Introduction

- Memory usage is abstracted away on a CPU. e.g. Main memory, L1 and L2 cache. Paging.
- Not so on the GPU. Different types of memory must be utilised and programmed for maximum performance.
- However, some memory abstraction is appearing on the new Fermi cards.
Memory hierarchy

Each thread can:
Read/write per-thread **registers**
Read/write per-thread **local memory**
Read/write per-block **shared memory**
Read/write per-grid **global memory**
Read/only per-grid **constant memory**

Declaring variables:

```c
__device__ __local__ // __device__ __shared__
__device__ __constant__ // __device__ <-- global
```
Some suggestions

Carefully divide data according to access patterns
R/Only: constant memory (very fast if in cache)
R/W shared in a block: shared memory (very fast)
R/W in each thread: registers (very fast)
  But just hope you they don’t spill (cf. the profiling/perf talks)
R/W inputs/results: global memory (very slow)

Put another way: Where do you declare a var?
Host accessible? Make it a global/constant and put it outside all the functions
No? Then it’s a register/shared/local var and belongs in a kernel

Note on pointer use

Pointers can only point to memory allocated or declared in global memory:

Allocated in the host and passed to the kernel:
  __global__ void cuda_kernel(float* p)

Obtained as the address of a global variable:
  float* p = &g_Var;
Global memory

- **Main** GDDR3/GDDR4/GDDR5 memory on the GPU.
- Not close to the GPU chip.
- **Slow** to access. 400 – 600 cycles.
- But there's a lot of it.
- Latency hiding by the thread scheduler can ameliorate access times.

Coalescing Global Memory Acesses

- CUDA devices with compute capability 1.0 and 1.1 can fetch data in a single 64-byte or 128-byte transaction.
- Access patterns where threads access a chunk of contiguous data are very efficient. The memory reads/writes are "coalesced" into a single access.
- Scattered access patterns require more memory transactions.
  
  ```c
  s[threadIdx.x] = a[threadIdx.x]; // coalesces
  s[threadIdx.x] = a[threadIdx.x*7]; // does not
  ```
- More on this in the performance session.
Quadro FX 5600

Specs

Memory
- 1.5 GB GDDR3
  - DDR => transfers 2 memory values per clock cycle
- 1600 MHz (2 * 800 MHz cf. DDR above)
- 384 bit bus width

Chip clock: 600 MHz

PCI-Express x16
- 16 lanes full duplex ~ 4GB/s total
- Proviso: Not sure if the card uses PCIe 1.0 or 2.0—the latter would mean 8GB/s total...

Quadro FX 5600

**Question 1:** How many “CUDA” cores are there?
128 = 16 SM * 8 SP  (since 8 SP per SM)

**Question 2:** What’s the GPU’s memory bandwidth?
1600 MHz * 384 bit / 8 = 76800 = 76.8 GB/s
These calculations have importance

Suppose you expect your kernel to perform 1 single-precision floating-point operation for every floating-point value read from global memory. What’s the peak performance you can achieve?

You can expect to process $768 / 4 = 19.2$ gigabyte single-precision data per second (19.2 GFLOPS)

For an 8800 GTX (~Quadro FX 5600)

- Theoretical peak performance is ~500 GFLOPS
- In practice, some sources suggest ~340 GFLOPS

Moral: Avoid global memory, calculate!
CHPC GPU: Quadro FX 5600

Note that the bandwidth between GPU memory and GPU chip is ~19x (or ~9x) greater than that between CPU and GPU memory offered through the PCI Express Bus.

So if your code depends on a processing massive amounts data, you may not be able to “feed” the GPU enough to take advantage of it’s compute power.

One may be able to ameliorate this using asynchronous transfers.

---

Texture Memory

- Buffers sections of global memory. Up to 8KB Texture cache is present in each SM. Read-only within an executing kernel.
- On a cache miss, data around the missed location is loaded into the cache.
- Good choice for spatially local accesses (1D/2D). Data that is almost coalesced.
- Same latency as DRAM, but special cases deliver data in < 100 cycles.
- However, don’t count blocks on the same multiprocessor using spatially similar accesses.
- Not a good idea to run one or two blocks on a SM when high texture dependency expected.
Texture Memory

texture<int, 1> texsrc;

__global__ the_kernel(int * array)
{
    // access texture in some pattern
    array[threadIdx.x] = tex1Dfetch(texsrc, index) +
    tex1Dfetch(texsrc, index+3);
}

int main()
{
    int * d_a;
    cudaMalloc((void**)&d_a, sizeof(int)*N);
    cudaMemcpy(d_a, &h_a, sizeof(int)*N);
    the_kernel<<<1,N>>>(d_a);
    cudaUnbindTexture(texsrc);
}

One very important use is for hardware-based interpolation

For example, if you have a piecewise-linear 2D function you wish to have values for over some domain

- Then first load a set of sample points (say a uniform sampling over a grid) from that function into texture memory
- Setup linear interpolation for that texture

Result: Texture fetches now interpolate results between samples

- tex2d(texsrc, 1.5, 2.6);
- Estimates the value at coordinate (1.5, 2.6) from the values around it.
Shared memory

- Really fast. 1 cycle. **Shared** amongst all threads in a block. 16KB per SM, 48KB on new cards.
- Use as much as possible. Split problem into chunks of shared memory and read that chunk in from global memory.
- Watch out for **bank conflicts**!
- Occur when different threads access the same shared memory address. Scheduler has to serialise the access.
- Rule of thumb to avoid these:
  - either all threads **access same memory address** or
  - all threads **access unique memory addresses**.

__global__ void the_kernel(int * a) {
  __shared__ int s[N];
  // Read from slow global memory into
  // fast shared memory
  s[threadIdx.x] = a[threadIdx.x];
  __syncthreads();
  // Do lots of processing with s[threadIdx.x]
  ... 
  // Write back to global memory
  a[threadIdx.x] = s[threadIdx.x];
  syncthreads();
}

---

*CHPC*

---

*CHPC*
Registers

- Really fast. 1 cycle to access.
- They are split among available threads. e.g. 16384 registers per SM / 256 threads = 64 registers per thread
- Compiler will “spill” registers into local memory if it doesn’t have enough.
- Recall local memory is a thread’s unique and private chunk of global memory, so access to this is slow!
- Inspect the PTX assembly code to know for sure.
- Large structs get split across multiple registers.

Constant Memory

- 64KB of constant memory, up to 8KB of constant cache per SM.
- If data is cached, access can be as low as 1 cycle, can be hundreds of cycles if not.
- No penalty for first access due to precaching.
- Great for lookup tables. e.g. precalculated trig values.
**Constant Memory**

```c
__device__ __constant__ ParamStructType params;
__global__ the_kernel(int * a, int * b)
{
    a[threadIdx.x] = params.snippet * b[threadIdx.x];
}
int main(void)
{
    ParamStructType host_params;
    cudaMemcpyToSymbol("params", hostParams, sizeof(ParamStructType));
}
```

**CUDA Streams**

- So far, the paradigm for calling GPU code has been **synchronous**.
- We do a memcpy or call a kernel and wait for the operation to finish, but the CPU or GPU can be idle during these operations.
- **CUDA Streams** allow concurrent execution of host and device code.
- Requires the programmer to understand asynchronous programming.
- Non-blocking Network I/O is the most obvious example.
CUDA Streams

Overlap computation as much as possible to maximise computational resource usage.

Sequential CPU/GPU Computation

Asynchronous CPU/GPU Computation Using 3 Streams

CUDA Streams

Modifications to existing code

- Kernel Execution
  - CudaStream passed as kernel parameter.
- Memory Allocation
  - CudaMallocHost (Pinned Memory)
- Memory Copy
  - CudaMemcpyAsync
CUDA Streams

cudaStream_t stream[2];
for(int i=0; i<2; ++i)
    cudaStreamCreate(&stream[i]);
float * a;
cudaMallocHost((void**)&a, 2*size);
for(int i=0; i<2; ++i)
    cudaMemcpyAsync(d_a + i*size, a + i*size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for(int i=0; i<2; ++i)
    kernel<<<100, 512, 0, stream[i]>>>(d_a + i*size, size);
for(int i=0;i<2;++i)
    cudaMemcpyAsync(a + i*size, d_a + i*size,
                    size, cudaMemcpyDeviceToHost, stream[i]);

CUDA Streams

- After the asynchronous calls, the CPU can do other work, or invoke other asynchronous calls.
- Once CPU tasks are completed, the CPU can invoke calls to see if the asynchronous streams have finished processing.
  - Wait for one stream
    - cudaStreamSynchronize()
  - Wait for all streams
    - cudaThreadSynchronize()