Australian National University

Launching Kernels

Dr Eric McCreath

Research School of Computer Science

The Australian National University

Launching Kernels

To maintain high occupancy you need to launch kernels that have many blocks (or work groups) with each block having many threads (work items) in them.

Blocks are arranged in grids of 1D, 2D, or 3D blocks. The threads within the block can also be arranged in 1D, 2D, or 3D.

These blocks will run on the stream multiprocessors, you can quickly synchronize and share data (via shared memory) between threads within a block.

On Nvidia GPUs threads are executed SIMT in groups 32 called **warps** . So generally one should make the number of threads in a block a multiple of 32 (AMD GPUs have similar groups although there are 64 rather than 32, and is called a "wave").

Launching Kernels

So in CUDA the syntax for launching a kernel is:

```
kernelFuntionName<<<blocks,threadsPerBlock,shareMemorySize, stream>>>(parameters);
```

Optional

Number of parameters are fixed.

For 1D grids/blocks you can just use integer values for number of block and the threadsPerBlock. However, if you are using 2D or 3D then the "dim3" syntax can be used. e.g.

```
dim3 dimBlock(32,64);
dim3 dimGrid;
dimGrid.x = (numx + dimBlock.x - 1) / dimBlock.x;
dimGrid.y = (numy + dimBlock.y - 1) / dimBlock.y;

myKernel<<<dimGrid,dimBlock>>>(data, numx, numy);
```

Launching Kernels

Starting a kernel in OpenCL looks a little more involved however it is really much the same as CUDA. The command you use is:

```
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                               cl_kernel kernel,
                               cl_uint work_dim,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event);
```

One difference to note is the "global_work_size" is the total number of threads (or work items) in each dimension.

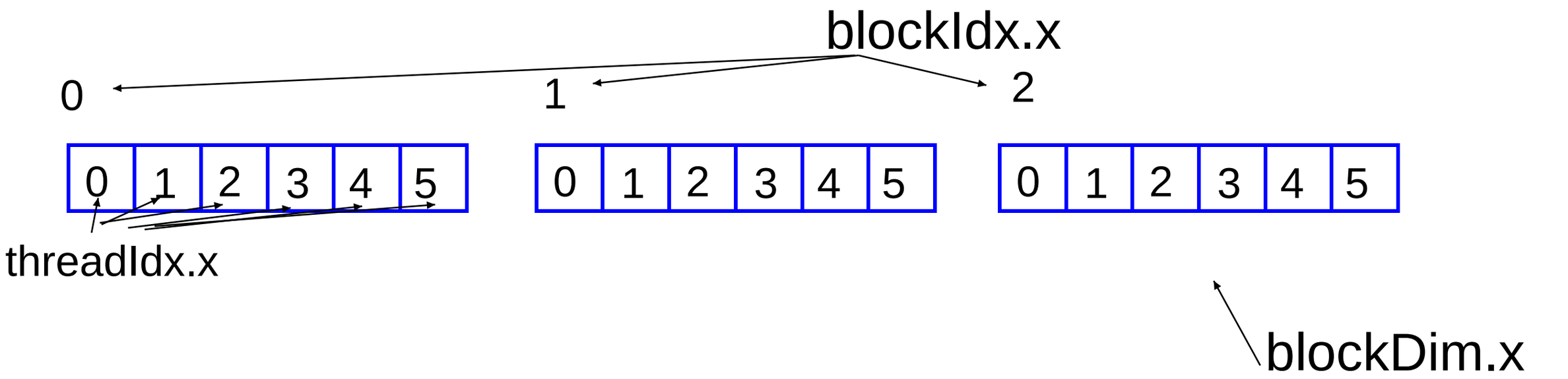
Item to work on

Once your kernel is launched you need to work out which data item (or part of your problem space) for that particular thread to work on. Within Cuda you use the following variables:

```
blockIdx.x - the first dimension's block index
blockDim.x - the number of threads across the first dimension in the block
threadIdx.x - the index of the first dimension
```

If we are dealing with just 1 dimensional grids and block then to obtain a unique index one can:

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



Item to work on

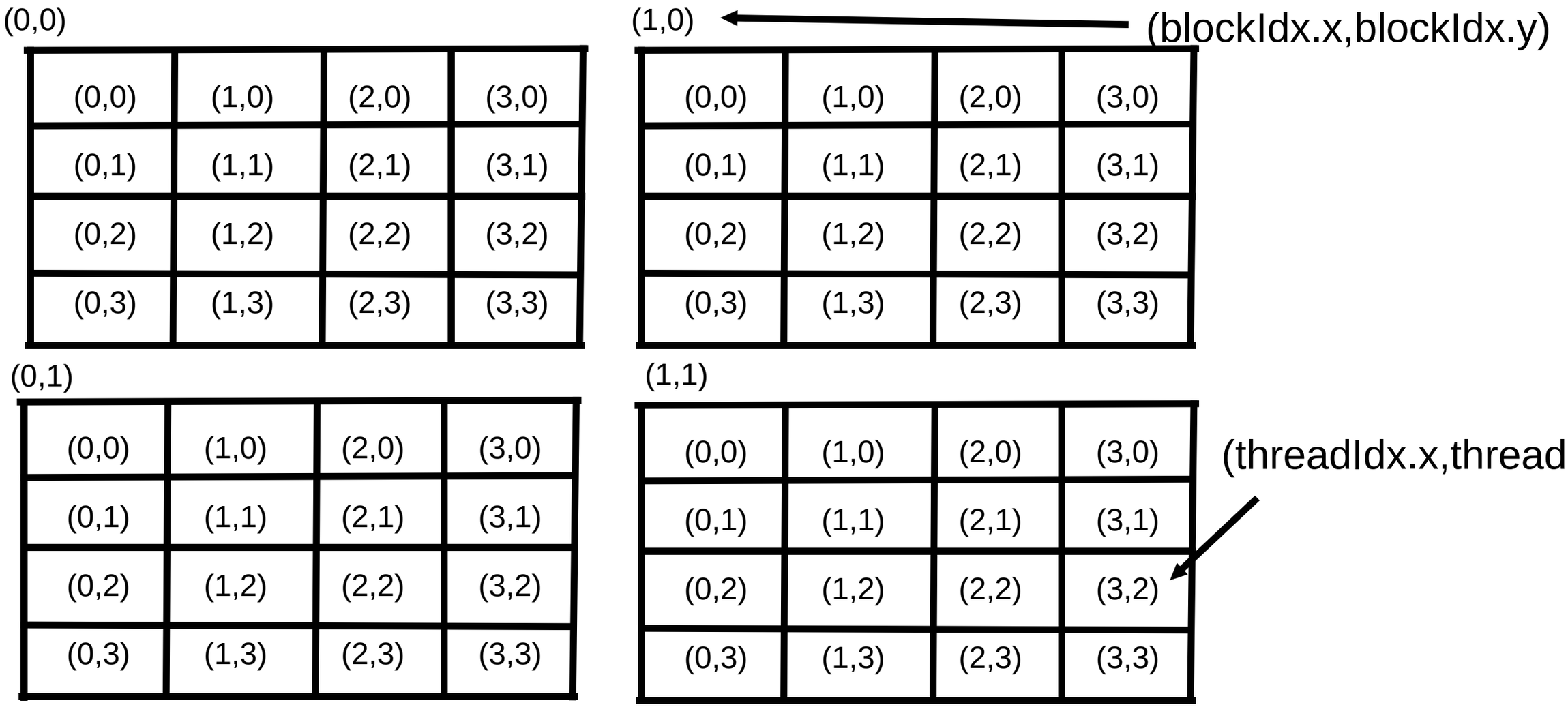
For 2D and 3D grids and blocks you use:

```
blockIdx.y, blockDim.y, threadIdx.y - for the second dimension
blockIdx.z, blockDim.z, threadIdx.z - for the third dimension
```

This is a variety of limitations on the size of these dimensions, the total number of blocks and maximum number of threads per block.

Item to work on

A 2x2 grid each having say 4x4 threads:



So a global coordinate can be calculated within each thread by:

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

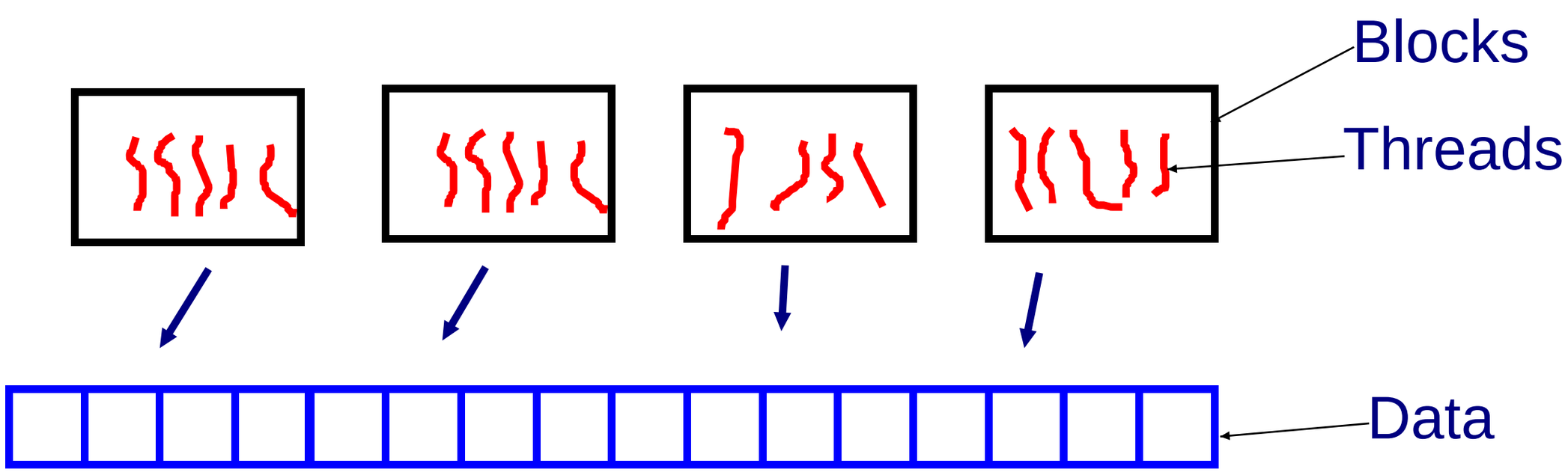
Item to work on

In OpenCL you use:

```
get_global_id(d) - to obtain the index of dimension "d"
get_local_id(d) - obtains the index of dimension "d" within the work group
get_local_size(d) - gets the size of to work group in dimension "d"
get_work_dim() - gives the number of dimensions
get_global_size(d) - the total number of work items in dimension "d"
```

Grid-Block Striding

- So when a kernel is started generally a large number of threads (or work items) are started.
- These are grouped into "blocks" (or work groups).
- The question is how do we map threads to the data we wish then to work on.
- There are lots of options!!



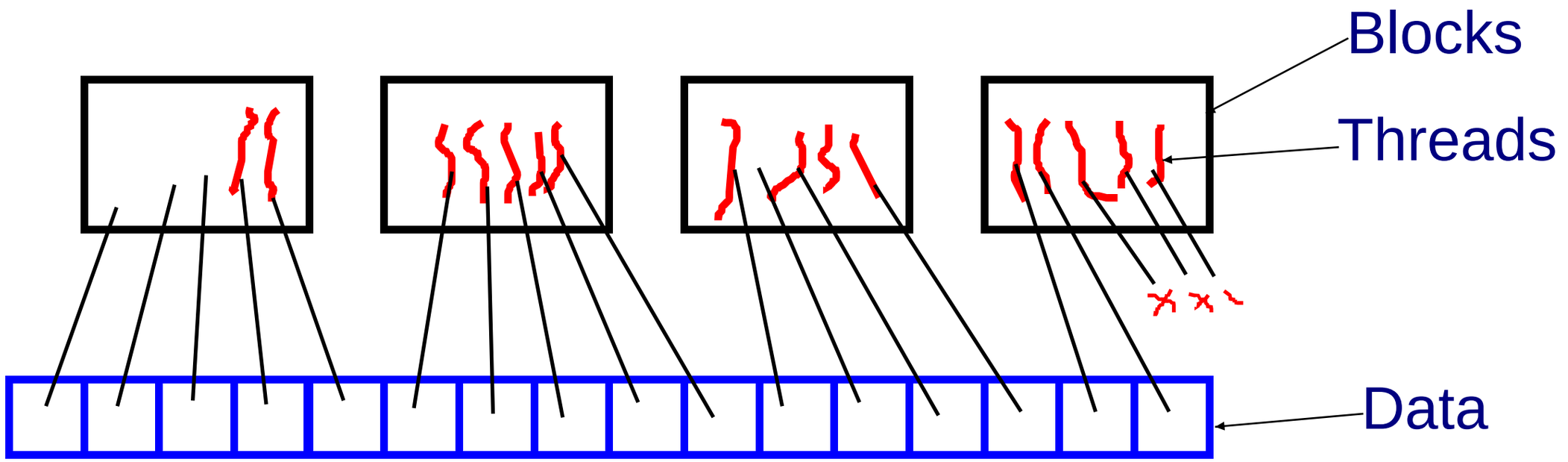
Grid-Block Striding

- One approach is to do a one-to-one mapping. So one thread does one item of work (let n be the number of data items and t be the number of threads per block). So to launch the kernel you could:

```
kernel<<< (n-1)/t + 1)  , t>>>(n, data);
```

- Then the kernel:

```
__global__ void kernel(int n, void *data) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < n) workOnItem(idx, data);
}
```



Grid-Block Striding

The one-to-one mapping approach is an okay approach. It is simple and memory access patterns are good (assuming the data has a linearly layout). However there are 2 down sides:

- There is some overhead (although not much) in launching a thread, so if a thread is not doing much work then this overhead becomes significant.
- There is a limit in the number of threads and blocks you can launch, so if the data is too big you may not be able to launch enough threads. Currently in CUDA the limit is:

$$(2^{31} - 1) * 1024$$

You could launch the kernel multiple times within a loop, although this would only help address the second of the above issues.

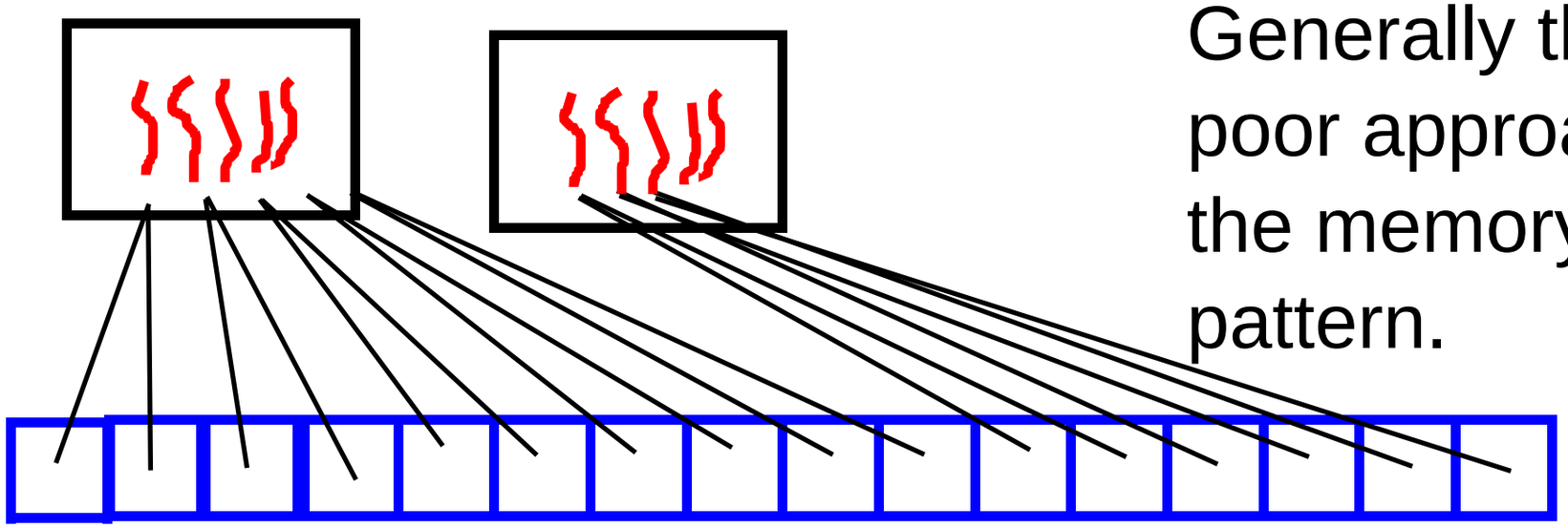
Grid-Block Striding

- You could get threads to work on consecutive data items. So to launch the kernel you could:

```
kernel<<<(n-1)/(m*t) + 1), t>>>(n, data); // m is the number of items per thread
```

- Then the kernel:

```
__global__ void kernel(int n, int m, void *data) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i=0;i<m && i+m*idx < n;i++) workOnItem(i+m*idx, data);
}
```



Generally this is a poor approach due to the memory access pattern.

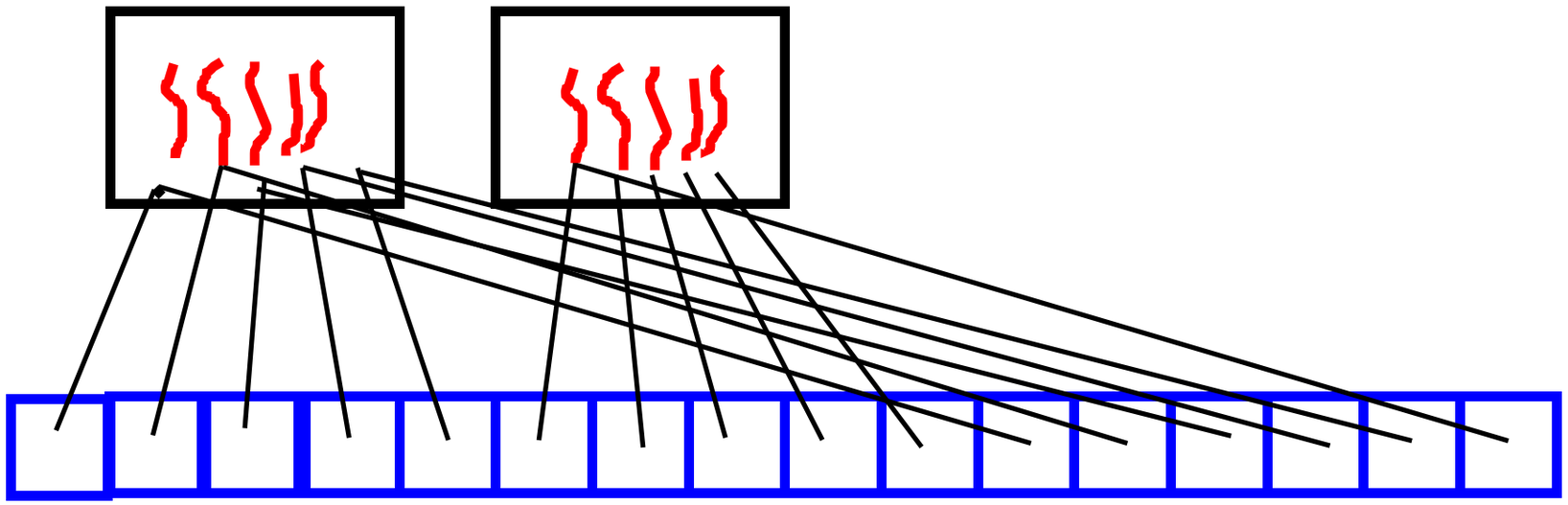
Grid-Block Striding

- A good approach is grid-block striding. So to launch the kernel you could:

```
kernel<<<1000, 256>>>(n, data); // note the fixed number of blocks/threads
```

- Then the kernel:

```
__global__ void kernel(int n, void *data) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;
  for (int i=idx; i < n;i += stride) workOnItem(i, data);
}
```



Memory Allocation

To allocate memory on the device you simply can use:

```
char *data_d;
cudaMalloc(&data_d, size);
```

This works just like malloc and the memory persists until it is freed with:

```
cudaFree(data_d);
```

Similarly you can also use the "cudaMallocHost" command to allocate memory on the host:

```
char *data_h;
cudaMallocHost(&data_h, size);
```

The advantage of this over just using "malloc", which also works, is that memory is pinned and transfers to the device can use DMA rather than the CPU. The disadvantage is that the memory is pinned so it takes and needs to fit in your physical memory.

Transfering Memory

To transfer data to the GPU you can use:

```
cudaMemcpy(data_d, data_h, size, cudaMemcpyHostToDevice); // size - in bytes
```

To transfer data back from the GPU to the host memory you:

```
cudaMemcpy(data_h, data_d, size, cudaMemcpyDeviceToHost);
```

The above commands are queued on the default stream and they block until complete. Now sometimes you may wish to do asynchronous transfers:

```
cudaMemcpyAsync(data_d, data_h, size,  cudaMemcpyHostToDevice, stream);
```

This command returns immediately and is queued to a particular "stream".

Checking Errors - Very Important!!!

It is always good to check that a CUDA commands worked correctly. Often people use a macro such as:

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

// this macro checks for errors in cuda calls
#define Err(ans) { gpucheck((ans), __FILE__, __LINE__); }
inline void gpucheck(cudaError_t code, const char *file, int line)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPU Err: %s %s %d\n", cudaGetErrorString(code), file, line);
      exit(code);
   }
}
```

Then wrap your CUDA commands with it. e.g. :

```
Err(cudaMalloc(&str_d, size));
```

References

- Using CUDA Warp-Level Primitives, Lin and Grover,
https://devblogs.nvidia.com/using-cuda-warp-level-primitives/
- Cuda C Programming Guide,
https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
- OpenCL Quick Reference Card
https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf
- Cuda Quick Reference Card
http://www.info.univ-angers.fr/pub/richer/cuda/CUDA_C_QuickRef.pdf