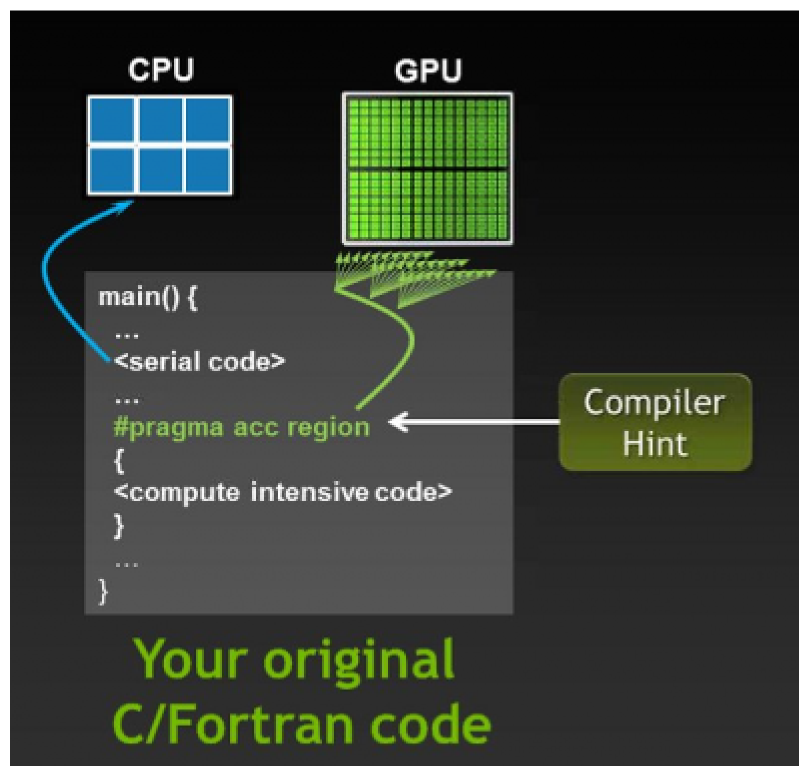
Broken

OpenACC by example

What is OpenACC?

It is a directive based standard to allow developers to take advantage of accelerators such as GPUs from NVIDIA and AMD, Intel's Xeon Phi (via CAPS compilerconversion to OpenCL), FPGAs, and even DSP chips.

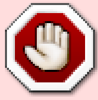
OpenACC compiler directives are **code annotations** that enable the compiler to parallelise code while ensuring thread-safety. The big difference between **OpenACC** and the existing **OpenMP** standard is that OpenACC primarily targets the **GPU** rather than CPU, whereas OpenMP is generally CPU only. That said, OpenACC can also target the CPU so it is flexible; the idea is that it will adapt to the target system.



Programming Model

- Bulk of computations executed in CPU, compute intensive regions offloaded to accelerators
- Accelerators execute parallel regions:
  - Use work-sharing and kernel directives
  - Specification of coarse and fine grain parallelization
- The host is responsible for
  - Allocation of memory in accelerator
  - Initiating data transfer
  - Sending the code to the accelerator
  - Waiting for completion
  - Transfer the results back to host

  - Deallocating memory
  - Queue sequences of operations executed by the device



OpenACC is not GPU Programming
OpenACC is Expressing Parallelism in your code.

Directives

OpenACC provides a fairly rich pragma language to annotate **data location**, **data transfer**, and **loop or code block parallelism**. The syntax of OpenACC pragmas (sometimes referred to as OpenACC directives) is:

- C/C++: `#pragma acc directive-name [clause [[,] clause]…] new-line`
- Fortran: `!$acc directive-name [clause [[,] clause]…] new-line`

Directives are:

- Regions
- Loops
- Data Structure
- …

Parallel Directive

Programmer identifies a loop as having parallelism, compiler generates a parallel kernel for that loop.

C/C++ `#pragma acc parallel [clause [,clause]…] new-line structured block`
Fortran `!$acc parallel [clause [[,] clause]…] new-line`

The main clauses for the `#pragma acc parallel` directive are:

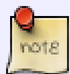
- `if`(condition)
- `async`[(scalar-integer-expression)]
- `num_gangs` (scalar-integer-expression)
- `num_workers` (scalar-integer-expression)
- `vector_length` (scalar-integer-expression)
- `reduction` (operator:list)
- `copy` (list)
- `copyout`(list)
- `create` (list)
- `private` (list)
- `firstprivate`(list)

In the most case, clauses are handled automatically by the compiler

C/C++	Fortran

`#pragma acc parallel loop`
`for (int i = ; i < n; ++i)`
`y[i] = a*x[i] + y[i];`
`}`
	`$!acc parallel loop`
	`do i=1,n`
	`y(i) = a*x(i)+y(i)`
	`enddo`
	`$!acc end parallel loop`

Kernel: A parallel function that runs on the **GPU**



Most often **parallel** directive will be used as **parallel loop**

Kernels Directive

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

C/C++ `#pragma acc kernels [clause [,clause]…] new-line structured block`
Fortran `!$acc kernels [clause [[,] clause]…] new-line`

Example

```
!$acc kernels
do i=1,n !loop 1
a(i) = 0.0
b(i) = 1.0
c(i) = 2.0
end do
do i=1,n !loop 2
a(i) = b(i) + c(i)
end do
!$acc end kernels
```

In this example, the compiler identifies **2 parallel loops** and

generates **2 kernels** (GPU kernel).

OpenACC parallel vs. kernels

PARALLEL

- 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.

Data Transfers

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/C++ `#pragma acc data [clause [,clause]…] new-line structured block`
Fortran `!$acc data [clause [[,] clause]…] new-line`

Example

```
!$acc data
do i=1,n
  a(i) = 0.0
  b(i) = 1.0
  c(i) = 2.0
end do

do i=1,n
  a(i) = b(i) + c(i)
end do

!$acc end data
```

Arrays a, b, and c will remain on the GPU until the end of the data region.

Data clause

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

Compile & Run

We use PGI compiler. First we load the compiler using module:

```
$ module load pgi
```

To compile C/C++:

```
$ pgcc –acc [-Minfo=accel] [-ta=tesla] –o saxpy_acc saxpy.c
```

To compile for Fortran:

```
$ pgf90 –acc [-Minfo=accel] [-ta=tesla] –o saxpy_acc saxpy.f90
```

The generated code can be executed directly in GPU using SGE with tesla.q queue

Example

Simple SAXPY program.

saxpy.c

```
#include <stdio.h>
#include <stdlib.h>

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

int main(void)
{
```

```
    const int n = 1 << 20;
    float *x = new float[n];
    float *y = new float[n];

    for (int i = ; i < n; ++i) {
        x[i] = (float)i / (float)n;
        y[i] = (float)i;

        // Perform SAXPY on 1M elements
         saxpy(1<<20, 2.0, x, y);
     }
}
```

Compile

```
$ pgcc -acc -fast -ta=tesla -Minfo=accel saxpy.c  -o acc_saxpy.pgi.exe
saxpy:
      7, Generating implicit copyin(x[:n])
         Generating implicit copy(y[:n])
      8, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Run on GPU node

The PGI compiler provides automatic instrumentation when **PGI_ACC_TIME=1** at runtime

```
./saxpy.pgi.exe

 saxpy  NVIDIA  devicenum=
    time(us): 1,322
    7: compute region reached 1 time
        8: kernel launched 1 time
            grid: [8192]  block: [128]
             device time(us): total=68 max=68 min=68 avg=68
            elapsed time(us): total=579 max=579 min=579 avg=579
    7: data region reached 2 times
        7: data copyin transfers: 2
             device time(us): total=835 max=427 min=408 avg=417
        10: data copyout transfers: 1
             device time(us): total=419 max=419 min=419 avg=419
```

Run on GPU node using SGE

acc.sge

```
#!/bin/bash

#$ -N test_openacc
#$ -o $JOB_NAME.$JOB_ID.out

#$ -q tesla.q

### load PGI
module load pgi

### Instrument execution
export PGI_ACC_TIME=1

./myACCProgram
```

Submit to SGE

```
$ qsub acc.sge
```

For short execution, you can use `gputest.q`

```
$ qsub -q gputest.q acc.sge
```

Benchmarks

We are used Matrix Multiply to compare execution of four executions.

Compilation level used is `-03`

Problem Size	OpenACC	OpenMP 8 cores (PGI)	Intel Compiler 8 cores
1024	0.5s	2.7s	0.2s
2048	1.03s	24s	2s
4096	4.0s	2m43s	15s
5120	6.7s	4mn32	30s
6144	10s	7mn49	50s

Links

- https://developer.nvidia.com/openacc
- https://devblogs.nvidia.com/parallelforall/getting-started-openacc/
- http://www.training.prace-ri.eu/uploads/tx_pracetmo/DirectiveBasedProgJU.pdf
- http://www.training.prace-ri.eu/uploads/tx_pracetmo/OpenACCDirectives.pdf

From:
http://mesowiki.univ-fcomte.fr/dokuwiki/ - **Wiki Utilisateurs - Mésocentre de calcul de Franche-Comté**

Permanent link:
http://mesowiki.univ-fcomte.fr/dokuwiki/doku.php/openacc

Last update: **2022/05/24 17:07**