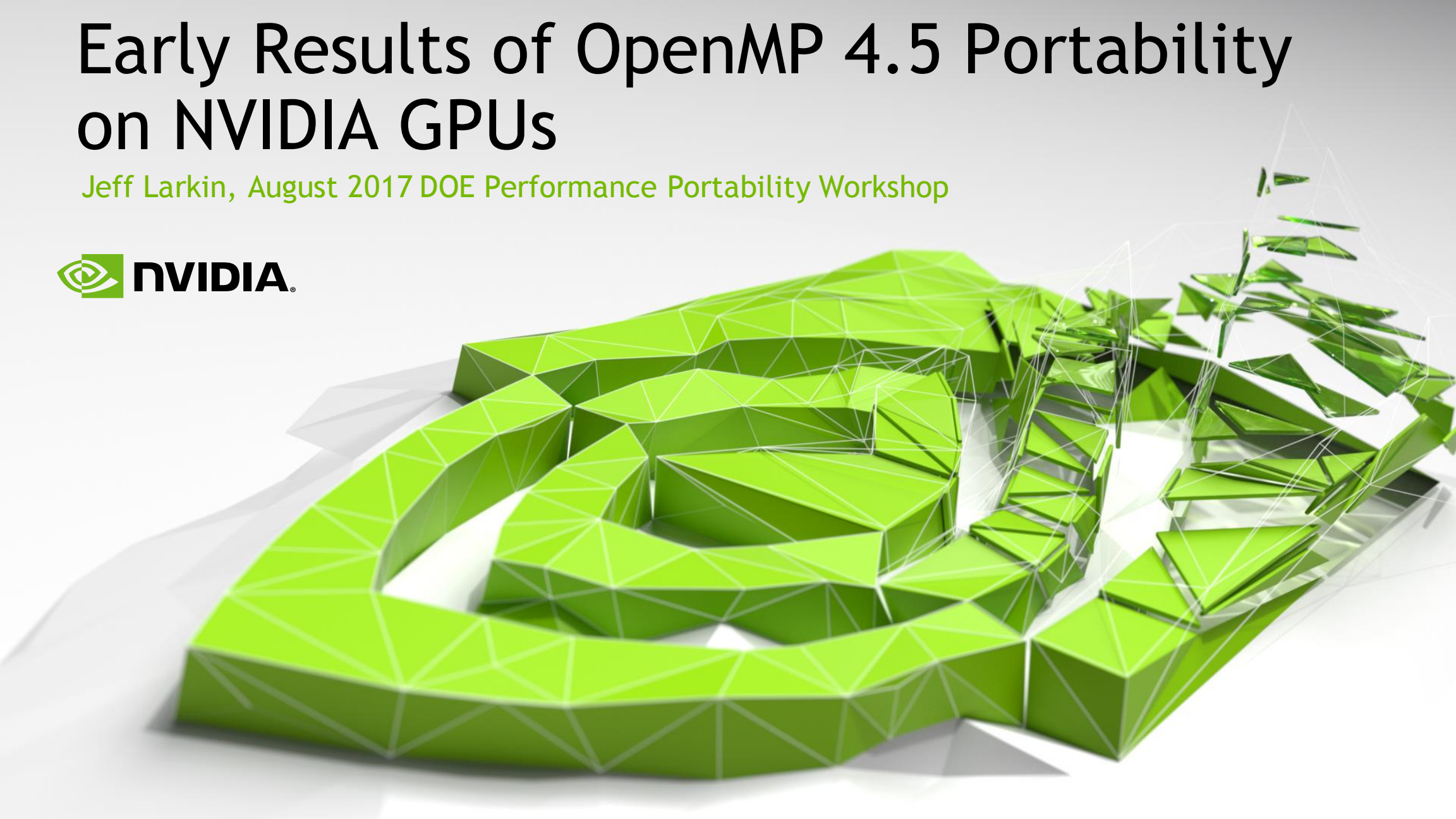
Early Results of OpenMP 4.5 Portability on NVIDIA GPUs

Jeff Larkin, August 2017 DOE Performance Portability Workshop



Background

Since the last performance portability workshop, several OpenMP implementations for NVIDIA GPUs have emerged or matured

As of August 2017, can these implementations deliver on performance, portability, and performance portability?

- Will OpenMP Target code be portable between compilers?
- Will OpenMP Target code be portable with the host?

I will compare results using 4 compilers: CLANG, Cray, GCC, and XL

OpenMP In Clang

Multi-vendor effort to implement OpenMP in Clang (including offloading)

Runtime based on open-sourced runtime from Intel.

Current status: much improved since last year!

Version used: clang/20170629

Compiler Options:

```
-O2 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda --cuda-path=$CUDA_HOME
```

OpenMP In Cray

Due to its experience with OpenACC, Cray's OpenMP 4.x compiler was the first to market for NVIDIA GPUs.

Observation: Does not adhere to OpenMP as strictly as the others.

Version used: 8.5.5

Compiler Options: None Required

Note: Cray performance results were obtained on an X86 + P100 system, unlike the other compilers. Only GPU performance is being compared.

OpenMP In GCC

Open-source GCC compiler with support for OpenMP offloading to NVIDIA GPUs

Runtime also based on open-sourced runtime from Intel

Current status: Mature on CPU, Very immature on GPU

Version used: 7.1.1 20170718 (experimental)

Compiler Options:

```
-O3 -fopenmp -foffload="-lm"
```

OpenMP In XL

IBM’s compiler suite, which now includes offloading to NVIDIA GPUs.

Same(ish) runtime as CLANG, but compilation by IBM’s compiler

Version used: xl/20170727-beta

Compiler Options:

-O3 -qsmp -qoffload

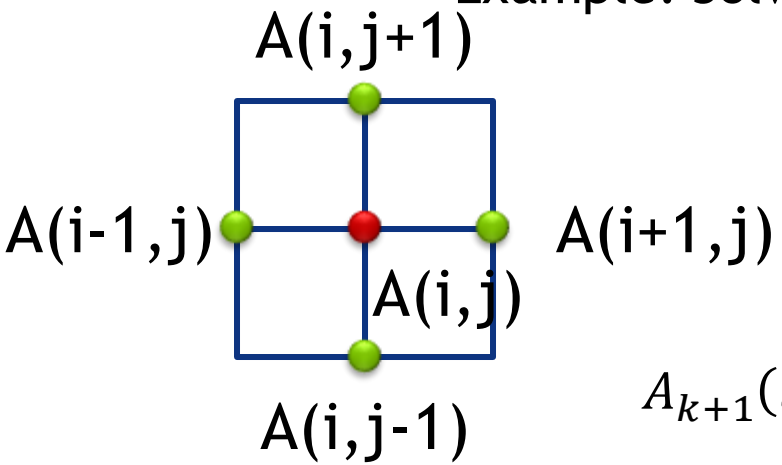
Case Study: Jacobi Iteration

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: $\nabla^2 f(x, y) = 0$



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

Teams & Distribute

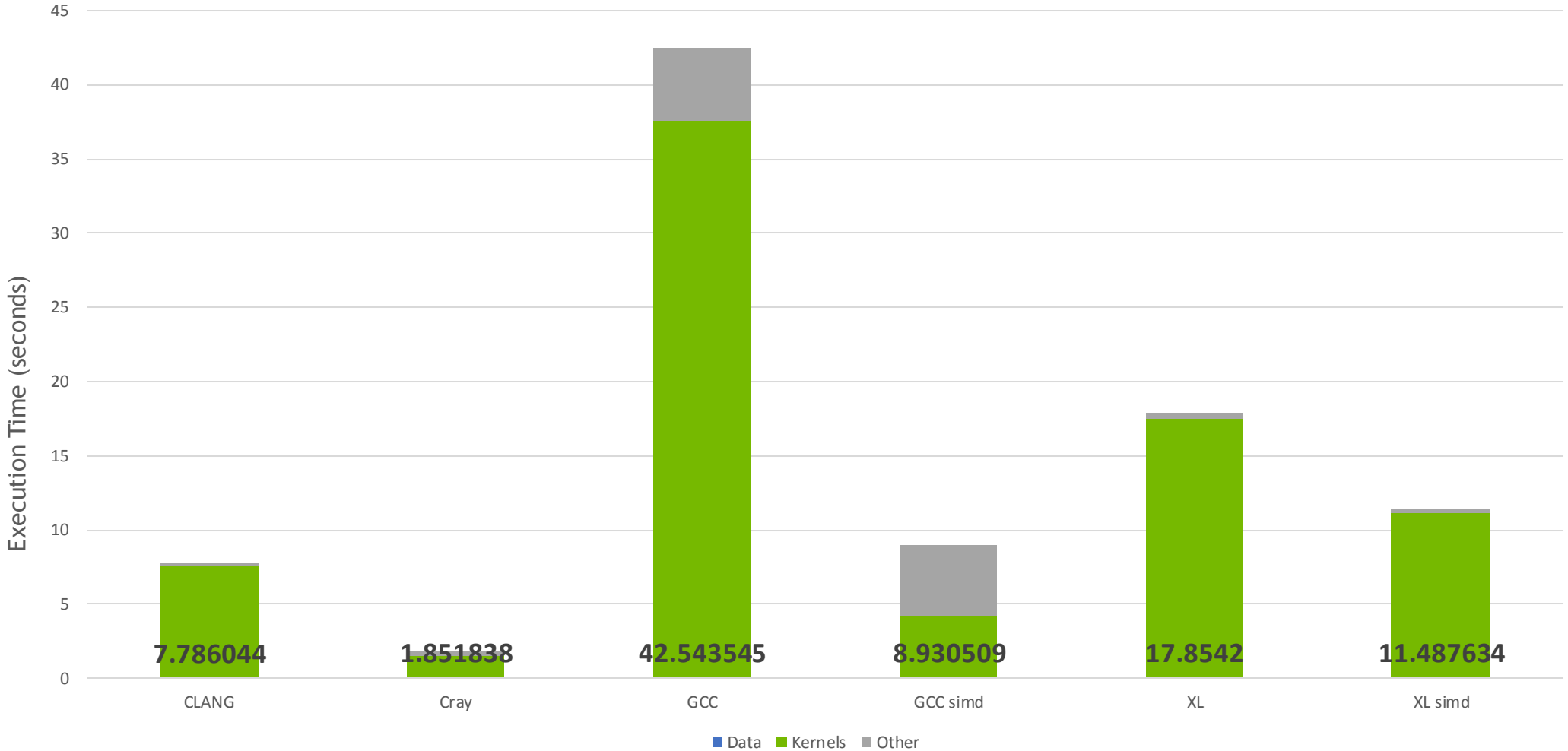
Teaming Up

```
#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++;
    }
```

Explicitly maps arrays for the entire while loop.

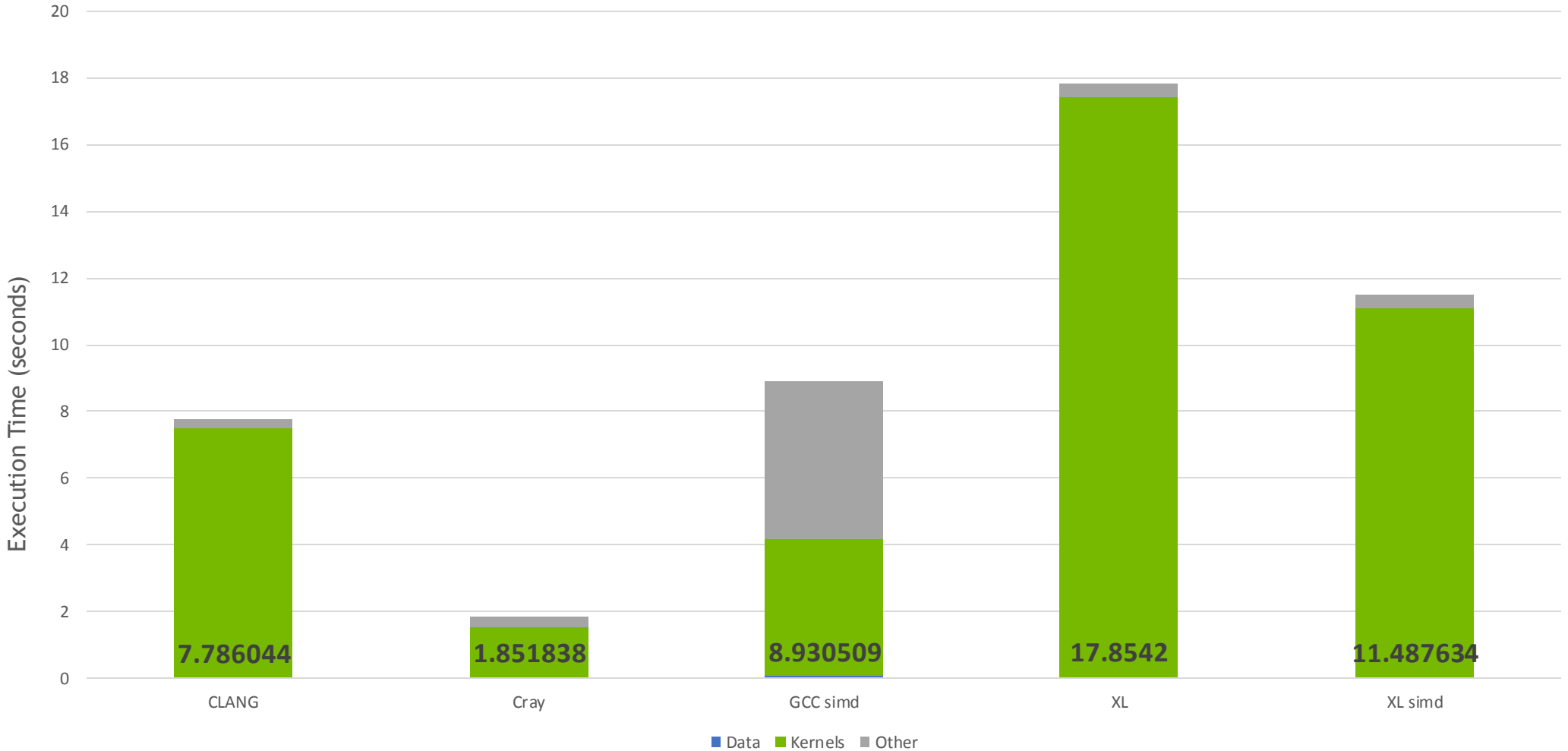
- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

Increasing Parallelism

Increasing Parallelism

Currently both our distributed and workshared parallelism comes from the same loop.

- We could collapse them together
- We could move the PARALLEL to the inner loop

The COLLAPSE(N) clause

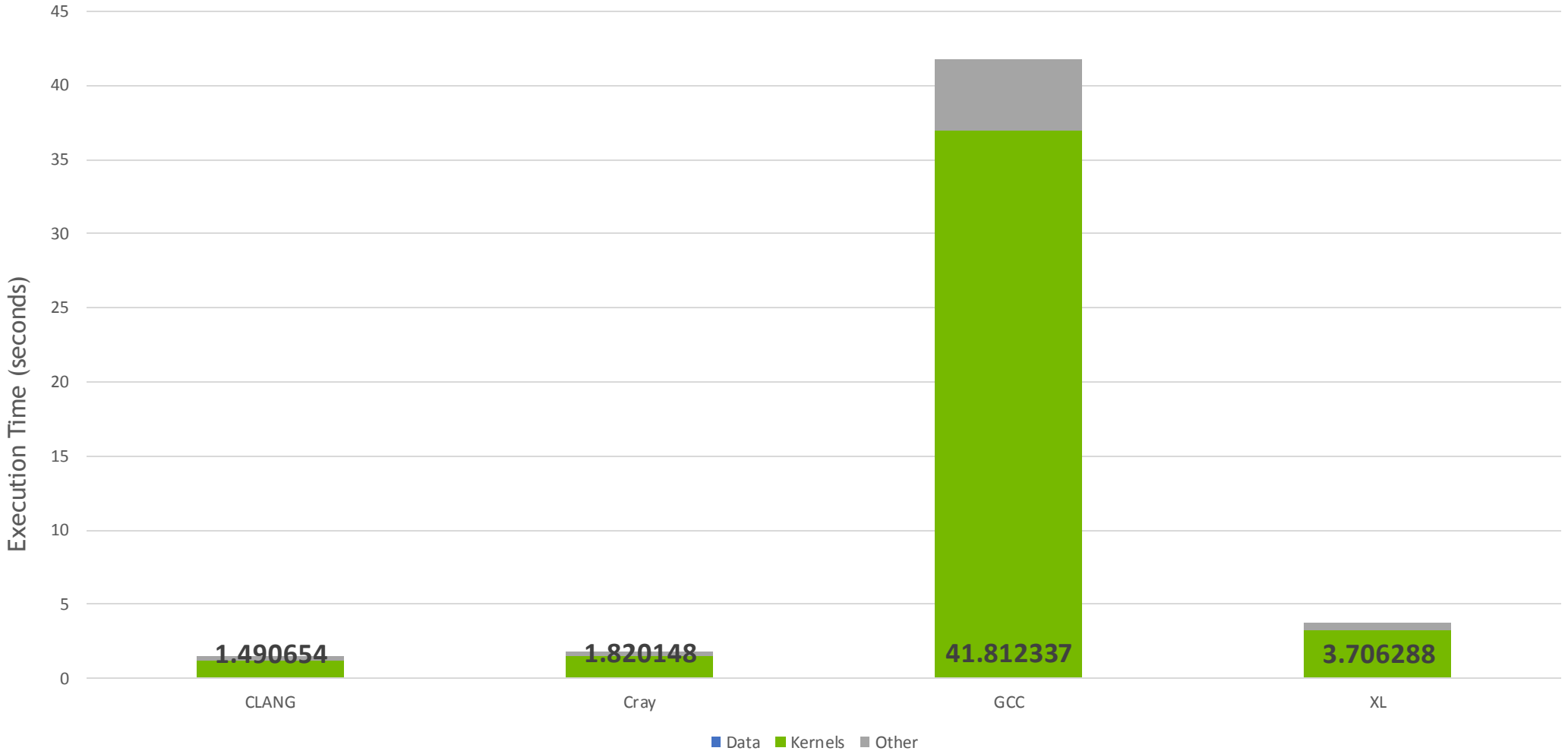
- Turns the next N loops into one, linearized loop.
- This will give us more parallelism to distribute, if we so choose.

Collapse

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

#pragma omp target teams distribute parallel for collapse(2)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

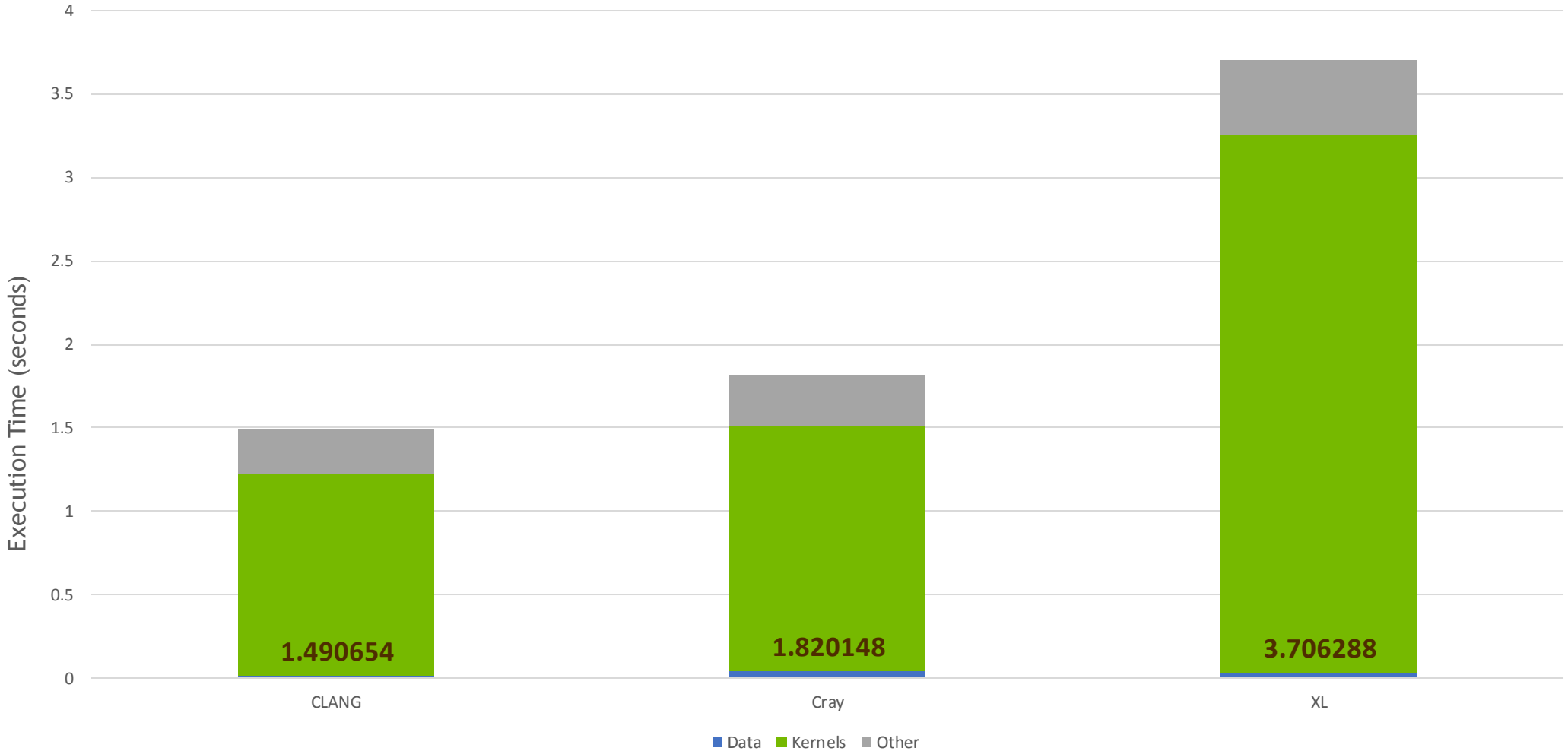
Collapse the two loops into one and then parallelize this new loop across both teams and threads.

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

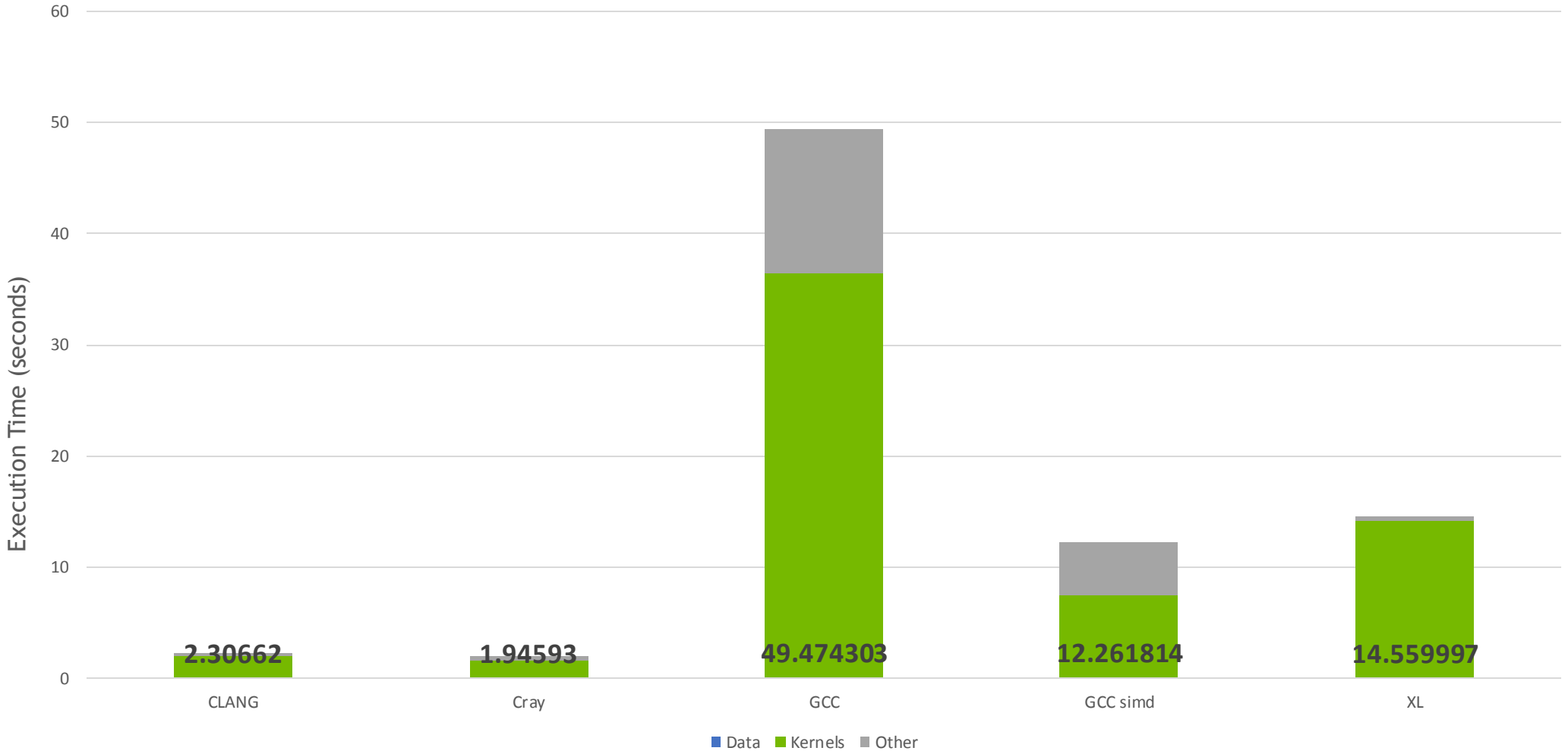
Splitting Teams & Parallel

```
#pragma omp target teams distribute map(error)
        for( int j = 1; j < n-1; j++)
        {
#pragma omp 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]));
            }
        }

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

Distribute the “j” loop over teams.

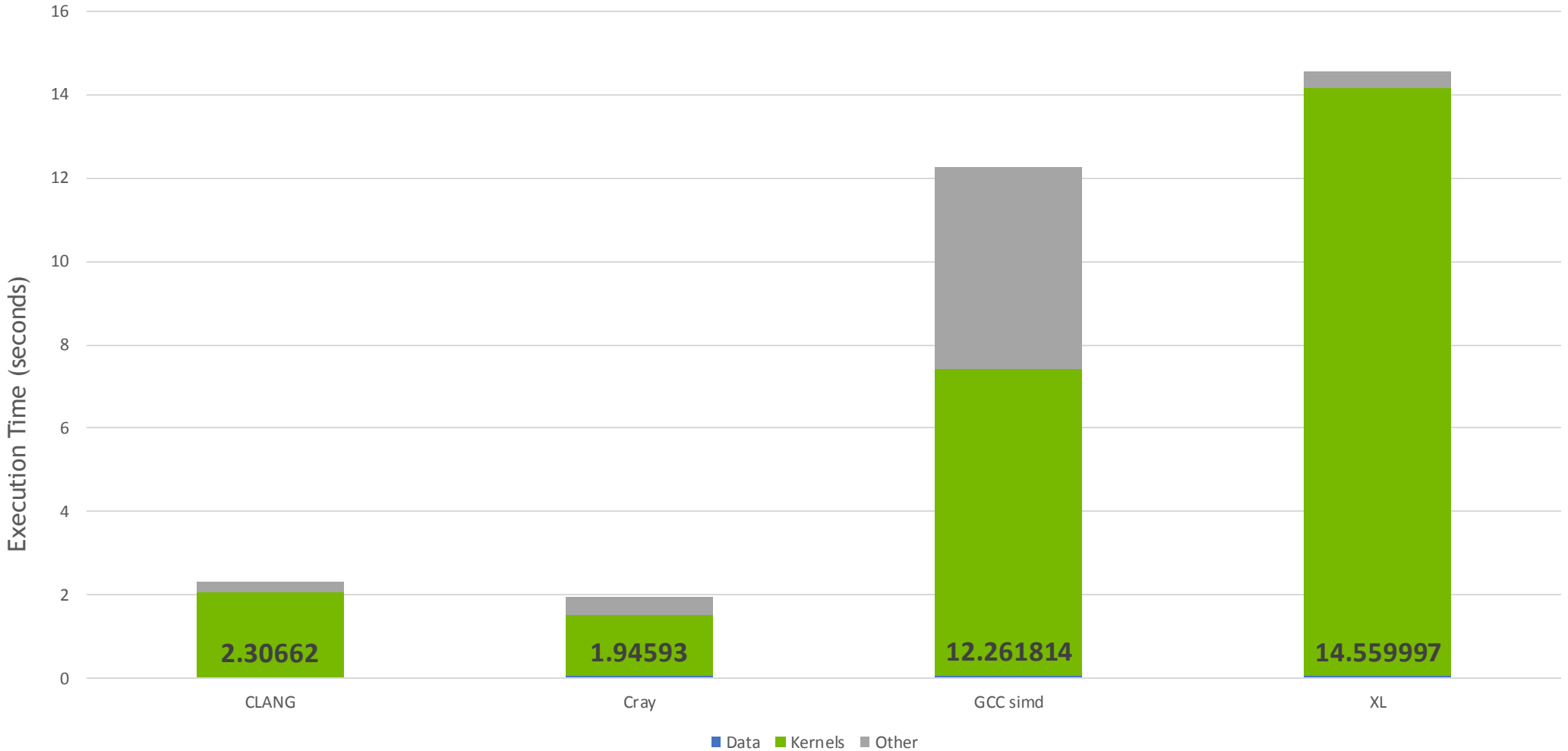
Workshare the “i” loop over threads.

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

Host Fallback

Fallback to the Host Processor

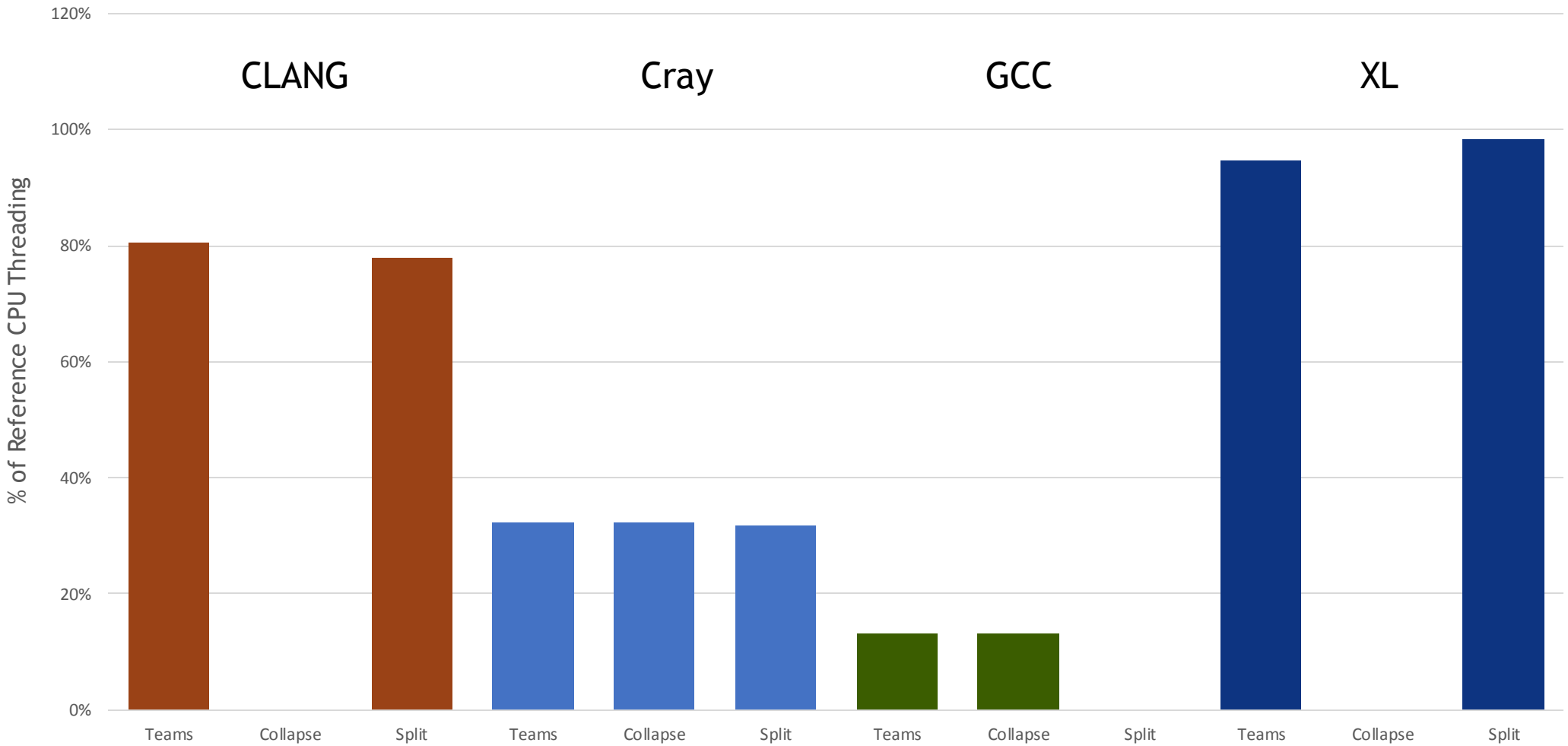
Most OpenMP users would like to write 1 set of directives for host and device, but is this really possible?

Using the “if” clause, offloading can be enabled/disabled at runtime.

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

Compiler must build CPU & GPU codes and select at runtime.

Host Fallback vs. Host Native OpenMP



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100

Conclusions

Conclusions

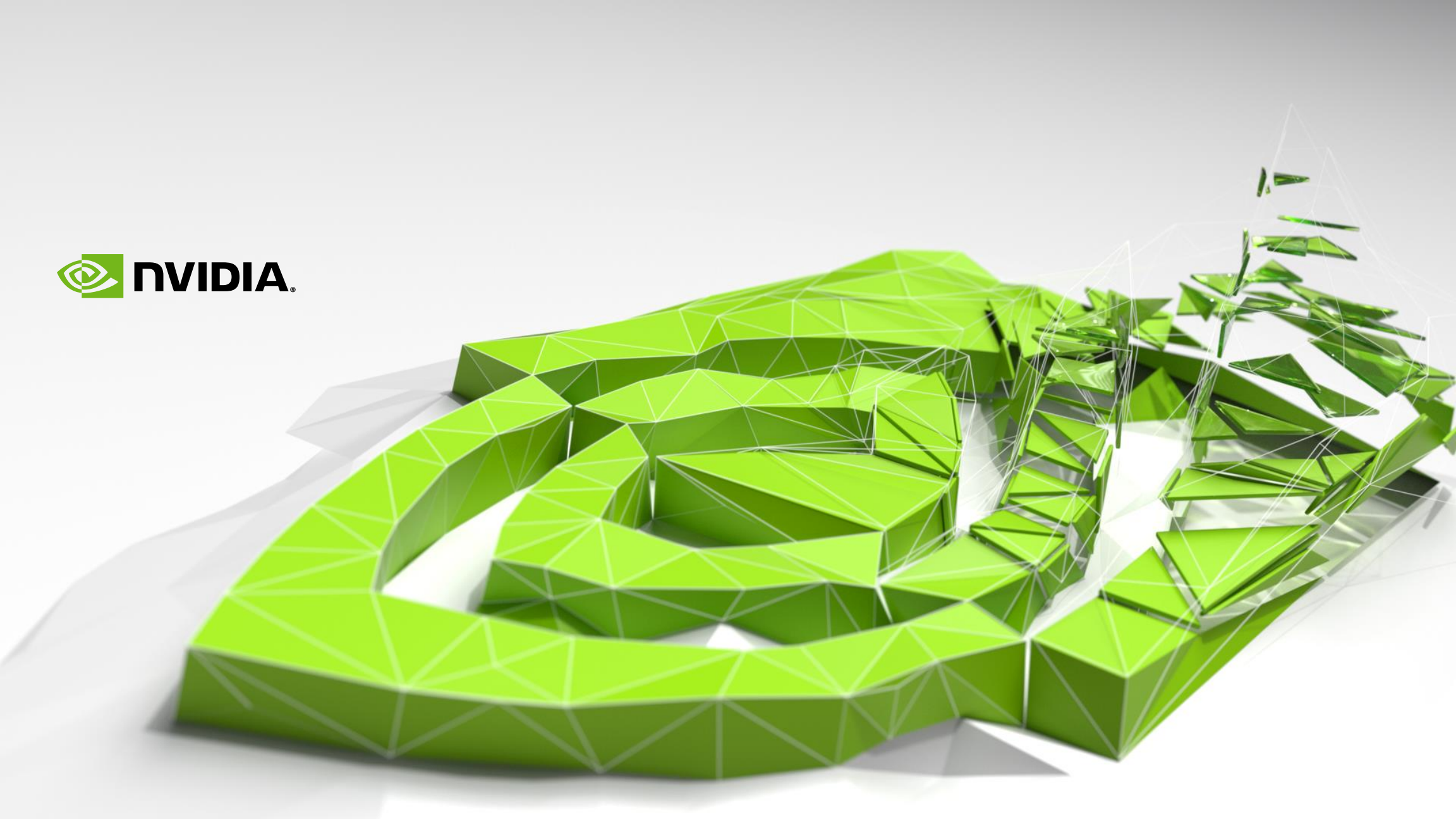
OpenMP offloading compilers for NVIDIA GPUs have improved *dramatically* over the past year and are ready for real use.

- Will OpenMP Target code be portable between compilers?

Maybe. Compilers are of various levels of maturity. SIMD support/requirement inconsistent.

- Will OpenMP Target code be portable with the host?

Highly compiler-dependent. XL does this very well, CLANG somewhat well, and GCC and Cray did poorly.


NVIDIA