

# GPU Computing with OpenACC Directives

Gundolf Haase, Uni Graz; (orig. Mark Harris, NVIDIA)

# A Very Simple Exercise: SAXPY

## *SAXPY in C*

```
void saxpy(int n,  
           float a,  
           float *x,  
           float *restrict y)  
{  
    #pragma acc kernels  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}  
  
...  
// Perform SAXPY on 1M elements  
saxpy(1<<20, 2.0, x, y);  
...
```

## *SAXPY in Fortran*

```
subroutine saxpy(n, a, x, y)  
    real :: x(:), y(:), a  
    integer :: n, i  
    $!acc kernels  
    do i=1,n  
        y(i) = a*x(i)+y(i)  
    enddo  
    $!acc end kernels  
end subroutine saxpy  
  
...  
$ Perform SAXPY on 1M elements  
call saxpy(2**20, 2.0, x_d, y_d)  
...
```

# Directive Syntax

## Fortran

```
!$acc directive [clause [,] clause] ...]
```

Often paired with a matching end directive surrounding a structured code block

```
!$acc end directive
```

## C

```
#pragma acc directive [clause [,] clause] ...]
```

Often followed by a structured code block

# kernels: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```
!$acc kernels
```

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

} kernel 1

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

} kernel 2

```
!$acc end kernels
```

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

# Kernels Construct

## Fortran

```
!$acc kernels [clause ...]  
    structured block  
!$acc end kernels
```

## C

```
#pragma acc kernels [clause ...]  
    { structured block }
```

## Clauses

```
if( condition )  
async( expression )
```

Also, any data clause  
(more later)

# C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler

Applied to a pointer, e.g.

```
float *restrict ptr
```

Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”\*

- Limits the effects of pointer aliasing
- OpenACC compilers often require `restrict` to determine independence
  - Otherwise the compiler can’t parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined

# Complete SAXPY example code

- Trivial first example
  - Apply a loop directive
  - Learn compiler commands

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

```
void saxpy(int n,  
          float a,  
          float *x,  
          float *restrict y)
```

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

\*restrict:  
“I promise y does not  
alias x”

```
int main(int argc, char **argv)  
{  
  int N = 1<<20; // 1 million floats  
  
  if (argc > 1)  
    N = atoi(argv[1]);  
  
  float *x = (float*)malloc(N * sizeof(float));  
  float *y = (float*)malloc(N * sizeof(float));  
  
  for (int i = 0; i < N; ++i)  
  {  
    x[i] = 2.0f;  
    y[i] = 1.0f;  
  }  
  
  saxpy(N, 3.0f, x, y);  
  
  return 0;  
}
```

# Compile and run

- C:

```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
```

- Fortran:

```
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc  
saxpy.f90
```

- Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c  
saxpy:
```

```
8, Generating copyin(x[:n-1])
```

```
Generating copy(y[:n-1])
```

```
Generating compute capability 1.0 binary
```

```
Generating compute capability 2.0 binary
```

```
9, Loop is parallelizable
```

```
Accelerator kernel generated
```

```
9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
```

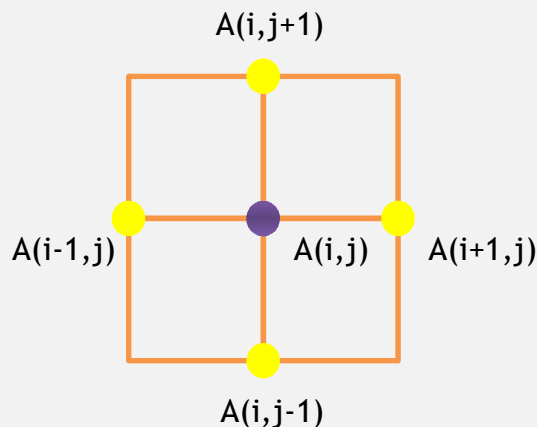
```
CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
```

```
CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```



# 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:



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

# Jacobi Iteration C Code

```
while ( error > tol && iter < iter_max )
{
    error=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]);

            error = max(error, 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++;
}
```



**Iterate until  
converged**



**Iterate across  
matrix elements**



**Calculate new  
value from  
neighbors**



**Compute max  
error for  
convergence**



**Swap  
input/output  
arrays**

# OpenMP C Code

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

    #pragma omp parallel for shared(m, n, Anew, A)
    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]);


            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma omp parallel for shared(m, n, Anew, A)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```



**Parallelize loop  
across CPU  
threads**



**Parallelize loop  
across CPU  
threads**

# GPU startup overhead

- If no other GPU process running, GPU driver may be swapped out
  - Linux specific
  - Starting it up can take 1-2 seconds
- Two options
  - Run `nvidia-smi` in persistence mode (requires root permissions)
  - Run `"nvidia-smi -q -l 30"` in the background
- If your running time is off by ~2 seconds from results in these slides, suspect this
  - Nvidia-smi should be running in persistent mode for these exercises

# First Attempt: OpenACC C

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

#pragma acc kernels
    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]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

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

    iter++;
}
```



**Execute GPU  
kernel for loop  
nest**



**Execute GPU  
kernel for loop  
nest**

# First Attempt: Compiler output (C)

```
pgcc -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
```

main:

```
57, Generating copyin(A[:4095][:4095])
    Generating copyout(Anew[1:4094][1:4094])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
58, Loop is parallelizable
60, Loop is parallelizable
    Accelerator kernel generated
    58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
    60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
        Cached references to size [18x18] block of 'A'
        CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
        CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
64, Max reduction generated for error
69, Generating copyout(A[1:4094][1:4094])
    Generating copyin(Anew[1:4094][1:4094])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
70, Loop is parallelizable
72, Loop is parallelizable
    Accelerator kernel generated
    70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
    72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
        CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
        CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy
```

# First Attempt: Performance

CPU: Intel Xeon X5680  
6 Cores @ 3.33GHz

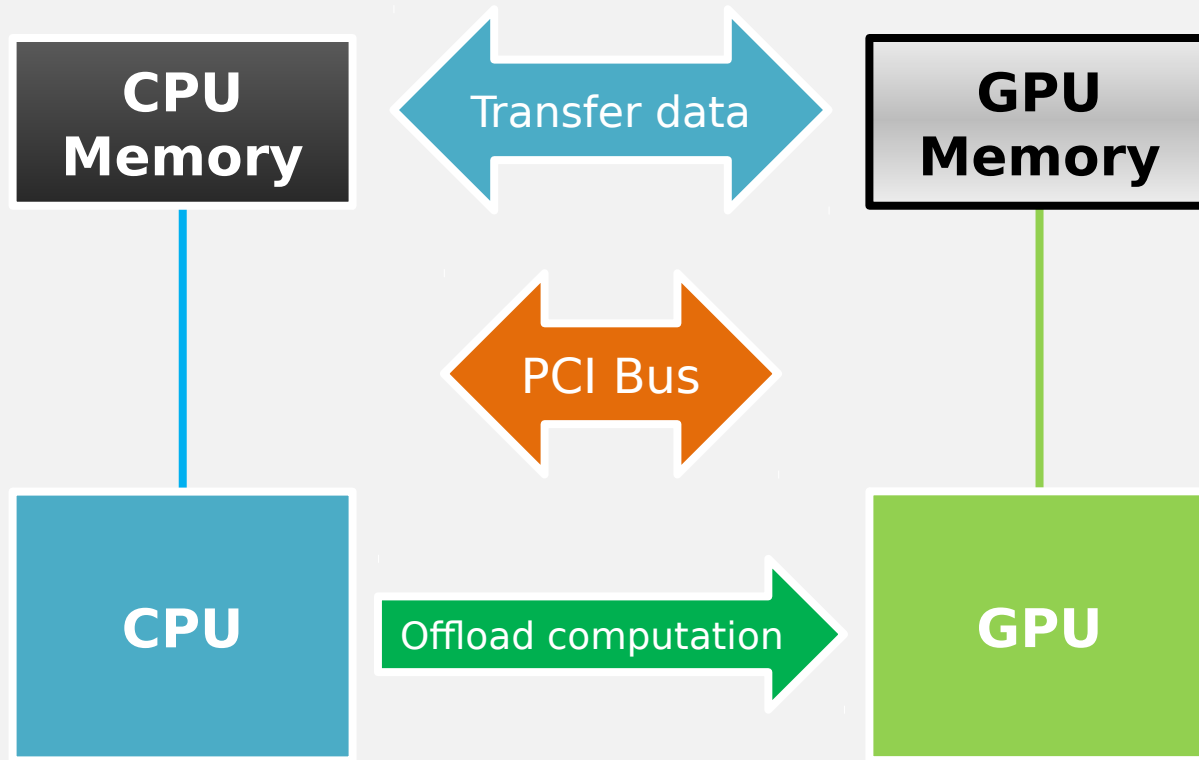
GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	162.16	0.24x FAIL

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

# Basic Concepts



For efficiency, decouple data movement and compute off-load



# Excessive Data Transfers

```
while ( error > tol && iter < iter_max )
```

```
{
```

```
  error=0.0;
```

A, Anew resident on host

Copy

#pragma acc kernels

A, Anew resident on accelerator

**These copies happen every iteration of the outer while loop!\***

```
  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]);
      error = max(error, abs(Anew[j][i] - A[j][i]));
    }
  }
```

Copy

A, Anew resident on accelerator

A, Anew resident on host

...

```
}
```

\*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!

# Data Management

# Data Construct

## Fortran

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

## C

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

## General Clauses

```
if( condition )  
async( expression )
```

Manage data movement. Data regions may be nested.

# 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.

`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.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

# Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

- C

```
#pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
```

- Fortran

```
!$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```

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

# Update Construct

## Fortran

```
!$acc update [clause ...]
```

## C

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

## Clauses

```
host( list )
```

```
device( list )
```

```
if( expression )
```

```
async( expression )
```

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU.

Data movement can be conditional, and asynchronous.

# Second Attempt: OpenACC C

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

    #pragma acc kernels
    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]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

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

    iter++;
}
```



**Copy A in at  
beginning of loop, out  
at end. Allocate  
Anew on accelerator**

# Second Attempt: Performance

CPU: Intel Xeon X5680  
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup	
CPU 1 OpenMP thread	69.80	--	
CPU 2 OpenMP threads	44.76	1.56x	
CPU 4 OpenMP threads	39.59	1.76x	
CPU 6 OpenMP threads	39.71	1.76x	Speedup vs. 1 CPU core
OpenACC GPU	13.65	2.9x	Speedup vs. 6 CPU cores

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU



# Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker, and vector clauses
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance

# Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: `restrict` keyword (C), `independent` clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.

# Tips and Tricks

- (PGI) Use time option to learn where time is being spent  
`-ta=nvidia,time`
- Eliminate pointer arithmetic
- Inline function calls in directives regions  
(PGI): `-inline` or `-inline,levels(<N>)`
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with `_OPENACC` macro

# Hints for data transfer

- Memory transfer requires shallow/flat data structures
- C-Arrays and C-structures (without pointers) are directly supported
- 
- Transfer of C++ classes,  
see Rob Farber's [tutorial](#)
- The [PGI Release notes Version 2014](#) §2.6.4 explains a clean solution for transferring classes, similar to a deep copy of data. New constructors as `todev()`, `fromdev()`, `updatehost()`, `updatedev()` have to be implemented.
-

# OpenACC Learning Resources

- OpenACC info, specification, FAQ, samples, and more
  - <http://openacc.org>
- 
- PGI OpenACC resources
  - <http://www.pgroup.com/resources/accel.htm>

# Complete OpenACC API

# Kernels Construct

## Fortran

```
!$acc kernels [clause ...]  
    structured block  
!$acc end kernels
```

## Clauses

```
if( condition )  
async( expression )
```

Also any data clause

## C

```
#pragma acc kernels [clause ...]  
    { structured block }
```

# Kernels Construct

Each loop executed as a separate kernel on the GPU.

```
!$acc kernels
```

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

} kernel 1

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

} kernel 2

```
!$acc end kernels
```



# Parallel Construct

## Fortran

```
!$acc parallel [clause ...]  
    structured block  
!$acc end parallel
```

C

```
#pragma acc parallel [clause ...]  
    { structured block }
```

## Clauses

```
if( condition )  
async( expression )  
num_gangs( expression )    private( list )  
num_workers( expression ) firstprivate( list )  
vector_length( expression ) reduction( operator:list )  
Also any data clause
```

# Parallel Clauses

`num_gangs ( expression )` Controls how many parallel gangs are created (CUDA `gridDim`).

`num_workers ( expression )` Controls how many workers are created in each gang (CUDA `blockDim`).

`vector_length ( list )` Controls vector length of each worker (SIMD execution).

`private( list )` A copy of each variable in list is allocated to each gang.

`firstprivate ( list )` private variables initialized from host.

`reduction( operator:list )` private variables combined across gangs.

# Loop Construct

## Fortran

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

## C

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

## Combined directives

```
!$acc parallel loop [clause ...]  
!$acc kernels loop [clause ...]
```

Detailed control of the parallel execution of the following loop.

# Loop Clauses

`collapse( n )` Applies directive to the following `n` nested loops.

`seq` Executes the loop sequentially on the GPU.

`private( list )` A copy of each variable in `list` is created for each iteration of the loop.

`reduction( operator:list )` `private` variables combined across iterations.

# Loop Clauses Inside parallel Region

`gang` Shares iterations across the gangs of the parallel region.

`worker` Shares iterations across the workers of the gang.

`vector` Execute the iterations in SIMD mode.

# Loop Clauses Inside kernels Region

`gang [( num_gangs )]` Shares iterations across across at  
`most num_gangs` gangs.

`worker [( num_workers )]` Shares iterations across at  
`most num_workers` of a single gang.

`vector [( vector_length )]` Execute the iterations in SIMD  
mode with maximum `vector_length`.

`independent` Specify that the loop iterations are independent.

# Other Syntax

# Other Directives

`cache` construct Cache data in software managed data cache (CUDA shared memory).

`host_data` construct Makes the address of device data available on the host.

`wait` directive Waits for asynchronous GPU activity to complete.

`declare` directive Specify that data is to allocated in device memory for the duration of an implicit data region created during the execution of a subprogram.



# Runtime Library Routines

## Fortran

```
use openacc  
#include "openacc_lib.h"
```

```
acc_get_num_devices  
acc_set_device_type  
acc_get_device_type  
acc_set_device_num  
acc_get_device_num  
acc_async_test  
acc_async_test_all
```

## C

```
#include "openacc.h"
```

```
acc_async_wait  
acc_async_wait_all  
acc_shutdown  
acc_on_device  
acc_malloc  
acc_free
```

# Environment and Conditional Compilation

`ACC_DEVICE device` Specifies which device type to connect to.

`ACC_DEVICE_NUM num` Specifies which device number to connect to.

`_OPENACC` Preprocessor directive for conditional compilation.  
Set to OpenACC version