Learning outcomes

In this second lecture we will look at memory systems and the differences between CPU and GPU memory hierarchies.

You will learn about:

• Some of the differences between CPUs and GPUs.
• The basic concepts of memory hierarchy and how to achieve good bandwidth throughput.
• Some more on the design of a GPU and how you achieve good memory bandwidth performance.
• Different types of GPU memory and different types of variables.
Modern memory systems

Processing cores become ever faster, but the rate that a computer can move data from memory to cores hasn't developed as quickly. This means that on a modern processor data movement is far more costly than computations.

So for a program to achieve peak performance on a modern processor, great care has to be taken to minimize data movement.

Differences between CPUs and GPUs

GPUs have a different design to CPUs. Their complexity is greatly reduced (they don't have the intelligence of a CPU, no prefetch, no branch prediction...). GPUs dedicate most of their transistors to processing (right). This makes GPUs ideal for compute-intensive, highly parallel computation.

More specifically, GPUs excel at data-parallel computations. Just like the AVX (SIMD) units in CPUs.

As mentioned in lecture one - GPUs have a reasonable amount of High Bandwidth Memory (HBM) onboard. This memory has greater bandwidth than server RAM, but there is less of it.

Remember though - GPUs attach to a server via the PCIe bus (apart from NVLink power systems), so this ultimately limits communication speed between a CPU and GPU.
A quick look at computational throughput

A CPU is a multi-core device (32+ cores now) and each core has one (or two) vector processing units, these are called Advanced Vector Extensions (AVX) units, in intel CPUs.

These units are capable of executing a Single Instruction on Multiple Data (SIMD) elements at the same time (in parallel). A program that exploits these vector features is said to be "vectorised".

Skylake has AVX512 vector units, meaning that the vectors are 512 bits long, so can store 16 single precision (4 byte) numbers in each. They can perform at most two instructions at once by issuing a fused multiply add (FMA) instruction (see right). It’s these units, along with the high core count that allow modern CPUs to perform very large and complex computations very quickly.

CPU computational throughput

One of the largest Intel CPUs at the moment is the Xeon Platinum 8180.

This has:

• 28 Cores
• Runs at a frequency of 2.5 GHz
• Has 2x AVX512 units per core (so can hold 16 SP values each)
• Can execute 2x operations per execution
• Costs ~ £7,000.

The peak computational performance is measured by the number of Floating Point Operations per second (flops) that can be performed.

So it’s peak single precision performance is given by the calculation on the right.
GPU computational throughput

One of the largest NVIDIA GPUs at the moment is the V100.

This has:

- 5120 Cores
- Runs at a frequency of 1.37 GHz
- Can execute 2x operations per execution
- Costs ~ £8,800.

Comparing this to our Xeon CPU we see that the peak computational performance is about 3x higher, but costs just 25% more.

It’s important to note though that the CPU is far more flexible that the GPU and can outperform the GPU for certain computational tasks.

Memory throughput

Whilst it is great to have such high computational throughput, it’s effectively useless if we can’t get data to and from processing cores quickly enough to exploit it.

So a key challenge in modern computer architecture design is to ensure processing cores can be fed with data to keep them busy.

This becomes difficult when we consider large applications that need lots of memory, and the fact that very fast memory is very expensive to produce.

This necessitates a hierarchical memory design.
Modern computers can have several different types of memory all of which have different bandwidths (bandwidth is a measure of the amount of data you can get from point A to point B in unit time).

They are arranged in a hierarchy. From lots of slower RAM up to very few, but very fast registers which are located closest to the CPU processing cores.

The diagram on the right shows the different bandwidths for a modern Intel Skylake CPU. Note that all of the caches (smaller areas of fast memory) are located on the CPU.

As noted on our previous slide the area of memory where we can store the most data is the RAM, but this has lower bandwidth.

So if we need to move our data from RAM to processing cores many times our code would run slowly. This is where Cache helps. Caches exploit data locality.

- Temporal locality: a data item just accessed is likely to be used again in the near future, so keep it in the cache.
- Spatial locality: neighbouring data is also likely to be used soon, so load them into the cache at the same time using a wide bus (like a multi-lane motorway).

It is these wide lanes that feed processing cores with data.
Caches

In modern computer architectures, the cache line is the basic unit of data transfer; the typical size is 64 bytes ≡ 8 × 8-byte items.

With a single cache, when the CPU loads data into a register:
- it looks for line in cache.
- if there (cache hit), it gets the data.
- if not (cache miss), it gets entire line from main memory, displacing an existing line in cache (usually least recently used).

When the CPU stores data from a register:
- same procedure.

The importance of locality - CPUs

Let’s consider a typical workstation CPU – Intel Bronze 3104 (6 cores, 1.7Ghz, AVX512):
~50 Gflops per core.
40 GB/s L3 ←→ L2 cache bandwidth
64 bytes/line

\[
\begin{align*}
\text{40GB/s (64 bytes per line) } & \leq 600M \text{ cache line/s} \\
\text{(600M cache lines/s)/(8 doubles per cache line)} & \leq 75 \text{ double/s} \\
\text{At worst, each flop requires } 2 \text{ inputs and has } 1 \text{ output (e.g., addition)}, \text{ forcing loading of } 3 \text{ lines} \rightarrow \\
\text{(600M cache lines/s)/(2 cache lines for input, 1 cache line for output)} & \leq 200 \text{ Mflops} \\
\text{If all 8 doubles/line are used, then this increases to } 1.6 \text{ Gflops.} \\
\end{align*}
\]

To get up to 50 Gflops needs temporal locality, re-using data already in the L2 cache.
Recall the V100 architecture

The V100 has 80 SMs.
Each SM has:
A 256KB register file.
A unified shared memory / L1 cache – 128KB; hence 10MB total.
A 6MB L2 cache.
16/32 GB HBM2 device memory.

GPU memory architecture

- GPU L1/L2 cache lines are 128 bytes wide.
- These are formed from 4 sectors 32 bytes wide.
- Importantly, the cache "management" granularity is one cache line.
- Each cache line contains (surprise, surprise) 32 floats (one warp).
- Or 16 doubles.
- V100 has a 4096-bit memory path from HBM2 device memory to L2 cache, achieving up to 900 GB/s bandwidth.
- In contrast, GeForce RTX cards: 384-bit memory bus from GDDR6 device memory to L2 cache, achieving up to 600 GB/s bandwidth.

GPUs do not have global cache coherency (unlike CPUs), so should (almost) never have different blocks updating the same global array elements.
Previously we considered the CPU memory hierarchy (slide 9).

Let's compare this to that of GPUs.

We see that although GPUs have greater bandwidth, they have less cache and (importantly) greater latencies (the time taken to get data from memory to the processing cores).

Revisiting practical 1

Let's take a moment to look at the memory access pattern used in `my_first_kernel` from practical one.

- 32 threads in a warp will address neighbouring elements of array `x[]`
- if the data is correctly “aligned” so that `x[0]` is at the beginning of a cache line, then `x[0]` – `x[31]` will be in the same cache line.
- This is known as a “coalesced” transfer (or access). Hence we get perfect spatial locality.

An example of a good data access pattern.
An example of a bad GPU kernel

The example kernel on the right is exactly the same as the kernel used in practical 1, apart from the $x[i]$ array index is multiplied by 1000.

In this case, different threads within a warp will access widely spaced elements of array $x[i]$, a “stided” array access.

Meaning each access will be to a different cache line (accesses are no longer aligned and next to each other).

Each cache line accessed read writes 32 elements of the cache line, but modifies only one element($x[1000 \cdot tid]$).

In this example we are effectively using 1/32 of our available bandwidth, performance will be terrible!

Coalesced and uncoalescence accesses

Global memory access happens in transactions of 32 or 128 bytes (cache line).

So to achieve peak memory bandwidth we need to use all of the data stored in the cache line.

Coalesced memory access
A warp of threads access adjacent data (see right) in a cache line. In the best case this results in one memory transaction (best bandwidth).

Uncoalesced memory access
A warp of threads access scattered data all in different cache lines.
This would result in 32 different memory transactions (giving poor bandwidth).
Global arrays

So far, our GPU kernels have used global / device arrays. These are characterised by:

• Data is held in the large device memory.
• The array is allocated by host code - `cudaMalloc(void **d_x, nsize*sizeof(float));`
• Pointers to device arrays are held by host code and passed into kernels
  `my_first_kernel<<<nblocks,nthreads>>>(d_x);`
• They continue to exist until freed by host code `cudaFree(d_x);`
• Since thread blocks execute in an arbitrary order, if one block modifies an array element, no other block should read or write to that same element.

Global variables

Global variables can also be created by declarations with global scope within the kernel code file:

```c
__device__ int reduction_lock=0;
__global__ void kernel_1(...) {
  ... Kernel code ...
}
__global__ void kernel_2(...) {
  ... Kernel code ...
}
```

In this case the integer variable `reduction_lock` can be seen and accessed by `kernel_1` and `kernel_2` just like C.
Global variables

Looking at this is more detail:

• The **_device_** prefix tells **nvcc** this is a global variable on the GPU, not the CPU.

• The variable can be read and modified by any kernel.

• Its lifetime is the lifetime of the whole application.

It’s also possible to can also declare arrays of fixed size.

Importantly: To read/write from/to global device variables from host code you must use special routines in your host code:

• `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`

This is something that I don’t tend to use in my CUDA codes.

```c
__device__ int reduction_lock = 0;
__global__ void kernel_1(...) {
    ... Kernel code ...
}
__global__ void kernel_2(...) {
    ... Kernel code ...
}
```

Constant variables

Constant variables are very similar to global variables, except that they can’t be modified by kernels:

• They are defined with global scope within the kernel file using the prefix **__constant__**

• They are initialised by the host code using `cudaMemcpyToSymbol`

I use constants a lot in my codes, typically for smaller lookup tables. They have the advantage of very fast access and so can be more efficient that relying on device memory or caching.

It’s worth noting that as shared memory becomes more sophisticated (broadcasts and multicasts) for some applications shared memory can be used to achieve the same performance (however all resources are limited so use them wisely).

There is an example of using constant memory in practical 2.
Constant memory

Each SM has a read-only constant cache that is shared by all cores in the SM. This is used to speed up reads from the constant memory space, which resides in device memory.

The constant memory is just 64KB (on all GPU architectures so far). But the big benefit of it is that each SM has a 8KB cache [4KB on Pascal compute capability 6.0].

Meaning that when all threads read the same constant, its almost as fast as register access.

Also it doesn’t use register (or shared memory) space, so it’s very helpful in minimising the total resources (e.g. registers or shared memory) that your code requires.

Constant variables

A constant variable has its value set at run-time.

But often code has plain constants with values that are known at compile-time, for example:

```c
#define PI 3.1415926f
a = b / (2.0f * PI);
```

You should always leave these as they are – the compiler will embed these are actual numbers into the executable code so they don’t use up any registers.

This also provides more room for compile-time optimisation.

Don’t forget the f at the end if you want single precision;
in C/C++

```c
single × double = double
```

```c
```
Within each kernel, by default, individual variables are assigned to registers:

```c
__global__ void lap(int I, int J, float *u1, float *u2) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int id = i + j*I;
    if (i==0 || i==I-1 || j==0 || j==J-1) {
        u2[id] = u1[id]; // Dirichlet b.c.'s
    } else {
        u2[id] = 0.25f * ( u1[id-1] + u1[id+1] + u1[id-I] + u1[id+I] );
    }
}
```

### Registers

Device memory

- 64K 32-bit registers per SM (So 256KB register file per SM, a CPU in contrast has a few (1-2)KB per core).
- Up to 255 registers per thread.
- Up to 2048 threads (apart from Turing - 1024), at most 1024 threads per thread block.
- If a code uses the maximum number of registers per thread (255) and an SM has 64K of registers then the SM can support a maximum of 256 threads.
- If we use the maximum allowable number of threads per SM (2048), and a SM has 64K of registers, then each thread can use at most 32 registers per thread.

**Hence there is a big difference between “fat” threads, which use lots of registers, and “thin” threads that require very few!**

### Specifics of registers

Lots of useful information can be found in the NVIDIA cuda programming guide, specifically in this case Table 14. Technical Specifications per Compute Capability:

- 64K 32-bit registers per SM (So 256KB register file per SM, a CPU in contrast has a few (1-2)KB per core).
- Up to 255 registers per thread.
- Up to 2048 threads (apart from Turing - 1024), at most 1024 threads per thread block.
- If a code uses the maximum number of registers per thread (255) and an SM has 64K of registers then the SM can support a maximum of 256 threads.
- If we use the maximum allowable number of threads per SM (2048), and a SM has 64K of registers, then each thread can use at most 32 registers per thread.

**Hence there is a big difference between “fat” threads, which use lots of registers, and “thin” threads that require very few!**
Register spills

What happens if your application needs more registers? They “spill” over into L1 cache, and from there to device memory – for early hardware (Fermi) further details can be found here: https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf (Paulius Micikevicius, NVIDIA).

The memory for spills is often referred to as local memory (because each thread has its own private memory, which is stored in global/device memory). Local memory operates as follows:

A store writes a line to L1
• if evicted, that line is written to L2.
  • The line could also be evicted from L2, in which case it’s written to DRAM.

A load requests the line from L1
• if a cache hit, the operation is complete.
  • if a cache miss, then requests the line from L2.
  • if a cache miss in L2, then requests the line from DRAM.

In most instances the application suffers from the latency and bandwidth implications associated with spills

Local arrays

What happens if your application uses a small array?

```c
__global__ void lap(float *u) {
    float ut[3];
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    for (int k=0; k<3; k++) ut[k] = u[tid+k*gridDim.x*blockDim.x];
    for (int k=0; k<3; k++)
}
```
Local arrays

In simple cases like this the compiler converts the array to scalar registers, so our previous code would become:

```c
__global__ void lap(float *u) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    float ut0 = u[tid+0*gridDim.x*blockDim.x];
    float ut1 = u[tid+1*gridDim.x*blockDim.x];
    float ut2 = u[tid+2*gridDim.x*blockDim.x];

    u[tid+0*gridDim.x*blockDim.x] = A[0]*ut0 + A[1]*ut1 + A[2]*ut2;
}
```

In more complicated cases, it puts the array into device memory this is because registers are not dynamically addressable – compiler has to specify exactly which registers are used for each instruction still referred to in the documentation as a “local array” because each thread has its own private copy held in L1 cache by default, may never be transferred to device memory 48kB of L1 cache equates to 12k 32-bit variables, which is only 12 per thread when using 1024 threads beyond this, it will have to spill to device memory.
As mentioned in lecture one, shared memory is a used managed cache.

- All threads within a thread block can see shared memory allocated to that thread block, but cannot see shared memory for another thread block.
- Shared memory is an extremely useful resource. It’s possible to allocate up to 48KB (96KB for Volta, 64KB for Turing) per thread block. It allows you, the programmer, to cache data as you need it. This is great because it avoids the problems of having to try to coax a standard cache to do what you want.
- It has very high bandwidth, on a V100 this is about 14TB/s! (Turing is half this because it takes two clock cycles to perform a load).
- I try to use shared memory in all of my codes rather than relying on L1 cache. I always see better results compared to the same code using L1 cache.

Shared memory

Shared memory is declared in a kernel using the prefix `__shared__` some examples:

```c
__shared__ int x_dim;
__shared__ float x[128];
```

This declares variables (data) that is to be shared between all of the threads in the thread block – any thread (within the thread block) can set its value, or read it.

Further benefits of shared memory are:

- It can be essential for operations requiring communication between threads (e.g. summation in lecture 4).
- As mentioned it’s extremely useful for data re-use (as an alternative to local arrays in device memory).
Shared memory

If a thread block has more than one warp, it’s not pre-
determined when each warp will execute its
instructions. So warp 1 could be many instructions
ahead of warp 2, or well behind it.

Given this, in nearly all cases, you must use thread
synchronisation to ensure the correct use of shared
memory.

CUDA provides a very useful instruction for this –

```c
syncthreads();
```

“syncthreads” inserts a “barrier” into your code.

It ensures that no thread (or warp) is allowed to proceed
beyond this point until all other threads in the thread
block have reached it (like a roll call on a school outing).

```c
#define BLOCKDIMX 128
__shared__ float ut[BLOCKDIMX];
__global__ void lap(float *u) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    ut[threadIdx.x] = u[tid];
    __syncthreads();
    if(threadIdx.x < BLOCKDIMX - 32)
        u[tid] = (ut[threadIdx.x] - ut[threadIdx.x + 32]);
}
```

In the above example, without `__syncthreads()

warp 1 could read in data warp 2 should have
loaded, but hasn’t yet done so.

So far we have looked at statically-allocated shared
memory, this is where we know the size we require at
compile time.

It’s also possible to create dynamically allocated shared-
memory arrays, so that the shared memory array is
allocated at the point of kernel launch, but this is more
complex.

It does this, the total size for the shared memory required is
specified by an optional third argument when
launching the kernel:

```c
kernel<<<blocks,threads,shared_bytes>>>(...)
```

Using this within the kernel function is complicated/tedious!
(see B.2.3 in Programming Guide).

```c
extern __shared__ float array[];
__device__ void func() {
    short array0 = (short*)array;
    float array1 = (float*)&array0[128];
    int array2 = (int*)&array1[64];
}
```

The example from the CUDA programming
guide – pointers have to be aligned to the
type they point to. Essentially you are
mapping out the memory space by hand.

This is equivalent to:

```c
short array0[128];
float array1[64];
int array2[256];
```
Read-only arrays

With "constant" variables, each thread reads the same value.

In other cases, we have arrays where the data doesn't change, but different threads read different items.

In this case, it's possible to get improved performance by telling the compiler by declaring global array with `const __restrict__` qualifiers so that the compiler knows that it is read-only.

Non-blocking loads/stores

How does the following code execute?

```c
__global__ void lsp(float *u1, float *u2) {
    float a, b, c;
    a = u1[threadIdx.x + blockIdx.x*blockDim.x];
    //
    do something..
    
    c = b*a; // Stalls on load (if not already complete)
    u2[threadIdx.x + blockIdx.x*blockDim.x] = c;
    //
}
```

Load doesn't block until needed; store also doesn't block.
Active blocks per SM

Each thread block requires certain resources to execute:

- Threads
- Registers (registers per thread × number of threads)
- Shared memory (static + dynamic)

Together these determine how many blocks can be run simultaneously on each SM – up to a maximum of 32 blocks (16 for Kepler and below, 16 for Turing).

Active blocks per SM

Some general advice:

- The number of active threads depends on number of registers each needs – try to reduce register usage when possible.
- It’s good to have at least 4 active blocks, each with at least 128 threads (maybe 3 and 96 for Volta and above) to hide latencies.
- You will be forced to work with a smaller number of blocks when each needs lots of shared memory – in this case fat threads that have lots of ILP sometimes perform well.
- Try to work with a larger number of blocks when they don’t need (much) shared memory.
Active blocks per SM

As a rough guide on Volta:

- Maybe 3 big blocks (3 x 128 threads) if each block needs a lot of shared memory.
- Maybe 12 small blocks (96 threads) if no shared memory needed.
- Or 4 small blocks (128 threads) if each thread needs lots of registers.

However the best advice is to experiment.

We tend to parameterise our codes using `#define` and then use a script to run through different values of threads per block (for instance) to determine the best performing configuration.

Key reading

CUDA Programming Guide, version 10.1:
- Appendix B.1.8.4 – essential
- Chapter 3, sections 3.2.1-3.2.3

Other reading:
- Web article on caches [http://lwn.net/Articles/252125/](http://lwn.net/Articles/252125/)
- "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System": [http://portal.acm.org/citation.cfm?id=1637764](http://portal.acm.org/citation.cfm?id=1637764)
What have we learnt?

In this lecture we have discussed some of the differences between CPUs and GPUs.

We have looked at the basic concepts of memory hierarchy and the differences between CPU and GPU memories.

We’ve talked about coalesced and uncoalesced memory access patterns and how to achieve good bandwidth throughput.

We looked at different types of GPU memory and finally different types of variables.