Data Parallelism

- Concurrency arises from performing the **same operations on different pieces of data**
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors

- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)

- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel

- SIMD exploits instruction-level parallelism
  - Multiple instructions concurrent: instructions happen to be the same

SIMD Processing

- Single instruction operates on multiple data elements
  - In time or in space

- Multiple processing elements

- Time-space duality
  - **Array processor**: Instruction operates on multiple data elements at the same time
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps
**Array vs. Vector Processors**

**ARRAY PROCESSOR**

LD VR ← A[3:0]
ADD VR ← VR, 1
MUL VR ← VR, 2
ST A[3:0] ← VR

**VECTOR PROCESSOR**

LD ADD MUL ST

- **Same op @ same time**
  - LD0 LD1 LD2 LD3
  - AD0 AD1 AD2 AD3
  - MU0 MU1 MU2 MU3
  - ST0 ST1 ST2 ST3

- **Different ops @ same space**
  - LD0
  - LD1 AD0
  - LD2 AD1 MU0
  - LD3 AD2 MU1 ST0
  - AD3 MU2 ST1
  - MU3 ST2
  - ST3

**Time**

**Space**

**SIMD Array Processing vs. VLIW**

- **VLIW**
  - Program Counter
  - add r1, r2, r3
  - load r4, r5, r6
  - mov r6, r2
  - mul r7, r8, r9

- **Instruction Execution**
  - PE PE PE PE

**Vector Processors**

- **Array processor**
  - Program Counter
  - add VR, VR, 1
  - VLEN = 4

  - Instruction Execution
    - PE PE PE PE

- **A vector is a one-dimensional array of numbers**

- **Many scientific/commercial programs use vectors**
  - for (i = 0; i <= 49; i++)
  - \( C[i] = (A[i] + B[i]) / 2 \)

- **A vector processor is one whose instructions operate on vectors rather than scalar (single data) values**

- **Basic requirements**
  - Need to load/store vectors → **vector registers** (contain vectors)
  - Need to operate on vectors of different lengths → **vector length register** (VLEN)
  - Elements of a vector might be stored apart from each other in memory → **vector stride register** (VSTR)
    - Stride: distance between two elements of a vector
Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element

- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking within a vector
  - No control flow within a vector
  - Known stride allows prefetching of vectors into cache/memory

Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining, parallelization work well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth

+ Highly regular memory access pattern
  - Interleaving multiple banks for higher memory bandwidth
  - Prefetching

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence

Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)
++ Vector operations
  -- Very inefficient if parallelism is irregular
    -- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is


Vector Processor Limitations

-- Memory (bandwidth) can easily become a bottleneck, especially if
  1. compute/memory operation balance is not maintained
  2. data is not mapped appropriately to memory banks
Vector Registers
- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK[i] = (V[i] == 0)
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register

Vector Functional Units
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent

Vector Machine Organization (CRAY-1)
- CRAY-1
- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Memory Banking
- Example: 16 banks; can start one bank access per cycle
- Bank latency: 11 cycles
- Can sustain 16 parallel accesses if they go to different banks
Vector Memory System

Scalar Code Example

- For \( I = 0 \) to \( 49 \)
  - \( C[I] = (A[I] + B[I]) / 2 \)

- Scalar code
  - MOV R0 = 50
  - MOV R1 = A
  - MOV R2 = B
  - MOV R3 = C
  - X:  LD R4 = MEM[R1++]  ; autoincrement addressing
  - LD R5 = MEM[R2++]
  - ADD R6 = R4 + R5
  - SHFR R7 = R6 >> 1
  - ST MEM[R3++] = R7
  - DECBNZ --R0, X  ; decrement and branch if NZ

- 304 dynamic instructions

Scalar Code Execution Time

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved)
  - First two loads in the loop can be pipelined
  - 4 + 50*30 = 1504 cycles

- Why 16 banks?
  - 11 cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency

Vectorizable Loops

- A loop is **vectorizable** if each iteration is independent of any other

- For \( I = 0 \) to \( 49 \)
  - \( C[I] = (A[I] + B[I]) / 2 \)

- Vectorized loop:
  - MOV VLEN = 50
  - MOV VSTR = 1
  - VLD V0 = A
  - VLD V1 = B
  - VADD V2 = V0 + V1
  - VSHFR V3 = V2 >> 1
  - VST C = V3

- 7 dynamic instructions
Vector Code Performance

- No chaining
  - i.e., output of a vector functional unit cannot be used as the input of another (i.e., no vector data forwarding)
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

285 cycles

Vector Chaining

- Vector chaining: Data forwarding from one vector functional unit to another

16 memory banks (word-interleaved)

21 memory ports (one address generator)

285 cycles

Vector Code Performance - Chaining

- Vector chaining: Data forwarding from one vector functional unit to another

These two VLDs cannot be pipelined. WHY?

Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)

182 cycles

Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank

VLD and VST cannot be pipelined. WHY?

79 cycles
Questions (I)

- What if # data elements > # elements in a vector register?
  - Need to break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector strip-mining

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Use indirection to combine elements into vector registers
  - Called scatter/gather operations

Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

```plaintext
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (Gather)

- `LV vd, rd`  # Load indices in D vector
- `LVI vc, rc, vd`  # Load indirect from rC base
- `LV vb, rb`  # Load B vector
- `ADDV.D va, vb, vc`  # Do add
- `SV va, ra`  # Store result

Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse matrices
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

<table>
<thead>
<tr>
<th>Index Vector</th>
<th>Data Vector</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14</td>
<td>3.14</td>
</tr>
<tr>
<td>3</td>
<td>6.5</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>71.2</td>
<td>6.5</td>
</tr>
<tr>
<td>8</td>
<td>2.71</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?
  ```plaintext
  loop:
  if (a[i] != 0) then b[i]=a[i]*b[i]
  goto loop
  ```
- Idea: Masked operations
  - VMASK register is a bit mask determining which data element should not be acted upon
    ```plaintext
    VLD V0 = A
    VLD V1 = B
    VMASK = (V0 != 0)
    VMUL V1 = V0 * V1
    VST B = V1
    ```
  - Does this look familiar? This is essentially predicated execution.
Another Example with Masking

for (i = 0; i < 64; ++i)
    if (a[i] >= b[i]) then c[i] = a[i]
    else c[i] = b[i]

Steps to execute loop

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

A | B | VMASK
---|---|---
1 | 2 | 0
2 | 2 | 1
3 | 2 | 1
4 | 10| 0
-5| -4| 0
0 | -3| 1
6 | 5 | 1
-7| -8| 1

Some Issues

- Stride and banking
  - As long as they are relatively prime to each other and there are enough banks to cover bank access latency, consecutive accesses proceed in parallel

- Storage of a matrix
  - Row major: Consecutive elements in a row are laid out consecutively in memory
  - Column major: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column
Array vs. Vector Processors, Revisited

- Array vs. vector processor distinction is a “purist’s” distinction
  - Most “modern” SIMD processors are a combination of both
    - They exploit data parallelism in both time and space

---

Vector Instruction Execution

ADDV C, A, B

- Execution using one pipelined functional unit
- Execution using four pipelined functional units

Vector Unit Structure

- Functional Unit
- Memory Subsystem
- Elements 0, 4, 8, ...
- Elements 1, 5, 9, ...
- Elements 2, 6, 10, ...
- Elements 3, 7, 11, ...
- Lane
- Vector Registers
Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Complete 24 operations/cycle while issuing 1 short instruction/cycle

Automatic Code Vectorization

Scalar Sequential Code

C[i] = A[i] + B[i];

Vectorized Code

Vectorization is a compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Vector/SIMD Processing Summary

- Vector/SIMD machines good at exploiting **regular data-level parallelism**
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

- Performance improvement limited by **vectorizability** of code
  - Scalar operations limit vector machine performance
  - Amdahl’s Law
  - CRAY-1 was the fastest SCALAR machine at its time!

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

SIMD Operations in Modern ISAs
Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - Ala array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

No VLEN register
Opcode determines data type:
- 8 8-bit bytes
- 4 16-bit words
- 2 32-bit doublewords
- 1 64-bit quadword

Stride always equal to 1.


MMX Example: Image Overlaying (I)

Figure 8. Chroma keying image overlay using a background color.

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
High-Level View of a GPU

Concept of “Thread Warps” and SIMT
- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same kernel
- Warp: The threads that run lengthwise in a woven fabric ...

Loop Iterations as Threads

SIMT Memory Access
- Same instruction in different threads uses thread id to index and access different data elements

Slide credit: Krste Asanovic
Slide credit: Hyesoon Kim
Sample GPU SIMT Code (Simplified)

```c
for (ii = 0; ii < 100; ++ii) {
}
```

CUDA code

```c
// there are 100 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```

Sample GPU Program (Less Simplified)

**CPU Program**

```c
void add_matrix
    ( float *a, float *b, float *c, int N)
{
    int index;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
    }
}
```

**GPU Program**

```c
__global__ add_matrix
    ( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index] + b[index];
}
```

Latency Hiding with “Thread Warps”

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies
  - Register values of all threads stay in register file
  - No OS context switching
  - Memory latency hiding

- **Warp-based SIMD vs. Traditional SIMD**
  - **Traditional SIMD** contains a single thread
    - Lock step
    - Programming model is SIMD (no threads) → SW needs to know vector length
    - ISA contains vector/SIMD instructions
  - **Warp-based SIMD** consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
    - Does not have to be lock step
    - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - **ISA** is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

Slide credit: Tor Aamodt
SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization
  - Each processing element executes the same procedure, except on different data elements
    - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD computer

Branch Divergence Problem in Warp-based SIMD

- SPMD Execution on SIMD Hardware
  - NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution

Control Flow Problem in GPUs/SIMD

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps
- Branch divergence occurs when threads inside warps branch to different execution paths.

Branch Divergence Handling (I)
Dynamic Warp Formation

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warp at divergence
  - Enough threads branching to each path to create full new warps

Dynamic Warp Formation Example

What About Memory Divergence?

- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
  - Tolerate memory divergence
  - Integrate solutions to branch and memory divergence

Slide credit: Tor Aamodt
NVIDIA GeForce GTX 285

- **NVIDIA-speak:**
  - 240 stream processors
  - “SIMT execution”

- **Generic speak:**
  - 30 cores
  - 8 SIMD functional units per core

There are 30 of these things on the GTX 285: 30,720 threads