

MODULE TWO: PROFILING

Dr. Volker Weinberg | LRZ

MODULE OVERVIEW

Topics to be covered

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

COMPILING SEQUENTIAL CODE

NVIDIA'S HPC COMPILERS (AKA PGI)

NVIDIA Compiler Names (PGI names still work)

- `nvc` - The command to compile C code (formerly known as 'pgcc')
- `nvc++` - The command to compile C++ code (formerly known as 'pgc++')
- `nvfortran` - The command to compile Fortran code (formerly known as `pgfortran/pgf90/pgf95/pgf77`)
- The `-fast` flag instructs the compiler to optimize the code to the best of its abilities

```
$ nvc -fast main.c  
$ nvc++ -fast main.cpp  
$ nvfortran -fast main.F90
```

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

NVIDIA'S HPC COMPILERS (AKA PGI)

-Minfo flag

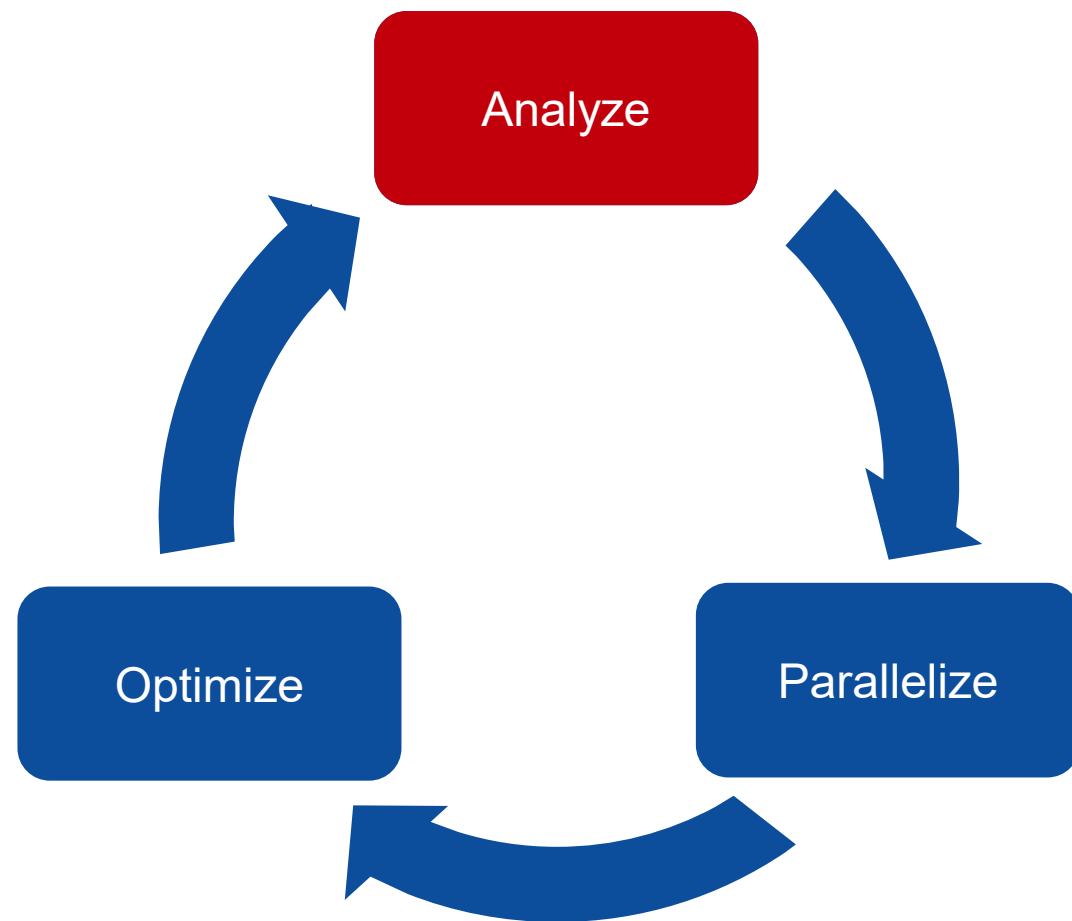
- The Minfo flag will instruct the compiler to print feedback about the compiled code
- -Minfo=accel will give us information about what parts of the code were accelerated via OpenACC
- -Minfo=opt will give information about all code optimizations
- -Minfo=all will give all code feedback, whether positive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

```
$ nvc -fast -Minfo=all main.c  
$ nvc++ -fast -Minfo=all main.cpp  
$ nvfortran -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

```
$ nvc -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.

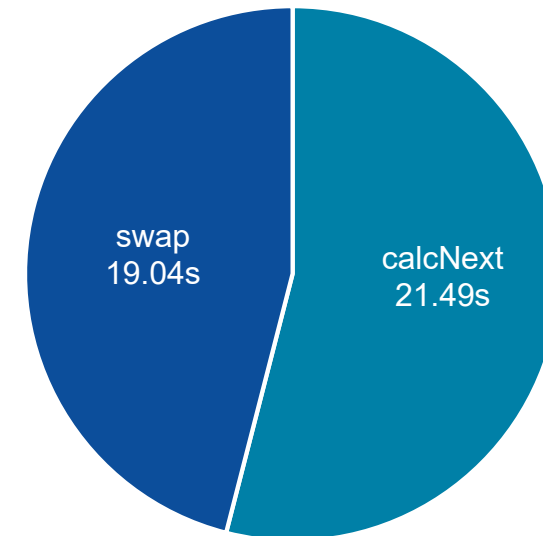
This can include information such as:

- 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.

Lab Code: Laplace Heat Transfer

Total Runtime: 39.43 seconds



PROFILING WITH NSIGHT SYSTEM AND NVTX

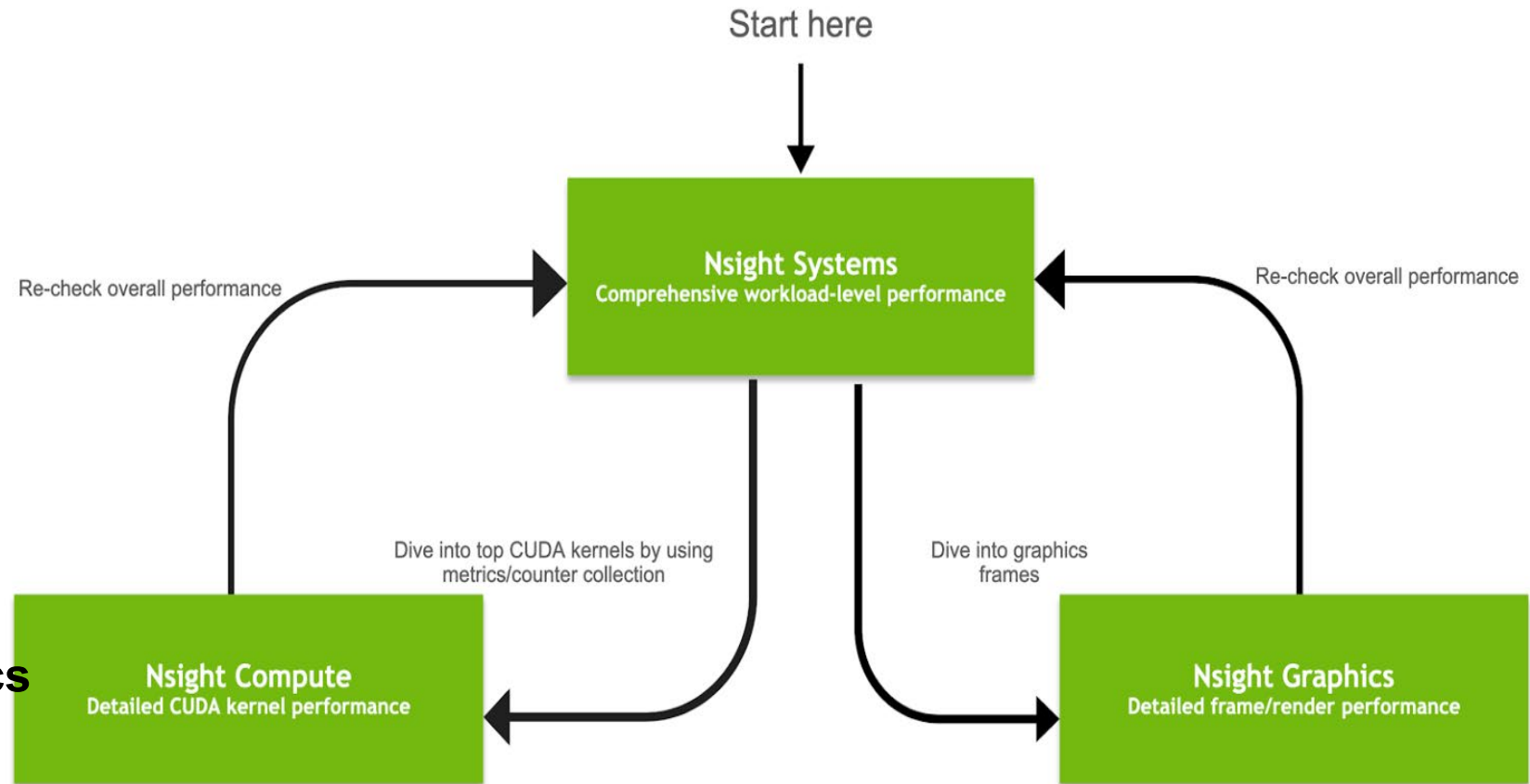
Nsight Product Family

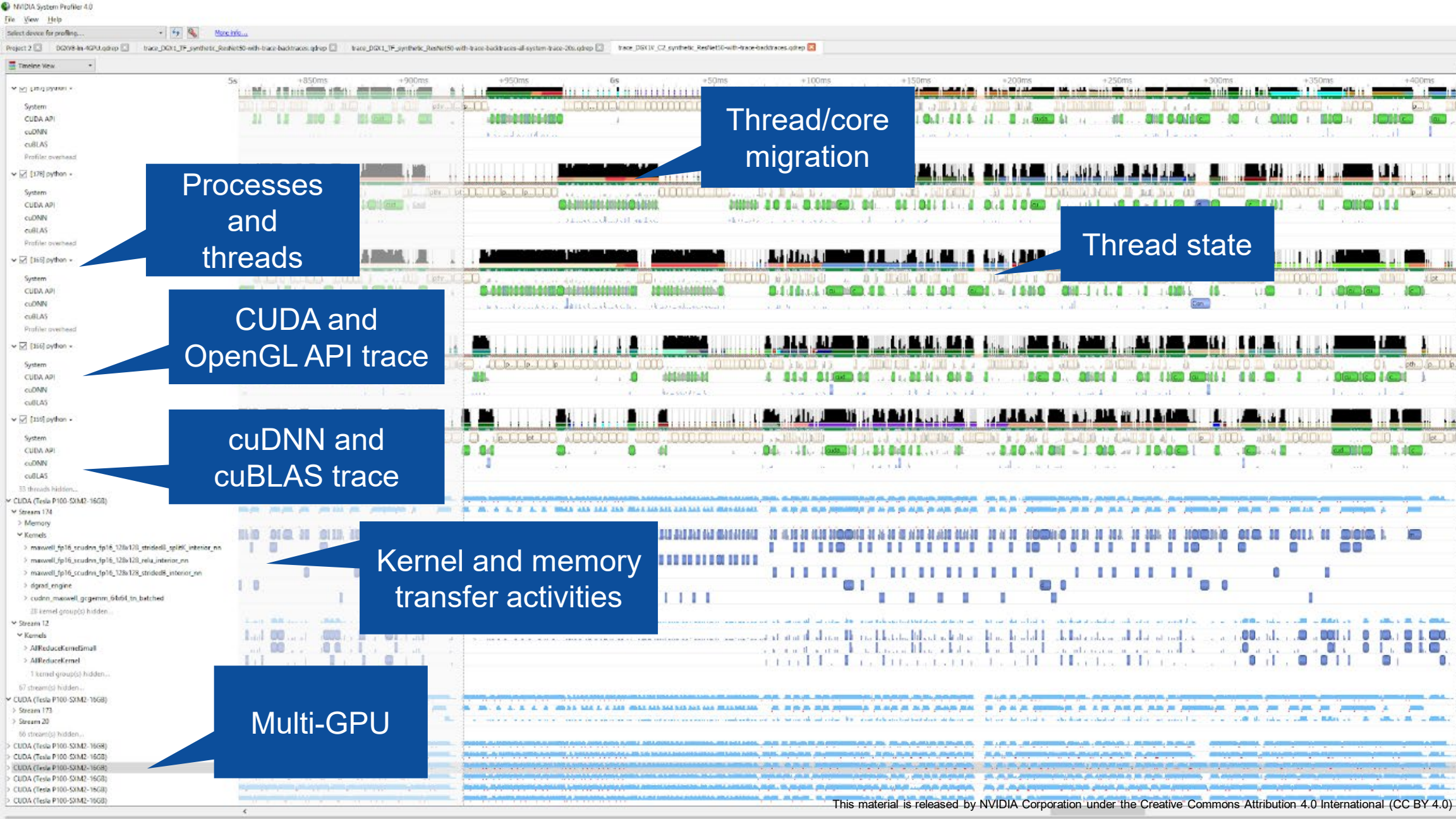
Workflow

**Nsight Systems -
Analyze application
algorithm system-wide**

**Nsight Compute -
Debug/optimize CUDA
kernel**

**Nsight Graphics -
Debug/optimize graphics
workloads**





Processes and threads

CUDA and OpenGL API trace

cuDNN and cuBLAS trace

Kernel and memory transfer activities

Multi-GPU

Thread/core migration

Thread state

PROFILING SEQUENTIAL CODE

Using Command Line Interface (CLI)

NVIDIA Nsight Systems CLI provides

- Simple interface to collect data
- Can be copied to any system and analysed later
- Profiles both serial and parallel code
- For more info enter `nsys --help` on the terminal

To profile a serial application with NVIDIA Nsight Systems, we use NVIDIA Tools Extension (NVTX) API functions in addition to collecting backtraces while sampling.

PROFILING SEQUENTIAL CODE

NVIDIA Tools Extension API (NVTX) library

What is it?

- A C-based Application Programming Interface (API) for annotating events
- Can be easily integrated to the application
- Can be used with NVIDIA Nsight Systems

Why?

- Allows manual instrumentation of the application
- Allows additional information for profiling (e.g: tracing of CPU events and time ranges)

How?

- Import the header only C library `nvToolsExt.h`
- Wrap the code region or a specific function with `nvtxRangePush()` and `nvtxRangePop()`

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "laplace2d.h"
#include <nvtx3/nvToolsExt.h>

int main(int argc, char** argv)
{
    const int n = 4096;
    const int m = 4096;
    const int iter_max = 1000;

    const double tol = 1.0e-6;
    double error = 1.0;

    double *restrict A = (double*)malloc(sizeof(double)*n*m);
    double *restrict Anew = (double*)malloc(sizeof(double)*n*m);

    nvtxRangePushA("init");
    initialize(A, Anew, m, n);
    nvtxRangePop();

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    double st = omp_get_wtime();
    int iter = 0;

    nvtxRangePushA("while");
    while ( error > tol && iter < iter_max )
    {
        nvtxRangePushA("calc");
        error = calcNext(A, Anew, m, n);
        nvtxRangePop();

        nvtxRangePushA("swap");
        swap(A, Anew, m, n);
        nvtxRangePop();

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
    nvtxRangePop();

    double runtime = omp_get_wtime() - st;

    printf(" total: %f s\n", runtime);

    deallocate(A, Anew);

    return 0;
}

```

jacobi.c
(starting and ending of ranges are highlighted with the same color)

- t Selects the APIs to be traced (nvtx in this example)
- status if true, generates summary of statistics after the collection
- b Selects the backtrace method to use while sampling. The option dwarf uses DWARF's CFI (Call Frame Information).
- force-overwrite if true, overwrites the existing results
- o sets the output (qdrep) filename

```

mozhgank@prm-dgx-32:~/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel1$ nsys profile -t nvtx --stats=true -b dwarf --force-overwrite true -o laplace-seq ./laplace-seq
Collecting data...
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: 55.754501 s
Processing events...
Capturing symbol files...
Saving intermediate "/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel1/laplace-seq.qdstrm" file to disk...
Importing [=====100%]
Saved report file to "/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel1/laplace-seq.qdrep"
Exporting 70802 events: [=====100%]
Exported successfully to
/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel1/laplace-seq.sqlite
Generating NVTX Push-Pop Range Statistics...
NVTX Push-Pop Range Statistics (nanoseconds)
Time(%) Total Time Instances Average Minimum Maximum Range
-----
49.9 55754497966 1 55754497966.0 55754497966 55754497966 while
26.5 29577817696 1000 29577817.7 29092956 65008545 calc
23.4 26163892482 1000 26163892.5 25761418 60129514 swap
0.1 137489808 1 137489808.0 137489808 137489808 init

```

NVTX range statistics

"calc" region (calcNext function) takes 26.6%
"swap" region (swap function) takes 23.4% of total execution time

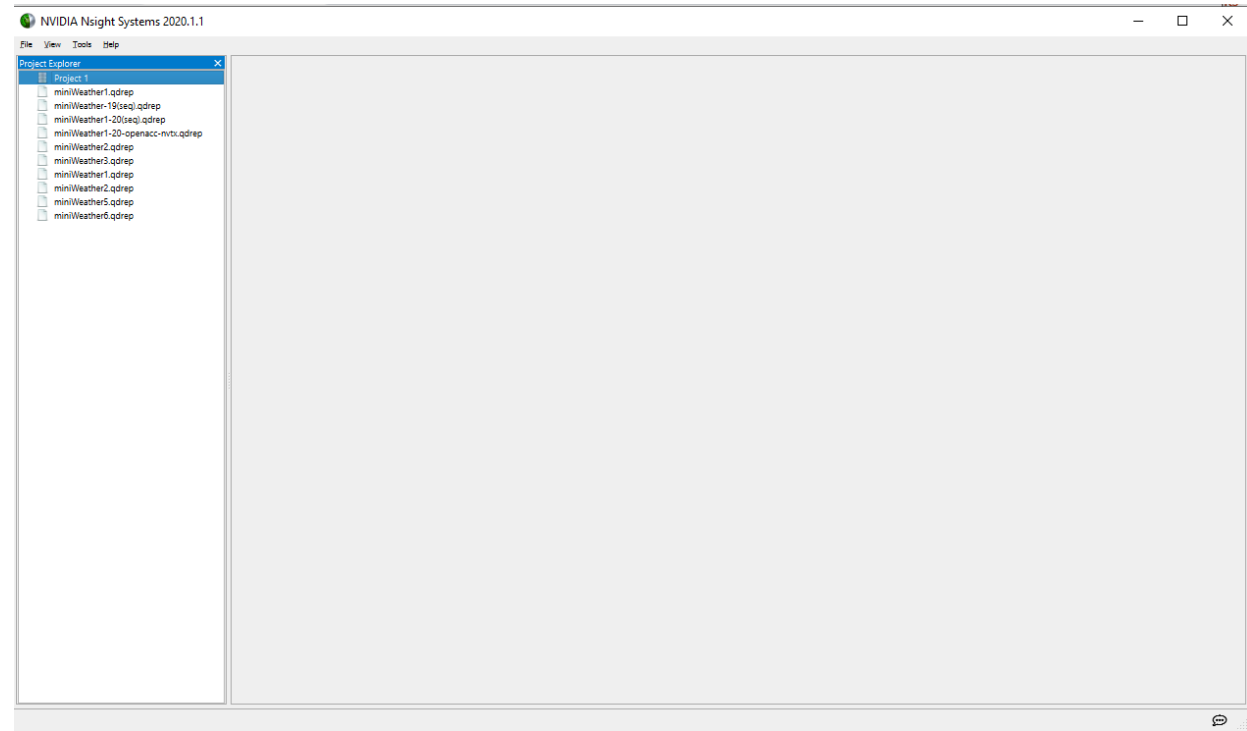
Open laplace-seq.qdrep with Nsight System GUI to view the timeline

PROFILING SEQUENTIAL CODE

Using Nsight Systems

Open the generated report files (*. nsys-rep) from command line in the Nsight Systems profiler.

File -> Open



PROFILING SEQUENTIAL CODE

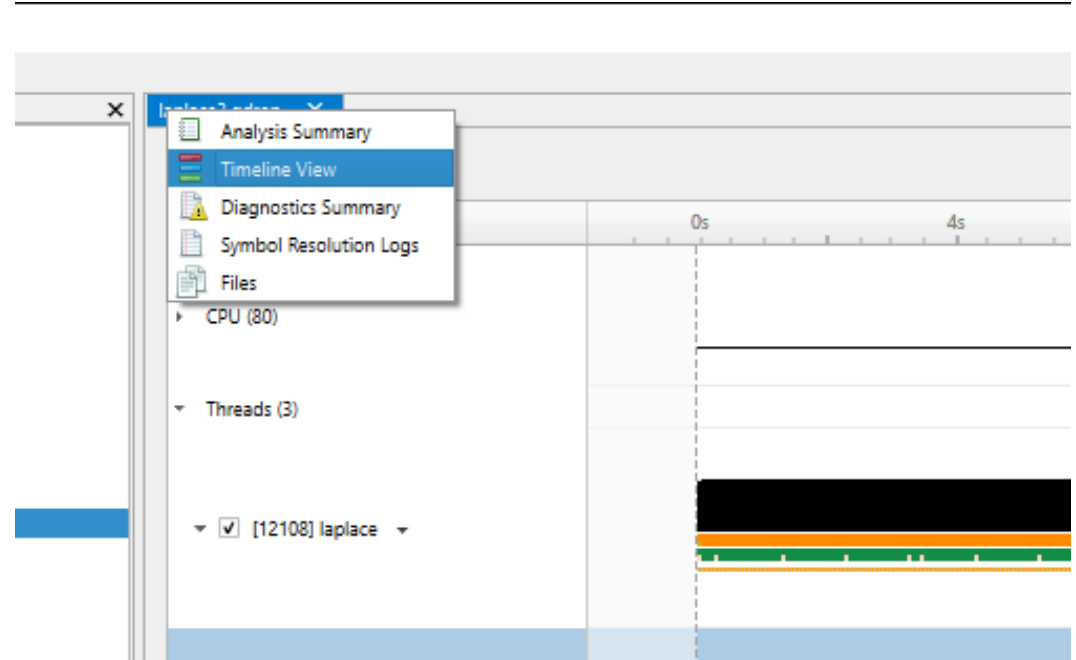
Using Nsight Systems

Navigate through the “view selector”.

“Analysis summary” shows a summary of the profiling session. To review the project configuration used to generate this report, see next slide.

“Timeline View” contains the timeline at the top, and a bottom pane that contains the events view and the function table.

Read more: <https://docs.nvidia.com/nsight-systems>



PROFILING SEQUENTIAL CODE

Using Nsight Systems

Analysis Summary

Profiling session duration: 00:55.623

Total number of threads	3
Number of events collected	70,773
Report size	851.22 KiB
Report capture date	19 March 2020 08:01:16
Host computer	prm-dgx-28
Profiling stop reason	Stopped by user
Imported from	/home/mozhgank/Code/openacc-training-materials/labs/module2/English/C/laplace3.qdstrm
Import host computer	prm-dgx-28
CLI command used	nsys profile -t nvtx --stats=true --force-overwrite true -o laplace3_laplace

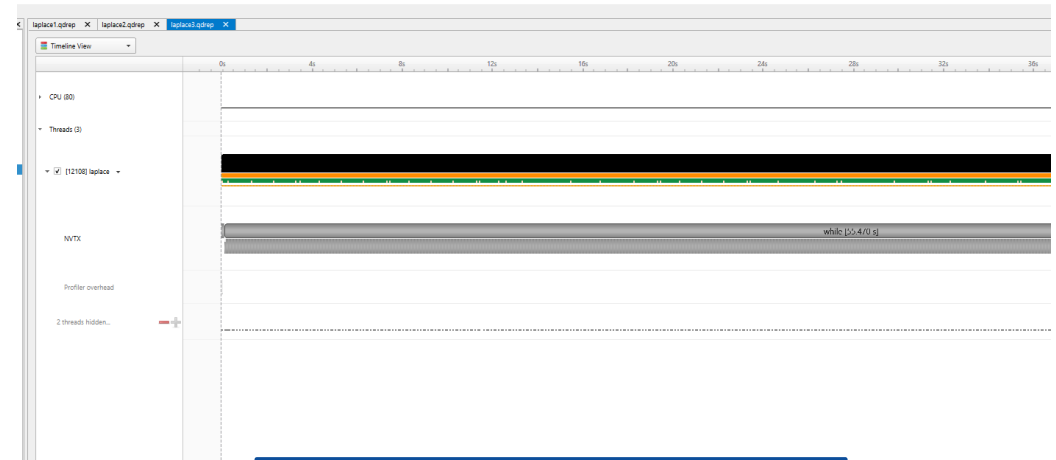
[Show report file in folder](#)

prm-dgx-28 (0:1)

Target

Target name	prm-dgx-28
Platform	Linux
OS	Ubuntu 18.04.3 LTS
Hardware platform	x86_64
Serial number	Local (CLI)
CPU description	Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
CUDA driver version	10.2
NVIDIA driver version	440.33.01
GPU context switch	supported

Analysis Summary



Timeline view
(charts and the hierarchy on the top pane)

Events View

	Name	Duration	TID	Start
1	int1333.102.mg	133.162 ms	12108	0.00089901s
2	while [0-4/0.g]	55.470 s	12108	0.141834s

Timeline view
(event view and function table on the bottom pane)

PROFILING SEQUENTIAL CODE

Using Nsight Systems

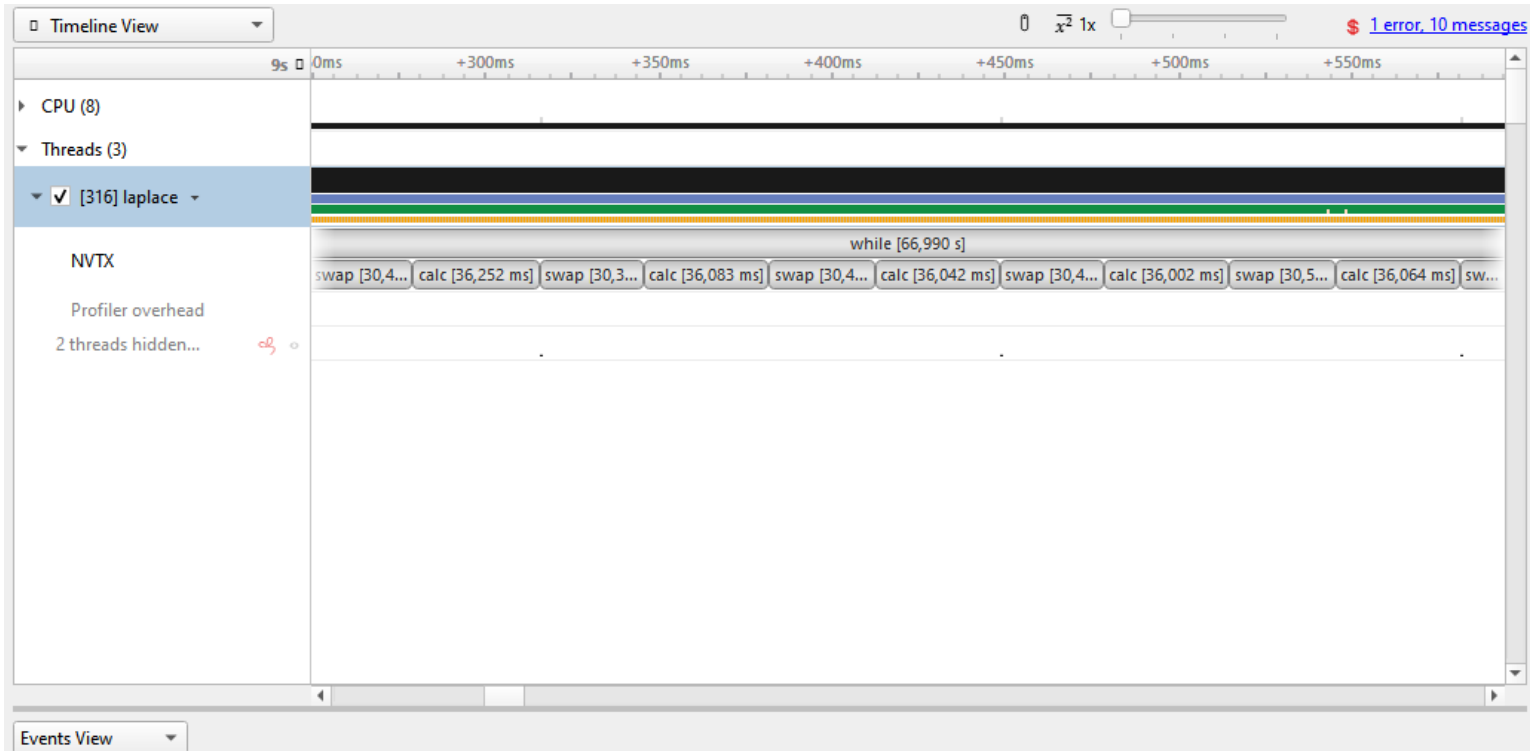
Enlarge view!

The screenshot displays the NVIDIA Nsight Systems 2021.2.1 interface. The main window shows a timeline view of a project named 'laplace.qdrep'. The timeline is divided into sections for CPU (8), Threads (3), and NVTX. A green selection box highlights a region of the timeline, and a context menu is open over it, showing options like 'Filter and Reorder', 'Filter and Zoom in', 'Remove Filter', 'Zoom into Selection', 'Undo Zoom (0)', and 'Reset Zoom'. A blue arrow points to the 'Zoom into Selection' option. The timeline shows a 'while [66,990 s]' event. The interface also includes a Project Explorer on the left and a menu bar at the top.

right click in
selected region
and Zoom into
selection!

PROFILING SEQUENTIAL CODE

Using Nsight Systems

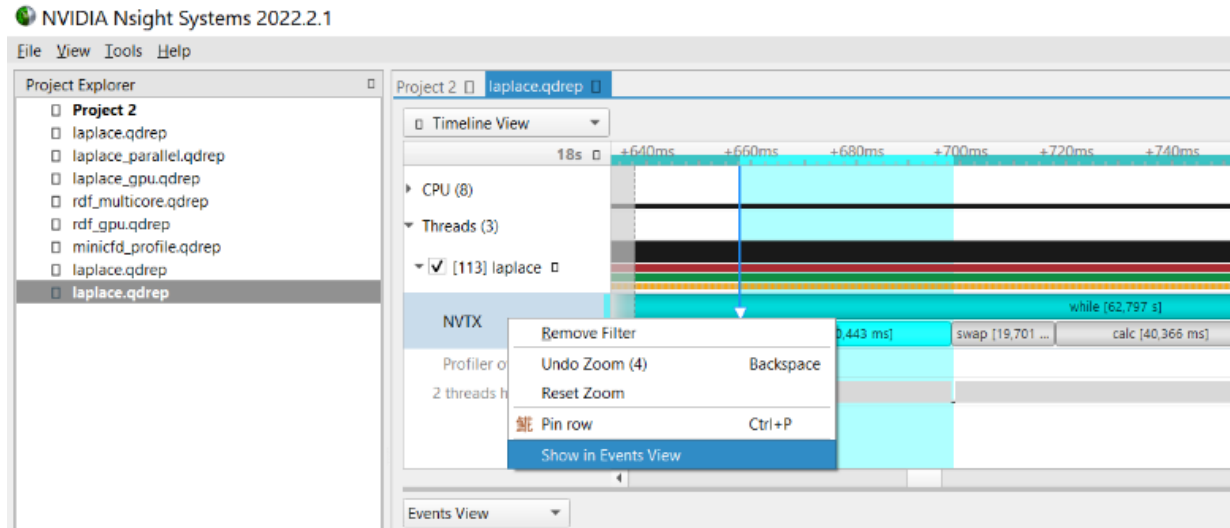


PROFILING SEQUENTIAL CODE

Viewing captured NVTX events and time ranges via Nsight Systems GUI

From the Timeline view, right click on the “NVTX” from the top pane and choose “Show in Events View”.

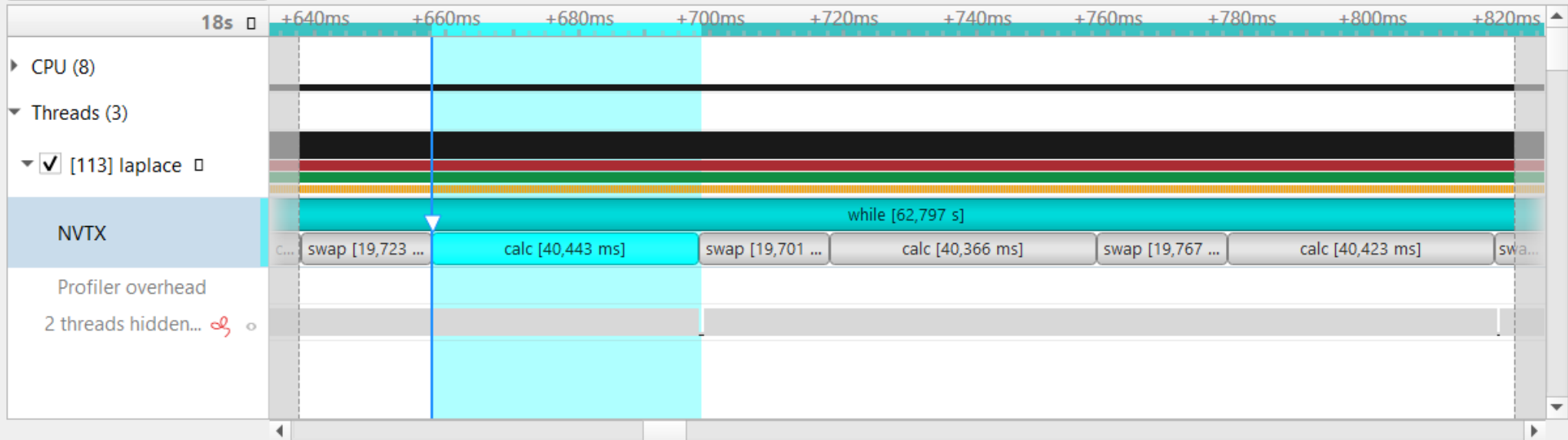
From the bottom pane, you can now see name of the events captured with the duration.



Timeline View

0 \bar{x}^2 1x

\$ 1 error, 10 messages



Events View

Name

#	Name	Start	Duration	TID	Category	Description:
1	while	0,193229s	62,797 s	113		while Begins: 0,193229s Ends: 62,9905s (+62,797 s) Thread: 113
2	calc	18,5981s	40,468 ms	113		
3	swap	18,6386s	19,723 ms	113		
4	calc	18,6583s	40,443 ms	113		
5	swap	18,6988s	19,701 ms	113		
6	calc	18,7185s	40,366 ms	113		
7	swap	18,7589s	19,767 ms	113		

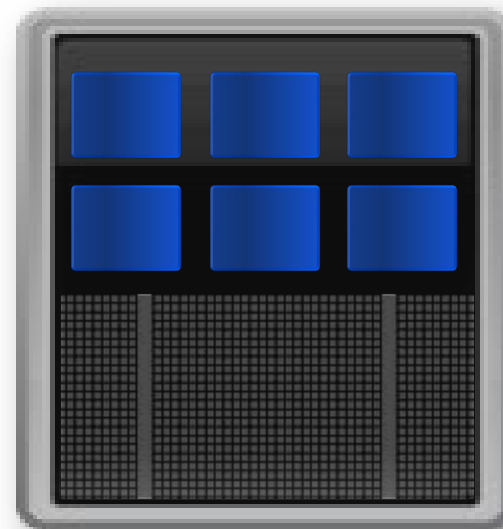
PROFILING MULTICORE CODE

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

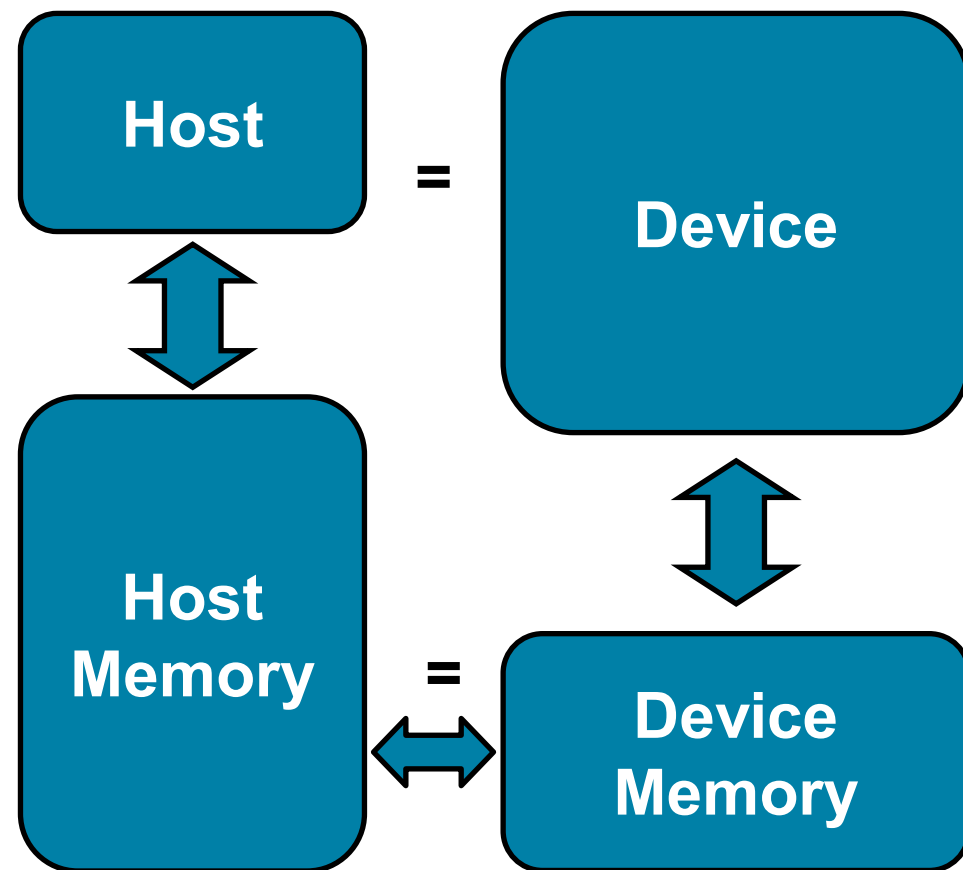
CPU



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 '-acc' flag (previously -ta) will allow us to compile our code for a specific, target parallel hardware
- 'acc' stands for "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

```
40o0002@a0122:~/OpenACC-solutions/C$ nvc -fast -acc=multicore -Minfo=accel,opt -I
/opt/nvidia/hpc_sdk/Linux_x86_64/25.3/compilers/include-stdpar/ -fast -o laplace-nvtx
jacobi-nvtx.c laplace2d-multicore-parallel.c
jacobi-nvtx.c:
laplace2d-multicore-parallel.c:
calcNext:
    47, Generating Multicore code
        49, #pragma acc loop gang
            Generating implicit reduction(max:error)
```

PROFILING MULTICORE CODE

Compiling code for a specific parallel hardware

```
a40o0002@a0122:~/OpenACC-solutions/C$ nsys profile -t nvtx --stats=true --force-overwrite true -o laplace-nvtx ./laplace-nvtx
Collecting data...
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: 17.956820 s
Generating '/tmp/3094607.alex/nsys-report-46fc.qdstrm'
[1/3] [=====100%] laplace-nvtx.nsys-rep
[2/3] [=====100%] laplace-nvtx.sqlite
[3/3] Executing 'nvtx_sum' stats report
```

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Style	Range
50.0	17956815751	1	17956815751.0	17956815751.0	17956815751	17956815751	0.0	PushPop	:while
25.3	9094576060	1000	9094576.1	8506796.0	8324557	23777785	2028271.5	PushPop	:swap
24.6	8837131024	1000	8837131.0	8219959.5	8020293	22946236	2055146.7	PushPop	:calc
0.1	25125688	1	25125688.0	25125688.0	25125688	25125688	0.0	PushPop	:init

```
Generated:
  /home/hpc/k_a40o/a40o0002/OpenACC-solutions/C/laplace-nvtx.nsys-rep
_  /home/hpc/k_a40o/a40o0002/OpenACC-solutions/C/laplace-nvtx.sqlite
a40o0002@a0122:~/OpenACC-solutions/C$
```

PARALLEL VS SEQUENTIAL

Compiler feedback

Have a close look at the compiler feedback for both sequential and parallel implementation of the application.

It provides information about how your program was optimized or why a particular optimization was not made.

Note: Adding `-Minfo` flag or `-Minfo=accel` or `-Minfo=all` when compiling, will enable compiler feedback messages, giving details about the parallel code generated.

```
jacobi.c:
laplace2d.c:
calcNext:
  38, Accelerator restriction: size of the GPU copy of Anew,A is unknown
  Loop is parallelizable
  Generating Multicore code
  38, #pragma acc loop gang
  38, Generating implicit reduction(max:error)
  41, Loop is parallelizable
swap:
  54, Accelerator restriction: size of the GPU copy of Anew,A is unknown
  Loop is parallelizable
  Generating Multicore code
  54, #pragma acc loop gang
  57, Loop is parallelizable
  Memory copy idiom, loop replaced by call to __c_mcopy8
/trainvw@hawk-ai01:~/OpenACC/C/solutions>
```

```
jacobi.c:
laplace2d-multicore-parallel.c:
calcNext:
  47, Generating implicit firstprivate(j,n,m)
  Generating NVIDIA GPU code
  49, #pragma acc loop gang /* blockIdx.x */
  Generating implicit reduction(max:error)
  52, #pragma acc loop vector(128) /* threadIdx.x */
  47, Generating implicit copyin(A[:]) [if not already present]
  Generating implicit copy(error) [if not already present]
  Generating implicit copyout(Anew[:]) [if not already present]
  52, Loop is parallelizable
swap:
  63, Generating implicit firstprivate(j,n,m)
  Generating NVIDIA GPU code
  65, #pragma acc loop gang /* blockIdx.x */
  68, #pragma acc loop vector(128) /* threadIdx.x */
  63, Generating implicit copyout(A[:]) [if not already present]
  Generating implicit copyin(Anew[:]) [if not already present]
  68, Loop is parallelizable
/trainvw@hawk-ai01:~/OpenACC/C/solutions>
```

```

#include <math.h>
#include <stdlib.h>

#define OFFSET(x, y, m) (((x)*(m)) + (y))

void initialize(double *restrict A, double *restrict Anew, int m, int n)
{
    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

    for(int i = 0; i < m; i++){
        A[i] = 1.0;
        Anew[i] = 1.0;
    }
}

double calcNext(double *restrict A, double *restrict Anew, int m, int n)
{
    double error = 0.0;
    #pragma acc parallel loop reduction(max:error)

    for( int j = 1; j < n-1; j++)
    {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ )
        {
            Anew[OFFSET(j, i, m)] = 0.25 * ( A[OFFSET(j, i+1, m)] + A[OFFSET(j, i-1, m)]
            + A[OFFSET(j-1, i, m)] + A[OFFSET(j+1, i, m)]);
            error = max( error, fabs(Anew[OFFSET(j, i, m)] - A[OFFSET(j, i, m)]));
        }
    }
    return error;
}

void swap(double *restrict A, double *restrict Anew, int m, int n)
{
    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++)
    {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ )
        {
            A[OFFSET(j, i, m)] = Anew[OFFSET(j, i, m)];
        }
    }
}

void deallocate(double *restrict A, double *restrict Anew)
{
    free(A);
    free(Anew);
}

```

laplace2d.c
 (Parallelised using OpenACC parallel
 directives (pragmas highlighted))

Importing [=====100%]
 Saved report file to "/home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel/laplace-par.qdrep"
 Exporting 59427 events: [=====100%]

Exported successfully to
 /home/mozhgank/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel/laplace-par.sqlite

Generating CUDA API Statistics...
 CUDA API Statistics (nanoseconds)

Time (%)	Total Time	Calls	Average	Minimum	Maximum	Name
93.1	1107821913	4000	276955.5	2205	63176580	cuStreamSynchronize
2.2	25667004	1	25667004.0	25667004	25667004	cuMemHostAlloc
1.8	21314088	1	21314088.0	21314088	21314088	cuMemAllocManaged
1.6	18919205	3000	6306.4	4141	38827	cuLaunchKernel
0.6	6575301	1000	6575.3	5719	37100	cuMemcpyDtoHAsync_v2
0.4	4672638	1000	4672.6	3898	23307	cuMemsetD32Async
0.2	1957950	1000	1958.0	1656	9112	cuEventRecord
0.2	1835300	1000	1835.3	1685	15602	cuEventSynchronize
0.1	1013630	1	1013630.0	1013630	1013630	cuMemAllocHost_v2
0.0	498778	3	166259.3	7731	462241	cuMemAlloc_v2
0.0	251874	1	251874.0	251874	251874	cuModuleLoadDataEx
0.0	9531	1	9531.0	9531	9531	cuStreamCreate
0.0	3936	2	1968.0	1136	2800	cuEventCreate

Generating CUDA Kernel Statistics...
 CUDA Kernel Statistics (nanoseconds)

Time (%)	Total Time	Instances	Average	Minimum	Maximum	Name
58.9	643260973	1000	643261.0	563448	63174901	calcNext_35_gpu
40.4	441862005	1000	441862.0	434490	452729	swap_51_gpu
0.7	7350076	1000	7350.1	6880	7936	calcNext_35_gpu__red

Generating CUDA Memory Operation Statistics...
 CUDA Memory Operation Statistics (nanoseconds)

Time (%)	Total Time	Operations	Average	Minimum	Maximum	Name
94.0	47346144	8191	5780.3	2688	71456	[CUDA Unified Memory memcpy HtoD]
3.1	1567908	1000	1567.9	1535	2816	[CUDA memcpy DtoH]
2.9	1447303	1000	1447.3	1407	2080	[CUDA memset]

CUDA Memory Operation Statistics (KiB)

Total	Operations	Average	Minimum	Maximum	Name
7.813	1000	0.008	0.008	0.008	[CUDA memcpy DtoH]
7.813	1000	0.008	0.008	0.008	[CUDA memset]
260072.000	8191	31.751	4.000	732.000	[CUDA Unified Memory memcpy HtoD]

Generating NVTX Push-Pop Range Statistics...
 NVTX Push-Pop Range Statistics (nanoseconds)

Time (%)	Total Time	Instances	Average	Minimum	Maximum	Range
47.6	1199130726	1	1199130726.0	1199130726	1199130726	while
29.2	734660957	1000	734661.0	627348	89307285	calc
18.3	461447578	1000	461447.6	454563	490929	swap
4.9	122270944	1	122270944.0	122270944	122270944	init

CUDA API statistics

CUDA Kernel statistics

CUDA Memory Operation statistics

NVTX range statistics

"calc" region (calcNext function) takes 29.2%
 "swap" region (swap function) takes 18.3% of
 total execution time

Open laplace-par.qdrep
 with Nsight System GUI to
 view the timeline

PARALLEL VS SEQUENTIAL SPEEDUP

Viewing captured NVTX events

Have a close look at the captured NVTX events for both serial and parallel implementations.

Time spent in “while” loop has significantly decreased.

Achieved speedup: ~47

```
mozhgank@prm-dgx-32:~/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel$ nsys profile -t nvtx,openacc --stats=true -b dwarf --force-overwrite true -o laplace-par ./laplace-par
WARNING: openACC, cuDNN and cuBLAS rely on CUDA. CUDA tracing has been automatically enabled.
Collecting data...
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: 1.199133 s
```

Parallel

```
mozhgank@prm-dgx-32:~/Code/openacc-training-materials/labs/module4/English/C/solutions/parallel$ nsys profile -t nvtx --stats=true -b dwarf --force-overwrite true -o laplace-seq ./laplace-seq
Collecting data...
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: 55.754501 s
```

Sequential

#	Name	Duration	TID	Start
1	init [122.271 ms]	122.271 ms	45547	0.84707s
3	while [1.199 s]	1.199 s	45547	0.969399s

Parallel

#	Name	Duration	TID	Start
1	init [137.490 ms]	137.490 ms	46133	0.00975564s
3	while [55.754 s]	55.754 s	46133	0.147279s

Sequential

LAB CODE

LAPLACE HEAT TRANSFER

Introduction to lab code - visual

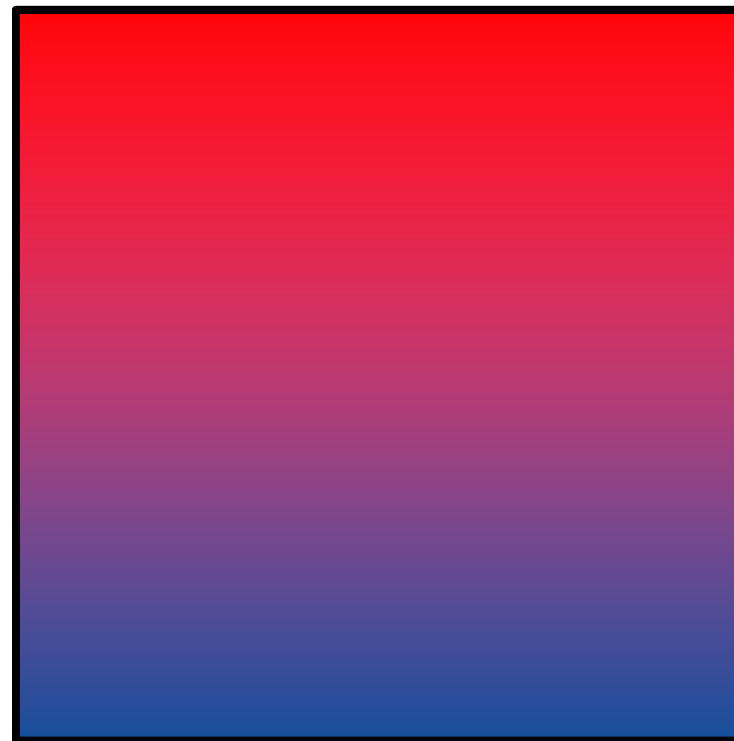
We will observe a simple simulation of heat distributing across a metal plate.

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

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

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 NVIDIA's HPC compiler
- Use Nsight Systems to understand where the program spends its time

THANK YOU