Australian National University

GPU Memory

Dr Eric McCreath

Research School of Computer Science

The Australian National University

Memory

So on the GP104 we have:

- L1 - 24KiB per SM, line 32B, hit latency 84 cycles, LRU
- L2 data - 2MiB, line 32B, hit latency ~216
- L1 constant 2KiB per SM, line size 64B, **broadcast latency ~25**
- Shared memory 64KiB per SM, non-conflict latency 23, bandwidth **3,919 GiB/s** in theory
- Global memory 8GiB, latency from ~1029 to ~400 down to 84, bandwidth **192 GiB/s** in theory

The basic message is if your kernel reuses data get it into registers or shared memory and use it from there. If all the threads are reading constant data get it into constant memory.

Noting Shared memory = Local Memory in the OpenCL.

Shared Memory

To declare some data as shared memory within a kernel you use the __shared__ keyword. So the below will declare an array of 256 integers. This data can be accessed by all the threads within the block.

```
__shared__ int data[256];
```

Although this can not be dynamically allocated it can be "dynamically" specified when the kernel is launched. To do this you leave the number of elements unspecified in the kernel and then giving the number in when launched. e.g.

```
__global__ void kernel() {
   extern __shared__ int data[];
   // ..
}

// in the host code
   int sharedSize = 256*sizeof(int);
   kernel<<<blocks,threads,sharedSize>>>();
```

Shared Memory

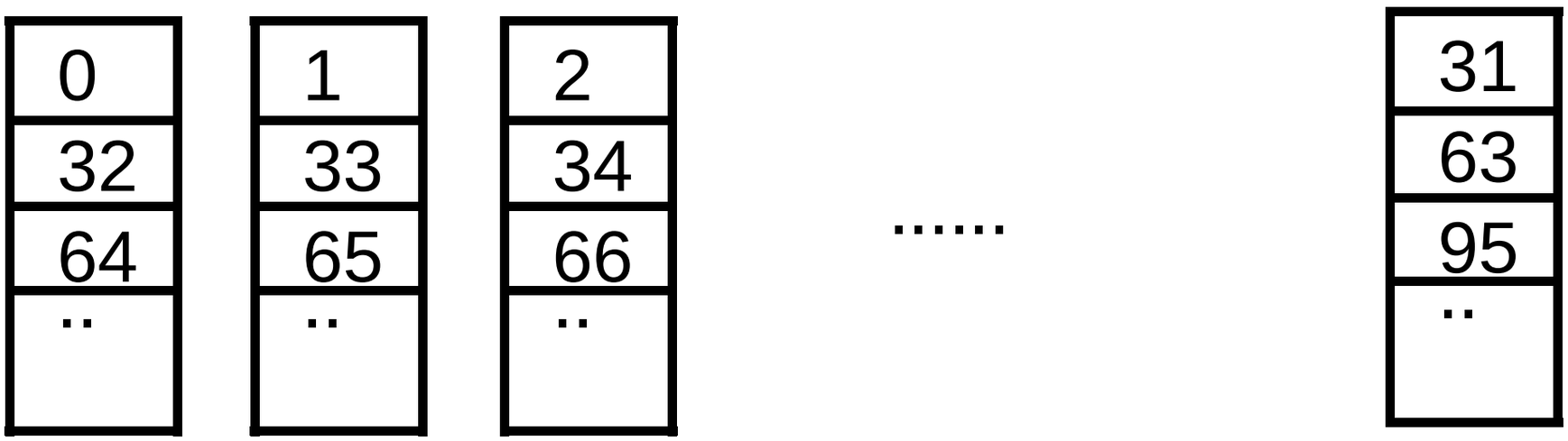
The shared memory is accessed by different threads in the block. The threads within the block will be at different stages in their execution, so if a thread wishes to read data from a shared location that another thread has written to you need to make certain the write is complete before you do the read. This is done with a "__syncthreads();". If you are using shared memory for storing data that is used repeatedly by different threads a typical pattern would be:

```
__global__ kernel(int *data, ....) {
   __shared__ int sharedData[];

   sharedData[threadIdx.x] = data[threadIdx.x];
   __syncthreads();
   // compute result using "sharedData"
}

// when you launch the kernel
kernel<<<blocks,size,size>>>(data, ...);
```

Shared Memory - bank conflicts

As shared memory is high bandwidth memory within the SMs. It is physically organized into 32 banks. Each bank has a width of 4 Bytes, so the addresses wrap around after 128 Bytes. Say we have an integer (which is 4 bytes) stored at address 0 it would go in the first bank, the next integer would be in the second bank, the 32 integer would wrap around and be stored in the first bank.

An integer array with {0,1,2,3, ...} in it would be stored:



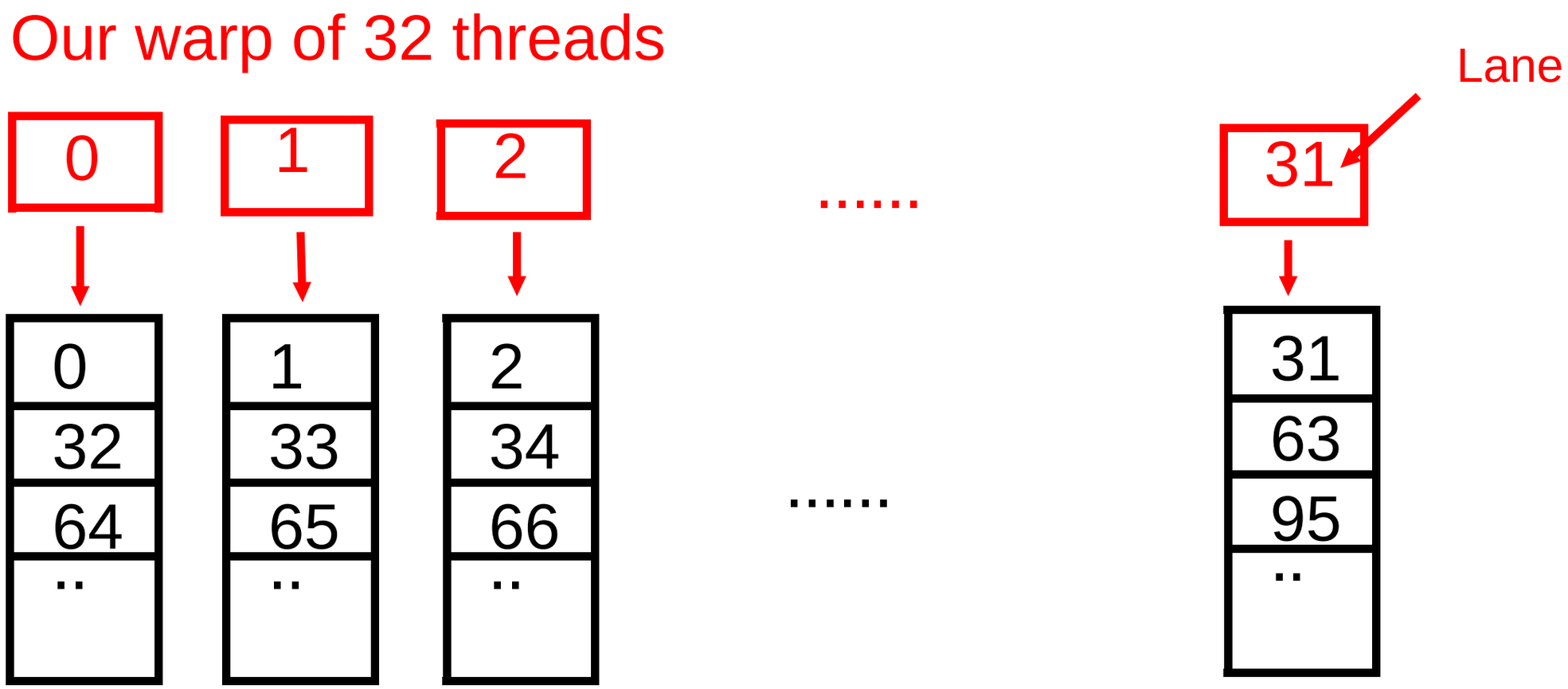
32 banks in shared memory

Shared Memory - bank conflicts

Lets say our shared memory integer array is called "data" then if we had code in our kernel that was:

```
res[idx] = data[idx] * 5;
```

Then our access pattern would be great, so no bank conflicts:

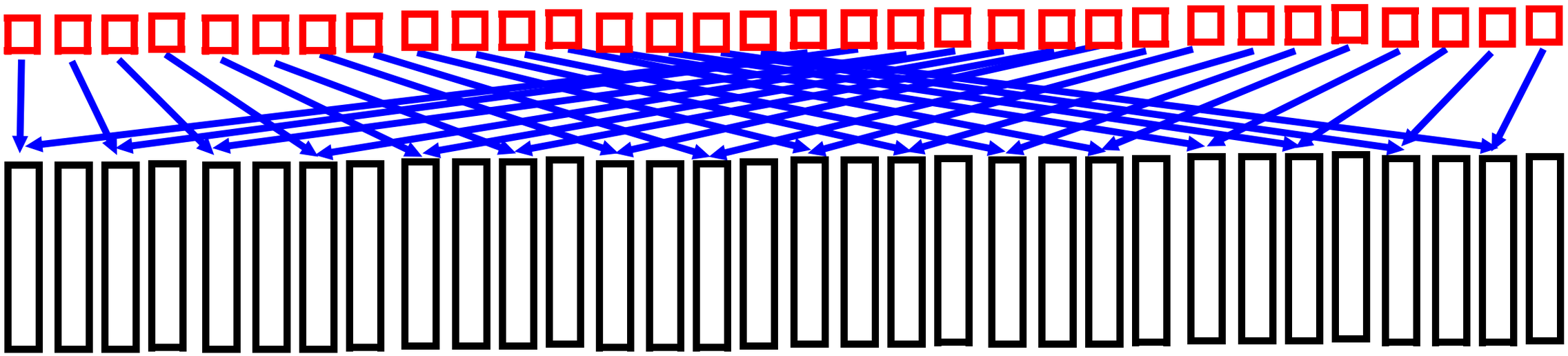


Shared Memory - bank conflicts

Now if we needed to access every second element in "data" such as:

```
res[idx] = data[idx*2] * 5;
```

We end up have every 2 threads in the warp wanting to access the same bank in the shared memory. This will still work fine, however it will be slower as the reads are serialized.

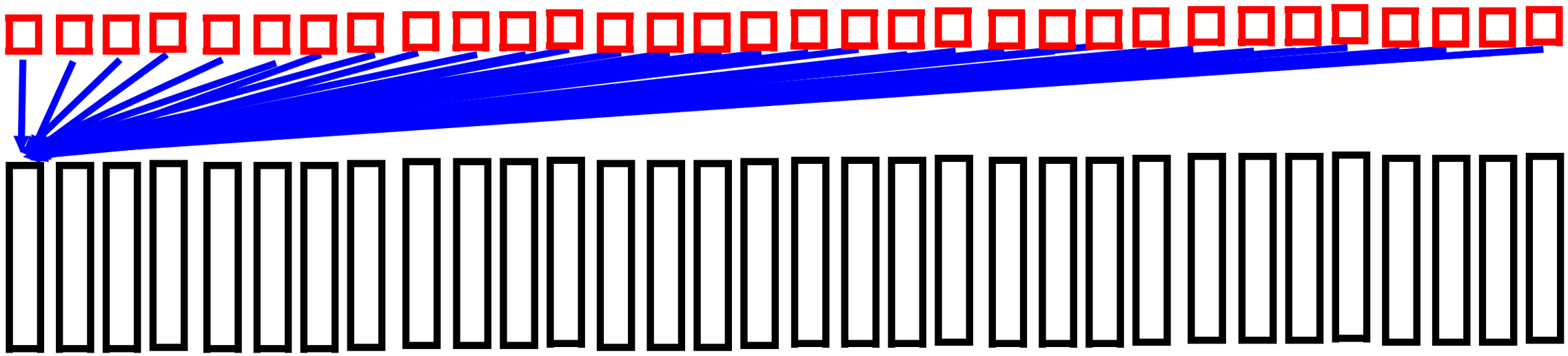


Shared Memory - bank conflicts

Now if we wished to assess every 32nd element of the array:

```
res[idx] = data[idx*32] * 5;
```

All that memory will sit in the same bank! This will make that access very slow as a single load instruction will require 32 serial loads from the one bank.



Shared Memory - bank conflicts

Sometimes we can rearrange the data to remove the bank conflict problem.

- Are 3 separate arrays of floats better than array of structs with 3 floats in each element?
- In our stride of 32 example that create all the request to go to one bank, could we pad the data within an extra integer every 32 elements (making a stride of 33)? Would this remove the bank conflicts?

Constant Memory

If your threads access the some constant data then good performance gains can be made by placing it in constant memory.
Constant memory is particularly optimized for cases when all the threads access the same constant data, so this data can be effectively "broadcast" to all the threads that require it. So for constant data it gives you register like performance without taking up any of your registers.

The down side is there is not much of it, so only 64KiB on current Nvidia GPUs. And it can't be dynamically allocated.

Constant Memory

Constant memory has global scope so just allocate it outside your kernels. So if you only have a few named short constants that are know at compile time your can just go :

```
__constant__ int myConstant = 42;
```

However if you wish to set it while running but before you execute your kernel you can:

```
__constant__ int myConstant;
// within your host code before the kernel is started
int myConstant_h = 42;
cudaMemcpyToSymbol(myConstant, &myConstant_h, sizeof(int));
```

Constant Memory

If we had a small array of data we can also do the much the same:

```
#define TSIZE 64
__constant__ float template[TSIZE];


// then within your host code before the kernel is started

float template_h[TSIZE];
// set the template values on the host in template_h

cudaMemcpyToSymbol(template, template_h, sizeof(float) * TSIZE);
```

Once you have defined and set your constants you can just use them within the kernel. e.g

```
float result = data[idx] * template[idx] + myConstant;
```

References

- Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking, Zhe Jia, Marco Maggioni, Benjamin Staiger, Daniele P. Scarpazza

https://arxiv.org/pdf/1804.06826.pdf