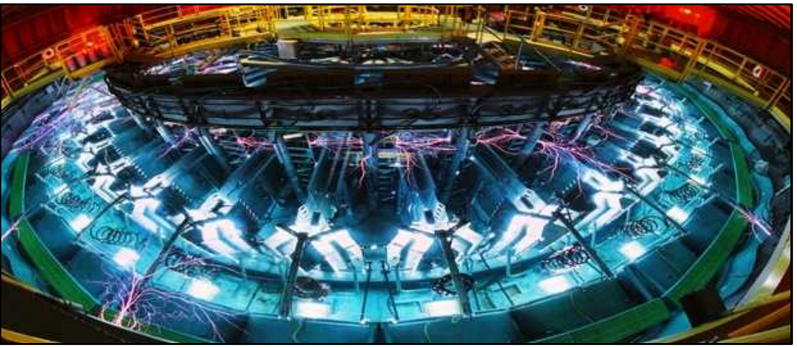
SAND2016-8879PE

Exceptional service in the national interest

Sandia National Laboratories



Getting Started with Vectorization

Si Hammond (sdhammo@sandia.gov)

Scalable Computer Architectures

Center for Computing Research

Sandia National Laboratories, NM

U.S. DEPARTMENT OF ENERGY   NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Overview

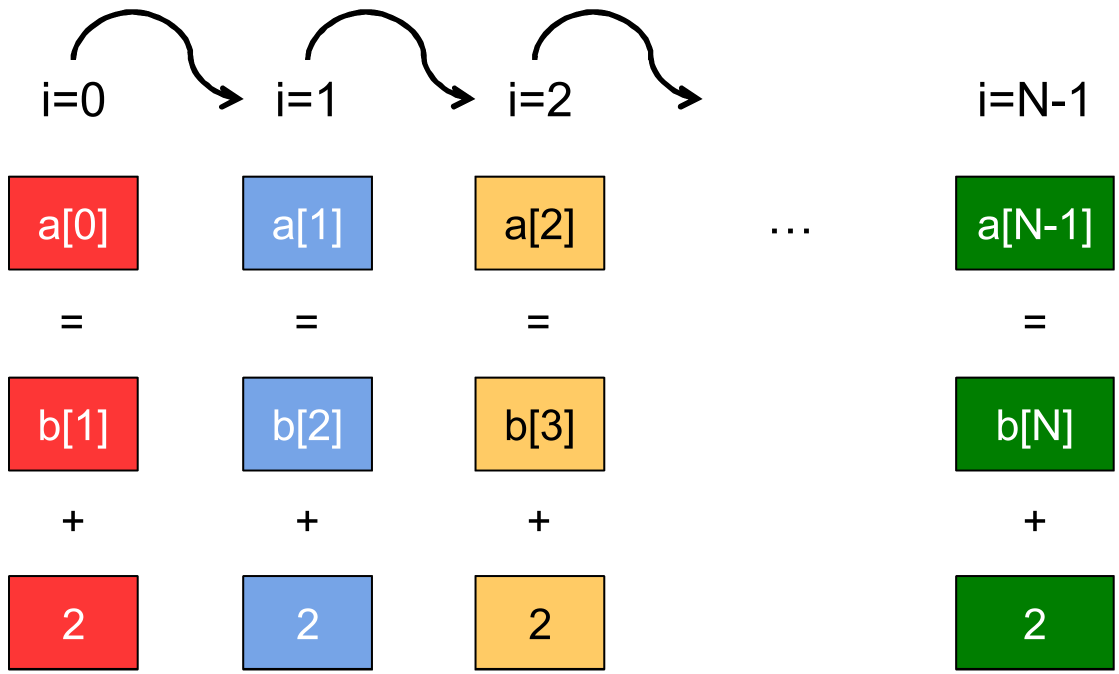
- **Basic Vectorization terminology and concepts**
  - This will probably be familiar ground for lots of you, but lets check we are saying the same thing
  - Intel compiler optimization reports
- **Intel Vector Advisor XE**
  - How to analyze and improve vectorization in your applications
- **Basic Tips and Techniques for Getting Better Vectorization**
- **Wrap Up and Some News!**

VECTORIZATION 101

Terminology

If we have this code each iteration of the loop we get one set of operations

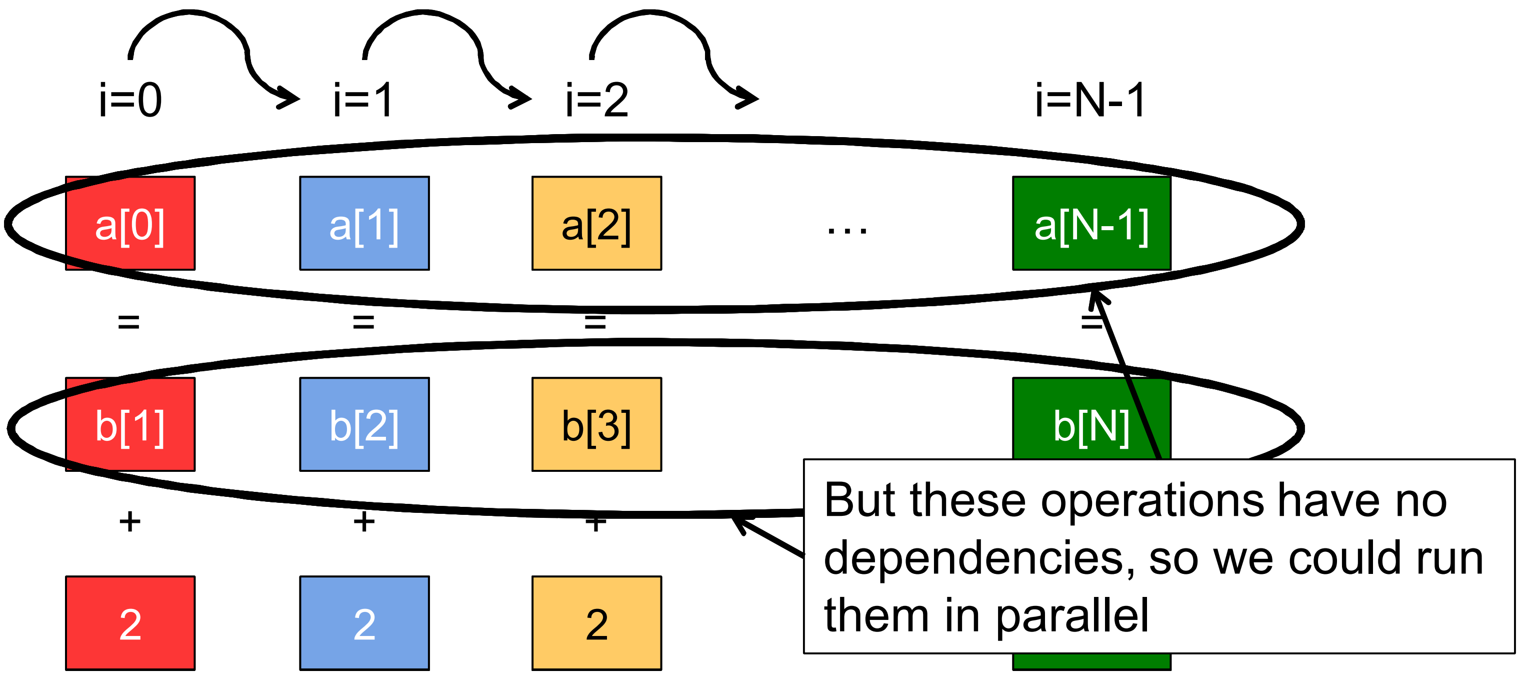
```
for(int i = 0; i < 129; i++) {
        a[i] = b[i+1] + 2;
}
```



Data Dependency

If we have this code each iteration of the loop we get one set of operations

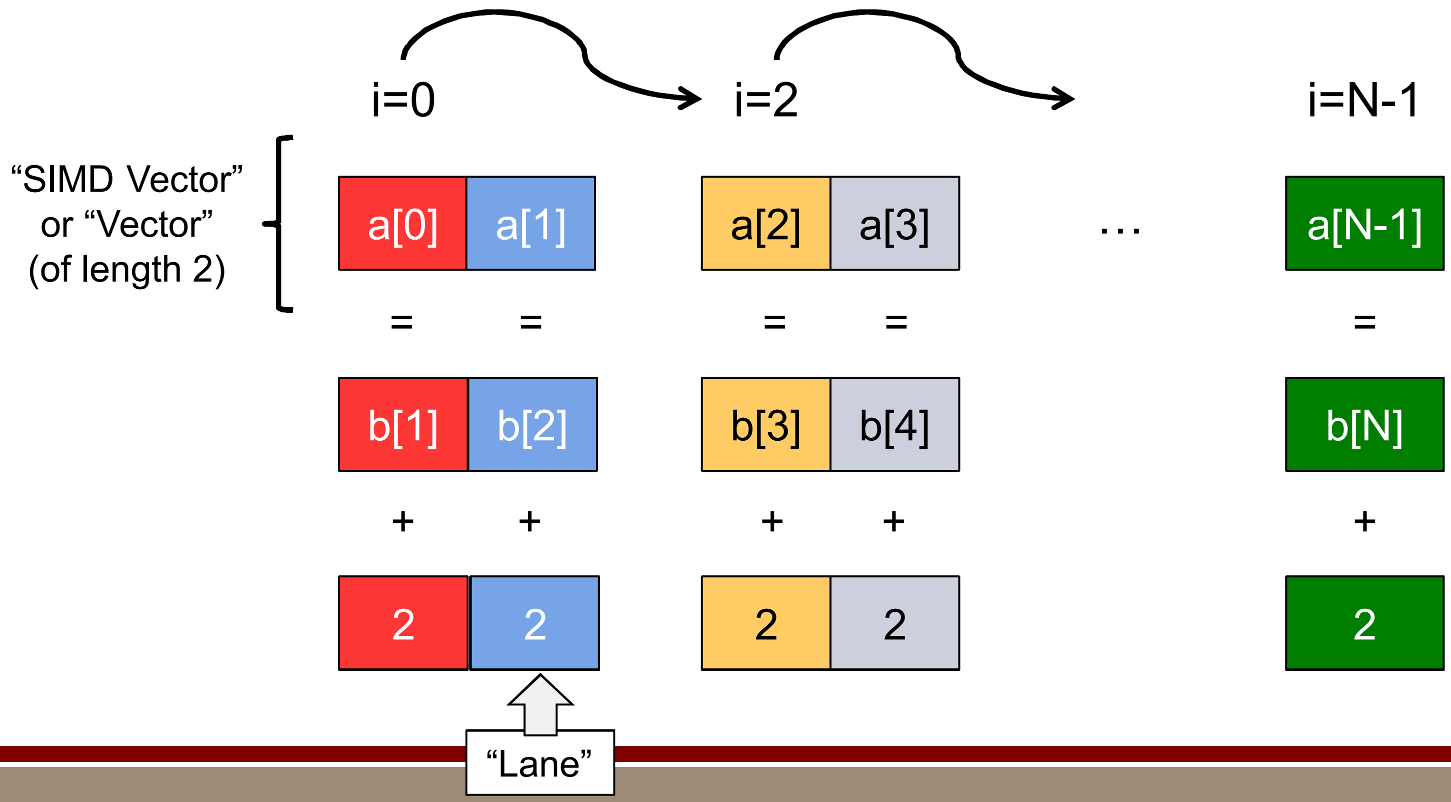
```
for(int i = 0; i < 129; i++) {
      a[i] = b[i+1] + 2;
}
```



Terminology Example

If we have this code each iteration of the loop we get one set of operations

```
for(int i = 0; i < 129; i++) {
	a[i] = b[i+1] + 2;
}
```



Transformed Code

```
for(int i = 0; i < 129; i+=2) {
        a[i] = b[i+1] + 2;
        a[i+1] = b[i+2] + 2;
}
```

- Transform our loop into logically doing two iterations per actual iteration
  - More efficient since we now spend less time in loop logic
  - Allows us to run many more operations in parallel = faster to execute

Correct Execution

```
for(int i = 0; i < 129; i+=2) {
        a[i] = b[i+1] + 2;
        a[i+1] = b[i+2] + 2;
}
```

- We need to ensure we execute **exactly** the same computation as before vectors
  - Have to be very careful exactly how the code is transformed
  - Compiler needs to do some more work

Remainder Loops

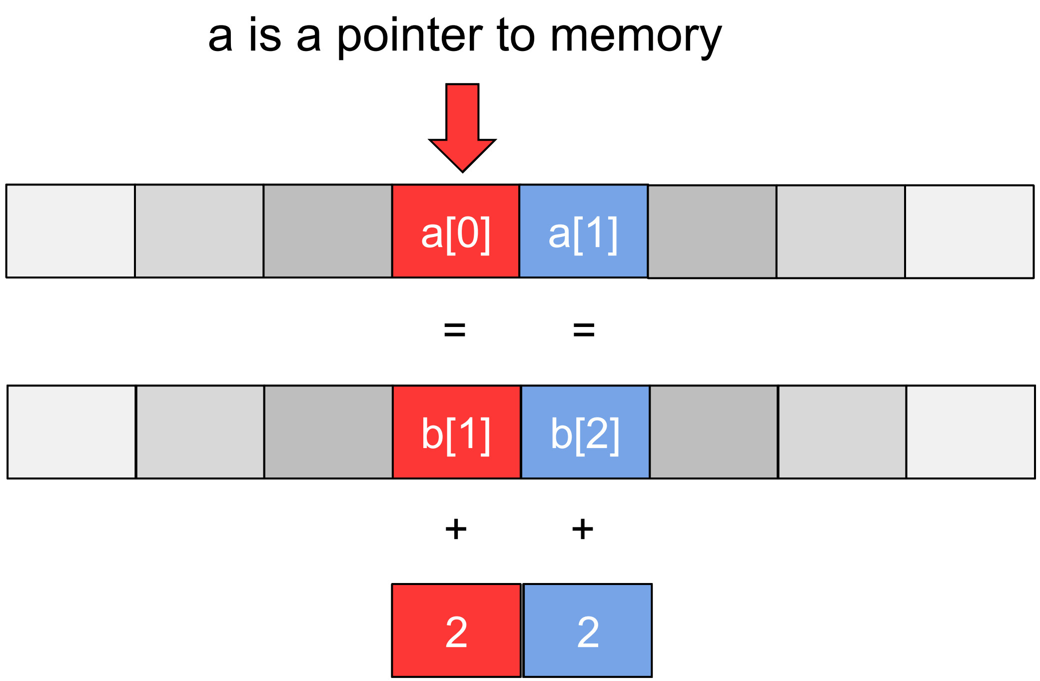
```
for(int i = 0; i < 128; i+=2) {
        a[i] = b[i+1] + 2;
        a[i+1] = b[i+2] + 2;
}

for(int i = 128; i < 129; i++) {
        a[i] = b[i+1] + 2;
}
```

Vector Body (first loop)

Epilogue
Or "**Remainder**" (second loop)

Sometimes we also introduce a **prologue** to ensure we get into vector alignment

Vector Alignment



Vector alignment usually means we want the pointer "a" to located at an address which is modulo the SIMD width == 0

Why? Because this usually means vectors don't span cache lines == more efficient loads and stores

General Rules

- Data dependency breaks the ability to vectorize
  - If dependencies are found or *might* exist the compiler will **not** generate vectorized code
- Data alignment costs performance
- Loops which have short loop trip counts will usually execute in the remainder loop (and so not benefit from the vectorization)
- Most compilers estimate the potential speedup from using vector instructions, if its too low, the compiler will not generate vector loops
  - Which means lost performance opportunities

Vectorization Reports

- Generally easy way if using the Intel compiler, add the following to your CFLAGS, CXXFLAGS or FFLAGS
  - -opt-report=5

- This will generate a lot of information about inlining, vectorization and OpenMP threading in a file put in the working directory

- Usually mapped onto files and line numbers (where the loop **starts**)

Good Example

Sandia National Laboratories

```
  LOOP BEGIN at lulesh.cc(300,3) inlined into lulesh.cc(2444,5)
     remark #15389: vectorization support: reference domain[i] has unalig
301,46) ]
     remark #15389: vectorization support: reference domain[i] has unaligned access   [ lulesh.cc(
301,60) ]
     remark #15389: vectorization support: reference sigzz[i] has unaligned access   [ lules
01,27) ]
     remark #15388: vectorization support: reference sigyy[i] has aligned access   [ lulesh.
,16) ]
     remark #15388: vectorization support: reference sigxx[i] has aligned access   [ lulesh.cc(301
,5) ]
     remark #15381: vectorization support: unaligned access used inside loop body
     remark #15305: vectorization support: vector length 2
     remark #15399: vectorization support: unroll factor set to 4
     remark #15309: vectorization support: normalized vectorization overhead 0.342
     remark #15300: LOOP WAS VECTORIZED
     remark #15442: entire loop may be executed in remainder
     remark #15449: unmasked aligned unit stride stores: 2
     remark #15450: unmasked unaligned unit stride loads: 2
     remark #15451: unmasked unaligned unit stride stores: 1
     remark #15475: --- begin vector loop cost summary ---
     remark #15476: scalar loop cost: 20
     remark #15477: vector loop cost: 9.500
     remark #15478: estimated potential speedup: 2.050
     remark #15488: --- end vector loop cost summary ---
  LOOP END

  LOOP BEGIN at lulesh.cc(300,3) inlined into lulesh.cc(2444,5)
  <Alternate Alignment Vectorized Loop>
  LOOP END
```

Source Code Location

Variable alignment info

Vector length and loop unrolling

Profitability Estimation

Loop variants

So vector width is 2, we get a speed up of 2.05X ☺

Bad Example (Dependence)

```
LOOP BEGIN at lulesh.cc(982,7)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW dependence between domain[*(this+i*4)] (983:16) a
nd domain (983:16)
   remark #15346: vector dependence: assumed ANTI dependence between domain (983:16) and domain[*(t
his+i*4)] (983:16)
   remark #25439: unrolled with remainder by 2
LOOP END

LOOP BEGIN at lulesh.cc(982,7)
<Remainder>
LOOP END
```

- Different types of dependence (long story, they are *all* bad if they generate this message)
- Means one iteration of the loop interferes with another

Bad Example (Performance)

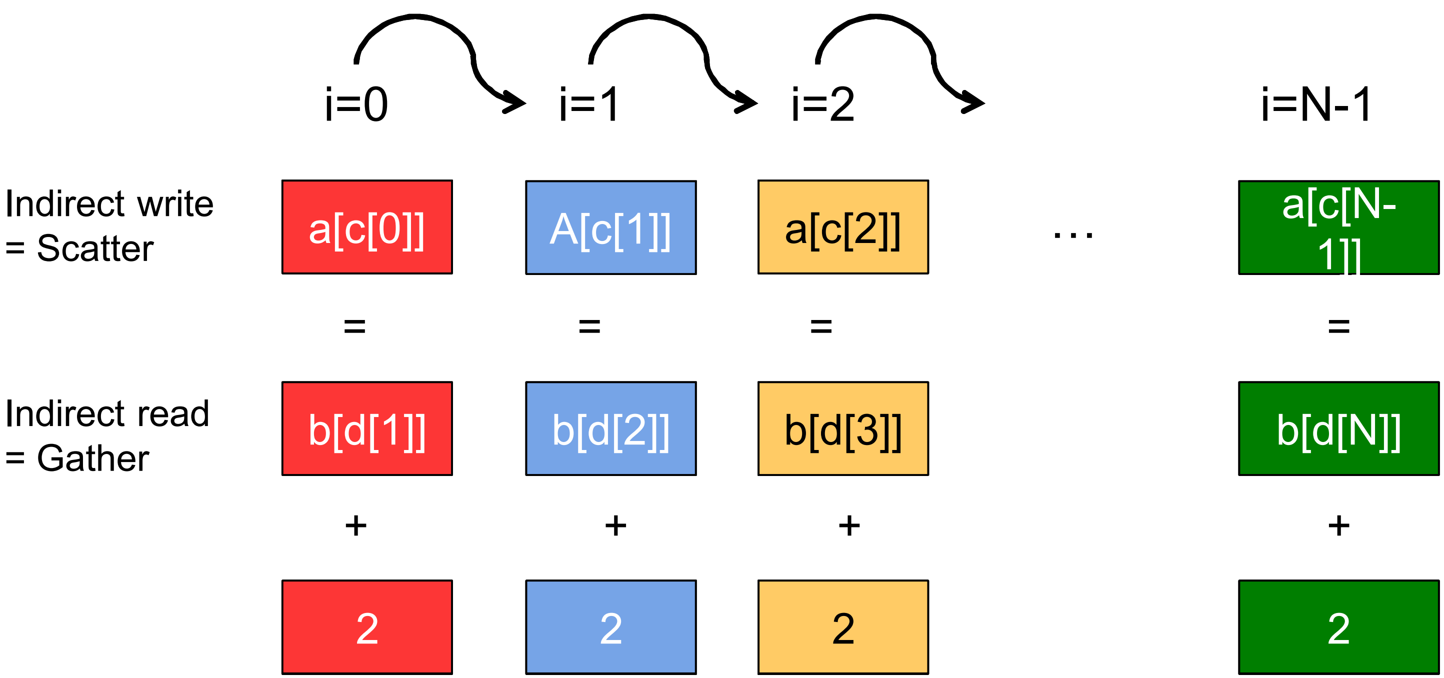
```
   LOOP BEGIN at lulesh.cc(300,3) inlined into lulesh.cc(2444,5)
   <Remainder loop for vectorization>
      remark #15389: vectorization support: reference domain[i] has unaligned access   [ lulesh.cc(
301,46) ]
      remark #15389: vectorization support: reference domain[i] has unaligned access   [ lulesh.cc(
301,60) ]
      remark #15389: vectorization support: reference sigzz[i] has unaligned access   [ lulesh.cc(3
01,27) ]
      remark #15389: vectorization support: reference sigyy[i] has unaligned access   [ lulesh.cc(3
01,16) ]
      remark #15388: vectorization support: reference sigxx[i] has aligned access   [ lulesh.cc(301
,5) ]
      remark #15381: vectorization support: unaligned access used inside loop body
      remark #15335: remainder loop was not vectorized: vectorization possible but seems inefficien
t. Use vector always directive or -vec-threshold0 to override
      remark #15305: vectorization support: vector length 2
      remark #15309: vectorization support: normalized vectorization overhead 0.806
   LOOP END
```

- Profitability analysis says that vectorization will be slower so it will not generate the slower sequence here
  - Typically because it does not know loop bounds

Gather/Scatter

If we have this code each iteration of the loop we get one set of operations

```
for(int i = 0; i < 129; i++) {
    a[c[i]] = b[d[i+1]] + 2;
}
```

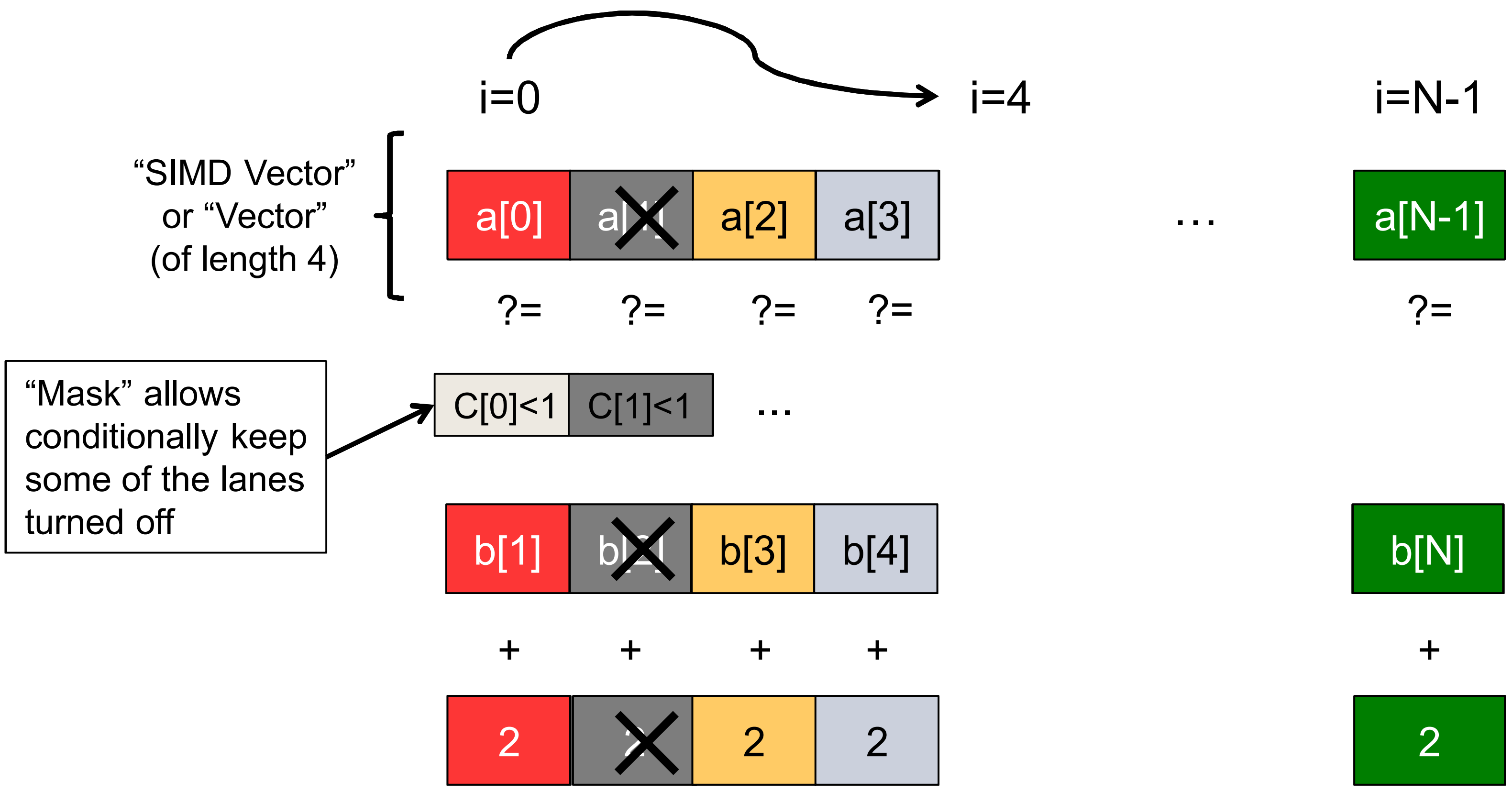


Gather/Scatter

```
      remark #15389: vectorization support: reference compHalfStep[i] has unaligned access   [ lule
sh.cc(2030,9) ]
      remark #15389: vectorization support: reference delvc[i] has unaligned access   [ lulesh.cc(2
029,32) ]
      remark #15381: vectorization support: unaligned access used inside loop body
      remark #15415: vectorization support: gather was generated for the variable <domain[elem]>, i
ndirect access, elem is read from memory   [ lulesh.cc(2017,27) ]
      remark #15415: vectorization support: gather was generated for the variable <domain[elem]>, i
ndirect access, elem is read from memory   [ lulesh.cc(2018,27) ]
      remark #15415: vectorization support: gather was generated for the variable <domain[elem]>, i
ndirect access, elem is read from memory   [ lulesh.cc(2019,27) ]
      remark #15415: vectorization support: gather was generated for the variable <domain[elem]>, i
ndirect access, elem is read from memory   [ lulesh.cc(2020,27) ]
      remark #15415: vectorization support: gather was generated for the variable <domain[elem]>, i
ndirect access, elem is read from memory   [ lulesh.cc(2021,28) ]
      remark #15415: vectorization support: gather was generated for the variable <domain[elem]>, i
ndirect access, elem is read from memory   [ lulesh.cc(2022,28) ]
      remark #15415: vectorization support: gather was generated for the variable <vnewc[elem]>, in
direct access, elem is read from memory   [ lulesh.cc(2028,39) ]
      remark #15415: vectorization support: gather was generated for the variable <vnewc[elem]>, in
direct access, elem is read from memory   [ lulesh.cc(2029,18) ]
      remark #15305: vectorization support: vector length 8
      remark #15309: vectorization support: normalized vectorization overhead 0.148
      remark #15301: FUSED LOOP WAS VECTORIZED
      remark #15442: entire loop may be executed in remainder
      remark #15450: unmasked unaligned unit stride loads: 2
      remark #15451: unmasked unaligned unit stride stores: 8
      remark #15458: masked indexed (or gather) loads: 8
      remark #15475: --- begin vector loop cost summary ---
      remark #15476: scalar loop cost: 109
      remark #15477: vector loop cost: 28.620
      remark #15478: estimated potential speedup: 3.550
      remark #15486: divides: 2
      remark #15488: --- end vector loop cost summary ---
      remark #25456: Number of Array Refs Scalar Replaced In L
   LOOP END
```

- In general gathers and scatters are slower than packed (direct) reads/writes) and the compiler explicitly tells you when it generates these

Vector Masks



Masking helps to vectorize more code *but* can mean we are less efficient if many lanes get disabled all the time – delicate balance

We don't know this until runtime

VECTOR ADVISOR XE

Vector Advisor XE

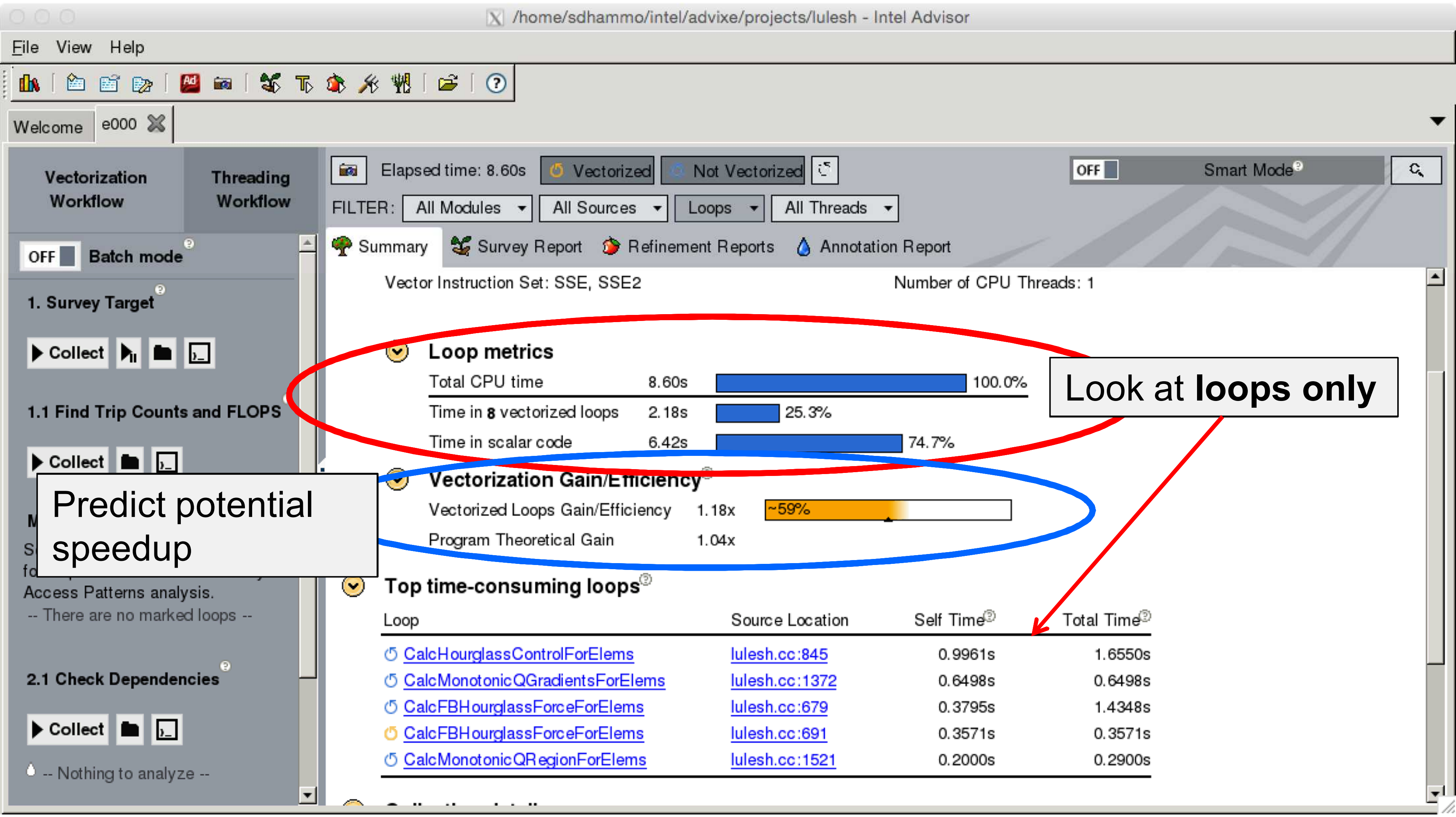
- Designed to be a survey, analysis and recommendation tool
  - **Vectorization Workflow** – add vectorization, make existing vectorization more efficient, plan for future KNL processors
  - **Threading Workflow** – add threading and parallelism into your application, check for safety conditions etc

- Latest version 2017 is in beta

- Installed on some of the ASC test bed machines, will be adding to ATDM test beds when product

- Should have a site license for installs on other machines

Vector Advisor XE GUI



General Vectorization Tips

General Tips

- **Use the const and __restrict__ keywords on arrays/variables**
  - Compiler can determine these are read-only/don’t have dependency on other variables
  - This makes a **HUGE** different not just for vectorization but general optimization

- **Don’t mix integer types in loop control**
  - For instance, unsigned int compared to an int or a long long int compare to a 32-bit int
  - Causes lots of additional data type conversion instructions and this breaks vectorization profitability analysis
  - Better to use standard int where you can (most compilers are built around this assumption)

General Tips

```
for(int i = 0; i < N; i++) {
        p[0] = 64 + i;
        p++;
}
```

- **Try to access arrays with p[i] notation, do not use pointer arithmetic**
  - Do **not** use pointer arithmetic
  - Compiler often cannot determine dependency pattern
  - Will often not vectorize or optimize this loop at all
- Kokkos Views stick with a(i, j, k) style, this is all handled correctly underneath abstraction layers

General Tips

```
for(int i = 0; i < N; i++) {
	p[i] = a[i] < 32 ? 1 : -1;
}
```

- **Use tertiary for comparison evaluations where possible**
  - This can lead to **very** efficient code generation on most modern platforms
  - Very efficient on KNL and Sky Lake Xeon processors which have strong masking capabilities
  - if-statement equivalent sometimes do not vectorize well

Advanced Tips

```
double* a __attribute__((aligned(64)));
..
posix_memalign(&a, 64, sizeof(double) * N));
..

for(int i = 0; i < N; i++) {
	a[i] = i * 111;
}
```

- Force alignment for allocations where using malloc/free
  - This allows compiler to bypass some of the prologue loops
  - Faster to get to main vector loops
  - Smaller binaries/code sequences = better instruction cache utilization
- Kokkos does this already (in general)

Compiler Flags

- Make sure you use the right compiler flags (these are for Intel):
  - -mavx for Sandy Bridge (Chama)
  - -xCORE-AVX2 for Haswell (Trinity) and Broadwell (CTS-1)
  - -xMIC-AVX512 for KNL (Trinity/Bowman/Ellis)
  - -mmic for Knights Corner **only** (Compton/Morgan)
- Otherwise you get Pentium-4 era instructions (yes 2004!)
- Compiler will generate vector instructions as much as it can
  - To disable –no-vec on Intel compiler

Wrap Up

Comments

- **Vectorization can have a huge impact on performance**
  - Not just compute performance, it helps cache efficient, memory subsystem *etc*
  - Very important moving in the future

- Lots of small changes (which aren't always measurable) eventually all add up as code gets more efficient
  - If you improve vectorization across your code base, in general it all adds up to better optimization opportunities over time

- Vectorization is an important part of running on CPU/many-core systems including Intel Xeon Phi, IBM POWER, AMD and other vendors
  - Also helps get efficient kernels ready for Kokkos where we can add threading too

And One More Thing…

SRN KNL Test Bed Cluster

- **ellis.sandia.gov joins the ASC test bed family**
  - 32 x KNL B0 Bin-1 silicon (68 cores, 1.4GHz, 16GB HBM, 96GB DDR4)
  - Intel OmniPath v1.0 network interconnect

- Familiar Linux environment, Intel 17.0 compilers, OpenMPI etc
- Will be a replica of bowman.sandia.gov (SON)

- **Expected in a few weeks for installation and then WebCARS access**

- SLURM queues, expecting heavy NGP and ATDM testing

Sandia National Laboratories

Exceptional service in the national interest