

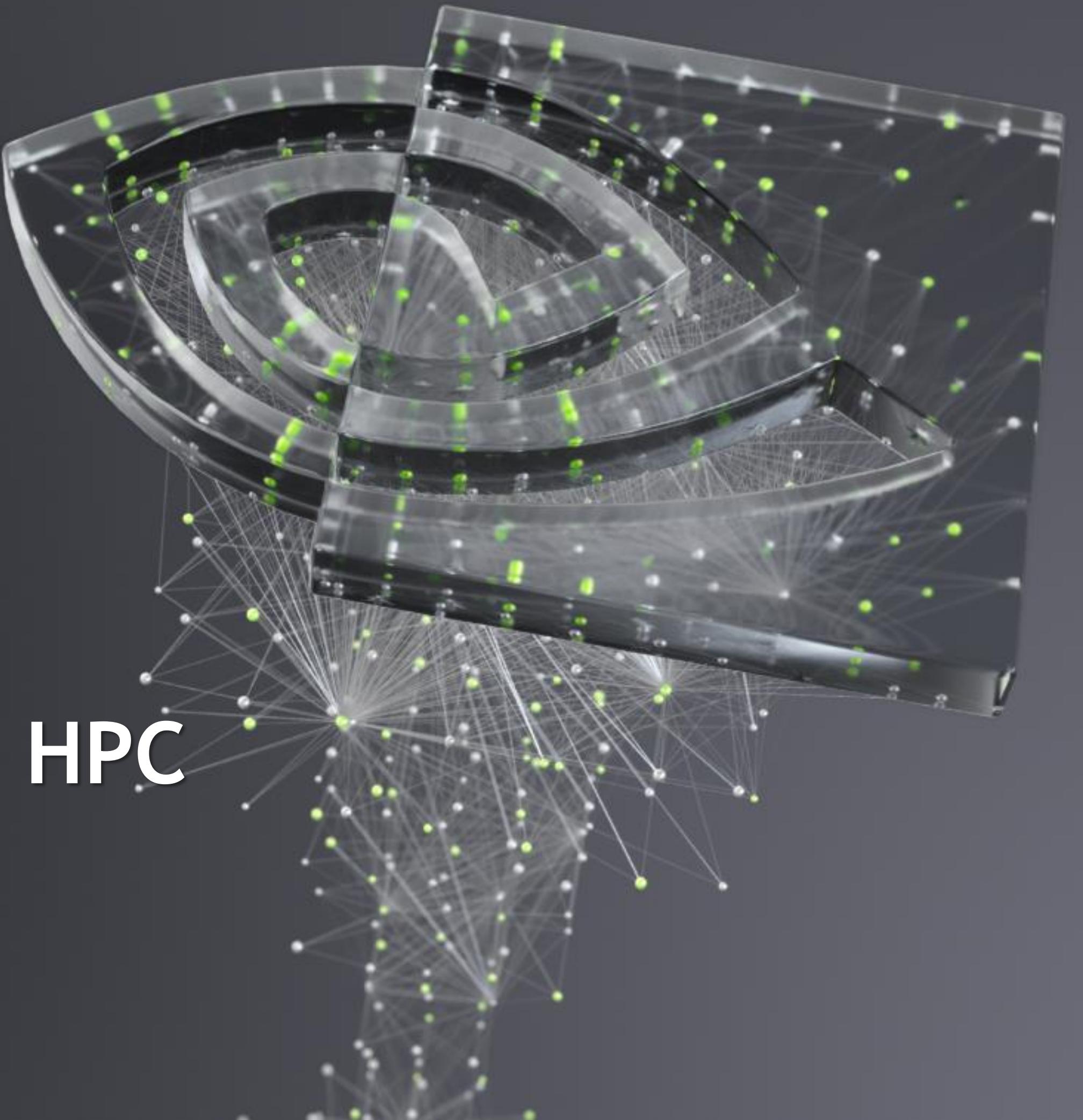


NVIDIA®

OPENMP IN NVIDIA'S HPC COMPILERS

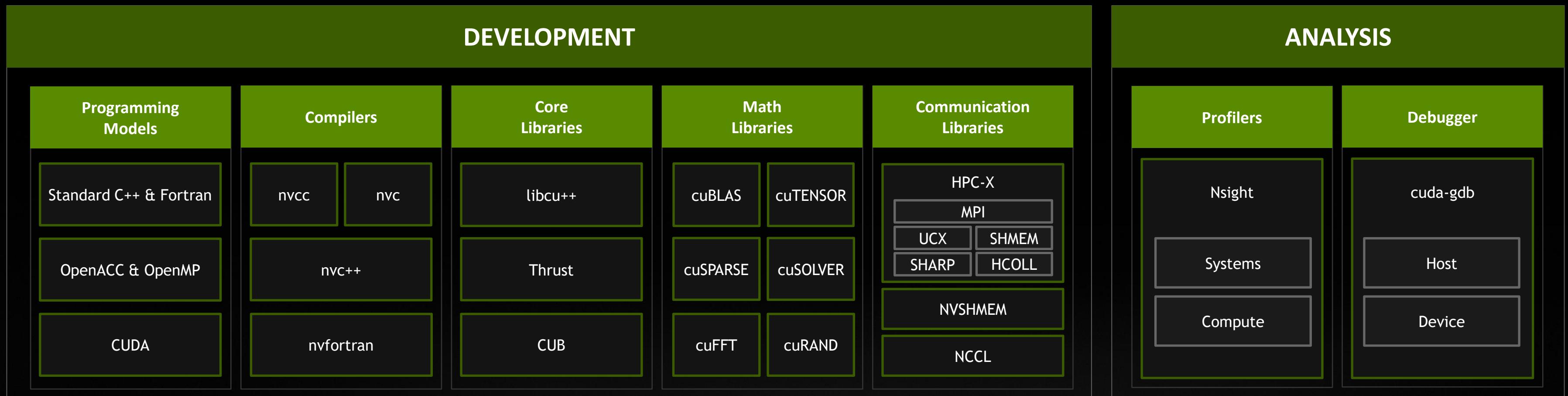
Jeff Larkin, HPC Architect

IWOMP 2021, September 2021



NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud



Develop for the NVIDIA Platform: GPU, CPU and Interconnect
Libraries | Accelerated C++ and Fortran | Directives | CUDA
7-8 Releases Per Year | Freely Available

PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

Accelerated Standard Languages

```
std::transform(par, x, x+n, y, y,
              [=] (float x, float y) { return y + a*x;
            });

do concurrent (i = 1:n)
    y(i) = y(i) + a*x(i)
enddo

import legate.numpy as np
...
def saxpy(a, x, y):
    y[:] += a*x
```

Incremental Portable Optimization

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
              [=] (float x, float y) {
                  return y + a*x;
            });
...
}

#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
              [=] (float x, float y) {
                  return y + a*x;
            });
...
}
```

Platform Specialization

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

int main(void) {
    ...
cudaMemcpy(d_x, x, ...);
cudaMemcpy(d_y, y, ...);

saxpy<<<(N+255)/256,256>>>(...);

cudaMemcpy(y, d_y, ...);
```

Core

Math

Communication

Data Analytics

AI

Quantum

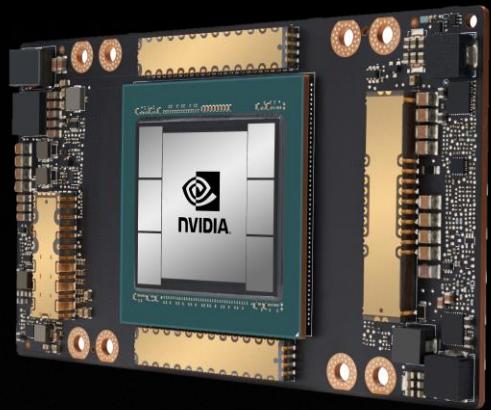
THE ROLE OF DIRECTIVES FOR PARALLEL PROGRAMMING

Directives convey additional information to the compiler.



NVIDIA HPC COMPILERS

NVC | NVC++ | NVFORTRAN



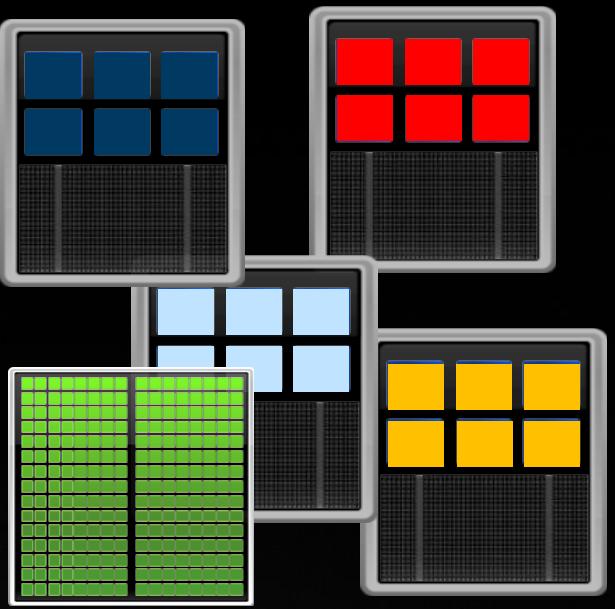
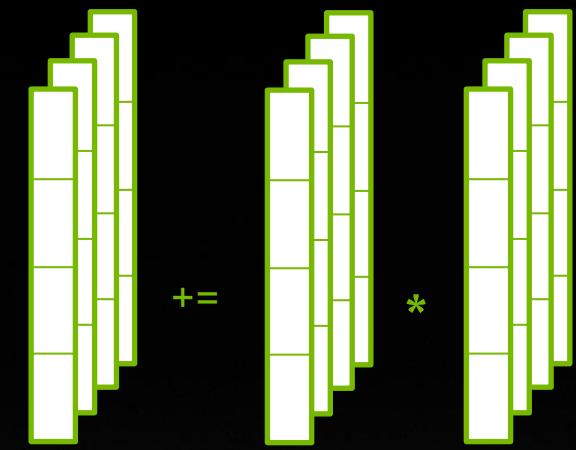
Accelerated
Latest GPUs
Automatic Acceleration

Fortran



OpenMP

Programmable
Standard Languages
Directives
CUDA



Multicore
Directives
Vectorization

Multi-Platform
x86_64
Arm
OpenPOWER

NVIDIA HPC COMPILER

Using OpenMP

- OpenMP
 - `-mp` → Enable OpenMP targeting Multicore
 - `-mp=gpu` → Enable OpenMP targeting GPU* and Multicore
- GPU Options
 - `-gpu=ccXX` → Set GPU target
- Compiler Diagnostics
 - `-Minfo=mp` → Compiler diagnostics for OpenMP
- Environment variable for NOTIFY
 - `export NVCOMPILER_ACC_NOTIFY = 1|2|3`

*Target directive support since HPC SDK 21.3.

OPENMP MODEL

OpenMP Execution Mapping to NVIDIA GPUs and Multicore

omp target

→ Starts Offload

omp teams

→ [GPU] CUDA Thread Blocks in grid
→ [CPU] num_teams(1)

omp parallel

→ [GPU] CUDA threads within thread block
→ [CPU] CPU threads

omp simd

→ [GPU] SIMDlen(1)
→ [CPU] Hint for vector instructions

BEST PRACTICES FOR OPENMP ON GPUS

Always use the **teams** and **distribute** directive to expose all available parallelism

Aggressively **collapse** loops to increase available parallelism

Use the **target data** directive and **map** clauses to reduce data movement between CPU and GPU

Use accelerated libraries whenever possible

Use OpenMP tasks to go asynchronous and better utilize the whole system

Use host fallback (**if** clause) to generate host and device code

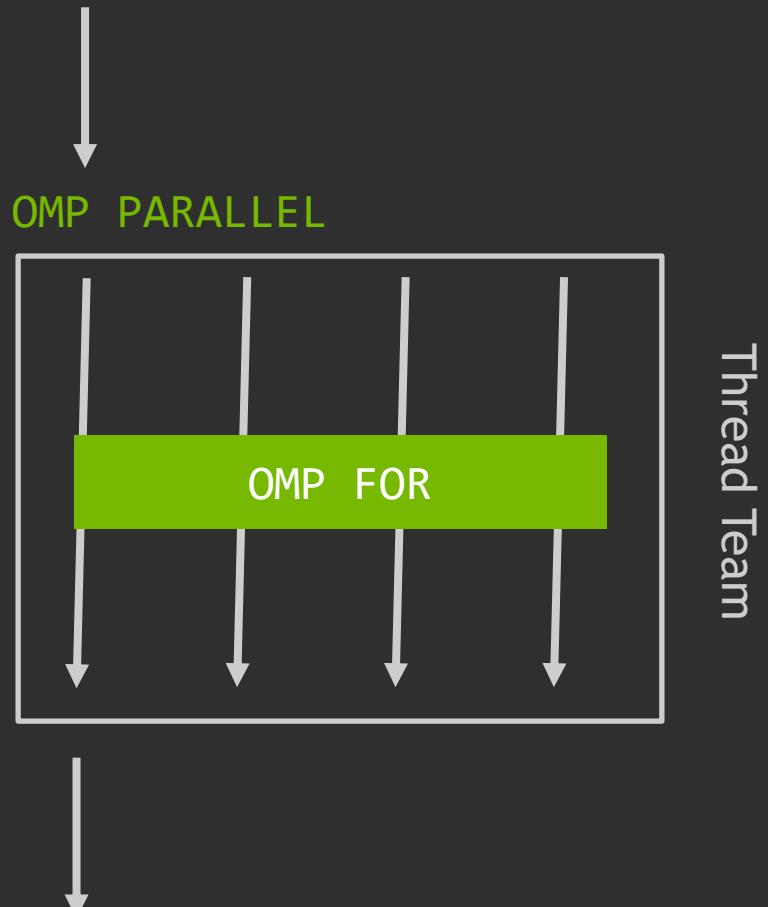
Expose More Parallelism Parallel For Isn't Enough

Many codes already have Parallel Worksharing loops (**Parallel For/Parallel Do**); Isn't that enough?

Parallel For creates a single contention group of threads with a shared view of memory and the ability to coordinate and synchronize.

This structure limits the degree of parallelism that a GPU can exploit and doesn't exploit many GPU advantages

```
#pragma omp parallel for reduction(max:error)
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]));
    }
}
```



Thread Team

Expose More Parallelism

Use Teams Distribute

The **Teams Distribute** directives exposes coarse-grained, scalable parallelism, generating more parallelism.

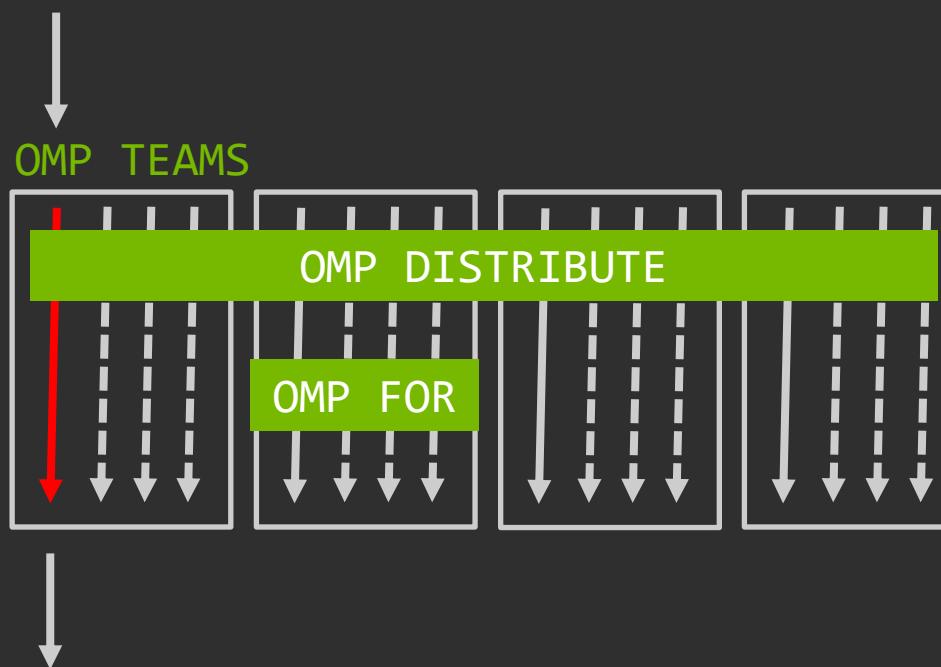
Parallel For is still used to engage the threads within each team.

Coarse and Fine-grained parallelism may be combined (A) or split (B).

How do I know which will be better for my code & machine?

```
#pragma omp target teams distribute \
    parallel for reduction(max:error)
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]));
    }
}
```

```
#pragma omp target teams distribute \
    reduction(max:error)
for( int j = 1; j < n-1; j++ ) {
#pragma parallel for reduction(max:error)
    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]));
    }
}
```



Expose Even More Parallelism!

Use Collapse Clause

Aggressively

Collapsing loops increases the parallelism available for the compiler to exploit.

It's likely possible for this code to perform equally well to parallel for on the CPU and still exploit more parallelism on a GPU.

Collapsing does erase possible gains from locality (maybe **tile** can help?)

Not all loops can be collapsed

```
#pragma omp target teams distribute \
    parallel for reduction(max:error) \
    collapse(2)
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]));
    }
}
```

Optimize Data Motion

Use Target Data Mapping

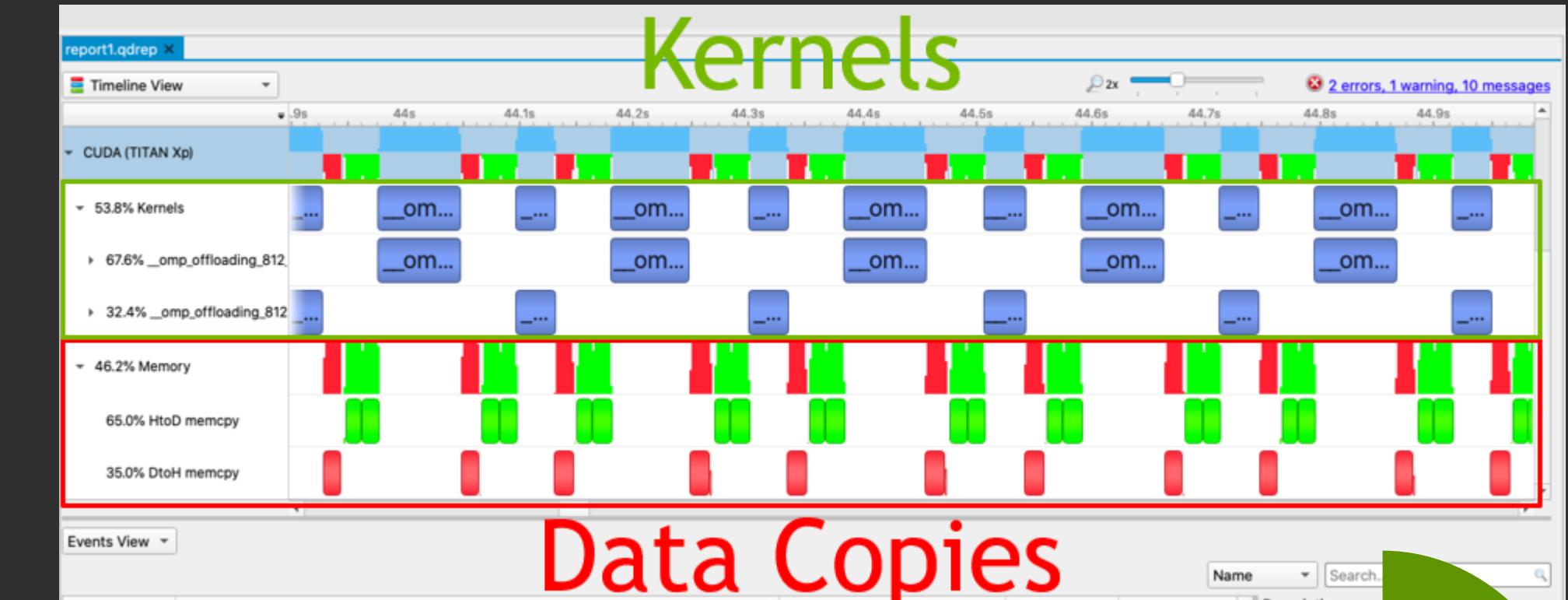
The more loops you move to the device, the more data movement you may introduce.

Compilers will always choose correctness over performance, so the programmer can often do better.

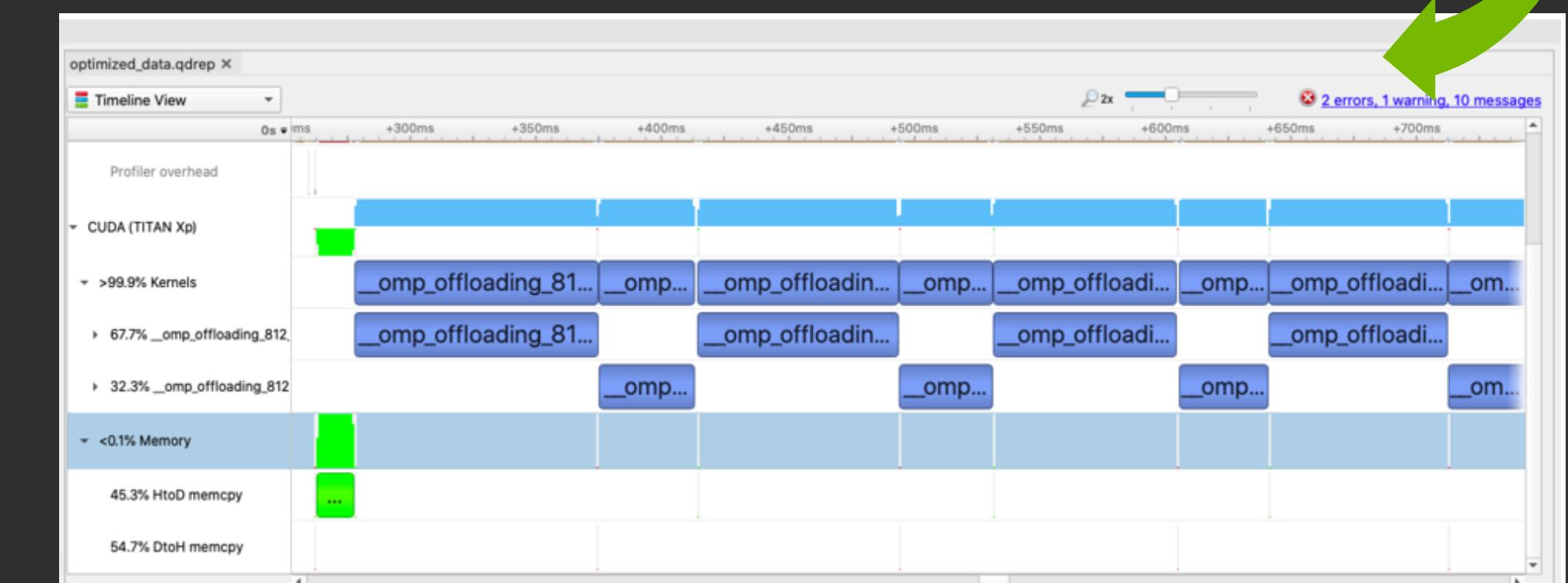
Target Data directives and **map** clauses enable the programmer to take control of data movement.

On **Unified Memory** machines, locality may be *good enough*, but optimizing the data movement is more portable.

Note: I've seen cases where registering arrays with CUDA may further improve remaining data movement.



```
#pragma omp target data map(to:Anew) map(tofrom:A)
while ( error > tol && iter < iter_max )
{
    ... // Use A and Anew in target directives
}
```



Using Managed Memory

NVIDIA compilers can automatically translate your dynamic data allocations to managed memory!

You don't need explicit target data

Note: Requires directive not supported at this time.

```
real(8), allocatable :: x(:), y(:)
!$omp target data map(x,y)
    !$omp target teams loop
    do i = 1, n
        call compute(x,y)
    enddo
    !$omp target teams loop
    do i = 1, n
        call compute2(x,y)
    enddo
 !$omp end target data
```

```
$ nvfortran test.c -mp=gpu -gpu=managed -Minfo=mp
```

Accelerated Libraries & OpenMP

Eat your free lunch

Accelerated libraries are free
performance

The `use_device_ptr` clause enables
sharing data from OpenMP to CUDA.

OpenMP 5.1 adds the interop construct
to enable sharing CUDA Streams

```
#pragma omp target data map(alloc:x[0:n],y[0:n])
{
    #pragma omp target teams distribute parallel for
    for( i = 0; i < n; i++)
    {
        x[i] = 1.0f;
        y[i] = 0.0f;
    }

    #pragma omp target data use_device_ptr(x,y)
    {
        cublasSaxpy(n, 2.0, x, 1, y, 1);
        // Synchronize before using results
    }
    #pragma omp target update from(y[0:n])
}
```

Tasking & GPUs

Keep the whole system busy

Data copies, GPU Compute, and CPU compute can all overlap.

OpenMP uses its existing tasking framework to manage asynchronous execution and dependencies.

Your mileage may vary regarding how well the runtime maps OpenMP tasks to CUDA streams.

Get it working synchronously first!

```
#pragma omp target data map(alloc:a[0:N],b[0:N])
{
#pragma omp target update to(a[0:N]) nowait depend(inout:a)
#pragma omp target update to(b[0:N]) nowait depend(inout:b)
#pragma omp target teams distribute parallel for \
    nowait depend(inout:a)
    for(int i=0; i<N; i++ )
    {
        a[i] = 2.0f * a[i];
    }
#pragma omp target teams distribute parallel for \
    nowait depend(inout:a,b)
    for(int i=0; i<N; i++ )
    {
        b[i] = 2.0f * a[i];
    }
#pragma omp target update from(b[0:N]) nowait depend(inout:b)
#pragma omp taskwait depend(inout:b)
}
```

Host Fallback

The “if” clause

The **if** clause can be used to selectively turn off OpenMP offloading.

Unless the compiler knows at compile-time the value of the **if** clause, both code paths are generated

For simple loops, written to use the previous guidelines, it may be possible to have unified CPU & GPU code (no guarantee it will be optimal)

```
#pragma omp target teams distribute \
parallel for reduction(max:error) \
collapse(2) if(target:USE_GPU)
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]));
    }
}
```

The Loop Directive

Give descriptiveness a try

The Loop directive was added in 5.0 as a descriptive option for programming. Loop asserts the ability of a loop to be run in any order, including concurrently.

As more compilers & applications adopt it, we hope it will enable more performance portability.

```
#pragma omp target teams loop \
reduction(max:error) \
collapse(2)
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]));
    }
}
```

START OFFLOADING ‘OMP LOOP’

Three Ways

1. `omp target teams loop`

- Recommended way
- You can use `num_teams` and `thread_limit` clauses

2. `omp target loop`

- Fully automatic
- You cannot use `num_teams` / `thread_limit`

3. `omp target parallel loop`

- Uses only threads, and doesn’t use teams
- Might be useful for light kernels

BEST PRACTICES FOR OPENMP ON GPUS

Always use the **teams** and **distribute** directive to expose all available parallelism

Aggressively **collapse** loops to increase available parallelism

Use the **target data** directive and **map** clauses to reduce data movement between CPU and GPU

Use accelerated libraries whenever possible

Use OpenMP tasks to go asynchronous and better utilize the whole system

Use host fallback (**if** clause) to generate host and device code

Bonus: Give **loop** a try