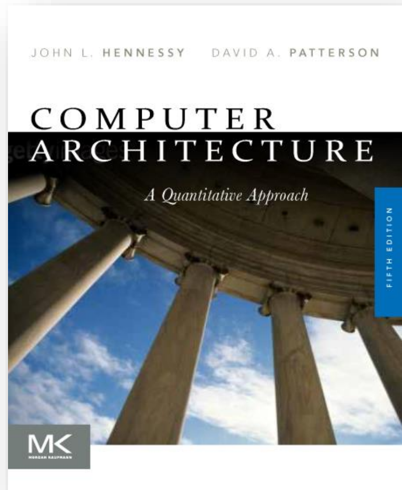


Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

Adapted by Prof. Gheith Abandah

Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions for Multimedia
- Graphics Processing Units
- Fallacies and Pitfalls

MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processing

- 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

Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions for Multimedia
- Graphics Processing Units
- Fallacies and Pitfalls

Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

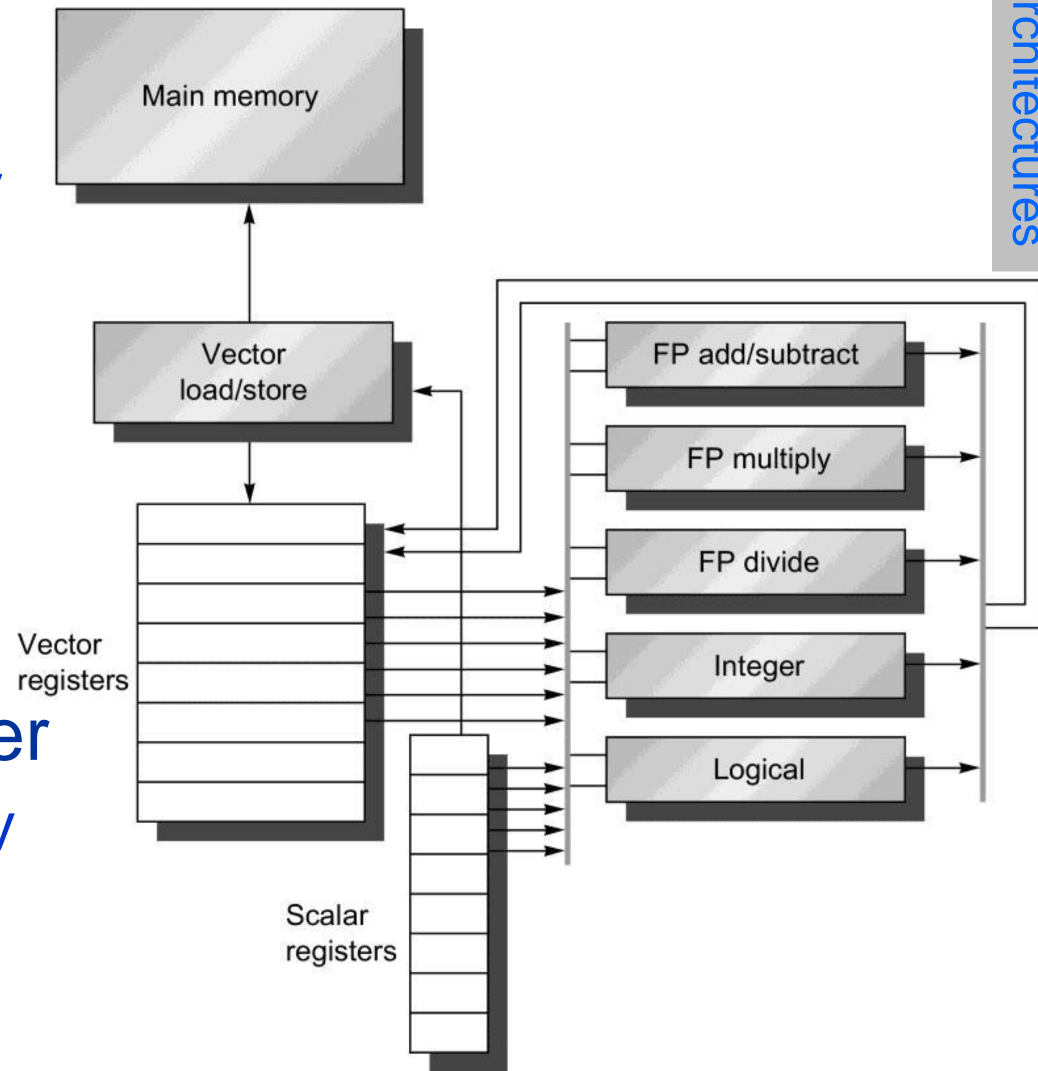
- Vector Architecture
  - Basic Idea
  - RV64V Extensions
  - RV64V Vector Instructions
  - Example: DAXPY
  - Vector Execution Time
  - Challenges
  - Improvements
    1. Multiple Lanes
    2. Vector-Length Registers
    3. Predicate Registers
    4. Memory Banks

MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

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



Copyright © 2019, Elsevier Inc. All rights Reserved

RV64V Extension

- Loosely based on Cray-1
- 32 64-bit vector registers
  - Register file has 16 read ports and 8 write ports
  - Elements per v. register depends on the configuration
- 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

Copyright © 2019, Elsevier Inc. All rights Reserved

RV64V Vector Instructions

- Operate on vectors or vectors and scalars
- Examples: `vadd`, `vmul`, `vld`, `vst`
- Have suffixes
  - `.vv`: two vector operands
  - `.vs` and `.sv`: vector and scalar operands
  - The compiler usually adds these suffixes according to the operands
  - Example: `vmul v1,v0,fo` is `vmul.vs`
- These instructions can operate on the following types:

Integer	8, 16, 32, and 64 bits	Floating point	16, 32, and 64 bits

- The operand type is implicit according to the register type.
- Registers are configurable at run time.

MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Example: DAXPY

- DAXPY: A times X plus Y

- C Code:

```
for (i=0; i<32; i++)
   Y[i]=a*X[i]+Y[i];
```

- RISC-V Code:

```
 fld     f0,a
 addi    x28,x5,#256
Loop:
 fld     f1,0(x5)
 fmul.d  f1,f1,f0
 fld     f2,0(x6)
 fadd.d  f2,f2,f1
 fsd     f2,0(x6)
 addi    x5,x5,#8
 addi    x6,x6,#8
 bne     x28,x5,Loop
```


Copyright © 2019, Elsevier Inc. All rights Reserved

Example: DAXPY

- RV64V Code:

```
vsetdcfg 4*FP64   # Enable 4 DP FP vregs
fld      f0,a     # Load scalar a
vld      v0,x5    # Load vector X
vmul     v1,v0,f0 # Vector-scalar mult
vld      v2,x6    # Load vector Y
vadd     v3,v1,v2 # Vector-vector add
vst      v3,x6    # Store the sum
vdisable          # Disable vector regs
```

- Only 8 instructions

Copyright © 2019, Elsevier Inc. All rights Reserved

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- RV64V 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

Copyright © 2019, Elsevier Inc. All rights Reserved

Chimes

- Sequences with read-after-write dependency hazards placed in 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 *n*
  - For vector length of *n*, requires *m* x *n* clock cycles

MK MORGAN KAUFMANN
Copyright © 2019, Elsevier Inc. All rights Reserved

Example

```
vld  v0,x5          # Load vector X
vmul v1,v0,f0       # Vector-scalar multiply
vld  v2,x6          # Load vector Y
vadd v3,v1,v2       # Vector-vector add
vst  v3,x6          # Store the sum
```

Convoys:

1 `vld` `vmul`

2 `vld` `vadd`

3 `vst`

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 32 element vectors, requires 32 x 3 = 96 clock cycles

MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

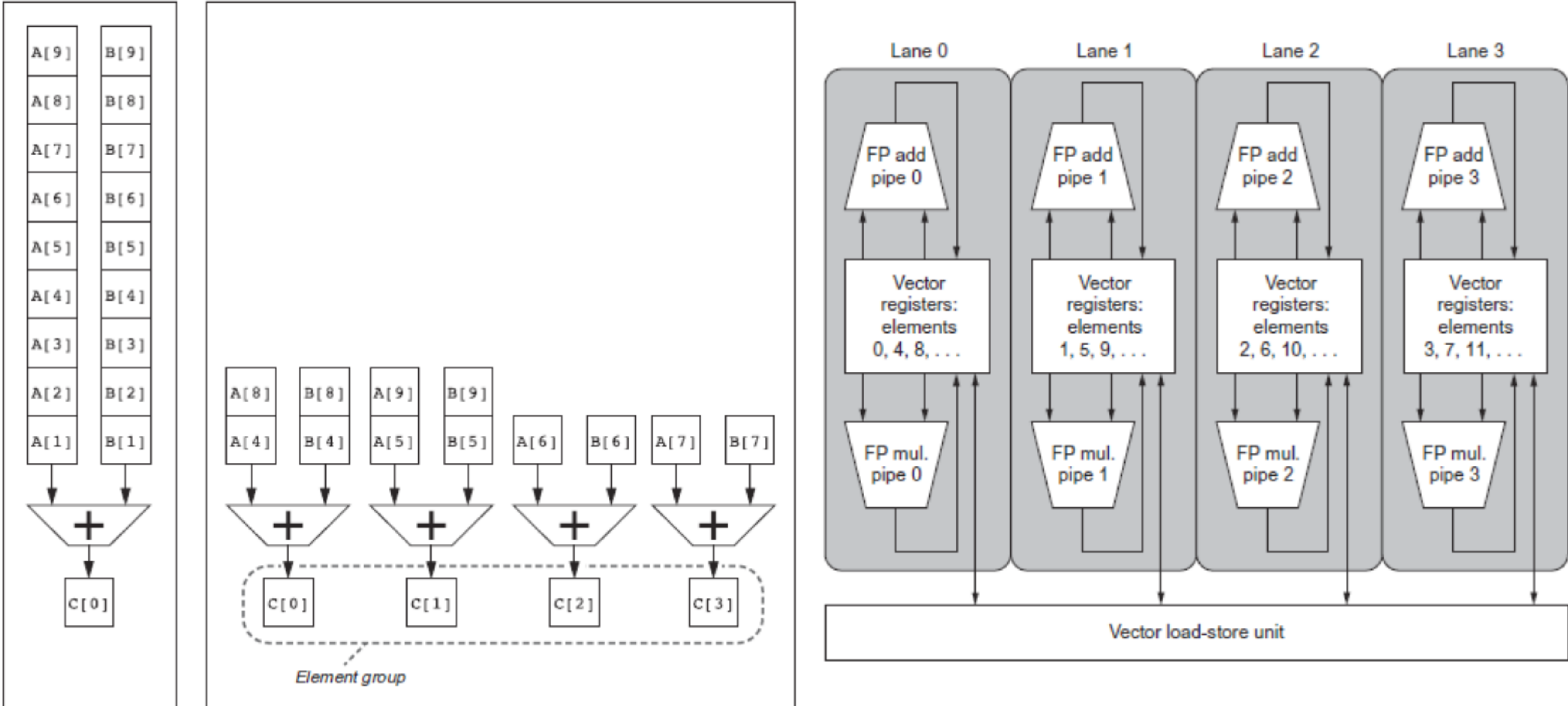
Challenges

- 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. > 1 element per clock cycle
  2. Non-32 wide vectors
  3. IF statements in vector code
  4. Memory system optimizations to support vector processors

Copyright © 2019, Elsevier Inc. All rights Reserved

Multiple Lanes

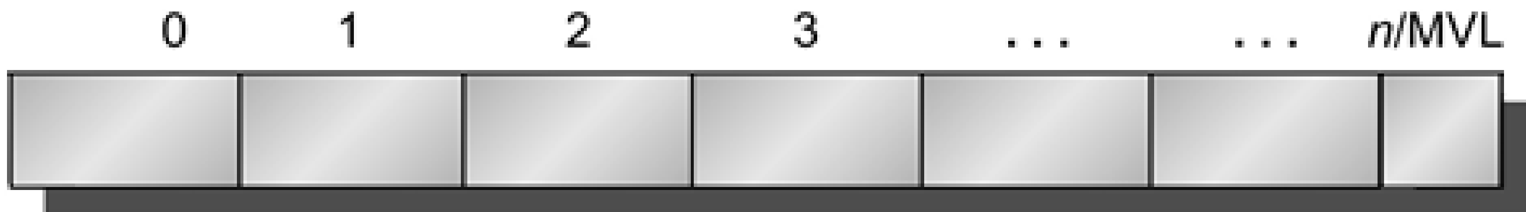
- Beyond one element per clock cycle
- Element *i* of vector register *A* is "hardwired" to element *i* of vector register *B*
  - Allows for multiple hardware lanes



Copyright © 2019, Elsevier Inc. All rights Reserved

Vector-Length Registers

- RV64V has two hardware registers:
  - MVL: maximum vector-length register
  - VL: vector-length register
- Vectors of arbitrary length are processed with ***strip mining***. All blocks but the last are of length MVL, utilizing the full power of the vector processor. The last block contains the reminder elements.



MK MORGAN KAUFMANN
Copyright © 2019, Elsevier Inc. All rights Reserved

Vector-Length Registers

Handling loops not equal to 32.

```
     vsetdcfg 2 DP FP    # Enable 2 64b FP registers
     fld    f0,a         # Load scalar a
  loop:
     setvl  t0,a0        # vl = t0 = min(mvl,n)
     vld    v0,x5        # Load vector X
     slli   t1,t0,3      # t1 = vl * 8 (in bytes)
     add    x5,x5,t1     # Increment pointer to X by vl*8
     vmul   v0,v0,f0     # Vector-scalar mult
     vld    v1,x6        # Load vector Y
     vadd   v1,v0,v1     # Vector-vector add
     sub    a0,a0,t0     # n -= vl (t0)
     vst    v1,x6        # Store the sum into Y
     add    x6,x6,t1     # Increment pointer to Y by vl*8
     bnez   a0,loop      # Repeat if n != 0
     vdisable            # Disable vector regs
```

MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Predicate Registers

- Consider:

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

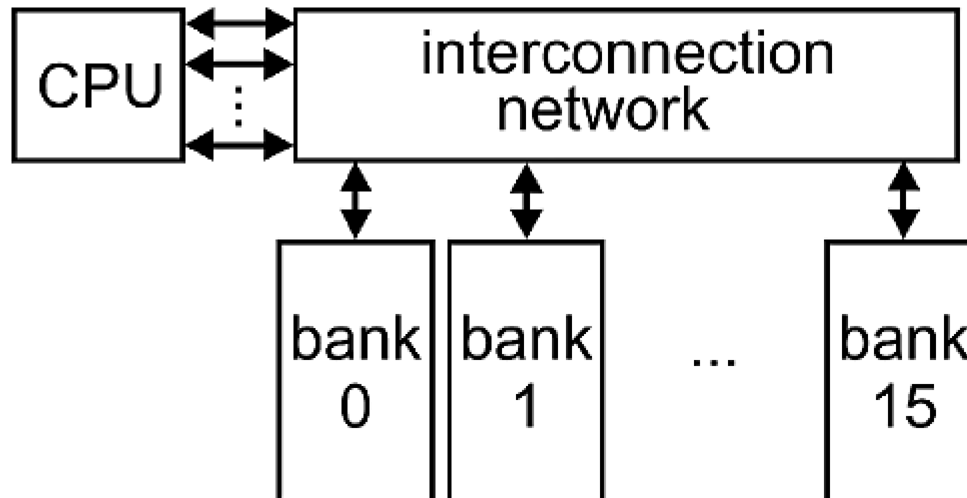
- Use predicate register to “disable” elements:

```
vsetdcfg  2*FP64     # Enable 2 64b FP vector regs
vsetpcfgi 1          # Enable 1 predicate register
vld       v0,x5      # Load vector X into v0
vld       v1,x6      # Load vector Y into v1
fmv.d.x f0,x0        # Put (FP) zero into f0
vpne      p0,v0,f0   # Set p0(i) to 1 if v0(i)!=f0
vsub      v0,v0,v1   # Subtract under vector mask
vst       v0,x5      # Store the result in X
vdisable             # Disable vector registers
vpdisable            # Disable predicate registers
```


Copyright © 2019, Elsevier Inc. All rights Reserved

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 (need independent bank addressing)
  - Support multiple vector processors sharing the same memory



MORGAN KAUFMANN
Copyright © 2019, Elsevier Inc. All rights Reserved

Memory Banks

- Example: Cray T932
  - 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?

SRAM cycle time = 15 / 2.167 = 7 cycles

Access rate = 32 x (4+2) = 192 access/cycle

192 x 7 = 1334 banks

MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions for Multimedia
- Graphics Processing Units
- Fallacies and Pitfalls

Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

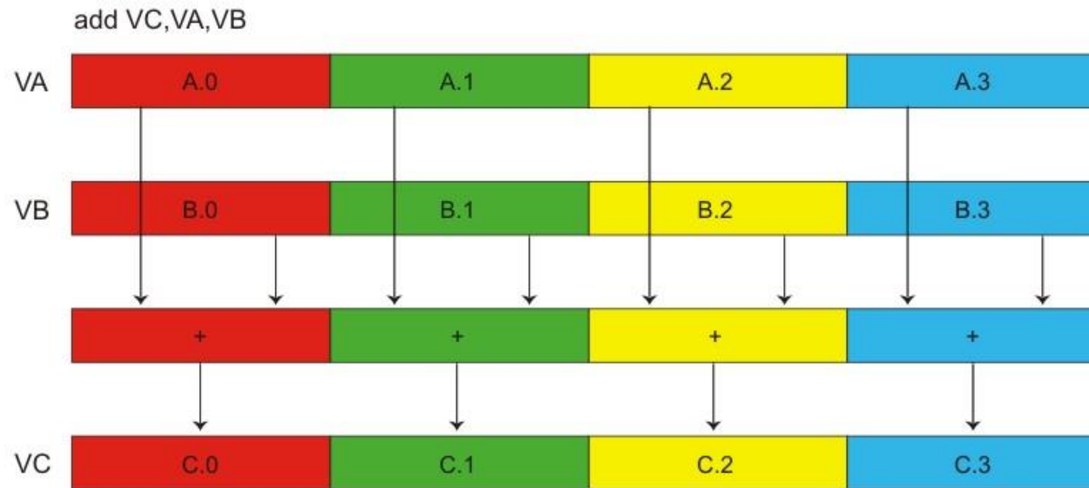
- SIMD Instruction Set Extensions for Multimedia
  - Introduction
  - Intel x86 SIMD Implementations
  - DAXPY SIMD Example
  - Roofline Performance Model

MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Introduction

- Media applications operate on data types narrower than the native word size



- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No predicate registers
  - No sophisticated addressing (strided, scatter-gather)

MORGAN KAUFMANN
Copyright © 2019, Elsevier Inc. All rights Reserved

Intel x86 SIMD 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
- AVX-512 (2017)
  - Eight 64-bit integer/fp ops
- Operands must be consecutive and aligned memory locations

Copyright © 2019, Elsevier Inc. All rights Reserved

DAXPY SIMD Example

```
    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+3]
    fmul.4D  f1,f1,f0     # a x X[i] ... a x X[i+3]
    fld.4D   f2,0(x6)     # Load Y[i] ... Y[i+3]
    fadd.4D  f2,f2,f1     # a x X[i]+Y[i] ...
                          # a x X[i+3]+Y[i+3]
    fsd.4D   f2,0(x6)     # Store Y[i]... 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
```

MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Roofline Performance Model

- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte accessed

Attainable GPLOPs/sec = Min (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Copyright © 2019, Elsevier Inc. All rights Reserved

Example: Vector vs SIMD Ext.

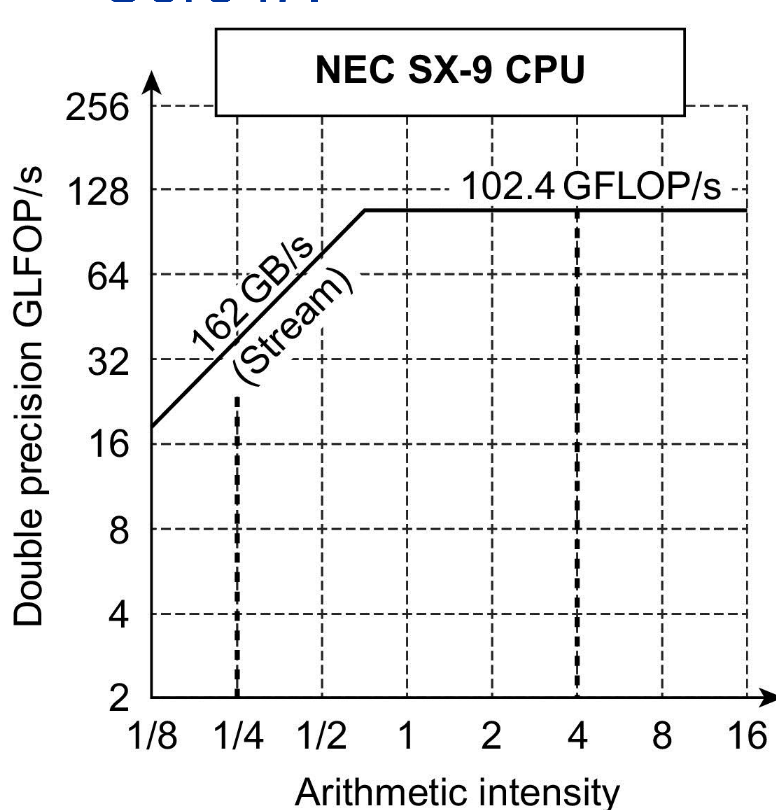
- **NEC SX-9** is a vector supercomputer announced in 2008 that cost millions of dollars. It has a peak DP FP performance of 102.4 GFLOP/s and a peak memory bandwidth of 162 GB/s from the Stream benchmark.
  - 3.2 GHz, eight vector pipes, each having two multiply and two addition units; peak vector performance of 102.4 GFLOP/s
- **Core i7 920** has a peak DP FP performance of 42.66 GFLOP/s and a peak memory bandwidth of 16.4 GB/s.
  - Peak Performance = 2.66 GHz * 4 (cores/chip) * 2 (ops/SIMD instr.) * 2 (FP add-mul/op) = 42.66 GFLOP/s

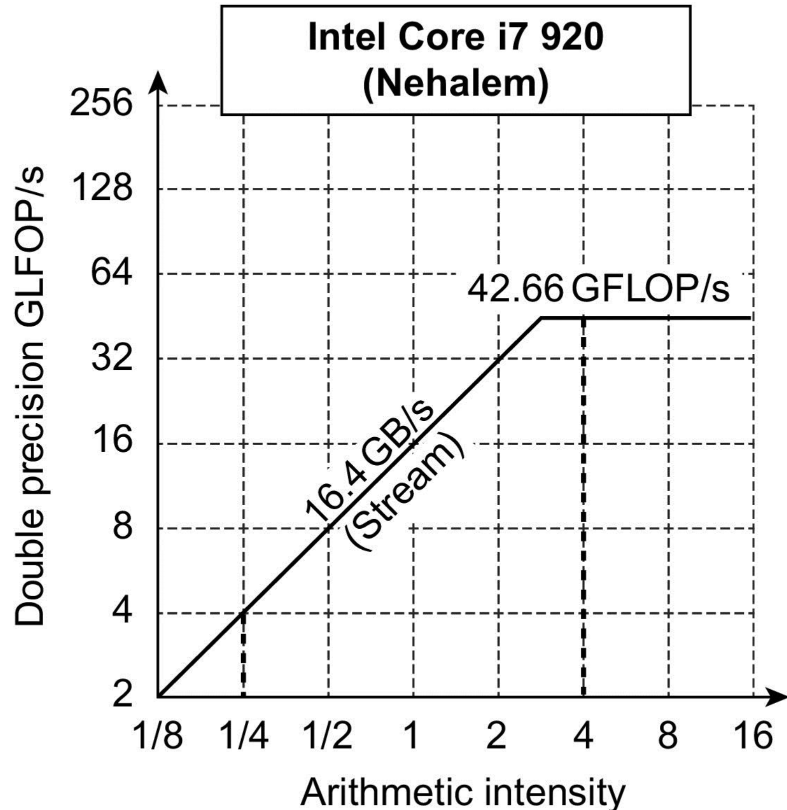
MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Example: Vector vs SIMD Ext.

- **Arith. Int. = 4 FLOP/byte:** both processors operate at peak performance. SX-9 is 2.4 × faster than Core i7.
- **Arith. Int. = 0.25 FLOP/byte**: SX-9 is 10 × faster than Core i7.





Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions for Multimedia
- Graphics Processing Units
- Fallacies and Pitfalls

Copyright © 2019, Elsevier Inc. All rights Reserved

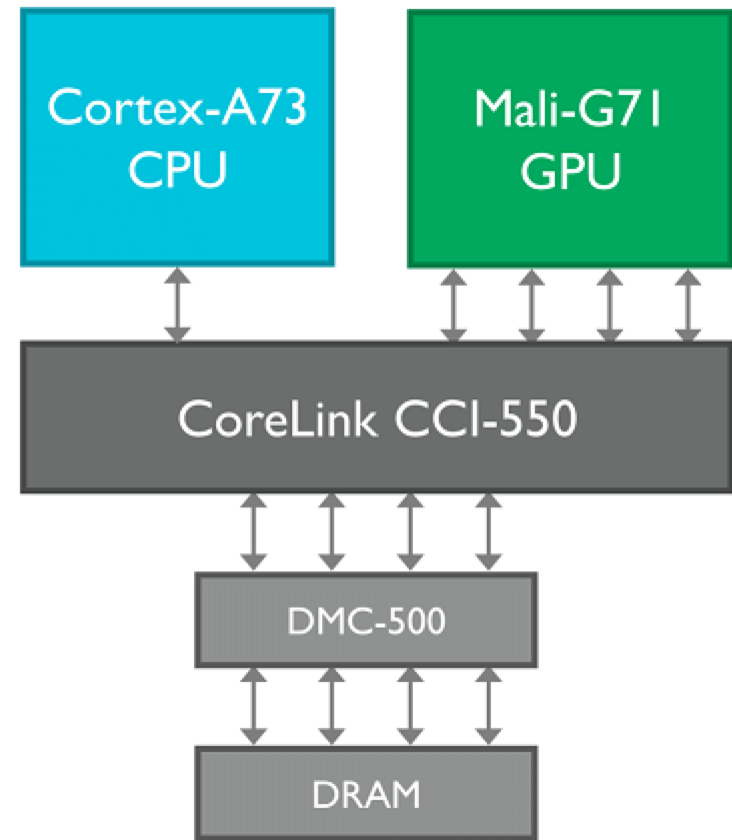
Contents

- Graphics Processing Units
  - Introduction
  - CUDA GPU Programming
  - NVIDIA GPU Architecture
  - NVIDIA GPU ISA
  - NVIDIA GPU Memory Structures
  - NVIDIA GPU Memory Structures

Copyright © 2019, Elsevier Inc. All rights Reserved

Introduction

- Heterogeneous execution model
  - CPU is the *host*, GPU is the *device*
- Develop a C-like programming language for GPU
- CUDA: Compute Unified Device Architecture
- Unify all forms of GPU parallelism as *CUDA thread*
- Programming model is "Single Instruction Multiple Thread"



MK MORGAN KAUFMANN
Copyright © 2019, Elsevier Inc. All rights Reserved

CUDA GPU Programming

- A thread is associated with each data element. *Single instruction, multiple threads.*
- C DAXPY function:

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

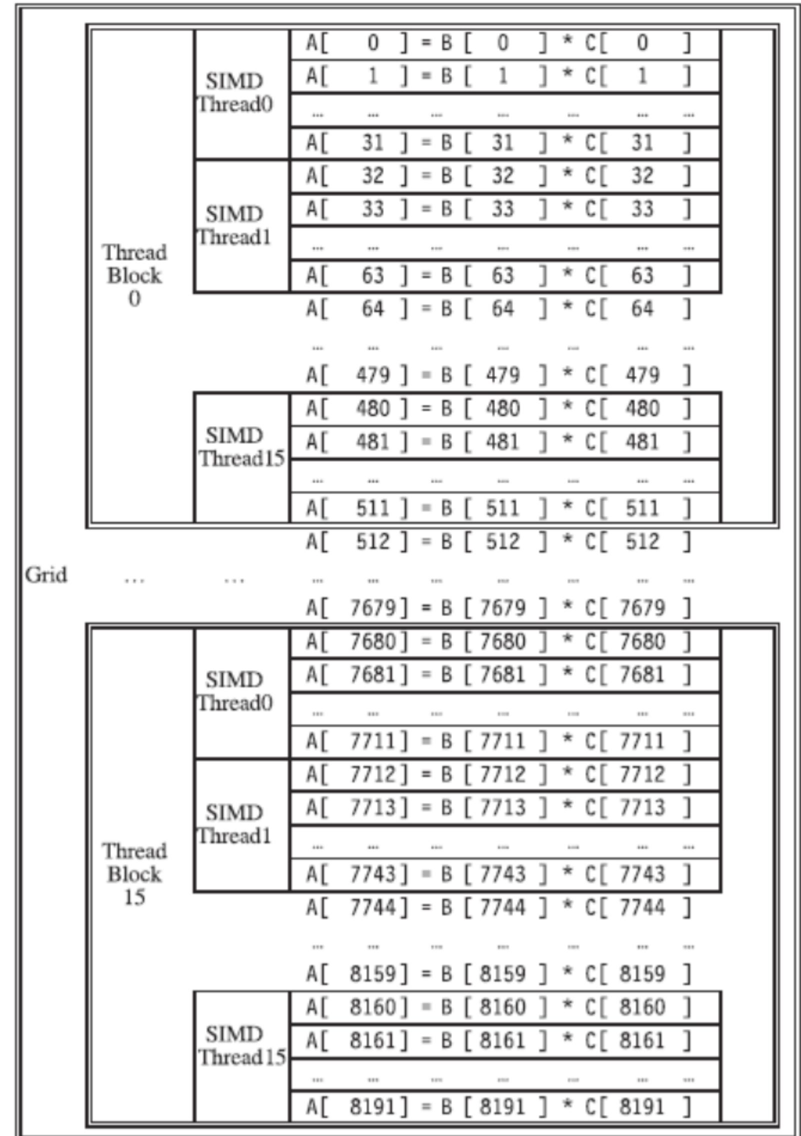
- CUDA DAXPY function:

```
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];
}
```


Copyright © 2019, Elsevier Inc. All rights Reserved

CUDA GPU Programming

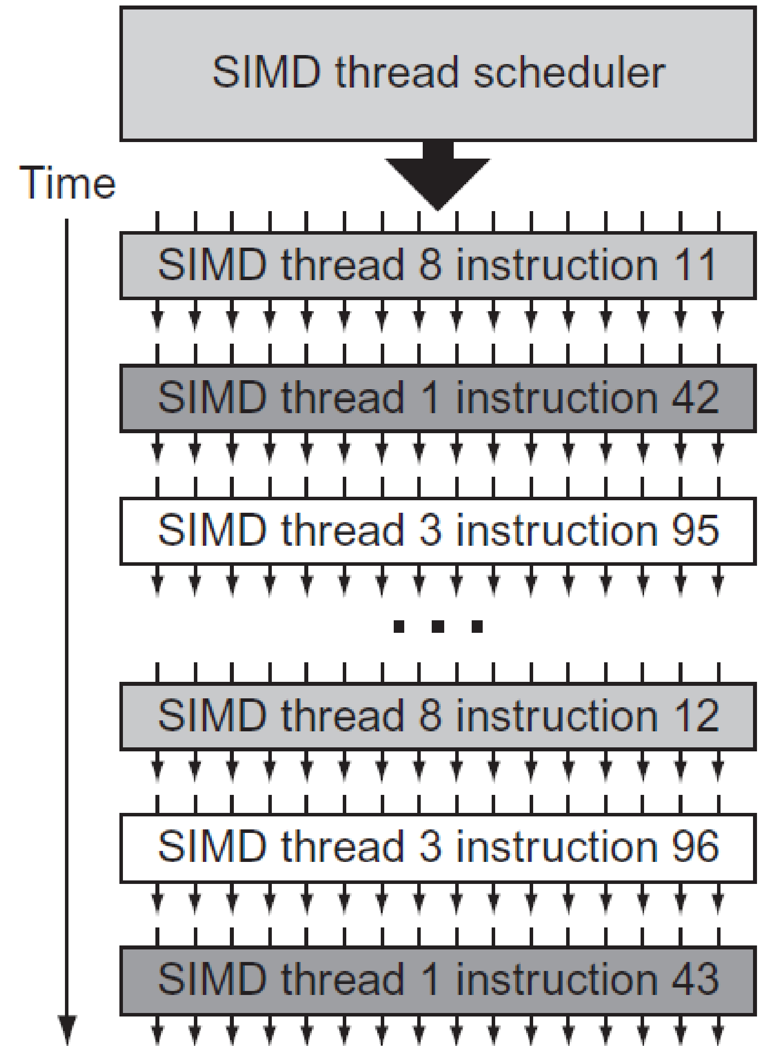
- Code that works over all elements is the grid.
- Grids are broken into thread blocks of manageable sizes, e.g., 512 threads per block.
- SIMD instruction executes 32 elements at a time (warp).
- Block = 512/32 = 16 warps.
- GPU hardware handles thread management, not applications or OS



Copyright © 2019, Elsevier Inc. All rights Reserved

NVIDIA GPU Architecture

- Current-generation GPUs have tens of multithreaded SIMD processors.
- Nvidia calls the multithreaded SIMD processor Streaming multiprocessor (SM).
- Blocks are assigned to a SMs by the Thread Block Scheduler.
- Each SM has SIMD Thread Scheduler (Warp scheduler) that schedules threads of SIMD instructions.



MK MORGAN KAUFMANN
Copyright © 2019, Elsevier Inc. All rights Reserved

NVIDIA GPU Architecture

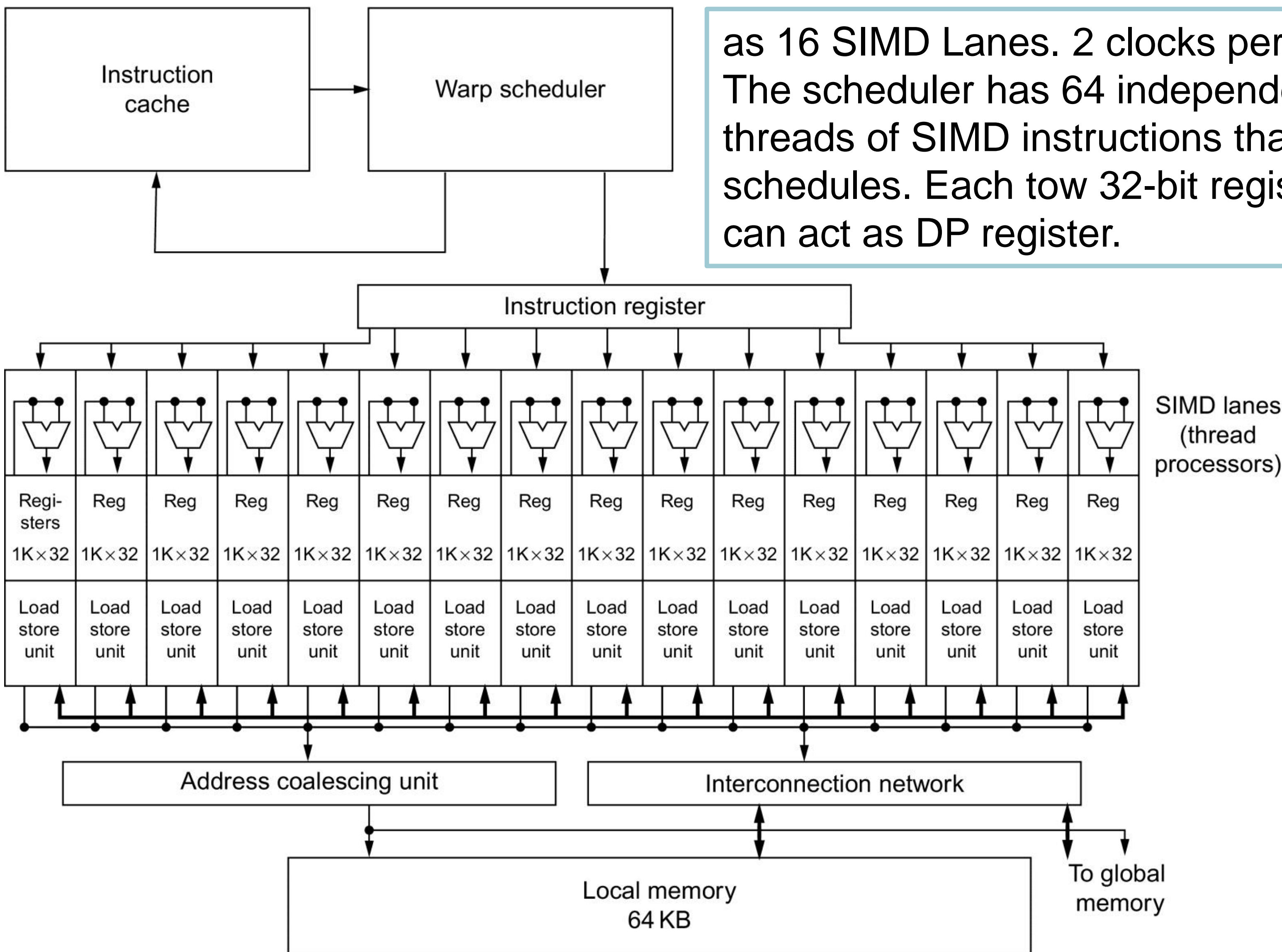
- **Pascal P100 GPU** has 60 SMs, four spares.

MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

Simplified SM Processor

as 16 SIMD Lanes. 2 clocks per instr. The scheduler has 64 independent threads of SIMD instructions that it schedules. Each tow 32-bit registers can act as DP register.



MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

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

Copyright © 2019, Elsevier Inc. All rights Reserved

Terminology

- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or “warp”
  - Mapped to 16 physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
  - Each warp has its own PC
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide (16 lanes) and shallow (2 ops per lane) compared to vector processors
- See figure 4.14

Copyright © 2019, Elsevier Inc. All rights Reserved

NVIDIA GPU ISA

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Format of PTX instruction: `opcode.type d,a,b,c;`

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	.f16, .f32, .f64

MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved

NVIDIA GPU ISA

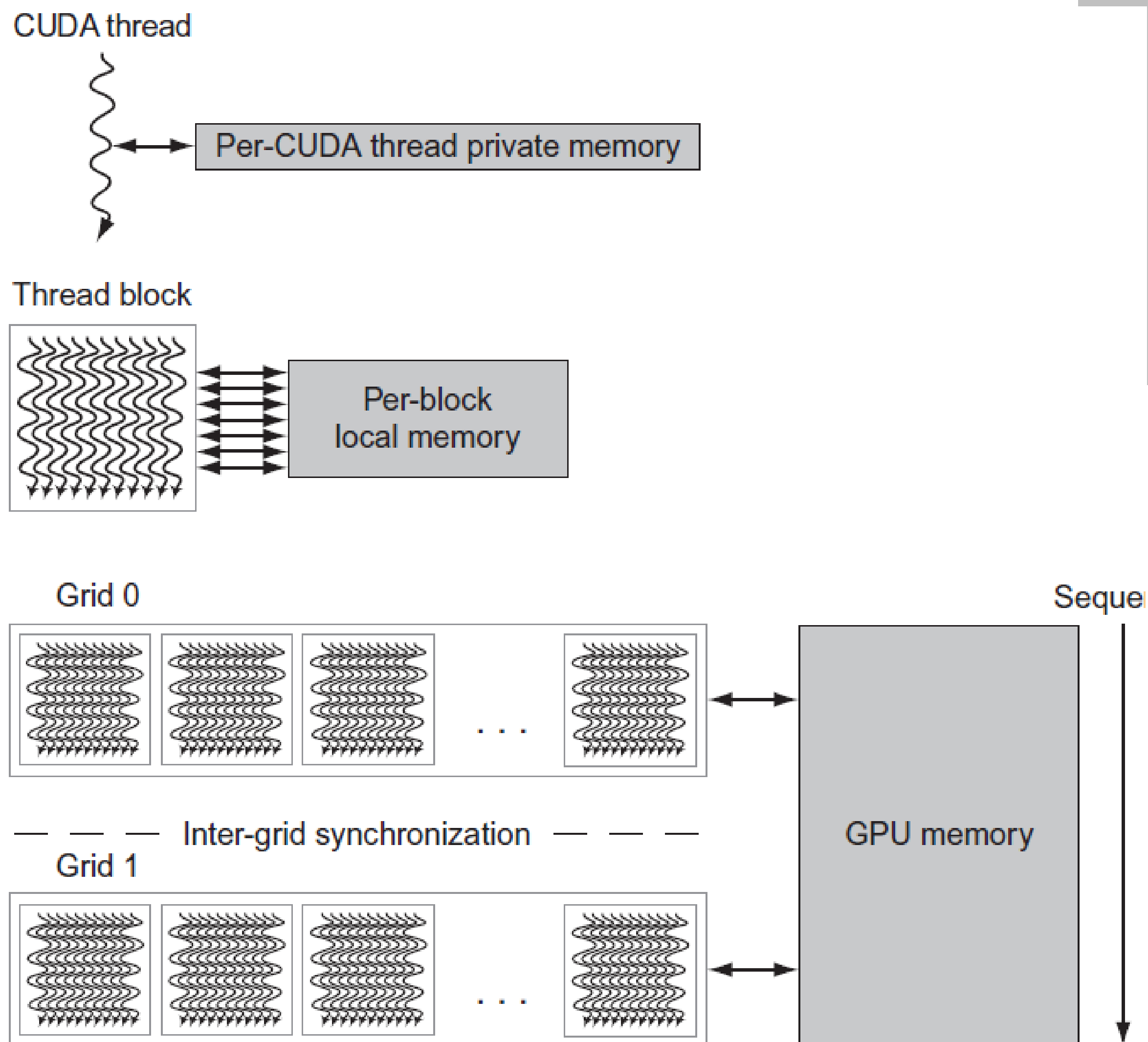
- DAXPY Example:

```
shl.s32         R8, blockIdx, 9   ; Block ID * Block size (512 or 2^9)
add.s32         R8, R8, threadIdx ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]       ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]       ; RD2 = Y[i]
mul.f64         RD0, RD0, RD4     ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64         RD0, RD0, RD2     ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64   [Y+R8], RD0       ; Y[i] = sum (X[i]*a + Y[i])
```


Copyright © 2019, Elsevier Inc. All rights Reserved

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - 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



Copyright © 2019, Elsevier Inc. All rights Reserved

Contents

- Introduction
- Vector Architecture
- SIMD Instruction Set Extensions for Multimedia
- Graphics Processing Units
- Fallacies and Pitfalls

Copyright © 2019, Elsevier Inc. All rights Reserved

Fallacies and Pitfalls

- F: GPUs suffer from being coprocessors
  - GPUs have flexibility to change ISA
- P: Concentrating on peak performance in vector architectures and ignoring start-up overhead
  - Overheads require long vector lengths to achieve speedup
- P: Increasing vector performance without comparable increases in scalar performance
- F: You can get good vector performance without providing memory bandwidth
- F: On GPUs, just add more threads if you don't have enough memory performance
  - This works only if threads have good memory locality.

MK MORGAN KAUFMANN

Copyright © 2019, Elsevier Inc. All rights Reserved