

ZERO TO GPU HERO WITH OPENACC

Jiqun Tu, HPC Developer Technology, NVIDIA

OUTLINE

Topics to be covered

- What is OpenACC
- Profile-driven Development
- OpenACC with CUDA Unified Memory
- OpenACC Data Directives
- OpenACC Loop Optimizations
- Where to Get Help

INTRODUCTION TO OPENACC

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

Add Simple Compiler Directive

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



3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

Easy to use
Most Performance

Compiler Directives

Easy to use
Portable code

OpenACC

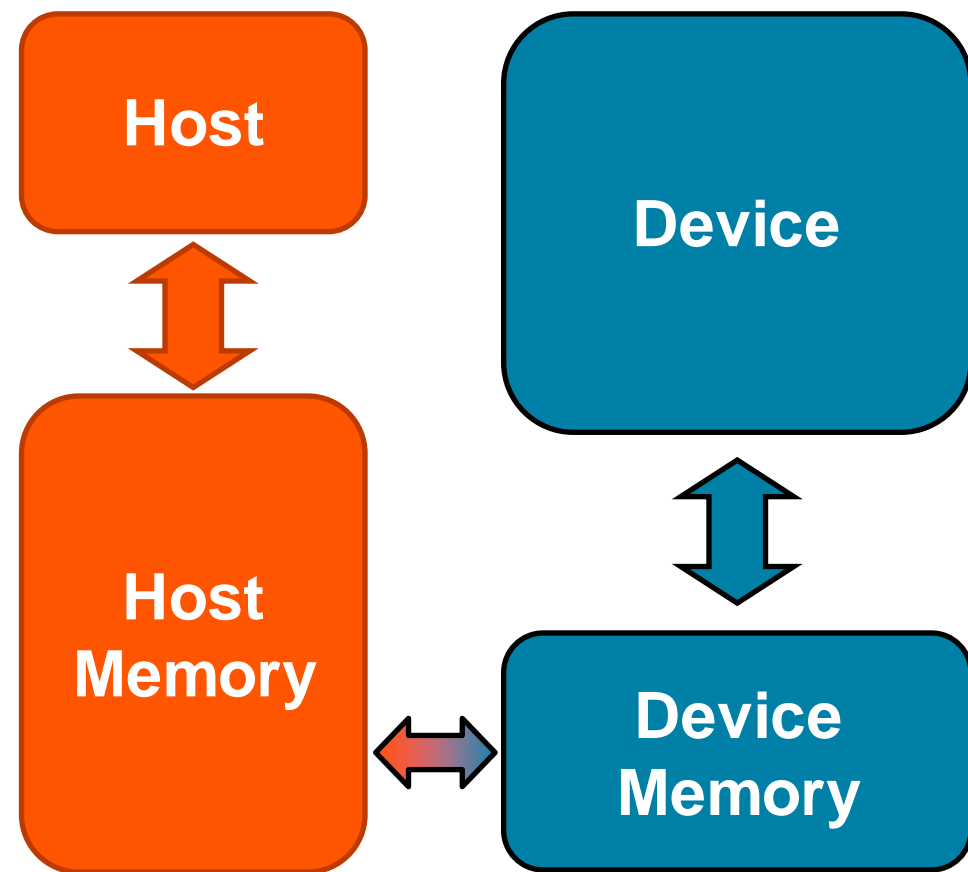
Programming Languages

Most Performance
Most Flexibility

OPENACC PORTABILITY

Describing a generic parallel machine

- OpenACC is designed to be portable to many existing and future parallel platforms
- The programmer need not think about specific hardware details, but rather express the parallelism in generic terms
- An OpenACC program runs on a *host* (typically a CPU) that manages one or more parallel *devices* (GPUs, etc.). The host and device(s) are logically thought of as having separate memories.



OPENACC

Three major strengths

Incremental

Single Source

Low Learning Curve

OPENACC

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Enhance Sequential Code

```
#pragma acc parallel loop  
for( i = 0; i < N; i++ )  
{  
    < loop code >  
}  
  
#pragma acc parallel loop  
for( i = 0; i < N; i++ )  
{  
    < loop code >  
}
```

Begin with a working sequential code.

Parallelize it with OpenACC.

Rerun the code to verify correct behavior, remove/alter OpenACC code as needed.

OPENACC

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source

Low Learning Curve

OPENACC

Supported Platforms

POWER

Sunway

x86 CPU

x86 Xeon Phi

NVIDIA GPU

PEZY-SC

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

```
int main(){  
  
...  
  
#pragma acc parallel loop  
for(int i = 0; i < N; i++)  
    < loop code >  
  
}
```

OPENACC

Incremental

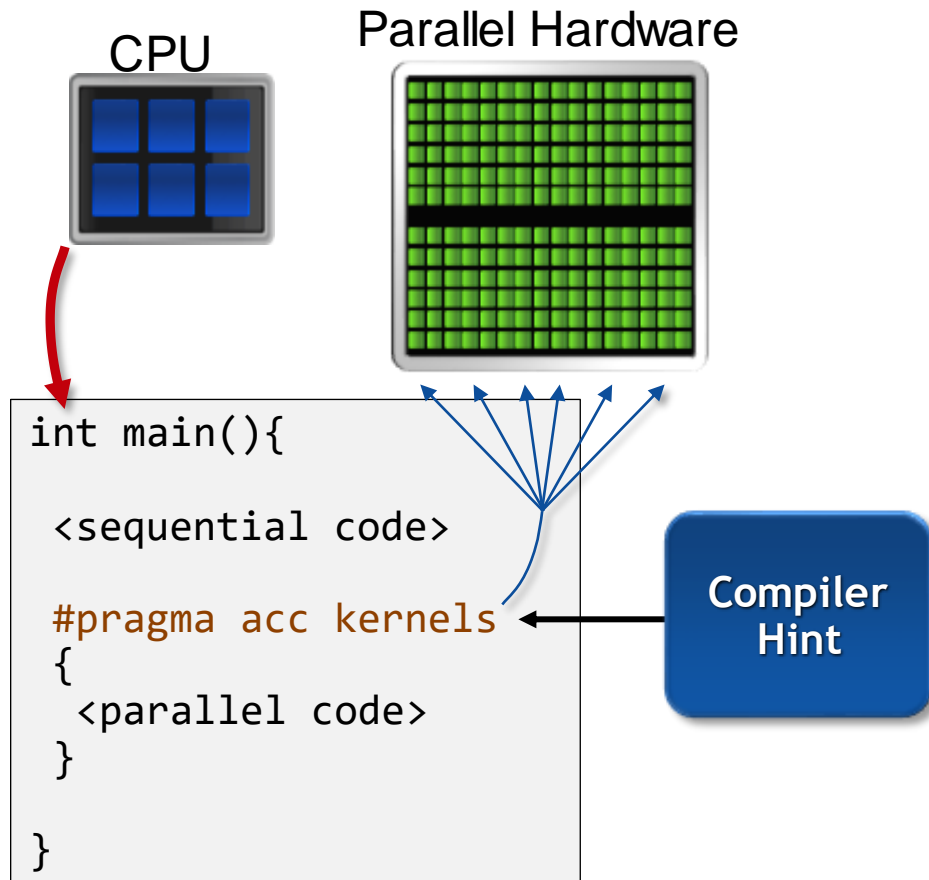
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve

OPENACC



The programmer will give hints to the compiler about which parts of the code to parallelize.

The compiler will then generate parallelism for the target parallel hardware.

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

OPENACC

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

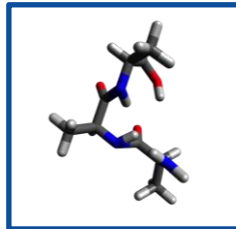
Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve

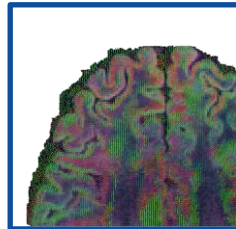
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

OPENACC SUCCESSES



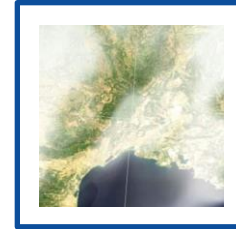
LSDalton

Quantum Chemistry
Aarhus University
12X speedup
1 week



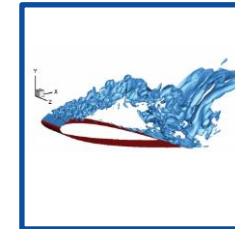
PowerGrid

Medical Imaging
University of Illinois
40 days to
2 hours



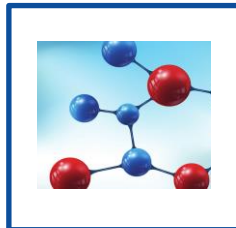
COSMO

Weather and Climate
MeteoSwiss, CSCS
40X speedup
3X energy efficiency



INCOMP3D

CFD
NC State University
4X speedup



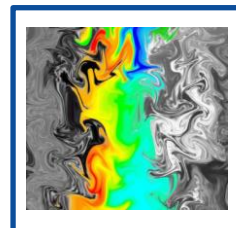
NekCEM

Comp Electromagnetics
Argonne National Lab
2.5X speedup
60% less energy



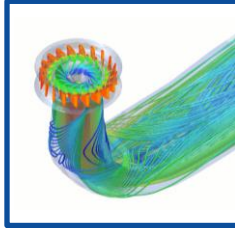
**MAESTRO
CASTRO**

Astrophysics
Stony Brook University
4.4X speedup
4 weeks effort



CloverLeaf

Comp Hydrodynamics
AWE
4X speedup
Single CPU/GPU code



FINE/Turbo

CFD
NUMECA
International
10X faster routines
2X faster app

OPENACC SYNTAX

OPENACC SYNTAX

Syntax for using OpenACC directives in code

C/C++

```
#pragma acc directive clauses  
<code>
```

Fortran

```
!$acc directive clauses  
<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 its compilation of the code and can be freely ignored.
- “***acc***” informs the compiler that what will come is an OpenACC directive
- ***Directives*** are commands in OpenACC for altering our code.
- ***Clauses*** are specifiers or additions to directives.

EXAMPLE CODE

LAPLACE HEAT TRANSFER

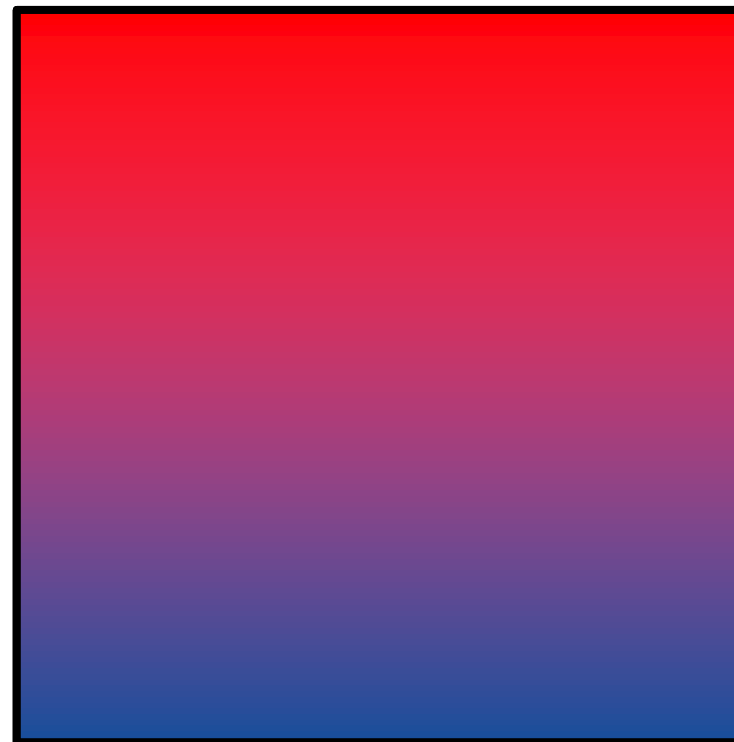
Very Hot Room Temp



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

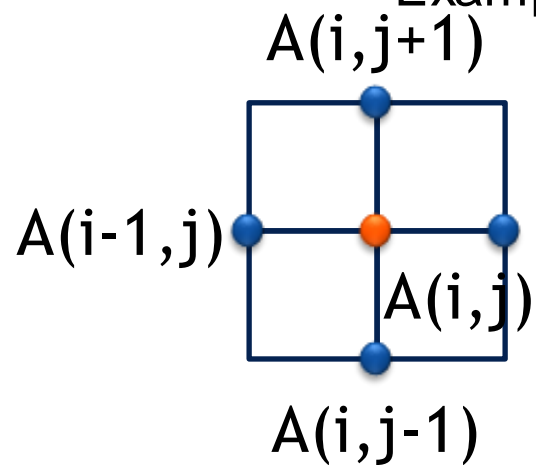
We will apply a consistent heat to the top of the plate.

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



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

JACOBI ITERATION: C CODE

```
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++) {  
  
            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]));  
        }  
    }  
  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

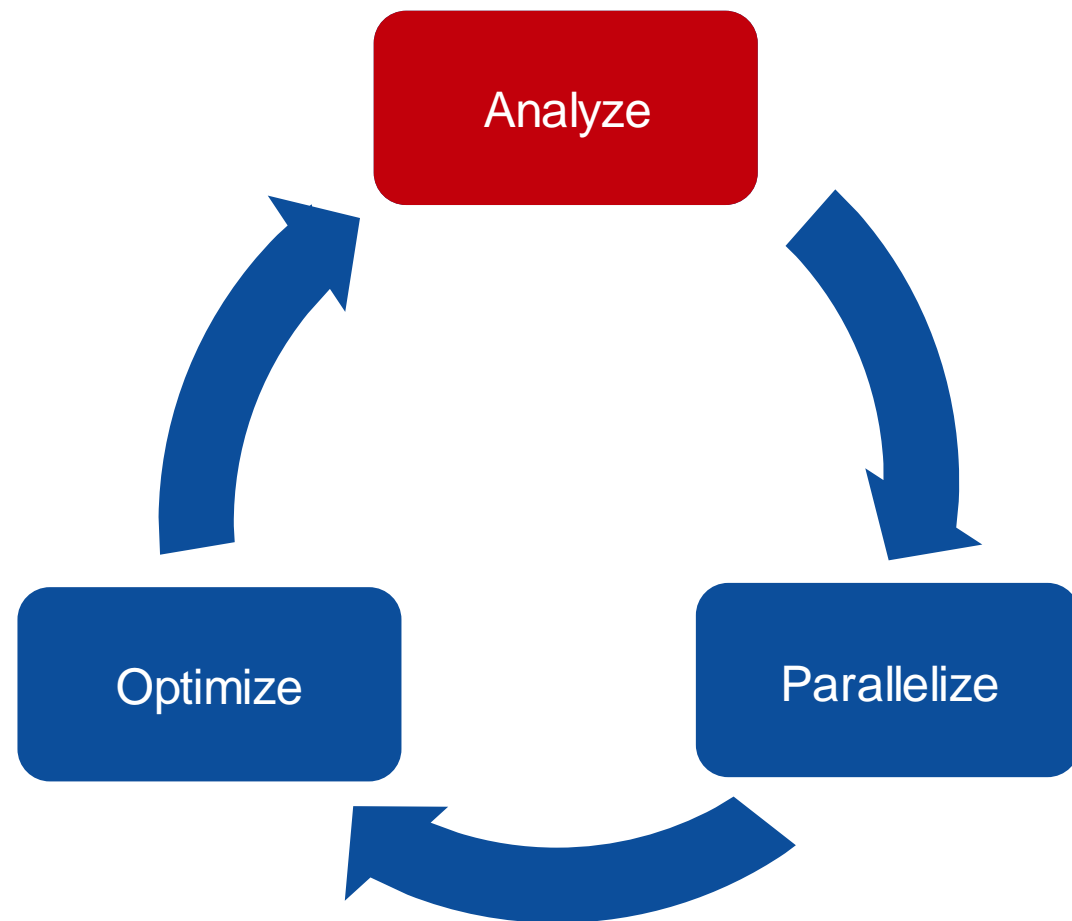
Compute max error for convergence

Swap input/output arrays

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.



OPENACC PARALLEL DIRECTIVE

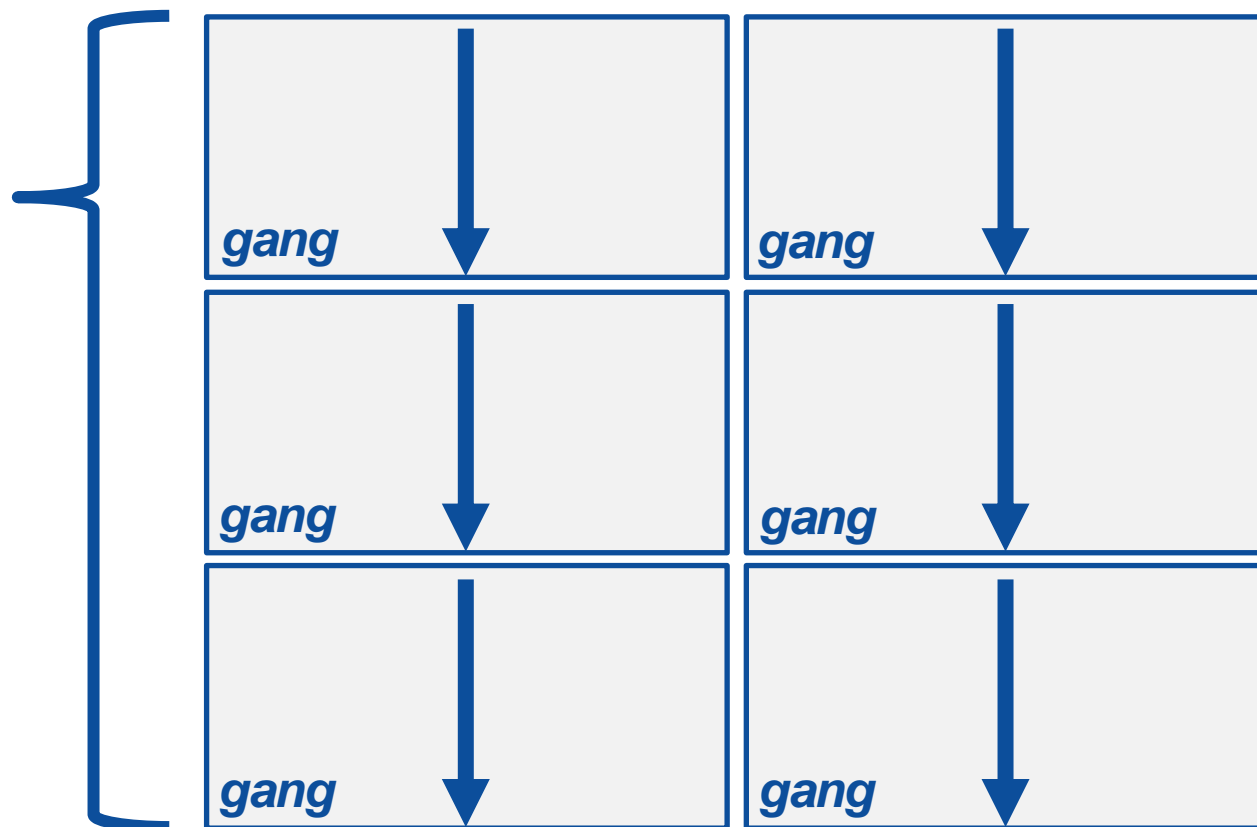
OPENACC PARALLEL DIRECTIVE

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

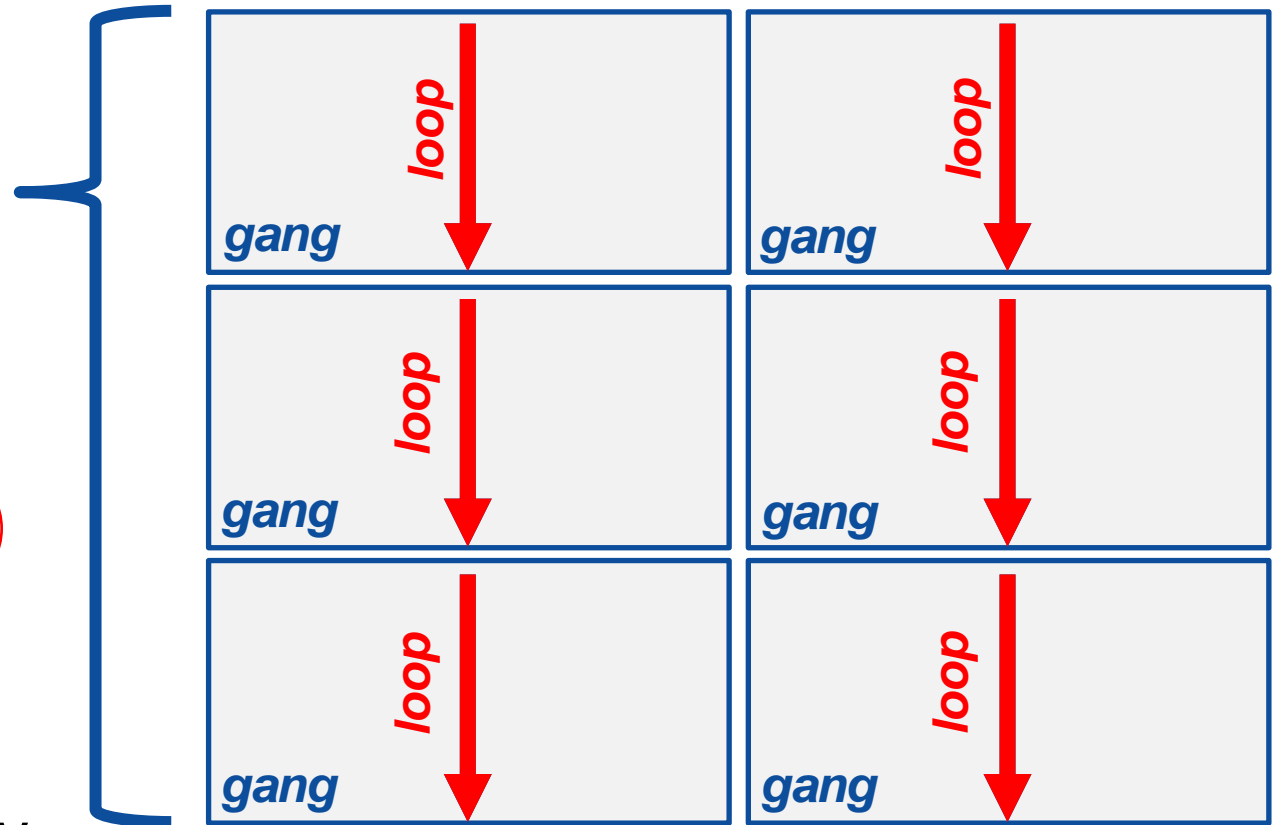
Expressing parallelism

```
#pragma acc parallel  
{
```

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

```
}
```

This loop will be
executed redundantly
on each gang



OPENACC PARALLEL DIRECTIVE

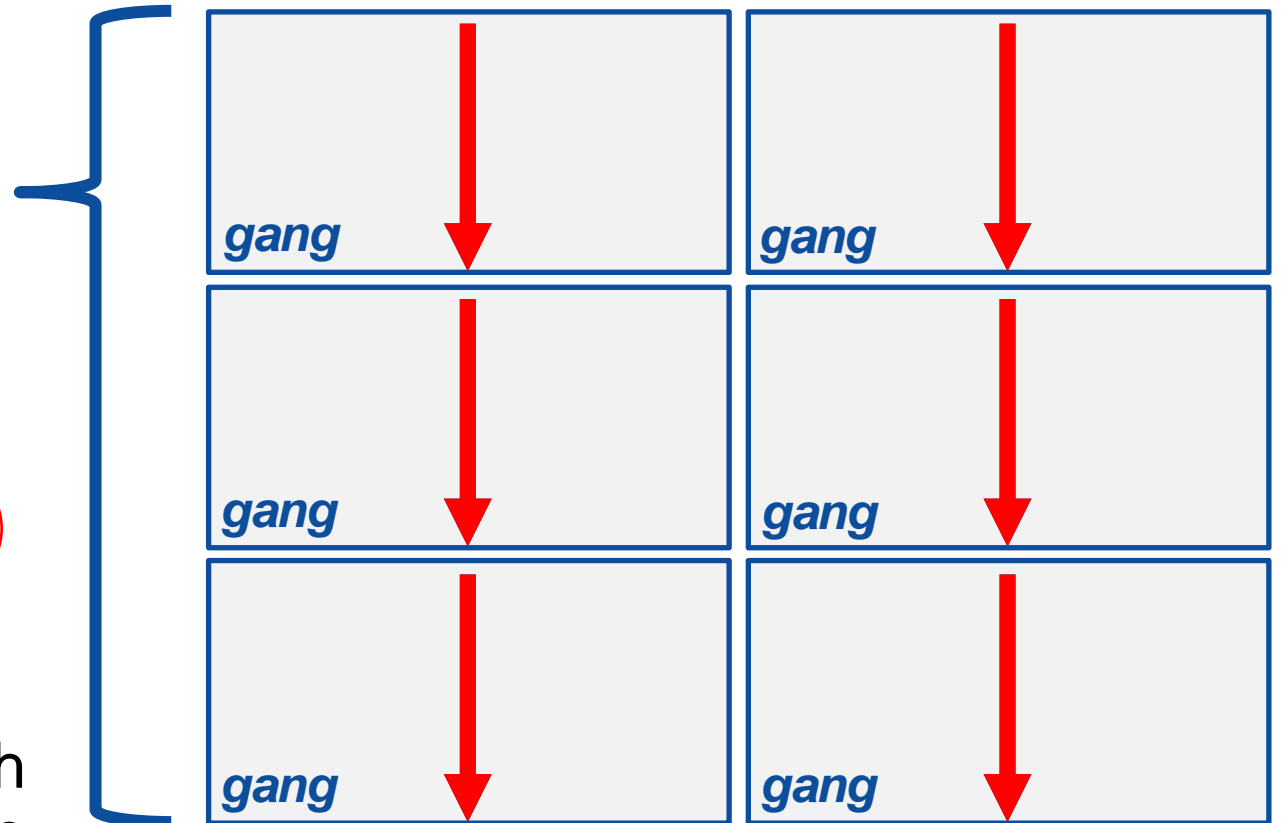
Expressing parallelism

```
#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++

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < 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
- 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

C/C++

```
#pragma acc parallel loop
for(int i = 0; i < 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

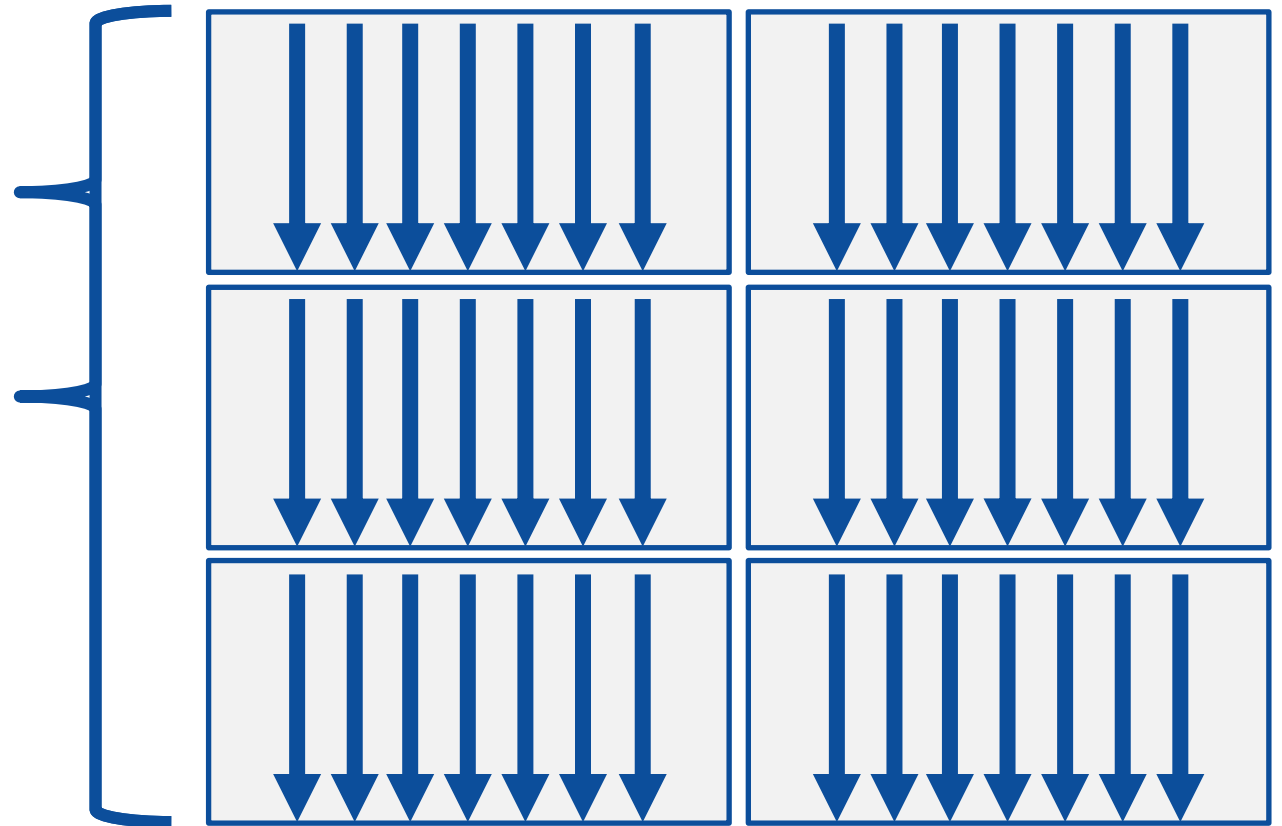
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#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

```
#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 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

PARALLELIZE WITH OPENACC PARALLEL LOOP

```
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++) {

            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
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Parallelize first loop nest,
max *reduction* required.

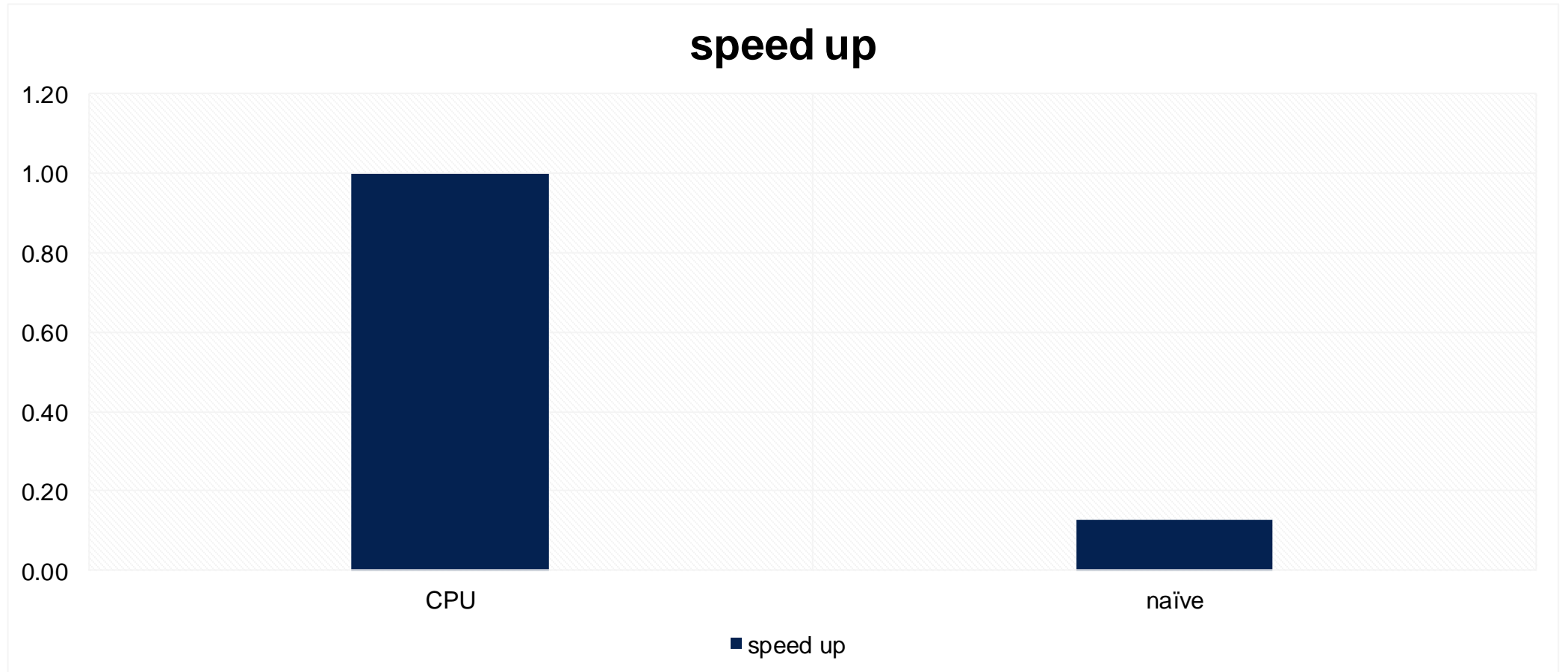
Parallelize second loop.

We didn't detail *how* to parallelize the loops, just *which* loops to parallelize.

BUILDING THE CODE (GPU)

```
pgc++ -acc -fast -Minfo=accel -ta=nvidia,cc75 heat_acc_naive.cpp -O3 -o
heat_acc_naive.x
main:
    39, Generating Tesla code
        42, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
            Generating reduction(max:error)
        43, #pragma acc loop seq
    39, Generating implicit copyin(A[:1000][:1000]) [if not already present]
        Generating implicit copy(error) [if not already present]
        Generating implicit copyout(Anew[1:998][1:998]) [if not already present]
    43, Complex loop carried dependence of A->,A->->,Anew->,Anew->-> prevents
parallelization
    48, Generating Tesla code
        51, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        52, #pragma acc loop seq
    48, Generating implicit copyin(Anew[1:998][1:998]) [if not already present]
        Generating implicit copyout(A[1:998][1:998]) [if not already present]
    52, Complex loop carried dependence of Anew->->,Anew->,A->->,A-> prevents
parallelization
```


OPENACC SPEED-UP



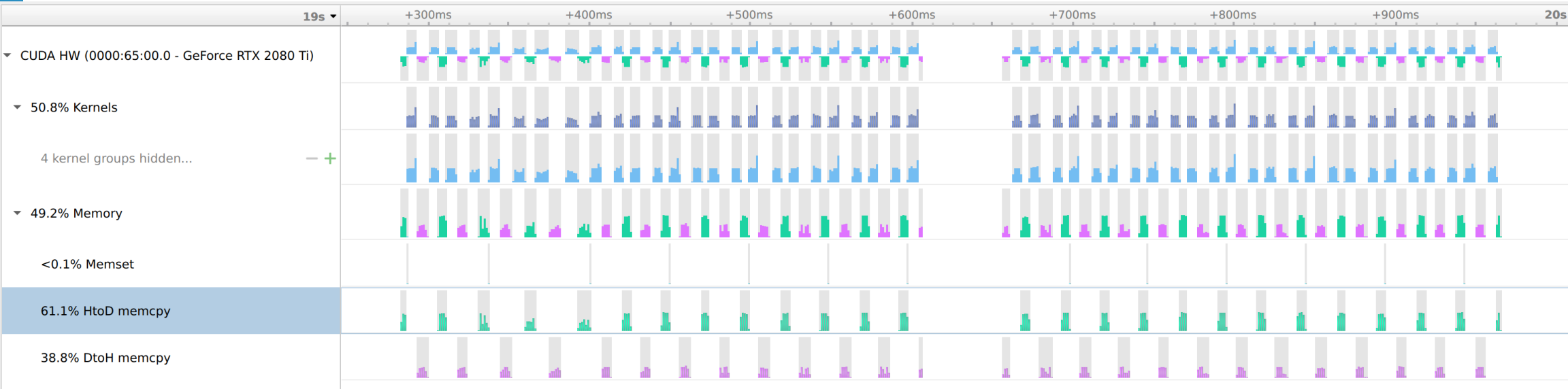
... THIS IS EMBARRASSING ...

- Nsight system is your (first) friend when you don't know what's going on with (NVIDIA) GPUs
- Command line way of profiling with nsight system

```
nsys profile -o heat_acc_naive ./heat_acc_naive.x
```

- Then open the output file with the GUI interface

NSIGHT SYSTEM PROFILE



- Data movements blobs (red/pink for device to host transfer, green for host to device transfer): for every iteration, data is copied from host to device in small chunks, and after compute they are copied back to host.
- (This should be avoided.)
- Most of the CUDA kernels here aren't the actual compute kernels – rather they are some `fill` kernels OpenACC generates.

... THIS IS EMBARRASSING ...

- We want to keep data on the device until all computations are done.
- We can tell the compiler to, either:
 1. "You can use unified virtual memory" (UVM/CUDA managed memory)
 2. "Please move these data to device when I *tell* you to do so" (explicit data movement)
- Managed memory: compiler prepare two copies of the same data, one on host and one on device. Synchronization between these two copies are *managed* automatically.
- If for a period all data accesses are from the device, no data movement will be performed in practice.

BUILDING THE CODE (GPU)

```
pgc++ -acc -fast -Minfo=accel -ta=nvidia,cc75,managed heat_acc_naive.cpp -O3 -o  
heat_acc_naive.x
```

```
main:
```

```
39, Generating Tesla code
```

```
42, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */  
Generating reduction(max:error)
```

```
43, #pragma acc loop seq
```

```
39, Generating implicit copyin(A[:1000][:1000]) [if not already present]
```

```
Generating implicit copy(error) [if not already present]
```

```
Generating implicit copyout(Anew[1:998][1:998]) [if not already present]
```

```
43, Complex loop carried dependence of A->,A->->,Anew->,Anew->-> prevents  
parallelization
```

```
48, Generating Tesla code
```

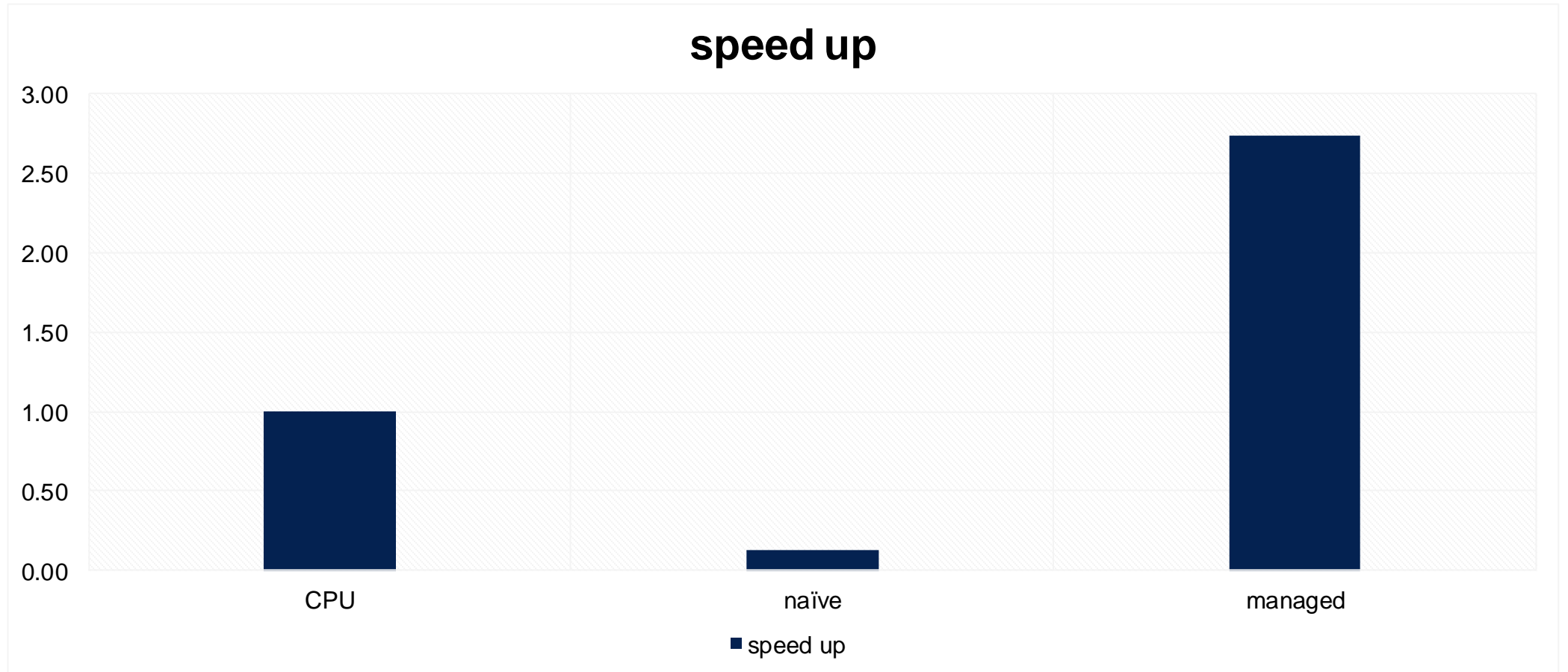
```
51, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */  
52, #pragma acc loop seq
```

```
48, Generating implicit copyin(Anew[1:998][1:998]) [if not already present]
```

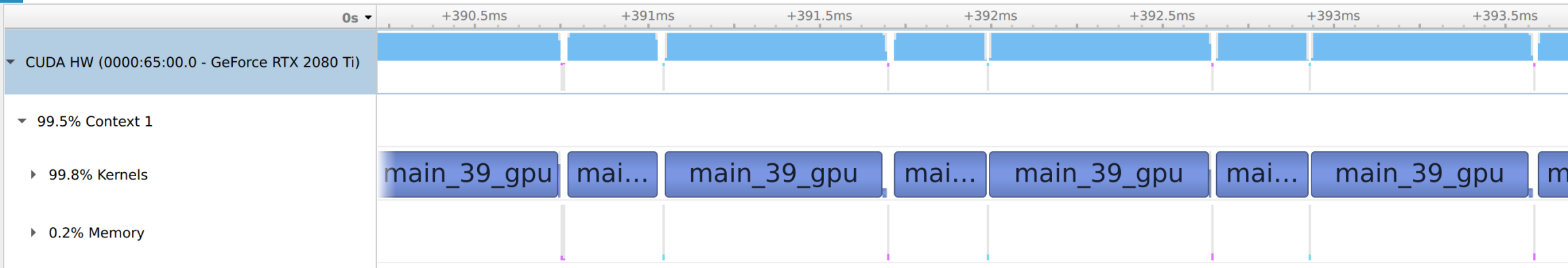
```
Generating implicit copyout(A[1:998][1:998]) [if not already present]
```

```
52, Complex loop carried dependence of Anew->->,Anew->,A->->,A-> prevents  
parallelization
```

OPENACC SPEED-UP



NSIGHT SYSTEM PROFILE



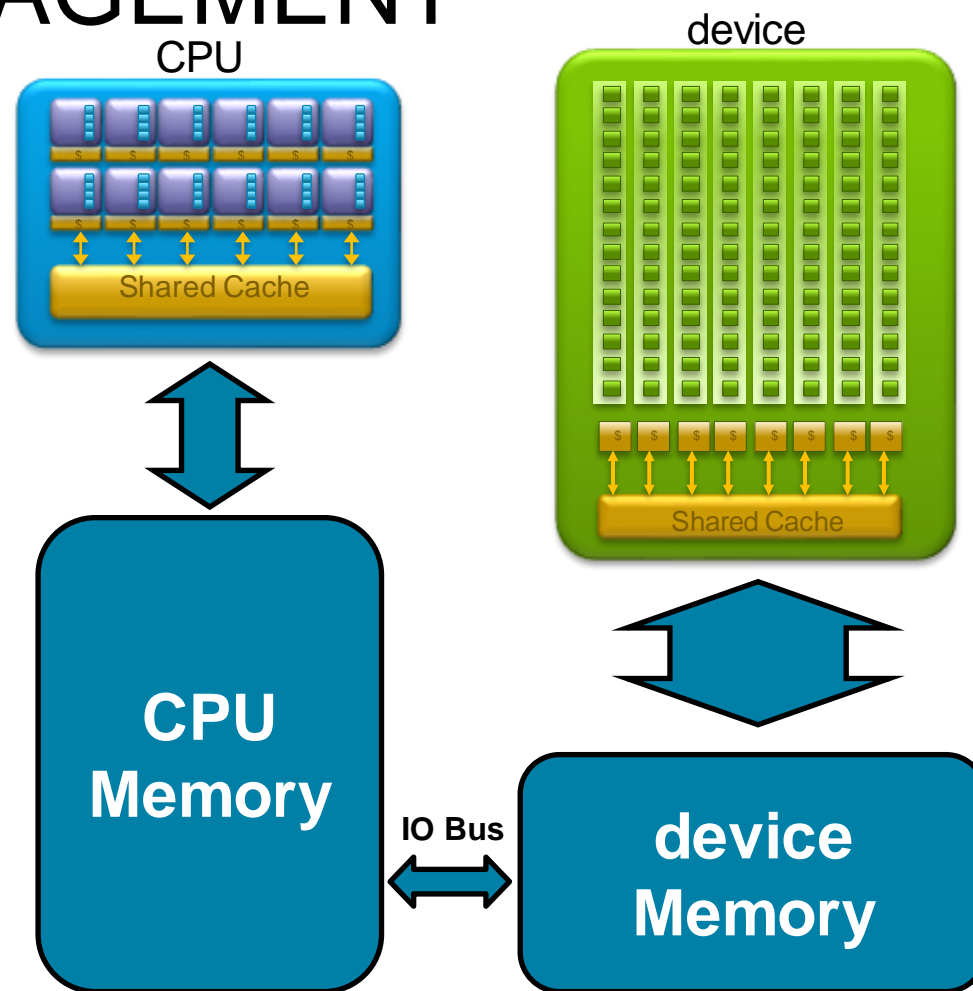
- Now the only data movements in the main loop are small memset's and memcpy's (for the reduction on error)

OPTIMIZE DATA MOVEMENT

EXPLICIT MEMORY MANAGEMENT

Key problems

- Many parallel accelerators (such as devices) have a separate memory pool from the host
- These separate memories can become out-of-sync and contain completely different data
- Transferring between these two memories can be a very time consuming process



OPENACC DATA DIRECTIVE

Definition

- The data directive defines a lifetime for data on the device
- During the region data should be thought of as residing on the accelerator
- Data clauses allow the programmer to control the allocation and movement of data

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

```
!$acc data clauses  
    < Sequential and/or Parallel code >  
!$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

- 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

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

ARRAY SHAPING (CONT.)

Multi-dimensional Array shaping

```
copy(array[0:N][0:M])
```

C/C++

Both of these examples copy a 2D array to the device

```
copy(array(1:N, 1:M))
```

Fortran

ARRAY SHAPING (CONT.)

Partial Arrays

```
copy(array[i*N/4:N/4])
```

C/C++

Both of these examples copy only $\frac{1}{4}$ of the full array

```
copy(array(i*N/4:i*N/4+N/4))
```

Fortran

STRUCTURED DATA DIRECTIVE

Example

```
#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];  
  }  
}
```

Action

Copy A & B from
Device to Device
Copy C from
Device to Device

Host Memory



Device memory



OPTIMIZED DATA MOVEMENT

```
#pragma acc data copyin(A[0:n][0:m]) copyout(Anew[0:n][0:m])
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++) {

            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
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Copy A to/from the accelerator only when needed.

Copy initial condition of Anew, but not final value

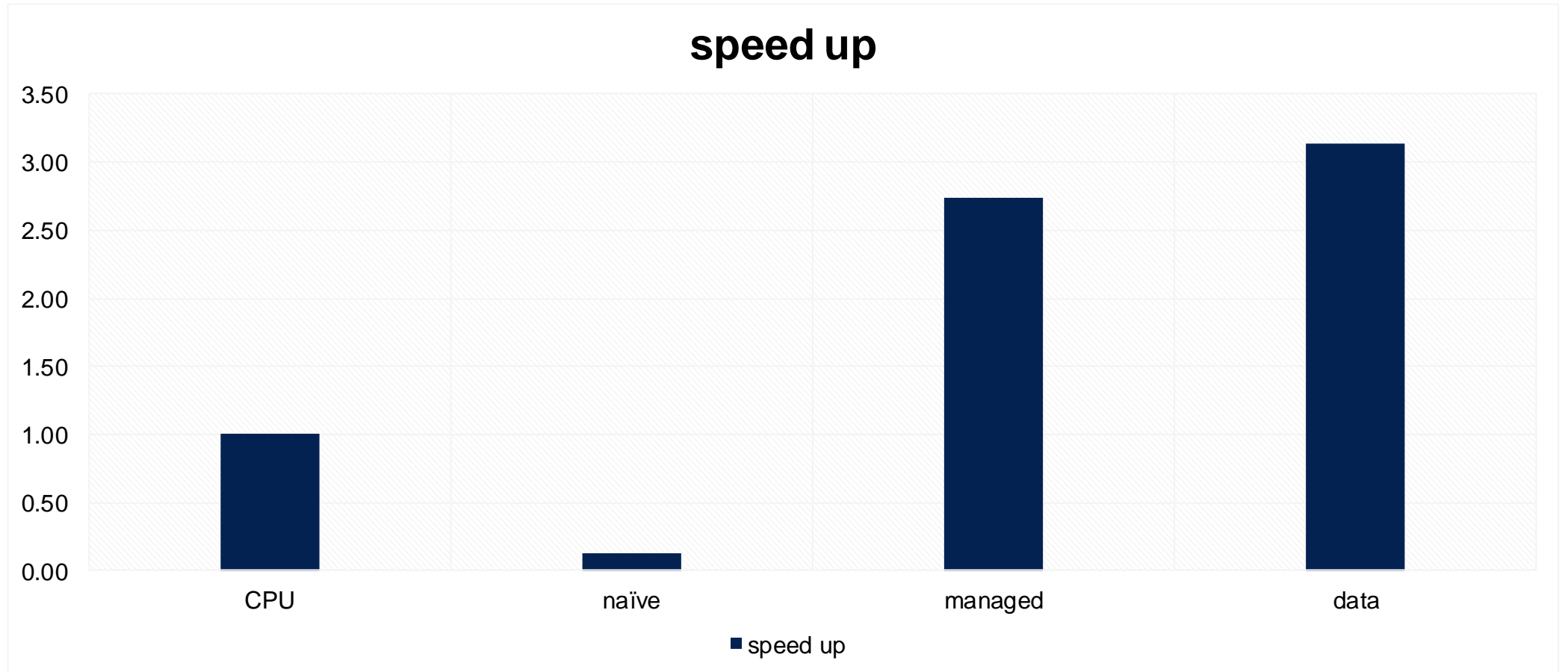
REBUILD THE CODE

```
pgc++ -acc -fast -Minfo=accel -ta=nvidia,cc75 heat_acc_data.cpp -O3 -o
heat_acc_data.x
main:
    39, Generating copyout(Anew[:n][:m]) [if not already present]
    Generating copyin(A[:n][:m]) [if not already present]
    40, Generating Tesla code
    43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        Generating reduction(max:error)
    44, #pragma acc loop seq
    40, Generating implicit copy(error) [if not already present]
    44, Complex loop carried dependence of A->,A->->,Anew->,Anew->-> prevents
parallelization
    49, Generating Tesla code
    52, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    53, #pragma acc loop seq
    53, Complex loop carried dependence of Anew->->,Anew->,A->->,A-> prevents
parallelization
```



Now data movement only happens at our data region.

OPENACC SPEED-UP



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

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

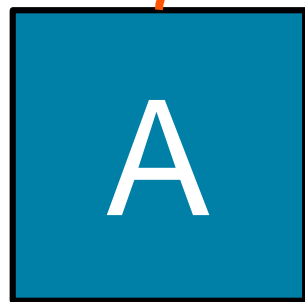
C/C++

```
!$acc update self(x(1:end_index))  
!$acc update device(x(1:end_index))
```

Fortran

OPENACC UPDATE DIRECTIVE

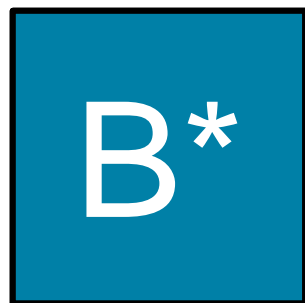
```
#pragma acc update device(A[0:N])
```



CPU Memory



device Memory



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

The data must exist on both the CPU and device for the update directive to work.

SYNCHRONIZE DATA WITH UPDATE

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

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

void initialize_array(int* A, int N){
    for(int i = 0; i < N; i++){
        A[i] = i;
    }
    #pragma acc update device(A[0:N])
}
```

- Inside the **initialize** function we alter the host copy of 'A'
- This means that after calling **initialize** the host and device copy of 'A' are out-of-sync
- We use the **update** directive with the **device** clause to update the device copy of 'A'
- Without the **update** directive later compute regions will use incorrect data.

LOOP OPTIMIZATIONS

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

```
#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)

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

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

TILE CLAUSE

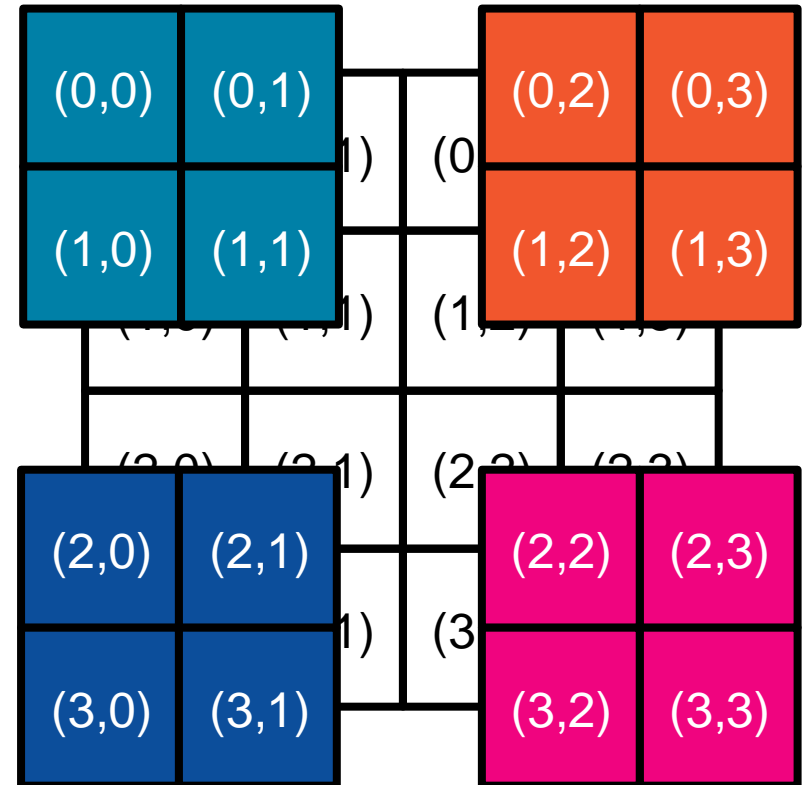
- **tile (x , y , z , ...)**
- Breaks multidimensional loops into “tiles” or “blocks”
- Can increase data locality in some codes
- Will be able to execute multiple “tiles” simultaneously

```
#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

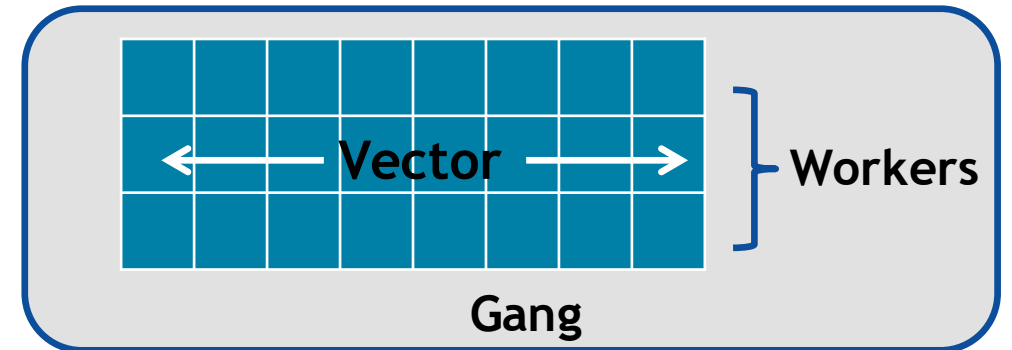
```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
    for(int y = 0; y < 4; y++){
        array[x][y]++;
    }
}
```

tile (2 , 2)



GANG WORKER VECTOR

- Gang / Worker / Vector defines the various levels of parallelism we can achieve with OpenACC
- This parallelism is most useful when parallelizing multi-dimensional loop nests
- OpenACC allows us to define a generic Gang / Worker / Vector model that will be applicable to a variety of hardware, but we will focus a little bit on a GPU specific implementation



BACK TO THE EXAMPLE

```
pgc++ -acc -fast -Minfo=accel -ta=nvidia,cc75 heat_acc_data.cpp -O3 -o
heat_acc_data.x
main:
    39, Generating copyout(Anew[:n][:m]) [if not already present]
    Generating copyin(A[:n][:m]) [if not already present]
    40, Generating Tesla code
    43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    Generating reduction(max:error)
    44, #pragma acc loop seq
    40, Generating implicit copy(error) [if not already present]
    44, Complex loop carried dependence of A->,A->->,Anew->,Anew->-> prevents
parallelization
    49, Generating Tesla code
    52, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    53, #pragma acc loop seq
    53, Complex loop carried dependence of Anew->->,Anew->,A->->,A-> prevents
parallelization
```




Now data movement only happens at our data region.

OPTIMIZED LOOP

```
#pragma acc data copyin(A[0:n][0:m]) copyout(Anew[0:n][0:m])
while ( err > tol && iter < iter_max ) {
    err=0.0;
    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++) {
        #pragma acc loop independent
        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
    for( int j = 1; j < n-1; j++) {
        #pragma acc loop independent
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```



Tell compiler that these
inner loop and really
independent from each
other

REBUILD THE CODE

```
pgc++ -acc -fast -Minfo=accel -ta=nvidia,cc75 heat_acc_tile.cpp -O3 -o  
heat_acc_tile.x
```

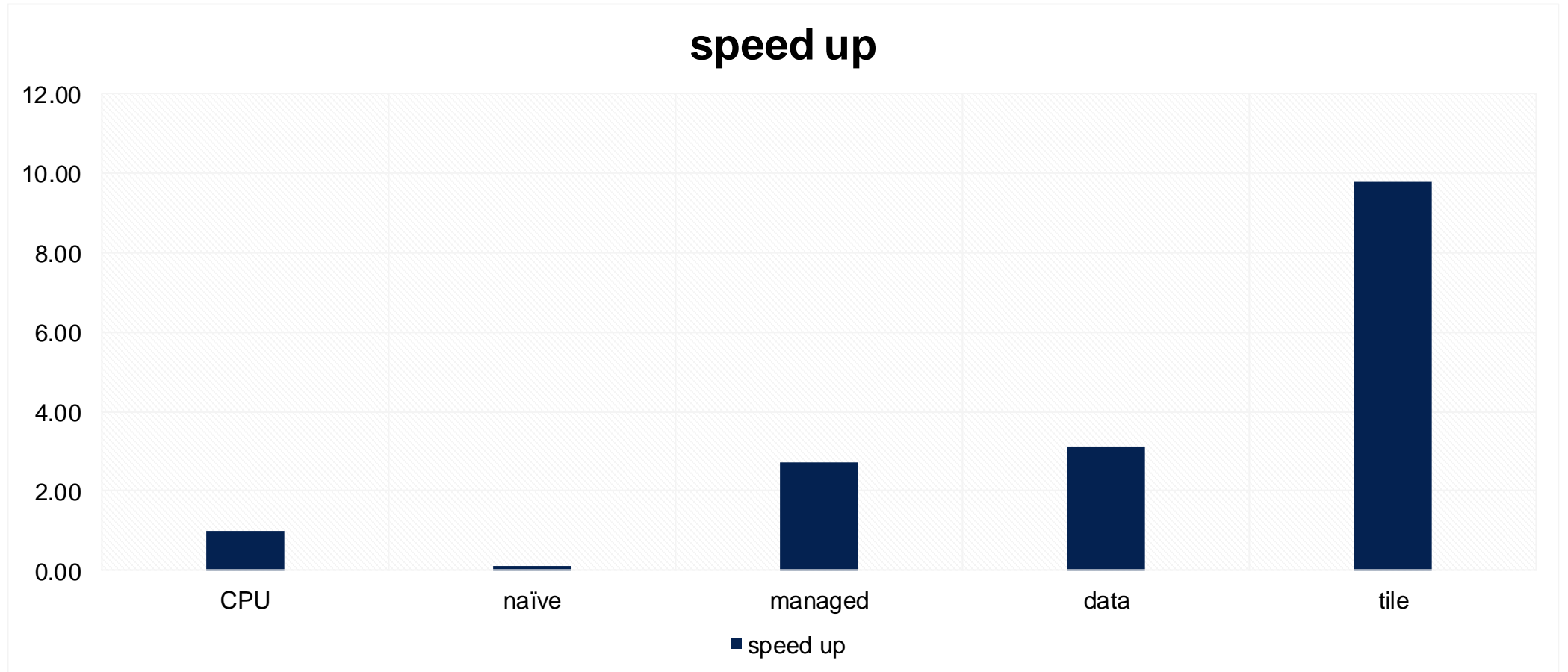
```
main:
```

```
39, Generating copyout(Anew[:n][:m]) [if not already present]  
    Generating copyin(A[:n][:m]) [if not already present]  
40, Generating Tesla code  
    43, #pragma acc loop gang /* blockIdx.x */  
        Generating reduction(max:error)  
    45, #pragma acc loop vector(128) /* threadIdx.x */  
40, Generating implicit copy(error) [if not already present]  
45, Loop is parallelizable  
50, Generating Tesla code  
    53, #pragma acc loop gang /* blockIdx.x */  
    55, #pragma acc loop vector(128) /* threadIdx.x */  
55, Loop is parallelizable
```

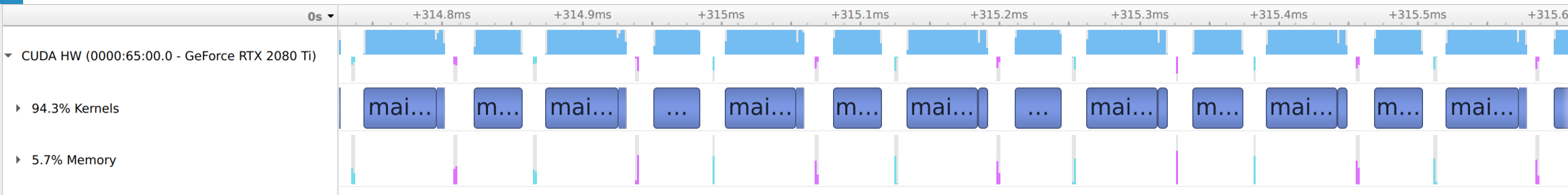


Now data movement only
happens at our data
region.

OPENACC SPEED-UP



NSIGHT SYSTEM PROFILE



- Now the compute kernels are much shorter.

LOOP OPTIMIZATION RULES OF THUMB

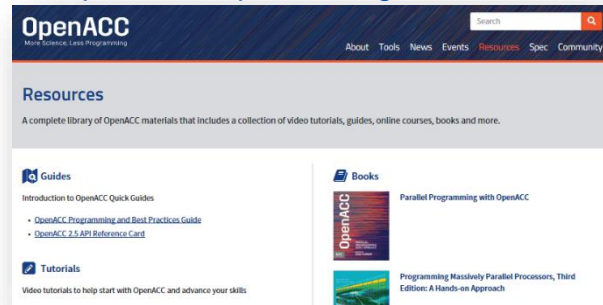
- It is rarely a good idea to set the number of gangs in your code, let the compiler decide.
- Most of the time you can effectively tune a loop nest by adjusting only the vector length.
- It is rare to use a worker loop. When the vector length is very short, a worker loop can increase the parallelism in your gang.
- When possible, the vector loop should step through your arrays
- Use the `device_type` clause to ensure that tuning for one architecture doesn't negatively affect other architectures.

OPENACC RESOURCES

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

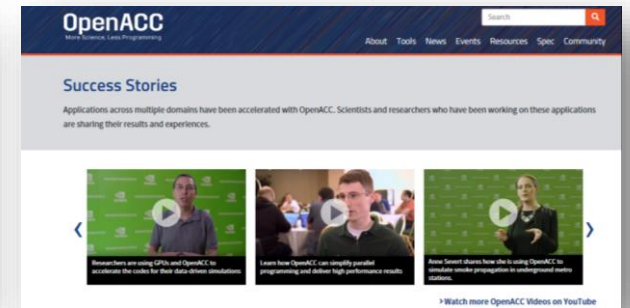
Resources

<https://www.openacc.org/resources>



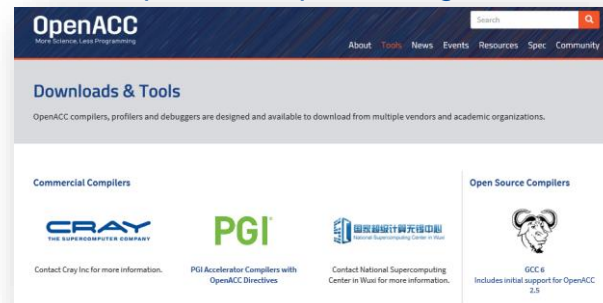
Success Stories

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



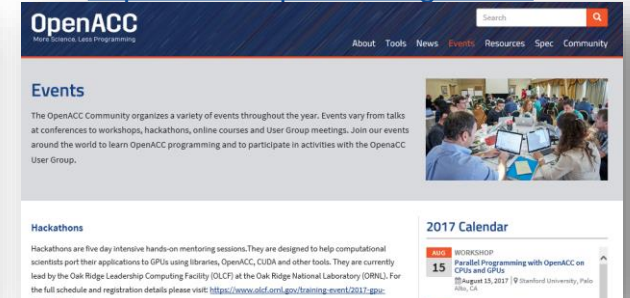
Compilers and Tools

<https://www.openacc.org/tools>



Events

<https://www.openacc.org/events>



CLOSING REMARKS

KEY CONCEPTS

In this lecture we discussed...

- How to profile a serial code to identify loops that should be accelerated
- How to use OpenACC's parallel loop directive to parallelize key loops
- How to use OpenACC's data clauses to control data movement
- How to optimize loops in the code for better performance