Lecture 6
Using multiple GPUs and loose ends

Prof Wes Armour

wes.armour@eng.ox.ac.uk

Prof Mike Giles

mike.giles@maths.ox.ac.uk

Oxford e-Research Centre

Department of Engineering Science

Learning outcomes

In this sixth lecture we will look at CUDA streams and how they can be used to increase performance in GPU computing.

You will learn about:

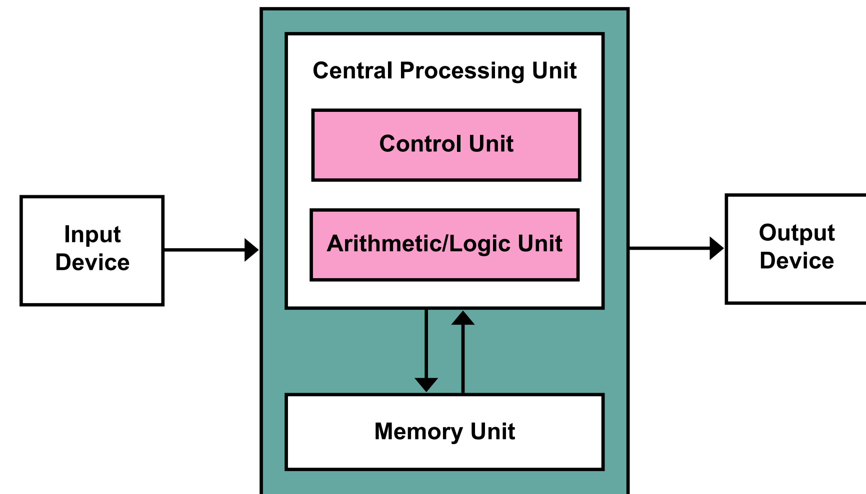
- Synchronicity between host and device.
- Multiple streams and devices.
- How to use multiple GPUs.

Synchronicity

Synchronicity

The von **Neumann model of a computer program is synchronous with each computational step taking place one after another** (because instruction fetch and data movement share the same communication bus).

This is an idealisation, and is almost never true in practice.

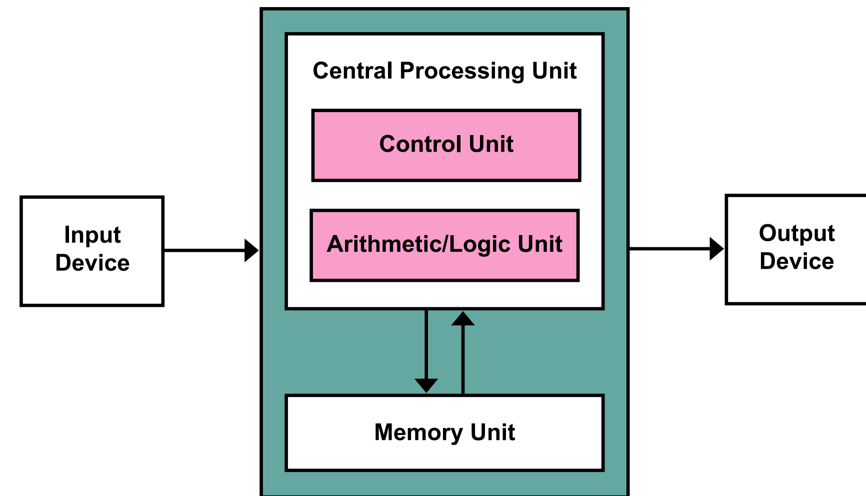


Kapooht [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)]

Synchronicity

Compilers will generate code with **overlapped instructions** (pipelining – see lecture one), **re-arrange execution order** and **avoid redundant computations** to produce more optimal code.

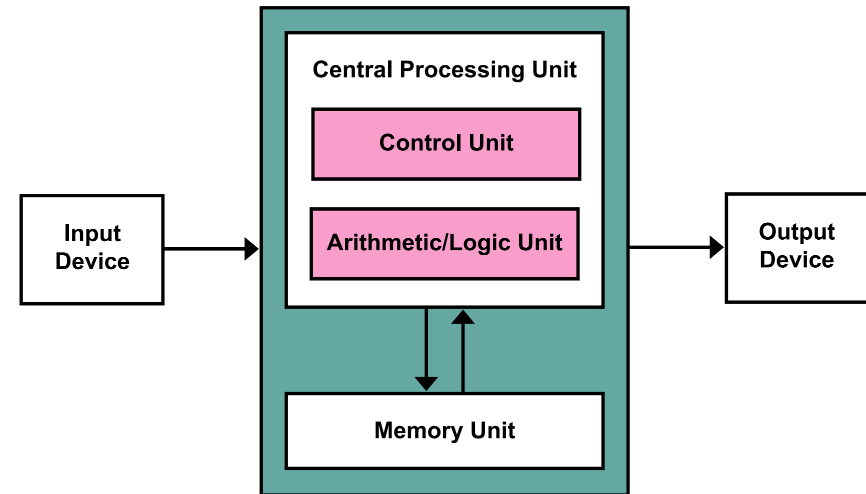
As a **programmer we don't normally worry** about this **and think of execution sequentially** when working out whether a program gives the correct result.



Kapooht [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)]

Synchronicity

However, when things become asynchronous, the programmer has to think very carefully about what is happening and in what order!



Kapooht [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)]

Synchronicity - GPUs

When writing code for GPUs we have to think even more carefully, because:

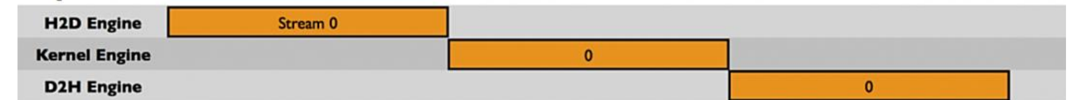
Our host code executes on the CPU(s)

Our kernel code executes on the GPU(s)

. . . but when (in time) do the different bits take place?

. . . can we get better performance by being clever?

. . . might we get the wrong results?



The most important thing is to try to get a clear idea of <u>what is going on, and when</u> – then you can work out the consequences…

Blocking and non-blocking calls

Host code – blocking calls

Most CUDA <u>calls</u> are synchronous (often called “blocking”).

An example of a blocking call is `cudaMemcpy()`.

1. Host call starts the copy (HostToDevice / DeviceToHost).
2. Host **<u>waits</u>** until it the copy has finished.
3. Host continues with the next instruction in the host code once the copy has completed.

```
cudaMalloc(&d_data, size);
float *h_data = (float*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Host code – blocking calls

Why do this???

This mode of operation **ensures correct execution**.

For example it **ensures that data is present if the next instruction needs to read from the data that has been copied**…

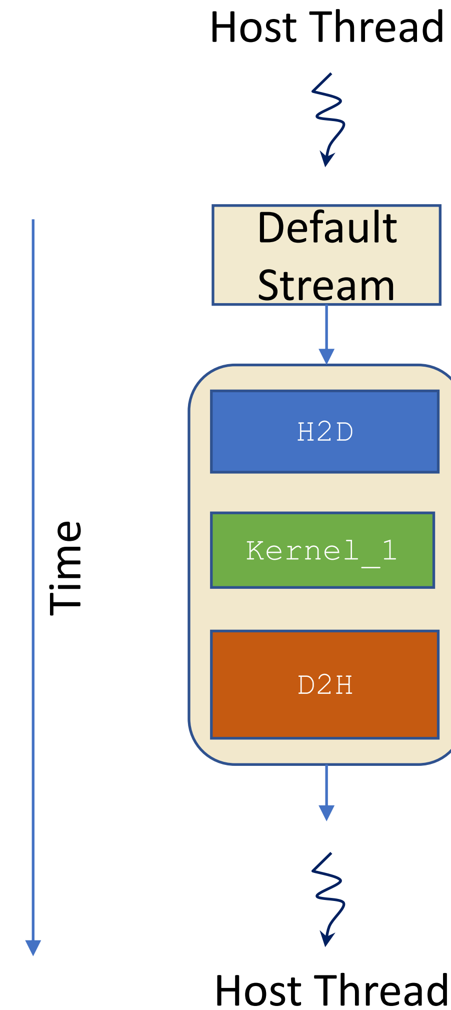
```
cudaMalloc(&d_data, size);
float *h_data = (float*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Host code – blocking calls

So the control flow for our code looks something like…

```
cudaMalloc(&d_data, size);
float *h_data = (float*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Note that, within the stream, execution is synchronous –
the second cudaMemcpy waits for the completion of Kernel_1



Host code – non-blocking calls

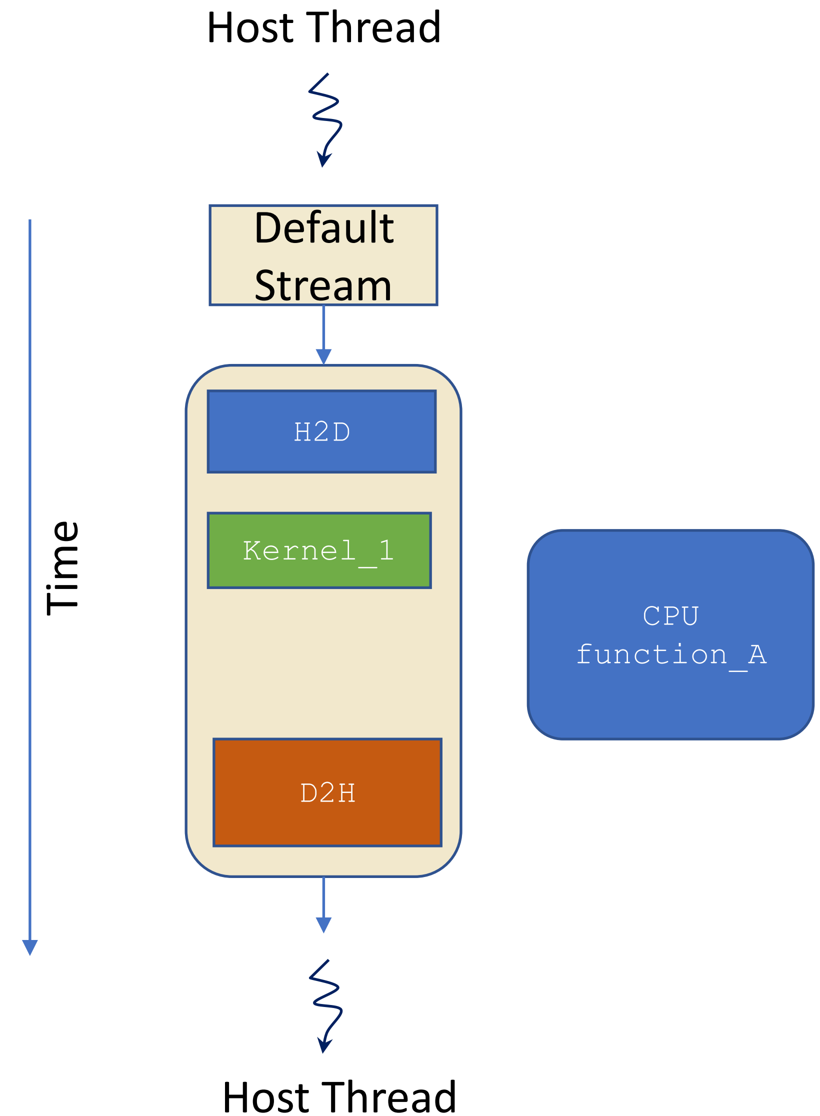
In CUDA, **kernel launches are <u>asynchronous</u>** (often called “non-blocking”).

An example of kernel execution from host perspective:

1. Host call starts the kernel execution.

2. Host does not wait for kernel execution to finish.

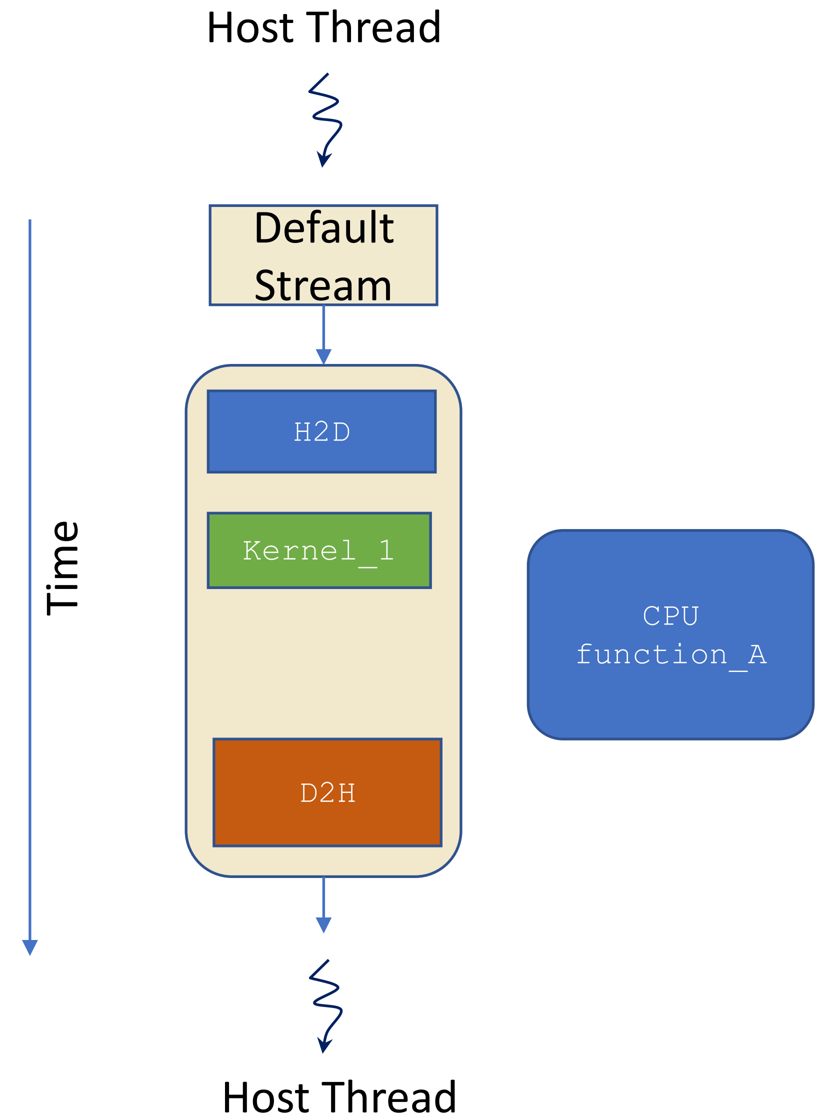
3. Host moves onto the next instruction.



Host code – non-blocking calls

The “crazy code” for our last control flow diagram might look like…

```
cudaMalloc(&d_data, size);
float *h_data = (float*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
CPU_function_A( … );
cudaMemcpy ( …, D2H );

…
```



A simple example

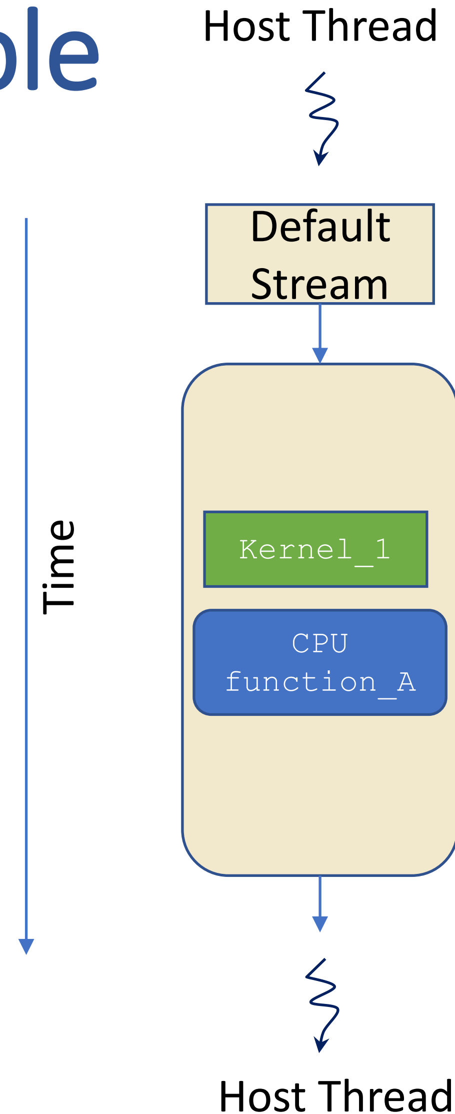
Here is an example of **synchronous** execution on the GPU followed by CPU.

Note: I’ve used managed memory only to reduce the size of my code so it fits onto a single slide.

1. We execute “kernel” on the GPU

2. We call cudaDeviceSynchronise();

3. We execute a for loop on the CPU



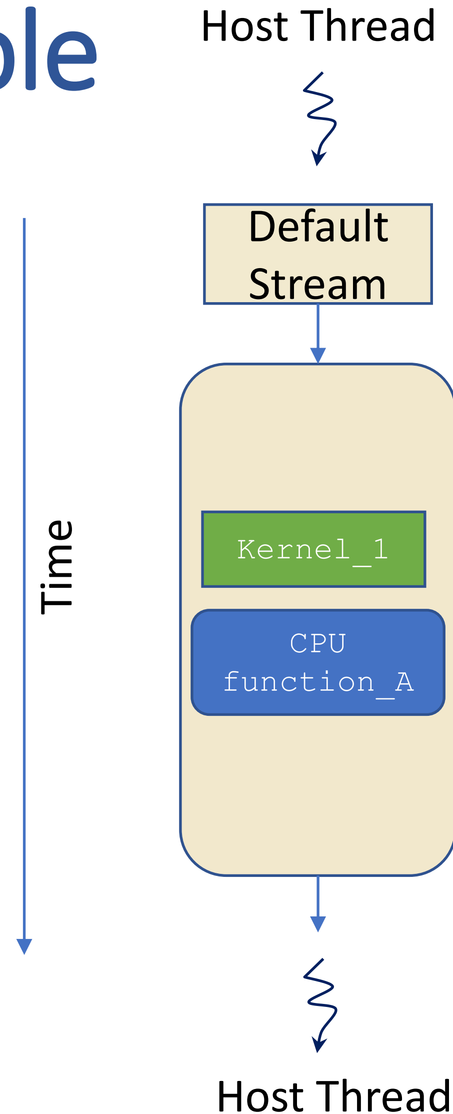

```
#include <stdio.h>
#include <cuda.h>
#include <time.h>

__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 100000000; // number of floats
    int size = n * sizeof(float);

    float *d_a, *h_a;;
    cudaMallocManaged(&d_a, size);
    h_a = (float*)malloc(size);

    // start timer
    clock_t start = clock();

    int offset = 10;

    // launch kernel
    kernel<<<n/128, 128>>>(d_a, offset);

    cudaDeviceSynchronize();

    for(int i = 0; i < n; i++) {
        h_a[i] = h_a[i] + offset;
    }

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;

    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

A simple example

Here is an example of synchronous execution on the GPU followed by CPU.

On my laptop execution takes about **640** milliseconds.

```
user@machine:~$ ./test && ./test && ./test &
Elapsed time: 661.366000 ms
Elapsed time: 634.194000 ms
Elapsed time: 651.963000 ms
Elapsed time: 622.979000 ms
Elapsed time: 613.481000 ms
Elapsed time: 679.147000 ms
Elapsed time: 621.406000 ms
Elapsed time: 618.451000 ms
Elapsed time: 646.294000 ms
```



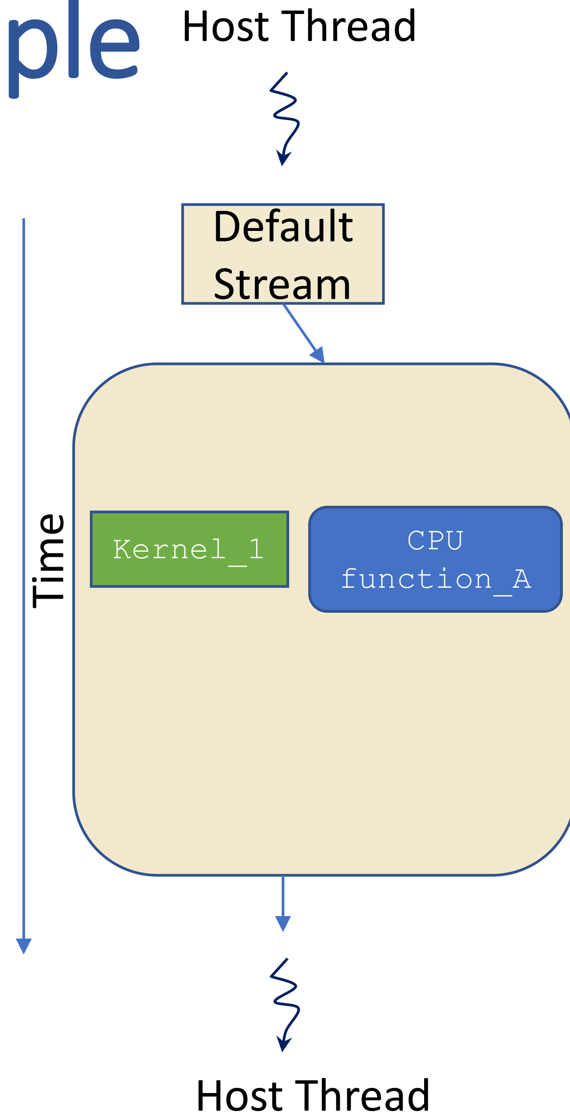

```c
#include <stdio.h>
#include <cuda.h>
#include <time.h>

__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 100000000; // number of floats
    int size = n * sizeof(float);

    float *d_a, *h_a;;
    cudaMallocManaged(&d_a, size);
    h_a = (float*)malloc(size);

    // start timer
    clock_t start = clock();

    int offset = 10;

    // launch kernel
    kernel<<<n/128, 128>>>(d_a, offset);

    cudaDeviceSynchronize();

    for(int i = 0; i < n; i++) {
        h_a[i] = h_a[i] + offset;
    }

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;

    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

A simple example

Here is an example of **asynchronous** execution on the GPU followed by CPU.

Note: I've used managed memory only to reduce the size of my code so it fits onto a single slide.

1. We execute "kernel" on the GPU
2. ~~We call cudaDeviceSynchronise();~~
3. We execute a for loop on the CPU




```
#include <stdio.h>
#include <cuda.h>
#include <time.h>

__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 100000000; // number of floats
    int size = n * sizeof(float);

    float *d_a, *h_a;;
    cudaMallocManaged(&d_a, size);
    h_a = (float*)malloc(size);

    // start timer
    clock_t start = clock();

    int offset = 10;

    // launch kernel
    kernel<<<n/128, 128>>>(d_a, offset);

    //cudaDeviceSynchronize();

    for(int i = 0; i < n; i++) {
        h_a[i] = h_a[i] + offset;
    }

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;

    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

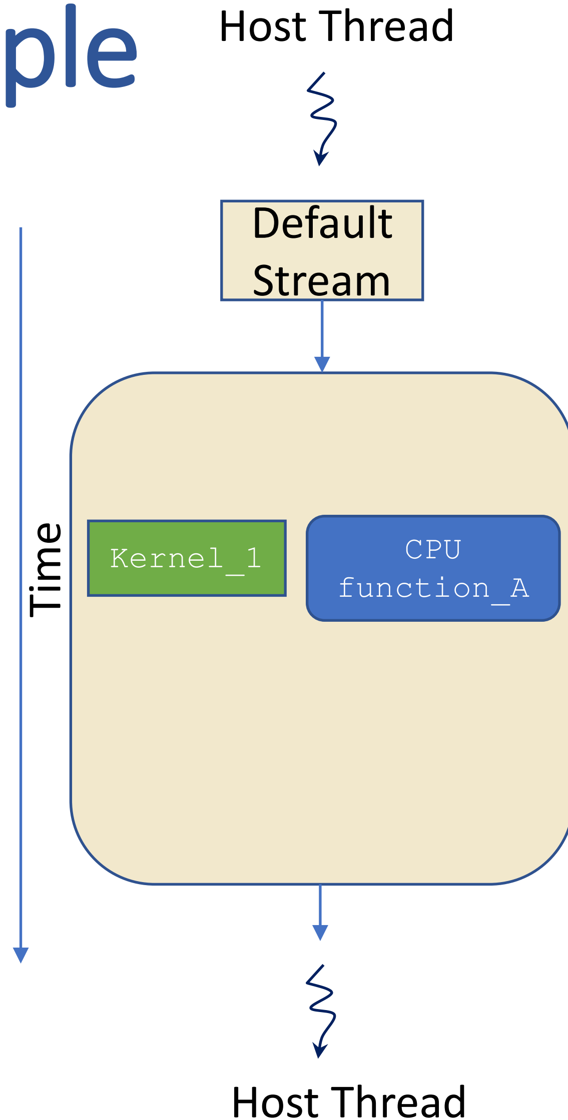
A simple example

Here is an example of **asynchronous** execution on the GPU followed by CPU.

On my laptop execution now takes about **520** milliseconds.

```
wa78@engs-29517:~$ ./test && ./test && ./te
Elapsed time: 541.461000 ms
Elapsed time: 534.683000 ms
Elapsed time: 475.033000 ms
Elapsed time: 508.959000 ms
Elapsed time: 519.161000 ms
Elapsed time: 532.170000 ms
Elapsed time: 515.032000 ms
Elapsed time: 525.524000 ms
Elapsed time: 508.968000 ms
```

So the synchronous code is about 1.25x slower than the asynchronous code!!




```
#include <stdio.h>
#include <cuda.h>
#include <time.h>

__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 100000000; // number of floats
    int size = n * sizeof(float);

    float *d_a, *h_a;;
    cudaMallocManaged(&d_a, size);
    h_a = (float*)malloc(size);

    // start timer
    clock_t start = clock();

    int offset = 10;

    // launch kernel
    kernel<<<n/128, 128>>>(d_a, offset);

    //cudaDeviceSynchronize();

    for(int i = 0; i < n; i++) {
        h_a[i] = h_a[i] + offset;
    }

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;

    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

Host code – non-blocking calls

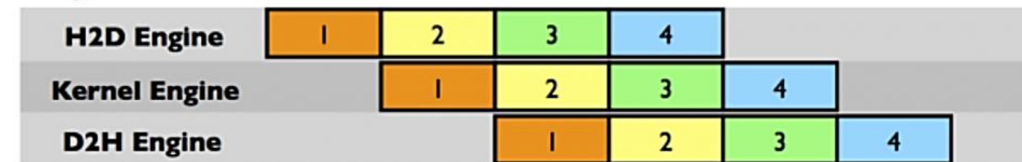
Another example of a **non-blocking** call is `cudaMemcpyAsync()`.

This function starts the copy but doesn't wait for completion.

Synchronisation is performed through a "stream".

You must use page-locked memory (also known as pinned memory) – see Documentation.

In both of our examples, the host eventually waits when at (for example) a `cudaDeviceSynchronize()` call.



The benefit of using streams is that you can improve performance (in some cases, not all) by overlapping communication and compute, or CPU and GPU execution.

Asynchronous host code

When using asynchronous calls, <u>things to watch out for</u>, and things that can go wrong are:

- 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).

CUDA Streams

Simple host code

The basic / simple / default behaviour in CUDA is that we have:

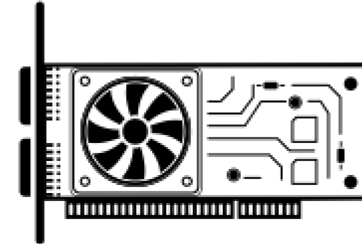
1x CPU.

1x GPU.

1x thread on CPU (i.e. scalar code).

1x “**stream**” on GPU (called the “**default stream**”).

The default stream is what we have been working with… Until now…



Device

Host

By RRZEicons [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons

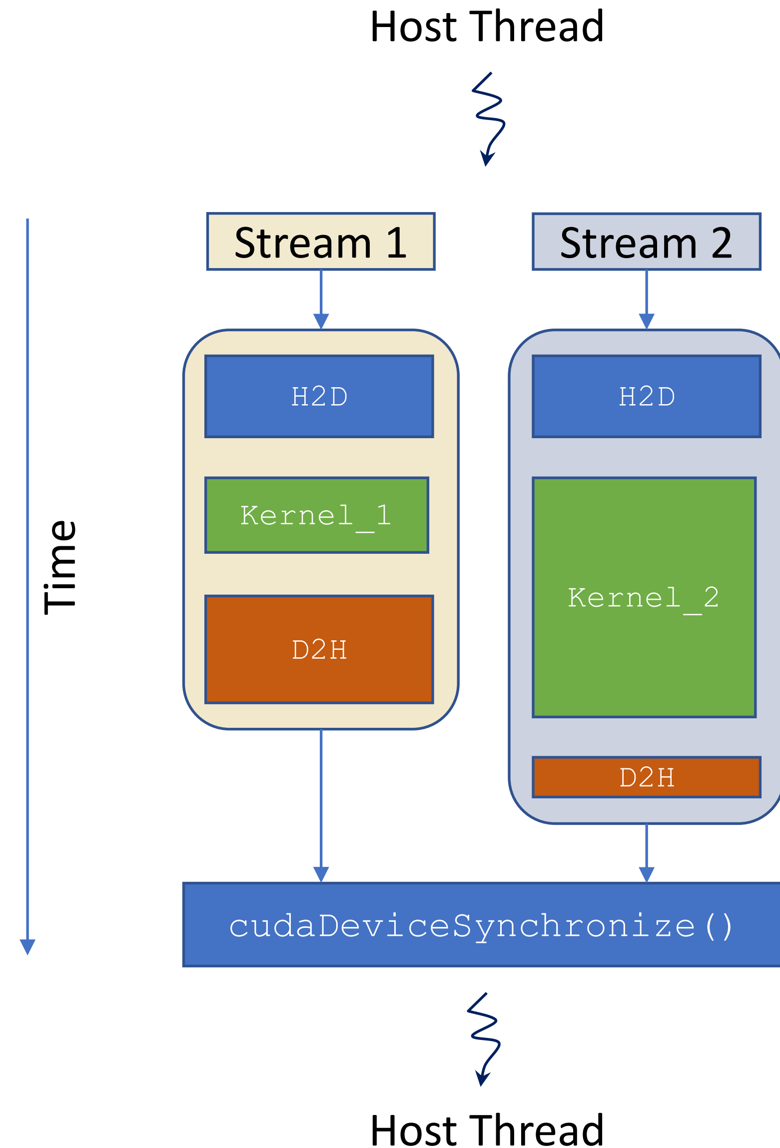
CUDA Streams

Quoting from section 3.2.8.5 in the CUDA Programming Guide:

Applications manage concurrency through streams.

A **stream** is a sequence of commands (*possibly issued by different host threads*) that **execute in order (serialised)**.

Different streams, on the other hand, may **execute** their commands **out of order** with respect to one another or concurrently.



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

Multiple CUDA Streams

When using streams in CUDA, you must supply a "stream" variable as an argument to:

- kernel launch
- `cudaMemcpyAsync()`

Which is created using `cudaStreamCreate();`

As shown over the last couple of slides:

- Operations within the **same stream** are ordered - (i.e. **FIFO** – first in, first out) – they cant overlap.
- Operations in **different streams** are unordered wrt each other and **can overlap**.

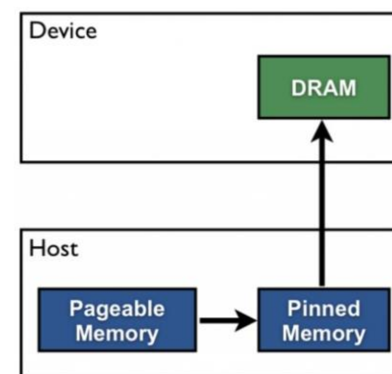
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
my_kernel_one<<<blocks,threads,0,stream1>>>(…);
cudaStreamDestroy(stream1);
```

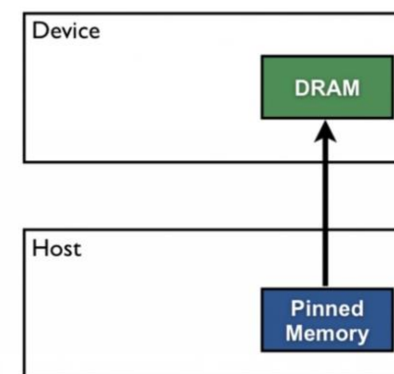
An example of launching a kernel in a stream that isn't the "default stream".

Page-locked / Pinned memory

Section 3.2.6 of the cuda programming guide:

- **To achieve asynchronous behaviour you must use page-locked memory with `cudaMemcpyAsync();`**
- Host memory is usually paged, so run-time system keeps track of where each page is located.
- For higher performance, pages can be fixed (fixed address space, always in RAM), 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
- Page-locked memory is allocated using `cudaHostAlloc()`, or registered by `cudaHostRegister();`



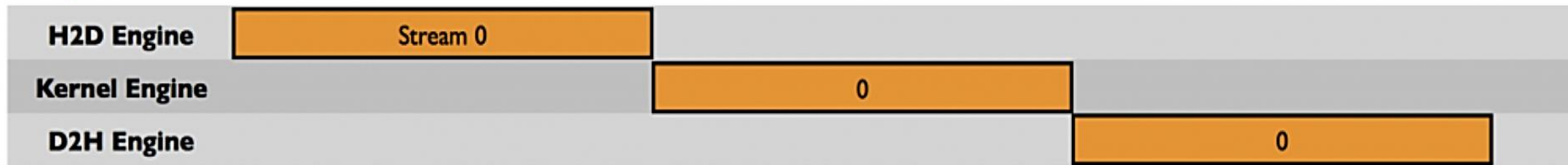


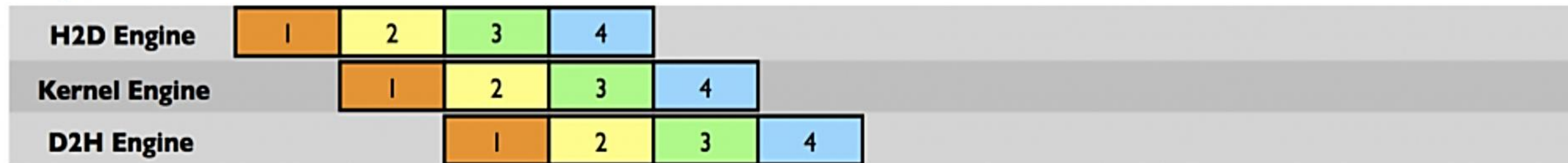
Pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.

https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

Example use

Use multiple streams to increase performance by overlapping memory communication with compute:



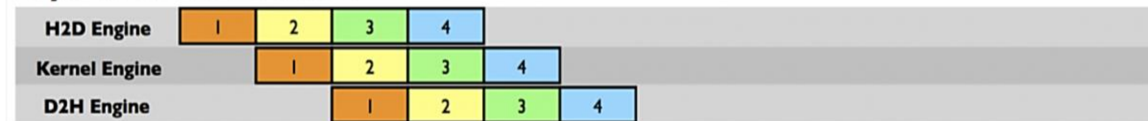


Example use

```
Time for sequential transfer and execute (ms): 43.706047
Time for sequential transfer and execute (ms): 43.709023
Time for sequential transfer and execute (ms): 43.828545
```

```
#include <stdio.h>
#include <cuda.h>

__global__ void kernel(float *a, int offset)
{
  int i = offset + threadIdx.x + blockIdx.x*blockDim.x;
  float x = (float)i;
  a[i] = x;
}

int main(int argc, char **argv)
{
  int blockSize = 128;
  int n = 4 * 1024 * blockSize;
  int bytes = n * sizeof(float);

  // allocate pinned host memory and device memory
  float *a, *d_a;
  cudaMallocHost((void**)&a, bytes); // host pinned
  cudaMalloc((void**)&d_a, bytes);   // device

  float milliseconds; // elapsed time in milliseconds

  // create events and streams
  cudaEvent_t startEvent, stopEvent;
  cudaEventCreate(&startEvent);
  cudaEventCreate(&stopEvent);

  // baseline case - sequential transfer and execute
  memset(a, 0, bytes);
  cudaEventRecord(startEvent,0);
  cudaMemcpy(d_a, a, bytes, cudaMemcpyHostToDevice);
  kernel<<<n/blockSize, blockSize>>>(d_a, 0);
  cudaMemcpy(a, d_a, bytes, cudaMemcpyDeviceToHost);
  cudaEventRecord(stopEvent, 0);
  cudaEventSynchronize(stopEvent);
  cudaEventElapsedTime(&milliseconds, startEvent, stopEvent);
  printf("Time for sequential transfer and execute (ms): %f\n", milliseconds);

  // cleanup
  cudaEventDestroy(startEvent);
  cudaEventDestroy(stopEvent);
  cudaFree(d_a);
  cudaFreeHost(a);

  return 0;
}
```

Example use




```
user@machine:$ ./test 1 && ./test 2 &&  ./test 4 && ./test 8 && \
> ./test 16 && ./test 32 && ./test 64 && ./test 128 && ./test 256
Time for asynchronous transfer and execute (ms): 43.503616
Time for asynchronous transfer and execute (ms): 35.928062
Time for asynchronous transfer and execute (ms): 32.003071
Time for asynchronous transfer and execute (ms): 30.224384
Time for asynchronous transfer and execute (ms): 28.522495
Time for asynchronous transfer and execute (ms): 26.265600
Time for asynchronous transfer and execute (ms): 25.830303
Time for asynchronous transfer and execute (ms): 26.091488
Time for asynchronous transfer and execute (ms): 26.097664
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

__global__ void kernel(float *a, int offset)
{
  int i = offset + threadIdx.x + blockIdx.x*blockDim.x;
  float x = (float)i;
  a[i] = x;
}

int main(int argc, char **argv)
{
  int blockSize = 128, nStreams = atoi(argv[1]);
  int n = 4 * 1024 * blockSize;
  int streamSize = n / nStreams;
  int streamBytes = streamSize * sizeof(float);
  int bytes = n * sizeof(float);

  // allocate pinned host memory and device memory
  float *a, *d_a;
  cudaMallocHost((void**)&a, bytes); // host pinned
  cudaMalloc((void**)&d_a, bytes);   // device

  float milliseconds; // elapsed time in milliseconds

  // create events and streams
  cudaEvent_t startEvent, stopEvent;
  cudaStream_t stream[nStreams];
  cudaEventCreate(&startEvent);
  cudaEventCreate(&stopEvent);
  for (int i = 0; i < nStreams; ++i)
    cudaStreamCreate(&stream[i]);

  // asynchronous version
  memset(a, 0, bytes);
  cudaEventRecord(startEvent,0);
  for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset],
                    streamBytes, cudaMemcpyHostToDevice,
                    stream[i]);
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset],
                    streamBytes, cudaMemcpyDeviceToHost,
                    stream[i]);
  }
  cudaEventRecord(stopEvent, 0);
  cudaEventSynchronize(stopEvent);
  cudaEventElapsedTime(&milliseconds, startEvent, stopEvent);
  printf("Time for asynchronous transfer and execute (ms): %f\n", milliseconds);

  // cleanup
  cudaEventDestroy(startEvent);
  cudaEventDestroy(stopEvent);
  for (int i = 0; i < nStreams; ++i)
    cudaStreamDestroy(stream[i]);
  cudaFree(d_a);
  cudaFreeHost(a);

  return 0;
}
```

The default stream

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

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

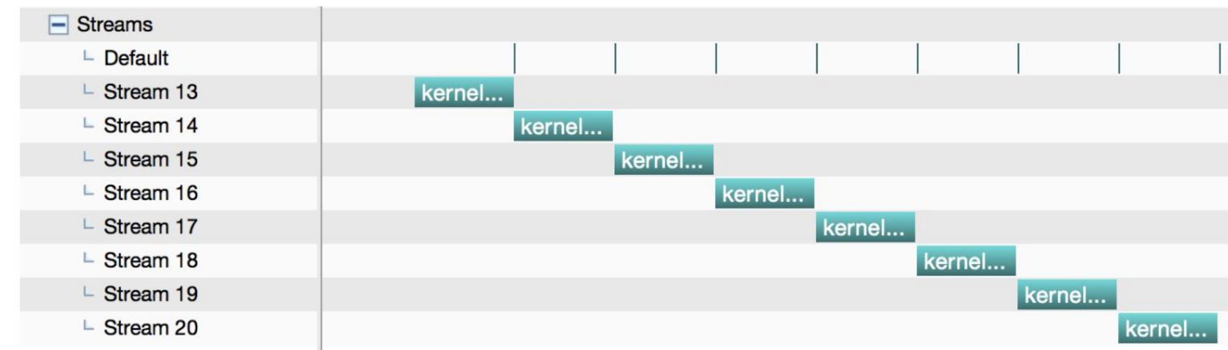
This forces old (bad) behaviour in which a cudaMemcpy or kernel launch on the default stream blocks/synchronizes with other streams.

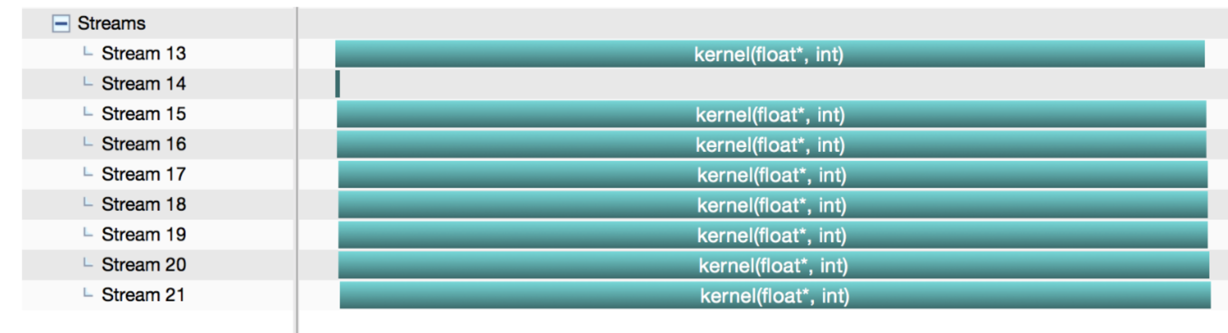
Or `--default-stream per-thread`

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

For more info see the nvcc documentation:

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-cuda-compilation





https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

Practical 11

An example is given in practical 11 for those interested, try with the two different flags:

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

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

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

  // do a Memcpy and launch a dummy kernel on default stream
  cudaMemcpy(d_data,h_data,sizeof(float),
             cudaMemcpyHostToDevice);
  kernel<<<1, 1>>>(d_data, 0);
}
cudaDeviceSynchronize();
```

The default stream

The second (most useful?) 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 separate independent (i.e. non-interacting) default streams for each thread.

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

However “proper” streams within cuda are very versatile and fully featured, so might be worth the time and complexity investment.

```
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

As previously shown, each stream executes a sequence of cuda calls. However to get the most out of your heterogeneous computer you might also want 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

Some useful stream commands are:

`cudaStreamCreate(&stream)`
Creates a stream and returns an opaque “handle” – the “stream variable”.

`cudaStreamSynchronize(stream)`
Waits until all preceding commands have completed.

`cudaStreamQuery(stream)`
Checks whether all preceding commands have completed.

`cudaStreamAddCallback()`
Adds a callback function to be executed on the host once all preceding commands have completed.

http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Stream commands

Functions 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 (doesn’t block the host).

`cudaEventQuery(event)`
Check whether event has occurred.

`cudaEventElapsedTime(time,event1,event2)`
Times between event1 and event2.

Multi-GPU computing

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 (this might not be what you want).
- Visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`.
- The current device can be chosen/set by using `cudaSetDevice()`
- `cudaGetDeviceProperties()` does what it says, and is very useful.
- Each stream is associated with a particular device, which is the “current” device for a kernel launch or a memory copy.
- see `simpleMultiGPU` example in SDK or section 6.2.9 for more information.

https://www.flickr.com/photos/ebastian/8804000077

Multiple devices

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

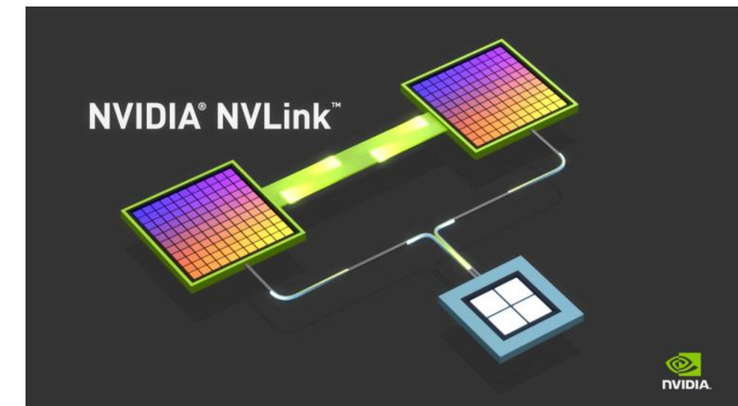
This is the **premise of the NVlink interconnect,** which is much faster than PCIe (**900GB/s P2P on Hopper**).

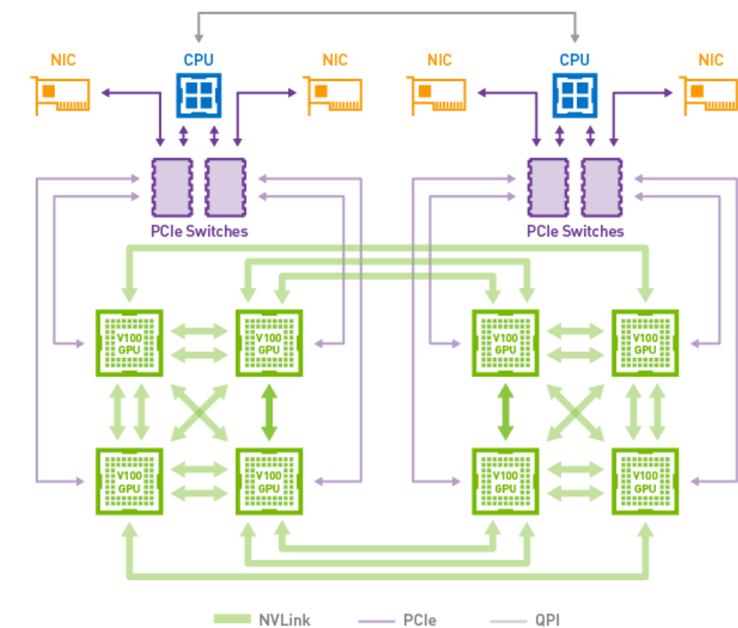
`cudaMemcpy()` can do direct copy from one GPU's memory to another.

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 6.13, "Peer Device Memory Access" in CUDA Runtime API documentation:
https://docs.nvidia.com/cuda/cuda-runtime-api/

https://fuse.wikichip.org/news/1224/a-look-at-nvidias-nvlink-interconnect-and-the-nvswitch/





Multi-GPU computing

Multi-GPU computing exists at all scales, from cheaper workstations using PCIe, to more expensive Quadro / Titan products using fewer NVLink, to high-end NVIDIA DGX servers.

Single workstation / server:

- a big enclosure for good cooling!
- up to 4 high-end cards in 16x PCIe v4 slots – up to 16GB/s interconnect.
- 2x high-end CPUs.
- 2-3kW power consumption – not one for the office!

NVIDIA DGX H100 Deep Learning server:

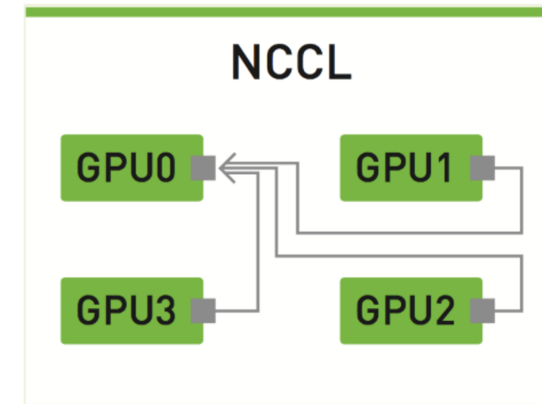
- 8 NVIDIA GH100 GPUs, each with 80GB HBM2.
- 2× 56-core Intel Xeons (Platinum 8480C 2.0 GHz).
- 2 TB RAM memory, 8x 3.84TB NVMe.
- 900GB/s NVlink interconnect between the GPUs.
- ~£379,000???

Multi-GPU computing

How do you use these machines?

This depends on hardware choice:

- For single machines, use shared-memory multithreaded host application.
- For DGX products you must use the NVIDIA Collective Communications Library (NCCL).
- For clusters / supercomputers, use distributed-memory MPI message-passing (NVSHMEM).



https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/

Multi-GPU Example

In the example on the right, we launch two kernels on two different GPUs, GPU 0 and GPU 1.

We use cudaSetDevice() to choose which GPU we are working with.

```
Elapsed time: 1105.097000 ms
Passed Varification!!

Elapsed time: 1103.599000 ms
Passed Varification!!

Elapsed time: 1105.509000 ms
Passed Varification!!

Elapsed time: 1109.698000 ms
Passed Varification!!
```

```
__global__ void kernel(float *d_input, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_input[i]=0;
    for (int j = 0; j < 1000000; j++) {
            d_input[i] += offset;
    }
}

int main(void)
{
    int n = 51200000/8; // number of floats
    int size = n * sizeof(float);
    int offset = 10;

    float *d_a, *d_b;;

    // start timer
    clock_t start = clock();

    // Work on GPU 0
    cudaSetDevice(0);
    cudaMallocManaged(&d_a, size);
    kernel<<<n/256, 256>>>(d_a, offset);

    // Work on GPU 1
    cudaSetDevice(1);
    cudaMallocManaged(&d_b, size);
    kernel<<<n/256, 256>>>(d_b, offset);

    cudaDeviceSynchronize();

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;
    printf("\nElapsed time: %f ms", elapsed);

    for (int i = 0; i < n; i++) {
            if (d_a[i] - d_b[i] != 0) {
                    printf("\nMulti GPU failed at index:\t%d", i);
                    exit(-1);
            }
    }
    printf("\nPassed Varification!!\n");

    cudaFree(d_a);
    cudaFree(d_b);

    return 0;
}
```

Multi-GPU Example

Now, let's change the code so that we launch both kernels on the same GPU.

We see that the execution time approximately doubles. As we would expect for code that is dominated by the kernel execution time.

```
Elapsed time: 1912.208000 ms
Passed Varification!!

Elapsed time: 1898.162000 ms
Passed Varification!!

Elapsed time: 1902.591000 ms
Passed Varification!!

Elapsed time: 1902.241000 ms
Passed Varification!!
```

```
__global__ void kernel(float *d_input, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_input[i]=0;
    for (int j = 0; j < 1000000; j++) {
            d_input[i] += offset;
    }
}

int main(void)
{
    int n = 51200000/8; // number of floats
    int size = n * sizeof(float);
    int offset = 10;

    float *d_a, *d_b;;

    // start timer
    clock_t start = clock();

    // Work on GPU 0
    cudaSetDevice(0);
    cudaMallocManaged(&d_a, size);
    kernel<<<n/256, 256>>>(d_a, offset);

    // Work on GPU 0
    cudaSetDevice(0);
    cudaMallocManaged(&d_b, size);
    kernel<<<n/256, 256>>>(d_b, offset);

    cudaDeviceSynchronize();

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;
    printf("\nElapsed time: %f ms", elapsed);

    for (int i = 0; i < n; i++) {
            if (d_a[i] - d_b[i] != 0) {
                    printf("\nMulti GPU failed at index:\t%d", i);
                    exit(-1);
            }
    }
    printf("\nPassed Varification!!\n");

    cudaFree(d_a);
    cudaFree(d_b);

    return 0;
}
```

GPUDirect

What is GPUDirect

- It enhances data movement and access for NVIDIA GPUs.

Key Features of GPUDirect

- GPUDirect Storage: Provides a direct data path between local or remote storage, such as NVMe or NVMe over Fabric (NVMe-oF).
- GPUDirect RDMA: Enables peripheral PCIe devices direct access to GPU memory.
- GPUDirect Peer to Peer (P2P): Allows for direct communication between NVIDIA GPUs in remote systems.

Benefits of GPUDirect

- Eliminates unnecessary memory copies.
- Decreases CPU overheads.
- Reduces latency.
- Results in significant performance improvements.

Simple GPUDirect Example

Here we use cudaSetDevice() to pick which GPU we are working on.

1. Initialise some values, vector length, offset, etc.
2. malloc n x sizeof(float) on the host (h_a).
3. Set values of h_a[i] = i + offset.
4. Start a clock
5. Allocate managed memory on GPU 0
6. Perform kernel computation on GPU 1
7. Check to see if our GPU and CPU results are the same.
8. Print the time taken.

```
__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 51200000/8; // number of floats
    int size = n * sizeof(float);
    int offset = 10;

    float *d_a, *h_a;;
    h_a = (float*)malloc(size);

    for (int i = 0; i < n; i++) {
        h_a[i] = i + offset;
    }

    // start timer
    clock_t start = clock();

    // Allocate memory on GPU 0
    cudaSetDevice(0);
    cudaMallocManaged(&d_a, size);

    // Launch kernel on GPU 1
    cudaSetDevice(1);
    kernel<<<n/256, 256>>>(d_a, offset);
    cudaDeviceSynchronize();

    for (int i = 0; i < n; i++) {
            if (h_a[i] - d_a[i] != 0) {
                    printf("\nMulti GPU failed at index:\t%d", i);
                    exit(-1);
            }
    }
    printf("\nPassed Varification!!");

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;
    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

Simple GPUDirect Example

For the code to complete correctly, the runtime environment needs to move data GPU 0 to GPU 1.

```
Passed Varification!!Elapsed time: 288.554000 ms

Passed Varification!!Elapsed time: 273.861000 ms

Passed Varification!!Elapsed time: 271.562000 ms

Passed Varification!!Elapsed time: 272.309000 ms

Passed Varification!!Elapsed time: 270.529000 ms
```

```
__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 51200000/8; // number of floats
    int size = n * sizeof(float);
    int offset = 10;

    float *d_a, *h_a;;
    h_a = (float*)malloc(size);

    for (int i = 0; i < n; i++) {
        h_a[i] = i + offset;
    }

    // start timer
    clock_t start = clock();

    // Allocate memory on GPU 0
    cudaSetDevice(0);
    cudaMallocManaged(&d_a, size);

    // Launch kernel on GPU 1
    cudaSetDevice(1);
    kernel<<<n/256, 256>>>(d_a, offset);
    cudaDeviceSynchronize();

    for (int i = 0; i < n; i++) {
            if (h_a[i] - d_a[i] != 0) {
                    printf("\nMulti GPU failed at index:\t%d", i);
                    exit(-1);
            }
    }
    printf("\nPassed Varification!!");

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;
    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

Simple GPUDirect Example

Here we work only on GPU 0. We have exactly the same code but this time we pass cudaSetDevice() 0 in both calls.

```
Passed Varification!!Elapsed time: 185.163000 ms

Passed Varification!!Elapsed time: 179.100000 ms

Passed Varification!!Elapsed time: 178.624000 ms

Passed Varification!!Elapsed time: 177.794000 ms

Passed Varification!!Elapsed time: 177.354000 ms
```

```
__global__ void kernel(float *d_a, int offset)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    d_a[i]  = i + offset;
}

int main(void)
{
    int n = 51200000/8; // number of floats
    int size = n * sizeof(float);
    int offset = 10;

    float *d_a, *h_a;;
    h_a = (float*)malloc(size);

    for (int i = 0; i < n; i++) {
        h_a[i] = i + offset;
    }

    // start timer
    clock_t start = clock();

    // Allocate memory on GPU 0
    cudaSetDevice(0);
    cudaMallocManaged(&d_a, size);

    // Launch kernel on GPU 1
    cudaSetDevice(0);
    kernel<<<n/256, 256>>>(d_a, offset);
    cudaDeviceSynchronize();

    for (int i = 0; i < n; i++) {
            if (h_a[i] - d_a[i] != 0) {
                    printf("\nMulti GPU failed at index:\t%d", i);
                    exit(-1);
            }
    }
    printf("\nPassed Varification!!");

    // stop timer
    clock_t stop = clock();

    // calculate elapsed time in milliseconds
    double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC * 1000;
    printf("Elapsed time: %f ms\n", elapsed);

    cudaFree(d_a);
    free(h_a);

    return 0;
}
```

MPI approach

In the MPI approach:

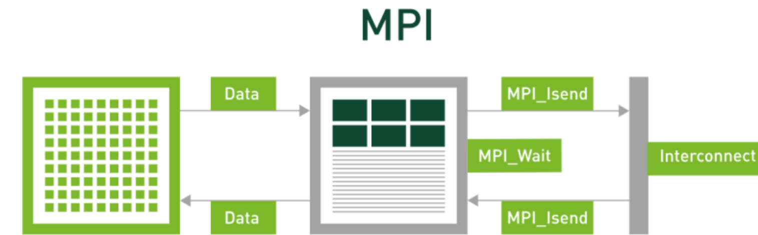
- One GPU per MPI process (nice and simple).
- Distributed-memory message passing between MPI processes (tedious but not difficult).
- Scales well to very large applications.
- Main difficulty is that the user has to partition their problem (break it up into separate large pieces for each process) and then explicitly manage the communication.
- Note: should investigate GPU Direct for maximum performance in message passing.

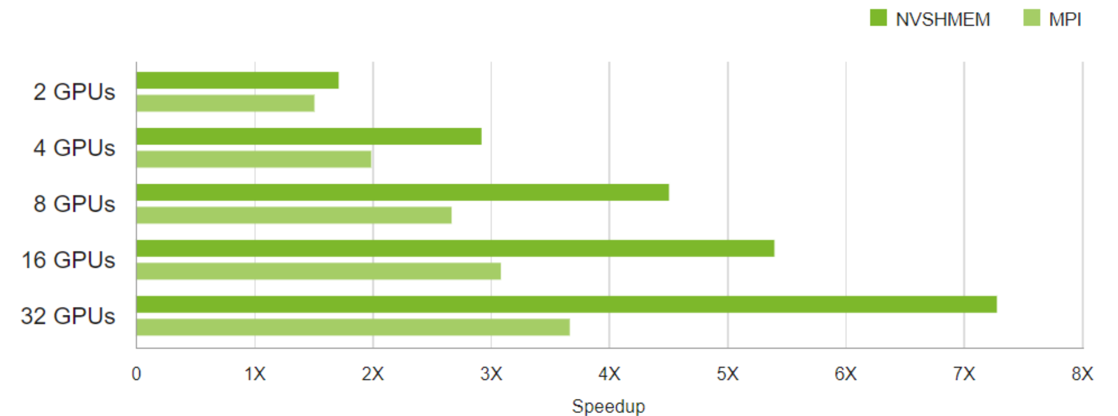
NVSHMEM

Based on OpenSHMEM.

MPI orchestrates data transfers using the CPU.

In contrast, NVSHMEM uses asynchronous, GPU-initiated data transfers.





https://developer.nvidia.com/nvshmem

Multi-user support

What if different processes try to use the same device?

The behaviour of the device depends on the system compute mode setting (section 3.4):

In “default” mode, each process uses the fastest device:

- This is good when one very fast card, and one very slow card.
- But not very useful when you have two identical fast GPUs (one sits idle).

In “exclusive” mode, each process is assigned to first unused device;
<u>However code will return an error if none are available</u>.

`cudaGetDeviceProperties()` reports the mode setting

The mode can be changed by a user account with sys-admin privileges using the `nvidia-smi` command line utility.

Summary

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.

Handout for 2024
Some tips and tricks

Loose ends – Loop unrolling

Section 10.37 (of the programming guide):
loop unrolling, If you have a loop:

```
for (int k=0; k<4; k++) a[i] += b[i];
```

Then nvcc will automatically unroll this to give:

```
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
```

This is a standard compiler trick to avoid the cost of incrementing and looping.

The pragma

```
#pragma unroll 5
```

will also force unrolling for loops that do not have explicit limits.

Loose ends – const __restrict__

Section 10.2.6 (of the programming guide):

`__restrict__` keyword

The qualifier asserts that there is no overlap (in memory space) between `a,b,c`, for example we do not have:

```
a[i]=q[i]
b[i]=q[i+1]
```

(you have no pointer aliasing) so the compiler can perform more optimisations.

The following blog post demonstrates how this can achieve a good speed increase:

https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/#disqus_thread

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
               float* __restrict__ c) {

        for (i=1; i<N; i++) {
                a[i] = b[i] + c[i];
        }
        ...
}
```

Loose ends - volatile

Section 17.5.3.3 (of the programming guide):
`volatile` keyword

Tells the compiler **the variable may change at any time**, so not to re-use a value which may have been loaded earlier and apparently not changed since.

This can sometimes be important when using shared memory because the compiler can optimize locations in shared memory by locating them in registers (but register scope is specific to a single thread), for any thread.

Loose ends - Compilation

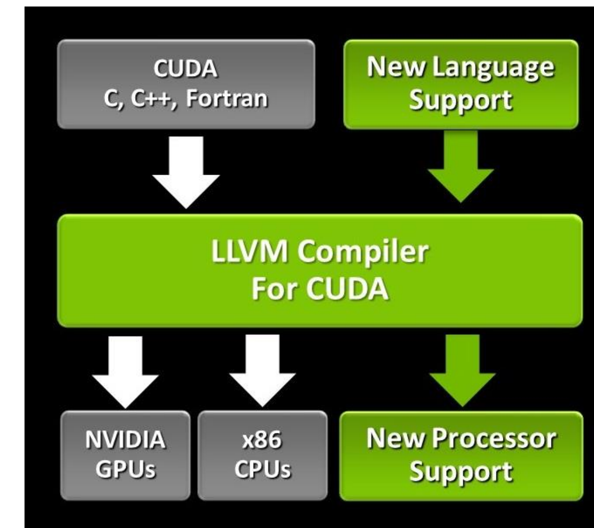
Compiling:

The `Makefile` for first few practicals uses `nvcc` to compile both the host and the device code.

Internally `nvcc` uses `gcc` for the host code (at least by default). The device code compiler is based on the open source LLVM compiler.

It often makes sense to use different compilers, for example `icc` which is for host code which does not have kernel launches.

To do this you must use the `-fPIC` flag to produce position-independent-code (this just generates machine code that will execute properly, independent of where it's held in memory).



https://developer.nvidia.com/cuda-llvm-compiler

Loose ends - Compilation

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)
```

Loose ends - Compilation

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)
```

Loose ends - Compilation

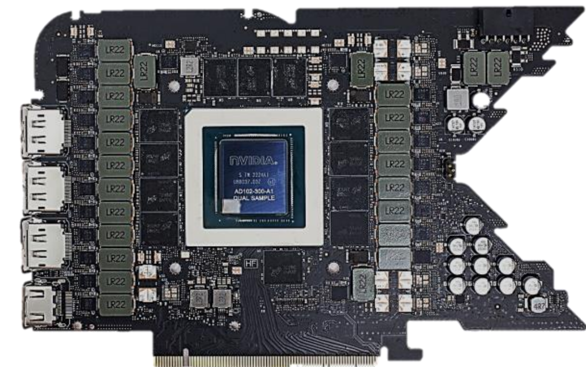
Other useful compiler options:

`-arch=sm_80`
This specifies GPU architecture (in this case sm_80 is for Ampere A100).

`-maxrregcount=n`
This asks the compiler to **generate code using at most n registers**; the compiler may ignore this if it's not possible, but it may also increase register usage up to this limit.

This is less important now since threads can have up to 255 registers, but can be useful in some instances to reduce register pressure and enable more thread blocks to run.

or



Loose ends – Compilation

Launch bounds (10.36):

`-maxrregcount` is given as an argument to the compiler (`nvcc`) and modifies the default for all kernels.

A per kernel approach can be taken by using the `__launch_bounds__` qualifier:

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,minBlocksPerMultiprocessor)
MyKernel(...) {
…
}
```