OpenMP on GPUs, First Experiences and Best Practices

Jeff Larkin, GTC2018 S8344, March 2018

NVIDIA.

AGENDA

What is OpenMP?

OpenMP Target Directives

Parallelizing for GPUs

Target Data Directives

Interoperability with CUDA

Asynchronous Data Movement

Best Practices

History of OpenMP

OpenMP is the defacto standard for directive-based programming on shared memory parallel machines

First released in 1997 (Fortran) and 1998 (C/C++), Version 5.0 is expected later this year

Beginning with version 4.0, OpenMP supports offloading to accelerator devices (non-shared memory)

In this session, I will be showing OpenMP 4.5 with the CLANG and XL compilers offloading to NVIDIA GPUs.

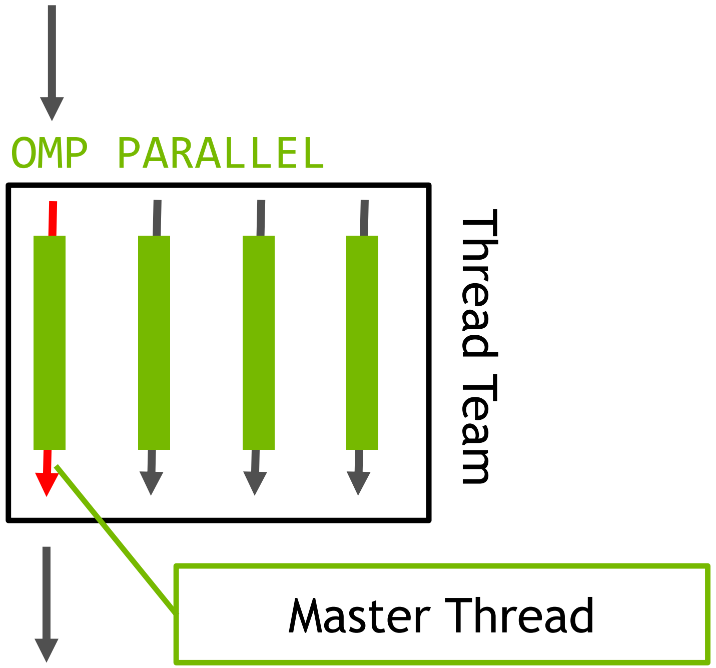
OPENMP EXAMPLE

```
error = 0.0;

#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]));
        }
    }
```

Create a team of threads and workshare this loop across those threads.

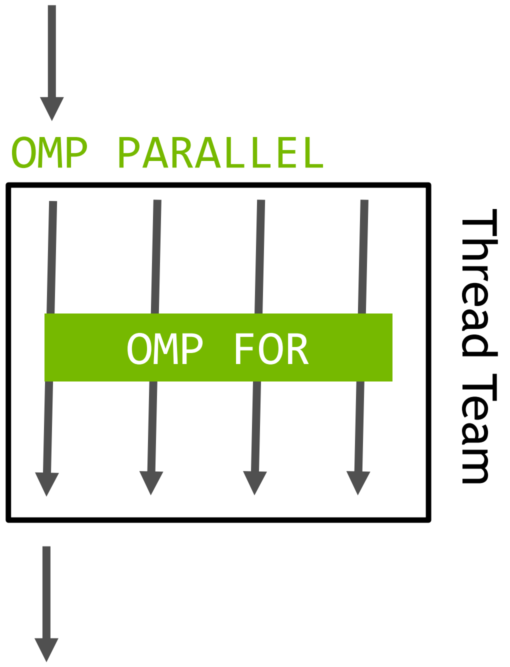
OPENMP WORKSHARING

- PARALLEL Directive
- Spawns a *team* of *threads*
- Execution continues <u>redundantly</u> on all threads of the team.
- All threads join at the end and the *master* thread continues execution.

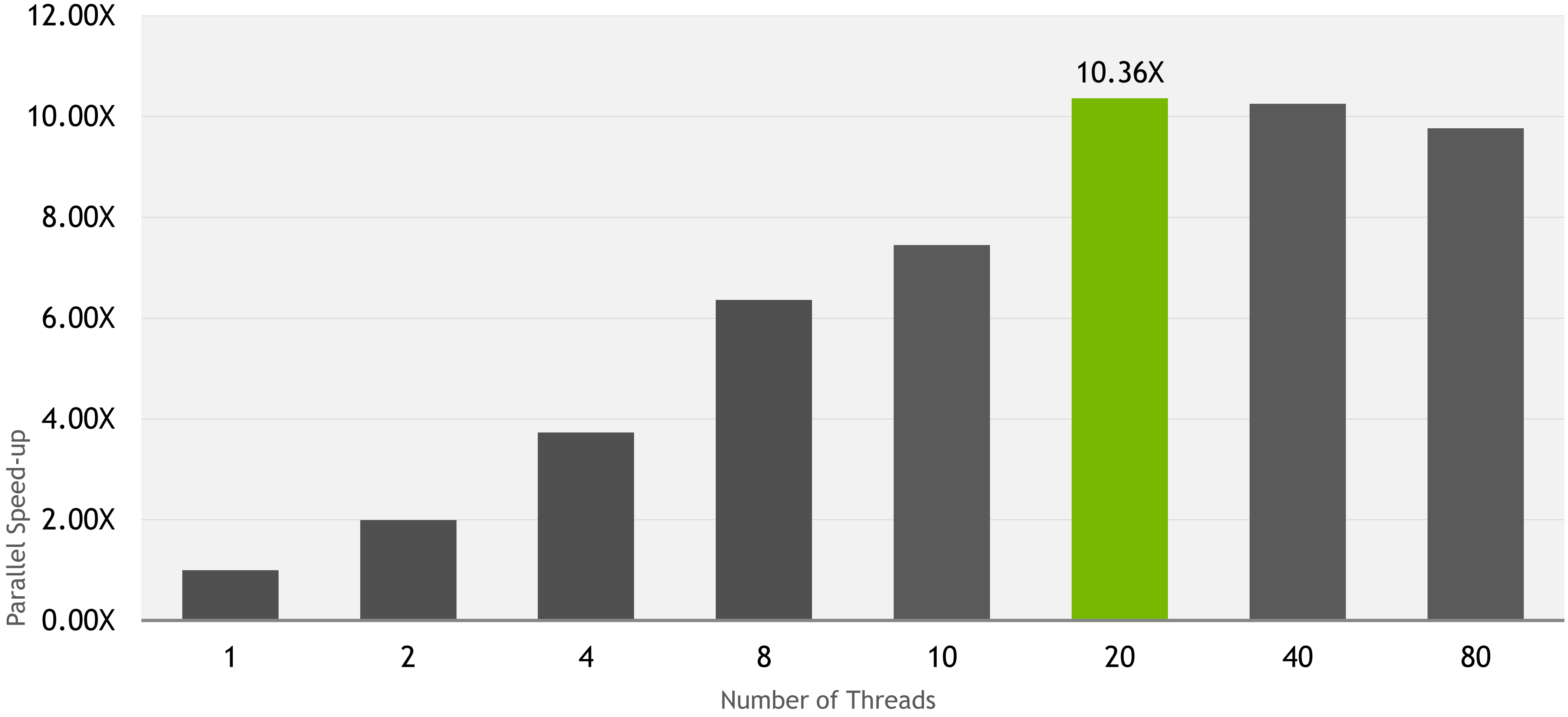


OPENMP WORKSHARING

- FOR/DO (Loop) Directive
- Divides ("*workshares*") the iterations of the next loop across the threads in the team
- How the iterations are divided is determined by a *schedule*.



CPU Threading Results



Source: Power8 CPU, Clang 3.8.0

GPU OFFLOADING COMPILER SUPPORT

CLANG – Open-source compiler, industry collaboration

XL – IBM Compiler Suite for P8/P100 and P9/V100

Cray Compiler Environment (CCE) – Only available on Cray machines

GCC – On-going work to integrate

OPENMP TARGET DIRECTIVES

The target directives provide a mechanism to move the thread of execution from the CPU to another device, also relocating required data.

Almost all of OpenMP can be used within a target region, but only a limited subset makes sense on a GPU.

OPENMP TARGET EXAMPLE

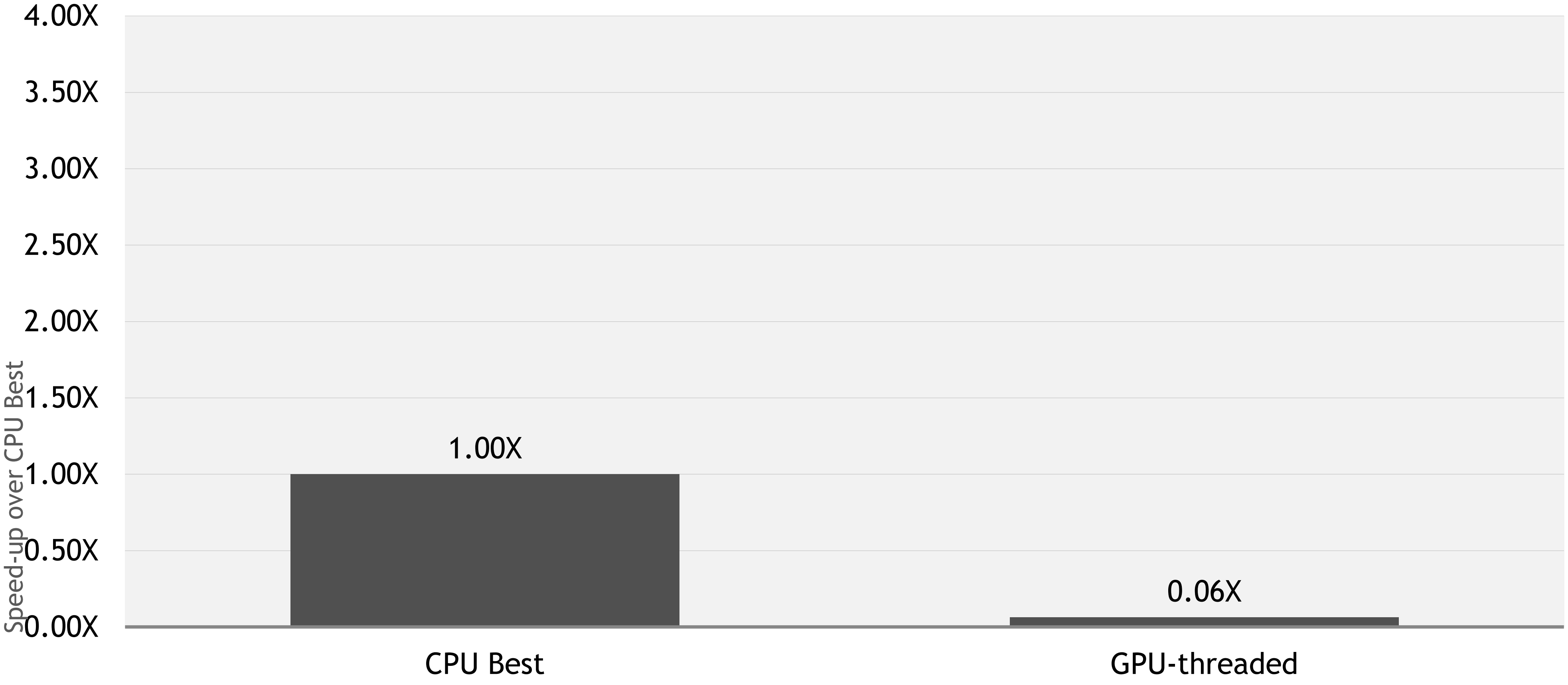
```
#pragma omp target
{
error = 0.0;

#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]));
        }
    }
}
```

Relocate execution to the *target device*

All scalars used in the target region will be made *firstprivate*.

All arrays will be copied *to* and *from* the device.

Offloading Performance



Source: Power8 CPU + NVIDIA Tesla P100, Clang 3.8.0

WHAT WENT WRONG?

OpenMP was originally designed for threading on a shared memory parallel computer, so the parallel directive only creates a single level of parallelism.

Threads must be able to synchronize (for, barrier, critical, master, single, etc.), which means on a GPU they will use 1 thread block

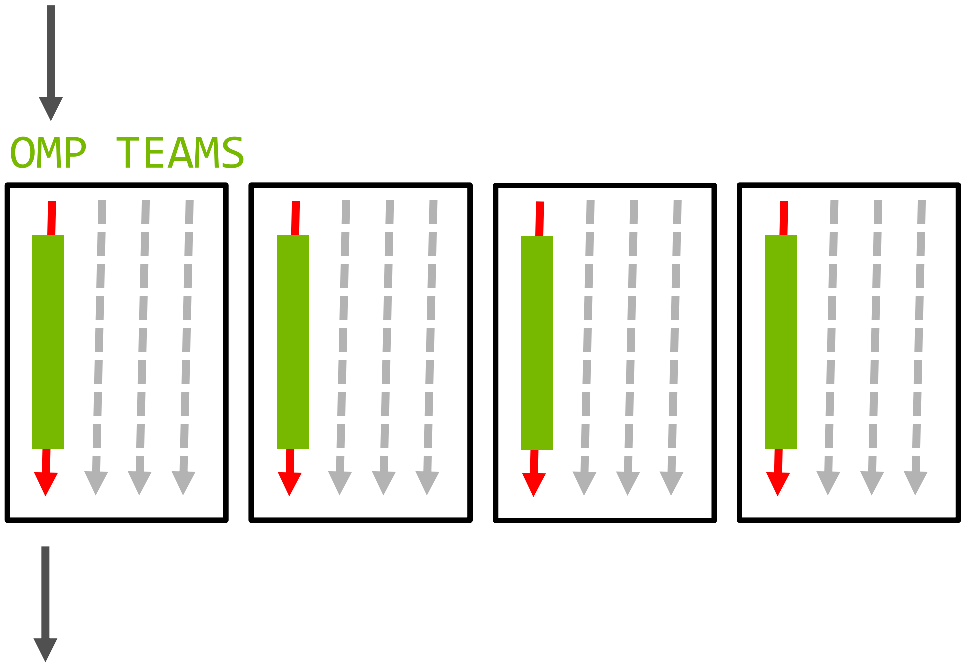
The *teams* directive was added to express a second level of scalable parallelism

OPENMP TEAMS

- TEAMS Directive

- To better utilize the GPU resources, use many thread teams via the TEAMS directive.

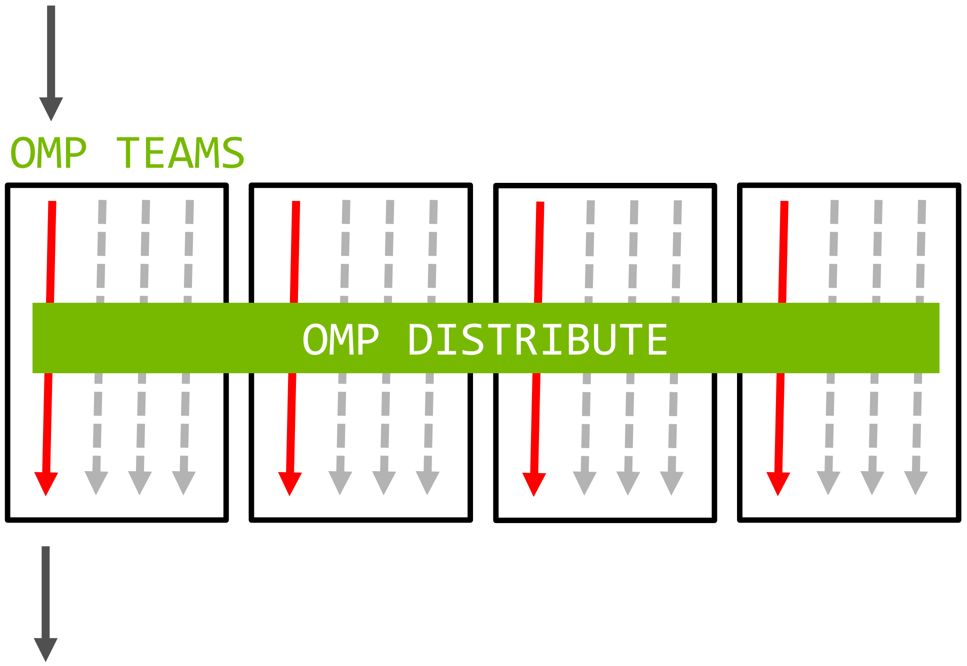
  - Spawns 1 or more thread teams with the same number of threads
  - Execution continues on the master threads of each team (redundantly)
  - No synchronization between teams



OPENMP TEAMS

- DISTRIBUTE Directive
- Distributes the iterations of the next loop to the master threads of the teams.

  - Iterations are distributed statically.
  - There's no guarantees about the order teams will execute.
  - No guarantee that all teams will execute simultaneously
  - Does not generate parallelism/worksharing within the thread teams.

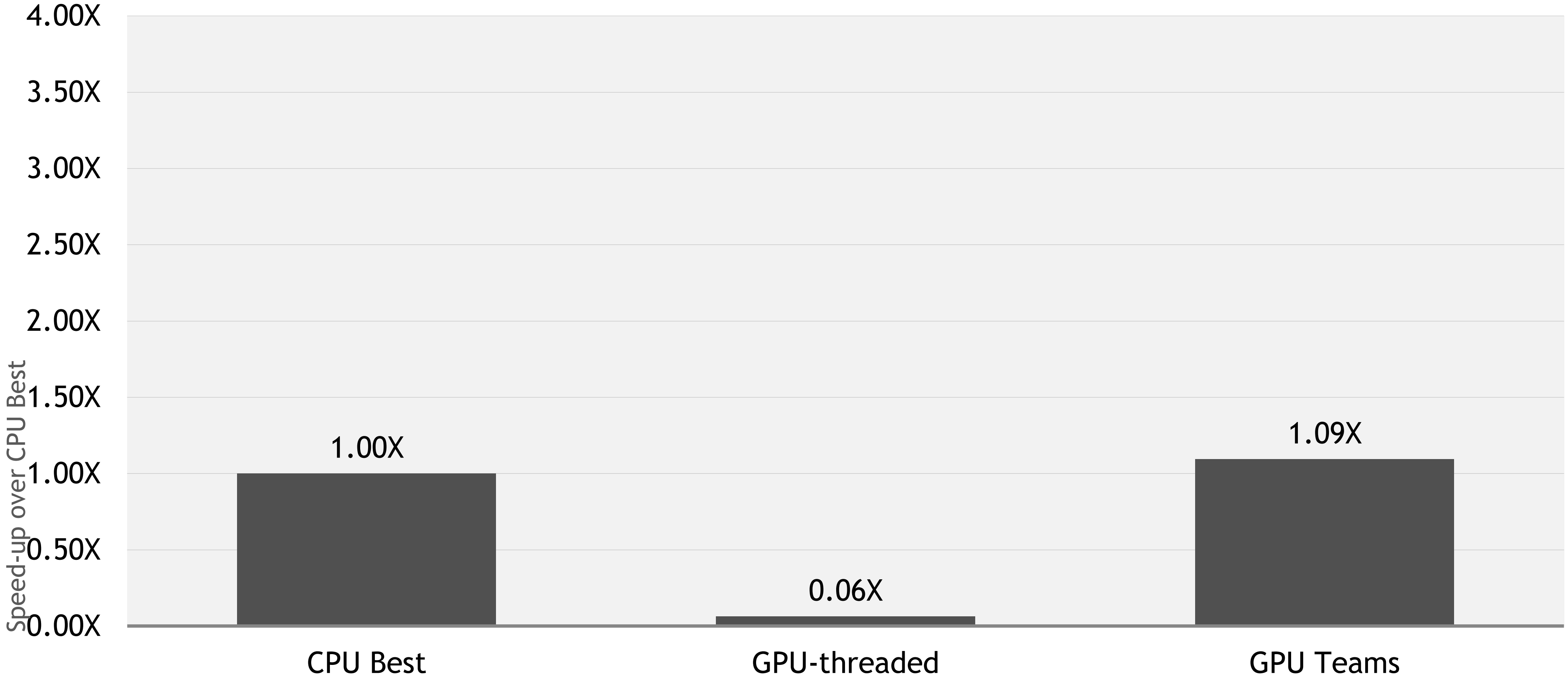


OPENMP TARGET TEAMS EXAMPLE

```
error = 0.0;

#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]));
        }
    }
```

Relocate execution to the *target device*, generate teams, distribute loop to teams, and workshare.

Offloading Performance



Source: Power8 CPU + NVIDIA Tesla P100, Clang 3.8.0

LESSON LEARNED

When writing OpenMP for GPUs, always use teams and distribute to spread parallelism across the full GPU.

Can we do better?

INCREASING PARALLELISM

Currently all of our parallelism comes from the outer loop, can we parallelize the inner one too?

Three possibilities

- Split Teams Distribute from Parallel For
- Collapse clause

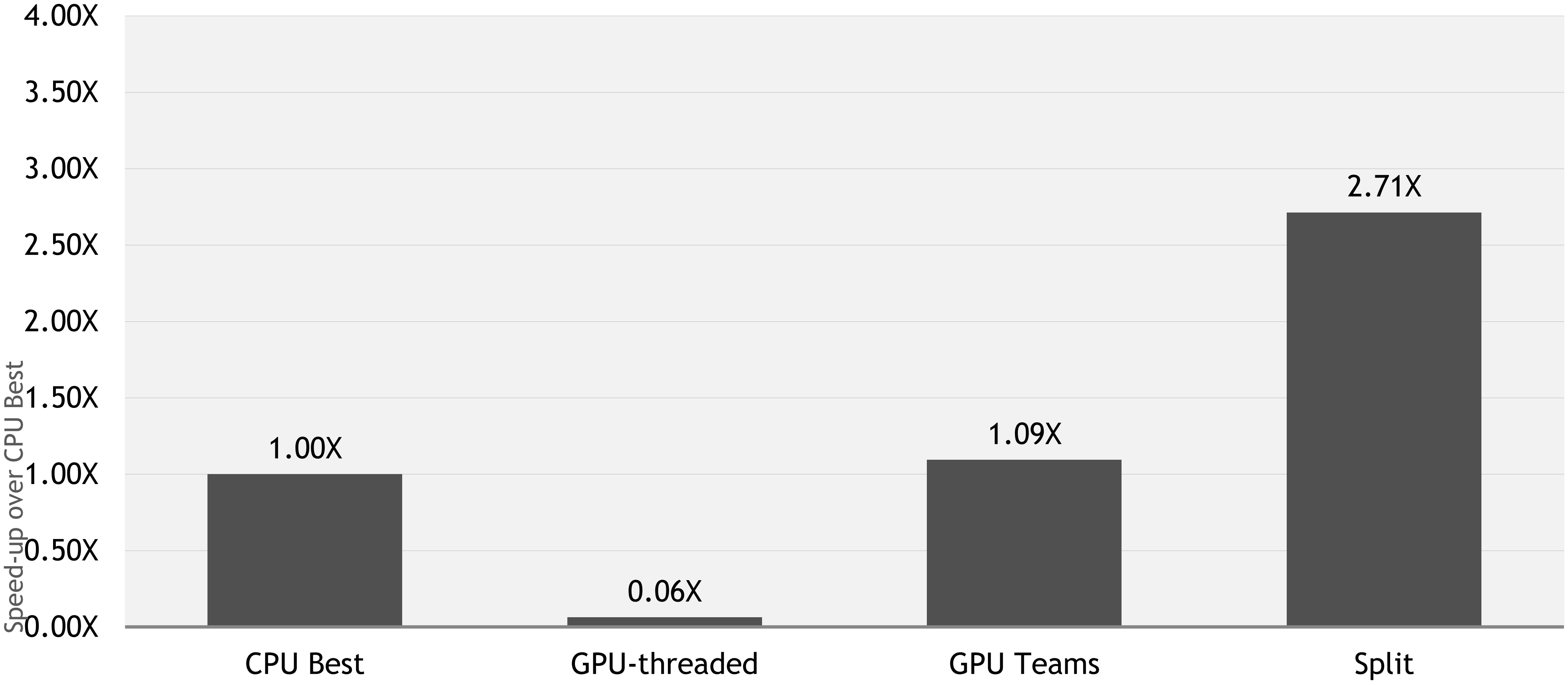
OPENMP TARGET TEAMS EXAMPLE

```
error = 0.0;

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

Distribute outer loop to thread teams.

Workshare inner loop across threads.

Offloading Performance



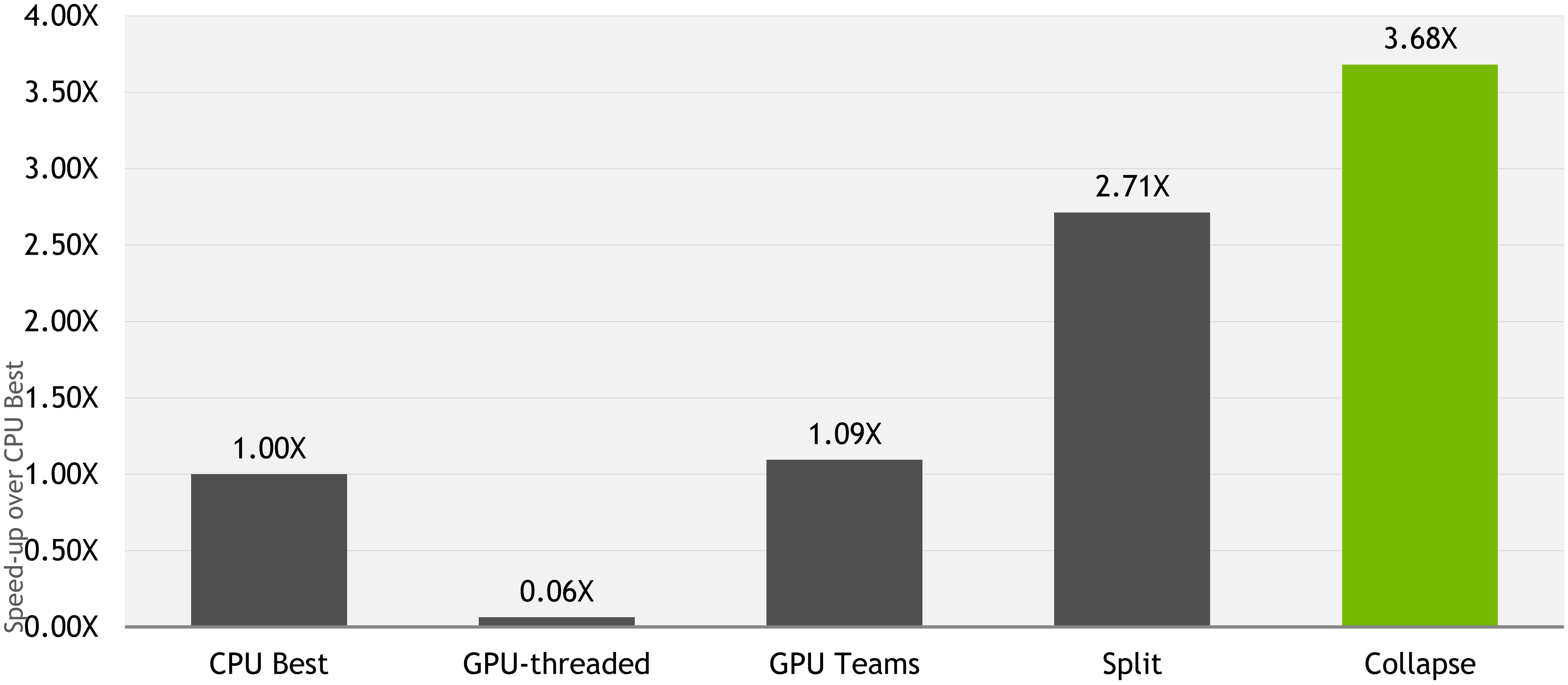
Source: Power8 CPU + NVIDIA Tesla P100, Clang 3.8.0

OPENMP TARGET TEAMS EXAMPLE

```
error = 0.0;

#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]));
        }
    }
```

Collapse the two loops before applying both teams and thread parallelism to both

Offloading Performance



Source: Power8 CPU + NVIDIA Tesla P100, Clang 3.8.0

TARGET DATA DIRECTIVES

Moving data between the CPU and GPU at every loop is inefficient

The target data directive and map clause enable control over data movement.

map(<options>)…

to – Create space on the GPU and copy input data

from – Create space on the GPU and copy output data

tofrom – Create space on the GPU and copy input and output data

alloc – Create space on the GPU, do not copy data

TARGET DATA EXAMPLE

Move the data outside of the convergence loop to share data in the two target regions

```
#pragma omp target data map(to:Anew) map(A)
while ( error > tol && iter < iter_max )
{
  error = 0.0;
  #pragma omp target teams distribute parallel for \
              reduction(max:error) map(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 parallel for
  for( int j = 1; j < n-1; j++)
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
  }
  if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
  iter++;
}
```

OPENMP HOST FALLBACK

```
error = 0.0;

#pragma omp target teams distribute \
            parallel for reduction(max:error) collapse(2) \
            if(n > 100)
    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 if clause defers the decision of where to run the loops until runtime and forces building both a host and device version.

CUDA INTEROPERABILITY

OpenMP is a high-level language, sometimes low level optimizations will be necessary for best performance.

CUDA Kernels or Accelerated libraries good examples

The use_device_ptr map type allows OpenMP device arrays to be passed to CUDA or accelerated libraries.

The is_device_ptr map clause allows CUDA arrays to be used within OpenMP target regions

EXAMPLE OF USE_DEVICE_PTR

```
#pragma omp target data map(alloc:x[0:n]) map(from: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);
   }
 }
```

Manage data movement using map clauses

Expose the device arrays to CUBLAS

EXAMPLE OF USE_DEVICE_PTR

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

set(n,1.0f,x);
set(n,0.0f,y);

saxpy(n, 2.0, x, y);
cudaMemcpy(&tmp,y,(size_t)sizeof(float),cudaMemcpyDeviceToHost);
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
  #pragma omp target teams distribute
              parallel for is_device_ptr(x,y)
    for(int i=0; i<n; i++)
      y[i] += a*x[i];
}
```

Manage data using CUDA

Use CUDA arrays within OpenMP region.

OPENMP TASKS

OpenMP tasks allow the programmer to represent independent blocks of work and allow the runtime to schedule them

All OpenMP target regions are tasks

- By default, synchronous with the host
- Can be made asynchronous with the nowait clause
- Can accept the depend clause to interact with other tasks

Using OpenMP's nowait and depend clauses, it's possible to do asynchronous data transfers and kernel launches to improve system utilization

ASYNCHRONOUS PIPELINING EXAMPLE

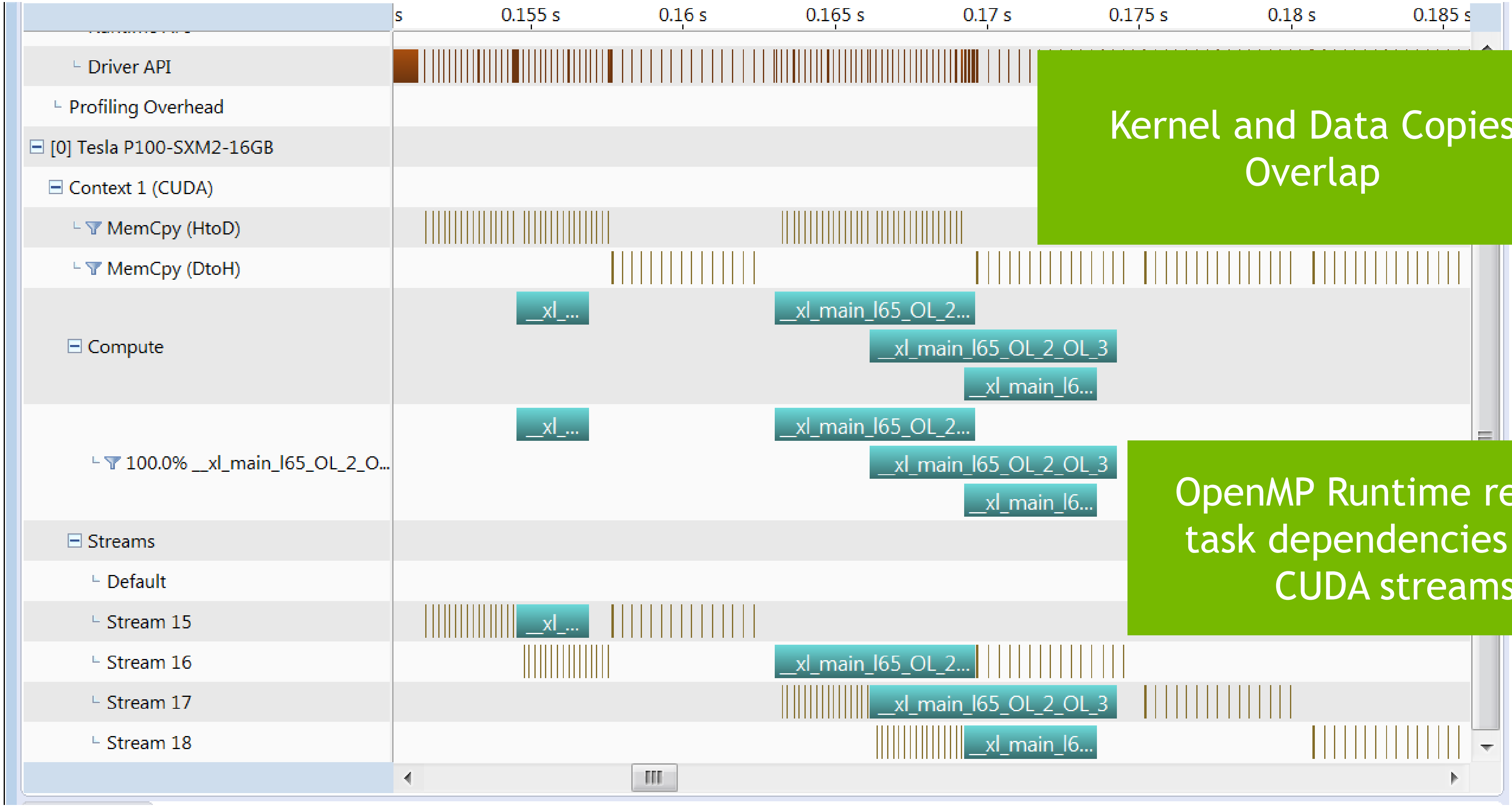
```
#pragma omp target data map(alloc:image[0:WIDTH*HEIGHT])
  for(block = 0; block < num_blocks; block++ ) {
    int start = block * (HEIGHT/num_blocks),
        end   = start + (HEIGHT/num_blocks);

#pragma omp target teams distribute \
            parallel for simd collapse(2) \
            depend(inout:image[block*block_size]) nowait
    for(int y=start;y<end;y++) {
      for(int x=0;x<WIDTH;x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
      }
    }

#pragma omp target update from(image[block*block_size:block_size])\
            depend(inout:image[block*block_size]) nowait
  }
#pragma omp taskwait
```

Launch kernel asynchronously, annotating the dependency

Target update launches asynchronously

Wait on all tasks (from the CPU)

ASYNCHRONOUS PIPELINING EXAMPLE



Kernel and Data Copies Overlap

OpenMP Runtime resolved task dependencies into 4 CUDA streams.

OPENMP TASK GRAPH & CUDA STREAMS

CUDA Streams

Simple mapping to the hardware

Developer maps the dependencies to CUDA streams explicitly

OpenMP Task Graph

Potentially more expressive

Task graph must be mapped to streams by the runtime.

Developer expresses the dependencies between different tasks

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 to generate host and device code