Lecture 6: streams, and some odds and ends

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Overview

- synchronicity
- streams
- multiple GPUs
- other odds and ends

Warnings

- I haven't tried most of what I will describe
- sometimes details change from one version of CUDA to the next – I think everything here is correct for the current version
- overall, keep things simple unless it's really needed for performance
- if it is needed, proceed with extreme caution, do practicals 6, 11 and 12, and check out the NVIDIA sample codes

Synchronicity

A computer system has lots of components:

- CPU(s)
- GPU(s)
- memory controllers
- network cards

Many of these can be doing different things at the same time – usually for different processes, but sometimes for the same process

Synchronicity

The von Neumann model of a computer program is synchronous with each computational step taking place one after another

- this is an idealisation – never true in practice
- compiler frequently generates code with overlapped instructions (pipelined CPUs) and does other optimisations which re-arrange execution order and avoid redundant computations
- however, it is usually true that as a programmer you can think of it as a synchronous execution when working out whether it gives the correct results
- when things become asynchronous, the programmer has to think very carefully about what is happening and in what order

Synchronicity

With GPUs we have to think even more carefully:

- host code executes on the CPU(s);
  kernel code executes on the GPU(s)
- . . . but when do the different bits take place?
- . . . can we get better performance by being clever?
- . . . might we get the wrong results?

Key thing is to try to get a clear idea of what is going on
– then you can work out the consequences

GPU code

- for each warp, code execution is effectively synchronous
- different warps execute in an arbitrary overlapped fashion – use `__syncthreads()` if necessary to ensure correct behaviour
- different thread blocks execute in an arbitrary overlapped fashion

All of this has been described over the past 3 days – nothing new here.

The focus of these new slides is on host code and the implications for CPU and GPU execution

Host code

Simple / default behaviour:

- 1 CPU
- 1 GPU
- 1 thread on CPU (i.e. scalar code)
- 1 default “stream” on GPU

Note: within the GPU, all operations in the default stream operate strictly in sequence, each one finishing before the next one starts

Host code

- most CUDA calls are synchronous / blocking:
- example: `cudaMemcpy`
  - host call starts the copying and waits until it has finished before the next instruction in the host code
  - why? – ensures correct execution if subsequent host code reads from, or writes to, the data being copied

NOTE: `cudaMemcpy` operates asynchronously when copying no more than 64kB from host to device – it does this by first copying the data to a host buffer, before returning to the host code (see Section 3.2.8.1 in the Programming Guide)

Host code

- CUDA kernel launch is asynchronous / non-blocking; host call starts the kernel execution, but doesn't wait for it to finish before going on to next instruction
- similar for `cudaMemcpyAsync`
  - starts the copy but doesn't wait for completion
  - has to be done through a "stream"
  - must use page-locked memory (also known as pinned memory) to guarantee it is asynchronous – see documentation
- host will wait for completion at a blocking `cudaMemcpy` or `cudaDeviceSynchronize` call
- benefit? can reduce execution time by overlapping CPU and GPU execution

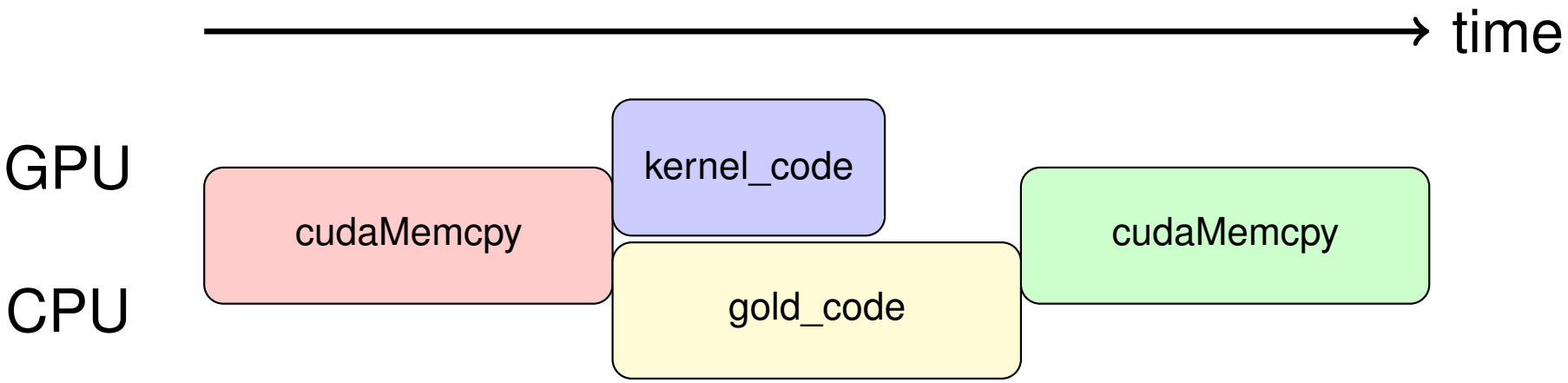
Page-locked memory

Section 3.2.6:

- host memory is usually paged, so run-time system keeps track of where each page is located
- for higher performance, can fix some pages, but means less memory available for everything else
- CUDA uses this for better host <–> GPU bandwidth, and also to hold “device” arrays in host memory
- can provide up to 100% improvement in bandwidth
- allocated using `cudaHostAlloc`, or registered by `cudaHostRegister`

Host code

```
cudaMemcpy(d_u1,h_u1,bytes,cudaMemcpyHostToDevice);
kernel_code<<<dimGrid, dimBlock>>>(d_u1, d_u2);
gold_code(h_u1,h_u2);
cudaMemcpy(h_u1,d_u1,bytes,cudaMemcpyDeviceToHost);
```



Host code

What could go wrong?

- kernel timing – need to make sure it's finished
- could be a problem if the host uses data which is read/written directly by kernel, or transferred by `cudaMemcpyAsync`
- `cudaDeviceSynchronize()` can be used to ensure correctness (similar to `__syncthreads()` for kernel code)

Multiple Streams

Quoting from Section 3.2.8.5 in the CUDA Programming Guide:

> Applications manage the concurrent operations described above through streams.
>
> A stream is a sequence of commands (possibly issued by different host threads) that execute in order.
>
> Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently.

Multiple Streams

Optional stream argument for

- kernel launch
- `cudaMemcpyAsync`

with streams creating using `cudaStreamCreate`

Within each stream, CUDA operations are carried out in order (i.e. FIFO – first in, first out); one finishes before the next starts

Key to getting better performance is using multiple streams to overlap things

Default stream

The way the default stream behaves in relation to others depends on a compiler flag:

- no flag, or `--default-stream legacy`

  old (bad) behaviour in which a `cudaMemcpy` or kernel launch on the default stream blocks/synchronizes with other streams

- `--default-stream per-thread`

  new (good) behaviour in which the default stream doesn't affect the others

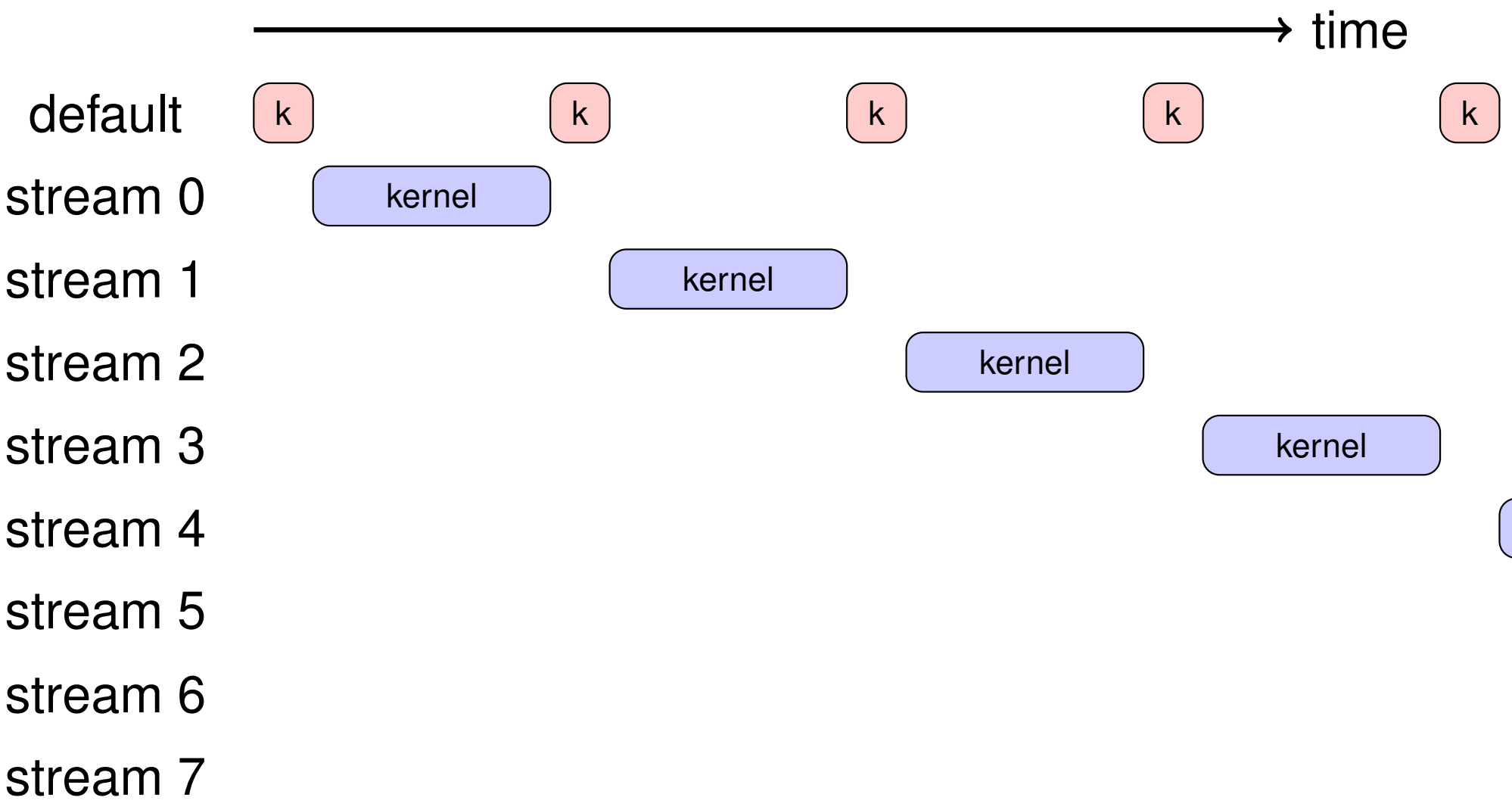
- note: flag label is a bit odd – it has other effects too

Example 1

```
cudaStream_t streams[8];
float        *data[8];

for (int i=0; i<8; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));
}

for (int i=0; i<8; i++) {
  // launch a tiny kernel on default stream
  k<<<1, 1>>>();

  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);
}
cudaDeviceSynchronize();
```

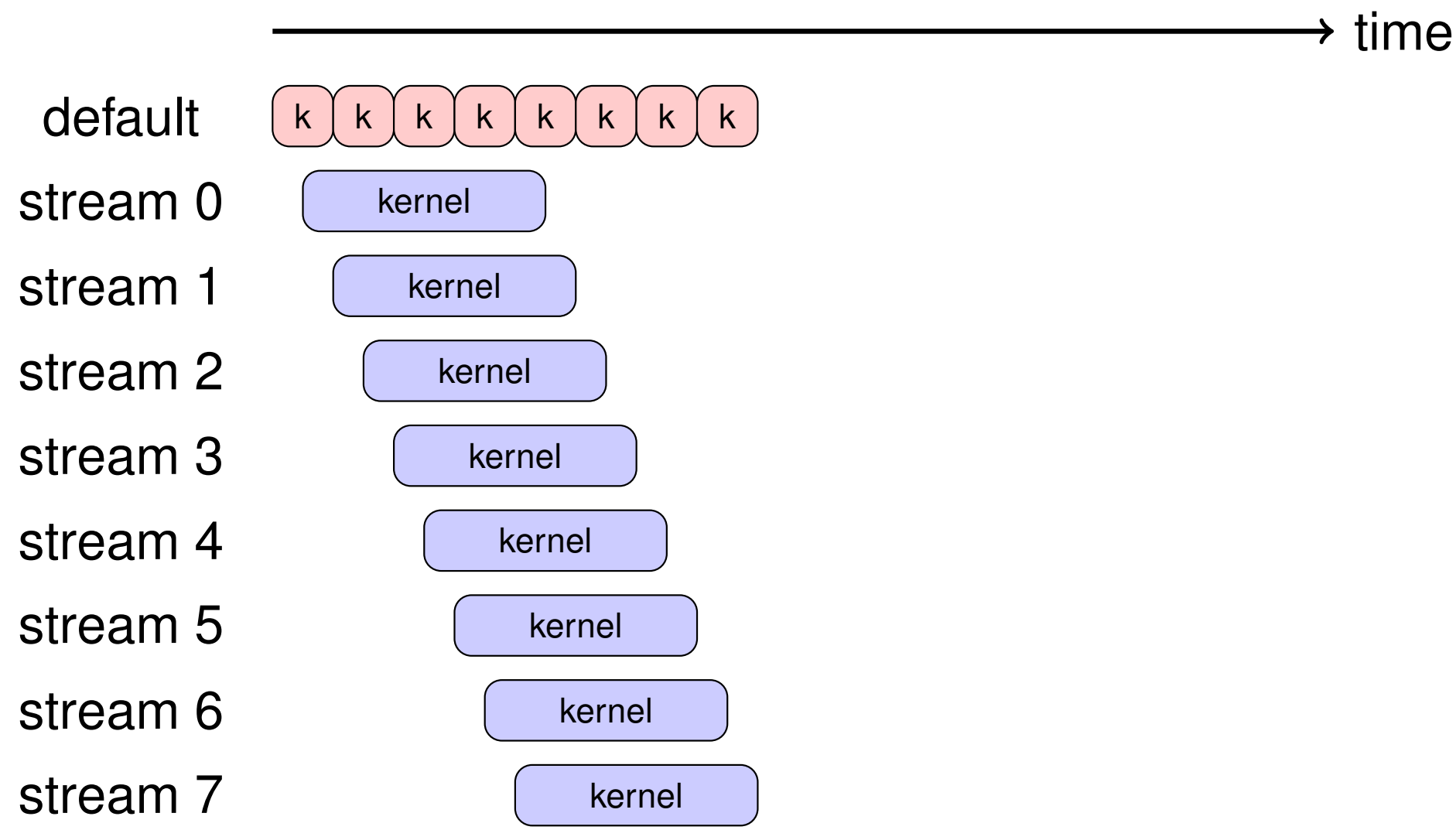
Example 1

old behaviour:



Example 1

new behaviour:



Default stream

The second (main?) effect of the flag comes when using multiple threads (e.g. OpenMP or POSIX multithreading)

In this case the effect of the flag is to create a separate independent (i.e. non-interfering) default stream for each thread

Using multiple default streams, one per thread, is a good alternative to using multiple “proper” streams

Example 2

```
omp_set_num_threads(8);
float *data[8];

for (int i = 0; i < 8; i++)
  cudaMalloc(&data[i], N * sizeof(float));

#pragma omp parallel for
for (int i = 0; i < 8; i++) {
  printf(" thread ID = %d \n",omp_get_thread_num());

  // launch one worker kernel per thread
  kernel<<<1, 64>>>(data[i], N);
}

cudaDeviceSynchronize();
```

Stream commands

Each stream executes a sequence of kernels, but sometimes you also need to do something on the host.

There are at least two ways of coordinating this:

- use a separate thread for each stream
  - it can wait for the completion of all pending tasks, then do what's needed on the host
- use just one thread for everything
  - for each stream, add a callback function to be executed (by a new thread) when the pending tasks are completed
  - it can do what's needed on the host, and then launch new kernels (with a possible new callback) if wanted

Stream commands

- `cudaStreamCreate()`
  creates a stream and returns an opaque “handle”
- `cudaStreamCreateWithPriority()`
  additionally defines an execution priority
- `cudaStreamSynchronize()`
  waits until all preceding commands have completed
- `cudaStreamQuery()`
  checks whether all preceding commands have completed
- `cudaStreamAddCallback()`
  adds a callback function to be executed on the host once all preceding commands have completed

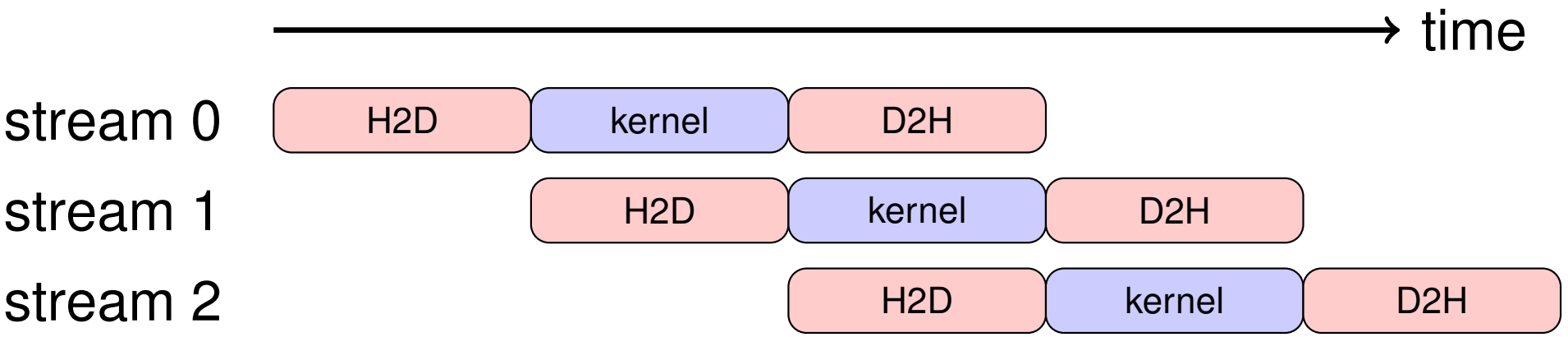
Stream events

Useful for synchronisation and timing between streams:

- `cudaEventCreate(event)`
  creates an “event”
- `cudaEventRecord(event,stream)`
  puts an event into a stream (by default, stream 0)
- `cudaEventSynchronize(event)`
  CPU waits until event occurs
- `cudaStreamWaitEvent(stream,event)`
  stream waits until event occurs in another stream
- `cudaEventQuery(event)`
  check whether event has occured
- `cudaEventElapsedTime(time,event1,event2)`

Two use cases

One important use case for streams is to overlap PCIe transfers with kernel computation for real-time signal processing.

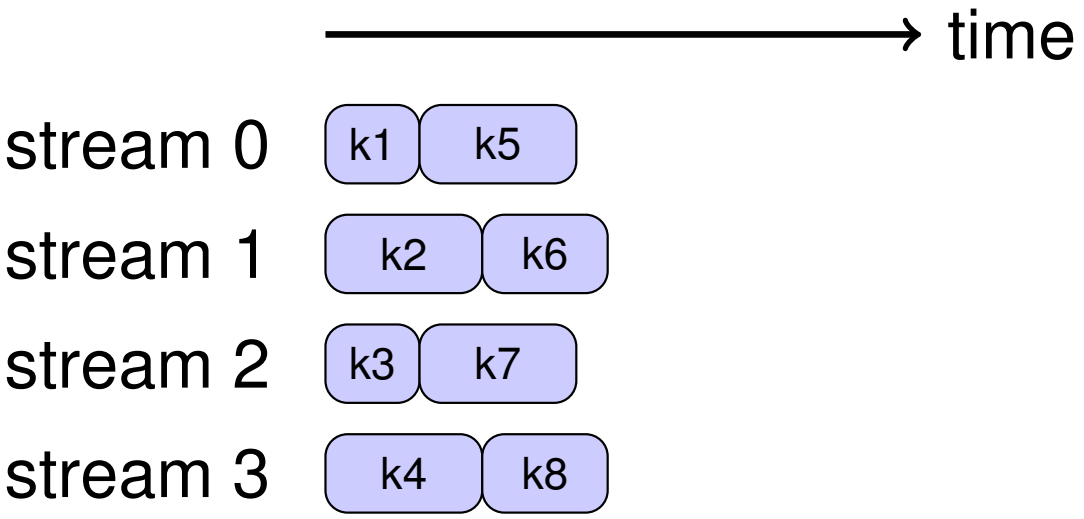


In the best case this gives a factor 3$\times$ improvement when the data transfers take as long as the kernel computation

Two use cases

A second use case is to overlap the execution of lots of small independent kernels which otherwise would execute sequentially.

Using multiple streams keeps all of the SMs in a big GPU busy.



CUDA graphs

CUDA graphs (Section 3.2.8.7):

- I think this looks really interesting as an alternative to streams with programmed interdependencies, but I haven't yet tried it out
- enables a programmer to specify a set of computational tasks as a task DAG (Directed Acyclic Graph)
- GPU is responsible for managing the DAG, noting when tasks complete and launching new tasks that are now able to run
- can also "capture" a DAG by noting what happens within streams

Occupancy and Cooperative Groups

CUDA Runtime API: Section 6.8 – Occupancy

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`

calculates the maximum number of copies of the kernel which can run in a single SM.

For an example of its use see `prac2_device.cu` and this NVIDIA blog

Multiplied by the number of SMs gives the maximum number of blocks which can execute simultaneously without any queueing. With new Cooperative Groups (see CUDA C++ Programming Guide: Section 8) can launch these together and synchronize across the group.

Multiple devices

What happens if there are multiple GPUs?

CUDA devices within the system are numbered, not always in order of decreasing performance

- by default a CUDA application uses the lowest number device which is “visible” and available
- visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`
- current device can be set by using `cudaSetDevice`
- each stream is associated with a particular device – current device for a kernel launch or a memory copy
- see `simpleMultiGPU` example in NVIDIA samples
- see Section 3.2.9 for more information

Multiple devices

If a user is running on multiple GPUs, data can go directly between GPUs (peer – peer) – doesn't have to go via CPU

- very important when using direct NVlink interconnect – much faster than PCIe
- `cudaMemcpy` can do direct copy from one GPU's memory to another
- a kernel on one GPU can also read directly from an array in another GPU's memory, or write to it
- this even includes the ability to do atomic operations with remote GPU memory
- for more information see Section 6.15, "Peer Device Memory Access" in CUDA Runtime API documentation: https://docs.nvidia.com/cuda/cuda-runtime-api/

Multi-user support

What if different processes try to use the same device?

Depends on system compute mode setting (Section 3.4):

- in “default” mode, each process uses the first device
  - not good when you have 2 identical fast GPUs
- in “exclusive” mode, each process is assigned to first unused device; it’s an error if none are available
- `cudaGetDeviceProperties` reports mode setting, and lots of other properties; see also `cudaDeviceGetAttribute`
- mode can be changed by sys-admin using `nvidia-smi` command line utility

Makefile

Compiling:

- `Makefile` for first few practicals uses `nvcc` to compile both the host and the device code
  - internally it uses `gcc` for the host code, at least by default
  - device code compiler based on open source LLVM compiler
- sometimes, prefer to use other compilers (e.g. `icc`, `mpicc`) for main code that doesn't have any CUDA calls
- this is fine provided you use `-fPIC` flag for position-independent-code (don't know what this means but it ensures interoperability)
- can also produce libraries for use in the standard way

Makefile

Prac 6 Makefile:

```
INC   := -I$(CUDA_HOME)/include -I.
LIB   := -L$(CUDA_HOME)/lib64 -lcudart
FLAGS := --ptxas-options=-v --use_fast_math

main.o: main.cpp
	g++ -c -fPIC -o main.o main.cpp

prac6.o: prac6.cu
	nvcc prac6.cu -c -o prac6.o $(INC) $(FLAGS)

prac6: main.o prac6.o
	g++ -fPIC -o prac6 main.o prac6.o $(LIB)
```

Makefile

Prac 6 Makefile to create a library:

```
INC    := -I$(CUDA)/include -I.
LIB    := -L$(CUDA)/lib64 -lcudart
FLAGS := --ptxas-options=-v --use_fast_math

main.o: main.cpp
     g++ -c -fPIC -o main.o main.cpp

prac6.a: prac6.cu
     nvcc prac6.cu -lib -o prac6.a $(INC) $(FLAGS)

prac6a: main.o prac6.a
     g++ -fPIC -o prac6a main.o prac6.a $(LIB)
```

Makefile

Other compiler options:

- `-arch=sm_80`
  specifies GPU architecture
- `-Xptxas -dlcm=ca`
  uses L1/L2 cache in usual way – general default, also implies 128 byte cache line
- `-Xptxas -dlcm=cg`
  bypass L1 cache / go straight to L2 – default for read-only access (?), 32 byte cache line

  see Section 16.4.2 for a little more info – applies to all recent GPUs as far as I know

  see also Sections 7.10, 7.11 for specialised load instructions

Conclusions

This lecture has discussed a number of more advanced topics

As a beginner, you can ignore almost all of them

As you get more experienced, you will probably want to start using some of them to get the very best performance