### Concepts Introduced in Chapter 4

- vector architectures
- SIMD ISA extensions
- graphics processing units (GPUs)
- loop dependence analysis

### SIMD Advantages

- SIMD architectures can significantly improve performance by exploiting DLP when available in applications.
- SIMD processors are more energy efficient than MIMD as they only need to fetch a single instruction to perform the same operation on multiple data items.
- SIMD allows programmers to continue to think sequentially and sometimes SIMD parallelism can be automatically exploited.

### Vector Architectures

- A vector architecture includes instruction set extensions to an ISA to support vector operations, which are deeply pipelined.
  - Vector operations are on vector registers, which are fixed-length bank of registers.
  - Data is transferred between a vector register and the memory system.
  - Each vector operation takes vector registers or a vector register and a scalar value as input.
- A vector architecture can only be effective on applications that have significant data-level parallelism (DLP).
- vector processing advantages
  - Greatly reduces the dynamic instruction bandwidth.
  - Generally execution time is reduced due to (1) eliminating loop overhead, (2) stalls only occurring on the first vector element rather than on each vector element, and (3) performing vector operations in parallel.

### Extending the MIPS to Support Vector Operations

- Add vector registers where each register has 64 elements with each element 64 bits wide.
- After an initial latency each vector functional unit can start a new operation on each clock cycle.
- Vector loads and stores also pay for the memory latency once and afterwards a word is transferred each cycle between the vector register and memory.
- The processor has to detect both structural and data hazards.
Example of Vector Code

```c
/* Scalar MIPS Code */
L.D F0,a <= for (i = 0; i < 64; i++)
DADDIU Rx,Rx,#512 Y[i] = a * X[i] + Y[i];

Loop:
L.D F2,0(Rx)
MUL.D F2,F2,F0 /* VMIPS Code */
L.D F4,0(Ry) L.D F0,a
ADD.D F4,F4,F2 LV V1,Rx
S.D F4,0(Ry) => MULVS.D V2,V1,F0
DADDIU Rx,Rx,#8 LV V3,Ry
DADDIU Ry,Ry,#8 ADDVV.D V4,V2,V3
DSUBU R20,R4,Rx SV V4,Ry
BNEZ R20,Loop
```

Chaining, Convoys, and Chimes

- **Chaining** allows the results of one vector operation to be directly used as input to another vector operation.
- A **convoys** is a set of vector instructions that can potentially execute together. Only structural hazards cause separate convoys as true dependencies are handled via chaining in the same convoy.
- A **chime** is the unit of time taken to execute one convoy, which is the vector length along with the startup cost.
- The following VMIPS code executes in three chimes since there are three convoys.

```c
/* VMIPS code */ /* convoys */
LV V1,Rx 1. LV V1,Rx
MULVS.D V2,V1,F0 MULVS.D V2,V1,F0
LV V3,Ry 2. LV V3,Ry
ADDVV.D V4,V2,V3 ADDVV.D V4,V2,V3
SV V4,Ry 3. SV V4,Ry
```

Startup Time

- The startup time for a convoy is primarily affected by the pipelining latency of the vector functional unit associated with the vector operation.
- Pipeline latencies in clock cycles for the VMIPS
  - FP add - 6
  - FP multiply - 7
  - FP divide - 20
  - load - 12
- Additional cycles need to be added for stalls between the chained vector operations, but only for the first element of each vector operation.
- The cycles to execute the following convoy should be the sum of the *startup time* and the *vector length*, or 83 (19+64).

```c
LV V1,Rx
MULVS.D V2,V1,F0
```

Using Multiple Lanes

- Vector operations on vector register elements can also be executed in parallel when there is an array of parallel pipelined functional units.
- So element N of a vector register will take part in operations with element N from other vector registers.
- Each portion of the vector operations that are performed in parallel are called a **lane**.
- Each lane i of n lanes operates on each k vector register file element where k % n is equal to i.
- No communication is needed between lanes.
- Convoy time is now *startup time + vector length/n*.
Vector Length Register

- A vector length register (VLR) allows the length of a vector operation to be determined at runtime.
- Loops are *strip mined* so that the maximum vector length (MVL) is no more than the length of a vector register.
- The innermost loop in the strip mined loop nest can be vectorized.

```c
/* original loop */
for (i = 0; i < n; i++)
    Y[i] = a * X[i] + Y[i];

/* strip mined loop nest */
for (j = 0; j <= n/MVL; j++) {
    for (i = low; i < low+VL; i++)
        Y[i] = a * X[i] + Y[i];
    low += VL;
    VL = MVL;
}
```

Vector Mask Registers

- Mask registers provide conditional execution of each element operation in a vector instruction.
- When the vector-mask register is enabled, vector instructions update results only for vector elements where the corresponding bit in the vector-mask register is set.
- No execution time is saved for the elements where the bits in the vector-mask register are zero.

```c
/* original loop */
for (i = 0; i < 64; i++)
    if (X[i] != 0)
        X[i] -= Y[i];

/* VMIPS assembly code */
for (i = 0; i < 64; i++)
    if (X[i] != 0)
        SNEVS.D V1,F0
    SUBVV.D V1,V1,V2

/* VMIPS code */
for (i = 0; i < 64; i++)
    if (X[i] != 0)
        MULVS.D V2,V1,F0
```

Using Cache/Memory Banks

- Recent vector computers use caches to reduce the latency of vector loads and stores.
- Word-interleaved banks for cache and main memory provide the ability for simultaneous independent accesses.
- Supporting multiple vector load or store operations to avoid a structural hazard.
- Supporting vector loads or stores that are not sequential.
- Supporting multiple processor cores sharing the same L3 cache and main memory.
Handling Non-Unit Strides

- The distance separating elements in memory can be nonsequential, which is called a non-unit stride.
- The vector stride can be put in a general-purpose register and can be accessed with vector load/store instructions.
- Supporting non-unit strides may cause more bank contention and complicates the vector load/store operations.

```c
/* matrix multiply loop nest */
for (i = 0; i < 64; i++)
    for (j = 0; j < 64; j++)
        for (k = 0; k < 64; k++)
            T += B[i][k]*C[k][j];
    A[i][j] = T;
```

Gather-Scatter Operations

- Sparse matrices are common and are usually stored in some compacted form and indirectly accessed.
- An index vector contains the indices of array elements to be accessed.
- A gather/scatter operation uses the index vector along with a base address to fetch/store elements in an array.

```c
/* sparse array loop */
for (i = 0; i < n; i++)
    A[K[i]] += B[M[i]]; 
```

SIMD Extensions to GP Processors

- Many GP processors now have SIMD extensions to support simultaneous operations on applications, including for multimedia.
- SIMD extensions are a subset of vector operations.
  - Operate on a fixed number of operands (no VLR register).
  - Do not support non-unit strides or gather-scatter access.
  - Do not support conditional execution of operations (no vector mask register).
- SIMD operations work on shorter vectors and all operations are typically performed in parallel, as opposed to being pipelined.
- Examples include the x86 SIMD extensions.
  - MultiMedia eXtensions (MMX) in 1996 - used FP registers
  - Streaming Simd Extensions (SSE) 1999 - separate 128-bit registers
  - Advanced Vector eXtensions (AVX) 2010 - separate 256-bit registers

SIMD Extensions to GP Processors

- Can be added with little cost. For instance, an option to a conventional integer adder can be to not perform carries across specific partitions (e.g. parallel 8-bit additions).
- Require little state as compared to vector architectures, which means it is easier to implement context switches.
- Need much less memory bandwidth.
- Operands in memory for SIMD extensions on many architectures have to be aligned within a L1 DC line, which means one instruction only needs one access to the memory system. However, due to this SIMD alignment problem, it is much harder for compilers to automatically exploit these SIMD extensions.
**SIMD Example**

- X and Y have to be aligned on a 32 byte boundary.

```c
/* sparse array loop */ /* MIPS SIMD code */
for (i = 0; i < 64; i++) L.D F0,a
    Y[i] = a * X[i] + Y[i];
MOV F1,F0
MOV F2,F0
MOV F3,F0
DADDIU R4,Rx,#512
Loop:
    L.4D F4,0(Rx)
    MUL.4D F4,F4,F0
    L.4D F8,0(Ry)
    ADD.4D F8,F8,F4
    S.4D F8,0(Rx)
    DADDIU Rx,Rx,#32
    DADDIU Ry,Ry,#32
    BNE Rx,R4,Loop
```

**Graphics Processing Units (GPUs)**

- GPUs were first developed as graphics accelerators, but now are also starting to be used in mainstream computing (GP GPUs).
- GPUs support many types of parallelism (ILP, SIMD, multithreading, MIMD), but work best with DLP applications.
- Some GPUs have their own programming language.
  - CUDA is offered by NVIDIA.
  - OpenCL is vendor-independent for multiple platforms.

**NVIDIA GPU Overview**

- heterogeneous execution model
  - CPU is the host.
  - GPU is the device.
- CUDA is a C-like programming language to exploit GPU features.
- The programming model is called single instruction, multiple thread (SIMT).

**NVIDIA Terminology**

- programming abstractions
  - A vectorizable loop is called a *grid*.
  - A *grid* is composed of *thread blocks*, which is equivalent to the body of a strip-mined loop.
  - A *thread block* consists of a set of *CUDA threads*.
  - Each *CUDA thread* processes one element of the vector registers and is equivalent to one iteration of a scalar loop.
- machine object
  - A *warp* is a thread of *PTX* instructions.
  - A *PTX* (Parallel Thread eXecution) instruction is a SIMD instruction.
- processing hardware
  - A *SIMD lane* executes the operations in a *CUDA thread* of SIMD instructions.
  - Multiple *SIMD lanes* within a *thread block* all simultaneously execute the same instruction or are idle.
CUDA Source Example

```c
// Invoke DAXPY in C.
daxpy(n, 2.0, x, y);

// 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];
}

=> // Invoke DAXPY in CUDA with 256 CUDA threads per thread block.
__host__
int nblocks = (n + 255)/256;
daxpy<<<nblocks, 256>>>(n, 2.0, 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];
}
```

Scheduling GPU Instructions

- GPU hardware handles the thread management, not the OS or the application.
- A **thread block** scheduler assigns **thread blocks** to **SIMD processors**.
- A **SIMD thread scheduler** allocates **SIMD threads** within a multithreaded **SIMD processor**.
- **SIMD threads** are used to hide memory latency.

### GPU Threads

- There are often more **SIMD threads** on a **SIMD processor** than can run at one time, which is useful for hiding memory latency.
- Uses a scoreboard to detect **SIMD threads** ready to run.
- Each **SIMD thread** has its own PC and each **SIMD instruction** within a thread simultaneously executes up to **n operations**.
- The **n parallel functional units** to perform a **SIMD operation** are called **lanes**.
- No dependences can exist between different **SIMD threads**.
- A **CUDA thread** (vertical cut of **SIMD instructions** within a **SIMD thread**) is typically assigned for each loop iteration.
- For each **CUDA thread**, virtual registers are assigned to distinct physical registers and a unique identifier number is used to determine the offsets into arrays so the same code can be invoked both within and across different threads.

CUDA PTX Assembly Code Example

```assembly
/* code for one loop iter */ /* CUDA PTX code */
Y[i] = a * X[i] + Y[i];
shl.u32 R8,blockIdx,9
add.u32 R8,R8,threadIdx
shl.u32 R8,R8,3
ld.global.f64 RD0,[X+R8]
ld.global.f64 RD2,[Y+R8]
mult.f64 RD0,RD0,RD4
add.f64 RD0,RD0,RD2
st.global.f64 [Y+R8],RD0
```
PTX Conditional Branching

- Predicate mask registers are used to handle conditional branches as conditionally executed code.
- Also uses a branch synchronization stack for complex control flow.
  - A branch synchronization entry is pushed when a conditional branch is executed and some lanes diverge (IF-THEN portion), which causes mask bits to be set based on the condition.
  - A branch synchronization marker is used to complement the mask bits (ELSE portion).
  - Another branch synchronization marker is used to pop the stack when the paths converge (end of IF).

PTX Conditional Branching Example

Assume R8 already has the appropriate offset and that *Push, *Comp, and *Pop indicate the branch synchronization markers inserted by the assembler.

```c
/* conditional construct */ /* CUDA PTX code */
... ld.global.f64 RD0,[X+R8]
if (X[i] != 0) setp.neq.s32 P1,RD0,#0
X[i] = X[i] - Y[i]; @!P1,bra ELSE1,*Push
else
  X[i] = Z[i];
sub.f64 RD0,RD0,RD2
... st.global.f64 [X+R8],RD0
@P1,bra ENDIF1,*Comp
ELSE1:
  ld.global.f64 RD0,[Z+R8]
  st.global.f64 [X+R8],RD0
ENDIF1:
  <next inst> *Pop
  st.global.f64 [Y+R8],RD0
```

Comparison with Vector Computers

- similarities to vector computers
  - Works well on data-level parallel problems.
  - Supports scatter-gather memory operations.
  - Uses mask registers to support conditional execution.
- differences with vector computers
  - Scalar instructions are not intermixed with GPU instructions.
  - Uses multithreading to hide memory latency.
  - Has many functional units, as opposed to a few deeply pipelined vector functional units.

Loop Dependences

- A loop can be parallelized if its iterations are all independent.
- A loop-carried dependence is when a data item in one loop iteration depends on a value produced in an earlier iteration.
- The loop below has two loop-carried dependences that prevent it from being parallelized.

```c
for (i = 1; i < 100; i++) {
  A[i] = A[i-1] * 2; /* S1 */
  B[i+1] = B[i] + A[i]; /* S2 */
}
```

- The distance in iterations for the loop-carried dependence is called the dependence distance. The following loop has a loop-carried dependence with a dependence distance of 4.

```c
for (i = 4; i < 100; i++)
```
A loop with a loop-carried dependence can be parallelized if the dependences in a loop do not form a cycle.

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

S1 is dependent on S2 from the previous iteration. This loop can be transformed so the only dependences are within a single iteration.

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

A reduction is where a vector is reduced to a single value.

The following loop cannot be parallelized due to the recurrence on the variable `sum`.

```c
for (i = 0; i < 1000; i++)
    sum = sum + x[i]*y[i];
```

Scalar expansion can be used to parallelize the loop at the expense of adding a simpler loop that cannot be parallelized afterwards.

```c
for (i = 0; i < 1000; i++)
    sum[i] = sum[i] + x[i]*y[i];
for (i = 0; i < 1000; i++)
    finalsum = finalsum + sum[i];
```

### Pipelining Reductions

The following loop cannot even be effectively pipelined due to the recurrence on the variable `sum` that results in stalls between iterations.

```c
for (i = 0; i < 1000; i++)
    sum = sum + x[i];
```

Accumulator expansion can be used to minimize these stalls.

```c
for (i = 0; i < 1000; i += 4) {
    sum1 = sum1 + x[i];
    sum2 = sum2 + x[i+1];
    sum3 = sum3 + x[i+2];
    sum4 = sum4 + x[i+3];
}
finalsum = sum1 + sum2 + sum3 + sum4;
```

### Dependence Analysis

Dependence analysis attempts to determine if two references can ever access the same variable. Array-oriented dependence analysis is performed when array references can be represented as affine functions of the form $a*i + b$, where $i$ is typically a loop index variable, $a$ is a constant, and $b$ is a constant.

One simple test is the GCD test, where if we have two elements to the same array indexed by $a*j+b$ and $c*k+d$, then GCD($c,a$) must divide $d-b$ with no remainder.

```c
for (i = 0; i < 100; i++)
    X[2*i+3] = X[4*i];
```

Here, $a=2$, $b=3$, $c=4$, and $d=0$. So GCD($a,c$) = 2, and $d-b = -3$. -3/2 does not produce an integer quotient, so these two references are not dependent.
Crosscutting Issues

- DLP processors tend to have lower clock rates and simpler issue logic than GP OoO processors.
- GPUs often have special DRAM chips, called GDRAM, that provide higher bandwidth at lower capacity. GPU memory controllers maintain separate queues of traffic for different GDRAM banks.
- GPUs currently transfer data between I/O devices and system memory and then between system memory and GPU memory, which lowers I/O performance.

Fallacies and Pitfalls

- Pitfall: Concentrating on peak performance in vector architectures and ignoring startup-overhead.
- Fallacy: On GPUs, just add more threads if you don’t have enough memory performance.