SOFT354 - CUDA memory

Date: 06-10-16

Memory matters

- Compute to global memory access ration
  - How many floating point operations does a program do for every global memory access operation
- GPUs have > 200GB/s global memory bandwidth
  - 50,000 floating point values per second
- Copying from CPU to the GPU is even slower than global memory access
- We need to minimise access to globabl memory

Host and device memory

- **Host memory** is the PC's normal RAM
- **Device memory** is the "video" RAM on the GPU
- Code in a *kernel* can **only** access **device memory**
- Host code (main() function) can access device memory using functions like cudaMemcpy

Sin(x) example

- In the workshop we wrote a program to compute sin(x) for a lots of values of x
- each **thread** computes one value
- two pairs of arrays:
  - One on the host: **input, result**
  - One on the device: **d__input, d__result**

Steps: 1. Serially initialise input data on host 2. Copy input to GPU - Using cudaMemcpy 3. Kernel runs, computes sin(x) in parallel - On each thread 4. Copy result beack to host to work with

Static vs Dynamic allocation

- Arrays can be *statically* or *dynamically* allocated
- With **static allocation** the sizxe of the array must be known at compile time
  - Space is reservered in the program's memory map
  - *float staticArray[10];*
- With **Dynamic allocation** the size of the array can be calculated at runt

  - **float* dynamicArray = (float)malloc(nsizeof(float));**

- how many bites you want
- **Array is just a pointer pointers are just integers**
- In both cases the variable is just the momory address of the first element in the array

	Allocate in Host Memory	Allocate in Device Memory
Static	`float h_array[10];`	`__device__ float d_array[10];`
Dynamic	`float* h_array = (float*)malloc(10*sizeof(float));`	`float* d_array; cudaMalloc(&d_array, 10*sizeof(float));`

- `cudaMalloc` returns a `cudaError_t`
  - Not the address of the allocatied memory
  - So you need to pass a pointer to a pointer
- If you use `malloc` inside a kernel
  - It will allocate momory on the device

Allocating memory in kernels is rare

- A common mistake is put the cudaMalloc call inside the kernel
- This is possible but not desired
- Every thread would allocate enough space for the whole array
  - would run out of space
- Instead we allocate enough space to hold one copy of the array in device memory
- and each thread accesses a different bit of it

Freeing dynamic memory

- Dynamically allocated memory isn't automatically cleaned up
- free(array)
- cudafree(array)

Copying to staic device arrays

- When you use a static allocation for device memory

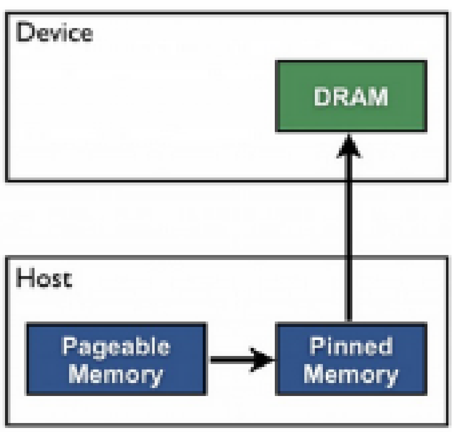
- the varaible isn't a pointer to a memory address on the GPU, it's a symbol
- This means you can't directly pass the variable into cudaMemcpy
  - Use cudaGetSymbolAddress
  - cudaMemcpyToSymbol/cudaMemcpyFromSymbol

Accessing static memory from a kernel

- if deivce memory was allocated statically in the host code
- It allows you access it *globally* on the device
  - i.e. not have to pass into functions
- **Better practise**

Pinned memory

- Technique for speeding up RAM < - > memory transfers
- Transfers between the RAM and GPY can be very fast
  - They use direct memory access to do the copy without involving the CPU
- **but**, operating systems used **paged virtual memory**
- The address you get from malloc() doesn't correspond to a physical address in RAM
  - but to a "page" that can be moved around
  - Which is no good for a DMA transfer
- So when transferring from RAM to GPU
  - Cuda first copies the data into **pinned** RAM
    - Where it doesn't get moved



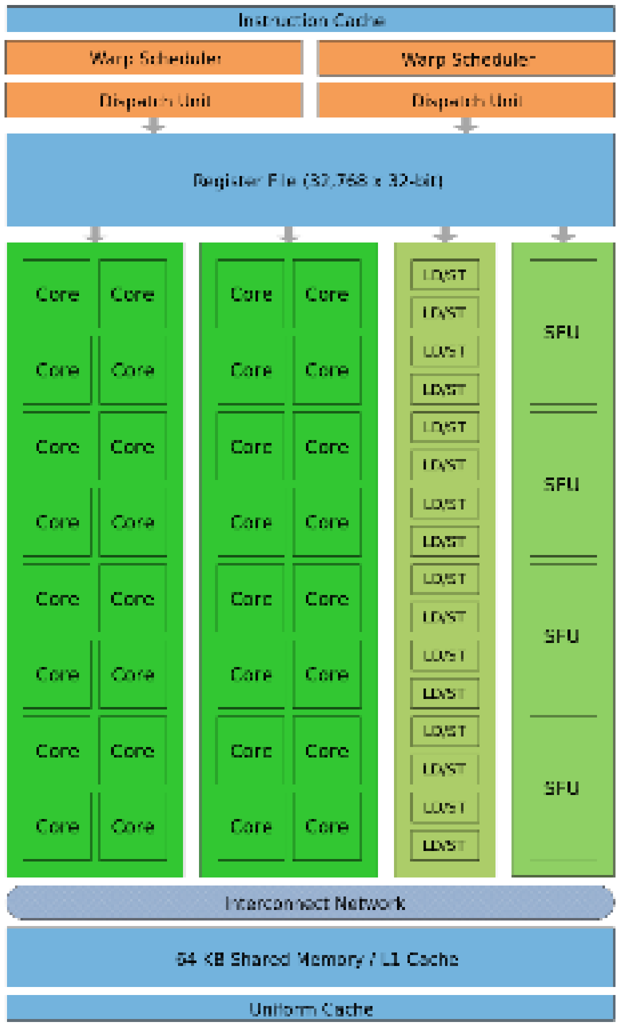
- When you allocate array on host - Can specifiy that it should be pinned - When you do any type of transfer - It can use DMA - Very fast - like malloc, dynamically allocates an array in RAM - Unlike malloc, the memory will be pinned - **Use codaFreeHost to deallocate when the memory isn't needed anymore**

Other CUDA memories

- So far we've only allocated in two ways
  - **Static**
  - **Dynamic**
- Both of these approaches allocate memory in the GPU's **global memory (RAM)** which is:
  - By far the **biggest** area of memory
  - Also the **slowest**
- There are other **smaller, faster** memories that we can use to dramatically increase performance
  - **shared memory**

Caching in GPUs

- The situation is a bit more complicated in GPYs due to how their cores are oganised
- Remember from last week:
  - A GPU is divided into a number of **streaming multiprocessors (SMs)**
  - Each SM has a number of parallel **CUDA cores**
- Picture shows one streaming multiprocessor (fermi architecture)



- Each SM has 32,768 registers available for the threads in the SM to use - It also has 64kb of shared memory / L1 cache that is shared between all cores in the SM - And a smaller "uniform cache" that is similarly fast

Uniform cache

- The uniform cache is an on-chip (part of each SM) cache that is designed for **broadcasting** data
- If multiple threads access the same address in the uniform cache at the same time
  - the data is sent to them all simultaneously

- Threads can't change the value of data in the uniform cache

Constanct memory

- should still be on the device
- but certify kernel code wont modify it
- This is to take advantage of the uniform cache we need to specify that a particular variable in global memory can't change
- use the `__constant__` modifier
- These variables will be stored in teh device's global memory, in a special area reserved for constant data
- When a thread accesses data from here, it will cached in the uniform cache and optimised for broadcast reads