MODULE SEVEN: ASYNCHRONOUS PROGRAMMING

Speaker, Date

OpenACC

More Science, Less Programming

MODULE OVERVIEW

Topics to be covered

- Definition of Asynchronous Programming (and pipelining)
- OpenACC Async and Wait clauses
- OpenACC function routine directive
- Implementing asynchronous behavior with OpenACC

ASYNCHRONOUS PROGRAMMING

ASYNCHRONOUS PROGRAMMING

Programming such that two or more unrelated operations can occur independently or even at the same time without immediate synchronization.

Real World Examples:

- Cooking a Meal: Boiling potatoes while preparing other parts of the dish.
- Three students working on a project on George Washington, one researches his early life, another his military career, and the third his presidency.
- Automobile assembly line: each station adds a different part to the car until it is finally assembled.

ASYNCHRONOUS EXAMPLE

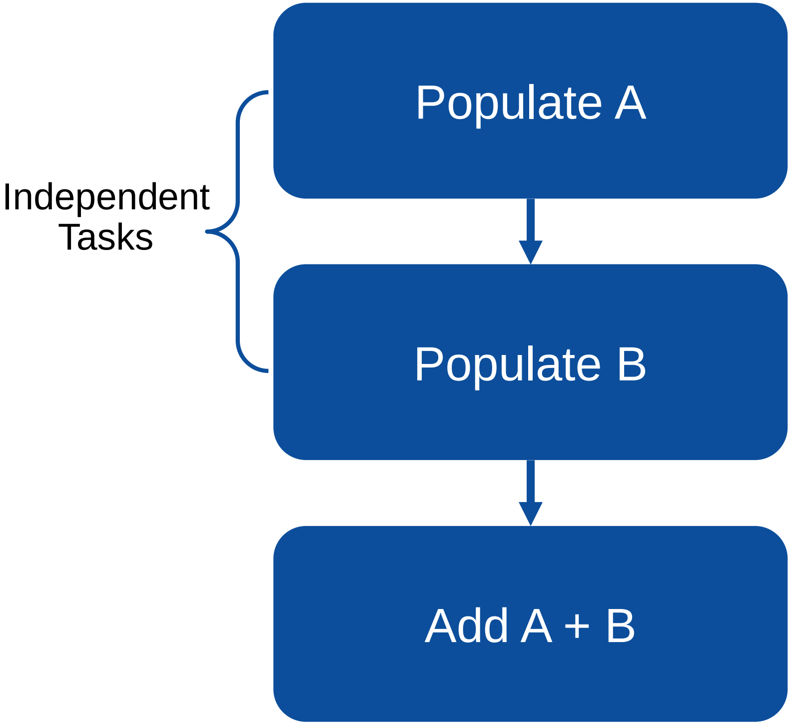
- I want to populate two arrays, A and B, with data, then add them together. This requires 3 distinct operations.

1. Populate A
2. Populate B
3. Add A + B

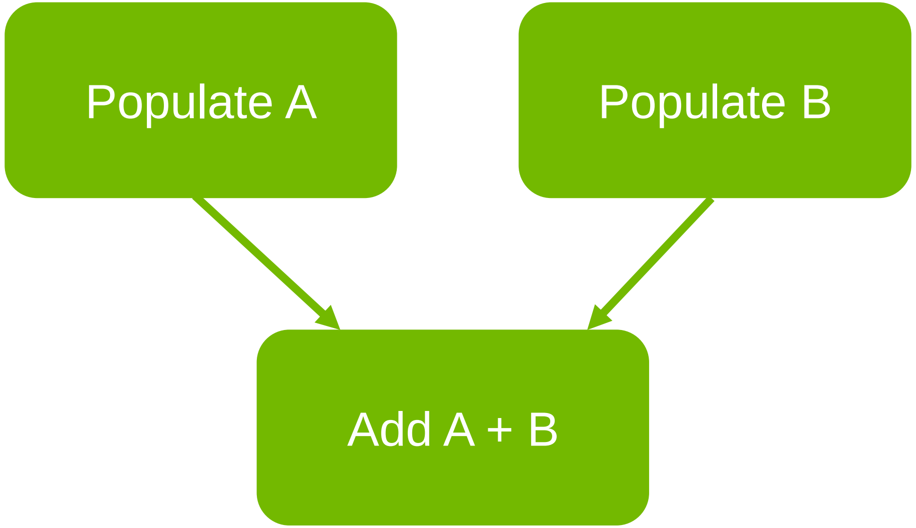
- Tasks 1 and 2 are independent, but task 3 is dependent on both.

ASYNCHRONOUS EXAMPLE

SYNCHRONOUS



ASYNCHRONOUS



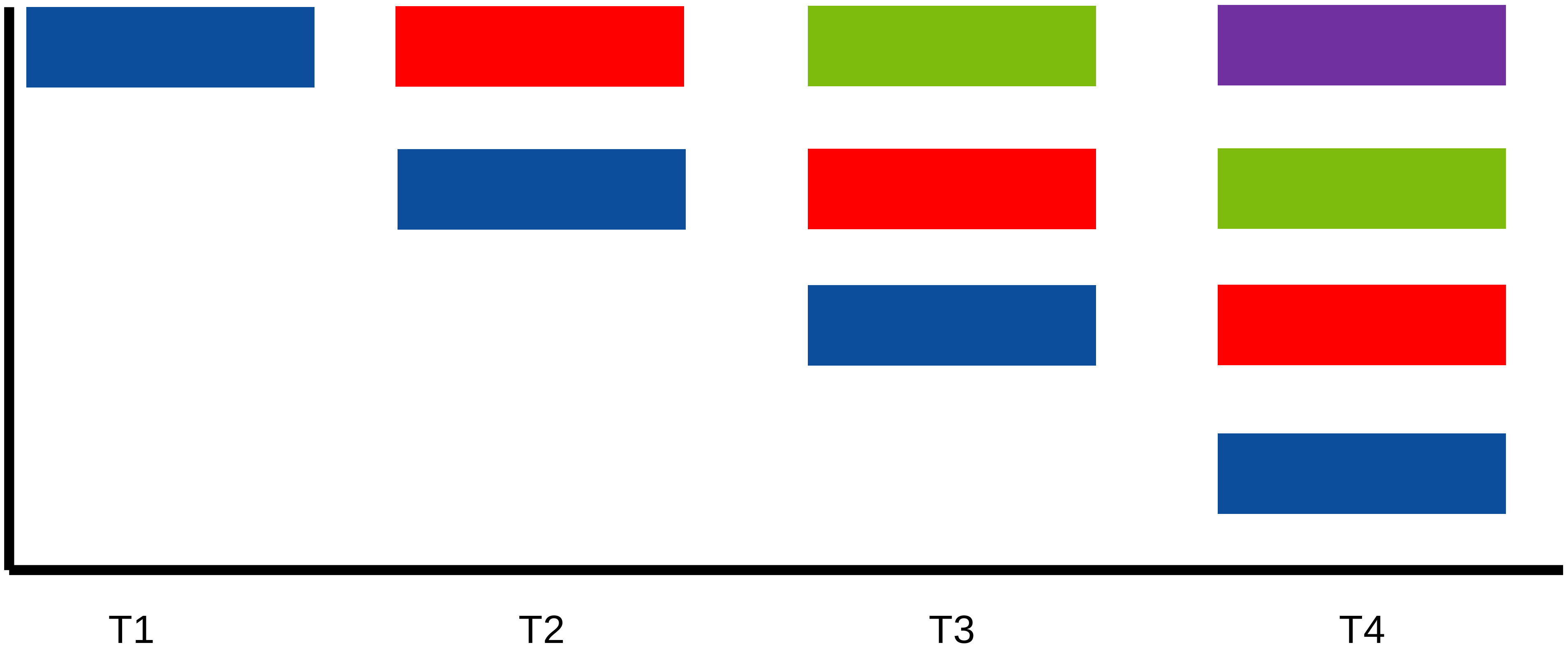
In order to do tasks asynchronously, they must be completely independent of each other.

ASYNCHRONOUS PIPELINING

- Very large operations may frequently be broken into smaller parts that may be performed independently and repeatedly.
- Pipeline Stage -A single step, which is frequently limited to 1 part at a time

Photo by Roger Wollstadt, used via Creative Commons

ASYNCHRONOUS PIPELINING



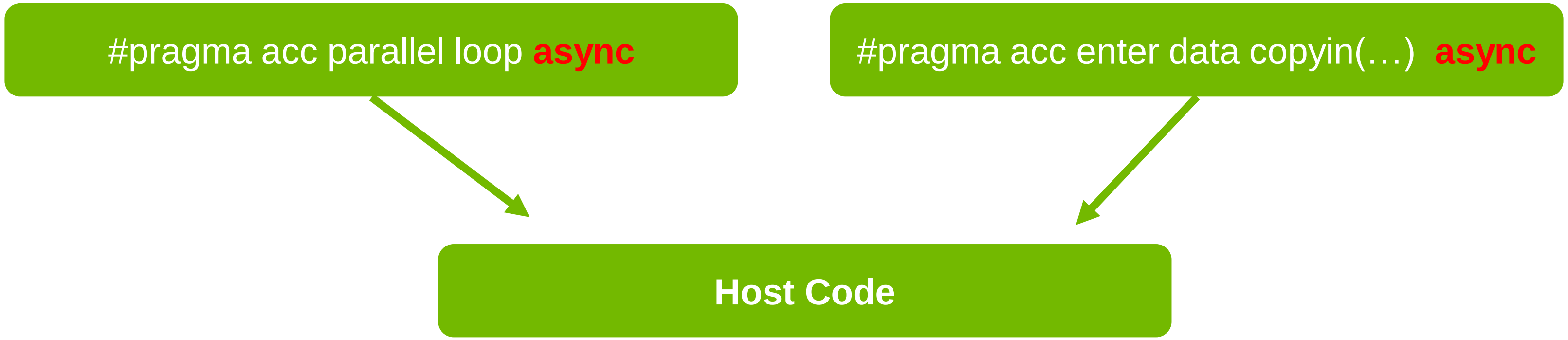
OPENACC ASYNC AND WAIT CLAUSE

ASYNC CLAUSE

```
#pragma acc <directive> <clauses> async(n)
```

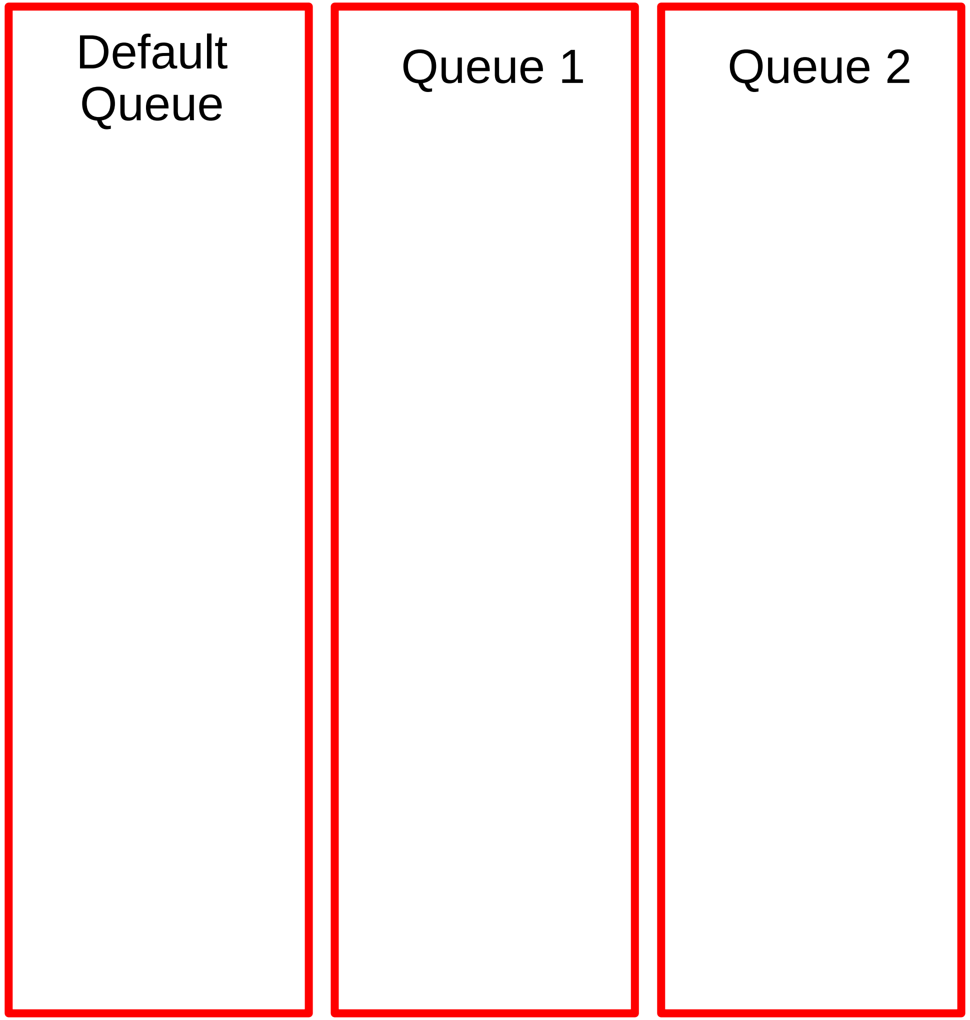
Async launches work and then returns to host immediately

- This allows some operations to run concurrently
- For example, a device compute construct can execute at the same time as host computation
- Another example, a data transfer can occur at the same time as device execution
- If *n* is not specified, the *default queue* is used



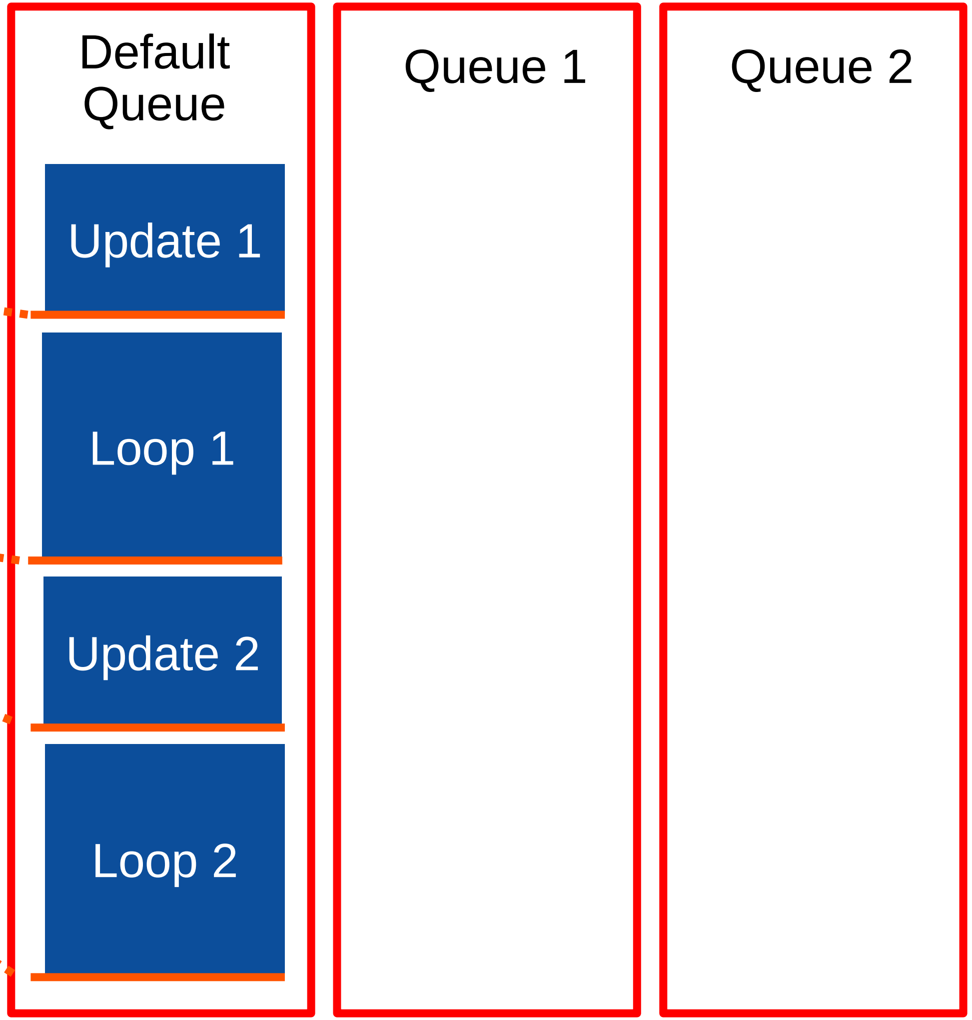
ASYNC EXAMPLE

- When thinking about running asynchronous operations, we can think of them as running in different “**queues**”
- Without any async queue numbers only a single (default) queue is used
- There’s an unlimited number of queues*, but initializing each queue may cause an overhead
- It is usually best to use as limit yourself to a small number of queue that are reused



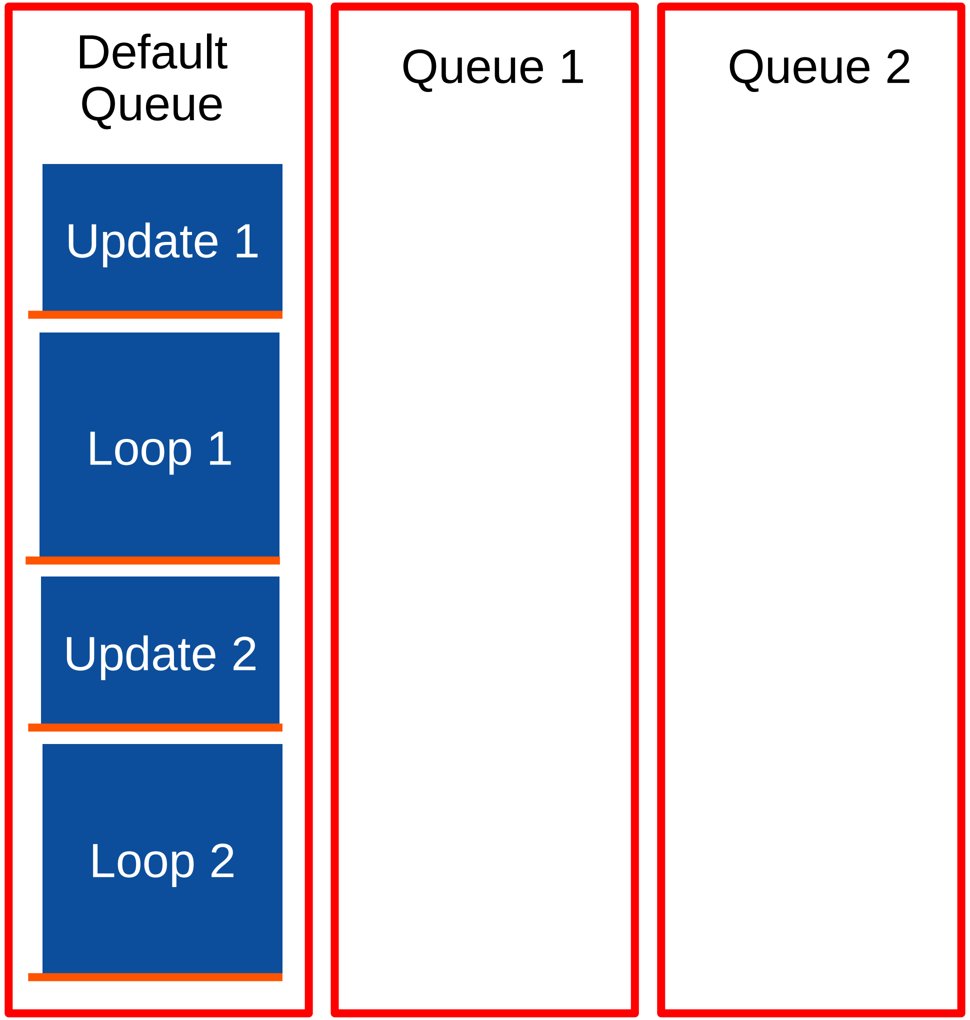
ASYNC EXAMPLE

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100])

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



ASYNC EXAMPLE

- Each operation waits for the previous to finish
- This is the default OpenACC behavior



ASYNC EXAMPLE

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async(1)
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async(2)

#pragma acc parallel loop async(2)
for( i = 100; i < 200; i++ )
    X[i] = ...
```

Since these two async clauses have different numbers, they can run simultaneously

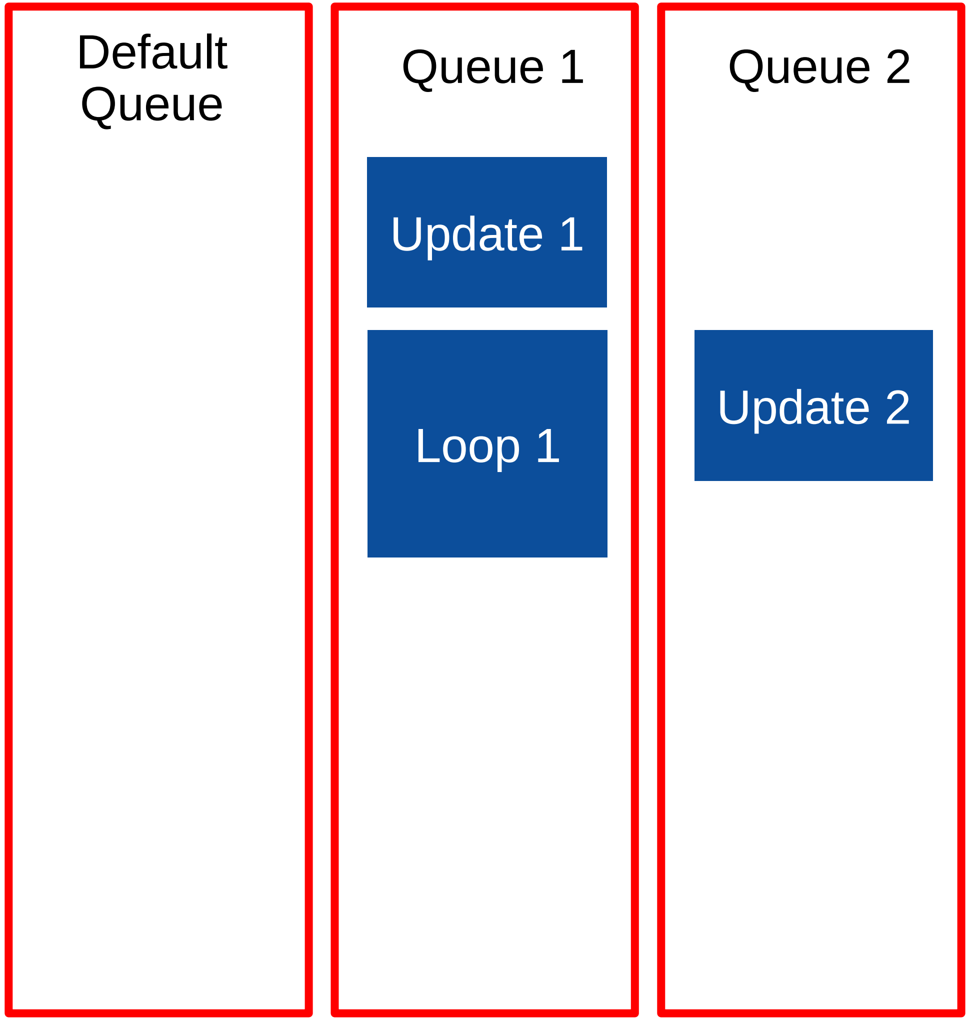
ASYNC EXAMPLE

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async(1)
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async(2)

#pragma acc parallel loop async(2)
for( i = 100; i < 200; i++ )
    X[i] = ...
```

The two operations are dependent, so they must go in the same queue.

ASYNC EXAMPLE

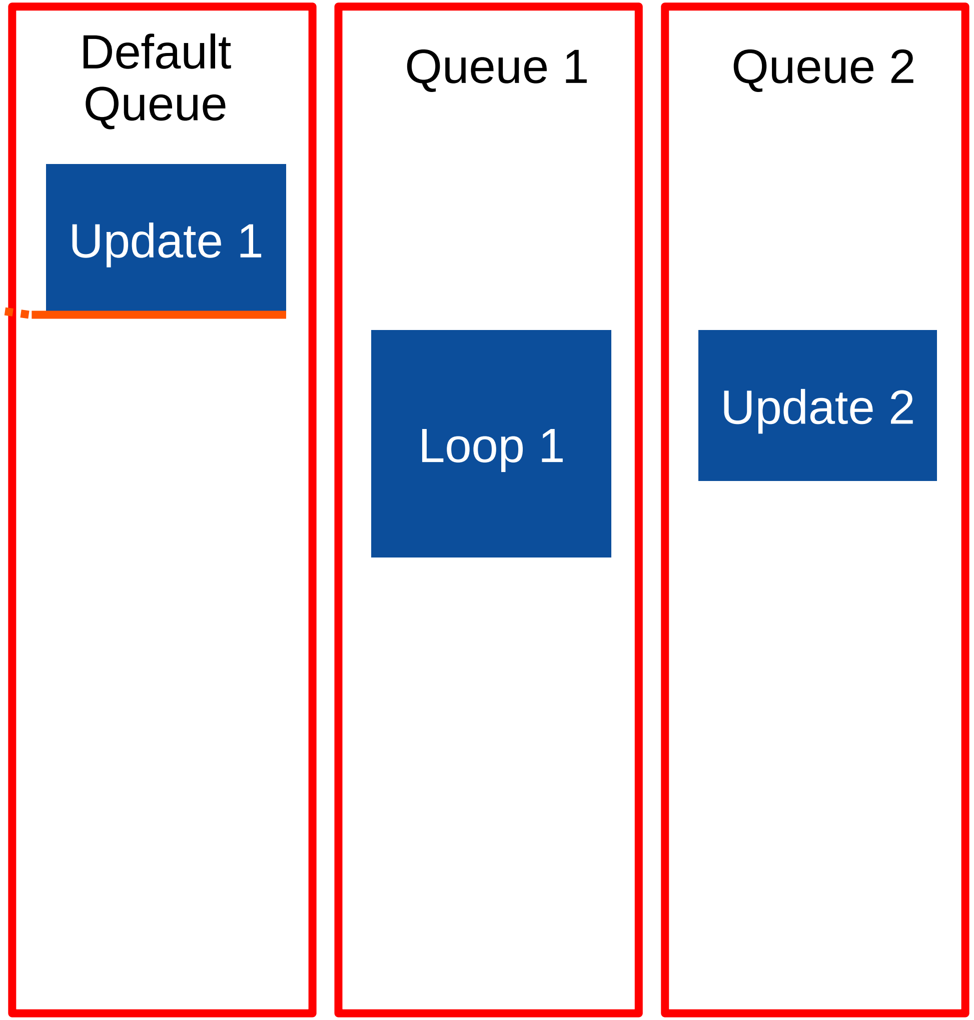
```
#pragma acc update device(X[0:100]) \
  async(1)

#pragma acc parallel loop async(1)
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) \
  async(2)

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



OpenACC

ASYNC EXAMPLE

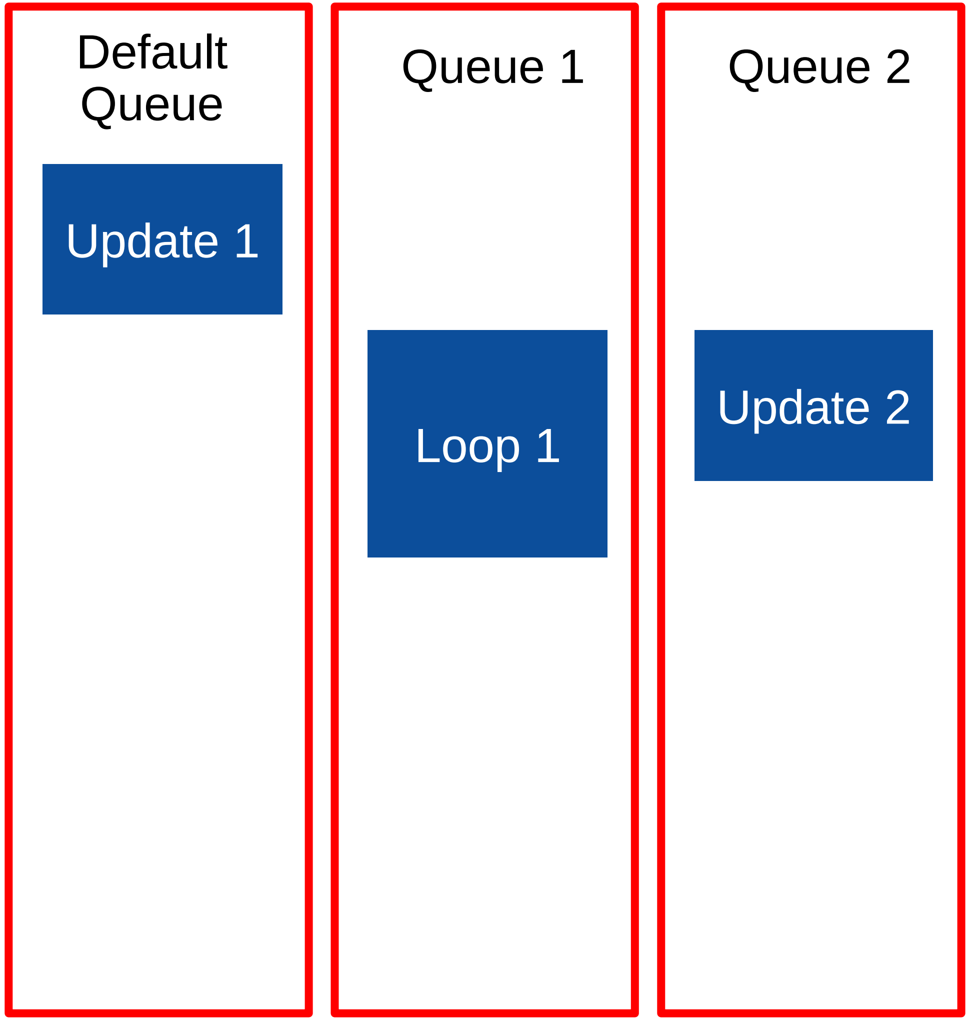
```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async(1)
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) \
  async(2)

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



OpenACC

ASYNC EXAMPLE

- By using the async clause, we allow the first loop and the second update to run simultaneously
- This means that we would be able to start the second loop sooner
- There is one problem though…



ASYNC EXAMPLE

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async(1)
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async(2)

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```

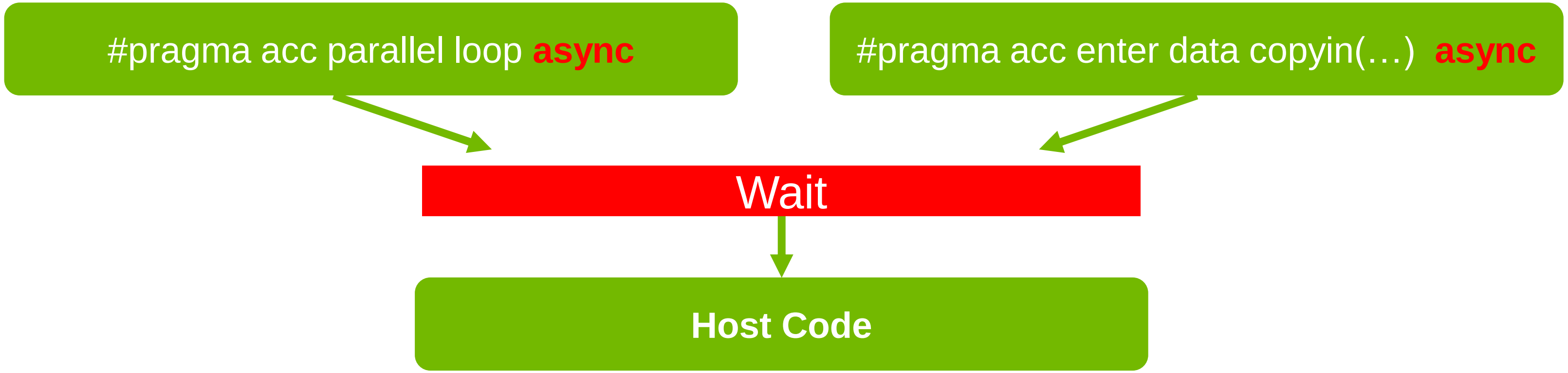
- By using the async clause, we launch work into separate queues
- Then any code afterwards could continue to run
- This means that there is a chance that the last loop my start execution before the update is finished
- We can fix this with the OpenACC **wait** directive

WAIT DIRECTIVE

`#pragma acc wait(n)`

Wait blocks host until all operations in queue *n* have completed

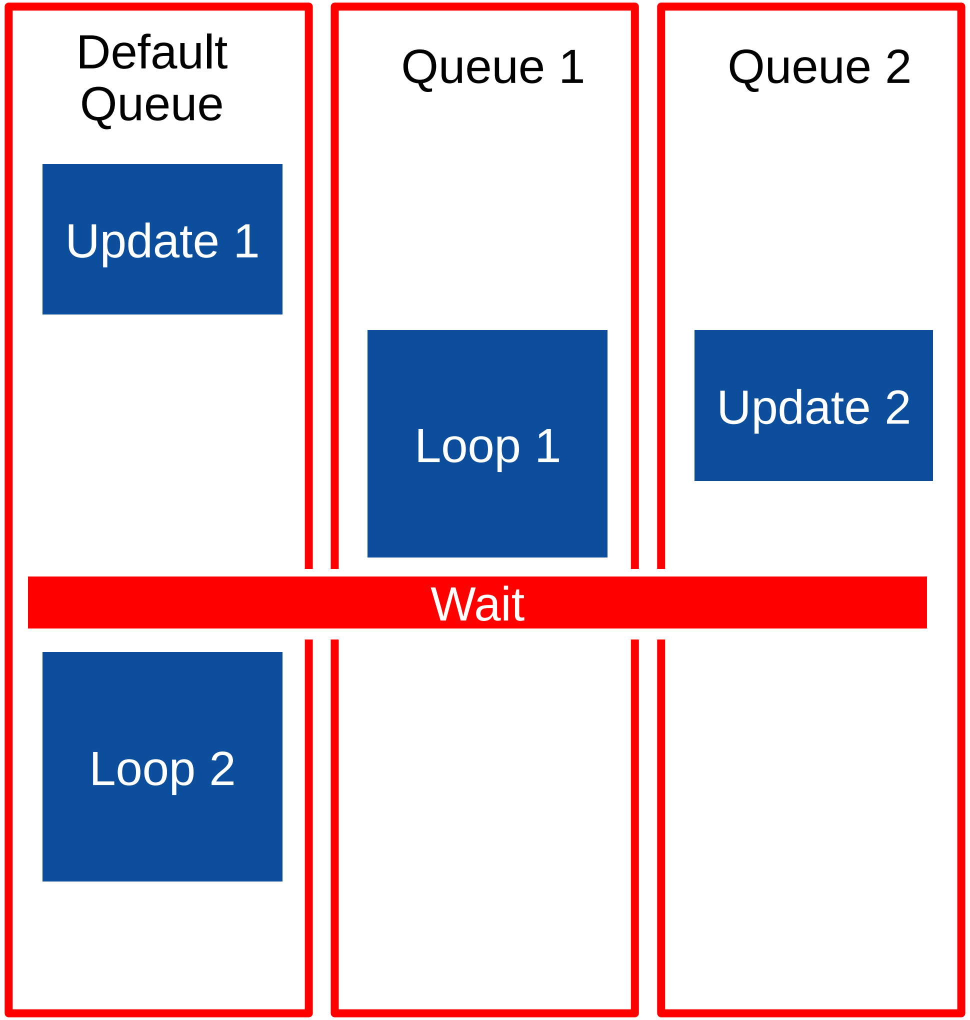
- The wait clause will pause your program until all previous async operations are completed
- If n is not specified, then your program will pause until **all** queues have completed



OpenACC

WAIT EXAMPLE

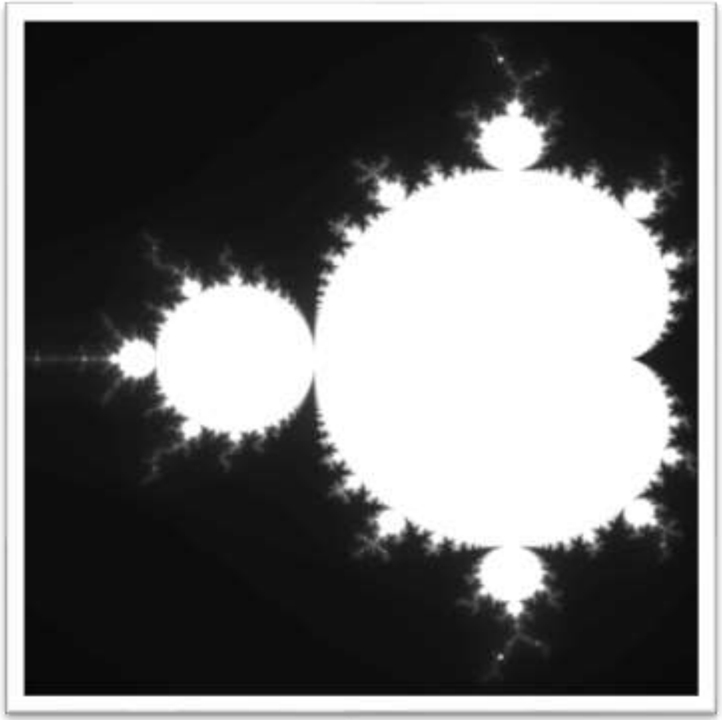
```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async

#pragma acc wait

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



MANDELBROT SAMPLE CODE

SAMPLE CODE: MANDELBROT

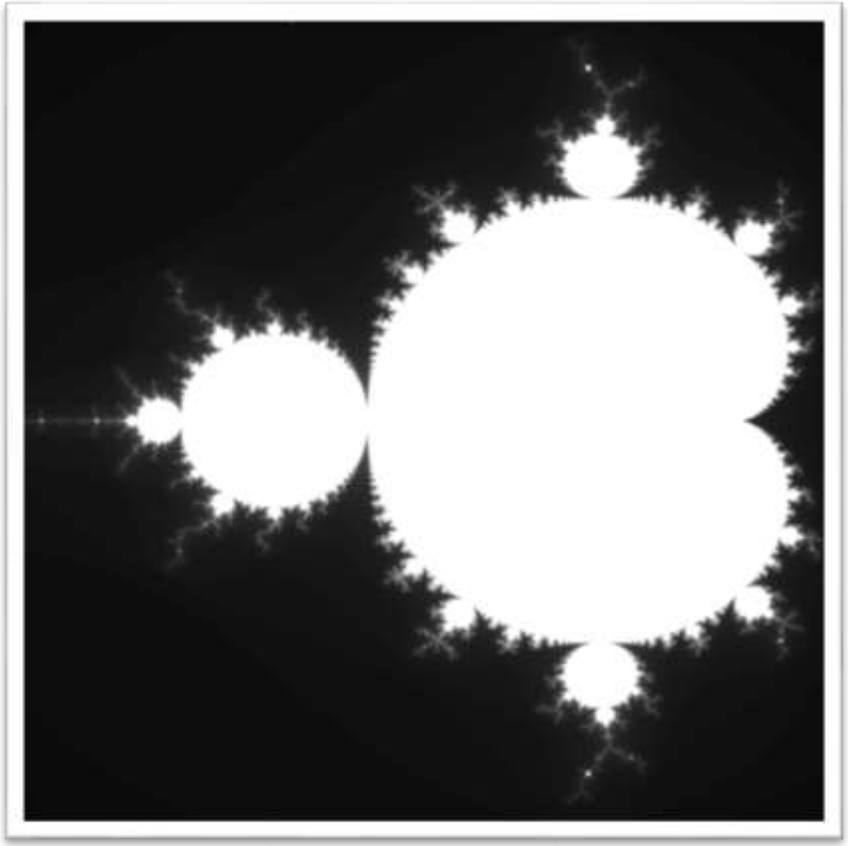
- Application generates the image to the right
- Each pixel in the image can be independently calculated
- Our goal is to use async to be able to pipeline some of the image generation process
- You may view this code at the below github link:



https://github.com/NVIDIA-OpenACC-Course/nvidia-openacc-course-sources

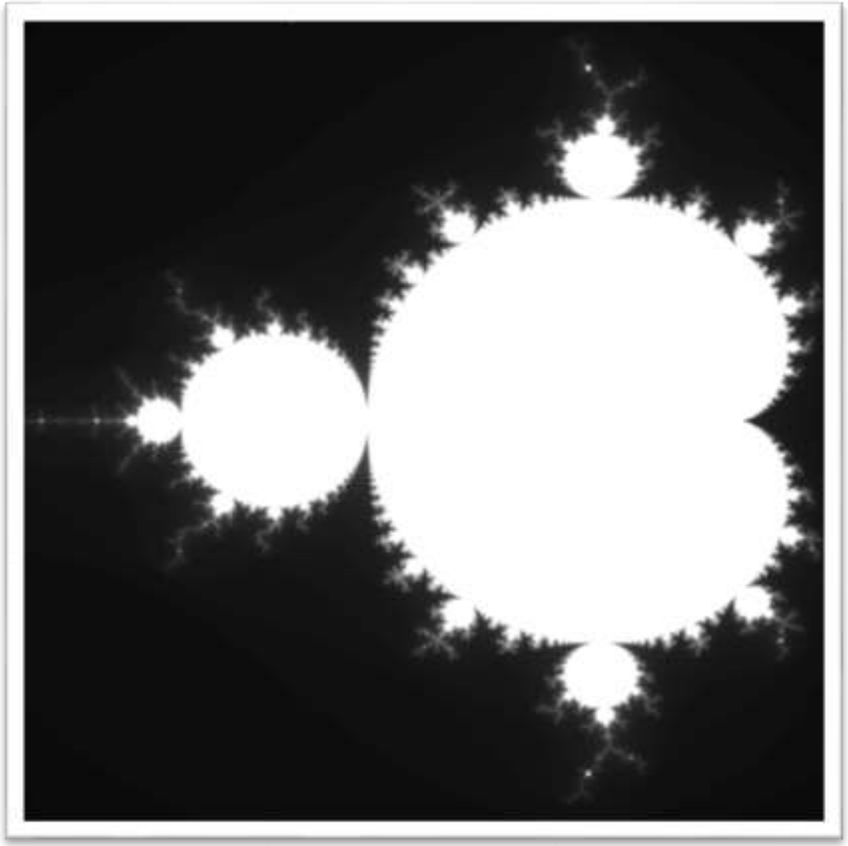
MAIN LOOP

```
int main()
{
    ...
    for(int y=0;y<HEIGHT;y++) {
        for(int x=0;x<WIDTH;x++) {
            image[y*WIDTH+x]=mandelbrot(x,y);
        }
    }
    ...
}
```



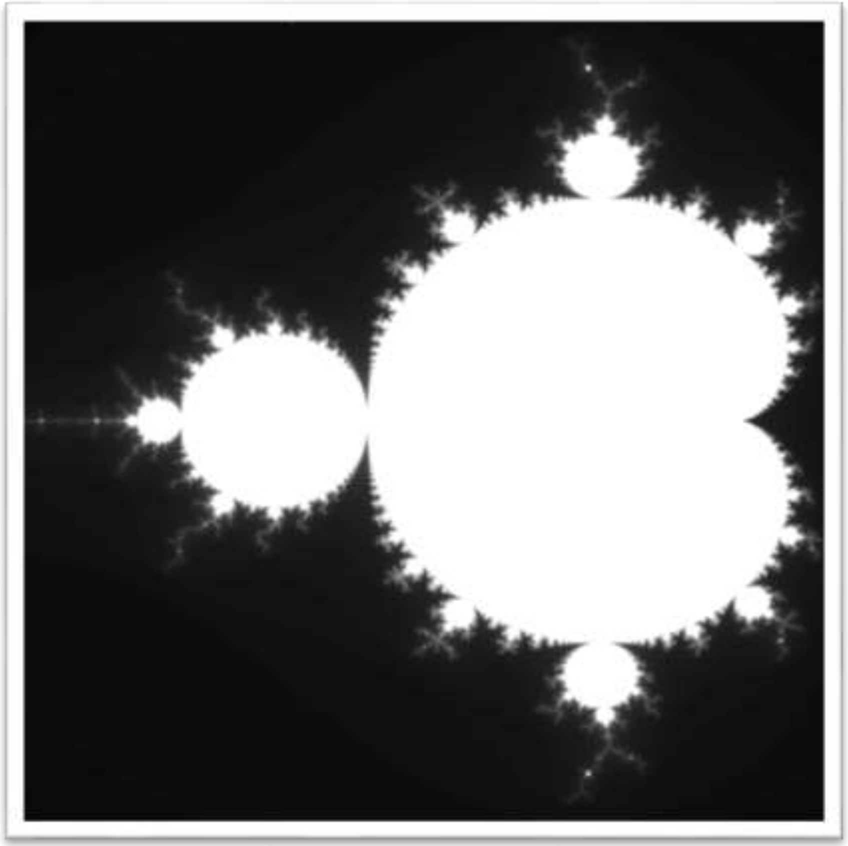
MANDLEBROT FUNCTION

```
// Calculate value for a pixel
unsigned char mandelbrot(int Px, int Py) {
  double x0=xmin+Px*dx;   double y0=ymin+Py*dy;
  double x=0.0;   double y=0.0;
  for(int i=0;x*x+y*y<4.0 && i<MAX_ITERS;i++) {
    double xtemp=x*x-y*y+x0;
    y=2*x*y+y0;
    x=xtemp;
  }
  return (double)MAX_COLOR*i/MAX_ITERS;
}
```



FUNCTIONS INSIDE COMPUTE REGIONS

```
int main()
{
    ...
#pragma acc parallel loop \
    copy(image[0:HEIGHT*WIDTH])
    for(int y=0;y<HEIGHT;y++) {
#pragma acc loop
        for(int x=0;x<WIDTH;x++) {
            image[y*WIDTH+x]=mandelbrot(x,y);
        }
    }
    ...
}
```



OpenACC

FUNCTIONS INSIDE COMPUTE REGIONS

```
int main()
{
    ...
#pragma acc parallel loop \
    copy(image[0:HEIGHT*WIDTH])
    for(int y=0;y<HEIGHT;y++) {
#pragma acc loop
        for(int x=0;x<WIDTH;x++) {
            image[y*WIDTH+x]=mandelbrot(x,y);
        }
    }
    ...
}
```

- We are currently unable to call the Mandelbrot function from within an OpenACC compute region
- We will have to provide OpenACC with additional information about the Mandelbrot function in order to run it on our device
- To accomplish this, we must use the OpenACC **routine** directive

ROUTINE DIRECTIVE

Specifies that the compiler should generate a device copy of the function/subroutine and what type of parallelism the routine contains.

Clauses:

gang/worker/vector/seq

Specifies the level of parallelism contained in the routine.

bind

Specifies an optional name for the routine, also supplied at call-site

no_host

The routine will only be used on the device

device_type

Specialize this routine for a particular device type

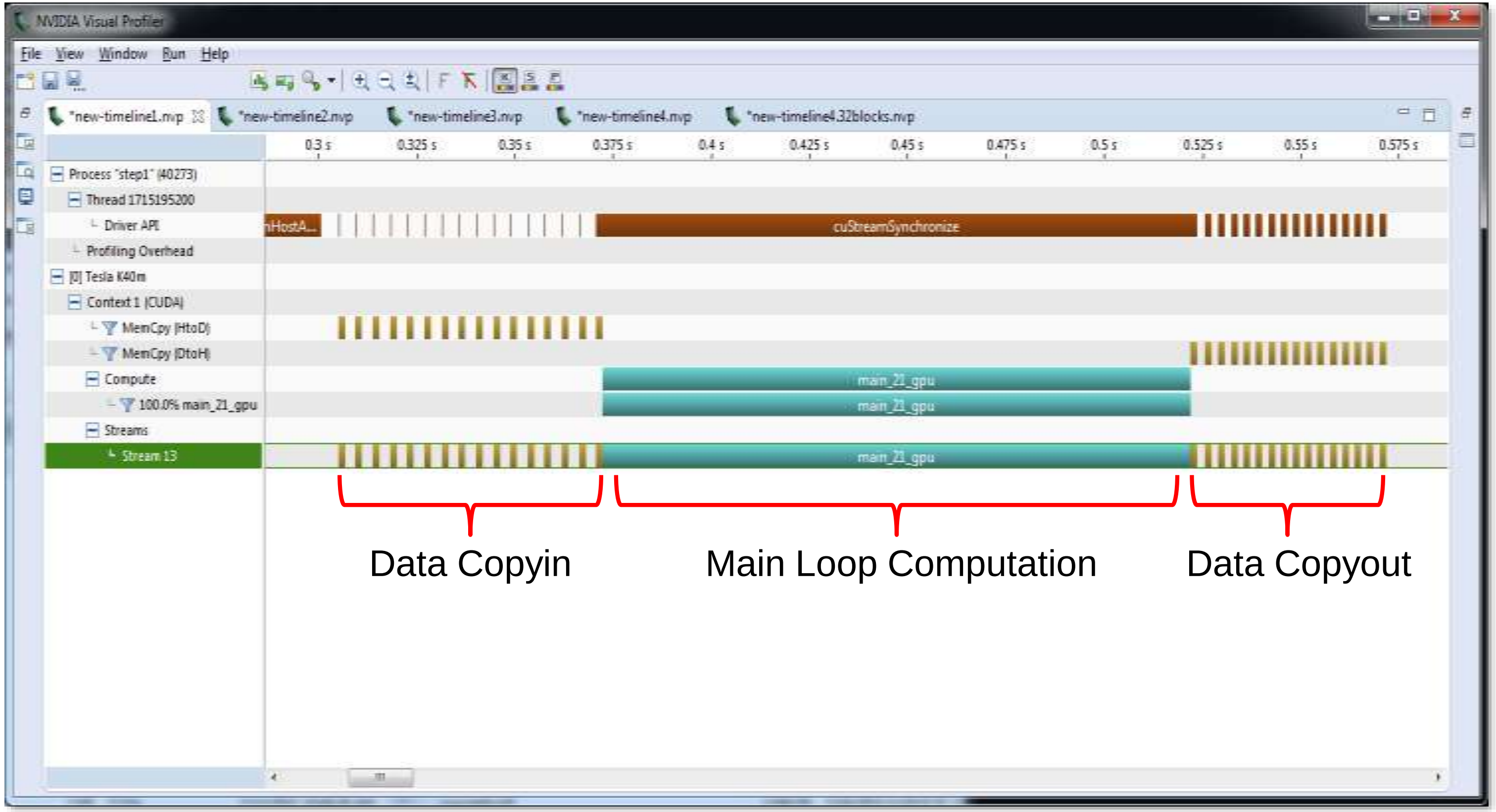
ADDING ROUTINE TO MANDELBROT

```
// Calculate value for a pixel
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py) {
  double x0=xmin+Px*dx;   double y0=ymin+Py*dy;
  double x=0.0;   double y=0.0;
  for(int i=0;x*x+y*y<4.0 && i<MAX_ITERS;i++) {
    double xtemp=x*x-y*y+x0;
    y=2*x*y+y0;
    x=xtemp;
  }
  return (double)MAX_COLOR*i/MAX_ITERS;
}
```

- Now when the program is compiled, an additional *device* version of the mandelbrot function will be generated
- When we call mandelbrot from within our parallel loop now, it will use the device version instead
- Since this routine is called by each iteration, it's marked as *sequential*

MANDELBROT PROFILE

- Right now our data copies and compute region occur synchronously
- Next we will apply the async clause to see if we could overlap some of this and see increased performance

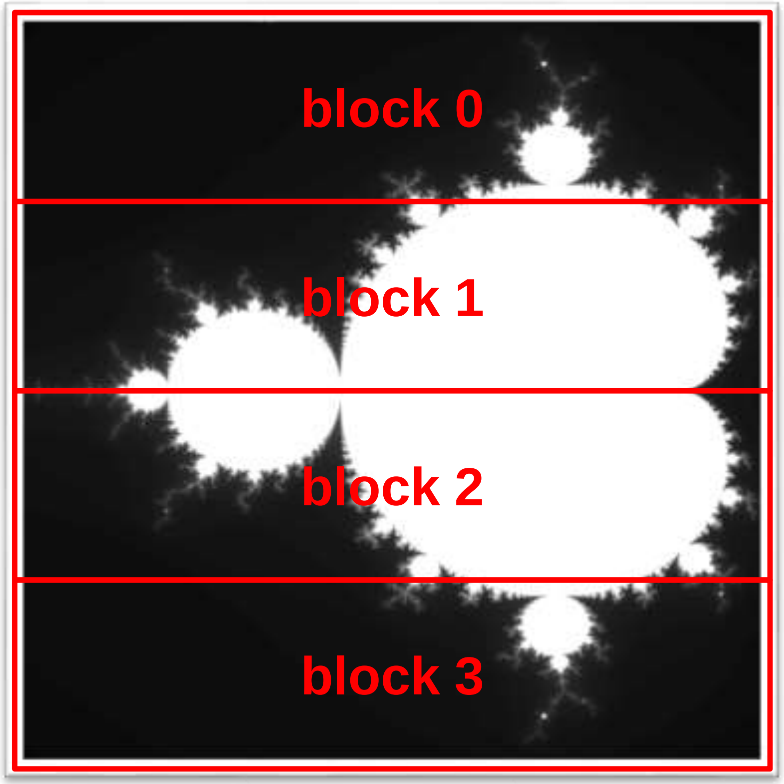


OpenACC

ASYNCHRONOUS MANDELBROT

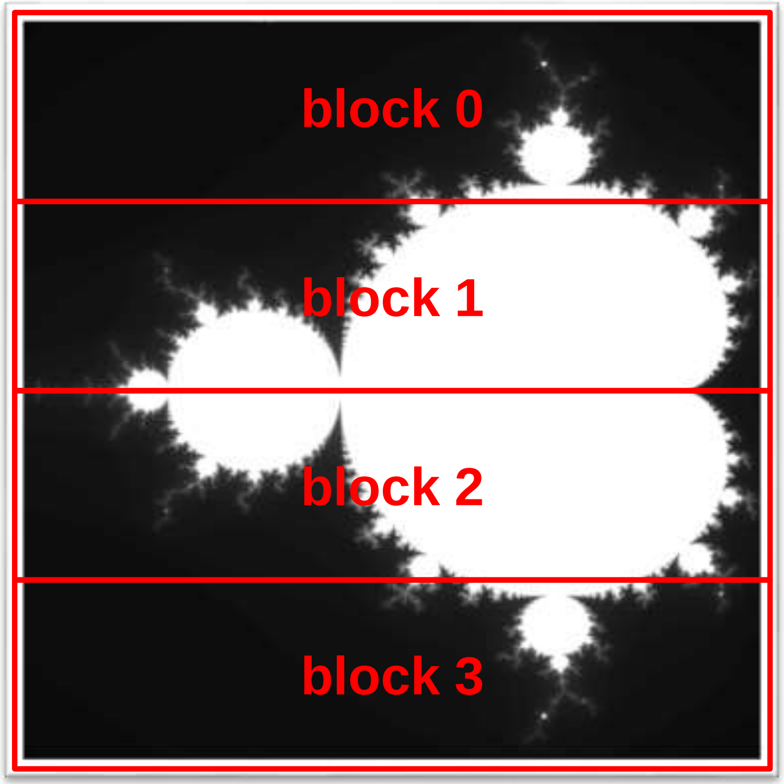
ASYNCHRONOUS MANDELBROT

- In order to get some asynchronous performance out of this code, we will break the *image* into multiple blocks
- Then we can process each block independently
- Applying the async clause to each block will then allow them to overlap



ASYNCHRONOUS MANDELBROT

```
int main()
{
    for(int block=0; block<4; block++) {
        int yStart = block*(HEIGHT/4);
        int yEnd   = yStart+(HEIGHT/4);
#pragma acc parallel loop \
    copy(image[0:HEIGHT*WIDTH])
        for(int y=yStart;y<yEnd;y++) {
#pragma acc loop
            for(int x=0;x<WIDTH;x++) {
                image[y*WIDTH+x]=mandelbrot(x,y);
            }
        }
    }
    ...
}
```



OpenACC

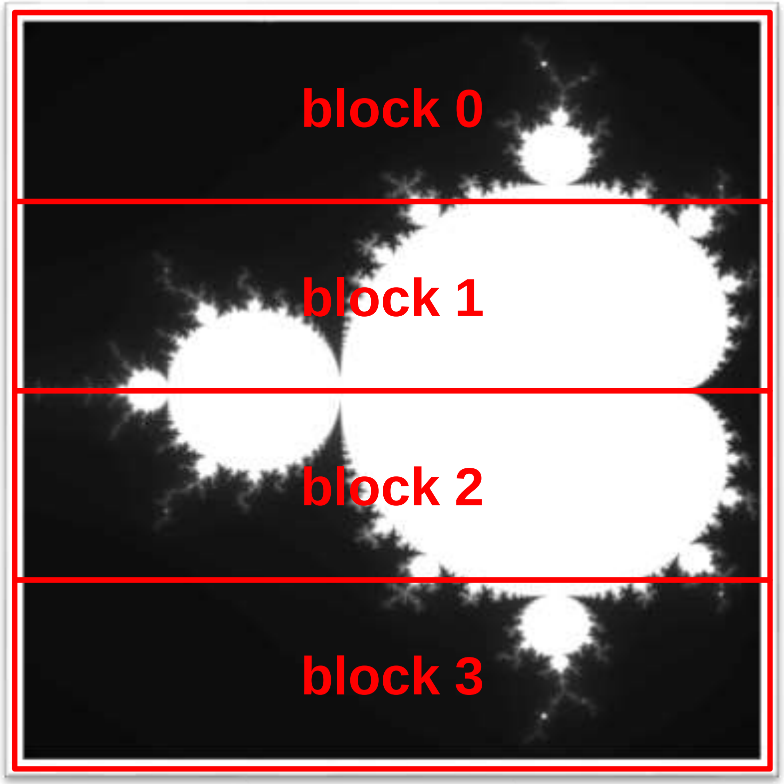
ASYNCHRONOUS MANDELBROT

- We are computing each block separately and can see multiple kernel launches in our profile.
- We also see our data transfers all happen at the end
- We need to change the code to copy each block after it's computed

OpenACC

ASYNCHRONOUS MANDELBROT

```
int main() {
    int block_size = WIDTH*HEIGHT/4;
#pragma acc data create(image[0:HEIGHT*WIDTH])
    for(int block=0; block<4; block++) {
        int yStart = block*(HEIGHT/4);
        int yEnd   = yStart+(HEIGHT/4);
#pragma acc parallel loop
        for(int y=yStart;y<yEnd;y++) {
#pragma acc loop
            for(int x=0;x<WIDTH;x++) {
                image[y*WIDTH+x]=mandelbrot(x,y);
            }
        }
#pragma acc update \
    host(image[yStart*WIDTH:block_size])
    }
}
```



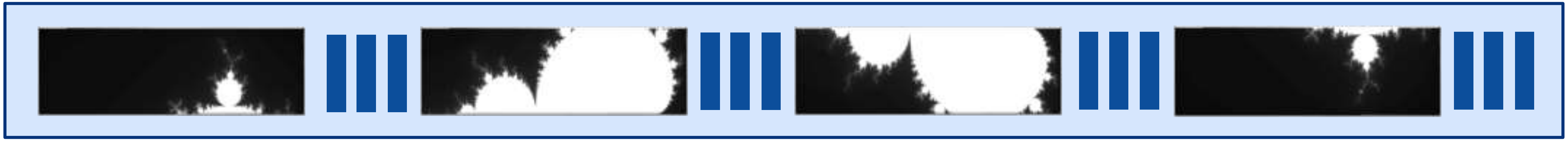
ASYNCHRONOUS MANDELBROT

- We have moved all of the data copies to after their block
- Next, we will use the async clause to overlap the data movement and compute regions

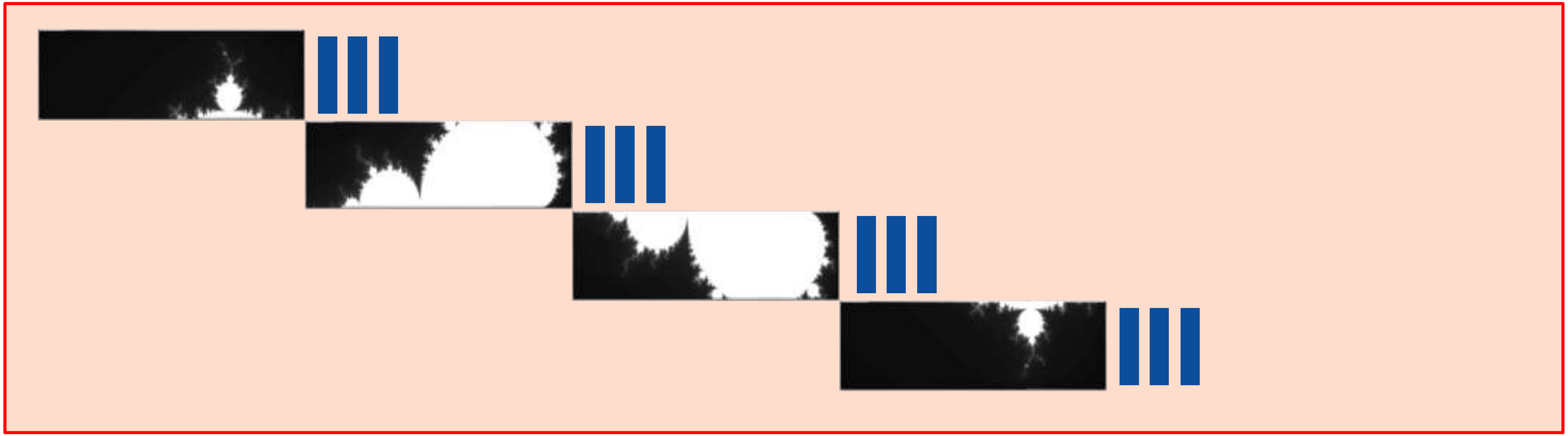
OpenACC

MANDELBROT PIPELINE

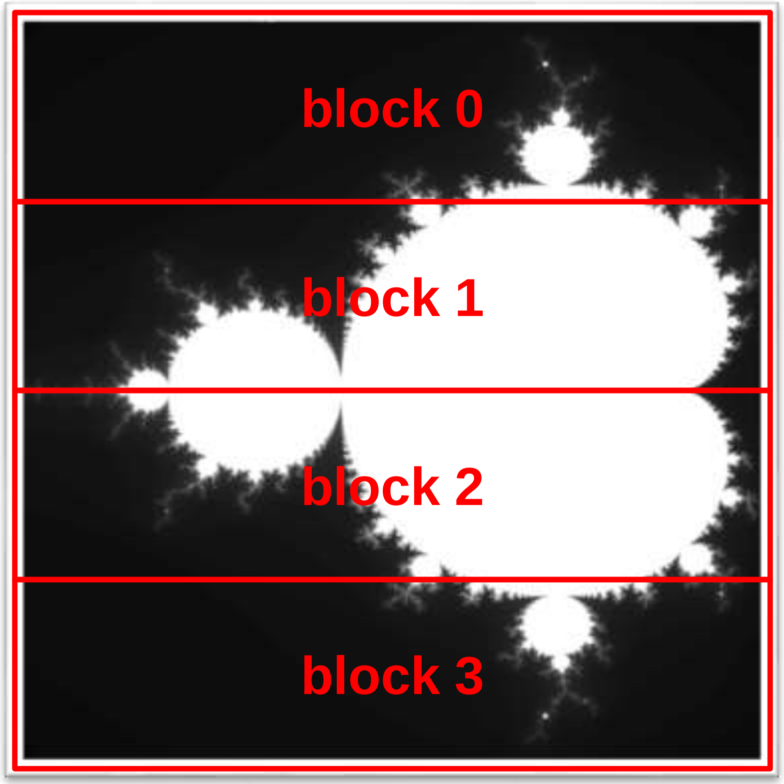
WITHOUT ASYNC



WITH ASYNC

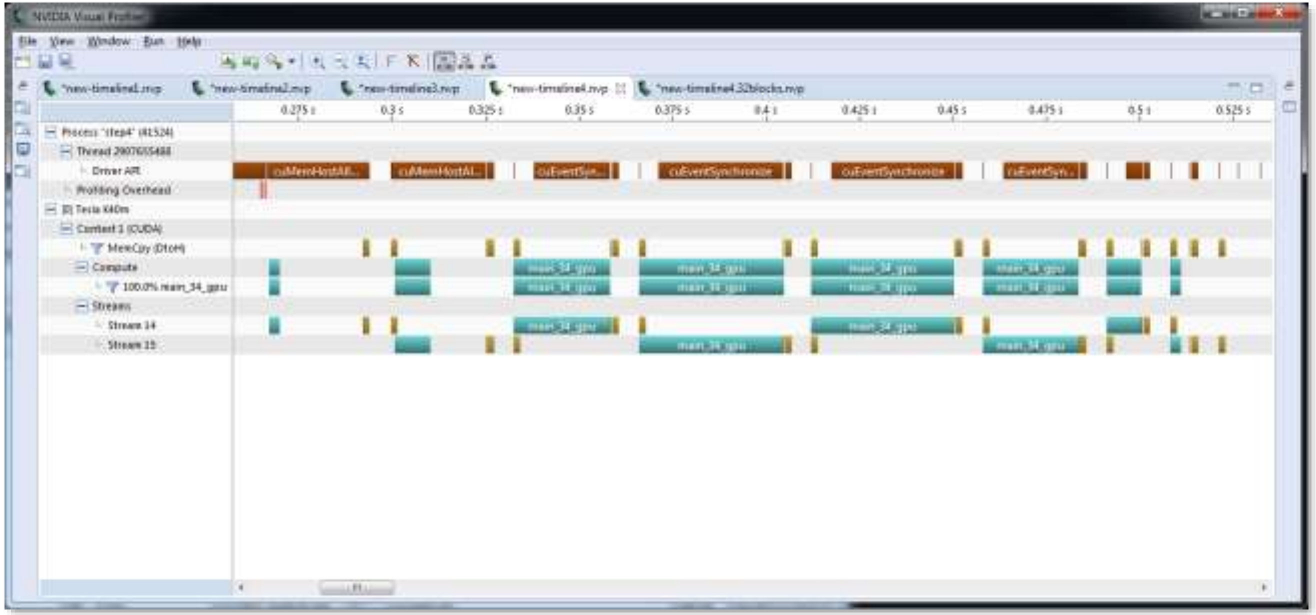


ASYNCHRONOUS MANDELBROT




```
int main() {
    int block_size = WIDTH*HEIGHT/4;
#pragma acc data create(image[0:HEIGHT*WIDTH])
    for(int block=0; block<4; block++) {
        int yStart = block*(HEIGHT/4);
        int yEnd   = yStart+(HEIGHT/4);
#pragma acc parallel loop async(block%2)
        for(int y=yStart;y<yEnd;y++) {
#pragma acc loop
            for(int x=0;x<WIDTH;x++) {
                image[y*WIDTH+x]=mandelbrot(x,y);
            }
        }
#pragma acc update \
host(image[yStart*WIDTH:block_size]) async(block%2)
    }
#pragma acc wait
}
```

ASYNCHRONOUS MANDELBROT



- Now in this profile, we can see that there is some overlap between the compute and data movement
- Because of how the code works, some blocks take longer to run than others
- We would most likely see better async performance when using more, but smaller, blocks

WRAP-UP

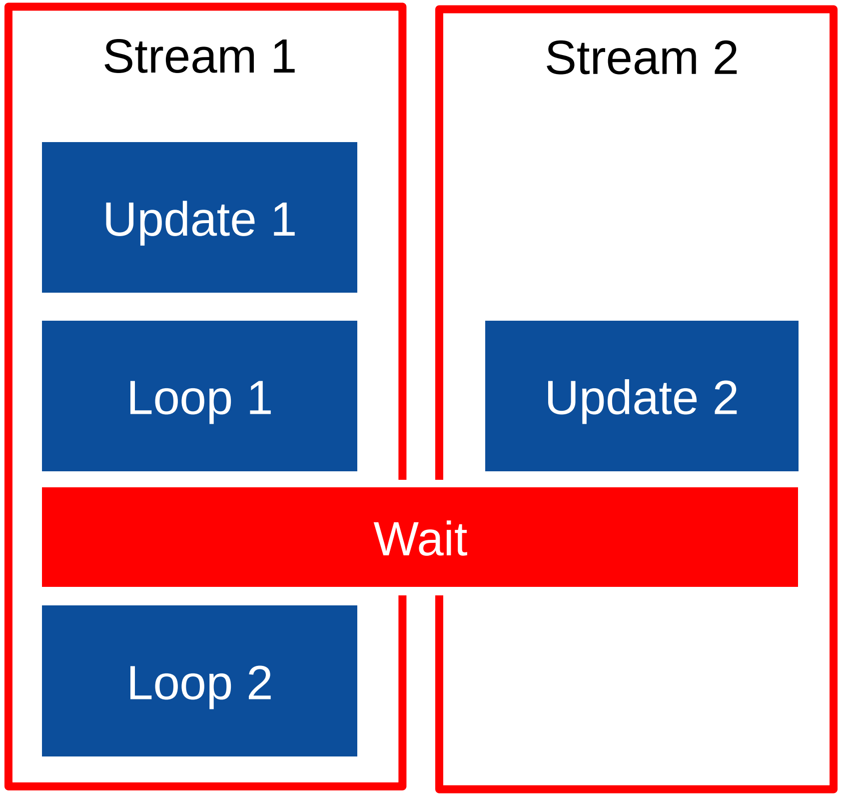
- Asynchronous programming is a way to allow two independent operations to occur concurrently
- You can accomplish asynchronous programming in OpenACC the async clause and wait directive
- You saw an example of *pipelining* for overlapping computation and data copies
- You saw an example of the *routine* directive to enable calling functions within compute regions

THANK YOU

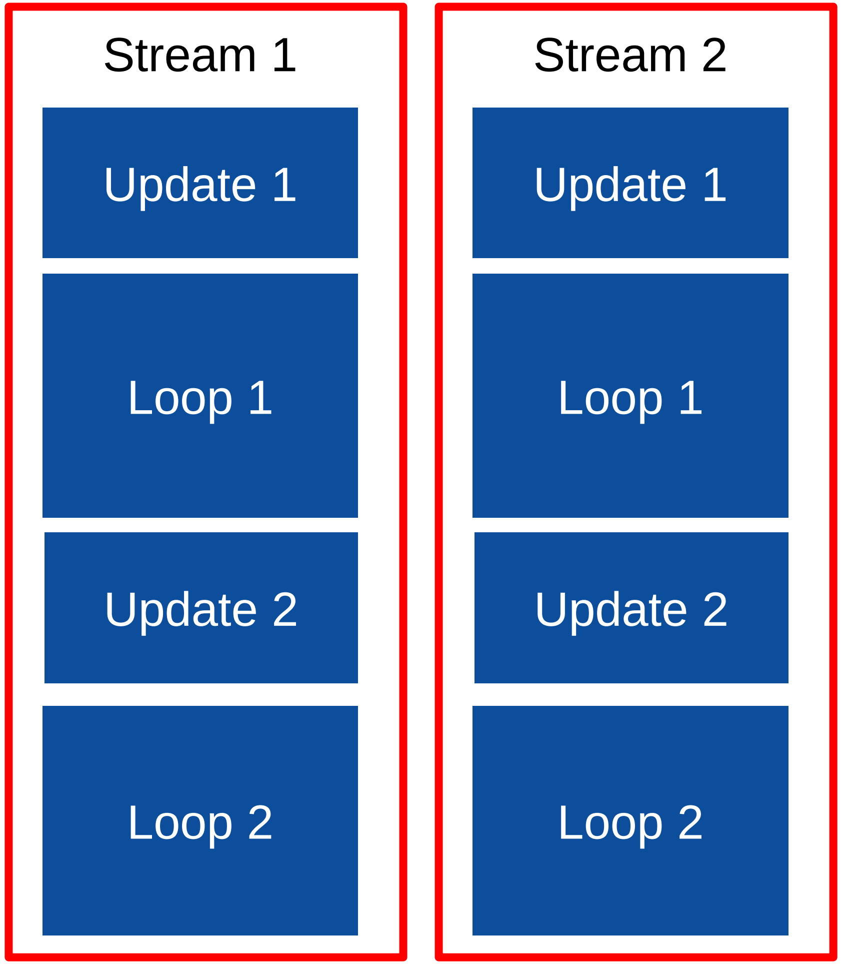
OpenACC
More Science, Less Programming

- Can we add a slide that shows performance results from varying the number of queues?

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async

#pragma acc wait

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```

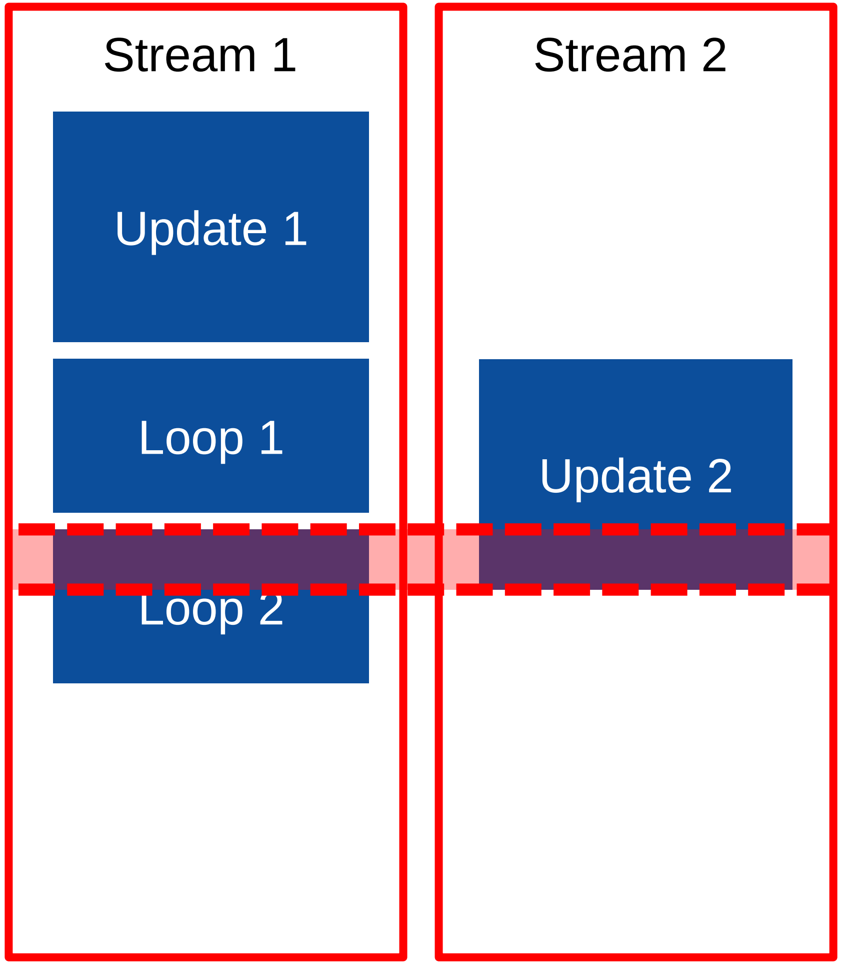


```
#pragma acc update device(X[0:100])

#pragma acc parallel loop
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100])

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



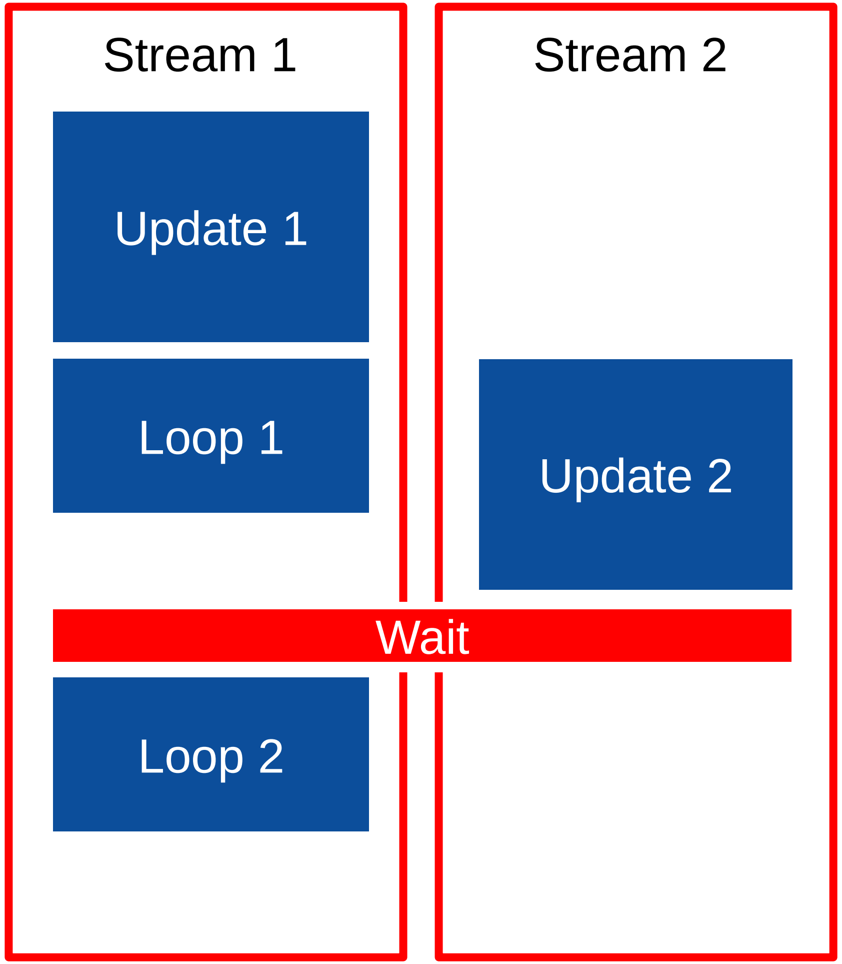
WAIT EXAMPLE

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



WAIT EXAMPLE

```
#pragma acc update device(X[0:100])

#pragma acc parallel loop async
for( i = 0; i < 100; i++ )
    X[i] = ...

#pragma acc update device(X[100:100]) async

#pragma acc wait

#pragma acc parallel loop
for( i = 100; i < 200; i++ )
    X[i] = ...
```



MANDELBROT PIPELINE