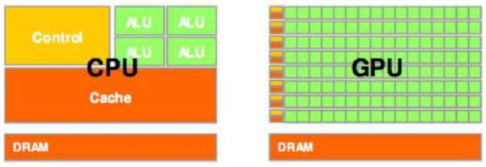
GPU

CPUs vs. GPUs

- The design of a CPU is optimized for sequential code performance.
  - out-of-order execution, branch-prediction
  - large cache memories to reduce latency in memory access
  - multi-core
- GPUs:
  - many-core
  - massive floating point computations for video games
  - much larger bandwidth in memory access
  - no branch prediction or too much control logic: just compute

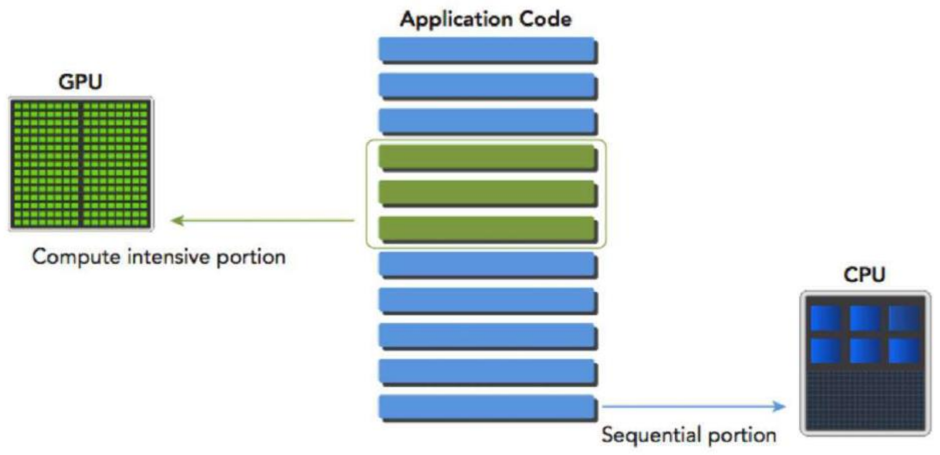


Heterogeneous Computing

- CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks.
- The CPU is optimized for dynamic workloads marked by short sequences of computational operations and unpredictable control flow;
- GPUs aim at workloads that are dominated by computational tasks with simple control flow.

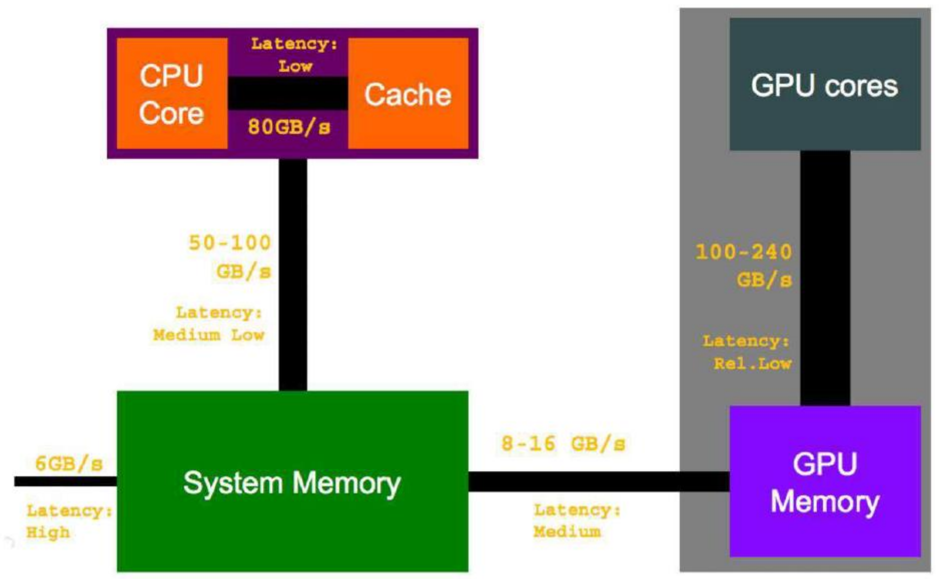
3

Heterogeneous Computing



4

Bandwidth in a CPU-GPU System



5

Cuda

CUDA, **Compute Unified Device Architecture**. CUDA produces C/C++ for the system processor (host) and a C and C++ dialect for the GPU (device).

A similar programming language is OpenCL, which several companies are developing to offer a vendor-independent language for multiple platforms.

6

The CUDA Thread

The unifying theme of all these forms of parallelism is the CUDA Thread.

Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, and SIMD.

NVIDIA classifies the CUDA programming model as single instruction, multiple thread (SIMT).

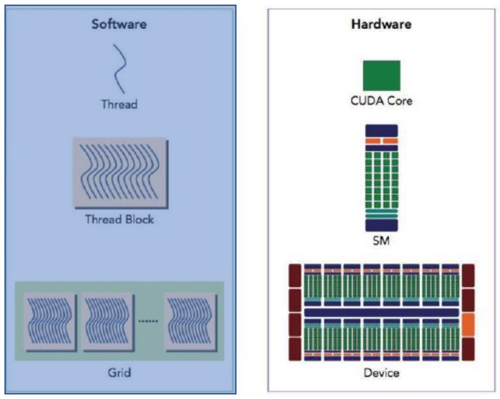
These threads are blocked together and executed in groups of threads, called a Thread Block.

A **Grid** is the **code** that runs on a **GPU that consists of a set of Thread Blocks**.

The organization of a Grid can influence the global execution by means of the **identifier for blocks**, the **identifier of the thread in the block, and** the **number of threads per block**.

We call the hardware that executes a whole block of threads a multithreaded SIMD Processor.

The programmer's view The run-time view



7

blockIdx, threadIdx, blockDim.

The function call for the function name that runs on the GPU is

name < <<dimGrid, dimBlock>> > (… parameter list…)

where **dimGrid** (in Thread Blocks) and **dimBlock** (in threads) specify the dimensions of the code and the dimensions of a block.

The code can use at run time:
- The identifier for blocks (**blockIdx**) and
- the identifier for each thread in a block (**threadIdx**),
- the number of threads per block (**blockDim**).

Different Thread Blocks cannot communicate directly, although they can coordinate using atomic memory operations in global memory.

To simplify scheduling by the hardware, CUDA requires that Thread Blocks be able to execute independently and in any order.
The **GPU hardware handles parallel execution and thread management**; it is not done by applications or by the operating system.

8

Grid and Thread Blocks

Let’s suppose we want to multiply two vectors together, each 8 k elements long:

A = B * C

The GPU code that works overall 8192 (8 k) element multiply is called a Grid.

The Grid is composed of Thread Blocks, each with up to 512 elements.

- Note that a SIMD instruction executes 32 elements at a time.

With 8192 elements in the vectors, this example thus has 16 Thread Blocks because

8192 / 512 = 16

The **Grid** and **Thread Block** are **programming abstractions implemented in** GPU **hardware** that help programmers organize their CUDA code.

9

Thread Block Scheduler

- A **Thread Block** is assigned to a processor that executes that code, which we call a **multithreaded SIMD Processor**, by the Thread Block Scheduler.

```
axpy<<<8192/512, 512>>>(8192, A, B);
```

- The programmer tells the Thread Block Scheduler, which is implemented in hardware, how many Thread Blocks to run.

In this example, it would send 16 Thread Blocks to multithreaded SIMD Processors to compute all 8192 elements of this loop

10

CPU code for DAXPY

Y=a*X+Y

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);


// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
            for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i]
}
```

Each iteration is independent from the others, allowing the loop to be transformed straightforwardly into a parallel code where each loop iteration becomes a separate thread.

11

The programmer's view: *the GPU code*

The programmer determines the parallelism in CUDA explicitly by specifying the grid dimensions and the number of threads per SIMD Processor.
By assigning a single thread to each element, there is no need to synchronize between threads when writing results to memory.

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);


// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
if (i < n) y[i] = a*x[i] + y[i];
}
```

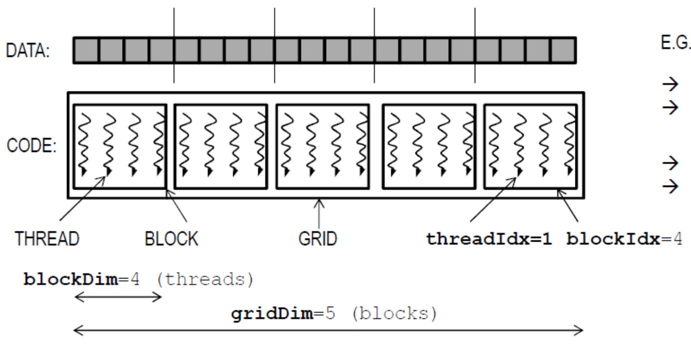
We launch **n** threads, one per vector element, with **256** CUDA Threads per Thread Block in a multithreaded SIMD Processor.

GPU function starts by calculating the corresponding element index i based on the block ID, the number of threads per block, and the thread ID.

As long as this index is within the array (i < n), it performs the multiply and add.

12

Subdividing the Work



E.G.: 20 data elements (e.g., 20 ints)

→ 5 data slices
→ 4 elements per slice

→ One **THREAD** per data element
→ One **BLOCK** per data slice

`threadIdx` `blockIdx` `blockDim` `gridDim` are built-in variables

`threadIdx` `blockIdx`: each thread will get its own value

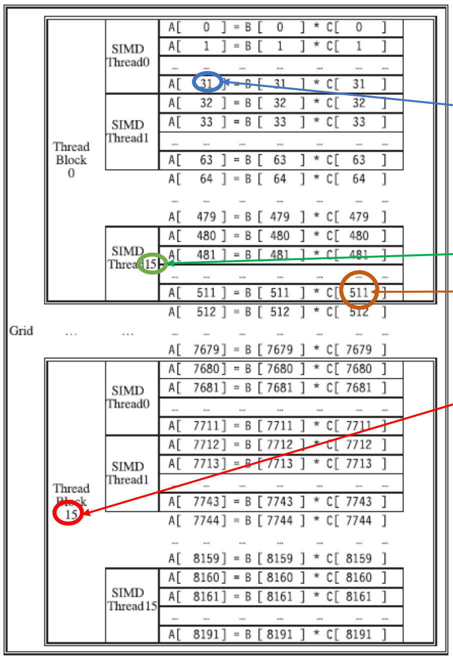
Note: for 1D data organization I should append the suffix "`.x`" to such built-in variables

13

The code for Y[i] = sum (X[i]*a + Y[i])

```
; calculate i
shl.u32 R8, blockIdx, 8;            Thread Block ID * Block size; (256 or 2^8)
add.u32 R8, R8, threadIdx ;         R8 = i = my CUDA Thread ID

; calculate the offset for i element
shl.u32 R8, R8, 3 ;                 offset of the i (R8) element (64-bit large)

; load X(i) and Y(i)
ld.global.f64 RD0, [X+R8];          RD0 = X[i]
ld.global.f64 RD2, [Y+R8];          RD2 = Y[i]

; calculate sum (X[i]*a + Y[i])
mul.f64 RD0, RD0, RD4 ;             Product in RD0 = RD0 * RD4; (scalar a)
add.f64 RD0, RD0, RD2 ;             Sum in RD0 = RD0 + RD2 (Y[i])

; store sum (X[i]*a + Y[i])  into Y(i)
st.global.f64 [Y+R8], RD0;          Y[i] = sum (X[i]*a + Y[i])
```

14


Grid
Thread Block 0
SIMD Thread0
A[0] = B[0] * C[0]
A[1] = B[1] * C[1]
A[31] = B[31] * C[31]
SIMD Thread1
A[32] = B[32] * C[32]
A[33] = B[33] * C[33]
A[63] = B[63] * C[63]
A[64] = B[64] * C[64]
A[479] = B[479] * C[479]
SIMD Thread15
A[480] = B[480] * C[480]
A[481] = B[481] * C[481]
A[511] = B[511] * C[511]
A[512] = B[512] * C[512]
A[7679] = B[7679] * C[7679]
Thread Block 15
SIMD Thread0
A[7680] = B[7680] * C[7680]
A[7681] = B[7681] * C[7681]
A[7711] = B[7711] * C[7711]
SIMD Thread1
A[7712] = B[7712] * C[7712]
A[7713] = B[7713] * C[7713]
A[7743] = B[7743] * C[7743]
A[7744] = B[7744] * C[7744]
A[8159] = B[8159] * C[8159]
SIMD Thread15
A[8160] = B[8160] * C[8160]
A[8161] = B[8161] * C[8161]
A[8191] = B[8191] * C[8191]

The software architecture

Threads, Thread Blocks, and Grid

- Each thread of SIMD instructions calculates **32** elements per instruction, and in this example, each Thread Block contains **16** threads of SIMD instrunctions and the Grid contains **16** Thread Blocks.
- Each Thread Block calculates 32*16=**512** elements.
- The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor.
- Only SIMD Threads in the same Thread Block can communicate via local memory.
  - The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.

15

The run-time view: SIMD Thread scheduler

The hardware creates, manages, schedules, and executes thread of SIMD instructions.

A traditional thread that contains exclusively SIMD instructions.

These threads of SIMD instructions have their own PCs, and they run on a multithreaded SIMD Processor.

The SIMD Thread Scheduler knows which threads of SIMD instructions are ready to run and then sends them off to a dispatch unit to be run on the multithreaded SIMD Processor.

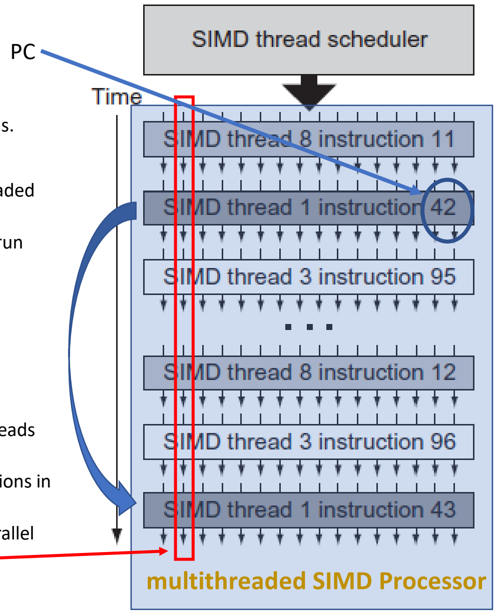
GPU hardware has two levels of hardware schedulers:

1. the Thread Block Scheduler that assigns Thread Blocks to multithreaded SIMD Processors and
2. the SIMD Thread Scheduler within a SIMD Processor, which schedules when threads of SIMD instructions should run.

The SIMD instructions of these threads are 16 wide, so each thread of SIMD instructions in this example would compute 16 of the elements of the computation.

Because the thread consists of SIMD instructions, the SIMD Processor must have parallel functional units to perform the operation.

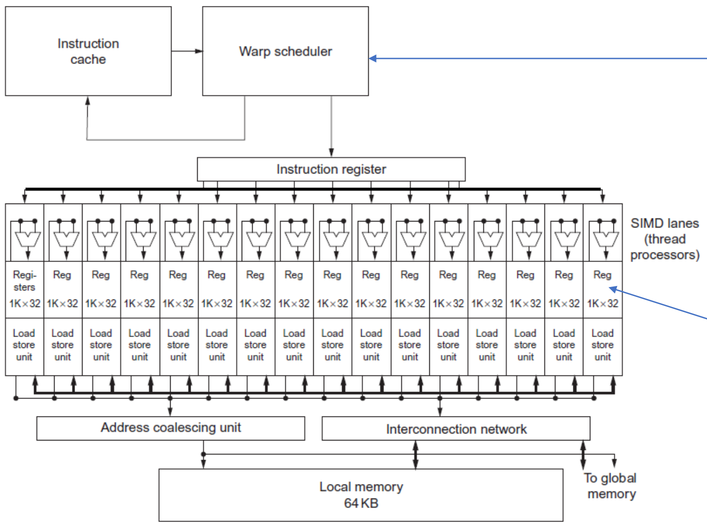
We call them **SIMD Lanes**, and they are quite similar to the Vector Lanes.


PC
SIMD thread scheduler
Time
SIMD thread 8 instruction 11
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43
multithreaded SIMD Processor

16

SIMT, multithreaded SIMD Processor

16 SIMD Lanes



The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs).

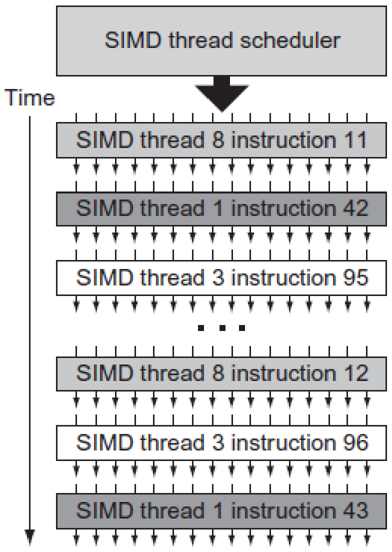
Each lane has 1024 32-bit registers.

With the Pascal GPU, each 32-wide thread of SIMD instructions is mapped to 16 physical SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes **2 clock** cycles to complete.

17

Latency of the memory instructions

- The latency of memory instructions is variable because of hits and misses in the caches and the TLB, thus the requirement of a scoreboard to determine when these instructions are complete.
- The assumption of GPU architects is that GPU applications have so many threads of SIMD instructions that multithreading can both **hide the latency to DRAM and increase the utilization of multithreaded SIMD Processors**.


SIMD thread scheduler
Time
SIMD thread 8 instruction 11
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43

18

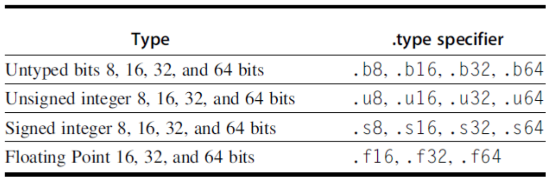
Instruction set

The hardware instruction set is hidden from the programmer.

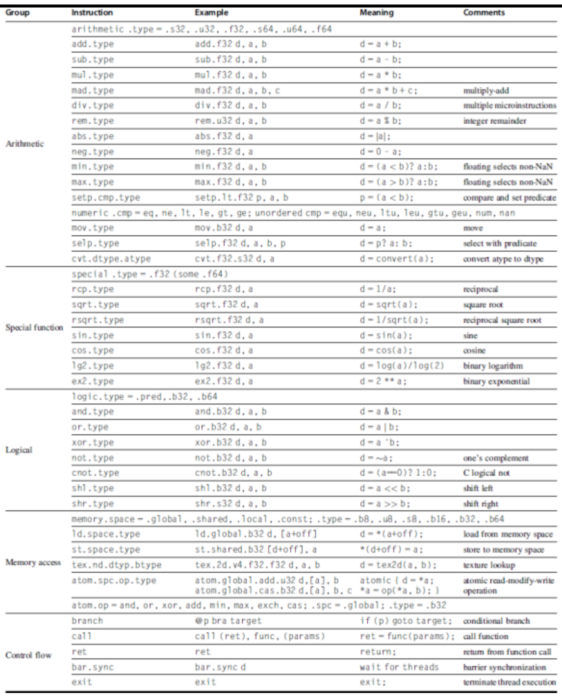
The format of a PTX instruction is:

opcode.type **d**, **a**, **b**, **c**;

where **d** is the destination operand; **a**, **b**, and **c** are source operands.



Type	.type specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating Point 16, 32, and 64 bits	.f16, .f32, .f64



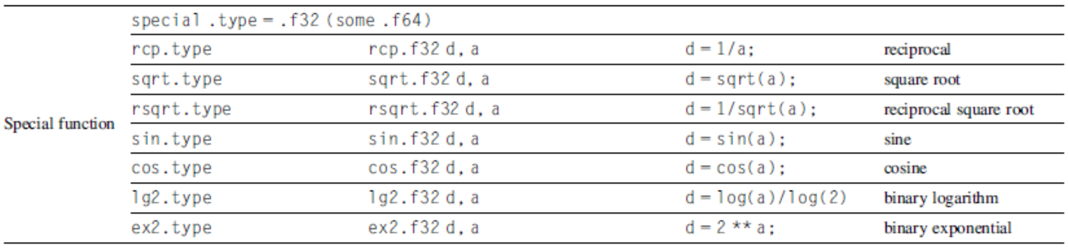
Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a - b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential
Logical	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
Memory access	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup

	atom.spc.op.type	atom.global.add.u32 d,[a], b
		atom.global.cas.b32 d,[a], b, c
			atomic { d = *a;
			*a = op(*a, b); }
				atomic read-modify-write operation

	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Control flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

19

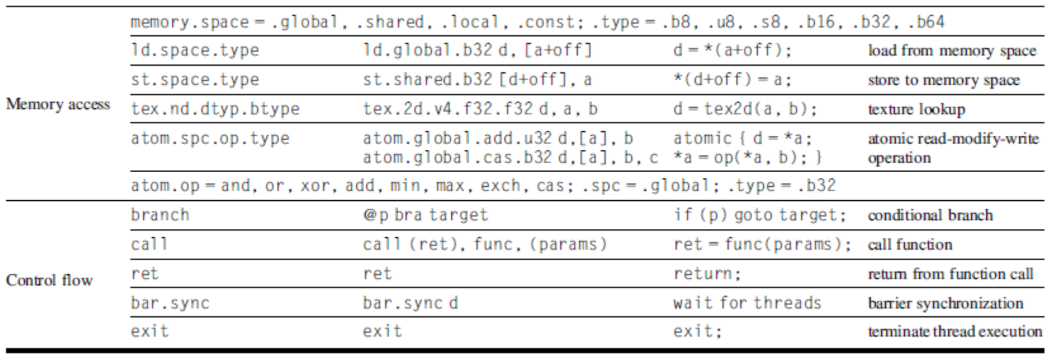
Special instructions



Special function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential

20

Memory and control flow

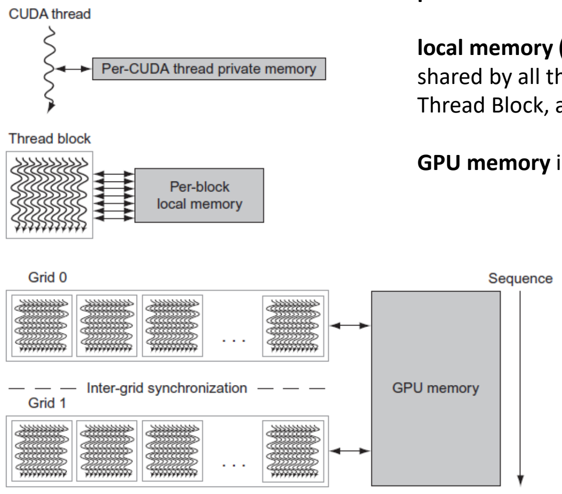


	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
Memory access	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup

	atom.spc.op.type	atom.global.add.u32 d,[a], b
		atom.global.cas.b32 d,[a], b, c
			atomic { d = *a;
			*a = op(*a, b); }
				atomic read-modify-write operation

	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
Control flow	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

NVIDIA GPU Memory Structures



private memory is private to a single CUDA Thread.

local memory (limited in size, typically to 48 Kbytes) is shared by all threads of SIMD instructions within a Thread Block, and

GPU memory is shared by all Grids,

The multithreaded SIMD Processor dynamically allocates portions of the local memory to a Thread Block when it creates the Thread Block, and frees the memory when all the threads of the Thread Block exit.

That portion of local memory is private to that Thread Block.

The system processor, called the host, can only read or write GPU Memory.

Caches and multithreading

GPUs traditionally uses smaller streaming caches,

- rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM.
- the chip area is spent instead on computing resources and on the large number of registers to hold the state of many threads of SIMD instructions.

All recent GPUs have caches to reduce latency.

- The more threads need to run during a memory access, which in turn requires more registers.

23