Portability of OpenMP Offload Directives

Jeff Larkin, OpenMP Booth Talk SC17

Background

Many developers choose OpenMP in hopes of having a single source code that runs effectively anywhere (performance portable)

As of November 2017, OpenMP compilers 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 6 compilers: CLANG, Cray, GCC, Intel, PGI, and XL

Goal of this study

1. For the GPU-enabled compilers, compare performance on a simple benchmark code.

Metric = Execution time on GPU

2. Quantify each compiler's ability to performantly fallback to the host CPU

In other words, if I write offloading code, will it still perform well on the host?

Metric = Time Native OpenMP / Time Host Fallback

Compiler Versions & Flags

CLANG (IBM Power8 + NVIDIA Tesla P100)

- clang/20170629
- -O2 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda --cuda-path=$CUDA_HOME

Cray (Cray XC50)

- 8.5.5

GCC (IBM Power8 + NVIDIA Tesla P100)

- 7.1.1 20170718 (experimental)
- -O3 -fopenmp -foffload="-lm"

Compiler Versions & Flags

Intel (Intel “Haswell”)

- 17.0
- -Ofast -qopenmp -std=c99 -qopenmp-offload=host

XL (IBM Power8 + NVIDIA Tesla P100)

- xl/20170727-beta
- -O3 -qsmp -qoffload

PGI (Intel CPU)

- 17.10 (llvm)
- -fast -mp -Minfo -Mllvm

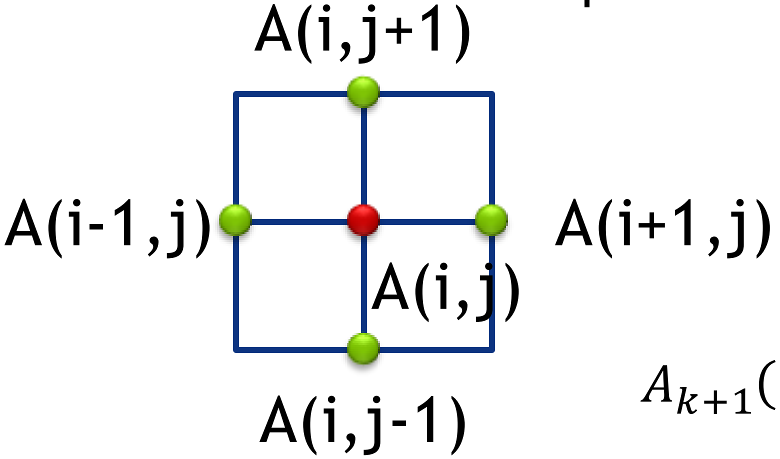
Case Study: Jacobi Iteration

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: $\boldsymbol{\nabla}^2 \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \mathbf{0}$



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

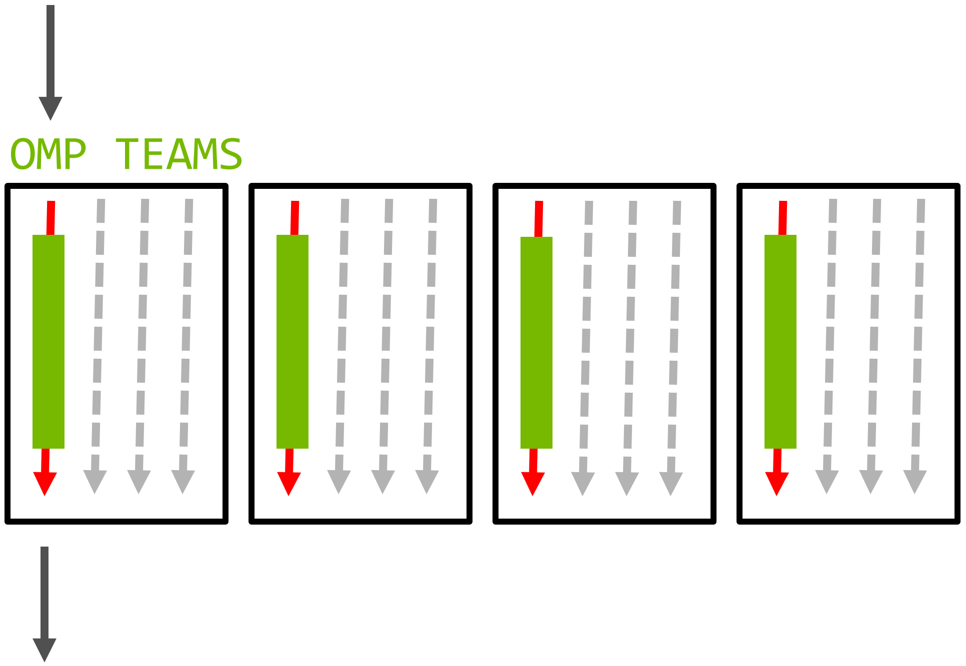
Teams & Distribute

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

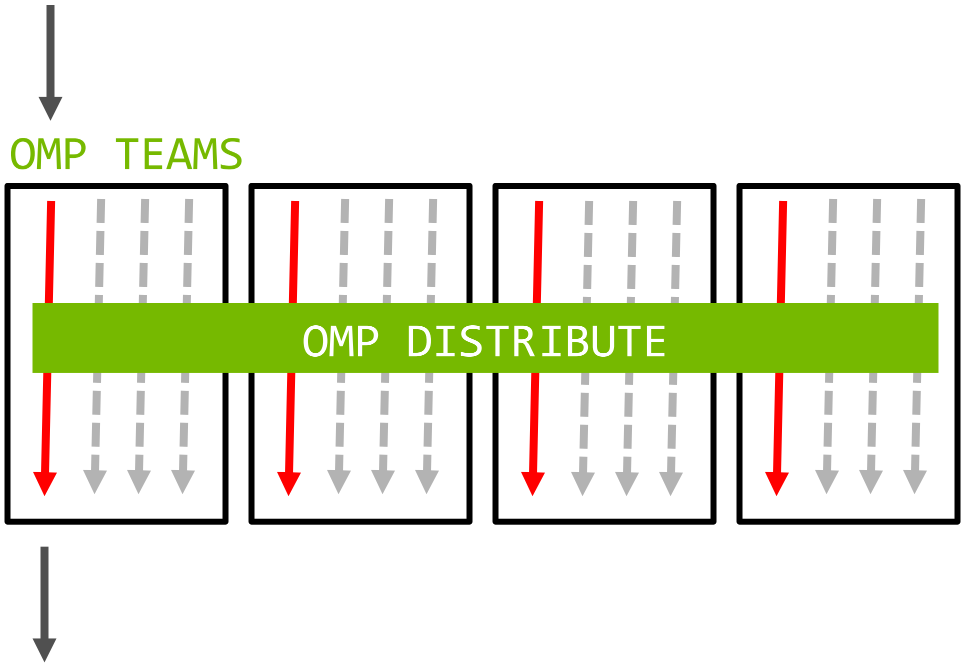


OpenMP Teams

- 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.



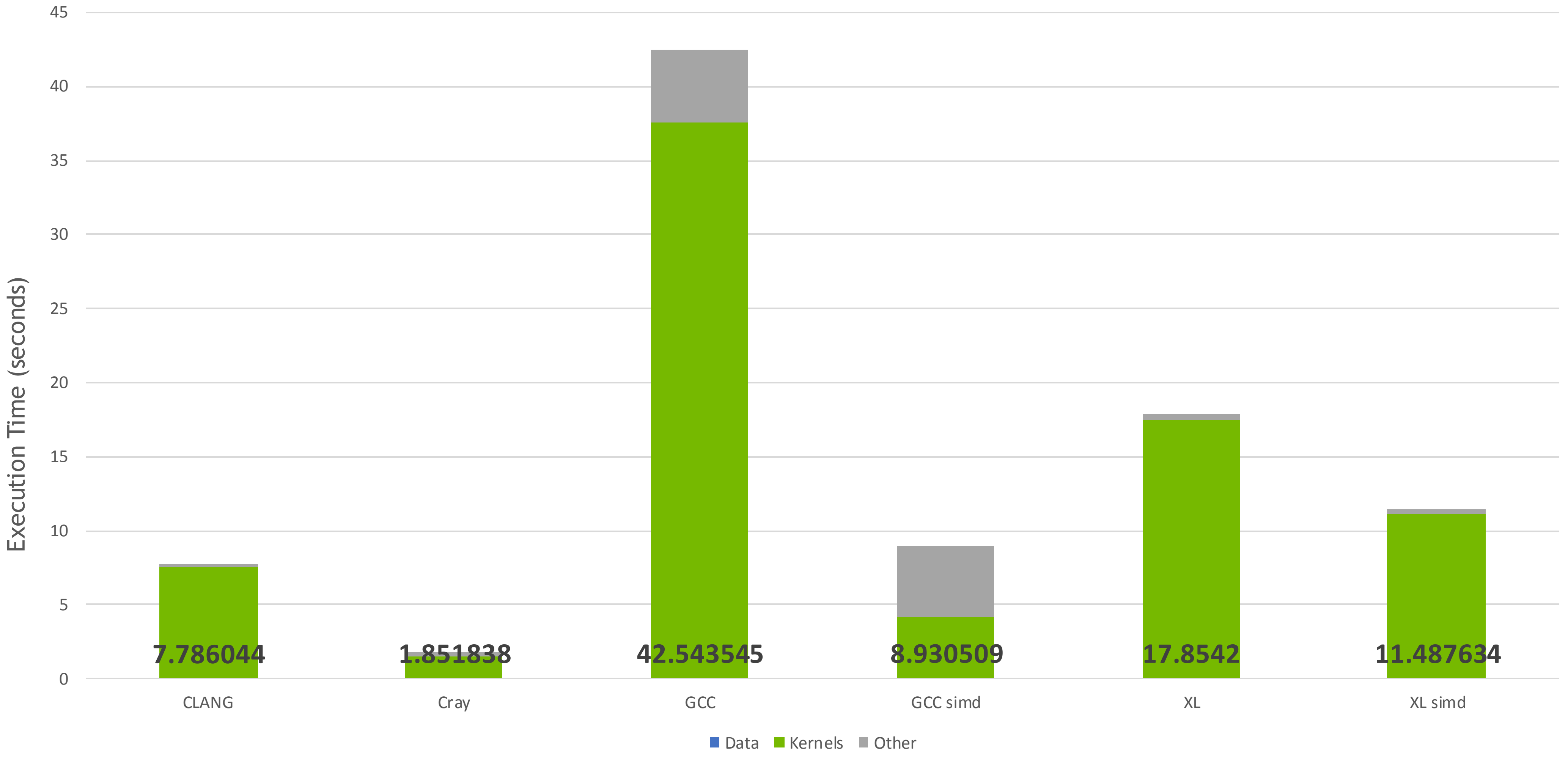
Teaming Up

```
#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.

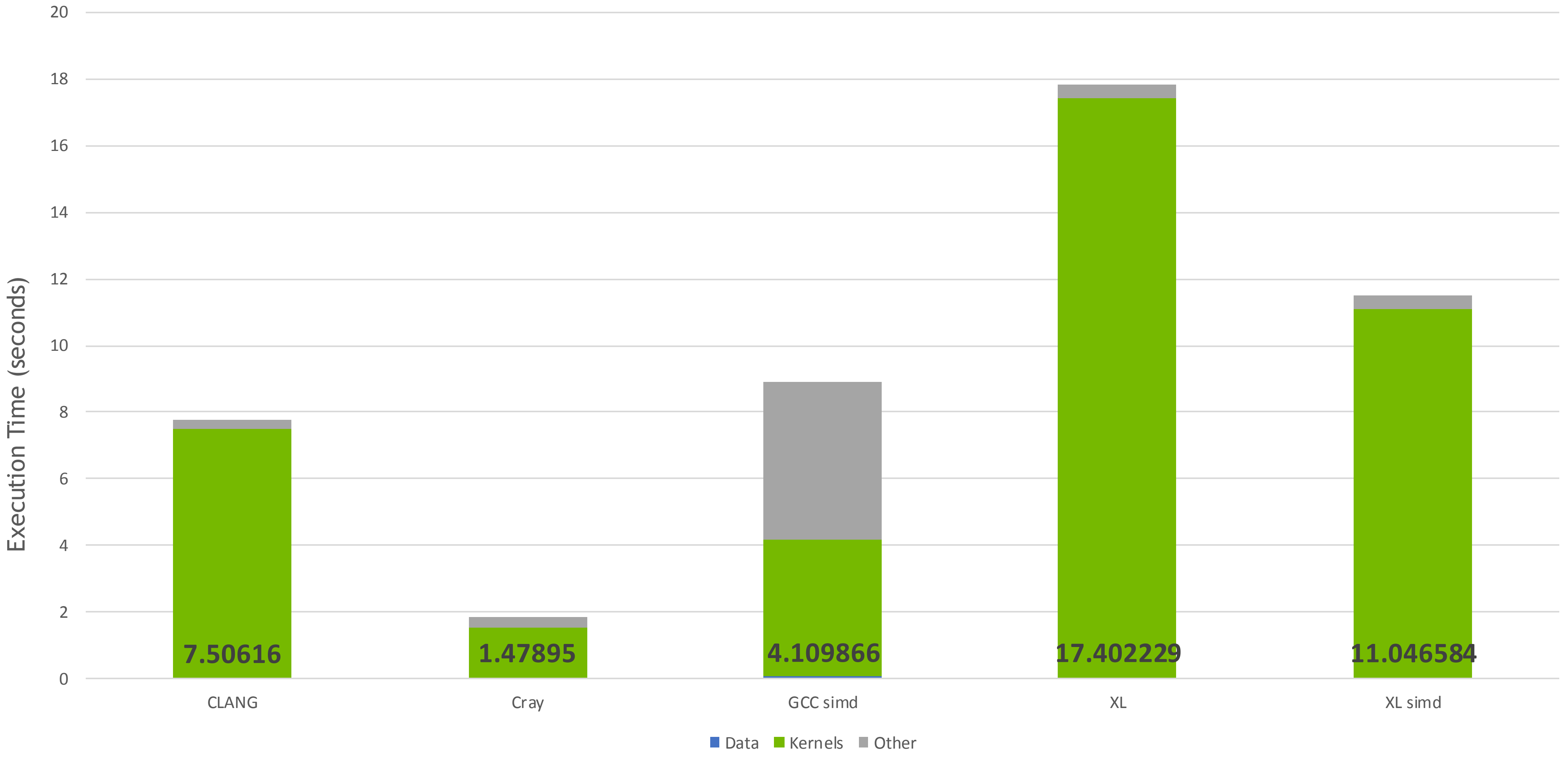
- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, 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 FOR to the inner loop

The COLLAPSE(N) clause

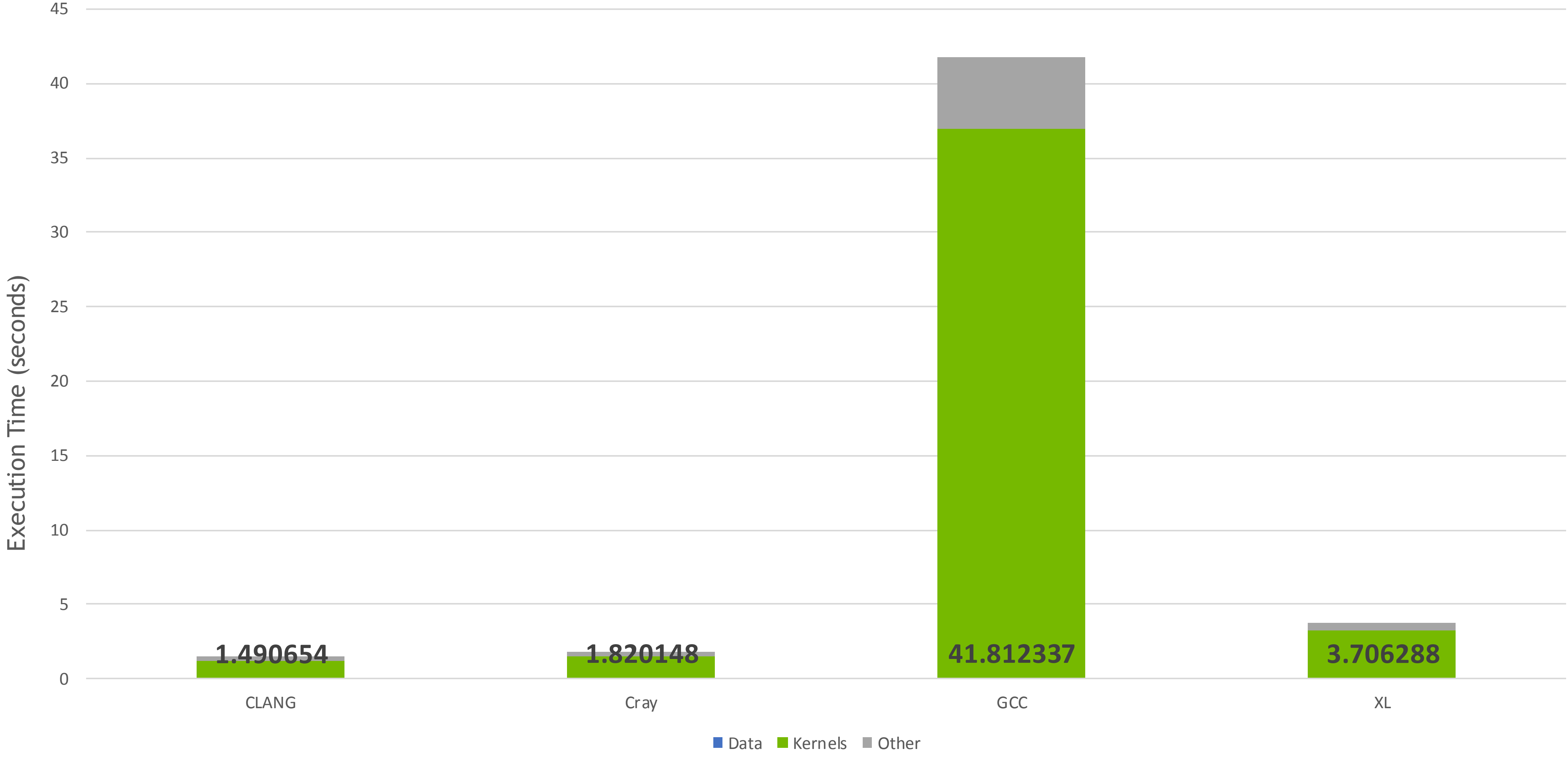
- Turns the next N loops into one, linearized loop.
- This will give us more parallelism to distribute, if we so choose.

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];
            }
        }
```

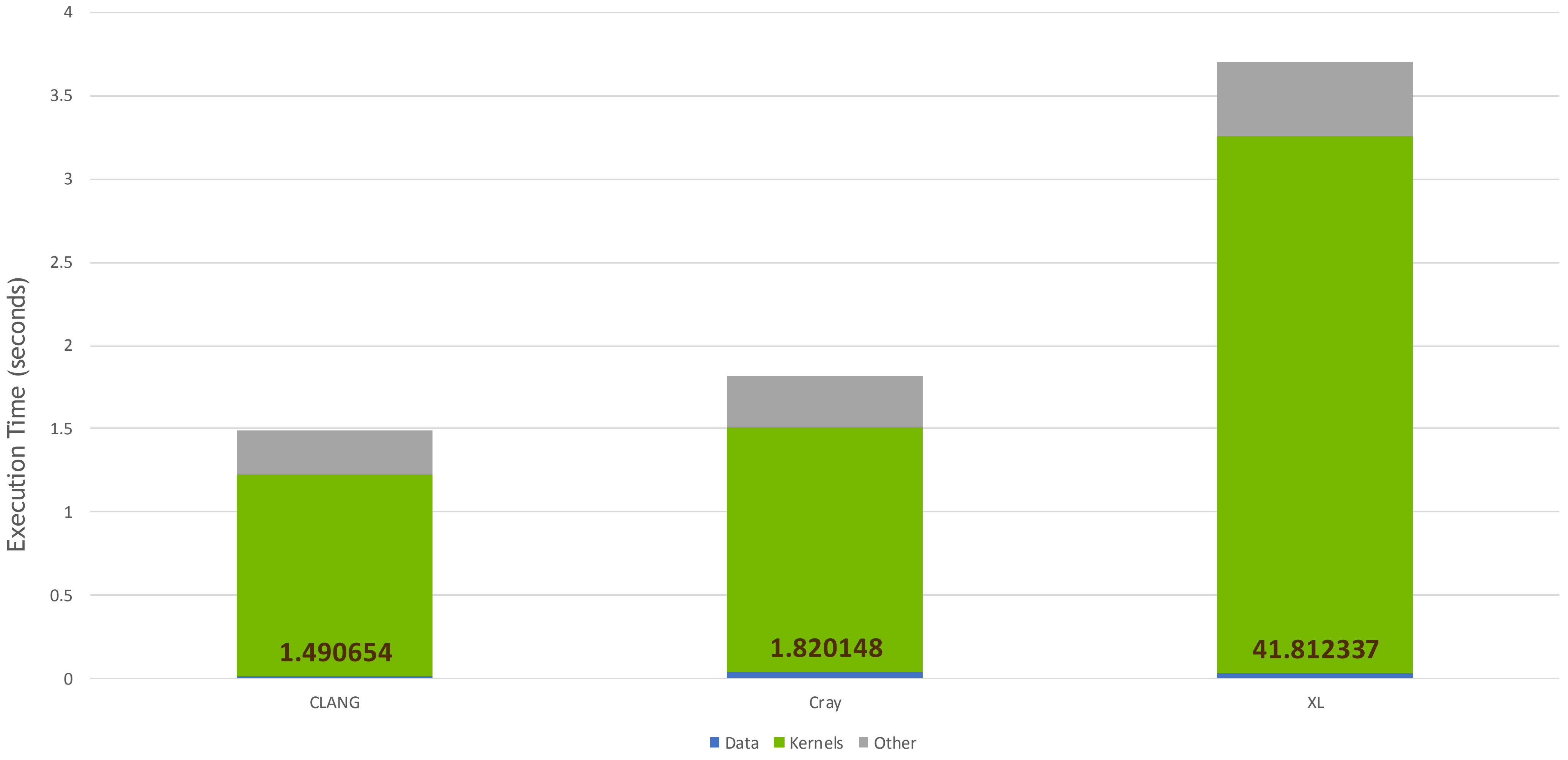
Collapse the two loops into one and then parallelize this new loop across both teams and threads.

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100

Execution Time (Smaller is Better)



CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100

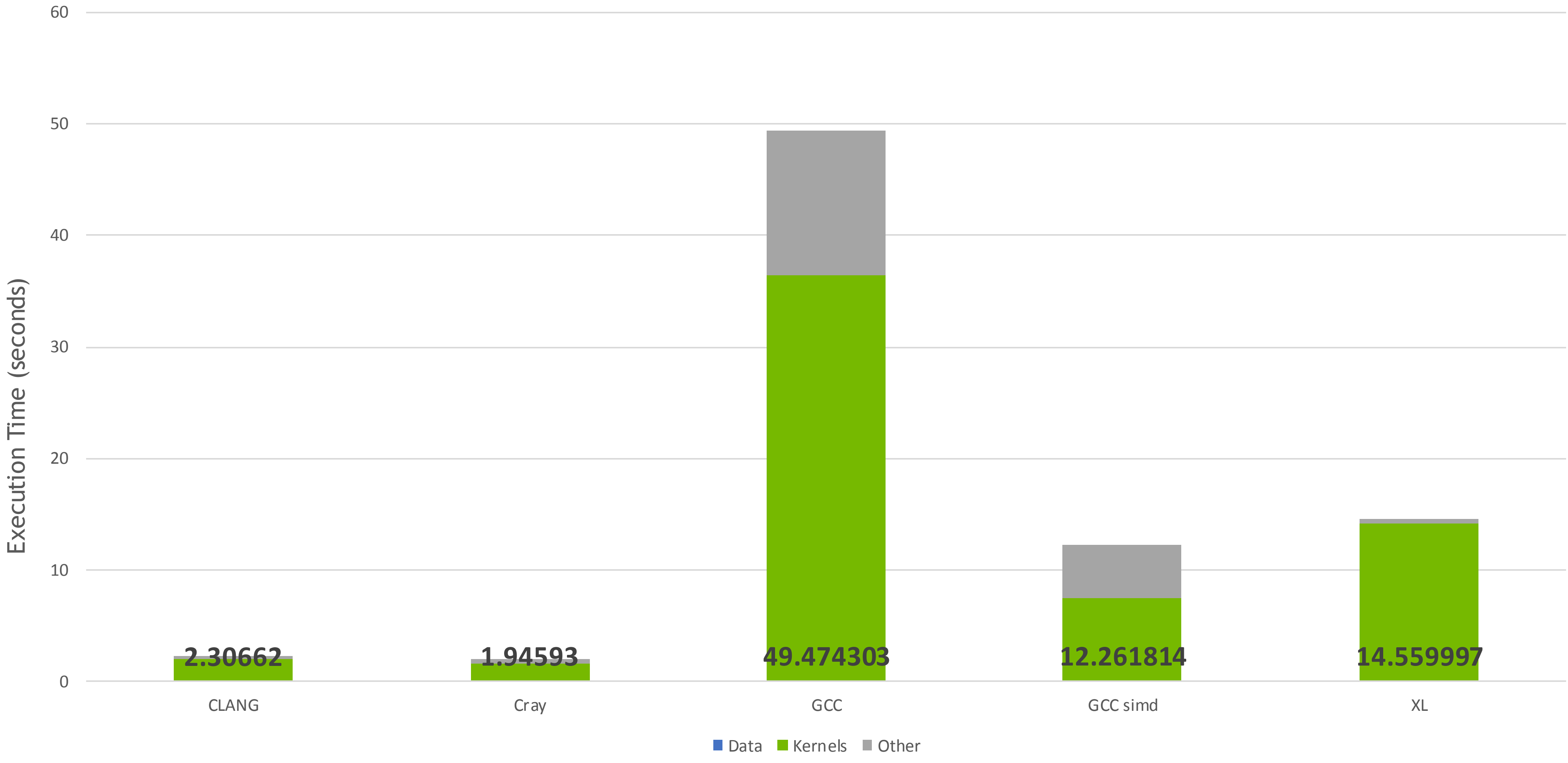
Splitting Teams & Parallel

```
#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.

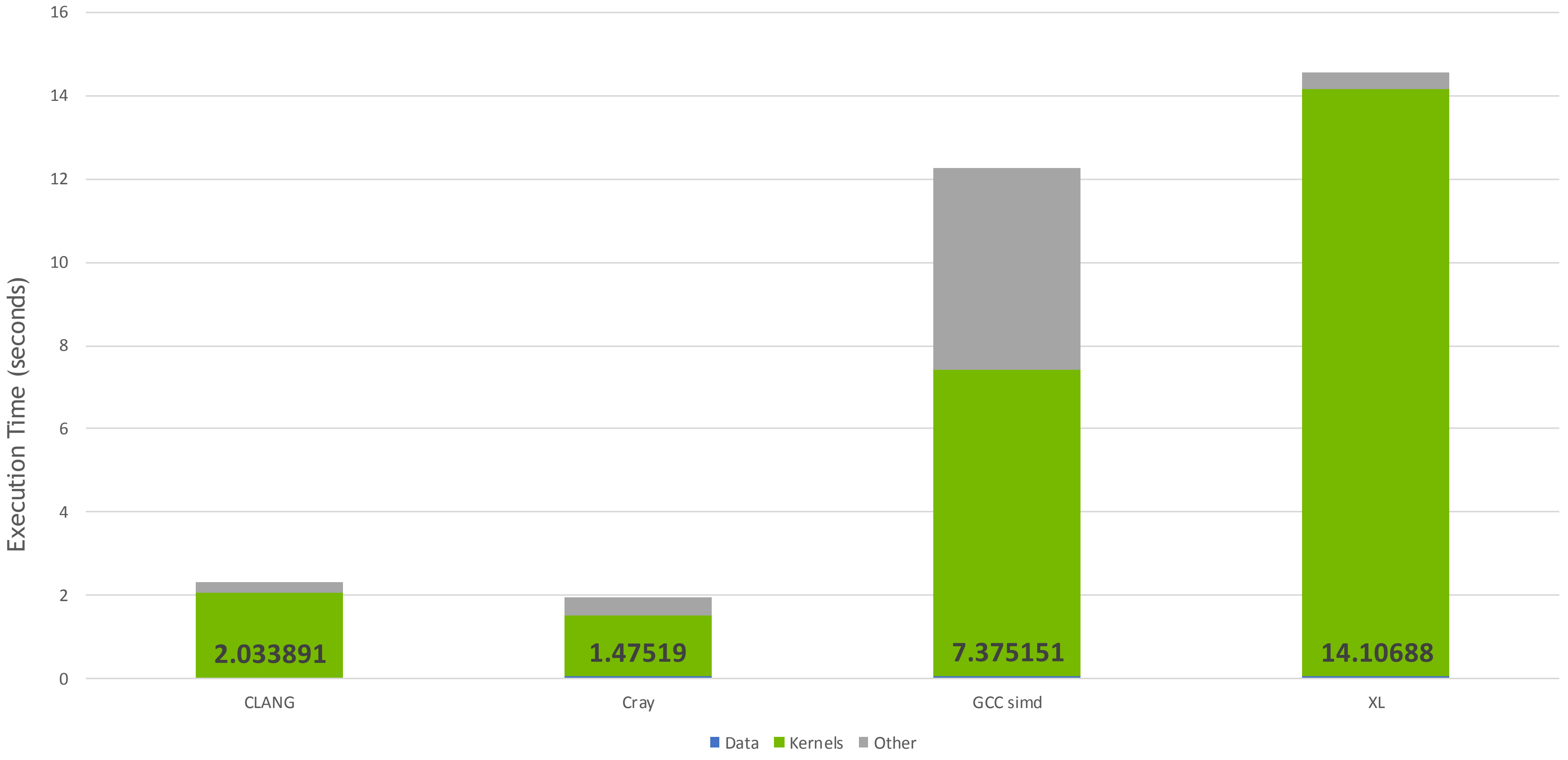
← Workshare the “i” loop over threads.

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-50, 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 Comparison

"Native" OpenMP = Standard OMP PARALLEL FOR (SIMD)

"Host Fallback" = Device OpenMP, forced to run on host

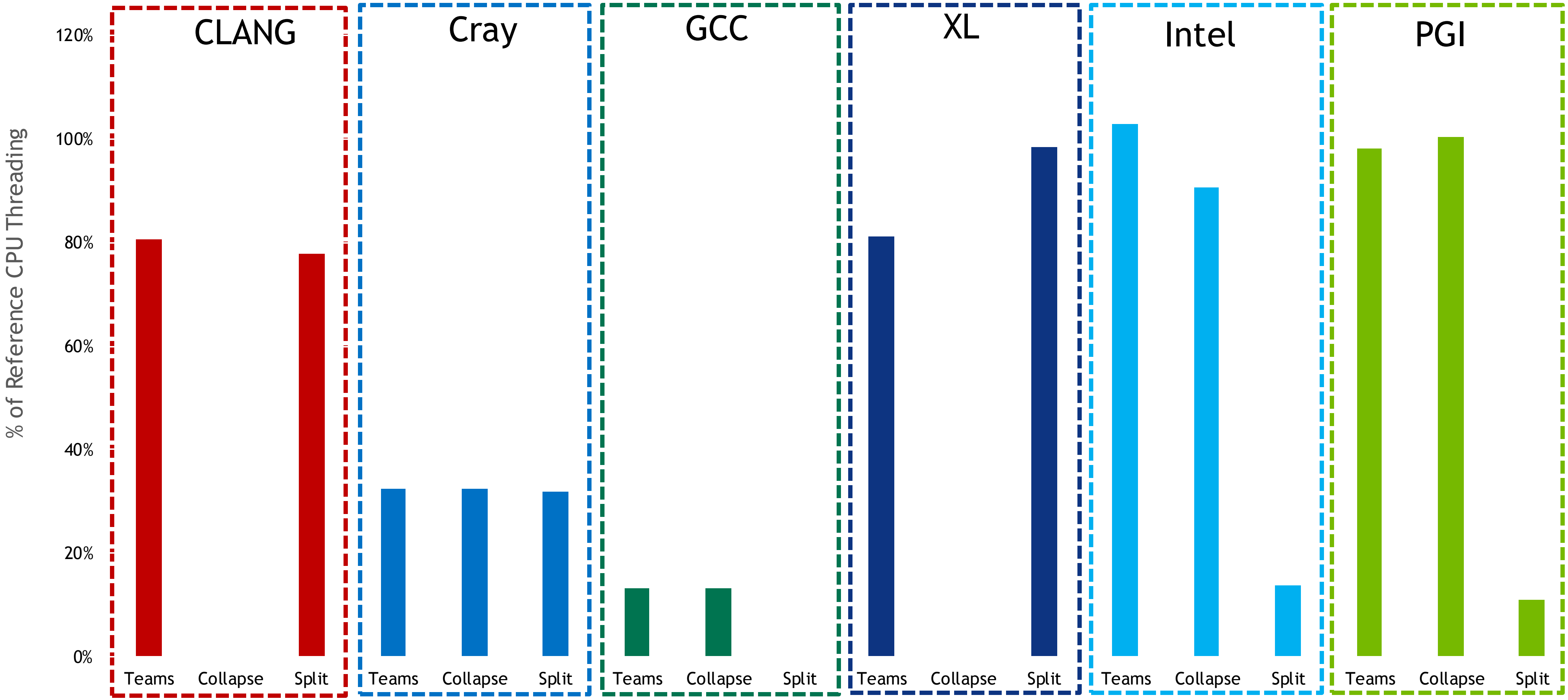
Metric: Native Time / Host Fallback Time

In other words...

100% means the perform equally well

50% means the host fallback takes 2X longer

Host Fallback vs. Host Native OpenMP



CLANG, GCC, XL: IBM “Minsky”, NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100, Intel, PGI: Intel “Haswell”

Conclusions

Conclusions

- 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. Intel, PGI, and XL do this very well, CLANG somewhat well, and GCC and Cray did poorly.

Future Work: Revisit these experiments as compilers are updated.

NVIDIA