中国科学院大学计算机学院专业选修课

GPU架构与编程

第五课：CUDA编程（三）

赵地
中科院计算所
2025年秋季学期

讲授内容

- Floating-Point Considerations
- GPU as Part of the PC Architecture
- Efficient Host-Device Data Transfer
- Computational Thinking
- Dynamic Parallelism
- CUDA libraries

讲授内容：Floating-Point Considerations

① Floating-Point Precision and Accuracy

② Numerical Stability

What is IEEE floating-point format?

- An industrywide standard for representing floating-point numbers to ensure that hardware from different vendors generate results that are consistent with each other
- A floating point binary number consists of three parts:
  - sign (S), exponent (E), and mantissa (M)
  - Each (S, E, M) pattern uniquely identifies a floating point number
- For each bit pattern, its IEEE floating-point value is derived as:
  - value = $(-1)^S * M * \{2^E\}$, where $1.0 \le M < 10.0_B$
- The interpretation of S is simple: S=0 results in a positive number and S=1 a negative number

Normalized Representation

- Specifying that $1.0_B \leq M < 10.0_B$ makes the mantissa value for each floating point number unique.
  - For example, the only mantissa value allowed for 0.5_D is M =1.0
    - $0.5_D = 1.0_B * 2^{-1}$
  - Neither $10.0_B * 2^{-2}$ nor $0.1_B * 2^{0}$ qualifies
- Because all mantissa values are of the form 1.XX…, one can omit the "1." part in the representation.
  - The mantissa value of 0.5_D in a 2-bit mantissa is 00, which is derived by omitting "1." from 1.00.
  - Mantissa without implied 1 is called the ***fraction***

Exponent Representation

- In an n-bit exponent representation, $2^{n-1}-1$ is added to its 2's complement representation to form its excess representation.
  - See Table for a 3-bit exponent representation
- A simple unsigned integer comparator can be used to compare the magnitude of two FP numbers
- Symmetric range for +/- exponents (111 reserved)

2's complement	Actual decimal	Excess-3
000	0	011
001	1	100
010	2	101
011	3	110
100	(reserved pattern)	111
101	-3	000
110	-2	001
111	-1	010

A simple, hypothetical 5-bit FP format

- Assume 1-bit S, 2-bit E, and 2-bit M
  - $0.5D = 1.00_B * 2^{-1}$
  - 0.5D = 0 00 00, where S = 0, E = 00, and M = (1.)00

2's complement	Actual decimal	Excess-1
00	0	01
01	1	10
10	(reserved pattern)	11
11	-1	00

Representable Numbers

- The representable numbers of a given format is the set of all numbers that can be exactly represented in the format.
- See Table for representable numbers of an unsigned 3-bit integer format

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Representable Numbers of a 5-bit Hypothetical IEEE Format

		Non-zero		Abrupt underflow		Gradual underflow	
E	M	S=0	S=1	S=0	S=1	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	0	0	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern						

Flush to Zero

- Treat all bit patterns with E=0 as 0.0
  - This takes away several representable numbers near zero and lump them all into 0.0
  - For a representation with large M, a large number of representable numbers will be removed

Flush to Zero

		No-zero		Flush to Zero		Denormalized	
E	M	S=0	S=1	S=0	S=1	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	0	0	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern						

Why is flushing to zero problematic?

- Many physical model calculations work on values that are very close to zero
  - Dark (but not totally black) sky in movie rendering
  - Small distance fields in electrostatic potential calculation
  - …
- Without Denormalization, these calculations tend to create artifacts that compromise the integrity of the models

Denormalized Numbers

- The actual method adopted by the IEEE standard is called "denormalized numbers" or "gradual underflow".
  - The method relaxes the normalization requirement for numbers very close to 0.
  - Whenever E=0, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX: In general, if the n-bit exponent is 0, the value is $0.M * 2^{-2^{(n-1)}+2}$

Denormalization

		No-zero		Flush to Zero		Denormalized	
E	M	S=0	S=1	S=0	S=1	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	0	0	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern						

IEEE 754 Format and Precision

- Single Precision
  - 1-bit sign, 8 bit exponent (bias-127 excess), 23 bit fraction
- Double Precision
  - 1-bit sign, 11-bit exponent (1023-bias excess), 52 bit fraction
  - The largest error for representing a number is reduced to $1/2^{29}$ of single precision representation

Special Bit Patterns

exponent	mantissa	meaning
11…1	≠ 0	NaN
11…1	=0	$(-1)^S * \infty$
00…0	≠0	denormalized
00…0	=0	0

- An ∞ can be created by overflow, e.g., divided by zero. Any representable number divided by +∞ or -∞ results in 0.
- NaN (Not a Number) is generated by operations whose input values do not make sense, for example, 0/0, 0*∞, ∞/∞, ∞ - ∞.
  - Also used to for data that has not been properly initialized in a program.
  - Signaling NaNs (SNaNs) are represented with most significant mantissa bit cleared whereas quiet NaNs are represented with most significant mantissa bit set.

Floating Point Accuracy and Rounding

- The accuracy of a floating point arithmetic operation is measured by the maximal error introduced by the operation.
- The most common source of error in floating point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding.
- Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly.

Rounding and Error

- Assume our 5-bit representation, consider

$1.0*2^{-2}$ (0, 00, 01) + $1.00*2^{1}$ (0, 10, 00)

exponent is 00 →denorm

- The hardware needs to shift the mantissa bits in order to align the correct bits with equal place value

$0.001*2^{1}$ (0, 00, 0001) + $1.00*2^{1}$ (0, 10, 00)

The ideal result would be $1.001 * 2^{1}$ (0, 10, 001) but this would require 3 mantissa bits!

Rounding and Error

- In some cases, the hardware may only perform the operation on a limited number of bits for speed and area cost reasons
  - An adder may only have 3 bit positions in our example so the first operand would be treated as a 0.00

$0.001*2^1$ (0, 00, 0001) + $1.00*2^1$ (0, 10, 00)

Error Measure

- If a hardware adder has at least two more bit positions than the total (both implicit and explicit) number of mantissa bits, the error would never be more than half of the place value of the mantissa
  - 0.001 in our 5-bit format
- We refer to this as 0.5 ULP (Units in the Last Place, the value 2^{-23}, which is approximately 1.192093e-07.)
  - If the hardware is designed to perform arithmetic and rounding operations perfectly, the most error that one should introduce should be no more than 0.5 ULP
  - The error is limited by the precision for this case.

Order of Operations Matter

- Floating point operations are not strictly associative
- The root cause is that some times a very small number can disappear when added to or subtracted from a very large number:

(Large + Small) + Small ≠ Large + (Small + Small)

Algorithm Considerations

- Sequential sum

$1.00*2^0 + 1.00*2^0 + 1.00*2^{-2} + 1.00*2^{-2}$

$= 1.00*2^1 + 1.00*2^{-2} + 1.00*2^{-2}$

$= 1.00*2^1 + 1.00*2^{-2}$

$= 1.00*2^1$

- Parallel reduction

$(1.00*2^0 + 1.00*2^0) + (1.00*2^{-2} + 1.00*2^{-2})$

$= 1.00*2^1 + 1.00*2^{-1}$

$= 1.01*2^1$

Runtime Math Library

- There are two types of runtime math operations
  - `__func()`: direct mapping to hardware ISA
    - Fast but low accuracy
    - Examples: `__sin(x)`, `__exp(x)`, `__pow(x,y)`
  - `func()` : compile to multiple instructions
    - Slower but higher accuracy (0.5 ulp, units in the least place, or less)
    - Examples: sin(x), exp(x), pow(x,y)
- The `-use_fast_math` compiler option forces every `func()` to compile to `__func()`

Make your program float-safe!

- Double precision likely have performance cost
  - Careless use of double or undeclared types may run more slowly
- Important to be explicit whenever you want single precision to avoid using double precision where it is not needed
  - Add 'f' specifier on float literals:
    - `foo = bar * 0.123;  // double assumed`
    - `foo = bar * 0.123f;  // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);  // double assumed`
    - `foo = sinf(bar);  // single precision explicit`

讲授内容：Floating-Point Considerations

① Floating-Point Precision and Accuracy

② Numerical Stability

Numerical Stability

- Linear system solvers may require different ordering of floating-point operations for different input values in order to find a solution
- An algorithm that can always find an appropriate operation order and thus a solution to the problem is a numerically stable algorithm
  - An algorithm that falls short is numerically unstable

讲授内容

- Floating-Point Considerations
- GPU as Part of the PC Architecture
- Efficient Host-Device Data Transfer
- Computational Thinking
- Dynamic Parallelism
- CUDA libraries

Review – Typical Structure of a CUDA Program

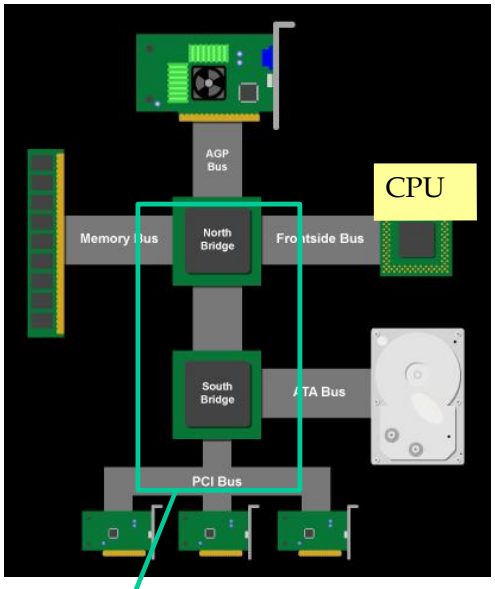
- Global variables declaration
- Function prototypes
  - `__global__ void kernelOne(…)`
- Main ()
  - allocate memory space on the device – `cudaMalloc(&d_GlblVarPtr, bytes )`
  - transfer data from host to device – `cudaMemCpy(d_GlblVarPtr, …Gl…)`
  - execution configuration setup
  - kernel call – `kernelOne<<<execution configuration>>>( args… );`
  - transfer results from device to host – `cudaMemCpy(h_GlblVarPtr,…)`
  - optional: compare against golden (host computed) solution
- Kernel – `void kernelOne(type args,…)`
  - variables declaration - `__local__, __shared__`
    - automatic variables transparently assigned to registers or local memory
  - `syncthreads()…`

Bandwidth – Gravity of Modern Computer Systems

- The bandwidth between key components ultimately dictates system performance
  - Especially true for massively parallel systems processing massive amount of data
  - Tricks like buffering, reordering and caching can temporarily defy the rules in some cases
  - Ultimately, the performance falls back to what the “speeds and feeds” dictate

History: Classic PC architecture

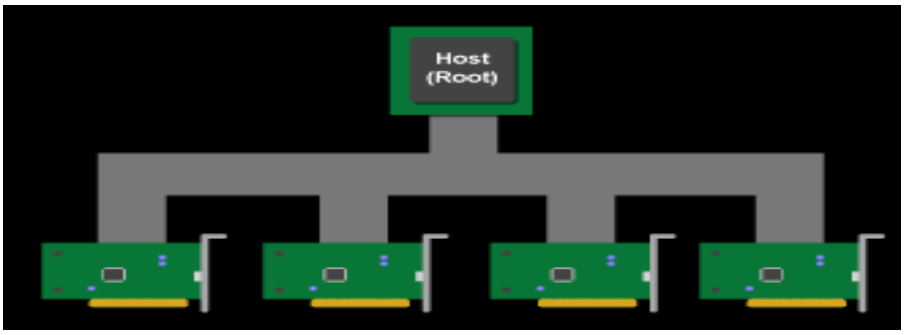
- Northbridge connects 3 components that must communicate at high speed
  - CPU, DRAM, video
  - Video also needs to have 1st-class access to DRAM
  - Previous NVIDIA cards are connected to AGP, up to 2 GB/s transfers
- Southbridge serves as a concentrator for slower I/O devices



Core Logic Chipset

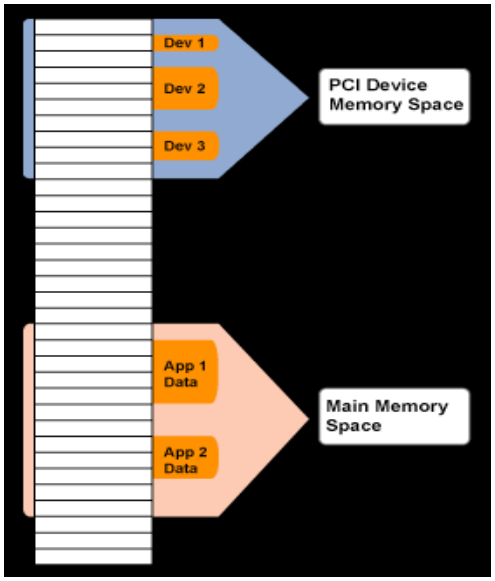
(Original) PCI Bus Specification

- Connected to the Southbridge
  - Originally 33 MHz, 32-bit wide, 132 MB/second peak transfer rate
  - More recently 66 MHz, 64-bit, 528 MB/second peak
  - Upstream bandwidth remain slow for device (~256 MB/s peak)
  - Shared bus with arbitration
    - Winner of arbitration becomes bus master and can connect to CPU or DRAM through the Southbridge and Northbridge



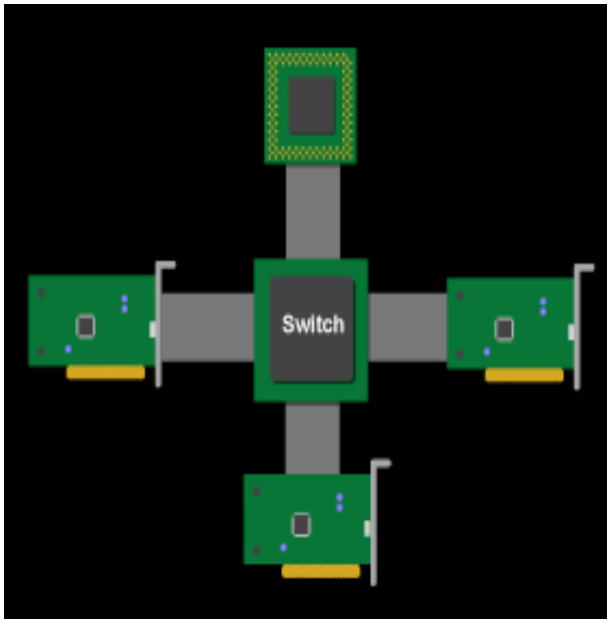
PCI as Memory Mapped I/O

- PCI device registers are mapped into the CPU's physical address space
  - Accessed through loads/ stores (kernel mode)
- Addresses are assigned to the PCI devices at boot time
  - All devices listen for their addresses



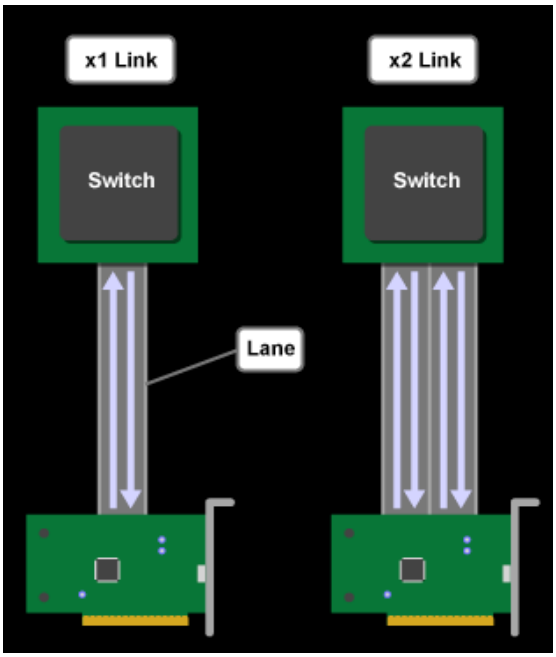
PCI Express (PCIe)

- Switched, point-to-point connection
  - Each card has a dedicated “link” to the central switch, no bus arbitration
  - Packet switches messages form virtual channel
  - Prioritized packets for QoS
    - E.g., real-time video streaming



PCIe 2 Links and Lanes

- Each link consists of one or more lanes
  - Each lane is 1-bit wide (4 wires, each 2-wire pair can transmit 2.5Gb/s in one direction)
    - Upstream and downstream now simultaneous and symmetric
  - Each Link can combine 1, 2, 4, 8, 12, 16 lanes- x1, x2, etc.
  - Each byte data is 8b/10b encoded into 10 bits with equal number of 1’s and 0’s; net data rate 2 Gb/s per lane each way
  - Thus, the net data rates are 250 MB/s (x1) 500 MB/s (x2), 1GB/s (x4), 2 GB/s (x8), 4 GB/s (x16), each way



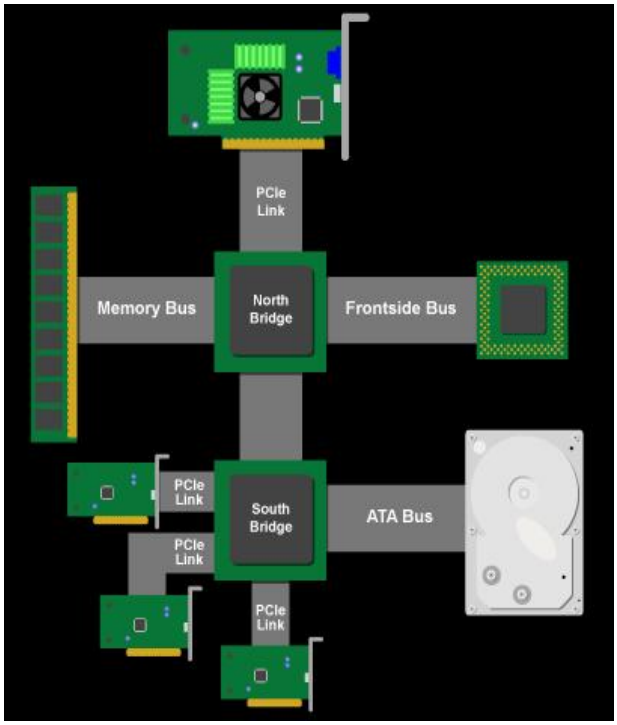
8/10 bit encoding

- Goal is to maintain DC balance while have sufficient state transition for clock recovery:
  - The difference of 1s and 0s in a 20-bit stream should be ≤ 2
  - There should be no more than 5 consecutive 1s or 0s in any stream

- 00000000, 00000111, 11000001 bad
- 01010101, 11001100 good
- Find 256 good patterns among 1024 total patterns of 10 bits to encode an 8-bit data
- 20% overhead

PCIe PC Architecture

- PCIe forms the interconnect backbone
  - Northbridge and Southbridge are both PCIe switches
  - Some Southbridge designs have built-in PCI-PCIe bridge to allow old PCI cards
  - Some PCIe I/O cards are PCI cards with a PCI-PCIe bridge
- Source: Jon Stokes, PCI Express: An Overview
  - http://arstechnica.com/articles/paedia/hardware/pcie.ars

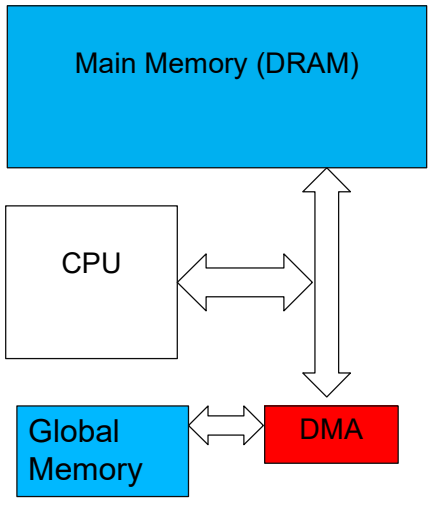


PCIe 3

- A total of 4 Giga Transfers per second in each direction
- No more 8/10 encoding but uses a polynomial transformation at the transmitter and its inverse at the receiver to achieve the same effect
- So the effective bandwidth is double of PCIe 2

PCIe Data Transfer using DMA

- DMA (Direct Memory Access) is used to fully utilize the bandwidth of an I/O bus
  - DMA uses physical address for source and destination
  - Transfers a number of bytes requested by OS
  - Needs pinned memory
  - DMA hardware is much faster than CPU software and frees the CPU for other tasks during the data transfer



Pinned Memory

- DMA uses physical addresses
- The OS could accidentally page out the data that is being read or written by a DMA and page in another virtual page into the same location
- Pinned memory cannot not be paged out

- If a source or destination of a `cudaMemCpy()` in the host memory is not pinned, it needs to be first copied to a pinned memory – extra overhead
- `cudaMemcpy` is much faster with pinned host memory source or destination

Allocate/Free Pinned Memory (a.k.a. Page Locked Memory)

- `cudaHostAlloc()`
  - Three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now
- `cudaFreeHost()`
  - One parameter
  - Pointer to the memory to be freed

Using Pinned Memory

- Use the allocated memory and its pointer the same way those returned by `malloc();`
- The only difference is that the allocated memory cannot be paged by the OS
- The `cudaMemcpy` function should be about 2X faster with pinned memory
- Pinned memory is a limited resource whose over-subscription can have serious consequences

Important Trends

- Knowing yesterday, today, and tomorrow
  - The PC world is becoming flatter
  - CPU and GPU are being fused together
  - Outsourcing of computation is becoming easier…

讲授内容

- Floating-Point Considerations
- GPU as Part of the PC Architecture
- Efficient Host-Device Data Transfer
- Computational Thinking
- Dynamic Parallelism
- CUDA libraries

讲授内容：Efficient Host-Device Data Transfer

① Pinned Host Memory

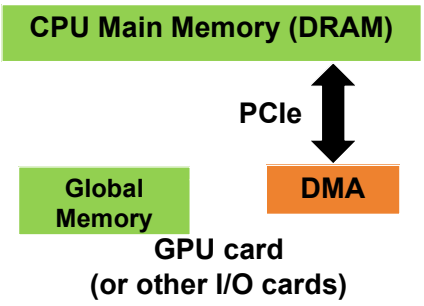
② Task Parallelism in CUDA

③ Overlapping Data Transfer with Computation

④ CUDA Unified Memory

CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for better efficiency
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe in today's systems



Virtual Memory Management

- Modern computers use virtual memory management
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time

Data Transfer and Virtual Memory

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved
- The OS could accidentally page-out the data that is being read or written by a DMA, and page-in another virtual page into the same physical location.

Pinned Memory and DMA Data Transfer

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA data transfer uses pinned memory.

- The DMA used by cudaMemcpy() requires that any source or destination in the host memory is allocated as pinned memory
- If a source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead
- `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Allocate/Free Pinned Memory

- `cudaHostAlloc()`, three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use `cudaHostAllocDefault` for now
- `cudaFreeHost()`, one parameter
  - Pointer to the memory to be freed

Using Pinned Memory in CUDA

- Use the allocated pinned memory and its pointer the same way as those returned by `malloc();`
- The only difference is that the allocated memory cannot be paged by the OS
- The `cudaMemcpy()` function should be about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

Putting It Together - Vector Addition Host Code Example

```
int main()
{
    float *h_A, *h_B, *h_C;
…
    cudaHostAlloc((void **) &h_A, N* sizeof(float),
        cudaHostAllocDefault);
    cudaHostAlloc((void **) &h_B, N* sizeof(float),
        cudaHostAllocDefault);
    cudaHostAlloc((void **) &h_C, N* sizeof(float),
        cudaHostAllocDefault);
…
    // cudaMemcpy() runs 2X faster
}
```

讲授内容：Efficient Host-Device Data Transfer

① Pinned Host Memory

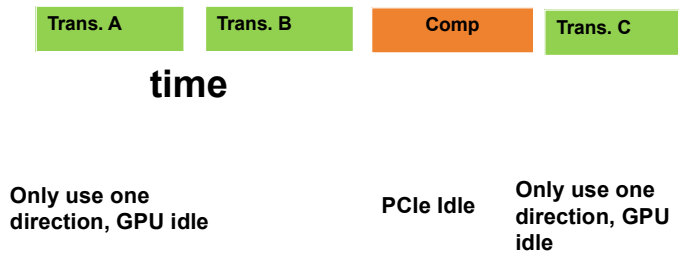
② Task Parallelism in CUDA

③ Overlapping Data Transfer with Computation

④ CUDA Unified Memory

Serialized Data Transfer and Computation

- So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation for `VecAddKernel()`



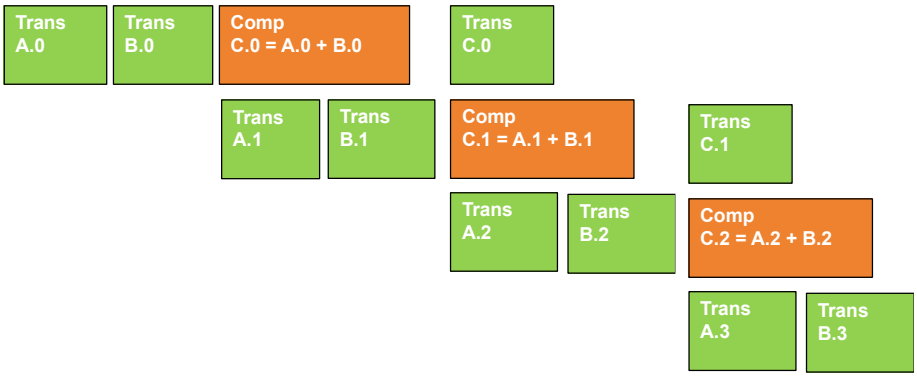
Device Overlap

- Some CUDA devices support device overlap
  - Simultaneously execute a kernel while copying data between device and host memory

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&prop, i);
  if (prop.deviceOverlap) …
```


Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

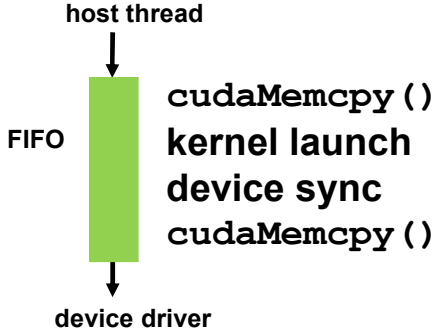


CUDA Streams

- CUDA supports parallel execution of kernels and `cudaMemcpy()` with "Streams"
- Each stream is a queue of operations (kernel launches and `cudaMemcpy()` calls)
- Operations (tasks) in different streams can go in parallel
  - "Task parallelism"

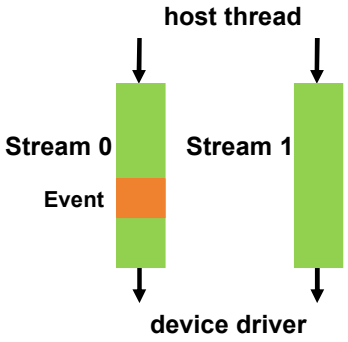
Streams

- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.

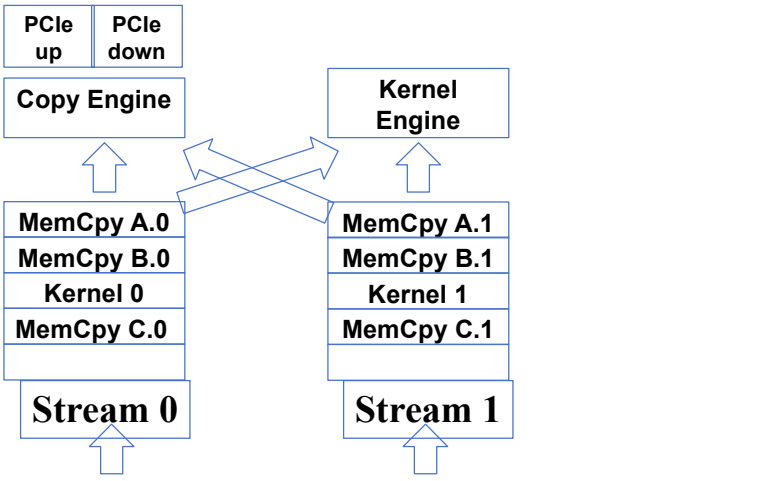


Streams cont.

- To allow concurrent copying and kernel execution, use multiple queues, called "streams"
  - CUDA "events" allow the host thread to query and synchronize with individual queues (i.e. streams).



Conceptual View of Streams



Operations (Kernel launches, `cudaMemcpy()` calls)

讲授内容：Efficient Host-Device Data Transfer

① Pinned Host Memory

② Task Parallelism in CUDA

③ Overlapping Data Transfer with Computation

④ CUDA Unified Memory

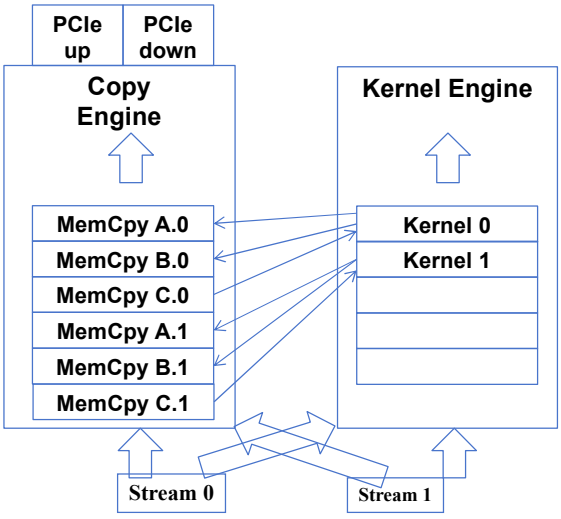
Simple Multi-Stream Host Code

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
```

Simple Multi-Stream Host Code (Cont.)

```
for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0,…);
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…,
stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…,
  stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),…,
  stream1);
}
```

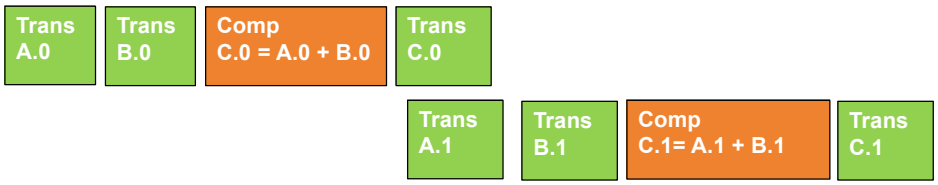

A View Closer to Reality in Previous GPUs



Operations (Kernel launches, `cudaMemcpy()` calls)

Not quite the overlap we want in some GPUs

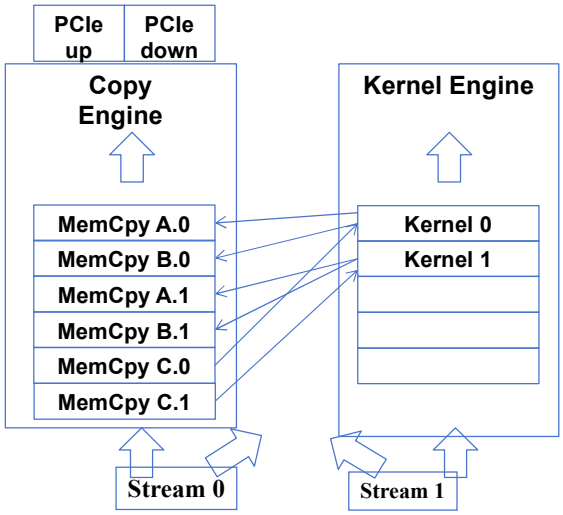
– C.0 blocks A.1 and B.1 in the copy engine queue



Better Multi-Stream Host Code

```
for (int i=0; i<n; i+=SegSize*2) {
  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…,
stream1);
  cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…,
stream1);

  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
  vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);

  cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),…,
stream1);
}
```

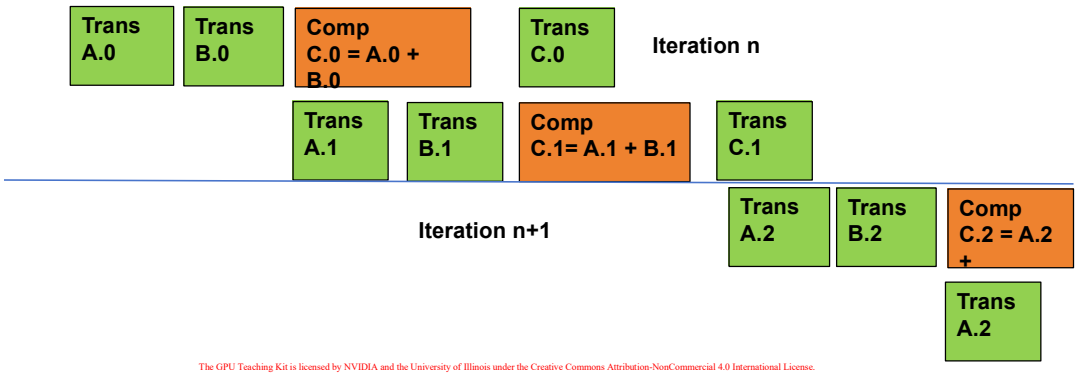
C.0 no longer blocks A.1 and B.1



Operations (Kernel launches, cudaMemcpy() calls)

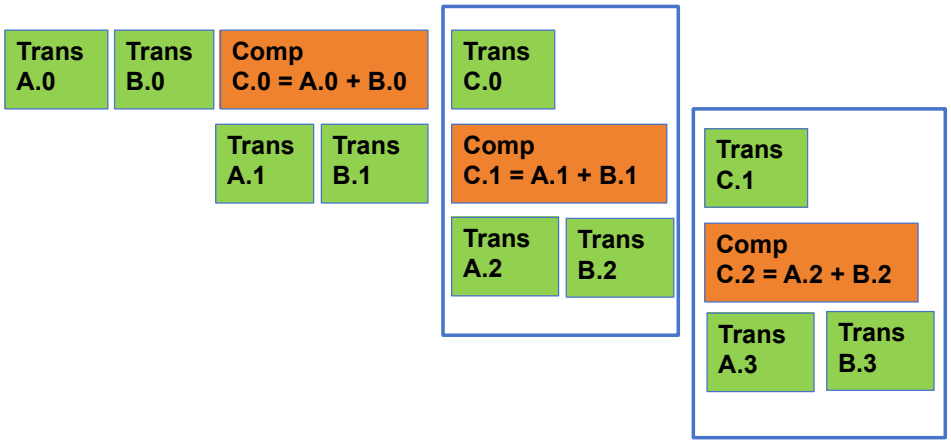
Better, not quite the best overlap

- C.1 blocks next iteration A.0 and B.0 in the copy engine queue



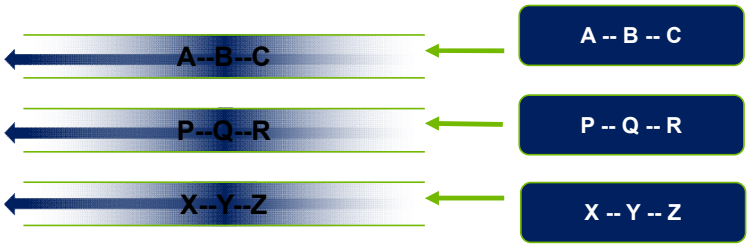
Ideal, Pipelined Timing

- Will need at least three buffers for each original A, B, and C, code is more complicated



Hyper Queues

- Provide multiple queues for each engine
- Allow more concurrency by allowing some streams to make progress for an engine while others are blocked



Wait until all tasks have completed

- `cudaStreamSynchronize(stream_id)`
  - Used in host code
  - Takes one parameter – stream identifier
  - Wait until all tasks in a stream have completed
  - `E.g., cudaStreamSynchronize(stream0)` in host code ensures that all tasks in the queues of stream0 have completed
- This is different from `cudaDeviceSynchronize()`
  - Also used in host code
  - No parameter
  - `cudaDeviceSynchronize()` waits until all tasks in all streams have completed for the current device

讲授内容：Efficient Host-Device Data Transfer

① Pinned Host Memory

② Task Parallelism in CUDA

③ Overlapping Data Transfer with Computation

④ CUDA Unified Memory

Data Prefetching in CUDA Unified Memory

- Main purpose being to avoid page faults
- Establishes data locality, providing a mechanism to improve the performance of the application
- Used to migrate data to a device or the host and map page tables onto the processor before it begins using data
- Most useful when the data is accessed from a single processor

Data Prefetching CUDA API

- cudaMemPrefetchAsync()
  - Four parameters
    - Unified Memory Address of the memory region to prefetch
    - Size of the region to prefetch in terms of bytes
    - Destination processor
    - CUDA stream

- Prefetching is an asynchronous operation with respect to the device, however the call may not be fully asynchronous with respect to the host .
- The destination processor must be a valid device ID or *cudaCpuDeviceId,* the later will prefetch the memory to the host.
- If the prefetch processor is a GPU, the device property cudaDevAttrConcurrentManagedAccess must be nonzero.

Putting it all together

```
void function(int * data, cudaStream_t stream) {
    // data must've been allocated with
cudaMallocManaged( (void**) &data, N);
    init(data,
N);
                                          // Init data on the host
    cudaMemPrefetchAsync(data, N * sizeof(int), myGpuId,
stream);                  // Prefetch to the device
    kernel<<<...,
stream>>>(dat
a, ...);
         // Execute on the device
    cudaMemPrefetchAsync(data, N * sizeof(int), cudaCpuDeviceId,
stream);  // Prefetch to the host
    cudaStreamSynchronize(stream);
    hostFunction(data, N);
}
```


CUDA Unified Memory - Memory Advisor

- Hints that can be provided to the driver on how data will be used during runtime

- One example of when we can give an advice is when data will be accessed from multiple processors at the same time

Memory Advisor CUDA API

- cudaMemAdvise()
  - Four parameters
    - Unified Memory Address of the memory region to advise
    - Size of the region to advise in terms of bytes
    - Memory advise
    - Destination processor

- The destination processor must be a valid device ID or *cudaCpuDeviceId.*
- Available memory hints are:
  - cudaMemAdviseSetReadMostly
  - cudaMemAdviseSetPreferredLocation
  - cudaMemAdviseSetAccessedBy

  To unset the advice:
  - cudaMemAdviseUnsetReadMostly
  - cudaMemAdviseUnsetPreferredLocation
  - cudaMemAdviseUnsetAccessedBy

Memory Advisor CUDA API

- cudaMemAdviseSetReadMostly:
  - This tells the UM driver that the memory region is mostly for reading purposes and occasionally writing.
  - All read accesses from a processor will create a read only copy to be accessed.
  - The device argument is ignored.
  - When used in conjunction with *cudaMemPrefetchAsync()* a read only copy will be created in the specified device.
- cudaMemAdviseSetPreferredLocation:
  - This tells the UM driver the preferred location of the data.
  - However this does not cause immediate data migration to the location, it only guides the migration policy for the UM driver.
  - If the destination processor is a GPU, then that GPU needs a nonzero value for the device flag *cudaDevAttrConcurrentManagedAccess*.

Memory Advisor CUDA API

- cudaMemAdviseSetAccessedBy:
  - This tell the UM driver that the data will be accessed by the specified processor.
  - This advice does not cause immediate data migration to the location, it only causes data to always be mapped in the specified processor's pages.
  - It is useful to avoid page faulting, like when in a multi-GPU system one device wants to access another GPU's memory but migrating the data may be more expensive than just reading it through the PCIE link.

Putting it all together

```
void function(int * data,
cudaStream_t stream) {

    // data must be addressable by
the Unified Memory driver.
    init(data, N);        // Init
data on the host

    cudaMemAdvise(data, N *
sizeof(int),
cudaMemAdviseSetReadMostly , 0 );
// Set the advice for read only.
    cudaMemPrefetchAsync(data, N *
sizeof(int), myGpuId1,
stream1);                 //
Prefetch to the 1st device.
    cudaMemPrefetchAsync(data, N *
sizeof(int), myGpuId2,
stream2);                 //
// Prefetch to the 2nd device.

    cudaSetDevice(myGpuId1);
// Execute read only operations
    kernel<<<...,
stream>>>(data, ...);
// on the 1st device.

    cudaSetDevice(myGpuId2);
                              //
Execute read only operations
    kernel<<<...,
stream>>>(data, ...);
// on the 2nd device.
}
```

讲授内容

- Floating-Point Considerations
- GPU as Part of the PC Architecture
- Efficient Host-Device Data Transfer
- Computational Thinking
- Dynamic Parallelism
- CUDA libraries

Fundamentals of Parallel Computing

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time (strong scaling), and/or
  - To solve bigger problems (weak scaling), and/or
  - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Shared Memory vs. Message Passing

- We have focused on shared memory parallel programming
  - This is what CUDA (and OpenMP, OpenCL) is based on
  - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
  - However, you will find parallels for almost every technique you learned in this course
  - Need to be aware of space-time constraints

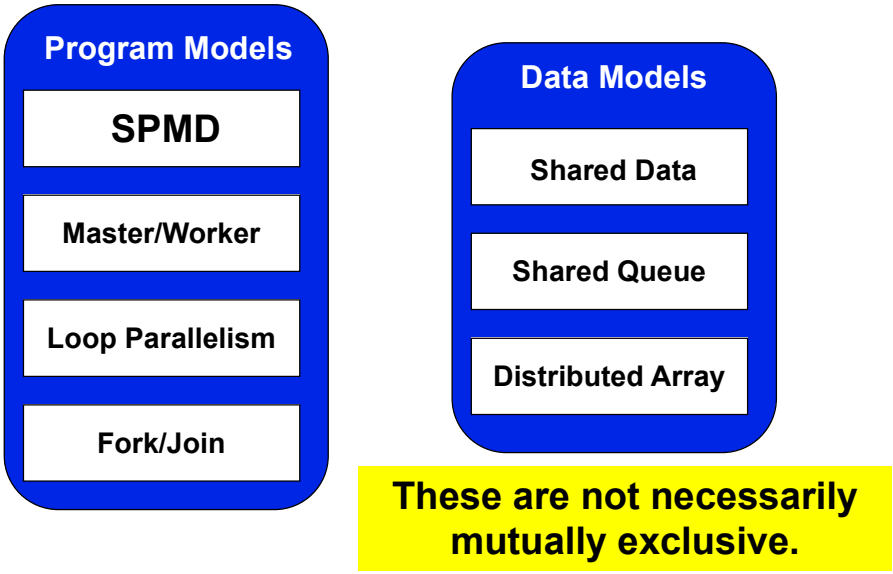
Data Sharing

- Data sharing can be a double-edged sword
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- Many:Many, One:Many, Many:One, One:One

Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
  - Watch out for serialization
- Important: be aware of which items of work are truly independent

Parallel Programming Coding Styles – Program and Data Models



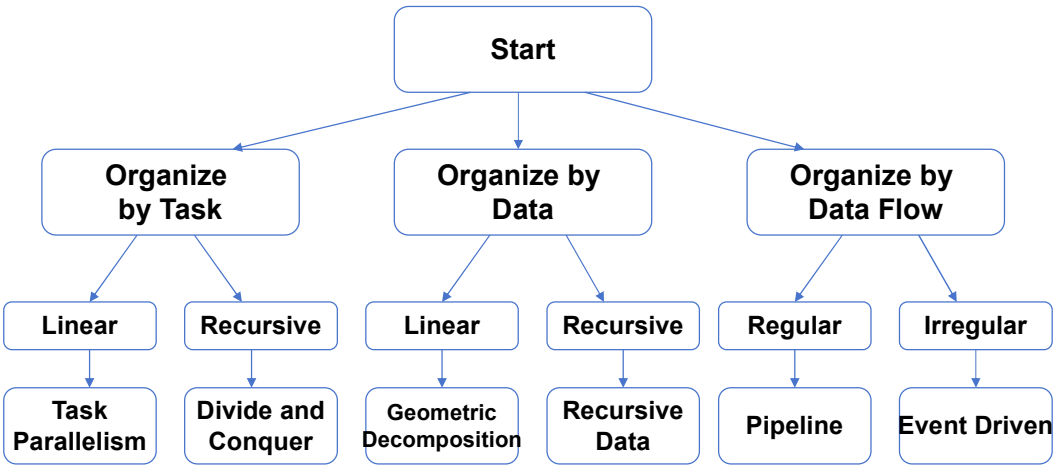
Program Models

- SPMD (Single Program, Multiple Data)
  - All PE's (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code
  - This is essentially the CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
  - Most general, generic way of creation of threads

Algorithm Structure



More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD (next to loop parallelism)
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - Use thread ID in loop index calculations to split loop iterations among threads
    - Potential for memory/data divergence
  - Use thread ID or conditions based on thread ID to branch to their specific actions
    - Potential for instruction/execution divergence

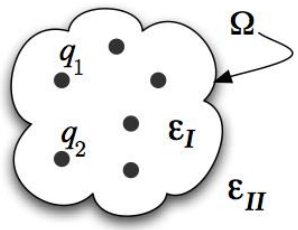
Both can have very different performance results and code complexity depending on the way they are done.

Making Science Better, not just Faster

or… in other words:
There will be no Nobel Prizes or Turing Awards awarded for "just recompile" or using more threads

As in Many Computation-hungry Applications

- **Three-step approach:**
  - **Restructure the mathematical formulation**
  - **Innovate at the algorithm level**
  - **Tune core software for hardware architecture**



Conclusion: Three Options

- Good: "Accelerate" Legacy Codes
  - Recompile/Run
  - Call CUBLAS/CUFFT/thrust/matlab/PGI pragmas/etc.
  - => good work for domain scientists (minimal CS required)
- Better: Rewrite / Create new codes
  - Opportunity for clever algorithmic thinking
  - => good work for computer scientists (minimal domain knowledge required)
- Best: Rethink Numerical Methods & Algorithms
  - Potential for biggest performance advantage
  - => Interdisciplinary: requires CS and domain insight
  - => Exciting time to be a computational scientist

Think, Understand... then, Program

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
  - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part ☺)

Future Studies

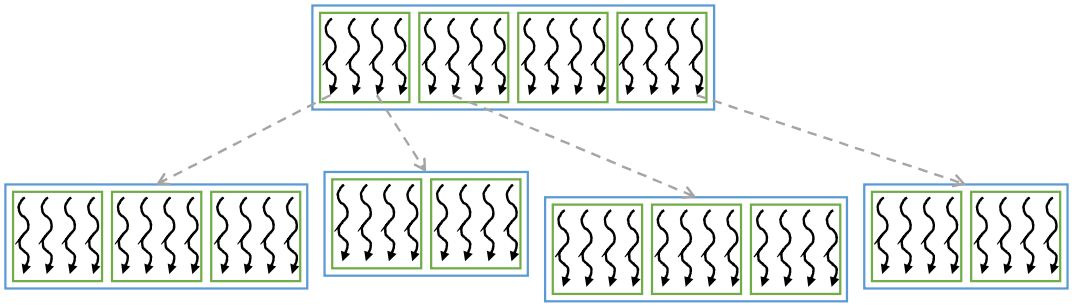
- More complex data structures
- More scalable algorithms and building blocks
- More scalable math models
- Thread-aware approaches
  - More available parallelism
- Locality-aware approaches
  - Computing is becoming bigger, and everything is further away

讲授内容

- Floating-Point Considerations
- GPU as Part of the PC Architecture
- Efficient Host-Device Data Transfer
- Computational Thinking
- Dynamic Parallelism
- CUDA libraries

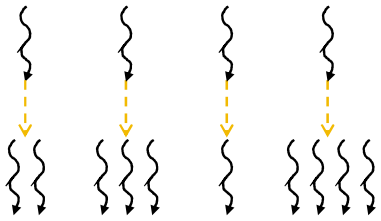
Dynamic Parallelism

- CUDA dynamic parallelism refers to the ability of threads executing on the GPU to launch new grids



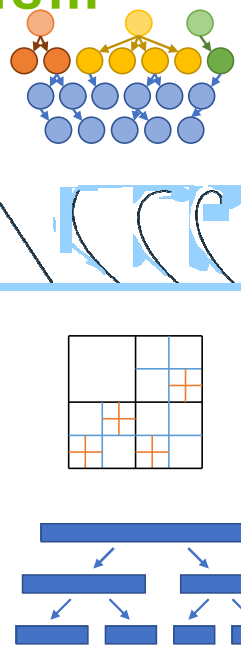
Nested Parallelism

- Dynamic parallelism is useful for programming applications with nested parallelism where each thread discovers more work that can be parallelized
- Dynamic parallelism is particularly useful when the amount of nested work is dynamically determined at execution time, so enough threads cannot be launched up front



Applications of Dynamic Parallelism

- Applications whose amount of nested work may be unknown before execution time:
  - Nested parallel work is irregular (varies across threads)
    - e.g., graph algorithms (each vertex has a different #neighbors)
    - e.g., Bézier curves (each curve needs different #points to draw)
  - Nested parallel work is recursive with data-depdent depth
    - e.g., tree traversal algorithms (e.g., quadtrees and octrees)
    - e.g., divide and conquer algorithms (e.g., quicksort)



讲授内容

- Floating-Point Considerations
- GPU as Part of the PC Architecture
- Efficient Host-Device Data Transfer
- Computational Thinking
- Dynamic Parallelism
- CUDA libraries

讲授内容：CUDA libraries

① cuBLAS

② cuSOLVER

③ cuFFT

④ Thrust

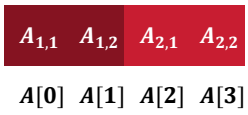
BLAS and cuBLAS

- BLAS (Basic Linear Algebra Subprograms) is a low-level linear algebra library originally written in Fortran and standardized by the BLAS Technical Forum
- It provides three levels of routines:
  - Level 1: Scalar and Vector-Vector operations
    - Example: Dot product and SAXPY
  - Level 2: Matrix-Vector operations
    - Example: Matrix vector multiplication, solving a triangular system
  - Level 3: Matrix-Matrix operations
    - Example: GEMM
- cuBLAS is a BLAS implementation for CUDA devices

Data layout in cuBLAS

- There are two natural data layouts to linearly store dense matrices:
  - Row major order:
    - Each row is stored contiguously
    - Used by C, C++ and derivatives
  - Column mayor order:
    - Each column is stored contiguously
    - Used by Fortran and derivatives
- cuBLAS uses column major order with 1 indexing for compatibility with Fortran numeric libraries

Matrix A

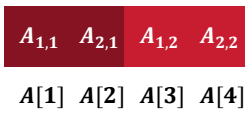


Row major order with 0 indexing

$$A_{i,j} = A[i * cols + j]$$

Addressing in row major order

Matrix A



Column major order with 1 indexing

$$A_{i,j} = A[j * rows + i]$$

Addressing in column major order

Creation and destruction of a cuBLAS environment

- cuBLAS needs an execution context to store internal resources. This context needs to be created before executing any cuBLAS routine
- After all cuBLAS executions are finished the context needs to be destroyed to free resources
- Creating and destroying contexts should be considered an expensive operation. Recommended that each thread and each device have its own context
- To create a context in a specific device call cudaSetDevice before the creation

- cublasHandle_t
  - Type used by cuBLAS to store contexts
- cublasCreate(cublasHandle_t* handle)
  - Creates a cuBLAS context
  - Parameters:
    - Pointer to cuBLAS handle to create
- cublasDestroy(cublasHandle_t handle)
  - Destroys a cuBLAS context
  - Parameters:
    - cuBLAS handle with the context to destroy
- cublasStatus_t
  - Type used by cuBLAS for reporting errors
  - Every cuBLAS returns an error status

cuBLAS streams API and thread safety

- cuBLAS permits the use of cuda streams (cudaStream_t) for increasing resource usage and introduce other levels of parallelism
- cuBLAS is a thread safe library, meaning that the cuBLAS host functions can be called from multiple threads safely

- cublasSetStream():
  - Sets the stream to be used by cuBLAS for subsequent computations
  - Parameters:
    - cuBLAS handle to set the stream
    - cuda stream to use
- cublasGetStream():
  - Gets the stream being used by cuBLAS
  - Parameters:
    - cuBLAS handle to get the stream
    - pointer to cuda stream

cuBLAS API naming convention for BLAS routines

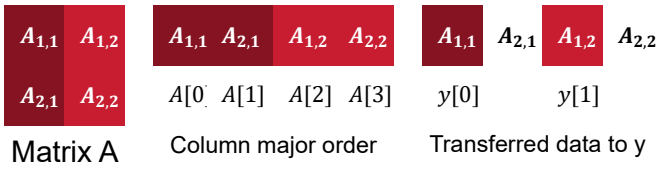
- Each of the three levels of BLAS routines in cuBLAS have multiple interfaces for the same operation, having the naming convention:
- cublas<t>operation where <t> is one of:
  - S for float parameters
  - D for double parameters
  - C for complex float parameters
  - Z for complex double parameters
- Example: For the axpy operation ($\mathrm{y}[\mathrm{i}] = \alpha x[i] + y[i]$), the available functions are:
  - cublasSaxpy, cublasDaxpy, cublasCaxpy, cublasZaxpy

cuBLAS memory API

- cuBLAS offers specialized data migration and copy functions for strided matrix and vector transfers
- Available functions:
  - cublasGetVector & cudaGetMatrix for device to host transfers
  - cublasSetVector & cudaSetMatrix for host to device transfers
  - cublas<t>copy for device to device transfers
- Useful for obtaining a row in a matrix with column major order

cuBLAS memory API

- cublasGetVector():
  - Parameters:
    - Number of elements to transfer in bytes
    - Element size in bytes
    - Source device pointer
    - Stride to use for the source vector
    - Destination host pointer
    - Stride to use for the destination vector
- Example
- cublasGetVector(2*sizeof(float), sizeof(float), A, 2, y, 1);



cuBLAS example: Conjugate Gradient

- The Conjugate Gradient method is an iterative method to compute an approximation for the solution of the linear algebraic system $Ax = b$
- Assumptions:
  - Let A be a $n * n$ symmetric and positive definite matrix (all the eigenvalues are positive)
  - Let b be a n-dimensional vector

Algorithm:

1. $r_0 = b - A * x_0$
2. $p_0 = r_0$
3. $k = 0$
4. loop
   1. $\alpha_k = \frac{r_k^T * r_k}{p_k^T * A * p_k}$
   2. $x_{k+1} = x_k + \alpha_k p_k$
   3. $r_{k+1} = r_k - \alpha_k A * p_k$
   4. if $\|r_{k+1}\|_2 < \varepsilon$, break the loop
   5. $\beta_k = \frac{r_{k+1}^T * r_{k+1}}{r_k^T * r_k}$
   6. $p_{k+1} = r_{k+1} + \beta_k p_k$
   7. $k = k + 1$
5. return x_{k+1}

cuBLAS implementation of the Conjugate Gradient method

```
1. double zero = 0, minusOne = -1, one = 1, alpha = 0, beta = 0, rxr = 0, tmp;
2. cublasDcopy(handle, n, b, 1, r, 1);
   // r_0 = b
3. cublasDgemv(handle, CUBLAS_OP_N, n, n, &minusOne, A, rows, x, 1, &one, r, 1);
   // r_0 = b - A * x_0
4. cublasDcopy(handle, n, r, 1, p, 1);
   // p_0 = r_0
5. cublasDdot(handle, n, r, 1, r, 1, &rxr);
   // r_k^T * r_k
6. while(k < maxit) {
7.   cublasDgemv(handle, CUBLAS_OP_N, n, n, &minusOne, A, rows, p, 1, &zero, Axp, 1);
   // A * p_k
8.   cublasDdot(handle, n, p, 1, Axp, 1, &tmp);
   // p_k^T * A * p_k
9.   alpha = rxr / tmp;
10.  cublasDaxpy(handle, n, &alpha, p, 1, x, 1);
   // x_{k+1} = x_k + alpha_k p_k
```

cuBLAS implementation of the Conjugate Gradient method

```
1.  tmp = -alpha;
2.  cublasDaxpy(handle, n, &tmp, Axp, 1, r, 1);
    // r_{k+1} = r_k − α_k A * p_k
3.  cublasDdot(handle, n, r, 1, r, 1, &tmp);
    // r_k^T * r_k
4.  if (sqrt(tmp) < epsilon) break;
5.  beta = tmp / rxr;
6.  rxr = tmp;
7.  cublasDscal(handle, n, beta, p, 1);
    // β_k p_k
8.  cublasDaxpy(handle, n, &one, r, 1, p, 1);
    // p_{k+1} = r_{k+1} + β_k p_k
9.  k += k;
10.}
```


讲授内容：CUDA libraries

① cuBLAS

② cuSOLVER

③ cuFFT

④ Thrust

cuSOLVER

- cuSOLVER is linear algebra library with LAPACK-like features focused on:
  - Direct factorization methods for dense matrices
  - Linear systems solving methods for dense and sparse matrices
  - Eigenvalue problems for dense and sparse matrices
  - Refactorization techniques for sparse matrices
  - Least square problems for sparse matrices
- In cases where the sparsity pattern produces low GPU utilization cuSOLVER provides CPU routines to handle those sparse matrices

cuSOLVER dense API I

- cuSOLVER dense matrices routines are available under the cuSolverDN API
- cuSolverDN provides two different APIs:
  - Legacy naming convention:
    - cusolverDn<T><operation>
  - Generic naming convention:
    - cusolverDn<operation>

- <T> is the data type used by the matrices:

<T>	Type
S	float
D	double
C	cuComplex
Z	cuDoubleComplex
X	Generic type

- <operation> is the operation to be performed, some examples are:
  - portf for Cholesky factorization
  - gesvd for the SVD decomposition
- Example:
  - cusolverDnDportf calls the routine performing the Cholesky decomposition for matrices of type double

cuSolverDN dense API II

- cuSolverDN assumes that matrices are stored in column-major format
- The cuSolverDN generic API provides 64bit support for integer parameters
- The cuSolverDN generic API uses the type cudaDataType in the routines arguments to specify the type of the input matrix

cudaDataType	Type
CUDA_R_32F	Single precision real number
CUDA_R_64F	Double precision real number
CUDA_C_32F	Single precision complex number
CUDA_C_64F	Double precision complex number

cuSOLVER dense environment

- cuSolverDN needs an execution context to store internal resources. This context needs to be created before executing any cuSolverDN routine
- After all cuSolverDN executions are finished the context needs to be destroyed to free resources
- To create a context in a specific device call cudaSetDevice before the creation

- cusolverDnHandle_t
  - Type used by cuSolverDN to store contexts handles
- cusolverDnCreate(cusolverDnHandle_t* handle)
  - Creates a cuSolverDN context
  - Parameters:
    - Pointer to cuSolverDN handle to create
- cusolverDnDestroy(cusolverDnHandle_t handle)
  - Destroys a cuSolverDN context
  - Parameters:
    - cuSolverDN handle with the context to destroy
- cusolverStatus_t
  - Type used by cuSOLVER for reporting errors

cuSOLVER dense environment

- cuSolverDN routines don't allocate workspace memory by themselves, the user needs to allocate the device workspace
- To find the size in bytes needed by a cuSolverDN routine, the user must call:
  - <routine>_bufferSize()
    - The arguments of the function varies according to the routine
    - <routine> is the name of the cuSolverDN routine
- Once the user allocates the workspace region the user can call the routine
- cuSolverDN routines accept an info parameter, if info is less than 0 this tells the user that the i-th parameter (not counting the handle) is invalid

cuSolverDN streams API and thread safety

- cuSolverDN permits the use of cuda streams (cudaStream_t) for increasing resource usage and introduce other levels of parallelism
- cuSolverDN is a thread safe library, meaning that the cuSolverDN host functions can be called from multiple threads safely

- cusolverDnSetStream():
  - Sets the stream to be used by cuSolverDN for subsequent computations
  - Parameters:
    - cuSolverDN handle to set the stream
    - cuda stream to use
- cusolverDnGetStream():
  - Gets the stream being used by cuSolverDN
  - Parameters:
    - cuSolverDN handle to get the stream
    - pointer to cuda stream

cuSolverDN LU factorization with generic API

- Given a matrix A, the LU factorization is given by:

$$P * A = L * U$$

where P is a permutation matrix produced by the algorithm with the row pivots. L is a lower triangular matrix with unit diagonal and U is an upper triangular matrix.

- The matrix A is assumed to be in column-major order
- If info=i and is positive, it means the factorization failed and U(i,i) = 0

cuSolverDN LU factorization with generic API

```
cusolverDnHandle_t handle;     // cuSolverDN handle
cusolverDnParams_t params; // Routine options
void* A;                              // Matrix to factorize
int64_t m, n;                       // Rows and columns respectively
int64_t* ipiv;                       // Row pivots, vector of m elements
size_t workSize;                // Workspace size in bytes
void* buffer;                       // Workspace buffer
cudaDataType typeA;    // Type of matrix A
int info = 0;                        // Info parameter
```

```
// Create and initialize options struct:
cusolverDnCreateParams(&params);
cusolverDnSetAdvOptions(params, CUSOLVERDN_GETRF, CUSOLVER_ALG_0);
// Get the buffer size and allocate it:
cusolverDnGetrf_bufferSize(handle, params, m, n, typeA, A, n, typeA, &workSize);
cudaMalloc((void**) &buffer, workSize);
// Compute the factorization
cusolverDnGetrf(handle, params, m, n, typeA, A, n, ipiv, typeA, buffer, workSize, &info);
// Destroy the options struct
cusolverDnDestroyParams(params);
```

cuSolverDN Cholesky factorization with legacy API

- Given a matrix A, the Cholesky factorization is given by either of:

$$A = L * L^H$$

$$A = U^H * U$$

Where L is a lower triangular matrix and U is an upper triangular matrix.

- cuSolverDN uses a fill mode parameter to determine which decomposition to compute
- If info=i and is positive, it means the factorization failed and U(i,i) = 0

```
cusolverDnHandle_t handle;     // cuSolverDN handle
double* A;                     // Matrix to factorize
cublasFillMode_t uplo;   // Type indicating filling mode
int n;                         // Number of rows
and columns
int workSize;                  // Workspace size in
bytes
void* buffer;                  // Workspace buffer
int info = 0;                  // Info parameter
// Get the buffer size and allocate it:
cusolverDnSpotrf_bufferSize(handle, uplo, n, A, n,
&workSize);
cudaMalloc((void**) &buffer, workSize);
// Compute the factorization
cusolverDnDpotrf(handle, uplo n, A, n, buffer, workSize,
&info);
```


List of some available routines in cuSolverDN

- **Factorization:**
  - potrf, potrs for Cholesky factorization and linear system solving, see potrfBatched and potrsBatched for batched versions
  - getrf, getrs for LU factorization and linear system solving
  - geqrf for QR factorization
  - sytrf for LDL factorization
- **Eigenvalue problems:**
  - gesvd for SVD decomposition
  - syevd for Eigenvalue decomposition
  - sygvd for general Eigenvalue decomposition

Compiling and linking against cuSOLVER

- In order to compile and link against cuSOLVER, the user needs to:
  - Include the appropriate header in the required files
    - #include <cusolverDn.h> for cuSOLVER dense functionality
    - #include <cusolverSp.h> for cuSOLVER sparse functionality
    - #include <cusolverRf.h> for cuSOLVER refactorization functionality
  - Link against the cuSOLVER library:
    - For dynamic linking use the flag -lcusolver
    - For static linking use the flags -lcusolver_static -llapack_static

讲授内容：CUDA libraries

① cuBLAS

② cuSOLVER

③ cuFFT

④ Thrust

Discrete Fourier Transform (DFT)

- Widely used in signal processing from astronomical radio signals to image processing and many other areas
- The forward Discrete Fourier Transform is given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\omega kn}$$

where $(x_0, \cdots, x_{N-1})$ is the input signal, $(X_0, \cdots, X_{N-1})$ the transformed signal, $\omega = \frac{2\pi}{N}$ and i the imaginary unit

- Given the transformed signal it's possible to recover the input signal, this process is known as the inverse transform

Fast Fourier Transform (FFT) & cuFFT

- A fast method to compute the DFT of a signal, with computational complexity of $O(N\log_2 N)$ whereas the DFT has $O(N^2)$ complexity
- A cornerstone of numeric algorithms for its many applications and speed
- cuFFT is CUDA implementation of several FFT algorithms
- cuFFT is especially fast for input sizes of $2^a 3^b 5^c 7^d$ elements
- cuFFT provides 1D, 2D and 3D transforms, as well as:
  - C2C complex input to complex output
  - R2C real input to complex output
  - C2R complex input to real output
- cuFFT provides Multi-GPU support trough the cuFFT Xt API

cuFFT plans & environment

- Plans contain necessary information to compute a transform, such as:
  - The best algorithm to use for the specified size
  - Memory resources needed by cuFFT to compute the transform
  - Plans need to be created before executing the transform and destroyed after the user is done using them
- There exists 2 mains way to create plans:
  - Through the basic plan API:
    - cufftPlan1D, cufftPlan2D, cufftPlan3D, cufftPlanMany
    - Aimed for easy setup
  - Through the extensible plan API:
    - cufftCreate, cufftMakePlan1D, cufftMakePlan2D, cufftMakePlan3D, cufftMakePlanMany, cufftMakePlanMany64
    - Aimed for customization and extensibility

cuFFT numeric data types

- cuFFT offers the following numeric types:
  - cufftReal for single precision real numbers
  - cufftDouble for double precision real numbers
  - cufftComplex for single precision complex numbers
  - cufftDoubleComplex for double precision complex numbers
- For simplicity, the rest of the slides will use real and complex for referring to cufftReal/cufftDouble and cufftComplex/cufftDoubleComplex respectively
- For memory size considerations is important to note that:

$$sizeof(complex) = 2 * sizeof(real)$$

Data layout for 1D out-of-place transforms

- Out of place transforms have different memory regions for the input and output signal
- Input and output sizes for each transform:

Type	Input size in bytes	Output size in bytes
C2C	$N * sizeof(complex)$	$N * sizeof(complex)$
C2R	$(\lfloor N/2 \rfloor + 1) * sizeof(complex)$	$N * sizeof(real)$
R2C	$N * sizeof(real)$	$(\lfloor N/2 \rfloor + 1) * sizeof(complex)$

- Custom strides in the layout causes cuFFT to overwrite the input array in the C2R transform
- The size of the output array in R2C consists of $\lfloor N/2 \rfloor + 1$ complex numbers due to "Hermitan" redundancy property of the transform

Data layout for 1D in-place transforms

- In place transforms stores the output result in the input array
- Array sizes for each transform:

Type	Array size in bytes
C2C	$N * sizeof(complex)$
C2R	$(\lfloor N/2 \rfloor + 1) * sizeof(complex)$
R2C	$(\lfloor N/2 \rfloor + 1) * sizeof(complex)$

- The sizes of C2R and R2C are due to the transform needs to hold at most $(\lfloor N/2 \rfloor + 1)$ complex numbers
- In R2C only the first N real entries are filled with input data, the rest of the array is allocated for the output

cuFFT extensible API

- Provided for customization and extensibility compared to the basic API
- Typical usage of the extensible API:
  - Initialize cufftHandle
  - Make a plan
  - Compute transforms
  - Destroy the plan in cufftHandle
- Every function returns an error status with type cufftResult

- cufftHandle
  - Type used by cuFFT to store plans
- cufftCreate(cufftHandle* handle)
  - Creates a cuFFT handle
  - Parameters:
    - Pointer to cuFFT handle to create
- cufftDestroy(cufftHandle handle)
  - Destroys resources of a cuFFT plan
  - Parameters:
    - cuFFT handle with the context to destroy
- cufftResult
  - Type used by cuFFT for reporting errors
  - Every cuFFT returns an error status

cuFFT workspace memory

- cuFFT by default auto allocates on plan creation the workspace memory needed for computations
- cuFFT allows the user to provide the memory region to be used for the workspace
- To disable auto allocation, the user needs to call cufftSetAutoAllocation() after cufftCreate() and before cufftMakeplan*()

- cufftSetAutoAllocation():
  - Sets if cuFFT should use auto workspace allocation
  - Parameters:
    - cufftHandle plan handle
    - integer indicating whether to allocate workspace area, 0:false & 1:true

cuFFT workspace memory

- If auto allocation has been disabled, the user needs to:
  - Get the size of the workspace area needed by cuFFT
  - Allocate device memory with the size specified by cuFFT
  - Set the memory region to be used by the plan
- Allocating and setting the workspace memory needs to be done after plan creation and before plan execution
- cuFFT provides a simple configuration mechanism called a plan that uses internal building blocks to optimize the transform for the given configuration and the particular GPU hardware selected.

- **cufftGetSize():**
  - Gets the size in bytes needed by the cuFFT plan
  - Parameters:
    - **cufftHandle** plan handle
    - pointer to size_t variable to store the size
- **cufftSetWorkArea():**
  - Sets the memory region to be used by the cuFFT plan
  - Parameters:
    - **cufftHandle** plan handle
    - Pointer to device memory region

Creating cuFFT plans

- cuFFT allows 1, 2 and 3 dimensional transforms
- Available plaining routines:
  - **cufftMakePlan1d:** for 1d transforms
  - **cufftMakePlan2d:** for 2d transforms
  - **cufftMakePlan3d:** for 3d transforms
  - **cufftMakePlanMany:** for 1, 2 and 3D transforms with support for strided input and output memory layouts
  - **cufftMakePlanMany64:** same as **cufftMakePlanMany** but with 64-bit support for sizes and strides

- **cufftMakePlan1d():**
  - Creates a plan for a 1D transform
  - Parameters:
    - **cufftHandle** plan handle
    - Size of the transform
    - **cufftType** type of the transform
    - Number of transforms to perform in batch
    - size_t pointer to the size in bytes of the workspace area
- **cufftType** types:
  - **CUFFT_R2C**, **CUFFT_C2R**, **CUFFT_C2C** for single precision transforms
  - **CUFFT_D2Z**, **CUFFT_Z2D**, **CUFFT_Z2Z** for double precision transforms

cuFFT plan and transform execution

- After plan creation the user can execute the transform
- The execution function depends on the type of the transform:
  - cufftExecC2C and cufftExecZ2Z for complex to complex transforms
  - cufftExecR2C and cufftExecD2Z for real to complex transforms
  - cufftExecC2R and cufftExecZ2D for complex to compreallex transforms

- cufftExec<t>():
  - Executes the transform specified by the plan:
  - Parameters:
    - cufftHandle plan handle
    - pointer to input array
    - pointer to output array
    - (*) Transform direction:
      - CUFFT_FORWARD for forward transform
      - CUFFT_INVERSE for the inverse transform
      - (*) This argument only exists when <t> is either C2C or Z2Z
  - If the input and output pointers are the same cuFFT will perform an in-place transform

cuFFT streams API and thread safety

- cuFFT permits the use of cuda streams (cudaStream_t) for increasing resource usage and introduce other levels of parallelism
- cuFFT is a thread safe library if functions host threads execute computations on different plans and output memory regions

- cufftSetStream():
  - Sets the stream to be used by cuFFT for subsequent computations
  - Parameters:
    - cuFFT handle to set the stream
    - cuda stream to use

Example: Cross correlation of two signals

```
cufftComplex *x, *y, *z;
// Allocate and initialize the signals
cufftHandle plan;
cufftCreate(&plan);      // Create the handle
cufftMakePlan1d(plan, n, CUFFT_C2C , 1);     // Create the plan
cufftExecC2C(plan, x, x, CUFFT_FORWARD);     // Compute the forward x
transform
cufftExecC2C(plan, y, y, CUFFT_FORWARD);     // Compute the forward y
transform
// Launch a kernel executing z[i]=x[i] * y[i]
cufftExecC2C(plan, z, z, CUFFT_INVERSE);
cufftDestroy(&plan);    // Destroy the plan
```


Compiling and linking against cuFFT

- In order to compile and link against cuFFT, the user needs to:
  - Include the header “#include <cufft.h>” in the appropriate files
  - Link against the cuFFT library:
    - For dynamic linking use the flag -lcufft
    - For static linking and a version of cuda 9.0 or later use the flags -lcufft_static -lculibos

讲授内容：CUDA libraries

① cuBLAS

② cuSOLVER

③ cuFFT

④ Thrust

C++ STL

- C++ Standard Template Library provides a set of type agnostic features as a way of simplifying C++ programming, some examples of these features are:
  - Type agnostic data structures: lists, map, vectors, etc.
  - Type agnostic algorithms: sort, fill, max, etc.
  - Iterators for data structures
- C++ STL features reside inside the C++ namespace std

- Example:

```
#include <algorithm>
#include <vector>
…
// Creates int vector with 10 elements
std::vector<int> a(10);
// Fill vector with 10 9 8 … 1
for(int i = 0 ; i < a.size(); ++i)
  a[i] = a.size() - i;
// Sort the vector
std::sort(a.begin(), a.end());
// a vector now contains 1 2 3 … 10
```

Thrust

- Thrust is a C++ template library written for CUDA devices and based on the C++ STL with the intention of simplifying certain aspects of the CUDA programming model
- Thrust provides users with three main functionalities:
  - The host and device vector containers
  - A collection of parallel primitives such as, sort, reduce and transformations
  - Fancy iterators
- All Thrust functions and containers reside inside the thrust namespace
- Thrust allows either host or device execution
- Since thrust is a template library it doesn't need to be linked against a library

Thrust vector containers

- Thrust provides 2 vector containers:
  - thrust::host_vector<T> for a vector stored in host memory
  - thrust::device_vector<T> for a vector stored in device memory
- Containers features:
  - Interoperability between the host_vector<T> and device_vector<T>, specifically:
    - Memory transfers using the assign operator and constructors
  - Interoperability with STL containers through iterators
  - API similar to the STL std::vector
  - Dynamic resizing of the container
- To use them include:
  - #<include> "thrust/host_vector.h"
  - #<include> "thrust/device_vector.h"

STL and Thrust Iterators

- Iterators provide a high-level abstraction for data access to containers, specifically:
  - Contains information of a specific element in the container
  - Contains information on how to access other elements in the container
- There are several kinds of iterators for example:
  - Random access iterators, allows access to any element in the container
  - Bidirectional iterators, allows access to the previous and next element
  - Forward iterators, allows access to the next element

- Example:

```
thrust::host_vector<int> a(10);

// Fill a with 0 1 2 … 9

auto it = a.begin(); // Iterator pointing to the first element

int tmp = *it; // Dereference the iterator, tmp holds 0

it = a.begin() + 5; // Iterator pointing to a[5]

++it; // Iterator pointing to a[6]

*it = tmp; // Equivalent to performing  a[6] = tmp;

for( it = a.begin(); it != a.end(); ++it)

  std::cout << *it << std::endl; // Print a contents
```

*** It's important to observe that a.end() doesn't point to a[9], it refers to a position past the final element

Example: Thrust vector containers interoperability

```
#include <thrust/host_vector.h>

#include <thrust/device_vector.h>

#include <list>

…

// Declare and initialize STL list container

std::list<float> stl_list;

stl_list.push_back(3.14);

stl_list.push_back(2.71);

stl_list.push_back(0.);

// Initialize thrust device vector with the STL list containing {3.14, 2.71, 0.}

thrust::device_vector<float> vector_D(stl_list.begin(), stl_list.end());

// Perform computations on vector_D

// Create host vector and copy back results to host memory

thrust::host_vector<float> vector_H = vector_D;
```

Thrust pointers and memory

- Thrust provides a pointer interface thrust::device_ptr<T> to be used when data was allocated using cudaMalloc or similar mechanisms
- The thrust::device_ptr<T> interface is compatible with all Thrust algorithms and allows similar semantics as iterators
- To use Thrust device pointer the user needs to include:
  - #include <thrust/device_ptr.h>

- Example:

```
#include <thrust/device_ptr.h>
…
float *a;
// Allocate device memory
cudaMalloc((void**)&a, 1024 * sizeof(float));
// Initialize thrust pointer with a
thrust::device_ptr<float> a_thrust(a);
// Assign to a_thrust 0, 0 + 2, 2 + 2, …, 2046
thrust::sequence(a_thrust, a_thrust + 1024, 0, 2);
// Extract the pointer being used by a_thrust
float *b = thrust::raw_pointer_cast(a_thrust);
```


Thrust algorithms

- Provides common parallel algorithms like:
  - Reductions
  - Sorting
  - Reorderings
  - Prefix-sums
  - Transformations
- All Thrust algorithms have host and device implementations
- If an algorithm is invoked with an iterator, thrust will execute the operation in the host or device according to the iterator memory region
- Except for thrust::copy which can copy data between host and device all other routines must have all its arguments reside in the same place

Thrust transformations

- Transformations apply a function to each element in the input range and stores the result in the output range
- Exists a diverse set of available transformations such as:
  - thrust::transform applies out-of-place a function to each element in a data range
  - thrust::transform_if applies out-of-place a function if a condition is met
  - thrust::for_each applies in-place a unary function
- To use Thrust transormation algorithms the user needs to include:
  - #include <thrust/transform.h>

```
#include <thrust/transform.h>
...
struct saxpy_functional {
  float alpha;
  float operator()(float &x, float &y) {
    return alpha * x + y;
  }
};
...
thrust::device_vector<float> a(1024);
thrust::device_vector<float> b(1024);
thrust::device_vector<float> c(1024);
// Initialize a and b
saxpy_functional saxpy;
saxpy.alpha = 2;
thrust::transform(a.begin(), a.end(), b.begin(), c.begin(), saxpy); // Perform c[i] = 2 * a[i] + b[i]
```


Thrust function objects

- Thrust provides a collection of predefined operators usable by Thrust algorithms like transform and reduce
- Available operator categories are:
  - Arithmetic operators
  - Comparison operators
  - Logical operators
  - Bitwise operators
  - Generalized identity operators

- Example:

```
#include <thrust/functional.h>
#include <thrust/transform_reduce.h>
...
thrust::device_vector<double> x;
double zero = 0;
double norm;
// Initialize data
// Compute x norm using the thrust::square (x * x)
and    thrust::plus (x + y) operators
norm = std::sqrt(thrust::transform_reduce(x.begin(),
       x.end(), thrust::square <double>(), zero,
       thrust::plus<double>()));
```

Thrust sorting routines

- Thrust provides sorting algorithms for GPU namely:
  - thrust::sort for sorting an array
  - thrust::sort_by_key for sorting an array by a key array
- Stable sorting routines are also available:
  - thrust::stable_sort
  - thrust::stable_sort_by_key
- To use Thrust sorting algorithms the user needs to include
  - #include <thrust/sort.h>

- Example:

```
#include <thrust/sort.h>

…

thrust::device_vector<int> keys(4);

thrust::device_vector<double> values(4);

// Initialize data with:

//       keys = {3, 1, 2, 7}

//       values = {1., 2., 3., 4. }

// Launch thrust sorting by key algorithm

thrust::sort_by_key(keys.begin(), keys.end(), values.begin())

// Values after sorting:

//       keys = {1, 2, 3, 7}

//       values = {2., 3., 1., 4. }
```


Thrust fancy iterators

- Thrust provides a collection of special iterators to be used by Thrust algorithms enhancing language programmability:
  - thrust::constant_iterator<T> is an iterator with constant value
  - thrust::counting_iterator<T> is an iterator addressing a range of numbers
  - thrust::zip_iterator is an iterator zipping two memory regions together into a single object of pairs

- Example:

```
#include <thrust/iterator/zip_iterator.h>

…

thrust::device_vector<int> A;

thrust::device_vector<float> B;

…

auto begin =
thrust::make_zip_iterator(thrust::make_tuple(A.be
gin(), B.begin()));

auto end  =
thrust::make_zip_iterator(thrust::make_tuple(A.en
d(),   B.end()));

thrust::maximum< thrust::tuple<int,float>> max_op;

thrust::reduce(begin, lendast, init, max _op);
```

Thrust execution policies and streams

- Execution policies gives users control over certain runtime execution decisions
- Common execution policies are:
  - thrust::seq for sequential execution
  - thrust::omp::par for parallel execution using the OpenMP backend
  - thrust::cuda::par for CUDA execution
  - thrust::host for host execution
  - thrust::device for device execution

- Thrust allows the use of CUDA streams through the execution policy:
  - thrust::cuda::par.on(stream):
    - Sets the stream to use by the Thrust algorithm
    - Parameters:
      - Stream to use
- To use this policy the user needs to include:
  - #include <thrust/system/cuda/execution_policy.h>
- Example:

```
cudaStream_t stream;
...
thrust::sort(thrust::cuda::par.on(stream),
dev_vector.begin(), dev_vector.end())
```