GUIDED CODE VECTORIZATION WITH INTEL® ADVISOR XE

Ryo Asai

Colfax International

April 12, 2016

Abstract

In this publication we discuss the usage of an optimization tool called Intel® Advisor. The discussion is illustrated with an example workload that computes the electric potential in a set of points in 3-D space produced by a group of charged particles. The example workload runs on a multi-core Intel Xeon processor with Intel AVX2 instructions.

The application was originally parallelized across cores, but otherwise neither optimized nor vectorized. In the publication, we discuss three performance issues that the Advisor detected: vector dependence, type conversion and inefficient memory access pattern. For each issue, we discuss how to interpret the data presented by the Advisor, and also how to to optimize the application to resolve these issues. After the optimization, we observed a 16x performance boost compared to the original, non-optimized implementation.

The example code used in this publication is available for download at the Colfax Research website.

Table of Contents

1 Introduction **2**

2 Example Workload **2**

2.1 Motivating Example: Electric Potential 2

2.2 System Specification and Compilation . 3

2.3 Some Considerations Before Starting . 4

2.4 Vector Dependence 4

2.5 Consistency of Precision 8

2.6 Inefficient Memory Access Pattern . . 10

3 Closing Words **16**

Appendix A Setting Up Advisor **18**

A.1 Installation, First-time Configuration . 18

A.2 Creating a Project 19

Appendix B Advisor Workflow **20**

B.1 Compilation/Recompilation 20

B.2 Survey Analysis 20

B.3 (Optional) Trip Count Analysis 21

B.4 Investigating Loops 21

B.5 (Optional) Deeper Analysis 22

B.6 Implementing Optimizations 23

1. INTRODUCTION

In recent years, modern processors used in the High Performance Computing (HPC) industry have become increasingly parallel: modern computer architectures rely on high core counts and wide vector registers for performance. At the same time, processor clock speeds no longer significantly increase from one generation of CPUs to another due to practical limitations on power efficiency and cooling. This shift in source of computing power, from high frequency to massive parallelism, means that applications running on modern computers must be adapted to take advantage of highly parallel architecture.

Adapting applications to take advantage of parallelism, a procedure often referred to as *code modernization* [3], is sometimes a difficult task, especially for developers lacking prior experience in parallel programming. At the same time, for legacy applications, modernization is synonymous with optimization because orders of magnitude of added performance may be unlocked [4]. Intel® Advisor XE is a software tool developed by Intel Corporation to assist developers with some of the aspects of code modernization.

Advisor is designed to help developers in the early stages of application optimization. It can provide detailed analysis of the code and make recommendations in multiple areas of performance optimization, including vectorization, thread parallelism, and memory access pattern. However, we will only discuss the Vectorization Advisor component of Intel Advisor XE in this paper [5].

To demonstrate the usage of Vectorization Advisor, we will use a non-optimized example application, electric potential computation, and optimize it step-by-step using the Advisor.

2. EXAMPLE WORKLOAD

In this section we will use the Intel Advisor to modernize an example application for better data parallelism. The application will start non-vectorized, with inefficient memory access patterns. One note is that although our application is non-vectorized at the start, it is parallelized across cores from the beginning. The Advisor is capable of assisting with adding and optimizing multi-threading, but we will start out multi-threaded so that our discussion can focus solely on vectorization.

2.1. MOTIVATING EXAMPLE: ELECTRIC POTENTIAL

We will use a physics application in electrostatics to serve as an example for usage model of the Advisor. In this application, we will compute the electric potential at various points in space produced by a group of charged point particles. At any given point $\{R_x, R_y, R_z\}$, the electric potential can be computed by adding individual contributions from each particle (principle of superposition) found by using Coulomb's law. Equation (1) shows the mathematical representation of this workload.

$$\Phi\left(\vec{R}_j\right) = -k\sum_{i=1}^{m}\frac{q_i}{\sqrt{\left(r_{i,x}-R_{j,x}\right)^2+\left(r_{i,y}-R_{j,y}\right)^2+\left(r_{i,z}-R_{j,z}\right)^2}}. \tag{1}$$

Here $\vec{R}_j$ are the locations in space where the electric potential Φ is computed, and $\vec{r}_i$ and q_i are the locations and charges of the particles producing the electric potential. k is Coulomb's constant approximately equal to $8.99 \times 10^9\ \mathrm{N\,m^2\,C^{-2}}$.

In our application, we represent each individual charged particle as a structure `Charge` that contains its location in three dimensions as well as its charge. We then create an array of these `Charge` structures.

```
struct Charge {
    float x, y, z, q; // Coordinates and value of this charge
};
// ... //
const int m = 128; // number of charges
Charge chg[m];
```

Listing 1: Initial data structure.

Then, for each given observation location $\{R_x, R_y, R_z\}$, we compute the electric potential using the function in Listing 2.

```
void CalculateElectricPotential(
        const int m,       // Number of charges
        const Charge* chg, // Charge distribution (array of structures)
        const float Rx, const float Ry, const float Rz, // Observation point
        float & phi  // Output: electric potential
        ) {
    phi=0.0f;
    for (int i=0; i<m; i++)  { // Coulomb's law
        const float dx=chg[i].x - Rx;
        const float dy=chg[i].y - Ry;
        const float dz=chg[i].z - Rz;
        phi -= 8.98755178737e9 * chg[i].q / sqrt(dx*dx+dy*dy+dz*dz);
    }
}
```

Listing 2: Initial implementation.

Our goal is to optimize this particular function, as well as the data structure itself, to take better advantage of vectorization. Note that, as mentioned earlier in Section 2, this application is already multi-threaded. Threading is done by distributing the calculations for different observation locations across OpenMP threads.

2.2. System Specification and Compilation

The initial implementation of our application can be downloaded from [1]. Note that you need the Intel C++ compiler and Intel MKL to compile our application. The performance-critical function, `CalculateElectricPotential()`, is implemented in the file `worker.cc`, which was compiled with the following flags.

```
vega@lyra% icpc -g -xCORE-AVX2 -O3 -c -o worker.o worker.cc
```

We are using Parallel Studio 16 update 2, which corresponds to Intel Advisor XE 2016 update 3. All the

benchmarks were taken on a two-way system with an Intel Xeon E5-2697 v3 processor (14 cores/package or 28 cores total, clocked at 2.60 GHz, with Intel hyper-threading technology enabled).

Our application has built-in performance benchmark, which reports attained performance in billion floating point operations per second (GFLOP/s). Note that this value is approximate, and was found by simply counting the number of floating operations inside the for-loop in `CalculateElectricPotential()`. We approximated the count to 10 because some operations (like sqrt) are heavier than simple multiplication.

The entire workload is repeated 10 times, and the reported performance is the average of all trials *except the first two*. This is to remove any effect of initialization so that we can get the sustained performance.

2.3. Some Considerations Before Starting

Before starting an optimization project, it is generally recommended to consider what workflow you follow: blindly going about the process of optimization can lead to headaches down the line. When first using the Intel Advisor it is recommended to follow the *Advisor workflow* shown in Figure 1.

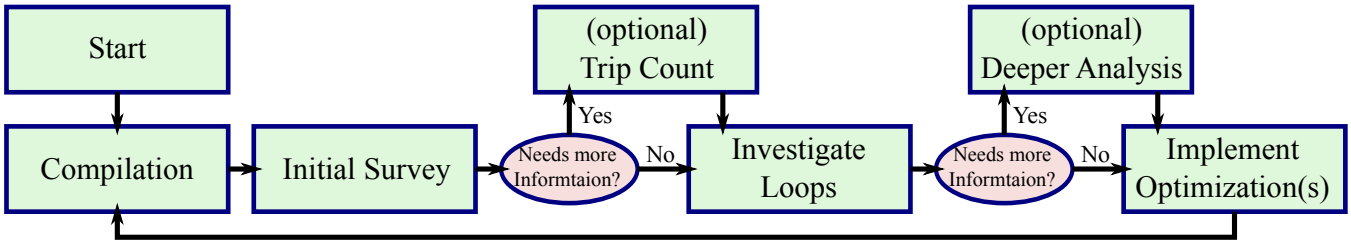

Figure 1: Advisor workflow block diagram.

The Advisor workflow is an iterative process, and generally it is a good idea to keep to one optimization per cycle especially if you are new to code modernization. For more detailed description of each step, see Appendix B.

Another important consideration is the problem size. During the process of optimization, you will have to run the application multiple times. If the application with the default problem size takes, for example, 2 days to complete, this can slow the process down tremendously. Consider using a representative workload, which is a subset of the full problem that is large enough to simulate the runtime performance of the full project, but small enough to complete in a reasonable amount of time. The exact size to use depends on the problem, but aim for execution time of at least few tens of seconds (to collect sufficient statistics).

We can now set up the project (see Appendix A.2) and run our first Survey Analysis. As the application is running, check the terminal that started the GUI. The output of the application can be found on this terminal. Our application has a built-in performance report, which tells us that we are getting 30.9 ± 0.0 GFLOP/s as our baseline performance.

2.4. Vector Dependence

After the application completes its run, Advisor compiles the collected data into a human-readable format. The Summary page shows a general overview of the results. The Summary page for our application is shown in Figure 2.

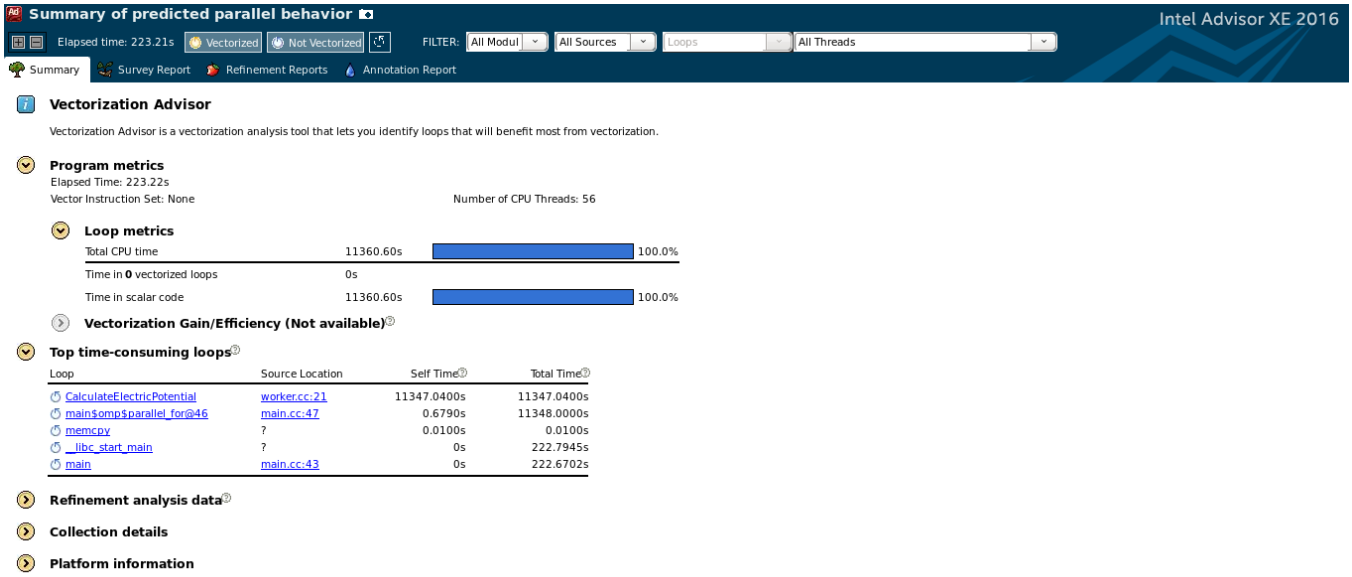

Figure 2: Advisor Summary page.

We start in a rather poor state, as `Loop metrics` section reports that our application spent all of its execution time in scalar code. Generally, properly optimized code should spend the majority of the execution time in vectorized loops.

Under `Top time-consuming loops` we immediately see that the loop in `CalculateElectricPotential()` function is the main factor in performance optimization, as evident from the massive `Self Time` value (11347.04s) compared to all the other loops (all less than 1s). This result suggests that we focus our optimization effort on `CalculateElectricPotential()` function.

To get a more detailed information about the `CalculateElectricPotential()` function, we go to Survey Report. Our initial Survey Analysis result is shown in Figure 3.

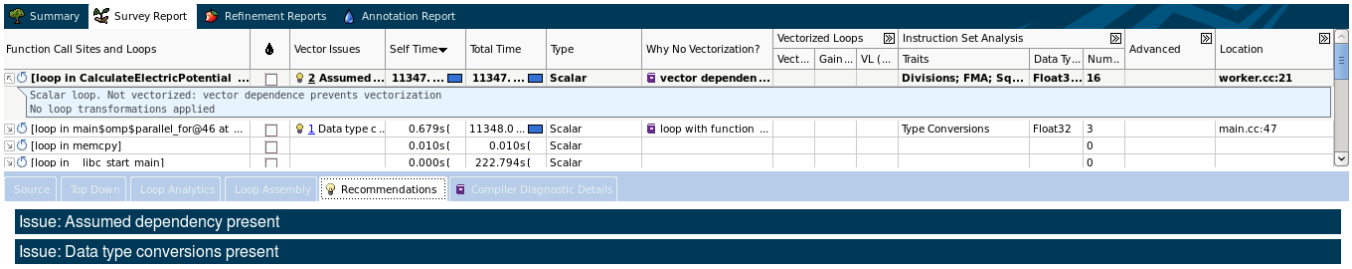

Figure 3: Survey report from our first Survey Analysis

On the row corresponding to `CalculateElectricPotential()`, the `Type` column tells us that this particular loop is not vectorized, and the Advisor has found 2 issues associated with this function which are reported in `Vector Issues` column.

- Issue: Assumed vector dependency present
- Issue: Data type conversion present

Which issue to address first? One strategy for determining the order of issues to address is to look at the `Why Not Vectorization` column to see why the loop is not vectorized in the first place. In our case, Advisor reports that the Vector Dependence issue is the reason behind why this loop is non-vectorized. So we will first focus on this issue. Figure 4 shows the recommendation from the Advisor.

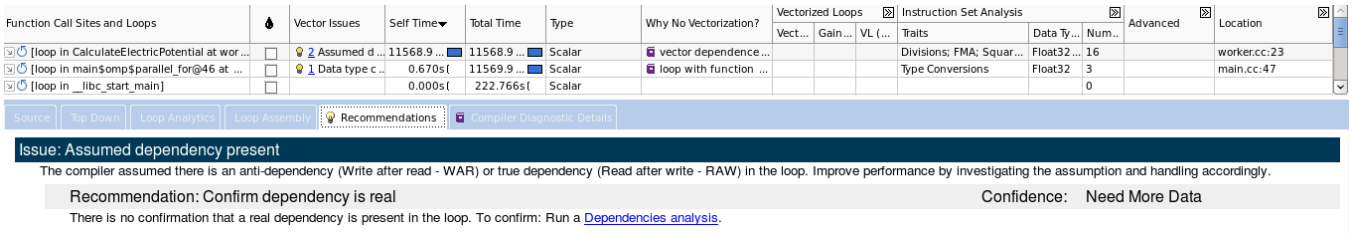

Figure 4: Recommendation on Vector Dependence.

Assumed vector dependence is a condition in which vectorization is potentially unsafe. For more on vector dependence, refer to episode 4.3 from our Colfax Video Course [6] (requires free registration on the website). We can have the Advisor confirm an assumed vector dependence by running a Dependency Analysis. Dependency Analysis is one of the Deeper Analysis options that investigate the read and write patterns inside selected loops to check if there are any unsafe access patterns. See Section B.5 for instructions on running Dependency Analysis.

The output of the Dependency Analysis is shown in Figure 5.

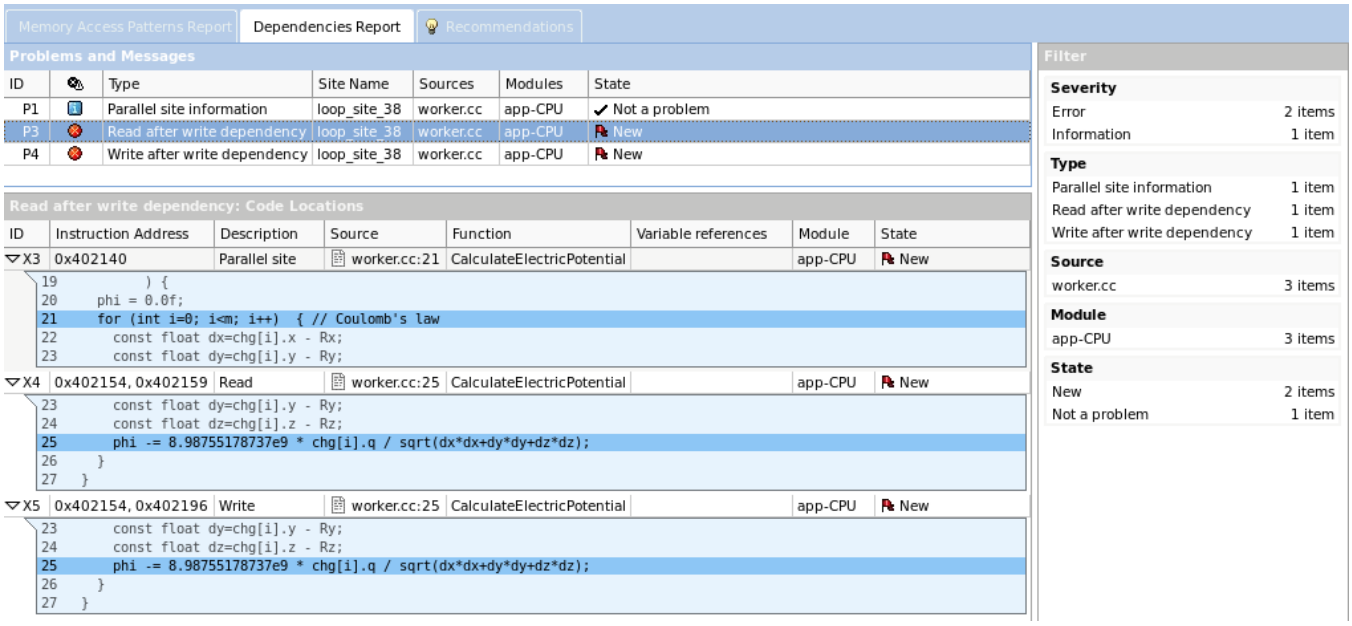

Figure 5: Result of the Dependency Analysis.

From the results, we can see that the culprit is the variable `phi`, and the Advisor reports that there is in fact a dependence here. In order to understand what is going on in this particular line, consider how the right-hand-side (RHS) modifies the left-hand-side (LHS). The RHS computation is calculating the

contribution of the i-th particle to the electric potential, which we will call `delta-phi`. When vectorized, the RHS will, in the case of our Haswell system, compute `delta-phi` for i-th to (i+7)-th particles at once using SIMD instructions. However, the problem occurs when the i-th `delta-phi` contribution must be added to the scalar `phi`, because (i+1)-th to (i+7)-th `delta-phi` contributions must also be added to `phi` simultaneously.

In order to resolve this issue, we can use a technique called reduction. Instead of having a scalar `phi` variable, we can create a `temp_phis` array which has a size equal to the vector width (in our case, 8). Now we can safely add the 8 different contributions from a single vector register at the same time to this `temp_phis` array. After all the iterations are done, we can sum up, or reduce, the results in `temp_phis` array into a single scalar result `phi`.

Implementing the procedure described above requires a moderate amount of code modification, as we would need to restructure the for-loops to incorporate the `temp_phis` (loop strip-mining) and deal with remainder when the loop count `n` is not a multiple of 8. However, there is a more convenient method to implementing reduction, which is to use the SIMD reduction feature of the OpenMP 4 standard. This feature is invoked by placing a directive with the following syntax before the vectorized loop:

```
#pragma omp simd reduction {reduction-identifier: list}
```

Here the `reduction-identifier` is the reduction operator that you want, and `list` is the list of variables that contains the result. In our particular case, the reduction operator that we need is the "minus" operator (i.e., `reduction-identified` must be set to "−") and the variable that needs to be reduced is `phi`. However, depending on the Intel OpenMP implementation, the reduction clause does not allow you to use variables passed in by reference. If this is the case, as it was in our situation, then an additional temporary variable needs to be created to use for reduction. Listing 3 shows our implementation. Note that only one line changed from our initial implementation in Listing2

```
void CalculateElectricPotential( ...
        float & phi  // Output: electric potential
        ) {
    float phi_p=0.0f;  // Creating a temporary variable
#pragma omp simd reduction(-: phi_p)
    for (int i=0; i<m; i++)  { // Coulomb's law
        const float dx=chg[i].x - Rx;
        const float dy=chg[i].y - Ry;
        const float dz=chg[i].z - Rz;
        phi_p -= 8.98755178737e9 * chg[i].q / sqrt(dx*dx+dy*dy+dz*dz);
    }
  phi = phi_p;
}
```

Listing 3: Reduction implementation.

We will now recompile our application. Note that adding OpenMP SIMD pragma requires you to add the OpenMP flag (`-qopenmp` for Intel Compiler and `-fopenmp` for GCC), but in our case we already

had this flag set. Upon doing another Survey Analysis, we see that the loop is now vectorized (see `Type` column). Figure 6 shows the survey result.

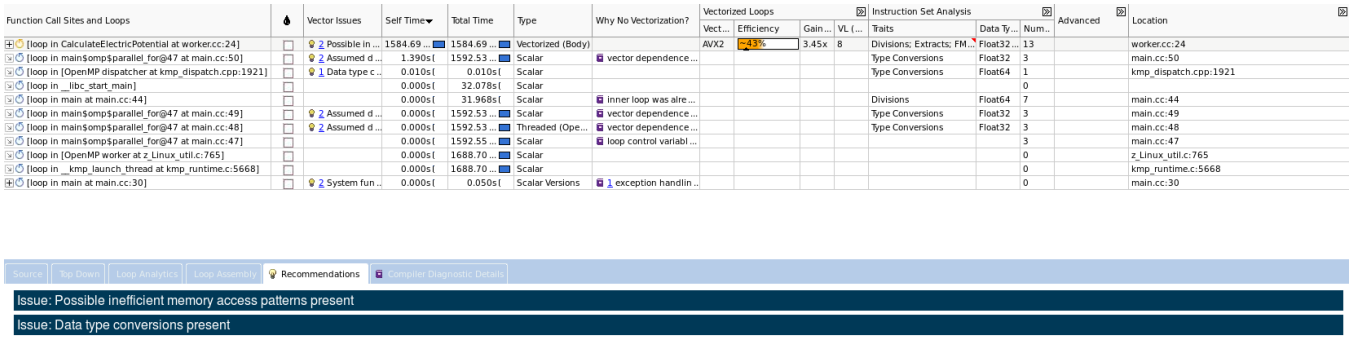

Figure 6: Loop is now vectorized.

Our application now runs at 55.6 ± 0.0 GFLOP/s, which is an ≈85% performance boost compared to our initial implementation. Even though this is a great speedup, SIMD instructions on Haswell architecture processors are theoretically capable of an 8x performance boost in single precision (SP). One way to assess how well we are doing is to look at the Vector Efficiency.

Vector Efficiency is the Advisor's estimate for the performance benefit from vectorization (compared to a scalar instruction).The statistic is often reported with the Gain, which is the vector efficiency times the maximum gain (8 for our Haswell system). Low efficiency is an indication that your application may have opportunities for more performance. There is no "rule-of-thumb" value that an application must achieve to be considered "good enough", but if this value is less than 60% or 70% you may want to take a closer look. Advisor reports that for our application the efficiency is at ≈43%, with an estimated vectorization gain of 3.45x (see Figure 6), so we still have a lot of performance left on the table.

We see that the Advisor found another issue, so we still have two issues. Next, let us tackle the other issue that we ignored on our very first survey: the type conversion issue.

2.5. Consistency of Precision

The source of type conversion issue is simple: type conversions are expensive. If there are type conversions occurring inside a performance critical region of your code (i.e., inside the innermost for-loop) the latency of the type conversion can be responsible for a large chunk of the total computation.

To find out where this problem is occurring, we go to the page titled "source" on bottom panel. See Figure 7.

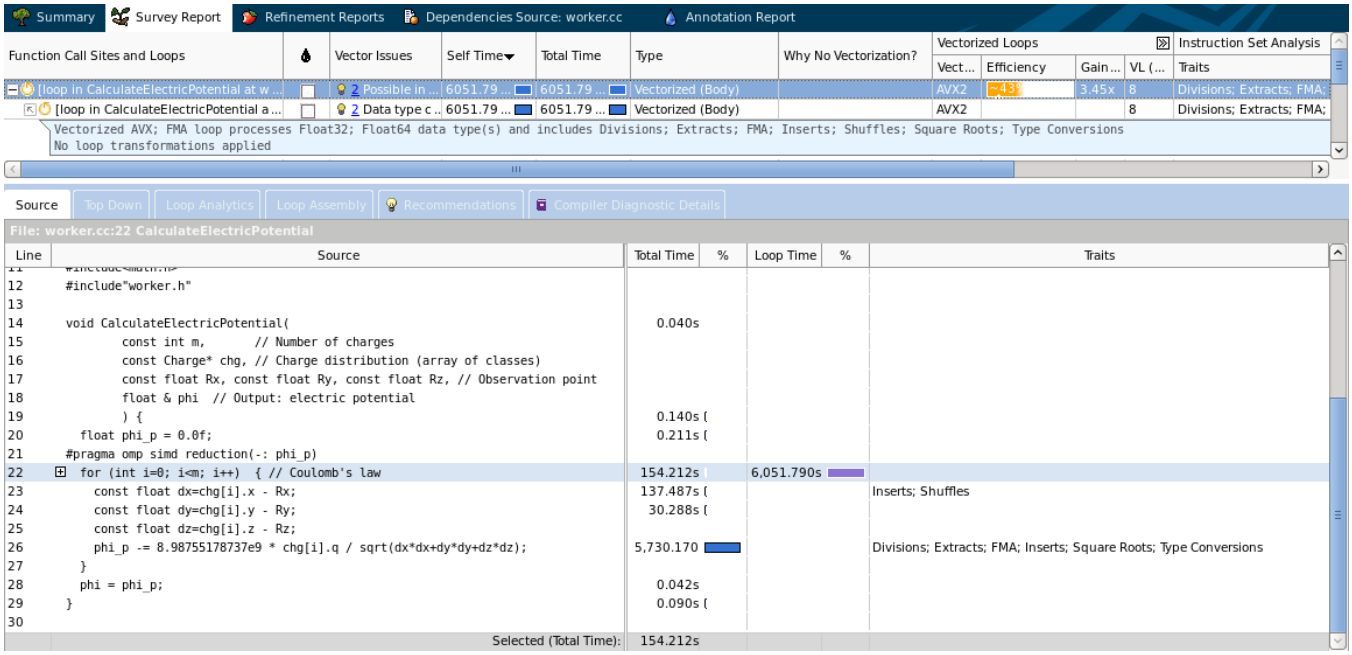

Figure 7: `Traits` column showing type conversion at line 26.

From the `Traits` column on the right, we see that line 26 has type conversions listed. The line in question mainly uses single precision floating-point numbers, so this means that there are other data types (e.g. double precision floating-point numbers or integers) mixed in. Developers familiar with assembly may be able to gather additional information from the Source Assembly tab of the same panel. But for our discussion, we will simply go back to the source code and inspect every element, both functions and constants/variables, on the RHS of line 26.

It is relatively simple to verify that the variables (`Rx`, `Ry`, `Rz`, `chg.q[i]`) are all single precision. Furthermore, basic floating point operators (`+`, `-`, `*`, and `/` in our code) are all single precision provided that the operands are single precision. This leaves the literal constant for Coulomb's constant and the `sqrt()` function. It turns out, these are both double precision.

First, let's discuss the literal constant. In C/C++, literal floating-point constants are treated as double precision floating-point numbers by default. In order to specify a single precision floating-point constant, you must add the character '`f`' to the end of the constant.

```
1 8.98755178737e9  // default interpretation: double precision constant
2 8.98755178737e9f // single precision representation
```

This may come as a surprise to some, but type conversion for written constants happen **at run-time** and **NOT at compile-time**. This is true for integers as well (e.g., literal integer constant "2" inside a floating-point computation is converted at runtime). In general, you should always take care to keep data consistent across your function, including constants.

Now let's discuss the `sqrt` function. Transcendental functions used by default (not in any namespace) are interpreted by Intel C++ compilers and GNU C++ compilers as **NOT** overloaded. The `sqrt` function

will always use double precision computation regardless of the precision of the input. To use the single precision implementation, use the `sqrtf` function instead. This is another fact that may be surprising to some, because the C++ standard library does transcendental functions that are overloaded, but only in namespace `std`. This is bad not only because of type conversion, but also because double precision transcendental functions are much more expensive than single precision versions.

Listing 4 shows our new implementation. Note that only change we made are two added characters (both 'f') to line 26.

```
1  void CalculateElectricPotential( ...
2          float & phi  // Output: electric potential
3          ) {
4      float phi_p=0.0f;  // Creating a temporary variable
5  #pragma omp simd reduction(-: phi_p)
6      for (int i=0; i<m; i++)  { // Coulomb's law
7          const float dx=chg[i].x - Rx;
8          const float dy=chg[i].y - Ry;
9          const float dz=chg[i].z - Rz;
10         phi_p -= 8.98755178737e9f * chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz);
11     }
12   phi = phi_p;
13 }
```

Listing 4: Consistency of precision.

Upon compiling and running the survey, we find that the type conversion warning is gone from the report and that our application now gets 326.1 ± 0.8 GFLOP/s: a 6x performance boost compared to our previous implementation.

We have one issue remaining for this loop,

- Issue: Possible inefficient memory access pattern present

We will now tackle this problem.

2.6. Inefficient Memory Access Pattern

The recommendation page for the issue, "Inefficient Memory Access Pattern", is shown in Figure 8.

Figure 8: Inefficient Memory Access Pattern.

It recommends us to try the Memory Access Pattern analysis, which is another Deeper Analysis supported by the Advisor. For more instructions of how to run a Deeper Analysis, refer to Section B.5.

We first ran the Memory Access Pattern analysis without any changes. The result is shown in Figure 9 and Figure 10.

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Site Name
[loop in CalculateElectricPotential at worker.cc: ...	No information available	100% / 0% / 0%	All unit strides	loop_site_39

Memory Access Patterns Report | Dependencies Report | Recommendations

ID	Stride	Type	Source	Site Name	Nested Function	Modules	Variable references
P1	1	Unit stride	worker.cc:23	loop_site_39		app-CPU	

```
21  #pragma omp simd reduction(-:phi_p)
22      for (int i=0; i<m; i++)  { // Coulomb's law
23          const float dx=chg[i].x - Rx;
24          const float dy=chg[i].y - Ry;
25          const float dz=chg[i].z - Rz;
```

P2		Parallel site information	worker.cc:24	loop_site_39		app-CPU	

```
22      for (int i=0; i<m; i++)  { // Coulomb's law
23          const float dx=chg[i].x - Rx;
24          const float dy=chg[i].y - Ry;
25          const float dz=chg[i].z - Rz;
26          phi_p -= 8.98755178737e9f * chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz);
```

P4	0	Uniform stride	worker.cc:26	loop_site_39		app-CPU	

```
24          const float dy=chg[i].y - Ry;
25          const float dz=chg[i].z - Rz;
26          phi_p -= 8.98755178737e9f * chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz);
27      }
28      phi = phi_p;
```

Figure 9: Output of Memory Access Pattern analysis.

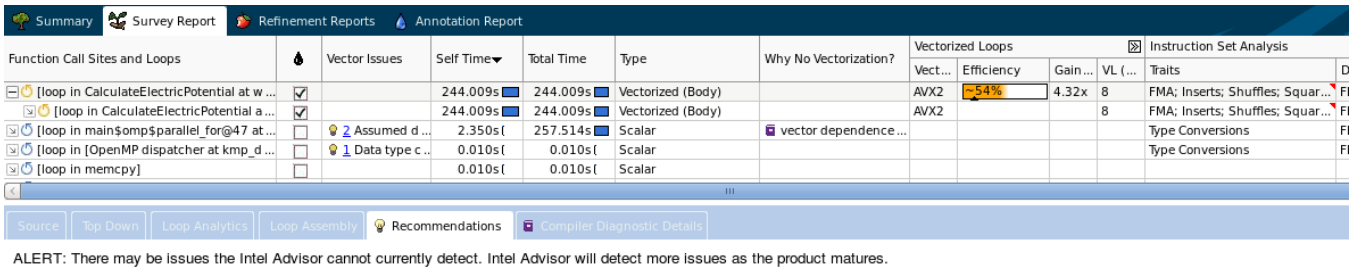

Figure 10: Output of the Survey Report after Memory Access Pattern analysis.

The analysis reports no memory access issues at this stage. However, in some cases further exploration is recommended when there is a warning that tested negative. This is because of compiler optimization. In some cases, the compiler is able to "mask" the symptoms of problems by applying aggressive optimization. Because of this, the Advisor may not directly report the problem because the compiler was able to recover most of the performance loss caused by the issue. But the evidence of the underlying problem is usually still reported by the Advisor: one indication is in the Loop Statistics page (see Figure 11)

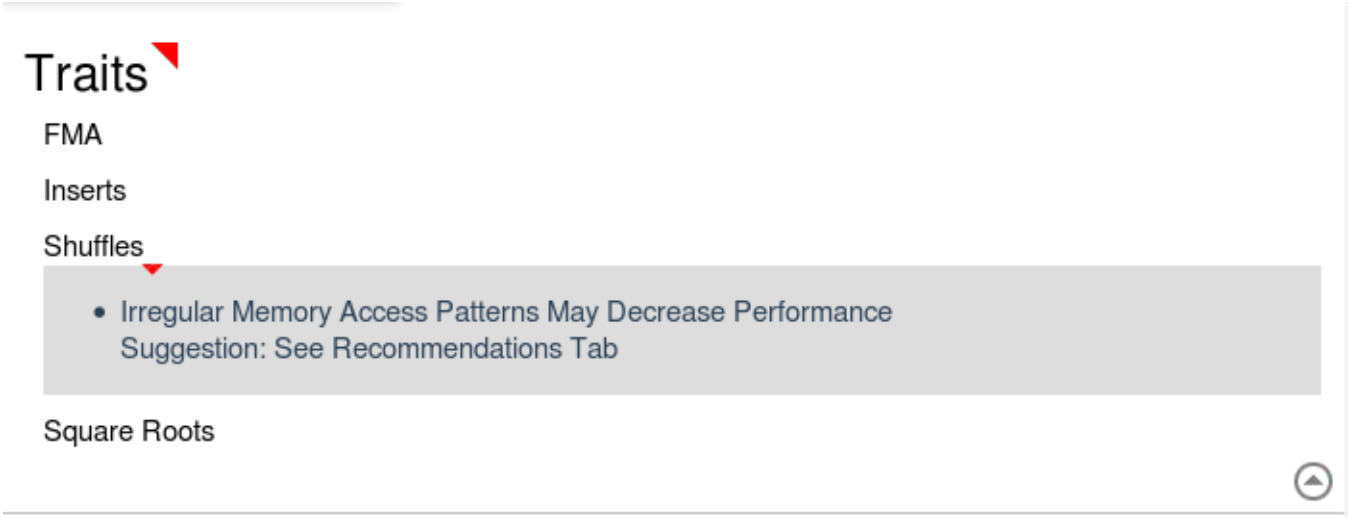

Figure 11: Loop statistics and Traits page.

Under `Shuffles`, we see a notice that this may be caused by irregular memory access patterns. In order to clearly see the problem we can compile the application without vectorization by using `#pragma novector`. This pragma will disable vectorization for the loop that follows. Listing 5 shows `#pragma novector` added to the application.

```
void CalculateElectricPotential( ... ) {
  float phi_p=0.0f;  // Creating a temporary variable
//#pragma omp simd reduction(-: phi_p)  // commenting out the simd clause
#pragma novector  // disable vectorization
  for (int i=0; i<m; i++)  { // Coulomb's law
    const float dx=chg[i].x - Rx;
    const float dy=chg[i].y - Ry;
    const float dz=chg[i].z - Rz;
    phi_p -= 8.98755178737e9 * chg[i].q / sqrt(dx*dx+dy*dy+dz*dz);
  }
  phi = phi_p;
}
```

Listing 5: No vector implementation.

After recompilation, we run the Memory Access Patterns analysis again. Now we see a different output.

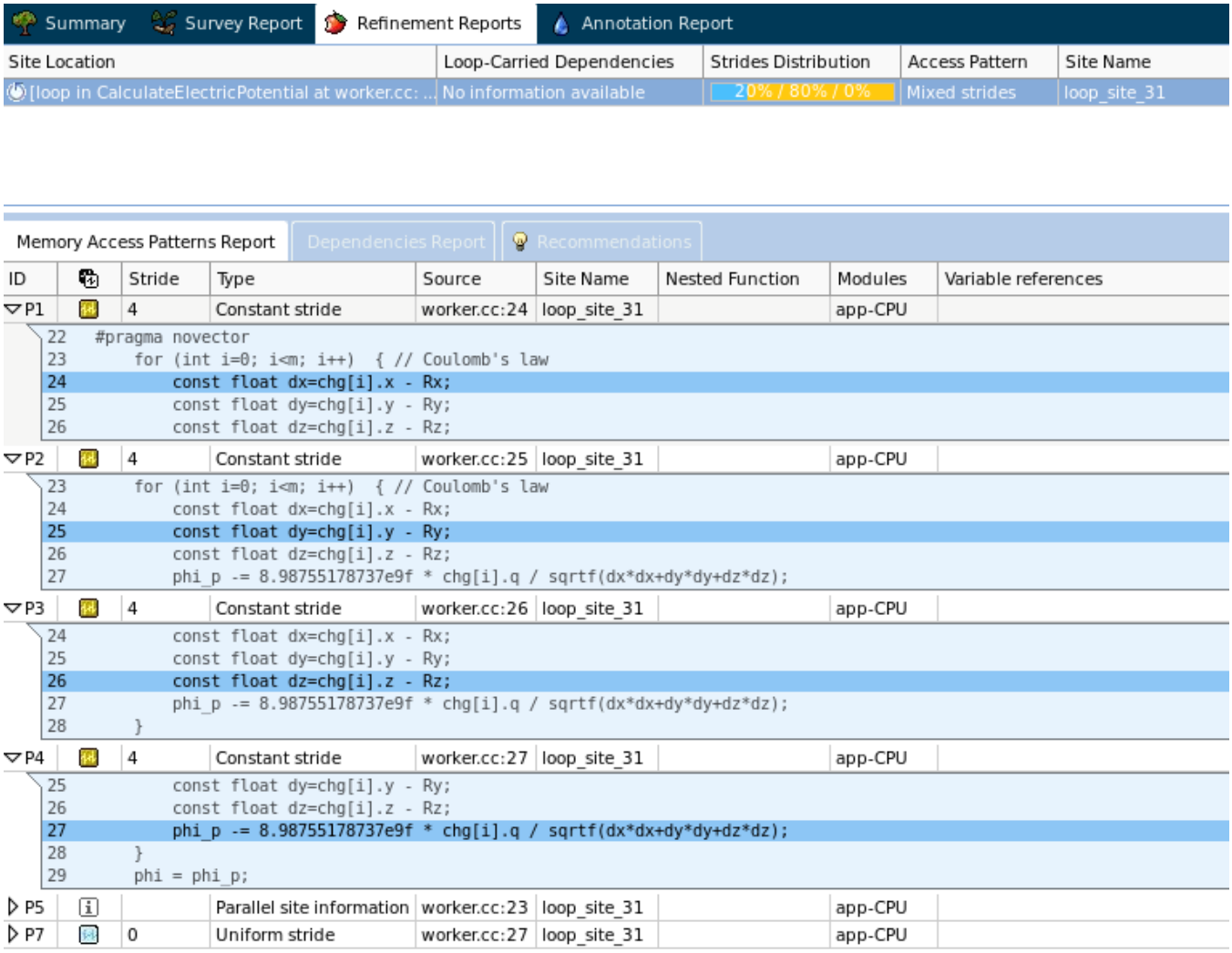

Figure 12: Output of Memory Access Pattern analysis. This time with `#pragma novector`.

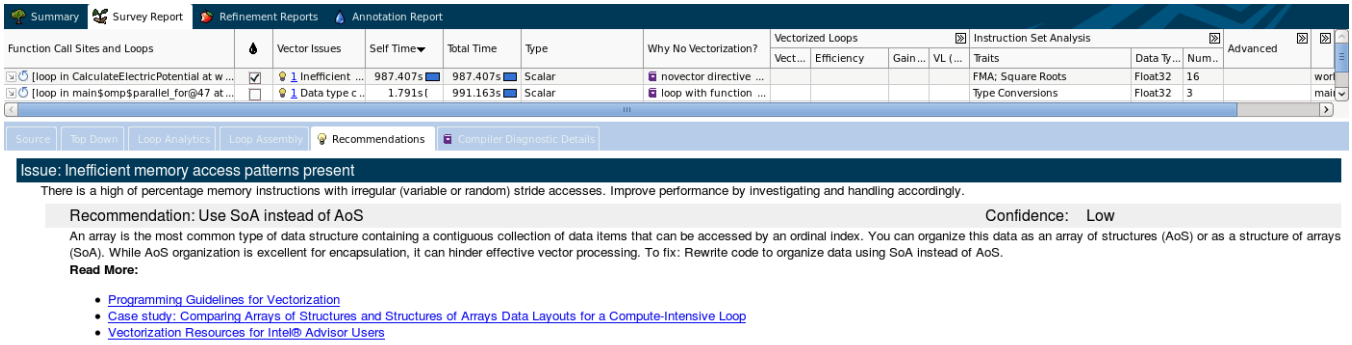

Figure 13: Output of the Survey Report. There is the recommendation for using SoA instead of AoS.

The Survey Report now recommends using Structure of Arrays (SoA) instead of Array of Structures

(AoS). This is an optimization that changes the data structure in order to improve the access patterns of vector instructions. In general, it is best to have unit-stride access to memory when you are vectorizing. However, in our example, because we have an Array of Structures, the adjacent values of `x`, `y`, `z` and `q` are all at a stride of four. In this publication we will not go into details about the theory behind this optimization, but for more on the AoS to SoA issue refer to episode 5.3 from our Colfax Video Course [6].

We will convert our array of `Charge` structures into `Charge_distribution` structure that contains separate arrays for `x`, `y`, `z` and `q`. Listing 6 shows the implementation of the new data structure.

```
struct Charge_distribution {
    float *x, *y, *z, *q; // Individual arrays for coordinates and charge
};
```

```
void CalculateElectricPotential( ...
        const Charge_distribution chg, // Structure of Arrays
        ) {
  float phi_p=0.0f;  // Creating a temporary variable
#pragma omp simd reduction(-: phi_p)
  for (int i=0; i<m; i++)  { // Coulomb's law
    const float dx=chg.x[i] - Rx;
    const float dy=chg.y[i] - Ry;
    const float dz=chg.z[i] - Rz;
    phi_p -= 8.98755178737e9f * chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
  }
  phi = phi_p;
}
```

Listing 6: Structure of Arrays, and the corresponding `CalculateElectricPotential()`.

Now all the variables, `x`, `y`, `z` and `q`, have unit-stride access in memory. Note that this optimization requires more work than just changing the structure and the function. Initialization and all other references to the `Charge_distribution` data structure in your application must be updated.

When we recompile our application and run the Survey Analysis, we see that the application is now running at 503.7 ± 1.4 GFLOP/s, another $\approx$54% improvement in performance. The Survey Report is shown in Figure 14.

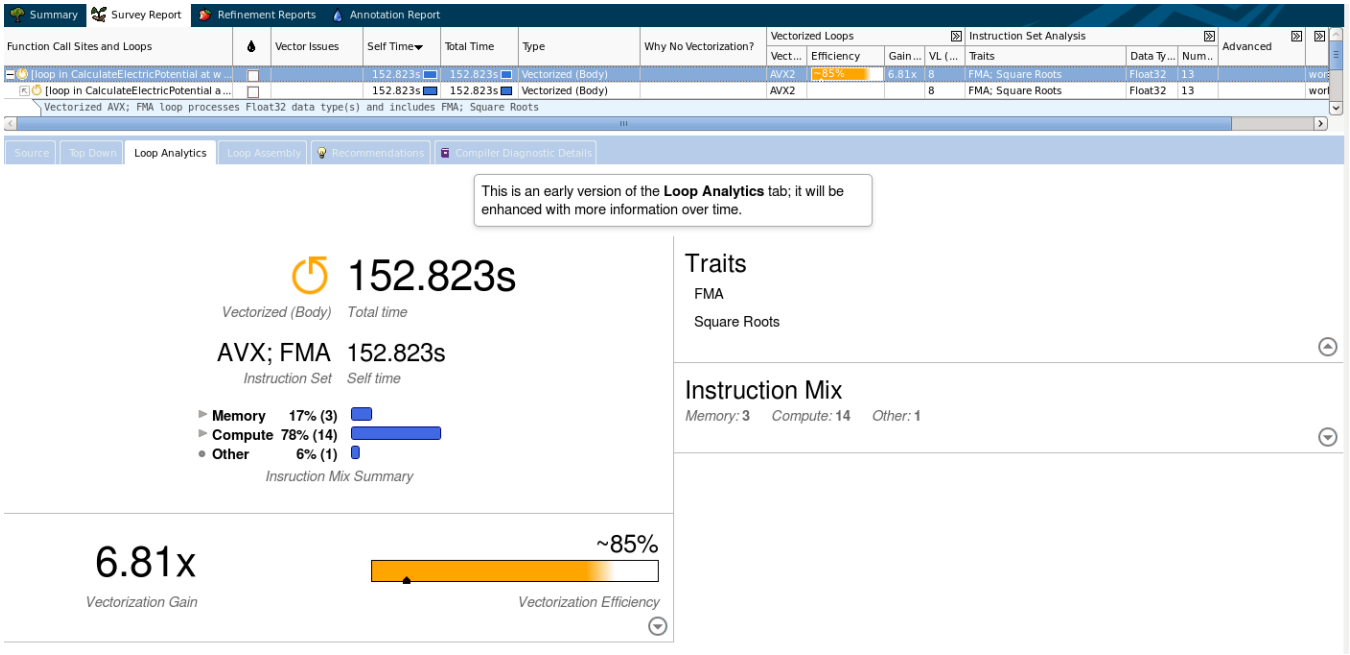

Figure 14: Survey report from our final Survey Analysis

The warning over Shuffles is no longer present, and we see no additional warnings. At this time, with no obvious issues reported by the Advisor, we can do several sanity checks to see how well optimized our vectorization is.

One check is looking at the vector efficiency, which is at ≈85%, with an estimated vectorization gain of 6.81x for our application (see Figure 14). 85% is a great number, so our application passes the first sanity check.

Another check is to measure the vector scaling by comparing performance with and without vectorization. We can insert `#pragma novector` into our application again and benchmark the performance without vectorization. Without vectorization, our application executed at 83.9 ± 0.0 GFLOP/s, which is a factor of 6 smaller than the vectorized performance. This is lower than the maximum theoretical gain, which is 8x for our system. Additional gains may be achieved by improving the memory access pattern with, for example, loop tiling, however, this is beyond the scope of this paper. With a vectorization speedup of 6x, our application passes the second sanity check.

With our application passing both sanity checks, and with Advisor not reporting any issues, we will stop our optimization efforts here. However, note that this is not necessarily the optimal performance attainable for this particular application: as mentioned above, there are opportunities for further optimization.

Figure 15 shows a graph of performance as a function of each optimization measure that we have applied.

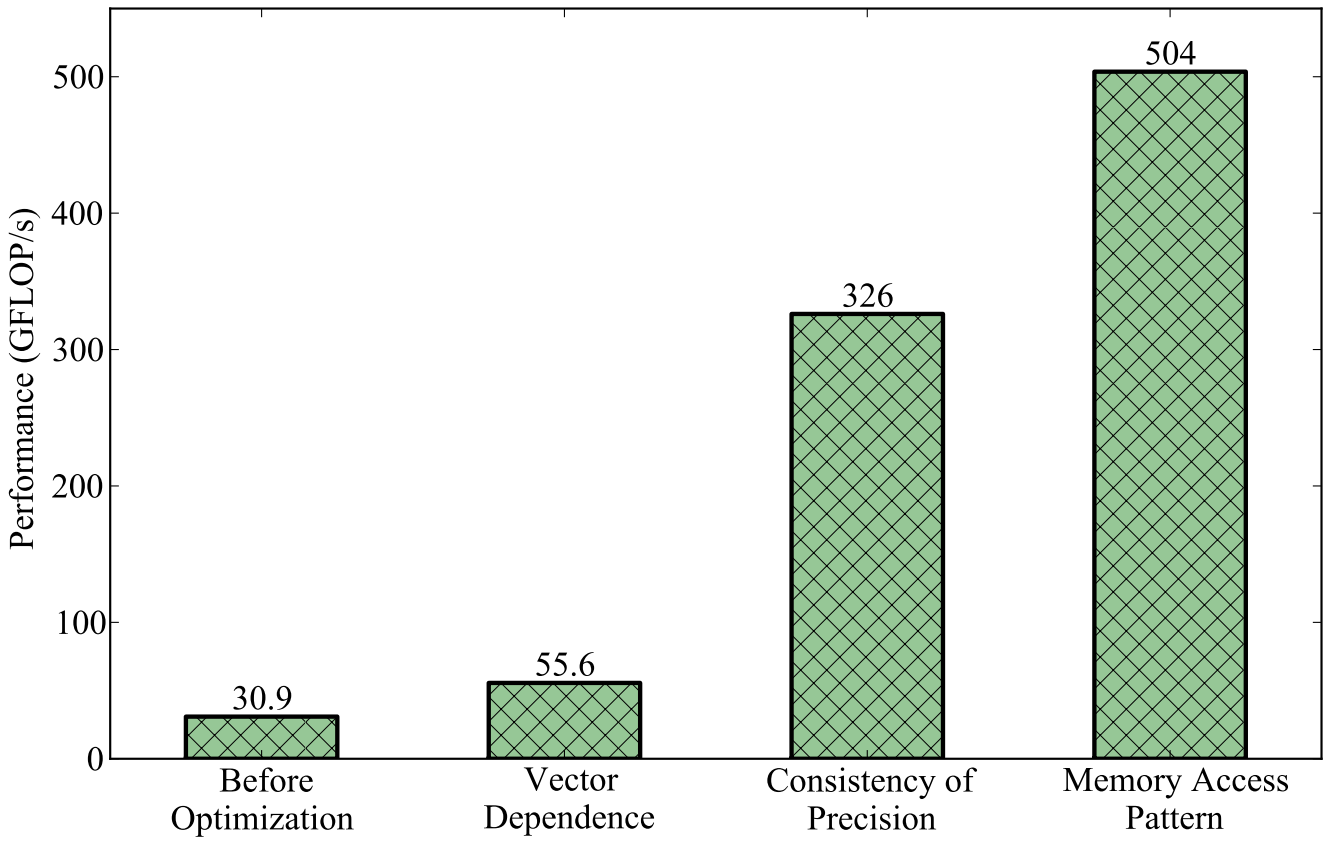

Figure 15: Performances at each step of the optimization process.

In the end, compared to our original, non-optimized application, we observed more than a 16x performance boost from vectorization optimization recommended to us by the Advisor.

3. CLOSING WORDS

In this publication we demonstrated a step-by-step walk-through of using the Intel Advisor to optimize the vectorization aspect of an application.

We started with an example electrostatics problem where we compute the electric potential produced by a group of charged particles at multiple points in a 3-D grid. The application was originally parallelized across cores, but otherwise not optimized and not vectorized. We then used the profiling features of the Advisor to first get the application to vectorize, and then to optimize the vectorization patterns of the calculation. We discussed three primary issues that the Advisor detected: vector dependence, type conversion and inefficient memory access patterns. After optimization, in our benchmarks we observed a 16x performance boost compared to the original, non-optimized implementation.

The example code used in this publication is available for download at the Colfax Research Website [1].

ACKNOWLEDGMENTS

Author thanks Dmitry Dontsov, Egor Kazachkov, Zakhar Matveev and Kirill Rogozhin from Intel Corporation for their helpful comments and reviews in the writing of this paper.

REFERENCES

[1] Ryo Asai. Guided Code Vectorization with Intel Advisor XE, 2016 *(landing page for this paper)*.
http://colfaxresearch.com/using-vectorization-advisor.

[2] Intel Advisor 2016 - Help for Linux* OS.
https://software.intel.com/en-us/intel-advisor-2016-user-guide-linux.

[3] Intel. Intel Modern Code.
https://software.intel.com/en-us/modern-code.

[4] Andrey Vladimirov. Are You Realizing the Payoff of Parallel Processing?
https://communities.intel.com/community/itpeernetwork/datastack/blog/2015/07/08/are-you-realizing-the-payoff-of-parallel-processing.

[5] Kirill Rogozhin. Vectorization Advisor FAQ.
https://software.intel.com/en-us/articles/vectorization-advisor-faq.

[6] Parallel Programming and Optimization with Intel Xeon Phi Coprocessors: Video Course.
http://colfaxresearch.com/cdt-v01/.

[7] Getting Started with Intel Advisor: Command Line and MPI.
https://software.intel.com/en-us/get-started-with-advisor-cli-mpi.

[8] Intel C++ Compiler Reference: -x.
https://software.intel.com/en-us/node/581749#260BF03D-0E6C-462F-A087-98512013B489.

Appendix A. Setting Up Advisor

A.1. Installation, First-time Configuration

Intel Advisor XE is part of a software suite called Intel Parallel Studio XE. Parallel Studio comes in three editions: composer, professional and cluster. The Advisor is *only available in professional and cluster editions.* Parallel Studio does require purchasing a license, however some student/academic and non-commercial discounts are available. For more information visit the Parallel Studio website.

If you have the professional or cluster edition, Intel Advisor will be installed by default when you install the Parallel Studio software suite. Parallel Studio installation is beyond the scope of this document: for information on installation, refer to the documentation included with your Parallel Studio distribution.

After installation, the first thing you need to do is to set the various environment variables for the Advisor, and to add the Advisor commands to your default path. This can be achieved by using the command `source` on the shell script `advixe-vars.sh` located in the installation folder for Advisor. Assuming that Advisor was installed to the default path (`/opt/intel`), type:

```
vega@lyra% source /opt/intel/advisor_xe_2016/advixe-vars.sh
```

You may want to set it up so that this command runs automatically by adding the above line to your `.bashrc` file, or ask your system administrator to add this line to the `/etc/profile.d/` files.

Once the environment variables are set, the Advisor commands will be in your default look up path. To start the Advisor GUI:

```
vega@lyra% advixe-gui  # starts the Intel Advisor GUI
```

In this publication, we primarily focus on the GUI version. However, Advisor has a command line interface (CLI) that allows you to profile code on remote systems that do not have monitors. If you have access to the GUI on some workstation, the safest (from typos or missed flags) way to get a CLI equivalent of some operation is to click the "Command Line" buttons on the left Vectorization Workflow panel. Figure 16 demonstrates this for the Survey Analysis.

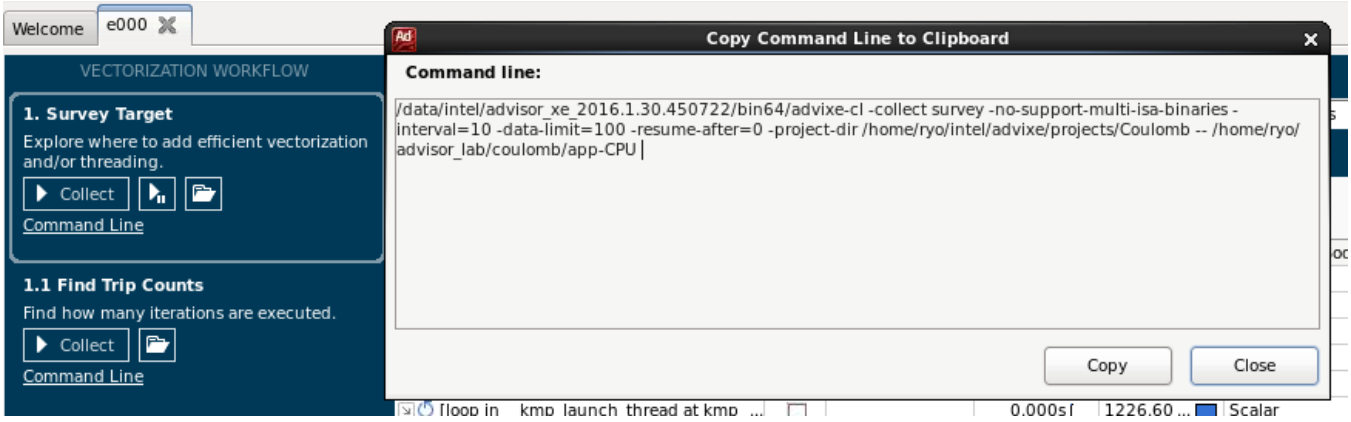

Figure 16: CLI equivalent of Survey Analysis.

For more information on Advisor CLI, refer to the Intel Advisor Reference [7].

A.2. Creating a Project

Once you have started the Advisor GUI, the first thing you need to do is to create a project. If you are familiar with Intel VTune Amplifier XE, this procedure may be familiar. Click on the create project button in the middle, or under the tools bar. This will prompt you to name a project, and then takes you to the project properties tab like the one shown in Figure 17.

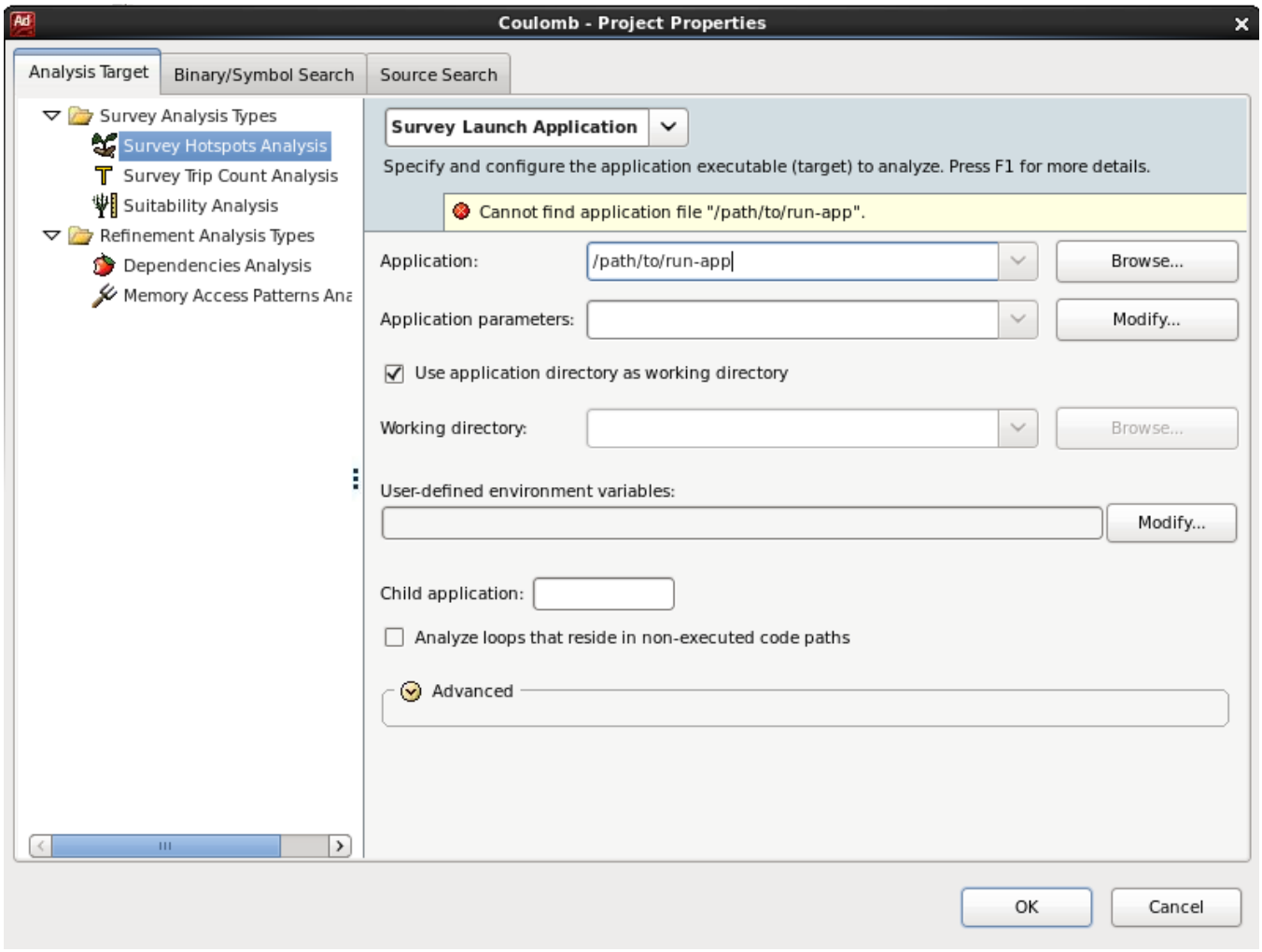

Figure 17: Project properties page.

In Advisor, a project is typically tied to a single executable path (you can modify the path, but this can get cumbersome). Thus, as you make improvements to the code, make sure you place the compiled binary to the same location (i.e. overwrite the previous one). In the `applications:` box, replace `/path/to/run-app` with the path to the executable that you want to profile.

You are now ready to start optimizing your application.

Appendix B. Advisor Workflow

The Advisor workflow is a step-by-step guideline for those who may be unfamiliar with the process of code modernization. Figure 1 from Section 2.3 shows the Advisor workflow for vectorization. Generally, it is recommended to follow the workflow in optimization.

The rest of the section will go into further detail about each step.

B.1. Compilation/Recompilation

The first order of business is to compile your application using the Intel Compiler. In order to get the most out of the Advisor, it is recommended to compile with the following flags.

```
vega@lyra% icpc -g -O3 -xhost foo.cc
```

- `-g` This flag adds debugging information to the application, which Advisor can use to give a more detailed analysis, specifically, to display the names of your functions and variables.
- `-O3` This flag enables compiler optimizations. Although compiler optimization is usually enabled by default for Intel Compilers, `-g` flag disables some compiler optimizations. Thus you will need to manually enable it again. Also note that some analysis types prefer other optimization levels.
- `-xhost` With this flag compiler will attempt to use the most up to date vector instruction set available in the system on which you are compiling. However, if you are cross-compiling and want to specify the exact instruction set supported by the target system, replace this flag with the appropriate `-x{code}` flag (see [8]) to guarantee that you are using the correct instruction set.

The above compilation will create an executable `a.out` by default. It is recommended to also set the `-o [app-name]` flag in order to set the name of the executable. In all subsequent discussions, we will assume the executable name of *run-app*

Once the application is compiled, move on to the Survey Analysis. **NOTE:** if this was the first time on this step, make sure you set the project up and target the correct executable (see Section A.2).

B.2. Survey Analysis

This is the first analysis that needs to be run for Advisor. Click on the `Start Survey Analysis` Button at the tool bar on the top (see Figure 18) to start the analysis. You can also use the left panel shown in Figure 19.

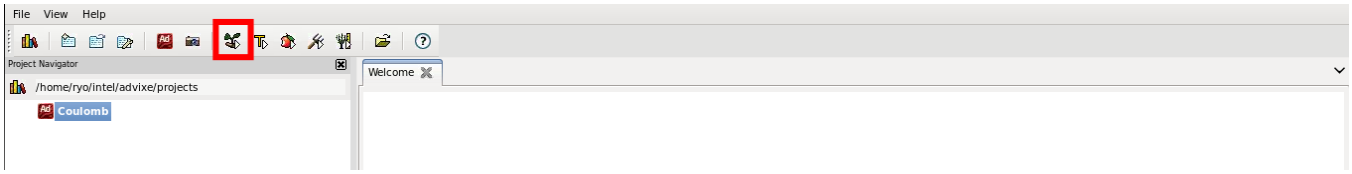

Figure 18: Tool bar. *Red Box:* The survey Analysis button.

It will run your application, collect hardware counter information and give an overview of statistics on various vectorized and non-vectorized loops in your application. Based on its findings, it will also provide recommendations, and give warning messages where some issue in the code is preventing vectorization.

Once the survey completes, either move on to Investigating Loops or Trip Count Analysis. If this is your first run, it is recommended to skip straight to Investigating Loops.

B.3. (Optional) Trip Count Analysis

The Trip Count Analysis collects information about the number of iterations in each loop, as well as the number of times a specific loop was called. This information can be helpful for the developer to find problem regions or to spot potential issues, and the Advisor can use this information to potentially make additional recommendations.

B.4. Investigating Loops

Once you have run the Survey Analysis (and optionally Trip Count Analysis), the survey report page becomes available. This page displays all the findings that the Advisor made in a human-friendly format. There is a lot of information available to you from this page, but for this publication we focus on the following five columns.

- `Vector Issues` Displays issues preventing vectorization and makes recommendations. If this section is not empty, the Advisor has found issues that can possibly be resolved by code modification.
- `Self Time` The time spent on the particular loop. Does *not* include the time spent in a loop nested inside. Generally the larger the Self Time of the loop, the more important the particular loop is to optimizing the application.
- `Type` The state of the loop, some examples of states are scalar, vectorized (Body) and threaded (OpenMP). You would want the performance-critical the loops to be vectorized and/or threaded.
- `Why Not Vectorization` If the loop **is not** vectorized, it displays the primary reason why the particular loop is not vectorized. Note that a non-vectorized loop may not necessarily be a problem. Cross-reference with Vector Issues column for more.
- `Vectorized Loop` If the loop **is** vectorized this column will contain details of the vectorized loop including vectorization efficiency. *Vectorized code is not necessarily optimized code*. If your vectorization efficiency is low, there may be more performance to be gained.

The amount of information can be overwhelming at first, especially for large applications. For less experienced users, one strategy is to sort the loops by `Self Time` (this should be done by default), and go down the `Vector Issues` column and address each problem in order. However, users with more optimization experience may want prioritize optimizations that may affect other loops.

At this point, the developer has a decision to make on where to go next. If there are issues reported by the Advisor with recommendations given, move to Source Code Modification and address this issue/recommendation. Sometimes Advisor may recommend further analysis for some issues, in which case move

to Deeper Analysis. Finally, if all of the important loops (based on `Self Time`) are vectorized without reported issues and have high efficiency, then you may have reached a satisfactory point to call the end of your optimization through Advisor. However, take this with a grain of salt: there still may be possible opportunities for optimization. Advisor is designed for early stages of application development, and may miss some issues associated with late-stage application development. For further optimization, use tools developed for late stage application design, like Intel® VTune Amplifier.

B.5. (Optional) Deeper Analysis

In Deeper Analysis, Advisor collects more detailed information that could not be gathered in Survey Analysis so that the it can give better recommendations. However, Deeper Analysis takes much longer than the Survey Analysis, which makes it impractical to run the analysis on your entire application. Thus, the recommended workflow is to run the Survey Analysis to find problem loops, and then mark these loops for Deeper Analysis.

Perhaps the easiest way of marking the loop is to use the GUI. In the Survey Report, on the second column, there is a box you can check to mark a certain loop for Deeper Analysis (see Figure 19)

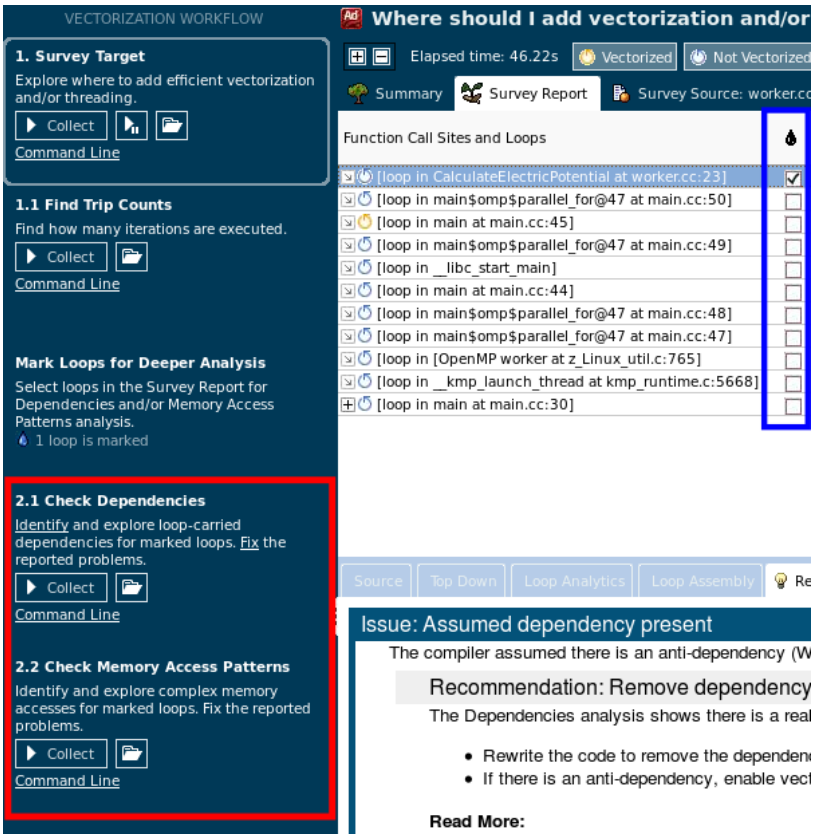

Figure 19: *Blue Box:* Marking loops for further analysis. *Red Box:* Deeper Analysis options.

More fine-grained alternative is to use Advisor annotations, however this is beyond the scope of this publication. Refer to the Advisor reference [2] for more information on annotations. Once the problem loops have been marked, there are two types of analysis that we can apply (as of writing of this paper):

- Dependency Analysis:
  Used for detecting data dependencies such as vector dependence (prevents vectorization) and data races (unsafe thread parallelism).

- Memory Access Pattern Analysis:
  Used for analyzing the access pattern of SIMD operations to find inefficiencies like strided access or unaligned access.

Beginning users should use these optional analyses when instructed to do so by the Advisor. To start the Deeper Analysis, use one of the options listed on the left panel or at the top (see 19) to start the Analysis.

Again note that the Deeper Analysis takes much longer than Survey analysis. You may want to reduce the problem size in order to reduce the analysis time.

B.6. Implementing Optimizations

Using the information obtained from the Survey Report and/or Deeper analysis reports, make modification to the source code to address the problems and recommendations that Advisor gave. Once the source code modification is complete, go back to the Compilation/Recompilation step in Section B.1.