The pattern of global memory access is vital
The pattern of global memory access is vital

The memory bus has a high latency, but a large bandwidth
The pattern of global memory access is vital.

The memory bus has a high latency, but a large bandwidth.

We have to wait a long time for bytes to arrive; but then they arrive in large chunks.
The pattern of global memory access is vital

The memory bus has a high latency, but a large bandwidth

We have to wait a long time for bytes to arrive; but then they arrive in large chunks

Memory is set up to deliver, say, 64 bytes at a time (512 bit bus)
The pattern of global memory access is vital.

The memory bus has a high latency, but a large bandwidth.

We have to wait a long time for bytes to arrive; but then they arrive in large chunks.

Memory is set up to deliver, say, 64 bytes at a time (512 bit bus).

And programs often ask for large chunks of data in parallel, e.g., working in parallel on an array.
The pattern of global memory access is vital

The memory bus has a high latency, but a large bandwidth

We have to wait a long time for bytes to arrive; but then they arrive in large chunks

Memory is set up to deliver, say, 64 bytes at a time (512 bit bus)

And programs often ask for large chunks of data in parallel, e.g., working in parallel on an array

64 bytes is 16 (half-warp) four-byte integers or 16 single precision floats
The pattern of global memory access is vital.

The memory bus has a high latency, but a large bandwidth.

We have to wait a long time for bytes to arrive; but then they arrive in large chunks.

Memory is set up to deliver, say, 64 bytes at a time (512 bit bus).

And programs often ask for large chunks of data in parallel, e.g., working in parallel on an array.

64 bytes is 16 (half-warp) four-byte integers or 16 single precision floats.

So a warp could be satisfied by just two reads.
If the reads are nicely arranged, a single read supplies many cores simultaneously
As long as your code is arranged to do this
Topics
GPUs

As long as your code is arranged to do this

There are many rules imposed by the hardware to make this kind of memory access coalescence work
As long as your code is arranged to do this

There are many rules imposed by the hardware to make this kind of memory access *coalescence* work

Such as alignments of areas of memory; the order in which neighbouring cores access memory; and so on
As long as your code is arranged to do this

There are many rules imposed by the hardware to make this kind of memory access *coalescence* work

Such as alignments of areas of memory; the order in which neighbouring cores access memory; and so on

If you get it right, reading 16 integers in parallel is as fast as reading a single integer
As long as your code is arranged to do this

There are many rules imposed by the hardware to make this kind of memory access *coalescence* work

Such as alignments of areas of memory; the order in which neighbouring cores access memory; and so on

If you get it right, reading 16 integers in parallel is as fast as reading a single integer

If you get it wrong, it can be 16 times as slow
\[ x = p[16\times me] \]
x = p[32]

x = p[16*me]
#include <stdio.h>
__global__ void setarray(int p[])
{
    int k = blockIdx.x * blockDim.x + threadIdx.x;
p[k] = k*k;
}

int main(void)
{
    int i, *dm, m[1024];
cudaMalloc(&dm, 1024*sizeof(int));
setarray<<<16,64>>>(dm);
cudaMemcpy(m, dm, 1024*sizeof(int),
            cudaMemcpyDeviceToHost);
    for (i = 0; i < 1024; i++)
        printf("m[%d] = %d\n", i, m[i]);
    return 0;
}
Back to the example: $d_m$ is the address of a chunk of memory on the device
Back to the example: $dm$ is the address of a chunk of memory on the device.

The device memory is separate from the CPU memory, so we need special functions to allocate memory on the device.
Back to the example: $dm$ is the address of a chunk of memory on the device.

The device memory is separate from the CPU memory, so we need special functions to allocate memory on the device.

And we need explicit copies to get the data in and out of the coprocessor.
As always, data copies are time consuming, so we want to minimise them relative to computation time.
As always, data copies are time consuming, so we want to minimise them relative to computation time.

We are used to the idea that the overhead can be so large that it is faster to do a computation sequentially on the CPU rather than send it to the GPU.
As always, data copies are time consuming, so we want to minimise them relative to computation time.

We are used to the idea that the overhead can be so large that it is faster to do a computation sequentially on the CPU rather than send it to the GPU.

The reverse is also true: if the data are on the GPU, it can be faster overall to use one of the wimpy GPU cores for a computation rather than copy back and forth to the CPU.
As always, data copies are time consuming, so we want to minimise them relative to computation time.

We are used to the idea that the overhead can be so large that it is faster to do a computation sequentially on the CPU rather than send it to the GPU.

The reverse is also true: if the data are on the GPU, it can be faster overall to use one of the wimpy GPU cores for a computation rather than copy back and forth to the CPU.

This kind of computation vs. data movement judgement happens a lot when programming GPUs.
In this example, we have only 16 blocks, so this would not be so good for a coprocessor with, say, 20 streaming multiprocessors.
In this example, we have only 16 blocks, so this would not be so good for a coprocessor with, say, 20 streaming multiprocessors.

Real code would either simply have more blocks, or would interrogate the device to see how many multiprocessors it has and adjust accordingly.
In this example, we have only 16 blocks, so this would not be so good for a coprocessor with, say, 20 streaming multiprocessors.

Real code would either simply have more blocks, or would interrogate the device to see how many multiprocessors it has and adjust accordingly.

Nvidia recommend “1000s” of blocks to allow scaling across many generations of GPU cards.
In this example, we have only 16 blocks, so this would not be so good for a coprocessor with, say, 20 streaming multiprocessors.

Real code would either simply have more blocks, or would interrogate the device to see how many multiprocessors it has and adjust accordingly.

Nvidia recommend “1000s” of blocks to allow scaling across many generations of GPU cards.

Exercise: but you wouldn’t want more than 32 blocks in our small example. Why?