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

The cost of this is small relative to the cost of copying data, but it’s another reminder that the GPU’s memory is separate from the CPU’s
We often forget that the system also has to copy the program, i.e., the kernels, to the GPU memory, too.

The cost of this is 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!)
We often forget that the system also has to copy the *program*, ie., the kernels, to the GPU memory, too.

The cost of this is 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!
For example, AMD’s Kaveri is a CPU+GPU
For example, AMD’s Kaveri is a CPU+GPU

4 CPU cores and 512 GPU cores that share cache and main memory
For example, AMD’s Kaveri is a CPU+GPU

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 new code
For example, AMD’s Kaveri is a CPU+GPU

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 new code

This is an example of a *Heterogeneous System Architecture* (HSA)
The idea is more of a symmetry between the CPU and GPU: the GPU is not just a coprocessor
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.
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.
Topics

GPUs

Back to CUDA
Back to CUDA

Here is an example of trivial CUDA code
Back to CUDA

Here is an example of trivial CUDA code

(Checking return values and tidying up omitted for brevity)
#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<<<16,64>>>(dm);
    cudaMemcpy(dm, m, 1024*sizeof(int), cudaMemcpyDeviceToHost);
    for (i = 0; i < 1024; i++)
        printf("m[%d] = %d\n", i, m[i]);
    return 0;
}
This starts 16 blocks, each containing 64 threads, each thread runs the kernel `setarray`
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
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 $k$, and each sets a different element of the array
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 $k$, and each sets a different element of the array

This assignment is a memory bottleneck that will take a relatively long time to complete
CUDA programmers try to mitigate the memory bottleneck by ensuring there are lots of threads.
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.
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.
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 or another block on the same multiprocessor) is scheduled to run instead.
Thus keeping the multiprocessor busy computing
Thus keeping the multiprocessor busy computing

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

GPUs

Thus we want a lot of threads to schedule between as they run then wait for memory
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
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.
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 multiprocessor is lying idle
Additionally, multiprocessors are given whole blocks to execute.
Additionally, multiprocessors are given whole blocks to execute.

So we want at least as many blocks as multiprocessors, to keep all the hardware busy.
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
How many blocks and how many threads per block?
How many blocks and how many threads per block?

It depends on how the program accesses memory: e.g., the use of block shared memory might be a factor.
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
The programmer might want to experiment to find the best combination of numbers of blocks and threads per block.
The programmer might want to experiment to find the best combination of numbers of blocks and threads per block.

There are profiling tools and spreadsheets available to help you make this decision.
The programmer might want to experiment to find the best combination of numbers of blocks and threads per block.

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.
The programmer might want to experiment to find the best combination of numbers of blocks and threads per block. 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.