GPUs

CUDA

Memory affects the execution of threads

GPUs
CUDA

Memory affects the execution of threads

Thread blocks are scheduled by the hardware on multiprocessors and more than one block can be simultaneously scheduled on a multiprocessor, thus sharing its resources, particularly shared memory and registers

GPUs
CUDA

Memory affects the execution of threads

Thread blocks are scheduled by the hardware on multiprocessors and more than one block can be simultaneously scheduled on a multiprocessor, thus sharing its resources, particularly shared memory and registers

So the pattern of use of shared memory can put a limit on the number of blocks in the grid, thus a limit on the rate of execution

GPUs
CUDA

Memory affects the execution of threads

Thread blocks are scheduled by the hardware on multiprocessors and more than one block can be simultaneously scheduled on a multiprocessor, thus sharing its resources, particularly shared memory and registers

So the pattern of use of shared memory can put a limit on the number of blocks in the grid, thus a limit on the rate of execution

Similarly, there is a limit on the number of threads per block: up to 65536 in one of the above GPUs

GPUs

CUDA

GPUs offers a huge amount of processing power at low cost, but in a way that is extremely sensitive to memory access

GPUs
CUDA

GPUs offers a huge amount of processing power at low cost, but in a way that is extremely sensitive to memory access

It is easy to get started with CUDA as it is basically C, but you do have to be very aware of the properties of memory

GPUs
CUDA

Modern GPUs support *unified memory spaces*

GPUs
CUDA

Modern GPUs support *unified memory spaces*

This allows you to use a single virtual address space for both host and device memory and not worry which is which (a bit like VSM)

GPUs
CUDA

Modern GPUs support *unified memory spaces*

This allows you to use a single virtual address space for both host and device memory and not worry which is which (a bit like VSM)

A hidden mechanism copies data between CPU and GPU as necessary

GPUs
CUDA

Modern GPUs support *unified memory spaces*

This allows you to use a single virtual address space for both host and device memory and not worry which is which (a bit like VSM)

A hidden mechanism copies data between CPU and GPU as necessary

Exercise Is this a good idea?

GPUs
CUDA

Modern GPUs support *unified memory spaces*

This allows you to use a single virtual address space for both host and device memory and not worry which is which (a bit like VSM)

A hidden mechanism copies data between CPU and GPU as necessary

Exercise Is this a good idea?

(Shortly we will see some systems that have physically shared memory)

GPUs
Memory

Next, there is the extra problem shared by all coprocessors: memory bandwidth between the main CPU and the coprocessor

GPUs

Memory

Next, there is the extra problem shared by all coprocessors: memory bandwidth between the main CPU and the coprocessor

Copying data in and out of the GPU is significantly time consuming

GPUs

Memory

Next, there is the extra problem shared by all coprocessors: memory bandwidth between the main CPU and the coprocessor

Copying data in and out of the GPU is significantly time consuming

So we need to worry about data movement between the GPU and the main CPU

GPUs
Memory

Next, there is the extra problem shared by all coprocessors: memory bandwidth between the main CPU and the coprocessor

Copying data in and out of the GPU is significantly time consuming

So we need to worry about data movement between the GPU and the main CPU

And, if possible, overlap data transfers with GPU and CPU computation

GPUs

Memory

Next, there is the extra problem shared by all coprocessors: memory bandwidth between the main CPU and the coprocessor

Copying data in and out of the GPU is significantly time consuming

So we need to worry about data movement between the GPU and the main CPU

And, if possible, overlap data transfers with GPU and CPU computation

And overlap CPU and GPU computations

GPUs

Memory

We often forget that the system also has to copy the *code*, ie., the kernels, to the GPU memory, too

GPUs

Memory

We often forget that the system also has to copy the *code*, ie., the kernels, to the GPU memory, too

The cost of this is usually small relative to the cost of copying data, but it’s another reminder that the GPU’s memory is separate from the CPU’s

GPUs

Memory

We often forget that the system also has to copy the *code*, ie., the kernels, to the GPU memory, too

The cost of this is usually small relative to the cost of copying data, but it’s another reminder that the GPU’s memory is separate from the CPU’s

But a recent trend is to integrate the GPU onto the same package as the CPU (or vice-versa!)

GPUs

Memory

We often forget that the system also has to copy the *code*, ie., the kernels, to the GPU memory, too

The cost of this is usually small relative to the cost of copying data, but it's another reminder that the GPU's memory is separate from the CPU's

But a recent trend is to integrate the GPU onto the same package as the CPU (or vice-versa!)

Using lots of transistors!

GPUs

Memory

For example, AMD’s Kaveri is a CPU+GPU on the one chip

GPUs

Memory

For example, AMD’s Kaveri is a CPU+GPU on the one chip

4 CPU cores and 512 GPU cores that share cache and main memory

GPUs
Memory

For example, AMD's Kaveri is a CPU+GPU on the one chip

4 CPU cores and 512 GPU cores that share cache and main memory

Of course, this changes all the memory access vs. compute balances, so needing you to revise your code

GPUs

Memory

For example, AMD's Kaveri is a CPU+GPU on the one chip

4 CPU cores and 512 GPU cores that share cache and main memory

Of course, this changes all the memory access vs. compute balances, so needing you to revise your code

This is an example of a *Heterogeneous System Architecture* (HSA)

GPUs

Memory

The idea is more of a symmetry between the CPU and GPU: the GPU is not just a coprocessor

GPUs

Memory

The idea is more of a symmetry between the CPU and GPU: the GPU is not just a coprocessor

The GPU can now pass tasks back to the CPU to do

GPUs
Memory

The idea is more of a symmetry between the CPU and GPU: the GPU is not just a coprocessor

The GPU can now pass tasks back to the CPU to do

Accompanying this is a new low-level virtual architecture *HSA Intermediate Layer* (HSAIL) that will be used to implement higher-level abstractions like OpenCL

GPUs
Memory

The idea is more of a symmetry between the CPU and GPU: the GPU is not just a coprocessor

The GPU can now pass tasks back to the CPU to do

Accompanying this is a new low-level virtual architecture *HSA Intermediate Layer* (HSAIL) that will be used to implement higher-level abstractions like OpenCL

In a similar way, Apple's M1 architecture has CPU and GPU *and memory* on the same chip, further confusing the memory vs. compute costs question

GPUs

CUDA

Back to CUDA

GPUs
CUDA

Back to CUDA

Here is an example of trivial CUDA code, `prog.cu`

GPUs
CUDA

Back to CUDA

Here is an example of trivial CUDA code, `prog.cu`

(Checking return values and tidying up omitted for brevity)

CUDA

```
#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;
}
```

GPUs

CUDA

This starts 16 blocks, each containing 64 threads, each thread runs the kernel `setarray`

GPUs
CUDA

This starts 16 blocks, each containing 64 threads, each thread runs the kernel `setarray`

Each invocation of `setarray` gets the same pointer to some global memory allocated on the GPU

GPUs

CUDA

This starts 16 blocks, each containing 64 threads, each thread runs the kernel `setarray`

Each invocation of `setarray` gets the same pointer to some global memory allocated on the GPU

Each computes a different value for the index `k`, and each sets a different element of the array

GPUs
CUDA

This starts 16 blocks, each containing 64 threads, each thread runs the kernel `setarray`

Each invocation of `setarray` gets the same pointer to some global memory allocated on the GPU

Each computes a different value for the index `k`, and each sets a different element of the array

This assignment is a memory bottleneck that will take a relatively long time to complete

GPUs

CUDA

CUDA programmers try to mitigate the memory bottleneck by ensuring there are lots of threads

GPUs
CUDA

CUDA programmers try to mitigate the memory bottleneck by ensuring there are lots of threads

Within a block, a warp of 32 threads is scheduled to run

GPUs
CUDA

CUDA programmers try to mitigate the memory bottleneck by ensuring there are lots of threads

Within a block, a warp of 32 threads is scheduled to run

These run (in SIMD) until they would have to wait for a lengthy memory access to complete: the assignment to `p` in the example

GPUs
CUDA

CUDA programmers try to mitigate the memory bottleneck by ensuring there are lots of threads

Within a block, a warp of 32 threads is scheduled to run

These run (in SIMD) until they would have to wait for a lengthy memory access to complete: the assignment to `p` in the example

Rather than simply waiting for the memory, this warp is put aside *while the memory access is still progressing* and another warp (from this block or another block on the same multiprocessor) is scheduled to run instead

GPUs
CUDA

Thus keeping the multiprocessor busy computing

GPUs
CUDA

Thus keeping the multiprocessor busy computing

When the memory access has completed, the original warp can be run again

GPUs
CUDA

Thus keeping the multiprocessor busy computing

When the memory access has completed, the original warp can be run again

All these scheduling decisions and actions are done by the hardware!

GPUs
CUDA

Thus keeping the multiprocessor busy computing

When the memory access has completed, the original warp can be run again

All these scheduling decisions and actions are done by the hardware!

Exercise Compare with hyperthreading as a way of keeping CPUs busy

GPUs
CUDA

Thus we want a lot of threads to schedule between as they run then wait for memory

GPUs
CUDA

Thus we want a lot of threads to schedule between as they run then wait for memory

If we don’t have enough threads the cores will be idle during their wait for memory

GPUs
CUDA

Thus we want a lot of threads to schedule between as they run then wait for memory

If we don’t have enough threads the cores will be idle during their wait for memory

Ideally each block should have a multiple of 32 threads, whenever possible, to get the most from the multiprocessor

GPUs

CUDA

Thus we want a lot of threads to schedule between as they run then wait for memory

If we don’t have enough threads the cores will be idle during their wait for memory

Ideally each block should have a multiple of 32 threads, whenever possible, to get the most from the multiprocessor

For example, running just 16 threads means half of the warp is lying idle

GPUs

CUDA

Additionally, multiprocessors are given whole blocks to execute

GPUs
CUDA

Additionally, multiprocessors are given whole blocks to execute

So we want at least as many blocks as multiprocessors, to keep all the hardware busy

GPUs
CUDA

Additionally, multiprocessors are given whole blocks to execute

So we want at least as many blocks as multiprocessors, to keep all the hardware busy

Thus it's good to have lots of threads per block and lots of blocks per multiprocessor to provide lots of choice of warps to schedule

GPUs

CUDA

How many blocks and how many threads per block?

GPUs

CUDA

How many blocks and how many threads per block?

It depends on how the program accesses memory: e.g., the use of shared resources like block shared memory might be a factor

GPUs
CUDA

From the NVIDIA documentation:

- How many blocks?
  - At least one block per SM to keep every SM occupied
  - At least two blocks per SM so something can run if block is waiting for a synchronization to complete
  - Many blocks for scalability to larger and future GPUs
- How many threads?
  - At least 192 threads per SM to hide read after write latency of 11 cycles (not necessarily in same block)
  - Use many threads to hide global memory latency
  - Too many threads exhausts registers and shared memory
  - Thread count a multiple of warp size
  - Typically, between 64 and 256 threads per block

GPUs
CUDA

The programmer might want to experiment to find the best combination of numbers of blocks and threads per block for the particular GPU they are running on

GPUs
CUDA

The programmer might want to experiment to find the best combination of numbers of blocks and threads per block for the particular GPU they are running on

There are profiling tools and spreadsheets available to help you make this decision

GPUs

CUDA

The programmer might want to experiment to find the best combination of numbers of blocks and threads per block for the particular GPU they are running on

There are profiling tools and spreadsheets available to help you make this decision

And to add to the complexity: later versions of CUDA allow multiple different kernels to run concurrently (i.e., it schedules between kernels), so supplying more blocks and more threads to keep the hardware busy

GPUs

CUDA

The programmer might want to experiment to find the best combination of numbers of blocks and threads per block for the particular GPU they are running on

There are profiling tools and spreadsheets available to help you make this decision

And to add to the complexity: later versions of CUDA allow multiple different kernels to run concurrently (i.e., it schedules between kernels), so supplying more blocks and more threads to keep the hardware busy

CUDA kernels run asynchronously from the CPU

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

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

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

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

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

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)

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

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

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

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

GPUs

Memory Coalescence

And the *pattern* of global memory access is vital, too

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)

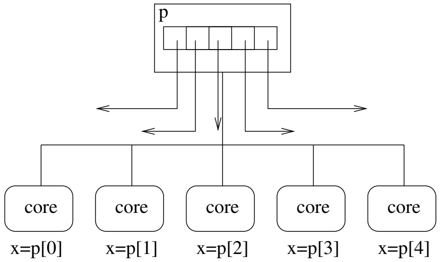
And programs often ask for large chunks of data in parallel, e.g., working in parallel on an array

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

So a warp could be satisfied by just two reads

GPUs

Memory Coalescence



```
x = p[me]
```

If the reads are nicely arranged, a single read supplies many cores simultaneously: this is memory access *coalescence* (as discussed earlier in vector architectures)

GPUs

Memory Coalescence

As long as your code can do this

GPUs

Memory Coalescence

As long as your code can do this

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

GPUs

Memory Coalescence

As long as your code can 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

GPUs

Memory Coalescence

As long as your code can 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

GPUs

Memory Coalescence

As long as your code can do this

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

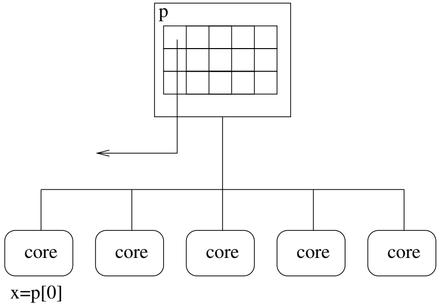
Such as alignments of areas of memory; the order in which neighbouring cores access memory; and so on

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

GPUs

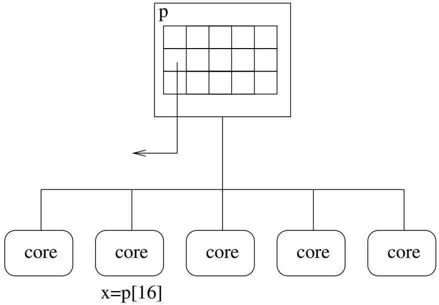
Memory Coalescence



```
x = p[16*me]
```

GPUs

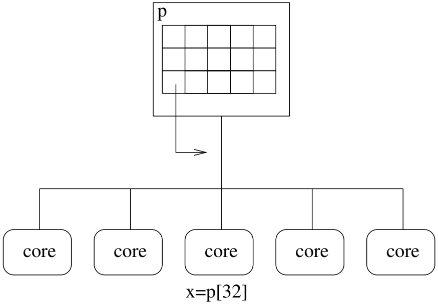
Memory Coalescence



```
x = p[16*me]
```

GPUs

Memory Coalescence



```
x = p[16*me]
```

GPUs

Memory Coalescence

In this case, it might be faster to read coalesced chunks of memory into the block shared memory, and then have cores read their values from there

GPUs

Memory Coalescence

In this case, it might be faster to read coalesced chunks of memory into the block shared memory, and then have cores read their values from there

Awkward coding, but this is how you can get good performance

CUDA

```
#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;
}
```

GPUs
CUDA

Back to the example: `dm` is the address of a chunk of memory on the device

GPUs
CUDA

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

GPUs
CUDA

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

GPUs
Memory

As always, data copies are time consuming, so we want to minimise them relative to computation time

GPUs

Memory

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

GPUs

Memory

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

GPUs
Memory

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

GPUs

CUDA

In this example, we have only 16 blocks, so this would not be so good for a coprocessor with, say, 20 streaming multiprocessors

GPUs
CUDA

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

GPUs
CUDA

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

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

GPUs

GPUs are becoming an ever more important method of computation

GPUs

GPUs are becoming an ever more important method of computation

Even in phones: ARM’s Mali GPU now has OpenCL support

GPUs

GPUs are becoming an ever more important method of computation

Even in phones: ARM’s Mali GPU now has OpenCL support

GPUs are good for phones as they give a good amount of processing power for only a small amount of energy used

GPUs
OpenCL

OpenCL takes a wider view of computation than CUDA

GPUs

OpenCL

OpenCL takes a wider view of computation than CUDA

While CUDA is explicitly about GPU computation, OpenCL tries to abstract away from the hardware and provide the programmer with a generic programming interface, independent of the underlying hardware

GPUs
OpenCL

OpenCL takes a wider view of computation than CUDA

While CUDA is explicitly about GPU computation, OpenCL tries to abstract away from the hardware and provide the programmer with a generic programming interface, independent of the underlying hardware

It tries hard not to assume there is a GPU coprocessor specifically, but just some “compute resource” coprocessor

GPUs
OpenCL

OpenCL takes a wider view of computation than CUDA

While CUDA is explicitly about GPU computation, OpenCL tries to abstract away from the hardware and provide the programmer with a generic programming interface, independent of the underlying hardware

It tries hard not to assume there is a GPU coprocessor specifically, but just some “compute resource” coprocessor

OpenCL is provided as a library that is callable from standard C (and other languages), thus not needing a special compiler

GPUs
OpenCL

Things that CUDA has special syntax for (in particular kernel setup and launch) are done via normal function calls in OpenCL

GPUs

OpenCL

Things that CUDA has special syntax for (in particular kernel setup and launch) are done via normal function calls in OpenCL

OpenCL kernel code is kept in separate files from the C/C++ CPU code

GPUs
OpenCL

Things that CUDA has special syntax for (in particular kernel setup and launch) are done via normal function calls in OpenCL

OpenCL kernel code is kept in separate files from the C/C++ CPU code

Kernel code is read, compiled and executed by calling functions in the CPU code

GPUs

OpenCL

Things that CUDA has special syntax for (in particular kernel setup and launch) are done via normal function calls in OpenCL

OpenCL kernel code is kept in separate files from the C/C++ CPU code

Kernel code is read, compiled and executed by calling functions in the CPU code

Much like the shader code in OpenGL and the like

GPUs

OpenCL

In being generic, it is harder to use than CUDA, which does one thing well

GPUs
OpenCL

In being generic, it is harder to use than CUDA, which does one thing well

CUDA can produce fast code, particularly if tuned to the specific hardware

GPUs
OpenCL

In being generic, it is harder to use than CUDA, which does one thing well

CUDA can produce fast code, particularly if tuned to the specific hardware

But the hardware must be an NVIDIA card

GPUs

OpenCL

In being generic, it is harder to use than CUDA, which does one thing well

CUDA can produce fast code, particularly if tuned to the specific hardware

But the hardware must be an NVIDIA card

Current OpenCL compilers produce code that runs universally but at sometimes uninspiring speeds (so code still needs the machine-specific tuning that OpenCL was supposed to avoid)

GPUs

OpenCL

In being generic, it is harder to use than CUDA, which does one thing well

CUDA can produce fast code, particularly if tuned to the specific hardware

But the hardware must be an NVIDIA card

Current OpenCL compilers produce code that runs universally but at sometimes uninspiring speeds (so code still needs the machine-specific tuning that OpenCL was supposed to avoid)

And there are features in the OpenCL programming model that reveal that the designers were still thinking of GPUs underneath the supposed genericity