Lecture 21: Data Level Parallelism
-- SIMD ISA Extensions for Multimedia and Roofline Performance Model

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics for Data Level Parallelism (DLP)

- Parallelism (centered around … )
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

- DLP Introduction and Vector Architecture
  - 4.1, 4.2

- SIMD Instruction Set Extensions for Multimedia
  - 4.3

- Graphical Processing Units (GPU)
  - 4.4

- GPU and Loop-Level Parallelism and Others
  - 4.4, 4.5, 4.6, 4.7

Finish in three sessions
Acknowledge and Copyright

- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley
  - Arvind (MIT), Krste Asanovic (MIT/UCB), Joel Emer (Intel/MIT), James Hoe (CMU), John Kubiatowicz (UCB), and David Patterson (UCB)

- [https://passlab.github.io/CSE564/copyrightack.html](https://passlab.github.io/CSE564/copyrightack.html)
REVIEW
Broad classification of parallel computing systems

- based upon the number of concurrent **Instruction**
  (or control) streams and **Data** streams

- **SISD**: Single Instruction, Single Data
  - conventional uniprocessor

- **SIMD**: Single Instruction, Multiple Data
  - one instruction stream, multiple data paths
  - distributed memory SIMD (MPP, DAP, CM-1&2, Maspar)
  - shared memory SIMD (STARAN, vector computers)

- **MIMD**: Multiple Instruction, Multiple Data
  - message passing machines (Transputers, nCube, CM-5)
  - non-cache-coherent shared memory machines (BBN Butterfly, T3D)
  - cache-coherent shared memory machines (Sequent, Sun Starfire, SGI Origin)

- **MISD**: Multiple Instruction, Single Data
  - Not a practical configuration

Michael J. Flynn:
http://arith.stanford.edu/~flynn/
SIMD: Single Instruction, Multiple Data (Data Level Paralleism)

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation processing multiple data elements
  - Makes SIMD attractive for personal mobile devices

- SIMD allows programmer to continue to think sequentially
SIMD Parallelism

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

For x86 processors:
- Expect two additional cores per chip per year (MIMD)
- SIMD width to double every four years
- Potential speedup from SIMD to be twice that from MIMD!
Vector Programming Model

**Scalar Registers**
- r15
- r0

**Vector Registers**
- v15
- v0
- [0] [1] [2] [VLRMAX-1]

**Vector Length Register**
- VLR

**Vector Arithmetic Instructions**
- ADDV v3, v1, v2

**Vector Load and Store Instructions**
- LV v1, (r1, r2)
  - Base, r1
  - Stride in r2
  - Memory
# VMIPS Vector Instructions

## Suffix
- **VV suffix**
- **VS suffix**

## Load/Store
- **LV/SV**
- **LVWS/SVWS**

## Registers
- **VLR (vector length register)**
- **VM (vector mask)**

## Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV.D</td>
<td>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>ADDVS.D</td>
<td>V1, V2, F0</td>
<td>Add F0 to each element of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBBV.D</td>
<td>V1, V2, V3</td>
<td>Subtract elements of V3 from V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBVS.D</td>
<td>V1, V2, F0</td>
<td>Subtract F0 from elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBSV.D</td>
<td>V1, F0, V2</td>
<td>Subtract elements of V2 from F0, then put each result in V1.</td>
</tr>
<tr>
<td>MULV.D</td>
<td>V1, V2, V3</td>
<td>Multiply elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>MULVS.D</td>
<td>V1, V2, F0</td>
<td>Multiply each element of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV.D</td>
<td>V1, V2, V3</td>
<td>Divide elements of V2 by V3, then put each result in V1.</td>
</tr>
<tr>
<td>DIVVS.D</td>
<td>V1, V2, F0</td>
<td>Divide elements of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVSV.D</td>
<td>V1, F0, V2</td>
<td>Divide F0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
</tr>
<tr>
<td>SV</td>
<td>R1, V1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
</tr>
<tr>
<td>LVWS</td>
<td>V1, (R1, R2)</td>
<td>Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
</tr>
<tr>
<td>SVWS</td>
<td>(R1, R2), V1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).</td>
</tr>
<tr>
<td>LVI</td>
<td>V1, (R1+V2)</td>
<td>Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).</td>
</tr>
<tr>
<td>SVI</td>
<td>(R1+V2), V1</td>
<td>Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).</td>
</tr>
<tr>
<td>CVI</td>
<td>V1, R1</td>
<td>Create an index vector by storing the values 0, 1 × R1, 2 × R1, ..., 63 × R1 into V1.</td>
</tr>
<tr>
<td>S--VV.D</td>
<td>V1, V2</td>
<td>Compare the elements (E0, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.</td>
</tr>
<tr>
<td>S--VS.D</td>
<td>V1, F0</td>
<td>Compare the elements (E0, NE, GT, LT, GE, LE) in V1 and F0. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM).</td>
</tr>
<tr>
<td>POP</td>
<td>R1, VM</td>
<td>Count the 1s in vector-mask register VM and store count in R1.</td>
</tr>
<tr>
<td>CVM</td>
<td></td>
<td>Set the vector-mask register to all 1s.</td>
</tr>
<tr>
<td>MTC1</td>
<td>VLR, R1</td>
<td>Move contents of R1 to vector-length register VLR.</td>
</tr>
<tr>
<td>MFC1</td>
<td>R1, VLR</td>
<td>Move the contents of vector-length register VLR to R1.</td>
</tr>
<tr>
<td>MVTM</td>
<td>VM, F0</td>
<td>Move contents of F0 to vector-mask register VM.</td>
</tr>
<tr>
<td>MVM</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register VM to F0.</td>
</tr>
</tbody>
</table>

---

*Figure 4.3 The VMIPS vector instructions, showing only the double-precision floating-point operations. In addition to the vector registers, there are two special registers, VLR and VM, discussed below. These special registers are used for arithmetic and logical operations on vectors.*
AXPY (64 elements) \((Y = a \cdot X + Y)\) in MIPS and VMIPS

\[
\text{for (i=0; i<64; i++) }
\]
\[
Y[i] = a \cdot X[i] + Y[i];
\]

- # instrs:
  - 6 vs ~600

- Pipeline stalls
  - 64x higher by MIPS

- Vector chaining (forwarding)
  - V1, V2, V3 and V4

The starting addresses of X and Y are in Rx and Ry, respectively

```
Loop:
L.D F0,a ;load scalar a
DADDIU R4,Rx,#512 ;last address to load
L.D F2,0(Rx) ;load X[i]
MUL.D F2,F2,F0 ;a \times X[i]
L.D F4,0(Ry) ;load Y[i]
ADD.D F4,F4,F2 ;a \times X[i] + Y[i]
S.D F4,9(Ry) ;store into Y[i]
DADDIU Rx,Rx,#8 ;increment index to X
DADDIU Ry,Ry,#8 ;increment index to Y
DSUBU R20,R4,Rx ;compute bound
BNEZ R20,Loop ;check if done
L.D F0,a ;load scalar a
LV V1,Rx ;load vector X
MULVS.D V2,V1,F0 ;vector-scalar multiply
LV V3,Ry ;load vector Y
ADDVV.D V4,V2,V3 ;add
SV V4,Ry ;store the result
```
History: Supercomputers

Definition of a supercomputer:
- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing $30M+
- Any machine designed by Seymour Cray (originally)

CDC6600 (Cray, 1964) regarded as first supercomputer
- A vector machine

In 70s-80s, Supercomputer ≡ Vector Machine

www.cray.com: The Supercomputer Company

The Father of Supercomputing

Seymour Cray
Electrical engineer

Seymour Roger Cray was an American electrical engineer and supercomputer architect who designed a series of computers that were the fastest in the world for decades, and founded Cray Research which built many of these machines. Wikipedia

Born: September 28, 1925, Chippewa Falls, WI
Died: October 5, 1996, Colorado Springs, CO
Awards: Eckert–Mauchly Award
Parents: Seymour R. Cray, Lillian Cray
Education: University of Minnesota, Chippewa Falls High School
Fields: Applied mathematics, Computer Science, Electrical engineering

https://en.wikipedia.org/wiki/Seymour_Cray
http://www.cray.com/company/history/seymour-cray
Vector Instruction Execution with Pipelined Functional Units

**ADDV C,A,B**

**Execution using one pipelined functional unit**


C[2]  
C[1]  
C[0]  

**Execution using four pipelined functional units**

A[27] B[27]  

A[22] B[22]  


Lane
Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum length (serialized version before vectorization by compiler)

```c
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```
Vector Mask Registers

for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] – Y[i];

- Use vector mask register to “disable” elements (1 bit per element):

  LV    V1,Rx       ;load vector X into V1
  LV    V2,Ry       ;load vector Y
  L.D   F0,#0       ;load FP zero into F0
  SNEVS.D V1,F0     ;sets VM(i) to 1 if V1(i)! = F0
  SUBVV.D V1,V1,V2   ;subtract under vector mask
  SV    Rx,V1       ;store the result in X

- GFLOPS rate decreases!
  - Vector operation becomes bubble (“NOP”) at elements where mask bit is clear
Stride

DGEMM (Double-Precision Matrix Multiplication)
for (i = 0; i < 100; i+=1)
    for (j = 0; j < 100; j+=1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k+=1)
    }

- Must vectorize multiplication of rows of B with columns of D
  - Row-major: B: 1 double (8 bytes), and D: 100 doubles (800 bytes)
- Use non-unit stride
  - LDWS R3, (R1, R2) where R2 = 800
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time
Scatter-Gather

- Sparse matrix:
  - Non-zero values are compacted to a smaller value array (A[ ])
  - indirect array indexing, i.e. use an array to store the index to value array (K[ ])

\[
\text{for (i = 0; i < n; i=i+1)}
\]

\[
A[K[i]] = A[K[i]] + C[M[i]];
\]

- Use index vector:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV</td>
<td>Vk, Rk</td>
</tr>
<tr>
<td>LVI</td>
<td>Va, (Ra+Vk)</td>
</tr>
<tr>
<td>LV</td>
<td>Vm, Rm</td>
</tr>
<tr>
<td>LVI</td>
<td>Vc, (Rc+Vm)</td>
</tr>
<tr>
<td>ADDVV.D</td>
<td>Va, Va, Vc</td>
</tr>
<tr>
<td>SVI</td>
<td>(Ra+Vk), Va</td>
</tr>
</tbody>
</table>
SIMD INSTRUCTION SET EXTENSION FOR MULTIMEDIA
What is Multimedia

- Multimedia is a combination of text, graphic, sound, animation, and video that is delivered interactively to the user by electronic or digitally manipulated means.

<table>
<thead>
<tr>
<th>Medium</th>
<th>Elements</th>
<th>Time-dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Printable characters</td>
<td>No</td>
</tr>
<tr>
<td>Graphic</td>
<td>Vectors, regions</td>
<td>No</td>
</tr>
<tr>
<td><strong>Image</strong></td>
<td><strong>Pixels</strong></td>
<td>No</td>
</tr>
<tr>
<td>Audio</td>
<td>Sound, Volume</td>
<td>Yes</td>
</tr>
<tr>
<td>Video</td>
<td>Raster images, graphics</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Videos contains frame (images)

https://en.wikipedia.org/wiki/Multimedia
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Binary images**
  - Each pixel is either 0 or 1
Image Format and Processing

- **Pixels**
  - Images are matrix of pixels

- **Grayscale images**
  - Each pixel value normally range from 0 (black) to 255 (white)
  - 8 bits per pixel
Pixels
- Images are matrix of pixels

Color images
- Each pixel has three/four values (4 bits or 8 bits each) each representing a color scale
Image Processing

- Mathematical operations by using any form of signal processing
  - Changing pixel values by matrix operations

The above is repeated for every pixel in the original image to generate the smoothed image.
Image Processing: The major of the filter matrix

- [http://lodev.org/cgtutor/filtering.html](http://lodev.org/cgtutor/filtering.html)

**Smoothing Image (Gaussian blur method)**

The above is repeated for every pixel in the original image to generate the smoothed image.
Image Data Format and Processing for SIMD Architecture

- Data element
  - 4, 8, 16 bits (small)

- Same operations applied to every element (pixel)
  - Perfect for data-level parallelism

Can fit multiple pixels in a regular scalar register

- E.g. for 8 bit pixel, a 64-bit register can take 8 of them
Multimedia Extensions (aka SIMD extensions) to Scalar ISA

<table>
<thead>
<tr>
<th>64b</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b</td>
</tr>
<tr>
<td>16b</td>
</tr>
<tr>
<td>16b</td>
</tr>
</tbody>
</table>

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    » 128b for PowerPC Altivec, Intel SSE2/3/4
    » 256b for Intel AVX

- Single instruction operates on all elements within register

4x16b adds

16b + 16b + 16b + 16b + 16b + 16b + 16b + 16b
A Scalar FU to A Multi-Lane SIMD Unit

- **Adder**
  - Partitioning the carry chains

---

**Figure 4.8** Summary of typical SIMD multimedia support for 256-bit-wide operations. Note that the IEEE 754-2008 floating-point standard added half-precision (16-bit) and quad-precision (128-bit) floating-point operations.
MMX SIMD Extensions to X86

- MMX instructions added in 1996
  - Repurposed the 64-bit floating-point registers to perform 8 8-bit operations or 4 16-bit operations simultaneously.
  - MMX reused the floating-point data transfer instructions to access memory.
  - Parallel MAX and MIN operations, a wide variety of masking and conditional instructions, DSP operations, etc.

- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler
MMX Instructions

- **Move 32b, 64b**
- **Add, Subtract in parallel: 8 8b, 4 16b, 2 32b**
  - opt. signed/unsigned saturate (set to max) if overflow
- **Shifts (sll, srl, sra), And, And Not, Or, Xor**
  - in parallel: 8 8b, 4 16b, 2 32b
- **Multiply, Multiply-Add in parallel: 4 16b**
- **Compare = , > in parallel: 8 8b, 4 16b, 2 32b**
  - sets field to 0s (false) or 1s (true); removes branches
- **Pack/Unpack**
  - Convert 32b<---> 16b, 16b <--> 8b
  - Pack saturates (set to max) if number is too large
SSE/SSE2/SSE3 SIMD Extensions to X86

- Streaming SIMD Extensions (SSE) successor in 1999
  - Added separate 128-bit registers that were 128 bits wide
    » 16 8-bit operations, 8 16-bit operations, or 4 32-bit operations.
    » Also perform parallel single-precision FP arithmetic.
  - Separate data transfer instructions.
    » increased the peak FP performance of the x86 computers.
  - Each generation also added ad hoc instructions to accelerate specific multimedia functions.
AVX SIMD Extensions for X86

- Advanced Vector Extensions (AVX), added in 2010
- Doubles the width of the registers to 256 bits
  - double the number of operations on all narrower data types. Figure 4.9 shows AVX instructions useful for double-precision floating-point computations.
- AVX includes preparations to extend to 512 or 1024 bits in future generations of the architecture.

<table>
<thead>
<tr>
<th>AVX Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADDPD</td>
<td>Add four packed double-precision operands</td>
</tr>
<tr>
<td>VSUBPD</td>
<td>Subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VMULPD</td>
<td>Multiply four packed double-precision operands</td>
</tr>
<tr>
<td>VDIVPD</td>
<td>Divide four packed double-precision operands</td>
</tr>
<tr>
<td>VFMAADDPD</td>
<td>Multiply and add four packed double-precision operands</td>
</tr>
<tr>
<td>VFMSUBPD</td>
<td>Multiply and subtract four packed double-precision operands</td>
</tr>
<tr>
<td>VCMPxx</td>
<td>Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...</td>
</tr>
<tr>
<td>VMOVAPD</td>
<td>Move aligned four packed double-precision operands</td>
</tr>
<tr>
<td>VBroadCastSD</td>
<td>Broadcast one double-precision operand to four locations in a 256-bit register</td>
</tr>
</tbody>
</table>
### AXPY

for (i=0; i<64; i++)
\[
Y[i] = a \times X[i] + Y[i];
\]

- 256-bit SIMD exts
  - 4 double FP

- MIPS: 578 insts
- SIMD MIPS: 149
  - 4× reduction
- VMIPS: 6 instrs
  - 100× reduction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,a</td>
<td>Load scalar a</td>
</tr>
<tr>
<td>DADDIU R4,Rx,#512</td>
<td>Last address to load</td>
</tr>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>L.D F2,0(Rx)</td>
<td>Load X[i]</td>
</tr>
<tr>
<td>MUL.D F2,F2,F0</td>
<td></td>
</tr>
<tr>
<td>L.D F0,a</td>
<td>Load scalar a</td>
</tr>
<tr>
<td>LV V1,Rx</td>
<td>Load vector X</td>
</tr>
<tr>
<td>MULVS.D V2,V1,F0</td>
<td>Vector-scalar multiply</td>
</tr>
<tr>
<td>LV V3,Ry</td>
<td>Load vector Y</td>
</tr>
<tr>
<td>ADDVV.D V4,V2,V3</td>
<td>Add</td>
</tr>
<tr>
<td>SV V4,Ry</td>
<td>Store the result</td>
</tr>
</tbody>
</table>

```
for (i=0; i<64; i++)
Y[i] = a* X[i] + Y[i];
```

```
L.D F0,a ;load scalar a
DADDIU R4,Rx,#512 ;last address to load
LV V1,Rx ;load vector X
MULVS.D V2,V1,F0 ;vector-scalar multiply
LV V3,Ry ;load vector Y
ADDVV.D V4,V2,V3 ;add
SV V4,Ry ;store the result
```

```
L.D F0,a ;load scalar a
MOV F1, F0 ;copy a into F1 for SIMD MUL
MOV F2, F0 ;copy a into F2 for SIMD MUL
MOV F3, F0 ;copy a into F3 for SIMD MUL
DADDIU R4,Rx,#512 ;last address to load
Loop:
L.4D F4,0(Rx) ;load X[i], X[i+1], X[i+2], X[i+3]
MUL.4D F4,F4,F0 ;a×X[i], a×X[i+1], a×X[i+2], a×X[i+3]
L.4D F8,0(Ry) ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
ADD.4D F8,F8,F4 ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
S.4D F8,0(Rx) ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
DADDIU Rx,Rx,#32 ;increment index to Y
DADDIU Ry,Ry,#32 ;increment index to X
DSUBU R20,R4,Rx ;compute bound
BNEZ R20,Loop ;check if done
```
Multimedia Extensions versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary

- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure

- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)
Programming Multimedia SIMD Architectures

- The easiest way to use these instructions has been through libraries or by writing in assembly language.
  - The ad hoc nature of the SIMD multimedia extensions,

- Recent extensions have become more regular
  - Compilers are starting to produce SIMD instructions automatically.
    » Advanced compilers today can generate SIMD FP instructions to deliver much higher performance for scientific codes.
    » Memory alignment is still an important factor for performance
Why are Multimedia SIMD Extensions so popular

- Cost little to add to the standard arithmetic unit and they were easy to implement.
- Require little extra state compared to vector architectures, which is always a concern for context switch times.
- Does not require a lot of memory bandwidth to support as what a vector architecture requires.
- Others regarding to the virtual memory and cache that make SIMD extensions less challenging than vector architecture.

The state of the art is that we are putting a full or advanced vector capability to multi/manycore CPUs, and Manycore GPUs
State of the Art: Intel Xeon Phi Manycore Vector Capability

- Intel Xeon Phi Knight Corner, 2012, ~60 cores, 4-way SMT
- Intel Xeon Phi Knight Landing, 2016, ~60 cores, 4-way SMT and HBM


#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel
{
  #pragma omp simd safelen(18)
  for (int i=0; i<N; i++) {
    #pragma omp simd
    for (int j=18; j<N-18; j++) {
      x[i][j] = x[i][j-18] + sinf(y[i][j]);
      y[i][j] = y[i][j+18] + cosf(x[i][j]);
    }
  }
}

http://primeurmagazine.com/repository/PrimeurMagazine-AE-PR-12-14-32.pdf
State of the Art: ARM Scalable Vector Extensions (SVE)

- Announced in August 2016
  - [https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture](https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture)

- Beyond vector architecture we learned
  - Vector loop, predict and speculation
  - Vector Length Agnostic (VLA) programming

  - Check the slide
The Roofline Visual Performance Model

- Self-study: two pages of text
  - You need it for some question in assignment 4

- More materials:
  - Slides: https://crd.lbl.gov/assets/pubs_presos/parlab08-roofline-talk.pdf
  - Paper: https://people.eecs.berkeley.edu/~waterman/papers/roofline.pdf
  - Website: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/