An Investigation of Compiler Vectorization on Current and Next-generation Intel Processors using Benchmarks and Sandia's SIERRA Applications

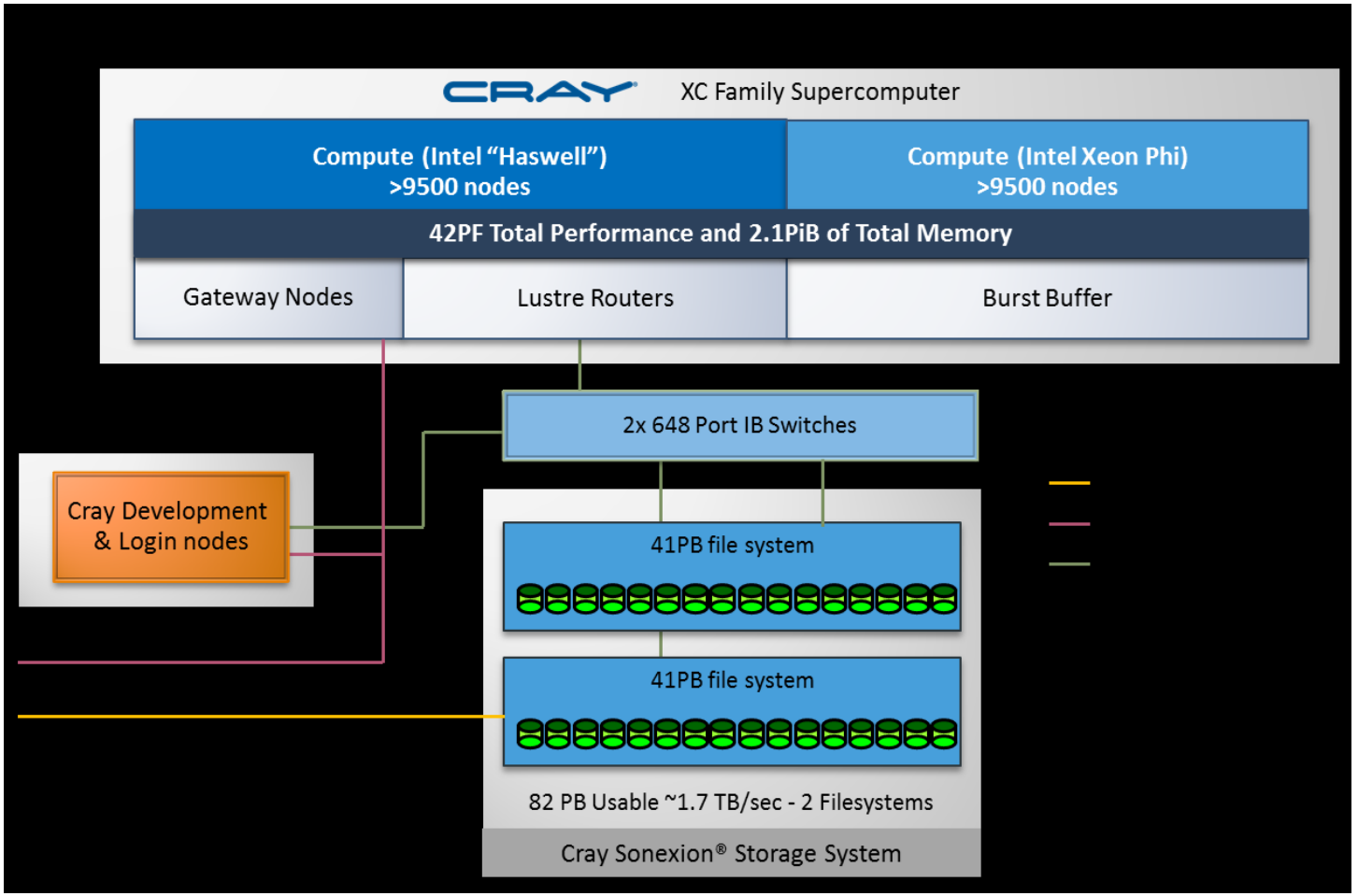
Mahesh Rajan[1], Doug Doerfler[2], Mike Tupek[1], Si Hammond[1]
[1]Sandia National Laboratories, [2]Lawrence Berkeley National Laboratory
Cray User Group Meeting, April 26-30, 2015, Chicago, IL

This work was supported in part by the U.S. Department of Energy. Sandia is a multi program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States National Nuclear Security Administration and the Department of Energy under contract DE-AC04-94AL85000.

Motivation

- Acquisition of Trinity (NNSA's ATS-1) by ACES (SNL & LANL Partnership)
  - >9500 nodes with Intel Haswell; SIMD unit:AVX2
  - > 9500 nodes of Intel Knights Landing (KNL); SIMD unit AVX-512F(AVX3.1)
  - Study vectorization to realize performance potential on Trinity
- Evaluate Cray, Intel and GNU compilers (auto-vectorization)
  - Study TSVC benchmark
  - Study LCALS benchmark
- Investigate approaches with real SNL SIERRA Mechanics kernels
  - Impact of data layout
  - Compiler auto-vectorization limitations and effective usage
  - Design and performance of a specially developed SIMD library

ACES (Sandia, LANL Partnership) new Advanced Technology System: Trinity



Processor Performance Trends

(from Eric Welch & James Evans; Multiple Processor Systems, 2013)

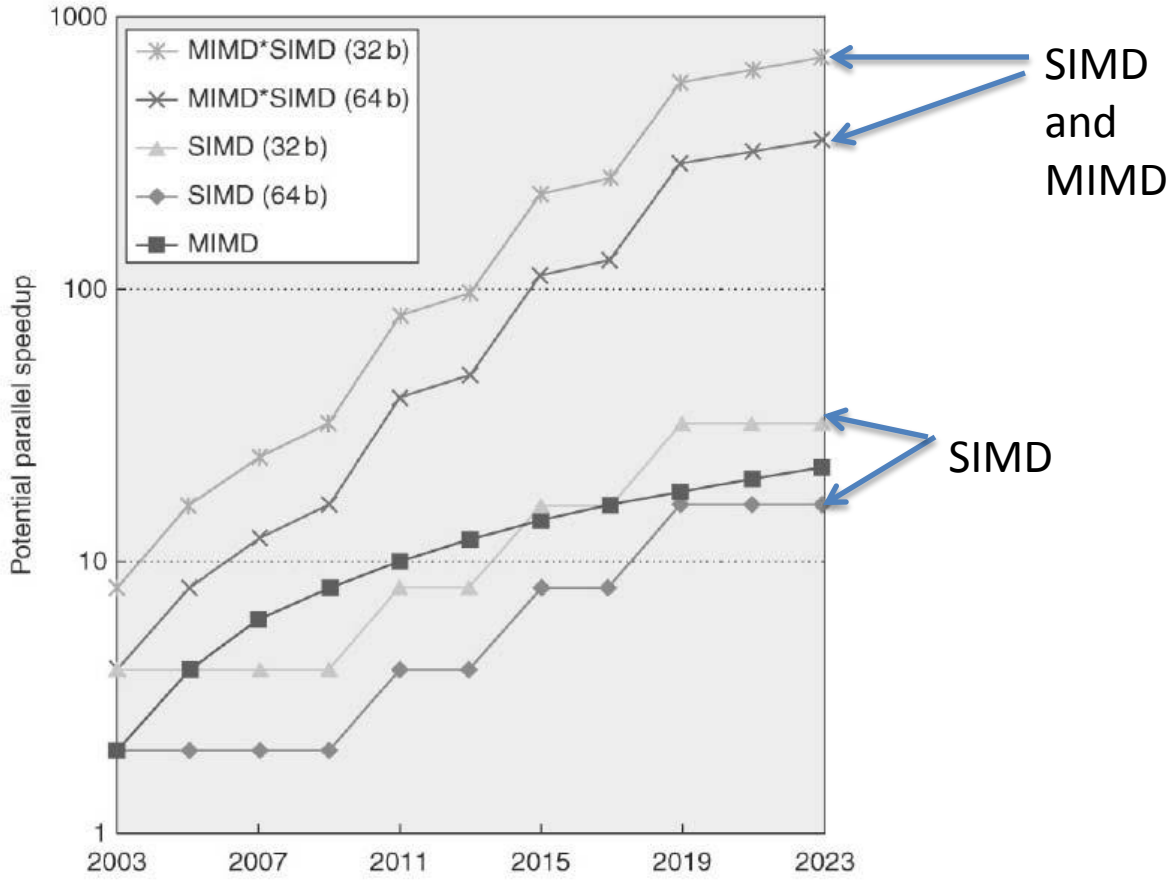
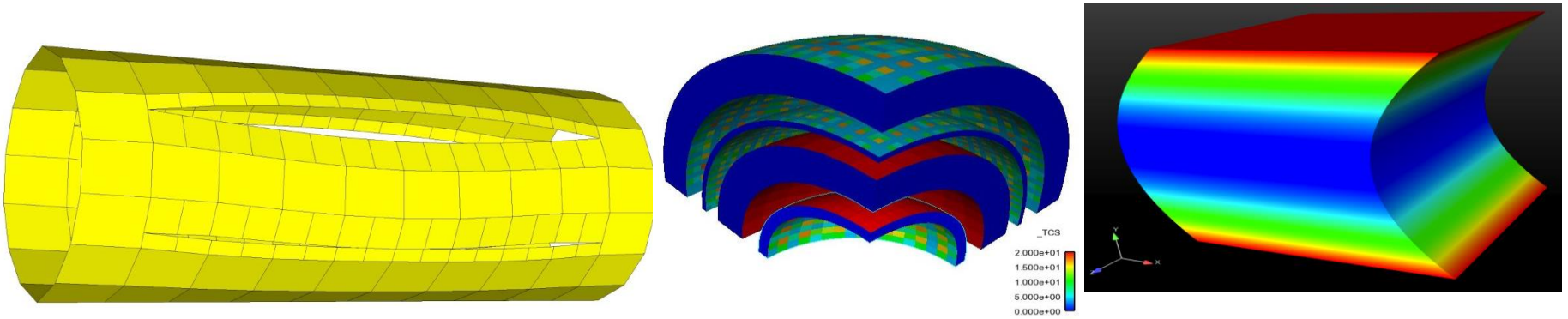


Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

Vectorization Kernels from SIERRA/SM (Solid Mechanics)



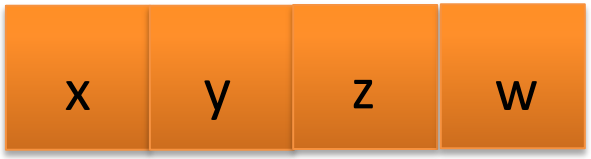
- **A general purpose massively parallel nonlinear solid mechanics finite element code for explicit transient dynamics, implicit transient dynamics and quasi-statics analysis**
- **Built upon extensive material, element, contact and solver libraries for analyzing challenging nonlinear mechanics problems for normal, abnormal, and hostile environments**
- **Similar to LSDyna or Abaqus commercial software systems**

SIERRA Mechanics; need and approaches

- Compiler Auto-Vectorization
  - For simple loops, compilers auto-vectorizes;
    - Example:
      - for (int i=0; i < N; ++i) {
        a[i] = b[i] + c[i] * d[i];
        }
- For "Complicated" loops compilers typically **will not** auto-vectorize
- SIERRA Solid Mechanics kernels have loops that are > 200 lines
  - Tensor33 multiply (symmetric **x** asymmetric)
  - Eigenvectors
  - Constitutive law evaluations
- Use SIMD vector intrinsics (low level functions):
  - Developed SIERRA **SimdLib with Intrinsics (SLI)** for easy port to different architectures

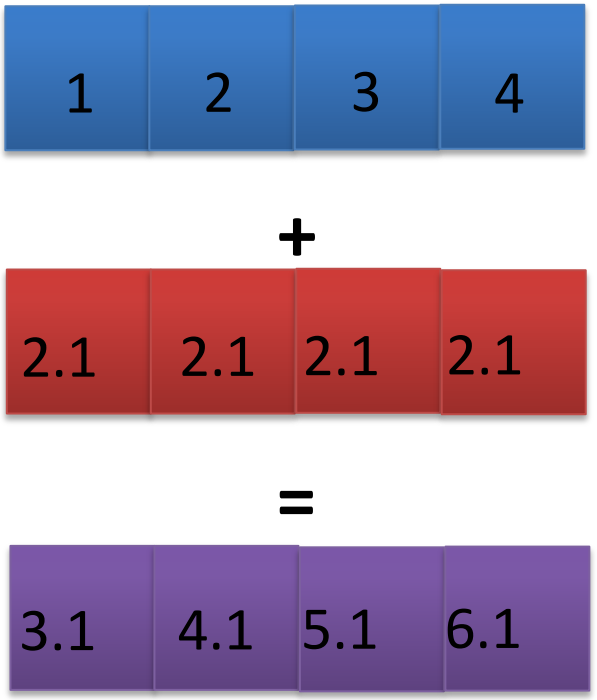
AVX Intrinsics

__m256d (4 doubles)



Compute {1,2,3,4} + 2.1:

```
double x[4] = {1,2,3,4};
__m256d a = _m256_loadu_pd(x);
__m256d b = _m256_set1_pd(2.1);
__m256d c = _m256_add_pd(a,b);
double result[4];
_m256_store_pd(result,c);
```



Platforms, Processors and compilers used in this study

Processor	Platform Name	Specification/CPU
Ivy Bridge	Edison, Morgan04	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Haswell	Mutrino, Shephard	Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
KNC	Corner, Morgan04	Intel(R) Xeon(R) Phi CPU @ 1.238 GHz

Compiler Versions used:

- Intel 15.0.2
- GNU gcc 4.9.2
- Cray compilers under Cray Programming environment 5.2.40

TSVC (Test Suite for Vectorizing Compilers) Benchmark

- Originally developed by Callahan, et. al. (1988) in Fortran
- Extended, and converted to *C* by Maleki, et. al.
- A total of 151 loops (Single Precision Floats)
- It provides a large collection of basic loops that could be found in scientific HPC codes
- Forms a good basis for investigating compiler auto-vectorization capabilities

Our Method for Determining "vectorization"

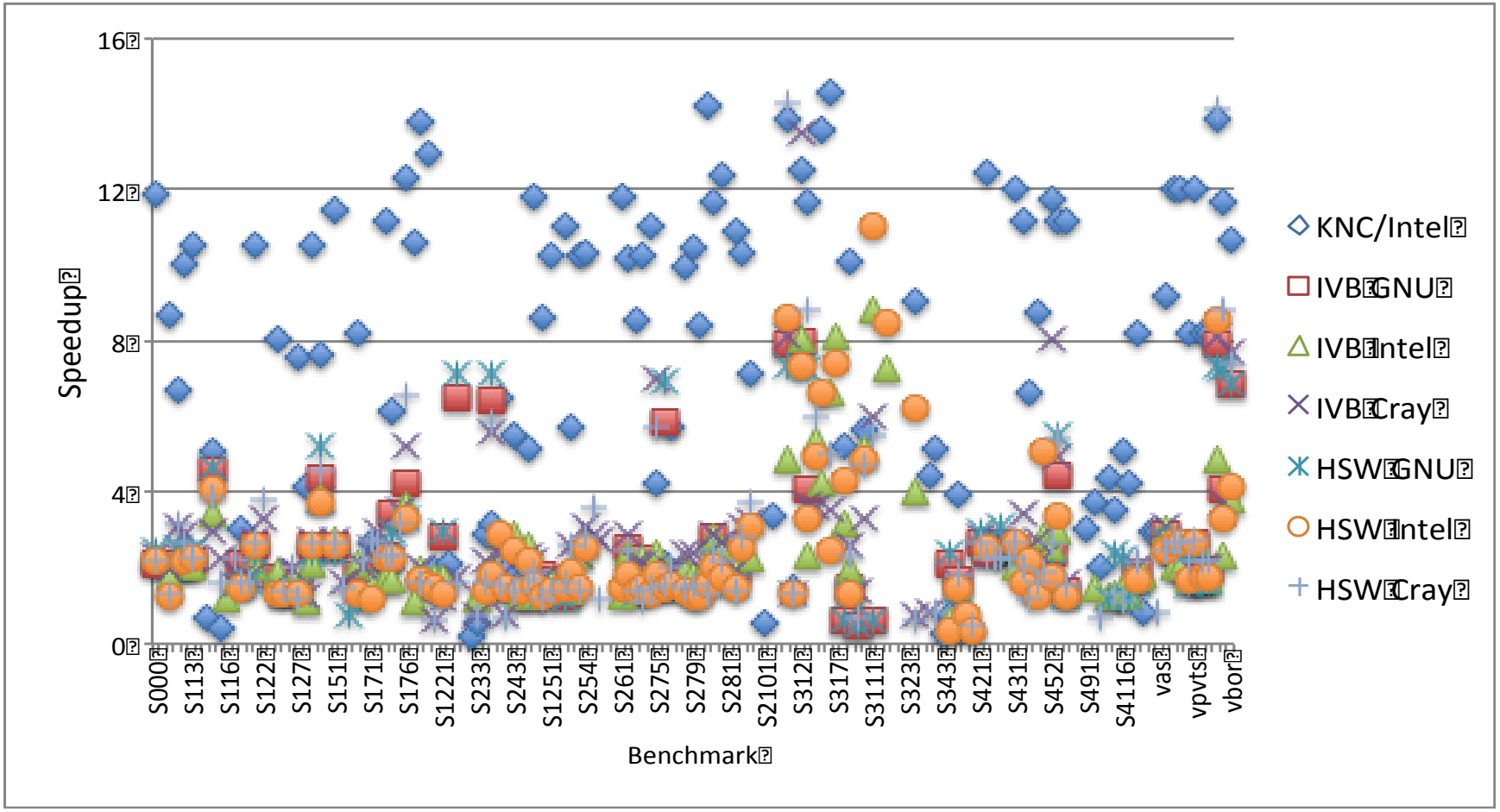
- Taken from Maleki paper
- Baseline measurement: use no vectorization flag (e.g. –no-vec) but include optimization (-O3)
- Measurement with vectorization: Include vectorization flag (e.g. –mavx) and optimization (-03)
- Speedup = (time w/o vectorization) / (time w/vectorization)
  - Greater than 1.5 is a "vectorized"
  - Less than 0.85 is "vectorized" but a slowdown
  - KNC max speedup=16; Ivy Bridge max=8; Haswell max=16 (w/fma)
- Benchmarks were modified to ensure array alignment on the appropriate SIMD width for the architecture
  - 32 bytes (256 bits) for Ivy Bridge and Haswell
  - 64 bytes (512 bits) for KNC

TSVC Results

	KNC	Ivy Bridge w/AVX			Haswell w/AVX2		
	Intel	GNU	Intel	Cray	GNU	Intel	Cray
vectorized	111	61	99	101	63	91	102
speedup	103	58	96	96	59	88	93
slowdown	8	3	3	5	4	3	9
average speedup	8.04	2.87	2.47	2.80	2.82	2.60	2.88
total time	177.82	21.41	17.15	16.53	17.29	14.45	13.56

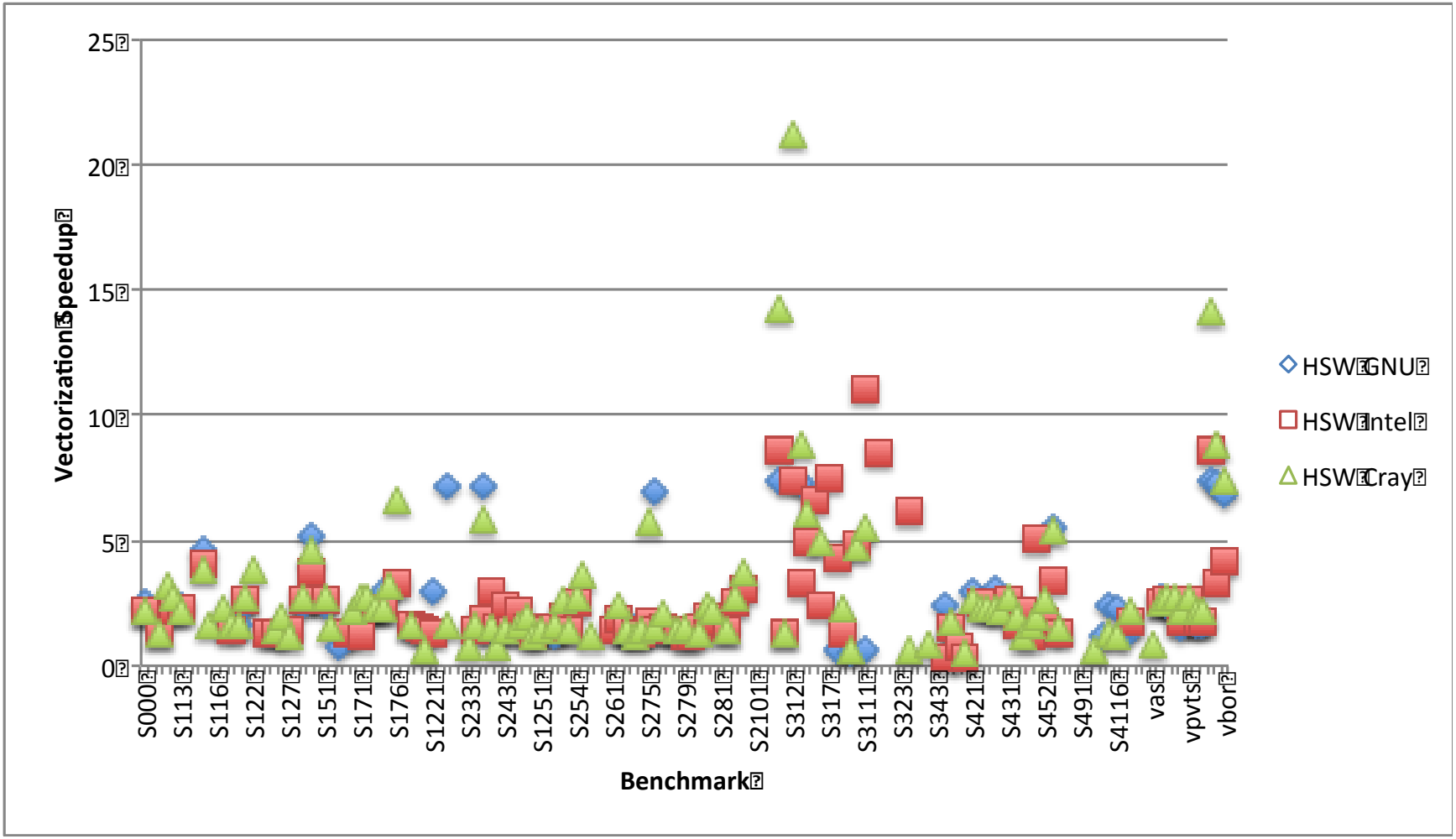
Intel & Cray on Ivy Bridge & Haswell showed a speed up for 66% of the loops; GNU 41%
From *total time* metric for Haswell: Cray faster by 1.07X of Intel and 1.28X of GNU
KNC total times are poor because of clock speed and not using minimum of 2 threads

TSVC Results



Speedup values > 16 not displayed

TSVC: Haswell Only



LCALS (Livermore Compiler Analysis Suite) Benchmark

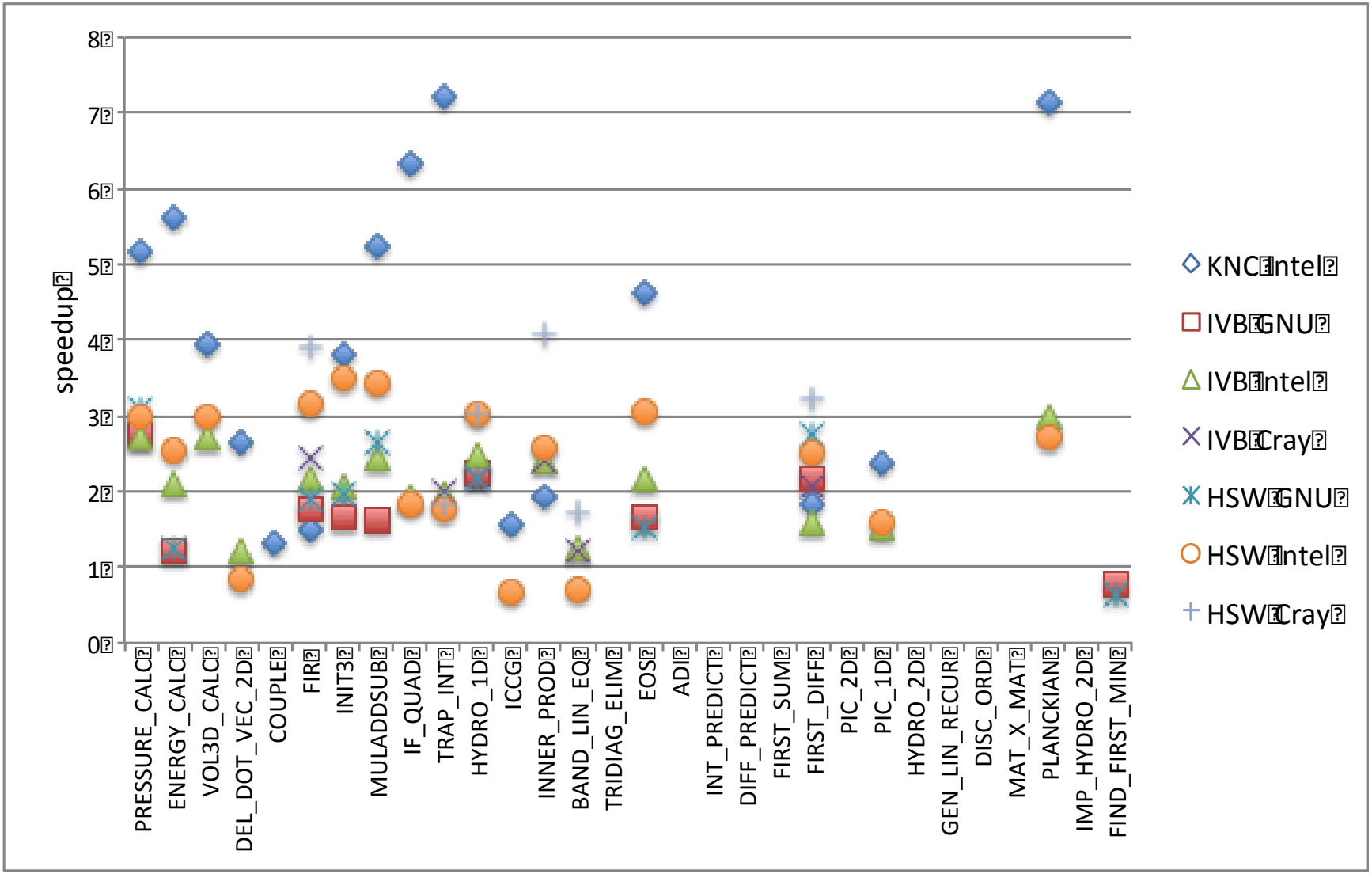
- Developed by Rich Hornung (LLNL)
- Represents **30** loops and kernels taken and/or derived from real codes
- Double Precision Floats
- Three variants
  - "Raw": C/C++ for-loop syntax -> used for this study
    - Subset A: loops used in application codes
    - Subset B: used to illustrate compiler optimization issues
    - Subset C: extracted from Livermore Loops in C by Steve Langer
  - Other variants include OpenMP, functors and C++11 lambda functions -> NOT utilized for this study

LCALS Results

	KNC	Ivy Bridge w/AVX			Haswell w/AVX2		
	Intel	GNU	Intel	Cray	GNU	Intel	Cray
vectorized	17	9	16	6	9	17	6
speedup	17	8	16	6	8	14	6
slowdown	0	1	0	0	1	3	0
average speedup	3.80	1.77	2.12	2.07	2.00	2.36	2.98
total time	5.57	0.83	0.59	0.87	0.65	0.42	0.65

Intel compiler vectorizes 53% on Ivy Bridge, 57% on Haswell; GNU 30%; Cray 20%;
Cray compiler showed good speed up on Haswell of the vectorized loops, 2.98X;

LCALS Results



SIERRA Kernels Chosen for this study

- ***Eigenvector kernel:***
  - ***Computes eigenvectors and eigenvalues of a symmetric 3x3 matrix***
  - ***Computation based on analytic formula***
  - ***Kernel code uses conditionals and trigonometric function evaluations***
- ***Elasticity Kernel:***
  - ***Computes mechanical stress from stretching tensor and rotation tensor ; all 3x3 matrices; rotation tensor non-symmetric***
  - ***Uses material properties Bulk Modulus and Shear Modulus***
  - ***Kernel code relatively straight forward; no conditionals; most complicated math is a cube-root***
- ***Plasticity Kernel:***
  - ***Computes stress tensor from strain-rate tensor and old-stress tensor (all symmetric 3x3 matrices); uses also an array of length 11 that stores the internal state history of the material***
  - ***Uses material properties Bulk Modulus ,Shear Modulus, Yield Stress, and Hardening Modulus***
  - ***Kernel code is complex a*****s it has structs with stride 11 (i.e. 11 doubles), has many inputs, has conditionals and even has a while loop at the inner most level to assess convergence of the material model's plastic strain updates**

Data structure layout investigated

AOS, SOA and SLI

Array of Structures **(AOS)**

Structure of Arrays **(SOA)**

SimdLib with Intrinsics **(SLI)** ; schematic SIMD Length=2

Sandia SIERRA/SM team's SIMDLIB

- Motivated by compiler limitations on complex loops
- Uses SIMD vector intrinsics
- Clever design using C++templates and *structs* to make it independent of platform and compilers (Portability a key design goal)
- Key components: "Doubles" struct, a "Bools" struct, and an integer valued vector-length
- At compile time for the target SIMD unit "Doubles" and "Bools" structs are then sized to the vector-length
- The most common mathematical operations (such as +,-,*,/,sqrt,<,<=,!=,&&,||,etc.) are overloaded to use the appropriate SIMD intrinsics on the data members of the "Doubles" and "Bools" structs

Ivy Bridge: SIERRA kernels speedup relative to AOS layout and no vectorization

	Eigenvector	Elasticity	Plasticity
AOS	1.62	1.01	0.99
AOS, IVDEP	1.67	1.61	0.98
SOA	1.09	0.99	0.70
SOA, IVDEP	2.45	2.19	0.71
SLI	2.27	1.86	1.80

Auto Vectorization requires implementing the kernel function as inline function in a header file and increase max inline size with flag:

-inline-max-total-size=10000

Haswell: SIERRA kernels speedup relative to AOS layout and no vectorization

	Eigenvector	Elasticity	Plasticity
AOS	1.80	1.00	0.97
AOS, IVDEP	1.74	1.37	0.97
SOA	0.90	0.99	0.58
SOA, IVDEP	2.53	2.45	0.59
SLI	2.03	1.79	1.54

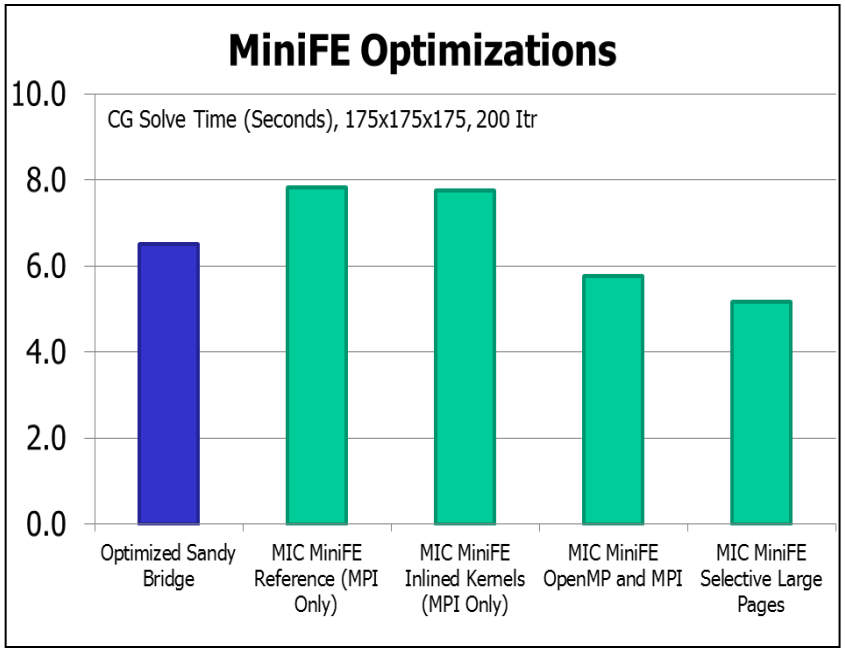
- Are prefetch instructions for compiled code the reason for SOA+IVDEP performance being better than the SLI performance?
  - Used CrayPat: ratio of the metric: *MEM_UOPS_RETIRED:ALL_LOADS* SimdLib/ SOA+IVDEP = 1.4;
  - Value close to run time ratio of SimdLib/ SOA+IVDEP = 1.38;
  - Also CrayPat metric that measures L2 prefetch hits: *L2_RQSTS:L2_PF_HIT* registered **3 times** higher value for SOA+IVDEP over Simdlib.
  - CrayPat metric that measures L2_RQSTS:L2_PF_MISS were nearly the same.

KNC: SIERRA kernels speedup relative to AOS layout and no vectorization

	Eigenvector	Elasticity	Plasticity
AOS	2.28	1.00	1.00
AOS, IVDEP	1.64	0.92	1.00
SOA	0.95	0.84	0.63
SOA, IVDEP	5.14	7.16	0.63
SLI	5.10	2.39	2.63

MiniApps on KNC

Application	miniFE	AMG	UMT	SNAP
% speedup with Vectorization	4.68%	6.52%	17.95%	19.52%



MiniFE Tuning:

- KNC performance, 23% slower than the front-end Sandy Bridge node
- Additional gains in performance were achieved by disabling transparent huge pages and using selectively large page allocations for vector data structures to lower TLB miss rates. These tuning measures improved the KNC performance by 33%
- Finally KNC exceeded FE Sandy Bridge by 20% (see figure)

Use of hardware counters on KNC; vectorization effectiveness

Investigated with a simple DGEMM matrix multiply benchmark:

Vectorization intensity defined as:
Vectorization Intensity = VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED

vectorization intensity measured for DGEMM = 7.84

Metric upper bound of 8. Values close 8 suggest efficient use of MIC's SIMD units.

However since the VPU_ELEMENTS_ACTIVE counter measures in addition to the double precision floating point instructions, vector load/stores from memory and instructions to manipulate vector mask registers this metric is misleading.

The fact that our measurements of this metric achieves close to the peak showing high vectorization intensity is misleading if our goal is to achieve high floating point operations throughput. The percentage of peak double precision floating point operations achieved with MKL DGEMM in this test is about 30%; Need **DP_OPS** counter!!

Conclusions

- The TSVC and LCAL benchmarks show a performance gain of 3X if the compute intensive kernels are vectorized
- Our need for SIERRA/SM SimdLib as typified by the plasticity kernel; compiler is unable to vectorize some complex loops even with pragmas.
- SimdLib designed for easy portability to processors with different lengths of the vector registers
- Compiler can indeed give the best performance when kernels have appropriate data structure and compiler vectorization is aided by pragma
- The importance of hardware performance counter measures to identify all aspects of effective use of the SIMD units is pointed out