Unit -II

SIMD

- SIMD Arch have significant DLP
- Single Instruction can launch many data opns
- SIMD is more energy efficient than MIMD
  - MIMD needs to fetch and execute one instruction per data opns.
- SIMD is more attractive for PMDs.
- Advantage of SIMD over MIMD
  - Programmer thinks sequential execution yet achieves parallel speedup by having parallel data operations

SIMD

- SIMD has 3 variations:
  - Vector Architectures
    - Allows pipelined execution of many data operations
  - SIMD MMX
    - Allows simultaneous parallel data operations that support Multimedia applications.
  - GPU Architectures
    - They offer higher performance than traditional multicore
    - They have system processor, system memory & graphics memory.

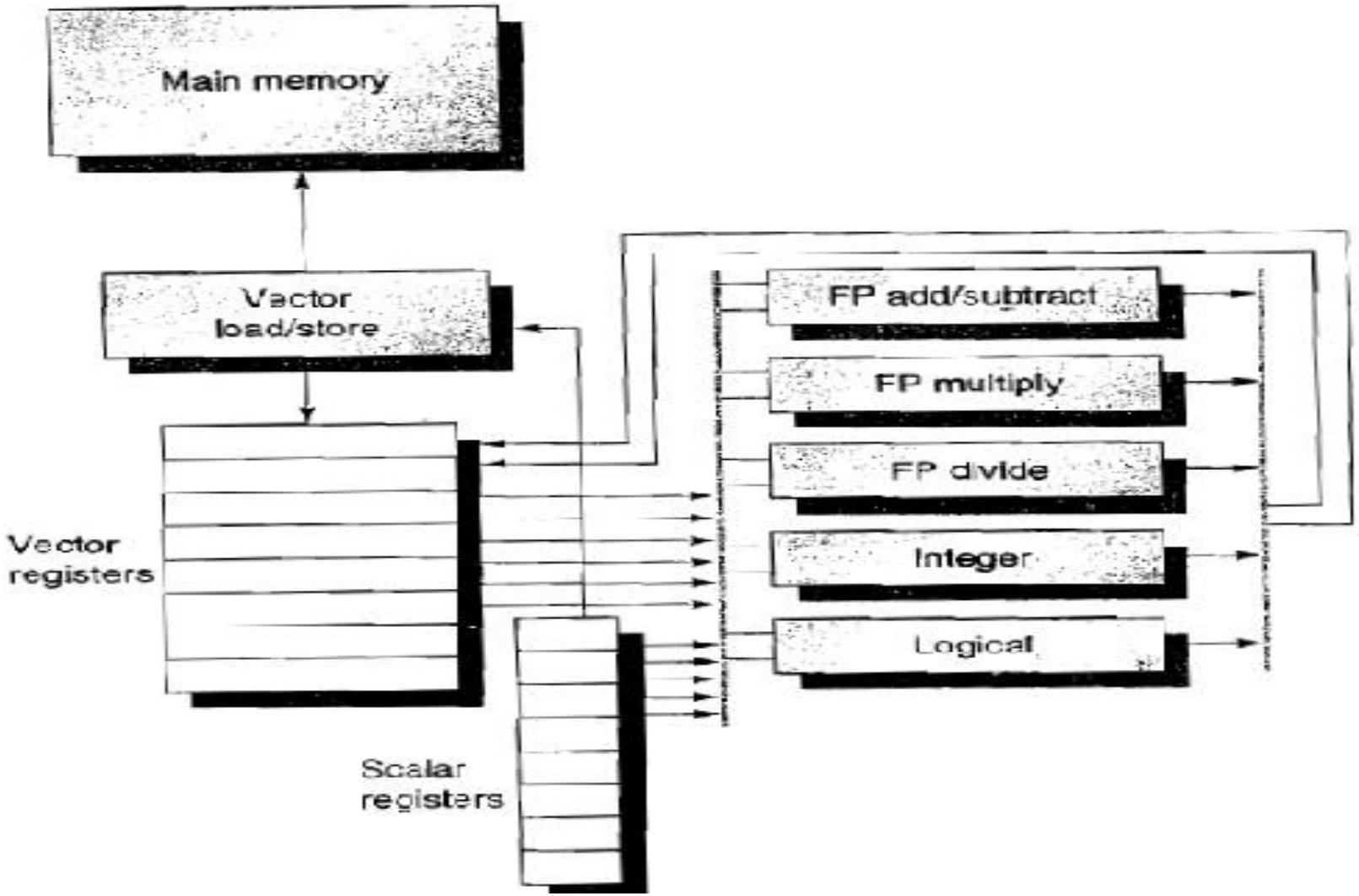
Vector Processor

- Efficient way to execute a vectorizable application is by Vector processor- Jim Smith
- ***vector processor*** is a CPU that executes instructions that operate on arrays of data.
  - It collects set of data elements put them in the register file
  - operates on the data in those register files and stores the results back in memory.
  - These reg files acts like buffers and hide the memory latency.

Vector processor

- SIMD classification
- Also be called as **array processor**
- Improves performance on numerical simulations
- Used in Video game console and Graphic accelerators
- Ex: VIS, MMX, SSE, AltiVec and AVX

VMIPS


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

VMIPS

- It is loosely based on cray-1
- VMIPS instruction set
  - Scalar portion is similar to MIPS
  - Vector portion is logical vector extension of MIPS
- Registers:
  - It has 8 vector registers
  - Fixed length holding
  - Each vector register holds single vector
  - Each vector register holds 64 elements of 64 bits

VMIPS

- Vector registers
  - Vector register file has 16 read and 8 write ports
  - Supply operands to VFUs.
  - Registers and the FUs are connected by a pair of cross bar switches (thick gray lines)
- Scalar registers
  - 32 GPRs and 32 FPRs as in MIPS.
  - These supply operands to VFUs
  - Supply addresses to L/S units.

VMIPS

- Vector functional units
  - Each unit is fully pipelined
  - start a new operation on every clock cycle
  - Control Unit is needed to detect hazards
    - structural hazards for functional units
    - data hazards on register accesses

VMIPS

- VMIPS has five functional units
  - Integer unit
  - Logical Unit
  - Floating point Add/Sub
  - Floating point Multiply
  - Floating point Divide

VMIPS

- VMIPS has scalar architecture like MIPS.
- Vector load/store unit-
  - vector L/S unit loads/stores a vector to/from memory.
  - This unit is fully pipelined
  - words can be moved b/w the vector reg and memory one word per clock cycle
  - This unit also handles scalar loads and stores.

How Vector Processors Work: An Example

- Let's take a typical vector problem Y = a* X+ Y
- X and Y are vectors resident in memory
- a is a scalar
- This problem is the so-called SAXPY or DAXPY
  - SAXPY –single precision a * X plus Y
  - DAXPY -double precision a * X plus Y

How Vector Processors Work: An Example

- MIPS code for the DAXPY loop

```
        L.D     FO,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,FO    ;a x X[i]
        L.D     F4,0(Ry)    ;load Y[i]
        ADD.D   F4,F4,F2    ;a x 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
```

How Vector Processors Work: An Example

- VMIPS code for DAXPY
  - L.D FO,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
  - SV V4,Ry ;store the result

How Vector Processors Work: An Example

- vector processor reduces the instruction bandwidth
- executes only 6 instructions vs almost 600 for MIPS
- It occurs because the vector operations work on 64 elements
- overhead instructions that constitute half the loop on MIPS are not present in the VMIPS code
- compiler produces vector instructions for such a sequence
- resulting code spends its time running in vector mode , the code is said to be vectorized or vectorizable.

How Vector Processors Work: An Example

- Loops can be vectorized when they do not have dependences between iterations of a loop, are called loop-carried dependences.
- Another important difference between MIPS and VMIPS is the frequency of pipeline interlocks.
- In the MIPS code, every ADD. D must wait for a MUL. D, and every S. D must wait for the ADD. D.
- On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline.

How Vector Processors Work: An Example

- Vector architects call forwarding of element dependent operations chaining, in that the dependent operations are "chained" together.
- Thus, pipeline stalls are required only once per vector instruction, rather than once per vector element.

Vector Execution Time

- **Vector Execution time** depends on 3 factors:
  - length of the operand vectors
  - structural hazards among the operations
  - data dependences
- Given the vector length and the initiation rate, the rate at which a vector unit consumes new operands and produces new results, we can compute the time for a single vector instruction.
- All modern vector computers have vector functional units with multiple parallel pipelines (or lanes) that can produce two or more results per clock cycle

Vector Execution Time

- **convoy :**
  - is the set of vector instructions that could execute together.
  - The instructions in a convoy must not contain any structural hazards
  - if such hazards were present, the instructions would need to be serialized and initiated in different convoys.
  - we assume that a convoy of instructions must complete execution before any other instructions can begin execution.
  - vector instruction sequences with structural hazards sequences should be in separate convoys

Vector Execution Time

- **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.
  - allows them to be in the same convoy

Vector Execution Time

- **chime**
  - a timing metric to estimate the time for a convoy
  - simply the unit of time taken to execute one convoy
  - vector sequence that consists of *m* convoys executes in *m* chimes for a vector length of n
  - for VMIPS this is approximately in m x n clock cycles.
  - measuring time in chimes is a better approximation for long vectors.
  - If we know the number of convoys in a vector sequence, we know the execution time in chimes

Vector Execution Time

- Show how the following code sequence lays out in convoys, assuming a single copy of each vector functional unit:
  - LV V1,Rx ;load vector X
  - MLILVS.D V2,V1,F0 ;vector-scalar multiply
  - LV V3,Ry ;load vector Y
  - ADDVV.D V4,V2,V3 ;add two vectors
  - SV V4,Ry ;store the sum
- How many chimes will this vector sequence take? How many cycles per FLOP (floating-point operation) are needed, ignoring vector instruction issue overhead?

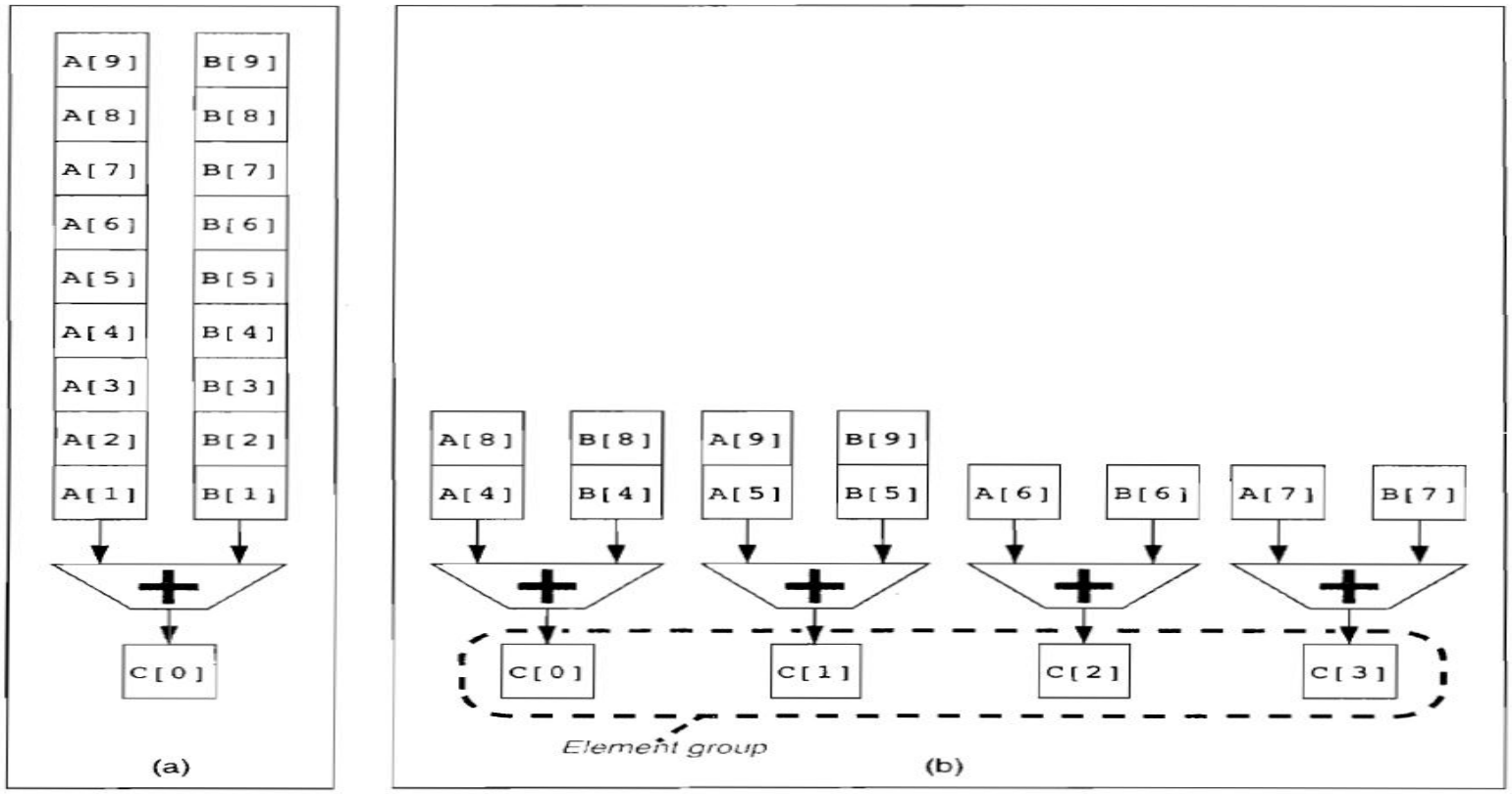
Vector Execution Time

- The first convoy starts with the first LV instruction. The MULVS. D is dependent on the first LV, but chaining allows it to be in the same convoy.
- The second LV instruction must be in a separate convoy since there is a structural hazard on the load/store unit for the prior LV instruction
- The ADDVV. D is dependent on the second LV, but it can again be in the same convoy via chaining
- Finally, the SV has a structural hazard on the LV in the second convoy, so it must go in the third convoy

Vector Execution Time

- The sequence requires three convoys
  - LV MULVS.D
  - LV ADDVV.D
  - SV
  - Since the sequence takes three chimes and there are two floating-point operations per result
  - number of cycles per FLOP is 1.5 (ignoring vector instruction issue overhead).
  - we allow the LV and MULVS.D both to execute in the first convoy,
  - the chime approximation is reasonably accurate for long vectors
  - for 64-element vectors, the time in chimes is 3, so the sequence would take about 64 x 3 or 192 clock cycles.

Multiple Lanes



multiple FUs to improve the performance of a single vector add instruction, C = A + B

Multiple Lanes

- The vector processor
  - (a) single add pipeline and can complete one addition per cycle.
  - (b) four add pipelines and can complete four additions per cycle.
  - The elements within a single vector add instruction are interleaved across the four pipelines.
  - The set of elements that move through the pipelines together is termed an element group
  - Going to four lanes from one. lane reduces the number of clocks for a chime from 64 to 16.
  - For multiple lanes to be advantageous, both the applications and the architecture must support long vectors

Multiple Lanes

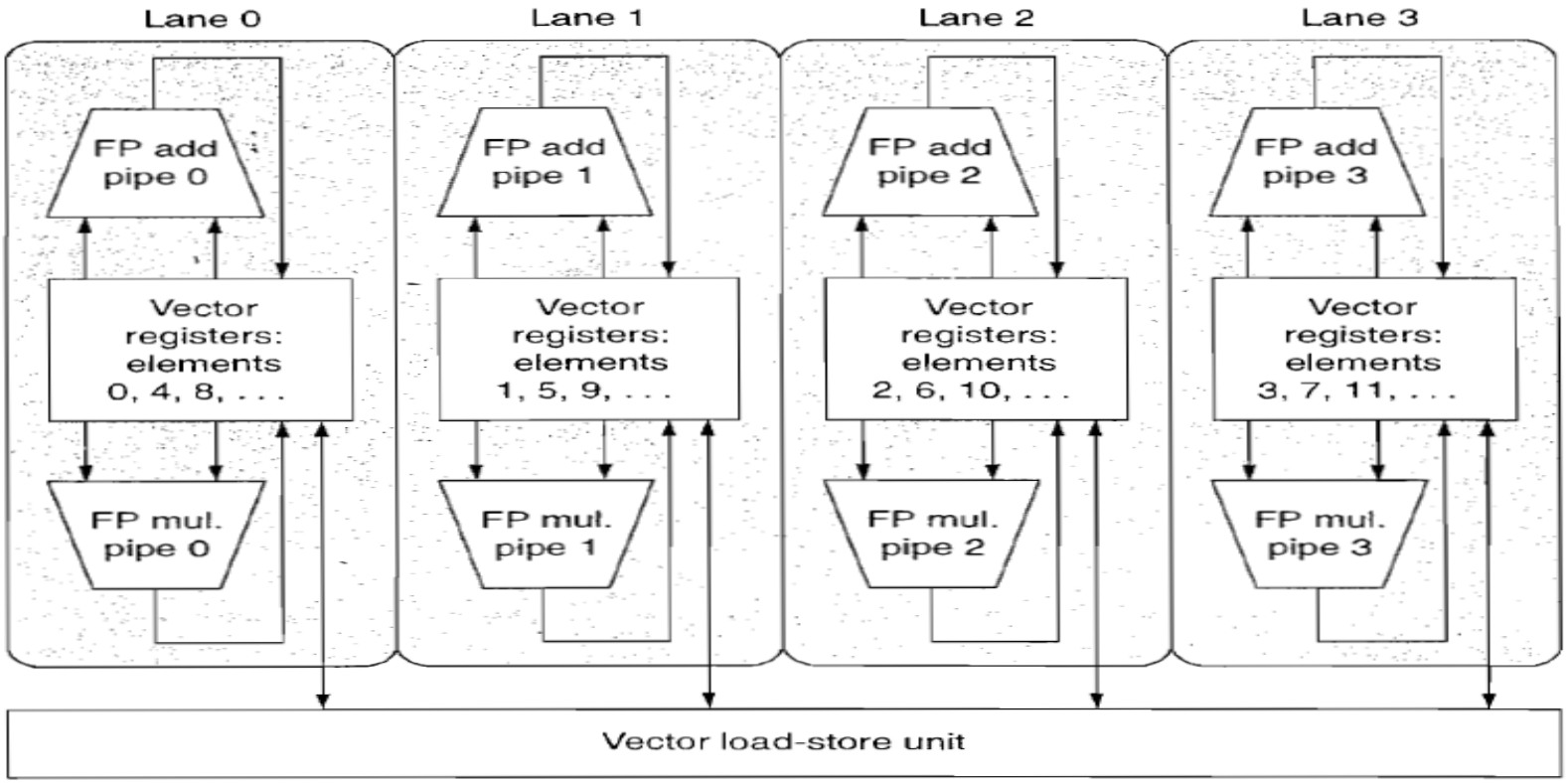


Fig: Structure of a vector unit containing four lanes

Multiple Lanes

- Each lane contains one portion of the vector reg file and one execution pipeline from each vector FU.
- Each vector FU executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane.
- The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane.

Multiple Lanes

- Vector register storage is divided across the lanes, with each lane holding every fourth element of each vector register
- three vector functional units:
  - an FP add
  - an FP multiply
  - a load-store unit.
- Adding multiple lanes is a popular technique to improve vector performance.

Vector-Length Registers

- A vector registers processor has a natural vector length determined by the number of elements in each vector register.
- This length, which is 64 for VMIPS
- In a real program the length of a particular vector operation is often unknown at compile time

Vector Length Register

- single piece of code may require different vector lengths

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

- size of all the vector operations depends on n
- value of n might subject to change during execution

vector-length register

- vector-length register (VLR)
  - controls the length of any vector operation
  - value in the VLR cannot be greater than the length of the vector registers
  - This solves our problem as long as the real length is less than or equal to the maximum vector length (MVL)

vector-length register

- if the value of n is greater than the MVL
- ***strip mining*** is the generation of code such that each vector operation is done for a size less than or equal to the MVL.
- We create one loop that handles any number of iterations that is a multiple of the MVL
- another loop that handles any remaining iterations and must be less than the MVL.

Vector Mask Registers

- The presence of conditionals (IF statements) inside loops and the use of sparse matrices are two main reasons for lower levels of vectorization.
- Consider the following loop written in C:

```
For(I =0; i< 64;   i=i+1)
    if(X[i]!=0)
    X[i]= X[i]- Y[i];
```

Vector Mask Registers

- This loop cannot normally be vectorized because of the conditional execution of the body
- Mask registers essentially provide conditional execution of each element operation.
- The vector-mask control uses a Boolean vector to control the execution of a vector instruction

Vector Mask Registers

- When the vector mask register is enabled, any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are one.
- The entries in the destination vector register that correspond to a zero in the mask register are unaffected by the vector operation

Vector Mask Registers

```
 LV          V1,Rx    ;load vector X into V1
 LV          V2,Ry    ;load vector Y
L.D          FO,#O    ;load FP zero into FO
SNEVS.D      V1,FO    ;sets VM(i) to1if V1(i)!=F0
 SLIBVV.D    V1,V1,V2;subtract under vector mask
SV           V1,Rx    ;store the result in X
```

Memory Banks

- penalties for start-ups on load/store units are higher than those for arithmetic units
- over 100 clock cycles on many processors.
- For VMIPS a start-up time of 12 clock cycles.
- To maintain an initiation rate of one word fetched or stored per clock
  - memory system must be capable of producing or accepting this much data.
  - accesses across multiple independent memory banks usually delivers the desired rate

Memory Banks

- Most vector processors use memory banks, which allow multiple independent accesses rather than simple memory interleaving for three reasons:
  - To support simultaneous accesses from multiple loads or stores, the memory system needs multiple banks and to be able to control the addresses to the banks independently

Memory Banks

- Most vector processors support the ability to load or store data words that are not sequential. In such cases, independent bank addressing, rather than interleaving, is required.
- Most vector computers support multiple processors sharing the same memory system, so each processor will be generating its own independent stream of addresses

SIMD Instruction Set Extensions for Multimedia

- media applications operate on narrower data types.
- Many graphics systems used 8 bits to represent each of the three primary colors plus 8 bits for transparency.
- Depending on the application, audio samples are usually represented with 8 or 16 bits.
- Like vector instructions, a SIMD instruction specifies the same operation on vectors of data.
- SIMD instructions tend to specify fewer operands and hence use much smaller register files this is in contrast to vector arch which has large reg files

SIMD Instruction Set Extensions for Multimedia

- multimedia support for 256-bit-wide operations

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**

SIMD Instruction Set Extensions for Multimedia

- Multimedia SIMD extensions fix the number of data operands in the opcode
- Multimedia SIMD does not offer the more sophisticated addressing modes of vector architectures
- Multimedia SIMD usually does not offer the mask registers to support conditional execution
- 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

SIMD Instruction Set Extensions for Multimedia

- Advanced Vector Extensions (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

SIMD Instruction Set Extensions for Multimedia

AVX Instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD precision	Multiply and subtract four packed double- operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE,
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

Graphics Processing Unit

- **A graphics processing unit** (GPU), is similar CPU
- Designed specifically for performing the complex mathematical and geometric calculations that are necessary for graphics rendering.

Graphics Processing Unit

- A graphics processing unit (GPU) is a computer chip that performs rapid mathematical calculations, primarily for the purpose of rendering images.
- occasionally called **visual processing unit** (**VPU**)
- GPU is able to render images more quickly than a CPU because of its parallel processing architecture
- Nvidia introduced the first GPU, the GeForce 256, in 1999
- Others include AMD, Intel and ARM.
- In 2012, Nvidia released a virtualized GPU, which offloads graphics processing from the server CPU in a virtual desktop infrastructure.

Graphics Processing Unit

- GPUs are used in
  - Embedded Systems
  - Mobile phones
  - Personal computers
  - Workstations
  - Game consoles

GPU Vs CPU

- A GPU is tailored for highly parallel operation while a CPU executes programs serially
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clock speeds
- A GPU is for the most part deterministic in its operation
- GPUs have much deeper pipelines (several thousand stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs

What are GPU's Growth?

- Entertainment Industry has driven the economy of these chips?
  - Males age 15-35 buy $10B in video games / year
- Moore's Law ++
- Simplified design (stream processing)
- Single-chip designs

GPU

- Very Efficient For
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access
- Not Efficient For
  - Double Precision
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

CUDA

- **CUDA -Compute Unified Device Architecture**
  - is a parallel computing platform and programming model created by NVIDIA
  - Implemented by the GPUs
  - CUDA gives developers access to the instruction set and memory of the parallel computational elements in CUDA GPUs.
  - Using CUDA, GPUs become accessible for computation like CPUs.

CUDA

- functions for the GPU (device) and functions for the system processor (host),
- CUDA uses _device_or_global_ for GPU and --host-for the CPU.
- CUDA variables declared in the device or global functions are allocated to the GPU Memory

Definitions

- dimGrid - dimensions of the code (in blocks)
- dimBlock- dimensions of a block (in threads)
- *Blockldx*- identifier for blocks
- *threadIdx*- identifier for threads per block
- *blockDim*- number of threads per block

DAXPY C code

```
/* Sequential code */
// 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]
}
```

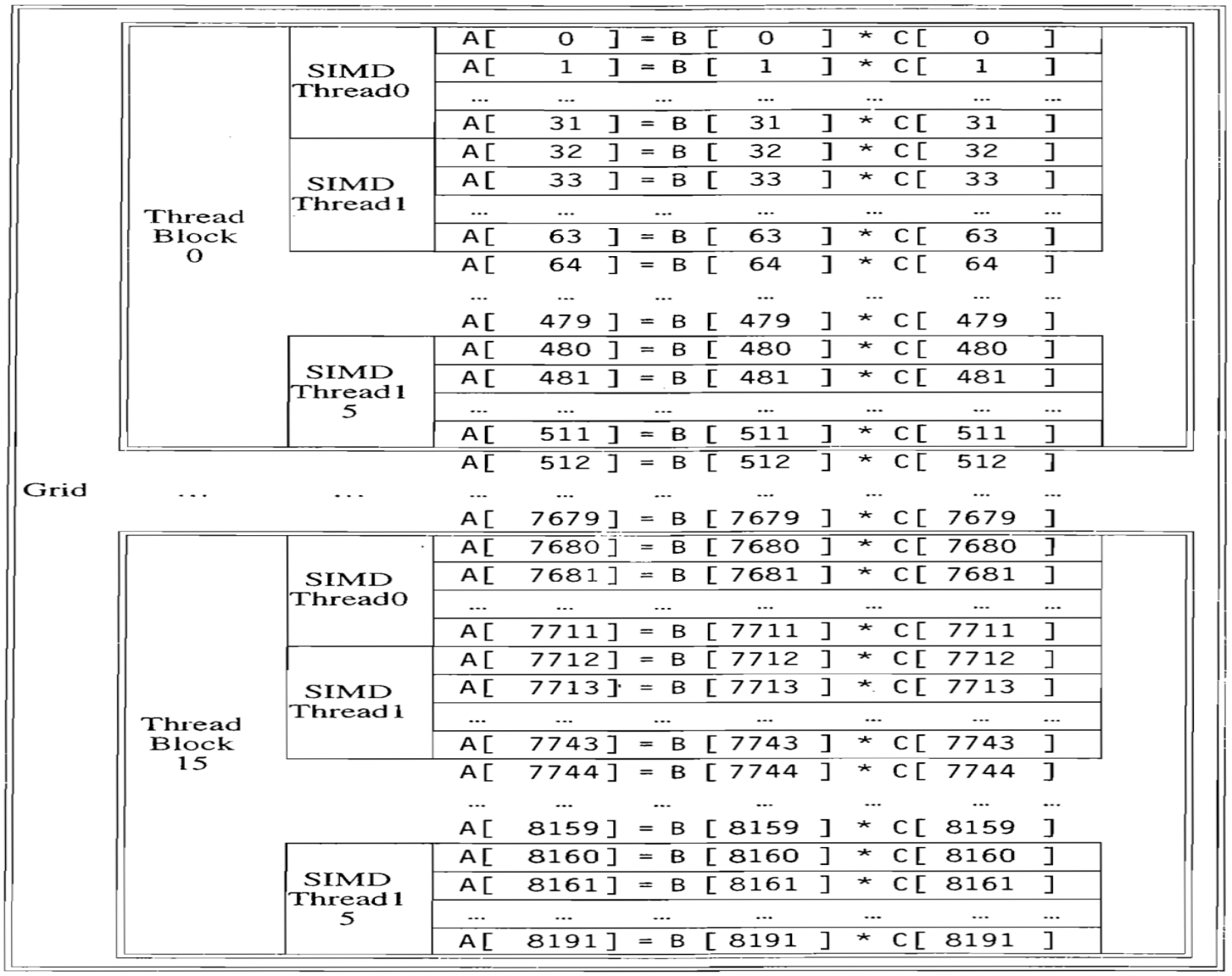
DAXPY CUDA version

```
// Invoke DAXPY with 256 threads per Thread Block _
-host-
                int  nblocks  =(n+255)/256;
                daxpy<<<nblocks,256>>>(n,a, 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];
                }
```

DAXPY

- We launch n threads, one per vector element
- 256 CUDA Threads per thread block
- The GPU function starts by calculating the element index i
  - based on the block ID,
  - the number of threads per block,
  - and the thread ID.
  - As long as this index is within the array (i< n), it performs the multiply and add.

NVIDIA GPUs: Terminology



NVIDIA GPUs: Terminology

- Program abstractions :
- Grid
  - A vectorizable loop executed on GPU made up of one or more thread blocks
- Thread Block
  - A group of threads processing a portion of the loop on MTSIMD processor .
  - They communicate via local memory
- CUDA Thread

–Thread that processes one iteration of the loop executed on one SIMD lane

NVIDIA GPUs: Terminology

- Machine Object
- Warp
  - A thread of SIMD instruction executed on SIMD lane
- PTX instruction
  - Single SIMD instruction executed on SIMD lanes

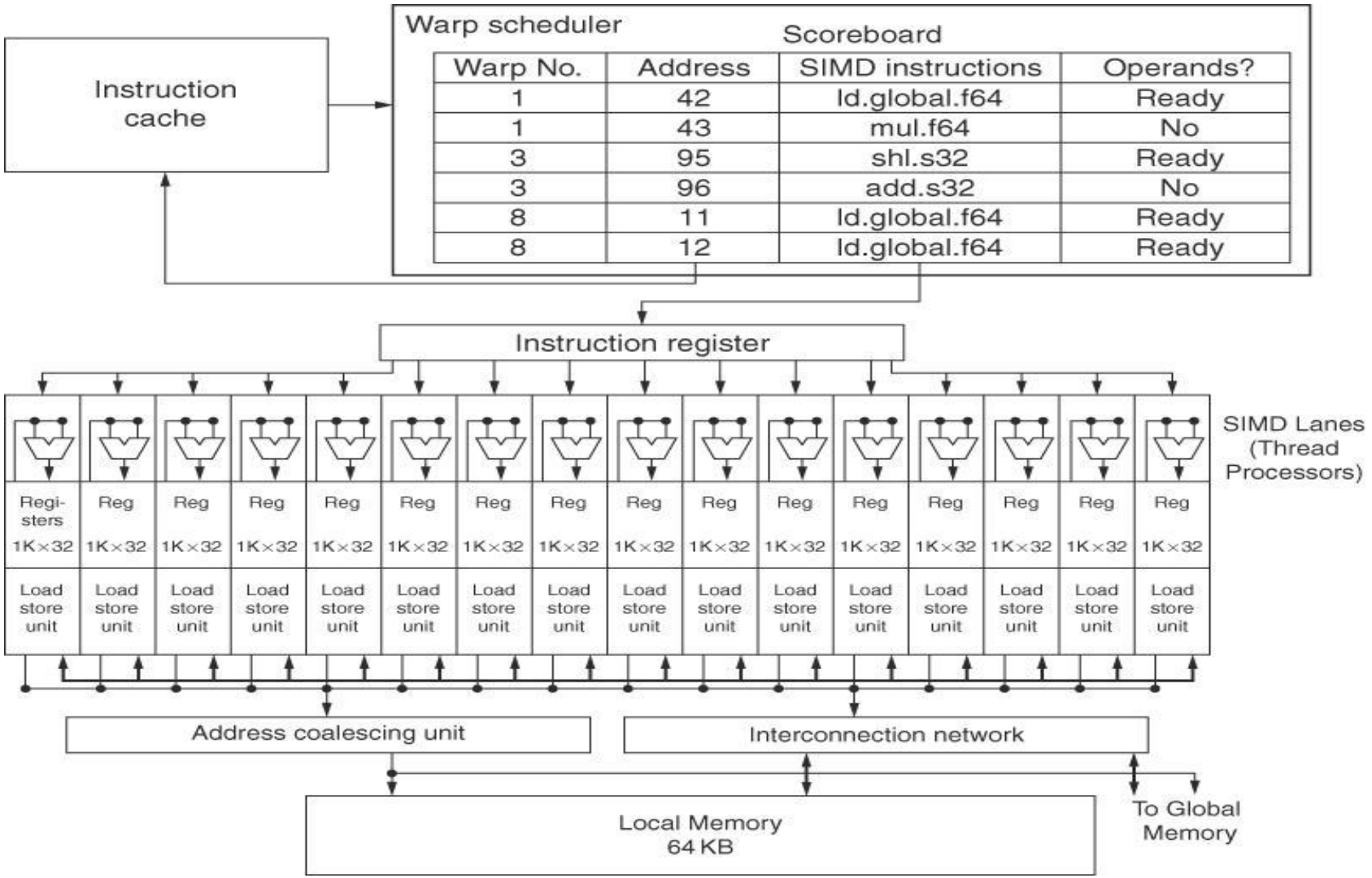
NVIDIA GPUs: Terminology

- Memory hardware
- Global Memory
  - DRAM available to all threads (SIMD processors in GPU)
- Local Memory
  - Private to the thread
- Shared Memory
  - Accessible to all threads of a Streaming Processor
- Thread Processor Registers

NVIDIA GPUs: Terminology

- Processing hardware
- Streaming Multiprocessor
  - Multithreaded SIMD processor executes threads of SIMD instructions
- Giga Thread Engine
  - Thread block scheduler assigns multiple thread blocks to MT SIMD processor
- Warp Scheduler
  - SIMD Thread Scheduler issues threads when they are ready to execute
- Thread Processor
  - SIMD lane executes operations in a thread of SIMD instructions on a single element

NVIDIA GPU –MTSIMD



NVIDIA GPU- MTSIMD

- GPU is a multiprocessor composed of MTSIMD processors.
- It is similar to vector processor but with many parallel FU's that are deeply pipelined.
- MTSIMD is a processor that executes code in the form of thread blocks.
- GPU H/W contains a collection of MTSIMD Processors execute a Grid of Thread Blocks.

NVIDIA GPU- MTSIMD

- GPU H/W has two levels of H/W schedulers

1. Thread Block Scheduler:

  - Thread block scheduler is similar to control unit in Vector processor
  - det the no of thread blocks for a loop and allocates them to diff MTSIMD processors.
  - ensures that thread blocks are assigned to the processors whose local memories have the corresponding data.

NVIDIA GPU- FERMI MTSIMD

2. SIMD Thread Scheduler:

- SIMD Thread scheduler has scoreboard logic
- It keeps track of 48 threads of SIMD instructions
- It tells that which thread of SIMD instructions are ready to run
- It sends those instructions to dispatch unit to be run on MTSIMD processor
- within a SIMD Processor, which schedules when threads of SIMD instructions should run

NVIDIA GPU- MTSIMD

- It has many parallel functional units
- SIMD Processors with separate PCs and are programmed using threads.
- Each MTSIMD Processor is assigned 512 elements of the vectors to work on
- SIMD processors have 32,768 registers
- Like vector processor these registers are logically divided across SIMD lanes.

NVIDIA GPU- MTSIMD

- Each SIMD Thread has 64 vector registers of 32 elements with 32 bit each.
- FERMI has 16 physical lanes each contain 2048 registers
- Thread Blocks would contain 512/32 = 16 SIMD threads.
- Each thread of SIMD instructions in this example compute 32 of the elements of the computation.

NVIDIA GPU- MTSIMD

- GPU applications have so many threads of SIMD instructions that multithreading can
  - hide the latency to DRAM
  - increase utilization of multithreaded SIMD Processors

NVIDA GPU ISA

- PTX (Parallel Thread Execution) provides a stable instruction set for GPUs
- H/W instruction set is hidden from the programmer
- PTX instructions describe the operations on a single CUDA thread
- PTX uses virtual registers
- Translation to machine code is performed in software

NVIDA GPU ISA

- Format of a PTX instruction is

  opcode.type d, a, b, c;

  – where d is the destination operand; a, b, and c are source operands
- Source operands are 32-bit or 64-bit registers or a constant value. Destinations are registers, except for store instructions.

NVIDA GPU ISA

- the operation type is one of the following:

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	.J16,	J32,	J64	

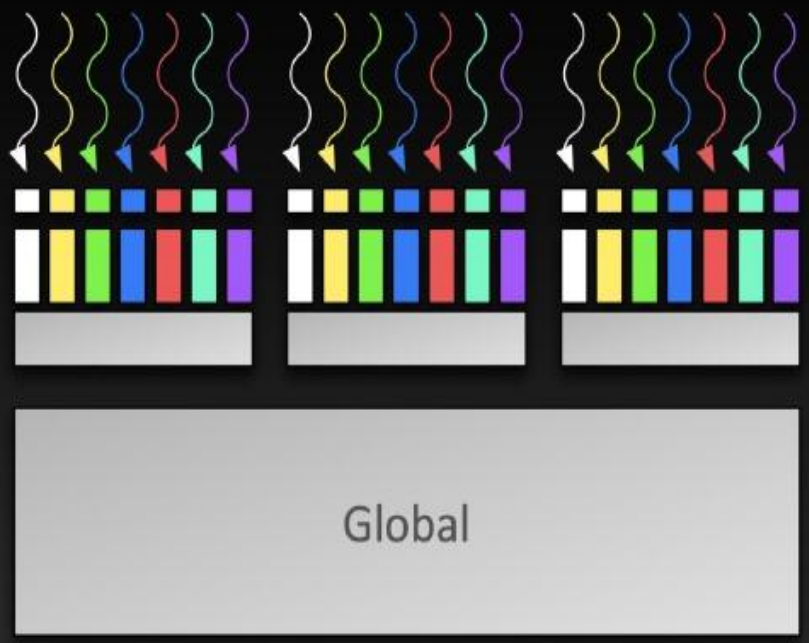
Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - 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

NVIDIA GPU Memory Structures

Memory hierarchy

- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory

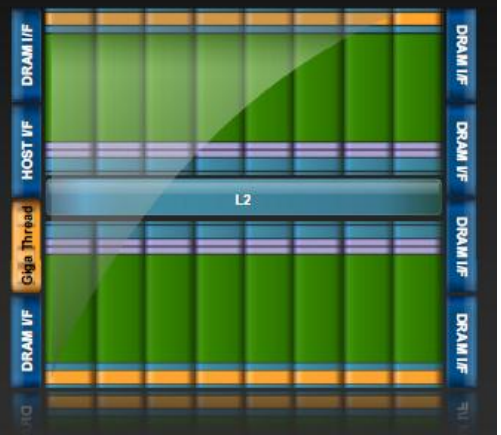


GPU Architecture

GPU Architecture

GPU Architecture: Two Main Components

- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to 250 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



FERMI GPU

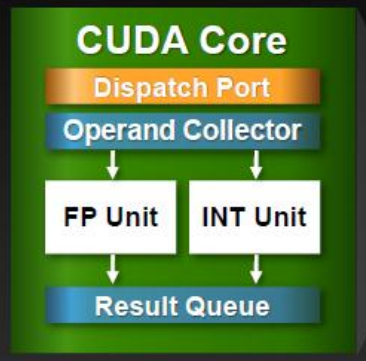
GPU Architecture - Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers

NVIDIA

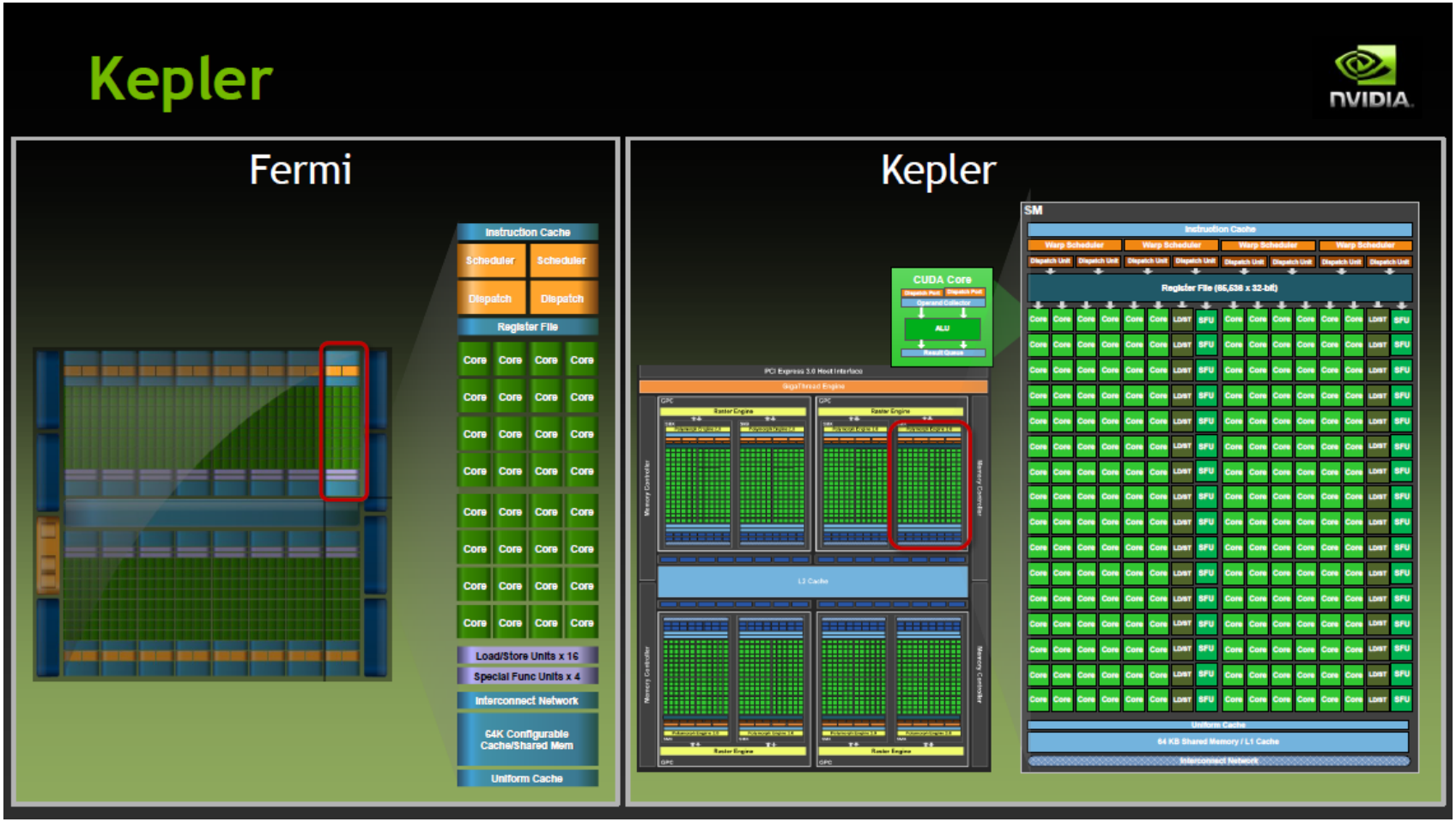
GPU Architecture - Fermi: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit

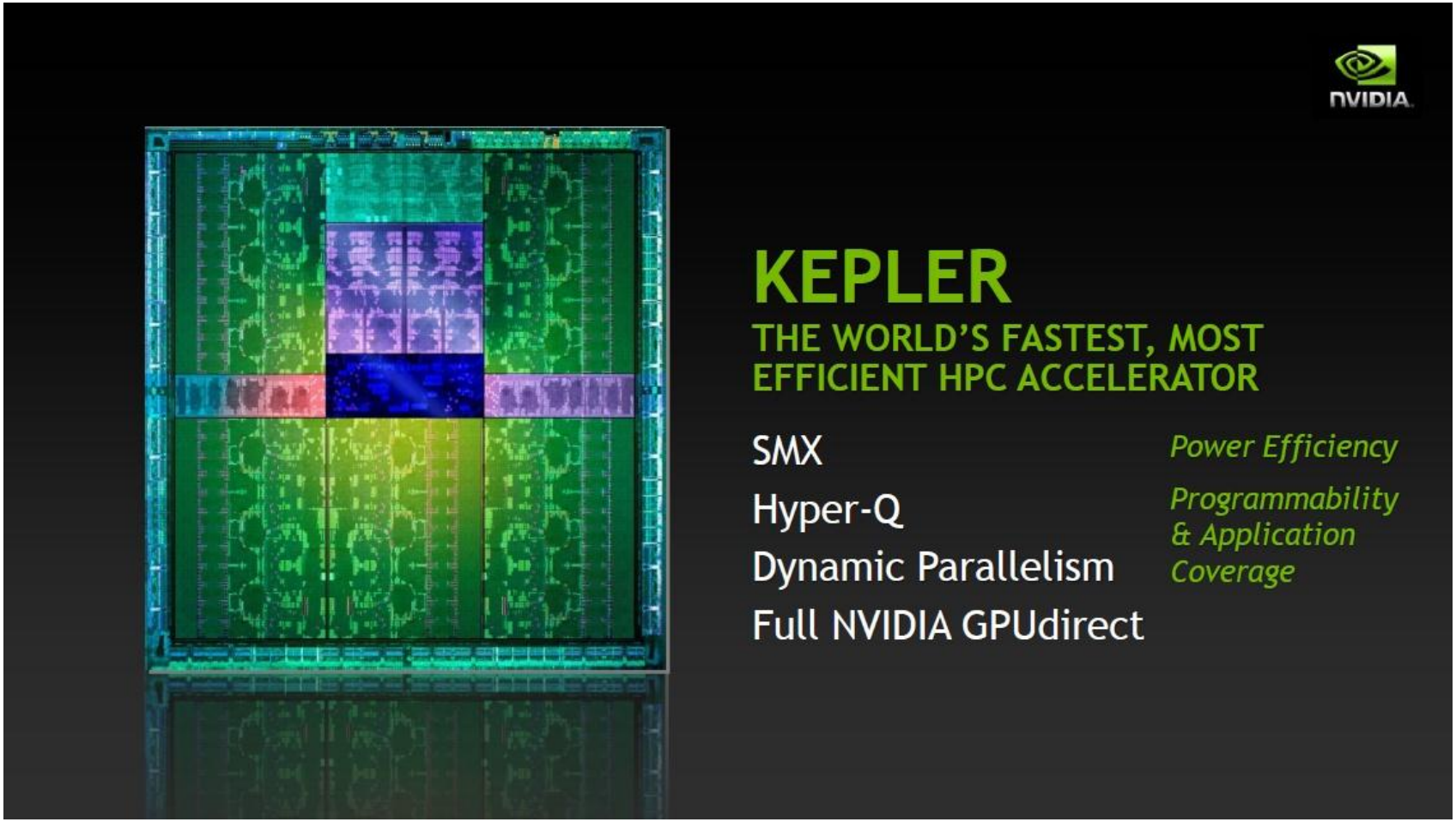


NVIDIA

FERMI vs KEPLER



FERMI vs KEPLER



FERMI vs KEPLER