Agenda for This Lecture

- SIMD Processing
  - Vector and Array Processors

- Graphics Processing Units (GPUs)
Exploiting Data Parallelism: SIMD Processors and GPUs
SIMD Processing: Exploiting Regular (Data) Parallelism
Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
Data Parallelism

- Concurrency arises from performing the **same operation 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 operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data
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 using different spaces
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps using the same space
Array vs. Vector Processors

Instruction Stream

**ARRAY PROCESSOR**

<table>
<thead>
<tr>
<th>Time</th>
<th>LD0</th>
<th>LD1</th>
<th>LD2</th>
<th>LD3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD0</td>
<td>AD1</td>
<td>AD2</td>
<td>AD3</td>
<td></td>
</tr>
<tr>
<td>MU0</td>
<td>MU1</td>
<td>MU2</td>
<td>MU3</td>
<td></td>
</tr>
<tr>
<td>ST0</td>
<td>ST1</td>
<td>ST2</td>
<td>ST3</td>
<td></td>
</tr>
</tbody>
</table>

Same op @ same time

Different ops @ same space

**VECTOR PROCESSOR**

<table>
<thead>
<tr>
<th>Time</th>
<th>LD0</th>
<th>AD0</th>
<th>MU0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD1</td>
<td>AD1</td>
<td>MU1</td>
<td>ST0</td>
</tr>
<tr>
<td>LD2</td>
<td>AD2</td>
<td>MU2</td>
<td>ST1</td>
</tr>
<tr>
<td>LD3</td>
<td>AD3</td>
<td>MU3</td>
<td>ST2</td>
</tr>
<tr>
<td>ST0</td>
<td></td>
<td></td>
<td>ST3</td>
</tr>
</tbody>
</table>

Different ops @ time

Same op @ space
SIMD Array Processing vs. VLIW

- VLIW: Multiple independent operations packed together by the compiler
**SIMD Array Processing vs. VLIW**

- Array processor: Single operation on multiple (different) data elements

![Diagram showing SIMD array processing and VLIW execution]
Vector Processors (I)

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors
  
  ```c
  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 in memory 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 needed within a vector
  - No control flow within a vector
  - Known stride allows easy address calculation for all vector elements
    - Enables prefetching of vectors into registers/cache/memory
Vector Processor Advantages

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

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

+ Highly regular memory access pattern

+ 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 Processing in More Depth
Vector Registers

- Each **vector data register** holds N M-bit values
- **Vector control registers**: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- **Vector Mask Register** (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., \( \text{VMASK}[i] = (V_k[i] == 0) \)

<table>
<thead>
<tr>
<th>V0,0</th>
<th>V0,1</th>
<th>V1,0</th>
<th>V1,1</th>
<th>V0,N-1</th>
<th>V1,N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-bit wide</td>
<td>M-bit wide</td>
<td>M-bit wide</td>
<td>M-bit wide</td>
<td>M-bit wide</td>
<td>M-bit wide</td>
</tr>
</tbody>
</table>

17
Vector Functional Units

- Use a deep pipeline to execute element operations → fast clock cycle
- Control of deep pipeline is simple because elements in vector are independent

Slide credit: Krste Asanovic
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
CRAY X-MP-28 @ ETH (CAB, E Floor)

Cray X-MP-28


Miniatürisierung und explodierende Leistung
Mit realem schnell sich die Leistungsfähigkeit der Hardware entwickelt hat, zeigt der Vergleich des gelben Rechners mit einem Mikrorechner von heute.

Cray X-MP28

Raspberry Pi: 1 model B+
CRAY X-MP Design Detail

CRAY X-MP design detail

Mainframe

CRAY X-MP single- and multiprocessor systems are designed to offer users outstanding performance on large-scale, compute-intensive and I/O-bound jobs.

CRAY X-MP mainframes consist of six (X-MP/1), eight (X-MP/2) or twelve (X-MP/4) vertical columns arranged in an arc. Power supplies and cooling are clustered around the base and extend outward.

<table>
<thead>
<tr>
<th>Model</th>
<th>Number of CPUs</th>
<th>Memory size (millions of 64-bit words)</th>
<th>Number of banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAY X-MP/416</td>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>CRAY X-MP/48</td>
<td>4</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/216</td>
<td>2</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/28</td>
<td>2</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/24</td>
<td>2</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/18</td>
<td>1</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CRAY X-MP/14</td>
<td>1</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/12</td>
<td>1</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>CRAY X-MP/11</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

A description of the major system components and their functions follows.

CPU computation section

Within the computation section of each CPU are operating registers, functional units and an instruction control network — hardware elements that cooperate in executing sequences of instructions. The instruction control network makes all decisions related to instruction issue as well as coordinating the three types of processing within each CPU: vector, scalar and address. Each of the processing modes has its associated registers and functional units.

The block diagram of a CRAY X-MP/4 (opposite page) illustrates the relationship of the registers to the functional units, instruction buffers, I/O channel control registers, interprocessor communications section and memory. For multiple-processor CRAY X-MP models, the interprocessor communications section coordinates processing between CPUs, and central memory is shared.

Registers

The basic set of programmable registers is composed of:

- Eight 24-bit address (A) registers
- Sixty-four 24-bit intermediate address (B) registers
- Eight 64-bit scalar (S) registers
- Sixty-four 64-bit scalar-save (T) registers
- Eight 64-element (4096-bit) vector (V) registers with 64 bits per element

The 24-bit A registers are generally used for addressing and counting operations. Associated with them are 64 B registers, also 24 bits wide. Since the transfer between an A and B register takes only one clock period, the B registers assume the role of data cache, storing information for fast access without tying up the A registers for relatively long periods.

# CRAY X-MP CPU Functional Units

<table>
<thead>
<tr>
<th>CRAY X-MP CPU functional units</th>
<th>Register usage</th>
<th>Time in clock periods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address functional units</strong></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>Addition</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>Multiplication</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Scalar functional units</strong></td>
<td>S</td>
<td>3</td>
</tr>
<tr>
<td>Addition</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Shift-single</td>
<td>S</td>
<td>3</td>
</tr>
<tr>
<td>Shift-double</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>Population, parity and leading zero</td>
<td>S</td>
<td>3 or 4</td>
</tr>
<tr>
<td><strong>Vector functional units</strong></td>
<td>V</td>
<td>3</td>
</tr>
<tr>
<td>Addition</td>
<td>V</td>
<td>3 or 4</td>
</tr>
<tr>
<td>Shift</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Full vector logical</td>
<td>V</td>
<td>2</td>
</tr>
</tbody>
</table>

# CRAZ Y X-MP System Configuration

<table>
<thead>
<tr>
<th>System configuration options</th>
<th>X-MP/1</th>
<th>X-MP/2</th>
<th>X-MP/4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mainframe</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Bipolar memory (64-bit words)</td>
<td>N/A</td>
<td>N/A</td>
<td>8 or 16M</td>
</tr>
<tr>
<td>MOS memory (64-bit words)</td>
<td>1, 2, 4 or 8M</td>
<td>4, 8 or 16M</td>
<td>N/A</td>
</tr>
<tr>
<td>6-Mbyte channels</td>
<td>2 or 4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>100-Mbyte channels</td>
<td>1 or 2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1000-Mbyte channels</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>I/O Subsystem</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O processors</td>
<td>2, 3 or 4</td>
<td>2, 3 or 4</td>
<td>4</td>
</tr>
<tr>
<td>Disk storage units</td>
<td>2-32</td>
<td>2-32</td>
<td>2-32</td>
</tr>
<tr>
<td>Magnetic tape channels</td>
<td>1-8</td>
<td>1-8</td>
<td>1-8</td>
</tr>
<tr>
<td>Front-end interfaces</td>
<td>1-7</td>
<td>1-7</td>
<td>1-7</td>
</tr>
<tr>
<td>Buffer memory (Mbytes)</td>
<td>8, 32 or 64</td>
<td>8, 32 or 64</td>
<td>64</td>
</tr>
<tr>
<td><strong>Solid-state Storage Device</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory size (Mbytes)</td>
<td>256, 512 or 1024</td>
<td>256, 512 or 1024</td>
<td>256, 512 or 1024</td>
</tr>
</tbody>
</table>

N/A signifies option is not available on the model.
Seymour Cray, the Father of Supercomputers

"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"
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
Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now

- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle

- Question: How do we achieve this with a memory that takes more than 1 cycle to access?

- Answer: Bank the memory; interleave the elements across banks
Memory Banking

- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- **Can sustain N parallel accesses if all N go to different banks**

![Diagram of Memory Banking](Picture credit: Derek Chiou)
Vector Memory System

- Next address = Previous address + Stride
- If (stride == 1) && (consecutive elements interleaved across banks) && (number of banks >= bank latency), then
  - we can sustain 1 element/cycle throughput

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

- Scalar code (instruction and its latency)

```
MOVI R0 = 50
MOVA R1 = A
MOVA R2 = B
MOVA R3 = C
X:  LD R4 = MEM[R1++]
    LD R5 = MEM[R2++]
    ADD R6 = R4 + R5
    SHFR R7 = R6 >> 1
    ST MEM[R3++] = R7
    DECBNZ R0, X
```

304 dynamic instructions
Scalar Code Execution Time (In Order)

- 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: consecutive words are stored in consecutive banks)
  - 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 (each instruction and its latency):
  - MOVI VLEN = 50  
  - MOVI VSTR = 1  
  - VLD V0 = A  
  - VLD V1 = B  
  - VADD V2 = V0 + V1  
  - VSHFR V3 = V2 >> 1  
  - VST C = V3  

  7 dynamic instructions
Basic Vector Code Performance

- Assume **no chaining** (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation

- 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

```
LV v1
MULV v3, v1, v2
ADDV v5, v3, v4
```

Slide credit: Krste Asanovic
Vector Code Performance - Chaining

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

- **182 cycles**

These two VLDs cannot be pipelined. WHY?

VLD and VST cannot be pipelined. WHY?

Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)
Vector Code Performance – Multiple Memory Ports

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

- 79 cycles
- 19X perf. improvement!
Questions (I)

- What if # data elements > # elements in a vector register?
  - Idea: 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 stripmining
(Vector) Stripmining

**Surface mining**, including **strip mining**, open-pit mining and mountaintop removal mining, is a broad category of mining in which soil and rock overlying the mineral deposit (the **overburden**) are removed, in contrast to **underground mining**, in which the overlying rock is left in place, and the mineral removed through shafts or tunnels. Surface mining began in the mid-sixteenth century[^1] and is practiced throughout the world, although the majority of surface coal mining occurs in North America[^2] It gained
Questions (II)

- What if vector data is not stored in a strided fashion in memory? *(irregular memory access to a vector)*
  - Idea: **Use indirection to combine/pack elements into vector registers**
  - Called **scatter/gather operations**
Want to vectorize loops with indirect accesses:

```c
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 vectors (matrices)
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

**Scatter example**

<table>
<thead>
<tr>
<th>Index Vector</th>
<th>Data Vector (to Store)</th>
<th>Stored Vector (in Memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.14</td>
<td>Base+0 3.14</td>
</tr>
<tr>
<td>2</td>
<td>6.5</td>
<td>Base+1 X</td>
</tr>
<tr>
<td>6</td>
<td>71.2</td>
<td>Base+2 6.5</td>
</tr>
<tr>
<td>7</td>
<td>2.71</td>
<td>Base+3 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+4 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+5 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+6 71.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+7 2.71</td>
</tr>
</tbody>
</table>
What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```c
loop: for (i=0; i<N; i++)
    if (a[i] != 0) then b[i] = a[i]*b[i]
```

**Idea: Masked operations**

- VMASK register is a bit mask determining which data element should not be acted upon
  
  ```
  VLD V0 = A  
  VLD V1 = B  
  VMASK = (V0 != 0)  
  VMUL V1 = V0 * V1  
  VST B = V1  
  ```

- This is *predicated execution*. Execution is *predicated* on mask bit.
Another Example with Masking

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

Steps to execute the loop in SIMD code

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>VMASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>-8</td>
<td>1</td>
</tr>
</tbody>
</table>
Masked Vector Instructions

Simple Implementation
- execute all N operations, turn off result writeback according to mask

\[ M[2]=0 \quad C[2] \]
\[ M[1]=1 \quad C[1] \]
\[ M[0]=0 \quad Write Enable \]

Density-Time Implementation
- scan mask vector and only execute elements with non-zero masks

\[ M[7]=1 \]
\[ M[6]=0 \]
\[ M[5]=1 \]
\[ M[4]=1 \]
\[ M[3]=0 \]
\[ M[2]=0 \]
\[ M[1]=1 \]
\[ M[0]=0 \]

Which one is better?
Tradeoffs?
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, we can sustain 1 element/cycle throughput

- 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
Matrix Multiplication

- A and B, both in **row-major order**

  A:
  - Load $A_0$ into vector register $V_1$
    - Each time, increment address by one to access the next column
    - Accesses have a **stride of 1**

  B:
  - Load $B_0$ into vector register $V_2$
    - Each time, increment address by 10
    - Accesses have a **stride of 10**

  Different strides can lead to bank conflicts

How do we minimize them?
Minimizing Bank Conflicts

- More banks

- Better data layout to match the access pattern
  - Is this always possible?

- Better mapping of address to bank
  - E.g., randomized mapping
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
  - GPUs are a prime example we will cover in a bit more detail
Recall: Array vs. Vector Processors

**ARRAY PROCESSOR**

- PE0
- PE1
- PE2
- PE3

**VECTOR PROCESSOR**

- LD
- ADD
- MUL
- ST

### Instruction Stream

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

### Time

- LD0
- LD1
- LD2
- LD3
- AD0
- AD1
- AD2
- AD3
- MU0
- MU1
- MU2
- MU3
- ST0
- ST1
- ST2
- ST3

### Space

- Same op @ same time
- Different ops @ time
- Different ops @ same space
- Same op @ space
Vector Instruction Execution

**VADD A, B → C**

**Execution using one pipelined functional unit**


**Execution using four pipelined functional units**


Time


Space

Slide credit: Krste Asanovic
Vector Unit Structure

Functional Unit

Partitioned Vector Registers

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Memory Subsystem

Slide credit: Krste Asanovic
Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

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

Slide credit: Krste Asanovic
Automatic Code Vectorization

Scalar Sequential Code

Vectorized Code

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

Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Vector Instruction

Slide credit: Krste Asanovic
Vector/SIMD Processing Summary

- Vector/SIMD machines are 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
  - Remember 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
Single Instruction Multiple Data (SIMD) extension instructions

- Single instruction acts on multiple pieces of data at once
- Common application: graphics
- Perform short arithmetic operations (also called *packed arithmetic*)

For example: add four 8-bit numbers

Must modify ALU to eliminate carries between 8-bit values
**Intel Pentium MMX Operations**

- **Idea:** One instruction operates on multiple data elements simultaneously
  - À la 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 is always equal to 1.

Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

MMX Example: Image Overlaying (I)

- **Goal:** Overlay the human in image 1 on top of the background in image 2.

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

**Figure 9. Generating the selection bit mask.**

```plaintext
PCMPEQD MM1, MM3

| MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0xFFFF | 0xFFFF |
| MM3 | X7=blue| X6=blue| X5=blue| X4=blue| X3=blue| X2=blue| X1=blue|
| MM1 | Blue   | Blue   | Blue   | Blue   | Blue   | Blue   | Blue   |
```

The code operation is:

```c
for (i=0; i<image size; i++) |
  if (x[i] == Blue) new_image[i] = y[i];
  else new_image[i] = x[i];
```
Fine-Grained Multithreading
Fine-Grained Multithreading (I)

- **Idea:** Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
-- Single thread performance suffers
-- Extra logic for keeping thread contexts
-- Does not overlap latency if not enough threads to cover the whole pipeline
Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently.

- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads.

- Improves pipeline utilization by taking advantage of multiple threads.


Multithreaded Pipeline Example

Slide credit: Joel Emer
Fine-grained Multithreading (III)

- **Advantages**
  - No need for dependency checking between instructions (only one instruction in pipeline from a single thread)
  - No need for branch prediction logic
  - Otherwise-bubble cycles used for executing useful instructions from different threads
  - Improved system throughput, latency tolerance, utilization

- **Disadvantages**
  - Extra hardware complexity: multiple hardware contexts (PCs, register files, …), thread selection logic
  - Reduced single thread performance (one instruction fetched every N cycles from the same thread)
  - Resource contention between threads in caches and memory
  - Some dependency checking logic between threads remains (load/store)
GPUs (Graphics Processing Units)
GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)

- However, the programming is done using threads, NOT SIMD instructions

- To understand this, let’s go back to our parallelizable code example

- But, before that, let’s distinguish between
  - Programming Model (Software)
  - Execution Model (Hardware)
Programming Model vs. Hardware Execution Model

- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...

- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...

- Execution Model can be very different from the Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)
How Can You Exploit Parallelism Here?

Scalar Sequential Code

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

Let’s examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)
Prog. Model 1: Sequential (SISD)

Scalar Sequential Code

- Can be executed on a:
  - Pipelined processor
  - Out-of-order execution processor
    - Independent instructions executed when ready
    - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
    - In other words, the loop is dynamically unrolled by the hardware
  - Superscalar or VLIW processor
    - Can fetch and execute multiple instructions per cycle

\[
\text{for } (i=0; i < N; i++) \quad C[i] = A[i] + B[i];
\]
**Prog. Model 2: Data Parallel (SIMD)**

Scalar Sequential Code

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

Vector Instruction

```
VLD     A → V1
VLD     B → V2
VADD    V1 + V2 → V3
VST     V3 → C
```

Vectorized Code

**Realization:** Each iteration is independent

**Idea:** Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

**Best executed by a SIMD processor (vector, array)**
Iter. 1

Scalar Sequential Code

Iter. 2

Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

for (i=0; i < N; i++)
C[i] = A[i] + B[i];
Prog. Model 3: Multithreaded

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

Iter. 1

load
load
add
store

Iter. 2

load
load
add
store

Realization: Each iteration is independent

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread
A GPU is a SIMD (SIMT) Machine

- Except it is **not** programmed using SIMD instructions

- It is **programmed using threads** (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
  - A warp is essentially a **SIMD operation formed by hardware**!
for (i=0; i < N; i++)

Iter. 1
Iter. 2

\textbf{Warp: A set of threads that execute the same instruction (i.e., at the same PC)}

This particular model is also called:

**SPMD: Single Program Multiple Data**

A GPU executes it using the SIMT model:

**Single Instruction Multiple Thread**
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
SIMD vs. SIMT Execution Model

- **SIMD:** A single **sequential instruction stream of SIMD instructions** → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT:** Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

**Two Major SIMT Advantages:**
- **Can treat each thread separately** → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
- **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
Fine-Grained Multithreading of Warps

- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps

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

Iter. \(19\times 32 + 1\)  Iter. \(20\times 32 + 2\)
Warps and Warp-Level FGMT

- **Warp:** A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code
- **Warp:** The threads that run lengthwise in a woven fabric ...

---

High-Level View of a GPU

Latency Hiding via Warp-Level FGMT

- **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 interlocking)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- FGMT enables long latency tolerance
  - Millions of pixels
Warp Execution (Recall the Slide)

32-thread warp executing ADD A[tid], B[tid] → C[tid]

Execution using one pipelined functional unit

Execution using four pipelined functional units

Time

Space

Slide credit: Krste Asanovic
SIMD Execution Unit Structure

- Registers for each Thread
  - Registers for thread IDs 0, 4, 8, ...
  - Registers for thread IDs 1, 5, 9, ...
  - Registers for thread IDs 2, 6, 10, ...
  - Registers for thread IDs 3, 7, 11, ...

- Memory Subsystem

- Lane

- Functional Unit

Slide credit: Krste Asanovic
Warp Instruction Level Parallelism

Can overlap execution of multiple instructions
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle
SIMT Memory Access

- Same instruction in different threads uses **thread id** to index and access different data elements

Let’s assume \( N=16 \), 4 threads per warp \( \rightarrow \) 4 warps

Slide credit: Hyesoon Kim
Warps *not* Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads

Serial Code (host)

Parallel Kernel (device)
KernelA\(\lll nBlk, nThr \rrr\)(args);

Serial Code (host)

Parallel Kernel (device)
KernelB\(\lll nBlk, nThr \rrr\)(args);

Slide credit: Hwu & Kirk
CPU code

```c
for (ii = 0; ii < 100000; ++ii) {
}
```

CUDA code

```c
// there are 100000 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```
**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];
        }
}

int main () {
    add_matrix (a, b, c, N);
}
```

**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];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```
From Blocks to Warps

- GPU cores: SIMD pipelines
  - Streaming Multiprocessors (SM)
  - Streaming Processors (SP)

- Blocks are divided into warps
  - SIMD unit (32 threads)
Warp-based SIMD vs. Traditional SIMD

- Traditional **SIMD** contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra threads) \(\rightarrow\) 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) \(\rightarrow\) programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar \(\rightarrow\) SIMD operations can be formed dynamically
  - Essentially, it is **SPMD programming model implemented on SIMD hardware**
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 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware
SIMD vs. SIMT Execution Model

- **SIMD**: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

Two Major SIMT Advantages:

- **Can treat each thread separately** → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
- **Can group threads into warps flexibly** → i.e., can group threads that are supposed to truly execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths
A GPU uses a SIMD pipeline to save area on control logic
- Groups scalar threads into warps

Branch divergence occurs when threads inside warps branch to different execution paths

This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?
Remember: Each Thread Is Independent

Two Major SIMT Advantages:

- Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
- Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

If we have many threads
- We can find individual threads that are at the same PC
- And, group them together into a single warp dynamically
- This reduces “divergence” → improves SIMD utilization
  - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)
- **Form new warps from warps that are waiting**
  - Enough threads branching to each path enables the creation of full new warps

![Diagram showing warp formation and merging](image_url)
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)

Dynamic Warp Formation Example

A new warp created from scalar threads of both Warp x and y executing at Basic Block D.

Legend
- Execution of Warp x at Basic Block A
- Execution of Warp y at Basic Block A
- A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Baseline

Dynamic Warp Formation

Slide credit: Tor Aamodt
Hardware Constraints Limit Flexibility of Warp Grouping

Can you move any thread flexibly to any lane?
Large Warps and Two-Level Warp Scheduling

- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - Long latency operations

Large Warp Microarchitecture Example

- Reduce **branch divergence** by having large warps
- **Dynamically break down a large warp into sub-warsps**

**Decode Stage**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sub-warp 0 mask: \[1 1 1 1\]

Sub-warp 0 mask: \[1 1 1 1\]

Sub-warp 0 mask: \[1 1 1 1\]

Narasiman et al., “**Improving GPU Performance via Large Warps and Two-Level Warp Scheduling**,” MICRO 2011.
Two-Level Round Robin

- **Scheduling in two levels** to deal with **long latency operations**

```
Core
All Warps Compute

Memory System
Req Warp 0
Req Warp 1

Req Warp 15

Round Robin Scheduling, 16 total warps

Core
Group 0
Compute

Group 1
Compute

Memory System
Req Warp 0
Req Warp 1

Req Warp 7

Req Warp 8
Req Warp 9

Req Warp 15

Two Level Round Robin Scheduling, 2 fetch groups, 8 warps each

Improving Warp Scheduling

- Veynu Narasiman, Chang Joo Lee, Michael Shebanow, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling"

Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011.

Slides (ppt)
