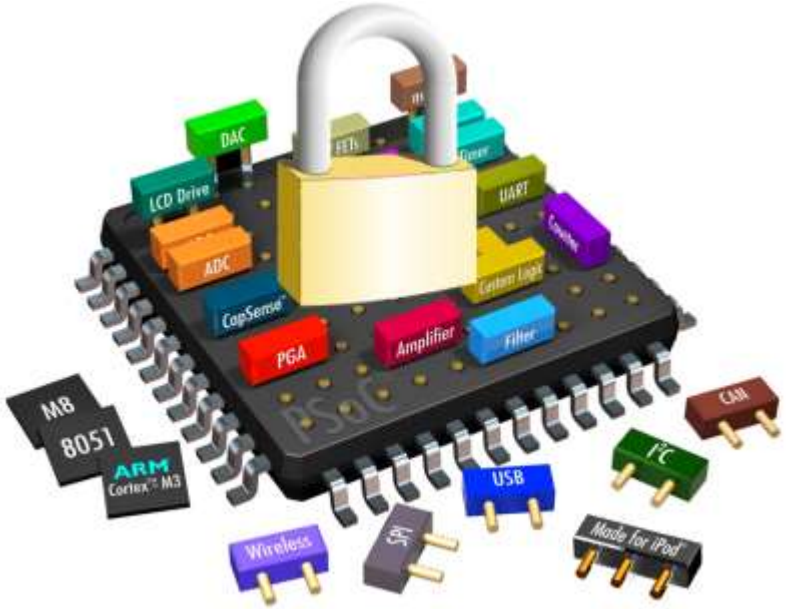
Advanced Computer Architecture

Data-Level Parallelism in Vector, SIMD, and GPU Architectures



Three Variations of SIMD Architecture

1

Vector Architectures

The oldest variation (by 30+ years), essentially pipelined execution of many data operations. Easier to understand and compile to than other SIMD variations, but historically considered too expensive for microprocessors until recently.

2

Multimedia SIMD Extensions

Found in most instruction set architectures today supporting multimedia applications. For x86, these include MMX (1996), followed by SSE versions, and continuing with AVX. Often necessary for achieving highest computation rates.

3

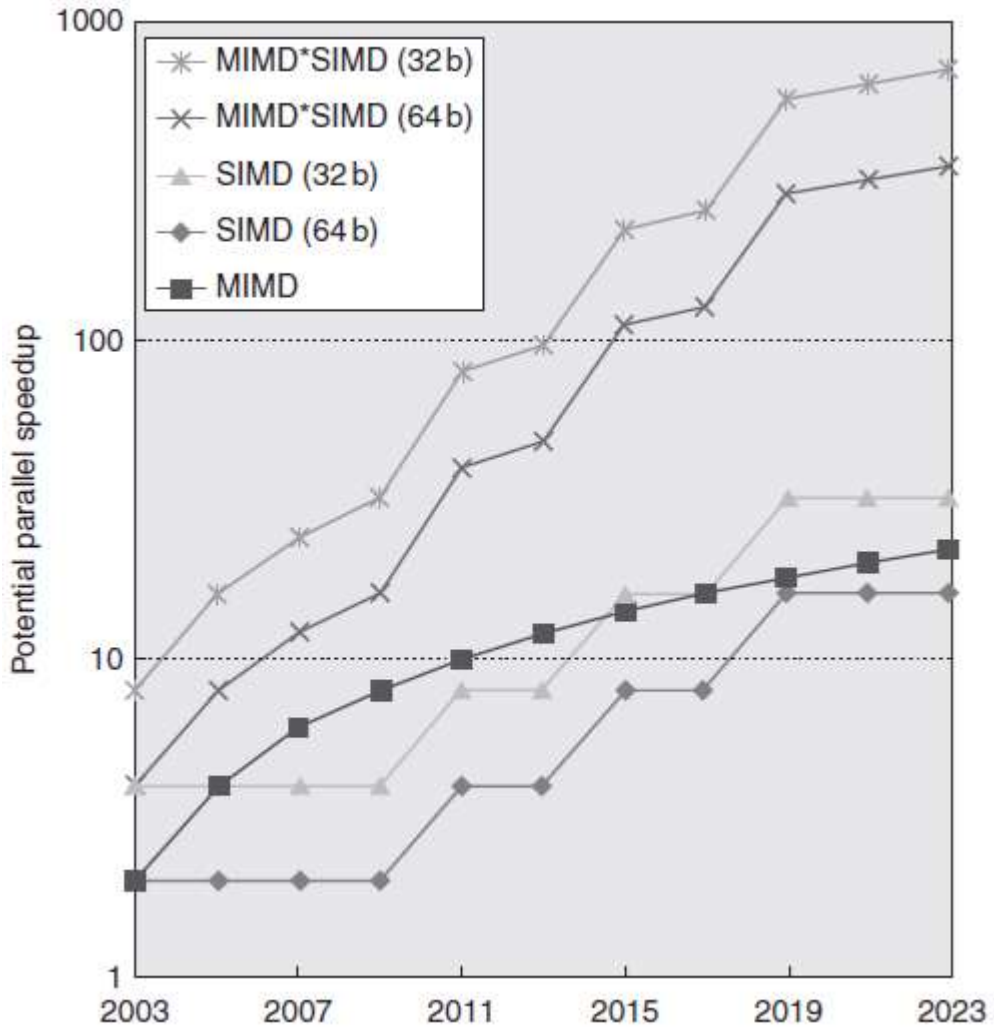
Graphics Processing Units (GPUs)

Offers higher potential performance than traditional multicore computers. While sharing features with vector architectures, GPUs have distinguishing characteristics due to their evolution in a heterogeneous ecosystem with system processors and separate memory.

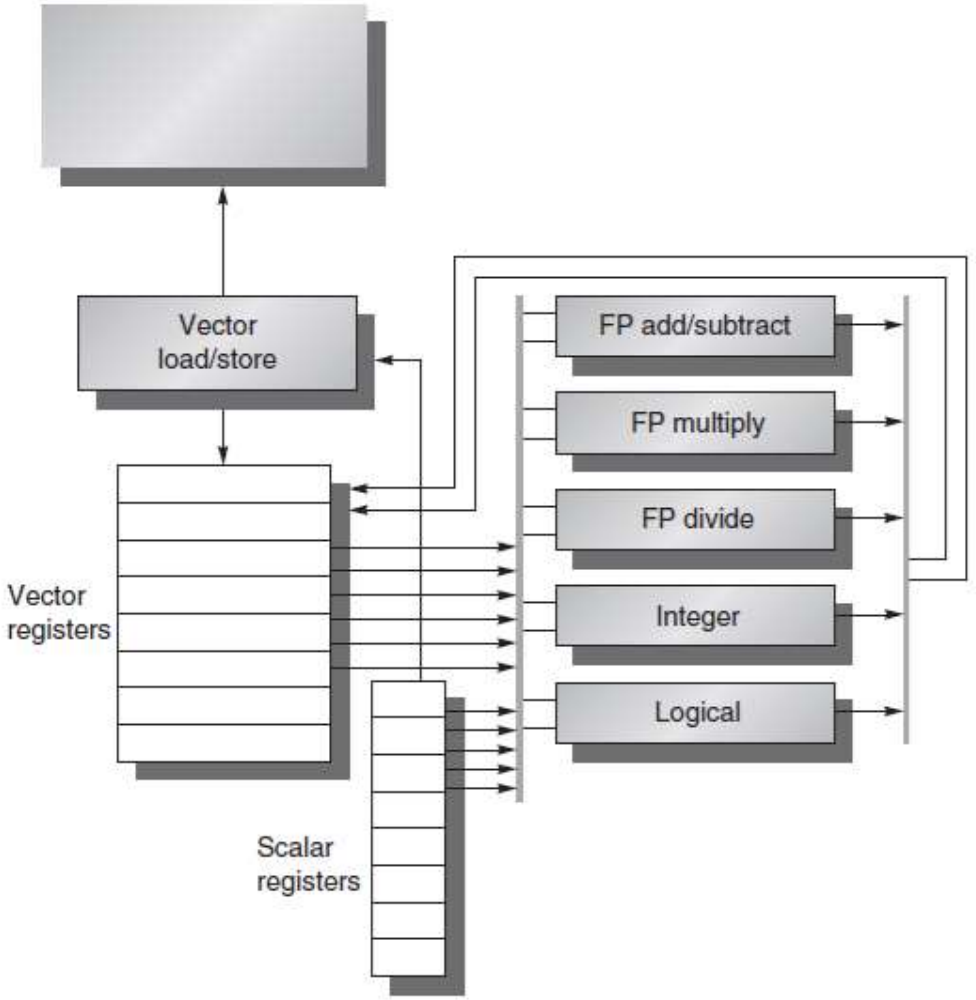
For problems with significant data parallelism, all three SIMD variations are easier to program than classic parallel MIMD programming.

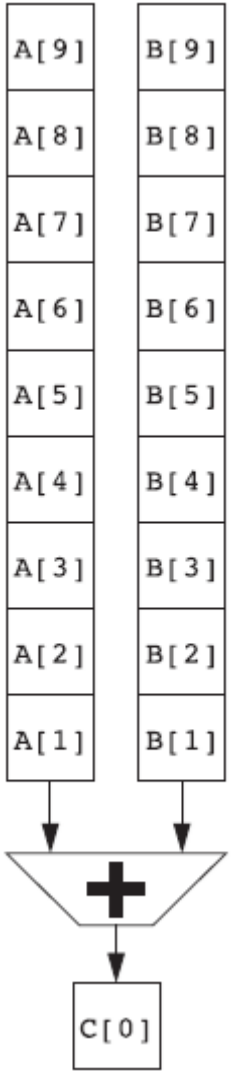
SIMD vs MIMD

- This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.
- For applications with both data-level parallelism and thread-level parallelism, the potential speedup in 2020 will be an order of magnitude higher than today, making understanding SIMD parallelism at least as important as MIMD parallelism.



VMIPS


Vector
load/store
FP add/subtract
FP multiply
FP divide
Integer
Logical
Vector
registers
Scalar
registers


A[9]
A[8]
A[7]
A[6]
A[5]
A[4]
A[3]
A[2]
A[1]
B[9]
B[8]
B[7]
B[6]
B[5]
B[4]
B[3]
B[2]
B[1]
C[0]

VMIPS ISA

- **Vector-Vector (VV):** Operations between two vector registers
- **Vector-Scalar (VS):** Operations between a vector register and a scalar value
- **Memory operations:** Load/store entire vectors or with stride/indexed addressing

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

DAXPY Example

DAXPY: Y = a × X + Y

MIPS (Scalar)

```
        L.D     F0,a          ;load scalar a
        DADDIU  R4,Rx,#512    ;last address to load
Loop:   L.D     F2,0(Rx)      ;load X[i]
        MUL.D   F2,F2,F0      ;a × X[i]
        L.D     F4,0(Ry)      ;load Y[i]
        ADD.D   F4,F4,F2      ;a × X[i] + Y[i]
        S.D     F4,9(Ry)      ;store into Y[i]
        DADDIU  Rx,Rx,#8      ;increment index to X
        DADDIU  Ry,Ry,#8      ;increment index to Y
        DSUBU   R20,R4,Rx     ;compute bound
        BNEZ    R20,Loop      ;check if done
```

VMIPS (Vector)

```
L.D     F0,a        ;load scalar a
LV      V1,Rx       ;load vector X
MULVS.D V2,V1,F0    ;vector-scalar multiply
LV      V3,Ry       ;load vector Y
ADDVV.D V4,V2,V3    ;add vectors
SV      V4,Ry       ;store the result
```

Vector code: 6 instructions vs. ~600 for scalar (for 64 elements)

Pipeline stalls occur once per vector instruction rather than once per element

Vector Execution Time

- Convoy approximation
  - **Convoy**: a set of vector instructions that can execute together without structural hazards (resource constraints, dependency, etc.)
  - **Chime**: the approximate time to execute one convoy
- Chaining
  - Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available
  - The results from the first functional unit in the chain are “forwarded” to the second functional unit

Example

```
L.D     F0,a        ;load scalar a
LV      V1,Rx       ;load vector X
MULVS.D V2,V1,F0    ;vector-scalar multiply
LV      V3,Ry       ;load vector Y
ADDVV.D V4,V2,V3    ;add vectors
SV      V4,Ry       ;store the result
```

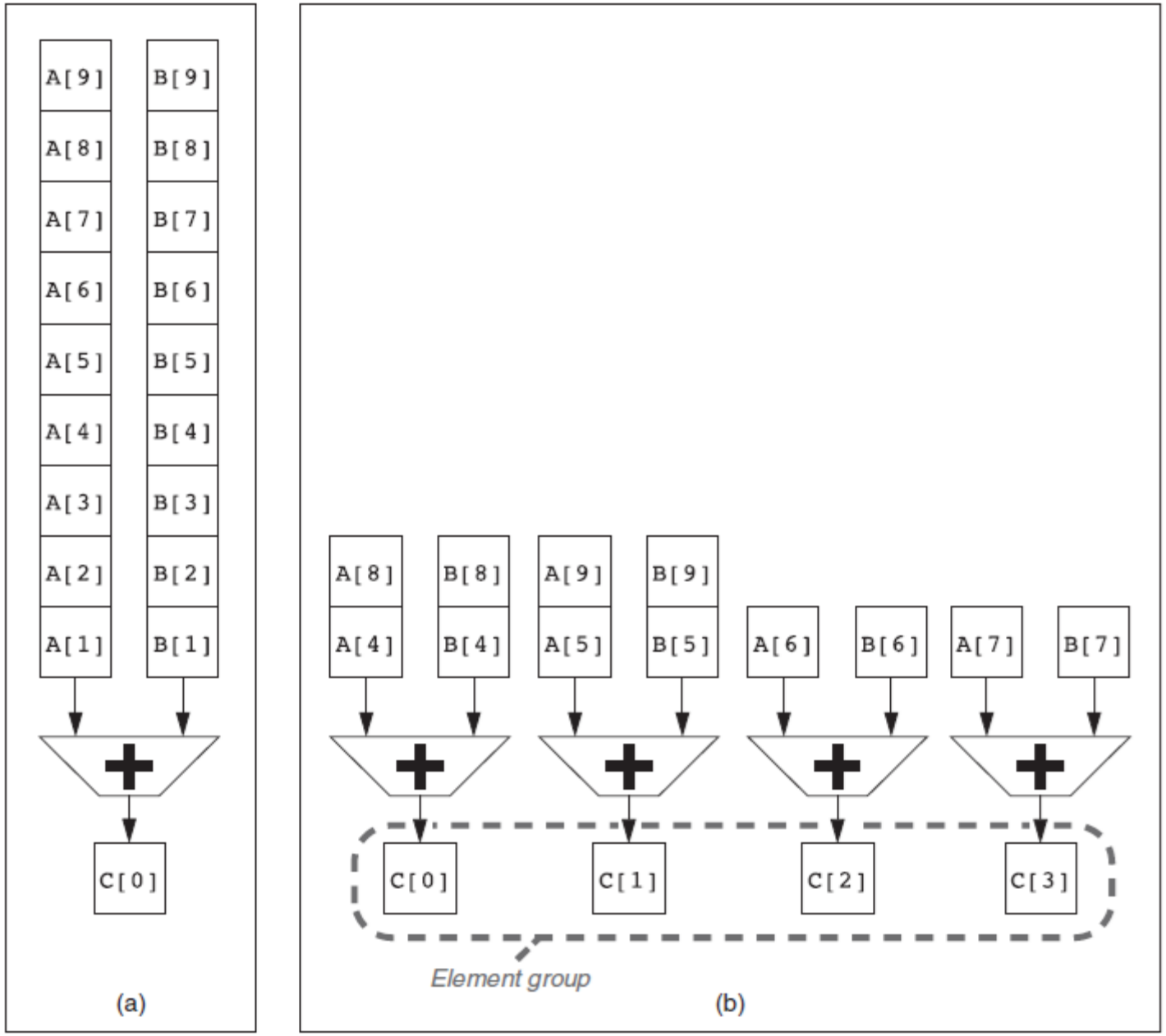
- 3 Convoys (single load/store)
  - Convoy 1: LV, MULVS.D (by chaining)
  - Convoy 2: LV, ADDVV.D (by chaining)
  - Convoy 3: SV
- It takes 3 chimes
- For instance, with 64-element vectors, it takes 3 * 64 = 192 cycles

Further Optimizations

- How can a vector processor execute a single vector faster than one element per clock cycle? Multiple elements per clock cycle improve performance.
- How does a vector processor handle programs where the vector lengths are not the same as the length of the vector register (64 for VMIPS)? Since most application vectors don't match the architecture vector length, we need an efficient solution to this common case.
- What happens when there is an IF statement inside the code to be vectorized? More code can vectorize if we can efficiently handle conditional statements.
- What does a vector processor need from the memory system? Without sufficient memory bandwidth, vector execution can be futile.
- How does a vector processor handle multiple dimensional matrices? This popular data structure must vectorize for vector architectures to do well.
- How does a vector processor handle sparse matrices? This popular data structure must vectorize also.
- How do you program a vector computer? Architectural innovations that are a mismatch to compiler technology may not get widespread use.

Multiple Lanes

- Each lane processes a subset of vector elements in parallel
- Four lanes reduce execution time by 4× compared to a single lane
- Requires minimal increase in control complexity
- Allows trading off area, clock rate, voltage, and energy without sacrificing peak performance



Vector-Length Registers

```
for (i=0; i <n; i=i+1)
  Y[i] = a * X[i] + Y[i];
```

- The value of n may be unknown at compile-time and not a multiple of the architectural vector length
- Vector-length registers (VLR) solve the problem of vector lengths not matching the architectural vector length

```
low = 0;
VL = (n % MVL);
for (j = 0; j <= (n/MVL); j=j+1) {
  for (i = low; i < (low+VL); i=i+1)
    Y[i] = a * X[i] + Y[i] ;
  low = low + VL;
  VL = MVL;
}
```

- Strip mining
  - Generation of code such that each vector operation is done for a size less than or equal to the maximum vector length (MVL)

Vector Mask Registers

```
for (i = 0; i < 64; i=i+1)
  if (X[i] != 0)
    X[i] = X[i] – Y[i];
```

Vector Mask Enabled

- This loop cannot normally be vectorized because of the conditional execution of the body
- If the inner loop could be run for the iterations for which X[i] ≠ 0, then the subtraction could be vectorized.

```
LV        V1,Rx        ;load vector X into V1
LV        V2,Ry        ;load vector Y
L.D       F0,#0        ;load FP zero into F0
SNEVS.D   V1,F0        ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D   V1,V1,V2     ;subtract under vector mask
SV        V1,Rx        ;store the result in X
```

Memory System for Vector Processors

1

Memory Banks

Multiple independent memory banks supply bandwidth for vector loads/stores

- Support multiple simultaneous accesses
- Allow non-sequential memory access patterns
- Support multiple processors sharing memory

2

Bank Conflicts

Occur when multiple accesses target the same bank

- Can significantly reduce effective memory bandwidth
- Worst case: stride is a multiple of number of banks
- Best case: unit stride with no conflicts

Example: Cray T932 required 1344 memory banks to support 32 processors with 6 references per processor!

Stride: Handling Multidimensional Arrays

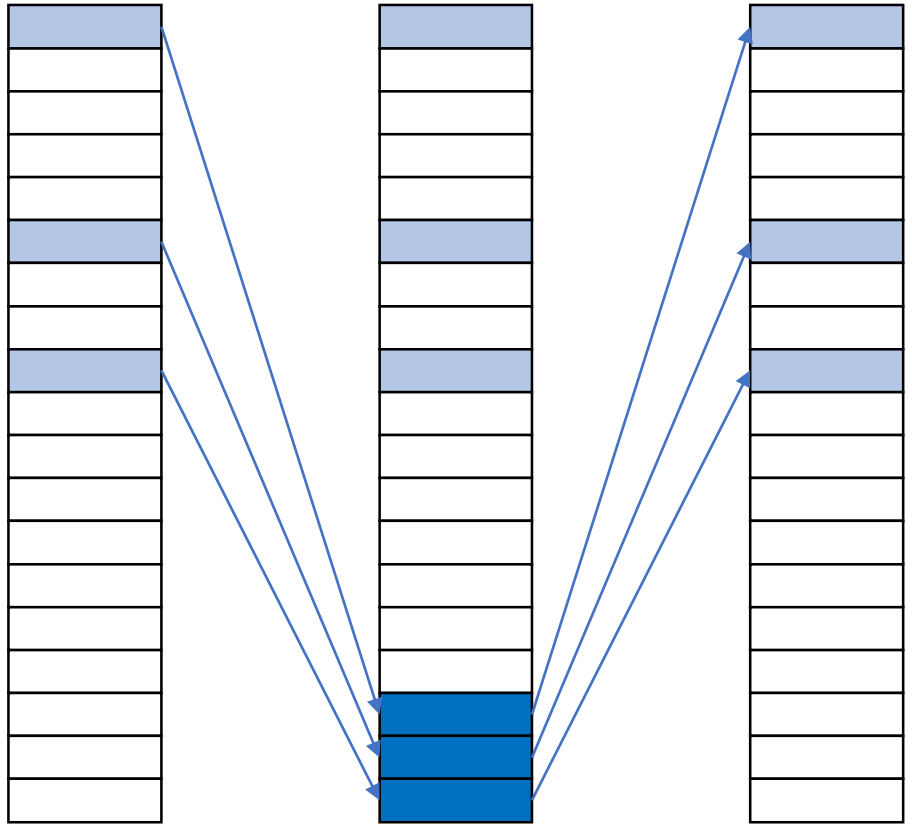
```
for (i = 0; i < 100; i=i+1)
  for (j = 0; j < 100; j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
  }
```

- In row-major order (as in C):
  - B has stride of 1 (adjacent elements)
  - D has stride of 100 (elements separated by row size)
- Vector processors support non-unit strides:
  - LVWS (load vector with stride)
  - SVWS (store vector with stride)

Gather-Scatter: Handling Sparse Matrices

```
for (i = 0; i < n; i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
```

```
LV      Vk, Rk       ;load K
LVI     Va, (Ra+Vk)  ;load A[K[]]
LV      Vm, Rm       ;load M
LVI     Vc, (Rc+Vm)  ;load C[M[]]
ADDVV.D Va, Va, Vc   ;add them
SVI     (Ra+Vk), Va  ;store A[K[]]
```



Programming Vector Architectures

- Compilers can identify vectorizable code at compile time
- Programmers receive feedback on why code didn't vectorize
- Programmers can provide hints when dependencies are safe to ignore
- Clear performance model helps optimize code

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Single Instruction Multiple Data (SIMD)

- Motivation
  - SIMD Multimedia Extensions started with the simple observation that many media applications operate on narrower data types than the 32-bit processors were optimized for
  - Many graphics systems used 8 bits to represent each of the three primary colors plus 8 bits for transparency
  - Audio samples are usually represented with 8 or 16 bits
- The additional cost of such partitioned adders was small

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

Example SIMD extensions for 256-bit-wide operations

SIMD vs. Vector Architectures

Fixed vs. Variable Length

SIMD extensions fix the number of data operands in the opcode, leading to hundreds of instructions in x86 extensions. Vector architectures use a vector length register to specify operand count.

Limited Addressing Modes

SIMD lacks sophisticated addressing modes of vector architectures (strided accesses and gather-scatter), limiting compiler vectorization capabilities.

Conditional Execution

SIMD usually doesn't offer mask registers to support conditional execution of elements, unlike vector architectures.

These limitations make it harder for compilers to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

Advantages

- Despite of their weaknesses, the SIMD extensions is gaining popularity because
  - They cost little to add to the standard arithmetic unit and they were easy to implement
  - They require little extra state compared to vector architectures, which is always a concern for context switch times
  - You need a lot of memory bandwidth to support a vector architecture, which many computers don't have
  - SIMD does not have to deal with problems in virtual memory when a single instruction that can generate 64 memory accesses can get a page fault in the middle of the vector. SIMD extensions use separate data transfers per SIMD group of operands that are aligned in memory, and so they cannot cross page boundaries.

Evolution of x86 SIMD Extensions

1

MMX (1996)

Repurposed 64-bit floating-point registers for 8×8-bit or 4×16-bit operations. Added parallel MAX/MIN operations and DSP-like instructions.

2

SSE (1999)

Added separate 128-bit registers for 16×8-bit, 8×16-bit, or 4×32-bit operations. Added single-precision floating-point arithmetic.

3

SSE2-SSE4 (2001-2007)

Added double-precision SIMD floating-point and specialized multimedia instructions.

4

AVX (2010)

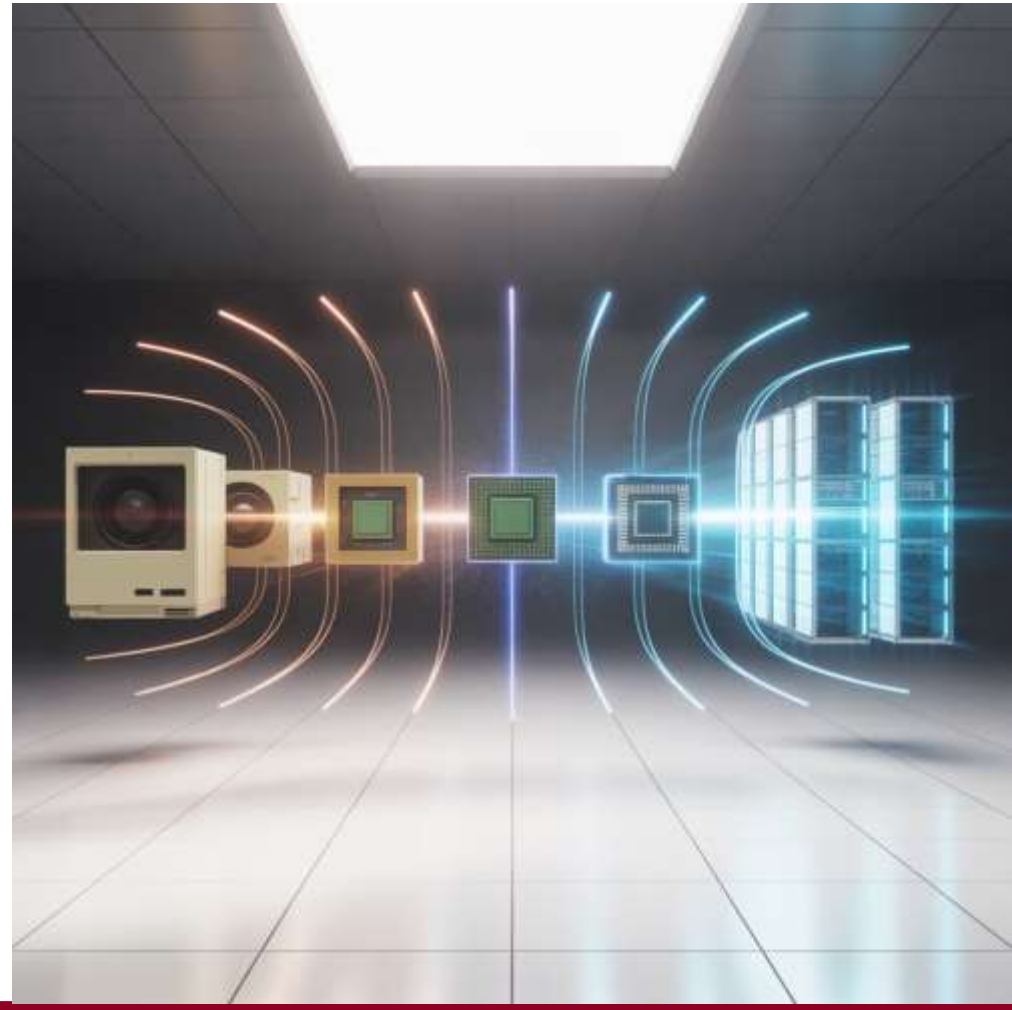
Doubled register width to 256 bits, doubling operations on all data types. Prepared for future 512-bit and 1024-bit extensions.

Example

```
        L.D     F0,a          ;load scalar a
        MOV     F1, F0        ;copy a into F1 for SIMD MUL
        MOV     F2, F0        ;copy a into F2 for SIMD MUL
        MOV     F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU  R4,Rx,#512    ;last address to load
Loop:   L.4D    F4,0(Rx)      ;load X[i], X[i+1], X[i+2], X[i+3] to F4, F5, F6 and F7
        MUL.4D  F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D    F8,0(Ry)      ;load Y[i], Y[i+1], Y[i+2], Y[i+3] to F8, F9, F10 and F11
        ADD.4D  F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D    F8,0(Rx)      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU  Rx,Rx,#32     ;increment index to X
        DADDIU  Ry,Ry,#32     ;increment index to Y
        DSUBU   R20,R4,Rx     ;compute bound
        BNEZ    R20,Loop      ;check if done
```

GPU Origins and Evolution

- GPUs and CPUs do not share a common architectural ancestor. While CPUs evolved from general-purpose processors, GPUs descended from specialized graphics accelerators.
- The primary purpose of GPUs remains graphics excellence, but they have evolved to support a wider range of computational applications.



GPU Computing Revolution

- GPU computing
  - The explosion of interest in GPU computing occurred when this hardware potential was combined with programming languages that made GPUs easier to program.
- Programming model
  - CUDA (Compute Unified Device Architecture) is NVIDIA's C-like language and programming environment designed to improve GPU programmer productivity by addressing both heterogeneous computing challenges and multifaceted parallelism

CUDA Programming Challenges

Heterogeneous Computing

Coordinating computation between the system processor (host) and the GPU (device)

Data Transfer Management

Efficiently moving data between system memory and GPU memory

Multiple Forms of Parallelism

Handling multithreading, MIMD, SIMD, and instruction-level parallelism simultaneously

CUDA produces C/C++ for the system processor (host) and a C/C++ dialect for the GPU (device). A similar vendor-independent language is OpenCL, which supports multiple platforms.

CUDA Thread: The Unifying Concept

- NVIDIA chose the CUDA Thread as the unifying theme for all forms of parallelism. Using this lowest level of parallelism as the programming primitive, the compiler and hardware can organize thousands of CUDA Threads to utilize various styles of parallelism within a GPU.
  - Multithreading
  - MIMD (Multiple Instruction, Multiple Data)
  - SIMD (Single Instruction, Multiple Data)
  - Instruction-level parallelism
- NVIDIA classifies this programming model as SIMT (Single Instruction, Multiple Thread)

CUDA Programming Basics

Key CUDA Elements

- Functions marked with `__device__` or `__global__` run on the GPU
- Functions marked with `__host__` run on the system processor
- Variables in `__device__` or `__global__` functions are allocated to GPU Memory
- Extended function call syntax:

```
name<<<dimGrid, dimBlock>>>(... parameter list ...)
```

Thread Organization

- `blockIdx`: identifier for blocks
- `threadIdx`: identifier for threads per block
- `dimGrid`: number of blocks being launched
- `dimBlock`: number of threads per block
- Threads are organized in blocks of 32 threads called Thread Blocks

DAXPY Example: C vs. CUDA

DAXPY in C

```
// 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];
}
```

DAXPY in CUDA

```
// 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
__device__
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 Terminology

- GPUs and CPUs do not go back in computer architecture genealogy to a common ancestor; there is no Missing Link that explains both
- The uncommon heritage mentioned above helps explain why GPUs have their own architectural style and their own terminology independent from CPUs

Program Abstractions

More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Textbook definition
Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.

Machine Object

More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Textbook definition
A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.

Processing Hardware

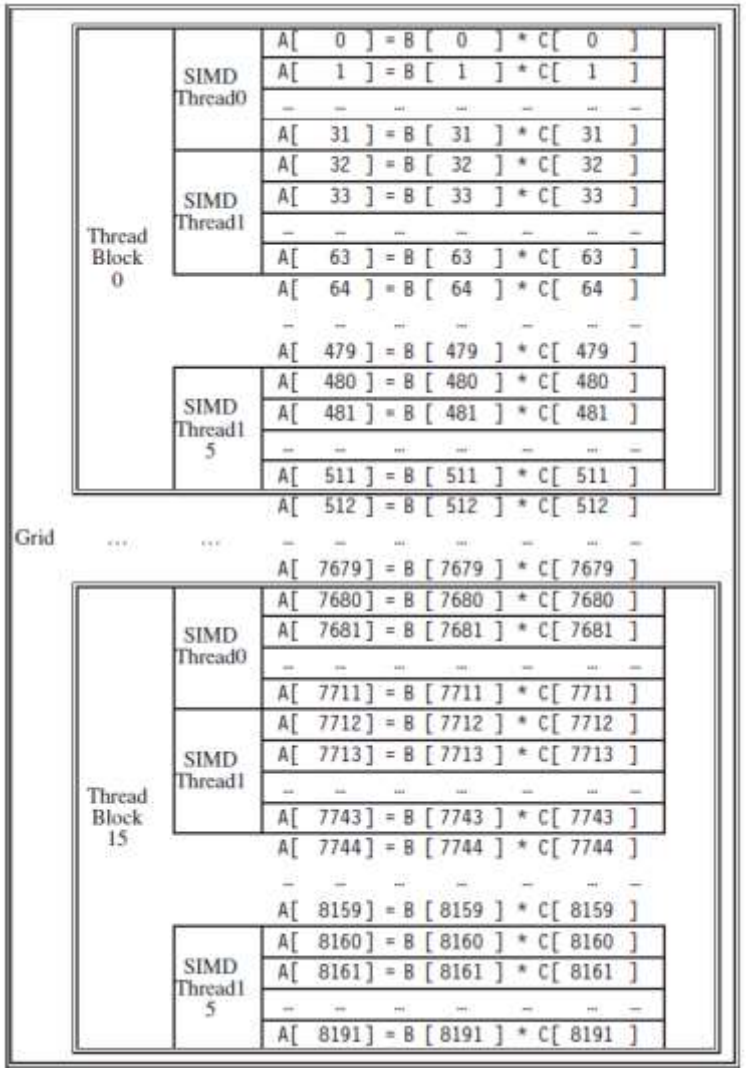
More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Textbook definition
Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.

Memory Hardware

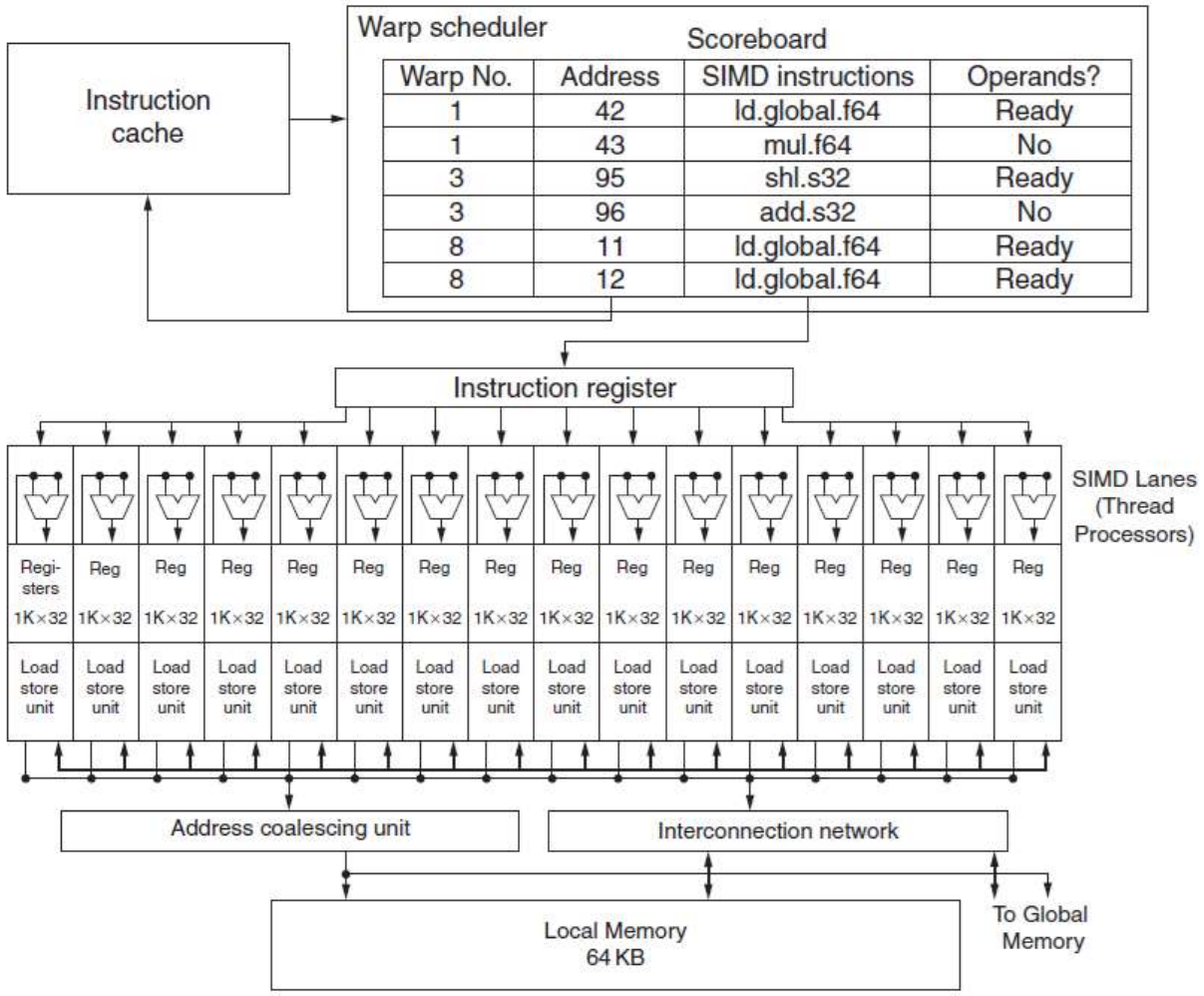
More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Textbook definition
GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Vector-Vector Multiply Example

- For a vector-vector multiply with 8192 elements:
  - The Grid (vectorizable loop) works on all 8192 elements
  - With 512 elements per Thread Block, we need 16 Thread Blocks
  - Each Thread Block contains 16 threads of SIMD instructions (SIMD Threads)
  - Each thread of SIMD instructions calculates 32 elements per instruction
- The Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the Thread Scheduler picks which thread of SIMD instructions to run each clock cycle.



Multithreaded SIMD Processor



- 16 SIMD lanes for parallel execution
- SIMD Thread (Warp) Scheduler with ~48 independent threads
- Scoreboard to track which threads are ready to run
- Dispatch unit to send threads to the processor

GPU Hardware Schedulers

1

Thread Block Scheduler

Assigns Thread Blocks (bodies of vectorized loops) to multithreaded SIMD Processors

Ensures thread blocks are assigned to processors whose local memories have the corresponding data

2

SIMD Thread Scheduler

Operates within a SIMD Processor

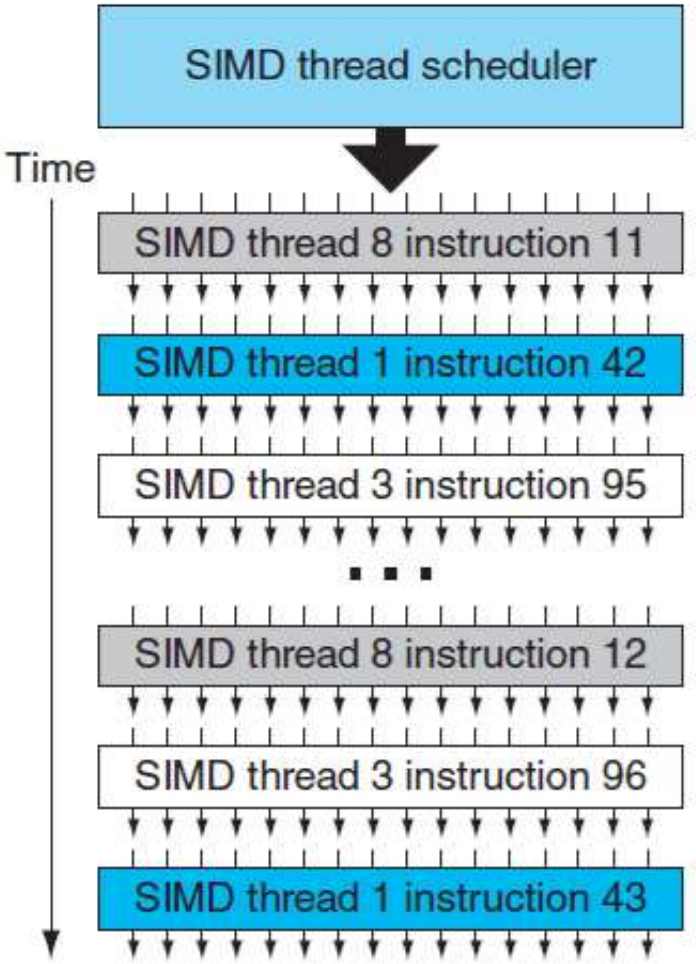
Schedules when threads of SIMD instructions should run

Includes a scoreboard to track up to 48 threads of SIMD instructions

These two levels of hardware schedulers work together to manage the execution of thousands of threads across multiple SIMD processors.

SIMD Thread Scheduling

- The SIMD Thread Scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing that thread.
- Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time, allowing it to hide memory latency and increase processor utilization.
- This approach differs from vector processors, which typically execute a vector instruction to completion before starting the next one.



GPU Register Organization

Register Capacity

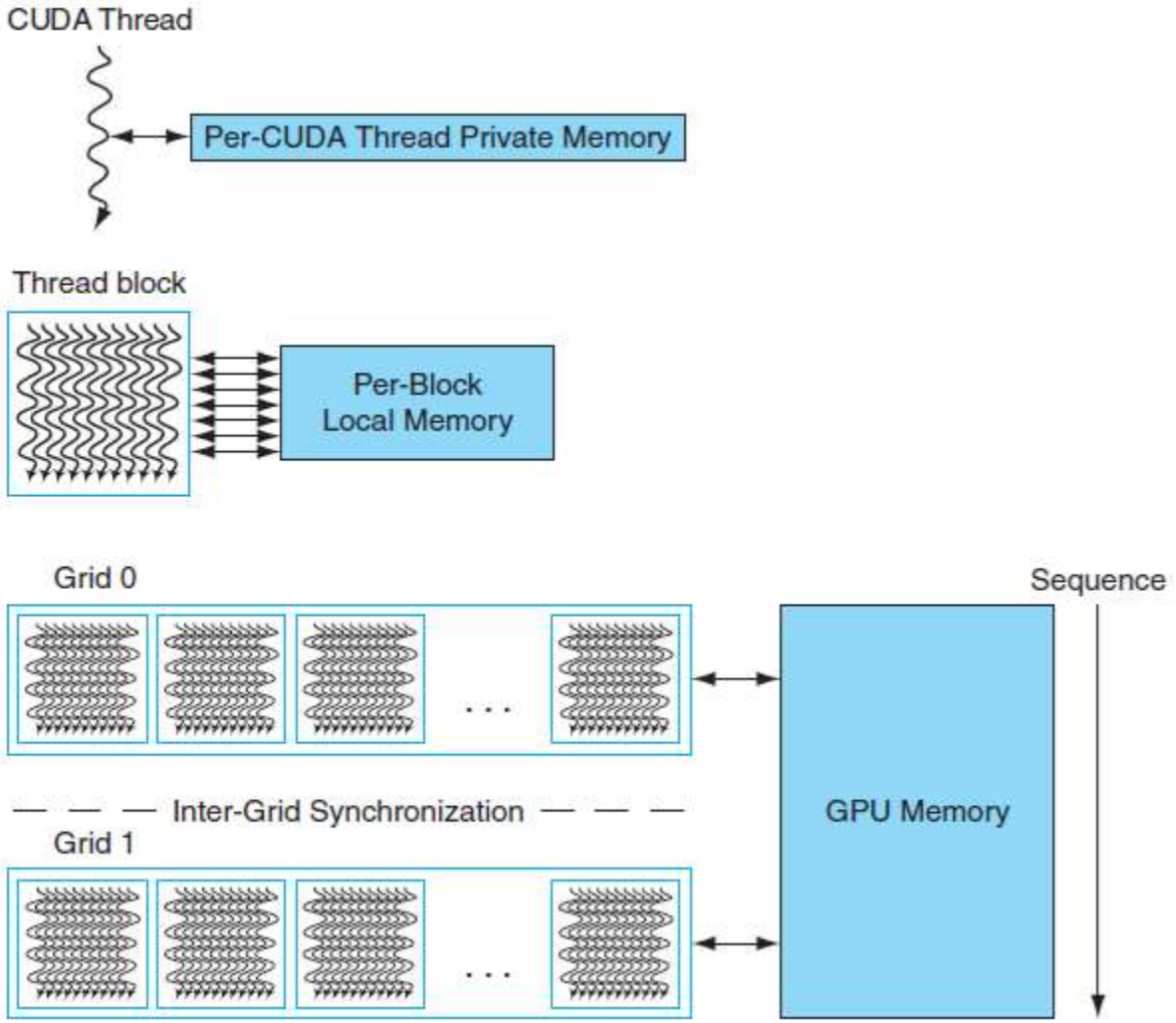
- 32,768 32-bit registers per SIMD Processor
- Registers divided logically across SIMD Lanes
- Each SIMD Thread limited to no more than 64 registers

- With 16 physical SIMD Lanes, each contains 2048 registers

Register Usage

- Think of a SIMD Thread as having up to 64 vector registers
- Each vector register has 32 elements (32 bits wide each)
- Double-precision operands use two adjacent 32-bit registers
- Registers dynamically allocated when threads are created
- Registers freed when the SIMD Thread exits

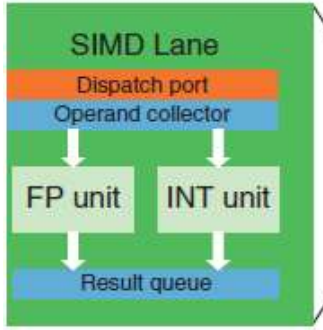
GPU Memory Structures

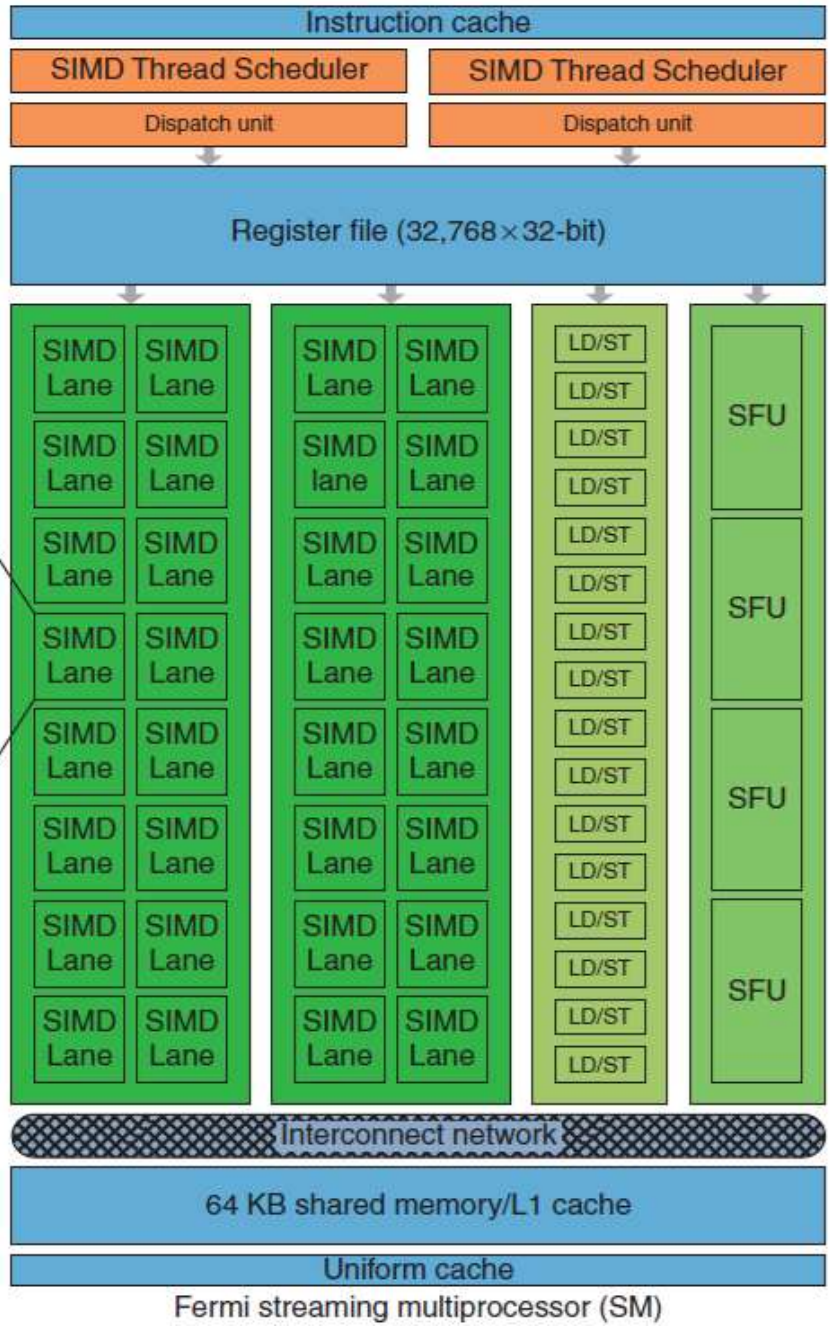


- Private Memory
  - Off-chip DRAM private to each SIMD Lane
  - Used for stack frames, register spilling, and private variables
  - Not shared between SIMD Lanes
- Local Memory (Shared Memory)
  - On-chip memory local to each multithreaded SIMD Processor
  - Shared by SIMD Lanes within a processor, but not between processors
  - Dynamically allocated to thread blocks
- GPU Memory (Global Memory)
  - Off-chip DRAM shared by the whole GPU and all thread blocks
  - Accessible by the host (system processor)

Fermi Architecture

- L1 Data and Instruction Cache for each SIMD Processor
- 768 KB L2 cache shared by all SIMD Processors
- Configurable SRAM: 16KB L1/48KB Local Memory or 48KB L1/16KB Local Memory





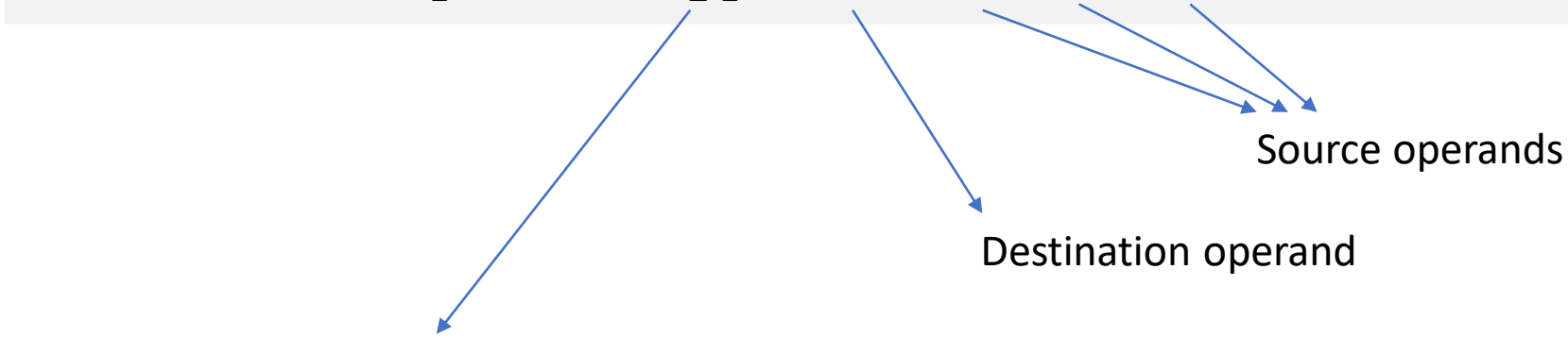
Fermi streaming multiprocessor (SM)

PTX: Parallel Thread Execution

- Unlike most system processors, NVIDIA compilers target PTX, an abstraction of the hardware instruction set that provides:
  - A stable instruction set for compilers
  - Compatibility across generations of GPUs
  - Virtual registers (physical register allocation happens at load time)
- PTX instructions describe operations on a single CUDA thread and usually map one-to-one with hardware instructions, but one PTX instruction can expand to many machine instructions, and vice versa.

PTX Instruction Format

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



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

Basic PTX Instructions

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,	atomic { d = *a; *a = cop(*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

Example

```
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];
}
```

<One iteration>

```
shl.u32 R8, blockIdx, 9     ; Thread Block ID * Block size (512 or 2^9)
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])
```

Address Coalescing

- To regain the efficiency of sequential (unit-stride) data transfers, GPUs include special Address Coalescing hardware that:
  - Recognizes when SIMD Lanes within a thread are collectively issuing sequential addresses
  - Notifies the Memory Interface Unit to request a block transfer of 32 sequential words
- For optimal performance, GPU programmers must ensure that adjacent CUDA Threads access nearby addresses at the same time that can be coalesced into one or a few memory or cache blocks.

Conditional Branching in GPUs

Branch Handling Mechanisms

- Explicit predicate registers
- Internal masks
- Branch synchronization stack
- Instruction markers

These mechanisms manage when a branch diverges into multiple execution paths and when paths converge.

Branch Efficiency

For equal length paths:

- IF-THEN-ELSE: 50% efficiency
- Doubly nested IF: 25% efficiency
- Triply nested IF: 12.5% efficiency

When all lanes agree on the branch condition, the processor can skip over unused code blocks entirely.

Conditional Branch Example

Original Code

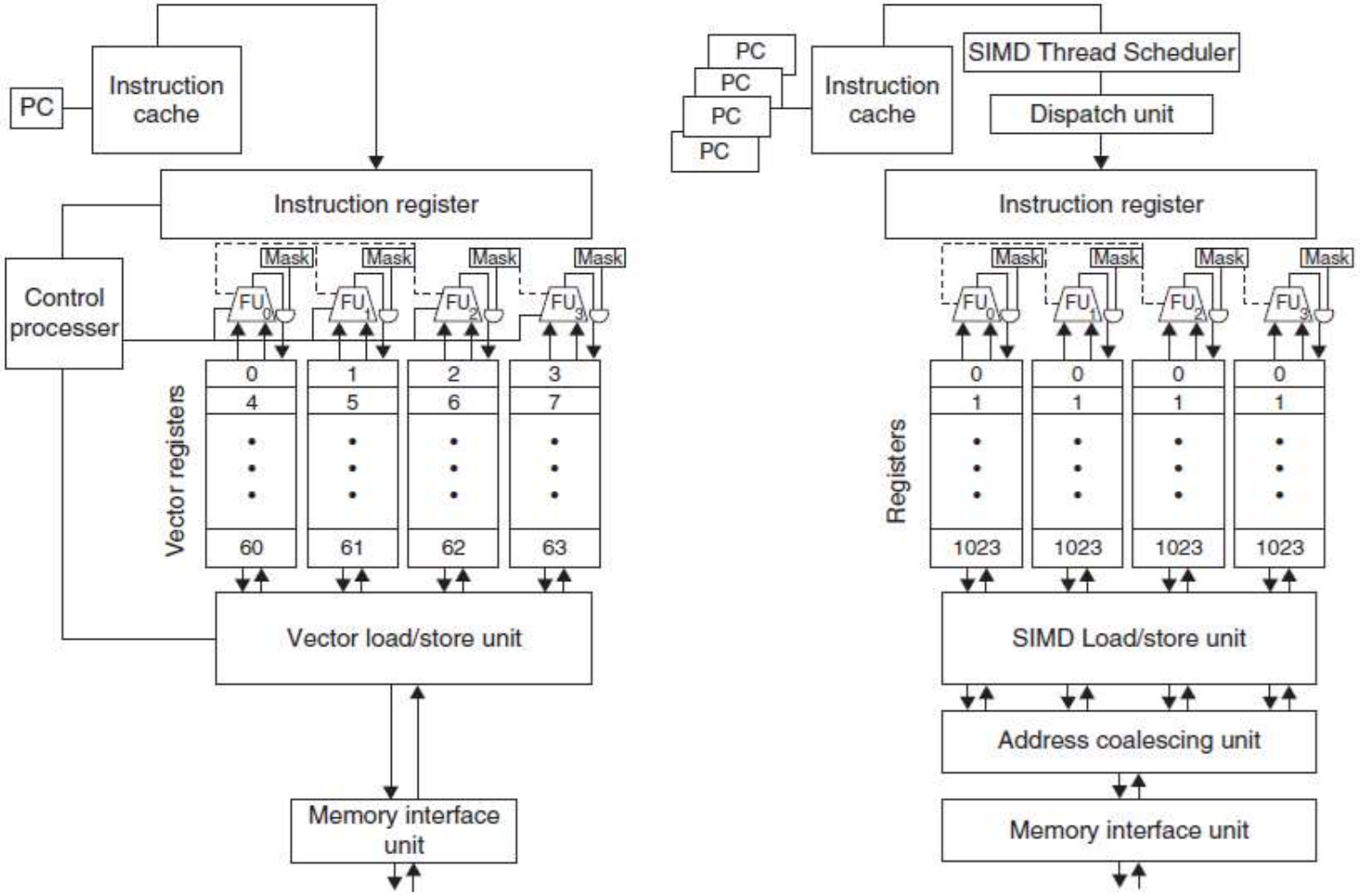
```
if (X[i] != 0)
  X[i] = X[i] - Y[i];
else  X[i] = Z[i];
```

PTX Code

```
ld.global.f64 RD0, [X+R8]    ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0     ; P1 is predicate register 1
@!P1, bra ELSE1, *Push       ; Push old mask, set new mask bits
                             ; if P1 false, go to ELSE1
ld.global.f64 RD2, [Y+R8]    ; RD2 = Y[i]
sub.f64 RD0, RD0, RD2        ; Difference in RD0
st.global.f64 [X+R8], RD0    ; X[i] = RD0
@P1, bra ENDIF1, *Comp       ; complement mask bits
                             ; if P1 true, go to ENDIF1
ELSE1:  ld.global.f64 RD0, [Z+R8]    ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0    ; X[i] = RD0
ENDIF1: <next instruction>, *Pop     ; pop to restore old mask
```

*Push, *Comp, and *Pop indicate branch synchronization markers inserted by the PTX assembler to manage mask registers.

Vector Processor vs. SIMD Processor



Program Abstractions & Machine Objects

Vector term	Closest CUDA/NVIDIA GPU term	Comment
Vectorized Loop	Grid	Concepts are similar, with the GPU using the less descriptive term.
Chime	-	Since a vector instruction (PTX Instruction) takes just two cycles on Fermi and four cycles on Tesla to complete, a chime is short in GPUs.
Vector Instruction	PTX Instruction	A PTX instruction of a SIMD thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction.
Gather/Scatter	Global load/store (ld.global/st.global)	All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it.
Mask Registers	Predicate Registers and Internal Mask Registers	Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically.

Processor

Vector term	Closest CUDA/NVIDIA GPU term	Comment
Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not.
Control Processor	Thread Block Scheduler	The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide.
Scalar Processor	System Processor	Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture.

Memory System

Vector term	Closest CUDA/NVIDIA GPU term	Comment
Vector Lane	SIMD Lane	Both are essentially functional units with registers.
Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 32 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD thread is flexible, but the maximum is 64, so the maximum number of vector registers is 64.
Main Memory	GPU Memory	Memory for GPU versus System memory in vector case.

SIMD Extension vs. GPU

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes per processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 to 256 GB	4 to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

Loop-Level Parallelism

Loops are the primary source of parallelism in programs. Compiler technology can discover and exploit this parallelism through careful analysis of data dependencies between loop iterations.

Loop-Carried Dependence

Occurs when data accesses in later iterations depend on values produced in earlier iterations. This is the key factor that determines whether a loop can be parallelized.

Induction Variables

Variables like loop counters that follow a predictable pattern can often be recognized and eliminated, enabling parallelization even when apparent dependencies exist.

Intra-Loop Dependence

Dependencies within a single iteration don't prevent parallelism across iterations, as long as statements within each iteration maintain their order.

Finding and manipulating loop-level parallelism is critical for exploiting both Data-Level Parallelism (DLP) and Thread-Level Parallelism (TLP), as well as more aggressive static Instruction-Level Parallelism (ILP) approaches.

Analyzing Loop Dependencies

```
for (i=0; i<100; i=i+1) {
    A[i+1] = A[i] + C[i];     /* S1 */
    B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

- This loop contains two different dependencies:
  - S1 uses A[i] computed in the previous iteration - a loop-carried dependence that forces sequential execution
  - S2 uses A[i+1] computed by S1 in the same iteration - not loop-carried, allowing parallel execution if statements remain ordered
- The compiler must identify these dependencies to determine how much parallelism can be exploited.

Transforming Loops for Parallelism

Not all loop-carried dependencies prevent parallelism. Consider this example:

```
for (i=0; i<100; i=i+1) {
    A[i] = A[i] + B[i];     /* S1 */
    B[i+1] = C[i] + D[i];   /* S2 */
}
```

S1 depends on the value of B[i] assigned by S2 in the previous iteration. However, this dependence isn't circular - S2 doesn't depend on S1.

We can transform the loop to expose parallelism:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Now the dependence is no longer loop-carried, allowing iterations to overlap if statements remain ordered within each iteration.

Finding Dependencies in Array Accesses

Affine Array Indices

Most dependence analysis algorithms work on array indices in the form a×i+b, where a and b are constants and i is the loop index variable.

Mathematical Tests

Determining dependence requires checking if two affine functions can have the same value for different indices within loop bounds.

GCD Test

A simple test: if a loop-carried dependence exists between array accesses a×i+b and c×i+d, then GCD(c,a) must divide (d-b).

While determining whether a dependence exists is generally NP-complete, many common cases can be analyzed precisely at low cost. Modern compilers use a hierarchy of exact tests increasing in generality and cost to efficiently analyze dependencies.

Dependence analysis is critical for exploiting parallelism, but it has limitations - it works best with affine array indices and struggles with pointer-based accesses and cross-procedure analysis.

Example

- Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=0; i<100; i=i+1) {
  X[2*i+3] = X[2*i] * 5.0;
}
```

- Answer
  - Given the values a = 2, b = 3, c = 2, and d = 0, then GCD(a,c) = 2, and d − b = −3.
  - Since 2 does not divide −3, no dependence is possible.

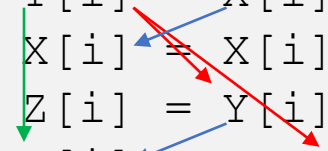
Example 2

- The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i=i+1) {
  Y[i] = X[i] / c; /* S1 */
  X[i] = X[i] + c; /* S2 */
  Z[i] = Y[i] + c; /* S3 */
  Y[i] = c - Y[i]; /* S4 */
}
```

Example 2

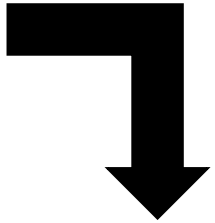
```
for (i=0; i<100; i=i+1) {
  Y[i] = X[i] / c; /* S1 */
  X[i] = X[i] + c; /* S2 */
  Z[i] = Y[i] + c; /* S3 */
  Y[i] = c - Y[i]; /* S4 */
}
```



- Answer
  - True dependences
    - From S1 to S3 and from S1 to S4 because of Y[i]
    - These are not loop carried, so they do not prevent the loop from being considered parallel
    - These dependences will force S3 and S4 to wait for S1 to complete
  - Antidependences
    - From S1 to S2, based on X[i], and from S3 to S4 for Y[i]
  - Output dependence
    - From S1 to S4, based on Y[i]

Example 2

```
for (i=0; i<100; i=i+1) {
  Y[i] = X[i] / c; /* S1 */
  X[i] = X[i] + c; /* S2 */
  Z[i] = Y[i] + c; /* S3 */
  Y[i] = c - Y[i]; /* S4 */
}
```



```
for (i=0; i<100; i=i+1) {
  T[i] = X[i] / c;  /* Y renamed to T to remove output dependence */
  X1[i] = X[i] + c; /* X renamed to X1 to remove antidependence */
  Z[i] = T[i] + c;  /* Y renamed to T to remove antidependence */
  Y[i] = c - T[i];
}
```

Handling Recurrences and Reductions

```
for (i=9999; i>=0; i=i-1)
  sum = sum + x[i] * y[i];
```

Make the loop parallel, but need a reduction.
Reductions are common in linear algebra.

```
for (i=9999; i>=0; i=i-1)
  sum[i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
  finalsum = finalsum + sum[i];
```

This loop, which sums up 1000 elements on each of the ten processors, is completely parallel.
A simple scalar loop can then complete the summation of the last ten sums.

```
for (i=999; i>=0; i=i-1)
  finalsum[p] = finalsum[p] + sum[i+1000*p];
```