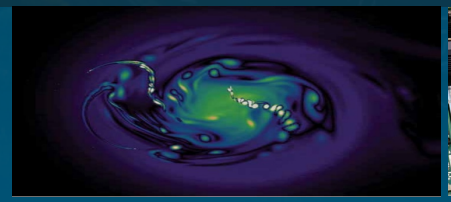
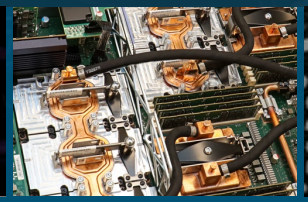
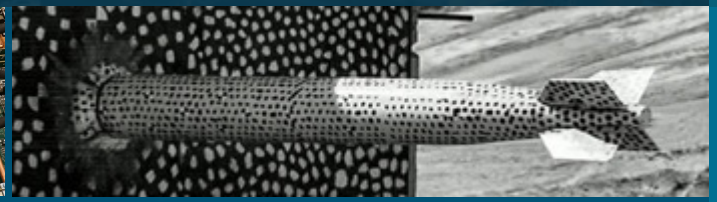
On the Use of Vectorization in Production Engineering Workloads







PRESENTED BY

Courtenay T. Vaughan (ctvaugh@sandia.gov)

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Introduction

We have been working on application porting, bring-up and optimization for Trinity Haswell/Knights Landing for three years

- Lots of experiences about how to make code run faster on our machines
- Been a useful way to drive application optimization

Getting to the stage of working on more optimization

- Focus on improving levels of vectorization because this helps drive good code design, memory access, compute efficiency etc.

This talk is about our analysis of applications in this context

Motivations and Background

Motivation for Studying Vectorization

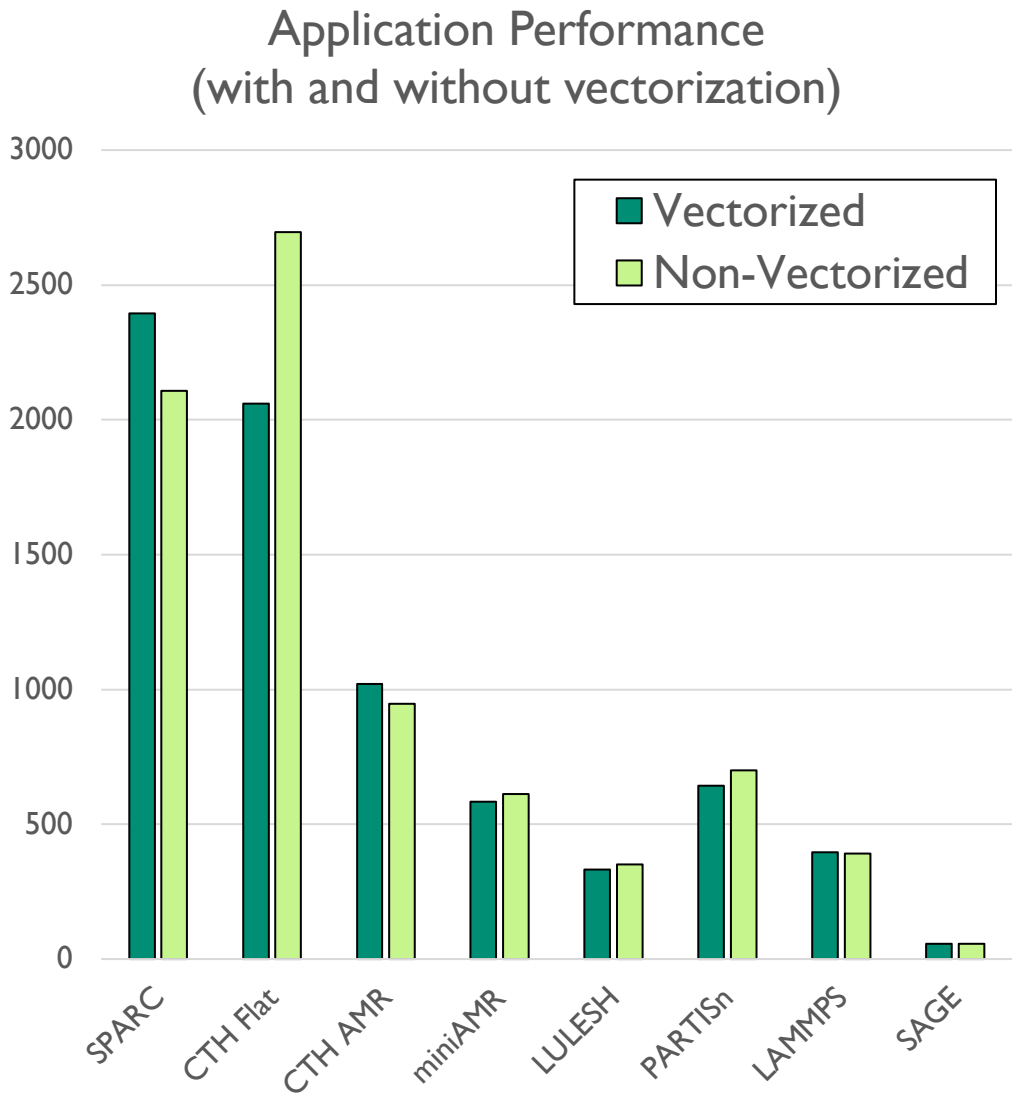
Deployment of Trinity (Cray XC) platform as LANL/SNL ACES partnership

- >9,000 nodes of dual-socket Haswell (dual-AVX256)
- >9,500 nodes of single-socket Knights Landing (dual-AVX512)
  - Where significant amount of application performance is available

Early experiences showed limited success of automatic vectorization by Intel compiler toolchain

- Compiler reports showed many loops refusing to vectorize
- Even when vectorizing often not getting performance delta
  - In some cases it appeared code would vectorize but we would use the scalar/remainder loops

Motivating Application Examples for Knights Landing



Application studies on Knights Landing showed small/no difference with and without vectorization enabled

- Why doesn't this show more improvement?
- Are codes not vectorizing?
- Or, are they vectorizing and not gaining?

Complex interaction of behavior for larger applications

Concern about compilers, libraries etc.

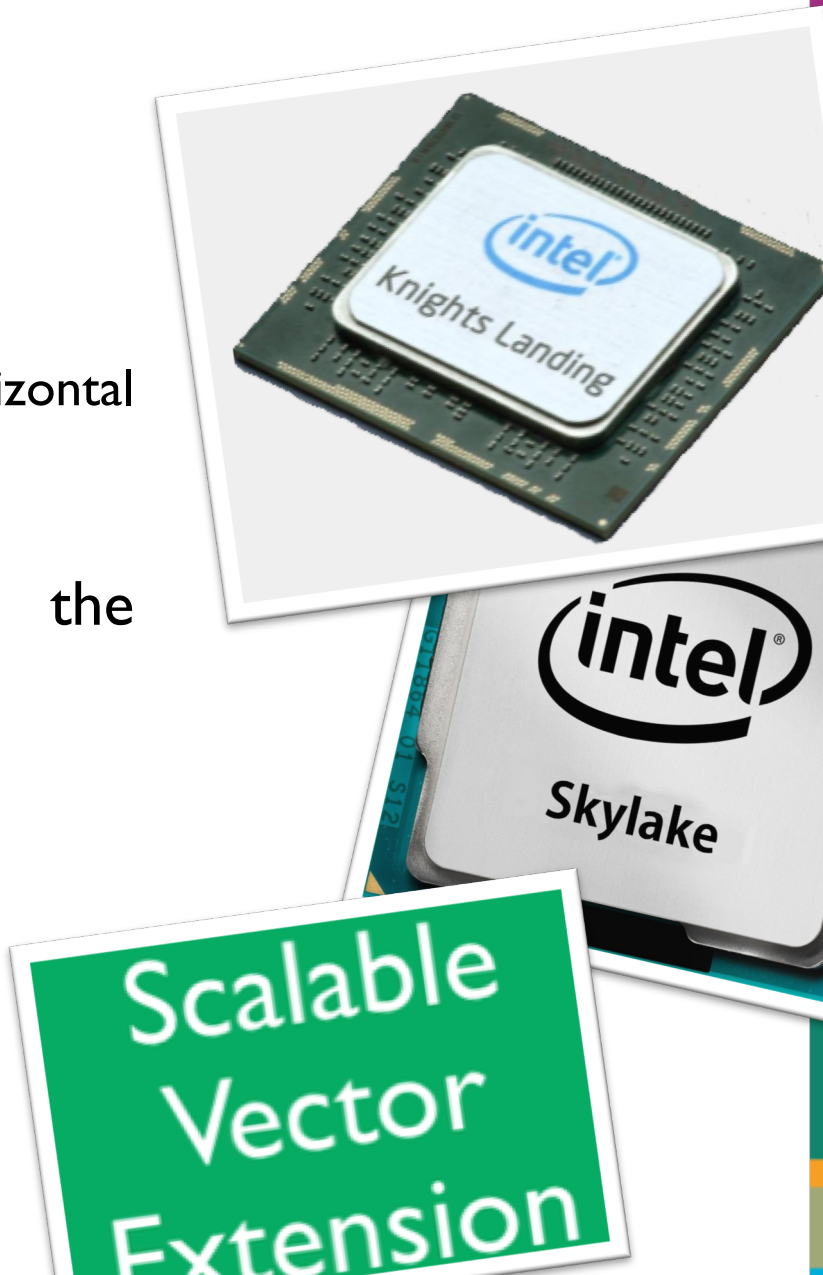
- Could we be getting much more from our platforms?

Motivation for Studying Vectorization

Complexity of hardware is growing
- AVX512 adds support for masked vector instructions
  - Now need to count the number of active lanes
- AVX256 and 512 adds more sophisticated instructions
  - Multiple operations per-lane per instruction (e.g. FMA, FMSUB, Horizontal Adds)

Growing use of vectorization in other products for the NNSA/ASC
- ARM, IBM POWER, implication of similar ideas for GPUs
- Want to ensure we have code that *can* vectorize

Want to make better compilers and toolchains
- Partner with vendors to make these better for our needs



Why is this Challenging?

Performance Counters on Haswell and Knights Landing are **not** sufficient

- No FLOP/s counting on Haswell (unbelievable)
- General performance counter support on KNL extremely limited
  - Particularly around wide-vector support (e.g. how do you count vectorized FLOP/s when also using masking)

- Compilers tell you things but sometimes not the complete truth
  - Vectorization reports are good but could be better
  - Drop to scalar loops
  - Appears that dependency analysis is not a perfected art (although getting better)

Vectorization Analysis Tools

APEX-Tesserae Tools

Need to develop in-house methods for counting operations and performing analysis of vectorization/application behavior

- Intention to analyze vectorization in codes
- But, can use this to look at memory access, instruction execution behavior etc

Complete visibility into instruction stream using binary instrumentation

- Limited to X86, but this is our main focus for Trinity work (and many platforms)
- Intel SDE (similar tool) does not allow us to customize the analysis
- Drive custom tool development using Intel's PIN framework
  - Works on all production platforms and KNL, good support for KNL

See: *Towards Accurate Application Characterization for Exascale*, S.D. Hammond, SAND2015-8051 (Technical Report, Sandia Nat. Labs), September, 2015

APEX-Tesserae Tools

Want to maintain highest-performance possible for analysis (otherwise this is hard to scale)

- At startup we perform an analysis of each program basic block (we perform our own blocking through instruction-by-instruction analysis)
- Generate a ”summary” for the block of instruction behavior
- Add a hook to atomically update counts when each block is encountered
  - Allows high-performance execution in multi-threaded environments

Works fine except for masking register analysis which require dynamic inspection

- Add a hook only for vector instructions which use masking, perform a dynamic population count on the mask register and atomically keep counts

Application Analysis

Application Analysis

Application	Languages	Domain
CTH Flat Mesh	Fortran 77 + Fortran 90 (tiny bit of C), O(1M)	Hydrodynamics, Structured
CTH AMR Mesh	Fortran 77 + Fortran 90 (tiny bit of C), O(1M)	Hydrodynamics, Structured
MiniAMR (using AMR Mesh)	C, O(10K)	Mini-Application, Hydrodynamics, Structured, Simple Stencil
LAMMPS	C, C++, O(10K)	Molecular Dynamics
LULESH	C, C++98, O(1K)	Mini-Application, Unstructured Hydrodynamics
SAGE	Fortran, O(10K)	Hydrodynamics, Unstructured
SPARC	C++11, Trilinos O(~1M incl. Trilinos)	CFD Modeling

Analysis Results

Taken each application and configured a suitable input deck/problem

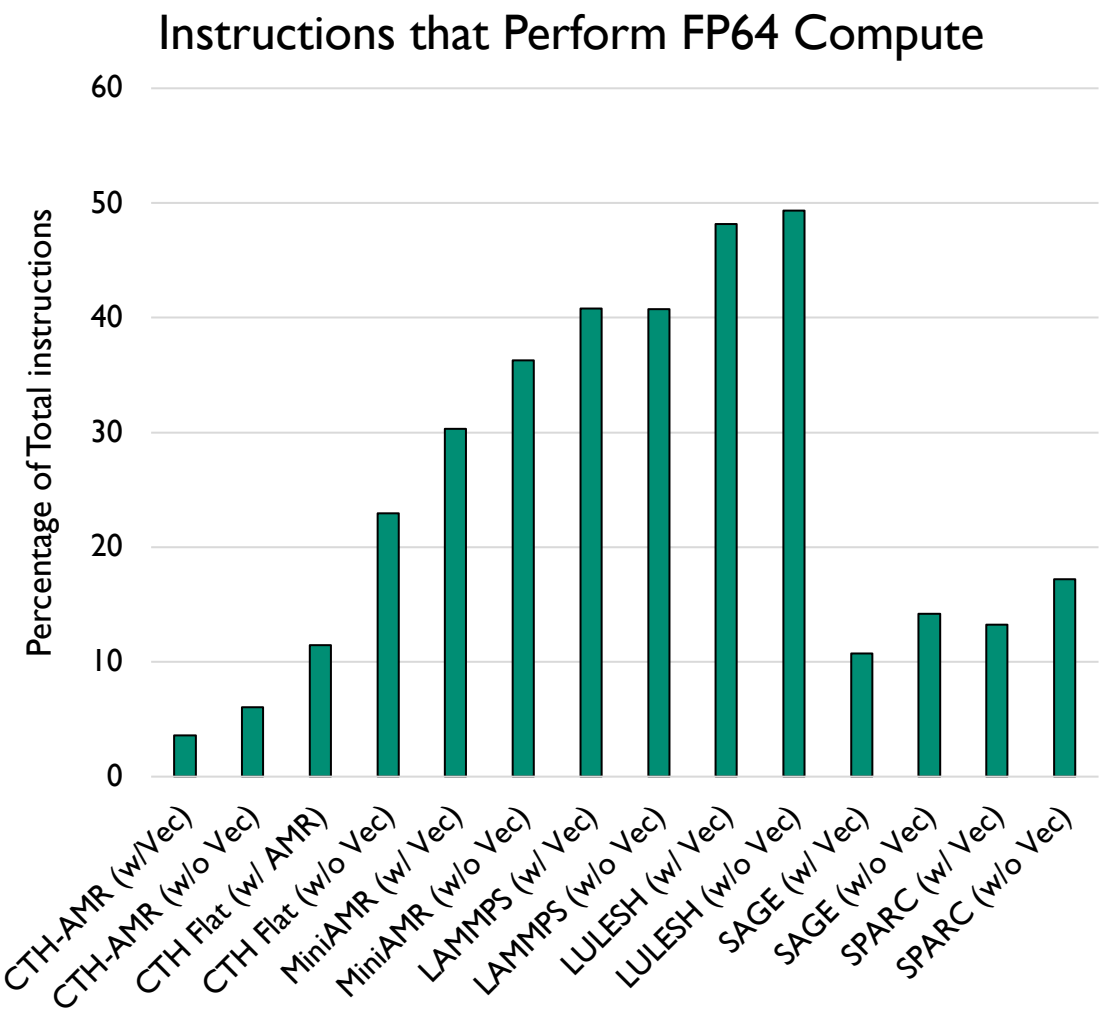
- Designed to run for O(minutes) of execution time to see real behavior
- Sufficiently large that correct memory subsystems and behaviors are observed

Compiled for KNL using Cray Programming Environment/compiler wrappers with the Intel PE loaded

Compiled each code with and with-out vectorization ("-no-vec")

- Still see vectorization in some codes because of calls to libraries on the platform that execute vector instructions (e.g. MKL, optimized calls to memset)
- See whether the compiled code with and with-out vectorization makes significant difference

Instruction Analysis

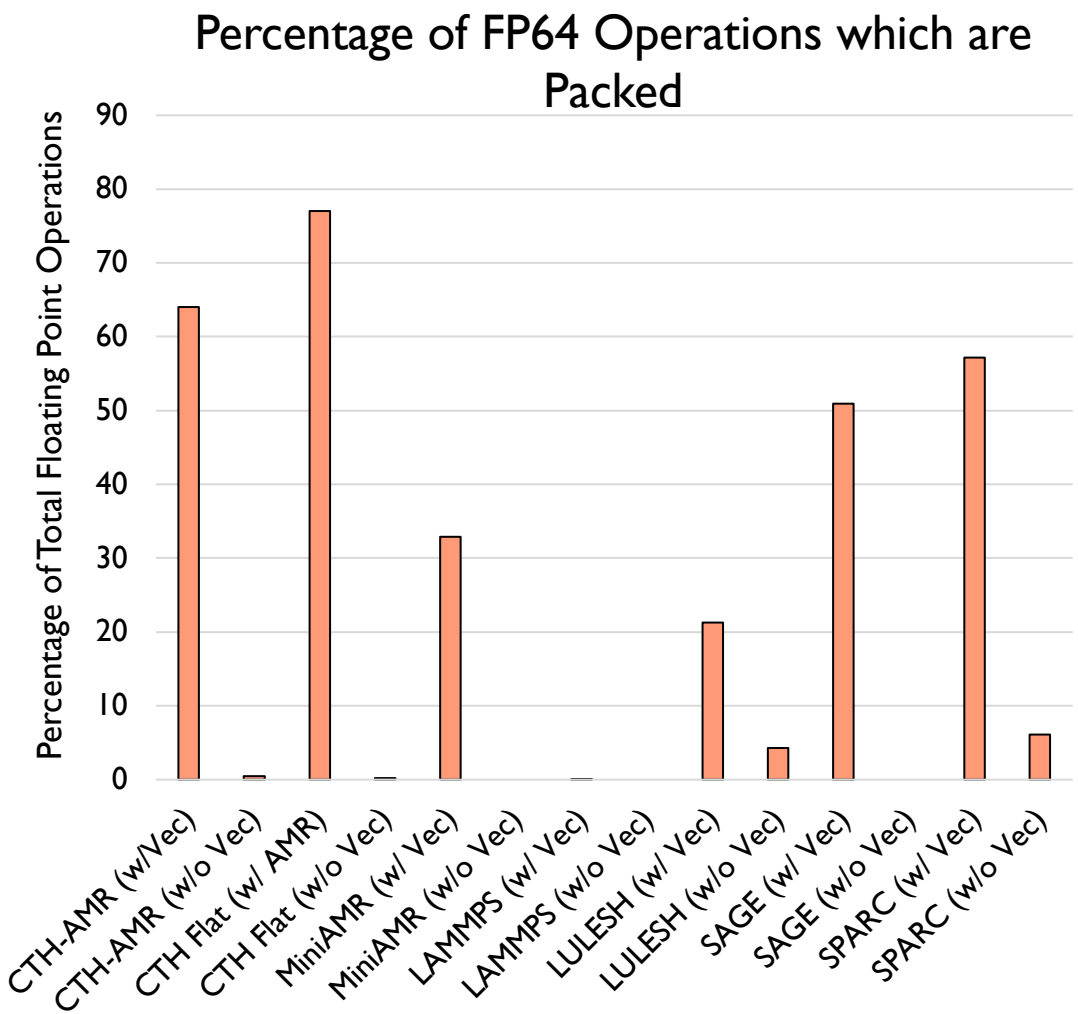


Number of instructions which utilize floating point arithmetic (percentage of entire program execution)

Mini-applications and LAMMPS have significantly denser compute-instruction intensity

- ”Simpler” access patterns
- LAMMPS has more optimized execution and algorithms
- CTH-AMR has poor intensity because of management of the adaptive refinement and complex stencils

Vectorization Analysis



Analyze the percentage of floating point operations which execute in "packed" mode

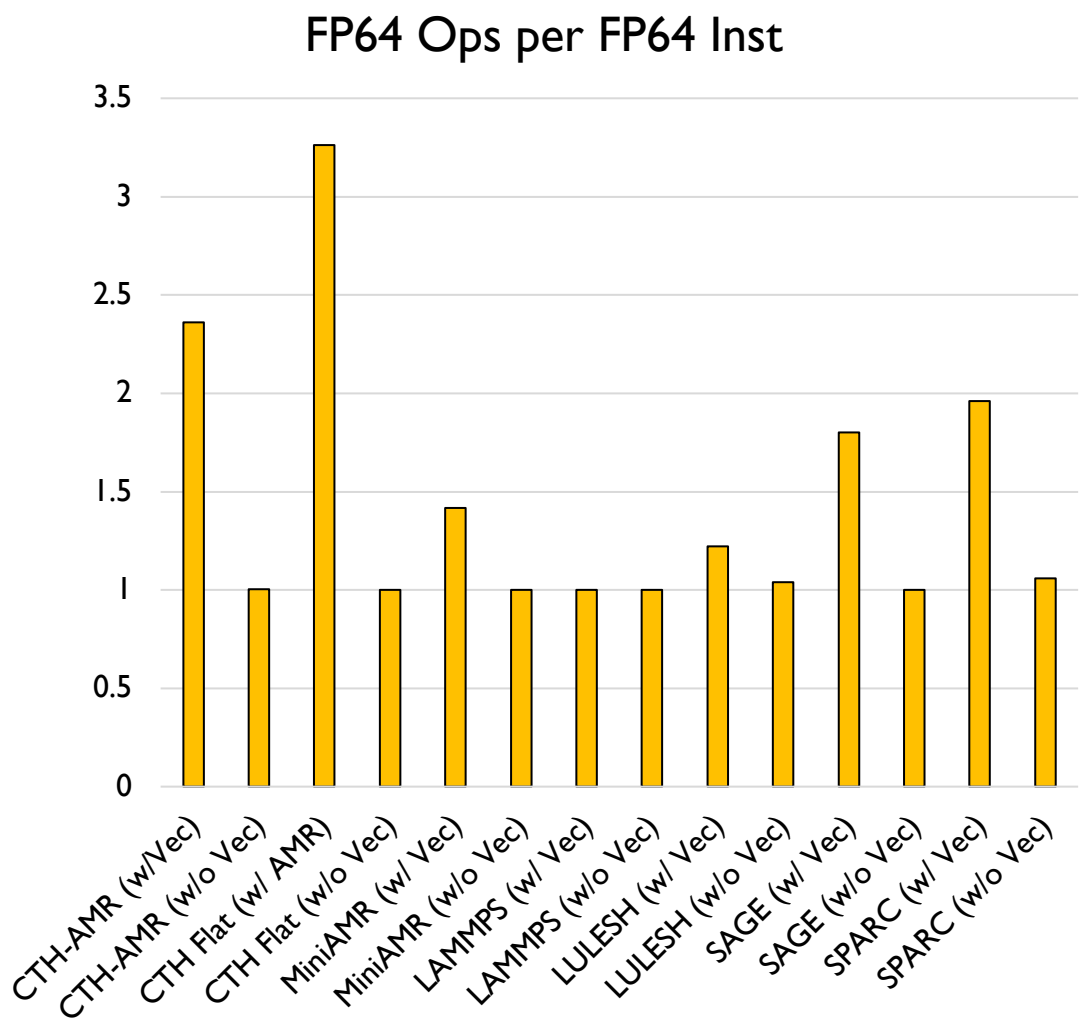
- Packed uses all lanes of the vector unit and does not perform any masking
- Represents the most efficient use of the vector unit

CTH and SAGE are typically higher because Fortran is anecdotally more likely to vectorize cleanly

SPARC is unusual for a C++ code because it has high levels of efficiency

- Because of use of manual intrinsics in Kokkos-Kernels/Trilinos libraries

Vectorization Analysis (Operation Density)



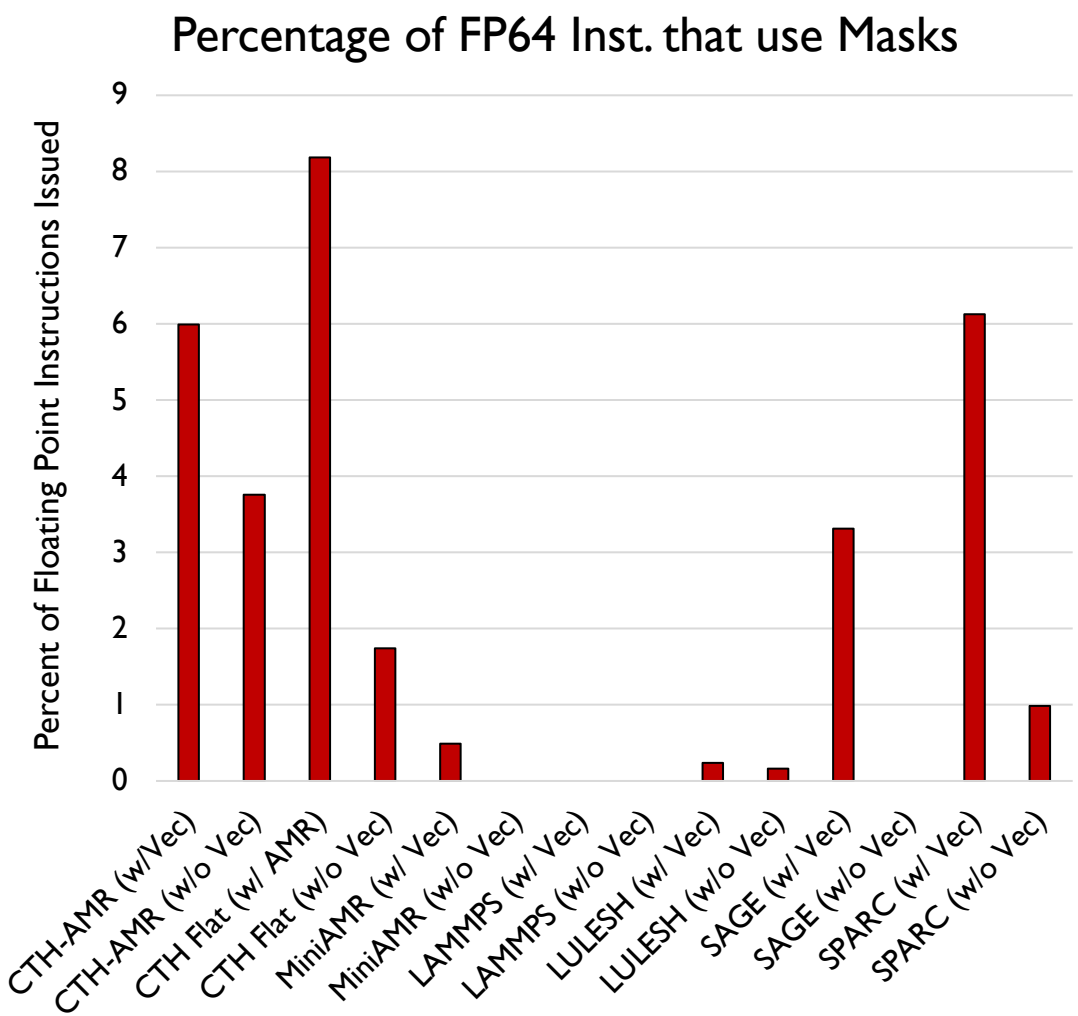
Number of floating point operations per instruction that performs a 64-bit double precision operation

- Ideal value here is 16 (8-wide, 2-per lane (e.g. FMA)
- Value of 1 indicates basic scalar-like behavior

Note – big difference here in vectorization level achieved in flat/AMR mesh modes for CTH

- AMR adds significant complexity to look up for data operations

Vectorization Analysis (Mask Operations)



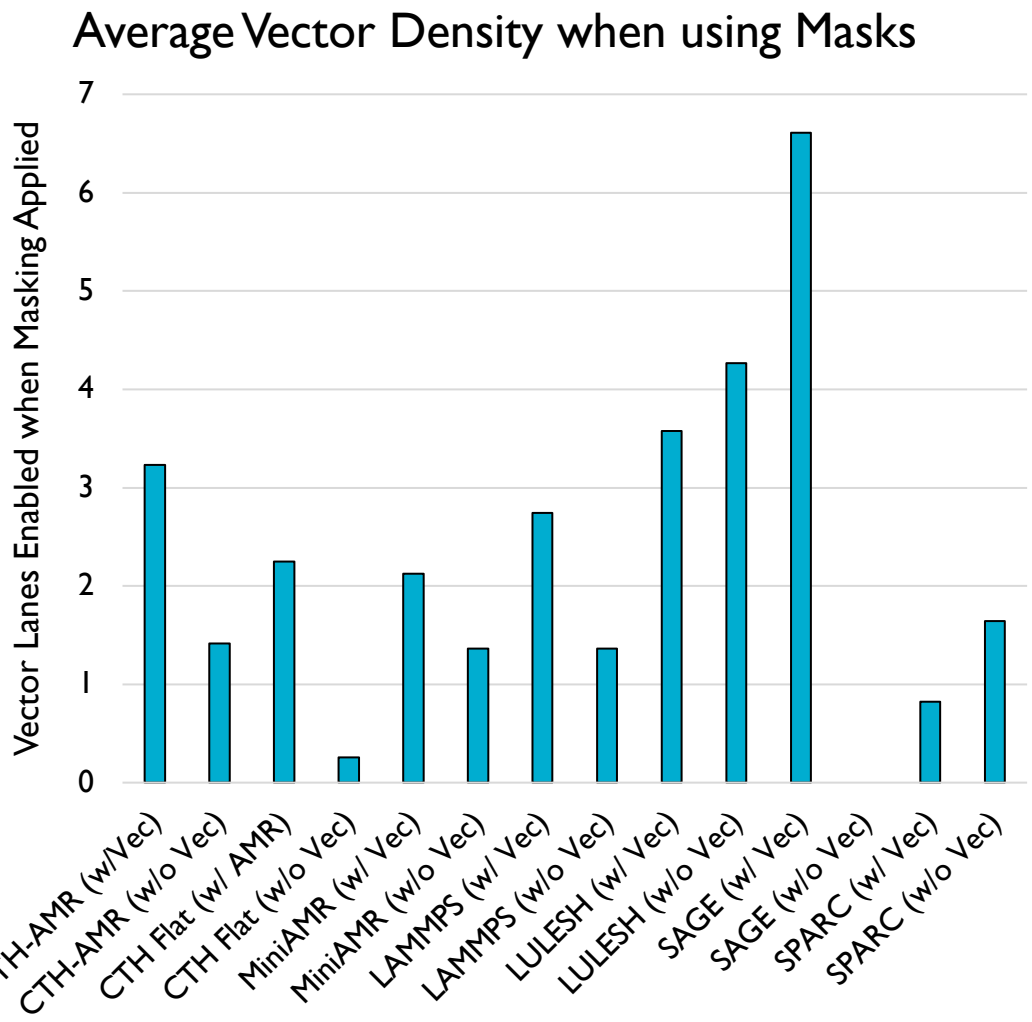
Masking lowers wide-vector efficiency by disabling lanes using predicates

- Lowers computational efficiency
- But .. can make very complex code easier to compile (otherwise just fall back to scalar)
- Dense vector use is ideal

Typically not able to be counted correctly by hardware (hardware counts all vector operations including memory loads/stores and integer operations with masking).

- We count only double precision to look at compute efficiency

Vectorization Analysis (Mask Operations)



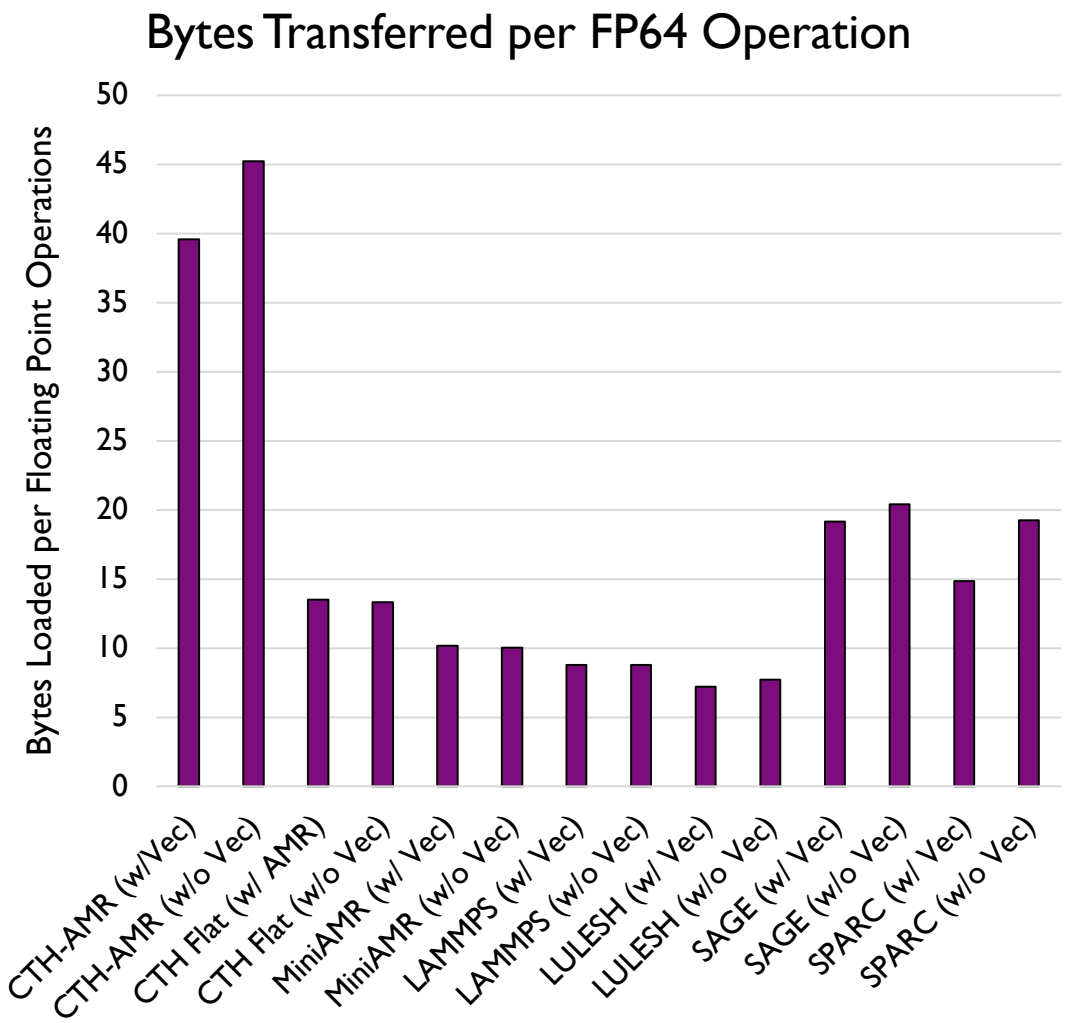
How many lanes are enabled during each instruction that utilizes masks?

- Ideal is for this to be close to SIMD width (8-wide)
- Only close for SAGE hydrocode

See poor masking efficiency for codes

- Represents divergence in the code paths (which generate masks)
- Implies we need to find ways to reduce divergence if we want to increase vector efficiency

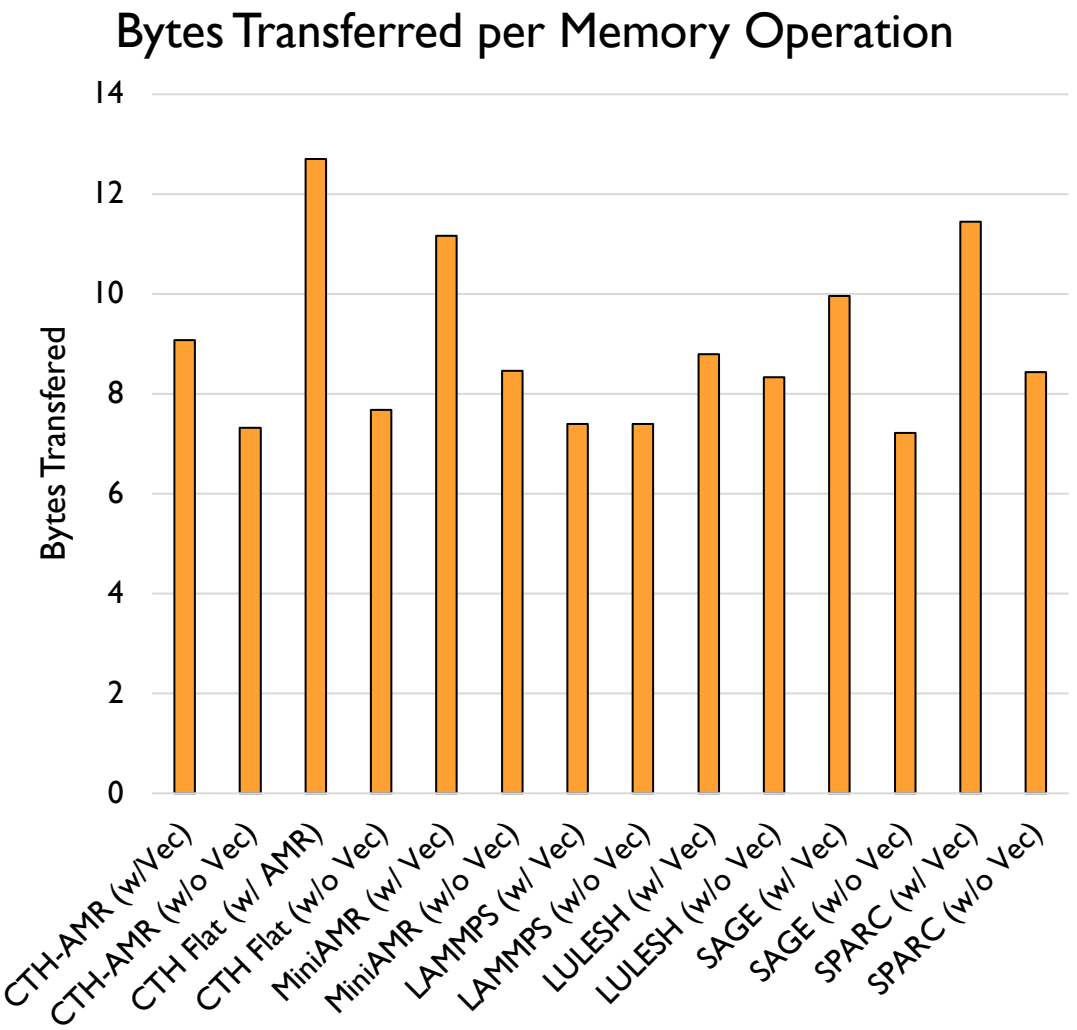
Bytes Transferred to Floating-Point Compute Intensity



How many bytes are loaded/stored per double precision operation?

- Common "bytes to FLOP" metric which is rough guide to system design
- High values for compute intensive codes are 24 bytes (two 8B operand loads, and an 8B operand store)
  - Values beyond this show codes are having to perform indirection/lookups, data movement, overheads etc
  - Show the weakness of memory subsystems that are delivering in the other of 0.25B per FLOP or worse

Bytes Transferred per Memory Operation



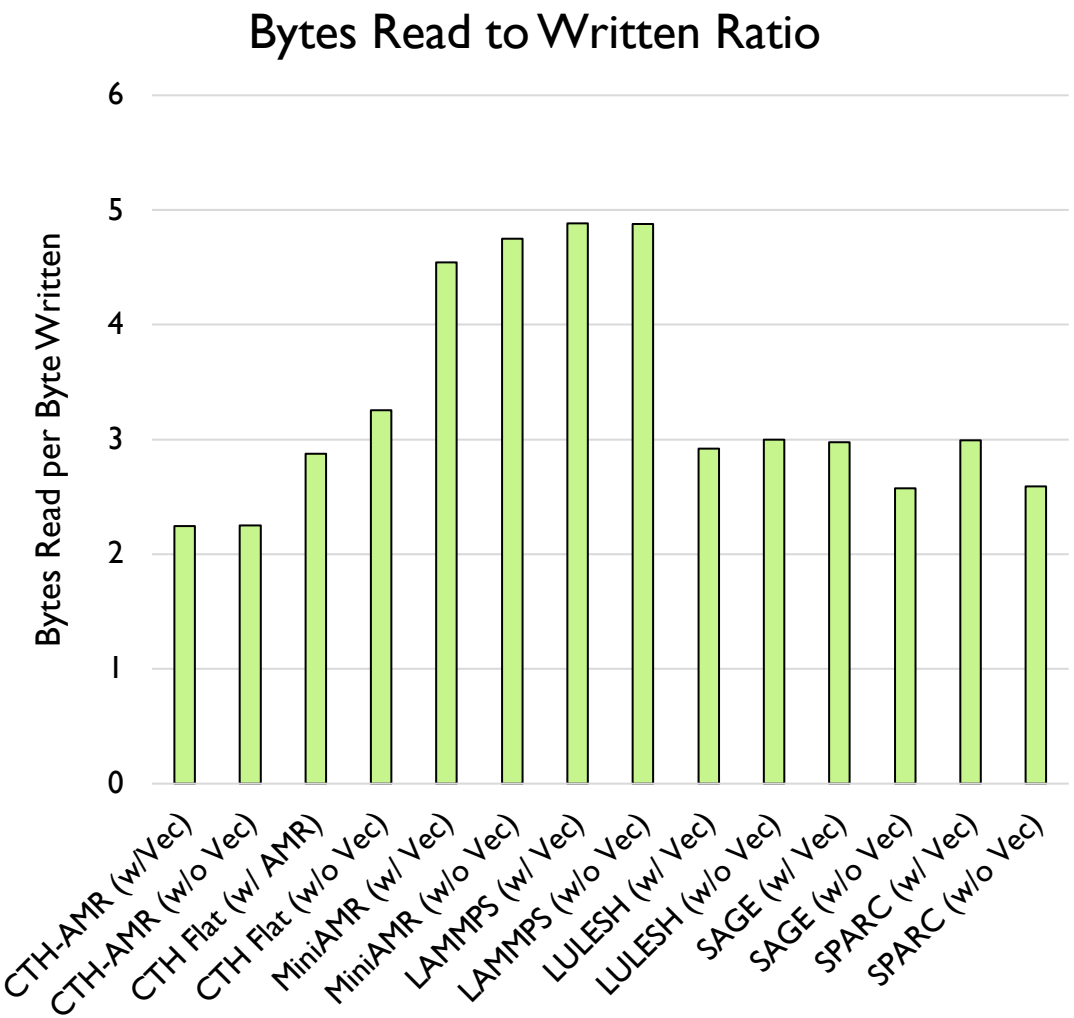
How many bytes are loaded/stored per memory operation?

- Hardware is optimized for wider data paths (typically 32 or 64B)
- KNL is optimized for 64B load/stores to support wide vector registers

Codes show much more modest load/store widths

- Consistent with lower vectorization level (loads scalar values)
- Indirect loads/stores where offset/indices must be loaded from memory

Bytes Read/Written Ratio



What is the ratio of bytes read to bytes written?

- Hardware is typically optimized for a 2:1 of read : write ratio (makes sense since need two operands for each result generated)
- Typical hardware ratio is close for most codes although slightly higher read throughput would improve performance for most codes

MiniAMR and LAMMPS have significant read dominance

- Read *many* operands to generate a single result

Conclusions and Discussion

Conclusions

Understanding vectorization levels is difficult (especially with limited hardware counter support)

- Yet essential, because we are trying to improve code performance on Haswell, Knights Landing, Skylake etc.
- Needed to produce our own tools to get the data we wanted from our codes

Want to use the data to guide application optimization efforts and future hardware design/optimization activities

- Growing use of vector/vector-like support in HPC-class processors
- Still useful for GPU porting as well (identifying/removing data dependencies)

Conclusions

Profiled the vectorization levels achieved in broad range of applications

- Not just mini-applications and benchmarks, and running complex input decks

Fortran codes produce higher levels of vectorization

- Language is designed to support analysis/generation of vectorized operations

SPARC C++ has higher levels of vectorization because of use of manually vectorized/intrinsics in the main application solver

- Using the Kokkos-Kernels/Trilinos C++ wrappers of SIMD types
- Automatic vectorization is still challenging

Conclusions

Evaluated several "rules of thumb" for hardware design

Read/Write Bytes Ratio of 2:1 seems to be reasonable, 3:1 would be better
- Need to support additional indirection indices as well as throughput for compute operations

Fairly limited use of masking hardware (most is 8%)
- But the use of these instructions makes compilation much easier
- Supported but perhaps don't need to be as aggressively optimized?

Typical bytes transferred per memory operation is in region of 7 to 16
- Hardware is optimized for 32 or 64B (64B for KNL)
- Need to see if better way to optimize around expected smaller width loads?

Conclusions

Bytes per FLOP ratio varies considerably from 8 to 45 with broad diversity.

- See considerable variation in hardware ratios today but implication is that we are still FLOP rich and bandwidth poor
- Exascale is not helping us get to where we want to be (when we focus on HPL/LINPACK)

Want to drive to more balanced and efficient systems for our applications, not just our benchmarks

Need good tools to work at application scale (or much better hardware counter support)


Sandia
National
Laboratories