INTRODUCTION TO OPENACC

Lecture 3: Advanced, November 9, 2016

NVIDIA

Course Objective:

Enable *you* to accelerate your applications with OpenACC.

Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:
https://developer.nvidia.com/intro-to-openacc-course-2016

ADVANCED OPENACC

Lecture 3: Jeff Larkin, NVIDIA

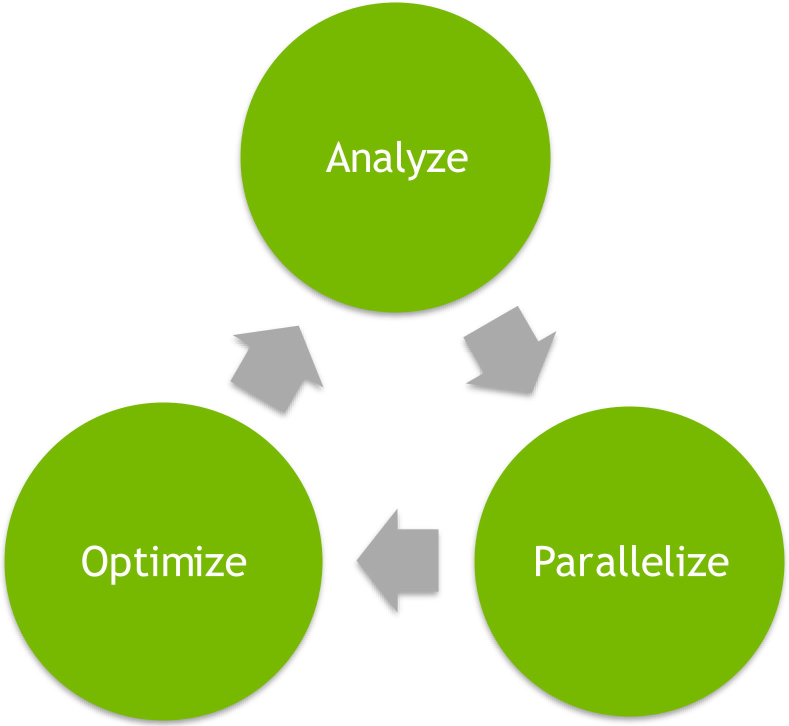
Today's Objectives

Understand several common refactoring techniques to improve performance

Understand the routine directive

Understand async and wait and how to apply them to data pipelining

Understand the OpenACC interoperability with libraries and CUDA



Additional Common Optimizations

The collapse Clause

collapse(n): Takes the next *n* tightly-nested loops, folds them into one, and applies the OpenACC directives to the new loop.

```
#pragma acc parallel loop \
  collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<M; j++)
    ...
```

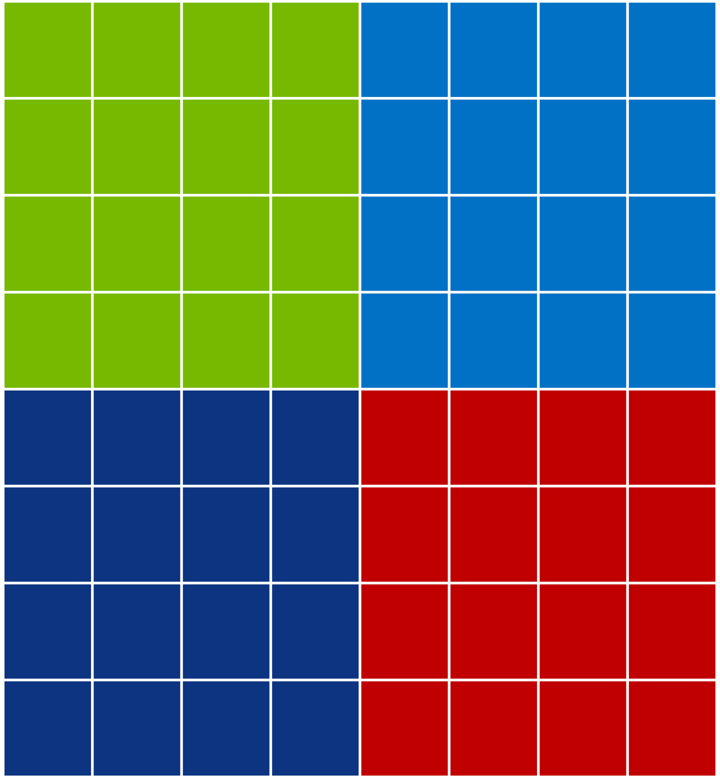
```
#pragma acc parallel loop
for(int ij=0; ij<N*M; ij++)
    ...
```

Why?

- Collapse outer loops to enable creating more gangs.
- Collapse inner loops to enable longer vector lengths.
- Collapse all loops, when possible, to do both.

The tile clause

Operate on smaller blocks of the operation to exploit data locality



```
#pragma acc loop tile(4,4)
  for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
      Temp[i][j] = 0.25 *
        (Temp_last[i+1][j] +
         Temp_last[i-1][j] +
         Temp_last[i][j+1] +
         Temp_last[i][j-1]);
    }
  }
```

Stride-1 Memory Accesses

```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
  {
    A[i][j][1] = 1.0f;
    A[i][j][2] = 0.0f;
  }
}
```

The fastest dimension is length 2 and fastest loop strides by 2.

```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
  {
    A[1][i][j] = 1.0f;
    A[2][i][j] = 0.0f;
  }
}
```

Now the inner loop is the fastest dimension through memory.

Stride-1 Memory Accesses

```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
  {
    A[i][j].a = 1.0f;
    A[i][j].b = 0.0f;
  }
}
```

If all threads access the “a” element, they will be accesses every-other memory element.

```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
  {
    Aa[i][j] = 1.0f;
    Ab[i][j] = 0.0f;
  }
}
```

Now all threads are access contiguous elements of Aa and Ab.

OpenACC Routine Directive

OpenACC 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.

You *must* declare *one* level of parallelism on the routine directive.

Routine Directive: C/C++

```
// foo.h
#pragma acc routine seq
double foo(int i);


// Used in main()
#pragma acc parallel loop
for(int i=0;i<N;i++) {
  array[i] = foo(i);
}
```

- At function source:
  - Function needs to be built for the GPU.
  - It will be called by each thread (sequentially)
- At call the compiler needs to know:
  - Function will be available on the GPU
  - It is a sequential routine

OpenACC Routine: Fortran

```
module foo_mod
 contains
 real(8) function foo(i)
    implicit none
    !$acc routine(foo) seq
    integer,intent(in),value :: i
    ...
 end function foo
end module foo_mod
```

The `routine` directive may appear in a Fortran function or subroutine definition, or in an interface block.

The save attribute is not supported.

Nested acc routines require the routine directive within each nested routine.

Asynchronous Programming with OpenACC

Asynchronous Programming

Programming multiple operations 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 OpenACC

So far, all OpenACC directives have been synchronous with the host

- Host waits for the parallel loop to complete
- Host waits for data updates to complete

Most OpenACC directives can be made asynchronous

- Host issues multiple parallel loops to the device before waiting
- Host performs part of the calculation while the device is busy
- Data transfers can happen before the data is needed

Asynchronous Pipelining

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

Photo by Roger Wollstadt, used via Creative Commons

Case Study: Image Filter

The example code reads an image from a file, applies a simple filter to the image, then outputs the file.

Skills Used:

- Parallelize the filter on the GPU
- Data Region and Update Directive
- Async and Wait directives
- Pipelining

Image Filter Code

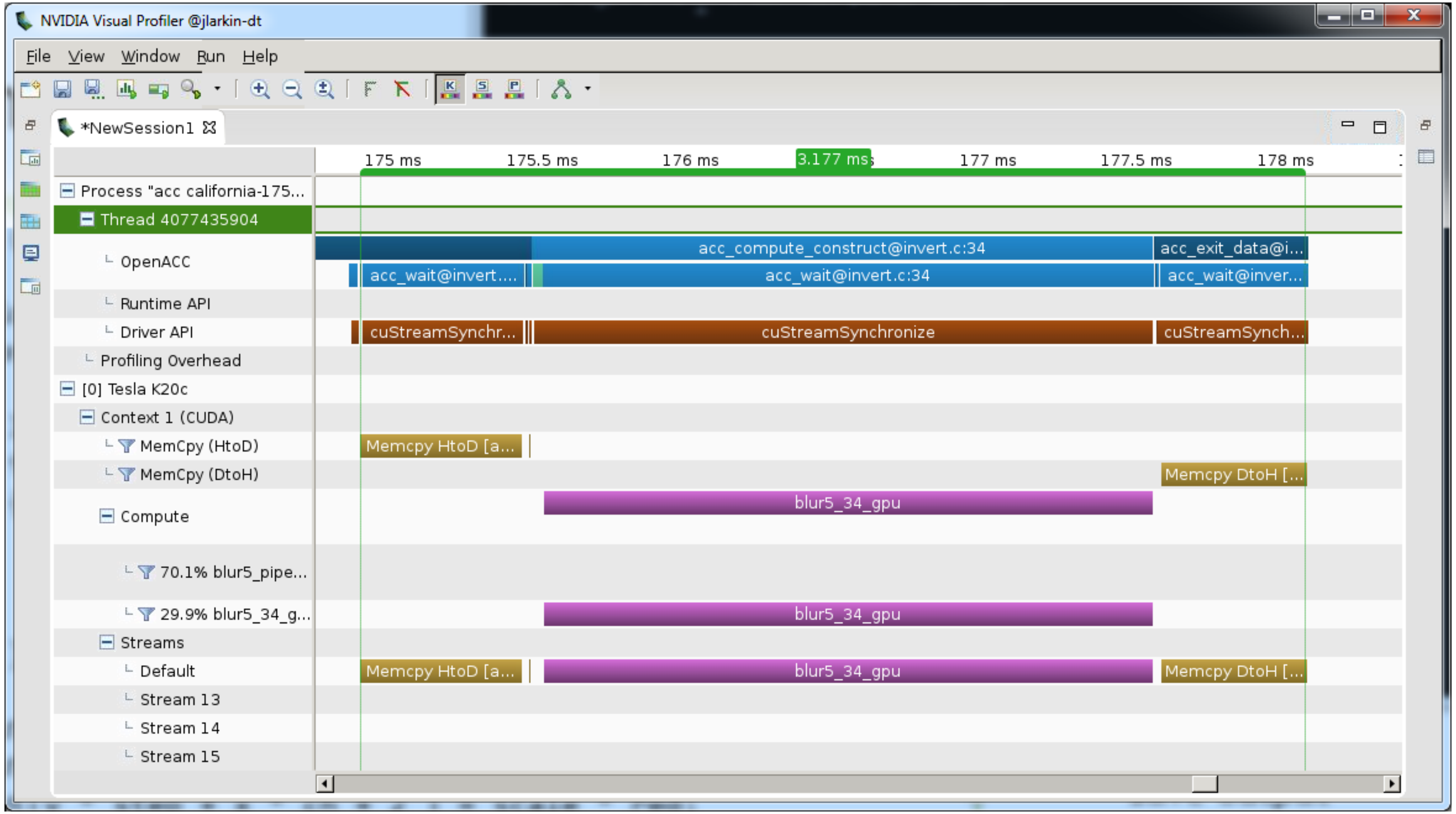
```
#pragma acc parallel loop collapse(2) gang vector
for ( y = 0; y < h; y++ ); for ( x = 0; x < w; x++ )
{
  float blue = 0.0, green = 0.0, red = 0.0;
  for ( int fy = 0; fy < filtersize; fy++ )
  {
    long iy = y - (filtersize/2) + fy;
    for ( int fx = 0; fx < filtersize; fx++ )
    {
      blue+=filter[fy][fx]*imgData[iy*step+ix*ch];
      green+=filter[fy][fx]*imgData[iy*step+ix*ch+1];
      red+=filter[fy][fx]*imgData[iy*step+ix*ch+2];
    }
  }
  out[y * step + x * ch]      = 255 - scale * blue;
  out[y * step + x * ch + 1 ] = 255 - scale * green;
  out[y * step + x * ch + 2 ] = 255 - scale * red;
}
```

Iterate over all pixels

Apply filter

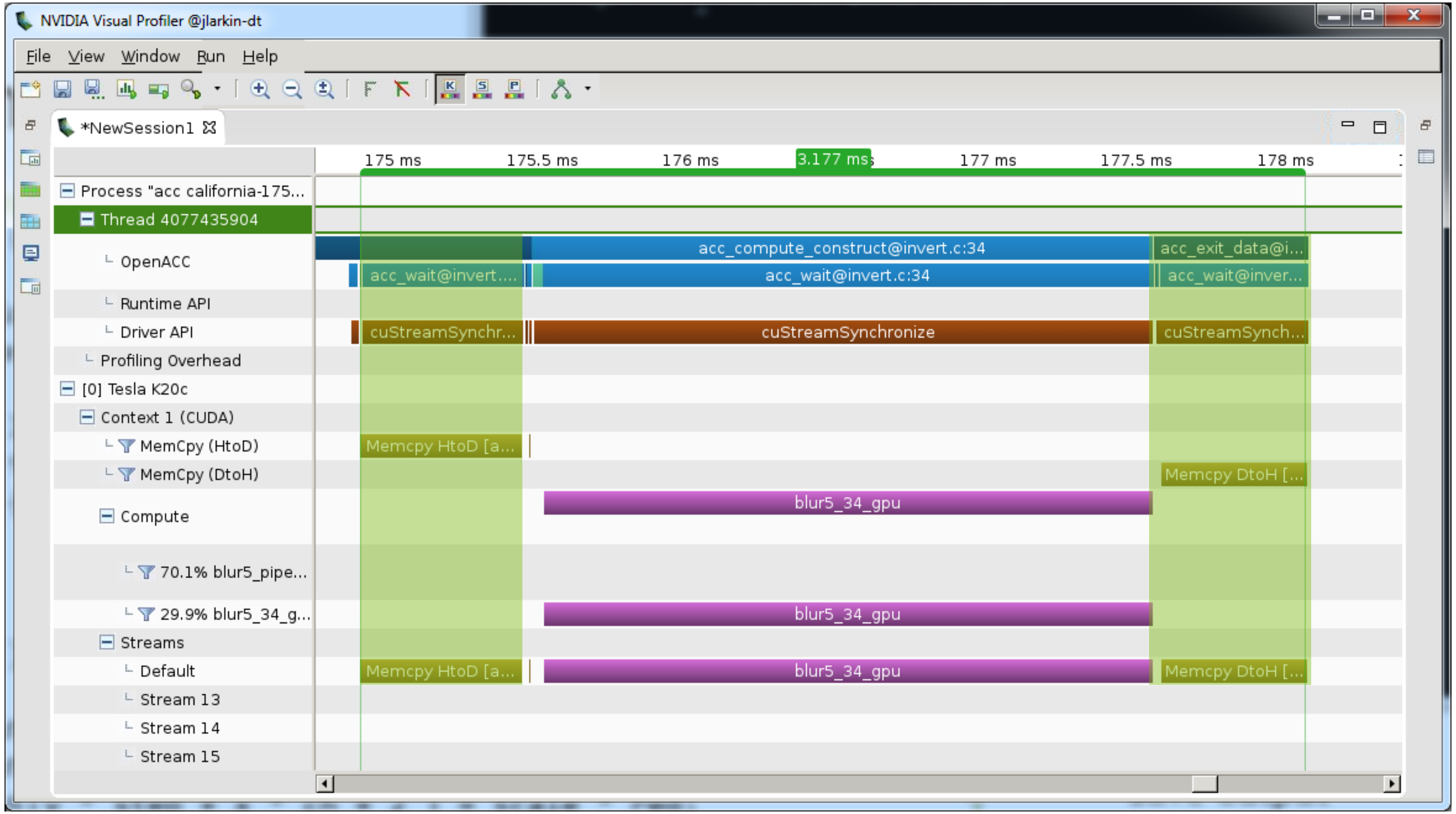
Save output

GPU Timeline

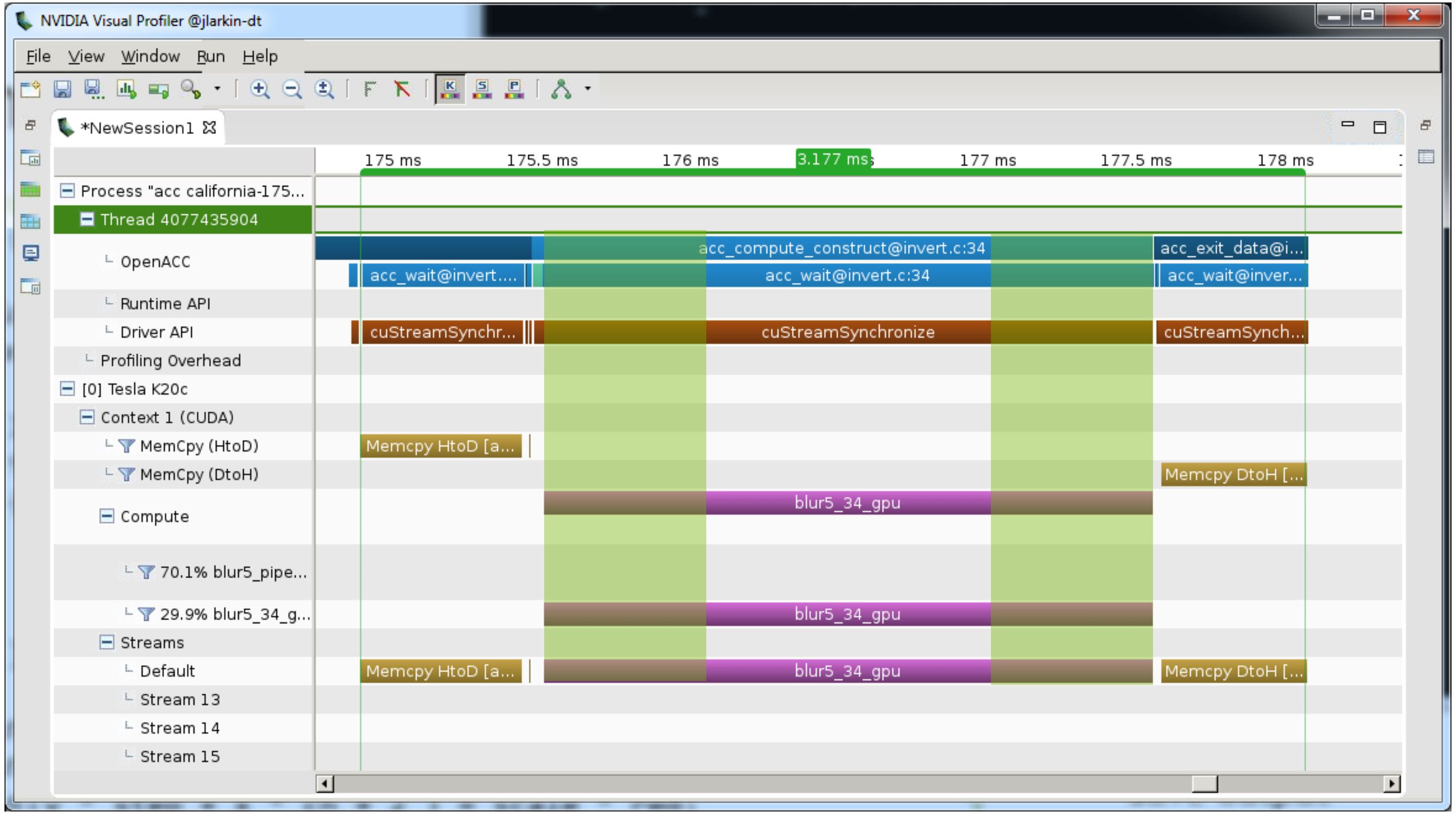


GPU Timeline

Roughly 1/3 of the runtime is occupied with data transfers.



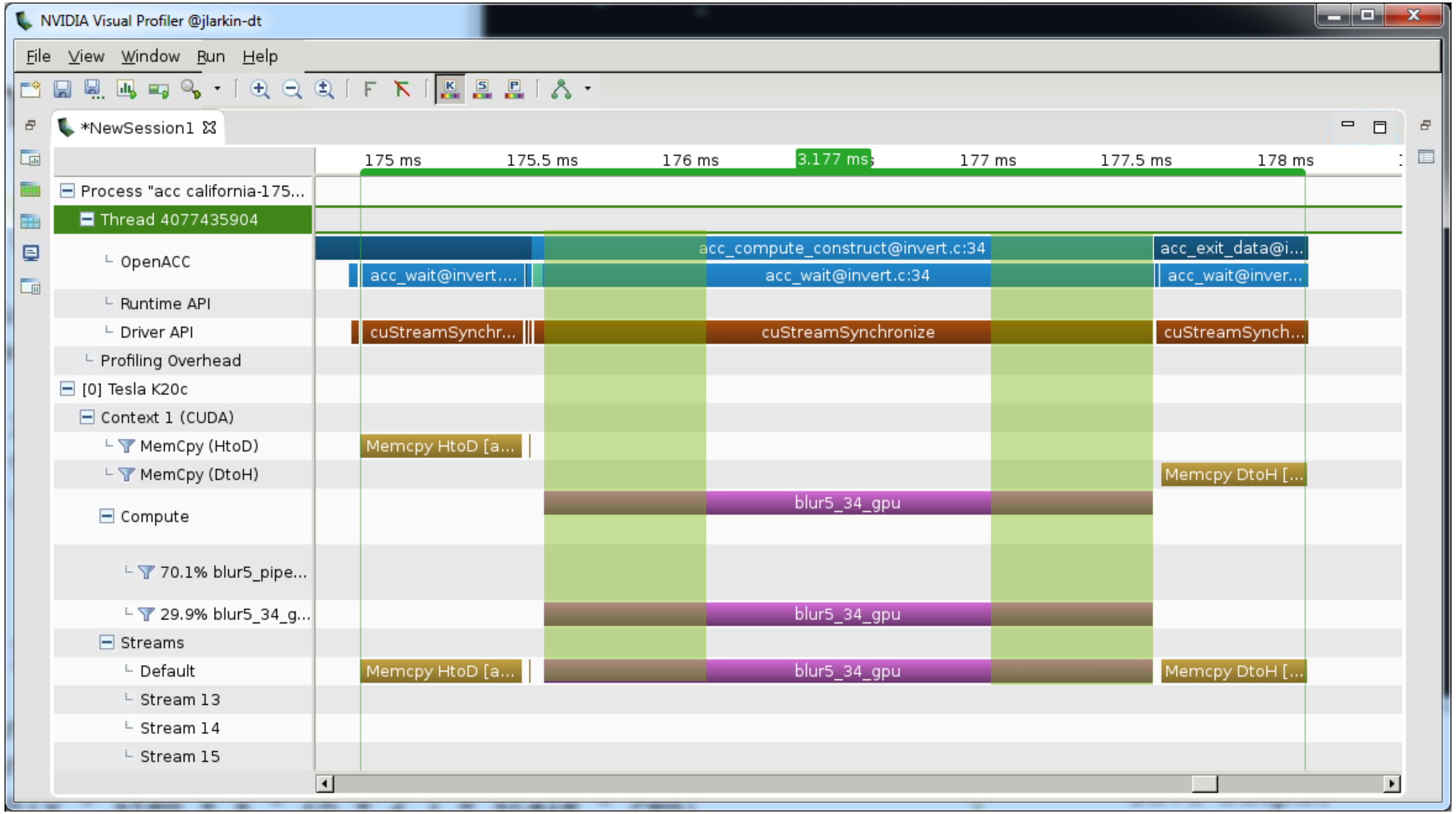
GPU Timeline



Roughly 1/3 of the runtime is occupied with data transfers.

What if we could overlap the copies with the computation?

GPU Timeline



Roughly 1/3 of the runtime is occupied with data transfers.

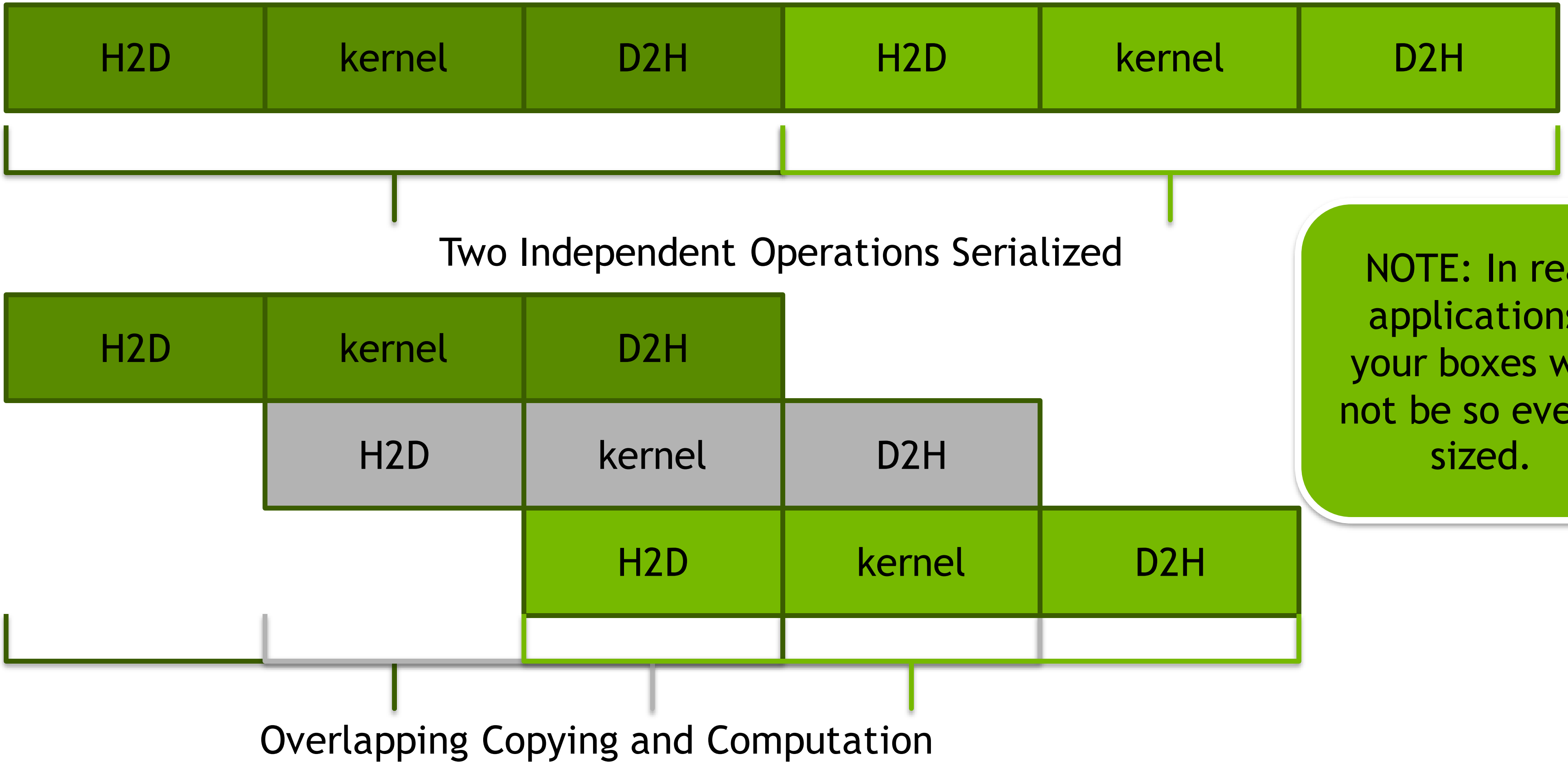
What if we could overlap the copies with the computation?

Rough Math:

3.2ms - .5ms - .5ms = 2.2 ms

3.2 / 2.2 ~= 1.45X

Pipelining Data Transfers



NOTE: In real applications, your boxes will not be so evenly sized.

Blocking the Code

Before we can overlap data transfers with computation, we need to break our work into smaller chunks.

Since each pixel is calculated independently, the work can be easily divided.

We'll divide along chunks of rows for convenience.

Blocked Image Filter Code

```
#pragma acc data copyin(imgData[:w*h*ch],filter)
                 copyout(out[:w*h*ch])
for ( long blocky = 0; blocky < nblocks; blocky++){
  long starty = blocky * blocksize;
  long endy   = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector
  for (y=starty; y<endy; y++); for(x=0; x<w; x++ ) {
    float blue = 0.0, green = 0.0, red = 0.0;
    for ( int fy = 0; fy < filtersize; fy++ ) {
      long iy = y - (filtersize/2) + fy;
      for ( int fx = 0; fx < filtersize; fx++ ){
        long ix = x - (filtersize/2) + fx;
        blue+=filter[fy][fx]*imgData[iy*step+ix*ch];
        green+=filter[fy][fx]*imgData[iy*step+ix*ch+1];
        red+=filter[fy][fx]*imgData[iy*step+ix*ch+2];
      }
    }
    out[y*step+x*ch]   = 255 - (scale * blue);
    out[y*step+x*ch+1] = 255 - (scale * green);
    out[y*step+x*ch+2] = 255 - (scale * red);
  }
}
```

1. Add blocking loop

Blocked Image Filter Code

```
#pragma acc data copyin(imgData[:w*h*ch],filter)
                 copyout(out[:w*h*ch])
for ( long blocky = 0; blocky < nblocks; blocky++){
  long starty = blocky * blocksize;
  long endy   = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector
  for (y=starty; y<endy; y++); for(x=0; x<w; x++ ) {
    float blue = 0.0, green = 0.0, red = 0.0;
    for ( int fy = 0; fy < filtersize; fy++ ) {
      long iy = y - (filtersize/2) + fy;
      for ( int fx = 0; fx < filtersize; fx++ ){
        long ix = x - (filtersize/2) + fx;
        blue+=filter[fy][fx]*imgData[iy*step+ix*ch];
        green+=filter[fy][fx]*imgData[iy*step+ix*ch+1];
        red+=filter[fy][fx]*imgData[iy*step+ix*ch+2];
      }
    }
    out[y*step+x*ch]   = 255 - (scale * blue);
    out[y*step+x*ch+1] = 255 - (scale * green);
    out[y*step+x*ch+2] = 255 - (scale * red);
  }
}
```

1. Add blocking loop

2. Adjust “y” to only iterate through rows within a single chunk.

Blocked Image Filter Code

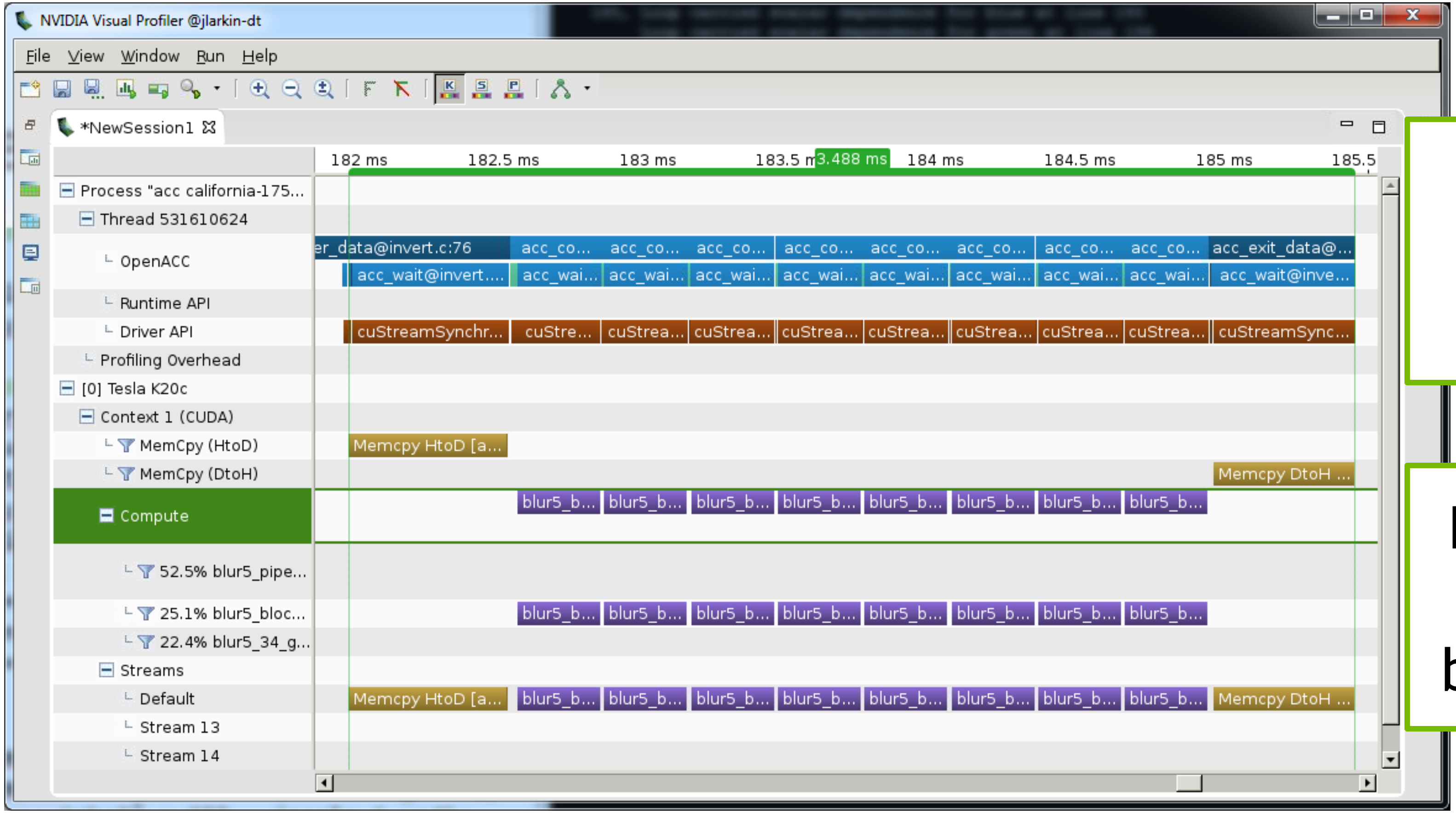
```
#pragma acc data copyin(imgData[:w*h*ch],filter)
                 copyout(out[:w*h*ch])
for ( long blocky = 0; blocky < nblocks; blocky++){
  long starty = blocky * blocksize;
  long endy   = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector
  for (y=starty; y<endy; y++); for(x=0; x<w; x++ ) {
    float blue = 0.0, green = 0.0, red = 0.0;
    for ( int fy = 0; fy < filtersize; fy++ ) {
      long iy = y - (filtersize/2) + fy;
      for ( int fx = 0; fx < filtersize; fx++ ){
        long ix = x - (filtersize/2) + fx;
        blue+=filter[fy][fx]*imgData[iy*step+ix*ch];
        green+=filter[fy][fx]*imgData[iy*step+ix*ch+1];
        red+=filter[fy][fx]*imgData[iy*step+ix*ch+2];
      }
    }
    out[y*step+x*ch]   = 255 - (scale * blue);
    out[y*step+x*ch+1] = 255 - (scale * green);
    out[y*step+x*ch+2] = 255 - (scale * red);
  }
}
```

3. Data region to handle copies

1. Add blocking loop

2. Adjust “y” to only iterate through rows within a single chunk.

GPU Timeline Blocked



Compute kernel is now broken in 8 blocks.

Data transfers still happen at beginning and end.

OpenACC Update Directive

Programmer specifies an array (or part of an array) that should be refreshed within a data region. (Host and Device copies are made coherent)

```
do_something_on_device()
!$acc update host(a)
do_something_on_host()
!$acc update device(a)
```

Copy “a” from GPU to CPU

Copy “a” from CPU to GPU

Note: Update “host” has been deprecated and renamed “self”

Blocked Update Code

Change data clauses to create

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update
device(imgData[starty*step:blocksize*step])
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]      = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step])
}
```

Blocked Update Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update
device(imgData[starty*step:blocksize*step])
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]      = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step])
}
```

Change data clauses to create

Update data one block at a time.

Blocked Update Code

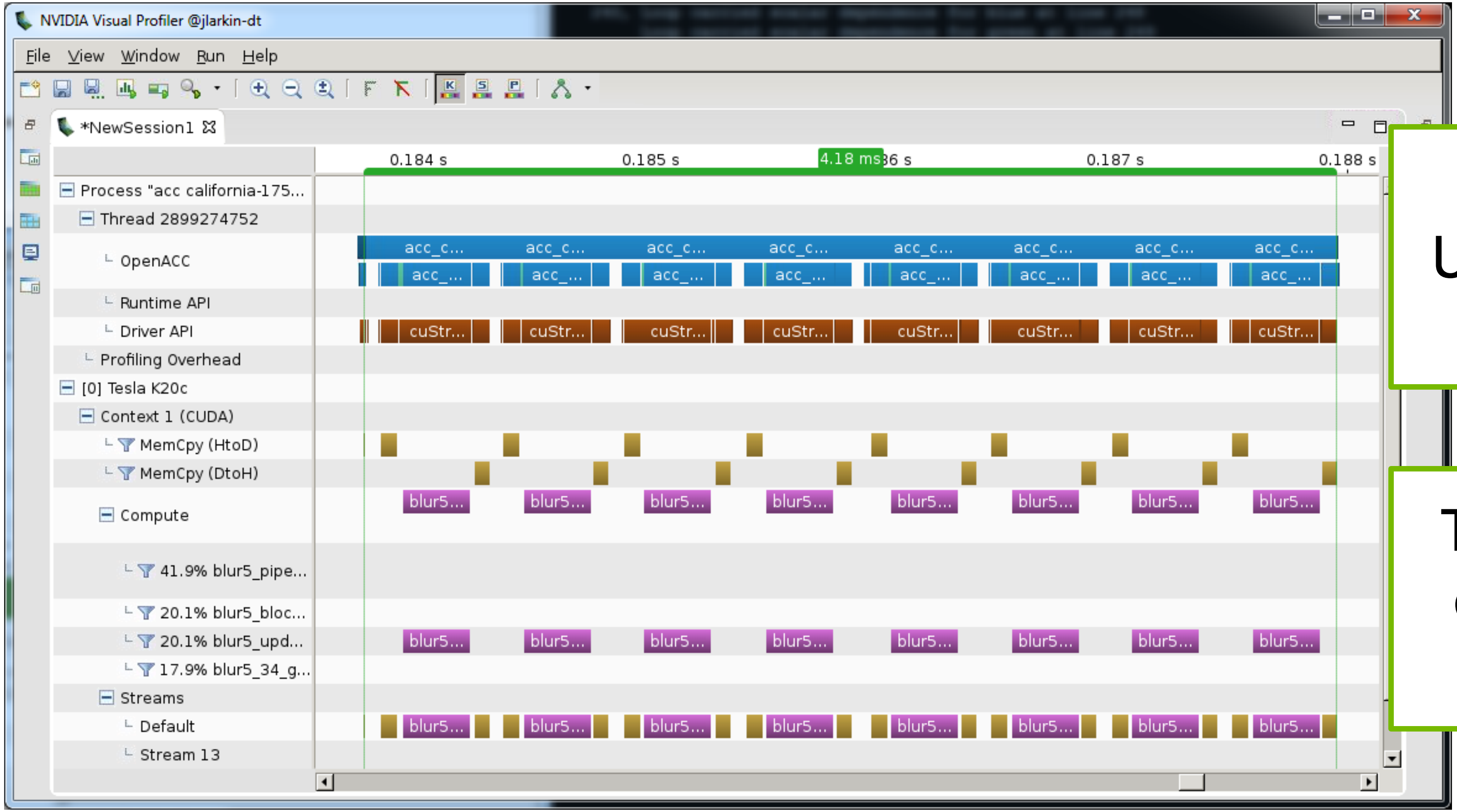
```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update
device(imgData[starty*step:blocksize*step])
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]      = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step])
}
```

Change data clauses to create

Update data one block at a time.

Copy results back one block at a time.

GPU Timeline Blocked Updates



Compute and Updates happen in blocks.

The last step is to overlap compute and copy.

OpenACC async and wait

async(n): launches work asynchronously in *queue n*

wait(n): blocks host until all operations in *queue n* have completed

Can significantly reduce launch latency and enables pipelining and concurrent operations

```
#pragma acc parallel loop async(1)
...
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc wait(1)
for(int i=0; i<N; i++)
```

If *n* is not specified, *async* will go into a default queue and *wait* will wait all previously queued work.

Pipelined Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3+1)
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3+1)
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]      = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3+1)
}
#pragma acc wait
}
```

Cycle between 3 async queues by blocks.

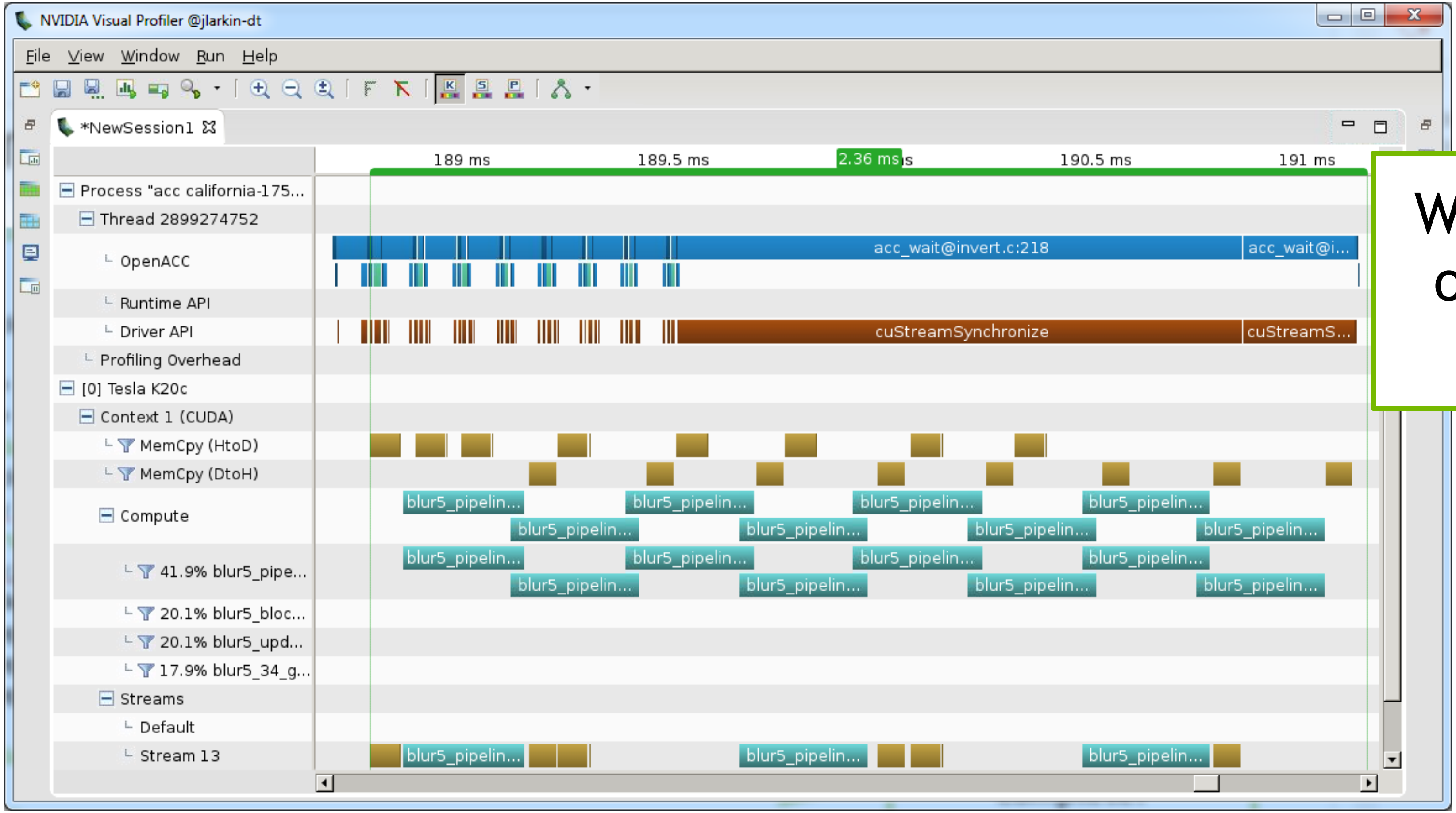
Pipelined Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                 copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3+1)
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3+1)
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]      = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3+1)
}
#pragma acc wait
}
```

Cycle between 3 async queues by blocks.

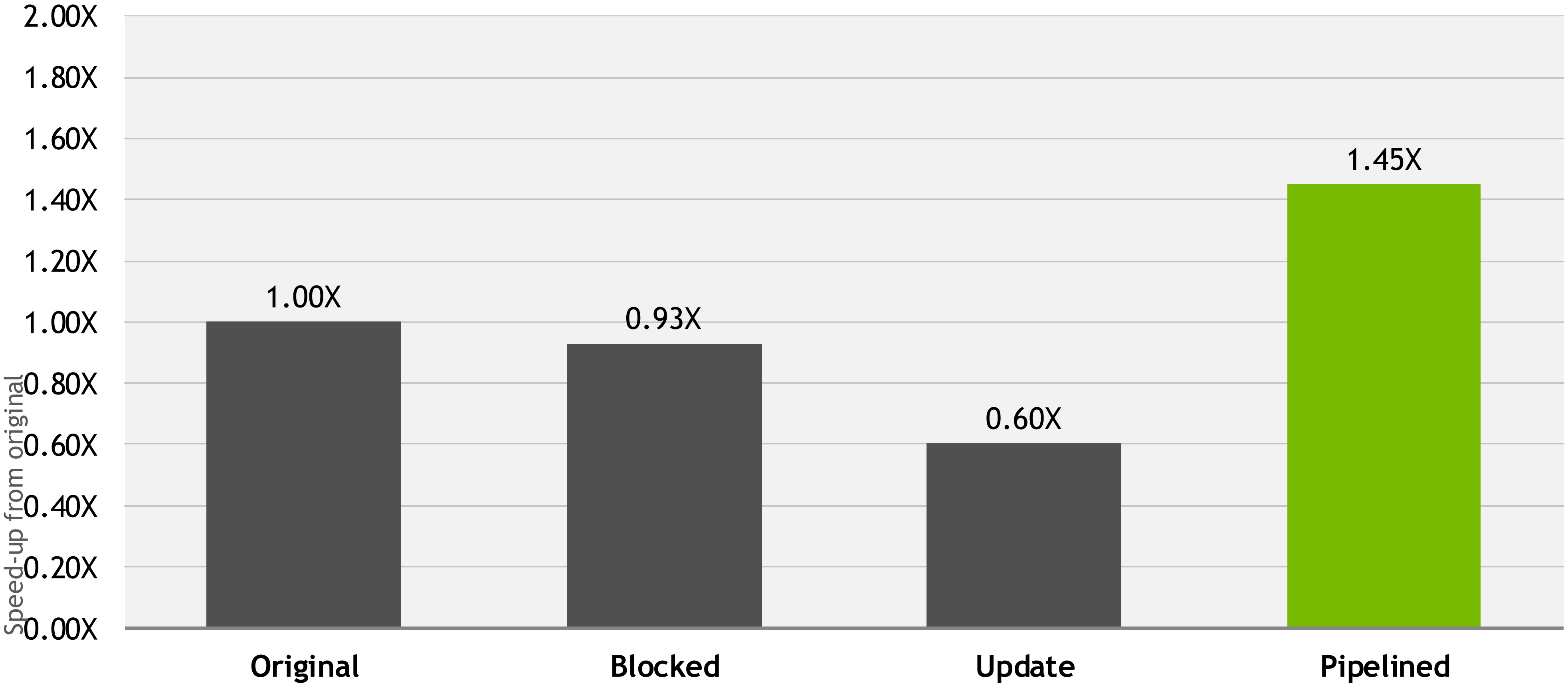
Wait for all blocks to complete.

GPU Timeline Pipelined



We're now able to overlap compute and copy.

Step-by-Step Performance



Source: PGI 16.9, NVIDIA Tesla K20c

Multi-GPU Pipelining

Multi-GPU OpenACC with OpenMP

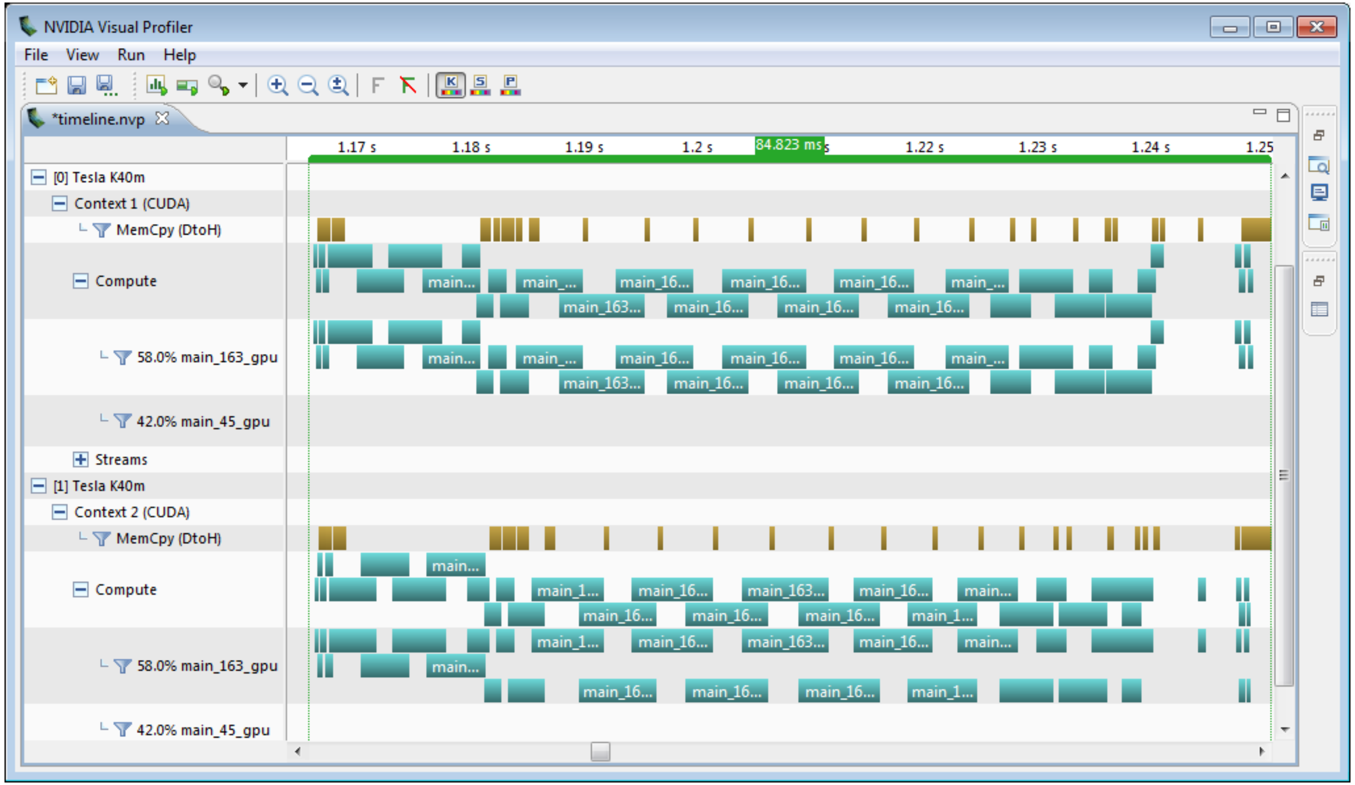
```
#pragma omp parallel
{
  int my_gpu = omp_get_thread_num();
  acc_set_device_num(my_gpu,acc_device_nvidia);

  #pragma acc data create(image[0:HEIGHT*WIDTH])
  {
    int queue = 1;
    #pragma omp for schedule(static,1) firstprivate(queue)
    for(int block=0; block < numblocks; block++)
    {
      int ystart = block * blocksize;
      int yend   = ystart + blocksize;
      #pragma acc parallel loop async(queue)
      for(int y=ystart;y<yend;y++) {
        for(int x=0;x<WIDTH;x++) {
          image[y*WIDTH+x]=mandelbrot(x,y);
        }
      }
      #pragma acc update self(image[ystart*WIDTH:WIDTH*blocksize]) async(queue)
      queue = queue%2+1;
    }
    #pragma acc wait
  }
}
```

Set the device number, all work will be sent to this device.

Use multiple queues per device to get copy compute overlap

Wait for all work to complete (per device)

Multi-GPU Mandelbrot Profile



OpenACC Interoperability

OpenACC Interoperability

OpenACC plays well with others.

Add CUDA or accelerated libraries to an OpenACC application

Add OpenACC to an existing accelerated application

Share data between OpenACC and CUDA

OpenACC host_data Directive

Exposes the *device* address of particular objects to the *host* code.

```
#pragma acc data copy(x,y)
{
// x and y are host pointers
#pragma acc host_data use_device(x,y)
{
 // x and y are device pointers
}
// x and y are host pointers
}
```

X and Y are device pointers here

host_data Example

OpenACC Main

```
program main
  integer, parameter :: N = 2**20
  real, dimension(N) :: X, Y
  real               :: A = 2.0

  !$acc data
  ! Initialize X and Y
  ...

  !$acc host_data use_device(x,y)
  call saxpy(n, a, x, y)
  !$acc end host_data
  !$acc end data

end program
```

- It's possible to interoperate from C/C++ or Fortran.
- OpenACC manages the data and passes device pointers to CUDA.

CUDA C Kernel & Wrapper

```
__global__
void saxpy_kernel(int n, float a,
      float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a, float *dx, float *dy)
{
  // Launch CUDA Kernel
  saxpy_kernel<<<4096,256>>>(N, 2.0, dx, dy);
}
```

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

CUBLAS Library & OpenACC

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust
- …

OpenACC Main Calling CUBLAS

```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...

cublasInit();

#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {

    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}

cublasShutdown();
```

Using QwikLabs

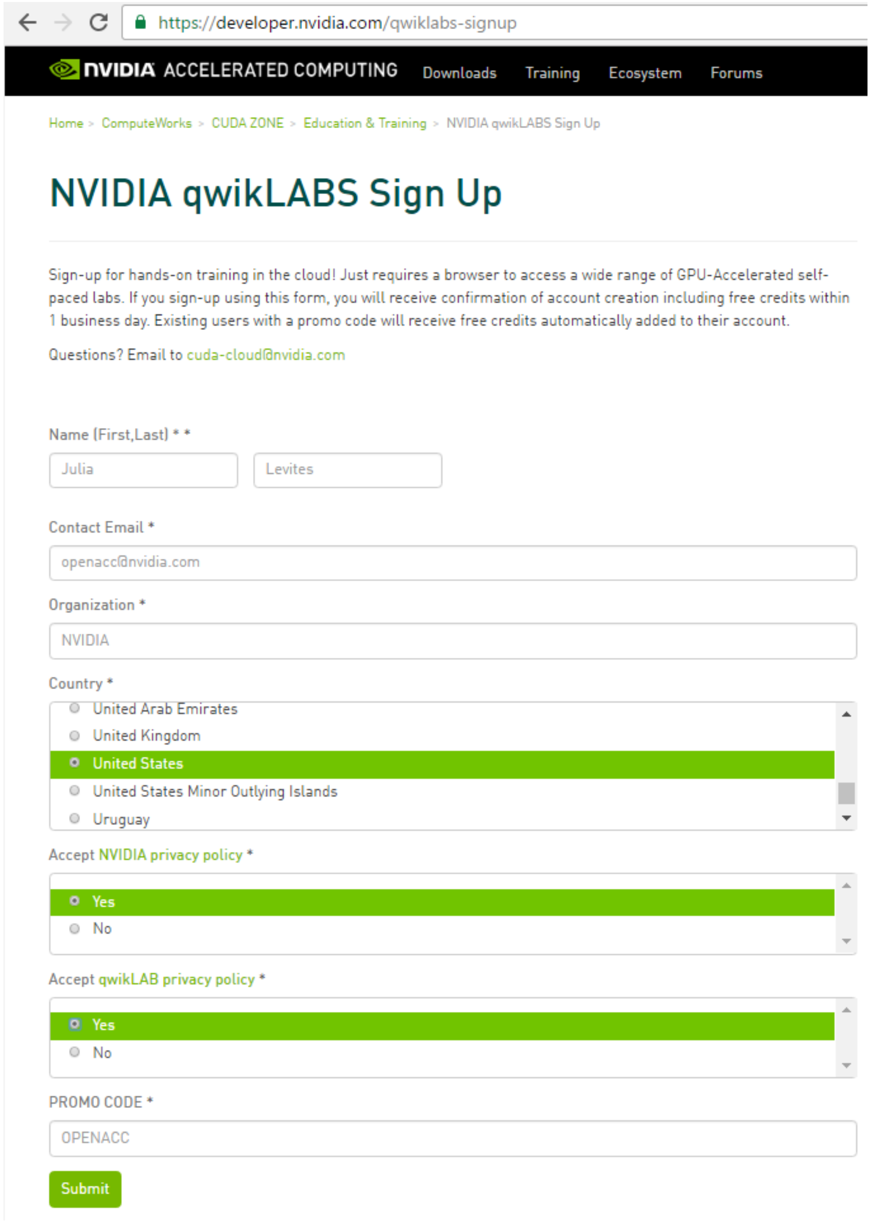
Getting access

1. Create an account with NVIDIA qwikLABS https://developer.nvidia.com/qwiklabs-signup

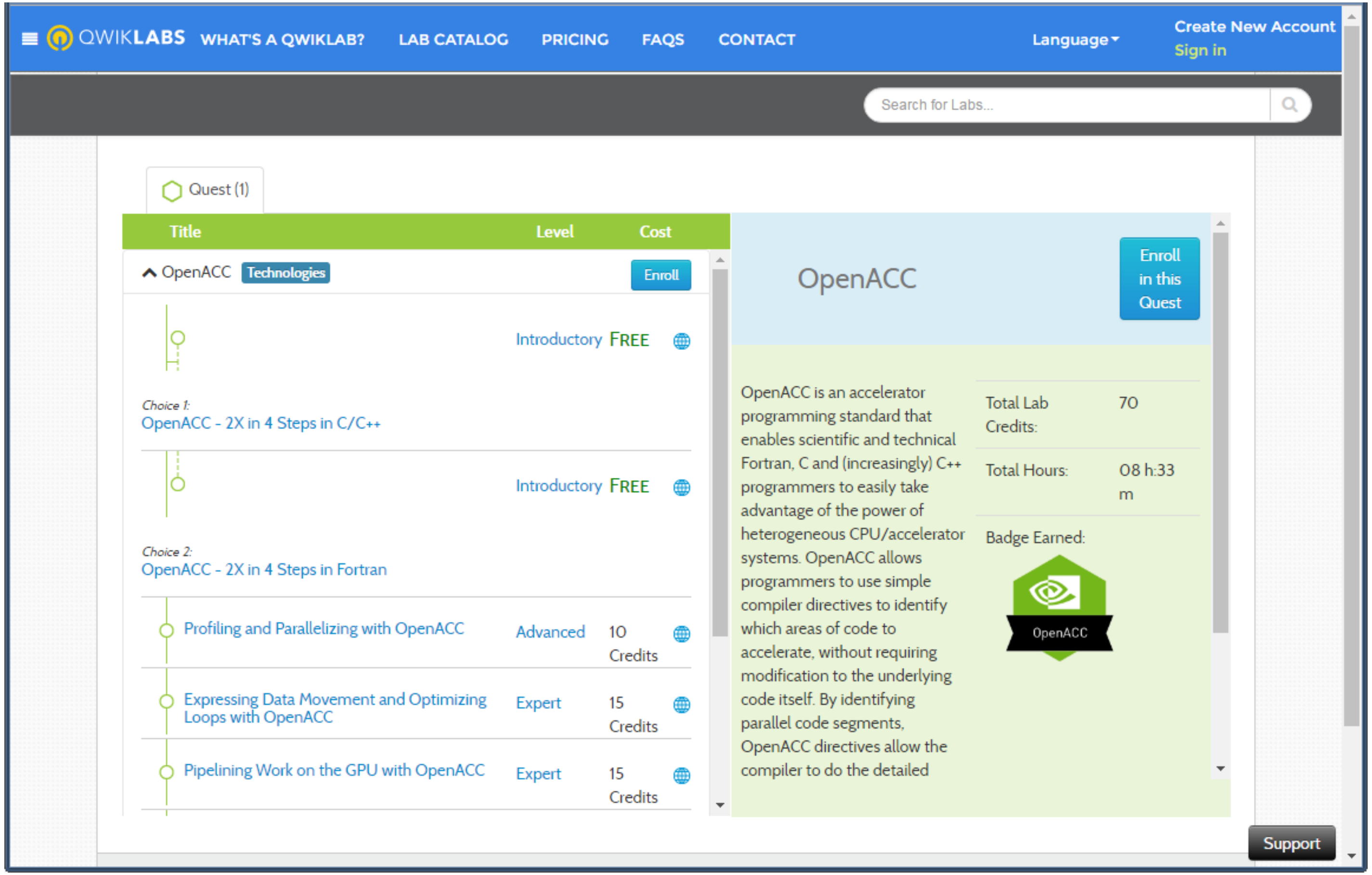
2. Enter a promo code OPENACC16 before submitting the form

3. Free credits will be added to your account

4. Start using OpenACC!



This week's labs

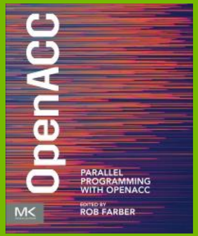


This week you should complete the "Pipelining Work on the GPU with OpenACC" lab. Please send questions to openacc@nvidia.com.

CERTIFICATION

Available after November 9th

1. Attend live lectures
2. Complete the test
3. Enter for a chance to win a Titan X or an OpenACC Book



Official rules:
http://developer.download.nvidia.com/compute/OpenACC-Toolkit/docs/TITANX-GIVEAWAY-OPENACC-Official-Rules-2016.pdf

OPENACC TOOLKIT

Free for Academia

Download link:
https://developer.nvidia.com/openacc-toolkit

NEW OPENACC BOOK

Parallel Programming with OpenACC

Now Available:

http://store.elsevier.com/Parallel-Programming-with-OpenACC/Rob-Farber/isbn-9780124103979/

Where to find help

- OpenACC Course Recordings - https://developer.nvidia.com/openacc-courses
- PGI Website - http://www.pgroup.com/resources
- OpenACC on StackOverflow - http://stackoverflow.com/questions/tagged/openacc
- OpenACC Toolkit - http://developer.nvidia.com/openacc-toolkit
- Parallel Forall Blog - http://devblogs.nvidia.com/parallelforall/
- GPU Technology Conference - http://www.gputechconf.com/
- OpenACC Website - http://openacc.org/

Questions? Email openacc@nvidia.com