Advanced OpenACC

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center
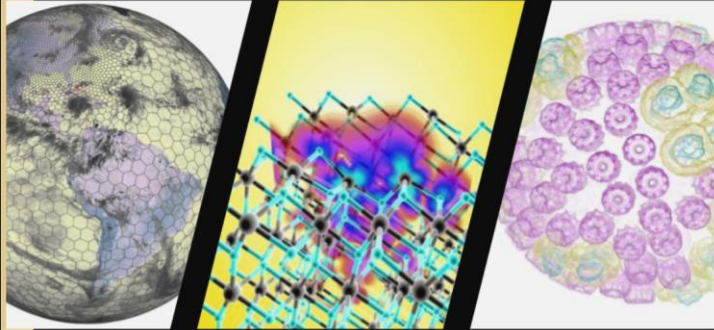
This is a goldmine of practical application of the advanced features we are about to discuss. I suggest at least giving it a glance.

https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf

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, Pascal etc.).

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

PITTSBURGH
SUPERCOMPUTING
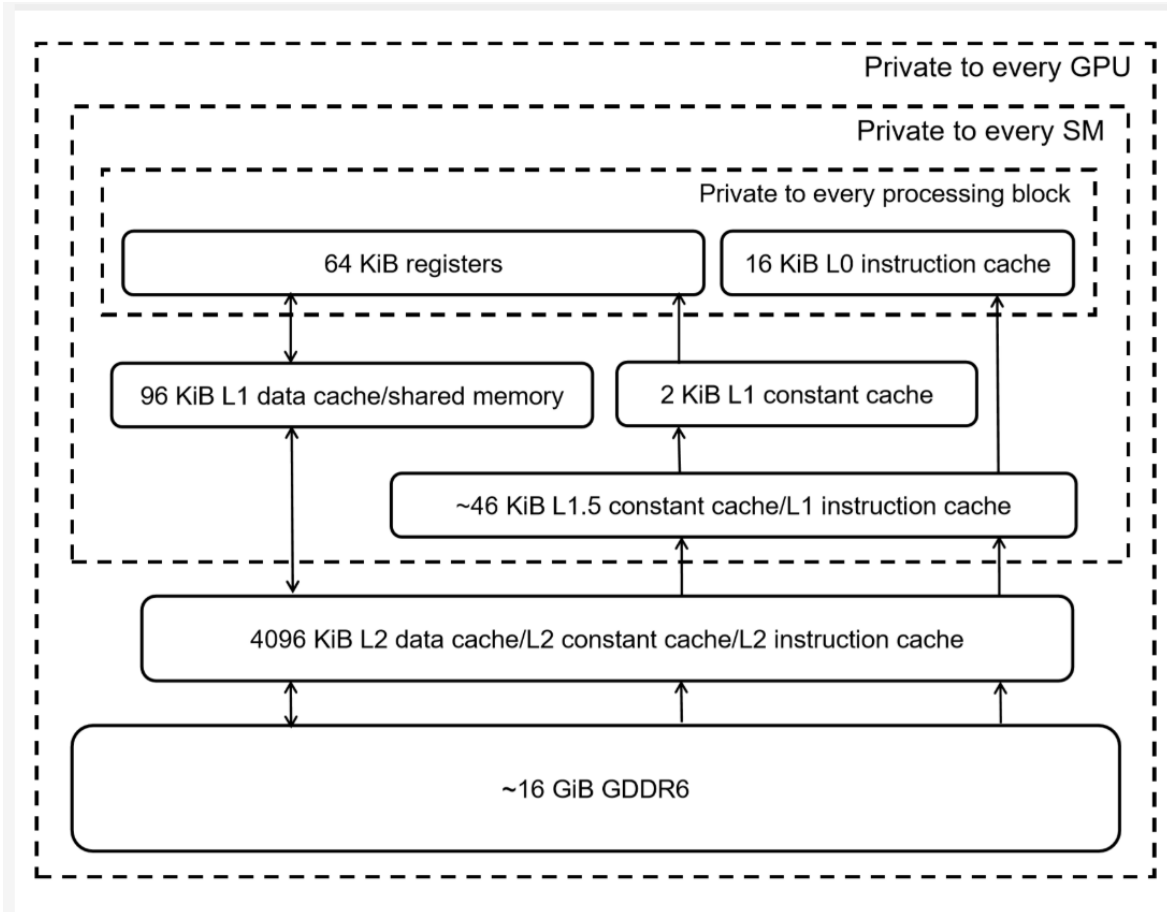CENTER

V100 GPU and SM

Volta GV100 GPU with 85 Streaming Multiprocessor (SM) units

Volta GV100 SM

Turing Memory Hierarchy

Image source: Dissecting the NVidia Turing T4 GPU via Microbenchmarking | Citadel Enterprise Americas LLC

CUDA Execution Model
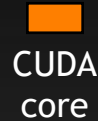
Software

Thread

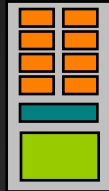**Thread Block**

Grid

Hardware

CUDA core

Multiprocessor (SM)

Device

Threads are executed by CUDA cores

Thread blocks are executed on multiprocessors (SM)

- Thread blocks do not migrate
- Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Blocks and grids can be multi dimensional (x,y,z)

Quantum of independence: Warps

Thread Block = Warps (32 Threads, 32 Threads, 32 Threads) → Multiprocessor

- A thread block consists of one or more warps

- A warp is executed physically in parallel (SIMD) on a multiprocessor

- The SM creates, manages, schedules and executes threads at warp granularity

- 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

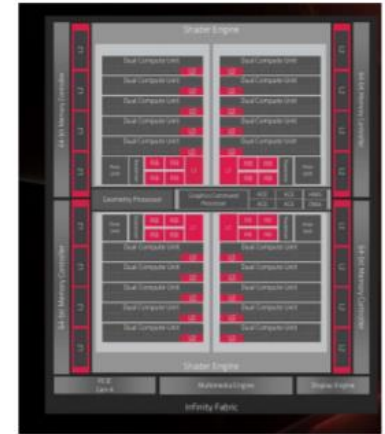- Currently all NVIDIA GPUs use a warp size of 32

But Every Generation and Manufacturer Is Different
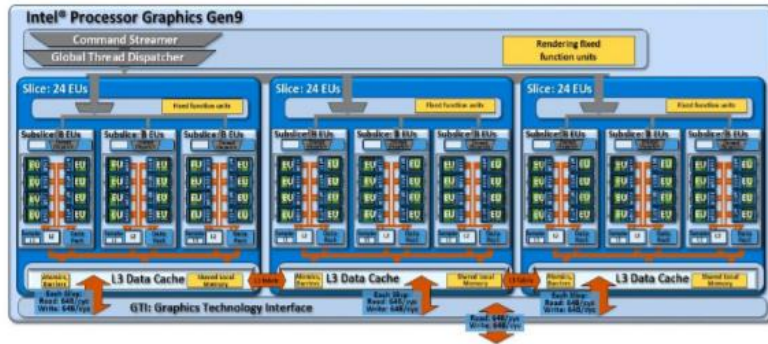
Intel Gen9 GPU

NVIDIA V100

AMD

And The Terminology Changes

Nvidia/CUDA Terminology	AMD Terminology	Intel Terminology	Description (pulled almost word-for-word from AMD slides)
Streaming Multiprocessor (SM)	Compute Unit (CU)	SubSlice (SS)	One of many independent parallel vector processors in a GPU that contain multiple SIMD ALUs.
Kernel	Kernel	Kernel	Functions launched to the GPU that are executed by multiple parallel workers on the GPU.
Warp	Wavefront	Vector thread, issuing SIMD instruction	Collection of operations that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off.
Thread block	Workgroup	Workgroup	Group of warps/wavefronts/vector threads that are on the GPU at the same time. Can synchronize together and communicate through local memory.
Thread	Work item/Thread	Work item/Vector lane	Individual lane in a warp/wavefront/vector thread
Global Memory	Global Memory	GPU Memory	DRAM memory accessible by the GPU that goes through some layers of cache
Shared memory	Local memory	Shared local memory	Scratchpad that allows communication between warps/wavefronts/vector threads in a threadblock/workgroup
Local memory	Private memory	GPRF	Per-thread private memory, often mapped to registers.

Rapid Evolution

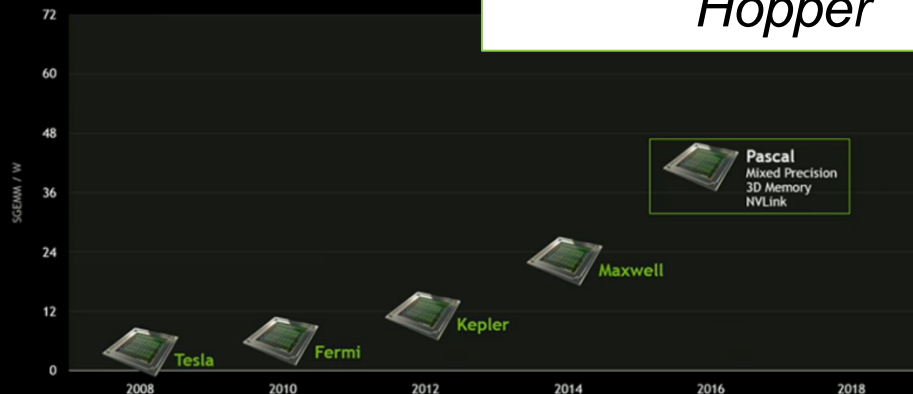|	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110	Maxwell GM107	Pascal GP100
Compute Capability	2.0	2.1	3.0	3.5	5.0	6.0
Threads / Warp	32	32	32	32	32	32
Max Warps / Multiprocessor	48	48	54	64	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048	2048	
Max Thread Blocks / Multiprocessor	8	8	16	16	32	
32-bit Registers / Multiprocessor						
Max Registers / Thr						
Max Threads / Threa						
Shared Memory Siz Configurations						
Max X Grid Dimensi						
Hyper-Q						
Dynamic Parallelism						

- **Do you want to have to keep up with this?**

- **Maybe the compiler knows more about this than you? Is that possible?**

Evolution continues with:
 Turing
 Ampere
 Hopper

GPU ROADMAP
Pascal 2x SGEMM/W

Pascal
Mixed Precision
3D Memory
NVLink

Maxwell

Kepler

Fermi

Tesla

SGEMM / W

72

60

48

36

24

12

0

2008   2010   2012   2014   2016   2018

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOP/s[*]	5.04	6.8	10.6	15.7
Peak FP64 TFLOP/s[*]	1.68	.21	5.3	7.8
Peak Tensor Core TFLOP/s[*]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

This is a *good* thing.

- **Don't put yourself in a situation where this becomes a maintenance nightmare.**

- **Leave that problem to the compiler writers.**

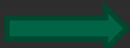# OpenACC Task Granularity

- 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.
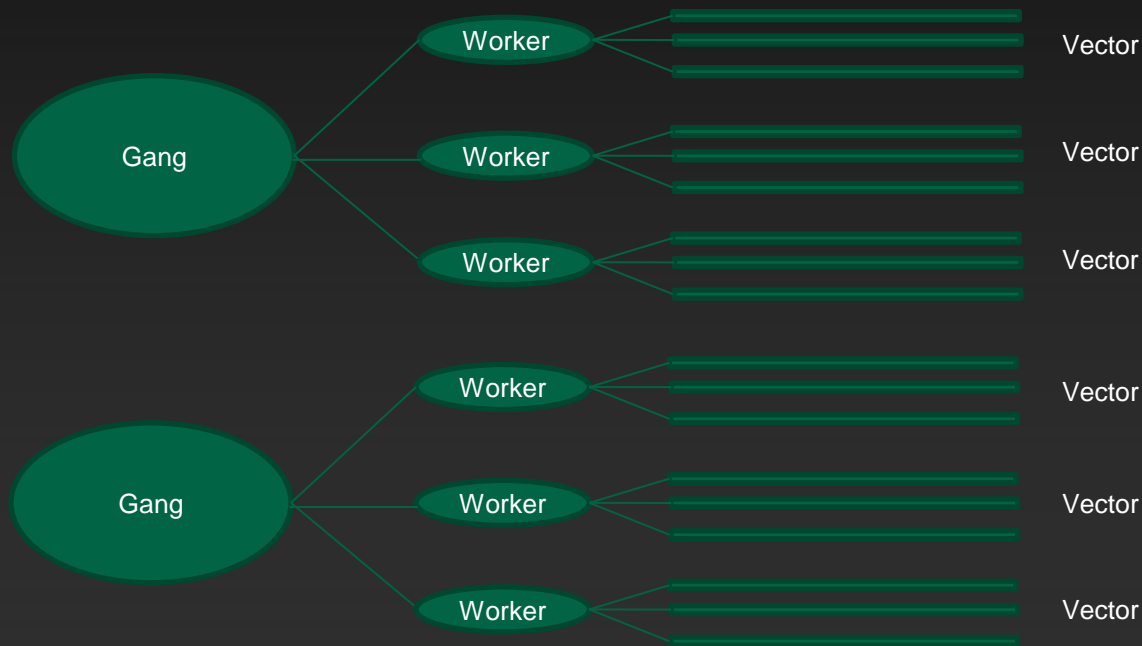
Targeting the Architecture

As we said, OpenACC assumes a device will contain multiple processing elements (PEs) 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

Kepler, for example

- Block Size Optimization:
  - 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.

- Grid Size Optimization:
  - 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…*

Parallel Regions vs. Kernels

We have been using *kernels* thus far, to great effect. However OpenACC allows us to very explicitly control the flow and allocation of tasks using *parallel* regions.

These approaches come from different backgrounds.

PGI Accelerator
Region*   →   OpenACC
                                         kernels

OpenMP
parallel   →   OpenACC
                                         parallel

*Similar philosophy to preferring OpenMP *omp parallel for*

Parallel Regions

When you start an accelerator parallel region, one or more gangs of workers are created to execute the accelerator parallel region. The number of gangs, and the number of workers in each gang and the number of vector lanes per worker remain constant for the duration of that parallel region.

Each gang begins executing the code in the structured block in gang-redundant mode. This means that code within the parallel region, but outside of a loop construct with gang-level worksharing, will be executed _redundantly_ by all gangs. One worker in each gang begins executing the code in the structured block of the construct.

This means you are setting the cores free, allowing them to attack the work with maximum efficiency. Now it is up to you to coral them into some kind of sensible activity!

PITTSBURGH
SUPERCOMPUTING
CENTER

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.
copy(),copyin(),copyout(),create()	Same behavior we already know.
present(list)	Variable already there from some other data clause. Suppress any desire of compiler to copy and do nothing.
asynch()/wait()	Just getting to this in a few slides...

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 major 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] = 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];
}
```

Don't Do This

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, to say the least.

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.
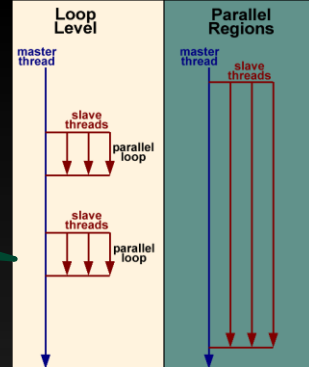
The sequence of loops here

```
#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 and then 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 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. This will result in incorrect results.

But the most common reason we use parallel regions is because we want to eliminate these wasted blocking cycles. So we just need some means of controlling them…

Controlling Waiting

We can allow workers, or our CPU, to continue ahead while a loop is executing as we wait at the appropriate times (so we don't get ahead of our data arriving or a calculation finishing. We do this with **asynch** and **wait** statements.

```
#pragma acc parallel loop async(1)
for (i = 0; i < n; ++i)
  c[i] += a[i];
#pragma acc parallel loop async(2)
for (i = 0; i < n; ++i)
  b[i] = expf(b[i]);
#pragma acc wait
// host waits here for all async activities to complete
```

We are combining the parallel region and loop directives together. A common idiom.

Note that there is no sync available *within* a parallel region (or kernel)!

Using Separate Queues

We have up to 16 queues that we can use to manage completion dependencies.

```
#pragma acc parallel loop async(1)            // on queue 1
for (i = 0; i < n; ++i)
  c[i] += a[i];
#pragma acc parallel loop async(2)            // on queue 2
for (i = 0; i < n; ++i)
  b[i] = expf(b[i]);
#pragma acc parallel loop async(1) wait(2)    // waits for both
for (i = 0; i < n; ++i)
  d[i] = c[i] + b[i];
// host continues executing while GPU is busy
```

Dependencies

We can use these with kernels too.

```
#pragma acc kernels loop independent async(1)
for (i = 1; i < n-1; ++i) {
  #pragma acc cache(b[i-1:3], c[i-1:3])
  a[i] = c[i-1]*b[i+1] + c[i]*b[i] + c[i+1]*b[i-1];
}
#pragma acc parallel loop async(2) wait(1)      // start queue 2
for (i = 0; i < n; ++i)                          // after 1
  c[i] += a[i];                                  // need a to finish
#pragma acc parallel loop async(3) wait(1)      // start queue 3
for (i = 0; i < n; ++i)                          // after 1
  b[i] = expf(b[i]);                             // don't mess with b
// host continues executing while GPU is busy
```

Private Variables

One other important consideration for parallel regions is what happens with scaler (non-array) variables inside loops. Unlike arrays, which are divided up amongst the cores, the variables are shared by default. This is often not what you want.

If you have a scaler inside a parallel loop that is being changed, you probably want each core to have a *private* copy. This is similar to what we saw earlier with a reduction variable.

```fortran
integer nsteps, i
double precision step, sum, x
nsteps = ...
sum = 0
step = 1.0d0 / nsteps
!$acc parallel loop private(x) reduction(+:sum)
do i = 1, nsteps
  x = (i + 0.5d0) * step
  sum = sum + 1.0 / (1.0 + x*x)
enddo
pi = 4.0 * step * sum
```

Consistent with this philosophy, scaler variables default to *firstprivate* inside of *parallel* regions where *kernel* regions default to *copy*. Both regions default to *copy* for aggregate types.

Loop Clauses

`private (list)` Each thread gets it own copy (implied for index variable).
`reduction (operator:list)` Also private, but combine at end. Your responsibility now!

`gang/worker/vector( )` We've seen these.

`independent` Independent. Ignore any suspicions.
`seq` Opposite. Sequential, don't parallelize.
`auto` Compiler's call.

`collapse()` Says how many nested levels apply to this loop. Unrolls. Good for small inner loops.

`tile(,)` Opposite. Splits each specified level of nested loop into two. Good for locality.

`device_type()` For multiple devices.

Kernels vs. Parallel

Advantages of kernels

- compiler autoparallelizes
- best with nested loops and no procedure calls
- one construct around many loop nests can create many device kernels

Advantages of parallel

- some compilers are bad at parallelization
- more user control, esp. with procedure calls
- one construct generates one device kernel
- similar to OpenMP

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._

Data Management

Again, as you get farther from a simple program, you may find yourself needing to manage data transfers in a more explicit manner. We restricted ourselves to the data copy type commands for our initial work, but still found update to be necessary. In general you won't find yourself frustrated for lack of a convenient data movement action.

enter data　　　　　Like *copyin* except that they do not need to apply to a structured block or scope. Could just stick one in some initialization routine. Clauses can be *async*, *wait*, *copyin* or *create*.

exit data　　　　　　Bookend of above, but in addition to *async* and *wait* has *copyout*, and *delete* (decrement reference count) and **finalize** (force count to zero).

update　　　　　　　As used earlier, but has *async*, *wait* and some other clauses.

Dynamic Memory

You may have wondered how these data transfers cope with dynamic memory. The answer is, very naturally as OpenACC is intended for serious codes which usually use dynamic allocation. Here is one way that you might find yourself allocating/deallocating a dynamic structure on both the host and device.

C

```
tmp = (double *) malloc(count*sizeof(double));
#pragma acc enter data create(tmp[0:count])
.
.
.
#pragma acc exit data delete(tmp)
free(tmp)
```

Fortran

```
allocate(tmp(count))
!$acc enter data create(tmp)
.
.
.
!$acc exit data delete(tmp)
deallocate(tmp)
```

Declare Directive

You can put your data movement specification close to your natural variable declarations.

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

Data Structures

Somebody has probably asked this by now, but if not, it is important for me to note that complex data structures are just fine for OpenACC. Feel free to use:

- **Complex structs**
- **C++ classes**
- **Fortran derived types**
- **Dynamically allocated memory**
- **STL?   Yes and No**

The major caveat is that pointer based structures will not naturally move from CPU to GPU. This should be no surprise. You must do you own "deep copy" if you need to move such data.
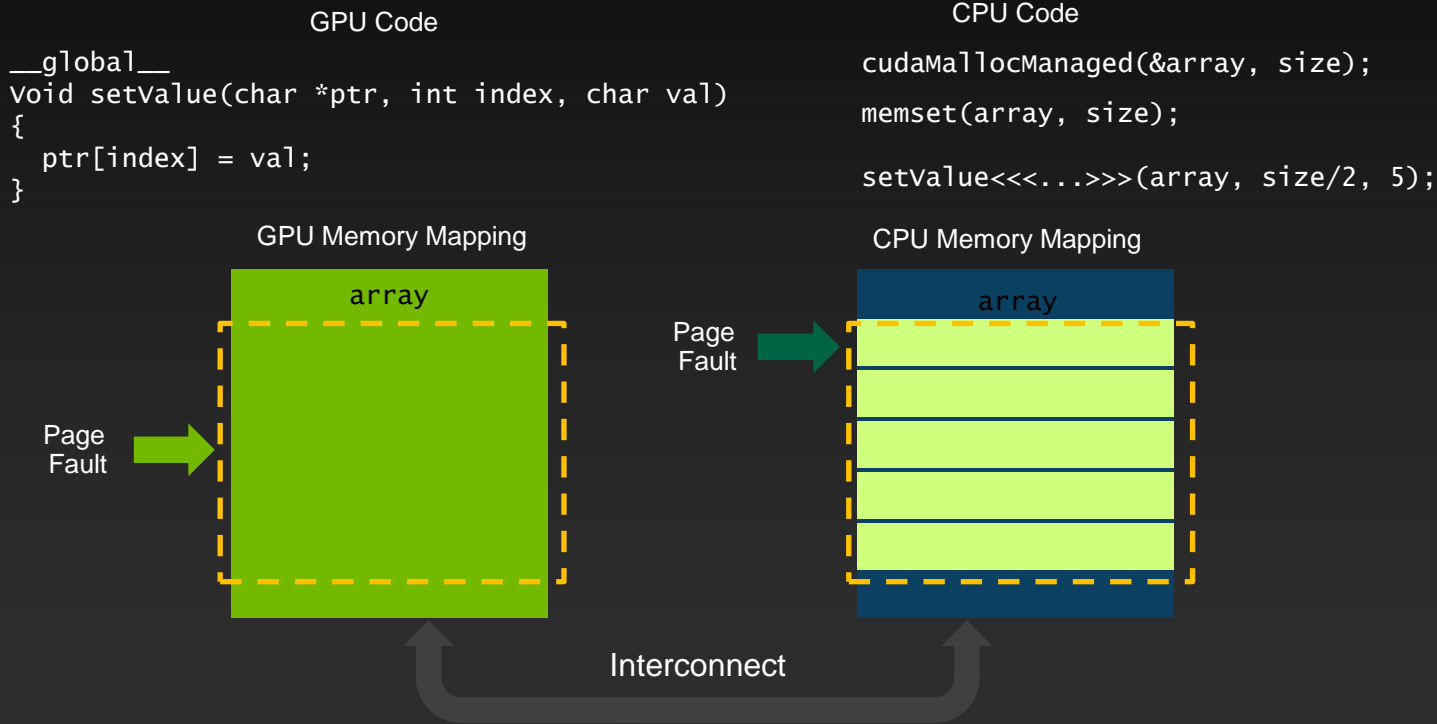
Cache Directive

CUDA programmers always want to know how to access CUDA shared memory. All of you should be interested in how you can utilize this small (~48KB) shared (by the gang) memory for items that should be kept close at hand.

```fortran
real temp(64)

!$acc parallel loop gang vector_length(64) private(temp)
do i = 1, n
  !$acc cache(temp)
  !$acc loop vector
  do j = 1, 64
    temp(j) = a(i,j)
    ....
```

CUDA Unified Memory*

Speaking of memory, a few realistic words are in order concerning the awesome sounding Unified Memory. No more data management?

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```

GPU Memory Mapping

array

Page Fault

CPU Memory Mapping

array

Page Fault

Interconnect

* "*CUDA 8 and Beyond*", Mark Harris, GPU Technology Conference, April 4-7, 2016

PITTSBURGH SUPERCOMPUTING CENTER

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 );
}
```

Line continuation syntax
(in case we haven't see this yet)

PITTSBURGH
SUPERCOMPUTING
CENTER

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 (i.e. logically merge devices into one), 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 25 of the PGI 2016 OpenACC Getting Started Guide

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.

Visual Profiler and our Laplace first attempt.

We zoom in to get a better view of the timeline. As expected, it looks like our program is spending a significant amount of time transferring data between the host and device. We also see that the compute regions are very small, with a lot of distance between them.
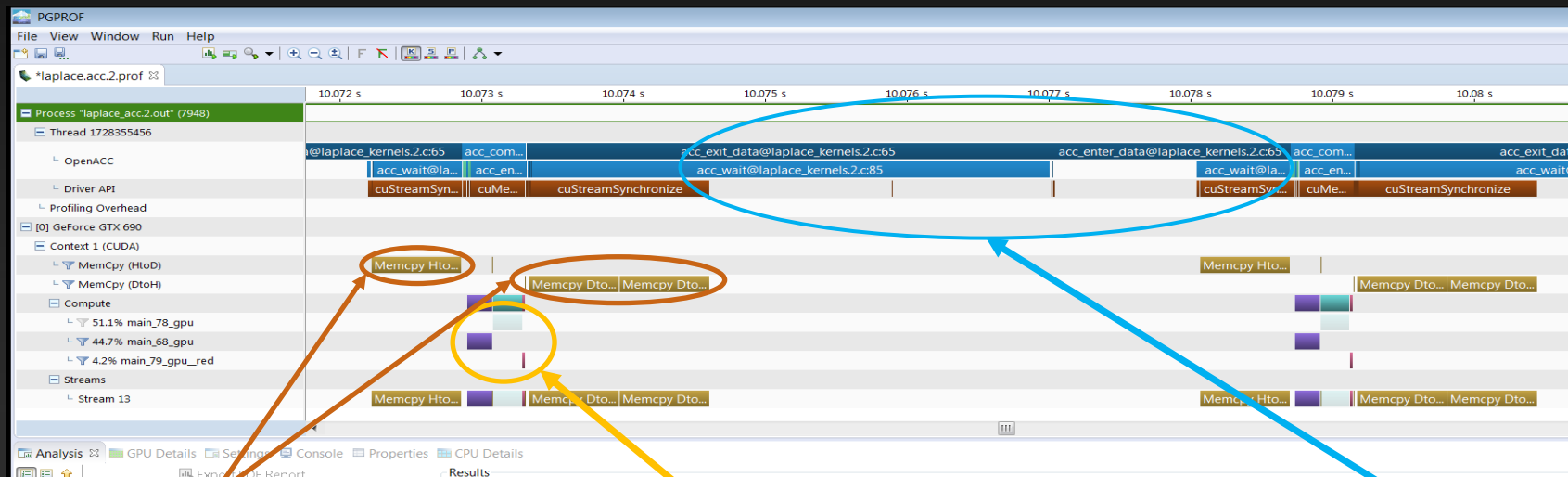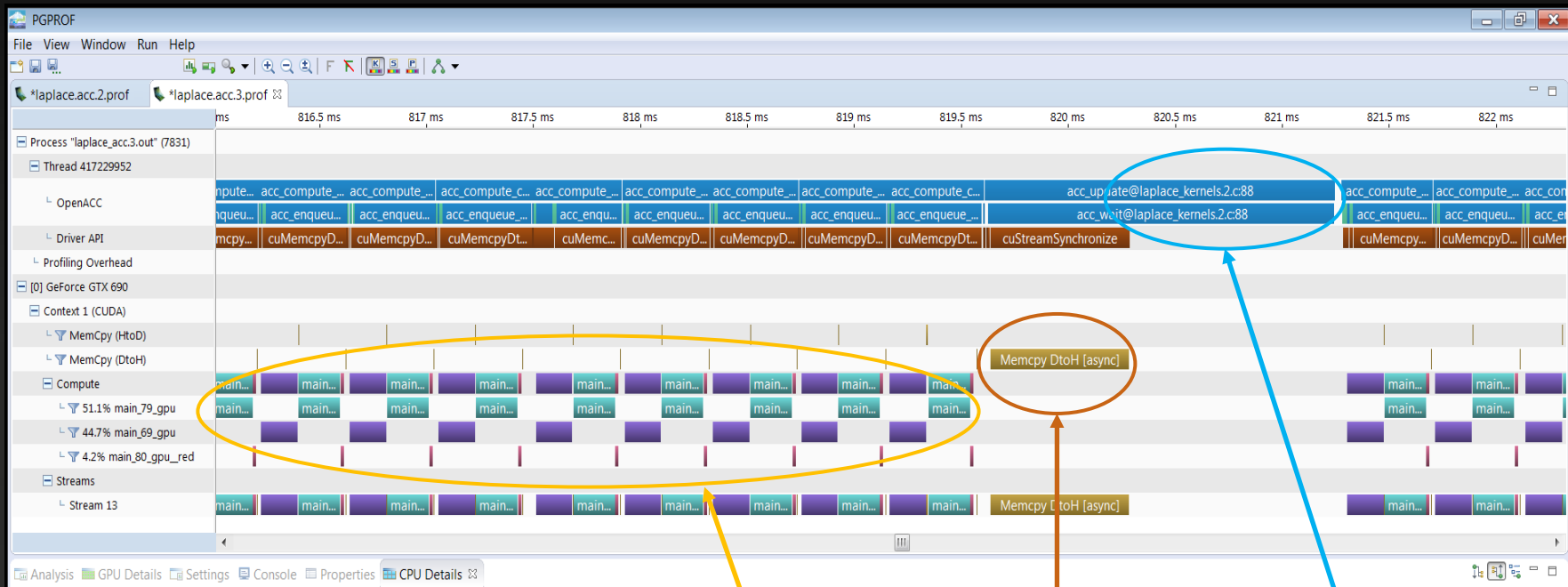

Device Memory Transfers

Compute on the Device

CPU overhead time including:
- Device data creation and deletion
- Host Memory Copy

After our data management fix.

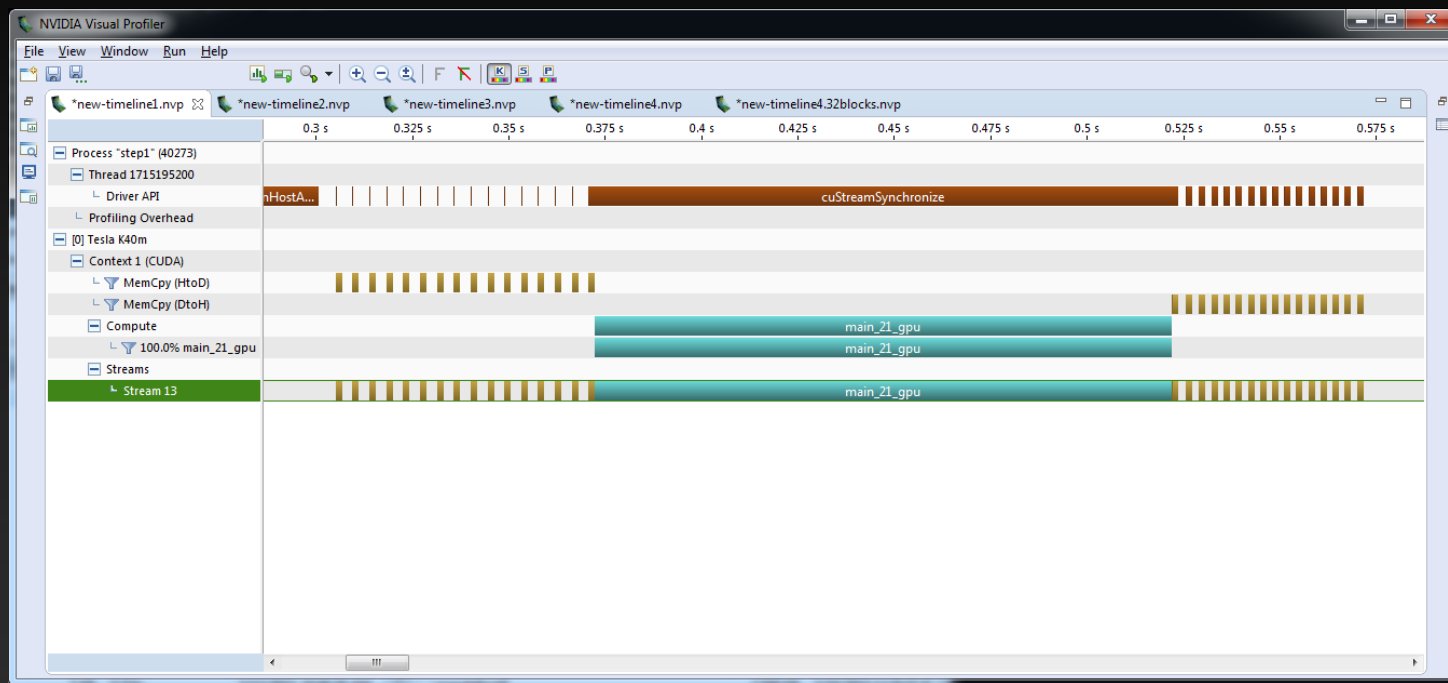After adding the data region, we see lots of compute and data movement only when we update.
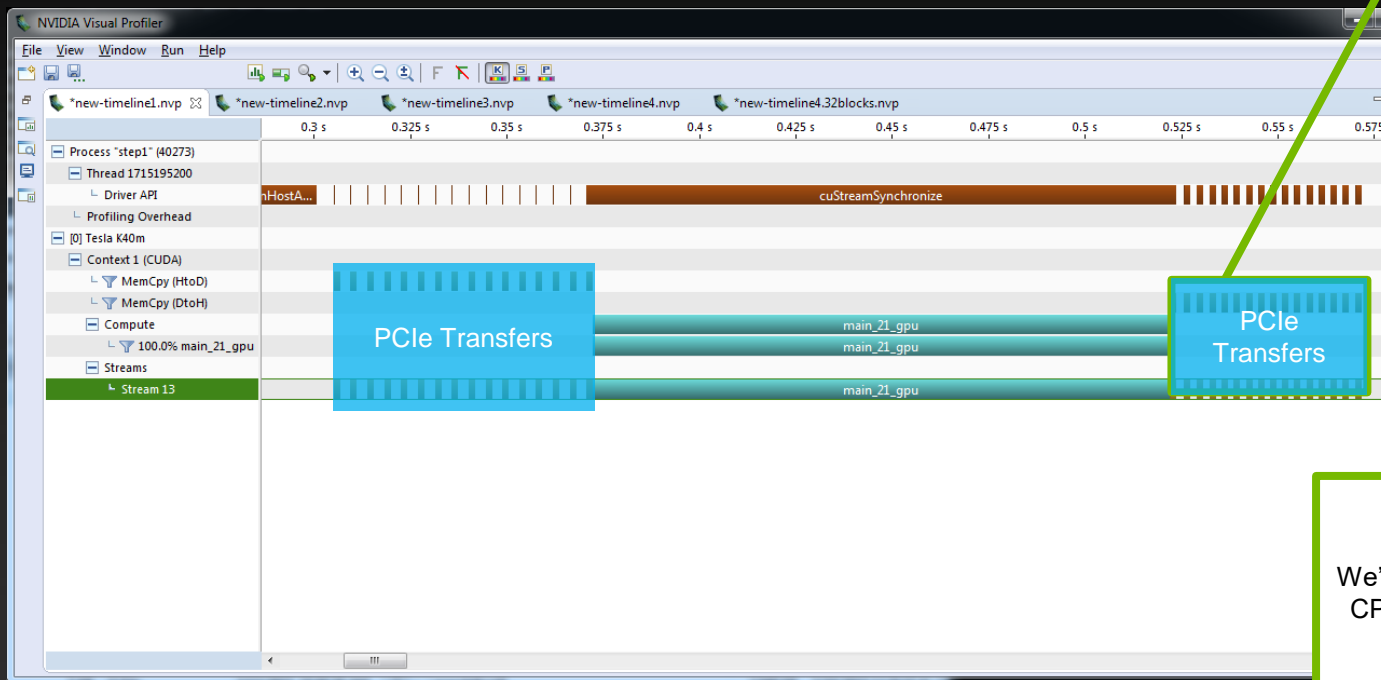

Device compute

Update to host

Host memory copy

PITTSBURGH SUPERCOMPUTING CENTER

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

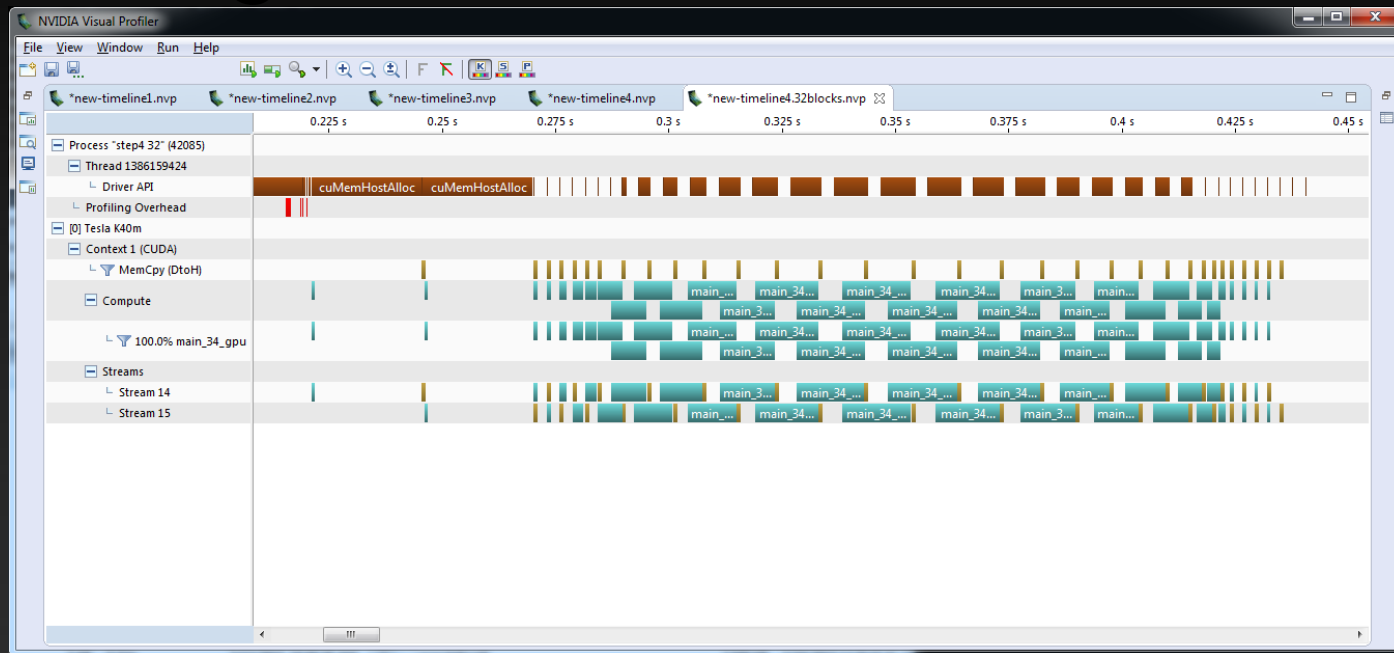# Lots of Data Transfer Time

Step 1 Profile

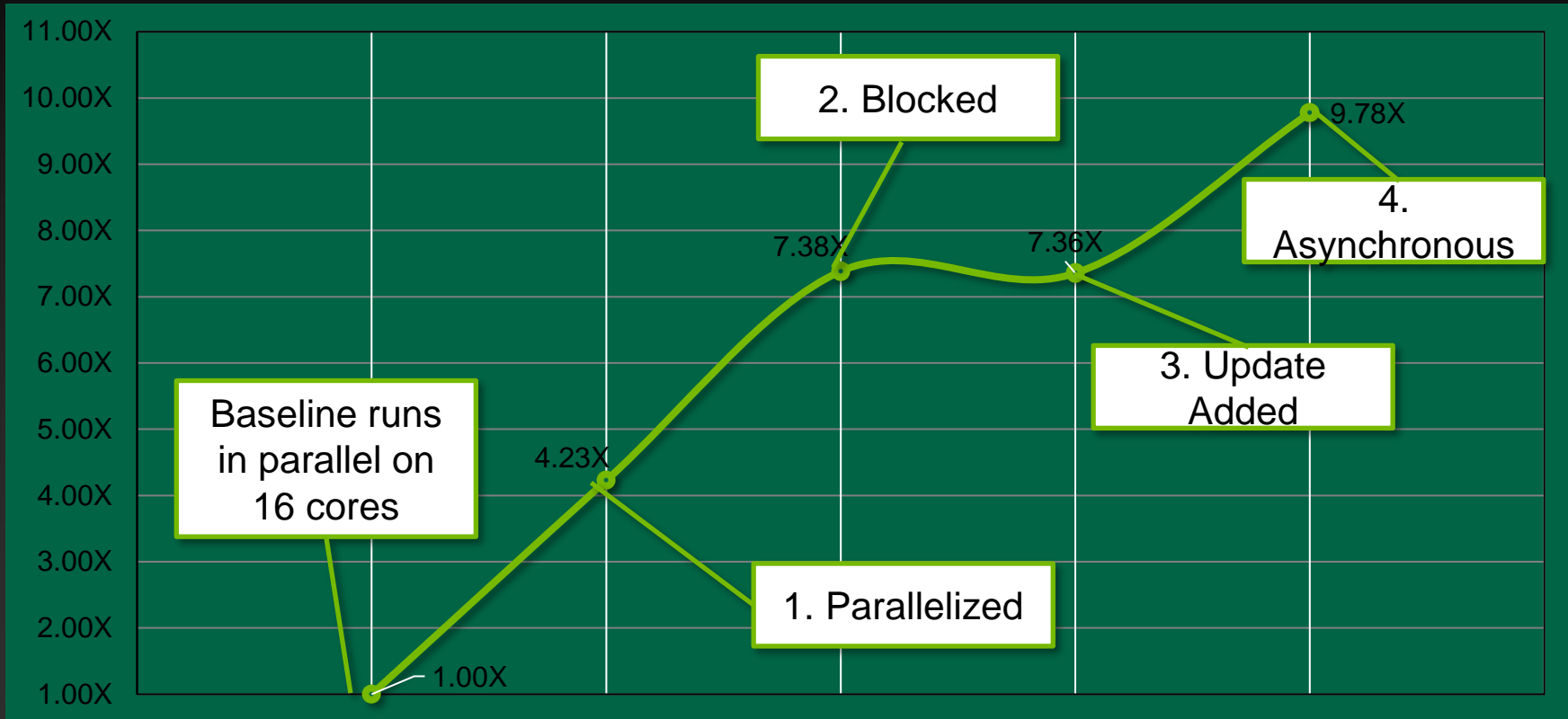Half of our time is copying, none of it is overlapped.

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.

- Language specific features: C dynamic arrays, C++ classes, Fortran derived types. These particular items are well supported. An excellent guide to this topic is the PGI OpenACC Getting Started Guide (http://www.pgroup.com/doc/openacc_gs.pdf).

- 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...

Should I Learn CUDA?

The situation today has a very similar historical precedent. Namely the evolution away from machine languages ("assembly") to C. To use PCs as a particular example.

1980's	1990's
DOS (Machine Language)	Windows, Linux (C)
Games (Machine Language)	Games (C)
Desktop Apps (C, Pascal, Basic)	Desktop Apps (C, C++, VB)

So, the answer is increasingly "probably not". I will guess most of you fall on the "no" side. Just like ML, you aren't really an "expert" unless you do understand what the compiler is doing with your high level approach, but that may not be necessary for your purposes.

A very important principle that remains valid is that any performant approach must allow you to understand what you are ultimately asking the hardware to do at the low level. A programmer that knows assembly knows fairly well what the C statement

$$X = X + 1$$

will become in ML: It will take a some cycles to fetch a value from memory/cache into a register and add a 1 to it. If you know how Python works, then you know that this same instruction might well take many hundreds of cycles, and it may be impossible to tell. If you know C++, this same line might generate exactly the same instructions as C, or it might involve an object and take a thousands of instructions (although you can almost always tell by closer inspection with C++).

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