DIRECTIVES-BASED PARALLEL PROGRAMMING

Jeff Larkin <jlarkin@nvidia.com>, 2/12/2020
1. You should have received an email from Okta, please activate your account.


3. Open terminal and run: sft enroll --team nv-demo

4. Approve in browser

5. Back in your terminal, log in to the hackathon head node: sft ssh -L 9090:localhost:9090 raplab-hackathon

6. Open http://localhost:9090
WHAT’S A COMPILER DIRECTIVE?

Compiler Directives are instructions, hints, or other information given to the compiler beyond the base language source code.

Examples:

• GCC, unroll this loop 4 times (#pragma GCC unroll 4)

• Ignore vector dependencies in this loop (#pragma ivdep)

These are different from pre-processor macros, since they affect the actual code generation during compilation, can’t always be pre-processed.

These are different from attributes because they are not defined by the language.
DIRECTIVES FOR PARALLEL PROGRAMMING

OpenMP - Established 1997

OpenMP was founded to create a unified set of compiler directives for shared-memory-parallel computers (SMP) that were becoming commonplace in the late 90s. In 2008 this was expanded to include task-based parallelism. In 2013 this was expanded again to include offloading to co-processors.

OpenACC - Established 2011

OpenACC was founded to create a unified set of compiler directives for “accelerators” that began emerging around 2010 (primarily GPUs) with potentially disjoint memories, non-smp architectures, and offloading.

Both directives support C, C++, and Fortran and require support from the compiler.
A BRIEF HISTORY OF OPENMP

- **OpenMP 1.0** (1996): Basic shared-memory parallelism, C/C++ and Fortran two specs.
- **OpenMP 2.0** (2000, 2002): Clarifications and Improvements
- **OpenMP 2.5** (2005): Unified into single spec.
- **OpenMP 3.0** (2008): Tasking Added
- **OpenMP 3.1** (2011): Tasking Improvements
- **OpenMP 4.0** (2013): Target Offloading, Teams, SIMD, Atomics, Tasking Improvements
- **OpenMP 4.5** (2015): Improved Offloading, Task Priority
- **OpenMP 5.0** (2018): Loop Construct, Metadirective, Base language updates, Requires directive
A BRIEF HISTORY OF OPENACC

**Incorporation**
ORNL asks CAPS, Cray, & PGI to unify efforts with the help of NVIDIA

2011

**OpenACC 1.0**
Basic parallelism, structured data, and async/wait semantics

Oct. 2011

**OpenACC 2.0**
Unstructured Data Lifetimes, Routines, Atomic, Clarifications & Improvements

June 2013

**OpenACC 2.5**
Reference Counting, Profiling Interface, Additional Improvements from User Feedback

Oct. 2015

**OpenACC 2.6**
Serial Construct, Attach/Detach (Manual Deep Copy), Misc. User Feedback

Nov. 2016

**OpenACC 2.7**
Compute on Self, readonly, Array Reductions, Lots of Clarifications, Misc. User Feedback

Nov. 2018

**OpenACC 2.8**
Serial Construct, Attach/Detach (Manual Deep Copy), Misc. User Feedback

Nov. 2019

**OpenACC 3.0**
Updated Base Languages, C++, Lambdas, Zero modifier, Improved multi-device support

Nov. 2019
3 WAYS TO ACCELERATE APPLICATIONS

- Libraries: Easy to use, Most Performance
- Compiler Directives: Easy to use, Portable code
- Programming Languages: Most Performance, Most Flexibility
THE FUTURE OF PARALLEL PROGRAMMING
Standard Languages | Directives | Specialized Languages

Drive Base Languages to Better Support Parallelism
Augment Base Languages with Directives
Maximize Performance with Specialized Languages & Intrinsics

```
std::for_each_n(POL, idx(0), n, [&](Index_t i)
    y[i] += a*x[i];
});

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

#pragma acc data copy(x,y) {
...
std::for_each_n(POL, idx(0), n, [&](Index_t i)
    y[i] += a*x[i];
});
...
}

__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, ...);
```
DIRECTIVE SYNTAX
Syntax for using compiler directives in code

A *pragma* in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A *directive* in Fortran is a specially formatted comment that likewise instructs the compiler in the compilation of the code and can be freely ignored.

The *sentinel* informs the compiler the directive language that will follow (acc = OpenACC, omp = OpenMP)

*Directives* are commands and information to the compiler for altering or interpreting the code

*Clauses* are specifiers or additions to directives, like function parameters.
INTRODUCTION TO OPENACC
OpenACC Directives

- Manage Data Movement
- Initiate Parallel Execution
- Optimize Loop Mappings

```c
#pragma acc data copyin(a,b) copyout(c) {
    ...
#pragma acc parallel
    {
#pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            c[i] = a[i] + b[i];
            ...
        }
    }
    ...
}
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore
### OPENACC

<table>
<thead>
<tr>
<th>Incremental</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintain existing sequential code</td>
</tr>
<tr>
<td>Add annotations to expose parallelism</td>
</tr>
<tr>
<td>After verifying correctness, annotate more of the code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rebuild the same code on multiple architectures</td>
</tr>
<tr>
<td>Compiler determines how to parallelize for the desired machine</td>
</tr>
<tr>
<td>Sequential code is maintained</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Learning Curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC is meant to be easy to use, and easy to learn</td>
</tr>
<tr>
<td>Programmer remains in familiar C, C++, or Fortran</td>
</tr>
<tr>
<td>No reason to learn low-level details of the hardware.</td>
</tr>
</tbody>
</table>
OPENACC DIRECTIVES

a directive-based parallel programming model designed for usability, performance and portability

3 OF TOP 5 HPC

18% OF INCITE AT SUMMIT

PLATFORMS SUPPORTED

NVIDIA GPU
X86 CPU
POWER CPU
Sunway
ARM CPU
AMD GPU

OPENACC APPS

OPENACC SLACK MEMBERS

>200K DOWNLOADS
EXAMPLE CODE
LAPLACE HEAT TRANSFER
Introduction to lab code - visual

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.
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: \( \nabla^2 f(x, y) = 0 \)

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
while ( err > tol && iter < iter_max ) {
    err=0.0;
    
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
    
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    
    iter++;
}
PROFILE-DRIVEN DEVELOPMENT
OPENACC DEVELOPMENT CYCLE

- **Analyze** your code to determine most likely places needing parallelization or optimization.

- **Parallelize** your code by starting with the most time consuming parts and check for correctness.

- **Optimize** your code to improve observed speed-up from parallelization.
Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing.

Lab Code: Laplace Heat Transfer
Total Runtime: 39.43 seconds

- swap 19.04s
- calcNext 21.49s
OPENACC PARALLEL LOOP DIRECTIVE
OPENACC PARALLEL DIRECTIVE
Expressing parallelism

```c
#pragma acc parallel
{
    When encountering the `parallel` directive, the compiler will generate 1 or more parallel `gangs`, which execute redundantly.
}
```
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
  for(int i = 0; i < N; i++)
  {
    // Do Something
  }
}
```

This loop will be executed redundantly on each gang.
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

#pragma acc parallel
{
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }
}

This means that each gang will execute the entire loop.
OPENACC PARALLEL DIRECTIVE
Parallelizing a single loop

**C/C++**

```c
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; j < N; i++)
        a[i] = 0;
}
```

**Fortran**

```fortran
!$acc parallel
  !$acc loop
  do i = 1, N
      a(i) = 0
  end do
!$acc end parallel
```

- Use a `parallel` directive to mark a region of code where you want parallel execution to occur.
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran.
- The `loop` directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs.
OPENACC PARALLEL DIRECTIVE
Parallelizing a single loop

- This pattern is so common that you can do all of this in a single line of code
- In this example, the parallel loop directive applies to the next loop
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency, parallel loop may produce incorrect results.

**C/C++**

```c
#pragma acc parallel loop
for(int i = 0; j < N; i++)
  a[i] = 0;
```

**Fortran**

```fortran
!$acc parallel loop
do i = 1, N
  a(i) = 0
end do
```
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }
}
```

The `loop` directive informs the compiler which loops to parallelize.
OPENACC PARALLEL LOOP DIRECTIVE
Parallelizing many loops

- To parallelize multiple loops, each loop should be accompanied by a parallel directive
- Each parallel loop can have different loop boundaries and loop optimizations
- Each parallel loop can be parallelized in a different way
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results

```c
#pragma acc parallel loop
for(int i = 0; i < N; i++)
    a[i] = 0;
```

```c
#pragma acc parallel loop
for(int j = 0; j < M; j++)
    b[j] = 0;
```
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++) {


            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++;
}
**REDUCTION CLAUSE**

- The **reduction** clause takes many values and “reduces” them to a single value, such as in a sum or maximum.
- Each thread calculates its part.
- The compiler will perform a final reduction to produce a **single global result** using the specified operation.

```c
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

```c
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma parallel acc loop \
    reduction(+:tmp)\n    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```
### REDUCTION CLAUSE OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition/Summation</td>
<td>reduction(+:sum)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication/Product</td>
<td>reduction(*:product)</td>
</tr>
<tr>
<td>max</td>
<td>Maximum value</td>
<td>reduction(max:maximum)</td>
</tr>
<tr>
<td>min</td>
<td>Minimum value</td>
<td>reduction(min:minimum)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
<td>reduction(&amp;:val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
<td>reduction(&amp;&amp;:val)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BUILD AND RUN THE CODE
PGI COMPILER BASICS

pgcc, pgc++ and pgfortran

- The command to compile C code is ‘pgcc’
- The command to compile C++ code is ‘pgc++’
- The command to compile Fortran code is ‘pgfortran’
- The -fast flag instructs the compiler to optimize the code to the best of its abilities

```bash
$ pgcc -fast main.c
$ pgc++ -fast main.cpp
$ pgfortran -fast main.F90
```
PGI COMPILER BASICS

-Minfo flag

- The -Minfo flag will instruct the compiler to print feedback about the compiled code
- -Minfo=accel will give us information about what parts of the code were accelerated via OpenACC
- -Minfo=opt will give information about all code optimizations
- -Minfo=all will give all code feedback, whether positive or negative

```
$ pgcc -fast -Minfo=all main.c
$ pgc++ -fast -Minfo=all main.cpp
$ pgfortran -fast -Minfo=all main.f90
```
PGI COMPILER BASICS

- `ta` flag

- The `ta` flag enables building OpenACC code for a “Target Accelerator” (TA)

- `ta=multicore` – Build the code to run across threads on a multicore CPU

- `ta=tesla:managed` – Build the code for an NVIDIA (Tesla) GPU and manage the data movement for me (more next week)

$ pgcc -fast -Minfo=accel -ta=tesla:managed main.c
$ pgc++ -fast -Minfo=accel -ta=tesla:managed main.cpp
$ pgfortran -fast -Minfo=accel -ta=tesla:managed main.f90
BUILDING THE CODE (MULTICORE)

$ pgcc -fast -ta=multicore -Minfo=accel laplace2d_uvm.c
main:

63, Generating Multicore code
64, #pragma acc loop gang
64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
Generating reduction(max:error)
66, Loop is parallelizable
74, Generating Multicore code
75, #pragma acc loop gang
75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
77, Loop is parallelizable
OPENACC SPEED-UP

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
$ pgcc -fast -ta=tesla:managed -Minfo=accel laplace2d_uvm.c

main:

63, Accelerator kernel generated
Generating Tesla code
64, #pragma acc loop gang /* blockIdx.x */
    Generating reduction(max:error)
66, #pragma acc loop vector(128) /* threadIdx.x */

63, Generating implicit copyin(A[:])
Generating implicit copyout(Anew[:])
Generating implicit copy(error)

66, Loop is parallelizable

74, Accelerator kernel generated
Generating Tesla code
75, #pragma acc loop gang /* blockIdx.x */
77, #pragma acc loop vector(128) /* threadIdx.x */

74, Generating implicit copyin(Anew[:])
Generating implicit copyout(A[:])

77, Loop is parallelizable
OPENACC SPEED-UP

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
CPU AND GPU MEMORIES
CPU + GPU
Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCI-e)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory
CUDA UNIFIED MEMORY
CUDA UNIFIED MEMORY
Simplified Developer Effort

Without Managed Memory

With Managed Memory

Commonly referred to as “managed memory.”

CPU and GPU memories are combined into a single, shared pool.
CUDA MANAGED MEMORY

Usefulness

- Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
- The PGI compiler can utilize CUDA Managed Memory to defer data management
- This allows the developer to concentrate on parallelism and think about data movement as an optimization

```bash
$ pgcc -fast -acc -ta=tesla:managed -Minfo=accel main.c

$ pgfortran -fast -acc -ta=tesla:managed -Minfo=accel main.f90
```
MANAGED MEMORY

Limitations

- The programmer will almost always be able to get better performance by manually handling data transfers.
- Memory allocation/deallocation takes longer with managed memory.
- Cannot transfer data asynchronously.
- Currently only available from PGI on NVIDIA GPUs.
SPEC ACCEL 1.2 OPENACC BENCHMARKS
OpenACC with Unified Memory vs OpenACC Data Directives

100% = Pure Directive-based Data Movement

PGI 18.4 Compilers OpenACC SPEC ACCEL™ 1.2 performance measured June, 2018
SPEC® and the benchmark name SPEC ACCEL™ are registered trademarks of the Standard Performance Evaluation Corporation.

* Slide Courtesy of PGI
WE’RE USING UNIFIED MEMORY
Now let’s make our code run without.

Why?

- Removes reliance on PGI and NVIDIA GPUs
- Currently the data always arrives “Just Too Late”, let’s do better
BASIC DATA MANAGEMENT
BASIC DATA MANAGEMENT

Between the host and device

- The **host** is traditionally a CPU
- The **device** is some parallel accelerator
- When our target hardware is multicore, the host and device are the same, meaning that their memory is also the same
- There is no need to explicitly manage data when using a shared memory accelerator, such as the multicore target
BASIC DATA MANAGEMENT
Between the host and device

- When the target hardware is a GPU data will usually need to migrate between CPU and GPU memory
- Each array used on the GPU must be allocated on the GPU
- When data changes on the CPU or GPU the other must be updated
TRY TO BUILD WITHOUT “MANAGED”
Change –ta=tesla:managed to remove “managed”

pgcc -ta=tesla -Minfo=accel laplace2d.c jacobi.c

laplace2d.c:
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol (laplace2d.c: 47)
calcNext:
  47, Accelerator kernel generated
    Generating Tesla code
    48, #pragma acc loop gang /* blockIdx.x */
    Generating reduction(max:error)
  50, #pragma acc loop vector(128) /* threadIdx.x */
     48, Accelerator restriction: size of the GPU copy of Anew,A is unknown
     50, Loop is parallelizable

PGC-F-0704-Compilation aborted due to previous errors. (laplace2d.c)
PGC/x86-64 Linux 18.7-0: compilation aborted

jacobi.c:
DATA SHAPING
DATA CLAUSES

**copy(list)**
Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

**Principal use:** For many important data structures in your code, this is a logical default to input, modify and return the data.

**copyin(list)**
Allocates memory on GPU and copies data from host to GPU when entering region.

**Principal use:** Think of this like an array that you would use as just an input to a subroutine.

**copyout(list)**
Allocates memory on GPU and copies data to the host when exiting region.

**Principal use:** A result that isn’t overwriting the input data structure.

**create(list)**
Allocates memory on GPU but does not copy.

**Principal use:** Temporary arrays.
ARRAY SHAPING

- Sometimes the compiler needs help understanding the *shape* of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index

```c
copy(array[starting_index:length])  // C/C++
```

```fortran
copy(array(starting_index:ending_index))  // Fortran
```
Both of these examples copy a 2D array to the device.
ARRAY SHAPING (CONT.)

Partial Arrays

\[
\text{copy}(\text{array}[i\times N/4:N/4]) \quad \text{C/C++}
\]

Both of these examples copy only ¼ of the full array

\[
\text{copy}(\text{array}(i\times N/4:i\times N/4+N/4)) \quad \text{Fortran}
\]
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
    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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
TRY TO BUILD WITHOUT “MANAGED”
Change –ta=tesla:managed to remove “managed”

pgcc -ta=tesla -Minfo=accel laplace2d.c jacobi.c
laplace2d.c:
calcNext:
  47, Generating copyin(A[:m*n])
  Accelerator kernel generated
  Generating Tesla code
  48, #pragma acc loop gang /* blockIdx.x */
       Generating reduction(max:error)
  50, #pragma acc loop vector(128) /* threadIdx.x */
  47, Generating implicit copy(error)
  Generating copy(Anew[:m*n])
  50, Loop is parallelizable
swap:
  62, Generating copyin(Anew[:m*n])
  Generating copyout(A[:m*n])
  Accelerator kernel generated
  Generating Tesla code
  63, #pragma acc loop gang /* blockIdx.x */
  65, #pragma acc loop vector(128) /* threadIdx.x */
  65, Loop is parallelizable
jacobi.c:
OPENACC SPEED-UP SLOWDOWN

Speed-up

- SERIAL: 1.00X
- MULTICORE: 3.23X
- V100: 41.80X
- V100 (DATA CLAUSES): 0.33X
WHAT WENT WRONG?

- The code now has all of the information necessary to build without managed memory, but it runs much slower.
- Profiling tools are here to help!
APPLICATION PROFILE (2 STEPS)
APPLICATION PROFILE (2 STEPS)
RUNTIME BREAKDOWN

Nearly all of our time is spent moving data to/from the GPU.
OPTIMIZED DATA MOVEMENT

```c
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
    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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Currently we’re copying to/from the GPU for each loop, can we reuse it?
OPTIMIZE DATA MOVEMENT
OPENACC DATA DIRECTIVE

Definition

- The data directive defines a lifetime for data on the device beyond individual loops
- During the region data is essentially “owned by” the accelerator
- Data clauses express shape and data movement for the region

```c
#pragma acc data clauses
{
  < Sequential and/or Parallel code >
}

!$acc data clauses
  < Sequential and/or Parallel code >
!$acc end data
```
STRUCTURED DATA DIRECTIVE

Example

```c
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
    #pragma acc parallel loop
    for(int i = 0; i < N; i ++){
        c[i] = a[i] + b[i];
    }
}
```

<table>
<thead>
<tr>
<th>Action</th>
<th>Host Memory</th>
<th>Device memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate A on device</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Copy A from CPU to device</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>Allocate B on device</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Allocate C on device</td>
<td>C'</td>
<td>C'</td>
</tr>
<tr>
<td>Execute loop on device</td>
<td>C'</td>
<td>C'</td>
</tr>
<tr>
<td>Copy C from device to CPU</td>
<td>C'</td>
<td>C'</td>
</tr>
<tr>
<td>Deallocate C from device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deallocate B from device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deallocate A from device</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while (err > tol && iter < iter_max) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) copyin(A[0:n*m])
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
REBUILD THE CODE

pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
main:
  60, Generating copy(A[:m*n])
  Generating copyin(Anew[:m*n])
  64, Accelerator kernel generated
  Generating Tesla code
  64, Generating reduction(max:error)
  65, #pragma acc loop gang /* blockIdx.x */
  67, #pragma acc loop vector(128) /* threadIdx.x */
  67, Loop is parallelizable
  75, Accelerator kernel generated
  Generating Tesla code
  76, #pragma acc loop gang /* blockIdx.x */
  78, #pragma acc loop vector(128) /* threadIdx.x */
  78, Loop is parallelizable

Now data movement only happens at our data region.
OPENACC SPEED-UP

Speed-up

<table>
<thead>
<tr>
<th>Speed-Up</th>
<th>SERIAL</th>
<th>MULTICORE</th>
<th>V100</th>
<th>V100 (DATA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00X</td>
<td>1.00X</td>
<td>3.23X</td>
<td>41.80X</td>
<td>42.99X</td>
</tr>
</tbody>
</table>
WHAT WE’VE LEARNED SO FAR

- CUDA Unified (Managed) Memory is a powerful porting tool
- GPU programming without managed memory often requires data shaping
- Moving data at each loop is often inefficient
- The OpenACC Data region can decouple data movement and computation
DATA SYNCHRONIZATION
**OPENACC UPDATE DIRECTIVE**

**update**: Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region

Clauses:

- **self**: makes host data agree with device data
- **device**: makes device data agree with host data

```C/C++
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

```Fortran
!$acc update self(x(1:end_index))
!$acc update device(x(1:end_index))
```
The data must exist on both the CPU and device for the update directive to work.

```
#pragma acc update device(A[0:N])
```

```
#pragma acc update self(A[0:N])
```
SYNCHRONIZE DATA WITH UPDATE

- Sometimes data changes on the host or device inside a data region
- Ending the data region and starting a new one is expensive
- Instead, update the data so that the host and device data are the same
- Examples: File I/O, Communication, etc.

```c
int* A=(int*) malloc(N*sizeof(int))
#pragma acc data create(A[0:N])
while( timesteps++ < numSteps )
{
    #pragma acc parallel loop
    for(int i = 0; i < N; i++){
        a[i] *= 2;
    }

    if (timestep % 100 ) {
        #pragma acc update self(A[0:N])
        checkpointAToFile(A, N);
    }
}
```
UNSTRUCTURED DATA DIRECTIVES
UNSTRUCTURED DATA DIRECTIVES

Enter Data Directive

- Data lifetimes aren’t always neatly structured.
- The **enter data** directive handles device memory allocation
- You may use either the **create** or the **copyin** clause for memory allocation
- The enter data directive is **not** the start of a data region, because you may have multiple enter data directives

```c
#pragma acc enter data clauses
< Sequential and/or Parallel code >
#pragma acc exit data clauses
```

```c
!$acc enter data clauses
< Sequential and/or Parallel code >
!$acc exit data clauses
```
UNSTRUCTURED DATA DIRECTIVES

Exit Data Directive

- The **exit data** directive handles device memory **deallocation**
- You may use either the **delete** or the **copyout** clause for memory deallocation
- You should have as many **exit data** for a given array as **enter data**
- These can exist in different functions

```c
#pragma acc enter data clauses
< Sequential and/or Parallel code >
#pragma acc exit data clauses
```

```c
!$acc enter data clauses
< Sequential and/or Parallel code >
!$acc exit data clauses
```
UNSTRUCTURED DATA CLAUSES

- **copyin** (list) Allocates memory on device and copies data from host to device on enter data.
- **copyout** (list) Allocates memory on device and copies data back to the host on exit data.
- **create** (list) Allocates memory on device without data transfer on enter data.
- **delete** (list) Deallocates memory on device without data transfer on exit data.
UNSTRUCTURED DATA DIRECTIVES

Basic Example

```c
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
}
```
UNSTRUCTURED DATA DIRECTIVES
Basic Example

```c
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
}

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```
### UNSTRUCTURED VS STRUCTURED

#### With a simple code

<table>
<thead>
<tr>
<th>Unstructured</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Can have multiple starting/ending points</td>
<td>▪ Must have explicit start/end points</td>
</tr>
<tr>
<td>▪ Can branch across multiple functions</td>
<td>▪ Must be within a single function</td>
</tr>
<tr>
<td>▪ Memory exists until explicitly deallocated</td>
<td>▪ Memory only exists within the data region</td>
</tr>
</tbody>
</table>

#### Code Examples

**Unstructured**

```c
#pragma acc enter data copyin(a[0:N],b[0:N]) \ create(c[0:N])

#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
}

#pragma acc exit data copyout(c[0:N]) \ delete(a,b)
```

**Structured**

```c
#pragma acc data copyin(a[0:N],b[0:N]) \
    copyout(c[0:N])
{
    #pragma acc parallel loop
    for(int i = 0; i < N; i++){
        c[i] = a[i] + b[i];
    }
}
```
C++ STRUCTS/CLASSES
With dynamic data members

- C++ Structs/Classes work the same exact way as they do in C
- The main difference is that now we have to account for the implicit “this” pointer

```cpp
class vector {
    private:
        float *arr;
        int n;
    public:
        vector(int size){
            n = size;
            arr = new float[n];
            #pragma acc enter data copyin(this)
            #pragma acc enter data create(arr[0:n])
        }
        ~vector(){
            #pragma acc exit data delete(arr)
            delete(arr);
        }
};
```
C++ CLASS DATA SYNCHRONIZATION

- Since data is encapsulated, the class needs to be extended to include data synchronization methods
- Including explicit methods for host/device synchronization may ease C++ data management
- Allows the class to be able to naturally handle synchronization, creating less code clutter

```cpp
void accUpdateSelf() {
    #pragma acc update self(arr[0:n])
}
void accUpdateDevice() {
    #pragma acc update device(arr[0:n])
}
```
UNSTRUCTURED DATA DIRECTIVES

Branching across multiple functions

```c
int* allocate_array(int N){
    int* ptr = (int*) malloc(N * sizeof(int));
    #pragma acc enter data create(ptr[0:N])
    return ptr;
}

void deallocate_array(int* ptr){
    #pragma acc exit data delete(ptr)
    free(ptr);
}

int main(){
    int* a = allocate_array(100);
    #pragma acc kernels
    {
        a[0] = 0;
    }
    deallocate_array(a);
}
```

- In this example enter data and exit data are in different functions
- This allows the programmer to put device allocation/deallocation with the matching host versions
- This pattern is particularly useful in C++, where structured scopes may not be possible.
GANGS, WORKERS, AND VECTORS DEMYSTIFIED
GANGS, WORKERS, AND VECTORS DEMYSTIFIED
GANGS, WORKERS, AND VECTORS DEMYSTIFIED
How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.
Even if we increase the size of his roller, he can only paint so fast.

We need more workers!
Multiple workers can do more work and share resources, if organized properly.
GANGS, WORKERS, AND VECTORS DEMYSTIFIED

By organizing our workers into groups (gangs), they can effectively work together within a floor.

Groups (gangs) on different floors can operate independently.

Since gangs operate independently, we can use as many or few as we need.
GANGS, WORKERS, AND VECTORS DEMYSTIFIED

Even if there’s not enough gangs for each floor, they can move to another floor when ready.
Our painter is like an OpenACC **worker**, he can only do so much.

His roller is like a **vector**, he can move faster by covering more wall at once.

Eventually we need more workers, which can be organized into **gangs** to get more done.
LOOP OPTIMIZATIONS
OPENACC LOOP DIRECTIVE

Expressing parallelism

- Mark a single for loop for parallelization
- Allows the programmer to give additional information and/or optimizations about the loop
- Provides many different ways to describe the type of parallelism to apply to the loop
- Must be contained within an OpenACC compute region (either a kernels or a parallel region) to parallelize loops

C/C++

```c
#pragma acc loop
for(int i = 0; i < N; i++)
// Do something
```

Fortran

```
!$acc loop
do i = 1, N
  ! Do something
```
COLLAPSE CLAUSE

- `collapse( N )`
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- This can be extremely useful for increasing memory locality, as well as creating larger loops to expose more parallelism

```c
#pragma acc parallel loop collapse(2)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma acc loop reduction(+:tmp)
    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```
COLLAPSE CLAUSE

collapse( 2 )

<table>
<thead>
<tr>
<th></th>
<th>(0,0)</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
<td></td>
</tr>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td></td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td></td>
</tr>
<tr>
<td>(3,0)</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td></td>
</tr>
</tbody>
</table>

#pragma acc parallel loop collapse(2)
for( i = 0; i < 4; i++ )
    for( j = 0; j < 4; j++ )
        array[i][j] = 0.0f;
COLLAPSE CLAUSE

When/Why to use it

- A single loop might not have enough iterations to parallelize
- Collapsing outer loops gives more scalable (gangs) parallelism
- Collapsing inner loops gives more tight (vector) parallelism
- Collapsing all loops gives the compiler total freedom, but may cost data locality
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) collapse(2) \ 
    copyin(A[0:n*m]) copy(Anew[0:n*m])
    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 collapse(2) \ 
    copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    } iter++;

Collapse 2 loops into one for more flexibility in parallelizing.
OPENACC SPEED-UP

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
TILE CLAUSE

- **tile (x, y, z, ...)**
- Breaks multidimensional loops into “tiles” or “blocks”
- Can increase data locality in some codes
- Will be able to execute multiple “tiles” simultaneously

```c
#pragma acc kernels loop tile(32, 32)
for (i = 0; i < size; i++)
  for (j = 0; j < size; j++)
    for (k = 0; k < size; k++)
      c[i][j] += a[i][k] * b[k][j];
```
TILE CLAUSE

```c
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
    for(int y = 0; y < 4; y++){
        array[x][y]++;
    }
}
```
OPTIMIZED DATA MOVEMENT

```c
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) tile(32,32) \ 
        copyin(A[0:n*m]) copyin(Anew[0:n*m])
    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 tile(32,32) \ 
        copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Create 32x32 tiles of the loops to better exploit data locality.
TILING RESULTS (V100)

The collapse clause often requires an exhaustive search of options.

For our example code…

• CPU saw no benefit from tiling
• GPU saw anywhere from a 23% loss of performance to a 10% improvement
OPENACC SPEED-UP

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses:

- **gang** – Mark this loop for gang parallelism
- **worker** – Mark this loop for worker parallelism
- **vector** – Mark this loop for vector parallelism

These can be combined on the same loop.

```c
#pragma acc parallel loop gang
for( i = 0; i < size; i++ )
#pragma acc loop worker
  for( j = 0; j < size; j++ )
    #pragma acc loop vector
      for( k = 0; k < size; k++ )
        c[i][j] += a[i][k] * b[k][j];
```

```c
#pragma acc parallel loop \ collapse(3) gang vector
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```
SEQ CLAUSE

- The `seq` clause (short for sequential) will tell the compiler to run the loop sequentially.
- In the sample code, the compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially.
- The compiler may automatically apply the `seq` clause to loops as well.

```c
#pragma acc parallel loop
for( i = 0; i < size; i++ )
#pragma acc loop
for( j = 0; j < size; j++ )
#pragma acc loop seq
for( k = 0; k < size; k++ )
c[i][j] += a[i][k] * b[k][j];
```
ADJUSTING GANGS, WORKERS, AND VECTORS

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses.

- **num_gangs(N)** – Generate N gangs for this parallel region
- **num_workers(M)** – Generate M workers for this parallel region
- **vector_length(Q)** – Use a vector length of Q for this parallel region

```c
#pragma acc parallel num_gangs(2) \num_workers(2) vector_length(32)
{
#pragma acc loop gang worker
for(int x = 0; x < 4; x++){
    #pragma acc loop vector
    for(int y = 0; y < 32; y++){
        array[x][y]++;
    }
}
} 
```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) collapse(2) vector_length(1024) \
        copyin(A[0:n*m]) copyin(Anew[0:n*m])
    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 collapse(2) vector_length(1024) \
        copyin(Anew[0:n*m]) copyout(A[0:n*m])
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
OPENACC SPEED-UP

PGI 18.7, NVIDIA Tesla V100, Intel i9-7900X CPU @ 3.30GHz
LOOP OPTIMIZATION RULES OF THUMB

- It is rarely a good idea to set the number of gangs in your code, let the compiler decide.
- Most of the time you can effectively tune a loop nest by adjusting only the vector length.
- It is rare to use a worker loop. When the vector length is very short, a worker loop can increase the parallelism in your gang.
- When possible, the vector loop should step through your arrays.
- Gangs should come from outer loops, vectors from inner.
CLOSING REMARKS
KEY CONCEPTS

In this lab we discussed...

▪ Some details that are available to use from a GPU profile
▪ Gangs, Workers, and Vectors Demystified
▪ Collapse clause
▪ Tile clause
▪ Gang/Worker/Vector clauses
OPENACC RESOURCES

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

FREE Compilers

PGI Community Edition

https://www.openacc.org/community#slack

Resources
https://www.openacc.org/resources

Success Stories
https://www.openacc.org/success-stories

Compilers and Tools
https://www.openacc.org/tools

Events
https://www.openacc.org/events
INTRODUCTION TO OPENMP
OpenMP Worksharing

- PARALLEL Directive
- Spawns a team of threads
- Execution continues redundantly 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.
while ( error > tol && iter < iter_max )
{
    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++ ) {
            error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
        }
    }

    #pragma omp 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);
}
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    #pragma omp parallel
    {
        #pragma omp for reduction(max:error)
        for( int j = 1; j < n-1; j++) {
            for( int i = 1; i < m-1; i++ ) {
                error = fmax(error, fabs(Anew[j][i] - A[j][i]));
            }
        }

        #pragma omp 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);
}
```c
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    #pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        #pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        #pragma omp simd
        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);
}
```

Some compilers want a SIMD directive to *simdize* on CPUs.
Targeting the GPU
OpenMP Offloading

TARGET Directive

Offloads execution and associated data from the CPU to the GPU

- The *target device* owns the data, accesses by the CPU during the execution of the target region are forbidden.
- Data used within the region may be implicitly or explicitly *mapped* to the device.
- All of OpenMP is allowed within target regions, but only a subset will run well on GPUs.
OpenMP Data Mapping
while ( error > tol && iter < iter_max )
{
    error = 0.0;
    #pragma omp target
    {
        #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]));
            }
        }
    }
    #pragma omp 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
", iter, error);
while (error > tol && iter < iter_max)
{
    error = 0.0;
    #pragma omp target
    {
        #pragma omp parallel for reduction(max:error)
        for(int j = 1; j < n - 1; j++)
        {
            for(int i = 1; i < m - 1; i++)
            {
                error = fmax(error, fabs(Anew[j][i] - A[j][i]));
            }
        }
        #pragma omp 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);
}
while ( error > tol && iter < iter_max )
{
    error = 0.0;
    #pragma omp target map(to:Anew[:n+2][:m+2]) \ 
        map(tofrom:A[:n+2][:m+2],tofrom:error)
    {
        #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]));
                }
    }
    #pragma omp 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);
Target the GPU

while ( error > tol && iter < iter_max )
{
    error = 0.0;
    #pragma omp target map(to:Anew[:n+2][:m+2]) \ 
        map(tofrom:A[:n+2][:m+2],tofrom:error)
    {
        #pragma omp parallel for reduction(max:error)
            for( int j = 1; j < n - 1; j++)
                for( int i = 1; i < m - 1; i++)
                error = fmax(error, fabs(Anew[j][i] - A[j][i]));
    }
    #pragma omp 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);
Excessive Data Movement

A and Anew arrays are copied for each iteration.
OpenMP Data Offloading

TARGET DATA Directive

Offloads data from the CPU to the GPU, but not execution

- The *target device* owns the data, accesses by the CPU during the execution of contained target regions are forbidden.

- Useful for sharing data between TARGET regions

- NOTE: A TARGET region *is a* TARGET DATA region.
#pragma omp target data map(to:Anew) map(A)
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma omp target map(error)
{
#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]) );
        }
    }
#pragma omp parallel for
    for( int j = 1; j < n-1; j++ )
    {
        for( int i = 1; i < m-1; i++ )
        {
            A[j][i] = Anew[j][i];
        }
    }
} // OMP TARGET

Explicitly map the data once for the whole simulation.
Improved Data Movement

A and Anew arrays are copied before and after the simulation, only “error” is copied per iteration.
Speed-Up

IBM Power9, NVIDIA Tesla V100
GPU Architecture Basics

GPUs are composed of 1 or more independent parts, known as Streaming Multiprocessors ("SMs")

*Threads* are organized into *threadblocks*.

Threads within the same threadblock run on an SM and can synchronize.

Threads in different threadblocks (even if they’re on the same SM) cannot synchronize.

CUDA 9 introduces cooperative groups, they may address this.
Teaming Up
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.
#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++ )
        {
            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++;
}
Speed-Up

![Bar chart showing speed-up comparisons between XLC, CLANG, and GCC for different workloads: CPU Best, target teams, and target teams simd.](image)

IBM Power9, NVIDIA Tesla V100
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

```c
#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.
Splitting Teams & Parallel

```c
#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]));
}
}
```

---

Distribute the “j” loop over teams.

```c
#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];
}
}
```

---

Workshare the “i” loop over threads.
Speed-Up

IBM Power9, NVIDIA Tesla V100