

Data-Level Parallelism in Vector, SIMD and GPU Architectures

(Chapter 4)

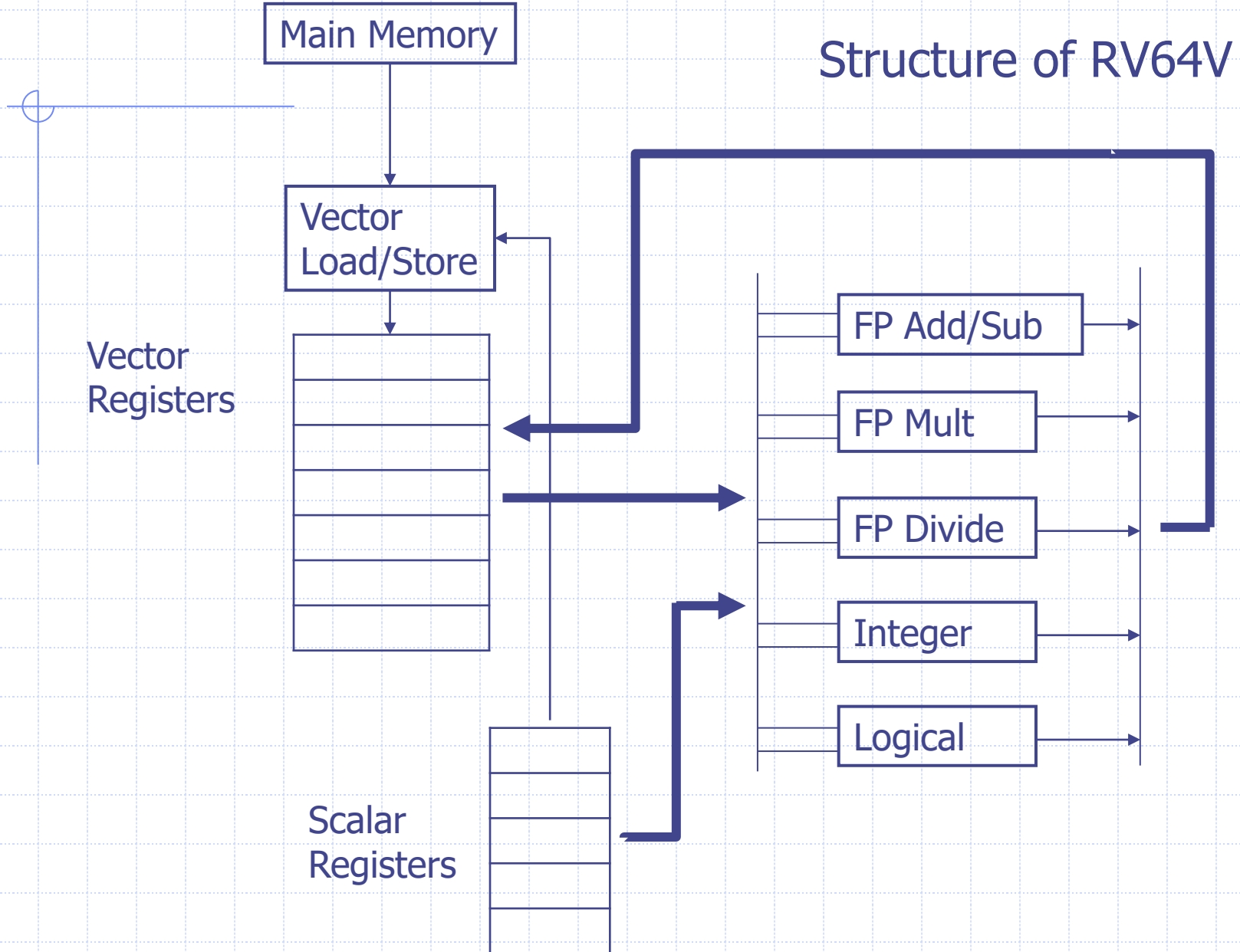
Outline

- ◆ Vector Architecture
- ◆ SIMD Instruction Set Extensions for Multimedia
- ◆ Graphics Processing Units

Vector Architecture

- ◆ In a vector machine, a single instruction operates on vectors of data.
- ◆ Basic Idea
 - Read sets of data elements into “vector registers”
 - Operate on those registers
 - Disperse the results back into memory

◆ An example of vector machine- RV64V



The RV64V contains vector registers, vector functional units, vector load/store unit, and a set of scalar registers.

1. Vector registers: RV64V has eight vector registers, and each vector register holds 64 elements, each 64 bits wide.
2. Vector functional units: Each unit is fully pipelined, and it can start a new operation on every clock cycle.
3. Vector load/store unit: The vector memory unit loads or stores a vector to or from memory.
4. A set of scalar registers: Scalar registers can also provide data as input to the vector functional unit, as well as compute addresses to pass to the vector load/store unit.

◆ RV64V Instructions

- vld v0, x1: Load vector register v1 from memory starting at address stored in x1
- vst v1, x1: Store vector register v1 into memory starting at address stored in x1
- vadd v1,v2,v3: Add elements of v2 and v3, and put each result in v1.
- vadd v1,v2,f0: Add f0 to each element of v2, then put each result in v1.
- vmul v1,v2,v3: Multiply elements of v2 and v3, and put each result in v1.
- vmul v1,v2,f0: Multiply each element of v2 by f0, then put each result in v1.

◆ Example:

⊕ Consider the following vector problem

$$Y = a \times X + Y,$$

where X and Y are vectors, and a is a scalar.

The RV64V code is given by

fld	f0,a	; load scalar a
vld	v0,x5	; load vector X
vmul	v1,v0,f0	; vector-scalar multiply
vld	v2,x6	; load vector Y
vadd	v3,v1,v2	; add
vst	v3,x6	; store the result

◆ Vector Execution Time

The execution time of a sequence of vector operations depends on three factors:

1. The length of operand vectors,
2. Structure hazards among the operations,
3. Data dependence.



We also assume RV64V functional units consume one element per clock cycle.

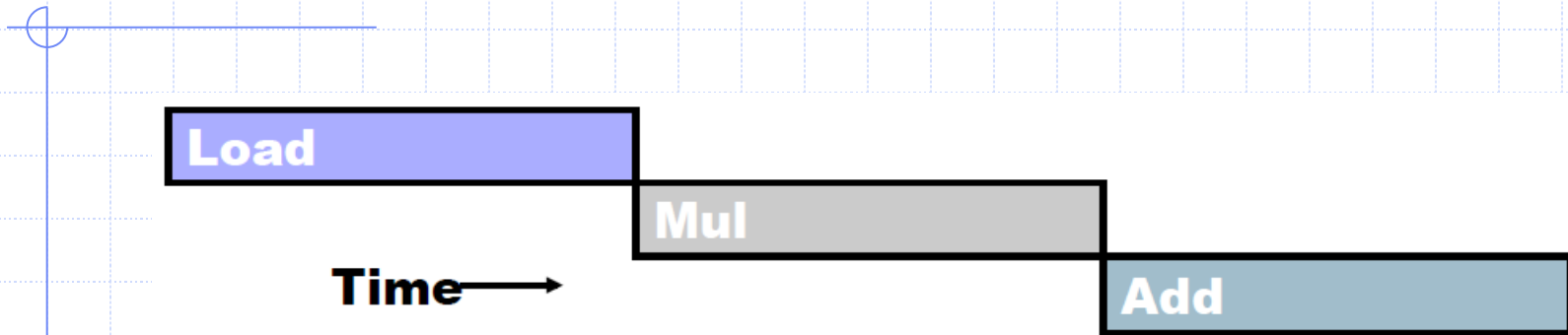
Therefore, execution time for a single vector instruction is approximately the vector length.

We define **convoy** as a set of vector instructions that could potentially execute together.

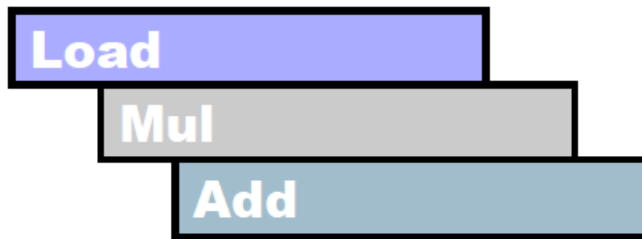
Sequences with read-after-write dependency hazards can be in the same convoy via **chaining**.

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

Without chaining, we must wait for last element of result to be written before starting dependent Instruction.



With chaining, we can start dependent instruction as soon as first result appears.



We define a **chime** as the unit of time to execute one convoy.

Therefore, m convoys execute in m chimes.

Suppose the vector dimension is n . Each chime is n clock cycles.

The execution of m convoys then needs $m \times n$ clock cycles.

◆ Example:

Consider the following example.

fld	f0,a	; load scalar a
vld	v0,x5	; load vector X
vmul	v1,v0,f0	; vector-scalar multiply
vld	v2,x6	; load vector Y
vadd	v3,v1,v2	; add
vst	v3,x6	; store the result

There are three convoys:

1. vld and vmul
2. vld and vadd
3. vst

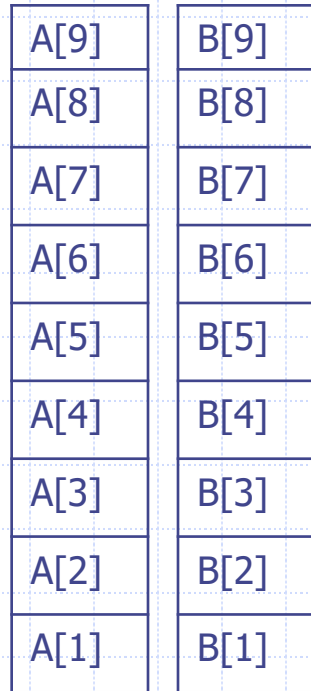
There are 3 chimes.

For 64 element vectors, requires $64 \times 3 = 192$ clock cycles

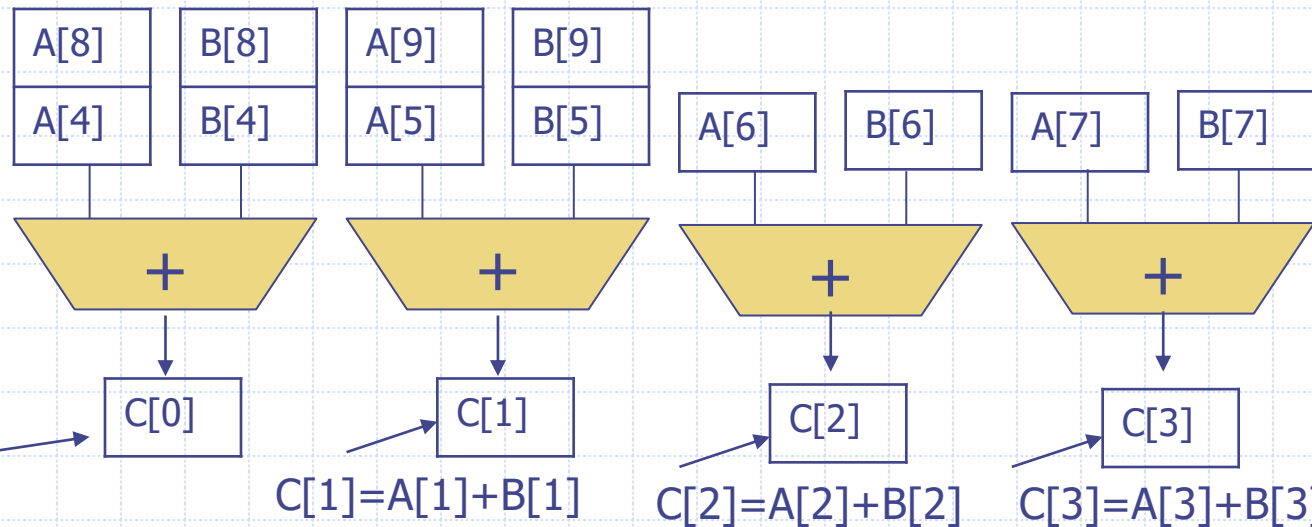
◆ Multiple lanes

The number of **lanes** indicates the number of parallel pipelines in the vector functional unit. So far we only considered single-lane vector functional unit. The extension of RV64V for multiple lanes is also possible.

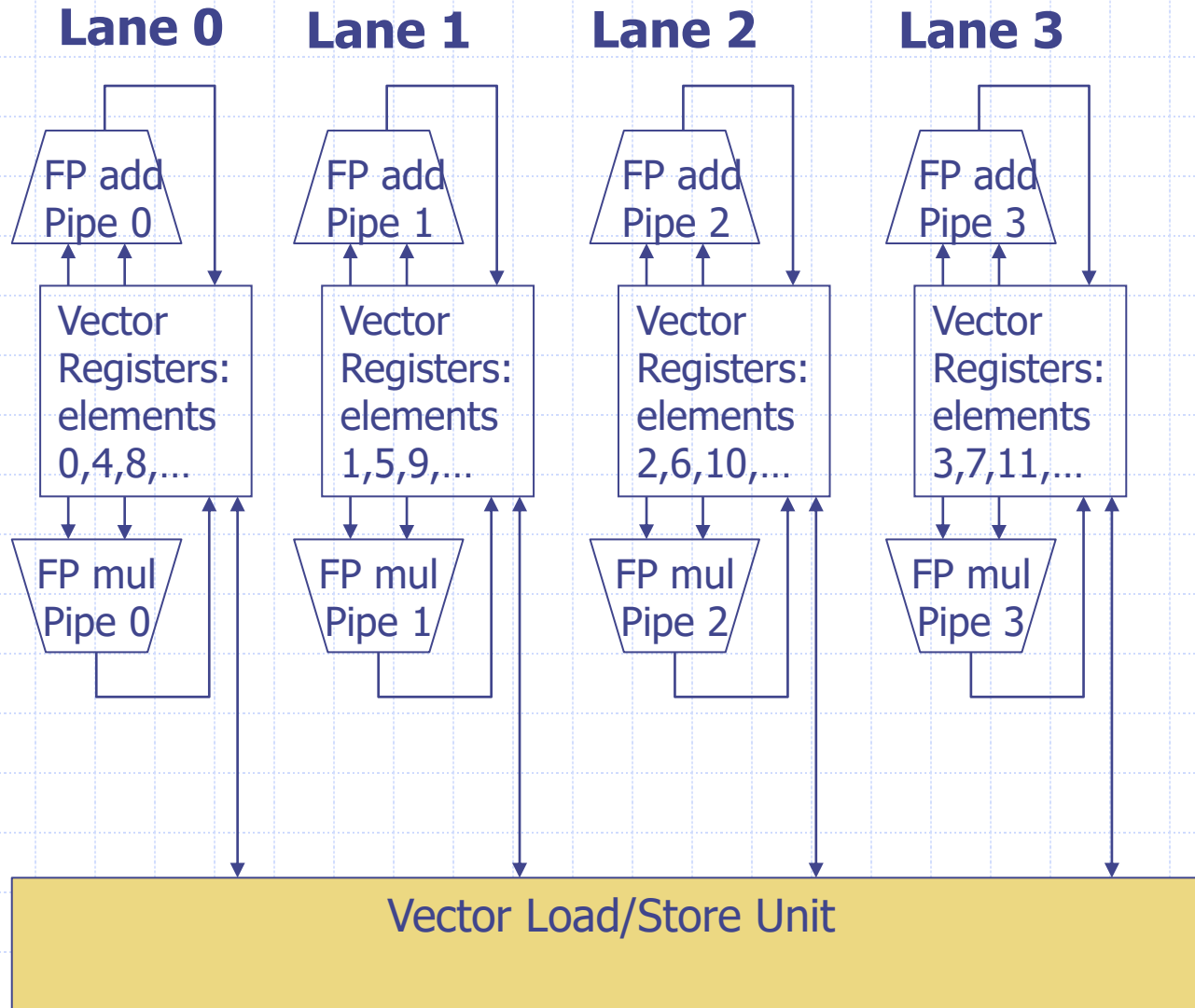
Single-lane for $C \leftarrow A+B$

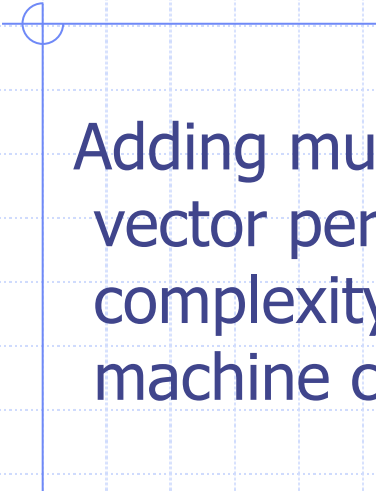


4-lane For $C \leftarrow A+B$



Architecture of 4-lane vector functional unit: the functional unit consists of 4 adders, and 4 multipliers.





Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code.

It also allows designers to trade off die area, clock rate, voltage, and energy without sacrificing peak performance. If the clock rate of a vector processor is halved, doubling the number of lanes will retain the same potential performance.

SIMD Instruction Set Extensions for Multimedia

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.

For example, many graphics systems used 8 bits to represent each of the three primary colors.

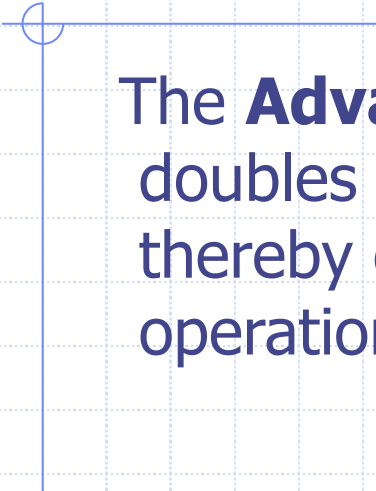
◆ Basic Idea

By partitioning the carry chain within a 256-bit adder, a processor could perform simultaneous operations on short vectors of thirty-two 8-bit operands, sixteen 16-bit operands, eight 32-bits operands, or four 64-bits operations

◆ Implementations

For the **x86 architecture**, the **MMX instructions** added in 1996 repurposed the 64-bit floating-point registers, so that the basic instructions could perform eight 8-bit operations, or four 16-bit operations simultaneously.

The Streaming SIMD Extensions (SSE) successor in 1999 added separate registers that were 128 bits wide, so now instructions could simultaneously perform sixteen 8-bit operations, eight 16-bit operations, or four 32 bit operations.



The **Advanced Vector Extension (AVX)**, added in 2010, doubles the width of the registers again to 256 bits and thereby offers instructions that double the number of operations on all narrower data types.

◆ Examples

Here we add 256-bit SIMD multimedia instructions to RISC-V. We add the suffix “4D” on instructions that operate on Four double-precision operands at once.

Again, consider the following vector problem

$$Y = a \times X + Y,$$

where X and Y are vectors, and each contains 4 double precision numbers.

Assume all the F registers (i.e.,f0-f32) have 256 bits.

	fld	f0,a	;load scalar a
	splat.4D	f0,f0	; Make 4 copies of a
	addi	x28,x5,#256	;last address to load
Loop:	fld.4D	f1,0(x5)	;load X[i], X[i+1], X[i+2], X[i+3]
	fmul.4D	f1,f1,f0	;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
	fld.4D	f2,0(x6)	;load Y[i], Y[i+1], Y[i+2], Y[i+3]
	fadd.4D	f2,f2,f1	;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
	fsd.4D	f2,0(x6)	;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
	addi	x5,x5,#32	;increment index to X
	addi	x6,x6,#32	;increment index to Y
	bne	x28,x5,Loop	;check if done

Note that in this example, the SIMD instruction

`fld.4D f1,0(x5)`

Actually performs $f1 \leftarrow \{\text{Mem}[x5], \dots, \text{Mem}[x5+31]\}$
(32 bytes, 256 bits),
where

$X[i] = \{\text{Mem}[x5], \dots, \text{Mem}[x5+7]\}$ (8 bytes, 64 bits)

$X[i+1] = \{\text{Mem}[x5+8], \dots, \text{Mem}[x5+15]\}$ (8 bytes, 64 bits)

$X[i+2] = \{\text{Mem}[x5+16], \dots, \text{Mem}[x5+23]\}$ (8 bytes, 64 bits)

$X[i+3] = \{\text{Mem}[x5+24], \dots, \text{Mem}[x5+31]\}$ (8 bytes, 64 bits)

Graphics Processing Units

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

One solution is to develop a **general purpose** Graphical Processing Unit (GPU).

◆ Basic Idea

1. A heterogeneous execution model is proposed.
CPU is the host, GPU is the device.
2. A C-like programming language is developed for GPU.
A typical example is **CUDA** (Compute Unified Device Architecture) provided by NVIDIA.
3. All forms of GPU parallelism is unified as **CUDA thread**.
4. CUDA Programming model is “**Single Instruction Multiple Thread (SIMT)**”

◆ Thread and Blocks in CUDA

1. A **thread** is associated with each data element.
2. Threads are organized into **blocks**.
3. Blocks are organized into a **grid**.
4. GPU hardware handles thread management, not applications or OS.
5. The hardware that executes a whole **block** of threads is called a **streaming multiprocessor (SM)**.

◆ Some CUDA details

1. To distinguish between functions for the GPU (device) and functions for the CPU, CUDA uses `_global_` or `_device_` for the former, and `_host_` for the later.
2. The syntax of function call for the function *name* that runs on the GPU is

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

where *dimGrid* and *dimBlock* specify the dimensions of a grid (in blocks), and the dimensions of a block (in threads).

3. Other important keywords include

`blockIdx`: the identifier for blocks,
`threadIdx`: the identifier for threads per block,
`blockDim`: number of threads per block (comes from `Dimblock`).

◆ Example


Let's start with conventional C code for the following vector problem

$$Y = a \times X + Y.$$

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

Below is the CUDA version. It consists of a host code and a device code.

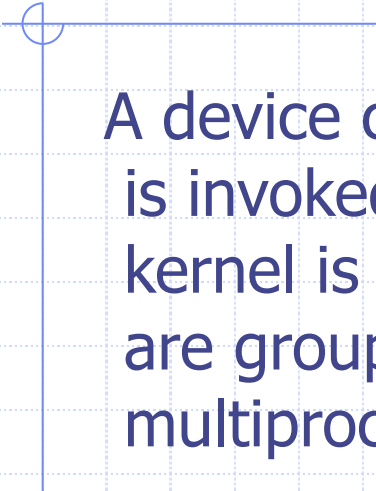
```
_host_  
int nblocks = (n+255)/256;  
daxpy<<<nblocks,256>>>(n,2.0,x,y);  
  
_global_  
void daxpy(int n, double a, double *x, double *y);  
{  
    int i= blockIdx.x*blockDim.x+threadIdx.x;  
    if (i<n) y[i]=a*x[i]+y[i];  
}
```



As we can see from this example, a CUDA program consists of one or more phases that are executed on either the host (CPU), or a device such as a GPU.

The phases that exhibit little or no data parallelism are implemented in host code.

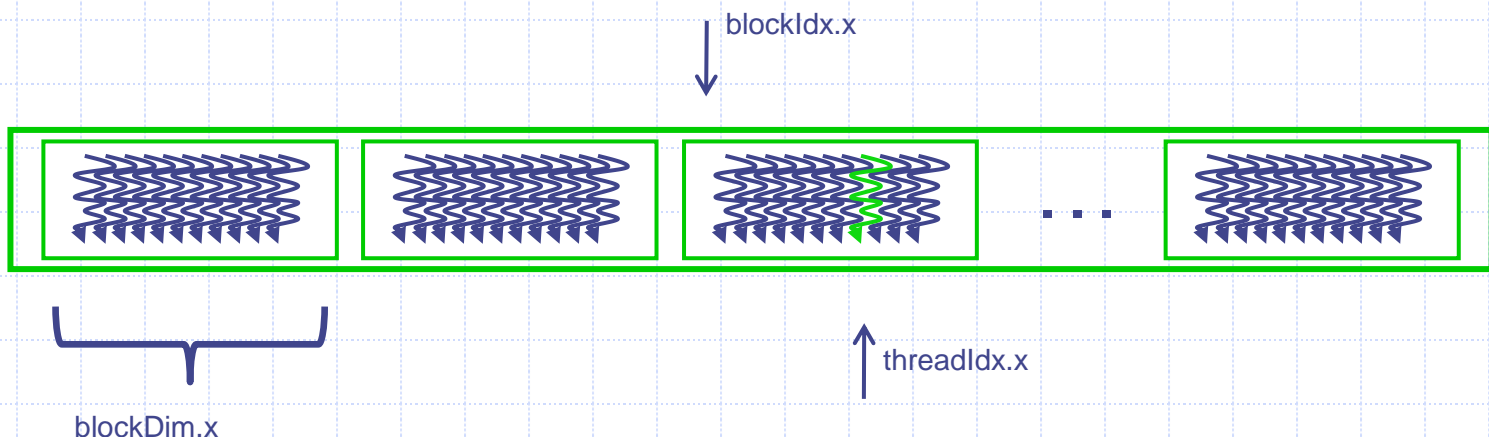
The phases that exhibit rich amount of data parallelism are implemented in the device code.



A device code is called a **kernel** in CUDA. When a kernel is invoked (or launched), a grid of threads is created. The kernel is executed by each individual thread. The threads are grouped into blocks for the execution in a streaming multiprocessor.

In this example, n threads, one per vector elements, are created. Each block contains 256 threads.

The index for each thread in this example contains two components: `blockIdx.x` and `threadIdx.x`. The first component specifies the block ID number of the thread, and the second component specifies the thread ID number.

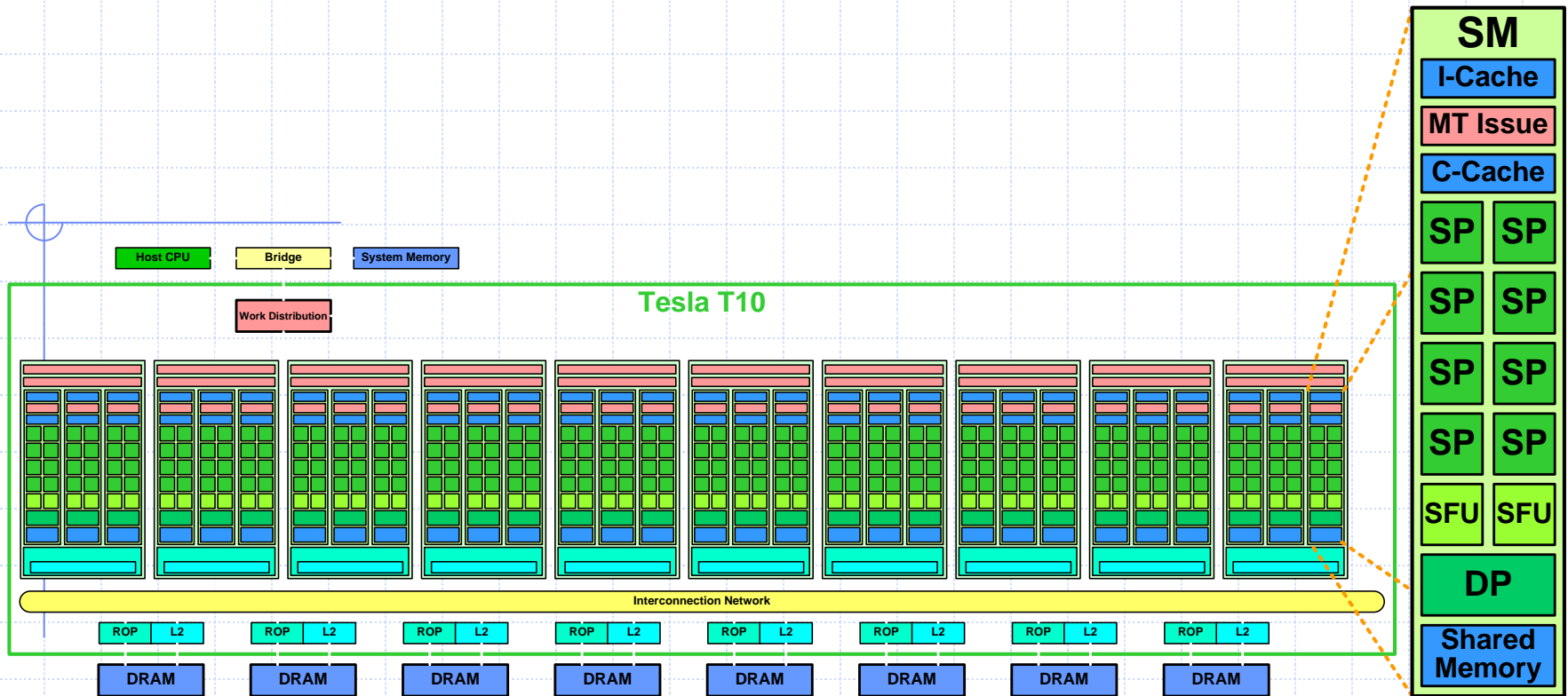


◆ Thread Assignment

Recall that threads are assigned to streaming multiprocessors on a **block-by-block** basis.

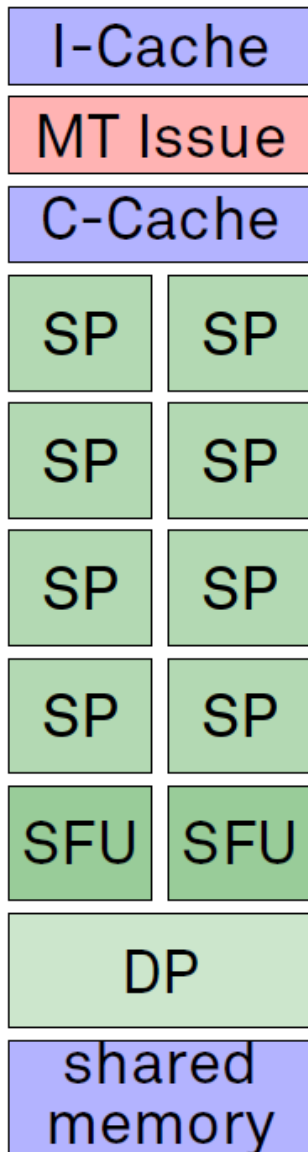
In a GPU, there are a number of streaming multiprocessors capable of operating concurrently.

For example, NVIDIA GT200 implementation has 30 streaming multiprocessors.



Architecture of GT200: there are 30 SMs.
 Each SM contains 8 **streaming processors** (SPs, or **cores**).
 Therefore, there are $30 \times 8 = 240$ cores in the GT200.

◆ Block Diagram of a Streaming Multiprocessor



I-Cache: Instruction Cache

MT Issue: Multi-Thread Issue (Warp Scheduler)

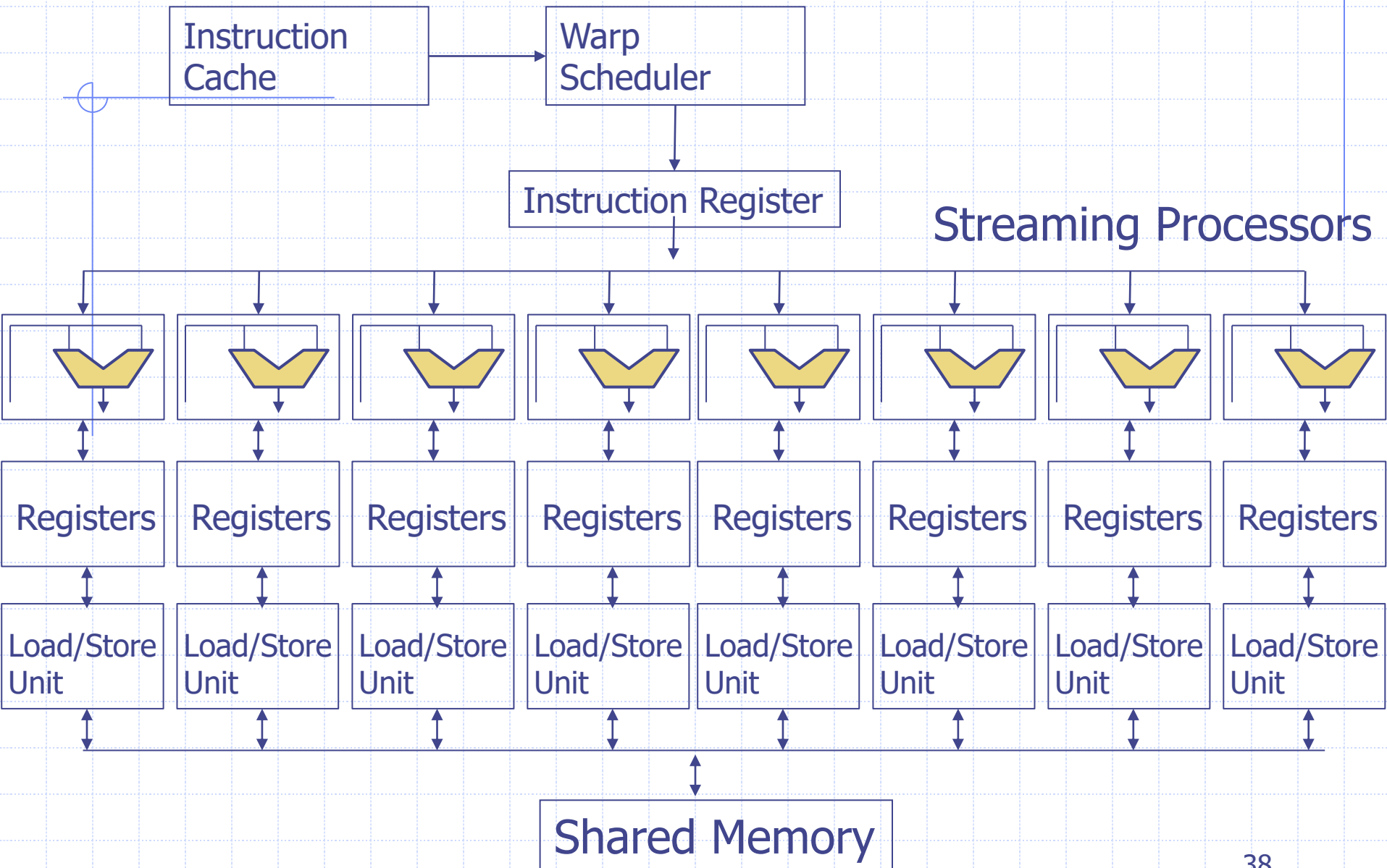
C-Cache: Constant Cache (Read-Only Data Cache)

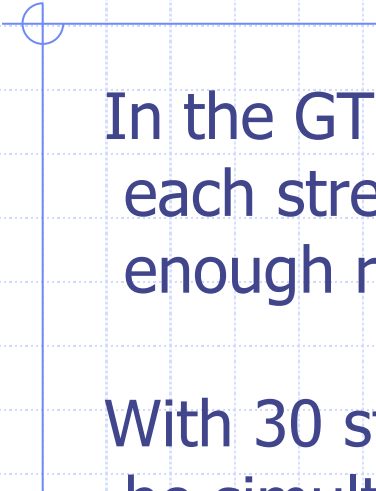
SP: Streaming Processor (Core)
Single Precision floating point,
Integer

SFU: Special Function Unit (sin, cos, log,...)

DP: Double Precision unit

◆ Architecture of a Streaming Multiprocessor





In the GT200, up to 8 blocks can be assigned to each streaming multiprocessor as long as there are enough resources to satisfy the needs of all the blocks.

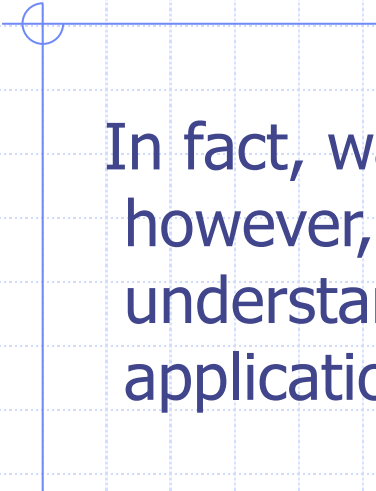
With 30 streaming multiprocessors, up to 240 blocks can be simultaneously assigned to them. Most grids contains many more than 240 blocks. The runtime system maintains a list of blocks that need to execute, and assigns new blocks to streaming multiprocessors as they complete the execution of blocks previously assigned to them.

◆ Thread Scheduling

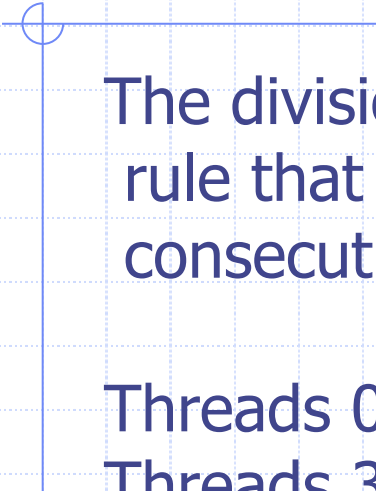
Thread scheduling is strictly an implementation concept, and must be discussed in the context of specific hardware implementation.

In the GT200 implementation, once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread unit called **warps**.

The warp is the unit of thread scheduling in streaming multiprocessors.



In fact, warps are not part of the CUDA specification; however, knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices.



The division of blocks into warps is based on the simple rule that each warp consists of 32 threads of consecutive threadIdx values.

Threads 0 through 31 form the first warp,
Threads 32 through 63 the second warp, and so on.



Each streaming multiprocessor contains 8 cores in GT200.

Each core can execute 1 thread.

Therefore, each streaming multiprocessor can execute 8 threads in parallel.

SIMT can discover which threads of a warp can execute the same instruction together.

When SIMT is used, the execution time is then 4 clock cycles for one instruction in each warp.

◆ Example

Suppose 4 blocks are assigned to a streaming multiprocessor. Each block has 256 threads. Therefore, there are $256/32=8$ warps per block.

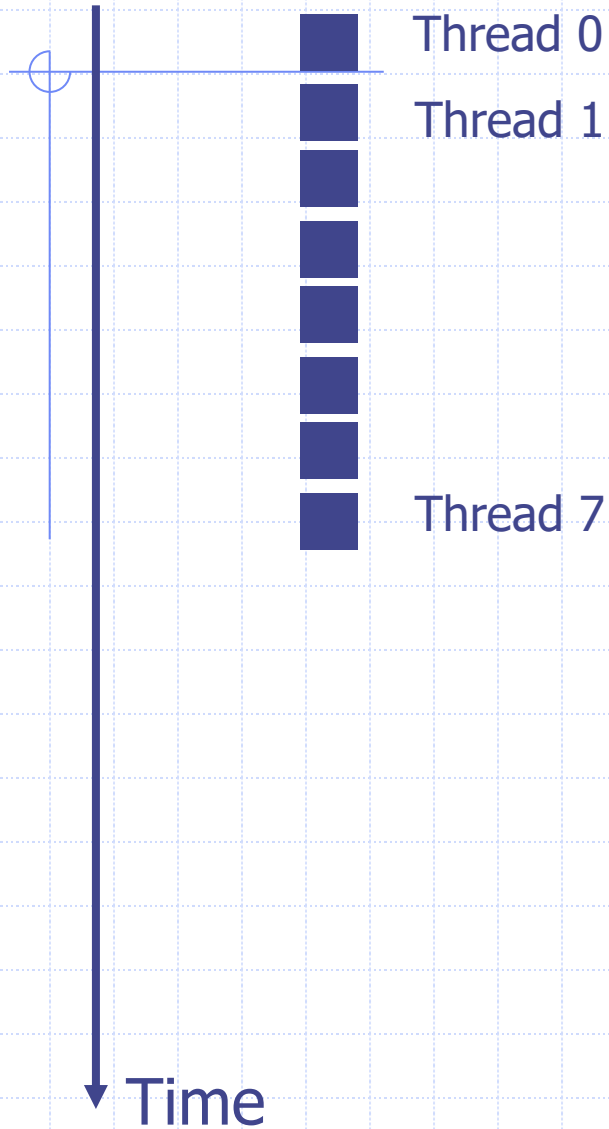
It then follows that there are $4 \times 8 = 32$ warps in the streaming multiprocessor.

We now compare a GT200 SM to a SUN UltraSPARC T2 core. Both architectures are multithreaded by scheduling threads over time.

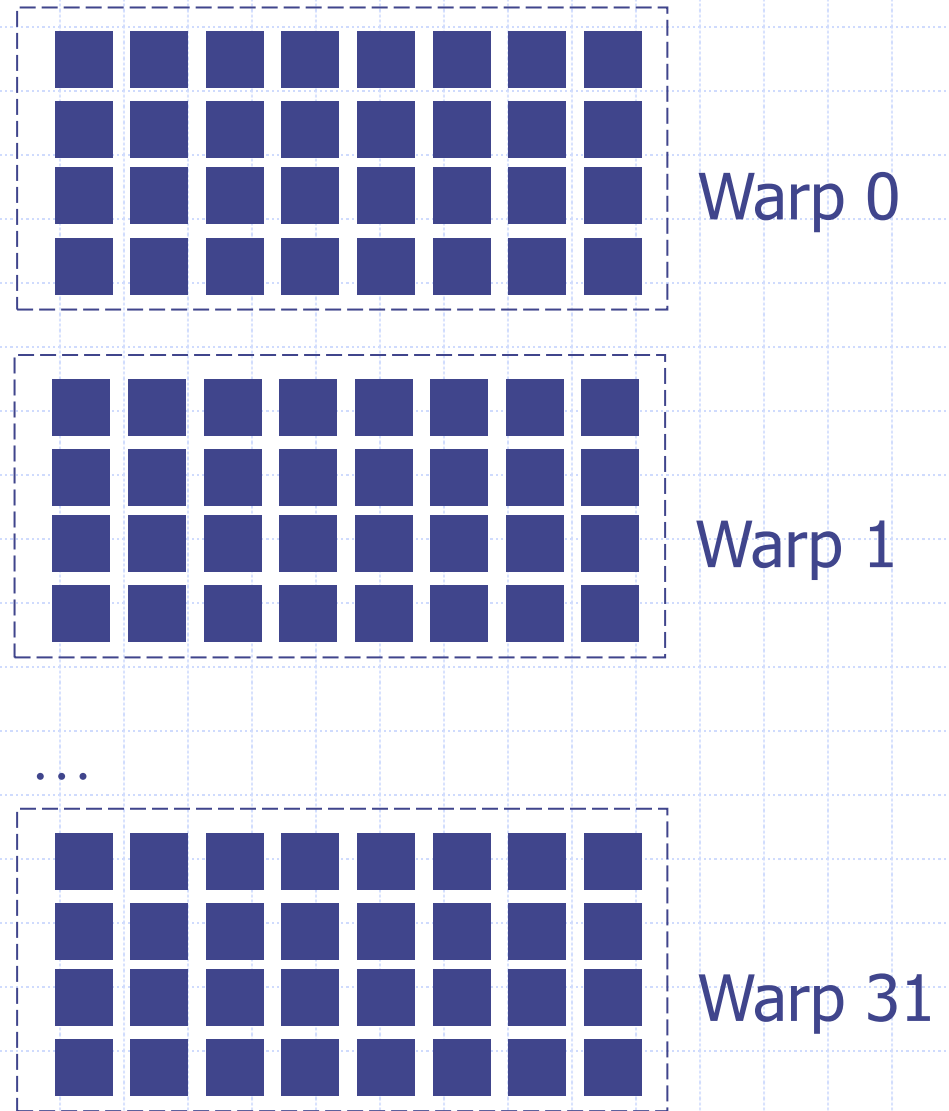
The GT200 SM can schedule up to 32 warps over time based on fine-grained hardware multithreading. The execution time is 4 clock cycles for one instruction in each warp.

Each UltraSPARC T2 core can schedule 8 threads over time. The T2 core contains only a single multithreaded processor. The execution time is 1 clock cycle for one instruction in each thread.

Ultra Sparc T2



GT200 Streaming Multiprocessor




◆ Latency Hiding

○ Why do we need to have so many warps in a streaming processor?

The answer is that this is how GPU efficiently execute the long-latency operations such as global memory accesses.

When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operations, the warp is not selected for execution.

Another resident warp that is no longer waiting for results is selected for execution.



Note that warp scheduling is also used for tolerating other types of long-latency operations such as floating point arithmetic and branch instructions.

With enough warps around, the hardware will likely to find a warp to execute at any point in time, thus making full use of execution hardware.