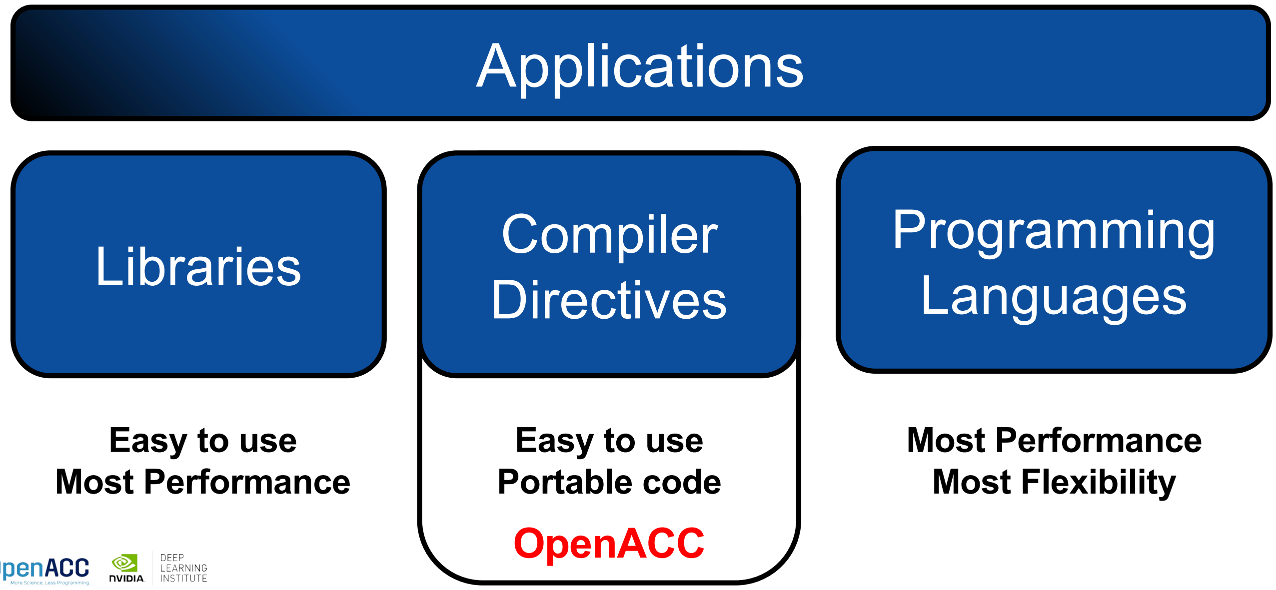
Module I: Introduction to OpenACC

Similarly to **OpenMP**, **OpenACC** is a directives-based programming approach to **parallel computing** but 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



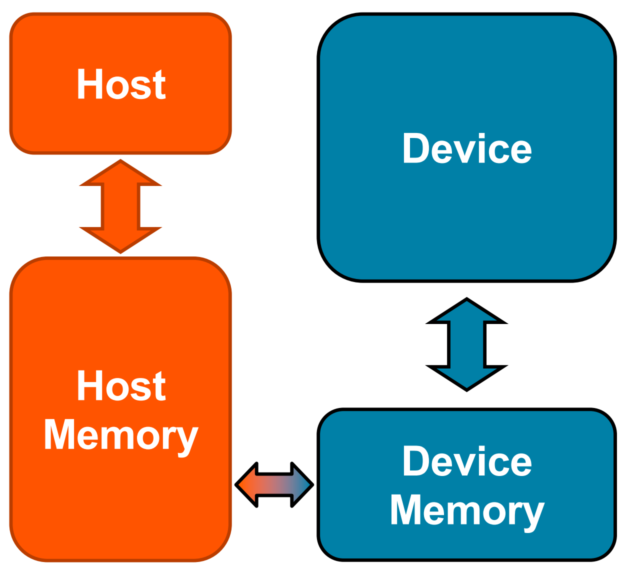
OPENACC PORTABILITY

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

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 NVIDIA DEEP LEARNING INSTITUTE

OPENACC

Incremental

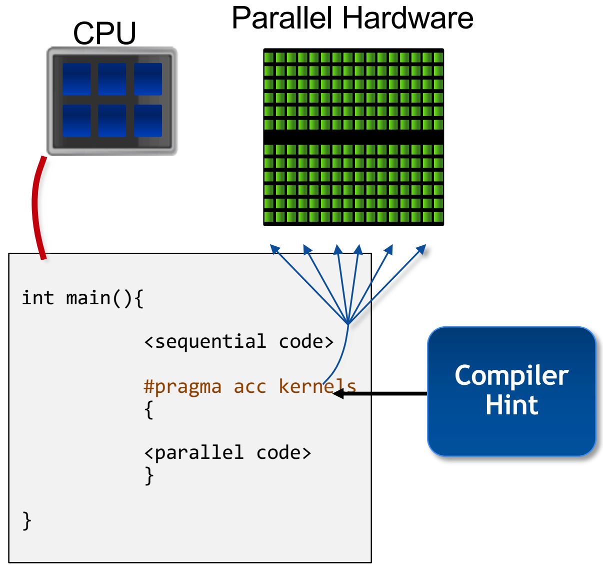
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

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.

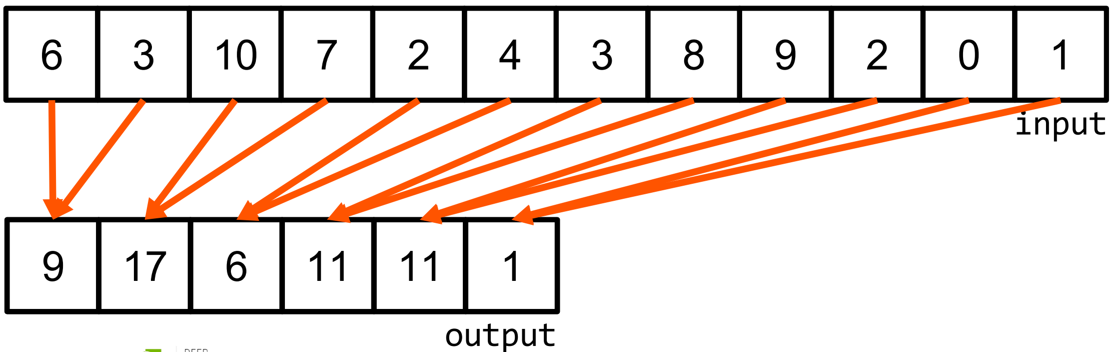
EXPRESSING PARALLELISM WITH OPENACC

OpenACC NVIDIA DEEP LEARNING INSTITUTE

CODING WITH OPENACC

Array pairing example

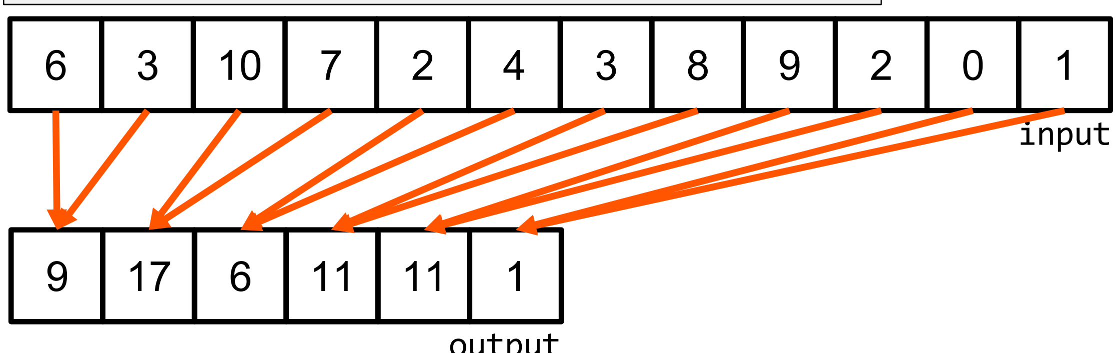
```
void pairing(int *input, int *output, int N){

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



CODING WITH OPENACC

Array pairing example - parallel

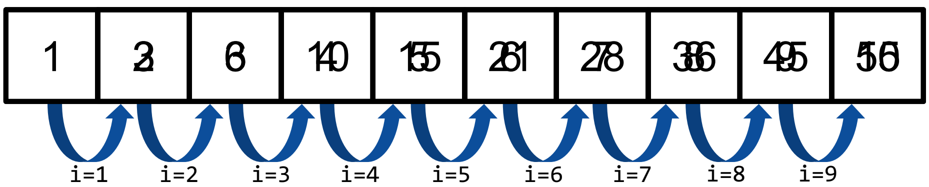
```
void pairing(int *input, int *output, int N){
        #pragma acc parallel loop
        for(int i = 0; i < N; i++)
                output[i] = input[i*2] + input[i*2+1];
}
```



DATA DEPENDENCIES

Not all loops are parallel

```
void pairing(int *a, int N){

        for(int i = 1; i < N; i++)
                a[i] = a[i] + a[i-1];
}
```

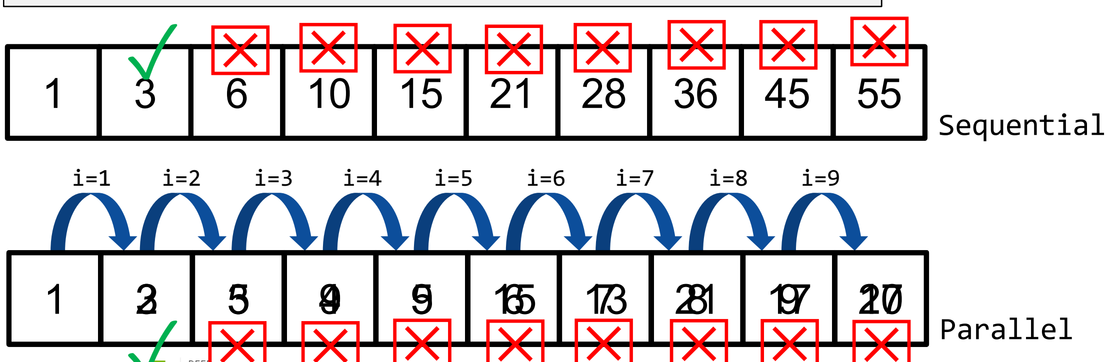


DATA DEPENDENCIES

Not all loops are parallel


```
void pairing(int *a, int N){
        #pragma acc parallel loop
        for(int i = 1; i < N; i++)
                a[i] = a[i] + a[i-1];
}
```

If we attempted to parallelize this loop we would get wrong answers due to a *forward dependency.*

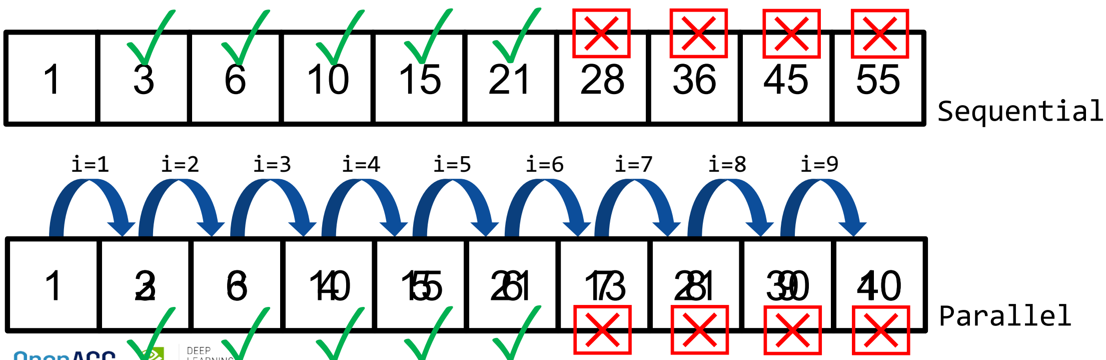


DATA DEPENDENCIES

Not all loops are parallel

```
void pairing(int *a, int N){
        #pragma acc parallel loop
        for(int i = 1; i < N; i++)
                a[i] = a[i] + a[i-1];
}
```

Even changing how the iterations are parallelized will not make this loop safe to parallelize.



Profiling

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 -fast flag instructs the compiler to optimize the code to the best of its abilities

```
$       pgcc –fast main.c
$       pgc++ -fast main.cpp
```

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
```

GCC COMPILER BASICS

gcc, gc++ and gfortran

The command to compile C code is ‘gcc’

The command to compile C++ code is ‘g++’

The command to compile Fortran code is ‘gfortran’

The -O2 flag sets the optimization level to 2 (a safe starting point)

```
$       gcc -O2 main.c
$       g++ -O2 main.cpp
```

GCC COMPILER BASICS

Compiler feedback

The -fopt-info flag will print limited compiler feedback

The -flto-report flag will also print link-time optimizations, but should be used sparingly due to volume of information

```
$      gcc -O2 –fopt-info main.c
$      g++ -O2 –fopt-info main.cpp
```

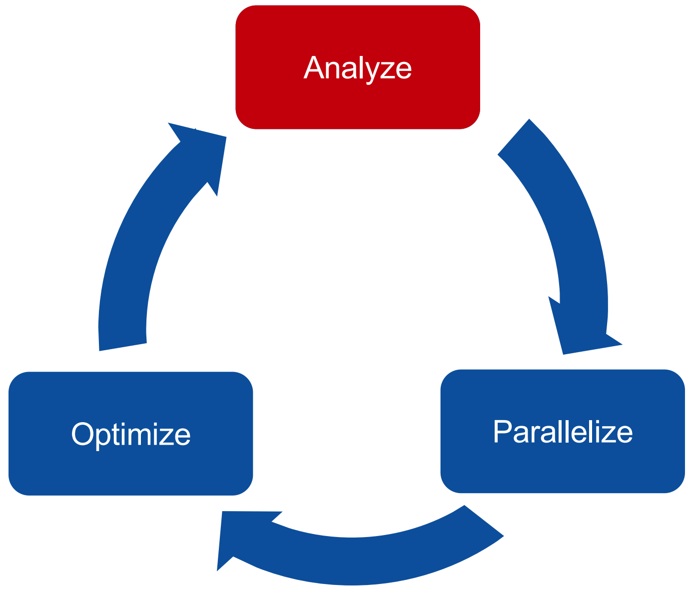

PROFILING SEQUENTIAL CODE

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

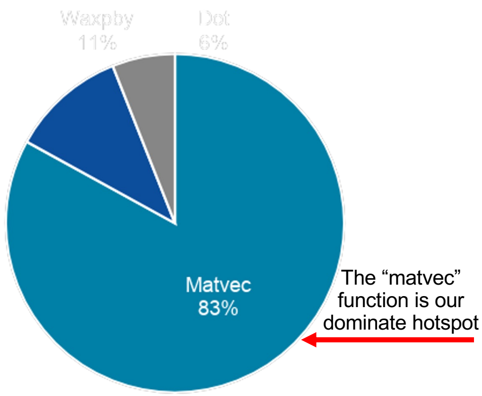
Obtain detailed information about how the code ran.

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.

Sample Code: Conjugate Gradient

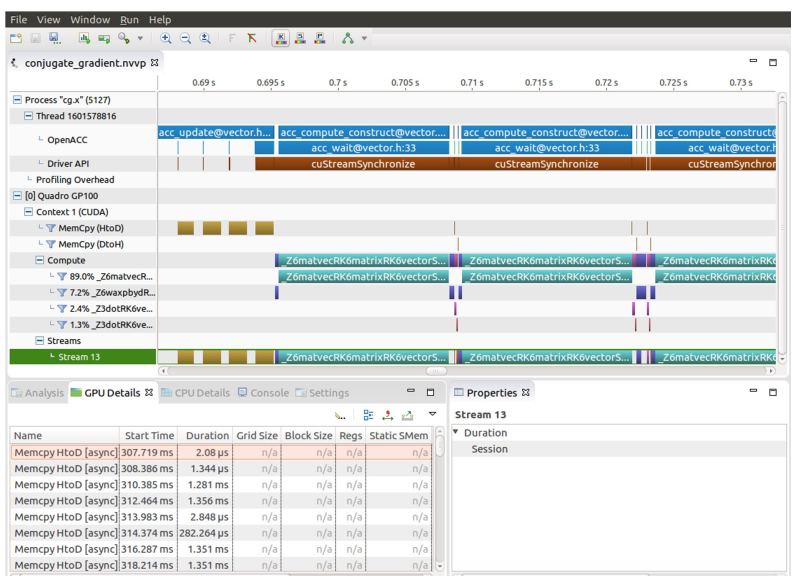
Total Runtime: 22.38 seconds



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



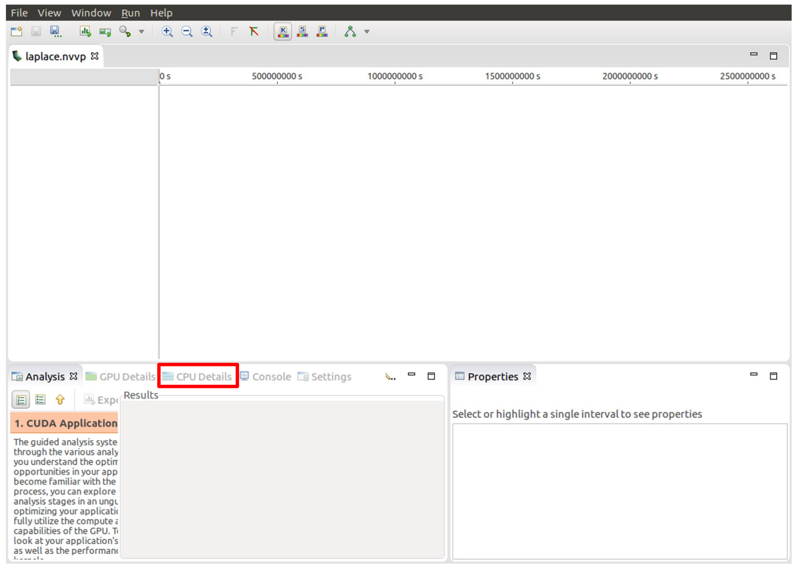
PROFILING SEQUENTIAL CODE

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



OpenACC DEEP LEARNING INSTITUTE

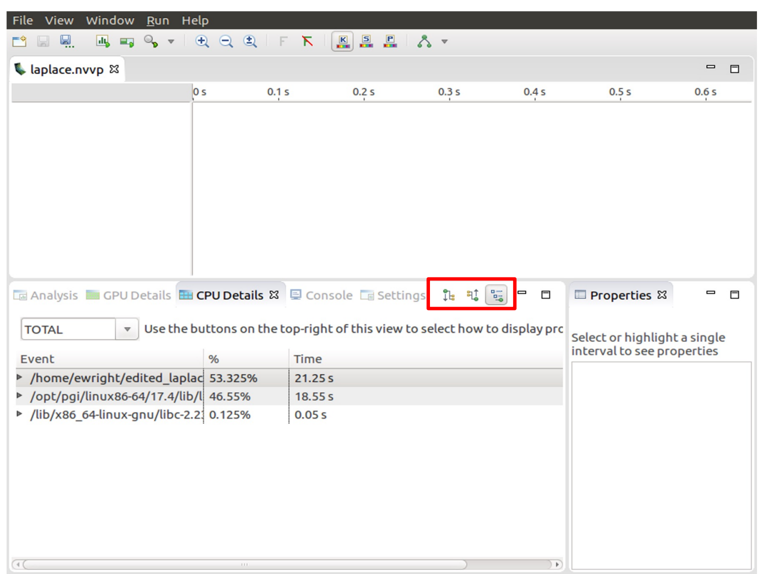
PROFILING SEQUENTIAL CODE

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



OpenACC DEEP LEARNING INSTITUTE

PROFILING SEQUENTIAL CODE

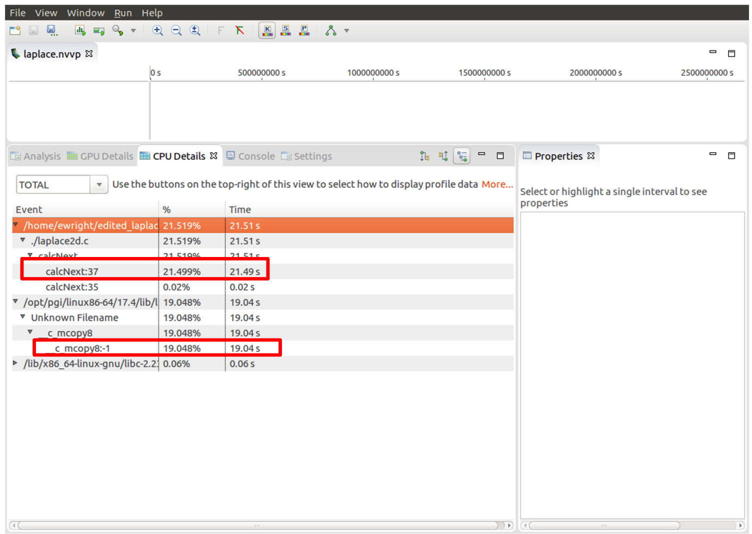
CPU Details

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



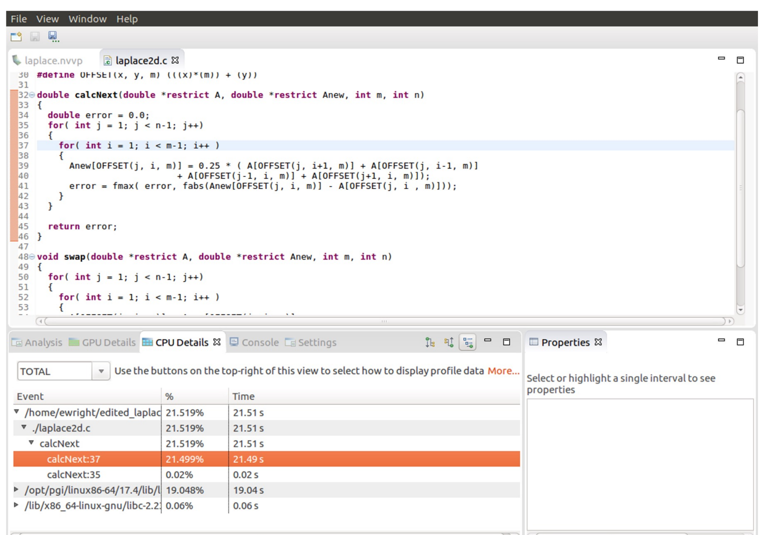
OpenACC NVIDIA DEEP LEARNING INSTITUTE

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:37” element, which opened up our code to show the exact line (line 37) that is associated with that element



OpenACC NVIDIA DEEP LEARNING INSTITUTE

PROFILING SEQUENTIAL CODE

Step 2: Profile Your Code

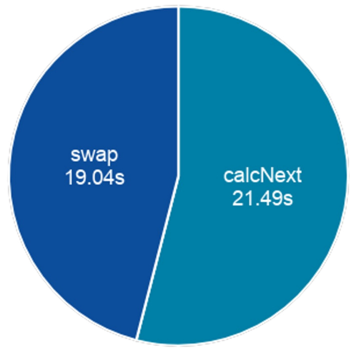
Obtain detailed information about how the code ran.

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

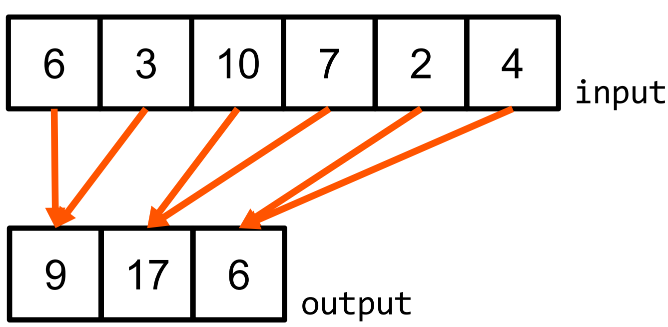
Step 3: Identify Parallelism

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];
}
```



THANK YOU