Introduction to OpenACC

Alexander B. Pacheco

User Services Consultant
LSU HPC & LONI
sys-help@loni.org

HPC Training Spring 2014
Louisiana State University
Baton Rouge
March 26, 2014

- OpenACC Application Program Interface describes a collection of compiler directive to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator.
- provides portability across operating systems, host CPUs and accelerators

The Standard for GPU Directives

Simple: Directive are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforwards and portable across parallel and multi-core processors

Powerful: GPU directives allow complete access to the massive parallel power of a GPU

High Level

- Compiler directives to specify parallel regions in C & Fortran
  - Offload parallel regions
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogenous programs
  - Without explicit accelerator intialization
  - Without explicit data or program transfers between host and accelerator

High Level · · · with low-level access

- Programming model allows programmers to start simple
- Compiler gives additional guidance
  - Loop mappings, data location and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc

- Directives are easy and powerful.
- Avoid restructuring of existing code for production applications.
- Focus on expressing parallelism.

OpenACC is not GPU Programming

OpenACC is Expressing Parallelism in your code

- Did you attend/review the trainings on C/C++ or Modern Fortran?
- Recall the following three exercises:
  1. SAXPY: Generalized vector addition
  2. Matrix Multiplication
  3. Calculate pi by Numerical Integration

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$

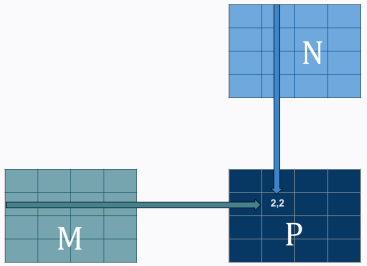
- Write a SAXPY code to multiply a vector with a scalar.

Algorithm 1 Pseudo Code for SAXPY

```
program SAXPY
    n ← some large number
    x(1 : n) ← some number say, 1
    y(1 : n) ← some other number say, 2
    a ← some other number ,say, 3
    do i ← 1 · · · n
        y_i ← y_i + a ∗ x_i
    end do
end program SAXPY
```

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Algorithm 2 Pseudo Code for MATMUL

program MATMUL

$m, n \leftarrow$ some large number ≤ 1000

Define a_{mn}, b_{nm}, c_{mm}

$a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$

do $i \leftarrow 1 \cdots m$

do $j \leftarrow 1 \cdots m$

$c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$

end do

end do

end program MATMUL

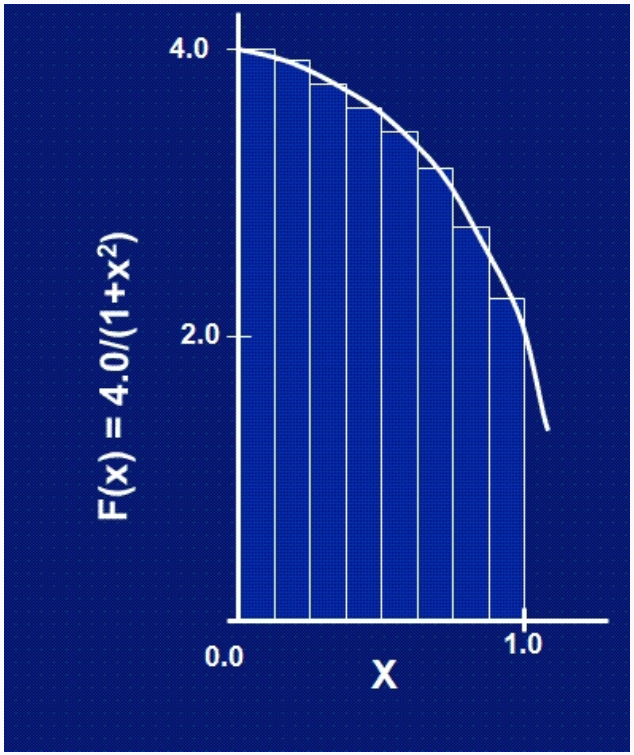
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A "hands-on" introduction to OpenMP, SC09



Algorithm 3 Pseudo Code for Calculating Pi

program CALCULATE_PI

$step \leftarrow 1/n$

$sum \leftarrow 0$

do $i \leftarrow 0 \cdots n$

$x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$

end do

$pi \leftarrow sum * step$

end program

Serial Code

```
program saxpy

  implicit none
  integer, parameter :: dp = selected_real_kind(15)
  integer, parameter :: ip = selected_int_kind(15)
  integer(ip) :: i,n
  real(dp),dimension(:),allocatable :: x, y
  real(dp) :: a,start_time, end_time

  n=5000000
  allocate(x(n),y(n))

  x = 1.0d0
  y = 2.0d0
  a = 2.0

  call cpu_time(start_time)
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  call cpu_time(end_time)
  deallocate(x,y)

  print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy
```

OpenMP Code

```
program saxpy

  implicit none
  integer, parameter :: dp = selected_real_kind(15)
  integer, parameter :: ip = selected_int_kind(15)
  integer(ip) :: i,n
  real(dp),dimension(:),allocatable :: x, y
  real(dp) :: a,start_time, end_time

  n=500000000
  allocate(x(n),y(n))
  !$omp parallel sections
  !$omp section
  x = 1.0
  !$omp section
  y = 1.0
  !$omp end parallel sections
  a = 2.0

  call cpu_time(start_time)
  !$omp parallel do default(shared) private(i)
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  !$omp end parallel do
  call cpu_time(end_time)
  deallocate(x,y)

  print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy
```

OpenACC Code

```
program saxpy

  use omp_lib

  implicit none
  integer :: i,n
  real,dimension(:),allocatable :: x, y
  real :: a,start_time, end_time

  n=500000000
  allocate(x(n),y(n))
  a = 2.0
  !$acc data create(x,y) copyin(a)
  !$acc parallel
  x(:) = 1.0
  !$acc end parallel
  !$acc parallel
  y(:) = 1.0
  !$acc end parallel

  start_time = omp_get_wtime()
  !$acc parallel loop
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  !$acc end parallel loop
  end_time = omp_get_wtime()
  !$acc end data
  deallocate(x,y)

  print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, 'in secs'

end program saxpy
```

CUDA Fortran Code

```
module mymodule
contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  integer, parameter :: n = 100000000
  real, device :: x_d(n), y_d(n)
  real, device :: a_d
  real :: start_time, end_time

  x_d = 1.0
  y_d = 2.0
  a_d = 2.0

  call cpu_time(start_time)
  call saxpy<<<4096, 256>>>(n, a, x_d, y_d)
  call cpu_time(end_time)

  print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, 'in secs'
end program main
```

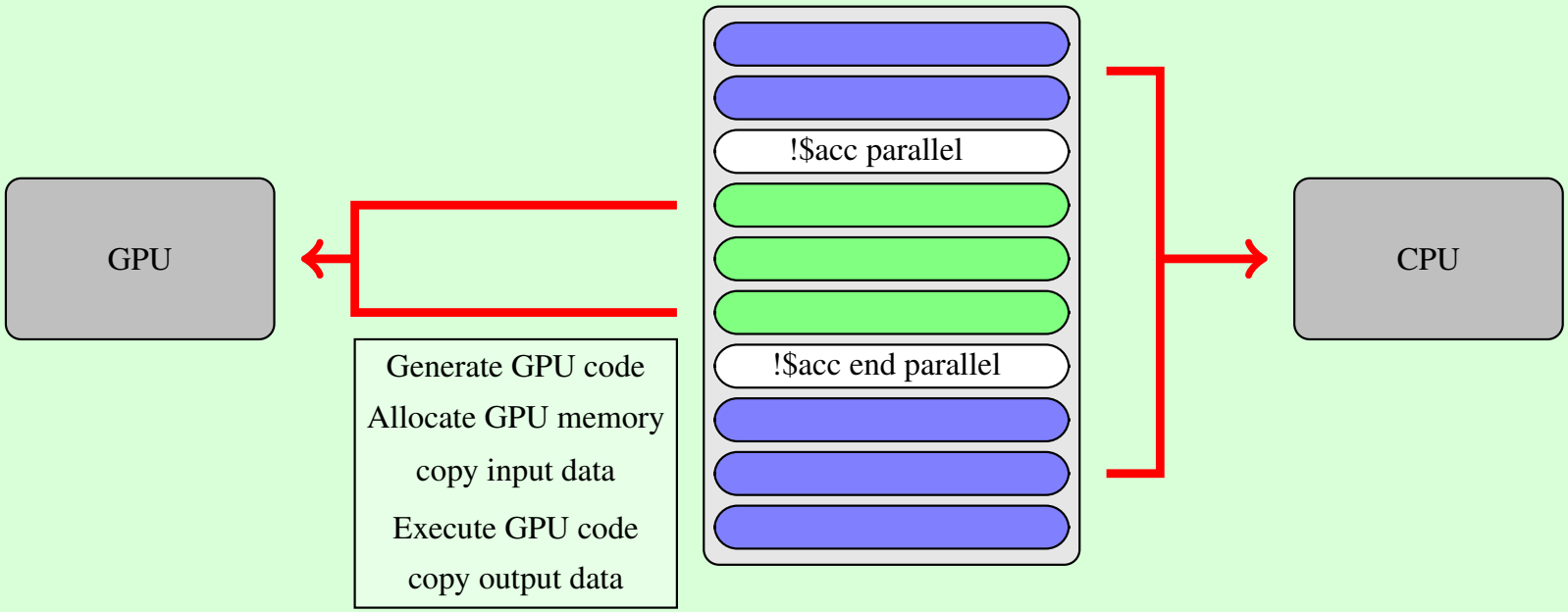
Compile

```
[apacheco@mike1 2013-LONI]$ pgf90 -o saxpy saxpy.f90
[apacheco@mike1 2013-LONI]$ pgf90 -mp -o saxpy_omp saxpy_omp.f90
[apacheco@mike1 2013-LONI]$ pgf90 -acc -ta=nvidia -o saxpy_acc saxpy_acc.f90
[apacheco@mike1 2013-LONI]$ pgf90 -o saxpy_cuda saxpy.cuf
```

Speed Up

Algorithm	Device	Time (s)	Speedup
Serial	Xeon E5-2670	0.986609	1
OpenMP (8 threads)	Xeon E5-2670	0.241465	4.1x
OpenACC	M2090	0.059418	16.6x
CUDA	M2090	0.005205	189.5x

- Application code runs on the CPU (sequential, shared or distributed memory)
- OpenACC directives indicate that the following block of compute intensive code needs to be offloaded to the GPU or accelerator.



- Program directives
  - Syntax
    - C/C++: `#pragma acc <directive> [clause]`
    - Fortran: `!$acc <directive> [clause]`
  - Regions
  - Loops
  - Synchronization
  - Data Structure
  - . . .
- Runtime library routines

- `if (condition)`
- `async (expression)`
- data management clauses
  - `copy(···),copyin(···), copyout(···)`
  - `create(···), present(···)`
  - `present_or_copy{,in,out}(···)` or `pcopy{,in,out}(···)`
  - `present_or_create(···)` or `pcreate(···)`
- `reduction(operator:list)`

- System setup routines
  - `acc_init(acc_device_nvidia)`
  - `acc_set_device_type(acc_device_nvidia)`
  - `acc_set_device_num(acc_device_nvidia)`
- Synchronization routines
  - `acc_async_wait(int)`
  - `acc_async_wait_all()`

C: `#pragma acc kernels [clause]`

Fortran `!$acc kernels [clause]`

- The kernels directive expresses that a region may contain parallelism and the compiler determines what can be safely parallelized.
- The compiler breaks code in the kernel region into a sequence of kernels for execution on the accelerator device.
- For the codes on the right, the compiler identifies 2 parallel loops and generates 2 kernels.
- What is a kernel? A function that runs in parallel on the GPU.
- When a program encounters a kernels contruct, it will launch a sequence of kernels in order on the device.

```
!$acc kernels
do i = 1, n
   x(i) = 1.0
   y(i) = 2.0
end do

do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end kernels
```

```
#pragma acc kernels
{
  for (i = 0; i < n; i++){
    x[i] = 1.0 ;
    y[i] = 2.0 ;
  }

  for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
  }
}
```

- The **parallel** directive identifies a block of code as having parallelism.
- Compiler generates a parallel kernel for that loop.

C: `#pragma acc parallel [clauses]`

Fortran: `!$acc parallel [clauses]`

```
!$acc parallel
do i = 1, n
   x(i) = 1.0
   y(i) = 2.0
end do

do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end parallel
```

```
#pragma acc parallel
{
  for (i = 0; i < n; i++){
    x[i] = 1.0 ;
    y[i] = 2.0 ;
  }

  for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
  }
}
```

- Loops are the most likely targets for Parallelizing.
- The Loop directive is used within a parallel or kernels directive identifying a loop that can be executed on the accelerator device.

C: `#pragma acc loop [clauses]`

Fortran: `!$acc loop [clauses]`

- The loop directive can be combined with the enclosing parallel or kernels

C:

`#pragma acc kernels loop [clauses]`

Fortran: `!$acc parallel loop [clauses]`

- The loop directive clauses can be used to optimize the code. This however requires knowledge of the accelerator device.

Clauses: gang, worker, vector, num_gangs, num_workers

```
!$acc loop
do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end loop


#pragma acc loop
for (i = 0; i < n; i++){
  y[i] = a*x[i] + y[i];
}
```

PARALLEL

- Requires analysis by programmer to ensure safe parallelism.
- Straightforward path from OpenMP

KERNELS

- Compiler performs parallel analysis and parallelizes what it believes is safe.
- Can cover larger area of code with single directive.

Both approaches are equally valid and can perform equally well.

```fortran
program saxpy

  use omp_lib

  implicit none
  integer :: i,n
  real,dimension(:),allocatable :: x, y
  real :: a,start_time, end_time

  n=500000000
  allocate(x(n),y(n))
  a = 2.0
  x(:) = 1.0
  y(:) = 1.0

  start_time = omp_get_wtime()
  !$acc parallel loop
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  !$acc end parallel loop
  end_time = omp_get_wtime()
  deallocate(x,y)

  print '(a,f15.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy
```

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>

int main() {
  long long int i, n=500000000;
  float a=2.0;
  float x[n];
  float y[n];
  double start_time, end_time;

  a = 2.0;
  for (i = 0; i < n; i++){
    x[i] = 1.0;
    y[i] = 2.0;
  }

  start_time = omp_get_wtime();
#pragma acc kernels loop
  {
    for (i = 0; i < n; i++){
      y[i] = a*x[i] + y[i];
    }
  }
  end_time = omp_get_wtime();

  printf ("SAXPY Time: %f\n", end_time - start_time);

}
```

- C:
  `pgcc -acc [-Minfo=accel] [-ta=nvidia] -o saxpyc_acc saxpy_acc.c`
- Fortran 90:
  `pgf90 -acc [-Minfo=accel] [-ta=nvidia] -o saxpyf_acc saxpy_acc.f90`

Compiler Output

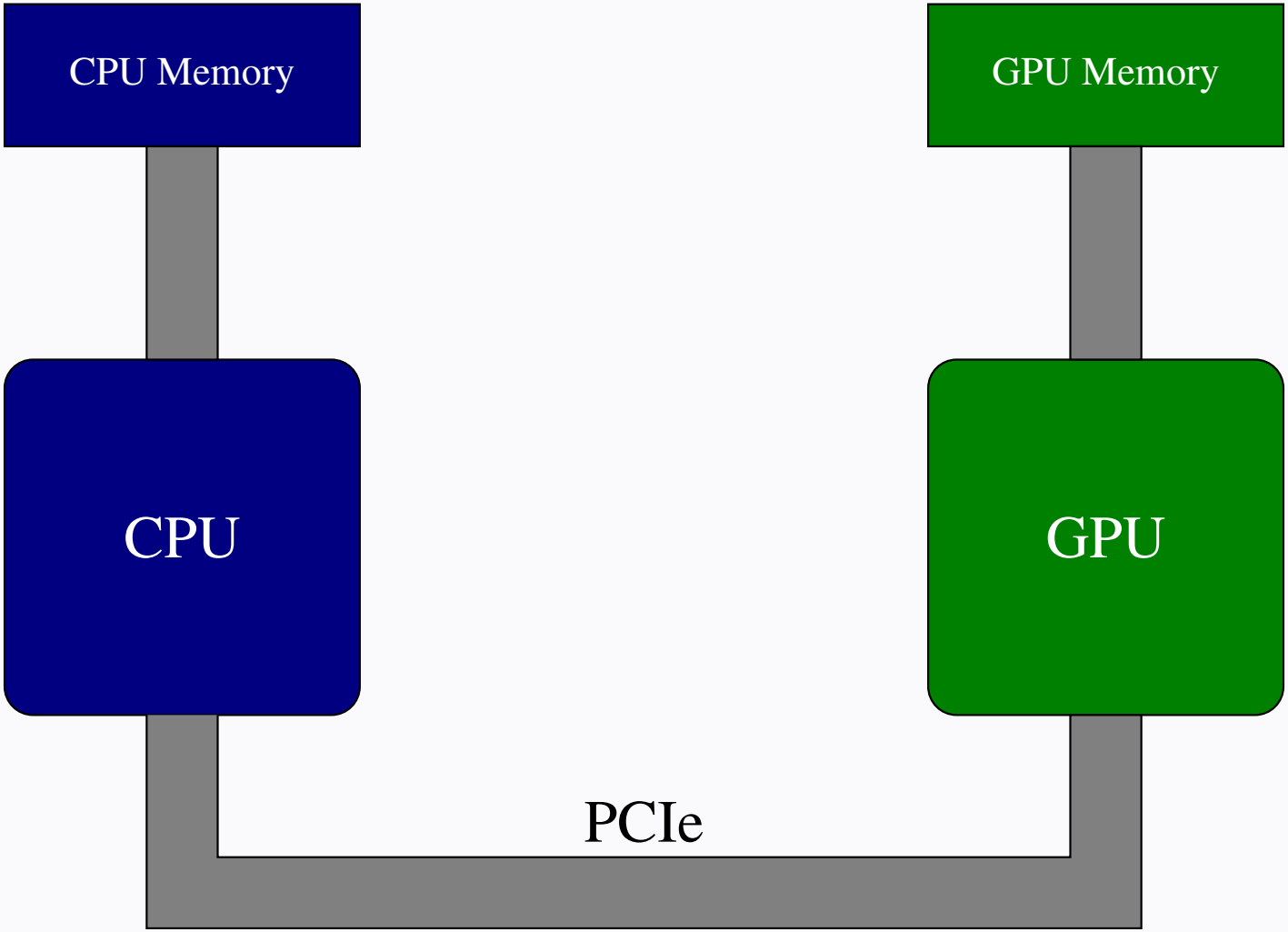
```
[apacheco@mike1 nodataregion]$ pgcc -acc -ta=nvidia -Minfo=accel  -o saxpyc_acc saxpy_acc.c
main:
     19, Generating present_or_copyin(x[0:500000000])
         Generating present_or_copy(y[0:500000000])
         Generating NVIDIA code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     21, Loop is parallelizable
         Accelerator kernel generated
         21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
[apacheco@mike1 nodataregion]$ pgf90 -acc -ta=nvidia -Minfo=accel -o saxpyf_acc saxpy_acc.f90
saxpy:
     17, Accelerator kernel generated
         18, !$acc loop gang, vector(256) ! blockidx%x threadidx%x
     17, Generating present_or_copy(y(1:500000000))
         Generating present_or_copyin(x(1:500000000))
         Generating NVIDIA code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
[apacheco@mike1 nodataregion]$
```

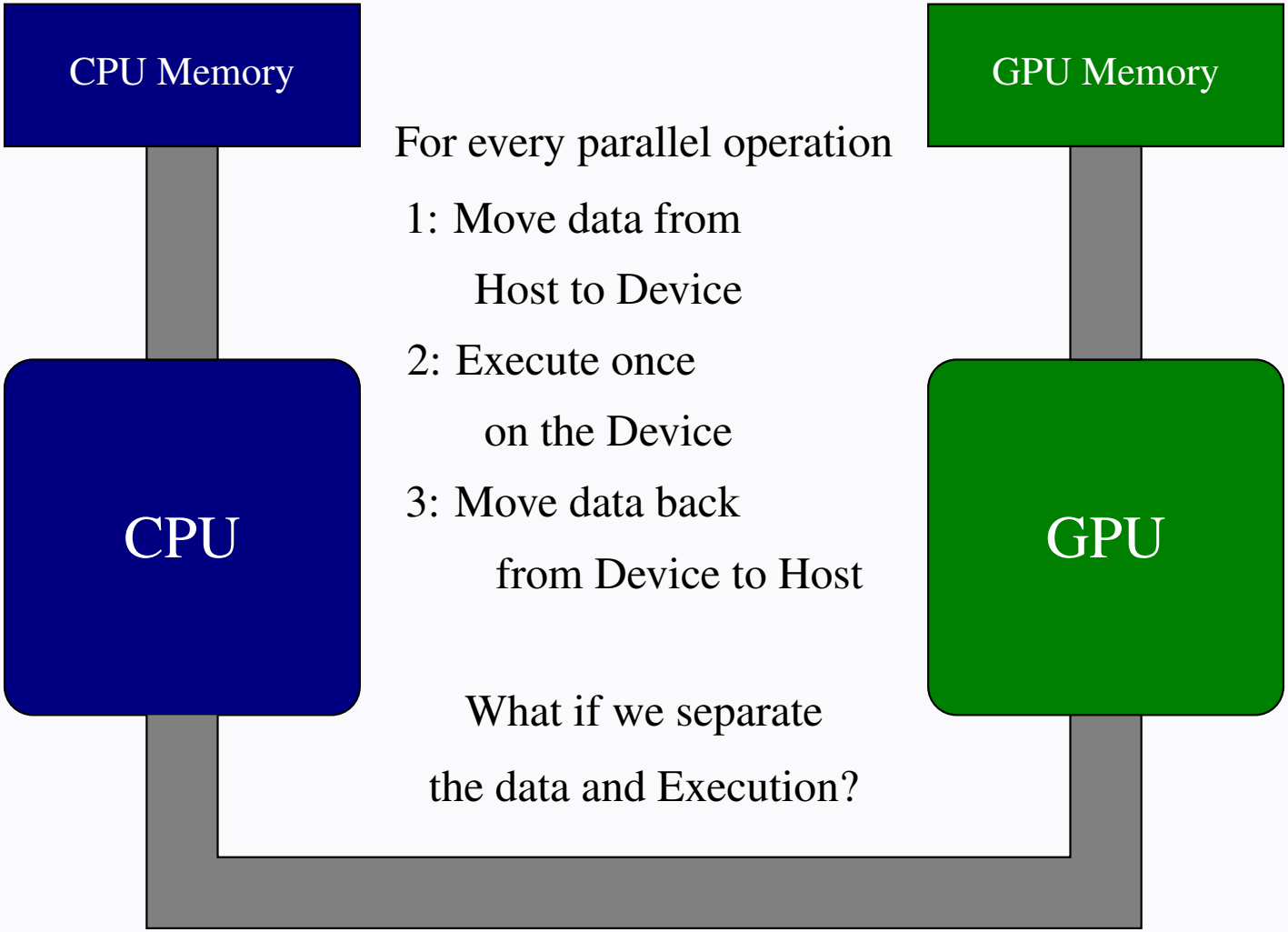
- The PGI compiler provides automatic instrumentation when PGI_ACC_TIME=1 at runtime

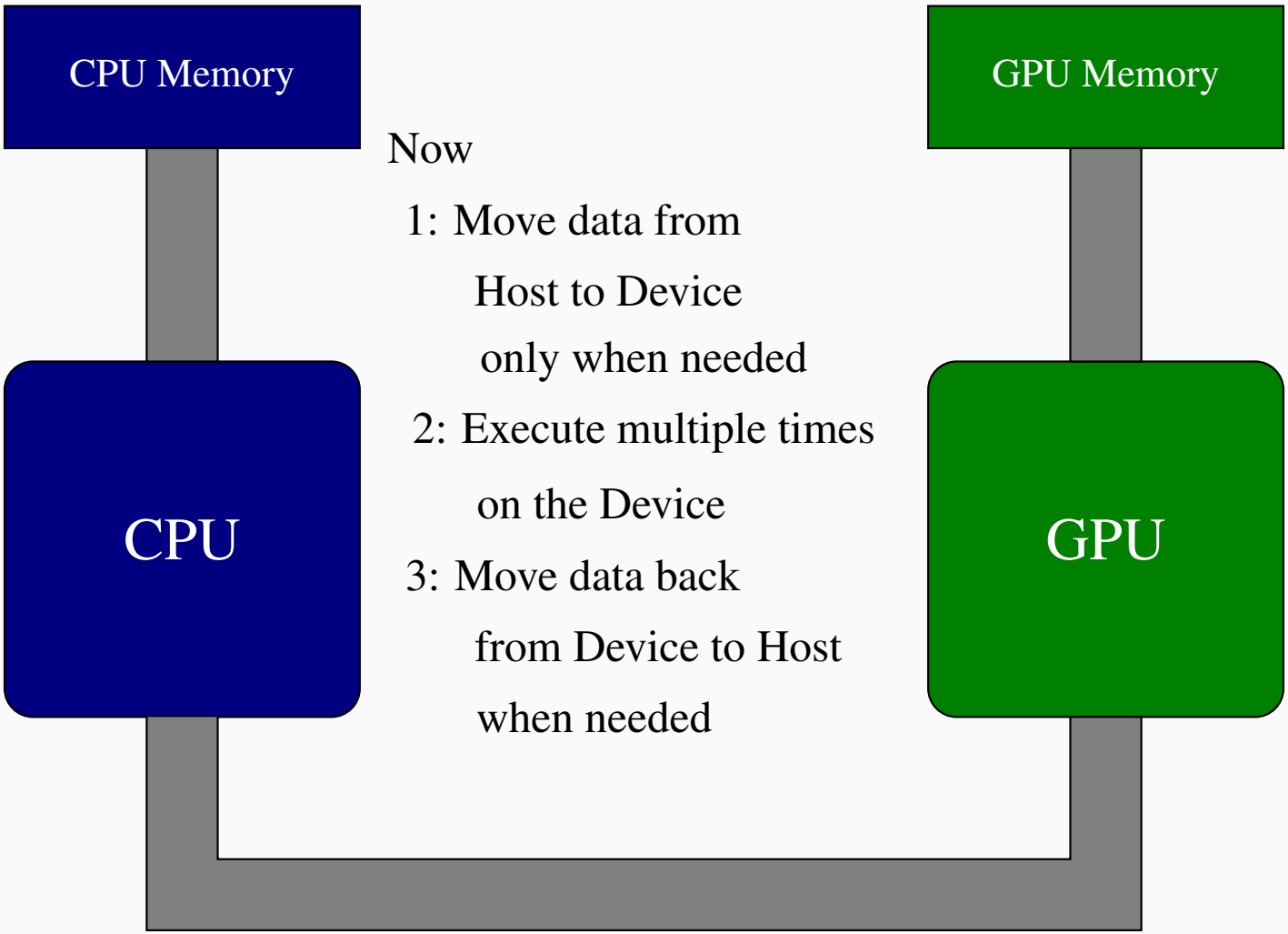
```
[apacheco@mike407 nodataregion]$ PGI_ACC_TIME=1 ./saxpyc_acc
SAXPY Time: 6.369176

Accelerator Kernel Timing data
/ddnB/work/apacheco/2013-LONI/openmp/saxpy/nodataregion/saxpy_acc.c
  main  NVIDIA  devicenum=0
    time(us): 1,029,419
    19: compute region reached 1 time
        19: data copyin reached 2 times
             device time(us): total=667,515 max=339,175 min=328,340 avg=333,757
        21: kernel launched 1 time
            grid: [65535]  block: [128]
             device time(us): total=57,999 max=57,999 min=57,999 avg=57,999
            elapsed time(us): total=58,014 max=58,014 min=58,014 avg=58,014
        25: data copyout reached 1 time
             device time(us): total=303,905 max=303,905 min=303,905 avg=303,905
[apacheco@mike407 nodataregion]$ PGI_ACC_TIME=1 ./saxpyf_acc
SAXPY Time:        6.488910

Accelerator Kernel Timing data
/ddnB/work/apacheco/2013-LONI/openmp/saxpy/nodataregion/saxpy_acc.f90
  saxpy  NVIDIA  devicenum=0
    time(us): 1,018,988
    17: compute region reached 1 time
        17: data copyin reached 2 times
             device time(us): total=655,958 max=327,991 min=327,967 avg=327,979
        17: kernel launched 1 time
            grid: [65535]  block: [256]
             device time(us): total=59,148 max=59,148 min=59,148 avg=59,148
            elapsed time(us): total=59,165 max=59,165 min=59,165 avg=59,165
        21: data copyout reached 1 time
             device time(us): total=303,882 max=303,882 min=303,882 avg=303,882
```

Execution	C		Fortran	
	Time	SpeedUp	Time	Speedup
Serial	0.511232		0.969819	
OpenMP (8 Threads)	0.180301	2.84	0.237585	4.08
OpenACC (M2090)	9.211521	0.06	9.188178	0.11

- What's going with OpenACC code?
- Why even bother with OpenACC if performance is so bad?


CPU Memory
GPU Memory
CPU
GPU
PCIe


CPU Memory
GPU Memory
For every parallel operation
1: Move data from
Host to Device
2: Execute once
on the Device
3: Move data back
from Device to Host
What if we separate
the data and Execution?
CPU
GPU


CPU Memory
GPU Memory
Now
1: Move data from
Host to Device
only when needed
2: Execute multiple times
on the Device
3: Move data back
from Device to Host
when needed
CPU
GPU

- 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

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

Arrays used within the data region will remain on the GPU until the end of the data region.

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.

- Other clauses: present_or_copy[in|out], present_or_create, deviceptr.

- Compiler sometime cannot determine size of arrays
  - Must specify explicitly using the data clauses and array "shape"

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))
```

- Note: data clauses can be used on data, parallel or kernels

- Used to update existing data after it has changed in its corresponding copy (e.g. upate device copy after host copy changes).
- Move data from GPU to host, or host to GPU.
- Data movement can be conditional and asynchronous.
- Fortran

```
!$acc update [clause ···]
```

- C

```
#pragma acc update [clause ···]
```

- Clause
  - `host(list)`
  - `device(list)`
  - `if(expression)`
  - `async(expression)`

```fortran
program saxpy

  use omp_lib

  implicit none
  integer :: i,n
  real,dimension(:),allocatable :: x, y
  real :: a,start_time, end_time

  n=500000000
  allocate(x(n),y(n))
  a = 2.0
  !$acc data create(x,y) copyin(a)
  !$acc parallel
  x(:) = 1.0
  !$acc end parallel
  !$acc parallel
  y(:) = 1.0
  !$acc end parallel

  start_time = omp_get_wtime()
  !$acc parallel loop
  do i = 1, n
     y(i) = y(i) + a * x(i)
  end do
  !$acc end parallel loop
  end_time = omp_get_wtime()
  !$acc end data
  deallocate(x,y)

  print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, '
          in secs'

end program saxpy
```

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>

int main() {
  long long int i, n=500000000;
  float a=2.0;
  float x[n];
  float y[n];
  double start_time, end_time;

  a = 2.0;
#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;
      }

      start_time = omp_get_wtime();
#pragma acc kernels loop
      {
  for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
  }
      }
      end_time = omp_get_wtime();
    }

  printf ("SAXPY Time: %f\n", end_time - start_time);

}
```

Execution	C		Fortran	
	Time	SpeedUp	Time	Speedup
Serial	0.510000		0.986609	
OpenMP (8 Threads)	0.179959	2.83	0.241465	4.09
OpenACC (M2090)	0.058131	8.77	0.059418	16.61

C

Execution	Time	SpeedUp	GFlops/s
Serial	6.227		0.964
OpenMP (8 Threads)	0.823	7.566	7.290
OpenMP (16 Threads)	0.445	13.993	13.493
OpenACC	0.188	33.122	31.917

Fortran

Execution	Time	SpeedUp	GFlops/s
Serial	7.112		0.844
OpenMP (8 Threads)	0.931	7.639	6.445
OpenMP (16 Threads)	0.494	14.397	12.146
OpenACC	0.214	33.234	28.037

- Reduction clause is allowed on *parallel* and *loop* constructs

Fortran

```
!$acc parallel reduction(operation: var)
  structured block with reduction on var
!$acc end parallel
```

C

```
#pragma acc kernels reduction(operation: var) {
  structured block with reduction on var
}
```

Fortran		
Execution	Time	SpeedUp
Serial	133.782	1
OpenMP (8 Threads)	17.303	7.73
OpenACC	0.149	897.87
C		
Execution	Time	SpeedUp
Serial	134.214	1
OpenMP (8 Threads)	17.3379	7.74
OpenACC	0.151	888.83

- OpenACC gives us more detailed control over parallelization
  - Via **gang**, **worker** and **vector** clauses
- By understanding more about specific GPU on which you're running, using these clauses may allow better performance.
- By understanding bottlenecks in the code via profiling, we can reorganize the code for even better performance.

- (Nested) for/do loops are best for parallelization
- Large loop counts are best
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
  - Use subscripted arrays, rather than pointer-indexed arrays
- Data regions should avoid wasted bandwidth
  - Can use directive to explicitly control sizes
- Various annoying things can interfere with accelerated regions.
  - Function calls within accelerated region must be inlineable.
  - No IO

- High-level. No involvement of OpenCL, CUDA, etc
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports GPU accelreators and co-processors from multiple vendors, current and future versions.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.

Lecture derived from slides and presentations by

- Michael Wolfe, PGI
- Jeff Larkin, NVIDIA
- John Urbanic, PSC

Search for OpenACC presentations at the GPU Technology Conference Website for further study
`http://www.gputechconf.com/gtcnew/on-demand-gtc.php`

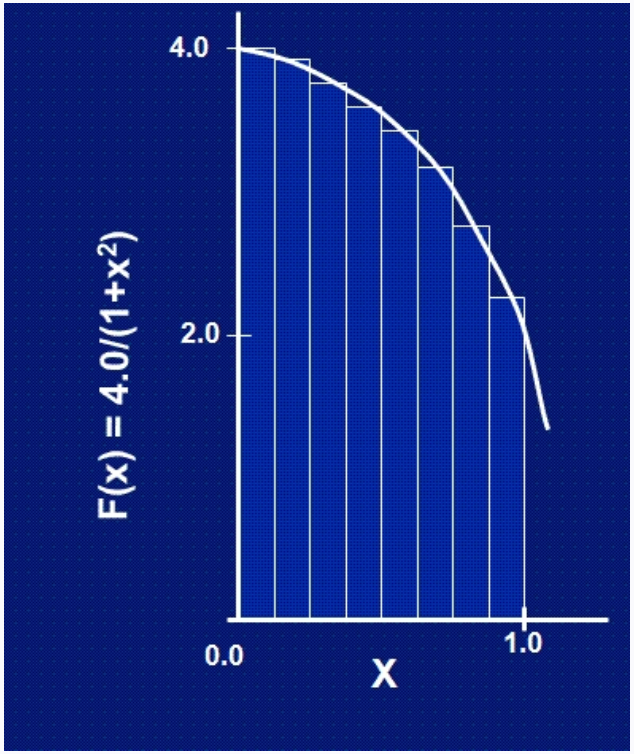
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A "hands-on" introduction to OpenMP, SC09



Algorithm 1 Pseudo Code for Calculating Pi

program CALCULATE_PI

$step \leftarrow 1/n$

$sum \leftarrow 0$

do $i \leftarrow 0 \cdots n$

$x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$

end do

$pi \leftarrow sum * step$

end program

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$

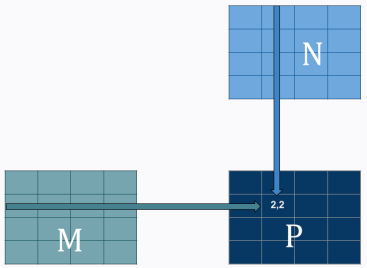
- Write a SAXPY code to multiply a vector with a scalar.

Algorithm 2 Pseudo Code for SAXPY

```
program SAXPY
    n ← some large number
    x(1 : n) ← some number say, 1
    y(1 : n) ← some other number say, 2
    a ← some other number ,say, 3
    do i ← 1 · · · n
        y_i ← y_i + a ∗ x_i
    end do
end program SAXPY
```

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Algorithm 3 Pseudo Code for MATMUL

program MATMUL
  $m, n \leftarrow$ some large number ≤ 1000
  Define a_{mn}, b_{nm}, c_{mm}
  $a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$
  do $i \leftarrow 1 \cdots m$
    do $j \leftarrow 1 \cdots m$
      $c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$
    end do
  end do
end program MATMUL