# Intel Technologies for High Performance Computing Applications


Andrey Semin

Principal Engineer
Software and Services Group


**September 7, 2016**

To Compete, You Must Compute!*

* Susan Baldwin, Executive Director of Compute Canada

# Legal Information

# Agenda

- Demand for high performance computing
- Intel computing architectures for HPC
- Cores: pipelines, execution units
- AVX-512 overview

# The Three Pillars of Modern Science, Research & Engineering

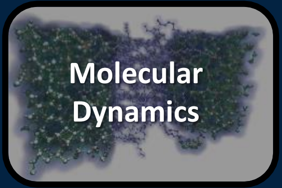
Experiment, Observation

Theory

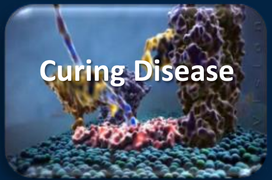
Numerical Simulation

# High Performance Computing: A Fundamental Tool for Breakthroughs

*Government & Academia*





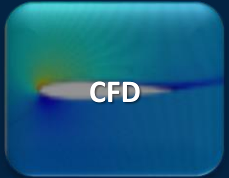




**Discovery**

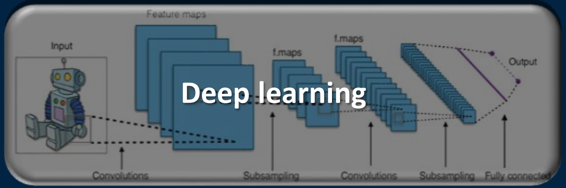
*Commercial/Industrial*

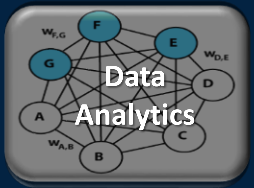





**Business Transformation**

*New Users – New Uses*



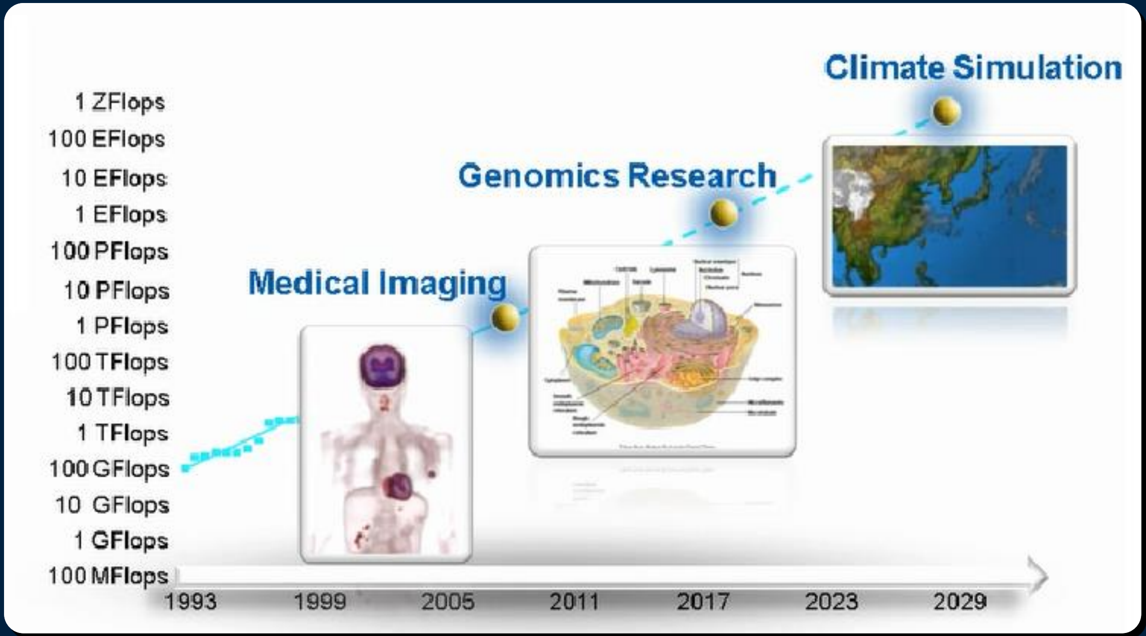





**Making insights**

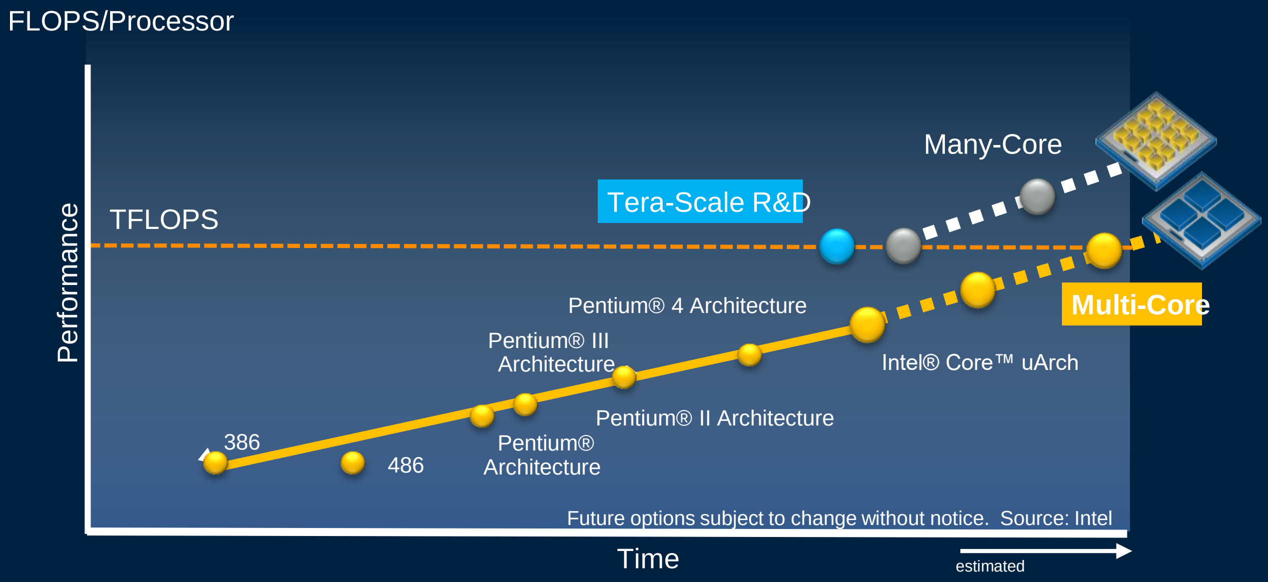
**To Compete You Must Compute**

# Need for Speed



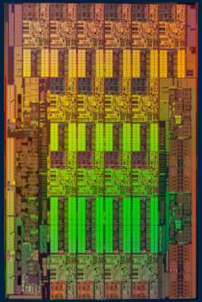

Source: www.top500.org

# Increasing Processor Performance

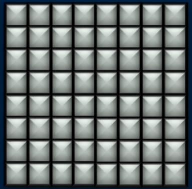


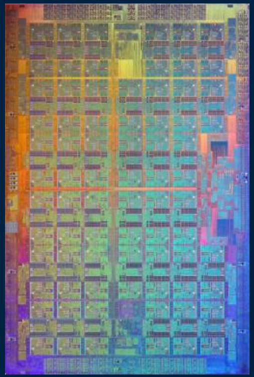

For illustration only, not drawn to scale. All dates, product descriptions, features, availability, and plans are forecasts and subject to change without notice.

# „Big Core“ – „Small Core“



Different Optimization Points
Common Programming Models
and Architectural Elements





## Intel® Xeon® Processor

Simply aggregating more cores generation after generation is not sufficient

Performance per core/thread must increase each generation, be as fast as possible

Power envelopes should stay flat or go down each generation

Balanced platform (Memory, I/O, Compute)

**Cores, Threads, Caches, SIMD**

## Intel® Xeon Phi™ Processor

Optimized for highest compute per watt

Willing to trade performance per core/thread for aggregate performance

Power envelopes should also stay flat or go down every generation

Optimized for highly parallel workloads

**Cores, Threads, Caches, SIMD**

For illustration only

# Parallel is the Path Forward

## Intel® Xeon® and Intel® Xeon Phi™ Product Families are both going parallel

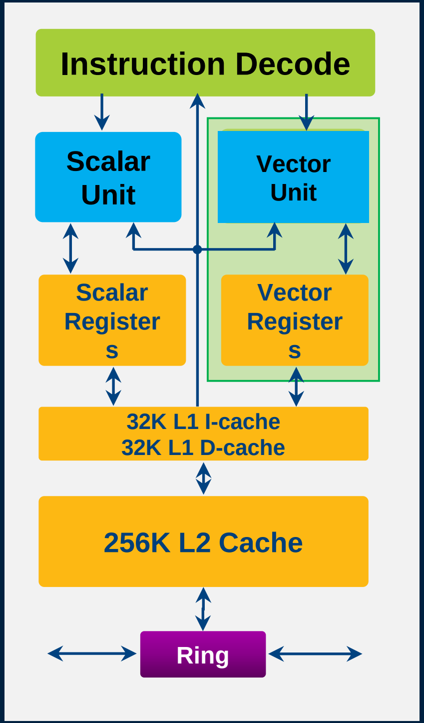
| | Intel® Xeon® processor 5100 series | Intel® Xeon® processor 5500 series | Intel® Xeon® processor 5600 series | Intel® Xeon® E5-2600 processor code-named Sandy Bridge EP | Intel® Xeon® E5-2600 v2 processor code-named Ivy Bridge EP | Intel® Xeon® E5-2600 v3 processor code-named Haswell EP | Intel® Xeon® E5-2600 v4 processor code-named Broadwell EP | Intel® Xeon Phi™ coprocessor code-named Knights Corner | Intel® Xeon Phi™ processor code-named Knights Landing |
|---|---|---|---|---|---|---|---|---|---|
| Core(s) up to | 2 | 4 | 6 | 8 | 12 | 18 | 22 | 61 | 72 |
| Threads up to | 2 | 8 | 12 | 16 | 24 | 36 | 44 | 244 | 288 |
| SIMD Width (bits) | 128 | 128 | 128 | 256 | 256 | 256 | 256 | 512 | 512 |

## More Cores → More Threads → Wider Vectors

Potential future options subject to change without notice. Codenames.
All timeframes, features, products and dates are preliminary forecasts and subject to change without further notification.
Product specification for launched and shipped products available on ark.intel.com.

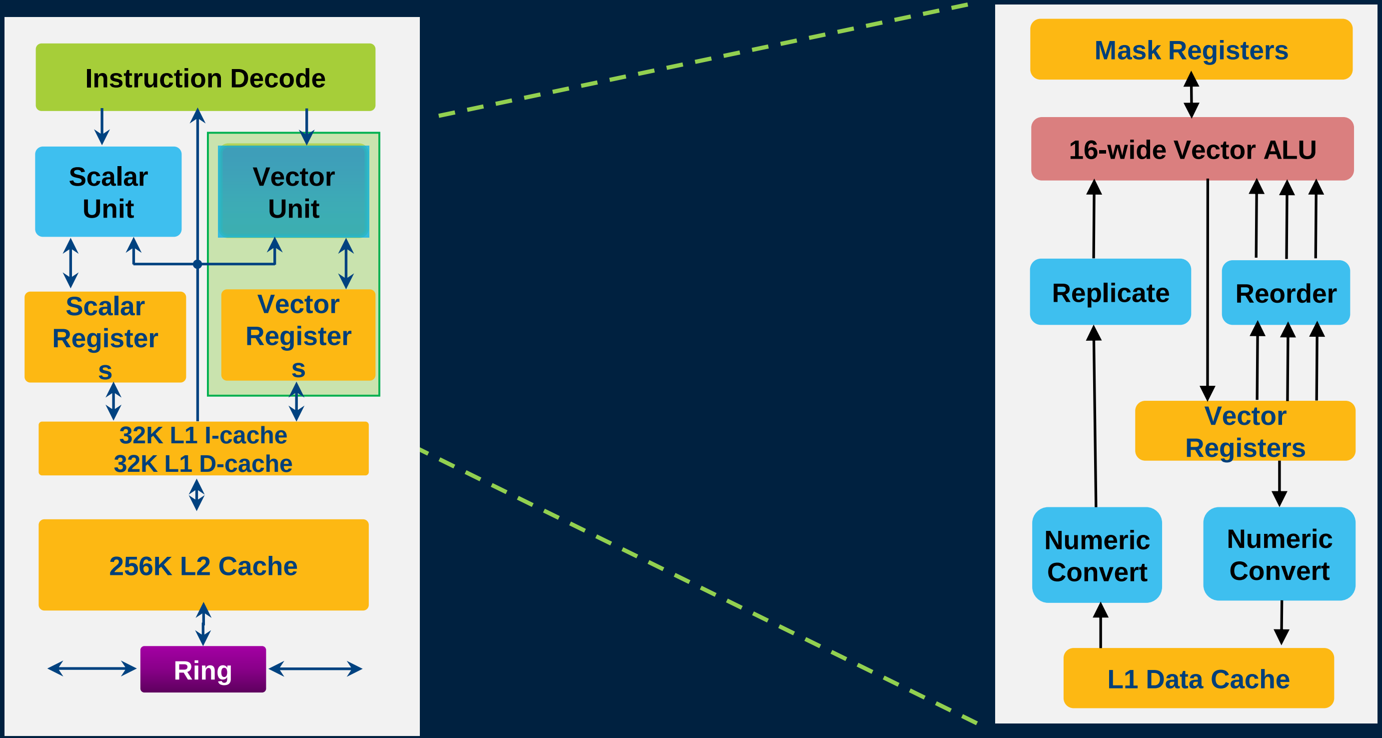
(die sizes not to scale, for illustration only)

# Knights Corner Architecture Overview
## Features of an Individual Core




- Up to 61 in-order cores
- 4 hardware threads per core
- Two pipelines
  - Pentium® processor family-based scalar units
  - Fully-coherent L1 and L2 caches
  - 64-bit addressing
- All new vector unit
  - 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
  - 32x 512-bit wide vector registers
    - Hold 16 singles or 8 doubles per register
  - Pipelined one-per-clock throughput
    - 4 clock latency, hidden by round-robin scheduling of threads
  - Dual issue with scalar instructions

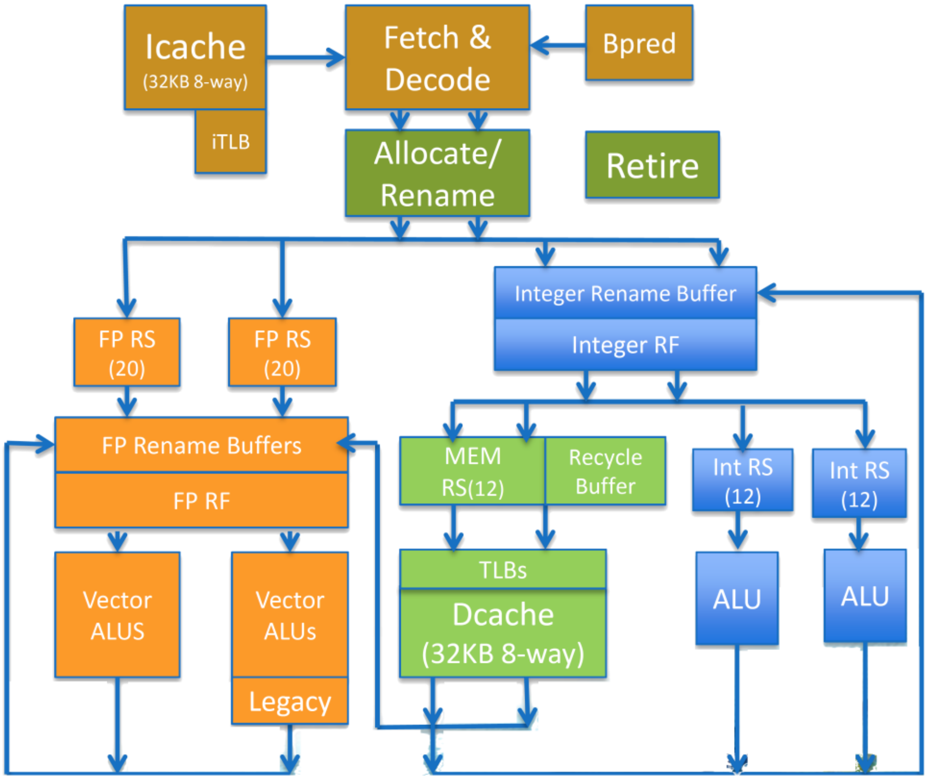
# Vector/SIMD High Computational Density



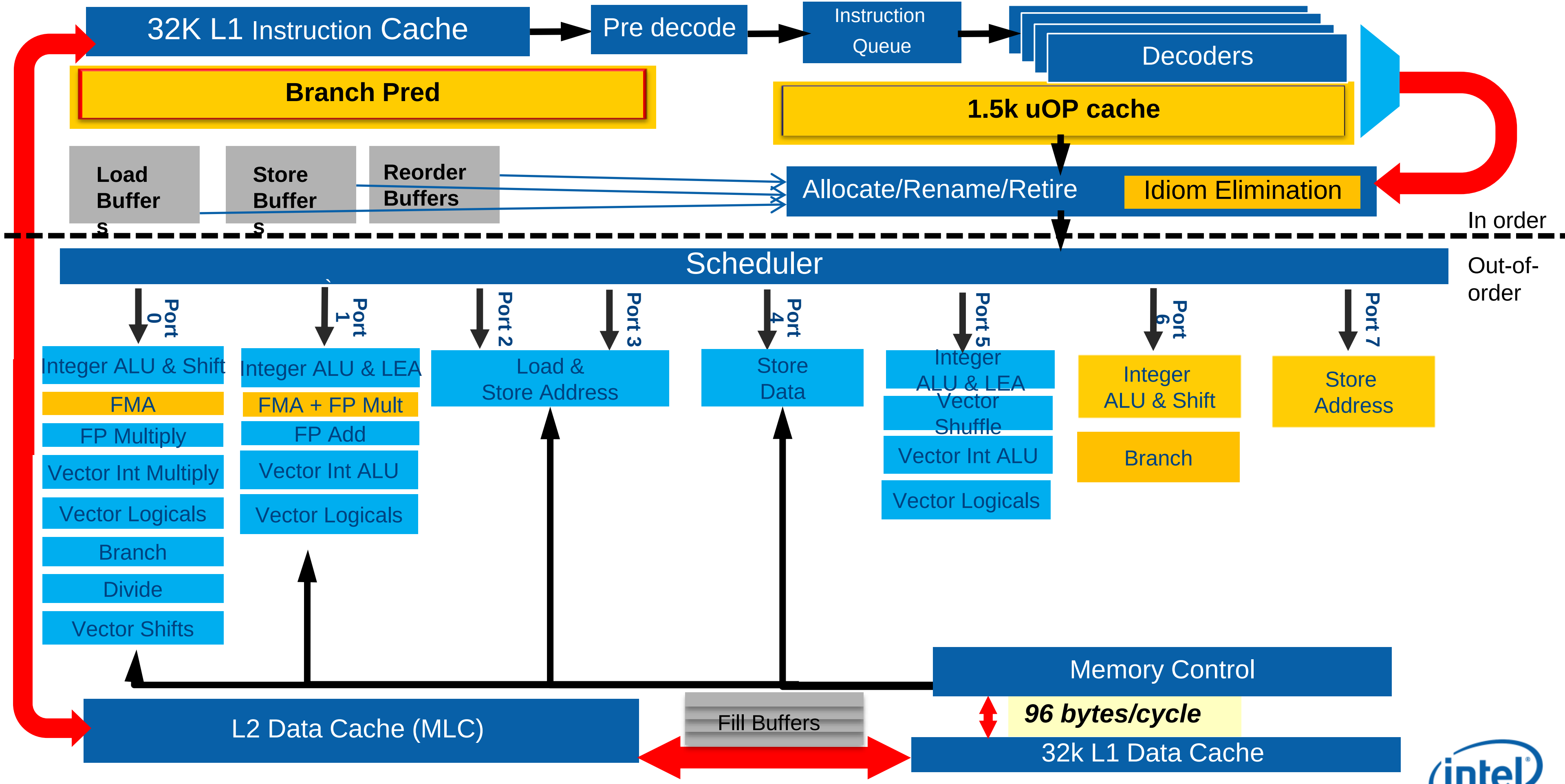

Vector/SIMD Unit

# Knights Landing Core & VPU

- Out-of-order core w/ 4 SMT threads: 3x over KNC
- VPU tightly integrated with core pipeline
- 2-wide Decode/Rename/Retire
- ROB-based renaming. 72-entry ROB & Rename Buffers
- Up to 6-wide at execution
- Integer (Int) and floating point (FP) RS are OoO
- MEM RS in-order with OoO completion - Recycle Buffer holds memory ops waiting for completion
- Int and MEM RS hold source data, FP RS does not
- 2x 64B Load & 1x 64B Store ports in Dcache
- 1st level uTLB: 64 entries
- 2nd level dTLB: 256 4K, 128 2M, 16 1G pages
- L1 Prefetcher (IPP) and L2 Prefetcher
- 46/48 PA/VA bits
- Fast unaligned and cache-line split support
- Fast Gather/Scatter support



# Haswell/Broadwell Core Microarchitecture

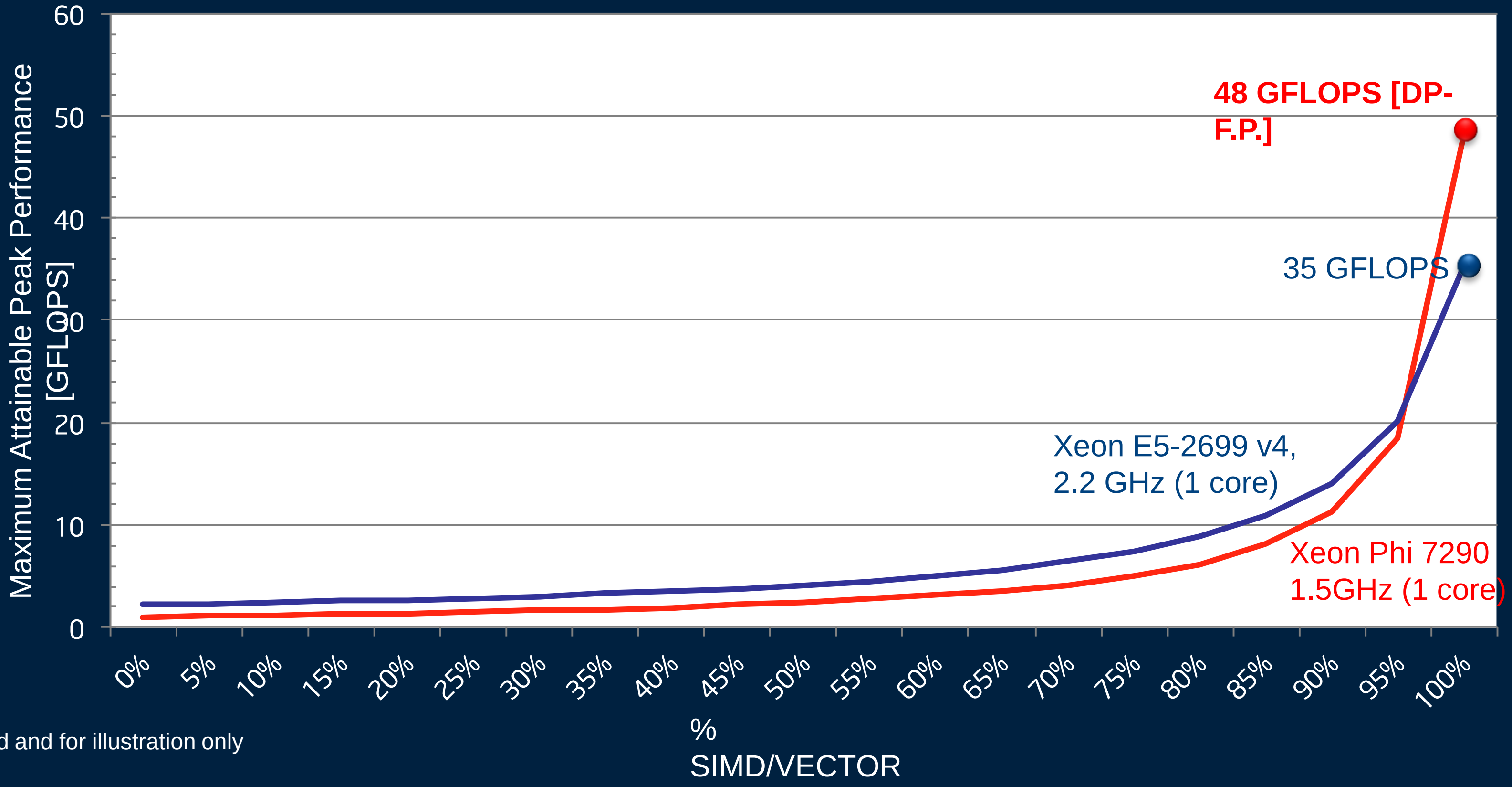



HSW - Intel® Next Generation Microarchitecture

AVX= Intel® Advanced Vector Extensions (Intel® AVX)

# The Effect of SIMD (Single Core)

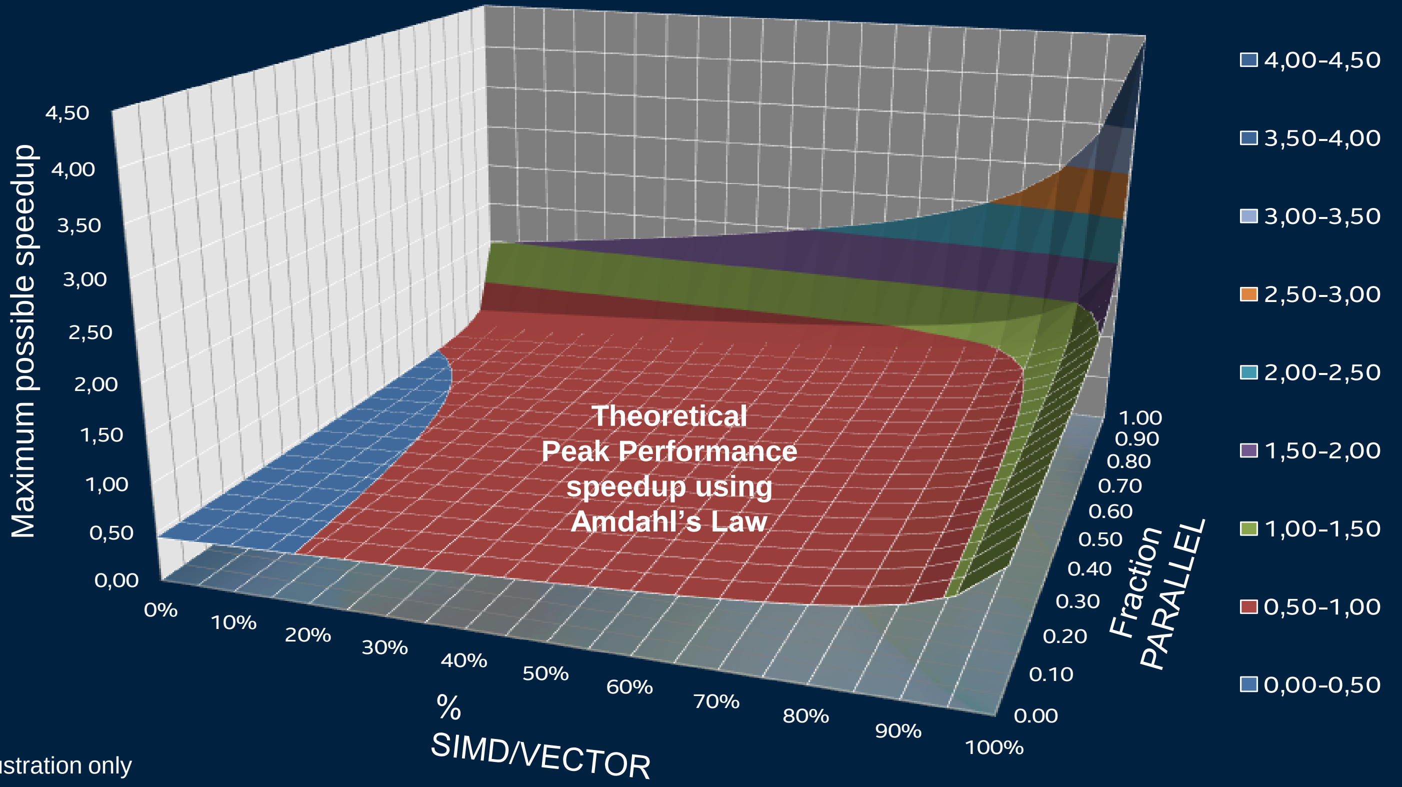
Based on Amdahl's Law




Simplified and for illustration only

# Maximum Theoretical Speedups

## 1 Xeon Phi 7290 vs. 2 socket Xeon E5-2699 v4 (2.2GHz, 22 cores)




Simplified and for illustration only

# Positioning of SIMD Features

| | |
|---|---|
| Fully automatic vectorization | Ease of use |
| Auto vectorization hints (#pragma ivdep) | ↑ |
| SIMD feature (#pragma omp simd and simd function annotation) | ↕ |
| SIMD intrinsic class (F32vec4 add) | ↕ |
| Vector intrinsic (_mm_add_ps()) | ↓ |
| ASM code (addps) | Programmer control |

# Sample: Manual Vectorization

```
void vecmul(float *a, float *b, float *c, int n)
{
    for (int k=0; k<n; k++)
        c[k] = a[k] * b[k];
}
```

- For the following slide we make the following assumptions (otherwise, we'd run out of space)
  - Input and output data is properly aligned to 64 bytes
  - Vector length is a multiple of the vector length
- If assumptions do not hold, add code to:
  - Peel off iterations 0..m to get rid of alignment issue
  - Have a vectorized loop to do the work
  - Peel off iterations n..N-1 to deal with remaining data

# Sample: Manual Vectorization

```
void vecmul(float *a, float *b, float *c, int n)
{
  __m512 va;
  __m512 vb;
  __m512 vc;
  for (int i = 0; i < a->size; i += 16, a += 16, b += 16, c += 16) {
    va = _mm512_loadd(a, _MM_FULLUPC_NONE, _MM_BROADCAST_16X16, _MM_HINT_NONE);
    vb = _mm512_loadd(b, _MM_FULLUPC_NONE, _MM_BROADCAST_16X16, _MM_HINT_NONE);
    vc = _mm512_mul_ps(va, vb);
    _mm512_stored(vc, c, _MM_DOWNC_NONE, _MM_SUBSET32_16, _MM_HINT_NONE);
  }
}
```

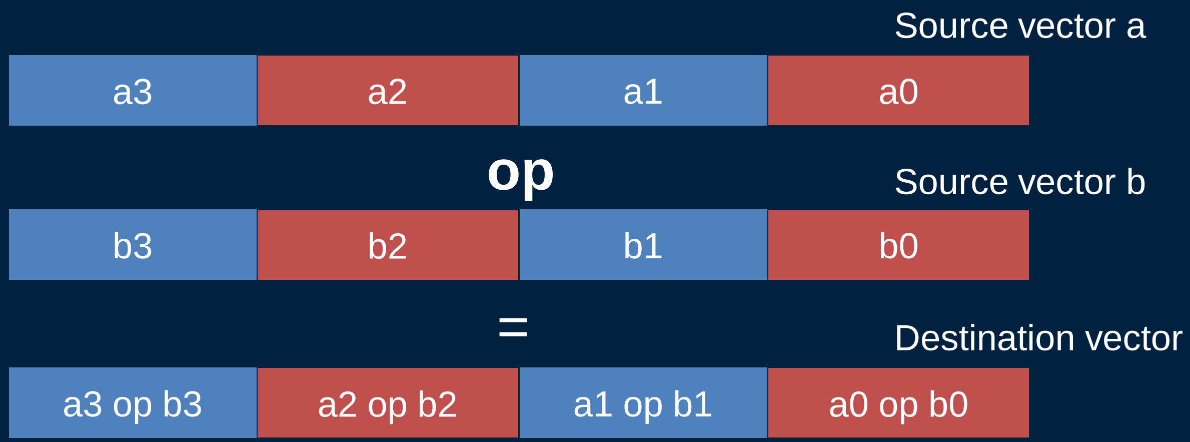
Vector registers

- Loop unrolling by 16 (i.e. vector length)
- Increment pointers

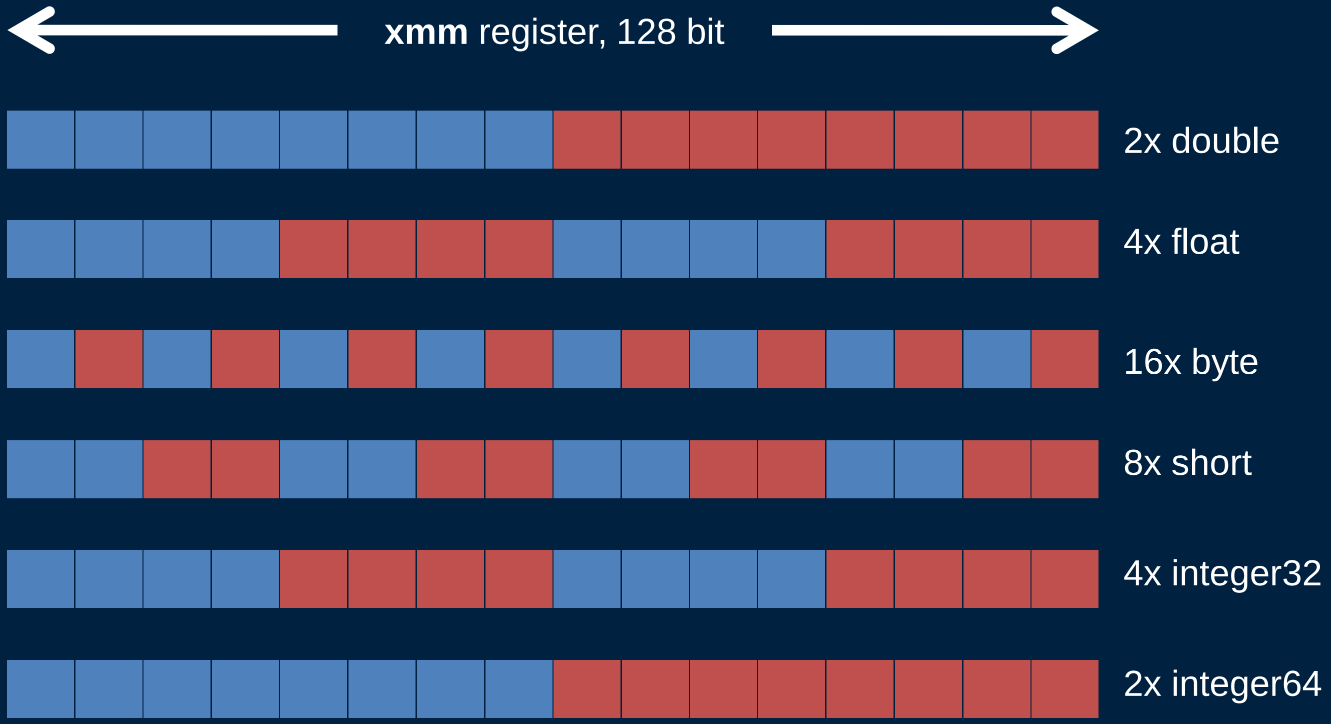
Vector instructions

# SIMD Instructions / Vectorization

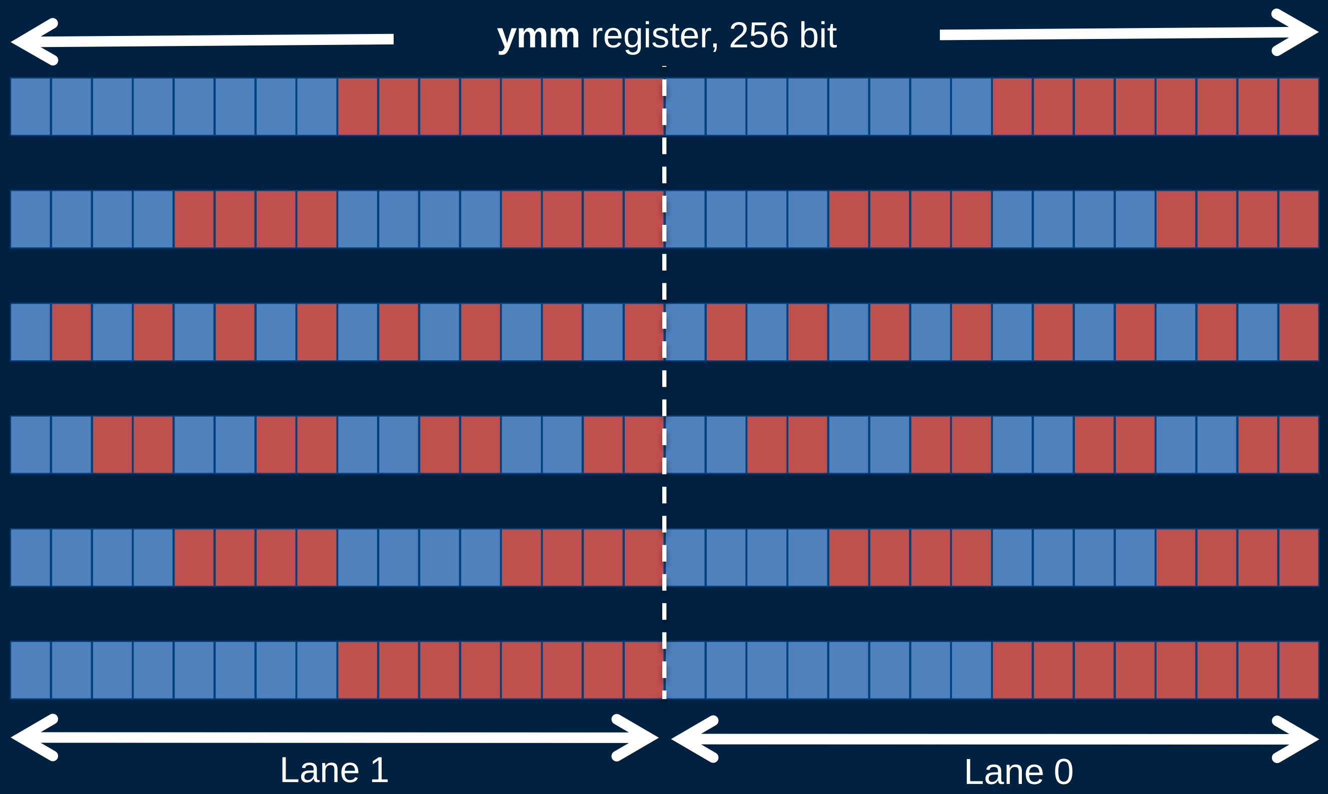
- SSE: Streaming SIMD extension
- SIMD: Single instruction, Multiple Data (Flynn's Taxonomy)
  - e.g., SSE allows the identical treatment of 2 double, 4 floats and 4 integers at the same time



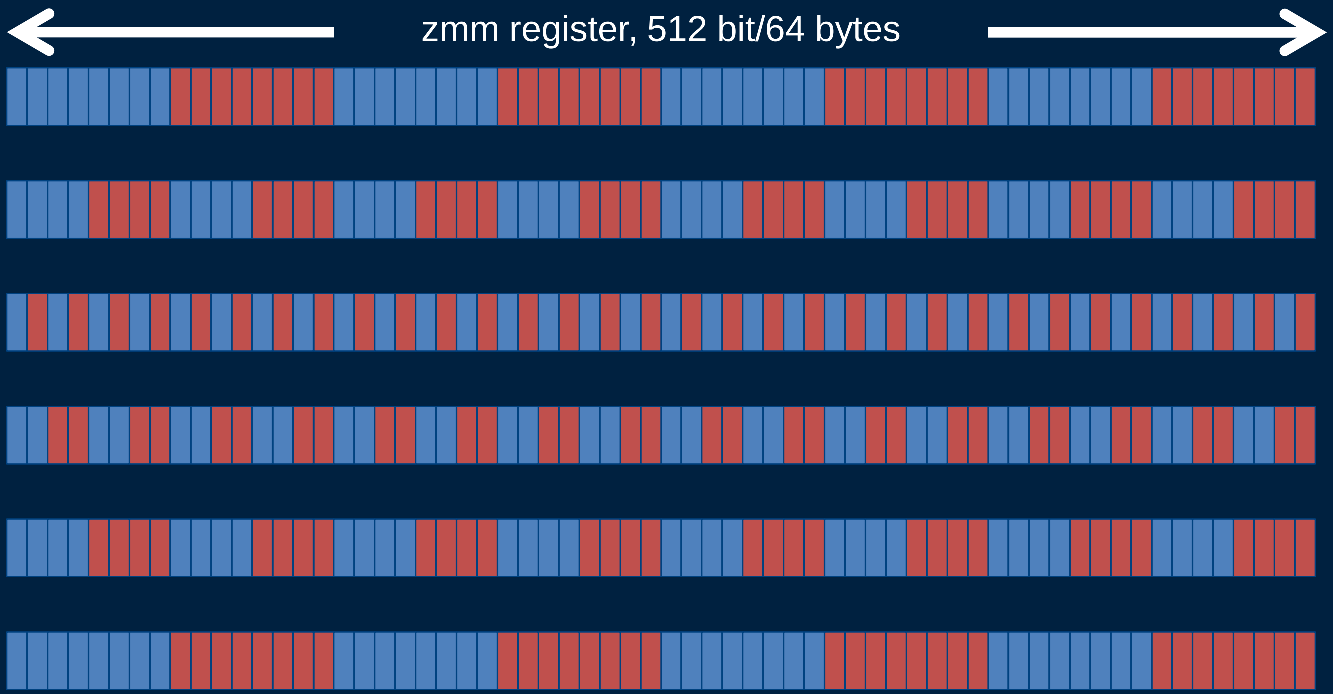
# Vectorization: SSE Data Types



# Evolution to Intel AVX

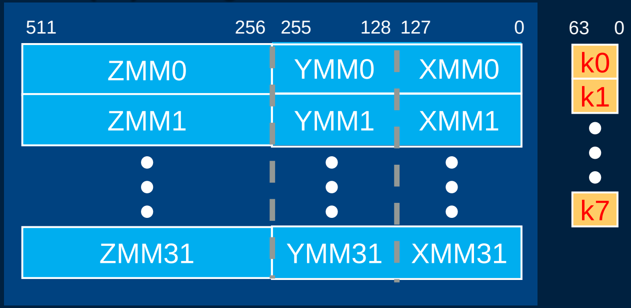


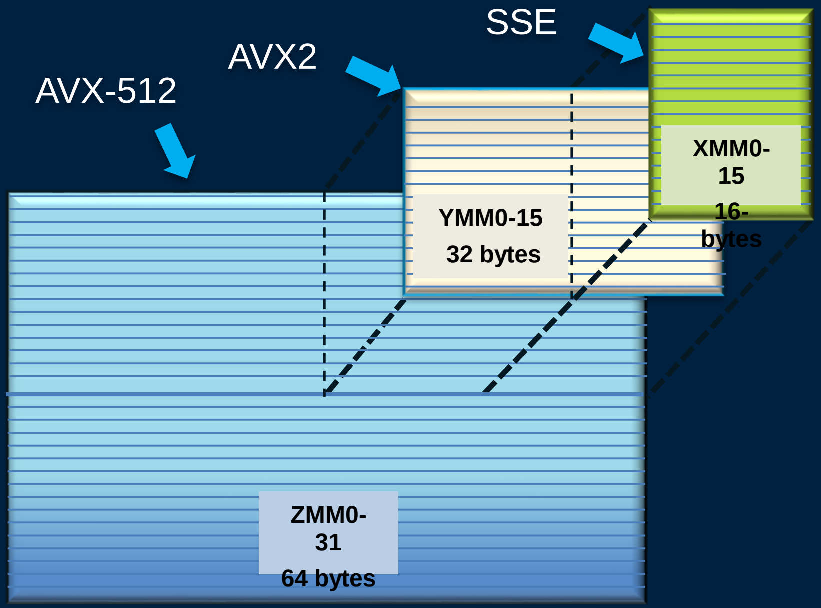
# SIMD Instructions on Intel MIC and AVX512



# AVX512 - Greatly increased register file

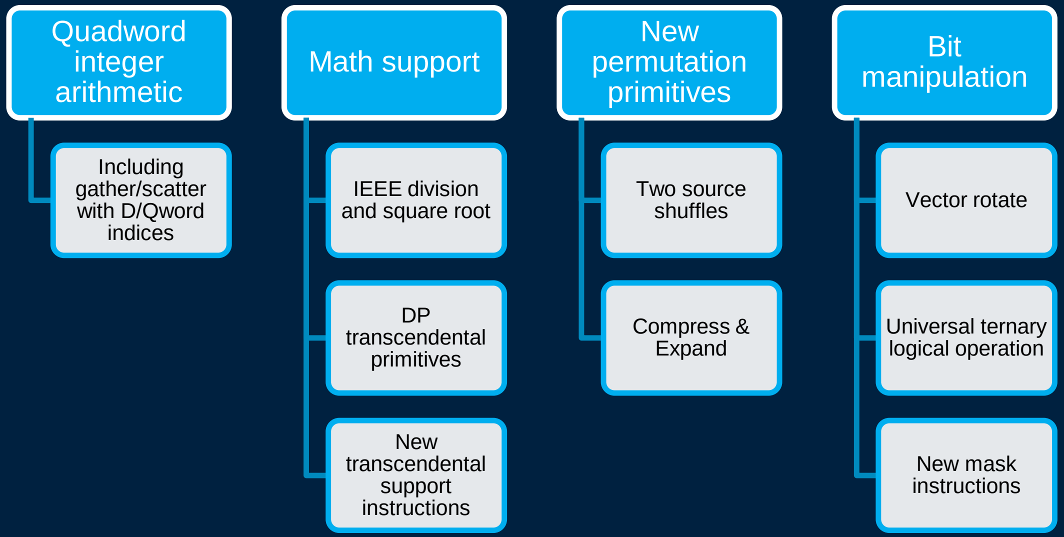
- Higher throughput
- Greatly improved unrolling and inlining opportunities
- 32 vector registers, 512b wide: zmm0 through zmm31
  - Overlaid on top of existing YMM arch state
  - Writing to xmm zeroes bits [511:128]
  - writing to ymm zeroes bits [511:256]
- 8 mask registers, 64b wide: k0 through k7
  - KNL only uses bits [15:0] though (PS,PD,D,Q)
  - EVEX.aaa=000 is an indicator of "no mask"
  - {k0} is illegal



# AVX-512 F Designed for HPC

- Promotions of many AVX and AVX2 instructions to AVX-512
  - 32-bit and 64-bit floating-point instructions from AVX
  - Scalar and 512-bit
  - 32-bit and 64-bit integer instructions from AVX2
- Many new instructions to speedup HPC workloads



# Wider data vector

**AVX2**

```
float A[N], B[N], C[N]

for(i=0; i<8; i++)
{
      C[i] = A[i] + B[i];
}
```

`VADDPS YMM0, YMM1, YMM2`

<u>16 x 256-bit registers</u>

In each register:

- 8 float or 4 double
- 8 integer or 4 long

**AVX-512**

```
float A[N], B[N], C[N]

for(i=0; i<16; i++)
{
      C[i] = A[i] + B[i];
}
```

`VADDPS ZMM0, ZMM1, ZMM2`

<u>32 x 512-bit registers</u>

In each register:

- 16 float or 8 double
- 16 integer or 8 long

# Masking – new feature in AVX

8 new mask registers k0-k7

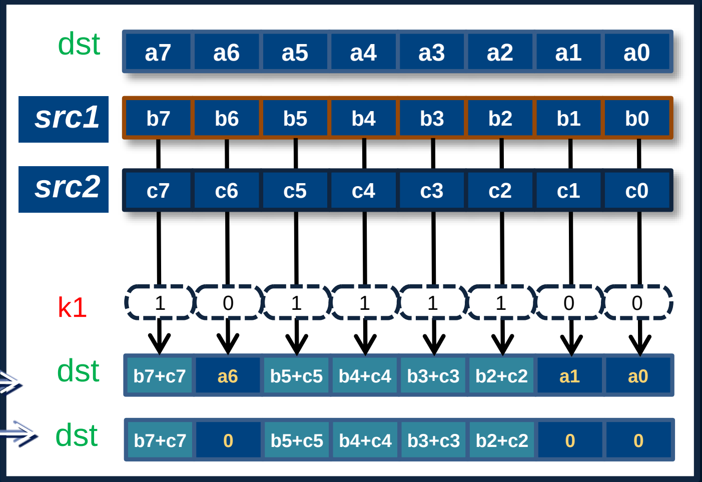
***Create mask:***

VCMPPS k1, zmm1, zmm2, imm
k1 = ..0101100111 /* 16 bits */

VCMPPD k1, zmm1, zmm2, imm
k1 = ..01011001 /* 8 bits */

***Use mask:***

VADDPD dst {k1}, src1, src2




**Unmasked elements remain unchanged:**
**VADDPD zmm1 {k1}, zmm2, zmm3**

**Or zeroed:**
**VADDPD zmm1 {k1} {z}, zmm2, zmm3**

# Why masking?

- Memory fault suppression
  - Vectorize code using masked load/store
  - Typical examples are if-conditional statements or loop remainders
- Avoid spurious floating-point exceptions
- Zeroing/merging
  - Use zeroing to avoid false dependencies **VADDPD zmm1 {k1} {z}, zmm2, zmm3**
  - Use merging to preserve unmasked values **VADDPD zmm1 {k1} , zmm2, zmm3**

```
float A[N], B[N], C[N];
for(i=0; i<16; i++)
{
  if (B[i] != 0)
    A[i] = A[i] / B[i];
  else
    A[i] = A[i] / C[i];
}
```

```
VMOVUPS zmm2, A[16]
VCMPPS k1, zmm0, B
VDIVPS  zmm1 {k1}{z}, zmm2, B
KNOT k2, k1
VDIVPS  zmm1 {k2}, zmm2, C
VMOVUPS A[16], zmm1
```

# Why Separate Mask Registers?

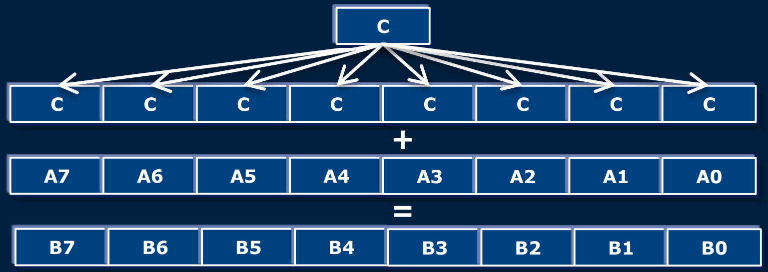
- Don’t waste away real vector registers for vector of booleans
- Separate control flow from data flow
- Boolean operations on logical predicates consume less energy (separate functional unit)
- Tight encoding allows orthogonal operand
  - Every instruction now has an extra mask operand

# Embedded Broadcast

**Broadcast one scalar from memory into all vector elements**

```
long A[N], B[N], C

for(i=0; i<8; i++)
{
  if(A[i]!=0.0)
    B[i] = A[i] + C;
}
```

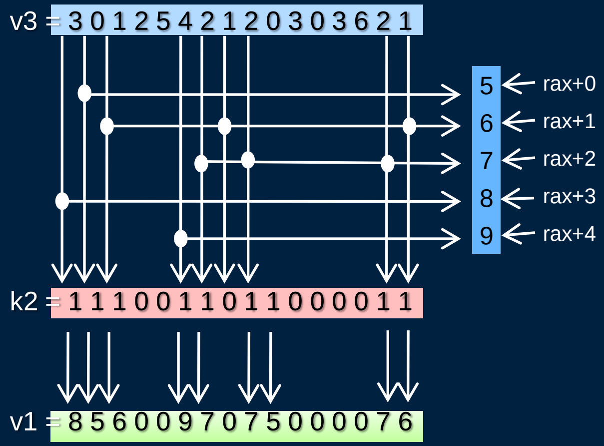



**VADDPS zmm1 {k1}, zmm2, C {1to16}**

- Scalars *from memory* are first class citizens
- Broadcast one scalar from memory into all vector elements before operation
- Memory fault suppression avoids fetching the scalar if no mask bit is set to 1

# VGATHER/VSCATTER Operation

vgather v1{k2},[rax+v3]




**vscatter [rax+v3]{k2}, v1**

**-- same as vgather, but in reverse**

# Embedded Rounding Control

- Set Rounding Control
  - AVX2 and before – access MXCSR.RC
  - Saving, modifying and restoring MXCSR is usually slow and cumbersome

```
STMXCSR [ESI]                     ;store the MXCSR into memory
MOV     EAX,[ESI]                 ;put into EAX
AND     AH,9Fh                    ;clear existing rounding bits (bits 13/14 of eax)
OR      AH,20h                    ;set rounding down
MOV     [ESI],EAX                 ;put back into memory
LDMXCSR [ESI]                     ;and put that into processor ;
```

- AVX-512 – define rounding control per instruction

```
VADDPS ZMM1 , ZMM2, ZMM3 {rne-sae}
```

  - "Suspend All Exceptions"
    - Always implied by using embedded RC
    - NO MXCSR updates / exception reporting for any lane

# Expand & Compress

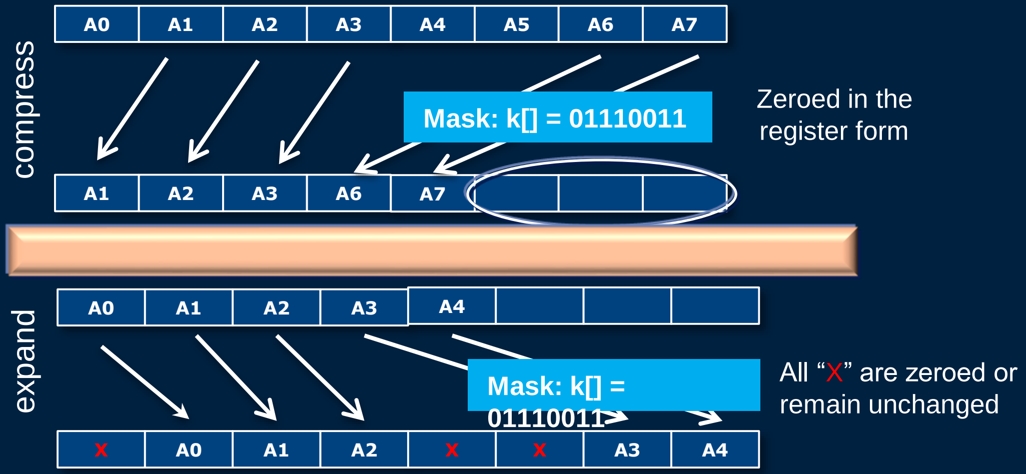
```
double A[N], B[N], C[N];
for(i=0; i<8; i++)
{
  if (B[i] != 0)

    *dst++ = A[i];
}
```

```
VMOVUPD zmm2, A[8]

VCMPPD k1, zmm0, B

VCOMPRESSPD [dst] {k1}, zmm2
```



# Quadword Integer Arithmetic

Useful for pointer manipulation
64-bit becomes a first class citizen
Removes the need for expensive SW emulation sequences

| Instruction | Description |
|---|---|
| VPADDQ zmm1 {k1}, zmm2, zmm3 | INT64 addition |
| VPSUBQ zmm1 {k1}, zmm2, zmm3 | INT64 subtraction |
| VP{SRA,SRL,SLL}Q zmm1 {k1}, zmm2, imm8 | INT64 shift (imm8) |
| VP{SRA,SRL,SLL}VQ zmm1 {k1}, zmm2, zmm3 | INT64 shift (variable) |
| VP{MAX,MIN}Q zmm1 {k1}, zmm2, zmm3 | INT64 max, min |
| VP{MAX,MIN}UQ zmm1 {k1}, zmm2, zmm3 | UINT64 max, min |
| VPABSQ zmm1 {k1}, zmm2, zmm3 | INT64 absolute value |
| VPMUL{DQ,UDQ} zmm1 {k1}, zmm2, zmm3 | 32x32 = 64 integer multiply |

*Note: VPMULQ and int64 <-> FP converts not in AVX-512 F*

# Math Support

Package to aid with Math library writing

- Good value upside in financial applications
- Available in PS, PD, SS and SD data types
- Great in combination with embedded RC

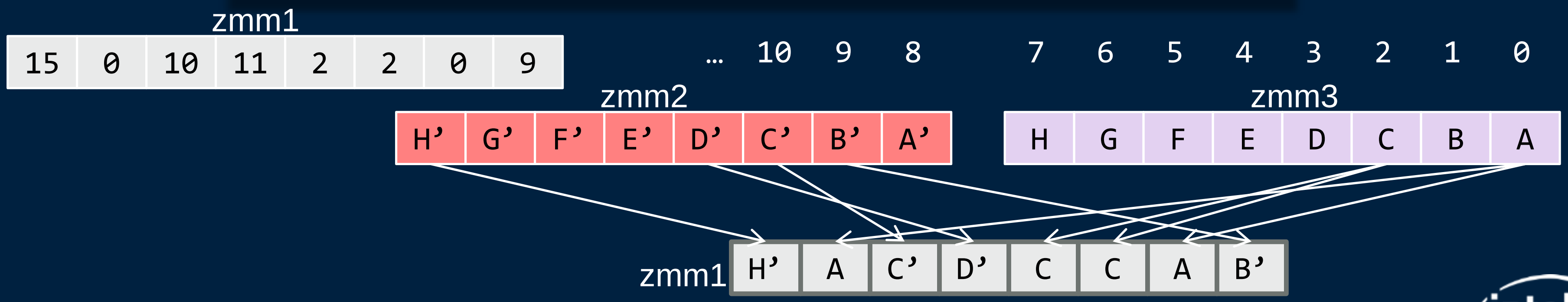
| Instruction | | |
|---|---|---|
| VGETXEXP$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2 | Obtain exponent in FP format |
| VGETMANT$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2 | Obtain normalized mantissa |
| VRNDSCALE$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2, imm8 | Round to scaled integral number |
| VSCALEF$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2, zmm3 | $X*2^{y}$, X <= getmant, Y <= getexp |
| VFIXUPIMM$_{\{PS,PD,SS,SD\}}$ | zmm1, zmm2, zmm3, imm8 | Patch output numbers based on inputs |
| VRCP14$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2 | Approx. reciprocal() with rel. error $2^{-14}$ |
| VRSQRT14$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2 | Approx. rsqrt() with rel. error $2^{-14}$ |
| VDIV$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2, zmm3 | IEEE division |
| VSQRT$_{\{PS,PD,SS,SD\}}$ | zmm1 {k1}, zmm2 | IEEE square root |

# New 2-Source Shuffles

| 2-Src Shuffles |
|---|
| VSHUF{PS,PD} |
| VPUNPCK{H,L}{DQ,QDQ} |
| VUNPCK{H,L}{PS,PD} |
| VPERM{I,D}2{D,Q,PS,PD} |
| VSHUF{F,I}32X4 |

Long standing customer request
- 16/32-entry table lookup (transcendental support)
- AOS ⇔ SOA support, matrix transpose
- Variable VALIGN emulation

| | | | | |
|---|---|---|---|---|
| EVEX.U1.512.NDS.66.0F38.W1 77 /r<br>VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/B64(mV) | A | V/V | AVX3.1 | Permute double-precision values on floating-point in zmm3/mV and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1. |



# Bit Manipulation

Basic bit manipulation operations on mask and vector operands
- Useful to manipulate mask registers
- Have uses in cryptography algorithms

| Instruction | Description |
|---|---|
| `KUNPCKBW k1, k2, k3` | Interleave bytes in k2 and k3 |
| `KSHIFT{L,R}W k1, k2, imm8` | Shift bits left/right using imm8 |
| `VPROR{D,Q} zmm1 {k1}, zmm2, imm8` | Rotate bits right using imm8 |
| `VPROL{D,Q} zmm1 {k1}, zmm2, imm8` | Rotate bits left using imm8 |
| `VPRORV{D,Q} zmm1 {k1}, zmm2, zmm3/mem` | Rotate bits right w/ variable ctrl |
| `VPROLV{D,Q} zmm1 {k1}, zmm2, zmm3/mem` | Rotate bits left w/ variable ctrl |

# VPTERNLOG – Ternary Logic Instruction

- Mimics a FPGA cell
  - Take every bit of three sources to obtain a 3-bit index N
  - Obtain Nth bit from imm8

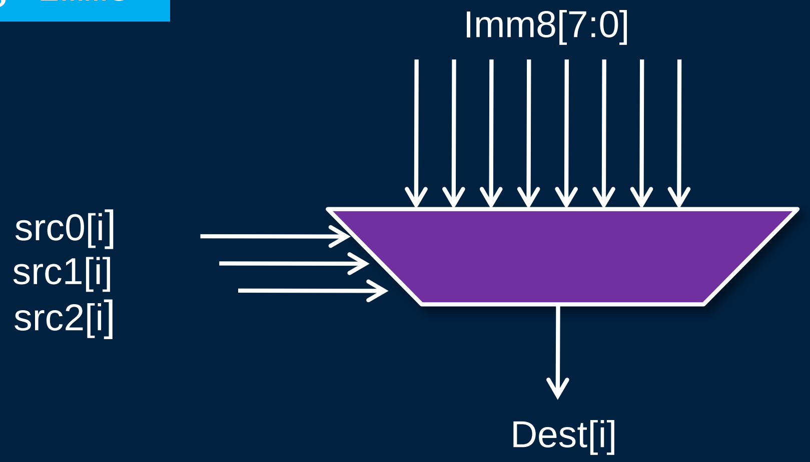
```
VPTERNLOGD  zmm0 {k2}, zmm15, zmm3/[rax], imm8
```

Any arbitrary truth table of 3 values can be implemented

***andor***, ***andxor***, ***vote***, ***parity***, bitwise-***cmov***, etc.

each column in the right table corresponds to imm8

| S1 | S2 | S3 | ANDOR | VOTE | (S1)?S3:S2 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |



# Motivation for Conflict Detection

- Sparse computations are common in HPC, but hard to vectorize due to race conditions
- Consider the “histogram” problem:

```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

```
index = vload &B[i]                 // Load 16 B[i]
old_val = vgather A, index          // Grab A[B[i]]
new_val = vadd old_val, +1.0        // Compute new values
vscatter A, index, new_val          // Update A[B[i]]
```

- Code above is wrong if any values within B[i] are duplicated
  - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts

# Conflict Detection – how does it work?

**Iteration 1**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| mask | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| indices | 9 | 3 | 2 | 2 | 2 | 7 | 8 | 7 |
| **conflict-free mask** | 1 | 1 | 1 | **0** | **0** | 1 | 1 | **0** |

**Iteration 2**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| mask | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| indices | 9 | 3 | 2 | 2 | 2 | 7 | 8 | 7 |
| **conflict-free mask** | **0** | **0** | **0** | 1 | **0** | **0** | **0** | 1 |

**Iteration 3**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| mask | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| indices | 9 | 3 | 2 | 2 | 2 | 7 | 8 | 7 |
| **conflict-free mask** | **0** | **0** | **0** | **0** | 1 | **0** | **0** | **0** |

# Conflict Free Code

```
for(i=0; i<16; i++)
{
  j = B[i];
  A[j]++;
}
```

| CDI instr. |
| --- |
| VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem |
| VPBROADCASTM{W2D,B2Q} zmm1, k2 |
| VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem |
| VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem |

```
j = vload &B[i]
pending_elts = 0xFFFF;
do {
  mask = conflict_free(j, pending_elts)
  val_A = vgather {mask} A, j          // Grab A[j]
  val_A++                              // Compute new values
  vscatter A {mask}, j, val_A          // Update A[j]
  pending_elts ^= mask                 // remove done idx
} while (pending_elts)
```

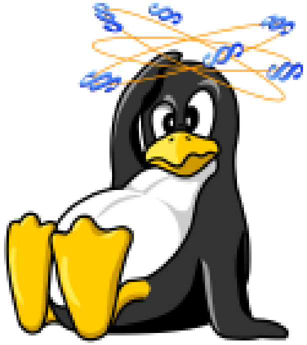
VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes

Allows to generate a mask with a subset of elements that are guaranteed to be conflict free

# Summary

- Continuous demand for high performance computing solution fuels innovation in architectures to address technical challenges
- Intel offers highly optimized architectures for HPC solutions
- AVX-512 is the greatest addition to x86 ISA family to drive continuous performance improvements

# Questions?



intel®