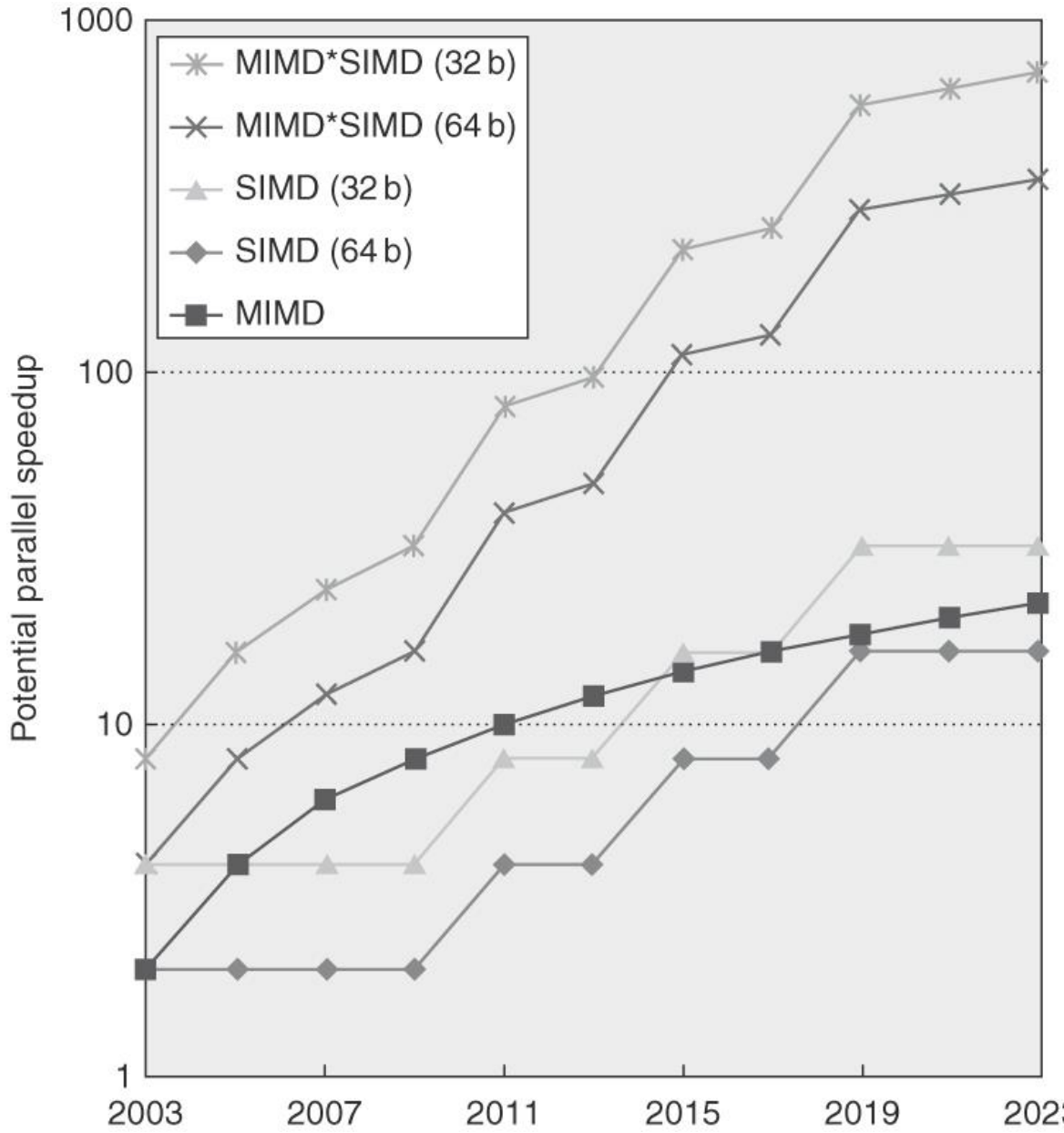
Data-Level Parallelism and GPUs

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions
- Graphics Processing Units (GPU)
- Loop Level Parallelism
- Conclusion

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to think sequentially
- SIMD can be even faster than MIMD!

Potential Speedup via Parallelism



Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD for x86 computers. It assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD (width) will double every four years.

SIMD is comparable or better than MIMD

Data-Level Parallelism and GPUs

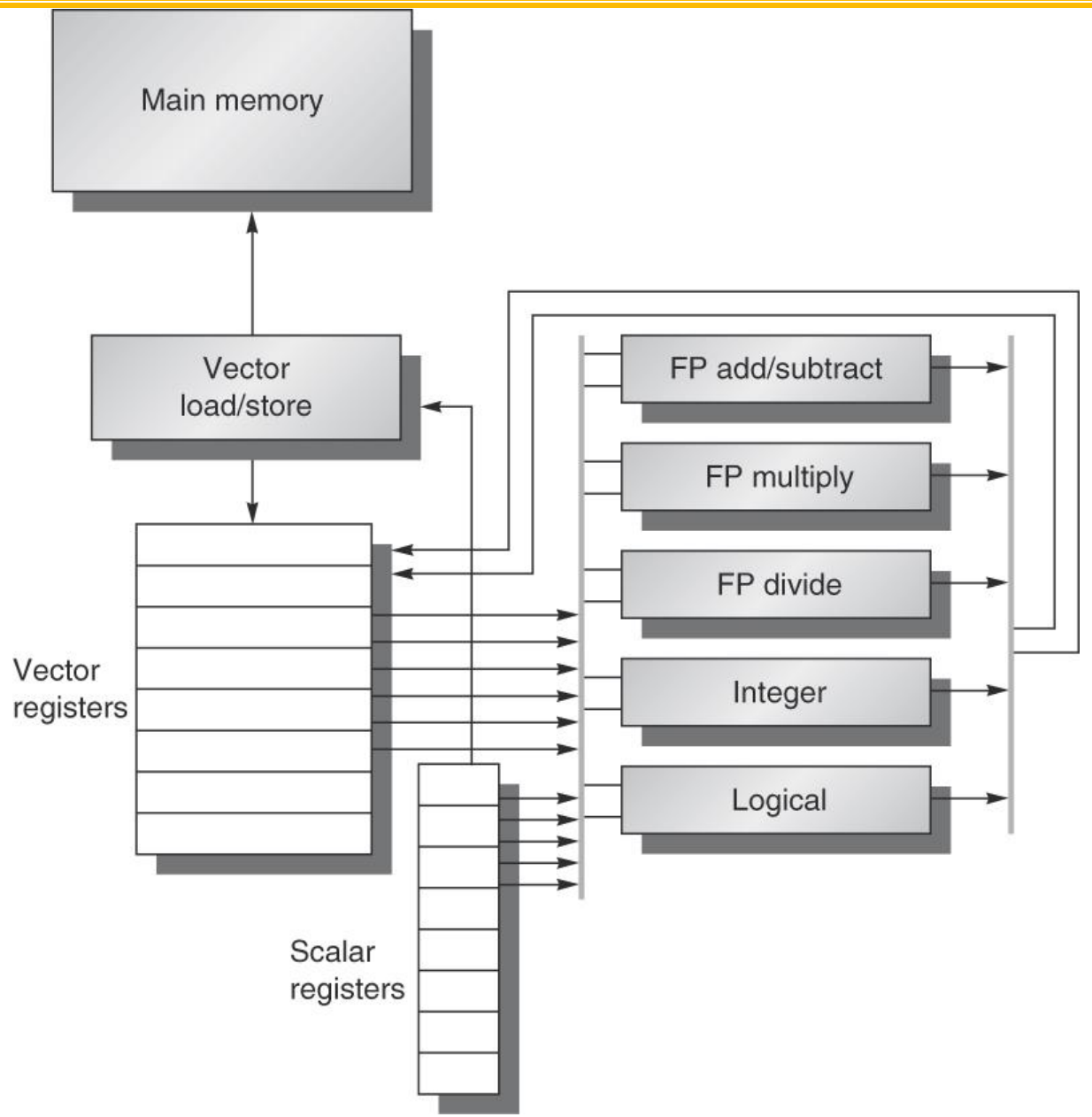
- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions
- Graphics Processing Units (GPU)
- Loop Level Parallelism
- Conclusion

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Example: VMIPS



Example: VMIPS

- VMIPS is loosely based on Cray-1
- Vector registers
  - 8 64 element vector (each element 64 bits)
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers

Y = a.X + Y (a scalar, X and Y vectors)

MIPS Code

```
        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
```

≈ 600 dynamic instructions

[each array has 64 elements]

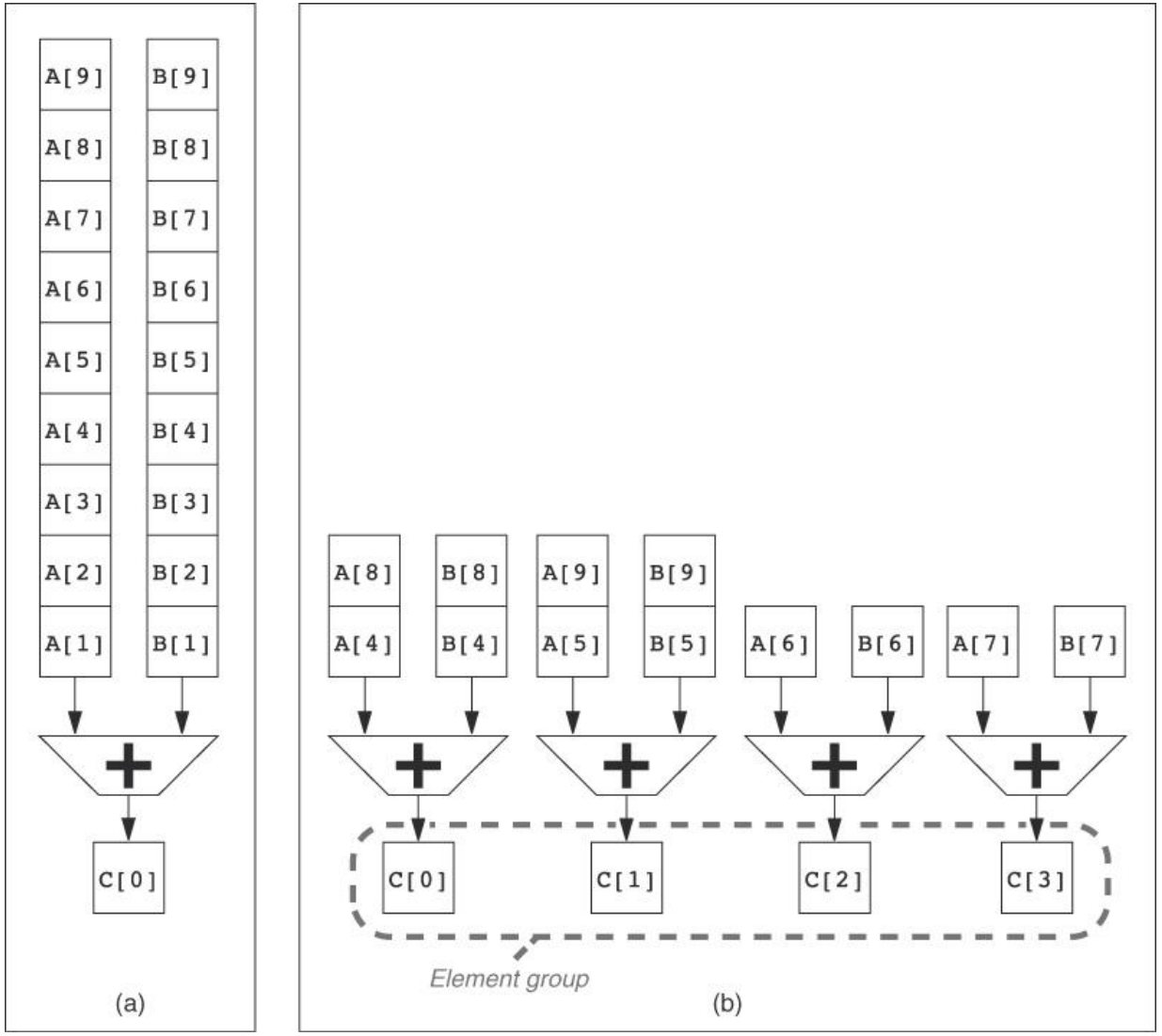
VMIPS Code

```
        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 two vectors
        SV      Ry,V4          ;store the sum
```

➔ 6 dynamic instructions

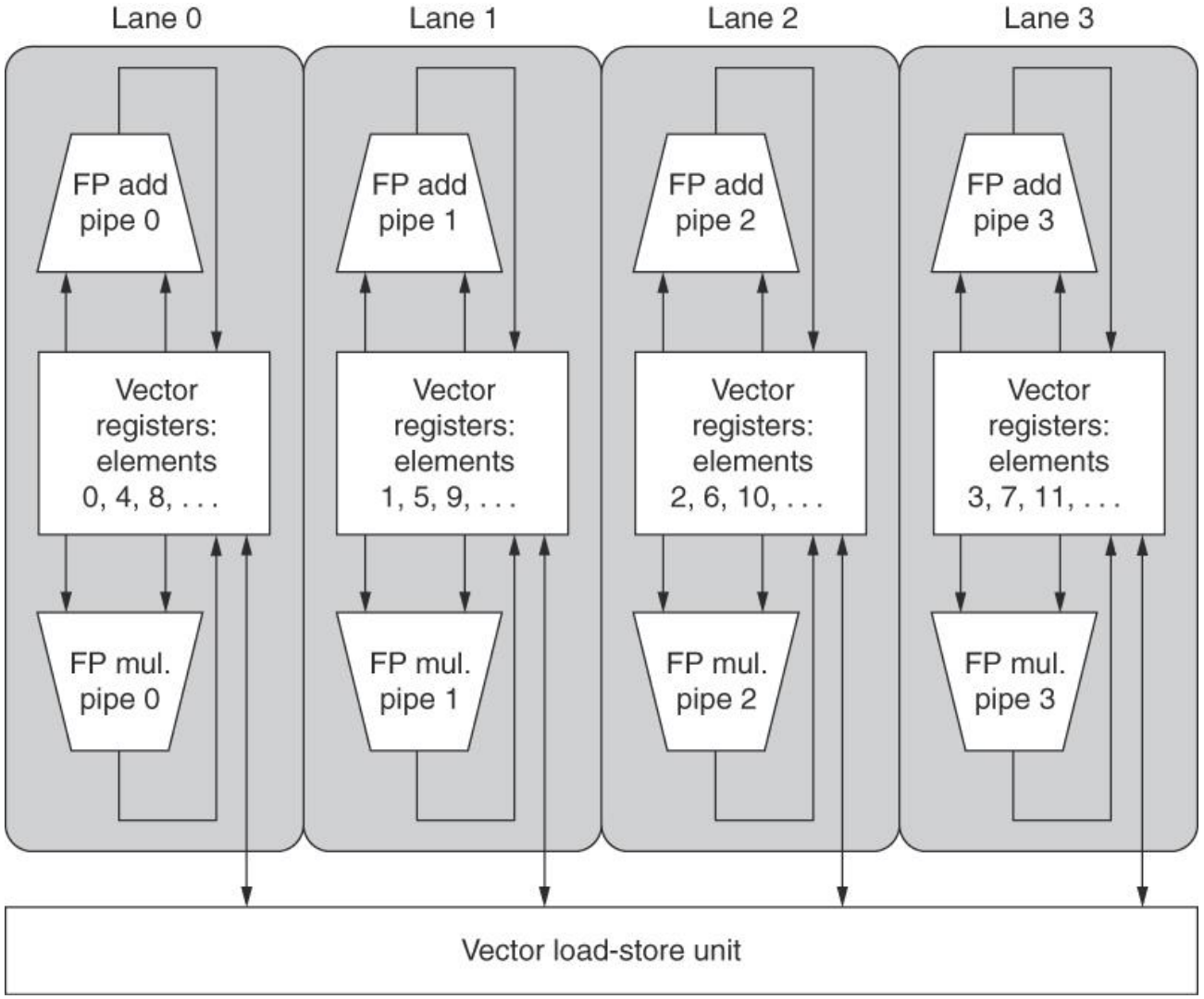
MIPS has 64x more stalls and 100x more instructions than VMIPS.

Multiple Functional Units



Multiple functional units to improve performance of a vector add instruction, C = A + B. The vector processor (a) has a single add pipeline and can complete one addition per cycle. The vector processor (b) has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines.

Multiple Lanes



Element *n* of vector register *A* is "hardwired" to element *n* of vector register *B*. The vector register is divided across the lanes, with each lane holding every fourth element of each vector register. Each of the vector arithmetic units contains four execution pipelines, one per lane, which act in concert to complete a single vector instruction.

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length

- *Convoy*
  - Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*

- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

- *Chime*
  - Unit of time to execute one convoy
  - *m* convoys execute in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

Example

```
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 two vectors
SV        Ry,V4      ;store the sum
```

- Three Convoys
  - LV MULVS.D ; Chaining allows them to be in the same convoy
  - LV ADDVV.D
  - SV
- Three Chimes
  - For 64-element vectors, takes 64 x 3 = 192 cycles

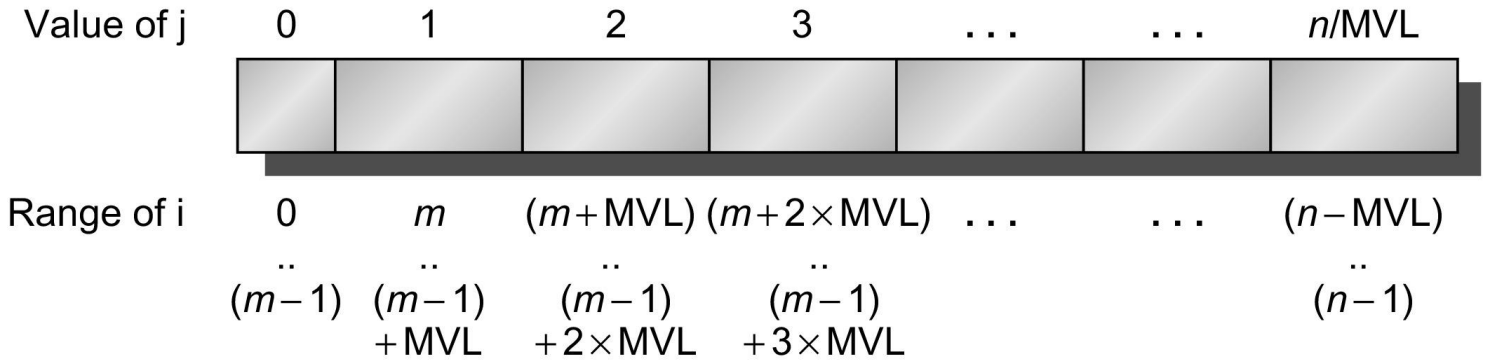
Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Performance improvements
  - More than one element per clock cycle
  - Non-64 wide vectors
  - IF statements in vector code
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- **Strip mining** for vectors over the maximum length:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
      Y[i] = a * X[i] + Y[i] ; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```



Vector Mask Registers

- Consider:

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

- Use vector mask register to “disable” elements:

```
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          Rx,V1       ;store the result in X
```

- GFLOPS rate decreases!

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing same memory
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?
    - Number of memory references each cycle = 32 x (4+2) = 192
    - SRAM bank busy for 15/2.167 ≈ 7 cycles
    - Need a minimum of 192 x 7 = 1344 memory banks

Stride: Handling Multidimensional Arrays

- Consider:

```
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];
}
```

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time

Scatter-Gather: Handling Sparce Matrices

- Consider:

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

- Use index vector:

```
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 provide feedback to programmers
- Programmers can provide hints to compiler

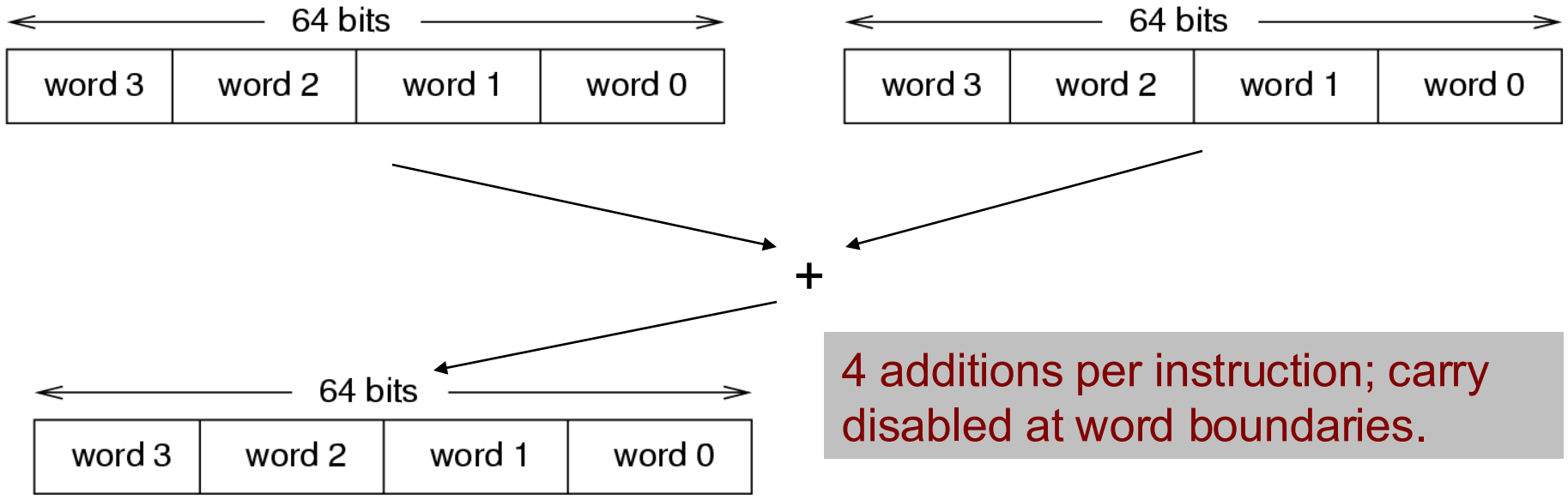
Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	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

Data-Level Parallelism and GPUs

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions
- Graphics Processing Units (GPU)
- Loop Level Parallelism
- Conclusion

Multimedia Instructions

- Multimedia instructions exploit the fact that
  - Many registers, adders etc. are wide (32/64 bit)
  - Most multimedia data types are narrow
    - e.g., 8 bit per color, 16 bit per audio sample per channel
- 2-8 values can be stored/register and added.



4 additions per instruction; carry disabled at word boundaries.

SIMD: Single Instruction Multiple Data

Example SIMD Code

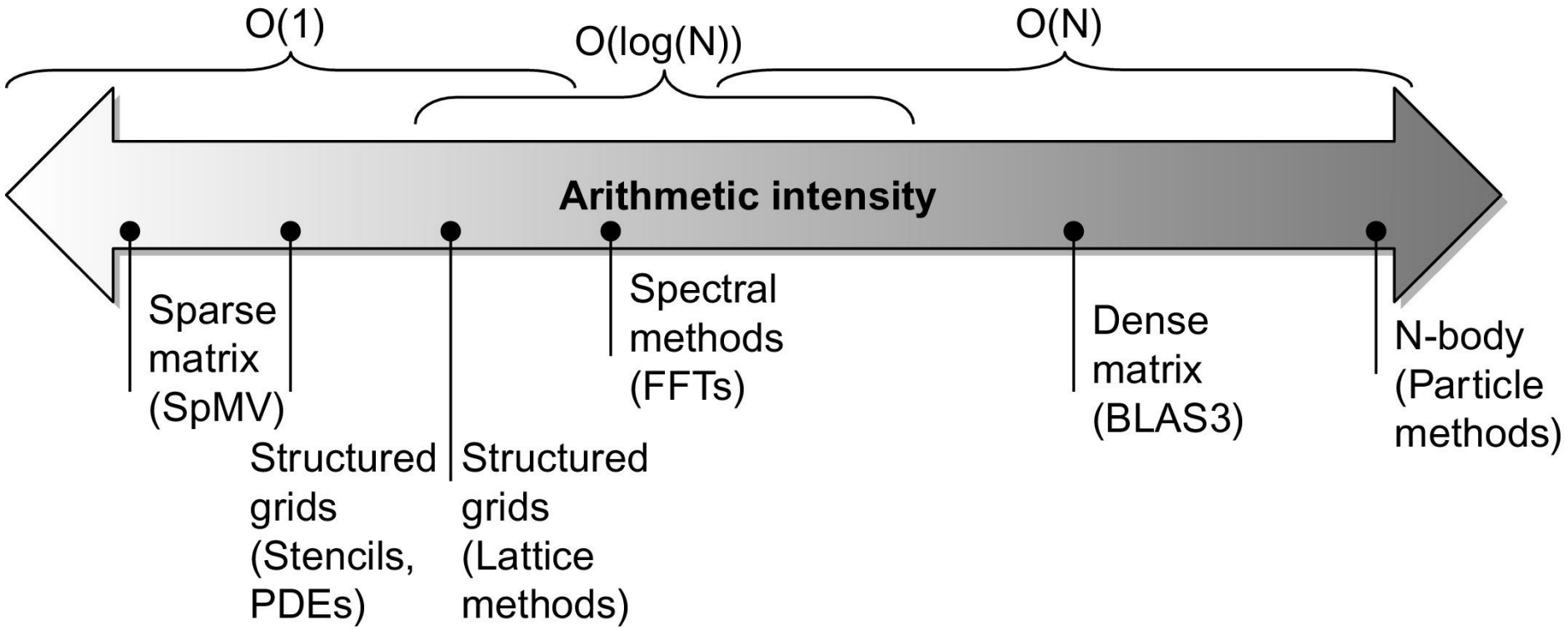
```
        L.D       F0,a          ;load scalar a
        MOV       F1, F0        ;copy a into F1 for SIMD MUL
        MOV       F2, F0        ;copy a into F2 for SIMD MUL
        MOV       F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU    R4,Rx,#512    ;last address to load

Loop:   L.4D      F4,0[Rx]      ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4
        AD
        S.4
        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
```

MUL.4D F4,F4,F0 → F4*F0, F5*F1, F6*F2, F7*F3

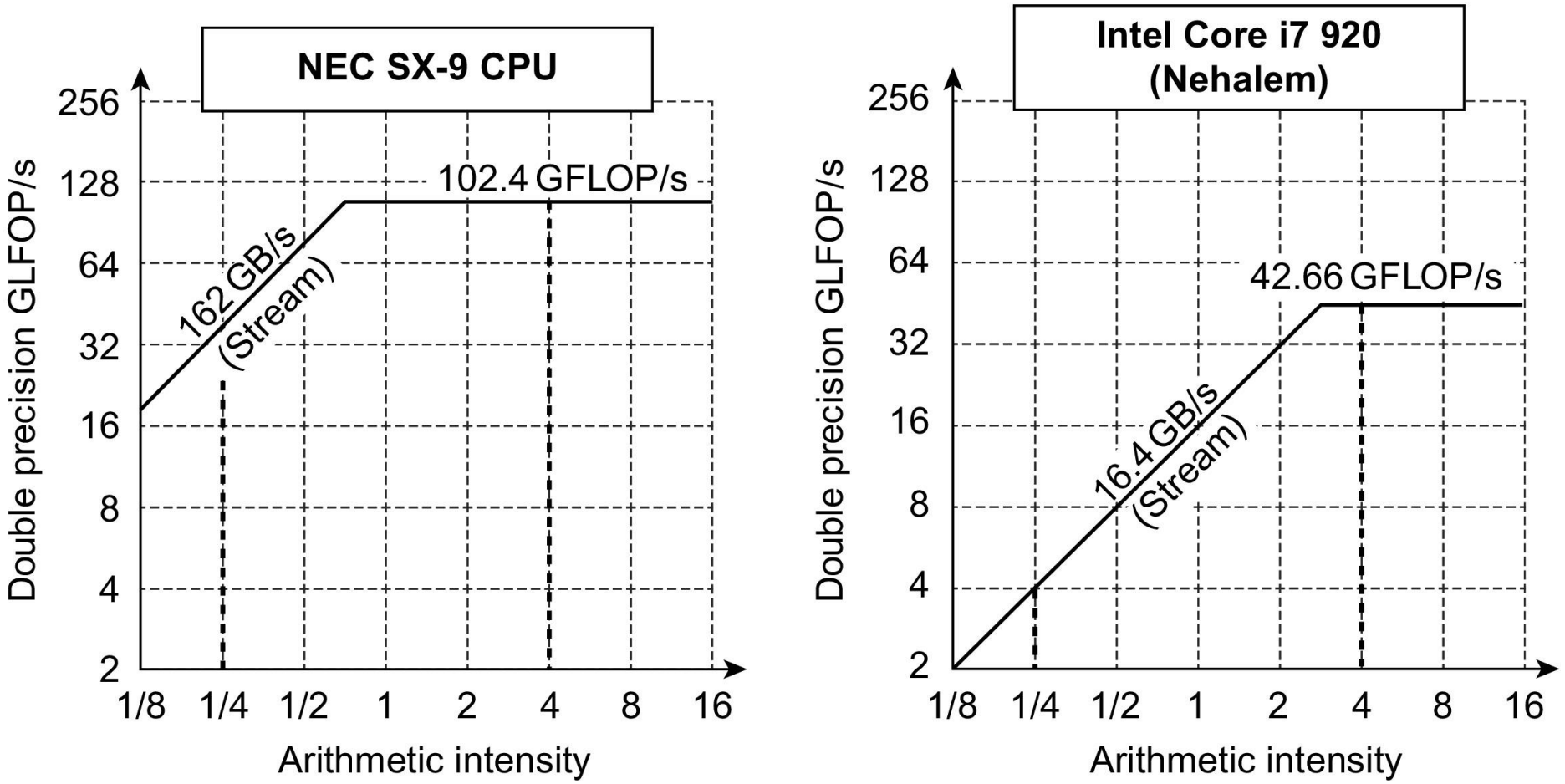
SIMD gets 4x reduction unlike 100x reduction in VMIPS

Arithmetic Intensity



Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory (Williams et al., 2009). Some kernels have an arithmetic intensity that scales with problem size, such as a dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

Roofline Model



Roofline model for one NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer with SIMD extensions on the right (Williams et al., 2009). The dashed vertical lines at an arithmetic intensity of 4 FLOP/byte show that both processors operate at peak performance. In this case, the SX-9 at 102.4 FLOP/s is 2.4 × faster than the Core i7 at 42.66 GFLOP/s. At an arithmetic intensity of 0.25 FLOP/byte, the SX-9 is 10 × faster at 40.5 GFLOP/s vs 4.1 GFLOP/s for the Core i7.

Data-Level Parallelism and GPUs

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions
- Graphics Processing Units (GPU)
- Loop Level Parallelism
- Conclusion

Graphics Processing Units (GPU)

- Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is “Single Instruction Multiple Thread”

Simple CUDA Example

```
int main()
{
    …
    // Vector addition with N threads
    VecAdd<1, N>>>(A, B, C);
    …
}
```

CPU

1 block

N threads per block

Asynchronous call

Which of the N threads?

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    // Vector addition with N threads
    C[i] = A[i] + B[i]
}
```

N instances of VecAdd is spawned in GPU

NVIDIA GPU Architecture

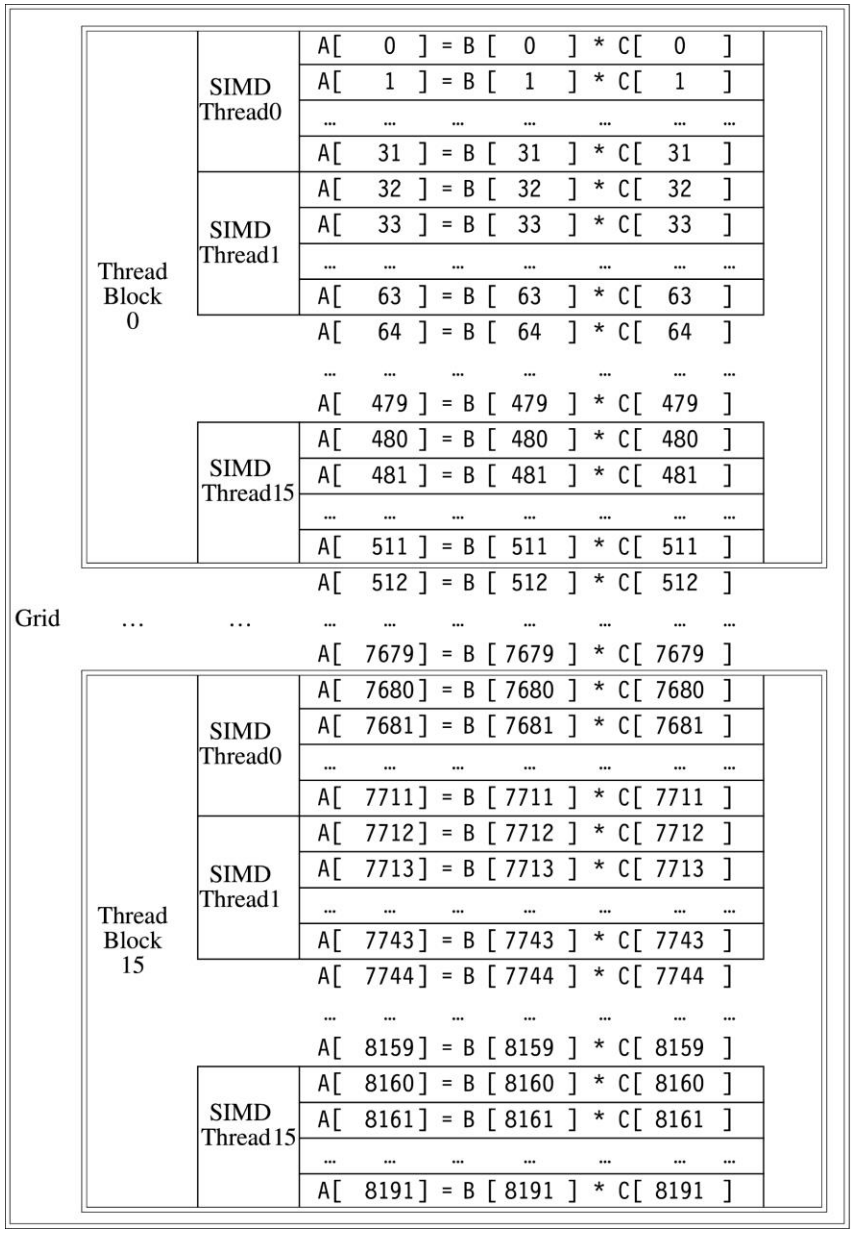
- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Threads and Blocks

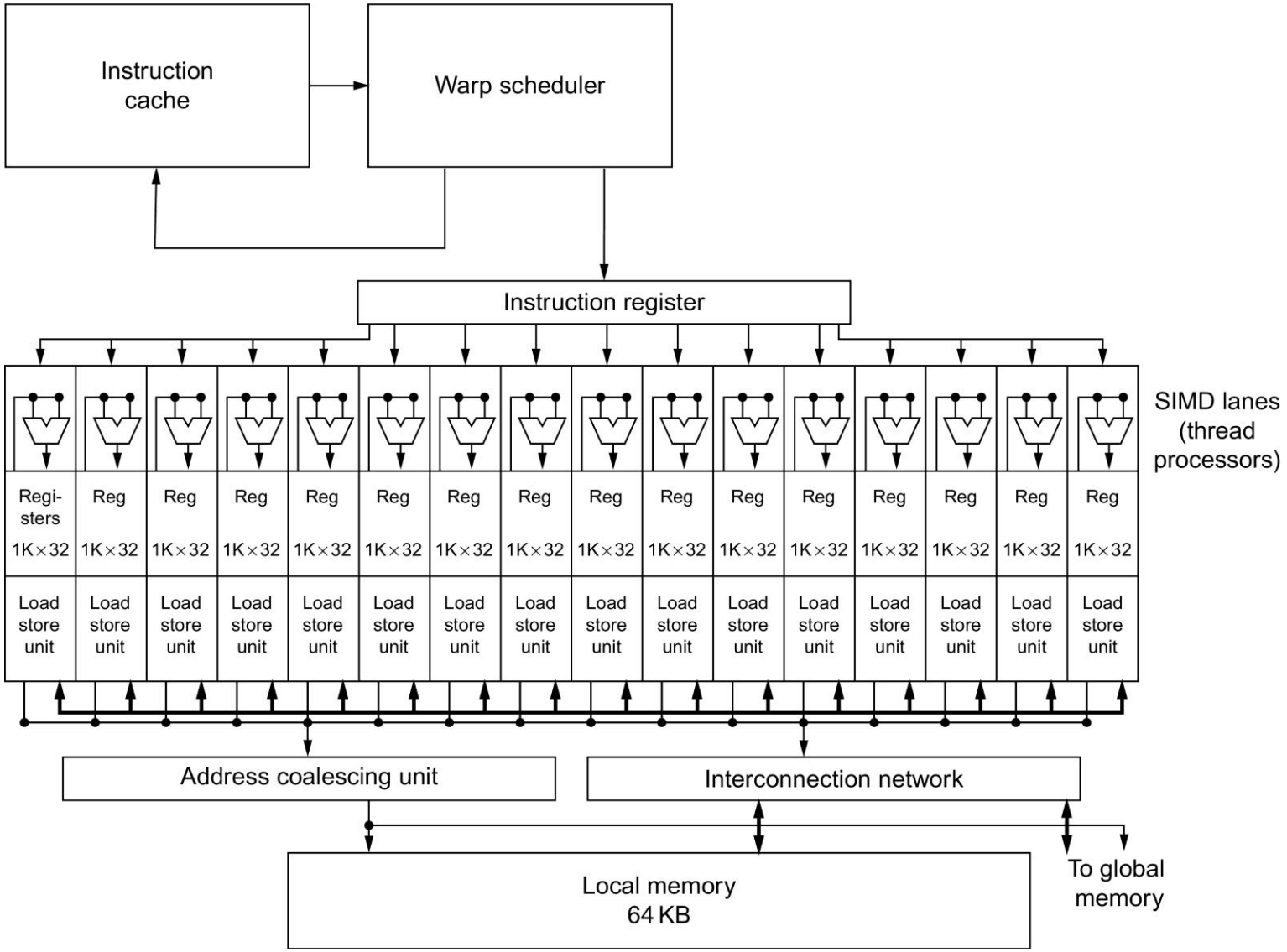
- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid
- GPU hardware handles thread management, not applications or operating system (OS)

SIMD Threads in NVIDIA GPUs



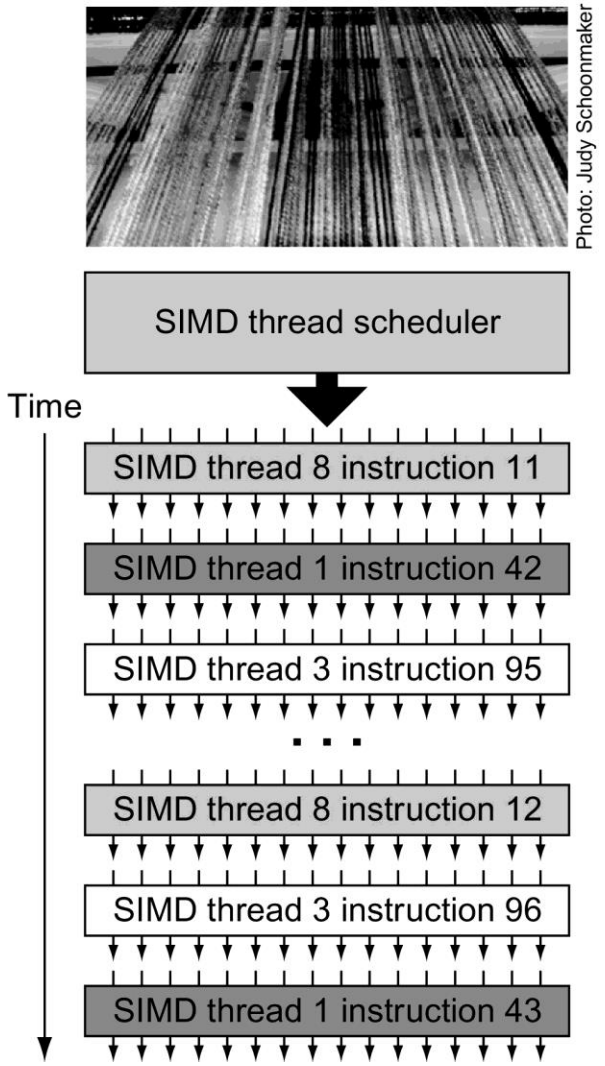
- The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long.
- Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks.
- 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.

Typical Multithreaded SIMD Processor



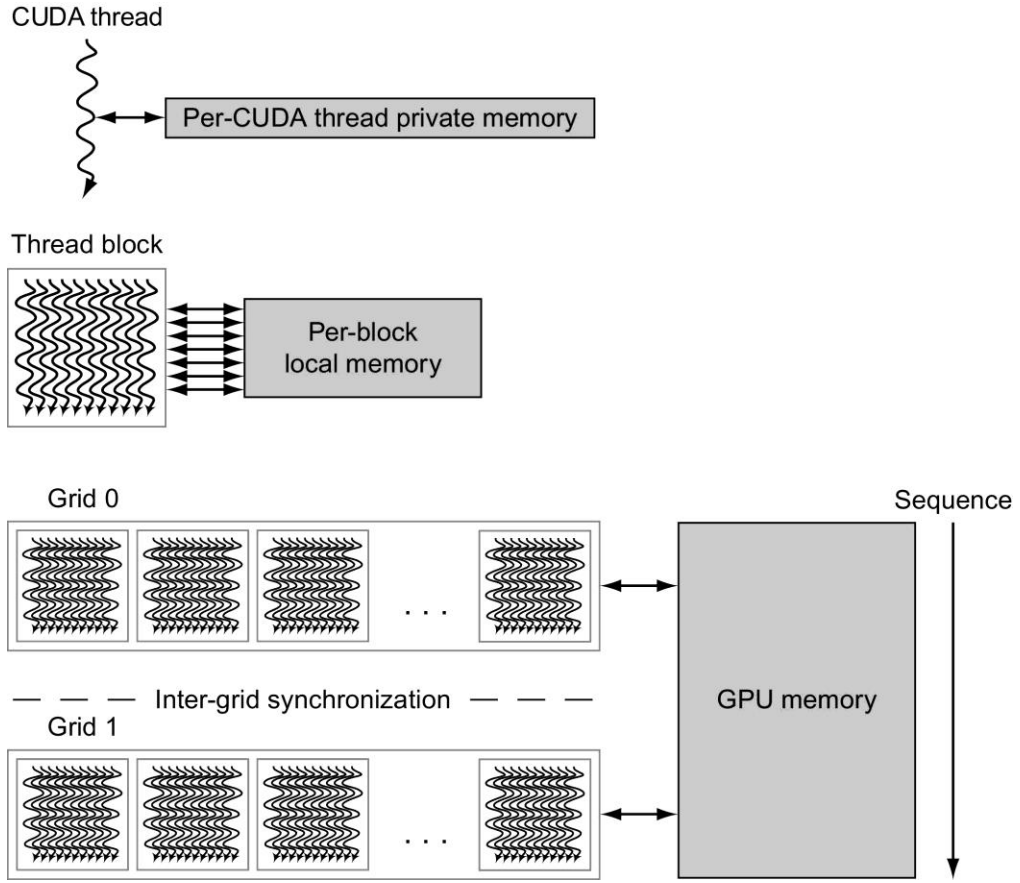
It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

SIMD Thread Scheduler



The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD Thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD Thread each time.

GPU Memory Structures

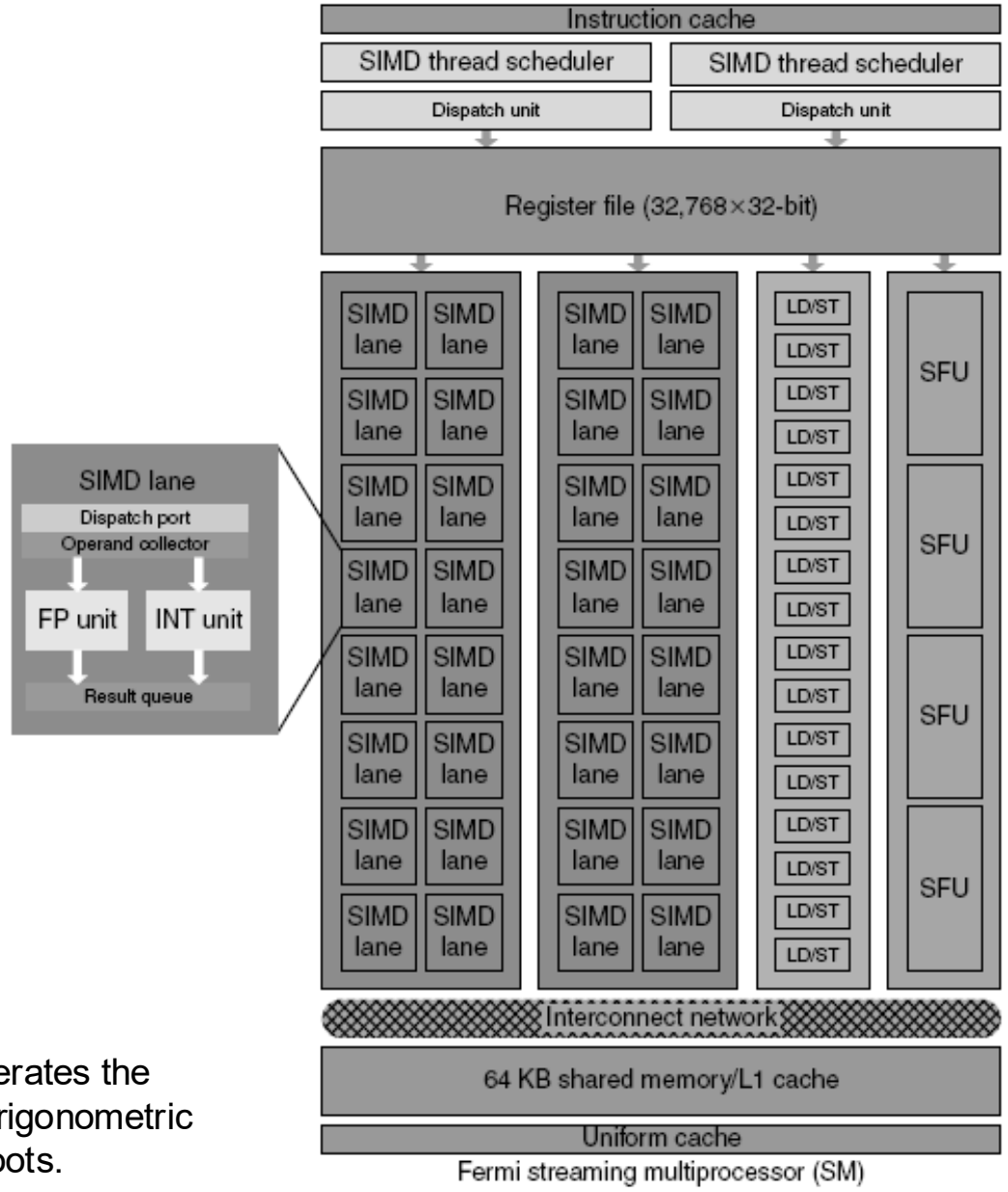


GPU memory is shared by all Grids (vectorized loops), local memory is shared by all threads of SIMD instructions within a Thread Block (body of a vectorized loop), and private memory is private to a single CUDA Thread. Pascal allows preemption of a Grid, which requires that all local and private memory be able to be saved in and restored from global memory. The GPU can also access CPU memory via the PCIe bus. This path is commonly used for a final result when its address is in host memory. This option eliminates a final copy from the GPU memory to the host memory.

Example: Multiply two vectors of length 8192

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes
  - 512 threads per block
- SIMD instruction executes 32 elements at a time
- Thus grid size = 16 blocks
- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
- Fermi has 7-15 multithreaded SIMD processors

Fermi GPU (32 SIMD Lanes)



Fermi streaming multiprocessor (SM)

Special Functional Unit (SFU) accelerates the computation of transcendental and trigonometric functions like *sin*, *cos*, and square roots.

Nvidia GPU Architectures

- Fermi (2010)
- Kepler (2012)
- Maxwell (2014)
- Pascal (2016)
- Volta (2017)
- Turing (2018
- GeForce (2019-2021)
- Blackwell (2024)

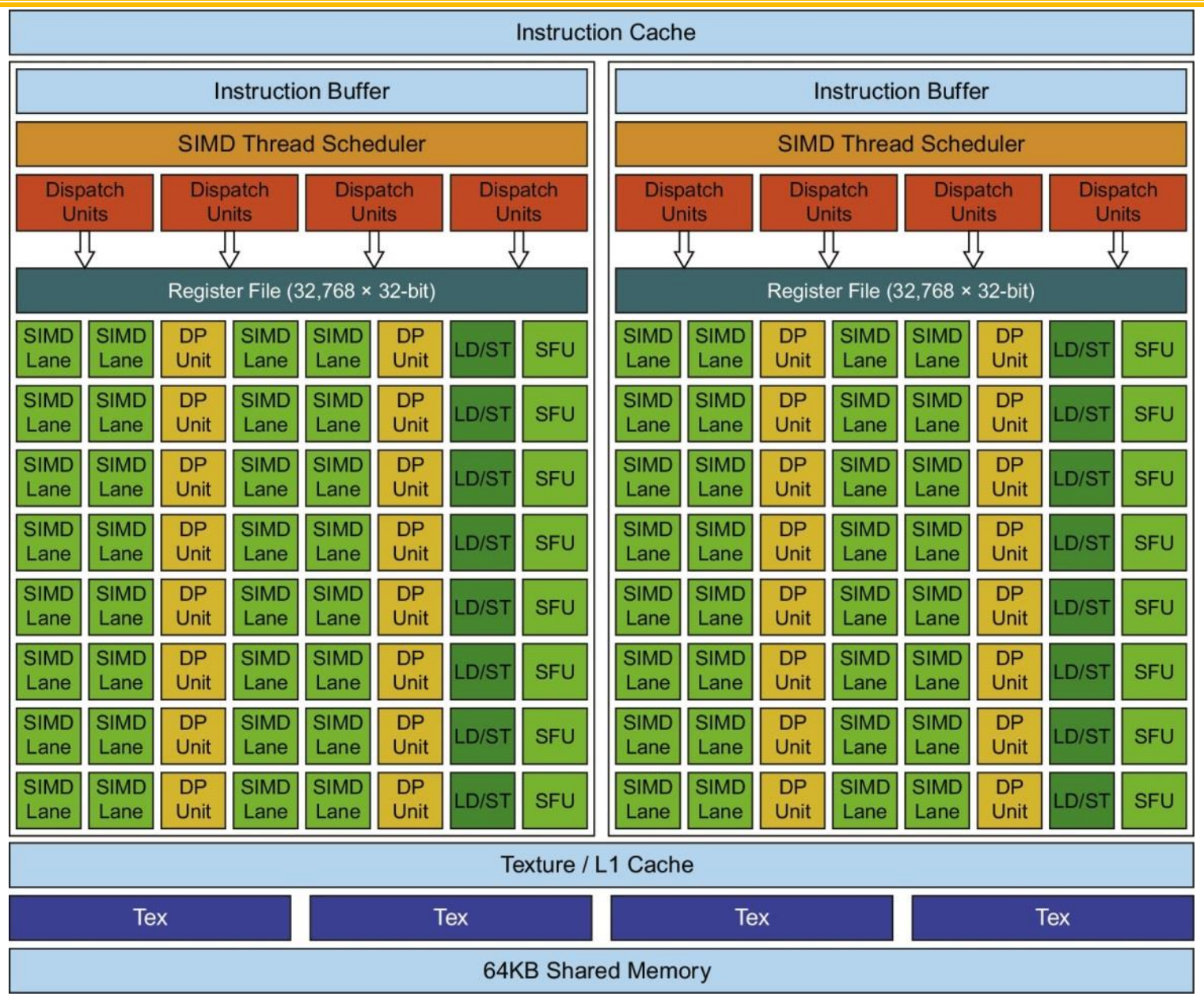
Intel CPU Architectures

- 8086
- …
- P5
- P6
- NetBurst
- Intel Core
- Nehalem (2008)
- …
- Arrow Lake (2024)

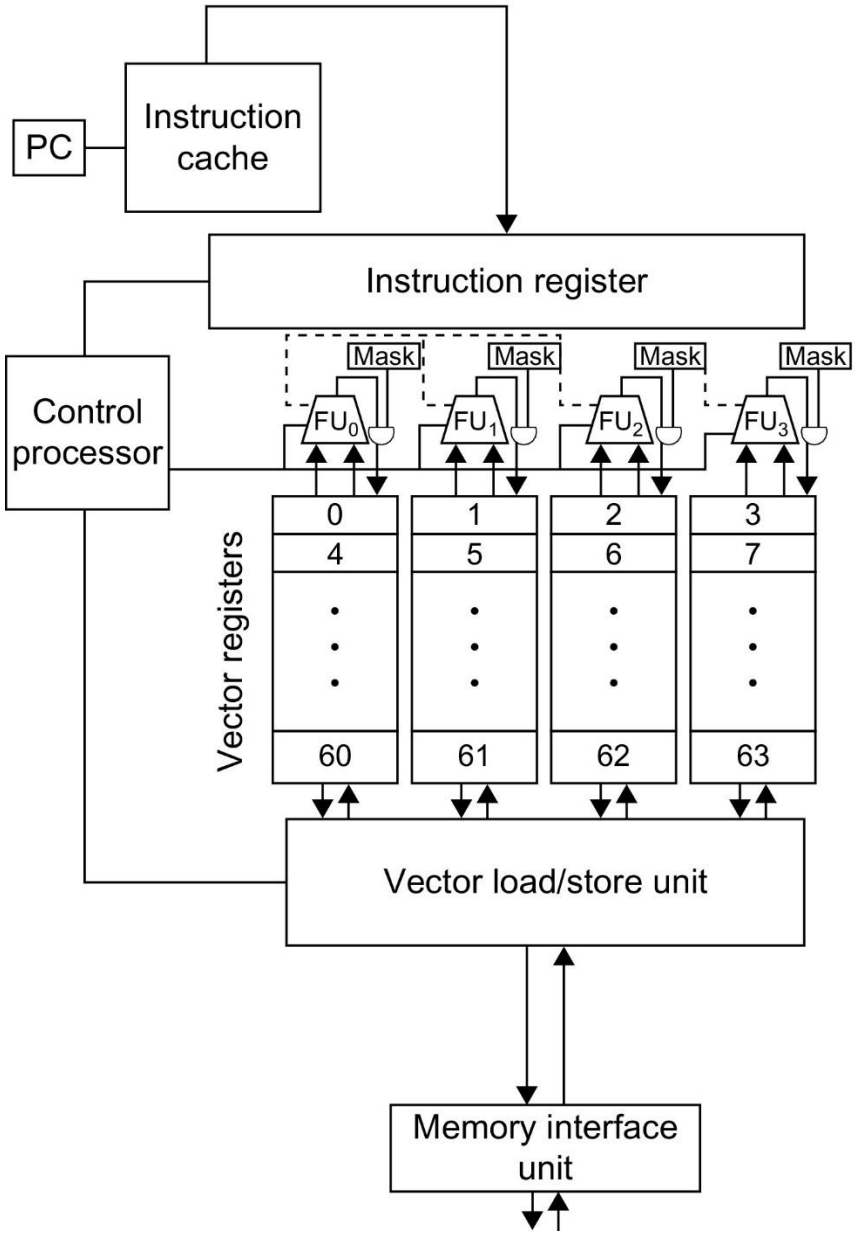
Pascal P100 GPU

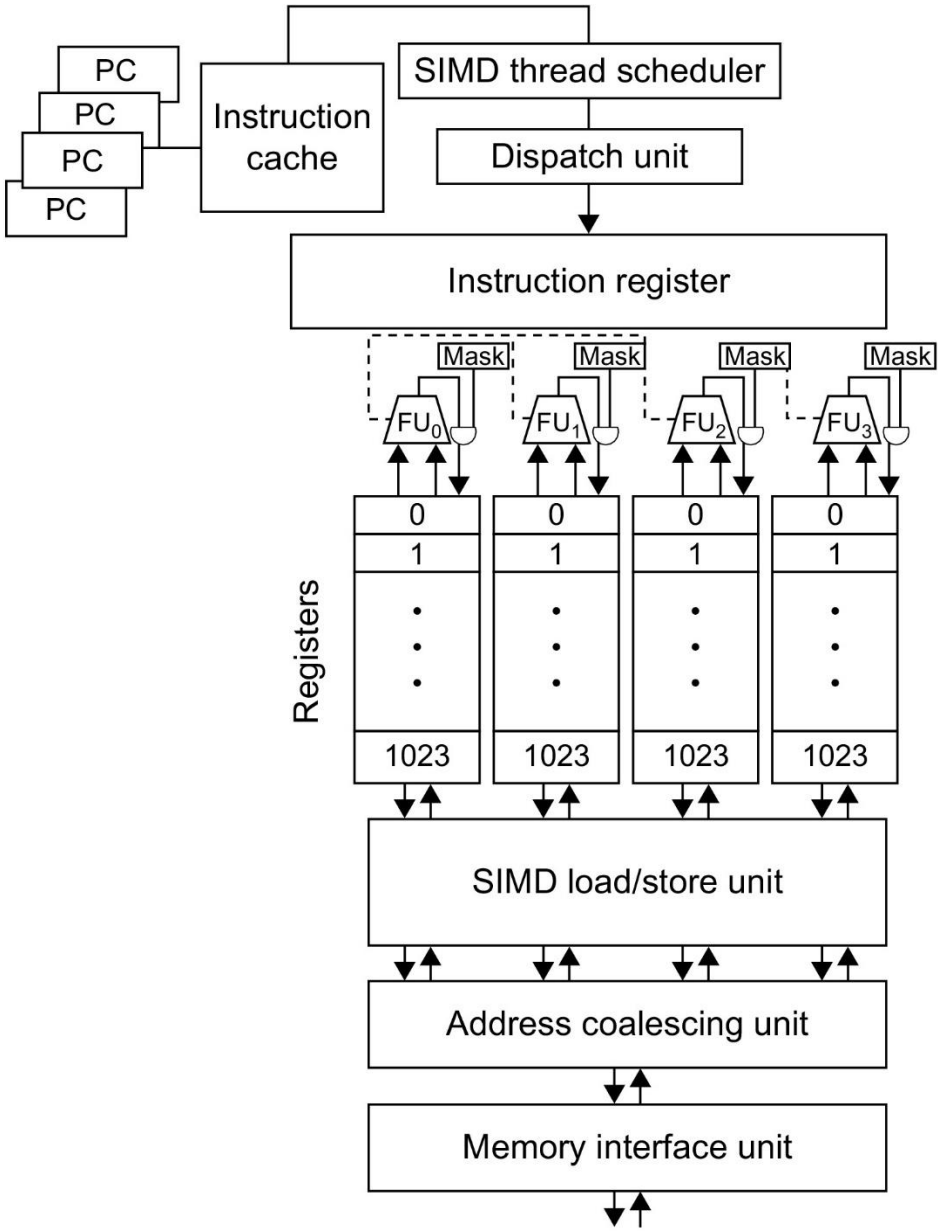
The P100 has 56 multithreaded SIMD Processors, each with an L1 cache and local memory, 32 L2 units, and a memory-bus width of 4096 data wires. (It has 60 blocks, with four spares to improve yield.) The P100 has 4 HBM2 ports supporting up to 16 GB of capacity. It contains 15.4 billion transistors.

Pascal GPU (64 SIMD Lanes)



Vector Processor versus SIMD (GPU)





Intel Core i7-960 versus GTX 280

	Core i7-960	GTX 280	Ratio 280/i7
Number of processing elements (cores or SMs)	4	30	7.5
Clock frequency (GHz)	3.2	1.3	0.41
Die size	263	576	2.2
Technology	Intel 45 nm	TSMC 65 nm	1.6
Power (chip, not module)	130	130	1.0
Transistors	700 M	1400 M	2.0
Memory bandwidth (GB/s)	32	141	4.4
Single-precision SIMD width	4	8	2.0
Double-precision SIMD width	2	1	0.5
Peak single-precision scalar FLOPS (GFLOP/S)	26	117	4.6
Peak single-precision SIMD FLOPS (GFLOP/S)	102	311–933	3.0–9.1
(SP 1 add or multiply)	N.A.	(311)	(3.0)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(6.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	(9.1)
Peak double-precision SIMD FLOPS (GFLOP/S)	51	78	1.5

The rightmost column shows the ratios of Nvidia GTX 280 to Intel Core i7. For single-precision SIMD FLOPS on the GTX 280, the higher speed (933) comes from a very rare case of dual issuing of fused multiply-add and multiply. More reasonable is 622 for single fused multiply-adds.

Data-Level Parallelism and GPUs

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions
- Graphics Processing Units (GPU)
- Loop Level Parallelism
- Conclusion

Loop-Level Parallelism (LLP)

- LLP analysis focuses on finding whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependency
    - No dependency; can be executed in parallel

```
for (I=1; I<=1000; I++)
    x[I] = x[I] + y[I];
```

    - Dependency exists; can not execute in parallel

```
for (I=1; I<=1000; I++)
    sum = sum + x[I]
```

- We need to answer two related questions.
  - How to detect loop carried dependency?
  - How to eliminate dependent computations?

Finding Dependency

- Greatest Common Divisor (GCD) test
  - Consider two index values a*i + b and c*i + d
  - Dependency exists if there is no remainder when (d – b) is divided by GCD (c, a).
- Example

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

  - Here, a = 2, b = 3, c = 2 and d =0.
  - GCD(a,c) = 2 and (d-b) = 3.
  - Since 2 does not divide -3, no dependency.

Eliminating Dependent Computations

- Divide the following loop into two loops

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

- Completely parallel component

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

- Partially parallel ("reduction" friendly)

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

  - Assume we have 10 processors. Each processor executes the following code (p ranges from 0 to 9):

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

This loop, which sums up to 1000 elements on each of the 10 processors, is completely parallel. A simple scalar loop can then complete the summation of the last 10 sums

Conclusion

- Vector, SIMD and GPU architectures
  - Exploits data-level parallelism
  - Relies on parallel execution units
- NVIDIA GPUs are popular today
  - Enables real-time multimedia applications
  - CUDA-based parallel programming
  - SIMD-based execution of multiple threads
- Loop-level parallelism
- Please read Section 4.1 – 4.5
- Next, we discuss architecture verification