EE 4683/5683: COMPUTER ARCHITECTURE

Lecture 5B: Data Level Parallelism

Avinash Kodi, kodi@ohio.edu

Thanks to Morgan Kauffman and Krtse Asanovic

Agenda

2

- Flynn's Classification
- Data Level Parallelism
  - Vector Processors
  - SIMD Extensions
  - Graphical Processing Units (GPUs)

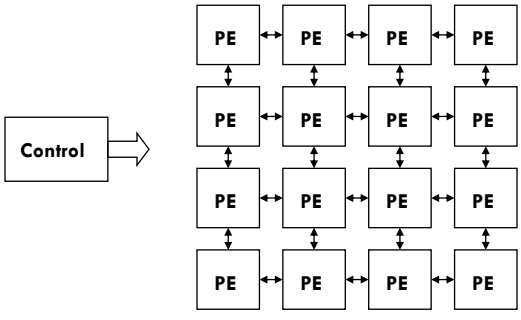
Flynn's Classification Scheme

3

- SISD – single instruction, single data stream
  - aka uniprocessor - what we have been talking about all quarter
- SIMD – single instruction, multiple data streams
  - single control unit broadcasting operations to multiple datapaths
- MISD – multiple instruction, single data
  - no such machine (although some people put vector machines in this category)
- MIMD – multiple instructions, multiple data streams
  - aka multiprocessors (SMPs, MPPs, clusters, NOWs)

SIMD Processors

4



- Single control unit
- Multiple datapaths (processing elements – PEs) running in parallel

Introduction

- SIMD (single instruction multiple data) architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Parallelism

- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!
- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

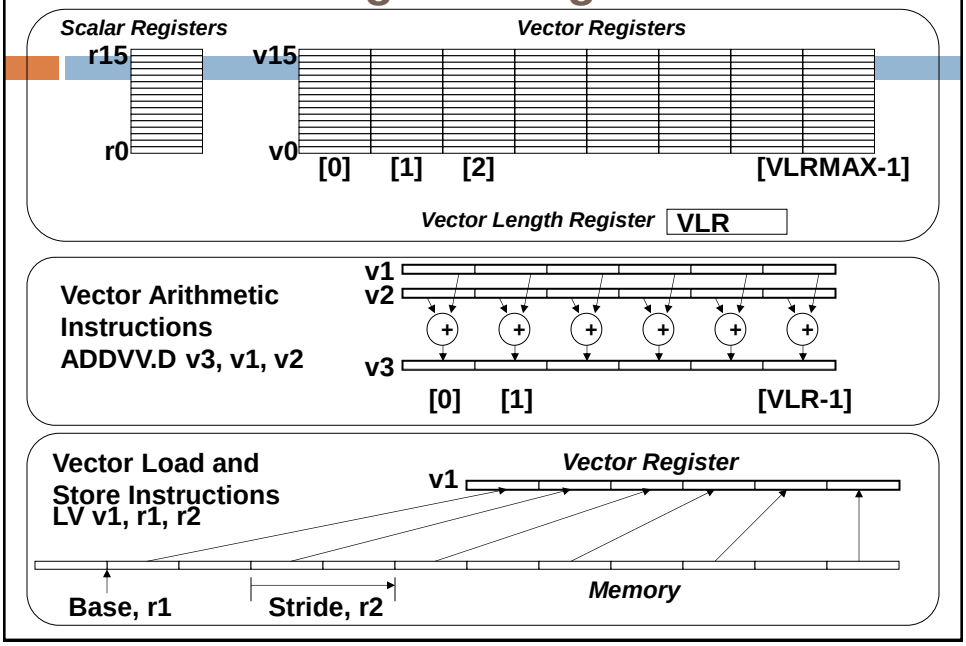
Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

VMIPS

- Example architecture: VMIPS
  - Loosely based on Cray-1
  - Vector registers
    - Each register holds a 64-element, 64 bits/element vector
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers

Vector Programming Model



VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

- Example: DAXPY

```
L.D      F0,a      ; load scalar a
LV       V1,Rx     ; load vector X
MULVS.D  V2,V1,F0  ; vector-scalar multiply
LV       V3,Ry     ; load vector Y
ADDVV.D  V4,V2,V3  ; add
SV       Ry,V4     ; store the result
```

- Requires 6 instructions vs. almost 600 for MIPS

Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

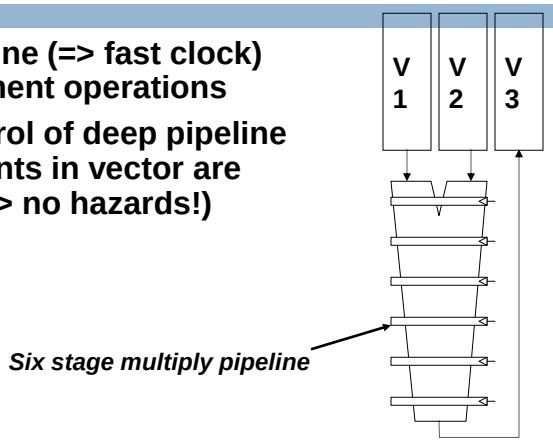
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same object code on more parallel pipelines or *lanes*

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

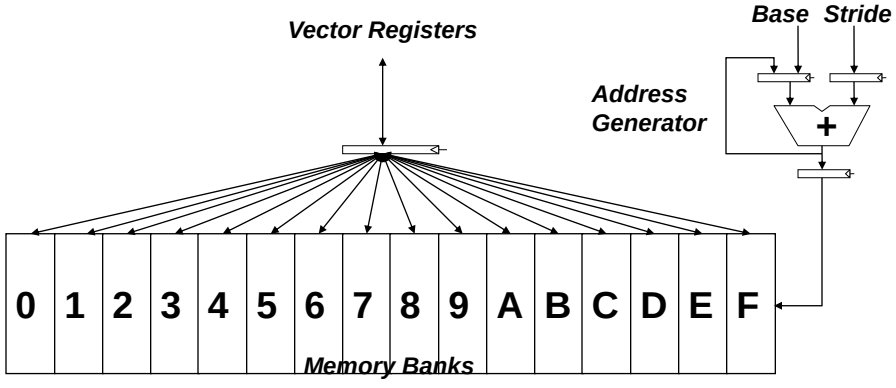



```
V3 <- v1 * v2
```

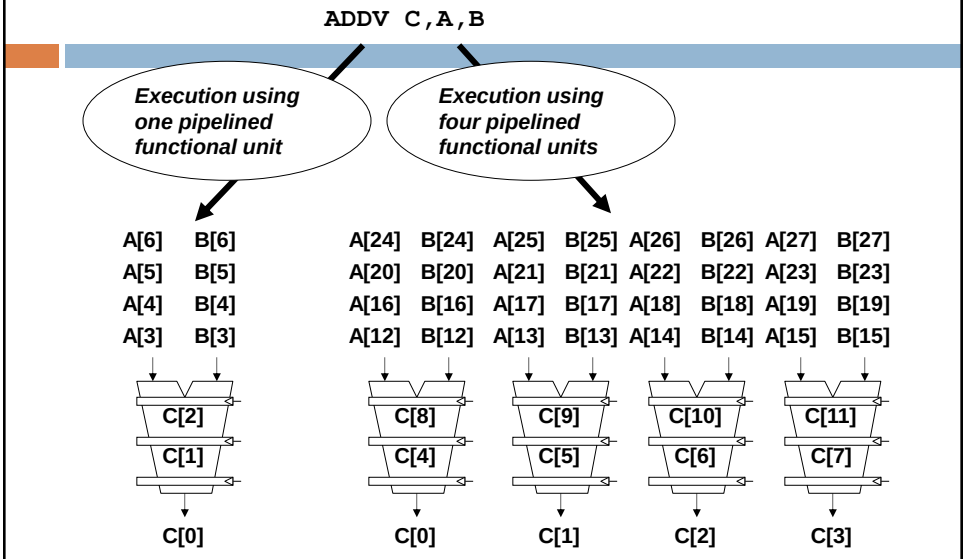
Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

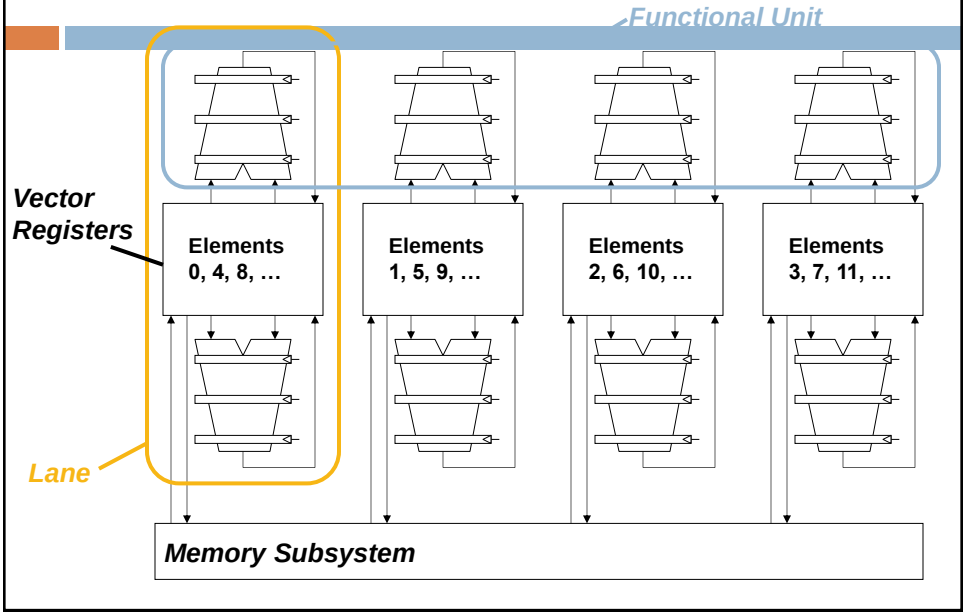
- ***Bank busy time*: Cycles between accesses to same bank**



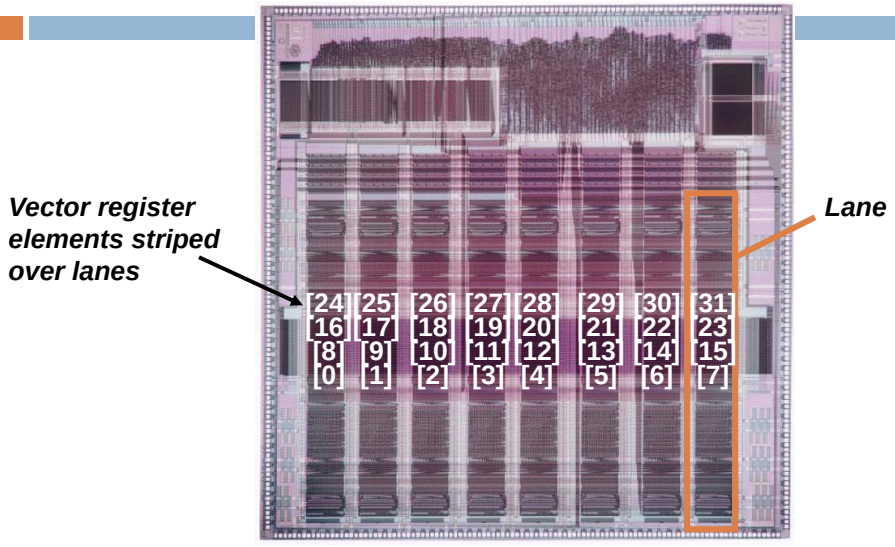
Vector Instruction Execution



Vector Unit Structure



T0 Vector Microprocessor (1995)



Vector Execution Time

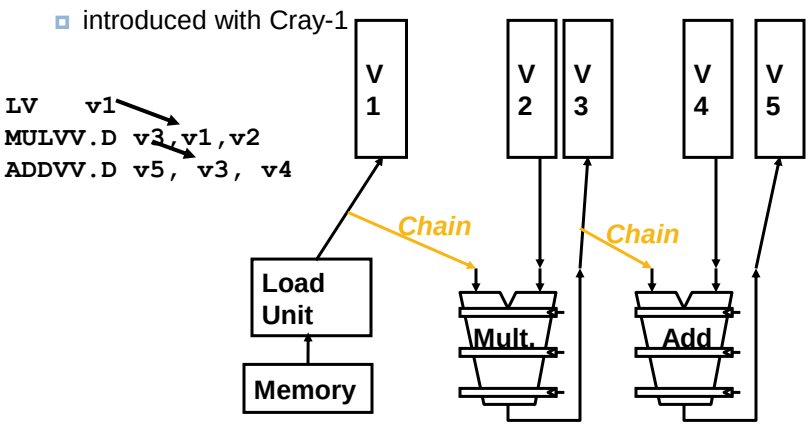
- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convey*
  - Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards can be in the same convey via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
  - Unit of time to execute one convey
  - *m* conveys executes in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

Vector Chaining

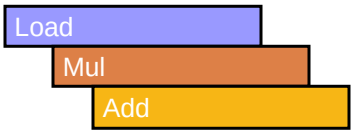
- Vector version of register bypassing
  - introduced with Cray-1



Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction

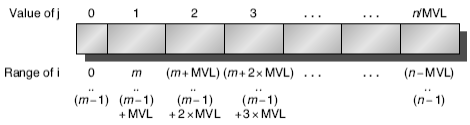
- With chaining, can start dependent instruction as soon as first result appears



Vector Length Register

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

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



Vector Mask Registers

- Consider:

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

- Use vector mask register to “disable” elements:

```
LV          V1,Rx          ;load vector X into V1
LV          V2,Ry          ;load vector Y
L.D         F0,#0          ;load FP zero into F0
SNEVS.D     V1,F0          ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2       ;subtract under vector mask
SV          Rx,V1          ;store the result in X
```

- GFLOPS rate decreases!

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?

Stride

- Consider:

```
for (i = 0; i < 100; i=i+1)
        for (j = 0; j < 100; j=j+1) {
                A[i][j] = 0.0;
                for (k = 0; k < 100; k=k+1)
                A[i][j] = A[i][j] + B[i][k] * D[k][j];
        }
```

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

Scatter-Gather

- Consider:

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

- Use index vector:

```
LV      Vk, Rk          ;load K
LVI     Va, (Ra+Vk)     ;load A[K[]]
LV      Vm, Rm          ;load M
LVI     Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D Va, Va, Vc      ;add them
SVI     (Ra+Vk), Va     ;store A[K[]]
```

Challenges

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

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
  - Operands must be consecutive and aligned memory locations

Example SIMD Code

- Example DXPY:

```
        L.D     F0,a            ;load scalar a
        MOV     F1, F0          ;copy a into F1 for SIMD MUL
        MOV     F2, F0          ;copy a into F2 for SIMD MUL
        MOV     F3, F0          ;copy a into F3 for SIMD MUL
        DADDIU  R4,Rx,#512      ;last address to load
Loop:   L.4D    F4,0[Rx]        ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D  F4,F4,F0  ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D    F8,0[Ry]        ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D  F8,F8,F4  ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D    0[Ry],F8        ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU  Rx,Rx,#32 ;increment index to X
        DADDIU  Ry,Ry,#32 ;increment index to Y
        DSUBU   R20,R4,Rx       ;compute bound
        BNEZ  R20,Loop  ;check if done
```

SIMD Parallelism

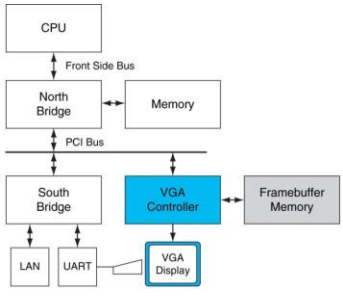
- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

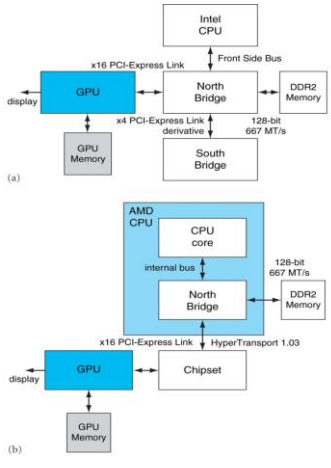
Graphical Processing Units

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

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

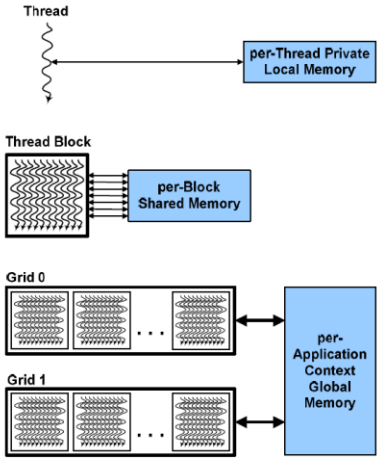
Historical PC vs Intel/AMD CPU+GPU





Threads and Blocks

35



- A thread is associated with each data element
- Executing threads cooperate through barrier synchronization
- Threads are organized into blocks, and blocks into grid
- A grid is an array of thread blocks that execute the same kernel, reads inputs from global memory, write results to global memory and synchronize between dependent kernel calls
- GPU hardware handles thread management, not applications or OS

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

Terminology

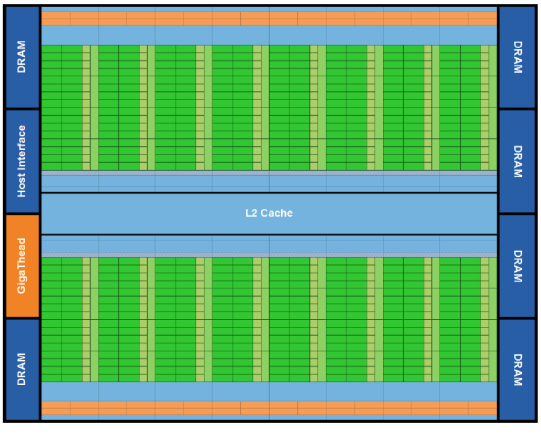
- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

Nvidia Fermi

40



- 3 billion transistors with 512 CUDA cores
- 512 cores organized in 16 SM (stream multiprocessors) of 32 cores
- GPU has six 64-bit memory partitions for a 384-bit memory interface supporting a total of 6 GB of GDDR5 DRAM

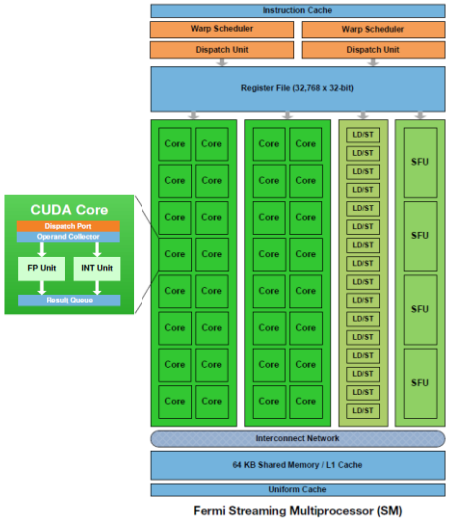
NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - "Private memory"
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

Fermi Architecture Innovations

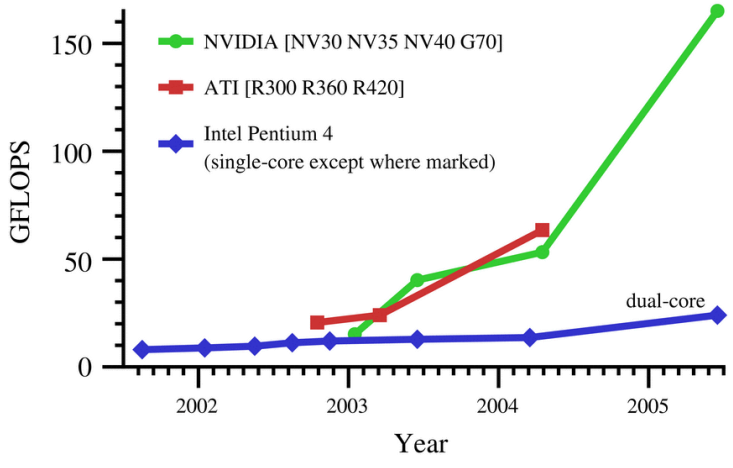
- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

Fermi Multithreaded SIMD Processor



Fermi Streaming Multiprocessor (SM)

Computational Power



Thanks to Ian Buck

Computational Power

45

- GPUs are fast…
  - 3 GHz Pentium4 *theoretical*: 6 GFLOPS, 5.96 GB/sec peak
  - GeForceFX 5900 *observed*: 20 GFLOPs, 25.3 GB/sec peak
- GPUs are getting faster, faster
  - CPUs: annual growth ; 1.5× → decade growth ; 60×
  - GPUs: annual growth > 2.0× → decade growth > 1000
- *Why are GPUs getting faster so fast?*
  - Arithmetic intensity: the specialized nature of GPUs makes it easier to use additional transistors for computation not cache
  - Economics: multi-billion dollar video game market is a pressure cooker that drives innovation

GPGPU

46

- The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
- Example applications range from in-game physics simulation to conventional computational science
- Goal: make the inexpensive power of the GPU available to developers as a sort of computational coprocessor

Difficulties…

47

- GPUs designed for and driven by video games
  - Programming model is unusual & tied to computer graphics
  - Programming environment is tightly constrained
  - Poorly suited for sequential or “pointer-chasing” code
- Underlying architectures are:
  - Inherently parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret
- Can’t simply “port” code written for the CPU!