Parallel Programming

Data-Parallel Programming with OpenACC

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Acknowledgement

- This set of slides is adopted from
  - Online course organized by OpenACC.org, the Pittsburgh Supercomputing Center, and NVIDIA

Outline

- Introduction to OpenACC
  - What is OpenACC?
  - OpenACC Kernel Directives
  - Data Dependencies
- Programming with OpenACC
  - Case Study: Laplace Solver
  - Data Management
  - Analyzing/Profiling
- Optimizing and Best Practices for OpenACC
  - OpenACC Parallel Directives
  - OpenACC Loop Directives
  - Managed Memory
  - Unstructed Data Directives
  - C/C++ Struct/Classes
  - OpenACC Routine Directives
- References

WHAT IS OPENACC?

3 WAYS TO ACCELERATE APPLICATIONS

- Math libraries
  - cuBLAS, cuFFT, cuRAND, cuSPARSE, etc.
- Parallel algorithms
  - Thrust
- Deep learning
  - cuDNN, TensorRT, Jarvis, etc.

Applications

Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code OpenACC	Most Performance Most Flexibility CUDA, OpenCL

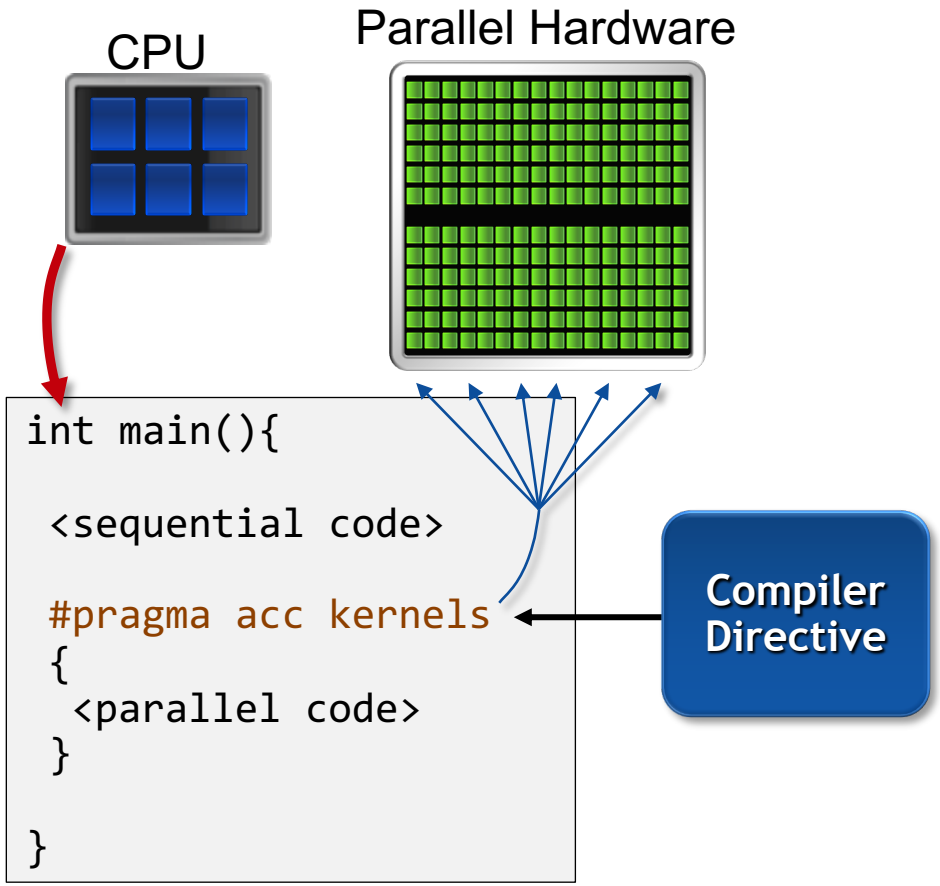
OpenACC
More Science, Less Programming

OpenACC is a directives-based programming approach to **parallel computing** designed for **performance** and **portability** on CPUs and accelerators for HPC.

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

OPENACC DIRECTIVES



1. Simple compiler hints from programmer
2. Compiler generates parallel threaded code
3. Ignorant compiler just sees some comments.

Incremental

Single Source

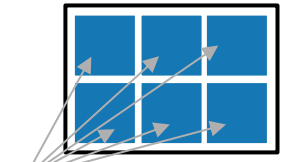
Low Learning Curve

More on this later!

OpenACC
More Science, Less Programming

FAMILIAR TO OPENMP PROGRAMMERS

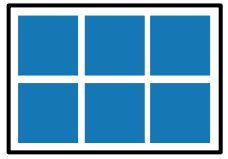
CPU

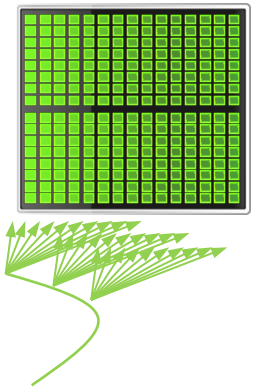


```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

OpenMP

CPU **Parallel Hardware**





```
main() {
  double pi = 0.0; long i;

  #pragma acc kernels
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

printf("pi = %f\n", pi/N);
}
```

OpenACC

More on this later!

OPENACC: KEY ADVANTAGES

- **High-level**. Minimal modifications to the code. Less than with OpenCL, CUDA, etc. Non-GPU programmers can play along.
- **Single source**. No GPU-specific code. Compile the same program for accelerators or serial.
- **Efficient**. Experience shows very favorable comparison to low-level implementations of same algorithms.
- **Performance portable**. Supports CPUs, 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*.

TRUE OPEN STANDARD

- Full OpenACC 1.0 and 2.0 and now 3.0 specifications available at http://www.openacc.org
- Quick reference card also available and useful: https://www.openacc.org/resources
- Implementations available now from PGI, Cray, and GCC: https://www.openacc.org/tools
- GCC version of OpenACC in 5.x and 6.x, but use 10.x: https://www.openacc.org/tools
- Best free option is very probably PGI Community Edition: http://www.pgroup.com/products/community.htm

OpenACC.org Members

OPENACC FOR EVERYONE

New PGI Community Edition Now Available

DOWNLOAD NOW

	FREE PGI Community Edition	PGI Professional Edition	PGI Enterprise Edition
PROGRAMMING MODELS OpenACC, CUDA Fortran, OpenMP, C/C++/Fortran Compilers and Tools	✓	✓	✓
PLATFORMS X86, OpenPOWER, NVIDIA GPU	✓	✓	✓
UPDATES	1-2 times a year	6-9 times a year	6-9 times a year
SUPPORT	User Forums	PGI Support	PGI Professional Services
LICENSE	Annual	Perpetual	Volume/Site

OpenACC
More Science, Less Programming

OPENACC KERNEL DIRECTIVES

A SIMPLE EXAMPLE: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Somewhere in main
// call SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end kernels
end subroutine saxpy


...
$ From main program
$ call SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

OpenACC
More Science, Less Programming

KERNELS: OUR FIRST OPENACC DIRECTIVE

We request that each loop execute as a separate *kernel* on the GPU. This is an incredibly powerful directive.

```
!$acc kernels
    do i=1,n
       a(i) = 0.0
       b(i) = 1.0
       c(i) = 2.0
    end do

    do i=1,n
       a(i) = b(i) + c(i)
    end do
!$acc end kernels
```

kernel 1

kernel 2

Kernel:
A parallel routine to run on the parallel hardware

GENERAL DIRECTIVE SYNTAX AND SCOPE

C

```
#pragma acc kernels [clause …]
{
        structured block
}
```

Fortran

```
!$acc kernels [clause …]
    structured block
!$acc end kernels
```

I may indent the directives at the natural code indentation level for readability. It is a common practice to always start them in the first column (ala #define/#ifdef). Either is fine with C or Fortran 90 compilers.

COMPLETE SAXPY EXAMPLE CODE

```
int main(int argc, char **argv)
{
  int N = 1<<20; // 1 million floats
  if (argc > 1)
    N = atoi(argv[1]);

  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }
  saxpy(N, 3.0f, x, y);

  return 0;
}
```

```
#include <stdlib.h>

void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}
```

"I promise y is not aliased by Anything else (esp. x)"

OpenACC
More Science, Less Programming

C DETAIL: THE “RESTRICT” KEYWORD

- Standard C (as of C99).
- Important for optimization of serial as well as OpenACC and OpenMP code.
- Promise given by the programmer to the compiler for a pointer: float *restrict ptr
  Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”
- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence
- Otherwise the compiler can’t parallelize loops that access ptr
- Note: if programmer violates the declaration, behavior is undefined

COMPILE AND RUN

- C: `pgcc -acc saxpy.c` or `gcc -fopenacc saxpy.c`
- Fortran: `pgf90 -acc saxpy.f90`
- Run: `a.out`

-acc will enable OpenACC directives
Default here targets serial CPU and GPU

Compiler Output

```
pgcc -acc -Minfo=accel -ta=tesla saxpy.c

saxpy:
     8, Generating copyin(x[:n-1])
        Generating copy(y[:n-1])
        Generating compute capability 1.0 binary
        Generating compute capability 2.0 binary
     9, Loop is parallelizable
        Accelerator kernel generated
         9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
            CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
            CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

-ta=tesla will *only* target a GPU
-ta=multicore will only target a multicore CPU
-Minfo=accel turns on helpful compiler reporting

COMPARE: OPENACC AND CUDA IMPLEMENTATIONS

OpenACC: Complete SAXPY Example Code

```
#include <stdlib.h>

void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
  int N = 1<<20; // 1 million floats
  if (argc > 1)
    N = atoi(argv[1]);

  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }
  saxpy(N, 3.0f, x, y);

  return 0;
}
```

CUDA: Partial CUDA C SAXPY Code SAXPY Example Code

```
__global__ void saxpy_kernel( float a,
float* x, float* y, int n ){
  int i;
  i = blockIdx.x*blockDim.x +
threadIdx.x;
  if( i <= n ) x[i] = a*x[i] + y[i];
}
void saxpy( float a, float* x, float* y,
int n ){
  float *xd, *yd;
  cudaMalloc( (void**)&xd,
n*sizeof(float) );
  cudaMalloc( (void**)&yd,
n*sizeof(float) ); cudaMemcpy( xd, x,
n*sizeof(float),

cudaMemcpyHostToDevice );
  cudaMemcpy( yd, y, n*sizeof(float),

cudaMemcpyHostToDevice );
  saxpy_kernel<<< (n+31)/32, 32 >>>( a,
xd, yd, n );
  cudaMemcpy( x, xd, n*sizeof(float),

cudaMemcpyDeviceToHost );
  cudaFree( xd ); cudaFree( yd );
}
```

```
module kmod
 use cudafor
contains
 attributes(global) subroutine
    saxpy_kernel(A,X,Y,N)
  real(4), device :: A, X(N), Y(N)
  integer, valu :: N
  integer :: i
  i = (blockidx%x-1)*blockdim%x + threadidx%x
  if( i <= N ) X(i) = A*X(i) + Y(i)
 end subroutine
end module


subroutine saxpy( A, X, Y, N )
  use kmod
  real(4) :: A, X(N), Y(N)
  integer :: N
  real(4), device, allocatable, dimension(:)::
    &
    Xd, Yd
  allocate( Xd(N), Yd(N) )
  Xd = X(1:N)
  Yd = Y(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd,
    Yd, N)
  X(1:N) = Xd
  deallocate( Xd, Yd )
 end subroutine
```

OpenACC
More Science, Less Programming

BIG DIFFERENCE!

OpenACC vs CUDA implementations

- **CUDA: Hard to Maintain, OpenACC: Easy to Maintain**
  With CUDA, we changed the structure of the old code. Non-CUDA programmers can't understand new code. It is not even ANSI standard code.

- **CUDA: Rewrite Original Code, OpenACC: Augment Original Code**
  We have separate sections for the host code and the GPU code. Different flow of code. Serial path now gone forever.

- **CUDA: Optimized for Specific Hardware, OpenACC: One Source Everywhere**
  Where did these "32"s and other mystery numbers come from? This is a clue that we have some hardware details to deal with here.

- **CUDA: Assembler-like Programming, OpenACC: Relies on Compiler**
  Exact same situation as assembly used to be. How much hand-assembled code is still being written in HPC now that compilers have gotten so efficient?

THIS LOOKS EASY! TOO EASY…

Questions:

1. If it is this simple, why don't we just throw *kernels* in front of every loop?
2. Better yet, why doesn't the compiler do this for me?

Answers:

There are two general issues that prevent the compiler from being able to just automatically parallelize every loop:

1. Data Dependencies in Loops
2. Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results and reasonable performance

DATA DEPENDENCIES

DATA DEPENDENCIES

Most directive based parallelization consists of splitting up big do/for loops into independent chunks that the many processors can work on simultaneously.

Take, for example, a simple for loop like this:

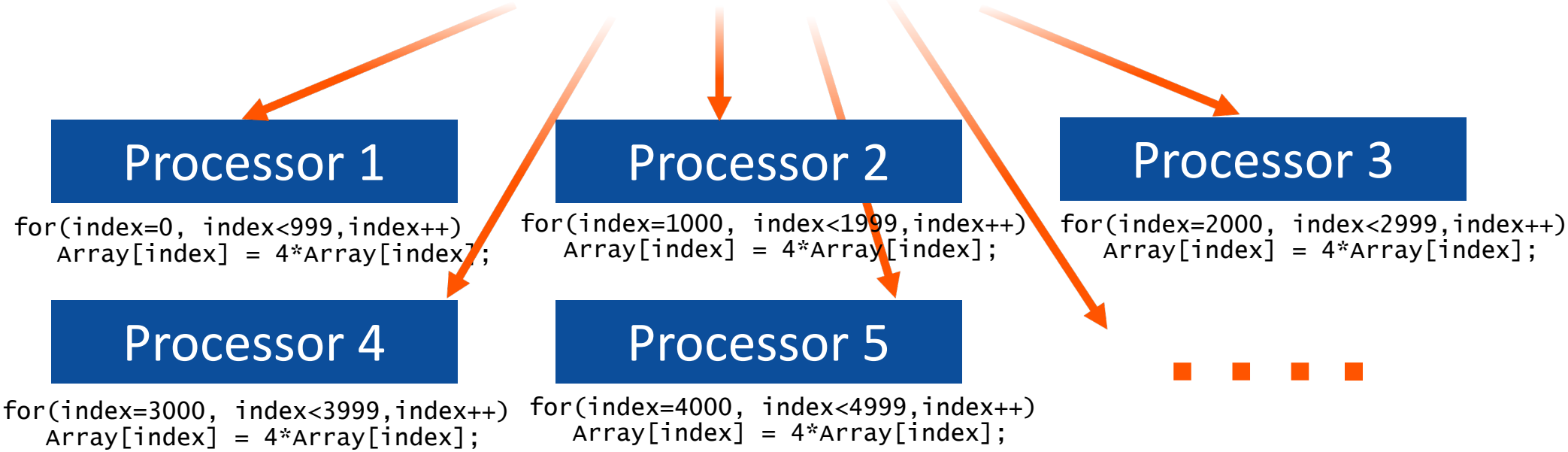
```
for(index=0, index<1000000,index++)
        Array[index] = 4 * Array[index];
```

When run on 1000 processors, it will execute something like this…

NO DATA DEPENDENCIES

A run on 1000 processors for the loop below

```
for(index=0, index<1000000,index++)
    Array[index] = 4 * Array[index];
```



WITH DATA DEPENDENCIES

But what if the loops are not entirely independent?

Take, for example, a similar loop like this:

```
for(index=1, index<1000000,index++)
      Array[index] = 4 * Array[index] – Array[index-1];
```

Added data dependency

This is a perfectly valid serial code.

WITH DATA DEPENDENCIES

Processor 1

```
for(index=0, index<999,index++)
   Array[index] = 4*Array[index]-
Array[index-1];
```

Processor 2

```
for(index=1000, index<1999,index++)
   Array[index] = 4*Array[index]-
Array[index-1];
```

Result from Processor 1

```
for(index=1000, index<1999,index++)
      Array[1000] = 4 * Array[1000] – Array[999];
```

Needs the result of Processor 1's last iteration.

If we want the correct ("same as serial") result, we need to wait until processor 1 finishes. Likewise for processors 3, 4, …

OpenACC
More Science, Less Programming

DATA DEPENDENCIES

If the compiler even *suspects* that there is a data dependency, it will, for the sake of correctness, refuse to parallelize that loop.

```
11, Loop carried dependence of 'Array' prevents parallelization
    Loop carried backward dependence of 'Array' prevents vectorization
```

As large, complex loops are quite common in HPC, especially around the most important parts of your code, the compiler will often balk most when you most need a kernel to be generated. What can you do?

HOW TO MANAGE DATA DEPENDENCIES

- Rearrange your code to make it more obvious to the compiler that there is not really a data dependency.
- Eliminate a real dependency by changing your code.
- There is a common bag of tricks developed for this as this issue goes back 40 years in HPC. Many are quite trivial to apply.
- The compilers have gradually been learning these themselves.
- Override the compiler's judgment (`independent` clause) at the risk of invalid results. Misuse of `restrict` has similar consequences.

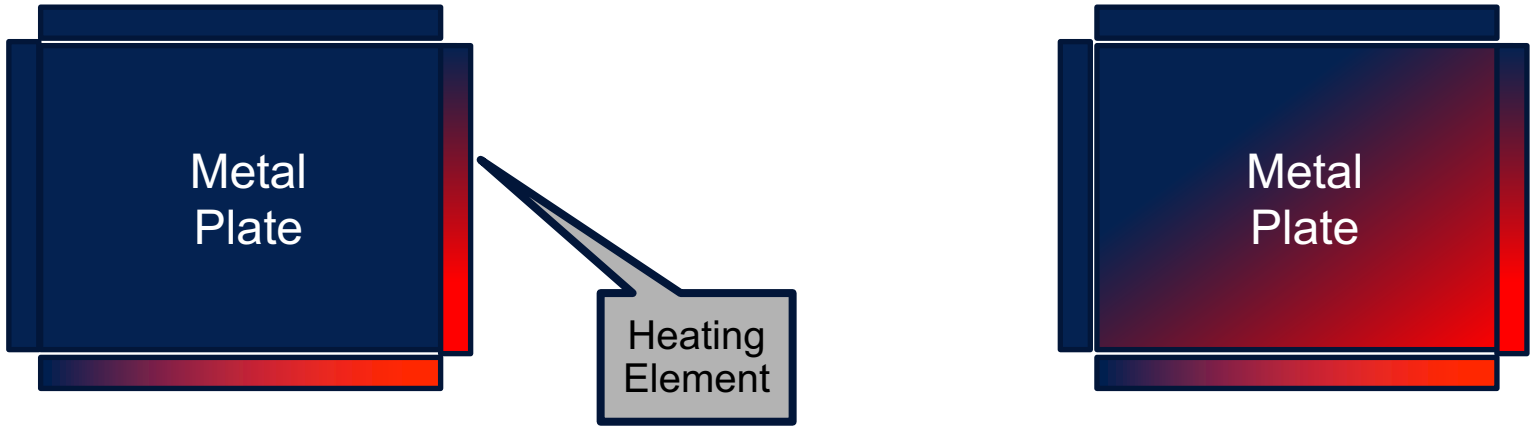
Outline

- Introduction to OpenACC
  - What is OpenACC?
  - OpenACC Kernel Directives
  - Data Dependencies
- Programming with OpenACC
  - Case Study: Laplace Solver
  - Data Management
  - Analyzing/Profiling
- Optimizing and Best Practices for OpenACC
  - OpenACC Parallel Directives
  - OpenACC Loop Directives
  - Managed Memory
  - Unstructed Data Directives
  - C/C++ Struct/Classes
  - OpenACC Routine Directives
- References

CASE STUDY: LAPLACE SOLVER

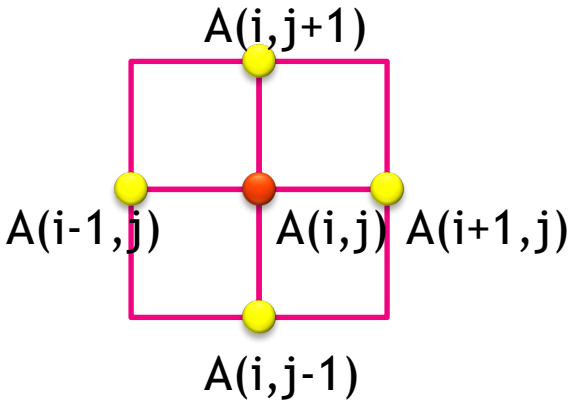
FOUNDATION EXERCISE: LAPLACE SOLVER

- It is a great simulation problem, not rigged for OpenACC.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including: electrostatics, fluid flow, and temperature
- For temperature, it is the Steady State Heat Equation:



EXERCISE FOUNDATION: JACOBI ITERATION

- The Laplace equation on a grid states that each grid point is the average of it's neighbors.
- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.
- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate.



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

SERIAL CODE IMPLEMENTATION

C

```
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]);
     }
}
```

Fortran

```
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
```

SERIAL C CODE (KERNEL)

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

     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;

     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++;

}
```

Done?

Calculate

Update temp array and find max change

Output

SERIAL C CODE SUBROUTINES

```
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

```
void track_progress(int iteration) {

  int i;

  printf("-- Iteration: %d --\n", iteration);
  for(i = ROWS-5; i <= ROWS; i++) {
     printf("[%d,%d]: %5.2f ", i, i,Temperature[i][i]);
  }
  printf("\n");
}
```

BCs could run from 0 to ROWS+1 or from 1 to ROWS. We chose the former.

WHOLE C CODE

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS    1000
#define ROWS       1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];      // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

//   helper routines
void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;                                            // grid indexes
    int max_iterations;                                  // number of iterations
    int iteration=1;                                     // current iteration
    double dt=100;                                       // largest change in t
    struct timeval start_time, stop_time, elapsed_time;  // timers

    printf("Maximum iterations [100-4000]?\n");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time,NULL); // Unix timer

    initialize();                   // initialize Temp_last including boundary conditions

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        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
        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) {
            track_progress(iteration);
        }

        iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine

    printf("\nMax error at iteration %d was %f\n", iteration-1, dt);
    printf("Total time was %f seconds.\n", elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);

}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

OpenACC
More Science, Less Programming

C WITH KERNEL DIRECTIVES

```
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++;
}
```

Generate a parallel Kernel

Generate a parallel Kernel

OpenACC
More Science, Less Programming

CPU PERFORMANCE

3372 steps to convergence

Execution	Problem Size			
	1000x1000 (Lab1)		2000x2000	
	Time (s)	Speedup	Time (s)	Speedup
CPU Serial	6.2	--	40.3	--
OpenACC 2 CPU Cores	3.0	2.07	26.2	1.54
OpenACC 4 CPU Cores	1.5	4.13	21.3	1.89

MOVING TO THE GPU

Change the target accelerator flag from “-ta=multicore” to “-ta=tesla”

```
% pgcc -acc -ta=tesla -Minfo=accel -DMAX_ITER=4000 laplace_kernels.c -o laplace.out
main:
    65, Generating implicit copyin(Temperature_last[:][:])
        Generating implicit copyout(Temperature[1:1000][1:1000])
    66, Loop is parallelizable
    67, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        66, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        67, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
    76, Generating implicit copyin(Temperature[1:1000][1:1000])
        Generating implicit copy(Temperature_last[1:1000][1:1000])
    77, Loop is parallelizable
    78, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        77, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        78, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
        79, Generating implicit reduction(max:dt)
```

RUNNING ON THE GPU

```
% ./laplace.out
…
---------- Iteration number: 3100 ------------
[995,995]: 97.58  [996,996]: 98.18  [997,997]: 98.71  [998,998]: 99.17  [999,999]: 99.55
[1000,1000]: 99.86
---------- Iteration number: 3200 ------------
[995,995]: 97.62  [996,996]: 98.21  [997,997]: 98.73  [998,998]: 99.18  [999,999]: 99.56
[1000,1000]: 99.86
---------- Iteration number: 3300 ------------
[995,995]: 97.66  [996,996]: 98.24  [997,997]: 98.75  [998,998]: 99.19  [999,999]: 99.56
[1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 40.462415 seconds.
```

Our time slows down by almost 7x from the original serial version of the code!

WHAT WENT WRONG?

```
% pgprof –cpu-profiling-mode top-down ./laplace.out
….

==455== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 57.41%  13.2519s     13488  982.49us     959ns  1.3721ms  [CUDA memcpy HtoD]
 36.08%  8.32932s     10116  823.38us  2.6230us  1.5129ms  [CUDA memcpy DtoH]
  3.42%  788.42ms      3372  233.81us  231.79us  235.79us  main_80_gpu
  2.80%  646.58ms      3372  191.75us  189.69us  194.04us  main_69_gpu
  0.29%  66.916ms      3372  19.844us  19.519us  20.223us  main_81_gpu_red


======== CPU profiling result (top down):
Time(%)      Time  Name
…
  9.15%  16.3609s  | __pgi_uacc_dataexitdone
  9.11%  16.2809s  | __pgi_uacc_dataonb
…
```

13.2 seconds copying data from host to device

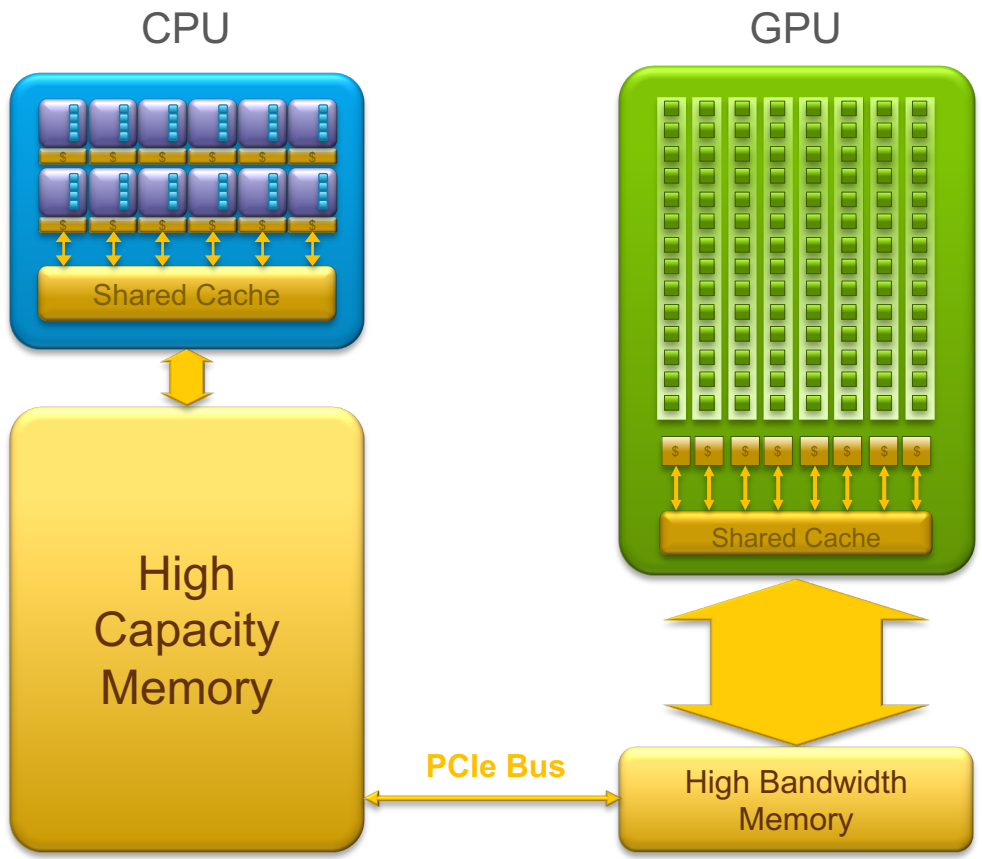
8.3 seconds copying data from device to host

Including the copy, there is an additional CPU overhead needed to create and delete the device data

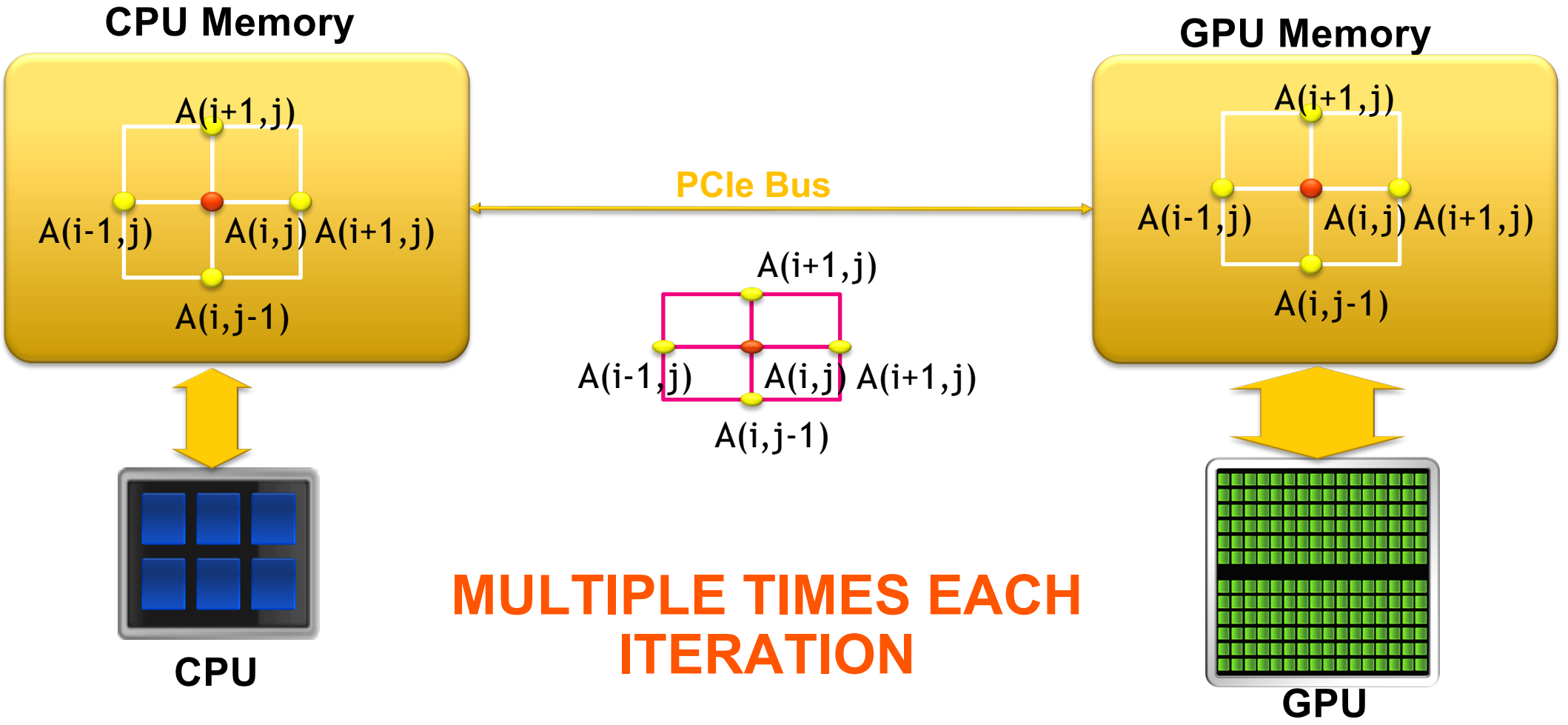
OpenACC
More Science, Less Programming

ARCHITECTURE BASIC CONCEPT

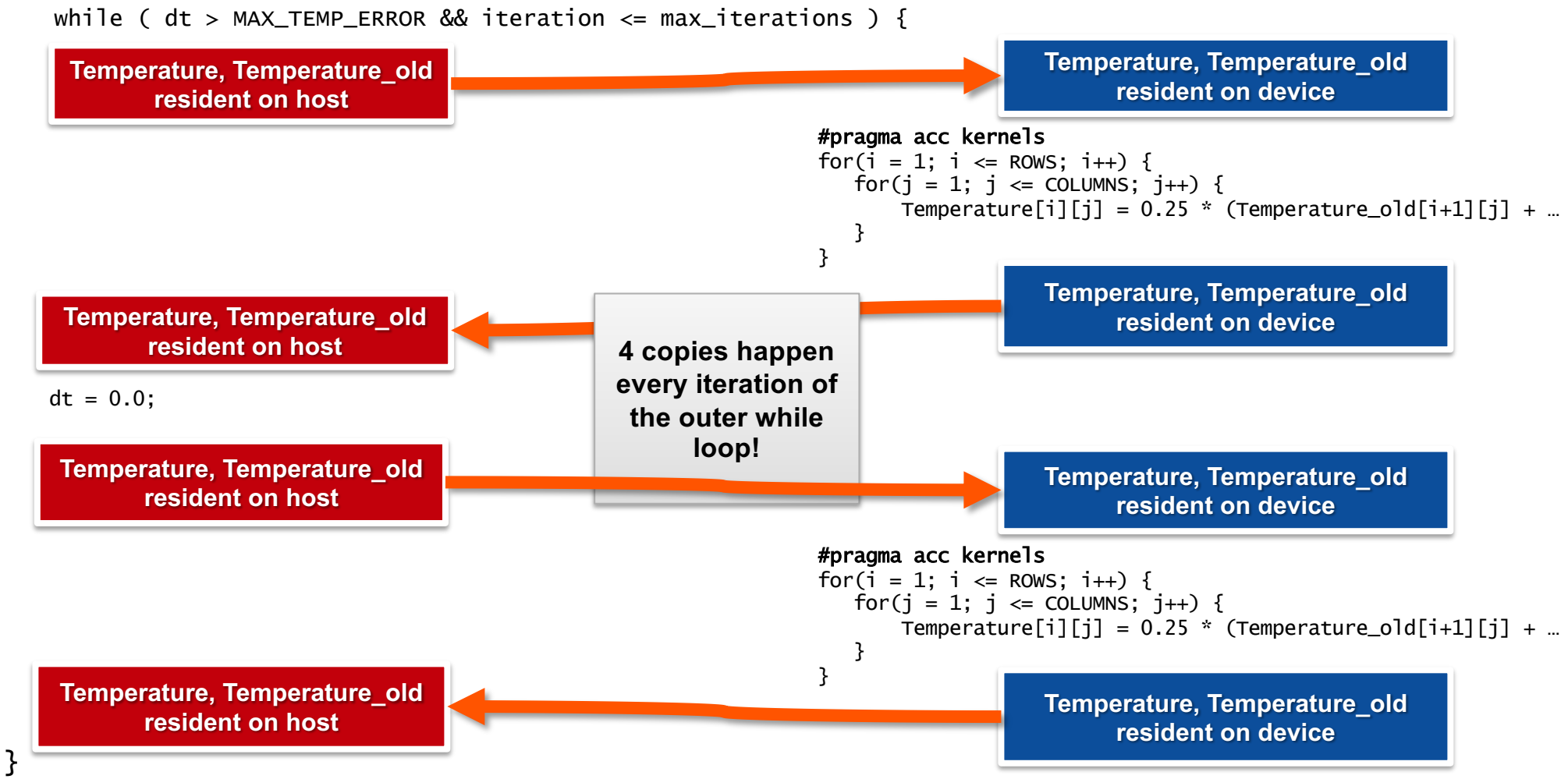
SIMPLIFIED, BUT SADLY TRUE



Excessive Data Transfers



EXCESSIVE DATA TRANSFERS



OpenACC
More Science, Less Programming

DATA MANAGEMENT

THE FIRST, MOST IMPORTANT, AND POSSIBLY ONLY OPENACC OPTIMIZATION

DATA CONSTRUCT SYNTAX AND SCOPE

C

```
#pragma acc data [clause …]
{
    structured block
}
```

Fortran

```
!$acc data [clause …]
    structured block
!$acc end data
```

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

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

Fortran

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

- Fortran uses *start:end* and C uses *start:count*
- Data clauses can be used on `data`, `kernels` or `parallel`

OpenACC
More Science, Less Programming

COMPILER WILL (INCREASINGLY) OFTEN MAKE A GOOD GUESS…

C Code

```
int main(int argc, char *argv[]) {

  int i;
  double A[2000], B[1000], C[

  #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];

  }
}
```

Smarter

Smart

Smartest

Compiler Output

```
pgcc -acc -Minfo=accel loops.c
main:
     6, Generating implicit copy(B[:])
        Generating implicit copyout(C[:])
        Generating implicit copyout(A[:1000])
     7, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
         7, #pragma acc loop gang, vector(128)
```

DATA REGIONS HAVE REAL CONSEQUENCES

Simplest Kernel

```
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;


  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

A[ ] Copied To Host

Runs On Host

Output:
A[10] = 2.0

With Global Data Region

```
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;

}

  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

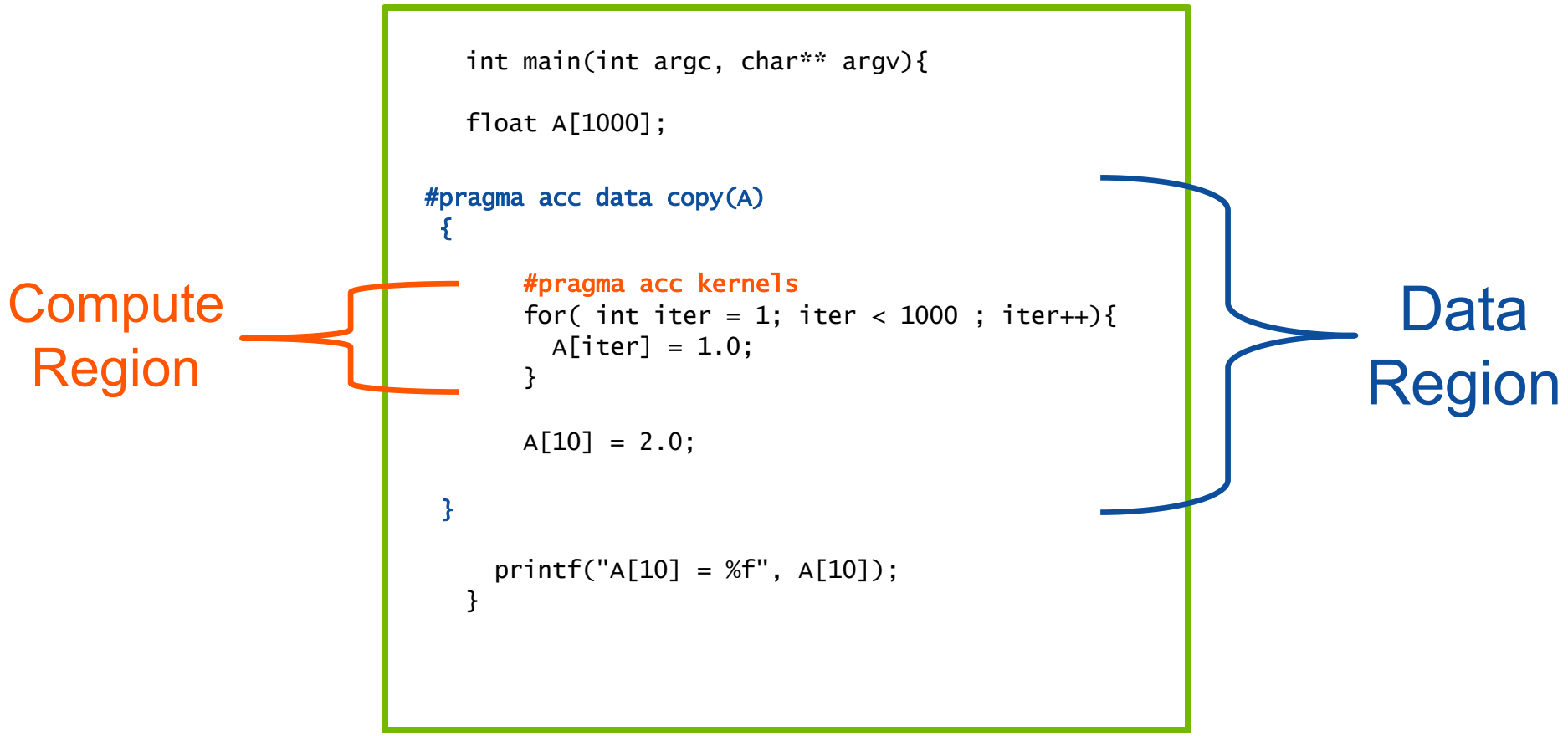
Still Runs On Host

A[ ] Copied To Host

Output:
A[10] = 1.0

OpenACC
More Science, Less Programming

DATA REGIONS VS. *COMPUTE* REGIONS



Output:
A[10] = 1.0

OpenACC
More Science, Less Programming

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:

C

```
#pragma acc update [self(), device(), …]
```

Fortran

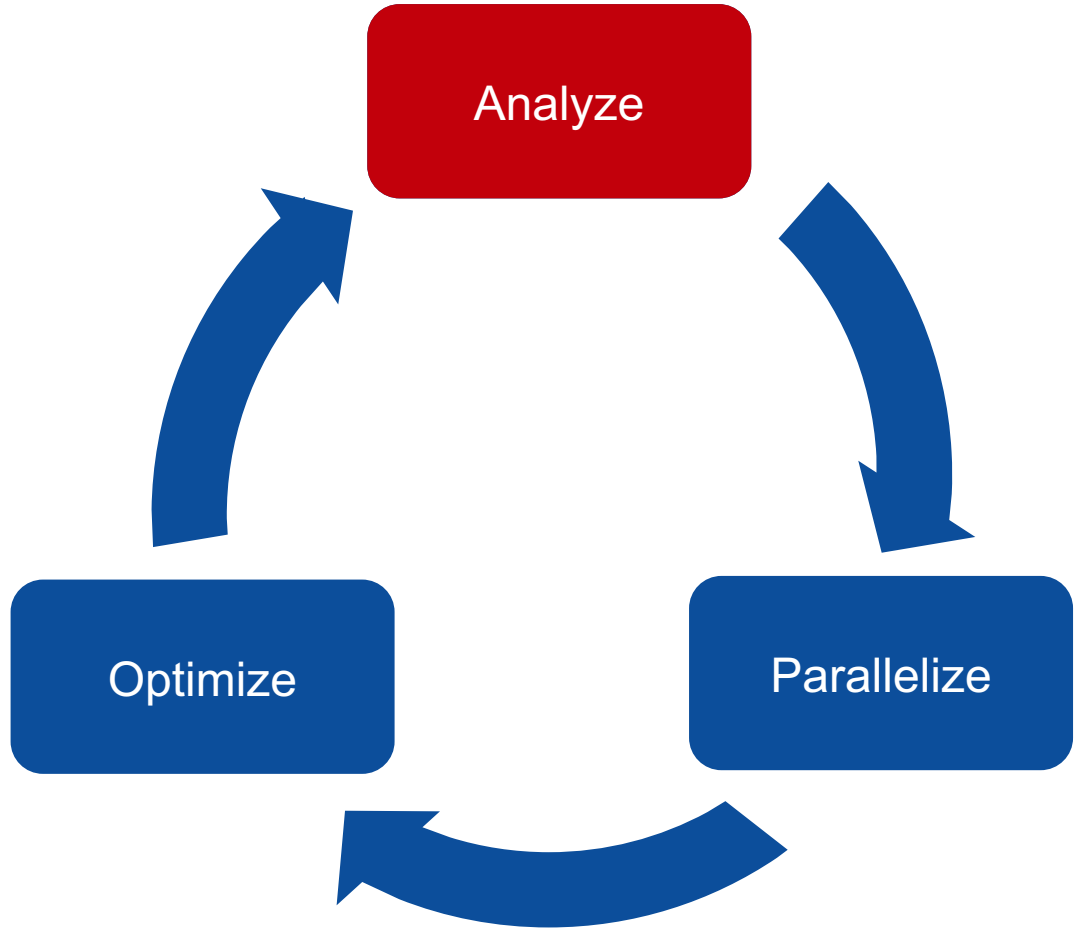
```
!$acc update [self(), device(), …]
```

```
Ex: #pragma acc update self(Temp_array)  //Get host a copy from device
```

ANALYZING / PROFILING

OPENACC DEVELOPMENT CYCLE

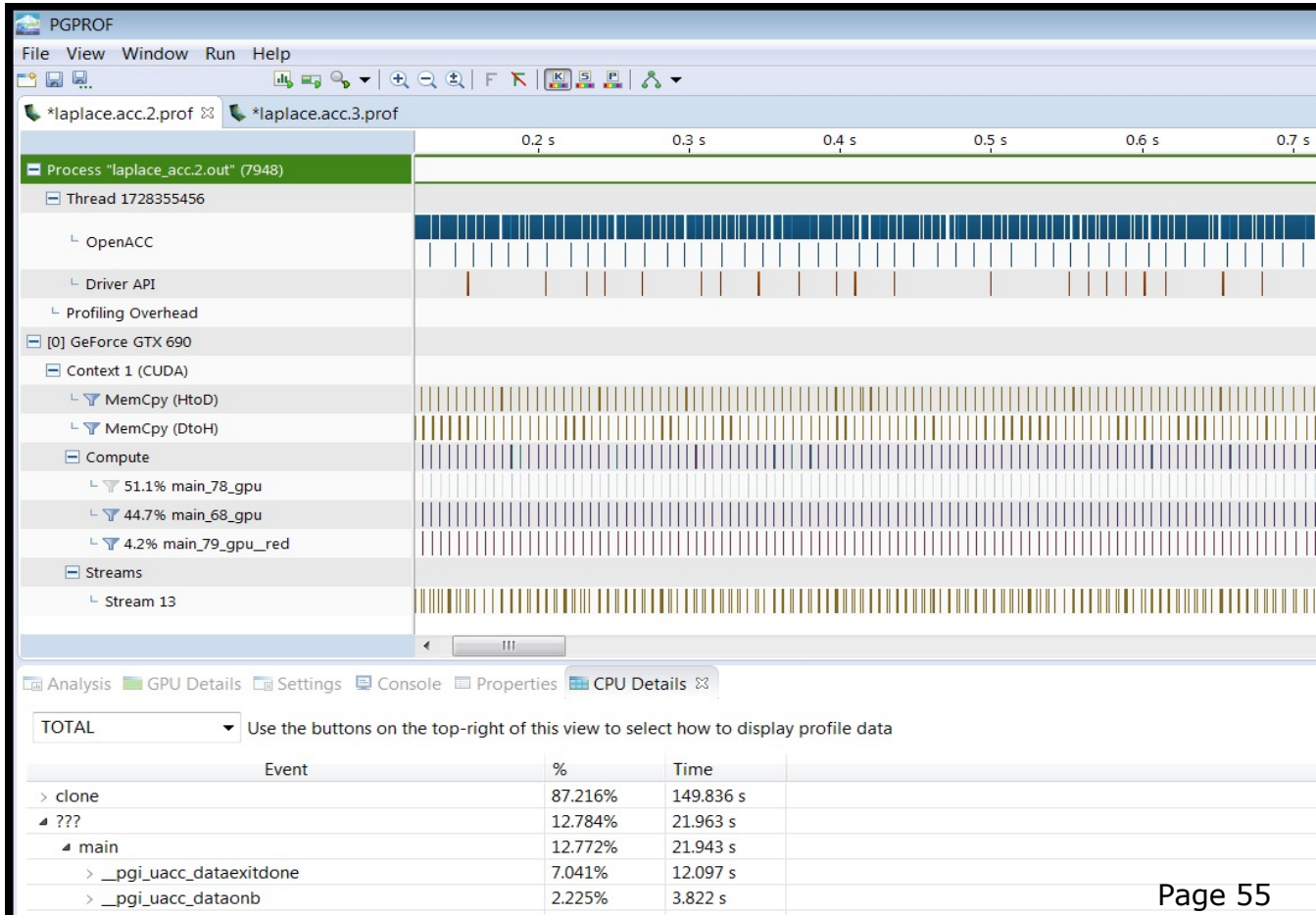
- **Analyze** your code, and predict where potential parallelism can be uncovered. Use profiler to help understand what is happening in the code and were parallelism may exist
- **Parallelize** your code, starting with the most time consuming parts. Focus on maintaining correct results from your program
- **Optimize** your code, focusing on maximizing performance. Performance may not increase all-at-once during early parallelization



PROFILING GPU CODE

Using PGPROF to profile GPU code

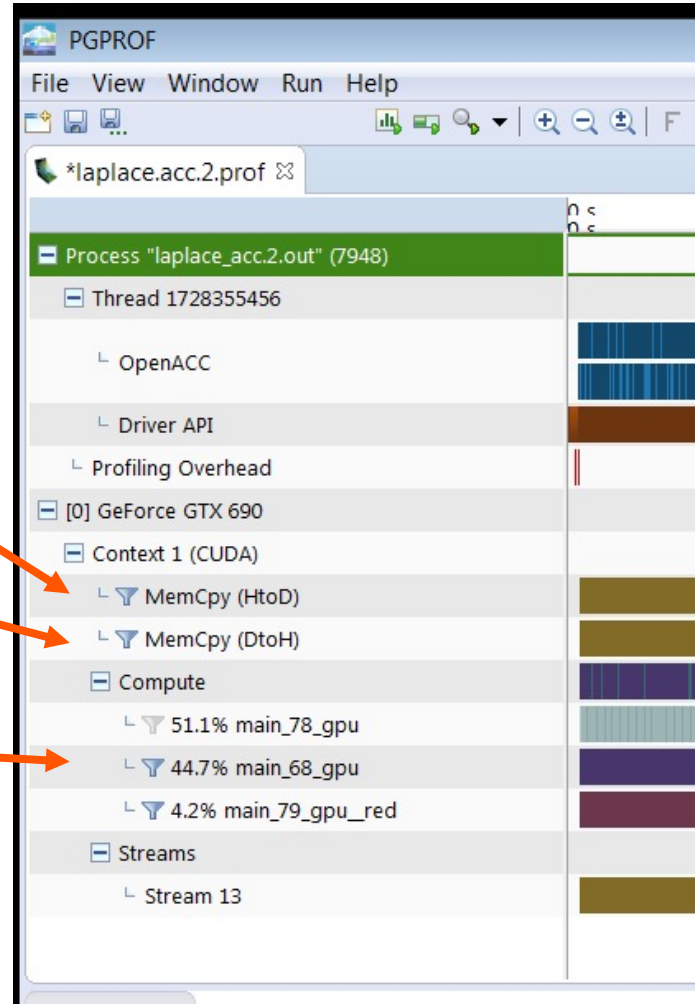
- This is the view of PGPROF after profiling a GPU application
- The first obvious difference when compared to our command line profile: a visual timeline of events



PROFILING GPU CODE

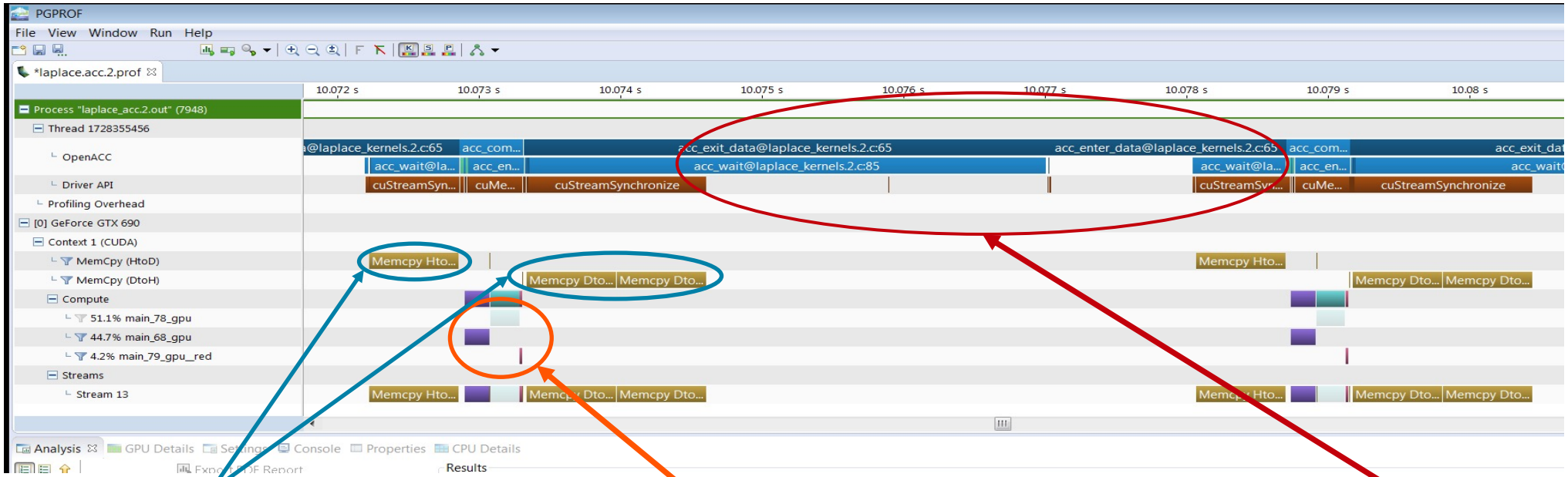
Using PGPROF to profile GPU code

- **MemCpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **MemCpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)
- **Compute:** These are our computational functions.



INSPECTING THE PGPROF TIMELINE

- Zooming in gives us a better view of the timeline. It looks like our program is spending a significant amount of time transferring data between the host and device. We also see that the compute regions are very small, with a lot of distance between them.



Device Memory Transfers

Compute on the Device

CPU overhead time including:
- Device data creation and deletion
- Virtual to Physical Host Memory Copy

OpenACC
More Science, Less Programming

* Note we use a 1 kernel solution (available from the Resources console)

C WITH KERNEL AND DATA DIRECTIVES

```
#pragma acc data create(Temperature_last[0:ROWS+2][0:COLUMNS+2], Temperature[0:ROWS+2][0:COLUMNS+2])
{
    initialize();      // initialize Temperature_last including boundary conditions on device

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        #pragma acc kernels present(Temperature_last, Temperature)
        {
        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
        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];
        }}
        } // end acc kernels
        // periodically print test values
        if((iteration % 100) == 0) {
#pragma acc update self(Temperature[0:ROWS+2][0:COLUMNS+2])
          track_progress(iteration);
        }

      iteration++;
    }
} // end acc data region
```

Allocate space on the device

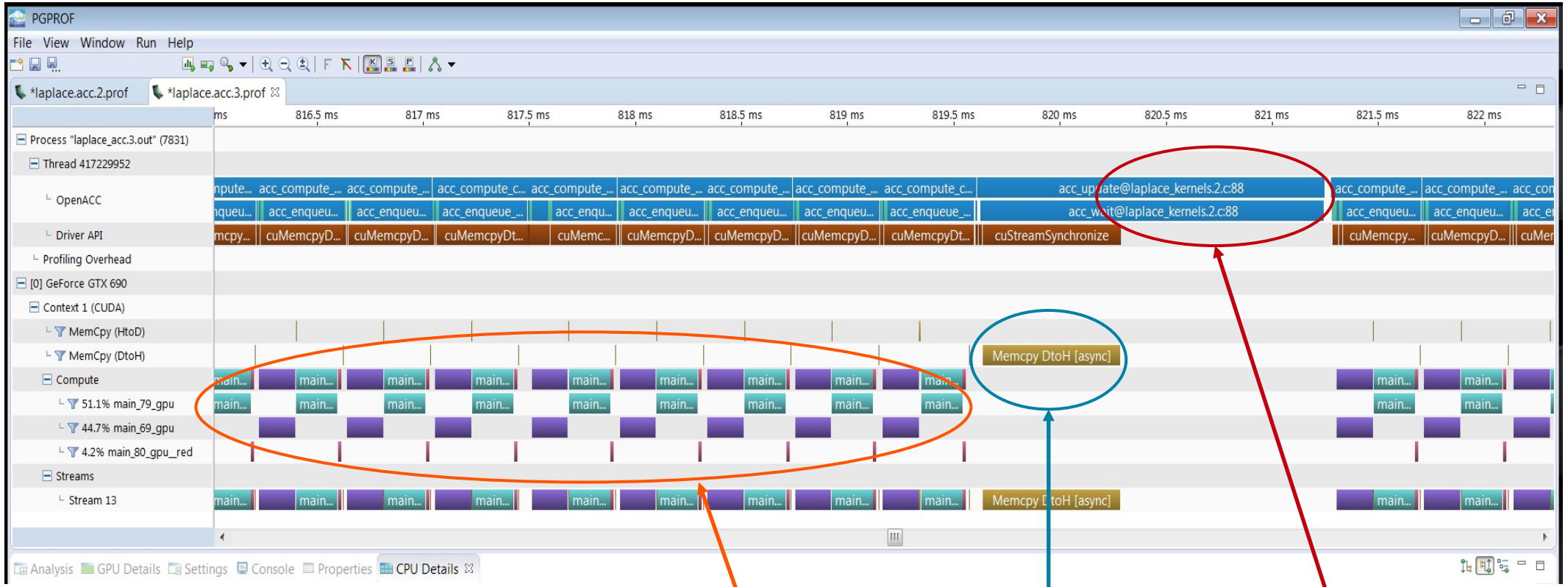
Generate parallel kernels

Periodically synchronize data between host and device

OpenACC
More Science, Less Programming

PROFILE AFTER ADDING A DATA REGION

After adding a data region, we see lots of compute and data movement only when we update.



EXERCISE PERFORMANCE

Same number of steps to convergence

Run	1000x1000		2000x2000	
	Time (s)	Speed-up	Time (s)	Speed-up
Serial CPU	6.2	1.0X	40.3	1.0X
OpenACC 4-core CPU	1.5	4.1X	21	1.9X
OpenACC Tesla	1.6	3.9X	6.6	6.1X

Notice that the performance improvement will vary by the size of the problem. This is why it's important to work with representative problem sizes.

Outline

- Introduction to OpenACC
  - What is OpenACC?
  - OpenACC Kernel Directives
  - Data Dependencies
- Programming with OpenACC
  - Case Study: Laplace Solver
  - Data Management
  - Analyzing/Profiling
- Optimizing and Best Practices for OpenACC
  - OpenACC Parallel Directives
  - OpenACC Loop Directives
  - Managed Memory
  - Unstructed Data Directives
  - C/C++ Struct/Classes
  - OpenACC Routine Directives
- References

OPENACC PARALLEL DIRECTIVE

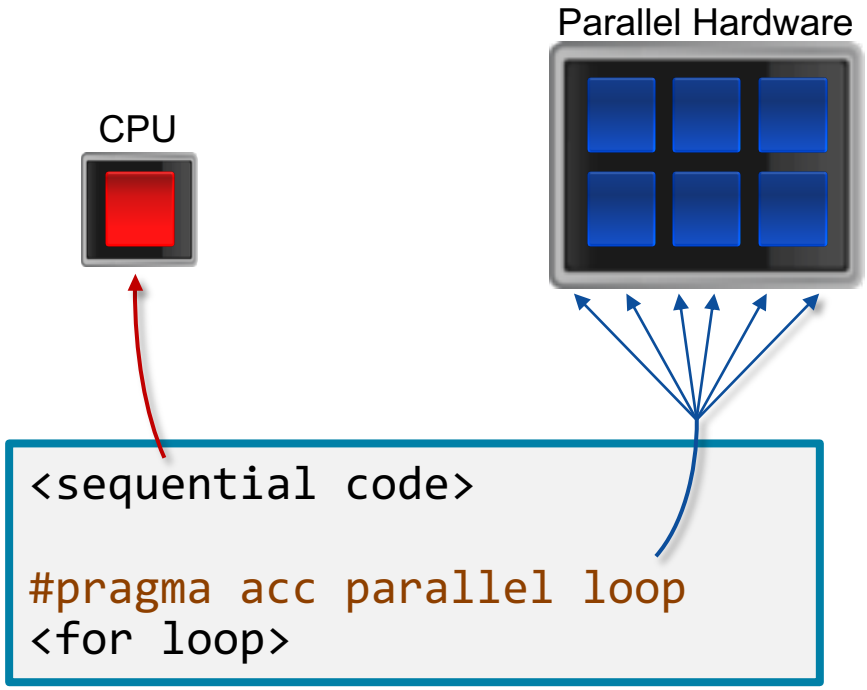
THE PARALLEL LOOP DIRECTIVE

Take Control of Your Parallelization

- So far we've concentrated on the KERNELS directive.
- The KERNELS directive tells the compiler I *may* want the loops in a section of code parallelized, but leaves the final decision to the compiler.
- But what if I *know* I want the loop to be parallelized? Can I take more control?

OPENACC PARALLEL DIRECTIVE

Programmer-driven Parallelism



- The parallel directive instructs the compiler to create *parallel gangs* on the accelerator
- The loop directive asserts to the compiler that a loop is safe to parallelize. Without it the code runs redundantly in each gang.
- Compiler will use this assertion to parallelize anything the programmer tells it to, regardless of whether or not it will break the program

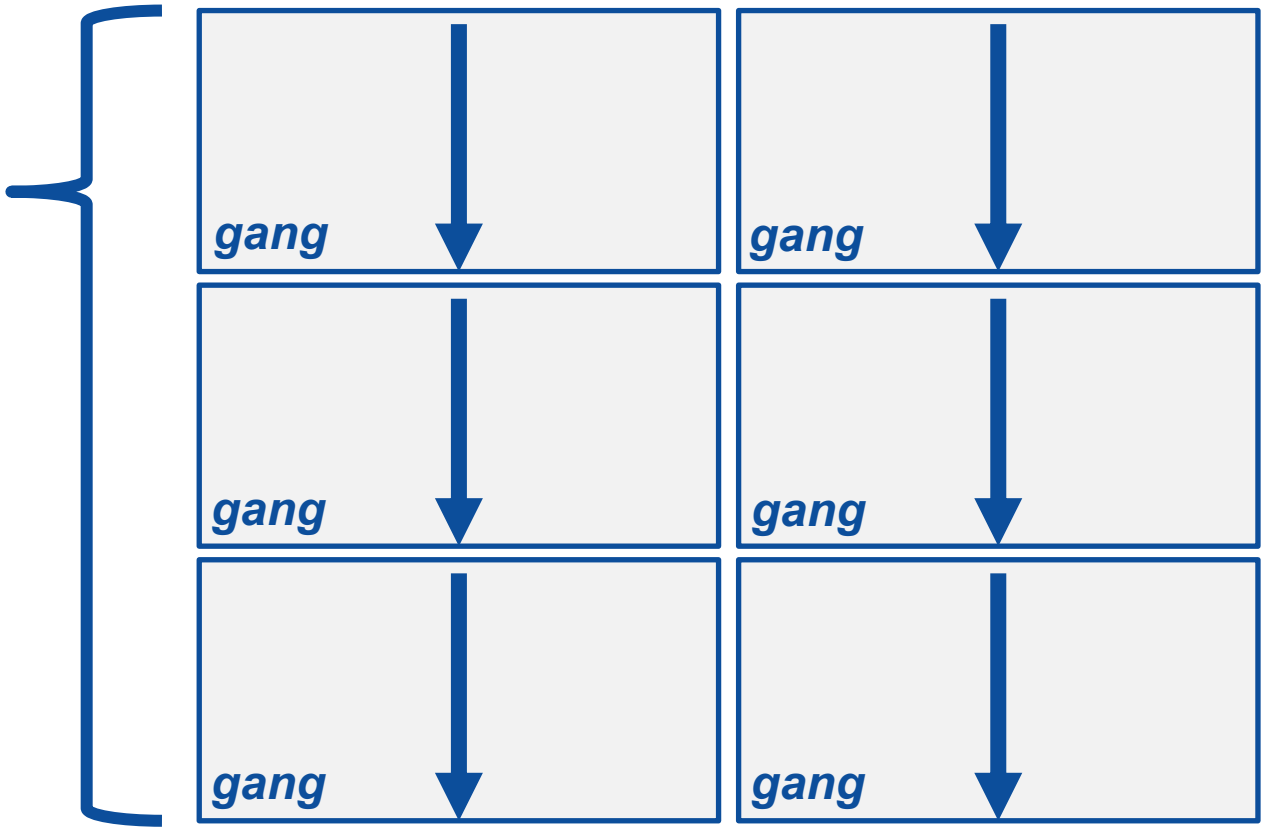
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{
```

When encountering the ***parallel*** directive, the compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.

```
}
```



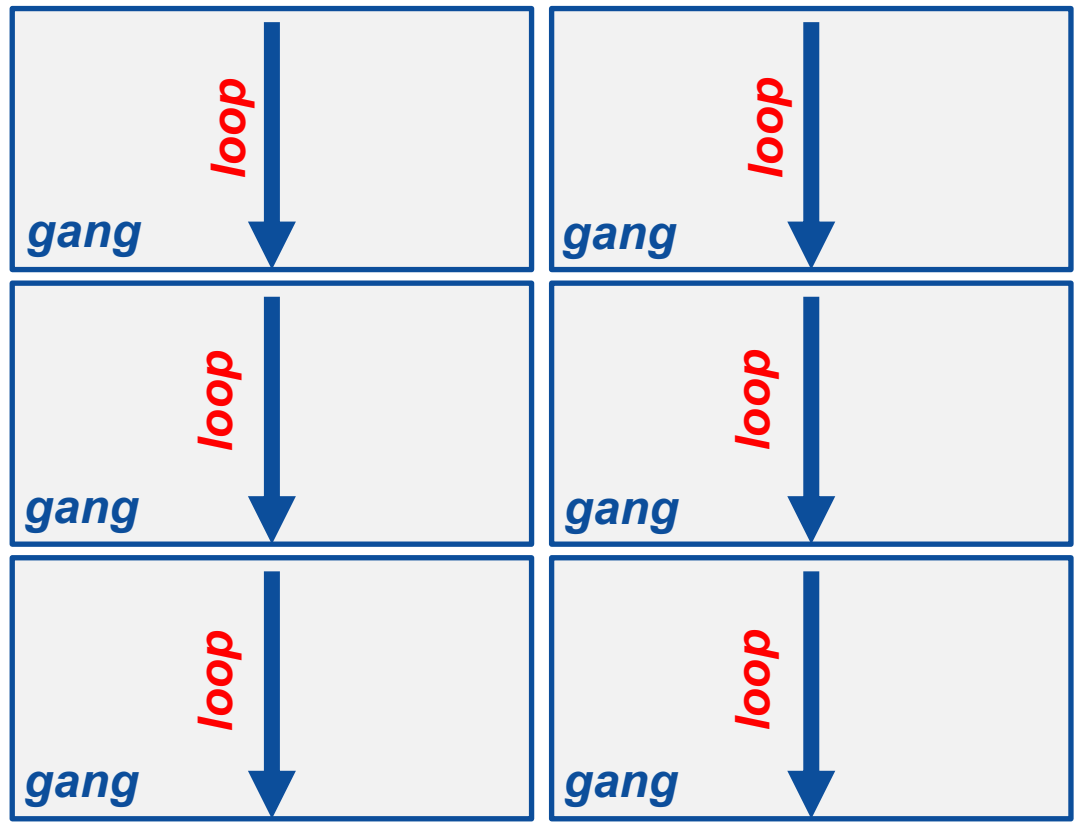
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

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

loop

This loop will be redundantly parallelized with each ***gang*** executing the entire loop



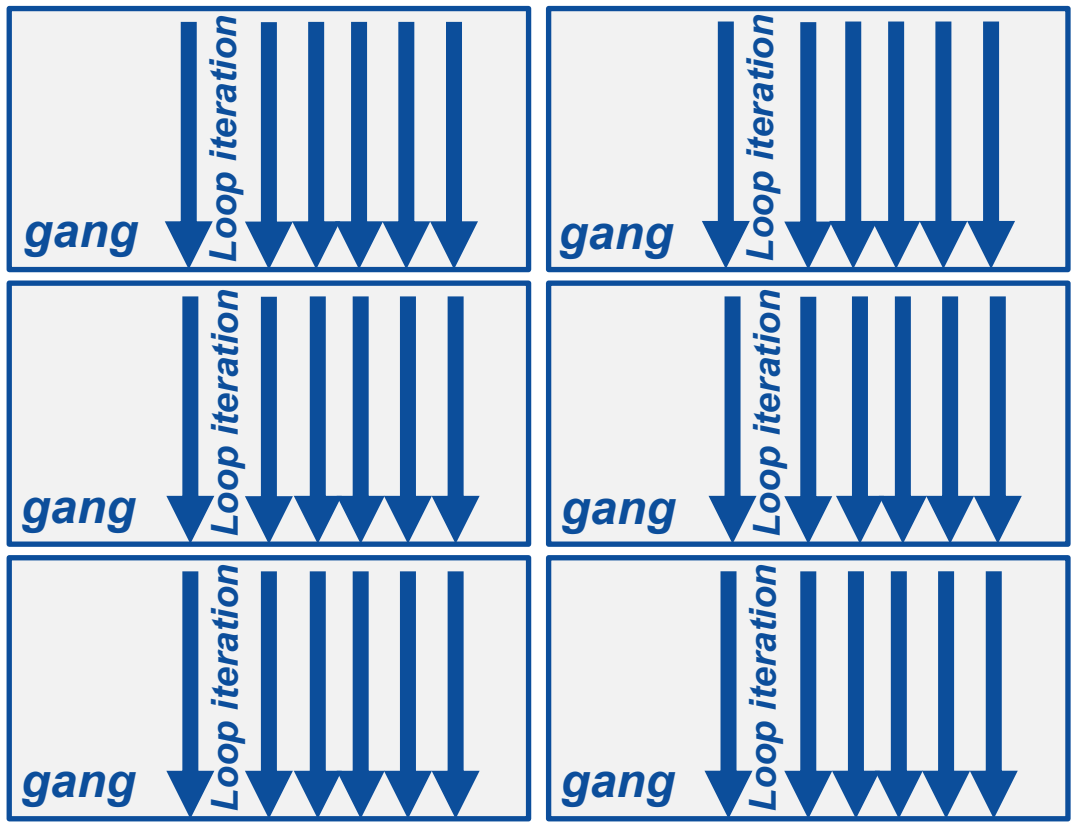
OpenACC
More Science, Less Programming

OPENACC PARALLEL DIRECTIVE

Expressing parallelism

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

}
```

Now the iterations of the loop will be distributed to the gangs and run in parallel.



OpenACC
More Science, Less Programming

OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

C/C++

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

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
- Then mark the loop(s) you want to run in parallel with the loop directive.

OpenACC
More Science, Less Programming

OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

C/C++

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

Fortran

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

- 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

OPENACC PARALLEL DIRECTIVE

Parallelizing many loops

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

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

- To parallelize multiple loops, each loop should be accompanied by a parallel loop 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

KERNELS VS PARALLEL

Kernels

- Compiler decides what to parallelize with direction from user
- Compiler guarantees correctness
- Can cover multiple loop nests

Parallel

- Programmer decides what to parallelize and communicates that to the compiler
- Programmer guarantees correctness
- Must decorate each loop nest

When fully optimized, both will give similar performance.

LOOP REDUCTIONS

Reduce Many Results to One

```
#pragma acc parallel loop collapse(2) \
            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];
  }
}
```

- Some loops are only safe to parallelize with additional care
- What about the MAX operation in this code?
- In order to obtain correct results, the compiler must reduce all values of dt to only the maximum
- This is enabled with the reduction clause

OpenACC
More Science, Less Programming

OPENACC LOOP DIRECTIVE

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++

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

Fortran

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

OPENACC LOOP DIRECTIVE

Inside of a parallel compute region

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

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

- In this example, the first loop is not marked with the loop directive
- This means that the loop will be “redundantly parallelized”
- Redundant parallelization, in this case, means that the loop will be run in its entirety, multiple times, by the parallel hardware
- The second loop is marked with the loop directive, meaning that the loop iterations will be properly split across the parallel hardware

OPENACC LOOP DIRECTIVE

Inside of a kernels compute region

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

  #pragma acc loop
  for(int j = 0; j < M; j++)
    b[i] = 0;
}
```

- With the kernels directive, the loop directive is implied
- The programmer can still explicitly define loops with the loop directive, however this could affect the optimizations the compiler makes
- The loop directive is not needed, but does allow the programmer to optimize the loops themselves

OPENACC LOOP DIRECTIVE

Parallelizing loop nests

C/C++

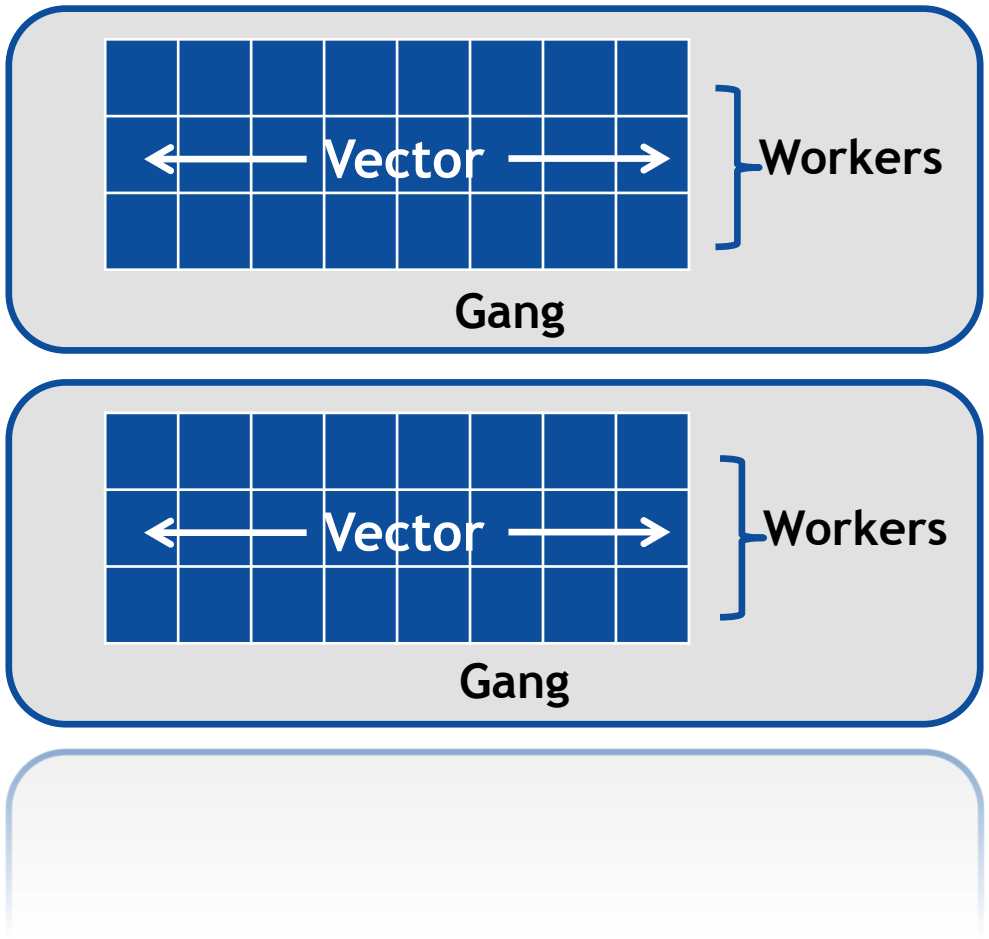
```
#pragma acc parallel loop
for(int i = 0; i < N; i++){
  #pragma acc loop
  for(int j = 0; j < M; j++){
    a[i][j] = 0;
  }
}
```

Fortran

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

- You can include multiple loop directives to parallelize multi-dimensional loop nests
- On some parallel hardware, this will allow you to express more parallelism to increase performance
- When the additional parallelism isn't needed the compiler will choose what it believes is best and ignore the other loop directives

OPENACC: 3 LEVELS OF PARALLELISM



- *Vector* threads work in lockstep (SIMD/SIMT parallelism)
- *Workers* compute a vector
- *Gangs* have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other

GANG, WORKER, VECTOR CLAUSES

gang, *worker*, and *vector* can be added to a loop clause

A parallel region can only specify one of each gang, worker, vector

Control the size using the following clauses on the parallel region

num_gangs(n), num_workers(n), vector_length(n)

```
#pragma acc parallel loop gang
 for (int i = 0; i < n; ++i)
 #pragma acc loop vector
  for (int j = 0; j < n; ++j)
   ...
```

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

GANG, WORKER, VECTOR CLAUSES

gang, *worker*, and *vector* can be added to a kernels loop clause too

Since different loops in a kernels region may be parallelized differently, fine-tuning is done as a parameter to the gang, worker, and vector clauses.

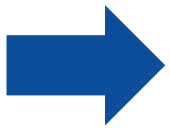
```
#pragma acc kernels loop gang
 for (int i = 0; i < n; ++i)
  #pragma acc loop vector
   for (int j = 0; j < n; ++j)
    ...
```

```
#pragma acc kernels loop gang worker
for (int i = 0; i < n; ++i)
  #pragma acc loop vector(32)
  for (int j = 0; j < n; ++j)
    ...
```

THE COLLAPSE CLAUSE

collapse(n): Takes the next *n* tightly-nested loops, folds them into one, and applies the OpenACC directives to the new loop.

```
#pragma acc parallel loop \
  collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<M; j++)
    ...
```



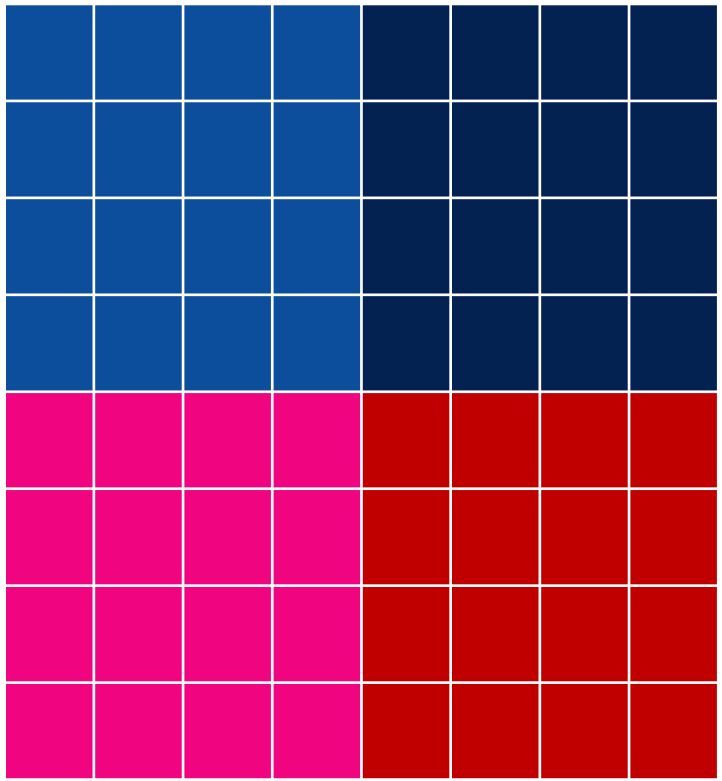
```
#pragma acc parallel loop
for(int ij=0; ij<N*M; ij++)
    ...
```

Why?

- Collapse outer loops to enable creating more gangs.
- Collapse inner loops to enable longer vector lengths.
- Collapse all loops, when possible, to do both.

THE TILE CLAUSE

Operate on smaller blocks of the operation to exploit data locality



```
#pragma acc loop tile(4,4)
  for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
      Temp[i][j] = 0.25 *
       (Temp_last[i+1][j] +
        Temp_last[i-1][j] +
        Temp_last[i][j+1] +
        Temp_last[i][j-1]);
    }
  }
```

OpenACC
More Science, Less Programming

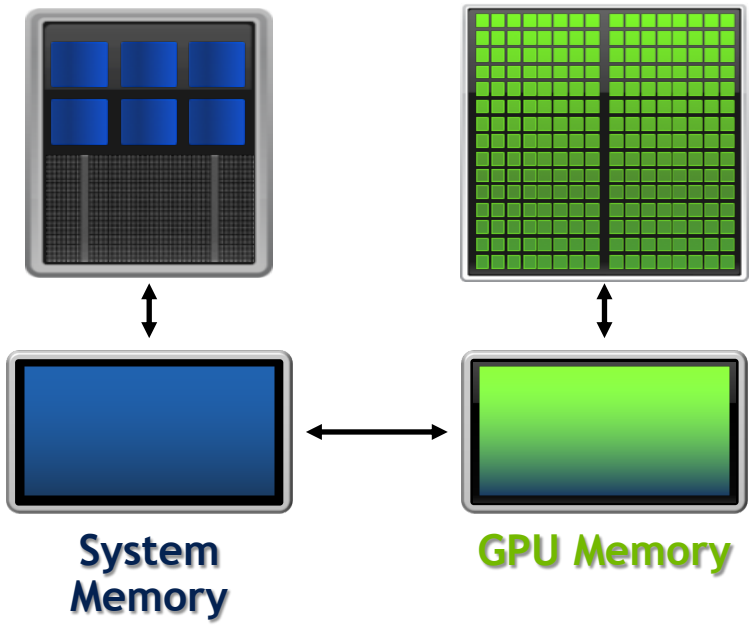
MANAGED MEMORY

CUDA MANAGED MEMORY

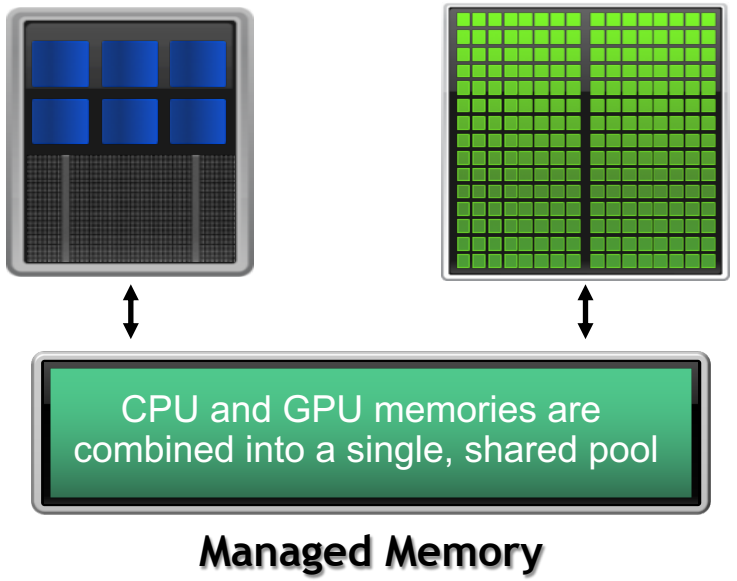
Simplified Developer Effort

Commonly referred to as “unified memory.”

Without Managed Memory



With Managed Memory

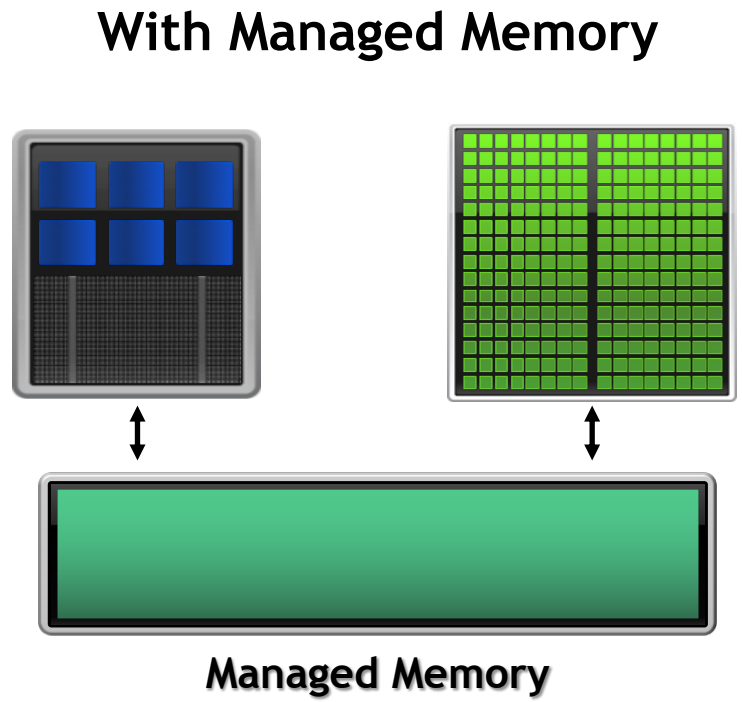


OpenACC
More Science, Less Programming

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
- Cannot be used with static memory*
- Performance depends on how the data is accessed



MANAGED MEMORY

Why and How to Use It

- Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
- PGI provides the managed target option for NVIDIA Tesla GPUs
- This will tell the compiler allocate all memory as CUDA Managed Memory
- This generally means that the programmer will do less work, but the code is less portable.

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

PROS & CONS OF MANAGED MEMORY

Pro

- Simple porting of complex data structures
- Concentrate on parallelism first and data later

Con

- Limited to the PGI compiler and NVIDIA GPUs, no portability
- Performance will depend heavily on access pattern

Using managed memory should be a stepping stone to quickly port the code.

UNSTRUCTURED DATA DIRECTIVES

UNSTRUCTURED DATA DIRECTIVES

Enter Data Directive

- The **enter data** directive handles device memory **allocation**
- You may use either the **create** or the **copyin** clause for memory allocation
- You may allocate more than one array at a time, and you may allocate arrays in any function
- The enter data directive is **not** the start of a data region, because you may have multiple enter data directives

```
#pragma acc enter data clauses

  < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

```
!$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 may use the exit data directive to deallocate any array that was previously allocated with the enter data directive
- One of the biggest advantages of using unstructured data directives is the ability to do device memory allocation and deallocation in **completely different functions**

```
#pragma acc enter data clauses

  < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

```
!$acc enter data clauses

  < Sequential and/or Parallel code >

!$acc exit data clauses
```

OpenACC
More Science, Less Programming

UNSTRUCTURED DATA DIRECTIVES

Basic Example

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

UNSTRUCTURED DATA DIRECTIVES

Basic Example

```
#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])
```

UNSTRUCTURED DATA DIRECTIVES

Basic Example

```
#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])
```

Action
Copy A to GPU Copy B to GPU Create C on GPU Delete A from GPU Delete B from GPU Copy C to CPU Delete C from GPU

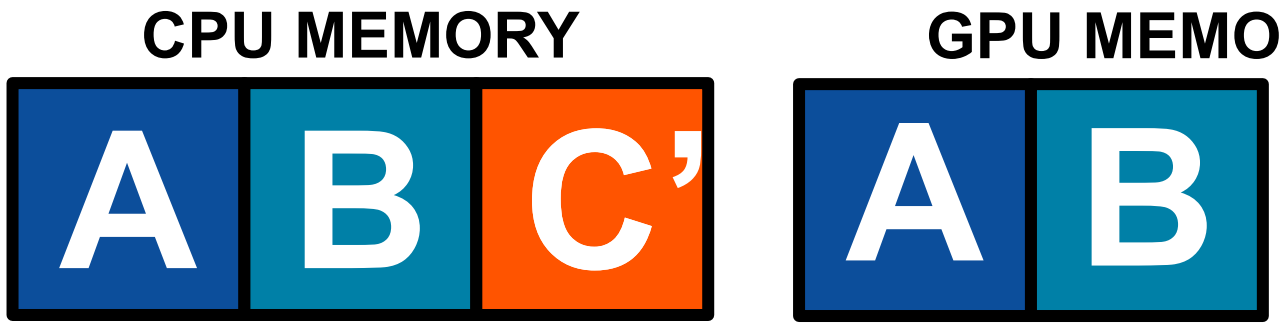
OpenACC
More Science, Less Programming

UNSTRUCTURED DATA DIRECTIVES

Basic Example – proper memory deallocation

```
#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)
```

Action
Deallocate A, B from GPU



OpenACC
More Science, Less Programming

UNSTRUCTURED VS STRUCTURED

With a simple code

Unstructured

- Can have multiple starting/ending points
- Memory exists until explicitly deallocated
- Can branch across multiple functions

```
#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

- Must have a explicit start/end point
- Memory only exists within the data region
- Must be within a single function

```
#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];
  }

}
```

UNSTRUCTURED DATA DIRECTIVES

Branching across multiple functions

```
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);
}
```

- This is an example code where the memory allocation/deallocation is handled in separate functions
- The data region is not explicitly defined by a starting point and an ending point
- The data enter and exit will be decided by whenever the programmer calls the allocate/deallocate functions

OPENACC REFERENCE COUNTING

- The OpenACC runtime keeps track of how many times a variable is placed on or removed from the device using reference counting
- Each time you create, copyin, or copy a variable to the device, the reference counts increase. Each time you delete, copyout, or copy a variable from the device, the reference counts decrease.
- Data movement between the host and device only occurs when a variable is first put on the device and the last time it leaves the device.

REFERENCE COUNTING EXAMPLES

```
// First time A is reference on the device,
// copy occurs
#pragma acc enter data copyin(A[0:N]

// Since it A is already on the device, no
// copy will occur at the beginning or end
#pragma acc data copy(A[0:N])
{
  ...
}

// This is the last time A is removed from
// the device, copy occurs
#pragma acc exit data copyout(A[0:N])
```

```
// First time A is reference on the device,
// copy occurs
#pragma acc enter data copyin(A[0:N])

// Since it A is already on the device, no
// copy will occur here
#pragma acc enter data copyin(A[0:N])

  ...

// Since we've "entered" A twice and this
// is the first exit, data will not be copied
#pragma acc exit data copyout(A[0:N])

// This is the last time A is removed from
// the device, copy occurs
#pragma acc exit data copyout(A[0:N])
```

C/C++ STRUCTS/CLASSES

C STRUCTS

Without dynamic data members

- Dynamic data members are anything contained within a struct that can have a **variable size** (dynamically allocated arrays)
- OpenACC is easily able to copy our struct to device memory because everything in our float3 struct has a **fixed size**
- If float3 has any members with a varying size, then the programmer will need to explicitly allocate that member in device memory

```
typedef struct {
  float x, y, z;
} float3;

int main(int argc, char* argv[]){
  int N = 10;
  float3* f3 = malloc(N * sizeof(float3));

  #pragma acc enter data create(f3[0:N])

  #pragma acc kernels
  for(int i = 0; i < N; i++){
    f3[i].x = 0.0f;
    f3[i].y = 0.0f;
    f3[i].z = 0.0f;
  }

  #pragma acc exit data delete(f3)
  free(f3);
}
```

C STRUCTS

With dynamic data members

- OpenACC is not automatically able to copy dynamic pointers to the device
- You must first copy the struct into device memory
- Then you must allocate/copy the dynamic members into device memory
- To deallocate, you must first deallocate the dynamic members
- Then deallocate the struct

```
typedef struct {
  float *arr;
  int n;
} vector;

int main(int argc, char* argv[]){

  vector v;
  v.n = 10;
  v.arr = (float*) malloc(v.n*sizeof(float));

  #pragma acc enter data copyin(v)
  #pragma acc enter data create(v.arr[0:v.n])

  ...

  #pragma acc exit data delete(v.arr)
  #pragma acc exit data delete(v)
  free(v.arr);
}
```

C++ STRUCTS/CLASSES

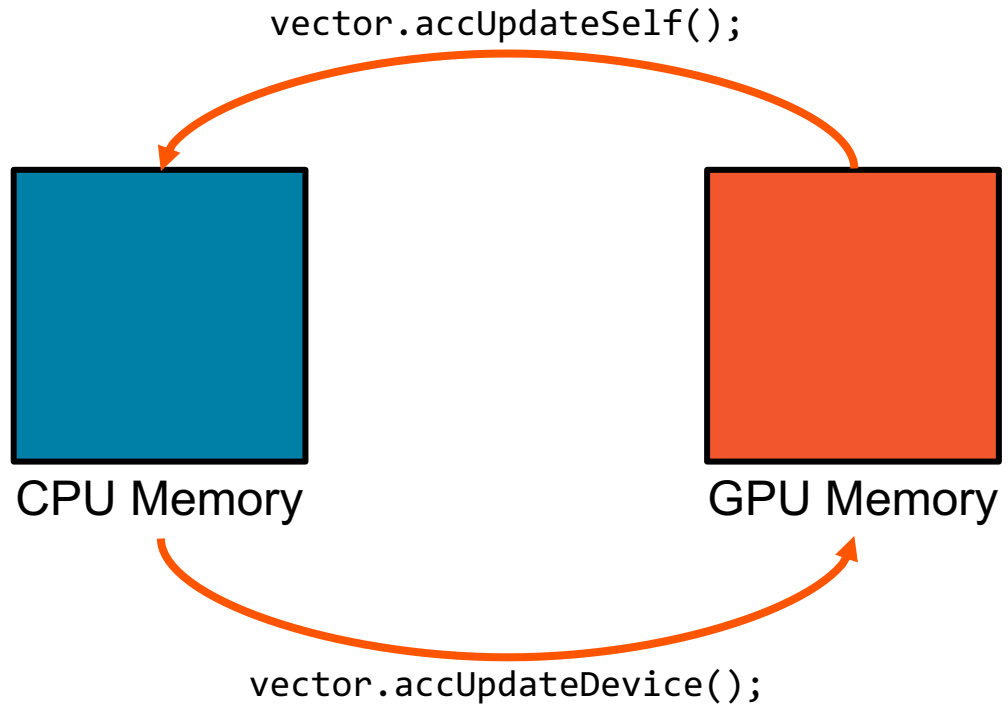
With dynamic data members

- 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

```
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)
      #pragma acc exit data delete(this)
      delete(arr);
    }
};
```

C++ CLASS DATA SYNCHRONIZATION

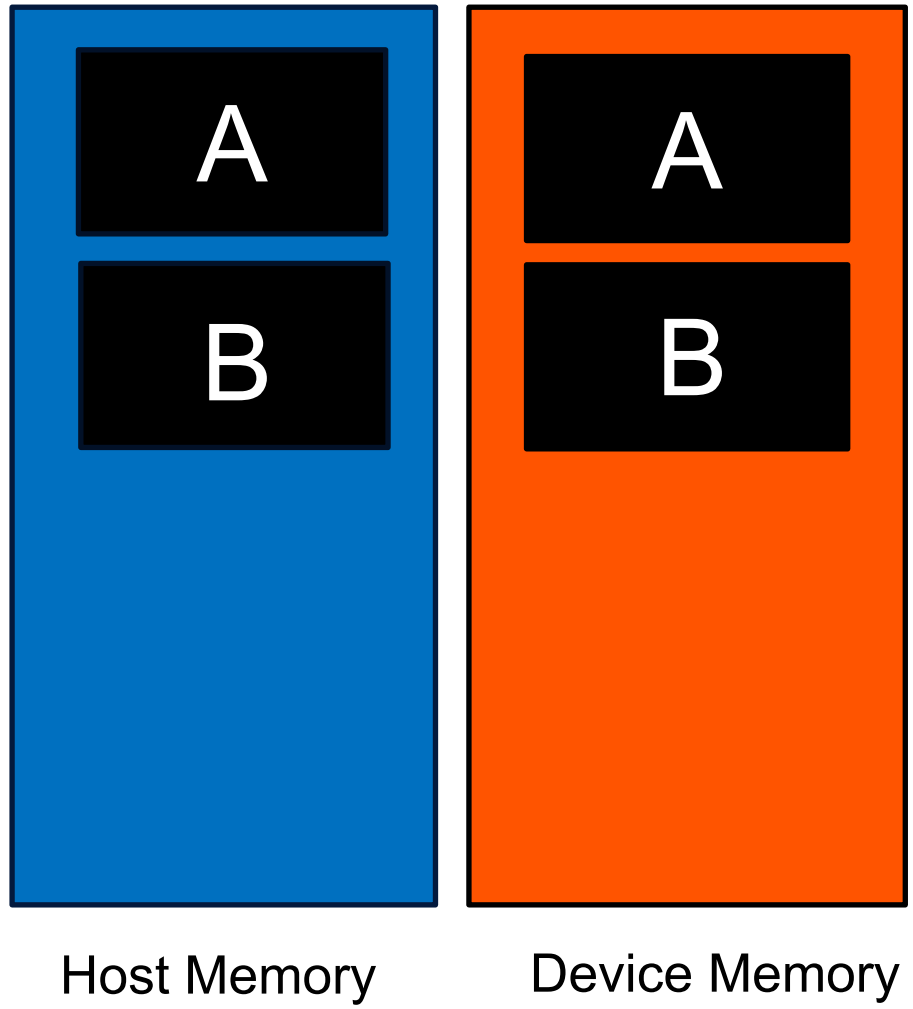
- Since data is encapsulated, the class needs to be extended to include data synchronization methods
- Including explicit methods for host/device synchronization may ease C++ data management
- Allows the class to be able to naturally handle synchronization, creating less code clutter




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

USING AN OPENACC AWARE C++ CLASS

```
#include “vector.h"

int main() {

    vector A(N), B(N);
    for (int i=0; i < B.size(); ++i) {
        B[i]=2.5;
    }
    B.accUpdateDevice();
    #pragma acc parallel loop present(A,B)
    for (int i=0; i < A.size(); ++i) {
           A[i]=B[i]+i;
    }
    A.accUpdateSelf();
    for(int i=0; i<10; ++i) {
        cout << "A[" << i << "]: " << A[i] << endl;
    }
    exit(0);
}
```



OPENACC ROUTINE DIRECTIVE

OPENACC ROUTINE DIRECTIVE

Specifies that the compiler should generate a device copy of the function/subroutine and what type of parallelism the routine contains.

Clauses:

`gang/worker/vector/seq`
Specifies the level of parallelism contained in the routine.

`bind`
Specifies an optional name for the routine, also supplied at call-site

`no_host`
The routine will only be used on the device

`device_type`
Specialize this routine for a particular device type.

You *must* declare *one* level of parallelism on the routine directive.

ROUTINE DIRECTIVE: C/C++

```
// foo.h
#pragma acc routine seq
double foo(int i);
```

```
// Used in main()
#pragma acc parallel loop
for(int i=0;i<N;i++) {
  array[i] = foo(i);
}
```

- At function source:
  - Function needs to be built for the GPU.
  - It will be called by each thread (sequentially)
- At call the compiler needs to know:
  - Function will be available on the GPU
  - It is a sequential routine

Outline

- Introduction to OpenACC
  - What is OpenACC?
  - OpenACC Kernel Directives
  - Data Dependencies
- Programming with OpenACC
  - Case Study: Laplace Solver
  - Data Management
  - Analyzing/Profiling
- Optimizing and Best Practices for OpenACC
  - OpenACC Parallel Directives
  - OpenACC Loop Directives
  - Managed Memory
  - Unstructed Data Directives
  - C/C++ Struct/Classes
  - OpenACC Routine Directives
- References

ADVANCED TOPICS

Where to find more help

- Asynchronous OpenACC:

https://developer.nvidia.com/openacc-overview-course Class #4

- Using OpenACC with MPI:

https://developer.nvidia.com/openacc-advanced-course Class #2

- Multi-GPU OpenACC:

http://on-demand-gtc.gputechconf.com/gtc-quicklink/hhZdn

OPENACC RESOURCES

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

FREE Compilers

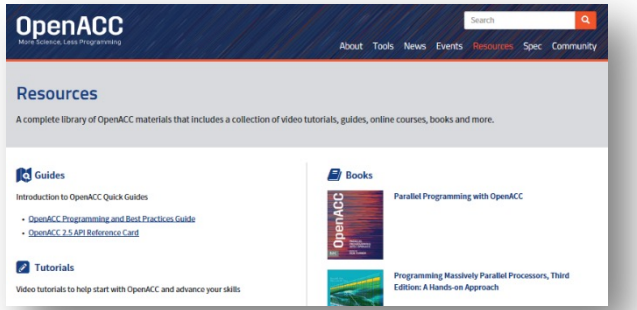
PGI® Community EDITION

slack

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

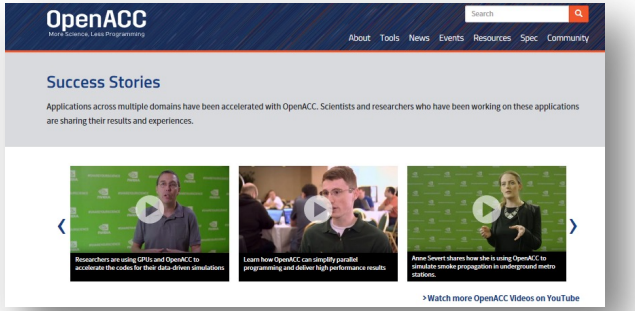
Resources

https://www.openacc.org/resources



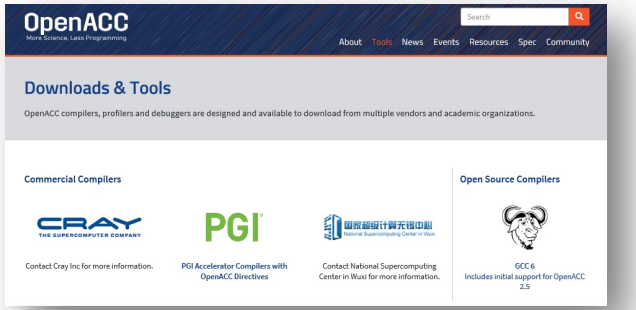
Success Stories

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



Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events