

GPU Computing: Advanced Memory

Dr Paul Richmond

<http://paulrichmond.shef.ac.uk>



The
University
Of
Sheffield.



GPU
RESEARCH
CENTER

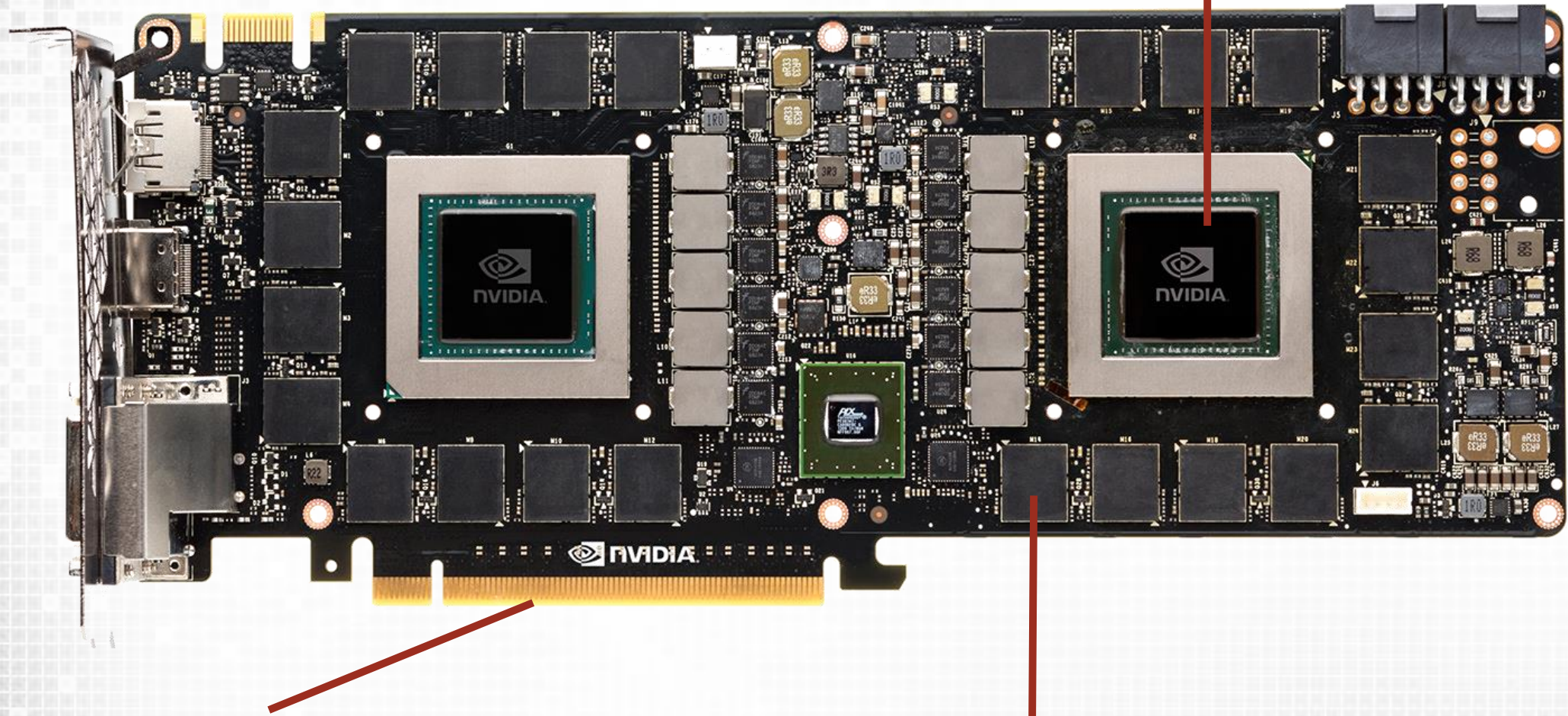
❑ Memory Hierarchy Overview

❑ Constant and Read-only Cache Memory

❑ Shared Memory

GPU Memory (GTX Titan Z)

Shared Memory, cache and registers



Host Memory (via PCIe)

GPU DRAM Memory

Simple Memory View

❑ Threads have access to;

❑ **Registers**

❑ Read/Write **per thread**

❑ **Local memory**

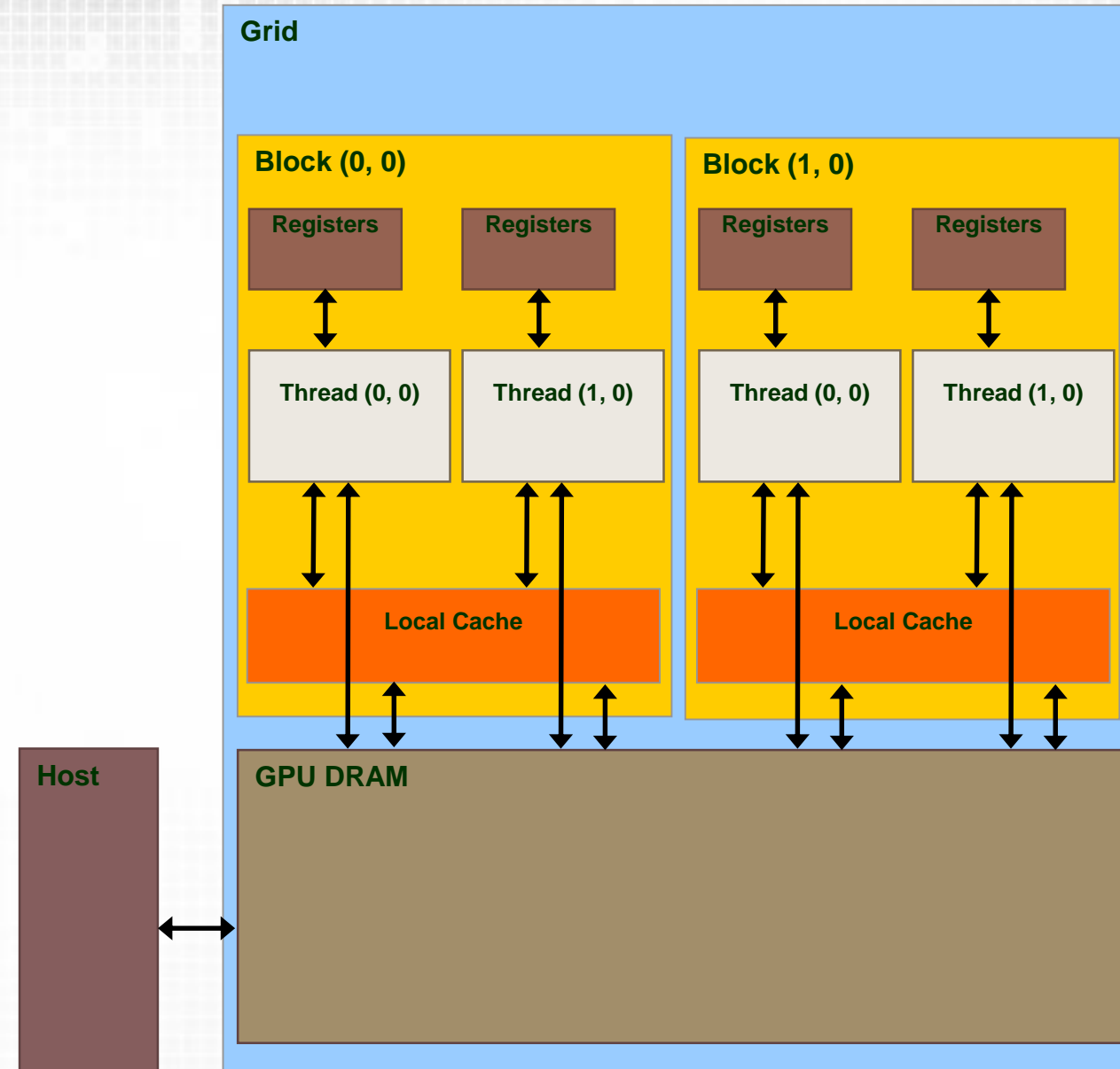
❑ Read/Write **per thread**

❑ **Local Cache**

❑ Read/Write **per block**

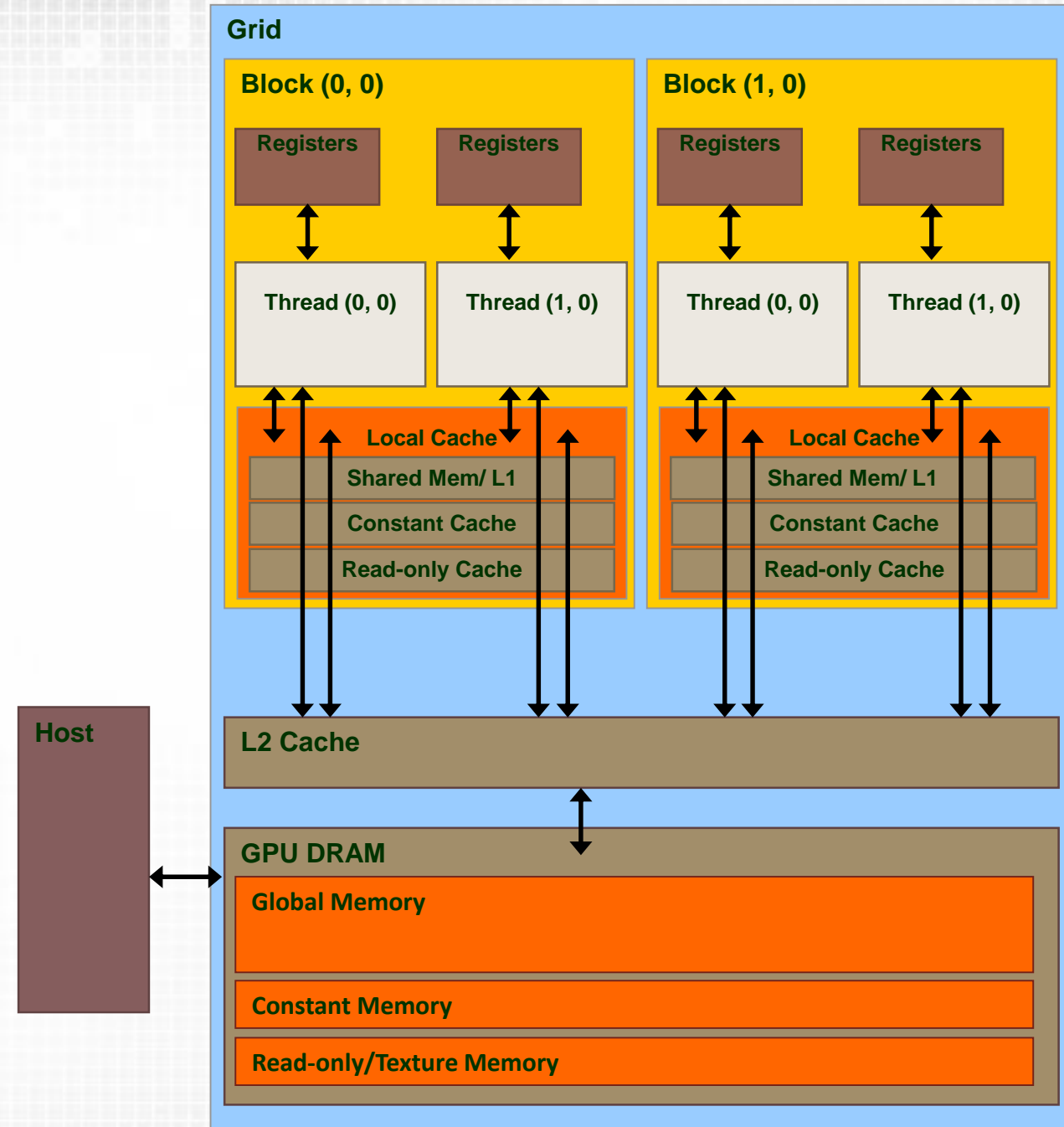
❑ **Main DRAM Memory**

❑ Read/Write **per grid**



Kepler Memory View

- ❑ Each Thread has access to
 - ❑ **Registers**
 - ❑ Read/Write per thread
 - ❑ **Local memory**
 - ❑ Read/Write per thread
 - ❑ Does not physically exist (reserved area in global memory)
 - ❑ Cached in L1
 - ❑ Overspill for registers
- ❑ Main DRAM Memory via cache
 - ❑ Global Memory
 - ❑ Via **L2 cache** and configurable per block **Shared Memory / L1 cache**
 - ❑ Constant Memory
 - ❑ Via **L2 cache** and per block **Constant cache**
 - ❑ Read-only/Texture Memory
 - ❑ Via **L2 cache** and per block **Read-only cache**



Memory Latencies

- ❑ What is the cost of accessing each area of memory?
 - ❑ On chip caches are MUCH lower latency

| | Cost (cycles) |
|---------------|-------------------------------------|
| Register | 1 |
| Global | 200-800 |
| Shared memory | ~1 |
| L1 | 1 |
| Constant | ~1 (if cached) |
| Read-only | 1 if cached (same as global if not) |

- ❑ Memory Hierarchy Overview
- ❑ Constant and Read-only Cache Memory
- ❑ Shared Memory

Constant Memory

☐ Constant Memory

- ☐ Stored in the devices global memory
- ☐ Read through the per SM constant cache
- ☐ Set at runtime
- ☐ When using correctly only 1/16 of the traffic compared to global loads

☐ When to use it?

- ☐ When small amounts of data are **read only**
- ☐ When values are **broadcast** to threads in a half warp (of 16 threads)
- ☐ Very fast when cache hit
- ☐ Very slow when no cache hit

☐ How to use

- ☐ Must be **statically** defined as a symbol using `__constant__` qualifier
- ☐ Value must be copied using **cudaMemcpyToSymbol**.

Constant Memory Broadcast

□.... When values are **broadcast** to threads in a half warp (groups of 16 threads)

```
__constant__ int my_const[16];

__global__ void vectorAdd() {
    int i = blockIdx.x;

    int value = my_const[i % 16];
}
```

```
__constant__ int my_const[16];

__global__ void vectorAdd() {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    int value = my_const[i % 16];
}
```

Which is good use of constant memory?

Constant Memory Broadcast

❑.... When values are **broadcast** to threads in a half warp (groups of 16 threads)

```
__constant__ int my_const[16];

__global__ void constant_test() {
    int i = blockIdx.x;

    int value = my_const[i % 16];
}
```

```
__constant__ int my_const[16];

__global__ void constant_test() {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    int value = my_const[i % 16];
}
```

Which is good use of constant memory?

- ❑ Best possible use of constant memory
- ❑ Every thread in half warp reads the same
 - ❑ Index based on `blockIdx`
- ❑ No serialisation
 - ❑ 1 read request for every thread!
- ❑ Other threads in the block will also hit cache

- ❑ Worst possible use of constant memory
- ❑ Every thread in half warp reads different value
 - ❑ Index based on `threadIdx`
- ❑ Each access will be serialised
 - ❑ 16 different read requests!
- ❑ Other threads in the block will likely miss the cache

Read-only Memory

- ❑ Useful where threads have good coherence
 - ❑ Encourages the compiler to use L2 Read-only cache
- ❑ Indicates to the compiler that the data is read-only
 - ❑ Using the `const` and `__restrict__` qualifiers
- ❑ Not the same as `__constant__` memory
 - ❑ Does not require broadcast reading

```
#define N 1024

__global__ void kernel(float const* __restrict__ buffer) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = buffer[i];
}

int main() {
    float *buffer;
    cudaMalloc(&buffer, N*sizeof(float));
    kernel << <grid, block >> >(buffer);
    cudaFree(buffer);
}
```


- ❑ Memory Hierarchy Overview
- ❑ Constant and Read-only Cache Memory
- ❑ Shared Memory

Shared Memory

- ❑ Its just another Cache, right?

 - ❑ User configurable

 - ❑ Requires manually loading and synchronising data

- ❑ Performance

 - ❑ Shared memory is very fast

 - ❑ Bandwidth > 1 TB/s

 - ❑ Latency ~10 cycles

- ❑ Block level computations

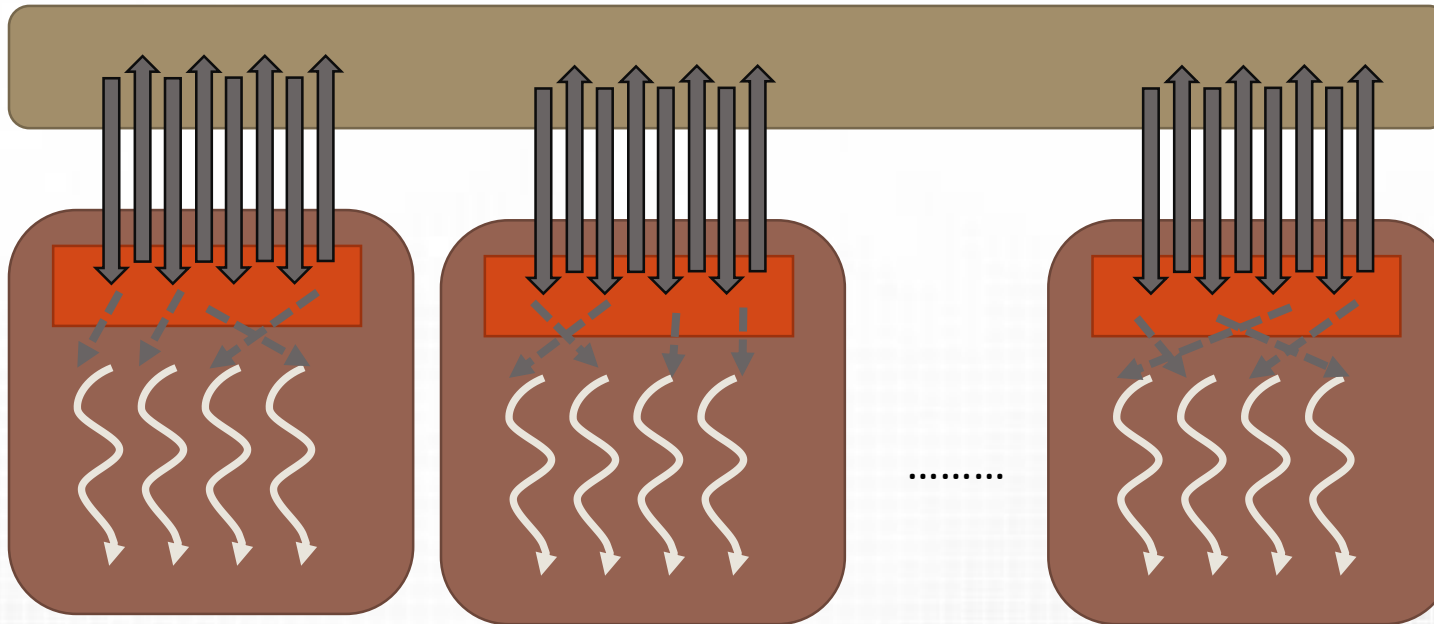
 - ❑ Allows data to be shared between threads in the same block

 - ❑ User configurable cache at the thread block level

 - ❑ Still no broader synchronisation beyond the level of thread blocks

Block Local Computation

- ❑ Partition data into groups that fit into shared memory
- ❑ Load subset of data into shared memory
- ❑ Perform computation on the subset
- ❑ Copy subset back to global memory



A Case for Shared Memory

```
__global__ void sum3_kernel(int *c, int *a)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int left, right;

    //load value at i-1
    left = 0;
    if (i > 0)
        left = a[i - 1];

    //load value at i+1
    right = 0;
    if (i < (N - 1))
        right = a[i + 1];

    c[i] = left + a[i] + right; //sum three values
}
```

A Case for Shared Memory

```
__global__ void sum3_kernel(int *c, int *a)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int left, right;

    //load value at i-1
    left = 0;
    if (i > 0)
        left = a[i - 1];

    //load value at i+1
    right = 0;
    if (i < (N - 1))
        right = a[i + 1];

    c[i] = left + a[i] + right; //sum three values
}
```

- ❑ Thread-local computation
- ❑ Bandwidth limited
 - ❑ Requires three loads per thread (at index $i-1$, i , and $i+1$)
- ❑ Wouldn't it be nice if we could load each value only once!

CUDA Shared memory

- ❑ Shared memory between threads in the same block can be defined using `__shared__`
- ❑ Shared variables are only accessible from within device functions
 - ❑ Not addressable in host code
- ❑ Must be careful to avoid race conditions
 - ❑ Multiple threads writing to the same shared memory variable
 - ❑ Results in undefined behaviour
 - ❑ Typically access shared memory using `threadIdx`
 - ❑ Thread level synchronisation is available through `__syncthreads()`
 - ❑ Ensures data is ready for access

```
__shared__ int s_data[BLOCK_SIZE];
```


Example

```
__global__ void sum3_kernel(int *c, int *a)
{
    __shared__ int s_data[BLOCK_SIZE];

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int left, right;

    s_data[threadIdx.x] = a[i];
    __syncthreads();

    //load value at i-1
    left = 0;
    if (i > 0){
        if (threadIdx.x > 0)
            left = s_data[threadIdx.x - 1];
        else
            left = a[i - 1];
    }

    //load value at i+1
    right = 0;
    if (i < (N - 1)){
        if (threadIdx.x < (BLOCK_SIZE-1))
            right = s_data[threadIdx.x + 1];
        else
            right = a[i + 1];
    }

    c[i] = left + s_data[threadIdx.x] + right; //sum
}
```

- ❑ Allocate a shared array
 - ❑ One integer element per thread
- ❑ Each thread loads a single item to shared memory
- ❑ Call `__syncthreads` to ensure shared memory data is populated by all threads
- ❑ Check boundary conditions for the edge of the block
- ❑ Load all elements through shared memory

Problems with Shared memory

- ❑ Shared memory is accessed in banks
 - ❑ A bank conflict occurs when two threads request addresses from the same bank
- ❑ In the example we saw the introduction of boundary conditions
 - ❑ Global loads still present at boundaries
 - ❑ We have introduced divergence in the code (remember the SIMD model)
 - ❑ This is even more prevalent in 2D examples where we *tile* data into shared memory

```
//boundary condition
left = 0;
if (i > 0) {
    if (threadIdx.x > 0)
        left = s_data[threadIdx.x - 1];
    else
        left = a[i - 1];
}
```

Summary

- ❑ The CUDA Memory Hierarchy varies between hardware generations
- ❑ Utilisation of local caches can have a big impact on the expected performance (1 cycle vs. 100s)
- ❑ Constant cache is good for small read only data accessed with a broadcast pattern
- ❑ Read-Only cache is for caching data read where access patterns are closely aligned in nearby threads
- ❑ Shared memory is user configurable cache, the moving of data and synchronisation are left to the user (you).