Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

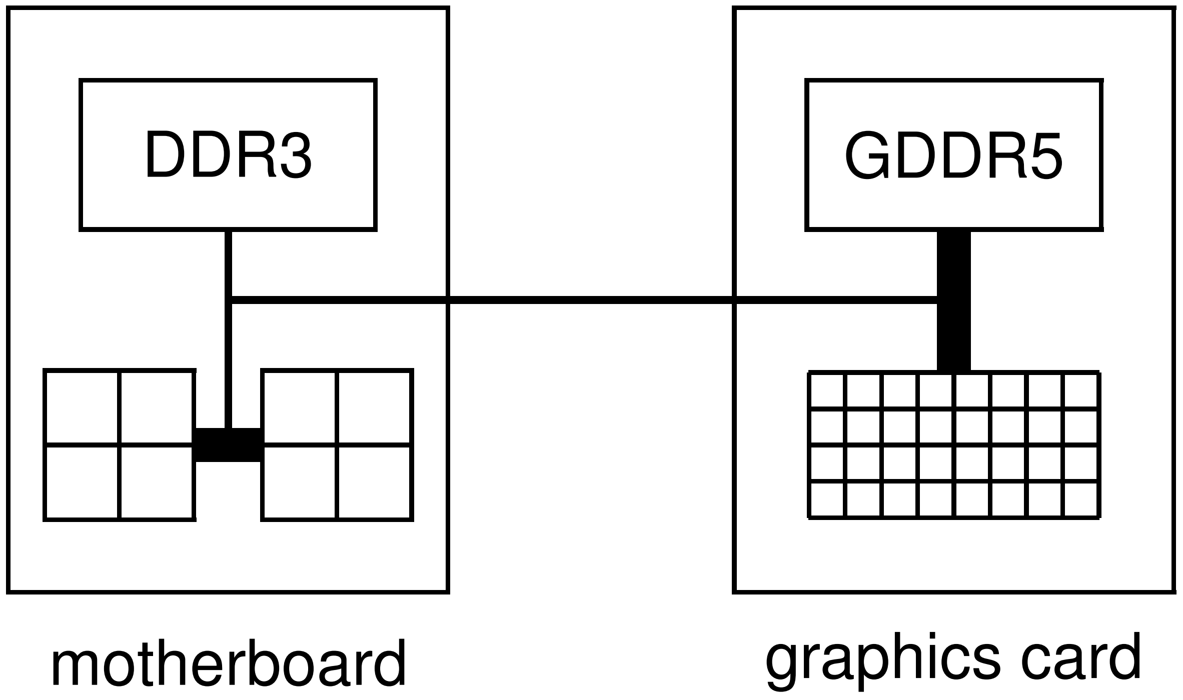
Introduction to CUDA

Lecture 7

István Reguly
reguly.istvan@itk.ppke.hu

CPU-GPU system

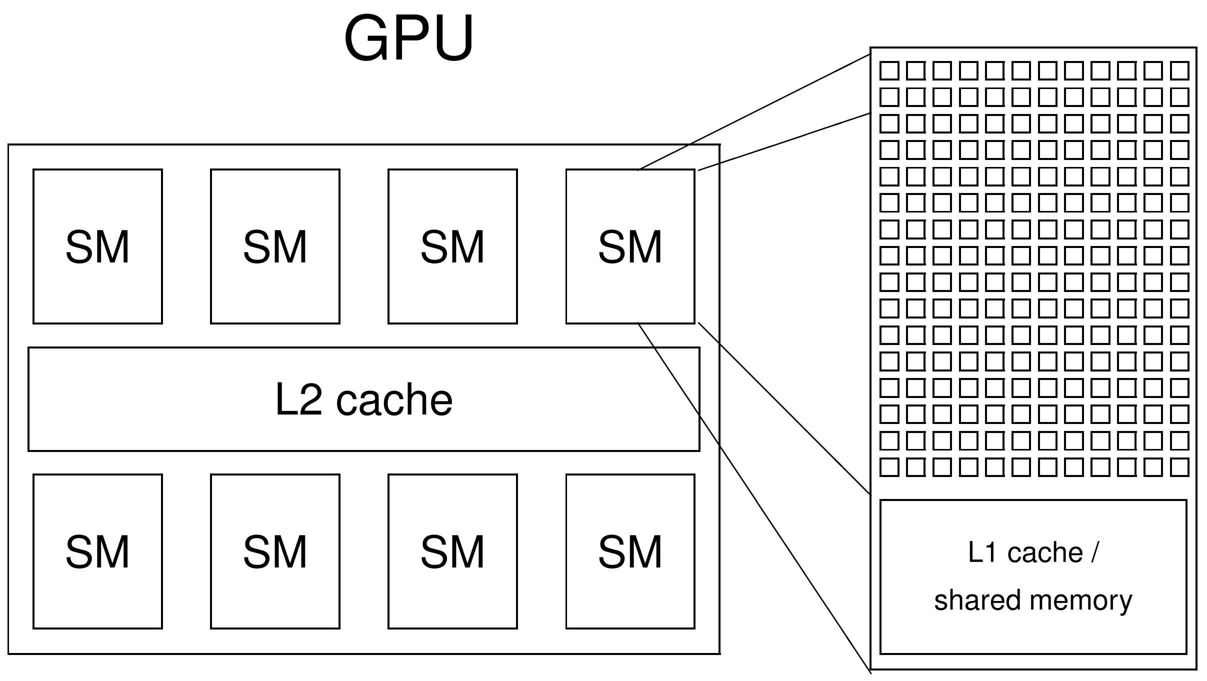
Different memory spaces!



Hardware view

- Building block is a “streaming multiprocessor” (SM):
  - 64 cores and 256KB registers
  - 192KB of shared memory + L1 cache
  - Up to 2048 threads per SM
- P100 card
  - 108 SMs -> 6912 cores
  - 40 MB shared L2 cache
  - 40 GB HBM memory, 1555 GB/s bandwidth

Hardware view

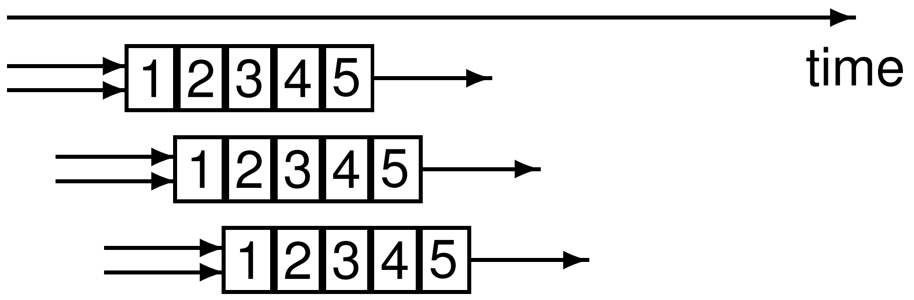


Multithreading

- Hardware & CUDA programming from a Single Instruction Multiple Threads view:
  - Groups (of 32 called a **warp**) of threads execute the same instructions, but with different data
  - Natural choice for graphics applications
  - Key to performance: no “context switching” unlike on CPUs: an SM can execute any warp that is ready

Multithreading

- For each thread, one operation completes before the next starts, but we have lots of threads and many processing cores:



- Memory access from device memory has a delay of 400-600 cycles, which is how much a thread will have to wait, so we need a lot of threads that can do other things in the meanwhile

Software view

At the top level what we usually do is:

1. Initialise card
2. Allocate memory on host and device
3. Copy data from host to device
4. Launch multiple instances of a “computational kernel”
5. Copy data back to the host
6. (repeat)
7. De-allocate memory and terminate

CUDA programming

- CUDA is an extension of C/C++ that allows you to implement SIMT code that will run on the GPU and launch it
  - SIMT abstraction:
    - Write code from the perspective of a single thread
    - Threads are organized into groups called blocks – threads in the same block can talk to each other
    - Many blocks may be launched, which cannot interact directly
- Also has API calls for managing the GPU and to handle memory transfers

CUDA programming

- Launching a GPU operation in its simplest form:

```
kernel_routine<<<gridDim, blockDim>>>(args);
```

- `gridDim` is the number of instances of the kernel: the number of groups of threads
- `blockDim` is the number of threads in each instance: the size of the groups
- `args` is a number of arguments to be passed to the GPU kernel
- More general form allows you to launch 2D and 3D kernels

CUDA programming

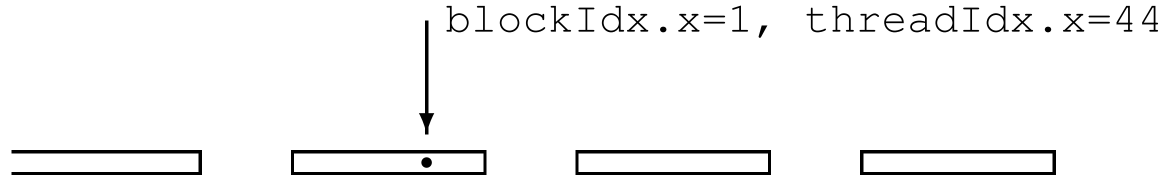
At the lower level, one instance (group) of the kernel is assigned to an SM, and is executed by a number of threads, each of which knows about:

- Some variables passed as arguments
- Pointers to arrays in device memory
- Global constants
- Shared memory and private memory (registers)
- Some special vairables:
    - `gridDim(.x/.y/.z)` size of the grid of blocks
    - `blockIdx (.x/.y/.z)` index of the current block in the grid
    - `blockDim(.x/.y/.z)` size of each block
    - `threadIdx (.x/.y/.z)` index of the current thread in the block

CUDA programming

```
kernel_routine<<<4,64>>>(args)
```

- 1D grid with 4 blocks, each with 64 threads
- gridDim=4
- blockDim=64
- blockIdx.x goes from 0 to 3, blockDim.x is 64
- threadIdx.x goes from 0 to 63



```
int tid = blockIdx.x*blockDim.x + threadIdx.x;
```

CUDA programming

- Kernel code looks fairly normal once you get used to two things:
  - Code is written from the point of view of a single thread
    - Not like OpenMP, but similar to MPI, where you have a ”rank” to identify you
    - All local variables are private to that thread, everything else is shared
  - Need to think about where each variable lives
    - Register, shared memory, or global memory
- Some operations (kernel launch, functions with *Async) are asynchronous
  - Initiates an operation on the GPU, but does not wait for it to complete
  - If you need the results, CPU needs to “synchronise” with the GPU

Host code

```
int main(int argc, char **argv) {
 float *h_x, *d_x;         //  h=host, d=device
 int   nblocks=2, nthreads=8, nsize=2*8;

 h_x = (float *)malloc(nsize*sizeof(float));
 cudaMalloc((void **)&d_x,nsize*sizeof(float));

 my_first_kernel<<<nblocks,nthreads>>>(d_x);

 cudaMemcpy(h_x,d_x,nsize*sizeof(float),
                  cudaMemcpyDeviceToHost);

 for (int n=0; n<nsize; n++)
   printf(" n,  x  =  %d  %f \n",n,h_x[n]);

 cudaFree(d_x); free(h_x);
}
```

Key API calls

- `cudaMalloc(void** ptr, size_t size)`
- `cudaMemcpy(void * to, void *from, size_t size, direction)`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
- `cudaFree(void *ptr)`
- `cudaDeviceSynchronize()`
  - Kernel launches are asynchronous
- All of these return success/failure
  - May be delayed, due to async kernel launches!

Kernel code

```
__global__ void my_first_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;

 x[tid] = (float) threadIdx.x;
}
```

- `__global__` identifier says it’s a kernel function
- Each thread sets one element of x array
- Within each block of threads threadIdx.x ranges from 0 to blockDim.x-1, need to add bklockDim.x*blockIdx.x to get globally unique ID

Let’s checkout our first.cu code

- Compile with nvcc instead of gcc/cc
    - File extension “.cu”
    - No extra arguments needed
    - Should add -arch=sm_80

source /project/p_covidpre/source_libs

- nvcc first.cu -o first -arch=sm_80

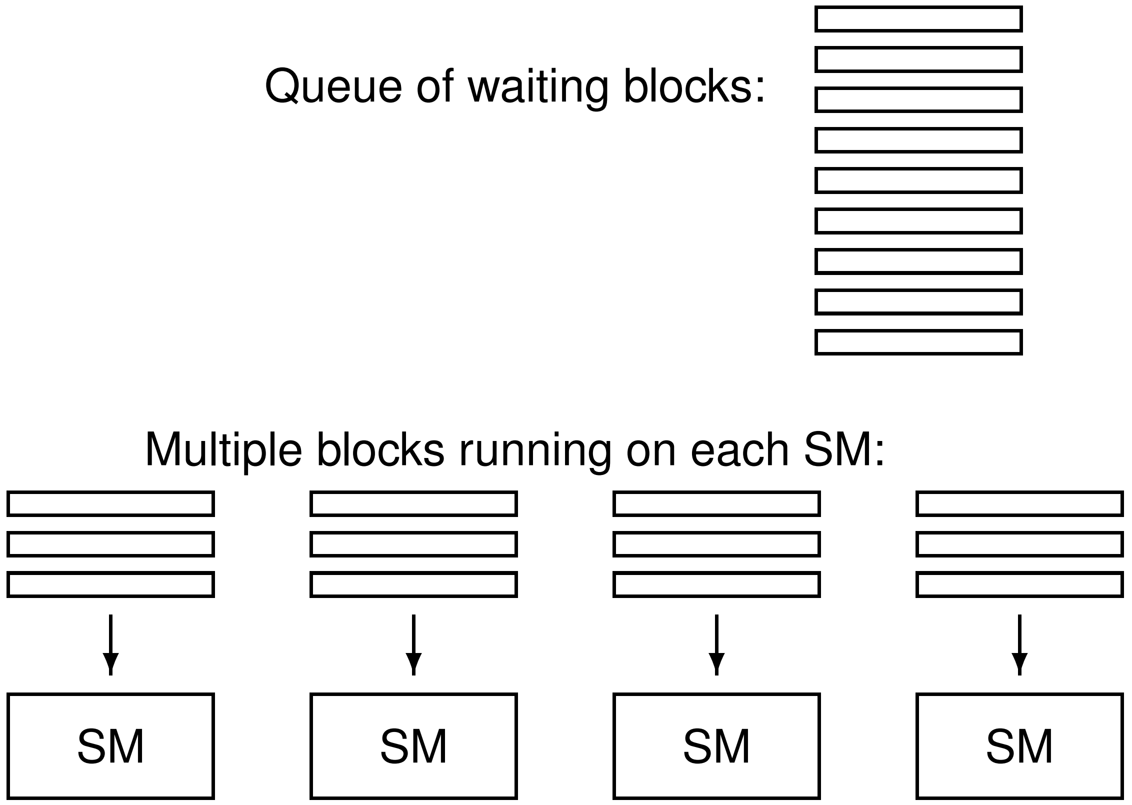
Exercise: let’s implement saxpy

- Implement the ”saxpy” kernel
  - y vector = a * x vector + y vector

```
void saxpy(int n, float alpha, float * X, float * Y)
{
  for (int i=0; i<n; i++)
    Y[i] = alpha*X[i] + Y[i];
}
```

- n=2000 … 200000000
- Modify first.cu – 3 arrays needed
- Populate X & Y on the CPU, copy up to the GPU
- Call saxpy
- Copy Y to CPU
- Print first 20 values
- **Add timing to each step (copy, execute, copy)**

Scheduling


Queue of waiting blocks:
Multiple blocks running on each SM:
SM
SM
SM
SM

CUDA programming

- At most 1024 threads/block
- Suppose we have 1000 blocks, each with 128 threads – how does it get executed?
- On Pascal hardware, you would probably have 8-16 blocks running at the same time on each SM, and each block has 4 warps: 32-64 warps
- At each clock cycle, SM warps scheduler picks a warp that is ready to execute
  - Not waiting for memory
  - Not waiting for previous instructions to execute
- What programmer needs to make sure of, is that there are plenty of warps to choose from

Memory – importance of locality

- 9.7 Tflops/s GPU
- 1555 GB/s memory bandwidth
  - 32 byte cache lines (4 doubles)
  - 48 G lines/s -> 192G double/s
- Each computation requires 2 inputs, one output
  - If we load a cache line for each, the most compute we get is 16 Gflops!
  - If we use the full cache line, we get 64 Gflops!
  - We need to re-use data we transferred to get up to 9.7 teraflops!

Memory access patterns

- Whenever one accesses memory, spatial and temporal locality are important for performance
  - Spatial: if we access x[i], we’ll access x[i+1] too
  - Temporal: if we access x[i], we’ll likely access it again
  - Caches exist to speed up these accesses – one memory access moves a whole cache line
- There is very little cache on the GPU
  - If every thread accesses different data, there isn’t enough room to exploit temporal locality
  - **Spatial locality on a GPU is key**: but by hoping that if given thread accesses x[i] it will access x[i+1] too, but by hoping if thread p accesses x[p], then thread p+1 accesses x[p+1]
    - This is due to threads p and p+1 always executing the same instruction (such as a memory access) but with a different index

A “good” kernel

```
__global__ void my_first_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;

 x[tid] = threadIdx.x;
}
```

- 32 threads in a warp will access neighbouring elements of array x
- Covers a full cache line
    - If the first element is aligned
- Gives us perfect spatial locality

A bad kernel

```
__global__ void bad_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;

 x[1000*tid] = threadIdx.x;
}
```

- Different threads in the warp access data on different cache lines – a strided access pattern
- Has to move an entire cache line for each thread
  - Slow!
  - Don’t have that much cache!

Importance of access patterns

- Take a look at exercise 2
  - Understand the differences between version 1 and version 2
    - Comment in/out in 3 places in the kernel
    - Don’t mind about the CPU stuff, just what is in the kernel
  - Run both
  - Try and understand why one is faster than the other

```
source /project/p_covidpre/source_libs
```

```
nvcc -O3 prac2.cu -o prac -arch=sm_80 –lcurand
```

Global arrays

- So far we have been looking at global/device arrays
- Held in large device memory
- Allocated through host code
- Pointers held in host code and passed into the kernels
- Continue to exist across different kernels, until host frees them
- Since blocks execute in an arbitrary order, if one thread in a block modifies an array element, no other thread in a different block should access that same element

Global variables

- Variables can be declared in global scope as well, just like ordinary C global variables

```
__device__ int reduction_lock=0;

__global__ void kernel_1(...) {
   ...
}

__global__ void kernel_2(...) {
   ...
}
```

Global variables

- The `__device__` prefix tells nvcc that this is a global variable on the GPU not the CPU (not directly accessible on the CPU!)
- Can be read and modified by any kernel
- Can also declare arrays of a fixed size
- Can read/write from CPU using special routines: `cudaMemcpyToSymbol`

Constant variables

- Very similar to global variables, except they cannot be modified by the kernels
- `__constant__`
- Initialised/modified the same way as globals
- Goes through a special constant cache
  - Ideal for broadcast: all threads read the same constant
  - `a[tid] = alpha*b[tid]`
- Use for variables only set once and used throughout

Registers

- Everything declared in local scope (within the `__global__` function) are mapped to registers

```
__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

- 65k 32-bit registers per SM
- Up to 255 registers per thread
- Up to 2048 threads per SM
- If you want to use 255 registers, you can only have 256 threads per SM
- If you want to use 2048 threads, then you can only use 32 registers per thread

Registers

- What happens if your application needs more registers?
- They get allocated to the global memory by the compiler
    - They get cached in L1
- Really expensive!
- Compile with –Xptxas=-v to get a report of how many registers you are using

2D blocks

- In this simple case we had a 1D grid of blocks, and a 1D set of threads within each block
- If we want to use a 2D set of threads, then we use `blockDim.x` and `blockDim.y`, as well as `threadIdx.x` and `threadIdx.y`, etc.
- To launch a 2D grid of blocks:

```
dim3 threads(16,4)
dim3 blocks(8,22)
my_new_kernel<<<blocks,threads>>>(d_x)
```

- Threads are assigned to warps in a row-contiguous way

2D kernels

```
__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] );
  }
}
```

Texture cache

- Texture memory originally intended for graphics
- Good for read-only data, especially for non-uniform accesses
- 24 kB on Pascal
- Uses 32 byte cache lines
  - More cache lines per thread!
- Need to declare pointers with
  `const type * __restrict__ var`

Laplace2D in CUDA

- Now we know almost everything to parallelise our laplace 2D simulation from last week
- Reduction: write into a separate array, then call

```
//To the top
#include <thrust/reduce.h>
#include <thrust/device_ptr.h>
//Allocate and pass "array" to kernel
...
//Perform a max reduction of all values in "array"
double result = thrust::reduce(
     thrust::device_ptr<double>(array),
     thrust::device_ptr<double>(array+size),
     0.0,thrust::maximum<double>());
```