while (dt > MAX_TEMP_ERROR && iteration <= max_iterations) {

#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
    }
}

dt = 0.0; // reset largest temperature change

#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
        Temperature_last[i][j] = Temperature[i][j];
    }
}

if((iteration % 100) == 0) {
    track_progress(iteration);
}

iteration++;
}
Exercise 1 Fortran Solution

do while ( dt > max_temp_error .and. iteration <= max_iterations )

!$acc kernels
do j=1,columns
do i=1,rows
    temperature(i,j) = 0.25*(temperature_last(i+1,j) + temperature_last(i-1,j) + &
    temperature_last(i,j+1) + temperature_last(i,j-1) )
enddo
enddo
!$acc end kernels

dt = 0.0

!$acc kernels
do j=1,columns
do i=1,rows
    dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
    temperature_last(i,j) = temperature(i,j)
enddo
enddo
!$acc end kernels

if( mod(iteration,100).eq.0 ) then
    call track_progress(temperature, iteration)
endif

iteration = iteration+1
enddo
Exercise 1: Compiler output (C)

instr009@h2ologin2:~/Update> cc -acc -Minfo=accel laplace_bad_acc.c
main:

62, Generating present_or_copyout(Temperature[1:1000][1:1000])
  Generating present_or_copyin(Temperature_last[0:][0:])
  Generating NVIDIA code
  Generating compute capability 1.3 binary
  Generating compute capability 2.0 binary
  Generating compute capability 3.0 binary

63, Loop is parallelizable
64, Loop is parallelizable

Accelerator kernel generated
  63, #pragma acc loop gang /* blockIdx.y */
  64, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */

73, Generating present_or_copyin(Temperature[1:1000][1:1000])
  Generating present_or_copyin(Temperature_last[1:1000][1:1000])
  Generating NVIDIA code
  Generating compute capability 1.3 binary
  Generating compute capability 2.0 binary
  Generating compute capability 3.0 binary

74, Loop is parallelizable
75, Loop is parallelizable

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

76, Max reduction generated for dt

Compiler was able to parallelize
Compiler was able to parallelize
Exercise 1: Performance
3372 steps to convergence

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Serial</td>
<td>18</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>9.4</td>
<td>1.99</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>4.7</td>
<td>3.98</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>2.5</td>
<td>7.48</td>
</tr>
<tr>
<td>CPU 16 OpenMP threads</td>
<td>1.4</td>
<td>13.4</td>
</tr>
<tr>
<td>CPU 28 OpenMP threads</td>
<td>0.9</td>
<td>21.5</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>29</td>
<td>0.6x</td>
</tr>
</tbody>
</table>
What’s with the OpenMP?

- We can compare our GPU results to the best the multi-core CPUs can do.

- If you are familiar with OpenMP, or even if you are not, you can compile and run the OpenMP enabled versions in your OpenMP directory as:

  ```
  pgcc -mp laplace_omp.c or pgf90 -mp laplace_omp.f90
  ```

  then to run on 8 threads do:

  ```
  export OMP_NUM_THREADS=8
  a.out
  ```

- Note that you probably only have 8 real cores if you are still on a GPU node. Do something like “interact -n28” if you want a full node of cores.
What went wrong?

`export PGI_ACC_TIME=1` to activate profiling and run again:

Accelerator Kernel Timing data
/mnt/a/u/training/instr009/Update/laplace_bad_acc.c
main NVIDIA devicenum=0
time(us): 22,902,870
62: compute region reached 3372 times
   62: data copyin reached 3372 times
     device time(us): total=4,561,531 max=1,362 min=1,350 avg=1,352
64: kernel launched 3372 times
   grid: [8x1000] block: [128]
   device time(us): total=441,105 max=268 min=129 avg=130
   elapsed time(us): total=487,585 max=282 min=141 avg=144
70: data copyout reached 3372 times
   device time(us): total=4,063,246 max=1,230 min=1,202 avg=1,204
73: compute region reached 3372 times
   73: data copyin reached 6744 times
     device time(us): total=9,135,367 max=1,428 min=1,346 avg=1,354
75: kernel launched 3372 times
   grid: [8x1000] block: [128]
   device time(us): total=546,820 max=296 min=155 avg=162
   elapsed time(us): total=593,424 max=309 min=171 avg=175
75: reduction kernel launched 3372 times
   grid: [1] block: [256]
   device time(us): total=91,638 max=161 min=25 avg=27
   elapsed time(us): total=136,871 max=174 min=38 avg=40
82: data copyout reached 3372 times
   device time(us): total=4,063,163 max=1,259 min=1,202 avg=1,204
Basic Concept
Simplified, but sadly true

CPU Memory

CPU

GPU Memory

GPU

PCI Bus
Multiple Times Each Iteration

CPU Memory

- A(i-1,j)
- A(i,j-1)
- A(i,j)
- A(i+1,j)

GPU Memory

- A(i-1,j)
- A(i,j-1)
- A(i,j)
- A(i+1,j)

CPU

GPU
Excessive Data Transfers

```c
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    Temperature, Temperature_old resident on host

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + ...
        }
    }

    Temperature, Temperature_old resident on device

    Temperature, Temperature_old resident on host

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + ...
        }
    }

    Temperature, Temperature_old resident on device

    dt = 0.0;
}
```

4 copies happen every iteration of the outer while loop!
Data Management

The First, Most Important, and possibly Only OpenACC Optimization
while (dt > MAX_TEMP_ERROR && iteration <= max_iterations) {
    #pragma acc kernels
    for (i = 1; i <= ROWS; i++) {
        for (j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }
    dt = 0.0;
    #pragma acc kernels loop reduction (max:dt)
    for (i = 1; i <= ROWS; i++) {
        for (j = 1; j <= COLUMNS; j++) {
            dt = fmax(fabs(Temperature[i][j] - Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }
    .
    .
    iteration++;
}
Data Construct Syntax and Scope

Fortran

!$acc data [clause ...]
    structured block
!$acc end data

C

#pragma acc data [clause ...]
{
    structured block
}

**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

Compilers sometimes cannot determine the size of arrays, so we must specify explicitly using data clauses with an array “shape”. The compiler will let you know if you need to do this. Sometimes, you will want to for your own efficiency reasons.

C

```c
#pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])
```

Fortran

```fortran
!$acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```

Fortran uses `start:end` and C uses `start:length`

Data clauses can be used on data, kernels or parallel
int main(int argc, char *argv[]) {
    int i;
    double A[2000], B[1000], C[1000];
    #pragma acc kernels
    for (i = 0; i < 1000; i++) {
        A[i] = 4 * i;
        B[i] = B[i] + 2;
        C[i] = A[i] + 2 * B[i];
    }
}

gcc -acc -Minfo=accel loops.c
main:
  6, Generating present_or_copyout(C[:])
  Generating present_or_copy(B[:])
  Generating present_or_copyout(A[:1000])
  Generating NVIDIA code
  7, Loop is parallelizable
  Accelerator kernel generated
Data Regions Have Real Consequences

**Simplest Kernel**

```c
int main(int argc, char** argv){
    float A[1000];
    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++ ){
        A[iter] = 1.0;
    }
    A[10] = 2.0;
}
```

**With Global Data Region**

```c
int main(int argc, char** argv){
    float A[1000];
    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++ ){
        A[iter] = 1.0;
    }
    #pragma acc data copy(A)
    {
        A[10] = 2.0;
    }
}
```

**Output:**

**Simplest Kernel**

- A[10] = 2.0

**With Global Data Region**

- A[10] = 1.0
```c
int main(int argc, char** argv){
    float A[1000];
    #pragma acc data copy(A)
    {
        #pragma acc kernels
        for( int iter = 1; iter < 1000 ; iter++)
        {
            A[iter] = 1.0;
        }
        A[10] = 2.0;
    }
}
```

Output:
```
A[10] = 1.0
```
Data Movement Decisions

- Much like loop data dependencies, sometime the compiler needs your human intelligence to make high-level decisions about data movement. Otherwise, it must remain conservative - sometimes at great cost.

- You must think about when data truly needs to migrate, and see if that is better than the default.

- Besides the scope based data clauses, there are OpenACC options to let us manage data movement more intensely or asynchronously. We could manage the above behavior with the `update` construct:

  Fortran :  
  !$acc update [host(), device(), ...] 

  C:  
  #pragma acc update [host(), device(), ...]

  Ex:  #pragma acc update host(Temp_array)  //Gets host a current copy
Exercise 2: Use acc data to minimize transfers
(about 40 minutes)

Q: What speedup can you get with data + kernels directives?

• Start with your Exercise 1 solution or grab laplace_bad_acc.c/f90 from the Solutions subdirectory. This is just the solution of the last exercise.

• Add data directives where it helps.
  • Think: when should I move data between host and GPU? Think how you would do it by hand, then determine which data clauses will implement that plan.
  • Hint: you may find it helpful to ignore the output at first and just concentrate on getting the solution to converge quickly (at 3372 steps). Then worry about updating the printout.
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature)
        track_progress(iteration);
    }

    iteration++;
}

No data movement in this block.

Except once in a while here.
Exercise 2 Fortran Solution

```fortran
!$acc data copy(temperature_last), create(temperature)

do while ( dt > max_temp_error .and. iteration <= max_iterations)

!$acc kernels

  do j=1,columns
    do i=1,rows
      temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+温度_last(i,j+1)+temperature_last(i,j-1))
    enddo
  enddo

!$acc end kernels

dt=0.0

!copy grid to old grid for next iteration and find max change

!$acc kernels

  do j=1,columns
    do i=1,rows
      dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
      temperature_last(i,j) = temperature(i,j)
    enddo
  enddo

!$acc end kernels

!periodically print test values

if( mod(iteration,100).eq.0 ) then

!$acc update host(temperature) 

  call track_progress(temperature, iteration)

endif

iteration = iteration+1

enddo

!$acc end data
```

Extra efficient:

Extra efficient:

!$acc update host(temperature(columns-5:columns,rows-5:rows))

Except bring back a copy here
# Exercise 2: Performance

3372 steps to convergence

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Serial</td>
<td>18</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>9.4</td>
<td>1.99</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>4.7</td>
<td>3.98</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>2.5</td>
<td>7.48</td>
</tr>
<tr>
<td>CPU 16 OpenMP threads</td>
<td>1.4</td>
<td>13.4</td>
</tr>
<tr>
<td>CPU 28 OpenMP threads</td>
<td>0.9</td>
<td>21.5</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>1.5</td>
<td>12</td>
</tr>
</tbody>
</table>
OpenACC or OpenMP?

Don’t draw any grand conclusions yet. We have gotten impressive speedups from both approaches. But our problem size is pretty small. Our main data structure is:

\[ 1000 \times 1000 = 1 \text{M elements} = 8 \text{MB of memory} \]

We have 2 of these (temperature and temperature\_last) so we are using roughly **16 MB** of memory. Not very large. When divided over cores it gets even smaller and can easily fit into cache.

The algorithm is very realistic, but the memory bandwidth stress is very low.
OpenACC or OpenMP on Larger Data?

We can easily scale this problem up, so why don’t I? Because it is nice to have exercises that finish in a few minutes or less.

We scale this up to 10K x 10K (1.6 GB problem size) for the hybrid challenge. These numbers start to look a little more realistic. But the serial code takes over 30 minutes to finish. That would have gotten us off to a slow start!

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Serial</td>
<td>2187</td>
<td>--</td>
</tr>
<tr>
<td>CPU 16 OpenMP threads</td>
<td>183</td>
<td>12</td>
</tr>
<tr>
<td>CPU 28 OpenMP threads</td>
<td>162</td>
<td>13.5</td>
</tr>
<tr>
<td>OpenACC</td>
<td>103</td>
<td>21</td>
</tr>
</tbody>
</table>

10K x 10K Problem Size
Unified Memory
- Unified address space allows us to pretend we have shared memory
- Skip data management, hope it works, and then optimize if necessary
- For dynamically allocated memory can eliminate need for pointer clauses

NVLink
- One route around PCI bus (with multiple GPUs)
Further speedups

- OpenACC gives us even more detailed control over parallelization
  - Via gang, worker, and vector clauses

- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code

- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance

- But you have already gained most of any potential speedup, and you did it with a few lines of directives!
General Principles: Finding Parallelism In Code

- Nested for/do loops are best for parallelization
  - Large loop counts are best
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
  - Use subscripted arrays, rather than pointer-indexed arrays (C)
- Data regions should avoid wasted transfers
  - If applicable, could use directives to explicitly control sizes
- Various other annoying things can interfere with accelerated regions
  - IO
  - Limitations on function calls and nested parallelism (relaxed much in 2.0)
Is OpenACC Living Up To My Claims?

- High-level. No involvement of OpenCL, CUDA, etc.

- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.

- Efficient. Experience show very favorable comparison to low-level implementations of same algorithms. *kernels* is magical!

- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.

- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be *quick*. 
In Conclusion…

OpenMP

OpenACC

MPI