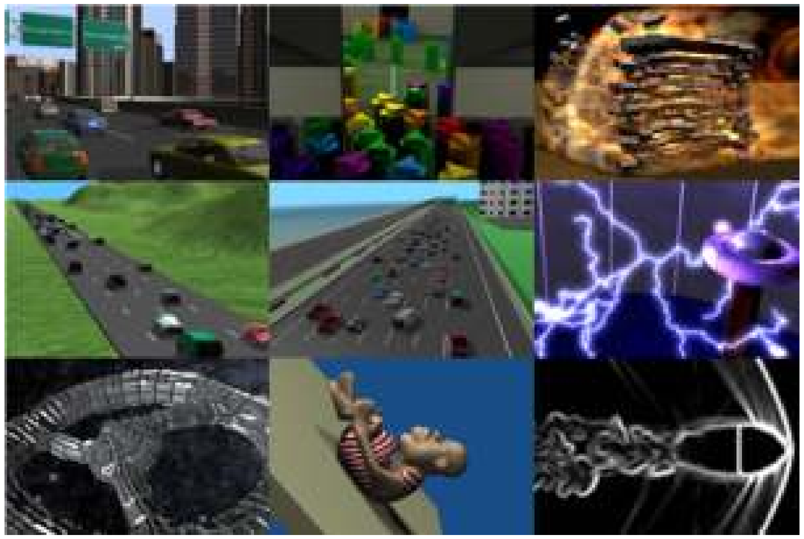
Preparing for a smooth landing: Intel’s Knights Landing and Modern Applications

Jason Sewall
TCG MICRO

Tutorial 1C
8:30-12pm, April 27
CUG 2015

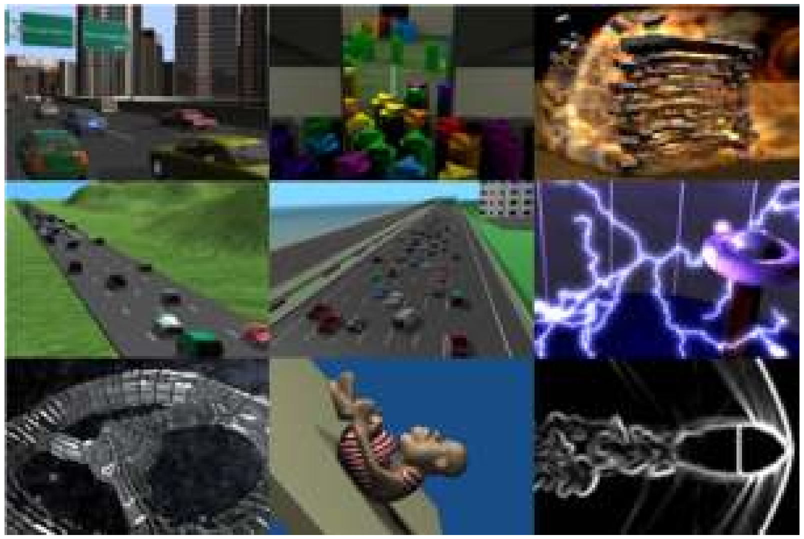
Yrs. truly

- Education
  - University of Maine
    - B.A. Mathematics ’04
    - B. S. Computer Science ’04
  - University of North Carolina – Chapel Hill
    - M.S. Computer Science, ‘08,
    - Ph.D. Computer Science ’10
    - Entered fall ‘04, advisor Ming C. Lin
    - ‘Graphics’ (physically-based animation)
    - GPGPU + Parallel computing



Yrs. truly

- Intel
  - Intern in ARL (now PCL) ‘06 & ’07
    - » Physics KNF: collision detection, fluid simulation
  - PCL full-time 9/2010
    - » Traffic simulation, CFD, database indexes, linear algebra, graphs, reservoir sim.
  - Joined DCG MICRO in 12/2013 (Virtual site in Pemaquid, ME)
    - » Training
    - » Optimization
    - » Ex officio PCL



Agenda

1. What is Knights Landing?
2. How do I get good performance on Knights Landing?
3. Break
4. What can I do *now* to get ready for Knights Landing?

What is Knights Landing?

The next generation of Intel® Xeon Phi ™

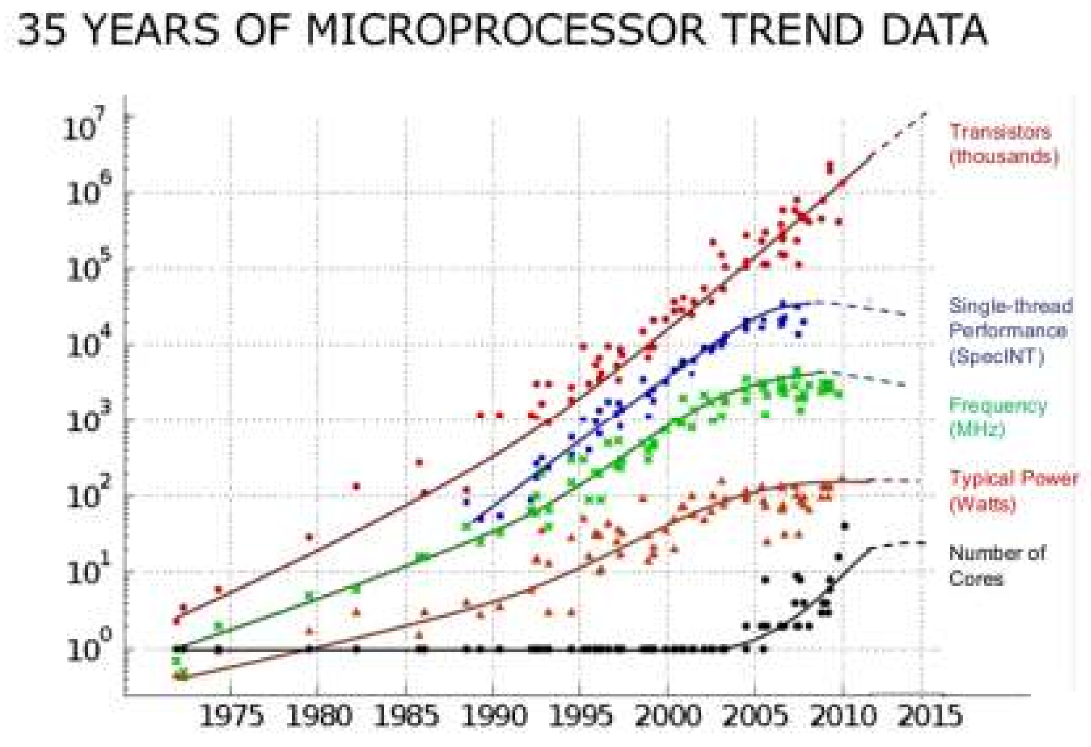
Increasing parallelism in Xeon and Xeon Phi

	Intel® Xeon® processor 64-bit series	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EX	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing[1]
Core(s)	1	2	4	6	8	12	18	61	*60+*
Threads	2	2	8	12	16	24	36	244	4x #cores
SIMD Width	128	128	128	128	256	256	256	512	*512*

**Product specification for launched and shipped products available on ark.intel.com.*

1. Not launched.

Moore's Law and Parallelism



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Parallelism and Performance

Peak GFLOP/s in Single Precision

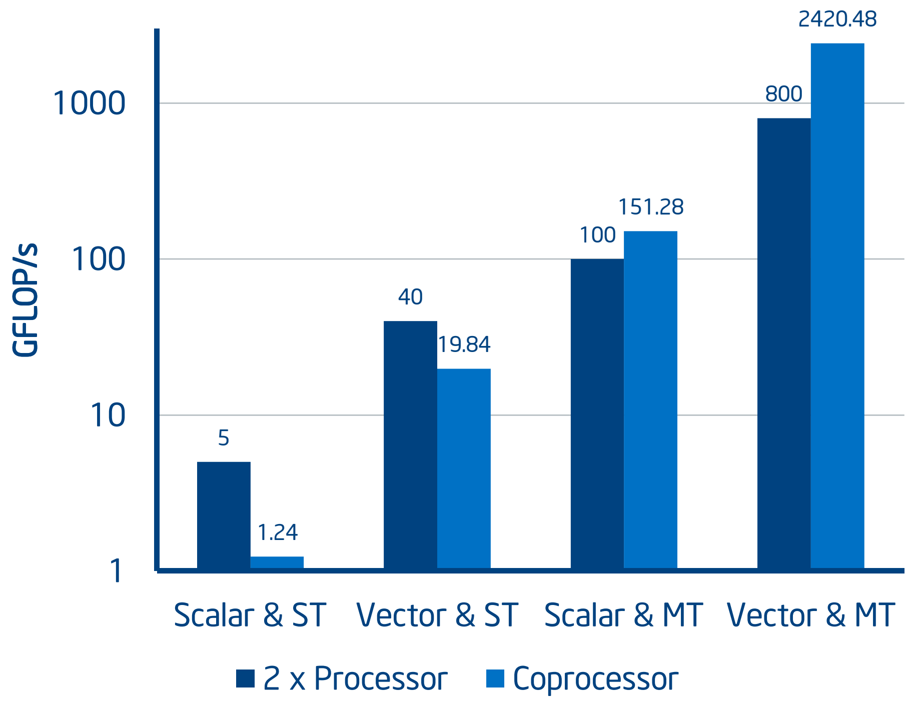
- Clock Rate x Cores x Ops/Cycle x SIMD

2 x Intel® Xeon® Processor E5-2670v2

- 2.5 GHz x 2 x 10 cores x 2 ops x 8 SIMD = 800 GFLOP/s

Intel® Xeon Phi™ Coprocessor 7120P

- 1.24 GHz x 61 cores x 2 ops x 16 SIMD = 2420.48 GFLOP/s



Note the logarithmic scale on the y-axis.
ST = Single Thread, MT = Multiple Threads

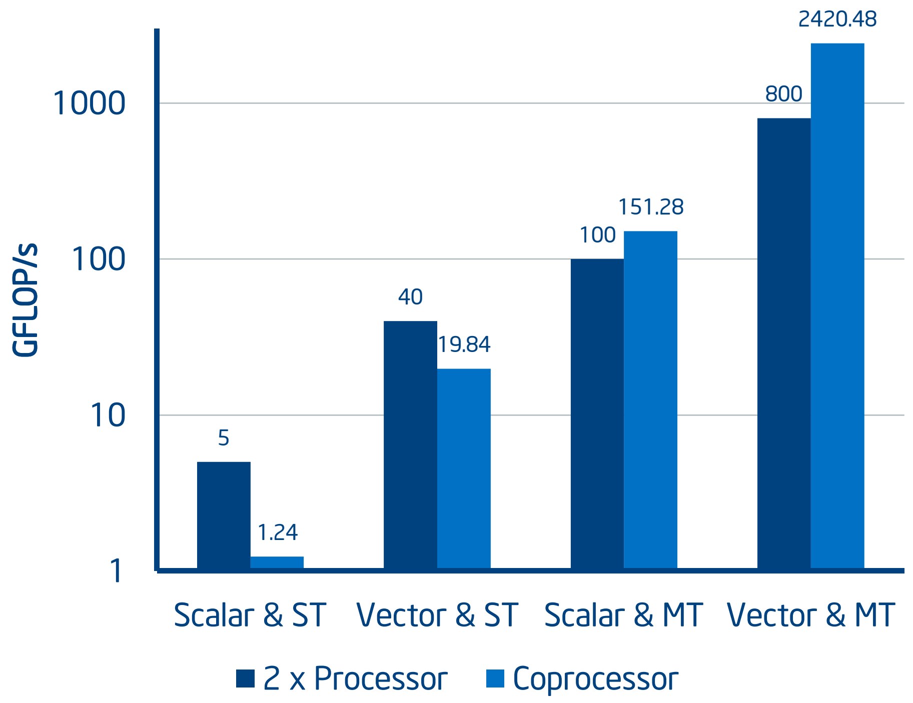
Parallelism and Performance

On modern hardware,
Performance = Parallelism

Flat programming model on parallel hardware is not effective.

Parallel programming is not optional.

Codes need to be made parallel (“modernized”) before they can be tuned for the hardware (“optimized”).



Note the logarithmic scale on the y-axis.
ST = Single Thread, MT = Multiple Threads

Parallel concepts

Parallel computing uses multiple computing units in parallel to

- Solve problems more quickly than a single processor ("strong scaling")
- Solve larger problems in the same time as a single processor ("weak scaling")
- Solve problems with higher fidelity

High-performance parallel computing is hard and requires

- Finding enough parallelism
- Deciding the optimal granularity, locality and load balance
- Coordination and synchronization

Real-world applications/algorithms are complex and often hierarchical: monolithic programming model is limited

Amdahl's Law

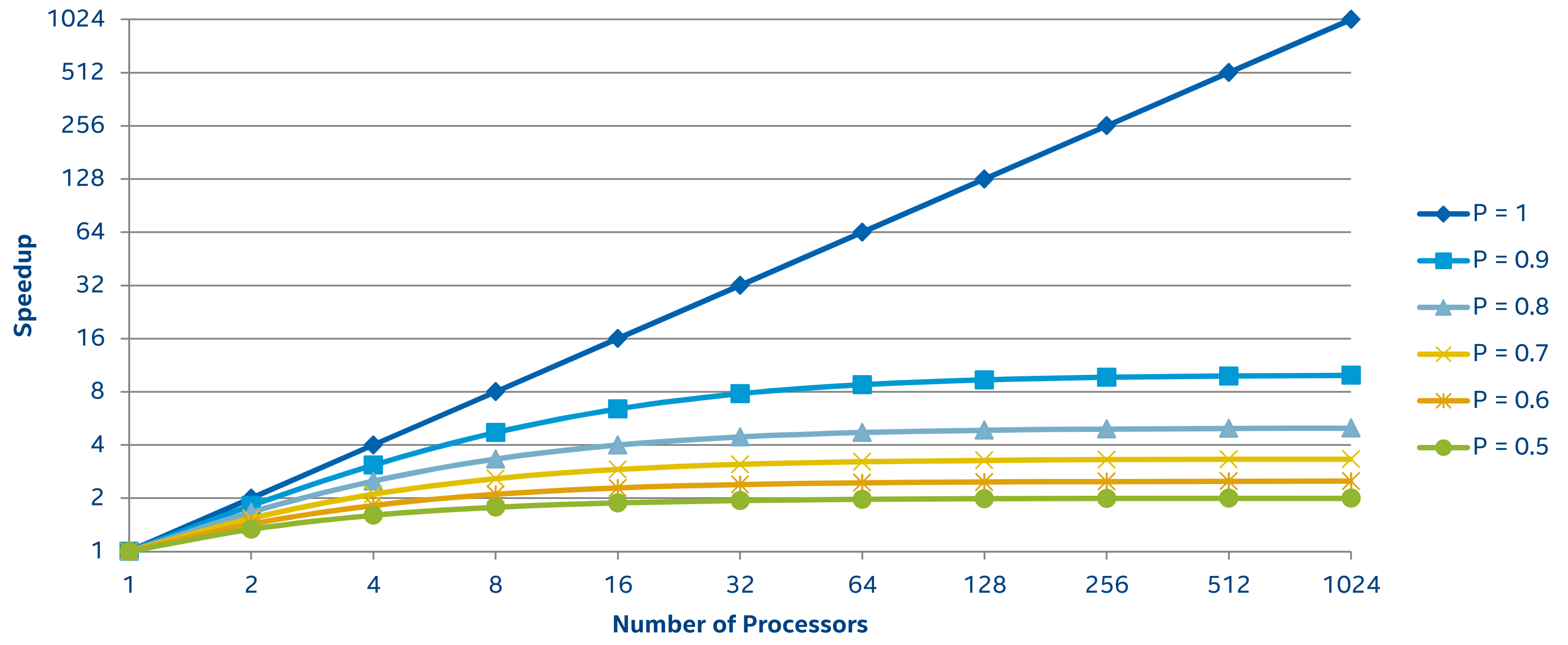
$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

where:

- $S(N)$ = speedup on N processors
- P = fraction of code that can be parallelised
- N = number of processors

The speedup of "strong scaling" applications is governed by Amdahl's Law.
As $N \to \infty$, $S(N) \to \frac{1}{(1-P)}$.

Impact of Amdahl's Law

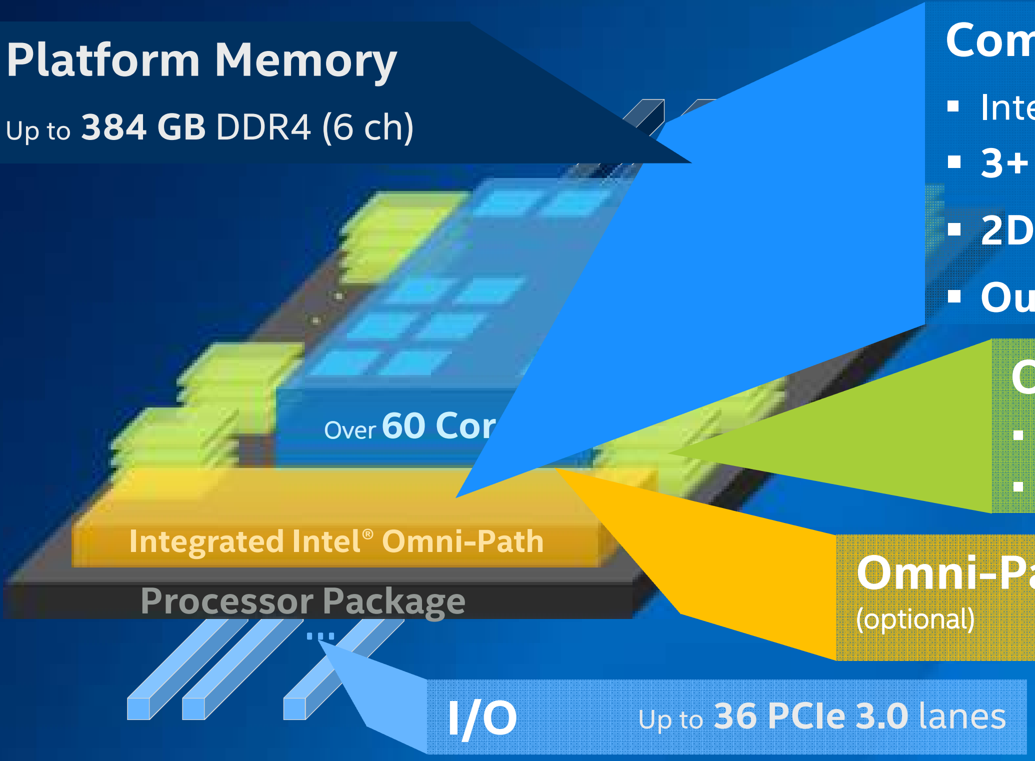


Amdahl's Law in Practice

- Assumption that P and N are independent is unrealistic.
  - Strong Scaling:
    All-to-all communication costs increase with N.
    For sufficiently large N, applications will start to slow down again!
  - Weak Scaling:
    Increasing problem size may not linearly increase compute time.
- Key takeaway from both laws:
  maximize P to maximize efficiency and performance at scale.
- Parallelism "bolted on" to scalar applications will not scale.

Knights Landing

Holistic Approach to Real Application Breakthroughs



Platform Memory

Up to **384 GB** DDR4 (6 ch)

Compute

- Intel® Xeon® Processor Binary-Compatible
- **3+ TF**LOPS[1], **3X ST**[2] perf. vs Xeon Phi™ coprocessor
- **2D Mesh** Architecture
- **Out-of-Order** Cores

On-Package Memory

- Over **5x** STREAM vs. DDR4[3]
- Up to **16 GB** at launch

Omni-Path (optional)

- **1st** Intel processor to integrate

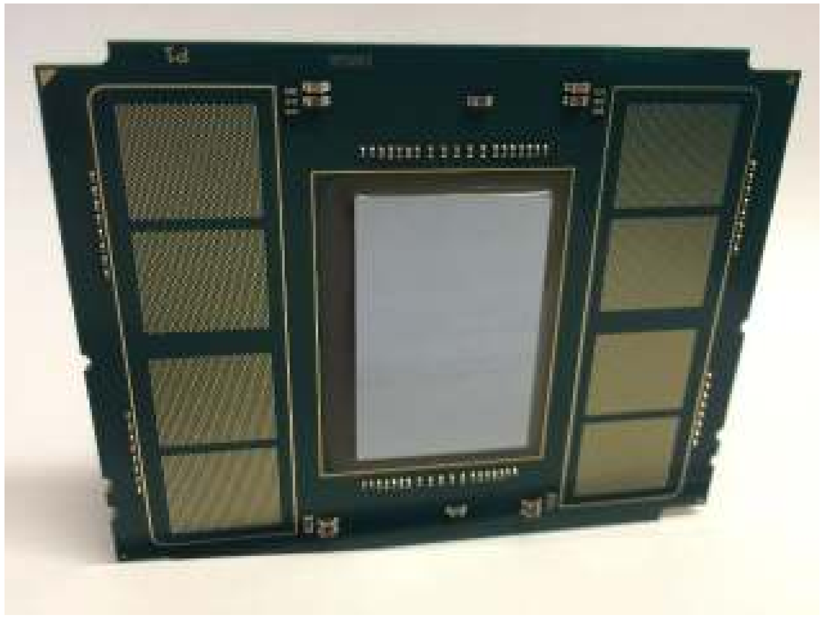
I/O Up to **36 PCIe 3.0** lanes

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.
Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
For more complete information visit http://www.intel.com/performance.

Knights Landing

- > 8 billion transistors
- 14nm process

- Over 10 GF/W



All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. All projections are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

Knights Landing at large

- Cori @ NESRC, built by Cray (> 9,300 nodes, mid-2016)
- Trinity @ NNSA, also built by Cray
- >100 Petaflops of deals to date

Form factors

- Bootable, standalone processor
    - Up to 384GB DD4 using 6 channels
    - 3 or more KNL in 1U
- PCIe coprocessor
- Both native and offload programming models

Single-threaded performance

- 3x single-thread performance compared to current generation
- ‘Silvermont’-based core with modifications
    - 2 VPUs/core
    - Advanced branch prediction
    - 2x ROB depth
- 32 kb Icache, 32 kbDcache
- 1MB L2 per tile
- 2 64B load ports
- First processor that supports AVX-512
    - AVX-512F, CDI, ERI, & PFI

AVX-512 in Knights Landing

ISA	AVX-512F	AVX-512 CDI	AVX-512 ERI	AVX-512 PFI
Features	‘Foundation’: double, float, int32, int64 arithmetic, load/store, with masks	vector confict detection, lzcnt	Transcendentals	Prefetches

Many-core performance

- 2D mesh of tiles
- 30+ tiles
  - 2 cores/tile
  - 1MB shared L2/tile

Near memory

- AKA ‘on-package memory’, ‘high-bandwidth memory’, MCDRAM
  - Partnership with Micron Technology
- > 400 GB/s
- Up 16GB (at launch)
- NUMA support
- Over 5x energy efficiency, 3x density vs. GDDR5
- Multiple usage models, including ‘L3 cache’ and ‘flat’

Intel® Omni Path fabric integration

- See more from Intel's Barry Davis tomorrow @ 3pm (Technical Session 8B)

  "A Storm (Lake) is Coming to Fast Fabrics: The Next-Generation Intel® Omni-Path Architecture"

Q & A

How do I get good performance on Knights Landing?

Efficiency on Knights Landing

- 1st Knights Landing systems appearing by end of year
- How do we prepare for this new processor without it at hand?
- Let's review the main performance-enabling features:
    - 60+ cores
    - 2x VPU, AVX-512
    - High-bandwidth MCDRAM
- Plenty of **parallelism** needed for best performance.

MPI needs help

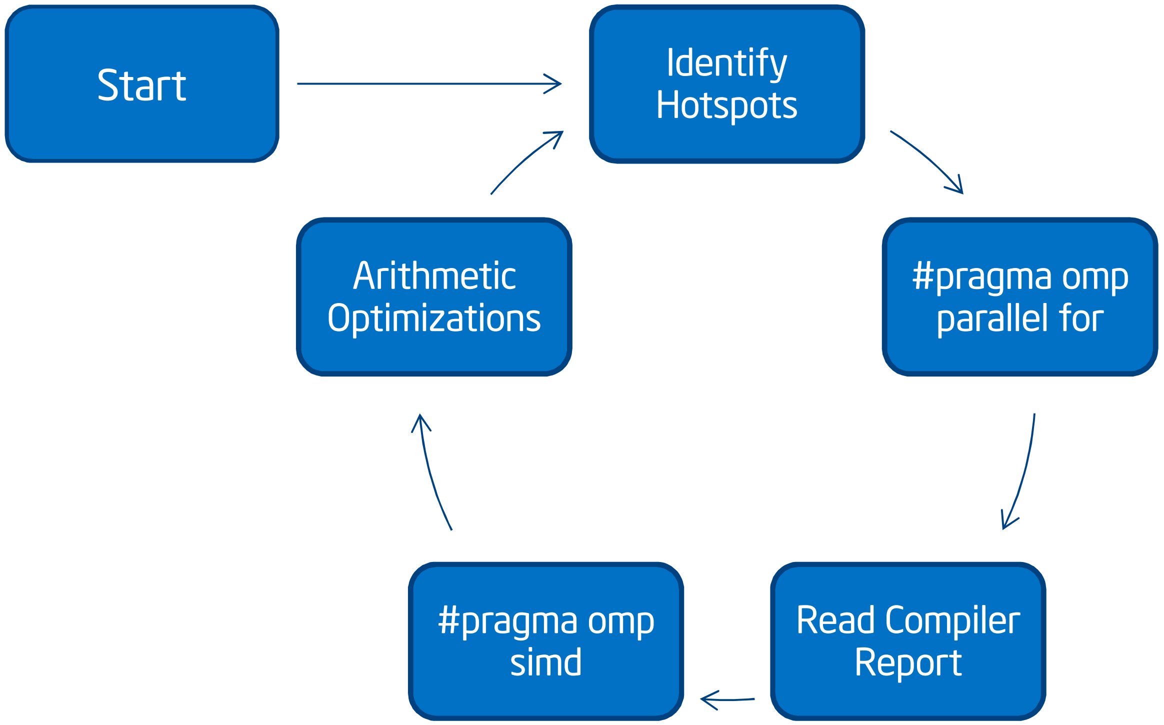
- Many codes are already parallel (MPI)
  - May scale well, but…
  - What is single-node efficiency?
  - MPI isn’t vectorizing your code…
  - It has trouble scaling on large shared-memory chips.
    - Process overheads
    - Handling of IPC
    - Lack of aggregation off-die
- Threads are most effective for many cores on a chip
- Adopt a hybrid thread-MPI model for clusters of many-core

OpenMP 4

- OpenMP helps express thread- and vector-level parallelism via directives

  (like `#pragma omp parallel`, `#pragma omp simd`)
- Portable, and powerful
- Don’t let simplicity fool you!
  - It doesn’t make parallel programming easy
  - There is no silver bullet
- Developer still must expose parallelism & test performance

The “Evolutionary” Approach



Case Study: European Options Pricing

- European Options Pricing kernel
  - Longstanding HPC proxy for financial workloads
  - Simulation phase only
- 1D Monte Carlo integration
  - Many (1e5-1e7+) options considered
  - Long (256k) paths
- Uses normally-distributed random numbers
  - Pre-generated and streamed

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

S :: Spot price
K :: Strike price
C(S,t) :: Price of call option with price S at time t
N(x) :: CDF of normal dist. with variance x
σ :: volatility of returns
T :: expiration date

Baseline Implementation

- Performance is $O((c_0 + c_1 * npath) * nopt)$
- Since we do ~2^{18} paths per option, path computation (c_1) dominates runtime
  - > 99% of work in 3 lines of code
- Conventionally viewed as compute bound
  - Exponentiation, lots of arithmetic
  - Small working set (1 RNG per path step, reduction on output)

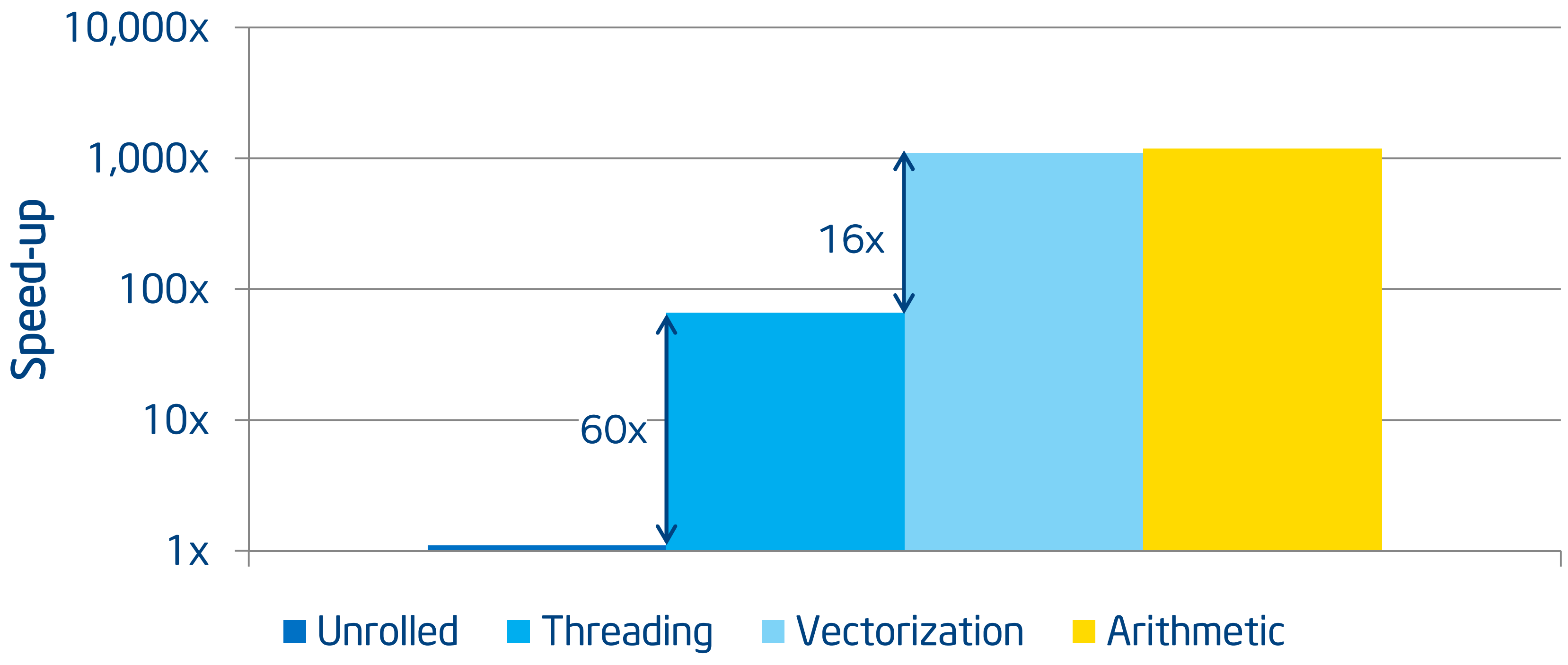
```
for(int o = 0; o < nopt; o++) {
  const REAL_T v_rt_t = sqrt(T[o]) * vol;
  const REAL_T mu_t   = T[o]   * mu;
  REAL_T v0 = 0, v1 = 0, res;
  for(int p = 0; p < npath; ++p) {
    res = max(0, S[o]*exp(v_rt_t*m_r[p]
             + mu_t)-X[o]);
    v0 += res;
    v1 += res*res;
  }
  result    [o] += v0;
  confidence[o] += v1;
}
```

Optimized & Modernized Implementation

- Loop Unrolling (#pragma unroll)
  - Short loop hurts instruction scheduling.
- Threading (#pragma omp parallel)
  - Embarrassingly parallel.
  - No write conflicts and small working set.
- Vectorization (#pragma omp simd)
  - v0/v1 must be reduced.
  - max() call introduces control divergence.
  - m_r[p] should be aligned.
- Arithmetic
  - Use native exp2() call on coprocessor.

```
#pragma omp parallel for
for(int o = 0; o < nopt; o++) {
 const REAL_T _rt_tLN2=sqrt(T[o])*vol/M_LN2;
  const REAL_T mu_tLN2 = T[o]*mu/MLN2;
  REAL_T v0 = 0, v1 = 0, res;
#pragma omp simd reduction(+:v0), reduction(+:v1),
 aligned(m_r:64)
#pragma unroll(4)
  for(int p = 0; p < npath; ++p) {
    res = max(0, S[o]*exp2(v_rt_tLN2*m_r[p]
                + mu_tLN2)-X[o]);
    v0 += res;
    v1 += res*res;
  }
  result    [o] += v0;
  confidence[o] += v1;
}
```

Performance Results



The “Evolutionary” Approach – Summary

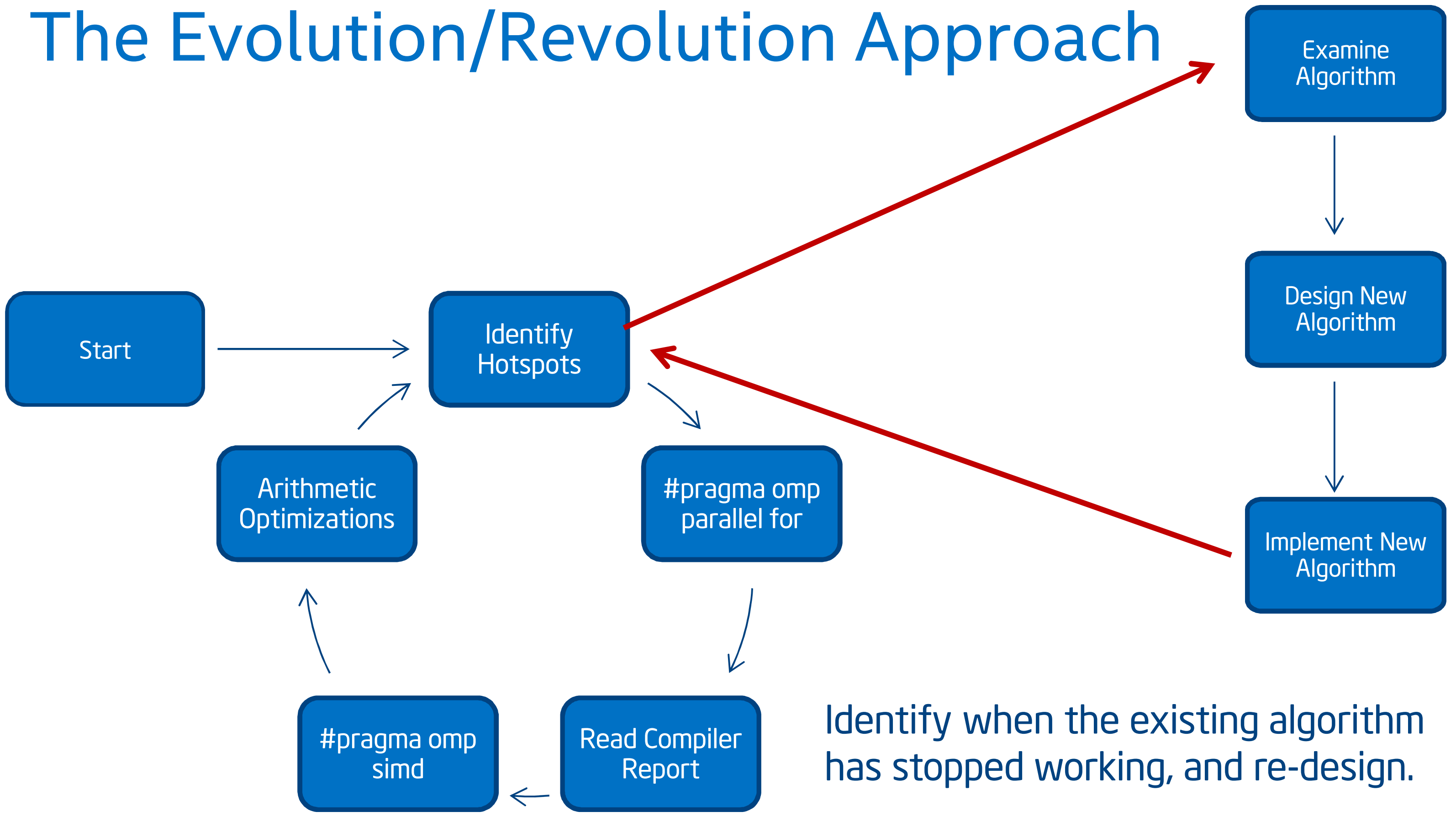
The “evolutionary” approach can get you 1000x **if**:

- Your baseline is serial and scalar.
- Your code is embarrassingly parallel with no dependencies.
- You have one hotspot – no file I/O, MPI* communication.

If only more codes looked like this… ☹

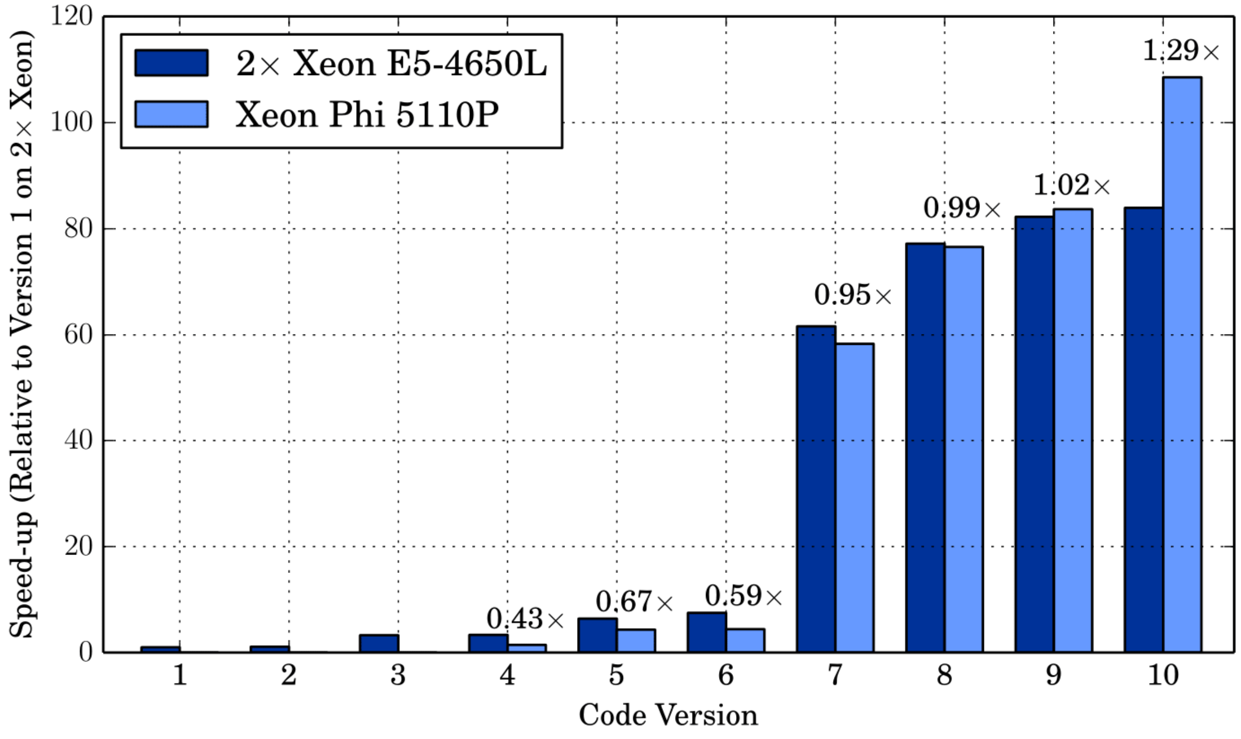
Real codes will get somewhere between 0x and 1000x, depending on: code, compiler ability and the amount of exposed parallelism.

The Evolution/Revolution Approach



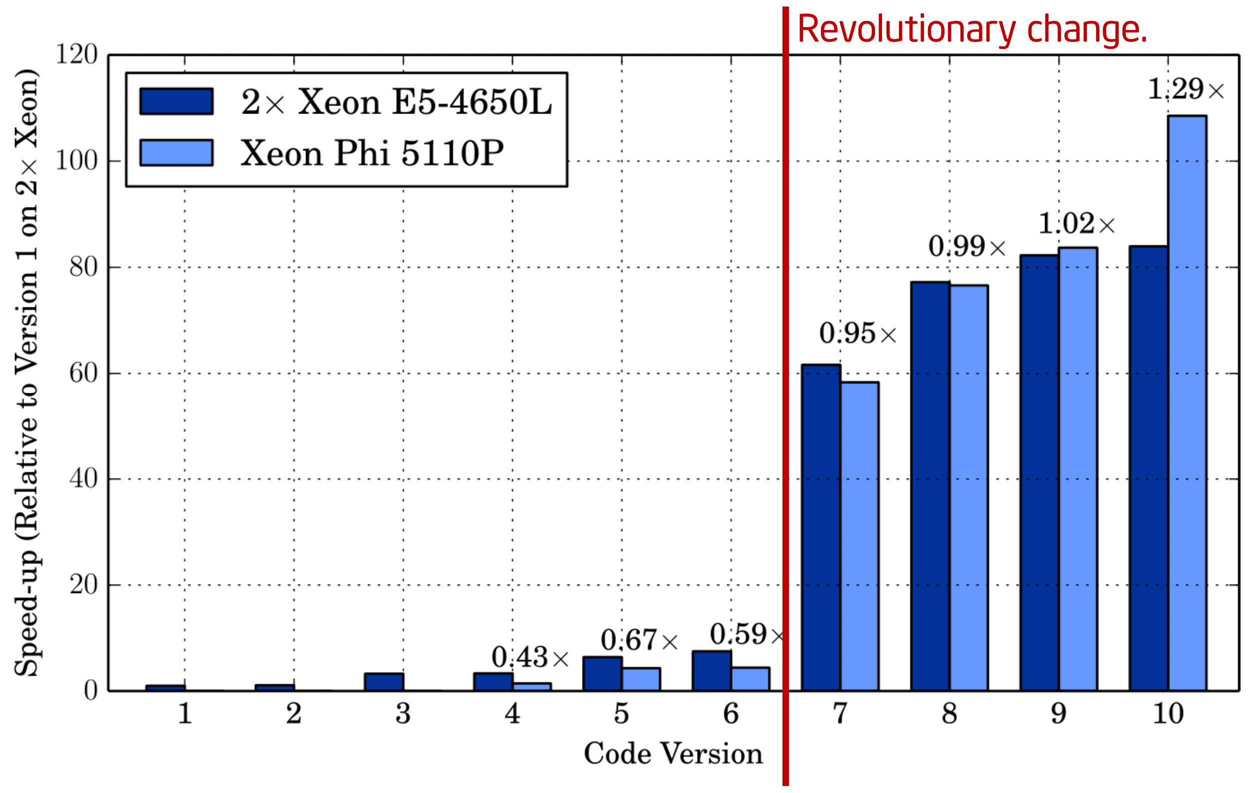
Identify when the existing algorithm has stopped working, and re-design.

Case Study: Cosmic Microwave Background Analysis



Version Decoder:
1: Baseline
2+3: Low-hanging Fruit
4: Flat OpenMP*
5: Nested OpenMP*
6: Blocking

Case Study: Cosmic Microwave Background Analysis



Version Decoder:

1: Baseline

2+3: Low-hanging Fruit

4: Flat OpenMP*

5: Nested OpenMP*

6: Blocking

7: Trapezium Rule

8: DGEMM

9: Alignment

10: Prefetching

Summary

- Adding pragmas and crossing fingers rarely solves the problem.
- Consider how hardware **should** be used before worrying about the implementation.
  - Can't know if the compiler is doing the "right thing" without first knowing what that is.
  - Choosing a programming model/methodology first may restrict algorithmic choice.
- Revisit your algorithms and throw out your assumptions. A parallel implementation of a "slower" algorithm may be faster at scale.
- Investment in code yields improved performance **and** better science.

After the break

- Getting code built for Knights Landing & testing correctness
- Working with the best proxy for Knights Landing

Q & A

Break

What can I do *now* to get ready for Knights Landing?

Proxies for Knights Landing

- Two pronged approach for Knights Landing readiness
  - Software support tools
    - Intel® Composer Studio XE 2015 compiles for Knights Landing
    - Intel® Software Development Emulator functionally emulates; rough perf. analysis.
    - Memkind & hbw_malloc harness MCDRAM
  - Use the best available real-world proxy for performance testing
    - Intel® Xeon Phi™

memkind & hbw_malloc

A Heterogeneous Memory Management Framework

- The memkind library
  - Defines a plug-in architecture
  - Each plug-in is called a "kind" of memory
  - Built on top of jemalloc: the FreeBSD OS default heap manager
  - Partition is defined by functions that provide inputs for operating system calls
  - High level memory management functions can be over-ridden as well
  - Available via github: https://github.com/memkind

- The hbwmalloc interface
  - The high bandwidth memory interface
  - Implemented on top of memkind interface
  - Simplifies memkind plug-in (kind) selection
  - Uses all kinds featuring on package memory on the Knights Landing architecture
  - Provides support for 2MB and 1GB pages
  - Select fallback behavior when on package memory does not exist or is exhausted
  - Check for existence of on package memory

Knights Landing Memory Overview

- Large numbers of cores can consume large amounts of memory bandwidth.
- Knights Landing is equipped with 6 bidirectional DDR4 memory channels and MCDRAM.
- Memory requests are serviced by a mesh network maintaining cache coherence.
- MCDRAM can be configured as a third level cache, as a flat, distinct region of memory, or somewhere in-between.

- Basic memory features
  - MCDRAM is high bandwidth, lower capacity.
  - DDR is high capacity lower bandwidth.

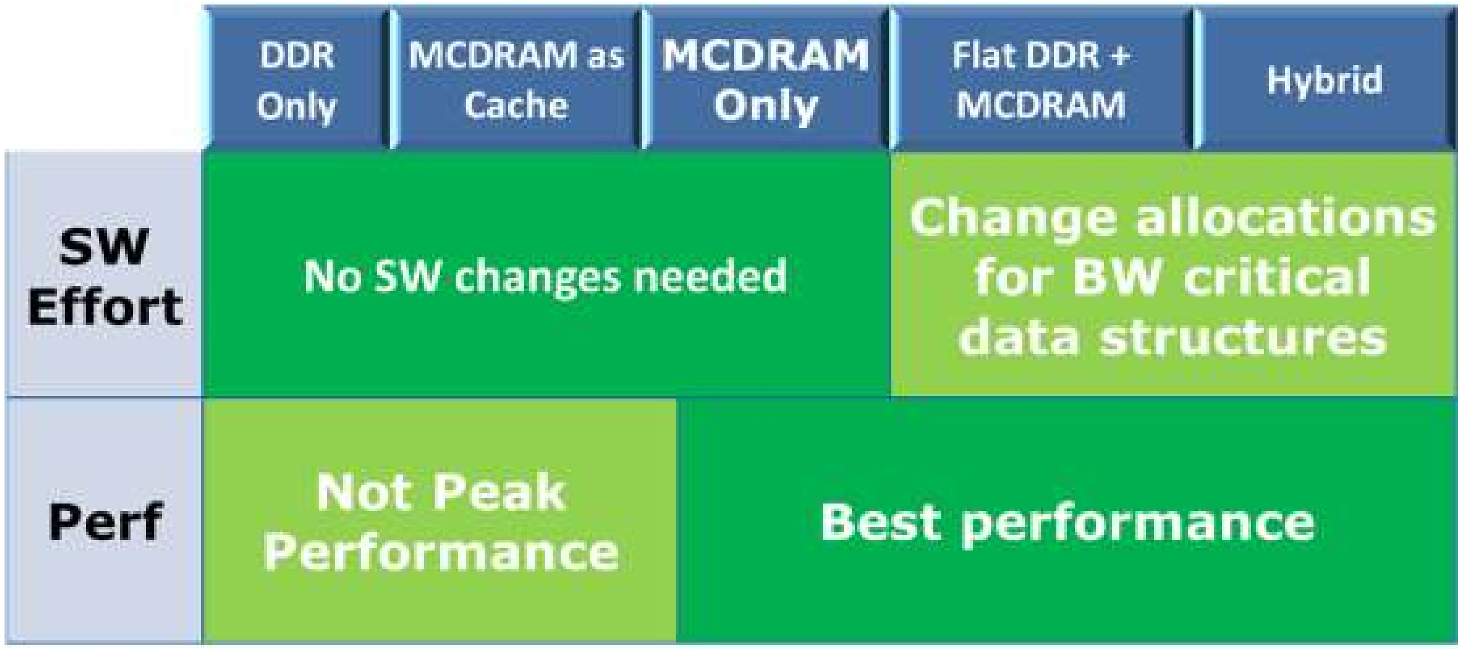
MCDRAM as Cache

- Upside
  - No software modifications required.
  - Bandwidth benefit.
- Downside
  - Latency hit to DDR.
  - Limited sustained bandwidth.
  - All memory is transferred DDR -> MCDRAM -> L2.
  - Less addressable memory.

Flat Mode

- Upside
  - Maximum bandwidth and latency performance.
  - Maximum addressable memory.
  - Isolate MCDRAM for HPC application use only.
- Downside
  - Software modifications required to use DDR and MCDRAM in the same application.
  - Which data structures should go where?
  - MCDRAM is a limited resource and tracking it adds complexity.

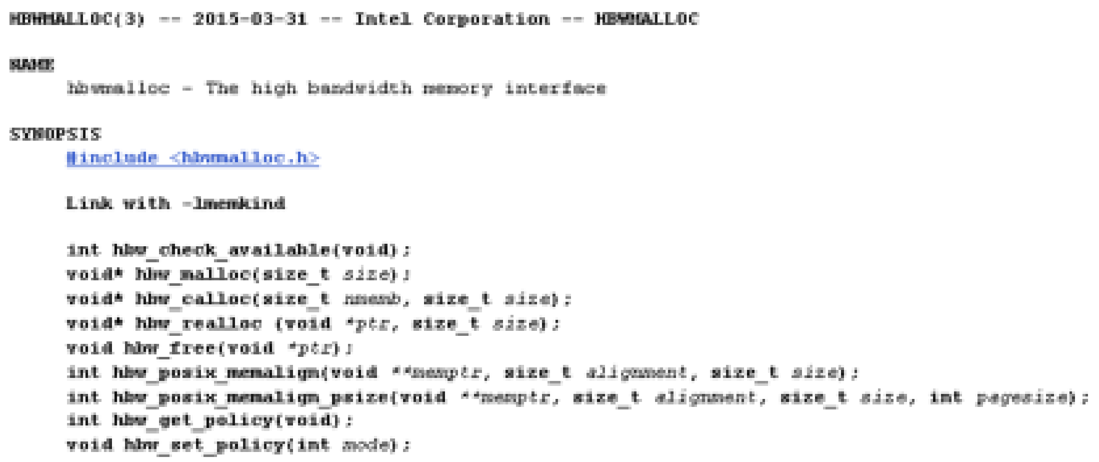
What mode should I pick?



	DDR Only	MCDRAM as Cache	MCDRAM Only	Flat DDR + MCDRAM	Hybrid
SW Effort	No SW changes needed			Change allocations for BW critical data structures	
Perf	Not Peak Performance		Best performance		

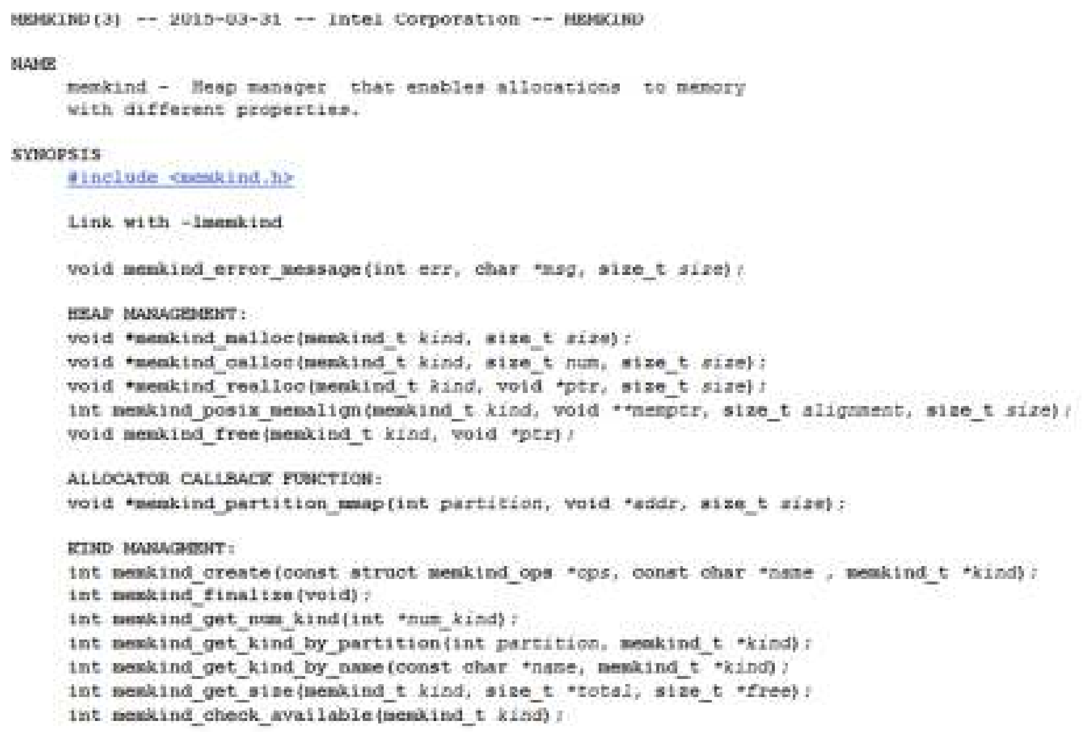
Flexible tradeoffs available

hbwmalloc Interface



```
HBWMALLOC(3) -- 2015-03-31 -- Intel Corporation -- HBWMALLOC

NAME
     hbwmalloc - The high bandwidth memory interface

SYNOPSIS
     #include <hbwmalloc.h>

     Link with -lmemkind

     int hbw_check_available(void);
     void* hbw_malloc(size_t size);
     void* hbw_calloc(size_t nmemb, size_t size);
     void* hbw_realloc (void *ptr, size_t size);
     void hbw_free(void *ptr);
     int hbw_posix_memalign(void **memptr, size_t alignment, size_t size);
     int hbw_posix_memalign_psize(void **memptr, size_t alignment, size_t size, int pagesize);
     int hbw_get_policy(void);
     void hbw_set_policy(int mode);
```

memkind Interface



```
MEMKIND(3) -- 2015-03-31 -- Intel Corporation -- MEMKIND

NAME
     memkind -  Heap manager  that enables allocations  to memory
     with different properties.

SYNOPSIS
     #include <memkind.h>

     Link with -lmemkind

     void memkind_error_message(int err, char *msg, size_t size);

     HEAP MANAGEMENT:
     void *memkind_malloc(memkind_t kind, size_t size);
     void *memkind_calloc(memkind_t kind, size_t num, size_t size);
     void *memkind_realloc(memkind_t kind, void *ptr, size_t size);
     int memkind_posix_memalign(memkind_t kind, void **memptr, size_t alignment, size_t size);
     void memkind_free(memkind_t kind, void *ptr);

     ALLOCATOR CALLBACK FUNCTION:
     void *memkind_partition_mmap(int partition, void *addr, size_t size);

     KIND MANAGMENT:
     int memkind_create(const struct memkind_ops *ops, const char *name , memkind_t *kind);
     int memkind_finalize(void);
     int memkind_get_num_kind(int *num_kind);
     int memkind_get_kind_by_partition(int partition, memkind_t *kind);
     int memkind_get_kind_by_name(const char *name, memkind_t *kind);
     int memkind_get_size(memkind_t kind, size_t *total, size_t *free);
     int memkind_check_available(memkind_t kind);
```

End Goal Usage: Code Snippets

Allocate 1000 floats from DDR

```
float   *fv;

fv = (float *)malloc(sizeof(float) * 1000);
```

Allocate 1000 floats from MCDRAM

```
float   *fv;

fv = (float *)hbw_malloc(sizeof(float) * 1000);
```

Allocate arrays from MCDRAM & DDR in Intel FORTRAN

```
c     Declare arrays to be dynamic
      REAL, ALLOCATABLE :: A(:), B(:), C(:)

!DIR$ ATTRIBUTES FASTMEM :: A

      NSIZE=1024
c
c     allocate array ‘A’ from MCDRAM
c
      ALLOCATE (A(1:NSIZE))
c
c     Allocate arrays that will come from DDR
c
      ALLOCATE  (B(NSIZE), C(NSIZE))
```

Intel® Software Development Emulator

Functional emulation

Intel® Software Development Emulator

- Freely available instruction emulator
- Emulates existing ISA as well as ISAs for upcoming processors (including Knights Landing)
- Record dynamic instruction mix; useful for tuning/assessing vectorization content
- http://www.intel.com/software/sde

- First step: compile for Knights Landing:

```
$ icpc -xMIC-AVX512 <compiler args>
```

Running SDE

- SDE invocation is very simple:

```
$ sde <sde-opts> -- <binary> <command args>
```

- By default, SDE will execute the code with the CPUID of the host.
  - The code may run more slowly, but will be functionally equivalent to the target architecture.
- For Knights Landing, you can specify the `-knl` option.
- In addition to emulation, SDE can summarize the types of instructions that were executed

```
$ sde <sde-opts> -omix <output-file> -- <binary> <command args>
```

- The output file will contain statistics about the instruction mix, with adjustable granularity

Basic Block Stats from SDE mix

```
▪ # ===============================================
▪ # STATS FOR TID 0 EMIT# 1
▪ # ===============================================
▪ # EMIT_TOP_BLOCK_STATS FOR TID 0 EMIT # 1 EVENT=ICOUNT
▪ BLOCK: 00000   PC: 0000000000410c23   ICOUNT: 15983666400   EXECUTIONS: 270909600   #BYTES: 272   %:    15   cumltv%:
  15  FN: swUpdatePress2ndTiltedZ_DDz1_8  IMG: swell-TTC2-12x12x8 Source: swUpdatePress2ndTilted-orig.tc 274,273
▪ XDIS 0000000000410c23:  SSE 430F101498               movups xmm2, xmmword ptr [r8+r11*4]
▪ XDIS 0000000000410c28:  SSE 420F101C98               movups xmm3, xmmword ptr [rax+r11*4]
▪ XDIS 0000000000410c2d:  SSE 0F59D1                   mulps xmm2, xmm1
▪ XDIS 0000000000410c30:  SSE 410F59DF                 mulps xmm3, xmm15
▪ XDIS 0000000000410c34: BASE 4C8BBC2468010000         mov r15, qword ptr [rsp+0x168]
▪ XDIS 0000000000410c3c:  SSE 0F58D3                   addps xmm2, xmm3
▪ XDIS 0000000000410c3f:  SSE 430F105C9D00             movups xmm3, xmmword ptr [r13+r11*4]
```

- ICOUNT: total dynamic instructions executed by this basic block. Basic blocks sorted by ICOUNT from highest to lowest.
- EXECUTION: number of time is basic block is invoked
- %: percent of total instructions that come from this basic block
- cumltv%: cumulative % of instruction count up to this basic block

Basic Block Stats from SDE mix (cont.)

- Look for unpack *ss or *sd instructions (i.e., scalar instructions) in top basic blocks
- Are they SSE, AVX, AVX2, AVX512 instructions?
  - For KNL, we want as many AVX512 instructions as possible
  - Sometimes, non AVX512 instructions come from math libraries and some math functions are not optimized for AVX512 yet.
- Are there gathers/scatters (non-unit stride)? Can code be transformed to remove gathers/scatters?

Xeon Phi^{TM} as proxy for Xeon Phi^{TM}

A performance proxy

- Functional emulation and advanced APIs are invaluable for testing out new instructions and features
- Performance testing very important
- How will your code run on Knights Landing?
  - Know your code! Is it compute-hungry? Memory bandwidth-hungry?
  - How do the performance features of Knights Landing affect those limits?
- Is there an existing processor that is a good proxy for Knights Landing for your code?

Proxy Matching

	Intel® Xeon® E5-2696v3	Intel® Xeon Phi™ 7120	Knights Landing
Cores/threads	14/28	61/244	60+/240+
Nom. Hz	2.6GHz	1.3GHz	N/A
STREAM BW	~50 GB/s	~170 GB/s	>400 GB/s (MCDRAM)
SIMD width	256 bits	512 bits	512 bits
LLC capacity	35MB	30.5MB	30+MB
DRAM cap.	768GB	16GB	384GB

- Modern Xeon® processors have more cores and wider SIMD than ever
- Xeon Phi™ is still much closer to Knights Landing in width
- Compute-bound codes should seek peak on Xeon Phi™
- Bandwidth-bound codes will find Xeon Phi™ closer
- Only high memory-capacity codes favor Xeon® in this comparison
  - May not be throughput codes anyway
  - Consider using reduced problem size for tuning

Using the proxy

- The current Xeon Phi™ is almost always the closest proxy to Knights Landing
  - Native Xeon Phi corresponds to bootable Knights Landing
- What drives performance on Xeon Phi™?
  - Thread scalability
    - Load balancing
    - Divergence
    - (Inter-core) communication costs
  - SIMD vectorization
  - High bandwidth GDDR5
- Thread-level parallelism is not easy, but shares similarities with MPI-type
- SIMD is less familiar ground for most…

Opening the Flop floodgates

Efficiency from a SIMD Mindset

Motivation

- Single-instruction, multiple data (SIMD) available on most processors
    - 4x-16x ‘on the table’
- Powerful, but more restrictive than multiple cores
- New tools, better hardware: the stars are right!

What is SIMD?

Data Level Parallelism

SIMD primer

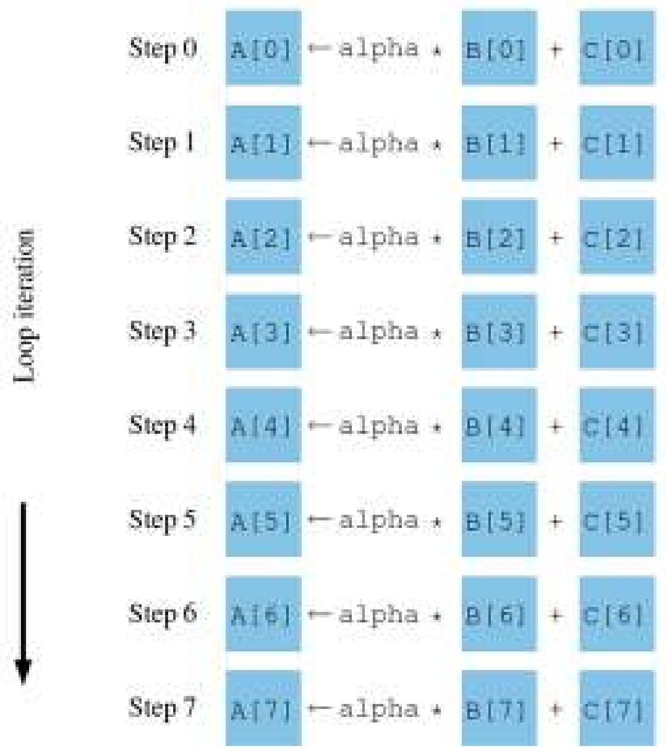
SIMD and other forms of parallelism

Data level Parallelism

- “I have lots of data I want to do the same operation to”
  - *Very* common idiom in HPC
- Classic for loop:

```
for(int i=0;i<8;++i)
    A[i] = alpha*B[i] + C[i];
```

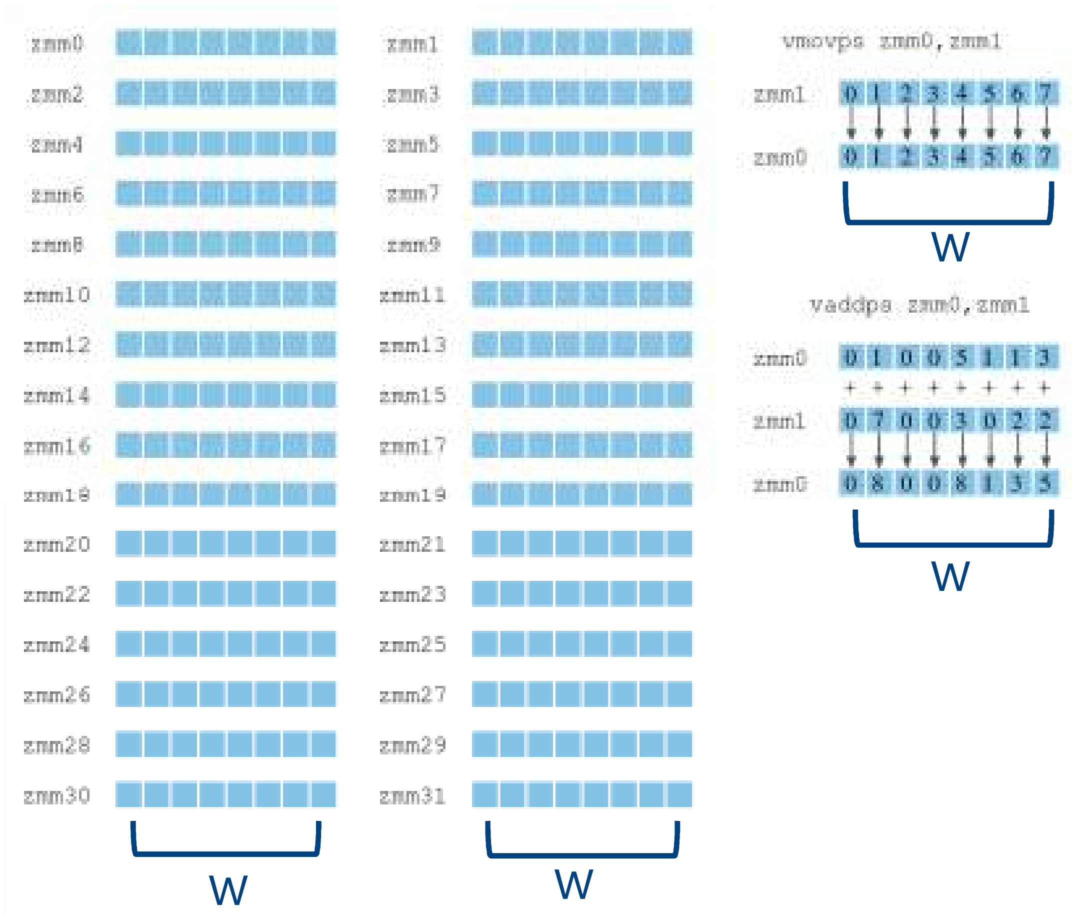
- Lots of iterations
- Independent
- Arithmetic
- Multi-node, multi-core possible
  - Overheads
  - Is there a more efficient way?



Loop iteration
Step 0 A[0] ← alpha * B[0] + C[0]
Step 1 A[1] ← alpha * B[1] + C[1]
Step 2 A[2] ← alpha * B[2] + C[2]
Step 3 A[3] ← alpha * B[3] + C[3]
Step 4 A[4] ← alpha * B[4] + C[4]
Step 5 A[5] ← alpha * B[5] + C[5]
Step 6 A[6] ← alpha * B[6] + C[6]
Step 7 A[7] ← alpha * B[7] + C[7]

SIMD primer

- SIMD execution (Wx)
  - Vector registers
  - Vector functional units
  - Vector instructions
  - ~W FLOPs/instruction
  - Conceptually: ‘lanes’



SIMD in the Parallel Pantheon

- Modern processors have multiple cores
  - SIMD is in each core
  - (Mostly) orthogonal to threading
- Modern processors are superscalar
  - SIMD can be superscalar
- How is it different than threading?
  - SIMD data layout demanding
    - Can affect data structures and memory
  - Linked control flow
    - Branches can be tricky!
  - ‘Communication’ fine-grained

How does SIMD work?

Field guide to SIMD

How it helps

SIMD schemes

SIMD considerations

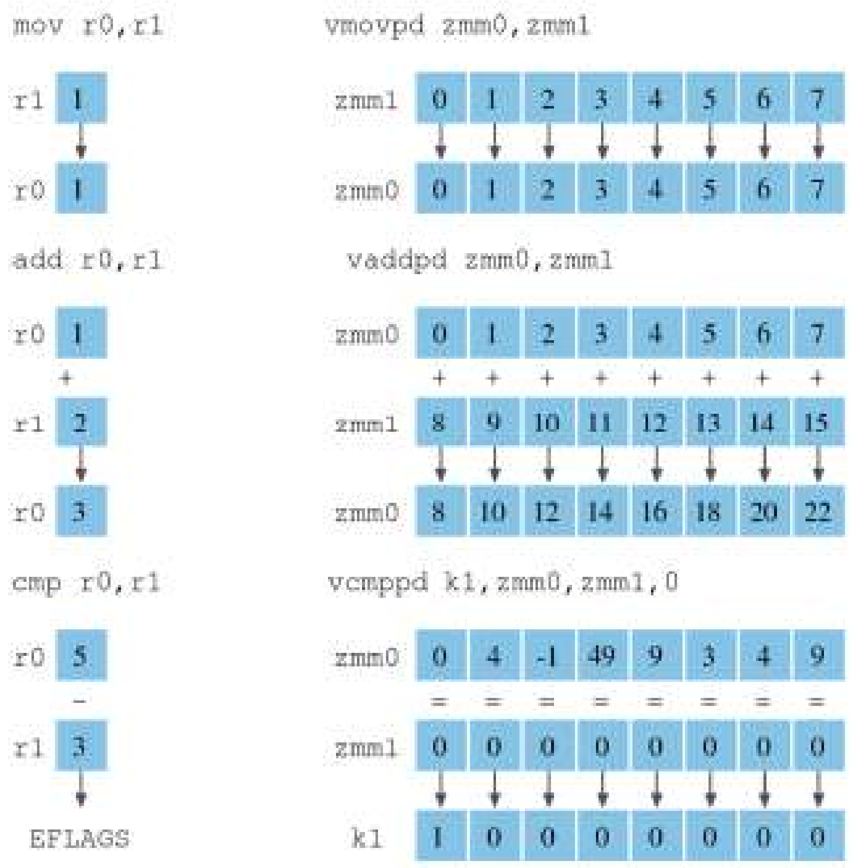
A field guide to SIMD

- Which scalar instructions should be ‘ported’ to SIMD?
  - Which data types? (multiple widths!)
- Always a ‘common core’: arithmetic
  - add, sub, mul, div, fma
- Some make no sense
  - cpuid?? cli? int?
- Some new instructions only for SIMD

x86_64	SIMD
mov	vmov
add	vadd
mul	vmul
fmadd	vfmadd
rdtsc	-
jmp	-
cli	-
cmpxchg	-
cpuid	-
nop	-
-	vgather
-	vshuff
-	hadd

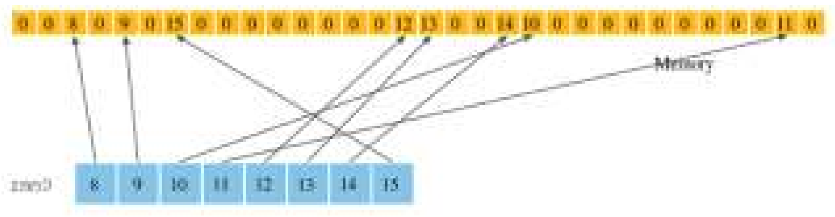
“Vanilla” SIMD

- Arithmetic (add, sub, mul, div, fma)
- Conversion
- Bit manip (and, or, xor, not, shift)
- Math (min, max, abs, sqrt, rsqrt, rcp, exp, sin, cos, etc…)
- Comparison (eq, lt, gt, etc)



MOV, writ large

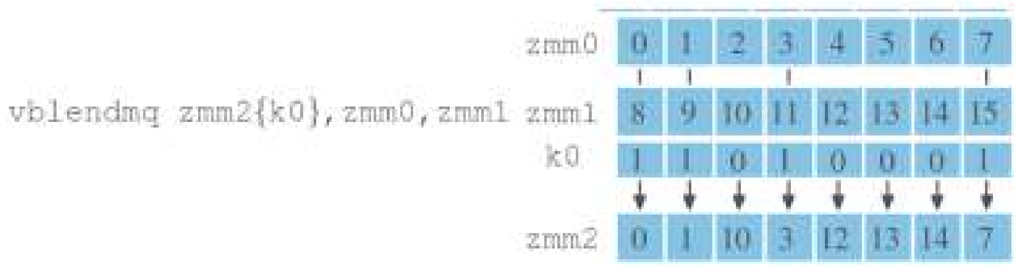
- New problem: how do you fill these registers?
  - Fill with 1 value from a (scalar) register/memory (broadcast)
  - Fill with W values from memory
    - Are they packed (contiguous)? (load)
    - Are they not? (gather)
- How do you store the result?
  - Write W values to memory
    - Packed? (store)
    - Not? (scatter)



SIMD Exclusives

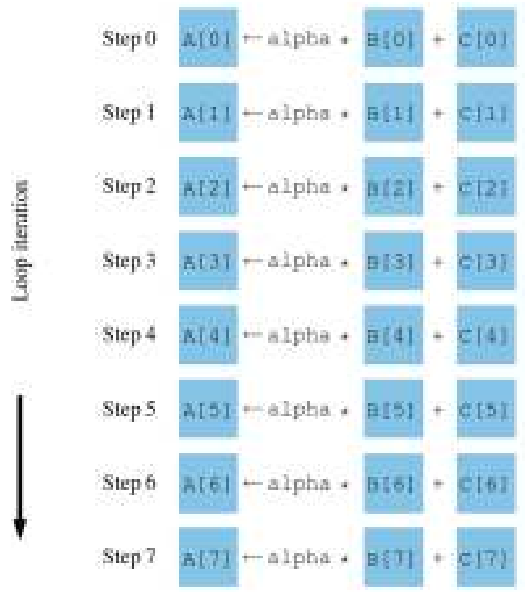
- Concepts in SIMD with no scalar meaning
  - Inter-lane shuffle
    - `shuffle/permute`
  - Combining registers
    - `blend`

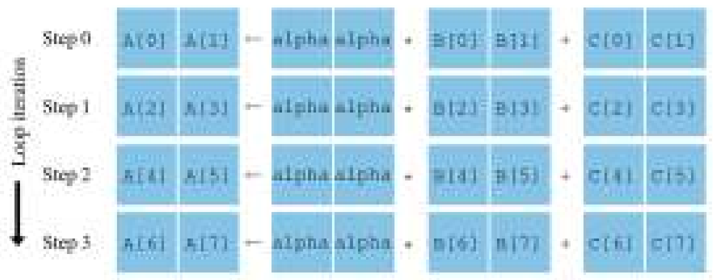
- Domain-specific
  - 'Horizontal' add
  - Dot product
  - Absolute difference
  - String search/compare
  - Encryption



How it helps

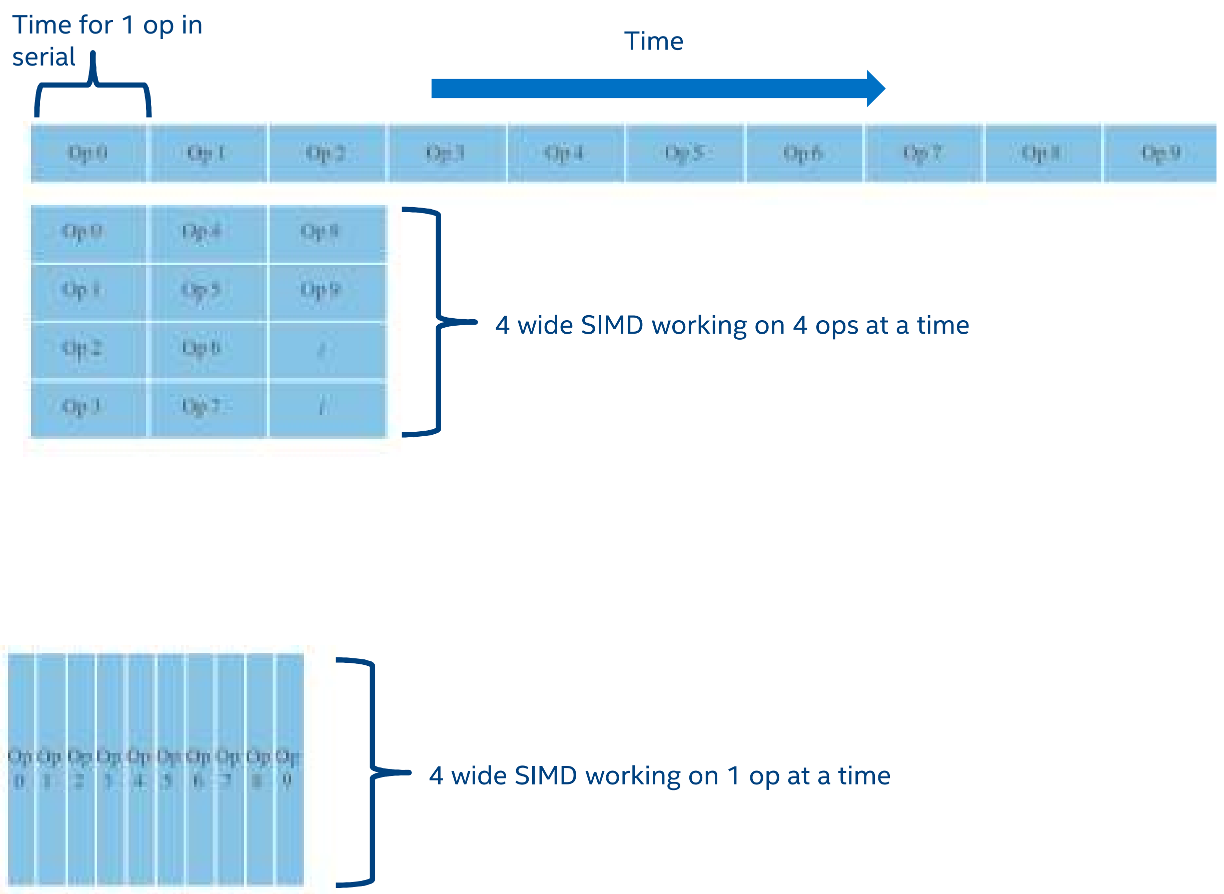
- SIMD version of inst. usually same throughput as serial
  - Same flops, less cycles
- Fewer instructions
- Performance gain depends on ratio of instructions compared to serial
  - Overhead from algorithm/ISA causes slowdown
  - Look for most efficient mapping


Loop iteration
Step 0 A[0] ← alpha · B[0] + C[0]
Step 1 A[1] ← alpha · B[1] + C[1]
Step 2 A[2] ← alpha · B[2] + C[2]
Step 3 A[3] ← alpha · B[3] + C[3]
Step 4 A[4] ← alpha · B[4] + C[4]
Step 5 A[5] ← alpha · B[5] + C[5]
Step 6 A[6] ← alpha · B[6] + C[6]
Step 7 A[7] ← alpha · B[7] + C[7]

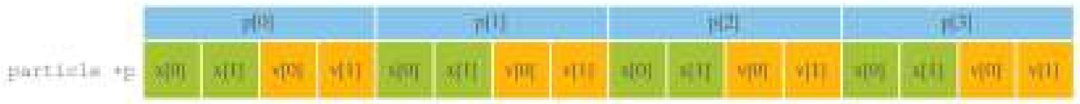

Loop iteration
Step 0 A[0] A[1] ← alpha alpha · B[0] B[1] + C[0] C[1]
Step 1 A[2] A[3] ← alpha alpha · B[2] B[3] + C[2] C[3]
Step 2 A[4] A[5] ← alpha alpha · B[4] B[5] + C[4] C[5]
Step 3 A[6] A[7] ← alpha alpha · B[6] B[7] + C[6] C[7]

SIMD schemes

- ‘Op’: an (arbitrary) piece of independent work
- For SIMD of width 4…
- Vertical
    - 4 ‘ops’; same time 1 serial
- Horizontal
    - 1 ‘op’; 1/4 the time 1 serial


Time for 1 op in serial
Time
Op 0
Op 1
Op 2
Op 3
Op 4
Op 5
Op 6
Op 7
Op 8
Op 9
4 wide SIMD working on 4 ops at a time
4 wide SIMD working on 1 op at a time

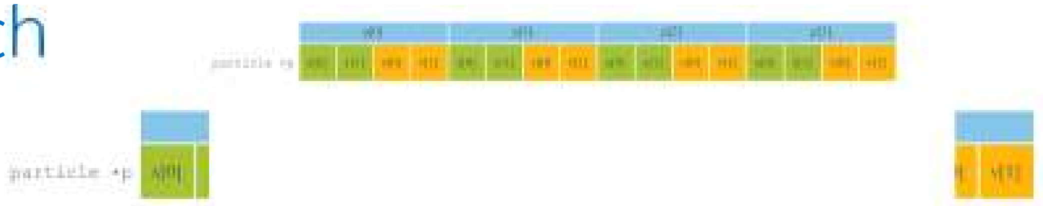
Example: Particle integration
Serial code



```
struct particle {
    double x[2];
    double v[2];
};

int N = 2000;
double  dt = 2e-3;
particle *p = ….
….
for(int i = 0; i < N; ++i) {
      for(int j = 0; j < 2; ++j) {
                p[i].x[j] += p[i].v[j]*dt;
            }
}
```

2 fused multiply-adds/particle

Example: Particle integration
‘Horizontal’ approach

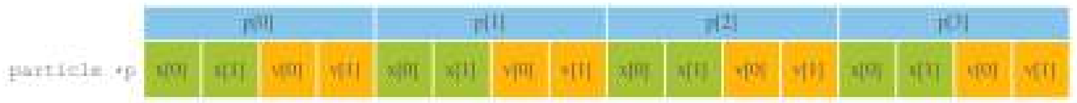


```
struct particle {
    double x[2];
    double v[2];
};

int N = 2000;
double  dt = 2e-3;
particle *p = ….
….
for(int i = 0; i < N; ++i) {
      for(int j = 0; j < 2; ++j) {
                p[i].x[j] += p[i].v[j]*dt;
            }
}
```

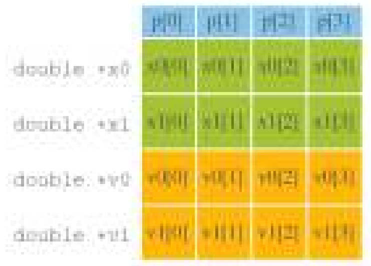
- Maximum speedup: 2x
- Independent of N
- Same data layout
- Depends on W

Example: Particle integration
‘Vertical’ approach

```
struct particle {
    double x[2];
    double v[2];
};

int N = 2000;
double  dt = 2e-3;
particle *p = ….
….
for(int i = 0; i < N; ++i) {
      for(int j = 0; j < 2; ++j) {
                p[i].x[j] += p[i].v[j]*dt;
            }
}
```



Data layout for SIMD



- Array-of-Structure
  - For ‘vertical’ methods, lots of movement
  - Inputs ‘gathered’ into SIMD, ‘scattered’ back
- Gather/scatter more work
  - Serialize for load
  - Special gather/scatter hardware speeds it up
- Structure-of-Array
  - Data layout changed to make SIMD more efficient

```
struct particle_arrays {
    double *x0, *x1;
    double *v0, *v1;
};
particle_arrays p = { new double[200],
                      new double[200],
                      new double[200],
                      new double[200] };
```

```
struct particle {
    double x[2];
    double v[2];
};
particle *p = new particle[200];
```

How do you use SIMD?

Contemporary ISAs

Low-level

Mid-level

High-level

SIMD in your code

- Many flavors of SIMD
  - MMX, SSE{2,3,4,etc}, AVX{2}, IMCI, AVX-512
  - Different widths/types (bytes->doubles, 128-bits->512-bits)
  - Common ‘basic’ core
  - Different ‘specials’
- For portability, stay as high level as possible
  - Horizontal/hybrid schemes necessarily tied to width; operations too
- Think out ‘SIMD version’ of algorithm first!

Elemental Functions

- Write a function for one element and add `#pragma omp declare simd`

```
#pragma omp declare simd
float foo(float a, float b, float c, float d) {
  return a * b + c * d;
}
```

- Call the scalar version:

```
e = foo(a, b, c, d);
```

- Call scalar version via auto-vectorized or SIMD loop:

```
for(i = 0; i < n; i++) {
  A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

- Call it with array notations:

```
A[:] = foo(B[:], C[:], D[:], E[:]);
```

Loop Vectorization

- Auto-Vectorization:
  - One of the loop nest optimizations
  - Heuristics-driven
  - Write optimizable code (with optimization hints) and it just happens
- SIMD Pragma:
  - Vector programming construct: “vectorize here”
  - Similar to OpenMP*, which is for parallel programming: “parallelize here”.
  - Optimizable code (and optimization hints) still helpful for better code generation

SIMD Pragma

- Programmer asserts:
  - `*p` is loop invariant
  - `A[]` not aliased with `B[]`, `C[]` and `sum`
  - `sum` not aliased with `B[]` and `C[]`
  - `+` operator is associative (compiler can reorder for better vectorization)
- Vectorized code generated even if efficiency heuristic does not indicate a gain*

```
#pragma omp simd
reduction(+:sum)
for(i = 0; i < *p; i++) {
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

*Some things, like intrinsics & assembly can prevent it

SIMD Pragma & OpenMP*

- OpenMP*-like pragma for vector programming
- "Go ahead and generate vector code" model
- Additional semantics (private, reduction, linear, etc.) given to compiler via clauses

	directive	hint
vector	SIMD	IVDEP
thread	OpenMP*	PARALLEL

```
#pragma omp simd
for(int ray=0; ray < N; ray++) {
  float Color = 0.0f, Opacity = 0.0f;
  int len = 0;
  int upper = raylen[ray];
  while (len < upper) {
    int voxel = ray + len;
    len++;
    if(visible[voxel] == 0) continue;
    float O = opacity[voxel];
    if(O == 0.0) continue;
    float Shading = O + 1.0;
    Color += Shading * (1.0f - Opacity);
    Opacity += O * (1.0f - Opacity);
    if(Opacity > THRESH) break;
  }
  color_out[ray] = Color;
}
```

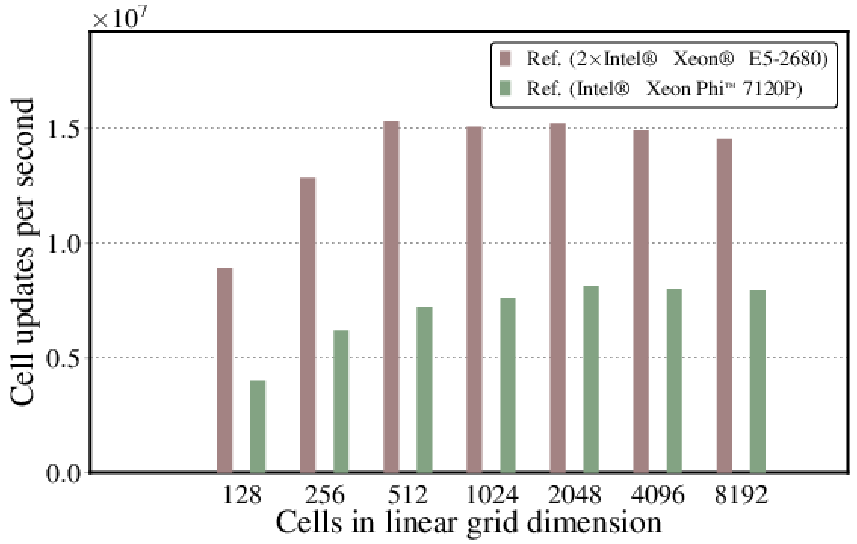
Think SIMD

- Efficient tool for data-level parallelism
  - Several variations, increasing sophistication
- Widely available, many models for use
  - Rapidly improving tools
- Understanding what you ***want*** to happen is key
  - Realization follows
- SIMD rarely works *ex post facto*
  - Build data structures & algorithms that work with it

From ‘Correct’ To ‘Correct & Efficient’: A Case Study With Hydro2D

Case study: 2D Shock Hydrodynamics

- Open-source version of production code from CEA
- Port from Fortran with so-so Xeon performance and awful Phi performance



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Source:intel

Case study: 2D Shock Hydrodynamics

- 2D Euler equations
  - Nonlinear system of 4 PDEs
  - Equation of state
- ρ is density
- u, v are x- and y- components of velocity
- E is total energy
- p is pressure
- γ is adiabatic constant
- In *conservation form*; each eq. is a conservation law

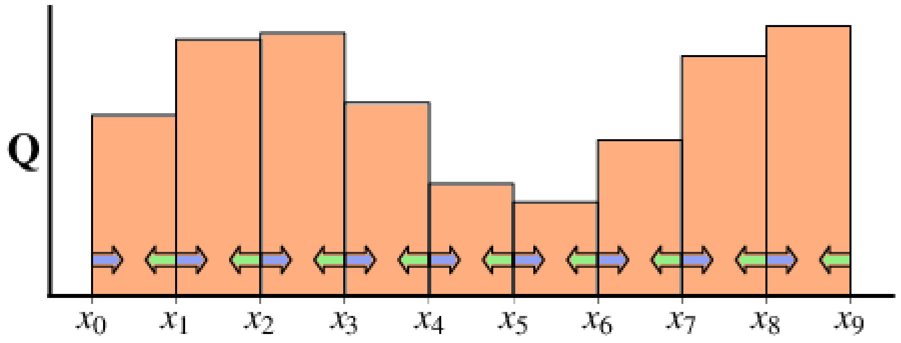
$$q_t + f(q)_x + g(q)_y = 0\,,$$

$$[q\ f(q)\ g(q)] = \begin{bmatrix} \rho & \rho u & \rho v \\ \rho u & \rho u^2 + p & \rho u v \\ \rho v & \rho u v & \rho v^2 + p \\ E & (E+p)u & (E+p)v \end{bmatrix},$$

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho(u^2 + v^2)$$

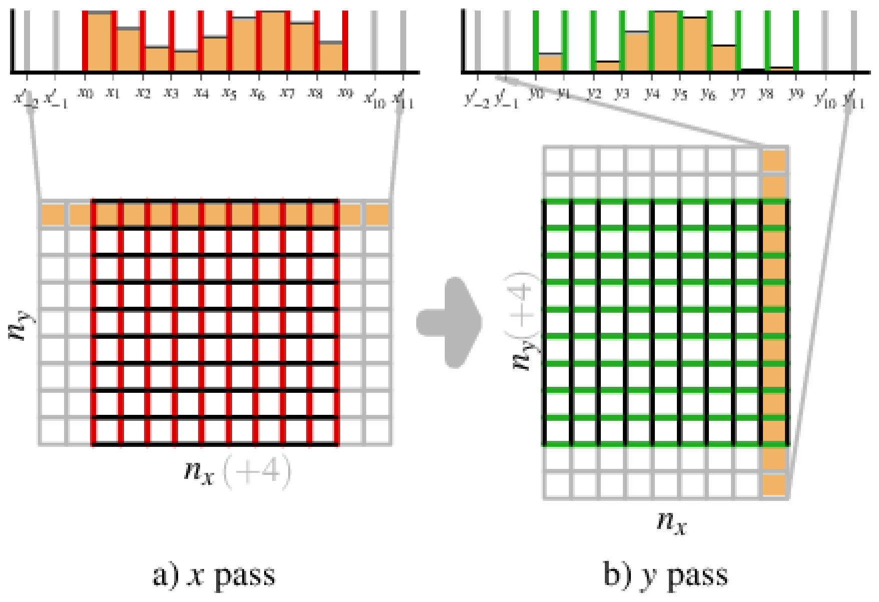
Godunov’s scheme

- Classically hyperbolic nonlinear system
  - Shocks expected!
  - Need integral form
- Solve with Godunov’s method
  - Finite Volume representation
    - Piecewise-constant (vector) state in ea. cell
  - Estimate state for left, right of ea. cell interface
  - Compute fluxes between cells based on that
    - Riemann problem
  - Explicitly integrate contributions for each cell
  - “Reconstruct-evolve-average”



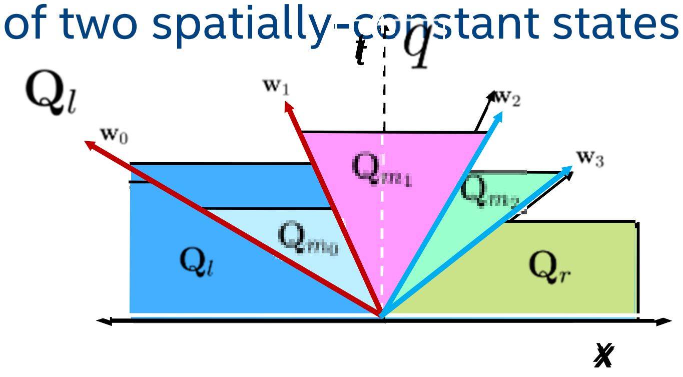
Dimensional splitting

- Above scheme is 1-dimensional
- For 2+ spatial dimensions, use *dimensional splitting*
  - Form of operator splitting
- Solve *x*-interfaces to intermediate state, then solve *y*-interfaces using that state to next timestate
- Same kernel, different access patterns

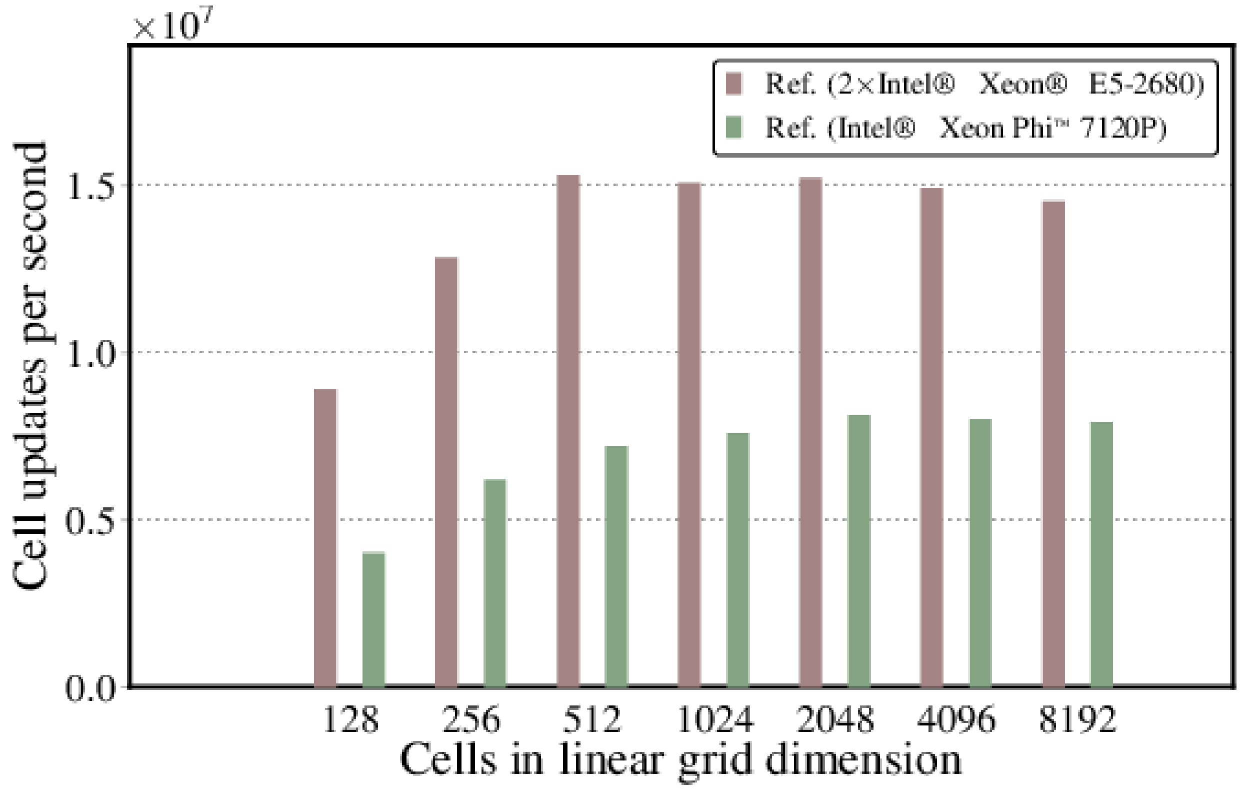


Riemann solver throughput

- Explicit Euler integration
  - Trivial compute
- Flux computation requires that we solve *2(n+1)* (independent) *Riemann problems*
- The Riemann problem is a “zoom in” of two spatially-constant states
  - How do they evolve?
  - This gives flux
- Complex structure
  - Use Newton-Raphson to solve



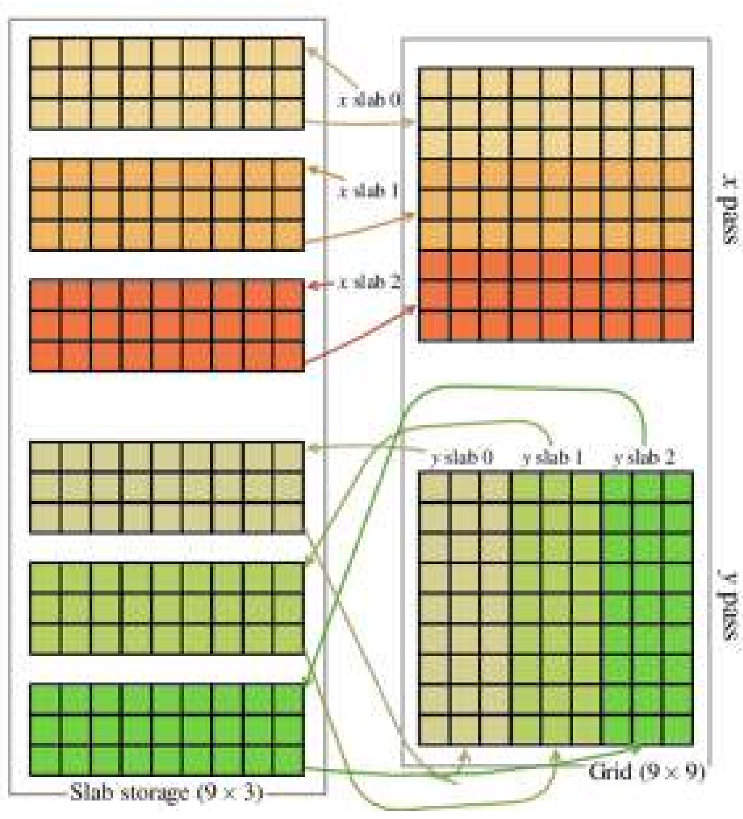
Our baseline: The reference code



Making it faster: Better threading, less data

Problem I: Bad threading, too much state

- Original code used ‘slabs’ for compute
  - Copy slice of grid to buffer, do all compute in buffer, copy back
- The good:
  - Vectorization/code path for slabs always the same
- The bad:
  - Lots of data movement (transpose in y!)
  - Parallelism restricted by slab size

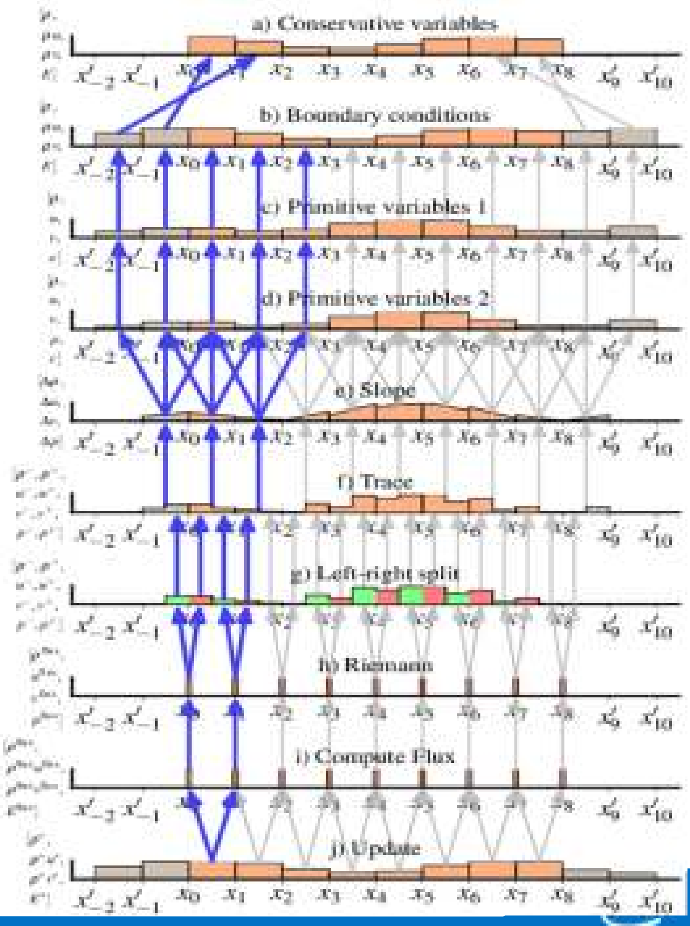


Problem I: Bad threading, too much state

- Original code does each step across all cells in slab, synchronizes
  - No race conditions
  - Ballooning working set

```
make_boundary();
constoprim();
equation_of_state();
slope();
trace();
qleftright();
riemann();
cmpflx();
updateConservativeVars();
```

Each applied to all cells in slab inside function, sync in between

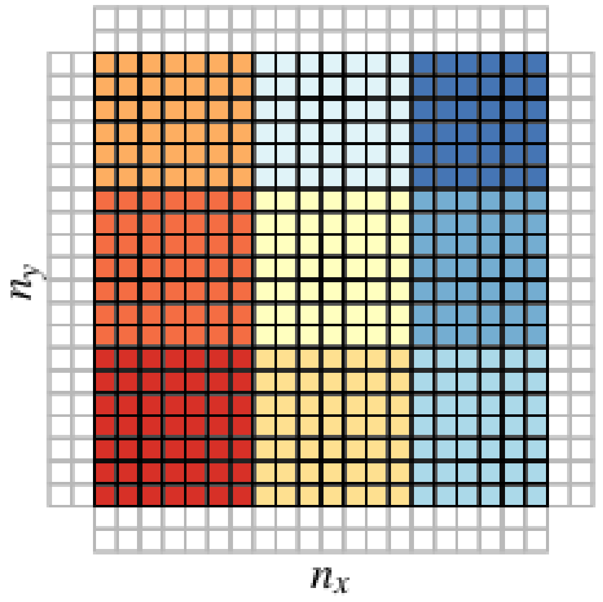


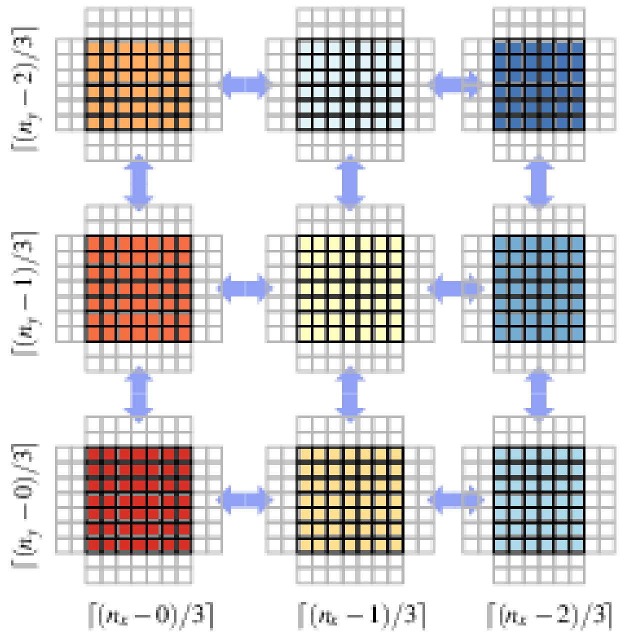
Problem I: Bad threading, too much state

- Stablity
  - Advance by dt in stability region
  - dt < dx/maxspeed
    - dx constant
    - Maxspeed largest wavespeed among cells
- We must find maximum speed across all cells
- Special, low-intensity pass to compute at each step
  - Bandwidth hog
  - Thread control overhead

Optimization I: Tiling decomposition, narrowed updates

- Decompose domain into exclusive tiles
  - Locality (NUMA)
  - Communication control
  - Local sync
- Surprisingly simple
  - Each tile looks like a whole domain; boundaries become communication





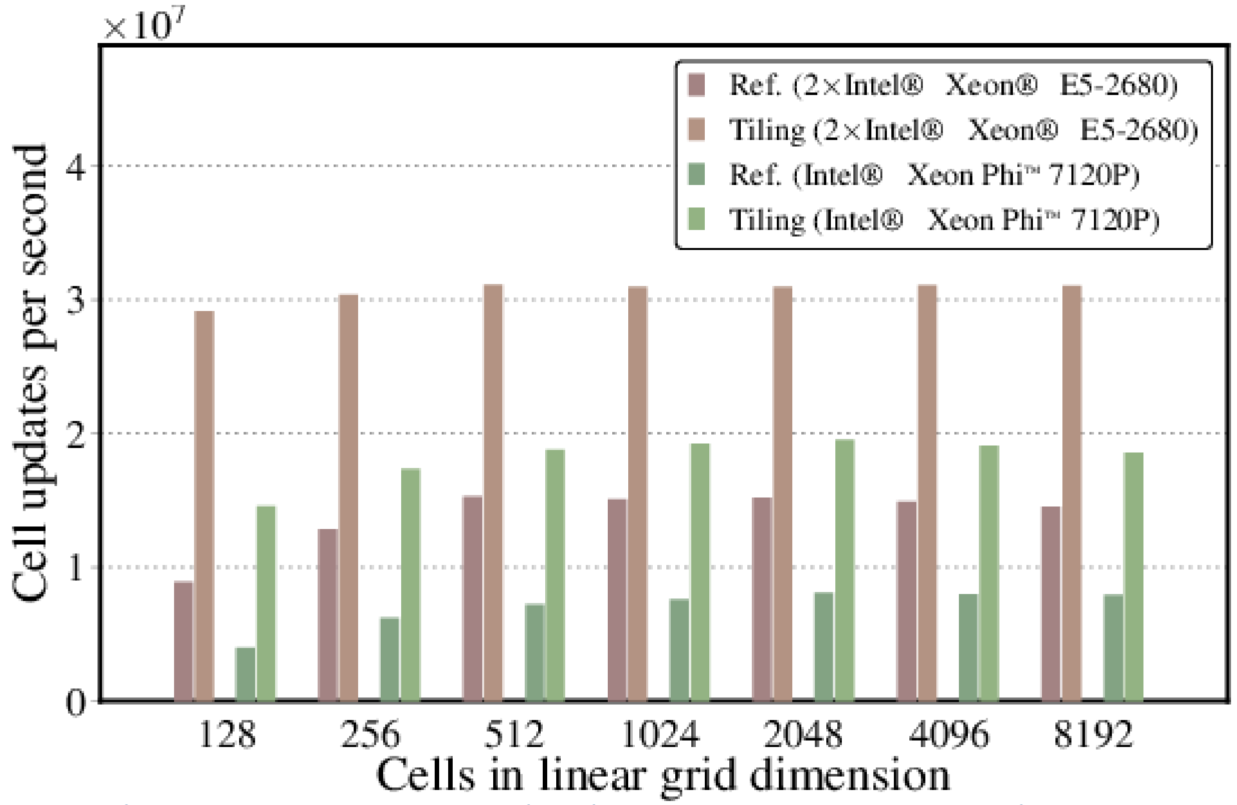
$\lceil (n_y - 2)/3 \rceil$
$\lceil (n_y - 1)/3 \rceil$
$\lceil (n_y - 0)/3 \rceil$
$\lceil (n_x - 0)/3 \rceil$
$\lceil (n_x - 1)/3 \rceil$
$\lceil (n_x - 2)/3 \rceil$

Optimization I: Tiling decomposition, narrowed updates

- Use data flow analysis to reduce working set
  - Compute only what is needed to update a cell, save what is needed for next cell
- 'Rolling updates'; operating on each 1D section per thread minimizes redundant compute
- For stability: combine speed computation with update stage

```
for(int i = 0; i < n[d]; ++i) {
  make_boundary(i);
  constoprim(i);
  equation_of_state(i);
  slope(i);
  trace(i);
  qleftright(i);
  riemann(i);
  cmpflx(i);
  updateConservativeVars(i);
}
```

Optimization I results



Making it faster: Less & cheaper math

Problem II: Lots of divisions

- Original code used lots of divisions and sqrts
  - Divisions can take 40+ cycles in DP
  - Less throughput in SIMD

```
for (iter = 0; iter < Hniter_riemann ; iter++) {
    if (goon) {
        double wwl, wwr;
        wwl = sqrt(cl_i * (one + gamma6 * (pstar_i - pl_i) / pl_i));
        wwr = sqrt(cr_i * (one + gamma6 * (pstar_i - pr_i) / pr_i));
        double ql = two * wwl * Square(wwl) / (Square(wwl) + cl_i);
        double qr = two * wwr * Square(wwr) / (Square(wwr) + cr_i);
        double usl = ul_i - (pstar_i - pl_i) / wwl;
        double usr = ur_i + (pstar_i - pr_i) / wwr;
        double delp_i = MAX((qr * ql / (qr + ql) * (usl - usr)), (-
pstar_i));
        pstar_i = pstar_i + delp_i;
        // Convergence indicator
        double uo_i = DABS(delp_i / (pstar_i + smallpp));
        goon = uo_i > PRECISION;
    }
```

Optimization II: Cache reciprocals, use algebra

- rcp instruction is *much* faster than division
  - Usually acceptable for accuracy
- Cache reciprocals and square roots (rho and c, in particular)
  - Rolling update makes this less memory-intensive (constant storage overhead)

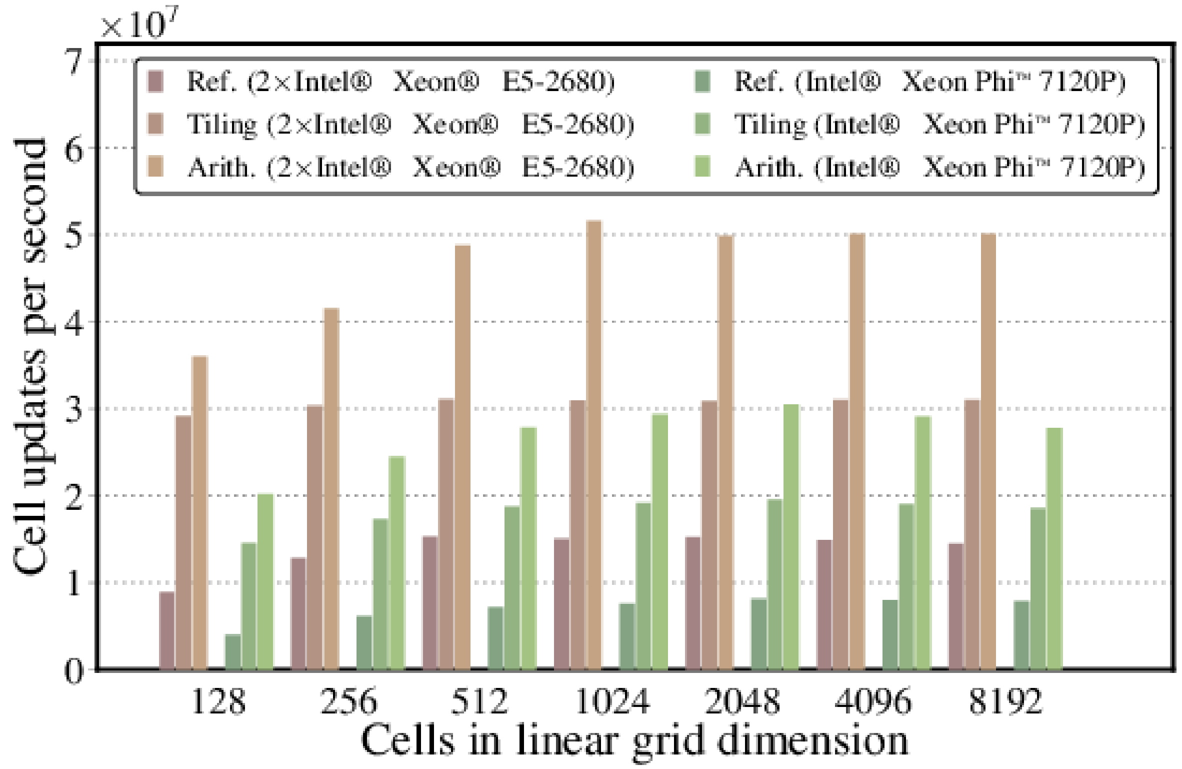
```
const REAL_T inv_rho_0 = rcp(rho_0);
const REAL_T c_0 = std::max(my_sqrt(std::abs(GAMMA * p_0 * inv_rho_0)), SMALLC);
const REAL_T ushock = w_0 * inv_rho_0 - sgnm * u_0;
```

Optimization II: Cache reciprocals, use algebra

- Algebraic tweaking of Riemann terms to reduce divs

$$
\begin{aligned}
\Delta p^* &= \frac{q_r q_l \left(u_l^* - u_r^*\right)}{\left(q_r + q_l\right)} \\
&= \frac{\frac{2{w'_r}^3}{{w'_r}^2 + c_r} \frac{2{w'_l}^3}{{w'_l}^2 + c_l} \left(u_l - \frac{p^* - p_l}{w'_l} - u_r - \frac{p^* - p_r}{w'_r}\right)}{\frac{2{w'_r}^3}{{w'_r}^2 + c_r} + \frac{2{w'_l}^3}{{w'_l}^2 + c_l}} \\
&= \frac{2{w'_r}^2 {w'_k}^2 \left(w'_r w'_k \left(u_l - u_r\right) - w'_l \left(p^* - p_r\right) - w'_r \left(p^* - p_l\right)\right)}{{w'_r}^3 \left({w'_l}^2 + c_l\right) + {w'_l}^3 \left({w'_r}^2 + c_r\right)}
\end{aligned}
$$

Optimization II results



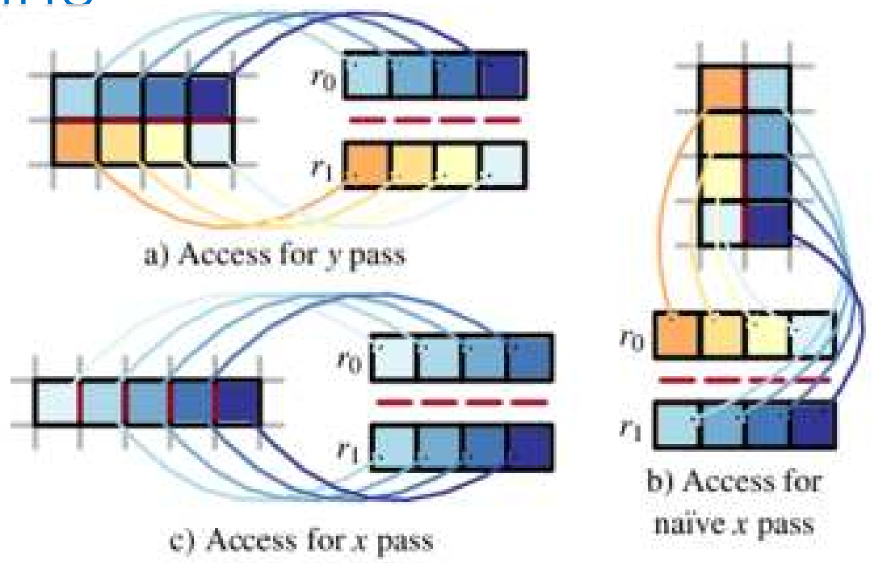
Making it faster: Vectorization

Problem III: Poor vector efficiency

- Vector units allow same work, fewer instructions
  - Overhead when data not packed
  - Overhead for control divergence
- Placing #pragma simd in existing code might not always work well
  - Compiler must be conservative about assumptions
- Some architectures have penalty for un-aligned data
- Vectorization can expose bandwidth (arithmetic intensity drops precipitously!)

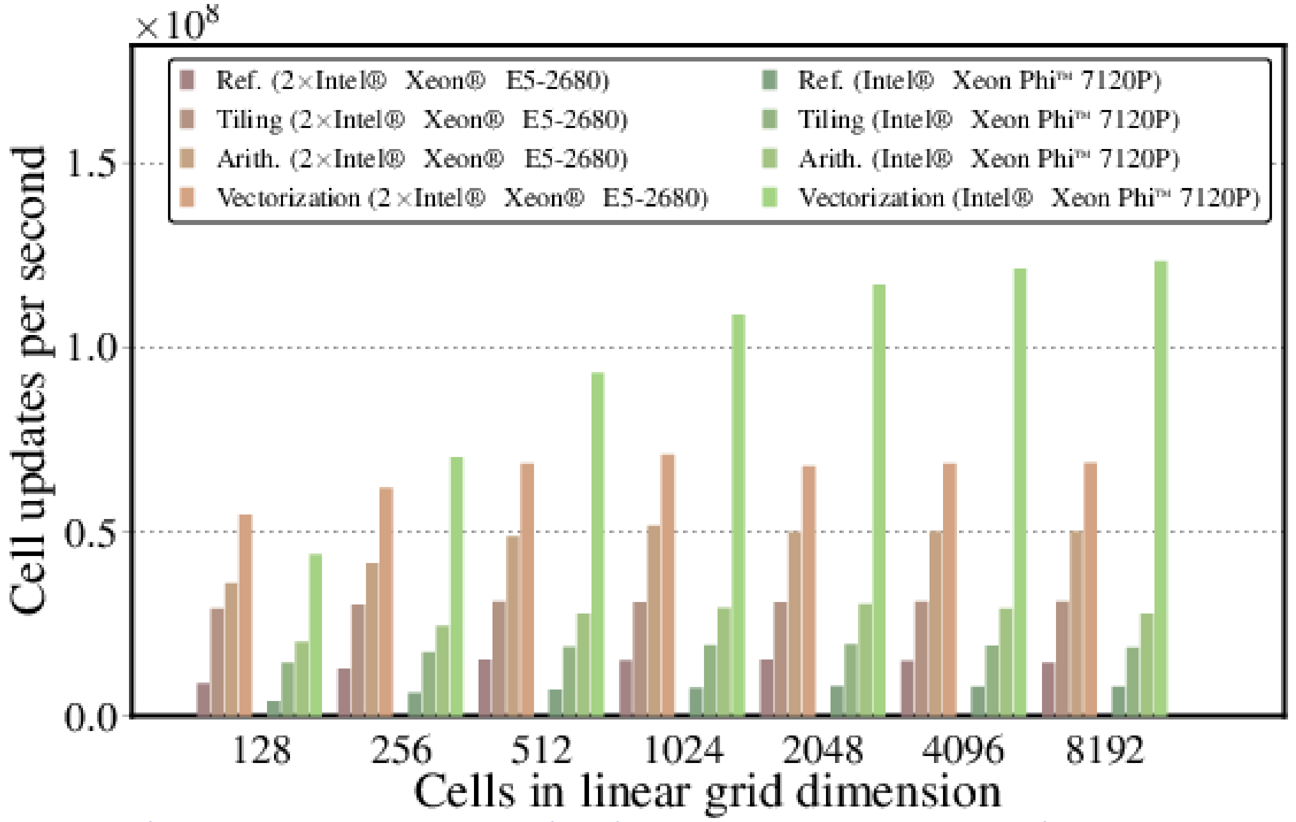
Optimization III: Vector thinkin

- First: know how SIMD *should* be used
  - Worry about how to achieve it later
- Here, y-passes are easy (two adjacent y-rows are packed)
- X-passes are tricky
  - We can gather…
  - Or do shifting
- C++ SIMD classes unify serial and vector code
- Two intrinsic-based functions rotate reused data in SIMD



```
void rotate_left_wm2(F64vec8 *v0, const F64vec8 v1)
{
    static const I32vec16 shift2(11, 10, 9, 8,  7,  6,  5,  4, \
                                  3,  2, 1, 0, 15, 14, 13, 12);
    *v0 = _mm512_permutevar_epi32     (              shift2,*v0);
    *v0 = _mm512_mask_permutevar_epi32(*v0, 0xFFF0U, shift2, v1);
}

void rotate_left_wm1(F64vec8 *v0, const F64vec8 v1)
{
    static const I32vec16 shift1(13, 12, 11, 10, 9, 8,  7,  6, \
                                  5,  4,  3,  2, 1, 0, 15, 14);
    *v0 = _mm512_permutevar_epi32     (              shift1, *v0);
    *v0 = _mm512_mask_permutevar_epi32(*v0, 0xFFFCU, shift1, v1);
}
```

Optimization III results



Summary

Overall progress

- Xeon Phi started out looking bad
  - ~0.5x 2-socket Xeon performance
- As it turns out, Xeon wasn't doing well either…
- With systematic optimization, as much as
  - 12x on Xeon Phi
  - 5x on Xeon

Learnings

- Adding pragmas, crossing fingers rarely solves the problem
  - No silver bullet
- Consider how hardware *should* be applied
  - Then worry about how; the realization is often simple
- PhD in EE not required
  - A working model of major components is enough to tap resources
- A rising tide lifts all boats
  - Xeon and Xeon Phi benefit from the same optimizations

Summary

- Knights Landing is a high-throughput successor to the first Xeon Phi
  - Socketable, bootable processor with access to large amounts of RAM
  - Greatly improved single-thread performance
  - Very high bandwidth, flexible MCDRAM
  - Power-efficient
  - Optional on-chip interconnect (Omni Path)
- Much of Knights Landing's throughput comes from parallelism
  - Codes will need to be modernized to fully exploit the features of the chip
  - The current generation Xeon Phi has parallelism at similar scales and is the best proxy for performance on Knights Landing

Summary

- Peak performance is not automatic
- Parallelism of all types requires forethought and careful design
- Design and coding modernization efforts invested now will pay dividends in future hardware
- Evolving standards like OpenMP help realize modernization
- Numerous tools (Intel®SDE, memkind, Intel® Composer XE) exist today to help test Knights Landing, in addition to other Intel® performance analysis tools

Q&A

intel®

Legal Disclaimers

Legal Disclaimers

Legal Disclaimers

Legal Disclaimers