Advanced OpenACC

Steve Abbott <sabbott@nvidia.com>, November 17, 2017

AGENDA

Expressive Parallelism

Pipelining

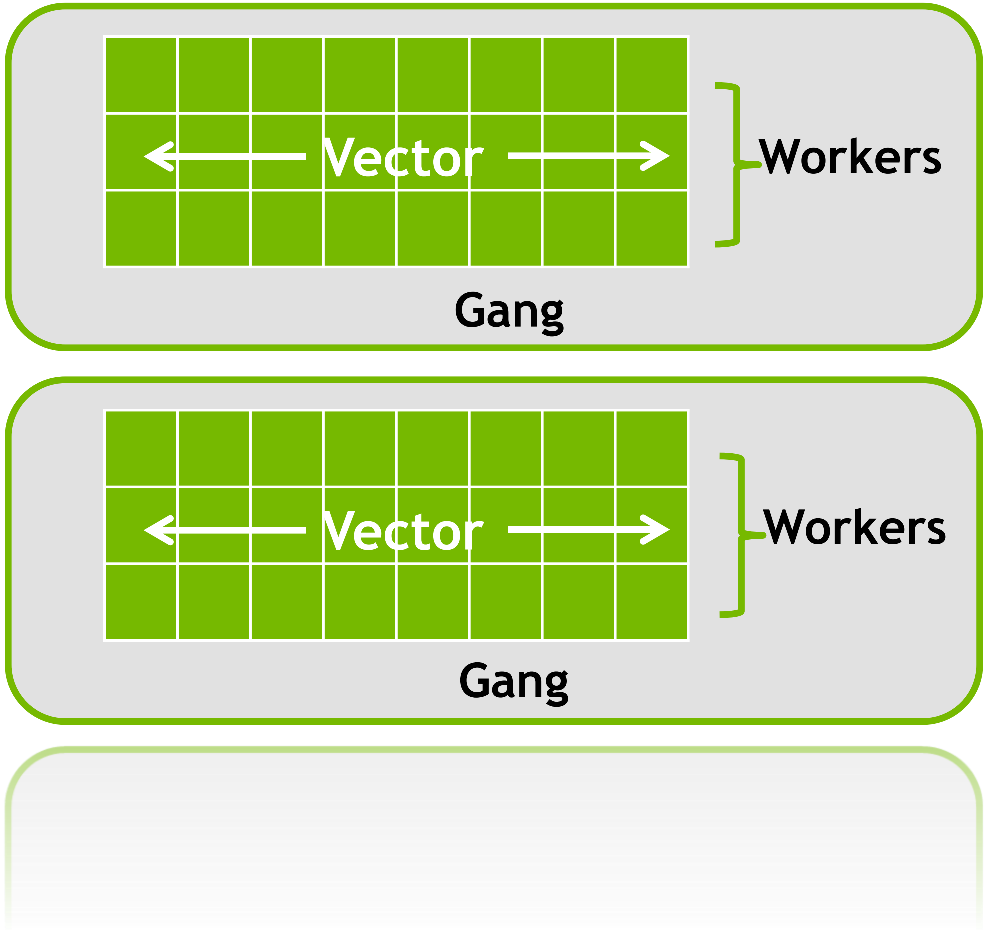
Routines

The loop Directive

The `loop` directive gives the compiler additional information about the *next* loop in the source code through several clauses.

- `independent` – all iterations of the loop are independent
- `collapse(N)` – turn the next N loops into one, flattened loop
- `tile(N[,M,…])` - break the next 1 or more loops into *tiles* based on the provided dimensions.

OpenACC: 3 Levels of Parallelism



- *Vector* threads work in lockstep (SIMD/SIMT parallelism)
- *Workers* compute a vector
- *Gangs* have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other

Execution Model

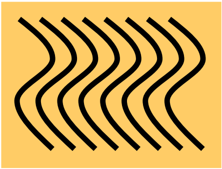
Software

Hardware

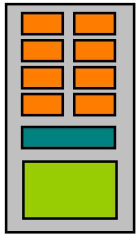
Thread

Scalar Processor

Threads are executed by scalar processors



Thread Block

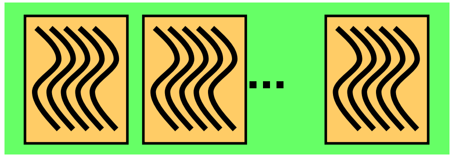


Multiprocessor

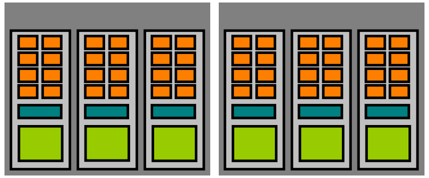
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



Grid



Device

A kernel is launched as a grid of thread blocks

gang, worker, vector Clauses

gang, *worker*, and *vector* can be added to a loop clause

A parallel region can only specify one of each gang, worker, vector

Control the size using the following clauses on the parallel region

num_gangs(n), num_workers(n), vector_length(n)

```
#pragma acc parallel loop gang
 for (int i = 0; i < n; ++i)
 #pragma acc loop vector
  for (int j = 0; j < n; ++j)
   ...
```

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

gang, worker, vector Clauses

gang, *worker*, and *vector* can be added to a kernels loop clause too

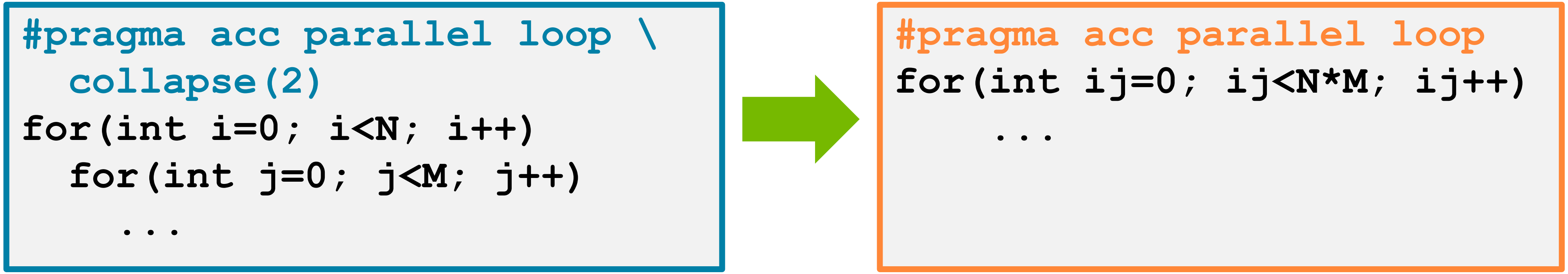
Since different loops in a kernels region may be parallelized differently, fine-tuning is done as a parameter to the gang, worker, and vector clauses.

```
#pragma acc kernels loop gang
 for (int i = 0; i < n; ++i)
  #pragma acc loop vector
   for (int j = 0; j < n; ++j)
    ...
```

```
#pragma acc kernels loop gang worker
for (int i = 0; i < n; ++i)
  #pragma acc loop vector(32)
  for (int j = 0; j < n; ++j)
    ...
```

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++)
    ...
```

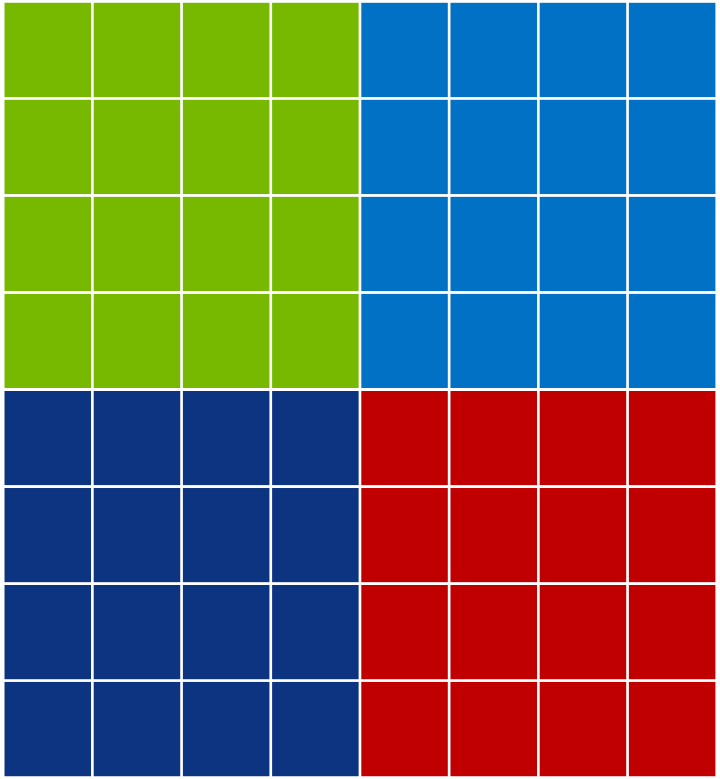
```
#pragma acc parallel loop
for(int ij=0; ij<N*M; ij++)
    ...
```

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.

Optimize Loop Performance

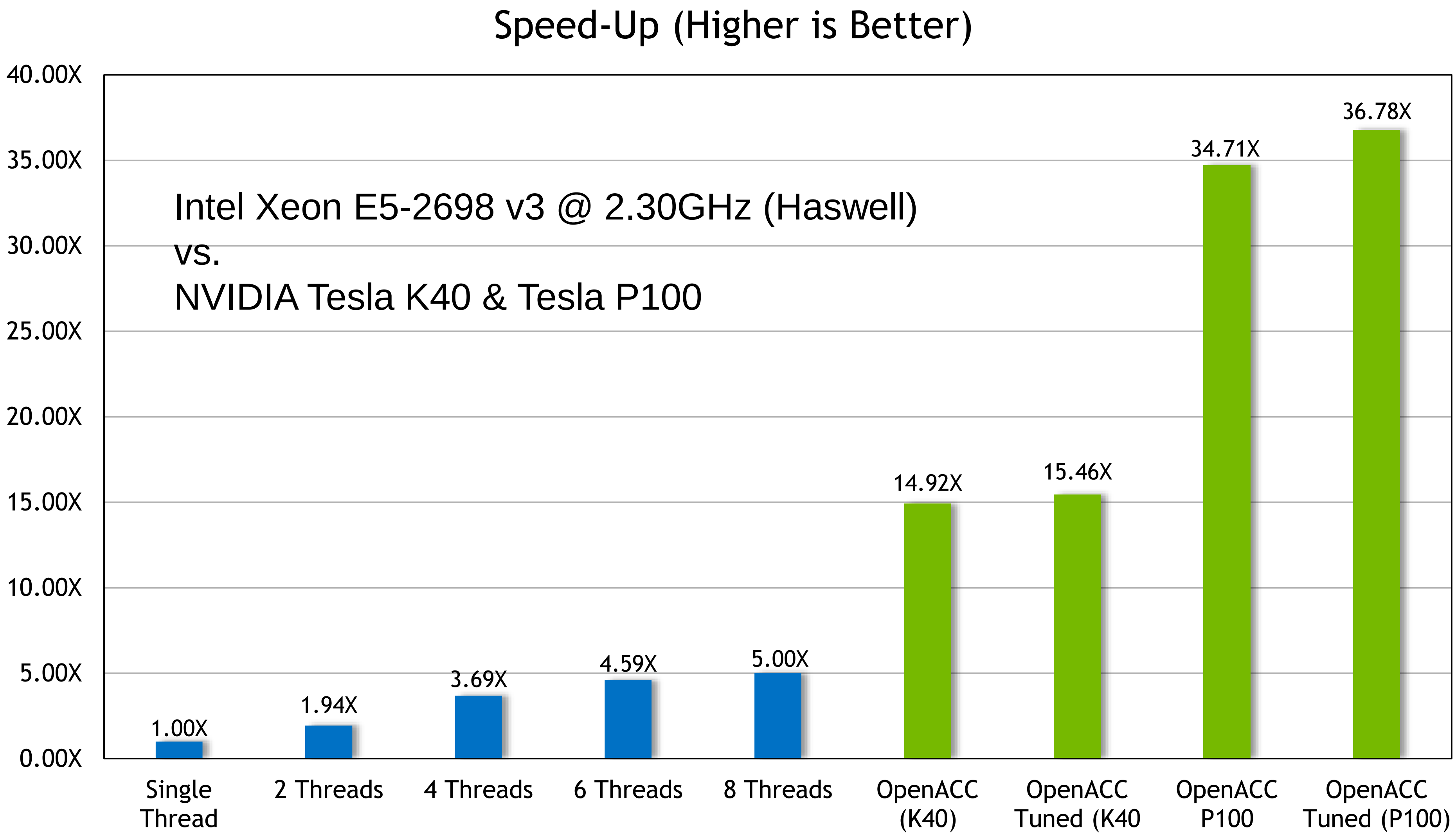
```
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop device_type(nvidia) tile(32,4)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
#pragma acc parallel loop device_type(nvidia) tile(32,4)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
}
  iter++;
}
```

“Tile” the next two loops into 32x4 blocks, but only on NVIDIA GPUs.



Speed-Up (Higher is Better)

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell)
vs.
NVIDIA Tesla K40 & Tesla P100

Compiler: PGI 16.10

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

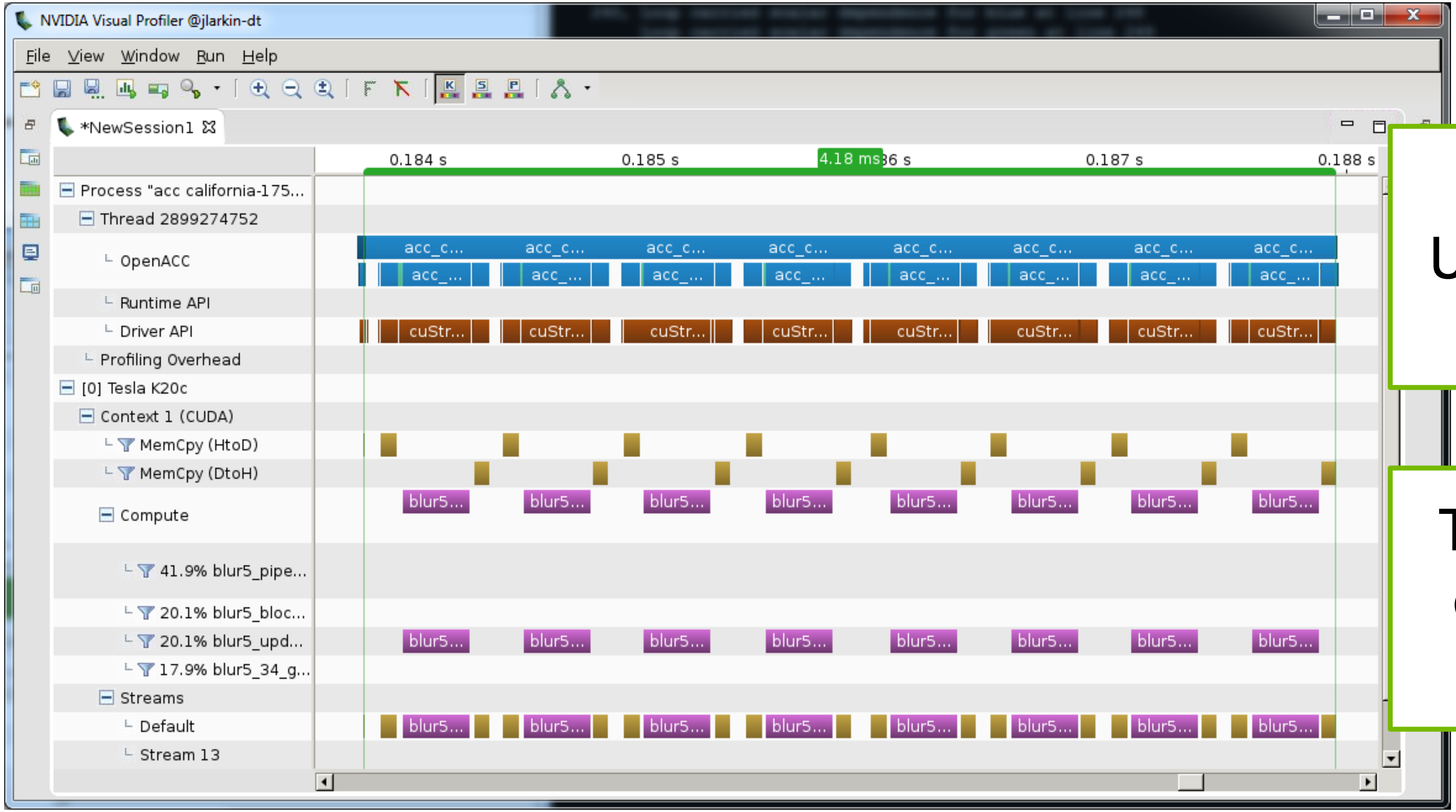
So far, all OpenACC directives have been synchronous with the host

- 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

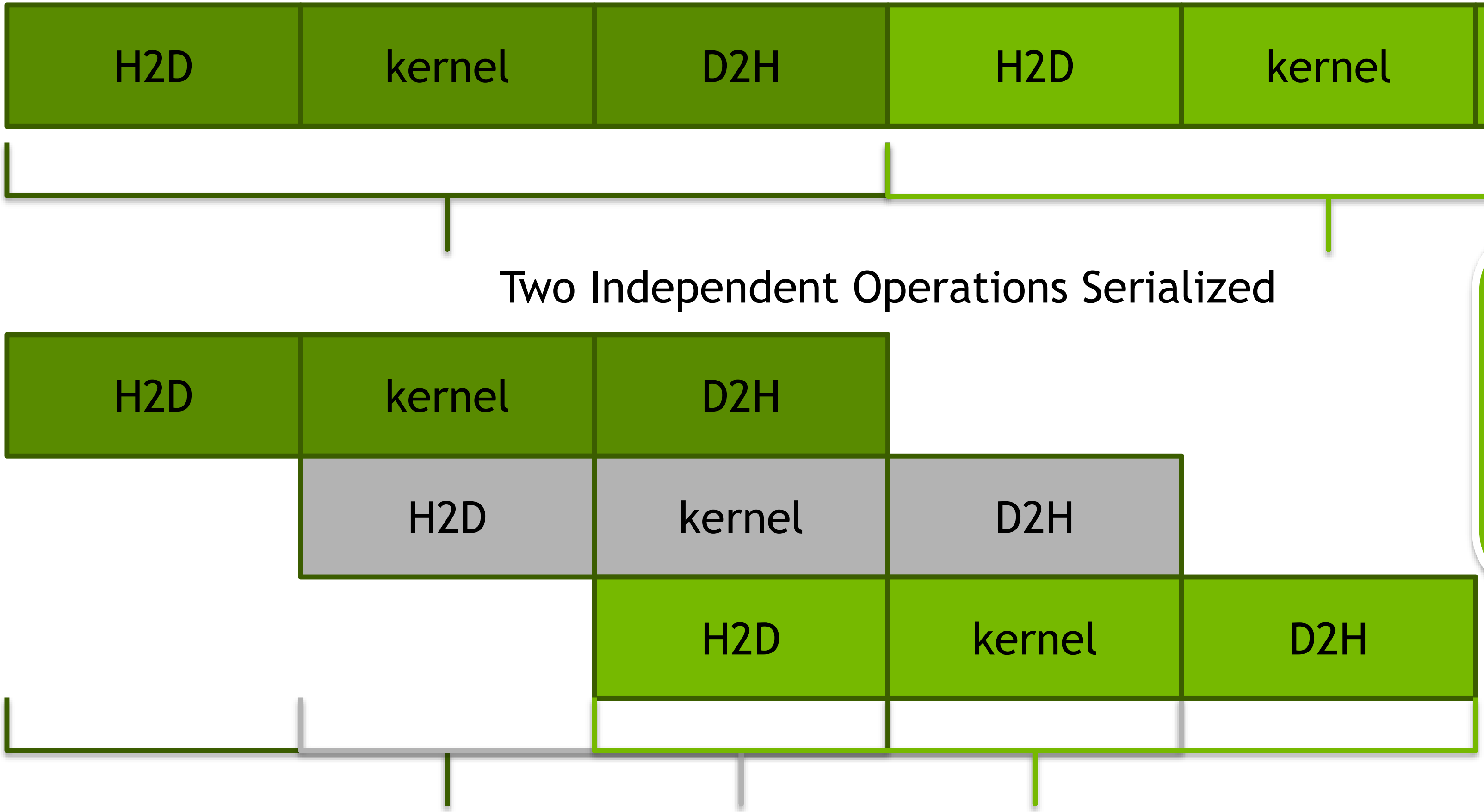
GPU Timeline Blocked Updates



Compute and Updates happen in blocks.

The last step is to overlap compute and copy.

Pipelining Data Transfers



Two Independent Operations Serialized

Overlapping Copying and Computation

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

OpenACC async and wait

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

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

Work queues operate in-order, serving as a way to express dependencies.

Work queues of different numbers *may* (*or may not*) run concurrently.

```
#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.

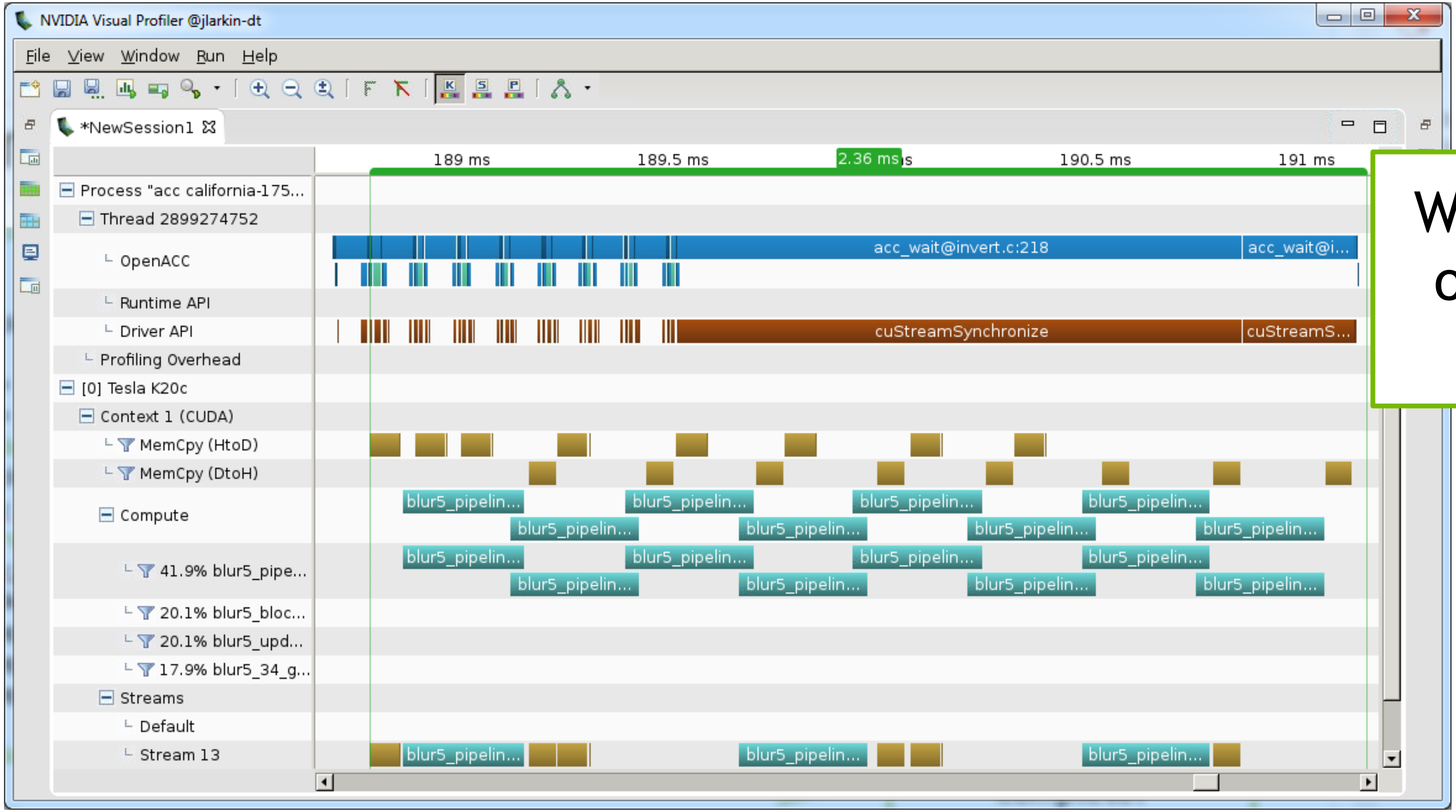
Pipelined Code

```
#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:(endy-starty)*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.

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 Resources

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

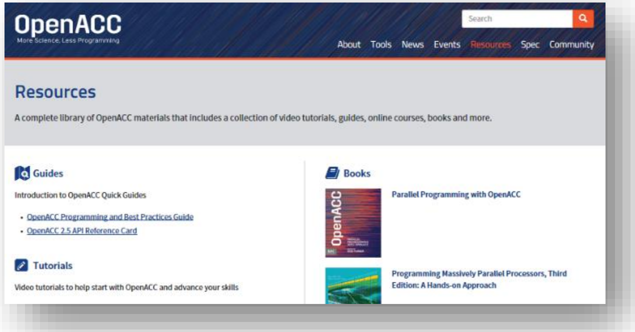
FREE Compilers

slack

https://www.openacc.org/community#slack

Resources

https://www.openacc.org/resources



Resources

A complete library of OpenACC materials that includes a collection of video tutorials, guides, online courses, books and more.

Guides

Introduction to OpenACC Quick Guides

- OpenACC Programming and Best Practices Guide
- OpenACC 2.5 API Reference Card

Tutorials

Video tutorials to help start with OpenACC and advance your skills

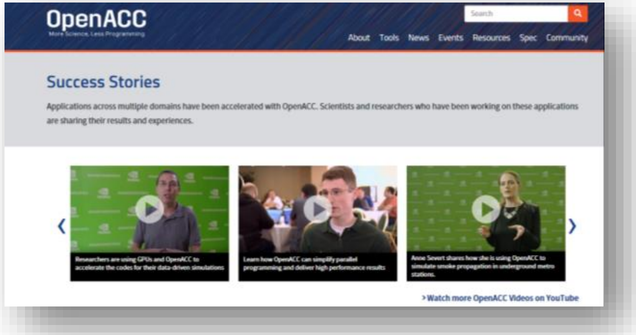
Books

Parallel Programming with OpenACC

Programming Massively Parallel Processors, Third Edition: A Hands-on Approach

Success Stories

https://www.openacc.org/success-stories



Success Stories

Applications across multiple domains have been accelerated with OpenACC. Scientists and researchers who have been working on these applications are sharing their results and experiences.

Researchers are using GPUs and OpenACC to accelerate the codes for their data-driven simulations

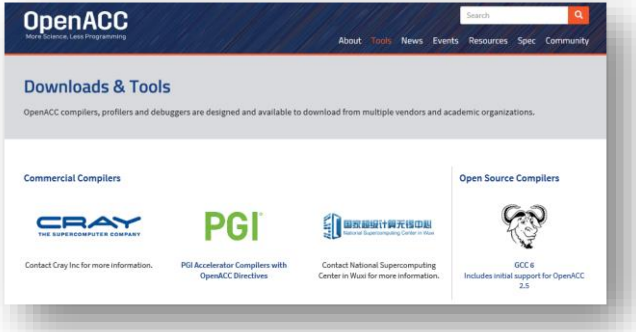
Learn how OpenACC can simplify parallel programming and deliver high performance results

Anne Severt shares how she is using OpenACC to simulate smoke propagation in underground metro stations.

› Watch more OpenACC Videos on YouTube

Compilers and Tools

https://www.openacc.org/tools


OpenACC
More Science, Less Programming
Search
About
Tools
News
Events
Resources
Spec
Community
CRAY
THE SUPERCOMPUTER COMPANY
PGI
国家超级计算无锡中心
National Supercomputing Center in Wuxi

Downloads & Tools

OpenACC compilers, profilers and debuggers are designed and available to download from multiple vendors and academic organizations.

Commercial Compilers

Contact Cray Inc for more information.

PGI Accelerator Compilers with OpenACC Directives

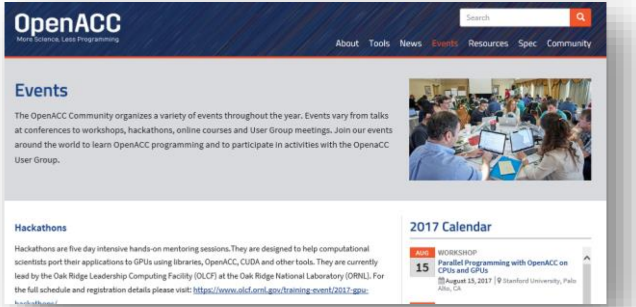
Contact National Supercomputing Center in Wuxi for more information.

Open Source Compilers

GCC 6
Includes initial support for OpenACC 2.5

Events

https://www.openacc.org/events



Events

The OpenACC Community organizes a variety of events throughout the year. Events vary from talks at conferences to workshops, hackathons, online courses and User Group meetings. Join our events around the world to learn OpenACC programming and to participate in activities with the OpenaCC User Group.

Hackathons

Hackathons are five day intensive hands-on mentoring sessions.They are designed to help computational scientists port their applications to GPUs using libraries, OpenACC, CUDA and other tools. They are currently lead by the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory (ORNL). For the full schedule and registration details please visit: https://www.olcf.ornl.gov/training-event/2017-gpu-hackathons/

2017 Calendar

AUG 15 WORKSHOP
Parallel Programming with OpenACC on CPUs and GPUs
August 15, 2017 | Stanford University, Palo Alto, CA