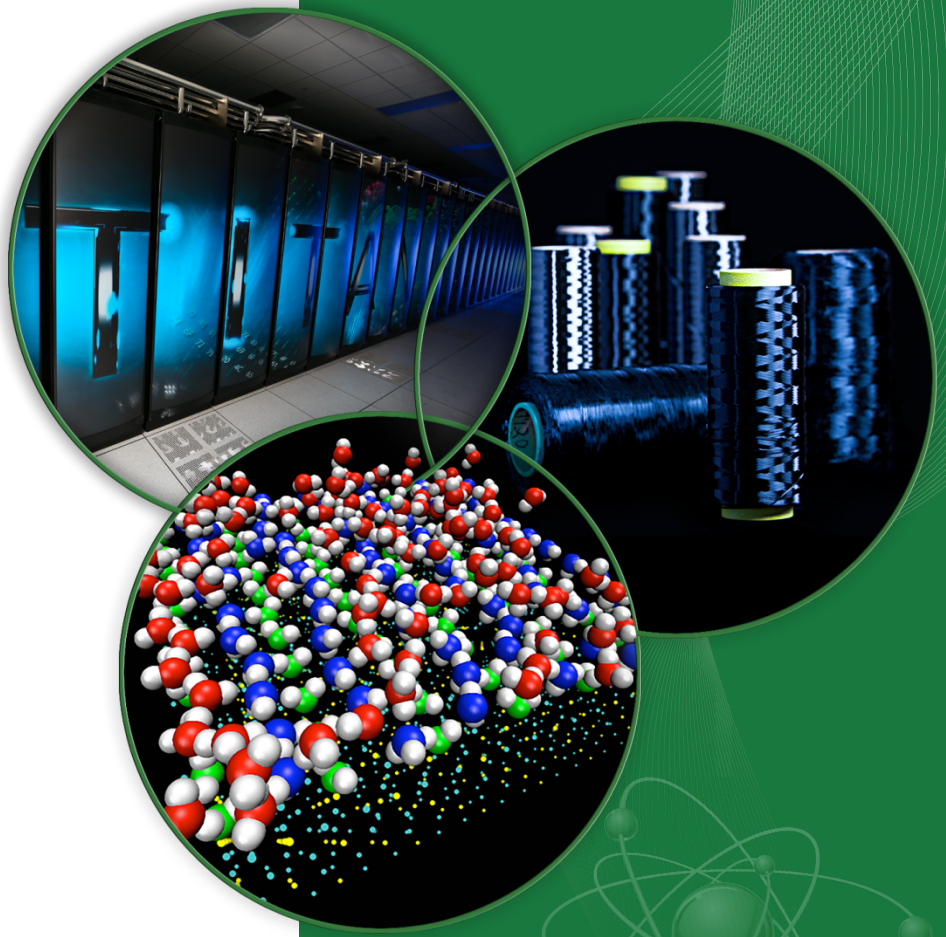
Hands On: Mandelbrot

OpenACC



Verónica G. Vergara Larrea

Adam B. Simpson

Tom Papatheodore

CSGF 2017

July 26, 2017

ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Outline of this Document

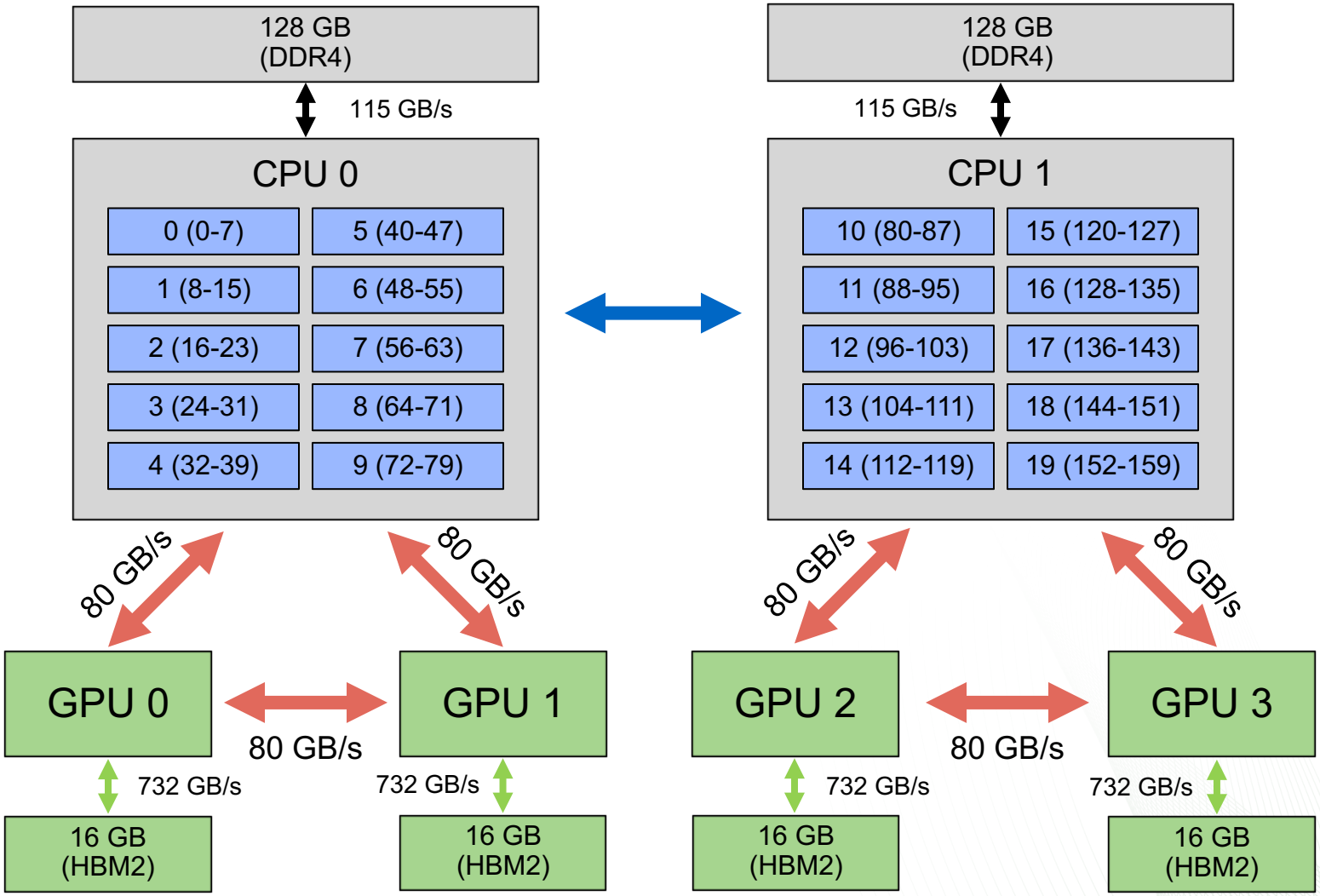
- This document assumes no previous experience with OpenACC, so it will begin by introducing the basic features of OpenACC using a simple vector addition program.
- Afterward, it will guide you through the steps of accelerating a Mandelbrot program using OpenACC, introducing more advanced features along the way.

  The outline of these steps will be as follows:

  1. Write a serial Mandelbrot program using C or C++.
  2. Add OpenACC directives to offload the work of computing pixels to a single GPU.
  3. Add pipelining to overlap computation and data transfers for “blocks” of the image array (still single GPU).
  4. Add ability to target multiple GPUs using OpenMP (or MPI)
  5. Choose from (a list of) other options your group wishes to pursue.

SummitDev "Minsky" Node

(2) IBM Power8 + (4) NVIDIA Pascal P100

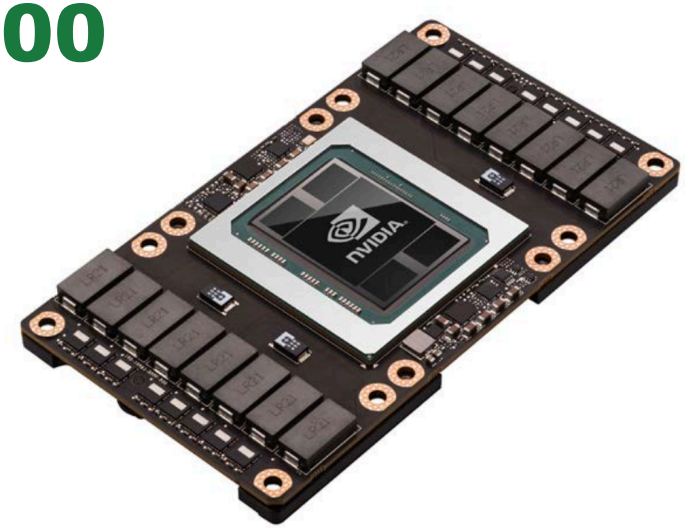


NVIDIA P100
(nvidia.com)

SPECIFICATIONS

GPU Architecture	**NVIDIA Pascal**
NVIDIA CUDA® Cores	**3584**
Double-Precision Performance	**5.3 TeraFLOPS**
Single-Precision Performance	**10.6 TeraFLOPS**
Half-Precision Performance	**21.2 TeraFLOPS**
GPU Memory	**16 GB CoWoS HBM2**
Memory Bandwidth	**732 GB/s**
Interconnect	**NVIDIA NVLink**
Max Power Consumption	**300 W**
ECC	**Native support with no capacity or performance overhead**
Thermal Solution	**Passive**
Form Factor	**SXM2**
Compute APIs	**NVIDIA CUDA, DirectCompute, OpenCL™, OpenACC**

TeraFLOPS measurements with NVIDIA GPU Boost™ technology



P100 - Streaming Multiprocessor (SM)

56 SMs per P100

More Summitdev Info

OLCF Summitdev Quick Start Guide

https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/

Documents from Summitdev training workshop

https://www.olcf.ornl.gov/training-event/summitdev-training-class/

Quick Tutorial of OpenACC Basics

This section of the document is meant to serve as a quick tutorial to bring you up to speed on the basics of OpenACC.

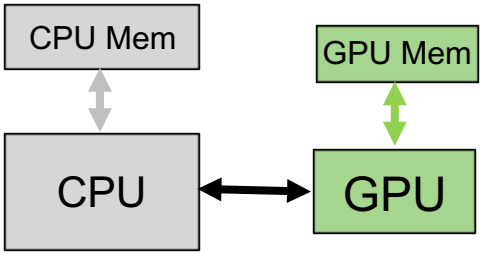
The code used in this section can be obtained from

```
git clone https://github.com/tpapathe/quick_intro_openacc.git
```

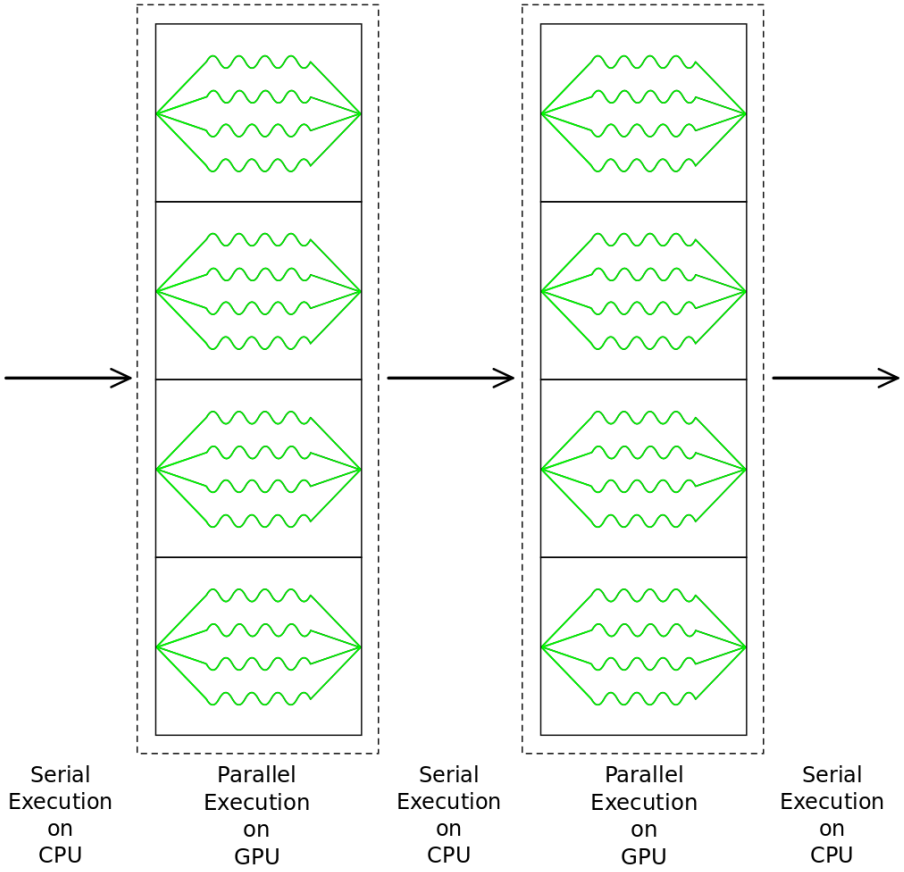
Heterogeneous Programming Model

CPU and GPU have separate memory spaces

- Data must be passed back and forth where/when needed



Serial work is performed on CPU while parallel work is offloaded to GPU



What is OpenACC?

OpenACC is an Application Programming Interface (API) for offloading portions of programs to run on accelerator devices.

- Compiler directives
  - Programmer gives the compiler hints about where parallelism can be found in the code. The compiler then creates device code to run on GPU and tries to determine when/where data needs to be transferred between CPU and GPU.
- Library routines
- Environment variables

Compiler Directives

```
#pragma acc kernels
{
for(int i=0; i<N; i++){
  C[i] = A[i] + B[i];
}
}
```

An OpenACC compiler directive, which is ignored unless compiled with the proper flag (e.g, -acc for PGI)

In a serial vector addition code, each iteration of the loop is performed sequentially. However, each of the pair-wise additions are independent of each other, so they can be performed in parallel (data parallelism).

The `kernels` directive tells the compiler to look for parallelism in the structured block of code, and if possible, create (kernel) code to run on an accelerator.

- The for loop is converted into a GPU kernel, where each pair-wise addition is performed by a separate CUDA core on the GPU (i.e. no more loop).

```
C[i] = A[i] + B[i];
```

NOTE: There is no guarantee of the order each "iteration" is computed on the GPU.

Example: Vector Addition

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *A = (double*)malloc(bytes_in_array);
14   double *B = (double*)malloc(bytes_in_array);
15   double *C = (double*)malloc(bytes_in_array);
16
17   // Initialize vector values
18   for(int i=0; i<N; i++){
19     A[i] = 1.0;
20     B[i] = 2.0;
21   }
22
23   // Perform element-wise addition of vectors
24   for(int i=0; i<N; i++){
25     C[i] = A[i] + B[i];
26   }
27
28   // Check for correctness
29   for(int i=0; i<N; i++){
30     if(C[i] != 3.0){
31       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
32       exit(0);
33     }
34   }
35
36   printf("__SUCCESS__\n");
37
38 }
```

In the following slides, we'll show some of the basic features of OpenACC using this simple vector addition program.

NOTE: In order to ensure the "code images", such as the one to the left, are large enough to read, deallocation of arrays are not shown. [e.g. free(A); etc.]

Code found in directory

`quick_intro_openacc/vector_addition/0_serial/`

Parallel Loop vs Kernels Region

```
#pragma acc parallel [clause-list]
    // structured block
```

This fundamental construct starts parallel execution on the current accelerator device.

clause-list is one of the following:

```
async[(int-expr)]
wait[(int-expr-list)]
num_gangs(int-expr)
num_workers(int-expr)
vector_length(int-expr)
device_type(device-type-list)
if(condition)
reduction(operator:var-list)
copy(var-list)
copyin(var-list)
copyout(var-list)
create(var-list)
present(var-list)
deviceptr(var-list)
private(var-list)
firstprivate(var-list)
default(none|present)
```

NOTE: The OpenACC `loop` construct can be combined with the `parallel` construct to create a `parallel loop` construct

```
#pragma acc kernels [clause-list]
 // structured block
```

This construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the current accelerator device

clause-list is one of the following:

```
async[(int-expr)]
wait[(int-expr-list)]
num_gangs(int-expr)
num_workers(int-expr)
vector_length(int-expr)
device_type(device-type-list)
if(condition)
copy(var-list)
copyin(var-list)
copyout(var-list)
create(var-list)
present(var-list)
deviceptr(var-list)
default(none|present)
```

For more details, see the OpenACC Specification.

Parallel Loop vs Kernels Region (Usage)

Two ways to offload structured blocks of code to GPU.

```
#pragma acc parallel loop
for(int i=0; i<N; i++){
  C[i] = A[i] + B[i];
}
```

Using the `parallel loop` directive, the programmer tells compiler this loop can be parallelized.

- Programmer knows what code is doing (e.g. understands loop bounds)

```
#pragma acc kernels
{
for(int i=0; i<N; i++){
  C[i] = A[i] + B[i];
}
}
```

Using the `kernels` directive, the compiler decides whether or not the loop can be parallelized.

- Can wrap around larger regions of code (e.g. containing many loops)

Example: Vector Addition

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *A = (double*)malloc(bytes_in_array);
14   double *B = (double*)malloc(bytes_in_array);
15   double *C = (double*)malloc(bytes_in_array);
16
17   // Initialize vector values
18   for(int i=0; i<N; i++){
19     A[i] = 1.0;
20     B[i] = 2.0;
21   }
22
23   // Perform element-wise addition of vectors
24   #pragma acc parallel loop
25   for(int i=0; i<N; i++){
26     C[i] = A[i] + B[i];
27   }
28
29   // Check for correctness
30   for(int i=0; i<N; i++){
31     if(C[i] != 3.0){
32       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
33       exit(0);
34     }
35   }
36
37   printf("__SUCCESS__\n");
38
39 }
```

Simply added parallel loop directive to serial code

NOTE: This could have been written instead as:

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i=0; i<N; i++){
        C[i] = A[i] + B[i];
    }
}
```

Code found in directory
`quick_intro_openacc/vector_addition/1_parallel_loop/`

Example: Vector Addition
Compile the Program

Make sure PGI compiler and CUDA module are loaded, then compile

- `module load cuda`
- `module swap xl pgi`
- `pgcc -acc -ta=tesla:cc60 -Minfo=accel -fast main.c -o run`

Compile for NVIDIA telsa GPU with compute capability 6.0

Give information about accelerator code

Tell compiler to look for #pragma acc directives

Optimization

Example: Vector Addition

Compile the Program

```
1  #include <stdlib.h>
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *A = (double*)malloc(bytes_in_array);
14   double *B = (double*)malloc(bytes_in_array);
15   double *C = (double*)malloc(bytes_in_array);
16
17   // Initialize vector values
18   for(int i=0; i<N; i++){
19     A[i] = 1.0;
20     B[i] = 2.0;
21   }
22
23   // Perform element-wise addition of vectors
24   #pragma acc parallel loop
25   for(int i=0; i<N; i++){
26     C[i] = A[i] + B[i];
27   }
28
29   // Check for correctness
30   for(int i=0; i<N; i++){
31     if(C[i] != 3.0){
32       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
33       exit(0);
34     }
35   }
36
37   printf("__SUCCESS__\n");
38
39 }
```

Output from compiler

```
$ pgcc -acc -ta=tesla:cc60 -Minfo=accel -fast main.c -o run
main:
     24, Accelerator kernel generated
         Generating Tesla code
         25, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     24, Generating implicit copyout(C[:1048576])
         Generating implicit copyin(B[:1048576],A[:1048576])
```

Important to look at compiler output to see what it is doing. Let's look in more detail…

At line 24 in the code, create GPU kernel from for loop, copy in vectors A and B (from CPU to GPU memory), and copy out vector C (from GPU to CPU memory).

The compiler decided that A, B, and C are used on the GPU, so memory needed to be allocated for these arrays on the device (this step is not actually shown in the output).

In addition, the compiler decided that only A and B were initialized on the CPU, so only data from A and B needed to be copied from host to device memory.

The compiler also decided that only array C was needed on the CPU after the kernel execution, so it only copied array C from device to host memory.

Example: Vector Addition
Run the Program

Submit via LSF script

```
#!/bin/bash -l
#BSUB -P CSC261SUMMITDEV
#BSUB -J vector_addition
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -R "span[ptile=1]"
#BSUB -n 1
#BSUB -W 10

numactl --physcpubind=0 nvprof -s ./run
```

To submit the script

```
$ bsub < submit.lsf
```

<OR>

Run on interactive node

First, request an interactive session on a compute node:

```
$ bsub -P CSC261SUMMITDEV -n 20 -W 60 -Is $SHELL
```

Then, run the job:

```
$ numactl --physcpubind=0 nvprof -s ./run
```

"nvprof -s" runs the NVIDIA Profiler

numactl -physcpubind=0 ensures you're only using cpu 0

(see Summitdev node diagram at end of slides)

Example: Vector Addition
Run the Program

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *A = (double*)malloc(bytes_in_array);
14   double *B = (double*)malloc(bytes_in_array);
15   double *C = (double*)malloc(bytes_in_array);
16
17   // Initialize vector values
18   for(int i=0; i<N; i++){
19     A[i] = 1.0;
20     B[i] = 2.0;
21   }
22
23   // Perform element-wise addition of vectors
24   #pragma acc parallel loop
25   for(int i=0; i<N; i++){
26     C[i] = A[i] + B[i];
27   }
28
29   // Check for correctness
30   for(int i=0; i<N; i++){
31     if(C[i] != 3.0){
32       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
33       exit(0);
34     }
35   }
36
37   printf("__SUCCESS__\n");
38
39 }
```

NVPROF output from running the program

```
$ nvprof -s ./run
==26009== NVPROF is profiling process 26009, command: ./run
__SUCCESS__
==26009== Profiling application: ./run
==26009== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 65.05%  555.88us         2  277.94us  266.56us  289.31us  [CUDA memcpy HtoD]
 29.28%  250.24us         1  250.24us  250.24us  250.24us  [CUDA memcpy DtoH]
  5.67%  48.417us         1  48.417us  48.417us  48.417us  main_24_gpu
```

nvprof – NVIDIA Profiler

CUDA memcpy HtoD
(copy data from host memory to device memory)

CUDA memcpy DtoH
(copy data from device memory to host memory)

main_24_gpu
(kernel execution on GPU – from line 24 in code)

Example: Vector Addition

Now, let's instead wrap both the initialization loop and compute loop in an `acc kernels` construct.

Example: Vector Addition

Wrap Init and Compute Loop in acc kernels construct

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double * A = (double*)malloc(bytes_in_array);
14   double * B = (double*)malloc(bytes_in_array);
15   double * C = (double*)malloc(bytes_in_array);
16
17 #pragma acc kernels
18 {
19   // Initialize vector values
20   for(int i=0; i<N; i++){
21     A[i] = 1.0;
22     B[i] = 2.0;
23   }
24
25   // Perform element-wise addition of vectors
26   for(int i=0; i<N; i++){
27     C[i] = A[i] + B[i];
28   }
29 }
30
31   // Check for correctness
32   for(int i=0; i<N; i++){
33     if(C[i] != 3.0){
34       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
35       exit(0);
36     }
37   }
38
39   printf("__SUCCESS__\n");
40
41 }
```

Compile the program

```
$ pgcc -acc -ta=tesla:cc60 -Minfo=accel -fast main.c -o run
main:
     17, Generating implicit copyout(A[:1048576],B[:1048576],C[:1048576])
     20, Complex loop carried dependence of A-> prevents parallelization
         Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         20, #pragma acc loop seq
     26, Complex loop carried dependence of B->,A-> prevents parallelization
         Loop carried dependence of C-> prevents parallelization
         Loop carried backward dependence of C-> prevents vectorization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         26, #pragma acc loop seq
```

At line 17, the compiler decides we need to copy arrays A, B, and C from GPU to CPU memory after kernel execution. There is no need to copy anything to the GPU since initialization occurs on CPU.

At lines 20 and 26, the compiler complains about “loop carried dependencies”, but there are clearly no iterations of the loops that depend on results from other iterations of the loops.

This is because *compiler cannot be certain that A, B, C are not aliases to other pointers*, but we can fix this with **restrict** keyword.

Example: Vector Addition

Wrap Init and Compute Loop in acc kernels construct

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *restrict A = (double*)malloc(bytes_in_array);
14   double *restrict B = (double*)malloc(bytes_in_array);
15   double *restrict C = (double*)malloc(bytes_in_array);
16
17 #pragma acc kernels
18 {
19   // Initialize vector values
20   for(int i=0; i<N; i++){
21     A[i] = 1.0;
22     B[i] = 2.0;
23   }
24
25   // Perform element-wise addition of vectors
26   for(int i=0; i<N; i++){
27     C[i] = A[i] + B[i];
28   }
29 }
30
31   // Check for correctness
32   for(int i=0; i<N; i++){
33     if(C[i] != 3.0){
34       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
35       exit(0);
36     }
37   }
38
39   printf("__SUCCESS__\n");
40
41 }
```

Compile the program

```
$ pgcc -acc -ta=tesla:cc60 -Minfo=accel -fast main.c -o run
main:
     17, Generating implicit copyout(A[:1048576],C[:1048576],B[:1048576])
     20, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         20, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     26, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         26, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

At line 17, the compiler decides we need to copy arrays A, B, and C from GPU to CPU memory after kernel execution. There is no need to copy anything to the GPU since initialization occurs on CPU.

Now at lines 20 and 26, the compiler creates kernel code from the loops to run on GPU.

The **restrict keyword** is an assurance from the programmer to the compiler that the pointer is not an alias.

Example: Vector Addition

Wrap Init and Compute Loop in acc kernels construct

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6   // Number of elements in arrays
7   unsigned int N = 1 << 20;
8
9   // Bytes in arrays
10  size_t bytes_in_array = N*sizeof(double);
11
12  // Allocate memory for arrays on host
13  double *restrict A = (double*)malloc(bytes_in_array);
14  double *restrict B = (double*)malloc(bytes_in_array);
15  double *restrict C = (double*)malloc(bytes_in_array);
16
17 #pragma acc kernels
18 {
19  // Initialize vector values
20  for(int i=0; i<N; i++){
21    A[i] = 1.0;
22    B[i] = 2.0;
23  }
24
25  // Perform element-wise addition of vectors
26  for(int i=0; i<N; i++){
27    C[i] = A[i] + B[i];
28  }
29 }
30
31  // Check for correctness
32  for(int i=0; i<N; i++){
33    if(C[i] != 3.0){
34      printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
35      exit(0);
36    }
37  }
38
39  printf("__SUCCESS__\n");
40
41 }
```

Run the program

```
$ nvprof -s ./run
==39885== NVPROF is profiling process 39885, command: ./run
__SUCCESS__
==39885== Profiling application: ./run
==39885== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.66%  758.86us         3  252.95us  250.15us  255.17us  [CUDA memcpy DtoH]
  5.76%  48.192us         1  48.192us  48.192us  48.192us  main_26_gpu
  3.59%  30.017us         1  30.017us  30.017us  30.017us  main_20_gpu
```

Notice that DtoH data transfers are 90% of time!

Also, recall from compiler output (previous slide) that all 3 arrays are transferred back to CPU from GPU. *But only C is actually needed!*

We can tell the compiler to only transfer array C back from GPU to CPU with data clauses!

Code found in directory

`quick_intro_openacc/vector_addition/2_kernels/`

Data Directives

Structured Data Regions
(Variables remain on GPU until end of data region)

```
#pragma acc data [clause-list]
{
  // Structured Block
}
```

Defines scalars, arrays, and subarrays to be allocated in the current device memory for the duration of the region

Clause list is one of the following

```
if (condition)
copy (var-list)
copyin (var-list)
copyout (var-list)
create (var-list)
present (var-list)
deviceptr (var-list)
```

Unstructured Data Regions
(Allows allocate/deallocate in separate routines)

```
#pragma acc enter data [clause-list]
```

Used to define scalars, arrays, and subarrays to be allocated in the current device memory for the duration of the program, or until an exit data directive deallocates the data.

Clause list is one of the following

```
if (condition)
async [(int-expr)]
wait [(int-expr-list)]
copyin (var-list)
create (var-list)
```

```
#pragma acc exit data [clause-list]
```

Clause list is one of the following

```
if (condition)
async [(int-expr)]
wait [(int-expr-list)]
copyout (var-list)
delete (var-list)
finalize
```

For more information, see the OpenACC Specification.

Example: Vector Addition

Let’s first look at how to use structured data regions.

Example: Vector Addition

Add Structured Data Region

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *restrict A = (double*)malloc(bytes_in_array);
14   double *restrict B = (double*)malloc(bytes_in_array);
15   double *restrict C = (double*)malloc(bytes_in_array);
16
17 #pragma acc data create(A[0:N], B[0:N]), copyout(C[0:N])
18 {
19 #pragma acc kernels
20 {
21   // Initialize vector values
22   for(int i=0; i<N; i++){
23     A[i] = 1.0;
24     B[i] = 2.0;
25   }
26
27   // Perform element-wise addition of vectors
28   for(int i=0; i<N; i++){
29     C[i] = A[i] + B[i];
30   }
31 }
32 }
33
34   // Check for correctness
35   for(int i=0; i<N; i++){
36     if(C[i] != 3.0){
37       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
38       exit(0);
39     }
40   }
41
42   printf("__SUCCESS__\n");
43
44 }
```

Compile the program

```
$ pgcc -acc -ta=tesla:cc60 -Minfo=accel -fast main.c -o run
main:
     17, Generating create(A[:N])
         Generating copyout(C[:N])
         Generating create(B[:N])
     22, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         22, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     28, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         28, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Line 17 tells the compiler to create vectors A, B, and C on the device, but only copy out vector C at the end of the data region.

Note: the curly braces define the lifetime of the data region where the arrays A, B, and C persist.

Code found in directory

`quick_intro_openacc/vector_addition/3_kernels_structData/`

Example: Vector Addition

Add Structured Data Region

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 int main()
 5 {
 6   // Number of elements in arrays
 7   unsigned int N = 1 << 20;
 8
 9   // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *restrict A = (double*)malloc(bytes_in_array);
14   double *restrict B = (double*)malloc(bytes_in_array);
15   double *restrict C = (double*)malloc(bytes_in_array);
16
17 #pragma acc data create(A[0:N], B[0:N]), copyout(C[0:N])
18 {
19 #pragma acc kernels
20 {
21   // Initialize vector values
22   for(int i=0; i<N; i++){
23     A[i] = 1.0;
24     B[i] = 2.0;
25   }
26
27   // Perform element-wise addition of vectors
28   for(int i=0; i<N; i++){
29     C[i] = A[i] + B[i];
30   }
31 }
32 }
33
34   // Check for correctness
35   for(int i=0; i<N; i++){
36     if(C[i] != 3.0){
37       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
38       exit(0);
39     }
40   }
41
42   printf("__SUCCESS__\n");
43
44 }
```

Run the program

```
$ nvprof -s ./run
==64582== NVPROF is profiling process 64582, command: ./run
__SUCCESS__
==64582== Profiling application: ./run
==64582== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 76.37%  249.99us         1  249.99us  249.99us  249.99us  [CUDA memcpy DtoH]
 14.60%  47.777us         1  47.777us  47.777us  47.777us  main_28_gpu
  9.03%  29.568us         1  29.568us  29.568us  29.568us  main_22_gpu
```

Now the DtoH data transfers only account for 76% of the time.

Keep in mind that data transfers can be very costly, so minimizing these transfers is often very important.

Also notice that this is a very simple program, where the GPU kernels have little computation to perform, so the data transfers will dominate. In general, this will be problem dependent.

Example: Vector Addition

Now let's look at how to use unstructured data regions.

Example: Vector Addition

Add Unstructured Data Region

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *restrict A = (double*)malloc(bytes_in_array);
14   double *restrict B = (double*)malloc(bytes_in_array);
15   double *restrict C = (double*)malloc(bytes_in_array);
16 #pragma acc enter data create(A[0:N], B[0:N], C[0:N])
17
18 #pragma acc kernels present(A[0:N], B[0:N], C[0:N])
19 {
20   // Initialize vector values
21   for(int i=0; i<N; i++){
22     A[i] = 1.0;
23     B[i] = 2.0;
24   }
25
26   // Perform element-wise addition of vectors
27   for(int i=0; i<N; i++){
28     C[i] = A[i] + B[i];
29   }
30 }
31 #pragma acc exit data copyout(C[0:N])
32
33   // Check for correctness
34   for(int i=0; i<N; i++){
35     if(C[i] != 3.0){
36       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
37       exit(0);
38     }
39   }
40
41   printf("__SUCCESS__\n");
42
43 }
```

Allocate memory on the GPU for arrays A, B, and C. Each of these arrays will start at element 0 and contain N elements. This allocation of memory on the host and device could have been performed in a separate function, which is a benefit of using unstructured data regions.

Tell compiler that arrays A, B, and C are already on the device, so there is not need to create them or copy them in.

Exit the scope of the arrays A, B, and C, and copy the results of C from the GPU to the CPU memory.

Code found in directory

`quick_intro_openacc/vector_addition/4_kernels_unstructData/`

Example: Vector Addition

Add Unstructured Data Region

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6    // Number of elements in arrays
7    unsigned int N = 1 << 20;
8
9    // Bytes in arrays
10   size_t bytes_in_array = N*sizeof(double);
11
12   // Allocate memory for arrays on host
13   double *restrict A = (double*)malloc(bytes_in_array);
14   double *restrict B = (double*)malloc(bytes_in_array);
15   double *restrict C = (double*)malloc(bytes_in_array);
16 #pragma acc enter data create(A[0:N], B[0:N], C[0:N])
17
18 #pragma acc kernels present(A[0:N], B[0:N], C[0:N])
19 {
20   // Initialize vector values
21   for(int i=0; i<N; i++){
22     A[i] = 1.0;
23     B[i] = 2.0;
24   }
25
26   // Perform element-wise addition of vectors
27   for(int i=0; i<N; i++){
28     C[i] = A[i] + B[i];
29   }
30 }
31 #pragma acc exit data copyout(C[0:N])
32
33   // Check for correctness
34   for(int i=0; i<N; i++){
35     if(C[i] != 3.0){
36       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
37       exit(0);
38     }
39   }
40
41   printf("__SUCCESS__\n");
42
43 }
```

Compile the program

```
pgcc -acc -ta=tesla:cc60 -Minfo=accel -fast -c main.c
main:
     16, Generating enter data create(B[:N],C[:N],A[:N])
     18, Generating present(A[:N],C[:N],B[:N])
     21, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     27, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         27, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     31, Generating exit data copyout(C[:N])
```

Line 16 tells the compiler to create vectors A, B, and C on the device.

Line 18 tells the compiler that A, B, and C are present on GPU.

At lines 21 and 27, the compiler generates kernels from the two for loops.

Line 31 tells the compiler to exit the unstructured data region and copy out array C (from GPU to CPU)

Note: The scope of the data region is unstructured, so the allocation of data on the host and device could have performed in a separate function.

Example: Vector Addition

Add Unstructured Data Region

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6   // Number of elements in arrays
7   unsigned int N = 1 << 20;
8
9   // Bytes in arrays
10  size_t bytes_in_array = N*sizeof(double);
11
12  // Allocate memory for arrays on host
13  double *restrict A = (double*)malloc(bytes_in_array);
14  double *restrict B = (double*)malloc(bytes_in_array);
15  double *restrict C = (double*)malloc(bytes_in_array);
16 #pragma acc enter data create(A[0:N], B[0:N], C[0:N])
17
18 #pragma acc kernels present(A[0:N], B[0:N], C[0:N])
19 {
20   // Initialize vector values
21   for(int i=0; i<N; i++){
22     A[i] = 1.0;
23     B[i] = 2.0;
24   }
25
26   // Perform element-wise addition of vectors
27   for(int i=0; i<N; i++){
28     C[i] = A[i] + B[i];
29   }
30 }
31 #pragma acc exit data copyout(C[0:N])
32
33   // Check for correctness
34   for(int i=0; i<N; i++){
35     if(C[i] != 3.0){
36       printf("Error: Element C[%d] = %f instead of 3.0\n", i, C[i]);
37       exit(0);
38     }
39   }
40
41   printf("__SUCCESS__\n");
42
43 }
```

```
==38217== NVPROF is profiling process 38217, command: ./run
==38217== Profiling application: ./run
==38217== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 83.18%  385.22us         1  385.22us  385.22us  385.22us  [CUDA memcpy DtoH]
 10.36%  48.001us         1  48.001us  48.001us  48.001us  main_27_gpu
  6.46%  29.920us         1  29.920us  29.920us  29.920us  main_21_gpu
```

Run the program

On to the Mandelbrot Program

Now we'll use what you've learned to accelerate your Mandelbrot program using OpenACC. Along the way, we'll introduce new directives and library routines as they are needed.

Step 1: Create Serial Program

Create a serial version of the Mandelbrot program that you will then accelerate using OpenACC.

NOTE: Each group will likely see different performance results based on the parameters they choose. For instance, setting an iteration count to a higher value will increase the time spent computing, which will change the ratio of compute time versus data transfer times.

Step 2: OpenACC Targeting Single GPU

Add OpenACC directives to your serial version of the Mandelbrot program to offload the pixel computation to a single GPU.

In addition to the directives you encountered working with the vector addition examples in the previous slides, you might also need to use the `#pragma acc routine` directive.

Used to tell the compiler to compile a given procedure for an accelerator as well as the host.

```
#pragma acc routine [clause-list]
```

clause-list can be

```
gang
worker
vector
seq
...
```

For our purposes, we will use the `seq` clause.

It should appear before a function definition or prototype.

For more information, please refer to the OpenACC specification.

Aside: NVPROF & PGPROF

In addition to the text-based profiling results, you can also use the NVIDIA Visual Profiler or PGPROF to see a visual representation of your profile.

NVIDIA Visual Profiler - https://developer.nvidia.com/nvidia-visual-profiler

PGI Profiler - https://www.pgroup.com/resources/pgprof-quickstart.htm

These two profilers are essentially the same, but there are different methods to using them with Summitdev.

In both cases, first generate an output file on Summitdev (run as before but add -o option and output file name)

```
nvprof -s -o step2.nvvp ./run
```

Run as before but add -o option and output file name

The next two slides will outline the two different methods to view the output file using either the NVIDIA Visual Profiler or PGPROF. (You can choose which profiler to use)

Aside: NVPROF & PGPROF

NVIDIA Visual Profiler - https://developer.nvidia.com/nvidia-visual-profiler

To use the NVIDIA Visual Profiler, you will first need to install it on your local machine. Then you can visualize your profiler results as follows:

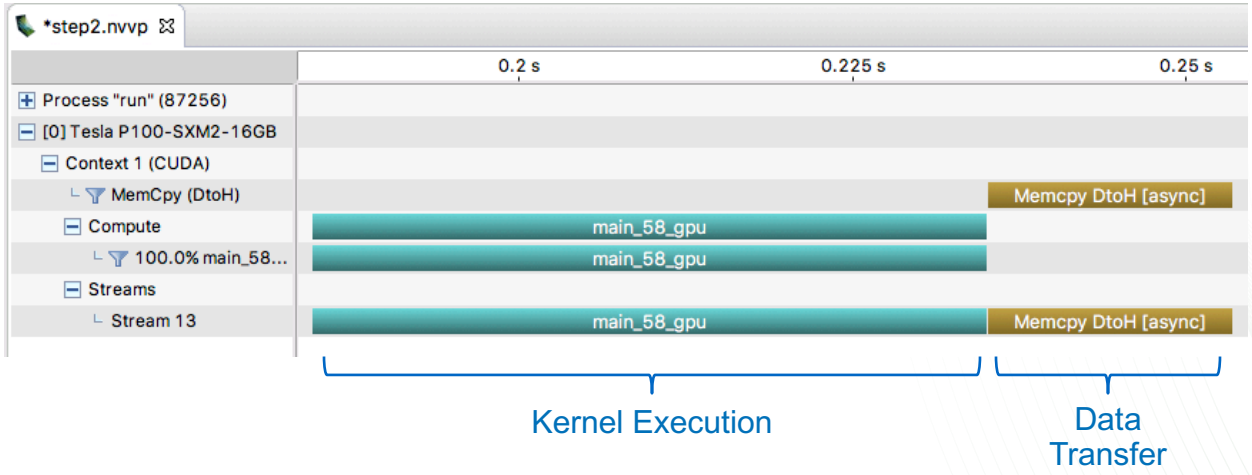
1. Generate output file from Summitdev (run as before but add -o option and output file name)
   `nvprof -s -o step2.nvvp ./run`

   Run as before but add -o option and output file name

2. scp the output file from Summitdev to your local machine.

3. Start the NVIDIA Visual Profiler on your local machine

4. Open the output file using the profiler



Aside: NVPROF & PGPROF

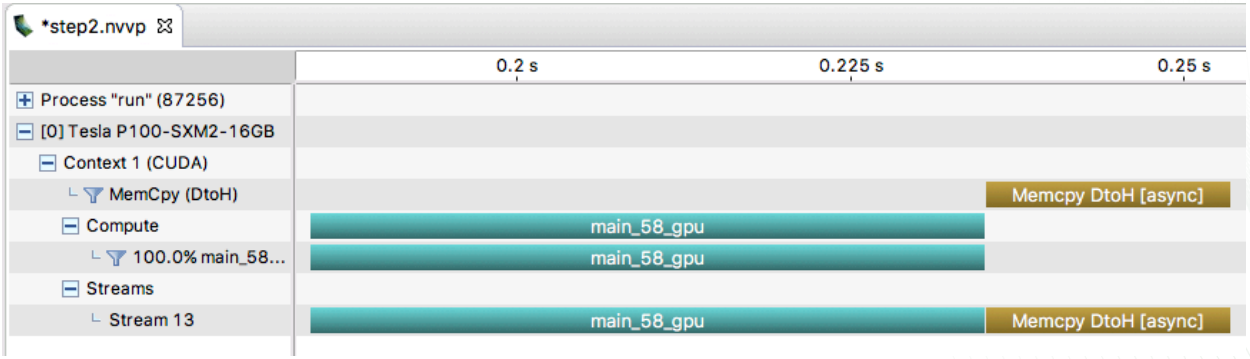
PGI Profiler - https://developer.nvidia.com/nvidia-visual-profiler

You can use the PGI Profiler directly from Summitdev, but there are certain requirements.

- You will need to have software on your local machine to allow X windows
- You will need to pass the -X flag to ssh when logging in to both home and Summitdev to enable X11 forwarding.

Then, you can use the profiler as follows:

1. Generate output file from Summitdev (run as before but add -o option and output file name)
   `nvprof -s -o step2.nvvp ./run`
   Run as before but add -o option and output file name
2. On Summitdev, issue the command 'pgprof' (this will open GUI on local machine)
3. Open the output file using the profiler (you will be able to browse files on Summitdev)



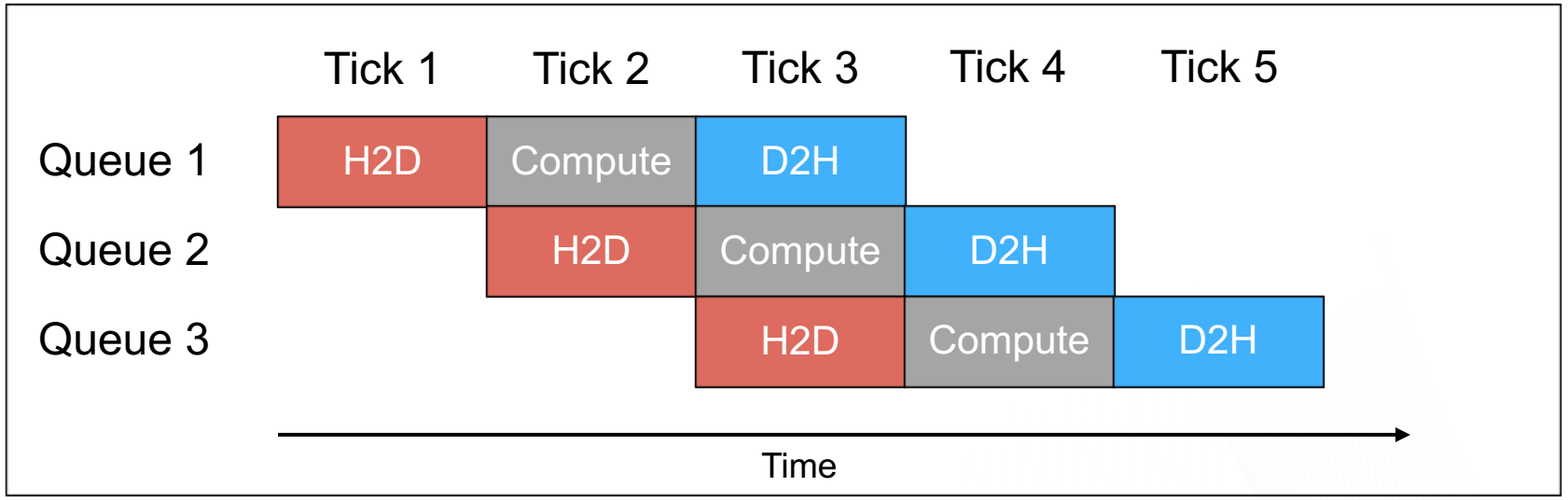
NOTE: you still generate the output file using nvprof, even though you will visualize it using PGPROF.

Step 3: Pipelining

As previously mentioned, data transfers can be expensive, but we can minimize the cost by overlapping data transfers with computation.

The idea is to break up the image array into blocks (or tiles) so that the sequence of H2D, compute, and D2H for each block is handled by a separate work queue.

Assume each step (H2D, Compute, D2H) takes the same amount of time (1 “Tick”).



In this example, by breaking work up into 3 separate work queues, we can perform the same 9 steps in 5 “Ticks” instead of 9!

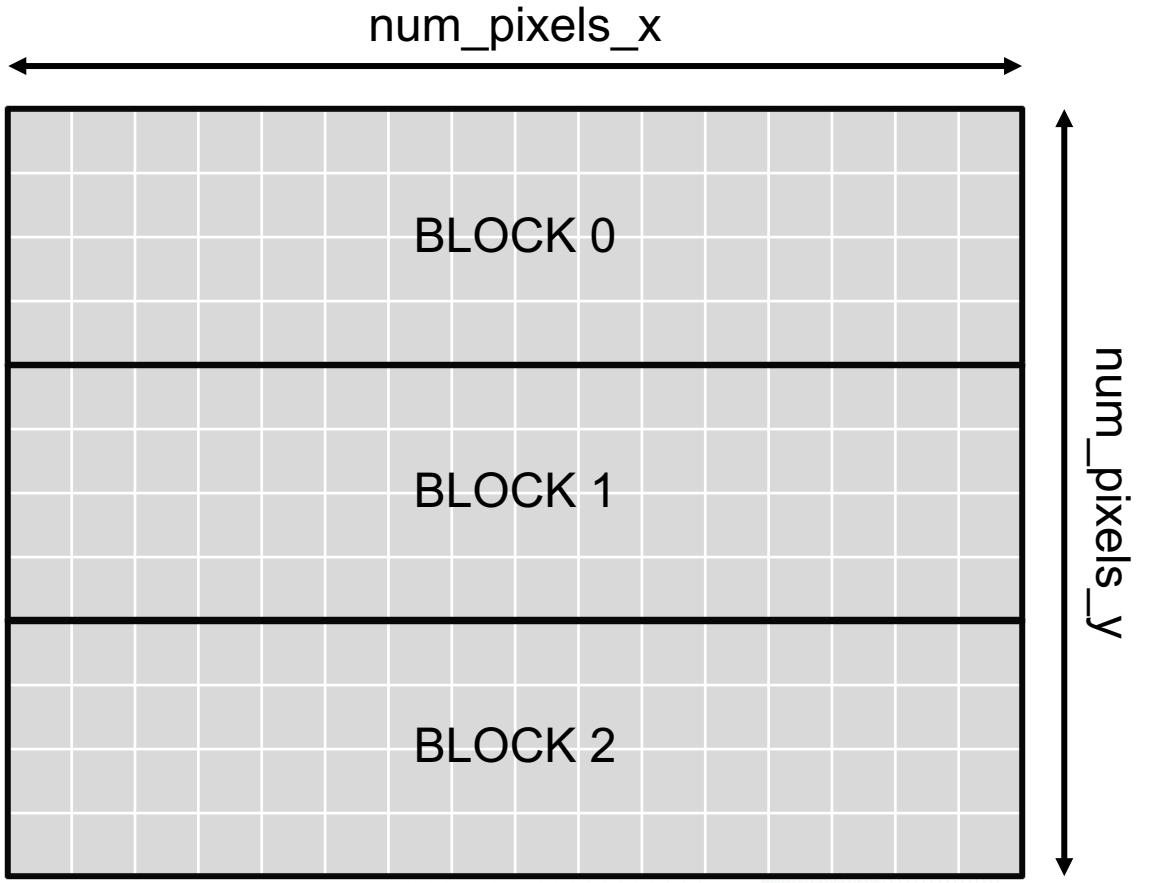
NOTE: You should only have D2H transfers in your Mandelbrot program whereas the cartoon above shows pipelining in a general case.

Step 3a: Pipelining – Add Block/Tile Loop

First, add a loop to break up the computation of pixels into blocks (or tiles).

```
#pragma acc parallel loop
for(ny=0; ny<num_pixels_y; ny++){
   for(nx=0; nx<num_pixels_x; nx++){
      // calculate pixel
   }
}
```

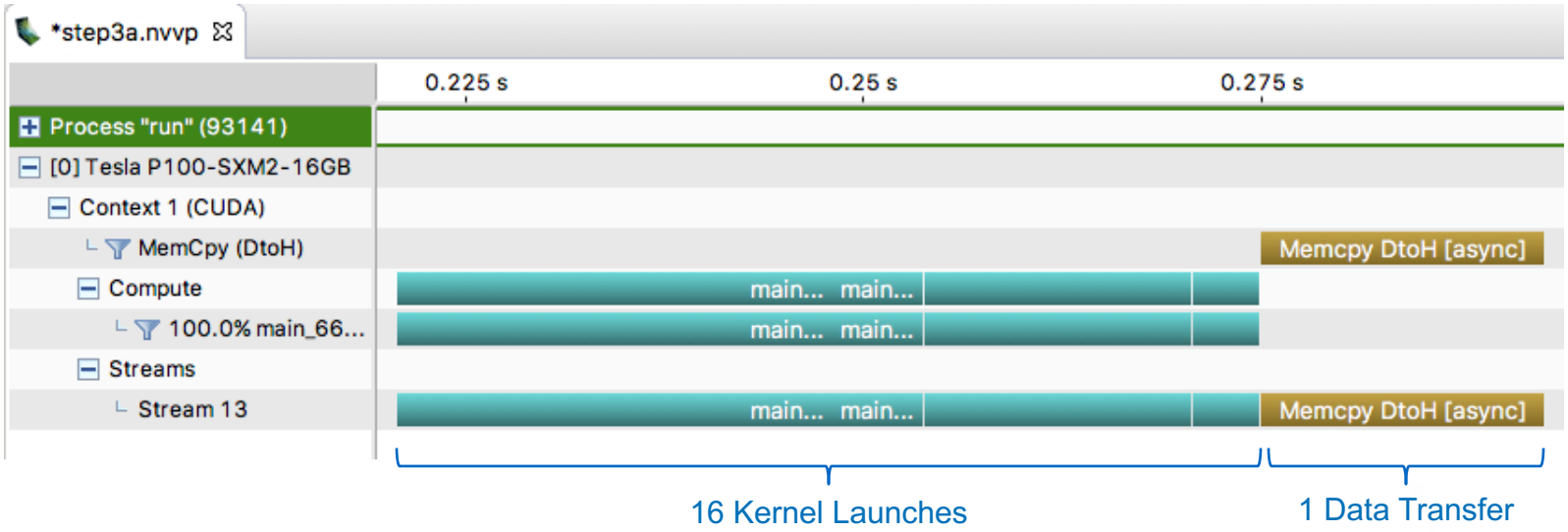
```
for(block=0; block<num_blocks; block++){
   start_y = …
   stop_y = …
   #pragma acc parallel loop
   for(ny=start_y; ny<stop_y; ny++){
      for(nx=0; nx<num_pixels_x; nx++){
         // calculate pixel
      }
   }
}
```



By blocking in only the y-dimension, we preserve contiguous memory segments for each block.

Step 3a: Pipelining – Add Block/Tile Loop

The profile should look similar to the following:



Although it is not resolved in this figure, there are 16 separate kernels because the program divided the image array into 16 blocks.

When actually using the visual profiler, you would be able to hover over the blue bar to see the individual kernel launches.

Note that we are still currently transferring the entire image array from DtoH after all pixels have been computed.

Step 3b: Pipelining – Break Up Data Transfers

Now we will send the data from GPU to CPU for each block instead of for the entire image array at the end. This will allow us to overlap data transfers with computation in the upcoming steps.

To do so, you can use the `#pragma acc update` clause inside your structured or unstructured data region.

Used during the lifetime of accelerator data to update all or part of local variables or arrays with values from the corresponding data in device memory, or to update all or part of device variables or arrays with values from the corresponding local memory.

```
#pragma acc update [clause-list]
```

clause-list can be

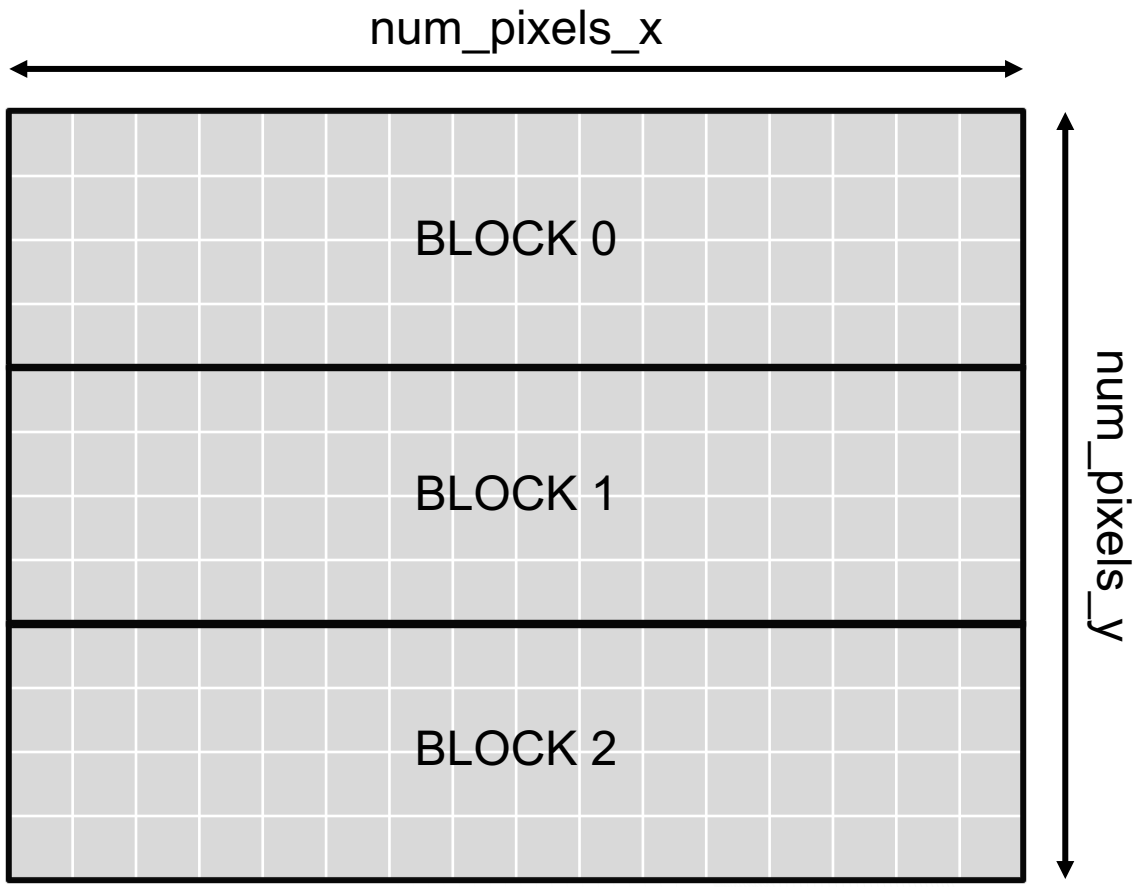
```
async [(int-expr)]
wait [(int-expr-list)]
self(var_list)
host(var_list)
device(var-list)
…
```

For more information, please refer to the OpenACC specification.

Step 3b: Pipelining – Break Up Data Transfers

```
for(block=0; block<num_blocks; block++){
    start_y = …
    stop_y = …
    #pragma acc parallel loop
    for(ny=start_y; ny<stop_y; ny++){
        for(nx=0; nx<num_pixels_x; nx++){
            // calculate pixel
        }
    }
}
```

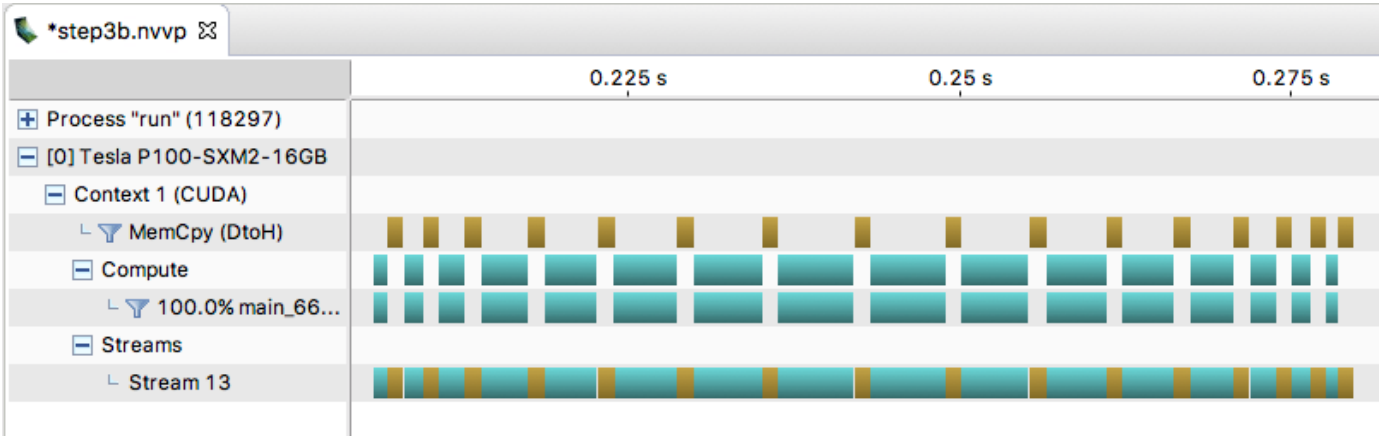
Add the `#pragma acc update` after the pixel loops, but before the end of the block loop.



Now, instead of computing all the pixels and then copying the entire image array from the GPU to CPU, you should be computing the pixels for a block, copying the data for that block from GPU to CPU, then moving on to the next block and doing the same…

Step 3b: Pipelining – Break Up Data Transfers

The profile should look similar to the following:



Now we have also broken up the data transfer per block.

- For each block, we are computing the pixels and transferring the data from D2H.
  - But each sequence of “compute + data transfer” is serialized with the others. In the next step, we will add asynchronous behavior.

Step 3c: Pipelining – Add asyc & wait Clauses

Now add `async` clause to your `#pragma acc parallel loop` or `#pragma acc kernels` construct and to your `#pragma acc update` construct.

The `async` clause may appear on a parallel or kernels construct, or an enter data, exit data, update or wait directive.

When there is no `async` clause on a compute or data construct, the local thread will wait until the compute construct or data operations for the current device are complete before executing any of the code that follows.

When there is an `async` clause, the parallel or kernels region or data operations may be processed asynchronously while the local thread continues with the code following the construct or directive.

Also add a `wait` directive after your `update` to ensure that all asynchronous queues finish transferring their results to the CPU before writing to the image.

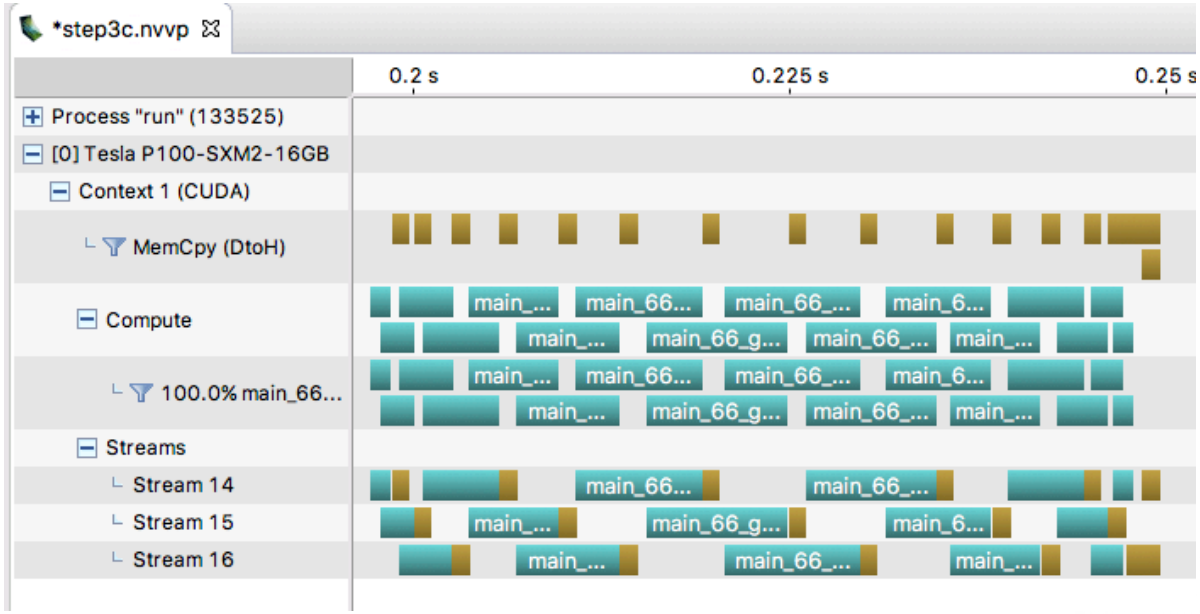
Causes the local thread to wait for completion of asynchronous operations on the current device, such as an accelerator parallel or kernels region or an update directive, or causes one device activity queue to synchronize with one or more other activity queues on the current device.

```
#pragma acc wait
```

For more information, please refer to the OpenACC specification.

Step 3c: Pipelining – Add asyc & wait Clauses

The profile should look similar to the following:



Now we are overlapping the data transfers with compute regions and even overlapping separate compute regions.

Step 4: Targeting Multiple GPUs

Now let's target multiple GPUs to compute the pixel values. This can be accomplished in different ways, such as using OpenMP or MPI.

To specify a particular GPU, you can use the `acc_set_device_num` and `acc_get_num_devices` routines. In order to use these routines, you will need to `#include <openacc.h>`

- *The* `acc_set_device_num` *routine tells the runtime which device to use among those attached of the given type for accelerator compute or data regions in the current thread and sets the value of acc-device-num-var.*

```
void acc_set_device_num(int deviceID, acc_device_t deviceType);
```

- *The* `acc_get_num_devices` *routine returns the number of accelerator devices of the given type to the host.*

```
void acc_set_device_num(int deviceID, acc_device_t deviceType);
```

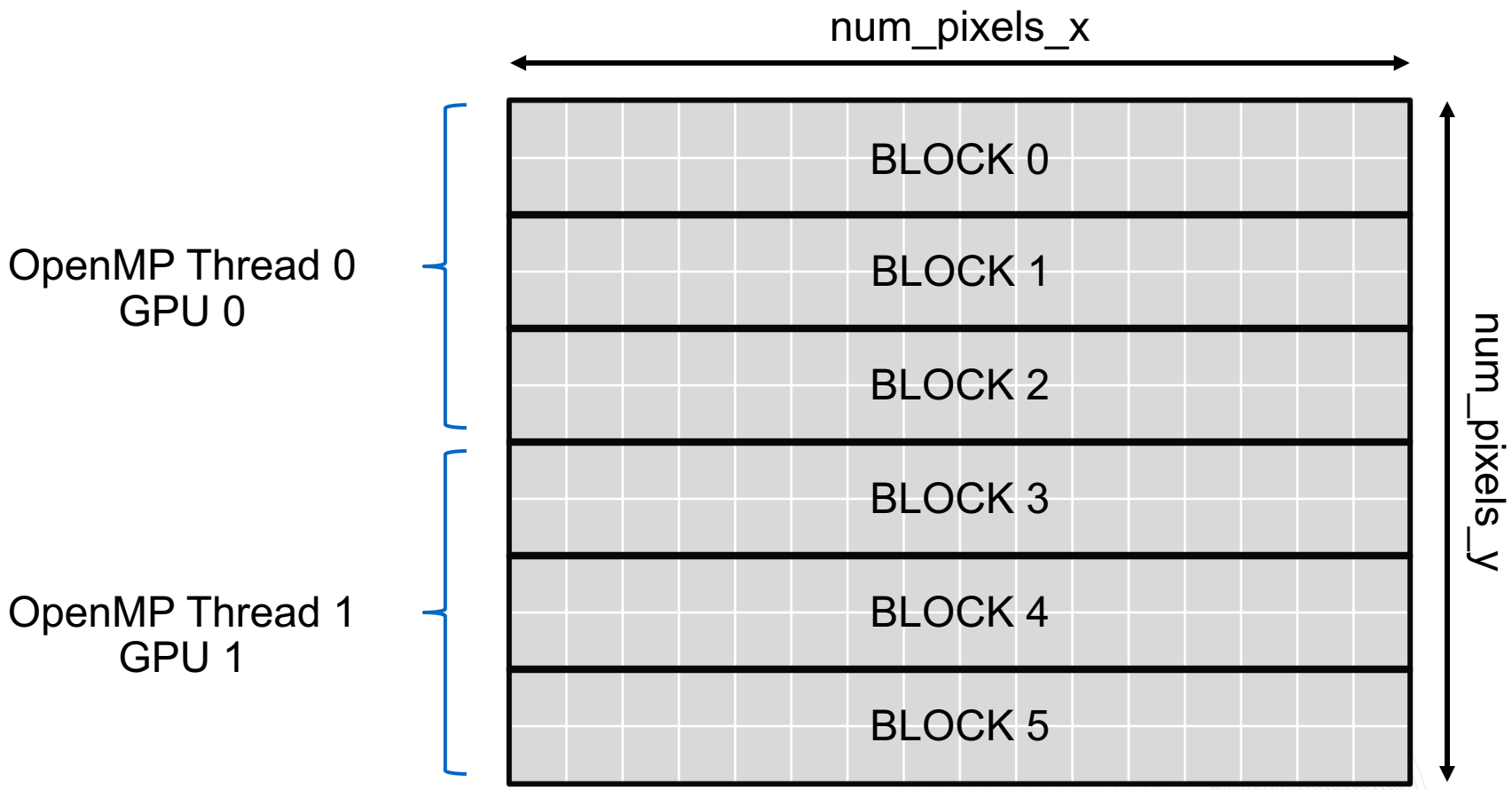
`deviceID:`	ID of device being targeted (0-3 on Summitdev)
`deviceType:`	Type of accelerator (`acc_device_nvidia` on Summitdev)

For more information, please refer to the OpenACC specification.

Step 4: Targeting Multiple GPUs

Example – Using OpenMP

Assume we have 2 OpenMP threads. We could add an OpenMP parallel loop directive to the block loop so that each OpenMP thread gets half the blocks.

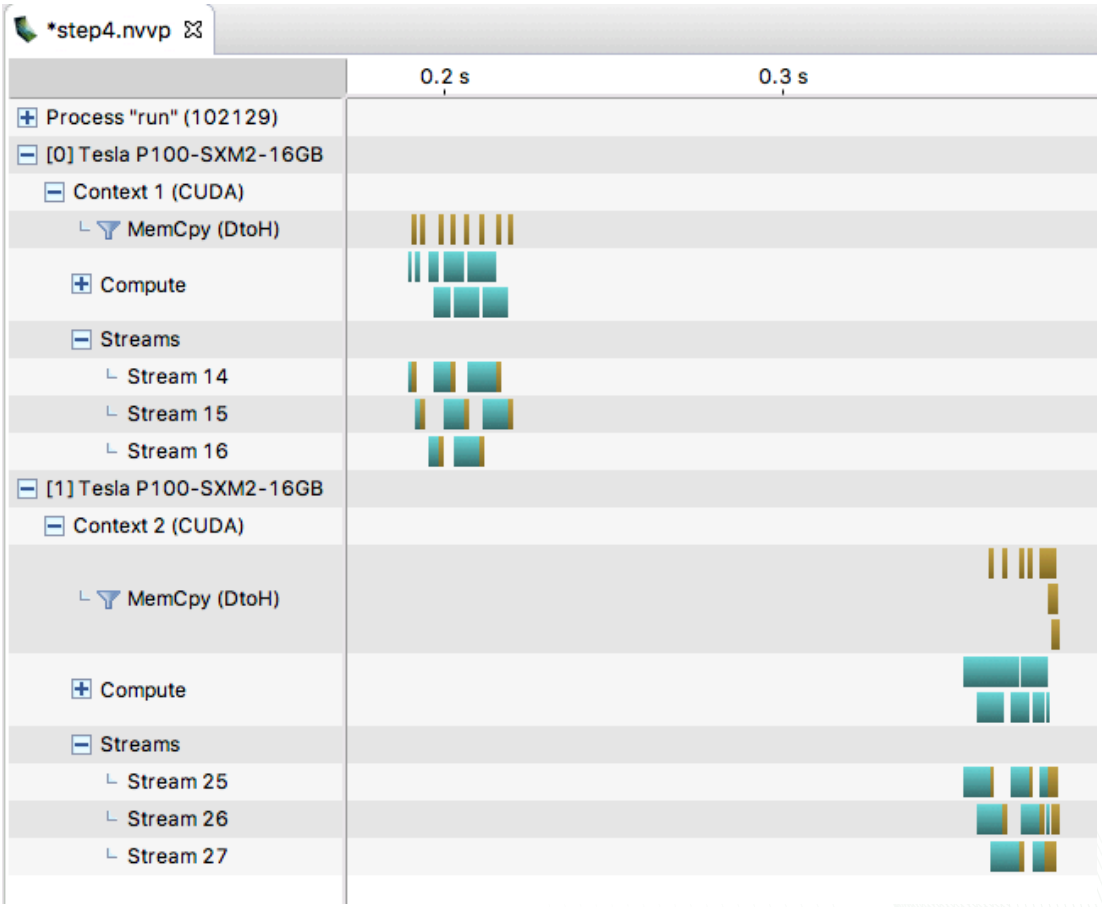


Step 4: Targeting Multiple GPUs

The profile should look similar to the following:

GPU 0
OpenMP Thread 0

GPU 1
OpenMP Thread 1



Note: This program used 2 OpenMP threads to target 2 GPUs (4 GPUs was not easy to show with screenshot)

More Steps!

At this point, instead of giving other specific steps, your group should choose from the following possibilities (or make up your own):

- Target multiple GPUs using only MPI (if you used OpenMP in the previous step)
- Try to find optimal number of blocks/tiles to use
- Try to improve load balancing in your MPI or OpenMP version
  - Although all blocks have the same amount pixels, some have more iterations to perform than others. This causes a load imbalance (among ranks or threads).
- Make a prettier picture
  - There are algorithms that can make many different images
  - Add color to you image (if you have not already done so)
- Make a time sequence where you zoom in on the Mandelbrot set to show its self-similarity.
  - Because each “timestep” does not depend on previous steps, you can parallelize the creation of this sequence.