

Introduction to OpenACC

Hossein Pourreza

May 2019

Acknowledgement: Most of examples and pictures are from PSC (https://www.psc.edu/images/xsedetraining/OpenACC_Nov2017/OpenACC_Introduction_To_OpenACC.PDF)
Please do not redistribute the slides without permission of the original author (John Urbanic).



THE UNIVERSITY OF
CHICAGO

Office of Research and
National Laboratories
Research Computing Center

Logistics for this workshop

- Login to Midway using your CNetID username/password
 - Using terminal:
 - `ssh -Y cnetid@midway2.rcc.uchicago.edu`
 - Using ThinLinc:
 - <https://midway2.rcc.uchicago.edu>
- Once you are logged in, download the workshop material by running:
 - `git clone https://github.com/rcc-uchicago/openacc-intro.git`

Logistics for this workshop

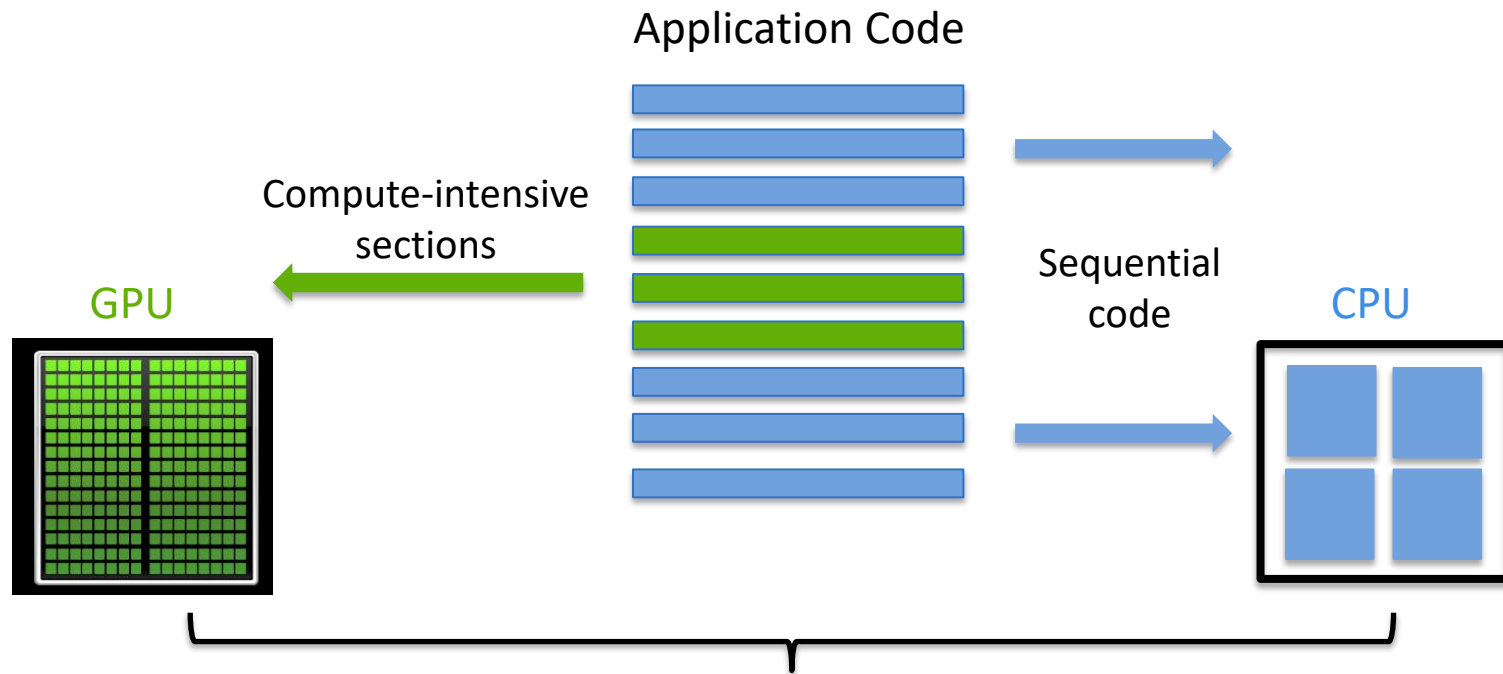
- Please run the `module load pgi/2017` command to access the PGI compilers
- We will be use the `openacc` reservation from the `gpu2` partition on Midway2
- Please use editors such as `vim`, `nano`, etc. to edit the files
 - Copying codes/commands from slides may result in unexpected error messages

GPU: Graphical Processing Unit

- Designed for graphics
- Optimized for parallel tasks with hundreds of cores
- Good at doing simple and repeated calculations
- High memory bandwidth and flops/watt
- We will use NVIDIA GPUs in this workshop



Heterogeneous computing

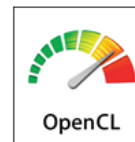


Picture is adapted from <https://computing.llnl.gov/tutorials/2014.09.15-16.NVIDIA-OpenACC.pdf>

GPU computing

- Libraries
 - NVIDIA Math Libraries, CULA , MAGMA, etc.
- Compiler directives
 - OpenACC
- Programming languages
 - NVIDIA CUDA toolkit for C/C++, PGI CUDA Fortran, NVIDIA OpenCL, PyCUDA

Increased
flexibility and
complexity



OpenACC

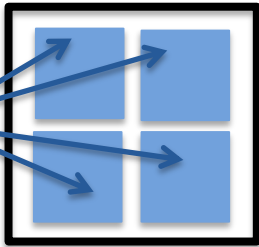
- Programming standard developed by NVIDIA, Cray, CAPS, and PGI
- Includes GPU directives to annotate C/C++ and Fortran code
- Can be used on different GPUs (NVIDIA and AMD)

Key advantages

- High-level
 - No involvement of OpenCL, CUDA, etc.
- Single source
 - Compile the same program for accelerators or serial execution
- Efficient
 - Comparable to the low-level implementations of the same algorithm
- Performance portable
 - Supports GPU accelerators and co-processors from multiple vendors
- Incremental

OpenACC is similar to OpenMP

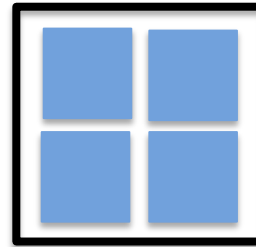
CPU



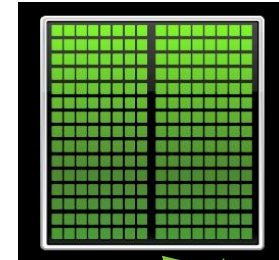
```
main(){
    double pi = 0.0; long i;

    #pragma omp parallel for reduction(+:pi)
    for(i=0; i < N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```

CPU



GPU



```
main(){
    double pi = 0.0; long i;

    #pragma acc kernels
    for(i=0; i < N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```

Basics

- In C/C++ all OpenACC directives start with `#pragma acc`
 - These lines will be skipped by non-OpenACC compilers
 - You may get a warning message but it should be OK

```
#pragma acc <directive> [clause]
{
    ...
}
```

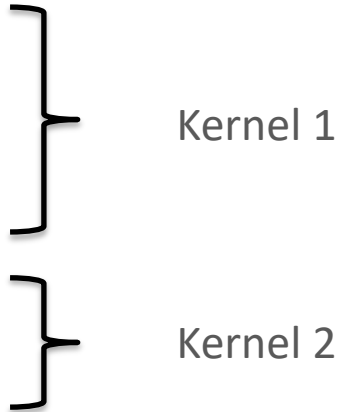
- In Fortran all OpenACC directives start with `!$acc`

```
!$acc <directive> [clause]
...
!$acc end directive
```

kernels directive

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

```
#pragma acc kernels
{
    for(i=1;i<n;i++){
        a[i] = 0.0;
        b[i] = 1.0;
    }
    for(i=0;i < n;i++){
        a[i] = b[i] + 5;
    }
}
```



The diagram shows two code blocks from the previous block, each enclosed in a right-facing curly brace. The first brace, spanning the first for loop, is labeled "Kernel 1". The second brace, spanning the second for loop, is labeled "Kernel 2".

Kernel: A parallel routine to run on the GPU

parallel loop directive

C

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

Fortran

```
!$acc parallel loop
...
!$acc end parallel loop
```

Similar to OpenMP we can use `parallel for` in C and `parallel do` in Fortran. To keep it consistent, `parallel loop` can be used in both.

kernels vs. parallel loop

- kernels
 - Gives more flexibility to compiler to parallelize loops
 - Covers larger sections of code
- parallel loop
 - Requires programmer to analyze the dependencies
 - Is a simple transition from OpenMP

SAXPY

```
#include "ticktock.h"
int main(){
    int n=1e7; float a = 5./3.;int i;
    float *x = malloc(sizeof(float)*n);
    float *y = malloc(sizeof(float)*n);

    //initialize vectors
    for (i=0; i<n; i++){
        x[i] = 2.0f;
        y[i] = (i+1.)*(i-1.);
    }
    tick_tock tt;
    tick(&tt);
    saxpy(n,a,x,y);
    tock(&tt);
    free(x);
    free(y);
    return 0;
}
```

```
void saxpy(int n, float a, float *x,
float *restrict y){
    int i;
    #pragma acc kernels
    for (i=0; i<n; i++)
        y[i] = a * x[i] + y[i];
} //end of saxpy
```

The `restrict` keyword: C99 standard

- Important for optimization of serial as well as OpenACC and OpenMP code.
- Promise to the compiler for a pointer
`float *restrict y`
 - Meaning: “for the lifetime of `y`, only it or a value directly derived from it (such as `y + 1`) will be used to access the object to which it points”
- Limits the effects of pointer aliasing

Compile and run

```
$pgcc -acc -Minfo=accel saxpy.c ticktock.c -o saxpy
$pgf90 -acc -Minfo=accel saxpy.f90 ticktock.f90 -o
saxpy
```

```
saxpy.c:
```

```
saxpy:
```

```
5, Generating implicit copy(y[:n])
```

```
Generating implicit copyin(x[:n])
```

```
6, Loop is parallelizable
```

```
Accelerator kernel generated
```

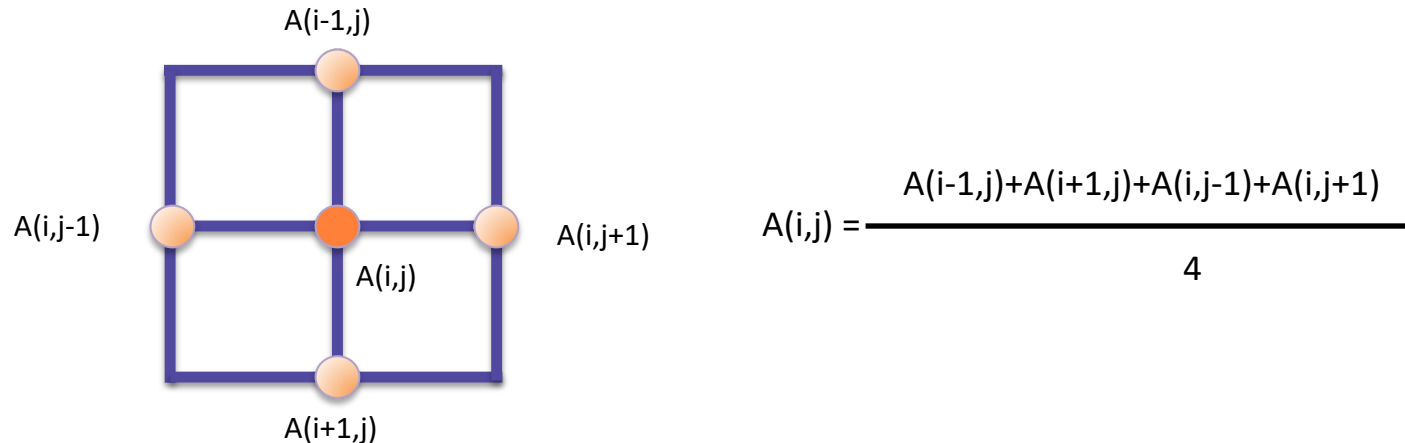
```
Generating Tesla code
```

```
6, #pragma acc loop gang, vector(128)/* blockIdx.x threadIdx.x */
```

```
$. /saxpy
```


Fundamental example

- Laplace equation for two dimensional heat conduction problem:



- Jacobi iteration:

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

From PSC (https://www.psc.edu/images/xsedetraining/OpenACC_Nov2017/OpenACC_Introduction_To_OpenACC.PDF), Copyright John Urbanic.

Serial code

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {  
    for(i = 1; i <= ROWS; i++)  
        for(j = 1; j <= COLUMNS; j++)  
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] +  
                Temperature_last[i-1][j] + Temperature_last[i][j+1] +  
                Temperature_last[i][j-1]);  
  
    dt = 0.0;  
  
    for(i = 1; i <= ROWS; i++)  
        for(j = 1; j <= COLUMNS; j++){  
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);  
            Temperature_last[i][j] = Temperature[i][j];  
        }  
  
    if((iteration % 100) == 0) {  
        track_progress(iteration);  
    }  
    iteration++;  
}
```

Compute

Update

Print

Helper functions

```
void initialize(){
    int i,j;
    for(i = 0; i <= ROWS+1; i++)
        for(j = 0; j <= COLUMNS+1; j++)
            Temperature_last[i][j] = 0.0;
```

```
void track_progress(int iteration){
    int i;
    printf("-- Iteration: %d --\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++)
        printf("[%d,%d]: %5.2f ", i, i,
                Temperature[i][i]);
    printf("\n");
}
```

```
// these boundary conditions never change throughout run
```

```
// set left side to 0 and right to a linear increase
```

```
for(i = 0; i <= ROWS+1; i++){
    Temperature_last[i][0] = 0.0;
    Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
}
```

```
// set top to 0 and bottom to linear increase
```

```
for(j = 0; j <= COLUMNS+1; j++){
    Temperature_last[0][j] = 0.0;
    Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
}
```

```
}
```

OpenACC version

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {  
    #pragma acc kernels  
    for(i = 1; i <= ROWS; i++)  
        for(j = 1; j <= COLUMNS; j++)  
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] +  
                Temperature_last[i-1][j] + Temperature_last[i][j+1] +  
                Temperature_last[i][j-1]);  
    dt = 0.0;  
    #pragma acc kernels  
    for(i = 1; i <= ROWS; i++)  
        for(j = 1; j <= COLUMNS; j++){  
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);  
            Temperature_last[i][j] = Temperature[i][j];  
        }  
  
    if((iteration % 100) == 0) {  
        track_progress(iteration);  
    }  
    iteration++;  
}
```

Compile and run

```
$pgcc -acc -Minfo=accel jacobi_acc.c -o jacobi_acc  
$pgf90 -acc -Minfo=accel jacobi_acc.f90 -o jacobi_acc
```

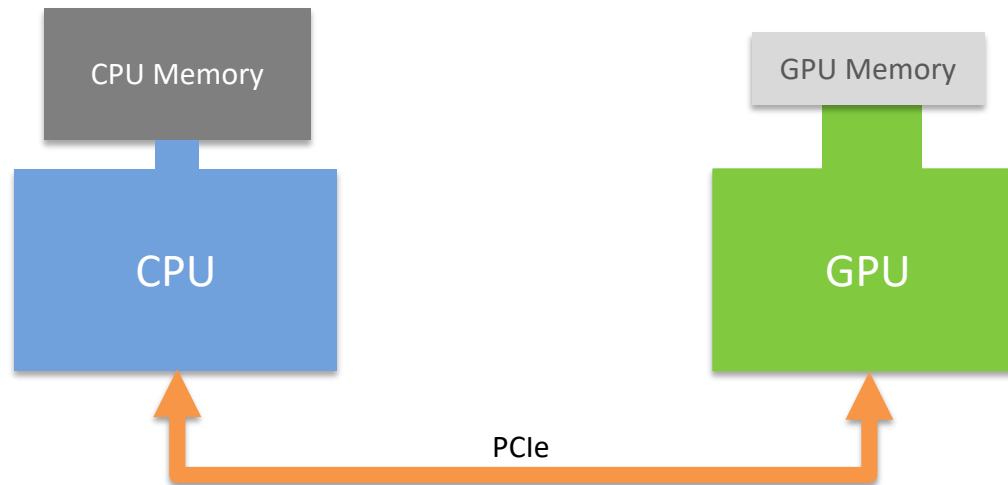
```
main:  
36, Generating implicit copyin(Temperature_last[:][:])  
    Generating implicit copyout(Temperature[1:1000][1:1000])  
37, Loop is parallelizable  
38, Loop is parallelizable  
    Accelerator kernel generated  
    Generating Tesla code  
37, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */  
38, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */  
44, Generating implicit copyin(Temperature[1:1000][1:1000])  
    Generating implicit copy(Temperature_last[1:1000][1:1000])  
45, Loop is parallelizable  
46, Loop is parallelizable  
    Accelerator kernel generated  
    Generating Tesla code  
45, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */  
46, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */  
47, Generating implicit reduction(max:dt)  
$./laplace_acc
```

Slower than serial and much slower than OpenMP version!!

Running with profiling on

```
main  NVIDIA  devicenum=0
time(us): 2,600,570
36: compute region reached 600 times
    38: kernel launched 600 times
        grid: [32x250]  block: [32x4]
        device time(us): total=66,868 max=114 min=111 avg=111
        elapsed time(us): total=82,196 max=499 min=134 avg=136
36: data region reached 1200 times
    36: data copyin transfers: 600
        device time(us): total=478,660 max=841 min=793 avg=797
    42: data copyout transfers: 600
        device time(us): total=479,890 max=854 min=795 avg=799
44: compute region reached 600 times
    44: data copyin transfers: 600
        device time(us): total=5,470 max=19 min=8 avg=9
    46: kernel launched 600 times
        grid: [32x250]  block: [32x4]
        device time(us): total=93,708 max=163 min=140 avg=156
        elapsed time(us): total=122,285 max=367 min=185 avg=203
    46: reduction kernel launched 600 times
        grid: [1]  block: [256]
        device time(us): total=13,817 max=32 min=23 avg=23
        elapsed time(us): total=24,132 max=85 min=38 avg=40
    46: data copyout transfers: 600
        device time(us): total=9,508 max=46 min=13 avg=15
44: data region reached 1200 times
    44: data copyin transfers: 1200
        device time(us): total=966,408 max=843 min=796 avg=805
    51: data copyout transfers: 600
        device time(us): total=486,241 max=852 min=802 avg=810
```

Data movement



Data management

C

```
#pragma acc data [clause]
{
    ...
}
```

Fortran

```
!$acc data [clause]
...
!$acc end data
```


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.
 - Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.
- `copyin(list)`
 - Allocates memory on GPU and copies data from host to GPU when entering region.
 - Principal use: Think of this like an array that you would use as just an input to a subroutine.
- `copyout(list)`
 - Allocates memory on GPU and copies data to the host when exiting region.
 - Principal use: A result that isn't overwriting the input data structure.
- `create(list)`
 - Allocates memory on GPU but does not copy.
 - Principal use: Temporary arrays.

Array shaping

- Compilers sometimes cannot determine the size of arrays, so we must explicitly specify using data clauses
 - The compiler will let you know if you need to do this.
 - Sometimes, you do it for your own efficiency reasons.
- C

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

```
!$acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```
- Fortran uses start:end and C uses start:length

update directive

#pragma acc update host(a)

host: copies data from the device (GPU) to the host (CPU)

device: copies from the host to the device

- Explicitly transfers data between the host and the device
 - Useful when you want to update data in the middle of a data region
- Clauses:

OpenACC solution with data clause

```
#pragma acc data copy(Temperature_last) create(Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++)
        for(j = 1; j <= COLUMNS; j++)
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] +
                Temperature_last[i-1][j] + Temperature_last[i][j+1] +
                Temperature_last[i][j-1]);
    dt = 0.0;
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++)
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature)
        track_progress(iteration);
    }
    iteration++;
}
```

Conclusion

- OpenACC is an easy and quick way to make use of GPUs to run your code faster
- Data dependency and data movement are two important challenges to get a good speedup from GPU-based code

References

- References and useful links:
 - <http://www.openacc.org>
 - https://www.psc.edu/images/xsedetraining/OpenACC_No_v2017/OpenACC_Introduction_To_OpenACC.PDF
 - <http://courses.cms.caltech.edu/cs101gpu/>
 - <http://on-demand.gputechconf.com/gtc/2017/presentation/S7133-jiri-kraus-multi-gpu-programming.PDF>