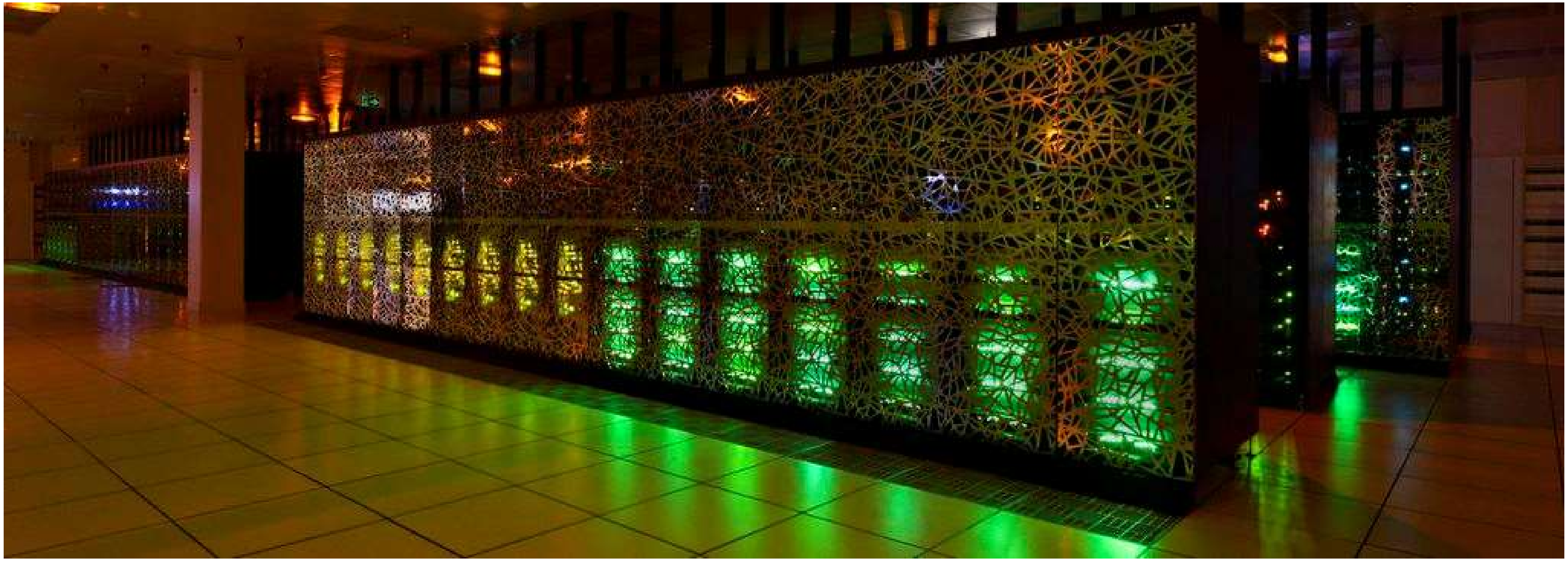
center for excellence in parallel programming

# CUDA

**Georges-Emmanuel Moulard**
**Paul Karlshöfer**





Bull atos technologies

# Data Transfers

16/09/2019

# Data Transfers

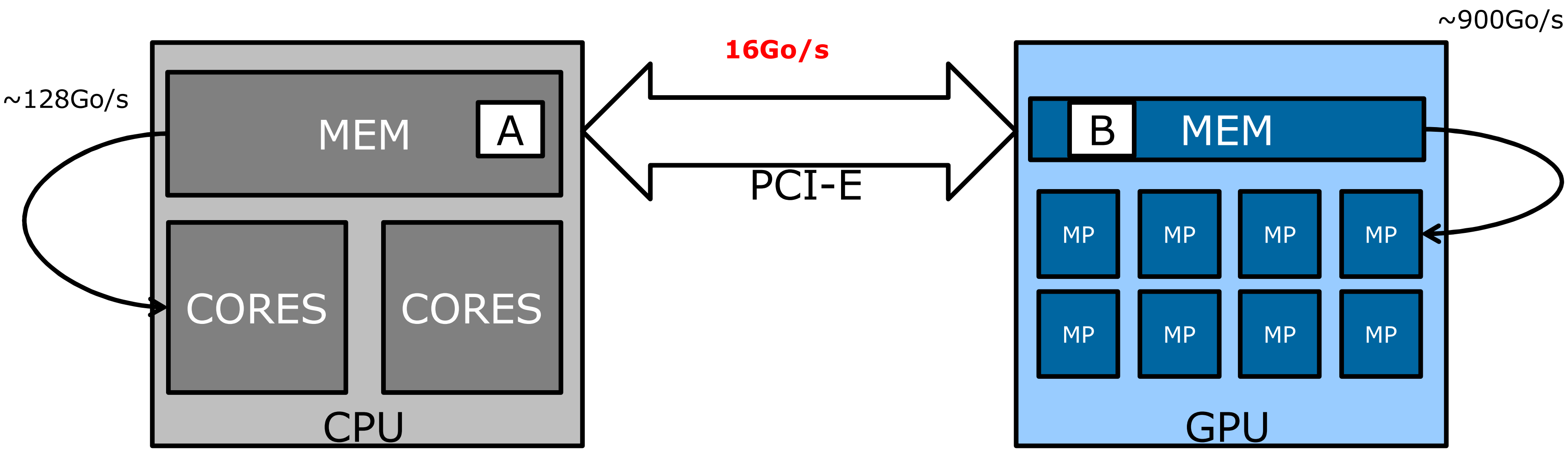
- When you start porting a code:
  - Find a parallel hotspot
  - Port it on GPU: it implies many data transfers
  - Start again

- Then ... don't expect to have a faster application!
  - Usually your application will be slower
  - Why?

# Data Transfers

- **GPUs are far from the main memory**
  - PCIe 3.0 x16: 16 Go/s (32 full duplex)
  - Skylake: ~128Go/s per socket
  - Tesla V100: ~900Go/s

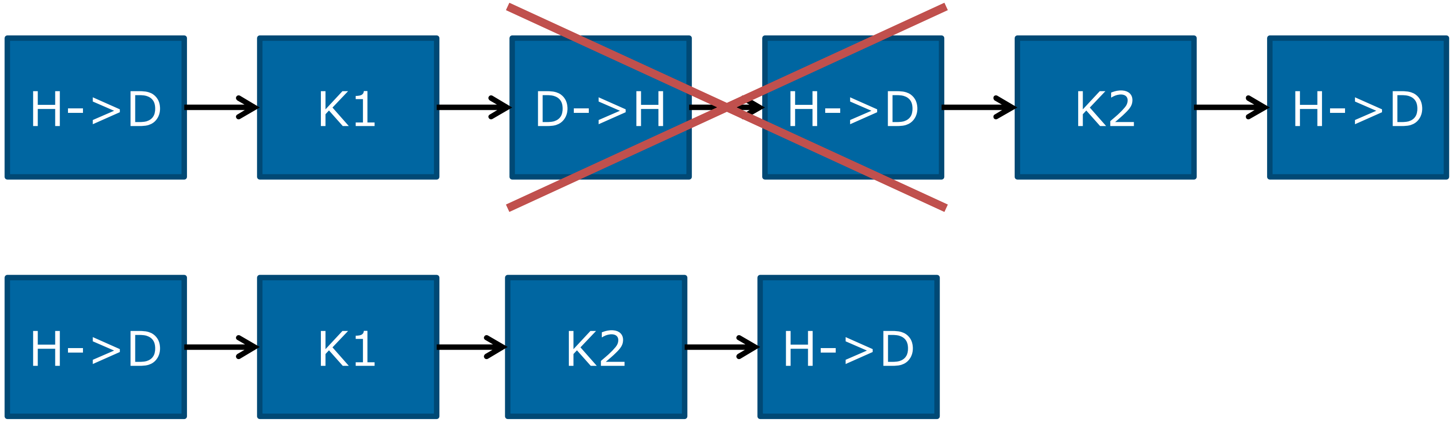
Data transfer
=
Less performance



# Data Transfer Optimizations

- Common sense:
  - Remove all useless transfers (can be hard to identify)




  - Do not transfer arrays only used on the GPU side
    - temporary arrays

# Data Transfer Optimizations

- Minimize the amount of data to transfers
  - use partial transfers if possible

# Data Transfer Optimizations

- Minimize the amount of data to transfers
  - use partial transfers if possible

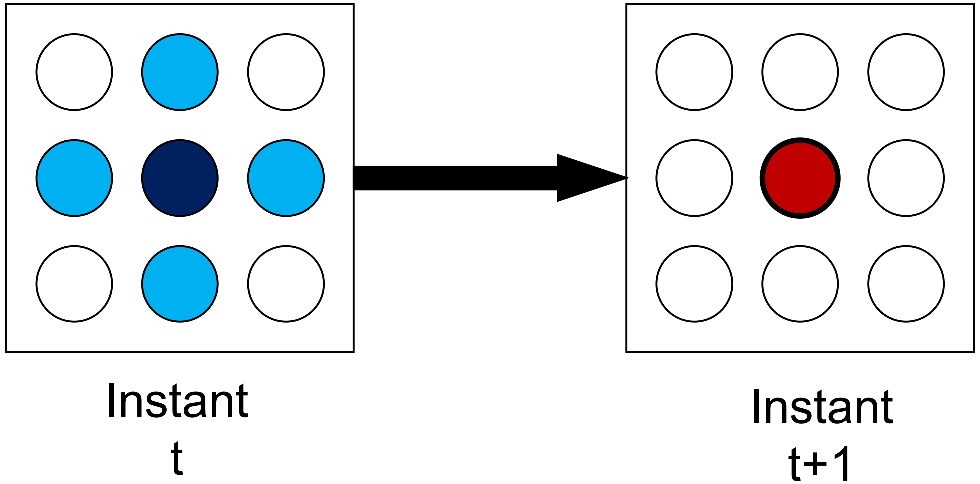
# Data Transfer Optimizations

- Minimize the amount of data to transfers
  - use partial transfers if possible


- Batch small transfers into larger ones:
  - Reduce data transfer initialization cost
  - Best PCIe BW utilization = total latency lowered

# Convolution Example

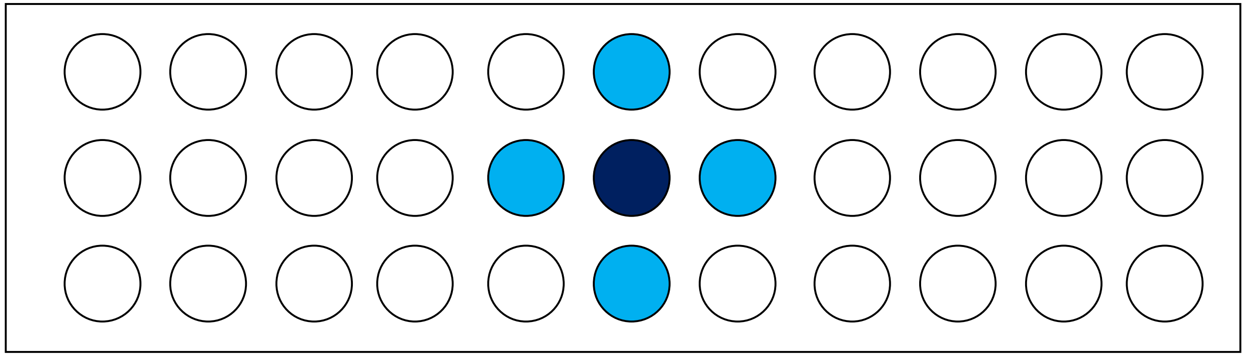
- Requested to process a point for t to t+1:
  - The point
  - Neighbours of that point



# Convolution Example

- Use of 2 processes for 1 domain

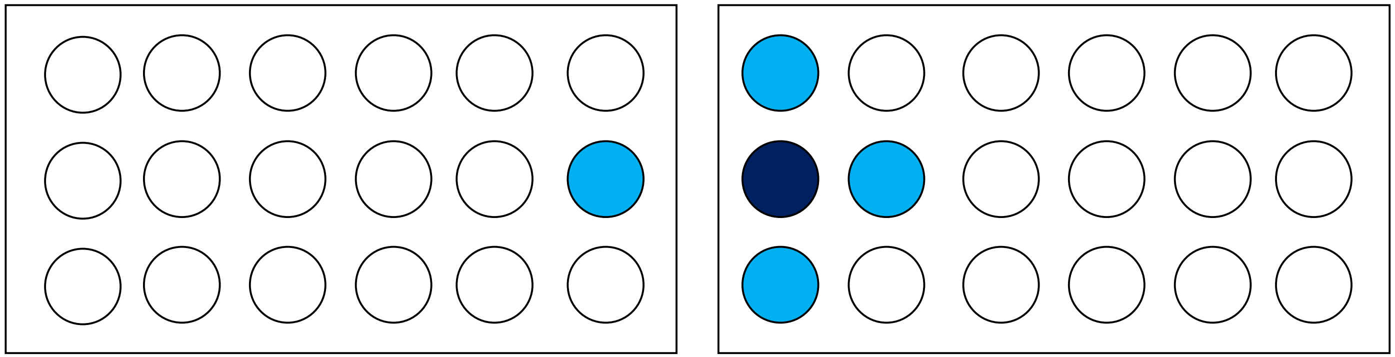
- Requested to process a point for t to t+1:
  - The point
  - Neighbours of that point
  - Extra, read-only points are needed on each side of the domain:
    - the ghosts



# Convolution Example

- Use of 2 processes for 1 domain

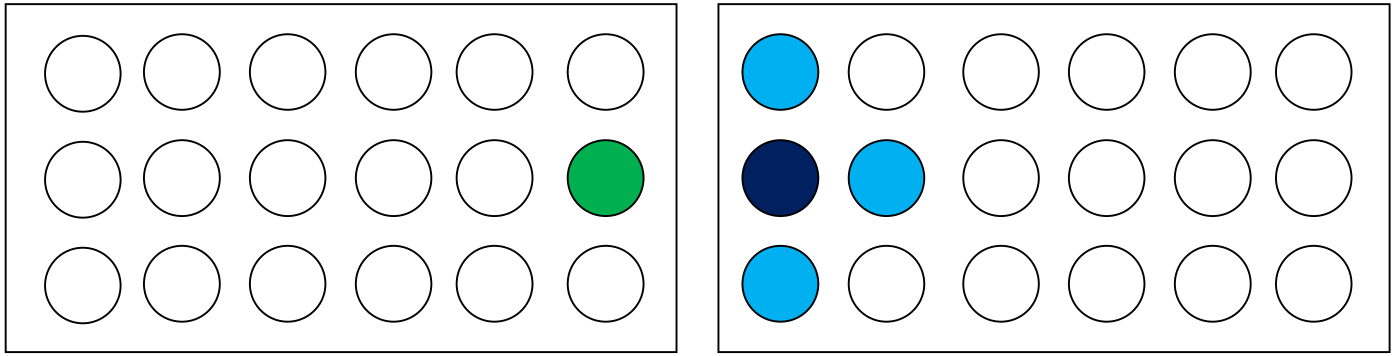
- Requested to process a point for t to t+1:
  - The point
  - Neighbours of that point
  - Extra, read-only points are needed on each side of the domain:
    - the ghosts



# Convolution Example

- Use of 2 processes for 1 domain

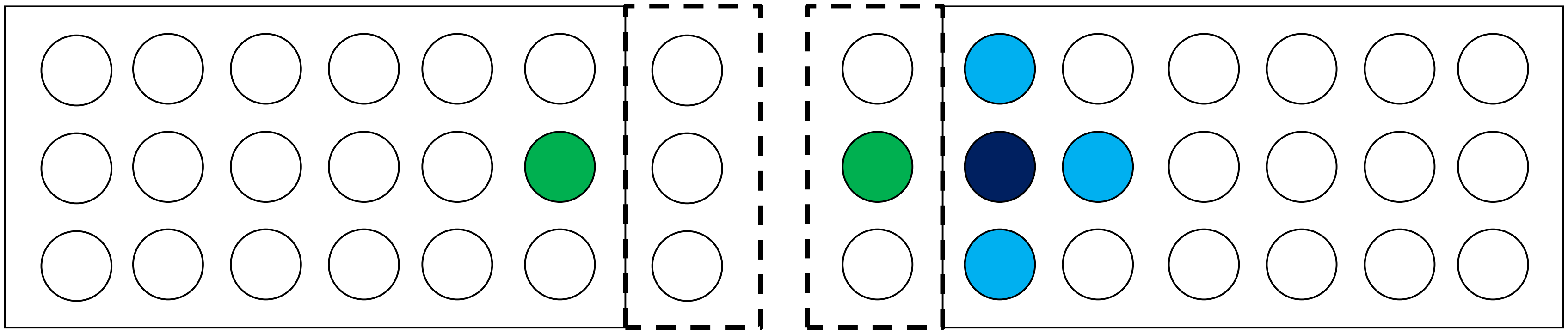
- Requested to process a point for t to t+1:
  - The point
  - Neighbours of that point
  - Extra, read-only points are needed on each side of the domain:
    - the ghosts



# Convolution Example

- Use of 2 processes for 1 domain

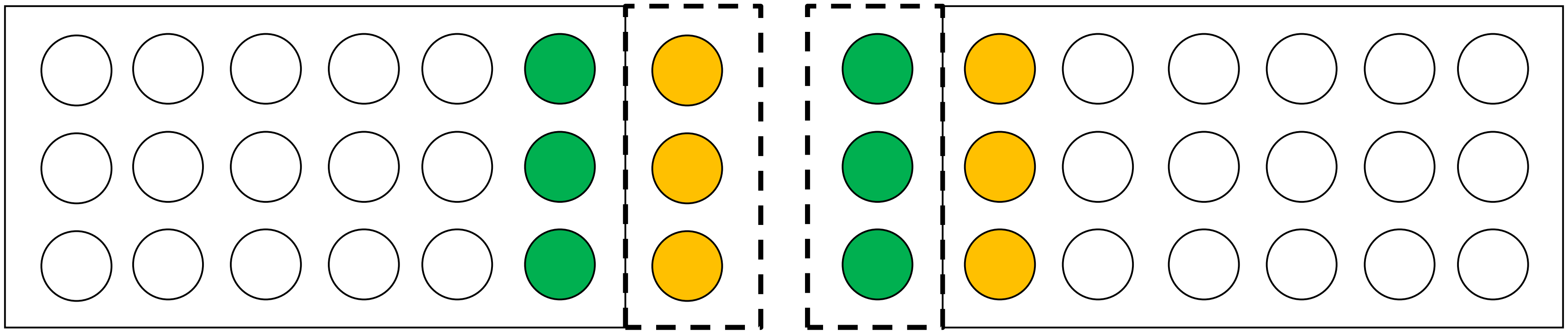
- Requested to process a point for t to t+1:
  - The point
  - Neighbours of that point
  - Extra, read-only points are needed on each side of the domain:
    - the ghosts



# Convolution Example

- Use of 2 processes for 1 domain

- Requested to process a point for t to t+1:
  - The point
  - Neighbours of that point
  - Extra, read-only points are needed on each side of the domain:
    - the ghosts

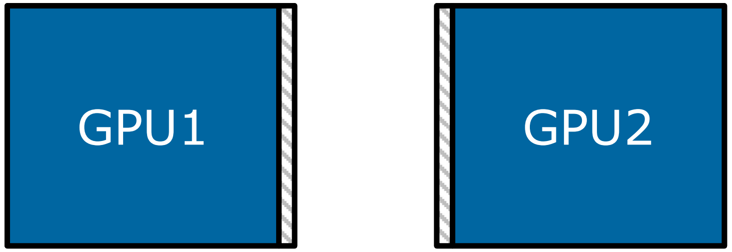


# Convolution Example

- Domain is to big for one GPU

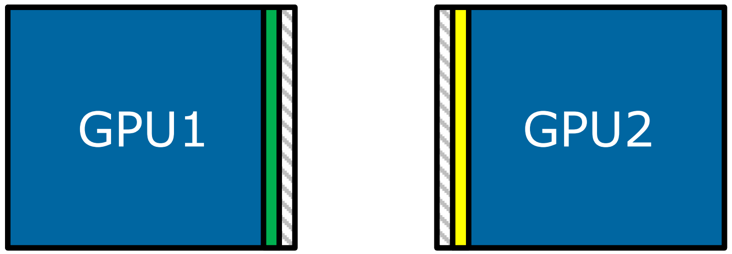
# Convolution Example

- Domain is to big for one GPU
- Split the domain into 2 sub-domains and use multiple GPUs



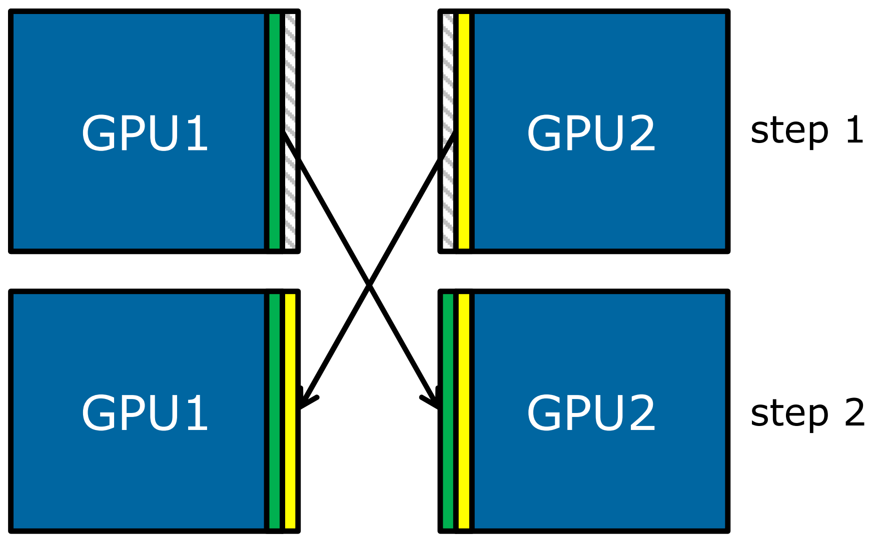
# Convolution Example

- Domain is to big for one GPU
- Split the domain into 2 sub-domains and use multiple GPUs
- In order to be able to compute the full convolution, partial data need to be shared between the GPUs



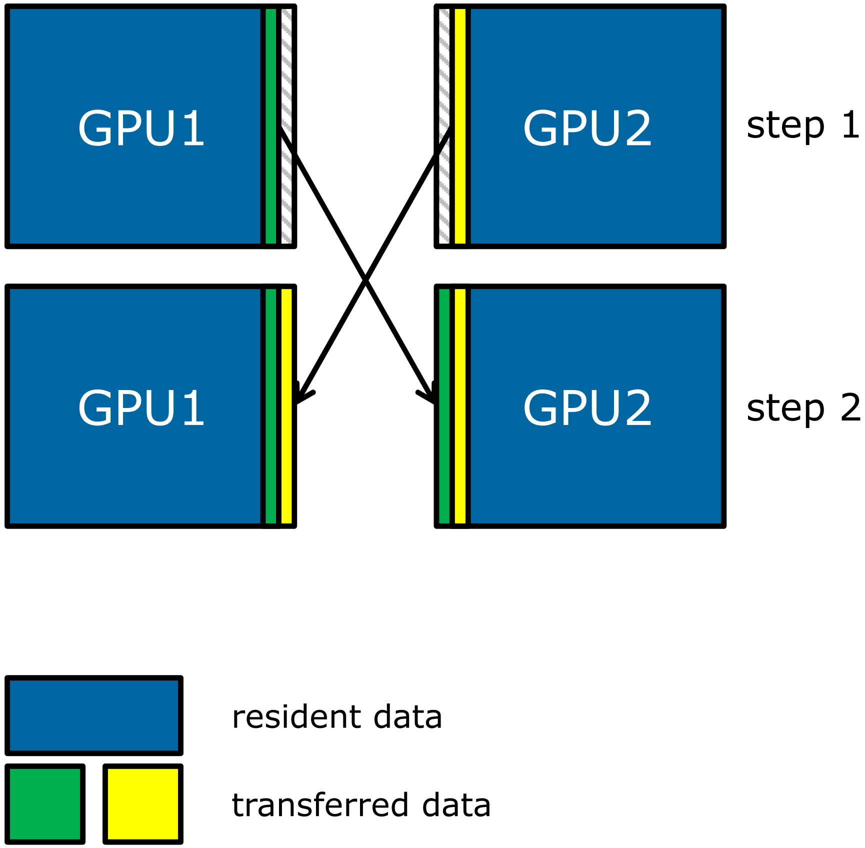
# Convolution Example

- Domain is to big for one GPU
- Split the domain into 2 sub-domains and use multiple GPUs
- In order to be able to compute the full convolution, partial data need to be shared between the GPUs
- At each step, the border of each domain is transferred to produce the ghosts of the neighboring domain



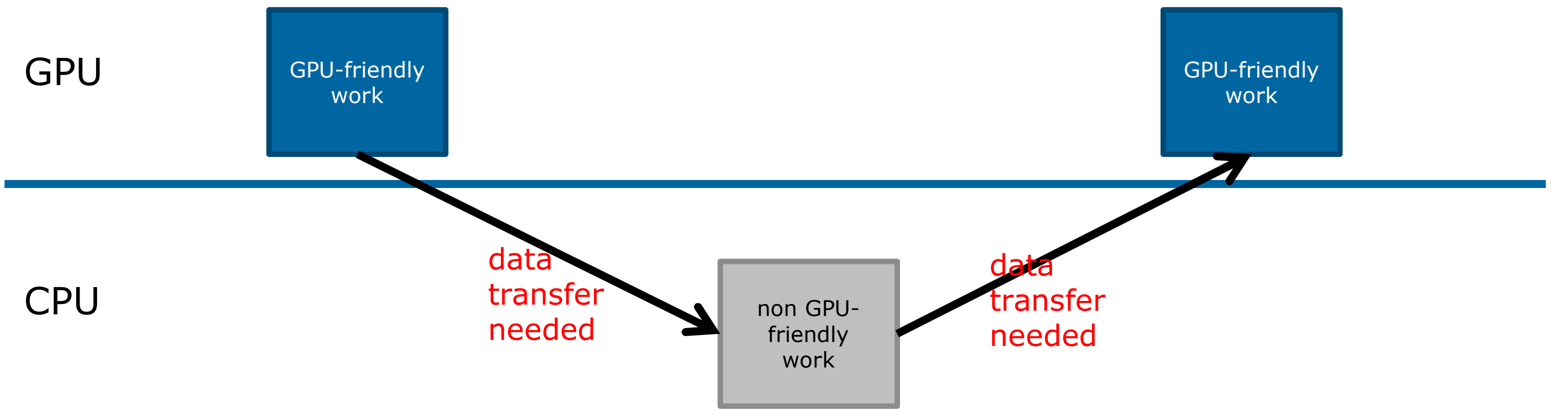
# Convolution Example

- Domain is to big for one GPU
- Split the domain into 2 sub-domains and use multiple GPUs
- In order to be able to compute the full convolution, partial data need to be shared between the GPUs
- At each step, the border of each domain is transferred to produce the ghosts of the neighboring domain



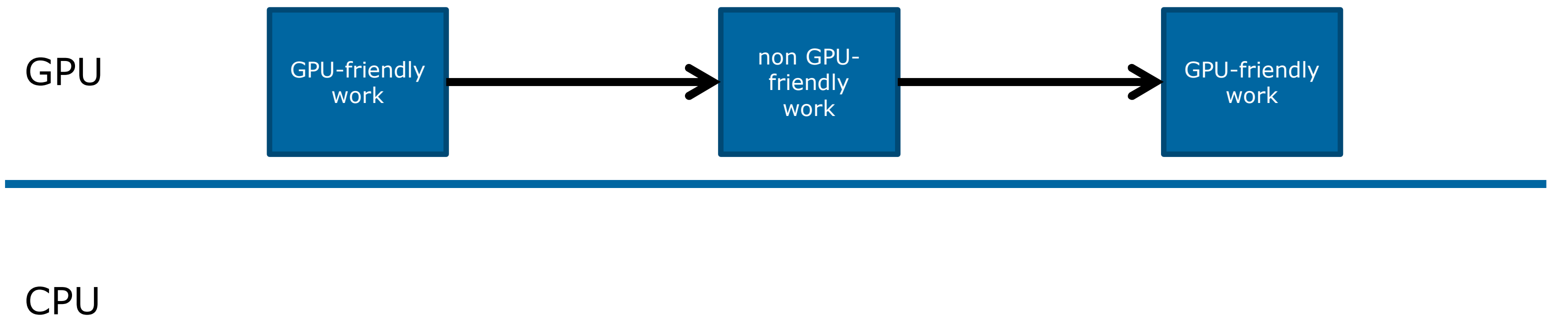
# Data Transfer Optimizations

- Port on GPU non compute intensive parts of the code if it can avoid transfer



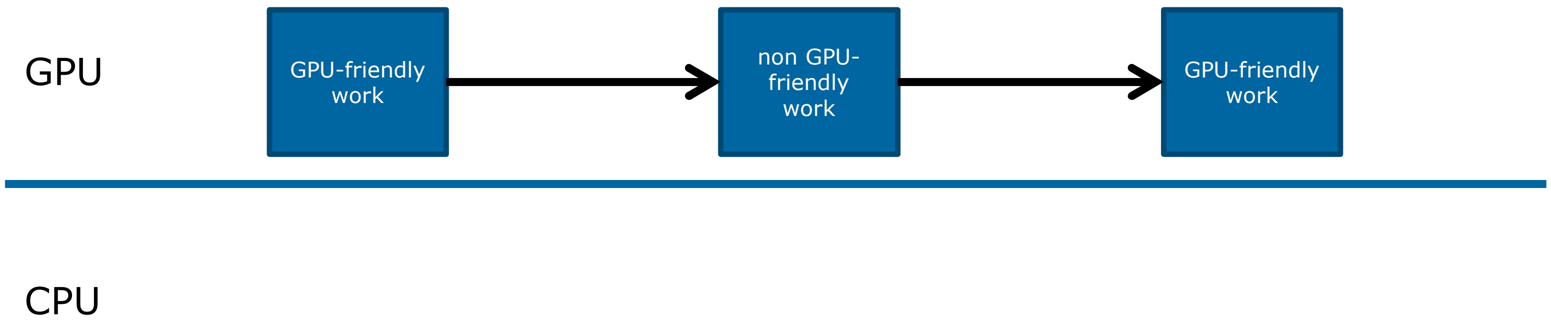
# Data Transfer Optimizations

- Port on GPU non compute intensive parts of the code if it can avoid transfer



# Data Transfer Optimizations

- Port on GPU non compute intensive parts of the code if it can avoid transfer




- Other optimizations: Functionalities
  - Use pinned memory
  - Overlap data transfers & kernel execution (streams)

# LAB: Transfers Optimization

- Copy nBody_a_square_omp.c in nBody_a_square.cu

- Port update_velocity on GPU:
  - allocate data on device
  - copy data on device
  - call a kernel to update data on device
  - copy data from device to host
  - free data on device

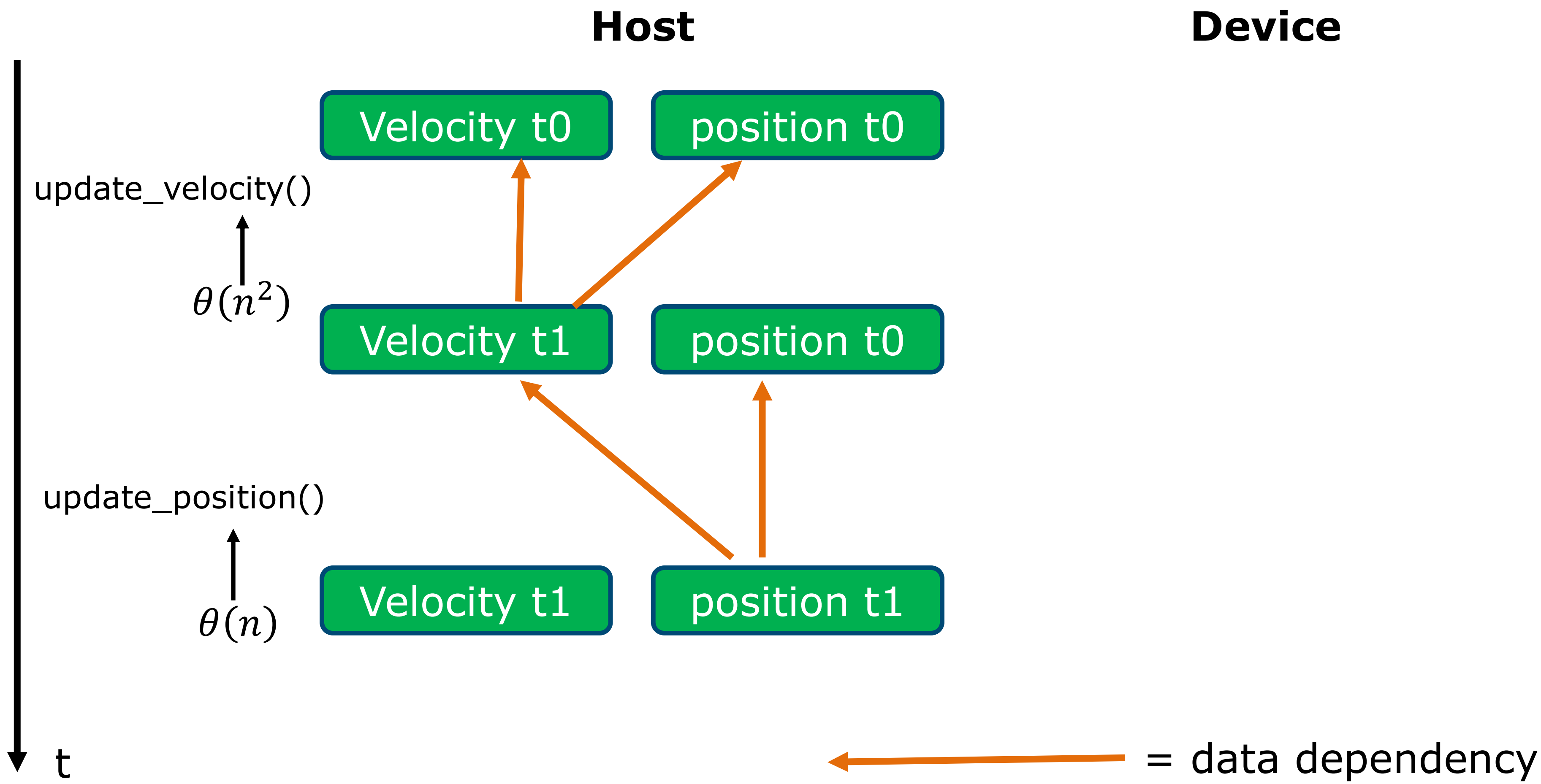
- Analyze data movement and then minimize data transfers

```
nvprof --export-profile v1_prof.nvvp  ./nbody_cuda_1.exe 1000 50
```

- Port update_position on GPU

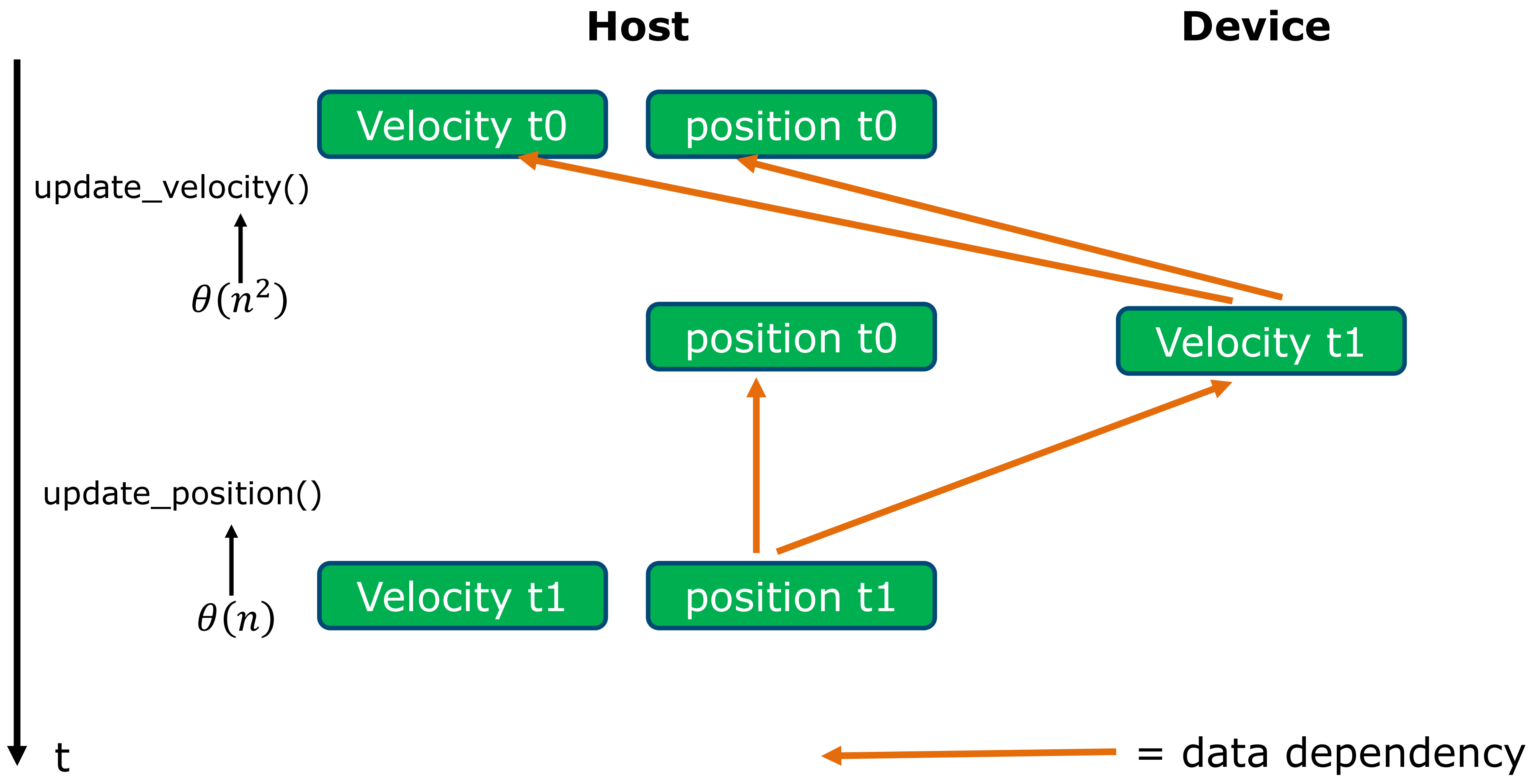
- Analyze data movement and then minimize data transfers

# LAB NBody



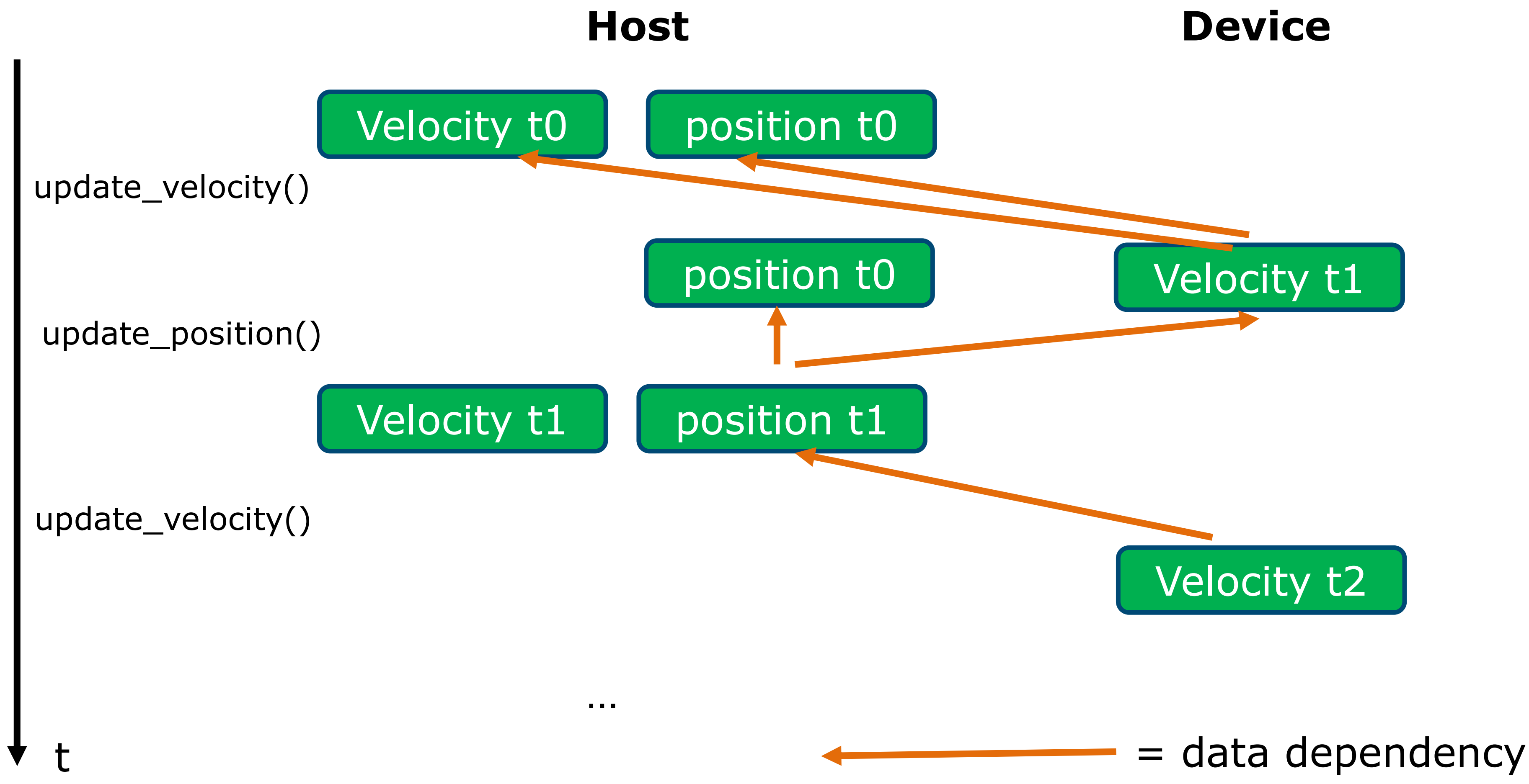
Host
Device
update_velocity()
$\theta(n^2)$
update_position()
$\theta(n)$
t
Velocity t0
position t0
Velocity t1
position t0
Velocity t1
position t1
= data dependency

# LAB NBody



Host
Device
Velocity t0
position t0
update_velocity()
$\theta(n^2)$
position t0
Velocity t1
update_position()
$\theta(n)$
Velocity t1
position t1
t
= data dependency

# LAB NBody



Host
Device
Velocity t0
position t0
update_velocity()
position t0
Velocity t1
update_position()
Velocity t1
position t1
update_velocity()
Velocity t2
...
t
= data dependency

# N Body – Performance figures 2000 bodies

| | CPU | | GPU |
|---|---|---|---|
| 1 | 20 | v1 | .12 |
| 2 | 10 | | |
| 4 | 5 | | |
| 8 | 3 | | |
| 16 | 1.6 | | |

# Access Issues

- Data need to be sent to and recovered from the GPU memory
- GPU memory is not directly accessible through the PCI Express bus
  - GPU-initiated transfers, not host-initiated
- But PCIe device can only DMA from physical address, not from the process virtual address
- No direct transfer between process and GPU through the PCI Express bus

# Data Transfers

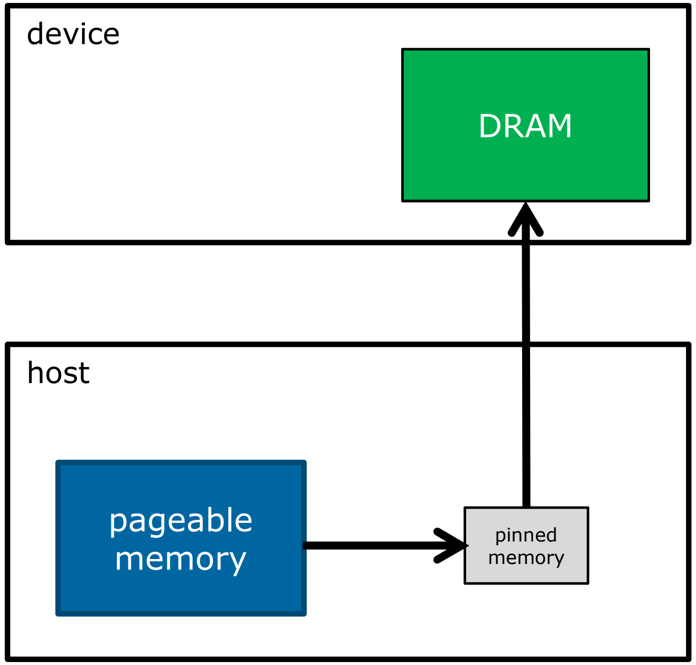
- Pageable Host Memory
  - Default allocation (e.g. malloc, calloc, new, etc)
  - Memory pages associated with the memory can be moved around by the OS Kernel, e.g. to swap space on hard disk

- When data transfer between **pageable host memory** and device memory is invoked, the driver:
  - allocate a temporary page-locked host array (probably just once)
  - copy the host data to the pinned array
  - transfer data from pinned array to device memory

- The **host is blocked** until the end of the transfer
  - prevent asynchronous operations

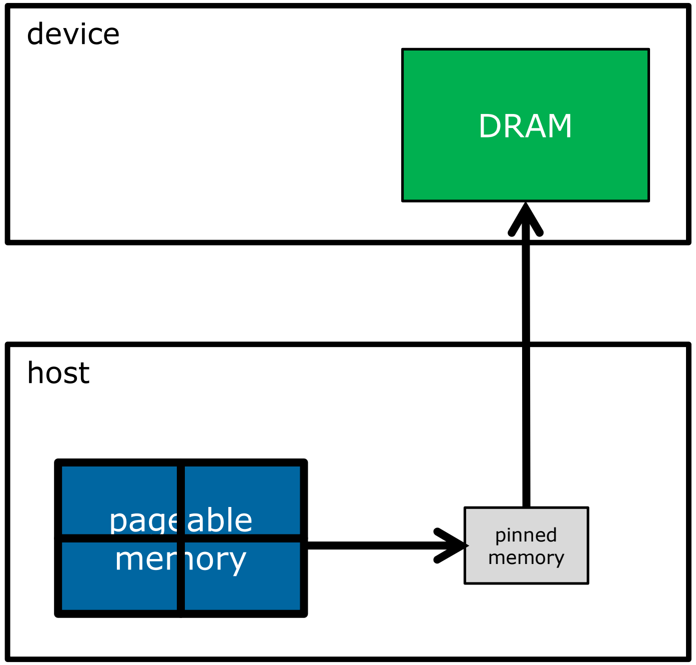
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



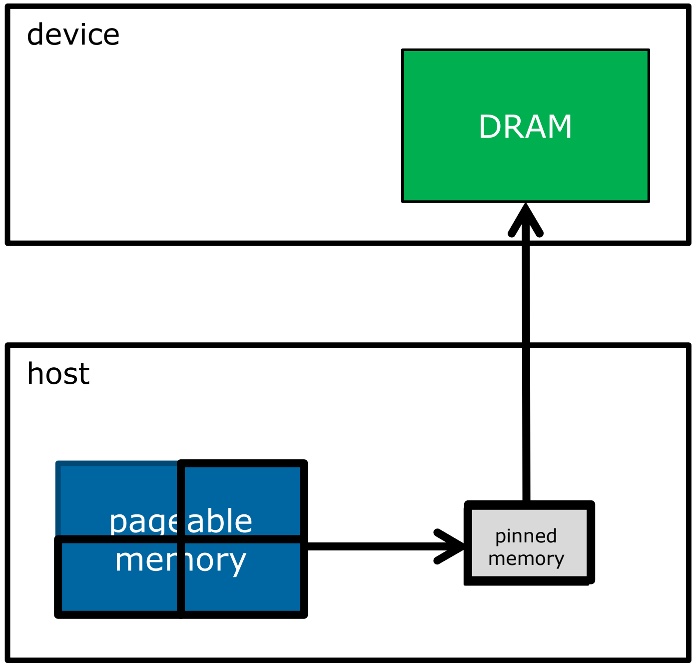
# Transfers with Pageable Memory

```
...
A=malloc(n*sizeof(int))
...
free(A)
```



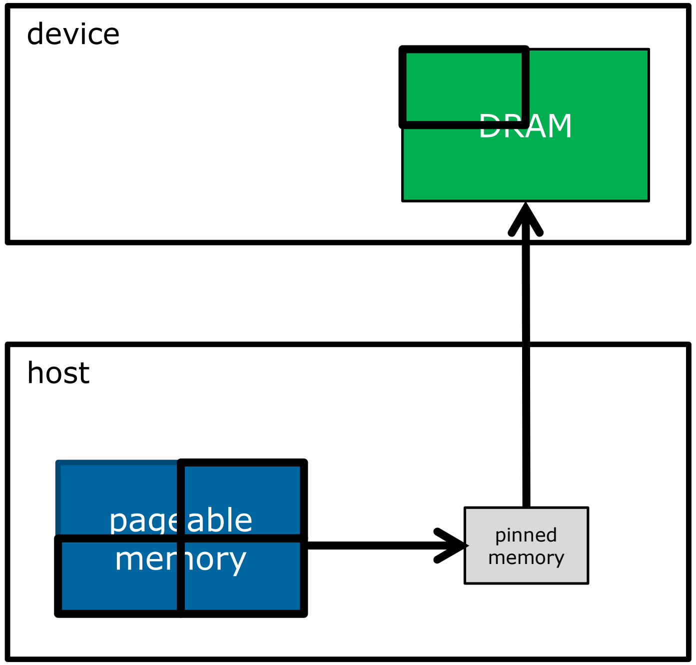
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



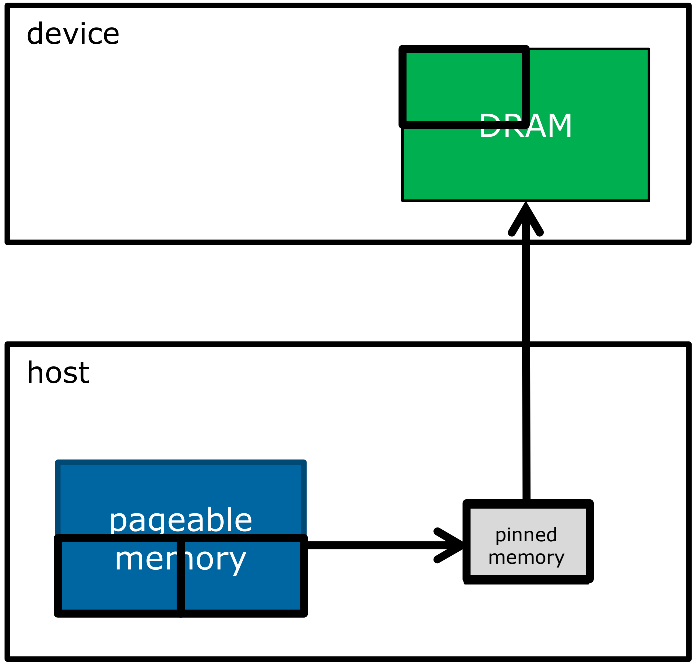
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



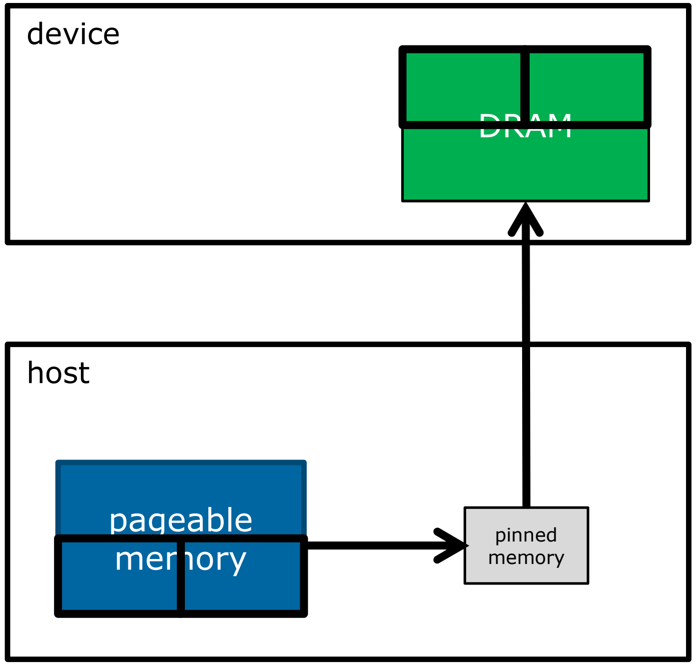
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



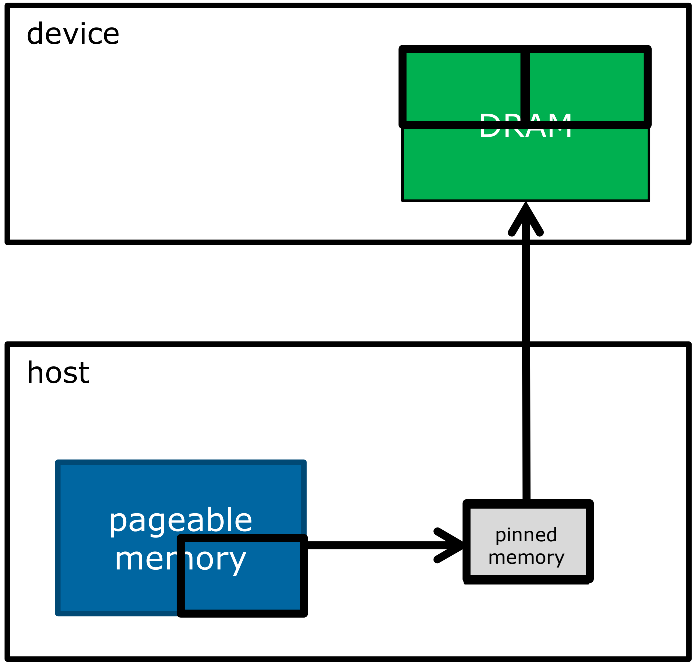
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



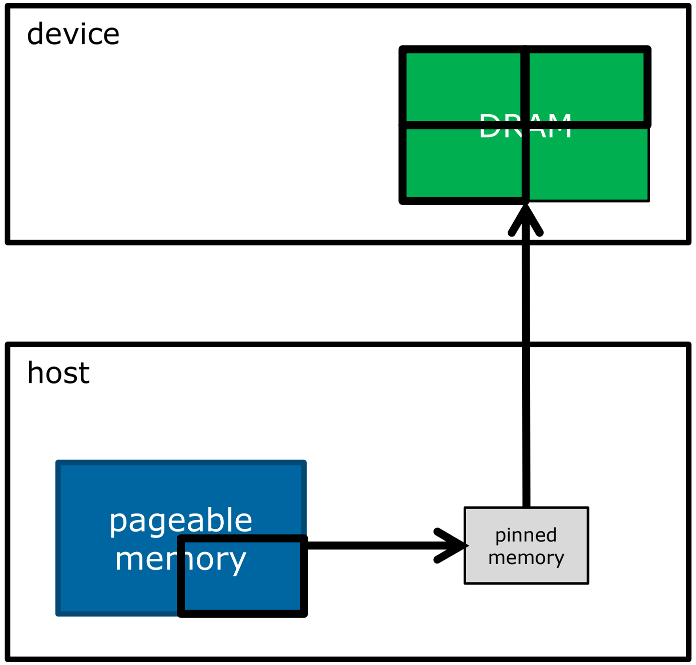
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



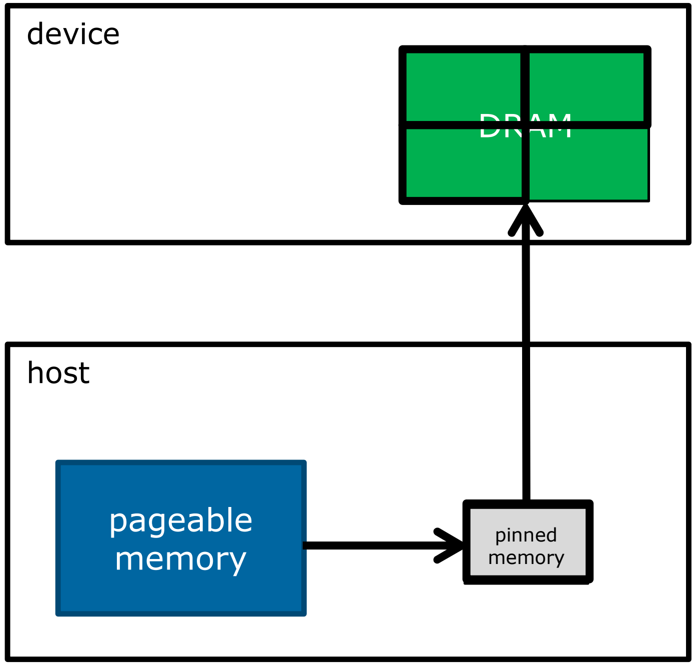
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



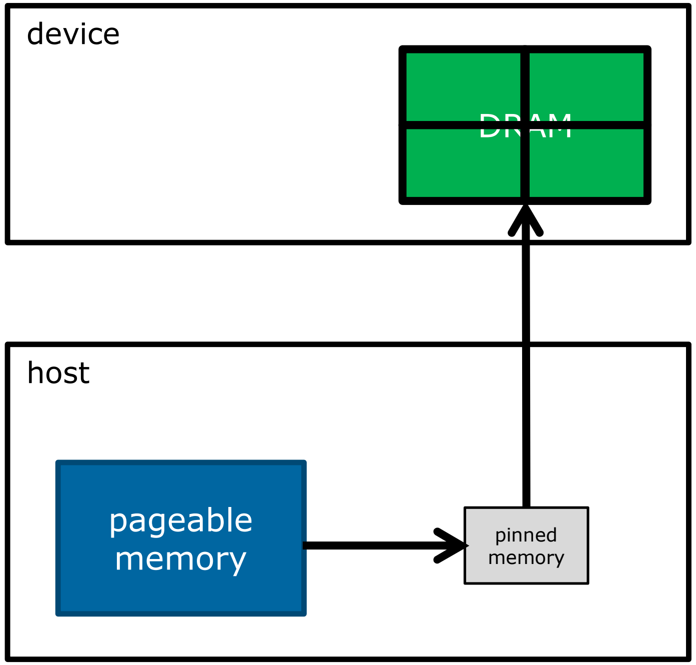
# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



# Transfers with Pageable Memory

```
…
A=malloc(n*sizeof(int))
…
free(A)
```



# Pinned Memory

- Pinned (Page-Locked) Host Memory
  - Allocated using special allocators
  - Cannot be paged out by the OS

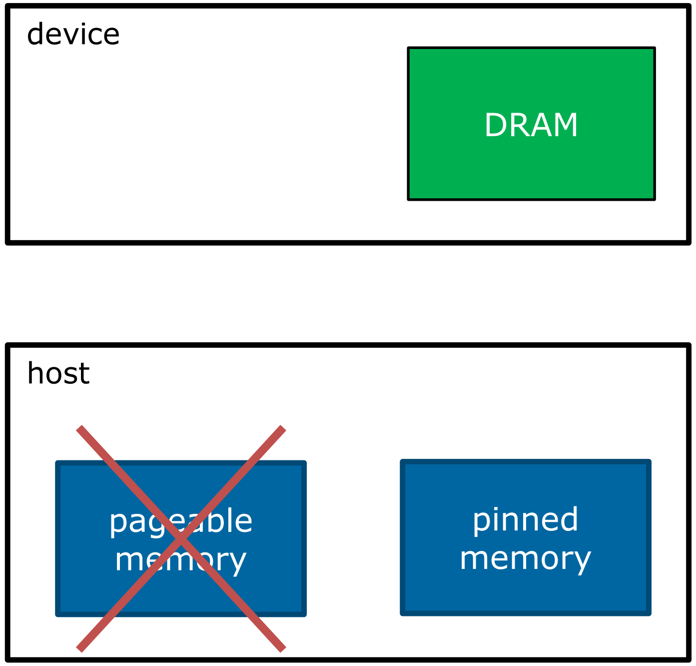
- Why pin memory?
  - Pageable memory is transferred using the host CPU
  - Pinned memory is **transferred using the DMA engines**
    - Achieves a higher percent of peak bandwidth
    - Frees the CPU for **asynchronous execution**

- When data transfer between pinned memory and device memory is invoked, the driver:
  - ~~allocate a temporary page-locked host array~~
  - ~~copy the host data to the pinned array~~
  - transfer data from pinned array to device memory

# Tranfers with Pinned Memory

```
A=malloc(n*sizeof(int))
…
free(A)
```

```
cudaMallocHost(&A, n*sizeof(int))
…
cudaFreeHost(A)
```

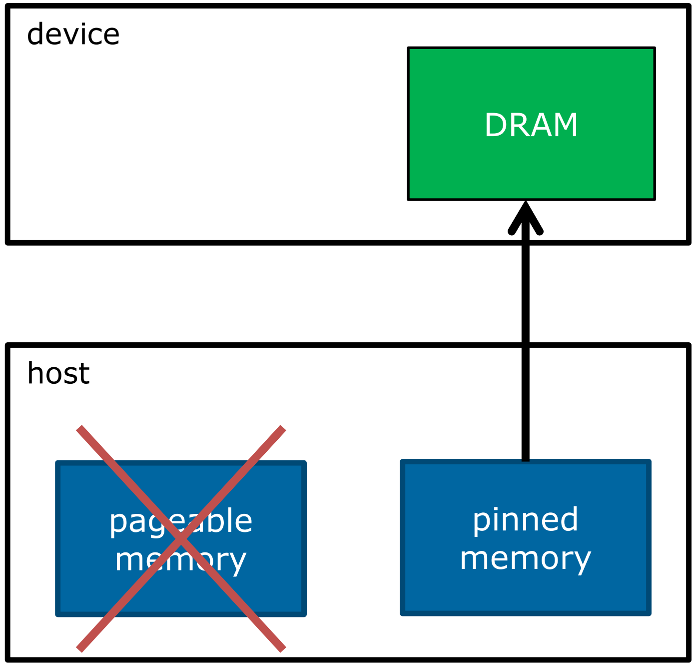


# Tranfers with Pinned Memory

~~A=malloc(n*sizeof(int))~~
~~…~~
~~free(A)~~

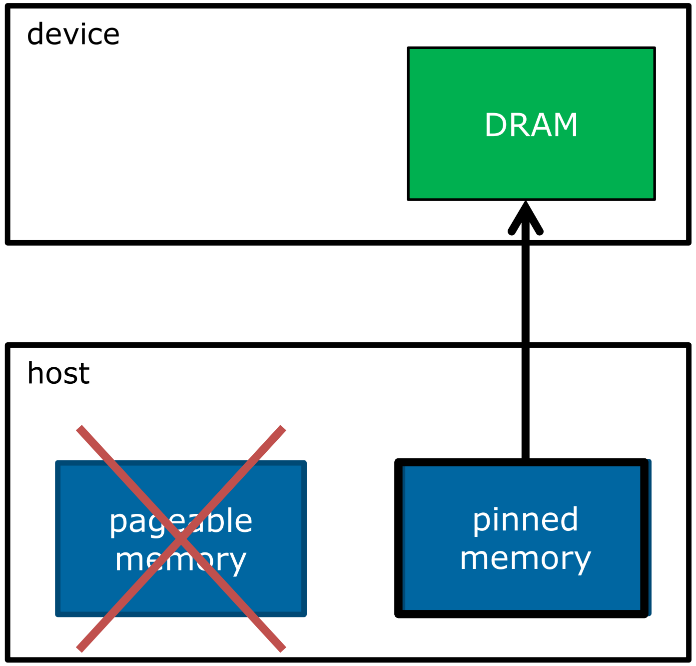
```
cudaMallocHost(&A, n*sizeof(int))
…
cudaFreeHost(A)
```



# Tranfers with Pinned Memory

~~A=malloc(n*sizeof(int))~~
~~…~~
~~free(A)~~

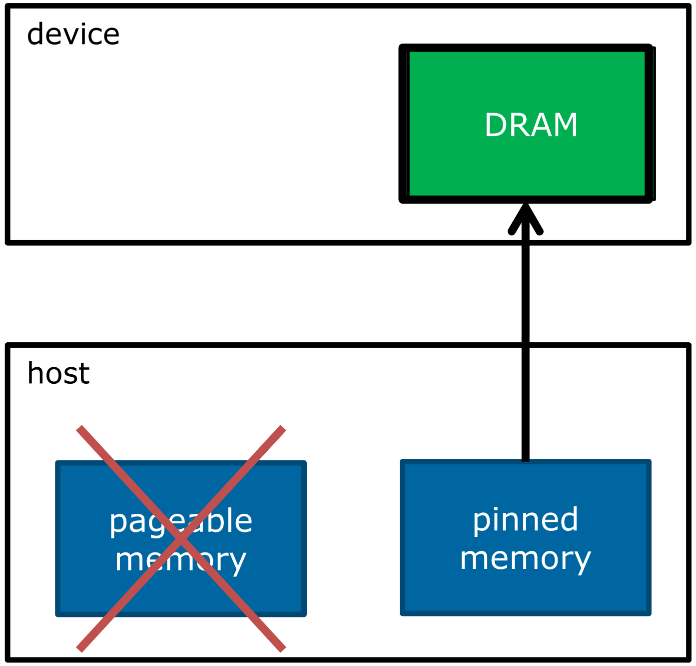
```
cudaMallocHost(&A, n*sizeof(int))
…
cudaFreeHost(A)
```



# Tranfers with Pinned Memory

~~A=malloc(n*sizeof(int))~~
~~…~~
~~free(A)~~

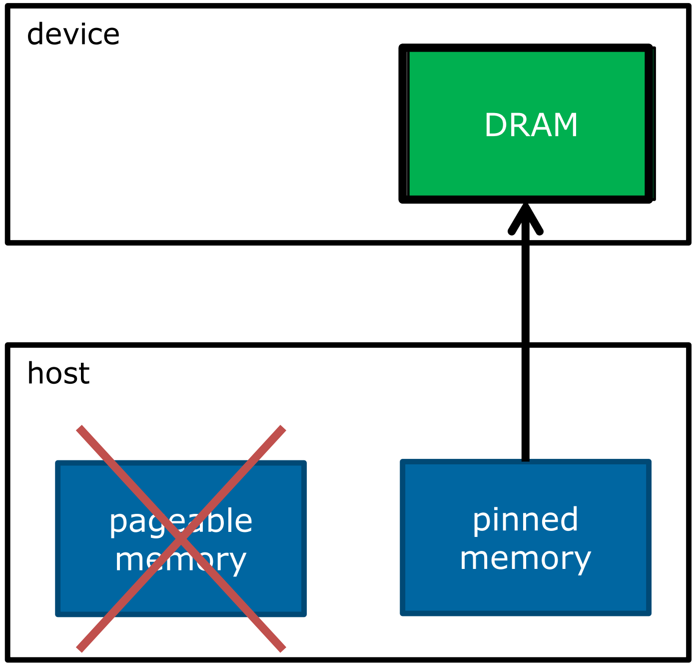
cudaMallocHost(&A, n*sizeof(int))
…
cudaFreeHost(A)



# Tranfers with Pinned Memory

```
A=malloc(n*sizeof(int))
…
free(A)
```

```
cudaMallocHost(&A, n*sizeof(int))
…
cudaFreeHost(A)
```




- **Use carefully!**
  - Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging

# Pinned Memory

- Using POSIX functions like **mlock** is not sufficient, because the CUDA driver needs to know that the memory is pinned

- **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

# LAB: Bandwidth Test

- Copy bandwidthTest.cu from the nvidia samples
  - #cp ${CUDA_HOME}/samples/1_Utilities/bandwidthTest/bandwidthTest.cu ./

- Compile :
  - #nvcc -I ${CUDA_HOME}/samples/common/inc ./bandwidthTest.cu -o ./bandwidthTest.exe

- Execute binaries using pageable memory then pinned memory:
  - #./bandwidthTest.exe

# cudaHostAlloc

```
cudaError_t cudaHostAlloc ( void **  ptr, size_t size, unsigned int flags )
```

- Flags:
  - **cudaHostAllocDefault**: emulate cudaMallocHost()
  - **cudaHostAllocPortable**: Considered pinned memory by all CUDA contexts
  - **cudaHostAllocMapped**: Maps the allocation into the CUDA address space. Device pointer to the memory may be obtained by calling cudaHostGetDevicePointer()
  - **cudaHostAllocWriteCombined**: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.
- All of these flags are orthogonal to one another
  - memory can be portable, mapped and/or write-combined with no restrictions

# Mapped Pinned Memory / Zero-copy

- The GPU can directly access the mapped pinned memory from kernels

- Which pointer to use in the kernel? The one return by cudaHostAlloc?

- Depends of two  device properties:
  - unifiedAddressing
  - canUseHostPointerForRegisteredMem

- To check these attributes:

```
cudaGetDeviceProperties(struct cudaDeviceProp * prop, int device)
```

# Mapped Pinned Memory / Zero-copy

- First check **unifiedAddressing**
  - if 1**:** the device can use the pointer return by cudaHostAlloc
  - else:

- Check **canUseHostPointerForRegisteredMem**
  - if 1: the device can use the pointer return by cudaHostAlloc
  - else call:

> cudaHostGetDevicePointer*(void** pDevice, void* pHost, unsigned int flags)*
>
> pDevice: Returned device pointer for mapped memory
>
> pHost: Requested host pointer mapping
>
> flags: Flags for extensions (must be 0 for now)

- **cudaHostGetDevicePointer is more portable**:
  - if unifiedAddressing or canUseHostPointerForRegisteredMem equal 1:
    - pDevice is the same as pHost

# Mapped Pinned Memory / Zero-copy

- Mapped Pinned memory (Zero-copy) is useful when either:
  - GPU has no memory on its own and uses RAM
  - Host needs to change/add data or read results, while kernel is still running
  - Data does not fit into GPU memory
  - You load data exactly once & you have a lot of computation to perform on it
    - Zero-copy can hide memory transfer latencies (no streams necessary)

- Pinned, but not mapped memory is better when:
  - Device loads/stores data multiple times
  - Application is already memory bound

# LAB: Mapped Memory

- Complete main.cu

# Recap - memory

- **Physical CUDA memory**
  - Global Memory (device memory)
  - Constant / texture memory
  - Shared memory
  - Register memory

- Logical CUDA memory
  - Local memory

- **Host-side memory**
  - Pageable memory (malloc)
  - Pinned memory (cudaMallocHost)
  - Mapped memory (cudaHostAlloc)

- Memory concepts (driver handled)
  - Mapped memory
  - *Managed memory*

# Concurrent Execution

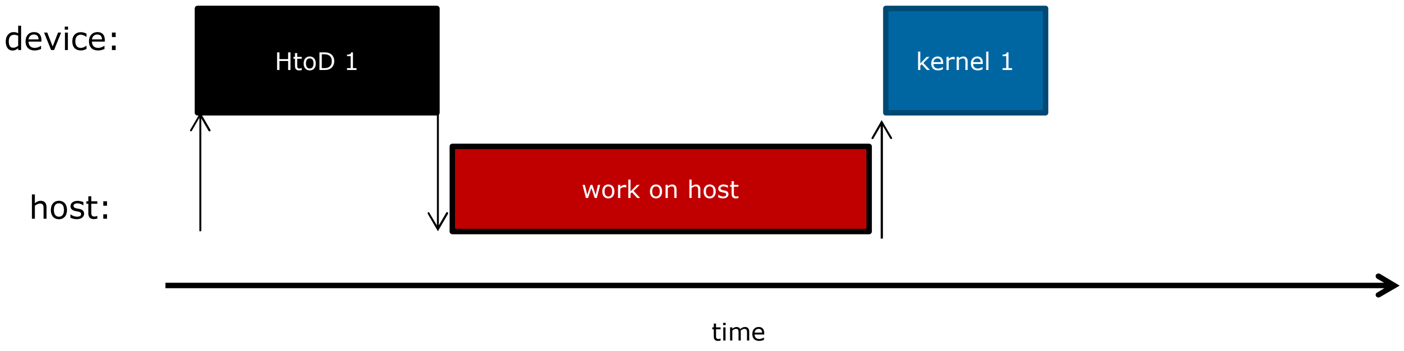
16/09/2019

# Concurrent Execution between Host and Kernels

- Execution is always launched **from CPU** (offload)

```
cudaMemcpy( ... , cudaMemcpyHostToDevice);

do work on host

kernel1 <<<100, 512 >>>(...)
```

# Concurrent Execution between Host and Device



# Concurrent Execution between Host and Kernels

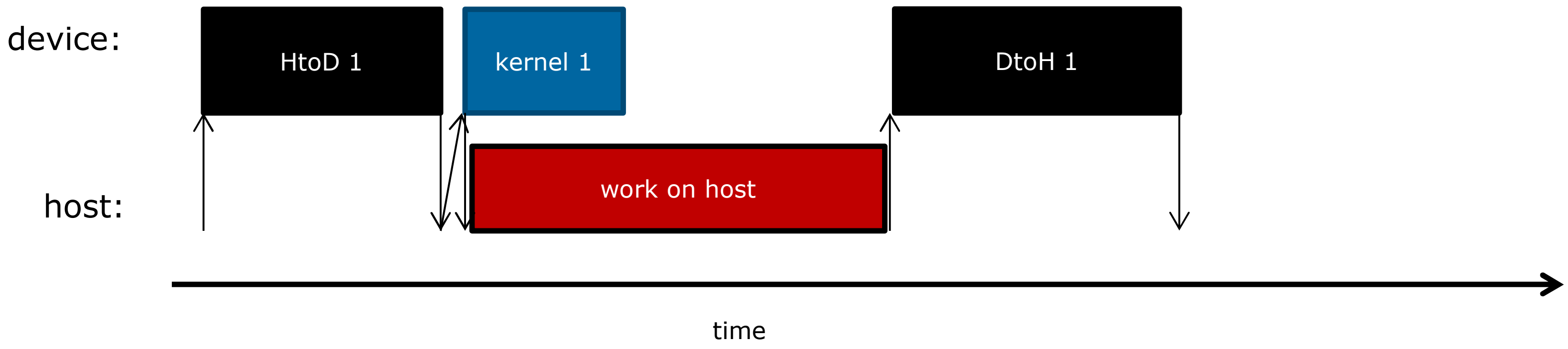
- Execution is always launched **from CPU** (offload)

- CPU continues its execution while a **kernel** is running on GPU
  - operations on the GPU are synchronous

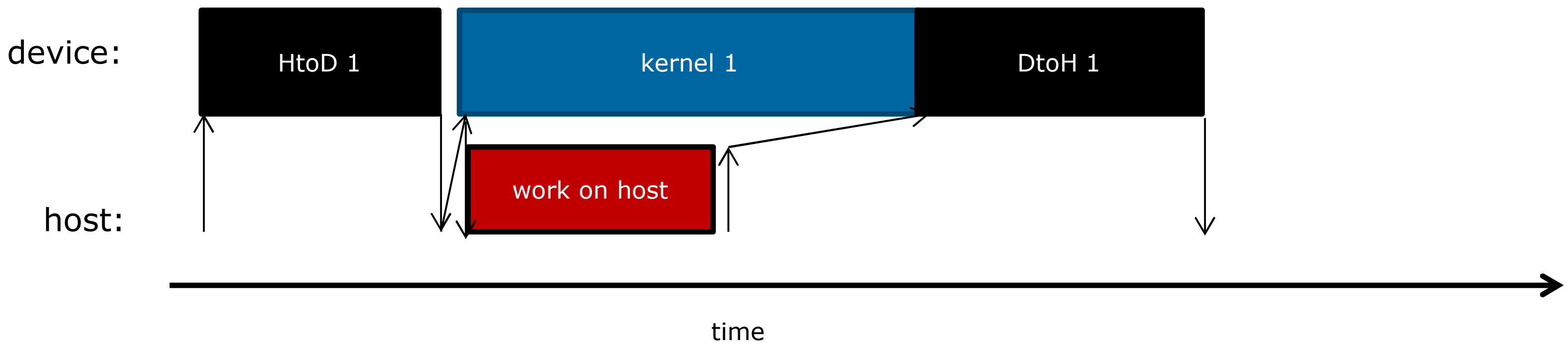
```
cudaMemcpy( ... , cudaMemcpyHostToDevice);

kernel1 <<<100, 512 >>>(…)

do work on host

cudaMemcpy( ... , cudaMemcpyDeviceToHost );
```

# Concurrent Execution between Host and Device

- 2 possibilities:




  - or



# Asynchronous Concurrent Execution

- Concurrent Execution between Host and Device:
  - Kernel launches
  - Memory copies within a single device's memory
  - Memory copies from host to device of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async
    - require page locked memory on the host
  - Memory set function calls
- Return control to the host thread before the device completes the requested task.

# LAB: Concurrent Execution

- Complete the file *concurrent.cu in CUDA/TP_CUDA_X/STREAMS_EVENTS*
- Create a profile trace with nvprof: « nvprof --export-profile profile.nvvp ./bin »
- Read the profile using nvvp

salloc -t 03:30:00 -p CSL-6248_GPU_hdr100_192gb_2933 --gres=gpu:1
ssh –X spartanXXX
source /software/load_me.sh
module load cuda/10.2
nvvp &

# LAB: Concurrent Execution

```
size = 100000000;

cudaMemcpy(d_i, h_i, size * sizeof(int), cudaMemcpyHostToDevice);

double t0 = wallclock();

cudaMemcpy(d_c, h_c, 10*sizeof(int), cudaMemcpyHostToDevice);

double t1 = wallclock();

cudaMemcpy(d_a, d_b, size*sizeof(int), cudaMemcpyDeviceToDevice);

double t2 = wallclock();

myKernel<<< numBlocks, threadsPerBlock >>>(size, d_i);

double t3  = wallclock();

cudaMemcpy(h_i, d_i, size * sizeof(int), cudaMemcpyDeviceToHost);

double t4  = wallclock();
```

# Asynchronous Concurrent Execution

- But the device allows to:
  - run several kernels concurrently
  - perform copies while kernels are running
    - require page locked memory on the host
  - perform a device to host copy in parallel of a host to device copy
    - require page locked memory on the host

- How to use concurrent execution on GPU?

# STREAMS

16/09/2019

# CUDA Streams

- A **stream** is a queue of work
  - The host places work (operation) 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** and **cannot overlap**
  - executed sequentially within the stream (FIFO)

- Operations in **different streams** are **unordered** and **can overlap**
  - no specific order between operations from different streams

# Managing Streams

- **cudaStream_t** stream;
  - Declares a stream handle

- **cudaStreamCreate**(cudaStream_t *stream);
  - Allocates a stream

- **cudaStreamDestroy**(cudaStream_t stream);
  - Deallocates a stream
  - No synchronization with the host:
    - In case the device is still doing work in the stream the function will return immediately.

# cudaMemcpyAsync

- **cudaMemcpyAsync()** is asynchronous with respect to the host
  - call may return before the copy is complete

```
cudaError_t cudaMemcpyAsync(    void *dst, const void *src, size_t count,
                                enum cudaMemcpyKind kind, cudaStream_t stream)
```

- **Only works on page-locked host memory**

- Copy can be associated to a stream (non-zero stream argument)

- If the stream is non-zero:
  - cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost
    - copy may overlap with operations in other streams
  - cudaMemcpyDeviceToDevice:
    - asynchronous with respect to the host
    - never overlap with kernel execution.

# Kernel Launch in a Stream

- The execution configuration: <<< Dg, Db, Ns, **S** >>>

- dim3 Dg:
  - the number of blocks being launched

- dim3 Db:
  - the number of threads per block

- size_t Ns:
  - number of bytes in shared memory that is dynamically allocated per block
    - in addition to the statically allocated memory
    - optional argument (which defaults to)

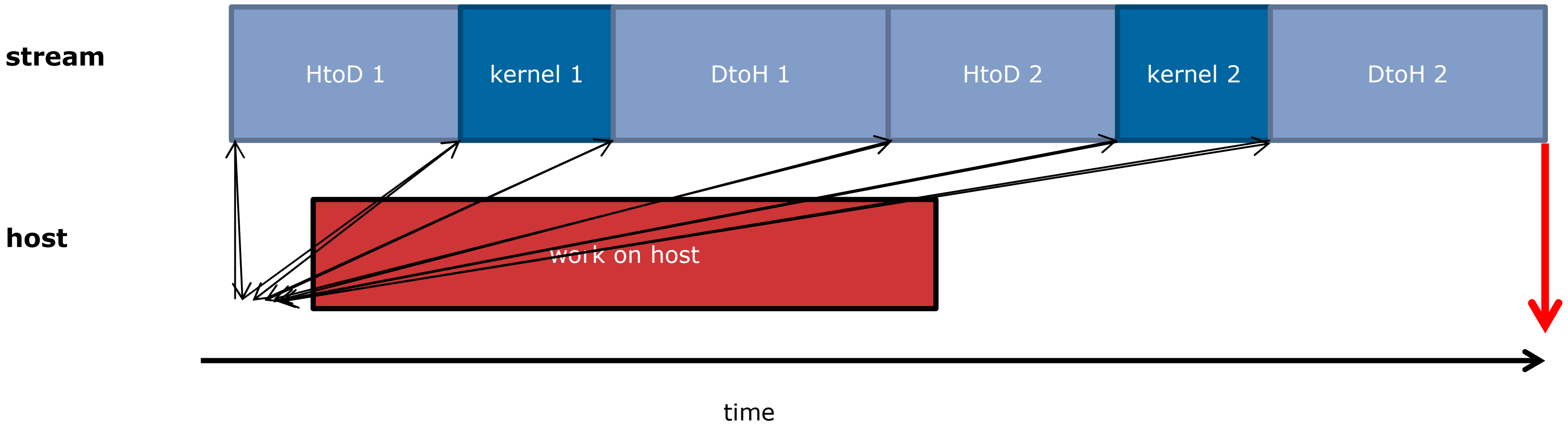
- cudaStream_t S:
  - specifies an associated stream
    - an optional argument which defaults to 0.

# Example: 1 stream

```
cudaStream_t stream [1];

//stream creation
cudaStreamCreate (& stream[0]);

float * hostPtr ;
cudaMallocHost (& hostPtr , 2 * size ); //page-locked memory allocation

cudaMemcpyAsync ( … , cudaMemcpyHostToDevice , stream[0]);
kernel1 <<<100, 512 , 0, stream[0]>>>(…)
cudaMemcpyAsync ( … , cudaMemcpyDeviceToHost , stream[0]);

cudaMemcpyAsync ( … , cudaMemcpyHostToDevice , stream[0]);
kernel2 <<<100, 512 , 0, stream[0]>>>(…)
cudaMemcpyAsync ( … , cudaMemcpyDeviceToHost , stream[0]);

do work on host

…
cudaDeviceSynchronize();
//stream destruction
cudaStreamDestroy (stream[0]);
```

# Example: 1 stream

▶ Concurrent execution between host and device (stream):



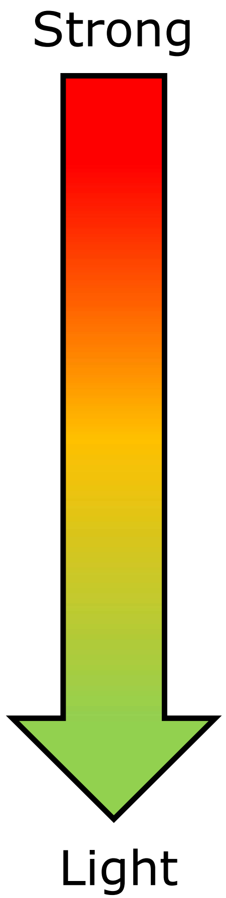

▶ User need mechanisms to synchronize host and streams

# How to Synchronize Streams

- **cudaDeviceSynchronize()**
  - Synchronize everything
    - Blocks host until all issued CUDA calls are complete

- **cudaStreamSynchronize(stream)**
  - Synchronize host with regard to a specific stream
    - Blocks host until all issued CUDA calls in stream are complete

- Synchronize host or devices using **events**
  - allow to synchronize several streams



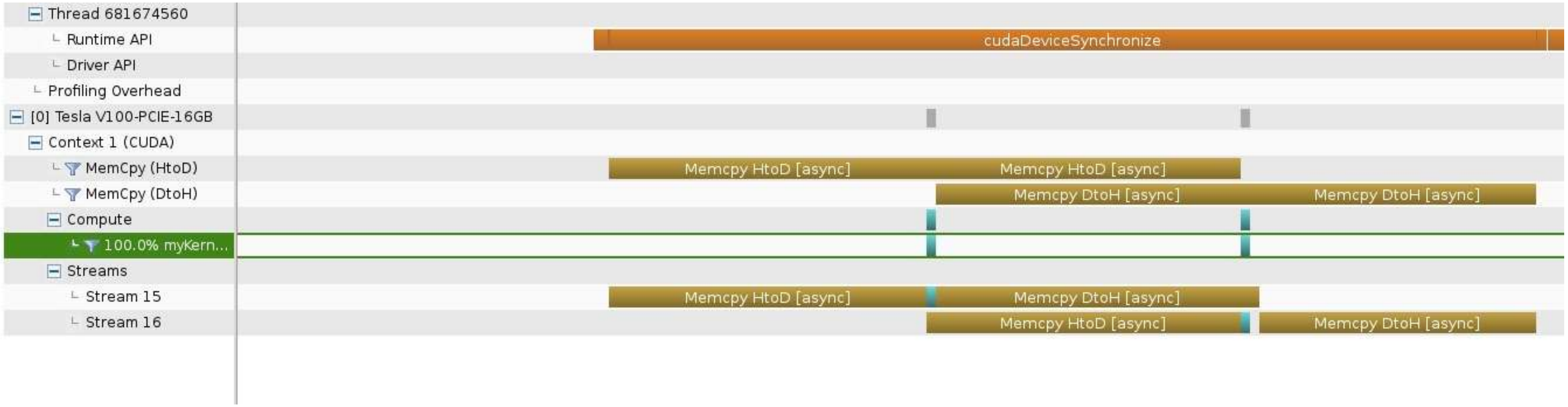
# LAB: Two Streams

- Use two different streams to perform copies and kernel launches:
  - one stream to:
    - copy h_array1 in d_array1 asynchronously
    - launch myKernel on d_array1
    - copy d_array1 in h_array1 asynchronously
  - an other stream to:
    - copy h_array2 in d_array2 asynchronously
    - launch myKernel on d_array2
    - copy d_array2 in h_array2 asynchronously
- Create a profile trave with nvprof: « nvprof --export-profile profile.nvvp ./bin »
- Read the profile using nvvp

# Example: 2 concurrent streams

```
cudaStream_t stream[2];
//streams creation
for ( int i = 0; i < 2; ++i)
  cudaStreamCreate (& stream[i]);

float * hostPtr ;
cudaMallocHost (...); //page-locked memory allocation

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream[0]);
kernel1 <<<100, 512 , 0, stream[0]>>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream[0]);

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream[1]);
kernel2 <<<100, 512 , 0, stream[1]>>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream[1]);

cudaDeviceSynchronize();

//streams destruction
for ( int i = 0; i < 2; ++i)
  cudaStreamDestroy (stream [i]);
```

# Example: 2 streams



# LAB: Stream Synchronization

- Compile two_stream.cu
  - generate a trace with nvprof and look at it with nvvp

- Add a call to *cudaDeviceSynchronize* before t1
  - compile
  - generate a trace with nvprof and look at it with nvvp

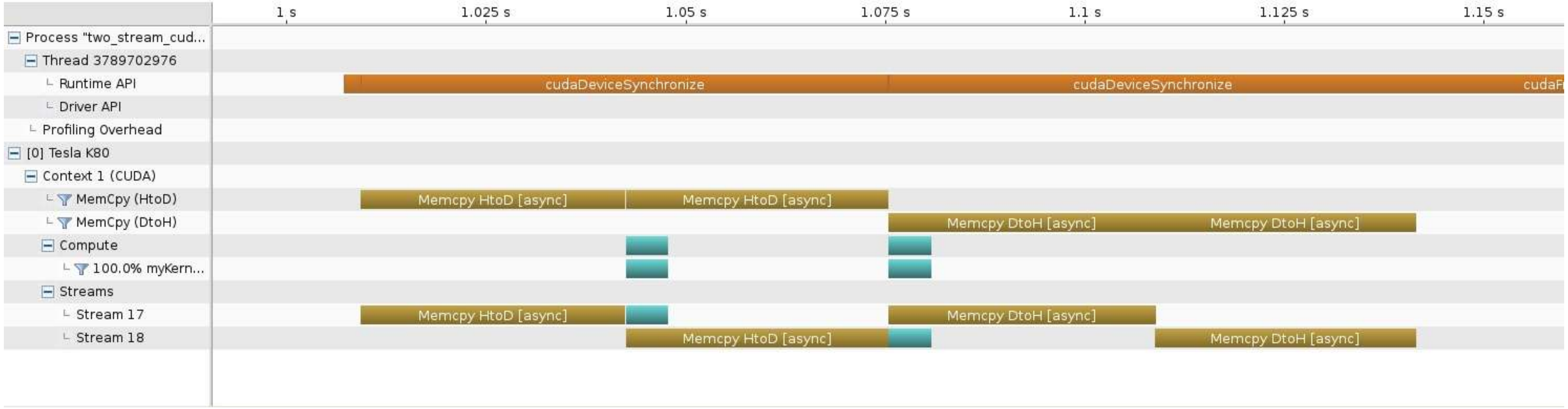
- Comment the call to *cudaDeviceSynchronize*

- Add a call to *cudaStreamSynchronize* to synchronize *stream[0]* before t1
  - compile
  - generate a trace with nvprof and look at it with nvvp

# Example: 2 concurrent streams + cudaDeviceSynchronize

```
cudaStream_t stream[2];
//streams creation
for ( int i = 0; i < 2; ++i)
  cudaStreamCreate (& stream[i]);

float * hostPtr ;
cudaMallocHost(...); //page-locked memory allocation

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream[0]);
cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream[1]);

kernel1 <<<100, 512 , 0, stream[0]>>>(...)
cudaDeviceSynchronize();
kernel2 <<<100, 512 , 0, stream[1]>>>(...)

cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream[0]);
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream[1]);

cudaDeviceSynchronize();

//streams destruction
for ( int i = 0; i < 2; ++i)
  cudaStreamDestroy (stream [i]);
```

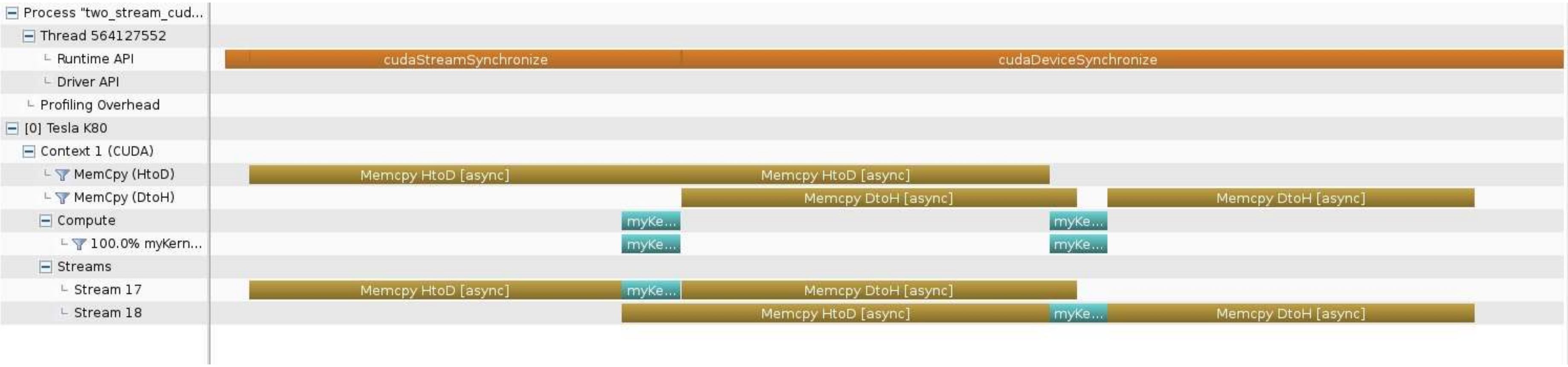
# 2 concurrent streams + cudaDeviceSynchronize



# Example: 2 concurrent streams + cudaStreamSynchronize

```
cudaStream_t stream[2];
//streams creation
for ( int i = 0; i < 2; ++i)
  cudaStreamCreate (& stream[i]);

float * hostPtr ;
cudaMallocHost(...); //page-locked memory allocation

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream[0]);
cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream[1]);

kernel1 <<<100, 512 , 0, stream[0]>>>(...)
cudaStreamSynchronize(stream[0]);
kernel2 <<<100, 512 , 0, stream[1]>>>(...)

cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream[0]);
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream[1]);

cudaDeviceSynchronize();

//streams destruction
for ( int i = 0; i < 2; ++i)
  cudaStreamDestroy (stream [i]);
```

# 2 concurrent streams + cudaStreamSynchronize



# Legacy Default Stream

- Unless otherwise specified all calls are placed into a **default stream**
  - Often referred to as "**Stream 0 or NULL**"

- 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**)
  - Use to get concurrency with libraries out of your control (e.g. MPI)

# Per Thread Default Stream

- The synchronization behavior of the default thread can be changed from legacy to per-thread
- When using per-thread default stream, the default stream is an implicit stream which does not synchronize with other streams
  - like explicitly created streams
- The behavior can be controlled per compilation unit with the --default-stream nvcc option:
  - --default-stream legacy (default) or --default-stream per-thread
- Alternatively, per-thread behavior can be enabled by defining the CUDA_API_PER_THREAD_DEFAULT_STREAM macro before including any CUDA headers.
- Either way, the CUDA_API_PER_THREAD_DEFAULT_STREAM macro will be defined in compilation units using per-thread synchronization behavior. (-D)

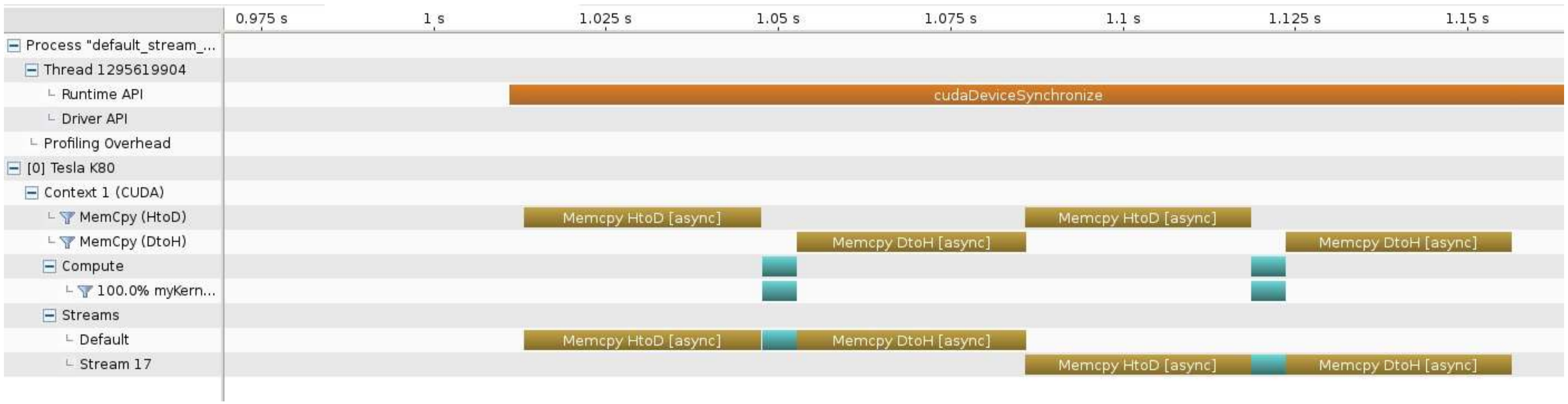
# LAB: Default Stream

- Use two different streams to perform copies and kernel launches:
  - **the 0 or NULL stream to**:
    - copy h_array1 in d_array1 asynchronously
    - launch myKernel on d_array1
    - copy d_array1 in h_array1 asynchronously
  - **an other stream to**:
    - copy h_array2 in d_array2 asynchronously
    - launch myKernel on d_array2
    - copy d_array2 in h_array2 asynchronously
- Create a profile trace with nvprof: « nvprof --export-profile profile.nvvp ./bin »
- Read the profile using nvvp

# Example: legacy default stream + another stream

```
cudaStream_t mystream;
//streams creation
cudaStreamCreate (&mystream);

float * hostPtr ;
cudaMallocHost (...); //page-locked memory allocation

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice, 0 );
kernel1 <<<100, 512, 0, 0 >>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost, 0 );

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , mystream);
kernel2 <<<100, 512 , 0, mystream>>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , mystream);

cudaDeviceSynchronize();

//streams destruction
cudaStreamDestroy (mystream);
```

# Example: legacy default stream + another stream



# Example: legacy default stream + another stream


cudaMemcpyAsync ( … , cudaMemcpyHostToDevice, 0 );
cudaMemcpyAsync ( … , cudaMemcpyHostToDevice , **mystream**);

kernel1 <<<100, 512, 0, 0 >>>(...)
kernel2 <<<100, 512 , 0, **mystream**>>>(...)

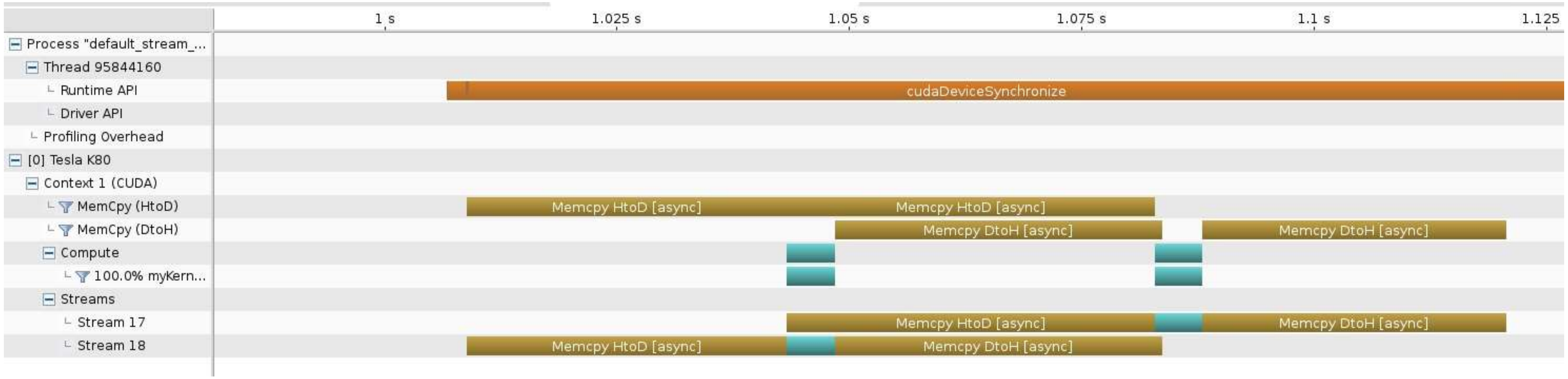
cudaMemcpyAsync ( … , cudaMemcpyDeviceToHost, 0 );
cudaMemcpyAsync ( … , cudaMemcpyDeviceToHost , **mystream**);

# LAB: Default Stream

- Compile the previous file with --default-stream per-thread
- Create a profile trace with nvprof: « nvprof --export-profile profile.nvvp ./bin »
- Read the profile using nvvp

# Example: per thread default stream + another stream



# LAB: Default Stream

- Use two different streams to perform copies and kernel launches:

  - the 0 or NULL stream to:
    - copy h_array1 in d_array1 asynchronously
    - launch myKernel on d_array1
    - copy d_array1 in h_array1 asynchronously

  - an other stream created with **cudaStreamCreateWithFlags** with the flag ***cudaStreamNonBlocking*** to:
    - copy h_array2 in d_array2 asynchronously
    - launch myKernel on d_array2
    - copy d_array2 in h_array2 asynchronously

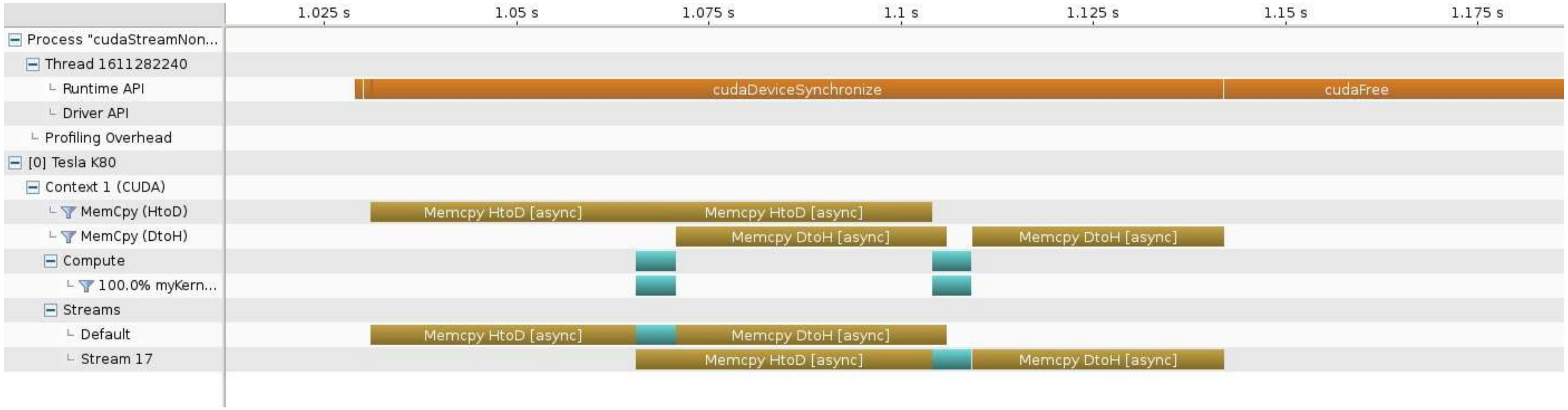
- Create a profile trace with nvprof: « nvprof --export-profile profile.nvvp ./bin »

- Read the profile using nvvp

# Example: legacy default stream + another non blocking stream

```
cudaStream_t mystream;
//streams creation
cudaStreamCreateWithFlags(&mystream, cudaStreamNonBlocking);

float * hostPtr ;
cudaMallocHost (...); //page-locked memory allocation

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice, 0 );
kernel1 <<<100, 512, 0, 0 >>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost, 0 );

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , mystream);
kernel2 <<<100, 512 , 0, mystream>>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , mystream);

cudaDeviceSynchronize();

//streams destruction
cudaStreamDestroy (mystream);
```

# Example: legacy default stream + another non blocking stream



# Stream Priority

- CUDA provides a mechanism to create a stream with a specified priority
  - **cudaStreamCreateWithPriority** (cudaStream_t* stream, unsigned int flags, int priority)

- Work in a higher priority stream may preempt work already executing in a low priority stream

- The range of allowable priorities can be obtained using:
  - **cudaDeviceGetStreamPriorityRange** ( int* leastPriority, int* greatestPriority )

- Priority follows a convention where:
  - **lower numbers** represent **higher priorities:** [greatestPriority, leastPriority]
  - '0' represents default priority
  - priorities outside the priority range are automatically clamped down or up

# Callbacks

- **cudaStreamAddCallback** ( cudaStream_t stream, cudaStreamCallback_t callback, void* userData, unsigned int  flags )

- Allows to add a function that will be executed on the host in a stream
  - the callback executes when all previous work in the stream is done
  - work place in a stream after a call to cudaStreamAddCallback do not start executing before the callback has completed

- A callback must not make CUDA API calls (directly or indirectly)
  - avoid deadlock

# Staged concurrent copy and execute

- Possible, if a kernel can start executing with only a chunk of the date present
  - E.g. elementwise, stencil operations

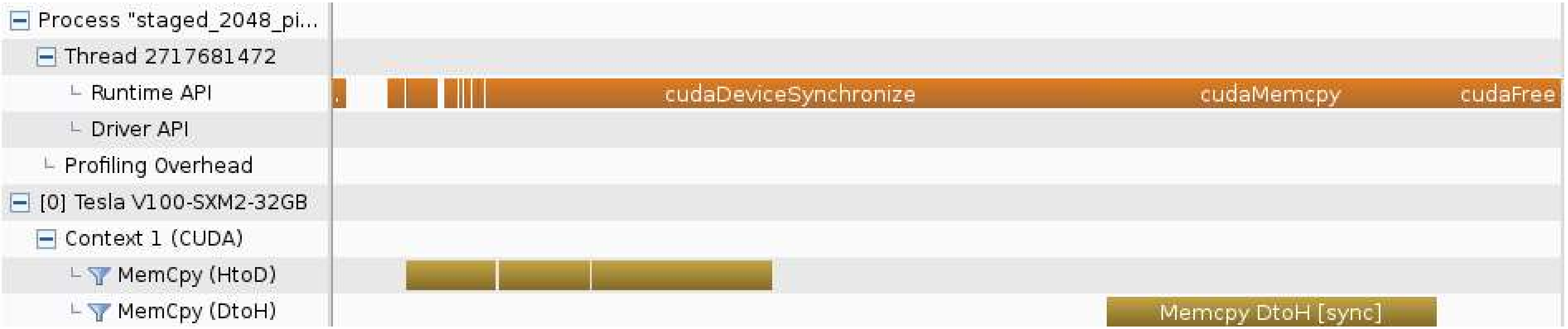
- Idea:
  - Copy successive chunks of data on the GPU and directly start the kernel (stage)
  - Map each stage to a stream -> (copyAsync + kernel execution)

- Constraints:
  - We need pinned memory!
  - Data dependencies

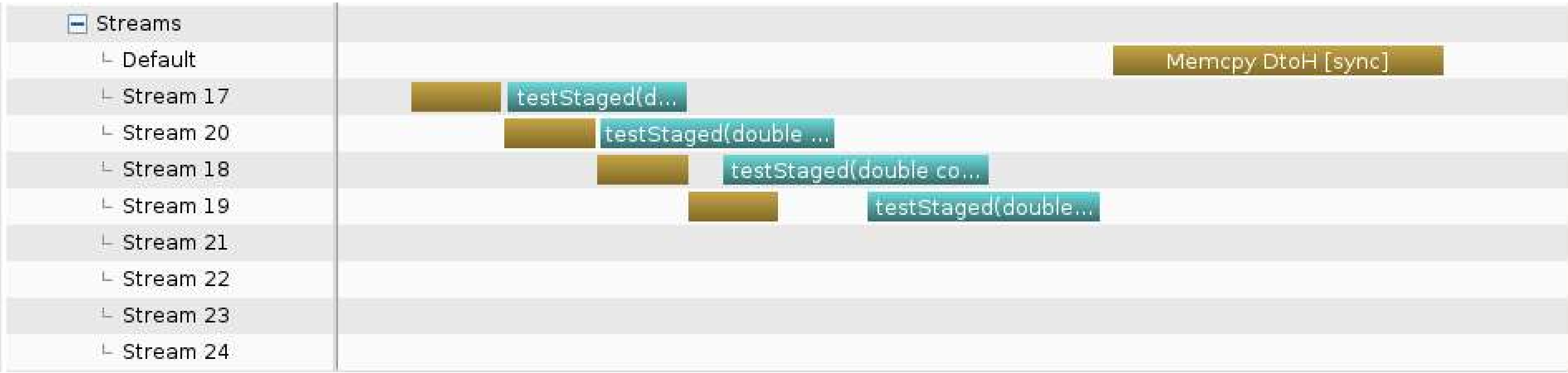
# Staged concurrent copy and execute - Pseudocode

```
for i in [1 .. stages]
    cudaStreamCreate(&stream[i]);
//allocate Buffer on GPU (N elements)
//setup gridsize and blocksize
int chunk_size = N * sizeof(elem) / stages;
for i in [1 .. stages] {
      int offset = i*N/nStreams;
      cudaMemcyAsync(devPtr +  offset, hostPtr + offset,
                           chunksize, H->D, stream[i]);

     my_kernel<<<gird, block, 0, stream[i]>>>(devPtr+offset)
}
cudaDeviceSynchronize();
//..
cudaMemcpy(D->H)
```

# Staged concurrent copy and execute - Profile




...





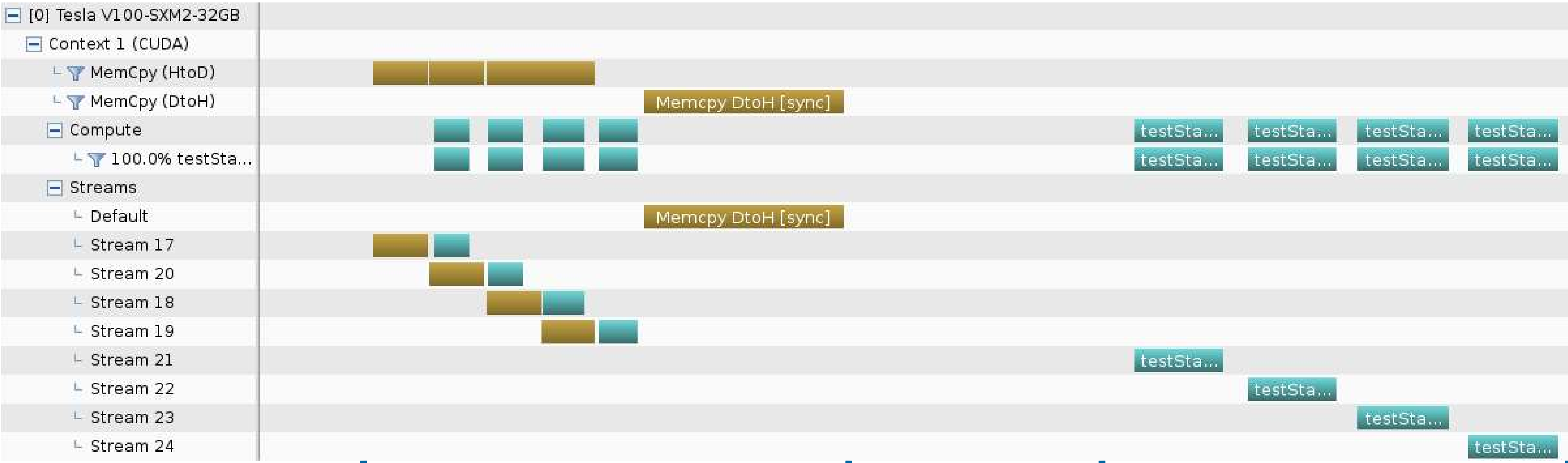


Bull
atos technologies

# Staged concurrent copy and execute - Tutorial

- /TP_CUDA_C/STREAM_EVENTS/STAGED_COPY_EXECUTE
- Implement Task 1
- Bonus-Task (make it work with zero-copy) -> Task 2
- (compile with –-std=c++11)
- Generate a profile

# Staged concurrent copy and execute - Profile




**No zero copy**

- "Shorter kernels", because no fetching via PCIe
- Only one H->D at a time

**With zero copy**

- Memory access (via PCIe within kernel execution)

Be careful with zero-copy. Its possible application is limited.

# EVENTS

16/09/2019

# CUDA EVENTS

- A mechanism to signal when operations have occurred in a stream
  - Useful for profiling and synchronization

- Events act as booleans:
  - Occurred
  - Not Occurred
  - !!! Default state = occurred

- **cudaEvent_t** event
  - Declares a event handle

# Managing Events

- **cudaEventCreate**(cudaEvent_t *event)
  - Creates an event

- **cudaEventCreateWithFlags**(cudaEvent_t *event, unsigned int flag)
  - flag: **cudaEventDisableTiming**
    - Disables timing to increase performance and avoid synchronization issues

- **cudaEventRecord**(cudaEvent_t event, cudaStream_t stream)
  - Set the event state to not occurred
  - Enqueue the event into a stream
    - event and stream must be on the same device
  - Event state is set to occurred when it reaches the front of the stream

- **cudaEventDestroy**(cudaEvent_t event)
  - Destroys an event

# Synchronization with Events

- **cudaEventQuery**(cudaEvent_t event)
  - non-blocking call
  - returns cudaSuccess if event has occurred
  - cudaErrorNotReady if any captured work is incomplete

- **cudaEventSynchronize**(cudaEvent_t event)
  - Blocks host until event has occured

- **cudaStreamWaitEvent**(cudaStream_t stream, cudaEvent_t event, unsigned int flags)
  - flags must be 0
  - Blocks stream until event occurs
  - Only blocks launches after this call
  - Does not block the host!

# LAB: Events

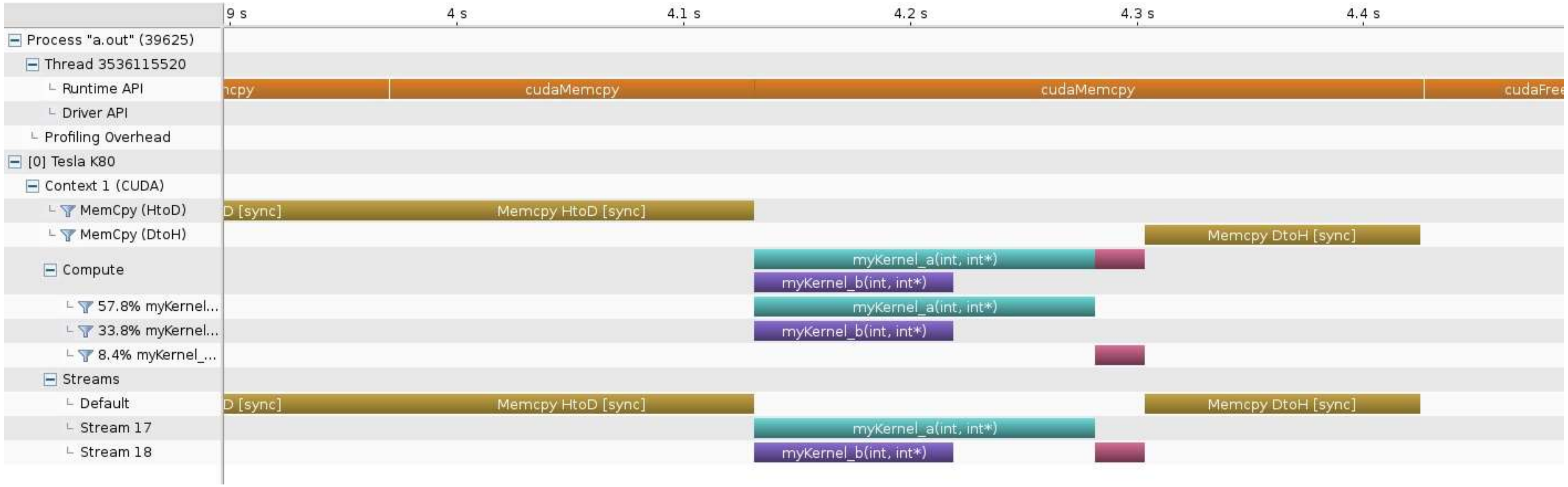
- Complete main.cu
  - call myKernel_a in a stream
  - call myKernel_b then myKernel_c in an other stream
  - ensure than myKernel_c start only after myKernel_a complete using event

- Generate a trace with nvprof
  - check that myKernel_c starts after mykernel_a

# LAB: Events

```
cudaStream_t stream [2];
cudaStreamCreate (&stream[0]);
cudaStreamCreate (&stream[1]);

cudaEvent_t event;
cudaEventCreateWithFlags(&event, cudaEventDisableTiming);

myKernel_a<<< nbBlocks, threadsPerBlock, 0, stream[0] >>>(nbchunk, d_a);
myKernel_b<<< nbBlocks, threadsPerBlock, 0, stream[1] >>>(nbchunk, d_b);

cudaEventRecord(event, stream[0]);

cudaStreamWaitEvent(stream[1], event, 0);
...
myKernel_c<<< nbBlocks2, threadsPerBlock2, 0, stream[1] >>>(size, d_a, d_b, d_c);

cudaMemcpy(h_c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);

cudaDeviceSynchronize();
```

# LAB: Events



9 s
4 s
4.1 s
4.2 s
4.3 s
4.4 s
Process "a.out" (39625)
Thread 3536115520
Runtime API
Driver API
Profiling Overhead
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
57.8% myKernel...
33.8% myKernel...
8.4% myKernel_...
Streams
Default
Stream 17
Stream 18
cudaMemcpy
cudaMemcpy
cudaFree
Memcpy HtoD [sync]
Memcpy DtoH [sync]
myKernel_a(int, int*)
myKernel_b(int, int*)

# Timing Using Event

- **cudaEventElapsedTime**(float* ms, cudaEvent_t start, cudaEvent_t end)
  - Computes the elapsed time between two events
    - in milliseconds with a resolution of around 0.5 microseconds

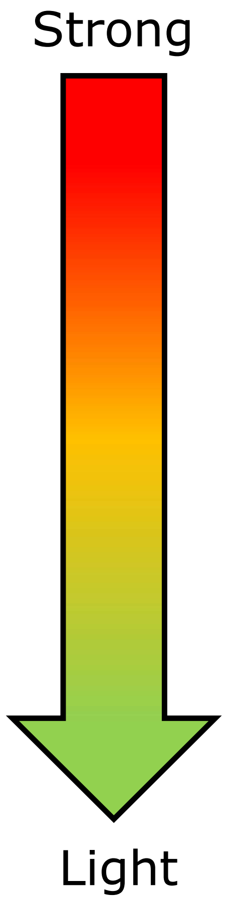
```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);

for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync( ... , stream[i]);
    MyKernel<<< ... , stream[i]>>>( ... );
    cudaMemcpyAsync( ... , stream[i]);
}

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

# Reminder: Synchronization

- **cudaDeviceSynchronize()**
  - Synchronize everything
    - Blocks host until all issued CUDA calls are complete

- **cudaStreamSynchronize(stream)**
  - Synchronize host with regard to a specific stream
    - Blocks host until all issued CUDA calls in stream are complete

- Synchronize host or devices using **events**
  - allow to synchronize several streams



# Implicit Synchronization

- Two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:
  - a page-locked host memory allocation
  - a device memory allocation
  - a device memory set
  - a memory copy between two addresses to the same device memory,
  - any CUDA command to the NULL stream
  - a switch between the L1/shared memory configurations described in Compute Capability 3.x and Compute Capability 7.x

# Useful Link

- http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

# Copyright

# Thanks

For more information please contact:

Georges-Emmanuel Moulard
M+ 33 6 85529054
georges-emmanuel.moulard@atos.net



29-10-2018