



INTRODUCTION TO GPU ARCHITECTURE

Balakrishnan Srinivasan, 20/04/2021



OUTLINE

Classes of Parallelism

Parallel Architecture Taxonomy

CPU Architecture Concepts

Visual Performance Analysis

Challenges & Trends in Computer Architecture

GPU Architectures & programming

GPU Computational Structures

Key Takeaways

DEFINING PERFORMANCE

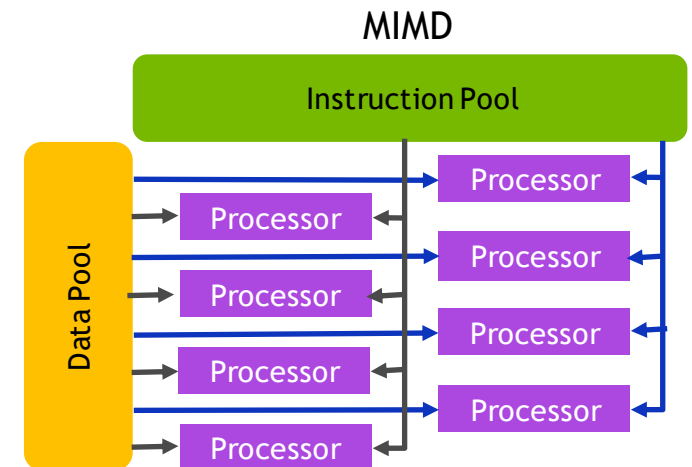
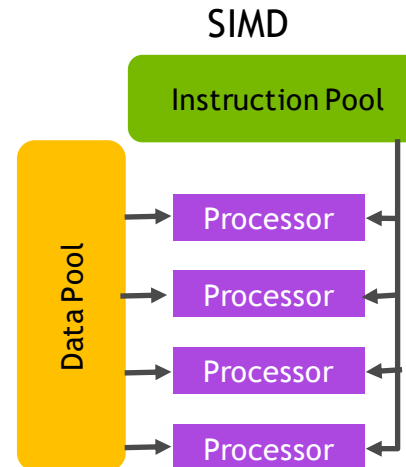
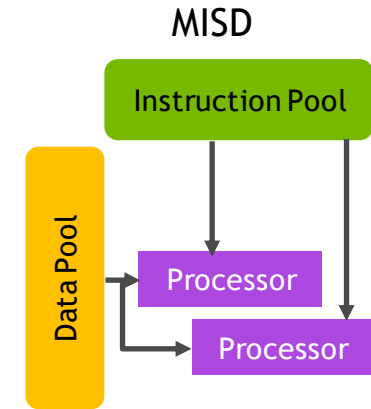
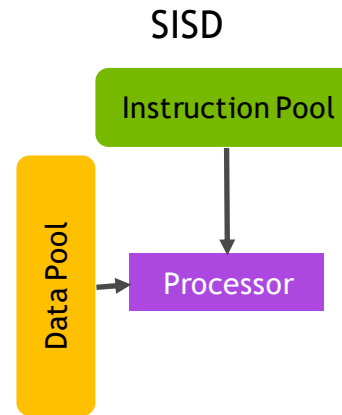
- ▶ *Response-time* - time between start and completion of task - also *execution time*, *latency*
 - ▶ Usually, a concern for devices like mobile phones
- ▶ Throughput - Amount of work done in a given time - also called Bandwidth
 - ▶ Datacenter managers want better throughput from their servers
- ▶ Clocking a processor faster can improve both response time and throughput
- ▶ Throughput-oriented systems can however tradeoff latency of a single task with overall system performance
- ▶ Exploiting parallelism is key to improving both response-time and throughput

CLASSES OF PARALLELISM

- ▶ Application point of view:
 - ▶ Data-level parallelism (DLP) - many data items are processed at the same time
 - ▶ Task-level parallelism (TLP) - work is independent and can mostly run in parallel
- ▶ Hardware point of view:
 - ▶ Instruction-level parallelism - exploits low level parallelism in program by identifying independent operations (often with the help of a compiler) that can be performed in parallel or at a medium level through speculative execution
 - ▶ Data-level parallelism - single instruction is applied to a collection of data items
 - ▶ Thread-level parallel - exploits both DLP and TLP in a tightly coupled hardware model - interaction among threads is possible
 - ▶ Request-level parallelism - exploits parallelism among decoupled tasks - programmer specified, or operating system driven

FLYNN'S TAXONOMY FOR COMPUTER ARCHITECTURES

- ▶ Single instruction stream, single data stream (SISD)
- ▶ Single instruction stream, multiple data streams (SIMD)
- ▶ Multiple instruction streams, single data stream (MISD)
- ▶ Multiple instruction streams, multiple data streams (MIMD)



Flynn, M., "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21, No. 9, September 1972.

COMPUTER ARCHITECTURE

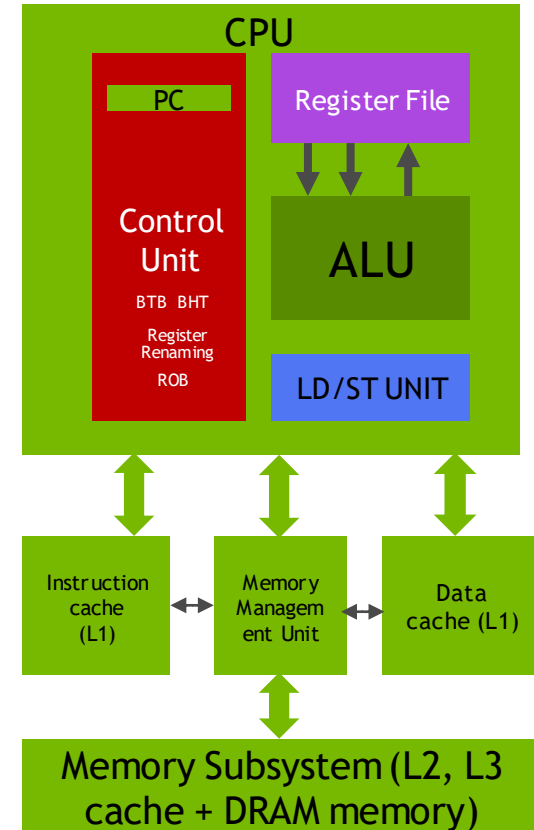
- ▶ Instruction set architecture (ISA)
 - ▶ Programmer-visible interface
- ▶ Microarchitecture/Organization
 - ▶ Design of Central Processing Unit (CPU)
 - ▶ Memory organization
 - ▶ Memory Interconnect
- ▶ Hardware
 - ▶ Detailed logic design
 - ▶ Packaging
 - ▶ clock-rate, memory systems etc.

INSTRUCTION SET ARCHITECTURE

- ▶ Instruction set architecture choices
 - ▶ **Class of ISA:** load-store, register-memory
 - ▶ **Memory Addressing:** byte address, aligned
 - ▶ **Addressing modes:** Register, immediate, displacement
 - ▶ **Type & Size of operands:** 8-bit, 16-bit, 32-bit, double-word, IEEE 754 floating-point - single & double precision, extended double precision
 - ▶ **Operation:** arithmetic, logical, data transfer, control, floating point
 - ▶ **Control-flow instruction:** conditional branches, unconditional branches, procedure calls and returns
 - ▶ **Encoding:** Fixed-length, variable length

CPU ARCHITECTURE

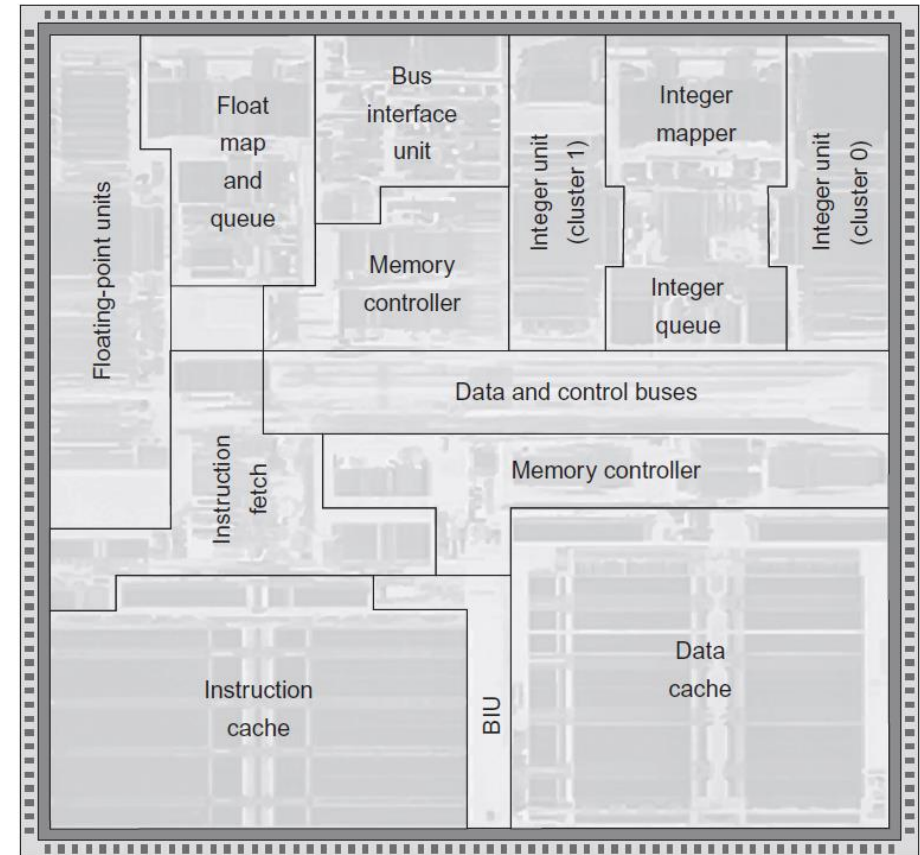
- ▶ Performance driven by Instruction Level Parallelism (ILP)
- ▶ Pipelining delivers instruction throughput
 - ▶ Instruction per cycle (IPC)
- ▶ Branch Prediction improves IPC in the presence of branches
 - ▶ In integer code, every 4th instruction is likely a branch
 - ▶ Out-of-order execution delivers more IPC
 - ▶ Misprediction can be expensive - flush pipelines and restart
- ▶ Large caches deliver data fast to the CPU
 - ▶ Exploits principle of locality - temporal & spatial



Simplified Block Diagram of a single core processor

CPU ARCHITECTURE

- ▶ AIM: Make a sequential program execute faster
- ▶ Lots of resources dedicated to:
 - ▶ Instruction fetch
 - ▶ Caches
 - ▶ Branch prediction
- ▶ Sustaining an ILP greater than 6 is tough
- ▶ For a massively parallel application, better architectural choices can be made



Floorplan of the Alpha 21264

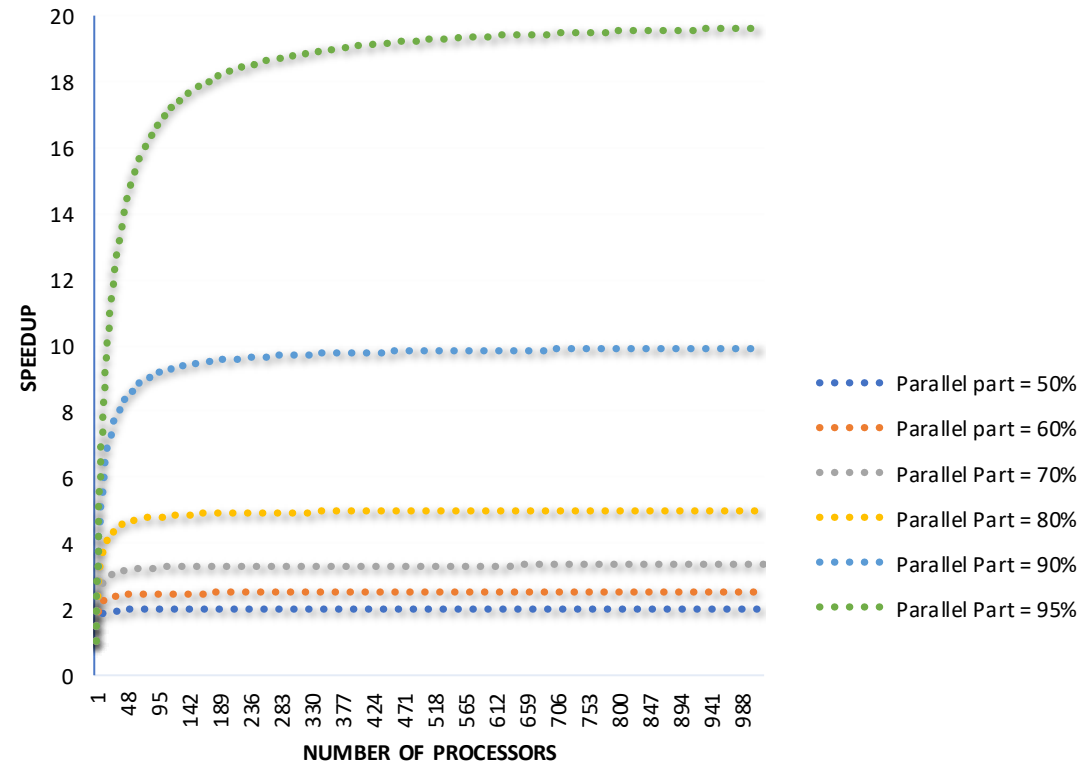
AMDAHL'S LAW

$$Speedup = \frac{t_s + t_p}{t_s + \frac{t_p}{N}} = \frac{1}{f + (1-f)/N} < \frac{1}{f}$$

t_s and t_p : time spent in the sequential and parallel portions of the code and $f = \frac{t_s}{t_s+t_p}$ is the fraction of program that is sequential

N is the number of processors

- ▶ The theoretical speedup is limited by the part of the program that cannot benefit from an architectural enhancement
- ▶ A missed opportunity to use the enhancement will contribute to the performance limiter



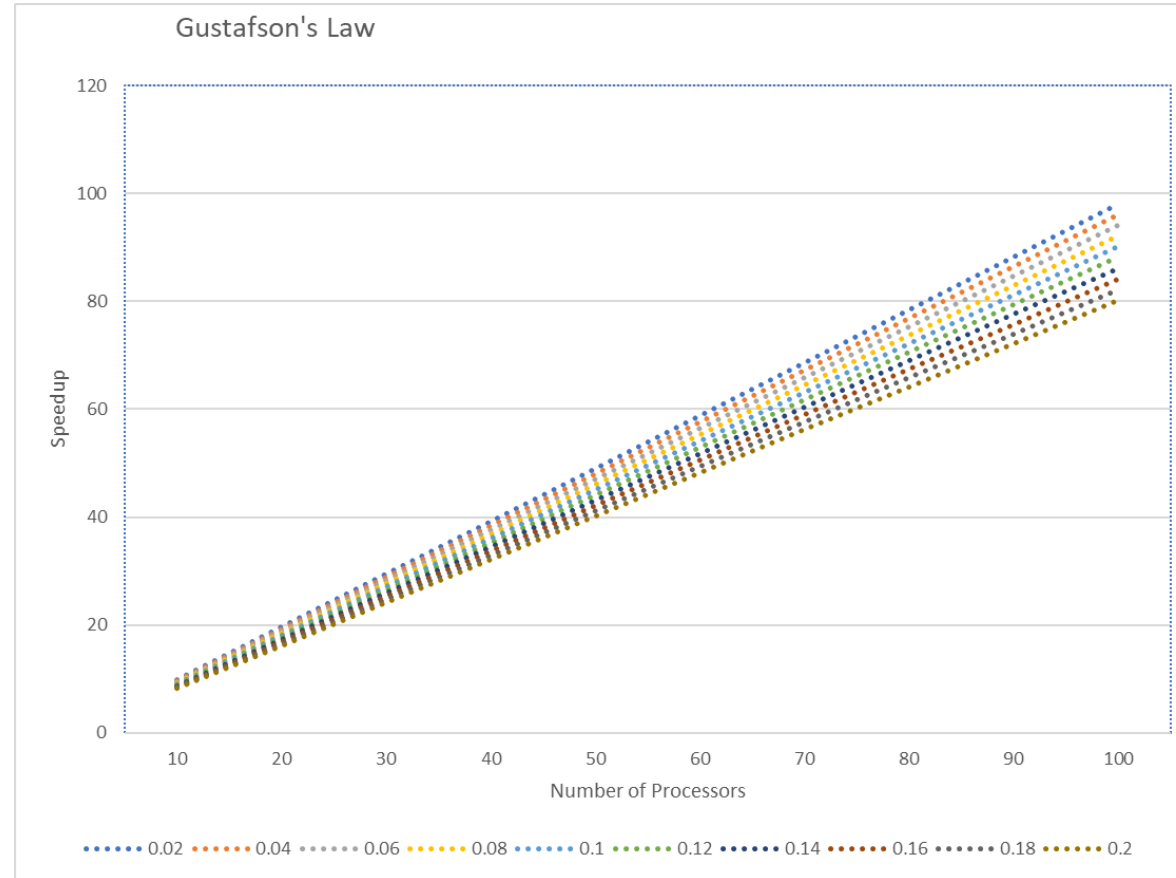
AMDAHL'S AND GUSTAFSON'S LAW

- ▶ Assume a task that has 1 Hour of serial work and 100 hours of work that can be parallelized
 - ▶ On a serial machine we would take 101 Hours
 - ▶ On a 100-processor machine we will take $1 + 100/100 = 2$ hours
 - ▶ Speedup = $101/2 = 50.5$
 - ▶ Amdahl's law says that given a problem size the parallel portion of the work can be executed faster but the serial portion will be the bottleneck
 - ▶ The best speedup we can get close to 101
 - ▶ Gustafson's law says that given 2 hours of compute-time, we can solve for a problem-size that would take 1000 hours to solve by using 1000 processors
 - ▶ Speedup = $(1 + 1000)/2 = 500.5$

Having chosen a problem size, we are limited by Amdahl's law. However, this should not prevent us from solving larger problems

GUSTAFSON'S LAW

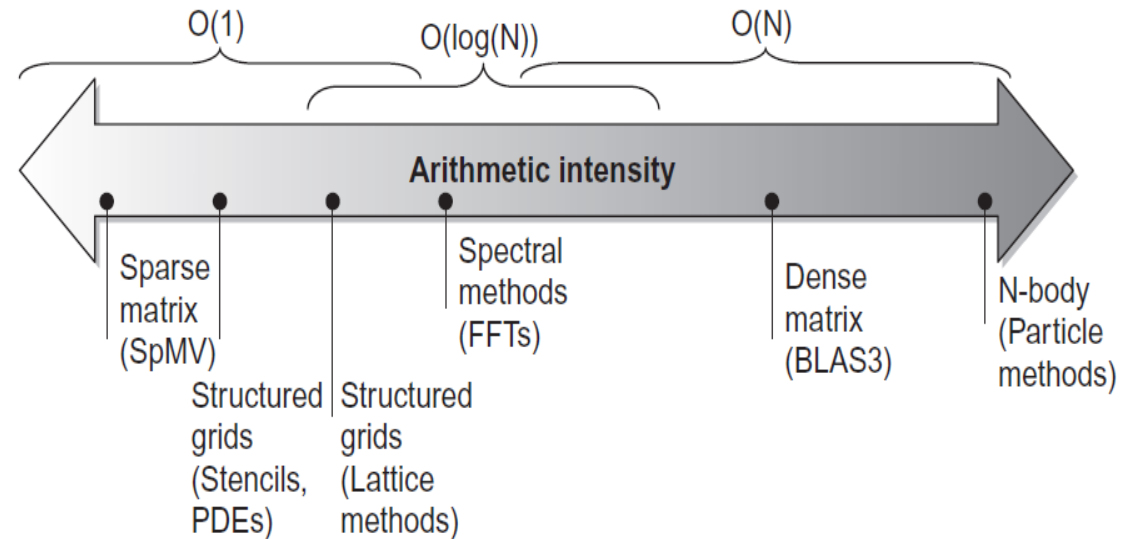
- ▶ $Speedup = \frac{t_s + t_p}{t_s + \frac{t_p}{N}}$ - Amdahl's law
- ▶ Sequential fraction f^* on parallel machine = $\frac{t_s}{t_s + \frac{t_p}{N}}$
- ▶ $\frac{t_p}{t_s + \frac{t_p}{N}} = N * (1 - f^*)$
- ▶ $Speedup = f^* + N * (1 - f^*)$
- ▶ Amdahl's law - problem size is constant
- ▶ Gustafson's law - time is constant



ANALYZING PERFORMANCE

The Roofline Visual Performance Model

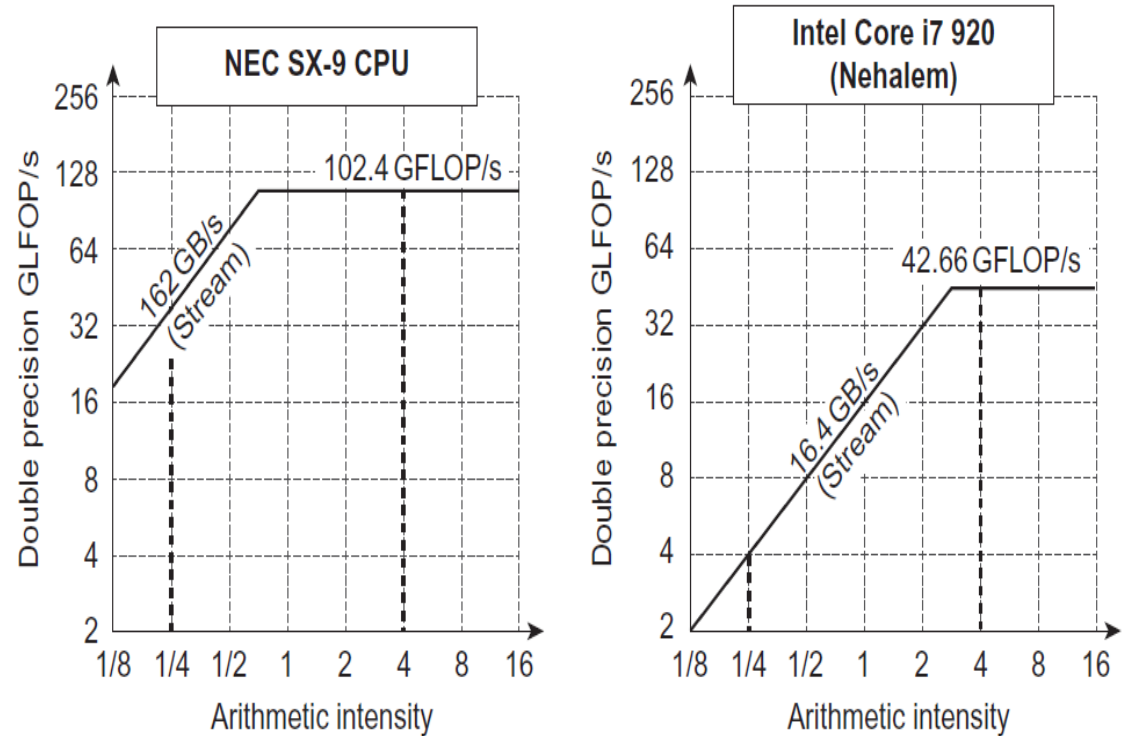
- ▶ Ties together floating-point performance, memory performance and arithmetic intensity in a 2D graph
- ▶ Arithmetic intensity - ratio of floating-point operations per byte of memory accesses
 - ▶ Dense Matrix computations - arithmetic intensity scales with problem size
 - ▶ Many kernels exist with arithmetic intensities independent of problem size



* Computer Architecture: A quantitative Approach - 6th Edition, John L. Hennessey and David A. Patterson

THE ROOFLINE MODEL

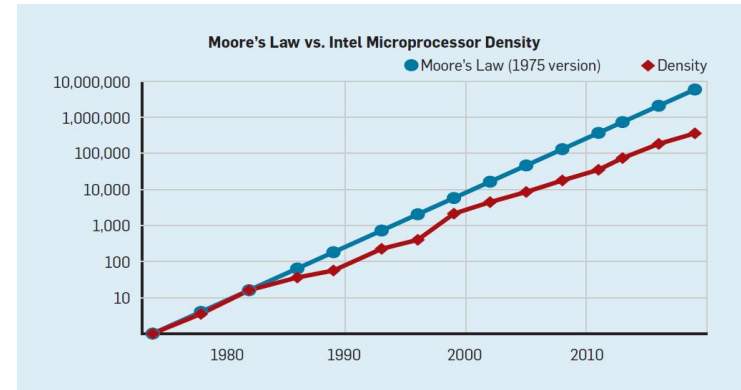
- ▶ NEC SX-9 – Vector supercomputer
 - ▶ Peak double-precision FP performance - 102.4 GFLOPS/s
 - ▶ Peak memory bandwidth - 162 GB/s (Stream benchmark)
- ▶ Core i7 920
 - ▶ Peak double-precision FP performance – 42.66 GFLOP/s
 - ▶ Peak memory bandwidth – 16.4 GB/s
- ▶ At arithmetic-intensity of 4 FLOPS/byte both processors operate at peak performance
- ▶ At arithmetic-intensity of $\frac{1}{4}$ FLOPS/byte SX-9 is 10x faster than Core i7



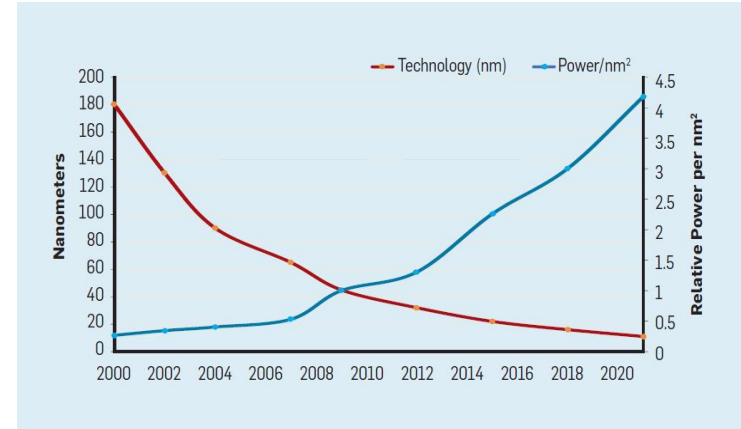
CURRENT CHALLENGES IN COMPUTER ARCHITECTURE

End of Moore's law & Dennard Scaling

- ▶ Moore's law (more a prediction):
 - ▶ transistor count doubles approximately every 24 months
 - ▶ We now have a 15-fold gap between Moore's prediction and current capability - all exponential laws come to an end
- ▶ Dennard Scaling
 - ▶ As transistor density increase, power consumption per transistor would drop, so the power per mm^2 of silicon would be near constant
 - ▶ Dennard Scaling slowed down in 2007 to almost nothing in 2012



Transistors per chip on Intel Microprocessors vs Moore's Law



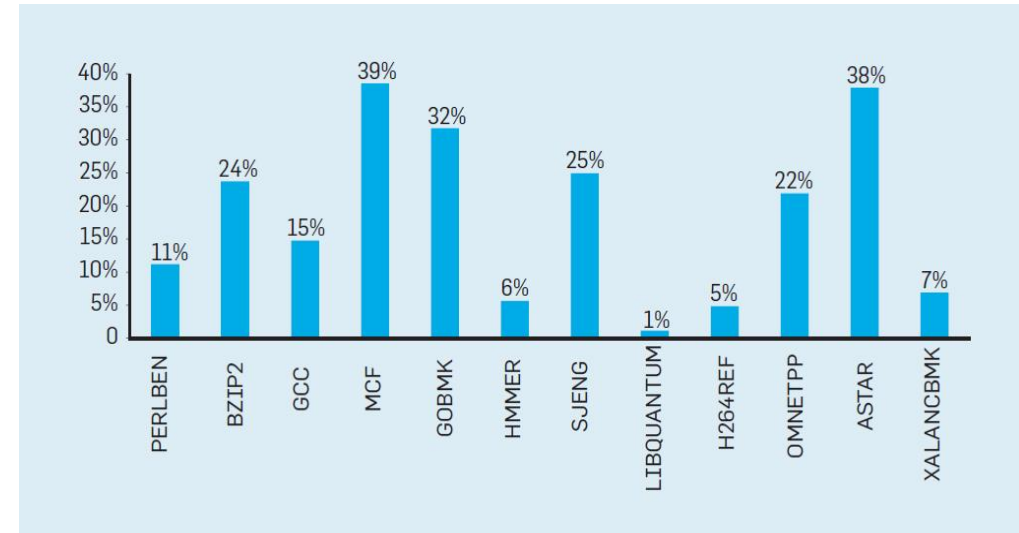
CURRENT CHALLENGES IN COMPUTER ARCHITECTURE

▶ Wasted Instructions

- ▶ A 4-issue processor & 15-stage pipeline has 60 instructions in the pipeline every cycle
- ▶ Includes 15 branch instructions
- ▶ Need to predict branches and speculate code. Speculation - source of performance and inefficiency
- ▶ limiting wasted work to 10% of the time requires processor to predict each branch correctly 99.3% of the time - difficult to achieve

▶ Overlooked Security

- ▶ Side channel attacks
- ▶ Meltdown, Spectre, and other attacks.
- ▶ Flaw in the hardware implementation is used to access protected information



Wasted instructions as a percentage of all instructions completed on an Intel Core i7 for a variety of SPEC integer benchmarks

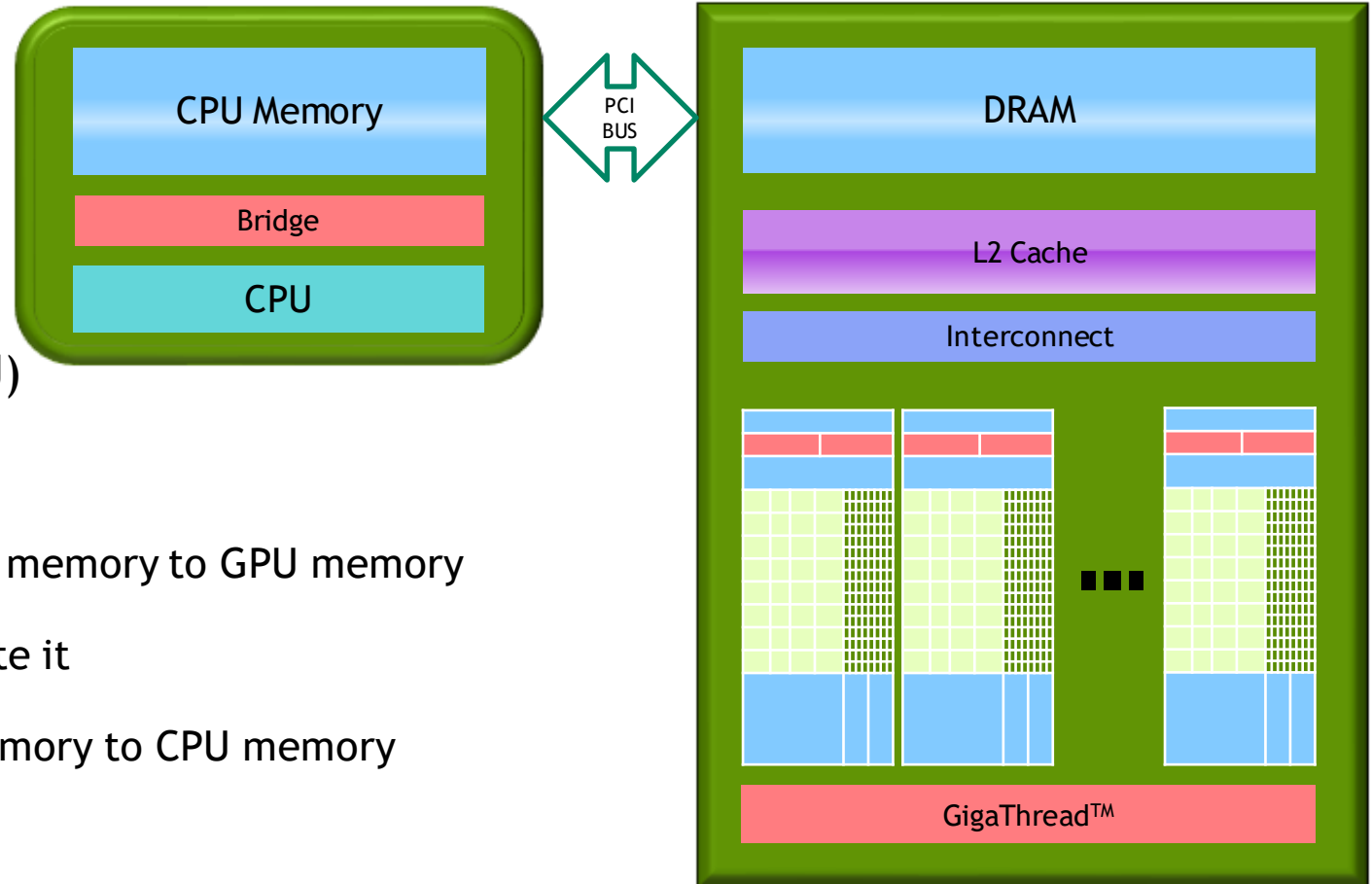
John L. Hennessy and David A. Patterson, A New Golden Age for Computer Architecture, Turing Lecture, Communications of the ACM, February 2019, pages 48-60.

OPPORTUNITIES IN COMPUTER ARCHITECTURE

- ▶ Domain Specific Architectures
 - ▶ Class of architectures tuned for a particular domain
 - ▶ GPUs, neural network processors for deep learning etc.
 - ▶ Use hardware structures more suited to the domain - e.g., SIMD parallelism
 - ▶ Make more effective use of memory hierarchy
 - ▶ Exploit memory-access patterns known at compile time
- ▶ Domain Specific Languages
 - ▶ Make vector, dense matrix and sparse matrix operations explicit - mapping operations to HW is efficient
 - ▶ Makes use of local memory and shared memory explicit
 - ▶ More control on mapping & higher power efficiency

THE GRAPHICS PROCESSING UNIT (GPU)

- ▶ Originally designed to accelerate graphics
- ▶ Extension to the processing capabilities helps accelerate general purpose programs - General-Purpose GPU (GPGPU)
- ▶ Processing Flow:
 1. Copy input data from CPU memory to GPU memory
 2. Load GPU code and execute it
 3. Copy results from GPU memory to CPU memory



GPU PROGRAMMING USING CUDA

- ▶ **CUDA** - Compute Unified Device Architecture
 - ▶ Computing platform, API & language
- ▶ `__device__` or `__global__` indicates that the function is executed on the GPU
- ▶ `__host__` indicates that the function is executed on the system processor
- ▶ Function call syntax extended to `funcName <<dimGrid, dimBlock>>> (... parameter list...)`
 - ▶ `dimGrid` specifies the dimensions of the code in *Thread Blocks* - a group of threads blocked together
 - ▶ `dimBlock` - dimensions of a block in threads
- ▶ Identifier for blocks – `blockIdx`
- ▶ Identifier for thread in a block - `threadIdx`

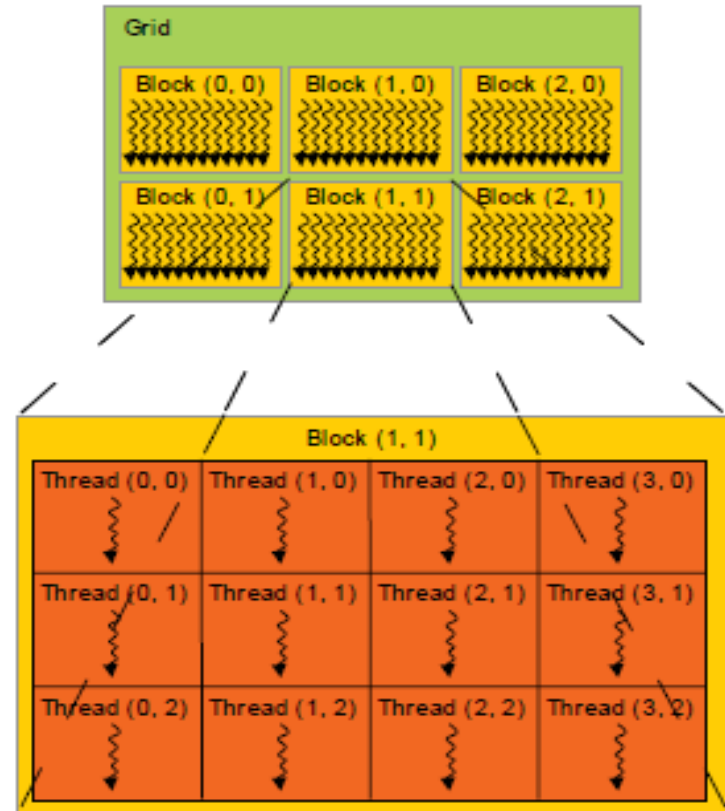
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

```
// 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];
}
```

GPU PROGRAMMING USING CUDA

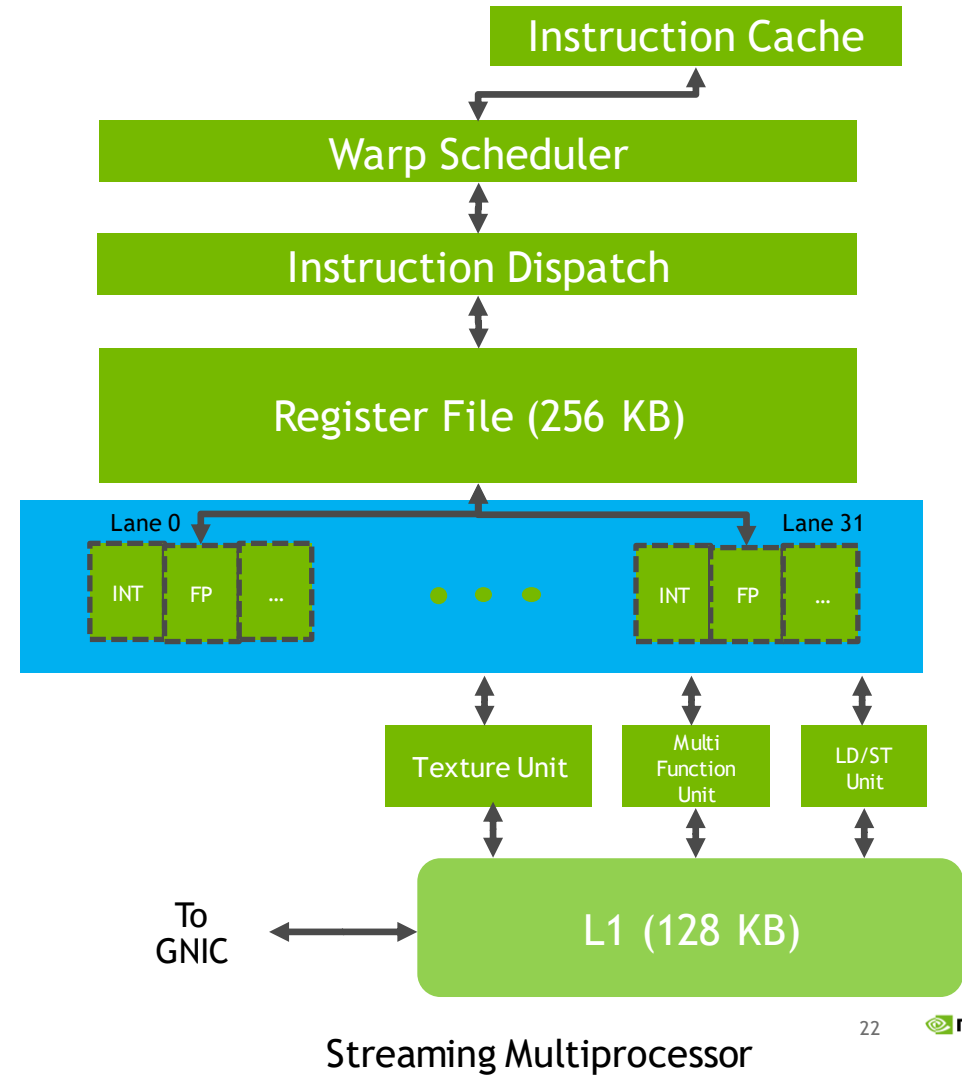
KEY OBSERVATIONS

- ▶ Programmer expresses the parallelism in CUDA explicitly
 - ▶ **Kernel** - portion of an application that is executed many times, but independently on different data
 - ▶ Kernels launch a grid of thread **blocks**
 - ▶ Threads with a block cooperate via shared memory
 - ▶ Threads within a block can synchronize using an intrinsic function `__syncthreads`
 - ▶ Threads in different blocks cannot cooperate
- ▶ Parallel execution and threads are managed by the hardware - no OS intervention



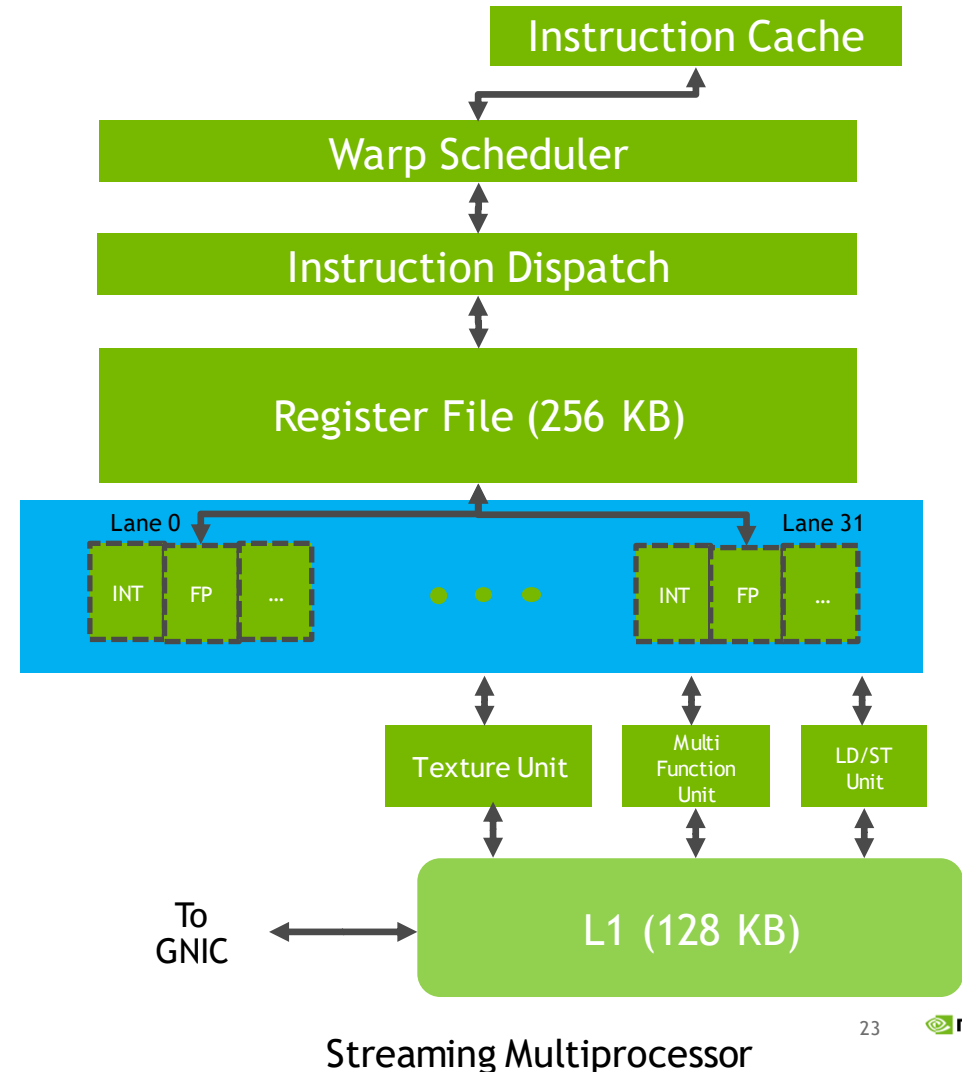
GPU COMPUTATIONAL STRUCTURES

- ▶ GPU: multiprocessor composed of multithreaded SIMD Processors
 - ▶ Each such SIMD processor is called the Streaming Multiprocessor (SM)
 - ▶ SIMT: Single-Instruction Multi-Thread processor, executes SIMD threads
 - ▶ SIMT provides easy single-thread scalar programming with SIMD efficiency
 - ▶ Warp: Group of 32 threads executing SIMD instructions
 - ▶ Hardware implements zero-overhead warp and thread scheduling



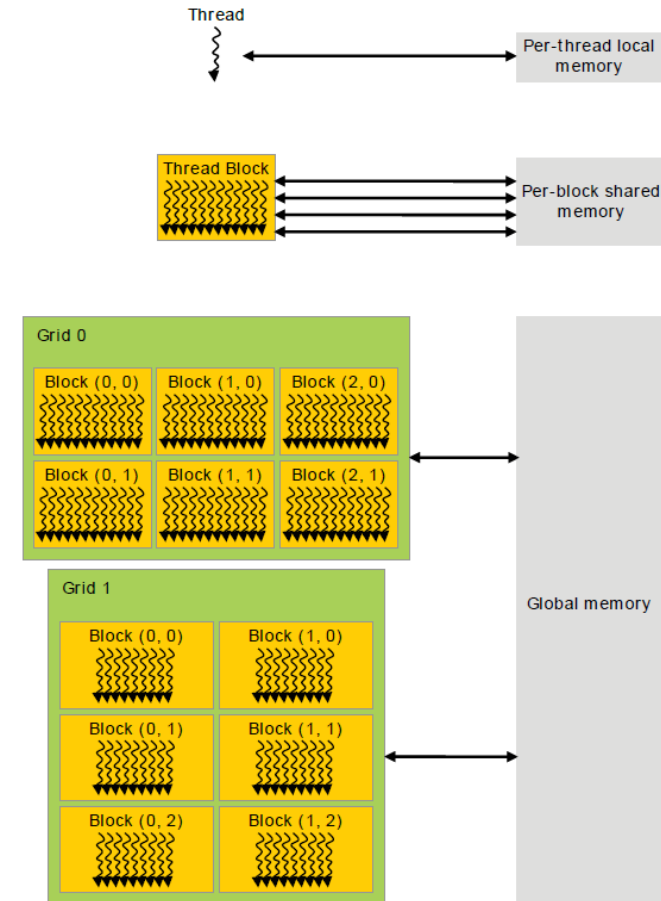
GPU COMPUTATIONAL STRUCTURES

- ▶ Warps are launched only if they have enough hardware resources
 - ▶ Registers
 - ▶ Shared memory
- ▶ Parallel functional units to execute SIMD operations - SIMD lanes
 - ▶ Floating point/integer units
- ▶ Functional units shared among SIMD processors
 - ▶ Texture Unit
 - ▶ Load/Store Units



MEMORY HIERARCHY


- ▶ Each thread uses private local memory
- ▶ Each thread block has access to shared memory
 - same lifetime as the block
 - ▶ Typically takes 10s of processor cycles
- ▶ All threads have access to global memory
 - ▶ Memory access on a cache miss can take excess of 2500 processor cycles
- ▶ Additional read-only global memory spaces
 - ▶ Constant memory
 - ▶ Texture memory



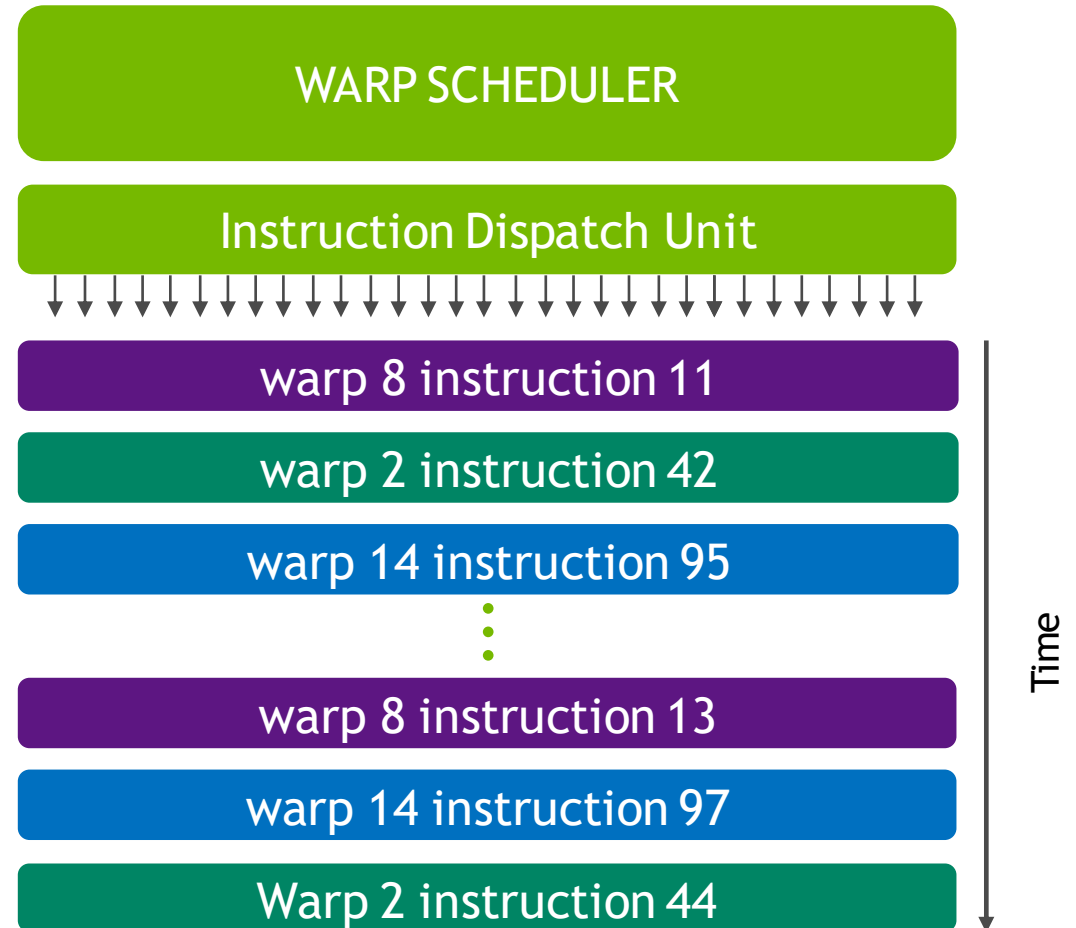
WARP SCHEDULER

- ▶ Warp scheduler picks eligible warps to issue instructions from
- ▶ GPU applications have many threads of SIMD instructions
 - ▶ multithreading hides latency of operations behind useful work done in other warps
 - ▶ Example: Memory dependencies can cause instruction issue stalls

```
ld r0, [addr1]  
ld r1, [addr2]  
add r2, r1, r0
```



- ▶ Warp scheduler will switch warps, thereby hiding latency of operations behind useful work



THE VOLTA/TURING SM

Four Processing Blocks each with:

- ❑ 16 FP32 Cores
- ❑ 16 INT32 Cores
- ❑ 2 Tensor Cores
- ❑ 1 Texture Unit
- ❑ 64 KB register File
- ❑ L0 instruction cache
- ❑ One Warp Scheduler
- ❑ One Dispatch Unit
- ❑ Combined 96 KB L1/Shared memory cache
- ❑ 1 RT Core/SM (Turing)



GPU INSTRUCTION SET ARCHITECTURE

The Parallel Thread Execution ISA (PTX ISA)

- ▶ Format: opcode.type d, a, b, c;
 - ▶ d – destination operand
 - ▶ a, b and c - source operands
 - ▶ Type is one of the following:

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

```
// 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];
}
```

```
shl.u32 R8, blockIdx, 8 ; // Thread Block ID * Block size ;(256 or 28)
add.u32 R8, R8, threadIdx ; // R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3 ; // byte offset
ld.global.f64 RD0, [X+R8]; // RD0 = X[i]
ld.global.f64 RD2, [Y+R8]; // RD2 = 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])
st.global.f64 [Y+R8], // RD0; Y[i] = sum (X[i]*a + Y[i])
```

PTX code for one iteration of DAXPY

GPU INSTRUCTION SET ARCHITECTURE

The Parallel Thread Execution ISA (PTX ISA)

Table 2. Reserved Instruction Keywords

abs	div	or	sin	vavg2, vavg4
add	ex2	pmevent	slct	vmad
addc	exit	popc	sqr	vmax
and	fma	prefetch	st	vmax2, vmax4
atom	isspacep	prefetchu	sub	vmin
bar	ld	prmt	subc	vmin2, vmin4
bfe	ldu	rcp	suld	vote
bfi	lg2	red	suq	vset
bfind	mad	rem	sured	vset2, vset4
bra	mad24	ret	sust	vshl
brev	madc	rsqr	testp	vshr
brkpt	max	sad	tex	vsub
call	membar	selp	tld4	vsub2, vsub4
clz	min	set	trap	xor
cnot	mov	setp	txq	
copysign	mul	shf	vabsdiff	
cos	mul 24	shf1	vabsdiff2, vabsdiff4	
cvt	neg	shl	vadd	
cvta	not	shr	vadd2, vadd4	

- ▶ All instructions can be predicated by a 1-bit predicate
- ▶ All loads and stores are scatter-gather - each SIMD lane loads/stores at an independent address - HW optimizes the data transfer
- ▶ Special instructions like bar (barrier) & atom (atomic op) support efficient parallel processing

CONDITIONAL EXECUTION - DIVERGENCE

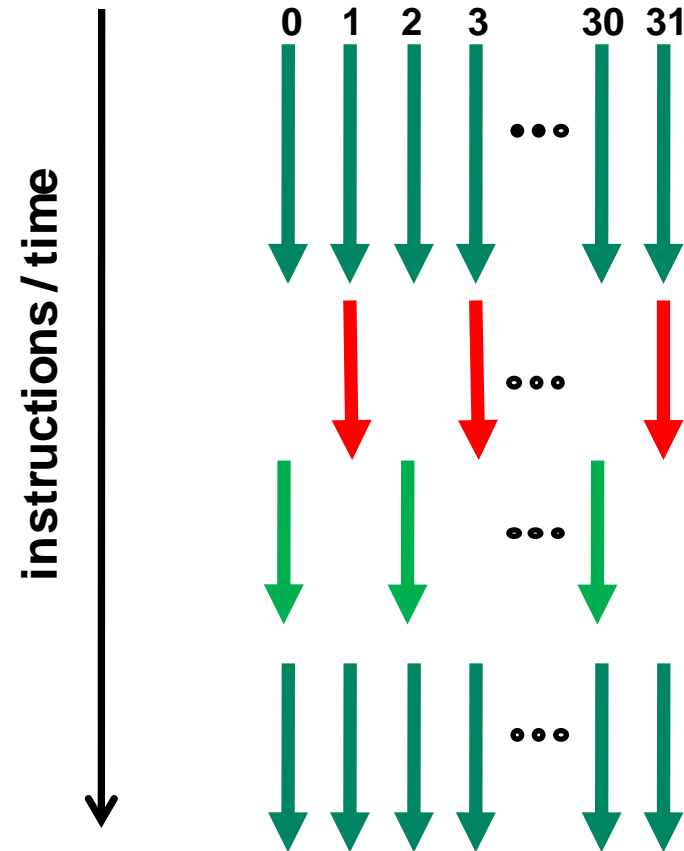
- ▶ In example on the right: Within a **warp**, threads with odd threadIdx.x execute $a[i] + b[i]$ while threads with even threadIdx.x execute $a[i] - b[i]$
 - ▶ Different threads execute different code – the threads in the warp diverge
 - ▶ SIMD efficiency = $\frac{\text{\#activeThreads}}{\text{\#threadsInWarp}}$ drops to $\frac{1}{2}$
- ▶ `__activemask()` returns the mask of all currently active threads in the calling warp

```
__global__ void add(int *a, int *b, int *c)
{
    int i = threadIdx.x;
    if (i % 2)
        c[i] = a[i] + b[i];
    else
        c[i] = a[i] - b[i];
}

main()
{
    ...
    add <<< 1, n >>> (a, b, c);
    ...
}
```

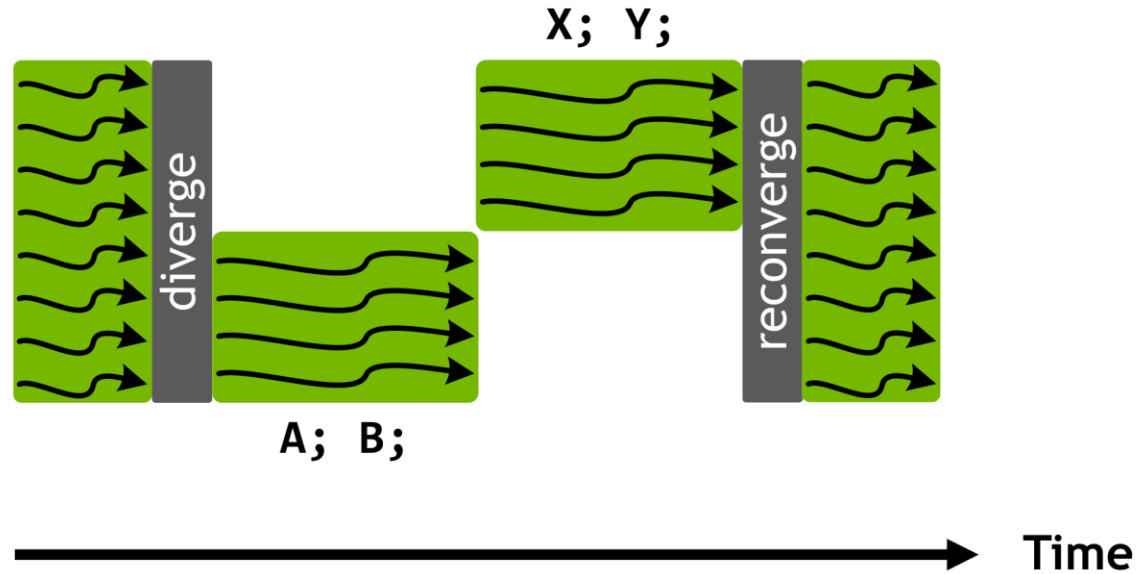
CONTROL DIVERGENCE

- ▶ Execution is serialized across the threads in the warp
- ▶ Threads can diverge
 - ▶ In some cases, the programming model can dictate whether code can diverge
- ▶ Best efficiency and performance is achieved when threads of a warp execute together
- ▶ Hardware and compiler work together to ensure re-convergence, wherever possible



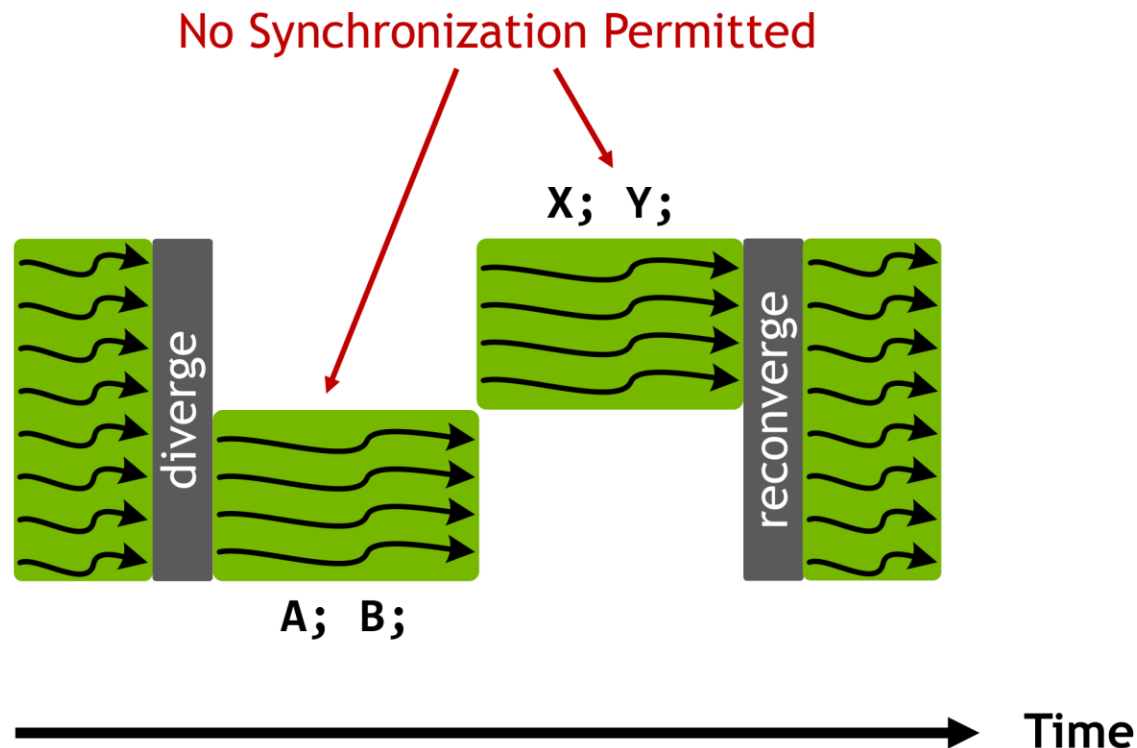
PASCAL WARP EXECUTION MODEL

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}
```



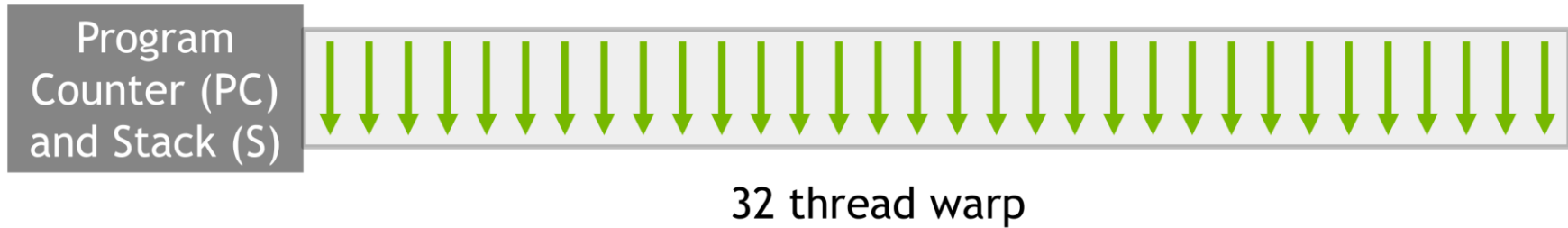
PASCAL WARP EXECUTION MODEL

```
if (threadIdx.x < 4) {  
    A;  
  
    B;  
} else {  
    X;  
  
    Y;  
}
```

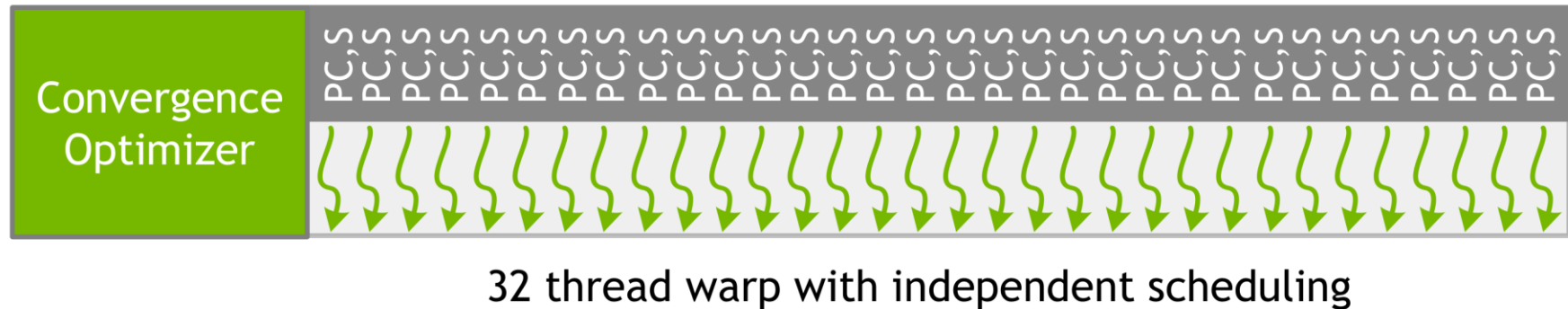


WARP IMPLEMENTATION

Pre-Volta



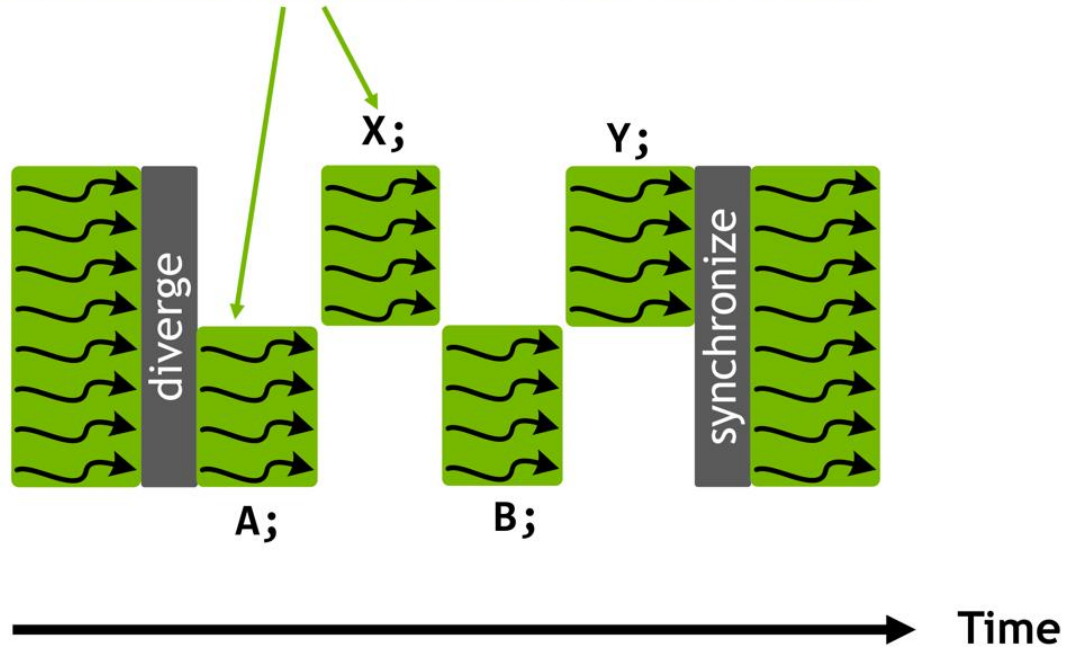
Volta



VOLTA+: WARP EXECUTION MODEL

Synchronization may lead to interleaved scheduling!

```
if (threadIdx.x < 4) {  
    A;  
    __syncwarp();  
    B;  
} else {  
    X;  
    __syncwarp();  
    Y;  
}  
__syncwarp();
```



GPU ARCHITECTURE

Delivering high performance per watt

- ▶ Task Level Parallelism - explicitly programmed as SIMT parallelism
 - ▶ Naturally uses SIMD parallelism wherever possible
- ▶ Fewer wasted instructions - Minimal speculation support, compiler extracts ILP
- ▶ Simultaneous Multithreading (SMT)* support - interleaves instructions from different warps to achieve high hardware utilization
- ▶ Domain specific functional units for high performance - e.g., TEX, MMA instructions
- ▶ Memory hierarchy built specially for efficiency in the domain of operation
- ▶ Scale up - Multiple cores in a GPU for horizontal scaling - more performance within a chip
- ▶ Scale out - Multiple GPUs work together on a high-speed link (NVLINK) for vertical scaling

Tullsen, D.M.; Eggers, S.J.; Levy, H.M. (1995). ["Simultaneous multithreading: Maximizing on-chip parallelism"*](#). 22nd Annual International Symposium on Computer Architecture. IEEE. pp. 392–403. [*ISBN 978-0-89791-698-1*](#).

GPU APPLICATION PERFORMANCE

- ▶ Speedups obtained for the whole application using CPU & GPU coprocessing over CPU alone as measured by application developers
- ▶ Speedups reported using GeForce 8800, Tesla T8, GeForce GTX 280, Tesla T10, and GeForce GTX 285
 - ▶ range from 9x to more than 130x
 - ▶ Higher speedups reflect applications where more of the work ran in parallel on the GPU
 - ▶ Lower speedups reflect applications limited by the code's CPU portion, coprocessing overhead, or by divergence in the code's GPU fraction

Table 3. Representative CUDA application coprocessing speedups.

Application	Field	Speedup
Two-electron repulsion integral ¹²	Quantum chemistry	130×
Gromacs ¹³	Molecular dynamics	137×
Lattice Boltzmann ¹⁴	3D computational fluid dynamics (CFD)	100×
Euler solver ¹⁵	3D CFD	16×
Lattice quantum chromodynamics ¹⁶	Quantum physics	10×
Multigrid finite element method and partial differential equation solver ¹⁷	Finite element analysis	27×
N-body physics ¹⁸	Astrophysics	100×
Protein multiple sequence alignment ¹⁹	Bioinformatics	36×
Image contour detection ²⁰	Computer vision	130×
Portable media converter*	Consumer video	20×
Large vocabulary speech recognition ²¹	Human interaction	9×
Iterative image reconstruction ²²	Computed tomography	130×
Matlab accelerator**	Computational modeling	100×

* Elemental Technologies, Badaboom media converter, 2009; <http://badaboomit.com>.

** Accelereyes, Jacket GPU engine for Matlab, 2009; <http://www.accelereyes.com>.

KEY TAKEAWAYS

- ▶ The end of Moore's law and Dennard scaling has opened new challenges in computer architecture
 - ▶ Domain specific accelerators like the GPUs use novel techniques and application parallelism to accelerate applications
- ▶ The GPU architecture is targeted to a parallel workload
 - ▶ Best suited for applications with SIMD parallelism, though hybrid workloads can be handled
- ▶ Extracting the best possible performance for parallel code written in CUDA C++ requires a good understanding of the architecture and performance tradeoffs
 - ▶ Mechanisms for thread synchronization provides a general model for multithreaded execution
 - ▶ Special instructions exposed as intrinsics (shfl.sync - enables exchange of register data within threads of a warp) provide hooks to efficient execution
 - ▶ Correctness of the program in terms of exposed parallelism is the programmer's responsibility



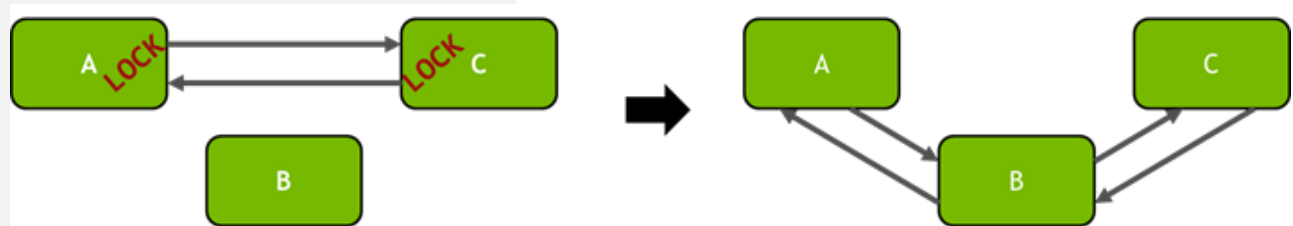
VOLTA+: WARP EXECUTION MODEL

```
__device__ void insert_after(Node *a, Node *b)
{
    Node *c;
    lock(a); lock(a->next);
    c = a->next;

    a->next = b;
    b->prev = a;

    b->next = c;
    c->prev = b;

    unlock(c); unlock(a);
}
```



Starvation-Free Algorithms: Software synchronization also supported, e.g. locks for doubly linked lists!

How Feature Size Affects Speed

■ Robert Dennard's Scaling Law (1974)

- If scale the physical parameters of an integrated circuit equally by factor K, then performance parameters scale as follows:

Geometry & Supply voltage	L_g, W_g T_{ox}, V_{dd}	K	Scaling K : K=0.7 for example	Device or Circuit Parameter	Scaling Factor
				Device dimension t_{ox}, L, W	K
				Doping concentration N_a	1/K
				Voltage V	K
				Current I	K
				Capacitance eA/t	K
				Delay time per circuit VC/I	K
				Power dissipation per circuit VI	K ²
				Power density V/I A	1
Drive current in saturation	I_d	K	$I_d = v_{sat} W_g C_o (V_g - V_{th})$ $\rightarrow W_g (t_{ox}^{-1})(V_g - V_{th}) = W_g t_{ox}^{-1} (V_g - V_{th}) = KK^{-1}K = K$	C_o : gate C per unit area	
I_d per unit W_g	$I_d / \mu m$	1	I_d per unit $W_g = I_d / W_g = 1$		
Gate capacitance	C_g	K	$C_g = \epsilon_o \epsilon_{ox} L_g W_g / t_{ox} \rightarrow KK/K = K$		
Switching speed	τ	K	$\tau = C_g V_{dd} / I_d \rightarrow KK/K = K$		
Clock frequency	f	1/K	$f = 1/\tau = 1/K$		
Chip area	A_{chip}	α	α : Scaling factor \rightarrow In the past, $\alpha > 1$ for most cases		
Integration (# of Tr)	N	α/K^2	$N \rightarrow \alpha/K^2 = 1/K^2$, when $\alpha=1$		
Power per chip	P	α	$fNCV^2/2 \rightarrow K^{-1}(\alpha K^{-2})K(K^1)^2 = \alpha = 1$, when $\alpha=1$		

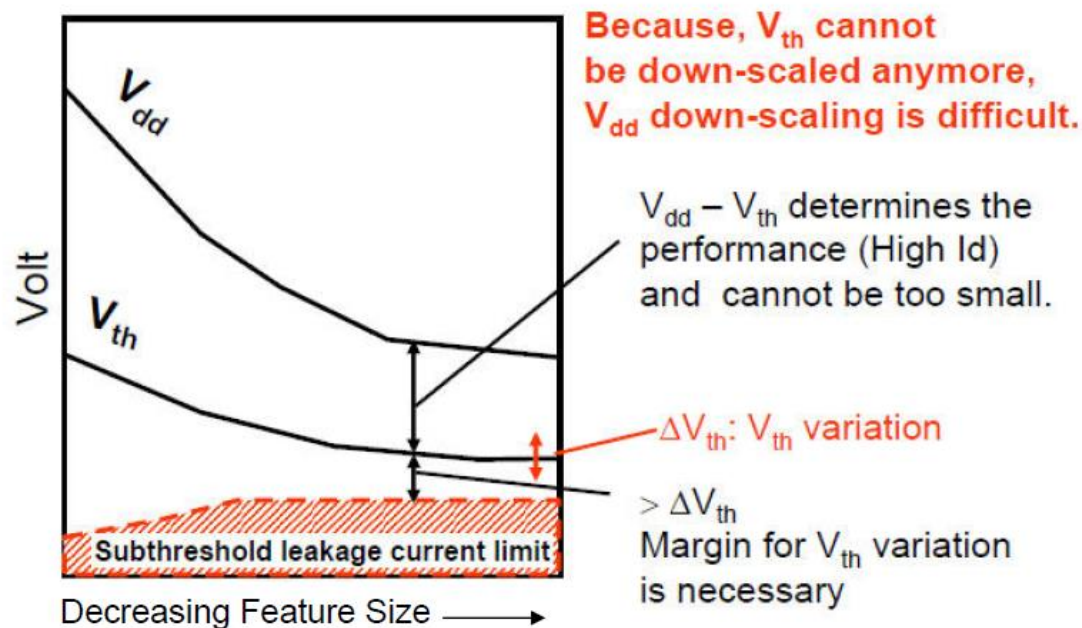
Table Courtesy of Mark Bohr, IBM

Slide Courtesy of Hirosi Iwai, Tokyo Institute of Technology

How Feature Size Affects Speed (Cont.)

■ Examining Speed versus Technology (Cont.)

- V_{dd} scales with K , so V_{dd} decreases with smaller feature size
 - ◆ Difficulty in Down-scaling of Supply Voltage: V_{dd}



Slide Courtesy of
Hiroshi Iwai,
Tokyo Institute of Technology