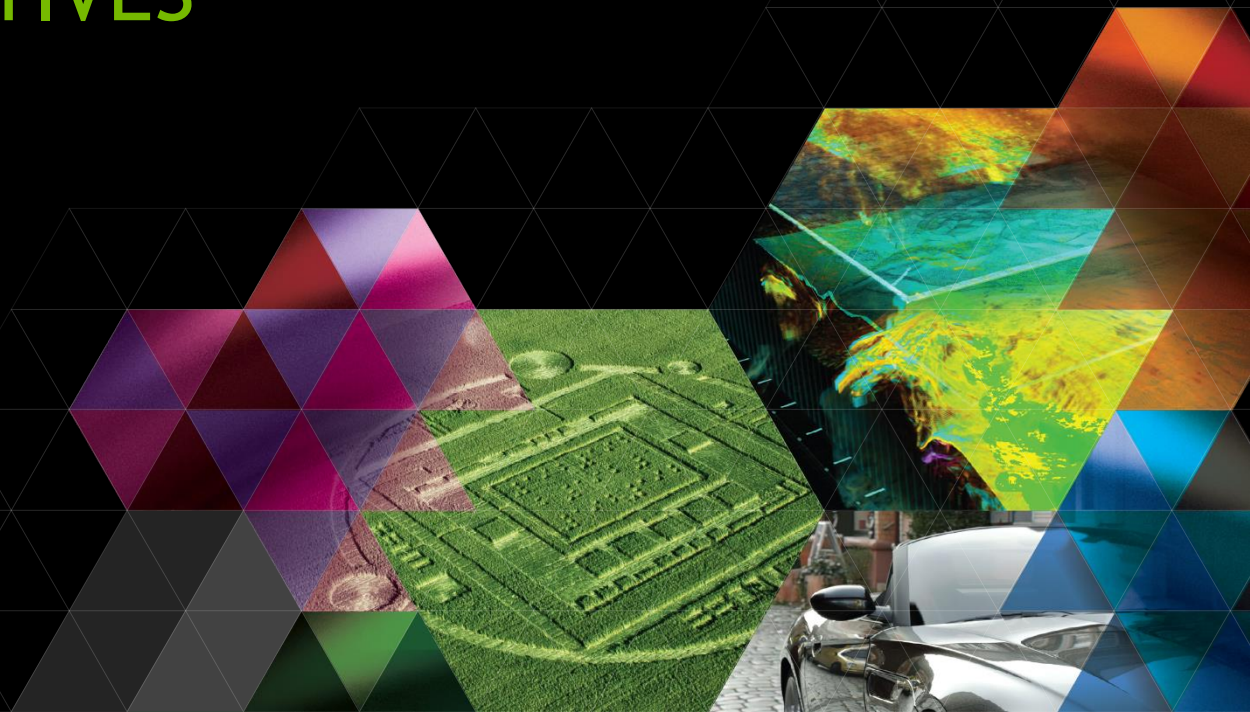
GPU TECHNOLOGY CONFERENCE

NVIDIA

ADVANCED ACCELERATED COMPUTING USING COMPILER DIRECTIVES

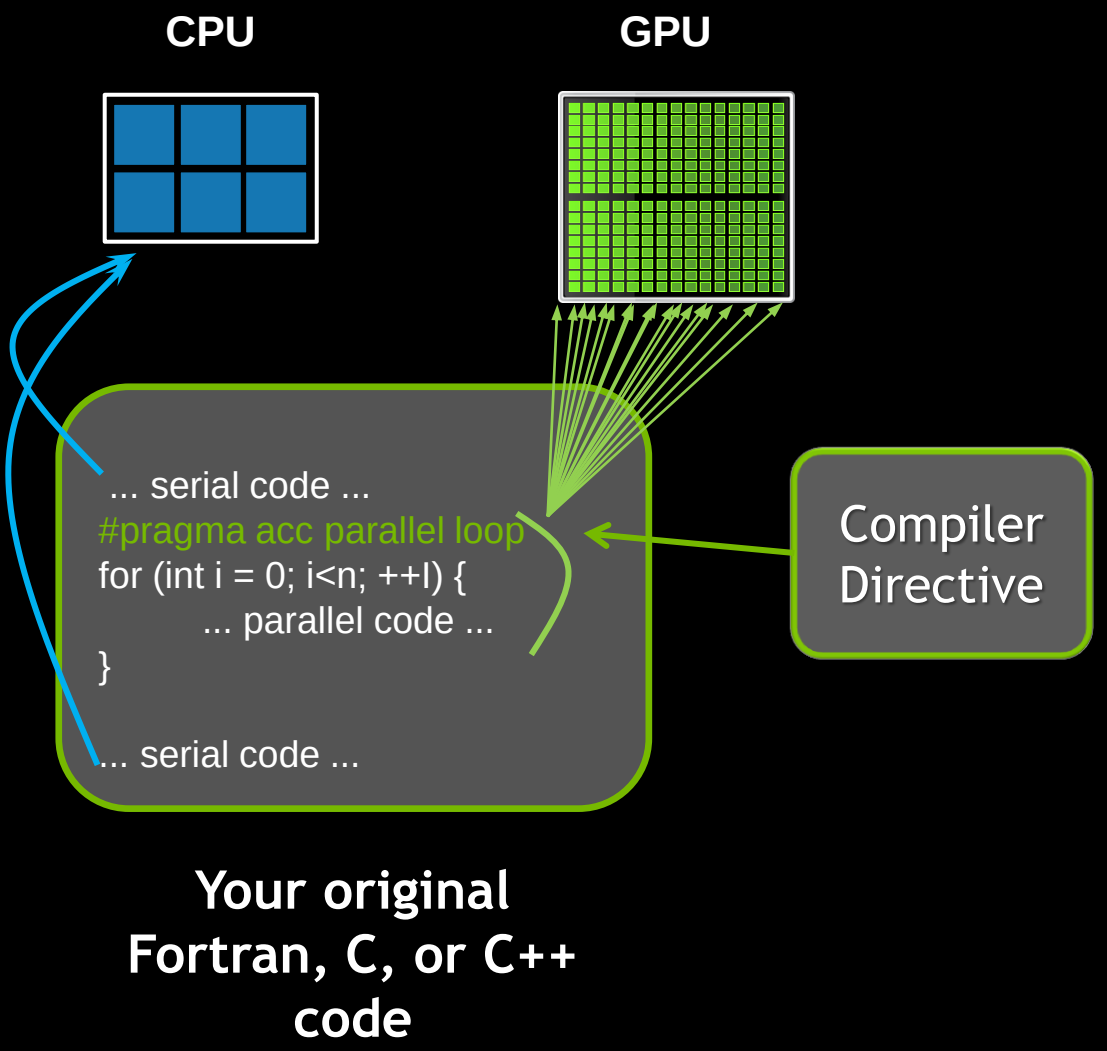
Jeff Larkin, NVIDIA



OUTLINE

- Compiler Directives Review
- Asynchronous Execution
- OpenACC Interoperability
- OpenACC `routine`
- Advanced Data Directives
- OpenACC atomics
- Miscellaneous Best Practices
- Next Steps

WHAT ARE COMPILER DIRECTIVES?



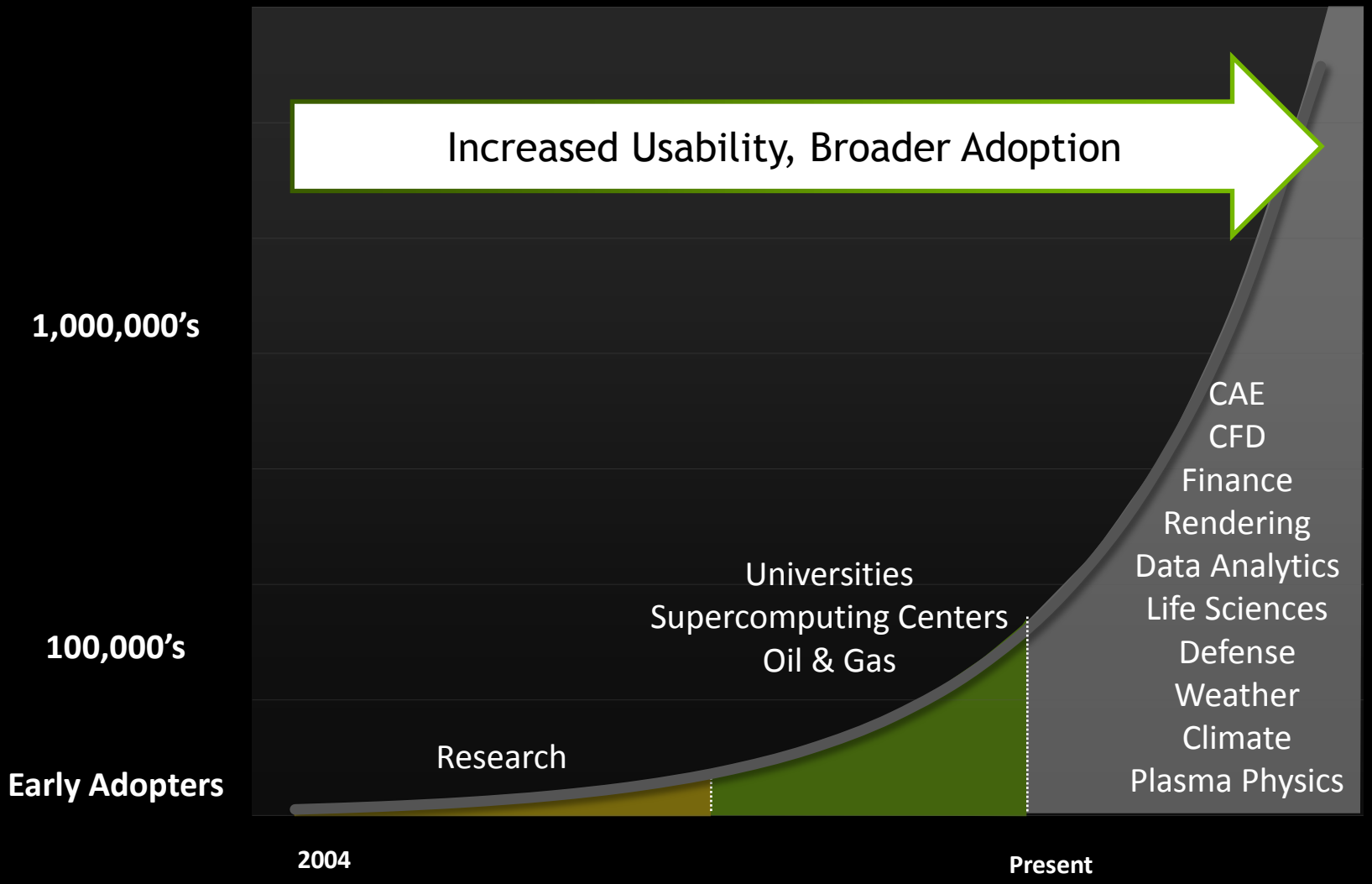
When a compiler directive is encountered the compiler/runtime will…

1. Generate parallel code for GPU
2. Allocate GPU memory and copy input data
3. Execute parallel code on GPU
4. Copy output data to CPU and deallocate GPU memory

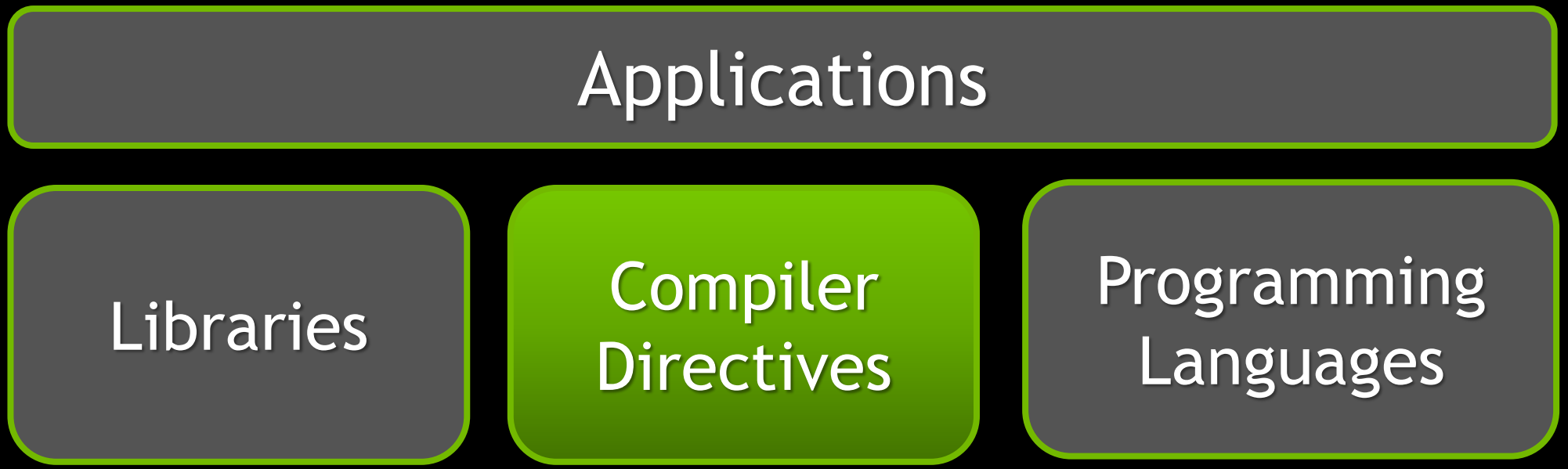
WHY USE COMPILER DIRECTIVES?

- Single Source Code
  - No need to maintain multiple code paths
- High Level
  - Abstract away device details, focus on expressing the parallelism and data locality
- Low Learning Curve
  - Programmer remains in same language and adds directives to existing code
- Rapid Development
  - Fewer code changes means faster development

WHY SUPPORT COMPILER DIRECTIVES?



3 WAYS TO ACCELERATE APPLICATIONS



"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

OPENACC 2.0 AND OPENMP 4.0

OPENACC 2.0

- OpenACC is a specification for high-level, compiler directives for expressing parallelism for accelerators.
  - Aims to be performance portable to a wide range of accelerators.
  - Multiple Vendors, Multiple Devices, One Specification
- The OpenACC specification was first released in November 2011.
  - Original members: CAPS, Cray, NVIDIA, Portland Group
- OpenACC 2.0 was released in June 2013, expanding functionality and improving portability
- At the end of 2013, OpenACC had more than 10 member organizations

OPENACC DIRECTIVE SYNTAX

- C/C++

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

…often followed by a structured code block

- Fortran

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

…often paired with a matching end directive surrounding a structured code block:

```
!$acc end directive
```

OPENACC EXAMPLE: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop
  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(n), y(n), a
  integer :: n, i

  !$acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$acc end parallel loop
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...
```

OPENMP 4.0

- OpenMP has existed since 1997 as a specification for compiler directives for shared memory parallelism.
- In 2013, OpenMP 4.0 was released, expanding the focus beyond shared memory parallel computers, including accelerators.
- The OpenMP 4.0 **`target`** construct provides the means to offload data and computation to accelerators.
- Additional directives were added to support multiple thread teams and simd parallelism.
- OpenMP continues to improve upon its support for offloading to accelerators.

OPENMP DIRECTIVE SYNTAX

- C/C++

```
#pragma omp target directive [clause [,] clause]…]
```

…often followed by a structured code block

- Fortran

```
!$omp target directive [clause [,] clause] …]
```

…often paired with a matching end directive surrounding a structured code block:

```
!$omp end target directive
```

OPENMP TARGET EXAMPLE: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp target teams \
        distribute parallel for
  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(n), y(n), a
  integer :: n, i

  !$omp  target teams &
  !$omp& distribute parallel do
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$omp end target teams &
  !$omp& distribute parallel do
end subroutine saxpy
 ...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...
```

ASYNCHRONOUS EXECUTION

OPENACC `async` CLAUSE

An async clause may be added to most OpenACC directives, causing the affected section of code to run asynchronously with the host.

`async(handle)` – Add this directive to the asynchronous queue represented by the integer *handle*.

- Asynchronous activities on the same queue execute in order
- The user must *wait* before accessing variables associated with the asynchronous region.

```
#pragma acc parallel loop async(1)
```

OPENACC `WAIT` DIRECTIVE

The wait directive instructs the compiler to wait for asynchronous work to complete.

```
#pragma acc wait(1)
```

`wait` - Wait on all asynchronous work to complete

`wait(handle)` - Wait strictly for the asynchronous queue represented by *handle* to complete.

`wait(handle1) async(handle2)` - The asynchronous queue represented by *handle2* must wait for all work in *handle1* to complete before proceeding, but control is returned to the CPU.

OPENACC PIPELINING

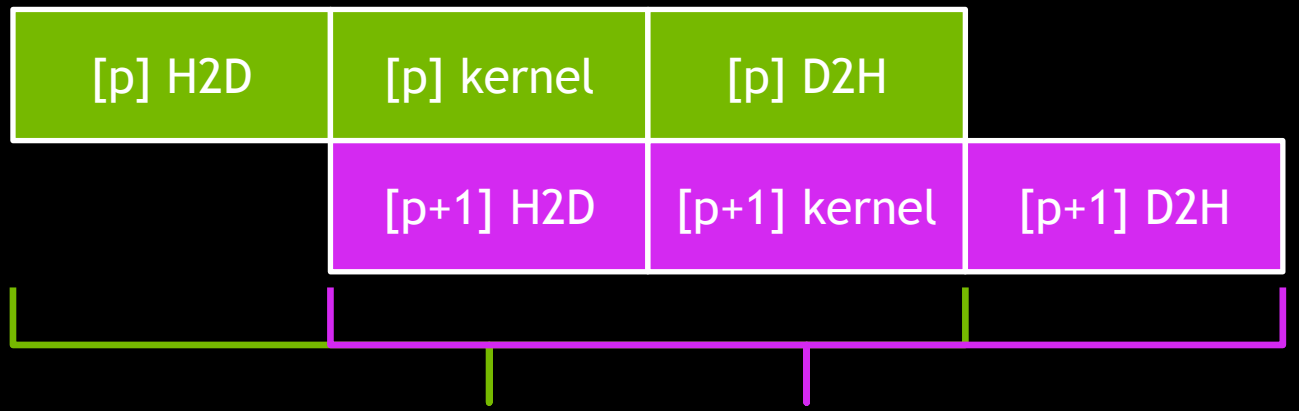
```
#pragma acc data
for(int p = 0; p < nplanes; p++)
{
  #pragma acc update device(plane[p])
  #pragma acc parallel loop
  for (int i = 0; i < nwork; i++)
  {
    // Do work on plane[p]
  }
  #pragma acc update self(plane[p])
}
```

For this example, assume that each “plane” is completely independent and must be copied to/from the device.

As it is currently written, plane[p+1] will not begin copying to the GPU until plane[p] is copied from the GPU.

OPENACC PIPELINING (CONT.)

P and P+1 Serialize



P and P+1 Overlap Data Movement

NOTE: In real applications, your boxes will not be so evenly sized.

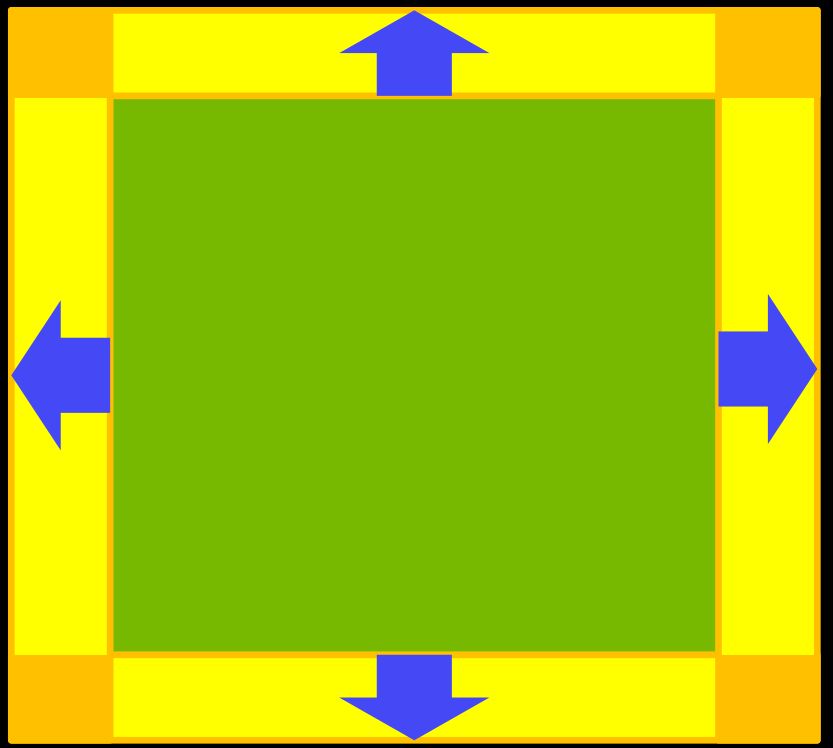
OPENACC PIPELINING (CONT.)

```
#pragma acc data
for(int p = 0; p < nplanes; p++)
{
  #pragma acc update device(plane[p]) async(p)
  #pragma acc parallel loop async(p)
  for (int i = 0; i < nwork; i++)
  {
    // Do work on plane[p]
  }
  #pragma acc update self(plane[p]) async(p)
}
#pragma acc wait
```

Enqueue each plane in a queue to execute in order

Wait on all queues.

PIPELINED BOUNDARY EXCHANGE



Some algorithms, such as this boundary exchange, have a natural pipeline.

Each boundary zone is exchanged with one neighbor, so a separate queue may be used for each zone.

ASYNCHRONOUS TIP

- Even if you can't achieve overlapping, it may be beneficial to use async to queue up work to reduce latency.

- The Cray compiler can do this automatically with the `-h accel_model=auto_async_all` flag

OPENACC INTEROPERABILITY

OPENACC INTEROPERABILITY

OpenACC plays well with others.

- Add CUDA, OpenCL, or accelerated libraries to an OpenACC application
- Add OpenACC to an existing accelerated application
- Share data between OpenACC and CUDA

OPENACC & CUDA STREAMS

OpenACC *suggests* two functions for interoperating with CUDA streams:

- `void* acc_get_cuda_stream( int async );`
- `int acc_set_cuda_stream( int async, void* stream );`

OPENACC HOST_DATA DIRECTIVE

Exposes the *device* address of particular objects to the *host* code.

```
#pragma acc data copy(x,y)
{
// x and y are host pointers
#pragma acc host_data use_device(x,y)
{
 // x and y are device pointers
}
// x and y are host pointers
}
```

X and Y are device pointers here

HOST_DATA EXAMPLE

OpenACC Main

```
program main
  integer, parameter :: N = 2**20
  real, dimension(N) :: X, Y
  real               :: A = 2.0

  !$acc data
  ! Initialize X and Y
  ...

  !$acc host_data use_device(x,y)
  call saxpy(n, a, x, y)
  !$acc end host_data
  !$acc end data

end program
```

- It's possible to interoperate from C/C++ or Fortran.
- OpenACC manages the data and passes device pointers to CUDA.

CUDA C Kernel & Wrapper

```
__global__
void saxpy_kernel(int n, float a,
      float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a, float *dx, float *dy)
{
  // Launch CUDA Kernel
  saxpy_kernel<<<4096,256>>>(N, 2.0, dx, dy);
}
```

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

CUBLAS LIBRARY & OPENACC

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust
- …

OpenACC Main Calling CUBLAS

```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...

cublasInit();

#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}

cublasShutdown();
```

OPENACC `DEVICEPTR`

The `deviceptr` clause informs the compiler that an object is already on the device, so no translation is necessary.

- Valid for `parallel`, `kernels`, and `data`

```
cudaMalloc((void*)&x,(size_t)n*sizeof(float));
cudaMalloc((void*)&y,(size_t)n*sizeof(float));

#pragma acc parallel loop deviceptr(x,y)
for(int i=0; i<n ; i++)
{
  y(i) = a*x(i)+y(i)
}
```

Do not translate x and y, they are already on the device.

DEVICEPTR EXAMPLE

OpenACC Kernels

```
void saxpy(int n, float a, float * restrict
x, float * restrict y)
{
#pragma acc kernels deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

By passing a device pointer to an OpenACC region, it's possible to add OpenACC to an existing CUDA code.

CUDA C Main

```
int main(int argc, char **argv)
{
  float *x, *y, tmp;
  int n = 1<<20, i;

  cudaMalloc((void*)&x,(size_t)n*sizeof(float));
  cudaMalloc((void*)&y,(size_t)n*sizeof(float));

  ...

  saxpy(n, 2.0, x, y);
  cudaMemcpy(&tmp,y,(size_t)sizeof(float),
             cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory is managed via standard CUDA calls.

OPENACC & THRUST

Thrust (thrust.github.io) is a STL-like library for C++ on accelerators.

- High-level interface
- Host/Device container classes
- Common parallel algorithms

It's possible to cast Thrust vectors to device pointers for use with OpenACC

```
void saxpy(int n, float a, float * restrict
x, float * restrict y)
{
#pragma acc kernels deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
for(int i=0; i<N; i++)
{
  x[i] = 1.0f;
  y[i] = 0.0f;
}

// Copy to Device
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

saxpy(N,2.0, d_x.data().get(),
             d_y.data().get());

// Copy back to host
y = d_y;
```

OPENACC `ACC_MAP_DATA` FUNCTION

The `acc_map_data` (`acc_unmap_data`) maps (unmaps) an existing device allocation to an OpenACC variable.

```
cudaMalloc((void*)&x_d,(size_t)n*sizeof(float));
acc_map_data(x, x_d, n*sizeof(float));
cudaMalloc((void*)&y_d,(size_t)n*sizeof(float));
acc_map_data(y, y_d, n*sizeof(float));

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

Allocate device arrays with CUDA and *map* to OpenACC

Here x and y will reuse the memory of x_d and y_d

OPENACC ROUTINE DIRECTIVE

OPENACC `ROUTINE` DIRECTIVE

The routine directive specifies that the compiler should generate a device copy of the function/subroutine in addition to the host copy.

Clauses:

- `gang/worker/vector/seq`
  - **Specifies the level of parallelism contained in the routine.**
- `bind`
  - Specifies an optional name for the routine, also supplied at call-site
- `no_host`
  - The routine will only be used on the device
- `device_type`
  - Specialize this routine for a particular device type.

OPENACC ROUTINE: BLACK SCHOLES

```
////////////////////////////////////////////////////////////////////////////////
// Polynomial approximation of cumulative normal distribution function
////////////////////////////////////////////////////////////////////////////////
#pragma acc routine seq
real CND(real d)
{
    const real       A1 = (real)0.31938153;
    const real       A2 = (real)-0.356563782;
    const real       A3 = (real)1.781477937;
    const real       A4 = (real)-1.821255978;
    const real       A5 = (real)1.330274429;
    const real RSQRT2PI = (real)0.39894228040143267793994605993438;

    real
        K = (real)1.0 / ((real)1.0 + (real)0.2316419 * FABS(d));

    real
        cnd = RSQRT2PI * EXP(- (real)0.5 * d * d) *
        (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));

    if(d > 0)
        cnd = (real)1.0 - cnd;

    return cnd;
}
```

OPENACC ROUTINE: FORTRAN

```
subroutine foo(v, i, n) {
  use …
  !$acc routine vector
  real :: v(:,:)
  integer :: i, n
  !$acc loop vector
  do j=1,n
    v(i,j) = 1.0/(i*j)
  enddo
end subroutine

!$acc parallel loop
do i=1,n
  call foo(v,i,n)
enddo
!$acc end parallel loop
```

The `routine` directive may appear in a fortran function or subroutine definition, or in an interface block.

Nested acc routines require the routine directive within each nested routine.

The save attribute is not supported.

ADVANCED DATA DIRECTIVES

OPENACC DATA REGIONS REVIEW

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

#pragma acc parallel loop reduction(max:err)
  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]);

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

#pragma acc parallel loop
  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 to/from the accelerator only when needed.
Create Anew as a device temporary.

Data is shared within this region.

OPENACC UPDATE DIRECTIVE

Programmer specifies an array (or partial array) that should be refreshed within a data region.

The programmer may choose to specify only part of the array to update.

```
do_something_on_device()

!$acc update self(a)

do_something_on_host()

!$acc update device(a)
```

Copy “a” from GPU to CPU

Copy “a” from CPU to GPU

UNSTRUCTURED DATA REGIONS

OpenACC 2.0 provides a means for beginning and ending a data region in different program scopes.

```
double a[100];

#pragma acc data copy(a)
{
  // OpenACC code
}
```

```
double a[100];

#pragma acc enter data copyin(a)
// OpenACC code
#pragma acc exit data copyout(a)
```

UNSTRUCTURED DATA REGIONS: C++ CLASSES

```
class Matrix {
  Matrix(int n) {
    len = n;
    v = new double[len];
    #pragma acc enter data create(v[0:len])
  }
  ~Matrix() {
    #pragma acc exit data delete(v[0:len])
    delete[] v;
  }

  private:
    double* v;
    int len;
};
```

- Unstructured Data Regions enable OpenACC to be used in C++ classes
- Unstructured data regions can be used whenever data is allocated and initialized in a different scope than where it is freed.

EXTENDED DATA API

Most of the data directives can also be used as function calls from C:

- `acc_malloc / acc_free`
- `acc_update_device / acc_update_self`
- `acc_create / acc_delete`
- `acc_memcpy_to_device / acc_memcpy_from_device`
- `acc_copy* / acc_present_or_*`

These are useful for more complex data structures.

OPENACC *SLIDING WINDOW*

When the GPU does not have sufficient memory for a problem, **acc_map_data** may be used to implement a sliding window for moving data to the GPU.

Multiple windows + async may hide PCIe costs, but increase code complexity.

```
double *bigDataArray, *slidingWindow;
// Allocate on device
slidingWindow = (double*)acc_malloc(...);
for(...)
{
  // Map window to a slide of data
  acc_map_data(&bigDataArray[X],
               slidingWindow,...);
  #pragma acc update device(bigDataArray[X:N])
  #pragma acc kernels
  {
  ...
  }
  #pragma acc update self(bigDataArray[X:N])
  // Unmap slice for next iteration
  acc_unmap_data(&bigDataArray[X]);
}
acc_free(slidingWindow);
```

OPENACC ATOMICS

OPENACC `ATOMIC` DIRECTIVE

- Ensures a variable is accessed atomically, preventing race conditions and inconsistent results.

```
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
  if ( x[i] > 0 )
  {
    #pragma acc atomic capture
    {
      cnt++;
    }
  }
}
```

The atomic construct may `read`, `write`, `update`, or `capture` variables in a section of code.

`cnt` can only be accessed by 1 thread at a time.

MISCELLANEOUS BEST PRACTICES

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
- Compilers often require `restrict` to determine independence (true for OpenACC, OpenMP, and vectorization)
  - Otherwise the compiler can't parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

http://en.wikipedia.org/wiki/Restrict

EFFICIENT LOOP ORDERING

- Stride-1 inner loops (each loop iteration accesses the next element in memory) will generally work best both for the compiler and the GPU.
  - Simpler to vectorize
  - Better memory throughput
- In C, loop on the right-most dimension
- In Fortran, loop on the left-most dimension

LOOP FUSION/FISSION

It's often beneficial to…

- Fuse several small loops into one larger one
  - Multiple small loops result in multiple small kernels, which may not run long enough to cover launch latencies.
- Split large loops into multiple small ones
  - Very large loops consume a lot of resources (registers, shared memory, etc.) which may result in poor GPU utilization

REDUCING DATA MOVEMENT

- It’s often beneficial to move a loop to the GPU, even if it’s not actually faster on the GPU, if it prevents copying data to/from the device.

NEXT STEPS

NEXT STEPS: TRY OPENACC

- GTC14 has multiple hands-on labs for trying OpenACC
  - S4803 - Getting Started with OpenACC – Wednesday 5PM
  - S4796 - Parallel Programming: OpenACC Profiling – Thursday 2PM
- Get a PGI Trial license for your machine
- NVIDIA Developer Blogs
  - Parallel Forall Blog
  - CUDACasts Screencast Series

MORE AT GTC14

- S4514 - Panel on Compiler Directives for Accelerated Computing
  - Wednesday @ 4PM
  - Wednesday afternoon in LL20C has all compiler-directives talks
- Check the sessions agenda for more talks on compiler directives

Please remember to fill out your surveys.