. We’re excited to collaborate with NVIDIA and PGI as an early adopter of CUDA Unified Memory.
Using OpenACC our scientists were able to achieve the acceleration needed for integrated fusion simulation with a minimum investment of time and effort in learning to program GPUs.
Adding OpenACC into MAS has given us the ability to migrate medium-sized simulations from a multi-node CPU cluster to a single multi-GPU server. The implementation yielded a portable single-source code for both CPU and GPU runs. Future work will add OpenACC to the remaining model features, enabling GPU-accelerated realistic solar storm modeling.
OPENACC SYNTAX
A **pragma** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A **directive** in Fortran is a specially formatted comment that likewise instructs the compiler in the compilation of the code and can be freely ignored.

“**acc**” informs the compiler that what will come is an OpenACC directive.

**Directives** are commands in OpenACC for altering our code.

**Clauses** are specifiers or additions to directives.
EXAMPLE CODE
LAPLACE HEAT TRANSFER
Introduction to lab code - visual

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.
EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
- Common, useful algorithm
- Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

$$A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}$$
while ( err > tol && iter < iter_max ) {
    err = 0.0;

    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i] ) ;
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
PROFILE-DRIVEN DEVELOPMENT
**OPENACC DEVELOPMENT CYCLE**

- **Analyze** your code to determine most likely places needing parallelization or optimization.

- **Parallelize** your code by starting with the most time consuming parts and check for correctness.

- **Optimize** your code to improve observed speed-up from parallelization.
Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing.

Lab Code: Laplace Heat Transfer

Total Runtime: 39.43 seconds

- swap 19.04s
- calcNext 21.49s
PROFILING SEQUENTIAL CODE

First sight when using PGPROF

- Profiling a simple, sequential code
- Our sequential program will run on the CPU
- To view information about how our code ran, we should select the “CPU Details” tab
Within the “CPU Details” tab, we can see the various parts of our code, and how long they took to run.

We can reorganize this info using the three options in the top-right portion of the tab.

We will expand this information, and see more details about our code.
PROFILING SEQUENTIAL CODE

CPU Details

- We can see that there are two places that our code is spending most of its time
- 21.49 seconds in the “calcNext” function
- 19.04 seconds in a memcpy function
- The c_mcopy8 that we see is actually a compiler optimization that is being applied to our “swap” function
We are also able to select the different elements in the CPU Details by double-clicking to open the associated source code.

Here we have selected the “calcNext:37” element, which opened up our code to show the exact line (line 37) that is associated with that element.
OPENACC PARALLEL LOOP DIRECTIVE
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{
    When encountering the `parallel` directive, the compiler will generate 1 or more parallel `gangs`, which execute redundantly.
}
```
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
    for (int i = 0; i < N; i++)
    {
        // Do Something
    }
}
```

This loop will be executed redundantly on each gang.
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
  for(int i = 0; i < N; i++)
  {
    // Do Something
  }
}
```

This means that each gang will execute the entire loop.
OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

C/C++

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

Fortran

```fortran
!$acc parallel
  !$acc loop
  do i = 1, N
    a(i) = 0
  end do
!$acc end parallel
```

- Use a `parallel` directive to mark a region of code where you want parallel execution to occur.
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran.
- The `loop` directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs.
OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

- This pattern is so common that you can do all of this in a single line of code

- In this example, the parallel loop directive applies to the next loop

- This directive both marks the region for parallel execution and distributes the iterations of the loop.

- When applied to a loop with a data dependency, parallel loop may produce incorrect results

C/C++

```c
#pragma acc parallel loop
for(int i = 0; j < N; i++)
    a[i] = 0;
```

Fortran

```fortran
!$acc parallel loop
do i = 1, N
    a(i) = 0
end do
```
The `loop` directive informs the compiler which loops to parallelize.
OPENACC PARALLEL LOOP DIRECTIVE

Parallelizing many loops

- To parallelize multiple loops, each loop should be accompanied by a parallel directive.
- Each parallel loop can have different loop boundaries and loop optimizations.
- Each parallel loop can be parallelized in a different way.
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results.

```c
#pragma acc parallel loop
for(int i = 0; i < N; i++)
    a[i] = 0;

#pragma acc parallel loop
for(int j = 0; j < M; j++)
    b[j] = 0;
```
PARALLELIZE WITH OPENACC PARALLEL LOOP

```c
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                              A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
        iter++;
    }
}
```

Parallelize first loop nest, max reduction required.

Parallelize second loop.

We didn’t detail how to parallelize the loops, just which loops to parallelize.
The reduction clause takes many values and “reduces” them to a single value, such as in a sum or maximum.

Each thread calculates its part.

The compiler will perform a final reduction to produce a single global result using the specified operation.

```cpp
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];

for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma parallel acc loop \
    reduction(+:tmp)
    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition/Summation</td>
<td><code>reduction(+:sum)</code></td>
</tr>
<tr>
<td>*</td>
<td>Multiplication/Product</td>
<td><code>reduction(*:product)</code></td>
</tr>
<tr>
<td>max</td>
<td>Maximum value</td>
<td><code>reduction(max:maximum)</code></td>
</tr>
<tr>
<td>min</td>
<td>Minimum value</td>
<td><code>reduction(min:minimum)</code></td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
<td><code>reduction(&amp;:val)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
<td><code>reduction(&amp;&amp;:val)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BUILD AND RUN THE CODE
The command to compile C code is 'pgcc'

The command to compile C++ code is 'pgc++'

The command to compile Fortran code is 'pgfortran'

The -fast flag instructs the compiler to optimize the code to the best of its abilities

$ pgcc -fast main.c
$ pgc++ -fast main.cpp
$ pgfortran -fast main.F90
PGI COMPILER BASICS

-Minfo flag

- The -Minfo flag will instruct the compiler to print feedback about the compiled code
- -Minfo=accel will give us information about what parts of the code were accelerated via OpenACC
- -Minfo=opt will give information about all code optimizations
- -Minfo=all will give all code feedback, whether positive or negative

```
$ pgcc -fast -Minfo=all main.c
$ pgc++ -fast -Minfo=all main.cpp
$ pgfortran -fast -Minfo=all main.f90
```
PGI COMPILER BASICS

- **ta flag**

- The `-ta` flag enables building OpenACC code for a “Target Accelerator” (TA)
- `-ta=multicore` – Build the code to run across threads on a multicore CPU
- `-ta=tesla:managed` – Build the code for an NVIDIA (Tesla) GPU and manage the data movement for me (more next week)

```
$ pgcc -fast -Minfo=accel -ta=tesla:managed main.c
$ pgc++ -fast -Minfo=accel -ta=tesla:managed main.cpp
$ pgfortran -fast -Minfo=accel -ta=tesla:managed main.f90
```
$ pgcc -fast -ta= multicore -Minfo=accel laplace2d_uvm.c
main:

  63, Generating Multicore code
  64, #pragma acc loop gang
  64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
      Generating reduction (max: error)
  66, Loop is parallelizable
  74, Generating Multicore code
  75, #pragma acc loop gang
  75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
  77, Loop is parallelizable
OPENACC SPEED-UP

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
BUILDING THE CODE (GPU)

```
$ pgcc -fast -ta=tesla:managed -Minfo=accel laplace2d_uvm.c
main:

63, Accelerator kernel generated
Generating Tesla code
64, #pragma acc loop gang /* blockIdx.x */
  Generating reduction(max:error)
66, #pragma acc loop vector(128) /* threadIdx.x */
63, Generating implicit copyin(A[:])
Generating implicit copyout(Anew[:])
Generating implicit copy(error)
66, Loop is parallelizable
74, Accelerator kernel generated
Generating Tesla code
75, #pragma acc loop gang /* blockIdx.x */
77, #pragma acc loop vector(128) /* threadIdx.x */
74, Generating implicit copyin(Anew[:])
Generating implicit copyout(A[:])
77, Loop is parallelizable
```
OPENACC SPEED-UP

Speed-up

- SERIAL: 1.00X
- MULTICORE: 3.05X
- NVIDIA TESLA V100: 37.14X

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
CLOSING REMARKS
KEY CONCEPTS
This week we discussed…

- What is OpenACC
- How profile-driven programming helps you write better code
- How to parallelize loops using OpenACC’s **parallel loop** directive to improve time to solution

Next Week:

- Managing your data with OpenACC
OPENACC RESOURCES

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

Resources
https://www.openacc.org/resources

Success Stories
https://www.openacc.org/success-stories

Compilers and Tools
https://www.openacc.org/tools

Events
https://www.openacc.org/events

FREE Compilers