Intel® Advisor
Vectorization Optimization and Thread Prototyping
Software Must Vectorize & Thread or Performance Dies

True today – More true tomorrow – Difference can be substantial!

Vectorization - Have you:
- Recompiled for AVX2 or AVX512 with little gain
- Wondered where to vectorize?
- Recoded intrinsics for new arch.?
- Struggled with compiler reports?

Threading - Have you:
- Threaded an app, but seen little benefit?
- Hit a “scalability barrier”?
- Delayed release due to sync. errors?

“Intel® Advisor’s Vectorization Advisor fills a gap in code performance analysis. It can guide the informed user to better exploit the vector capabilities of modern processors and coprocessors.”

Dr. Luigi Iapichino
Scientific Computing Expert
Leibniz Supercomputing Centre

“Intel® Advisor has allowed us to quickly prototype ideas for parallelism, saving developer time and effort”

Simon Hammond
Senior Technical Staff
Sandia National Laboratories
Design Parallelism

• No disruption to regular development
• All test cases continue to work
• Tune and debug the design before you implement it
Intel Advisor Annotation Concepts

- Advisor uses 3 primary concepts to create a model
  - SITE
    - A region of code in your application you want to transform into parallel code
  - TASK
    - The region of code in a SITE you want to execute in parallel with the rest of the code in the SITE
  - LOCK
    - Mark regions of code in a TASK which must be serialized

NOTE
- All of these regions may be nested
- You may create more than one SITE
- Just macros, so work with any C/C++ compiler

```c
#include "advisor-annotate.h"
...
void solve() {
  int * queens = new int[size];
  //array representing queens placed on a chess board...
  ANNOTATE_SITE_BEGIN(solve);
  for(int i=0; i<size; i++)
    { // try all positions in first row
      ANNOTATE_ITERATION_TASK(setQueen);
      setQueen(queens, 0, i);
    }
  ANNOTATE_SITE_END();
...}
```
Suitability report

Estimated Overall Speed-up

Modeling of Runtime and loops

Scalability Graph
Adjustable: Target architecture, threading models and number of CPU

- To set up data collection determine the target architecture, threading model and number of CPU’s.
- Collect the Scalability data, and determine how it differs between the architectures and threading models.
Intel® Advisor - Vectorization

Recommended methodology

1. Characterize your code (e.g., scalar vs. vector, efficiency). Focus on most impactful parts.
   - Scalar Loops
   - SIMD Loops

2. Explore root cause preventing (compilers) from Vectorization. Implement low-hanging fix.
   - Localize memory/memory-access-bound cases.
   - Done with all low-hanging impactful parts of your code?
     - yes
       - Scalar Loops
     - no

3. Check if Dependencies are real. Resolve dependencies.
   - SIMD Loops

   - yes
     - Memory-bound loops
   - no

The Right Data At Your Fingertips
Get all the data you need for high impact vectorization

- Filter by which loops are vectorized!
- What prevents vectorization?
- Focus on hot loops
- What vectorization issues do I have?
- Which Vector instructions are being used?
- How efficient is the code?

Get Fast Code Fast!
5 Steps to Efficient Vectorization - Vector Advisor
(part of Intel® Advisor, Parallel Studio, Cluster Studio 2016)

1. Compiler diagnostics + Performance Data + SIMD efficiency information

2. Guidance: detect problem and recommend how to fix it

3. “Precise” Trip Counts + FLOPs & MASKS: understand utilization, parallelism granularity & overheads

4. Loop-Carried Dependency Analysis

5. Memory Access Patterns Analysis
Summary View: Plan Your Next Steps

What can I expect to gain?

Where do I start?
Vectorization Advisor runs on and optimizes for Intel® Xeon Phi™ architecture

AVX-512 ERI – specific to Intel® Xeon Phi

- Efficiency (72%), Speed-up (11.5x), Vector Length (16)

Performance optimization problem and advice how to fix it

Program metrics
- Elapsed Time: 142.79s
- Vector Instruction Set: AVX, AVX2, AVX512, SSE, SSE2
- Number of CPU Threads: 4

Loop metrics
- Total CPU time: 454.08s
- Time in 88 vectorized loops: 41.86s
Check if It Is Safe to Vectorize
Loop-Carried Dependencies Analysis Verifies Correctness

Select loop for Dependency Analysis and press play!

Vector Dependence prevents Vectorization!
Run Memory Access Patterns analysis, just to check how memory is used in the loop and the called function.
Get specific advice for Improving Vectorization.

Advisor XE shows hints how to decrease vectorization overhead.
Loop Analytics
Get detailed information about your loops
Highlight “impactful” AVX-512 instructions.

**Survey Static Analysis - AVX-512 “Traits”**

Survey Static Analysis - AVX-512 “Traits”

Summarized Traits in *Survey Report*.

Simplify “performance-aware” reading of *Source and Assembly*
Gather/Scatter analysis is very important for AVX-512

- AVX512 Gather/Scatter in wider use than on previous instruction sets
  - Many more applications can now be vectorized
  - Gives good average performance but far from optimal
  - Much greater need for Gather/Scatter profiling
  - With Intel® Advisor you get both dynamic and static gather/scatter information
Gather Instructions

- \( c[i] = a[b[i]] \) // indirect reference

if \( p[i] == q[i] \) \( c[i] = a[b[i]] \) // also masked

VPCMPEQQ ymm3, ymm2, ymm1
VPGATHERQQ ymm1, ptr [rax+ymm0], ymm3

Fundamental building block for sparse or indirect memory accesses, easing vectorization
Irregular access patterns decreases performance!

Gather profiling

- Run Memory Access Pattern Analysis (MAP)

2.2 Check Memory Access Patterns

- Collect

-- Nothing to analyze --

- 0%: percentage of memory instructions with unit stride or stride 0 accesses
  - Unit stride (stride 1) = Instruction accesses memory that consistently changes by one element from iteration to iteration
  - Uniform stride (stride 0) = Instruction accesses the same memory from iteration to iteration

- 50%: percentage of memory instructions with fixed or constant non-unit stride accesses
  - Constant stride (stride N) = Instruction accesses memory that consistently changes by N elements from iteration to iteration
  - Example: for the double floating point type, stride 4 means the memory address accessed by this instruction increased by 32 bytes, (4*sizeof(double)) with each iteration

- 50%: percentage of memory instructions with irregular (variable or random) stride accesses
  - Irregular stride = Instruction accesses memory addresses that change by an unpredictable number of elements from iteration to iteration
  - Typically observed for indirect indexed array accesses, for example, a[index[i]]
  - gather (irregular) accesses, detected for v(p)gather* instructions on AVX2 Instruction Set Architecture
### Am I bound by VPU/CPU or by Memory?: Advisor Memory Access Pattern and Footprint

<table>
<thead>
<tr>
<th>Footprint</th>
<th>Small enough</th>
<th>Big enough</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Access Pattern</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Stride</td>
<td>Effective SIMD No Latency and BW bottlenecks</td>
<td>Effective SIMD Bandwidth bottleneck</td>
</tr>
<tr>
<td>Const stride</td>
<td>Medium SIMD Latency bottleneck possible</td>
<td>Medium SIMD Latency and Bandwidth bottleneck possible</td>
</tr>
<tr>
<td>Irregular Access, Gather/Scatter</td>
<td>Bad SIMD Latency bottleneck possible</td>
<td>Bad SIMD Latency bottleneck</td>
</tr>
</tbody>
</table>
AVX-512 FLOPS and Mask profiler

Low mask population -> low performance (in spite of “high SIMD efficiency”)

Efficiency, FLOPS and Arithmetic Intensity correlation (more memory bound -> lower SIMD performance)
Why is Mask Utilization important?

3 elements suppressed

SIMD Utilization = 5/8 (62.5%)

Not utilizing full vectors!!

Fully utilized!
Advisor Roofline: under the hood

Roofline application profile:

Axis Y: \( \text{FLOP/S} = \#\text{FLOP} \) (mask aware) / \#\text{Seconds} \\
Axis X: \( \text{AI} = \#\text{FLOP} / \#\text{Bytes} \)

Seconds
User-mode sampling
Root access not needed

Performance = Flops/seconds

Roofs
Microbenchmarks
Actual peak for the current configuration

#FLOP
Binary Instrumentation
Does not rely on CPU counters

Bytes
Binary Instrumentation
Counts operands size (not cachelines)
# Getting Roofline in Advisor

FLOP/S = #FLOP/Seconds

<table>
<thead>
<tr>
<th>Seconds</th>
<th>#FLOP Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Mask Utilization</td>
<td></td>
</tr>
<tr>
<td>- #Bytes</td>
<td></td>
</tr>
</tbody>
</table>

## Step 1: Survey
- Non intrusive. *Representative*
- Output: Seconds (+much more)

## Step 2: FLOPS
- Precise, instrumentation based
- Physically count Num-Instructions
- Output: #FLOP, #Bytes
Cache-Aware Roofline

Next Steps

If under or near a memory roof...
- Try a MAP analysis. Make any appropriate cache optimizations.
- If cache optimization is impossible, try reworking the algorithm to have a higher AI.

If Under the Vector Add Peak
Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to induce FMA usage.

If just above the Scalar Add Peak
Check vectorization efficiency in the Survey. Follow the recommendations to improve it if it’s low.

If under the Scalar Add Peak...
Check the Survey Report to see if the loop vectorized. If not, try to get it to vectorize if possible. This may involve running Dependencies to see if it’s safe to force it.
Typical Vectorization Optimization Workflow

- There is no need to recompile or relink the application, but the use of `-g` is recommended.

1. Collect survey and tripcounts data
   - Investigate application place within roofline model
   - Determine vectorization efficiency and opportunities for improvement

2. Collect memory access pattern data
   - Determine data structure optimization needs

3. Collect dependencies
   - Differentiate between real and assumed issues blocking vectorization
Nbody demonstration
Let’s consider a distribution of point masses $m_1, \ldots, m_n$ located at $r_1, \ldots, r_n$.

We want to calculate the position of the particles after a certain time interval using the Newton law of gravity.

```plaintext
struct Particle {
    public:
        Particle() { init(); }
        void init() {
            pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
            vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
            acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
            mass = 0.;
        }
        real_type pos[3];
        real_type vel[3];
        real_type acc[3];
        real_type mass;
};
```

```plaintext
for (i = 0; i < n; i++) {    // update acceleration
    for (j = 0; j < n; j++){
        real_type distance, dx, dy, dz;
        real_type distanceSqr = 0.0;
        real_type distanceInv = 0.0;

        distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
        distanceInv = 1.0 / sqrt(distanceSqr);

        particles[i].acc[0] += dx * G * particles[j].mass *
            distanceInv * distanceInv * distanceInv;
        particles[i].acc[1] += ...;
        particles[i].acc[2] += ...;
    }
}
```
Collect Roofline Data

- Starting with version 2 of the code we collect both survey and tripcounts data:

  ```bash
  mpirun -n 1 -N 1 advixe-cl --collect survey --project-dir ./adv_res --search-dir src=:./ \n  --search-dir bin=:./ -- ./nbody.x
  mpirun -n 1 -N 1 advixe-cl --collect tripcounts -flops-and-masks --project-dir ./adv_res \n  --search-dir src=:./ --search-dir bin=:./ -- ./nbody.x
  mpirun -n 1 -N 1 advixe-cl -collect roofline (available starting 2018 U1)
  ```

- If finalization is too slow on compute add `-no-auto-finalize` to collection line.
GUI left panel provides access to further tests

Summary provides overall performance characteristics

Lists instruction set(s) used

Top time consuming loops are listed individually

Loops are annotated as vectorized and non-vectorized

Vectorization efficiency is based on used ISA, in this case Intel® Advanced Vector Extensions 512 (AVX512)
Survey Report (Source)

- Inline information regarding loop characteristics
- ISA used
- Types processed
- Compiler transformations applied
- Vector length used
- ...
Survey Report (Code Analytics)

- Detailed loop information
  - Instruction mix
  - ISA used, including subgroups
  - Loop traits
    - FMA
    - Square root
    - Gathers / Blends point to memory issues and vector inefficiencies
Using single threaded roof

- Code vectorized, but performance on par with scalar add peak?
- Irregular memory access patterns force gather operations.
- Overhead of setting up vector operations reduces efficiency.

Next step is clear: perform a Memory Access Pattern analysis
Memory Access Pattern Analysis (Refinement)

Storage of particles is in an Array Of Structures (AOS) style

This leads to regular, but non-unit strides in memory access

- 33% unit
- 33% uniform, non-unit
- 33% non-uniform

Re-structuring the code into a Structure Of Arrays (SOA) may lead to unit stride access and more effective vectorization
Vectorization: gather/scatter operation

- The compiler might generate gather/scatter instructions for loops automatically vectorized where memory locations are not contiguous.

```cpp
struct Particle
{
    public:
        ...
        real_type pos[3];
        real_type vel[3];
        real_type acc[3];
        real_type mass;
};

struct ParticleSoA
{
    public:
        ...
        real_type *pos_x,*pos_y,*pos_z;
        real_type *vel_x,*vel_y,*vel_z;
        real_type *acc_x,*acc_y,*acc_z
        real_type *mass;
};
```
In this new version (version 3 in github sample) we introduce the following change:

- Change particle data structures from AOS to SOA
- Note changes in report:
  - Performance is lower
  - Main loop is no longer vectorized
  - Assumed vector dependence prevents automatic vectorization

Next step is clear: perform a Dependencies analysis
Dependencies Analysis (Refinement)

```
aprun -n 1 -N 1 advixe-cl --collect dependencies --project-dir ./adv_res \   --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

- Dependencies analysis has high overhead:
- Run on reduced workload
- Advisor Findings:
  - RAW dependency
  - Multiple reduction-type dependencies
Recommendations

Recommendation: Resolve dependency

The Dependencies analysis shows there is a real (proven) dependency in the loop. To fix: Do one of the following:

1. If there is an anti-dependency, enable vectorization using the directive `#pragma omp simd safelen(length)`, where `length` is smaller than the distance between dependent iterations in anti-dependency. For example:

```c
#pragma omp simd safelen(4)
for (i = 0; i < n - 4; i += 4)
{
    a[i + 4] = a[i] * c;
}
```

2. If there is a reduction pattern dependency in the loop, enable vectorization using the directive `#pragma omp simd reduction(operator:sum)`. For example:

```c
#pragma omp simd reduction(+:sumx)
for (k = 0; k < size2; k++)
{
    sumx += x[k] * b[k];
}
```
Performance After Resolved Dependencies

New memory access pattern plus vectorization produces much improved performance!
What next?

- Let’s explore threading with a suitability analysis.
- Recompile including annotation definitions
- Add headers to file
- Annotate suggested loops
- Run suitability collection
Annotating the code

- Add annotations as shown on the left sample
- Complex sites may be analyzed in more detail using task sections if needed
- `ANNOTATE_SITE_BEGIN` / `ANNOTATE_SITE_END`
- `ANNOTATE_TASK_BEGIN` / `ANNOTATE_TASK_END`
- Recompile including annotation definitions:
  - `-I/opt/intel/advisor/include`
- Collect suitability data

```c
#include "advisor-annotate.h"
...
ANNOTATE_SITE_BEGIN(steps)
for (int s=1; s<=get_nsteps(); ++s) {
    ...
    ANNTOATE_TASK_BEGIN(particles)
    for (i = 0; i < n; i++)
    { 
        ...
    }
    ANNTOATE_TASK_END(particles)
} 
ANNTOATE_SITE_END(steps)
```

```shell
mpirun -n 1 -N 1 advixe-cl --collect suitability --project-dir ./adv_res \
   --search-dir src=:./ --search-dir bin=:./ -- ./nbody.x
```
Suitability report

- Good speedup expected, but far from ideal (~56% efficiency).
- Modeling shows that increasing the task length would improve efficiency.
- Next step: add omp parallel region to code and re-test.
Now using regular roofline, instead of single-threaded

Still room for improvement, but at this point we need additional detail regarding shared resource utilization

```c
for (int s=1; s<=get_nsteps(); ++s)
{
    ts0 += time.start();

    #pragma omp parallel for
    for (i = 0; i < n; i++) // update acceleration
    {
        ...
        real_type ax_i = particles->acc_x[i];
        real_type ay_i = particles->acc_y[i];
        real_type az_i = particles->acc_z[i];

        #pragma omp simd reduction(+:ax_i,ay_i,az_i)
        for (j = 0; j < n; j++)
        {
            real_type dx, dy, dz;
            real_type distanceSqr = 0.0f;
            real_type distanceInv = 0.0f;

            dx = particles->pos_x[j] - particles->pos_x[i];
```