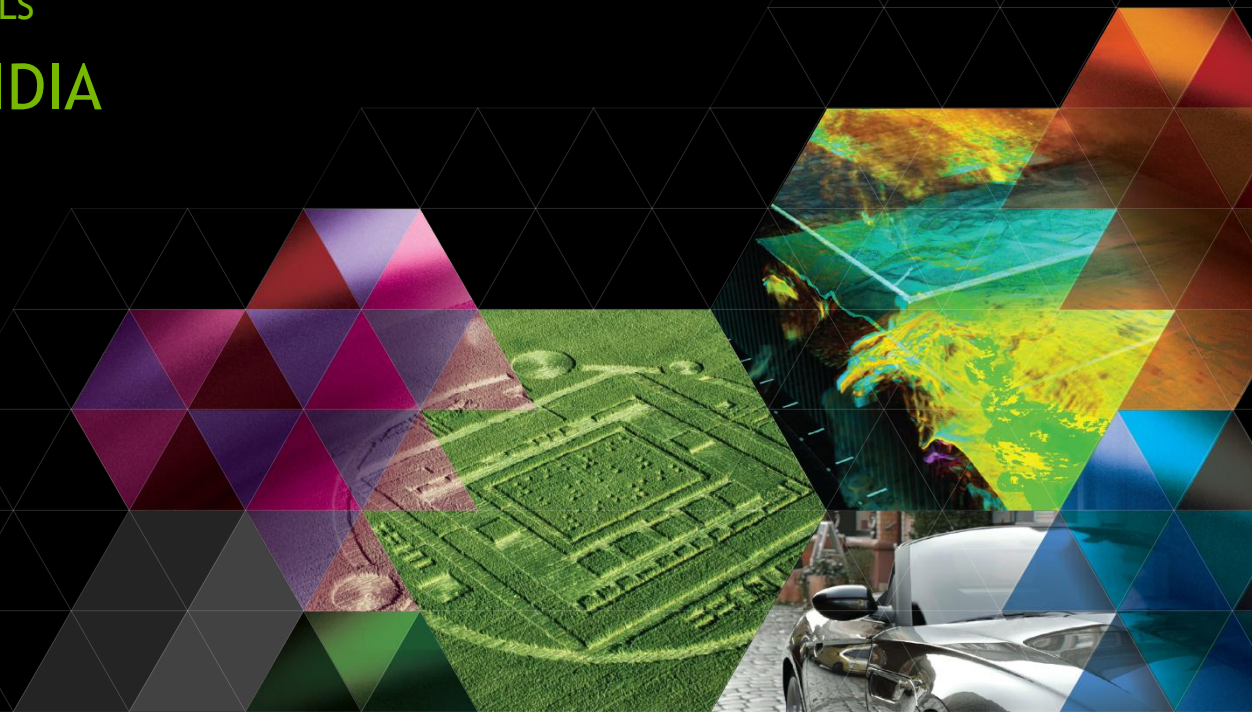
GPU TECHNOLOGY CONFERENCE

NVIDIA

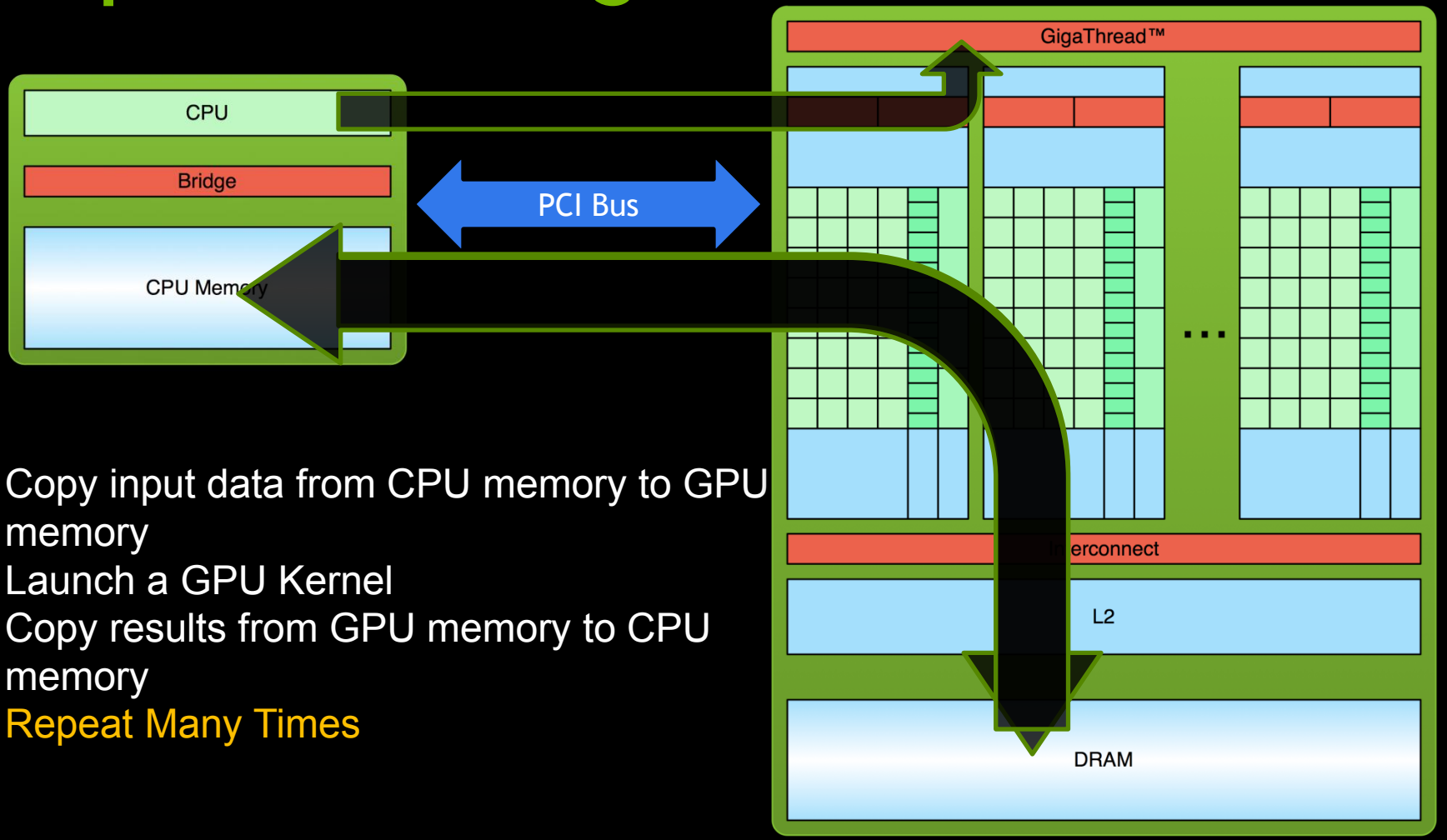
CUDA STREAMS

BEST PRACTICES AND COMMON PITFALLS

Justin Luitjens - NVIDIA



Simple Processing Flow

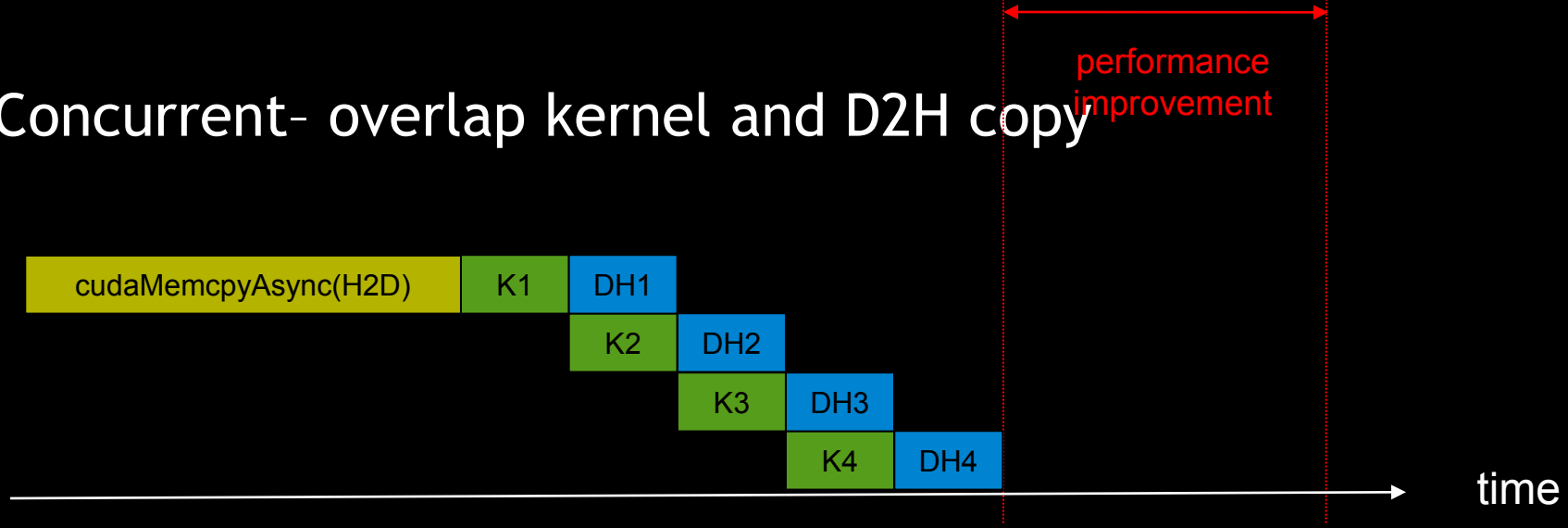


1. Copy input data from CPU memory to GPU memory
2. Launch a GPU Kernel
3. Copy results from GPU memory to CPU memory
4. Repeat Many Times

CONCURRENCY THROUGH PIPELINING

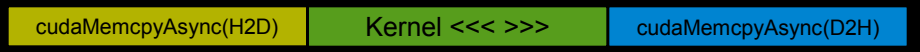
- Serial

- Concurrent- overlap kernel and D2H copy

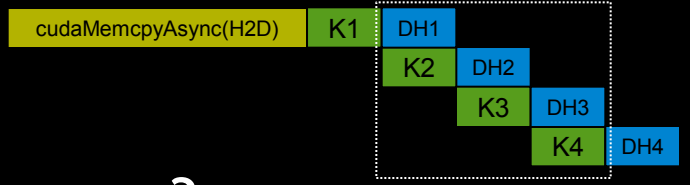


CONCURRENCY THROUGH PIPELINING

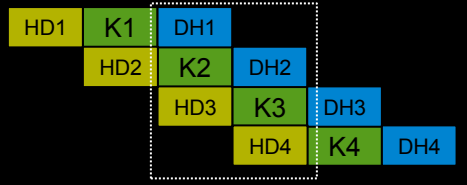
- Serial (1x)



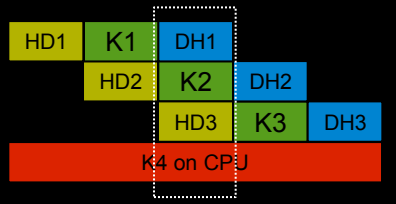
- 2-way concurrency (up to 2x)



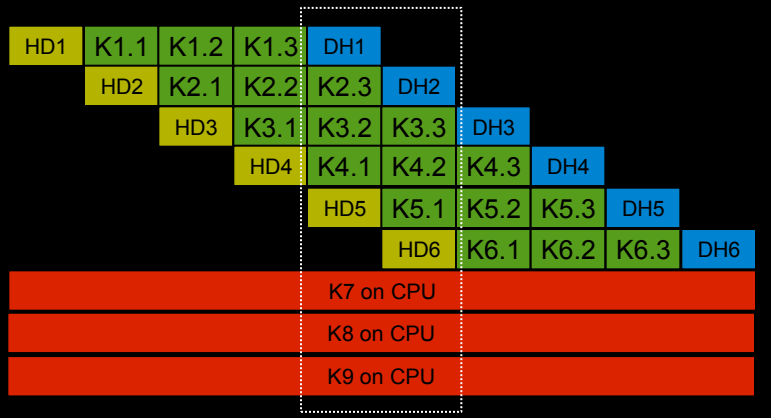
- 3-way concurrency (up to 3x)



- **4-way concurrency** (3x+)



- **4+ way concurrency**

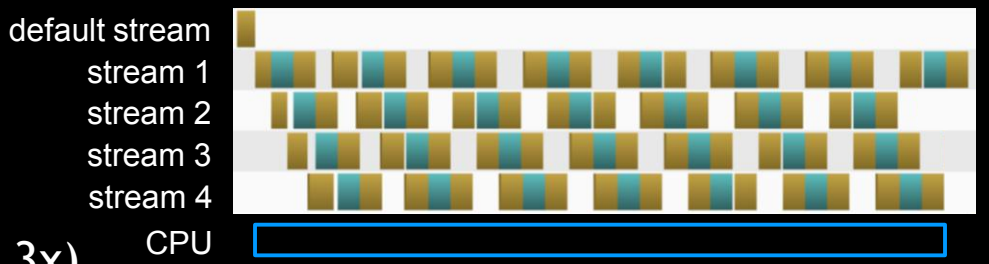


EXAMPLE – TILED DGEMM

- CPU (dual 6 core SandyBridge E5-2667 @2.9 Ghz, MKL)
  - 222 Gflop/s
- GPU (K20X)
  - Serial: 519 Gflop/s (2.3x)
  - 2-way: 663 Gflop/s (3x)
  - 3-way: 990 Gflop/s (4x)
- GPU + CPU
  - 4-way con.: 1180 Gflop/s (5.3x)

DGEMM: m=n=16384, k=1408

Nvidia Visual Profiler (nvvp)



- Obtain maximum performance by leveraging concurrency
- All PCI-E traffic is hidden
  - Effectively removes device memory size limitations!

ENABLING CONCURRENCY WITH STREAMS

SYNCHRONICITY IN CUDA

- All CUDA calls are either synchronous or asynchronous w.r.t the host
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately
- Kernel Launches are asynchronous Automatic overlap with host

CUDA STREAMS

- A stream is a queue of device work
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - e.g. Kernel launches, memory copies
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap

MANAGING STREAMS

- `cudaStream_t stream;`
  - Declares a stream handle
- `cudaStreamCreate(&stream);`
  - Allocates a stream
- `cudaStreamDestroy(stream);`
  - Deallocates a stream
  - Synchronizes host until work in stream has completed

PLACING WORK INTO A STREAM

- Stream is the 4th launch parameter
  - kernel<<< blocks , threads, smem, stream>>>();
- Stream is passed into some API calls
  - cudaMemcpyAsync(dst, src, size, dir, stream);

DEFAULT STREAM

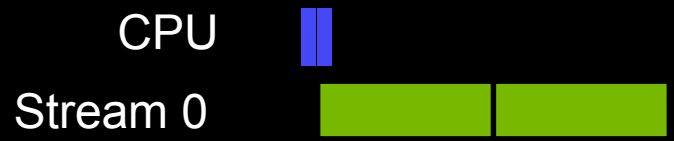
- Unless otherwise specified all calls are placed into a default stream
  - Often referred to as “Stream 0”
- Stream 0 has special synchronization rules
  - Synchronous with all streams
    - Operations in stream 0 cannot overlap other streams
- Exception: Streams with non-blocking flag set

```
cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking)
```

KERNEL CONCURRENCY

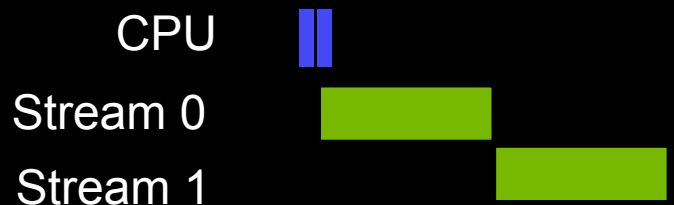
- Assume foo only utilizes 50% of the GPU
- Default stream

```
foo<<<blocks,threads>>>();
foo<<<blocks,threads>>>();
```



- Default & user streams

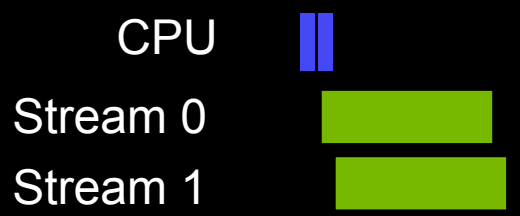
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
- Default & user streams

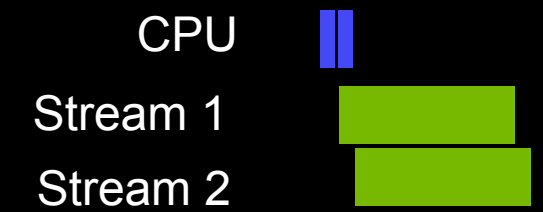
```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1,cudaStreamNonBlocking);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
  User streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
foo<<<blocks,threads,0,stream1>>>();
foo<<<blocks,threads,0,stream2>>>();
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



REVIEW

- The host is automatically asynchronous with kernel launches
- Use streams to control asynchronous behavior
  - Ordered within a stream (FIFO)
  - Unordered with other streams
  - Default stream is synchronous with all streams.

Concurrent Memory Copies

CONCURRENT MEMORY COPIES

- First we must review CUDA memory

THREE TYPES OF MEMORY

- Device Memory
  - Allocated using cudaMalloc
  - Cannot be paged
- Pageable Host Memory
  - Default allocation (e.g. malloc, calloc, new, etc)
  - Can be paged in and out by the OS
- Pinned (Page-Locked) Host Memory
  - Allocated using special allocators
  - Cannot be paged out by the OS

ALLOCATING PINNED MEMORY

- `cudaMallocHost(...) / cudaHostAlloc(...)`
  - Allocate/Free pinned memory on the host
  - Replaces malloc/free/new
- `cudaFreeHost(...)`
  - Frees memory allocated by cudaMallocHost or cudaHostAlloc
- `cudaHostRegister(...) / cudaHostUnregister(...)`
  - Pins/Unpins pagable memory (making it pinned memory)
  - Slow so don't do often
- Why pin memory?
  - Pagable memory is transferred using the host CPU
  - Pinned memory is transferred using the DMA engines
    - Frees the CPU for asynchronous execution
    - Achieves a higher percent of peak bandwidth

CONCURRENT MEMORY COPIES

- `cudaMemcpy(...)`
  - Places transfer into default stream
  - Synchronous: Must complete prior to returning
- `cudaMemcpyAsync(..., &stream)`
  - Places transfer into stream and returns immediately
- To achieve concurrency
  - Transfers must be in a non-default stream
  - Must use async copies
  - 1 transfer per direction at a time
  - Memory on the host must be pinned

PAGED MEMORY EXAMPLE

```
int *h_ptr, *d_ptr;

h_ptr=malloc(bytes);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

free(h_ptr);
cudaFree(d_ptr);
```

PINNED MEMORY: EXAMPLE 1

```
int *h_ptr, *d_ptr;

cudaMallocHost(&h_ptr,bytes);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

cudaFreeHost(h_ptr);
cudaFree(d_ptr);
```

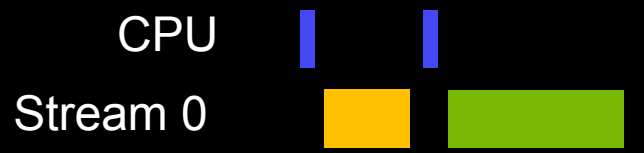
PINNED MEMORY: EXAMPLE 2

```
int *h_ptr, *d_ptr;

h_ptr=malloc(bytes);
cudaHostRegister(h_ptr,bytes,0);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

cudaHostUnregister(h_ptr);
free(h_ptr);
cudaFree(d_ptr);
```

CONCURRENCY EXAMPLES

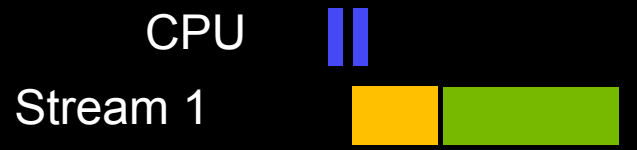
Synchronous

```
cudaMemcpy(...);
foo<<<...>>>();
```



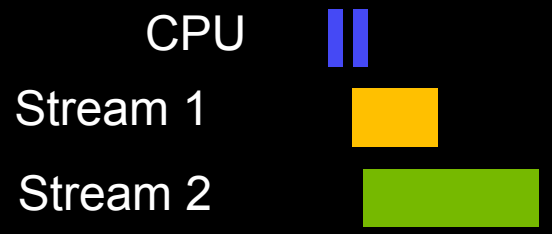
Asynchronous Same Stream

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream1>>>();
```



Asynchronous Different Streams

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream2>>>();
```



REVIEW

- Memory copies can execute concurrently if (and only if)
  - The memory copy is in a different non-default stream
  - The copy uses pinned memory on the host
  - The asynchronous API is called
  - There isn’t another memory copy occurring in the same direction at the same time.

Synchronization

SYNCHRONIZATION APIS

- Synchronize everything
  - cudaDeviceSynchronize()
    - Blocks host until all issued CUDA calls are complete
- Synchronize host w.r.t. a specific stream
  - cudaStreamSynchronize (stream)
    - Blocks host until all issued CUDA calls in stream are complete
- Synchronize host or devices using events

CUDA EVENTS

- Provide a mechanism to signal when operations have occurred in a stream
  - Useful for profiling and synchronization
- Events have a boolean state:
  - Occurred
  - Not Occurred
  - Important: Default state = occurred

MANAGING EVENTS

- `cudaEventCreate(&event)`
  - Creates an event
- `cudaEventDestroy(&event)`
  - Destroys an event
- `cudaEventCreateWithFlags(&ev, cudaEventDisableTiming)`
  - Disables timing to increase performance and avoid synchronization issues
- `cudaEventRecord(&event, stream)`
  - Set the event state to not occurred
  - Enqueue the event into a stream
  - Event state is set to occurred when it reaches the front of the stream

SYNCHRONIZATION USING EVENTS

- Synchronize using events
  - cudaEventQuery (event)
    - Returns CUDA_SUCCESS if an event has occurred
  - cudaEventSynchronize (event)
    - Blocks host until stream completes all outstanding calls
  - cudaStreamWaitEvent (stream, event)
    - Blocks stream until event occurs
    - Only blocks launches after this call
    - Does not block the host!
- Common multi-threading mistake:
  - Calling cudaEventSynchronize before cudaEventRecord

CUDA_LAUNCH_BLOCKING

- Environment variable which forces sychronization
  - export CUDA_LAUNCH_BLOCKING=1
  - All CUDA operations are synchronous w.r.t the host
- Useful for debugging race conditions
  - If it runs successfully with CUDA_LAUNCH_BLOCKING set but doesn't without you have a race condition.

REVIEW

- Synchronization with the host can be accomplished via
  - cudaDeviceSynchronize()
  - cudaStreamSynchronize(stream)
  - cudaEventSynchronize(event)
- Synchronization between streams can be accomplished with
  - cudaStreamWaitEvent(stream,event)
- Use CUDA_LAUNCH_BLOCKING to identify race conditions

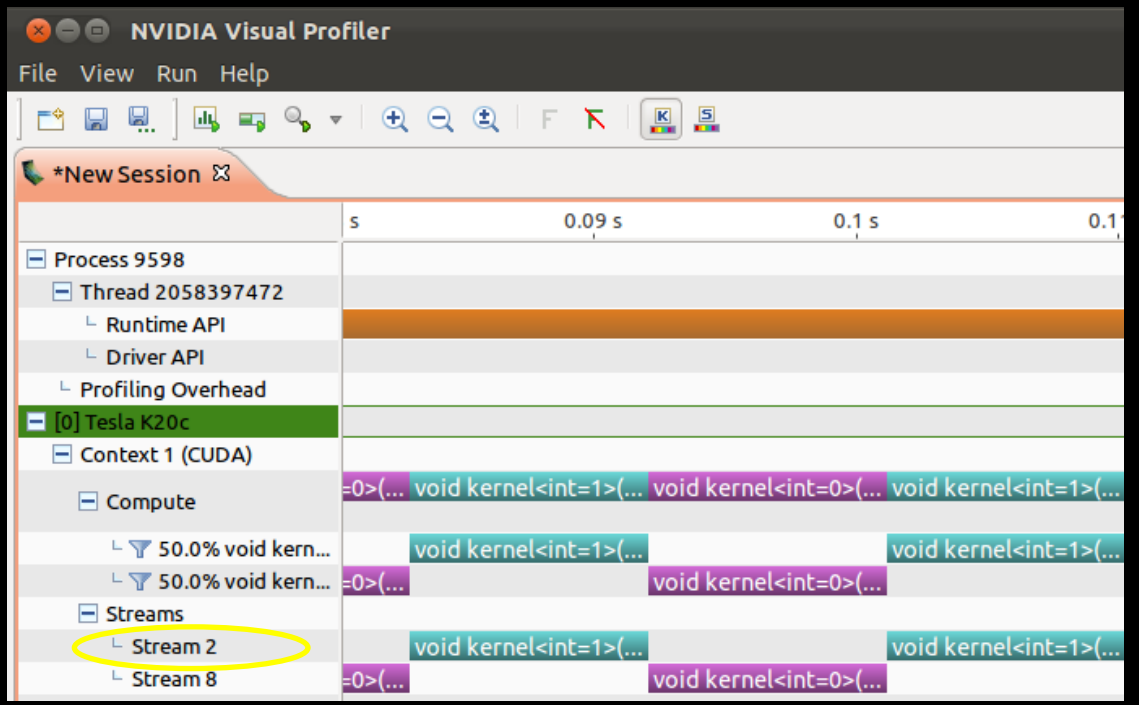
COMMON STREAMING PROBLEMS

COMMON STREAMING PROBLEMS

- The following is an attempt to demonstrate the most common streaming issues I've seen in customers applications
- They are loosely ordered according to how common they are

CASE STUDY 1-A

```
for(int i=0;i<repeat;i++)
{
  kernel<<<1,1,0,stream1>>>();
  kernel<<<1,1>>>();
}
```



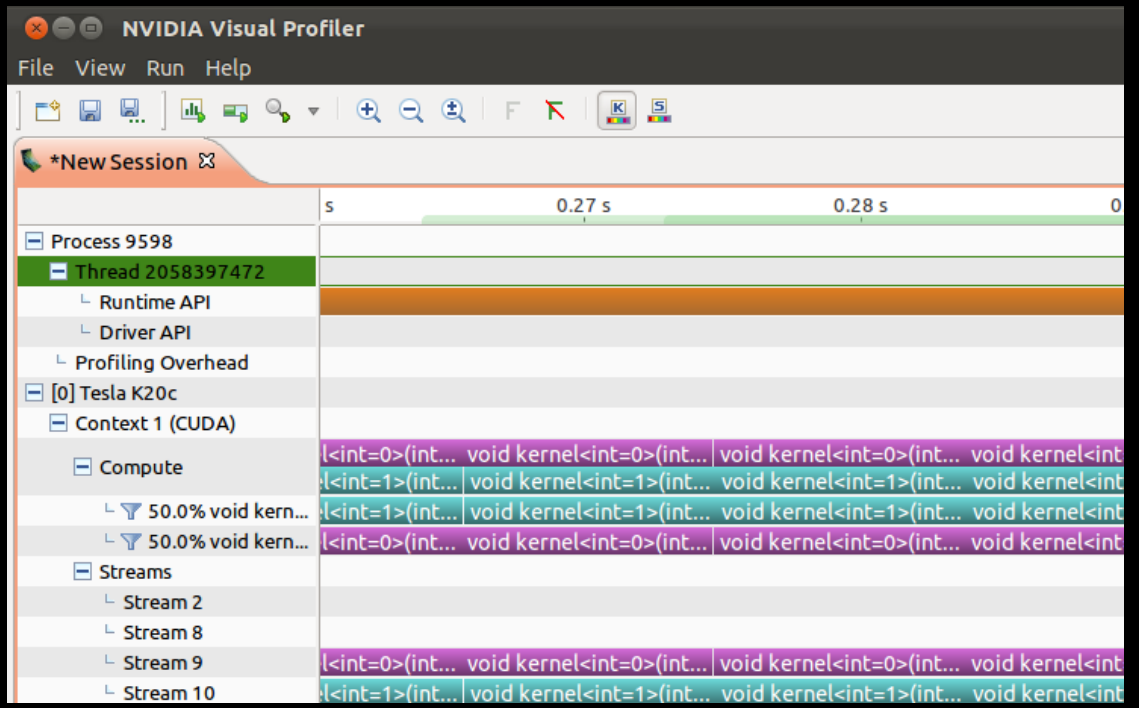
Problem:

One kernel is in the default stream

Stream 2 is the default stream

CASE STUDY 1-A

```
for(int i=0;i<repeat;i++) {
  kernel<<<1,1,0,stream1>>>();
  kernel<<<1,1,0,stream2>>>();
}
```



Solution:

Place each kernel in its own stream

PROBLEM 1: USING THE DEFAULT STREAM

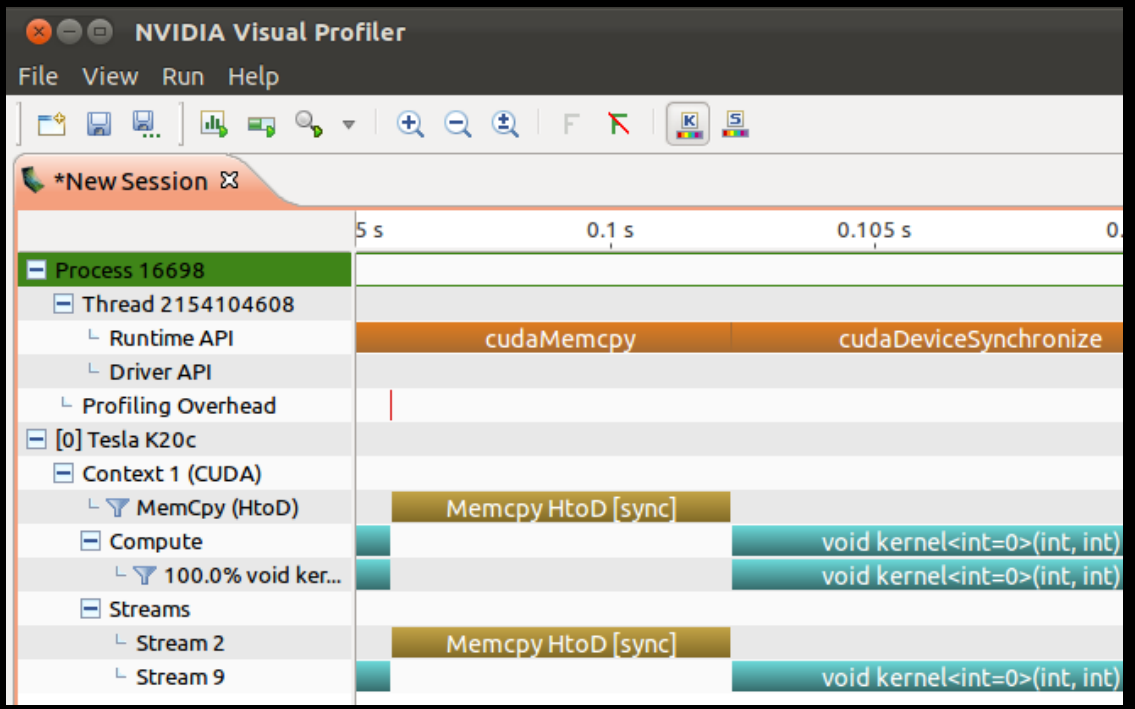
- Symptoms
  - One stream will not overlap other streams
    - In Cuda 5.0 stream 2 = default stream
  - Search for cudaEventRecord(event) , cudaMemcpyAsync(), etc.
    - If stream is not specified it is placed into the default stream
  - Search for kernel launches in the default stream
    - <<<a,b>>>
- Solutions
  - Move work into a non-default stream
  - cudaEventRecord(event,stream), cudaMemcpyAsync(…,stream)
  - Alternative: Allocate other streams as non-blocking streams

CASE STUDY 2-A

```
for(int i=0;i<repeat;i++) {
  cudaMemcpy(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice);
  kernel<<<1,1,0,stream2>>>();
  cudaDeviceSynchronize();
}
```

Problem:

Memory copy is synchronous

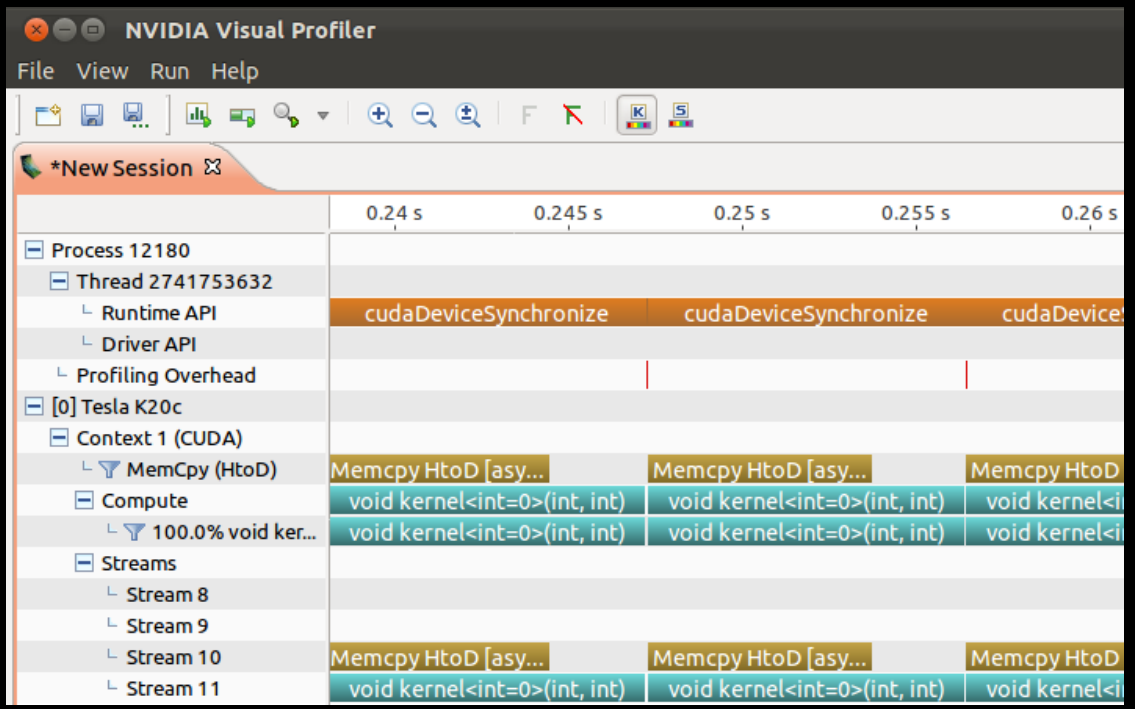


CASE STUDY 2-A

```
for(int i=0;i<repeat;i++) {
  cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);
  kernel<<<1,1,0,stream2>>>();
  cudaDeviceSynchronize();
}
```

Solution:

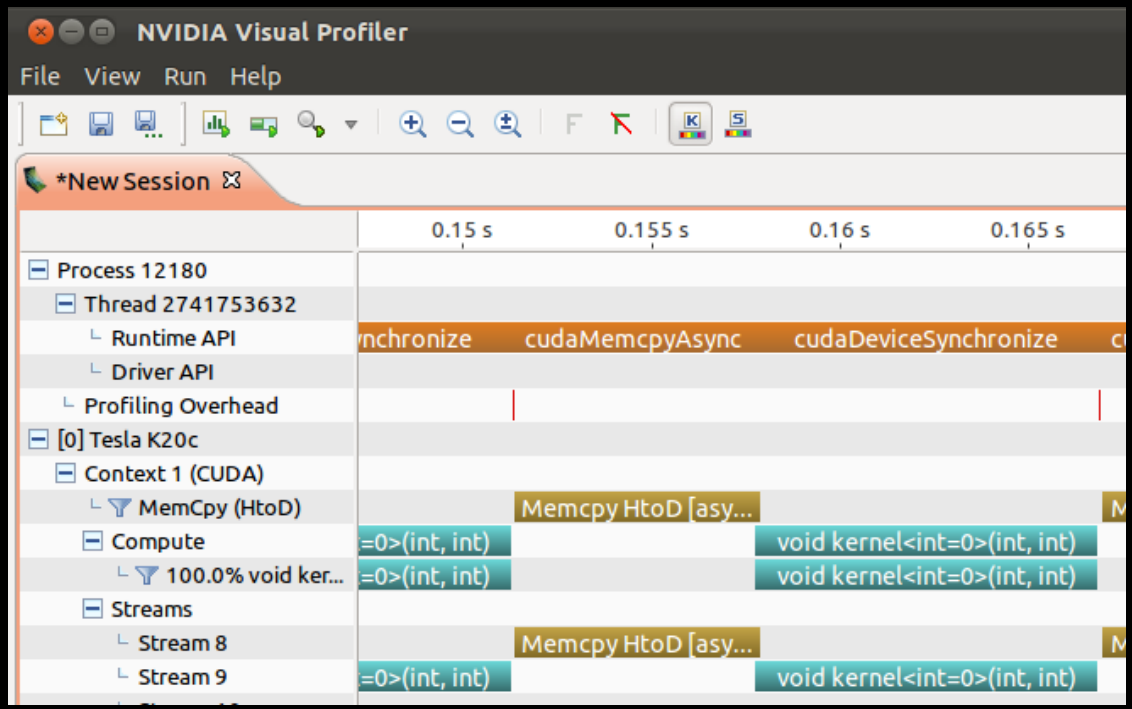
Use asynchronous API



CASE STUDY 2-B

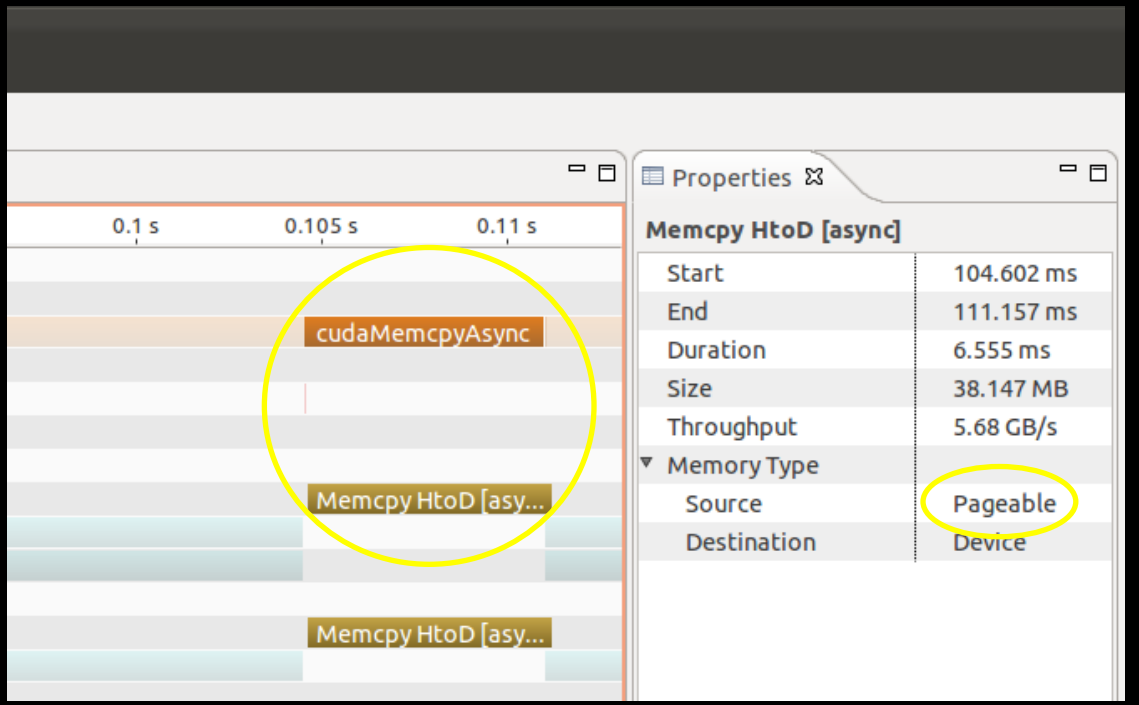
```
for(int i=0;i<repeat;i++) {
  cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);
  kernel<<<1,1,0,stream2>>>();
  cudaDeviceSynchronize();
}
```

Problem: ??



CASE STUDY 2-B

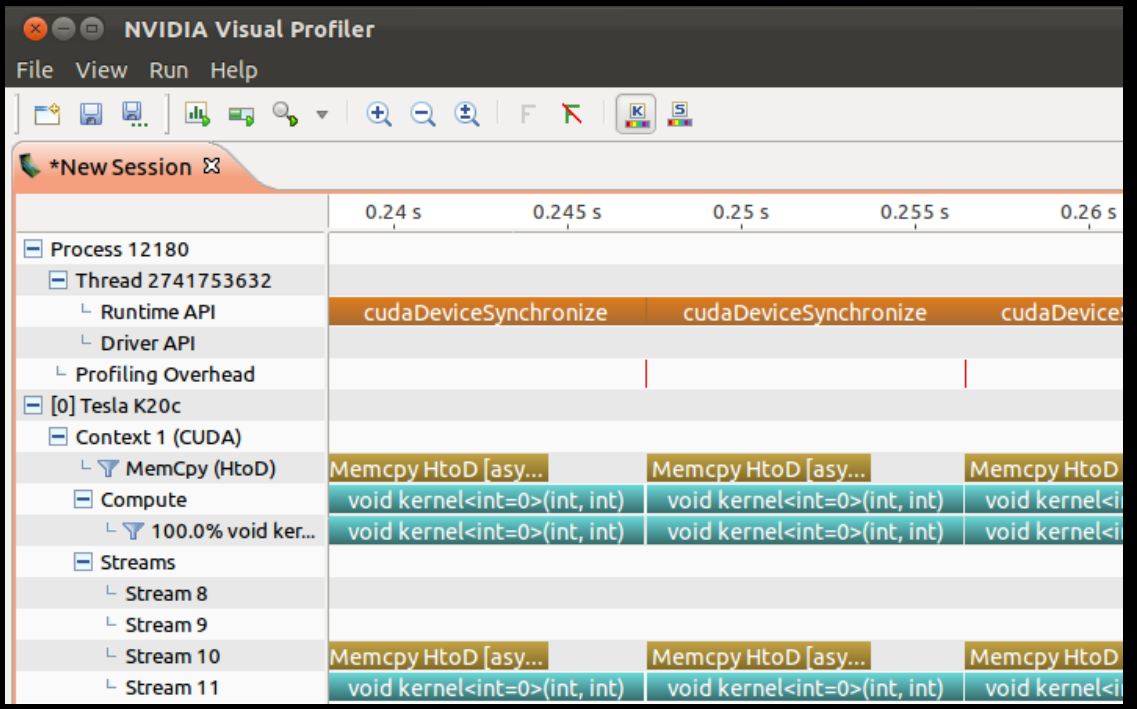
```
for(int i=0;i<repeat;i++) {
  cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);
  kernel<<<1,1,0,stream2>>>();
  cudaDeviceSynchronize();
}
```



Host doesn’t get ahead
Cuda 5.5 reports “Pageable” type

CASE STUDY 2-B

```
cudaHostRegister(h_ptr,bytes,0);
for(int i=0;i<repeat;i++) {
  cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);
  kernel<<<1,1,0,stream2>>>();
  cudaDeviceSynchronize();
}
cudaHostUnregister(h_ptr);
```



Solution:

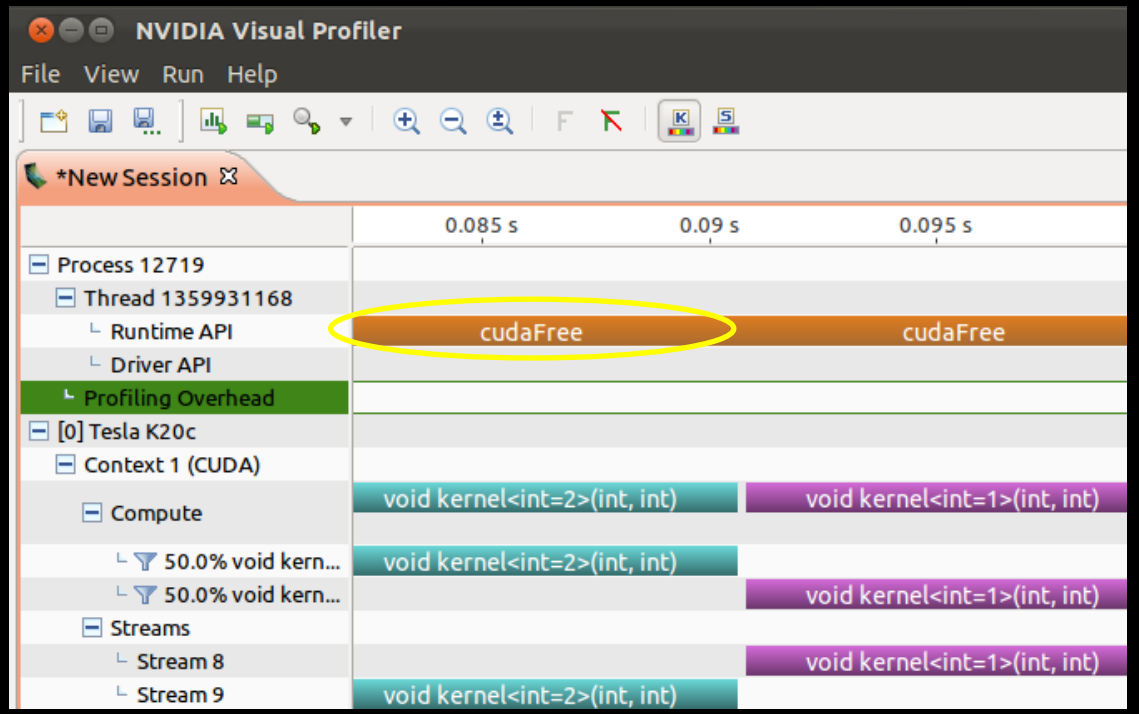
Pin host memory using cudaHostRegister or cudaMallocHost

PROBLEM 2: MEMORY TRANSFERS ISSUES

- Symptoms
  - Memory copies do not overlap
  - Host spends excessive time in memory copy API
  - Cuda reports “Pageable” memory (Cuda 5.5+)
- Solutions
  - Use asynchronous memory copies
  - Use pinned memory for host memory
    - cudaMallocHost or cudaHostRegister

CASE STUDY 3

```
void launchwork(cudaStream_t stream) {
  int *mem;
  cudaMalloc(&mem,bytes);
  kernel<<<1,1,0,stream>>>(mem);
  cudaFree(mem);
}
...

  for(int i=0;i<repeat;i++) {
    launchwork(stream1);
    launchwork(stream2);
  }
```



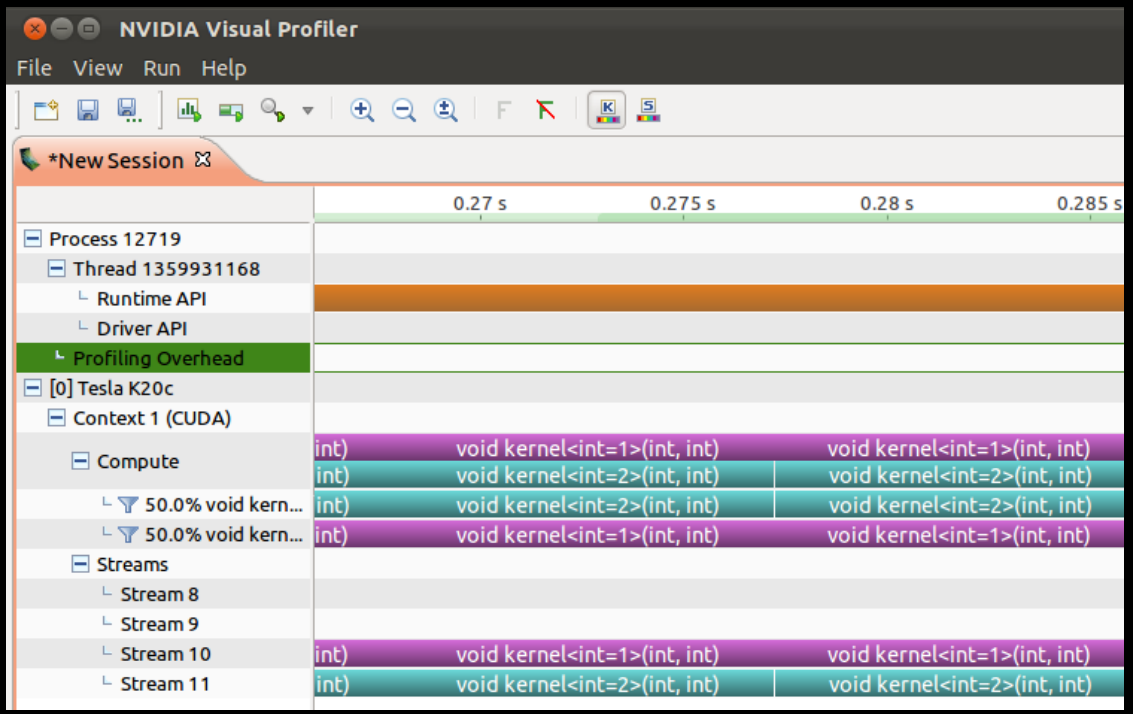
Host blocked in allocation/free

Problem:

Allocation & deallocation synchronize the device

CASE STUDY 3

```
void launchwork(cudaStream_t stream, int *mem) {
  kernel<<<1,1,0,stream>>>(mem);
}
...

  for(int i=0;i<repeat;i++) {
    launchwork<1>(stream1,mem1);
    launchwork<2>(stream2,mem2);
  }
```



Solution:

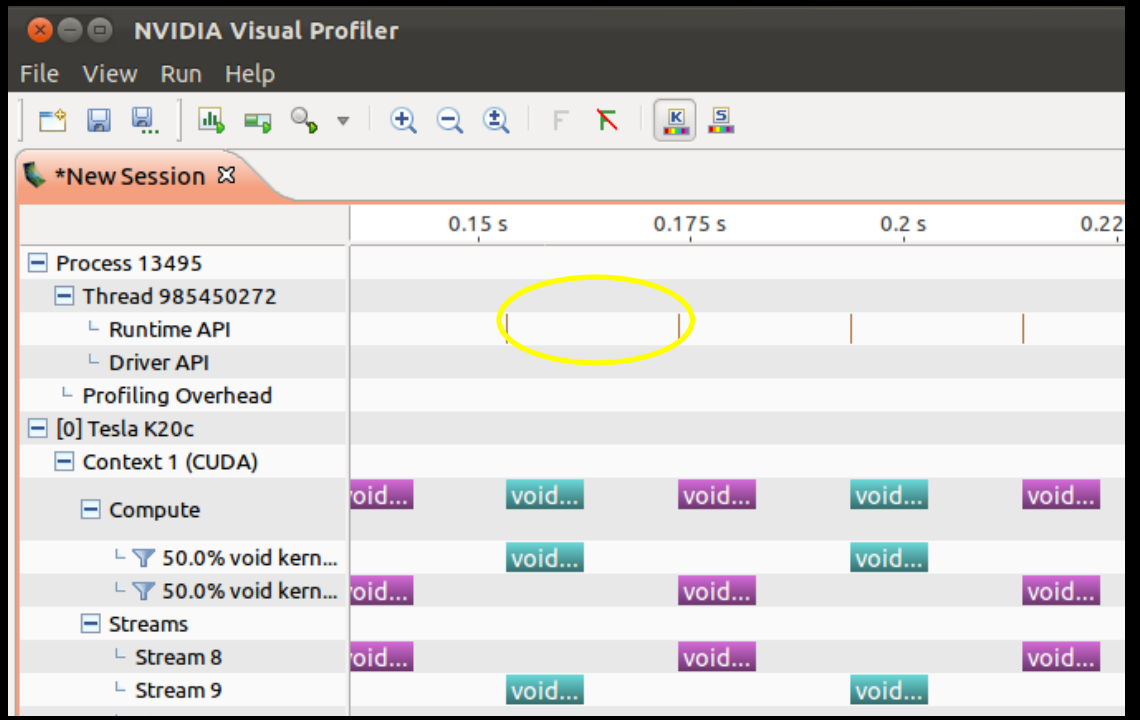
Seperate cuda memory and objects in streams and events

PROBLEM 3: IMPLICIT SYNCHRONIZATION

- Symptoms
  - Host does not get ahead
  - Host shows excessive time in certain API calls
    - cudaMalloc, cudaFree, cudaEventCreate, cudaEventDestroy, cudaStreamCreate, cudaStreamCreate, cudaHostRegister, cudaHostUnregister, cudaFuncSetCacheConfig
- Solution:
  - Reuse memory and data structures

CASE STUDY 4

```
for(int i=0;i<repeat;i++)
{
  hostwork();
  kernel<<<1,1,0,stream1>>>();
  hostwork();
  kernel<<<1,1,0,stream2>>>();
}
```



Problem:

Host is limiting performance

Host is outside of API calls

PROBLEM 4: LIMITED BY HOST

- Symptoms
  - Host is outside of cuda APIs
  - Large gaps in timeline where the host and device are empty
- Solution
  - Move more work to the GPU
  - Multi-thread host code

PROBLEM 5: LIMITED BY LAUNCH OVERHEAD

- Symptoms
  - Host does not get ahead
  - Kernels are short <30 us
  - Time between successive kernels is >10 us
- Solutions
  - Make longer running kernels
    - Fuse nearby kernels together
    - Batch work within a single kernel
    - Solve larger problems