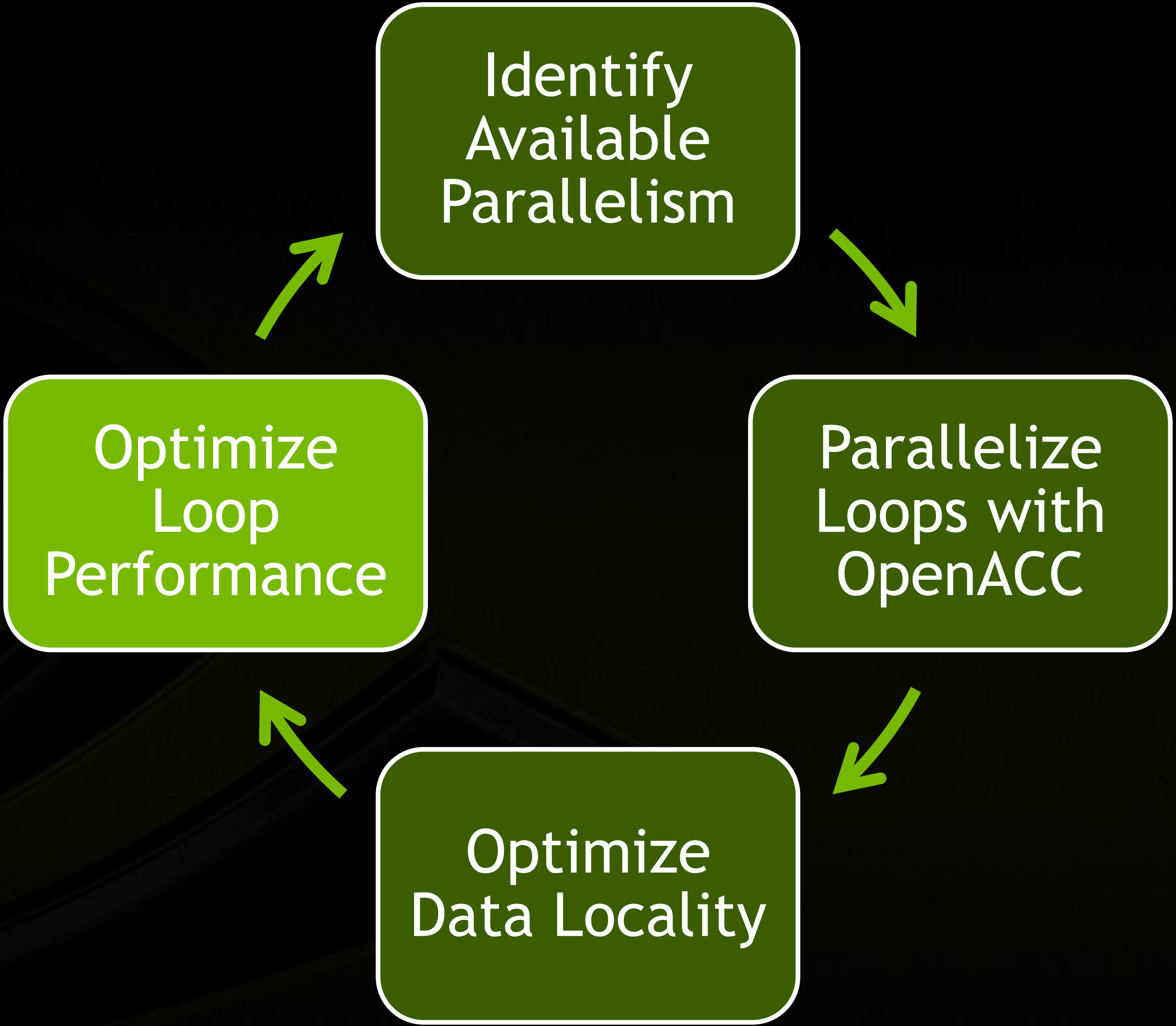
ADVANCED OPENACC PROGRAMMING

DR. CHRISTOPH ANGERER, NVIDIA

*) THANKS TO JEFF LARKIN, NVIDIA, FOR THE SLIDES

AGENDA

- Optimizing OpenACC Loops
- Routines
- Update Directive
- Asynchronous Programming
- Multi-GPU Programming
- OpenACC Interoperability
- Atomic Directive
- Misc. Advice & Techniques
- Q&A



Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance

SPARSE MATRIX/VECTOR PRODUCT

```
do i=1,a%num_rows
   tmpsum = 0.0d0
   row_start = arow_offsets(i)
   row_end   = arow_offsets(i+1)-1
   do j=row_start,row_end
     acol = acols(j)
     acoef = acoefs(j)
     xcoef = x(acol)
     tmpsum = tmpsum + acoef*xcoef
   enddo
   y(i) = tmpsum
 enddo
```

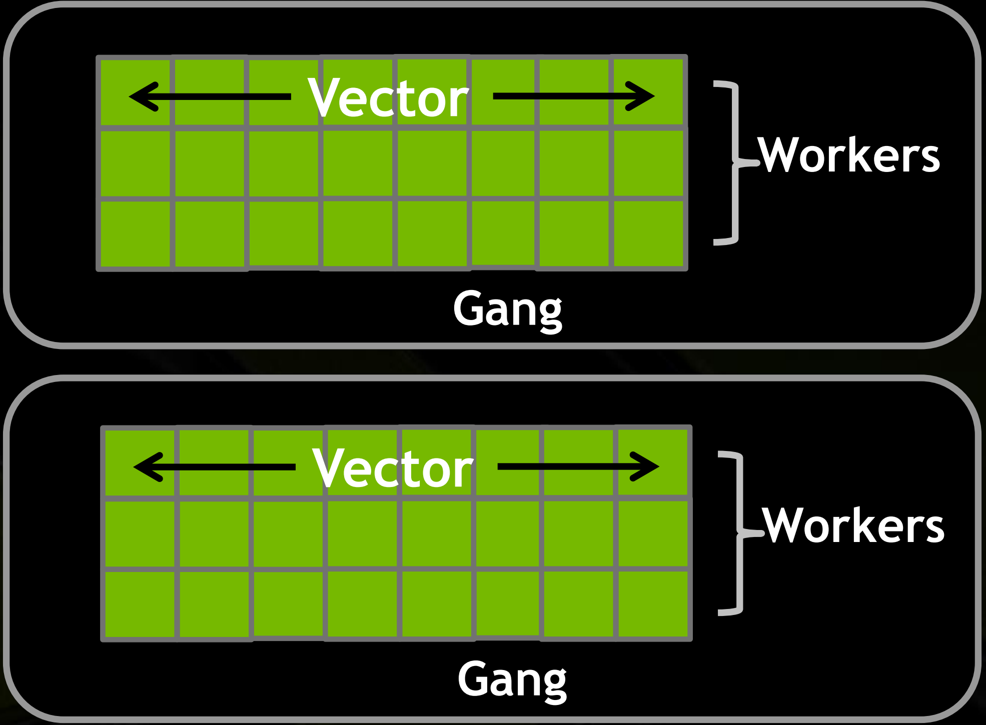
- Performs Mat/Vec product of sparse matrix
- Matrices are stored in a row-compressed format
- Parallelism per-row will vary, but is generally not very large

PARALLELIZED SPMV

```
106     !$acc parallel loop present(arow_offsets,acols,acoefs) &
107     !$acc& private(row_start,row_end,acol,acoef,xcoef) &
108     !$acc& reduction(+:tmpsum)
109     do i=1,a%num_rows
110       tmpsum = 0.0d0
111       row_start = arow_offsets(i)
112       row_end   = arow_offsets(i+1)-1
113       do j=row_start,row_end
114         acol = acols(j)
115         acoef = acoefs(j)
116         xcoef = x(acol)
117         tmpsum = tmpsum + acoef*xcoef
118       enddo
119       y(i) = tmpsum
120     enddo
```

- Data already on device
- Compiler has vectorized the loop at 113 and selected a vector length of 256
- Total application speed-up (including other accelerated routines): 1.08X

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

OPENACC GANG, WORKER, VECTOR CLAUSES

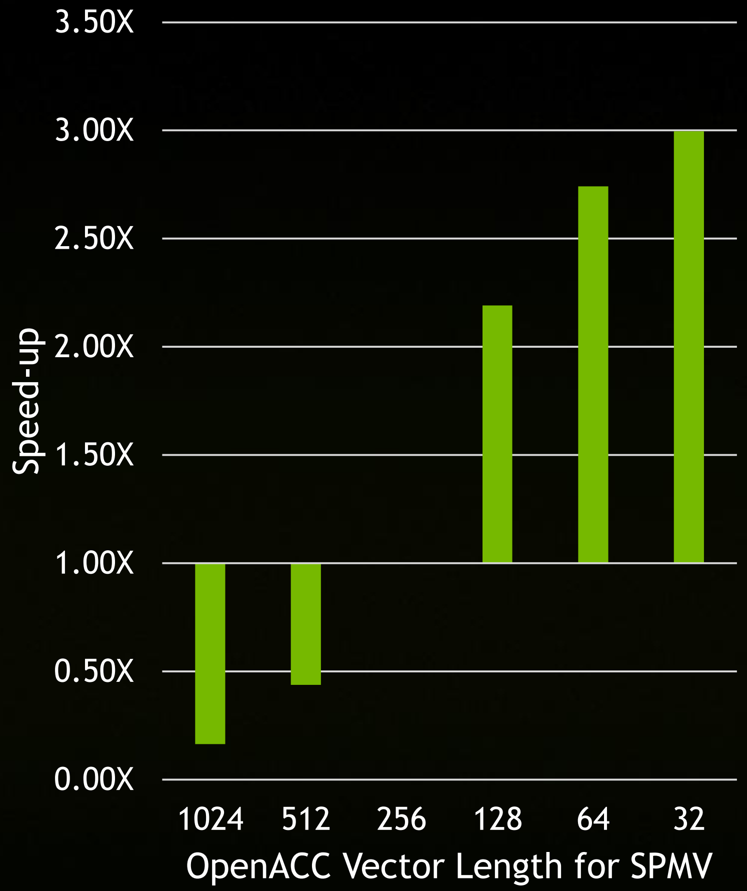
- gang, worker, and vector can be added to a loop clause
- A parallel region can only specify one of each gang, worker, vector
- Control the size using the following clauses on the parallel region
  - num_gangs(n), num_workers(n), vector_length(n)

```
#pragma acc kernels loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector(128)
  for (int j = 0; j < n; ++j)
   ...
```

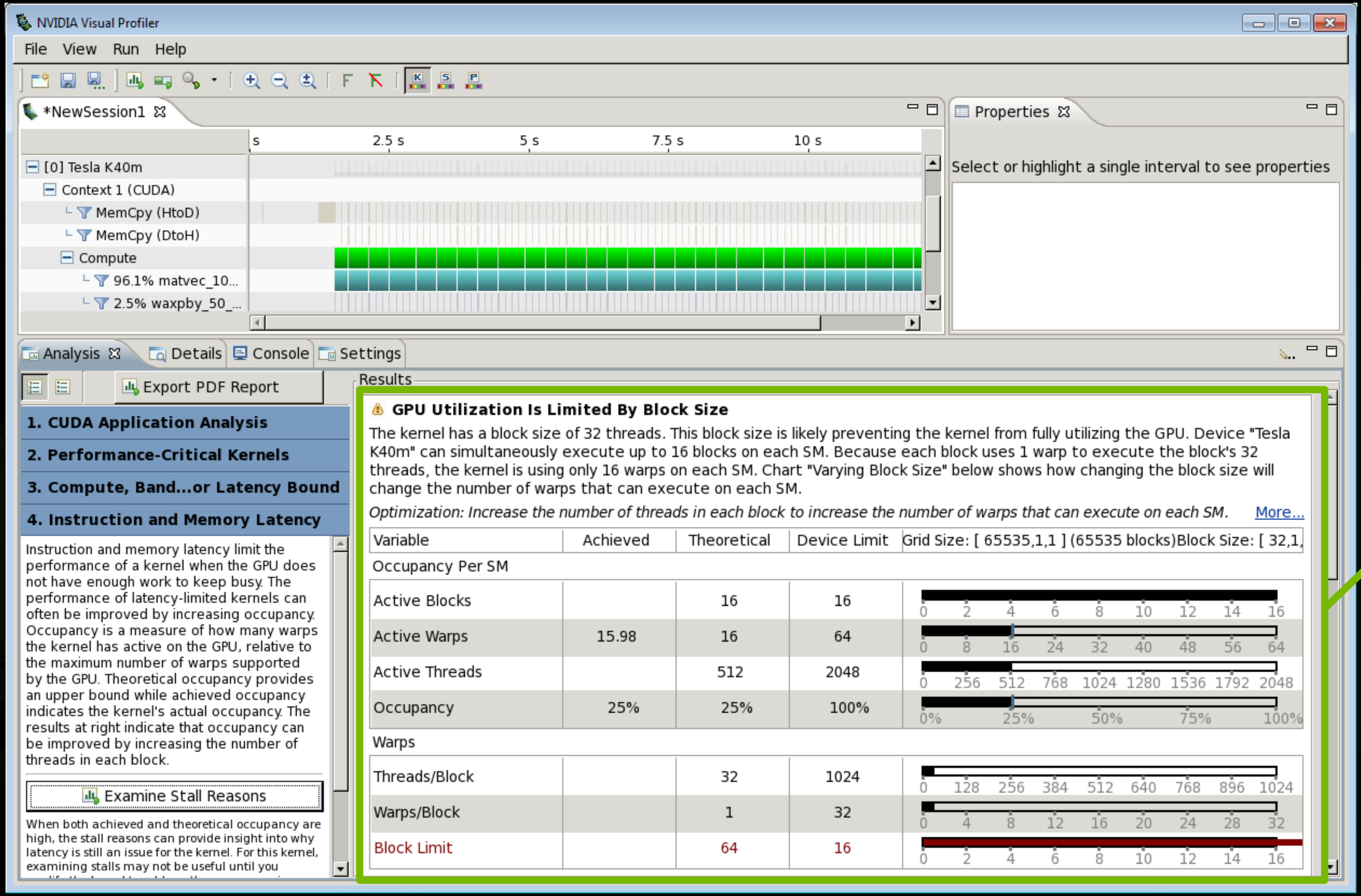
```
#pragma acc parallel vector_length(128)
#pragma acc loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
    ...
```

OPTIMIZED SPMV VECTOR LENGTH

```
!$acc parallel loop present(arow_offsets,acols,acoefs) &
!$acc& private(row_start,row_end,acol,acoef,xcoef) &
!$acc& vector_length(32)
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop vector reduction(+:tmpsum)
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
```



PERFORMANCE LIMITER: OCCUPANCY



Select or highlight a single interval to see properties

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy. The results at right indicate that occupancy can be improved by increasing the number of threads in each block.

When both achieved and theoretical occupancy are high, the stall reasons can provide insight into why latency is still an issue for the kernel. For this kernel, examining stalls may not be useful until you

GPU Utilization Is Limited By Block Size

The kernel has a block size of 32 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "Tesla K40m" can simultaneously execute up to 16 blocks on each SM. Because each block uses 1 warp to execute the block's 32 threads, the kernel is using only 16 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

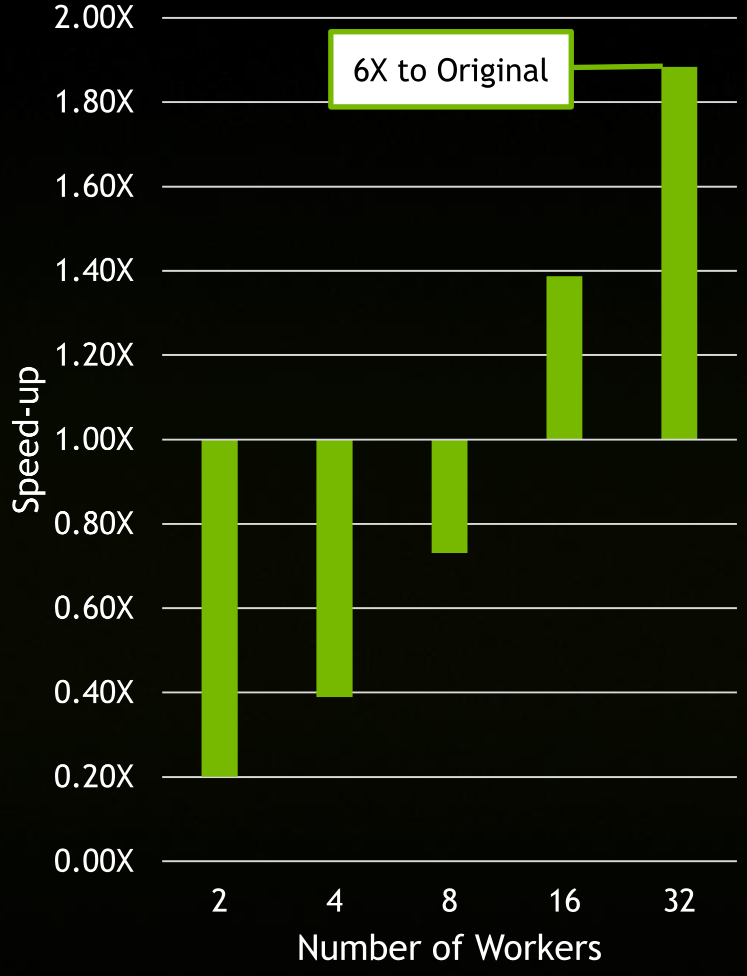
Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [32,1,
Occupancy Per SM				
Active Blocks		16	16	
Active Warps	15.98	16	64	
Active Threads		512	2048	
Occupancy	25%	25%	100%	
Warps				
Threads/Block		32	1024	
Warps/Block		1	32	
Block Limit		64	16	

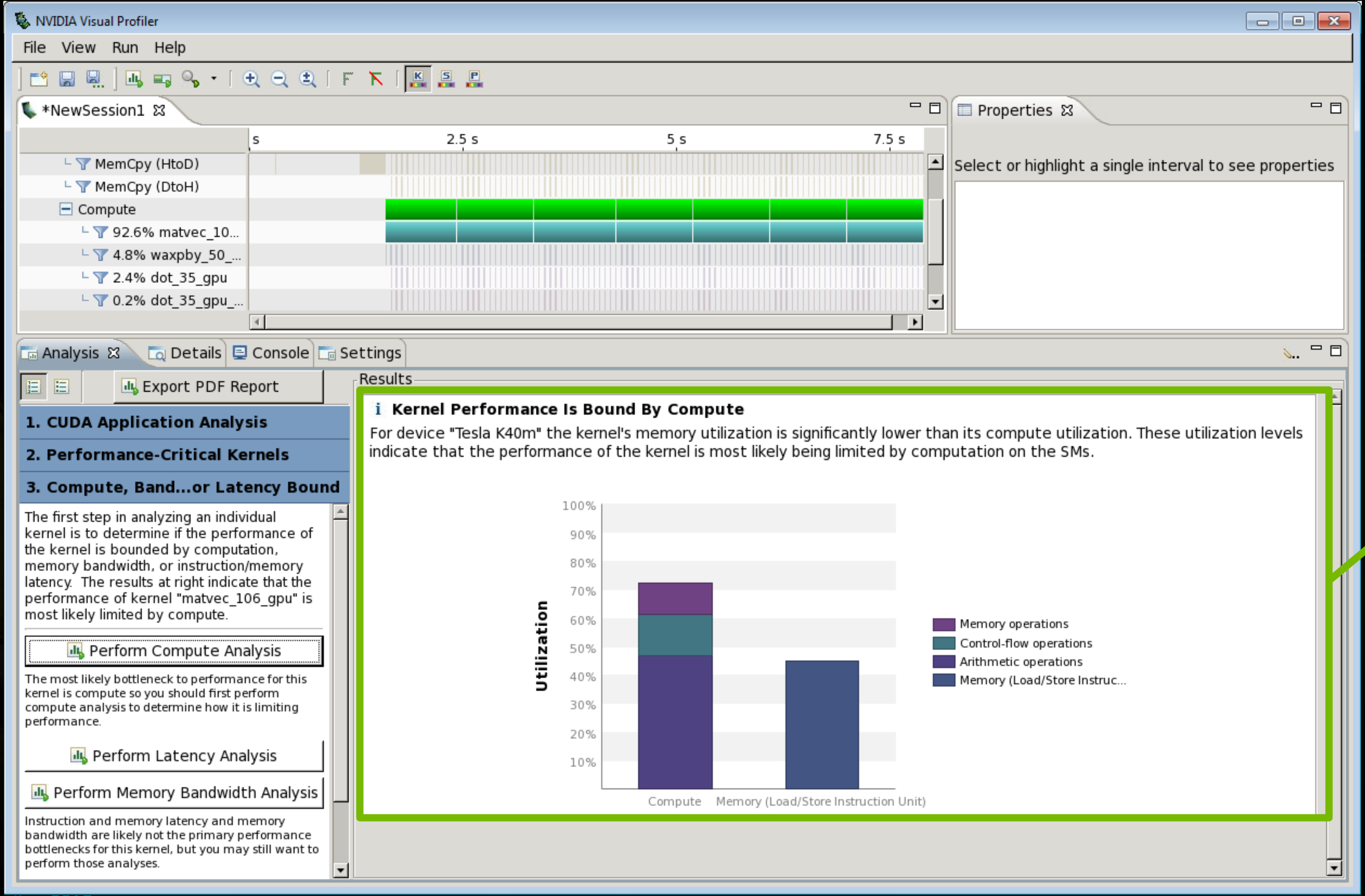
We need more threads!

INCREASED PARALLELISM WITH WORKERS

```
!$acc parallel loop present(arow_offsets,acols,acoefs) &
!$acc& private(row_start,row_end,acol,acoef,xcoef) &
!$acc& gang worker vector_length(32) num_workers(32)
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop vector reduction(+:tmpsum)
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
```



PERFORMANCE LIMITER: COMPUTE



Select or highlight a single interval to see properties

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Band...or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "matvec_106_gpu" is most likely limited by compute.

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

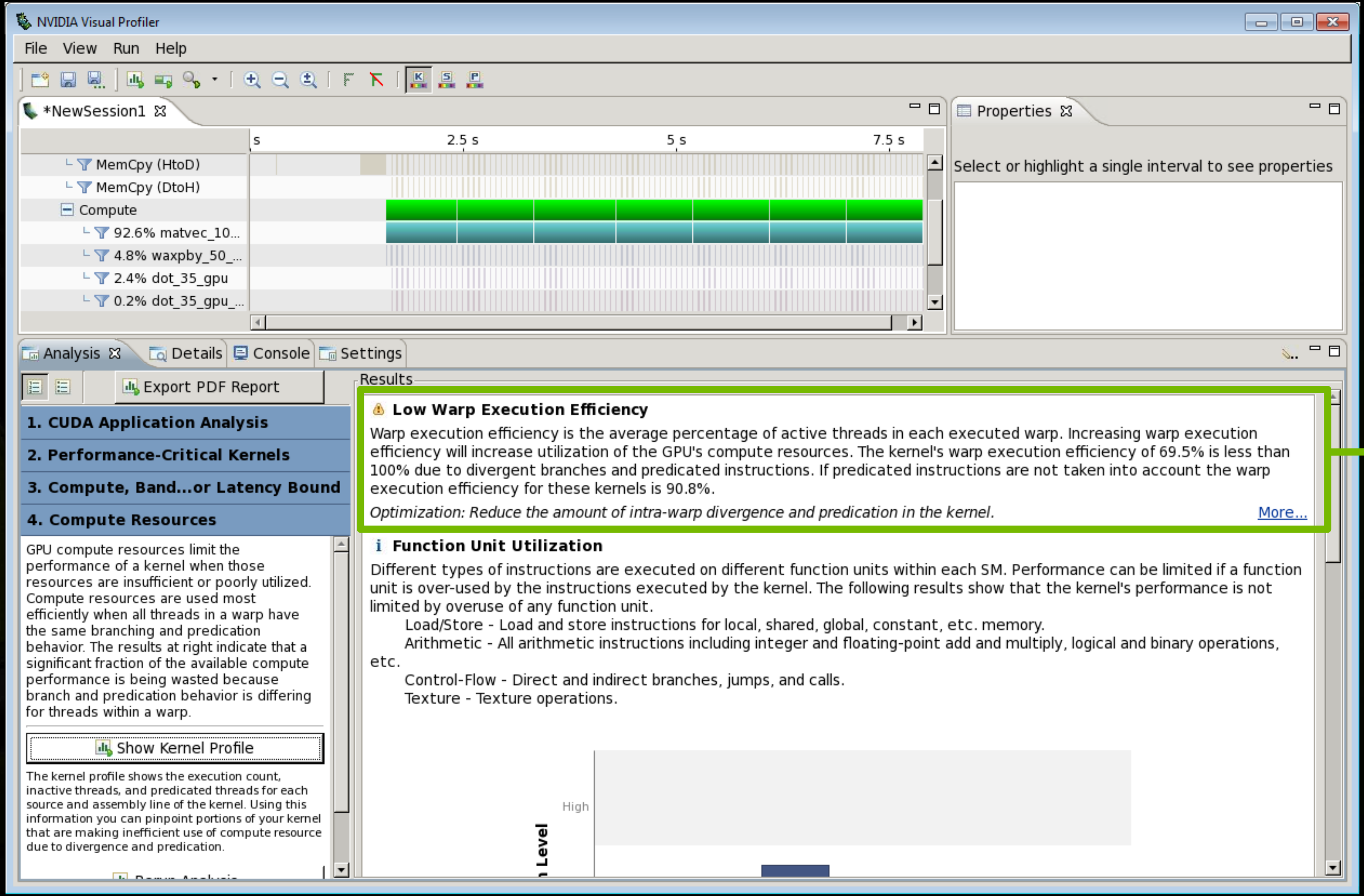
Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

i **Kernel Performance Is Bound By Compute**

For device "Tesla K40m" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

Now we’re compute bound

PERFORMANCE LIMITER: PARALLELISM



Select or highlight a single interval to see properties

Low Warp Execution Efficiency

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The kernel's warp execution efficiency of 69.5% is less than 100% due to divergent branches and predicated instructions. If predicated instructions are not taken into account the warp execution efficiency for these kernels is 90.8%.

Optimization: Reduce the amount of intra-warp divergence and predication in the kernel. More...

Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for local, shared, global, constant, etc. memory.

Arithmetic - All arithmetic instructions including integer and floating-point add and multiply, logical and binary operations, etc.

Control-Flow - Direct and indirect branches, jumps, and calls.

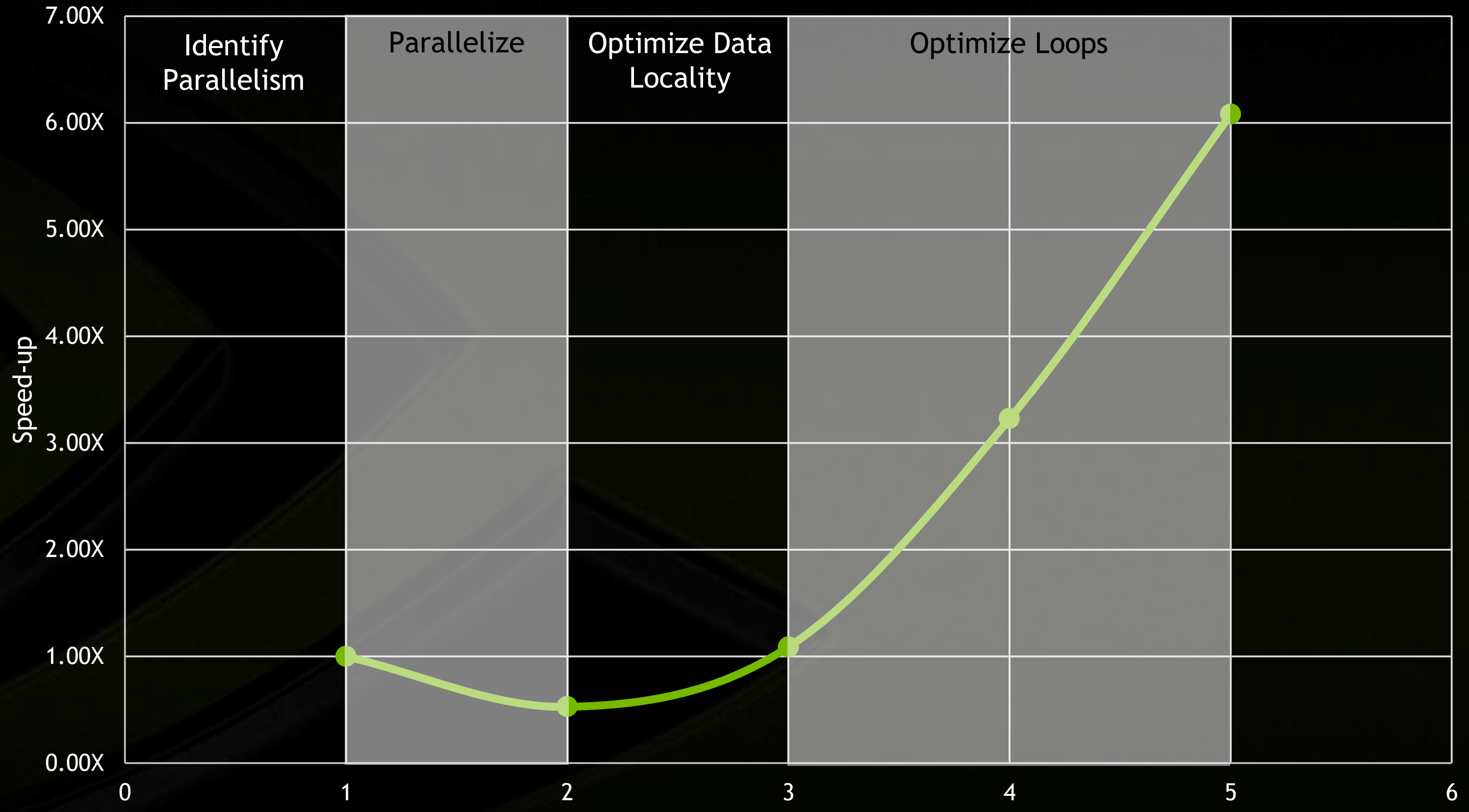
Texture - Texture operations.

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized. Compute resources are used most efficiently when all threads in a warp have the same branching and predication behavior. The results at right indicate that a significant fraction of the available compute performance is being wasted because branch and predication behavior is differing for threads within a warp.

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Really, we're limited by parallelism per-row.

SPEED-UP STEP BY STEP



OPENACC COLLAPSE CLAUSE

collapse(n): Transform the following *n* tightly nested loops into one, flattened loop.

- Useful when individual loops lack sufficient parallelism or more than 3 loops are nested (gang/worker/vector)

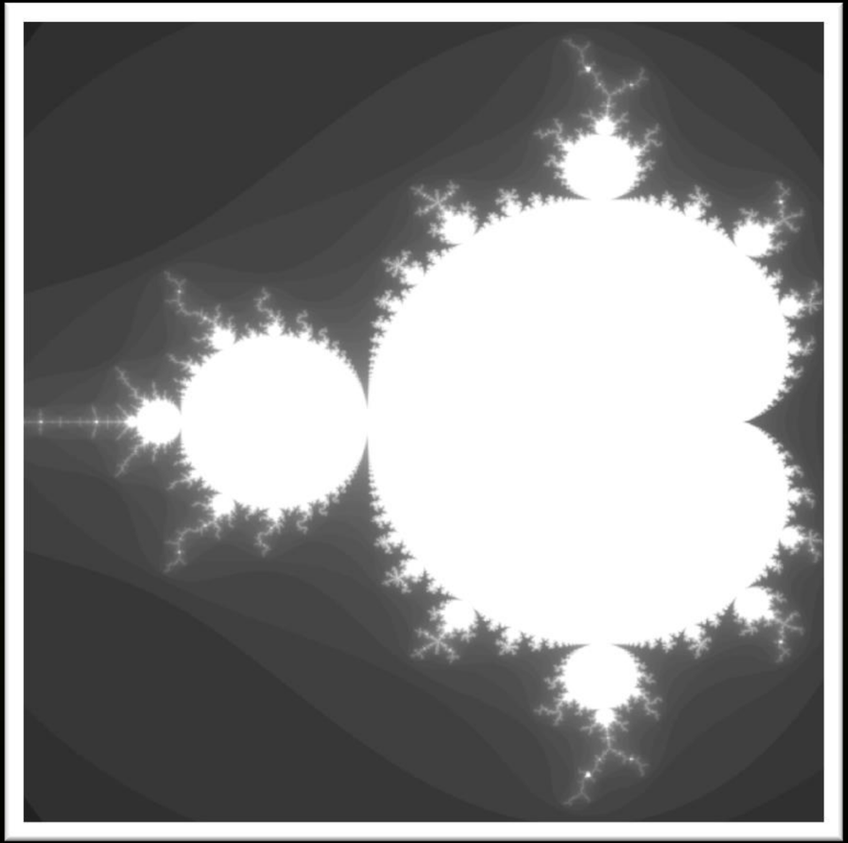
```
#pragma acc parallel
#pragma acc loop collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
    ...
```

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

⚠ Loops must be tightly nested

NEW CASE STUDY: MANDELBROT SET

- Application generates the image to the right.
- Each pixel in the image can be independently calculated.
- Skills Used:
  - Parallel Loop
  - Data Region
  - Update Directive
  - Asynchronous Pipelining



MANDELBROT CODE

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

The mandelbrot() function calculates the color for each pixel.

Within main() there is a doubly-nested loop that calculates each pixel independently.

ROUTINES

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.

MANDELBROT: ROUTINE DIRECTIVE

```
// mandelbrot.h
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py);

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

- 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

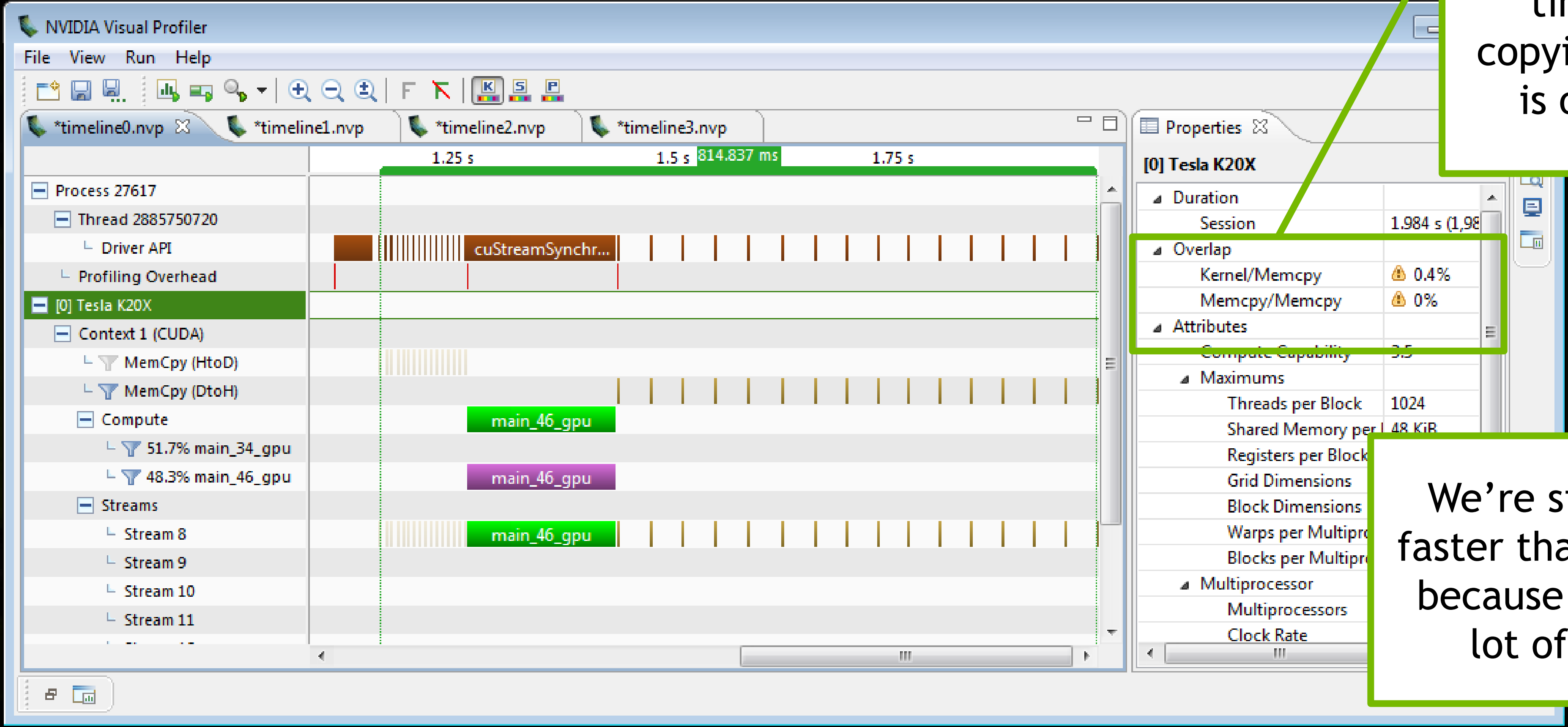
```fortran
module mandelbrot_mod
  implicit none
  integer, parameter :: HEIGHT=16384
  integer, parameter :: WIDTH=16384
  integer, parameter :: MAXCOLORS = 255

contains
 real(8) function mandlebrot(px,py)
    implicit none
    !$acc routine(mandlebrot) seq
    ...
 end function mandlebrot

end module mandelbrot_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.

BASELINE PROFILE



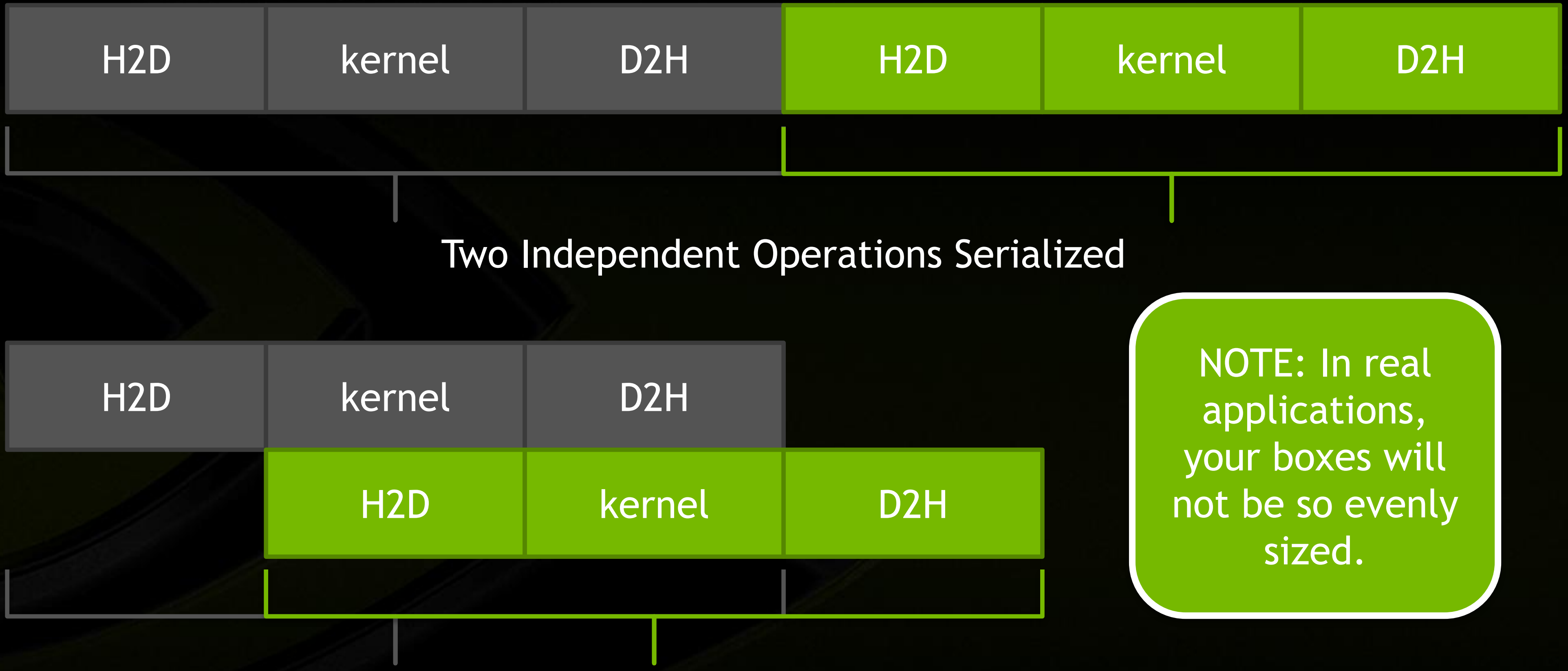
[0] Tesla K20X

Duration	
Session	1.984 s (1,98
Overlap	
Kernel/Memcpy	⚠ 0.4%
Memcpy/Memcpy	⚠ 0%
Attributes	
Compute Capability	3.5
Maximums	
Threads per Block	1024
Shared Memory per	48 KiB
Registers per Block	
Grid Dimensions	
Block Dimensions	
Warps per Multipro	
Blocks per Multipro	
Multiprocessor	
Multiprocessors	
Clock Rate	

Roughly 25% of our time is spent copying, none of it is overlapped.

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

PIPELINING DATA TRANSFERS



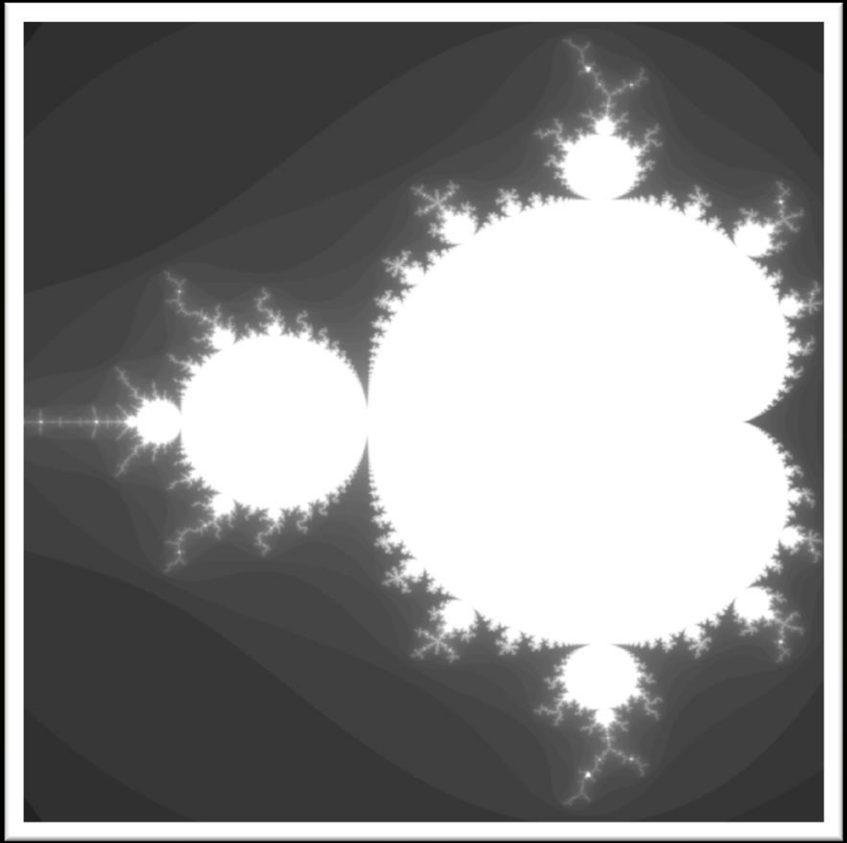
Two Independent Operations Serialized

Overlapping Copying and Computation

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

PIPELINING MANDELBROT SET

- We only have 1 kernel, so there’s nothing to overlap.
- Since each pixel is independent, computation can be broken up
- Steps
  1. Break up computation into blocks along rows.
  2. Break up copies according to blocks
  3. Make both computation and copies asynchronous



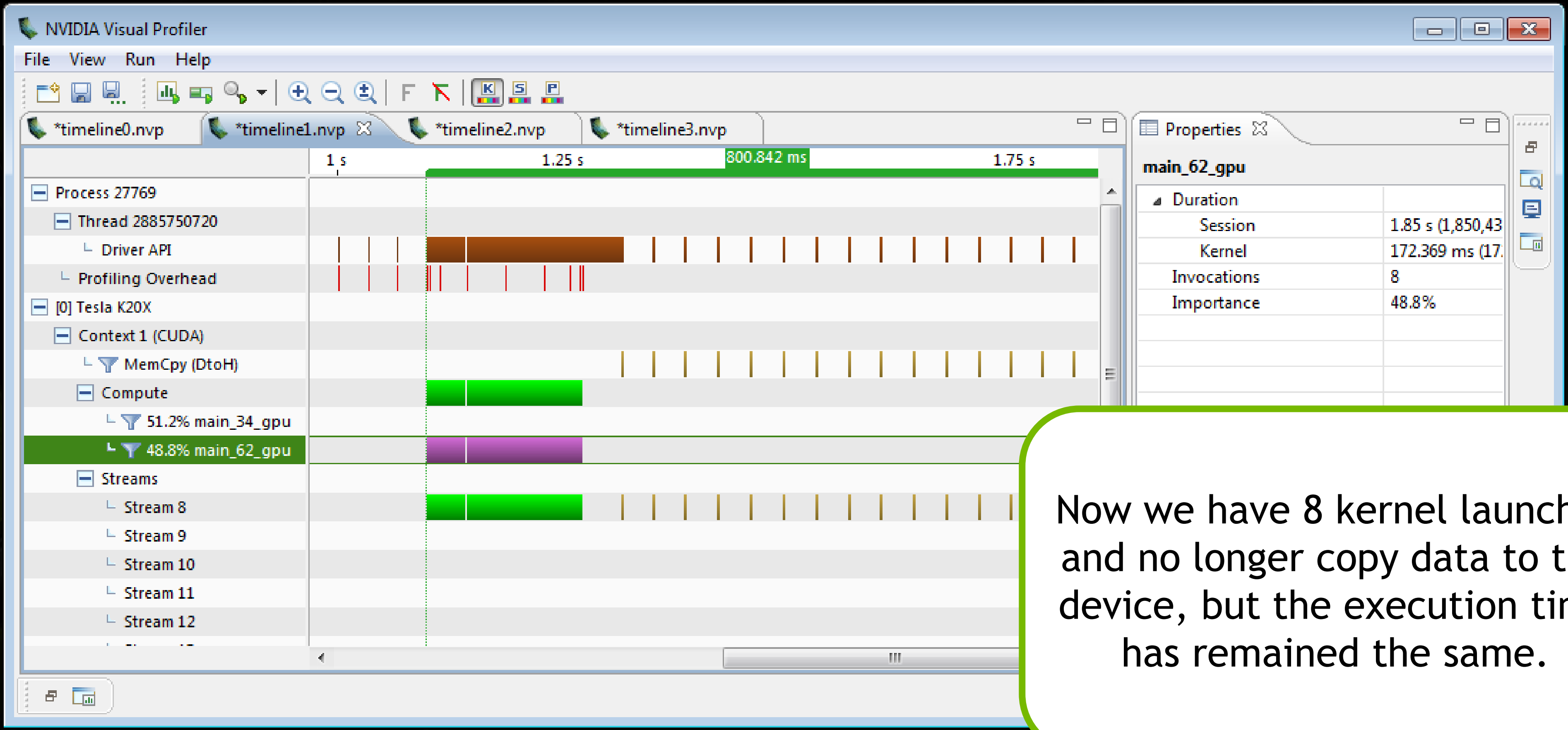
STEP 1: BLOCKING COMPUTATION

```
  numblocks = ( argc > 1 ) ? atoi(argv[1]) : 8;
  blocksize = HEIGHT / numblocks;
  printf("numblocks: %d, blocksize: %d\n",
numblocks, blocksize);

#pragma acc data copyout(image[:bytes])
  for(int block=0; block < numblocks; block++)
  {
    int ystart = block * blocksize;
    int yend   = ystart + blocksize;
#pragma acc parallel loop
    for(int y=ystart;y<yend;y++) {
      for(int x=0;x<WIDTH;x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
      }
    }
  }
```

- Add a loop over blocks
- Modify the existing row loop to only work within blocks
- Add data region around blocking loop to leave data local to the device.
- Check for correct results.

- NOTE: We don’t need to copy in the array, so make it an explicit copyout.

BLOCKING TIMELINE



main_62_gpu

Duration	
Session	1.85 s (1,850,43
Kernel	172.369 ms (17.
Invocations	8
Importance	48.8%

Now we have 8 kernel launches and no longer copy data to the device, but the execution time has remained the same.

UPDATE DIRECTIVE

OPENACC DATA REGIONS REVIEW

```
#pragma acc data copyout(image[:bytes])
  for(int block=0; block < numblocks; block++)
  {
    int ystart = block * blocksize;
    int yend   = ystart + blocksize;
#pragma acc parallel loop
    for(int y=ystart;y<yend;y++) {
      for(int x=0;x<WIDTH;x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
      }
    }
  }
```

Data is shared within this region.

OPENACC UPDATE DIRECTIVE

Programmer specifies an array (or part of an array) that should be refreshed within a data region.

```
do_something_on_device()

!$acc update self(a)

do_something_on_host()

!$acc update device(a)
```

Copy “a” from GPU to CPU

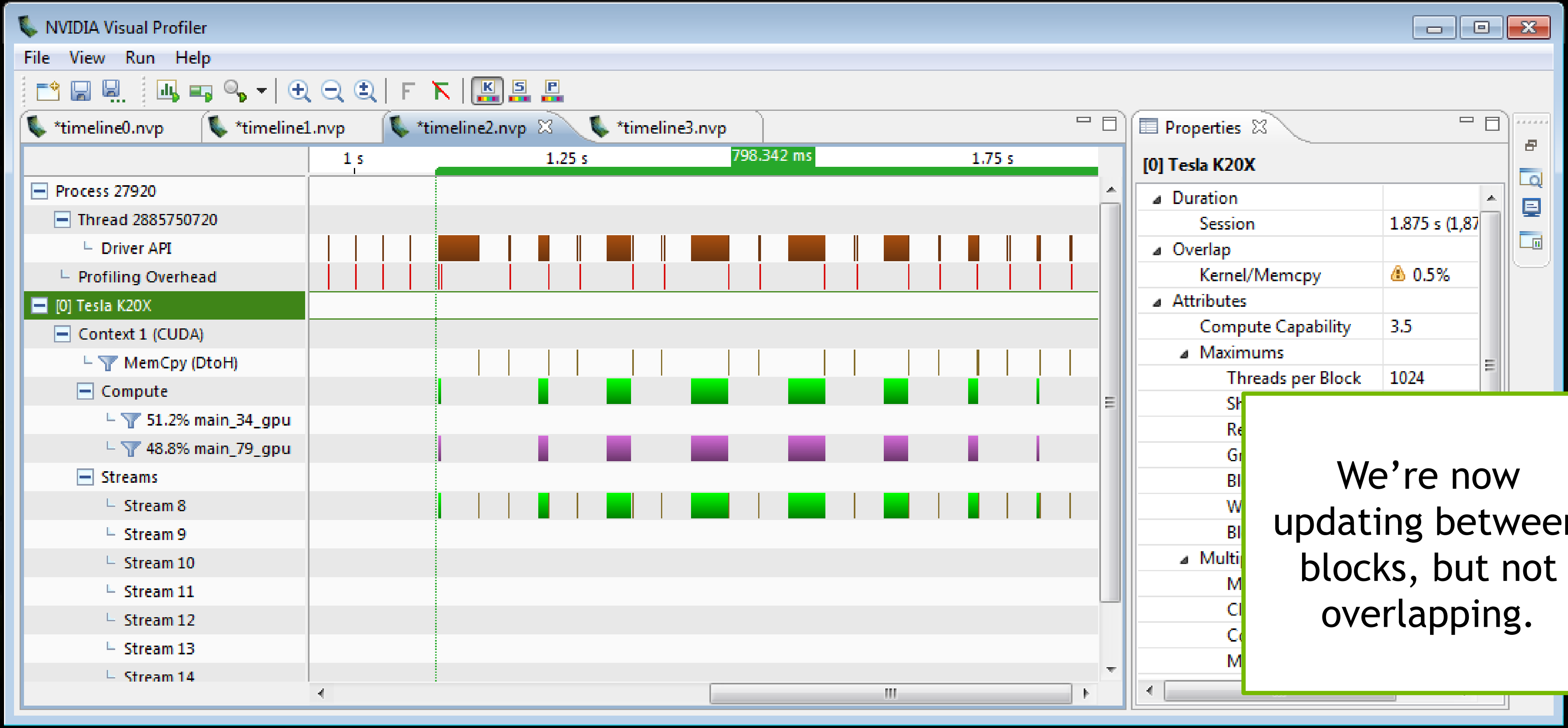
Copy “a” from CPU to GPU

STEP 2: COPY BY BLOCK

```
#pragma acc data create(image[:bytes])
  for(int block=0; block < numblocks; block++)
  {
    int ystart = block * blocksize;
    int yend   = ystart + blocksize;
#pragma acc parallel loop
    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])
  }
```

- Change the data region to only create the array on the GPU
- Use an update directive to copy individual blocks back to the host when complete
- Check for correct results.

TIMELINE: UPDATING BY BLOCKS



[0] Tesla K20X

Duration	
Session	1.875 s (1,87
Overlap	
Kernel/Memcpy	⚠ 0.5%
Attributes	
Compute Capability	3.5
Maximums	
Threads per Block	1024
Sh	
Re	
Gr	
Bl	
W	
Bl	
Multi	
M	
Cl	
Co	
M	

We’re now updating between blocks, but not overlapping.

ASYNCHRONOUS PROGRAMMING

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, 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++)
```

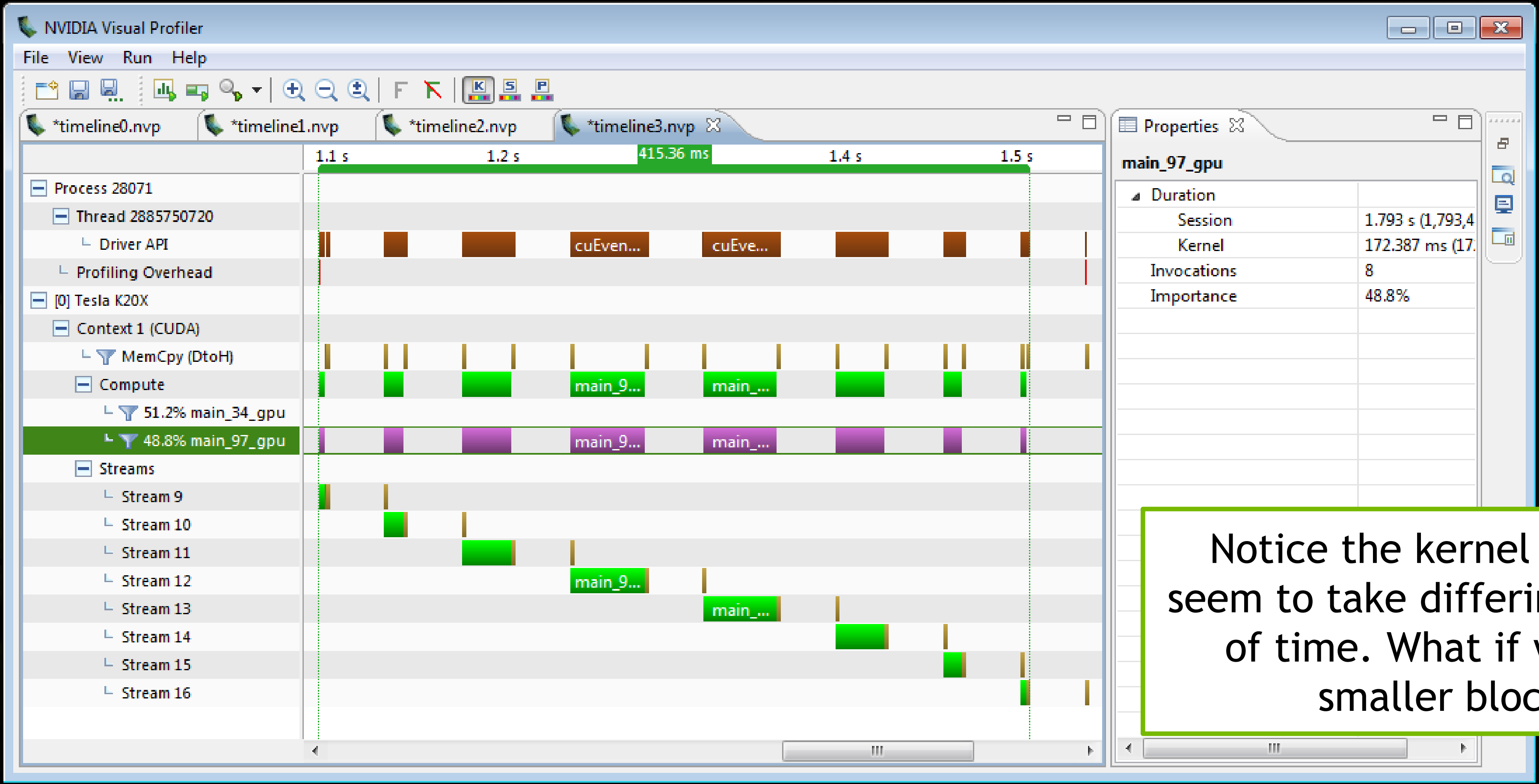
If *n* is not specified, *async* will go into a default queue and *wait* will wait all previously queued work.

STEP 3: GO ASYNCHRONOUS

```
#pragma acc data create(image[:bytes])
  for(int block=0; block < numblocks; block++)
  {
    int ystart = block * blocksize;
    int yend   = ystart + blocksize;
#pragma acc parallel loop async(block)
    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(block)
  }
#pragma acc wait
```

- Make each parallel region asynchronous by placing in different queues.
- Make each update asynchronous by placing in same stream as the parallel region on which it depends
- Synchronize for all to complete.
- Check for correct results.

TIMELINE: PIPELINING

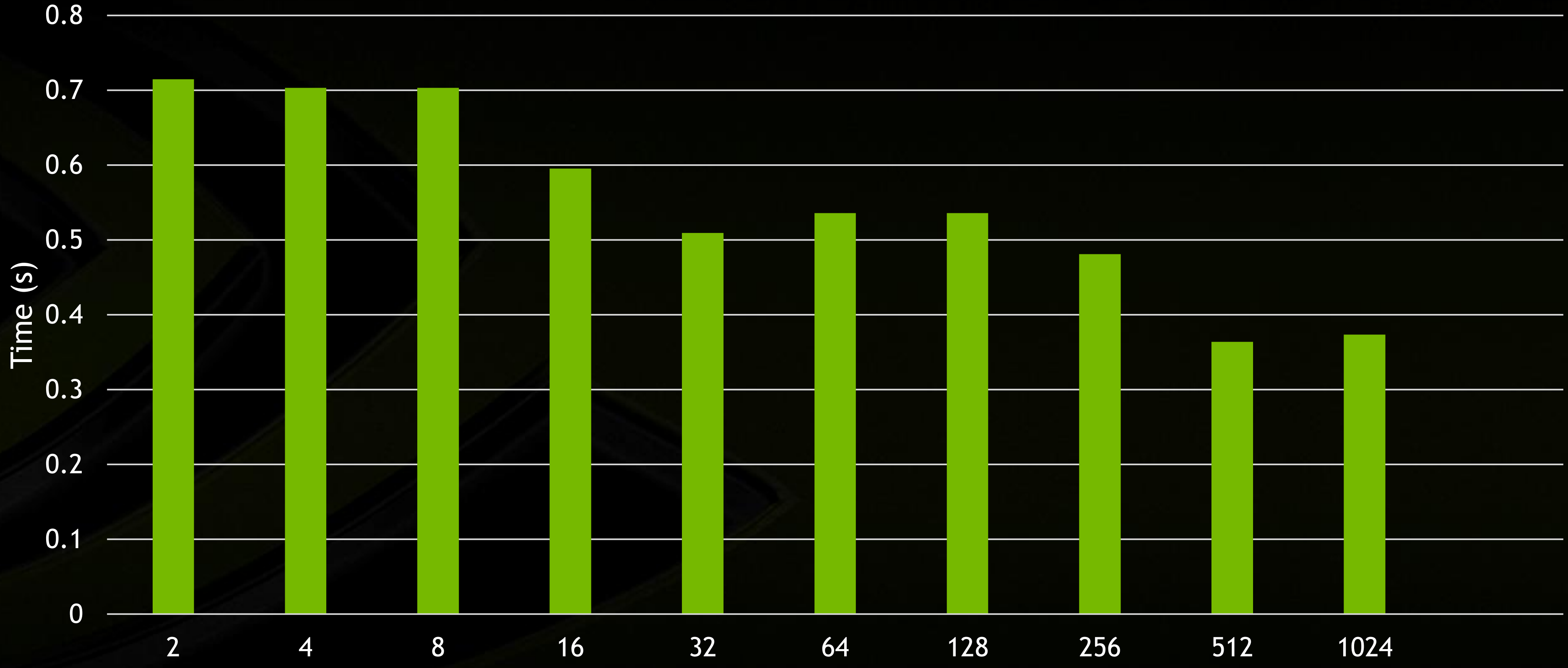


main_97_gpu

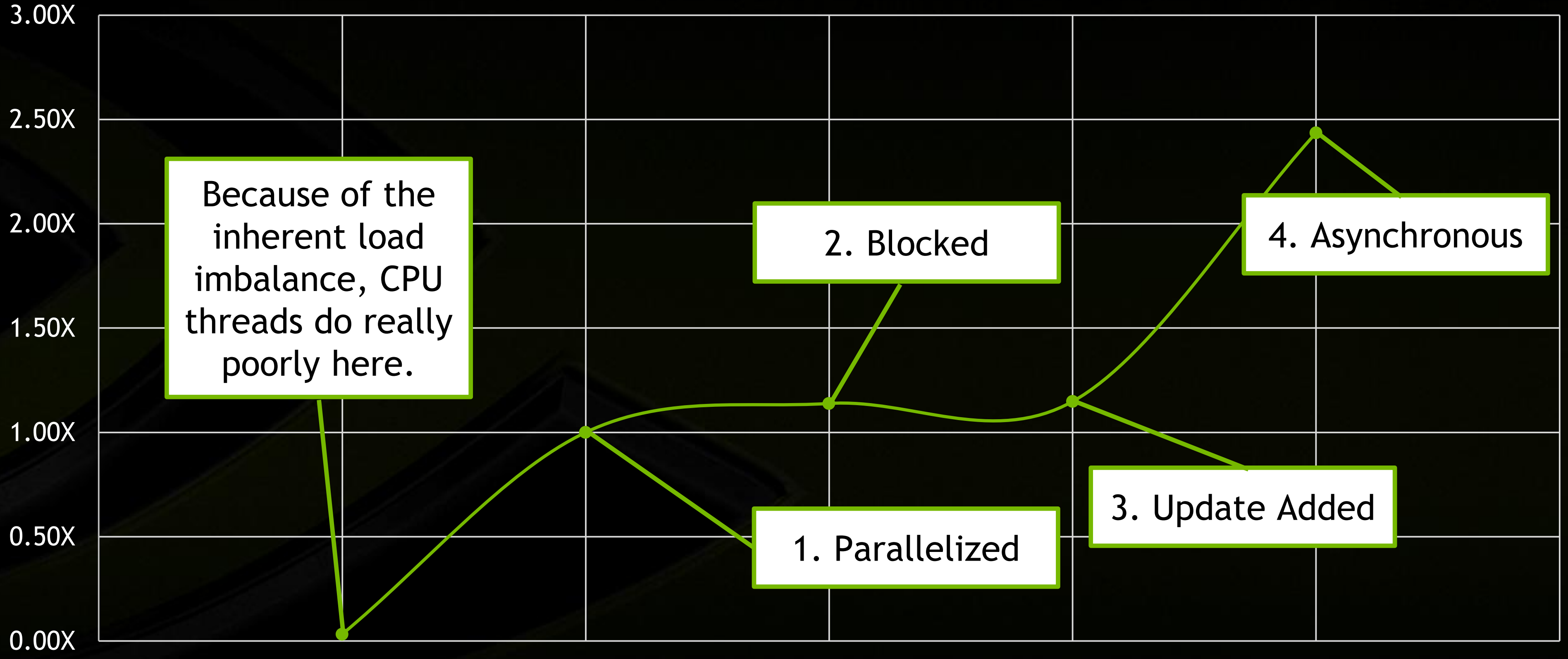
Duration	
Session	1.793 s (1,793,4
Kernel	172.387 ms (17.
Invocations	8
Importance	48.8%

Notice the kernel launches seem to take differing amounts of time. What if we tried smaller blocks?

VARYING THE NUMBER OF BLOCKS



SPEED-UP STEP BY STEP



ASYNCHRONOUS TIPS

- Reuse streams, they're expensive to create
  - Pre-create them
  - Consider `async(block%2)` to re-use just 2 streams
- Don't forget to `wait`
- Test with 1 stream first

MULTI-GPU PROGRAMMING

MULTI-GPU OPENACC

`acc_set_device_num(number, device_type)`

- Selects the device to use for all regions that follow

`acc_get_num_devices(device_type)`

- Queries how many devices are available of a given type

- Most often, one will set a device number once per CPU thread

MULTI-GPU MANDELBROT

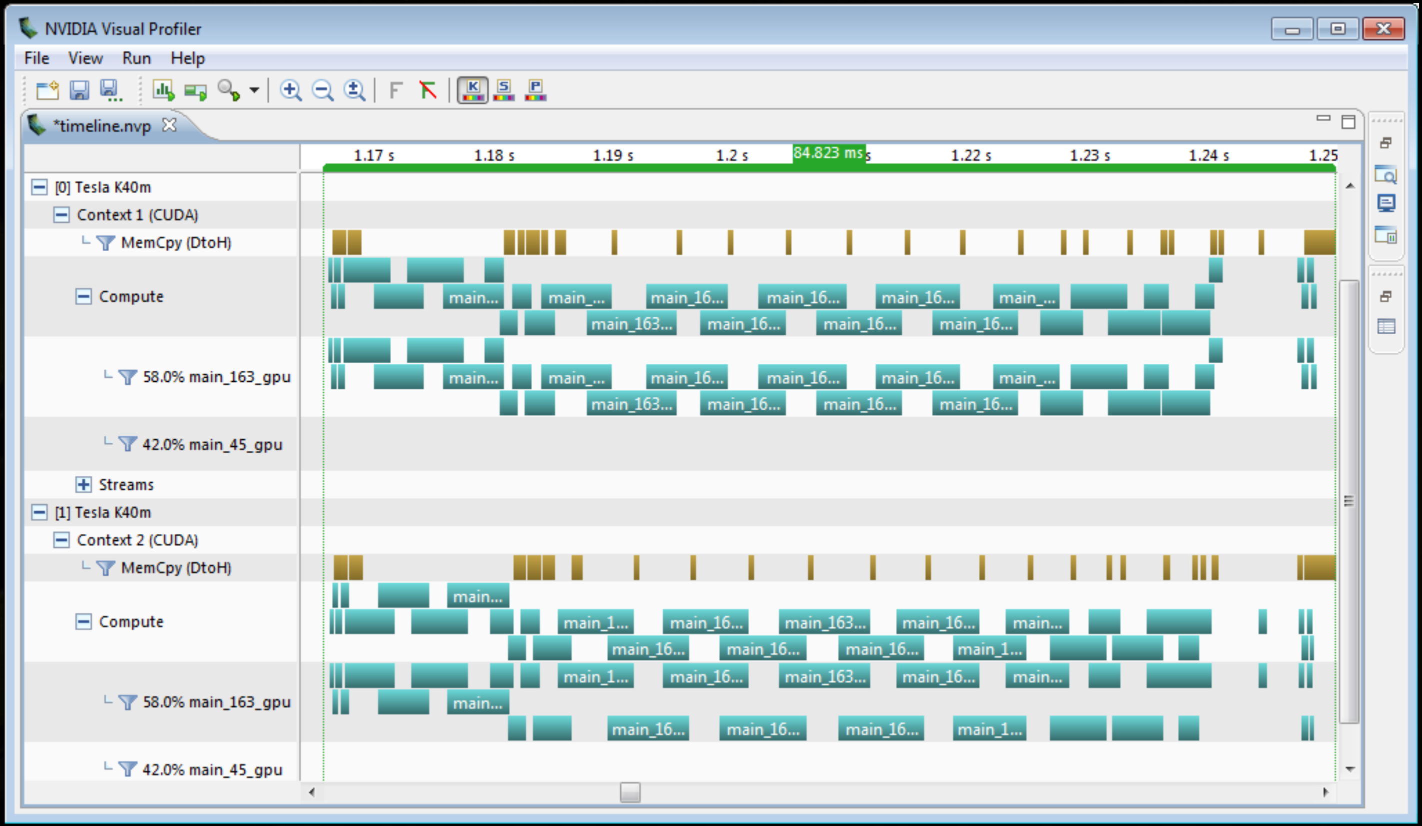
```
 for (int gpu=0; gpu < 2 ; gpu ++)
 {
  acc_set_device_num(gpu,acc_device_nvidia);
#pragma acc enter data create(image[:bytes])
 }

 for(int block=0; block < numblocks; block++)
 {
  int ystart = block * blocksize;
  int yend   = ystart + blocksize;
  acc_set_device_num(block%2,acc_device_nvidia);
#pragma acc parallel loop async(block)
  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(block)
 }
 for (int gpu=0; gpu < 2 ; gpu ++)
 {
  acc_set_device_num(gpu,acc_device_nvidia);
#pragma acc wait
#pragma acc exit data delete(image)
 }
```

Allocate space on each device

Alternate devices per block

Clean up the devices

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

CUDA ZONE

Home › CUDA Zone

Explore the world's biggest GPU developer conference.

EXPLORE CUDA ZONE

WHAT IS CUDA

Learn more about the CUDA parallel computing platform and programming model.

GET STARTED - PARALLEL COMPUTING

Find out about different paths and options for deploying CUDA and GPU Computing in your project

CUDA IN ACTION - RESEARCH & APPS

Supercomputing is now accessible for every researcher and scientist. Find latest research, applications and links to how CUDA is transforming the industry

CUDA TOOLKIT

The NVIDIA CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications.

CUDA EDUCATION & TRAINING

Get educated with online courses, webinars, University Courses and wealth of technical papers & documentation

CUDA TOOLS & ECOSYSTEM

Learn more about powerful CUDA tools, libraries, languages, and other development aids available from NVIDIA & partners.

PARALLEL FOR ALL BLOG

5 Things You Should Know About the New Maxwell GPU Architecture

February 21, 2014

The introduction this week of NVIDIA's first-generation "Maxwell" GPUs is a very exciting moment for GPU computing. These first Maxwell products, such as the GeForce GTX 750 Ti, are based on the GM107 GPU and are designed for use in low-power environments such as notebooks and small form factor computers. What is exciting about this announcement [...]
...

CUDACasts Episode 17: Unstructured Data Lifetimes in OpenACC 2.0

QUICKLINKS

- CUDA Downloads
- CUDA GPUs
- NVIDIA Nsight Visual Studio Edition
- Get Started - Parallel Computing
- CUDA Tools & Ecosystem
- CUDA FAQ

NVIDIA DEVELOPER PROGRAMS

Get exclusive access to the latest software, report bugs and receive notifications for special events.

LATEST NEWS

- NVIDIA Nsight Visual Studio Edition 3.2 Available Now With Windows 8.1 Support And Improved DirectCompute Profiling
- OpenACC Training: Nov 5th
- Nsight Visual Studio Edition 3.1 Final Now Available With Visual Studio 2012, Direct3D 11.1 And CUDA 5.5 Support!
- Robotics Expert Starts A New Facebook GPU Computing Community
- CUDA 5.5 Production Release - Now Available

More

BOOKS

- CUDA Fortran For Scientists And Engineers
- CUDA HANDBOOK: A COMPREHENSIVE GUIDE TO GPU PROGRAMMING

OPENACC & CUDA STREAMS

OpenACC *suggests* two functions for interoperating with CUDA streams:

- `void* acc_get_cuda_stream( int async );`

- `int acc_set_cuda_stream( int async, void* stream );`

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)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}

cublasShutdown();
```

OPENACC DEVICEPTR

The deviceptr clause informs the compiler that an object is already on the device, so no translation is necessary.

- Valid for parallel, kernels, and data

```
cudaMallocManaged((void*)&x,(size_t)n*sizeof(float));
cudaMallocManaged((void*)&y,(size_t)n*sizeof(float));
```

```
#pragma acc parallel loop deviceptr(x,y)
for(int i=0; i<n ; i++)
{
  y(i) = a*x(i)+y(i)
}
```

Do not translate x and y, they are already on the device.

DEVICEPTR EXAMPLE

OpenACC Kernels

```
void saxpy(int n, float a, float * restrict
x, float * restrict y)
{
#pragma acc kernels deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

By passing a device pointer to an OpenACC region, it's possible to add OpenACC to an existing CUDA code.

CUDA C Main

```
int main(int argc, char **argv)
{
  float *x, *y, tmp;
  int n = 1<<20, i;

  cudaMalloc((void*)&x,(size_t)n*sizeof(float));
  cudaMalloc((void*)&y,(size_t)n*sizeof(float));

  ...

  saxpy(n, 2.0, x, y);
  cudaMemcpy(&tmp,y,(size_t)sizeof(float),
             cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory is managed via standard CUDA calls.

OPENACC & THRUST

Thrust (thrust.github.io) is a STL-like library for C++ on accelerators.

- High-level interface
- Host/Device container classes
- Common parallel algorithms

It’s possible to cast Thrust vectors to device pointers for use with OpenACC

```
void saxpy(int n, float a, float * restrict
x, float * restrict y)
{
#pragma acc kernels deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
for(int i=0; i<N; i++)
{
  x[i] = 1.0f;
  y[i] = 0.0f;
}

// Copy to Device
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

saxpy(N,2.0, d_x.data().get(),
             d_y.data().get());

// Copy back to host
y = d_y;
```

CUDA DEVICE ROUTINES AND OPENACC

```
extern “C” __device__ void
f1dev( float* a, float* b, int i ){
  a[i] = .... b[i] .... ;
}
```

Even CUDA `__device__` functions can be called from OpenACC if declared with `acc routine`.

```
#pragma acc routine seq
extern “C” void f1dev( float*,
float* int );
...
#pragma acc parallel loop \
  present( a[0:n], b[0:n] )
for( int i = 0; i < n; ++i )
{
  f1dev( a, b, i );
}
```

OPENACC ACC_MAP_DATA FUNCTION

The `acc_map_data` (`acc_unmap_data`) maps (unmaps) an existing device allocation to an OpenACC variable.

```
cudaMalloc((void*)&x_d,(size_t)n*sizeof(float));
acc_map_data(x, x_d, n*sizeof(float));
cudaMalloc((void*)&y_d,(size_t)n*sizeof(float));
acc_map_data(y, y_d, n*sizeof(float));
```

Allocate device arrays with CUDA and *map* to OpenACC

```
#pragma acc parallel loop
for(int i=0; i<n ; i++)
{
  y(i) = a*x(i)+y(i)
}
```

Here x and y will reuse the memory of x_d and y_d

ATOMIC DIRECTIVE

OPENACC ATOMIC DIRECTIVE

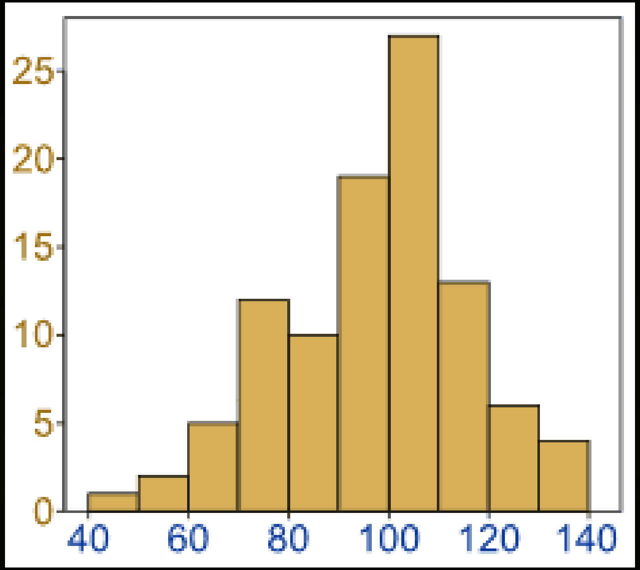
atomic: subsequent block of code is performed atomically with respect to other threads on the accelerator

Clauses: read, write, update, capture

```
#pragma acc parallel loop
for(int i=0; i<N; i++) {
  #pragma acc atomic update
  a[i%100]++;
}
```

OPENACC ATOMIC: HISTOGRAM

```
#pragma acc data copyin(a[0:N]) copyout(h[0:HN])
for(int it=0;it<ITERS;it++)
{
  #pragma acc parallel loop
  for(int i=0;i<HN;i++)
    h[i]=0;

  #pragma acc parallel loop
  for(int i=0;i<N;i++) {
    #pragma acc atomic
    h[a[i]]+=1;
  }
}
```



MISC. ADVICE AND TECHNIQUES

WRITE PARALLELIZABLE LOOPS

Use countable loops

C99: while->for

Fortran: while->do

```
bool found=false;
while(!found && i<N) {
  if(a[i]==val) {
    found=true
    loc=i;
  }
  i++;
}
```

```
bool found=false;
for(int i=0;i<N;i++) {
  if(a[i]==val) {
    found=true
    loc=i;
  }
}
```

Avoid pointer arithmetic (use array syntax)

Write rectangular loops (compiler cannot parallelize triangular loops)

```
for(int i=0;i<N;i++) {
  for(int j=i;j<N;j++) {
    sum+=A[i][j];
  }
}
```

```
for(int i=0;i<N;i++) {
  for(int j=0;j<N;j++) {
    if(j>=i)
      sum+=A[i][j];
  }
}
```

ALIASING CAN PREVENT PARALLELIZATION

```
23, Loop is parallelizable
        Accelerator kernel generated
        23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    25, Complex loop carried dependence of 'b->' prevents parallelization
        Loop carried dependence of 'a->' prevents parallelization
        Loop carried backward dependence of 'a->' prevents vectorization
        Accelerator scalar kernel generated
    27, Complex loop carried dependence of 'a->' prevents parallelization
        Loop carried dependence of 'b->' prevents parallelization
        Loop carried backward dependence of 'b->' prevents vectorization
        Accelerator scalar kernel generated
```

C99: RESTRICT KEYWORD

- Declaration of intent given by the programmer to the compiler

  Applied to a pointer, e.g.

  ```
  float *restrict ptr
  ```

  Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”*

- Parallelizing compilers often require `restrict` to determine independence
  - Otherwise the compiler can’t parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

~~`float restrict *ptr`~~

`float *restrict ptr`

http://en.wikipedia.org/wiki/Restrict

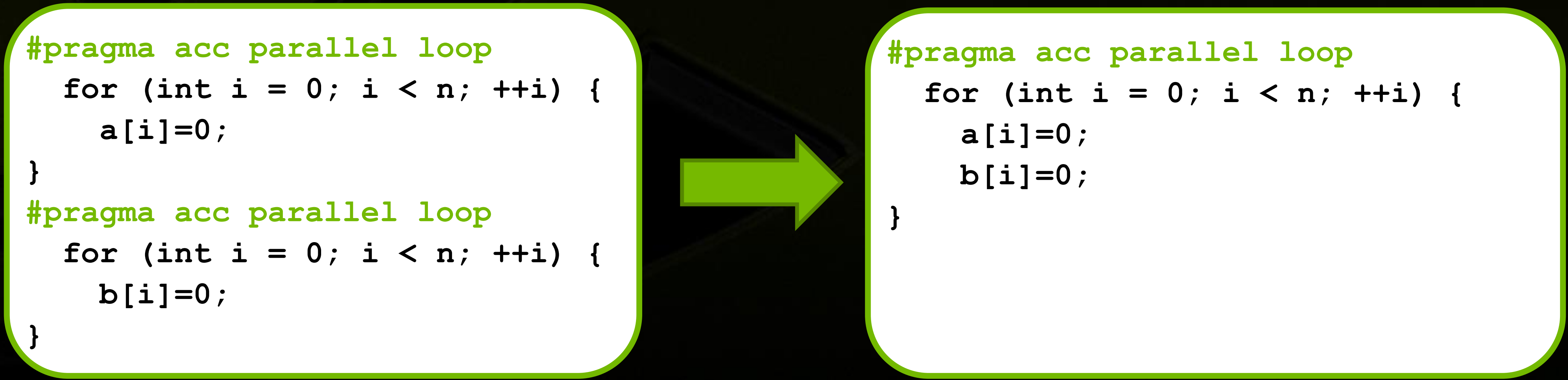
INLINING

- When possible aggressively inline functions/routines
  - This is especially important for inner loop calculations
  - Inlined routines frequently perform better than acc routines because the compiler has more information.

```
#pragma acc routine seq
inline
int IDX(int row, int col, int LDA) {
  return row*LDA+col;
}
```

KERNEL FUSION

- Kernel calls are expensive
  - Each call can take over 10us in order to launch
  - It is often a good idea to combine loops of same trip counts containing very few lines of code
- Kernel Fusion (i.e. Loop fusion)
  - Join nearby kernels into a single kernel



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

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

LOOP FISSION

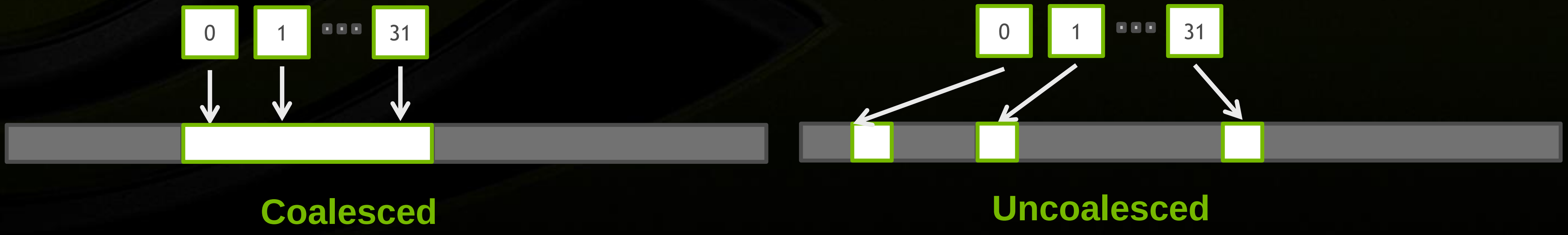
- Loops that are exceptionally long may result in kernels that are resource-bound, resulting in low GPU occupancy.
- This is particularly true for outer parallel loops containing nested loops
- Caution: This may introduce temporaries.

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

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

MEMORY COALESCING

- *Coalesced* access:
  - A group of 32 contiguous threads (“warp”) accessing adjacent words
  - Few transactions and high utilization
- *Uncoalesced* access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization
- For best performance the vector loop should access memory contiguously (stride-1)



COMPLEX DATA LAYOUTS

- OpenACC works best with flat arrays
- Some compilers handle complex types (structs, classes, derived types) better than others
    - Doesn’t always work, particularly if members are dynamically allocated
    - Work around: Use local pointers to struct members (C99 & Fortran)

May work

```
#pragma acc parallel loop \
        copy(a, a.data[0:a.N])
for(i=0;i<a.N;i++)
  a.data[i]=0;
```

Generally Works

```
int N=a.N;
float *data=a.data;
#pragma acc parallel loop \
        copy(data[0:N])
for(i=0;i<N;i++)
  data[i]=0;
```

OPENACC INDEPENDENT CLAUSE

Specifies that loop iterations are data independent. This overrides any compiler dependency analysis. This is implied for *parallel loop*.

```
#pragma acc kernels
{
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  a[i] = 0.0;
  b[i] = 1.0;
  c[i] = 2.0;
}
#pragma acc loop independent
for(int i=0; i<N; i++)
{
  a(i) = b(i) + c(i)
}
}
```

kernel 1

kernel 2

Informs the compiler that both loops are safe to parallelize so it will generate both kernels.

OPENACC DEBUGGING

- Most OpenACC directives accept an **`if(condition)`** clause

```
#pragma acc update self(A) if(debug)
#pragma acc parallel loop if(!debug)
[…]
#pragma acc update device(A) if(debug)
```

- Use **`default(none)`** to force explicit data directives

```
#pragma acc data copy(…) create(…) default(none)
```

QUESTIONS & DISCUSSION