Advanced OpenACC

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Outline

Loop Directives

Data Declaration Directives

Data Regions Directives

Cache directives

Wait / update directives

Runtime Library Routines

Environment variables

.

.

.

Outline

- How OpenACC work is organized
  - Gangs/Workers/Threads
  - kernels
  - parallel regions
- Things you didn't know were missing (OpenACC 2.0)
  - Procedure calls
  - Nested Parallelism
- More complex hardware configurations
  - Device Specific Tuning
  - Multi-threading and multiple devices
  - Alternative threading approaches
- Using asynchronous features
- Manual Data Management
- Profiling

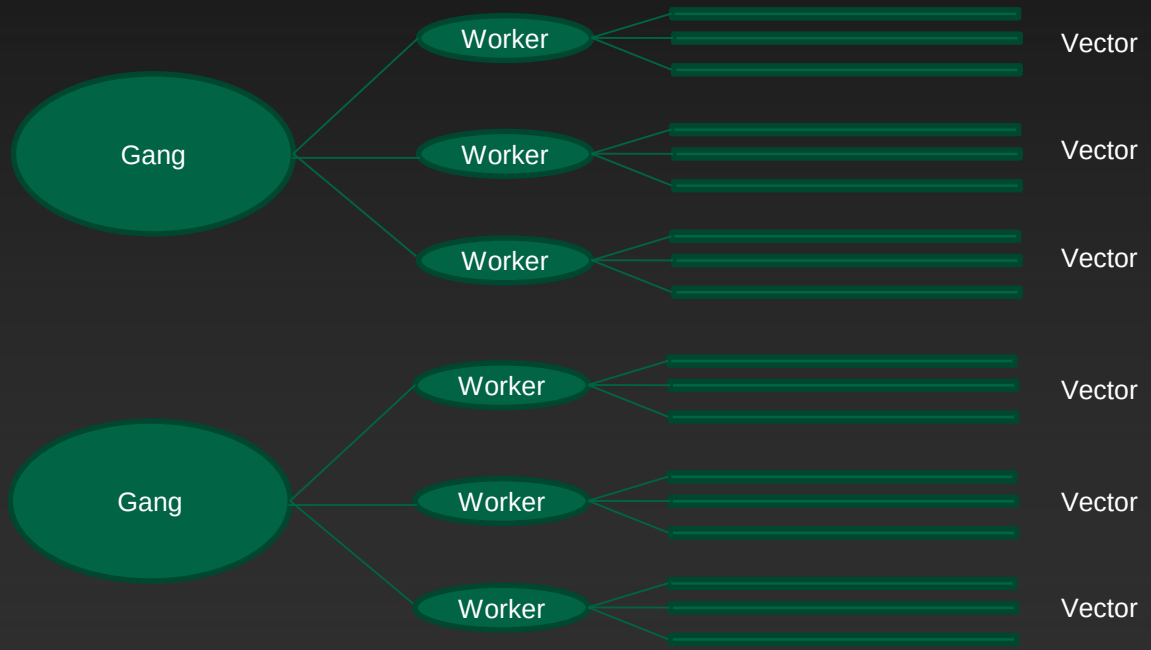
Targeting the Architecture (But Not Admitting It)

Part of the awesomeness of OpenACC has been that you have been able to ignore the hardware specifics. But, now that you know a little bit more about CUDA/GPU architecture, you might suspect that you can give the compiler still more help in optimizing. In particular, you might know the hardware specifics of a particular model. The compiler might only know which "family" it is compiling for (Fermi, Kepler, etc.).

Indeed, the OpenACC spec has some clauses to target architecture hierarchies, and not just GPUs (think Intel MIC). Let's see how they map to what we know about GPUs.

OpenACC Task Granularity

- The OpenACC execution model has three levels: *gang*, *worker* and *vector*
- This is supposed to map to any architecture that is a collection of Processing Elements (PEs) where each PE is multithreaded and each thread can execute vector instructions.

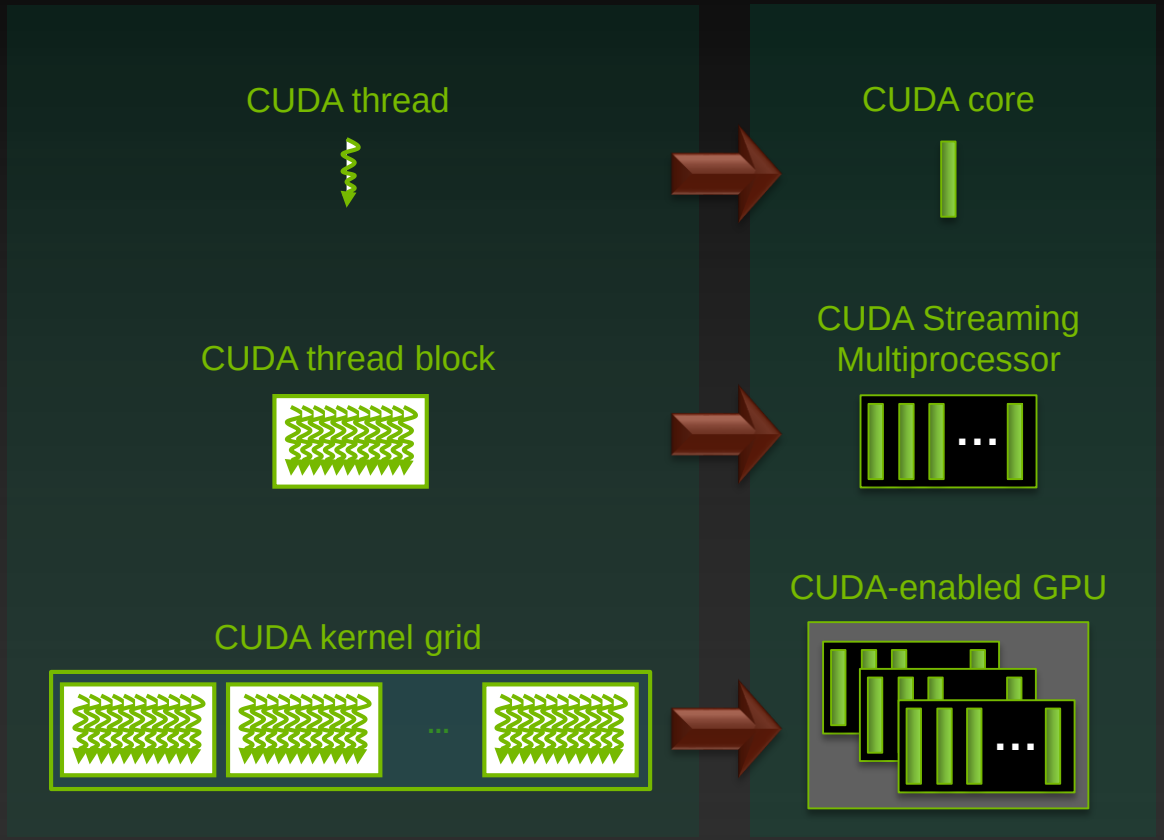


Targeting the Architecture

As we said, OpenACC assumes a device will contain multiple processing elements (PE) that run in parallel. Each PE also has the ability to efficiently perform vector-like operations. For NVIDIA GPUs, it is reasonable to think of a PE as a streaming multiprocessor (SM). Then an OpenACC *gang* is a threadblock, a *worker* is effectively a warp, and an OpenACC *vector* is a CUDA thread. Phi, or similar Intel SMP architectures also map in a logical, but different, fashion.

	GPU	SMP (Phi)
Vector	Thread	SSE Vector
Worker	Warp	Core
Gang	SM	CPU

NVIDIA GPU Task Granularity (Take Notes!)



- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Warps – on Kepler (Still taking notes?)

- Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future
- The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds
- All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken
- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible

Determining block size - on Kepler (You can stop now)

- 32 thread wide blocks are good for Kepler, since warps are allocated by row first.
  - 32 thread wide blocks will mean all threads in a warp are reading and writing contiguous pieces of memory
  - Coalescing
- Try to keep total threads in a block to be a multiple of 32 if possible
  - Non-multiples of 32 waste some resources & cycles
- Total number of threads in a block: between 256 and 512 is usually a good number.

Determining grid size - on Kepler

- Most people start with having each thread do one unit of work
- Usually better to have fewer threads so that each thread could do multiple pieces of work.
- What is the limit to how much smaller we can make the number of total blocks?
  - We still want to have at least as many threads as can fill the GPU many times over (for example 4 times). That means we need at least 2880 x 15 x 4 = ~173,000 threads
  - Experiment by decreasing the number of threads

Mapping OpenACC to CUDA Threads and Blocks

```
#pragma acc kernels
for( int i = 0; i < n; ++i )
     y[i] += a*x[i];
```

16 blocks, 256 threads each.

```
#pragma acc kernels loop gang(100) vector(128)
for( int i = 0; i < n; ++i )
     y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop.

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
   #pragma acc loop gang vector
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel

SAXPY Returns For Some Fine Tuning

The default (will work OK):

```
#pragma acc kernels loop
for( int i = 0; i < n; ++i )
        y[i] += a*x[i];
```

Some suggestions to the compiler:

```
#pragma acc kernels loop gang(100), vector(128)
for( int i = 0; i < n; ++i )
         y[i] += a*x[i];
```

Specifies that the kernel will use 100 thread blocks, each with 128 threads, where each thread executes one iteration of the loop. This beat the default by ~20% *last time I tried…*

Rapid Evolution

	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	54	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations	16k/48k	16k/48k	16k/32k/48k	16k/32k/48k
Max X Grid Dimension	2^16	2^16	2^32	2^32
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

From http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

- Do you want to have to keep up with this?
- Maybe the compiler knows more about this than you? Is that possible?
- CUDA programmers do have to worry about all of this, and much more.
- But doesn't hurt much to try.

Parallel Regions vs. Kernels

We have been using *kernels* thus far, to great effect. However OpenACC allows us to more explicitly control the creation of tasks via the *gang*, *worker* and *vector* clauses. We can do this inside of *parallel* regions.

These approaches come from different backgrounds.

PGI Accelerator *region*		OpenACC *kernels*
OpenMP *parallel*		OpenACC *parallel*

Parallel Construct

Fortran

```
!$acc parallel [clause …]
    structured block
!$acc end parallel
```

C

```
#pragma acc parallel [clause …]
    { structured block }
```

Clauses

```
if( condition )
async( expression )
num_gangs( expression )
num_workers( expression )
vector_length( expression )
```

```
private( list )
firstprivate( list )
reduction( operator:list )
```

Also any data clause

Parallel Clauses

`num_gangs ( expression )`	Controls how many parallel gangs are created.
`num_workers ( expression )`	Controls how many workers are created in each gang.
`vector_length ( list )`	Controls vector length of each worker.
`private( list )`	A copy of each variable in list is allocated to each gang.
`firstprivate ( list )`	`private` variables initialized from host.
`reduction( operator:list )`	`private` variables combined across gangs.

Parallel Regions

As in OpenMP, the OpenACC parallel construct creates a number of parallel gangs that immediately begin executing the body of the construct *redundantly*. When a gang reaches a work-sharing loop, that gang will execute a subset of the loop iterations. One difference between the OpenACC parallel construct and OpenMP is that there is no barrier at the end of a work-sharing loop in a parallel construct.

SAXPY as a parallel region

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
#pragma acc loop gang, vector
for( int i = 0; i < n; ++i )
        y[i] += a*x[i];
}
```

Compare and Contrast

Let's look at how this plays out in actual code.

This

```
#pragma acc kernels
{
        for( i = 0; i < n; ++i )
                a[i] = b[i] + c[i];
}
```

Is the same as

```
#pragma acc parallel
{
        #pragma acc loop
        for( i = 0; i < n; ++i )
                a[i] = b[i] + c[i];
}
```

Compare and Contrast

But not

```
#pragma acc parallel
{
        for( i = 0; i < n; ++i )
                a[i] = b[i] + c[i];
}
```

By leaving out the loop directive we get totally redundant execution of the loop by each gang. This is not desirable.

Parallel Regions vs. Kernels

From these simple examples you could get the impression that simply putting in loop directives everywhere would make parallel regions equivalent to kernels. That is not the case.

The sequence of loops here

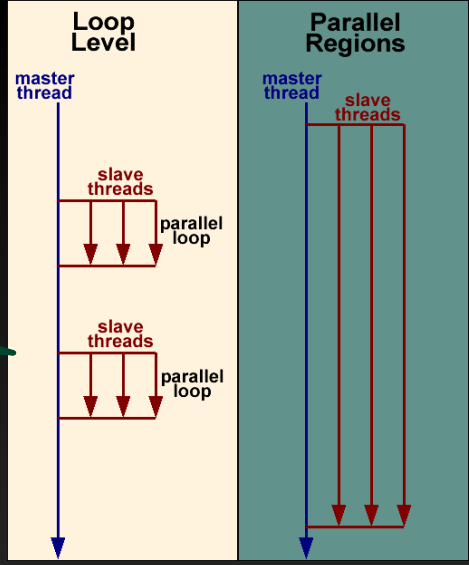
```
#pragma acc kernels
{
for (i=0; i<n; i++)
        a(i) = b(i)*c(i)
for (i=1; i<n-1; i++)
        d(i) = a(i-1) + a(i+1)
}
```

Does what you might think. Two kernels are generated and the first completes before the second starts.

A parallel region will work differently

```
#pragma acc parallel
{
#pragma acc loop
for (i=0; i<n; i++)
        a(i) = b(i)*c(i)
#pragma acc loop
for (i=1; i<n-1; i++)
        d(i) = a(i-1) + a(i+1)
}
```

Straight from the pages of our OpenMP lecture!



The compiler will start some number of gangs, work-share the iterations of the first loop across those gangs, and work-share the iterations of the second loop across the same gangs. There is no guarantee that for the same loop iteration, the same value of i will be executed by the same gang for both loops. In fact, that's likely to be untrue, and some value of i will be executed by different gangs between the first and second loop. There is also no synchronization between the first and second loop, so there's no guarantee that the assignment to a(i) from the first loop will be complete before its value is fetched by some other gang for the assignment in the second loop.

Parallel Regions vs. Kernels (Which is best?)

To put it simply, kernels leave more decision making up to the compiler. There is nothing wrong with trusting the compiler (“trust but verify”) and that is probably a reasonable place to start.

If you are an OpenMP programmer, you will notice a strong similarity between the tradeoffs of kernels and regions and that of OpenMP parallel for/do versus parallel regions. We will discuss this later when we talk about OpenMP 4.0.

As you gain experience, you may find that the parallel construct allows you to apply your understanding more explicitly. On the other hand, as the compilers mature, they will also be smarter about just doing the right thing. *History tends to favor this second path heavily.*

OpenACC 2.0 & 2.5

Things you didn't know were missing.

The latest version of the specification has a lot of improvements. The most anticipated ones remove limitations that you, as new users, might not have known about. However, they may still linger until all of the compilers get up to spec.

- Procedure Calls
- Nested Parallelism

As well as some other things that you might not have thought about

- Device specific tuning
- Multiple host thread support

Don't be afraid to review the full spec at

http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf

Procedure Calls

In OpenACC 1.0, all procedures had to be inlined. This limitation has been removed, but you do need to follow some rules.

```
#pragma acc routine worker
extern void solver(float* x, int n);
.
.
.
#pragma acc parallel loop num_gangs(200)
for( int index = 0; index < N; index++ ){
          solver( X, n);
          .
          .
}
```

```
#pragma acc routine worker
void solver(float* x, int n){
.
.
          #pragma acc loop
          for( int index = 0; index < n; index++ ){
                    x[index] = x[index+2] * alpha;
                    .
                    .
          }
          .
}
```

In this case, the directive tells the compiler that “solver” will be a device executable and that it may have a loop at the worker level. No caller can do worker level parallelism.

Nested Parallelism

The previous example had gangs invoking workers. But, it is now possible to have kernels actually launch new kernels.

```
#pragma acc routine
extern void solver(float* x, int n);
.
.
#pragma acc parallel loop
for( int index = 0; index < N; index++ ){
          solver( X, index);
}
```

```
#pragma acc routine
void solver(float* x, int n){
          #pragma acc parallel loop
          for( int index = 0; index < n; index++ ){
                    x[index] = x[index+2] * alpha
                    .
                    .
          }
          .
}
```

Having thousands of lightweight threads launching lightweight threads is probably not the most likely scenario.

Nested Parallelism

This is a more useful case. We have a single thread on the device launching parallelism from its thread.

```
#pragma acc routine
extern void solver(float* x, int n);
.
.
#pragma acc parallel num_gangs(1)
{
          solver( X, n1 );
          solver( Y, n2 );
          solver( Z, n3 );
}
```

```
#pragma acc routine
void solver(float* x, int n){
          #pragma acc parallel loop
          for( int index = 0; index < n; index++){
                    x[index] = x[index+2] * alpha;
                    .
                    .
          }
          .
}
```

The objective is to move as much of the application to the accelerator and minimize communication between it and the host.

Device Specific Tuning

I hope from our brief detour into GPU hardware specifics that you have some understanding of how hardware specific these optimizations can be. Maybe one more reason to let kernel do its thing. However, OpenACC does have ways to allow you to account for various hardware details. The most direct is device_type().

```
#pragma acc parallel loop  device_type(nvidia) num_gangs(200) \
                           device_type(radeon) num_gangs(800)
for( index = 0; index < n; index++ ){
        x[i] += y[i];
        solver( x, y, n );
}
```

Multiple Devices and Multiple Threads

- Multiple threads and one device: fine. You are responsible for making sure that the data is on the multi-core host when it needs to be, and on the accelerator when it needs to be there. But, you have those data clauses in hand already (present_or_copy will be crucial), and OpenMP has its necessary synchronization ability.

- Multiple threads and multiple devices. One might hope that the compilers will eventually make this transparent, but at the moment you need to:

  - Assign threads to devices:
    - omp_get_thread_num
    - call acc_set_device_num
  - Manually break up the data structure into several pieces:
    - `!$acc kernels loop copyin(x(offs(i)+1:offs(i)+nsec),y(offs(i)+1:offs(i)+nsec))`
    - From excellent example on Page 15 of the PGI 12.6 OpenACC Getting Started Guide

Asynchronous Behavior

There are synchronization rules associated with each type of loop construct, and some of the data constructs (involving updates and independent data management). You may want to keep them in mind if you drift very far from a *kernels* model. In those cases you have *wait()*, *asynch()* and *atomic* clauses, directives or APIs to manage your flow. There are several variations of each to accommodate multiple types of conditions to continue (one or multiple waits, test or block).

As data movement can take so much time, overlapping computation by using these commands can be very effective.

Data Management

Again, as you get farther from a simple *kernels* model, you may find yourself needing to manage data transfers in a more explicit manner. You can manually:

- Create global data:
  - *declare create* (create on host and device, you will probably use *update* to manage)
  - *declare device_resident* (create on device only, only accessible in compute regions)
  - *declare link* and *declare create pointer* (pointers are created for data to be copied)
- Create data transfer regions: *enter data* (in effect until *exit data*).
  - Like *copyin*, etc. except that they do not need to apply to a structured block. Can just stick one in some initialization routine.
- Update data directly: *update*

You should never find yourself frustrated for lack of control. You can move data at will with the available options. And you can be fearless with the new "OK to copy even if data is already there" default (the old *present_* commands are obsolete).

There are clause, directive and even API versions of these, depending on appropriateness.

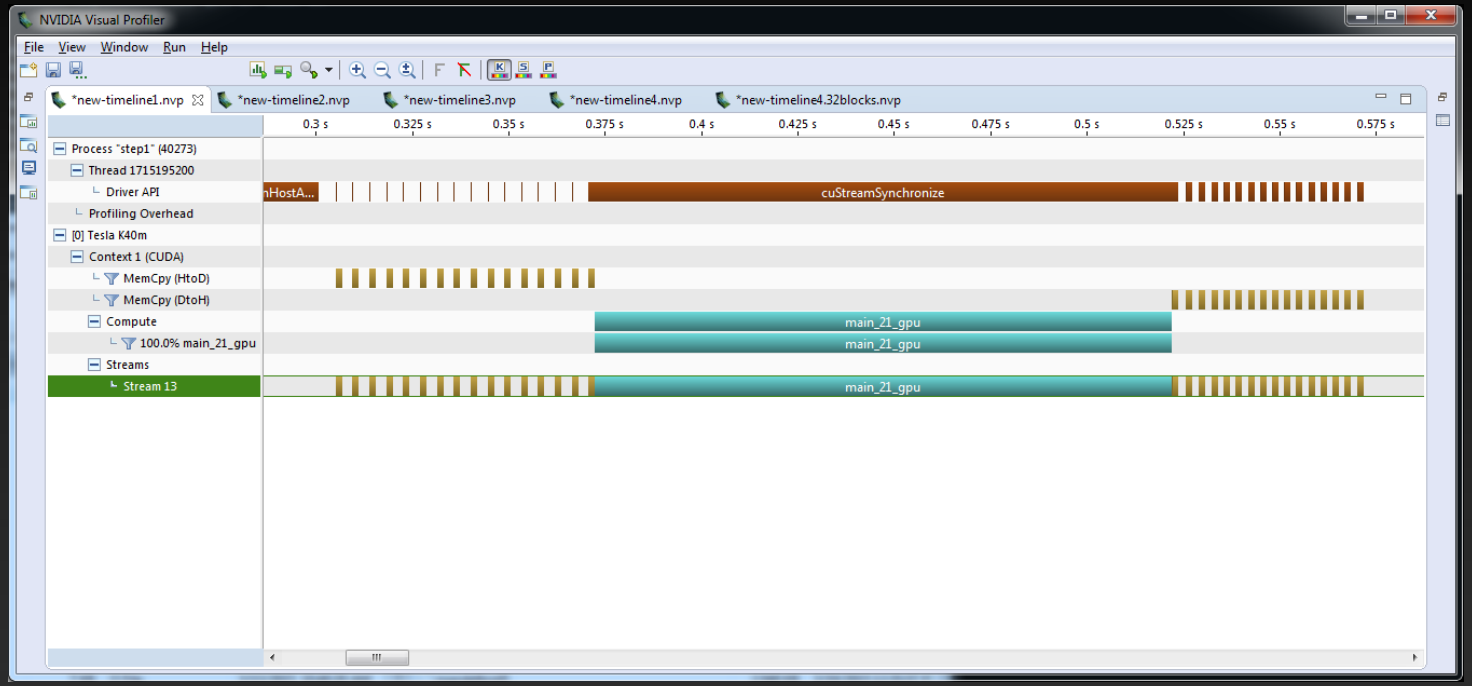
Profiling

So, how do you recognize these problems (opportunities!) besides the relatively simple timing output we have used in this class?

One benefit of the NVIDIA ecosystem is the large number of tools from the CUDA community that we get to piggyback upon.

The following uses the NVIDIA Visual Profiler which is part of the CUDA Toolkit.

Mandlebrot Code

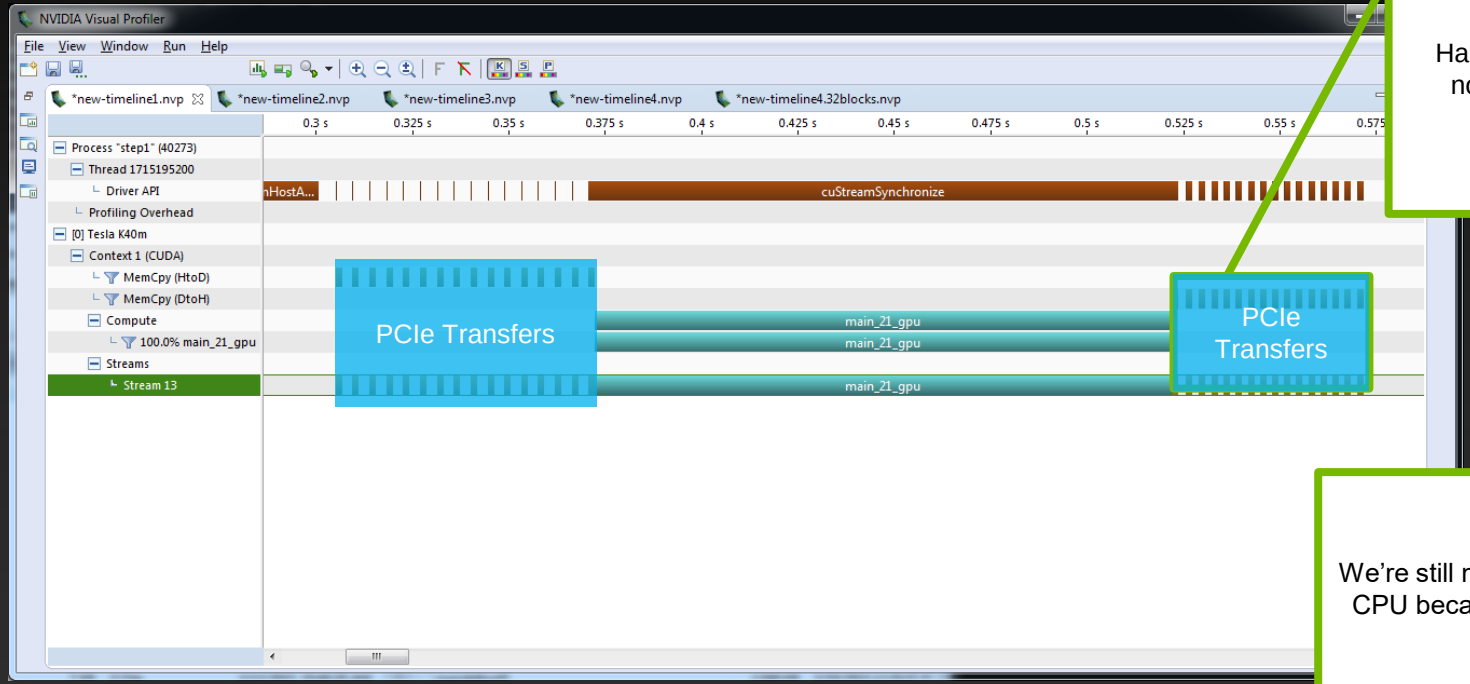


This is for an OpenACC Mandlebrot set image generation code from NVIDIA . You can grab it at

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

Lots of Data Transfer Time

Step 1 Profile

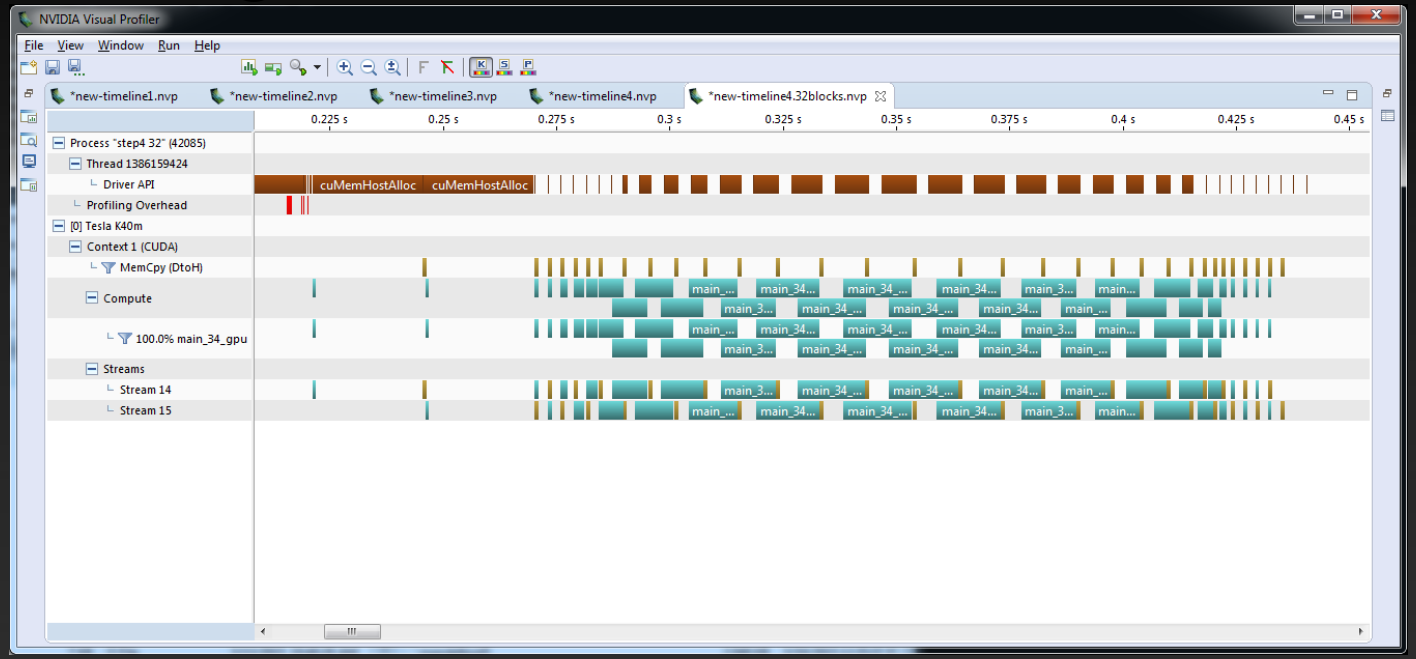


Half of our time is copying, none of it is overlapped.

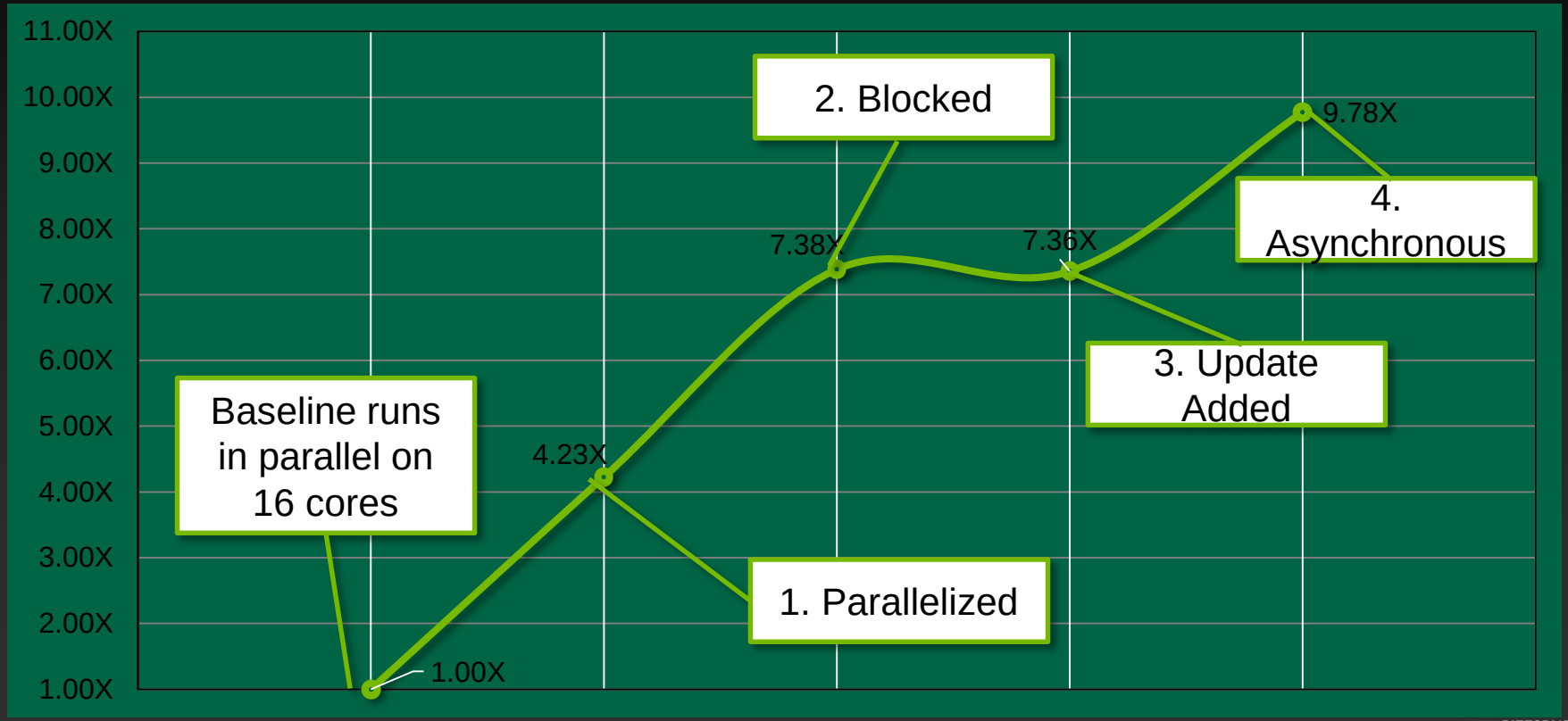
We're still much faster than the CPU because there's a lot of work.

Broken Into Blocks With Asynchronous Transfers

Pipelining with 32 blocks



Optimized In A Few Well-Informed Stages



OpenACC Things Not Covered

The OpenACC specification has grown quite accommodating as of Version 2.5. You have already seen some redundancy between directives, clauses and APIs, so I have made no attempt to do "laundry lists" of every option along the way. It would be quite repetitive. I think you are well prepared to glance at the OpenACC Specification and grasp just about all of it.

We have omitted various and sundry peripheral items. Without attempting to be comprehensive, here are a few topics of potential interest to some of you.

- **Environment variables: Useful for different hardware configurations**
- **if clauses, macros and conditional compilation: allow both runtime and compile time control over host or device control flow.**
- **API versions of nearly all directives and clauses**
- **Hybrid programming. *Works great!* Don't know how meaningful this is to you…**

Credits

Some of these examples are derived from excellent explanations by these gentlemen, and more than a little benefit was derived from their expertise.

Michael Wolfe, PGI

Jeff Larkin, NVIDIA

Mark Harris, NVIDIA

Cliff Woolley, NVIDIA