EECS 570
Lecture 3
Data-level Parallelism
Winter 2019
Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Martin, Roth, Nowatzyk, and Wenisch of EPFL, CMU, UPenn, U-M, UIUC.
Announcements

Discussion this Friday: Project Kick-off

No class or office hour Monday 1/21 (MLK Day)
Readings

For today:

- H Kim, R Vuduc, S Baghsorkhi, J Choi, Wen-mei Hwu, xPerformance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU), Ch. 1

For Wednesday 1/18:

- Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Ch. 3.1-3.3, 4.1-4.3
Programming Model Elements

- For both Shared Memory and Message Passing

- Processes and threads
  - **Process**: A shared address space and one or more threads of control
  - **Thread**: A program sequencer and private address space
  - **Task**: Less formal term – part of an overall job
  - Created, terminated, scheduled, etc.

- Communication
  - Passing of data

- Synchronization
  - Communicating control information
  - To assure reliable, deterministic communication
Historical View

- **Join at:** I/O (Network)
- **Program with:** Message passing
- **Memory**
- **Processor**
  - Dataflow, SIMD, VLIW, CUDA, other data parallel
Message Passing Programming Model
Message Passing Programming Model

- User level send/receive abstraction
  - Match via local buffer \((x,y)\), process \((Q,P)\), and tag \((t)\)
  - Need naming/synchronization conventions
Message Passing Architectures

- Cannot directly access memory of another node
- IBM SP-2, Intel Paragon, Myrinet Quadrics QSW
- Cluster of workstations (e.g., MPI on flux cluster)
**MPI - Message Passing Interface API**

- A widely used standard
  - For a variety of distributed memory systems
    - SMP Clusters, workstation clusters, MPPs, heterogeneous systems
- Also works on Shared Memory MPs
  - Easy to emulate distributed memory on shared memory HW
- Can be used with a number of high level languages
- Available in the Flux cluster at Michigan
Processes and Threads

- Lots of flexibility (advantage of message passing)
  1. Multiple threads sharing an address space
  2. Multiple processes sharing an address space
  3. Multiple processes with different address spaces
     - and different OSes

- 1 and 2 easily implemented on shared memory HW (with single OS)
  - Process and thread creation/management similar to shared memory

- 3 probably more common in practice
  - Process creation often external to execution environment; e.g. shell script
  - Hard for user process on one system to create process on another OS
Communication and Synchronization

- Combined in the message passing paradigm
  - Synchronization of messages part of communication semantics
- Point-to-point communication
  - From one process to another
- Collective communication
  - Involves groups of processes
  - e.g., broadcast
Message Passing: Send()

- Send( <what>, <where-to>, <how> )

  - What:
    - A data structure or object in user space
    - A buffer allocated from special memory
    - A word or signal

  - Where-to:
    - A specific processor
    - A set of specific processors
    - A queue, dispatcher, scheduler

  - How:
    - Asynchronously vs. synchronously
    - Typed
    - In-order vs. out-of-order
    - Prioritized
Message Passing: Receive()

• Receive( <data>, <info>, <what>, <how> )

• Data: mechanism to return message content
  □ A buffer allocated in the user process
  □ Memory allocated elsewhere

• Info: meta-info about the message
  □ Sender-ID
  □ Type, Size, Priority
  □ Flow control information

• What: receive only certain messages
  □ Sender-ID, Type, Priority

• How:
  □ Blocking vs. non-blocking
Synchronous vs Asynchronous

- **Synchronous Send**
  - Stall until message has actually been received
  - Implies a message acknowledgement from receiver to sender

- **Synchronous Receive**
  - Stall until message has actually been received

- **Asynchronous Send and Receive**
  - Sender and receiver can proceed regardless
  - Returns *request handle* that can be tested for message receipt
  - Request handle can be tested to see if message has been sent/received
Deadlock

- Blocking communications may deadlock

  <Process 0>
  Send(Process1, Message);
  Receive(Process1, Message);

  <Process 1>
  Send(Process0, Message);
  Receive(Process0, Message);

- Requires careful (safe) ordering of sends/receives

  <Process 0>
  Send(Process1, Message);
  Receive(Process1, Message);

  <Process 1>
  Receive (Process0, Message);
  Send (Process0, Message);
Message Passing Paradigm Summary

Programming Model (Software) point of view:

- Disjoint, separate name spaces
- “Shared nothing”
- Communication via explicit, typed messages: send & receive
Message Passing Paradigm Summary

Computer Engineering (Hardware) point of view:

- Treat inter-process communication as I/O device

- Critical issues:
  - How to optimize API overhead
  - Minimize communication latency
  - Buffer management: how to deal with early/unsolicited messages, message typing, high-level flow control
  - Event signaling & synchronization
  - Library support for common functions (barrier synchronization, task distribution, scatter/gather, data structure maintenance)
Shared Memory Programming Model
Shared-Memory Model

- Multiple execution contexts sharing a single address space
  - Multiple programs (MIMD)
  - Or more frequently: multiple copies of one program (SPMD)
- Implicit (automatic) communication via loads and stores
- Theoretical foundation: PRAM model
Global Shared Physical Address Space

- Communication, sharing, synchronization via loads/stores to shared variables
- Facilities for address translation between local/global address spaces
- Requires OS support to maintain this mapping
Why Shared Memory?

Pluses
- For applications looks like multitasking uniprocessor
- For OS only evolutionary extensions required
- Easy to do communication without OS
- Software can worry about correctness first then performance

Minuses
- Proper synchronization is complex
- Communication is implicit so harder to optimize
- Hardware designers must implement

Result
- Traditionally bus-based Symmetric Multiprocessors (SMPs), and now CMPs are the most success parallel machines ever
- And the first with multi-billion-dollar markets
Thread-Level Parallelism

struct acct_t { int bal; };  
shared struct acct_t accts[MAX_ACCT];  
int id, amt;  
if (accts[id].bal >= amt)  
  {  
    accts[id].bal -= amt;  
    spew_cash();  
  }

• Thread-level parallelism (TLP)
  □ Collection of asynchronous tasks: not started and stopped together
  □ Data shared loosely, dynamically

• Example: database/web server (each query is a thread)
  □ accts is shared, can’t register allocate even if it were scalar
  □ id and amt are private variables, register allocated to r1, r2
Synchronization

- Mutual exclusion : locks, ...
- Order : barriers, signal-wait, ...

- Implemented using read/write/modify to shared location
  - Language-level:
    - libraries (e.g., locks in pthread)
    - Programmers can write custom synchronizations
  - Hardware ISA
    - E.g., test-and-set

- OS provides support for managing threads
  - scheduling, fork, join, futex signal/wait

  We’ll cover synchronization in more detail in a few weeks
Paired vs. Separate Processor/Memory?

• Separate processor/memory
  - Uniform memory access (UMA): equal latency to all memory
    + Simple software, doesn’t matter where you put data
    - Lower peak performance
  - Bus-based UMAs common: symmetric multi-processors (SMP)

• Paired processor/memory
  - Non-uniform memory access (NUMA): faster to local memory
    - More complex software: where you put data matters
    + Higher peak performance: assuming proper data placement
**Shared vs. Point-to-Point Networks**

- **Shared network**: e.g., bus (left)
  - Low latency
  - Low bandwidth: doesn’t scale beyond ~16 processors
  - Shared property simplifies cache coherence protocols (later)

- **Point-to-point network**: e.g., mesh or ring (right)
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are complex
Implementation #1: Snooping Bus MP

- Two basic implementations
- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - Sometimes multiple processors on single chip (CMP)
    - Symmetric multiprocessors (SMPs)
  - Common, I use one everyday
Implementation #2: Scalable MP

- General point-to-point network-based systems
  - Typically processor/memory/router blocks (NUMA)
    - **Glueless MP**: no need for additional “glue” chips
  - Can be arbitrarily large: 1000’s of processors
    - **Massively parallel processors (MPPs)**
  - In reality only government (DoD) has MPPs...
    - Companies have much smaller systems: 32–64 processors
    - **Scalable multi-processors**
Cache Coherence

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track \texttt{accts[241].bal} (address is in \texttt{r3})
No-Cache, No-Problem

- Scenario I: processors have no caches
  - No problem

Processor 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash

Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
### Cache Incoherence

**Scenario II:** processors have write-back caches

- Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
- Can get incoherent (inconsistent)

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1, accts, r3</td>
<td>0: addi r1, accts, r3</td>
</tr>
<tr>
<td>1: ld 0(r3), r4</td>
<td>1: ld 0(r3), r4</td>
</tr>
<tr>
<td>2: blt r4, r2, 6</td>
<td>2: blt r4, r2, 6</td>
</tr>
<tr>
<td>3: sub r4, r2, r4</td>
<td>3: sub r4, r2, r4</td>
</tr>
<tr>
<td>4: st r4, 0(r3)</td>
<td>4: st r4, 0(r3)</td>
</tr>
<tr>
<td>5: call spew_cash</td>
<td>5: call spew_cash</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>V:500</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D:400</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>D:400</th>
<th>V:500</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>D:400</td>
<td>500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>D:400</th>
<th>D:400</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>D:400</td>
<td>500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
  - invalidate
  - update
  - supply value
- depends on state of the block and the protocol
Scalable Cache Coherence

• **Scalable cache coherence**: two part solution

• **Part I**: *bus bandwidth*
  - Replace non-scalable bandwidth substrate (bus)...
  - ...with scalable bandwidth one (point-to-point network, e.g., mesh)

• **Part II**: *processor snooping bandwidth*
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)...
  - ...with scalable *directory protocol* (only spam processors that care)

• We will cover this in Unit 3
Shared Memory Summary

• Shared-memory multiprocessors
  + Simple software: easy data sharing, handles both DLP and TLP
  - Complex hardware: must provide illusion of global address space

• Two basic implementations
  □ Symmetric (UMA) multi-processors (SMPs)
    ○ Underlying communication network: bus (ordered)
      + Low-latency, simple protocols that rely on global order
      - Low-bandwidth, poor scalability
  □ Scalable (NUMA) multi-processors (MPPs)
    ○ Underlying communication network: point-to-point (unordered)
      + Scalable bandwidth
      - Higher-latency, complex protocols
Amdahl’s Law for Tail Latency
[Delimitrou & Kozyrakis]

1. Very strict QoS puts a lot of pressure on 1-thread perf
2. With low QoS constraints, balance ILP and TLP
3. Limited parallelism calls for more powerful cores

Figure 3. Homogeneous server configurations for a budget of R = 100 resource units:
(a) 100 1BCE cores; (b) 25 4BCE cores; and (c) one 100BCE core.

(a) Throughput (QPS) under a tail latency constraint as a system architect increases the resources per core when parallelism is unlimited.

(b) Throughput under a tail latency constraint when parallelism is not plentiful.

Throughput under a tail latency constraint when parallelism is not plentiful.

Throughput under a tail latency constraint as a system architect increases the resources per core when parallelism is unlimited.

Throughput under a tail latency constraint when parallelism is unlimited.

Limited parallelism also affects the degree of parallelism case; overall throughput decreases further, more performant cores are needed to drive down tail latency. When 50% of execution is sequential, a single 100BCE core is optimal, but increasing to 10% serialization, an architect would determine that 10BCE cores are optimal. When core size now shifts to 25 BCEs.

When, for example, the parallel fraction of the computation is 99%, 100% parallelism is unlimited; otherwise, a system architect increases the resources per core when parallelism is limited; these findings remain consistent for individual regions. Using Hill’s and Marty’s notations, the following can be written:

\[
\lambda_t = \frac{1}{\mu_t} = \frac{1}{T_s} \in [0, \infty]
\]

(2)

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)
These findings highlight a disparity in performance at high cost. At the same time, some core components (microservices) will only make sense if QoS = 100 resource units:

\[
T_{s} = 50, 90, 100 \%
\]

Finding 2.

A lower latency constraint is not plentiful.

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)

With QoS = 100 resource units:

\[
Q_{o} = 100
\]

Limited parallelism also affects the degree of parallelism case; overall throughput decreases further, more performant cores are needed to drive down tail latency. When 50% of execution is sequential, a single 100BCE core is optimal, but increasing to 10% serialization, an architect would determine that 10BCE cores are optimal. When core size now shifts to 25 BCEs.

When, for example, the parallel fraction of the computation is 99%, 100% parallelism is unlimited; otherwise, a system architect increases the resources per core when parallelism is limited; these findings remain consistent for individual regions. Using Hill’s and Marty’s notations, the following can be written:

\[
\lambda_t = \frac{1}{\mu_t} = \frac{1}{T_s} \in [0, \infty]
\]

(2)

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)
These findings highlight a disparity in performance at high cost. At the same time, some core components (microservices) will only make sense if QoS = 100 resource units:

\[
T_{s} = 50, 90, 100 \%
\]

Finding 2.

A lower latency constraint is not plentiful.

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)

With QoS = 100 resource units:

\[
Q_{o} = 100
\]

Limited parallelism also affects the degree of parallelism case; overall throughput decreases further, more performant cores are needed to drive down tail latency. When 50% of execution is sequential, a single 100BCE core is optimal, but increasing to 10% serialization, an architect would determine that 10BCE cores are optimal. When core size now shifts to 25 BCEs.

When, for example, the parallel fraction of the computation is 99%, 100% parallelism is unlimited; otherwise, a system architect increases the resources per core when parallelism is limited; these findings remain consistent for individual regions. Using Hill’s and Marty’s notations, the following can be written:

\[
\lambda_t = \frac{1}{\mu_t} = \frac{1}{T_s} \in [0, \infty]
\]

(2)

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)
These findings highlight a disparity in performance at high cost. At the same time, some core components (microservices) will only make sense if QoS = 100 resource units:

\[
T_{s} = 50, 90, 100 \%
\]

Finding 2.

A lower latency constraint is not plentiful.

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)

With QoS = 100 resource units:

\[
Q_{o} = 100
\]

Limited parallelism also affects the degree of parallelism case; overall throughput decreases further, more performant cores are needed to drive down tail latency. When 50% of execution is sequential, a single 100BCE core is optimal, but increasing to 10% serialization, an architect would determine that 10BCE cores are optimal. When core size now shifts to 25 BCEs.

When, for example, the parallel fraction of the computation is 99%, 100% parallelism is unlimited; otherwise, a system architect increases the resources per core when parallelism is limited; these findings remain consistent for individual regions. Using Hill’s and Marty’s notations, the following can be written:

\[
\lambda_t = \frac{1}{\mu_t} = \frac{1}{T_s} \in [0, \infty]
\]

(2)

Arrival rate: λ
Service time: \( T_s = \frac{1}{\mu} \)
Amdahl’s Law for Tail Latency

[Delimitrou & Kozyrakis]

4. For medium QoS, ratio of big-to-small cores should follow ratio of big-to-small requests

5. But, as $f_{\text{parallel}}$ decreases, big cores are rapidly favored

(c) Throughput (QPS) under a tail latency constraint as a system architect increases the resources for small cores ($U_1=1$) under the assumption of unlimited parallelism;
Amdahl’s Law for Tail Latency
[Delimitrou & Kozyrakis]

Figure 6. Server configurations with 10 BCE cores when dedicating (a) 10 resource units and (b) 70 resource units toward caching.

6. 30-50% area for cache is ideal for workloads with locality & strict QoS
7. Less cache needed (~30%) with QoS less strict
8. Less parallelism $\rightarrow$ need more cache

Arrival rate: $\lambda$

Service time: $T = 1/\mu$
Data-Level Parallelism
How to Compute This Fast?

- Performing the **same** operations on **many** data items
  - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

- Instruction-level parallelism (ILP) - fine grained
  - Loop unrolling with static scheduling –or– dynamic scheduling
  - Wide-issue superscalar (non-)scaling limits benefits

- Thread-level parallelism (TLP) - coarse grained
  - Multicore

- Can we do some “medium grained” parallelism?
Data-Level Parallelism

• Data-level parallelism (DLP)
  - Single operation repeated on multiple data elements
    - SIMD (Single-Instruction, Multiple-Data)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with vectors

• Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point “Vector registers”
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector addition, subtraction, multiply, etc.
    - Vector+Constant addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!
  - ALUs were expensive, did not perform 64 ops in parallel!
Vector Architectures

- One way to exploit data level parallelism: **vectors**
  - Extend processor with **vector “data type”**
  - Vector: array of 32-bit FP numbers
    - Maximum vector length (MVL): typically 8–64
  - Vector register file: 8–16 vector registers (v0–v15)
Today's Vectors / SIMD
Example Vector ISA Extensions (SIMD)

• Extend ISA with floating point (FP) vector storage ...
  - Vector register: fixed-size array of 32- or 64-bit FP elements
  - Vector length: For example: 4, 8, 16, 64, ...

• ... and example operations for vector length of 4
  - Load vector: \texttt{ldf.v [X+r1]} \rightarrow v1
    \begin{align*}
    \text{ldf} & \ [X+r1+0] \rightarrow v1_0 \\
    \text{ldf} & \ [X+r1+1] \rightarrow v1_1 \\
    \text{ldf} & \ [X+r1+2] \rightarrow v1_2 \\
    \text{ldf} & \ [X+r1+3] \rightarrow v1_3 \\
    \end{align*}
  - Add two vectors: \texttt{addf.vv v1,v2} \rightarrow v3
    \begin{align*}
    \text{addf} & \ v1_i,v2_i \rightarrow v3_i \text{ (where } i \text{ is 0,1,2,3)} \\
    \end{align*}
  - Add vector to scalar: \texttt{addf-vs v1,f2,v3}
    \begin{align*}
    \text{addf} & \ v1_i,f2 \rightarrow v3_i \text{ (where } i \text{ is 0,1,2,3)}
    \end{align*}

• Today’s vectors: short (128 bits), but fully parallel
Example Use of Vectors - 4-wide

Operations

- Load vector: *ldf.v [X+r1]→v1*
- Multiply vector to scalar: *mulf.vs v1,f0→v2*
- Add two vectors: *addf.vv v1,v2→v3*
- Store vector: *stf.v v1→[X+r1]*

• Performance?
  - Best case: 4x speedup
  - But, vector instructions don’t always have 1-cycle throughput
    - Execution width (implementation) vs vector width (ISA)
Vector Datapath & Implementation

- Vector insn. are just like normal insn... only “wider”
  - Single instruction fetch (no extra $N^2$ checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate FP unit (same as superscalar)
  - Wide bypass (avoid $N^2$ bypass problem)
  - Wide cache read & write (single cache tag check)

- Execution width (implementation) vs vector width (ISA)
  - E.g. Pentium 4 and “Core 1” executes vector ops at half width
  - “Core 2” executes them at full width

- Because they are just instructions...
  - ...superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle
Intel’s SSE2/SSE3/SSE4...

- Intel SSE2 (Streaming SIMD Extensions 2) - 2001
  - 16 128bit floating point registers (xmm0–xmm15)
  - Each can be treated as 2x64b FP or 4x32b FP (“packed FP”)
    - Or 2x64b or 4x32b or 8x16b or 16x8b ints (“packed integer”)
    - Or 1x64b or 1x32b FP (just normal scalar floating point)
  - Original SSE: only 8 registers, no packed integer support

- Other vector extensions
  - AMD 3DNow!: 64b (2x32b)
  - PowerPC AltiVEC/VMX: 128b (2x64b or 4x32b)

- Intel’s AVX-512
  - Intel’s “Haswell” and Xeon Phi brought 512-bit vectors to x86
Other Vector Instructions

- These target specific domains: e.g., image processing, crypto
  - Vector reduction (sum all elements of a vector)
  - Geometry processing: 4x4 translation/rotation matrices
  - Saturating (non-overflowing) subword add/sub: image processing
  - Byte asymmetric operations: blending and composition in graphics
  - Byte shuffle/permute: crypto
  - Population (bit) count: crypto
  - Max/min/argmax/argmin: video codec
  - Absolute differences: video codec
  - Multiply-accumulate: digital-signal processing
  - Special instructions for AES encryption

- More advanced (but in Intel’s Xeon Phi)
  - Scatter/gather loads: indirect store (or load) from a vector of pointers
  - Vector mask: predication (conditional execution) of specific elements
Using Vectors in Your Code
Using Vectors in Your Code

- Write in assembly
  - Ugh

- Use “intrinsic” functions and data types
  - For example: _mm_mul_ps() and “__m128” datatype

- Use vector data types
  - typedef double v2df __attribute__((vector_size (16)));

- Use a library someone else wrote
  - Let them do the hard work
  - Matrix and linear algebra packages

- Let the compiler do it (automatic vectorization, with feedback)
  - GCC’s “-ftree-vectorize” option, -ftree-vectorizer-verbose=n
  - Limited impact for C/C++ code (old, hard problem)
New Developments in “CPU” Vectors
Emerging Features

• Past vectors were limited
  □ Wide compute
  □ Wide load/store of consecutive addresses
  □ Allows for “SOA” (structures of arrays) style parallelism

• Looking forward (and backward)...
  □ Vector masks
    □ Conditional execution on a per-element basis
    □ Allows vectorization of conditionals
  □ Scatter/gather
    □ $a[i] = b[y[i]]$  $b[y[i]] = a[i]$
    □ Helps with sparse matrices, “AOS” (array of structures) parallelism

• Together, enables a different style vectorization
  □ Translate arbitrary (parallel) loop bodies into vectorized code (later)
Vector Masks (Predication)

- **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations
    
    ```
    for (I=0; I<N; I++) if (maskI) { vop... }
    ```
  - Usually stored in a “scalar” register (up to 64-bits)
  - Used to vectorize loops with conditionals in them
    
    ```
    cmp_eq.v, cmp_lt.v, etc.: sets vector predicates
    ```
    
    ```
    for (I=0; I<32; I++)
        if (X[I] != 0.0) Z[I] = A/X[I];
    ```

    ```
    ldf.v [X+r1] -> v1
    cmp_ne.v v1,f0 -> r2     // 0.0 is in f0
    divf.sv {r2} v1,f1 -> v2    // A is in f1
    stf.v {r2} v2 -> [Z+r1]
    ```
Scatter Stores & Gather Loads

• How to vectorize:
  
  for(int i = 1, i<N, i++) {
    int bucket = val[i] / scalefactor;
    found[bucket] = 1;
  }  
  
  Easy to vectorize the divide, but what about the load/store?

• Solution: hardware support for vector “scatter stores”
  
  o stf.v v2->[r1+v1]

  Each address calculated from r1+v1:
  stf v20->[r1+v10], stf v21->[r1+v11],
  stf v22->[r1+v12], stf v23->[r1+v13]

• Vector “gather loads” defined analogously
  
  o ldf.v [r1+v1]->v2

• Scatter/gathers slower than regular vector load/store ops
  
  Still provides throughput advantage over non-vector version