ZERO TO GPU HERO WITH OPENACC

Qingping Fu, 2018

AGENDA

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

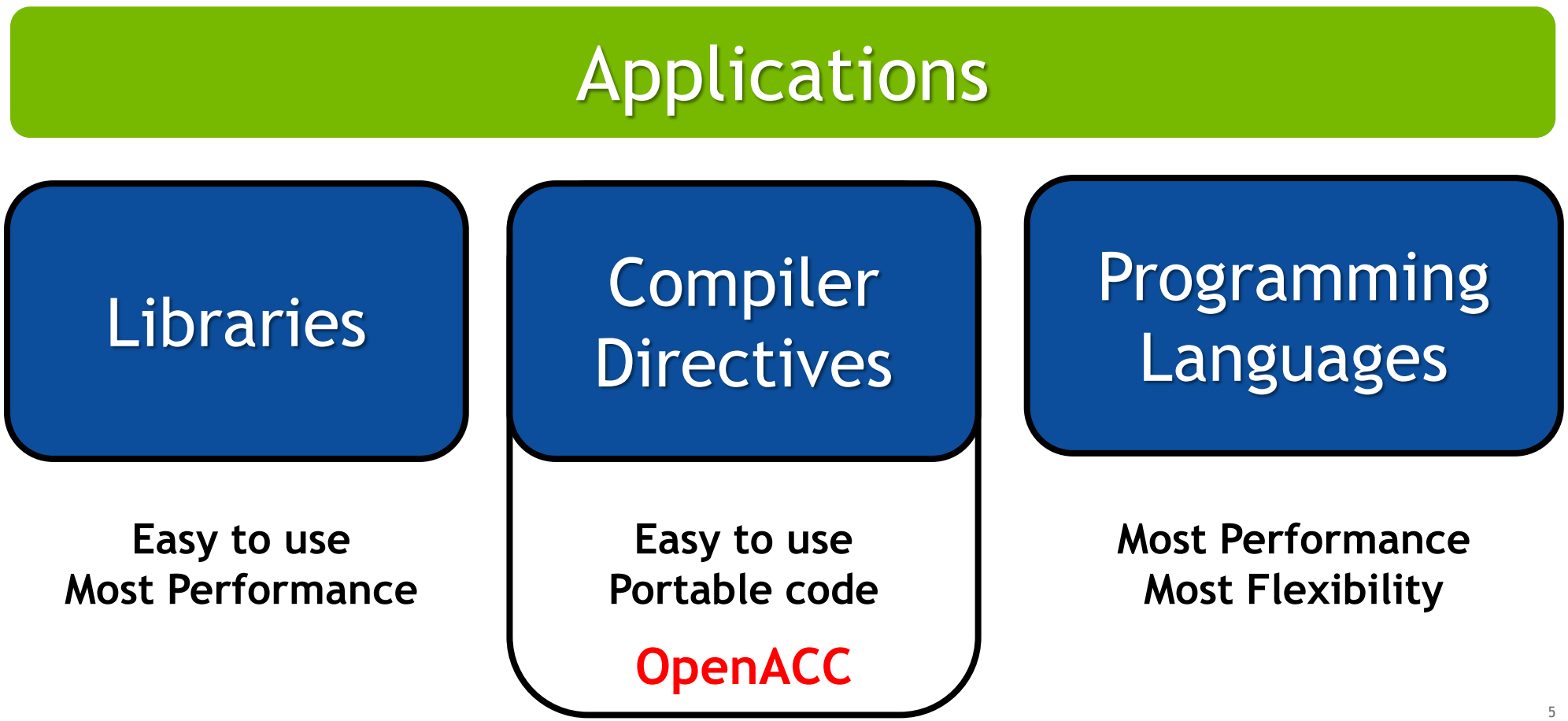
INTRODUCTION TO OPENACC

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>
  }
}
```

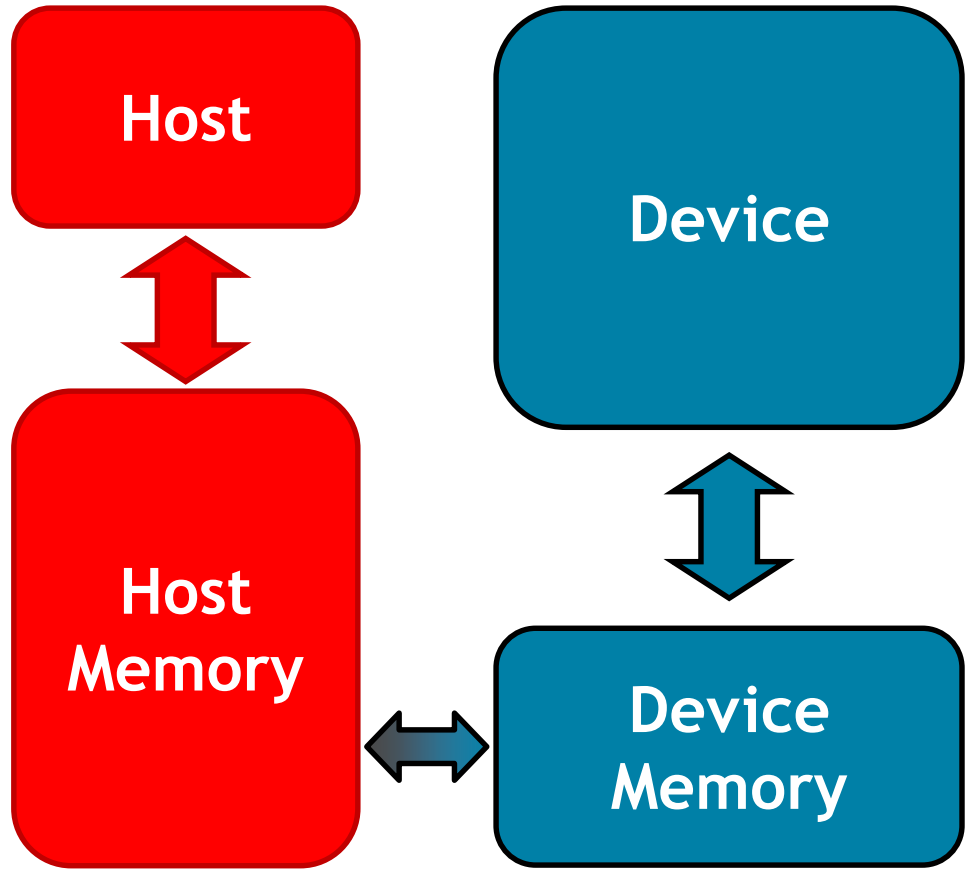
3 WAYS TO ACCELERATE APPLICATIONS



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

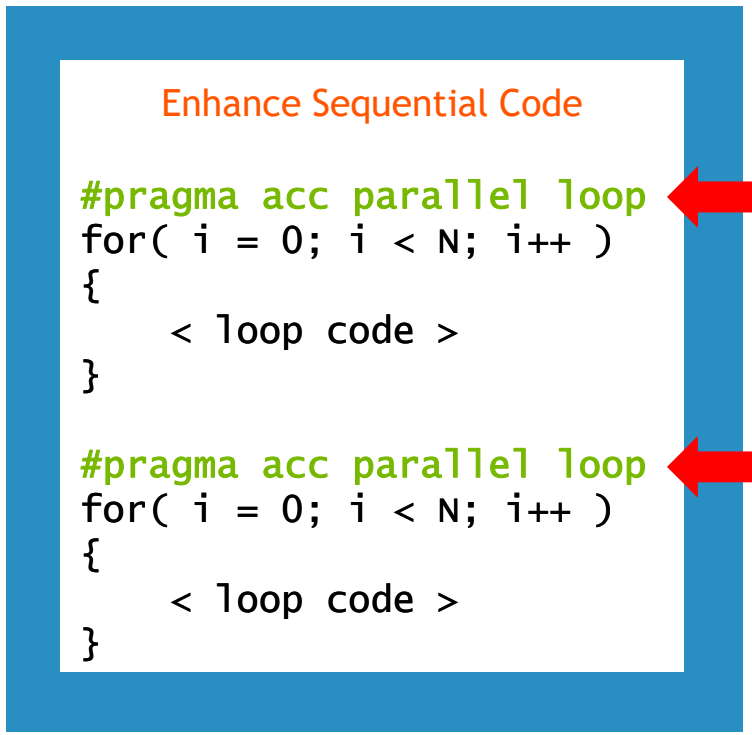
Three major strengths

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



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

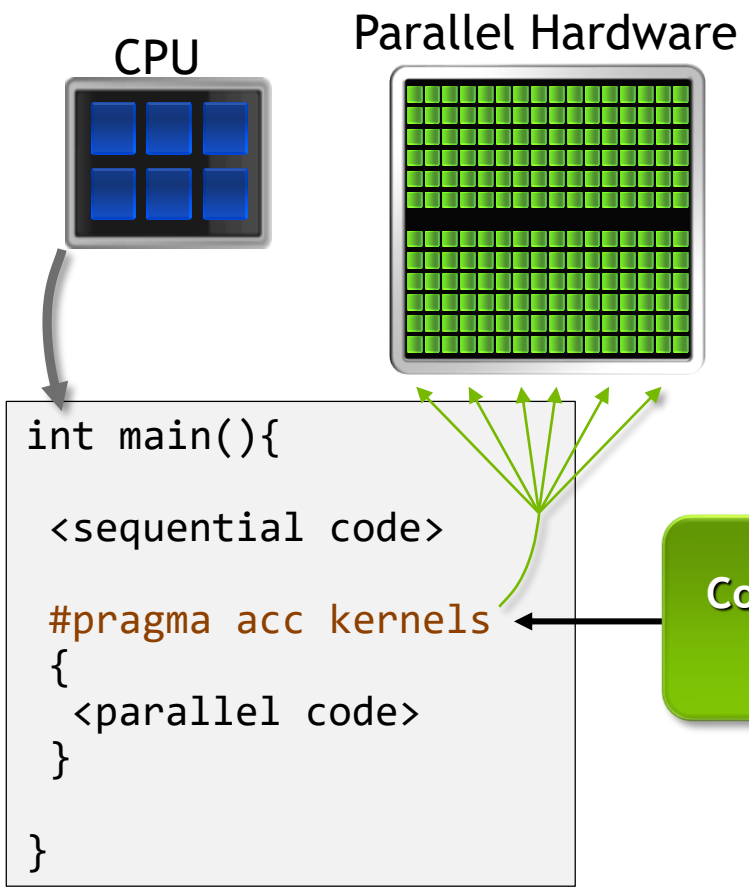
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

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

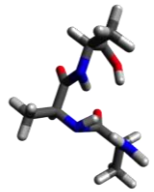
Single Source

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

Low Learning Curve

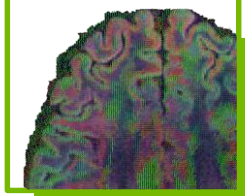
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

OPENACC SUCCESSES



LSDalton

Quantum Chemistry
Aarhus University
12X speedup
1 week

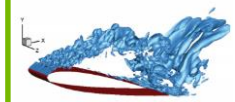


PowerGrid

Medical Imaging
University of Illinois
40 days to
2 hours

COSMO

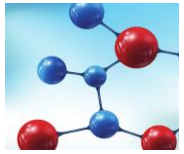
Weather and Climate
MeteoSwiss, CSCS
40X speedup
3X energy efficiency



INCOMP3D

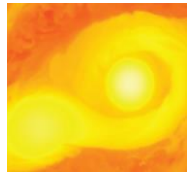
CFD
NC State University

4X speedup



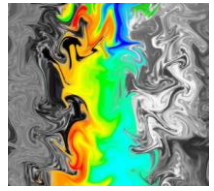
NekCEM

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 instructions the compiler in it 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

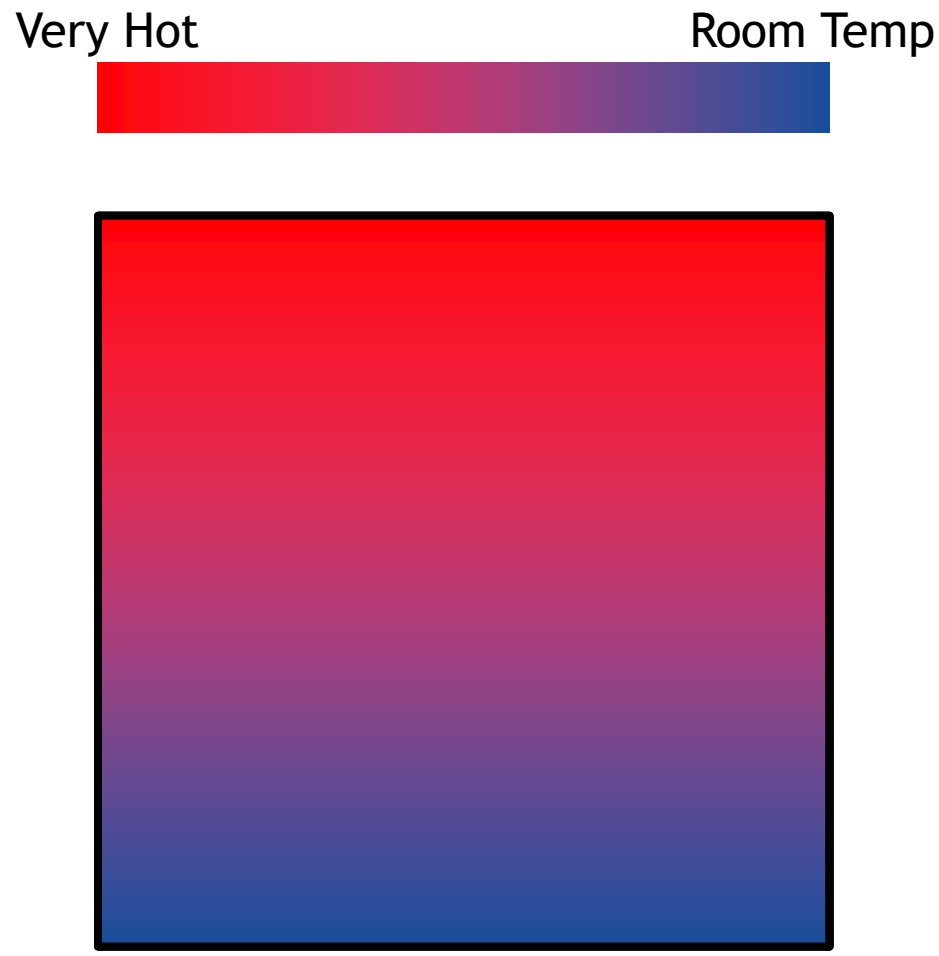
LAPLACE HEAT TRANSFER

Introduction to lab code - visual

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

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

Then, we will simulate the heat distributing across the plate.

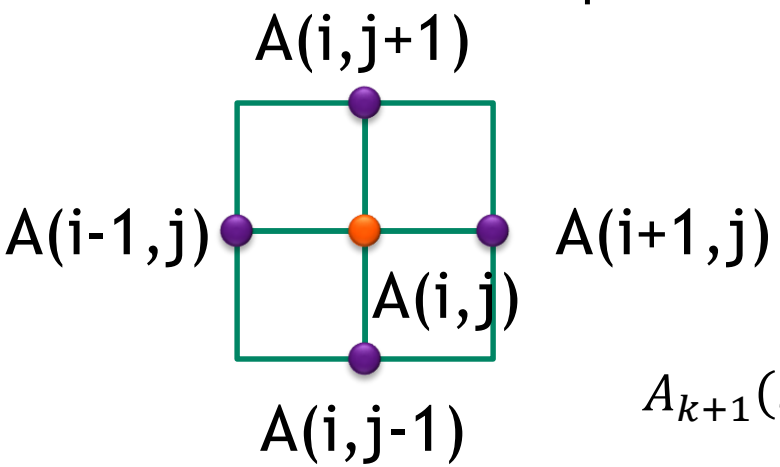


EXAMPLE: JACOBI ITERATION

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

Common, useful algorithm

Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



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

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

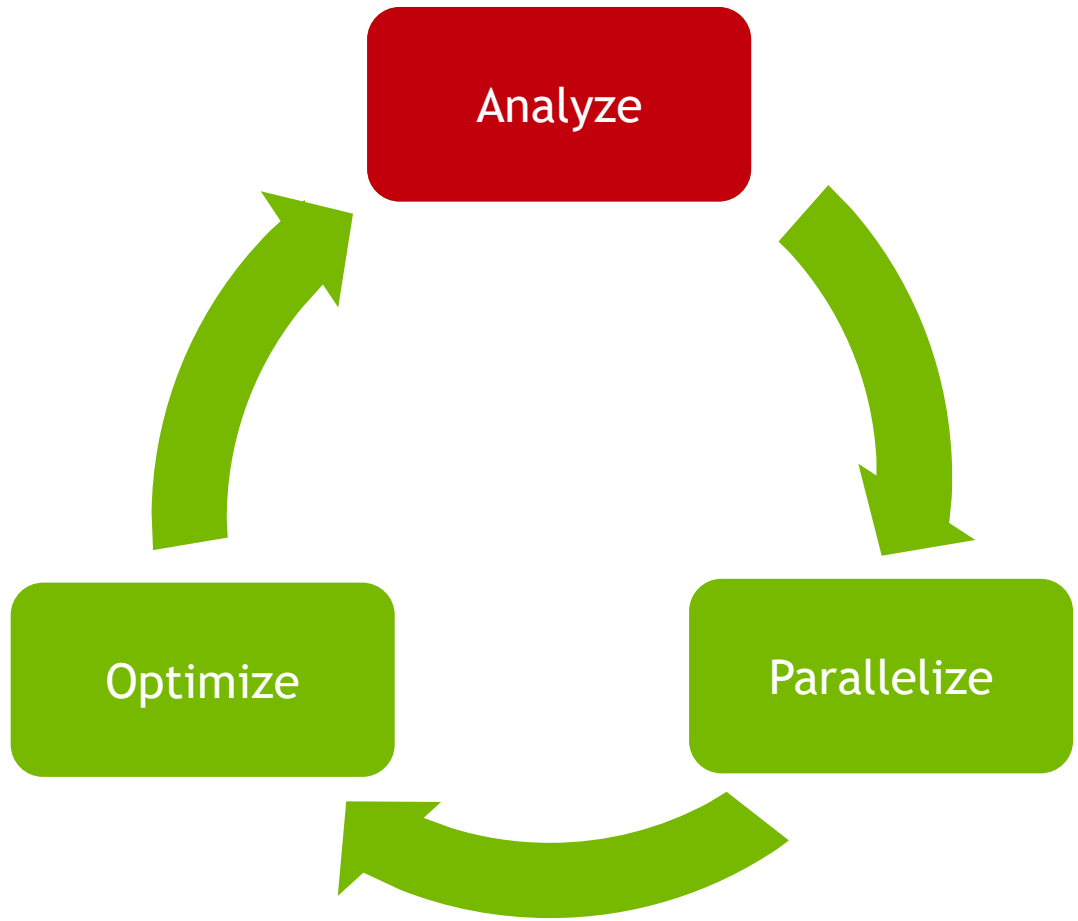
Compute max error for convergence

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.



PROFILING SEQUENTIAL CODE

Profile Your Code

Obtain detailed information about how the code ran.

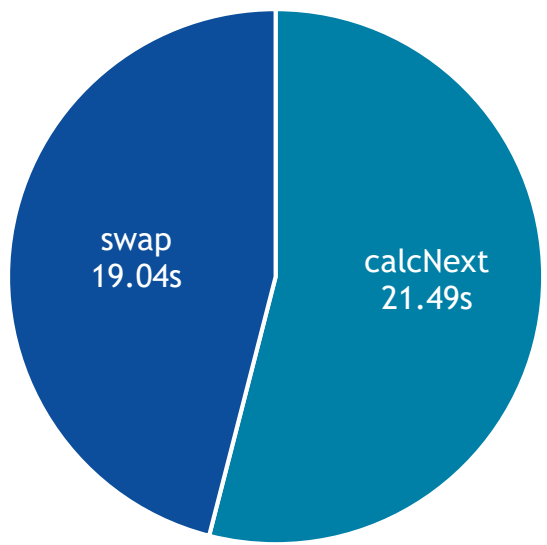
This can include information such as:

- Total runtime
- Runtime of individual routines
- Hardware counters

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

Lab Code: Laplace Heat Transfer

Total Runtime: 39.43 seconds



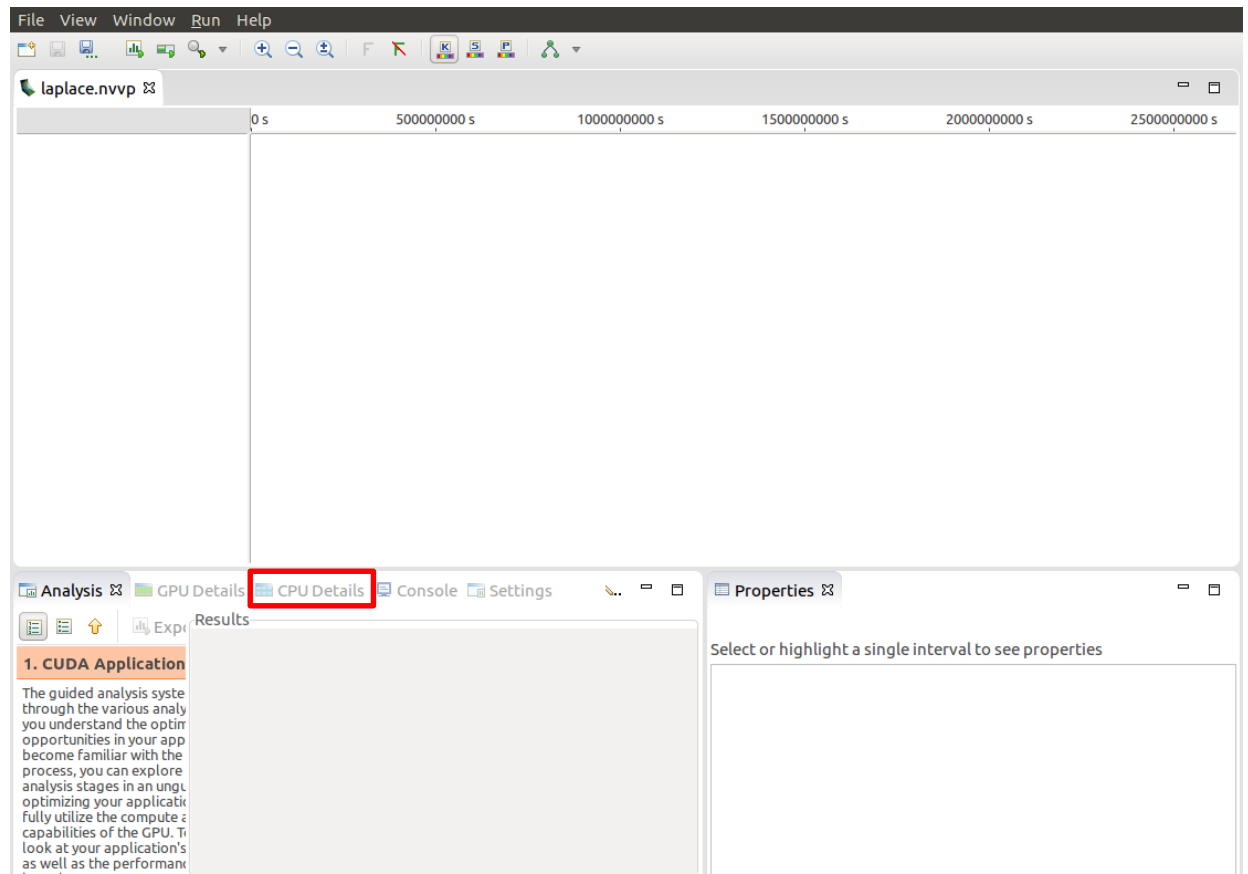
PROFILING SEQUENTIAL CODE

First sight when using PGPROF

Profiling a simple, sequential code

Our sequential program will on run on the CPU

To view information about how our code ran, we should select the “CPU Details” tab



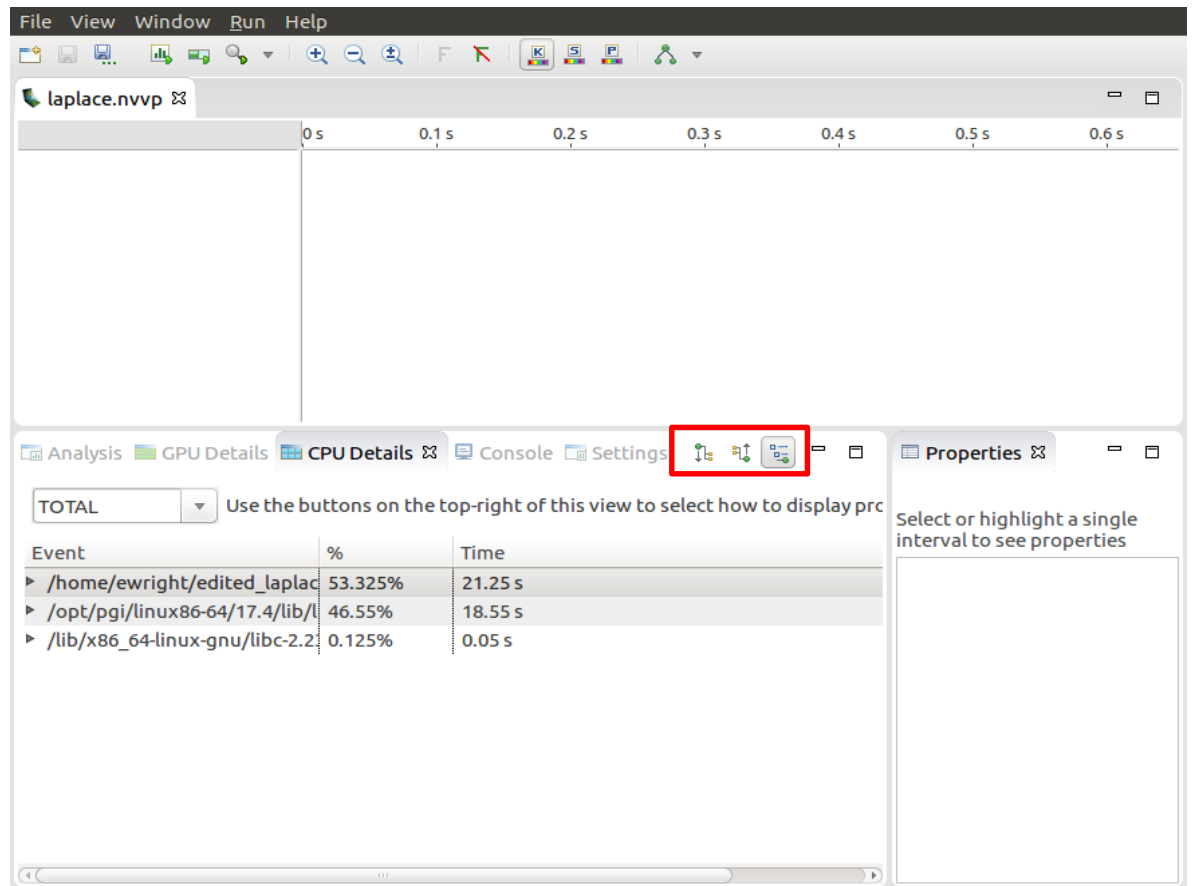
PROFILING SEQUENTIAL CODE

CPU Details

Within the “CPU Details” tab, we can see the various parts of our code, and how long they took to run

We can reorganize this info using the three options in the top-right portion of the tab

We will expand this information, and see more details about our code



PROFILING SEQUENTIAL CODE

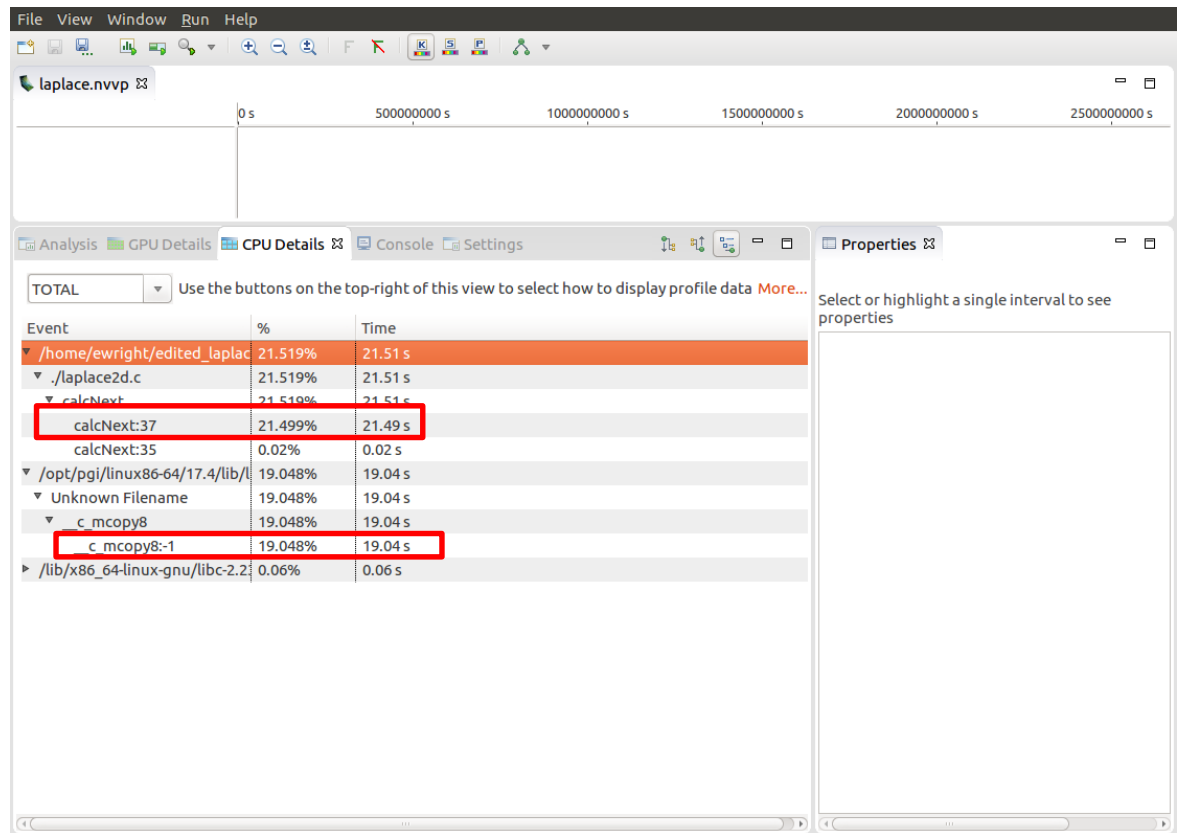
CPU Details

We can see that there are two places that our code is spending most of its time

21.49 seconds in the “calcNext” function

19.04 seconds in a memcpy function

The c_mcopy8 that we see is actually a compiler optimization that is being applied to our “swap” function



OPENACC PARALLEL DIRECTIVE

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

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; 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

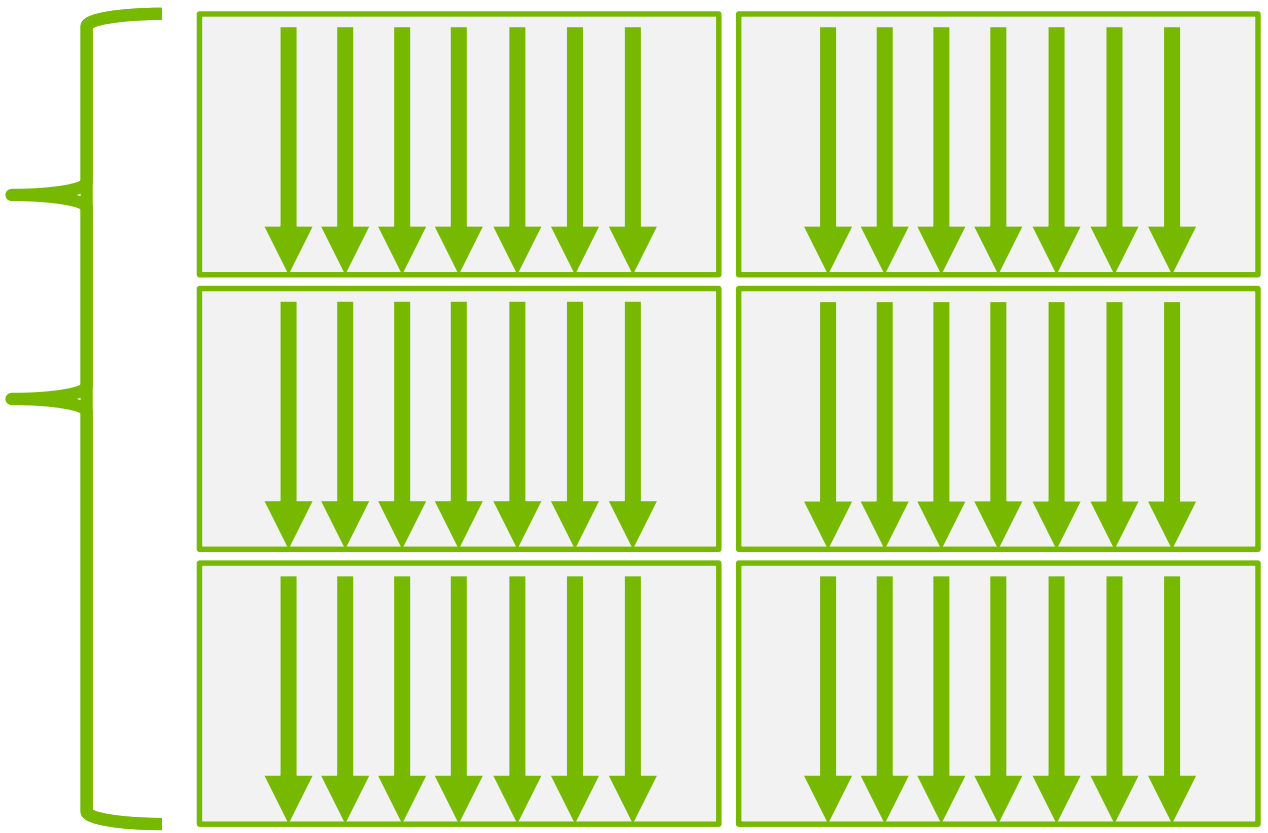
Expressing parallelism

```
#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.



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.

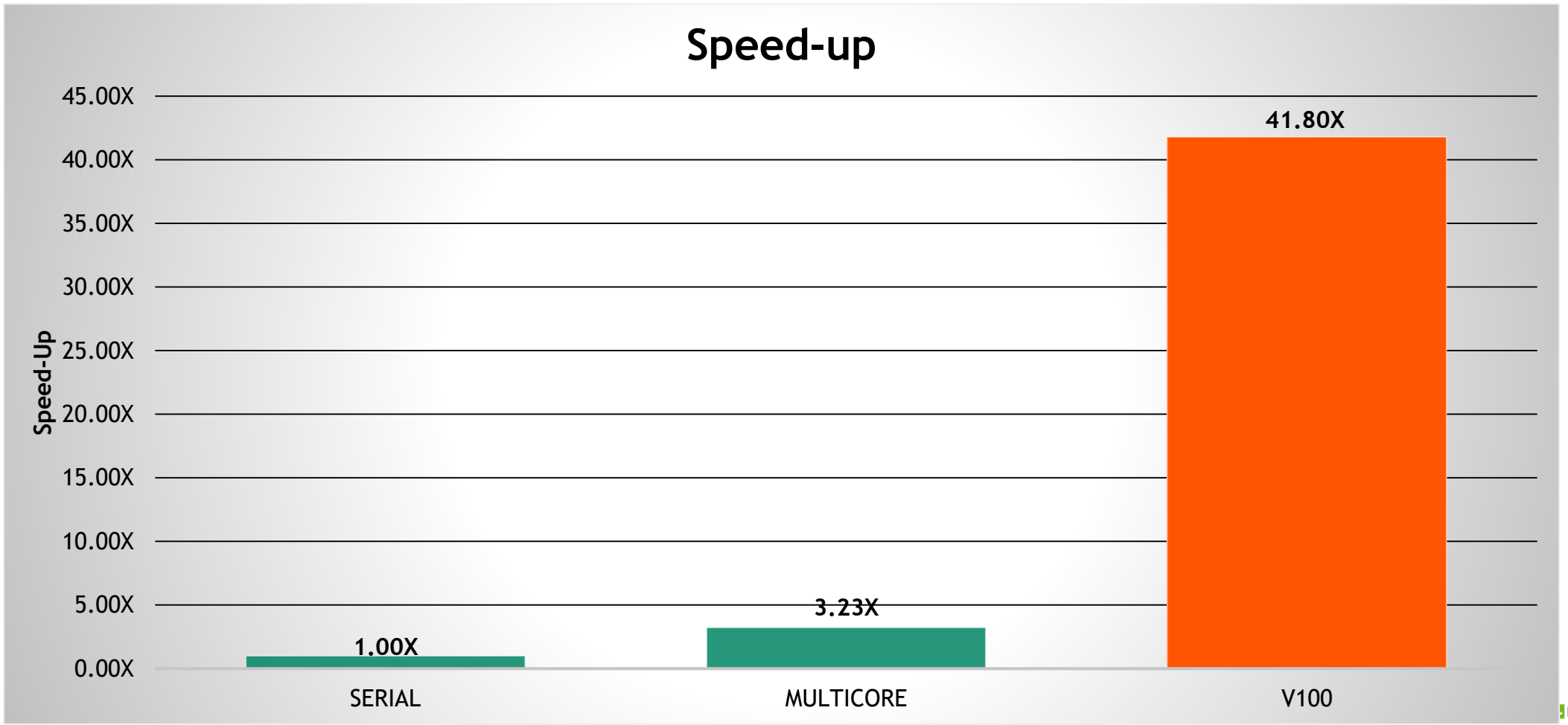
BUILDING THE CODE (GPU)

```
$ pgcc -fast -ta=tesla:managed -Minfo=accel laplace2d_uvm.c
main:
     63, Accelerator kernel generated
         Generating Tesla code
         64, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         66, #pragma acc loop vector(128) /* threadIdx.x */
     63, Generating implicit copyin(A[:])
         Generating implicit copyout(Anew[:])
         Generating implicit copy(error)
     66, Loop is parallelizable
     74, Accelerator kernel generated
         Generating Tesla code
         75, #pragma acc loop gang /* blockIdx.x */
         77, #pragma acc loop vector(128) /* threadIdx.x */
     74, Generating implicit copyin(Anew[:])
         Generating implicit copyout(A[:])
     77, Loop is parallelizable
```

BUILDING THE CODE (MULTICORE)

```
$ pgcc -fast -ta=multicore -Minfo=accel laplace2d_uvm.c
main:
     63, Generating Multicore code
         64, #pragma acc loop gang
     64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
         Generating reduction(max:error)
     66, Loop is parallelizable
     74, Generating Multicore code
         75, #pragma acc loop gang
     75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
     77, Loop is parallelizable
```

OPENACC SPEED-UP



BUILDING THE CODE (GPU)

```
$ pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages):
Could not find allocated-variable index for symbol (laplace2d_uvm.c: 63)
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages):
Could not find allocated-variable index for symbol (laplace2d_uvm.c: 74)
main:
     63, Accelerator kernel generated
         Generating Tesla code
         63, Generating reduction(max:error)
         64, #pragma acc loop gang /* blockIdx.x */
         66, #pragma acc loop vector(128) /* threadIdx.x */
     64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
     66, Loop is parallelizable
     74, Accelerator kernel generated
         Generating Tesla code
         75, #pragma acc loop gang /* blockIdx.x */
         77, #pragma acc loop vector(128) /* threadIdx.x */
     75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
     77, Loop is parallelizable
```

OPTIMIZE DATA MOVEMENT

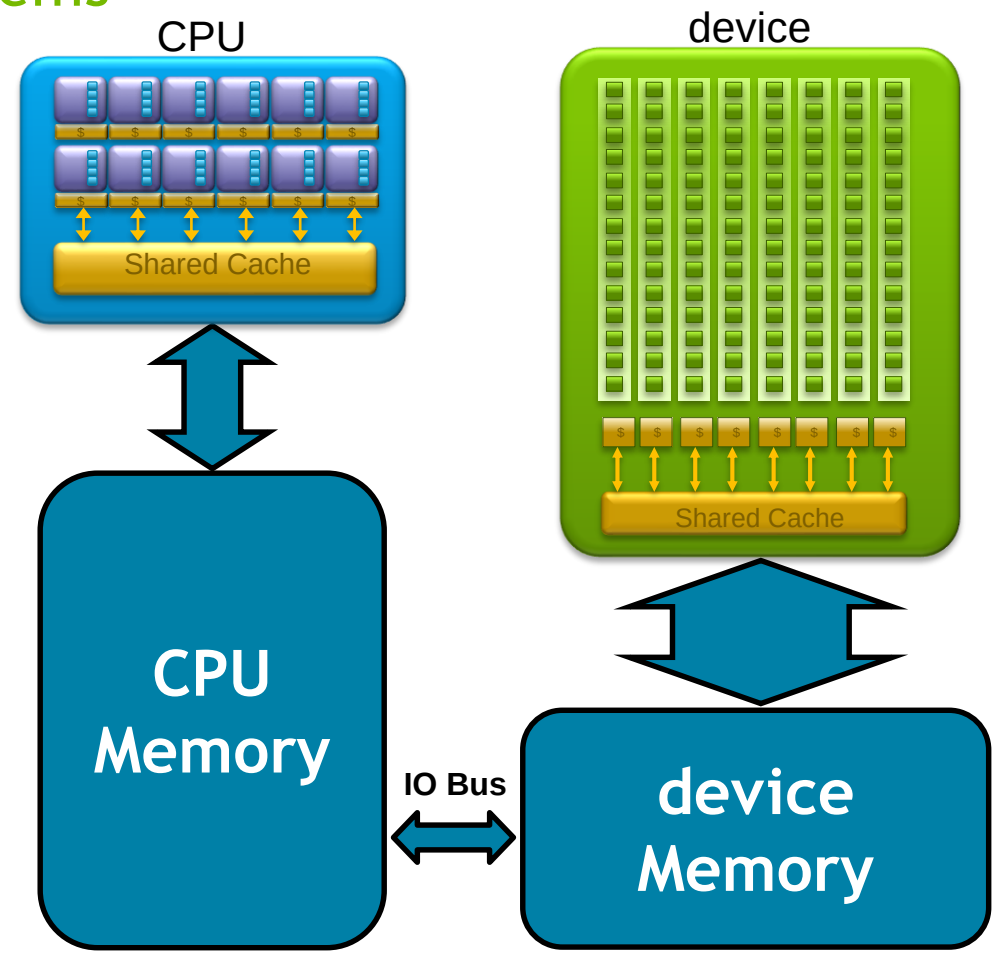
EXPLICIT MEMORY MANAGEMENT

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

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
Device allocation of a, b, c Host to device copy of a, b Device to host copy of c

OPTIMIZED DATA MOVEMENT

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

#pragma acc parallel loop reduction(max:err)
  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.

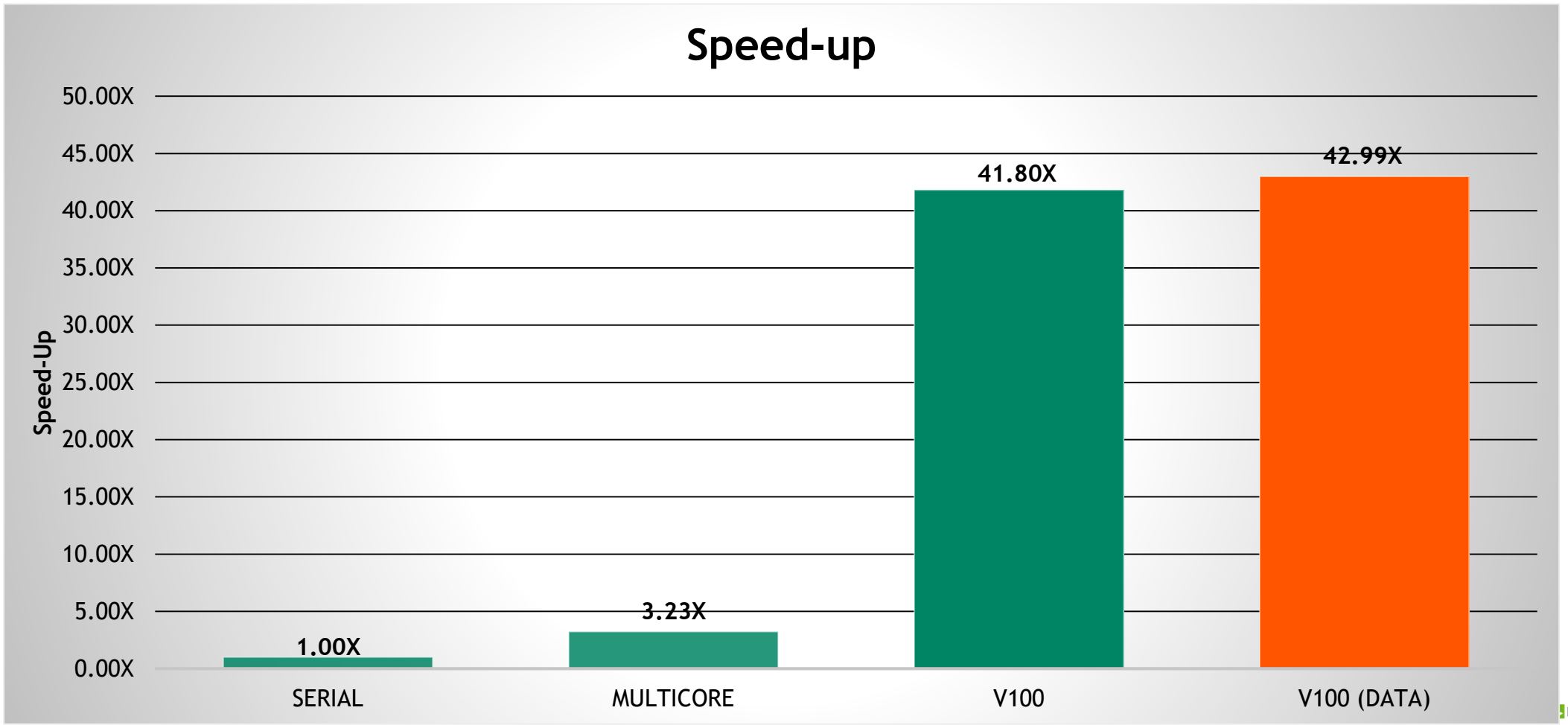
Copy initial condition of Anew, but not final value

REBUILD THE CODE

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

Now data movement only happens at our data region.

OPENACC SPEED-UP



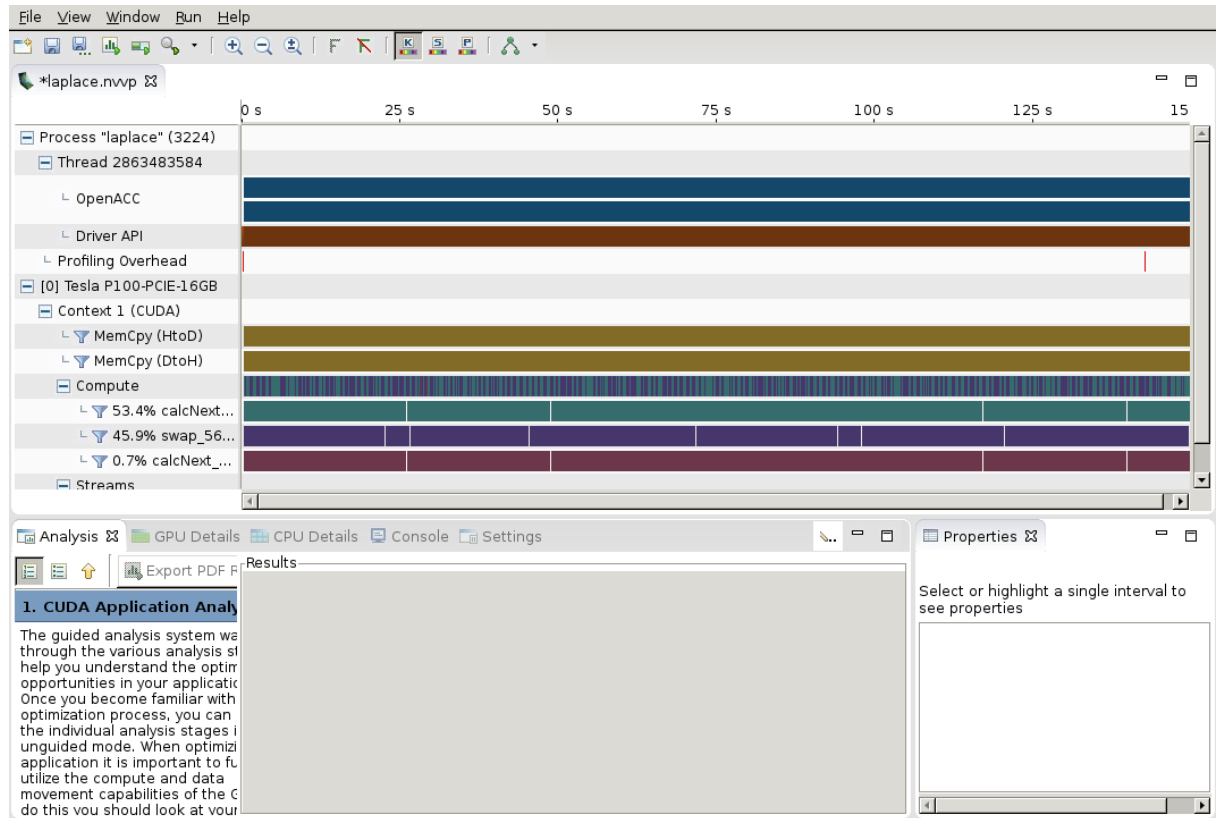
FURTHER OPTIMIZATIONS

PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

PGPROF presents far more information when running on a GPU

We can view CPU Details, GPU Details, a Timeline, and even do Analysis of the performance



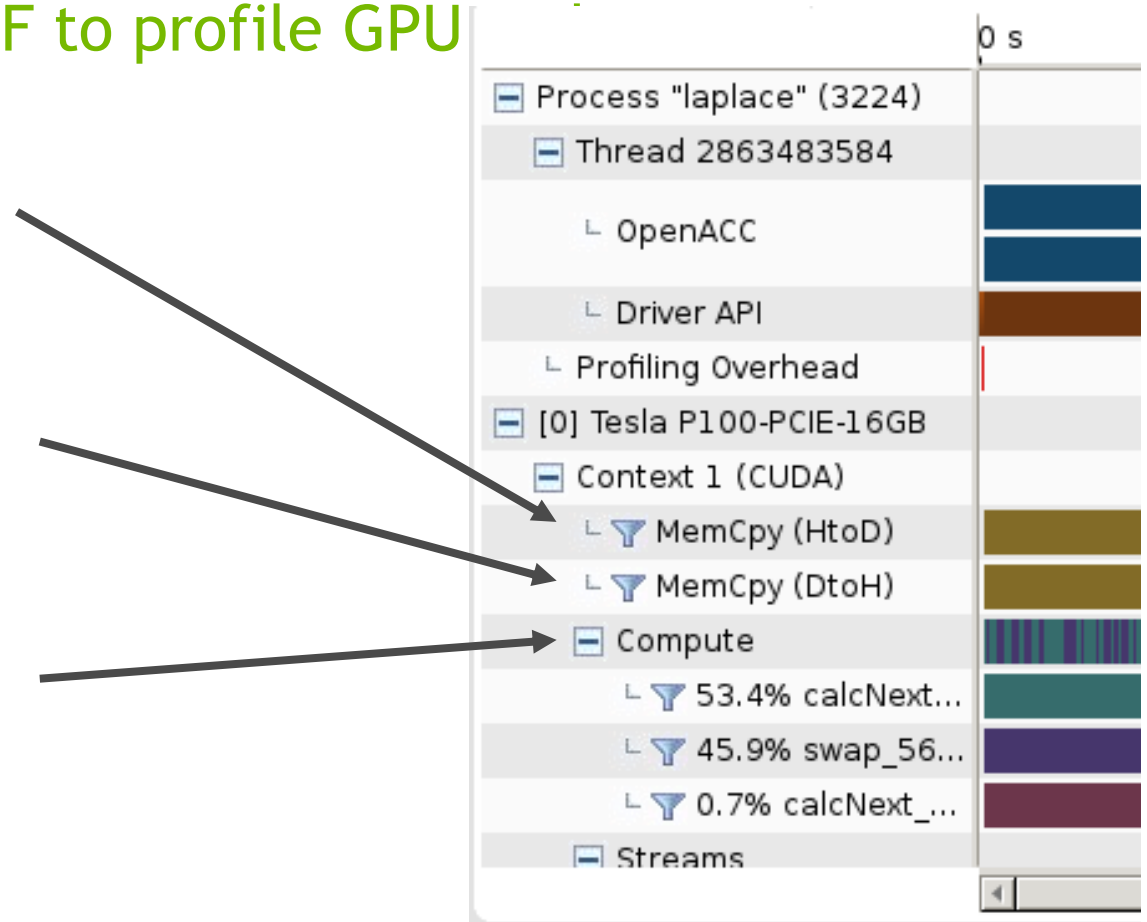
PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU

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. We can see our calcNext and swap function



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”

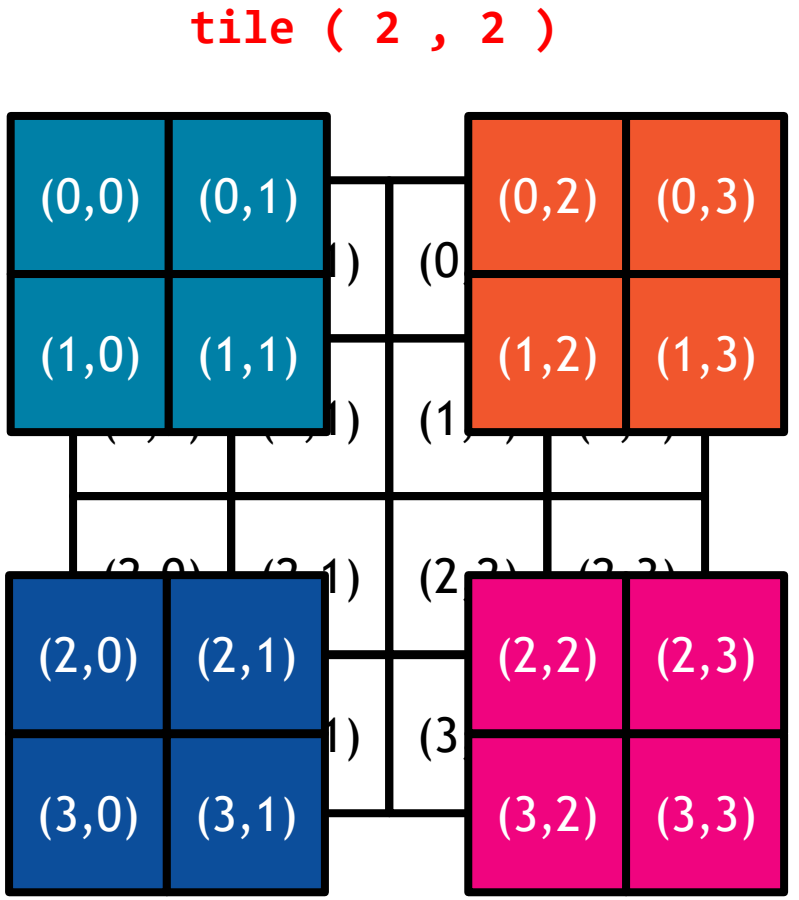
Can increase data locality in some codes

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]++;
  }
}
```

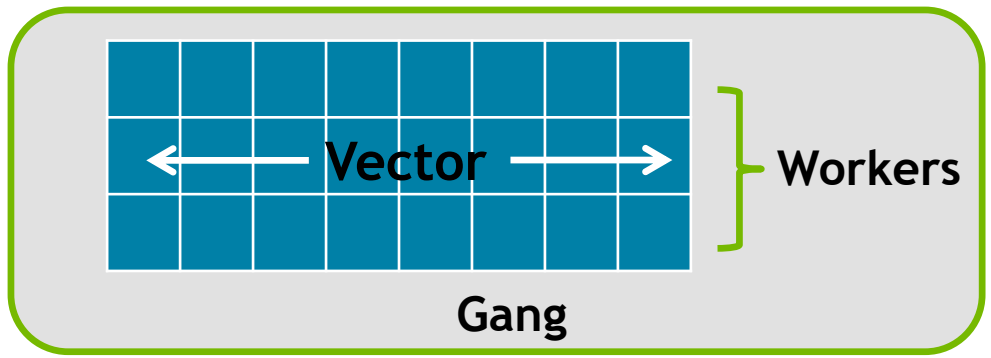


GANG WORKER VECTOR

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 fill focus a little bit on a GPU specific implementation



OPTIMIZED LOOP

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

#pragma acc parallel loop reduction(max:err) tile(32,32)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop tile(32,32)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

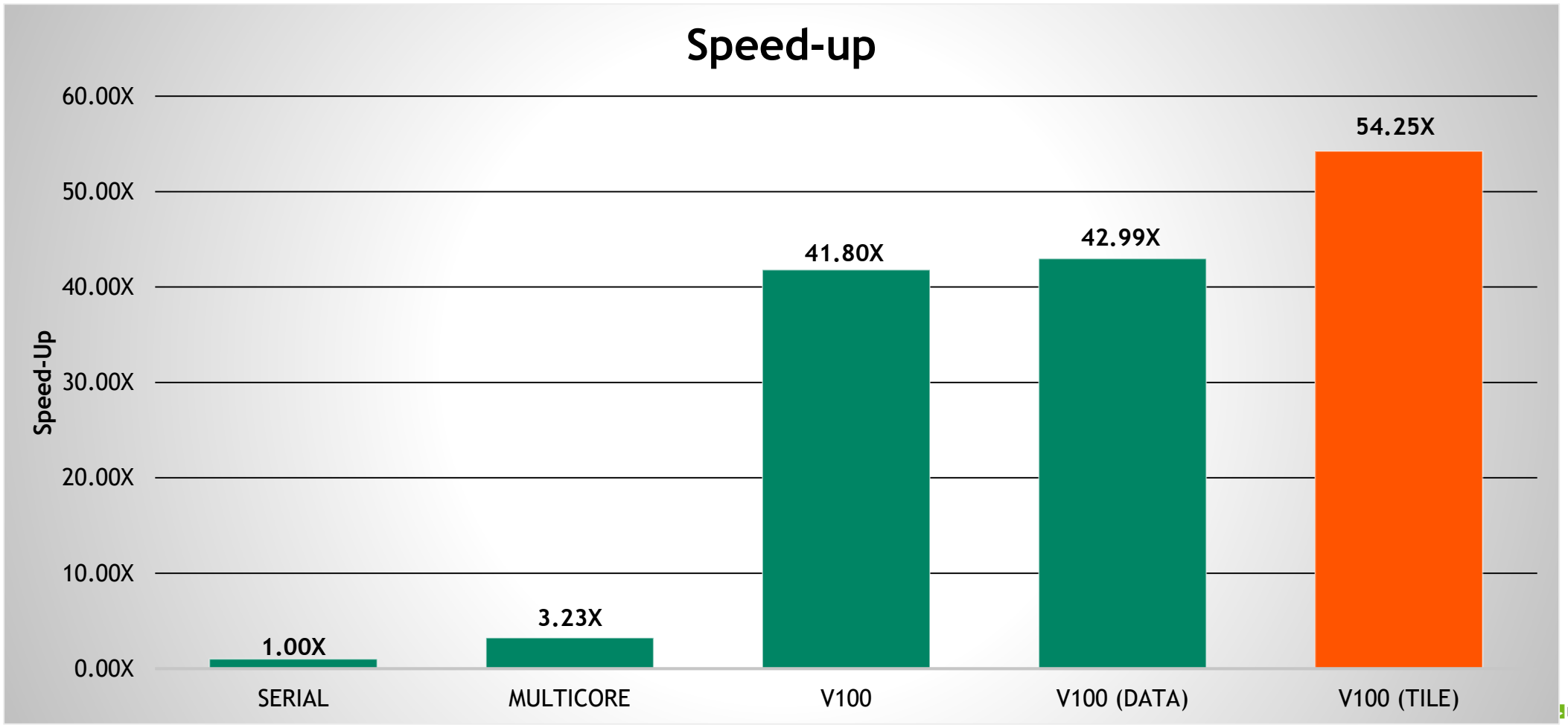
Create 32x32 tiles of the loops to better exploit data locality.

REBUILD THE CODE

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

Now data movement only happens at our data region.

OPENACC SPEED-UP



OPENACC RESOURCES

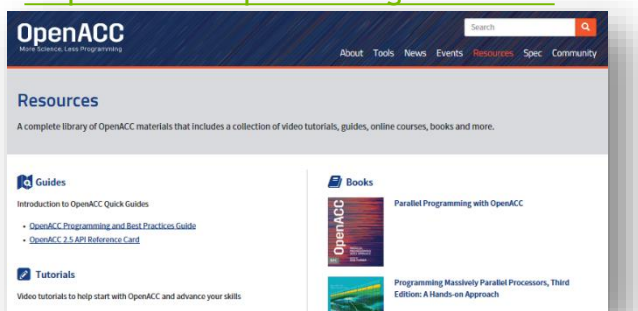
Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

FREE Compilers

PGI® Community EDITION

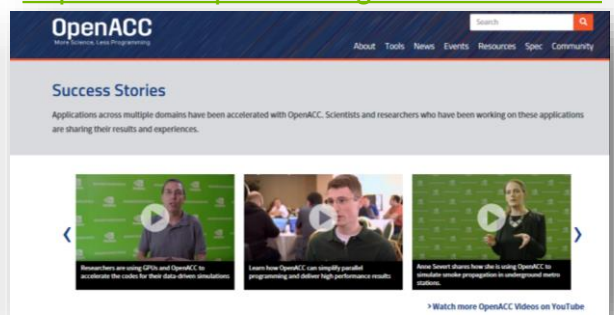
Resources

https://www.openacc.org/resources



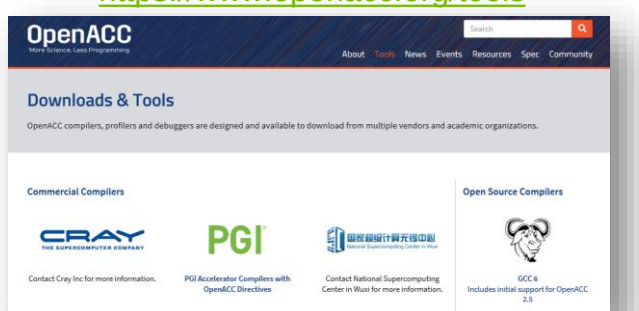
Success Stories

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



Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events

NVIDIA


NVIDIA