# Flynn's Taxonomy



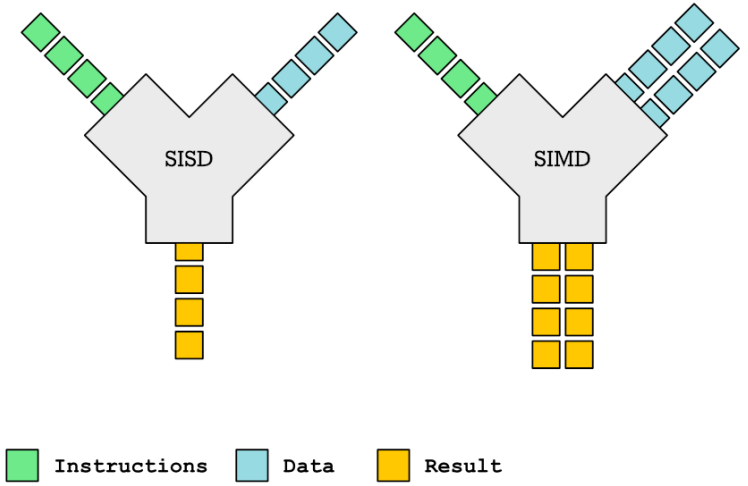
- *SISD* - Single instruction stream, single data stream
- *SIMD* - Single instruction stream, multiple data streams
  - Extend pipelined execution of many data operations, superscalar
  - Simultaneous parallel data operations in most instruction set. E.g. SSE( streaming SIMD extensions), AVX
  - New: *SIMT* – Single Instruction Multiple Threads (for GPUs)
- *MISD* - Multiple instruction streams, single data stream
  - No commercial implementation
- *MIMD* - Multiple instruction streams, multiple data streams
  - Tightly-coupled MIMD (shared memory, NUMAs), OpenMP and Pthreads
  - Loosely-coupled MIMD (distributed memory system, e.g. cluster), MPI

# Advantages of SIMD architectures

1. Can exploit significant data-level parallelism for:
    1. A set of applications has significant data-level parallelism
        Matrix-oriented scientific computing
        Media-oriented image and sound processors
        Tensor-oriented AI applications (training, inference)
2. More energy efficient than MIMD
    1. Only needs to fetch one instruction per multiple data operations, rather than one instr. per data op.
    2. Makes SIMD attractive for personal mobile devices
3. Allows programmers to continue thinking sequentially

# SIMD parallelism

- SIMD architectures
  - A. Vector architectures, extends pipelined execution (SI) of many data operations (MD).
    - SIMD extensions for mobile systems and multimedia applications, multimedia extensions (MMX) in 1996, streaming SIMD extensions (SSE), advanced vector extensions (AVX)
  - B. Graphics Processor Units (SIMT, GPUs)



# Example of vector architecture

- RV64V → RV64G, RV64V extended RV64G with vector instructions, here RV (RISC-V)
  - RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles.
- Vector registers
  - Each register holds a 32-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports, A register file is an array of processor registers in a central processing unit (CPU).
- Vector functional units – FP add and multiply
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general-purpose registers
  - 32 floating-point registers
  - Provide data as input to the vector functional units
  - Compute addresses to pass to the vector load/store unit

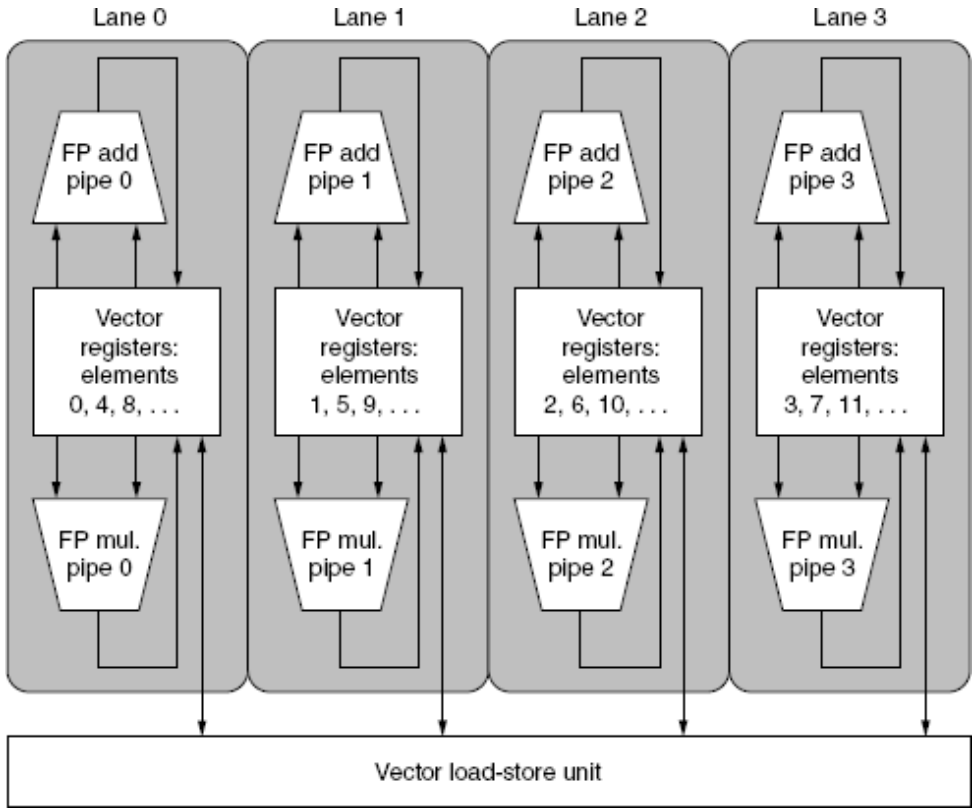
# Optimizations

1. *Multiple Lanes* *（多条车道）* → processing more than one element per clock cycle
2. *Vector Length Registers* *（向量长度寄存器）* → handling non-32 wide vectors
3. *Vector Mask Registers* *（向量遮罩寄存器）* → handling IF statements in vector code
4. *Memory Banks* *（内存组）* → memory system optimizations to support vector processors
5. *Stride* *（步幅）* → handling multi-dimensional arrays
6. *Scatter-Gather* *（分散-集中）* → handling sparse matrices
7. *Programming Vector Architectures* *（向量体系结构编程）* → program structures affecting performance

Question: how these optimizations map to MPI and OpenMP?

# 1. A four lane vector unit

- RV64V instructions only allow element N of one vector to take part in operations involving element N from other vector registers→ this simplifies the construction of a highly parallel vector unit
  - Lane → contains one portion of the vector register elements and one execution pipeline from each functional unit
  - Analog with a highway with multiple lanes!!



# Optimizations

1. *Multiple Lanes* （多条车道）→ processing more than one element per clock cycle
2. *Vector Length Registers* （向量长度寄存器）→ handling non-32 wide vectors
3. *Vector Mask Registers* （向量遮罩寄存器）→ handling IF statements in vector code
4. *Memory Banks* （内存组）→ memory system optimizations to support vector processors
5. *Stride* （步幅）→ handling multi-dimensional arrays
6. *Scatter-Gather* （分散-集中）→ handling sparse matrices
7. *Programming Vector Architectures* （向量体系结构编程）→ program structures affecting performance

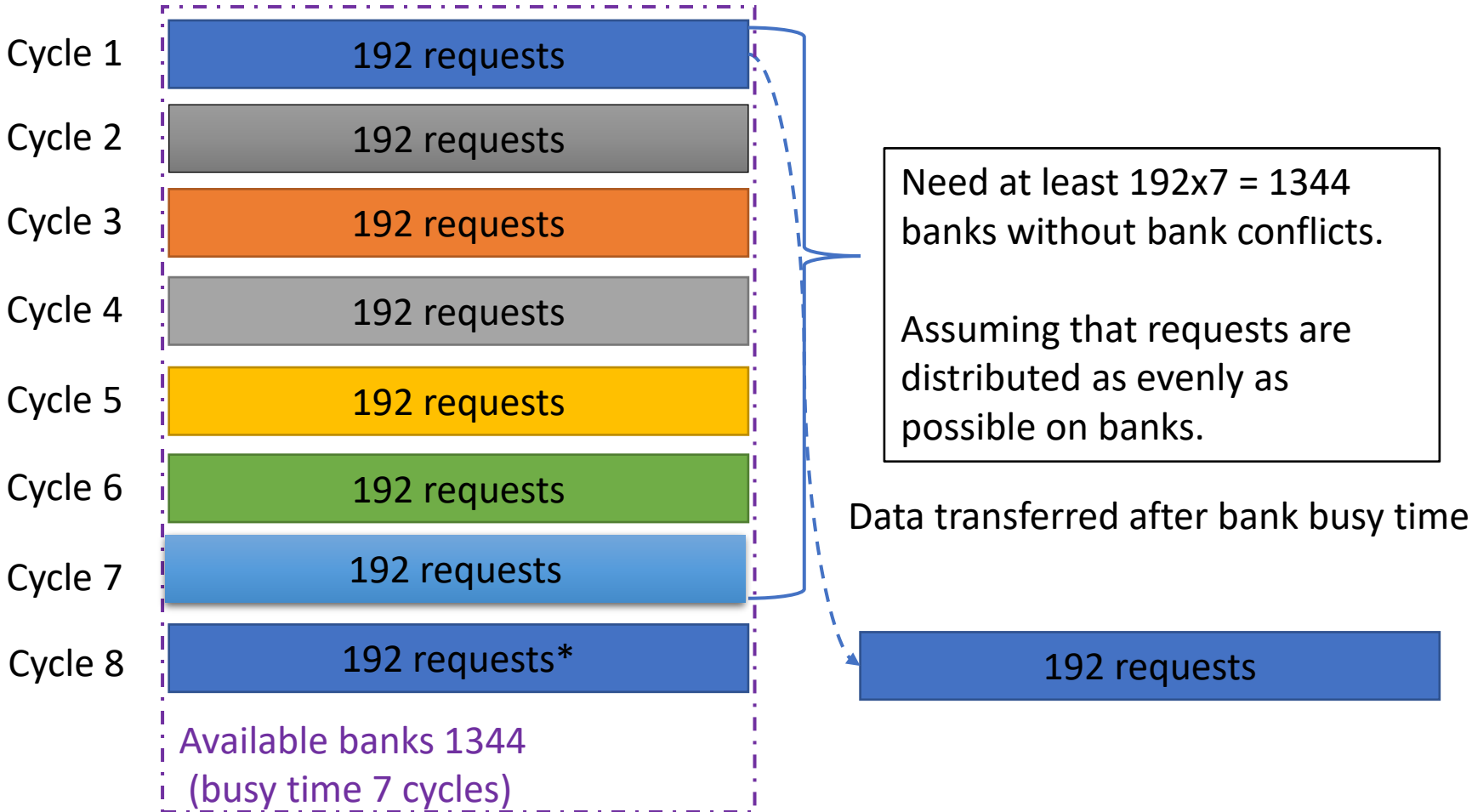
Question: how these optimizations map to MPI and OpenMP?

# 4. Memory banks

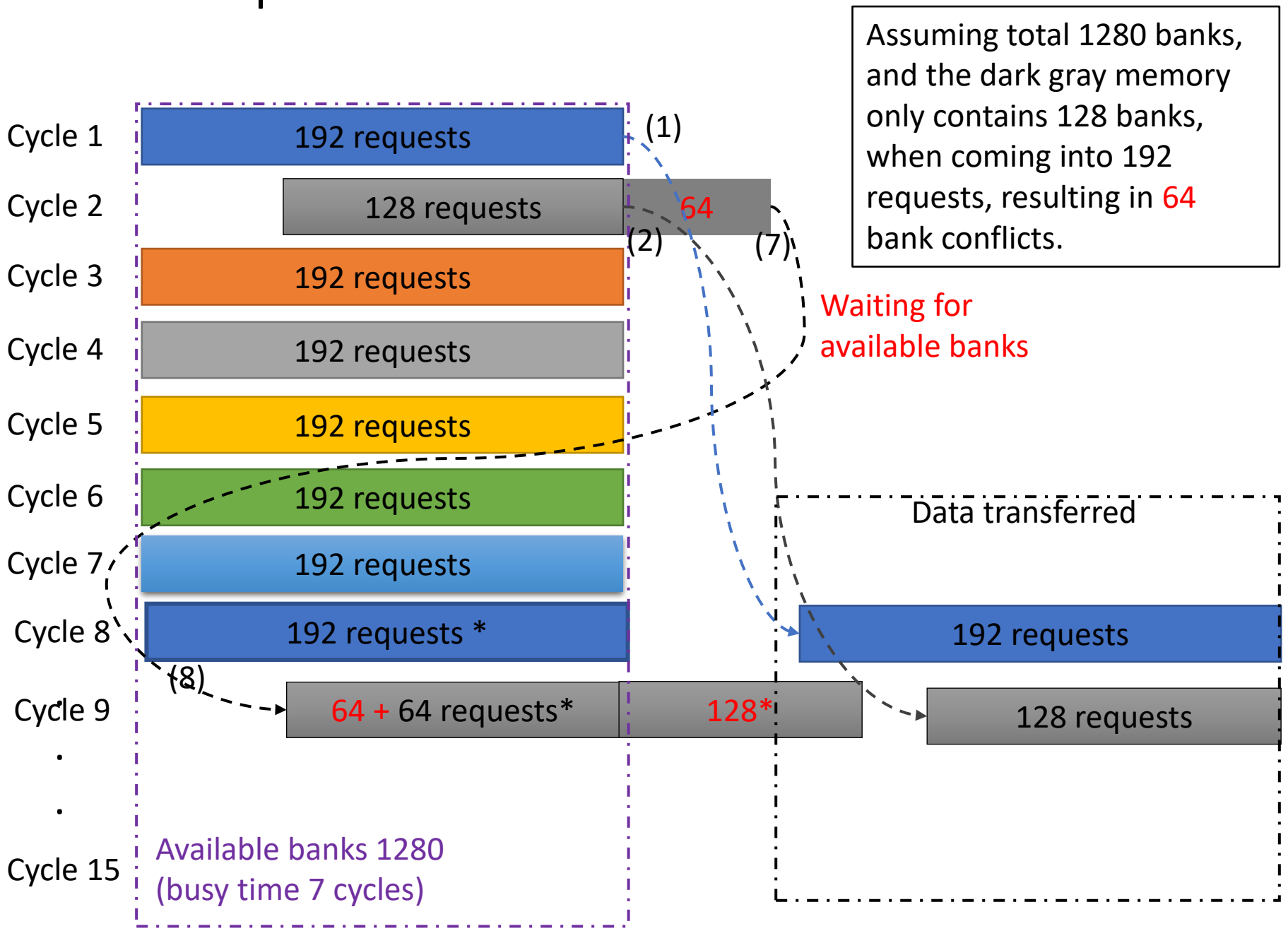
- Memory system must be designed to support high bandwidth for vector loads and stores
  - Memory start-up time: get the first word from memory in to a register
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non-sequential words
  - Support multiple vector processors sharing the same memory
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, DRAM bank busy time (and bank latency) is 15 ns or about 67 M per second.
  - How many minimum memory banks needed to allow all processors to run at the full memory bandwidth (no bank conflicts)?
    - 32 processors x 6 =192 memory accesses,
    - 15ns DRAM cycle /2.167ns processor cycle≈7 processor cycles
    - 7 x 192 →1344!

# Memory Banks & Full Memory Bandwidth

- No bank conflict is a critical condition for achieving full memory bandwidth.



- Example for bank conflict



# Theoretical Memory Bandwidth

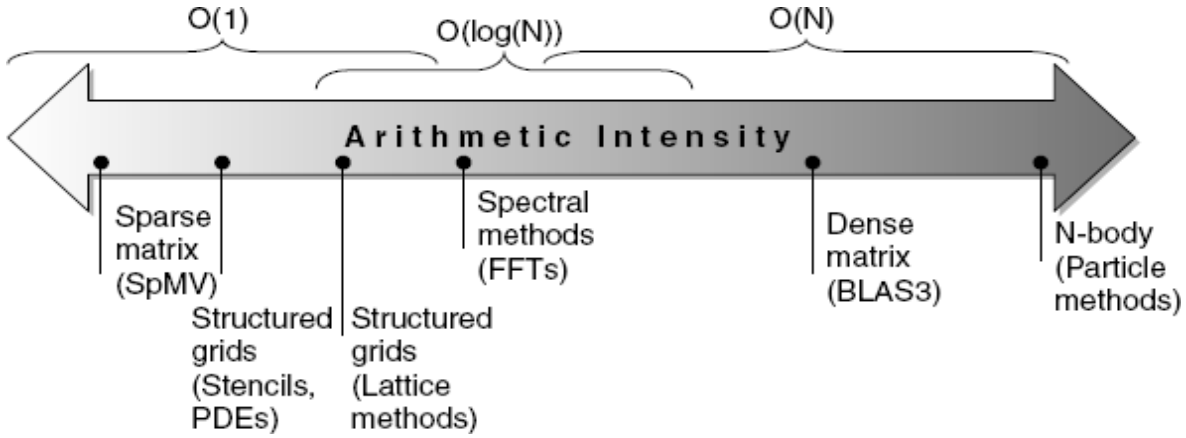
- DRAM clock frequency
- Number of data transfers per clock: Two, in the case of "double data rate" (DDR, DDR2, DDR3, DDR4) memory.
- Memory bus (interface) width: Each DDR, DDR2, or DDR3 memory interface is 64 bits wide. Those 64 bits are sometimes referred to as a "line."
- Number of interfaces: Modern personal computers typically use two memory interfaces (dual-channel mode) for an effective 128-bit bus width.
- For example, a computer with dual-channel memory and one DDR2-800 module per channel running at 400 MHz would have a theoretical maximum memory bandwidth of:
- 400,000,000 clocks per second × 2 transfers per clock × 64 bits per line × 2 interfaces = 102,400,000,000 (102.4 billion) bits per second (in bytes, 12,800 MB/s or 12.8 GB/s)

# Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

# Roofline performance model

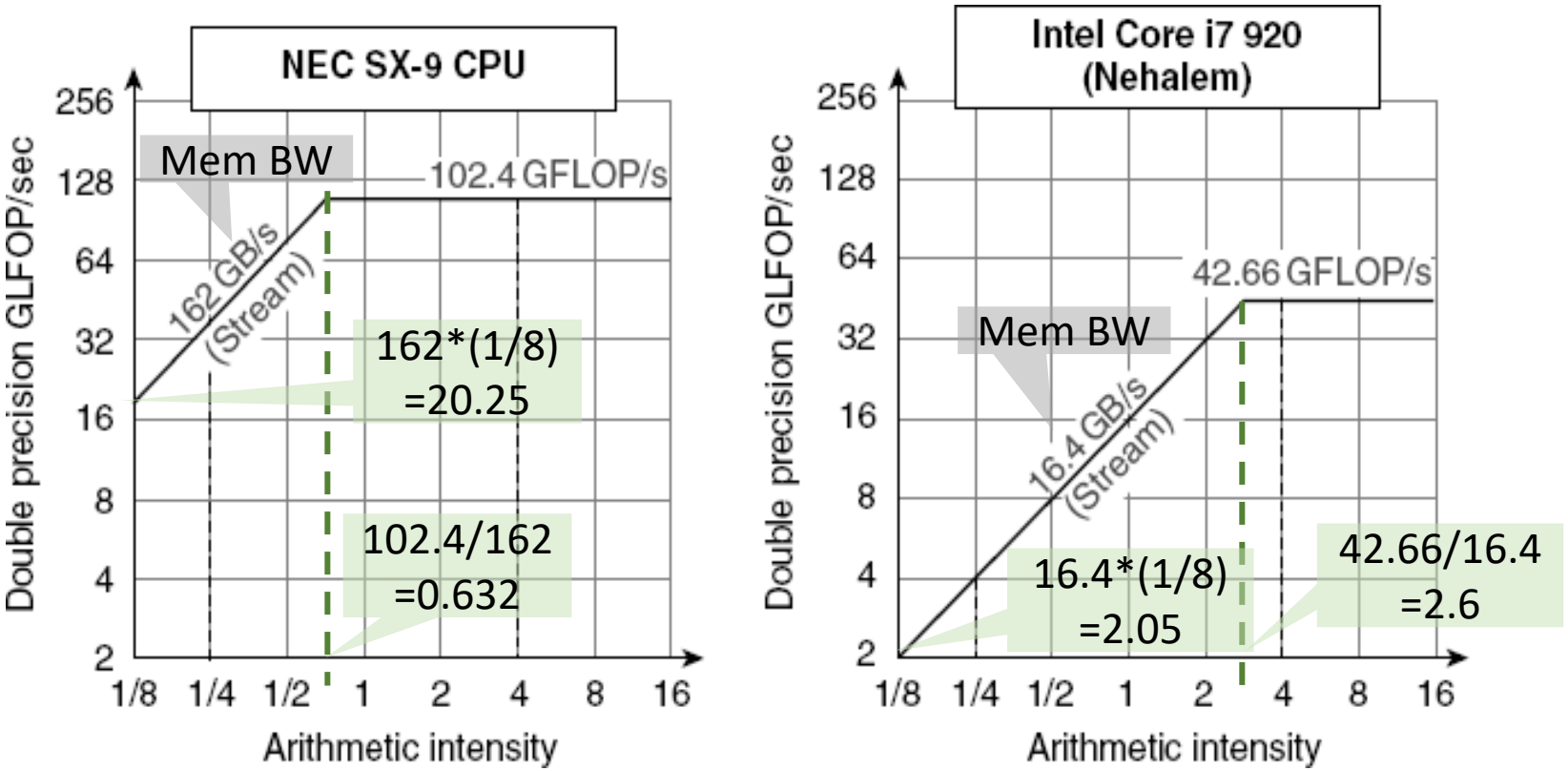
- Basic idea:
  - Peak floating-point performance as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance
- *Arithmetic intensity* (*计算密度、计算访存比*)→ Floating-point operations per byte read c
- Max Attainable GFLOPs/sec = Min (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Performance)
  Stream is a benchmark for memory performance




- Dense matrix operations scale with problem size but sparse matrix operations do not!!

# Examples

Roofline: on the sloped portion of the roof the performance is limited by the memory bandwidth, on the flat portion it is limited by peak floating point performance



# C. Graphical Processing Unit - GPU

- Given the hardware invested to do graphics well, how can it be supplemented to improve performance of a wider range of applications (General-purpose GPU)?
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
    - Compute Unified Device Architecture (CUDA)
    - OpenCL for vendor-independent language
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model: “Single Instruction Multiple Thread” (SIMT)

# Threads, blocks, and grid (CUDA terminology)

- A *CUDA thread* is associated with each data element
  - *CUDA threads* ➔ thousands of threads are utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP
- CUDA threads with index are organized into *thread blocks*
  - *Thread Blocks*: groups of up to 512 elements
  - *Executed on Multithreaded SIMD Processor*: hardware that executes a whole thread block (32 elements executed per SIMD thread at a time)
- Thread Blocks with index are organized into a *grid*
  - Blocks are executed independently and in any order
  - Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory
- CUDA thread management handled by GPU hardware and CUDA runtime not by applications or OS
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler (Hardware)

# Terms used in this chapter to official NVIDIA/CUDA

Academic Industry

| Type | Descriptive name | Closest old term outside of GPUs | Official CUDA/NVIDIA GPU term | Short explanation |
|---|---|---|---|---|
| Program abstractions | 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 | A Thread of SIMD Instructions (SIMD Thread) | Thread of Vector Instructions | Warp | A traditional thread, but it only contains 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 |

# Terms used in this chapter to official NVIDIA/CUDA

| | | | | |
|---|---|---|---|---|
| Processing hardware | 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 | 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) |

# Example: multiply two vectors of length 8192

- C code version:

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n = 8192, double a, double *A, double *B, double *C)
{
        for (int i = 0; i < n; ++i)
        A[i] = B[i] * C[i];
}
```

Q: Principle differences between C SIMD and CUDA SIMT?

- CUDA version:
- It seems that we are programming on a CUDA thread at element level. (Abstraction: SIMT)

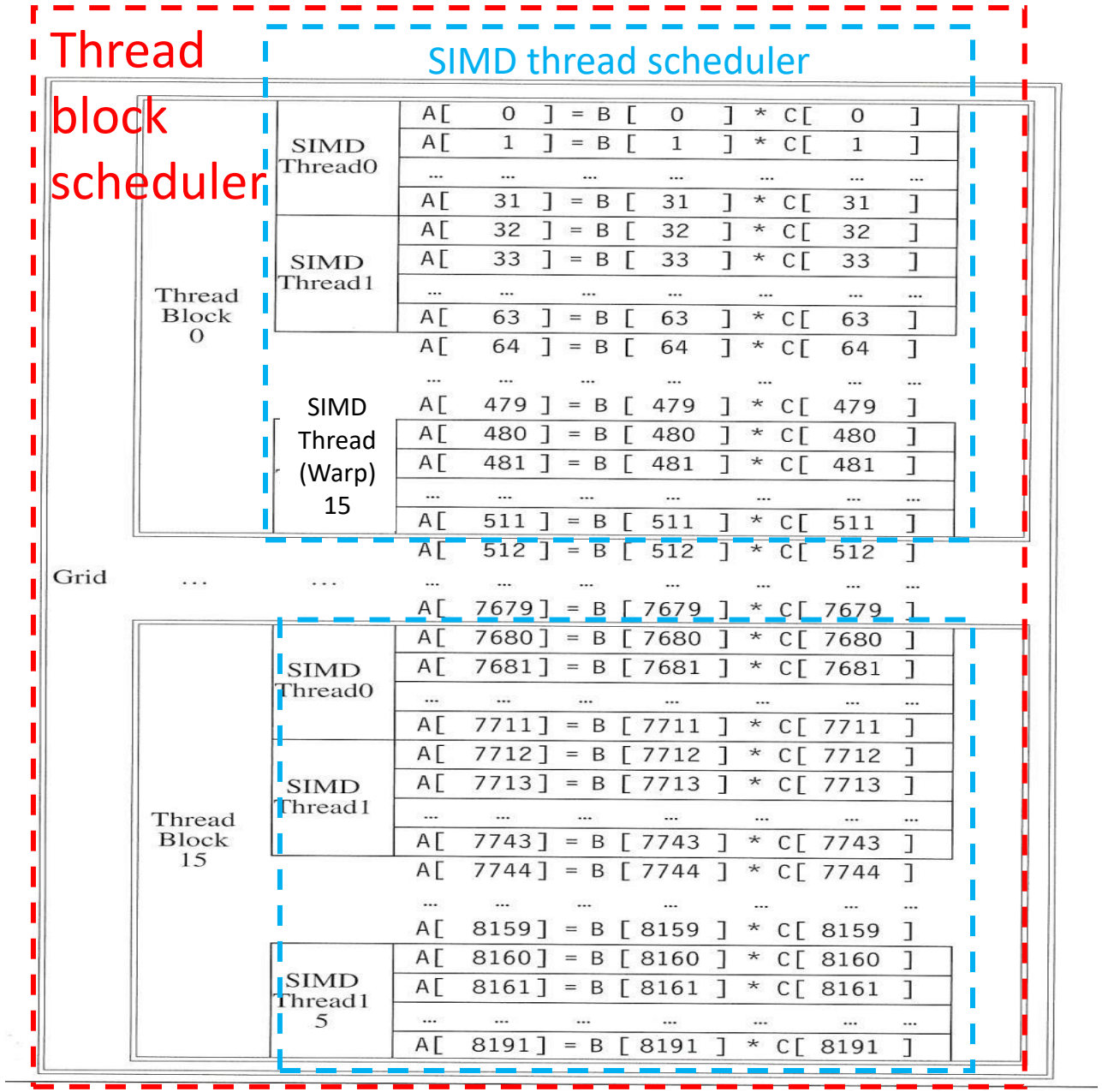
```
// Invoke DAXPY with 512 threads per Thread Block
// code in C
int nblocks = (n+ 511) / 512;
daxpy<<<nblocks, 512 >>>(n, 2.0, A, B, C);
// DAXPY in CUDA

__global__  void daxpy (int n, double a, double *A,
                        double *B, double *C)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n)
                A[i] = B[i] * C[i];
}
```

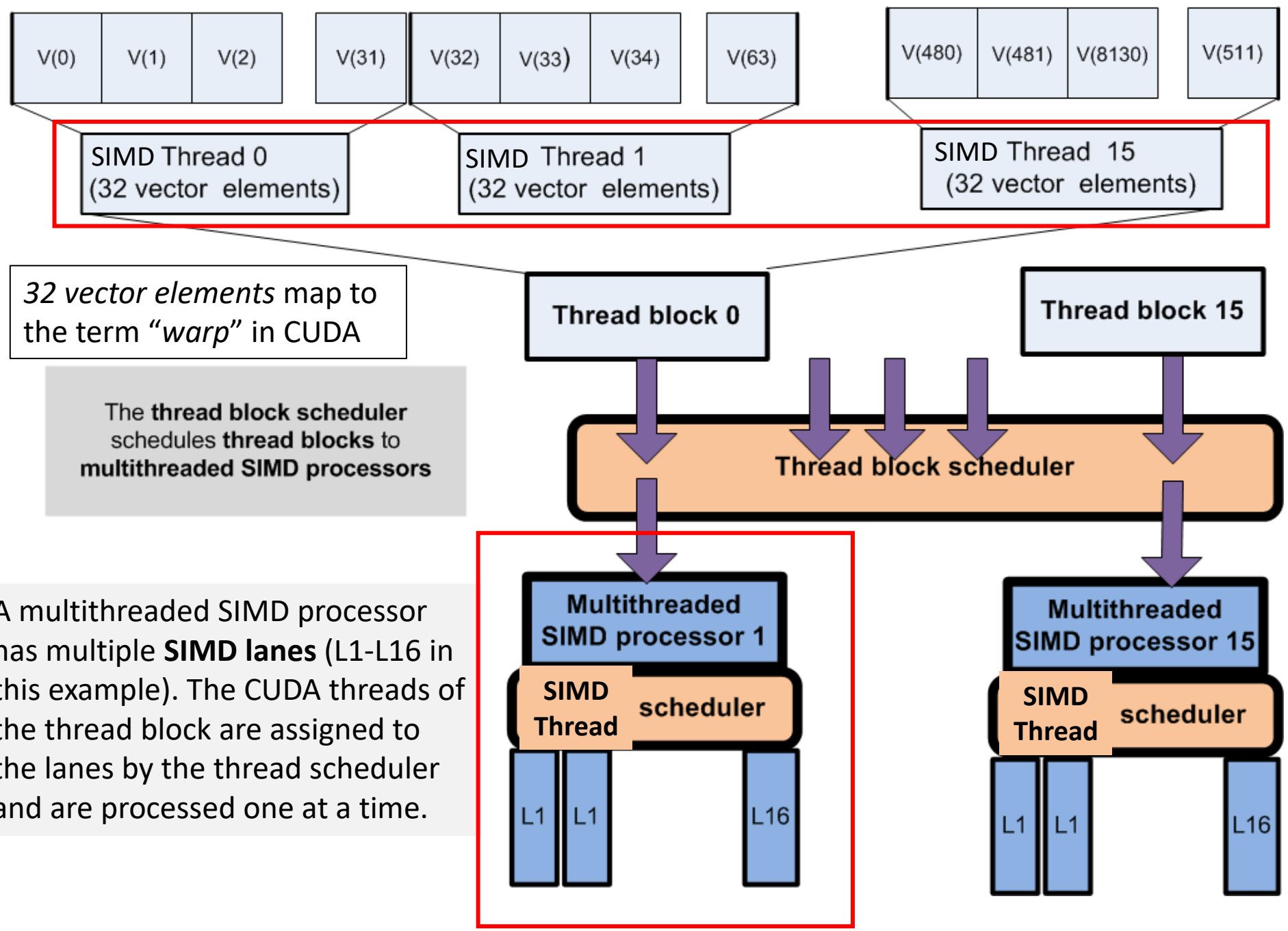
# Example: multiply two vectors of length 8192

- Grid → Code that works over all elements
- Thread block→ analogous to a vector loop with a vector length of typically up to 512. Breaks down the vectorized loop into manageable set (warp) of 32 CUDA threads
  - 32 elements (CUDA threads) per SIMD thread (Warp) x 16 SIMD threads=512 elements per thread block
  - → 512 elements (CUDA threads) per thread block, SIMD instruction executes 32 elements at a time
  - Grid size = 8192 vector elements / 512 elements per block = 16 thread blocks
- *Thread block scheduler* → assigns a thread block to a *multithreaded SIMD processor*
- Current-generation GPUs (Pascal P100) have 56 multithreaded SIMD processors (Streaming processors)

# Threads, blocks, and grid example

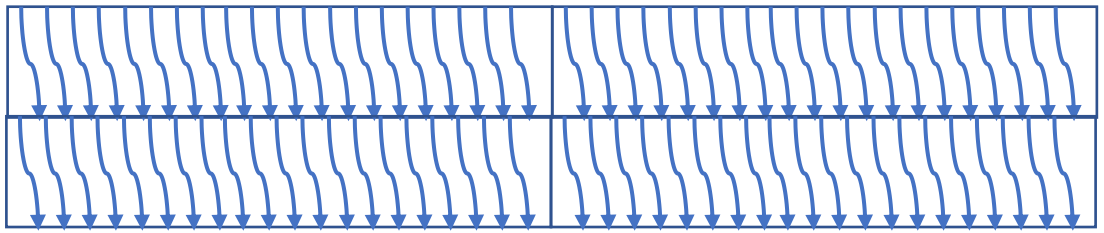


2
2

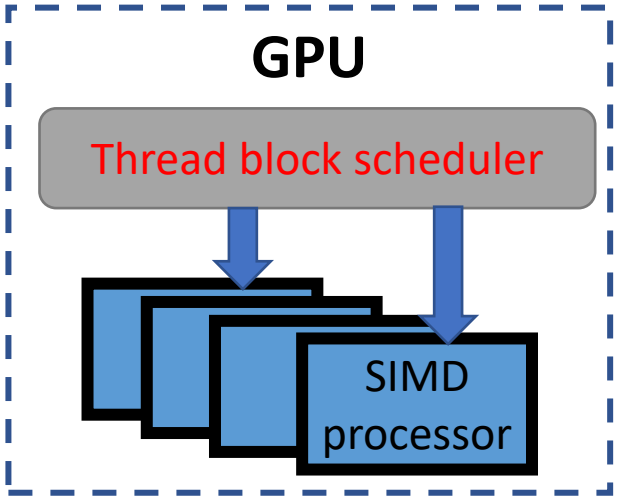



A multithreaded SIMD processor has multiple **SIMD lanes** (L1-L16 in this example). The CUDA threads of the thread block are assigned to the lanes by the thread scheduler and are processed one at a time.

# Terms in CUDA Prog. Model vs GPU Architecture

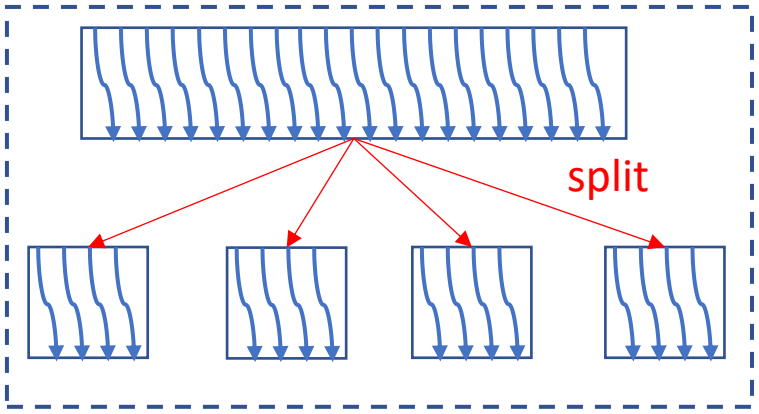


CUDA Grid contains 4 thread blocks

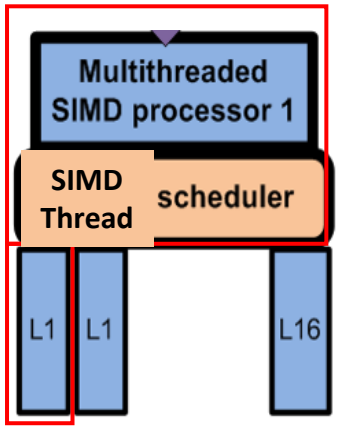



CUDA
Thread
Block

CUDA
Warps




Warps of a CUDA thread block can be viewed as SIMD threads in GPU architecture concurrently running on a SIMD processor and scheduled by SIMD thread scheduler.




CUDA Thread

SIMD processor's lane

# Threads, blocks, and grid example

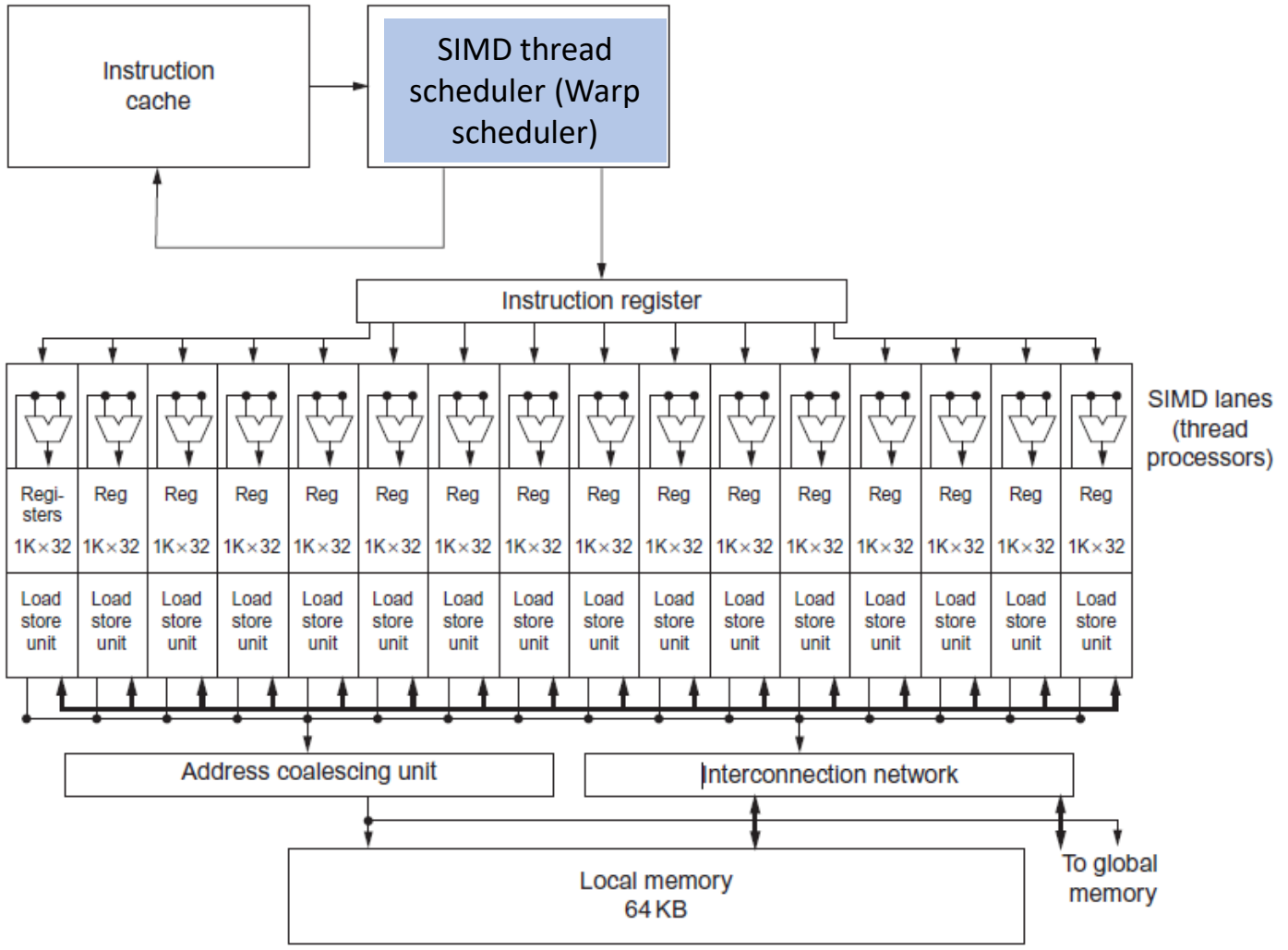



**Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor.** It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

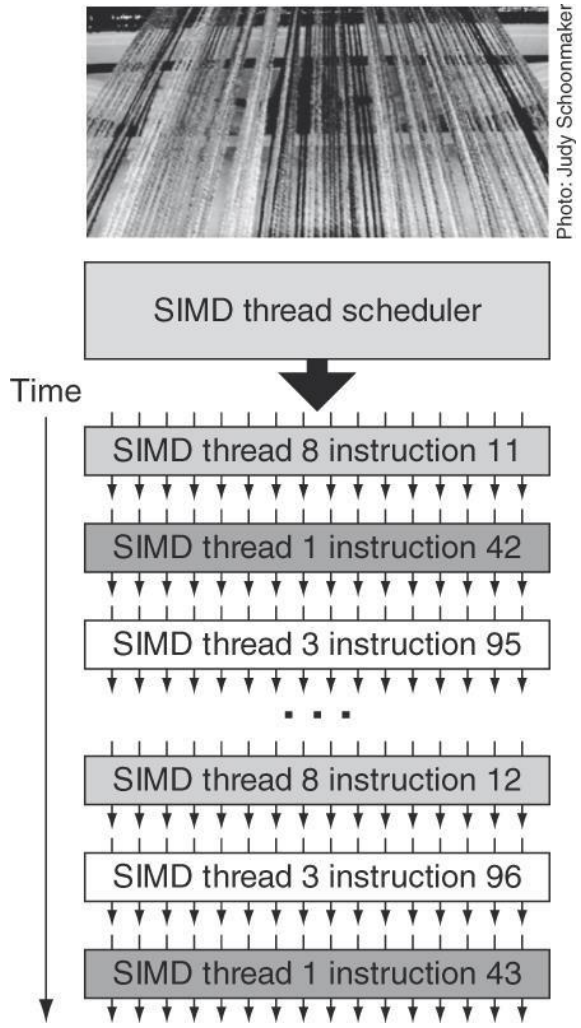



**Figure 4.16 Scheduling of threads of SIMD instructions.** The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

# NVIDIA GPU memory structures



- Each SIMD Lane (CUDA Thread) has private section of *off-chip* DRAM
  - "Private memory", not shared by any other lanes (CUDA Thread)
  - Contains stack frame, register spilling, and private variables that don't fit in the registers.
  - Recent GPUs cache this private memory in L1 and L2 caches
- Each multithreaded SIMD processor also has local memory (Shared Memory) that is *on-chip*
  - Shared by SIMD lanes / threads *within a block only*
  - Latency is 100x lower than GPU Memory.
  - Threads can access data in local memory loaded from global memory by other threads within the same thread block
- The *off-chip* memory shared by SIMD processors is *GPU* memory (Global memory)
  - Host can read and write GPU memory

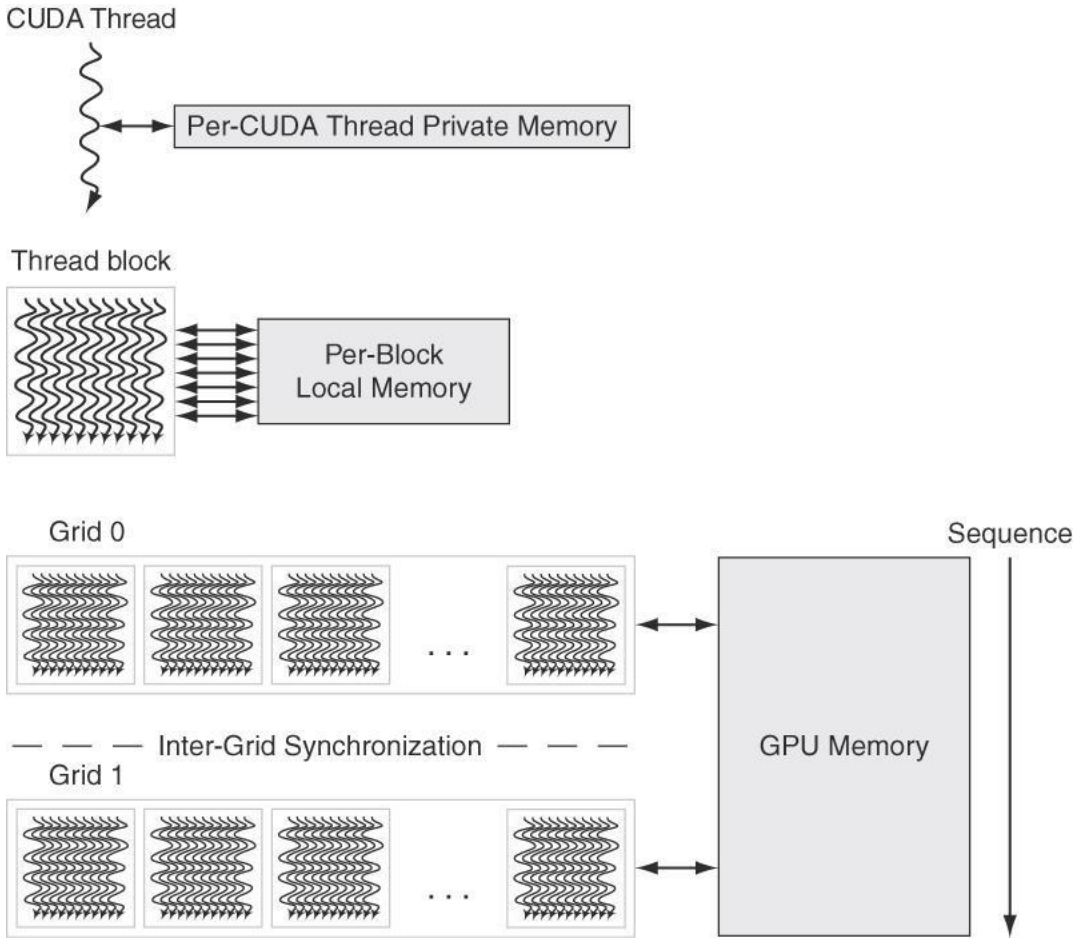



**Figure 4.18 GPU Memory structures.**

- *GPU Memory* (*Global Memory*) shared by all Grids (vectorized loops),
- *Local Memory* (*Shared Memory*) shared by all threads of SIMD instructions within a thread block (body of a vectorized loop).
- *Private Memory* private to a single CUDA thread.

# Terminology (GPU hardware)

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to e.g. 48 threads of SIMD instructions
    - Hides memory latency
- *Thread block scheduler*
  - Schedule thread blocks to SIMD processors, each thread block is split to manageable sets (warps) (e.g. 32 CUDA threads / warp) and assigned to SIMD threads running on a SIMD processor.
- Within each SIMD processor:
  - 16 or 32 SIMD lanes
  - Wide and shallow compared to vector processors

# Conditional branching

- GPU branch hardware uses:
  - Internal mask registers
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - i.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

# Example

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

with *Push, *Comp, *Pop indicating the branch synchronization markers inserted by the PTX assembler that push the old mask, complement the current mask, and pop to restore the old mask

```
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        ;bitwise 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
```

Q. How the PTX codes are executed on GPU?

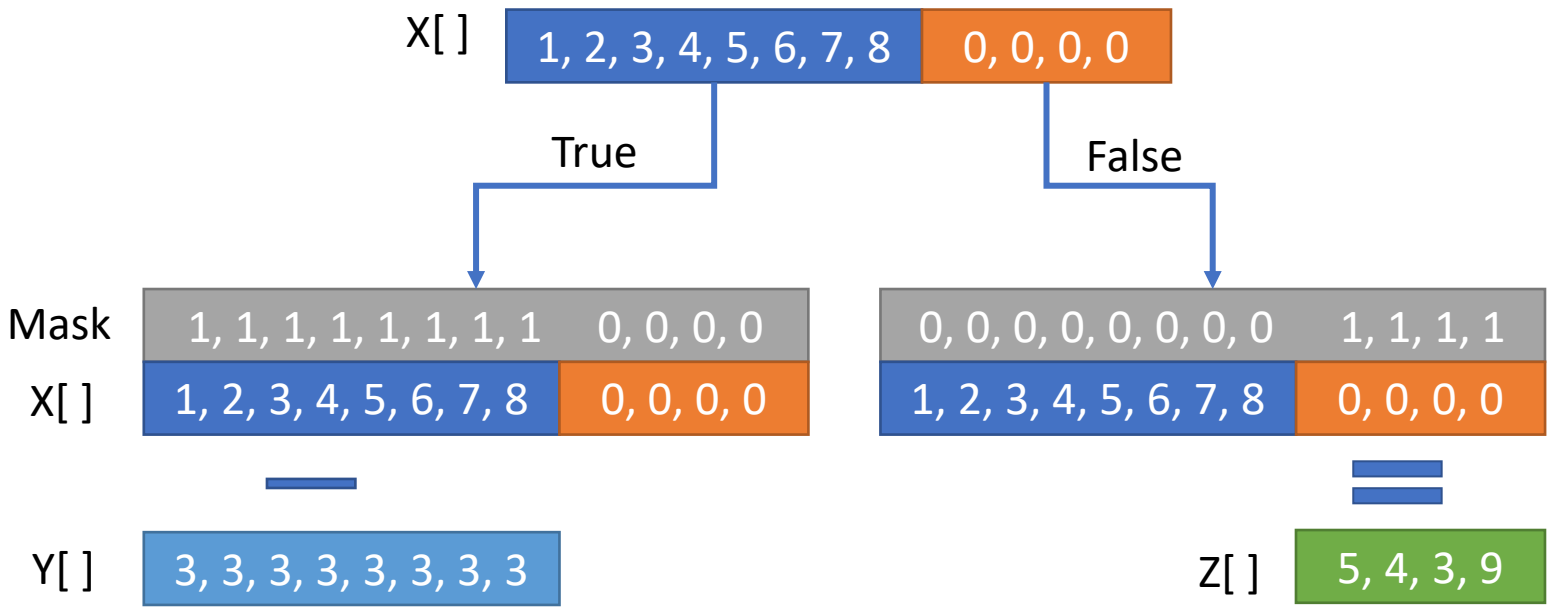
**SIMT**

*Note: a thread has 64 vector components, each a 32 bit floating point*

# Example

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

with *Push, *Comp, *Pop indicating the branch synchronization markers inserted by the PTX assembler that push the old mask, complement the current mask, and pop to restore the old mask




Q. How the PTX codes are executed on GPU?

**SIMT**

# Pascal architecture innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - Two sets of 16 SIMD lanes (SIMD instruction width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast single-precision, double-precision, and half-precision floating-point arithmetic: The single precision floating-point is up to 10 TeraFLOP/s. Double-precision is at 5 TeraFLOP/s, and half-precision is about at 20 TeraFLOP/s when expressed as 2-element vectors.
- High-bandwidth memory: Stacked, high-bandwidth memory (HBM2) has a wide bus with 4096 data wires running at 0.7 GHz offering a peak bandwidth of 732 GB/s, which is more than twice as fast as previous GPUs.
- High-speed chip-to-chip interconnect: Pascal GP100 introduces the NVLink communication channel that supports data transfers of up to 20 GB/s in each direction.

# Loop-level parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

No loop-carried dependence

# Loop-level parallelism example 2

```
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 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration
- There are two different dependences:
- 1. S1 uses a value computed by S1 in an earlier iteration, because iteration i computes A[i+1], which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1]. (loop-carried)
- 2. S2 uses the value A[i+1] computed by S1 in the same iteration. (no loop-carried)

# Loop-level parallelism example 3

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

S1 uses value computed by S2 in previous iteration but dependence is not circular， neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences because the absence of a cycle means that the dependences give a partial ordering on the statements. so loop is parallel. Transform to:

```
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];
```

# Loop-level parallelism examples 4 and 5

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

- No loop-carried dependence

```
for (i=1;i<100;i=i+1)  {
        Y[i] = Y[i-1] + Y[i];
}
```

- Loop-carried dependence in the form of *recurrence*

# Reductions

- Reduction Operation:

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

- Transform to…

```
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];
```

- Do on p processors:

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

- Note: assumes associativity!