



DEEP  
LEARNING  
INSTITUTE

# PRACE Workshop: Deep Learning and GPU programming workshop

7 – 10 September 2020



VSB TECHNICAL  
UNIVERSITY  
OF OSTRAVA

IT4INNOVATIONS  
NATIONAL SUPERCOMPUTING  
CENTER

# MODULE TWO: PROFILING

Dr. Volker Weinberg | LRZ | 08.09.2020

# MODULE OVERVIEW

## Topics to be covered

- Compiling and profiling sequential code
- Explanation of multicore programming
- Compiling and profiling multicore code

# COMPILING SEQUENTIAL CODE

# PGI COMPILER BASICS

pgcc, pgc++ and pgfortran

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

```
$ pgcc -fast main.c  
$ pgc++ -fast main.cpp  
$ pgfortran -fast main.F90
```

# PGI COMPILER BASICS

## -Minfo flag

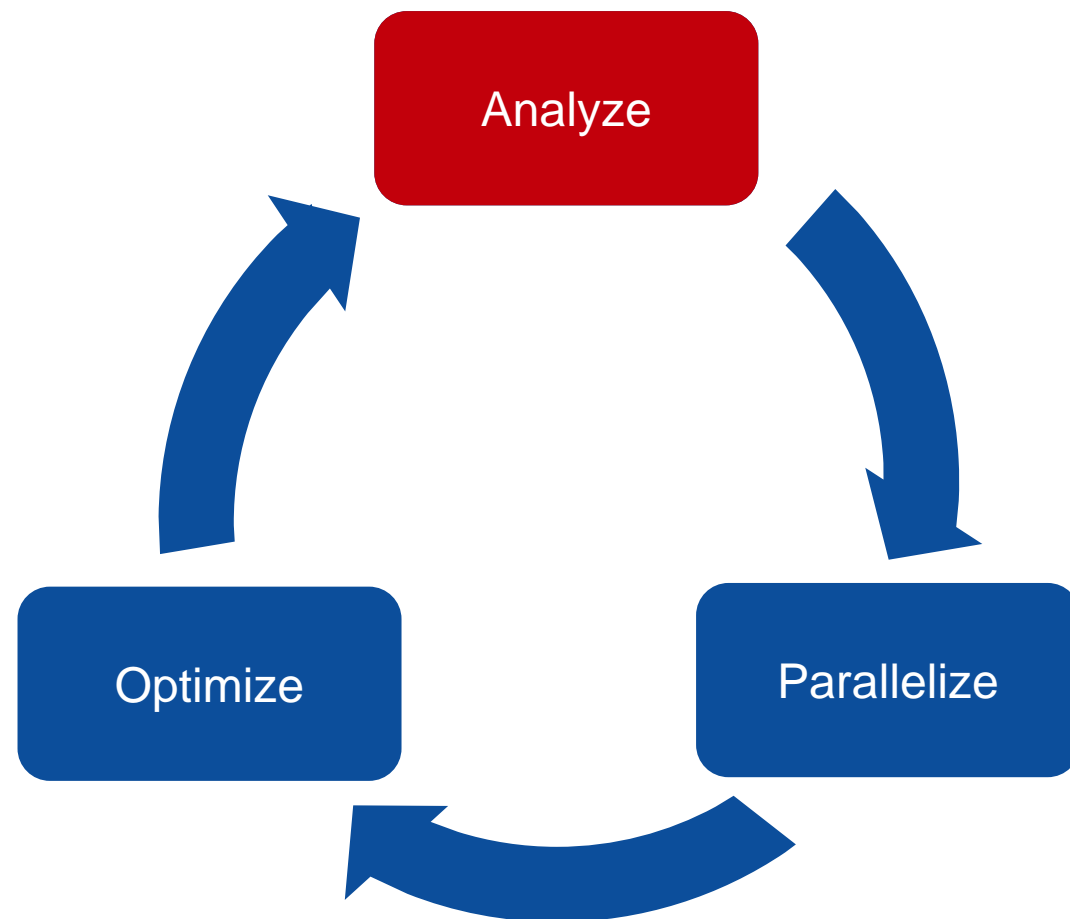
- The Minfo flag will instruct the compiler to print feedback about the compiled code
- -Minfo=accel will give us information about what parts of the code were accelerated via OpenACC
- -Minfo=opt will give information about all code optimizations
- -Minfo=all will give all code feedback, whether positive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

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

# PROFILING SEQUENTIAL CODE

# 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, check for correctness and then analyze it again.
- **Optimize** your code to improve observed speed-up from parallelization.





# PROFILING SEQUENTIAL CODE

## Step 1: Run Your Code

Record the time it takes for your sequential program to run.

Note the final results to verify correctness later.

Always run a problem that is representative of your real jobs.

## Terminal Window

```
$ pgcc -fast jacobi.c laplace2d.c
$ ./a.out
  0, 0.250000
 100, 0.002397
 200, 0.001204
 300, 0.000804
 400, 0.000603
 500, 0.000483
 600, 0.000403
 700, 0.000345
 800, 0.000302
 900, 0.000269
total: 39.432648 s
```

# PROFILING SEQUENTIAL CODE

## Step 2: 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.

```
! pgprof ./laplace
```

```
Jacobi relaxation Calculation: 4096 x 4096 mesh
0, 0.250000
100, 0.002397
200, 0.001204
300, 0.000804
400, 0.000603
500, 0.000483
600, 0.000403
700, 0.000345
800, 0.000302
900, 0.000269
total: 60.611229 s
```

```
===== CPU profiling result (bottom up):
```

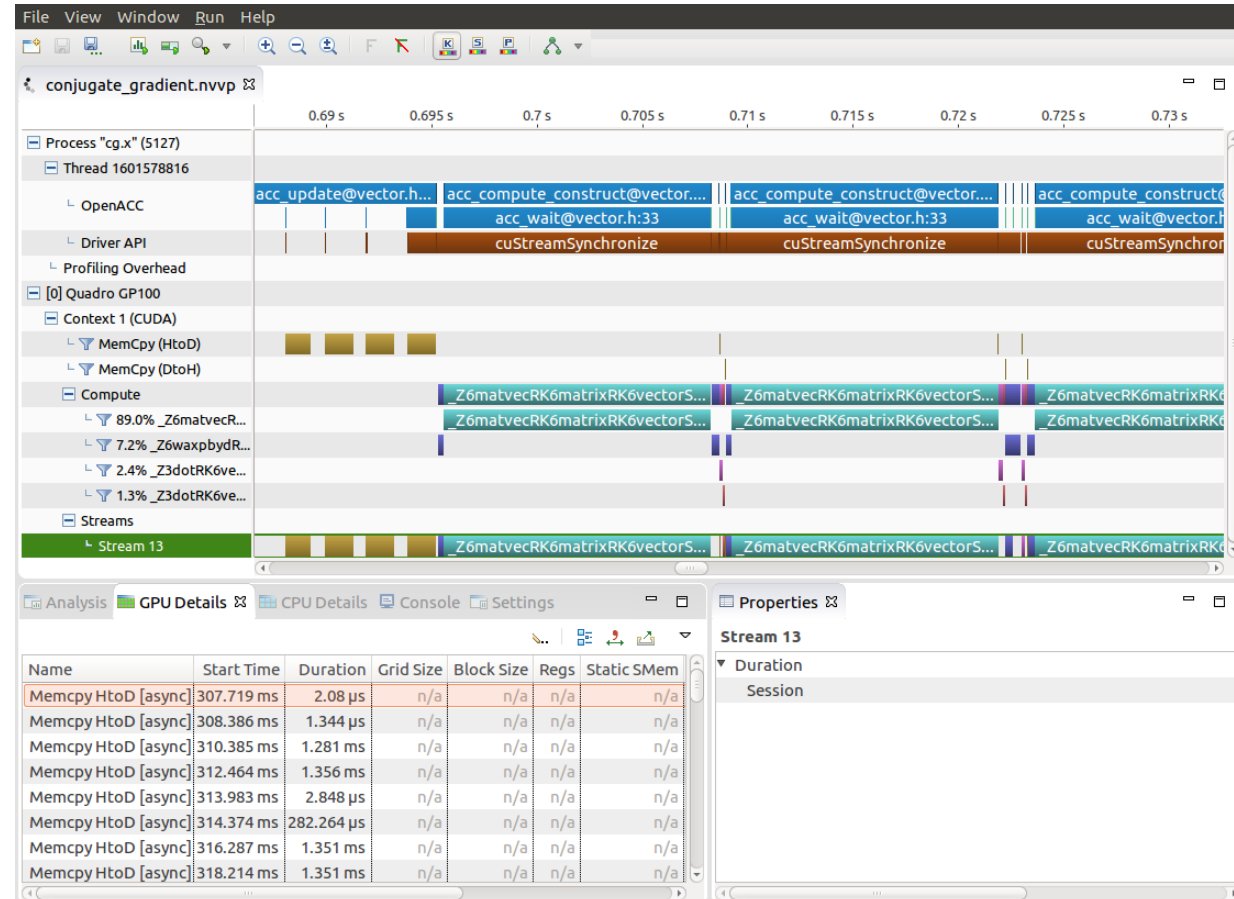
Time(%)	Time	Name
54.51%	32.43s	calcNext
54.51%	32.43s	main
45.40%	27.01s	__c_mcopy8_avx
0.05%	30ms	swap
0.05%	30ms	main
0.03%	20ms	__c_mcopy8

```
===== Data collected at 100Hz frequency
```

# PROFILING SEQUENTIAL CODE

## Introduction to PGProf

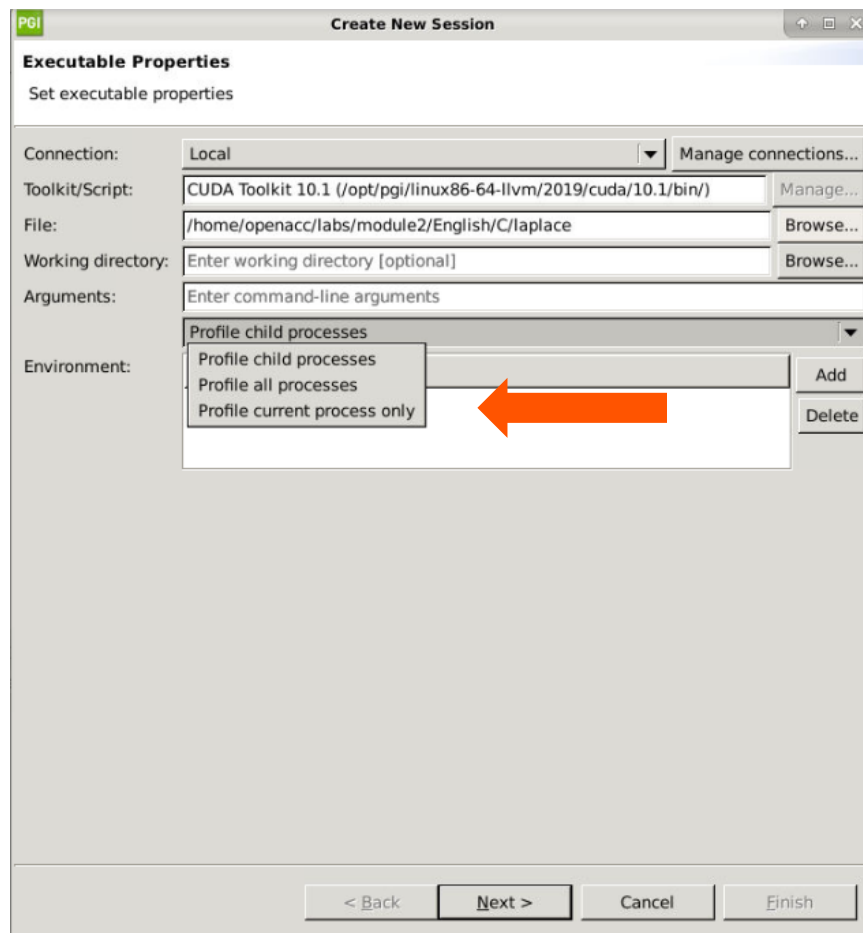
- Gives visual feedback of how the code ran
- Gives numbers and statistics, such as program runtime
- Also gives runtime information for individual functions/loops within the code
- Includes many extra features for profiling parallel code



# PROFILING SEQUENTIAL CODE

## CPU Details

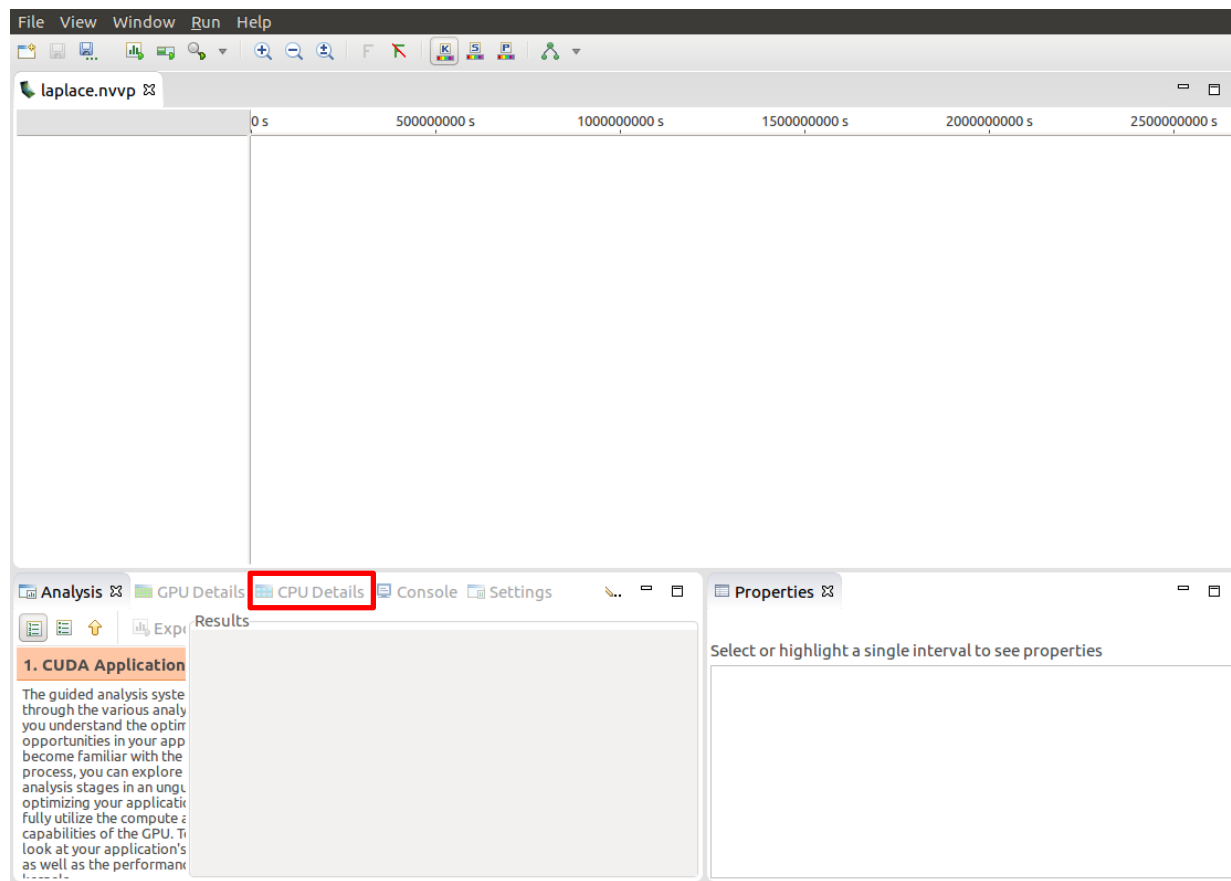
- Please make sure to choose "Profile current process only" from the dropdown instead of "Profile child process". The CPU details will not be displayed in the profiler otherwise.



# PROFILING SEQUENTIAL CODE

## First sight when using PGPROF

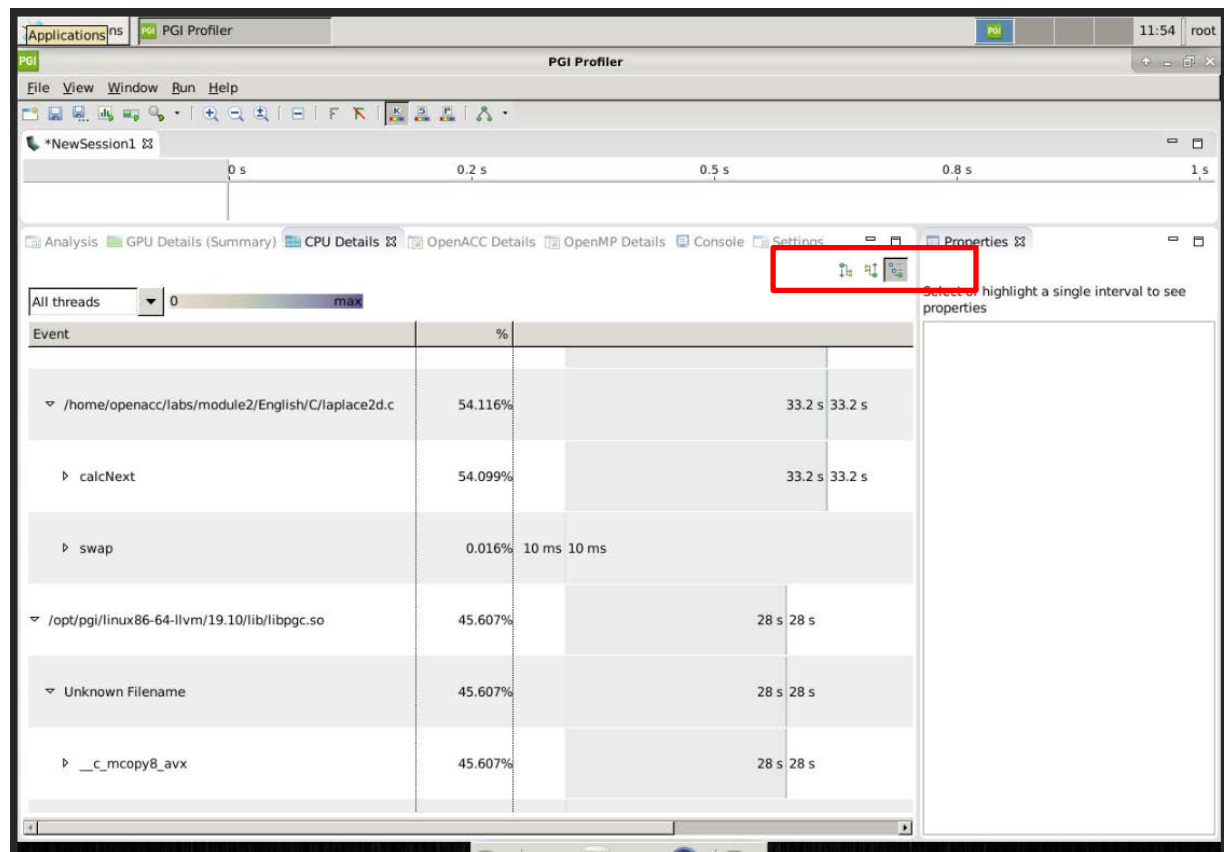
- Profiling a simple, sequential code
- Our sequential program will run on the CPU
- To view information about how our code ran, we should select the “CPU Details” tab



# PROFILING SEQUENTIAL CODE

## CPU Details

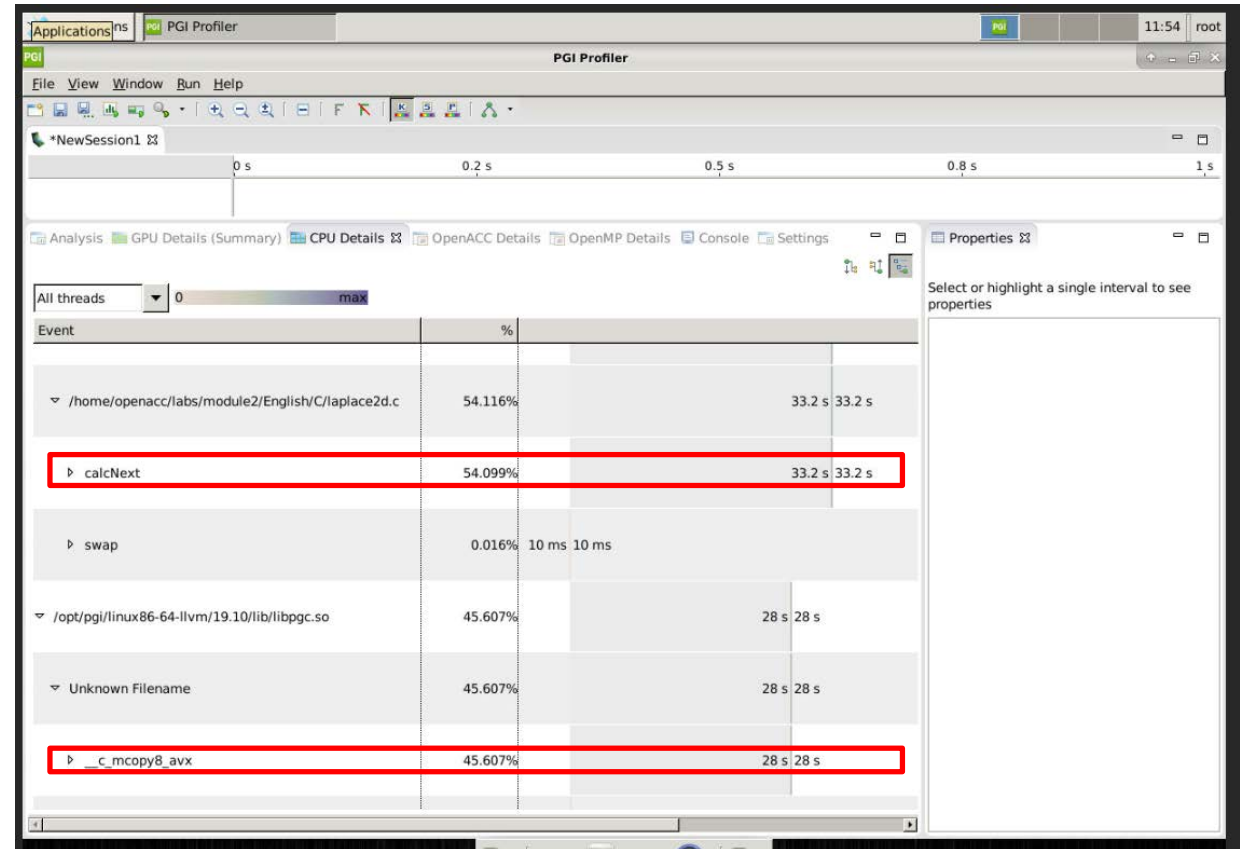
- Within the “CPU Details” tab, we can see the various parts of our code, and how long they took to run
- We can reorganize this info using the three options in the top-right portion of the tab
- We will expand this information, and see more details about our code



# PROFILING SEQUENTIAL CODE

## CPU Details

- We can see that there are two places that our code is spending most of its time
- 33.2 seconds in the “calcNext” function
- 28 seconds in a memcpy function
- The `c_memcpy8_avx` that we see is actually a compiler optimization that is being applied to our “swap” function



# PROFILING SEQUENTIAL CODE

## PGPROF

- We are also able to select the different elements in the CPU Details by double-clicking to open the associated source code
- Here we have selected the “calcNext” element, which opened up the source file in the top part of the window

The screenshot shows the PGI Profiler application interface. The top pane displays the source code for a file named 'laplace2d.c'. The code includes headers for math and stdlib, defines an OFFSET macro, and contains two functions: 'initialize' and 'calcNext'. The 'calcNext' function is highlighted in the CPU Details table below.

Event	%	Time	Count
calcNext	53.667%	33.2 s	33.2
__c_mcopy8_avx	45.994%	28.5 s	28.5 s

The bottom pane shows the CPU Details table with a color scale from 0 to max. The 'calcNext' event is selected, and its source code is displayed in the top pane. The 'Properties' pane on the right is empty, with the instruction 'Select or highlight a single interval to see properties'.



# PROFILING SEQUENTIAL CODE

## Step 2: Profile Your Code

Obtain detailed information about how the code ran.

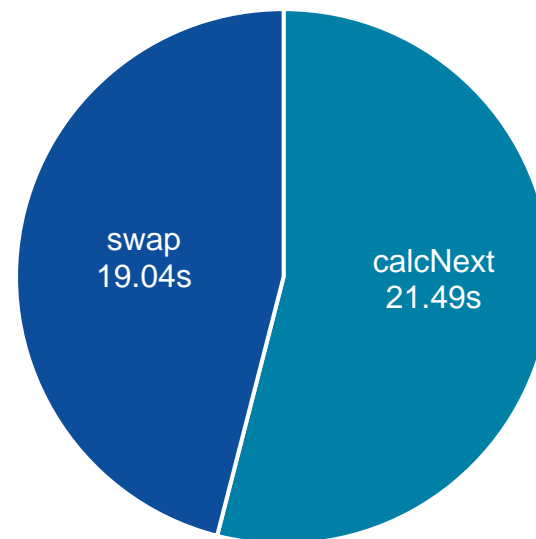
This can include information such as:

- 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

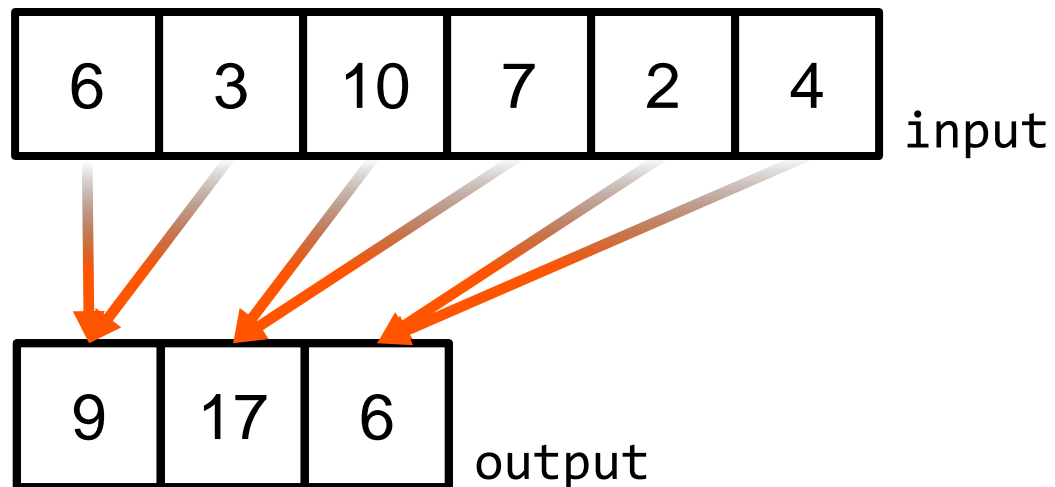
## Step 3: Identify Parallelism

Observe the loops contained within the identified hotspots

Are these loops parallelizable?  
Can the loop iterations execute independently of each other?  
Are the loops multi-dimensional, and does that make them very large?

Loops that are good to parallelize tend to have a lot of iterations to map to parallel hardware.

```
void pairing(int *input, int *output, int N){  
    for(int i = 0; i < N; i++){  
        output[i] = input[i*2] + input[i*2+1];  
    }  
}
```



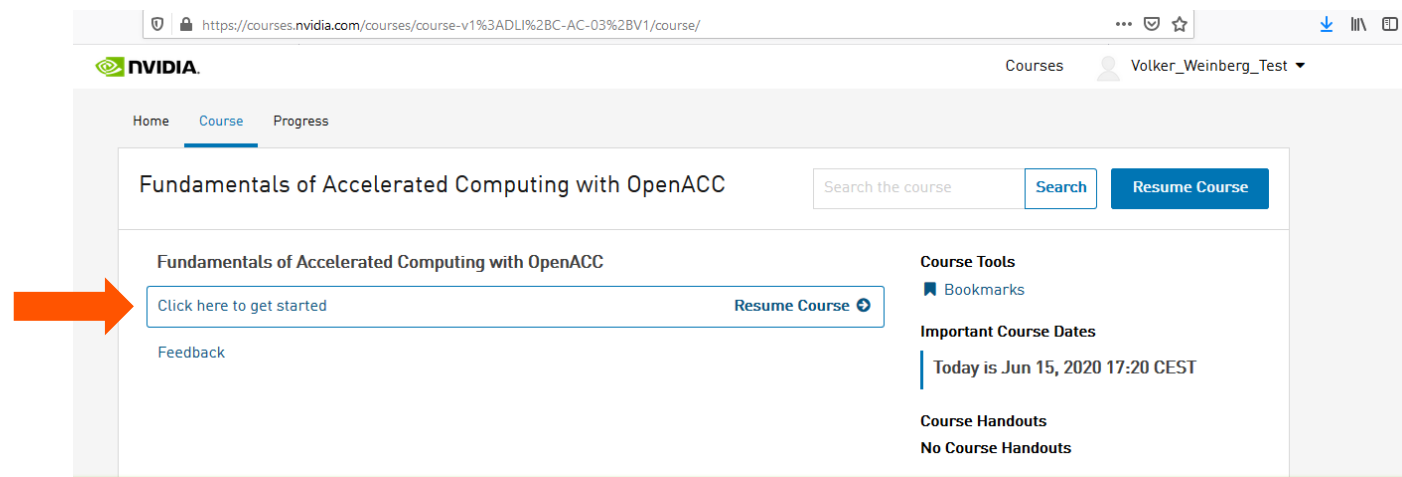
PLEASE START LAB NOW!

# TRAINING SETUP

- To get started, follow these steps:
- Create an NVIDIA Developer account at <http://courses.nvidia.com/join> Select "Log in with my NVIDIA Account" and then "Create Account" (done yesterday)
- Visit <http://courses.nvidia.com/dli-event> and enter the event code

**PRACE\_OACC\_AMBASSADOR\_SE20**

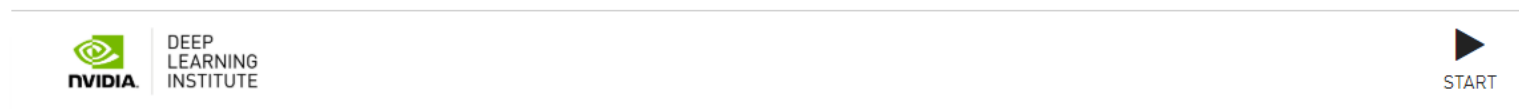
# TRAINING SETUP



The screenshot shows the NVIDIA course page for "Fundamentals of Accelerated Computing with OpenACC". The page includes a search bar, a "Resume Course" button, and a "Click here to get started" button. An orange arrow points to the "Click here to get started" button.

## Fundamentals of Accelerated Computing with OpenACC

[Bookmark this page](#)



The block contains the NVIDIA Deep Learning Institute logo on the left and a "START" button with a play icon on the right. An orange arrow points to the "START" button.

To get started with this live GPU enabled interactive content please click the "Start" button on the top right of this block.

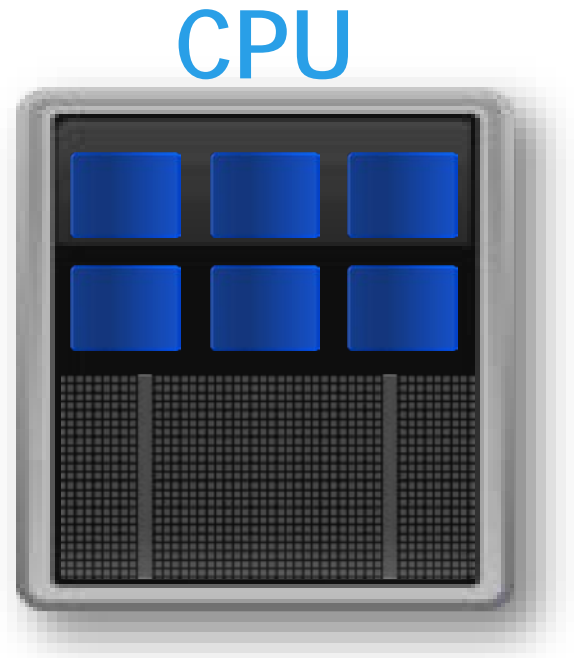
This will launch a pre-configured GPU workstation, it may take 5-10 minutes.

# PROFILING MULTICORE CODE

# PROFILING MULTICORE CODE

## What is multicore?

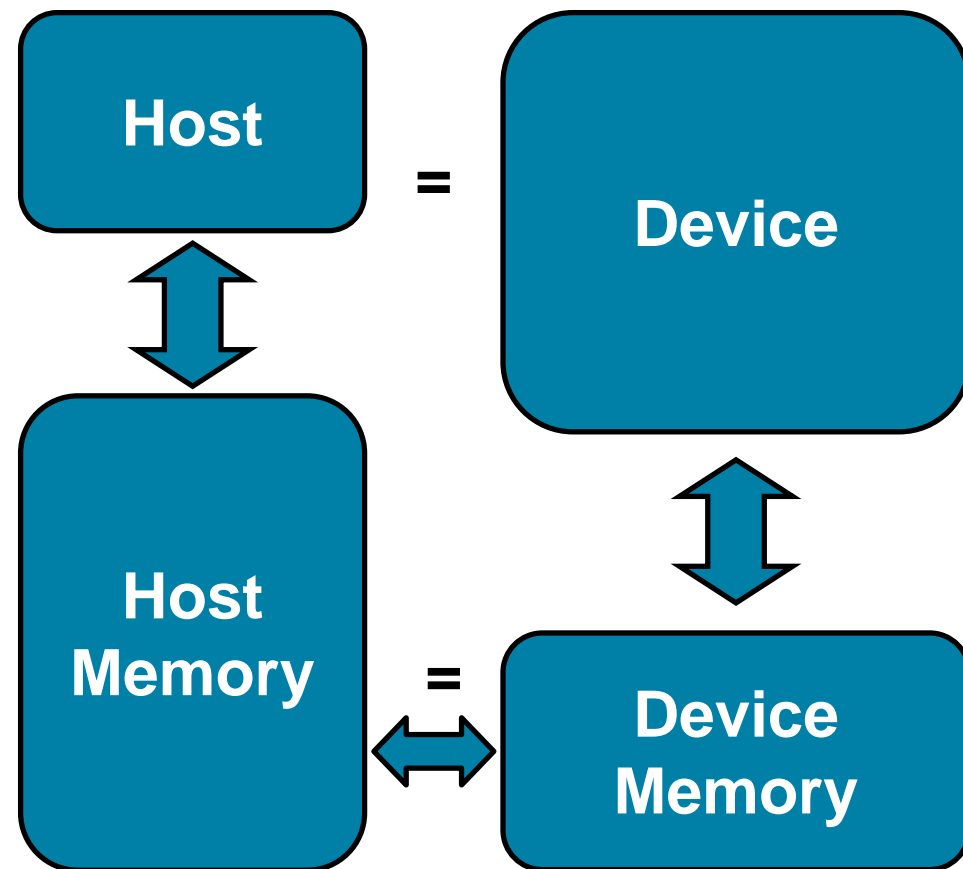
- *Multicore* refers to using a CPU with multiple computational cores as our parallel device
- These cores can run independently of each other, but have shared access to memory
- Loop iterations can be spread across CPU threads and can utilize SIMD/vector instructions (SSE, AVX, etc.)
- Parallelizing on a multicore CPU is a good starting place, since data management is unnecessary



# PROFILING MULTICORE CODE

## Using a multicore CPU with OpenACC

- OpenACC's generic model involves a combination of a host and a device
- Host generally means a CPU, and the device is some parallel hardware
- When running with a multicore CPU as our device, typically this means that our host/device will be the same
- This also means that their memories will be the same





# PROFILING MULTICORE CODE

## Compiling code for a specific parallel hardware

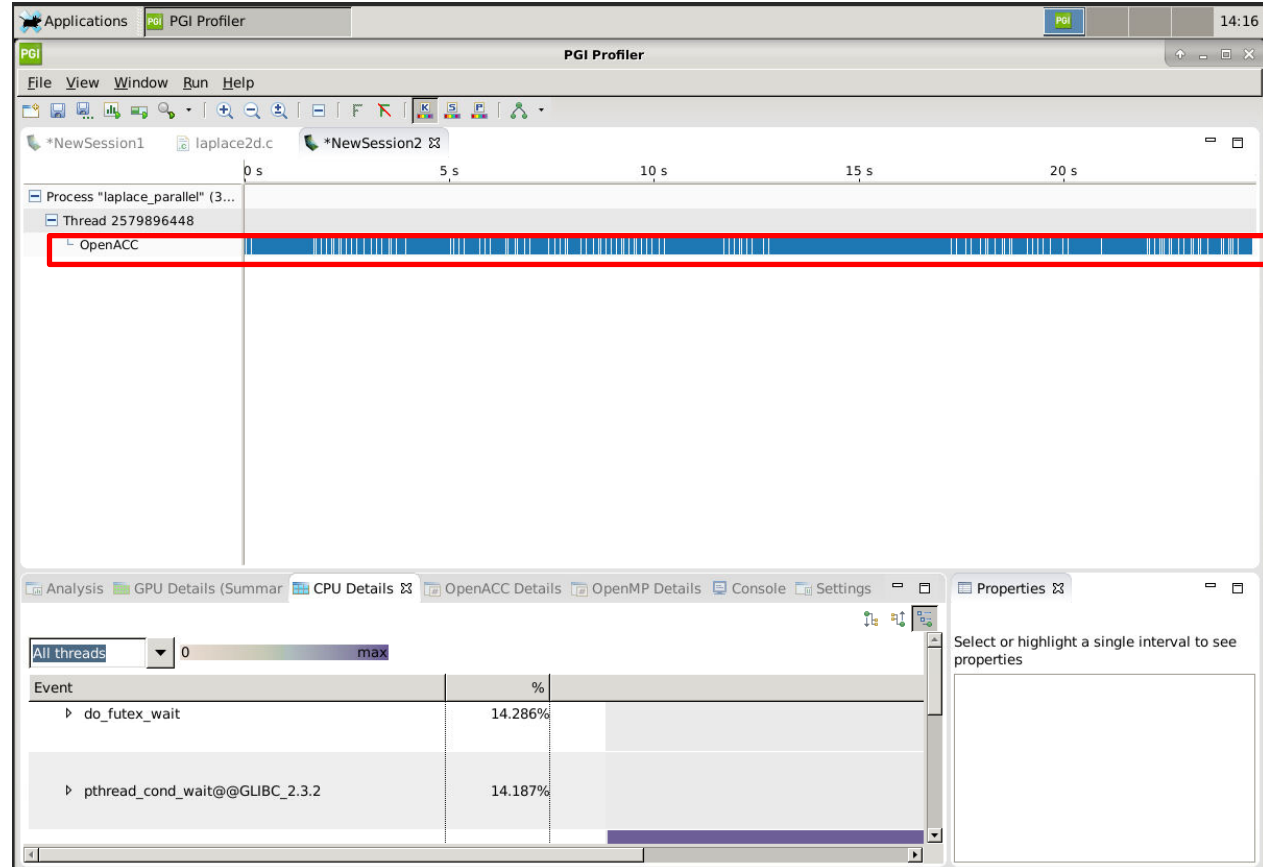
- The '-ta' flag will allow us to compile our code for a specific, target parallel hardware
- 'ta' stands for "Target Accelerator," an accelerator being another way to refer to a parallel hardware
- Our OpenACC code can be compiled for many different kinds of parallel hardware without having to change the code

```
$ pgcc -fast -Minfo=accel -ta=multicore laplace2d.c  
calcNext:  
    35, Generating Multicore code  
    36, #pragma acc loop gang
```

# PROFILING MULTICORE CODE

## PGPROF

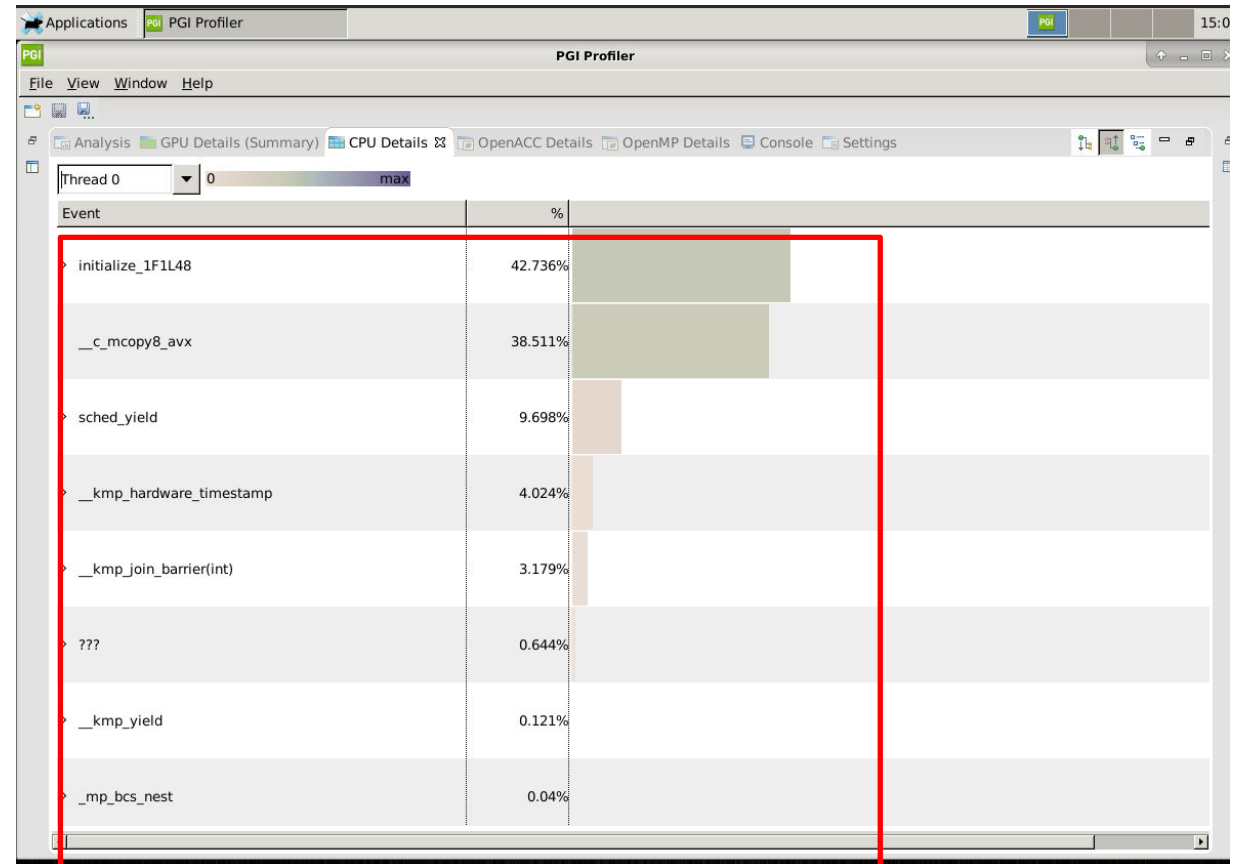
- The first difference we see in this multicore profile is that there is now a “timeline”
- This timeline will show when our parallel hardware is being used, and how it is being used
- Each of the blue bars represent a portion of our program that was run on the multicore CPU



# PROFILING MULTICORE CODE

## CPU Details

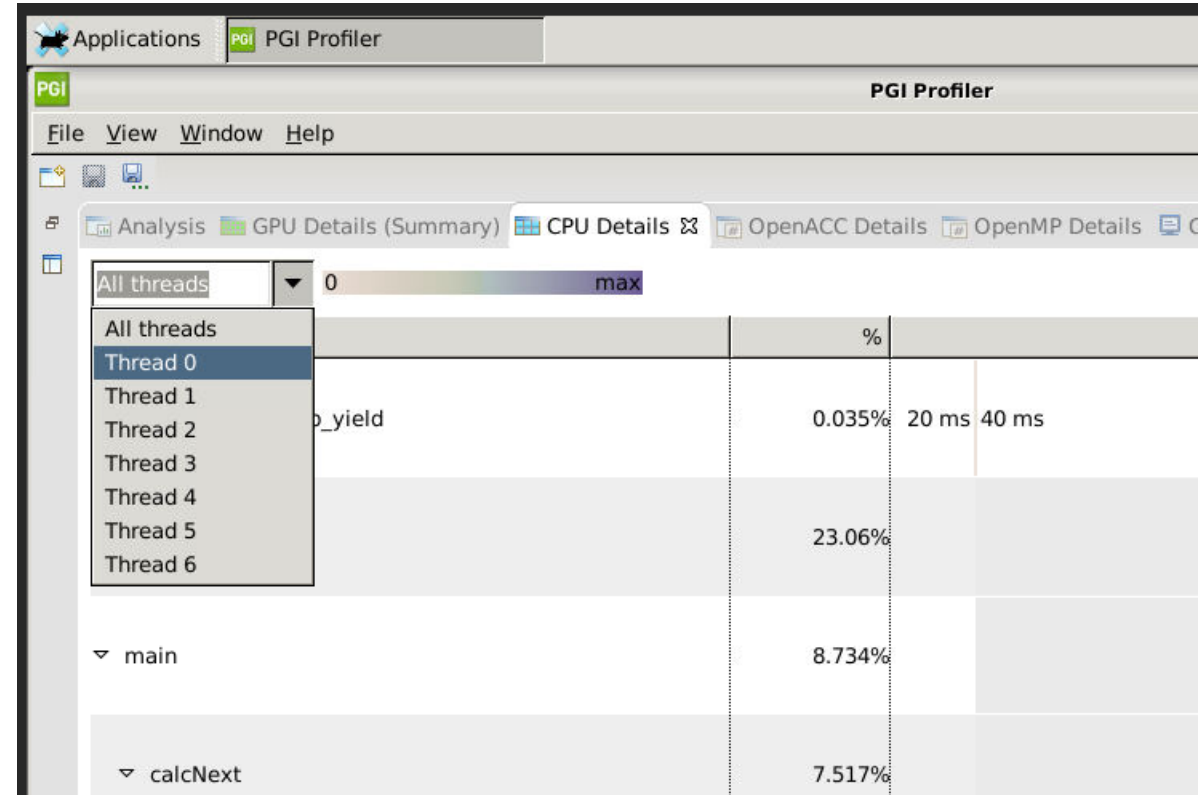
- Looking at our CPU Details, we can see that there is a lot more happening compared to our sequential program
- For the most part, these extra details revolve around the need for the CPU cores to communicate with each other



# PROFILING MULTICORE CODE

## View of computational threads

- You can see statistics for all threads or select a specific thread in the box on the top left of the CPU Details tab.



# PROFILING MULTICORE CODE

## View of all computational threads

- When moving the mouse on the % value, one can see
  - Mean across all threads
  - Total across all threads
  - Total as a percentage of all the time spent on one / all threads.

19.3%

Function: calcNext  
Mean across all threads: 13 s  
Total across all threads: 13 s  
Percentage of all the time spent on Thread 0: 52.193%

51.1%

Function: \_\_c\_mcopy8\_avx  
Mean across all threads: 9.9 s  
Total across all threads: 39.8 s  
Percentage of all the time spent on Thread 0: 38.511%

# PROFILING MULTICORE CODE

## OpenACC Details

Summary of OpenACC events on process: 384

Name	%	Time	Calls
laplace2d.c:36	52.244%	9.74278 s	1000
acc_compute_construct	52.244%	9.74278 s	1000
laplace2d.c:52	47.334%	8.82718 s	1000
acc_compute_construct	47.334%	8.82718 s	1000

Properties for acc\_compute\_construct@laplace2d...:

Start	5.62979 s
End	5.6403 s
Duration	10.51828
Event Kind	acc_com
Parent Construct	unknown
Version	201711
Implicit	false
Device Type	acc_devic

Summary of OpenACC events on process: 384

Name	%	Time	Calls
laplace2d.c:36	52.244%	9.74278 s	1000
acc_compute_construct	52.244%	9.74278 s	1000
laplace2d.c:52	47.334%	8.82718 s	1000

Properties for acc\_compute\_construct@laplace2d...:

Start	5.62979 s
End	5.6403 s
Duration	10.51828
Event Kind	acc_com
Parent Construct	unknown
Version	201711
Implicit	false
Device Type	acc_devic

# LAPLACE HEAT TRANSFER

## Introduction to lab code - visual

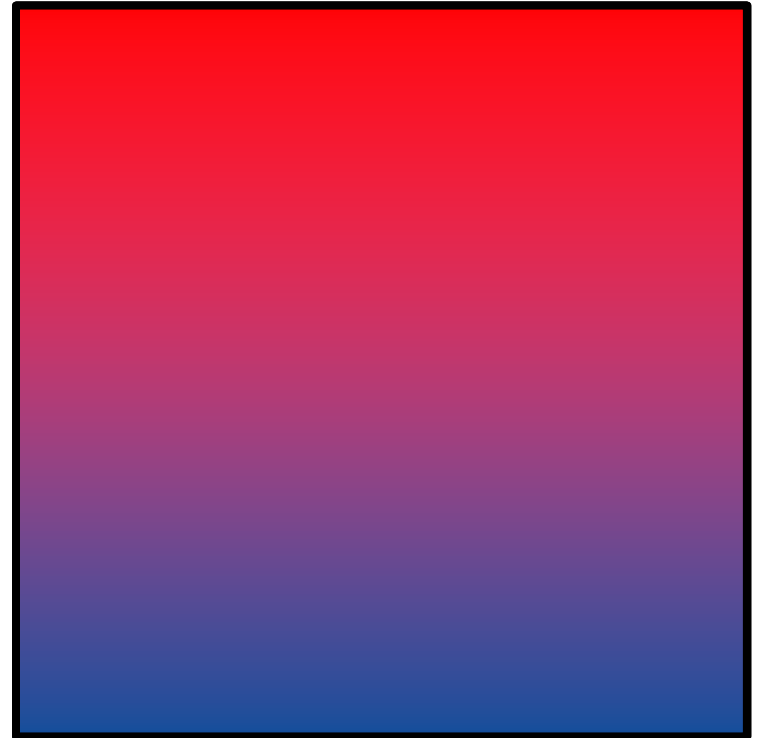
We will observe a simple simulation of heat distributing across a metal plate.

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

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

Very Hot

Room Temp



# LAPLACE HEAT TRANSFER

## Introduction to lab code - technical

The lab simulates a very basic 2-dimensional heat transfer problem. We have two 2-dimensional arrays, **A** and **Anew**.

The arrays represent a 2-dimensional, metal plate. Each element in the array is a **double** value that represents temperature.

We will simulate the distribution of heat until a **minimum change value** is achieved, or until we exceed a **maximum number of iterations**.

A

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Anew

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



# LAPLACE HEAT TRANSFER

## Introduction to lab code - technical

We initialize the top row to be a temperature of 1.0

The **calcNext** function will iterate through all of the inner elements of array A, and update the corresponding elements in Anew

We will take the average of the neighboring cells, and record it in **Anew**.

The **swap** function will copy the contents of Anew to A

A

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



Anew

0.0	0.0	0.0	0.0
0.0	0.25	0.25	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

# LAPLACE HEAT TRANSFER

## Introduction to lab code

A

1.0	1.0	1.0	1.0
0.0	0.25	0.25	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



Anew

1.0	1.0	1.0	1.0
0.0	0.25	0.25	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

The **swap** function will copy the contents of Anew to A

# KEY CONCEPTS

In this module we discussed...

- Compiling sequential and parallel code
- CPU profiling for sequential and parallel execution
- Specifics of our Laplace Heat Transfer lab code

# LAB GOALS

In this lab you will do the following...

- Build and run the example code using the PGI compiler
- Use PGProf to understand where the program spends its time

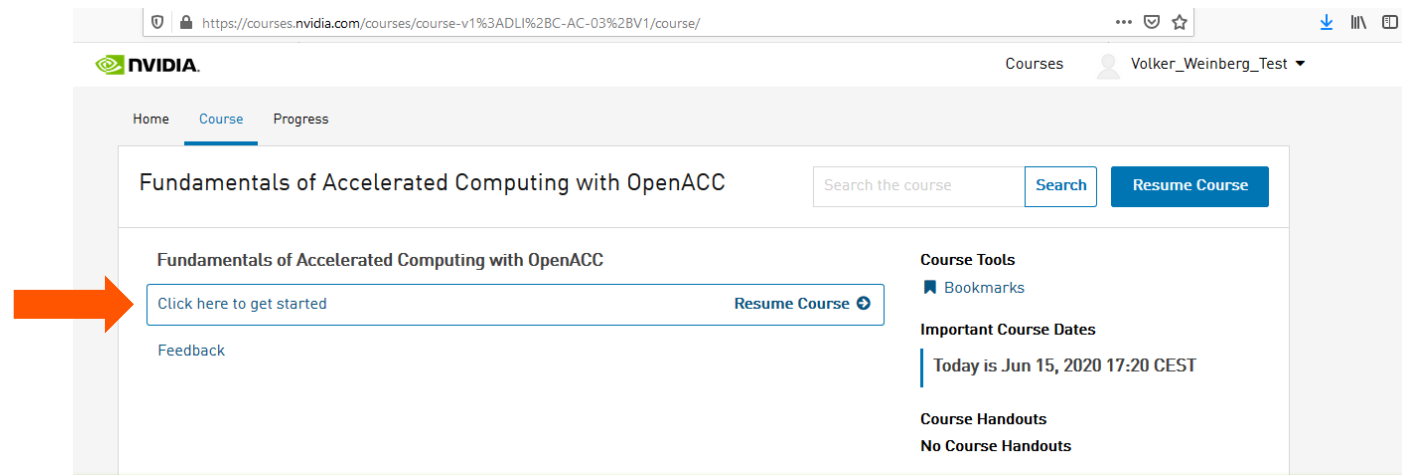
# THANK YOU

# TRAINING SETUP

- To get started, follow these steps:
- Create an NVIDIA Developer account at <http://courses.nvidia.com/join> Select "Log in with my NVIDIA Account" and then "Create Account" (done yesterday)
- Visit <http://courses.nvidia.com/dli-event> and enter the event code

**PRACE\_OACC\_AMBASSADOR\_SE20**



# TRAINING SETUP



The screenshot shows the NVIDIA course page for "Fundamentals of Accelerated Computing with OpenACC". The page includes a search bar, a "Resume Course" button, and a "Click here to get started" button. An orange arrow points to the "Click here to get started" button.

## Fundamentals of Accelerated Computing with OpenACC

[Bookmark this page](#)



The footer of the page features the NVIDIA and Deep Learning Institute logos on the left and a "START" button with a play icon on the right. An orange arrow points to the "START" button.

To get started with this live GPU enabled interactive content please click the "Start" button on the top right of this block.

This will launch a pre-configured GPU workstation, it may take 5-10 minutes.

# TRAINING SETUP

## Fundamentals of Accelerated Computing with OpenACC

[Bookmark this page](#)



DEEP  
LEARNING  
INSTITUTE

1 : 56 : 47

REMAINING TIME



LAUNCH TASK



STOP TASK

Please wait 5 - 10 minutes while your interactive GPU enabled environment loads. When the "Launch" button appears, click it to get started.



# TRAINING SETUP

Files Running Clusters

Select items to perform actions on them. Upload New ▾ ↻

<input type="checkbox"/> 0 ▾	📁 /	Name ▾	Last Modified	File size
<input type="checkbox"/>	📁 module2		vor 4 Tagen	
<input type="checkbox"/>	📁 module3		vor 4 Tagen	
<input type="checkbox"/>	📁 module4		vor 4 Tagen	
<input type="checkbox"/>	📁 module5		vor 4 Tagen	
<input type="checkbox"/>	📁 module6		vor 4 Tagen	
<input type="checkbox"/>	📄 START HERE.ipynb		vor 4 Tagen	1.36 kB



# TRAINING SETUP

## Welcome to the OpenACC labs

Please select the appropriate lab below.



- [Module 2](#) - Application Profiling with PGProf Lab - This lab introduces students to application profiling using the PGProf profiler.
- [Module 3](#) - OpenACC Directives Basics - This lab introduces OpenACC directives.
- [Module 4](#) - GPU Programming with OpenACC - This lab introduces GPU programming with OpenACC.
- [Module 5](#) - Data Management with OpenACC - This lab introduces OpenACC data management directives.
- [Module 6](#) - OpenACC Loop Optimizations - This lab introduces students to loop optimizations in OpenACC.

## Application Profiling with PGProf Lab

This lab is meant to accompany Module 2 of the OpenACC.org teaching materials. The purpose of this lab is to introduce students to application profiling using the PGProf profiler. Lab instructions and source code is available for C/C++ and Fortran.

Please see the following files to begin the lab:



- [C/C++](#)
- [Fortran](#)

# PROFILING SEQUENTIAL CODE

## CPU Details

- Please make sure to choose "Profile current process only" from the dropdown instead of "Profile child process". The CPU details will not be displayed in the profiler otherwise.

