

# MODULE FOUR: GPU PROGRAMMING

Dr. Volker Weinberg | LRZ

# MODULE OVERVIEW

## OpenACC Directives

- Multicore CPU vs GPU
- Introduction to GPU Data Management
- CUDA Managed Memory
- GPU Profiling with Nsight Systems

# CPU VS GPU

# CPU VS GPU

## Number of cores and parallelism

- Both are extremely popular parallel processors, but with different degrees of parallelism
- CPUs generally have a small number of very fast physical cores
- GPUs have thousands of simple cores able to achieve high performance in aggregate
- Both require parallelism to be fully utilized, but GPUs require much more

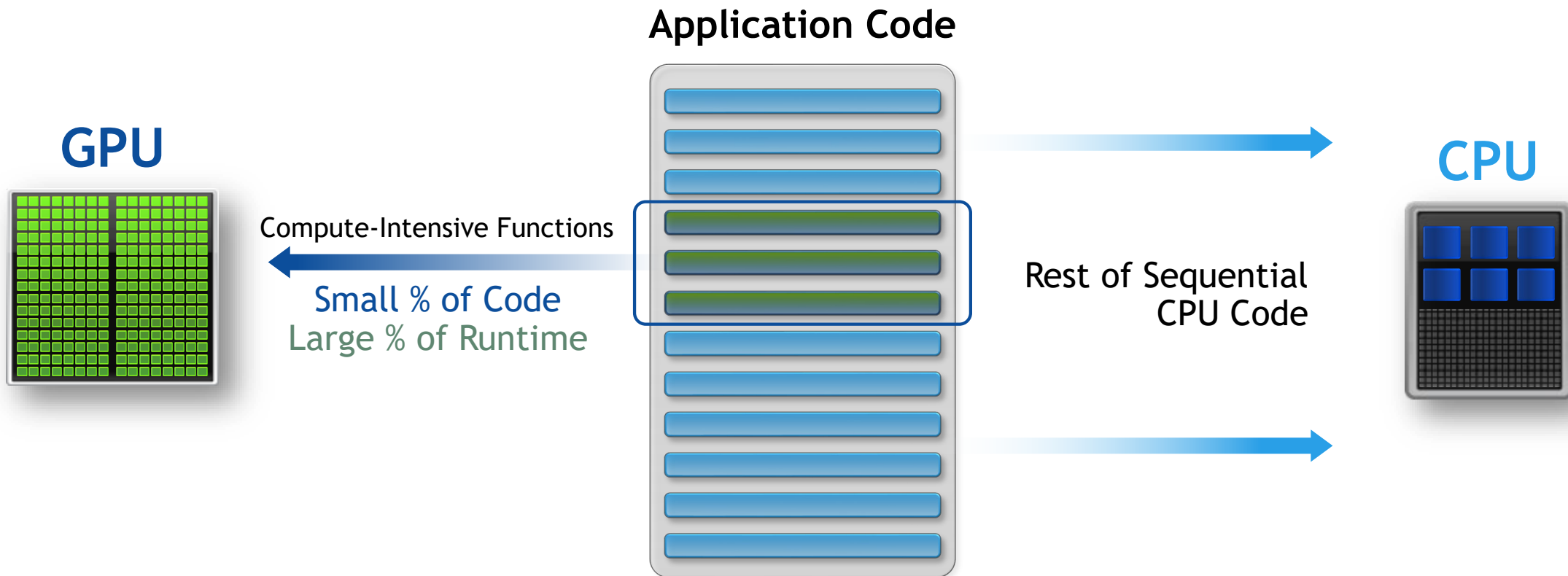


CPU  
MULTIPLE CORES



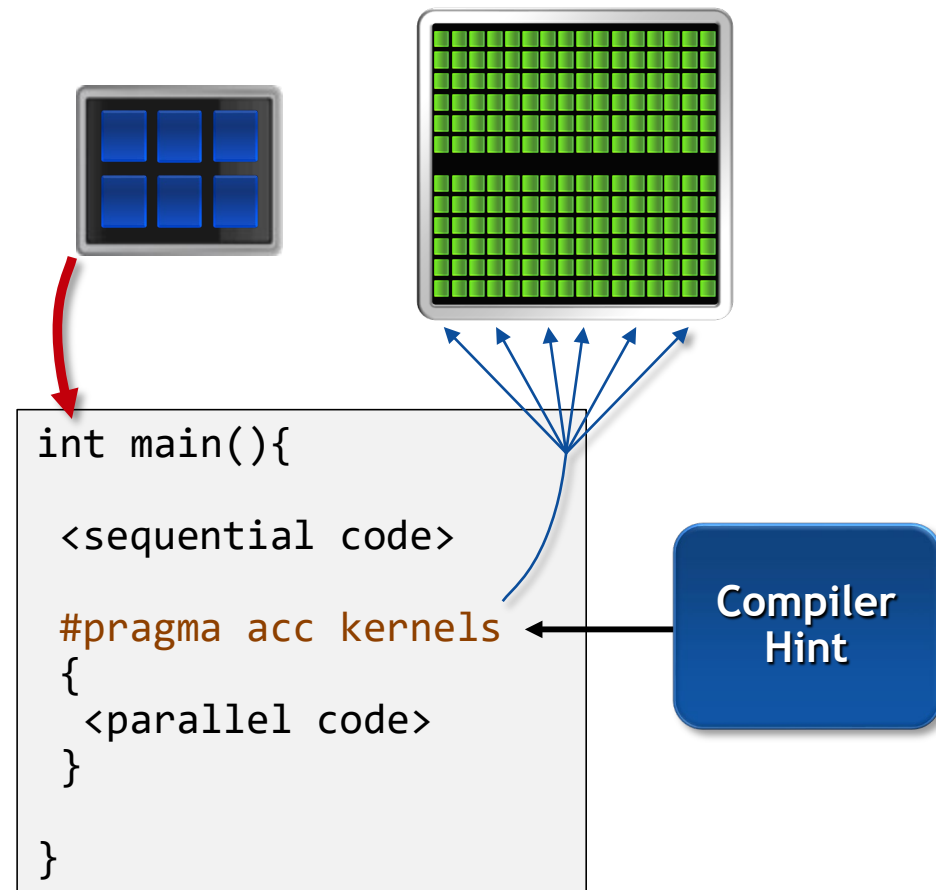
GPU  
THOUSANDS OF CORES

# CPU + GPU WORKFLOW



# GPU PROGRAMMING IN OPENACC

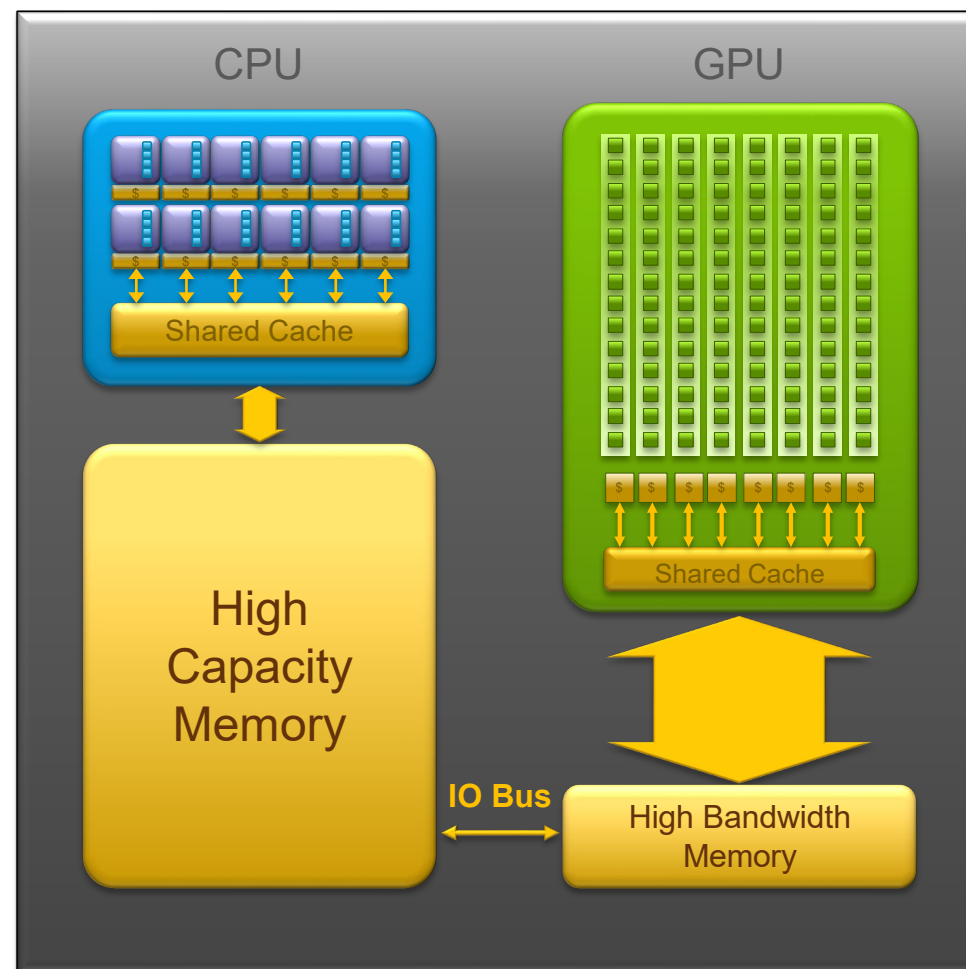
- Execution always begins and ends on the *host* CPU
- Compute-intensive loops are offloaded to the GPU using directives
- Offloading may or may not require data movement between the *host* and *device*.



# CPU + GPU

## Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCI-e)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory

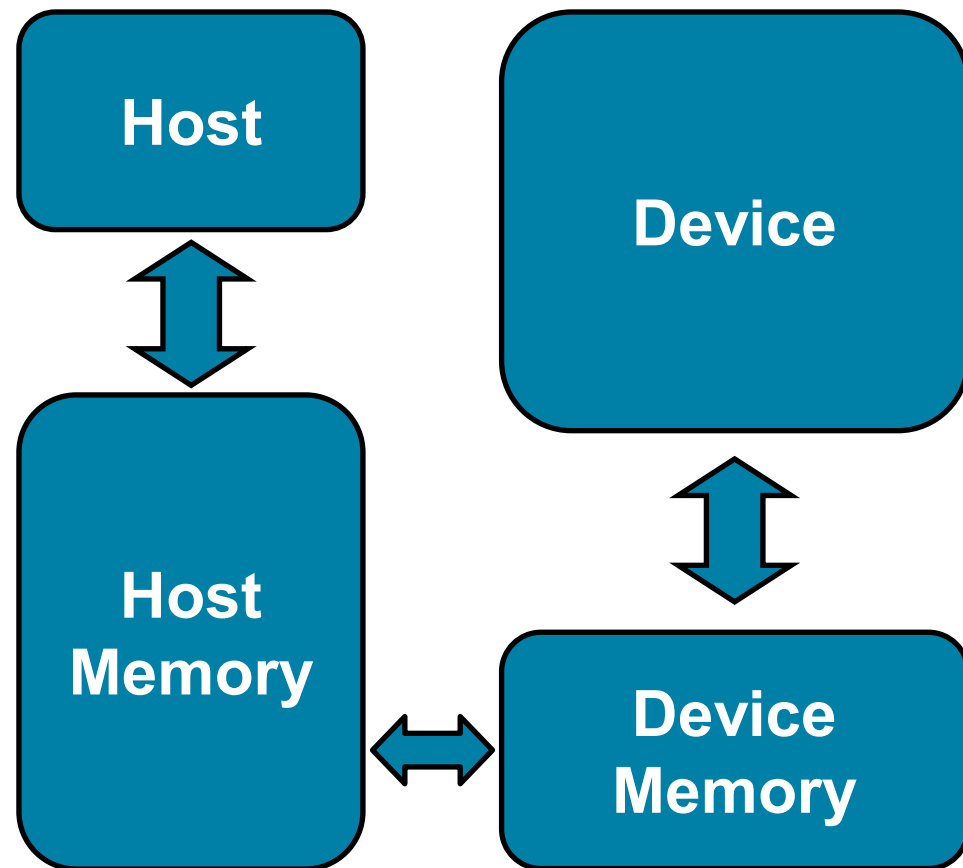


# BASIC DATA MANAGEMENT

# BASIC DATA MANAGEMENT

## Between the host and device

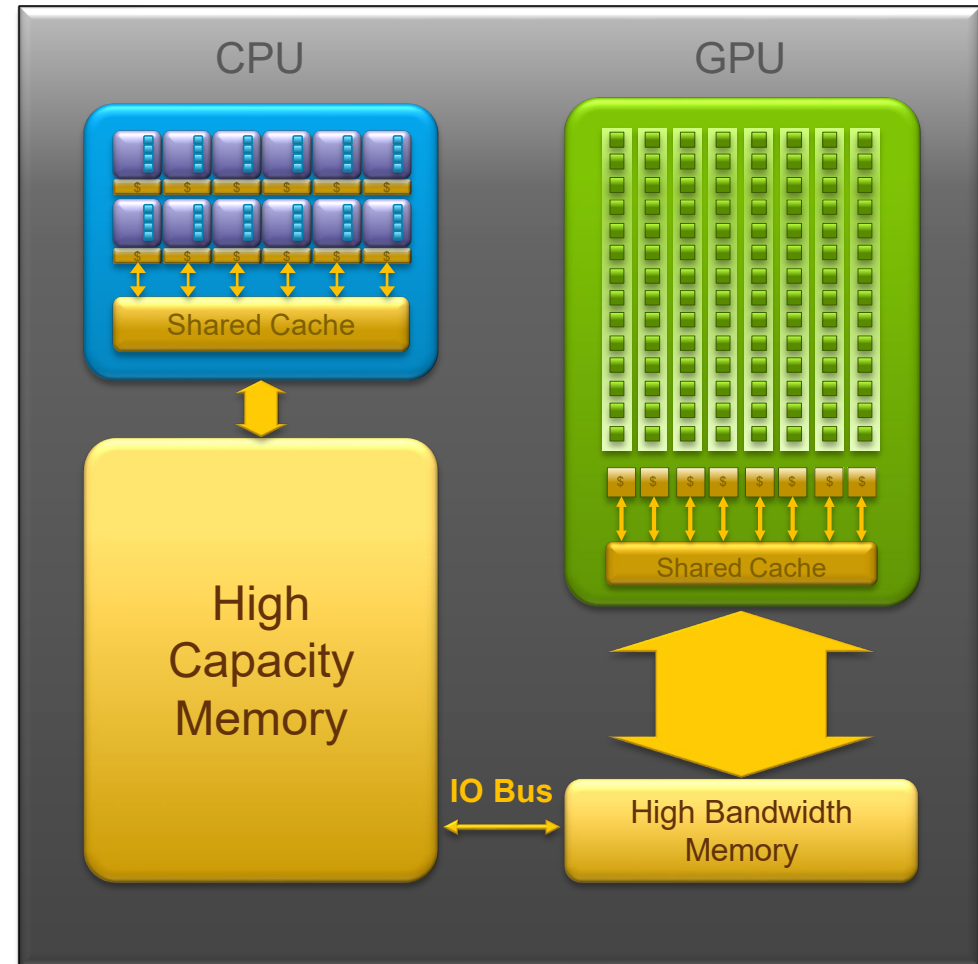
- The **host** is traditionally a CPU
- The **device** is some parallel accelerator
- When our target hardware is multicore, the host and device are the same, meaning that their memory is also the same
- There is no need to explicitly manage data when using a shared memory accelerator, such as the multicore target



# BASIC DATA MANAGEMENT

## Between the host and device

- When the target hardware is a GPU data will usually need to migrate between CPU and GPU memory
- The next lecture will discuss OpenACC data management, for now we'll assume a unified Host/Accelerator memory



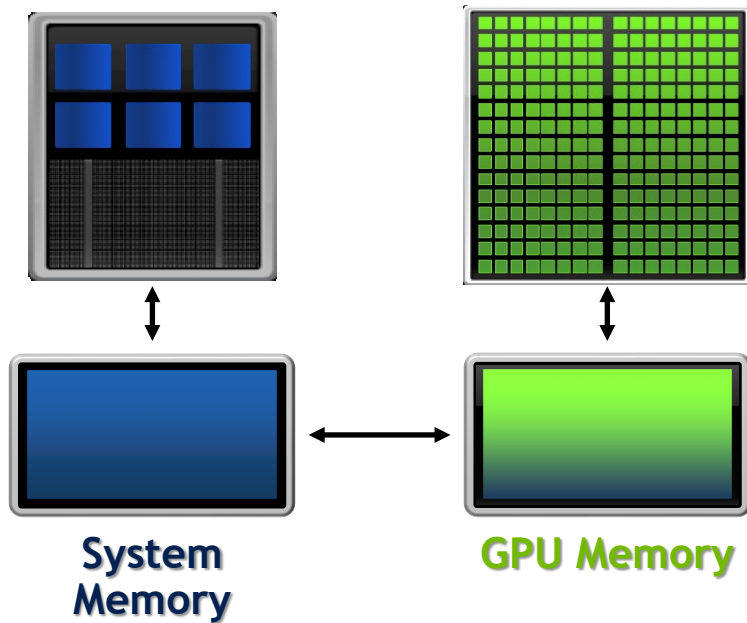
# CUDA MANAGED MEMORY

# CUDA MANAGED MEMORY

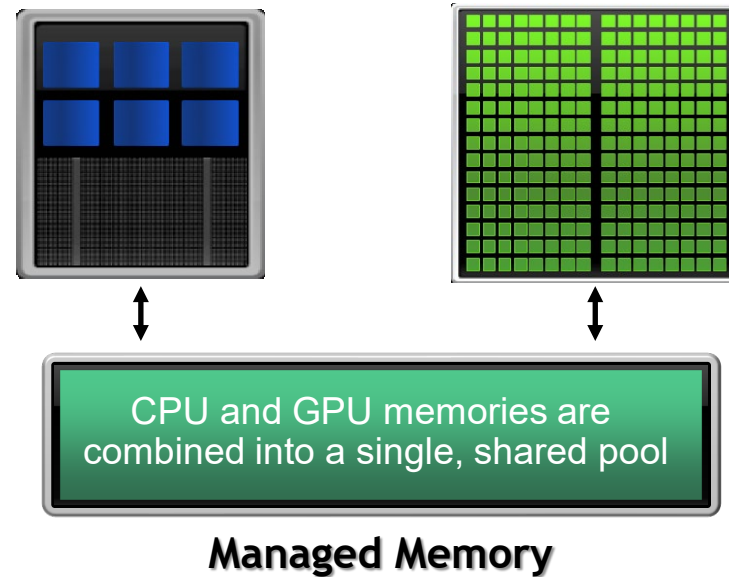
Simplified Developer Effort

Commonly referred to as “unified memory.”

Without Managed Memory



With Managed Memory



# CUDA MANAGED MEMORY

## Usefulness

- Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
- The PGI compiler can utilize CUDA Managed Memory to defer data management
- This allows the developer to concentrate on parallelism and think about data movement as an optimization

```
$ nvc -fast -acc=gpu -gpu=managed -Minfo=accel main.c
```

```
$ nvfortran -fast -acc=gpu -gpu=managed -Minfo=accel main.f90
```

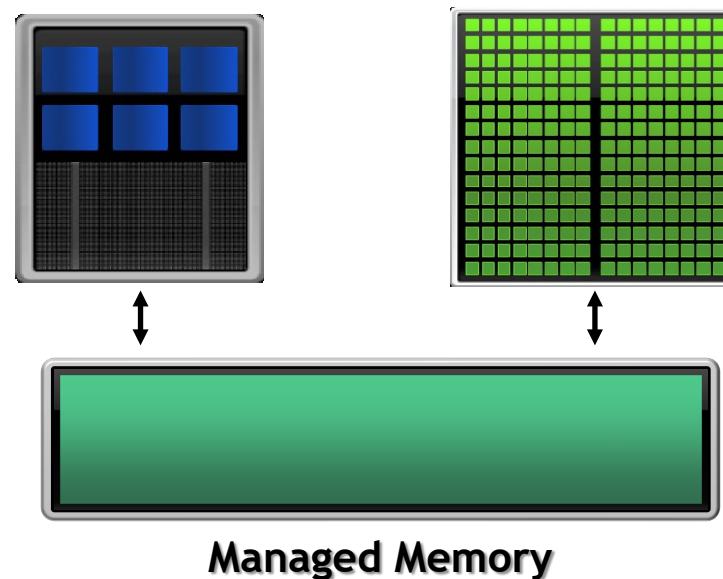
- **NEW:** The `-gpu=[no]managed` option is deprecated; please use `-gpu=mem:managed` or `-gpu=mem:separate` instead

# MANAGED MEMORY

## Limitations

- The programmer will almost always be able to get better performance by manually handling data transfers
- Memory allocation/deallocation takes longer with managed memory
- Cannot transfer data asynchronously
- Currently only available from PGI on NVIDIA GPUs.

### With Managed Memory



# OPENACC WITH MANAGED MEMORY

## An Example from the Lab Code

```
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 = fmax( error, fabs(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];
            }
        }
    }
}
```

Without Managed Memory the compiler must determine the size of A and Anew and copy their data to and from the GPU each iteration to ensure correctness

With Managed Memory the underlying runtime will move the data only when needed

# INTRODUCTION TO DATA CLAUSES

# BASIC DATA MANAGEMENT

## Moving data between the Host and Device using copy

- Data clauses allow the programmer to tell the compiler which data to move and when
- Data clauses may be added to **kernels** or **parallel** regions, but also **data**, **enter data**, and **exit data**, which will be discussed shortly

C/C++

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

# BASIC DATA MANAGEMENT

## Moving data between the Host and Device using copy

- Data clauses allow the programmer to tell the compiler which data to move and when
- Data clauses may be added to **kernels** or **parallel** regions, but also **data**, **enter data**, and **exit data**, which will be discussed shortly

C/C++

```
#pragma acc parallel loop copyout(a[0:n])  
for(int i = 0; i < N; i++){  
    a[i] = 0;  
}
```

I don't need the initial value of a, so I'll only copy it out of the region at the end.

# BASIC DATA MANAGEMENT

Moving data between the Host and Device using copy



```
#pragma acc parallel loop copy(a[0:N])  
for(int i = 0; i < N; i++){  
    a[i] = 2 * a[i];  
}
```

# BASIC DATA MANAGEMENT

Moving data between the Host and Device using copy



**CPU MEMORY**



**GPU MEMORY**



# 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

- Sometimes the compiler needs help understanding the *shape* of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

# BASIC DATA MANAGEMENT

## Multi-dimensional Array shaping

```
copy(array[0:N][0:M])
```

C/C++

```
copy(array(1:N, 1:M))
```

Fortran

# PROFILING GPU CODE

# PROFILING GPU CODE

## Obtaining information about your GPU

- Using the **nvaccelinfo** command will display information about available accelerators

### Terminal Window

```
$nvaccelinfo

CUDA Driver Version:           13000
...
Device Number:                 0
Device Name:                   NVIDIA A40
Device Revision Number:       8.6
Managed Memory:              Yes
...
Memory Models Flags:          -gpu=mem:separate,
                              -gpu=mem:managed
```

# PROFILING GPU CODE

## Obtaining information about your GPU

- Using the **nvaccelinfo** command will display information about available accelerators
- Each device is numbered starting with 0

### Terminal Window

```
$nvaccelinfo  
CUDA Driver Version:          13000  
Device Number:                0  
Device Name:                  NVIDIA A40  
Device Revision Number:      8.6  
Managed Memory:              Yes  
...  
Memory Models Flags:          -gpu=mem:separate,  
                              -gpu=mem:managed
```

# PROFILING GPU CODE

## Obtaining information about your GPU

- Using the **nvaccelinfo** command will display information about available accelerators
- Each device is numbered starting with 0
- The Device Name identifies the type of accelerator

### Terminal Window

```
$nvaccelinfo
CUDA Driver Version:      13000
...
Device Number:           0
Device Name:              NVIDIA A40
Device Revision Number:   8.6
Managed Memory:         Yes
...
Memory Models Flags:     -gpu=mem:separate,
                        -gpu=mem:managed
```

# PROFILING GPU CODE

## Obtaining information about your GPU

- Using the **nvaccelinfo** command will display information about available accelerators
- Each device is numbered starting with 0
- The Device Name identifies the type of accelerator
- Can Managed Memory be used?

### Terminal Window

```
$nvaccelinfo
CUDA Driver Version:      13000
...
Device Number:           0
Device Name:              NVIDIA A40
Device Revision Number:   8.6
Managed Memory:          Yes
...
Memory Models Flags:     -gpu=mem:separate,
                        -gpu=mem:managed
```

# PROFILING GPU CODE

## Obtaining information about your GPU

- Using the **nvaccelinfo** command will display information about available accelerators
- Each device is numbered starting with 0
- The Device Name identifies the type of accelerator
- Can Managed Memory be used?
- What compiler options should be used to target this device?

### Terminal Window

```
$nvaccelinfo

CUDA Driver Version:      13000
...
Device Number:           0
Device Name:             NVIDIA A40
Device Revision Number:  8.6
Managed Memory:        Yes
...
Memory Models Flags:    -gpu=mem:separate,
                        -gpu=mem:managed
```

Without Manage Memory

```
$ nvc -acc=gpu [-gpu=mem:separate] main.c
```

With Manage Memory

```
$ nvc -acc=gpu -gpu=mem:managed main.c
```

deprecated 

```
$ nvc -acc=gpu -gpu=managed main.c
```

# COMPILING GPU CODE

## Terminal Window

```
$ nvc -fast -acc=gpu -Minfo=accel jacobi.c laplace2d.c
calcNext:
  37, Generating copy(Anew[:m*n],A[:m*n]) ←
  Accelerator kernel generated
  Generating Tesla code
  37, Generating reduction(max:error)
  38, #pragma acc loop gang /* blockIdx.x */
  41, #pragma acc loop vector(128) /* threadIdx.x */
41, Loop is parallelizable
swap:
  56, Generating copy(Anew[:m*n],A[:m*n]) ←
  Accelerator kernel generated
  Generating Tesla code
  57, #pragma acc loop gang /* blockIdx.x */
  60, #pragma acc loop vector(128) /* threadIdx.x */
60, Loop is parallelizable
```

We can see that our data copies are being applied by the compiler

# COMPILING GPU CODE

## Terminal Window

```
$ nvc -fast -acc=gpu -Minfo=accel jacobi.c laplace2d.c
calcNext:
  37, Generating copy(Anew[:m*n],A[:m*n])
    Accelerator kernel generated
    Generating Tesla code ←
  37, Generating reduction(max:error)
  38, #pragma acc loop gang /* blockIdx.x */
  41, #pragma acc loop vector(128) /* threadIdx.x */
41, Loop is parallelizable
swap:
  56, Generating copy(Anew[:m*n],A[:m*n])
    Accelerator kernel generated
    Generating Tesla code ←
  57, #pragma acc loop gang /* blockIdx.x */
  60, #pragma acc loop vector(128) /* threadIdx.x */
60, Loop is parallelizable
```

We also see that the compiler is generating code for our GPU

# COMPILING GPU CODE

## Terminal Window

```
$ nvc -fast -acc=gpu -Minfo=accel jacobi.c laplace2d.c
calcNext:
  37, Generating copy(Anew[:m*n],A[:m*n])
  Accelerator kernel generated
  Generating Tesla code ←
  37, Generating reduction(max:error)
  38, #pragma acc loop gang /* blockIdx.x */ ←
  41, #pragma acc loop vector(128) /* threadIdx.x */
41, Loop is parallelizable
swap:
  56, Generating copy(Anew[:m*n],A[:m*n])
  Accelerator kernel generated
  Generating Tesla code ←
  57, #pragma acc loop gang /* blockIdx.x */ ←
  60, #pragma acc loop vector(128) /* threadIdx.x */
60, Loop is parallelizable
```

This is the parallelization of  
the **outer loop**

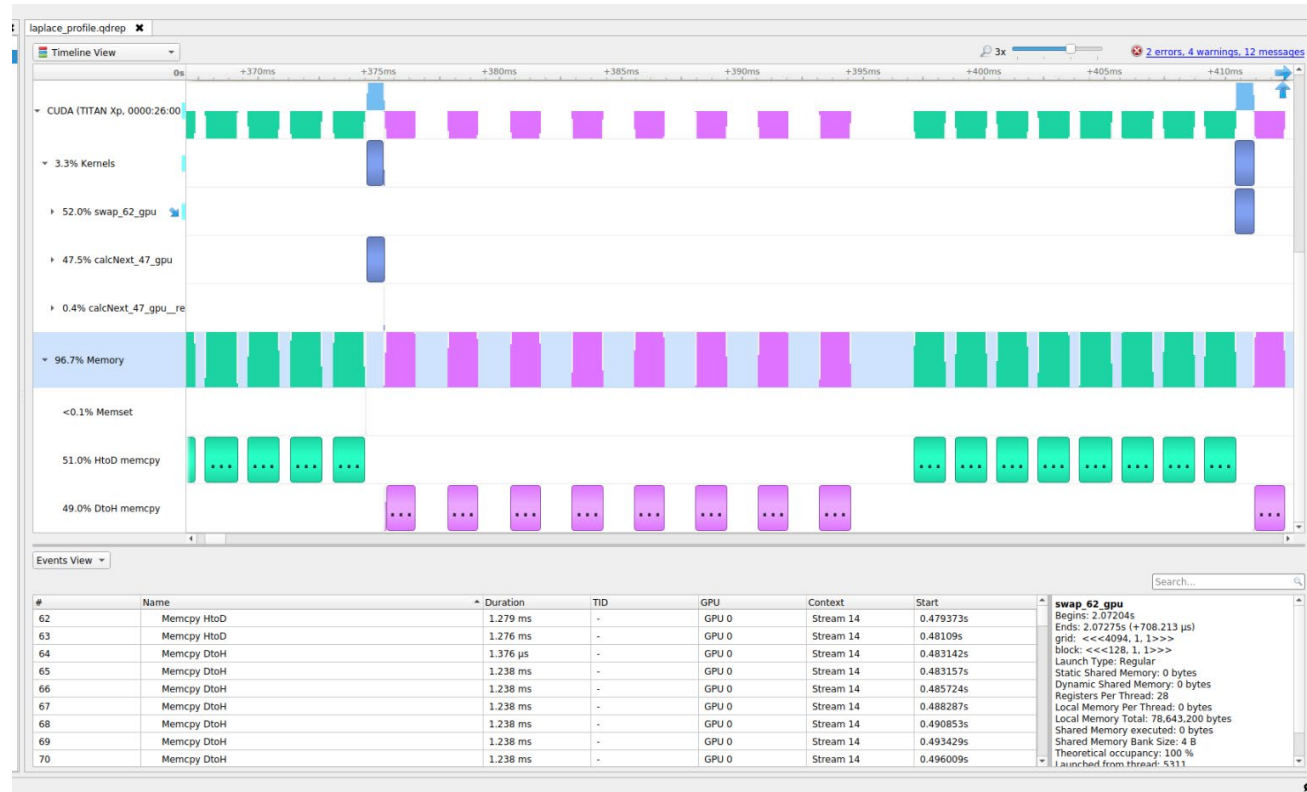
# COMPILING GPU CODE @ ALEX

```
a40o0002@a0122:~/OpenACC-solutions/C$ nvc -fast -acc=gpu -Minfo=accel,opt -o laplace-gpu-parallel jacobi.c laplace2d-gpu-parallel.c
jacobi.c:
laplace2d-gpu-parallel.c:
calcNext:
    47, Generating copy(A[:n*m]) [if not already present]
        Generating implicit firstprivate(j,n,m)
        Generating NVIDIA GPU code
    49, #pragma acc loop gang /* blockIdx.x */
        Generating implicit reduction(max:error)
    52, #pragma acc loop vector(128) /* threadIdx.x */
47, Generating implicit copy(error) [if not already present]
    Generating copy(Anew[:n*m]) [if not already present]
52, Loop is parallelizable
swap:
    63, Generating copy(A[:n*m],Anew[:n*m]) [if not already present]
        Generating implicit firstprivate(j,n,m)
        Generating NVIDIA GPU code
    65, #pragma acc loop gang /* blockIdx.x */
    68, #pragma acc loop vector(128) /* threadIdx.x */
68, Loop is parallelizable
a40o0002@a0122:~/OpenACC-solutions/C$ █
```

# PROFILING GPU CODE (NSIGHT SYSTEMS)

## Using *nsys* to profile GPU code

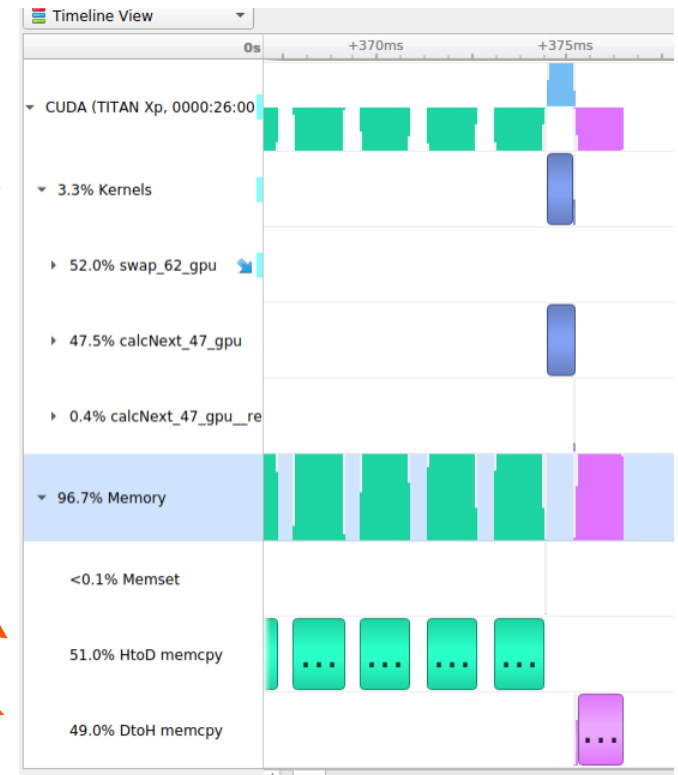
- Nsight Systems presents far more information when running on a GPU
- It is capable of capturing information about CUDA execution in the profiled process.
- In the Timeline view, you can see all the information about kernels and memory movements (expand the CUDA row)



# PROFILING GPU CODE (NSIGHT SYSTEMS)

Using *nsys* to profile GPU code

- **Kernels:** These are our computational functions. We can see our `calcNext` and `swap` function
- **MemCpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **MemCpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)



# PROFILING GPU CODE

## Receiving unexpected code results

- Here we can see the runtime of our application: 151 seconds
- The program is now performing over 3 times **worse** than the sequential version
- A profiler can help us understand why this performance is worse

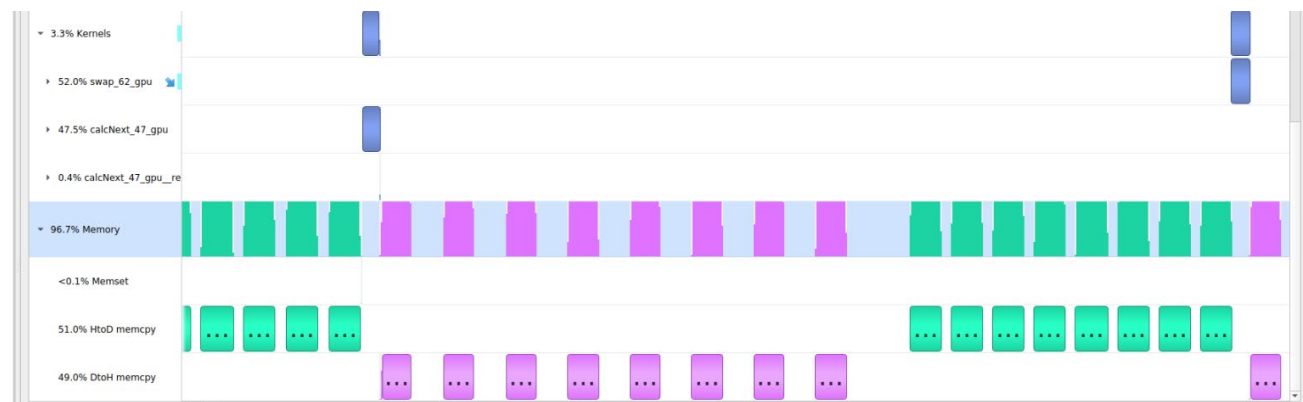
### Terminal Window

```
$ nvc -acc=gpu jacobi.c laplace2d.c
$ ./a.out
  0, 0.250000
 100, 0.002397
 200, 0.001204
 300, 0.000804
 400, 0.000603
 500, 0.000483
 600, 0.000403
 700, 0.000345
 800, 0.000302
 900, 0.000269
total: 151.772627 s
```

# PROFILING GPU CODE

## Inspecting the Nsight Systems timeline

- Let's focus on the data movement (Memory row)
- At a first glance, it looks like our program is spending a significant amount of time transferring data between the host and device
- We also see that the compute regions are very small and spread out
- What if we try Managed Memory?



# PROFILING GPU CODE

## Using managed memory

- Using managed memory drastically improves performance
- This managed memory version is performing over 20x better than the sequential code
- What does the profiler tell us about this?

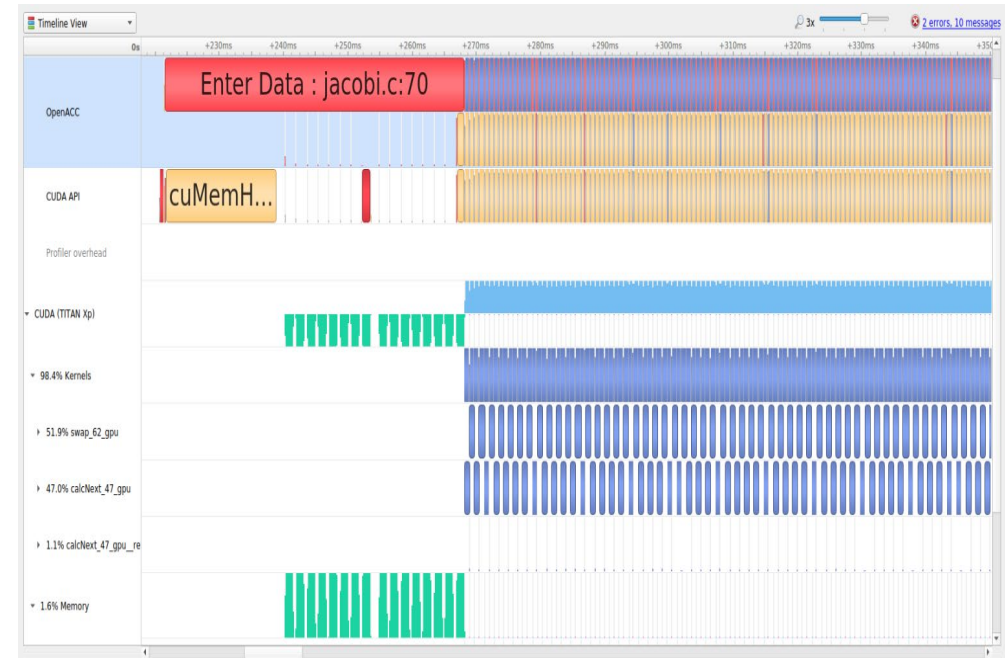
### Terminal Window

```
$ nvc -acc=gpu -gpu=managed jacobi.c  
laplace2d.c  
$ ./a.out  
  0, 0.250000  
100, 0.002397  
200, 0.001204  
300, 0.000804  
400, 0.000603  
500, 0.000483  
600, 0.000403  
700, 0.000345  
800, 0.000302  
900, 0.000269  
total: 1.474951 s
```

# PROFILING GPU CODE

## Using managed memory

- The data no longer needs to transfer between each kernel
- The data is only moved when it's first accessed on the GPU or CPU
- During the timestepping data remains on the device
- Now a higher percentage of time is spent computing



# KEY CONCEPTS

In this module we discussed...

- The fundamental differences between CPUs and GPUs
- Assisting the compiler by providing information about array sizes for data management
- Managed memory

# THANK YOU