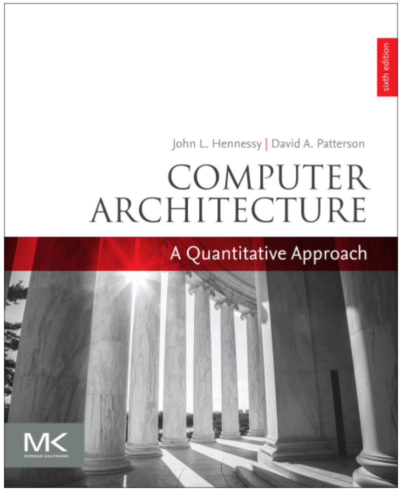


Chapter 4

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

Introduction

- SIMD 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

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

- 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

- 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

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
- 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?
    - 32x(4+2)x15/2.167 = ~1330 banks

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:

```
vsetdcfg    4*FP64        # 4 64b FP vector registers
vld         v0, x7        # Load K[]
vldx        v1, x5, v0    # Load A[K[]]
vld         v2, x28       # Load M[]
vldi        v3, x6, v2    # Load C[M[]]
vadd        v1, v1, v3    # Add them
vstx        v1, x5, v0    # Store A[K[]]
vdisable                  # Disable vector registers
```

Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

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

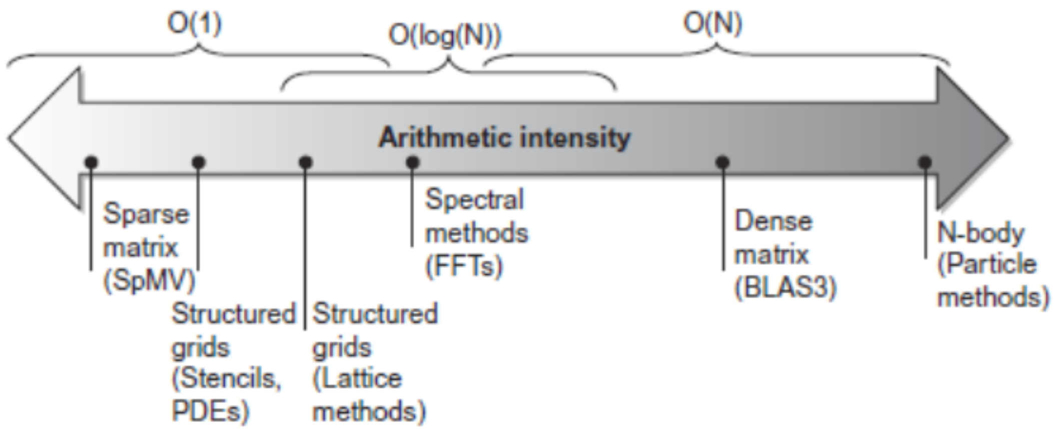
Example SIMD Code

- Example DXPY:

```
      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
```

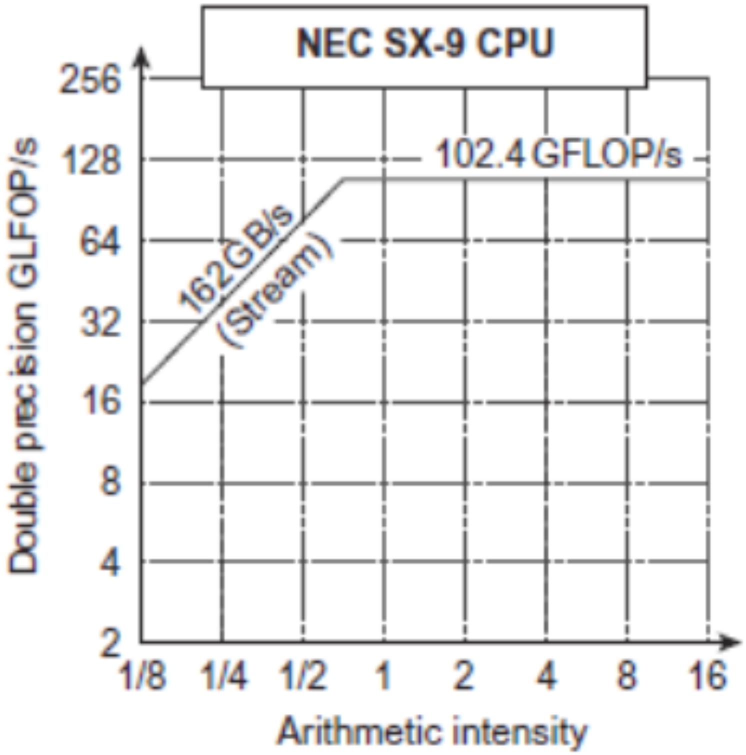
Roofline Performance Model

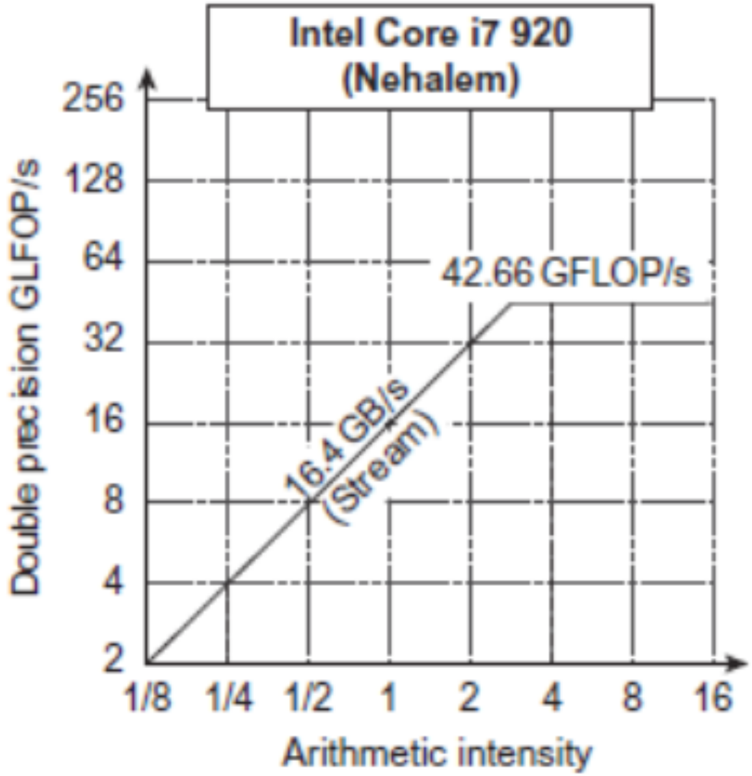
- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read



Examples

- Attainable GFLOPs/sec = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)





Graphical Processing Units

- 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”

Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- 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

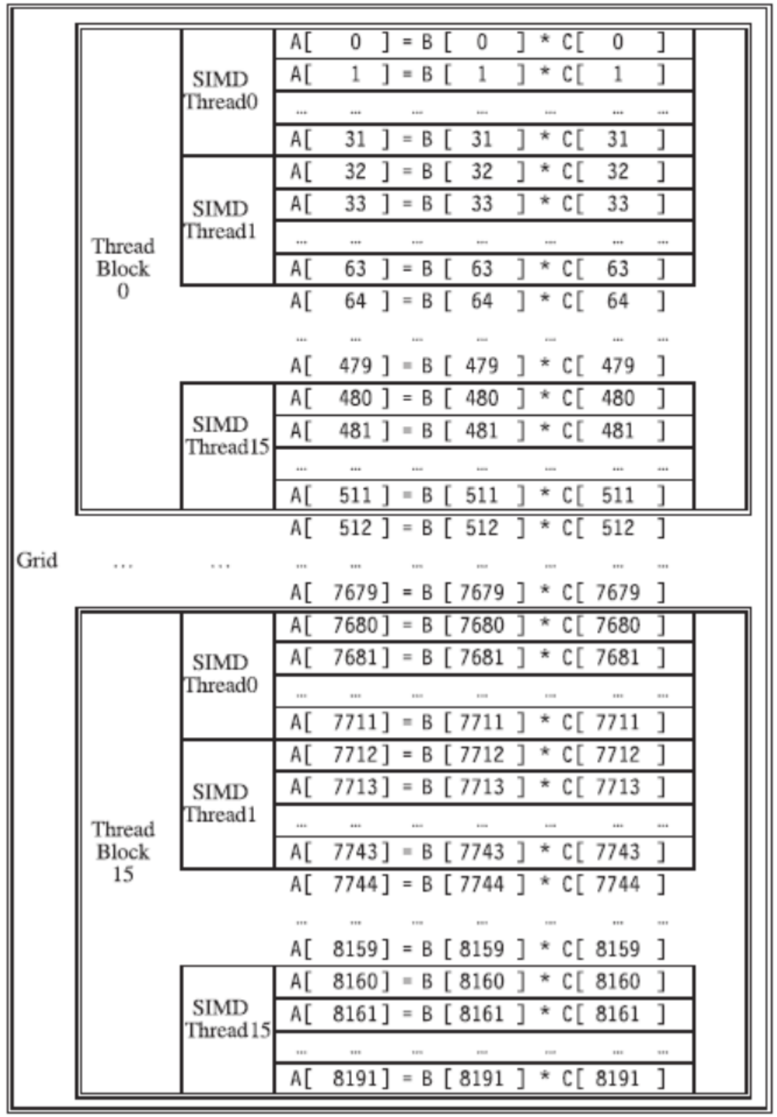
Example

- 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 have 7-15 multithreaded SIMD processors

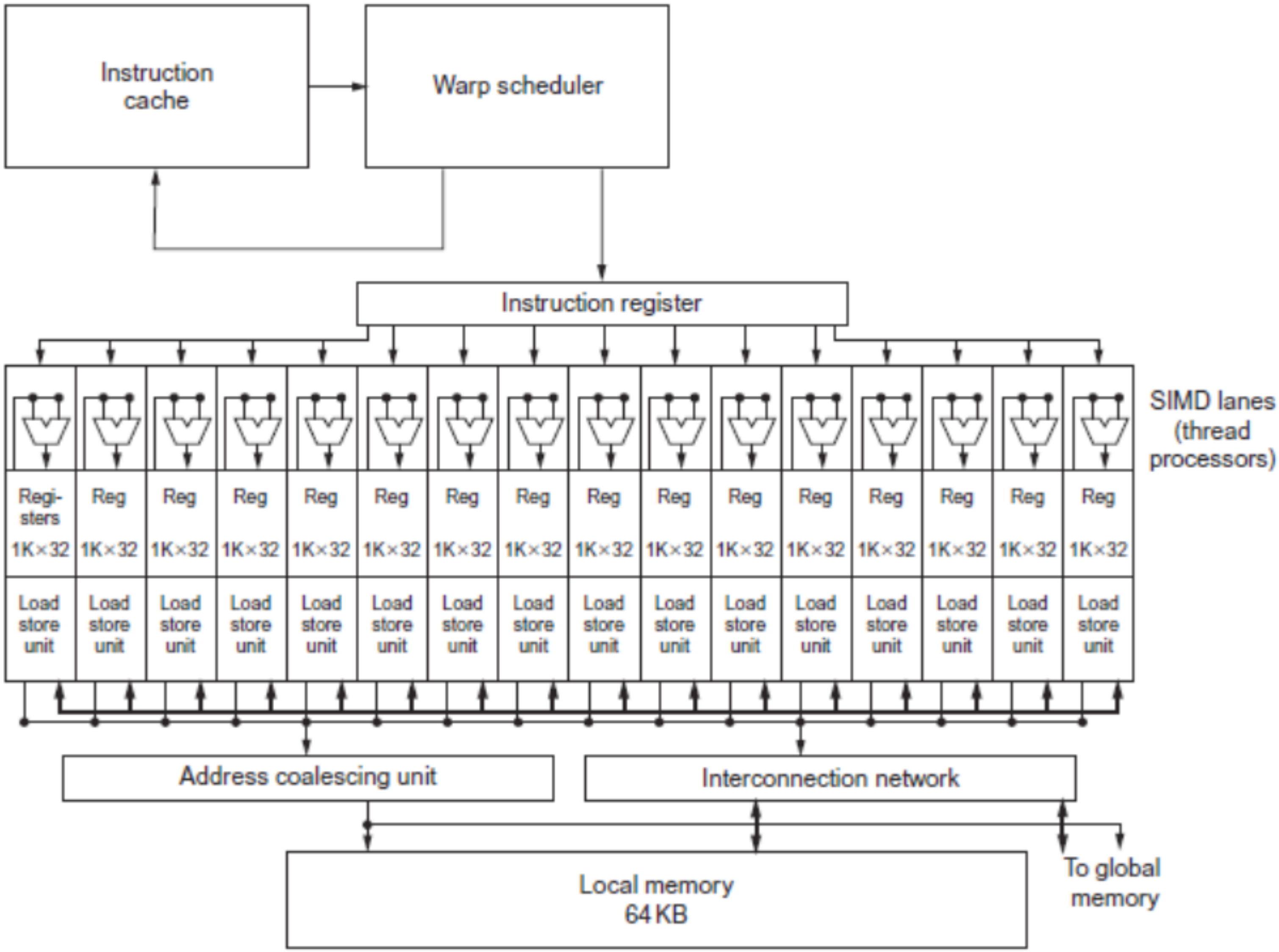
Terminology

- 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 and shallow compared to vector processors

Example


Grid
Thread Block 0
SIMD Thread0
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
SIMD Thread1
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
SIMD Thread15
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
Thread Block 15
SIMD Thread0
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
SIMD Thread1
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
SIMD Thread15
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

GPU Organization



NVIDIA Instruction Set Arch.

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

```
shl.s32         R8, blockIdx, 9        ; Thread Block ID * Block size (512 or 29)
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 R0D, RD0, RD4  ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2  ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0              ; Y[i] = sum (X[i]*a + Y[i])
```

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

Example

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];

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, go to 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, go to 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
```

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

Pascal Architecture Innovations

- Each SIMD processor has
  - Two or four SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Two threads of SIMD instructions are scheduled every two clock cycles
- Fast single-, double-, and half-precision
- High Bandwith Memory 2 (HBM2) at 732 GB/s
- NVLink between multiple GPUs (20 GB/s in each direction)
- Unified virtual memory and paging support

Pascal Multithreaded SIMD Proc.

Vector Architectures vs GPUs

- SIMD processor analogous to vector processor, both have MIMD
- Registers
  - RV64V register file holds entire vectors
  - GPU distributes vectors across the registers of SIMD lanes
  - RV64 has 32 vector registers of 32 elements (1024)
  - GPU has 256 registers with 32 elements each (8K)
  - RV64 has 2 to 8 lanes with vector length of 32, chime is 4 to 16 cycles
  - SIMD processor chime is 2 to 4 cycles
  - GPU vectorized loop is grid
  - All GPU loads are gather instructions and all GPU stores are scatter instructions

SIMD Architectures vs GPUs

- GPUs have more SIMD lanes
- GPUs have hardware support for more threads
- Both have 2:1 ratio between double- and single-precision performance
- Both have 64-bit addresses, but GPUs have smaller memory
- SIMD architectures have no scatter-gather support

Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i]; /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration

Loop-Level Parallelism

- Example 3:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

- Example 4:

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

- Example 5:

```
for (i=1;i<100;i=i+1) {
    Y[i] = Y[i-1] + Y[i];
}
```

Finding dependencies

- Assume indices are affine:
  - $a \times i + b$ (i is loop index)
- Assume:
  - Store to $a \times i + b$, then
  - Load from $c \times i + d$
  - i runs from m to n
  - Dependence exists if:
    - Given j, k such that $m \le j \le n$, $m \le k \le n$
    - Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

Finding dependencies

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - GCD test:
    - If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)
- Example:

```
for (i=0; i<100; i=i+1) {
    X[2*i+3] = X[2*i] * 5.0;
}
```

Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies

Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies

Reductions

- Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- Do on p processors:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note: assumes associativity!

Fallacies and Pitfalls

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