CE538
Parallel Computer Architecture
Spring 2015

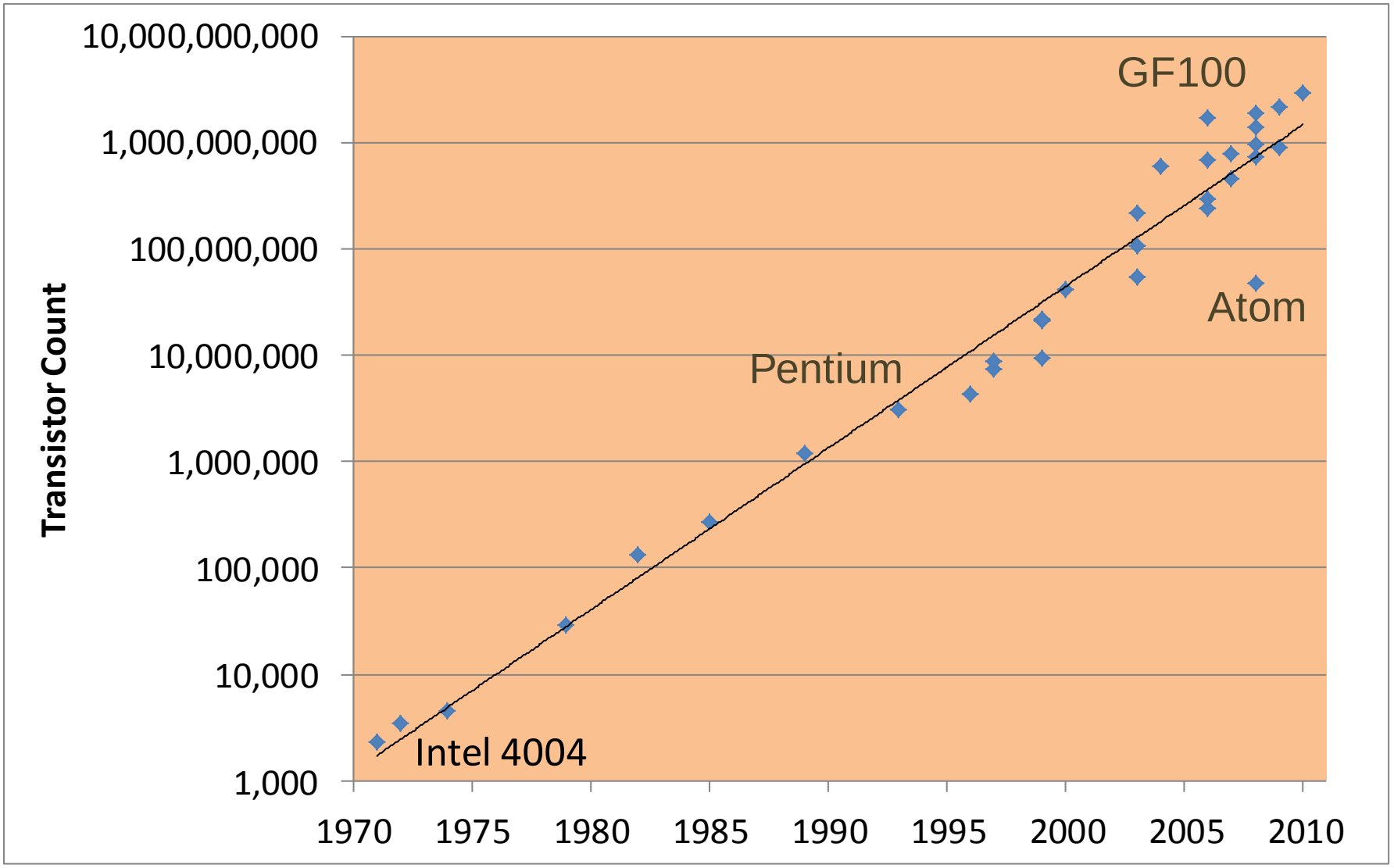
Graphics Processor Units (GPU) Architecture

Nikos Bellas

Computer and Communications Engineering Department
University of Thessaly

Some slides borrowed from ECE1742S at University of Toronto, CE432 at University of Thessaly

Moore's law continues



Serial Performance Scaling is Over

Cannot continue to scale processor frequencies

no 10 GHz chips

Cannot continue to increase power consumption

can’t melt chip

Can continue to increase transistor density

as per Moore’s Law

How to Use Transistors?

Instruction-level parallelism

out-of-order execution, speculation, …

vanishing opportunities in power-constrained world

Data-level parallelism

vector units, SIMD execution, …

increasing … SSE, AVX, Cell SPE, Clearspeed, GPU

Thread-level parallelism

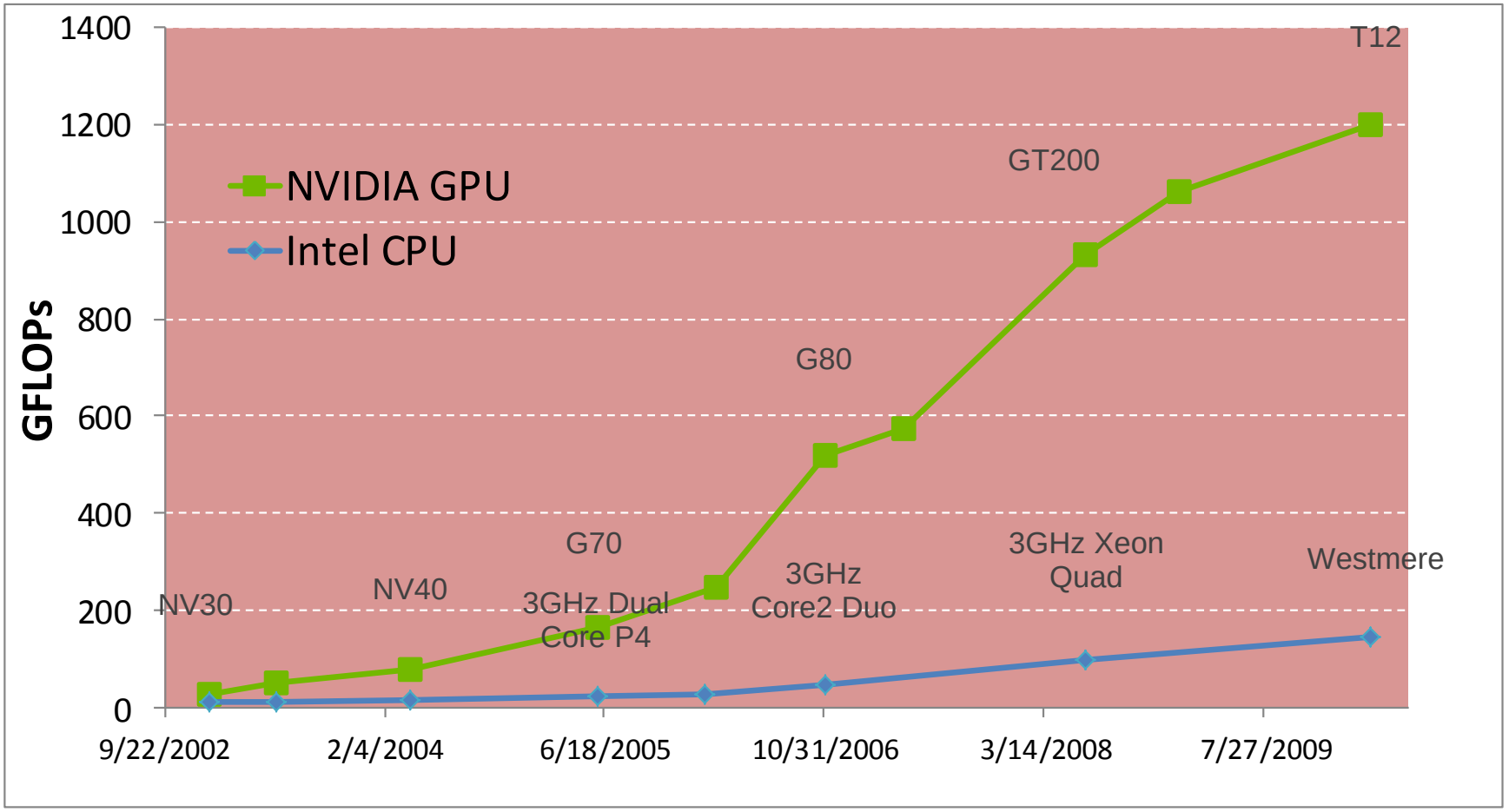
increasing … multithreading, multicore, manycore

Intel CMP (Ivy Bridge), Sun Niagara, Cell, NVIDIA Kepler, Titan, …

Why Massively Parallel Processing?

A quiet revolution and potential build-up

Computation: TFLOPs vs. 100 GFLOPs

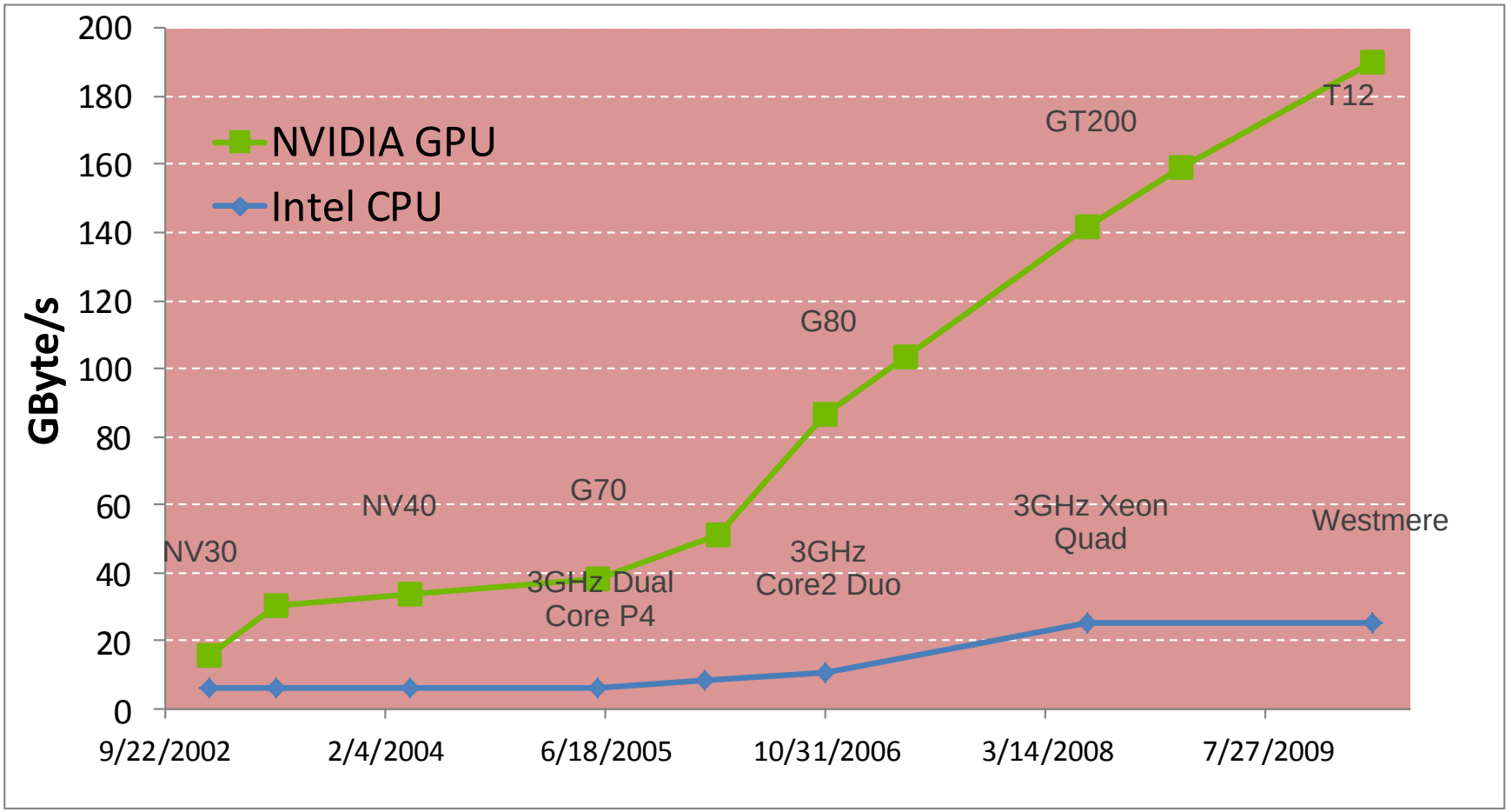


GPU in every PC – massive volume & potential impact

Why Massively Parallel Processing?

A quiet revolution and potential build-up

Bandwidth: ~10x



GPU in every PC – massive volume & potential impact

The “New” Moore’s Law

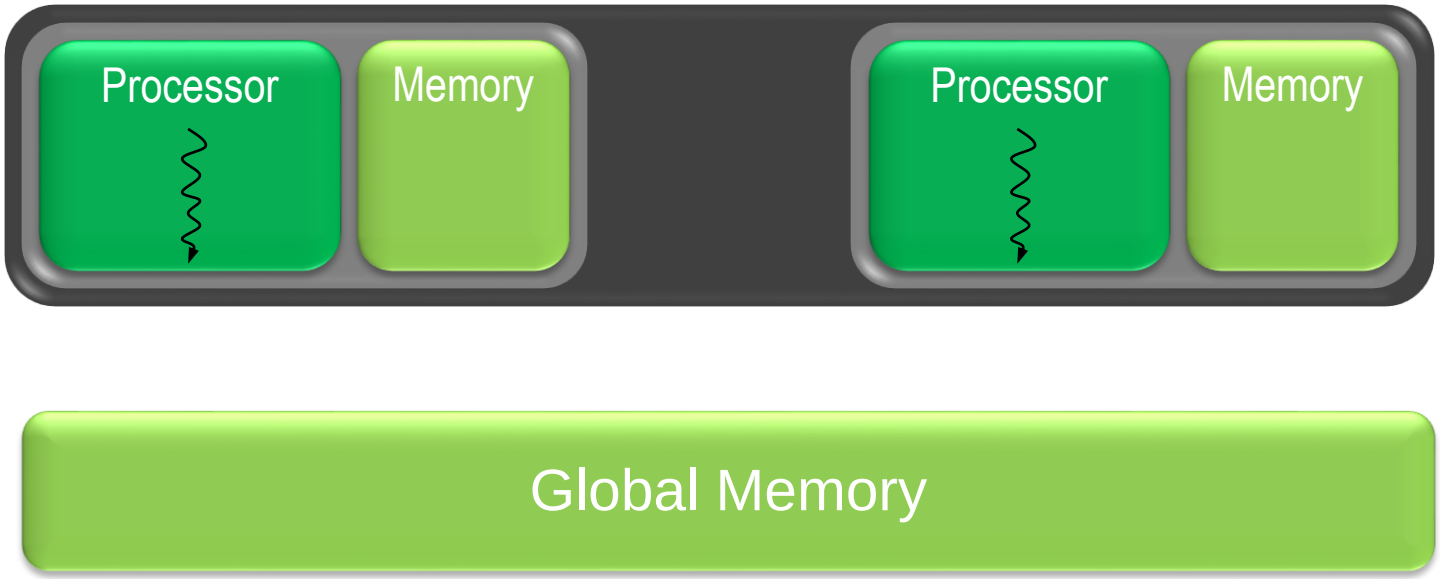
Computers no longer get faster, just wider

You *must* re-think your algorithms to be parallel !

Data-parallel computing is most scalable solution

- Otherwise: refactor code for 2 ~~cores~~ 4 ~~cores~~ 8 ~~cores~~ 16 cores…
- You will always have more data than cores – build the computation around the data

Generic Multicore Chip

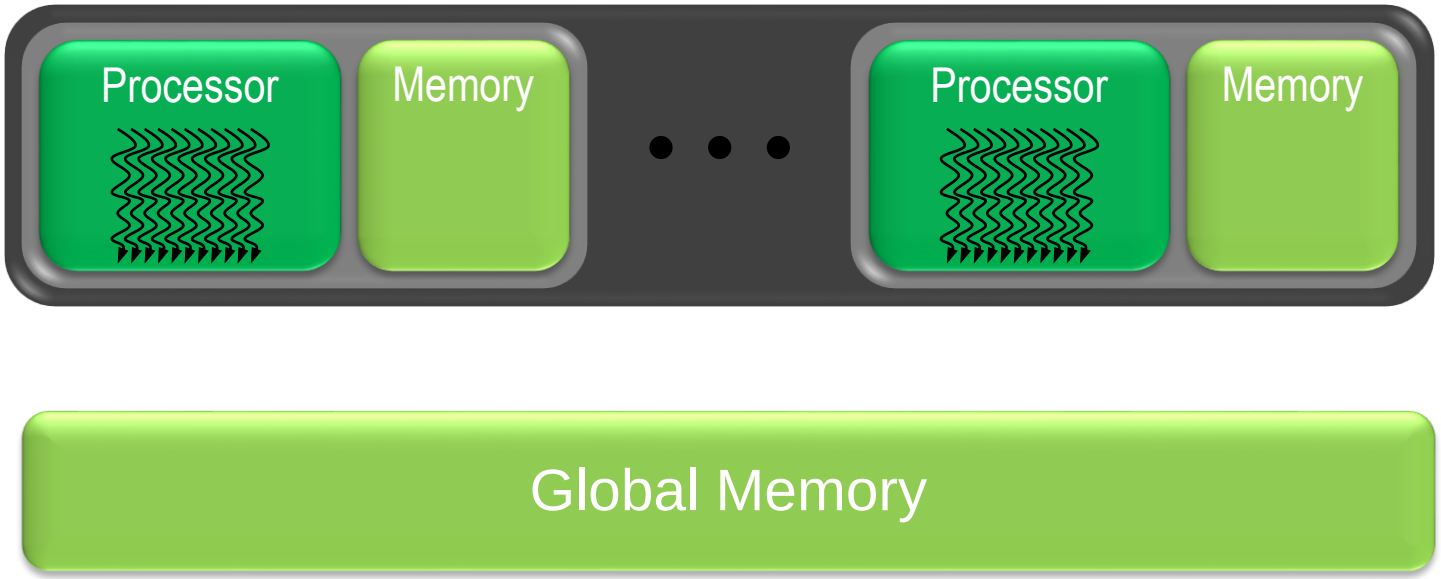


Handful of processors each supporting ~1 hardware thread

On-chip memory near processors (cache, RAM, or both)

Shared global memory space (external DRAM)

Generic Manycore Chip



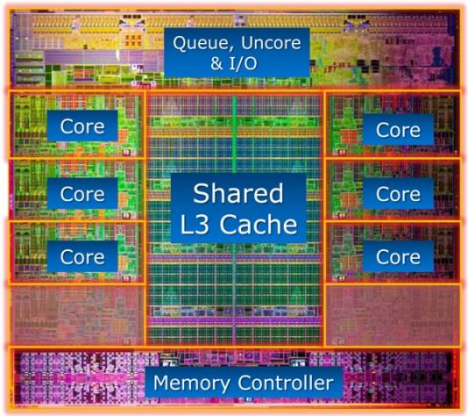
Many processors each supporting many hardware threads

On-chip memory near processors (cache, RAM, or both)

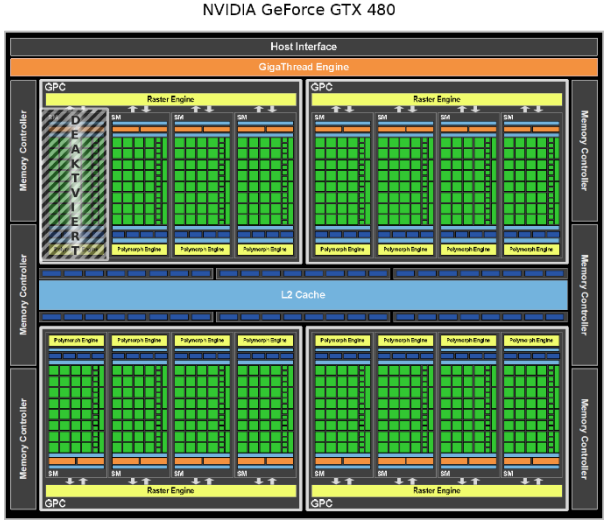
Shared global memory space (external DRAM)

Multicore & Manycore, *cont.*

Specifications	Intel Sandy Bridge (32nm)	GeForce GTX480
Processing Elements	6 cores, 4 way SIMD @3.2 GHz	15 SMs, 32 SPs per SM
Resident Strands/Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	15*32 = 480 threads
SP GFLOP/s	153.6	1345
Memory Bandwidth	32 GB/s	177.4 GB/s
Power	130 W	250 W



Intel Sandy Bridge



nVIDIA GTX 480

Enter the GPU

Massive economies of scale

Massively parallel mainstream computing

Why is this different from a CPU?

Different goals produce different designs

- CPU must be good at everything, parallel or not
- GPU assumes work load is highly parallel

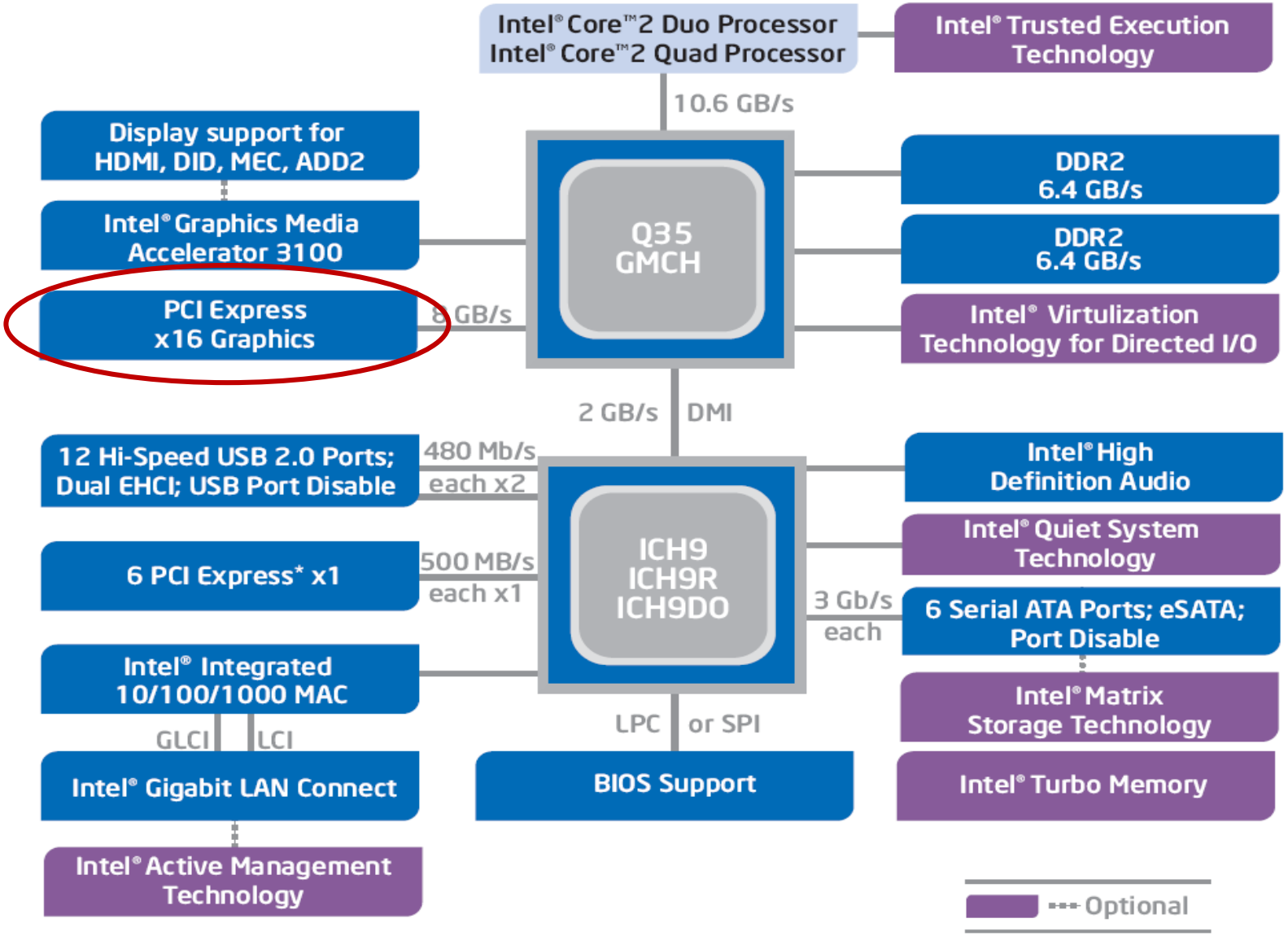
CPU: minimize latency experienced by 1 thread

- big on-chip caches
- sophisticated control logic

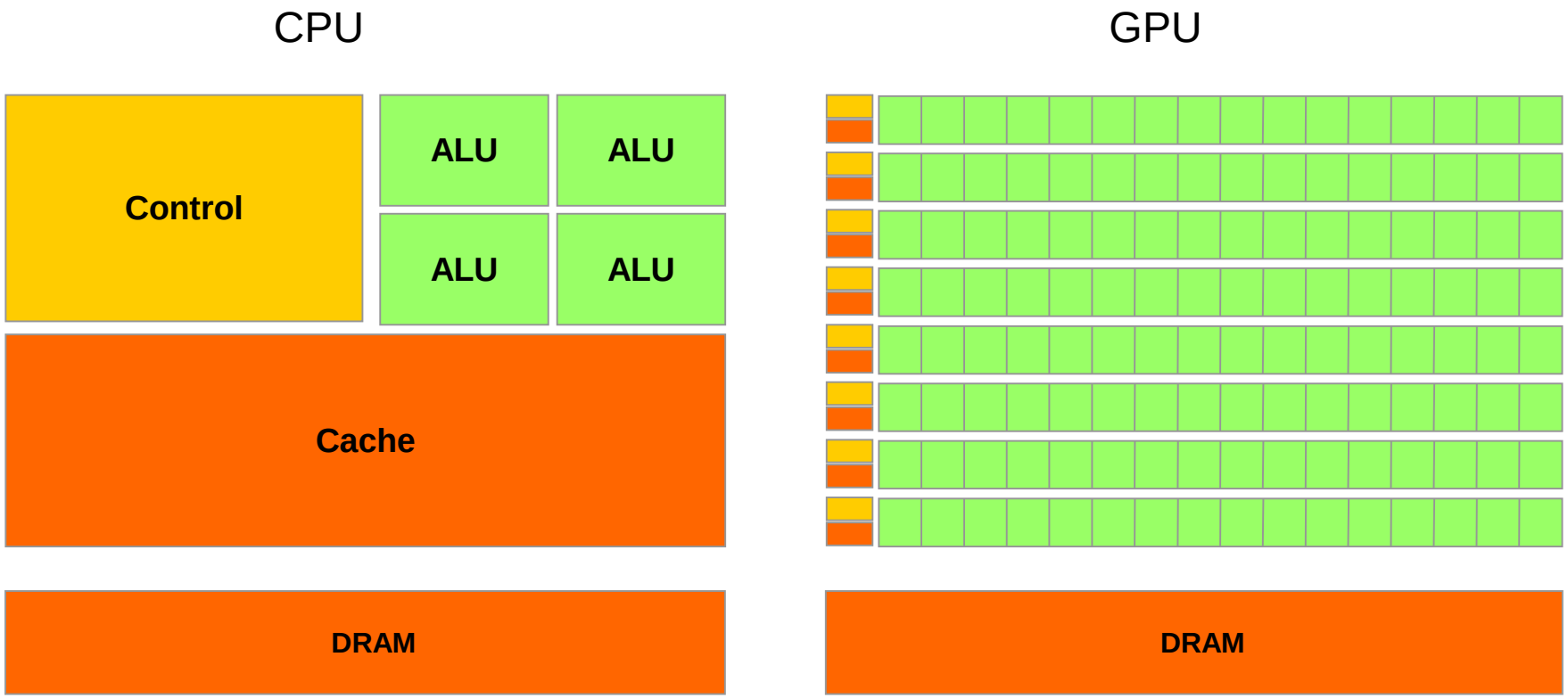
GPU: maximize throughput of all threads (amortization)

- # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
- multithreading can hide latency => skip the big caches
- However, Fermi architecture include caches
- share control logic across many threads

System Architecture for a typical Intel based system



CPUs and GPUs: Different Design Philosophies

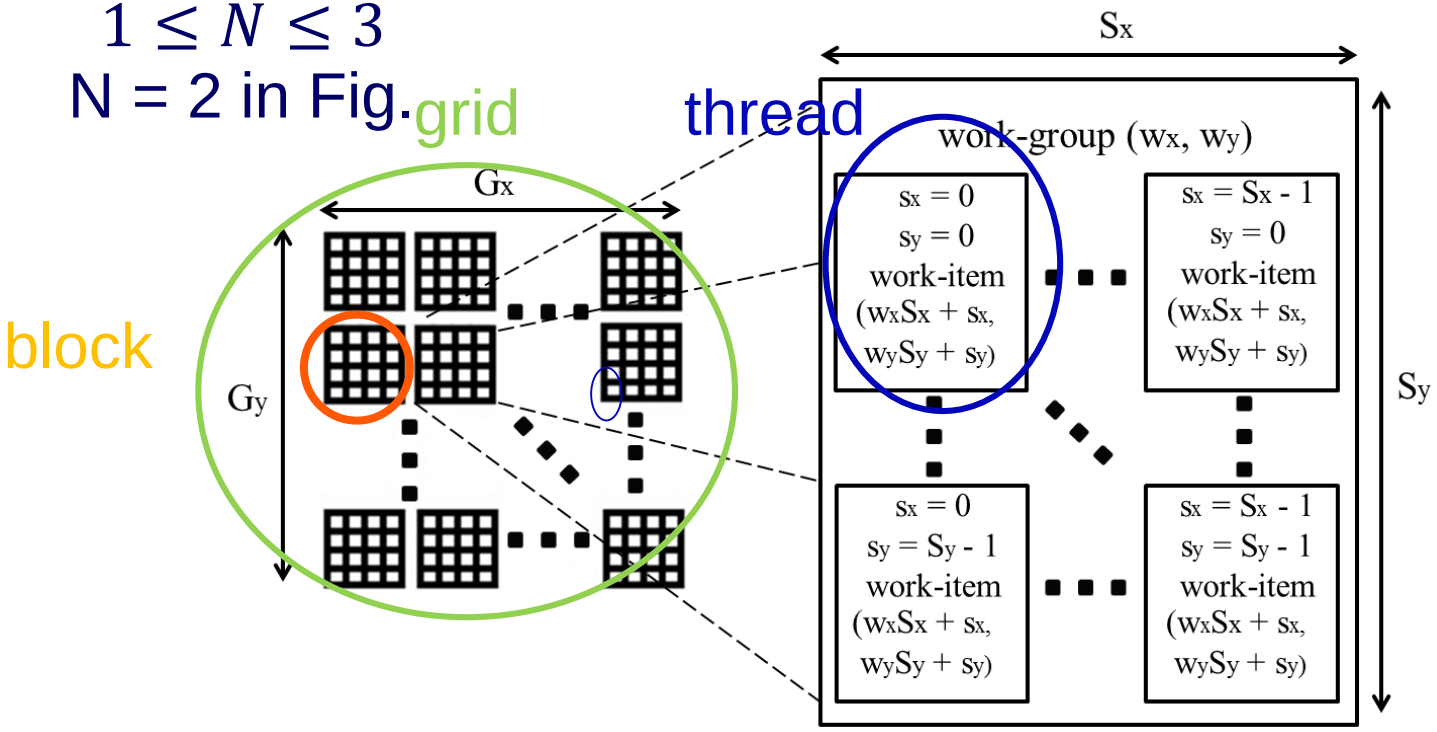


CUDA Main Ideas

- CUDA defines a geometric partitioning of grid of computations
- Grid consists of N dimensional space of blocks
- Each block consists of N dimensional space of threads

$1 \leq N \leq 3$

N = 2 in Fig.



CUDA Simple Example

CUDA kernel describes the computation of a work-item

- Finest parallelism granularity

e.g add two integer vectors (N=1)

```
void add(int* a,
         int* b,
         int* c) {
for (int idx=0; idx<sizeof(a); idx++)
   c[idx] = a[idx] + b[idx];
}
```

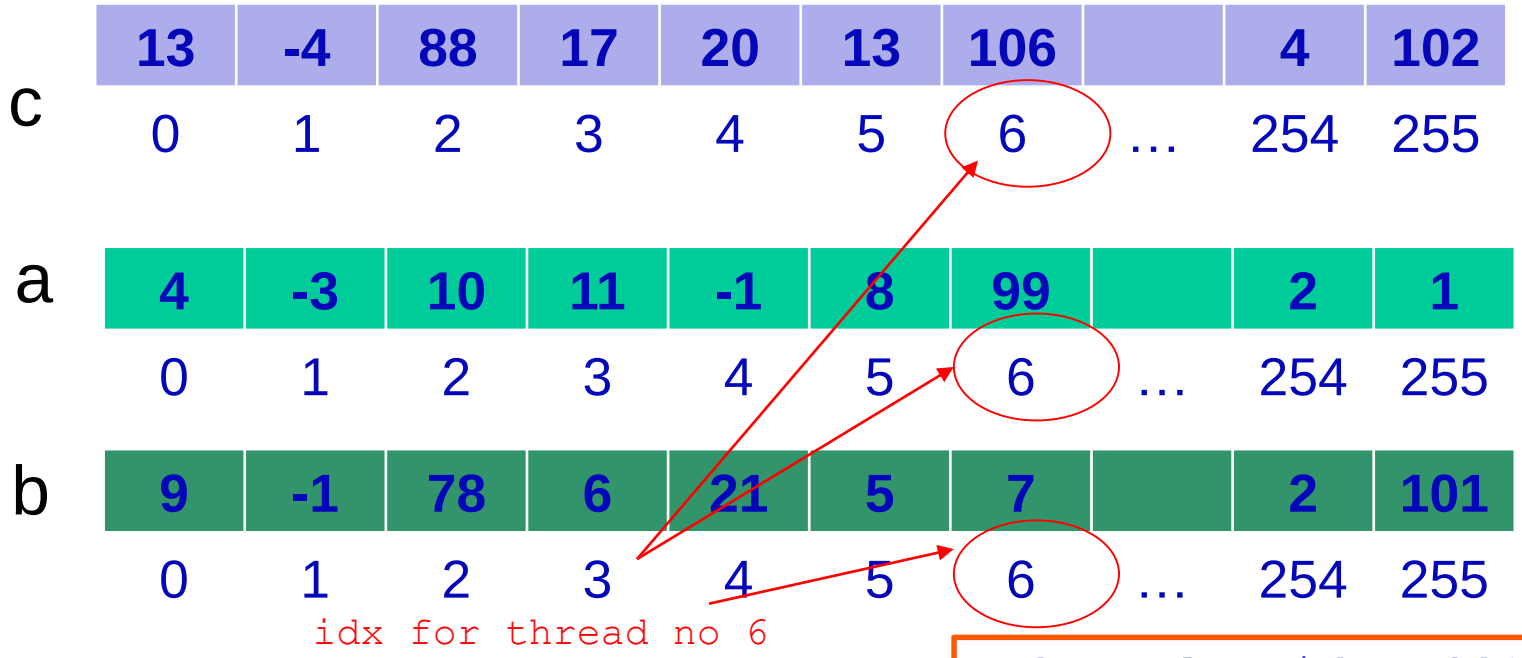
C code

Run-time calls
Used to differentiate execution for each thread

```
__kernel void vadd(
            __global int* a,
            __global int* b,
            __global int* c) {
  idx = threadIDx.x +
        blockDim.x * blockIdx.x;
  c[idx] = a[idx] + b[idx];
}
```

CUDA kernel code

CUDA Simple Example



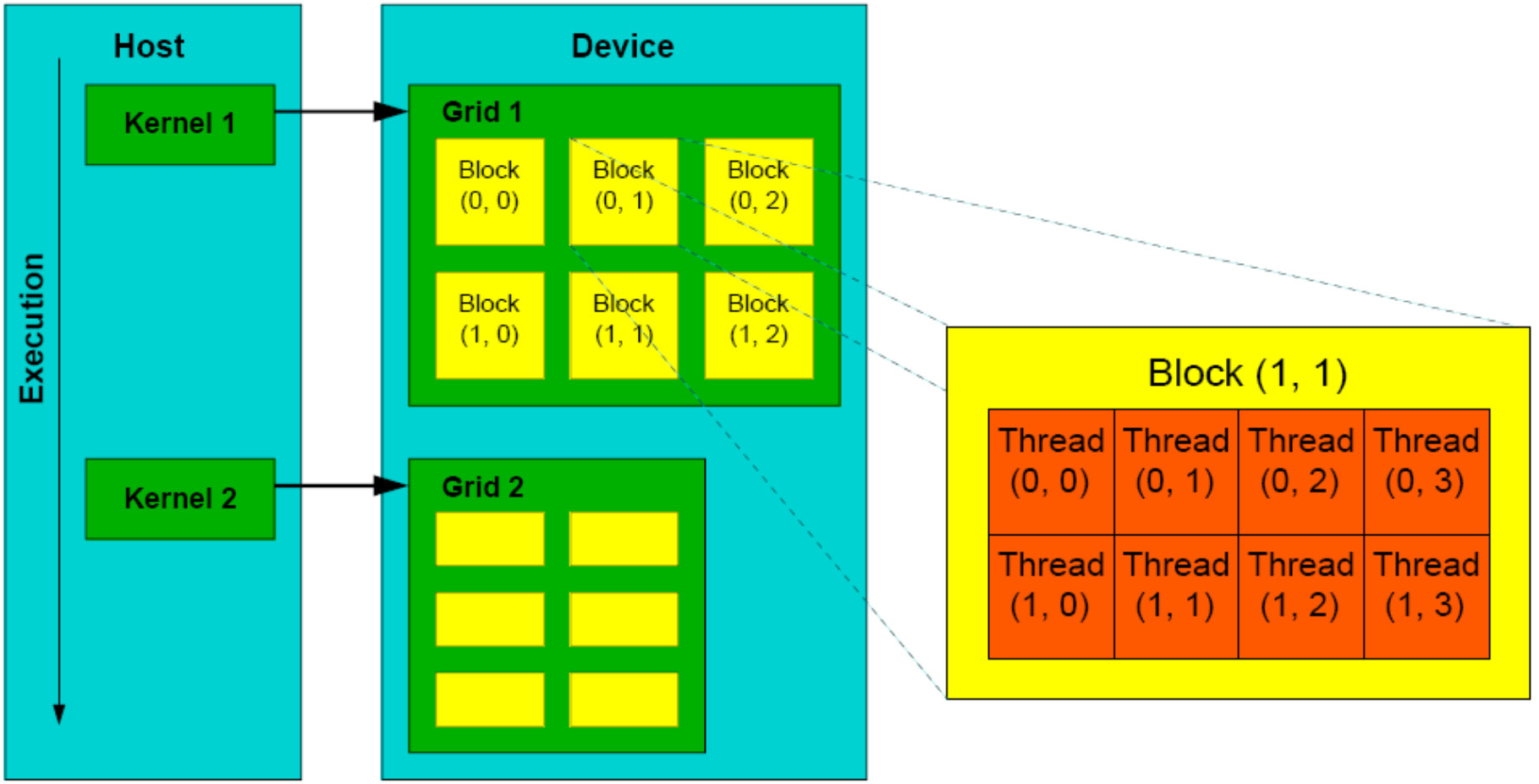

```
int main() {
    // Run grid of N=1 blocks
    // of N=256 thread each
 vadd <<< 1, 256>>>(a,b,c)
}
```

Host code

```
__kernel void vadd(
            __global int* a,
            __global int* b,
            __global int* c) {
  idx = threadIDx.x +
        blockDim.x * blockIdx.x;
  c[idx] = a[idx] + b[idx];
}
```

CUDA kernel code

CUDA Refresher

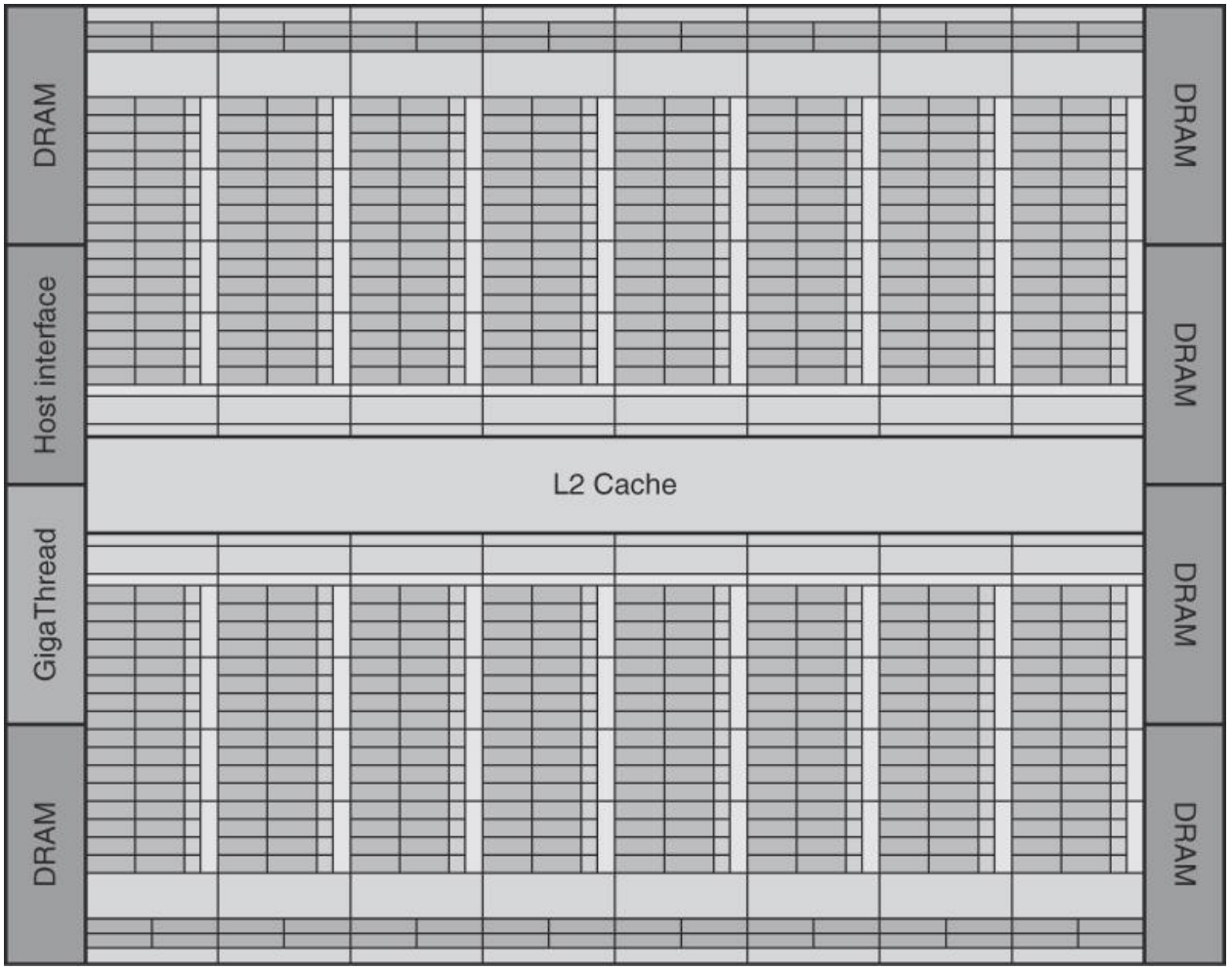


Why? Realities of integrated circuits: need to cluster computation and storage to achieve high speeds

GPU Architecture Overview

GeForce GTX 480 Diagram

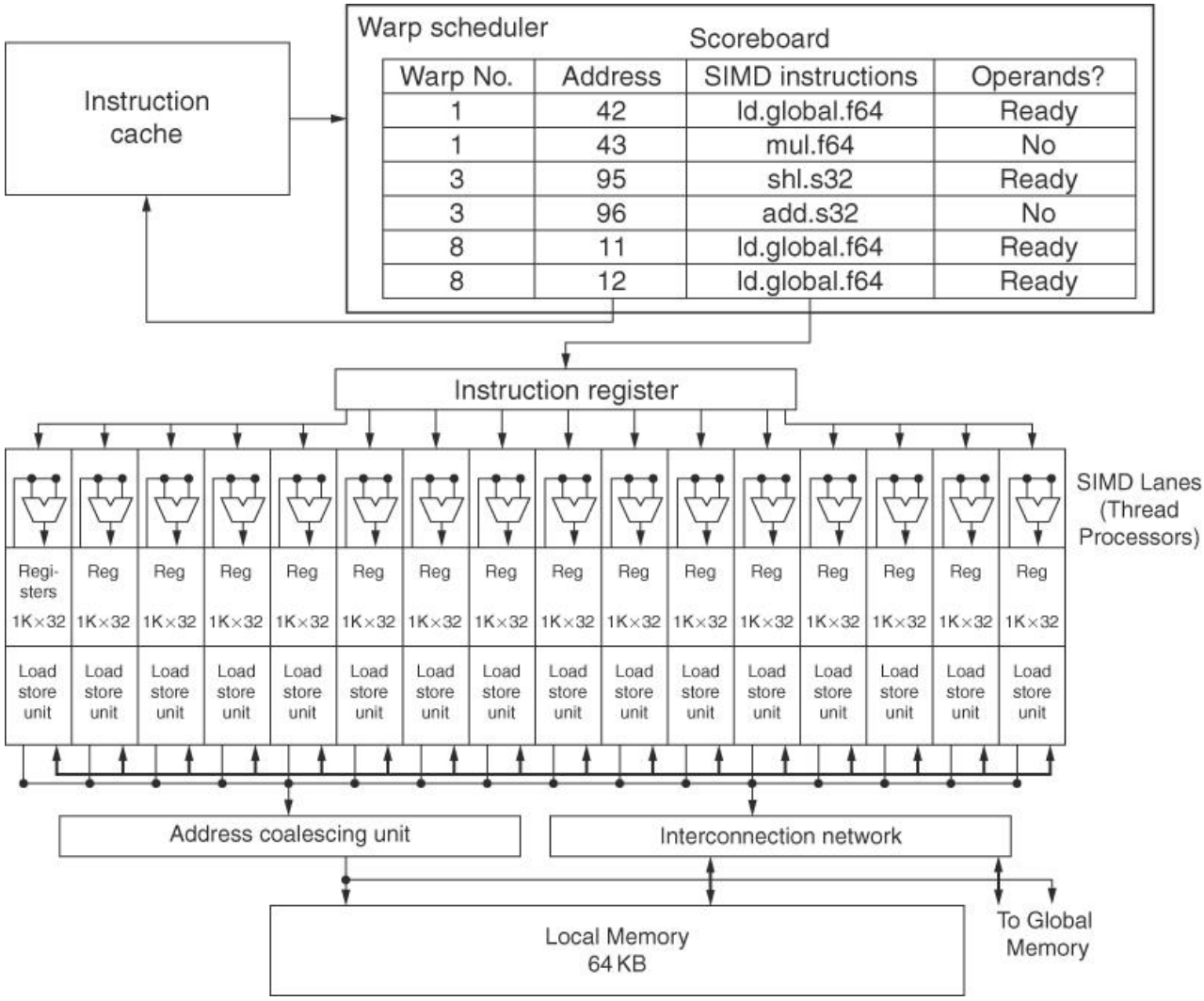
- 16 Streaming Multiprocessors (SM)
- 32 Streaming Processors (SPs) per SM
- No scalar processor
- Grid is launched on the Streaming Processor Array (SPA)
- A thread block is assigned to a SM
- **Thread Block Scheduler** schedules Blocks to SMs
  - Thread Blocks are distributed to all the SMs
  - Potentially >1 Blocks in each SM
- A CUDA thread is assigned to a SP



GPU Architecture Overview

Streaming Multiprocessor (SM) Diagram (1)

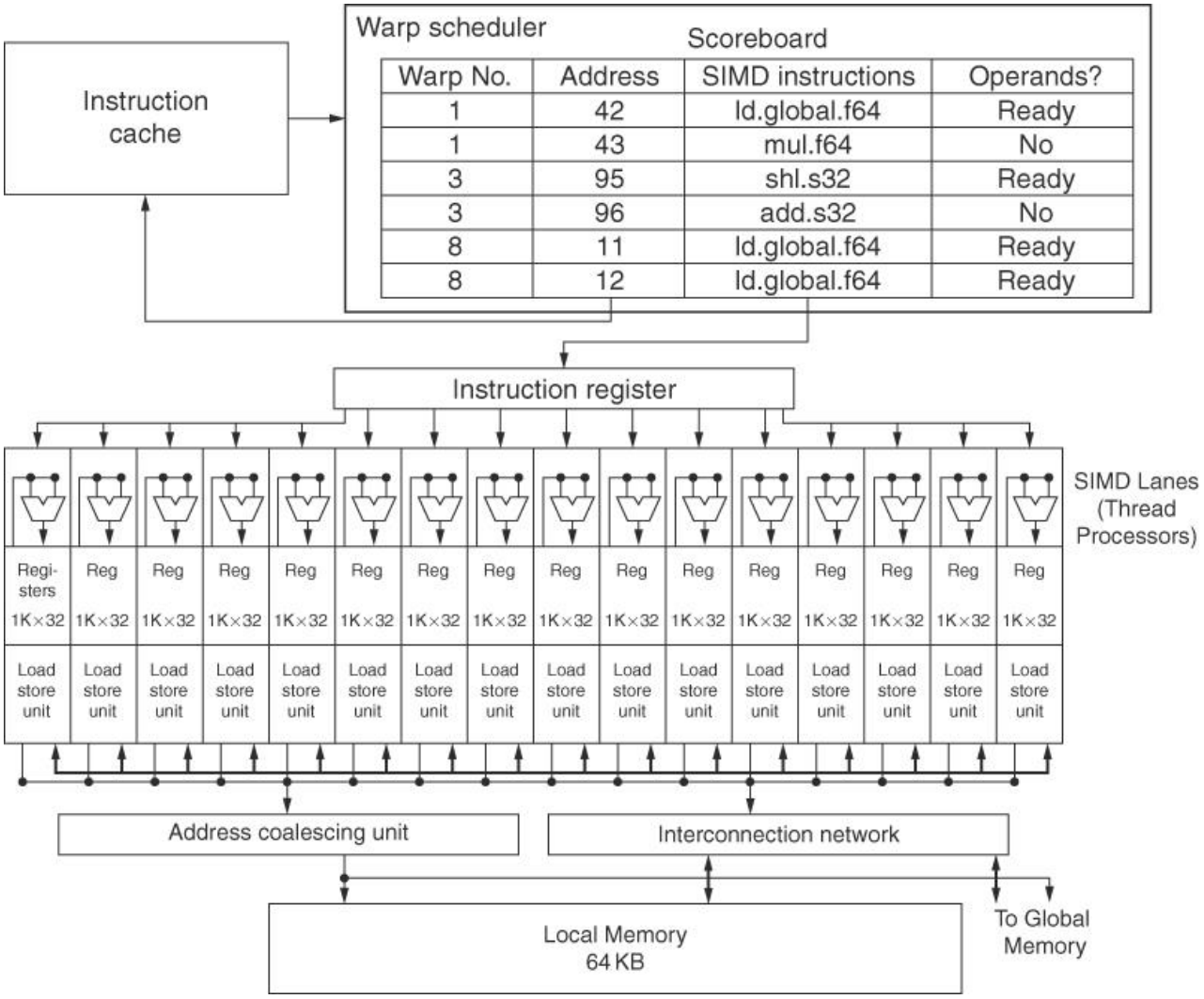
- In this diagram each SM has 16 SPs (not 32)
  - Each SP has a simple data path with 1K 32-bit registers
  - Also ports to memory
- Each SM launches **Warps** of Threads
  - For NVIDIA 1 warp has 32 threads
  - Implementation decision, not part of the CUDA programming model
- **All threads in the Warp execute the same instruction**
  - With probably different operands
  - Single Instruction Multiple Threads (SIMT)



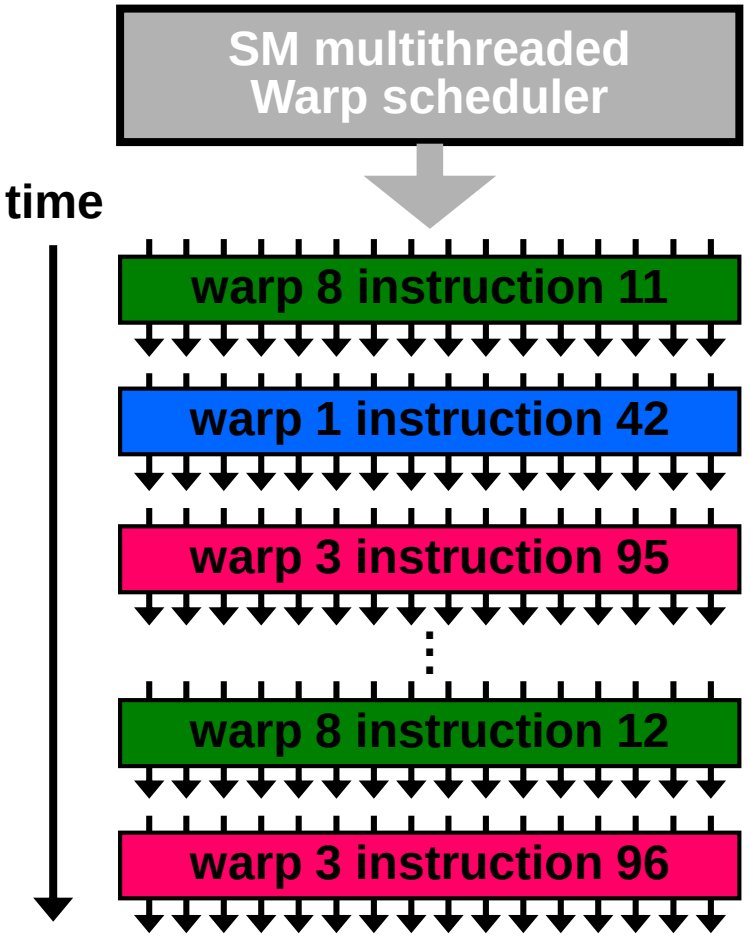
GPU Architecture Overview

Streaming Multiprocessor (SM) Diagram (2)

- SM schedules and executes Warps that are ready to run
  - Using the **Warp scheduler**
- Threads in a block are independent (by definition)
- Therefore, no need to check dependencies between warps
- All 32 instructions of the warp are executed in lockstep mode
  - One PC per warp
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks



Warp Scheduling



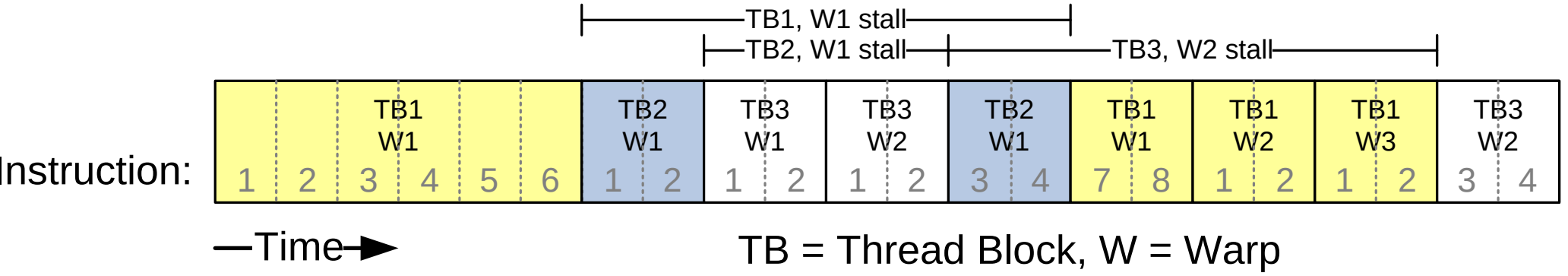
- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G200

How many warps are there?

If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?

- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps
- At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.

Warp Scheduling: Hiding Thread stalls



Warp Scheduling Ramifications

If one global memory access is needed for every 4 instructions

A minimal of 13 Warps are needed to fully tolerate a 200-cycle memory latency

Why?

- Need to hide 200 cycles every four instructions
- Every Warp occupies 4 cycles during which the same instruction executes
- Every 4 instructions a thread stalls
- Every 16 cycles a thread stalls
- 200/16 =12.5 or at least 13 warps

GPU ISA

- Parallel Thread Execution (PTX)
  - Virtual ISA as abstraction of the hardware instruction set
  - Uses virtual registers
  - Translation to machine code is performed in software
    - Compare to x86 uops
  - Example *vadd* kernel:

GPU ISA

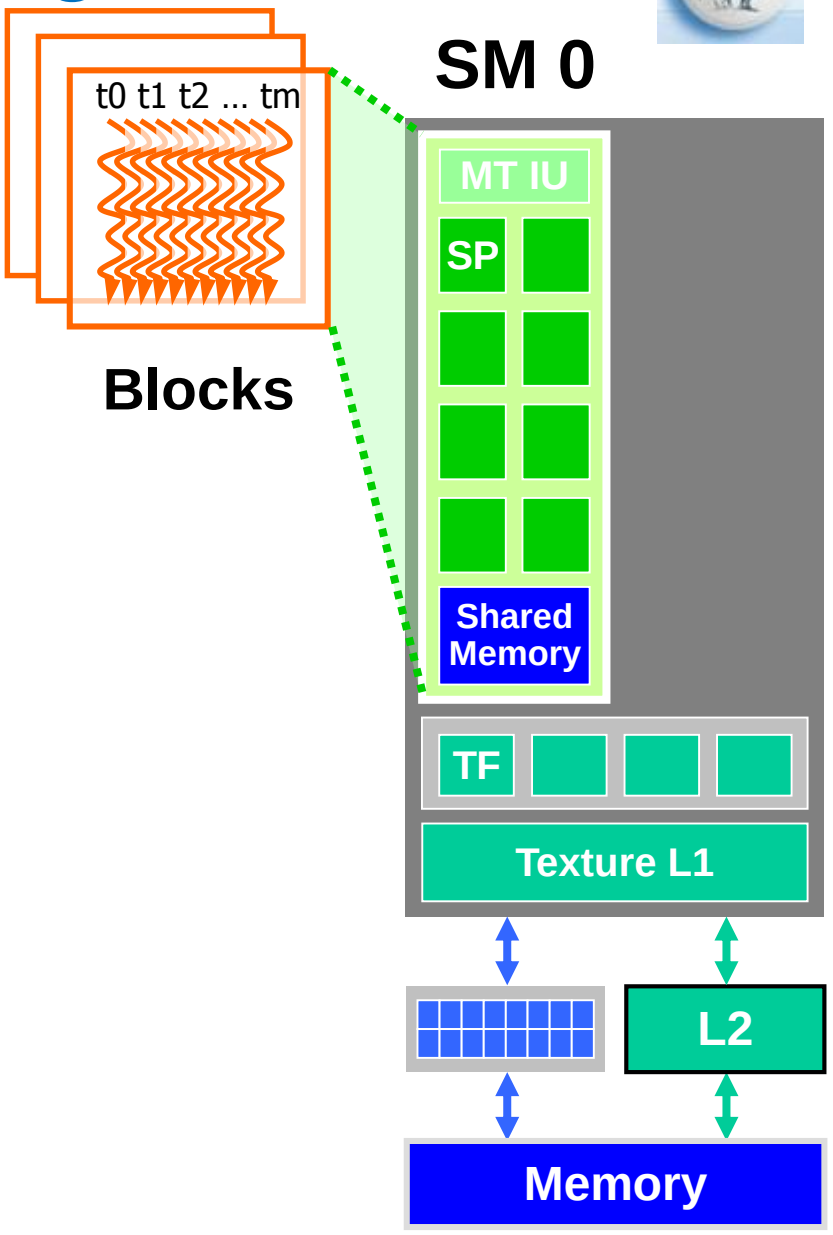
Example *vadd* kernel:

```
__kernel void vadd(
            __global int* a,
            __global int* b,
            __global int* c) {
  idx = threadIDx.x +
        blockDim.x * blockIdx.x;
  c[idx] = a[idx] + b[idx];
}
```

```
shl.s32       R8, blockIdx, 8      ; blockIdx.x * blockDim.x (=256)
add.s32       R8, R8, threadIdx    ; R8 = idx = my CUDA thread ID
ld.global.f64 RD0, [a+R8]          ; RD0 = a[idx]
ld.global.f64 RD2, [b+R8]          ; RD2 = b[idx]
add.f64       RD0, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (c[idx])
st.global.f64 [c+R8], RD0          ; c[idx] = a[idx]+b[idx]
```

Conditional Branching in GPUs

- As we mentioned, each thread in a Warp executes the same instruction in every clock cycle
- What if some of the 32 threads diverge in an if-then-else statement?
- GPU approach:
  - Use predication to either execute an instruction
  - Or Nullify it (execute NOP)
- Per-thread 1-bit predicate register, specified by programmer
- Predicate register is the bit-mask to decide if instruction executes or is NOP
- Conditional branching may be source of inefficiencies



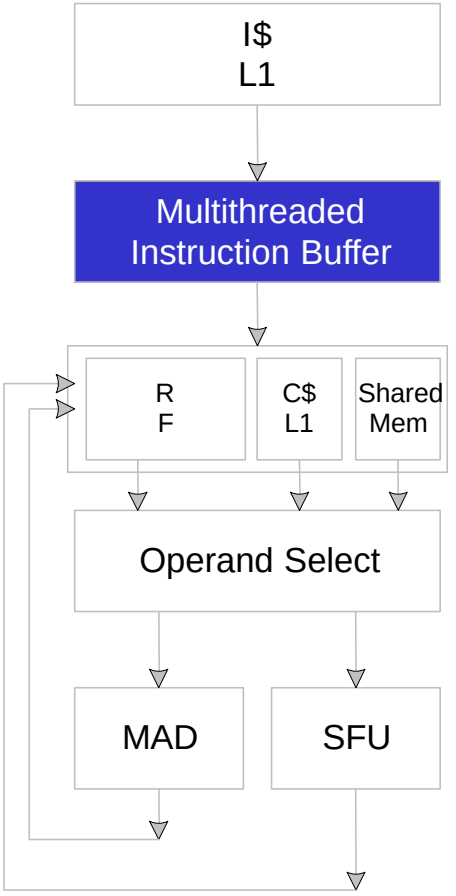
Conditional Branching Example

CUDA Kernel	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, execute 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, execute 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*

SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one “ready-to-go” warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboarding used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp



Scoreboarding

All register operands of all instructions in the Instruction Buffer are scoreboarded

- Status becomes ready after the needed values are deposited
- remember Tomasulo's algorithm
- cleared instructions are eligible for issue

Decoupled Memory/Processor pipelines

- any thread can continue to issue instructions until scoreboarding prevents issue

Granularity Considerations

For a 2D grid , should I use 8X8, 16X16 or 32X32 blocks?

Constraints:

1 SM can take at most 1024 threads

1 SM can take at most 8 blocks

1 block can have at most 512 threads

- For 8X8, we have 64 threads per Block. Since each SM can take up to 1024 threads, it can take up to 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM

- For 16X16, we have 256 threads per Block. Since each SM can take up to 1024 threads, it can take up to 4 Blocks and achieve full capacity unless other resource considerations overrule.

- For 32X32, we have 1024 threads per Block. Not even one can fit into an SM.

Scalar Units

- 32 bit ALU and Multiply-Add
- IEEE Single-Precision Floating-Point
- Integer
- Latency is 4 cycles
- FP: NaN, Denormals become signed 0.
- Round to nearest even

Special Function Units

Transcendental function evaluation and per-pixel attribute interpolation

Function evaluator:

- rcp, rsqrt, log2, exp2, sin, cos approximations
- Uses quadratic interpolation based on Enhanced Minimax Approximation
- 1 scalar result per cycle

Latency is 16 cycles

Memory System Goals

GOAL: High-Bandwidth

As much parallelism as possible

wide. 512 pins in G200 / Many DRAM chips

fast signaling. max data rate per pin.

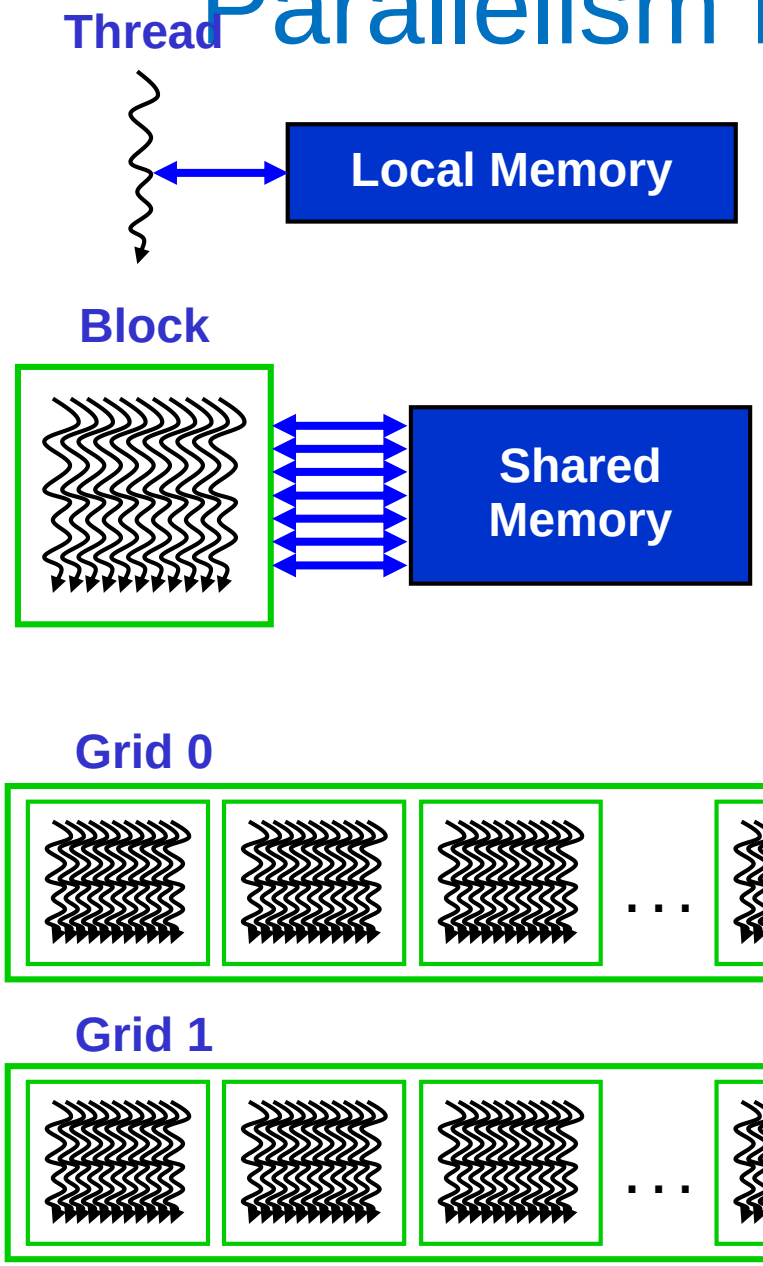
maximize utilization

- Multiple bins of memory requests
- Coalesce requests to get as wide as possible
- Goal to use every cycle to transfer from/to memory

Compression: lossless and lossy

Caches where it makes sense. Small

Parallelism in the Memory System



- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-grid
  - Shared by all threads
  - Inter-Grid communication

SM Memory Architecture

Threads in a Block share data & results

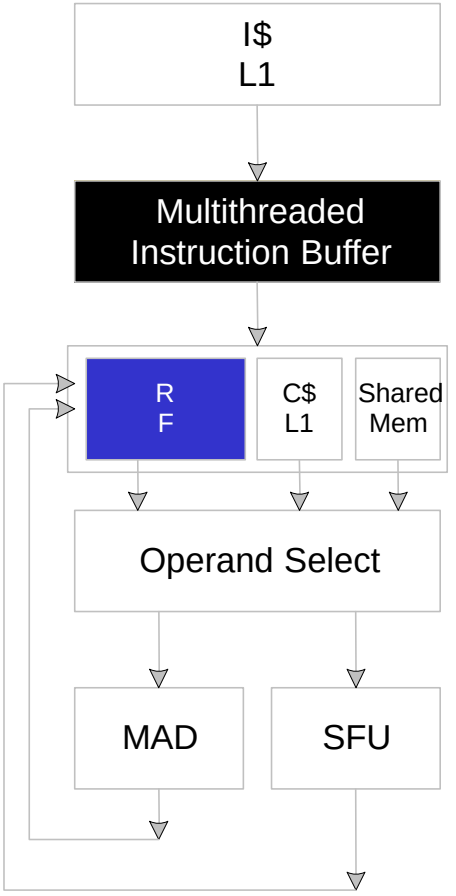
- In Shared Memory and Global Memory
- Synchronize at barrier instruction

Per-Block Shared Memory Allocation

- Keeps data close to processor
- Minimize trips to global Memory
- SM Shared Memory dynamically allocated to Blocks, one of the limiting resources

SM Register File

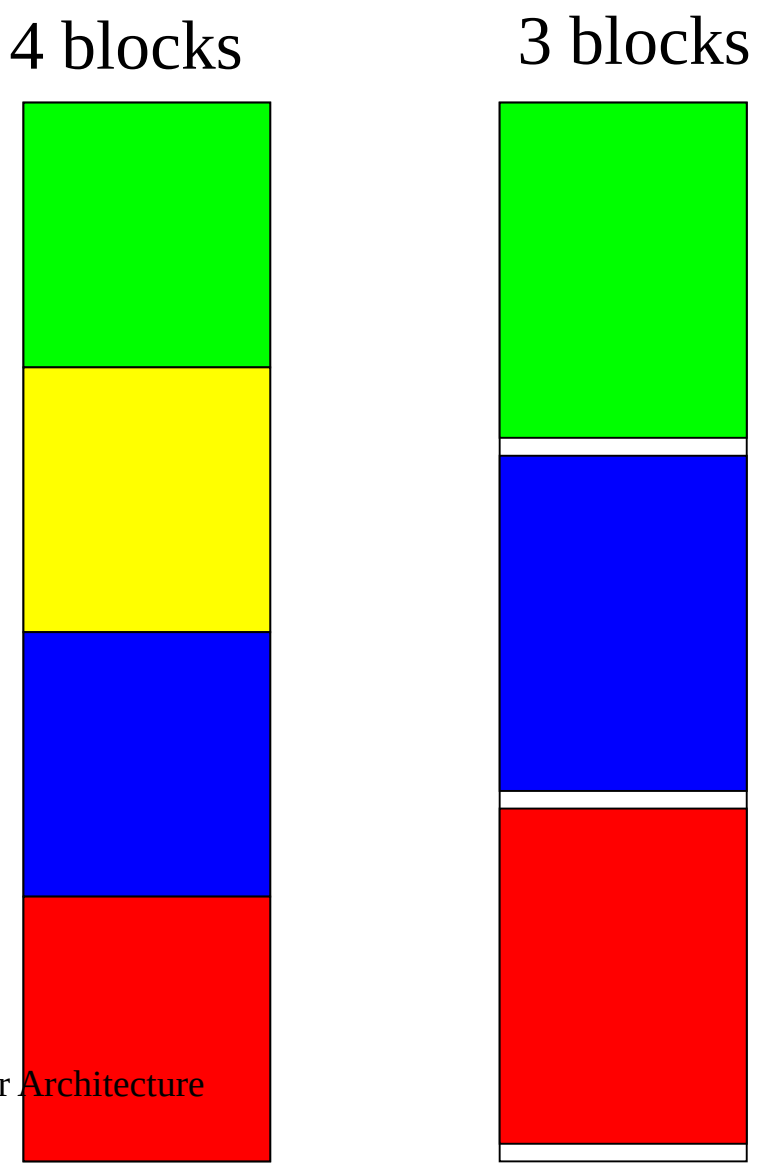
- Register File (RF)
- Implements Local Memory
  - 64 KB
  - 16K 32-bit registers
  - Provides 4 operands/clock
- Load/Store pipe can also read/write RF



Programmer's View of Register File

There are 16K registers in each SM in G200

- This is an implementation decision, not part of CUDA
- Registers are dynamically partitioned across all Blocks assigned to the SM
- Once assigned to a Block, the register is NOT accessible by threads in other Blocks
- Each thread in the same Block only access registers assigned to itself

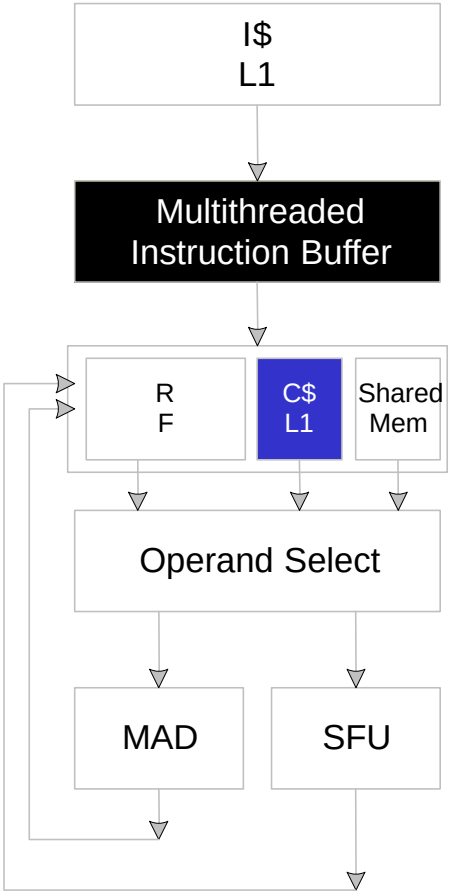


Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism

Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
  - 64KB total in DRAM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!



Shared Memory

Each SM has 16 KB of Shared Memory

- 16 banks of 32bit words

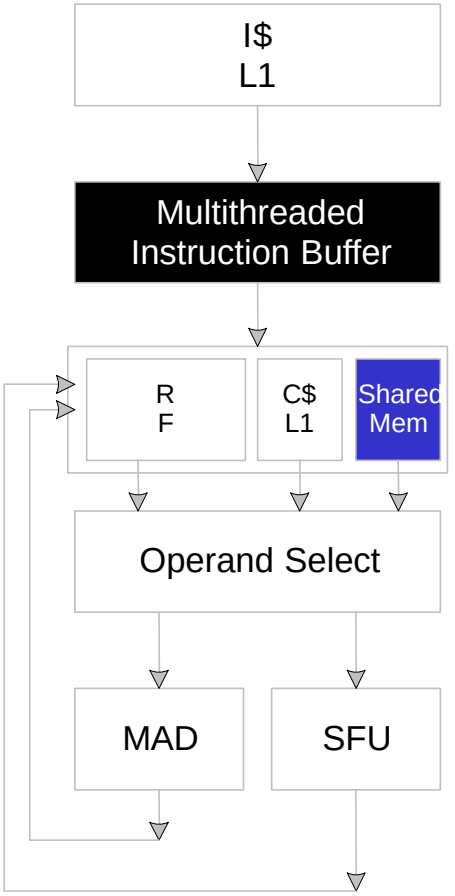
CUDA uses Shared Memory as shared storage visible to all threads in a thread block

- read and write access

Key Performance Enhancement

Move data in Shared memory

Operate in there



Parallel Memory Architecture

In a parallel machine, many threads access shared memory

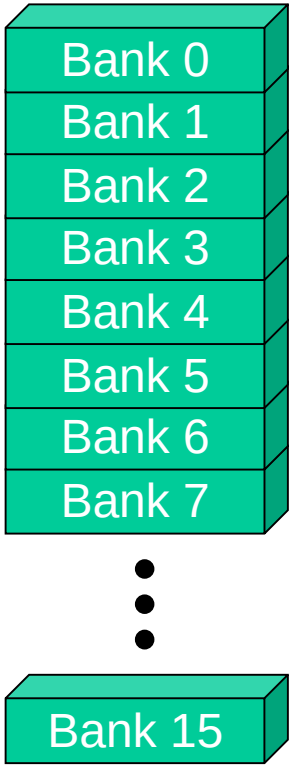
- Therefore, memory is divided into banks
- Essential to achieve high bandwidth

Each bank can service one address per cycle

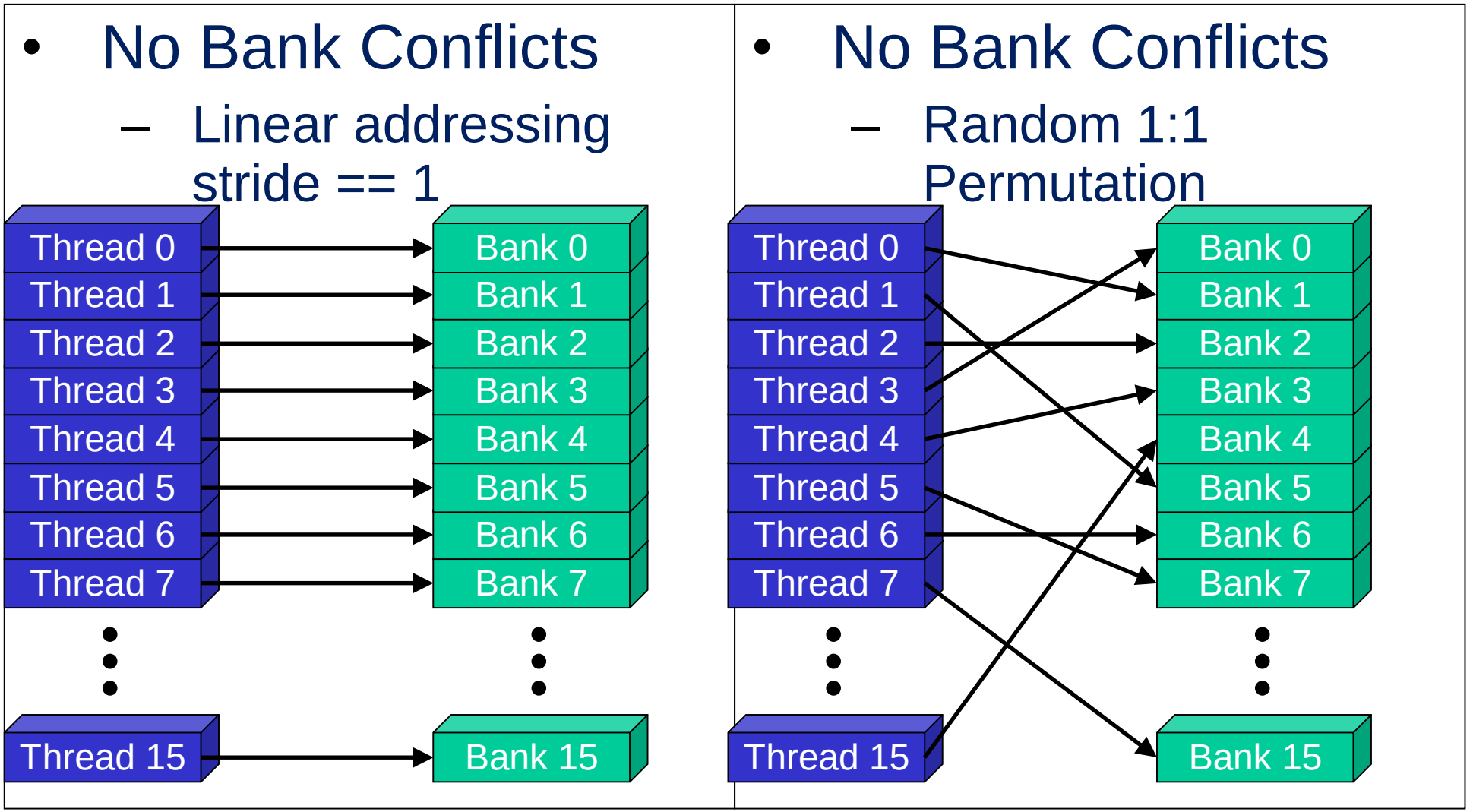
- A memory can service as many simultaneous accesses as it has banks

Multiple simultaneous accesses to a bank result in a bank conflict

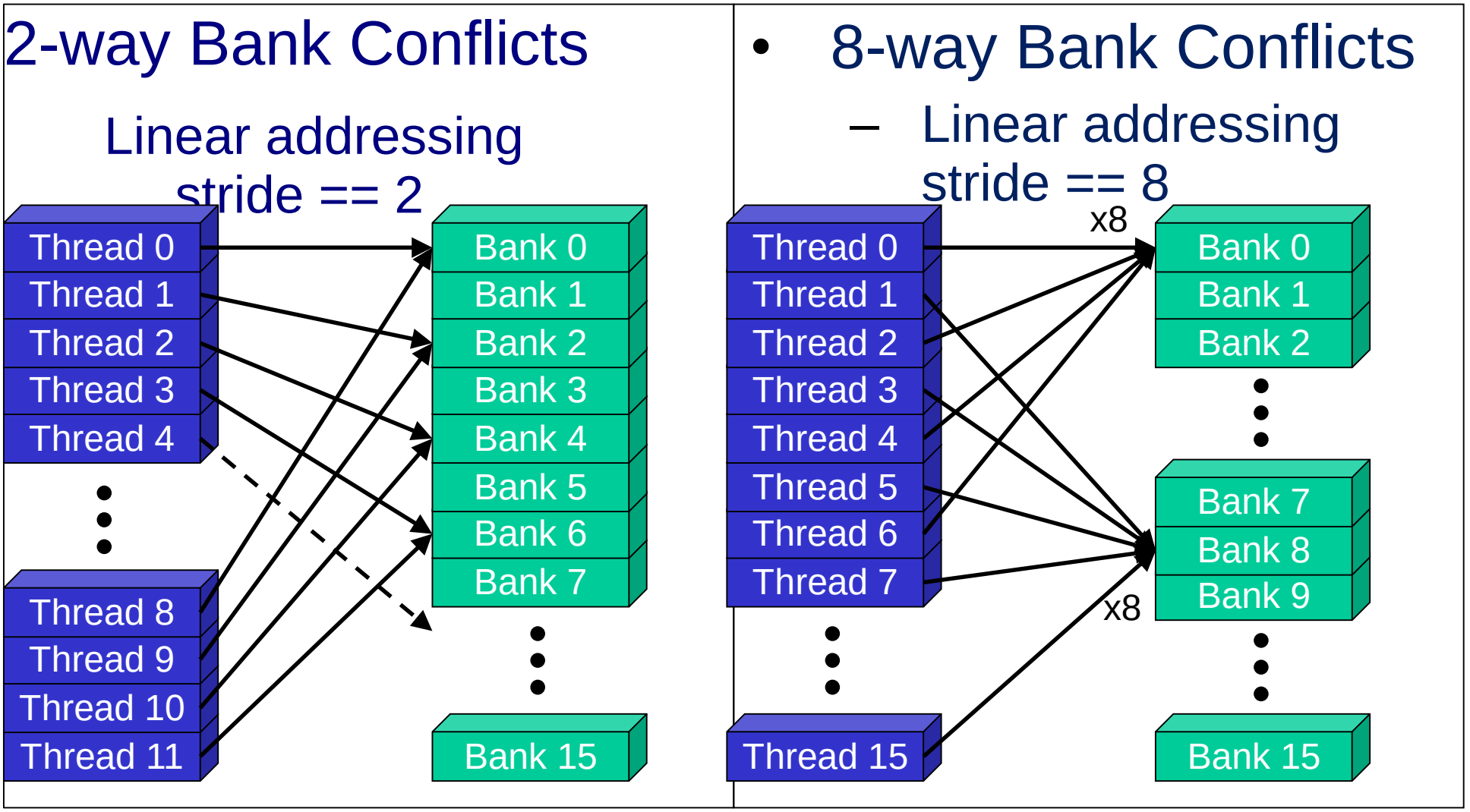
- Conflicting accesses are serialized



Bank Addressing Examples



Bank Addressing Examples



Example: how addresses map to banks

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- Assume memory has 16 banks
  - bank = (word address) % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

Shared memory is as fast as registers if there are no bank conflicts

The fast case:

- If all threads of a half-warp access different banks, there is no bank conflict
- If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)

The slow case:

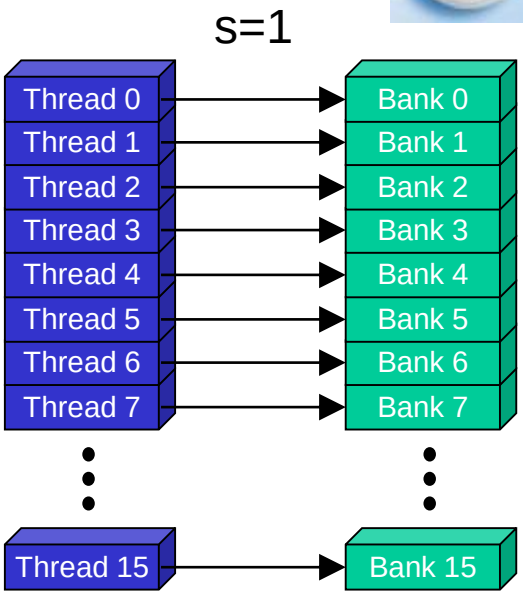
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank

Linear Addressing

Given:

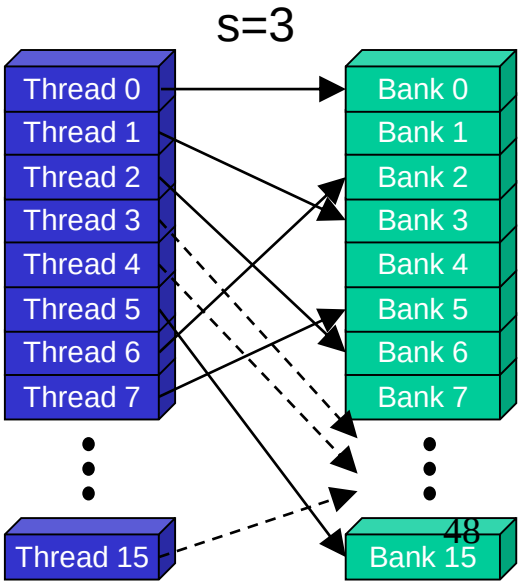
```
__shared__ float buffer[256];

float foo = buffer[baseIndex + s *
   threadIdx.x];
```



This is only bank-conflict-free if s shares no common factors with the number of banks

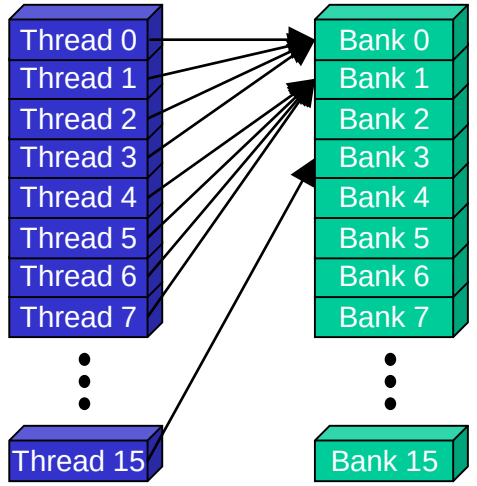
Here, `s` must be odd



Data types and bank conflicts

This has no conflicts if type of `shared` is 32-bits:

```
__shared__ float buffer[256];
foo = buffer [baseIndex + threadIdx.x]
```



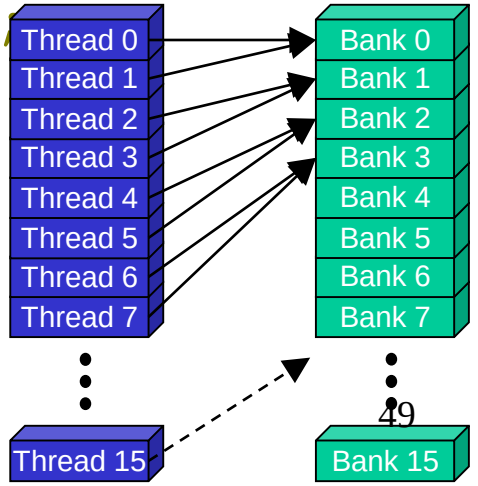
But not if the data type is smaller

4-way bank conflicts:

```
__shared__ char buffer[256];
foo = buffer [baseIndex + threadIdx.x];
```

2-way bank conflicts:

```
__shared__ short buffer[256];
foo = buffer[baseIndex + threadIdx.x];
```



Common Array Bank Conflict Patterns 1D

Each thread loads 2 elements into shared memory:

2-way-interleaved loads result in 2-way bank conflicts:

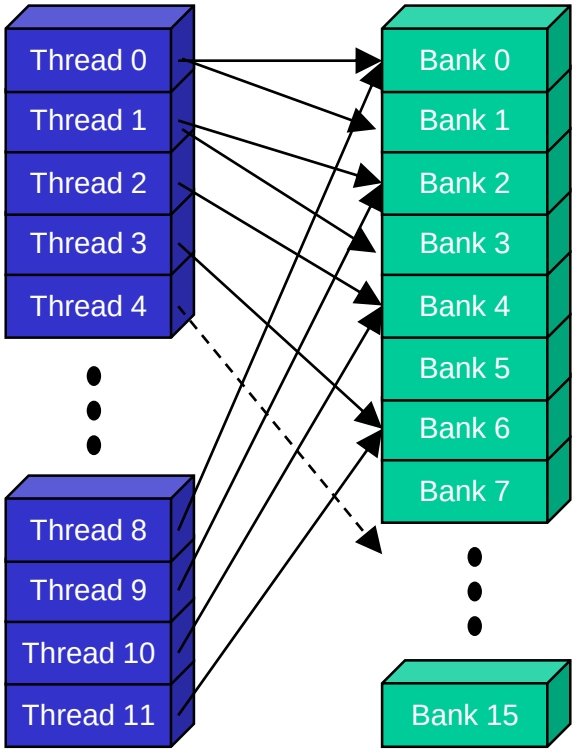
```
int tid = threadIdx.x;

shared[2*tid] = global[2*tid];

shared[2*tid+1] = global[2*tid+1];
```

This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.

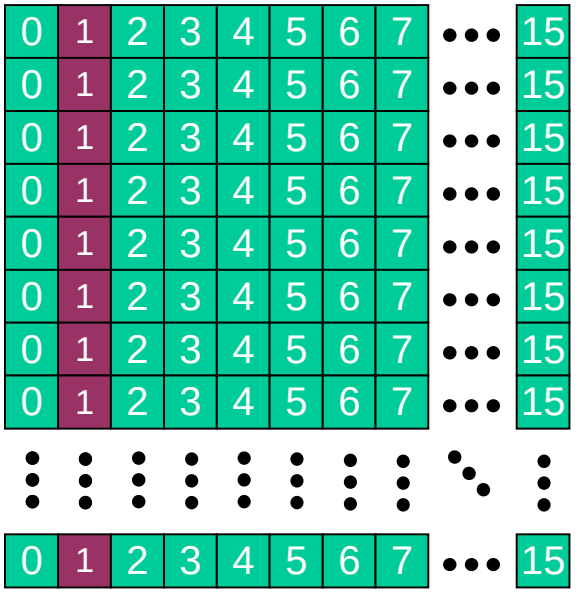
Not in shared memory usage where there is no cache line effects but banking effects



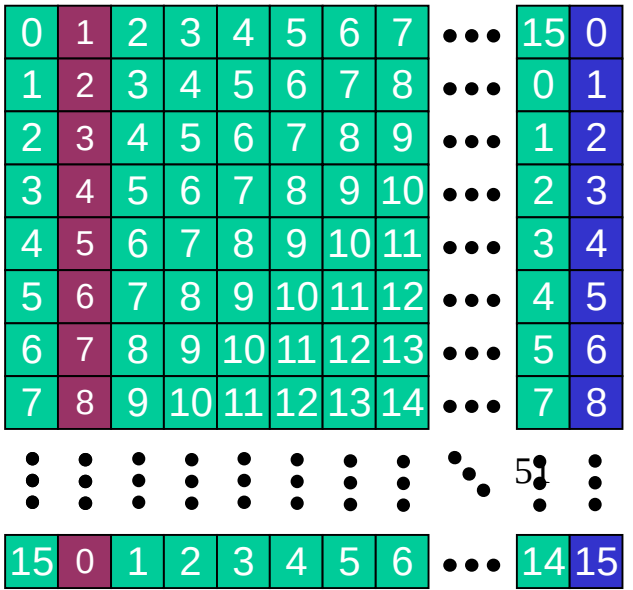
Common Bank Conflict Patterns (2D)

- Operating on 2D array of floats in shared memory
  - e.g., image processing
- Example: 16x16 2D array, 16 threads in block
  - Each thread processes a row
  - So threads in a block access the elements in each column simultaneously (example: column1 in purple)
  - 16-way bank conflicts: rows all start at bank 0
- Solution 1) pad the rows
  - Add one float to the end of each row
- Solution 2) transpose before processing
  - Suffer bank conflicts during transpose
  - But possibly save them later

Bank Indices without Padding



Bank Indices with Padding



Load/Store (Memory read/write) Clustering/Batching

Use LD to hide LD latency (non-dependent LD ops only)

- Use same thread to help hide own latency

Instead of:

- LD 0 (long latency)
- Dependent MATH 0
- LD 1 (long latency)
- Dependent MATH 1

Do:

- LD 0 (long latency)
- LD 1 (long latency - hidden)
- MATH 0
- MATH 1

Compiler handles this!

- But, you must have enough non-dependent LDs and Math