OPENACC ONLINE COURSE

Module 1 – Introduction to OpenACC

Robert Searles
NVIDIA Corporation
ABOUT THIS COURSE

3 Part Introduction to OpenACC

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

Each module will have a corresponding lab
COURSE OBJECTIVE

Enable **YOU** to accelerate **YOUR** applications with OpenACC.
MODULE 1 OUTLINE

Topics to be covered

- What is OpenACC and Why Should You Care?
- Profile-driven Development
- First Steps with OpenACC
- Lab 1
- Where to Get Help
INTRODUCTION TO OPENACC
3 WAYS TO ACCELERATE 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>
    }
}
```

Add Simple Compiler Directive
OpenACC Directives

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

Initiate Parallel Execution

Optimize Loop Mappings

• Incremental
• Single source
• Interoperable
• Performance portable
• CPU, GPU, Manycore
## OPENACC STRENGTHS

### 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 requirement 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

Begin with a working sequential code.
Parallelize it with OpenACC.
Rerun the code to verify correctness and performance
OpenACC: 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 is meant to be easy to use, and easy to learn. The programmer remains in familiar C, C++, or Fortran. No requirement to learn low-level details of the hardware. The programmer will give hints to the compiler. The compiler parallelizes the code.
DIRECTIVE-BASED HPC PROGRAMMING

Who’s Using OpenACC?

3 OF TOP 5 HPC APPS

5 OF 13 CAAR CODES

ACCELERATED APPS

725 TRAINED EXPERTS

SLACK MEMBERS

160,000+ DOWNLOADS

Intersect360

OpenACC

PGI

GTC16  GTC17  GTC18  GTC19

53  100  116  194

102  326  846

GTC17  GTC18  GTC19

Community Edition
OpenACC is an industry initiative to help mathematicians and engineers write high performance applications on GPUs and multi- and many-core systems. It is a simple, portable, and easy-to-use API for data-parallel programming. OpenACC can be used as a complement to other programming models, such as MPI and OpenMP, to simplify the programming of complex algorithms. It enables developers to leverage the power of GPUs without the need for deep knowledge of GPU programming, thus accelerating code development and reducing the time to market.

OpenACC provides a way to express data parallelism in an application, allowing the compiler to automatically generate efficient code for execution on GPUs. This approach simplifies the programming of applications that require high computational power, such as scientific simulations, data analysis, and machine learning.

OpenACC is supported by leading vendors of hardware and software, including NVIDIA, Intel, and AMD, as well as major programming language vendors. This broad industry support ensures that OpenACC is a viable and robust solution for GPU programming.

Incorporating OpenACC into existing applications or developing new applications can significantly improve performance by leveraging the parallel processing capabilities of GPUs. OpenACC simplifies the transition to GPU computing, making it accessible to a wider range of developers and researchers.

OpenACC is a practical solution for developers to accelerate GPU-based hardware while retaining a single source for almost all the COSMO physics codes.
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 instructions the compiler in it 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
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++) {
            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.
PROFILE SEQUENTIAL CODE

Profile Your Code

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 NSight Systems

- Profiling a simple, sequential code
- Our sequential program will run on the CPU.
- To view information about how our code ran, we should right click the “NVTX” row and select “Show in Events View”.
PROFILING SEQUENTIAL CODE

CPU Details

- Within the “Events View” tab, we can see the various parts of our code, and how long they took to run.
- We see the “while” loop taking up the majority of our runtime (42.2 s).
- We can click the arrow to examine the calls made within this loop.
PROFILING SEQUENTIAL CODE

CPU Details

- We see repeated calls to “calcNext” and “swap” within our while loop. We will focus on these!

- We also can see this on our zoomed in timeline.

- NVTX lets us push ranges onto a stack. A new row within the NVTX section will be created for each nested range.
PROFILING SEQUENTIAL CODE

CPU Details

- Nsight’s --stats flag also prints out runtime statistics
- We see that initialize and deallocate take almost no runtime
- calcNext accounts for 36.2% of total runtime
- swap accounts for 13.8% of total runtime

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Total Time</th>
<th>Instances</th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.0</td>
<td>42293194393</td>
<td>1</td>
<td>42293194393.0</td>
<td>42293194393</td>
<td>42293194393</td>
<td>while ( error &gt; tol &amp;&amp; iter &lt; iter_max )</td>
</tr>
<tr>
<td>36.2</td>
<td>30601618780</td>
<td>1000</td>
<td>30601618.8</td>
<td>28602296</td>
<td>40445804</td>
<td>calcNext</td>
</tr>
<tr>
<td>13.8</td>
<td>11673978245</td>
<td>1000</td>
<td>11673978.2</td>
<td>11322706</td>
<td>13462161</td>
<td>swap</td>
</tr>
<tr>
<td>0.0</td>
<td>23810757</td>
<td>1</td>
<td>23810757.0</td>
<td>23810757</td>
<td>23810757.0</td>
<td>initialize</td>
</tr>
<tr>
<td>0.0</td>
<td>1973444</td>
<td>1</td>
<td>1973444.0</td>
<td>1973444</td>
<td>1973444</td>
<td>deallocate</td>
</tr>
</tbody>
</table>
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.
}

*gang*
*gang*
*gang*
*gang*
*gang*
*gang*
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

#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;
}
```

- 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.

Fortran

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

- 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 LOOP 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
```
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

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

The **loop** directive informs the compiler which loops to parallelize.

---

**gang**

**gang**

**gang**

**gang**

---
Parallelizing many loops

- To parallelize multiple loop nests, each should be accompanied by a parallel directive.

- Each parallel loop nest can have different loop boundaries and loop optimizations.

- Each parallel loop nest can be parallelized in a different way.

- This is the recommended way to parallelize multiple loop nests. Attempting to parallelize multiple loop nests within the same parallel region may give performance issues or unexpected results.
PARALLELIZE WITH OPENACC PARALLEL LOOP

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++;
}
REDUCTION CLAUSE

- The **reduction** clause takes many values and “reduces” them to a single value, such as in a sum or maximum.
- Each partial result is calculated in parallel.
- A **single result** is created by combining the partial results using the specified operation.

```c
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];
```

```c
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];
```

```c
for( i = 0; i < size; i++ )
    for( j = 0; j < size; j++ )
        double tmp = 0.0f;
        #pragma acc parallel 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>reduction(+:sum)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication/Product</td>
<td>reduction(*:product)</td>
</tr>
<tr>
<td>max</td>
<td>Maximum value</td>
<td>reduction(max:max:maximum)</td>
</tr>
<tr>
<td>min</td>
<td>Minimum value</td>
<td>reduction(min:min:minimum)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
<td>reduction(&amp;:val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
<td>reduction(&amp;&amp;:val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BUILD AND RUN THE CODE
PGI COMPILER BASICS
pgcc, pgc++ and pgfortran

- 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

```bash
$ 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 automatically (more next module)

```
$ 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
```
The -Mcuda flag is needed when using NVTX regions in our code.

-lnvToolsExt – link the NVTX API

This allows us to use NVTX regions in our code for both CPU and GPU profiling.

$ pgcc -fast -Minfo=accel -ta=tesla:managed -Mcuda -lnvToolsExt main.c
$ pgc++ -fast -Minfo=accel -ta=tesla:managed -Mcuda -lnvToolsExt main.cpp
$ pgfortran -fast -Minfo=accel -ta=tesla:managed -Mcuda -lnvToolsExt main.f90
BUILDING THE CODE (MULTICORE)

$ pgcc -fast -ta=multicore -Minfo=accel -Mcuda -lnvToolsExt 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 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz
BUILDING THE CODE (GPU)

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

   63, Accelerator kernel generated
   Generating Tesla code
   Generating reduction
   Generating implicit copyin
   Generatingimplicit copyout

   64, #pragma acc loop gang /* blockIdx.x */ */
   Generating reduction(max:error)
   Generating implicit copyin(A[:])
   Generating implicit copyout(Anew[:])

   66, #pragma acc loop vector(128) /* threadIdx.x */ */
   Generating implicit copy(error)

   63, Generating implicit copyin(Anew[:])
   Generating implicit copyout(Anew[:])
   Generating implicit copy(error)

   66, Loop is parallelizable

   74, Accelerator kernel generated
   Generating Tesla code
   Generating reduction
   Generating implicit copyin
   Generating implicit copyout

   75, #pragma acc loop gang /* blockIdx.x */ */
   Generating reduction(max:error)
   Generating implicit copyin(Anew[:])
   Generating implicit copyout(A[:])

   77, Loop is parallelizable
OPENACC SPEED-UP

PGI 19.10, NVIDIA Tesla V100, IBM POWER9 22-core CPU @ 3.07GHz
CLOSING REMARKS
KEY CONCEPTS
This module 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 module:
- 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

FREE Compilers
PGI
Community Edition

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

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

https://www.openacc.org/community#slack